NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
Algorithms for Generating

Integer Partitions

By

ANTOINE C. ZOGHBI, B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
University of Ottawa
Ottawa, Ontario
July 20, 1993

©copyright 1993,
Antoine Zoghbi
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Abstract

In this thesis we consider the problem of generating integer partitions. We provide an overview of all known algorithms for the sequential generation of partitions of an integer. The performance is measured and compared separately for the standard and multiplicity representation of integer partitions.

We present two new algorithms for generating integer partitions in the standard representation which generate partitions in lexicographic and antilexicographic order respectively. We prove that both algorithms generate partitions with constant average delay (exclusive of the output; output is generated and not printed). Historically, all existing algorithms for generating integer partitions in the multiplicity representation showed better performance than all the existing algorithms for generating integer partitions in the standard representation. An empirical test shows that both new algorithms are a few times faster than any previously known algorithms for generating unrestricted integer partitions in the standard representation. Moreover, they are faster than any known algorithm for generating integer partitions in the multiplicity representation (exclusive of the output).
We describe several modifications to existing algorithms, and a transformation of one algorithm from the standard to the multiplicity representation. Finally, we provide a brief overview of sequential and parallel algorithms that generate partitions at random, and an analysis of a parallel algorithm for generating all partitions.
ACKNOWLEDGMENTS

I wish to acknowledge several persons who directly or indirectly contributed to this thesis. I would first of all like to thank my supervisor, Professor Ivan Stojmenovic. Professor Stojmenovic has provided knowledgeable guidance and assistance throughout the course of my research. Indirectly, he has encouraged me by saying "it is so simple". But in reality it is not!

I would also like to thank the Computer Science Department at the University of Ottawa, which gave me the opportunity to do my research. All the professors, secretaries, and persons in charge have been very helpful.

I would like to thank my wife who arranged many things to assure me the maximum possible time for research and studies. My wife and my little daughter have suffered a lot from my involvements in my Master's program. I'm grateful for their patience and support.

Finally, I would like to thank my special friends who encouraged me to study in the Master's program. Although they prefer to remain anonymous, their help will never be forgotten.
# Contents

Abstract ................................................................. 1

Acknowledgments ........................................................ 3

Contents ................................................................. 4

List of Tables ........................................................... 6

List of Algorithms ....................................................... 8

List of Figures .......................................................... 9

List of Programs ........................................................ 10

1 Introduction ............................................................ 12
   1.1 Definitions ....................................................... 14
   1.2 Research Problems and Contributions ....................... 24
   1.3 Applications ..................................................... 27

2 Survey on Generating Integer Partitions .......................... 31

3 Implementation of Algorithms ...................................... 43
   3.1 Introduction ..................................................... 43
   3.2 Antilexicographic Order and Standard Representation .... 48
   3.3 Antilexicographic Order and Multiplicity Representation .. 50
   3.4 Lexicographic Order and Multiplicity Representation ...... 51
   3.5 Unranking ......................................................... 53
   3.6 Part Order ......................................................... 54
   3.7 Non Lexicographic Order ....................................... 56
   3.8 Modifications to Existing Algorithms ....................... 57
## Tables

Table 1  Unrestricted partitions of 8  
Table 2  Number of partitions of n using at most m parts  
Table 3  Number of unrestricted partitions of n  
Table 4  Algorithm M1  
Table 5  Algorithm PW1  
Table 6  Algorithm PW2  
Table 7  Algorithm A  
Table 8  Algorithm NW  
Table 9  Algorithm AS  
Table 10  Algorithm FL3  
Table 11  Algorithm FL1  
Table 12  Algorithm E  
Table 13  Algorithm S  
Table 14  Algorithm S’  
Table 15  Algorithm M2  
Table 16  Algorithm W  
Table 17  Algorithm GLW  

16  
20  
21  
48  
48  
49  
49  
50  
50  
51  
51  
52  
52  
53  
53  
54  
54
Algorithms

1- Algorithm RP ................................................................. 19
2- Algorithm ZS1 ............................................................... 61
3- Algorithm ZS2 ............................................................... 64
4- Algorithm Z ................................................................. 76
Figures

Fig 1  Ferrers Graph ......................................................... 22
Fig 2  Graphs ................................................................. 28
Fig 3  Classification of all algorithms .......................... 33
Fig 4  Binary tree ............................................................. 39
### Programs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FL1</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>ZS2</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>NW</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>AS</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>FL3</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>M1</td>
<td>101</td>
</tr>
<tr>
<td>8</td>
<td>PW1</td>
<td>101</td>
</tr>
<tr>
<td>9</td>
<td>PW2</td>
<td>101</td>
</tr>
<tr>
<td>10</td>
<td>ZS1</td>
<td>101</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>102</td>
</tr>
<tr>
<td>12</td>
<td>M2</td>
<td>103</td>
</tr>
<tr>
<td>13</td>
<td>FL2</td>
<td>104</td>
</tr>
<tr>
<td>14</td>
<td>Z</td>
<td>104</td>
</tr>
<tr>
<td>15</td>
<td>H</td>
<td>104</td>
</tr>
<tr>
<td>16</td>
<td>S</td>
<td>105</td>
</tr>
</tbody>
</table>
Chapter I

Introduction

Given an integer $n$, it is possible to represent it as the sum of one or more positive integers, i.e. $n = x_1 + x_2 + \ldots + x_m$. This representation is called a partition if the order of the $x_i$ is of no consequence. Thus two partitions of an integer $n$ are distinct if they differ with respect to the $x_i$ they contain. For example, there are seven distinct partitions of the integer 5: $5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1$.

The partitions of integers have been the subject of extensive study for over 300 years. In 1669, Leibniz asked Bernoulli if he had investigated $P(n)$, the number of partitions of an integer $n$. Details of the history and the state of the art as of 1920 can be found in Chapter 3 of [D]. Additional details and later results can be found in most combinatorics texts, in particular [An, RND, Li, Ri]. The first known algorithm, $H$, was discovered by Hindenburg in 1778. Since the early 1960’s there has been a dramatic increase in the
discovery and publication of new algorithms beginning with Algorithm S [S] discovered by Stockmal and published in 1962. Algorithms M1 [MK1] and M2 [MK] by McKay were published in 1965 and 1970. Algorithm W [Wh] by White was published in 1970. Algorithm We [W] by Wells was published in 1971. Algorithm NW [NW] by Nijenhuis and Wilf was published in 1975. In 1976, algorithm A [An] by Andrews, algorithm Ru [Ru2] by Rubin, and algorithms RJ1, RJ2 [RJ] by Riha and James were published. Algorithm E [RND] by Ehrlich was published in 1977. Algorithms PW1 and PW2 [PW] by Page and Wilson were published in 1979. Algorithms FL1, FL2 [FL] and FL3 [FL3] by Fenner and Louizou were published in 1980 and 1981. Algorithm T [T] by Tomasi was published in 1982. Algorithm GLW [GLW] by Gupta, Lee and Wong was published in 1983. Algorithm Sa [Sa] by Savage was published in 1988. Algorithm Ss [Ss] by Skiena was published in 1990. Finally algorithm AS [A] by Akl and Stojmenovic appeared recently. This interest is partly motivated by the important role played by partitions and compositions in many problems of combinatorics and algebra [Be, 1971 and B, 1973]. In general, a list of all combinatorial objects of a given type might be used to search for a counter-example to some conjecture, or to test and analyze an algorithm for its correctness or computational complexity. For computational purposes one is often interested in generating all the partitions of an integer, or those satisfying various restrictive conditions. Several such algorithms, dealing with both the unrestricted [An, AS, FL, FL2, Le, Mk, Mk1, NW, PW, RND, RJ, Sa, T] and restricted [An, Le, GLW, NMS, RJ, Wh, Ru2, S] cases, have appeared in combinatorics literature.
This thesis is organized as follows. Chapter I gives definitions and relations between various kinds and representations of integer partitions. Chapter II gives a survey of generating algorithms. Chapter III provides an implementation of all existing algorithms. Chapter IV describes two new algorithms for generating integer partitions in *standard representation*, and a new algorithm for generating integer partitions in *multiplicity representation*. Chapter V shows the analysis of the proposed new algorithms compared with the existing ones. Chapter VI surveys the generation of random partitions sequentially and in parallel, the generation of the integer partitions in parallel, the average number of parts used in partitions, and its impact on the efficiency of parallel generating algorithms. Finally, Chapter VII presents open problems and conclusions.

1.1 Definitions

**Lexicographic order** of combinatorial objects is defined as follows. If \( A = (a_1, a_2, ..., a_r) \) and \( B = (b_1, b_2, ..., b_r) \) are representations of objects, then \( A \) precedes \( B \) lexicographically if and only if, for some \( j \geq 1 \), \( a_i = b_i \) when \( i < j \), and \( a_j \) precedes \( b_j \). For example, partitions of 5 in lexicographic order are: 11111, 2111, 221, 311, 32, 41, 5 (note that the '+' sign is omitted).
Lexicographic order is desirable as it is the natural (dictionary) order, and can be easily characterized and traced manually. The antilexicographic order is the reverse of the lexicographic order. If the partitions are arranged in neither lexicographic nor antilexicographic order, then we refer to them as being in non lexicographic order. The Gray code order or minimal change order [Sa] is the list of all partitions of an integer \( n \) in such a way that a partition differs from its predecessor on the list only in that one part has increased by 1 and another part has decreased by 1. (A part of size 1 may decrease to 0 or a part of size 0 may increase to 1). For the integer 15, the successors of the partition 7521 are 75111, 66111, 6621, 663, 762, 7611, 771, 87, 861, and so on. The part order is the list of all partitions containing exactly \( m \) parts, where \( m \) varies from 1 to \( n \). For integer 7 the partitions are 7, 61, 52, 43, 511, 421, 331, 322, 4111, 3211, 2221, 31111, 22111, 211111, and 111111. For each \( m \) the order is not necessarily antilexicographic. There exist other orders which are primarily concerned with the objects under consideration.

In standard representation, a partition of \( n \) is given by a sequence \( x_1, \ldots, x_m \), where \( x_1 \geq x_2 \geq \ldots \geq x_m \), and \( x_1 + x_2 + \ldots + x_m = n \). Hereafter, \( x \) will denote an arbitrary partition and \( m \) will denote the number of parts of \( n \) (\( m \) is usually not fixed).
It is sometimes more convenient to use a *multiplicity representation* for partitions in terms of a list of the distinct parts of the partition and their respective multiplicities. Let $y_1, \ldots, y_d$ be all distinct parts in a partition, and $c_1, \ldots, c_d$ their respective (positive) multiplicities. Clearly $c_1 y_1 + \ldots + c_d y_d = n$.

Table 1  Unrestricted partitions of 8

<table>
<thead>
<tr>
<th>#</th>
<th>standard representation</th>
<th>multiplicity representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>(8X1)</td>
</tr>
<tr>
<td>2</td>
<td>7 1</td>
<td>(7X1) (1X1)</td>
</tr>
<tr>
<td>3</td>
<td>6 2</td>
<td>(6X1) (2X1)</td>
</tr>
<tr>
<td>4</td>
<td>6 1 1</td>
<td>(6X1) (1X2)</td>
</tr>
<tr>
<td>5</td>
<td>5 3</td>
<td>(5X1) (3X1)</td>
</tr>
<tr>
<td>6</td>
<td>5 2 1</td>
<td>(5X1) (2X1) (1X1)</td>
</tr>
<tr>
<td>7</td>
<td>5 1 1 1</td>
<td>(5X1) (1X3)</td>
</tr>
<tr>
<td>8</td>
<td>4 4</td>
<td>(4X2)</td>
</tr>
<tr>
<td>9</td>
<td>4 3 1</td>
<td>(4X1) (3X1) (1X1)</td>
</tr>
<tr>
<td>10</td>
<td>4 2 2</td>
<td>(4X1) (2X2)</td>
</tr>
<tr>
<td>11</td>
<td>4 2 1 1</td>
<td>(4X1) (2X1) (1X2)</td>
</tr>
<tr>
<td>12</td>
<td>4 1 1 1 1</td>
<td>(4X1) (1X4)</td>
</tr>
<tr>
<td>13</td>
<td>3 3 2</td>
<td>(3X2) (2X1)</td>
</tr>
<tr>
<td>14</td>
<td>3 3 1 1</td>
<td>(3X1) (2X2) (1X1)</td>
</tr>
<tr>
<td>15</td>
<td>3 2 2 1</td>
<td>(3X1) (2X1) (1X3)</td>
</tr>
<tr>
<td>16</td>
<td>3 2 1 1 1</td>
<td>(3X1) (1X5)</td>
</tr>
<tr>
<td>17</td>
<td>3 1 1 1 1 1</td>
<td>(2X4)</td>
</tr>
<tr>
<td>18</td>
<td>2 2 2 2</td>
<td>(2X3) (1X2)</td>
</tr>
<tr>
<td>19</td>
<td>2 2 2 1 1</td>
<td>(2X2) (1X4)</td>
</tr>
<tr>
<td>20</td>
<td>2 2 1 1 1 1</td>
<td>(2X1) (1X6)</td>
</tr>
<tr>
<td>21</td>
<td>2 1 1 1 1 1 1</td>
<td>(1X8)</td>
</tr>
</tbody>
</table>

Reading the chart from the top down, we see all unrestricted partitions in antilexicographic order, while reading it from the bottom up the partitions appear in lexicographic order.
Unrestricted partitions refers to the case of partitions without any limitations. Let restricted partitions be those partitions for which \( x_m \leq L2 \) is satisfied, i.e. partitions whose largest part is no greater than \( L2 \). Let \( RP(n,L2) \) be the number of restricted partitions of \( n \) whose largest part is no larger than \( L2 \). Restricted partitions can be generated using an algorithm to generate unrestricted partitions in lexicographic order and stopping the algorithm when the first part becomes greater than \( L2 \) (or starting with first partition \( y_1 = L2, c_1 = \lfloor n/y_1 \rfloor \) define \( x_1 \), \( y_2 = n - c_1 y_1 \), \( c_2 = 1 \) if \( y_2 > 0 \), \( c_2 = 0 \) otherwise, in case of antilexicographic order).

The cardinality of the number of unrestricted partitions \( P(n) \) was given by the asymptotic formula as follows:

\[
P(n) = \frac{1}{4 \pi \sqrt{3}} e^{\pi \sqrt{2/3 + c_n n^{-3/2}}}
\]

where the relative error is \( O(n^{-1/2}) \) [HR].

Note that the relative error is defined as \( \frac{p - p'}{p} \)

with \( p \) and \( p' \) are the exact and the estimated value respectively.
Now

\[ \log P(n) = \log e^{\sqrt{\frac{n}{3}}} - \log 4n\sqrt{3} \]

\[ = \sqrt{2\frac{n}{3}} - \log 4\sqrt{3} - \log n \]

\[ = O(\sqrt{n}) \]

Thus, the representation of \( P(n) \) as an integer requires \( O(\sqrt{n}) \) digits, or \( O(\sqrt{n}/\log n) \) memory locations, assuming that one memory location is capable of storing an integer \( O(n) \) (having \( O(\log n) \) bits).

The number of partitions that generate \( n \) using at most \( m \) parts can be calculated using the following recursive formula:

\[ \text{RP}(n,m) = \text{RP}(n,m-1) + \text{RP}(n-m, \min(n-m,m)) \]

with \( \text{RP}(n,1)=1 \), \( \text{RP}(n,0)=0 \), \( \text{RP}(0,0)=1 \), and \( \text{RP}(n,m)=0 \) if \( m>n \).
That the recurrence relation in the formula is true may be seen by classifying the partitions into those with their largest part exactly equal to \( m \) (the second summand) and those with their largest part less than \( m \) (the first summand). Using these numbers we may establish a one-to-one correspondence between the unrestricted partitions of \( n \) and the numbers 0, 1, \ldots, \text{RP}(n, n)-1 \ [\text{Mk}^\text{"n}]. \) The number of unrestricted partitions \( \text{P}(n) \) is equal to \( \text{RP}(n, n) \).

The formula can be implemented in the following way without using recursive techniques (see programs M2 and W).

**Algorithm RP**

\[
\text{R}_{0,0} = 1;
\]

\textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \) \textbf{do} \{

\text{R}_{i,0} = 0
\]

\textbf{for} \( j \leftarrow 1 \) \textbf{to} \( i \) \textbf{do} \text{ R}_{ij} = \text{R}_{ij-1} + \text{R}_{ij, \text{min}(i-j,0)}

\]

The table on the following page illustrates the calculation of \( \text{RP}(n,m) \).
Table 2  Number of partitions of \( n \) using at most \( m \) parts

<table>
<thead>
<tr>
<th>( m ) →</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>18</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>12</td>
<td>18</td>
<td>23</td>
<td>26</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>14</td>
<td>23</td>
<td>30</td>
<td>35</td>
<td>38</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>16</td>
<td>27</td>
<td>37</td>
<td>44</td>
<td>49</td>
<td>52</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>19</td>
<td>34</td>
<td>47</td>
<td>58</td>
<td>65</td>
<td>70</td>
<td>73</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>

The diagonal shows \( P(n) \).

The exact number of unrestricted partitions \( P(n) \) can be obtained by implementing \( RP(n, m) \) or \( PE(n, m) \). In both cases the implementation needs a two dimensional array of size \( n \). On the SUN workstation and with 4 MBytes memory, the maximum size of \( n \) that could be used was 510. \( P(n) \) can be summarized as follows:
Table 3  Number of unrestricted partitions of \( n \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>176</td>
</tr>
<tr>
<td>30</td>
<td>5,604</td>
</tr>
<tr>
<td>45</td>
<td>89,134</td>
</tr>
<tr>
<td>60</td>
<td>966,467</td>
</tr>
<tr>
<td>75</td>
<td>8,118,264</td>
</tr>
<tr>
<td>90</td>
<td>56,634,173</td>
</tr>
<tr>
<td>105</td>
<td>342,325,709</td>
</tr>
<tr>
<td>150</td>
<td>40,853,235,548</td>
</tr>
<tr>
<td>195</td>
<td>2,580,840,231,841</td>
</tr>
<tr>
<td>240</td>
<td>105,882,249,516,398</td>
</tr>
<tr>
<td>285</td>
<td>3,160,137,919,927,934</td>
</tr>
<tr>
<td>330</td>
<td>73,653,287,464,863,616</td>
</tr>
<tr>
<td>375</td>
<td>1,406,207,445,431,407,104</td>
</tr>
<tr>
<td>420</td>
<td>22,755,289,805,217,304,576</td>
</tr>
<tr>
<td>465</td>
<td>320,103,127,009,195,196,416</td>
</tr>
<tr>
<td>510</td>
<td>3,991,268,667,606,861,086,720</td>
</tr>
</tbody>
</table>

Doubly restricted partitions contain parts of size between \( L_1 \) and \( L_2 \), i.e. \( L_1 \leq x_i \leq L_2 \) for \( i=1,2,\ldots,m \). Multiply restricted partitions of \( n \) is a common name for various kinds of special partitions. Examples are partitions with prescribed part sizes (those with parts which are selected from an array \( v_i, i=1,2,\ldots,r \)). Tournament scores are studied in [NMS], while graph degree sequences are studied in [JR].
The case of partitions whose largest part is exactly $L_2$ is given in [NW] as a special case of their general method for listing, ranking and unranking combinatorial objects. Note that the case of partitions of $n$ whose largest part is no greater than $L_2$ (by adding one more part of size $L_2$) is equivalent to the case of partitions of $n+L_2$ with the largest part exactly $L_2$.

Using the Ferrers graph [PW], a one-to-one correspondence between partitions of $n$ into $m$ parts and partitions of $n$ whose largest part is $m$ is established. Let $z_1 \ldots z_m$ be a partition into $m$ nonincreasing parts, $z_1 \geq \ldots \geq z_m$, and $x_1 \ldots x_k$, $x_1 \geq \ldots \geq x_k$, be a partition of $n$ into any number of nonincreasing parts (i.e. $k$ varies) with largest part $x_1 = m$. The following Ferrers graph illustrates the relationship between the two kinds of partitions.

\begin{center}
\begin{tabular}{ccccccc}
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & z_1=6 \\
  \cdot & \cdot & \cdot & \cdot & \cdot & & & z_2=5 \\
  \cdot & \cdot & \cdot & & & & & z_3=3 \\
  \cdot & \cdot & \cdot & & & & & z_4=3 \\
  \cdot & \cdot & & & & & & z_5=2 \\
\end{tabular}
\end{center}

$x_1=5 \quad x_2=5 \quad x_3=4 \quad x_4=2 \quad x_5=2 \quad x_6=1$

Fig. 1. Ferrers graph
From the Ferrers graph it follows that $z_j = \max \{ j \mid x_j \geq i \}$. Consider now the corresponding \textit{multiplicity representation} of partitions with largest part $m$: $c_1, \ldots, c_d$ are the multiplicities of $y_1, \ldots, y_d$, where $m = y_1 > \ldots > y_d$, and $c_1y_1 + \ldots + c_dy_d = n$. For the partitions into $m$ parts, let $e_1, \ldots, e_d$ be the multiplicities of $w_1, \ldots, w_d$ (clearly the two sequences have the same number $d$ of different parts), where $w_1 > \ldots > w_d$, $e_1w_1 + \ldots + e_dw_d = n$, and $e_1 + \ldots + e_d = m$. It then follows that $w_{m+1} + 1 = c_1 + c_2 + \ldots + c_i$, and $e_{m+1} + 1 = y_1 - y_{i+1}$ where $y_{d+1} = 0$. The sums for $w_i$ can be easily maintained during the execution of a program for generating partitions of $n$ with largest part $m$ in the \textit{multiplicity representation}.

The delay between two partitions is the time required to generate a new partition from an existing one. Delay is constant if the time is constant, assuming that the time to output partitions is not counted. Obviously, a procedure for generating the next partition from a current one has constant delay if it is loop-free and recursion-free. The average delay is the ratio of the total time to generate all partitions to the total number of partitions generated. An algorithm has \textit{constant average delay} property if the ratio is less than a constant for any $n$, again, exclusive of the output time.

Hereafter, $[x]$ is an integer $k$ such that $k-1 < [x] \leq k$. For example $[3.2] = 4$. Also, $\lfloor x \rfloor$ is an integer $k'$ such that $k' \leq \lfloor x \rfloor < k' + 1$. For example $\lfloor 3.2 \rfloor = 3$. 

23
1.2 Research Problems and Contributions

Our research focused on finding fast sequential algorithms for generating integer partitions. Studying all the existing algorithms, we felt that the cost time necessary to generate partitions could be reduced with the use of a new algorithm. Also, we analyzed the generation of the unrestricted partitions of an integer in the standard representation and in a lexicographic order; this had not previously been studied. Our contributions in this domain can be summarized as follows:

1- A new algorithm, ZS1, which generates unrestricted partitions in the standard representation and in an antilexicographic order was proposed. This algorithm demonstrates the best performance ever known, and could be close to the upper bound for the generation of the unrestricted partitions (excluding output). Details of these aspects of the new algorithm are in Chapter IV of this thesis.

2- A new algorithm, ZS2, which generates unrestricted partitions in the standard representation and in lexicographic order was proposed. This is the only algorithm
in this category. Its performance is very competitive with ZS1 and better than all other previously known algorithms. Details of ZS2 are in Chapter IV of this thesis.

3- A given proof shows that both algorithms ZS1 and ZS2 have a constant delay property. To the best of our knowledge, they are the only algorithms in the standard representation form that have such a property. Details are in Chapter IV of this thesis.

4- An intensive survey of all existing sequential algorithms was conducted, including the arrangement of the algorithms into different categories and groups. Details appear in Chapter II of this thesis.

5- All known algorithms were implemented, measured, and compared. To the best of our knowledge, this is the first comparison of algorithms for generating integer partitions. Similar analyses exist for permutations [Se], combinations [A] and set partitions [DMSSS]. Implementation details are in Chapter III and analysis details are in Chapter V of this thesis.
6- A new algorithm, Z, is presented in the *multiplicity representation*. This algorithm is in fact a transformation of the algorithm which was discovered by K.F. Hindenburg in 1778. The latter generates all unrestricted partitions of an integer in the *standard representation*. Details are in Chapter IV of this thesis.

7- Algorithms that were traditionally written in an outdated language have been recoded; in some cases major changes were necessary. Details are in Chapter III of this thesis.

8- The cost of an optimal parallel algorithm for generating integer partitions is recalculated. Some algorithms generate the partitions of an integer in parallel ([AS], [S1]). In [AS] an algorithm for generating all partitions of the integer \( n \) in antilexicographic order using \( n \) processors on a linear array is given and has cost \( O(nP(n)) \). Our measure indicates that the average number of parts in the unrestricted partitions is \( O(n^{23}) \). Thus, the cost-optimal solution for \( n \) processors should be \( O(n^{23}P(n)) \). Details are in Chapter VI of this thesis.

Items 1, 2, and 3 of the above list were accomplished jointly with my supervisor,
1.3 Applications

Partitions play a fundamental role in many combinatorial and graph theoretical problems. Let \((x_1, \ldots, x_k)\) be a partition of \(n\) where we assume \(x_1 \leq x_2 \leq \ldots \leq x_k\). In many combinatorial problems it is of interest, for moderate \(n\), to be able to list these partitions explicitly and to know the number \(k\) of parts of \(n\), where \(x_i (i=1, \ldots, k)\) satisfies certain additional restrictions. For example, if

\[
n = \frac{k!}{2! (k-2)!} \quad \text{with} \quad \sum_{i=1}^{j} x_i \geq \frac{j!}{2! (j-2)!}
\]

for \(j = 1, \ldots, k-1\), we obtain all the different score structures in a round-robin tournament, [NMS].
Consider the following graphs

Graph (a)

Graph (b)

Fig. 2. Graphs

In a graph the degree of the vertex is defined as the number of edges connected to it. The above graphs illustrate the degree of vertices. The list of all possible degrees in a graph of \( p \) vertices and \( q \) edges, is obtained by partitioning an integer of \( 2q \) into exactly \( p \) parts with some lower and upper bounds. For example, with \( p = q = 5 \), \( PE(10,5) = 7 \) \( (22222, 32221, 33211, 42211, 43111, 52111, \) and \( 61111) \). The partition \( 33211 \) represents graph (a), and the partition \( 22222 \) represents graph (b). A similar idea is discussed by [RJ].
A B-tree of order \( m \) is a tree satisfying the following properties [BM]: (1) All leaves are on the same level; (2) the root has \( k \) descendants, \( \lceil m/2 \rceil \leq k \leq m \); and (3) other internal nodes have \( k' \) descendants, \( \lceil m/2 \rceil \leq k' \leq m \). Given an integer \( n \), a partition of size \( s \) and the set of consecutive integers \( M = \{m_1, m_1+1, \ldots, m_2\} \), where \( m_1, m_2, n, s \) are all positive numbers, let \( P(n, s; m_1, m_2) \) be a procedure that generates all partitions of \( n \) of size \( s \) using integers between \( m_1 \) and \( m_2 \). To generate all B-trees of order \( m \) with a fixed number of leaves, we need to generate \( P(n, s; \lfloor m/2 \rfloor, m) \) [GLW].

The generation schemes for partitions of a whole number and for partitions of a set are required in many combinatorial investigations, often as a basis for more involved generations. Since numerical partitions naturally arise in many classification schemes (one example is the classification of group elements by cycle structure), their serial number calculation often appears as one step in an involved indexing scheme. Such an application appears in the incidence matrix invariant calculation. An application of the special set partition generation scheme appears in the four-colour problem discussion [W].

In Volterra’s system, which describes the transformation of a certain input into a certain output form, \( \Sigma_{v_{p,k}} \) denotes the summation over sets of integers \( v_i \) such that

\[
v_1 + v_2 + \ldots + v_p = k, \ 1 \leq v_1 \leq v_2 \leq \ldots \leq v_p.
\]
In other words, the summation is taken over those partitions of $k$ which have $p$ parts $[T]$. 

Finally, a free memory is the memory available for use. Due to a variety of reasons this free memory is fragmented into many pieces. The storage of files on disks is a similar problem. At a certain time some files are deleted and some files are added of variant sizes. The listing of all possible free areas into the computer memory or into the disk may be represented as the problem of generating all unrestricted partitions of an integer $k$ where $k$ MBytes represents the total free memory size or the total free disk storage size.
Chapter II

Survey on Generating Integer Partitions

In this chapter we briefly describe all known sequential algorithms for generating integer partitions. Algorithms are divided according to the kind of partitions generated (unrestricted partitions, restricted partitions, doubly restricted partitions, and multiply restricted partitions), the representation of the partitions generated (standard representation, multiplicity representation, and combined representation), and the order of generating partitions (lexicographic order, antilexicographic order, non lexicographic, Gray code order, part order and M-order; M-order will be discussed later on in this Chapter). See Chapter I for the other definitions.

An algorithm for generating doubly restricted partitions clearly can be used to generate restricted or unrestricted partitions. Also, an algorithm which generates restricted partitions can be used to produce unrestricted partitions.
Each algorithm is named with the initials of its authors. These abbreviations are used later in comparison tables. The algorithms are categorized as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>a - Algorithms that generate unrestricted partitions</td>
<td>19</td>
</tr>
<tr>
<td>b - &quot;       &quot;       &quot; restricted partitions</td>
<td>1</td>
</tr>
<tr>
<td>c - &quot;       &quot;       &quot; doubly restricted partitions</td>
<td>3</td>
</tr>
<tr>
<td>d - &quot;       &quot;       &quot; Multiply restricted partitions</td>
<td>2</td>
</tr>
</tbody>
</table>

According to the representation form

<table>
<thead>
<tr>
<th>Representation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a - Algorithms that generate the output in the standard representation</td>
<td>16</td>
</tr>
<tr>
<td>b - Algorithms that generate the output in the multiplicity representation</td>
<td>9</td>
</tr>
</tbody>
</table>

According to the output order

<table>
<thead>
<tr>
<th>Order</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a - Algorithms that generate the partitions in lexicographic order</td>
<td>4</td>
</tr>
<tr>
<td>b - &quot;       &quot;       &quot; antilexicographic order</td>
<td>10</td>
</tr>
<tr>
<td>c - &quot;       &quot;       &quot; non lexicographic order</td>
<td>5</td>
</tr>
<tr>
<td>d - &quot;       &quot;       &quot; part order</td>
<td>4</td>
</tr>
<tr>
<td>e - &quot;       &quot;       &quot; Unranking order</td>
<td>2</td>
</tr>
</tbody>
</table>

The classification of all known sequential algorithms is shown on the next page.
Fig. 3. Classification of all sequential algorithms

UP = Unrestricted Partitions
RP = Restricted Partitions
DRP = Doubly Restricted Partitions
MRP = Multiply Restricted Partitions

L = Lexicographic Order
A = Antilexicographic Order
U = Unranking
P = Part Order
N = Non lexicographic Order
M = Multiplicity Representation
S = Standard Representation
In antilexicographic order, a partition is derived from the previous one by subtracting 1 from the rightmost part greater than 1, and distributing the remainder as quickly as possible. For example, the partition following 9+7+6+1+1+1+1+1+1 is 9+7+5+5+2. Such an algorithm was first reported by F. Stockmal [S] (algorithm S), and it generates restricted partitions. In algorithm S, each partition is represented by the integers c[1] through c[L2], where the part is represented by the array index j, and c[j] represents its multiplicity. To generate a new partition the algorithm calculates the remainder by scanning the existing partition from left to right until the remainder to be distributed is greater than the current index k. Then, c[k] is increased by 1, and c[1] is set to the remainder. For example, the integer 5 can be generated as follows: 50000, 31000, 12000, 20100, 01100, 10010 and 00001, where 20100 means 1X2 + 3X1. This combined representation contains zeros and does not have constant delay property.

In standard representation and antilexicographic order, the next partition is determined from the current one \(x_1 \times x_2 \ldots \times x_m\) in the following way: Let \(h\) be the number of parts of \(x\) greater than 1, i.e. \(x_i > 1\) for \(1 \leq i \leq h\), and \(x_i = 1\) for \(h < i \leq m\). If \(x_m > 1\) (or \(h = m\)) then the next partition is \(x_1, x_2, \ldots, x_{m-1}, x_m-1, 1\). Otherwise (i.e. \(h < m\)), the next partition with \((x_{h-1}), (x_{h-1}), \ldots, (x_{h-1})\), \(d\), contains \(c\) elements, where \(0 < d \leq x_h - 1\) and \((x_{h-1})(c-1)+d = x_h + m-h\). Based on this general idea, several algorithms were developed: algorithm A [An] by
Andrews, M1 [Mk1] by McKay, PW1 [PW], PW2 [PW] by Page and Wilson. The delay between the generation of two consecutive partitions in any of these algorithms is $O(n)$ in the worst case (even exclusive of the output). The average delay property was not studied by any of the authors, and we prove for our two new algorithms (which generate partitions in the standard representation) that the property is satisfied.

The same strategy can be used to generate partitions in the multiplicity representation and antilexicographic order. The computation of the next partition from the current one affects at most the two smallest different parts, and creates at most two new different parts. It is possible to perform this update in constant delay per partition (exclusive of the output). Algorithms based on the description are the following: AS [AS] by Akl and Stojmenovic, FL3 [FL3] by Fenner and Loizou, and NW [NW] by Nijenhuis and Wilf.

All known algorithms for generating partitions in lexicographic order use the multiplicity representation. If $c_d > 1$ then one of parts $y_d$ is increased by 1 and $y_d(c_d-1)-1$ parts of size 1 are added. Otherwise one of parts $y_{d-1}$ is increased by 1 and $y_{d-1}(c_{d-1}-1)+y_{d-1}$ parts of size 1 are added. In both cases the part which is increased by 1 may be the same as the previous part or parts; in such cases the multiplicities are corrected. For example, the next partition for $5x3 + 4x3 + 2x3$ is $5x3 + 4x3 + 3x1 + 1x3$ while the next partition for
$5 \times 3 + 4 \times 3 + 2 \times 1$ is $5 \times 4 + 1 \times 9$. This method is due to G. Ehrlich [RND] and is referred to as algorithm E [RND]. Algorithm FL1 [FL] by Fenner and Loizou also belongs to this category. Since the changes are performed only on the last few parts, the method has constant delay property.

Algorithms M2 [Mk] by McKay and W [Wh] by White make use of a procedure that maps an integer between 0 to P(n)-1 into an integer partition. This map is usually called unranking. Algorithm M2 generates unrestricted partitions of $n$ while algorithm W generates doubly restricted partitions of $n$ with $L1 \leq x \leq L2= n$. Using the map, which is a bijection (or one-to-one relationship), it is possible to generate all partitions in various orders. For example, if the mapping is applied from 0 to P(n)-1 one gets partitions in lexicographic order, while the application from P(n)-1 to 0 gives antilexicographic order. To accomplish the mapping, the algorithm must calculate and store all $RP(n', m)$ for $1 \leq n' \leq n$ and $1 \leq m \leq n$. This makes the method ineffective, since $O(RP(n', m)) = O(P(n))$ is exponential in $n$; each $RP(n', m)$ requires $O(\sqrt{n})$ bits to be stored (see Chapter 1).

There exist several algorithms that generate partitions of $n$ into exactly $m$ parts (part order). Algorithm GLW [GLW] by Gupta, Lee and Wong generates restricted partitions in the *multiplicity representation* in lexicographic order. Algorithm RJ1 [RJ] by Riha and
James generates doubly restricted partitions while algorithm RJ2 [RJ] generates multiply restricted partitions (algorithm RJ2 also allows limiting the number of occurrences of each part). Both algorithms generate the partitions in standard representation antilexicographically.

There exists another solution for the case of partitions of \( n \) into exactly \( m \) parts in the standard representation. In [Le and RND] algorithms are presented for generating unrestricted partitions in lexicographic order for each fixed \( m \), but considering the parts of the partition in non-decreasing rather than non-increasing order. In fact, this algorithm was discovered by K.F. Hindenburg in 1778 [RND], and we refer to it as algorithm H. To obtain the next partition from the current one, the elements are scanned from right to left, stopping at the rightmost \( x_i \) such that \( x_m-x_i \geq 2 \). Then \( x_j \) by \( x_i+1 \) is replaced for \( j=i,i+1,\ldots,m-1 \), and \( x_m \) is replaced by the remainder to get the sum \( n \). For example, in the partition 11334, \( i=2 \) and the next partition is 12225. If the multiplicity representation is used, the algorithm is loop-free and works on the last indices only, thus having constant delay property. Note that the parts in algorithm H can easily be reversely indexed to correspond to our conventional notation; the order, however, will be neither lexicographic nor antilexicographic. We coded algorithm H in the multiplicity representation as algorithm Z in chapter IV. When \( m \) varies from 1 to \( n \), all mentioned algorithms generate all unrestricted partitions.
There exists another way of nonlexicographically generating unrestricted partitions in *standard representation*. Algorithm T [T], by Tomasi, simply generates the partitions of \( n \) using exactly \( m \) parts with \( m \) varying from 1 to \( n \). The generation of partitions is accomplished recursively. For a given \( n \), the first partition of size 2 starts with \( \lfloor n/2 \rfloor \) as the first part in the partition; the remainder determines the second part. The next partition is derived by adding 1 to the first part and subtracting 1 from the last part (Gray code theory) and continuing this process. The generation of the partitions is represented in a binary tree form. In order to generate the next partition of exactly the same size of \( m \), the algorithm needs to store all the recursive calls used for generating the children of any particular node. Otherwise, the algorithm will lose track. This makes the memory complexity exponential. The algorithm shows good performance for small \( n \). With 4 Mbytes memory, a memory overflow occurred on testing the algorithm for \( n = 75 \). For \( n = 7 \), the partitions were listed as follows: 7, 43, 331, 322, 2221, 52, 421, 3211, 22111, 61, 511, 4111, 31111, 211111, and 1111111.

Algorithm FL2 [FL] by Fenner and Loizou uses binary tree representation for generating unrestricted partitions in the *multiplicity representation*. By traversing the binary tree \( T(n) \) using in_order (or symmetric) traversal, i.e. left subtree in in_order, root, right subtree in in-order (Knuth, 1973), the partitions are generated in nonlexicographic order, the so-called M-order. For \( n = 6 \), the list of generated partitions shows the following: 222, 2211, 21111, 321, 33, 3111, 42, 411, 51, 6, and 111111 (see Fig. 4).
A binary tree is constructed as follows: Recall $c_1 y_1 + c_2 y_2 + \ldots + c_d y_d = n$.

Let $P_n$ denote the set of all partitions of $n$, and $p$ denote an arbitrary partition in $P_n$.

(If $c_i = 1$, then the corresponding part $y_i$ is said to be simple). Let $T$ be $1^n$.

We now define the following two operations:

A: add one part of size two and remove two parts of size unity.

B: If the smallest part greater than unity is simple then increase it by one and remove one part of size unity.

The partition obtained by applying A to $p$ is denoted by $pA$.

We define a binary tree with partitions as the nodes by letting the left and right sons of any node $p$ be $pA$ and $pB$ respectively, whenever these are defined. If either $pA$ or $pB$ is not defined then the corresponding subtree is empty. If the root of the binary tree is $T$ then it contains every partition in $P_n$. In this case we denote the binary tree by $T(n)$; for example, $T(6)$ is shown in Fig. 4.
[Sa] by Savage discusses the generating of all partitions in a minimal change (or Gray code) order. Strictly speaking there is no well defined algorithm to be implemented, however, for the purpose of this thesis, it will be considered as an algorithm, Sa. Our straightforward analysis led to the following conclusion: the described method requires saving of L(n,k), the list of partitions that generate n using at most k parts. This list must be saved for further use, such as attaching it to another part or parts, or reversing it. In the worst case the algorithm requires L(n/2,n/2) which is in reality P(n/2). The memory complexity is therefore exponential. More effective implementation is a matter for future research.

[Ru2] by Rubin analyses the generation of partitions as follows: Let $S = (s_1, s_2, ..., s_r)$ be a vector of positive integers, and let $M$ be a positive integer. A partition of $S$ is a vector $D = (d_1, d_2, ..., d_r)$ such that $d_i = 0$ or $1$ for $1 \leq i \leq r$. The partition is called an $M$-partition of $S$ if its weight, defined as the inner product $D.S = \sum_i d_i s_i$, is $M$. The integer partitioning problem requires enumeration of all $M$-partitions of $S$ for some fixed $M$ and $S$. Each partition, $D$, is naturally associated with the binary integer $d_1, d_2, ..., d_r = \sum_i d_i 2^{r-i}$. Thus the enumeration of the partition can be regarded as a list of binary integers or $r$-bit string. In order to generate all partitions of $n$, the algorithm Ru requires
\[ S = (n, n-1, n-2, \ldots, \frac{n}{2}, \frac{n}{2}, \frac{n}{2}-1, \frac{n}{2}-1, \ldots, 2, 2, \ldots, 2, 1, 1, \ldots, 1) \]

such that 1 is repeated \( n \) times, and 2 is repeated \( n/2 \) times, etc. For \( M = 6 \),

\[ S = (6, 5, 4, 3, 3, 2, 2, 2, 1, 1, 1, 1, 1, 1). \]

Algorithm Ru generates more general items than partitions and must store the list of partitions named T for further use as storing and push down search operations. This concludes that the algorithm suffers from a memory complexity problem and it can not compete with the existing algorithms.

Procedure We [W] by Wells described the generation of unrestricted partitions of \( n \) in the standard representation antilexicographically which he called "signature form". In fact he gave the same description used by A [An] by Andrews, M1 [Mk1] by McKay, PW1 [PW], and PW2 [PW] by Page and Wilson. The procedure code is very outdated, and the implementation seems unclear. Reworking his code by using the C compiler, we would get a similar algorithm to M1 [Mk1] by Mckay. For the purpose of this thesis we consider it as an algorithm, We.
Procedure Ss [Ss] by Skiena describes the generation of unrestricted partitions of n in the standard representation antilexicographically. The generation of partitions is accomplished recursively. The rules for generating the successor are fairly straightforward and are similar to those already described. If the smallest part $x_1 > 1$, peel off 1 from it, thus increasing the number of parts by 1. If not, find the smallest k such that part $x_k > 1$, and replace parts $x_1, ..., x_k$ by $\lfloor (x_k + k - 1) / (x_k - 1) \rfloor$ copies of $x_k - 1$, with a last element containing remainder. The wrap-around condition occurs when the partition is all ones. The procedure was written in Mathematica language which requires some explanation to be understood. It is also not available on the PC. Compared with the other computer languages (FORTRAN, C, etc.) its performance is slow. In order to have a comparison of algorithm Ss, it should be transformed to C language. However, it then becomes similar to algorithm PW2. For the purpose of this thesis we consider it as an algorithm, Ss.
Chapter III

Implementation of Algorithms

3.1 Introduction

In this section we describe thoroughly the implementation of all existing sequential algorithms. The algorithms were coded in C language (coding in PASCAL was also performed with the comparison yielding results similar to those observed from C language coding). The algorithms were implemented on PC286, Sun, and NeXT workstations in the laboratory of the Computer Science Department, University of Ottawa. The actual CPU times and the average number of arithmetic, logical and assignment instructions per partition of the algorithms are summarized in tables in the following pages. CPU times are measured when algorithms are run without printing partitions. On the Sun and NeXT workstations the CPU time may vary about 1 second due to loading the pages and other shared routines. The CPU time may also depend on the number of jobs running
simultaneously, a factor which could cause more delay. This inaccuracy in measurement does not affect the performance order of the algorithms since it is minimal in the overall running time cost of the algorithm. Such a problem does not exist on the PC because it is a standalone system; however the C compiler on the PC provides the CPU time in seconds only (microsecond measurement is not possible). As a result, the CPU time on a PC may vary about one second. On the NeXT, the execution was repeated several times due to upgrades on Unix which slowed down the performance of all operations on the system. A separate measurement of the number of instructions rather than running time was also performed. There are some differences in instruction count and CPU times. See FL3 vs AS, NW vs Z, PW1 vs RJ1, and RJ2 vs PW2. In instruction count, the corresponding latter algorithms show better performance than the former algorithms. In CPU times however, the former algorithms performed more effectively. Since the instructions are a mixture of costly and inexpensive ones, we conclude that a comparison based upon CPU time is more reliable.

The running times for partitions of the integers 15, 30, 45, 60, and 90 are given in the tables which follow. For some of the slower algorithms the number 90 was excluded.

Due to their exponential memory complexity, algorithms Sa and T were eliminated from the competition. Similarly, algorithm Ru which possesses memory complexity and generates more general items than partitions was eliminated. Because algorithm We
resembles algorithm M1, it was not included in the competition. Because algorithm Ss was written in mathematica language, it was not included in the competition. These restrictions were explained in the previous chapter. Also, since algorithm S generates output with zeroes, we chose to include another version of it called S' in which we tried to shape the output with minimum cost. The output is not in the final form; it is, however, much more accepted as per the other algorithms.

In each algorithm the instructions were categorized as:

a) arithmetic denoted by ct1. exp: /,+,−,∗.

b) while "" "" ct2. "" While (cond).

c) if then "" "" ct3. "" if (cond) in case (cond) is true.

d) if else "" "" ct4. "" if (cond) in case (cond) is false and else clause exists.

e) if unused "" "" ct5. "" if (cond) in case (cond) is false and else clause does not exists.

f) others "" "" ct6. ,, for, assign, goto, ... etc.
(ct1, ct2, ct3, ct4, ct5, and ct6 were used by every program for the same purpose)

example:

```c
if (bb[d-1]==1)
{
    bb[d-1] =2;
    mm[d-1] +=1;
}
else if (mm[d]==2)
{
    bb[d]=2;
}
while (t<pp[t])
{
    t +=1;
}
```

was counted as

<table>
<thead>
<tr>
<th>ins.</th>
<th>op.</th>
<th>#</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (bb[d-1]==1)</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct1</td>
<td>+=1;</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ct3</td>
<td>+=1;</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>bb[d-1] =2;</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mm[d-1] +=1;</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct1</td>
<td>+=3;</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ct6</td>
<td>+=2;</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (mm[d]==2)</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct1</td>
<td>+=1;</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ct3</td>
<td>+=1;</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ct4</td>
<td>+=1;</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>bb[d] =2;</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct6</td>
<td>+=1;</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ct5</td>
<td>+=1;</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
while (t<pp[t])
{
  ct2  +=1;  10
  t    +=1;  7
  ct1  +=1;  11
  ct6  +=1;  12
}

The op#1 means the arithmetic instructions ct1 are increased by 1 because of the minus in the ins#1. The op#2 means the if then instructions ct3 are increase by 1 because the condition in ins#1 is true. The op#4 means the others instructions ct6 are increased by 2 because of the assign instructions in ins#2 and ins#3. The op#9 means the if unused instructions ct5 are increased by 1 because the condition in ins#4 is false. Finally, the op#10 means the while instructions ct2 are increased by 1 because of the while itself in inst#6. Notice that the reference calls of the array index were not counted as separate instructions.
3.2 Antilexicographic Order and Standard Representation

Table 4  Algorithm (M1) by McKay

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>1831</td>
<td>79240</td>
<td>1518808</td>
<td>18816929</td>
<td>175465153</td>
<td>1333745222</td>
</tr>
<tr>
<td>while</td>
<td>507</td>
<td>23024</td>
<td>451500</td>
<td>5672881</td>
<td>53419130</td>
<td>40938375</td>
</tr>
<tr>
<td>if then</td>
<td>209</td>
<td>6450</td>
<td>101047</td>
<td>1083066</td>
<td>9022074</td>
<td>62521988</td>
</tr>
<tr>
<td>if else</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if unused</td>
<td>141</td>
<td>4756</td>
<td>77219</td>
<td>849866</td>
<td>7214452</td>
<td>50746356</td>
</tr>
<tr>
<td>others</td>
<td>2085</td>
<td>86733</td>
<td>1633818</td>
<td>20034646</td>
<td>185515995</td>
<td>1402905462</td>
</tr>
<tr>
<td>totals</td>
<td>4773</td>
<td>200203</td>
<td>3782392</td>
<td>46457386</td>
<td>430636804</td>
<td>3258957403</td>
</tr>
<tr>
<td>inst./part.</td>
<td>27.1</td>
<td>35.7</td>
<td>42.4</td>
<td>48.0</td>
<td>53.0</td>
<td>57.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>&lt; 1</th>
<th>&lt; 1</th>
<th>5.0</th>
<th>55.0</th>
<th>517.0</th>
<th>3916.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>0.0</td>
<td>0.0</td>
<td>0.4</td>
<td>4.9</td>
<td>45.6</td>
<td>348.5</td>
<td></td>
</tr>
<tr>
<td>NeXT</td>
<td>0.0</td>
<td>0.1</td>
<td>2.7</td>
<td>32.8</td>
<td>306.0</td>
<td>2333.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5  Algorithm (PW1) by Page and Wilson

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>2047</td>
<td>85885</td>
<td>1621900</td>
<td>19918042</td>
<td>184612180</td>
<td>1397017642</td>
</tr>
<tr>
<td>while</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if then</td>
<td>507</td>
<td>23024</td>
<td>451500</td>
<td>5672881</td>
<td>53419130</td>
<td>409038375</td>
</tr>
<tr>
<td>if else</td>
<td>175</td>
<td>5603</td>
<td>89133</td>
<td>966466</td>
<td>8118263</td>
<td>56834172</td>
</tr>
<tr>
<td>if unused</td>
<td>508</td>
<td>23025</td>
<td>451501</td>
<td>5672882</td>
<td>53419131</td>
<td>409038376</td>
</tr>
<tr>
<td>others</td>
<td>2909</td>
<td>120117</td>
<td>2251671</td>
<td>27523860</td>
<td>254267841</td>
<td>1919324366</td>
</tr>
<tr>
<td>totals</td>
<td>6146</td>
<td>257651</td>
<td>4865705</td>
<td>59754131</td>
<td>553836545</td>
<td>4191052931</td>
</tr>
<tr>
<td>inst./part.</td>
<td>34.9</td>
<td>45.9</td>
<td>54.5</td>
<td>61.8</td>
<td>66.2</td>
<td>74.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>&lt; 1</th>
<th>&lt; 1</th>
<th>7.0</th>
<th>86.0</th>
<th>800.0</th>
<th>6066.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
<td>6.6</td>
<td>61.2</td>
<td>469.5</td>
<td></td>
</tr>
<tr>
<td>NeXT</td>
<td>0.0</td>
<td>0.1</td>
<td>2.1</td>
<td>26.3</td>
<td>241.9</td>
<td>1812.5</td>
<td></td>
</tr>
</tbody>
</table>

48
Table 6  Algorithm (PW2) by Page and Wilson

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>1523</td>
<td>69074</td>
<td>1354502</td>
<td>17018645</td>
<td>160257392</td>
<td>1227115127</td>
</tr>
<tr>
<td>if then</td>
<td>684</td>
<td>28629</td>
<td>540635</td>
<td>6639349</td>
<td>61537395</td>
<td>465672549</td>
</tr>
<tr>
<td>if else</td>
<td>508</td>
<td>23025</td>
<td>451501</td>
<td>5672882</td>
<td>53419131</td>
<td>409038376</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>2562</td>
<td>108914</td>
<td>2073408</td>
<td>25590931</td>
<td>237031318</td>
<td>1806056025</td>
</tr>
<tr>
<td>totals</td>
<td>5277</td>
<td>229642</td>
<td>4420046</td>
<td>54921807</td>
<td>513245236</td>
<td>3907882077</td>
</tr>
<tr>
<td>inst./part.</td>
<td>29.9</td>
<td>45.9</td>
<td>49.5</td>
<td>56.8</td>
<td>63.2</td>
<td>69.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; 1</td>
<td>1.0</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>25.5</td>
<td>36.9</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>2.9</td>
<td>346.7</td>
</tr>
</tbody>
</table>

Table 7  Algorithm (A) by Andrews

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>7954</td>
<td>400934</td>
<td>8543322</td>
<td>115066417</td>
<td>1149059414</td>
<td>---</td>
</tr>
<tr>
<td>if then</td>
<td>560</td>
<td>31538</td>
<td>689444</td>
<td>9320269</td>
<td>92635761</td>
<td>---</td>
</tr>
<tr>
<td>if else</td>
<td>336</td>
<td>11177</td>
<td>178222</td>
<td>1932873</td>
<td>16236452</td>
<td>---</td>
</tr>
<tr>
<td>if unused</td>
<td>14</td>
<td>29</td>
<td>44</td>
<td>59</td>
<td>74</td>
<td>---</td>
</tr>
<tr>
<td>others</td>
<td>6895</td>
<td>401401</td>
<td>9246307</td>
<td>131095110</td>
<td>1359084944</td>
<td>---</td>
</tr>
<tr>
<td>totals</td>
<td>15759</td>
<td>845079</td>
<td>18657339</td>
<td>257414728</td>
<td>2617016704</td>
<td>---</td>
</tr>
<tr>
<td>inst./part.</td>
<td>89.5</td>
<td>150.7</td>
<td>209.3</td>
<td>266.3</td>
<td>322.3</td>
<td>---</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td>23.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.1</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.3</td>
<td>8.2</td>
</tr>
</tbody>
</table>

|        | 307.0       | 3138.0    |           |
|        | 27.4        | 280.9     |           |
|        | 115.4       | 1183.2    |           |
### 3.3 Antilexicographic Order and Multiplicity Representation

Table 8  Algorithm (NW) by Nijenhuis and Wilf

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>1358</td>
<td>45035</td>
<td>729939</td>
<td>8004570</td>
<td>67778486</td>
<td>475676372</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>607</td>
<td>18887</td>
<td>295317</td>
<td>3168692</td>
<td>26412317</td>
<td>183179012</td>
</tr>
<tr>
<td>if else</td>
<td>445</td>
<td>14733</td>
<td>239483</td>
<td>2630106</td>
<td>22297263</td>
<td>156626022</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>2962</td>
<td>94425</td>
<td>1503781</td>
<td>16315680</td>
<td>137123784</td>
<td>956982628</td>
</tr>
<tr>
<td>totals</td>
<td>5548</td>
<td>178684</td>
<td>2857654</td>
<td>31085515</td>
<td>261730114</td>
<td>1829098207</td>
</tr>
<tr>
<td>inst./part.</td>
<td>31.5</td>
<td>31.8</td>
<td>32.0</td>
<td>32.1</td>
<td>32.2</td>
<td>32.2</td>
</tr>
</tbody>
</table>

| PC286 (sec) | < 1 | < 1 | 3.0 | 39.0 | 326.0 | 2274.0 |
| SUN (sec)    | 0.0 | 0.0 | 0.4 | 4.4  | 37.3  | 263.6  |
| NeXT (sec)   | 0.0 | 0.0 | 1.5 | 16.3 | 136.6 | 957.2  |

Table 9  Algorithm (AS) by Akl and Stojmenovic

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>1316</td>
<td>42949</td>
<td>687615</td>
<td>7482829</td>
<td>63030643</td>
<td>440636822</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>268</td>
<td>9128</td>
<td>150348</td>
<td>1663638</td>
<td>14178998</td>
<td>99991848</td>
</tr>
<tr>
<td>if else</td>
<td>41</td>
<td>1036</td>
<td>13959</td>
<td>134647</td>
<td>1028764</td>
<td>6638248</td>
</tr>
<tr>
<td>if unused</td>
<td>216</td>
<td>6642</td>
<td>103092</td>
<td>1101113</td>
<td>9147027</td>
<td>63272420</td>
</tr>
<tr>
<td>others</td>
<td>1336</td>
<td>43995</td>
<td>715779</td>
<td>7869922</td>
<td>66749721</td>
<td>469038123</td>
</tr>
<tr>
<td>totals</td>
<td>3353</td>
<td>109357</td>
<td>1760127</td>
<td>19218616</td>
<td>162253417</td>
<td>1136211634</td>
</tr>
<tr>
<td>inst./part.</td>
<td>19.0</td>
<td>19.5</td>
<td>19.7</td>
<td>19.8</td>
<td>19.9</td>
<td>20.0</td>
</tr>
</tbody>
</table>

| PC286 (sec) | < 1  | < 1  | 3.0  | 25.0  | 212.0  | 1483.0  |
| SUN (sec)    | 0.0  | 0.0  | 0.3  | 3.6   | 30.9   | 221.6   |
| NeXT (sec)   | 0.0  | 0.0  | 1.4  | 14.9  | 126.9  | 879.4   |
### Table 10 Algorithm (FL3) by Fenner and Loizou

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>2073</td>
<td>70284</td>
<td>1156941</td>
<td>12766033</td>
<td>108703730</td>
<td>765567690</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>350</td>
<td>11206</td>
<td>178266</td>
<td>1932932</td>
<td>1623526</td>
<td>113268344</td>
</tr>
<tr>
<td>if else</td>
<td>175</td>
<td>5603</td>
<td>89133</td>
<td>966466</td>
<td>8118263</td>
<td>56634172</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>1048</td>
<td>33937</td>
<td>542308</td>
<td>5890334</td>
<td>49518287</td>
<td>345562760</td>
</tr>
<tr>
<td>totals</td>
<td>3822</td>
<td>126834</td>
<td>2055782</td>
<td>22532232</td>
<td>190695070</td>
<td>1337667139</td>
</tr>
<tr>
<td>inst./part.</td>
<td>21.7</td>
<td>22.6</td>
<td>23.0</td>
<td>23.3</td>
<td>23.4</td>
<td>23.6</td>
</tr>
</tbody>
</table>

| PC286 (sec) | < 1 | < 1 | 3.0 | 25.0 | 213.0 | 1491.0 |
| SUN (sec)   | 0.0 | 0.0 | 0.3 | 3.3  | 28.3  | 204.1  |
| NeXT (sec)  | 0.0 | 0.0 | 1.4 | 15.5 | 131.2 | 921.0 |

### 3.4 Lexicographic Order and Multiplicity Representation

### Table 11 Algorithm (FL1) by Fenner and Loizou

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>1136</td>
<td>35097</td>
<td>546898</td>
<td>5836435</td>
<td>48427203</td>
<td>334483810</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>296</td>
<td>9969</td>
<td>161843</td>
<td>1777895</td>
<td>15065700</td>
<td>105790159</td>
</tr>
<tr>
<td>if else</td>
<td>312</td>
<td>8865</td>
<td>133390</td>
<td>1391599</td>
<td>11363750</td>
<td>77536141</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>677</td>
<td>21878</td>
<td>350258</td>
<td>3812578</td>
<td>32101906</td>
<td>224324539</td>
</tr>
<tr>
<td>totals</td>
<td>2597</td>
<td>81413</td>
<td>1281523</td>
<td>13784974</td>
<td>115076823</td>
<td>798768822</td>
</tr>
<tr>
<td>inst./part.</td>
<td>14.7</td>
<td>14.5</td>
<td>14.3</td>
<td>14.2</td>
<td>14.1</td>
<td>14.1</td>
</tr>
</tbody>
</table>

| PC286 (sec) | < 1 | < 1 | 2.0 | 21.0 | 174.0 | 1212.0 |
| SUN (sec)   | 0.0 | 0.0 | 0.1 | 1.7  | 14.4  | 100.7  |
| NeXT (sec)  | 0.0 | 0.0 | 1.2 | 12.5 | 106.4 | 742.5  |
Table 12  Algorithm (E) by Ehrlich

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td>1485</td>
<td>47713</td>
<td>765255</td>
<td>8335040</td>
<td>70262206</td>
<td>491472685</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>269</td>
<td>9129</td>
<td>150349</td>
<td>1663639</td>
<td>14178999</td>
<td>99991849</td>
</tr>
<tr>
<td>if else</td>
<td>96</td>
<td>2277</td>
<td>30383</td>
<td>289685</td>
<td>2199591</td>
<td>14116434</td>
</tr>
<tr>
<td>if unused</td>
<td>163</td>
<td>5406</td>
<td>86670</td>
<td>946077</td>
<td>7976202</td>
<td>55794236</td>
</tr>
<tr>
<td>others</td>
<td>1048</td>
<td>32989</td>
<td>525781</td>
<td>5704943</td>
<td>47964952</td>
<td>334846672</td>
</tr>
<tr>
<td>totals</td>
<td>3238</td>
<td>103118</td>
<td>1647572</td>
<td>17905851</td>
<td>150700214</td>
<td>1052856049</td>
</tr>
<tr>
<td>inst./part.</td>
<td>18.4</td>
<td>18.4</td>
<td>18.4</td>
<td>18.5</td>
<td>18.5</td>
<td>18.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>&lt; 1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>0.2</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>22.0</td>
<td>2.5</td>
<td>14.5</td>
</tr>
<tr>
<td></td>
<td>191.0</td>
<td>20.8</td>
<td>121.9</td>
</tr>
<tr>
<td></td>
<td>1327.0</td>
<td>146.9</td>
<td>849.5</td>
</tr>
</tbody>
</table>

Table 13  Algorithm (S) by Stockmal

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td>1150</td>
<td>27530</td>
<td>370026</td>
<td>3631288</td>
<td>28530858</td>
<td>189565100</td>
</tr>
<tr>
<td>while</td>
<td>375</td>
<td>9684</td>
<td>137073</td>
<td>1391055</td>
<td>11191846</td>
<td>75708361</td>
</tr>
<tr>
<td>if then</td>
<td>174</td>
<td>5602</td>
<td>89132</td>
<td>966465</td>
<td>8118262</td>
<td>56634171</td>
</tr>
<tr>
<td>if else</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>1509</td>
<td>38745</td>
<td>548301</td>
<td>5564229</td>
<td>44767393</td>
<td>302833453</td>
</tr>
<tr>
<td>totals</td>
<td>3209</td>
<td>81562</td>
<td>1144533</td>
<td>11553038</td>
<td>92608360</td>
<td>624741086</td>
</tr>
<tr>
<td>inst./part.</td>
<td>18.2</td>
<td>14.5</td>
<td>12.8</td>
<td>11.9</td>
<td>11.4</td>
<td>11.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>&lt; 1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>&lt; 1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>22.0</td>
<td>1.6</td>
<td>5.4</td>
</tr>
<tr>
<td></td>
<td>181.0</td>
<td>13.3</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>1221.0</td>
<td>89.8</td>
<td>292.2</td>
</tr>
</tbody>
</table>
Table 14 Algorithm (S') by Stockmal

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>4266</td>
<td>218613</td>
<td>4832465</td>
<td>67292068</td>
<td>690819637</td>
<td></td>
</tr>
<tr>
<td>while</td>
<td>375</td>
<td>9684</td>
<td>137073</td>
<td>1391055</td>
<td>11191846</td>
<td></td>
</tr>
<tr>
<td>if then</td>
<td>680</td>
<td>28625</td>
<td>540631</td>
<td>6639345</td>
<td>61537391</td>
<td></td>
</tr>
<tr>
<td>if else</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>if unused</td>
<td>2104</td>
<td>145037</td>
<td>3559441</td>
<td>52315020</td>
<td>555450521</td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>4624</td>
<td>229828</td>
<td>5010740</td>
<td>69225009</td>
<td>707056172</td>
<td></td>
</tr>
<tr>
<td>totals</td>
<td>12051</td>
<td>631788</td>
<td>14080351</td>
<td>196862498</td>
<td>2026055568</td>
<td></td>
</tr>
<tr>
<td>inst./part.</td>
<td>68.4</td>
<td>112.7</td>
<td>157.9</td>
<td>203.6</td>
<td>249.5</td>
<td></td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>1.0</td>
<td>31.0</td>
<td>449.0</td>
<td>4670.0</td>
<td></td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.1</td>
<td>1.9</td>
<td>27.4</td>
<td>285.4</td>
<td></td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.3</td>
<td>7.6</td>
<td>103.2</td>
<td>1069.7</td>
<td></td>
</tr>
</tbody>
</table>

3.5 Unranking

Table 15 Algorithm (M2) by McKay

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>5600</td>
<td>285384</td>
<td>6387712</td>
<td>88950808</td>
<td>909113466</td>
<td></td>
</tr>
<tr>
<td>while</td>
<td>2800</td>
<td>173693</td>
<td>4100118</td>
<td>58954426</td>
<td>616987988</td>
<td></td>
</tr>
<tr>
<td>if then</td>
<td>64</td>
<td>240</td>
<td>529</td>
<td>930</td>
<td>1444</td>
<td></td>
</tr>
<tr>
<td>if else</td>
<td>56</td>
<td>225</td>
<td>506</td>
<td>900</td>
<td>1406</td>
<td></td>
</tr>
<tr>
<td>if unused</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>7962</td>
<td>410154</td>
<td>8934407</td>
<td>121831923</td>
<td>1225570900</td>
<td></td>
</tr>
<tr>
<td>totals</td>
<td>16482</td>
<td>869696</td>
<td>19423272</td>
<td>269738987</td>
<td>2751675204</td>
<td></td>
</tr>
<tr>
<td>inst./part.</td>
<td>93.6</td>
<td>155.1</td>
<td>217.9</td>
<td>279.0</td>
<td>338.9</td>
<td></td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>1.0</td>
<td>3.0</td>
<td>51.0</td>
<td>703.0</td>
<td>8086.0</td>
<td></td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.1</td>
<td>2.8</td>
<td>39.4</td>
<td>405.2</td>
<td></td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.6</td>
<td>14.8</td>
<td>210.0</td>
<td>2165.9</td>
<td></td>
</tr>
</tbody>
</table>
### Table 16 Algorithm (W) by White

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>6699</td>
<td>340000</td>
<td>752875</td>
<td>1039409</td>
<td>1055168</td>
<td>1055168</td>
</tr>
<tr>
<td>if then</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>811826</td>
<td></td>
</tr>
<tr>
<td>if else</td>
<td>2705</td>
<td>168361</td>
<td>401156</td>
<td>579889</td>
<td>608871</td>
<td></td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td>2459</td>
<td>2432743</td>
<td>31650839</td>
<td>306290</td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>10638</td>
<td>578340</td>
<td>12945533</td>
<td>17982006</td>
<td>18344408</td>
<td></td>
</tr>
<tr>
<td>totals</td>
<td>22677</td>
<td>1210791</td>
<td>27007718</td>
<td>37437040</td>
<td>3812889</td>
<td></td>
</tr>
<tr>
<td>inst./part.</td>
<td>128.8</td>
<td>216.0</td>
<td>303.0</td>
<td>387.3</td>
<td>469.6</td>
<td></td>
</tr>
<tr>
<td>C PC286</td>
<td>1.0</td>
<td>3.0</td>
<td>61.0</td>
<td>843.0</td>
<td>8508.0</td>
<td></td>
</tr>
<tr>
<td>C SUN</td>
<td>0.0</td>
<td>0.1</td>
<td>3.3</td>
<td>46.4</td>
<td>469.1</td>
<td></td>
</tr>
<tr>
<td>C NeXT</td>
<td>0.0</td>
<td>0.8</td>
<td>17.8</td>
<td>246.1</td>
<td>2492.0</td>
<td></td>
</tr>
</tbody>
</table>

### 3.6 Part Order

### Table 17 Algorithm (GLW) by Gupta, Lee and Wong

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>3757</td>
<td>128238</td>
<td>2059535</td>
<td>22409986</td>
<td>18873785</td>
<td>13192912</td>
</tr>
<tr>
<td>if then</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>811826</td>
<td>56634173</td>
</tr>
<tr>
<td>if else</td>
<td>715</td>
<td>24944</td>
<td>418247</td>
<td>4680904</td>
<td>40195421</td>
<td>285019268</td>
</tr>
<tr>
<td>if unused</td>
<td>195</td>
<td>5170</td>
<td>70168</td>
<td>675517</td>
<td>5160853</td>
<td>33280654</td>
</tr>
<tr>
<td>others</td>
<td>552</td>
<td>16568</td>
<td>250479</td>
<td>2626496</td>
<td>21522332</td>
<td>147299430</td>
</tr>
<tr>
<td>totals</td>
<td>7914</td>
<td>264141</td>
<td>4246789</td>
<td>46287748</td>
<td>390286677</td>
<td>2730444922</td>
</tr>
<tr>
<td>inst./part.</td>
<td>44.9</td>
<td>47.1</td>
<td>47.6</td>
<td>47.8</td>
<td>48.0</td>
<td>48.2</td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>5.0</td>
<td>59.0</td>
<td>500.0</td>
<td>3484.0</td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.6</td>
<td>7.3</td>
<td>61.6</td>
<td>431.1</td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.1</td>
<td>2.7</td>
<td>29.9</td>
<td>253.0</td>
<td>1774.5</td>
</tr>
</tbody>
</table>
Table 18 Algorithm (RJ1) by Riha and James

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>2721</td>
<td>97702</td>
<td>1864543</td>
<td>23298220</td>
<td>218737077</td>
<td>1671630018</td>
</tr>
<tr>
<td>if then</td>
<td>267</td>
<td>11954</td>
<td>232511</td>
<td>2911310</td>
<td>27340644</td>
<td>208951626</td>
</tr>
<tr>
<td>if else</td>
<td>329</td>
<td>14450</td>
<td>277645</td>
<td>3442447</td>
<td>32065476</td>
<td>243369297</td>
</tr>
<tr>
<td>if unused others</td>
<td>2669</td>
<td>104955</td>
<td>2060004</td>
<td>26123812</td>
<td>247729333</td>
<td>1907425497</td>
</tr>
<tr>
<td>totals</td>
<td>5986</td>
<td>229061</td>
<td>4434703</td>
<td>55775789</td>
<td>525872530</td>
<td>4031376640</td>
</tr>
<tr>
<td>inst./part.</td>
<td>34.0</td>
<td>40.8</td>
<td>50.3</td>
<td>57.7</td>
<td>64.7</td>
<td>71.1</td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>6.0</td>
<td>69.0</td>
<td>652.0</td>
<td>4991.0</td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
<td>7.1</td>
<td>67.9</td>
<td>527.9</td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.1</td>
<td>2.2</td>
<td>28.3</td>
<td>266.0</td>
<td>2013.7</td>
</tr>
</tbody>
</table>

Table 19 Algorithm (RJ2) by Riha and James

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic while</td>
<td>3993</td>
<td>137823</td>
<td>2468074</td>
<td>29658155</td>
<td>271312676</td>
<td>2034678117</td>
</tr>
<tr>
<td>if then</td>
<td>324</td>
<td>6485</td>
<td>68370</td>
<td>560206</td>
<td>3864293</td>
<td>23240319</td>
</tr>
<tr>
<td>if else</td>
<td>542</td>
<td>16841</td>
<td>246746</td>
<td>2899460</td>
<td>24354866</td>
<td>169902608</td>
</tr>
<tr>
<td>if unused others</td>
<td>266</td>
<td>11953</td>
<td>232510</td>
<td>2911309</td>
<td>27340643</td>
<td>208951625</td>
</tr>
<tr>
<td>totals</td>
<td>401</td>
<td>12448</td>
<td>233590</td>
<td>2913199</td>
<td>27343568</td>
<td>208955810</td>
</tr>
<tr>
<td>inst./part.</td>
<td>5954</td>
<td>192659</td>
<td>3437601</td>
<td>41299408</td>
<td>377701552</td>
<td>2831452796</td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>9.0</td>
<td>109.0</td>
<td>997.0</td>
<td>7437.0</td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.7</td>
<td>9.2</td>
<td>84.9</td>
<td>634.2</td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.2</td>
<td>3.8</td>
<td>45.5</td>
<td>416.5</td>
<td>3092.1</td>
</tr>
</tbody>
</table>

55
### 3.7 Non Lexicographic Order

Table 20  Algorithm (FL2) by Fenner and Loizou

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>1414</td>
<td>45024</td>
<td>710420</td>
<td>7650661</td>
<td>63884399</td>
<td>443411582</td>
</tr>
<tr>
<td>while</td>
<td>176</td>
<td>5604</td>
<td>89134</td>
<td>966467</td>
<td>8118264</td>
<td>56634173</td>
</tr>
<tr>
<td>if then</td>
<td>360</td>
<td>11247</td>
<td>173548</td>
<td>1847257</td>
<td>15279598</td>
<td>105312145</td>
</tr>
<tr>
<td>if else</td>
<td>355</td>
<td>11451</td>
<td>182986</td>
<td>1986684</td>
<td>16696247</td>
<td>116475245</td>
</tr>
<tr>
<td>if unused</td>
<td>111</td>
<td>3128</td>
<td>49270</td>
<td>525320</td>
<td>4377983</td>
<td>30327411</td>
</tr>
<tr>
<td>others</td>
<td>1046</td>
<td>33364</td>
<td>526734</td>
<td>5687480</td>
<td>47584683</td>
<td>330909305</td>
</tr>
<tr>
<td>totals</td>
<td>3462</td>
<td>109818</td>
<td>1732092</td>
<td>18663869</td>
<td>155941179</td>
<td>1083069861</td>
</tr>
<tr>
<td>inst./part.</td>
<td>19.6</td>
<td>19.5</td>
<td>19.4</td>
<td>19.3</td>
<td>19.2</td>
<td>19.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; 1</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>23.0</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>245.0</td>
<td>17.0</td>
<td>15.7</td>
</tr>
<tr>
<td></td>
<td>1700.0</td>
<td>120.9</td>
<td>920.3</td>
</tr>
</tbody>
</table>

Table 21  Algorithm (H) by Hindenburg

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>3269</td>
<td>136399</td>
<td>2651294</td>
<td>33616178</td>
<td>320638315</td>
<td>2488138501</td>
</tr>
<tr>
<td>while</td>
<td>390</td>
<td>9714</td>
<td>137118</td>
<td>1391115</td>
<td>11191921</td>
<td>75708451</td>
</tr>
<tr>
<td>if then</td>
<td>161</td>
<td>5574</td>
<td>89089</td>
<td>966407</td>
<td>8118189</td>
<td>56634083</td>
</tr>
<tr>
<td>if else</td>
<td>15</td>
<td>30</td>
<td>45</td>
<td>60</td>
<td>75</td>
<td>90</td>
</tr>
<tr>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>3356</td>
<td>138363</td>
<td>2693384</td>
<td>34159832</td>
<td>325685777</td>
<td>2525702496</td>
</tr>
<tr>
<td>totals</td>
<td>7191</td>
<td>290080</td>
<td>5570830</td>
<td>70133592</td>
<td>665634277</td>
<td>5146183621</td>
</tr>
<tr>
<td>inst./part.</td>
<td>40.8</td>
<td>51.7</td>
<td>62.5</td>
<td>72.5</td>
<td>81.9</td>
<td>90.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>8.0</td>
<td>0.6</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>95.0</td>
<td>8.0</td>
<td>32.7</td>
</tr>
<tr>
<td></td>
<td>892.0</td>
<td>75.1</td>
<td>307.5</td>
</tr>
<tr>
<td></td>
<td>6830.0</td>
<td>583.4</td>
<td>2343.2</td>
</tr>
</tbody>
</table>
3.8 Modifications to Existing Algorithms

Algorithms that were originally written in an outdated fashion (for example, using GOTO instructions or the FORTRAN language) are re-coded in C language and compared in modified form. The modified form was slightly faster than the original one; in some algorithms the improvement was about 10% compared with the original implementation of the algorithm. In particular some remarkable changes occurred on the M1, M2, GLW, and S algorithms. Algorithm M1 tests at the end of each run if m=0 etc... Yet m never is zero! The entire portion therefore is meaningless and was eliminated in the modified version. Algorithm GLW was coded with some errors. This algorithm generates doubly restricted partitions of an integer n into exactly m parts. The condition of ending the computation for given m was incorrect and incomplete (other conditions were added). A remarkable amount of time was required to fix it (see program GLW).
CHAPTER IV

New Sequential Algorithms

This chapter is concerned with our major contributions in the domain of generating integer partitions and is organized into five sections. Section (1) describes a new algorithm, ZS1, for generating integer partitions in standard representation and antilexicographic order. In section (2) we describe a new algorithm, ZS2, for generating integer partitions in standard representation and lexicographic order. Section (3) proves that both algorithms ZS1 and ZS2 have constant average delay property and in section (4) we prove that both algorithms remain cost-effective when n increases. Finally, in section (5) we describe a new algorithm Z in the multiplicity representation and nonlexicographic order.
4.1 Algorithm ZS1

The algorithm ZS1 generates partitions in antilexicographic order. Recall that h is the index of the last part of the partition which is greater than 1 while m is the number of parts. The central feature of algorithm ZS1 comes from observing the distribution of $x_n$. A study shows that most partitions contain at least one part of size 2. More precisely, $x_n=2$ has increasing frequency; it appears in 66% of cases for $n=30$ and in 78% of partitions for $n=90$ and appears to be increasing with n. We will show in section 4.4 that the number of partitions that contain at least one part of size 2 is equal to $P(n-2)$. This special case is treated separately, and we will prove that it is sufficient to argue the constant average delay of algorithm ZS1. The case $x_n>2$ is coded in a manner similar to the earlier algorithms, except that the assignment of parts which are supposed to receive the value 1 is avoided by, first, an initialization step that assigns 1 to each part and, second, the observation that inactive parts (these with index $> m$) are always left with the value 1. For any $x_n = 2$, the next partition will be generated by setting $x_n$ and $x_{m+1}$ to 1. Any attempt to treat the case of $x_n = 3, \ldots, n$ in a similar way will be more costly than the proposed one in ZS1. Consider the case of $x_n = 3$, which consists of the majority of the remaining $P(n) - P(n-2)$ cases. More precisely, for $n = 15, 30, 45, 60, 75$ and 90, it appears respectively in 47%, 56%, 61%, 65%, 68% and 70% of the remaining cases and appears to be increasing with n. The next partition will be reached by distributing the
remainder \( t \) as quickly as possible. That necessitates starting from \( x_h \) and changing \( \lfloor t/2 \rfloor \) parts to size 2. This operation is performed by the algorithm without any remarkable loss of time. Before entering the loop the calculation of the remainder and the new part to be distributed is performed. Then \( x_h \) is replaced by the new part. This replacement is very important because it eliminates the use of the iteration for \( h=m \) cases. In each iteration the algorithm uses 5 inexpensive statements (2 arithmetic statements and 3 assign statements).

To the best of our knowledge, this is the minimum number of statements required to calculate the next part by an algorithm using iterations. Other algorithms require more statements, some of which are expensive. See algorithms A, M1, PW1, PW2, We. The only improvement that could be achieved would be to force the algorithm to use 2 as a constant instead of deriving it as \( (x_h-1) \). This calculation is performed only once before entering the loop. To segregate the case of \( x_h = 3, \ldots, n \) from the others, however, the algorithm needs to use the expensive if statement \( P(n) - P(n-2) \) times. Also, this does not eliminate the use of the iteration which is needed to change \( \lfloor t/2 \rfloor \) parts to 2. This makes any attempt more costly. For example the next partition for 5 5 3 1 1 1 1 1 1 is 5 5 2 2 2 2 2 1, and for 5 5 4 4 3 is 5 5 4 4 2 1. In the first example the use of the iteration is a must. In the second example \( (h=m) \), however, the change is performed directly by the algorithm. Compared with other algorithms that generate partitions in the standard representation, this algorithm is expected to remain at least four times faster for any size of \( n \). For "big" \( n \), the special cases will represent almost all the cases; each partition requires 2 arithmetic, 3 assign, and one conditional statement (total 6). On the
other hand, the other algorithms use 4 arithmetic, 6 assign, and 4 conditional statements before starting the iteration routine. Each iteration costs 2 arithmetic, 3 assign, and 1 conditional statement. By using the iteration 3 times (for n=75, the average number of iterations is 5.58), a total of 26 statements are required. As a result algorithm ZS1 proves to be at least 4 times faster.

Algorithm ZS1.
1-   for i←1 to n do x_i←1;
2-   x_i←n; m←1; h←1; output x_i;
3-   while x_i≠1 do {
4-       if x_h=2 then { m←m+1;
5-           x_h←1;
6-       } h←h-1 }
7-   else { r←x_h-1;
8-       t←m-h+1;
9-       x_h←r;
10-      while t ≥ r do { h←h+1;
11-          x_h←r;
12-          t←t-r }
13-     if t=0 then m←h

61
else { m←h+1;

if t>1 then { h←h+1;
    x_n←t }}

for i←1 to m do output x_i;

Lines 1 and 2 indicate the initializations and the output before entering the while loop. Lines 4 - 6 indicate the treatment of the special cases. Lines 7 - 16 indicate the treatment of the other cases left; t represents the remainder to be distributed which is the difference between h and m; r represents the new calculated part to be distributed. Line 17 shows the output of each case.

Table 22  Algorithm (ZS1) by Zoghbi and Stojmenovic

<table>
<thead>
<tr>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>658</td>
<td>21372</td>
<td>342572</td>
<td>3731216</td>
<td>31444287</td>
<td>219898439</td>
</tr>
<tr>
<td>while</td>
<td>255</td>
<td>8930</td>
<td>147884</td>
<td>1643248</td>
<td>14036936</td>
<td>99151911</td>
</tr>
<tr>
<td>if then</td>
<td>175</td>
<td>5345</td>
<td>84193</td>
<td>908694</td>
<td>7623530</td>
<td>53176663</td>
</tr>
<tr>
<td>if else</td>
<td>94</td>
<td>2533</td>
<td>35321</td>
<td>347455</td>
<td>2694322</td>
<td>17573941</td>
</tr>
<tr>
<td>if unused</td>
<td>34</td>
<td>847</td>
<td>11914</td>
<td>116600</td>
<td>903811</td>
<td>5887816</td>
</tr>
<tr>
<td>others</td>
<td>885</td>
<td>29461</td>
<td>478548</td>
<td>5257870</td>
<td>44577518</td>
<td>313162540</td>
</tr>
<tr>
<td>totals</td>
<td>2101</td>
<td>68488</td>
<td>1100432</td>
<td>12005083</td>
<td>101280304</td>
<td>708851310</td>
</tr>
<tr>
<td>inst./part.</td>
<td>11.9</td>
<td>12.2</td>
<td>12.3</td>
<td>12.4</td>
<td>12.4</td>
<td>12.5</td>
</tr>
<tr>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>1.0</td>
<td>13.0</td>
<td>110.0</td>
<td>765.0</td>
</tr>
<tr>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>1.1</td>
<td>9.1</td>
<td>64.6</td>
</tr>
<tr>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.4</td>
<td>4.9</td>
<td>41.2</td>
<td>285.7</td>
</tr>
</tbody>
</table>
4.2 Algorithm ZS2

We now describe the method for generating partitions in lexicographic order and standard representation of partitions. From our observations in the previous section, the rightmost part of size 2 was replaced by two parts of size 1. In the lexicographic order, which is the reverse order, we can simply conclude that the first two part of size 1 can be replaced by one part of size 2. That means that the number of the special cases is equal to P(n-2). It was empirically observed that in 66% of cases for n=30 and 78% of cases for n=90 there exist at least two parts of size 1 in a given partition (i.e. m-h>1). The coding of the special case is made simpler, in fact with constant delay, by replacing first two parts of size 1 by one part of size 2. The position h of last part >1 is always maintained. Otherwise, to find the next partition in lexicographic order, an algorithm will do a backward search to find the first part that can be increased. The last part x_m cannot be increased. The next-to-last part x_{m-1} can be increased only if x_{m-2} > x_{m-1}. The element which will be increased is x_j where x_{j+1} > x_j and x_j = x_{j+1} = ... = x_{m-1}. The j-th part becomes x_{j+1}, h receives the value j, and an appropriate number of parts equal to 1 are added to complete the sum to n. In the previous section, the special case was known by x_h = 2 and we have seen that any segregation of x_h > 2 was not feasible. In this section, any attempt to test the value of x_h, i.e. x_h = 2 is not worthwhile since the same process will occur for any value of x_h. What in fact makes sense is the value of x_{h-1} as it is described above. For example,
in the partition 5 5 5 4 4 4 1 the leftmost 4 is increased, and the next partition is 5 5 5 5 1 1 1 1 1 1. In the partition 5 2 1, the next partition is 5 3, simply because the increase of 2 by 1 does not affect the 5.

Algorithm ZS2.

1- for i←1 to n do x_i←1; output x_i, i=1,2,..., n;

2- x_0←1; x_1←2; h←1; m←n-1; output x_i, i=1,2,..., m;

3- while x_i≠ n do {

4- if m-h>1 then { h←h+1;

5- x_h←2;

6- m←m-1 }

7- else { j←m-2;

8- while x_j = x_{m-1} do { x_j←i;

9- j←j-1 }

10- h←j+1;

11- x_h←x_{m-1}+1;

12- r←x_m+x_{m-1}(m-h-1);

13- x_m←1;

14- if m-h>1 then { x_{m-1}←1 }

15- m←h+r-1 }

64
for \( i \leftarrow 1 \) to \( m \) do output \( x_i \) }

Lines 1 and 2 indicate the initialization and output of the first two partitions before entering the while loop. Lines 4 - 6 indicate the treatment of the special cases. Lines 7 - 15 indicate the treatment of the remaining cases left; \( j \) represents the index of the part that could be changed; \( r \) represents the remainder to be distributed. Line 16 indicates the output of each case.

Table 23 Algorithm (ZS2) by Zoghbi and Stejmenovic

<table>
<thead>
<tr>
<th>( n )</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>938</td>
<td>29704</td>
<td>469488</td>
<td>5065525</td>
<td>42398932</td>
<td>294946487</td>
</tr>
<tr>
<td>while</td>
<td>234</td>
<td>8281</td>
<td>138434</td>
<td>1547038</td>
<td>13275187</td>
<td>94104032</td>
</tr>
<tr>
<td>if then</td>
<td>133</td>
<td>4804</td>
<td>80024</td>
<td>888881</td>
<td>7580397</td>
<td>53433160</td>
</tr>
<tr>
<td>if else</td>
<td>74</td>
<td>1885</td>
<td>25872</td>
<td>251246</td>
<td>1932574</td>
<td>12526063</td>
</tr>
<tr>
<td>if unused</td>
<td>41</td>
<td>798</td>
<td>9108</td>
<td>77584</td>
<td>537865</td>
<td>3201011</td>
</tr>
<tr>
<td>others</td>
<td>1106</td>
<td>35356</td>
<td>561427</td>
<td>6071007</td>
<td>50883141</td>
<td>354267464</td>
</tr>
<tr>
<td>totals</td>
<td>2526</td>
<td>80828</td>
<td>1284353</td>
<td>13901281</td>
<td>116608096</td>
<td>812478217</td>
</tr>
<tr>
<td>inst./part.</td>
<td>14.3</td>
<td>14.4</td>
<td>14.4</td>
<td>14.3</td>
<td>14.3</td>
<td>14.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>PC286 (sec)</th>
<th>SUN (sec)</th>
<th>NeXT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; 1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>15.0</td>
<td>1.2</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>124.0</td>
<td>10.3</td>
<td>44.7</td>
</tr>
<tr>
<td></td>
<td>859.0</td>
<td>71.7</td>
<td>311.0</td>
</tr>
</tbody>
</table>
4.3 Constant Average Delay Property for ZS1 and ZS2

The output size of each of P(n) partitions is O(n). This means that the total output size is O(nP(n)). In some applications, however, the objects which are generated do not need to be printed out, for they merely serve as the source of information for other procedures that work on combinatorial objects and check some criteria which may be verified without always looking at the whole new object. It is therefore worthwhile to consider generating combinatorial objects without outputting them. Optimal algorithms in this sense work in O(P(n)) time, i.e. in constant time per partition. Algorithms that generate the next object from the current one with constant average delay (exclusive of the output time) exist for various kinds of combinatorial objects [BH, BS, Ru1, Ha, VB, W, ZR]. To the best of our knowledge, none of the existing algorithms for generating integer partitions in standard representation has the constant average delay property. Our implementation in the previous section shows that the average number of instructions per partition in the standard representation form in all algorithms is increasing with n. For n = 30 and 75 the average number of instructions was respectively (M1,35,53), (PW1,46,66), (RJ1,40,64), (H,51,81), (RJ2,67,90), (PW2,50,63), (M2,155,338), and (W,216,469). The increase is between 50% to 100% when n almost doubled. The average number of instructions used to generate the partitions in the multiplicity representation form remains constant for any size of n, and it is remarkably lower than the one in the standard representation. In this
section we prove that algorithms ZS1 and ZS2 have the constant delay property. We need the following lemma in our proof.

**Lemma 1.** $\text{RP}(n,L2) \geq n^2/12$ for $L2 \geq 3$.

**Proof.** Since $\text{RP}(n,L2) \geq \text{RP}(n,3)$ for $L2 > 3$, it is sufficient to prove $\text{RP}(n,3) \geq n^2/12$. In the *multiplicity representation*, the partitions in $\text{RP}(n,3)$ are of the following kind: $n = 3c_1 + 2c_2 + c_3$ (i.e. $y_1 = 3$, $y_2 = 2$, $y_3 = 1$). The number of such partitions is equal to the number of solutions of the above equation. Clearly $0 \leq c_1 \leq \lfloor n/3 \rfloor$. For each $c_1$ in the interval we solve the equation $2c_2 + c_3 = n - 3c_1$. This equation has a unique solution $c_3$ for each $c_2$, $0 \leq c_2 \leq (n - 3c_1)/2$. Therefore, for fixed $c_1$, the number of solutions is $\lfloor (n - 3c_1)/2 \rfloor + 1 \geq (n - 3c_1)/2$. Taking all values of $c_1$ into account, the number of solutions is $\geq n/2 + (n-3)/2 + (n-6)/2 + \ldots + (n-3\lfloor n/3 \rfloor)/2 = (n/3) + 1)n/2 - \lfloor n/3 \rfloor + 1)/3/4 = (n/3) + 1)(n/2 - n/3 + n/3)/3/4 \geq (n/3) + 1)(n/2 - n/4) \geq n^2/12$.

**Theorem 1.** Algorithms ZS1 and ZS2 generate unrestricted integer partitions in *standard representation* with constant average delay, exclusive of the output.

**Proof.** Consider part $x_i \geq 3$ in the current partition. It received its value after a backtracking search (starting from the last part) was performed to find an index $j \leq i$, called
the turning point, that should change its value by 1 (increase/decrease for lexicographic/antilexicographic order) and to update values $x_i$ for $j \leq i$. The time to perform both backtracking searches is $O(r_j)$ where $r_j = n - x_1 - x_2 - ... - x_j$ is the remainder to distribute after first $j$ parts are fixed. We decided to change the cost of the backtrack search evenly to all “swept” parts, such that each of them receives constant $O(1)$ time. Part $x_i$ will be changed only after a similar backtracking step “swept” over $i$-th part or recognized $i$-th part as the turning point (note that $i$-th part is the turning point in at least one of the two backtracking steps). There are $RP(r_i, x_i)$ such partitions which all keep $x_i$ intact. For $x_i \geq 3$ the number of such partitions, according to Lemma 1, is $\geq r_i^2/12$. Therefore the average number of operations that are performed by such part $i$ during the “run” of $RP(r_i, x_i)$, including the change of its value, is $O(1)/RP(r_i, x_i) \leq O(1)/r_i^2 = O(1/r_i^2) < q_i/r_i^2$ where $q_i$ is a constant. Thus the average number of operations for all parts of size $\geq 3$ is $\leq q_1/r_1^2 + q_2/r_2^2 + ... + q_t/r_t^2 \leq q(1/r_1^2 + ... + 1/r_t^2) < q(1/n^2 + 1/(n-1)^2 + ... + 1/1^2) < 2q$ (the last inequality can be obtained easily by applying integral operation on the last sum), which is a constant. The case in which $x_i \leq 2$ was not considered. In this case, however, both algorithms ZS1 and ZS2 perform a constant number of steps per partition on all such parts. Therefore the algorithms ZS1 and ZS2 have overall constant time average delay.
4.4 Number of Partitions Containing 2

In this section our focus is to prove that the number of special cases treated by ZS1 and ZS2 is increasing when \( n \) is increasing. In fact in both algorithms the number of the special cases is \( P(n-2) \). In algorithm ZS1 the partition is considered as a special case if and only if it contains at least one part of size 2. In algorithm ZS2, however, the partition is considered as special case if and only if it contains at least two parts of size 1. Given any unrestricted partition of \( n-2 \), by attaching 2 to it we get a partition of \( n \). Thus the number of partitions that contain the part 2 for any given \( n \) is equal to \( P(n-2) \) where \( P(n) \) is the number of unrestricted partitions of \( n \). Referring to Table 3 in Chapter I, Table 24 below shows the number of \( P(n-2) \) with correspondance to \( P(n) \) and the ratio expressed as a percentage. Due to memory limitations, the maximum possible number of \( n \) for computation was 510. As it was mentioned before, this is due to using two dimensional array of size \( n \).

The following table illustrates the number of partitions that contain 2.
The table indicates that the average $P(n-2)/P(n)$ is increasing when $n$ is increasing.
Theorem 2.

\[ \lim_{n \to \infty} \frac{p(n-2)}{p(n)} = 1 \]

Proof. Consider the cardinality of \( p(n) \) in chapter (I) [HR].

\[ p(n) = \frac{1}{4n^{\frac{2}{3}}} e^{\sqrt{\frac{a}{3}}} + C_n n^{-\frac{1}{2}} \quad \text{where} \quad C_n = O(1). \]

Let

\[ A(n) = \frac{1}{4n^{\frac{2}{3}}} e^{\sqrt{\frac{a}{3}}} \]

\[ \frac{p(n-2)}{p(n)} = \frac{A(n-2) + C_{n-2} (n-2)^{-\frac{1}{2}}}{A(n) + C_n n^{-\frac{1}{2}}} \]

\[ = \frac{A(n-2)}{A(n)} \frac{1 + C_{n-2} (n-2)^{-\frac{1}{2}}}{1 + C_n n^{-\frac{1}{2}}} \]

now,

\[ \frac{A(n-2)}{A(n)} = \frac{n-2}{n} e^{\sqrt{\frac{a}{3}} - \sqrt{\frac{a}{3}}} \]
\[ \frac{n-2}{n} e^{\pi \sqrt{\frac{2}{3} (\sqrt{n^2 - 2} - \sqrt{n})}} \]

\[ = \frac{n-2}{n} e^{\pi \sqrt{\frac{2}{3} (\sqrt{n^2 - 2} + \sqrt{n}) (\sqrt{n^2 - 2} - \sqrt{n})}} \]

\[ = \frac{n-2}{n} e^{\pi \sqrt{\frac{2}{3} \left( \frac{n^2 - 2 - n}{\sqrt{n^2 - 2} + \sqrt{n}} \right)}} \]

\[ = \frac{n-2}{n} e^{\left( \frac{-2\pi \sqrt{\frac{2}{3}}}{\sqrt{n^2 - 2} + \sqrt{n}} \right)} \]

\[ = (1 - \frac{2}{n}) \frac{1}{e^{\left( \frac{-2\pi \sqrt{\frac{2}{3}}}{\sqrt{n^2 - 2} + \sqrt{n}} \right)}} \]

but when \( n \to \infty \), we get

\[ \left( \frac{-2\pi \sqrt{\frac{2}{3}}}{\sqrt{n^2 - 2} + \sqrt{n}} \right) \to 0. \]
Let us analyse the other part of the equation.

When \( n \to \infty \), we get:

\[
C_n \frac{n^{-\frac{1}{2}}}{\sqrt{n}A(n)} = \frac{C_n}{\sqrt{n}A(n)} \to 0.
\]

and

\[
C_{n-2} \frac{(n-2)^{-\frac{1}{2}}}{A(n)} = \frac{C_{n-2}}{\sqrt{n-2}A(n)} \to 0.
\]

so,

\[
\frac{P(n-2)}{P(n)} = \frac{1+0}{1+0} = 1.
\]

as a result

\[
\lim_{n \to \infty} \frac{P(n-2)}{P(n)} = 1.
\]

73
For every tested $n$, the number of special cases in both algorithms ZS1 and ZS2 was \( P(n-2) \). As mentioned before, these special cases were treated quickly without any iterations. Let us consider ZS1 as a reference for discussions in the remaining parts of this Chapter.

In *standard representation* other algorithms use only the method described in lines 7 - 16 of ZS1. In particular, algorithm M1 was shown to have the best performance before algorithm ZS1 was proposed. According to table 24, for $n = 60, 75$, and $90$, the special cases were 74%, 76%, and 78% respectively. Comparing the CPU time between ZS1 and M1 we get 24%, 22%, and 18% on PC286, 21%, 20%, and 16% on SUN, and 15%, 13%, and 12% on NeXT. This shows that ZS1 treats the special cases in nonremarkable time, and that when $n$ increases ZS1 will show better improvement in comparison with M1.

In *multiplicity representation*, algorithm FL1 was shown to have the best performance before algorithm ZS1 was proposed. For $n = 60, 75, 90$, by comparing the CPU time between ZS1 and FL1 we get approximately 61% on PC286, 64% on SUN, and 43% on NeXT. The percentage on each machine was constant. The number of instructions per partition has shown unremarkable changes in each of them. By proving that ZS1 has
constant average delay property, and by showing that \( P(n-2) \) is increasing whenever \( n \) is increasing, we can conclude that the CPU time percentage will remain the same whenever \( n \) increases.

4.5 Algorithm Z

The algorithm Z is the transformation of the algorithm H first discovered by Hindenburg. The parts of the partition are in non-decreasing order. While every algorithm in the multiplicity representation generates the partition in the form \( c_1 y_1 + \ldots + c_d y_d = n \) with \( y_1 \geq \ldots \geq y_d \), this algorithm generates the partitions with \( y_1 < \ldots < y_d \). Using exactly \( m \) parts to generate the next partition of \( n \), this algorithm searches backward \( y_i \) such that \( y_d - y_i \geq 2 \). If \( c_i = 1 \) then \( y_i \) is increased by 1, otherwise \( c_i \) is decreased by 1 and a new part is added. In either case the appropriate multiplicity is calculated. Next, the remainder is examined: if it is equal to the actual part \( y_i \), then the multiplicity \( c_i \) is updated. Otherwise a new part equal to the remainder is added. When \( m \) varies from 1 to \( n \), the algorithm generates all partitions of \( n \); \( m \) is changed automatically by the algorithm. For example, the partitions that are generated after 1X2 2X1 4X1, are 1X2 3X2, 1X1 2X2 3X1, 2X4, 1X4 4X1 etc. Following is a code of the algorithm Z.
Algorithm Z

1- for i←1 to n do c_i=y_i=0;
2- y_0←1; c_0←1; y_i←n; c_i←1; h←1;
3- while c_i≠n do {
4- for i←1; to h do output c_i; y_i;
5- i←h-1;
6- k←c_h;
7- r←c_h*y_h
8- while (y_h-y_i)<2 do {
9- k←k+c_i;
10- r←r+c_i*y_i;
11- i←i-1;
12- if c_i=1 then if i≠0
13- then { r←r+c_i*y_i;
14- y_i←y_i+1 }
15- else { i←1;
16- y_i←1 }
17- else { c_i←c_i-1;
18- r←r+y_i;
19- \[ i \leftarrow i + 1; \]
20- \[ y_i \leftarrow y_{i+1}; \]
21- \[ c_i \leftarrow k; \]
22- \[ r \leftarrow r - c_i * y_i; \]
23- \[ h \leftarrow i + 1 \]
24- \[ \text{if } r = y_i \text{ then } \{ c_i \leftarrow c_i + 1; \]
25- \[ h \leftarrow i \} \]
26- \[ \text{else } \{ y_h \leftarrow r; \]
27- \[ c_h \leftarrow 1 \} \}

Lines 1 and 2 describe the initialization. Line 4 describes the output of each generated partition. In line 6 k represents the calculated part. In line 7 r represents the calculated remainder. Lines 8 - 11 describe the calculation of the remainder. Lines 12 - 16 describe the update of the current part in the case where the current part is equal to 1, and lines 17 - 20 describe the other case (current part differs from 1). Lines 21 - 23 describe the calculation of the next parts based on the remainder left from the previous update. Lines 24 - 25 describe the update of the last modified part in case the actual part is equal to the remainder. Lines 26 - 27 describe the case of adding a new part with multiplicity equal to 1 in case the remainder is not equal to the last part.
Table 25  Algorithm (Z) by Zoghbi

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
<th>75</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>arithmetic</td>
<td>1831</td>
<td>56455</td>
<td>869682</td>
<td>9220633</td>
<td>76227808</td>
<td>525334970</td>
</tr>
<tr>
<td></td>
<td>while</td>
<td>229</td>
<td>6840</td>
<td>105556</td>
<td>1121503</td>
<td>9289089</td>
<td>64112357</td>
</tr>
<tr>
<td></td>
<td>if then</td>
<td>215</td>
<td>7722</td>
<td>131505</td>
<td>1488270</td>
<td>12866184</td>
<td>91673815</td>
</tr>
<tr>
<td></td>
<td>if else</td>
<td>229</td>
<td>6840</td>
<td>105556</td>
<td>1121503</td>
<td>9289089</td>
<td>64112357</td>
</tr>
<tr>
<td></td>
<td>if unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>others</td>
<td>2083</td>
<td>64244</td>
<td>1001284</td>
<td>10709030</td>
<td>89094149</td>
<td>617008972</td>
</tr>
<tr>
<td></td>
<td>totals</td>
<td>4587</td>
<td>142101</td>
<td>2213583</td>
<td>23660939</td>
<td>196766319</td>
<td>1362242471</td>
</tr>
<tr>
<td></td>
<td>inst./part.</td>
<td>26.0</td>
<td>25.3</td>
<td>24.8</td>
<td>24.4</td>
<td>24.2</td>
<td>24.0</td>
</tr>
<tr>
<td></td>
<td>PC286 (sec)</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>4.0</td>
<td>38.0</td>
<td>319.0</td>
<td>2218.0</td>
</tr>
<tr>
<td></td>
<td>SUN (sec)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
<td>4.7</td>
<td>39.0</td>
<td>274.2</td>
</tr>
<tr>
<td></td>
<td>NeXT (sec)</td>
<td>0.0</td>
<td>0.1</td>
<td>2.1</td>
<td>22.9</td>
<td>193.5</td>
<td>1344.5</td>
</tr>
</tbody>
</table>
Chapter V

Analysis

In this chapter we present the results of the performance evaluation of known methods of generating integer partitions. The comparison includes our newly proposed algorithms. All algorithms are divided into two groups according to representation - *multiplicity* or *standard*. A detailed comparison is performed for unrestricted and restricted partitions (algorithms that generate doubly restricted partitions are also compared here for $L_1=1$ and $L_2=n$).

The CPU time in tables below are in seconds. The names of algorithms are defined earlier, and the orders of generations are abbreviated as follows: A (antilexicographic), L (lexicographic), P(part order), U (unranking) and N (none of them). The tables refer to the partition of the integer 75.
For all algorithms, the SUN workstation exhibited better performance than the NeXT workstation, which in turn exhibited better performance than the PC286.

The results show clearly that both algorithms ZS1 and ZS2 are superior to all other known algorithms (M1, RJ1, PW1, H, RJ2, PW2, M2, W, A) that generate partitions in the standard representation. While their speed was comparable to each other, each of them was at least four times faster on any of three machines when partitions of the integer 75 were generated. Moreover, both algorithms are faster than any algorithm for generating integer partitions in the multiplicity representation.

Among algorithms that generate partitions in the multiplicity representation (as defined), algorithm FL1 was fastest on all three machines, and for n=75 was between about 10% and 100% faster than other algorithms E, FL2, AS, FL3, NW, Z while GLW proved inefficient compared with the other algorithms. Algorithm S was faster than algorithm FL1 and is included in the group since its combined representation is closer to multiplicity than to standard representation. But, as we have mentioned in Chapter II, the output in S is not ready to be used as in all of the other algorithms. By trying to eliminate the parts that have zero multiplicities, this algorithm S' demonstrates poor performance compared with the original algorithm. The elimination was performed with an average linear time delay.
<table>
<thead>
<tr>
<th>ALG.</th>
<th>rep.</th>
<th>ord.</th>
<th>inst. per part.</th>
<th>time on pc_286</th>
<th>time on SUN</th>
<th>time on NeXT</th>
<th>time com.</th>
<th>memory com.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZS1</td>
<td>S</td>
<td>A</td>
<td>12.4</td>
<td>110.0</td>
<td>9.1</td>
<td>41.2</td>
<td>o(n)</td>
<td>C</td>
</tr>
<tr>
<td>ZS2</td>
<td>S</td>
<td>L</td>
<td>14.3</td>
<td>124.0</td>
<td>10.3</td>
<td>44.7</td>
<td>o(n)</td>
<td>C</td>
</tr>
<tr>
<td>S</td>
<td>M</td>
<td>L</td>
<td>11.4</td>
<td>181.0</td>
<td>13.3</td>
<td>43.4</td>
<td>o(n)</td>
<td>C</td>
</tr>
<tr>
<td>FL1</td>
<td>M</td>
<td>L</td>
<td>14.1</td>
<td>174.0</td>
<td>14.4</td>
<td>106.4</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>M</td>
<td>L</td>
<td>18.5</td>
<td>191.0</td>
<td>20.8</td>
<td>121.9</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>FL2</td>
<td>M</td>
<td>N</td>
<td>19.2</td>
<td>245.0</td>
<td>17.0</td>
<td>131.8</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>FL3</td>
<td>M</td>
<td>A</td>
<td>23.4</td>
<td>214.0</td>
<td>28.3</td>
<td>131.2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>AS</td>
<td>M</td>
<td>A</td>
<td>19.9</td>
<td>212.0</td>
<td>30.9</td>
<td>126.9</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>NW</td>
<td>M</td>
<td>A</td>
<td>32.2</td>
<td>326.0</td>
<td>37.3</td>
<td>136.6</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Z</td>
<td>M</td>
<td>N</td>
<td>24.2</td>
<td>319.0</td>
<td>39.0</td>
<td>193.5</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>M1</td>
<td>S</td>
<td>A</td>
<td>53.0</td>
<td>517.0</td>
<td>45.6</td>
<td>306.0</td>
<td>o(n)</td>
<td>o(n)</td>
</tr>
<tr>
<td>FW1</td>
<td>S</td>
<td>A</td>
<td>66.2</td>
<td>800.0</td>
<td>61.2</td>
<td>241.9</td>
<td>o(n)</td>
<td>o(n)</td>
</tr>
<tr>
<td>GLW</td>
<td>M</td>
<td>P</td>
<td>48.0</td>
<td>500.0</td>
<td>61.6</td>
<td>253.0</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>RJ1</td>
<td>S</td>
<td>P</td>
<td>64.7</td>
<td>652.0</td>
<td>67.9</td>
<td>266.0</td>
<td>o(n)</td>
<td>o(n)</td>
</tr>
<tr>
<td>H</td>
<td>S</td>
<td>N</td>
<td>81.9</td>
<td>892.0</td>
<td>75.1</td>
<td>307.5</td>
<td>o(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>RJ2</td>
<td>S</td>
<td>P</td>
<td>90.1</td>
<td>997.0</td>
<td>84.9</td>
<td>416.5</td>
<td>o(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>PW2</td>
<td>S</td>
<td>A</td>
<td>63.2</td>
<td>1064.0</td>
<td>250.2</td>
<td>346.7</td>
<td>o(n)</td>
<td>o(n)</td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>A</td>
<td>322.3</td>
<td>3138.0</td>
<td>280.9</td>
<td>1183.2</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>S'</td>
<td>M</td>
<td>L</td>
<td>249.5</td>
<td>4670.0</td>
<td>285.4</td>
<td>1069.7</td>
<td>o(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>M2</td>
<td>S</td>
<td>U</td>
<td>338.9</td>
<td>8086.0</td>
<td>405.2</td>
<td>2165.9</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>W</td>
<td>S</td>
<td>U</td>
<td>469.6</td>
<td>8508.0</td>
<td>469.1</td>
<td>2492.0</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>
Table 27 Generating unrestricted partitions of 60 and 90 in the multiplicity representation.

<table>
<thead>
<tr>
<th>ALG</th>
<th>ORD</th>
<th>n = 60</th>
<th>n = 90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on PC286</td>
<td>on SUN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on NeXT</td>
<td>on PC286</td>
</tr>
<tr>
<td>S</td>
<td>L</td>
<td>11.9</td>
<td>22.0</td>
</tr>
<tr>
<td>FL1</td>
<td>L</td>
<td>14.2</td>
<td>21.0</td>
</tr>
<tr>
<td>E</td>
<td>L</td>
<td>18.5</td>
<td>22.0</td>
</tr>
<tr>
<td>FL2</td>
<td>N</td>
<td>19.3</td>
<td>23.0</td>
</tr>
<tr>
<td>FL3</td>
<td>A</td>
<td>23.3</td>
<td>25.0</td>
</tr>
<tr>
<td>AS</td>
<td>A</td>
<td>19.8</td>
<td>25.0</td>
</tr>
<tr>
<td>NW</td>
<td>A</td>
<td>32.1</td>
<td>39.0</td>
</tr>
<tr>
<td>Z</td>
<td>N</td>
<td>24.4</td>
<td>38.0</td>
</tr>
<tr>
<td>GLW</td>
<td>P</td>
<td>47.8</td>
<td>59.0</td>
</tr>
<tr>
<td>S'</td>
<td>L</td>
<td>157.9</td>
<td>449.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALG</th>
<th>ORD</th>
<th>n = 60</th>
<th>n = 90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on PC286</td>
<td>on SUN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on NeXT</td>
<td>on PC286</td>
</tr>
<tr>
<td>S</td>
<td>L</td>
<td>11.0</td>
<td>1221.0</td>
</tr>
<tr>
<td>FL1</td>
<td>L</td>
<td>14.1</td>
<td>1212.0</td>
</tr>
<tr>
<td>E</td>
<td>L</td>
<td>18.5</td>
<td>1327.0</td>
</tr>
<tr>
<td>FL2</td>
<td>N</td>
<td>19.1</td>
<td>1700.0</td>
</tr>
<tr>
<td>FL3</td>
<td>A</td>
<td>23.6</td>
<td>1491.0</td>
</tr>
<tr>
<td>AS</td>
<td>A</td>
<td>20.0</td>
<td>1483.0</td>
</tr>
<tr>
<td>NW</td>
<td>A</td>
<td>32.2</td>
<td>2274.0</td>
</tr>
<tr>
<td>Z</td>
<td>N</td>
<td>24.0</td>
<td>2218.0</td>
</tr>
<tr>
<td>GLW</td>
<td>P</td>
<td>48.2</td>
<td>3484.0</td>
</tr>
</tbody>
</table>

Table 28 Generating unrestricted partitions of 60 and 90 in the standard representation.

<table>
<thead>
<tr>
<th>ALG</th>
<th>ORD</th>
<th>n = 60</th>
<th>n = 90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on PC286</td>
<td>on SUN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on NeXT</td>
<td>on PC286</td>
</tr>
<tr>
<td>ZS1</td>
<td>A</td>
<td>12.4</td>
<td>13.0</td>
</tr>
<tr>
<td>ZS2</td>
<td>L</td>
<td>14.3</td>
<td>15.0</td>
</tr>
<tr>
<td>M1</td>
<td>A</td>
<td>48.0</td>
<td>55.0</td>
</tr>
<tr>
<td>PW1</td>
<td>A</td>
<td>61.8</td>
<td>86.0</td>
</tr>
<tr>
<td>RJ1</td>
<td>P</td>
<td>57.7</td>
<td>69.0</td>
</tr>
<tr>
<td>H</td>
<td>N</td>
<td>72.5</td>
<td>95.0</td>
</tr>
<tr>
<td>RJ2</td>
<td>P</td>
<td>83.0</td>
<td>109.0</td>
</tr>
<tr>
<td>PW2</td>
<td>A</td>
<td>56.8</td>
<td>114.0</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>266.3</td>
<td>307.0</td>
</tr>
<tr>
<td>M2</td>
<td>U</td>
<td>279.0</td>
<td>703.0</td>
</tr>
<tr>
<td>W</td>
<td>U</td>
<td>387.3</td>
<td>843.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALG</th>
<th>ORD</th>
<th>n = 60</th>
<th>n = 90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on PC286</td>
<td>on SUN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on NeXT</td>
<td>on PC286</td>
</tr>
<tr>
<td>ZS1</td>
<td>A</td>
<td>12.5</td>
<td>765.0</td>
</tr>
<tr>
<td>ZS2</td>
<td>L</td>
<td>14.3</td>
<td>859.0</td>
</tr>
<tr>
<td>M1</td>
<td>A</td>
<td>57.5</td>
<td>3916.0</td>
</tr>
<tr>
<td>PW1</td>
<td>A</td>
<td>74.0</td>
<td>6066.0</td>
</tr>
<tr>
<td>RJ1</td>
<td>P</td>
<td>71.1</td>
<td>4991.0</td>
</tr>
<tr>
<td>H</td>
<td>N</td>
<td>90.8</td>
<td>6830.0</td>
</tr>
<tr>
<td>RJ2</td>
<td>P</td>
<td>96.7</td>
<td>7437.0</td>
</tr>
<tr>
<td>PW2</td>
<td>A</td>
<td>69.0</td>
<td>8087.0</td>
</tr>
</tbody>
</table>

82
Table 29 Generating unrestricted partitions of 75 in the standard representation

<table>
<thead>
<tr>
<th>Alg.</th>
<th>order</th>
<th>CPU time</th>
<th>CPU time</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PC-286</td>
<td>SUN</td>
<td>NeXT</td>
</tr>
<tr>
<td>ZS1</td>
<td>A</td>
<td>110.0</td>
<td>9.1</td>
<td>41.2</td>
</tr>
<tr>
<td>ZS2</td>
<td>L</td>
<td>124.0</td>
<td>10.3</td>
<td>44.7</td>
</tr>
<tr>
<td>M1</td>
<td>A</td>
<td>517.0</td>
<td>45.6</td>
<td>306.0</td>
</tr>
<tr>
<td>PW1</td>
<td>A</td>
<td>800.0</td>
<td>61.2</td>
<td>241.9</td>
</tr>
<tr>
<td>RJ1</td>
<td>P</td>
<td>652.0</td>
<td>67.9</td>
<td>266.0</td>
</tr>
<tr>
<td>H</td>
<td>N</td>
<td>892.0</td>
<td>75.1</td>
<td>307.5</td>
</tr>
<tr>
<td>RJ2</td>
<td>P</td>
<td>997.0</td>
<td>84.9</td>
<td>416.5</td>
</tr>
<tr>
<td>PW2</td>
<td>A</td>
<td>1064.0</td>
<td>250.2</td>
<td>346.7</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>3138.0</td>
<td>280.9</td>
<td>1183.2</td>
</tr>
<tr>
<td>M2</td>
<td>U</td>
<td>8086.0</td>
<td>405.2</td>
<td>2165.9</td>
</tr>
<tr>
<td>W</td>
<td>U</td>
<td>8508.0</td>
<td>469.1</td>
<td>2492.0</td>
</tr>
</tbody>
</table>

Table 30 Generating unrestricted partitions of 75 in the multiplicity representation

<table>
<thead>
<tr>
<th>Alg.</th>
<th>order</th>
<th>CPU time</th>
<th>CPU time</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PC-286</td>
<td>SUN</td>
<td>NeXT</td>
</tr>
<tr>
<td>S</td>
<td>L</td>
<td>181.0</td>
<td>13.3</td>
<td>43.4</td>
</tr>
<tr>
<td>FL1</td>
<td>L</td>
<td>174.0</td>
<td>14.4</td>
<td>106.4</td>
</tr>
<tr>
<td>E</td>
<td>L</td>
<td>191.0</td>
<td>20.8</td>
<td>121.9</td>
</tr>
<tr>
<td>FL2</td>
<td>N</td>
<td>245.0</td>
<td>17.0</td>
<td>131.8</td>
</tr>
<tr>
<td>FL3</td>
<td>A</td>
<td>214.0</td>
<td>28.3</td>
<td>131.2</td>
</tr>
<tr>
<td>AS</td>
<td>A</td>
<td>212.0</td>
<td>30.9</td>
<td>126.9</td>
</tr>
<tr>
<td>NW</td>
<td>A</td>
<td>326.0</td>
<td>37.3</td>
<td>136.6</td>
</tr>
<tr>
<td>Z</td>
<td>N</td>
<td>319.0</td>
<td>39.0</td>
<td>193.5</td>
</tr>
<tr>
<td>GLW</td>
<td>P</td>
<td>500.0</td>
<td>61.6</td>
<td>253.0</td>
</tr>
<tr>
<td>S'</td>
<td>L</td>
<td>4670.0</td>
<td>285.4</td>
<td>1069.7</td>
</tr>
</tbody>
</table>
We observe that both algorithms ZS1 and ZS2 can be used to generate restricted partitions. In ZS1 by initializing $x_i \leftarrow \text{L2}$, the algorithm generates all restricted partitions $\leq \text{L2}$. In ZS2 by forcing the algorithm to stop when $x_i > \text{L2}$, the algorithm generates all restricted partitions $\leq \text{L2}$. In fact, both algorithms show the same ratio of performance compared with the generation of unrestricted partitions. In this implementation, the other algorithms that generate unrestricted partitions were excluded due to their inability to show superior performance. Algorithm W was excluded because $\text{L2}=n$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>L2=15</th>
<th>L2=30</th>
<th>L2=45</th>
<th>L2=60</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZS1 (sec)</td>
<td>38.0</td>
<td>103.0</td>
<td>109.0</td>
<td>110.0</td>
</tr>
<tr>
<td>ZS2 (sec)</td>
<td>43.0</td>
<td>115.0</td>
<td>123.0</td>
<td>123.0</td>
</tr>
<tr>
<td>S (sec)</td>
<td>59.0</td>
<td>167.0</td>
<td>179.0</td>
<td>181.0</td>
</tr>
<tr>
<td>RJ1 (sec)</td>
<td>275.0</td>
<td>621.0</td>
<td>650.0</td>
<td>652.0</td>
</tr>
<tr>
<td>S' (sec)</td>
<td>345.0</td>
<td>1781.0</td>
<td>2818.0</td>
<td>3748.0</td>
</tr>
</tbody>
</table>

Table 32 Generating restricted partitions of 75 on SUN

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>L2=15</th>
<th>L2=30</th>
<th>L2=45</th>
<th>L2=60</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZS1 (sec)</td>
<td>3.2</td>
<td>8.5</td>
<td>9.1</td>
<td>9.1</td>
</tr>
<tr>
<td>ZS2 (sec)</td>
<td>3.5</td>
<td>9.5</td>
<td>10.2</td>
<td>10.3</td>
</tr>
<tr>
<td>S (sec)</td>
<td>4.1</td>
<td>11.8</td>
<td>13.0</td>
<td>13.0</td>
</tr>
<tr>
<td>RJ1 (sec)</td>
<td>28.8</td>
<td>64.9</td>
<td>68.0</td>
<td>68.0</td>
</tr>
<tr>
<td>S' (sec)</td>
<td>22.7</td>
<td>112.1</td>
<td>174.6</td>
<td>230.0</td>
</tr>
</tbody>
</table>
Table 33  Generating restricted partitions of 75 on NeXT

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>L2=15</th>
<th>L2=30</th>
<th>L2=45</th>
<th>L2=60</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZS1 (sec)</td>
<td>14.3</td>
<td>38.8</td>
<td>41.5</td>
<td>41.6</td>
</tr>
<tr>
<td>S  (sec)</td>
<td>13.7</td>
<td>39.9</td>
<td>43.1</td>
<td>43.3</td>
</tr>
<tr>
<td>ZS2 (sec)</td>
<td>15.1</td>
<td>41.2</td>
<td>43.9</td>
<td>44.1</td>
</tr>
<tr>
<td>RJ1 (sec)</td>
<td>110.4</td>
<td>251.1</td>
<td>263.3</td>
<td>264.0</td>
</tr>
<tr>
<td>S' (sec)</td>
<td>89.2</td>
<td>443.1</td>
<td>689.5</td>
<td>906.0</td>
</tr>
</tbody>
</table>

As mentioned before, algorithm S does not generate the output in the proper form. Only on the NeXT does it show better performance than ZS2.
Chapter VI

Parallel Algorithms

In this chapter we describe the generation of partitions at random, sequentially and in parallel. We describe briefly the generation of partitions in parallel in antilexicographic order using standard representation and multiplicity representation form. We discuss the calculation of the average number of parts in standard representation as well as in multiplicity representation form. Also, we define the cost, sequentially and in parallel, and the optimal cost solution for generating partitions in parallel.
6.1 Generating a Random Partition Sequentially

Choosing partition at "random" means that each partition of \( n \) is equally likely to be chosen. Let \( r \) be any real number generated by the random generator function \text{rand()}\) between 0 and 1 inclusively. Let \( p' = \lfloor r \times P(n,n) \rfloor \), referring to Table 2 in Chapter 1. \( \text{RP}(n,m) \) represents the number of restricted partitions using at most \( m \) parts. By locating \( p' \) in the table such that \( p' \leq \text{RP}(n,m) \) for minimum possible \( m \), we determine \( m \) which is the randomly selected part. The remainder to be distributed is \( (n - m) \). The next part of the partition is generated by repeating the same process. Example: \( n = 12 \) and \( r = .269 \), \( p' = .269 \times 77 = 20.71 \) (\( \text{RP}(12,12)=77 \)). By looking into the table for the number 20 using \( n = 12 \), since 20 falls between 19 and 34 we obtain \( m = 4 \) which is the first part. The remainder to be distributed is \( 12 - 4 = 8 \) (new \( n \) is 8 and the largest part is 4). For \( r = .645 \), respectively we get \( p' = .643 \times 15 = 9.645 \) (\( \text{RP}(8,4)=15 \)), and \( m = 3 \). The remainder now is \( 8 - 3 = 5 \) (new \( n \) is 5 and the largest part is 3). Suppose \( r = .859 \), we get \( p' = .859 \times 5 = 4.295 \) (\( \text{RP}(5,3)=5 \)), and \( m = 3 \). The remainder now is 2 (new \( n \) is 2 and the largest part is 2). Suppose \( r = .479 \), we get \( p' = .479 \times 2 = .958 \) (\( \text{RP}(2,2)=2 \)), and \( m = 1 \). The remainder now is 1 (new \( n \) is 1 and the largest part is 1). Suppose \( r = .777 \), we get \( p' = .777 \) (\( \text{RP}(1,1)=1 \)), and \( m = 1 \). The partition generated is 4 3 3 1 1. A similar idea is used by [NW].
6.2 Generating a Random Partition in Parallel

[S1] analyses the generation of random partitions in parallel by assuming the CREW PRAM (concurrent-read, exclusive-write parallel RAM) model of computation [GR] as follows:

Given $S = \{s_1, s_2, \ldots, s_p\}$ where $s_1 < s_2, \ldots, s_p$ denoted $P_s(n,k)$ the number of partitions of $n$ having largest part equal to $s_k$ and all parts belonging to the set $S$, and $P'_s(n,k)=\Sigma_{l=1}^{k}P_s(n,l)$. The idea behind the generating algorithms is to first generate in parallel all of the random choices that may be necessary for the construction of the random partition. The random choices that are actually used can be combined to produce the required partition in $O(\log N)$ parallel time, where $N$ is the integer being partitioned. In order to generate a random partition of $N$, one must first choose the largest part by generating a random number between $0$ and $P_s(N)$ and using the probabilities derived from the quantities $P_s'(N,1), \ldots, P_s'(N,p)$ (this can be done in constant parallel time using $p$ processors). The algorithm relies on the following formula:

$$P_s(n,k)=P_s(n-s_k,k)+P_s(n-s_k+s_{k-1},k-1)$$

with the assumption that the processors are indexed by the tuples $(n,k)$, for $1 \leq s_k \leq n \leq N$. 


6.3 Average Number of Parts in the Standard Representation

In this section we try to calculate the average number of parts per partition. This result will later be used to determine the optimal cost for generating partitions in parallel using standard representation. We are not aware of any formula that calculates the average number of parts in the standard representation. Our research led to the derived formula below. Recall Ferrers graph property: The number of partitions $\text{PE}(n,m)$ of an integer $n$ into exactly $m$ parts is equal to the number of partitions $\text{PL}(n,m)$ of the integer $n$ into parts, the largest of which is $m$. There are $\text{PE}(n,m)$ partitions with exactly $m$ parts. The total number of parts in these partitions is $m \times \text{PE}(n,m)$. The total number of parts $\text{TP}(n)$ is thus derived as

\[ \text{TP}(n) = \sum_{m=1}^{n} m \times \text{PE}(n,m) \]

and the average number of parts denoted by $\text{AVRP}(n)$ is equal to $\text{TP}(n) / P(n)$. By implementing this formula (vide program PAVER), we get the following:
Table 34  Average number of parts in standard representation

<table>
<thead>
<tr>
<th>n</th>
<th>P(n)</th>
<th>AVRP (n)</th>
<th>C</th>
<th>n^{2/3}</th>
<th>REL. ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>176</td>
<td>6.06</td>
<td>0.997</td>
<td>6.07</td>
<td>0.0016</td>
</tr>
<tr>
<td>30</td>
<td>5604</td>
<td>9.73</td>
<td>1.008</td>
<td>9.65</td>
<td>0.0082</td>
</tr>
<tr>
<td>45</td>
<td>89134</td>
<td>12.80</td>
<td>1.011</td>
<td>12.66</td>
<td>0.0109</td>
</tr>
<tr>
<td>60</td>
<td>966467</td>
<td>15.51</td>
<td>1.012</td>
<td>15.32</td>
<td>0.0122</td>
</tr>
<tr>
<td>75</td>
<td>8118264</td>
<td>17.99</td>
<td>1.011</td>
<td>17.79</td>
<td>0.0111</td>
</tr>
<tr>
<td>90</td>
<td>56634173</td>
<td>20.29</td>
<td>1.010</td>
<td>20.08</td>
<td>0.0103</td>
</tr>
<tr>
<td>105</td>
<td>342325709</td>
<td>22.45</td>
<td>1.009</td>
<td>22.24</td>
<td>0.0093</td>
</tr>
<tr>
<td>150</td>
<td>40853235548</td>
<td>28.35</td>
<td>1.001</td>
<td>28.32</td>
<td>0.0010</td>
</tr>
<tr>
<td>195</td>
<td>2580840231841</td>
<td>33.61</td>
<td>0.999</td>
<td>33.64</td>
<td>0.0008</td>
</tr>
<tr>
<td>240</td>
<td>105882249516398</td>
<td>38.42</td>
<td>0.994</td>
<td>38.65</td>
<td>0.0059</td>
</tr>
<tr>
<td>285</td>
<td>316013791927934</td>
<td>42.90</td>
<td>0.990</td>
<td>43.33</td>
<td>0.0100</td>
</tr>
<tr>
<td>330</td>
<td>7365328464863616</td>
<td>47.12</td>
<td>0.986</td>
<td>47.78</td>
<td>0.0140</td>
</tr>
<tr>
<td>375</td>
<td>1406207445431407104</td>
<td>51.12</td>
<td>0.983</td>
<td>52.00</td>
<td>0.0172</td>
</tr>
<tr>
<td>420</td>
<td>22755289805317304576</td>
<td>54.94</td>
<td>0.979</td>
<td>56.11</td>
<td>0.0212</td>
</tr>
<tr>
<td>465</td>
<td>32010312709195196416</td>
<td>58.60</td>
<td>0.976</td>
<td>60.04</td>
<td>0.0245</td>
</tr>
<tr>
<td>510</td>
<td>3991268667606861086720</td>
<td>62.13</td>
<td>0.973</td>
<td>63.75</td>
<td>0.0260</td>
</tr>
</tbody>
</table>

In order to determine the behavior of the average number of parts, we used an experimental approach hoping to find a function of the form \( f(n) = Cn^k \). The experiment looks for \( C \) and \( k \) in a binary search fashion. We set a value for \( k \) and we calculate \( C \) for all different rows. When \( n \) is increasing, if \( C \) is increasing we increase \( k \); if \( C \) is decreasing, we decrease \( k \). We repeat the process until \( C \) maintains the same value for all sizes of \( n \). By fixing \( k \) to 2/3, we get the above mentioned values of \( C \) (almost 1). The exact value of \( n^{2/3} \) is shown in the table, and the last column shows the relative error, which is less than 0.03% for \( n \leq 510 \). Our empirical measure shows that the average number of parts is \( O(n^{2/3}) \).
6.4 Average Number of Parts in the Multiplicity Representation

In this section we try to estimate the average number of parts per partition. Later on we will use the estimation to determine the optimal cost for generating partitions in parallel using multiplicity representation. We are not aware of any definite formula that calculates the average number of parts in the multiplicity representation form. Our research fails to find such formula. The only acceptable solution was to count the parts during program execution. The counting was measured while generating the unrestricted partitions. Since the generation of the unrestricted partitions for \( n > 120 \) takes more than 12 hours even on the SUN workstations, our measurement was limited to \( n = 120 \). More values, however, for \( n \) were measured in between.

We shall now prove that the maximum number of parts, \( d \), per partition is:

\[
d < \sqrt{2n}
\]

Recall \( c_1y_1 + c_2y_2 + ... + c_dy_d = n \).

The minimal value of the last part is \( 1 \), \( y_d \geq 1 \), the minimal value of the next to the last part is \( 2 \), \( y_{d-1} \geq 2 \), ..., the minimal value of the first part is \( d \), \( y_1 \geq d \).

Then \( n = c_1y_1 + c_2y_2 + ... + c_dy_d \geq y_1 + y_2 + ... + y_{d-1} \geq d + d + ... + 2 + 1 \)
Thus
\[ 1 + 2 + \ldots + d \leq n \]
\[ \frac{d(d+1)}{2} \leq n \]
\[ d^2 + d \leq 2n \]
\[ d^2 < 2n \]
and, as a result
\[ d < \sqrt{2n}. \]

Table 35  Average number of parts in the multiplicity representation

<table>
<thead>
<tr>
<th>n</th>
<th>P(n)</th>
<th>TP(n)</th>
<th>AVRP(n)</th>
<th>C</th>
<th>n^{4/9}</th>
<th>REL. ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>176</td>
<td>508</td>
<td>2.89</td>
<td>0.867</td>
<td>3.33</td>
<td>0.1522</td>
</tr>
<tr>
<td>22</td>
<td>1002</td>
<td>3506</td>
<td>3.50</td>
<td>0.886</td>
<td>3.95</td>
<td>0.1285</td>
</tr>
<tr>
<td>30</td>
<td>5604</td>
<td>23025</td>
<td>4.11</td>
<td>0.907</td>
<td>4.53</td>
<td>0.1021</td>
</tr>
<tr>
<td>37</td>
<td>21637</td>
<td>99133</td>
<td>4.58</td>
<td>0.921</td>
<td>4.97</td>
<td>0.0851</td>
</tr>
<tr>
<td>45</td>
<td>89134</td>
<td>451501</td>
<td>5.07</td>
<td>0.935</td>
<td>5.42</td>
<td>0.0690</td>
</tr>
<tr>
<td>52</td>
<td>281589</td>
<td>1535914</td>
<td>5.45</td>
<td>0.942</td>
<td>6.78</td>
<td>0.0505</td>
</tr>
<tr>
<td>60</td>
<td>966467</td>
<td>5672882</td>
<td>5.87</td>
<td>0.951</td>
<td>6.17</td>
<td>0.0511</td>
</tr>
<tr>
<td>67</td>
<td>2679689</td>
<td>16644217</td>
<td>6.21</td>
<td>0.958</td>
<td>6.48</td>
<td>0.0434</td>
</tr>
<tr>
<td>75</td>
<td>8118264</td>
<td>53419131</td>
<td>8.58</td>
<td>0.966</td>
<td>6.81</td>
<td>0.0354</td>
</tr>
<tr>
<td>82</td>
<td>20506255</td>
<td>141227966</td>
<td>6.89</td>
<td>0.973</td>
<td>7.08</td>
<td>0.0275</td>
</tr>
<tr>
<td>90</td>
<td>56634173</td>
<td>409038376</td>
<td>7.22</td>
<td>0.978</td>
<td>7.38</td>
<td>0.0221</td>
</tr>
<tr>
<td>97</td>
<td>133230930</td>
<td>999764335</td>
<td>7.50</td>
<td>0.982</td>
<td>7.63</td>
<td>0.0173</td>
</tr>
<tr>
<td>105</td>
<td>342325709</td>
<td>2674789388</td>
<td>7.81</td>
<td>0.987</td>
<td>7.91</td>
<td>0.0128</td>
</tr>
<tr>
<td>112</td>
<td>761002156</td>
<td>6145056855</td>
<td>8.07</td>
<td>0.991</td>
<td>8.14</td>
<td>0.0086</td>
</tr>
<tr>
<td>120</td>
<td>1844349559</td>
<td>15425991887</td>
<td>8.36</td>
<td>0.996</td>
<td>8.39</td>
<td>0.0035</td>
</tr>
</tbody>
</table>

Using the same approach used in the previous section, we derive \( f(n) = n^{4/9} \) with relative error \( \leq 0.2\% \). Our empirical measure shows that the average number of parts is \( O(n^{4/9}) \).
6.5 Generating Partitions in

Antilexicographic Order in Parallel

In this section, our focus is to determine the cost for generating the unrestricted partitions sequentially and in parallel. The time complexity for generating all unrestricted partitions of \( n \) is \( O(P(n)) \). Sequentially, in the *multiplicity representation* the partitions can be generated with constant average delay time. In such case the cost is defined as \( O(P(n)) \) (excluding the output), and \( O(P(n)\cdot AVRG(n)) \) (including the output). In the *standard representation* this is applied only on ZS1 and ZS2 since they have constant average delay property. The other algorithms failed to have this property. In parallel the algorithms described in [AS] work with constant delay time, and generate the output; also, every part in the partition is generated by a processor. In the *multiplicity representation* the cost is \( O(\sqrt{n}\cdot P(n)) \) (since \( d\leq\sqrt{2n} \), and \( d \) represents the number of processors involved). In the *standard representation* obviously \( n \) processors are required. In this case the cost is \( O(n\cdot P(n)) \). The number of processors used in those two algorithms is equal to the maximal number of parts in the corresponding representation.

Paper [AS] shows two algorithms that generate the unrestricted partitions of \( n \) in antilexicographic order. The first algorithm generates the partitions in the *multiplicity*
representation while the second algorithm generates the same partitions in the standard representation.

In the multiplicity representation as well as in the standard representation, the partition is generated by using all processors such that the processor directs the remainder to be distributed from its neighbour on the left side. After generating the part, the next neighbour processor on the right side can start the process for generating the proper part.

In the multiplicity representation, and by reserving $n^{1/2}$ processors, paper [AS] shows an optimal solution of $O(n^{1/2}P(n))$. However, since our empirical measure indicates that the average number of parts per partition AVRP($n$) is $n^{4/9}$, by using our definition for the optimal cost the optimal solution should actually be $O(n^{4/9}P(n))$.

Similarly, in the standard representation, and by reserving $n$ processors, paper [AS] shows an optimal solution of $O(nP(n))$. Since, however, our empirical measure indicates that the average number of parts per partition AVRP($n$) is $n^{2/3}$, by using our definition for the optimal cost the optimal solution is actually reduced to $O(n^{2/3}P(n))$. 

94
Chapter VII

Conclusions

7.1 Major Achievement

We have described in this paper two new algorithms, referred to as ZS1 and ZS2, for generating integer partitions in standard representation form. Both prove to have constant average delay. To the best of our knowledge, these are the first such algorithms for which the constant delay property has been proven. They are the fastest algorithms in the standard representation as well as in the multiplicity representation. By proving that they have constant average delay property, we can consider them as a kind of upper bound for the generation of unrestricted as well as restricted partitions of an integer.
7.2 Future Work

The first open problem resulting from this research is to find a constant time (worst case) delay algorithm for generating unrestricted integer partitions in standard representation, exclusive of the output. This means that there should be a constant number of differences in parts in neighbouring partitions. Algorithm [Sa] achieves this (with one minimal change) but fails to do so in constant time.

Further research could also be directed at an attempt to determine whether there exist algorithms with significantly better performance than any of ZS1 and ZS2.

As well, the idea of the special cases used in ZS1 and ZS2 may be applied to the generating of doubly restricted partitions.

Cost-optimal parallel algorithms for generating integer partitions which have constant delay property in the worst case using a linear array of processors may also be investigated.
A related open problem is to find theoretically the average number of parts in the *standard representation* as well as in the *multiplicity representation* form.
Programs
```c
int main() {
    int x, i, j; // Declaration of variables
    printf("Please enter the value of X: ");
    scanf("%d", &x); // Reading input
    int i, j, a, b, c, d, e; // Declaration of variables
    printf("Please enter the value of I: ");
    scanf("%d", &i); // Reading input
    printf("Please enter the value of J: ");
    scanf("%d", &j); // Reading input
    printf("Please enter the value of A: ");
    scanf("%d", &a); // Reading input
    printf("Please enter the value of B: ");
    scanf("%d", &b); // Reading input
    printf("Please enter the value of C: ");
    scanf("%d", &c); // Reading input
    printf("Please enter the value of D: ");
    scanf("%d", &d); // Reading input
    printf("Please enter the value of E: ");
    scanf("%d", &e); // Reading input
    // Perform calculations
    // Output results
    return 0; // Return success
}
```
```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

int main(int argc, char** argv)
{
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int* matrix = (int*) malloc(n * n * sizeof(int));
    if (matrix == NULL)
    {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }

    printf("Enter the matrix:
");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("Enter element (%d,%d): ", i, j);
            scanf("%d", &matrix[i * n + j]);
        }
    }

    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            sum += matrix[i * n + j];
        }
    }

    printf("The sum is: 
";}
```

```c
/* **************************** ****************************************** */
/* **************************** ****************************************** */

xil[1] = xil[0];
for (x = 1; x <= 10; x++)
    xil[x] = xil[x-1];

for (x = 1; x <= 10; x++)
    printf("\%d\n", xil[x]);

xil[10] = 10;
for (x = 1; x <= 10; x++)
    printf("\%d\n", xil[x]);

for (x = 1; x <= 10; x++)
    printf("\%d\n", xil[x]);

for (x = 1; x <= 10; x++)
    printf("\%d\n", xil[x]);
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* program: r2avrg.c */

#include <time.h>

int main()
{
  float p[100][100];
  double pi;
  double (f2z, avrg);

  n := 1
  for (n = 0; n <= 100; n++)
    for (m = 0; m <= 100; m++)
      pp[n][m] = 0;

  printf("Enter N: ");
  scanf("%d", &n);
  pp[0][0] = 1;
  for (n = 1; n <= 100; n++)
    for (m = 1; m <= n; m++)
      pp[n][m] = 0;
    for (m = 1; m <= n; m++)
      if (m > n) pp[n][m] = 0;
    else if (m == n) pp[n][m] = 1;
    else if (m < n) pp[n][m] = pp[n-1][m] + pp[n-1][m-1];

  /*
  for (n = 0; n <= 100; n++)
    printf("%d
", pp[n][n]);
  printf("\n");
  */

  pi := 0;
  sz := 0;
  for (elimination
    pt := pt + (double) pp[i][i];
  sz := sz + (double) pp[i][i];
  pi := pi + (double) pp[i][i];
  avrg := pi / sz;
  printf("Number of partitions %d\n", i);
  printf("Total parts %d\n", pt);
  printf("Average of parts %d\n", avrg);
  exit(0);
}
```
Bibliography


[Mk'] J.K.S. McKay, Number of restricted partitions of n, Alg. 262, CACM, 8, 1965, p.493.


2, No.4, December 1976, 364-374.


[W1] J.S. White, Number of doubly restricted partitions, Alg. 373, ibid.
