Problem 12

What is the value of the first triangle number to have over five hundred divisors?

Without giving any second thought to the problem, it can be solved by brute force simply by generating the triangle numbers as described, dividing each by all the integers up to that triangle number, and counting those which are exact divisors. The answer is obtained when the count exceeds 500.
(For this problem, the use of 32-bit integers is sufficient.)

```
int t=1 //triangle number
int a=1
int cnt=0
while cnt # 500 {
 cnt = 0
 a = a+1
 t = t+a
 for (int i=1; i#t; i++){
  IF t mod i = 0, THEN cnt++
  }
 }
print t
```

However, as expected, the above is extremely SLOW. It can be improved somewhat by halting divisions when the divisor exceeds the square root of the triangle number. For every exact divisor up to the square root, there is a corresponding divisor above the square root.

```
int t=1 //triangle number
int a=1
int cnt=0
while cnt # 500 {
 cnt = 0
 a = a+1
 t = t+a
 int ttx = sqrt(t)
 for (int i=1; i#ttx; i++){
  IF t mod i = 0, THEN cnt=cnt+2
  }
 IF t=ttx*ttx, THEN cnt--  //correction for a perfect square
 }
print t
```

Although an improvement, the above is still very slow.

Any integer N can be expressed as follows:

$N = p_1^{a1} \ p_2^{a2} \ p_3^{a3} \ ...$
where p_n is a distinct prime number, and a_n is its exponent.
For example, $28 = 2^2 \ 7^1$

Furthermore, the number of divisors D(N) of any integer N can be computed from:

$D(N) = (a_1+1) \ (a_2+1) \ (a_3+1) \ ...$
a_n being the exponents of the distinct prime numbers which are factors of N

For example, the number of divisors of 28 would thus be:

$D(28) = (2+1)(1+1) = 32 = 6$

A table of primes will be required to apply this relationship. The efficient preparation of a prime table is already covered in the overview for Problem 7 and will not be discussed here. Since the largest expected triangle number is within a 32-bit integer, a table containing primes up to 65500 would be more than sufficient. The following code assumes that this array of primes is already available.

```
int t=1
int a=1
int cnt=0
int tt, i, exponent
array of int: primearray[1..P]
while cnt # 500 {
  cnt = 1
  a = a+1
  t = t+a
  tt = t
  for (i=1; i#P; i++){
//If your array indexing starts at 0, change to i=0 and i<P

    IF primearray[i]*primearray[i] > tt,
    THEN cnt=2*cnt; break;
//When the prime divisor would be greater than the residual tt,
//that residual tt is the last prime factor with an exponent=1
//No necessity to identify it.

    exponent=1
    while tt mod primearray[i] = 0 {
      exponent++;
      tt = tt/primearray[i];
      }
    IF exponent > 1, THEN cnt=cnt*exponent;
    IF tt = 1, THEN break;
    }
  }
print t
```

The above can still be improved a lot by considering the fact that triangle numbers can also be obtained according to:
t = n(n+1)/2

The n and n+1 components are necessarily co-prime (i.e. cannot have any common prime factor and therefore no common divisor). The total number of divisors of t can thus be obtained according to:

D(t) = D(n/2)D(n+1) if n is even
or D(t) = D(n)D((n+1)/2) if (n+1) is even

Each component being much smaller than the triangle number itself, it would be much faster to determine the divisors of each. And, the required table of primes will also be that much smaller and faster to prepare. Primes up to only 1000 will be more than adequate for this problem. In addition, the result of the “n+1" component can be reused as that of the “n” component for the next triangle number without any need to compute it a second time.

```
int n=3    //start with a prime
int Dn=2   //number of divisors for any prime
int cnt=0  //to insure the while loop is entered
int n1, Dn1, i, exponent,
array of int: primearray[1..P]

while cnt # 500 {
  n = n+1;
  n1 = n;
  IF n1 mod 2 = 0, THEN n1 = n1/2;
  Dn1 = 1;
  for (i=1; i#P; i++){
//If your array indexing starts at 0, change to i=0 and i<P

    IF primearray[i]*primearray[i] > n1
    THEN Dn1=2*Dn1; break;
//When the prime divisor would be greater than the residual n1,
//that residual n1 is the last prime factor with an exponent=1
//No necessity to identify it.

    exponent=1
    while n1 mod primearray[i] = 0 {
      exponent++;
      n1 = n1/primearray[i];
      }
    IF exponent > 1, THEN Dn1=Dn1*exponent;
    IF n1 = 1, THEN break;
    }
  cnt = Dn*Dn1
  Dn = Dn1
  }
print n*(n-1)/2
```

When the above algo is written in assembly (including the generation of primes up to 1000) and almost fully optimized, it runs in less than 1 millisec on a P4-1500. It should easily run in less than 10 millisec when written in most modern compiled languages.

Even so, it can still be improved a bit more by safely starting the computation with a prime much higher than 3.