

Encoding Partitions As Ascending Compositions

A thesis submitted to the National University of Ireland
in accordance with the requirements of the degree of
Doctor of Philosophy in the Faculty of Science.

Jerome Kelleher

Department of Computer Science

University College Cork

December 2005

Research Supervisor Dr. Barry O'Sullivan

Head of Department Professor Gregory M. Provan

Abstract

Existing algorithms to generate all partitions of n (unordered collections of positive integers whose sum is n) in lexicographic order encode partitions as *descending* compositions (sequences of positive integers $d_1 \geq \dots \geq d_k$ such that $d_1 + \dots + d_k = n$). We develop algorithms using the alternative encoding, *ascending* compositions (sequences of positive integers $a_1 \leq \dots \leq a_k$ such that $a_1 + \dots + a_k = n$). By encoding partitions as ascending compositions we significantly improve on the efficiency and generality of existing partition generation techniques. The new algorithms are compared with their descending composition counterparts by an extensive theoretical and empirical analysis. In each case the ascending composition encoding leads to a significantly more efficient (yet simpler) algorithm. Using ascending compositions to encode partitions also allows us to define efficient algorithms to generate a wide variety of restricted partition. We develop a theoretical framework that concisely defines important classes of restricted partition (e.g. Rogers-Ramanujan and Göllnitz-Gordon partitions) using an integer function. We develop efficient generation and enumeration algorithms in which the required restriction is specified by supplying the relevant function as a parameter.

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced.

Jerome Kelleher
December 2005

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Barry O'Sullivan. His patience, hard work and attention to detail were of incalculable benefit to this research, especially with respect to the mathematical content. I would also like to thank Enterprise Ireland for generously supporting this research under their Basic Research Grant Scheme (Grant Number SC/02/289). Thanks also to Derek Bridge and Anne Marie Kelleher, whose meticulous reading and well-founded criticism greatly improved this dissertation. Finally, I would like to apologise to my family and friends for the absences enforced by this work, and thank them for their support and understanding.

Contents

1	Introduction	1
1.1	Partitions	2
1.2	Combinatorial Generation	3
1.3	Thesis and Dissertation Overview	6
1.3.1	Thesis	7
1.3.2	Dissertation Overview	9
2	Literature Review	11
2.1	Combinatorial Generation	11
2.1.1	General Background	12
2.1.2	Generation Order	13
2.1.3	Analysis	14
2.1.4	Auxiliary Problems	17
2.2	Compositions and Partitions	18
2.2.1	Compositions	19
2.2.2	Partitions	22
2.2.3	Encoding Partitions	24
2.3	Generating Compositions	26
2.3.1	Representations	26
2.3.2	Generation Order	28
2.3.3	Unrestricted Generators	30
2.3.4	Restricted Generators	36
2.3.5	Miscellaneous Algorithms	40
2.4	Summary	42

CONTENTS

3	Interpart Restricted Compositions	44
3.1	An Algorithmic Framework	44
3.1.1	Definitions and Notational Conventions	45
3.1.2	Fundamental Results	49
3.1.3	Examples	53
3.2	Enumeration	62
3.2.1	A General Recurrence Equation	63
3.2.2	Integer Sequences	66
3.3	Summary	69
4	Generating Interpart Restricted Compositions	70
4.1	Preliminaries	71
4.1.1	Definitions and Notation	71
4.1.2	Nondecreasing Restriction Functions	72
4.2	Recursive Algorithm	75
4.2.1	Basic Principle	75
4.2.2	Algorithm	76
4.2.3	Analysis	79
4.3	Succession Rule	80
4.3.1	Basic Principle	81
4.3.2	Algorithm	83
4.3.3	Analysis	90
4.3.4	Comparison with Existing Algorithms	93
4.4	Accelerated Algorithm	95
4.4.1	Basic Principle	96
4.4.2	Algorithm	101
4.4.3	Analysis	107
4.4.4	Comparison	109
4.5	Summary	116
5	Generating All Partitions	117
5.1	Preliminaries	118
5.2	Recursive Algorithms	121

CONTENTS

5.2.1	Ascending Compositions	121
5.2.2	Descending Compositions	124
5.2.3	Comparison	133
5.3	Succession Rules	135
5.3.1	Ascending Compositions	136
5.3.2	Descending Compositions	146
5.3.3	Comparison	151
5.4	Accelerated Algorithms	156
5.4.1	Ascending Compositions	157
5.4.2	Descending Compositions	171
5.4.3	Comparison	176
5.4.4	Other Algorithms	183
5.5	Summary	185
6	Conclusions and Future Work	187
6.1	Thesis Defence	187
6.2	Future Work	189

List of Figures

1.1	Example generation and consumer procedures. The generation procedure generates all ascending compositions of 4 in lexicographic order, and the consuming procedure uses these to compute the product of all parts in partitions of n	4
2.1	Fenner & Loizou’s binary tree representation of the set of descending compositions of 5, and the orders corresponding to a inorder, preorder and postorder traversal of this tree.	29
3.1	Illustration of recurrence to count interpart restricted compositions for $\sigma(x) = x + 2$	66
4.1	Array state-transition diagram for recursive ascending composition generation algorithm.	78
4.2	Array state-transition diagram for the lexicographic succession rule generation algorithm for interpart restricted compositions.	88
4.3	Composition blocks and terminal compositions.	97
5.1	Array state-transition diagram for the recursive ascending composition generation algorithm.	122
5.2	Array state-transition diagram for Ruskey’s algorithm.	127
5.3	Array state-transition diagram for the succession rule based ascending composition generation algorithm.	140

LIST OF FIGURES

5.4	Illustration of Theorem 5.6 for $n = 5$. The number of partitions of n may be obtained by summing $\lfloor (x + y)/(x + 1) \rfloor$ over all partitions of n , where y is the largest part and x is the second largest part (x can be equal to y).	145
5.5	Array state-transition diagram for the succession rule based descending composition generation algorithm.	148
5.6	Read and write tapes for the direct implementations of succession rules to generate ascending and descending compositions.	155
5.7	Read and write tapes for the accelerated algorithms to generate ascending and descending compositions.	180

List of Tables

2.1	All compositions of 4 with corresponding subsets of $\{1, \dots, n\}$, tilings of a 1-by- n board with 1-by- q tiles, and binary $(n - 1)$ -tuples.	21
2.2	Comparison of algorithms to generate unrestricted, descending and ascending compositions in lexicographic and reverse lexicographic order.	35
3.1	Table of restriction functions and partitions with parts satisfying certain congruence conditions which are enumerable via a partition identity.	68
4.1	A comparison the rule-based algorithm to generate interpart restricted compositions with Nārāyaṇa's and Riha & James' algorithms.	94
4.2	Number terminal and nonterminal interpart restricted compositions of n for several restriction functions.	111
4.3	A comparison of the rule-based and accelerated algorithms to generate interpart restricted compositions.	115
5.1	A comparison of recursive partition generators.	135
5.2	Empirical analysis of accelerated ascending and descending composition generation algorithms.	183
5.3	Empirical comparison of major partition generation algorithms in the various representations.	185

List of Algorithms

3.1	TABULATE $_{\sigma}(N)$	67
4.1	RECGEN $_{\sigma}(n, m, k)$	77
4.2	LEXMIN $_{\sigma}(n, m)$	86
4.3	RULEGEN $_{\sigma}(n, m)$	89
4.4	LEXMIN' $_{\sigma}(n, m)$	102
4.5	ACCELGEN $_{\sigma}(n, m)$	105
5.1	RECLASC (n, m, k)	122
5.2	RECDASC (n, m, k) [Rus01, §4.8]	126
5.3	RULEASC (n)	141
5.4	RULEDESC (n)	147
5.5	ACCELASC (n)	166
5.6	ACCELDESC (n) [ZS98]	173

Chapter 1

Introduction

This dissertation is concerned with the problem of systematically generating all partitions of a positive integer n . We shall argue in favour of a particular method of encoding partitions for this purpose and demonstrate that this encoding leads to algorithms that are significantly more efficient than those that currently exist. We conduct this argument by comparing existing algorithms with commensurable algorithms we develop using the new encoding. The algorithms are compared by performing an indepth theoretical analysis, and using these analyses to make qualitative and quantitative predictions about the relative efficiencies of the algorithms. The accuracy of these predictions (and the validity of the chosen theoretical model) is then assessed by empirical observation. In each case the algorithm using the encoding we propose is significantly more efficient than the existing algorithm. Furthermore, the increase in efficiency is not obtained at the cost of increased complexity of algorithmic expression: in each case the algorithms we propose are simpler and more concise.

In Section 1.1 we discuss partitions and their importance in mathematics and concrete applications. The subject of systematically generating combinatorial objects is an active research area and in Section 1.2 we discuss some applications and provide references to the literature. In Section 1.3 we precisely state the thesis we shall be defending and provide a brief outline of the dissertation.

1.1 Partitions

A partition of a positive integer n is an unordered collection of positive integers whose sum is n . For example, there are five partitions of 4 and these can be represented as $1 + 1 + 1 + 1$, $1 + 1 + 2$, $1 + 3$, $2 + 2$ and 4 . Partitions have been the subject of extensive study for many years and the theory of partitions is a large and diverse body of knowledge. Partitions are a fundamental mathematical concept and have connections with number theory [And76], elliptic modular functions [Sch86, p.224], Schur algebras and representation theory [Mar99, p.13], derivatives [Yan00], symmetric groups [Com55, BMSW54], Gaussian polynomials [AE04, ch.7] and several other mathematical fields [AO01].

The theory of partitions has many applications. For example, partitions can be used to model the possible outcomes of nuclear fission, where the resulting fragments of the nucleus [Dés02] correspond to a partition of the total number of protons and neutrons in the nucleus [Ski98, §1.3.6]. Partitions have also been used to develop a model of frequency fluctuations in a quartz crystal resonator [Pla03], and several other physical applications [KKZL05, TMB04, GH99, Act94, Tem49]. The Rogers-Ramanujan identities [BS05, §2], an important component in the theory of partitions, have seen several significant applications in physics [BM98]. For instance, “the Rogers-Ramanujan identities are crucial in studying the behaviour of liquid helium on a graphite plate” [And05, §4]. Concepts closely related to partitions also have applications in physics, such as group theory [Tun99, p.65] and Young tableaux [Ful97], which are essential “in the analysis of the symmetries of many electron systems” [Sch86, p.226]. Applications for partitions also arise in biology. In population genetics partitions are used in “models for the genetic variation of a set of gametes from a large population” [Kin78], and related models apply in Brownian motion [Pit97] and other important applications [FH98, HY97].

1.2 Combinatorial Generation

Combinatorial generation algorithms allow us to systematically traverse all possibilities in some combinatorial universe and have many applications in the study of “organised complexity” [Bec64, p.xi]. For instance, combinatorial generation algorithms are currently used to develop and elaborate theoretical models for biological data (arising, for example, from genome sequencing) [Sag00]. Combinatorial generation has also been suggested as a means of simplifying the search for drug-like compounds [BBGP04], along with other applications in computational chemistry [Saw01].

Combinatorial generation algorithms are also valuable from a mathematical perspective. Generation algorithms are useful to researchers investigating the properties of some class of combinatorial object. By having a systematic and reliable procedure to generate all of the patterns of interest, we can test hypotheses over much larger sets of data than otherwise possible. If a particular hypothesis holds when applied to a large number of objects, we may then begin the process of deriving a formal proof. The study of generation algorithms may also provide new mathematical insights and proofs. For example, Nijenhuis & Wilf use a random selection algorithm to prove an enumeration formula for Young tableaux [NW78, p.118–119]. (There is an example of such interplay between algorithms and mathematics in this dissertation. In Section 5.3.1 we develop a new algorithm to generate ascending compositions and the analysis of this algorithm allows us to prove an enumeration formula for partitions on page 143.)

To allow us to be more concrete about what we mean by a ‘combinatorial generation’ algorithm let us take a specific example. In Figure 1.1 we give an example of a generation procedure and a consumer procedure. These examples are clearly trivial, but serve to illustrate the general points that we shall be making. In this example we have a generation procedure \mathbb{G} which generates the sequences 1111, 112, 13, 22 and 4. The consumer procedure \mathbb{C} , using \mathbb{G} to provide the values in question, then computes the product of all parts in partitions of 4. (The product of all parts in partitions of n is equal to the determinant of the character table of a symmetric group on n

§ 1.2. Combinatorial Generation

Procedure $\mathbb{C}()$	Procedure $\mathbb{G}()$
$p \leftarrow 1$ for all $a_1 \dots a_k$ in $\mathbb{G}()$ do for $j \leftarrow 1$ to k do $p \leftarrow p \times a_j$ end for end for return p	$a_j \leftarrow 1$ for j in $\{1, \dots, 4\}$ visit $a_1 \dots a_4$ $a_3 \leftarrow 2$ visit $a_1 \dots a_3$ $a_2 \leftarrow 3$ visit $a_1 \dots a_2$ $a_j \leftarrow 2$ for j in $\{1, \dots, 2\}$ visit $a_1 \dots a_2$ $a_1 \leftarrow 4$ visit $a_1 \dots a_1$

Figure 1.1: Example generation and consumer procedures. The generation procedure generates all ascending compositions of 4 in lexicographic order, and the consuming procedure uses these to compute the product of all parts in partitions of n .

elements [SS84]; see also the corresponding sequence in the Online Encyclopedia of Integer Sequences [Slo05, Seq.A007870].) Thus, \mathbb{C} will receive the sequences 1111, 112, 13, 22, 4 in that order, and will compute the value 96.

The generation algorithm \mathbb{G} successively populates the array a with the aforementioned sequences and each time a complete sequence is present in the array, \mathbb{G} makes it available to \mathbb{C} . This is done using the **visit** keyword [Knu04b, p.1], which makes the appropriate portion of the array available to \mathbb{C} . Each time the **visit** keyword is invoked control is handed over to \mathbb{C} , which performs the relevant calculations. Then, each time \mathbb{C} requests another element from \mathbb{G} , execution of \mathbb{G} is resumed from the point that the last visit statement was executed.

Digressing, for a moment, from the particular case of *combinatorial* generation, there are some software design advantages of the framework we have just outlined. The consuming procedure \mathbb{C} illustrates the advantages of using generator algorithms in a more general sense. In the pseudocode description we treat \mathbb{G} *exactly* as if it were a data structure holding a set of sequences, which we consider one at a time. Thus, using generator algorithms we can simulate an arbitrarily large data structure using a finite amount of memory. This is an extension of the well-known Iterator Pattern of object oriented

§ 1.2. Combinatorial Generation

software design [GHJV95, p.257]. Weihe [Wei01, §7.2] has discussed the advantages and disadvantages of algorithmic generators, and efforts have been made to integrate the necessary language features into programming languages such as Sather [MOSS96] and Python [SPH01].

Returning to combinatorial generation, according to Flajolet, Nebel & Prodinger, the “subject of random generation and exhaustive listing of combinatorial structures is currently an active one” [FNP06]. Most recently, Martínez & Molinero have developed a generic means of generating a range of combinatorial objects [MM05], building on the work of Flajolet, Zimmerman & Van Cutsem [FZC94] on the random generation and enumeration of general combinatorial structures. In a forthcoming volume of *The Art of Computer Programming*, Knuth provides algorithms to generate all of the basic combinatorial patterns, including n -tuples [Knu04b], permutations [Knu04d], combinations [Knu04a], set and integer partitions [Knu04c], and trees [Knu04e], along with providing a history of the subject of combinatorial generation [Knu04f]. In some cases, many different algorithms exist to generate the same object. Indeed, in the case of permutations, Knuth notes that “almost as many algorithms have been published for unsorting as sorting” [Knu04d, p.1]. Sedgewick performed a survey of all known permutation generation methods in 1977, when there were more than thirty known algorithms [Sed77].

Generating the fundamental combinatorial patterns efficiently is an important problem, but it is also important to have efficient algorithms to generate specific subsets of these patterns. For example, a subset of the binary n -tuples known as necklaces have been used to classify certain DNA sequences [CL97]. A necklace of n beads of k colours is an equivalence class of n -tuples under rotation, and efficient algorithms have been published to generate necklaces [RSW92]. Furthermore, algorithms exist to generate restricted classes of necklace such as binary unlabelled necklaces [CRS⁺00], fixed density necklaces [RS99] and bracelets [Saw01]. We can generate alternating permutations [BR90] and permutations with a specified number of inversions [ER03]. Generating binary trees has also been well studied: more than thirty algorithms exist to accomplish this task [AS96]. Many more algorithms

§ 1.3. Thesis and Dissertation Overview

exist to generate classes of restricted tree, including free trees [WROM86], rooted trees [BH80], plane trees [Nak02], ordered trees [Ska88], non-regular trees [Er88], t -ary trees [Rus78, Tro78], 2-3 trees [GLW82], AVL trees [Li86] and B-trees [GLW83, BS94]. There are algorithms to generate formally specified languages [Kem98] and for more specific cases such as generalised Dyck languages [Lie03]. This list is far from exhaustive, and only a selection of the available algorithms has been mentioned here.

The point of the previous paragraphs is to establish that it is considered important not only to have efficient generation algorithms for the basic combinatorial patterns, but also to be able to efficiently generate restricted classes of these basic patterns. We attend to both of these problems in this dissertation: we design the most efficient known algorithm to generate *all* partitions, and also algorithms to generate a large class of restricted partition, for which no generation algorithm currently exists.

1.3 Thesis and Dissertation Overview

To generate partitions efficiently we must encode these unordered collections of positive integers as an *ordered* sum. For example, the ordered sums $1+1+2$, $1+2+1$ and $2+1+1$ all represent the same partition of 4 but are unique when the order of the parts is taken into consideration. Such ordered sums are known as ‘compositions’, and we shall refer to compositions in which the summands are arranged in nondecreasing order (e.g. $1+1+2$) as ‘ascending’ compositions and those in which the summands are in nonincreasing order (e.g. $2+1+1$) as ‘descending’ compositions. Clearly, generating all ascending or descending compositions of n will also provide us with all partitions of n . Thus, we shall speak of encoding partitions as ascending or descending compositions.

One goal of this dissertation is to demonstrate that it is fundamentally easier to generate all ascending compositions than it is to generate all descending compositions. This is an important result because all known algorithms to generate partitions in lexicographic order actually generate *descending compositions*. We have seen that partitions are a foundational math-

§ 1.3. Thesis and Dissertation Overview

emational concept used to model a wide variety of natural processes and that generation algorithms are a valuable tool in the process of accumulating reliable knowledge. An increase in the efficiency of generation algorithms widens the range of the problems they may be applied to, and the theoretical insights gained by the detailed study of ascending and descending compositions deepens our knowledge of partitions themselves.

There is currently a noticeable lack of efficient algorithms to generate classes of restricted partition. Ad hoc methods exist to generate partitions with a limited set of restrictions, but no unified approach to the problem has been presented. By encoding partitions as ascending compositions, however, we open up the possibility of efficiently generating a wide variety of restricted partition. In this dissertation we develop a coherent theoretical framework within which we can concisely specify and efficiently generate partitions with a flexible class of restriction imposed on the relationship between parts. We call partitions defined within this framework ‘interpart restricted compositions’ and we provide a simple means of enumerating these partitions and several algorithms to efficiently generate them.

These are the high-level aims of this dissertation: to demonstrate that generating all ascending compositions of n is less computationally intensive than generating all descending compositions of n , and to provide a useful theoretical framework for defining classes of restricted partition. In the interest of rigour we shall be defending a particular thesis in the dissertation. We now state this thesis and make our specific claims more precisely.

1.3.1 Thesis

The thesis defended in this dissertation is the conjunction of the following subtheses. We discuss each of these in turn and refer to the sections of the dissertation where the most pertinent evidence supporting each claim is assembled.

§ 1.3. Thesis and Dissertation Overview

Subthesis 1. *Important classes of restricted partition can be expressed concisely by a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$.*

Discussion. In Chapter 3 we develop the theory of interpart restricted compositions, a framework within which we can describe classes of restricted partition using an integer function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. Combinatorially important classes of restricted partition can be concisely described within this framework. For example, the partitions into distinct parts are defined by the function $\sigma(x) = x + 1$; the Rogers-Ramanujan partitions by $\sigma(x) = x + 2$; and the Göllnitz-Gordon partitions by $\sigma(x) = x + 2 + [x \text{ even}]$. The theory of interpart restricted compositions is fully developed and exemplified in Section 3.1.

Subthesis 2. *It is possible to define an efficient algorithm to enumerate partitions with restrictions expressed by such a function.*

Discussion. In Section 3.2 we develop a general purpose recurrence equation to enumerate interpart restricted compositions for any instance of the restriction function. We then define a dynamic programming enumeration algorithm that computes the number of interpart restricted compositions for all positive integers $n \leq N$ in $O(N^2)$ time and space.

Subthesis 3. *It is possible to efficiently generate all restricted partitions where the corresponding restriction function is nondecreasing.*

Discussion. In Chapter 4 we develop algorithms to generate all interpart restricted compositions of n for any nondecreasing instance of the restriction function. In Section 4.2 we define a recursive algorithm; in Section 4.3 we define an abstract succession rule for interpart restricted compositions and implement it as an iterative generation algorithm; and in Section 4.4 we develop some auxiliary theory and use this to improve the performance of the direct succession rule. For each of the algorithms defined we prove its correctness and prove that it has the required constant amortised time property.

§ 1.3. Thesis and Dissertation Overview

Subthesis 4. *It is possible to define algorithms to generate ascending compositions that are more efficient than the most efficient known commensurable algorithms to generate descending compositions.*

Discussion. In Chapter 5 we systematically compare the most efficient known algorithms to generate the unrestricted partitions of n with concrete instantiations of the algorithms we developed in Chapter 4. In each case, the existing algorithms encode partitions as *descending* compositions (see Section 2.2.3 for a discussion of encoding and descending compositions) because of the current convention of concretely defining a partition as a descending composition. We trace the origins of this convention in Section 2.2.2. The algorithms we describe all generate *ascending* compositions, which encode unordered partitions with equal validity. In Section 5.2.3 we compare Ruskey’s recursive descending composition generator [Rus01, §4.8] with our commensurable ascending composition generator. In Section 5.3.3 we compare direct implementations of the succession rules for ascending and descending compositions. Finally, in Section 5.4 we compare the most efficient examples of ascending and descending composition generation algorithms (our accelerated algorithm and Zoghbi & Stojmenović’s algorithm [ZS98] respectively). In each case our ascending composition generation algorithm is significantly more efficient, as is demonstrated in Section 5.4.3.

1.3.2 Dissertation Overview

Chapter 2 serves to establish the background for the results we shall derive by examining the basic ideas of combinatorial generation and reviewing all known partition and composition generation algorithms. Chapter 3 begins the process of defending our thesis by introducing and exemplifying interpart restricted compositions and establishing some fundamental results. Chapter 4 proceeds by developing, analysing and proving the correctness of three algorithms to generate interpart restricted compositions: a recursive algorithm, a succession-rule and an ‘accelerated’ generation algorithm. Chapter 5 proceeds by comparing the most efficient known examples of descending composition generation algorithms with concrete instances of the

§ 1.3. Thesis and Dissertation Overview

algorithms developed in Chapter 4. Chapter 6 concludes this dissertation by defending the above thesis and outlining some possibilities for future work.

Chapter 2

Literature Review

In Section 2.1 we briefly review the literature on combinatorial generation in general, and examine the most basic issues that arise in any generation algorithm: generation order and analysis. Then, in Section 2.2, we take a more detailed look at partitions and compositions. We first examine the basic properties of both objects, and then address the problem of how to encode partitions efficiently for the purpose of generation. In Section 2.3 we systematically review all known generation algorithms for partitions and compositions. We are interested in both algorithms to generate restricted classes of partition and algorithms to generate all partitions, and we therefore treat these cases separately. Finally, in Section 2.4 we summarise the state-of-the-art in the relevant literature.

2.1 Combinatorial Generation

In this section we address the general issues common to all combinatorial generation algorithms. We begin by identifying some resources that address the problem of combinatorial generation explicitly in Section 2.1.1. Then, in Section 2.1.2 we discuss the various orders that have been used to generate combinatorial objects. In Section 2.1.3 we deal with the issue of the analysis of generation algorithms, and discuss the metrics that have been proposed to quantify the efficiency of generation. In Section 2.1.4 we review the prob-

§ 2.1. Combinatorial Generation

lems of random selection and ranking/unranking, which are closely related to systematic generation.

2.1.1 General Background

The first systematic exposition on the subject of combinatorial generation was presented by Lehmer [Leh64] who described generation algorithms for permutations, combinations, partitions and compositions. Wells [Wel71, ch.5], Reingold, Nievergelt & Deo [RND77, ch.5] and Page & Wilson [PW79, ch.5] each dedicate a chapter to the problem of generating ‘elementary’ combinatorial configurations and provide algorithms to generate permutations, combinations and several other fundamental combinatorial patterns. Sedgewick’s review of permutation generation methods [Sed77] provides many deep insights into combinatorial generation. Nijenhuis & Wilf [NW78] provide both generation and random selection algorithms for several classes of combinatorial pattern, including some lesser known objects such as Young Tableaux [Ful97]. More recently, Stanton & White [SW86], Skiena [Ski90] and Pemmaraju & Skiena [PS03] have provided algorithms to systematically generate combinatorial objects as part of their wider treatment of combinatorial algorithms, as do Kreher & Stinson [KS98, ch.2]. Ruskey’s Combinatorial Object Server [Rus05] provides a HTML interface to generation algorithms, along with an information page, for many different combinatorial objects.

Two books of interest for this dissertation are currently in preparation. Ruskey’s monograph [Rus01] is dedicated to combinatorial generation and contains algorithms to generate many classes of object not covered in the references given above. The second work is volume four of *The Art of Computer Programming*, which deals extensively with the problem of combinatorial generation. Specifically, Knuth studies generating all n -tuples [Knu04b], permutations [Knu04d], combinations [Knu04a], integer and set partitions [Knu04c], and trees [Knu04e]. The history of combinatorial generation is also discussed [Knu04f], and the terminology and notation we use in this dissertation is largely that advocated by Knuth [Knu04b, p.1].

§ 2.1. Combinatorial Generation

2.1.2 Generation Order

Lexicographic order is the most common generation order for combinatorial objects. Lexicographic corresponds to the common ‘dictionary’ order and, in certain instances, leads to direct and efficient generation algorithms. For example, the binary 3-tuples in lexicographic order are

000, 001, 010, 011, 100, 101, 110, 111.

We can see that all tuples sharing a common prefix occur consecutively (e.g., 010 and 011 share the prefix 01) and this is a property of lexicographic generation algorithms in general. This property is useful, for instance, if we are searching for a feasible solution to some problem and we have some partial information about the solution. Kemp [Kem98] studied a general class of lexicographic generation algorithm and derived powerful general results concerning the complexity of generating all words of an ordered language in lexicographic order.

After lexicographic order, the most commonly used ordering for generation is the Gray code (minimal change) ordering. A generation algorithm that utilises a Gray code order generates objects such that successively visited objects differ in a small, pre-specified way. The canonical example is the binary-reflected Gray code, which produces 3-tuples in the order

000, 001, 011, 010, 110, 111, 101, 100.

In the binary reflected Gray code order we transition between consecutive tuples by flipping exactly one bit; in contrast, the lexicographic ordering above needs to flip a maximum of three bits. Other examples of Gray code orders include generating all permutations of $1 \dots n$ such that successive permutations differ only by interchanging one pair of adjacent elements [Knu04d, p.2] and generating all k -subsets of an n element set so consecutive sets differ by exactly one element [Knu04a, p.8]. A comprehensive survey of combinatorial Gray codes and their applications has been compiled by Savage [Sav97]. A variant on the idea of Gray codes has also been studied, where successive

§ 2.1. Combinatorial Generation

objects differ as *much* as possible [Wil89, ch.3].

Gray codes have two commonly attributed advantages over lexicographic generation orders. Firstly, as the difference between consecutive objects is small, it may be possible to implement these transitions highly efficiently. (Bitner, Ehrlich & Reingold’s algorithm [BER76] to generate n -tuples in binary-reflected Gray code order is “almost blindingly fast” [Knu04b, p.11], as it requires only five assignment operations and one test for termination to perform each transition.) Secondly, a minimal change order potentially makes the consuming procedure significantly more efficient. If we know that successive objects differ in some well-defined way, then it may be possible to reduce the computation required when visiting each object. If we are searching for a feasible solution to some particular problem we may also improve the efficiency of the search as “it is likely that combinatorial objects which differ in only a small way are associated with feasible solutions which differ by only a small amount of computation” [Sav97].

Gray code generation is not necessarily more efficient than lexicographic generation since the transition rules are rarely as simple as those required for the binary-reflected Gray code exemplified above. For instance, Akl [Akl81] performed an empirical analysis of nine different methods for generating combinations, and found that a lexicographic generator was more efficient than the various Gray code schemes.

2.1.3 Analysis

When generating all combinatorial objects of a given class there is one essential property required of the generation algorithms we define. We seek to define generation algorithms in which the total time spent generating objects is proportional to the number of objects generated. As we must systematically examine *every* object in the combinatorial universe U , then clearly the minimum time required to generate all of these patterns is $O(|U|)$. If the total time is not proportional to the number of objects generated, given the large number of objects we are potentially generating, systematic generation quickly becomes impracticable. This requirement, that the amount of

§ 2.1. Combinatorial Generation

time to generate all elements of U must be $O(|U|)$, is the basic idea behind the concepts of ‘loopless’ and ‘constant amortised time’ algorithms, although there are some subtle and important differences between the two metrics.

An algorithm is said to be loopless [Ehr73b] if the amount of computation required between successive visits is bounded in advance and there is never a long delay while a new pattern is generated [Knu04b, p.9]. A slightly weaker requirement is that the amount of work to effect transitions is constant on average: the total amount of work required to generate all of the objects is proportional to the number of objects generated. An algorithm satisfying this requirement is said to be constant amortised time [Rus01, §1.7], as the total time required to generate each object is constant in an amortised sense [CLR90, ch.18]. More precisely, we say that an algorithm is constant amortised time if the average amount of time required to generate an element of U is bounded, from above, by some constant. Many constant amortised time generation algorithms have been published, for example Ruskey et al. [RECS94], Effler & Ruskey [ER03] and Boyer [Boy05]. Constant amortised time performance has been referred to as the “ultimate goal in efficiency” [MM05] for combinatorial generation algorithms.

Note that if we are only considering the total amount of time required to generate the combinatorial universe U the loopless and constant amortised time properties tell us essentially the same thing: the total time will be proportional to the number of elements in U . It is when we consider the expected time required to generate a *particular* object in U that the difference between the two properties arises. With a loopless algorithm the interval between any two visits is guaranteed to be constant, whereas with a constant amortised time algorithm the interval between any two particular visits may not be constant. Thus, we can see that any loopless algorithm is constant amortised time, but a constant amortised time algorithm is not necessarily loopless.

Several concepts that are equivalent to either the loopless or constant amortised time properties have been used in the literature to discuss the performance of combinatorial generation algorithms. Goldberg [Gol93] for instance, discusses the ‘delay’ (an algorithm with constant delay is loopless)

§ 2.1. Combinatorial Generation

or ‘cumulative delay’ (an algorithm with constant cumulative delay is constant amortised time). Similarly, Zoghbi & Stojmenović [ZS98] speak of the ‘average delay’ of a generation algorithm and Barnes & Savage [BS97] speak of the ‘average time’ required to generate objects. All of the aforementioned concepts are equivalent to either the loopless or constant amortised time properties and we shall use the terms ‘loopless’ or ‘constant amortised time’ to describe the performance of algorithms throughout this dissertation.

The loopless and constant amortised time properties provide us with a useful high-level means of describing the performance of algorithms. In this dissertation, however, we shall be making detailed comparisons of similar algorithms. We shall therefore require a more sophisticated metric for comparison. In his general treatment of the problem of generating all words in a formal language [Kem98], Kemp pioneered the use of counting the total number of *read* and *write* operations. Kemp examines the general problem of generating all words in a formal language \mathcal{L} in lexicographic order. In his formulation we consider the problem of transforming each word $w \in \mathcal{L}$ into its immediate lexicographic successor w' . To do this we let $w = uv$, where u is the longest common prefix of w and w' , and then generate the new suffix v' , thereby obtaining $w' = uv'$. The problem of generating w' is then broken into two steps:

1. scan w from right-to-left until we find the last letter of the prefix u ;
2. attach the new suffix v' to the end of u .

The complexity of this process can be quantified by counting the number of letters of w we must scan to find the last letter of u (the number of *read* operations), and counting the number of new letters we must write to produce v' (the number of *write* operations). By summing the total number of read and write operations over all words in the language, we then know the complexity of generating the language. Kemp then provides general results for the complexity of generating all fixed-length words in a language and provides specific results on the complexity of generating regular sets, permutations, subsets, semi-Dyck words, Motzkin words, t -ary trees and ordered trees of various kinds.

§ 2.1. Combinatorial Generation

It is essential to note here that Kemp’s method allows us to analyse the difficulty of generation without reference to a specific implementation of an algorithm. Indeed, Kemp’s own analyses depend only on the combinatorial properties of the objects in question, and he pays attention to “algorithmic aspects but for the sake of a complete presentation only” [Kem98, p.83]. He goes on to note that “the advantage of such a strict separation between algorithmic aspects and aspects concerning complexity is quite obvious: Generally, there are many different encodings of the combinatorial objects to be generated. We can determine the average running time for each of these encodings first and then decide which encoding is worthwhile to explicitly develop a generating algorithm.” The issue of encoding is intrinsic to this dissertation, and so we shall adopt Kemp’s metric when performing our analyses in later chapters.

2.1.4 Auxiliary Problems

The problems of ranking/unranking [MM01] and randomly selecting [NW75] elements of a combinatorial universe U are closely related to the problem of generating all elements of U . There are important distinctions between these problems, however, and we therefore discuss the concepts of ranking, unranking and random selection briefly in this subsection.

Given a listing of U in some order, the *rank* of a given object is the number of objects preceding it in the list [KS98, p.31]. For example, here again are the binary 3-tuples in lexicographic order where each tuple is accompanied by its rank.

0	1	2	3	4	5	6	7
000,	001,	010,	011,	100,	101,	110,	111

A ranking algorithm determines the number of objects that precede a given object in the listing. Thus, if we were asked to rank the tuple 100 in the listing above we would return 4. An unranking algorithm does the opposite: we return the object that occurs at a particular rank within a given list. Thus, if we were asked to unrank the value 6 in the above list, we would return

§ 2.2. Compositions and Partitions

the tuple 110. Ranking and unranking algorithms can be obtained from enumeration formulas using general techniques [NW78, ch.13]. More efficient specialised algorithms also exist to rank and unrank many different types of combinatorial object, including permutations [MR01], B-trees [GLW83], 2-3-trees [GLW82], AVL-trees [Li86] and a generic class of labelled combinatorial object [MM01].

Unranking algorithms can certainly be used as generation algorithms. It is a simple matter to iterate through all possible ranks and to visit the unranking of each. This approach is inefficient for two reasons. The first reason is that, given that the number of configurations is likely to be large, we will need to store and manipulate very large integers to perform the unranking. Secondly, each unranking is treated as a separate problem and does not reuse any shared elements of successive patterns.

Another closely related problem to combinatorial generation is the random selection of combinatorial objects. In this problem we wish to select *one* object from the combinatorial universe U such that all elements of U have equal *a priori* probability of being selected [NW78, p.4]. Nijenhuis & Wilf [NW75] have devised a general means of solving this problem, requiring only a recurrence relation to enumerate the objects in the universe in question. Flajolet, Zimmermann & Van Cutsem [FZC94] have developed a very general approach for the random selection of decomposable combinatorial structures, requiring only a specification of the structure in question. Generating combinatorial objects at random is, in itself, a large area of study — see Stojmenović [Sto92] for further references to the relevant literature.

2.2 Compositions and Partitions

The standard reference for the theory of partitions begins by “acknowledging that the word ‘partition’ has numerous meanings in mathematics. Any time a division of some object into subobjects is undertaken, the word partition is likely to pop up” [And76, p.xiii]. This is also undoubtedly true of computer science, where the term is used for many purposes. The term ‘composition’ is equally overloaded: we may speak of the composition of functions, the

§ 2.2. Compositions and Partitions

compositions of graphs and so forth. Indeed, we are likely to encounter the word ‘composition’ any time the combining of parts to form a whole is undertaken.

The purpose of this section is therefore largely terminological. In Section 2.2.1 we examine the basic properties of compositions of integers, and repeat this in Section 2.2.2 for partitions of integers. Then, in Section 2.2.3 we examine an issue that will dominate Chapter 5: the choice of encoding for partitions for the purpose of systematic generation.

2.2.1 Compositions

A composition of a positive integer n is an expression of n as an ordered sum of positive integers [Sta86, p.14]. A composition $a_1 + \cdots + a_k = n$ can be represented by the sequence $a_1 \dots a_k$ without loss of information. A composition with exactly k summands (or *parts*) is referred to as a k -composition, and if the k is omitted the number of parts is understood to be arbitrary, following the conventions of Stanley [Sta86, p.14]. Enumerating compositions is trivial: there are (as we shall see momentarily) 2^{n-1} unrestricted compositions of n , and $\binom{n-1}{k-1}$ k -compositions of n .

Compositions have many connections to other combinatorial objects. For example, a direct correspondence between the k -compositions of n and the $(k-1)$ -subsets of $\{1, \dots, n-1\}$ exists, and can be established using the following bijective function. Given a k -composition $a_1 \dots a_k$, we define the corresponding subset to be

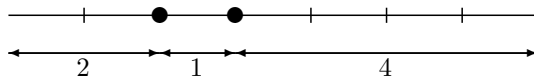
$$\theta(a_1 \dots a_k) = \{a_1, a_1 + a_2, \dots, a_1 + \cdots + a_{k-1}\},$$

thus demonstrating that the number of k -compositions of n is $\binom{n-1}{k-1}$, and that the total number of compositions of n is 2^{n-1} [Sta86, p.14]. MacMahon [Mac93, p.887-888] demonstrated a correspondence between the compositions of n and trees. MacMahon proved that there is a one-to-one correspondence between the ways we can place q symbols between the summands in the compositions of n and the trees of height q with n leaf nodes. Many other correspondences between compositions and well-known objects exist.

§ 2.2. Compositions and Partitions

For example, the number of compositions with no occurrence of the summand 1 is equal to the $(n - 1)$ th Fibonacci number [Gri01]. The theory of compositions has also found applications to specific problems such as polyomino enumeration [Odl96, p.1104] and the four colour conjecture [Cas04, p.62]. For further information on the theory of compositions see MacMahon [Mac93, Mac08], MacMahon [Mac15, Vol.I Sec.IV] or Andrews [And76, ch.3].

A useful graphical representation for compositions was introduced by MacMahon [Mac93, p.836]. In this graphical representation we draw a line of length n and mark each of the $n - 1$ internal unit divisions along the line. Then, counting from the left hand side of the line, we put a (different) mark at each point corresponding to a part of the composition. Thus, the composition 214 becomes



in MacMahon's graphical representation, as we have divided the line into segments of size 2, 1 and 4. The binary correspondence with this graphical representation is then obvious: for every point on the line, we record a 0 if the point has not been marked, and a 1 if it has. Thus, 214 becomes 011000 in what is known as the 'binary difference representation' [PW79, §5.4]. We can also obtain this representation for an arbitrary composition $a_1 \dots a_k$ more directly. For each $1 \leq j \leq k$, append $a_j - 1$ copies of 0 and a single 1; and finally strip away the last 1. Compositions can equivalently be viewed as tilings of a 1-by- n board with 1-by- q (where q is arbitrary) tiles [CH03c]. In this case we associate each part a_j in the composition $a_1 \dots a_k$ with a tile of length a_j . Thus, the total length of all the tiles is n , and each composition is associated with a unique tiling. All compositions of 4 are given in Table 2.1, along with the corresponding subset of $\{1, \dots, n-1\}$, tiling and binary $(n-1)$ -tuple in the difference representation.

Closely related to compositions are the representations of n as an ordered sum of nonnegative parts — compositions where parts with value 0 are al-

§ 2.2. Compositions and Partitions


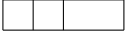


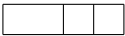
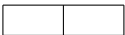


$a_1 + \dots + a_k$	$A \subseteq \{1, \dots, n\}$	Tiling	$b_1 \dots b_{n-1}$
$1 + 1 + 1 + 1$	$\{1, 2, 3, 4\}$		111
$1 + 1 + 2$	$\{1, 2, 4\}$		110
$1 + 2 + 1$	$\{1, 3, 4\}$		101
$1 + 3$	$\{1, 4\}$		100
$2 + 1 + 1$	$\{2, 3, 4\}$		011
$2 + 2$	$\{2, 4\}$		010
$3 + 1$	$\{3, 4\}$		001
4	$\{4\}$		000

Table 2.1: All compositions of 4 with corresponding subsets of $\{1, \dots, n\}$, tilings of a 1-by- n board with 1-by- q tiles, and binary $(n - 1)$ -tuples.

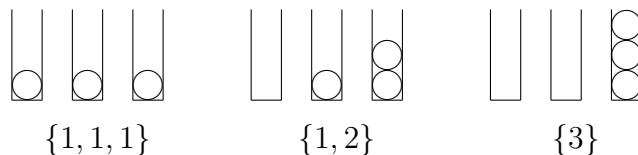
lowed. Following Stanley [Sta86, p.14] we refer to such ordered sums as *weak* compositions (or when the number of parts is fixed, weak k -compositions). We are not directly concerned with weak compositions in this dissertation and only mention them here to avoid confusion. There has been an unfortunate recent tendency in the literature on combinatorial generation to refer to weak compositions as ‘compositions’ without the qualifying prefix. For example, Pemmaraju & Skiena state that a “composition of n is a particular arrangement of nonnegative integers that sums to n ” [PS03, p.146]. (See Nijenhuis & Wilf [NW78, ch.5] and Skiena [Ski90, p.60] for similar definitions.) To further confuse matters, Bóna takes the opposite approach to Stanley [Sta86, p.14]: he defines a composition as a collection of nonnegative integers and a weak composition as a collection of positive integers [Bón02, p.89].

We shall adhere to the MacMahon’s original definition of the term ‘composition’ [Mac93] in this dissertation in accordance with the majority of the mathematical literature on the subject. See, for example, Andrews [And76, p.54], Bóna [Bón04, p.46], Charalambides [Cha02, p.401], Comtet [Com74, p.123], Dence & Dence [DD99, p.363], Goulden & Jackson [GJ04, p.51], Heubach & Mansour [HM04], Finch [Fin03, p.293], Riordan [Rio58, p.107] or Schumer [Sch04, p.154–155].

§ 2.2. Compositions and Partitions

2.2.2 Partitions

A partition of a positive integer n is an unordered collection of positive integers whose sum is n . We can therefore formalise a partition of n as a multiset of positive integers that sums to n [GS96], as this encapsulates the fundamental requirement that the summands are *unordered*. The function $p(n)$ is conventionally used to denote the number of partitions of n , and this function has been the subject of much study — see Andrews [And76] for information on the Hardy-Ramanujan-Rademacher formula for $p(n)$. One simple combinatorial interpretation of partitions is to consider the distribution of n identical balls into n identical urns [Knu04c, p.1]. For example, if we have 3 balls and 3 urns the possibilities are



In each of the distributions the corresponding multiset of positive integers is given. Partitions have numerous connections to disparate areas of mathematics which are beyond the scope of this dissertation. See Section 1.1 for further references in this regard.

The multiset formalisation of partitions correctly encapsulates the requirement that partitions are an unordered structure. It is, however, conventional to define partitions as an *ordered* sequence of positive integers. For example, Andrews defines a partition of n as a “a finite nonincreasing sequence of integers whose sum is n ” [And76, p.xiii]. As there is a unique way of writing any unordered collection in nonincreasing order, there is a unique representation of each partition of n as such an ordered sum¹.

It is therefore conventional to define a partition as a sequence of integers whose sum is n with parts in descending order. This fact is of great importance to this dissertation, and so we shall take some care to trace the historical reasons for the convention. In the late nineteenth century Sylvester

¹For our purposes the terms ‘ascending’ and ‘descending’ are synonymous with ‘non-decreasing’ and ‘nonincreasing’, respectively.

§ 2.2. Compositions and Partitions

et al. founded the modern combinatorial theory of partitions [And88]. Before the work of Sylvester et al., the theory of partitions consisted largely of the manipulation of algebraical identities [Har66] and rested on the “intuitive observation of Euler” [Mac15, Vol.II Sec.VII ch.I]. Sylvester’s 1882 omnibus paper [Syl82] summarises many of the results obtained in the “new method of partitions”, and begins as follows.

In the new method of partitions it is essential to consider a partition as a *definite thing*, which end is attained by regularization of the succession of its parts according to some prescribed law. The simplest law for the purpose is that the arrangement of the parts shall be according to their order of magnitude. [Syl82, emphasis Sylvester’s]

Here, Sylvester initiates today’s convention of defining a partition as a sequence of integers, although he does not specify the particular order to be applied. MacMahon developed Sylvester’s ‘constructive theory of partitions’ further, and made the definition of a partition more concrete.

A partition of a number may be regarded as any collection of positive integers whose sum is equal to the number. There is no specification of order amongst the numbers which are the parts of the partition, and that being so, we may import into the definition any order or arrangement that is convenient. There are only two arrangements that are universally applicable. We may in all cases arrange the parts either in descending order or ascending order of numerical magnitude. We choose the former of these and make a new definition of a partition of a number, viz.: “A partition of a number is any collection of positive integers *arranged in descending order of magnitude* whose sum is equal to the number.” [Mac15, Vol.II p.91, emphasis MacMahon’s]

Sylvester and MacMahon considered it essential that partitions be given a fixed ordering (the theory they established on this premise has been extremely successful [Pak05]) and the ordering that has emerged is the ‘descending’ order. Many computations involving partitions are indeed more

§ 2.2. Compositions and Partitions

convenient if we enforce the descending order, but the choice seems to be, fundamentally, an arbitrary one. It seems that MacMahon might equally have chosen the ascending order.

There are exceptions to this rule. In one of the first texts on the subjects of combinatorial computing, Lehmer [Leh64, p.25] states that the “compositions of n whose components are monotone nondecreasing are called the unrestricted partitions of n .” Tucker [Tuc84, p.240] defines a partition as a “collection of positive integers” and states that “normally we write this collection as a sum and list the integers in increasing order”. There have been several recent examples in the applied and mathematical literature of partitions with parts arranged in ascending order, such as Lecture-hall partitions [BME97a, Yee01], non-squashing partitions [SS05], M -partitions [O’S04], in counting the total number of parts in partitions [KR05] and in applications to molecular symmetry [Öhr00, p.114].

The point of this historical digression is that all known partition generation algorithms have followed this standard convention of enforcing a descending order on the parts in partitions. The consequences of generating partitions with an ascending order on parts has not been considered. We shall examine this problem in detail in Chapter 5. For now, we shall make the problem a little more precise, by discussing the issue of *encoding* for partitions in the next subsection.

2.2.3 Encoding Partitions

It is not controversial to state that the encoding we use to represent combinatorial objects in memory affects the efficiency of generation. As Kemp noted, “generally, there are many different encodings of the combinatorial objects to be generated” [Kem98, §4]; not all of these encodings will lead to equally efficient algorithms. Considering the problem in the case of partitions, we appear to have three immediate options. The first choice is to encode partitions as an unordered multiset data structure. This provides us with the most direct encoding of partitions as they are fundamentally defined as an unordered structure. This approach, however, will inevitably lead to an inef-

§ 2.2. Compositions and Partitions

efficient generation algorithm because of the inherent difficulty of maintaining an unordered structure in computer memory [CLR90, Sec.III]. Our other two options encode partitions as an ordered structure, as a sequence of positive integers in either ascending or descending order². Such encodings mean that we are generating a subset of the *compositions* of n , either the ascending or descending compositions.

For the remainder of this dissertation we shall not speak of ‘partition’ generation algorithms, but of ascending or descending composition generation algorithms, depending on the encoding used. This distinction may seem pedantic but, as we shall see in Chapter 5, there are real and important differences between ascending and descending composition generation algorithms. For the moment it will suffice to demonstrate that generating ascending and descending compositions are at least *different* problems. We do this by means of an example.

In this example we examine the sets of ascending and descending compositions of 5, $\mathcal{A}(5)$ and $\mathcal{D}(5)$ respectively, arranged in lexicographic order. Each element in these sets corresponds to exactly one partition.

	0	1	2	3	4	5	6
$\mathcal{A}(5) =$	11111	1112	113	122	14	23	5
$\mathcal{D}(5) =$	11111	2111	221	311	32	41	5

We can see in this example that the same partitions of n occur at ranks 0, 1 and 6, as, for example, the compositions 1112 and 2111 represent the same partition. At all other ranks, however, the partitions represented are *not* equal. Clearly then, generating ascending and descending compositions are fundamentally different problems. We shall discuss this asymmetry in detail in Chapter 5. For now it is sufficient to note that generating partitions is not *necessarily* synonymous with generating descending compositions, and we are therefore justified in identifying algorithms as ascending or descending composition generators. Having established the basis for this terminological

²Binary encodings of partitions have also been considered in the literature [Com55]; generation algorithms utilising these encodings have not been investigated, and so we shall not consider them further here.

§ 2.3. Generating Compositions

convention, we can now proceed with our review of all known composition generation algorithms — ascending, descending or otherwise.

2.3 Generating Compositions

In Section 2.3.1 we discuss the particular details of the in-memory representation of compositions, and stress the distinction between this and *encoding* for partitions. Then, in Section 2.3.2, we review the various orderings that have been used in generating classes of composition. Section 2.3.3 then begins the actual review of composition generation algorithms, where we consider the algorithms that generate ‘unrestricted’ classes of composition. In Section 2.3.4 we then examine the algorithms that generate some class of restricted composition, and make the distinction between ‘local’ and ‘global’ restrictions. Finally, in Section 2.3.5, we close this review of composition generation algorithms by briefly examining some algorithms of more tangential interest.

2.3.1 Representations

In this section we review algorithms that generate some subset of the compositions of n . For particular classes of composition certain representations are appropriate, which we shall briefly examine in this subsection. It is important to note that the term ‘representation’ is distinct from our usage of ‘encoding’. The encodings we are interested in are those that encode the partitions of n in terms of some subset of the compositions of n . The term ‘representation’, on the other hand, refers to the actual representation format of the compositions in computer memory. The distinction is admittedly a confusing one and there is no rubicon between encoding on one hand and representation on the other. Nevertheless, we shall persist with this distinction with the understanding that when we speak of ‘encoding’ it refers to high-level conceptual issues concerning the objects we are generating, and when we speak of a ‘representation’ it is in terms of implementing a particular encoding as a concrete generation algorithm.

§ 2.3. Generating Compositions

When generating the unrestricted compositions of n , the direct ‘sequence’ representation is most often used; to represent a composition $a_1 + \dots + a_k = n$ we simply store the sequence $a_1 \dots a_k$ of its parts. (This representation is also known as the ‘signature’ representation by Lehmer [Leh64] and Wells [Wel71].) Descending composition generators, on the other hand, use one of three representations: the sequence representation, the ‘multiplicity’ representation or the ‘part-count’ representation. The sequence representation for descending compositions is precisely the same as for unrestricted compositions: we store a composition $d_1 + \dots + d_k = n$ with $d_1 \geq \dots \geq d_k$ as the sequence $d_1 \dots d_k$. The multiplicity and part-count representations both utilise the fact that equal parts are contiguous in descending compositions, although they do it in slightly different ways.

In the multiplicity representation, the descending composition $d_1 + \dots + d_k = n$ is stored as two separate sequences of positive integers, $\delta_1 \dots \delta_p$ and $m_1 \dots m_p$, where p is the number of distinct parts in $d_1 \dots d_k$. As $\delta_1 > \dots > \delta_p$ we have $\delta_1 = d_1$ and $\delta_p = d_k$, and m_j is the number of times part δ_j occurs in $d_1 \dots d_k$. For example, in the multiplicity representation the descending composition $d_1 \dots d_5 = 55331$ will be stored as the sequences $\delta_1 \dots \delta_3 = 531$ and $m_1 \dots m_3 = 221$: there are two parts of size 5, two parts of size 3 and one part of size 1.

In the part-count representation we regard a descending composition of n as an n -tuple of nonnegative integers $c_1 \dots c_n$, where $n = c_1 + 2c_2 + \dots + nc_n$; c_1 counts the number of 1s in the partition, c_2 the number of 2s and so on [Knu04c, p.3]. For instance, the descending composition 4211 becomes the 8-tuple 21030000, since there are two parts of size 1, one part of size 2, zero parts of size 3, etc. The part-count representation has at least one useful application. Fàa Di Bruno’s formula [Yan00] calculates the n th order derivative of a composite function and requires as input all partitions of n in part-count form [Kli73]. (The part-count representation is perhaps more correctly regarded as an encoding. To do so would further complicate our later discussions, as the concept of ordering becomes unclear, and so we shall continue to refer to it as a representation.)

Note that both the part-count and multiplicity representations, although

§ 2.3. Generating Compositions

presented in terms of descending compositions, can equally be applied to ascending compositions. We have phrased the explanations here in terms of descending compositions because all of the existing algorithms that utilise these representations generate descending compositions.

2.3.2 Generation Order

Unrestricted compositions are most commonly generated in lexicographic order, but there is another ordering to be considered. Since there is a direct correspondence between the unrestricted compositions of n and binary $(n - 1)$ -tuples, algorithms can be derived to generate compositions in orders corresponding to the various orders defined over binary n -tuples. For example, Knuth [Knu04b, ex.12] provides an algorithm to generate unrestricted compositions in an order corresponding to the binary-reflected Gray code [Knu04b, p.3]. The order that arises for the compositions of 3 is

$$3(00), 21(01), 111(11), 12(10), \tag{2.1}$$

where each composition is paired with its corresponding bit-string in the difference representation in parentheses.

Several generation orders have been suggested in the case of descending compositions. In Figure 2.1 we can see the descending compositions of 5 arranged in a binary tree. The root of this tree is the composition 11111, and each node represents a unique descending composition of 5. The left child of a given node is produced by replacing two parts of size 1 by one part of size 2 (e.g., the left child of 311 is 32), and exists if and only if the descending composition has at least two parts of size 1. The right child of a given node is derived by removing one part of size 1 and increasing the smallest non-1 part by 1 (e.g., the right child of 311 is 41), and exists if and only if there is at least one 1 and there is exactly one copy of the smallest part greater than 1. (See Fenner & Loizou [FL80, FL83] or Knuth [Knu04c, ex.10] for a complete discussion of these operations and further properties of this tree representation of the set of descending compositions.) Fenner & Loizou [FL83] use the resulting binary tree to define three orders over

§ 2.3. Generating Compositions

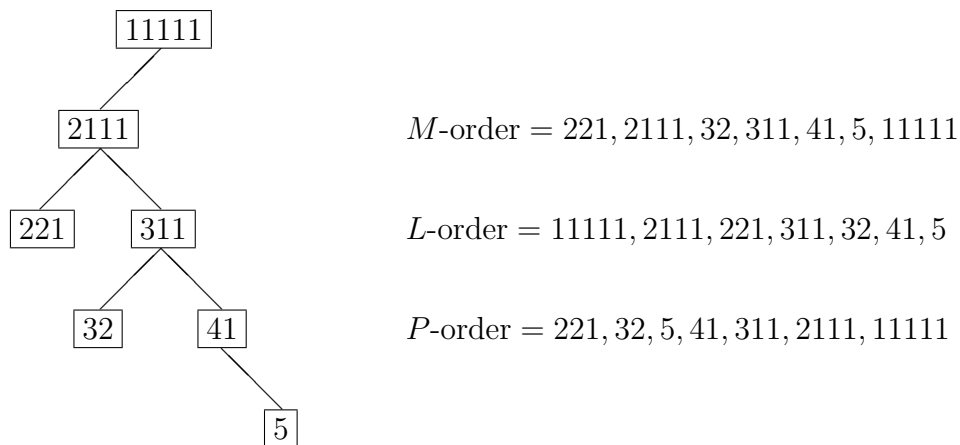


Figure 2.1: Fenner & Loizou’s binary tree representation of the set of descending compositions of 5, and the orders corresponding to a inorder, preorder and postorder traversal of this tree.

the set of descending compositions, corresponding to preorder, inorder, and postorder traversals of the tree. These orders are referred to L -order, M -order and P -order respectively, and the descending compositions of 5 in these orders are given in Figure 2.1. L -order corresponds to lexicographic order. Furthermore, we can consider the reverse of each of the orders associated with the traversal of Fenner & Loizou’s tree, giving us a total of six orders in which we can generate descending compositions derived from this tree.

In Savage’s [Sav89] Gray code order for descending compositions, the succeeding composition is generated by adding 1 to one part and subtracting 1 from another. For example, the descending compositions of 5 in this minimal-change order are

$$11111, 2111, 311, 221, 32, 41, 5. \tag{2.2}$$

(See also Knuth [Knu04c, p.15–17].) Subsequent work [RSW95] extended the original Gray code to operate on restricted classes of descending composition, where all parts are distinct, or all parts satisfy certain congruence conditions.

2.3.3 Unrestricted Generators

In this subsection we shall consider the algorithms in the literature to generate the unrestricted compositions, ascending compositions and descending compositions of a positive integer n . Many algorithms have been suggested, which we shall attempt to categorise. Algorithms are discussed under the headings of ‘recursive’ and ‘iterative’, as these strategies result in broadly different generation algorithms, each with particular advantages and disadvantages. The algorithms examined here are ‘unrestricted’ in that they can generate the entire set of compositions of interest if suitably invoked. It may be possible to enforce further restrictions on the compositions visited; we consider the problem of generating compositions with restrictions placed on the parts in the next subsection.

Recursive Generators

The best-known recursive composition generation technique is Page & Wilson’s descending composition generation algorithm [PW79, §5.5], and the closely related k -composition generator [PW79, §5.4]. Page & Wilson’s algorithm operates by noting that a descending composition with largest part at most m either begins with m or it does not. Those that begin with m can be constructed by prepending m to all descending compositions of $n - m$ with largest part at most m . The descending compositions that do not contain m are the descending compositions of n with largest part at most $m - 1$ [Ski90, p.51]. Page & Wilson’s algorithms avoid the cost of generating and storing large sets of compositions during generation by using a single array to store the compositions involved. At each level of recursion an assignment is made to a particular array index, effectively prepending that value to all compositions generated by subsequent levels of recursion. Variants of this algorithm have appeared in several texts, including Kreher & Stinson [KS98, p.68], Skiena [Ski90, p.51] and Pemmaraju & Skiena [PS03, p.136]. Page & Wilson’s descending composition generator is, however, not constant amortised time. Ruskey [Rus01, §4.8] improves Page & Wilson’s basic algorithm by applying ‘path elimination techniques’, and the resulting algorithm is constant

§ 2.3. Generating Compositions

amortised time. We study Ruskey’s algorithm in detail in Section 5.2.2.

The only known means of generating all ascending compositions in lexicographic order is a recursive algorithm due to de Moivre [dM97]. In de Moivre’s method we generate the ascending compositions of n by prepending m to all ascending compositions of $n - m$, for $m = 1, \dots, n$ [Knu04f, p.20]. Boyer’s [Boy05] algorithm also generates ascending compositions, but it generates compositions into a fixed number of parts (that is, k -compositions). Boyer’s algorithm works on similar principles to Page & Wilson’s, and generates ascending k -compositions in constant amortised time.

Fenner & Loizou’s tree construction operations [FL80] can also be used to recursively generate descending compositions in lexicographic order (or any of the other five orders associated with Figure 2.1). Savage’s [Sav89] Gray code construction can be implemented recursively to generate descending k -compositions in minimal-change order. Tomasi [Tom82] has defined two recursive algorithms to generate all descending k -compositions. Both of these algorithms, unfortunately, are thoroughly inefficient as they require the computation, storage and subsequent reprocessing of large sets of compositions.

Iterative Generators

The unrestricted compositions of n are in one-to-one correspondence with the binary $(n - 1)$ -tuples; consequently, there are many different approaches and orders in which we can generate unrestricted compositions. Directly translating binary $(n - 1)$ -tuples into compositions in the sequence representation requires a linear pass through each tuple generated, and so a naive approach — where we generate all binary $(n - 1)$ -tuples and compute the corresponding composition for each tuple — will be inefficient. We can, however, convert the original $(n - 1)$ -tuple generation algorithm to directly output compositions in the form that we require. Knuth [Knu04b, ex.12] gives an example of this approach, where an algorithm to generate n -tuples in binary-reflected Gray code order is modified to produce unrestricted compositions in the sequence representation. Lehmer [Leh64, §1.7], Wells [Wel71, p.147], and Page & Wil-

§ 2.3. Generating Compositions

son [PW79, §5.4] proposed an algorithm to generate all compositions based on the difference representation. This algorithm involves converting each integer from 0 to $2^{n-1} - 1$ to its corresponding composition in the sequence representation, and is therefore more correctly regarded as an unranking algorithm. The only other known approach to generating all compositions of n is Nārāyaṇa’s fourteenth century algorithm [Knu04f, ex.15], which generates compositions in reverse lexicographic order.

Ehrlich [Ehr73a, Ehr73b] modified a composition generation algorithm to produce a loopless k -composition generation algorithm. Two other iterative algorithms have been published to generate unrestricted k -compositions and both output compositions in lexicographic order. Hellerman & Ogden’s algorithm [HO61] sequentially transforms the current k -composition into its lexicographic successor until the final composition has been found. The algorithm provided by Wells [Wel71, p.145] operates on similar principles.

It is widely accepted that the most efficient means of generating descending compositions is in reverse lexicographic order: see Andrews [And76, p.230], Knuth [Knu04c, p.1], Nijenhuis & Wilf [NW78, p.65–68], Page & Wilson [PW79, §5.5], Skiena [Ski90, p.52], Stanton & White [SW86, p.13], Wells [Wel71, p.150] or Zoghbi & Stojmenović [ZS98]. McKay [McK70] refers to reverse lexicographic order as the “natural order” for descending compositions. Many algorithms have been suggested to generate descending compositions in reverse lexicographic order, and all are based on the following succession rule. Given a descending composition $d_1 \dots d_k$, we let q be the index of the smallest part greater than 1 (thus, $d_j = 1$ for $q < j \leq k$). Then, letting $d_q = x + 1$, we obtain the next descending composition in reverse lexicographic order by replacing the suffix $(x + 1)1 \dots 1$ by $x \dots xr$ for the appropriate remainder $r \leq x$ [Knu04c, p.1]. (We shall study this succession rule in detail in Section 5.3.2.) Many implementations of this particular succession rule have been published in the sequence (e.g. McKay [McK70]), multiplicity (e.g. Nijenhuis & Wilf [NW78, ch.9]) and part-count [Sto62b] representations. Recently, Zoghbi & Stojmenović [ZS98] have improved on the complexity of a direct implementation of this rule. By observing that when $d_q = 2$ we can implement the transition by setting $d_q = 1$ and ap-

§ 2.3. Generating Compositions

pending an extra 1 to the end of the composition, Zoghbi & Stojmenović's provided a constant amortised time implementation of the succession rule. (We shall study Zoghbi & Stojmenović's algorithm in detail in Section 5.4.2.) Knuth [Knu04c, p.2] uses the same idea, and provides an analysis of the resulting algorithm [Knu04c, p.13].

This succession rule can be reversed to provide us with an algorithm to generate all descending compositions of n in lexicographic order, and several such algorithms have been published. Knuth [Knu94, p.147] and Zoghbi & Stojmenović [ZS98] provide algorithms to generate descending compositions in lexicographic order in the sequence representation. Reingold, Nievergelt & Deo [RND77, p.193] and Fenner & Loizou [FL81] describe algorithms using the multiplicity representation. Fenner & Loizou thoroughly analyse algorithms to generate descending compositions in the multiplicity representation in lexicographic and reverse lexicographic order [FL81], and compare the total computation required for both orders. Klimko [Kli73] describes an algorithm to generate descending compositions in lexicographic order using the part-count representation (see also Knuth [Knu04c, ex.5]).

Algorithms implementing the lexicographic succession rules for descending compositions in the multiplicity or part-count representations can be implemented looplessly [Ehr73b]. This is because we can directly update the multiplicities of individual parts without needing to iterate to make copies of a particular part. In practice, these algorithms tend to be less efficient than their sequence representation counterparts, as a large cost can be incurred by the extra multiplications implied by this approach [Knu04c, ex.5]. In an empirical analysis, Zoghbi & Stojmenović [ZS98] demonstrated that their sequence representation algorithms are significantly more efficient than all known multiplicity and part-count representation algorithms.

Algorithms to generate descending k -compositions have also been published. Gupta, Lee & Wong [GLW83] provide an algorithm to generate descending k -compositions in lexicographic order as part of their methods for ranking, unranking and generating B-trees. Riha & James [RJ76] provide algorithms to generate descending k -compositions in reverse lexicographic order, where an upper and lower bound on part values can be specified, or

§ 2.3. Generating Compositions

parts may be drawn from some extensionally defined set. Neither Gupta, Lee & Wong's nor Riha & James' algorithms generate descending k -compositions in constant amortised time.

Several iterative algorithms exist to generate ascending k -compositions. The best known and most widely cited of these is Hindenburg's [Dic52, p.106][Knu04f, p.21] eighteenth century algorithm. Hindenburg's algorithm generates ascending k -compositions in lexicographic order (as do all other published ascending k -composition generators) and is regarded as the canonical method to generate partitions into a fixed number of parts. Using Hindenburg's algorithm, the successor of a given ascending k -composition $a_1 \dots a_k$ is found by scanning from right-to-left, stopping at the rightmost a_t such that $a_k - a_t \geq 2$. We then replace a_j by $a_j + 1$ for $t \leq j < k$ and replace a_k by the remainder, ensuring that $a_1 + \dots + a_k = n$. See Knuth [Knu04c, p.2], Andrews [And76, p.232] or Reingold, Nievergelt & Deo [RND77, p.191] for a description of this algorithm and Knuth [Knu04c, p.14–15] for an analysis of the algorithm implemented in the sequence representation. The algorithm due to Narayana, Mathsen & Sarangi [NMS71] generates ascending k -compositions, and using Stockmal's algorithm [Sto62a] we can stipulate an upper bound on part values.

There has been some recent interest in generating compositions in parallel. Akl & Stojmenović [AS93] provide a range of algorithms to generate both descending compositions and unrestricted compositions in parallel. Akl & Stojmenović modify the generation algorithms discussed above to operate in a parallel context, and devise efficiency measures suitable for parallel generation.

In summary, many algorithms exist to generate a variety of classes of composition in both lexicographic and reverse lexicographic order. All known algorithms to generate unrestricted compositions, descending compositions and ascending compositions are given in Table 2.2. Algorithms are classified according to whether they generate compositions into either a fixed or arbitrary number of parts. Each algorithm is associated with the relevant information: the representation used; whether the algorithm is iterative or recursive; the order in which compositions are generated; and whether the

§ 2.3. Generating Compositions

algorithm has been proved to generate compositions in constant amortised time. Some clear patterns emerge from this table, which we shall discuss in the conclusion of this chapter.

2.3.4 Restricted Generators

In this subsection we are concerned with *restricted* generation algorithms: algorithms that efficiently generate some well-specified subset of the six classes of composition identified in Table 2.2, i.e., the unrestricted, ascending and descending compositions and k -compositions. All of the algorithms discussed in the previous section are said to be ‘unrestricted’ in that they generate complete sets of compositions when suitably invoked. Some of these algorithms allow us to stipulate some further restrictions on the parts, and we shall discuss these algorithms again where necessary.

We consider here only restrictions that are not trivial to impose. If, for example, we are generating descending compositions in reverse lexicographic order, we can easily restrict the value of the largest part. We can, for instance, ensure that the largest part is no greater than m by first computing the lexicographically largest descending composition $d_1 \dots d_k$ of n such that $d_1 = m$ and iteratively apply the succession rule discussed on page 32. Similarly, when we are dealing with ascending compositions, it is trivial to restrict the value of the *smallest* part in the corresponding partitions. It is also usually quite simple to modify algorithms to ensure that only compositions with a maximum number of parts are visited.

We shall discuss restricted composition generation under two headings: ‘local’ and ‘global’ restrictions. Local restrictions are defined as specifying restrictions on individual parts, for example each part is at least m , or each part is odd, or each part is an element of some pre-specified set. Global restrictions, on the other hand, restrict the *entire* composition, and may require complex interrelationships between parts. An example of a global restriction is the requirement that all parts be distinct: this is not a local restriction because the restriction is only defined in terms of the other parts.

§ 2.3. Generating Compositions

Local Restrictions

An algorithm enforces a local restriction on generated compositions if we can specify some bounds or congruence conditions on each part. Several algorithms exist to enforce magnitude restrictions on ‘unrestricted’ compositions. Nārāyaṇa’s algorithm [Knu04f, ex.15] allows us to generate compositions where all parts are no greater than some specified value. Ehrlich [Ehr73a, Ehr73b] provides an algorithm to generate all k -compositions lower-bounded by a sequence of values $l_1 \dots l_k$: for all k -compositions $a_1 \dots a_k$ visited by Ehrlich’s algorithm, we will have $l_1 \leq a_1, \dots, l_k \leq a_k$. An algorithm given by Wells [Wel71, p.145] allows us to generate k -compositions with a stipulated upper and lower bound on part values.

Algorithms due to Riha & James [RJ76] allow us to generate descending k -compositions with a flexible set of restrictions. The first algorithm is a method to generate descending k -compositions with an upper and lower bound on part values (as well as specifying a minimum interpart distance — we discuss this algorithm further under the heading of global restrictions later in this subsection). The second algorithm developed by Riha & James generates descending k -compositions where the parts must belong to some extensionally defined set. The algorithm also allows us to restrict the number of times a particular value appears in compositions. A slightly different version of this problem has been studied by Horowitz & Sahni [HS74], and was subsequently expanded on by Rubin [Rub76]. In this problem we are given a sequence $s_1 \dots s_r$ of values and we must generate all combinations of values from this sequence that sum to n . More precisely, the algorithms generate all sequences $b_1 \dots b_r$, where each $b_j \in \{0, 1\}$, such that $b_1 s_1 + \dots + b_r s_r = n$. Horowitz and Sahni propose a number of solutions based on backtracking search and examine applications to the knapsack problem. Rubin [Rub76] criticises some of the heuristics adopted by Horowitz & Sahni, and provides a number of alternatives which are evaluated empirically.

The algorithms of Riha & James [RJ76], Horowitz & Sahni [HS74] and Rubin [Rub76] are very general but do not generate compositions in constant amortised time. More efficient approaches exist for some specialised

§ 2.3. Generating Compositions

local restrictions. Binary descending compositions (i.e., where all parts are powers of 2) can be generated looplessly [Knu04c, ex.64] using a Gray code sequence [CK05] in which each step replaces a part of size $2^j + 2^j$ by one of size 2^{j+1} , or vice-versa. Bhatt provides a method to generate descending compositions in which all parts are $2^j - 1$ (e.g. 7311 has the required form), and demonstrates the applications of such partitions to cryptographic problems [Bha99]. Boscovich [Knu04f, ex.30] described an algorithm to generate descending compositions where all parts are 1, 7 or 10. Rasmussen, Savage & West [RSW95] developed a Gray code construction over descending compositions satisfying certain congruence conditions which can be implemented to generate, for example, descending compositions into strictly odd parts. Ruskey also provides an algorithm to generate descending compositions into odd parts [Rus05].

Global Restrictions

Local restrictions allow us to specify restrictions on *individual* parts of compositions, but we can also specify restrictions that apply to *all* parts. Global restrictions specify some required relationship between all parts in compositions, and there are many possible restrictions of this type. As an example, Coleman & Taylor's algorithm [CT71] generates k -compositions that are equivalence classes under cyclic rotation, which they refer to as 'circular' compositions. The circular 3-compositions of 8 are

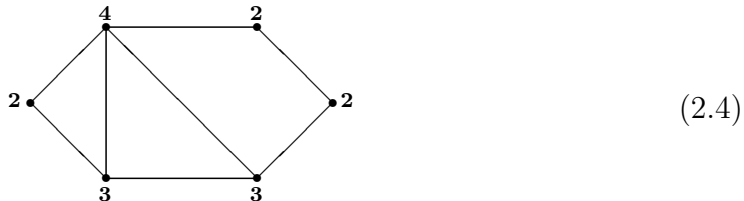
$$116, 125, 134, 143, 152, 224, 233. \tag{2.3}$$

The subset of the 3-compositions of 8 in this example are not specified in any way by the magnitude of the individual parts. They are restricted, rather, by a required relationship between parts. Each composition in this example is an equivalence class under circular rotation; thus, only the composition 116, and not 611 or 161, appears in (2.3) as they are all equivalent under circular rotation.

Several varieties of global restriction relating to graphs have been studied. A descending composition $d_1 \dots d_k$ is said to be graphical if it is the degree

§ 2.3. Generating Compositions

sequence of some simple graph (i.e., a graph that can be drawn without multiple edges or loops). As the sum of the degrees of the vertices in an undirected graph is equal to twice the number of edges, a necessary condition for a composition of n to be graphical is that n is even [NSST98]. For example, the composition 433222 is graphical, as it is the degree sequence of the graph



in which each node is labelled with its degree. The problem of generating all the graphical partitions, or more precisely, the graphical descending compositions, has been studied. Barnes & Savage [BS97] give a constant amortised time algorithm to generate all graphical partitions. Nolan et al. [NSST98] provide an algorithm to generate all graphical basis partitions, which are the graphical partitions that are also basic. (Each basis partition is uniquely identified by a given rank vector, a quantity relating to the conjugate of a given partition and its Durfee square — see Nolan, Savage & Wilf [NSW98] for information on these partitions and the aforementioned concepts.) Ruskey et al. [RECS94] present an algorithm to generate all possible degree sequences of length n and James & Riha [JR76] provide an algorithm to generate all of the graphs which correspond to a given graphical partition (for example, (2.4) is not the only graph corresponding to the degree sequence 433222).

Another class of global restriction related to graphs has been studied. A round-robin tournament can be seen as a complete directed graph [RECS94]. A sequence $a_1 \dots a_k$ is a score vector of a round-robin tournament if $a_1 \leq \dots \leq a_k$, $a_1 + \dots + a_j \geq \binom{j}{2}$ for $1 \leq j < k$ and $a_1 + \dots + a_k = \binom{k}{2}$ (note that in this instance, the restriction has been defined on ascending compositions). Narayana, Mathsen & Sarangi [NMS71] provide an algorithm to generate all score vectors. Ruskey et al. [RECS94] provide two simple recursive algorithms to generate round-robin tournament score vectors, both of which have empirically observed constant amortised time performance.

§ 2.3. Generating Compositions

Round-robin tournament score vectors may also be computed as a special case of an algorithm that computes all descending compositions majorised by a given descending composition and which majorise another [Knu04c, ex.56]. (A descending composition $d_1 \dots d_k$ majorises the descending composition $e_1 \dots e_l$ if $d_1 + \dots + d_j \geq e_1 + \dots + e_j$ for all $j \geq 0$ [Knu04c, ex.54]. A related concept is the lattice of integer partitions [Bry73, LP00].)

A number of techniques have been developed to generate ascending and descending compositions into distinct parts (that is, where all parts in the compositions are different). Boyer [Boy05] demonstrates the simple modifications required to his ascending k -composition generator to ensure that all parts are distinct. Furthermore, these modifications preserve the constant amortised time property of Boyer's algorithm. Rasmussen, Savage & West's Gray code construction [RSW95] can be implemented to generate descending compositions with distinct parts. Using Riha & James's algorithm [RJ76] to generate descending k -compositions into prescribed parts, we can generate descending k -compositions into distinct prescribed parts, by limiting the number of occurrences of each part to 1. Similarly, we can use Horowitz & Sahni's [HS74] or Rubin's [Rub76] algorithms to generate compositions into distinct parts by suitably invoking the algorithms. Ruskey also provides a non constant amortised time algorithm to generate descending compositions into distinct odd parts [Rus05].

Riha & James's algorithm [RJ76] to generate bounded descending k -compositions can also be used to generate descending k -compositions into distinct parts. More generally, it allows us to specify a minimum interpart distance δ such that for each descending k -composition $d_1 \dots d_k$ visited we will have $d_j - \delta \geq d_{j+1}$ for $1 \leq j < k$. Thus, by setting $\delta = 1$ we obtain descending k -compositions such that $d_1 > \dots > d_k$.

2.3.5 Miscellaneous Algorithms

In this subsection we review some algorithms of more tangential interest to this dissertation. Some of the algorithms mentioned are not true generation algorithms and some are generation algorithms but not for compositions of in-

§ 2.3. Generating Compositions

tegers. We first consider ranking/unranking and random selection algorithms (see Section 2.1.4) for various classes of composition. We then summarise some algorithms for objects closely related to partitions and compositions, and finish by briefly reviewing the literature on weak k -compositions (i.e. compositions into nonnegative parts).

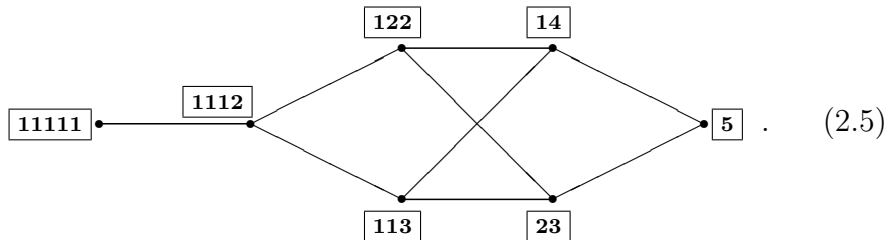
Many ranking and unranking algorithms exist for varieties of composition. In the case of the unrestricted compositions and k -compositions the problem is easily solved because convenient enumeration formulas exist, and is an easy application of more general techniques [NW78, ch.13]. (See Lehmer [Leh64, §1.7], Wells [Wel71, p.147] or Page & Wilson [PW79, §5.4] for an unranking algorithm for compositions based on the binary-difference representation.) Ranking and unranking algorithms are also known for descending compositions. McKay developed algorithms to rank [McK65c] and unrank [McK65b] descending compositions. White generalised McKay's work to give an algorithm to unrank descending compositions with a lower bound on part values [Whi70b]. Several texts provide methods to rank and unrank descending compositions: see Wells [Wel71, p.151], Stanton & White [SW86, p.14] and Kreher & Stinson [KS98, p.77–78].

Random selection algorithms for unrestricted compositions are also easily obtained from general techniques [NW75] because of the existence of simple enumeration formulas. Nijenhuis & Wilf's algorithm [NW78, ch.10] for the random selection of partitions remains the canonical method — see Reingold, Nievergelt & Deo [RND77, p.193–196], Skiena [Ski90, p.58] or Knuth [Knu04c, ex.47] for implementations of the algorithm.

Nijenhuis & Wilf [NW78, ch.14] provide algorithms to randomly select and generate Young Tableaux; Bratley & McKay [BM67] present an algorithm to generate multi-dimensional partitions [And76, ch.11]. Knuth provides an algorithm to generate partition graphs in the *Stanford Graphbase* [Knu94, p.145–149]. A partition graph (not to be confused with graphical partitions, see Section 2.3.4) is a graph of a *set* of partitions, where each partition is a node in the graph. Two nodes in this graph are adjacent if one can be obtained by combining two parts in the other (each arc in the graph

§ 2.4. Summary

is of length 1). For example, the partition graph of 5 is



We can see in (2.5) that 113 and 122 are both adjacent to 23, as we can obtain 23 by either adding together two 1s in 113 or adding a 1 and a 2 in 122. For further information see Knuth [Knu94, p.35,145].

While weak k -compositions are not of direct relevance to this dissertation, we can quickly summarise the available algorithms in the interest of completeness. Nijenhuis & Wilf provide algorithms to efficiently generate [NW78, ch.5] and randomly select [NW78, ch.6] weak k -compositions. Skiena [Ski90, p.61] and Knuth [Knu04a, ex.3] also provide algorithms to generate weak k -compositions. Klingsberg [Kli82] (see also Wilf [Wil89]) developed a Gray code for weak k -compositions, which Walsh [Wal00] has extended to weak k -compositions with bounds on the part values (see also Knuth [Knu04a, ex.59–62]). Bitner, Ehrlich & Reingold [BER76] consider the generation of weak k -compositions as an application of their algorithm to generate binary n -tuples with exactly k 1-bits in the binary reflected Gray code order. Knuth [Knu94, p.127–128] provides an algorithm to generate all weak k -compositions of n where all parts are square, i.e., all solutions of $a_1^2 + \dots + a_k^2 = n$ where each a_j is nonnegative. This problem occurs in the context of constructing a graph based on the moves of generalised chess pieces on a generalised chessboard [Knu94, p.122].

2.4 Summary

There are a number of trends we should note from the literature that are important to contextualise the remainder of this dissertation which we shall attempt to illustrate here. The first trend of note is that almost all known

§ 2.4. Summary

‘partition’ generation algorithms generate *descending compositions*. As we noted in Section 2.2.3, descending composition are only one possible *encoding* for partitions. Although ascending compositions are an equally valid encoding, the possibility has not been considered because of the tendency in the literature to *literally* define partitions as descending compositions, as we saw in Section 2.2.2. Thus, we can see that although there is a large number of different ‘partition’ generation algorithms, by sharing the common property of generating descending compositions, an equally large space of possible algorithm designs has been largely uninvestigated, as illustrated in Table 2.2. In chapters 4 and 5 we perform the first systematic investigation of algorithms drawn from this design space.

The second trend from the literature we would like to call attention to here is the lack of a unified approach to specifying and generating restricted partitions. Ad hoc techniques exist to generate partitions with both local and global restrictions. (Recall that a ‘local’ restriction is defined as a restriction placed on individual parts, whereas ‘global’ restrictions are enforced on all parts.) With the exception of Riha & James’ algorithms [RJ76] which are not constant amortised time, there has been no attempt made to generate a class of restricted partition. In the next chapter we begin the task of redressing this shortcoming in the literature by introducing a coherent framework within which we can express a wide variety of global restrictions.

Chapter 3

Interpart Restricted Compositions

In Section 3.1 we define the interpart restricted compositions framework and demonstrate, via examples from the combinatorial literature, how this framework can be instantiated and the types of restriction we can impose on partitions. In Section 3.2 we develop the first concrete application of this framework and provide a simple, general means of counting interpart restricted compositions. Finally, in Section 3.3 we summarise the content of this chapter.

3.1 An Algorithmic Framework

In this section we develop an algorithmic framework for generating partitions with a flexible class of restrictions placed on the parts. For reasons that will become apparent, we refer to this framework and the objects defined under its formalisms as ‘interpart restricted compositions’. Several other generic approaches to defining classes of restricted partition and composition have been proposed, each of which is unsuitable for our purposes [MM05, BBGP04, CS04, BC05]. The framework proposed by Martínez & Molinero [MM05] provides a general means of generating combinatorial classes including trees, permutations and set partitions. As such, their frame-

§ 3.1. An Algorithmic Framework

work is more general than we require since we wish to defend a particular thesis specific to classes of partition. Similarly, the specifications of combinatorial objects used by Bacchelli et al. [BBGP04] are not limited to partitions and compositions and are therefore more general than we require. The means of specifying general restrictions on partitions and compositions proposed by Corteel & Savage explicitly encode partitions as descending compositions [CS04]. Since we are defending the thesis that it is more difficult to generate all descending compositions than it is to generate all ascending compositions, we cannot base our general algorithms on their framework. Finally, Bender & Canfield's framework [BC05] allows us to specify a flexible class of restriction on compositions but is not immediately amenable to defining classes of restriction on *partitions*, and therefore also falls outside the scope of this dissertation.

Interpart restricted compositions allow us to easily specify classes of restricted partition. The restrictions are defined by an integer function which provides us with both a concise and efficiently implementable representation format. The framework is designed specifically with the efficiency of generation in mind and, as we shall see in the next chapter, the algorithms that arise are both simple and efficient. Before we define generation algorithms we must develop a concrete foundation for the framework and demonstrate the type of restriction that we may impose.

In Section 3.1.1 we develop the basic definitions required for the framework, and establish the notational conventions we shall be utilising for the remainder of this dissertation. Then, in Section 3.1.2 we develop the fundamental results that are the basis of all our subsequent work with interpart restricted compositions. Finally, in Section 3.1.3 we demonstrate how the framework may be instantiated to describe important classes of restricted partition from the literature.

3.1.1 Definitions and Notational Conventions

In this subsection we shall define the fundamental concepts necessary for the interpart restricted compositions framework. Before defining the concepts

§ 3.1. An Algorithmic Framework

in question we discuss the notational and terminological conventions used in the remainder of the dissertation. The most fundamental notation we require concerns sequences of integers and as we shall be performing many operations over sequences we require some specific notation.

Ordinarily, we denote a sequence of integers as $a_1 \dots a_k$, which denotes a sequence of k integers indexed a_1, a_2 , etc. When referring to short specific sequences it is convenient to enclose each element using \langle and \rangle . Thus, if we let $a_1 \dots a_k = \langle 3 \rangle \langle 23 \rangle$, we have $k = 2$, $a_1 = 3$ and $a_2 = 23$. We will also use the idea of prepending a particular value to the head of a sequence: thus, the notation $3 \cdot \langle 23 \rangle$ is the same sequence as given in the preceding example.

In defining the classes of partition and composition which we are interested in we shall often use Iversonian brackets [GKP94, p.24], as this technique allows us state some of the restrictions we wish to impose very concisely. An Iversonian condition consists of some logical statement \mathbf{s} encapsulated in brackets. Then, if \mathbf{s} is true, $[\mathbf{s}] = 1$ and if \mathbf{s} is false, $[\mathbf{s}] = 0$. For example, $[x \text{ even}] = 0$ and $[x \text{ prime}] = 1$ when $x = 3$.

Other notational devices will be utilised throughout the dissertation, which we shall introduce where necessary. We begin the development of interpart restricted compositions by formally defining the basic property of interpart restriction and then define the required enumeration and set functions. We then define the concept of initial part feasibility, providing us with sufficient formal concepts to develop the basic theory of interpart restricted compositions.

Definition 3.1 (Composition). *A sequence of positive integers $a_1 \dots a_k$ is a composition of the positive integer n if $a_1 + \dots + a_k = n$. A composition $a_1 \dots a_k$ with a fixed number of parts k is known as a k -composition.*

Definition 3.2 (Interpart Restricted Sequence). *A sequence of positive integers $a_1 \dots a_k$ is interpart restricted by the function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ if $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$.*

Then, combining Definitions 3.1 and 3.2, we can see that a composition $a_1 \dots a_k$ of n is interpart restricted by the function σ (or equivalently, $a_1 \dots a_k$ is an interpart restricted composition of n and σ) if the sum of all the parts

§ 3.1. An Algorithmic Framework

is n , and $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$. This definition is the key definition of the interpart restricted compositions framework, and specifies a set of $k - 1$ inequalities on each composition with k parts. Each inequality states that for a given part a_j , the value $\sigma(a_j)$ must be less than or equal to the part a_{j+1} . This system of $k - 1$ inequalities can be viewed figuratively as follows:

$$\begin{aligned} \sigma(a_1) &\leq a_2 \\ \sigma(a_2) &\leq a_3 \\ &\vdots \\ \sigma(a_{k-1}) &\leq a_k \end{aligned} \tag{3.1}$$

As we shall see in the next section, this model of embedding an integer function into a system of inequalities can represent many classes of restricted partition from the literature.

We shall also regularly refer to the set of all compositions of n interpart restricted by σ and, correspondingly, the number of compositions of n interpart restricted by σ . Formally, we have the following definitions.

Definition 3.3 (Set of Interpart Restricted Compositions). *For some positive integer n and a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, let \mathcal{C}_σ be a function such that $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$ iff $a_1 \dots a_k$ is composition of n which is interpart restricted by σ .*

Determining if a given sequence $a_1 \dots a_k$ is an element of the set $\mathcal{C}_\sigma(n)$ is an important problem, and so we shall codify the requirements of Definition 3.3 into three necessary and sufficient conditions:

1. $a_j \in \mathbb{Z}^+$ for $1 \leq j \leq k$;
2. $a_1 + \dots + a_k = n$;
3. $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$.

Thus, if all three of these conditions hold true, we know that a given sequence must be an element of the set $\mathcal{C}_\sigma(n)$.

§ 3.1. An Algorithmic Framework

Definition 3.4 (Number of Interpart Restricted Compositions). *For some positive integer n and a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, let C_σ be a function such that $C_\sigma(n) = |\mathcal{C}_\sigma(n)|$.*

Thus, the function $\mathcal{C}_\sigma(n)$ is defined as the set of all compositions of n interpart restricted by σ and $C_\sigma(n)$ is the number of compositions of n interpart restricted by σ .

Given a sequence $a_1 \dots a_k$, its *initial part* is a_1 , and so every nonempty sequence has an initial part. For computational purposes we require an overloaded version of set and enumeration functions in which we restrict the value of the initial part of all compositions $a_1 \dots a_k$ in $\mathcal{C}_\sigma(n)$. Thus, we let $\mathcal{C}_\sigma(n, m)$ be the set of all compositions of n interpart restricted by σ where the initial part is at least m . Formally,

$$\mathcal{C}_\sigma(n, m) = \{a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 \geq m\},$$

and then define $C_\sigma(n, m)$ as the number of compositions of n interpart restricted by σ where the initial part is at least m , i.e. $C_\sigma(n, m) = |\mathcal{C}_\sigma(n, m)|$.

The final formal device we require is the concept of *initial part feasibility*, which we shall define in a specific technical sense.

Definition 3.5 (Feasible Initial Part). *Let n be a positive integer and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be some function. A positive integer x is a feasible initial part for n and σ if and only if there exists some $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$ such that $a_1 = x$.*

Thus, a positive integer x is a feasible initial part for n and σ if and only if there is some composition of n interpart restricted by σ such that the initial part is equal to x . Correspondingly, a positive integer x is an *infeasible initial part* for n and σ if and only if there is no $a_1 \dots a_k$ in the set $\mathcal{C}_\sigma(n)$ such that $a_1 = x$. Determining the feasibility of a given x for some n and σ is an important aspect of both our enumeration and generation techniques. Intuitively, if we can efficiently determine whether a given x is infeasible, we can be assured that no ‘dead ends’ will be explored when enumerating and generating interpart restricted compositions.

3.1.2 Fundamental Results

In this subsection we develop some fundamental results that are the basis of our enumerative and generative techniques. Three fundamental results are proved here. We prove that the set $\mathcal{C}_\sigma(n)$ always contains the singleton composition $\langle n \rangle$. We then develop necessary and sufficient conditions, which can be evaluated efficiently, to determine if a value x is a feasible initial part for some n and σ . Finally, we prove the fundamental bijection which allows us to decompose problems in terms of the interpart restricted compositions of n into a corresponding problem consisting of a feasible initial part x and the interpart restricted compositions of $n - x$ with initial part at least $\sigma(x)$.

The most basic property of the set of interpart restricted compositions is that it is nonempty — irrespective of the value of n or the restriction function σ , there is always at least one composition in the set. This composition is the singleton composition $\langle n \rangle$. We formally prove the existence of the singleton composition in the following lemma.

Lemma 3.1. *For all $n \in \mathbb{Z}^+$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $\langle n \rangle \in \mathcal{C}_\sigma(n)$.*

Proof. Let $n \in \mathbb{Z}^+$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be arbitrary, and let $a_1 \dots a_k = \langle n \rangle$. Using the three necessary and sufficient conditions for membership of the set $\mathcal{C}_\sigma(n)$, we now demonstrate that $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$:

1. $a_j \in \mathbb{Z}^+$ for $1 \leq j \leq k$;
2. $a_1 + \dots + a_k = n$;
3. $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$.

Properties (1) and (2) follow immediately as $k = 1$ and n is a positive integer. Property (3) is vacuously true as $k = 1$. □

Lemma 3.1 shows that n is always a feasible initial part for any function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, and so there is always at least one composition in the set $\mathcal{C}_\sigma(n)$. We also require a means of determining whether a positive integer $x < n$ is a feasible initial part. The following lemmas show that $x + \sigma(x) \leq n$ is a necessary and sufficient condition for $x < n$ to be a feasible initial part for n and σ .

§ 3.1. An Algorithmic Framework

Lemma 3.2. *For all positive integers x and n such that $x < n$ and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$,*

$$\exists a_1 \dots a_k (a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = x) \implies x + \sigma(x) \leq n.$$

Proof. Let $x < n$ be arbitrary positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be an arbitrary function. Suppose $\exists a_1 \dots a_k (a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = x)$. As $x < n$ and all parts of $a_1 \dots a_k$ must be positive integers, we conclude that $k \geq 2$. Then as $a_1 = x$ and $a_1 + \dots + a_k = n$, we get $a_2 = n - x - (a_3 + \dots + a_k)$. Furthermore, as $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$, we see that $\sigma(x) \leq a_2$. Combining this information, we get $\sigma(x) \leq n - x - (a_3 + \dots + a_k)$, or $x + \sigma(x) \leq n - (a_3 + \dots + a_k)$. Thus, we see that $x + \sigma(x) \leq n - (a_3 + \dots + a_k) \leq n$, and $x + \sigma(x) \leq n$, as required. \square

Lemma 3.3. *For all positive integers x and n and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$,*

$$x + \sigma(x) \leq n \implies \exists a_1 \dots a_k (a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = x).$$

Proof. Let n and x be arbitrary positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be an arbitrary function. Suppose $x + \sigma(x) \leq n$, or $\sigma(x) \leq n - x$. Then, as the codomain of σ is \mathbb{Z}^+ , we know that $1 \leq \sigma(x)$, and so $1 \leq \sigma(x) \leq n - x$. Thus, $x < n$. Suppose that $b_1 b_2 = (x)(n - x)$. By the preceding argument $n - x \geq 1$ and x is a positive integer. Clearly, $x + n - x = n$, and furthermore, as $\sigma(x) \leq n - x$, $b_1 b_2$ fulfils the three necessary and sufficient conditions for membership of the set $\mathcal{C}_\sigma(n)$. \square

In Lemma 3.2 we showed that if there exists some $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$ where the initial part is x , and $x < n$, then the condition $x + \sigma(x) \leq n$ must hold true. Then, in Lemma 3.3 we showed that if $x + \sigma(x) \leq n$ then there must exist some $a_1 \dots a_k$ in the set $\mathcal{C}_\sigma(n)$ such that the initial part is equal to x . Note that in Lemma 3.3 the requirement that $x < n$ is directly implied by the premise ($x + \sigma(x) \leq n$). In Lemma 3.2, on the other hand, we must assert that $x < n$ if the premise (there exists some $a_1 \dots a_k$ in $\mathcal{C}_\sigma(n)$ where $a_1 = x$) is to imply that $x + \sigma(x) \leq n$. This slight asymmetry is caused by

§ 3.1. An Algorithmic Framework

the singleton composition, and we shall now develop necessary and sufficient conditions for a positive integer x to be a feasible initial part for n and σ .

Theorem 3.1 (Feasible Initial Parts). *If n and x are positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is a function, then*

$$\exists a_1 \dots a_k (a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = x) \iff x + \sigma(x) \leq n \vee x = n. \quad (3.2)$$

Proof. By Lemmas 3.2 and 3.3 the biconditional

$$\exists a_1 \dots a_k (a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = x) \iff x + \sigma(x) \leq n \quad (3.3)$$

holds for all positive integers $x < n$. By Lemma 3.1 we know that n is always a feasible initial part, but if $x = n$, then $x + \sigma(x) > n$, as the codomain of σ is the positive integers. Thus, combining this with (3.3) we get (3.2), as required. \square

Thus, by Theorem 3.1 we know that for any sequence $a_1 \dots a_k$ in the set $\mathcal{C}_\sigma(n)$ the initial part a_1 either obeys the condition $a_1 + \sigma(a_1) \leq n$ or $a_1 = n$. Stated another way, if we are given a value x such that $x \neq n$ and $x + \sigma(x) > n$ then we know that there cannot be any compositions in the set $\mathcal{C}_\sigma(n)$ such that the initial part is equal to x .

Theorem 3.1 provides us with an efficient means of determining what values may be initial parts for a given value of n and σ . For our enumeration and generation techniques we require one further fundamental result: although we now know how to determine whether a given value is a feasible initial part, we still do not know how to proceed with solving the rest of the problem. The following theorem shows that if we are given an interpart restricted composition $a_1 \dots a_k$ of n and σ , then the sequence $a_2 \dots a_k$ is an interpart restricted compositions of $(n - a_1)$ and σ , where the initial part is at least $\sigma(a_1)$.

Theorem 3.2 (Fundamental Bijection). *Let x and n be positive integers and*

§ 3.1. An Algorithmic Framework

$\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be some function. Then, if $x + \sigma(x) \leq n$,

$$\begin{aligned} & \{x \cdot a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{C}_\sigma(n - x, \sigma(x))\} \\ & = \{b_1 \dots b_l \mid b_1 \dots b_l \in \mathcal{C}_\sigma(n) \wedge b_1 = x\}. \end{aligned}$$

Proof. Suppose x and n are positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary function. Suppose $x + \sigma(x) \leq n$, and let $A = \{x \cdot a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{C}_\sigma(n - x, \sigma(x))\}$ and $B = \{b_1 \dots b_l \mid b_1 \dots b_l \in \mathcal{C}_\sigma(n) \wedge b_1 = x\}$. We now demonstrate that $A \subseteq B$ and $B \subseteq A$.

Let $a_1 \dots a_k$ be an arbitrary element of A . Since $a_2 \dots a_k \in \mathcal{C}_\sigma(n - x, \sigma(x))$, we know that $a_2 + \dots + a_k = n - x$ and $\sigma(a_j) \leq a_{j+1}$ for $2 \leq j < k$. Then, as $a_1 = x$ and $\sigma(x) \leq a_2$, we know that $a_1 + \dots + a_k = n$ and $\sigma(a_j) \leq a_{j+1}$ for $1 \leq j < k$, and hence $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$. Furthermore, as $a_1 = x$, we know that $a_1 \dots a_k \in B$, and thus, as $a_1 \dots a_k$ is an arbitrary element of A , $A \subseteq B$.

Let $b_1 \dots b_l$ be an arbitrary element of B . Since $b_1 \dots b_l \in \mathcal{C}_\sigma(n)$ and $b_1 = x$, we know that $b_2 + \dots + b_l = n - x$, $\sigma(b_j) \leq b_{j+1}$ for $2 \leq j < l$ and $\sigma(x) \leq b_2$. Thus, $b_2 \dots b_l \in \mathcal{C}_\sigma(n - x, \sigma(x))$ and furthermore, as $b_1 = x$, $b_1 \dots b_l \in A$. Therefore, as $b_1 \dots b_l$ is an arbitrary element of B , $B \subseteq A$. Thus, as $A \subseteq B$ and $B \subseteq A$, $A = B$, as required. \square

Theorem 3.2 is essential as it shows that there is a bijection between the sets of interpart restricted compositions of $n - x$ where the initial part is at least $\sigma(x)$ and the set of interpart restricted compositions of n where the initial part is *exactly* x . Thus, when we have a value x that we know is a feasible initial part (from Theorem 3.1) we can assign this value and operate recursively on the set $\mathcal{C}_\sigma(n - x, \sigma(x))$.

The results of this section show that we can determine whether a given value is a feasible initial part in constant time, and indicate how we may proceed with solving the remainder of our problems. Therefore, when given a value x we do not have to attempt to build a composition matching the required criteria, only to find after some unspecified time that the composition we have been constructing does not meet our requirements. Computing

§ 3.1. An Algorithmic Framework

the feasibility of a given value allows us make a backtrack-free guarantee: we can *always* ascertain in constant time whether a given value can be extended to a complete composition.

3.1.3 Examples

In this subsection we illustrate some of the possible instantiations of interpart restricted compositions by examining some classes of restricted partition from the literature, and how they may be represented by different functions. This is not an exhaustive list of all possible instantiations of the framework and is perhaps not even a representative subset of the different types of restriction that may be imposed. In the interest of keeping the examples relevant to previous work in the area we have restricted the examples chosen to classes of restricted partition and composition that have been previously studied in the literature.

Unrestricted Compositions

An unrestricted composition of n is the simplest instance of an interpart restricted composition. Using the function $\sigma(x) = 1$, the set of interpart restricted compositions of n will contain the unrestricted compositions of n . If we substitute this function into (3.1), we obtain

$$1 \leq a_2, 1 \leq a_3, \dots, 1 \leq a_k$$

which makes the relationship plain: no restriction other than the requirement that all parts must be positive integers is stipulated. Computing the feasibility of initial parts for the unrestricted compositions is trivial — substituting $\sigma(x) = 1$ into the conditions of Theorem 3.1, we see that a positive integer x is a feasible initial part if $x + 1 \leq n$ or $x = n$. Therefore, all values $1 \leq x \leq n$ are feasible initial parts, and this corresponds to our condition $1 \leq a_2$ above.

§ 3.1. An Algorithmic Framework

Unrestricted Partitions

An unrestricted partition of n is an unordered collection of positive integers whose sum is n . As we shall explore in detail in Chapter 5, partitions encoded as ascending compositions can be generated more efficiently than partitions encoded as descending compositions. We shall therefore use ascending compositions to encode partitions in preference to the standard convention of encoding partitions as descending compositions [And76, p.xiii], and defer our justification of this practice to Chapter 5. Thus, the unrestricted partitions of 5 are represented by the ascending compositions

$$1 + 1 + 1 + 1 + 1 = 1 + 1 + 1 + 2 = 1 + 1 + 3 = 1 + 2 + 2 = 1 + 4 = 2 + 3 = 5.$$

We can, therefore, represent the partitions of n by instantiating the interpart restriction framework to ensure that the compositions specified are arranged in ascending order. We can achieve this using the interpart restriction function $\sigma(x) = x$, which, when substituted into (3.1) gives the system of inequalities

$$a_1 \leq a_2, a_2 \leq a_3, \dots, a_{k-1} \leq a_k$$

which clearly specify that parts are arranged in ascending order. Evaluating our conditions for initial part feasibility in this instance (Theorem 3.1), we see that a positive integer x is a feasible initial part if $2x \leq n$ or $x = n$. We can see this in the example of the partitions of 5 above, where 1, 2 and 5 are the only initial parts: $2 \times 3 \not\leq 5$ and $2 \times 4 \not\leq 5$, so 3 and 4 are infeasible initial parts.

Partitions into Distinct Parts

One of the first results in the theory of partitions demonstrates that the number of partitions of n where all parts are different (or partitions into distinct parts) is equal to the number of partitions of n where all parts are odd [BS05, §2]. Partitions into distinct parts can also be seen as sets of positive integers whose sum is n , and are a fundamental combinatorial quantity. Similarly to the one-to-one correspondence between the unrestricted partitions of n and

§ 3.1. An Algorithmic Framework

the ascending compositions of n , there is also a one-to-one correspondence between the distinct partitions of n and the compositions of n whose parts are in strictly increasing order. Thus, the partitions into distinct parts of 8 are represented by the compositions

$$1 + 2 + 5 = 1 + 3 + 4 = 1 + 7 = 2 + 6 = 3 + 5 = 8.$$

The rule that parts must be arranged in strictly increasing order can easily be seen as requiring that any composition $a_1 \dots a_k$ must have $a_j < a_{j+1}$ for $1 \leq j < k$. This system of inequalities can be represented using the restriction function $\sigma(x) = x + 1$, because when we substitute into (3.1) we get the inequalities

$$a_1 + 1 \leq a_2, a_2 + 1 \leq a_3, \dots, a_{k-1} + 1 \leq a_k$$

which clearly require that all parts be distinct.

Using the function $\sigma(x) = x + 1$ in our conditions for initial part feasibility in Theorem 3.1, we see that a positive integer x is a feasible initial part for n and $\sigma(x) = x + 1$, if $2x + 1 \leq n$ or $x = n$. Thus, we can see from the example above that 4 is not a feasible initial part for the distinct partitions of 8.

Rogers-Ramanujan Partitions

The first Rogers-Ramanujan identity [BS05, §2] states that the number of partitions of n where the difference between parts is at least 2 is equal to the number of partitions of n where all parts are $\equiv \pm 1 \pmod{5}$ — that is, all parts leave a remainder of either 1 or 4 when divided by 5. The second Rogers-Ramanujan identity [BS05, §2] demonstrates that the number of partitions of n where the difference between parts is at least 2, and all parts are at least 2, is equal to the number of partitions of n where all parts are $\equiv \pm 2 \pmod{5}$. The Rogers-Ramanujan identities have many applications in mathematics [Ful00] and in physics [AB89] where, as Andrews [And05] explains, “in simple terms, the Rogers-Ramanujan identities are crucial in studying the behaviour of helium on a graphite plate.” These identities are of central im-

§ 3.1. An Algorithmic Framework

portance in almost all treatments of integer partitions (see Andrews [And76, ch.7], Andrews & Eriksson [AE04, ch.4], Hardy & Wright [HW54, §19.13], Wilf [Wil02, p.11] and others) and represent an extensive field of study in their own right. (See Fulman [Ful00] and Berkovich & McCoy [BM98] for further references and Hardy [Har40] or Andrews [And05] for an account of the history of these identities.)

The Rogers-Ramanujan identities are an important part of the theory of partitions, and are concerned with partitions where the difference between parts is at least 2, which we shall call ‘Rogers-Ramanujan’ partitions for convenience. Once again, there is a one-to-one correspondence between the Rogers-Ramanujan partitions and a class of ascending composition. In this case the requirement that the difference between parts be at least 2 can be translated into the requirement that the difference between *adjacent* parts be at least 2, as parts must be arranged in ascending order. Therefore, the Rogers-Ramanujan partitions of 12 are represented by the compositions

$$1+3+8 = 1+4+7 = 1+11 = 2+4+6 = 2+10 = 3+9 = 4+8 = 5+7 = 12,$$

and, in general, a composition $a_1 \dots a_k$ of n represents a Rogers-Ramanujan partition of n if $a_j + 2 \leq a_{j+1}$ for $1 \leq j < k$. The interpart restriction function $\sigma(x) = x + 2$ encapsulates these conditions.

Solving the initial part feasibility conditions of Theorem 3.1 is quite simple when we are dealing with Rogers-Ramanujan partitions. Substituting $\sigma(x) = x + 2$ into these conditions shows that a positive integer x is a feasible initial part if $2x + 2 \leq n$ or $x = n$. In the example above, we see that $6, \dots, 11$ are all infeasible initial parts for 12 and $\sigma(x) = x + 2$, which follows directly from this condition.

Conditional Restrictions on Partitions

The first Göllnitz-Gordon identity [Ald69] states that the number of partitions of n where the difference between parts is at least 2 and no contiguous even values appear (thus, if $2j$ is a part, $2j + 2$ is not [AE04, p.33]) is equal to the number of partitions of n where all parts are $\equiv \pm 1, 4 \pmod{8}$. Fol-

§ 3.1. An Algorithmic Framework

Following Alladi & Berkovich [AB05], we refer to partitions of the former class as ‘Göllnitz-Gordon’ partitions. If we encode the Göllnitz-Gordon partitions of 13 as ascending compositions we get

$$1+3+9 = 1+4+8 = 1+5+7 = 1+12 = 2+11 = 3+10 = 4+9 = 5+8 = 13.$$

A composition $a_1 \dots a_k$ represents a Göllnitz-Gordon partition if $a_j + 2 + [a_j \text{ even}] \leq a_{j+1}$ for $1 \leq j < k$. This condition ensures that there is a difference of at least 2 between consecutive parts, but also that the difference is at least 3 if the part is even, ensuring that consecutive multiples of 2 do not appear. Therefore, we can represent the Göllnitz-Gordon partitions using the restriction function $\sigma(x) = x + 2 + [x \text{ even}]$.

Substituting the restriction function $\sigma(x) = x + 2 + [x \text{ even}]$ into the conditions of Theorem 3.1 we see that a positive integer x is a feasible initial part for n and σ if $2x + 2 + [x \text{ even}] \leq n$. Thus, we see, in the example of $n = 13$ above, that 5 is a feasible initial part because $2 \times 5 + 2 + [5 \text{ even}] = 12 \leq 13$. On the other hand, 6 is not a feasible initial part because $2 \times 6 + 2 + [6 \text{ even}] = 15 \not\leq 13$.

A similar class of partition was studied by Schur [AG95]. In the partitions of interest in Schur’s theorem, the difference between parts is at least 3 and no contiguous multiples of 3 appear. We can represent such ‘Schur’ partitions using the restriction function $\sigma(x) = x + 3 + [x \equiv 0 \pmod{3}]$. Göllnitz’s Big Theorem [AB05] is concerned with partitions where the difference between parts is at least 6, or 7 if the part is $\equiv 0, 1$ or $3 \pmod{6}$; these partitions are represented using the restriction function $\sigma(x) = x + 6 + [x \equiv 0, 1, 3 \pmod{6}]$.

Göllnitz-Gordon, Schur, and Göllnitz partitions are all examples of what we refer to as ‘conditional’ restrictions — that is, the restriction function contains a conditional additive factor. In the next chapter we develop generation algorithms for any class of partition or composition in which the corresponding restriction function is nondecreasing. While the restriction functions for the unrestricted compositions, unrestricted partitions, distinct partitions and Rogers-Ramanujan partitions are clearly nondecreasing, it is

§ 3.1. An Algorithmic Framework

not immediately obvious that the conditional restriction functions we examine here are nondecreasing. We will therefore examine a generalisation of these functions, and show that it must be nondecreasing.

Lemma 3.4. *If $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an increasing function and $\mathbf{s}(x)$ is an arbitrary logical statement in terms of a positive integer x , then the function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ such that $\sigma(x) = f(x) + [\mathbf{s}(x)]$ is nondecreasing.*

Proof. Suppose $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an increasing function and $\mathbf{s}(x)$ is an arbitrary logical statement in terms of a positive integer x . Let $\sigma(x) = f(x) + [\mathbf{s}(x)]$ and suppose y is an arbitrary positive integer. Consider the following inequalities (which correspond to possibilities for $\mathbf{s}(y)$ and $\mathbf{s}(y+1)$)

$$f(y)+0 < f(y+1)+0, \quad f(y)+0 < f(y+1)+1, \quad f(y)+1 < f(y+1)+1 \quad (3.4)$$

each of which is an obvious consequence of f being an increasing function. Only one possibility remains in the ‘truth-table’, that is, where $[\mathbf{s}(y)] = 1$ and $[\mathbf{s}(y+1)] = 0$. Since f is an increasing function, $f(y) < f(y+1)$ and it follows that

$$f(y) + 1 \leq f(y+1) + 0. \quad (3.5)$$

We know from (3.4) and (3.5) that $\sigma(y) \leq \sigma(y+1)$ for an arbitrary integer y and all possible values of $\mathbf{s}(y)$ and $\mathbf{s}(y+1)$, and σ is therefore nondecreasing. \square

Thus, by Lemma 3.4 we are assured that any conditional restriction function of the type given above can be correctly generated using the algorithms of the next chapter. (The techniques of this chapter make no assumption about the properties of the restriction function other than its type signature; it is for the purposes of efficient generation that we require the nondecreasing property.)

Multiplicative Restrictions

Bousquet-Mélou & Eriksson [BME97b] generalised Euler’s theorem (the number of partition into distinct parts = the number of partitions into odd

§ 3.1. An Algorithmic Framework

parts), by noting that the requirements of distinctness can “be perceived as demanding that the *quotient* between consecutive parts be greater than one” [BME97b, §1]. One of Bousquet-Mélou & Eriksson’s results asserts that the number of partitions in which the quotient between parts is greater than γ , where $\gamma = (r + \sqrt{r^2 - 4})/2$ for some integer $r \geq 2$, is equal to the number of partitions of n into parts from the set $\{e_1, e_2, \dots\}$, with $e_1 = 1$, $e_2 = r + 1$ and $e_j = re_{j-1} - e_{j-2}$ for $j > 2$. For example, if we set $r = 3$ we obtain the partitions of n where the quotient of consecutive parts is greater than $(3 + \sqrt{5})/2$; and when we express these as ascending compositions we get

$$1 + 3 + 12 = 1 + 4 + 11 = 1 + 15 = 2 + 14 = 3 + 13 = 4 + 12 = 16.$$

In general, we may require compositions $a_1 \dots a_k$ such that $\gamma a_j < a_{j+1}$ for $1 \leq j < k$, where γ is some real value ≥ 1 . Representing such compositions as interpart restricted compositions is quite simple, complicated only by the fact that strict inequality is required and that the quotient is real valued. To ensure that $\gamma a_j < a_{j+1}$ for $1 \leq j < k$ we use the restriction function

$$\sigma(x) = \lceil \gamma x \rceil + \llbracket \lceil \gamma \rceil = \gamma \rrbracket. \quad (3.6)$$

It is straightforward to see that, if γ is not an integer, then we can enforce the required inequality using $\sigma(x) = \gamma x$. The Iversonian expression $\llbracket \lceil \gamma \rceil = \gamma \rrbracket$ is required when γ is an integer, as parts must be strictly greater than γ times the previous. We can also see that (3.6) is a nondecreasing function as the logical condition is in terms of the multiplicative factor γ and not of the parameter x . Therefore, for an arbitrary positive integer y , we will have $\sigma(y) \leq \sigma(y + 1)$.

In the example above we can see that the integers $1, \dots, 4$ and 16 are the feasible initial parts for $n = 16$ and $\sigma(x) = \lceil x(3 + \sqrt{5})/2 \rceil$. This can be verified from Theorem 3.1 which shows that a positive integer x is a feasible initial part in this instance when $x + \lceil x(3 + \sqrt{5})/2 \rceil \leq n$ or $x = n$. Thus, for example, 6 is not a feasible initial part because $6 + \lceil 6 \times (3 + \sqrt{5})/2 \rceil = 22 \not\leq 16$.

§ 3.1. An Algorithmic Framework

A similar class of restricted partition was studied by Hickerson [Hic74], who investigated partitions where the ratio between consecutive parts is at least some positive integer r . More precisely, Hickerson studied the partitions $a_1 \dots a_k$ where $ra_j \leq a_{j+1}$ for $1 \leq j < k$; such partitions can be represented using the restriction function $\sigma(x) = rx$.

Discussion

The examples of restricted partition we have just reviewed are chosen from the literature to reflect both the type of restriction we can impose using interpart restricted compositions and the classes of restricted partition that have been previously studied. We have seen that we can impose a rich class of restriction on partitions with our framework. The classes of partition we have examined in this section are of combinatorial importance and a large literature exists on the mathematical aspects of these partitions. For example, we considered the partitions into distinct parts and Rogers-Ramanujan partitions, both of which are an instance of the restriction function $\sigma(x) = x + d$. Partitions with general difference conditions of this type are the subject of an open conjecture under active investigation [Yee04]. We also saw more general conditional restrictions and multiplicative restrictions, but these are a small subset of the restrictions that can be imposed. Rather than propose contrived examples, we have examined instances of the framework which have been studied previously. We could, for example, specify partitions in which each part is at least the cube of the previous part ($\sigma(x) = x^3$), or where the minimum relationship between parts is given by some polynomial — e.g., $\sigma(x) = x^2 + 2x + 3$. *Any* function can be used to specify a class of interpart restricted composition and there is therefore a large number of classes of restriction that can be imposed.

We can therefore imagine the set of all restriction functions arranged on a ‘number-line’, arranged in increasing order of ‘restrictiveness’. In this illustration we show the restriction functions and corresponding asymptotic formulas for the unrestricted [AS81, p.825] and distinct [AS81, p.825–826]

§ 3.1. An Algorithmic Framework

partitions.

$$\begin{array}{ccccccc}
 C_\sigma(n) \sim & 2^{n-1} & & \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}} & & \frac{e^{\pi\sqrt{n/3}}}{4n^{3/4}3^{1/4}} & & 1 \\
 \sigma(x) = & 1 & & x & & x+1 & & n
 \end{array}$$

(This illustration is a purely figurative device, and no implications of scale should be drawn.) On the left-hand extreme of the continuum we have the unrestricted compositions of n , specified by the restriction function $\sigma(x) = 1$, and of which there are 2^{n-1} . On the other extreme we have the restriction function $\sigma(x) = n$, and there is only one composition that satisfies the interpart restriction criteria for this function. These are the bounds between which all other restriction functions must be placed, as there cannot be any functions either more or less restrictive than these functions. Between these functions then lie all the restriction functions we have considered in this section.

We have concentrated on classes of restricted partition, but we may also represent classes of restricted composition. The restrictions we can impose on compositions, however, are relatively limited. It is much more difficult to impose restrictions on the values of compositions because we cannot enforce a global ordering over the parts, and so we only retain information about adjacent parts. Thus, while it is easy to enforce distinctness on partitions, we cannot define compositions with distinct parts within the framework we have defined.

We can, however, impose a lower-bound on the value of each individual part in compositions [And76, p.63]. If we let d be the minimum value allowed in compositions, then each composition $a_1 \dots a_k$ must have $d \leq a_j$ for $1 \leq j \leq k$. We can represent such compositions using the restriction function $\sigma(x) = d$. Substituting this function into (3.1) we get the inequalities

$$d \leq a_2, d \leq a_3, \dots, d \leq a_k$$

We can see that $d \leq a_2$, but no restriction is made on the initial part —

§ 3.2. Enumeration

therefore, if we wish to obtain only the compositions where *all* parts are at least d , then we must also restrict the minimum value of the initial part to be d . Fortunately, it is trivial for us to enforce a minimum value on the initial part of the compositions we require, as this is the means by which all of our algorithms decompose the problem of enumerating or generating interpart restricted compositions. Thus, if we define the restriction function as $\sigma(x) = 2$, the set $\mathcal{C}_\sigma(6, 2)$ is given by

$$2 + 2 + 2 = 2 + 4 = 3 + 3 = 4 + 2 = 6$$

which are all the compositions of 6 where all parts are at least 2. (These may alternatively be referred to as compositions with no occurrence of 1, the properties of which have been studied by Cayley [Cay76] and Grimaldi [Gri01].)

3.2 Enumeration

Enumeration, or “counting the number of elements of a finite set” [Sta86, p.1], is fundamental to the study of combinatorics. Many techniques exist to enumerate classes of partition and composition, the most common being recurrence relations, generating functions and direct formulas. Exact formulas are not known for all of the examples we considered in the previous section, and some of those that are known are extremely complex: see, for instance, the Hardy-Ramanujan-Rademacher formula for the unrestricted partitions of n [And76, ch.5]. Generating functions [Wil90] are known for all of the classes of restricted partition we have used to exemplify interpart restricted compositions and are a fundamental tool in the study of enumerative combinatorics [Sta86]. Our topic of study in this dissertation is not enumeration per se, and so we shall not investigate the possibility of a generalised generating function to enumerate interpart restricted compositions. For the purposes of this dissertation a general recurrence equation to enumerate interpart restricted compositions is sufficient, and is, at any rate, a useful starting point for a more in-depth enumerative study of the framework.

§ 3.2. Enumeration

Thus, in Section 3.2.1 we develop a recurrence equation which enumerates the interpart restricted compositions of n for any restriction function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. In the interest of completeness, we also provide a simple dynamic programming implementation of this recurrence, allowing us to enumerate the interpart restricted compositions for all $1 \leq j \leq n$ in $O(n^2)$ time and space. Then, in Section 3.2.2 we demonstrate a simple application of this algorithm in computing integer sequences from the Online Encyclopedia of Integer Sequences [Slo05], and provide formulas to compute some sequences of interest from the literature.

3.2.1 A General Recurrence Equation

In this section we develop two recurrence equations to enumerate interpart restricted compositions. These recurrences are equivalent but presented in alternative forms, suitable for applications to generation (as we shall see in the next chapter) and enumeration (which we use to implement a dynamic programming enumeration algorithm in this section). Recurrence relations are crucial for the development and analysis of recursive generation algorithms for combinatorial objects [Rus01, §4.11]. Recurrence relations to enumerate classes of restricted partition have also been used to develop efficient parallel enumeration algorithms [SS96] and in applications to percolation theory and nuclear physics [Dés02].

The first recurrence we shall consider operates by summing $C_\sigma(n-x, \sigma(x))$ over all feasible initial parts x for n and σ . By Theorem 3.2 we know that $C_\sigma(n-x, \sigma(x))$ is equal to the number of compositions of n interpart restricted by σ where the initial part is exactly x , and so the summation outlined above will correctly compute $C_\sigma(n, m)$. We shall now prove the validity of this approach. For the purposes of the following theorem it is useful to define the function $C_\sigma^*(n, m)$ as the number of elements of $\mathcal{C}_\sigma(n)$ where the initial part is exactly m ; thus, $C_\sigma^*(n, m) = |\{a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{C}_\sigma(n) \wedge a_1 = m\}|$.

§ 3.2. Enumeration

Theorem 3.3. *For all positive integers $m \leq n$ and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $C_\sigma(n, m)$ satisfies the recurrence*

$$C_\sigma(n, m) = 1 + \sum_{\substack{x \geq m \\ x + \sigma(x) \leq n}} C_\sigma(n - x, \sigma(x)). \quad (3.7)$$

Proof. Let $m \leq n$ be arbitrary positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be an arbitrary function. By Theorem 3.1, a positive integer x is a feasible initial part for n and σ if $x + \sigma(x) \leq n$ or $x = n$. Clearly, we can compute $C_\sigma(n, m)$ by summing $C_\sigma^*(n, x)$ over all feasible initial parts that are $\geq m$, and therefore we have

$$C_\sigma(n, m) = C_\sigma^*(n, n) + \sum_{\substack{x \geq m \\ x + \sigma(x) \leq n}} C_\sigma^*(n, x).$$

By Theorem 3.2 we know that $C_\sigma^*(n, x) = C_\sigma(n - x, \sigma(x))$, and clearly $C_\sigma^*(n, n) = 1$, giving us the required result. \square

Recurrence (3.7) is an example of what Ruskey refers to as *positive recurrences* [Rus01, §1.2]: it is stated in a form that involves no division or subtraction and involves only positive values. This recurrence will prove useful in the next chapter when we examine the problem of recursively generating all interpart restricted compositions; as the recurrence is positive, we can expect that this algorithm will be efficient [Rus01, §1.2].

For the purposes of enumeration, however, it is useful to have a recurrence that does not require a summation over an indeterminate number of values. In particular, to develop an efficient dynamic programming [CLR90, ch.16] enumeration algorithm, we require a recurrence which involves a constant number of terms. We can reduce the number of recursive terms involved in computing $C_\sigma(n, m)$ by noting that if we extract the first term from the summation in (3.7) then the remaining terms are equivalent to $C_\sigma(n, m + 1)$. This observation then reduces the number of terms in the equation to two and requires the introduction of a base case, making the resulting recurrence ideal for dynamic programming.

§ 3.2. Enumeration

Theorem 3.4. *For all positive integers $m \leq n$ and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $C_\sigma(n, m)$ satisfies the recurrence*

$$C_\sigma(n, m) = C_\sigma(n - m, \sigma(m)) + C_\sigma(n, m + 1), \quad (3.8)$$

with the initial conditions $C_\sigma(n, n) = 1$ and $C_\sigma(n, m) = 0$ when $m > n$.

Proof. We begin by examining the initial conditions. Suppose $m > n$. Then, as the codomain of σ is \mathbb{Z}^+ , $m + \sigma(m) > n$, so m is not a feasible initial part by Theorem 3.1. Suppose, alternatively, that $m = n$. Then, by Theorem 3.3 above $C_\sigma(n, m) = 1$.

More generally, suppose that $1 \leq m < n$. From (3.7) we have

$$\begin{aligned} C_\sigma(n, m) &= 1 + \sum_{\substack{x \geq m \\ x + \sigma(x) \leq n}} C_\sigma(n - x, \sigma(x)) \\ &= C_\sigma(n - m, \sigma(m)) + 1 + \sum_{\substack{x \geq m+1 \\ x + \sigma(x) \leq n}} C_\sigma(n - x, \sigma(x)) \\ &= C_\sigma(n - m, \sigma(m)) + C_\sigma(n, m + 1) \end{aligned}$$

Therefore, since we have verified each of the base cases and the general case, the recurrence is valid. \square

Informally we can see that the general case of recurrence (3.8) works because the first term counts all of the compositions in which the first part is equal to m and we therefore have $n - m$ left to assign, and the first part of the compositions of $n - m$ must be at least $\sigma(m)$ to maintain the inequalities required for the interpart restrictions. The second term then counts the compositions in which the initial part is greater than m , giving us the total number of compositions in question. This idea is illustrated in Figure 3.1, where we examine the evaluation of $C_\sigma(11, 1)$ when $\sigma(x) = x + 2$. On the left-hand side of this illustration we see the set of Rogers-Ramanujan partitions of 11 where the initial part is exactly 1; and these are enumerated by $C_\sigma(11 - 1, \sigma(1)) = C_\sigma(10, 3)$. On the right-hand side of Figure 3.1 we then see the Rogers-Ramanujan partitions of 11 where the initial part is at

§ 3.2. Enumeration

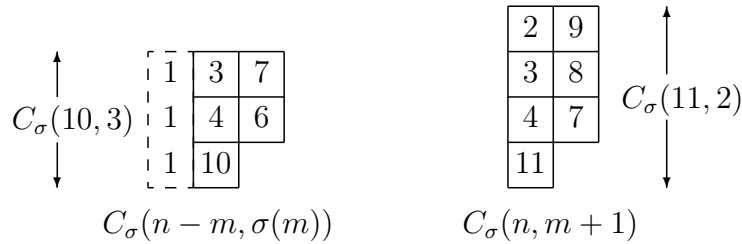


Figure 3.1: Illustration of recurrence to count interpart restricted compositions for $\sigma(x) = x + 2$. We can construct the Rogers-Ramanujan partitions of 11 by prefixing 1 to all Rogers-Ramanujan partitions of 10 with initial part at least 3. The remaining partitions, where the initial part is at least 2, are found recursively.

least 2, and these are enumerated recursively by the same method.

It is useful to have an algorithm to compute $C_\sigma(n, m)$ efficiently for an arbitrary instance of the restriction function. A simple recursive implementation of (3.8) is inefficient, but using dynamic programming it can be used to good effect. By storing a lower-triangular matrix of values for the $C_\sigma(n, m)$ function we avoid recursively recomputing the values $C_\sigma(n - m, \sigma(m))$ and $C_\sigma(n, m + 1)$. If we compute each row of the matrix from right-to-left we ensure that the value of $C_\sigma(n, m + 1)$ is available; and if we begin at $n = 1$ and calculate each row completely before moving on to the next, we know that $C_\sigma(n - m, \sigma(m))$ must also be at hand. Algorithm 3.1 is a straightforward application of this technique. The operation of the algorithm is similar to partition enumeration algorithms [McK65a, Whi70a, BS73] from the literature, and has similar time and space complexity: we require $O(n^2)$ time and space to fully compute the tabulation of $C_\sigma(j, m)$ for all values of $j \leq n$.

3.2.2 Integer Sequences

Integer sequences relating to partitions and compositions have been studied extensively, and the Online Encyclopedia of Integer Sequences [Slo05] contains many entries relating to these combinatorial objects. For example, if

§ 3.2. Enumeration

Algorithm 3.1 $\text{TABULATE}_\sigma(N)$

Require: $N > 0$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$

```

for  $n \leftarrow 1$  up to  $N$  do
   $c_{n,n} \leftarrow 1$ 
  for  $m \leftarrow n - 1$  down to  $1$  do
    if  $m + \sigma(m) \leq n$  then
       $c_{n,m} \leftarrow c_{n-m, \sigma(m)} + c_{n,m+1}$ 
    else
       $c_{n,m} \leftarrow c_{n,m+1}$ 
    end if
  end for
end for

```

we set $\sigma(x) = x + 2$ and $m = 2$, we compute the sequence

$$0, 1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 6, 6, 8, 9, 11, 12, 15, 16, 20$$

when we evaluate $C_\sigma(n, m)$ for $1 \leq n \leq 20$, and this is sequence A003106 in Sloane's encyclopedia. We know that this sequence counts the number of partitions with minimum interpart distance of 2 where the initial part is at least 2, but the second Rogers-Ramanujan identity (see Section 3.1.3) tells us that this sequence also counts the partitions of n into parts $\equiv \pm 2 \pmod{5}$. If enumeration of a particular class of composition is all that we require then we can use the framework on many more classes of partition and composition using the numerous partition identities [Ald69, Pak05]. An overview of these identities and the types of congruence condition we can enumerate is given in Table 3.1.

We can also compute sequences which are not directly enumerable, but can be obtained more tangentially. Alladi [All99] studied the "number partitions of n into parts $\neq 2$ and differing by ≥ 6 , where the inequality is strict whenever a part is even", which he proves to be equal to the number of partitions into distinct odd parts. We can enumerate such partitions using our recurrence with minimal difficulty, using the restriction function $\sigma(x) = x + 6 + [x \text{ even}]$. As 2 is excluded, however, we cannot simply evaluate $C_\sigma(n, 1)$ for $n = 1, 2, \dots$. This difficulty is easily circumvented by noting that

<i>Restriction Function</i>	<i>Formula</i>	<i>Congruence</i>	<i>Sequence</i>	<i>Proof</i>
$x + 1$	$C_\sigma(n)$	$\equiv 1 \pmod{2}$	A000009	[And76, p.5]
$x + 2$	$C_\sigma(n)$ $C_\sigma(n, 2)$	$\equiv \pm 1 \pmod{5}$ $\equiv \pm 2 \pmod{5}$	A003114 A003106	[And76, ch.7] [And76, ch.7]
$x + [x \text{ even}]$	$C_\sigma(n)$	$\not\equiv 0 \pmod{4}$	A001935	[Hon85, p.68]
$x + [x \text{ odd}]$	$C_\sigma(n)$	$\not\equiv 2 \pmod{4}$	A006950	
$x + 2 + [x \text{ even}]$	$C_\sigma(n)$ $C_\sigma(n, 3)$	$\equiv \pm 1, 4 \pmod{8}$ $\equiv \pm 3, 4 \pmod{8}$	A036016 A036015	[AB05] [AB05]
$x + 2 + [x \text{ odd}]$	$C_\sigma(n)$ $C_\sigma(n, 2)$	$\not\equiv 3 \pmod{4}$ $\not\equiv 1 \pmod{4}$		[AB05] [AB05]
$x + 3 + [x \equiv 0 \pmod{3}]$	$C_\sigma(n)$ $C_\sigma(n)$	$\equiv \pm 1 \pmod{6}$ $\not\equiv \pm 1 \pmod{3}$	A003105 A003105	[AE04, p.36] [AG95]
$x + 6 + [x \text{ even}]$	$C_\sigma(n, 2) - C_\sigma(n, 1) + C_\sigma(n, 3)$	$\not\equiv 1 \pmod{2}$	A000700	[All99]
$x + 6 + [x \equiv 0, 1, 3 \pmod{6}]$	$C_\sigma(n, 3) - C_\sigma(n, 2) + C_\sigma(n, 4)$	$\not\equiv 2, 4, 5 \pmod{6}$	A056970	[All99]

Table 3.1: Table of restriction functions and partitions with parts satisfying certain congruence conditions which are enumerable via a partition identity. The restriction function required to enumerate the class of partition specified is provided along with the corresponding formula in terms of $C_\sigma(n, m)$. The class of partitions specified by the congruence condition are of the form “all partitions of n with all parts satisfying α ” where α is the specified congruence condition. Conditions marked with \diamond are partitions with distinct parts only. For each class the appropriate sequence from the online encyclopedia of integer sequences [Slo05] is provided, as is a reference to the literature where the identity in question is proved.

§ 3.3. Summary

if the part 2 is to occur, it must be the initial part, as all subsequent parts must differ from the initial by at least 6. Thus, we can compute the number of such partitions by setting the restriction function to the aforementioned value and evaluating $C_\sigma(n, 2) - C_\sigma(n, 1) + C_\sigma(n, 3)$ for $n = 1, 2, \dots$. Göllnitz's Big Theorem [All99, §3] is of a similar form, and requires that the partitions involved do not include the value 3, which we can again safely remove by reasoning about the initial part.

3.3 Summary

In this chapter we have introduced the concept of *interpart restricted compositions*, an economically expressive means of specifying classes of restricted partition and composition. We have seen that many classes of restricted partition studied in the literature can be expressed within this framework using a simple integer function. The function used to describe the restrictions, known as the restriction function, can then be used as a parameter to general-purpose computational routines. The first application of this framework was also developed in this chapter, where we demonstrate that a general recurrence equation can be used to enumerate restricted partitions, by specifying only the restriction function in question. We also saw how this recurrence can be used to efficiently compute integer sequences from the Online Encyclopedia of Integer Sequences [Slo05], via a simple dynamic programming algorithm. In the next chapter we shall consider, in depth, the problem of *generating* interpart restricted compositions.

Chapter 4

Generating Interpart Restricted Compositions

This chapter is concerned with defining efficient algorithms to generate interpart restricted compositions. In Section 4.1 we deal with some preliminary issues such as the basic definitions we shall require for the purposes of discussing generation algorithms, and discuss the specific subset of the interpart restricted compositions for which we shall develop generation algorithms. Then, in Section 4.2 we begin our development of generation algorithms for interpart restricted compositions with a simple recursive method. We then move on to develop an abstract succession rule for interpart restricted compositions in Section 4.3, and provide a concrete implementation of this rule as a generation algorithm, pausing briefly to compare this algorithm with some existing algorithms from the literature. In Section 4.4 we develop a coherent means of improving the efficiency of this algorithm for all instances of the restriction function by introducing the theory of ‘terminal’ and ‘non-terminal’ compositions. We implement the resulting algorithm, and compare it with the direct implementation of the succession rule to determine what, if any, improvements we can expect. Finally, in Section 4.5 we summarise the results of this chapter.

4.1 Preliminaries

In Section 4.1.1 we define the basic concepts we require for discussing generation algorithms. Then, in Section 4.1.2 we discuss our fundamental requirement of the restriction functions we shall be considering in this chapter.

4.1.1 Definitions and Notation

In this subsection we are concerned with the basic properties required in any generation algorithm, the ideas of ordering and succession. We shall therefore formally define lexicographic order, and also define the notation we shall be using throughout this chapter.

Definition 4.1 (Lexicographic Order). *Let $a = a_1 \dots a_k$ and $b = b_1 \dots b_l$, where $a, b \in \mathcal{C}_\sigma(n)$ for some $n \geq 1$, be arbitrary. We say that a precedes b lexicographically, or $a \prec b$, if there exists some $t < \min(k, l)$ such that $a_j = b_j$ for $1 \leq j < t$ and $a_t < b_t$.*

Definition 4.1 defines the relation \prec over interpart restricted compositions, and is adapted from Williamson's definition of lexicographic order over functions [Wil85, p.14]. An equivalent definition, which will be useful in certain contexts, can be made recursively (also adapted from Williamson [Wil85, p.15]). In this case we let $a_1 \dots a_k$ and $b_1 \dots b_l \in \mathcal{C}_\sigma(n)$ for some $n \geq 1$. We then say that $a_1 \dots a_k \prec b_1 \dots b_l$ if $a_1 < b_1$ or $a_2 \dots a_k \prec b_2 \dots b_l$.

In defining lexicographic succession rules, where we derive the next composition in the succession from the current, it is useful to have formal functions that compute the lexicographically least element of a given set of compositions. We also define the function that computes the immediate lexicographic successor of a given composition.

Definition 4.2 (Lexicographic Minimum). *For some positive integers $m \leq n$ and a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, the function $M_\sigma(n, m)$ computes the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$.*

Definition 4.3 (Lexicographic Successor). *For any $a_1 \dots a_k \in \mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$ the function $S_\sigma(a_1 \dots a_k)$ computes the immediate lexicographic successor of $a_1 \dots a_k$.*

§ 4.1. Preliminaries

Definition 4.3 does not define the value of the function $S_\sigma(a_1 \dots a_k)$ when $a_1 \dots a_k = \langle n \rangle$. This is because the composition $\langle n \rangle$ is the lexicographically largest composition and hence has no lexicographic successor.

Finally, in discussing and proving properties of generation algorithms, it is useful to have a formal and concrete definition of the term ‘generate’.

Definition 4.4 (Generate). *An algorithm generates the set $\mathcal{C}_\sigma(n)$ if it executes the statement **visit** $a_1 \dots a_k$ exactly once for each $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$, and does not execute the statement **visit** $a_1 \dots a_k$ for any $a_1 \dots a_k \notin \mathcal{C}_\sigma(n)$.*

Having defined the basic concepts we shall require for this chapter, we shall now attend to a pressing issue. The problem we turn to now is essential in ensuring that the algorithms we define are as efficient as possible. To do this, we require a simple property of the restriction function: for the algorithms we define in this chapter the restriction function must be *nondecreasing*.

4.1.2 Nondecreasing Restriction Functions

In the previous chapter we developed necessary and sufficient conditions for a positive integer to be a feasible initial part for any $n \geq 1$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. These conditions allowed us compute $C_\sigma(j, m)$ for an all $1 \leq m \leq j \leq n$ and an arbitrary function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ in $O(n^2)$ time and space. For the purposes of efficient generation, however, we require an extra property of the restriction functions which we shall consider: any restriction function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ must be *nondecreasing*. That is, for an arbitrary positive integer x , the condition $\sigma(x) \leq \sigma(x + 1)$ must hold. To see why we must impose this condition, consider the following example.

Suppose we let $\sigma(x) = x + 1 + 10[x = 3 \vee x = 4]$; thus, if x is not equal to 3 or 4 we return $x + 1$, but if x is one of these values, we return $x + 11$. The set of compositions of 13 interpart restricted by this function is given by

$$1 + 2 + 10 = 1 + 5 + 7 = 1 + 12 = 2 + 5 + 6 = 2 + 11 = 5 + 8 = 6 + 7 = 13.$$

The crux of the problem is shown in above the example: 1 and 2 are feasible

§ 4.1. Preliminaries

initial parts for 13 and $\sigma(x) = x + 1 + 10[x = 3 \vee x = 4]$, as are 5 and 6. But 3 and 4 are *not*. From the perspective of generation, where we must systematically generate all possibilities, this property of possible non-contiguity of feasible initial parts leads to inefficient algorithms in general. In this example, because of the non-contiguity, we are forced to iterate through all values x between 1 and $n - 1$ to test if they are feasible initial parts: and the uncertainty introduced by this behaviour propagates throughout the algorithm.

To avoid this unacceptable uncertainty and inefficiency in generation algorithms, we introduce a requirement on the restriction functions that we shall consider. All of the algorithms in this chapter are defined only for *nondecreasing* restriction functions. As we shall see momentarily, the requirement of the nondecreasing property ensures that all feasible initial parts are contiguous, thereby obviating the need to iterate over all values between 1 and $n - 1$. If the restriction function is nondecreasing, once we have encountered a value x such that x is not a feasible initial part for n and σ , we know that there can be no value y such that $x \leq y < n$ and y is a feasible initial part. We can therefore immediately skip to the singleton composition $\langle n \rangle$, safe in the knowledge that we have not discarded any compositions that match our requirements. We formally prove this result in the following theorem.

Theorem 4.1 (Contiguity of Feasible Initial Parts). *For all positive integers x and n and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, if $x + \sigma(x) > n$ then for all $y > x$, $y + \sigma(y) > n$.*

Proof. Suppose x and n are arbitrary positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. Suppose that $x + \sigma(x) > n$, and let $y > x$ be arbitrary. Since σ is nondecreasing, $\sigma(y) \geq \sigma(x)$, and adding y to both sides of this inequality we see that $y + \sigma(y) \geq y + \sigma(x)$. Then, as $y > x$, $y + \sigma(x) > x + \sigma(x)$, and so $y + \sigma(y) > x + \sigma(x)$. Hence, $y + \sigma(y) > n$, as required. \square

Corollary 4.1. *For all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, if $m + \sigma(m) > n$ then $\mathcal{C}_\sigma(n, m) = \{\langle n \rangle\}$.*

§ 4.1. Preliminaries

Proof. Suppose $m \leq n$ are arbitrary positive integers and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. Suppose that $m + \sigma(m) > n$. By Theorem 4.1 there is no $y > m$ such that $y + \sigma(y) \leq n$, and therefore, by Theorem 3.1, the only feasible initial part that is at least m for n and σ is n . Thus, $\mathcal{C}_\sigma(n, m) = \{\langle n \rangle\}$. \square

Theorem 4.1 demonstrates that feasible initial parts are contiguous if the restriction function is nondecreasing (recall from Theorem 3.1 that a positive integer $x < n$ is a feasible initial part for n and σ iff $x + \sigma(x) \leq n$). Corollary 4.1 then uses this result to compute the set $\mathcal{C}_\sigma(n, m)$ when $m + \sigma(m) > n$. Since there is no feasible initial part with value at least m when $m \neq n$, the only composition which remains is the singleton composition $\langle n \rangle$. These results allow us to define efficient generation algorithms for all compositions interpart restricted by nondecreasing functions, which we shall spend the remainder of this chapter developing.

It is worth noting, however, that the required nondecreasing property of the restriction functions should not be seen as a limiting factor of the overall framework. It is not difficult to introduce an extra function into the framework which, given a particular value x computes the *next* feasible initial part for n and σ . If such a function were defined by the user, then we could easily modify the algorithms given in this chapter to efficiently generate compositions interpart restricted by an arbitrary function. We have not taken this step for two reasons. Firstly, doing so would unnecessarily complicate the algorithms and formalisms involved, without significantly altering the essential nature of the framework. Secondly, the number of classes of partition which have been studied that are represented by nondecreasing functions is very much larger than those that are not. All of the examples we provided in the previous chapter are defined by nondecreasing functions. Partitions restricted by functions that are *not* nondecreasing are relatively obscure (see, for example, Andrews' generalisation of difference conditions [And69]). Thus, an argument could be made for the proposition that nondecreasing restriction functions represent an interesting subset of all possible restrictions and are deserving of study in their own right.

4.2 Recursive Algorithm

In this section we shall develop a simple recursive generation algorithm for interpart restricted compositions for any nondecreasing instance of the restriction function. Despite this algorithm's simplicity, it has the property of being constant amortised time — the “ultimate goal in efficiency” [MM05] for generation algorithms. That is, for any nondecreasing instance of the restriction function, we can guarantee that the algorithm will generate each of the compositions in question in constant time, in an amortised sense [CLR90, ch.18].

The section proceeds as follows. In Section 4.2.1 we discuss the basic principle behind the generation algorithm. In Section 4.2.2 we implement this basic principle and prove the correctness of the resulting algorithm. Then, in Section 4.2.3 we analyse this algorithm, and prove that it has the required constant amortised time property.

4.2.1 Basic Principle

In Section 3.1.2 we proved the validity of what we referred to as the ‘fundamental bijection’ for interpart restricted compositions. In essence, the fundamental bijection tells us that we can obtain all compositions of n interpart restricted by σ , where the initial part is exactly m , by prepending m to all compositions of $n - m$ interpart restricted by σ , where the initial part is *at least* $\sigma(m)$. This procedure ensures that we retain the properties required for a composition $a_1 \dots a_k$ to be a member of the set $\mathcal{C}_\sigma(n)$. We shall briefly summarise the reasons for this here since it is the fundamental basis of our recursive generation algorithm (and indeed all subsequent algorithms).

Suppose m and n are positive integers such that $m + \sigma(m) \leq n$, and suppose the composition $a_1 \dots a_k$ is an element of the set $\mathcal{C}_\sigma(n - m, \sigma(m))$; then let $b_1 \dots b_{k+1} = m \cdot a_1 \dots a_k$. We wish to verify that $b_1 \dots b_{k+1}$ is an element of the set $\mathcal{C}_\sigma(n)$. Clearly, all elements of $b_1 \dots b_{k+1}$ are positive integers, and it is not difficult to see that $b_1 + \dots + b_{k+1} = n$. Thus, the only remaining property we need to demonstrate is that $\sigma(b_j) \leq b_{j+1}$ for $1 \leq j < k + 1$; and we already know that $\sigma(b_j) \leq b_{j+1}$ for $2 \leq j < k + 1$,

§ 4.2. Recursive Algorithm

as $b_2 \dots b_{k+1}$ is an element of the set $\mathcal{C}_\sigma(n - m, \sigma(m))$. It remains, then, to show that $\sigma(b_1) \leq b_2$. By definition, the set $\mathcal{C}_\sigma(n - m, \sigma(m))$ contains all compositions of $n - m$ interpart restricted by σ where the initial part is at least $\sigma(m)$: therefore, $\sigma(m) \leq b_2$. We know $b_1 = m$, and the property is therefore fulfilled.

This is the essence of the algorithm. We iterate over all feasible initial parts x for n and σ and for each value of x we recursively generate the interpart restricted compositions of $n - x$ with initial part at least $\sigma(x)$. There is a technical aspect to how this is actually implemented. If we were to naively implement this approach, it would involve the storage and manipulation of the entire *set* of interpart restricted compositions. But for the purpose of generation we do not need the entire set of objects to be in memory concurrently: we seek only to successively populate a single data structure with the objects we require. Once a complete object is available in memory, we make it available to the program on whose behalf we are generating the objects. We shall discuss the means of implementing this strategy in a recursive generation algorithm in the next subsection.

4.2.2 Algorithm

There is a conceptual difficulty in the recursive generation of combinatorial objects. The natural mode for recursive algorithms is to decompose the problem at hand into recursive subproblems, and compute the required value by combining the values returned by its recursive invocations on the subproblems. If we apply this conceptual idea directly to combinatorial generation, however, we are faced with the problem of returning entire *sets* of objects; we then extend *each* object in the set in the appropriate manner and return the (possibly enlarged) set back up the call chain. This approach is obviously untenable, as we may require exponential space to store all of the objects we generate.

Page & Wilson provided an elegant solution to this problem [PW79]. By recursing over the index of the value under consideration, instead of recursing over complete sets of objects, we can use one global data structure to hold

§ 4.2. Recursive Algorithm

Algorithm 4.1 $\text{RECGEN}_\sigma(n, m, k)$

Require: $1 \leq m \leq n$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is nondecreasing

```
1:  $x \leftarrow m$ 
2: while  $x + \sigma(x) \leq n$  do
3:    $a_k \leftarrow x$ 
4:    $\text{RECGEN}_\sigma(n - x, \sigma(x), k + 1)$ 
5:    $x \leftarrow x + 1$ 
6: end while
7:  $a_k \leftarrow n$ 
8: visit  $a_1 \dots a_k$ 
```

the object we are currently attending to. In this way we can sequentially generate all of the objects in turn, making them available, when appropriate, to the consuming procedure via the **visit** keyword. Using this technique we can treat recursive invocations *as if* they were generating the entire set of objects we require, and the generation algorithms are then relatively simple to describe.

RECGEN_σ (Algorithm 4.1) uses Page & Wilson’s technique to generate all interpart restricted compositions. Each invocation $\text{RECGEN}_\sigma(n, m, k)$ populates a global array a with each interpart restricted composition of n and σ with initial part at least m in array indexes above k . Each time a complete composition is present in the array, we visit the composition before visiting the next one in some other invocation. Thus, we implement the fundamental bijection for interpart restricted compositions via Page & Wilson’s technique in a recursive algorithm with parsimonious memory requirements.

Therefore, when we speak of ‘prefixing m to all interpart restricted compositions of $n - m$ with first part at least $\sigma(m)$ ’, we simply assign the value m to the array index currently under consideration, and use subsequent recursive invocations to visit all interpart restricted compositions of $n - m$ with first part at least $\sigma(m)$, which are then effectively prefixed with value m . An example of Algorithm 4.1 generating a complete set of interpart restricted compositions is given in Figure 4.1, where we demonstrate the complete set of states the array a traverses in the process of generating all compositions of 7 interpart restricted by the function $\sigma(x) = x + 1$.

§ 4.2. Recursive Algorithm

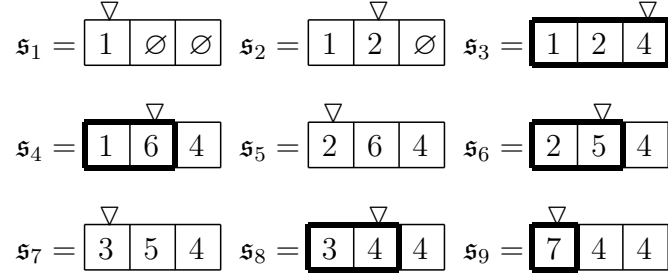


Figure 4.1: Array state-transition diagram for recursive ascending composition generation algorithm. Indices marked with the ∇ symbol are the indices in which a value is assigned to change the state of the array, and array segments outlined in bold are the portions that are visited.

In general, the initial invocation $\text{RECGEN}_\sigma(n, 1, 1)$ generates all compositions of n interpart restricted by σ . We assume that arrays are indexed from 1; minor modifications only are required to use a 0 indexed array. We shall now establish that $\text{RECGEN}_\sigma(n, m, 1)$ correctly generates the set $\mathcal{C}_\sigma(n, m)$ for any nondecreasing restriction function.

Theorem 4.2. *For every nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers $m \leq n$, $\text{RECGEN}_\sigma(n, m, 1)$ generates the set $\mathcal{C}_\sigma(n, m)$ in lexicographic order.*

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. We proceed by strong induction on n .

Base case: $n = 1$. Since $1 \leq m \leq n$, we know that $m = 1$. Then, as the codomain of σ is \mathbb{Z}^+ the condition on line 2 fails, and so we proceed directly to line 7, assign $a_1 \leftarrow 1$, and visit this composition. Clearly $\mathcal{C}_\sigma(1, 1) = \langle 1 \rangle$, and so the base case of our induction holds.

Induction step: Suppose for some positive integer n that the invocation $\text{RECGEN}_\sigma(n', m', 1)$ generates the set $\mathcal{C}_\sigma(n', m')$ in lexicographic order for all positive integers $m' \leq n' < n$. Consider the invocation $\text{RECGEN}_\sigma(n, m, 1)$.

Suppose that $m + \sigma(m) > n$. Upon entering the algorithm we assign $x \leftarrow m$, and as $m + \sigma(m) > n$ the test on line 2 fails, and we do not

§ 4.2. Recursive Algorithm

enter the while loop. We then assign $a_1 \leftarrow n$, visit $\langle n \rangle$ and terminate. By Corollary 4.1 $\mathcal{C}_\sigma(n, m) = \{\langle n \rangle\}$, and so the algorithm correctly generates the set $\mathcal{C}_\sigma(n, m)$. Furthermore, as there is only one element in the set, we also visit each element in lexicographic order.

Suppose, on the other hand, that $m + \sigma(m) \leq n$. Upon entry we assign $x \leftarrow m$, enter the while loop and assign $a_1 \leftarrow m$. We then invoke $\text{RECGEN}_\sigma(n - m, \sigma(m), 2)$. By the inductive hypothesis we know that the invocation $\text{RECGEN}_\sigma(n - m, \sigma(m), 1)$ correctly generates the set $\mathcal{C}_\sigma(n - m, \sigma(m))$. Therefore, by Page and Wilson's technique we know that assigning $a_1 \leftarrow m$ and invoking $\text{RECGEN}_\sigma(n - m, \sigma(m), 2)$ generates all interpart restricted compositions of n where the initial part is exactly m . As m is the smallest possible value for the initial part, there cannot be any compositions that lexicographically precede the first composition visited by this invocation; and as values for the initial part are generated in strictly increasing order, we know that all compositions are visited in lexicographic order. It is easy to establish that all feasible values of the initial part are generated, as we start at the smallest possible value and terminate when $x + \sigma(x) > n$. Theorem 4.1 demonstrates that all feasible initial parts are contiguous, and by Corollary 4.1 we know that when $x + \sigma(x) > n$ only one composition remains. Therefore, we know that we visit all compositions in which the number of parts is at least 2 in lexicographic order, and we then visit the remaining composition $\langle n \rangle$, which is the lexicographically last, and terminate.

Therefore, $\text{RECGEN}_\sigma(n, m, 1)$ correctly generates the set $\mathcal{C}_\sigma(n, m)$ for all positive integers $m \leq n$, and as σ is arbitrary, for all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. \square

Having proved the correctness of RECGEN_σ we shall now consider the efficiency of the algorithm.

4.2.3 Analysis

The constant amortised time performance of RECGEN_σ (Algorithm 4.1) is easily established because of the simple structure of the generation procedure. This structure ensures that exactly one composition is visited per invocation,

§ 4.3. Succession Rule

and as the algorithm visits all elements of $\mathcal{C}_\sigma(n, m)$, counting the number of invocations is trivial. We formally prove this property as follows.

Theorem 4.3. *Algorithm 4.1 generates $\mathcal{C}_\sigma(n, m)$ in constant amortised time.*

Proof. The cost of each invocation of RECGEN_σ is constant, as each loop iteration is associated with a recursive invocation. We can therefore obtain an asymptotic bound on the total time to generate the set $\mathcal{C}_\sigma(n, m)$ by counting the total number of invocations required. (We explicitly disregard time spent visiting objects, as we are interested in the cost of generation only.) Inspection of RECGEN_σ reveals that we visit exactly one sequence per invocation, and so we know that it requires exactly $C_\sigma(n, m)$ invocations to visit all compositions restricted by the function σ . The algorithm therefore requires $O(C_\sigma(n, m))$ time to generate all composition interpart restricted by σ , thus requiring only constant time for each composition, on average. Therefore the algorithm is constant amortised time. \square

We shall compare a concrete instantiation of RECGEN_σ to some algorithms from the literature in Section 5.2, where we consider the special case of generating the unrestricted partitions of n . No other commensurable algorithms exist in the literature and so we shall proceed with our development of interpart restricted composition generation algorithms in the next section, postponing our comparisons until Chapter 5.

4.3 Succession Rule

In the previous section we developed a simple recursive algorithm to generate all interpart restricted compositions of n for any nondecreasing instance of the restriction function. Recursive algorithms are useful in certain contexts, but it is also useful to have a more direct means of generating combinatorial objects. In Section 4.3.1 we develop the abstract succession rule, the fundamental basis of the generation algorithm we devise in Section 4.3.2. Then, in Section 4.3.3 we analyse the algorithm proving that it has the required constant amortised time property. Finally, in Section 4.3.4 we compare our succession-rule based algorithm with some algorithms from the literature.

4.3.1 Basic Principle

A lexicographic succession rule for combinatorial objects is some simple expression that transforms a given object into its immediate lexicographic successor [Kem98]. Lexicographic succession rules form the basis of any iterative lexicographic generation algorithm, and defining a procedure to transform a particular object into its successor is the classical mode of combinatorial generation [NW78, p.3]. The algorithms we define follow the model advocated by Knuth [Knu04b, p.1] where, instead of defining a procedure to transform a *particular* object into its successor, we define algorithms that generate *all* objects of the required class. This model is more efficient [Sed77, p.143] and also makes the process of analysing generation algorithms more transparent. Nevertheless, a simple succession rule is invaluable for both the development and verification of generation algorithms.

In this subsection we develop the formal succession rule for interpart restricted compositions. Thus, given a composition $a_1 \dots a_k$ of n interpart restricted by some function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, we develop a rule that transforms $a_1 \dots a_k$ into its immediate lexicographic successor $S_\sigma(a_1 \dots a_k)$. Knuth explains the general solution to finding the lexicographic successor of some combinatorial pattern $a_1 \dots a_n$ [Knu04d, p.2]:

In general, the lexicographic successor of any combinatorial pattern $a_1 \dots a_n$ is obtainable by a three step procedure:

1. Find the largest j such that a_j can be increased.
2. Increase a_j by the smallest feasible amount.
3. Find the lexicographically least way to extend the new $a_1 \dots a_j$ to a complete combinatorial pattern.

This procedure identifies three separate problems we need to solve to generate the lexicographic successor of a given interpart restricted composition. We shall address these subproblems in turn: we first examine the problem of finding the largest index j such that a_j can be increased, while the resulting composition still has the properties required of an element of $\mathcal{C}_\sigma(n)$. Steps

§ 4.3. Succession Rule

(2) and (3) prove to be interrelated, and reduce to computing the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$; we shall address this problem immediately after solving step (1) in Knuth's three step process in the following lemma.

Lemma 4.1. *For all positive integers n and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, for all $a_1 \dots a_k \in \mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$ there exists some $b_1 \dots b_l \in \mathcal{C}_\sigma(n)$, where $l \geq k-1$, such that $a_j = b_j$ for $1 \leq j \leq k-2$ and $a_{k-1} < b_{k-1}$.*

Proof. Suppose n is an arbitrary positive integer and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary function. Let $a_1 \dots a_k$ be an arbitrary element of $\mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$. Let $b_1 \dots b_l = a_1 \dots a_{k-2} \langle a_{k-1} + a_{k+1} \rangle$. Clearly all parts of $b_1 \dots b_l$ are positive integers, as all parts of $a_1 \dots a_k$ are positive integers. We also know that $b_1 + \dots + b_l = n$, as $a_1 + \dots + a_k = n$. Furthermore, as $\sigma(a_{k-2}) \leq a_{k-1}$, it is apparent that $\sigma(a_{k-2}) \leq a_{k-1} + a_k$, and so $\sigma(b_j) \leq b_{j+1}$ for $1 \leq j < l$. Therefore, $b_1 \dots b_l \in \mathcal{C}_\sigma(n)$. Then, as $a_{k-1} < b_{k-1}$, the proof is complete. \square

In Lemma 4.1 we proved that there is *always* some value that we can add to a_{k-1} to obtain another element of $\mathcal{C}_\sigma(n)$, and so we know that $k-1$ is the largest value j such that a_j can be increased. Thus, step (1) of our three step procedure to find the lexicographic successor of $a_1 \dots a_k$ is solved, and we are now in a position to address steps (2) and (3) directly.

Steps (2) and (3) require that we increase the value of a_{k-1} by the smallest feasible amount and extend $a_1 \dots a_{k-1}$ in the lexicographically least way possible to a complete interpart restricted composition of n . We can equivalently consider this to be problem of generating the lexicographically least composition in the set $\mathcal{C}_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$, as this satisfies both of our requirements: we are increasing a_{k-1} by the smallest increment possible, and we are also computing the lexicographically least extension for the existing sequence. Thus, generating the lexicographic successor of any $a_1 \dots a_k$ in $\mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$, $S_\sigma(a_1 \dots a_k)$, reduces to attaching $M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$ to the end of $a_1 \dots a_{k-2}$. (Recall from Definition 4.2 that the function $M_\sigma(n, m)$ is defined as computing the lexicographically least composition of the set $\mathcal{C}_\sigma(n, m)$.) Formally, we have the following theorem.

§ 4.3. Succession Rule

Theorem 4.4 (Lexicographic Successor). *For all positive integers n and all functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, for all elements of $\mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$,*

$$S_\sigma(a_1 \dots a_k) = a_1 \dots a_{k-2} M_\sigma(a_{k-1} + a_k, a_{k-1} + 1) \quad (4.1)$$

Proof. Suppose n is an arbitrary positive integer and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary function. Let $a_1 \dots a_k$ be an arbitrary element of $\mathcal{C}_\sigma(n) \setminus \{\langle n \rangle\}$. Clearly, there is no positive integer x such that $a_1 \dots a_{k-1} \langle a_k + x \rangle \in \mathcal{C}_\sigma(n)$. By Lemma 4.1 there exists at least one sequence $b_1 \dots b_l \in \mathcal{C}_\sigma(n)$ such that $a_j = b_j$ for $1 \leq j \leq k-2$ and $a_{k-1} < b_{k-1}$. The initial part of $M_\sigma(a_k + a_{k-1}, a_{k-1} + 1)$ is the least possible value we can assign to a_{k-1} ; and the remaining parts (if any) are the lexicographically least way to extend $a_1 \dots a_{k-1}$ to a complete interpart restricted composition of n and σ . Therefore, $S_\sigma(a_1 \dots a_k) = a_1 \dots a_{k-2} M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$, as required. \square

Using (4.1) we can now compute the lexicographic successor of any given composition (except $\langle n \rangle$, which has no successor) from the set $\mathcal{C}_\sigma(n)$. We have shown that computing the lexicographic successor of a given composition essentially reduces to computing the lexicographically least element of an intensionally specified set of interpart restricted compositions. We have not, however, indicated how we might compute this sequence; we address this problem, along with the full implementation of (4.1), in the next subsection.

4.3.2 Algorithm

In the previous subsection we developed an abstract succession rule for interpart restricted compositions; we now turn to the problem of implementing this rule as an iterative generation algorithm. Note that the succession rule we developed did not require that the restriction function be nondecreasing. Just so: (4.1) is an abstract device, and can be applied to any composition of n interpart restricted by an arbitrary function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. The reason that we must impose the nondecreasing requirement on the restriction function is that it is required for computational efficiency. The succession rule relies on the computation of the lexicographically least element of a set of inter-

§ 4.3. Succession Rule

part restricted compositions, and the efficient computation of this sequence requires that the restriction function must be nondecreasing.

Definition 4.2 states that the function $M_\sigma(n, m)$ is defined as computing the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$. In the following theorem we prove the correctness of a simple recurrence equation to compute this sequence for any positive integer n and any nondecreasing restriction function.

Theorem 4.5 (Lexicographic Minimum). *For all positive integers $m \leq n$ and nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, the recurrence*

$$M_\sigma(n, m) = m \cdot M_\sigma(n - m, \sigma(m)) \quad (4.2)$$

holds, where $M_\sigma(n, m) = \langle n \rangle$ if $m + \sigma(m) > n$.

Proof. Suppose that $m \leq n$ are arbitrary positive integers, and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. By the fundamental bijection Theorem 3.2, $m \cdot M_\sigma(n - m, \sigma(m))$ is an element of the set $\mathcal{C}_\sigma(n, m)$, and by Lemma 3.1, $\langle n \rangle$ is always an element of the set $\mathcal{C}_\sigma(n, m)$. Thus, the recurrence (4.2) computes an element of the set $\mathcal{C}_\sigma(n, m)$.

By Corollary 4.1, $\mathcal{C}_\sigma(n, m) = \{\langle n \rangle\}$ if $m + \sigma(m) > n$, and the basis of recurrence (4.2) therefore trivially holds. As m is the minimum value for the initial part of all compositions in $\mathcal{C}_\sigma(n, m)$ by definition, it is clear that the initial part of the lexicographically least element of $\mathcal{C}_\sigma(n, m)$ must be m ; hence, the initial part a_1 of $M_\sigma(n, m)$ is correctly chosen by recurrence (4.2). By the fundamental bijection, Theorem 3.2, the remaining parts $a_2 \dots a_k$ of $M_\sigma(n, m)$ must be from the set $\mathcal{C}_\sigma(n - m, \sigma(m))$, and clearly $a_2 \dots a_k$ must be the lexicographically least element of this set, or $M_\sigma(n - m, \sigma(m))$. Thus, the recurrence (4.2) holds for all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. \square

Less formally, Theorem 4.5 shows that we obtain the lexicographically least composition in $\mathcal{C}_\sigma(n, m)$ by appending m to the lexicographically least composition in $\mathcal{C}_\sigma(n - m, \sigma(m))$. If $m + \sigma(m) > n$ then there are no compositions in $\mathcal{C}_\sigma(n, m)$ with more than one part, leading us to conclude that there

§ 4.3. Succession Rule

is only one possible composition; and this must be the lexicographically least. As an example, let $\sigma(x) = x + 1$ and let us compute the sequence $M_\sigma(10, 2)$. Using recurrence (4.2), we compute $M_\sigma(10, 2) = 2 \cdot M_\sigma(8, 3)$; we then compute $M_\sigma(8, 3) = 3 \cdot M_\sigma(5, 4)$. Evaluating $M_\sigma(5, 4)$ we see that $4 + \sigma(4) > 5$ and so terminate recursion, returning $M_\sigma(5, 4) = \langle 5 \rangle$. Then, gathering the successive initial parts we computed in the previous invocations together, we arrive at $M_\sigma(10, 2) = 235$.

To see why we need to enforce the condition that σ is a nondecreasing function for the recurrence above to work, consider the following example. Using again our contrived restriction function $\sigma(x) = x + 1 + 10[x = 3 \vee x = 4]$, we shall examine the problem of computing $M_\sigma(13, 3)$. As $3 + \sigma(3) \not\leq 13$, we know that 3 is not a feasible initial part; but that does not, in this case, mean that $\langle 13 \rangle$ is the lexicographically least element of $\mathcal{C}_\sigma(13, 3)$. It transpires that $M_\sigma(13, 3) = 58$; to find this composition we must test each value $3 \dots$ until we find a feasible initial part, and *then* recurse to find the remaining parts in the sequence. Clearly, this approach can be significantly less efficient than the direct and predictable method given in (4.2). Thus, solving the problem of computing $M_\sigma(n, m)$ in the full generality of the framework would have unacceptable consequences on the efficiency of generation for the instances where we do not need to search for feasible initial parts: namely, those instances where the restriction function is nondecreasing.

Accepting the limitations of what we can generate efficiently under the current formulation of the interpart restricted compositions framework, we now move on to implementing (4.2) as an iterative algorithm. (As we have previously stressed, for all of the examples given in Chapter 3, the corresponding restriction function is nondecreasing, and many of these classes of restricted partition currently have no generation algorithm.) This algorithm, LEXMIN_σ (Algorithm 4.2), is quite simple, but as it is the nucleus of the direct implementation of the lexicographic succession rule, we shall consider it in some detail.

LEXMIN_σ directly computes the value $M_\sigma(n, m)$ for any nondecreasing function σ — the invocation $\text{LEXMIN}_\sigma(n, m)$ returns the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$. LEXMIN_σ is a direct iterative implemen-

§ 4.3. Succession Rule

Algorithm 4.2 $\text{LEXMIN}_\sigma(n, m)$

Require: $1 \leq m \leq n$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is nondecreasing

```
1:  $k \leftarrow 1$ 
2:  $x \leftarrow m$ 
3:  $y \leftarrow n - m$ 
4: while  $\sigma(x) \leq y$  do
5:    $a_k \leftarrow x$ 
6:    $x \leftarrow \sigma(x)$ 
7:    $y \leftarrow y - x$ 
8:    $k \leftarrow k + 1$ 
9: end while
10:  $a_k \leftarrow x + y$ 
11: return  $a_1 \dots a_k$ 
```

tation of the recursive rule given in (4.2), and works in precisely the same fashion. Specifically, we assign $a_1 \leftarrow m$, and then iteratively implement the recursive rule until we have reached a value that is not a feasible initial part. We then insert the singleton composition into the appropriate index and return the resulting composition. The variables in LEXMIN_σ are arranged in a slightly different manner from our previous discussions: we assign $x \leftarrow m$, and use this variable to keep track of the value for the next value for a_k . We also set $y \leftarrow n - m$; the test $m + \sigma(m) \leq n$ is now equivalent to $\sigma(x) \leq y$, and so requires one less memory access and addition per iteration of the while loop. We shall now prove the correctness of LEXMIN_σ .

Theorem 4.6. *For all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $\text{LEXMIN}_\sigma(n, m) = M_\sigma(n, m)$.*

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function, and proceed by strong induction on n .

Base case: $n = 1$. Since $1 \leq m \leq n$, we know that $m = 1$. Upon entry, we assign $x \leftarrow 1$ and $y \leftarrow 0$. As the codomain of σ is \mathbb{Z}^+ , the loop entry condition on line 4 fails, and so we proceed directly to line 10. We then assign $a_1 \leftarrow 1 + 0$, and return the composition $\langle 1 \rangle$. Clearly $M_\sigma(1, 1) = \langle 1 \rangle$ and therefore $\text{LEXMIN}_\sigma(1, 1) = M_\sigma(1, 1)$, as required.

§ 4.3. Succession Rule

Induction step: Suppose, for some positive integer n , $\text{LEXMIN}_\sigma(n', m') = \text{M}_\sigma(n', m')$ for all positive integers $m' \leq n' < n$. Consider the invocation $\text{LEXMIN}_\sigma(n, m)$, for an arbitrary positive integer $m \leq n$.

Suppose $m + \sigma(m) > n$. Upon entry of the algorithm we assign $x \leftarrow m$ and $y \leftarrow n - m$, and so $m + \sigma(m) > n \iff \sigma(x) > y$. Hence, the loop entry condition on line 4 fails, and we immediately proceed to line 10. We then assign $a_1 \leftarrow m + n - m$, and return the composition $\langle n \rangle$. By (4.2), $\text{M}_\sigma(n, m) = \langle n \rangle$, and hence $\text{LEXMIN}_\sigma(n, m) = \text{M}_\sigma(n, m)$, as required.

Suppose that $m + \sigma(m) \leq n$. Upon entry of the algorithm we assign $x \leftarrow m$ and $y \leftarrow n - m$, and so $m + \sigma(m) \leq n \iff \sigma(x) \leq y$. Therefore, we enter the loop and assign $a_1 \leftarrow m$. Upon reaching line 8, the variables have the following states: $x = \sigma(m)$, $y = n - m - \sigma(m)$ and $k = 2$. As we have $a_1 = m$, we have correctly assigned the initial part by (4.2) during the first iteration of the loop: the remaining parts, i.e. the composition $\text{M}_\sigma(n - m, \sigma(m))$, must clearly be assigned in the remaining iterations.

By the inductive hypothesis, $\text{LEXMIN}_\sigma(n - m, \sigma(m)) = \text{M}_\sigma(n - m, \sigma(m))$. Therefore we must show that the variables x and y are in the same state immediately *after* the first iteration of the loop in $\text{LEXMIN}_\sigma(n, m)$ and immediately *before* the first iteration of the loop in $\text{LEXMIN}_\sigma(n - m, \sigma(m))$. Upon entry to $\text{LEXMIN}_\sigma(n - m, \sigma(m))$, we set $x \leftarrow \sigma(m)$ and $y \leftarrow n - m - \sigma(m)$; and these are the same values held by the corresponding variables after the first iteration of the loop in $\text{LEXMIN}_\sigma(n, m)$. Thus, by the inductive hypothesis, $\text{LEXMIN}_\sigma(n, m) = \text{M}_\sigma(n, m)$. Therefore, as m is an arbitrary positive integer such that $m \leq n$, $\text{LEXMIN}_\sigma(n, m) = \text{M}_\sigma(n, m)$ for all m ; hence $\text{LEXMIN}_\sigma(n, m) = \text{M}_\sigma(n, m)$ for all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. \square

Having proved the correctness of our procedure for computing the lexicographically least element of $\mathcal{C}_\sigma(n, m)$, we can now move on to the algorithm to generate $\mathcal{C}_\sigma(n, m)$ in lexicographic order, RULEGEN_σ (Algorithm 4.3). Each iteration of the main loop (lines 5 to 15) in RULEGEN_σ implements a single application of the succession rule (4.1). On lines 5 and 7 we assign the values of $a_k - 1$ and $a_{k-1} + 1$ to variables y and x respectively. Then, on lines 8

§ 4.3. Succession Rule

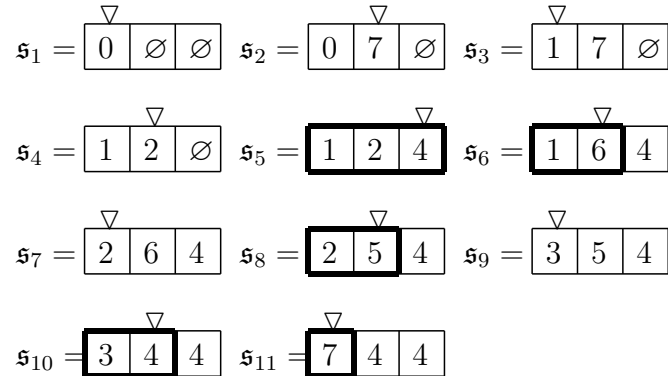


Figure 4.2: Array state-transition diagram for the lexicographic succession rule generation algorithm for interpart restricted compositions. Indices marked with the ∇ symbol are the indices in which a value is assigned to change the state of the array, and array segments outlined in bold are the portions that are visited.

to 14 we attach $M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$ after a_{k-2} , and on line 15 we visit the resulting composition, making it available to the consuming procedure.

An example of RULEGEN_σ generating a complete set of interpart restricted compositions is given in Figure 4.2. In this figure we see the sequence of states the generation array goes through during the execution of $\text{RULEGEN}_\sigma(7, 1)$ when $\sigma(x) = x + 1$. States \mathfrak{s}_1 and \mathfrak{s}_2 correspond to the initialisation steps of the algorithm, where we insert the appropriate values into the array to ensure that the first composition visited is $M_\sigma(7, 1)$. States \mathfrak{s}_3 – \mathfrak{s}_5 correspond to the insertion of $M_\sigma(7, 1) = 124$ into the array, and this composition is subsequently visited. Returning to the head of the main loop, we discover that $\sigma(x) \not\leq y$, and so we do not enter the inner loop, and instead assign $a_2 \leftarrow 6$, giving us state \mathfrak{s}_6 , which is then visited. Execution continues along these lines until we reach \mathfrak{s}_{11} , where we visit the singleton composition $\langle 7 \rangle$ and terminate.

We formally prove the correctness of RULEGEN_σ in the following theorem. In the proof we use the concept of the *rank* [KS98, p.31–32] of an interpart restricted composition. The rank of a combinatorial object is the number of objects that precede it in the listing order. Hence, in this context, we see

§ 4.3. Succession Rule

Algorithm 4.3 $\text{RULEGEN}_\sigma(n, m)$

Require: $1 \leq m \leq n$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is nondecreasing

```

1:  $k \leftarrow 2$ 
2:  $a_1 \leftarrow m - 1$ 
3:  $a_2 \leftarrow n - m + 1$ 
4: while  $k \neq 1$  do
5:    $y \leftarrow a_k - 1$ 
6:    $k \leftarrow k - 1$ 
7:    $x \leftarrow a_k + 1$ 
8:   while  $\sigma(x) \leq y$  do
9:      $a_k \leftarrow x$ 
10:     $x \leftarrow \sigma(x)$ 
11:     $y \leftarrow y - x$ 
12:     $k \leftarrow k + 1$ 
13:  end while
14:   $a_k \leftarrow x + y$ 
15:  visit  $a_1 \dots a_k$ 
16: end while

```

that the rank of $M_\sigma(n, m)$ is 0, and the rank of $\langle n \rangle$ is $C_\sigma(n, m) - 1$, and if any given composition $a_1 \dots a_k \in C_\sigma(n, m) \setminus \{\langle n \rangle\}$ has rank r , then the rank of $S_\sigma(a_1 \dots a_k)$ must be $r + 1$.

Theorem 4.7. *For every nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers $m \leq n$, $\text{RULEGEN}_\sigma(n, m)$ generates the set $C_\sigma(n, m)$ in lexicographic order.*

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function, and suppose $m \leq n$ are arbitrary positive integers. We proceed by induction on the rank r of the elements of $C_\sigma(n, m)$.

Base case: $r = 0$. Upon initialisation of $\text{RULEGEN}_\sigma(n, m)$ we enter the main loop, and set $x \leftarrow m$, $y \leftarrow n - m$ and $k \leftarrow 1$ before reaching line 8. These values are identical to the values of the corresponding variables in the invocation $\text{LEXMIN}_\sigma(n, m)$ before reaching line 4 in that algorithm. Since lines 4–10 in LEXMIN_σ and lines 8–14 in RULEGEN_σ are identical, we know that $a_1 \dots a_k = M_\sigma(n, m)$ upon reaching line 15 in the first iteration of

§ 4.3. Succession Rule

$\text{RULEGEN}_\sigma(n, m)$ by Theorem 4.6. Therefore, the composition at rank $r = 0$ is correctly visited.

Induction step: Suppose for some $r \leq C_\sigma(n, m) - 1$ that the composition $a_1 \dots a_k$ has been visited by $\text{RULEGEN}_\sigma(n, m)$. Let us examine the next iteration of the main loop.

Suppose that $r = C_\sigma(n, m) - 1$. The lexicographically greatest element of $\mathcal{C}_\sigma(n, m)$ is $\langle n \rangle$, and therefore $k = 1$. Upon returning to line 4 we see that the loop entry condition is false, and terminate. Therefore, if we have visited the lexicographically last composition, we terminate correctly.

Suppose that $r < C_\sigma(n, m) - 1$. By our argument above, $k \geq 2$, and so we return to the head of the loop, setting $x \leftarrow a_{k-1} + 1$, $y \leftarrow a_k - 1$ and $k \leftarrow k - 1$. By the analogy with LEXMIN_σ we will then insert $M_\sigma(a_k + a_{k-1}, a_{k-1} + 1)$ into array indices from $k - 1$ onwards, leaving indices $\leq k - 2$ unmodified. Therefore, we will visit the composition $a_1 \dots a_{k-2} M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$ on line 15, which is the immediate successor of $a_1 \dots a_k$ by Theorem 4.4.

Therefore, $\text{RULEGEN}_\sigma(n, m)$ visits all elements of $\mathcal{C}_\sigma(n, m)$ in lexicographic order. Hence, the algorithm correctly generates the set $\mathcal{C}_\sigma(n, m)$ for any nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers $m \leq n$. \square

Having proved the correctness of the direct implementation of the lexicographic succession rule, we can now move on to the analysis of this algorithm. We know that we can generate all elements of the set $\mathcal{C}_\sigma(n, m)$, but can we do so efficiently? It is this question that we address in the next subsection.

4.3.3 Analysis

In the analysis of RULEGEN_σ we wish to determine the total number of ‘read’ and ‘write’ operations [Kem98] required to generate all compositions of n , interpart restricted by a nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. The primitive operations in question occur when we read from or write to the generation array; that is, a statement of the form $x \leftarrow a_j$ is counted as a single read operation, and a statement $a_j \leftarrow x$ represents a single write. We shall proceed by determining the frequency of certain key instructions in

§ 4.3. Succession Rule

the algorithm, from which determining the total number of read and write operations is then a rudimentary application of Kirchhoff's Law [Knu72].

The key to analysing RULEGEN_σ is to note that the k variable is modified only via increment (line 12) and decrement (line 6) operations. Then, as k is initialised to 2 and the algorithm terminates when $k = 1$, we know that there must be one more decrement than increments performed on the variable. Formally, let $t_6(n, m)$ and $t_{12}(n, m)$ be the number of times lines 6 and 12 are executed, respectively, in the process of generating the set $\mathcal{C}_\sigma(n, m)$ using RULEGEN_σ (Algorithm 4.3); we then have the following lemmas.

Lemma 4.2. *The number of times line 6 is executed during the execution of Algorithm 4.3 is given by $t_6(n, m) = C_\sigma(n, m)$.*

Proof. By Theorem 4.7, Algorithm 4.3 correctly generates the set $\mathcal{C}_\sigma(n, m)$, and so line 15 must be executed $C_\sigma(n, m)$ times. By Kirchhoff's Law, line 6 is executed as many times as line 15, and so $t_6(n, m) = C_\sigma(n, m)$. \square

Lemma 4.3. *The number of times line 12 is executed during the execution of Algorithm 4.3 is given by $t_{12}(n, m) = C_\sigma(n, m) - 1$.*

Proof. Algorithm 4.3 begins with $k = 2$ and terminates when $k = 1$, and k is modified only on lines 6 and 12. The statement $k \leftarrow k - 1$ is executed $C_\sigma(n, m)$ times by Lemma 4.2; therefore, the statement $k \leftarrow k + 1$ (line 12) must be executed $C_\sigma(n, m) - 1$ times. Hence, $t_{12}(n, m) = C_\sigma(n, m) - 1$. \square

Using the frequency counts of Lemmas 4.2 and 4.3 we can now count the total number of read and write operations required to generate the set $\mathcal{C}_\sigma(n, m)$ using Algorithm 4.3.

Theorem 4.8. *Algorithm 4.3 requires $R_{A4.3}(n, m) = 2C_\sigma(n, m)$ read operations to generate the set $\mathcal{C}_\sigma(n, m)$.*

Proof. Read operations are carried out on lines 5 and 7 which, by Lemma 4.2, are executed $C_\sigma(n, m)$ times each. Thus, the total number of read operations is $R_{A4.3}(n) = 2C_\sigma(n, m)$. \square

Theorem 4.9. *Algorithm 4.3 requires $W_{A4.3}(n, m) = 2C_\sigma(n, m) - 1$ write operations to generate the set $\mathcal{C}_\sigma(n, m)$, excluding initialisation.*

§ 4.3. Succession Rule

Proof. Write operations are carried out in Algorithm 4.3 on lines 9 and 14. By Lemma 4.3, line 9 is executed $C_\sigma(n, m) - 1$ times, and by Lemma 4.2, line 14 is executed $C_\sigma(n, m)$ times. Thus, $W_{A4.3}(n, m) = 2C_\sigma(n, m) - 1$. \square

Thus, as we require a total of $2C_\sigma(n, m)$ read and $2C_\sigma(n, m) - 1$ write operations, we can see that we require an average of two read and two write operations (in an amortised sense [CLR90, ch.18]) to generate each element of $\mathcal{C}_\sigma(n, m)$, for any nondecreasing instance of the restriction function. Regardless of whether we are generating unrestricted compositions or Göllnitz-Gordon partitions (see Section 3.1.3), we require only two read and two write operations, on average, for each composition generated.

Counting the read and write operations incurred during the execution of an algorithm may seem like a rather drastic over-simplification of the computational cost of generation. Although we address this problem in greater detail in Chapter 5, a word of justification to allay the reader's misgivings is warranted at this juncture. If we consider RULEGEN_σ again for a moment, we can see that read and write operations occur within both of the loops of the algorithm, and all operations within these loops are certainly constant time operations — assuming, of course, that the restriction function σ is a constant time operation. Thus the total execution time of the algorithm will be proportional to the total number of read and write operations.

Counting read and write operations, however, also provides us with a deeper analysis of the problem of generation: it provides us with a measure of the *complexity* of the problem [Kem98]. Algorithm 4.3 is only one particular implementation of the lexicographic succession rule (4.1), and one that assumes the restriction function is nondecreasing. But it is a direct and literal implementation of this rule, and hence provides us with a valuable worst-case measure of the cost of generating interpart restricted compositions. Therefore, our theoretical model, our “simplified idealisation of reality” [KY76], should be a reasonable reflection of actual computational experience. Then, to simplify matters even further, we can say that generating all interpart restricted compositions of n requires $O(C_\sigma(n))$ read and $O(C_\sigma(n))$ write operations. Moreover, this is true for any instance of the

§ 4.3. Succession Rule

restriction function, nondecreasing or otherwise. Setting aside the details of how we might achieve this, it seems reasonable to assume that we can always determine the next feasible initial part in constant time. Thus, with minor modifications to RULEGEN_σ , we can generate all interpart restricted compositions for an arbitrary restriction function in $O(C_\sigma(n))$ read and $O(C_\sigma(n))$ write operations.

It is possible to improve on this complexity by noting a special case in the succession rule. The special case allows us to generate the lexicographic successor of certain compositions in exactly two write and zero read operations. Making this observation reduces both the complexity of generation in terms of decreasing the total number of read operations, and the actual computational cost of generating all interpart restricted compositions. We develop this idea fully in the next section. First, however, to gain a little perspective on the efficiency of RULEGEN_σ we shall perform a short empirical analysis of some algorithms from the literature that generate some subset of the interpart restricted compositions.

4.3.4 Comparison with Existing Algorithms

Chapter 5 is entirely dedicated to the task of comparing concrete instantiations of the algorithms we develop in this chapter with existing descending composition generation algorithms. Generating all partitions is an important problem [Knu04c], and so we conduct our comparison of ascending composition and descending composition generation algorithms at length. There are, however, some existing algorithms that are also capable of generating some subset of the interpart restricted compositions. Thus, in this subsection we briefly compare these algorithms with our RULEGEN_σ to provide a degree of perspective on its efficiency, relative to existing algorithms.

The first algorithm we consider is Nārāyaṇa’s algorithm to generate all unrestricted compositions of n in reverse lexicographic order, as presented by Knuth [Knu04f, ex.15]. Although other algorithms exist to generate unrestricted compositions [Knu04b, ex.12], Nārāyaṇa’s algorithm is the only true generator available that generates all unrestricted compositions lexico-

§ 4.3. Succession Rule

	$\sigma(x) = 1$			$\sigma(x) = x + 1$			$\sigma(x) = x + 2$		
n	27 6×10^7	28 1×10^8	30 5×10^8	164 5×10^7	175 1×10^8	201 5×10^8	206 5×10^7	220 1×10^8	252 5×10^8
J	0.86	0.85	0.86	0.53	0.50	0.49	0.58	0.54	0.53
C	0.67	0.68	0.69	0.55	0.55	0.54	0.59	0.58	0.57
	<i>Nārāyaṇa [Knu04f, ex.15]</i>			<i>Riha & James [RJ76]</i>			<i>Riha & James [RJ76]</i>		

Table 4.1: A comparison the rule-based algorithm to generate interpart restricted compositions with Nārāyaṇa’s and Riha & James’ algorithms. The figures provided are the time required by RULEGEN_σ to generate the specified set of compositions divided by the time required by the particular algorithm.

graphically. Thus, we shall compare Nārāyaṇa’s algorithm with a concrete instantiation of RULEGEN_σ where we replace all instances of $\sigma(x)$ with 1.

The second algorithm we shall compare, and the only available algorithm that generates any substantial subset of the interpart restricted compositions, is due to Riha & James [RJ76]. Riha & James’ algorithm generates descending k -compositions in reverse lexicographic order, with a flexible class of restrictions on the parts. The particular restriction that we are interested in here allows us to specify a minimum difference between consecutive pairs of parts, thus allowing us to generate, for instance, the partitions into distinct parts and Rogers-Ramanujan partitions. We therefore compare Riha and James’ algorithm with concrete instantiations of RULEGEN_σ where we replace all instances of $\sigma(x)$ with $x + 1$ and $x + 2$.

The results of these comparisons are given in Table 4.1, where we report the ratio of the total time required by RULEGEN_σ , divided by the total time required by the other algorithms, to generate the compositions in question for a range of values of n . We shall explain the methodology used to obtain such results in detail in Section 4.4.4, but for now we shall simply note that the algorithms were implemented in the Java and C languages, and the minimum of five runs of each algorithm is used as the definitive duration for each value of n . The values of n are chosen such that n is smallest integer where $C_\sigma(n) > 5 \times 10^7$, $C_\sigma(n) > 1 \times 10^7$ and $C_\sigma(n) > 5 \times 10^8$.

§ 4.4. Accelerated Algorithm

We can immediately see from Table 4.1 that RULEGEN_σ is substantially more efficient than the competing methods. In the case of Nārāyaṇa’s algorithm, the difference between the algorithms is a constant factor of 14% in the Java implementations and around 30% in the C versions. This is consistent with Nārāyaṇa’s algorithm being constant amortised time, but clearly the hidden constant in RULEGEN_σ is smaller, resulting in a more efficient algorithm. Riha & James’ algorithm is *not* constant amortised time, and this is reflected in Table 4.1. In the cases of both the distinct and Rogers-Ramanujan partitions of n , the difference between the algorithms increases for the number of compositions we generating.

The purpose of this section has not been to definitively establish that RULEGEN_σ is more efficient than either Nārāyaṇa’s or Riha & James’ algorithms: to do so would require a much deeper analysis. The purpose, rather, has been to illustrate that RULEGEN_σ is an efficient algorithm in its own right, and can certainly generate compositions in time comparable to more specific algorithms from the literature. Having made this point, we now move on to improving RULEGEN_σ , not for any particular instance of the restriction function, but for *all* instances. We do this by developing the auxiliary theory of ‘terminal’ and ‘nonterminal’ compositions, and using this theory to identify some common structure within the set of interpart restricted compositions.

4.4 Accelerated Algorithm

Lexicographic succession rules are an important theoretical device by which we may determine an upper bound on the complexity of generating a particular combinatorial object [Kem98], but literal implementations of these rules can lead to inefficient algorithms. (We shall see an example of this in Section 5.3.2.) A more efficient generation algorithm may be possible if we can identify some special cases in the succession rule that can be implemented efficiently.

In this section we identify special cases in the lexicographic succession rule for interpart restricted compositions. These special cases are based on

§ 4.4. Accelerated Algorithm

the observation that we can often generate the successor of a given interpart restricted composition by simply subtracting 1 from the last part, and adding 1 to the second-last part. In Section 4.4.1 we formalise this idea and develop the basic theory required to formulate, prove the correctness of, and analyse the resulting algorithm. Then, in Section 4.4.2, we present the algorithm itself and formally prove its correctness. Following this, in Section 4.4.3 we analyse the algorithm in terms of the total number of read and write operations incurred in the process of generating all interpart restricted compositions of n for a nondecreasing restriction function.

4.4.1 Basic Principle

The special case that we identify in our general succession rule for interpart restricted compositions is most easily seen by means of an example. In Figure 4.3(a) the compositions of 12 interpart restricted by the function $\sigma(x) = x + 1$ (i.e. the distinct partitions) are shown. Considering, for a moment, the lexicographically first compositions in this set, 1236 and 1245, it is not difficult to see that we can efficiently obtain the latter from the former: we simply set $a_3 \leftarrow 4$ and $a_4 \leftarrow 5$. Furthermore, if we examine the composition that immediately follows 1245, it is also not difficult to see that we can obtain 129 by adding the last two parts of the preceding composition together. This is the essential principle upon which the accelerated algorithm of this section operates; but to fully develop, prove the correctness of, and analyse this algorithm we must develop some auxiliary theory first.

The theory behind the accelerated algorithm of this section relies on identifying two disjoint subsets of the set $\mathcal{C}_\sigma(n)$, which together comprise the entire set. We shall refer to these disjoint subsets of $\mathcal{C}_\sigma(n)$ as the *terminal* and *nonterminal* compositions of n and σ . The properties required for a composition to be terminal or nonterminal are formally defined as follows.

Definition 4.5 (Terminal Composition). *For some positive integer n and a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, a composition $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$ is terminal if $k = 1$ or $\sigma(a_{k-1}) + \sigma(\sigma(a_{k-1})) \leq a_k$. Let $\mathcal{T}_\sigma(n, m)$ denote the set of terminal compositions in $\mathcal{C}_\sigma(n, m)$, and $T_\sigma(n, m)$ denote the cardinality of this set (i.e.*

§ 4.4. Accelerated Algorithm

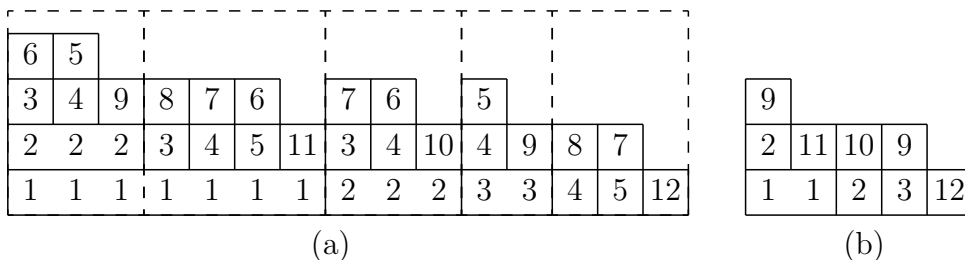


Figure 4.3: Composition blocks and terminal compositions. In (a) we see the ascending compositions into distinct parts of 12 with block-boundaries, and in (b) we see the terminal compositions of this set.

$$T_\sigma(n, m) = |\mathcal{T}_\sigma(n, m)|.$$

Definition 4.6 (Nonterminal Composition). *For some positive integer n and a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, a composition $a_1 \dots a_k \in \mathcal{C}_\sigma(n)$ is nonterminal if $k \neq 1$ and $\sigma(a_{k-1}) + \sigma(\sigma(a_{k-1})) > a_k$. Let $\mathcal{N}_\sigma(n, m)$ denote the set of nonterminal compositions in $\mathcal{C}_\sigma(n, m)$, and $N_\sigma(n, m)$ denote the cardinality of this set (i.e. $N_\sigma(n, m) = |\mathcal{N}_\sigma(n, m)|$).*

From Definitions 4.5 and 4.6 it is easy to see that $\mathcal{T}_\sigma(n) \cup \mathcal{N}_\sigma(n) = \mathcal{C}_\sigma(n)$. The properties in question are perhaps a little opaque, and we shall therefore examine the basic motivation for the definitions before moving to developing the required theoretical results.

The motivation behind the properties required for a composition to be terminal or nonterminal is simple: we wish to determine if $M_\sigma(a_k + a_{k-1}, a_{k-1} + 1)$ has two or fewer parts. Since computing the lexicographically least composition of the set $\mathcal{C}_\sigma(a_k + a_{k-1}, a_{k-1} + 1)$ is the nucleus of the procedure for generating the lexicographic successor of $a_1 \dots a_k$, then clearly we can improve over the general case if we know that this sequence has two or fewer parts. In the following lemma we formalise this notion for the case where the lexicographically least element of the set $M_\sigma(n, m)$ has exactly two parts.

Lemma 4.4. *For all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers n and m such that $m + \sigma(m) \leq n$ and $m + \sigma(m) + \sigma(\sigma(m)) > n$, $M_\sigma(m, m) = \langle m \rangle \langle n - m \rangle$.*

§ 4.4. Accelerated Algorithm

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function, and suppose n and m are arbitrary positive integers such that $m + \sigma(m) \leq n$ and $\sigma(m) + \sigma(\sigma(m)) > n - m$. Let $a_1 \dots a_k = M_\sigma(n, m)$. Since $m + \sigma(m) \leq n$, we know that $a_1 = m$ and $a_2 \dots a_k = M_\sigma(n - m, \sigma(m))$ by Theorem 4.5. Then, examining the value of $M_\sigma(n - m, \sigma(m))$, we see that, since $\sigma(m) + \sigma(\sigma(m)) > n - m$ we must have $a_2 \dots a_k = \langle n - m \rangle$ by Theorem 4.5. Therefore, $M_\sigma(m, m) = \langle m \rangle \langle n - m \rangle$, as required. \square

Using Lemma 4.4 we can now compress the generic succession rule into two efficiently computable alternatives that can be applied to nonterminal compositions. The key to this rule is that we know the lexicographic successor has no more than two parts (which, as we have seen, is the effective definition of the nonterminal property); having this knowledge, we can implement the succession rule on nonterminal compositions quite efficiently. The formal rules are codified in the following theorem.

Theorem 4.10 (Nonterminal Successor). *For all positive integers n and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, for all $a_1 \dots a_k \in \mathcal{N}_\sigma(n)$,*

$$S_\sigma(a_1 \dots a_k) = \begin{cases} a_1 \dots a_{k-2} \langle a_{k-1} + 1 \rangle \langle a_k - 1 \rangle & \text{if } \sigma(a_{k-1} + 1) \leq a_k - 1; \\ a_1 \dots a_{k-2} \langle a_{k-1} + a_k \rangle & \text{otherwise.} \end{cases}$$

Proof. Suppose n is an arbitrary positive integer and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. Suppose $a_1 \dots a_k$ is an arbitrary element of $\mathcal{N}_\sigma(n)$. By the generic succession rule of Theorem 4.4, $S_\sigma(a_1 \dots a_k) = a_1 \dots a_{k-2} M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$. Clearly, the first $k - 2$ parts of the sequence computed by the nonterminal succession rule above and the generic succession rule are equal, and so it remains to show that $M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$ is equal to $\langle a_{k-1} + 1 \rangle \langle a_k - 1 \rangle$ if $\sigma(a_{k-1} + 1) \leq a_k - 1$ or $\langle a_{k-1} + a_k \rangle$ if $\sigma(a_{k-1} + 1) > a_k - 1$.

Suppose $\sigma(a_{k-1} + 1) \leq a_k - 1$. Since $a_1 \dots a_k$ is nonterminal, $\sigma(a_{k-1}) + \sigma(\sigma(a_{k-1})) > a_k$, and as σ is nondecreasing, we can deduce that $\sigma(a_{k-1} + 1) + \sigma(\sigma(a_{k-1} + 1)) > a_k - 1$. Then, adding $a_{k-1} + 1$ to both sides of this inequality, we see that $a_{k-1} + 1 + \sigma(a_{k-1} + 1) + \sigma(\sigma(a_{k-1} + 1)) > a_{k-1} + a_k$, and therefore,

§ 4.4. Accelerated Algorithm

by Lemma 4.4, we see that $M_\sigma(a_{k-1} + a_k, a_{k-1} + 1) = \langle a_{k-1} + 1 \rangle \langle a_k - 1 \rangle$, as required.

Suppose $\sigma(a_{k-1} + 1) > a_k - 1$. Then by Theorem 4.5, $M_\sigma(a_{k-1} + a_k, a_{k-1} + 1) = \langle a_{k-1} + a_k \rangle$, as required. Therefore, the nonterminal succession rule above is verified. \square

Theorem 4.10 provides us with sufficient information to derive an efficient generation algorithm based on the codified succession rules for nonterminal compositions. To analyse the resulting algorithm we require one further formal result. In the generation algorithm we treat the cases of terminal and nonterminal compositions differently: the nonterminal compositions via a specialised internal loop, and the terminal compositions via a modified version of the standard lexicographic succession rule. To determine the actual efficiency of this approach we require a means of enumerating terminal and nonterminal compositions. We shall approach this problem by counting the terminal compositions, $T_\sigma(n, m)$, via a modified version of the recurrence used to enumerate interpart restricted compositions. This provides us with a complete solution to the problem of counting terminal and nonterminal compositions, since the number of nonterminal compositions is given by $N_\sigma(n, m) = C_\sigma(n, m) - T_\sigma(n, m)$.

Counting the terminal compositions of n , m and σ proves to be quite similar to counting interpart restricted compositions in general. Consider again the example given in Figure 4.3 (p.97), where we show the distinct partitions of 12 and the block structure imposed by terminal and nonterminal compositions on this set. Figure 4.3(b) contains the terminal compositions of 12 and $\sigma(x) = x + 1$; these can be seen to share a similar structure as the overall set, except fewer values are feasible initial parts. We use this property in the following theorem to enumerate terminal compositions.

Theorem 4.11. *For all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ $T_\sigma(n, m)$ satisfies the recurrence*

$$T_\sigma(n, m) = T_\sigma(n - m, \sigma(m)) + T_\sigma(n, m + 1) \quad (4.3)$$

§ 4.4. Accelerated Algorithm

with $T_\sigma(n, m) = 1$ if $m + \sigma(m) + \sigma(\sigma(m)) > n$.

Proof. The general case of (4.3) is similar to the general case of (3.8) and can be verified using precisely the same techniques. It remains to demonstrate that $T_\sigma(n, m) = 1$ if $m + \sigma(m) + \sigma(\sigma(m)) > n$.

Suppose $a_1 a_2 \in \mathcal{C}_\sigma(n, m)$ and $m + \sigma(m) + \sigma(\sigma(m)) > n$, and suppose that $a_1 = m$, and therefore, $a_2 = n - m$. As $m + \sigma(m) + \sigma(\sigma(m)) > n$, we see that $a_1 + \sigma(a_1) + \sigma(\sigma(a_1)) > a_2 + a_1$, and so $\sigma(a_1) + \sigma(\sigma(a_1)) > a_2$. Therefore, $a_1 a_2$ is nonterminal. Since σ is nondecreasing, we can see that any $a_1 a_2$ such that $a_1 = m + x$ for some positive integer x must also be nonterminal. Therefore, there are no terminal compositions in $\mathcal{C}_\sigma(n, m)$ with two or more parts. Thus, the only remaining composition is the singleton composition $\langle n \rangle$, and this is terminal by definition. Hence, $T_\sigma(n, m) = 1$ if $m + \sigma(m) + \sigma(\sigma(m)) > n$. \square

It would be more satisfactory to derive $T_\sigma(n, m)$ in terms of the total number of interpart restricted compositions $C_\sigma(n, m)$; unfortunately, there does not appear to be any simple relationship between these numbers in general. In some particular cases the relationship is apparent — for instance, in the case of the unrestricted compositions of n , exactly half of the compositions in the set are terminal. It is not difficult to prove this assertion in the particular case when $\sigma(x) = 1$, but for more complex instances of the restriction function, the relationship between $T_\sigma(n, m)$ and $C_\sigma(n, m)$ is more elaborate. In the next chapter we consider this question again for the unrestricted partitions of n , and prove that the number of nonterminal partitions of n is given by $p(n) - p(n - 2)$, where $p(n)$ is the number of unrestricted partitions of n .

Having developed the theoretical tools required for the construction and analysis of our accelerated algorithm to generate interpart restricted compositions, we shall now move on and consider the generation algorithm itself in the next subsection. In this algorithm we utilise the lexicographic succession rules for nonterminal compositions to both decrease the average cost of a write operation in the algorithm and to reduce the number of read operations required to exactly $T_\sigma(n, m)$.

§ 4.4. Accelerated Algorithm

4.4.2 Algorithm

The direct implementation of the lexicographic succession rule for interpart restricted compositions, RULEGEN_σ , generates the successor of $a_1 \dots a_k$ by computing the lexicographically least element of the set $\mathcal{C}_\sigma(a_{k-1}+a_k, a_{k-1}+1)$, and visiting the resulting composition. The algorithm operates by implementing exactly one transition per iteration of the main loop. The accelerated algorithm, ACCELGEN_σ , developed in this subsection operates on a slightly different principle: we compute the lexicographically least composition of $\mathcal{C}_\sigma(a_{k-1}+a_k, a_{k-1}+1)$, as before, but we now keep a watchful eye to see if the resulting composition is nonterminal. If it is we know we can compute the lexicographic successor efficiently, and furthermore, if its successor, in turn, is nonterminal, we can repeat the process. Otherwise, we revert to the standard means of computing the lexicographic successor.

To implement this principle we require a modified version of our algorithm to compute the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$, one that identifies nonterminal compositions. We could, of course, use the standard means and test the resulting composition to see whether it is nonterminal: if it is, we can apply the nonterminal succession rules, if not, continue as before. Such an algorithm is, however, inelegant as it involves reading values back from the generation array and resetting the variables x and y (which are used to hold the values of a_{k-1} and a_k) back to the appropriate values. We have all of the information that we require if we terminate the loop used to generate $M_\sigma(n, m)$ one iteration early, and this is the approach that we shall take. The resulting algorithm is more direct and efficient than the more obvious approach outlined above.

The key segment of our generation algorithm is again the procedure used to compute $M_\sigma(n, m)$, and so we shall follow the same pattern in developing the generation algorithm as in the previous section. That is, we shall first develop and prove the correctness of the procedure required to generate the lexicographically least element of the set $\mathcal{C}_\sigma(n, m)$, and then use this procedure to define the generation algorithm itself. Our modified procedure to compute $M_\sigma(n, m)$ is given in Algorithm 4.4, and it operates as follows.

§ 4.4. Accelerated Algorithm

Algorithm 4.4 $\text{LEXMIN}'_{\sigma}(n, m)$

Require: $1 \leq m \leq n$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is nondecreasing

```
1:  $k \leftarrow 1$ 
2:  $x \leftarrow m$ 
3:  $y \leftarrow n - m$ 
4: while  $\sigma(x) + \sigma(\sigma(x)) \leq y$  do
5:    $a_k \leftarrow x$ 
6:    $x \leftarrow \sigma(x)$ 
7:    $y \leftarrow y - x$ 
8:    $k \leftarrow k + 1$ 
9: end while
10: if  $\sigma(x) \leq y$  then
11:    $a_k \leftarrow x$ 
12:    $a_{k+1} \leftarrow y$ 
13:   return  $a_1 \dots a_{k+1}$ 
14: end if
15:  $a_k \leftarrow x + y$ 
16: return  $a_1 \dots a_k$ 
```

Consider the invocation $\text{LEXMIN}'_{\sigma}(n, m)$, which computes $M_{\sigma}(n, m)$. As before, upon entering the algorithm we assign $x \leftarrow m$ and $y \leftarrow n - m$. When we reach line 4, however, we notice a difference between LEXMIN_{σ} and LEXMIN'_{σ} : the loop entry condition has changed from $\sigma(x) \leq y$ to $\sigma(x) + \sigma(\sigma(x)) \leq y$. This condition corresponds to our informal notion of ‘stopping the loop one iteration early’. Other than this change, the internal operation of the loop is identical, and we shall therefore assign parts of $M_{\sigma}(n, m)$ as before. We then test if x and y are appropriate values for a_k and a_{k-1} . If this condition is true, we enter the conditional block of lines 11–13, and return the resulting composition, which we know *must be nonterminal* because by the time we reach line 10 we know that the condition $\sigma(x) + \sigma(\sigma(x)) > y$ must hold. Otherwise, the algorithm operates as before. We shall formally prove the correctness of Algorithm 4.4 in the following theorem.

Theorem 4.12. *For all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $\text{LEXMIN}'_{\sigma}(n, m) = M_{\sigma}(n, m)$.*

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function, and

§ 4.4. Accelerated Algorithm

proceed by strong induction on n .

Base case: $n = 1$. Since $1 \leq m \leq n$, we know that $m = 1$. Upon entry, we assign $x \leftarrow 1$ and $y \leftarrow 0$. As the codomain of σ is \mathbb{Z}^+ , the loop entry condition on line 4 fails, as does the condition on line 10; therefore, we proceed to line 15. We then assign $a_1 \leftarrow 1 + 0$, and return the composition $\langle 1 \rangle$. Clearly $M_\sigma(1, 1) = \langle 1 \rangle$ and therefore $\text{LEXMIN}'_\sigma(1, 1) = M_\sigma(1, 1)$.

Induction step: Suppose, for some positive integer n , $\text{LEXMIN}'_\sigma(n', m') = M_\sigma(n', m')$ for all positive integers $m' \leq n' < n$. Consider the invocation $\text{LEXMIN}'_\sigma(n, m)$, for an arbitrary positive integer $m \leq n$.

Suppose $m + \sigma(m) > n$. Upon entry to the algorithm we assign $x \leftarrow m$ and $y \leftarrow n - m$, and so $m + \sigma(m) > n \iff \sigma(x) > y$. Therefore, as the codomain of σ is \mathbb{Z}^+ , we shall proceed to line 15, assign $a_1 \leftarrow n$, and return $\langle n \rangle$. By (4.2), $M_\sigma(n, m) = \langle n \rangle$, and hence $\text{LEXMIN}'_\sigma(n, m) = M_\sigma(n, m)$.

Suppose $m + \sigma(m) \leq n$ and $m + \sigma(m) + \sigma(\sigma(m)) > n$. Again, we assign $x \leftarrow m$ and $y \leftarrow n - m$ and so $m + \sigma(m) \leq n \iff \sigma(x) \leq y$ and $m + \sigma(m) + \sigma(\sigma(m)) > n \iff \sigma(x) + \sigma(\sigma(x)) > y$. Therefore we proceed to line 11, assign $a_1 \leftarrow m$ and $a_2 \leftarrow n - m$, and return the composition $\langle m \rangle \langle n - m \rangle$. By Lemma 4.4, $M_\sigma(n, m) = \langle m \rangle \langle n - m \rangle$, and therefore $\text{LEXMIN}'_\sigma(n, m) = M_\sigma(n, m)$.

Now, suppose that $m + \sigma(m) + \sigma(\sigma(m)) \leq n$. In this case we will proceed to line 5, and upon reaching line 8, the variables have the following states: $x = \sigma(m)$, $y = n - m - \sigma(m)$ and $k = 2$. As we have $a_1 = m$, we have correctly assigned the initial part by (4.2) during the first iteration of the loop: the remaining parts, i.e. the composition $M_\sigma(n - m, \sigma(m))$, must clearly be assigned in the remaining iterations.

By the inductive hypothesis, $\text{LEXMIN}'_\sigma(n - m, \sigma(m)) = M_\sigma(n - m, \sigma(m))$. Therefore we must show that the variables x and y are in the same state immediately *after* the first iteration of the loop in $\text{LEXMIN}'_\sigma(n, m)$ and immediately *before* the first iteration of the loop in $\text{LEXMIN}'_\sigma(n - m, \sigma(m))$. Upon entry to $\text{LEXMIN}'_\sigma(n - m, \sigma(m))$, we set $x \leftarrow \sigma(m)$ and $y \leftarrow n - m - \sigma(m)$; and these are the same values held by the corresponding variables after the

§ 4.4. Accelerated Algorithm

first iteration of the loop in $\text{LEXMIN}'_\sigma(n, m)$. Thus, by the inductive hypothesis, $\text{LEXMIN}'_\sigma(n, m) = M_\sigma(n, m)$. Therefore, as m is an arbitrary positive integer such that $m \leq n$, $\text{LEXMIN}'_\sigma(n, m) = M_\sigma(n, m)$ for all m ; hence $\text{LEXMIN}'_\sigma(n, m) = M_\sigma(n, m)$ for all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. \square

LEXMIN'_σ is, of course, no advantage if all we wish to do is compute $M_\sigma(n, m)$ as quickly as possible. It is, however, highly advantageous for our application in this instance, as it identifies terminal and nonterminal compositions. As this property is essential for our analysis of the resulting generation algorithm, it is worthwhile formally proving this property; we do so in the following lemma.

Lemma 4.5. *For all positive integers $m \leq n$ and all nondecreasing functions $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, if the composition $\text{LEXMIN}'_\sigma(n, m) = a_1 \dots a_k$ is returned on line 13 it is nonterminal. If $a_1 \dots a_k$ is returned on line 16 it is terminal.*

Proof. Suppose $m \leq n$ are arbitrary positive integers and suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function. Let $a_1 \dots a_k$ be the composition returned by the invocation $\text{LEXMIN}'_\sigma(n, m)$. Suppose $a_1 \dots a_k$ was returned on line 13 of Algorithm 4.4. Then, $a_{k-1} = x$ and $a_k = y$; upon reaching line 10 we know that the condition $\sigma(x) + \sigma(\sigma(x)) > y$ must hold true, and therefore $\sigma(a_{k-1}) + \sigma(\sigma(a_{k-1})) > a_k$, and $a_1 \dots a_k$ is by definition nonterminal.

Suppose, alternatively, that $a_1 \dots a_k$ was returned on line 16 of Algorithm 4.4. Then, by the argument above, $a_1 \dots a_k$ must be terminal, as all nonterminal compositions are returned on line 13. \square

We can now compute $M_\sigma(n, m)$ and identify terminal and nonterminal compositions, and so we can now proceed to the accelerated generation algorithm itself. The algorithm, ACCELGEN_σ (Algorithm 4.5), operates along the principles outlined above. Unlike the direct implementation of the lexicographic succession rule, however, there are some obvious differences between the procedure to compute $M_\sigma(n, m)$ and the corresponding section of the generation algorithm. This difference arises in the **while** loop of lines 15–20; instead of visiting the single nonterminal composition that arises from

§ 4.4. Accelerated Algorithm

Algorithm 4.5 ACCELGEN $_{\sigma}(n, m)$

Require: $1 \leq m \leq n$ and $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is nondecreasing

```
1:  $k \leftarrow 2$ 
2:  $a_1 \leftarrow m - 1$ 
3:  $a_2 \leftarrow n - m + 1$ 
4: while  $k \neq 1$  do
5:    $y \leftarrow a_k - 1$ 
6:    $k \leftarrow k - 1$ 
7:    $x \leftarrow a_k + 1$ 
8:   while  $\sigma(x) + \sigma(\sigma(x)) \leq y$  do
9:      $a_k \leftarrow x$ 
10:     $x \leftarrow \sigma(x)$ 
11:     $y \leftarrow y - x$ 
12:     $k \leftarrow k + 1$ 
13:  end while
14:   $\ell \leftarrow k + 1$ 
15:  while  $\sigma(x) \leq y$  do
16:     $a_k \leftarrow x$ 
17:     $a_{\ell} \leftarrow y$ 
18:    visit  $a_1 \dots a_{\ell}$ 
19:     $x \leftarrow x + 1$ 
20:     $y \leftarrow y - 1$ 
21:  end while
22:   $a_k \leftarrow x + y$ 
23:  visit  $a_1 \dots a_k$ 
24: end while
```

computing $M_{\sigma}(a_{k-1} + a_k, a_{k-1})$ as we did in RULEGEN $_{\sigma}$ we visit a sequence of compositions. We do this using the succession rule for nonterminal compositions of Theorem 4.10. We know that the first composition visited by this loop must be nonterminal, and we can keep applying the nonterminal succession rule to this composition until we find that $\sigma(x) > y$. In this case we visit the composition given in the second case of Theorem 4.10, and then return to the head of the loop and repeat the process. The correctness of ACCELGEN $_{\sigma}$ is proved in the following theorem.

Theorem 4.13. *For every nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers $m \leq n$, ACCELGEN $_{\sigma}(n, m)$ generates the set $\mathcal{C}_{\sigma}(n, m)$ in*

§ 4.4. Accelerated Algorithm

lexicographic order.

Proof. Suppose $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is an arbitrary nondecreasing function, and suppose $m \leq n$ are arbitrary positive integers. We proceed by induction on the rank r of the elements of $\mathcal{C}_\sigma(n, m)$.

Base case: $r = 0$. Upon initialisation of $\text{ACCELGEN}_\sigma(n, m)$ we enter the main loop, and set $x \leftarrow m$, $y \leftarrow n - m$ and $k \leftarrow 1$ before reaching line 8. These values are identical to the values of the corresponding variables in the invocation $\text{LEXMIN}'_\sigma(n, m)$ before reaching line 4 in that algorithm. Comparing Algorithms 4.4 and 4.5, it is easy to verify that the first composition visited by $\text{ACCELGEN}_\sigma(n, m)$ is equal to the composition returned by $\text{LEXMIN}'_\sigma(n, m)$. Therefore, by Theorem 4.12, the composition at rank $r = 0$ is correctly visited.

Induction step: Suppose for some $r \leq C_\sigma(n, m) - 1$ that the composition $a_1 \dots a_k$ has been visited by $\text{ACCELGEN}_\sigma(n, m)$. Let us examine the next iteration of the main loop.

Suppose that $r = C_\sigma(n, m) - 1$. The lexicographically greatest element of $\mathcal{C}_\sigma(n, m)$ is $\langle n \rangle$, and therefore $k = 1$. Thus, as there must be at least two parts in all compositions visited on line 18, we know that this composition must have been visited on line 23. Therefore, returning to line 4, we see that the loop entry condition is false, and terminate. Therefore, if we have visited the lexicographically last composition, we terminate correctly.

Suppose that $r < C_\sigma(n, m) - 1$, and let $a_1 \dots a_k$ be the composition at rank r . Suppose that $a_1 \dots a_k$ has been visited on line 18. Before entering the loop at line 15, the condition $\sigma(x) + \sigma(\sigma(x)) > y$ must hold true. As σ is nondecreasing, this property is invariant under any number of subsequent increments to x and decrements to y , and so, as $a_{k-1} = x$ and $a_k = y$, $a_1 \dots a_k$ is nonterminal. Then, suppose that $\sigma(a_{k-1} + 1) \leq a_k - 1$. Clearly, we will immediately visit the composition $a_1 \dots a_{k-2} \langle a_{k-1} + 1 \rangle \langle a_k - 1 \rangle$, which, by Theorem 4.10, is $S_\sigma(a_1 \dots a_k)$. Suppose, on the other hand, that $\sigma(a_{k-1} + 1) > a_k - 1$. Then, also by Theorem 4.10, $S_\sigma(a_1 \dots a_k) = a_1 \dots a_{k-2} \langle a_{k-1} + a_k \rangle$,

§ 4.4. Accelerated Algorithm

which we will then visit on line 23. Therefore, if $a_1 \dots a_k$ is visited on line 18, then $S_\sigma(a_1 \dots a_k)$ is visited immediately after.

Suppose $a_1 \dots a_k$ has been visited on line 23. Then, by the analogy with Algorithm 4.4, we know that the next composition visited must be $a_1 \dots a_{k-2} M_\sigma(a_{k-1} + a_k, a_{k-1} + 1)$, and therefore we correctly visit the composition at rank $r + 1$ by the generic lexicographic succession rule.

Therefore, $\text{ACCELG}_{\sigma}(n, m)$ visits all elements of $C_\sigma(n, m)$ in lexicographic order. Hence, the algorithm correctly generates the set $C_\sigma(n, m)$ for any nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and all positive integers $m \leq n$. \square

Thus, ACCELG_{σ} correctly generates all interpart restricted compositions of n for any nondecreasing function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. We shall evaluate the effectiveness of our modifications to RULEG_{σ} in the next section.

4.4.3 Analysis

In our analysis we wish to determine the total number of read and write operations [Kem98] incurred in generating the set $C_\sigma(n, m)$, for some nondecreasing restriction function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, using ACCELG_{σ} (Algorithm 4.5). The key to this analysis is the variable k : it controls termination of the algorithm, and it is updated only via increment and decrement operations. We can therefore infer the total number of iterations of each of the loops by reasoning about the number of increments and decrements performed on k . Before we can do this, we must determine the number of compositions visited in the inner loop. Thus, letting $t_{18}(n, m)$ be the number of times line 18 is executed in the process of generating the set $C_\sigma(n, m)$ using ACCELG_{σ} , we get the following lemma.

Lemma 4.6. *The number of times line 18 is executed during the execution of Algorithm 4.5 is given by $t_{18}(n, m) = N_\sigma(n, m)$.*

Proof. By Lemma 4.5 and the analogy between Algorithm 4.4 and Algorithm 4.5 we know that the composition visited on line 18 during the first iteration of the while loop must be nonterminal. Furthermore, we know that

§ 4.4. Accelerated Algorithm

compositions visited during subsequent iterations of this loop must be non-terminal since, because σ is nondecreasing, the condition $\sigma(x) + \sigma(\sigma(x)) > y$ is invariant under any number of subsequent increments to x and decrements to y . Then, by Lemma 4.5 we know that all compositions visited on line 23 must be terminal, and so all nonterminal compositions must be visited on line 18. Therefore, line 18 is executed exactly $N_\sigma(n, m)$ times and so $t_{18}(n, m) = N_\sigma(n, m)$, as required. \square

Lemma 4.6 shows us that all nonterminal interpart restricted compositions in $\mathcal{C}_\sigma(n, m)$ are visited on line 18, and therefore the terminal compositions must be visited on line 23. From this information we can determine the number of decrement operations on k , and hence the number of increments. Thus, letting $t_6(n, m)$ and $t_{12}(n, m)$ be the number of times lines 6 and 12 are executed, respectively, during the execution of $\text{ACCELGEN}_\sigma(n, m)$, we get the following lemmas.

Lemma 4.7. *The number of times line 6 is executed during the execution of Algorithm 4.5 is given by $t_6(n, m) = T_\sigma(n, m)$.*

Proof. By Lemma 4.6 we know that $N_\sigma(n, m)$ compositions from the set $\mathcal{C}_\sigma(n, m)$ are visited on line 18. By Theorem 4.13, Algorithm 4.5 correctly visits all $\mathcal{C}_\sigma(n, m)$ interpart restricted compositions in $\mathcal{C}_\sigma(n, m)$, and so the remaining $\mathcal{C}_\sigma(n, m) - N_\sigma(n, m) = T_\sigma(n, m)$ compositions must be visited on line 23. Thus, line 23 is executed $T_\sigma(n, m)$ times, and by Kirchhoff's Law, line 6 is executed an equal number of times. Therefore, $t_6(n, m) = T_\sigma(n, m)$, as required. \square

Lemma 4.8. *The number of times line 12 is executed during the execution of Algorithm 4.5 is given by $t_{12}(n, m) = T_\sigma(n, m) - 1$.*

Proof. Algorithm 4.5 begins with $k = 2$ and terminates when $k = 1$, and k is modified only on lines 6 and 12. The statement $k \leftarrow k - 1$ is executed $T_\sigma(n, m)$ times by Lemma 4.7; therefore, the statement $k \leftarrow k + 1$ (line 12) must be executed $T_\sigma(n, m) - 1$ times. Therefore, $t_{12}(n, m) = T_\sigma(n, m) - 1$ \square

Using Lemmas 4.6, 4.7 and 4.8 we can now determine the exact number of read and write operations performed during the complete execution of

§ 4.4. Accelerated Algorithm

the $\text{ACCELGEN}_\sigma(n, m)$. Letting $R_{A4.5}(n, m)$ and $W_{A4.5}(n, m)$ be the total number of reads and writes, respectively, performed by ACCELGEN_σ in the process of generating $\mathcal{C}_\sigma(n, m)$, we get the following theorems.

Theorem 4.14. *Algorithm 4.5 requires $R_{A4.5}(n, m) = 2T_\sigma(n, m)$ read operations to generate the set $\mathcal{C}_\sigma(n, m)$.*

Proof. Read operations are performed on lines 5 and 7. By Lemma 4.7 we know that these statements are executed $T_\sigma(n, m)$ times each, and so we get a total of $2T_\sigma(n, m)$ read operations. Therefore, $R_{A4.5}(n, m) = 2T_\sigma(n, m)$, as required. \square

Theorem 4.15. *Algorithm 4.5 requires $W_{A4.5}(n, m) = 2C_\sigma(n, m) - 1$ write operations to generate the set $\mathcal{C}_\sigma(n, m)$, excluding initialisation.*

Proof. Write operations are performed on lines 9, 16, 17 and 22 of Algorithm 4.5. By Lemma 4.8 line 9 is executed $T_\sigma(n, m) - 1$ times; Lemma 4.6 demonstrates that lines 16 and 17 are executed $N_\sigma(n, m)$ times each; and by Lemma 4.7 line 22 is executed $T_\sigma(n, m)$ times. Thus, summing these individual contributions we get $W_{A4.5}(n, m) = 2T_\sigma(n, m) + 2N_\sigma(n, m) - 1 = 2C_\sigma(n, m) - 1$. \square

Having completed our analysis of ACCELGEN_σ , we complete our development of lexicographic generation algorithms for interpart restricted compositions. In this section, by identifying a special case in the lexicographic succession rule for interpart restricted compositions we developed an accelerated algorithm to generate all interpart restricted compositions. We have now analysed this algorithm, and so we are in a position to compare our basic and accelerated algorithms for generating interpart restricted compositions. This is the purpose of the next subsection.

4.4.4 Comparison

In this section we have developed an ‘accelerated’ algorithm to generate interpart restricted compositions, ACCELGEN_σ . It is not clear whether this

§ 4.4. Accelerated Algorithm

algorithm is actually more efficient than RULEGEN_σ , which is, after all, constant amortised time. We shall examine these algorithms from a theoretical and empirical perspective in this subsection.

Considering RULEGEN_σ (Algorithm 4.3) first, we derived the following totals for the number of read and write operations required.

$$R_{A4.3}(n, m) = 2C_\sigma(n, m) \quad \text{and} \quad W_{A4.3}(n, m) = 2C_\sigma(n, m) - 1 \quad (4.4)$$

Therefore, using RULEGEN_σ we can generate all $C_\sigma(n, m)$ elements of the set $\mathcal{C}_\sigma(n, m)$ using $2C_\sigma(n, m)$ read and $2C_\sigma(n, m) - 1$ write operations. Then, considering the accelerated algorithm, ACCELGEN_σ (Algorithm 4.5), we derived the following totals for the number read and write operations.

$$R_{A4.5}(n, m) = 2T_\sigma(n, m) \quad \text{and} \quad W_{A4.5}(n, m) = 2C_\sigma(n, m) - 1 \quad (4.5)$$

From the perspective of the number of read and write operations involved, the only difference between the algorithms is that RULEGEN_σ requires $2C_\sigma(n, m)$ reads whereas ACCELGEN_σ requires $2T_\sigma(n, m)$. Clearly, the difference between the algorithms, if any, is entirely dependent on the relationship between the number of terminal compositions, $T_\sigma(n, m)$, and the total number of interpart restricted compositions, $C_\sigma(n, m)$.

As we have previously noted, there does not appear to be any simple general relationship between $T_\sigma(n, m)$ and $C_\sigma(n, m)$. For some particular cases, such as when $\sigma(x) = 1$, the relationship is obvious. In general, however, the relationship between these numbers does not appear to follow any simple pattern. Thus, having been unable to derive a general relationship between $C_\sigma(n, m)$ and $T_\sigma(n, m)$, we shall have to satisfy ourselves with some numerical data.

In Table 4.2 we see the numbers of terminal and nonterminal compositions for $n = 10, 100, 1000$ for several restriction functions. We can see from these data that the maximum fraction of terminal compositions in the set $\mathcal{C}_\sigma(n)$ is 0.5, and this occurs for the unrestricted compositions of n . For other restriction functions this fraction is much smaller, the extreme example being

§ 4.4. Accelerated Algorithm

$\sigma(x)$	n	$C_\sigma(n)$	$T_\sigma(n)$	$N_\sigma(n)$	$T_\sigma(n)/C_\sigma(n)$
1	10	512	256	256	0.5000
	100	6.34×10^{29}	3.17×10^{29}	3.17×10^{29}	0.5000
	1000	5.36×10^{300}	2.68×10^{300}	2.68×10^{300}	0.5000
2	10	55	21	34	0.3818
	100	3.54×10^{20}	1.35×10^{20}	2.19×10^{20}	0.3820
	1000	4.35×10^{208}	1.66×10^{208}	2.69×10^{208}	0.3820
x	10	42	20	22	0.4762
	100	1.91×10^{08}	4.04×10^{07}	1.50×10^{08}	0.2118
	1000	2.41×10^{31}	1.83×10^{30}	2.22×10^{31}	0.0761
$x + 1$	10	10	4	6	0.4000
	100	444793	68537	376256	0.1541
	1000	8.64×10^{21}	4.69×10^{20}	8.17×10^{21}	0.0544
$2x$	10	6	2	4	0.3333
	100	1189	116	1073	0.0976
	1000	4.42×10^{07}	769343	4.34×10^{07}	0.0174

Table 4.2: Number terminal and nonterminal interpart restricted compositions of n for several restriction functions. For each value of n and σ the proportion of the compositions in $\mathcal{C}_\sigma(n)$ that are terminal is also shown.

when $\sigma(x) = 2x$. Putting these data back into the context of our generation algorithm, since the relative durations of the algorithms is inextricably tied to the fraction of terminal compositions in the set $\mathcal{C}_\sigma(n)$, we should expect to see, not a constant difference between the algorithms, but one that varies widely. The accelerated algorithm should always require less time than the direct succession rule algorithm, but this difference will vary depending on the restriction function.

Empirical Evaluation

We have predicted theoretically that the accelerated algorithm developed in this section (ACCELGEN_σ) should be significantly more efficient than the succession rule based algorithm developed in the previous section (RULEGEN_σ).

§ 4.4. Accelerated Algorithm

We shall now perform an empirical analysis of these algorithms, to determine whether our theory reflects practice on an actual computer. Before we begin our comparison, we shall first deal with the necessary methodological issues.

Theoretical analyses are subject to pitfalls associated with the innate simplification of the problem which is required [Knu73], but empirical comparisons are also prone to leading to false conclusions. In the context of his comparison of permutation generation algorithms [Sed77], Sedgewick has noted that “many misleading conclusions have been drawn and reported in the literature based on empirical performance statistics comparing particular implementations of particular algorithms.” Sedgewick then goes on to remark that, in many comparisons of permutation generation methods, “the empirical tests which have been performed have really been comparisons of compilers, programmers and computers, not algorithms.”

Sedgewick’s concerns are pertinent, and we alleviate the three factors he identifies (namely, the compilers, programmers and computers involved) by taking the following steps in all of the experiments we conduct in this dissertation. The issue of comparing programmers is mitigated by using literal and unmeliorated implementations of the algorithm listings involved. That is, in the high level languages used we implement the algorithms literally and exactly as given. Any programmer given the algorithm listings can then implement the algorithms exactly as provided, and reproduce the experiments. Certain improvements can be made to all of the algorithms, such as using pointers to refer to array elements in the C implementations, but such improvements tend to apply equally to all implementations, and thus cancel out. It is not practical to exhaustively compare the algorithms piecewise with all conceivable implementation improvements applied to each; even by taking this step there would surely be some further improvement to be made. The most we can hope to provide with our empirical comparison is to provide an assurance of *likelihood*: given the theoretical and empirical evidence presented, it seems likely that a certain conclusion can be drawn, with a reasonable degree of confidence in its validity.

We address Sedgewick’s assertion that empirical comparisons are often comparisons of compilers and computers by taking two steps, both of which

§ 4.4. Accelerated Algorithm

are intimately related to addressing Sedgewick’s criticisms. Firstly, compilation is performed in the simplest feasible manner by disabling all extra compiler ‘optimisations’. We can then hope that it is the algorithms themselves compared and not the implementation’s suitability for compiler meliorations. Secondly, in an attempt to alleviate the problems caused by performing experiments on a single computer, we compare implementations of the algorithms in the C and Java languages. (Other languages such as Python and Haskell were also considered. The programs written in these languages, however, were unable to generate numbers of partitions of the orders of magnitude appropriate for C and Java in a feasible amount of time. If the numbers of partitions generated in experiments were reduced to values suitable for Python or Haskell, the amount of time required by the C and Java implementations would be far too brief to measure accurately.) We should, in this way, get a good indication of the computational properties of the algorithms involved on any reasonable computer. Specifically, all experiments in this dissertation we performed on the following platforms. The C programs were compiled using GCC version 3.3.4 (with compiler ‘optimisations’ turned off), and the Java programs compiled and run on the Java[™] 2 Standard Edition, version 1.4.2. All programs were executed on an Intel[®] Pentium[®] 4 processor running Linux kernel 2.6.8.

In each case the total amount of time required to generate the set of compositions in question was measured. In the case of C it was possible to measure the ‘user time’, or the total processor time the process used in executing instructions of its program, using the `times` function. When measuring the time required to generate compositions using the Java programming language, it was necessary to measure the ‘wall clock’ time, or the amount of time which elapsed from the initiation of the generation algorithm and its completion. In either case each experiment was repeated five times, and the minimum value from these five runs recorded. (The minimum rather than the average is appropriate here as the algorithms require a fixed number of instructions to run. Thus the minimum is the value least affected by other processes running on the processor, and represents the closest approximation to the actual time spent executing the algorithm.)

§ 4.4. Accelerated Algorithm

The precise time required for each experiment is of little interest to us here. We wish, instead, to ascertain whether our theoretical models for the runtime of each algorithm are accurate representations of actual implementations of the algorithms. Thus we report the amount of time required to generate a given set of compositions using the improved algorithm divided by the time required by the succession rule based algorithm. In this way we can determine the difference in running times between the two algorithms, under a given implementation. We also report the predicted ratio of the running times under the total read-write operation model discussed in the previous subsection. In this case we simply added the total number of read and write operations incurred for each algorithm (hence assuming that the cost of a read operation is equal to the cost of a write operation), and reported the computed ratio between the two algorithms.

Returning to the particular experiments we are interested in for this section, we implemented both algorithms in the C and Java languages, as discussed above. For each value of the restriction function we implemented a separate algorithm with the appropriate value for $\sigma(x)$ substituted into the algorithm listings. Direct implementations of RULEGEN_σ and ACCELGEN_σ were used. A representative set of restriction functions were used for the experiment. These include the restriction functions to describe the unrestricted compositions, the unrestricted partitions and the partitions into distinct parts. For each restriction function values of n were chosen such that n is the smallest integer where $C_\sigma(n) > 1 \times 10^x$ and $C_\sigma(n) > 5 \times 10^x$ for $x = 7$ and $x = 8$. In this way we can compare the algorithms on different instances of the restriction function while generating ‘realistic’ numbers of compositions. The numbers of compositions generated simply reflect the power of the computer used in the experiments. The upper bound, 5×10^8 , is chosen such that the overall experiment can be run in a reasonable amount of time and the lower bound chosen such that the amount of time required can be measured with some degree of accuracy. (The total amount of time required to run all of the experiments in this dissertation on the machine in question is approximately ten hours.)

The results of these experiments are shown in Table 4.3. We can see that

§ 4.4. Accelerated Algorithm

$\sigma(x)$	n	$C_\sigma(n)$	Java	C	Theoretical
1	25	1.68×10^7	0.54	0.66	0.75
	27	6.71×10^7	0.54	0.69	0.75
	28	1.34×10^8	0.54	0.69	0.75
	30	5.37×10^8	0.55	0.69	0.75
2	36	1.49×10^7	0.54	0.63	0.69
	39	6.32×10^7	0.55	0.64	0.69
	40	1.02×10^8	0.55	0.64	0.69
	44	7.01×10^8	0.55	0.64	0.69
x	77	1.06×10^7	0.44	0.63	0.62
	90	5.66×10^7	0.43	0.64	0.61
	95	1.05×10^8	0.43	0.62	0.61
	109	5.42×10^8	0.42	0.63	0.60
$x + 1$	141	1.03×10^7	0.33	0.50	0.57
	164	5.00×10^7	0.31	0.48	0.56
	175	1.03×10^8	0.30	0.47	0.56
	201	5.17×10^8	0.30	0.46	0.56
$x + 2$	177	1.03×10^7	0.33	0.52	0.55
	206	5.06×10^7	0.31	0.50	0.55
	220	1.05×10^8	0.31	0.49	0.55
	252	5.15×10^8	0.30	0.48	0.55
$2x$	771	1.00×10^7	0.22	0.41	0.51
	1022	5.02×10^7	0.21	0.40	0.51
	1148	1.00×10^8	0.21	0.40	0.51
	1491	5.01×10^8	0.21	0.39	0.51

Table 4.3: A comparison of the rule-based and accelerated algorithms to generate interpart restricted compositions.

in all cases, for all values of n and all values of the restriction function, the accelerated algorithm is significantly more efficient. The difference is quite extreme when $\sigma(x) = 2x$, as is to be expected. The theoretically derived ratio of the two algorithms, computed by counting the total read and write operations for the two algorithms as per our analyses, is also shown. We can see that this ratio consistently overestimates the ratio, and this is also to be expected. In our theoretical predictions, we specifically assume that the costs

of read and write operations are equal. One of the effects of the accelerated algorithm, however, is to significantly reduce the cost of a write operation. Thus, a write operation will cost less on average in the accelerated algorithm than in the rule-based algorithm, and therefore a direct comparison based on these counts will not take this property into account.

4.5 Summary

In this chapter we have developed efficient algorithms to generate interpart restricted compositions of n for any nondecreasing restriction function. We have developed a simple recursive algorithm that can be trivially shown to have the constant amortised time property. We then developed an abstract succession rule for interpart restricted compositions, a rule that transforms any given composition into its immediate lexicographic successor. We then implemented this rule as a generation algorithm, and proved that it also has the constant amortised time property by counting the number of read and write operations (on the compositions themselves) incurred by the algorithm in generating all interpart restricted compositions of n . We also briefly examined some commensurable algorithms from the literature, and demonstrated that this direct implementation of our succession rule is at least competitive with the algorithms from the literature to generate specific subsets of the interpart restricted compositions. We then developed a means of improving the efficiency of this algorithm by introducing the theory of ‘terminal’ and ‘nonterminal’ compositions. This auxiliary theory allowed us to find efficiently implementable special cases in the general succession rule. Empirical observation demonstrates that these special cases are quite successful: our accelerated algorithm is strictly more efficient than the direct implementation of the succession rule, and can lead to large improvements in efficiency in certain instances.

Chapter 5

Generating All Partitions

A fundamental choice that must be made in developing any combinatorial generation algorithm is that of the encoding we use to represent the objects of interest. Using a poor encoding will inevitably lead to inefficient generation algorithms. All existing partition generation techniques implicitly encode partitions as descending compositions; in this chapter we investigate the alternative of encoding partitions as ascending compositions. Fundamentally, then, we are comparing algorithms to generate ascending and descending compositions; we demonstrate that ascending compositions can be generated more efficiently, and are therefore a more appropriate encoding for partitions.

The chapter proceeds as follows. In Section 5.1 we discuss notational conventions and the methodology adopted in our analyses. Section 5.2 then begins the comparison of ascending and descending composition generation methods by directly comparing two recursive algorithms. Section 5.3 makes a more abstract comparison of ascending and descending composition generation by comparing the relevant succession rules, and analysing the total computational cost implied by a direct implementation of these rules. In Section 5.4 we compare the most efficient known algorithms to generate ascending and descending compositions, from both a theoretical complexity and practical efficiency point of view. We then conclude the chapter in Section 5.5 by reviewing the evidence assembled in the chapter, and drawing our conclusions based on this.

5.1 Preliminaries

The differences between ascending and descending compositions can be rather subtle, and so we require some carefully chosen notation to maintain clarity. Mainly, we distinguish between quantities referring to ascending and descending compositions by using either an upper or lowercase ‘a’ or ‘d’ in the obvious way. For instance, whenever we use the notation $d_1 \dots d_k$ it is understood that we are referring to a sequence in which the parts are in descending order. It is useful to have formal definitions of the concepts involved, and we shall begin by defining precisely what is meant by an ‘ascending composition,’ and develop some notation in terms of these objects.

Definition 5.1 (Ascending Composition). *A sequence of positive integers $a_1 \dots a_k$ is an ascending composition of the positive integer n if $a_1 + \dots + a_k = n$ and $a_1 \leq \dots \leq a_k$.*

Using this definition we can now define some useful notation that allows us to discuss sets of ascending compositions and the cardinalities of those sets (i.e. enumeration functions).

Definition 5.2 (Set and Enumeration Functions). *Let $\mathcal{A}(n)$ be the set of all ascending compositions of n for some $n \geq 1$, and let $\mathcal{A}(n, m) \subseteq \mathcal{A}(n)$ be defined for $1 \leq m \leq n$ as $\mathcal{A}(n, m) = \{a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{A}(n) \text{ and } a_1 \geq m\}$. Also, let $A(n) = |\mathcal{A}(n)|$ and $A(n, m) = |\mathcal{A}(n, m)|$.*

Thus, the set $\mathcal{A}(n)$ contains all ascending compositions of n and the set $\mathcal{A}(n, m)$ contains all ascending compositions of n where the first part is at least m . As an example, consider the set of ascending compositions of 5:

$$\mathcal{A}(5) = \{11111, 1112, 113, 122, 14, 23, 5\}. \quad (5.1)$$

As $\mathcal{A}(n, m)$ is the set of all ascending compositions where the first part $\geq m$, we see that $\mathcal{A}(5, 1) = \mathcal{A}(5)$, as all ascending compositions of n have first part at least 1. Then, we see from the example above that $\mathcal{A}(5, 2) = \{23, 5\}$, and furthermore, that $\mathcal{A}(5, 3) = \mathcal{A}(5, 4) = \mathcal{A}(5, 5) = \{5\}$. Thus, to continue the

§ 5.1. Preliminaries

example with the enumeration functions, we see that $A(5) = A(5, 1) = 7$, $A(5, 2) = 2$ and $A(5, 3) = A(5, 4) = A(5, 5) = 1$.

As we are comparing algorithms to generate ascending compositions with existing algorithms to generate descending compositions we shall also define some similar functions in terms of descending compositions, although with some subtle and important differences.

Definition 5.3 (Descending Composition). *A sequence of positive integers $d_1 \dots d_k$ is a descending composition of the positive integer n if $d_1 + \dots + d_k = n$ and $d_1 \geq \dots \geq d_k$.*

Analogous to Definition 5.1, a descending composition is any sequence of positive integers whose sum is n where the sequence elements are arranged in descending order. Again, we shall now define some set and enumeration functions in terms of these objects.

Definition 5.4 (Set and Enumeration Functions). *Let $\mathcal{D}(n)$ be the set of all descending compositions of n for some $n \geq 1$, and let $\mathcal{D}^*(n, m) \subseteq \mathcal{D}(n)$ be defined for $1 \leq m \leq n$ as $\mathcal{D}^*(n, m) = \{d_1 \dots d_k \mid d_1 \dots d_k \in \mathcal{D}(n) \text{ and } d_1 = m\}$. Also, let $D(n) = |\mathcal{D}(n)|$ and $D^*(n) = |\mathcal{D}^*(n, m)|$.*

The set $\mathcal{D}(n)$ contains all descending compositions of n ; as an example, consider the set $\mathcal{D}(5)$:

$$\mathcal{D}(5) = \{11111, 2111, 221, 311, 32, 41, 5\}. \quad (5.2)$$

Considering the examples of (5.1) and (5.2) it is easy to see that there is a one-to-one correspondence between the set $\mathcal{A}(n)$ and $\mathcal{D}(n)$, and so we easily have $A(n) = D(n)$ for all $n \geq 1$. There is, however, an unfortunate asymmetry between the two parameter recurrences used for ascending and descending compositions, which we have attempted to highlight using the notation¹. The function $\mathcal{D}^*(n, m)$ is defined as the set of descending compositions where the first part is *exactly* m (as opposed to the function $\mathcal{A}(n, m)$ which computes

¹We require enumeration functions of this type for our analysis of Ruskey's algorithm [Rus01, §4.8] in Section 5.2.2

§ 5.1. Preliminaries

the set of ascending compositions of n where the first part is *at least* m). Thus, to take examples from (5.2) above, we see that $\mathcal{D}^*(5, 1) = \{11111\}$, $\mathcal{D}^*(5, 2) = \{2111, 221\}$, $\mathcal{D}^*(5, 3) = \{311, 32\}$, and so on. Correspondingly, the enumeration function $D^*(n, m)$ returns the cardinality of these sets, and so we have $D^*(5, 1) = 1$, $D^*(5, 2) = 2$, $D^*(5, 3) = 2$, etc. As the function $\mathcal{D}^*(n, m)$ computes the total number of descending compositions where the first part is exactly m we can express the total number of descending compositions by summing over all possible values for the first part. Thus, we get the equation $D(n) = \sum_{m=1}^n D^*(n, m)$.

Although ascending and descending compositions are in one-to-one correspondence, there is at least one notable difference in the computational techniques defined in terms of the two classes of composition. In the functions $A(n, m)$ and $D^*(n, m)$ we have defined above we restrict the value of the first part. Since the parts are arranged in ascending and descending order, respectively, we are effectively restricted the value of the *smallest* part in ascending compositions and the *largest* part in descending compositions. This difference is most noticeable in the recursive algorithms we shall study in Section 5.2.

Both ascending and descending compositions are in correspondence with another class of combinatorial object, the *partitions* of n . A partition of a positive integer n is defined as any *unordered* collection of positive integers whose sum is n . Then, since there is a unique way of writing any unordered collection in either ascending or descending order, there are one-to-one correspondences between the ascending compositions, descending compositions and the partitions of n . The function $p(n)$ is conventionally used to denote the number of partitions of n [And76, p.1]; thus, we can see that $p(n) = A(n) = D(n)$. In our analyses of the algorithms we investigate in this chapter our ultimate goal will be to express the quantities of interest in terms of the function $p(n)$, about which much is known. Thus, once we have a quantity which may be expressed in terms of $p(n)$ we shall immediately do so.

Various other notations shall be utilised in the progress of this chapter. All of these notations are either well known or simple extensions of the nota-

tions used in previous chapters, and so we shall not unnecessarily complicate this discussion by introducing them all here.

5.2 Recursive Algorithms

In this section we examine recursive algorithms to generate ascending and descending compositions. Recursion is a popular technique in combinatorial generation as it leads to elegant and concise generation procedures [Rus01]. In Section 5.2.1 we instantiate our recursive interpart restricted composition generator from Section 4.2, and analyse the resulting algorithm. Then, in Section 5.2.2 we study Ruskey's descending composition generator [Rus01, §4.8], and provide a new analysis of this algorithm. We compare these algorithms in Section 5.2.3 in terms of the total number of recursive invocations required to generate all $p(n)$ partitions of n .

5.2.1 Ascending Compositions

In Section 4.2 we developed a recursive algorithm to generate interpart restricted compositions, RECGEN_σ . In this chapter we are interested in generating all unrestricted partitions of n , and by suitably instantiating RECGEN_σ we easily obtain such an algorithm. In the interpart restricted compositions framework (Chapter 3) we represent the unrestricted partitions using the identity function. Therefore, by replacing all instance of $\sigma(x)$ in RECGEN_σ with x we obtain an ascending composition generator; the resulting algorithm is REASC (Algorithm 5.1).

$\text{REASC}(n, m, k)$ generates all ascending compositions of n where the first part is at least m in lexicographic order; thus, invoking $\text{REASC}(6, 2, 1)$ will successively visit the compositions 222, 24, 33, 6. To generate all ascending compositions (and hence all partitions) of a positive integer n we invoke $\text{REASC}(n, 1, 1)$, as 1 is the minimum value for the first part. The parameter k is used recursively to assign values into the correct index, and the initial value represents the index of the first part in all of the compositions generated. We are assured of the correctness of the algorithm by Theorem 4.2,

§ 5.2. Recursive Algorithms

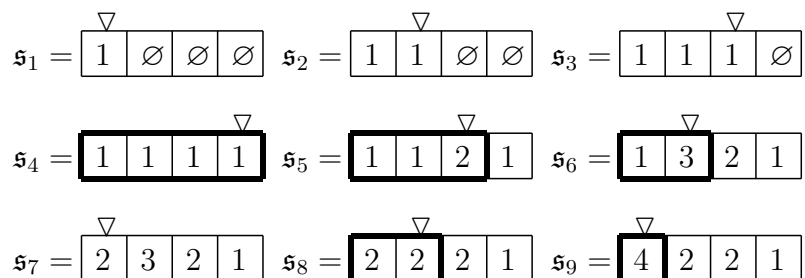


Figure 5.1: Array state-transition diagram for the recursive ascending composition generation algorithm. Indices marked with the ∇ symbol are the indices in which a value is assigned to change the state of the array, and array segments outlined in bold are the portions that are visited.

Algorithm 5.1 $\text{RECASC}(n, m, k)$

Require: $1 \leq m \leq n$

- 1: $x \leftarrow m$
 - 2: **while** $2x \leq n$ **do**
 - 3: $a_k \leftarrow x$
 - 4: $\text{RECASC}(n - x, x, k + 1)$
 - 5: $x \leftarrow x + 1$
 - 6: **end while**
 - 7: $a_k \leftarrow n$
 - 8: **visit** $a_1 \dots a_k$
-

where we verified the correctness of Algorithm 4.1 by an inductive argument.

The effect of the invocation $\text{RECASC}(4, 1, 1)$ is given in Figure 5.1, where we see all states of the generation array the algorithm traverses in generating the ascending compositions of 4. The state of the array changes each time we make a write operation $a_j \leftarrow x$ on lines 3 or 7. Since filling the array a with the appropriate values is the essence of generation, we can get an intuitive grasp on how an algorithm operates by examining such diagrams.

After initialisation, the algorithm proceeds by filling the array with four copies of 1; each of which is assigned in an invocation where the parameter n is equal to $4, \dots, 1$ and $k = 1, \dots, 4$. Once all of the 1s have been written, execution falls back down through the recursion chain, we assign the values of the parameter to n to a_k , and visit the composition $a_1 \dots a_k$ (states $\mathfrak{s}_{4..6}$).

§ 5.2. Recursive Algorithms

State \mathfrak{s}_7 arises when we return to the initial invocation of Algorithm 5.1, where $n = 4$ and $m = 1$; in this invocation we increment x to 2, find that $2x \leq n$, and so recursively reinvoke the algorithm. We then arrive at state \mathfrak{s}_8 , where we visit the composition 22, and return again to the initial invocation. The variable x is then set to 3, but as $2x \not\leq n$ we do not reenter the loop and so proceed to state \mathfrak{s}_9 by assigning $a_1 = 4$, visit the resulting composition, and terminate.

Analysis

Following the standard practice for the analysis of recursive generation algorithms, we count the number of recursive calls required to generate the set of combinatorial objects in question [ER03, RECS94, NSST98, BS97, Boy05, Saw01, Rus01]. By counting the total number of recursive invocations required, we obtain a bound on the total time required, as each invocation, discounting the time spent in recursive calls, requires constant time. To establish that Algorithm 5.1 generates the set $\mathcal{A}(n)$ in constant amortised time we must count the total number of invocations, and show that this value is proportional to $A(n)$. Although the relevant results are proved in the general treatment of Section 4.2, the arguments are concise enough to reproduce. Letting $I_{A5.1}(n)$ be the total number of invocations of Algorithm 5.1 required to generate all ascending compositions of n , we obtain the following result.

Theorem 5.1. *For all positive integers n , $I_{A5.1}(n) = p(n)$.*

Proof. Each invocation of Algorithm 5.1 visits exactly one composition (line 8). By Theorem 4.2 the invocation $\text{REASC}(n, m, 1)$ correctly visits all $p(n)$ ascending compositions of n ; it immediately follows, therefore, that there must be $p(n)$ invocations. Hence, $I_{A5.1}(n) = p(n)$. \square

Theorem 5.1 gives us an asymptotic measure of the total computational effort required to generate all partitions of n using Algorithm 5.1. It is also useful, for the purposes of our comparative analysis, to know the average amount of effort that this total implies per partition. Therefore, we let $\bar{I}_{A5.1}(n)$ denote the average number of invocations of REASC required to

§ 5.2. Recursive Algorithms

generate an ascending composition of n . To determine this value we simply have to divide the total number of invocations required to generate all partitions by the number of partitions generated; that is, $\bar{I}_{A5.1}(n) = I_{A5.1}(n)/p(n)$. The following corollary then follows easily from Theorem 5.1.

Corollary 5.1. *For all positive integers n , $\bar{I}_{A5.1}(n) = 1$.*

Proof. By Theorem 5.1, we know that $I_{A5.1}(n) = p(n)$. As $\bar{I}_{A5.1}(n) = I_{A5.1}(n)/p(n)$, we then immediately have $\bar{I}_{A5.1}(n) = 1$. \square

Although the result follows trivially from Corollary 5.1, it is useful to formally prove that Algorithm 5.1 generates ascending compositions in constant amortised time. Recall that a generation algorithm is constant amortised time if the average computational effort per object generated is bounded, from above, by some constant. We then immediately have the following result.

Corollary 5.2. *Algorithm 5.1 is constant amortised time.*

Proof. By Corollary 5.1 we know that the average number of invocations of Algorithm 5.1 per composition generated is 1. Therefore, assuming the cost of each invocation, excluding the cost of recursive invocations, is constant, we see that the average computational effort per composition generated is constant. \square

We now have sufficient information to compare REASC with existing recursive descending composition generators. In the next subsection we study the most efficient known example of such an algorithm. Then, in Section 5.2.3, we compare recursive algorithms to generate ascending and descending compositions, and conclude this section on the recursive generation of all partitions.

5.2.2 Descending Compositions

The problem of generating all partitions has long been synonymous with the problem of generating all descending compositions, and there is a range

§ 5.2. Recursive Algorithms

of algorithms available to solve the latter problem — see Table 2.2 (p.35). Essentially, two recursive algorithms are available: Page & Wilson’s [PW79, §5.5] generator and Ruskey’s improvement thereof [Rus01, §4.8]. Page & Wilson’s algorithm is not constant amortised time (i.e. the total number of invocations required to generate all partitions divided by the number of partitions generated is not bounded, from above, by a constant) [Rus01, §4.8], but is nevertheless cited in certain texts as the preferred means of generating all partitions — see Kreher & Stinson [KS98, p.68] or Skiena [Ski90, p.51]. Ruskey’s algorithm improves on Page & Wilson’s basic technique to obtain constant amortised time performance, and we shall therefore study Ruskey’s algorithm in detail in order to compare it with the algorithm presented in the previous subsection.

Ruskey’s algorithm, given in Algorithm 5.2, generates all descending compositions of n in which the first (and largest) part is *exactly* m . Thus the invocation `RECDISC(8, 4, 1)` visits the compositions 41111, 4211, 422, 431, 44, which are in lexicographic order. `RECDISC` uses what Ruskey refers to as a ‘path elimination technique’ [Rus01, §4.3] to attain constant amortised time performance.

Path elimination techniques are required when recursion passes through parameter paths that do not terminate in a valid object, and so we must trace back up along the call chain. Path elimination techniques remove the necessity to backtrack by predicting ‘dead-ends’ and taking action to prevent them occurring. Ruskey suggests two general methods to avoid unfruitful recursion chains. The first technique is obtained by observing that chains of recursive calls are sometimes caused by parameters reaching boundary conditions; by initialising the array to hold the values of the string occurring at the boundary conditions in question we can avoid the chains required to assign these values. If we restore these values after recursion backs up, we can avoid all chains resulting from the boundary condition in question. The second technique suggested by Ruskey is to halt recursion some number of steps before reaching a terminal node, and to use some simpler procedure to fill in the required values. For more information, and further applications of these techniques, see Ruskey [Rus01].

§ 5.2. Recursive Algorithms

Algorithm 5.2 RECDDESC(n, m, k) [Rus01, §4.8]

Require: $1 \leq m \leq n$ and $d_j = 1$ for $j > k$

```
1:  $d_k \leftarrow m$ 
2: if  $n = m$  or  $m = 1$  then
3:   visit  $d_1 \dots d_{k+n-m}$ 
4: else
5:   for  $x \leftarrow 1$  to  $\min(m, n - m)$  do
6:     RECDDESC( $n - m, x, k + 1$ )
7:   end for
8:    $d_k \leftarrow 1$ 
9: end if
```

RECDDESC uses a path elimination technique of the first type to attain constant amortised time performance. If $m = 1$ we know that the composition in question must consist of n copies of the value 1. By initialising the array d to hold n copies of 1 (and by resetting array values to 1 after recursion backs up — see line 8) we can avoid the recursive calls required to assign n copies of 1 into the array.

Another slight complication arises when we wish to use RECDDESC to generate *all* descending compositions. As the algorithm generates all descending compositions where the first part is exactly m , we must iterate through all $j \in \{1, \dots, n\}$ and invoke RECDDESC($n, j, 1$). Following Ruskey's recommendations, and for the purposes of analysis, we shall consider instead the invocation RECDDESC($2n, n, 1$). This invocation will generate all descending compositions of $2n$ where the first part is exactly n ; therefore the remaining parts will be a descending composition of n . Thus, if we alter line 3 to ignore the first part in a (i.e. **visit** $a_2 \dots a_{k+n-m}$), we will visit all descending compositions of n in lexicographic order.

The sequence of array states traversed by RECDDESC($8, 4, 1$) is given in Figure 5.2. We assume in this instance that the array has been initialised to contain 5 copies of 1, and that the algorithm has been modified to visit values from d_2 onwards. The initial invocation of the algorithm assigns $d_1 \leftarrow 4$ (we invoke the algorithm with $n = 8$ and $m = 4$), moving the array from its original state to \mathfrak{s}_1 . After this assignment, we enter the loop and invoke RECDDESC($4, 1, 2$); here we assign $d_2 \leftarrow 1$, and so bring the array into state

§ 5.2. Recursive Algorithms

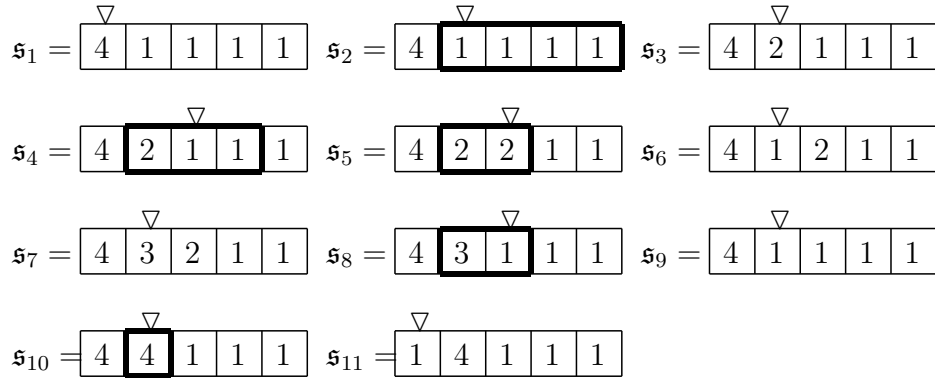


Figure 5.2: Array state-transition diagram for Ruskey’s algorithm. Indices marked with the ∇ symbol are the indices in which a value is assigned to change the state of the array, and array segments outlined in bold are the portions that are visited.

\mathfrak{s}_2 . Then, as $m = 1$, we visit the composition held in $d_2 \dots d_5$, and return to the initial invocation. We then set $x = 2$, and reinvoke, bringing the array into state \mathfrak{s}_3 . This process then continues until $x = 4$, where we assign $d_2 \leftarrow 4$, giving us state \mathfrak{s}_{10} , from which we immediately visit d_2 . Then, as the loop has finished iterating, we assign $d_1 \leftarrow 1$ (giving us \mathfrak{s}_{11}), and terminate.

Analysis

In Section 5.2.1 we proved that the total number of invocations of the ascending composition generator (REASC) required to generate all partitions of n is $p(n)$, which, because of the structure of the algorithm, was trivial to derive. To compare the recursive generators for ascending and descending compositions we must also derive the number of invocations of Ruskey’s algorithm required to generate all partitions. This analysis is, however, far from trivial because the **visit** statement in RECDISC (Algorithm 5.2) is enclosed in a conditional statement. Thus, we know that the **visit** statement is executed $p(n)$ times, and so there must be at least $p(n)$ invocations; but we do not know the total number of invocations. We use a recurrence relation to enumerate the total invocations of $\text{RECDISC}(n, m, 1)$ and then solve this

§ 5.2. Recursive Algorithms

recurrence in terms of the partition function, $p(n)$.

Ruskey’s algorithm generates descending compositions where the largest part is exactly m , and so we require a recurrence relation to count objects of this type. Recall from Section 5.1 that the function $D^*(n, m)$ counts the descending compositions of n where the first part is exactly m ; Ruskey [Rus01, §4.8] provides a recurrence relation to compute $D^*(n, m)$, which we shall use for our analysis. Thus, we define $D^*(n, n) = D^*(n, 1) = 1$, and in general,

$$D^*(n, m) = \sum_{x=1}^{\min(m, n-m)} D^*(n-m, x). \quad (5.3)$$

Recurrence (5.3) is useful here because it is, in fact, the recurrence relation upon which RECDISC is based. Developing a recurrence to count the number of invocations of RECDISC required to generate the descending compositions of n where the first part is exactly m , $I'_{A5.2}(n, m)$, is then relatively simple. Following Ruskey’s instructions [Rus01, §4.11], we obtain the recurrence to describe the number of invocations by “adding +1 to each non-constant case” of (5.3). Thus, we define $I'_{A5.2}(n, n) = I'_{A5.2}(n, 1) = 1$, and

$$I'_{A5.2}(n, m) = 1 + \sum_{x=1}^{\min(m, n-m)} I'_{A5.2}(n-m, x). \quad (5.4)$$

Recurrence (5.4) computes the number of invocations of Algorithm 5.2 required to generate all descending compositions of n with first part exactly m , but tells us little about the actual magnitude of this value. As a step towards solving this recurrence in terms of the partition function $p(n)$ we require the following lemma, in which we relate the $I'_{A5.2}(n, m)$ numbers to the $D^*(n, m)$ numbers.

Lemma 5.1. *If $1 < m \leq n$ then $I'_{A5.2}(n, m) = D^*(n, m) + D^*(n-1, m)$.*

Proof. Proceed by strong induction on n .

Base case: $n = 2$. Suppose $1 < m \leq 2$; it follows immediately that $m = 2$. Thus, by recurrence (5.4) we compute $I'_{A5.2}(2, 2) = 1$ and by recurrence (5.3)

§ 5.2. Recursive Algorithms

compute $D^*(2, 2) = 1$ and $D^*(1, 2) = 0$. Therefore, $I'_{A5.2}(2, 2) = D^*(2, 2) + D^*(1, 2)$, and so the inductive basis holds.

Induction step: Suppose, for some positive integer n , that $I'_{A5.2}(n', m') = D^*(n', m') + D^*(n' - 1, m')$ for all positive integers $1 < m' \leq n' < n$. Then, suppose m is an arbitrary positive integer such that $1 < m \leq n$. Now, suppose $m = n$. By (5.4) we know that $I'_{A5.2}(n, m) = 1$ since $m = m$. Also, $D^*(n, m) = 1$ as $m = n$, and $D^*(n - 1, m) = 0$ as $n - 1 \neq m$, $m \neq 1$ and $\min(m, n - m - 1) = -1$, ensuring that the sum in (5.3) is empty. Therefore, $I'_{A5.2}(n, m) = D^*(n, m) + D^*(n - 1, m)$.

Suppose, on the other hand, that $1 < m < n$. We can see immediately that $\min(m, n - m) \geq 1$, and so there must be at least one term in the sum of (5.4). Extracting this first term where $x = 1$ from (5.4) we get

$$I'_{A5.2}(n, m) = 1 + I'_{A5.2}(n - m, 1) + \sum_{x=2}^{\min(m, n-m)} I'_{A5.2}(n - m, x),$$

and furthermore, as $I'_{A5.2}(n, 1) = 1$, we obtain

$$I'_{A5.2}(n, m) = 2 + \sum_{x=2}^{\min(m, n-m)} I'_{A5.2}(n - m, x). \quad (5.5)$$

We are assured that $1 < x \leq n - m$ by the upper and lower bounds of the summation in (5.5), and so we can apply the inductive hypothesis to get

$$\begin{aligned} I'_{A5.2}(n, m) &= 2 + \sum_{x=2}^{\min(m, n-m)} (D^*(n - m, x) + D^*(n - m - 1, x)) \\ &= 2 + \sum_{x=2}^{\min(m, n-m)} D^*(n - m, x) + \sum_{x=2}^{\min(m, n-m)} D^*(n - m - 1, x). \end{aligned}$$

By the definition of D^* we know that $D^*(n, 1) = 1$, and so $D^*(n - m, 1) + D^*(n - m - 1, 1) = 2$. Replacing the leading 2 above with this expression, and inserting the terms $D^*(n - m, 1)$ and $D^*(n - m - 1, 1)$ into the appropriate

§ 5.2. Recursive Algorithms

summations we find that

$$I'_{A5.2}(n, m) = \sum_{x=1}^{\min(m, n-m)} D^*(n-m, x) + \sum_{x=1}^{\min(m, n-m)} D^*(n-m-1, x). \quad (5.6)$$

By (5.3) we know that the first term of (5.6) is equal to the first term of $I'_{A5.2}(n, m) = D^*(n, m) + D^*(n-1, m)$, it therefore remains to show that

$$D^*(n-1, m) = \sum_{x=1}^{\min(m, n-m)} D^*(n-m-1, x),$$

or equivalently, that

$$\sum_{x=1}^{\min(m, n-m-1)} D^*(n-m-1, x) = \sum_{x=1}^{\min(m, n-m)} D^*(n-m-1, x). \quad (5.7)$$

Suppose $m \leq n-m-1$. Then, $\min(m, n-m-1) = \min(m, n-m)$, and so the left and right-hand sides of (5.7) are equal. Suppose, alternatively, that $m > n-m-1$. Hence, $\min(m, n-m-1) = n-m-1$ and $\min(m, n-m) = n-m$ and so we get

$$\sum_{x=1}^{n-m} D^*(n-m-1, x) = \sum_{x=1}^{n-m-1} D^*(n-m-1, x) + D^*(n-m-1, n-m).$$

Since $n-m-1 < n-m$ we know that $D^*(n-m-1, n-m) = 0$, and therefore (5.7) is verified.

Therefore, by (5.6) and (5.7) we know that $I'_{A5.2}(n, m) = D^*(n, m) + D^*(n-1, m)$, as required. \square

Lemma 5.1 is a crucial step in our analysis of Algorithm 5.2 as it relates the number of invocations required to generate a given set of descending compositions to the function $D^*(n, m)$. Much is known about the $D^*(n, m)$ numbers, as they count the partitions of n where the largest part is m ; thus, we can then relate the number of invocations required to the partition numbers, $p(n)$. Therefore, let us formally define $I_{A5.2}(n)$ to be number of invocations of Algorithm 5.2 required to generate all $p(n)$ descending compositions of n .

§ 5.2. Recursive Algorithms

We then get the following result.

Theorem 5.2. *If $n > 1$ then $I_{A5.2}(n) = p(n) + p(n - 1)$.*

Proof. Suppose $n > 1$. To generate all descending compositions of n we invoke `RECDISC(2n, n, 1)` (see discussion above), and as $n > 1$ we can apply Lemma 5.1, to obtain $I'_{A5.2}(2n, n) = D^*(2n, n) + D^*(2n - 1, n)$, and thus $I_{A5.2}(n) = D^*(2n, n) + D^*(2n - 1, n)$. We know that $D^*(2n, n) = p(n)$, as we can clearly obtain a descending composition of n from a descending composition of $2n$ where the first part is exactly n by removing that first part. Similarly, $D^*(2n - 1, n) = p(n - 1)$, as we can remove the first part of size n from any descending composition of $2n - 1$ with first part equal to n , obtaining a descending composition of $n - 1$. Thus, the descending compositions counted by the functions $D^*(2n, n) = p(n)$ and $D^*(2n - 1, n) = p(n - 1)$. Hence, $I_{A5.2}(n) = p(n) + p(n - 1)$, completing the proof. \square

Note that in Theorem 5.2, and in many of the following analyses, we restrict our attention to values $n > 1$. This is to avoid unnecessary complication of the relevant formulas in accounting for the case where $n = 1$. In the above, if we compute $I_{A5.2}(n) = p(n) + p(n - 1)$ for $n = 1$, we arrive at the conclusion that the number of invocations required is 2, as $p(0) = 1$ by convention. In the interest of clarity we shall ignore such contingencies, as they do not affect the general conclusions we shall draw.

We generate all partitions of n by invoking `RECDISC(2n, n, 1)`. It is, however, possible to generate all descending compositions of n by iterating through all possible values x of the first part and invoking `RECDISC(n, x, 1)` for each value. This approach, however, will save only one invocation, as this is precisely what the invocation `RECDISC(2n, n, 1)` does, aside from two redundant array writes. Thus, there is no real advantage to manually iterating through the possible values for the first part; and, as we have seen in Theorem 5.2, analysis of the algorithm is greatly facilitated by examining the case of generating all descending compositions of $2n$ where the first part is exactly n .

Using Theorem 5.2 it is now straightforward to show that `RECDISC` generates all descending compositions of n in constant amortised time. To show

§ 5.2. Recursive Algorithms

that the algorithm is constant amortised time we must demonstrate that the average number of invocations of the algorithm per object generated is bounded, from above, by some constant [ER03, RECS94, NSST98, BS97, Boy05, Saw01, Rus01]. To do this, let us formally define $\bar{I}_{A5.2}(n)$ as the average number of invocations of RECDISC required to generate a descending composition of n . Clearly, as the total number of invocations is $I_{A5.2}(n)$ and the number of objects generated is $p(n)$, we have $\bar{I}_{A5.2}(n) = I_{A5.2}(n)/p(n)$. We can then prove the following corollary of Theorem 5.2.

Corollary 5.3. *Algorithm 5.2 is constant amortised time.*

Proof. By definition $\bar{I}_{A5.2}(n) = I_{A5.2}(n)/p(n)$, and by Theorem 5.2 we know that $I_{A5.2}(n) = p(n) + p(n-1)$, and so we have $\bar{I}_{A5.2}(n) = 1 + p(n)/p(n-1)$. It is well known that $p(n) > p(n-1)$ for all $n > 1$, and therefore $p(n-1)/p(n) < 1$. From this inequality we can then deduce that $\bar{I}_{A5.2}(n) < 2$, completing the proof. \square

It is useful from the perspective of comparing the algorithms for generating descending and ascending compositions to get a qualitative idea of the amount of work involved in generating descending and ascending compositions. To facilitate this comparison, we derive an asymptotic expression for the average number of invocations required to generate a descending composition using RECDISC, $\bar{I}_{A5.2}(n)$.

Corollary 5.4. *If $n > 1$ then*

$$\bar{I}_{A5.2}(n) = 1 + \frac{1 + O(n^{-1/6})}{e^{\pi/\sqrt{6n}}}. \quad (5.8)$$

Proof. From Theorem 5.2 and Corollary 5.3 we know that $\bar{I}_{A5.2}(n) = 1 + p(n-1)/p(n)$. By the asymptotic estimate for $p(n-t)/p(n)$ [Knu04c, p.11] we then get $\bar{I}_{A5.2}(n) = 1 + e^{-C/\sqrt{n}} (1 + O(n^{-1/6}))$, with $C = \pi/\sqrt{6}$. Simplifying this expression we get $\bar{I}_{A5.2}(n) = 1 + e^{-\pi/\sqrt{6n}} (1 + O(n^{-1/6}))$, as required. \square

Corollary 5.4 uses the asymptotic formulas for $p(n)$ to allow us qualitatively assess the average amount of work Algorithm 5.2 does per partition

§ 5.2. Recursive Algorithms

generated. We can see from (5.8) that this quantity is $1 + e^{-\pi/\sqrt{6n}}$ with relative error of $O(n^{-1/6})$. We therefore know that $\bar{I}_{A5.2}(n)$ will approach 2 as n becomes large. This concludes our analysis of Ruskey's algorithm.

In this subsection we have described and provided a new analysis for the most efficient known recursive descending composition generation algorithm, which is due to Ruskey [Rus01, §4.8]. Ruskey demonstrated that RECDISC is constant amortised time by reasoning about the number of children each node in the computation tree has, but does not derive the precise number of invocations involved. In this section we rigorously counted the number of invocations required to generate all descending compositions of n using this algorithm, and related the recurrence involved to the partition numbers. We then used an asymptotic formula for $p(n)$ to derive the number of invocations required to generate each partition, on average. In the next subsection we use this analysis to compare Ruskey's descending composition generator with our new ascending composition generator.

5.2.3 Comparison

In this section we have studied examples of recursive generation algorithms for ascending and descending compositions. The ascending composition generator, REASC, is a concrete implementation of the general recursive algorithm we developed in the previous chapter. The algorithm to generate descending compositions, RECDISC, is the most efficient known example of such an algorithm, and due to Ruskey [Rus01, §4.8]. We analysed both of these algorithms by counting the number of invocations of each algorithm that is required to generate all partitions of n ; as both algorithms require a constant number of operations, discounting subsequent recursive invocations, this provides us with a reasonable means of comparing the algorithms.

Performing this comparison is then a simple procedure. REASC requires $p(n)$ invocations to generate all $p(n)$ partitions of n whereas RECDISC requires $p(n) + p(n - 1)$ invocations. The asymptotics of $p(n)$ show that, as n becomes large, $p(n - 1)/p(n)$ approaches 1. Thus, we can reasonably expect the descending composition generator to require approximately twice as long

§ 5.2. Recursive Algorithms

as the ascending composition generator to generate all partitions of n .

In Table 5.1 we see a comparison of the actual time spent in generating partitions of n using two well known descending composition generators and our ascending composition generator, Algorithm 5.1. In this table we report the time spent by Algorithm 5.1 in generating all ascending compositions of n , divided by the time required by the descending composition generator in question. (We report these ratios as the actual durations are of little interest). The ratios reported are for implementations in the Java and C languages — see Section 4.4.4 for a description of the methodology adopted in making these observations. The values of n are selected such that n is the smallest integer where $p(n) > 1 \times 10^x$ and $p(n) > 5 \times 10^x$ for $6 \leq x \leq 8$. Orders of magnitude larger than these values proved to be infeasible on the experimental platform; similarly, the time elapsed in generating fewer than a million partitions was too brief to measure accurately.

The top rows of Table 5.1 report the total time required by Algorithm 5.1 to generate all partitions of n divided by the time required by Page & Wilson's algorithm [PW79, §5.5] to generate the same set of partitions. We can clearly see a large disparity in the amount of time required to generate partitions of n using these two algorithms: for instance, when generating all partitions of 109, Algorithm 5.1 requires only 11% and 15% of the time required by Page & Wilson's algorithm, in the C and Java implementations, respectively. It is possible (using the same techniques as we used to analyse Ruskey's algorithm) to show that the number of invocations of Page & Wilson's algorithm required to generate all partitions of n is given by $p(1) + \dots + p(n)$. The algorithm is manifestly inefficient, and so we shall not consider it further.

Ruskey's algorithm, as presented in Algorithm 5.2 and analysed in the previous subsection, is the most efficient known recursive descending composition generator, and results of a similar empirical investigation are shown in Table 5.1. Along with the observed ratios of the time required by RECASC and RECDASC we also report the theoretically predicted ratio of the running times: $p(n)/(p(n) + p(n-1))$. We can see from Table 5.1 that these theoretically predicted ratios agree well with the empirical evidence. We can also see that as n becomes larger, Ruskey's algorithm is tending towards taking

§ 5.3. Succession Rules

$n =$		61	72	77	90	95	109
$p(n) =$		1.12×10^6	5.39×10^6	1.06×10^7	5.66×10^7	1.05×10^8	5.42×10^8
PW	J	0.18	0.17	0.17	0.16	0.15	0.15
	C	0.11	0.13	0.13	0.12	0.12	0.11
R	J	0.56	0.56	0.56	0.55	0.55	0.55
	C	0.40	0.48	0.49	0.50	0.50	0.50
Theoretical		0.54	0.54	0.53	0.53	0.53	0.53

Table 5.1: A comparison of recursive partition generators. The ratio of the time required by our ascending composition generation algorithm and two other well-known recursive generation algorithms in the Java and C languages is shown.

twice as long as RECA_{SC} to generate the same partitions.

5.3 Succession Rules

In the previous section we analysed and compared recursive algorithms to generate all ascending and descending compositions. We found that the ascending composition generation method is significantly more efficient, despite being rather simpler to analyse and implement. Recursive generation algorithms are useful in some contexts, but there are compelling reasons for us to consider more direct iterative solutions. Besides the obvious issue of outright efficiency, iterative techniques are more flexible in how they may be adapted for a particular task. In particular, iterative techniques are far more suitable for the generator paradigm, where programming language constructs allow us to transparently simulate an arbitrarily large data structure using a special type of procedure (see Chapter 1).

In this section we consider algorithms of the form studied by Kemp, in his general treatment of the problem of generating combinatorial objects [Kem98]. As we discussed in Section 2.1.3, Kemp reduced the problem of generating combinatorial objects to the generation of all words in a formal language \mathcal{L} , and developed powerful general techniques to analyse such algorithms. Specifically, Kemp studied “direct generation algorithms” that

§ 5.3. Succession Rules

obey a simple two step procedure: (1) scan the current word from right-to-left until we find the end of the common prefix shared by the current word and its immediate successor; and (2) attach the new suffix to the end of this shared prefix. The cost of this process can be easily quantified by counting the number of ‘read’ operations required in step (1), and the number of ‘write’ operations in step (2). To determine the complexity of generating a given language, we can count the number of these operations incurred in the process of generating all words in the language.

In this section we evaluate the complexity of generating all ascending and descending compositions under these conditions. We consider only direct generation algorithms where we scan the current object to find the last part of the common prefix and write each element of the new suffix. It is important to note that under this model we only count the operations that operate on the *compositions*: we disregard all manipulations of auxiliary variables, in a deliberate attempt to determine the underlying inherent difficulty of generating ascending and descending compositions. Thus, although our analyses are based on particular algorithms, we are in fact analysing two *classes* of algorithm; that is, the classes of direct succession rule generation algorithm for ascending and descending compositions.

The section proceeds as follows. In Section 5.3.1 we examine a concrete instantiation of the succession rule developed in the previous chapter for interpart restricted compositions, and analyse the resulting algorithm. Then, in Section 5.3.2 we repeat this analysis for the well-known lexicographic succession rule for descending compositions; finally, in Section 5.3.2 we compare the average computational effort required per partition generated using both algorithms.

5.3.1 Ascending Compositions

Generating all partitions of n is an important problem [Knu04c], and any iterative solution to this problem must be based on some abstract succession rule for the encoding utilised. Succession rules for descending compositions are well-known, but, as the possibility of encoding partitions as ascending

§ 5.3. Succession Rules

compositions has not been previously considered, no such rule was previously known for ascending compositions. In Section 4.1 we developed a simple rule by which we can take any given interpart restricted composition and transform it into its immediate lexicographic successor. This rule can be used in its general form, using the relevant concrete instance of the restriction function (see Chapter 3), to generate the successor of any given ascending composition of n . But, given the importance of partition generation, we shall fully develop a more specific rule for ascending compositions in this subsection. Before we enter into our discussion of the lexicographic succession rule, it is convenient to formally define the notation involved.

Definition 5.5 (Lexicographic Minimum). *For some positive integers $m \leq n$, the function $M_{\mathcal{A}}(n, m)$ computes the lexicographically least element of the set $\mathcal{A}(n, m)$.*

Computing the lexicographically least element of an intensionally specified set of ascending compositions is essential for deriving the lexicographic successor of a given ascending composition, as we shall see. The lexicographically least element of a set of ascending compositions is easily found. For example, $M_{\mathcal{A}}(10, 3) = 334$, as we cannot have any ascending compositions that lexicographically precede 334 in $\mathcal{A}(10, 3)$; the minimum value for the first part is specified to be 3, and all parts that follow this must be ≥ 3 . Thus, for example, letting $a = 334$ and $b = 37$, we know that $a \prec b$ as $a_2 < b_2$, and while 3331 is lexicographically less than 334, it is not an ascending composition. We shall prove a simple rule for generating the lexicographically least element of any given set of ascending compositions momentarily; first, however, we define the concept of the lexicographic successor of a given ascending composition.

Definition 5.6 (Lexicographic Successor). *For any $a_1 \dots a_k \in \mathcal{A}(n) \setminus \langle n \rangle$ the function $S_{\sigma}(a_1 \dots a_k)$ computes the immediate lexicographic successor of $a_1 \dots a_k$.*

In Section 4.3 we used a recursive definition to compute the lexicographically least element of a set of interpart restricted compositions. Since the

§ 5.3. Succession Rules

unrestricted partitions are specified by the identity function, we can replace $\sigma(x)$ with x in the general recurrence (4.2) to compute the lexicographically least element of a set of ascending compositions. We then arrive at the following recurrence for $M_{\mathcal{A}}(n, m)$, defined for all $1 \leq m \leq n$:

$$M_{\mathcal{A}}(n, m) = m \cdot M_{\mathcal{A}}(n - m, m) \quad (5.9)$$

where $M_{\mathcal{A}}(n, m) = \langle n \rangle$ if $2m > n$. This recurrence is derived directly from Theorem 4.5, and so has already been proved to be correct. In this more concrete setting we can also derive a nonrecursive rule for the lexicographically least element of the set $\mathcal{A}(n, m)$, which we prove in the following lemma.

Lemma 5.2. *For all positive integers $m \leq n$, the lexicographically least element of the set $\mathcal{A}(n, m)$ is given by*

$$M_{\mathcal{A}}(n, m) = \overbrace{m \dots m}^{\mu} \langle n - \mu m \rangle, \quad (5.10)$$

where $\mu = \lfloor n/m \rfloor - 1$.

Proof. Proceed by strong induction on n .

Base case: $n = 1$. Since $1 \leq m \leq n$ and $n = 1$, then $m = 1$, and so $2m > n$. Then, by (5.9), we know that $M_{\mathcal{A}}(n, m) = \langle n \rangle$. Thus, as $\mu = 0$ when $n = 1$, (5.10) correctly computes $M_{\mathcal{A}}(n, m)$ when $n = 1$.

Induction step: Suppose, for some positive integer n that (5.10) holds true for all positive integers $m' \leq n' < n$. Suppose m is an arbitrary positive integer such that $m \leq n$. Suppose then that $2m > n$. By dividing both sides of this inequality by m , we see that $n/m < 2$, and so $\lfloor n/m \rfloor \leq 1$. Similarly, as $m \leq n$, it follows that $1 \leq n/m$, and so $1 \leq \lfloor n/m \rfloor$. Thus, $1 \leq \lfloor n/m \rfloor \leq 1$, and so $\lfloor n/m \rfloor = 1$; hence $\mu = 0$. By (5.9) $M_{\mathcal{A}}(n, m) = \langle n \rangle$, and as $\mu = 0$, zero copies of m are concatenated with $\langle n - \mu m \rangle$, and so (5.10) correctly computes $M_{\mathcal{A}}(n, m)$.

Suppose then that $2m \leq n$. By the inductive hypothesis and (5.9) we

§ 5.3. Succession Rules

have

$$M_{\mathcal{A}}(n, m) = m \cdot \overbrace{m \dots m}^{\mu'} \langle n - m - \mu' m \rangle$$

Clearly, if $\mu = \mu' + 1$, then (5.10) correctly computes the lexicographically least element of $\mathcal{A}(n, m)$. We know that $\mu' = \lfloor (n - m)/m \rfloor - 1$, which clearly gives us $\mu' = \lfloor n/m - 1 \rfloor - 1$. It follows that $\mu' = \lfloor n/m \rfloor - 2$, and, as $\mu = \lfloor n/m \rfloor - 1$ from (5.10), we have $\mu = \mu' + 1$, completing the proof. \square

Lemma 5.2 provides us with a convenient technique to compute the lexicographically least element of the set $\mathcal{A}(n, m)$, and this in turn provides us with a simple lexicographic succession rule for ascending compositions. This succession rule is expressed and proved in the following theorem.

Theorem 5.3. *If $a_1 \dots a_k \in \mathcal{A}(n) \setminus \{\langle n \rangle\}$ then*

$$S_{\mathcal{A}}(a_1 \dots a_k) = a_1 \dots a_{k-2} \overbrace{m \dots m}^{\mu} \langle n' - \mu m \rangle \quad (5.11)$$

where $m = a_{k-1} + 1$, $n' = a_{k-1} + a_k$, and $\mu = \lfloor n'/m \rfloor - 1$.

Proof. Proof is immediate by Theorem 4.4 and Lemma 5.2. \square

This succession rule is implemented in RULEASC (Algorithm 5.3), which is trivially derived from RULEGEN $_{\sigma}$ (Algorithm 4.3) by replacing $\sigma(x)$ with x . Since RULEASC is a special case of RULEGEN $_{\sigma}$ we inherit the correctness of that algorithm, Theorem 4.7. (The succession rule (5.11) can of course be implemented more literally, but algorithms involving division and multiplication are much less efficient in practice.) In RULEASC, and all subsequent algorithms we shall study, we are interested only in generating *all* partitions of n . Therefore, we shall not include any mechanism to restrict the initial parts of the compositions generated by these algorithms.

Figure 5.3 shows the array states arising from the invocation RULEASC(4). At the outset, we set $a_1 \leftarrow 0$ and $a_2 \leftarrow 4$, giving us states \mathfrak{s}_1 and \mathfrak{s}_2 . After these steps, which insert what we may call the ‘initialisation composition’ into the array, the algorithm begins iteration. After entering the main loop, we set $x \leftarrow 1$, $y \leftarrow 3$, and $k \leftarrow 1$. Then, as $x \leq y$, we enter the while loop

§ 5.3. Succession Rules

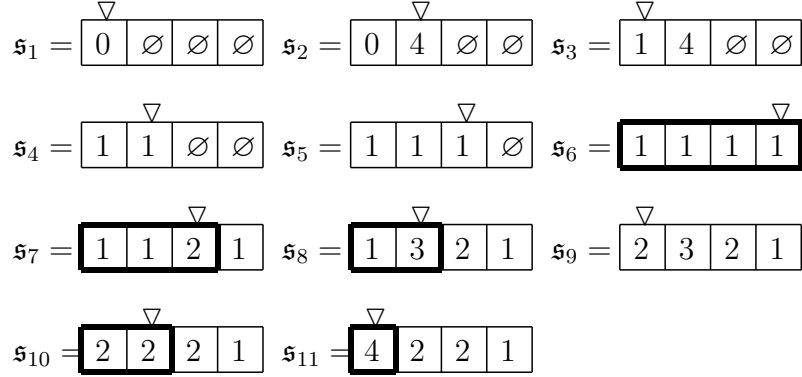


Figure 5.3: Array state-transition diagram for the succession rule based ascending composition generation algorithm.

of lines 8–12 and set $a_1 \leftarrow 1$, giving us state \mathfrak{s}_3 . Subsequently, we set $y \leftarrow 2$, $k \leftarrow 2$ and return to the head of the loop. Iteration continues along these lines, successively producing states \mathfrak{s}_4 and \mathfrak{s}_5 , until we assign $y \leftarrow 0$ and $k \leftarrow 4$. Then, as $x \not\leq y$ we do not enter the loop of lines 8–12, and instead proceed to line 13 where we assign $a_4 \leftarrow 1 + 0$, giving us state \mathfrak{s}_6 , which we then visit. As $k = 4$, we return to the head of the main loop and set $x \leftarrow 2$, $y \leftarrow 0$, and $k \leftarrow 3$; during this iteration we do not enter the loop of lines 8–12, and instead proceed directly to line 13 and assign $a_3 \leftarrow 2 + 0$, giving us state \mathfrak{s}_7 , which we then visit. The remainder of the execution continues along similar lines, and does not require further elaboration.

Analysis

The goal of our analysis is to derive a simple expression, in terms of the number of partitions of n , for the total number of read and write operations [Kem98] made in the process of generating all ascending compositions of n . Read and write operations are defined only on the compositions themselves, and not on any of the auxiliary local variables used to implement the algorithmic details. Therefore, any statement of the form $a_j \leftarrow x$ is counted as one write operation, and any statement of the form $x \leftarrow a_j$ is counted as one read. To count the number of times such statements are executed, we

§ 5.3. Succession Rules

Algorithm 5.3 RULEASC(n)

Require: $n > 0$

```
1:  $k \leftarrow 2$ 
2:  $a_1 \leftarrow 0$ 
3:  $a_2 \leftarrow n$ 
4: while  $k \neq 1$  do
5:    $y \leftarrow a_k - 1$ 
6:    $k \leftarrow k - 1$ 
7:    $x \leftarrow a_k + 1$ 
8:   while  $x \leq y$  do
9:      $a_k \leftarrow x$ 
10:     $y \leftarrow y - x$ 
11:     $k \leftarrow k + 1$ 
12:  end while
13:   $a_k \leftarrow x + y$ 
14:  visit  $a_1 \dots a_k$ 
15: end while
```

shall first determine the number of times that certain key instructions are executed; the main result is then a rudimentary application of Kirchhoff's Law [Knu72]. The key to our analysis is to observe the variable k , as this tells us exactly how many times the bodies of the inner and outer loops are executed. Therefore, let $t_6(n)$ and $t_{11}(n)$ be the number of times lines 6 and 11 are executed, respectively, in the process of generating all ascending compositions of n with Algorithm 5.3.

Lemma 5.3. *The number of times line 6 is executed during the execution of Algorithm 5.3 is given by $t_6(n) = p(n)$.*

Proof. As Algorithm 5.3 correctly visits all $p(n)$ ascending compositions of n , we know that line 14 is executed exactly $p(n)$ times. Clearly line 6 is executed precisely the same number of times as line 14, and so we have $t_6(n) = p(n)$, as required. \square

Lemma 5.4. *The number of times line 11 is executed during the execution of Algorithm 5.3 is given by $t_{11}(n) = p(n) - 1$.*

Proof. The variable k is used to control termination of the algorithm. From line 1 we know that k is initially 2, and from line 4 we know that the algorithm

§ 5.3. Succession Rules

terminates when $k = 1$. Furthermore, the value of k is modified only on lines 6 and 11. By Lemma 5.3 we know that k is decremented $p(n)$ times; it then follows immediately that k must be incremented $p(n) - 1$ times, and so we have $t_{11}(n) = p(n) - 1$, as required. \square

Using these frequency counts we can now easily count the total number of read and write operations required by RULEASC to generate all ascending compositions of n .

Theorem 5.4. *Algorithm 5.3 requires $R_{A5.3}(n) = 2p(n)$ read operations to generate the set $\mathcal{A}(n)$.*

Proof. Read operations are carried out on lines 7 and 5, which are executed $p(n)$ times each by Lemma 5.3. Thus, the total number of read operations is $R_{A5.3}(n) = 2p(n)$. \square

Theorem 5.5. *Algorithm 5.3 requires $W_{A5.3}(n) = 2p(n) - 1$ write operations to generate the set $\mathcal{A}(n)$, excluding initialisation.*

Proof. After initialisation, write operations are carried out in Algorithm 5.3 only on lines 9 and 13. Line 13 is executed $p(n)$ times by Lemma 5.3. We can also see that line 9 is executed exactly as many times as line 11, and by Lemma 5.4 we know that this value is $p(n) - 1$. Therefore, summing these contributions, we get $W_{A5.3}(n) = 2p(n) - 1$, completing the proof. \square

From Theorem 5.5 and Theorem 5.4 it is easy to see that we require an average of two read and two write operations per partition generated: since we required $2p(n)$ of both operations to generate all $p(n)$ partitions of n , the result follows immediately. Thus, for any value of n we are assured that the total time required to generate all partitions of n will be proportional to the number of partitions generated, implying that the algorithm is constant amortised time. This completes our analysis of RULEASC.

A Peripheral Result

Although not strictly relevant to our analyses of ascending and descending composition generation algorithms, another result follows directly from the

§ 5.3. Succession Rules

analysis of Algorithm 5.3. Recall that in Theorem 5.3 we proved a succession rule to compute the lexicographic successor of an arbitrary ascending composition, which, in the interest of clarity, we shall reproduce here. For any $a_1 \dots a_k \in \mathcal{A}(n) \setminus \{\langle n \rangle\}$ the lexicographic successor is given by

$$S_{\mathcal{A}}(a_1 \dots a_k) = a_1 \dots a_{k-2} \overbrace{m \dots m}^{\mu} \langle n' - \mu m \rangle$$

where $m = a_{k-1} + 1$, $n' = a_{k-1} + a_k$, and $\mu = \lfloor n'/m \rfloor - 1$. Our analysis of Algorithm 5.3 allows us prove a nonobvious theorem by summing the μ values over all ascending compositions of n . If we compare the lexicographic succession rule above and Algorithm 5.3 carefully, we realise that the μ copies of m must be inserted into the array within the inner loop of lines 8–12; and our analysis has given us the precise number of times that this happens. Therefore, we know that the sum of μ values over all ascending compositions of n (except the last composition, $\langle n \rangle$), must equal the number of write operations made in the inner loop. Having made this observation, proving the following theorem is quite simple, as we have only a few algorithmic details to factor out to reach the true structural property in question.

Theorem 5.6. *For all $n \geq 1$*

$$p(n) = \frac{1}{2} \left(1 + n + \sum_{\substack{a_1 \dots a_k \in \\ \mathcal{A}(n) \setminus \{\langle n \rangle\}}} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor \right) \quad (5.12)$$

Proof. We know from Lemma 5.4 that the total number of write operations made by Algorithm 5.3 in the inner loop of lines 8–12 is given by $p(n) - 1$. Algorithm 5.3 applies the lexicographic succession rule above to all elements of $\mathcal{A}(n) \setminus \{\langle n \rangle\}$, as well as one extra composition, which we referred to as the ‘initialisation composition’. The initialisation composition is not in the set $\mathcal{A}(n)$ as $a_1 = 0$, and so we must discount the number of writes incurred by applying the succession rule to this composition. The composition visited immediately after $0n$ is $1 \dots 1$, and so $n-1$ copies of 1 must have been inserted into the array in the inner loop during this transition. Therefore, the total

§ 5.3. Succession Rules

number of writes made within the inner loop in applying the succession rule to all elements of $\mathcal{A}(n) \setminus \{\langle n \rangle\}$ is given by $p(n) - 1 - (n - 1) = p(n) - n$. Therefore, from this result and the succession rule of Theorem 5.3 we get the following:

$$\begin{aligned} p(n) - n &= \sum_{\substack{a_1 \dots a_k \in \\ \mathcal{A}(n) \setminus \{\langle n \rangle\}}} \left(\left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor - 1 \right) \\ &= \sum_{\substack{a_1 \dots a_k \in \\ \mathcal{A}(n) \setminus \{\langle n \rangle\}}} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor - \sum_{\substack{a_1 \dots a_k \in \\ \mathcal{A}(n) \setminus \{\langle n \rangle\}}} 1 \\ &= \sum_{\substack{a_1 \dots a_k \in \\ \mathcal{A}(n) \setminus \{\langle n \rangle\}}} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor - p(n) + 1. \end{aligned}$$

From here it is easy to derive (5.12), which completes the proof. \square

We can, in fact, rather simplify (5.12) if we suppose that all $a_1 \dots a_k \in \mathcal{A}(n)$ are prefixed by a value 0. More formally, a direct consequence of Theorem 5.6 is that

$$p(n) = \frac{1}{2} \left(1 + \sum_{a_1 \dots a_k \in \mathcal{A}'(n)} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor \right), \quad (5.13)$$

where $\mathcal{A}'(n) = \{0 \cdot a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{A}(n)\}$. An example of how (5.13) computes the number of partitions of n is given in Figure 5.4, where we apply the formula to the partitions of 5. Fundamentally, what Theorem 5.6 shows us is that if we let y be the largest part and x the second largest part in an arbitrary partition of n , we can count the partitions of n by summing $\lfloor (x + y)/(x + 1) \rfloor$ over all partitions of n .

Theorem 5.6 does not represent an efficient means of computing $p(n)$; in fact it is difficult to imagine a *less* efficient means of computing the partition numbers than (5.12), given that we must first generate all partitions of n to do so. Of more interest, perhaps, than directly enumerating the unrestricted partitions of n are the indirect implications of (5.13). For example,

§ 5.3. Succession Rules

0	1	1	1	1	1	$\lfloor (1+1)/(1+1) \rfloor = 1$
0	1	1	1	2	$\lfloor (1+2)/(1+1) \rfloor = 1$	
0	1	1	3	$\lfloor (1+3)/(1+1) \rfloor = 2$		
0	1	2	2	$\lfloor (2+2)/(2+1) \rfloor = 1$		
0	1	4	$\lfloor (1+4)/(1+1) \rfloor = 2$			
0	2	3	$\lfloor (2+3)/(2+1) \rfloor = 1$			
0	5	$\lfloor (0+5)/(0+1) \rfloor = 5$				
$\therefore 1 + \sum \lfloor (a_{k-1} + a_k)/(a_{k-1} + 1) \rfloor = 14$						

Figure 5.4: Illustration of Theorem 5.6 for $n = 5$. The number of partitions of n may be obtained by summing $\lfloor (x+y)/(x+1) \rfloor$ over all partitions of n , where y is the largest part and x is the second largest part (x can be equal to y).

a straightforward consequence of Theorem 5.6 is

$$\sum_{a_1 \dots a_k \in \mathcal{A}'(n)} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor \equiv 1 \pmod{2},$$

that is, the sum of $\lfloor (x+y)/(x+1) \rfloor$ over all partitions of n (where y and x are the largest and second-largest parts, respectively) is odd. Arithmetic properties of enumeration functions are of great interest in the theory of partitions [AO01]. According to Hardy and Wright [HW54, §19.12], “in spite of the simplicity of the definition of $p(n)$, very little is known about its arithmetic properties. There is, for example, no simple criterion for deciding whether $p(n)$ is odd or even.” While our incidental result does not bring us any closer to answering the particular question of the parity of $p(n)$, it does, perhaps, raise some other interesting questions; and these question are raised through the analysis of our generation algorithm, `RULEASC`.²

Our brief digression into the more abstract theory of partitions aside, we can now return to our analysis of ascending and descending composition generation algorithms. Having now developed and analysed a simple lex-

²It unclear to the author if Theorem 5.6 is known or not. It seems rather unlikely, however, that the result has been proved using the same means, and is therefore a novel contribution in either case.

§ 5.3. Succession Rules

icographic succession rule to generate all ascending compositions of n , we now turn, in the next subsection, to examine the problem of generating all *descending* compositions of n , using a commensurable method.

5.3.2 Descending Compositions

Up to this point we have considered only algorithms that generate compositions in lexicographic order. The majority of descending composition generation algorithms, however, visit compositions in *reverse* lexicographic order (McKay [McK70] refers to it as the ‘natural order’ for partitions). For instance, in reverse lexicographic order the descending compositions of 5 are 5, 41, 32, 311, 221, 2111, 11111. There are many different presentations of the succession rule required to transform a descending composition from this list into its immediate successor: see Andrews [And76, p.230], Knuth [Knu04c, p.1], Nijenhuis & Wilf [NW78, p.65–68], Page & Wilson [PW79, §5.5], Skiena [Ski90, p.52], Stanton & White [SW86, p.13], Wells [Wel71, p.150] or Zoghbi & Stojmenović [ZS98]. No analysis of this succession rule in terms of the number of read and write operations [Kem98] involved has been published, and in this section we analyse a basic implementation of the rule (we analyse more sophisticated techniques in Section 5.4.2).

If we formally define $S_{\mathcal{D}}(d_1 \dots d_k)$ to be the immediate lexicographic predecessor of a $d_1 \dots d_k \in \mathcal{D}(n) \setminus 1 \dots 1$, the succession rule can be formulated as follows. Given a descending composition $d_1 \dots d_k$ where $d_1 \neq 1$, we obtain the next composition in the ordering by applying the transformation

$$S_{\mathcal{D}}(d_1 \dots d_k) = d_1 \dots d_{q-1} \overbrace{m \dots m}^{\mu} \langle n' - \mu m \rangle \quad (5.14)$$

where q is the rightmost non-1 value (i.e., $d_j > 1$ for $1 \leq j \leq q$ and $d_j = 1$ for $q < j \leq k$), $m = d_q - 1$, $n' = d_q + k - q$ and $\mu = \lfloor n'/m \rfloor - \lfloor n' \bmod m = 0 \rfloor$. This presentation can readily be derived from the treatments cited in the opening paragraph of this subsection.

As an example, consider the descending composition $d_1 d_2 = 32$. Finding the rightmost non-1 value in this case is easy, as there are no 1s in the

§ 5.3. Succession Rules

Algorithm 5.4 RULEDESC(n)

Require: $n > 0$

```

1:  $d_1 \leftarrow n$ 
2:  $k \leftarrow 1$ 
3: visit  $d_1$ 
4: while  $k \neq n$  do
5:    $\ell \leftarrow k$ 
6:    $m \leftarrow d_q$ 
7:   while  $m = 1$  do
8:      $k \leftarrow k - 1$ 
9:      $m \leftarrow d_q$ 
10:  end while
11:   $n' \leftarrow m + \ell - k$ 
12:   $m \leftarrow m - 1$ 
13:  while  $m < n'$  do
14:     $d_k \leftarrow m$ 
15:     $n' \leftarrow n' - m$ 
16:     $k \leftarrow k + 1$ 
17:  end while
18:   $d_k \leftarrow n'$ 
19:  visit  $d_1 \dots d_k$ 
20: end while

```

composition. Thus, $q = 2$, and therefore $m = d_2 - 1 = 1$. After computing the value of m we calculate $n' = d_q + k - q = 2 + 2 - 2 = 2$, and we therefore find that $\mu = \lfloor 2/1 \rfloor - \lfloor 2 \bmod 1 \rfloor = 2 - 1 = 1$. Thus, we insert one copy of 1 after d_{q-1} , and append $n' - \mu m = 2 - 1 \times 1$ to the end of the resulting composition, giving us $S_{\mathcal{D}}(32) = 311$. In another example, consider the descending composition $d_1 \dots d_7 = 5531111$ of 17. In this case $q = 3$, as the last four parts are 1. Hence, $m = 2$, $n' = 7$ and so $\mu = 3$; we then get $S_{\mathcal{D}}(5531111) = 552221$.

The succession rule (5.14) is implemented in RULEDESC (Algorithm 5.4), where each iteration of the main loop implements a single application of the rule. We begin by visiting the composition $\langle n \rangle$ on line 3, and then successively apply the rule until the array d contains the composition $1 \dots 1$. The internal loop of lines 7–9 implements a right-to-left scan for the largest index q such that $d_q > 1$, and then on lines 11 and 12 we calculate the values of m and

§ 5.3. Succession Rules

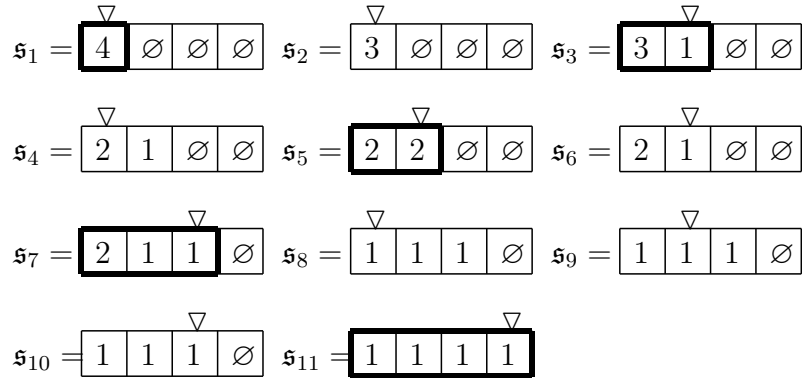


Figure 5.5: Array state-transition diagram for the succession rule based descending composition generation algorithm. Indices marked with the ∇ symbol are the indices in which a value is assigned to change the state of the array, and array segments outlined in bold are the portions that are visited.

n' based on this information. (We have not followed the precise formulation of (5.14), but it can be easily verified that the effect is the same, and the particular form of RULEDESC greatly facilitates our analysis.) Then, the loop of lines 13–17 straightforwardly inserts μ copies of m into the array from index d_q onwards, and on line 16 appends $n' - \mu m$ to the end of the resulting composition, before visiting $d_1 \dots d_k$ on line 19.

The sequence of array states traversed by RULEDESC in the process of generating all descending compositions of 4 is given in Figure 5.5. As we have already discussed the operation of the succession rule in detail, we shall not labour the point with an exhaustive description of the internal workings of the algorithm. One notable aspect of the operation of RULEDESC that we have not discussed is, however, apparent in this illustration. If we consider states \mathfrak{s}_9 and \mathfrak{s}_{10} we see an undesirable property of Algorithm 5.4 manifesting itself. In these states we insert the value 1 into array elements d_2 and d_3 respectively; but these indices *already* contain the value 1, and so these write operations are redundant. This behaviour proves to be a key feature of the algorithm, as we shall see presently.

§ 5.3. Succession Rules

Analysis

The variable k is the key to the analysis of RULEDESC, as it controls the termination of the algorithm, and is updated only via increment (line 16) and decrement (line 8) operations. Thus, by counting the number of increment and decrement operations required by the algorithm to generate all descending compositions of n , we determine the total number of iterations of the loops involved. Therefore, let $t_8(n)$ be the total number of times line 8 is executed in the progress of RULEDESC(n). The following lemma then derives this number in terms of the number of partitions of n .

Lemma 5.5. *The number of times line 8 is executed during the execution of Algorithm 5.4 is given by $t_8(n) = 1 - n + \sum_{x=1}^{n-1} p(x)$.*

Proof. As exactly one descending composition is visited per iteration of the outer while loop, we know that upon reaching line 6 there is a complete descending composition of n contained in $d_1 \dots d_k$. Furthermore, as $d_1 \geq \dots \geq d_k$, we know that all parts of size 1 are at the end of the composition, and so it is clear that line 7 will be executed exactly once for each part of size 1 in any given composition. As we visit the compositions at the end of the loop and we terminate when $k = n$ we will not reach line 5 when the composition in question consists of n copies of 1 (as this is the lexicographically least, and hence the last descending composition in reverse lexicographic order). Thus, line 7 will be executed exactly as many times as there are parts of size 1 in all partitions of n , minus the n 1s contained in the last composition. It is well known [Hon85, p.8] that the number of 1s in all partitions of n is $1 + p(1) + \dots + p(n-1)$, and therefore we see that line 7 is executed exactly $1 - n + \sum_{x=1}^{n-1} p(x)$, as required. \square

Lemma 5.5 derives the total number of decrement operations carried out on the variable k , and arrives at its conclusion by using a well known combinatorial property of the set of all partitions of n . We can now use Lemma 5.5 to derive the total number of increment operations. Letting $t_{16}(n)$ be the total number of times line 16 is executed by RULEDESC(n), we get the following lemma.

§ 5.3. Succession Rules

Lemma 5.6. *The number of times line 16 is executed during the execution of Algorithm 5.4 is given by $t_{16}(n) = \sum_{x=1}^{n-1} p(x)$.*

Proof. The variable k is used to control termination of Algorithm 5.4: the algorithm begins with $k = 1$ and terminates when $k = n$. Examining Algorithm 5.4 we see that k is modified on only two lines: it is incremented on line 16 and decremented on line 8. Thus, we must have $n - 1$ more increment operations than decrements; by Lemma 5.5 there are exactly $1 - n + \sum_{x=1}^{n-1} p(x)$ decrement operations, and so we see that line 14 is executed $\sum_{x=1}^{n-1} p(x)$ times, as required. \square

Lemma 5.6 tells us the total number of times the loop of lines 13–17 is executed. It is now straightforward to determine the total number of read and write operations incurred by RULEDESC(n).

Theorem 5.7. *Algorithm 5.4 requires $R_{A5.4}(n) = \sum_{x=1}^n p(x) - n$ read operations to generate the set $\mathcal{D}(n)$.*

Proof. Read operations are performed on lines 6 and 9 of Algorithm 5.4. By Lemma 5.5 we know that line 8 is executed $1 - n + \sum_{x=1}^{n-1} p(x)$ times, and so line 9 is executed an equal number of times. Clearly line 6 is executed $p(n) - 1$ times, and so we get a total of $R_{A5.4}(n) = \sum_{x=1}^n p(x) - n$, as required. \square

Theorem 5.8. *Algorithm 5.4 requires $W_{A5.4}(n) = \sum_{x=1}^n p(x) - 1$ write operations to generate the set $\mathcal{D}(n)$, excluding initialisation.*

Proof. The only occasions in Algorithm 5.4 where a value is written to the array d are lines 14 and 18. By Lemma 5.6 we know that line 16 is executed exactly $\sum_{x=1}^{n-1} p(x)$, and it is straightforward to see that line 14 is executed precisely the same number of times. As we visit exactly one composition per iteration of the outer while loop, and all descending compositions except the composition $\langle n \rangle$ are visited with this loop, we then see that line 18 is executed $p(n) - 1$ times in all. Therefore, summing these contributions we get $W_{A5.4}(n) = \sum_{x=1}^{n-1} p(x) + p(n) - 1 = \sum_{x=1}^n p(x) - 1$ as required. \square

Theorems 5.7 and 5.8 derive the precise number of read and write operations required to generate all descending compositions of n using Algorithm 5.4, and this completes our analysis of the algorithm. We shall discuss

§ 5.3. Succession Rules

the implications of these results in the next subsection, where we compare the total number of read and write operations required by $\text{RULEASC}(n)$ and $\text{RULEDESC}(n)$.

After the incidental result of Theorem 5.6, we are alerted to the possibility of an interesting structural result emerging from our analysis of generation algorithms. In Theorem 5.6 we showed that the number of partitions of n may be computed by summing $\lfloor (x+y)/(x+1) \rfloor$ over all partitions of n , where y is the largest part in a given partition and x is the second largest part ($x \leq y$). An analogous result is implied by the analysis of Algorithm 5.4, although it is more complex. From the succession rule for descending compositions (5.14) we know that μ copies of a certain value are inserted into the composition at each transition, and from our analysis we know that these values must be inserted in the inner loop of lines 11–15 of Algorithm 5.4. By Lemma 5.6 we know the precise number write operations incurred within this loop when summed over all $p(n) - 1$ transitions made by the algorithm. Combining knowledge with the succession rule we get the following equation.

$$\sum_{x=1}^{n-1} p(x) = \sum_{\substack{d_1 \dots d_k \in \\ \mathcal{D}(n) \setminus \{1 \dots 1\}}} (\lfloor n'/m \rfloor - [n' \bmod m = 0]) \quad (5.15)$$

where $m = d_q - 1$, $n' = d_q + k - q$ and q is the rightmost non-1 value (i.e., $d_j > 1$ for $1 \leq j \leq q$ and $d_j = 1$ for $q < j \leq k$). We must exclude the composition $1 \dots 1$, (i.e. the composition consisting of n 1s) since the algorithm terminates after this composition is visited, and so we do not transition over it. Unfortunately, the combinatorial interpretation of this identity is not clear, and it is not obvious how (5.15) may be subsequently simplified. We shall, therefore, not pursue the matter further, having at least identified the possibility for future work.

5.3.3 Comparison

In this section we have examined the basic succession rules for both ascending and descending compositions. We developed a new succession rule

§ 5.3. Succession Rules

for ascending compositions based on our general methods of the previous chapter, and also examined the well-known and widely-cited succession rule for generating descending compositions in reverse lexicographic order. The amount of effort required to generate the set of all partitions using these rules can be quantified by counting the total number of read and write operations incurred by the corresponding algorithms.

In this section we developed two algorithms. The first algorithm we considered, RULEASC (Algorithm 5.3), generates ascending compositions of n ; the second algorithm, RULEDESC (Algorithm 5.4), generates descending compositions of n . We analysed the total number of read and write operations required by these algorithms to generate all partitions of n by iteratively applying the succession rule involved. The totals obtained, disregarding unimportant trailing terms, for the ascending composition generator are summarised as follows.

$$R_{A5.3}(n) \approx 2p(n) \quad \text{and} \quad W_{A5.3}(n) \approx 2p(n) \quad (5.16)$$

That is, we require $2p(n)$ operations of the form $x \leftarrow a_j$ and $2p(n)$ operations of the form $a_j \leftarrow x$ to generate all partitions of n using the ascending composition generator. Turning then to the descending composition generator, we obtained the following totals, again removing insignificant trailing terms.

$$R_{A5.4}(n) \approx \sum_{x=1}^n p(x) \quad \text{and} \quad W_{A5.4}(n) \approx \sum_{x=1}^n p(x) \quad (5.17)$$

These totals would appear to indicate a large disparity between the algorithms, but we must examine the asymptotics of $\sum_{x=1}^n p(x)$ to determine whether or not this is significant. We shall do this in terms of the average number of read and write operations per partition which is implied by these totals.

We know the total number of read and write operations required to generate all $p(n)$ partitions of n using both algorithms. Thus, to determine the expected number of read and write operations required to transform the average partition into its immediate successor we must divide these totals

§ 5.3. Succession Rules

by $p(n)$. In the case of the ascending composition generation algorithms this is trivial, as both expressions are of the form $2p(n)$, and so dividing by $p(n)$ plainly yields the value 2. Determining the average number of read and write operations using the succession rule for descending compositions is more difficult, however, as both expressions involve a factor of the form $\sum_{x=1}^n p(x)$.

Using the asymptotic expressions for $p(n)$ we can get a qualitative estimate of these functions. Odlyzko [Odl96, p.1083] derived an estimate for the value of sums of partition numbers which can be stated as follows

$$\sum_{x=1}^n p(x) = \frac{e^{\pi\sqrt{2n/3}}}{2\pi\sqrt{2n}} (1 + O(n^{-1/6})).$$

Then, dividing this by the asymptotic expression for $p(n)$ we get the following approximation

$$\frac{1}{p(n)} \sum_{x=1}^n p(x) \approx 1 + \frac{\sqrt{6n}}{\pi}, \quad (5.18)$$

which, although crude, is sufficient for our purposes. The key feature of (5.18) is that the value is not constant: it is $O(\sqrt{n})$. Using this approximation we obtain the following values for the number of read and write operations expected to transform a random partition of n into its successor.

	Reads	Writes
Ascending	2	2
Descending	$1 + 0.78\sqrt{n}$	$1 + 0.78\sqrt{n}$

To make the difference between these two succession rules clear, let us take a hypothetical example. Suppose we are presented with a random partition of n , and requested to produce its immediate successor using succession rules for either ascending or descending compositions. If we choose to use the rule for ascending compositions we expect to perform approximately two read and two write operations, *irrespective* of the value of n . On the other hand, if we choose to use the succession rule for descending compositions we can expect to perform around $1 + \sqrt{6n}/\pi$ read and $1 + \sqrt{6n}/\pi$ write operations:

§ 5.3. Succession Rules

the number of read and write operations required is dependent on the value of n .

We can see the qualitative difference between the algorithms by examining their read and write tapes in Figure 5.6. The tapes in question are generated by imagining that read and write heads mark a tape each time one of these operations is made. The horizontal position of each head is determined by the index of the array element involved. The tape is advanced one unit each time a composition is visited, and so we can see the number of read and write operations required for each individual partition generated. Regarding Figure 5.6 then, and examining the read tape for RULEASC, we can see that every partition requires exactly 2 reads; in contrast, the read tape for RULEDESC shows a maximum of $n - 1$ read operations per partition, and this oscillates rapidly as we move along the tape. Similarly, the write tape for RULEASC shows that we sometimes need to make a long sequence of write operations to make the transition in question, but that these are compensated for — as our analysis has shown — by the occasions where we need only one write. The behaviour of the write head in RULEDESC is very similar to that of its read head, and we again see many transitions where a large number of writes are required.

To conclude this section, we can make the following reasonable assertion. If we wish to generate all partitions of n using a simple succession rule of the form studied by Kemp [Kem98], it is *imperative* that we use ascending compositions rather than descending compositions. The difference in the algorithms is not due to an algorithmic nuance: it reflects of a fundamental property of the objects in question. The total suffix length [Kem98] of descending compositions is much greater than that of ascending compositions, because in many descending composition the suffix consists of the sequence of 1s; and we know that the total number of 1s in all partitions of n is $\sum_{x=1}^{n-1} p(x)$.

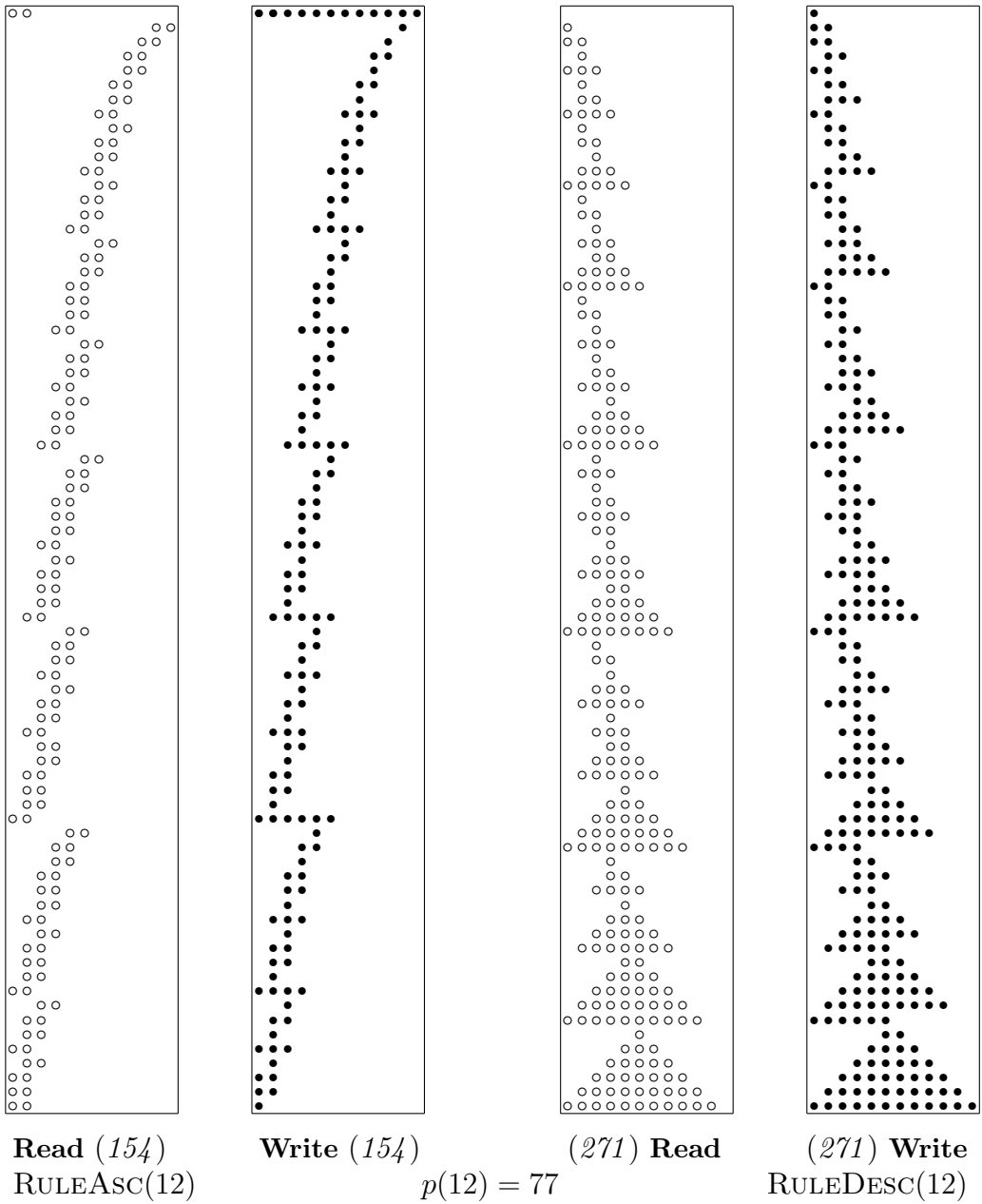


Figure 5.6: Read and write tapes for the direct implementations of succession rules to generate ascending and descending compositions. On the left we have the read and write tapes for the ascending composition generator, Algorithm 5.3; on the right, then, are the corresponding tapes for the descending composition generator, Algorithm 5.4. In both cases, the traces correspond to the read and write operations carried out in generating all partitions of 12.

5.4 Accelerated Algorithms

In the previous section we developed two partition generation algorithms: an ascending composition generator (RULEASC) and a descending composition generator (RULEDESC). Both of these algorithms are direct and literal implementations of their respective succession rules. In the case of RULEASC, the succession rule is a concrete instantiation of the rule we developed in Section 4.3; for RULEDESC, the succession rule is the well-known mechanism for deriving the lexicographic predecessor of a descending composition. Our analysis of these algorithms showed that RULEASC is far more efficient than RULEDESC. This is, however, not a fair representation of descending composition generation algorithms. Several techniques have been developed that dramatically reduce the number of read and write operations required to generate all descending compositions of n .

In this section we study algorithms utilising structural properties of the sets of ascending and descending compositions to reduce the number of read and write operations required. The algorithms presented are the most efficient known examples of ascending and descending composition generators, ensuring that we have a fair comparison of the algorithms arising from the two candidate encodings for partitions. In Section 5.4.1 we develop a new ascending composition generator that requires fewer read operations than RULEASC; this is achieved by utilising some auxiliary theory we developed in Section 4.4.1. Then, in Section 5.4.2 we study the most efficient known descending composition generation algorithm, due to Zoghbi & Stojmenović [ZS98], which requires far fewer read and write operations than RULEDESC. In Section 5.4.3, we compare these two algorithms to determine which of the two is more efficient. Finally, Section 5.4.4 we empirically compare the ascending composition generator with the most efficient examples of descending composition generators, including algorithms that utilise the multiplicity and part-count representations.

5.4.1 Ascending Compositions

In this subsection we improve on RULEASC (Algorithm 5.3) by applying the theory of ‘terminal’ and ‘nonterminal’ compositions, which we developed in a more general context in Section 4.4.1. To enable us to fully analyse the resulting algorithm we require an expression to enumerate terminal ascending compositions in terms of $p(n)$, the number of partitions of n . In the opening part of this subsection we develop the theory of terminal and nonterminal compositions in this more concrete context. A byproduct of this analysis is a new proof for a partition identity on the number of partitions where the largest part is less than twice the second largest part. After developing this necessary theory, we move on to the description of the algorithm itself, and its subsequent analysis.

Ascending Composition Blocks

In Section 4.4.1 we developed the theory of ‘terminal’ and ‘nonterminal’ compositions. This theory allowed us to develop a more efficient implementation of the abstract succession rule for interpart restricted compositions. By proving certain lexicographic contiguity results, we can make many of the necessary transitions more efficiently than is implied by the general succession rule. The greater the number of transitions that are effected via these special cases, the greater the gain in efficiency we can expect; and the number of times the special cases can be invoked is directly related to the number of terminal compositions within the set of interpart restricted compositions.

In general, determining the number of terminal compositions within a given set of interpart restricted compositions (in terms of the total number of compositions) is a difficult problem. For the particular instance of the restriction function that we are interested in for the purposes of this chapter (i.e. the ascending compositions, where $\sigma(x) = x$) we can completely solve this problem, allowing us to further elucidate the computational properties of the accelerated generation algorithm. In Section 4.4.1 we formally defined terminal and nonterminal compositions for interpart restricted compositions. It is useful to override these definitions with their concrete equivalents here.

§ 5.4. Accelerated Algorithms

Definition 5.7 (Terminal Ascending Composition). *For some positive integer n , an ascending composition $a_1 \dots a_k \in \mathcal{A}(n)$ is terminal if $k = 1$ or $2a_{k-1} \leq a_k$. Let $\mathcal{T}_{\mathcal{A}}(n, m)$ denote the set of terminal compositions in $\mathcal{A}(n, m)$, and $T_{\mathcal{A}}(n, m)$ denote the cardinality of this set (i.e. $T_{\mathcal{A}}(n, m) = |\mathcal{T}_{\mathcal{A}}(n, m)|$).*

Definition 5.8 (Nonterminal Ascending Composition). *For some positive integer n , $a_1 \dots a_k \in \mathcal{A}(n)$ is nonterminal if $k > 1$ and $2a_{k-1} > a_k$. Let $\mathcal{N}_{\mathcal{A}}(n, m)$ denote the set of nonterminal compositions in $\mathcal{A}(n, m)$, and let $N_{\mathcal{A}}(n, m)$ denote the cardinality of this set (i.e. $N_{\mathcal{A}}(n, m) = |\mathcal{N}_{\mathcal{A}}(n, m)|$).*

Definition 5.7 and Definition 5.8 are directly derived from their general counterparts in Section 4.4.1. An interpart restricted composition $a_1 \dots a_k \in \mathcal{C}_{\sigma}(n)$ is defined to be terminal if $k = 1$ or $\sigma(a_{k-1}) + \sigma(\sigma(a_{k-1})) \leq a_k$; then, replacing for $\sigma(x) = x$ we obtain Definition 5.7. We shall often refer to a grouping of terminal and nonterminal compositions as a ‘block’ of compositions; this informal name corresponds to the actual application of terminal and nonterminal compositions in the generation algorithm we shall develop presently.

The idea behind terminal and nonterminal compositions, and the corresponding block-based structure of the set $\mathcal{A}(n)$, is quite simple and can be easily seen by means of an example.

$$\begin{array}{c}
 \text{Nonterminal Compositions} \\
 \swarrow \\
 \begin{array}{|c|c|c|c|}
 \hline
 13 & 12 & 11 & 10 \\
 \hline
 7 & 8 & 9 & 10 \\
 \hline
 \end{array} \\
 \searrow \\
 \text{Terminal Composition}
 \end{array} \tag{5.19}$$

In this example we see the lexicographically last five ascending compositions of 20, which together constitute the last ascending composition block of the set $\mathcal{A}(20)$. The terminology ‘terminal’ and ‘nonterminal’ compositions becomes a little clearer when viewed in this light. The composition $\langle 20 \rangle$ is terminal as it contains only one part (see Definition 5.7); it is also the last composition in the block that we have shown, which provides the motivation for the term ‘terminal’. The compositions preceding $\langle 20 \rangle$ are nonterminal as, in each case, the last part is less than twice the second-last part. Our ex-

§ 5.4. Accelerated Algorithms

ample here may be a little misleading in one respect: terminal compositions can, of course, be of any length. If, for example, we prefixed 1 to all of the compositions in (5.19), the resulting compositions would still be nonterminal and terminal in the same way, as $2 \times 1 \leq 20$, ensuring that $\langle 1 \rangle \langle 20 \rangle$ is terminal, and prefixing any values to a nonterminal composition does not alter its nonterminal status.

The example in (5.19) immediately suggests an efficient means of generating such blocks. It is easy to see that we can transition between the nonterminal compositions by incrementing and decrementing the values of the second-last and last parts, respectively. When we cannot perform this operation (i.e. when it causes the second-last part to be greater than the last part) we simply add these two values together and visit the resulting terminal composition. This is the informal basis of our improved generator, but to accurately predict the behaviour of the algorithm we need to determine the number of terminal compositions in the set $\mathcal{A}(n)$.

In Section 4.4.1 we developed a recurrence equation to count the terminal compositions in a given set of interpart restricted compositions. We shall now study the concrete instantiation of this general recurrence, which enumerates the terminal ascending compositions of n . We then solve this recurrence in terms of the partition numbers $p(n)$, and discover that the terminal ascending compositions of n are enumerated by $p(n) - p(n - 2)$.

We begin by developing an iterated recurrence to enumerate the ascending compositions of n . Iterated recurrences are useful in this situation as all of the recursive invocations are in terms of smaller values of n , allowing us to apply the method of strong induction to prove the required results. To determine the upper bounds of the summations involved, we require the following lemma concerning the floor function [GKP94, §3.1].

Lemma 5.7. *For all positive integers x , m and n , $mx \leq n \iff x \leq \lfloor n/m \rfloor$.*

Proof. Suppose x , m and n are positive integers. Suppose also that $mx \leq n$. It follows immediately from this premise that $x \leq n/m$, and as x is an integer (and so x cannot be a fractional value between $\lfloor n/m \rfloor$ and n/m), we have $x \leq \lfloor n/m \rfloor$; thus $mx \leq n \implies x \leq \lfloor n/m \rfloor$.

§ 5.4. Accelerated Algorithms

Suppose $x \leq \lfloor n/m \rfloor$. Then, $mx \leq m\lfloor n/m \rfloor$, and since $n/m \leq \lfloor n/m \rfloor$, $mx \leq n$. Therefore, $x \leq \lfloor n/m \rfloor \implies mx \leq n$. Thus, $mx \leq n \iff x \leq \lfloor n/m \rfloor$, as required. \square

Using this lemma we can now use our general iterative recurrence of Section 3.2.1 to derive an equation where the upper and lower bounds on the summation are in closed form. Thus, letting $A(n, m)$ denote the number of ascending compositions of n where the initial part is at least m (see Section 5.1 for a full definition), we obtain the following result.

Theorem 5.9. *For all positive integers $m \leq n$, $A(n, m)$ satisfies the recurrence*

$$A(n, m) = 1 + \sum_{x=m}^{\lfloor n/2 \rfloor} A(n-x, x). \quad (5.20)$$

Proof. Replacing for $\sigma(x) = x$ in (3.7) we get

$$A(n, m) = 1 + \sum_{\substack{x \geq m \\ 2x \leq n}} A(n-x, x).$$

The condition on this summation implies that we sum over all values x such that $x \geq m$ and $2x \leq n$. From Lemma 5.7 we know that $2x \leq n \implies x \leq \lfloor n/2 \rfloor$, and so we must sum over all values x such that $m \leq x \leq \lfloor n/2 \rfloor$. Thus, we get $A(n, m) = 1 + \sum_{x=m}^{\lfloor n/2 \rfloor} A(n-x, x)$, as required. \square

We require a similar recurrence to enumerate the terminal ascending compositions, and so we let $T_{\mathcal{A}}(n, m)$ denote the number of terminal compositions in the set $\mathcal{A}(n, m)$. The terminal ascending compositions are a subset of the ascending compositions, and the construction rule implied is the same: the number of terminal ascending compositions of n where the initial part is *exactly* m is equal to the number of terminal compositions of $n-m$ with initial part *at least* m . The only difference, then, between the recurrences for ascending compositions and terminal ascending compositions occurs in the boundary conditions. The recurrence is proved formally as follows.

§ 5.4. Accelerated Algorithms

Theorem 5.10. *For all positive integers $m \leq n$, $T_{\mathcal{A}}(n, m)$ satisfies the recurrence*

$$T_{\mathcal{A}}(n, m) = 1 + \sum_{x=m}^{\lfloor n/3 \rfloor} T_{\mathcal{A}}(n-x, x). \quad (5.21)$$

Proof. Replacing for $\sigma(x) = x$ in Theorem 5.10 (Section 4.4.1) we obtain the recurrence

$$T_{\mathcal{A}}(n, m) = T_{\mathcal{A}}(n-m, m) + T_{\mathcal{A}}(n, m+1) \quad (5.22)$$

where $T_{\mathcal{A}}(n, m) = 1$ if $3m > n$. By the contrapositive of Lemma 5.7 we know that $3m > n \implies m > \lfloor n/3 \rfloor$, and so we know that $T_{\mathcal{A}}(n, m) = 1$ if $m > \lfloor n/3 \rfloor$. Then, expanding the term $T_{\mathcal{A}}(n, m+1)$ in (5.22) we see that

$$\begin{aligned} T_{\mathcal{A}}(n, m) &= T_{\mathcal{A}}(n-m, m) + T_{\mathcal{A}}(n-(m+1), m+1) + \cdots \\ &\quad + T_{\mathcal{A}}(n-\lfloor n/3 \rfloor, \lfloor n/3 \rfloor) + 1, \end{aligned}$$

as each term where $m \leq \lfloor n/3 \rfloor$ corresponds to the recursive expansion of $T_{\mathcal{A}}(n-m, m)$. Gathering the relevant terms into a summation we get $T_{\mathcal{A}}(n, m) = 1 + \sum_{x=m}^{\lfloor n/3 \rfloor} T_{\mathcal{A}}(n-x, x)$, completing the proof. \square

With Theorem 5.9 and Theorem 5.10 we have proved the correctness of the enumeration functions for ascending compositions and terminal ascending compositions, respectively. Before we move onto the main result, where we prove that $T_{\mathcal{A}}(n, m) = A(n, m) - A(n-2, m)$, we require some auxiliary results which simplify the proof of this assertion. In Lemma 5.8 we prove an equivalence between logical statements of a particular form involving the floor function, which is useful in Lemma 5.9; the latter lemma then provides the main inductive step in our proof of the central theorem of this section.

Lemma 5.8. *If x, m and n are positive integers then $x \leq \lfloor (n-x)/m \rfloor \iff x \leq \lfloor n/(m+1) \rfloor$.*

Proof. Suppose x, m and n are positive integers. Suppose $x \leq \lfloor (n-x)/m \rfloor$. Thus, $x \leq (n-x)/m$, and so $x \leq n/(m+1)$. Then, as $\lfloor n/(m+1) \rfloor \leq n/(m+1)$ and x is an integer, we know that $x \leq \lfloor n/(m+1) \rfloor$, and so $x \leq \lfloor (n-x)/m \rfloor \implies x \leq \lfloor n/(m+1) \rfloor$.

§ 5.4. Accelerated Algorithms

Suppose that $x \leq \lfloor n/(m+1) \rfloor$. Then, $x \leq n/(m+1)$, and so $x \leq (n-x)/m$. Once again, as x is an integer it is apparent that $x \leq \lfloor (n-x)/m \rfloor \leq (n-x)/m$, and so $x \leq \lfloor n/(m+1) \rfloor \implies x \leq \lfloor (n-x)/m \rfloor$. Therefore, as $x \leq \lfloor (n-x)/m \rfloor \implies x \leq \lfloor n/(m+1) \rfloor$ and $x \leq \lfloor n/(m+1) \rfloor \implies x \leq \lfloor (n-x)/m \rfloor$ we see that $x \leq \lfloor (n-x)/m \rfloor \iff x \leq \lfloor n/(m+1) \rfloor$, as required. \square

We shall momentarily prove that $T_{\mathcal{A}}(n) = p(n) - p(n-2)$, but our general argument requires that we restrict our attention to values of $n \geq 3$. If we consider $T_{\mathcal{A}}(1)$ and $T_{\mathcal{A}}(2)$ for a moment, however, we can see that the identity still holds. By convention $p(0) = 1$ and $p(n) = 0$ for all $n < 0$. Thus, our general rule asserts that $T_{\mathcal{A}}(1) = 1 - 0$, and $T_{\mathcal{A}}(2) = 2 - 1$, both of which are correct. Therefore, we may proceed with our proof, safe in the knowledge that we are proving our assertion for all positive integers. The key observation we require for our general argument is that $A(n-x, x) = 1$ if $x > \lfloor n/3 \rfloor$ and $x < n$; this is a direct consequence of the recurrence (5.20). The following lemma uses this observation to provide the key inductive step of the proof.

Lemma 5.9. *For all positive integers $n > 3$*

$$\sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor n/2 \rfloor} A(n-x, x) = 1 + \sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor (n-2)/2 \rfloor} A(n-2-x, x). \quad (5.23)$$

Proof. Suppose $n > 3$ and $1 \leq m \leq n$, and consider the left-hand side of (5.23). We know that $A(n, m) = 1$ if $m > \lfloor n/2 \rfloor$, as the summation in recurrence (5.20) will be empty. By the contrapositive of Lemma 5.8 we know that $x > \lfloor (n-x)/2 \rfloor \iff x > \lfloor n/3 \rfloor$, and we therefore know that each term in the summation of the left-hand side of (5.23) is equal to 1. Thus, we see that

$$\sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor n/2 \rfloor} A(n-x, x) = \lfloor n/2 \rfloor - \lfloor n/3 \rfloor - 1. \quad (5.24)$$

Similarly, as $x > \lfloor n/3 \rfloor \implies x > \lfloor (n-x)/2 \rfloor$, it clearly follows that $x > \lfloor n/3 \rfloor \implies x > \lfloor (n-x)/2 \rfloor - 1$, or $x > \lfloor n/3 \rfloor \implies x > \lfloor (n-2-x)/2 \rfloor$. Thus, each term in the summation on the right-hand side of (5.23) must also

§ 5.4. Accelerated Algorithms

equal 1, and so we get

$$\begin{aligned}
 1 + \sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor (n-2)/2 \rfloor} A(n-2-x, x) &= 1 + \lfloor (n-2)/2 \rfloor - \lfloor n/3 \rfloor - 1 \\
 &= \lfloor n/2 \rfloor - \lfloor n/3 \rfloor - 1.
 \end{aligned} \tag{5.25}$$

Therefore, as (5.24) and (5.25) show that the left-hand and right-hand side of (5.23) are equal, the proof is complete. \square

We are now in a position to prove the main result of this section, which shows that the number of terminal compositions of n is equal to the number of ascending compositions of n minus the number of ascending compositions $n-2$. Lemma 5.9 provides the crucial inductive step in this proof; by noting that the trailing terms we require to complete both recurrences are equal, we can add and subtract the appropriate value without affecting the overall value of the sum. The theorem is then stated and proved as follows.

Theorem 5.11. *If $n \geq 3$, then $T_{\mathcal{A}}(n, m) = A(n, m) - A(n-2, m)$ for all $1 \leq m \leq \lfloor n/2 \rfloor$.*

Proof. Proceed by strong induction on n .

Base Case: $n = 3$. As $1 \leq m \leq \lfloor n/2 \rfloor$ and $n = 3$, we know that $m = 1$. Computing $T_{\mathcal{A}}(3, 1)$, we get $1 + T_{\mathcal{A}}(2, 1) = 2$. We also find $A(3, 1) = 3$ and $A(1, 1) = 1$, and so the base case of the induction holds.

Inductive Case. Suppose $T_{\mathcal{A}}(n', m) = A(n', m) - A(n'-2, m)$ when $1 \leq m \leq \lfloor n'/2 \rfloor$, for all $3 < n' < n$, and some integer n . Then, as $x \leq \lfloor (n-x)/2 \rfloor \iff x \leq \lfloor n/3 \rfloor$, by Lemma 5.8, we can apply this inductive hypothesis to each term $T_{\mathcal{A}}(n-x, x)$ in (5.21), giving us

$$\begin{aligned}
 T_{\mathcal{A}}(n, m) &= 1 + \sum_{x=m}^{\lfloor n/3 \rfloor} (A(n-x, x) - A(n-2-x, x)) \\
 &= 1 + \sum_{x=m}^{\lfloor n/3 \rfloor} A(n-x, x) - \sum_{x=m}^{\lfloor n/3 \rfloor} A(n-2-x, x).
 \end{aligned} \tag{5.26}$$

§ 5.4. Accelerated Algorithms

By Lemma 5.9 we know that

$$\sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor n/2 \rfloor} A(n-x, x) - \sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor (n-2)/2 \rfloor} A(n-2-x, x) - 1 = 0,$$

and so we can add the left-hand side of this equation to the right-hand side of (5.26), to get

$$\begin{aligned} T_{\mathcal{A}}(n, m) &= 1 + \sum_{x=m}^{\lfloor n/3 \rfloor} A(n-x, x) - \sum_{x=m}^{\lfloor n/3 \rfloor} A(n-2-x, x) \\ &\quad + \sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor n/2 \rfloor} A(n-x, x) - \sum_{x=\lfloor n/3 \rfloor + 1}^{\lfloor (n-2)/2 \rfloor} A(n-2-x, x) - 1. \end{aligned}$$

Then, gathering the terms $A(n-x, x)$ and $A(n-2-x, x)$ into the appropriate summations we get

$$T_{\mathcal{A}}(n, m) = 1 + \sum_{x=m}^{\lfloor n/2 \rfloor} A(n-x, x) - 1 - \sum_{x=m}^{\lfloor (n-2)/2 \rfloor} A(n-2-x, x),$$

which by (5.21) gives us $T_{\mathcal{A}}(n, m) = A(n, m) - A(n-2, m)$, as required. \square

For the purposes of our analysis it is useful to know the total number of terminal and nonterminal compositions of n , and it is worthwhile formalising the results here for reference. Therefore, letting $T_{\mathcal{A}}(n) = T_{\mathcal{A}}(n, 1)$ and $N_{\mathcal{A}}(n) = N_{\mathcal{A}}(n, 1)$, we get the following corollaries defined in terms of the partition function $p(n)$.

Corollary 5.5. *For all positive integers n , $T_{\mathcal{A}}(n) = p(n) - p(n-2)$.*

Proof. As $T_{\mathcal{A}}(n) = T_{\mathcal{A}}(n, 1)$ and $A(n, 1) = p(n)$, proof is immediate by Theorem 5.11 for all $n \geq 3$. Since $p(n) = 0$ for all $n < 0$ and $p(0) = 1$, we can readily verify that $T_{\mathcal{A}}(2) = T_{\mathcal{A}}(1) = 1$, as required. \square

Corollary 5.6. *If n is a positive integer then $N_{\mathcal{A}}(n) = p(n-2)$.*

§ 5.4. Accelerated Algorithms

Proof. An ascending composition is either terminal or nonterminal. As the total number of ascending compositions of n is given by $p(n)$, we get $N_{\mathcal{A}}(n) = p(n) - (p(n) - p(n - 2)) = p(n - 2)$. as required. \square

Corollaries 5.5 and 5.6 prove a nontrivial structural property of the set of all ascending compositions, and can be phrased in more conventional partition theoretic language. Consider an arbitrary partition of n , and let y be the largest part in this partition. We then let x be the second largest part ($x \leq y$). Corollary 5.6 then shows that the number of partitions of n where $2x > y$ is equal to the number of partitions of $n - 2$. This result is known, and has been reported by Adams-Watters [Slo05, Seq.A027336]. The preceding treatment, however, would appear to be the first published proof of the identity.

Algorithm

In Section 4.4 we developed and analysed an efficient algorithm to generate interpart restricted compositions of n . The performance of this algorithm relative to the direct successor algorithm of Section 4.1 is dependent on the frequency of terminal compositions within the class of compositions in question: the smaller the number of terminal compositions, the more efficient the algorithm. In the previous part of this subsection we proved that the number of terminal compositions in the set of ascending compositions of n is equal to $p(n) - p(n - 2)$. From the asymptotics of $p(n)$ we know that $p(n - 2)/p(n) \approx 1/e^{2\pi/\sqrt{6n}}$ [Knu04c, p.11], and we can therefore see that the proportion of compositions in the set $\mathcal{A}(n)$ that are terminal will approach zero for large values of n . Thus, by using the accelerated algorithm of Section 4.4, we should see a substantial gain in efficiency over the rule-based ascending composition generation algorithm of the previous section.

The accelerated ascending composition generation algorithm, ACCELASC (Algorithm 5.5), is obtained by modifying RULEASC (Algorithm 5.3) to utilise the block structure we have been discussing. (The algorithm is derived from our accelerated interpart restricted composition generator, ACCELGEN $_{\sigma}$ (Algorithm 4.5) by replacing $\sigma(x)$ with x .) At the head of the main loop x

§ 5.4. Accelerated Algorithms

Algorithm 5.5 ACCELASC(n)

Require: $n \geq 1$

```
1:  $k \leftarrow 2$ 
2:  $a_1 \leftarrow 0$ 
3:  $y \leftarrow n - 1$ 
4: while  $k \neq 1$  do
5:    $k \leftarrow k - 1$ 
6:    $x \leftarrow a_k + 1$ 
7:   while  $2x \leq y$  do
8:      $a_k \leftarrow x$ 
9:      $y \leftarrow y - x$ 
10:     $k \leftarrow k + 1$ 
11:  end while
12:   $\ell \leftarrow k + 1$ 
13:  while  $x \leq y$  do
14:     $a_k \leftarrow x$ 
15:     $a_\ell \leftarrow y$ 
16:    visit  $a_1 \dots a_\ell$ 
17:     $x \leftarrow x + 1$ 
18:     $y \leftarrow y - 1$ 
19:  end while
20:   $y \leftarrow y + x - 1$ 
21:   $a_k \leftarrow y + 1$ 
22:  visit  $a_1 \dots a_k$ 
23: end while
```

and y contain the values of the largest and second-largest parts in the previously visited partition, as before. We then write a number of copies of x into the array, if required, also as before. The differences between the algorithms begin here. If we examine the loop condition on line 7, we can see that the termination condition is now $2x \leq y$ instead of $x \leq y$. Therefore, we insert one less copy of x into the array than in the corresponding section of Algorithm 5.3. After this loop has finished iterating, we set $\ell \leftarrow k + 1$, and test the loop condition on line 13.

Within the loop of lines 13–19 we visit a set of contiguous nonterminal compositions. We can see that the compositions visited on line 16 must be nonterminal; this is because, upon reaching line 7 the condition $2x > y$ must

§ 5.4. Accelerated Algorithms

hold. Then, as we assign $a_k \leftarrow x$ and $a_{k+1} \leftarrow y$ on lines 14 and 15, the composition $a_1 \dots a_\ell$ visited on line 16 is nonterminal, by definition. Using the contiguity results proved in Section 4.4.1, we know that the lexicographic successor of any nonterminal composition $a_1 \dots a_k$ is either $a_1 \dots a_{k-2} \langle a_{k-1} + 1 \rangle \langle a_k - 1 \rangle$ or $a_1 \dots a_{k-2} \langle a_{k-1} + a_k \rangle$. The former case, where we simply increment the second last part and decrement the last, arises when we can perform this operation without violating the ascending order property. The latter case, then, arises when we cannot increment and decrement the second-last and last parts, respectively, and we simply add the two values together. It is not difficult to see that the latter half of the main loop of Algorithm 5.5 implements this sequence of transitions correctly; and it is quite clear that transitions made within the loop of lines 13–18 are efficient.

Another means of reducing the total number of read operations is to maintain the value of y between iterations. Recall that in Algorithm 5.3 we set $y \leftarrow a_k - 1$ at the head of the loop. At the tail of the previous loop iteration, however, we set $a_k \leftarrow x + y$, and so instead of reading this value back from the array, we can maintain it between loop iterations. Then, by setting $y \leftarrow n - 1$ before iteration begins, we can remove an unnecessary read operation from the loop. This improvement is an incremental one, and does not make a significant difference to the overall running time of the algorithm. It does, however, increase the degree to which we do our housekeeping by using local variables alone, and there seems little reason to exclude it.

Analysis

Once again, the variable k is the key to the analysis of our algorithm because it is used to control termination of the algorithm and is modified via increment and decrement operations only. We can then infer the frequency of certain key instructions, and use this information to determine the total number of times the instructions that correspond to read and write operations occur.

The first step in this process is to determine the number of compositions that are visited on line 16. From there we can infer that the rest of the

§ 5.4. Accelerated Algorithms

compositions must be visited on line 22; we then have sufficient information to determine the total number of iterations of each of the three loops. We formalise these three steps in the following lemmas.

Lemma 5.10. *The number of times line 16 is executed during the execution of Algorithm 5.5 is given by $t_{16}(n) = p(n - 2)$.*

Proof. Compositions visited on line 16 must be nonterminal because upon reaching line 12, the condition $2x > y$ must hold. As x and y are the second-last and last parts, respectively, of the composition visited on line 16, then this composition must be nonterminal by definition. Subsequent operations on x and y within this loop do not alter the property that $2x > y$, and so all compositions visited on line 16 must be nonterminal.

Furthermore, we also know that all compositions visited on line 22 must be terminal. To demonstrate this fact, we note that if $a_1 \dots a_k$ is the last composition visited before we arrive at line 20, the composition visited on line 22 must be $a_1 \dots a_{k-2} \langle a_{k-1} + a_k \rangle$. Therefore, to demonstrate that this composition is terminal, we must show that $2a_{k-2} \leq a_{k-1} + a_k$. We know that $a_{k-2} \leq a_{k-1} \leq a_k$. It follows that $2a_{k-2} \leq 2a_{k-1}$, and also that $2a_{k-1} \leq a_{k-1} + a_k$. Combining these two inequalities, we see that $2a_{k-2} \leq 2a_{k-1} \leq a_{k-1} + a_k$, and so $2a_{k-2} \leq a_{k-1} + a_k$. Thus all compositions visited on line 22 must be terminal.

Then, as Algorithm 5.5 correctly visits all $p(n)$ ascending compositions of n (see Section 4.4), as all compositions visited on line 22 are terminal and as all compositions visited on line 16 are nonterminal, we know that all nonterminal compositions of n must be visited on line 16. By Corollary 5.6 there are $p(n - 2)$ nonterminal compositions of n , and hence $t_{16} = p(n - 2)$, as required. \square

Lemma 5.11. *The number of times line 5 is executed during the execution of Algorithm 5.5 is given by $t_5(n) = p(n) - p(n - 2)$.*

Proof. By Lemma 5.10 we know that the visit statement on line 16 is executed $p(n - 2)$ times. As Algorithm 5.5 correctly visits all $p(n)$ ascending compositions of n , then the remaining $p(n) - p(n - 2)$ compositions must

§ 5.4. Accelerated Algorithms

be visited on line 22. Clearly then, line 22 (and hence line 5) is executed $p(n) - p(n - 2)$ times. Therefore, $t_5 = p(n) - p(n - 2)$, as required. \square

Lemma 5.12. *The number of times line 10 is executed during the execution of Algorithm 5.5 is given by $t_{10}(n) = p(n) - p(n - 2) - 1$.*

Proof. The variable k is assigned the value 2 upon initialisation, and the algorithm terminates when $k = 1$. As the variable is only updated via increment (line 10) and decrement (line 5) operations, we know that there must be one more decrement operation than increments. By Lemma 5.11 we know that there are $p(n) - p(n - 2)$ decrements, and so there must be $p(n) - p(n - 2) - 1$ increments on the variable. Therefore, $t_{10} = p(n) - p(n - 2) - 1$. \square

Having determined the frequency counts of the key statements in Algorithm 5.5 in Lemmas 5.10, 5.11 and 5.12, we can now count the total number of read and write operations incurred by the invocation $\text{ACCELASC}(n)$.

Theorem 5.12. *Algorithm 5.5 requires $R_{A5.5}(n) = p(n) - p(n - 2)$ read operations to generate the set $\mathcal{A}(n)$.*

Proof. Only one read operation occurs Algorithm 5.5, and this is done on line 6. By Lemma 5.11 we know that line 5 is executed $p(n) - p(n - 2)$ times, and it immediately follows that line 6 is executed the same number of times. Therefore, $R_{A5.5}(n) = p(n) - p(n - 2)$, as required. \square

Theorem 5.13. *Algorithm 5.5 requires $W_{A5.5}(n) = 2p(n) - 1$ write operations to generate the set $\mathcal{A}(n)$, excluding initialisation.*

Proof. Write operations are performed on lines 8, 14, 15 and 21. Lemma 5.11 shows that line 21 is executed $p(n) - p(n - 2)$ times. From Lemma 5.12 we know that line 8 is executed $p(n) - p(n - 2) - 1$ times. Then, by Lemma 5.10 we know that lines 14 and 15 are executed $p(n - 2)$ times each. Summing these contributions we get $W_{A5.5}(n) = p(n) - p(n - 2) + p(n) - p(n - 2) - 1 + 2p(n - 2) = 2p(n) - 1$, as required. \square

Theorems 5.12 and 5.13 derive the precise number of read and write operations required to generate all $p(n)$ partitions of n using Algorithm 5.5. This

§ 5.4. Accelerated Algorithms

algorithm is a considerable improvement over our basic implementation of the succession rule, Algorithm 5.3, in two ways. Both of these improvements are a consequence of the observations derived from the study of terminal and nonterminal ascending compositions. Firstly, by keeping $p(n - 2)$ of the visit operations within the loop of lines 13–19, we significantly reduce the average cost of a write operation. Thus, although we do not appreciably reduce the total number of write operations involved, we ensure that $2p(n - 2)$ of those writes are executed at the cost of an increment and decrement on a local variable and the cost of a \leq comparison of two local variables — in short, *very* cheaply.

The second consequence of utilising the theory of terminal and nonterminal compositions is that we dramatically reduce the total number of read operations involved. Recall that RULEASC required $2p(n)$ read operations to generate all ascending compositions of n ; and Theorem 5.12 shows that ACCELASC requires only $p(n) - p(n - 2)$ read operations. We also reduced the number of read operations by a factor of 2 by maintaining the value of y between iterations of the main while loop, but this trick could equally be applied to RULEASC, and is only a minor improvement at any rate. The real gain here is obtained from exploiting the block-based nature of the set of ascending compositions, as we do not need to perform any read operations once we have begun iterating through the nonterminal compositions within a block.

In summary, ACCELASC should be an efficient generation algorithm, taking into consideration its quantitative behaviour, which we have studied in detail in this subsection. This completes our development of ascending composition generation algorithms. We have developed a basic succession rule, and by studying some auxiliary theory, developed a coherent means of making the resulting algorithm more efficient. This increase in efficiency is bought at the cost of only a modest increase in complexity of algorithmic expression. We now return to descending compositions and study a generation algorithm commensurable with ACCELASC. This is the most efficient known descending composition generation algorithm, and is due to Zoghbi & Stojmenović [ZS98]. We analyse this algorithm in detail, and compare it with

§ 5.4. Accelerated Algorithms

ACCELASC in Section 5.4.3.

5.4.2 Descending Compositions

In Section 5.3.2 we derived a direct implementation of the succession rule for descending compositions. We then analysed the cost of using this direct implementation to generate all descending compositions of n , and found that it implied an average of $O(\sqrt{n})$ read and write operations per partition. There are, however, several efficient algorithms to generate descending compositions, and in this section we shall study the most efficient example.

There is one basic problem with the direct implementation of the succession rule for descending compositions (RULEDESC): most of the read and write operations it makes are redundant. To begin with, the read operations incurred by RULEDESC in scanning the current composition to find the rightmost non-1 value are unnecessary. As McKay [McK70] noted, we can easily keep track of the index of the largest non-1 value between iterations, and thereby eliminate the right-to-left scan altogether. The means by which we can avoid the majority of the write operations is a little more subtle, and was first noted by Zoghbi & Stojmenović [ZS98]. For instance, consider the transition

$$3321111 \rightarrow 33111111. \quad (5.27)$$

RULEDESC implements the transition from 3321111 to 33111111 by finding the prefix 33 and writing six copies of 1 after it, oblivious to the fact that 4 of the array indices *already* contain 1. Thus, a more reasonable approach is to make a special case in the succession rule so that if $d_q = 2$, we simply set $d_q \leftarrow 1$ and append 1 to the end of the composition. This observation proves to be sufficient to remove the worst excesses of RULEDESC, as 1s are by far the most numerous part in the partitions of n .

Zoghbi & Stojmenović's algorithm implements both of these ideas, and makes one further innovation to reduce the number of write operations required. By initialising the array to hold n copies of 1, we know that any index $> k$ must contain the value 1, and so we can save another write operation in the special case of $d_q = 2$ outlined above. Thus, Zoghbi & Stojmenović's

§ 5.4. Accelerated Algorithms

algorithm is the most efficient example, and consequently it is the algorithm that we shall use for our comparative analysis. Knuth developed a similar algorithm [Knu04c, p.2]: he also noted the necessity of keeping track of the value of q between iterations, and also implemented the special case for d_q outlined above. Knuth's algorithm, however, does not contain the further improvement included by Zoghbi & Stojmenović (i.e. initialising the array to $1 \dots 1$ and avoiding the second write operation in the $d_q = 2$ special case), and therefore requires strictly more write operations than Zoghbi & Stojmenović's. Zoghbi & Stojmenović's algorithm also consistently outperforms Knuth's algorithm in empirical tests.

Zoghbi & Stojmenović's algorithm is presented in Algorithm 5.6, which we shall also refer to as ACCELDESC. Each iteration of the main loop implements a single transition, and two cases are identified for performing the transition. In the conditional block of lines 8–10 we implement the special case for $d_q = 2$: we can see that the length of the composition is incremented, d_q is assigned to 1 and the value of q is updated to point to the new rightmost non-1 part. The general case is dealt with in the block of lines 11–29; the approach is much the same as that of RULEDESC, except in this case we have the additional complexity of maintaining the value of q between iterations.

There is, unfortunately, an example of our theoretical model conflicting with computational reality in this algorithm. In our theoretical model we count only the read and write operations contained within the algorithm, under the assumption that the cost of other housekeeping operations will be constant [Kem98, §1]. If we are to take this idea to its logical conclusion, we should do our utmost to minimise the number of read and write operations in the algorithm, and it is here that we see the conflict between theory and practice. If we examine lines 7 and 12 we can see that d_q is read twice on the occasions that $d_q \neq 2$. Thus, if we were to minimise the number of read operations, we should assign the value d_q to some local variable immediately after entering the main loop. This approach, however, proves to be less efficient in practice, because the number of occasions on which $d_q = 2$ is much greater than those where it is not (as we shall see in the analysis), and the extra assignment incurred here represents a significant time expenditure.

§ 5.4. Accelerated Algorithms

Algorithm 5.6 ACCELDESC(n) [ZS98]

Require: $n \geq 1$

```
1:  $k \leftarrow 1$ 
2:  $q \leftarrow 1$ 
3:  $d_2 \dots d_n \leftarrow 1 \dots 1$ 
4:  $d_1 \leftarrow n$ 
5: visit  $d_1$ 
6: while  $q \neq 0$  do
7:   if  $d_q = 2$  then
8:      $k \leftarrow k + 1$ 
9:      $d_q \leftarrow 1$ 
10:     $q \leftarrow q - 1$ 
11:   else
12:      $m \leftarrow d_q - 1$ 
13:      $n' \leftarrow k - q + 1$ 
14:      $d_q \leftarrow m$ 
15:     while  $n' \geq m$  do
16:        $q \leftarrow q + 1$ 
17:        $d_q \leftarrow m$ 
18:        $n' \leftarrow n' - m$ 
19:     end while
20:     if  $n' = 0$  then
21:        $k = q$ 
22:     else
23:        $k \leftarrow q + 1$ 
24:       if  $n' > 1$  then
25:          $q \leftarrow q + 1$ 
26:          $d_q \leftarrow n'$ 
27:       end if
28:     end if
29:   end if
30:   visit  $d_1 \dots d_k$ 
31: end while
```

Thus, we shall not apply this technique to minimise the number of read operations.

§ 5.4. Accelerated Algorithms

Analysis

ACCELDASC is a little more difficult to analyse than the algorithms we have considered thus far, but, once again, the key to the analysis is an indexing variable. In this instance the variable we are interested in is q . It is updated only using increment and decrement operations, and we can use this information to determine the total number of read and write operations incurred by ACCELDASC(n).

Lemma 5.13. *The number of times line 10 is executed during the execution of Algorithm 5.6 is given by $t_{10}(n) = p(n - 2)$.*

Proof. The variable q points to the smallest non-1 value in $d_1 \dots d_k$, and we have a complete descending composition in the array each time we reach line 7. Therefore, line 10 will be executed once for every descending composition of n which contains at least one 2; and it is well known that this is $p(n - 2)$. Therefore, $t_{10}(n) = p(n - 2)$, as required. \square

Lemma 5.14. *The number of times line 16 is executed during the execution of Algorithm 5.6 is given by $t_{16}(n) + t_{25}(n) = p(n - 2) - 1$.*

Proof. The variable q controls the termination of the algorithm. It is initialised to 1 on line 2, and the algorithm terminates when $q = 0$. We modify q via increment operations on lines 16 and 25, and decrement operations on line 10 only. Therefore, there must be one more decrement operation than increments on q . By Lemma 5.13 there are $p(n - 2)$ decrements performed on q , and there must therefore be $p(n - 2) - 1$ increments. Therefore, $t_{16}(n) + t_{25}(n) = p(n - 2) - 1$, as required. \square

Lemma 5.13 and Lemma 5.14 provide us with the information we require to complete the analysis of Algorithm 5.6. The following theorems derive the precise number of read and write operations that occur during the invocation ACCELDASC(n).

Theorem 5.14. *Algorithm 5.6 requires $R_{A5.6}(n) = 2p(n) - p(n - 2) - 2$ read operations to generate the set $\mathcal{D}(n)$.*

§ 5.4. Accelerated Algorithms

Proof. Read operations are performed on lines 7 and 12 of Algorithm 5.6. Clearly, as all but the composition $\langle n \rangle$ are visited on line 30, line 7 is executed $p(n) - 1$ times. Then, as a consequence of Lemma 5.13, we know that line 12 is executed $p(n) - p(n - 2) - 1$ times. Therefore, the total number of read operations is given by $R_{A5.6}(n) = 2p(n) - p(n - 2) - 2$, as required. \square

Theorem 5.15. *Algorithm 5.6 requires $W_{A5.6}(n) = p(n) + p(n - 2) - 2$ write operations to generate the set $\mathcal{D}(n)$, excluding initialisation.*

Proof. After initialisation, write operations are performed on lines 9, 14, 17 and 26 of Algorithm 5.6. Line 9 contributes $p(n - 2)$ writes by Lemma 5.13; and similarly, line 14 is executed $p(n) - p(n - 2) - 1$ times. By Lemma 5.14 we know that the total number of write operations incurred by lines 17 and 26 is $p(n - 2) - 1$. Therefore, summing these contributions we get $W_{A5.6}(n) = p(n) + p(n - 2) - 2$, as required. \square

Theorems 5.14 and 5.15 show that Zoghbi & Stojmenović's algorithm is a vast improvement on RULEDESC. Recall that RULEDESC(n) requires approximately $\sum_{x=1}^n p(x)$ read and $\sum_{x=1}^n p(x)$ write operations; and we have seen that ACCELDESC(n) requires only $2p(n) - p(n - 2)$ read and $p(n) + p(n - 2)$ write operations.

Zoghbi & Stojmenović [ZS98] also provided an analysis of their algorithm (ACCELDESC), and proved that it generates partitions in constant amortised time. We shall briefly summarise this analysis to provide some perspective on the approach we have taken. Zoghbi & Stojmenović begin their analysis by demonstrating that $D(n, m) \geq n^2/12$ for all $m > 2$, where $D(n, m)$ enumerates the descending compositions of n in which the initial part is *no more* than m . They use this result to reason that, for each $d_q > 2$ encountered, the total number of iterations of the internal **while** loop is $< 2c$, for some constant c . Thus, since the number of iterations of the internal loop is constant whenever $d_q \geq 3$ (the case for $d_q = 2$ obviously requires constant time), the algorithm generates descending compositions in constant amortised time.

The preceding paragraph is not a rigorous argument proving that ACCELDESC is constant amortised time. It is intended only to illustrate the difference in the approach that we have taken in this section to Zoghbi &

§ 5.4. Accelerated Algorithms

Stojmenović’s analysis, and perhaps highlight some of the advantages of using Kemp’s abstract model of counting read and write operations [Kem98]. By using Kemp’s model we were able to ignore irrelevant details regarding the algorithm’s implementation, and concentrate instead on the algorithms *effect*: reading and writing parts in compositions. We shall empirically evaluate the validity of this theoretical model in the next section, as part of our comparison of Zoghbi & Stojmenović’s algorithm with our accelerated ascending composition generator.

5.4.3 Comparison

This, and the previous two sections have followed the same structure. First we present an ascending composition generator based on the general methods developed in Chapter 4 and analyse this algorithm; then we study a descending composition generator from the literature and perform a similar analysis; and finally we compare the two algorithms to determine which is more efficient. The algorithms we studied in Section 5.2 and Section 5.3 may be (correctly) criticised as being suboptimal. For example, our algorithm RECASC may be more efficient than its descending composition counterpart RECDASC, but both algorithms will certainly be less efficient than any reasonable iterative technique. In this section, however, we have studied the most efficient known examples of ascending and descending composition generators: our ACCELASC algorithm and Zoghbi & Stojmenović’s algorithm [ZS98], ACCELDESC.

We have analysed these algorithms in terms of the total number of read and write operations required to generate all ascending and descending compositions of n , and deliberately ignored all other aspects of the algorithms, following the approach advocated by Kemp [Kem98]. Using this theoretical device we abstract away the implementation details of a particular algorithm, in a similar spirit to counting the key-comparisons in sorting algorithms. One advantage of this approach is that by comparing the read and write operation counts for an algorithm, we evaluate the complexity of *any* algorithm utilising the same structural properties. In this section we have determined

§ 5.4. Accelerated Algorithms

the read and write operation counts for algorithms that generate ascending and descending compositions, both of which use some underlying structural properties of the compositions in question to make certain transitions more efficient. Thus, a comparison of these operation counts should be a comparison of the inherent *complexity* of generation using these structural properties. It may be possible to utilise other structural properties of both objects to gain more efficient generation algorithms. There are no examples of such algorithms in the literature, however, and so the comparison in this subsection is a comparison of the state-of-the-art in ascending and descending composition generation.

Considering ACCELASC (Algorithm 5.5) first, we derived the following numbers of read and write operations required to generate all ascending compositions of n , ignoring inconsequential trailing terms.

$$R_{A5.5}(n) \approx p(n) - p(n - 2) \quad \text{and} \quad W_{A5.5}(n) \approx 2p(n) \quad (5.28)$$

We can see that the total number of write operations is $2p(n)$; i.e., the total number of write operations is twice the total number of partitions generated. On the other hand, the total number of read operations required is only $p(n) - p(n - 2)$, which, as we shall see presently, is asymptotically negligible in comparison to $p(n)$. The number of read operations is small because we only require one read operation per iteration of the outer loop. Once we have stored a_{k-1} in a local variable, we can then extend the composition as necessary and visit all of the following nonterminal compositions without needing to perform a read operation. Thus, it is the write operations that dominate the cost of generation with this algorithm and, as we noted earlier, the average cost of a write operation in this algorithm is quite small.

Moving on to the descending composition generator, ACCELDESC (Algorithm 5.6), the following read and write totals were derived (we ignore the insignificant trailing terms in both cases).

$$R_{A5.6}(n) \approx 2p(n) - p(n - 2) \quad \text{and} \quad W_{A5.6}(n) \approx p(n) + p(n - 2) \quad (5.29)$$

§ 5.4. Accelerated Algorithms

The total number of write operations required by this algorithm to generate all partitions of n is $p(n) + p(n - 2)$. Although this value is strictly less than the write total for ACCELASC, the difference is not asymptotically significant as $p(n - 2)/p(n)$ tends towards 1 as n becomes large. Therefore, we should not expect any appreciable difference between the performances of the two algorithms in terms of the number of write operations involved. There is, however, an asymptotically significant difference in the number of read operations performed by the algorithms.

The total number of read operations required by ACCELDESC is given by $2p(n) - p(n - 2)$. This expression is complicated by an algorithmic consideration, where it proved to be more efficient to perform $p(n) - p(n - 2)$ extra read operations than to save the relevant value in a local variable. Essentially, ACCELDESC needs to perform one read operation for every iteration of the external loop, to determine the value of d_q . If $d_q = 2$ we execute the special case and quickly generate the next descending composition; otherwise, we apply the general case. We cannot keep the value of d_q locally because the value of q changes constantly, and so we do not spend significant periods of time operating on the same array indices, as we do in ACCELASC. Thus, we must read the value of d_q for every transition, and we can therefore simplify by saying that ACCELDESC(n) requires $p(n)$ read operations.

In the interest of the fairest possible comparison between ascending and descending compositions generation algorithms, let us therefore simplify, and assume that any descending composition generation algorithm utilising the same properties as ACCELDESC requires $p(n)$ read operations. We know from (5.28) that our ascending composition generation algorithm required only $p(n) - p(n - 2)$ reads. We can therefore expect that an ascending composition generator will require $p(n - 2)$ *less* read operations than a descending composition generator similar to ACCELDESC. Other things being equal, we should expect a significant difference between the total time required to generate all partitions using an ascending composition generation algorithm and a commensurable descending composition generator.

We can gain a qualitative idea of the differences involved if we examine the average numbers of read and write operations using the asymptotic values

§ 5.4. Accelerated Algorithms

of $p(n)$. Again, to determine the average number of read and write operations required per partition generated we must divide the totals involved by $p(n)$. We stated earlier that the value of $p(n) - p(n - 2)$ is asymptotically negligible compared to $p(n)$; we can quantify this statement using the asymptotic formulas for $p(n)$. Knuth [Knu04c, p.11] provides an approximation of $p(n - 2)/p(n)$, which can be expressed as follows:

$$\frac{p(n - 2)}{p(n)} \approx \frac{1}{e^{2\pi/\sqrt{6n}}}. \quad (5.30)$$

Considering this approximation for a moment, we can see that it quickly approaches 1. For example, $p(8)/p(10) \approx 0.44$, $p(98)/p(100) \approx 0.77$ and $p(998)/p(1000) \approx 0.92$. Using this approximation, we obtain the following estimates for the average number of read and write operations required to generate each ascending and descending composition of n .

	Reads	Writes
Ascending	$1 - e^{-2\pi/\sqrt{6n}}$	2
Descending	1	$1 + e^{-2\pi/\sqrt{6n}}$

Suppose we wished to generate all partitions of 1000. Then, using the best known descending composition generation algorithm we would expect to make 1 read and 1.92 write operations per partition generated. On the other hand, if we used ACCELASC, we would expect to make only 0.08 read and 2 write operations per partition.

The qualitative behaviour of ACCELASC and ACCELDESC can be seen from their read and write tapes (Figure 5.7). Comparing the write tapes for the algorithms, we can see that the total number of write operations is roughly equal in both algorithms, although they follow an altogether different spatial pattern. The read tapes for the algorithms, however, demonstrate the essential difference between the algorithms: ACCELDESC makes one read operation for every partition generated, while the read operations for ACCELASC are sparsely distributed across the tape.

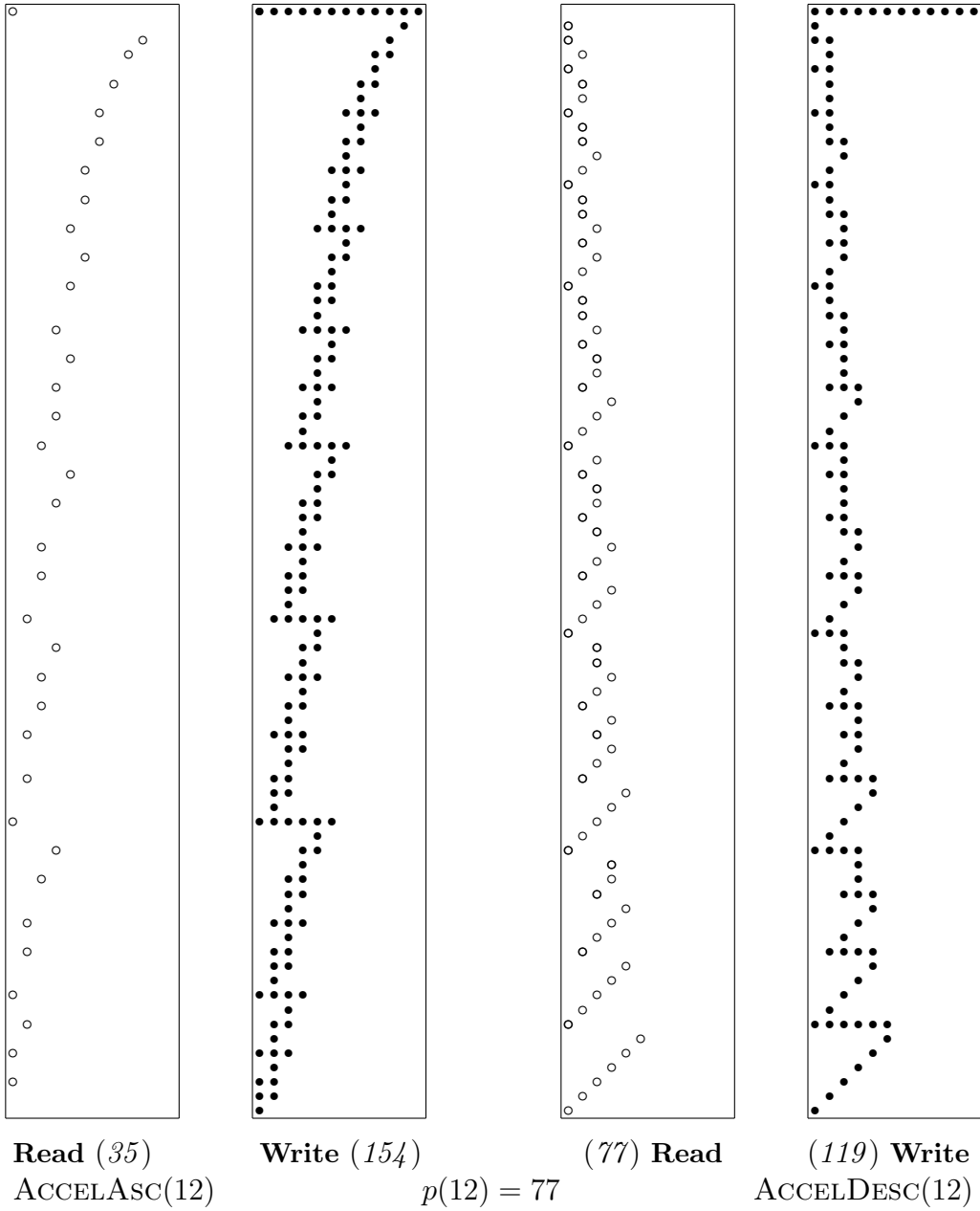


Figure 5.7: Read and write tapes for the accelerated algorithms to generate ascending and descending compositions. On the left we have the read and write tapes for the ascending composition generator, Algorithm 5.5; on the right, then, are the corresponding tapes for the descending composition generator, Algorithm 5.6. In both cases, the traces correspond to the read and write operations carried out in generating all partitions of 12.

Empirical Comparison

We have theoretically predicted that ACCELASC (Algorithm 5.5) should be significantly more efficient than its equivalent descending composition generator, ACCELDESC (Algorithm 5.6). It is not certain, however, that these theoretical predictions will translate into actual computational experience. Our theoretical model ignores all considerations except the number of read and write operations required to generate all partitions using a particular algorithm, and it may prove that this simplification is not justified by empirical evidence [Knu73]. Therefore, in this section we empirically measure the cost of generating all partitions using both algorithms, and compare the empirical observations with our theoretical predictions.

Up to this point we have made only qualitative predictions about the relative efficiencies of generating ascending and descending compositions: we have claimed that the ascending composition generator should be significantly more efficient than the descending composition generator. Our theory, however, is capable of making *quantitative* predictions in this regard, so we may also establish the validity of our read-write cost model, as well as assessing the accuracy of the qualitative predictions previously made.

Specifically, we have derived expressions to count the total number of read write operations required to generate all partitions of n using ACCELASC and ACCELDESC. If we assume that the cost of read and write operations are equal, we can then derive a prediction for the ratio of the total time elapsed using both algorithms. Therefore, let $E_{5.5}(n)$ be the expected total running time of ACCELASC(n), and similarly define $E_{5.6}(n)$ for ACCELDESC(n). We can then predict that the ratio of the running times should be equal to the ratio of their total read and write counts. Thus, using the values of (5.28) and (5.29), we get

$$\frac{E_{5.5}(n)}{E_{5.6}(n)} = \frac{3p(n) - p(n-2)}{3p(n)}. \quad (5.31)$$

Consequently, we expect that the total amount of time required to generate all ascending compositions of n should be a factor of $p(n-2)/3p(n)$ less than that required to generate all descending compositions of n . We shall put this hypothesis to test, and see if experimental evidence supports our theoretical

§ 5.4. Accelerated Algorithms

predictions.

In the experiments we measured the total elapsed time required to generate all partitions of n using ACCELASC and ACCELDESC. The methodology used to measure the elapsed time is the same as explained in Section 4.4.4 — we measure five runs of each algorithm, and report the minimum of these times. (Further measures required to address some of the more salient criticisms of the empirical evaluation of algorithms are discussed Section 4.4.4, and applied here.) We report the ratio of these times in Table 5.2, for both the C and Java implementations of the algorithms.

Table 5.2 supports our qualitative predictions well. The theoretical analysis of ascending and descending composition generation algorithms in this section suggests that the ascending composition generator should require significantly less time to generate all partitions of n than its descending composition counterpart; and the data of Table 5.2 supports this prediction. In the Java implementations, the ascending composition generator requires 15% less time to generate all partitions of 100 than the descending composition generation algorithm; in the C version, the difference is around 25%. These differences increase as the value of n increases: when $n = 135$, we see that ACCELASC requires 18% and 28% less time than ACCELDESC in the C and Java implementations, respectively.

We also made a quantitative prediction about the ratio of the time required to generate all partitions of n using ACCELASC and ACCELDESC. Using the theoretical analysis, where we counted the total number of read and write operations required by these algorithms, we can predict the expected ratio of the time required by both algorithms. This ratio is also reported in Table 5.2, and we can see that it is consistent with the measured ratios for the Java and C implementations of the algorithms. In the case of the Java implementation, the theoretically predicted ratios are too optimistic, suggesting that the model of counting only read and write operations is a little overly simplistic in this case. The correspondence between the measured and predicted ratios in the C implementation is much closer, as we can see from Table 5.2. In both cases there is a strong positive correlation — in excess of 0.9 — between the predicted and measured ratios, demonstrating that a

§ 5.4. Accelerated Algorithms

n	$p(n)$	Java	C	Theoretical
100	1.91×10^8	0.85	0.77	0.74
105	3.42×10^8	0.85	0.77	0.74
110	6.07×10^8	0.84	0.75	0.74
115	1.06×10^9	0.84	0.75	0.73
120	1.84×10^9	0.83	0.75	0.73
125	3.16×10^9	0.83	0.74	0.73
130	5.37×10^9	0.83	0.74	0.73
135	9.04×10^9	0.82	0.74	0.73
		$r_{\text{Java}} = 0.9891$	$r_{\text{C}} = 0.9321$	

Table 5.2: Empirical analysis of accelerated ascending and descending composition generation algorithms. The ratio of the time required to generate all partitions of n using ACCELASC and ACCELDESC is given: measured ratios for implementations in the Java and C languages as well as the theoretically predicted ratio are shown.

simple linear transformation can be applied to the predictions to obtain the measured values [Edw84, ch.3].

5.4.4 Other Algorithms

The algorithms we have studied in this chapter all use the sequence representation, but many published algorithms for generating all partitions utilise alternative representations — see Table 2.2 (p.35). Many of the non-sequence representation algorithms are constant amortised time; but not all constant amortised time algorithms are equally efficient. To provide a reference for the relative efficiency of the algorithms we have developed, we shall conclude this section with a brief empirical analysis. In this analysis we compare the most efficient example of algorithms to generate all partitions of n in the sequence, multiplicity and part-count representations (see Section 2.3.1) with our ascending composition generator, ACCELASC (Algorithm 5.5).

Fenner & Loizou’s algorithm [FL80, FL81], which generates descending compositions in lexicographic order, is our exemplar for algorithms that generate all partitions in the multiplicity representation. Fenner & Loizou im-

§ 5.4. Accelerated Algorithms

proved the efficiency of the basic multiplicity representation generation algorithm (see, for example, Reingold, Nievergelt & Deo [RND77, p.193]), by identifying a total of twelve individual cases in the lexicographic succession rule we studied in Section 5.3.2. The algorithm is therefore rather complex: when implemented using the standard Java conventions [JCC] it requires more than ninety lines of code. (When implemented in a similar manner, ACCELASC requires around thirty lines of code.)

The representative for the part-count representation is Klimko's algorithm [Kli73], as presented by Knuth [Knu04c, ex.5]. This algorithm generates descending compositions in lexicographic order, and is also rather complex to implement (more than seventy lines of code using the Java coding conventions [JCC]). We also present three algorithms to generate descending compositions in the sequence representation: Zoghbi & Stojmenović's algorithms to generate descending compositions in lexicographic (ZS2) and reverse lexicographic (ZS1) orders [ZS98], and Knuth's descending composition generation algorithm [Knu04c, p.2] (TAOCP).

In each case we generated all partitions of n using the particular algorithm and measured the elapsed time in using the methodology discussed in Section 4.4.4. It was necessary in the C versions of these algorithms to implement the array access instructions using pointers, because Klimko's and Fenner & Loizou's algorithms make reference to constant array indices (e.g. a_1). Since the C compiler automatically converts such constant array references into pointers, the algorithms are not fairly compared if we use direct implementations. By hand-coding each of the algorithms to use pointers, we can redress this imbalance, and the ratios agree reasonably well with the Java versions (for which no such problems arise).

In Table 5.3 we report the results of these experiments; as usual, we report the amount of time taken by our ascending composition generation algorithm divided by the time taken by the algorithm in question. We can see that, for example, ACCELASC requires only 60% of the time required by Fenner & Loizou's algorithm (FL) to generate all partitions of 130. Fenner & Loizou's algorithm is the most efficient known example of an algorithm to generate partitions in the multiplicity representation. In Zoghbi & Stojmen-

§ 5.5. Summary

	$n =$	77	90	95	109	115	130
	$p(n) =$	1.06×10^7	5.66×10^7	1.05×10^8	5.42×10^8	1.06×10^9	5.37×10^9
ZS1	J	0.88	0.85	0.85	0.84	0.83	0.83
	C	0.61	0.63	0.63	0.62	0.62	0.61
TAOCP	J	0.81	0.78	0.77	0.76	0.76	0.75
	C	0.61	0.61	0.61	0.60	0.59	0.58
ZS2	J	0.64	0.63	0.64	0.64	0.64	0.63
	C	0.47	0.48	0.49	0.48	0.48	0.48
Klimko	J	0.61	0.60	0.60	0.60	0.60	0.59
	C	0.44	0.46	0.47	0.47	0.47	0.47
FL	J	0.63	0.61	0.60	0.59	0.59	0.58
	C	0.56	0.57	0.59	0.58	0.58	0.58

Table 5.3: Empirical comparison of major partition generation algorithms in the various representations. Values reported are the amount of time taken by the accelerated ascending composition generator to generate a particular set of partitions divided by the time taken by the specified algorithm to generate the same set of partitions.

ović’s empirical evaluation [ZS98], it was by far the most efficient multiplicity representation algorithm tested. For each of the other algorithms in Table 5.3 — Zoghbi & Stojmenović’s algorithms [ZS98], Klimko’s algorithm [Kli73] and Knuth’s *The Art of Computer Programming* algorithm [Knu04c, p.2] — ACCELASC requires significantly less time to generate all partitions of n .

5.5 Summary

Having reached the end of this chapter on relative efficiencies of ascending and descending compositions as the encoding scheme for generating partitions, let us review the evidence we have accumulated, and attempt to draw some conclusions. In summary, for each of the usage scenarios we have considered — the recursive generation, direct rule-based generation, and efficiency oriented generation — there is a compelling efficiency advantage to encoding partitions as ascending, rather than descending, compositions.

§ 5.5. Summary

To recapitulate the main results of this chapter, we studied recursive generation of partitions in Section 5.2 and established that a simple recursive algorithm to generate ascending compositions requires approximately half the time required to generate all partitions using the most efficient known descending compositions generation algorithm. Then, in Section 5.3 we examined the succession rules for ascending and descending compositions, and demonstrated that there is a clear choice between the rules, if generating all partitions is our goal. The ascending composition succession rule implies a constant amount of work per partition generated, while the descending composition succession rule implies $O(\sqrt{n})$ time per partition. Finally, in Section 5.4, we compared the most efficient known examples of ascending and descending composition generation algorithms, and established that the ascending composition generation scheme is much more efficient.

Chapter 6

Conclusions and Future Work

In this chapter we conclude the dissertation by summarising the defence of our thesis in Section 6.1 and briefly examining some possibilities for future work in Section 6.2.

6.1 Thesis Defence

The thesis we set out to defend was the conjunction of four subtheses. We shall attend to each of these in turn and briefly summarise its defence.

Subthesis 1. *Important classes of restricted partition can be expressed concisely by a function $\sigma : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$.*

Defence. In Chapter 3 we developed the basic theory required for the inter-part restricted compositions framework. We demonstrated in Section 3.1.3 that, for example, the unrestricted partitions are represented by the function $\sigma(x) = x$; the partitions into distinct parts by $\sigma(x) = x + 1$; and the Rogers-Ramanujan partitions by $\sigma(x) = x + 2$. Each of these examples of restricted partition has been the subject of extensive study for many years, and many more classes may be defined within the framework. The representation format is clearly concise.

Subthesis 2. *It is possible to define an efficient algorithm to enumerate partitions with restrictions expressed by such a function.*

§ 6.1. Thesis Defence

Defence. In Section 3.2 we developed a recurrence equation to enumerate interpart restricted compositions. We proved the correctness of this recurrence in Theorem 3.4 and used it to define a dynamic programming enumeration algorithm. This enumeration algorithm requires $O(N^2)$ time and space to compute $C_\sigma(n, m)$ for all $1 \leq m \leq n \leq N$. There is no more efficient known method to enumerate restricted classes of partitions *in general* [SS96].

Subthesis 3. *It is possible to efficiently generate all restricted partitions where the corresponding restriction function is nondecreasing.*

Defence. In Chapter 4 we developed, and proved the correctness of, generation algorithms for interpart restricted compositions. Each of these algorithms requires that the restriction function be nondecreasing. We proved that these algorithms generate interpart restricted compositions in constant amortised time in Section 4.2.3, Section 4.3.3 and Section 4.4.3. Constant amortised time performance is considered to be the “ultimate goal in efficiency” [MM05] for combinatorial generation algorithms.

Subthesis 4. *It is possible to define algorithms to generate ascending compositions that are more efficient than the most efficient known commensurable algorithms to generate descending compositions.*

Defence. In Chapter 5 we compared concrete instantiations of our interpart restricted generation algorithms with commensurable descending composition generation algorithms from the literature. In Section 5.2.3 we compared our recursive ascending composition generation algorithm with Ruskey’s recursive descending composition generator. We proved that while the ascending composition generator required exactly $p(n)$ invocations to generate all partitions of n , Ruskey’s algorithm required $p(n) + p(n-1)$. We demonstrated that this is an asymptotically significant difference, since $p(n)/p(n-1)$ tends towards 1 as n becomes large. Therefore, it is possible to define a recursive ascending composition generator that is more efficient than the most efficient known commensurable descending composition generator.

In Section 5.3.3 we compared direct, unmeliorated implementations of the succession rules for ascending and descending compositions. We demonstrated that the ascending composition generation algorithm required $2p(n)$

§ 6.2. Future Work

read and $2p(n)$ write operations, and the descending composition generation method required $\sum_{x=1}^n p(x)$ read and $\sum_{x=1}^n p(x)$ write operations to generate all partitions of n . It follows that the average number of read and write operations required to generate an ascending composition is constant, whereas the average number of read and write operations required to generate a descending composition grows with \sqrt{n} , using direct implementations of the respective succession rules. Therefore, it is possible to define a direct implementation of the succession rule for ascending compositions that is more efficient than the most efficient known commensurable descending composition generator.

In Section 5.4.3 we compared the most efficient known algorithms to generate ascending and descending compositions. The ascending composition generator used was a concrete instantiation of our accelerated algorithm to generate interpart restricted compositions, and the descending composition generator was Zoghbi & Stojmenović's algorithm. We demonstrated, theoretically and empirically, that the ascending composition generator was significantly more efficient than Zoghbi & Stojmenović's descending composition generator. Therefore, it is possible to define an algorithm to generate ascending composition that is more efficient than the most efficient known descending composition generator.

6.2 Future Work

In this dissertation we have been chiefly concerned with justifying the usage of ascending compositions to encode partitions for systematic generation. The convention of explicitly defining a partition as a descending composition is a deeply established one, and it was necessary to thoroughly examine its fundamental implications. The decision to encode partitions as ascending compositions is not a frivolous one, and conventions should not be altered without good reason. For this reason, we have concentrated on material directly relevant to defending the usage of ascending compositions. Topics peripheral to this issue have been examined but not expanded upon, and many interesting possibilities for future work exist. In this section we present

§ 6.2. Future Work

a brief outline of some avenues for future research that seem promising.

A Gray Code

We have concentrated exclusively on the lexicographic ordering for our generation algorithms in this dissertation. As we saw in Section 2.1.2, there is one other ordering commonly used in combinatorial generation, known as the Gray code, or minimal change orders. Gray codes are not unique, and are not defined in a rigorous fashion across different objects: the term is loosely taken to mean an ordering in which successive objects differ by some small pre-specified amount.

11		
1	10	
1	3	7
1	4	6
2	9	
3	8	
4	7	

Savage has defined a Gray code over partitions [Sav89] where successive partitions differ by adding 1 to one part and subtracting 1 from another. Subsequent work extended Savage's Gray code over partitions to operate on partitions into distinct parts, and partitions where certain congruence conditions on the parts are met [RSW95]. We propose a different sort of Gray code here, where we limit not the amount of change to particular parts, but instead limit the number of parts which are changed. In the figure above we show the Rogers-Ramanujan partitions of 11 arranged in this new Gray code. The ordering obtained is easily found by modifying any of the generation algorithms. In the figure, we can see that no more than three parts are modified in any one transition, and this would appear to be a property of the Gray code in general. This property then ensures that the generation algorithms for interpart restricted compositions are *loopless* and so the delay between successive visits is fixed. The fact that a maximum of three parts change on successive iterations also allows consumer procedures to reduce the amount of work performed at each visit, thereby significantly improving the efficiency of the overall process of generation and consumption. Defining such algorithms, developing a succession rule for the Gray code and proving that it has the required properties for *all* instances of the restriction function is a challenging possibility for future work.

§ 6.2. Future Work

Optimum Generation of Partitions

We have discussed the relative efficiency of generating ascending and descending compositions at length. We have not considered, however, the concept of *optimum* algorithms: algorithms that make the fewest possible read and write operations to generate all ascending or descending compositions of n . Computing the optimum number of write operations is not a difficult concept: there is a perfect trajectory through the write-tape for ascending and descending compositions (this value does not appear to correspond to any known combinatorial quantity, however). A much more difficult problem is determining what the least possible number of *read* operations is. It seems likely that optimum write paths for generation algorithms may only be bought at the expense of extra read operations, and so there is an interesting concept of an optimum balance which may be found.

Generating Restricted Compositions

Numerous open problems exist with regard to generating classes of composition that do not correspond to partitions. For example, no efficient algorithm is known to generate compositions into odd parts [Gri00], compositions with no occurrence of a particular value [CH03b], $(1, k)$ -compositions [CH03a], compositions with parts drawn from some pre-specified set [HM04], compositions with a specified number of rises, levels and drops [HCG03] or Carlitz compositions [KP98]. Generating restricted compositions is far more challenging than generating partitions as the ordering we enforce to mimic the unsortedness requirement greatly facilitates the generation process. Thus, defining a constant amortised time algorithm to generate compositions into distinct parts would be a remarkable result.

Enumerating Interpart Restricted Compositions

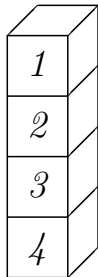
There are many aspects of Chapter 4 that can be extended. One possibility is a general asymptotic formula for interpart restricted compositions. Asymptotic formulas would appear to be known only for the instances $\sigma(x) = 1$, $\sigma(x) = 2$, $\sigma(x) = x$ and $\sigma(x) = x + 1$. Perhaps using this information an

§ 6.2. Future Work

asymptotic formula parameterised by the restriction function would be possible. A more realistic problem is to develop a generating function for the interpart restricted compositions, which may be derived from the recurrence relations provided in Section 3.2.

Another interesting problem that has arisen is to find a general relationship between the terminal interpart restricted compositions of n and the total number of interpart restricted compositions of n . We proved that the number of terminal ascending compositions of n is $p(n) - p(n - 2)$, and it is not difficult to derive the relationship for certain other simple instances of the framework. In general, however, there would not appear to be any obvious relationship between $T_\sigma(n)$ and $C_\sigma(n)$. It would be an interesting problem to prove (or disprove) that such a relationship exists.

Nonsquashing Interpart Restricted Compositions



A topical class of restricted partition from the literature are the *non-squashing partitions* of a given number [SS05]. Suppose that we have boxes labelled with positive integers. A box with label j weighs j grams and can support j grams in total. We wish to build stacks of boxes in such a way as to ensure that no box is squashed by the weight of the boxes above it. On the right we see an example of a squashed partition, since box 4 cannot support the 6 grams stacked upon it. The problem is then to count (or generate) the stacks of boxes with total weight n grams which are non-squashing. We can define the non-squashing partitions more precisely. Each stack of boxes is represented by a composition $n = a_1 + \dots + a_k$ where $a_1 + \dots + a_j \leq a_{j+1}$ for $1 \leq j < k$ [SS05]. We can modify Algorithm 4.1 to allow us generate partitions of this class by keeping track of the sum of the parts assigned so far and determining if the next assignment will squash any of these values. It is possible to define the algorithm such that the constant amortised time property is maintained. We can then also generate non-squashing partitions into distinct parts [RSC04], non-squashing Göllnitz-Gordon partitions, and so forth. Determining the combinatorial properties of

§ 6.2. Future Work

non-squashing interpart restricted compositions is an interesting possibility for future work.

Restricted σ -Chain Length

Gordon & Ono [GO97] studied the partitions of n where no parts are multiples of r . The number of such partitions is also equal to the number of partitions with no more than $r - 1$ copies of any part [Hon85, p.68], and it is this class of partition we are concerned with here. Given a composition $a = a_1 \dots a_k$ interpart restricted by σ , define a σ -chain in a as a sequence $a_l \dots a_t$ such that $\sigma(a_j) = a_{j+1}$ for $l \leq j < t$. For example, if we let $\sigma(x) = x + [x \text{ even}]$ the composition

$$2 + 3 + 3 + 3 + 3 + 4 + 6$$

of 24 contains one σ -chain of length 4 ($3 + 3 + 3 + 3$) and one σ -chain of length 2 ($2 + 3$). Alladi & Berkovich [AB05] studied Göllnitz-Gordon partitions with chains which they defined as “a maximal string of parts differing by exactly 2”, and the notion of σ -chains just defined is a generalisation of this. Other notions such as generalising p -regular partitions [All97, §7] also present an interesting possibility for future work.

Bibliography

- [AB89] George E. Andrews and Rodney J. Baxter. A motivated proof of the Rogers-Ramanujan identities. *The American Mathematical Monthly*, 96(5):401–409, May 1989.
- [AB05] Krishnaswami Alladi and Alexander Berkovich. Göllnitz-Gordon partitions with weights and parity conditions. In T. Aoki, S. Kanemitsu, M. Nakahara et al., editor, *Zeta functions topology and quantum physics*. Springer Verlag, 2005.
- [Act94] Alfred Arthur Actor. Infinite products, partition functions, and the Meinardus theorem. *Journal of Mathematical Physics*, 35(11):5749–5764, November 1994.
- [AE04] George E. Andrews and Kimmo Eriksson. *Integer Partitions*. Cambridge University Press, 2004.
- [AG95] Krishnaswami Alladi and Basil Gordon. Schur’s partition theorem, companions, refinements and generalizations. *Transactions of the American Mathematical Society*, 347(5):1591–1608, May 1995.
- [Akl81] Selim G. Akl. A comparison of combination generation methods. *ACM Transactions on Mathematical Software*, 7(1):42–45, March 1981.
- [Ald69] H. L. Alder. Partition identities — from Euler to the present. *The American Mathematical Monthly*, 76(7):733–746, August-September 1969.

BIBLIOGRAPHY

- [All97] Krishnaswami Alladi. Partition identities involving gaps and weights. *Transactions of the American Mathematical Society*, 349(12):5001–5019, December 1997.
- [All99] Krishnaswami Alladi. A variation on a theme of Sylvester — a smoother road to Göllnitz’s (big) theorem. *Discrete Mathematics*, 196(1–3):1–11, February 1999.
- [And69] George E. Andrews. A general theorem on partitions with difference conditions. *American Journal of Mathematics*, 91(1):18–24, January 1969.
- [And76] George E. Andrews. *The Theory of Partitions*. Encyclopedia of Mathematics and its Applications. Addison-Wesley, London, 1976.
- [And88] George E. Andrews. J.J. Sylvester, Johns Hopkins and partitions. In *A Century of Mathematics in America*, volume 1, pages 21–40. American Mathematical Society, Providence, R.I., 1988.
- [And05] George E. Andrews. Partitions. In *History of Combinatorics*, volume 1. To appear, 2005.
- [AO01] Scott Ahlgren and Ken Ono. Addition and counting: The arithmetic of partitions. *Notices of the AMS*, 48(9):978–984, October 2001.
- [AS81] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions with formulas, graphs and mathematical tables*. Basil Blackwell, Oxford, 1981.
- [AS93] Selim G. Akl and Ivan Stojmenović. Parallel algorithms for generating integer partitions and compositions. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 13:107–120, 1993.

BIBLIOGRAPHY

- [AS96] Selim G. Akl and Ivan Stojmenović. Generating t -ary trees in parallel. *Nordic Journal of Computing*, 3(1):63–71, Spring 1996.
- [BBGP04] Silvia Bacchelli, Elena Barcucci, Elisabetta Grazzini, and Elisa Pergola. Exhaustive generation of combinatorial objects by ECO. *Acta Informatica*, 40(8):585–602, July 2004.
- [BC05] Edward A. Bender and E. Rodney Canfield. Locally restricted compositions i. Restricted adjacent differences. *The Electronic Journal of Combinatorics*, 12(1), November 2005. R57.
- [Bec64] Edwin F. Beckenbach, editor. *Applied Combinatorial Mathematics*. Wiley, New York, 1964.
- [BER76] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, September 1976.
- [BH80] Terry Beyer and Sandra Mitchell Hedetniemi. Constant time generation of rooted trees. *SIAM Journal on Computing*, 9(4):706–712, November 1980.
- [Bha99] P. C. P. Bhatt. An interesting way to partition a number. *Information Processing Letters*, 71(3-4):141–148, August 1999.
- [BM67] P. Bratley and J. K. S. McKay. Algorithm 313: Multi-dimensional partition generator. *Communications of the ACM*, 10(10):666, October 1967.
- [BM98] Alexander Berkovitch and Barry M. McCoy. Rogers-Ramanujan identities: A century of progress from mathematics to physics. In *Documenta Mathematica Extra Volume ICM III*, pages 163–172, 1998.
- [BME97a] Mireille Bousquet-Mélou and Kimmo Eriksson. Lecture hall partitions. *The Ramanujan Journal*, 1(1):101–111, January 1997.

BIBLIOGRAPHY

- [BME97b] Mireille Bousquet-Mélou and Kimmo Eriksson. Lecture hall partitions 2. *The Ramanujan Journal*, 1(2):165–185, January 1997.
- [BMSW54] Robert L. Bivins, N. Metropolis, Paul R. Stein, and Mark B. Wells. Characters of the symmetric groups of degree 15 and 16. *Mathematical Tables and Other Aids to Computation*, 8(48):212–216, October 1954.
- [Bón02] Miklós Bóna. *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory*. World Scientific Publishing, 2002.
- [Bón04] Miklós Bóna. *Combinatorics of Permutations*. CRC Press, 2004.
- [Boy05] John M. Boyer. Simple constant amortized time generation of fixed length numeric partitions. *Journal of Algorithms*, 54(1):31–39, January 2005.
- [BR90] Bruce Bauslaugh and Frank Ruskey. Generating alternating permutations lexicographically. *BIT*, 30(1):17–26, 1990.
- [Bry73] Thomas Brylawski. The lattice of integer partitions. *Discrete Mathematics*, 6:201–219, 1973.
- [BS73] T. Beyer and D. F. Swinehart. Algorithm 448: Number of multiply restricted partitions. *Communications of the ACM*, 16(6):379, June 1973.
- [BS94] Mounir Belbaraka and Ivan Stojmenović. On generating B-trees with constant average delay and in lexicographic order. *Information Processing Letters*, 49(1):27–32, January 1994.
- [BS97] Tiffany M. Barnes and Carla D. Savage. Efficient generation of graphical partitions. *Discrete Applied Mathematics*, 78(1–3):17–26, October 1997.

BIBLIOGRAPHY

- [BS05] Anders Björner and Richard P. Stanley. A combinatorial miscellany. <http://www.math.kth.se/~bjorner/files/CUP.ps>, 2005. To appear in *L'Enseignement Mathématique*.
- [Cas04] Nello Castellini. *Four Color Conjecture*. Xlibris, 2004.
- [Cay76] Arthur Cayley. Theorem on partitions. *Messenger of Mathematics*, 5:188, 1876.
- [CH03a] Phyllis Chinn and Silvia Heubach. $(1, k)$ -compositions. *Congressus Numerantium*, 164:183–194, 2003.
- [CH03b] Phyllis Chinn and Silvia Heubach. Compositions of n with no occurrence of k . *Congressus Numerantium*, 164:33–51, 2003.
- [CH03c] Phyllis Chinn and Silvia Heubach. Integer sequences related to compositions without 2s. *Journal of Integer Sequences*, 6, 2003. Article 03.2.3.
- [Cha02] Charalambos A. Charalambides. *Enumerative Combinatorics*. CRC Press, 2002.
- [CK05] Thomas Colthurst and Michael Kleber. A Gray path on binary partitions. To Appear, 2005.
- [CL97] William Y. C. Chen and James D. Louck. Necklaces, MSS sequences, and DNA sequences. *Advances in Applied Mathematics*, 18(1):18–32, January 1997.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT Press, 1990.
- [Com55] Stig Comét. Notations for partitions. *Mathematical Tables and Other Aids to Computation*, 9(52):143–146, October 1955.
- [Com74] Louis Comtet. *Advanced combinatorics: the art of finite and infinite expansions*. Dordrecht Reidel, rev. and enlarged edition, 1974.

BIBLIOGRAPHY

- [CRS⁺00] Kevin Cattell, Frank Ruskey, Joe Sawada, Michaela Serra, and C. Robert Miers. Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over $\text{GF}(2)$. *Journal of Algorithms*, 37(2):267–282, November 2000.
- [CS04] Sylvie Corteel and Carla D. Savage. Partitions and compositions defined by inequalities. *The Ramanujan Journal*, 8(3):357–381, September 2004.
- [CT71] M. W. Coleman and M. S. Taylor. Algorithm 403: Circular integer partitioning. *Communications of the ACM*, 14(1):48, January 1971.
- [DD99] Thomas P. Dence and Joseph B. Dence. *Elements of the Theory of Numbers*. Elsevier, 1999.
- [Dés02] P. Désesquelles. Calculation of the number of partitions with constraints on the fragment size. *Physical Review C (Nuclear Physics)*, 65(3):034603, March 2002.
- [Dic52] Leonard E. Dickson. *History of the theory of numbers*, volume II, Diophantine Analysis, chapter 3. Chelsea, New York, 1952.
- [dM97] Abraham de Moivre. A method of raising an infinite multinomial to any given power, or extracting any given root of the same. *Philosophical Transactions*, 19(230):619–625, 1697.
- [Edw84] Allen L. Edwards. *An Introduction to Linear Regression and Correlation*. W. H. Freeman and Company, second edition, 1984.
- [Ehr73a] Gideon Ehrlich. Algorithm 466: Four combinatorial algorithms. *Communications of the ACM*, 16(11):690–691, November 1973.
- [Ehr73b] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM*, 20(3):500–513, July 1973.

BIBLIOGRAPHY

- [Er88] M. C. Er. A simple algorithm for generating non-regular trees in lexicographic order. *The Computer Journal*, 31(1):61–64, February 1988.
- [ER03] Scott Effler and Frank Ruskey. A CAT algorithm for generating permutations with a fixed number of inversions. *Information Processing Letters*, 86(2):107–112, April 2003.
- [FH98] Shui Feng and Fred M. Hoppe. Large deviation principles for some random combinatorial structures in population genetics and Brownian motion. *Annals of Applied Probability*, 8(4):975–994, November 1998.
- [Fin03] Steven R. Finch. *Mathematical Constants*. Cambridge University Press, 2003.
- [FL80] Trevor I. Fenner and Georghois Loizou. A binary tree representation and related algorithms for generating integer partitions. *The Computer Journal*, 23(4):332–337, 1980.
- [FL81] Trevor I. Fenner and Georghois Loizou. An analysis of two related loop-free algorithms for generating integer partitions. *Acta Informatica*, 16:237–252, 1981.
- [FL83] Trevor I. Fenner and Georghois Loizou. Tree traversal related algorithms for generating integer partitions. *SIAM Journal on Computing*, 12(3):551–564, August 1983.
- [FNP06] Philippe Flajolet, Markus Nebel, and Helmut Prodinger. The scientific works of Rainer Kemp (1949–2004). *Theoretical Computer Science*, 355(3):371–381, April 2006.
- [Ful97] William Fulton. *Young Tableaux: With Applications to Representation Theory and Geometry*. Cambridge University Press, 1997.

BIBLIOGRAPHY

- [Ful00] Jason Fulman. The Rogers-Ramanujan identities, the finite general linear groups, and the Hall-Littlewood polynomials. *Proceedings of the American Mathematical Society*, 128(1):17–25, 2000.
- [FZC94] Philippe Flajolet, Paul Zimmerman, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1–2):1–35, September 1994.
- [GH99] Siegfried Grossmann and Martin Holthaus. From number theory to statistical mechanics: Bose–Einstein condensation in isolated traps. *Chaos, Solitons & Fractals*, 10(4–5):795–804, April 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
- [GJ04] Ian P. Goulden and David M. Jackson. *Combinatorial Enumeration*. Courier Dover Publications, 2004.
- [GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.
- [GLW82] Udai Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of 2-3 trees. *SIAM Journal on Computing*, 11(3):582–590, August 1982.
- [GLW83] Udai I. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of B-trees. *Journal of Algorithms*, 4(1):51–60, March 1983.
- [GO97] Basil Gordon and Ken Ono. Divisibility of certain partition functions by powers of primes. *The Ramanujan Journal*, 1(1):25–35, January 1997.

BIBLIOGRAPHY

- [Gol93] Leslie A. Goldberg. *Efficient algorithms for listing combinatorial structures*. Cambridge University Press, 1993.
- [Gri00] Ralph P. Grimaldi. Compositions with odd summands. *Congressus Numerantium*, 142:113–127, December 2000.
- [Gri01] Ralph P. Grimaldi. Compositions without the summand 1. *Congressus Numerantium*, 152:33–43, 2001.
- [GS96] Ira M. Gessel and Richard P. Stanley. Algebraic enumeration. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, pages 1021–1062. MIT Press, Cambridge, MA, USA, 1996.
- [Har40] Godfrey H. Hardy. Ramanujan’s work on partitions. In *Ramanujan: twelve lectures on subjects suggested by his life and work*, chapter 6, pages 83–100. Cambridge University Press, Cambridge, 1940.
- [Har66] Godfrey H. Hardy. Asymptotic formulae in combinatory analysis. In *Collected papers of G.H. Hardy: including joint papers with J.E. Littlewood and others edited by a committee appointed by the London Mathematical Society*, volume 1, pages 265–273. Clarendon Press, Oxford, 1966.
- [HCG03] Silvia Heubach, Phyllis Chinn, and Ralph P. Grimaldi. Rises, levels, drops and “+” signs in compositions: Extensions of a paper by Alladi and Hoggatt. *The Fibonacci Quarterly*, 41(3):229–239, June-July 2003.
- [Hic74] D. R. Hickerson. A partition identity of the Euler type. *The American Mathematical Monthly*, 81(6):627–629, June-July 1974.
- [HM04] Silvia Heubach and Toufik Mansour. Compositions of n with parts in a set. *Congressus Numerantium*, 168:127–143, 2004.

BIBLIOGRAPHY

- [HO61] L. Hellerman and S. Ogden. Algorithm 72: Composition generator. *Communications of the ACM*, 4(11):498, November 1961.
- [Hon85] Ross Honsberger. *Mathematical Gems III*. Number 9 in Dolciani Mathematical Expositions. Mathematical Association of America, 1985.
- [HS74] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, April 1974.
- [HW54] Godfrey H. Hardy and Edward M. Wright. *An introduction to the theory of numbers*. Clarendon, Oxford, 3rd edition, 1954.
- [HY97] H. K. Hwang and Y. N. Yeh. Measures of distinctness for random partitions and compositions of an integer. *Advances in Applied Mathematics*, 19(3):378–414, October 1997.
- [JCC] Code conventions for the Java programming language. <http://java.sun.com/docs/codeconv/>.
- [JR76] K. R. James and W. Riha. Algorithm 28: Algorithm for generating graphs of a given partition. *Computing*, 16:153–161, 1976.
- [Kem98] Rainer Kemp. Generating words lexicographically: An average-case analysis. *Acta Informatica*, 35(1):17–89, January 1998.
- [Kin78] J. F. C. Kingman. Random partitions in population genetics. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 361(1704):1–20, May 1978.
- [KKZL05] Anna Kubasiak, Jarosław K. Korbicz, Jakub Zakrzewski, and Maciej Lewenstein. Fermi-Dirac statistics and the number theory. *Europhysics Letters*, 72(4):506–512, 2005.
- [Kli73] Eugene M. Klimko. An algorithm for calculating indices in Fàa Di Bruno’s formula. *BIT*, 13(1):38–49, 1973.

BIBLIOGRAPHY

- [Kli82] Paul Klingsberg. A gray code for compositions. *Journal of Algorithms*, 3(1):41–44, 1982.
- [Knu72] Donald E. Knuth. Mathematical analysis of algorithms. In *Proceedings of IFIP Congress 71 (Amsterdam: North Holland)*, pages 19–27, 1972. Reprinted in his *Selected Papers on the Analysis of Algorithms* (2000), 1–18.
- [Knu73] Donald E. Knuth. The dangers of computer science theory. In *Logic, Methodology and Philosophy of Science 4 (Amsterdam: North-Holland)*, pages 189–195, 1973. Reprinted in his *Selected Papers on the Analysis of Algorithms* (2000), 19–26.
- [Knu94] Donald E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. Addison-Wesley, 1994.
- [Knu04a] Donald E. Knuth. Generating all combinations, 2004. Pre-fascicle 3A of *The Art of Computer Programming*, A draft of section 7.2.1.3. <http://www-cs-faculty.stanford.edu/~knuth/fasc3a.ps.gz>.
- [Knu04b] Donald E. Knuth. Generating all n-tuples, 2004. Pre-fascicle 2A of *The Art of Computer Programming* A draft of section 7.2.1.1. <http://www-cs-faculty.stanford.edu/~knuth/fasc2a.ps.gz>.
- [Knu04c] Donald E. Knuth. Generating all partitions, 2004. Pre-fascicle 3B of *The Art of Computer Programming*, A draft of sections 7.2.1.4–5 <http://www-cs-faculty.stanford.edu/~knuth/fasc3b.ps.gz>.
- [Knu04d] Donald E. Knuth. Generating all permutations, 2004. Pre-fascicle 2B of *The Art of Computer Programming*, A draft of section 7.2.1.2. <http://www-cs-faculty.stanford.edu/~knuth/fasc2b.ps.gz>.

BIBLIOGRAPHY

- [Knu04e] Donald E. Knuth. Generating all trees, 2004. Pre-fascicle 4A of *The Art of Computer Programming*, A draft of section 7.2.1.6. <http://www-cs-faculty.stanford.edu/~knuth/fasc4a.ps.gz>.
- [Knu04f] Donald E. Knuth. History of combinatorial generation, 2004. Pre-fascicle 4B of *The Art of Computer Programming*, A draft of section 7.2.1.7. <http://www-cs-faculty.stanford.edu/~knuth/fasc4b.ps.gz>.
- [KP98] Arnold Knopfmacher and Helmut Prodinger. On Carlitz compositions. *European Journal of Combinatorics*, 19:579–589, 1998.
- [KR05] Arnold Knopfmacher and Neville Robbins. Identities for the total number of parts in partitions of integers. *Utilitas mathematica*, 2005. Preprint: to appear.
- [KS98] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC press LTC, Boca Raton, Florida, 1998.
- [KY76] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and Complexity*, pages 357–428. Academic Press, 1976. Reprinted in Knuth’s *Selected Papers on the Analysis of Algorithms* (2000), 545–603.
- [Leh64] Derrick H. Lehmer. The machine tools of combinatorics. In Edwin F. Beckenbach, editor, *Applied Combinatorial Mathematics*, chapter 1, pages 5–31. Wiley, New York, 1964.
- [Li86] Liwu Li. Ranking and unranking of AVL trees. *SIAM Journal on Computing*, 15(4):1025–1035, November 1986.
- [Lie03] Jens Liebehenschel. Lexicographical generation of a generalized Dyck language. *SIAM Journal on Computing*, 32(4):880–903, 2003.

BIBLIOGRAPHY

- [LP00] Matthieu Latapy and Ha Duong Phan. The lattice of integer partitions and its infinite extension. <http://arxiv.org/abs/math.CO/0008020>, 2000.
- [Mac93] Percy A. MacMahon. Memoir on the theory of the compositions of numbers. *Philosophical Transactions of the Royal Society of London. Series A*, 184:835–901, 1893.
- [Mac08] Percy A. MacMahon. Second memoir on the compositions of numbers. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 207:65–134, 1908.
- [Mac15] Percy A. MacMahon. *Combinatory Analysis*. Cambridge University Press, 1915.
- [Mar99] Stuart Martin. *Schur Algebras and Representation Theory*. Cambridge University Press, 1999.
- [McK65a] J. K. S. McKay. Algorithm 262: Number of restricted partitions of N . *Communications of the ACM*, 8(8):493, August 1965.
- [McK65b] J. K. S. McKay. Algorithm 263: Partition generator. *Communications of the ACM*, 8(8):493, August 1965.
- [McK65c] J. K. S. McKay. Algorithm 264: Map of partitions into integers. *Communications of the ACM*, 8(8):493, August 1965.
- [McK70] J. K. S. McKay. Algorithm 371: Partitions in natural order. *Communications of the ACM*, 13(1):52, January 1970.
- [MM01] Conrado Martínez and Xavier Molinero. A generic approach for the unranking of labeled combinatorial classes. *Random Structures and Algorithms*, 19(3–4):472–497, November 2001.
- [MM05] Conrado Martínez and Xavier Molinero. Efficient iteration in admissible combinatorial classes. *Theoretical Computer Science*, 346(2–3):388–417, November 2005.

BIBLIOGRAPHY

- [MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
- [MR01] Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, September 2001.
- [Nak02] Shin-ichi Nakano. Efficient generation of plane trees. *Information Processing Letters*, 84(3):167–172, November 2002.
- [NMS71] T. V. Narayana, R. M. Mathsen, and J. Saranji. An algorithm for generating partitions and its applications. *Journal of Combinatorial Theory, Series A*, 11(1):54–61, July 1971.
- [NSST98] Jennifer M. Nolan, Vijay Sivaraman, Carla D. Savage, and Pranav K. Tiwari. Graphical basis partitions. *Graphs and Combinatorics*, 14(3):241–261, August 1998.
- [NSW98] Jennifer M. Nolan, Carla D. Savage, and Herbert S. Wilf. Basis partitions. *Discrete Mathematics*, 179(1–3):277–283, January 1998.
- [NW75] Albert Nijenhuis and Herbert S. Wilf. A method and two algorithms on the theory of partitions. *Journal of Combinatorial Theory, Series A*, 18(2):219–222, March 1975.
- [NW78] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, New York, second edition, 1978.
- [Odl96] Andrew M. Odlyzko. Asymptotic enumeration methods. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of combinatorics*, volume II, pages 1063–1229. MIT Press, Cambridge, MA, USA, 1996.

BIBLIOGRAPHY

- [Öhr00] Yngve Öhrn. *Elements of Molecular Symmetry*. Wiley, 2000.
- [O'S04] Edwin O'Shea. M -partitions: optimal partitions of weight for one scale pan. *Discrete Mathematics*, 289(1–3):81–93, December 2004.
- [Pak05] Igor Pak. Partition bijections, a survey. To appear in *Ramanujan Journal*, 2005.
- [Pit97] Jim Pitman. Partition structures derived from Brownian motion and stable subordinators. *Bernoulli*, 3:79–96, 1997.
- [Pla03] Michel Planat. Thermal $1/f$ noise from the theory of partitions: application to a quartz resonator. *Physica A: Statistical Mechanics and its Applications*, 318(3–4):371–386, February 2003.
- [PS03] Sriram Pemmaraju and Steven S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory With Mathematica*. Cambridge University Press, 2003.
- [PW79] E. S. Page and L. B. Wilson. *An Introduction to Computational Combinatorics*. Cambridge University Press, Cambridge, 1979.
- [RECS94] Frank Ruskey, Peter Eades, Bob Cohen, and Aaron Scott. Alley CATs in search of good homes. *Congressus Numerantium*, 102:97–110, 1994.
- [Rio58] John Riordan. *An Introduction to Combinatorial Analysis*. John Wiley and Sons, New York, 1958.
- [RJ76] W. Riha and K. R. James. Algorithm 29: Efficient algorithms for doubly and multiply restricted partitions. *Computing*, 16:163–168, 1976.
- [RND77] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial algorithms: theory and practice*. Ridge Press/Random House, 1977.

BIBLIOGRAPHY

- [RS99] Frank Ruskey and Joe Sawada. An efficient algorithm for generating necklaces with fixed density. *SIAM Journal on Computing*, 29(2):671–684, October 1999.
- [RSC04] Øystein J. Rødseth, James A. Sellers, and Kevin M. Courtright. Arithmetic properties of non-squashing partitions into distinct parts. *Annals of Combinatorics*, 8(3):347–353, August 2004.
- [RSW92] Frank Ruskey, Carla Savage, and Terry Min Yih Wang. Generating necklaces. *Journal of Algorithms*, 13(3):414–430, September 1992.
- [RSW95] David Rasmussen, Carla D. Savage, and Douglas B. West. Gray code enumeration of families of integer partitions. *Journal of Combinatorial Theory, Series A*, 70(2):201–229, May 1995.
- [Rub76] Frank Rubin. Partition of integers. *ACM Transactions on Mathematical Software*, 2(4):364–374, December 1976.
- [Rus78] Frank Ruskey. Generating t -ary trees lexicographically. *SIAM Journal on Computing*, 7(4):424–436, November 1978.
- [Rus01] Frank Ruskey. *Combinatorial Generation*. Working version 1i <http://www.cs.usyd.edu.au/~algo4301/Book.ps>, 2001.
- [Rus05] Frank Ruskey. Combinatorial object server. <http://www.theory.csc.uvic.ca/~cos/>, 2005.
- [Sag00] Marie-France Sagot. Combinatorial algorithms in computational biology. *ERCIM News*, 43, October 2000. http://www.ercim.org/publication/Ercim_News/enw43/sagot.html.
- [Sav89] Carla D. Savage. Gray code sequences of partitions. *Journal of Algorithms*, 10(4):577–595, 1989.
- [Sav97] Carla D. Savage. A survey of combinatorial gray codes. *SIAM Review*, 39(4):605–629, December 1997.

BIBLIOGRAPHY

- [Saw01] Joe Sawada. Generating bracelets in constant amortized time. *SIAM Journal on Computing*, 31(1):259–268, 2001.
- [Sch86] Manfred R. Schroeder. *Number Theory in Science and Communication with Applications in Cryptography, Physics, Digital Information, Computing, and Self-Similarity*. Springer-Verlag, Berlin, second enlarged edition, 1986.
- [Sch04] Peter D. Schumer. *Mathematical Journeys*. Wiley, 2004.
- [Sed77] Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys*, 9(2):137–164, June 1977.
- [Ska88] Wladyslaw Skarbek. Generating ordered trees. *Theoretical Computer Science*, 57(1):153–159, April 1988.
- [Ski90] Steven S. Skiena. *Implementing discrete mathematics: combinatorics and graph theory with mathematica*. Addison-Wesley, Redwood City, California, 1990.
- [Ski98] Steven S. Skiena. *The Algorithm Design Manual*. TELOS—the Electronic Library of Science, Santa Clara, California, 1998.
- [Slo05] Neil J. A. Sloane. The on-line encyclopedia of integer sequences. <http://www.research.att.com/~njas/sequences/>, 2005.
- [SPH01] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Python extension proposal 255: Simple generators. <http://www.python.org/peps/pep-0255.html>, 2001.
- [SS84] Frank W. Schmidt and Rodica Simion. On a partition identity. *Journal of Combinatorial Theory, Series A*, 36(2):249–252, March 1984.
- [SS96] Laura A. Sanchis and Matthew B. Squire. Parallel algorithms for counting and randomly generating integer partitions. *Journal of Parallel and Distributed Computing*, 34:29–35, 1996.

BIBLIOGRAPHY

- [SS05] Neil J. A. Sloane and James A. Sellers. On non-squashing partitions. *Discrete Mathematics*, 294(3):259–274, May 2005.
- [Sta86] Richard P. Stanley. *Enumerative Combinatorics*. Wadsworth, Belmont, California, 1986.
- [Sto62a] Frank Stockmal. Algorithm 114: Generation of partitions with constraints. *Communications of the ACM*, 5(8):434, August 1962.
- [Sto62b] Frank Stockmal. Algorithm 95: Generation of partitions in part-count form. *Communications of the ACM*, 5(6):344, June 1962.
- [Sto92] Ivan Stojmenović. On random and adaptive parallel generation of combinatorial objects. *International Journal of Computer Math*, 42:125–135, 1992.
- [SW86] Dennis Stanton and Dennis White. *Constructive combinatorics*. Springer-Verlag, Berlin, 1986.
- [Syl82] James J. Sylvester. A constructive theory of partitions, arranged in three acts, an interact and an exodion. *American Journal of Mathematics*, 5(1/4):251–330, 1882.
- [Tem49] H. N. V. Temperley. Statistical mechanics and the partition of numbers. I. The transition of liquid helium. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 199(1058):361–375, November 1949.
- [TMB04] Muoi N. Tran, M. V. N. Murthy, and Rajat K. Bhaduri. On the quantum density of states and partitioning an integer. *Annals of Physics*, 311(1):204–219, May 2004.
- [Tom82] C. Tomasi. Two simple algorithms for the generation of partitions of an integer. *Alta Frequenza*, 51(6):352–356, 1982.

BIBLIOGRAPHY

- [Tro78] Anthony E. Trojanowski. Ranking and listing algorithms for k -ary trees. *SIAM Journal on Computing*, 7(4):492–509, November 1978.
- [Tuc84] Alan Tucker. *Applied Combinatorics*. Wiley, second edition, 1984.
- [Tun99] Wu-Ki Tung. *Group Theory in Physics*. World Scientific, 1999.
- [Wal00] Timothy R. Walsh. Loop-free sequencing of bounded integer compositions. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 33:323–345, 2000.
- [Wei01] Karsten Weihe. A software engineering perspective on algorithmics. *ACM Computing Surveys*, 33(1):89–134, March 2001.
- [Wel71] Mark B. Wells. *Elements of Combinatorial Computing*. Pergamon Press, Oxford, 1971.
- [Whi70a] J. S. White. Algorithm 373: Number of doubly restricted partitions. *Communications of the ACM*, 13(2):120, February 1970.
- [Whi70b] J. S. White. Algorithm 374: Restricted partition generator. *Communications of the ACM*, 13(2):120, February 1970.
- [Wil85] S. Gill Williamson. *Combinatorics for Computer Science*. Computer Science Press, 1985.
- [Wil89] Herbert S. Wilf. *Combinatorial Algorithms: an update*. Society for Industrial & Applied Mathematics, Philadelphia, 1989.
- [Wil90] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, San Diego, 1990.
- [Wil02] Herbert S. Wilf. Lectures on integer partitions. Pacific Institute for the Mathematical Sciences, 2002.

BIBLIOGRAPHY

- [WROM86] Robert Alan Wright, Bruce Richmond, Andrew Odlyzko, and Brendan D. McKay. Constant time generation of free trees. *SIAM Journal on Computing*, 15(2):540–548, May 1986.
- [Yan00] Winston C. Yang. Derivatives are essentially integer partitions. *Discrete Mathematics*, 222(1–3):235–245, July 2000.
- [Yee01] Ae Ja Yee. On the combinatorics of lecture hall partitions. *The Ramanujan Journal*, 5(3):247–262, September 2001.
- [Yee04] Ae Ja Yee. Partitions with difference conditions and Alder’s conjecture. *Proceedings of the National Academy of Sciences of the United States of America*, 101(47):16417–16418, November 2004.
- [ZS98] Antoine Zoghbi and Ivan Stojmenović. Fast algorithms for generating integer partitions. *International Journal of Computer Math*, 70:319–332, 1998.