Kevin Dittmar
Counting Partitions

Introduction

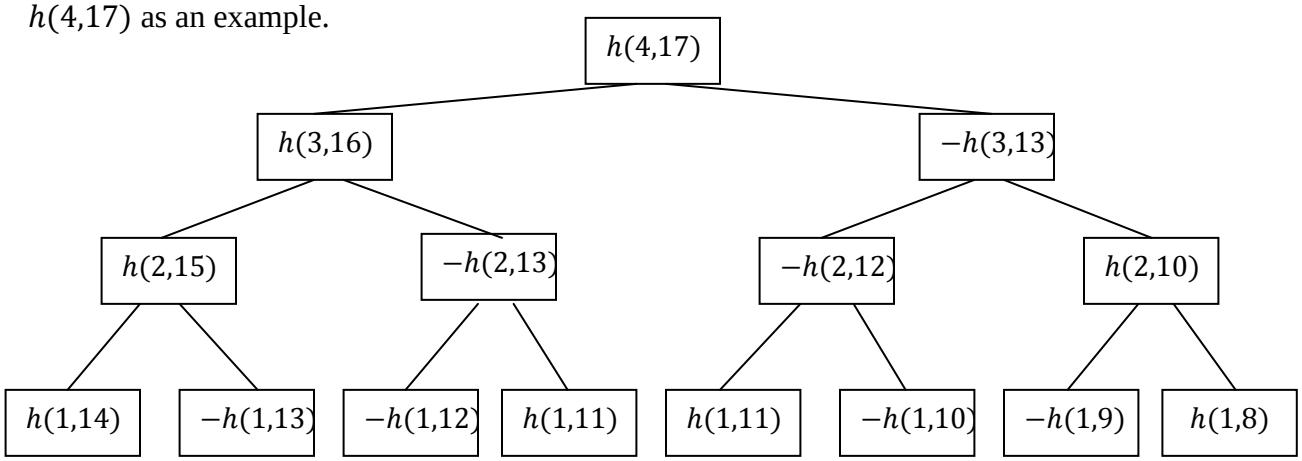
The purpose of this project is to count the number of restricted partitions of a given a number *n*. A partition in this context is defined as a set of numbers called parts that add up to a given number *n*. Various restrictions can be placed on a partition, such as requiring all parts to be distinct or requiring all parts to be odd. These and other criteria can be combined to form a more complex restriction. In this paper, we focus on the partitions of *n* with least part *m* and denote it as *h(m, n)*. Counting the number of partitions algorithmically is a particularly difficult endeavor. This is primarily because the partition counting problem, with and without restrictions, is an NP-Complete problem. The term NP-Complete refers to a problem that is both in the set NP and is NP-hard, where NP is an abbreviation for nondeterministic polynomial time. A problem is designated NP if a solution to the problem can be verified in polynomial time, and the problem is designated NP-hard if an algorithm to find a solution has a worst-case time complexity worse than polynomial time, which is usually exponential time. This problem is currently one of the most researched, unsolved problems in the field of computer science.

Mathematics:

As demonstrated by Chandrupatla, Hassen, and Osler in [1], this restricted partition satisfies the recurrence relation

$$h(m,n) = h(m-1,n-1) - h(n-1,n-m). \tag{1}$$

A tree of calls to the function h can be used to represent the decomposition of a call to the function into a pair of function calls that are easier to solve. The sum of the entries at the leaves or lowest level of the tree is equal to the original $h(m,n)$. Initially, the first row has only one element, which is $h(m,n)$. The second row is made up the two elements $h(m-1,n-1)$ and $-h(n-1,n-m)$. Every subsequent row has twice as many elements as the previous row until the elements in the row are of the form $h(1,x)$, which will be the m^{th} row. Hence, the m^{th} row will have 2^{m-1} elements because the first row has 1 element, and the number of elements doubles 2^{m-1} times. The elements of the m^{th} row will be referred to as leaves of the decomposition tree from this point on. The following tree represents a decomposition tree for $h(4,17)$ as an example.



The signs of the leaves of the function call tree follow a pattern, which can be described as follows. The values of the first and fourth in a group of four function calls always have the same sign, as do the values of the second and third. It is also true that the values of the first and fourth calls will have signs opposite of the values of second and third calls. As an example, the signs of the values of the first set of four function calls are positive, negative, negative, and positive respectively. Then, the signs switch, and the signs of the values of the second set of four function calls are negative, positive, positive, and negative respectively. The third pattern is the same as the second, and the fourth pattern is the same as the first. The pattern created by grouping the signs of the values in rows of four and in a block of sixteen is illustrated below with + symbols representing positive values for $h(i,j)$ and – symbols representing negative values.

(2)

$$+--+$$
$$-++-$$
$$-++-$$
$$+--+$$

The signs of the next sixteen function calls create the following pattern.

(3)

$$-++-$$
$$+--+$$
$$+--+$$
$$-++-$$

The third block of sixteen function calls will follow pattern (3), and the fourth block of sixteen function calls will follow pattern (2). These blocks can be grouped in four rows of four blocks to form the same recursive pattern formed by grouping the individual function calls in the same way. This recursive pattern is further demonstrated in Appendix B. For the rest of this paper, $f(k)$ will be used to represent the function defined as follows:

(4) $$f(k) = \begin{cases} 1, where\ the\ leaf\ at\ position\ k\ is\ positive \\ -1, where\ the\ leaf\ at\ position\ k\ is\ negative \end{cases}$$

Let each leaf in the bottom row of the decomposition tree of $h(m,n)$ be denoted as $h(1, x(k)))$, where $1 \leq x(k) \leq n$, x represents the number to partition for the $h(i,j)$ leaf at index k in the tree, the leftmost leaf is at $k = 0$, and

(5) $$h(m,n) = \sum_{k=0}^{2^{m-1}-1} h(1, x(k)).$$

Note that the leftmost $h(i,j)$ in the tree, which is $h(1,x(0))$ in the decomposition of $h(m,n)$, is

$$h(1,x(0)) = h\big(1, n-(m-1)\big).$$

From then on,

(6) $$x(k) = x(0) - d(k),$$

We define $d(k)$ as follows.

Write k as a linear combination

$$k = c_0 4^0 + c_1 4^1 + \cdots + c_p 4^p.$$

Let $[x]$ denote the largest integer that is at most x. For $k > 0$, let

$$k = \sum_{j=0}^{p} c_j 4^j \; c_j \in \{0,1,2,3\} \; where \; p = [\log_4 k].$$

We define $g(c_p, p)$ as

(8) $$g\big(c_p, p\big) = \begin{cases} 2p+1, c_p = 1 \\ (2p+1)+1, \; c_p = 2 \\ 2(2p+1)+1, \; c_p = 3 \end{cases}.$$

Sample values for $g(c_p, p)$ are provided in the following table.

p	4^p	$g(1,p)$	$g(2,p)$	$g(3,p)$
0	1	1	2	3
1	4	3	4	7
2	16	5	6	11
3	64	7	8	15
4	256	9	10	19
5	1024	11	12	23
6	4096	13	14	27
7	16384	15	16	31
⋮	⋮	⋮	⋮	⋮
p	4^p	$2p+1$	$(2p+1)+1$	$2(2p+1)+1$

Finally, we define $d(k)$, where k is the index of the leaf in the decomposition tree, recursively by

(9)
$$d(0) = 0$$
$$d(1) = 1$$
$$d(k) = g(c_p, p) + d(k - c_p 4^p).$$

The base case of the function returns 0 when its argument is 0 and 1 when its argument is 1. The recursive case of the function is made up of the sum of two operands. The first operand is the function g, which calculates the next portion of the decrement, and the second operand recursively calls the decrement function d with an updated argument. An example where $k = 156$ is given in Appendix C.

Using the definitions for the sign function $f(k)$ and the decrement function $d(k)$, the restricted partition $h(m, n)$ can be solved without using recurrence relation (1) directly given only the least part m and the number to partition n. This will allow us to calculate values for $h(m, n)$ using a computer program based on an iterative algorithm. This iterative algorithm is easier to optimize for efficiency than a recursive algorithm.

There is also a relationship between $h(m, n)$ and $p(n)$, which is the function that represents the total number of partitions of n with no restrictions. As demonstrated by Hassen and Osler in [2], the unrestricted partitions correspond with the pentagonal numbers, and they can be generated with the series

$$\sum_{k=1}^{\infty} (-1)^{k+1} \{p(n - f(k)) + p(n - f(-k))\},$$

Where $f(k)$ represents the k^{th} pentagonal number, and $f(-k)$ represents the k^{th} pentagonal number of negative index. The relationship between this generating sum and the generating sum for $h(m, n)$ is that $h(1, n)$ is equal to $p(n - 1)$. In other words, the generating sum for $h(1, n)$ is

$$\sum_{k=1}^{\infty} (-1)^{k+1} \{p((n-1) - f(k)) + p((n-1) - f(-k))\}.$$

The following table lists some examples demonstrating that $h(1, n + 1)$ is equal to $p(n)$, which is formally proven in Appendix D:

n	Unrestricted Partitions	$p(n)$	Partitions with Least Part 1	$h(1,n)$
1	1	1	1	1
2	1+1;2	2	1+1	1
3	1+1+1;1+2;3	3	1+2;1+1+1	2
4	1+1+1+1;1+1+2;1+3;2+2;4	5	1+1+1+1;1+1+2;1+3	3
5	1+1+1+1+1;1+1+1+2;1+2+2;1+1+3; 2+3;1+4;5	7	1+1+1+1+1; 1+1+1+2;1+1+3;1+4; 1+2+2	5
6	1+1+1+1+1+1;1+1+1+1+2;1+1+2+2; 2+2+2;1+1+1+3;1+2+3;3+3;1+1+4;2+4; 1+5;6	11	1+1+1+1+1+1; 1+1+1+1+2; 1+1+1+3;1+1+4;1+5; 1+2+3;1+1+2+2	7

Since all instances of $h(m,n)$ can be reduced to a sum of instances of $h(1,n)$ for some n through the recurrence relation mentioned in [1], this generating function can be used to generate any $h(m,n)$ in terms of a sum of the 2^{m-1} leaves its decomposition tree..

Computer Science:

When cast as a computer science problem, the partitions counting problem is an NP-Complete problem. This is because an algorithm implemented to solve the problem has, at best, an exponential time complexity. This can be easily ascertained through the use of a basic NP-Complete proof. The two criteria necessary for a problem to be NP-Complete is that the problem must exist within the NP set, and it must be possible to reduce a problem already classified as NP-Complete to this problem, which is done in the attached proof (Appendix A). Since the problem is NP-Complete, it is very difficult to find an efficient algorithm that produces a correct answer for large inputs. In an effort to increase the viability of solving this problem, several optimizations were made.

The C programming language, one of the most efficient programming languages available, is used. This choice was made because C is a programming language compiled to machine code and is much faster than interpreted languages and languages compiled to byte code.

The recurrence relation established by Chandrupatla, Hassen, and Osler in [1] is used to decompose the given $h(m,n)$ into simpler function calls before attempting to solve it. All of these function calls are in the form $h(m,n)\ where\ m = 1$, which eliminates a large portion of the recursive calls that would otherwise be necessary. This is important because the added addition operations execute much faster than the otherwise necessary recursive calls.

The previously mentioned patterns are used to generate the function calls associated with the decomposed $h(m,n)$ to further eliminate the necessity for recursion in favor of an iterative method. In particular, two files are generated when the program runs. The first file contains a

list of all of the values of n for which $h(1,n)$ will have to be computed. The second file contains a list of all of the signs for each $h(1,n)$. The program then pairs the sign off with the n to form the $h(1,n)$ values that, when added, will represent the number of partitions for $h(m,n)$. This reduces the space complexity of the program since recursive calls require more memory on the stack. This means that the iterative method will require fewer resources for the program to run.

To slightly improve efficiency, the program also eliminates $h(1,x)$ terms that cancel with other terms base on their index within a group of sixteen terms. In particular, as shown in Appendix E, the fourth term always cancels with the fifth term, the sixth term cancels with the ninth term, the eighth term cancels with the eleventh term, and the twelfth term cancels with the thirteenth term. This is true for every set of sixteen function calls iterating through the list of terms as long as all sixteen terms are necessary for calculating a partition count. Since eight of every sixteen function calls are eliminated, this optimization reduces the time required for the calculation by half.

A cache of pre-calculated values is also used to speed up the computation. In particular, $p(n) \mid 0 \leq n \leq 999{,}999$ is stored across fifty cache files in sets of $20{,}000$. This means that the program has access to $h(1,n)$ for all n up to one million. This data is stored across fifty cache files so that the program can dynamically load the portion or portions of the cache that it needs when it needs them, which reduces the amount of memory needed as well as the amount of time spent initializing the cache. This cache reduces the work of the program by preventing it from having to search for partitions. Instead, it only has to add the numbers whose sum represents the number of partitions requested by the given $h(m,n)$.

Conclusions:

As a result of this research project, $h(m,n)$ can be generated efficiently for all values of n given a small m. When m is 10, the calculation of $h(m,n)$ took approximately fifteen minutes regardless of the value of n. However, the calculation still has an exponential time complexity, which means that it is not efficient, because the calculation time is still exponentially related to the argument m. In other words, when m is increased by 1, the time to calculate $h(m,n)$ doubles. This is due to the fact that every increase of m by 1 doubles the number of terms that have to be added together, which doubles the amount of time needed to calculate $h(m,n)$. Although this method cannot be used to efficiently calculate $h(m,n)$ for all values of m and all values of n, $h(m,n)$ can be calculated efficiently for all values of n provided that the value of m is sufficiently small.

A more promising result of this research project is that the method described in this paper can be used for other restrictions. To make use of this method, the restricted partition has to be represented in terms of a recurrence relation in the form

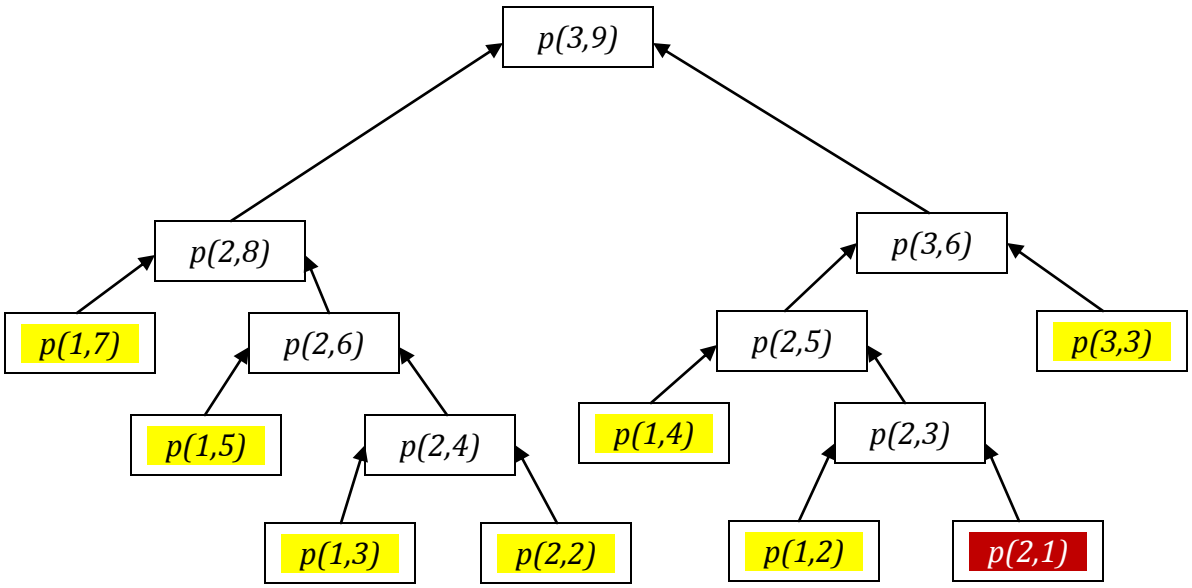
$$f(m,n) = f(m-a, n-b) + f(m-c, n-d),$$

where m is the restriction argument, n specifies the number to be partitioned, and a, b, c and d are integers. Additionally, either a or b and either c or d have to be positive to ensure that the recurrence relation will not be infinitely recursive. When the expansion of this recurrence relation is visualized as a binary tree, as it was for $h(m,n)$, the leaves of the binary tree will represent all of the simplest terms derived from expanding the recurrence relation. Moreover, the sum of all of these terms will equal the value of the original function, which is number of restricted partitions. These terms are also simple to evaluate. In the case of $h(m,n)$, $h(1,k) = p(k-1)$, $h(k,k) = 1$, and $h(k,x) = 0$, where k and x are positive integers and $k < x$.

An example of a restricted partition function for which this method can be used is $p(m,n)$. $p(m,n)$ is defined as the number of partitions such that m is the largest part, and n is the number to be partitioned. The recurrence relation for $p(m,n)$ is

$$p(m,n) = p(m-1,n-1) + p(m,n-m),$$

which is in the proper format and cannot be expanded infinitely. Shown below is the binary tree generated by expanding the recurrence relation for $p(3,9)$.



The terms at the leaves of the tree are marked in yellow and red, where all of the yellow leaves contribute 1 to the total count of partitions satisfying the conditions of $p(3,9)$. The red leaf is invalid and contributes nothing to the final count. Terms in the form $p(1,k)$, where k is a positive integer, represent 1 partition because the largest part and only valid part used to construct the partition is 1. Terms in the form $p(k,k)$, where k is a positive integer, represent 1 partition because the largest part in the partition is the number to be partitioned. Terms in the form $p(x,k)$, where x and k are positive integers and $k < x$ represent 0 valid partitions because it is impossible to have a partition whose largest part is greater than the number to be partitioned.

Since there are seven valid leaves, $p(3,9) = 7$, which is true because the valid partitions for $p(3,9)$ are the seven partitions listed below.

$$3 + 3 + 3$$

$$3 + 3 + 2 + 1$$

$$3 + 3 + 1 + 1 + 1$$

$$3 + 2 + 2 + 2$$

$$3 + 2 + 2 + 1 + 1$$

$$3 + 2 + 1 + 1 + 1 + 1$$

$$3 + 1 + 1 + 1 + 1 + 1 + 1$$

Other restricted partitions that meet the necessary criteria can be algorithmically solved in a similar manner.

Appendix A

NP-Complete Proof

1. The partition problem is a member of the set NP if it meets two criteria:
    - It is a decision problem.
    - A proposed solution to the problem is verifiable in polynomial time, or the time complexity is at worst $O(n^k) \mid k \in R$.

    First, the partition problem is a decision problem because the sum of a set of integers either is a partition or is not a partition. Since the answer is either yes or no, the problem is a decision problem (1).

    Second, a proposed solution is verifiable in polynomial time because the sum of a set of numbers can be ascertained in $O(n)$ time, which is polynomial time. This is the only requirement for an unrestricted partition. For a restricted partition, the restrictions must be checked, as well. The following is an example in the case of the main restricted partition examined during the course of this project, which is the restricted partition such that $h(m, n)$ yields the number of partitions of the integer n where the smallest part is the integer m (2).

    To verify that a proposed partition meets this restriction, it is only necessary to iterate through each number in the set. If the integer m is present in the set, and no number in the set is smaller than m, then the restriction is met, and the partition is valid provided that the sum of all numbers in the proposed solution is n. Verifying that the restriction is met has a time complexity of $O(n)$, which is polynomial time (3).

    By (1) and (2), the unrestricted partition problem is a member of the NP set. By (1), (2), and (3), the restricted partition problem where m must be the smallest part in the partition for the integer m is also a member of the NP set.

2. The partition problem is NP-hard if another NP-Complete problem can be reduced to the partition problem in polynomial time while preserving the solution set. For this proof, the Subset Sum problem will be reduced to the partition problem. The Subset Sum problem requires that a number n be the sum of a subset of a given set of numbers.

    Subset Sum -> Partition

    Generally speaking, a partition problem is a Subset Sum problem where repetition of the numbers in the set is allowed, and all numbers in the set must be positive. Hence, negative integers and zero must be removed.

    Let the set of numbers available for use in the subset sum be denoted as the set A such that

$$A = \{a_1, a_2, \dots, a_k\} \; where \; a_1 \leq a_2 \leq \cdots \leq a_k,$$

and let the subset target sum be defined as n. Therefore, a_1 is the minimum of the set and may or may not be negative. To make these potentially negative elements positive and non-zero, we add $(-a_1 + 1)$ to every element in the set A. After this transformation, $a_1 = 1$, and a_1 remains the minimum of set A. To preserve the potential equality of sums of subsets of A and the target sum n, it is necessary to redefine the target sum as $n + x(-a + 1)$, where x is the number of elements from set A used as summands in the subset sum. This ensures that the same value is added to both sides of the potential equality, which preserves the solution sets. This transformation has a time complexity of $O(n)$, which is linear time.

As of now, the problem has been transformed into a restricted partition problem where all parts must be distinct. However, changing the set of numbers available for the Subset Sum problem will change the type restriction on the partition. For example, allowing a number in the set to be chosen multiple times (or, equivalently, including each number k in the set $ceiling(\frac{n}{k})$ times) would yield an unrestricted partition problem. Since all variations of the Subset Sum problem have been proven to be NP-Complete, and the previously stated transformation works for all of them, the partition problems are also NP-Complete.

Partition -> Subset Sum:

This direction of the proof is trivial because a partition problem is a special type of Subset Sum problem. Restrictions on the partition only affect the set of numbers available, not the algorithm for finding a valid subset sum.

It is possible to reduce a Subset Sum problem to a partition problem in polynomial time while preserving the solutions and it is possible to reverse the operation. Hence, the partition problem is NP-Complete.

Appendix B

The following is a table of the signs of the functions produced through the decomposition of $h(10, n)$ into functions in the form $h(1, k)$ where $1 \leq k < n$. The value of k has no effect on the positivity or negativity of the functions representing the decomposition of $h(m, n)$. Both signs within a cell and the table cells themselves should be read from the upper-left corner to the upper-right corner, row by row, to establish the correct order. For example, the upper-leftmost cell is cell 1, to the right of that one is cell 2, and below cell 1 is cell 5. The same holds true for the signs in analogous positions.

+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−	−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+
−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+	+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−
−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+	+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−
+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−	−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+
−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+	+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−
+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−	−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+
+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−	−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+
−++− +−−+ +−−+ −++−	+−−+ −++− −++− +−−+	+−−+ −++− −++− +−−+	−++− +−−+ +−−+ −++−

The following table on the left represents the same data as the above table with the same rules, except that “X” represents the sign pattern established in (2), where the positive function calls make an “X” shape when plotted on a four by four grid. Similarly, the “O” represents the sign pattern established in (3), where the positive function calls make an “O” shape. The table on the

right represents the sign table for the decomposition of the function $h(6,n)$. Notice that the "X"s in the left table overlap with the "+"s in the right table, and the same is true for the "O"s in the left table and the "-"s in the right table. Hence, condensing the sign pattern of $h(m,n)$ by a factor of 16 yields the sign pattern of $h(m-4,\ n)$ where n is arbitrary.

X	O	O	X
O	X	X	O
O	X	X	O
X	O	O	X
O	X	X	O
X	O	O	X
X	O	O	X
O	X	X	O

+	−	−	+
−	+	+	−
−	+	+	−
+	−	−	+
−	+	+	−
+	−	−	+
+	−	−	+
−	+	+	−

Similarly, by further condensing the pattern of $h(10,n)$ and representing such that the "&" represents pattern (2), except where the "+"s are "X"s and the "-"s are "O"s, and the "*" represents pattern (3), $h(10,n)$ has the same pattern as $h(2,n)$, as shown below.

Considering this, it is also possible to replace any symbol with a four by four block of symbols representing an equivalent scheme to get the pattern for $h(m+4,n)$. For example, given the pattern for $h(m,n)$ made with "+"s and "-"s to denote positivity, it would be easily done to replace all of the "+"s with "X"s and all of the "-"s with "O"s. From there, it is easy to replace every "X" with the plus-minus pattern described in (2) and every "O" with the plus-minus pattern described in (3) to get the pattern for $h(m+4,n)$.

Appendix C

$d(k) = g(c_p, p) + d(k - c_p 4^p)$.

$p = [\log_4 k]$ *where p is the largest power in the linear combination of* k.

$$
\begin{aligned}
d(156) &= g(c_p, p) + d(156 - c_p 4^p) \\
&= g(2,3) + d(156 - 2 * 4^3) \\
&= g(2,3) + d(156 - 2 * 64) \\
&= g(2,3) + d(28) \\
&= (2(3) + 1) + 1 + d(28) \\
&= 8 + d(28) \\
&= 8 + g(c_p, p) + d(28 - c_p 4^p) \\
&= 8 + g(1,2) + d(28 - 1 * 4^2) \\
&= 8 + g(1,2) + d(12) \\
&= 8 + 2(2) + 1 + d(12) \\
&= 8 + 5 + d(12) \\
&= 13 + g(c_p, p) + d(12 - c_p 4^p) \\
&= 13 + g(3,1) + d(12 - 3 * 4^1) \\
&= 13 + 2(2(1) + 1) + 1 + d(0) \\
&= 13 + 7 + 0 \\
&= 20
\end{aligned}
$$

Hence, $x(156)$, the n argument passed to the leaf at position 156 is 20 less than the n argument passed to the leftmost leaf at position 0, which is $x(0)$.

Appendix D

$p(n)$ is the n^{th} pentagonal number

$h(1, n+1)$ is the number of partitions of $n+1$ with 1 as its smallest part.

$Proof$: $h(1, n+1) = p(n)\ where\ n > 0$

Let $p(n)$ be the number of unique members of a set such that

$$a_1 + a_2 + \cdots + a_k = n\ where\ a_1, \ldots, a_k\ are\ non-zero\ integers.$$

This represent any partition of 9 with no restrictions. Therefore,

$$1 + a_1 + a_2 + \cdots + a_k = n + 1\ where\ a_1, \ldots, a_k\ are\ non-zero\ integers.$$

represents the criterion for members of a set of partitions of $n+1$, each of which has 1 as its smallest part. Since non-positive integers including 0 are trivially forbidden from being parts of a partition, the set of numbers that form the sum $a_1 + a_2 + \cdots + a_k$ in each unrestricted partition of n and each partition of $n+1$ with least part 1 is exactly the same set. Since both sets trivially have the same size, $h(1, n+1) = p(n)$.

Appendix E

To speed up the computation, about half of the addition operations can be overlooked because half of the terms in every block of 16 cancel each other out. Below is an example from the start of the decomposition of $h(16{,}600)$ arranged in four row by four column blocks of sixteen terms. The bold terms will cancel each other out.

$$+h(1{,}585) - h(1{,}584) - h(1{,}583) + \boldsymbol{h(1, 582)}$$

$$\boldsymbol{-h(1, 582) + h(1, 581)} + h(1{,}580) \boldsymbol{- h(1, 579)}$$

$$\boldsymbol{-h(1, 581)} + h(1{,}580) \boldsymbol{+ h(1, 579) - h(1, 578)}$$

$$\boldsymbol{+h(1, 578)} - h(1{,}577) - h(1{,}576) + h(1{,}575)$$

This pattern persists in every block of sixteen terms, which reduces the number of operations required to calculate $h(m, n)$ by half.

Appendix F

```
/*
 * File:   least_part_m.c
 * Logic that counts the number of h(m,n) partitions.
 * Author: Kevin Dittmar
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gmp.h>
#include "least_part_m.h"

mpz_t cache[50][20000];
int is_loaded[50];
void h(mpz_t result, int m, int n)
{
    mpz_init(result);
    for (int i = 0; i < 50; i++)
    {
        //Initialize is_loaded array.
        is_loaded[i] = 0;
        //Initialize cache array
        for (int j = 0; j < 20000; j++)
        {
            mpz_init(cache[i][j]);
        }
    }
    //Cases where there is only one partition.
    if ((n == m) ||                //The smallest part is the only partition.
        (n >= (m * 2) && n < (m * 3)))  //1 partition for 2m <= n < 3m
    {
        mpz_set_ui(result, 1);
    }

    //3m == n yields exactly two unique partitions.
    else if ((m * 3) == n) //The smallest part is one third of the partition.
    {
        mpz_set_ui(result, 2);
    }

    //There are no partitions possible with these rules.
    else if ((n < m) ||  //The smallest part can't be bigger than the number.
             (n <= 0) || //No negative numbers; 0 has no partitions.
             (m < 0) ||  //Smallest part < 0 is invalid; 0 is unrestricted.
             ((2 * m) > n))  //Smallest part no more than half of the number.
    {
        mpz_set_ui(result, 0);
    }

    //Unrestricted partition.
    else if (m == 0)
    {
        cache_lookup(result, n + 1);
    }
```

```
    //Base case partition.
    else if (m == 1)
    {
        cache_lookup(result, n);
    }
    //Sum of base case partitions after decomposition.
    else
    {
        //Can be decomposed into parts through the recurrence relation, or
        //is already in the form h(1,n).
        FILE *function_file;
        FILE *sign_file;
        //Get the function tree that was generated earlier.
        function_file = fopen("function_tree.txt", "r");
        if (function_file == NULL)
        {
            printf("Error:  Cannot find function_tree.txt");
            exit(1);
        }
        //Get the sign tree that was generated earlier.
        sign_file = fopen("function_signs.txt", "r");
        if (sign_file == NULL)
        {
            printf("Error:  Cannot find function_signs.txt");
            exit(1);
        }

        mpz_t value;
        mpz_init(value);
        char sign;
        int read_n;
        //Pair off until we run out of numbers - always plenty of signs.
        while (fscanf(function_file, "%d;", &read_n) > 0 &&
               fscanf(sign_file, "%c", &sign) > 0)
        {
            if (read_n != 0)
            {
                cache_lookup(value, read_n);
                if (sign == '+')
                {
                    mpz_add(result, result, value);
                }
                else
                {
                    mpz_sub(result, result, value);
                }
            }
        }
    }
}

//Check if cache segment is initialized and return result.
void cache_lookup(mpz_t result, int n)
{
    //The cache stores the numbers from 1 to 1,000,000, but indices
    //start at 0.
    n -= 1;
```

```
    if (!is_loaded[n / 20000])
    {
        initialize_segment(n / 20000);
    }
    mpz_set(result, cache[n / 20000][n % 20000]);
}

//Load a needed segment of the cache into memory.
void initialize_segment(int index)
{
    FILE *file;
    char* name = "cachedir/h_cache_";
    char* extension = ".txt";
    char filename[32];
    int i = 0;
    mpz_t value;
    mpz_init(value);
    sprintf(filename, "%s%d%s", name, index, extension);
    file = fopen(filename, "r");
    while (gmp_fscanf(file, "%Zd\n", value) > 0)
    {
        mpz_set(cache[index][i], value);
        i++;
    }
    fclose(file);
}
```

```
/*
 * File:   unrestricted_partition.c
 * Generates a cache of unrestricted partitions to help find h(m,n).
 * Author: Kevin Dittmar
 */

#include <stdio.h>
#include <gmp.h>
#include "unrestricted_partition.h"
#include "least_part_m.h"

const int CHUNK_SIZE = 20000;
const int TOTAL_SIZE = 1000000;
mpz_t p_array[1000000];
/*Generate the p(n) cache array from 0-999999.  It just so happens that this
 *corresponds to h(1,n) from 1-1000000.
 */
void generate()
{
    mpz_init(p_array[0]);
    FILE *file;
    for (int i = 0; i < TOTAL_SIZE / CHUNK_SIZE; i++)
    {
        char* name = "cachedir/h_cache_";
        char* extension = ".txt";
        char filename[32];
        sprintf(filename, "%s%d%s", name, i, extension);
        file = fopen(filename, "w");
        if (file != NULL)
        {
            for (int n = (CHUNK_SIZE * i); n < (CHUNK_SIZE * (i + 1)); n++)
            {
                mpz_init(p_array[n]);
                if (n == 0)
                {
                    mpz_set_ui(p_array[0], 1);
                }
                else
                {
                    unrestricted_partition(p_array[n], n);
                }
                gmp_fprintf(file, "%Zd\n", p_array[n]);
            }
        }
        fclose(file);
    }
}

//Return the pentagonal number corresponding to k.
int pentagonal(int k)
{
    return k * ((3 * k) - 1) / 2;
}

//Return the number of unrestricted partitions of n.
void unrestricted_partition(mpz_t out, int n)
{
```

```
    int sign_counter = 0;
    int k = 0;
    int pent = 0;
    while (n - pent > 0)
    {
        k++;
        //Handle the case of positive k.
        pent = pentagonal(k);
        if (n - pent >= 0)
        {
            if (isPositive(sign_counter))
            {
                mpz_add(p_array[n], p_array[n], p_array[n - pent]);
            }
            else
            {
                mpz_sub(p_array[n], p_array[n], p_array[n - pent]);
            }
        }
        //If necessary, handle the case of negative k.
        if (n - pent > 0)
        {
            pent = pentagonal(-1 * k);
            if (n - pent >= 0)
            {
                if (isPositive(sign_counter))
                {
                    mpz_add(p_array[n], p_array[n], p_array[n - pent]);
                }
                else
                {
                    mpz_sub(p_array[n], p_array[n], p_array[n - pent]);
                }
            }
        }
        sign_counter++;
    }
    mpz_set(out, p_array[n]);
}

/* Based on the recurrence relation found by Drs. Hassen and Osler, the
 * following rules apply to the sign of the term.
 * If (counter / 2) is even, the sign is positive.
 * If (counter / 2) is odd, the sign is negative.
 * Hence, if (counter / 2) % 2 is 1 (true), the sign is negative.
 * else, if (counter / 2) % 2 is 0 (false), which is the only other case, the
 * sign is positive.
 * However, since the counter is incremented for every two pentagonal numbers
 * generated, the division by 2 is not necessary.
 */
int isPositive(int counter)
{
    if (counter % 2)
    {
        return 0;
    }
    else
```

```
    {
        return 1;
    }
}
```

```
/*
 * File:   h_sign_generator.c
 * Generates the signs needed to calculate h(m,n) in the
 * order that they are needed.
 * Author: Kevin Dittmar
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "h_sign_generator.h"

void h_sign_generator(int m)
{
    FILE *write;
    FILE *read;
    //correlates to X replace in report
    char plus_replace[17] = "+--+-++--++-+--+";
    //correlates to O replace in report
    char minus_replace[17] = "-++-+--++--+-++-";

    double bound = ceil((m - 2) / 4);
    write = fopen("function_signs.txt", "w");
    fprintf(write, "+");
    fclose(write);
    for (int i = 0; i <= bound; i++)
    {
        char next_sign;
        read = fopen("function_signs.txt", "r");
        write = fopen("sign_temp.txt", "w");
        while (fscanf(read, "%c", &next_sign) > 0)
        {
            if (next_sign == '+')
            {
                fprintf(write, "%s", plus_replace);
            }
            else if (next_sign == '-')
            {
                fprintf(write, "%s", minus_replace);
            }
        }
        fclose(read);
        fclose(write);
        system("mv sign_temp.txt function_signs.txt");
    }
}
```

```
/*
 * File:   main.c
 * Driver that finds the number of partitions for h(m,n), where
 * m is the first command line argument to the program and n is
 * the second command line argument to the program.
 * Author: Kevin Dittmar
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gmp.h>

#include "distinct.h"
#include "least_part_m.h"
#include "h_function_generator.h"
#include "h_sign_generator.h"
#include "unrestricted_partition.h"
mpz_t cache[1000000];
int main(int argc, char** argv)
{
    if (argc != 3)
    {
        printf("Usage: partitions_generating_working_copy.exe m n");
        exit(EXIT_FAILURE);
    }

    mpz_t result;
    mpz_init(result);

    //This will generate the cache of pentagonal numbers
    generate();

    //Generates order of signs needed for h(m,n) calculation
    h_sign_generator(atoi(argv[1]));

    //Generates h(1,n) function calls needed for h(m,n) calculation.
    h_function_generator(atoi(argv[1]), atoi(argv[2]));

    //Calculates h(m,n) result.
    h(result, atoi(argv[1]) , atoi(argv[2]));

    //Prints h(m,n) result.
    gmp_printf("%Zd\n", result);
    return (EXIT_SUCCESS);
}
```

References

[1] Chandrupatla , T. R, Hassen,A., Osler, T, *A Table of the Partition Function*, The Mathematical Spectrum, 34 (2001/2002), pp. 55 - 57.

[2] Hassen, A., Osler, T., *Playing with Partitons on the Computer*, Mathematics and Computer Education, 35(2001), pp. 5 – 17.