Project Euler 12: Triangle Number with 500 Divisors

mathblog.dk/triangle-number-with-more-than-500-divisors/

View all posts by Kristian December 3, 2010

MATHBLOG

Problem 12 of Project Euler has a wording which is somewhat different than previous problems. However, as we shall see deriving efficient solutions for the problem, we can use theory which is very similar to some of the previous problems. The problem reads

> **The sequence of triangle numbers is generated by adding the natural numbers. So the 7^{th} triangle number would be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. The first ten terms would be:**
>
> **1, 3, 6, 10, 15, 21, 28, 36, 45, 55, …**
>
> **Let us list the factors of the first seven triangle numbers:**
>
> **1: 1**
> **3: 1,3**
> **6: 1,2,3,6**
> **10: 1,2,5,10**
> **15: 1,3,5,15**
> **21: 1,3,7,21**
> **28: 1,2,4,7,14,28**
> **We can see that 28 is the first triangle number to have over five divisors.**
>
> **What is the value of the first triangle number to have over five hundred divisors?**

When I first read the problem description, I rather quickly saw a direct approach to find the answer to the problem, where I use trial division to find the number of divisors to a number. This is the first approach I will describe in the this blog post. After that I will describe two modifications to solution strategy to speed up the algorithm.

Brute Force with Trial Division

The trial division is pretty straight forward. The main piece of the code looks like

```
int number = 0;
int i = 1;

while(NumberOfDivisors(number) < 500){
    number += i;
    i++;
}
```

where the “intelligence” lies in the function it calls called NumberOfDivisors. The function uses trial division of all numbers up to and including the square root of the original number.

The code for that looks as

```
private int NumberOfDivisors(int number) {
    int nod = 0;
    int sqrt = (int) Math.Sqrt(number);

    for(int i = 1; i<= sqrt; i++){
        if(number % i == 0){
            nod += 2;
        }
    }
    //Correction if the number is a perfect square
    if (sqrt * sqrt == number) {
        nod--;
    }

    return nod;
}
```

I don’t see much reason to go into the details of describing the code as it speaks for it self. The result is

```
The first triangle number with over 500 divisors is: 76576500
Solution took 214 ms
```

So while the code is easy to read, it isn’t terribly fast to execute.

Prime Factorisation

Now that we have established a method to find the answer. The question is how to speed it up. It is pretty obvious that the time consuming part of the code is the NumberOfDivisors functions. So lets see if we can improve that.

The property we need in order to find an efficient method to derive the number of divisors of a number is based on the prime factorisation of the number which we also dealt with in Problem 3.
Any number N is uniquely described by its prime factorisation

Where p_n is the a distinct prime number, a_n is the exponent of the prime and *K* is the set of all prime numbers less than or equal to the square root of *N*. Taking any unique combination of the prime factors and multiplying them, will yield a unique divisor of *N*. That means we can use combinatorics to determine the number of divisors based on the prime factorisation. The prime p_n can be chosen 0,1,…,a_n times. That means in all we can choose p_n in a_n+1 different ways.

The total number of divisors of *N* , *D(N)* can be expressed as

Now we just need a function to carry out the factorisation.

We used a prime factorisation in the solution to Problem 3, I have modified it to only use primes, instead of odd numbers, and I have modified it to actually count the divisors instead of returning a divisor.

The code looks like

```
private int PrimeFactorisationNoD(int number, int[] primelist) {
    int nod = 1;
    int exponent;
    int remain = number;

    for (int i = 0; i < primelist.Length; i++) { // In case there is a remainder
this is a prime factor as well // The exponent of that factor is 1 if
(primelist[i] * primelist[i] > number) {
            return nod * 2;
        }

        exponent = 1;
        while (remain % primelist[i] == 0) {
            exponent++;
            remain = remain / primelist[i];
        }
        nod *= exponent;

        //If there is no remainder, return the count
        if (remain == 1) {
            return nod;
        }
    }
    return nod;
}
```

It requires a list of primes, it starts from the lowest prime, and finds the exponent for that prime. Just as in Problem 3, if there the remainder reaches 1, there are no more divisors, and if there is a left over, that left over will be a prime divisor as well.

In the main loop we will need to generate a list of primes as well. This is done using the Sieve of Eratosthenes developed in Problem 10. With the line of code

```
int[] primelist = ESieve(75000);
```

I am uncertain of the upper limit of the prime divisors, so I tried to set it to 75000, to have a reasonably conservative bound.

Using prime factorisation yields the same result, but significantly faster than using trial division.

```
The first triangle number with over 500 divisors is: 76576500
Solution took 16 ms
```

Coprime Property

The last improvement of the algorithm, that we will deal with today, is a decomposition of the problem, into two smaller problems, which are significantly easier to solve.

In problem 6, we established that the sum of natural numbers can be written analytically, such that

where N_k is the k'th triangle number.

The property that we will use is that k and k+1 are coprimes (which we also dealt with in Problem 9). You can convince your self of that in the general case of k, using Euclid's alogrithm.
Since *k* and *k+1* are coprimes it means that their set of primes factors are distinct, and

$D(N_k) = D(k/2)D(k+1)$ if k is even and
$D(N_k) = D(k)D((k+1)/2)$ if k is odd.

In the first case, if *k* and *k+1* are coprimes, so is *k/2* and *k+1*, since 2 will be in the set of prime factors for *k*, and therefore not for *k+1*. Since otherwise they would not be coprimes. The problem of factorisation has now been split into prime factorisation for two smaller numbers, which is a significantly easier task.

Furthermore if we code it a bit smart, we can reuse the prime factorisation for k+1 in the subsequent iteration, and thus we only need to factorise one number in each iteration.

The prime generator and the prime factorisation we have already implemented can be reused, but we need to rework the main algorithm. I have chosen an implementations that looks like

```
int number = 1;
int i = 2;
int cnt = 0;
int Dn1 = 2;
int Dn = 2;
int[] primelist = ESieve(1000);

while (cnt < 500) {
    if (i % 2 == 0) {
        Dn = PrimeFactorisationNoD(i + 1, primelist);
        cnt = Dn * Dn1;
    } else {
        Dn1 = PrimeFactorisationNoD((i + 1) / 2, primelist);
        cnt = Dn*Dn1;
    }
    i++;
}
number = i * (i - 1) / 2;
```

Based on the explanation the code should be fairly self explanatory. One of the more obvious observations is that I have tightened the upper bound of the prime number generation signifanctly.

The execution of the algorithm yields

```
The first triangle number with over 500 divisors is: 76576500
Solution took 1 ms
```

For a number with 500 divisors, there wont be much difference, but I lost the patience of the second algorithm at 1200 divisors. Where this version still ran smoothly at more than 5000 divisors. So for scalability, the last version is a significant improvement.

Wrapping up

I covered three iterations of the algorithm, each improving the execution of the algorithm significantly by using different areas of number theory and combinatorics. As usual the source code is available for download. Did it make sense?