New results in partition theory.

Luat. Bui
University of Windsor

NEW RESULTS

IN PARTITION THEORY

BY

LUAT BUI

A Thesis

Submitted to the Faculty of Graduate Studies and Research

through the Department of Mathematics and Statistics

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

1997

0-612-30935-5

To Chau Diep.

ABSTRACT

This thesis includes a brief review of the literature of methods of how partition identities may be derived.

In the third and fourth chapters we introduce a totally new and different approach in partition theory. With the use of computers we will apply the method of sampling and simulation to estimate the number of partitions of an integer. We give an algorithm for generating partitions so that every partition is equally likely to be generated. We use Capture-Recapture method and an estimation technique of Boneh, Boneh, and Caron.

In chapter five we give some new results in partition theory, using generating functions.

The last chapter summarizes the findings of this thesis and compares these findings with known results.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Partition theory is an area of additive number theory, a subject concerning the representation of integers as sums of other integers. The general problem of the theory may be stated as follows (Hardy and Wright, 1960).

Suppose A is a set of positive integers. Let n be a positive integer of the form $n = a_1 + a_2 + \dots + a_m$ where m may be fixed or unrestricted and the a_i's are taken from A, may or may not be distinct, and their order is irrelevant. Let $P(n)$ be the number of such representations of n. What can we say about $P(n)$? What properties does it have ? How large is it ? Is there always at least one such representation for each positive integer n ?

Given an integer n, let $R_1(n)$ be the total number of partitions of n restricted to some conditions. For example, the restrictions may be that the partitions contain only even parts, or only odd parts, or the partitions are restricted to parts that are at most k for some integer k, or the partitions are restricted to distinct parts only. Let $R_2(n)$ be the total number of partitions of n restricted to some other conditions. How is $R_1(n)$ related to $R_2(n)$? How large is each ? And how are they related to $P(n)$? These are questions that are studied in partition theory.

If we take the set A to be the set of all positive integers, with the a_i's unrestricted, where repetitions are allowed, we have the standard definition for a partition of a positive integer.

A *partition* of a nonnegative n integer is a representation of n as a sum of positive integers, called summands or parts. We will denote the total number of partitions of n by $P(n)$. $P(0)$ is defined to be 1.

G. W. Liebnitz (1646 - 1716) was among the first mathematicians who paid attention in the development stages in this area of mathematics (Griffin, 1954), but the greatest contributions in the early stages of partition theory were due to L. Euler (1707 - 1783) (according to Andrews, 1971). Over the centuries a great number of mathematicians had devoted their time in a search for new identities in partition theory and to find a formula for $P(n)$. Further developments in partition theory in the early 20th century were due to G. H. Hardy (1877 - 1947) and S. Ramanujan (1887 -1920), and in 1917 they showed that $P(n)$ is asymptotic to $\frac{1}{4n\sqrt{3}}e^{\pi\sqrt{\frac{2n}{3}}}$ (Cohen, 1978).

In chapter II of this thesis we summarize some methods in which the identities in partition theory may be derived.

In chapters III and IV of this thesis we will suggest a new approach for estimating $P(n)$. Instead of searching for a formula for $P(n)$, we will use simulation and estimation methods to estimate the number of partitions of a given integer n. In chapter III we will use the Capture-Recapture method and in chapter IV we will use a Prediction Function to estimate $P(n)$. We will not be concerned with the developments of the methods, but rather, we will apply the methods to simulated data.

In chapter V we develop some new formulas in partition theory, using generating functions.

Some of the books we found useful in our study of partitions include Andrews (1971), Cohen (1978), Griffen (1954), Hardy and Wright (1954), Niven and Zuckerman (1966), Riordan (1958) and Sloane and Plouffe (1995).

<u>CHAPTER II</u>

A SURVEY OF METHODS USED IN PARTITION THEORY

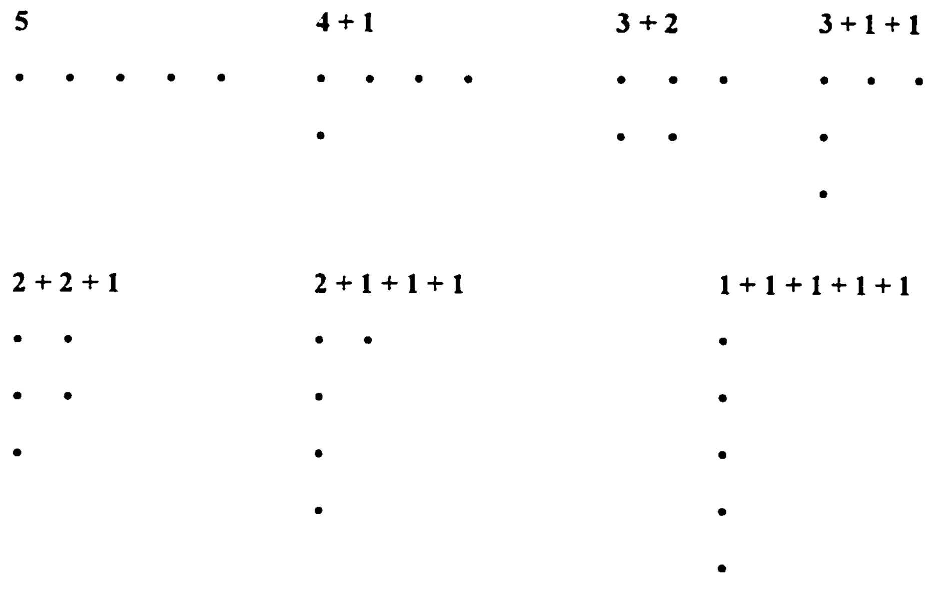
In this chapter we investigate three methods which can be used to derive partition identities, the graphical representation method, the discovery method and the generating function method. In chapter V we will use generating functions to obtain some new identities.

II.(i) The Method of Graphical Representation

The material in this section may be found in Andrews, 1971.

A *graphical representation* of a partition is the representation of the partition by horizontal dots.

Example: The graphical representations of the partitions of 5 are:



Notice that if we read the columns instead of the rows of graphical representation we still have a partition. This gives rise to the concept of conjugate partitions.

The *conjugate partition* of a given partition is formed by reading the number of dots in successive columns (instead of rows) in the graphical representation.

Example: The conjugate of the partition **5 + 4 + 1** of 10 is **3+2+2+2+1** since the graphical representation corresponding to **5 + 4 + 1** is

```
• • • • •
• • • •
•
```

Observe that the conjugate of the conjugate is the partition itself.

When the partition is identical to its conjugate we say that the partition is *self-conjugate*. We note that the graphical representation of a self-conjugate partition is symmetrical about the diagonal line.

Example: the partition **5+2+1+1+1** of 10 is a self-conjugate partition since its graphical representations are respectively

```
• • • • •
• •
•
•
•
```

We now demonstrate some uses of the graphical representation of partitions.

Theorem 1 (Andrew, 1971): Let $p_m(n)$ denote the number of partitions of n with at most m parts, and let $\pi_m(n)$ denote the number of partitions of n with no parts greater than m. Then $p_m(n) = \pi_m(n)$.

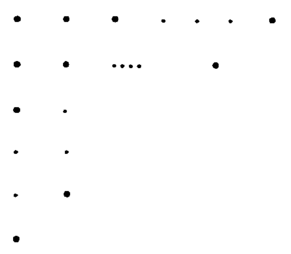
Proof: Let us consider an arbitrary partition of n in which at most m parts appear. Then the conjugate of such a partition has no parts larger than m since there would be at most m dots in any column of the graphical representation of the original partition.

This pairing of each partition of n of at most m parts with its conjugate, a partition of n in which no parts are larger than m, establishes a one-to-one correspondence between the two types of partitions. Hence, there must be the same number of each type; that is, $p_m(n) = \pi_m(n)$. ■

The following is a simple new result about self-conjugate partitions.

Property 1: Let k and n be positive integers such that $k = n^2$. Then k can be written as a sum of two self-conjugate partitions, one with the largest part equal to n and the other with the largest part less than or equal to n-1.

Proof: Suppose k and n are positive integers such that $k = n^2$. Take the graphical representation a self-conjugate partition with the largest part equal to n and complete the square using dots.



Then the added dots would be symmetric about the diagonal line, thus they form a self-conjugate partition with the largest part less than or equal to n-1. ■

II.(ii) **The Discovery Method**

In this section we will introduce a "discovery" method for finding partition identities. This method is not a method of proof, rather it is a method that helps discover partition identities (Andrews, 1971).

Let S be an initially unknown set of positive integers. Let $P(S,n)$ denote the number of partitions of n with each summand in S. Now, suppose $D(n)$ is the total number of partitions of n with some restrictions. The objective of this discovery method is to inductively determine whether a set S exists such that $P(S,n) = D(n)$, and if it exists, what properties does it have? Some of the well known identities in partition theory such as *Euler's Partition Theory Identity* (the number of partitions of n into odd parts equals the number of partitions of n into distinct parts) and *the First Rogers-Ramanujan Identity* (the number of partitions of n in which any two parts differ by at least two equals the number of partitions of n into parts congruent to either 1 or -1 modulo 5) can be discovered by this method. We now show by example how this method works.

Let $D_I(n)$ denote the number of partitions of n in which any two parts differ by at least 2 and no consecutive odd parts occur as summands. Our goal is to determine the existence and properties of a set S_I such that $D_I(n) = P(S_I,n)$ for all n.

Now, since $D_I(1) = 1$, therefore $1 \in S_I$. Since $D_I(2) = 1$, therefore $2 \notin S_I$ (otherwise we would have $P(S_I,2) = 2$). Since $D_I(3) = 1$, therefore $3 \notin S_I$ (otherwise we would have $P(S_I,3) = 2$). Continuing in this way, the first 7 elements of S_I are shown in Table 2.1.

<u>*Table 2.1*</u>: Discovering the properties of S_I

n	$D_I(n)$	$P(S_I,n)$ if $n\in S_I$	$P(S_I,n)$ if $n\notin S_I$	conclusion	$S_I\cap\{1,2,\ldots,n\}$
1	1	1	0	$1\in S_I$	$\{1\}$
2	1	2	1	$2\notin S_I$	$\{1\}$
3	1	2	1	$3\notin S_I$	$\{1\}$
4	1	2	1	$4\notin S_I$	$\{1\}$
5	2	2	1	$5\in S_I$	$\{1,5\}$
6	3	3	2	$6\in S_I$	$\{1,5,6\}$
7	3	4	3	$7\notin S_I$	$\{1,5,6\}$
8	3	4	3	$8\notin S_I$	$\{1,5,6\}$
9	4	4	3	$9\in S_I$	$\{1,5,6,9\}$
10	5	6	5	$10\notin S_I$	$\{1,5,6,9\}$
11	6	7	6	$11\notin S_I$	$\{1,5,6,9\}$
12	7	8	7	$12\notin S_I$	$\{1,5,6,9\}$
13	8	8	7	$13\in S_I$	$\{1,5,6,9,13\}$
14	10	10	9	$14\in S_I$	$\{1,5,6,9,13,14\}$
15	12	13	12	$15\notin S_I$	$\{1,5,6,9,13,14\}$
16	13	14	13	$16\notin S_I$	$\{1,5,6,9,13,14\}$
17	15	15	14	$17\in S_I$	$\{1,5,6,9,13,14,17\}$

If we continue further we would see that the set S_I exists and consists of exactly the integer congruent to 1, 5 or 6 modulo 8. Thus, we may conjecture that the number of partitions of n in which any two parts differ by at least 2 and no consecutive odd numbers occur as parts is equal to the number of partitions of n into parts congruent to 1, 5, or 6 modulo 8.

In the above example we saw that the sets S_1 exists. This is not true in general. For example, let $D_2(n)$ denote the number of partitions of n in which any two parts differ by at least 3, by the same process as above Table 2.2 shows that there does not exist a set S_2 of positive integers such that $P(S_2,n) = D_2(n)$ for all n.

Table 2.2: Discovering the properties of S_2

n	$D_2(n)$	$P(S_2,n)$ if $n\in S_2$	$P(S_2,n)$ if $n\notin S_2$	conclusion	$S_2\cap\{1,2,\ldots,n\}$
1	1	1	0	$1\in S_2$	$\{1\}$
2	1	2	1	$2\notin S_2$	$\{1\}$
3	1	2	1	$3\notin S_2$	$\{1\}$
4	1	2	1	$4\notin S_2$	$\{1\}$
5	2	2	1	$5\in S_2$	$\{1,5\}$
6	2	3	2	$6\notin S_2$	$\{1,5\}$
7	3	3	2	$7\in S_2$	$\{1,5,7\}$
8	3	4	3	$8\notin S_2$	$\{1,5,7\}$
9	4	4	3	$9\in S_2$	$\{1,5,7,9\}$
10	4	6	5	???	???

Observe that on the last line of Table 2.2, $P(S_2,n)\neq D_2(n)$ whether 10 belongs to S_2 or not. Hence, such a set does not exist.

II.(iii) Generating Functions

The use of generating functions is a very powerful technique with numerous applications in number theory and combinatorics. We will use techniques from this section in chapter V. If $A = \{a_1, a_2, a_3, \ldots\}$ is a sequence of real numbers, then

$G(A,x) = \sum_{n=1}^{\infty} a_n x^n$ is called the generating function for the sequence A.

We will use $G(P(n),x)$ to represent the generating function for $\{P(1), P(2), \ldots\}$. Suppose $D(n)$ is the number of partitions of n with some restrictions. Then we will denote the generating function for $\{D(1), D(2), \ldots\}$ by $G(D(n),x)$. Given a polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots$, we will denote $[x^n] f(x)$ to be the coefficient of x^n in $f(x)$, i.e., $[x^n] f(x) = a_n$. With these notations we have $R(n) = [x^n] G(R(n), x)$ and $P(n) = [x^n] G(P(n), x)$. By manipulation and interpretation of series and product identities, generating functions can be helpful in proving partition identities. Generating functions are most useful for their use in determining the coefficients of x^n or for getting upper or lower bounds of the coefficients of x^n. One may also use generating functions to arrive at the asymptotic value (if possible) of the coefficient of x^n. A good example of this application is to show that, $P(n) < e^{3\sqrt{n}}$ for all n (Cohen, 1978).

We shall now indicate how generating functions are created for certain types of partitions.

Start with the identities

$$1 + x + x^2 + \ldots = (1 - x)^{-1},$$

$$1+x^2+x^4+\ldots=\left(1-x^2\right)^{-1},$$

$$1+x^3+x^6+\ldots=\left(1-x^3\right)^{-1},$$

and from these identities form the equation

$$(1-x)^{-1}\left(1-x^2\right)^{-1}\left(1-x^3\right)^{-1}=\left(1+x+x^2+\ldots\right)\left(1+x^2+x^4+\ldots\right)\left(1+x^3+x^6+\ldots\right)$$

$$=1+x+x^2+3x^3+4x^4+5x^5+7x^6+8x^7+\ldots\ .$$

The coefficient of x^7 is 8, and Table 2.3 explains all the possibilities that result in x^7 in the product. The corresponding partition summand is indicated in brackets.

Table 2.3: all possibilities that result in x^7

From the first factor		From the second factor		From the third factor	
x^7	(1+1+1+1+1+1+1)	1		1	
x^5	(1+1+1+1+1)	x^2	(2)	1	
x^4	(1+1+1+1)	1		x^3	(3)
x^3	(1+1+1)	x^4	(2+2)	1	
x^2	(1+1)	x^2	(2)	x^3	(3)
x	(1)	x^6	(2+2+2)	1	
x	(1)	1		x^6	(3+3)
1		x^4	(2)	x^3	(1)

If we associate with each of the rows in Table 2.3 a partition of 7 into ones, twos, and threes then the total of the ones in the partition is given in brackets from the first factor, the total of the twos is given in brackets from the second factor, and the total of the threes is given in brackets from the third factor. Table 2.4 demonstrates this association.

Table 2.4: Association between the factors and the partitions of 7

Data from Table 2.3			Associate the partition			The partition
x^7	1	1	seven 1's,	no 2's,	no 3's	1+1+1+1+1+1+1
x^5	x^2	1	five 1's,	one 2,	no 3's	1+1+1+1+1+2
x^4	1	x^3	four 1's,	no 2's,	one 3's	1+1+1+1+3
x^3	1	x^4	three 1's,	two 2's,	no 3's	1+1+1+2+2
x^2	x^2	x^3	two 1's,	one 2,	one 3	1+1+2+3
x	x^6	1	one 1,	three 2's,	no 3's	1+2+2+2
x	1	x^6	one 1,	no 2's,	two 3's	1+3+3
1	x^4	x^3	no 1's,	two 2's,	one 3	2+2+3

Conversely, every partition of 7 will constitute a choice for obtaining the coefficient of x^7 in the above product.

In general, the above observation is true for every integer *n*. Hence we may deduce that $(1-x)^{-1}(1-x^2)^{-1}(1-x^3)^{-1}$ is the generating function for the number of partitions of an integer *n* into parts no greater than 3. The generalized result may be stated as in theorem below.

Theorem 2 (Andrews, 1971): The generating function for the number of partitions of an integer into parts equal to or less than m is $\prod_{j=1}^{m}(1-x^j)^{-1}$.

Proof: Given an integer *n*, let $\pi_m(n)$ denote the number of partitions of *n* into parts equal to or less than *m* an let $|x|<1$. It suffices to show that $\prod_{j=1}^{m}(1-x^j)^{-1} = \sum_{n=0}^{\infty}\pi_m(n)x^n$ for all *n*. Now, since $\sum_{k=0}^{\infty}(x^j)^k = \frac{1}{1-x^j}$ for all j = 1,2,... and $|x|<1$, therefore,

$$\prod_{j=1}^{m}\left(1-x^j\right)^{-1} = \prod_{j=1}^{m}\left(\sum_{k=0}^{\infty} x^{kj}\right)$$

$$=\left(1+x+x^2+\ldots\right)\left(1+x^2+x^4\ldots\right)\ldots\left(1+x^m+x^{2m}+\ldots\right)$$

$$=\pi_m(0) \quad + \quad \pi_m(1)x+ \quad \pi_m(2)x^2+\ldots$$

$$= \sum_{n=0}^{\infty}\pi_m(n)x^n \ . \qquad \blacksquare$$

Now, let A be a set of positive integers. With slight modification of the above we can show that the generating function for the number of partitions of an integer into parts from A is $\prod_{a_i \in A}\left(1-x^{a_i}\right)^{-1}$.

If we take the above set A to be (i) the set of all positive integers then it follows that the generating function for the unrestricted number of partitions of an integer is

$$G(P(n),x) = \prod_{k=1}^{\infty}\left(1-x^k\right)^{-1},$$

(ii) the set of all odd positive integers, and let $O(n)$ denote the number of partitions of n into odd parts, then the generating function for $O(n)$ is

$$G(O(n),x) = \prod_{k=1}^{\infty}\left(1-x^{2k-1}\right)^{-1}.$$

Similar to the above development, the coefficient of x^n in the expansion of $\left(1+x^{a_1}\right)\left(1+x^{a_2}\right)\left(1+x^{a_3}\right)\ldots$ is exactly the number of ways n can be represented as the sum of the a_i's where each a_i appears at most once. Thus we may deduce that if $D(n)$ denotes the number of partitions of n into distinct parts then the generating function for $D(n)$ is

$$G(D(n),x) = \prod_{k=1}^{\infty}\left(1+x^k\right).$$

With the above results, some identities in partition theory are easily proved.

Thereom 3: the number of partitions of any integer n into odd parts equals the number of partitions of n into distinct parts (i.e., $O(n) = D(n)$).

proof: Consider

$$1+x=\frac{1-x^2}{1-x},$$

$$1+x^2=\frac{1-x^4}{1-x^2},$$

$$1+x^3=\frac{1-x^6}{1-x^3},$$

$$\cdots\,.$$

By multiplying all the terms in the left hand side together we get the generating function for $D(n)$ while multiplying all the terms in the right hand side together we get the generating function for $O(n)$ (after cancellations), thus giving us the desired result. ■

The above example is only one of the many identities in partition theory which are proved by using generating functions. We will see another application in theorem 4.

In Andrews (1971) we find the following problem which the author stated as an excercise. "Prove that the number of partitions of *n* into distinct parts congruent to 1,2, or 4 (modulo 7) is equal to the number of partitions of *n* into parts congruent to 1,9, or 11 (modulo 14)". Theorem 4 is our new result which generalizes the above problem.

Theorem 4: Let q be a prime number. Let $n_1, n_2, \ldots, n_r$ be distinct positive integers less than q such that the set $B = \{k\in N: k\equiv 2n_i \pmod{2q},\ 1 \le i \le r\}$ is contained in the set $A = \{k\in N: k\equiv n_i \pmod{q},\ 1 \le i \le r\}$. Then the number of partitions of an integer into distinct parts congruent to $n_1, n_2, \ldots, n_r$ (modulo q) is equal to the number of partitions from the set $A \backslash B$.

Proof: Let $G(A,x)$ be the generating function for the number of partitions of an integer into distinct parts congruent to $n_1, n_2, \ldots, n_r$ (modulo q) and $G(A \backslash B, x)$ be the number of partitions into parts from the set $A \backslash B$. Then $G(A,x) = \prod_{k \in A} \left(1 + x^k\right)$ and $G(A \backslash B, x) = \prod_{k \in A \setminus B} \left(1 - x^k\right)^{-1}$.

Now,

$$G(A,x) = \prod_{\substack{k \in N \\ k \equiv n_1, \ldots, n_r (mod \quad q)}} \frac{\left(1 + x^k\right)\left(1 - x^k\right)}{\left(1 - x^k\right)}$$

$$= \left\{ \prod_{\substack{k \in N \\ k \equiv n_1, \ldots, n_r (\mathrm{mod} \quad q)}} \left(1 - x^{2k}\right) \right\} \left\{ \prod_{\substack{k \in N \\ k \equiv n_1, \ldots, n_r (\mathrm{mod} \quad q)}} \left(1 - x^k\right)^{-1} \right\}$$

$$= \left\{ \prod_{k \in B} \left(1 - x^k\right) \right\} \left\{ \prod_{k \in A} \left(1 - x^k\right)^{-1} \right\}$$

$$= \left\{ \prod_{k \in B} \left(1 - x^k\right) \right\} \left\{ \prod_{k \in B} \left(1 - x^k\right)^{-1} \bullet \prod_{k \in A \setminus B} \left(1 - x^k\right)^{-1} \right\}$$

$$= \prod_{k \in A \setminus B} \left(1 - x^k\right)^{-1}$$

$$= G(A \backslash B, x). \quad \blacksquare$$

CHAPTER III

THE CAPTURE-RECAPTURE METHOD

Suppose we want to estimate the total population of a certain species of animal. Assume that each member of this species is equally like to be caught. The Capture-Recapture method works as follows (Devore, 1987).

Let P be the unknown total population of the species.

(i) Catch and tag M_1 members of the species, and put them back into the population. Let T_1 be the number of tagged members of the population. Then $T_1 = M_1$. Thus, the ratio of tagged members to the total population is $\frac{T_1}{P}$, where T_1 is known and P is unknown.

(ii) Capture another M_2 members of the population, and suppose that there are S_1 tagged members among the M_2. Then, since each member is equally likely to be caught, we have $\frac{S_1}{M_2} \cong \frac{T_1}{P}$, where $\cong$ means "approximately equal". We may stop here with some terminating condition or continue with (iii), which repeats step (ii). Our estimate of P is $\frac{T_1 M_2}{S_1}$.

(iii) Tag all members in M_2 that were not previously tagged and release to the population. Let the new total of the tagged members of the population be T_2. Repeat (ii) using M_3, S_2, T_2 and $\frac{S_2}{M_3} \cong \frac{T_2}{P}$. Our new estimate of P is $\frac{T_2 M_3}{S_2}$.

Our objective is to estimate $P(n)$. To achieve our objective, we will first generate a set A of M_1 different random partitions of n, where each partition is equally likely to be generated. Next, we generate another set, B, of M_2 different partitions of n. Use the Capture-Recapture method on sets A and B. We either stop or continue by setting A=A∪B. To our knowledge, this method of "capturing" partitions by simulation and storing them in a list is new. Capture-Recapture methods using simulation could have broad application in combinatorics, when it is difficult to obtain a closed formula for a count of a particular type and when it is difficult to find an organized counting procedure for the item of interest. One difficulty would be the simulation to generate population members with equally likely probabilities. We now give a brief description of how our sets are generated, followed by some results after running the C program capture-recapture.c in appendix B.

III.(i) Algorithm to Generate Equally Likely Partitions

Given an integer n, represent n as n separated dots, i.e.,

$$\bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \dots\ \bullet \qquad (\, n \text{ dots}\,).$$

Then there are n-1 spaces between these dots. Given n, the program Capture-recapture.c in Appendix B will randomly generate a string of n-1 zeros and ones. For simplicity, think of a 1 at the j-th position as a vertical line drawn at the j-th space and a 0 as meaning no line is drawn at the j-th space. Assume that k lines are drawn randomly. Let n_1 be the number of dots before the first line, n_{k+1} be the number of dots after the last line. For i = 2 to k let n_i be the number of dots between line i-1 and line i. Thus we see that $n = n_1 + \dots + n_{k+1}$ is a partition of n. For example, take n=10 and assume

k=3 lines were randomly drawn as . . | . . . | . . . | . . . Then this would correspond to the partition 10 = 2 + 3 + 3 + 2.

Property 1: This process does not generate equally likely partitions.

Proof: For example, with n = 8 the partition 1+1+...+1 has probability of being generated equal to $\frac{1}{2^7}$ since there is only one way to divide the dots which gives the partition 1+1+...+1, which is

. | . | . | . | . | . | . | ..

On the other hand the partition 2+3+3 has probability of being generated equal to $\frac{3}{2^7}$, since there are three ways to divide the dots which yield the partition 2+3+3. They are

. . | . . . | . . .,

. . . | . . | . . .,

and . . . | . . . | . . . ■

Thus, we use an elimination process so that all partitions of n will be equally likely generated.

Elimination process: Suppose we randomly divide the dots above and get a partition of n. We group equal parts together, so we may write the above representation of n as $n = (r_1)n_1 + (r_2)n_2 + \ldots + (r_j)n_j$, where n_i appears r_i times in the partition. In our example 10 = 2+3+3+2, we would regroup this as 10 = (2)2 + (2)3. Next, we calculate $\frac{r_1!r_2! \ \ldots \ r_j!}{r!}$, where $r = \sum_{i=1}^{j} r_i$, and pick a uniform random

number u , with $u \in [0,1]$. If either $j = 1$ or u is less than or equal to $\frac{r_1!r_2!\ \dots\ r_j!}{r!}$,
we say that the partition is good and record it, otherwise we discard it and repeat the process.

Property 2: Each partition is equally likely to be generated using the elimination process.

Proof: Before the elimination process, the partition 1+1+...+1 has probability $\frac{1}{2^{n-1}}$ of being generated.

A partition of type $(r_1)n_1 + (r_2)n_2 + \dots + (r_j)n_j$ has probability $\frac{1}{2^{n-1}}\left(\frac{r!}{r_1!r_2!\ \dots\ r_j!}\right)$ of being generated since there are $\frac{r!}{r_1!r_2!\ \dots\ r_j!}$ different arrangements of r objects with r_1 of type 1, ..., r_j of type j. So, if we only keep a partition of type $(r_1)n_1 + (r_2)n_2 + \dots + (r_j)n_j$ with probability $\frac{r_1!r_2!\ \dots\ r_j!}{r!}$, the probability of generating a partition of type $(r_1)n_1 + (r_2)n_2 + \dots + (r_j)n_j$ which is not eliminated is $\frac{1}{2^{n-1}}\left(\frac{r!}{r_1!r_2!\ \dots\ r_j!}\right)\left(\frac{r_1!r_2!\ \dots\ r_j!}{r!}\right) = \frac{1}{2^{n-1}}$. Thus, each partition has the same probability of being generated and not eliminated, namely $\frac{1}{2^{n-1}}$. Since each partition has the same probability of being generated and kept and since there are $P(n)$ partitions, then the probability of each partition being generated given that it is kept must be $\frac{1}{P(n)}$. Of course, this means the probability that a generated partition is eliminated must be $\frac{2^{n-1} - P(n)}{2^{n-1}}$. ■

Four example, suppose n=10 and the partition (2)2+2(3) was generated, then the probability that this partition is accepted is $\frac{2!2!}{4!} = \frac{1}{6}$.

III.(ii) <u>Algorithm to Apply the Capture-Recapture Method to Simulated Partitions</u>

(a) With the above method we randomly generate a set A of different partitions of n.

(b) We generate another set, B, of different partitions of n.

(c) Apply the Capture-Recapture method and either stop or set A = A$\cup$B and repeat step (b).

The following pages show the data observed after running the C program Capture_recapture.c in Appendix B.

For n = 10 we present the actual data produced. Initially we generate 10 distinct partitions for set A. We generate a sample (set B) of 7 different partitions of 10 in each sample. Then the number of matches ($|A\cap B|$) are recorded, and the set A is updated. The estimated number of partitions, which we call E(10), is calculated after the second simulation by using the equation $E(10) = (7\times|A|)/|A\cap B|$.

For n = 6, 7, 8 and 9 we will construct the tables for the observed data.

For $n = 10$:

Simulation 1:

Initially set A is

{1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+2+2+3, 1+1+1+1+2+4,

1+1+2+2+2+2, 1+2+3+4, 2+2+3+3, 1+1+3+5, 4+6, 2+8}

|A| = 9.

Simulation 2:

B = {1+1+1+3+4, 1+1+2+2+4, 1+1+2+3+3, 1+2+2+2+3, 1+2+2+5, 2+2+3+3,

1+4+5}

The new partitions are :1+1+1+3+4, 1+2+2+5, 1+1+2+2+4, 1+1+2+3+3,

1+2+2+2+3, 1+4+5.

The number of matches are: 1

The updated set A is:

{1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+2+2+3, 1+1+1+1+2+4,

1+1+2+2+2+2, 1+2+3+4, 2+2+3+3, 1+1+3+5, 4+6, 2+8, 1+1+1+3+4,

1+2+2+5, 1+1+2+2+4, 1+1+2+3+3, 1+2+2+2+3, 1+4+5}

| A | = 16

Simulation 3:

B = {1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+1+1+2+3, 1+1+1+1+2+4,

1+1+1+1+1+5, 1+1+1+2+2+3, 1+1+1+3+4}.

The new partitions are:1+1+1+1+1+5, 1+1+1+1+1+2+3

The number of matches are: 5

The updated set A is:

{1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+2+2+3, 1+1+1+1+2+4,

1+1+2+2+2+2, 1+2+3+4, 2+2+3+3, 1+1+3+5, 4+6, 2+8, 1+1+1+3+4,

1+2+2+5, 1+1+2+2+4, 1+1+2+3+3, 1+2+2+2+3, 1+4+5, 1+1+1+1+1+5,

1+1+1+1+1+2+3}.

| A | = 18.

Simulation 4:

B = {1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+1+1+2+3, 1+1+1+1+3+3,

1+1+1+2+2+3, 1+1+2+2+4, 1+1+1+3+4}

The new partitions are : 1+1+1+1+3+3, 1+1+1+1+1+2+3

The number of matches are: 5

E(10) = 7× (18) / 5 ≈ 25.

The updated set A is:

{1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+2+2+3, 1+1+1+1+2+4,

1+1+2+2+2+2, 1+2+3+4, 2+2+3+3, 1+1+3+5, 4+6, 2+8, 1+1+1+3+4, 1+2+2+5,

1+1+2+2+4, 1+1+2+3+3, 1+2+2+2+3, 1+4+5, 1+1+1+1+1+5, 1+1+1+1+1+2+3,

1+1+1+1+3+3, 1+1+1+1+1+2+3}.

| A | = 20.

Simulation 5:

B = {1+1+1+1+1+1+1+3, 1+1+1+1+1+1+2+2, 1+1+1+1+1+2+3,

1+1+1+1+1+1+4, 1+1+1+1+3+3, 1+1+1+2+5, 1+1+2+3+3}

The new partitions are: 1+1+1+1+1+1+1+3, 1+1+1+1+1+1+4, 1+1+1+2+5

The number of matches are: 4

E(10) = 7×20 / 4 = 35.

The updated set A is:

{1+1+1+1+1+1+2+2, 1+1+1+1+2+2+2, 1+1+1+2+2+3, 1+1+1+1+2+4, 1+1+2+2+2+2, 1+2+3+4, 2+2+3+3, 1+1+3+5, 4+6, 2+8, 1+1+1+3+4, 1+2+2+5, 1+1+2+2+4, 1+1+2+3+3, 1+2+2+2+3, 1+4+5, 1+1+1+1+1+5, 1+1+1+1+1+2+3, 1+1+1+1+3+3, 1+1+1+1+1+2+3, 1+1+1+1+1+1+1+3, 1+1+1+1+1+1+4, 1+1+1+2+5}.

| A | = 23.

We stop at the fourth simulation with E(10) = 35. The actual value *P*(10) is 42. We may get large errors in this method if the sample size is not sufficiently large. We present summaries of our results for *n* = 6, 7, 8 and 9.

<u>*n* = 6. *P*(6) = 11.</u> Initially, 5 different partitions of 6 are generated.

Simulation #	Number of Matches	E(6)	Number of Distinct Partitions Generated
1	2	25/2	8
2	4	40/4	9
3	4	45/4	10
4	4	50/4	11
5	5	11	11

<u>*n* = 7. *P*(7) = 15.</u> Initially, 5 different partitions of 7 are generated.

Simulation #	Number of Matches	E(6)	Number of Distinct Partitions Generated
1	1	45/4	9
2	3	45/3	11
3	4	55/4	12
4	3	60/3	14
5	5	14	14

<u>$n = 8$. $P(8) = 22$.</u> Initially, 5 different partitions of 8 are generated.

Simulation #	Number of Matches	E(8)	Number of Distinct Partitions Generated
1	3	25/3	7
2	3	35/3	9
3	3	15	11
4	1	55/1	15
5	3	25	17

<u>$n = 9$. $P(9) = 31$.</u> Initially, 5 different partitions of 9 are generated.

Simulation #	Number of Matches	E(9)	Number of distinct Partitions Generated
1	1	25	9
2	2	45/2	12
3	2	60/2	15
4	4	74/4	16
5	4	80/4	17
6	3	85/3	19

We observe from the above tables that for n=6,7,8 and 9, the estimated values of $P(n)$ are good estimations of the actual values. We also note from the above tables that the total number of partitions generated in all simulations exceeds the true value for $P(n)$. For small n, there is a good chance that all or almost all of the partitions of n are generated during the simulations as happened in the first 3 tables for n = 6, 7 and 8. But as n gets larger, $P(n)$ increases exponentially (Andrew, 1971) so that it will be costly (time-wise) to generate almost all the partitions for n.

The partitions generated in the C program Capture_recapture.c are unrestricted, but the program could be easily modified to generate partitions of n with restrictions. For example, if in our program we restricted the random 0-1 strings so that at most m-1 1's can occur for some integer *m*, then this would restrict our generated partitions to partitions of *n* into at most *m* parts. It is not difficult to modify our program to put other restrictions on the partitions that we want to generate. With this flexibility we will could try to estimate the number of partitions of integers with restrictions (such as partitions with at most *m* parts, partitions into odd/even parts, partitions into parts greater than some positive integer *k*, partitions into parts less than or equal to *k*,...).

We have restricted our example to small values of *n* because of computer time needed, because it makes it easy to compare the estimate with the true value, and because small values of *n* illustrate the procedure as well as large values.

It should be noted that our method of generating equally likely partitions using a discard procedure can be very inefficient for large values of *n*.

CHAPTER IV

THE ESTIMATION FUNCTION

The use of estimation functions gives another method to predict the unknown population of a certain species. The method of using estimation functions also relies on sampling with replacement, but unlike the Capture-Recapture method which requires equally likely probabilities for each member of the species is to be captured, the estimation function doesn't need this condition. We use the data from the C program Prediction.c (which is also found in appendix B) to obtain our results.

One estimator for the number of unseen members of a population was developed by Efron & Thisted (1976). Another estimator, with additional desirable properties was developed by Boneh, Boneh, and Caron (1997). We use this latter estimator for our study. Briefly, their estimation function can be summarized as follows.

Suppose that in sampling with replacement of some species, we notice that there were k repetitions of a particular captured member for $k = 1,2,\ldots.$ Let N_k represent the number of distinct members of the species that were captured exactly k times and $M = \max\{k\colon N_k > 0\}$. Then the suggested function for estimating the number of unseen members in the population is

$$(1) \qquad \Psi(t) = \sum_{k=1}^{M} N_k e^{-k} - \sum_{k=1}^{M} N_k e^{-k(1+t)} \quad .$$

By letting $t \to \infty$ in $\Psi(t)$, we obtain the estimated number of the population of the species which were not detected in the sample. Hence the estimated total population is $\sum_{k=1}^{M} N_k + \Psi(\infty)$.

The data we use for our study is simulated by the C program Prediction.c listed in Appendix B. Given an integer n and a sample size S, this program generates S partitions of n, keeps track of the number of distinct partitions and how many partitions of each type were generated. The program generates each partition by first generating a string of zeros and ones of length n-1 then converting it to a partition as in the program Capture_recapture.c without the elimination process. Thus the partitions generated are not equally likely.

Here are some of our results using the above prediction function.

For n = 10, our sample consists of 200 random partitions of 10 of which 35 distinct partitions were detected, and M = 13. Thus, $\sum_{k=1}^{M} N_k = 35$. The following table summarizes this result.

<u>n = 10 and sample-size = 200.</u>

k	N_k	$k \times N_k$
1	13	13
2	4	8
3	2	6
4	1	4
5	3	15
6	1	6
7	2	14
8	2	16
9	0	0
10	2	20
11	1	11
12	0	0
15	1	15
19	1	19
26	1	26
27	1	27
Total	35	200

Letting $t \to \infty$, in (1), we may approximate $\Psi(t)$ by the first three terms in $\Psi(t)$ since starting with the fourth term, every term is relatively small. The first three terms of $\Psi(t)$ are $13e^{-1}+4e^{-2}+2e^{-3}$. Hence the number of undetected partitions in the above sample is 4.782 + 0.541 + 0.100 = 5.432. Thus the estimated value for $P(10)$ is 35 + 5.432 = 40.432. Compared with the actual value (42), this is quite a "good" estimate for $P(10)$.

The next table compares the estimated values against the true values of $P(n)$ for n=11, 12,..., 17. The data simulated for these numbers are found in Appendix A1. Appendix A2 contains the actual values of $P(n)$ for n ranging from 0 to 99.

Estimated Values for $P(n)$, n = 11,12,...,17

n	Sample Size	E(n)	$P(n)$
11	500	53.6200	56
12	1000	74.6900	77
13	1500	95.0000	101
14	2500	130.6922	135
15	4000	173.1445	176
16	5000	223.8152	231
17	8000	283.3109	297

CHAPTER V

TRADITIONAL METHODS

In this chapter we use generating functions to obtain new results in partition theory.

V.(i) A Recursive Formula For *P(n)*

In this section we give a new formula for *P*(*n*).

Property 1: $P(n) = \sum_{k=1}^{n} P(k, n-k)$, where *P*(*m*,*n*) denotes the number of partitions of *n* into parts less than or equal to *m*, and we define *P*(*n*,0) = 1 for all *n*.

Proof: Recall from chapter II that

(i) if $f(x) = \sum_{k=0}^{m} a_k x^k$ is a polynomial in *x* then $[x^n]f(x)$ denotes the coefficient a_n in $f(x)$, and

(ii) the generating function for {*P*(1), *P*(2),...,} is $G(P(n), x) = \prod_{k=1}^{\infty} (1 - x^k)^{-1}$.

Thus, for *n*≥2,

$$P(n) = [x^n]G(P(n), x) = [x^n] \prod_{k=1}^{\infty} (1 - x^k)^{-1}$$

$$= [x^n](1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots)\cdots(1 + x^{n-1} + x^{2(n-1)} + \cdots)(1 + x^n + \cdots)\cdots$$

$$= [x^n](1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots)\cdots(1 + x^{n-1} + x^{2(n-1)} + \cdots)(1 + x^n)$$

$$= [x^n](1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots)\cdots(1 + x^{n-2} + \cdots)(1 + x^{n-1})(1 + x^n)$$

$$= \left[x^n\right]\left(1+x+x^2+\cdots\right)\left(1+x^2+x^4+\cdots\right)\cdots\left(1+x^{n-2}+\cdots\right)\left(1+x^{n-1}+x^n\right)$$

$$= \left[x^n\right]\{\ \left(1+x+x^2+\cdots\right)\left(1+x^2+x^4+\cdots\right)\cdots\left(1+x^{n-2}+x^{2(n-2)}+\cdots\right)\ \}+1+$$

$$= \left[x^n\right]\{\ \prod_{k=1}^{n-2}\left(1-x^k\right)^{-1}\ \}+2$$

$$= P(n-2,n)+2, \qquad \ldots\ldots\ldots\ldots \qquad (1)$$

Now, $P(m,n)$ = The number of partitions of n with every part $\leq m$

= the number of partitions of n with every part $\leq m - 1$

+ the number of partitions of n with every part $\leq m$ and m must occur

= $P(m\text{-}1,n) + P(m,n\text{-}m)$ .... (2)

Thus, by (2), we have, for $n \geq 2$,

$P(n\text{-}2,n) = P(n\text{-}3,n) + P(n\text{-}2,2)$

$= P(n\text{-}4,n) + P(n\text{-}3,2) + P(n\text{-}2,n)$

.

.

.

$= P(1,n) + P(2,n\text{-}2) + P(2,n) + \ldots + P(n\text{-}2,2)$

$= P(1,n\text{-}1) + P(2,n\text{-}2) + P(2,n) + \ldots + P(n\text{-}2,2)$

Substitute this into (1) to get

$P(n) = P(1,n\text{-}1) + P(2,n\text{-}2) + \ldots + P(n\text{-}2,2) + 1 + 1$

or

$P(n) = \sum_{k=1}^{n} P(k,n-k)$, since $P(n\text{-}1,1) = P(0,n\text{-}1) = 1$ for $n \geq 1$. ■

V.(ii) The Number of Partitions of n Into Odd Parts, None of Which Greater Than $2m$

In this section we prove that the number of partitions of n into odd parts less than or equal to $2m$ is equal to the number of partitions of n into distinct parts from the set $\{1,2,\ldots,2m,2m+2,2m+4,\ldots,4m,4m+4,4m+8,\ldots,8m,8m+8,\ldots\}$.

Lemma 1: Given $m \in N$,

$$\text{let } P_i = \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 0 \bmod (2^{i-1})}} (1+x^k) \right\} \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod (2^i)}} (1-x^k) \right\}, i \geq 1.$$

Then

$$P_i = \left\{ \prod_{\substack{2^i m < k \leq 2^{i+1} m \\ k \equiv 0 \bmod (2^i)}} (1+x^k) \right\} P_{i+1} \quad \forall\ i \geq 1.$$

Proof: Note that

(3) for $i>1$, $k \equiv 0 \bmod (2^{i-1})$ iff either $k \equiv 0 \bmod (2^i)$ or $k \equiv 2^{i-1} \bmod (2^i)$

(4) $$\left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod (2^i)}} (1-x^{2k}) \right\} = \left\{ \prod_{\substack{k > 2^{i+1} m \\ k \equiv 2^i \bmod (2^{i+1})}} (1-x^k) \right\}$$

(5) $\{k : k > 2^i m,\ k \equiv 0 \bmod (2^i)\}$

$$= \{ 2^i m < k \le 2^{i+1} m \ : \ k \equiv 0 \bmod\left(2^i\right) \} \cup \{ k > 2^{i+1} m \ : \ k \equiv 0 \bmod\left(2^i\right) \}.$$

Thus,

$$P_i = \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 0 \bmod\left(2^{i-1}\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod\left(2^i\right)}} \left(1 - x^k\right) \right\}$$

$$= \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 0 \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod\left(2^i\right)}} \left(1 - x^k\right) \right\},$$

by (3)

$$= \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 0 \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 2^{i-1} \bmod\left(2^i\right)}} \left(1 - x^{2k}\right) \right\}$$

$$= \left\{ \prod_{\substack{k > 2^i m \\ k \equiv 0 \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^{i+1} m \\ k \equiv 2^i \bmod\left(2^{i+1}\right)}} \left(1 - x^k\right) \right\} \text{, by (4)}$$

$$= \left\{ \prod_{\substack{2^i m < k \le 2^{i+1} m \\ k \equiv 0 \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^{i+1} m \\ k \equiv 0 \bmod\left(2^i\right)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 2^{i+1} m \\ k \equiv 2^i \bmod\left(2^{i+1}\right)}} \left(1 - x^k\right) \right\} \text{,by (5).}$$

Observe that the last two terms in the above expression give P_{i+1}.

Therefore, for all $i \in N$ we have

$$P_i = \left\{ \prod_{\substack{2^i m < k \le 2^{i+1} m \\ k \equiv 0 \bmod (2^i)}} (1+x^k) \right\} P_{i+1}, \text{ as required.} \qquad \blacksquare$$

Lemma 2: Let $A_i = \prod_{\substack{2^i m < k \le 2^{i+1} m \\ k \equiv 0 \bmod (2^i)}} (1+x^k)$. Then

$[x^n] P_i = [x^n](A_i A_{i+1} \ldots)$ for all i.

Proof: Let's look at the recurrence relation $P_i = \left\{ \prod_{\substack{2^i m < k \le 2^{i+1} m \\ k \equiv 0 \bmod (2^i)}} (1+x^k) \right\} P_{i+1}$.

Then we have

$$P_i = A_i P_{i+1} = A_i A_{i+1} P_{i+2} \ldots = (A_i A_{i+1} \ldots A_{i+j}) P_{i+j+1}, \; j \in N.$$

Thus, given $n \in N$, we can choose $j \in N$ such that the exponents of x in P_{i+j+1} are all greater than n. Hence lemma 2 is proved. $\blacksquare$

Let $m \in N$, and let $\Omega_m = \{ 1, 2, \ldots, 2m, 2m+2, 2m+4, \ldots, 4m, 4m+4, 4m+8, \ldots \}$

***Proposition* 3**: The number of partitions of n into odd parts, none of which greater than $2m$ equals the number of partitions of n into distinct parts from Ω_m.

Proof: Let $_{2m}(n)$ be the number of partitions of n into odd parts, none of which greater than $2m$. Then its generating function is

$$G(\Phi_{2m}(n), x) = \prod_{k=1}^{m} (1 - x^{2k-1})^{-1}.$$

Let $G(\Omega_m, x)$ denote the generating function for the number of partitions of n into distinct parts from Ω_m. Without loss of generality, it suffices to show

$$\left[x^n\right] G(\Phi_{2m}(n), x) = \left[x^n\right] G(\Omega_m, x) \;\forall\; n.$$

Consider

$$\prod_{k=1}^{m}\left(1-x^{2k-1}\right)^{-1} = \left\{\prod_{k=1}^{\infty}\left(1-x^{2k-1}\right)^{-1}\right\}\left\{\prod_{k=m+1}^{\infty}\left(1-x^{2k-1}\right)\right\}$$

$$= \left\{\prod_{k=1}^{\infty}\left(1+x^{k}\right)\right\}\left\{\prod_{k=m+1}^{\infty}\left(1-x^{2k-1}\right)\right\},$$

by Theorem 3, chapter II,

$$= \left\{\prod_{k=1}^{2m}\left(1+x^{k}\right)\right\}\left\{\prod_{k=2m+1}^{\infty}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>2m\\ k\equiv 1\,mod(2)}}\left(1-x^{k}\right)\right\} \text{..... (6)}$$

Now, $$\prod_{k=2m+1}^{\infty}\left(1+x^{k}\right) = \left\{\prod_{\substack{k>2m\\ k\equiv 0\,mod(2)}}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>2m\\ k\equiv 1\,mod(2)}}\left(1+x^{k}\right)\right\}.$$

Therefore,

$$\left\{\prod_{k=2m+1}^{\infty}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>2m\\ k\equiv 1\,mod(2)}}\left(1-x^{k}\right)\right\} = \left\{\prod_{\substack{k>2m\\ k\equiv 0\,mod(2)}}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>2m\\ k\equiv 1\,mod(2)}}\left(1-x^{2k}\right)\right\}$$

$$= \left\{\prod_{\substack{2m<k\leq 4m\\ k\equiv 0\,mod(2)}}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>4m\\ k\equiv 0\,mod(2)}}\left(1+x^{k}\right)\right\}\left\{\prod_{\substack{k>2m\\ k\equiv 1\,mod(2)}}\left(1-x^{2k}\right)\right\}$$

$$= \left\{ \prod_{\substack{2m < k \le 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 4m \\ k \equiv 2 \bmod(4)}} \left(1 - x^k\right) \right\} \qquad \text{.............} \quad (7)$$

Substituting (7) into (6) gives

$$\prod_{k=1}^{m} \left(1 - x^{2k-1}\right)^{-1}$$

$$= \prod_{k=1}^{2m} \left(1 + x^k\right) \times \left\{ \prod_{\substack{2m < k \le 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{k > 4m \\ k \equiv 2 \bmod(4)}} \left(1 - x^k\right) \right\}.$$

Now, given $n \in$ N, $\exists\, j \in N$, such that

$$\left[x^n\right] \prod_{k=1}^{m} \left(1 - x^{2k-1}\right)^{-1} = \left[x^n\right] \left\{ \prod_{k=1}^{2m} \left(1 + x^k\right) \right\} \left\{ \prod_{\substack{2m < k < 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} P_i, \text{ by Lemma 1}$$

$$= \left[x^n\right] \prod_{k=1}^{2m} \left(1 + x^k\right) \times \left\{ \prod_{\substack{2m < k \le 4m \\ k \equiv 0 \bmod(2)}} \left(1 + x^k\right) \right\} [A_2 A_3 \ldots], \text{ by Lemma 2}$$

$$= \left[x^n\right] \left\{ \left[(1+x)(1+x^2)\cdots(1+x^{2m})\right] \left[(1+x^{2m+2})(1+x^{2m+4})\cdots(1+x^{4m})\right] \left[(1+x^{4m+4})\cdots(1+x^{8m})\right] \cdots \right\}$$

$= \left[x^n\right] G\left(\Omega_m, x\right)$. This completes our proof. ■

CHAPTER VI

CONCLUSIONS

In chapter III we gave a new method for estimating the number of partitions of an integer. In our examples in chapter III we only worked with unrestricted partitions, but as stated earlier, we could easily modify our C programs to estimate restricted partitions.

In chapter IV, we used an estimation technique developed by Boneh, Boneh, and Caron (1997) to obtain estimates of $P(n)$.

In chapter V we proved some new results using generating functions. The first result was a recursive formula for $P(n)$ and the second result showed that *the number of partitions of an integer n into odd parts less than or equal to 2m is equal to the number of partitions of n into distinct parts from the set* $\{1,2,\ldots 2m,2m+2,2m+4,\ldots,4m,4m+4,4m+8,\ldots,8m,\ldots\}$.

Thus, we have proved some new results using traditional methods and developed a new method (as applied to partition theory) to estimate the number of partitions.

Outside this thesis, a webpage for partition theory has been prepared by the author with URL http://www2.uwindsor.ca/~bui2/index.html.

REFERENCES

Andrews, G. E. *Number Theory.* W. B. Saunders Company (1971).

Boneh, S. Boneh, A. & Caron, R. J. *Estimating The Prediction Function And The Number of Unseen Species In Sampling With Replacement.* To be appear in the JASA in 1997.

Cohen, D. I. A. *Basic Techniques in Combinatorial Theory.* John Wiley & Sons Inc. (1978).

Devore, J. L. *Probability and Statistics for Engineering and the Sciences.* Wadsworth, Inc (1987).

Griffin, H. *Elementary Theory of Numbers.* McGraw-Hill Book Company, Inc (1954), pp 190 - 193.

Hardy, G. H. & Wright, E. M. *An Introduction to The Theory of Numbers.* Oxford University Press (1960).

Niven, A & Zuckerman, H. S . *An Introduction to The Theory of Numbers.* John Wiley & Sons Inc. (1966).

Riordan, J. *An Introduction to Combinatorial Analysis.* John Wiley & Sons Inc (1958).

Sloane N. J. A. & Plouffe. S. *The Encyclopedia of Integer Sequences.* Academic Press Inc. (1995).

APPENDIX A

(A1) Data To Be Used With The Prediction Function.

n=11			n=12			n=13		
k	n_k	$k\,n_k$	k	n_k	$k\,n_k$	k	n_k	$k\,n_k$
1	8	8	1	10	10	1	13	13
2	5	10	2	6	12	2	9	18
3	6	18	3	4	12	3	2	6
4	3	12	4	6	24	4	6	24
5	5	25	5	0	0	5	5	25
k>5	23	427	k>5	44	942	k>5	54	1414
Total	50	500	Total	70	1000	Total	89	1500

n=14			n=15			n=16		
k	n_k	$k\,n_k$	n_k	n_k	kn_k	k	n_k	kn_k
1	14	14	1	19	19	1	30	30
2	14	28	2	29	58	2	32	64
3	13	39	3	9	27	3	9	27
4	8	32	4	8	32	4	11	44
5	4	20	5	3	15	5	7	35
k>5	70	2367	k>5	104	3849	k>5	119	4800
Total	123	2500	Total	264	4000	Total	208	5000

n=17		
k	n_k	$k\,n_k$
1	41	41
2	24	48
3	16	48
4	10	40
5	14	70
k>5	159	7753
Total	264	8000

(A2) Table for the number of partitions of n for $n = 0,1,\ldots,99$.

$n = 10m + k$.

k m	*0*	*1*	*2*	*3*	*4*
0	1	42	627	5604	37338
1	1	56	792	6842	44583
2	2	77	1002	8349	53174
3	3	101	1255	10143	63261
4	5	135	1575	12310	75175
5	7	176	1958	14883	89134
6	11	231	2436	17977	105558
7	15	297	3010	21637	124754
8	22	385	3718	26015	147273
9	30	490	4565	31185	173525

k *m*	*5*	*6*	*7*	*8*	*9*
0	204226	966467	4087968	15796476	56634173
1	239943	1121505	4697205	18004327	64112359
2	281589	1300156	5392783	20506255	72533807
3	329931	1505499	6185689	23338469	82010177
4	386155	1741630	7089500	26543660	92669720
5	451276	2012558	8118264	30167357	104651419
6	526823	2323520	9289091	34262962	118114304
7	614154	2679689	10618963	38887673	133230930
8	715220	3087735	12132164	44108109	150198136
9	831820	3554345	13848650	49995925	169229875

APPENDIX B

(B1) Program Capture_recapture.c

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
int partition_length( int [], int );
int *string_to_partition( int *, int );
int add_array( int [], int );
float factorial( long int);
int compare_array(int [], int []);
float array_prob( int []);
int main(void)
{
      int a[1000][1000], b[1000][1000], c[1000][1000], length_a[1000],
         length_b[1000], temp_a[1000], temp_b[1000], string_a[1000], string_b[1000],
         *ttt,
         update[1000], match_array[1000][1000], row_sum[1000], match[1000],
         num_catches, catch[1000], row_a, row_b, old_set_1;
         set_1, m, n1, n, track_a, track_b, k, k1, l, index, sum, j, i, i1, i2, i3, number, temp,
         count;

      float rand_test, rand_test_2, test_value_a, rand_num_a, rand_num_b,
         rand_num_b_2,
         rand_test_b_2, r1, r2, rand_num_a_2, test_value_a_2, test_value_b,
         test_value_b_2;
      time_t t;
         printf(" enter the number to be partitioned : ");
                  scanf("%d", &number);
         printf("enter the number of partitions in set_1: ");
                  scanf("%d", &set_1);
      /*    PRODUCE A[1][i]        */
         srand((unsigned) time(&t));
         test_value_a = 1.50;
         while(test_value_a > 1)
         {        track_a = 1;
                  for ( i1=0; i1 < number - 1; i1++ )
                  {        string_a[i1] = rand() %2;        }
                           string_a[number-1] = 1;
```

```
                length_a[1] = partition_length( string_a, number);
                for( j = 0; j < length_a[1]; j++)
                {       ttt = string_to_partition(string_a, number);
                        temp_a[j+1] = ttt[j];
                }     temp_a[0] = length_a[1];
        r1 = rand() % LONG_MAX;
        rand_num_a = r1 / LONG_MAX;
        if( array_prob(temp_a) > rand_num_a )
                test_value_a = 0;
        else
                test_value_a = 2;
        }                       /*     end first while loop          */
        for( i1 = 0; i1 <= length_a[1]; i1++ )
        {       a[track_a][i1] = temp_a[i1];   }
/*     END PROCEDURE array a[1][*]     */
/*              PRODUCE SET_A                    */
        for( i2 = 2; i2 <= set_1; i2++ )
        {       rand_test_2 = 2;
                while( rand_test_2 > 1 )
                {       for ( i1=0; i1 < number - 1; i1++ )
                        {       string_a[i1] = rand() %2;      }
                                string_a[number-1] = 1;
                        length_a[i2] = partition_length(string_a, number);
                for( m= 0; m < length_a[i2]; m++)
                {       ttt = string_to_partition(string_a, number);
                        c[i2][m+1] = ttt[m];
                }     c[i2][0] = length_a[i2];
/*         ELIMINATION PROCESS           */
        i3 = 1;
        n1 = 0;
        while( i3 < i2 )
        {       if( compare_array(c[i2], a[i3]) == 0 )
                {       i3 = i3 + 1;
                        n1 = 0;
                }       else
                {       i3 = i2;
                        n1 = 1;
                }
        }
        if( n1 == 1 )
        {       rand_test_2 = 2;       }          else
        {       r2 = rand() %LONG_MAX;
                rand_num_a_2 = r2 / LONG_MAX;
                rand_test_2 = rand_num_a_2 / array_prob(c[i2]);
        }
        }
        for( count = 0; count <= length_a[i2]; count++ )
        {       a[i2][count] = c[i2][count];    }
}
        printf("enter the number of catches >");
                scanf("%d", &num_catches);
        for( k = 1; k  < num_catches +1; k++)
        {
                printf("enter how many partitions in this catch ? >");
                        scanf("%d", &catch[k]);
                test_value_b = 1.50;
                while(test_value_b > 1)
                {       track_b = 1;
                        for ( i1=0; i1 < number - 1; i1++ )
                        {       string_b[i1] = rand() %2;       }
                                string_b[number-1] = 1;
                        length_b[1] = partition_length(string_b, number);
                for( m=0; m < length_b[1]; m++)
                {       ttt = string_to_partition(string_b, number);
                        temp_b[m+1] = ttt[m];
                }       temp_b[0] = length_b[1];
                r1 = rand() % LONG_MAX;
                rand_num_b = r1 / LONG_MAX;
                if( array_prob(temp_b) > rand_num_b )
                {       test_value_b = 0;       }       else
                {       test_value_b = 2;       }
        }    /*     end while loop          */
        for( i1 = 0; i1 <= length_b[1]; i1++ )
        {       b[1][i1] = temp_b[i1];          }
/*     END PROCEDURE array b[1][*]     */
/*                  PRODUCE SET_B                         */
        for( i2= 2; i2 <= catch[k]; i2++ )
        {       rand_test_b_2 = 2;
                while( rand_test_b_2 > 1  )
                {       for ( i1=0; i1 < number - 1; i1++ )
                        {       string_b[i1] = rand() %2;       }
                                string_b[number-1] = 1;
                        length_b[i2] = partition_length( string_b, number);
                for( m=0; m < length_b[i2]; m++)
                {       ttt = string_to_partition(string_b, number);
                        c[i2][m+1] = ttt[m];
                }       c[i2][0] = length_b[i2];
  /*            ELIMINATION PROCESS                 */
        i3 = 1;
        n1 = 0;
        while( i3 < i2 )
        {       if( compare_array(c[i2], b[i3]) == 0 )
                {       i3 = i3 + 1;
                        n1 = 0;
                }       else
                {       i3 = i2;
                        n1 = 1;
                }
        }
                if( n1 == 1 )
                {       rand_test_b_2 = 2;      }       else
                {       r2 = rand() %LONG_MAX;
                        rand_num_b_2 = r2 / LONG_MAX;
                        rand_test_b_2 = rand_num_b_2 / array_prob(c[i2]);
                }
        }
        for( count = 0; count <= length_b[i2]; count++ )
        {     b[i2][count] = c[i2][count];    }
        }
/*    UPDATE SET A           */
        index = 0;
        for( i = 1; i <= catch[k]; i++ )
        {       j = 1;   count = 0;
                while( j <= set_1 )
                {
                        if( compare_array(b[i], a[j]) == 0 )
                        {       j++;   count++;      }       else
                        {     j = set_1 + 1;  }
                }
                if( count == set_1)
                {       update[index] = i;
                        index++;
                }
        }
        if( index > 0 )
        {       for( m = 0; m < index; m++ )
                {       for( l = 0; l <= b[update[m]][0]; l++ )
                        {       a[set_1 + m + 1][l] = b[update[m]][l];
                        }
                }
        }       old_set_1 = set_1;
        set_1 = set_1 + index;
printf("\nthe number of elements in the updated set_A is :  ");
printf("%d", set_1);
printf("\nthe number of non_matches were : ");
printf("%d", index);
printf("\nour estimate is :");
printf("%f", (catch[k]*old_set_1)/index);
printf("\n");
}
return(0);
/*          END MAIN PROGRAM               */
}
/*     FUNCTION: COMPARE_ARRAY     */
/*     Returns 1 if the arrays are the same, 0 if they are different      */
        int compare_array( int array_x[], int array_y[])
{
        int min_length;
        int e;
                if(array_x[0] <= array_y[0])
                {       min_length = array_x[0];        }       else
                {       min_length = array_y[0];        }
        e = 0;
        while( e <= min_length)
        {       if( array_x[e] == array_y[e] )
                {       e ++;   }       else
                {       e = min_length + 10;            }
        }
        if( e == min_length + 1 )
        {       return(1);      }       else
        {       return(0);      }
}

/*      FUNCTION:  FACTORIAL            */
/*      Returns the factorial of an integer     */
        float factorial( long int c3)
{
        if ( c3 <= 1 )
        return(1);
                else
        return( c3 * factorial( c3 - 1));
}
/*      FUNCTION ARRAY_PROB             */
/*      Returns the probability that a partition is accepted    */
        float array_prob( int x1[])
{
        long int y2[100];
        long int arr_ele_fac;
        int c5;
        long int z3;
        long int z4;
        x1[x1[0]+1] = 0;
        z3 = 0;
        y2[z3] = 1;
        z4 = 2;
        while( z4 <= x1[0] + 1 )
        {
                if( x1[z4] == x1[z4 - 1] )
                {
                        y2[z3] = y2[z3] + 1;
                        z4 = z4 + 1;
                }       else
                {
                        z3 = z3 + 1;
                        z4 = z4 + 1;
                        y2[z3] = 1;
                }
        }
        arr_ele_fac = 1;
        c5 = 0;
        while( c5 <= z3 )
        {
                arr_ele_fac = arr_ele_fac * factorial(y2[c5]);
                c5 = c5 + 1;
        }
        if( (arr_ele_fac / factorial(x1[0])) == 1 )
        {       return(2);      }
        else
        {       return( arr_ele_fac / factorial(x1[0]));        }
}
/*      FUNCTION: ADD _ARRAY            */
/*      Returns the sum of the elements in an array     */
int add_array( int array1[], int c6 )
{       int c7;
        int array_sum;
        if( c6 == 1 )
        {       array_sum = array1[0];          }
                else
        {       array_sum = 0;
                c7 = 0;
                while( c7 < c6 )
                {
                        array_sum = array_sum + array1[c7];
                        c7 = c7 + 1;
                }
        }
return(array_sum);
}
/*      FUNCTION: LENGTH OF PARTITION              */
/*      Returns the number of parts in a partition      */
int partition_length(int string_ [], int string_length)
{       int length;
        int c_x;
        length = 0;
        for(c_x = 0; c_x < string_length; c_x++)
        {       if (string_[c_x] == 1 )
                length++;
        }
return(length);
}
/*      FUNCTION: STRING_TO_PARTITION       */
/*      Converts a 0's and 1's string into a partition */
        int *string_to_partition( int *x, int k)
{       int c_1;
        int c_2;
        int length_t;
        int sum;
        int tt[100];
        int c_3;
        int c_4;
        int ttemp;
        int yy[100];
        length_t = 0;
        sum = 1;
        for( c_1=0; c_1 < k; c_1++ )
        {
                if( x[c_1] == 0 )
                {       sum++;          }
                else
                {       tt[length_t] = sum;
                        length_t ++;
                        sum = 1;
                }
        }
        for(c_3 = 0; c_3 < length_t; c_3++)
        {       for( c_4 = c_3 + 1; c_4 < length_t; c_4++)
                {       while(tt[c_4] > tt[c_3])
                        {       ttemp = tt[c_4];
                                tt[c_4] = tt[c_3];
                                tt[c_3] = ttemp;
                        }
                }
                yy[c_3] = tt[c_3];
        }
        for(c_2 = 0; c_2 < length_t; c_2++)
        {       return(yy);         }
}     /*    END ALL FUNCTIONS    */
/*      PROGRAM ENDS                */
```

(B2) <u>Program Prediction.c</u>

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
int add_array(int [] , int);
int compare_array(int [], int [], int, int);
int main(void)
{       int a[1000][1000], length[1000], new[1000], string[1000],
                match[1000][1000], total[1000], j1, set_size, i, k, m, n, sum, j, i1,
                index, index_t, number, temp, count, number_sets, test[1000], row;
        time_t t;
                printf(" enter the number to be partitioned > ");
                        scanf("%d", &number);
                printf("enter the number of partitions  > ");
                        scanf("%d", &set_size);
                printf("\n");
        srand((unsigned) time(&t));
        for( i = 1; i < set_size + 1; i++ )
        {
                for( k = 0; k < number - 1; k++ )
                {       string[k] = rand() %2;      }
                        string[number-1] = 1;
                length[i] = 0;
                sum = 1;
                for( j1=0; j1 < number; j1++ )
                {
                        if( string[j1] == 0 )
                        {       sum = sum + 1;          }
                                else
                                {       new[length[i]] = sum;
                        length[i]++;
                                sum = 1;
                                }
                }
                for( m=0; m < length[i]; m++ )
        {
                for(n=m+1; n < length[i]; n++)
                {
                        while(new[n] < new[m])
                                {
                                temp = new[n];
                                new[n] = new[m];
                                new[m] = temp;
                                }
                    }
            a[i][m] = new[m];
        }
}
        for( row = 2; row < set_size  + 1; row++ )
{
match[1][row - 2] = compare_array(a[1], a[row], length[1], length[row]);
}                   total[1] = add_array(match[1], set_size - 1);
                    printf(" there were  ");
                    printf("%d", total[1] + 1);
                    printf(" partitions of type n_1...");
                            for(j1 = 0; j1 < length[1]; j1 ++)
                            {       printf("%d", a[1][j1]);
                                    printf("+");
                            }       printf("\n");
index = 2;
index_t = index - 1;
count = 2;
while( count < set_size  )
{
        for( i1 = 1; i1 < count; i1++)
        {
test[i1 - 1] = compare_array( a[count], a[i1],length[count],length[i1] );
        }
                    if( add_array(test, count - 1) != 0 )
                    {       count = count + 1;
                    }
                    else
                    {
for( m = count + 1; m < set_size + 1; m ++ )
{
match[index][m - count - 1] = compare_array( a[count], a[m],length[count],
                                length[m] );
}
        total[index] = add_array( match[index], set_size - count );
                            count = count + 1;
                    printf(" there were  ");
                    printf("%d", total[index] + 1);
                    printf(" partitions of type n_");
                    printf("%d", index);
                    printf("\n");
                            for(j1 = 0; j1 < length[count - 1]; j1 ++)
                    {         printf("%d", a[count - 1][j1]);
                              printf("+");
                    }         printf("\n");
          index++;
          }
}

for( row = 1; row < set_size  ; row++ )
{
match[set_size][row - 1] = compare_array(a[set_size], a[row],
length[set_size], length[row]);
          }
          total[index] = add_array(match[set_size], set_size -1);
                    if( total[index ] == 0 )
                    {
                    printf(" there were  ");
                    printf("%d", total[index ] + 1);
                    printf(" partitions of type n_");
                    printf("%d", index );
                    printf("...");
                              for(j1 = 0; j1 < length[index]; j1 ++)
                                        {         printf("%d", a[index][j1]);
                                        printf("+");
                    }         printf("\n");
                    }         printf("\n");
return(0);
/*           END MAIN PROGRAM              */
}
/*        FUNCTION : COMPARE_ARRAY */
int compare_array( int x[], int y[], int c1, int c2)
{
          int z;
          int hit;
          if ( c1 != c2 )
          {         return(0);          }
          else
          {         hit = 1;
                    for( z = 0; z < c1; z++ )
                    {
                              if( x[z] == y[z] )
                              hit = hit * hit;
                              else
                              hit = hit * 0;
                    }
```

```
        }
        return(hit);
}
/*      FUNCTION ADD _ARRAY             */
int add_array( int array1[], int c6 )
{       int c7;
        int array_sum;
        if( c6 == 1 )
        {       array_sum = array1[0];  }
        else
        {       array_sum = 0;
                c7 = 0;
                while( c7 < c6 )
                {
                        array_sum = array_sum + array1[c7];
                        c7 = c7 + 1;
                }
        }
return(array_sum);
}
```

IMAGE EVALUATION TEST TARGET (QA-3)