Project 3 Report: Finding Integer Partition Number

Zhang Naifu
znf18@mails.tsinghua.edu.cn
funaizhang@github

Tsinghua University
October 31, 2018

Abstract

Project 3 aims to compute the number of possible integer partitions for a natural number input by the user. This is the number of distinct ways of representing n as a sum of natural numbers (with order irrelevant). The function to generate such integer partition numbers is called the partition function, denoted by $p(n)$.

calc_partition_number(n) in *Project_3.py* implements a version of Euler's recursive formula using his **pentagonal number theorem** and **dynamic programming** methods. Implementation details are discussed in Section 3. We find this is reasonably fast for integers less than 10000. Efficiency is discussed in Section 4.

This project makes no attempt at enumerating each partition.

The proofs of certain key mathematical theorems and statements go beyond the scope of the course, or the ability of the author, and they might thus be used without proof.

1 Illustrative Examples

Example 1

There are five partitions for natural number 4:
4
$3+1$
$2+2$
$2+1+1$
$1+1+1+1$

Example 2

By convention $p(0)=1$, $p(n)=0$ for negative n.

2 Brute Force Enumeration

An obvious way is to enumerate all the partitions of n. This could be made memory efficient, but is clearly extremely asymptotically time inefficient.

The partition number for large n is approximated by the Hardy-Ramanujan estimate:

$$\lim_{n\to\infty} p(n) = \frac{1}{4n\sqrt{3}} \exp(\pi\sqrt{\frac{2n}{3}})$$

In other words, the best we can do is not good enough,

$$p(n) = \Omega(\exp(\alpha\sqrt{n})) \quad \text{where } \alpha = \pi\sqrt{\frac{2}{3}}$$

The n^{-1} term disappears from the above time complexity equation because each partition does not take $\Theta(1)$ time to generate. Each component integer in a partition does - so each partition takes $\Theta(n)$ time.

3 Recurrence

Instead of counting, we use Euler's recurrence relation to calculate $p(n)$.

To begin with, we know the generating function for $p(n)$:

$$G(x) = \sum_{n=0}^{\infty} p(n)x^n = (1 + x + x^2 + ...)(1 + x^2 + x^4 + ...)(1 + x^3 + ...)... = \prod_{n=1}^{\infty} \frac{1}{1 - x^n}$$

Applying Euler's pentagonal number theorem without proof, the denominator could be expressed recursively,

$$\prod_{n=1}^{\infty}(1 - x^n) = \sum_{k=-\infty}^{\infty}(-1)^k x^{3(k-1)/2} = 1 + \sum_{k=1}^{\infty}(-1)^k(x^{3(k-1)/2} + x^{3(k+1)/2})$$

We can then derive the recurrence relation that corresponds to this generating function using the method taught in class. Omitting the full derivation steps, the resulting recurrence relation is:

$$p(n) = \sum_{k=1}^{n}(-1)^{k+1}(p(n - \frac{3k(k-1)}{2}) + p(n - \frac{3k(k+1)}{2}))$$

The last equation is implemented in *calc_partition_number(n)*. With $n = 1000$, we have the following output after a couple of seconds. For $n = 10000$, the programme would run for minutes.

Sample Output

```
$ ./Project_3.py

Please enter a natural number: 1000
p(1000) = 24061467864032622473692149727991
```

4 Complexity

calc_partition_number(n) contains two loops, one nested in the other. The inner loop over k implements the recursive formula, while the outer loop over n iterates over each natural number up to n for memoization. Therefore the time complexity with dynamic programming is,

$$p(n) = O(n^2)$$

This compares favorably with enumeration time complexity of $\Omega(\exp(\alpha\sqrt{n}))$.

Memoization takes up $O(n)$ space - a small price to pay for the improvement in speed.

Admittedly, there are more efficient solutions but these tend to trade off elegance and readability for efficiency. The author has on this occasion opted for more elegance code with marginally worse efficiency.

References

Wikipedia *https://en.wikipedia.org/wiki/Partition_(number_theory)*
Wikipedia *https://en.wikipedia.org/wiki/Pentagonal_number_theorem*
H. Wilf. *Lectures on Integer Partitions.* University of Pennsylvania. 2000. *https://www.math.upenn.edu/ wilf/PIMS/PIMSLectures.pdf*