National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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.

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.

Canada

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
Jully 20, 1993

ISBN 0-315-89677-9

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

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 greateful 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

Tables

Algorithms

Figures

Programs

Chapter I

Introduction

Given an integer n, it is possible to represent it as the sum of one or more positive integers, i.e. $n=x_1+ x_2 +...+ x_m$. 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 Nijenhius 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 $A = (a_1, a_2, ..., a_{s'})$ and $B = (b_1, b_2, ..., b_{s''})$ 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 1111111. 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 $x_1,...,x_m$, where $x_1 \geq x_2 \geq ... \geq x_m$, and $x_1+x_2+...+x_m=n$. 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 $y_1>...>y_d$ be all distinct parts in a partitions, and $c_1,...,c_d$ their respective (positive) multiplicities. Clearly $c_1y_1 + ...+ c_dy_d=n$.

Table 1 Unrestricted partitions of 8

#	standard representation	multiplicity representation
1	8	(8X1)
2	7 1	(7X1) (1X1)
3	6 2	(6X1) (2X1)
4	6 1 1	(6X1) (1X2)
5	5 3	(5X1) (3X1)
6	5 2 1	(5X1) (2X1) (1X1)
7	5 1 1 1	(5X1) (1X3)
8	4 4	(4X2)
9	4 3 1	(4X1) (3X1) (1X1)
10	4 2 2	(4X1) (2X2)
11	4 2 1 1	(4X1) (2X1) (1X2)
12	4 1 1 1 1	(4X1) (1X4)
13	3 3 2	(3X2) (2X1)
14	3 3 1 1	(3X2) (1X2)
15	3 2 2 1	(3X1) (2X2) (1X1)
16	3 2 1 1 1	(3X1) (2X1) (1X3)
17	3 1 1 1 1 1	(3X1) (1X5)
18	2 2 2 2	(2X4)
19	2 2 2 1 1	(2X3) (1X2)
20	2 2 1 1 1 1	(2X2) (1X4)
21	2 1 1 1 1 1 1	(2X1) (1X6)
22	1 1 1 1 1 1 1 1	(1X8)

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 $\lfloor x \rfloor$, $y_2= n - c_1y_1$, $c_2=1$ 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:

$$P(n) \approx \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{2\frac{n}{3}}} + C_n n^{-\frac{1}{2}}$$

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

Note that the relative error is defined as $\dfrac{p - p^*}{p}$

with p and p^* are the exact and the estimated value respectively.

Now

$$\log P(n) = \log e^{\pi\sqrt{2\frac{n}{3}}} - \log 4n\sqrt{3}$$

$$= \pi\sqrt{2\frac{n}{3}} - \log 4\sqrt{3} - \log n$$

$$= O(\sqrt{n})$$

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

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

$$RP(n,m) = RP(n,m-1) + RP(n-m,\min(n-m,m))$$

with RP(n,1)=1, RP(n,0)=0, RP(0,0)=1, and 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,...,RP(n,n)-1 [Mk"]. The number of unrestricted partitions P(n) is equal to RP(n,n).

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

Algorithm RP

$R_{0,0} = 1;$

for i←1 **to** n **do** {

  $R_{i,0} = 0$

  for j←1 **to** i **do** $R_{i,j} = R_{i,j-1} + R_{i-j,\min(i-j,j)}$}

The table on the following page illustrates the calculation of RP(n,m).

Table 2 Number of partitions of n using at most m parts

n \ m	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	2	0	0	0	0	0	0	0	0	0	0
3	1	2	3	0	0	0	0	0	0	0	0	0
4	1	3	4	5	0	0	0	0	0	0	0	0
5	1	3	5	6	7	0	0	0	0	0	0	0
6	1	4	7	9	10	11	0	0	0	0	0	0
7	1	4	8	11	13	14	15	0	0	0	0	0
8	1	5	10	15	18	20	21	22	0	0	0	0
9	1	5	12	18	23	26	28	29	30	0	0	0
10	1	6	14	23	30	35	38	40	41	42	0	0
11	1	6	16	27	37	44	49	52	54	55	56	0
12	1	7	19	34	47	58	65	70	73	75	76	77

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

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

Doubly restricted partitions contain parts of size between L1 and L2, i.e. $L1 \le x_i \le L2$ for i=1,2,...,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,...,r). Tournament scores are studied in [NMS], while graph degree sequences are studied in [JR].

The case of partitions whose largest part is exactly L2 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 L2 (by adding one more part of size L2) is equivalent to the case of partitions of n+L2 with the largest part exactly L2.

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 $z_1...z_m$ be a partition into m nonincreasing parts, $z_1 \geq ... \geq z_m$, and $x_1...x_k$, $x_1 \geq ... \geq x_k$, 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.

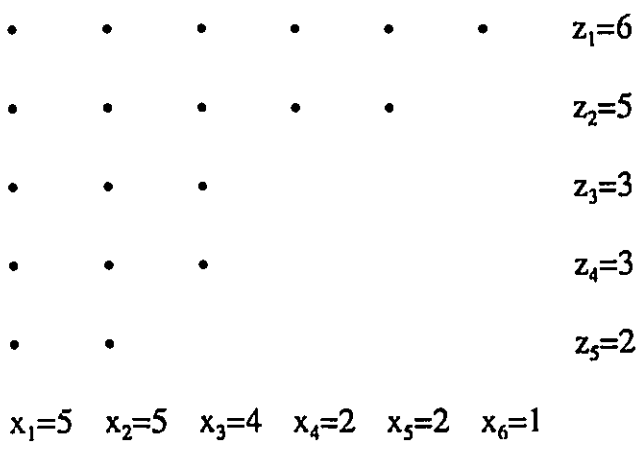


Fig. 1. Ferrers graph

From the Ferrers graph it follows that $z_j = \max\{j \mid x_j \geq i\}$. Consider now the corresponding *multiplicity representation* of partitions with largest part m: $c_1,...,c_d$ are the multiplicities of $y_1,...,y_d$, where $m=y_1>...>y_d$, and $c_1y_1+...+c_dy_d=n$. For the partitions into m parts, let $e_1,...,e_d$ be the multiplicities of $w_1,...,w_d$ (clearly the two sequences have the same number d of different parts), where $w_1>...>w_d$, $e_1w_1+...+e_dw_d=n$, and $e_1+...+e_d=m$. It then follows that $w_{m-1}+1=c_1+c_2+...+c_i$, 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 *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 **constant average delay** property if the ratio is less than a constant for any n, again, exclusive of the output time.

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

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 neccessary 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 re-coded; 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 $O(n^{2/3})$. Thus, the cost-optimal solution for n processors should be $O(n^{2/3}P(n))$. Details are in Chapter VI of this thesis.

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

Professor I. Stojmenovic. The remaining items have been completed by the candidate. A portion of this thesis has been published as a technical report [ZS] and submitted to a journal for publication.

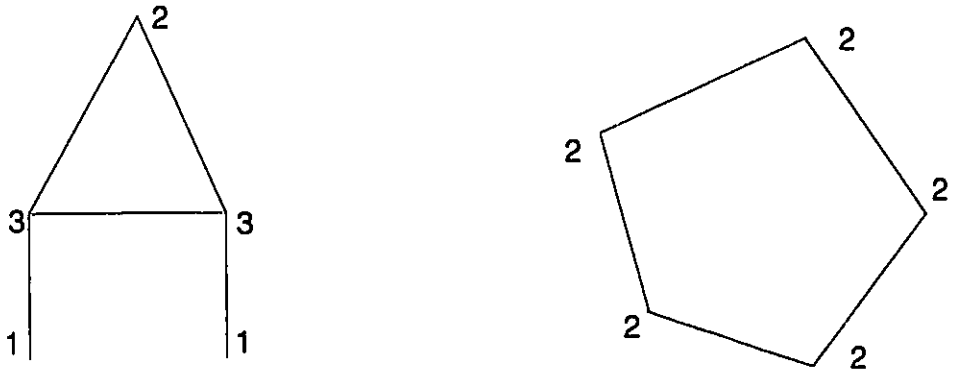
1.3 Applications

Partitions play a fundamental role in many combinatorial and graph theoretical problems. Let $(x_1,...,x_k)$ be a partition of n where we assume $x_1 \le x_2 \le ... \le x_k$. 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 $x_i (i=1,...,k)$ satisfies certain additional restrictions. For example, if

$$n = \frac{k!}{2!(k-2)!} \quad with \quad \sum_{i=1}^{j} x_i \ge \frac{j!}{2!(j-2)!}$$

for j = 1,...,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 vertice 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 (2 2 2 2 2, 3 2 2 2 1, 3 3 2 1 1, 4 2 2 1 1, 4 3 1 1 1, 5 2 1 1 1, and 6 1 1 1 1). The partition 3 3 2 1 1 represents graph (a), and the partition 2 2 2 2 2 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 $M=\{m_1,m_1+1,...,m_2\}$, 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

$$v_1+v_2+...+v_p = k,\ 1 \leq v_1 \leq v_2 \leq ... \leq v_p\ .$$

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:

	Total
a - Algorithms that generate unrestricted partitions	19
b - " " " restricted partitions	1
c - " " " doubly restricted partitions	3
d - " " " Multiply restricted partitions	2

According to the representation form

a - Algorithms that generate the output in the standard representation	16
b - Algorithms that generate the output in the multiplicity representation	9

According to the output order

a - Algorithms that generate the partitions in lexicographic order	4
b - " " " " " " antilexicographic order	10
c - " " " " " " non lexicographic order	5
d - " " " " " " part order	4
e - " " " " " " Unranking order	2

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

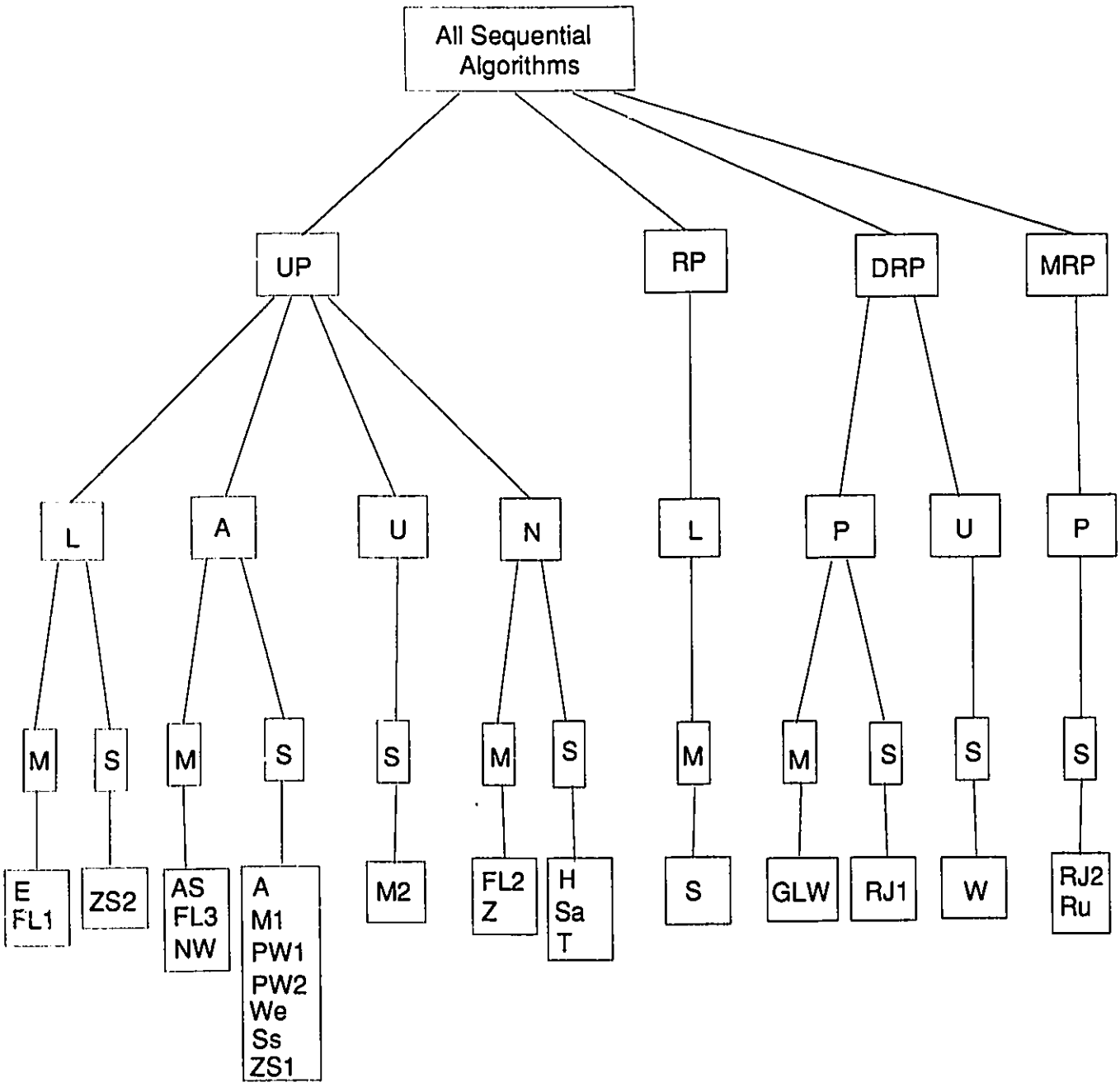


Fig. 3. Classification of all sequential algorithms

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 $x_1\ x_2\ \ldots\ x_m$ in the following way: Let h be the number of parts of x greater than 1, i.e. $x_i>1$ for $1\le i\le h$, and $x_i=1$ for $h<i\le m$. 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 $0<d\le x_h-1$ 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 Nijenhius 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

5x3 + 4x3 + 2x1 is 5x4 + 1x9. 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 I).

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 $j=i,i+1,...,m-1$, 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 $\lceil n/2 \rceil$ 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).

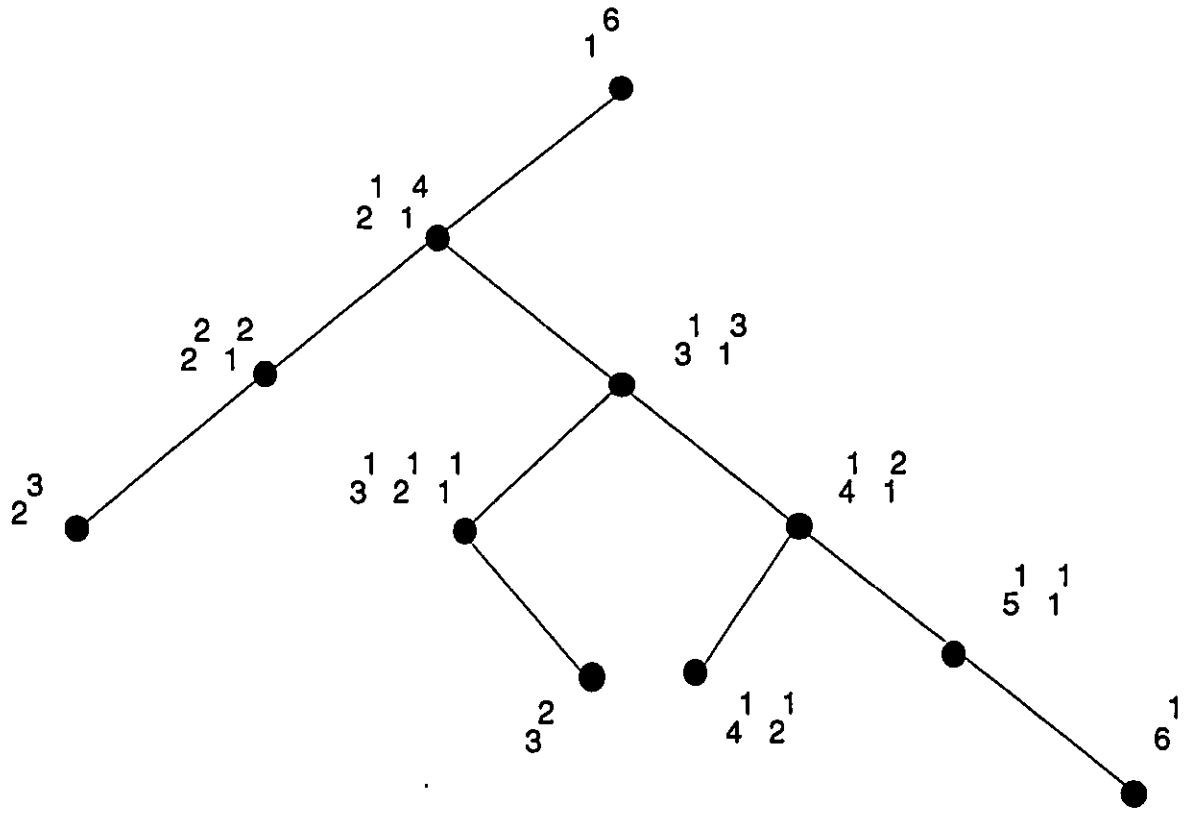


Fig. 4. Binary tree representation for n = 6 T(6)

A binary tree is constructed as follows: Recall $c_1y_1 + c_2y_2 + \ldots + c_dy_d = n$.

Let P_n denote the set of all partitons 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 $1 \le i \le r$. The partition is called an M-partition of S if its weight, defined as the inner product $D.S = \Sigma_i d_i s_i$, 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 $d_1,d_2,...,d_r = \Sigma i\ d_i 2^{r-i}$. 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 these 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:

```
            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

```
                                          ins.    op.
                                            #       #

            if (bb[d-1]==1)                 1
               {
                 ct1      +=1;                      1
                 ct3      +=1;                      2
                 bb[d-1]   =2;             2
                 mm[d-1]  +=1;             3
                 ct1      +=3;                      3
                 ct6      +=2;                      4
               }
       else if (mm[d]==2)                   4
               {
                 ct1      +=1;                      5
                 ct3      +=1;                      6
                 ct4      +=1;                      7
                 bb[d]     =2;             5
                 ct6      +=1;                      8
               }
          else
               {
                 ct5      +=1;                      9
               }
```

```
                                        ins.    op.
                                          #      #

while (t<pp[t])                           6
      {
        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

n	15	30	45	60	75	90
arithmetic	1831	79240	1518808	18816929	175465153	1333745222
while	507	23024	451500	5672881	53419130	409038375
if then	209	6450	101047	1083066	9022074	62521988
if else	———	———	———	———	———	———
if unused	141	4756	77219	849866	7214452	50746356
others	2085	86733	1633818	20034646	185515995	1402905462
totals	4773	200203	3782392	46457386	430636804	3258957403
inst./part.	27.1	35.7	42.4	48.0	53.0	57.5
PC286 (sec)	< 1	< 1	5.0	55.0	517.0	3916.0
SUN (sec)	0.0	0.0	0.4	4.9	45.6	348.5
NeXT (sec)	0.0	0.1	2.7	32.8	306.0	2333.0

Table 5 Algorithm (PW1) by Page and Wilson

n	15	30	45	60	75	90
arithmetic	2047	85885	1621900	19918042	184612180	1397017642
while	———	———	———	———	———	———
if then	507	23024	451500	5672881	53419130	409038375
if else	175	5603	89133	966466	8118263	56634172
if unused	508	23025	451501	5672882	53419131	409038376
others	2909	120117	2251671	27523860	254267841	1919324366
totals	6146	257651	4865705	59754131	553836545	4191052931
inst./part.	34.9	45.9	54.5	61.8	66.2	74.0
PC286 (sec)	< 1	< 1	7.0	86.0	800.0	6066.0
SUN (sec)	0.0	0.0	0.5	6.6	61.2	469.5
NeXT (sec)	0.0	0.1	2.1	26.3	241.9	1812.5

Table 6 Algorithm (PW2) by Page and Wilson

n	15	30	45	60	75	90
arithmetic	1523	69074	1354502	17018645	160257392	1227115127
while	——	——	——	——	——	——
if then	684	28629	540635	6639349	61537395	465672549
if else	508	23025	451501	5672882	53419131	409038376
if unused	——	——	——	——	——	——
others	2562	108914	2073408	25590931	237031318	1806056025
totals	5277	229642	4420046	54921807	513245236	3907882077
inst./part.	29.9	45.9	49.5	56.8	63.2	69.0
PC286 (sec)	< 1	1.0	10.0	114.0	1064.0	8087.0
SUN (sec)	0.0	0.1	1.8	25.5	250.2	2021.6
NeXT (sec)	0.0	0.1	2.9	36.9	346.7	2644.1

Table 7 Algorithm (A) by Andrews

n	15	30	45	60	75	90
arithmetic	7954	400934	8543322	115066417	1149059414	——
while	560	31538	689444	9320269	92635761	——
if then	336	11177	178222	1932873	16236452	——
if else	——	——	——	——	——	——
if unused	14	29	44	59	74	——
others	6895	401401	9246307	131095110	1359084944	——
totals	15759	845079	18657339	257414728	2617016704	——
inst./part.	89.5	150.7	209.3	266.3	322.3	——
PC286 (sec)	1.0	1.0	23.0	307.0	3138.0	——
SUN (sec)	0.0	0.1	2.0	27.4	280.9	——
NeXT (sec)	0.0	0.3	8.2	115.4	1183.2	——

3.3 Antilexicographic Order and Multiplicity Representation

Table 8 Algorithm (NW) by Nijenhius and Wilf

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

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

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

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

n	15	30	45	60	75	90
arithmetic	1485	47713	765255	8335040	70262206	491472685
while	176	5604	89134	966467	8118264	56634173
if then	269	9129	150349	1663639	14178999	99991849
if else	96	2277	30383	289685	2199591	14116434
if unused	163	5406	86670	946077	7976202	55794236
others	1049	32989	525781	5704943	47964952	334846672
totals	3238	103118	1647572	17905851	150700214	1052856049
inst./part.	18.4	18.4	18.4	18.5	18.5	18.5
PC286 (sec)	< 1	1.0	2.0	22.0	191.0	1327.0
SUN (sec)	0.0	0.0	0.2	2.5	20.8	146.9
NeXT (sec)	0.0	0.0	1.3	14.5	121.9	849.5

Table 13 Algorithm (S) by Stockmal

n	15	30	45	60	75	90
arithmetic	1150	27530	370026	3631288	28530858	189565100
while	375	9684	137073	1391055	11191846	75708361
if then	174	5602	89132	966465	8118262	56634171
if else	1	1	1	1	1	1
if unused						
others	1509	38745	548301	5564229	44767393	302833453
totals	3209	81562	1144533	11553038	92608360	624741086
inst./part.	18.2	14.5	12.8	11.9	11.4	11.0
PC286 (sec)	< 1	< 1	2.0	22.0	181.0	1221.0
SUN (sec)	0.0	0.0	0.1	1.6	13.3	89.8
NeXT (sec)	0.0	0.0	0.5	5.4	43.6	292.2

Table 14 Algorithm (S') by Stockmal

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

3.5 Unranking

Table 15 Algorithm (M2) by McKay

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

Table 16 Algorithm (W) by White

n	15	30	45	60	75	90
arithmetic	6699	340008	7528748	103944080	1055168509	—
while	176	5604	89134	966467	8118264	—
if then	2705	168361	4011560	57988951	608871265	—
if else	2459	118478	2432743	31650839	306290187	—
if unused	—	—	—	—	—	—
others	10638	578340	12945533	179820069	1834440856	—
totals	22677	1210791	27007718	374370406	3812889061	—
inst./part.	128.8	216.0	303.0	387.3	469.6	—
c PC286	1.0	3.0	61.0	843.0	8508.0	—
c SUN	0.0	0.1	3.3	46.4	469.1	—
c NeXT	0.0	0.8	17.8	246.1	2492.0	—

3.6 Part Order

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

n	15	30	45	60	75	90
arithmetic	3757	128238	2059535	22409986	188737854	1319291205
while	176	5604	89134	966467	8118264	56634173
if then	715	24944	418247	4680904	40195421	285019268
if else	195	5170	70168	675517	5160853	33280654
if unused	552	16568	250479	2626496	21522332	147299430
others	2519	83617	1359226	14928378	126551953	888920192
totals	7914	264141	4246789	46287748	390286677	2730444922
inst./part.	44.9	47.1	47.6	47.8	48.0	48.2
PC286 (sec)	< 1	< 1	5.0	59.0	500.0	3484.0
SUN (sec)	0.0	0.0	0.6	7.3	61.6	431.1
NeXT (sec)	0.0	0.1	2.7	29.9	253.0	1774.5

Table 18 Algorithm (RJ1) by Riha and James

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

Table 19 Algorithm (RJ2) by Riha and James

n	15	30	45	60	75	90
arithmetic	3993	137823	2468074	29658155	271312676	2034678117
while	324	6485	68370	560206	3864293	23240319
if then	542	16841	267446	2899460	24354866	169902608
if else	266	11953	232510	2911309	27340643	208951625
if unused	401	12448	233590	2913199	27343568	208955810
others	5954	192659	3437601	41299408	377701552	2831452796
totals	11480	378209	6707591	80241737	731917598	5477183275
inst./part.	65.2	67.4	75.2	83.0	90.1	96.7
PC286 (sec)	< 1	< 1	9.0	109.0	997.0	7437.0
SUN (sec)	0.0	0.0	0.7	9.2	84.9	634.2
NeXT (sec)	0.0	0.2	3.8	45.5	416.5	3092.1

3.7 Non Lexicographic Order

Table 20 Algorithm (FL2) by Fenner and Loizou

n	15	30	45	60	75	90
arithmetic	1414	45024	710420	7650661	63884399	443411582
while	176	5604	89134	966467	8118264	56634173
if then	360	11247	173548	1847257	15279598	105312145
if else	355	11451	182986	1986684	16696247	116475245
if unused	111	3128	49270	525320	4377988	30327411
others	1046	33364	526734	5687480	47584683	330909305
totals	3462	109818	1732092	18663869	155941179	1083069861
inst./part.	19.6	19.5	19.4	19.3	19.2	19.1
PC286 (sec)	< 1	1.0	3.0	23.0	245.0	1700.0
SUN (sec)	0.0	0.0	0.2	2.0	17.0	120.9
NeXT (sec)	0.0	0.0	1.4	15.7	131.8	920.3

Table 21 Algorithm (H) by Hindenburg

n	15	30	45	60	75	90
arithmetic	3269	136399	2651294	33616178	320638315	2488138501
while	390	9714	137118	1391115	11191921	75708451
if then	161	5574	89089	966407	8118189	56634083
if else	15	30	45	60	75	90
if unused						
others	3356	138363	2693384	34159832	325685777	2525702496
totals	7191	290080	5570830	70133592	665634277	5146183621
inst./part.	40.8	51.7	62.5	72.5	81.9	90.8
PC286 (sec)	1.0	1.0	8.0	95.0	892.0	6830.0
SUN (sec)	0.0	0.0	0.6	8.0	75.1	583.4
NeXT (sec)	0.0	0.1	2.6	32.7	307.5	2343.2

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 requird 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_h. A study shows that most partitions contain at least one part of size 2. More precisely, x_h=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_h>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_h = 2, the next partition will be generated by setting x_h and x_{m+1} to 1. Any attempt to treat the case of x_h = 3,...,n in a simillar way will be more costly than the proposed one in ZS1. Consider the case of x_h = 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 $\lceil t/2 \rceil$ 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 $x_h = 3,...,n$ 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 $\lceil t/2 \rceil$ parts to 2. This makes any attempt more costly. For example the next partition for 5 5 3 1 1 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_1←n; m←1; h←1; output x_1;
3-    while x_1≠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
```

14- **else** { m←h+1;

15- **if** t>1 **then** { h←h+1;

16- x_h←t }}

}

17- **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

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

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 $x_j = x_{j+1} = \ldots = x_{m-1}$. 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 1. In the partition 5 2 1, the next partition is 5 3, simply because the increase of 2 by 1 does not effect 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_1≠ 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←1;
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 }
```

16- **for** i←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 Stojmenovic

n	15	30	45	60	75	90
arithmetic	938	29704	469488	5065525	42398932	294946487
while	234	8281	138434	1547038	13275187	94104032
if then	133	4804	80024	888881	7580397	53433160
if else	74	1885	25872	251246	1932574	12526063
if unused	41	798	9108	77584	537865	3201011
others	1106	35356	561427	6071007	50883141	354267464
totals	2526	80828	1284353	13901281	116608096	812478217
inst./part.	14.3	14.4	14.4	14.3	14.3	14.3
PC286 (sec)	< 1	< 1	1.0	15.0	124.0	859.0
SUN (sec)	0.0	0.0	0.1	1.2	10.3	71.7
NeXT (sec)	0.0	0.0	0.4	5.3	44.7	311.0

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. $RP(n,L2) \geq n^2/12$ for $L2 \geq 3$.

Proof. Since $RP(n,L2) \geq RP(n,3)$ for $L2>3$, it is sufficient to prove $RP(n,3) \geq n^2/12$. In the *multiplicity representation*, the partitions in RP(n,3) are of the following kind: $n=3c_1+2c_2+1c_3$ (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, $0 \leq c_2 \leq \lfloor (n-3c_1)/2 \rfloor$. 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 $\geq n/2 + (n-3)/2 + (n-6)/2 + \ldots + (n-3\lfloor n/3 \rfloor)/2 = (\lfloor n/3 \rfloor+1)n/2 - \lfloor n/3 \rfloor(\lfloor n/3 \rfloor+1)3/4 = (\lfloor n/3 \rfloor+1)(n/2 - \lfloor n/3 \rfloor 3/4) \geq (\lfloor n/3 \rfloor+1)(n/2 - n/4) \geq n^2/12$.

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 $r_j = n - x_1 - x_2 - \ldots - x_j$ 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 ≥ 3 is $\leq q_1/r_1^2 + q_2/r_2^2 + \ldots + q_s/r_s^2 \leq q(1/r_1^2 + \ldots + 1/r_s^2) < q(1/n^2 + 1/(n-1)^2 + \ldots + 1/1^2) < 2q$ (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 limitiations, 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.

Table 24 Number of partitions containing 2

n	P(n)	P(n-2)	avrg.
1	1	-	———
2	2	-	———
3	3	1	33.3%
4	5	2	40.0%
5	7	3	42.8%
6	11	5	45.4%
7	15	7	46.6%
8	22	11	50.0%
9	30	15	50.0%
10	42	22	52.3%
15	176	101	57.3%
30	5604	3718	66.3%
45	89134	63261	70.9%
60	966467	715220	74.0%
75	8118264	6185089	76.1%
90	56634173	44108109	77.8%
105	342325709	271248950	79.2%
150	40853235548	33549419518	82.1%
195	2580840231841	2168627139338	84.0%
240	105882249516398	90436842388063	85.4%
285	3160137919927934	2732873223476004	86.4%
330	73653287464863616	64325373963334256	87.3%
375	1406207445431407104	1238057788230430208	88.0%
420	22755289805217304576	20170182776146399232	88.6%
465	320103127009195196416	285381548465593778176	89.1%
510	3991268667606861086720	3576209494650931118080	89.6%

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].

$$P(n)\approx\frac{1}{4n\sqrt{3}}e^{\pi\sqrt{2\frac{n}{3}}}+C_n n^{-\frac{1}{2}} \quad where \quad C_n=O(1).$$

Let

$$A(n)=\frac{1}{4n\sqrt{3}}e^{\pi\sqrt{2\frac{n}{3}}}$$

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

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

now,

$$\frac{A(n-2)}{A(n)}\approx\frac{n-2}{n}e^{\pi\left(\sqrt{2\frac{n-2}{3}}-\sqrt{2\frac{n}{3}}\right)}$$

$$\approx \frac{n-2}{n} e^{\pi\sqrt{\frac{2}{3}}(\sqrt{n-2}-\sqrt{n})}$$

$$\approx \frac{n-2}{n} e^{\pi\sqrt{\frac{2}{3}}\left(\frac{\sqrt{n-2}+\sqrt{n}}{\sqrt{n-2}+\sqrt{n}}(\sqrt{n-2}-\sqrt{n})\right)}$$

$$\approx \frac{n-2}{n} e^{\pi\sqrt{\frac{2}{3}}\left(\frac{n-2-n}{\sqrt{n-2}+\sqrt{n}}\right)}$$

$$\approx \frac{n-2}{n} e^{\left(\frac{-2\pi\sqrt{\frac{2}{3}}}{\sqrt{n-2}+\sqrt{n}}\right)}$$

$$\approx \left(1-\frac{2}{n}\right) \frac{1}{e^{\left(\frac{-2\pi\sqrt{\frac{2}{3}}}{\sqrt{n-2}-\sqrt{n}}\right)}}$$

but when $n \to \infty$, we get

$$\left(\frac{-2\pi\sqrt{\frac{2}{3}}}{\sqrt{n-2}+\sqrt{n}}\right) \to 0 .$$

Let us analyse the other part of the equation.

When n$\rightarrow \infty$, we get:

$$C_n \frac{n^{-\frac{1}{2}}}{A(n)} = \frac{C_n}{\sqrt{n}A(n)} \rightarrow 0 .$$

and

$$C_{n-2} \frac{(n-2)^{-\frac{1}{2}}}{A(n)} = \frac{C_{n-2}}{\sqrt{n-2}A(n)} \rightarrow 0 .$$

so,

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

as a result

$$\lim_{n\rightarrow\infty} \frac{P(n-2)}{P(n)} = 1 .$$

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 $c_1y_1 + ...+ c_dy_d=n$ with $y_1>...>y_d$, this algorithm generates the partitions with $y_1<...<y_d$. 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_1←n; c_1←1; h←1;
3-    while c_1≠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- $y_i \leftarrow y_{i-1}+1;\}$

21- $c_i \leftarrow k;$

22- $r \leftarrow r-c_i*y_i;$

23- $h \leftarrow i+1$

24- **if** $r=y_i$ **then** $\{ c_i \leftarrow c_i+1;$

25- $h \leftarrow i \}$

26- **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

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

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 L1=1 and L2=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 26 Generating unrestricted partitions of 75

ALG.	rep.	ord.	inst. per part.	time on pc_286	time on SUN	time on NeXT	time com.		memory com.
							wrst	avrg	
ZS1	S	A	12.4	110.0	9.1	41.2	o(n)	C	O(n)
ZS2	S	L	14.3	124.0	10.3	44.7	o(n)	C	O(n)
S	M	L	11.4	181.0	13.3	43.4	o(n)	C	O(n)
FL1	M	L	14.1	174.0	14.4	106.4	C	C	O(n)
E	M	L	18.5	191.0	20.8	121.9	C	C	O(n)
FL2	M	N	19.2	245.0	17.0	131.8	C	C	O(n)
FL3	M	A	23.4	214.0	28.3	131.2	C	C	O(n)
AS	M	A	19.9	212.0	30.9	126.9	C	C	O(n)
NW	M	A	32.2	326.0	37.3	136.6	C	C	O(n)
Z	M	N	24.2	319.0	39.0	193.5	C	C	O(n)
M1	S	A	53.0	517.0	45.6	306.0	O(n)	o(n)	O(n)
PW1	S	A	66.2	800.0	61.2	241.9	O(n)	o(n)	O(n)
GLW	M	P	48.0	500.0	61.6	253.0	C	C	O(n)
RJ1	S	P	64.7	652.0	67.9	266.0	O(n)	o(n)	O(n)
H	S	N	81.9	892.0	75.1	307.5	O(n)	O(n)	O(n)
RJ2	S	P	90.1	997.0	84.9	416.5	O(n)	O(n)	O(n)
PW2	S	A	63.2	1064.0	250.2	346.7	O(n)	o(n)	O(n)
A	S	A	322.3	3138.0	280.9	1183.2	n	n	O(n)
S'	M	L	249.5	4670.0	285.4	1069.7	O(n)	O(n)	O(n)
M2	S	U	338.9	8086.0	405.2	2165.9	n	n	$O(n^2)$
W	S	U	469.6	8508.0	469.1	2492.0	n	n	$O(n^2)$

Table 27 Generating unrestricted partitions of 60 and 90 in the multiplicity representation.

		n = 60				n = 90			
ALG	ORD	inst. per part.	time on PC286	time on SUN	time on NeXT	inst. per part.	time on PC286	time on SUN	time on NeXT
S	L	11.9	22.0	1.6	5.4	11.0	1221.0	89.8	292.2
FL1	L	14.2	21.0	1.7	12.5	14.1	1212.0	100.0	742.5
E	L	18.5	22.0	2.5	14.5	18.5	1327.0	146.9	849.5
FL2	N	19.3	23.0	2.0	15.7	19.1	1700.0	120.9	920.3
FL3	A	23.3	25.0	3.3	15.5	23.6	1491.0	204.1	921.0
AS	A	19.8	25.0	3.6	14.9	20.0	1483.0	221.6	879.4
NW	A	32.1	39.0	4.4	16.3	32.2	2274.0	263.6	957.2
Z	N	24.4	38.0	4.7	22.9	24.0	2218.0	274.2	1344.5
GLW	P	47.8	59.0	7.3	29.9	48.2	3484.0	431.1	1774.5
S'	L	157.9	449.0	27.4	103.2	———	———	———	———

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

		n = 60				n = 90			
ALG	ORD	inst. per part.	time on PC286	time on SUN	time on NeXT	inst. per part.	time on PC286	time on SUN	time on NeXT
ZS1	A	12.4	13.0	1.1	4.9	12.5	765.0	64.6	285.7
ZS2	L	14.3	15.0	1.2	5.3	14.3	859.0	71.7	311.0
M1	A	48.0	55.0	4.9	32.8	57.5	3916.0	348.5	2333.0
PW1	A	61.8	86.0	6.6	26.3	74.0	6066.0	469.5	1812.5
RJ1	P	57.7	69.0	7.1	28.3	71.1	4991.0	527.9	2013.7
H	N	72.5	95.0	8.0	32.7	90.8	6830.0	583.4	2343.2
RJ2	P	83.0	109.0	9.2	45.5	96.7	7437.0	634.2	3092.1
PW2	A	56.8	114.0	25.5	36.9	69.0	8087.0	2021.6	2644.1
A	A	266.3	307.0	27.4	115.4	———	———	———	———
M2	U	279.0	703.0	39.4	210.0	———	———	———	———
W	U	387.3	843.0	46.4	246.1	———	———	———	———

Table 29 Generating unrestricted partitions of 75 in the standard representation

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

Table 30 Generating unrestricted partitions of 75 in the multiplicity representation

Alg.	order	CPU time PC-286	CPU time SUN	CPU time NeXT
S	L	181.0	13.3	43.4
FL1	L	174.0	14.4	106.4
E	L	191.0	20.8	121.9
FL2	N	245.0	17.0	131.8
FL3	A	214.0	28.3	131.2
AS	A	212.0	30.9	126.9
NW	A	326.0	37.3	136.6
Z	N	319.0	39.0	193.5
GLW	P	500.0	61.6	253.0
S'	L	4670.0	285.4	1069.7

We observe that both algorithms ZS1 and ZS2 can be used to generate restricted partitions. In ZS1 by initializing $x_1 \leftarrow L2$, the algorithm generates all restricted partitions $\leq L2$. In ZS2 by forcing the algorithm to stop when $x_1 > L2$, the algorithm generates all restricted partitions $\leq L2$. 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 L2=n.

Table 31 Generating restricted partitions of 75 on PC286

Algorithm	L2=15	L2=30	L2=45	L2=60
ZS1 (sec)	38.0	103.0	109.0	110.0
ZS2 (sec)	43.0	115.0	123.0	123.0
S (sec)	59.0	167.0	179.0	181.0
RJ1 (sec)	275.0	621.0	650.0	652.0
S' (sec)	345.0	1781.0	2818.0	3748.0

Table 32 Generating restricted partitions of 75 on SUN

Algorithm	L2=15	L2=30	L2=45	L2=60
ZS1 (sec)	3.2	8.5	9.1	9.1
ZS2 (sec)	3.5	9.5	10.2	10.3
S (sec)	4.1	11.8	13.0	13.0
RJ1 (sec)	28.8	64.9	68.0	68.0
S' (sec)	22.7	112.1	174.6	230.0

Table 33 Generating restricted partitions of 75 on NeXT

Algorithm	L2=15	L2=30	L2=45	L2=60
ZS1 (sec)	14.3	38.8	41.5	41.6
S (sec)	13.7	39.9	43.1	43.3
ZS2 (sec)	15.1	41.2	43.9	44.1
RJ1 (sec)	110.4	251.1	263.3	264.0
S' (sec)	89.2	443.1	689.5	906.0

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 rand() between 0 and 1 inclusively. Let p' = $\lfloor r * P(n,n) \rfloor$, referring to Table 2 in Chapter 1. RP(n,m) represents the number of restricted partitions using at most m parts. By locating p' in the table such that p'$\leq$ 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 * 77 = 20.71 (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 * 15 = 9.645 (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 * 5 = 4.295 (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 * 2 = .958 (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 (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 $S=\{s_1,s_2,...,s_p\}$ where $s_1 < s_2,...,s_p$ 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 $P_s'(n,k)=\sum^k_{l=1}P_s(n,l)$. 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(logN) 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 $P_s'(N,1),...,P_s'(N,p)$ (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 $\langle n,k\rangle$, 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 PE(n,m) of an integer n into exactly m parts is equal to the number of partitions PL(n,m) of the integer n into parts, the largest of which is m. There are PE(n,m) partitions with exactly m parts. The total number of parts in these partitions is m*PE(n,m). The total number of parts TP(n) is thus derived as

$$TP(n) = \sum_{m=1}^{n} mPE(n,m)$$

and the average number of parts denoted by AVRP(n) is equal to TP(n) / P(n). By implementing this formula (vide program PAVER), we get the following:

Table 34 Average number of parts in standard representation

n	P(n)	AVRP (n)	C	$n^{2/3}$	REL. ERROR
15	176	6.06	0.997	6.07	0.0016
30	5604	9.73	1.008	9.65	0.0082
45	89134	12.80	1.011	12.66	0.0109
60	966467	15.51	1.012	15.32	0.0122
75	8118264	17.99	1.011	17.79	0.0111
90	56634173	20.29	1.010	20.08	0.0103
105	342325709	22.45	1.009	22.24	0.0093
150	40853235548	28.35	1.001	28.32	0.0010
195	2580840231841	33.61	0.999	33.64	0.0008
240	105882249516398	38.42	0.994	38.65	0.0059
285	3160137919927934	42.90	0.990	43.33	0.0100
330	7365328464863616	47.12	0.986	47.78	0.0140
375	14062074454314071O4	51.12	0.983	52.00	0.0172
420	227552898053173045 76	54.94	0.979	56.11	0.0212
465	3201031270091951964 16	58.60	0.976	60.04	0.0245
510	3991268667606861086720	62.13	0.973	63.75	0.0260

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 reseach 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 $n = c_1y_1 + c_2y_2 +...+ c_dy_d \geq y_1 + y_2 +...+ y_d \geq d+...+2+1$

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

n	P(n)	TP(n)	AVRP (n)	C	$n^{4/9}$	REL. ERROR
15	176	508	2.89	0.867	3.33	0.1522
22	1002	3506	3.50	0.886	3.95	0.1285
30	5604	23025	4.11	0.907	4.53	0.1021
37	21637	99133	4.58	0.921	4.97	0.0851
45	89134	451501	5.07	0.935	5.42	0.0690
52	281589	1535914	5.45	0.942	5.78	0.0605
60	966467	5672882	5.87	0.951	6.17	0.0511
67	2679689	16644217	6.21	0.958	6.48	0.0434
75	8118264	53419131	6.58	0.966	6.81	0.0349
82	20506255	141227966	6.89	0.973	7.08	0.0275
90	56634173	409038376	7.22	0.978	7.38	0.0221
97	133230930	999764335	7.50	0.982	7.63	0.0173
105	342325709	2674789388	7.81	0.987	7.91	0.0128
112	761002156	6145056855	8.07	0.991	8.14	0.0086
120	1844349559	15425991887	8.36	0.996	8.39	0.0035

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)*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 $O(\sqrt{n}*P(n))$ (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*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 $O(n^{2/3}P(n))$.

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

```
/* ------------------------------------------------------------------- */
/* program    : fil c                                                  */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int n,y,i,pd,d,py,bb[1000],mm[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    memset(bb,'\0',1000); memset(mm,'\0',1000);
    mm[1]=n; bb[1]=1; d=1;

    /*
    printf("(%dX%d)\n",bb[1],mm[1]);
    */
    while (d > 0)
     {
if (bb[d] == 1 && mm[d] > 1)
   if (d > 1 && bb[d-1] == 2)
      if (mm[d] == 2)
 {
  bb[d-1]=2; mm[d-1] +=1; d --1;
 }
      else
 {
  bb[d-1]=2; mm[d-1] +=1; bb[d]=1; mm[d] -=2;
 }
   else if (mm[d] == 2)
    {
     bb[d]=2; mm[d]=1;
    }
else
    {
     bb[d+1]=1; mm[d+1]=mm[d]-2; bb[d]=2, mm[d]=1; d +=1;
    }
else if (bb[d] == 1)
if (d > 2 && (bb[d-2] == bb[d-1]+1))
   if (mm[d-1] == 1)
      {
       mm[d-2] +=1; d --2;
      }
   else
      {
                        mm[d-1]=bb[d-1]*(mm[d-1]-1),
                        mm[d-2] +=1; bb[d-1]=1, d --1;
      }
else if (mm[d-1] == 1)
{
 bb[d-1] --1; d --1;
}
     else
{
 mm[d]=bb[d-1]*(mm[d-1]-1),
 bb[d-1] --1; mm[d-1]=1; bb[d]=1;
}
     else if (mm[d] > 1)
     if ( d > 1 && (bb[d-1] == bb[d] + 1))
{
 mm[d]=bb[d]*(mm[d]-1)-1;
 mm[d-1] --1; bb[d]=1;
}
     else
{
 bb[d-1]=1; mm[d+1]=bb[d]*(mm[d]-1)-1;
 bb[d]++1;mm[d]=1; d +=1;
}
  else if (d > 1)
  if (d > 2 && (bb[d-2] == bb[d-1] + 1))
     {
      mm[d-1]=bb[d-1]*(mm[d-1]-1)+bb[d]-1,
      mm[d-2] --1; bb[d-1]=1; d --1;
     }
  else
     {
      mm[d]=bb[d-1]*(mm[d-1]-1)+bb[d]-1,
      bb[d-1] --1; mm[d-1]=1; bb[d]=1,
     }
       else d = 0;
/*  .
     for ( i = 1; i <= d; ++i)
    printf("(%dX%d) ",bb[i],mm[i]);
        printf("\n");
*/
    }
   scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------- */
/* program    : e.c                                                    */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int n,i,j,k,t,pp[1000],mm[1000];
long int count;

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    memset(pp,'\0',1000);memset(mm,'\0',1000);
    pp[0]=1,mm[0]=0,pp[1]=n+1;mm[1]=0;pp[2]=1;mm[2]=n,i=2,

     while (i>1)
     {
       /*
       for (i=2,i<=i;i++)
          printf("(%dX%d) ",pp[i],mm[i]);
       printf("\n");
       */
       a  = mm[i]*pp[i];

       if (mm[i] == 1)
          {
             i  -= 1;
             a  += mm[i]*pp[i];
          }

       if (pp[i-1]==pp[i]+1)
          {
             i      -= 1;
             mm[i] += 1;
          }
       else
          {
             pp[i] += 1;
             mm[i]  = 1;
          }

       if (a>pp[i])
          {
             pp[i+1] = 1,
             mm[i+1] = a - pp[i],
             i       -= 1;
          }
     }
  scanf ("%d",&i);
}
```

```
 mm[d-1] --1; bb[d]=1;
}
     else
{
 bb[d-1]=1; mm[d+1]=bb[d]*(mm[d]-1)-1;
 bb[d]++1;mm[d]=1; d +=1;
}
  else if (d > 1)
  if (d > 2 && (bb[d-2] == bb[d-1] + 1))
     {
      mm[d-1]=bb[d-1]*(mm[d-1]-1)+bb[d]-1,
      mm[d-2] --1; bb[d-1]=1; d --1;
     }
  else
     {
      mm[d]=bb[d-1]*(mm[d-1]-1)+bb[d]-1,
      bb[d-1] --1; mm[d-1]=1; bb[d]=1,
     }
       else d = 0;
/*  .
     for ( i = 1; i <= d; ++i)
    printf("(%dX%d) ",bb[i],mm[i]);
        printf("\n");
*/
    }
   scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------- */
/* program    : fe2.c                                                  */
/* ------------------------------------------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int  n,m,i,j,k,a,b,t,e,r,pp[1000],

main ()
 {
    for (i=1;i<=1000;++i)
        pp[i] = 1,

    printf("\n please enter the value of N : ");
    scanf ("%d", &n);pp[0]=-1;pp[1]=2,h=1;m=n-1;

    while (pp[1] != n)
     {
       if (m - h > 1)
          {
            h       += 1;
            pp[h]    = 2;
            m       -= 1;
          }
       else
          {
            j        = m - 2;
            a        = m - 1;
            e        = pp[a];
            r        = pp[m];
            while (pp[j] == e)
                {
                  pp[j]= 1;
                  r   += e;
                  j   -= 1;
                }
            h       = j + 1;
            pp[h]   = e + 1;
            pp[a]   = 1;
            if (a!=h)
                {
                  pp[a]-1;
                }
            m       = h + r - 1;
          }
         /*
         for (i=1;i<=m;++i)   (printf("%d ", pp[i]);)   printf("\n");
         */
       }
    scanf ("%d",&i);
  }
```

```
/* ------------------------------------------------------------------------ */
/* program   : aa.c                                                         */
/* ------------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int n,d,lim,rem,dif,i,yy[1003],cc[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    for (i=1; i < 1000; ++i)
     {
yy[i]=0.cc[i]=0;
     }
    yy[1]=n,cc[1]=1;d=1;

 /* printf("(%dX%d)\n",cc[1],yy[1]);  */

    while (yy[1] != 1)
     {
if (yy[d] == 1)
   {
     rem = yy[d-1]*cc[d];
     d  -= 1;
   }
else rem = yy[d];
lim = yy[d] - 1;
if (cc[d] > 1)
   {
     cc[d] -= 1;
     d     += 1;
   }
yy[d] = lim;
cc[d] = rem / lim;
dif   = rem - cc[d] * lim;
if (dif > 0)
   {
     d     += 1;
     yy[d] = dif;
     cc[d] = 1;
   }
/*
        for ( i = 1; i <= d; i++)
            printf("(%dX%d) ",cc[i],yy[i]);
        printf("\n");
*/
     }
   scanf ("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : bb.c                                                         */
/* ------------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int n,d,q,a,a1,a2,bb[1003],mm[1003];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    for (i=1; i < 1000; ++i)
     {
        bb[i]=0,mm[i]=0,
     }
    bb[1]=n,mm[1]=1;d=1;

 /* printf("(%dX%d)\n",bb[1],mm[1]);  */

    while (bb[1] != 1)
     {
if (bb[d] == 1)
   {
      q      = (mm[d]+1)/(bb[d-1]-1);
      a1     = q * (bb[d-1]-1);
      a      = (mm[d]+1) - a1;
      if (mm[d-1] == 1)
 if ( a == 0 )
   {
     bb[d-1] += 1;
     mm[d-1]  = q + 1;
     d       -= 1;
   }
 else
   {
     bb[d-1] += 1;
     mm[d-1]  = q + 1;
     bb[d]    = a;
     mm[d]    = 1;
   }
     else  if ( a == 0 )
{
  mm[d-1] += 1;
  bb[d]    = bb[d-1] - 1;
  mm[d]    = q + 1;
}
    else
{
  mm[d-1] += 1;
  bb[d]    = bb[d-1] - 1;
  mm[d]    = q + 1;
  d       += 1;
  bb[d]    = a;
  mm[d]    = 1;
}
  }
else if (mm[d] == 1)
if (bb[d] == 2)
  {
     bb[d] = 1;
     mm[d] = 2;
  }
else
  {
     bb[d] += 1;
     mm[d]  = 1;
     q      = 1;
     bb[d]  = 1;
     mm[d]  = 1;
  }
     else if (bb[d] == 2)
     {
       bb[d]  = 2;
       mm[d] += 1;
       d     += 1;
       bb[d]  = 1;
       mm[d]  = 2;
     }
   else
     {
       mm[d] += 1;
       d     += 1;
       bb[d]  = bb[d-1] - 1;
       mm[d]  = 1;
       d     += 1;
       bb[d]  = 1;
       mm[d]  = 1;
     }
 /*
 for ( i = 1; i <= d; i++)
     printf("(%dX%d) ",bb[i],mm[i]);
        printf("\n");
 */
     }
   scanf ("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : m1.c                                                         */
/* ------------------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define false 0
#define true  1

int m,n,i,k,t,s,pp[1000];

prntp ()
 {
  for (i=1;i<=k-1;++i) {printf("%d ",pp[i]);}
 }

main ()
 {
    memset(pp,"\0",1000);
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    pp[1]=n;k=1;
 /*
    printf("%d\n",pp[1]);
 */
    m = k;
    while (pp[1] != 1)
     {
t       = k - m;
k       = m;
pp[m] -= 1;
while ( pp[k] <= t )
  {
    t       -= pp[k];
    k       += 1;
    pp[k]   = pp[k-1];
  }
k       += 1;
pp[k]   = t + 1;
if (pp[m] != 1)      {  m  = k;    }
if (pp[m] == 1)      {  m -= 1;    }

/*
   for (i=1;i<=k;++i)     {printf("%d ",pp[i]);}  printf("\n");
*/

  }

   scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------------ */
/* program   : pw2.c                                                        */
/* ------------------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define false 0
#define true  1

int n,i,r1,p1,aa[100];

rp(r,p)
  int r,p;
  {
    int j,k;
    if (r!=3)
       {
         if (p==1 || r<=aa[p-1])
            {
              k=r;
            }
         else
            {
              k=aa[p-1];
            }

         for (j=k;j>=1;j--)
            {
             aa[p]=j;rp(r-j,p+1);
            }
       }
   else
       {
        /*
        for (i=1;i<p;i++) printf("%d ",aa[i]); printf("\n");
        */
       }
  }

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    memset(aa,"\0",100);r1=n;p1=1;

    rp(r1,p1);
    scanf ("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : pw1.c                                                        */
/* ------------------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define false 0
#define true  1

int n,i,m,j,r,p,q,aa[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    memset(aa,"\0",1000);
    aa[1]=n;p=1;
 /*
    printf("%d \n",aa[1]);
 */
a1:
     for (k=p;k>=1;k--)
      {
        if (aa[k]!=1)
           {
            aa[k] -=1;
            q      =aa[k];
            r      =p-k+1;
            for (j=k+1;j<=n;j++)
               {
                if (q<r)
                   {
                    aa[j]  =q;
                    r      =r-q;
                   }
               else
                   {
                    aa[j]  =r;
                    p      =j;
                    /*
                    for (i=1;i<=p;i++) {printf("%d ",aa[i]);} printf("\n");
                    */
                    goto a1;
                   }
               }
           }
      }
   scanf ("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : se1.c                                                        */
/* ------------------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int  m,n,i,h,t,s,r,pp[1000];

main ()
 {
    for (i=1;i<=1000;++i)  pp[i] = 1;

    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    pp[1]=n;n=1;m=1;

    /*
    printf("%d \n",pp[1]);
    */

    while (pp[1] != 1)
     {
       if (pp[h] == 2)
          {
            m +=1;pp[h] =1;h -=1;
          }
       else
          {
            t = n-h+1; pp[h] -=1;r=pp[h];

            while (t >= r)
               {
                 h +=1;pp[h]=r;t -=r;
               }

            if ( t != 0 )
               {
                 r=h+1;

                 if ( t > 1 )
                    {
                      h +=1;pp[h]=t;
                    }
               }
            else
               {
                 m=h;
               }
          }
       /*
       for (i=1;i<=m;++i)   {printf("%d ", pp[i]);}   printf("\n");
       */
     }
  scanf ("%d",&i);
 }
```

```
/*                                                                          */
/*                                                                          */
/*                                                                          */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <math.h>

#define false 0
#define true  1

int i,m,n,k,j,s,ii,jj,iq,mq,nq,pp[1000];

main ()
 {
   n  = 0;
   memset (pp,'\0',1000);
   printf("\n please enter the value of N: ");
   scanf("%d",&n);
   j=1;

/*
   printf(" %d\n",n);
*/
   pp[1]=n;
   for (i=2;i<=n;++i)              pp[i]=0;
a1:
   /*
      for (i=1;i<=n;++i)           printf("%d ",pp[i]);
      printf("\n");
   */
   j=1;
   while(pp[j]-1>0)                j++;
   if (j-1>0)
      {
        m = j-2;
        k = n;
        if (m!=0)
           {
             for (i=1;i<=m;++i) k-=pp[i];
           }
        jq=pp[j-1];jq++;jq=k/(pp[j-1]-1);
        for (iq=1;iq<=jq;++iq)  pp[j-2+iq]=iq-1;
        pp[j-1+jq]=k-jq*(iq-1);mq=j+jq;
        for (nq=mq;nq<=n;++nq)  pp[nq]=0;
        goto a1;
      }
  printf ("%d",61);
}
```

```
/* ----------------------------------------------------------------------- */
/* program   : [illegible]2.c                                               */
/* ----------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int m,n,i,k,ms;
long int l,p,pan,len,pp[100][100],tt[100];

main ()
 {
   memset(pp,"\0",10000);
   printf("\n enter N :");
   scanf ("%d",&i);

   pp[0][0] = 1;
   for ( n = 1; n <= i; n++ )
       {
pp[n][0] = 0;
for ( m = 1; m <= n; m++ )
     {
       if ( ( n - m ) < m )
 k = n - m;
       else
 k = m;
       pp[n][m] = pp[n][m-1] + pp[n-m][k];
     }
       }
  /*
       for ( n = 0; n <= i; n++ )
  {
    printf("\n");
    for ( m = 0; m <= i; m++ )
printf("%4d",pp[n][m]);
  }
       printf("\n");
  */

  for ( m = 1; m <= i; m++) tt[m] = 1;

  l=pp[i][i];p=1;
  while ( p < l )
     {
pan=p;n=i;len=0;

while ( n != 0 )
{
   len +=1;m=1;
   while ( pp[n][m] <= pan )  { m += 1; }
   tt[len]=m;pan=pan-pp[n][m-1];n=n-m;
}
p += 1;

     /*
       for ( k = 1; k <= len; k++ )
     printf("%ld ",tt[k]);
       printf("\n");
     */
    }
  scanf("%d",&i);
 }
```

```
/*
/* program   : 8.c
/*
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <time.h>

#define true  1
#define false 0

int n,nt,n2,y,i,pi,d,even=0,bb[1000],mm[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    for (i=1; i <= 1000; ++i) bb[i]=0,mm[i]=0;
    bb[1]=2,nt=n/2,mm[1]=nt;d=1;n2=n-(nt*2);
    if (n2 != 0 ) { bb[2]=1;mm[2]=n2;d=2;}
 /*
    printf("(%d%d%d) (%d%d%d)\n",bb[1],mm[1],bb[2],mm[2]);
 */
    while (d > 0)
     {
even=false;pi=mm[d]/2;
if ( mm[d] == ( pi * 2 ) )  { even = true; }
if ( (bb[1] == 1) && (d > 1) && (mm[d-1] == 1) )
   {
   if ( (d > 2) && (bb[d-2] == bb[d-1] + 1) )
      {
       if       (mm[d] == 1)
      {
       mm[d-2] += 1;
       d         -= 2;
      }
       else if (mm[d] == 2)
      {
       mm[d-2] += 1;
       bb[d-1]  = 1;
       mm[d-1]  = 1;
       d         -= 1;
      }
       else if ( even == false )
      {
       mm[d-2] += 1;
       bb[d-1]  = 2;
       mm[d-1]  = (mm[d] - 1) / 2;
       d         -= 1;
      }
       else if (even == true)
      {
       mm[d-2] += 1;
       bb[d-1]  = 2;
       mm[d-1]  = (mm[d] - 1) / 2;
       bb[d]    = 1;
       mm[d]    = 1;
      }
      }
 else
      {
       if  ( mm[d] == 1 )
      {
       bb[d-1] += 1;
       mm[d-1]  = 1;
       d         -= 1;
      }
       else if ( mm[d] == 2 )
      {
       bb[d-1] += 1;
       mm[d-1]  = 1;
       bb[d]    = 1;
       mm[d]    = 1;
      }
       else if ( even == false )
      {
       bb[d-1] += 1;
       mm[d-1]  = 1;
       bb[d]    = 2;
       mm[d]    = (mm[d] - 1) / 2;
      }
       else if ( even == true )
      {
       bb[d-1] += 1;
       mm[d-1]  = 1;
       bb[d]    = 2;
       mm[d]    = (mm[d] - 2) / 2;
       d         -= 1;
       bb[d]    = 1;
       mm[d]    = 1;
      }
      }
 }
 else if (bb[d] > 1)
      {
 if ( mm[d] == 1 )
      {
       mm[d]    = bb[d];
       bb[d]    = 1;
      }
 else
      {
       mm[d]   -= 1;
       d         += 1;
       bb[d]    = 1;
       mm[d]    = bb[d-1];
      }
      }
  else if (d > 1)
      {
       mm[d-1] -= 1;
       mm[d]    = mm[d] + bb[d-1];
       bb[d]    = 1;
      }
        else d = 0;

/*
for ( i = 1; i <= d; ++i)
    printf("(%d%d%d) ",bb[i],mm[i]);
        printf("\n");
*/

     }
    scanf ("%d",&i);
}
```

```
/*
/* program   : 8.c
/*
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <time.h>

int n,i,j,k,q,r,pp[1000],mm[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    memset(pp,"\0",1000);memset(mm,"\0",1000);
    pp[1]=-1;mm[1]=1;pp[0]=n;mm[1]=1;i=1;

    while (mm[i]!=n)
     {
        /*
        for (i=1;i<=i+1;++i) printf("(%d%d%d)",pp[i],mm[i]);
        */
        i=i-1;k=mm[i];r=mm[i]*pp[i];
        while ((pp[i]-pp[i])<2) { k +=mm[i];r+=mm[i]*pp[i];i--;i-- }

        if (mm[i]==1)
           {
            if (i!=3)
               {
                  r     += mm[i]*pp[i];
                  pp[i] += 1;
               }
            else
               {
                  i     = 1;
                  pp[i]  = 1;
               }
            mm[i]  = k;
            r      -= mm[i]*pp[i];
            i       = i + 1;
           }
        else
           {
            mm[i] -= 1;
            r     += pp[i];
            i     += 1;
            pp[i]  = pp[i-1]+1;
            mm[i]  = k;
            r     -= mm[i]*pp[i];
            i      = i + 1;
           }

        if (r==pp[i])
           {
             mm[i] += 1;
             i      = 1;
           }
        else
           {
             pp[i]  = r;
             mm[i]  = 1;
           }
     }
  scanf("%d",&i);
}
```

```
/*
/* program   : 8.c
/*
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <time.h>

int n,i,j,k,pp[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    memset(pp,"\0",1000);pp[0]=-1;pp[1]=n;i=1;

    while (i<=n)
     {
        /*
        for (i=1;i<=i;i++) printf("%d ",pp[i]); printf("\n");
        */
        i = i - 1;
        while ((pp[i]-pp[i])<2)  { i --;}
        if (i!=0)
           {
             for (j=i-1;j>=1;j--)  pp[j]=pp[i]+1;
           }
        else
           {
             for (j=1;j<=i;j++)    pp[j]=1;
             i --;
           }
        k=0;
        for (j=1;j<i;j++)          k +=pp[j];
        pp[i]=n-k;
     }
  scanf ("%d",&i);
}
```

```
/*
/* program   : 8.c
/*
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <time.h>

int n,i,j,k,pp[1000];

main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d", &n);
    memset(pp,"\0",1000);pp[0]=-1;pp[1]=n;i=1;

    while (i<=n)
     {
        /*
        for (i=1;i<=i;i++) printf("%d ",pp[i]); printf("\n");
        */
        i = i - 1;
        while ((pp[i]-pp[i])<2)  { i --;}
        if (i!=0)
           {
             for (j=i-1;j>=1;j--)  pp[j]=pp[i]+1;
           }
        else
           {
             for (j=1;j<=i;j++)    pp[j]=1;
             i --;
           }
        k=0;
        for (j=1;j<i;j++)          k +=pp[j];
        pp[i]=n-k;
     }
  scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------- */
/* program   : a.c                                                     */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define false 0
#define true  1

int a,i,j,k,q,n,aa[1000];

main ()
 {
   n=0; memset (aa,"\0",1000);
   printf("\n please enter the value of N K : ");
   scanf("%d %d",&n,&k);
   q=true;aa[1]=n;j=k+1;
   for (i=2;i<=k;++i)  {aa[i]=0;}
   while(q==true)
     {
        j=2;a=aa[1];
        while ((a<j) && (j<k))
           {
             a=a+j*aa[j];j +=1;
           }
        if (j==k)
           {
             q=false;
           }
        else
           {
             aa[j]+=1; aa[1]=a-j;
             for (i=2;i<=j-1;++i) {aa[i]=0;}
             /*
             for (i=n;i>=1;--i)
                 if (aa[i]!=0) {printf("(%dX%d)",i,aa[i]);}
             printf("\n");
             */
           }
     }
   scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------- */
/* program   : a'.c                                                    */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define false 0
#define true  1

int a,i,j,k,q,n,aa[1000];
long int num,zero;

main ()
 {
   n=0; memset (aa,"\0",1000);
   printf("\n please enter the value of N K : ");
   scanf("%d %d",&n,&k);
   q=true;aa[1]=n;j=k+1; num=zero=0;
   for (i=2;i<=k;++i)  {aa[i]=0;}
   while(q==true)
     {
        j=2;a=aa[1];
        while ((a<j) && (j<k))
           {
             a=a+j*aa[j];j +=1;
           }
        if (j==k)
           {
             q=false;
           }
        else
           {
             aa[j]+=1; aa[1]=a-j;
             for (i=2;i<=j-1;++i) {aa[i]=0;}

             for (i=k;i>=1;--i)
                 if (aa[i]==0) {zero+=1;}

             /*
             for (i=k;i>=1;--i)
                 if (aa[i]!=0) {printf("(%dX%d)",i,aa[i]);}
             printf("\n");
             */
           }
     }
   scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------------ */
/* program   : qle.c                                                        */
/* ------------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int n,e,i,t,q,r,u,i,a,pp[1003],mm[1000];

lexmin ()
 {
   if ( n >= e)
      {
        q=n/e;r=q*e;r=n-r;
        if ( r != 0 )
           {
             u ++1;pp[u]=q+1;mm[u]=r;
           }
        if ( r != e && q !=0 )
           {
             u ++1;pp[u]=q;mm[u]=e-r;
           }
      }
 }


main ()
 {
    printf("\n please enter the value of N : ");
    scanf ("%d",&n);
    t=n;u=0;
        for (a=1; a<=t; ++a)
     {
        u=0;n=t;e=a;
        /*
        memset(pp,'\0',1000);
        memset(mm,'\0',1000);
        */
        lexmin();
        while (pp[1] <= t)
          {
            /*
            for (i=1;i<=u;++i)
            printf("(%d%d) ",pp[i],mm[i]);
            printf("\n");
            */

            if (u==1 && mm[1] == 1)             { goto a1; }
            if (pp[1] == 1)                     { goto a1; }
            if (pp[u] == 1)
               {
                if (mm[u-1] > 1)
                   {
                     n          = mm[u-1]*pp[u-1]+mm[u]*pp[u]-pp[u-1]-1;
                     e          = mm[u-1]+mm[u]-1;
                     u          -= 1;
                   }
                 else if (u == 2)
                        {
                          goto a1;
                        }
                      else
                        {
                          n       = mm[u-2]*pp[u-2]+mm[u-1]*pp[u-1];
                          n       = n+mm[u]*pp[u]-pp[u-2]-1;
                          e       = mm[u-2]+mm[u-1]+mm[u]-1;
                          u       -= 2;
                        }
               }
            else if (mm[u] > 1)
                   {
                     n        = mm[u]*pp[u]-pp[u]-1;
                     e        = mm[u]-1;
                   }
                 else
                   {
                     n        = mm[u-1]*pp[u-1]+mm[u]*pp[u]-pp[u-1]-1;
                     e        = mm[u-1]+mm[u]-1;
                     u        -= 1;
                   }
             mm[u]  = 1;
             pp[u]  += 1;
             if (u>1 && pp[u]==pp[u-1])
                {
                  u     -=1;
                  mm[u] +=1;
                }
             lexmin();
          } /* while pp[1] <= t  */
        a1:;
     } /* for a=1 ... */
    scanf ("%d",&i);
}
```

```
/* ------------------------------------------------------------------------ */
/* program   : rjl.c                                                        */
/* ------------------------------------------------------------------------ */

#include <std.o.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int f1,k,i,j,s,m,n,l1,l2,xx[1000],yy[1002],z1,m1,n1,l11,l22;
long int n,m;

prnte ()
 {
  for ( k = 1; k <= n; k++)
      printf (" %d",xx[k]);
  printf("\n");
 }

main ()
 {
   memset(yy,'\0',1000);memset(xx,'\0',1000);

   printf("a[0-arb, 1- dia] m-integer  n-parts  11&12 (bounds)\n");
   printf("enter a m l1 l2 n: \n");
   scanf ("%d %d %d %d %d",&a, &m, &l1, &l2, &n);
   z1=z;r1=m;n1=n;l11=l1;l22=l2;
   for ( n1 = m1; n1 >=1; n1--)
     {
        r=n1;z=z1;l1=l11;l2=l22;n=n1;
        j=z*n*(n-1);m=m-(n*l1)-(j/2);l2=l2-l1;
        if ( m >= 0 && m <= (n * l2) - j )
           {
             for ( i = 1; i <= n; i++)  xx[i]=yy[i]=l1+z*(n-i);
             i=i;l2=l2-z*(n-1);
   a1:       if ( m > l2 ) { m =m-l2;xx[i]=yy[i]+l2;i ++1;goto a1; }
             xx[i]=yy[i]=m;
             /*
             prnte();
             */
             if ( i < n && m > 1 )
                {
                  m=1;xx[i] --1;i ++1;xx[i]=yy[i]+1;
                  /*
                  prnte();
                  */
                }
              for (j = i - 1; j >= 1; j--)
                {
                  l2=xx[j]-yy[j]-1;m +=1;
                  if ( m <= ( n - j ) * l2 ) { xx[j]=yy[j]+l2;goto a1; }
                  m=m-l2;xx[i]=yy[i];i=j;
                }
           }
     }
  scanf("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : w.c                                                          */
/* ------------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int m,n,i,j,k,r,l,h;
long int q,p,pan,len,pp[100][100],tt[100];

main ()
 {
   memset(pp,'\0',10000); printf("enter N K :\n"); scanf ("%d %d",&n,&k);
   for ( l = 0; l <= n; l++ )
       for ( m = 0; m <= 1; m++)
   pp[l][m] = 0;
   pp[0][0] = 1;
   for ( l = k; l <= n; l++ )
       {
for ( m = k; m <= l; m++ )
    {
      if ( ( l - m ) < m )
 r = l - m;
      else
 r = m;
      pp[l][m] = pp[l][m-1] + pp[l-m][r];
          }
       }
  /*
  for ( i = 0; i <= n; i++)
     {
printf("\n");
for ( j = 0; j <= n; j++)  printf("%3d",pp[i][j]);
     }
  */
  q=pp[n][n]; p=0;
  while ( p < q )
    {
       pan=p;l=n;len=0;
a1:    len ++1;m=k;
b1:    if ( pp[l][m] < pan )
          {
            m ++1; goto b1;
          }
else if ( pp[l][m] > pan )
          {
c1:         tt[len]=m;pan=pan-pp[l][m-1];l=l-m;
            if ( l < k )  goto d1; else goto a1;
          }
        else
          {
            m  += 1;
            if ( m == l ) goto c1; else goto b1;
          }
d1:  p ++1;
    /*
       for ( h = 1; h <= len; h++ )
           printf("%ld ",tt[h]);
       printf("\n");
    */
    }
  scanf("%d",&i);
 }
```

```
/* ------------------------------------------------------------------- */
/* program   : ej2.c                                                   */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int fl,k,i,j,m,n,r,z,ll,lr,h;
int zl,ml,nl,rl,vv[1000],mm[1000],xx[1000],yy[1000],ii[1000];

prnts ()
 {
  for ( h = 1; h <= n; h++)
    printf (" %d",xx[h]);
  printf("\n");
 }

main ()
  {
   memset(vv,'\0',1000);memset(mm,'\0',1000);memset(xx,'\0',1000);
   memset(yy,'\0',1000);memset(ii,'\0',.000);
/*
   for ( i = 1; i <= 1000; i++)
       {
         vv[i]=0;mm[i]=0;xx[i]=0;yy[i]=0;ii[i]=0;
       }                                          */

   printf("z(0=exb, 1= dia) m=integer  n=parts r=# of elements\n");
   printf("enter z m n r: \n");
   scanf ("%d %d %d %d",&z, &m, &n, &r);
   zl=z;ml=m;nl=n;rl=r;

   /*
   printf("enter i v[i] \n");
vi;
   scanf ("%d %d",&i,&j);
   if ( i != 0 ) { vv[i] = j;goto vi; }
   */

   for ( i = 1; i <= m; i++) vv[i] = i;
   for (nl = ml; nl >= 1; nl--)
      {
        m=ml;n=nl;z=zl;r=rl;
        if ( z == 0 )
           {
             for ( i = 1; i <= r; i++) mm[i] = n;
           }
        else
           {
             for ( i = 1; i <= r; i++) mm[i] = 1;
           }
      }
   j=1;k=mm[1];ll=vv[1];
   for ( i = n; i >= 1; i-- )
     {
       xx[i]=yy[i]=ll;k --1; m --ll;
       if ( k == 0 )
          {
            if ( j == r )   { goto b3; }
            j  +=1;k =mm[j];ll=vv[j];
          }
     }
   lr=vv[r];ll=vv[1];
   if ( m < 0 || m > ( n * ( lr - ll )))       { goto b3; }

   if ( m == 0 ) { /* prnts(); */                goto b3; }
   i=1;m --yy[1];
b1;
   for ( j = mm[r]; j >= 1; j--)
       {
         if ( m <= lr )                          { goto b2; }
         xx[i]=lr;ii[i]=r-1;i ++1;m=m-lr+yy[i];
       }
   r --1;
b2;
   j = r;
   while (vv[r] > m) { r --1; }
   lr=vv[r];
   if ( m == lr )
      {
        xx[i] = lr;
        /*
        prnts ();
        */
        r --1;lr=vv[r];
      }
   k =yy[i];
   if ( lr > k  && ( m - lr ) <= ( n - i ) * ( lr - ll ) )
       goto b1;
   else xx[i] = k;

   for ( i = i - 1; i >= 1; i--)
       {
         r=ii[i];lr=vv[r]; m=m+xx[i]-k; k =yy[i];
         if ( lr > k && (m - lr) <= ( n - i ) * ( lr - ll ) )
              goto b1;
         else xx[i] = k;
       }
b3;;
  }    /*       for nl = ml          */
  scanf("%d",&i);
 }
```

```
/* ------------------------------------------------------------------- */
/* program   : rj2.c     (counting the instructions)                    */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int f1,k,i,j,m,n,r,s,ll,lr,h,ml,nl,rl,vv[1000],mm[1000],nn[1000],yy[1000],ii[
long int num,at1,at2,at3,at4,at5,at6;

prnta ()
 {
  for ( h = 1; h <= n; h++) printf (" %d",nn[h]);
  printf("\n");
 }

main ()
 {
     for ( i = 1; i <= 1000; i++)
       {
         vv[i]=0;mm[i]=0;nn[i]=0;yy[i]=0;ii[i]=0;
       }

   printf("s(0-srb, 1- dia) m=integer  n=parts r=# of elements\n");
   printf("enter s m n r; \n");
   scanf ("%d %d %d %d",&s, &m, &n, &r);
   ml=m; nl=n; rl=r;

/* printf("enter i v[i] \n");
v1:
   scanf ("%d %d",&i,&j);
   if ( i != 0 )
      {
vv[i] = j;
goto v1;
      }                          */

   for ( i = 1; i <= m; i++)
       vv[i] = i;

   at1=at2=at3=at4=at5=at6=0;
   at6=3;
   num = 0;
   for (nl = ml; nl >= 1; nl--)
   {
   at1 += 1;
   at6 += 1;
   m = ml;
   n = nl;
   s = sl;
   r = rl;
   at6 += 4;
   if ( s == 0 )
     {
at3 += 1;
for ( i = 1; i <= r; i++)
  {
            at1    += 1;
            at6    += 2;
            mm[i] = n;
  }
     }
   else
     {
          {
at4 += 1;
for ( i = 1; i <= r; i++)
 {
    at1    += 1;
    at6    += 2;
    mm[i] = 1;
 }
      }

   j    = 1;
   k    = mm[1];
   ll   = vv[1];
   at6 += 3;

   for ( i = n; i >= 1; i-- )
     {
nn[i] = yy[i] = ll;
k      += 1;
n      -= ll;
at1    += 3;
at6    += 5;
if ( k == 0 )
   {
     at3 += 1;
     if ( j == r )   { at3 += 1;at6 += 1; goto b3; }
     at5 += 1;
     j    += 1;
     k    = mm[j];
     ll   = vv[j];
     at1 += 1;
     at6 += 3;
   }
else at5 += 1;
     }
   lr   = vv[r];
   ll   = vv[1];
   at6 += 2;
   if ( m < 0 || m > ( n * ( lr - ll ))) { at1+=2;at3+=1;at6+=1;goto b3; }
   at5 += 1;
   at1 += 2;
   if ( m == 0 ) {   num = 1;   /* prnta(); */    at3+=1;at6+=1;goto b3; }
   at5 += 1;
   i    = 1;
   m   += yy[1];
   at1 += 1;
   at6 += 2;

b1:
    for ( j = mm[r]; j >= 1; j--)
       {
 at6    += 1;
 at1    += 1;
 if ( m <= lr )                    { at3+=1;at6+=1;goto b2; }
 at5    += 1;
 nn[i]  = lr;
 ii[i] = r - 1;
 i      += 1;
 m      = m - lr + yy[i];
 at1    += 4;
 at6    += 4;
       }
   r   -= 1;
   at1 += 1;
   at6 += 1;
b2:
   j    = r;
   at6 += 1;
   while (vv[r] > m)
     {
 at2 += 1;
 r   -= 1;
 at4 += 1;
 at1 += 1;
     }
   lr   = vv[r];
   at6 += 1;
   if ( m == lr )
     {
nn[i] = lr;
num    += 1;
       /*
        prnta ();
        */
r      -= 1;
lr     = vv[r];
at3    += 1;
at1    += 1;
at6    += 3;
     }
   else at5    += 1;
   k    = yy[i];
   at6 += 1;
   if ( lr > k  && ( m - lr ) <= ( n - i ) * ( lr - ll ) )
     {
at3 += 1;
at1 += 4;
at6 += 1;
goto b1;
     }
   else
     {
at4 += 1;
at6 += 1;
nn[i] = k;
     }

   for ( i = i - 1; i >= 1; i--)
     {
 at6 += 1;
 at1 += 1;
 r    = ii[i];
 lr   = vv[r];
 m    = m > nn[i] - k;
 k    = yy[i];
 at1 += 2;
 at6 += 4;
 if ( lr > k && (m - lr) <= ( n - i ) * ( lr - ll ) )
   {
     at3 += 1;
     at1 += 4;
     at6 += 1;
     goto b1;
   }
 else
   {
     at4    += 1;
     at6    += 1;
     nn[i] = k;
   }
     }

b3:;
  /*       for ml = ml          */
  printf("number of arithmetic instructions %ld\n",at1);
  printf("  ''    '' while          ''       %ld\n",at2);
  printf("  ''    '' if then        ''       %ld\n",at3);
  printf("  ''    '' if else        ''       %ld\n",at4);
  printf("  ''    '' if not used    ''       %ld\n",at5);
  printf("  ''    '' other          ''       %ld\n",at6);
  printf("       total       instructions %ld\n",at1+at2+at3+at4+at5+at6);
  scanf("%d",&i);
 }
```

```
/* ------------------------------------------------------------------------ */
/* program   : paver.c                                                      */
/* ------------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

int m,n,i,k;
float pp[311][311];
double p1;
double f1,f2,averg;

main ()
 {
   for ( n = 0; n <= 311; ++n)
       for ( m = 0; m<= 311; ++m)
           pp[n][m] = 0;

   printf("enter N : ");
   scanf ("%d",&i);

   pp[0][0] = 1;
   for ( n = 1; n <= i; n++ )
      {
        pp[n][0] = 0;
        for ( m = 1; m <= n; m++ )
            {
                  if ( m > n ) pp[n][m] = 0;
             else if ( m == 0) pp[n][m] = 0;
             else if ( m == 1) pp[n][m] = 1;
             else    pp[n][m] = pp[n-1][m-1] + pp[n-m][m];
            }
      }
    /*
    for (n=0;n<=i;n++)
        {
          printf("\n");
          for ( m=0;m<=i;m++)
              printf("%3d ",pp[n][m]);
        }
    printf("\n");
    */

    p1=0;
    f1=f2=averg=0;
    for (n=1;n<=i;n++)
        {
         p1 = p1+ (double) n * (double) pp[i][n];
         f1 = f1 + (double) pp[i][n];
        }
    averg = p1/f1;
    printf(" number of partitions %11.2f \n",f1);
    printf(" total  parts         %11.2f \n",p1);
    printf(" average of parts     %11.6f  \n",averg);
    scanf("%d",&i);
 }
```

Bibliography

[A] S.G. Akl, A comparison of combination generation methods, ACM Trans. on Math. Software, 7,1, 1981, 42-45.

[An] G.E. Andrews, The theory of partitions, Addison-Wesley,Reading, Ma, 1976.

[AS] S.G. Akl and I. Stojmenovic, Parallel algorithms generating integer partitions and compositions, J. of Combinatorial Mathematics and Combinatorial Computing, Vol. 13, April 1993, 107-120.

[B] T. Brylawski, The lattice of integer partitions, Discrete Math., 6, 1973, 201-209.

[Be] C. Berge, Principles of Combinatorics, Academic Press, 1971.

[BH] T. Beyer, and S. M. Hedetniemi, Constant time generation of rooted trees, SIAM J. Comput., 9, 4, 1980, 706-712.

[BS] M. Belbaraka and I. Stojmenovic, On generating B-trees with constant average delay and in lexicographic order, Technical Report TR-92-36, Computer Science Department, University of Ottawa, October 1992.

[D] L.E. Dickson, History of the theory of numbers, Vol. II, Diophantine Analysis, Chelsea Publishing Co., New York, 1971.

[DMSSS] B. Djokic, M. Miyakawa, S. Sekiguchi, I. Semba, I. Stojmenovic, A fast algorithm for generating set partitions, The Computer J., 32, 3, 1989, 281-282.

[FL] T.I. Fenner and G. Loizou, A binary tree representation and related algorithms for generating integer partitions, The Computer J., 23, 4, 1980, 332-337.

[FL2] T.I. Fenner and G. Loizou, Tree traversal related algorithms for generating integer partitions, SIAM J. Computing, 12, 3, 1983, 551-564.

[FL3] T.I. Fenner and G. Loizou, An analysis of two related loop-free algorithms for generating integer partitions, Acta Informatica 16, 1981, 237-252.

[GLW] U.I. Gupta, D.T. Lee and C.K. Wong, Ranking and unranking of B-trees, J. of Algorithms, 4, 1983, 51-60.

[GR] A. Gibbons and W. Rytter, Efficient parallel algorithms, Cambrige University Press, 1988.

[Ha] M. Hall, Combinatorial Theory, Blaisdell, Waltham, Mass., 1967.

[Hi] T. Hikita, Listing and counting subtrees of equal size of a binary tree, Inf. Proc. Lett., 17, 1983, 225-229.

[HR] G.H. Hardy and S. Ramanujan, Asymptotic formulae in combinatory analysis, Proc. London, Math. Soc., 17, 1918, 237-252.

[JR] K.R. James and W. Riha, Algorithm for generating graphs of a given partition, Computing 16, 1976, 153-161.

[Kn] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Ma, 1968.

[Le] D.H. Lehmer, The machine tools of combinatorics, in Applied Combinatorial Mathematics, Beckenbach (ed.), Wiley, NY, 1964.

[Li] C.L. Liu, Introduction to Combinatorial Mathematics, McGraw Hill, 1968.

[Mk] J.K.S. McKay, Partition generator, Alg. 263, CACM, 8, 1965, p.493.

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

[Mk"] J.K.S. McKay, Map of partitions into integers, Alg. 264, CACM, 8, 1965, p.493.

[Mk1] J.K.S. McKay, Partitions in natural order, Alg. 371, CACM, 13, 1970, p.52.

[NMS] T.V. Narayana, R.M. Mathsen, and J. Saranji, An algorithm for generating partitions and its applications, J. Comb. Theory, 11, 1971, 54-61.

[NW] A. Nijenhius, H.S. Wilf, Combinatorial Algorithms, Academic Press, NY, 1975.

[NW1] A. Nijenhius, H.S. Wilf, A method and two algorithms on the theory of partitions, J. Comb. Theory A, 18, 1975, 219-222.

[PW] E.S. Page and L.B. Wilson, An Introduction to Computational Combinatorics, Cambridge Univ. Press, 1979.

[Re] R.C. Read, A survey of graph generation techniques, Lect. Notes in Math., 884, 1980, 77-89.

[Ri] J. Riordan, An introduction to Combinatorial Analysis, John Wiley, 1958.

[RJ] W. Riha, and K.R. James, Efficient algorithms for doubly and multiply restricted partitions, Algorithm 29, Computing, 16, 1976, 163-168.

[RND] E.M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms, Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[Ru1] F. Ruskey, Generating t-ary trees lexicographically, SIAM J. Comput., 7, 1978, 492-509.

[Ru2] F. Rubin, Partition of integers, ACM Transactions on mathematical Software, Vol.

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

[S] F. Stockmal, Generation of partitions in part-count form, Algorithm 95, CACM 5, 1962, p. 344.

[S1] L.Sanchis, Counting and generating integer partitions in parallel, Proc. ICCI'92, 54-57.

[Sa] C.D. Savage, Gray code sequences of partitions, J. of Alg., 10, 1989, 577-595.

[Se] R.Sedgewick, Permutation generation methods, Comp. Surv. 9, 1977, 137-164.

[Ss] S.Skiena, Implementing Discrete Mathematics, State University of New York, Stony Brook, 1990, 51-59.

[T] C.Tomasi, Two simple algorithms for the generation of partitions of an integer, Alta frequenza, VOL.LI - N.6 NOV - DEC. 1982, 352-356.

[VB] D.R.Van Baronaigien, A loopless algorithm for generating binary tree sequences, Information Processing letters 39, 1991, 189-194.

[W] M.B. Wells, Elements of Combinatorial Computing, Pergamon Press, Oxford, 1971.

[Wh] J.S. White, Restricted partition generator, Alg. 374, CACM, 13, 1970, p 120.

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

[ZR] S. Zaks and D. Richards, Generating trees and other combinatorial objects lexicographically, SIAM J. on Comput., 8, 1, 1979, 73-81.

[ZS] A.Zoghbi and I.Stojmenovic, Fast algorithms for generating integer partitions, Technical Report TR-93-05, Computer Science Department, University of Ottawa, February 1993.8