**1. Introduction**

A partition function is the number of ways of expressing a given positive integer as a sum of positive integers. For example, 5 can be expressed as sum of positive integers in 7 different ways:



Such a partition is called unrestricted partition. The number of unrestricted partition of a positive integer is denoted by . Thus, by listing the number of ways we can write the first five positive integers as a sum of positive integers, we can see that



But to find  in this manner is nearly impossible. There are some methods to generate the values of  for large values of  For a readable exposition of such a method, we recommend [2]. We also recommend [1] for further reading.

The other types of partition that one can study include the partition of a positive integer in which each summand is distinct. For example 5 has 3 distinct partitions:



If we denote by  the number of partitions of  into distinct parts, then, by actual enumerations, we can see that



Again such enumerations are almost futile when the integer is very large.

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 or without restrictions, is an NP-Complete problem.  The term NP-Complete refers to a problem that is both NP and 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. In other words, algorithms than can compute a partition is usually exponential time.

Our goal in this paper is to give a heuristic approach to show that a restricted partition is NP-hard by considering  as our main restricted partition.

**2. The Restricted Partition** 

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

.

This recurrence will allow us to find  once we know  for smaller values of  We shall find  by building a tree of values in which the top (or the first row) is . Then we use equation (1), to find the second row, which is made up the two elements and . The next row below this will now have four entries, two for each of the summands appearing in (1) Every subsequent row has twice as many elements as the previous row until the elements in the row are of the form , which will be the row. Hence, the row will have elements because the first row has element, and the number of elements doubles times. The elements of the row will be referred to as leaves of the decomposition tree from this point on. The following tree represents a decomposition tree for as an example.

We shall refer to such a diagram as a function call tree and the elements and each row as leaves. 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. In other words, 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 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. Similarly, we can represent patterns (2) and (3) as single symbols, such as X and O respectively. These symbols were chosen because the arrangement of positive signs in pattern (2) makes an “X pattern”, and the positive signs in pattern (3) make an “O pattern”. We start by arranging the symbols like so.

|  |  |  |  |
| --- | --- | --- | --- |
| +--+  -++-  -++-  +--+ | -++-  +--+  +--+  -++- | -++-  +--+  +--+  -++- | +--+  -++-  -++-  +--+ |
| -++-  +--+  +--+  -++- | +--+  -++-  -++-  +--+ | +--+  -++-  -++-  +--+ | -++-  +--+  +--+  -++- |
| -++-  +--+  +--+  -++- | +--+  -++-  -++-  +--+ | +--+  -++-  -++-  +--+ | -++-  +--+  +--+  -++- |
| +--+  -++-  -++-  +--+ | -++-  +--+  +--+  -++- | -++-  +--+  +--+  -++- | +--+  -++-  -++-  +--+ |
| -++-  +--+  +--+  -++- | +--+  -++-  -++-  +--+ | +--+  -++-  -++-  +--+ | -++-  +--+  +--+  -++- |

Then, by replacing each “X pattern” with an X and each “O pattern,” we create the following pattern.

XOOX

OXXO

OXXO

XOOX

The Xs in this pattern are analogous to the minus signs in patterns (2) and (3), and the Os are analogous to the plus signs. This kind of pattern continues indefinitely. For the rest of this paper, will be used to represent the function defined as follows:

Let each leaf in the bottom row of the decomposition tree of be denoted as , where , represents the number to partition for the leaf at index in the tree, the leftmost leaf is at , and

Note that the leftmost in the tree, which is in the decomposition of , is

.

From then on,

where the are defend by the recurrence formula

(7) ,

Here the  are the digits of  in the base 4, that is, they satisfy the equation

where , , and denote the largest integer that is at most . Lastly, are defined by

Sample values for are provided in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 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 |
|  |  |  |  |  |
|  |  |  |  |  |

The base case of the function will return when its argument is and when its argument is . The recursive case of is made up of the sum of and a recursive call to with its new argument. The function calculates the next portion of the decrement. For example, for , we have

Hence, . This means that the 156th leaf in the function call tree is 20 less than the 0th leaf, which is the leftmost leaf in the function call tree. By (6), .

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

It is clear that . As demonstrated by Hassen and Osler in [3], the unrestricted partitions correspond with the pentagonal numbers, and they can be generated with the series

Where represents the pentagonal number, and represents the pentagonal number of negative index. Thus, the generating sum for is

The following table lists some examples demonstrating that is equal to .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Unrestricted Partitions |  | Partitions with Least Part 1 |  |
| 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 can be reduced to a sum of instances of for some through the recurrence relation of equation (1), this generating function can be used to generate any in terms of a sum of the leaves its decomposition tree.

**3. Computing**

As we mentioned in the introduction section, 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 [2] is used to decompose the given into simpler function calls before attempting to solve it. All of these function calls are in the form , 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 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 for which will have to be computed. The second file contains a list of all of the signs for each . The program then pairs the sign off with the to form the values that, when added, will represent the number of partitions for . 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 terms that cancel with other terms base on their index within a group of sixteen terms. For example, 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 arranged in four row by four column blocks of sixteen terms. The bold terms will cancel each other out.

This pattern persists in every block of sixteen terms, which reduces the number of operations required to calculate by half. 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, is stored across fifty cache files in sets of . This means that the program has access to for all 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 .

**4. Concluding Remarks**

As a result of this research project, can be generated efficiently for all values of given a small . When is 10, the calculation of took approximately fifteen minutes regardless of the value of . 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 . In other words, when is increased by 1, the time to calculate doubles. This is due to the fact that every increase of by 1 doubles the number of terms that have to be added together, which doubles the amount of time needed to calculate . Although this method cannot be used to efficiently calculate for all values of and all values of , can be calculated efficiently for all values of provided that the value of 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

where is the restriction argument, specifies the number to be partitioned, and and are integers. Additionally, either or and either or 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 , 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 , , , and , where and are positive integers and .

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

,

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)*

*p(2,8)*

*p(3,6)*

*p(1,7)*

*p(2,6)*

*p(1,5)*

*p(2,4)*

*p(1,3)*

*p(2,2)*

*p(3,3)*

*p(2,5)*

*p(1,4)*

*p(2,3)*

*p(1,2)*

*p(2,1)*

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 . The red leaf is invalid and contributes nothing to the final count. Terms in the form , where 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 , where is a positive integer, represent 1 partition because the largest part in the partition is the number to be partitioned. Terms in the form , where and are positive integers and 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, , which is true because the valid partitions for are the seven partitions listed below.

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 .

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 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 yields the number of partitions of the integer where the smallest part is the integer (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 is present in the set, and no number in the set is smaller than , then the restriction is met, and the partition is valid provided that the sum of all numbers in the proposed solution is . Verifying that the restriction is met has a time complexity of , 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 must be the smallest part in the partition for the integer is also a member of the NP set.

1. 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 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 such that

and let the subset target sum be defined as . Therefore, 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 to every element in the set . After this transformation, , and remains the minimum of set . To preserve the potential equality of sums of subsets of and the target sum , it is necessary to redefine the target sum as , where is the number of elements from set 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 , 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 in the set 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 – Implementation of of solution in the C programming language

/\*

\* 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] Andrews, G.

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

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