Problem 53
How many values of $C(n, r)$, for $1 \leq n \leq 100$, exceed one-million?

There are exactly ten ways of selecting three from five, 12345:

123, 124, 125, 134, 135, 145, 234, 235, 245, and 345

In combinatorics, we use the notation, ${}^5C_3 = 10$.

In general,

${}^nC_r = \frac{n!}{r!(n-r)!}$, where $r \leq n$, $n! = n \times (n-1) \times ... \times 3 \times 2 \times 1$, and $0! = 1$.

It is not until $n = 23$, that a value exceeds one-million: ${}^{23}C_{10} = 1144066$.

How many, not necessarily distinct, values of ${}^nC_r$, for $1 \leq n \leq 100$, are greater than one-million?

The naive approach would be to just compute the formula for every n and r where $23 \leq n \leq 100$ and $1 < r < n$. Unfortunately, the given formula requires handling extraordinarily large numbers. The numerator $n!$ overflows 32 bits at $n=13$ and, soon after that, it overflows 64 bits at $n=21$. In order to handle up to $n=100$ we need variables of more than 500 bits.

With a language —or library— capable of handling arbitrarily long numbers, an initial attempt could be:

Program A
```
function factorial (n)
    res = 1
    while (n > 1)
        res = res * n
        n = n - 1
    return res
count = 0
for n = 1 to 100 do:
    for r = 1 to n do:
        if factorial(n) / (factorial(r)*factorial(n-r)) > 1000000
            count = count + 1
print count
```

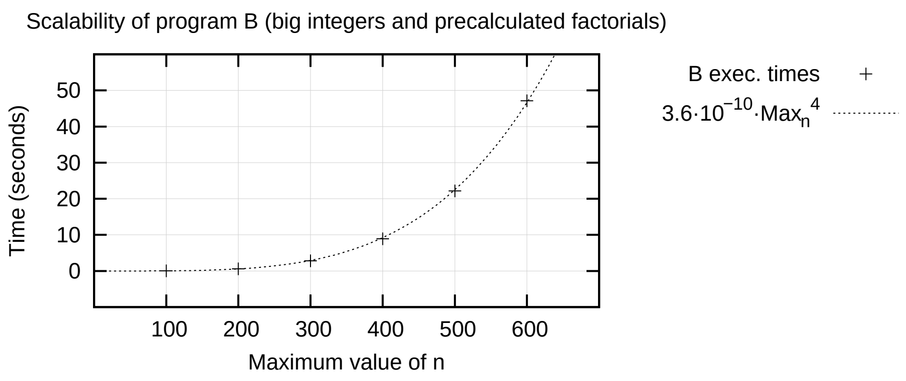
The previous program, implemented using the *"C++ Big Integer Library"* by Matt McCutchen, took 1.27 seconds to compute the result[1].

Obviously, the factorials can be precalculated and stored in a lookup table for later use. This simple optimization reduces the execution time to 51 ms:

Program B
```
f[0..100] = { uninitialized array }
f[0] = 1
for i = 1 to 100 do:
    f[i] = f[i-1] * i
count = 0
for n = 1 to 100 do:
    for r = 1 to n do:
        if f[n] / (f[r]*f[n-r]) > 1000000
            count = count + 1
print count
```

[1]All times were measured with implementations in C or C++, on an Intel® Core™2 T7200 @ 2.00GHz

If we change the limit —100 in the original problem— and try higher values, the previous program behaves as shown in the next graphic:



The execution time is acceptable with the original limit but, when this limit is increased, the execution time seems to follow a parabola of fourth degree. The code above contains only two nested loops, but the operations carried out inside these loops take more and more time as the limit grows, because they deal with astronomically growing numbers[2].

Let's analyse the given formula in order to simplify it as much as possible:

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} \tag{1}$$

The numerator $n!$ is the number of permutations of n elements[3]. If we take the first r elements of each permutation, we get all the possible selections of r elements, but every possible selection appears many times. Once with every possible ordering of both subsets: the r selected elements and the $n-r$ remaining elements. That is, exactly $r!\,(n-r)!$ times.

If we expand the formula, taking into account the fact that $r \leq n$, we can simplify the calculations cancelling $r!$ out of the fraction:

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} = \frac{n\cdot(n-1)\cdots(r+1)\cdot\cancel{r\cdot(r-1)\cdots 1}}{\cancel{r!}\,(n-r)!} = \frac{n\cdot(n-1)\cdots(r+1)}{(n-r)!} \tag{2}$$

This optimization is especially useful when r is nearly as large as n. In these cases the numerator will have just a few terms. On the other extreme, when r is very small, we can compute ${}^{n}C_{n-r}$ instead of ${}^{n}C_{r}$, since selecting r elements out of n —leaving the rest unselected— is equivalent to selecting the rest —leaving r unselected—:

$$ {}^{n}C_{n-r} = \frac{n!}{(n-r)!\,(n-(n-r))!} = \frac{n!}{(n-r)!\,(n-n+r)!} = \frac{n!}{(n-r)!\,r!} = {}^{n}C_{r} \tag{3}$$

In addition, due to this symmetry property, we can just compute ${}^{n}C_{r}$ with r ranging from $\lfloor \frac{n}{2} \rfloor$ to n, and count every value of ${}^{n}C_{r}$ as two values except when $r=\frac{n}{2}$.

[2]As an example, 59! is greater than 10^{80}, which is the estimated number of particles contained in the observable universe.

[3]A permutation is a linear arrangement or ordering of some elements. The permutations of n elements can be generated as follows: first we choose one of the n elements for the first place, then we choose one of the remaining $n-1$ elements for the second place, and so on. Thus, the number of permutations of n elements is $n\cdot(n-1)\cdot(n-2)\cdots 1 = n!$

Sadly, when $r \approx \frac{n}{2}$, the numbers involved are still too big compared to the numbers of combinations obtained after the division. If we could divide by some of the terms of the denominator while multiplying the terms of the numerator —and indeed we can, as we'll see in page 13— the numbers involved would be smaller.

Let's take a look at the numbers we are calculating:

${}^{n}C_{r}$	$r=0$	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$
$n=0$	1					
$n=1$	1	1				
$n=2$	1	2	1			
$n=3$	1	3	3	1		
$n=4$	1	4	6	4	1	
$n=5$	1	5	10	10	5	1

These numbers resemble the famous Pascal's Triangle, where every number is the sum of the two directly above it:

$$\begin{array}{c}
1 \\
1 \quad 1 \\
1 \quad 2 \quad 1 \\
1 \quad 3 \quad 3 \quad 1 \\
1 \quad 4 \quad 6 \quad 4 \quad 1 \\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1 \\
1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1 \\
1 \quad 7 \quad 21 \quad 35 \quad 35 \quad 21 \quad 7 \quad 1
\end{array}$$

We can verify that this —Pascal's Formula— holds for every ${}^{n}C_{r}$ where $n, r > 0$:

$$\begin{aligned}
{}^{n-1}C_{r} + {}^{n-1}C_{r-1} &= \frac{(n-1)!}{r!\,(n-1-r)!} + \frac{(n-1)!}{(r-1)!\,(n-\cancel{1}-(r-\cancel{1}))!} \\
&= \frac{n!/n}{r!\,(n-r)!/\,(n-r)} + \frac{n!/n}{(r!/r)\,(n-r)!} \\
&= \frac{n-r}{n} \cdot \frac{n!}{r!\,(n-r)!} + \frac{r}{n} \cdot \frac{n!}{r!\,(n-r)!} \\
&= \frac{n-r}{n} \cdot {}^{n}C_{r} + \frac{r}{n} \cdot {}^{n}C_{r} \;=\; \frac{n-\cancel{r}+\cancel{r}}{n} \cdot {}^{n}C_{r} \;=\; {}^{n}C_{r} \qquad (4)
\end{aligned}$$

Intuitively, if we have a set of $n-1$ elements and we add one more element, the number of combinations of r elements out of n is equal to:

- the number of combinations of r elements out of the original set —these do not include the new element—, plus
- the number of combinations formed by the new element and $r-1$ elements of the original set.

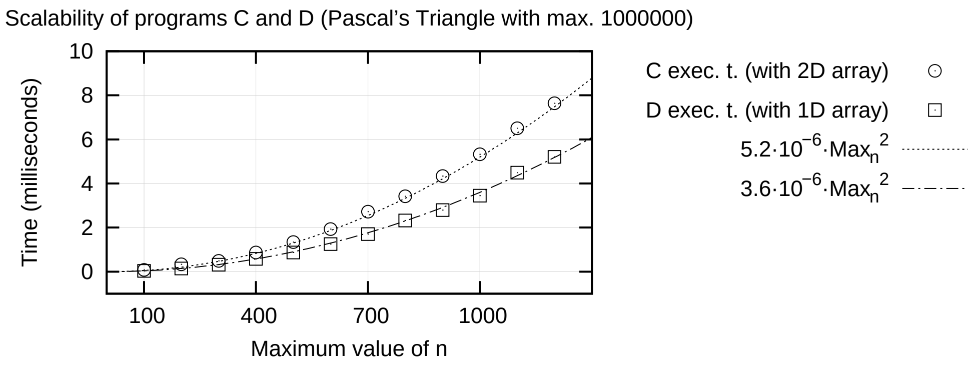
Therefore, we can simply compute Pascal's Triangle up to $n=100$. We don't even need to use big numbers because we are not interested in the exact value of those numbers that exceed one-million. We can simply store the value *one-million* instead, thus avoiding overflows.

```
count = 0                                                    Program C
C[1..100][0..100] = { uninitialized array }
for n = 1 to 100 do:
    C[n][0] = C[n][n] = 1
    for r = 1 to n-1 do:
        C[n][r] = C[n-1][r-1] + C[n-1][r]
        if C[n][r] > 1000000 do:
            count = count + 1
            C[n][r] = 1000000
print count
```

The previous program stores Pascal's Triangle in a 2D array, but every row is used only for computing the next row. The next program makes the same calculations using just a simple 1D array:

```
count = 0                                                    Program D
C[0..100] = { 1, 1, the rest can be uninitialized }
for n = 2 to 100 do:
    C[n-1] = 1
    for r = n-1 downto 1 do:
        C[r] = C[r] + C[r-1]
        if C[r] > 1000000 do:
            count = count + 1
            C[r] = 1000000
print count
```

These programs solve the original problem in 82 µs and 39 µs respectively[4]. The difference between them is probably due to cache effects and/or memory bandwidth usage. The next graphic shows their behaviour when the maximum value of n is increased:

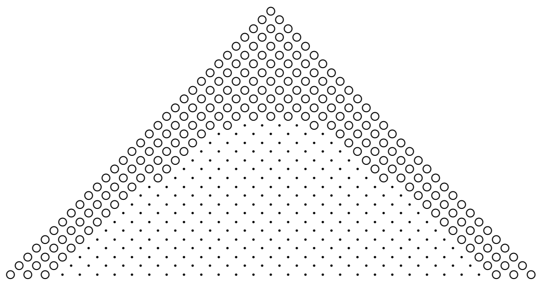


They are significantly faster than the initial programs. More importantly, when the limit is increased, the execution times follow parabolas of second degree. The execution time can be decreased taking benefit of the symmetry of Pascal's Triangle. The latter program, for example, takes 27 µs if this is taken into account. Ideally, it should take about 19 µs, but the additional logic tests and branches slow it down. With a further loop-unroll optimization it got to run in 20 µs.

[4]Average execution time in 100 consecutive repetitions.

In order to thoroughly test these algorithms, we will make experiments changing both parameters of the original problem. The maximum value of n will be henceforth called Max_n —originally 100—, and the limit value for ${}^nC_r$ will be called $Lim_{{}^nC_r}$ —originally 1,000,000—.

We can still make another dramatic improvement. Given the result obtained in equation (4), we can assume that, inside Pascal's Triangle, the area consisting in values greater than $Lim_{{}^nC_r}$ is in the centre, and it gets wider as we move down. The next figure, for instance, shows the area of Pascal's Triangle where ${}^nC_r \leq 1000$ —marked with '$\circ$'— and the area where ${}^nC_r > 1000$ —marked with '$\cdot$'—:



Therefore, in the inner loop, when a value grater than $Lim_{{}^nC_r}$ is found, the amount of such numbers in row n can be immediately deduced from the values of r and n:

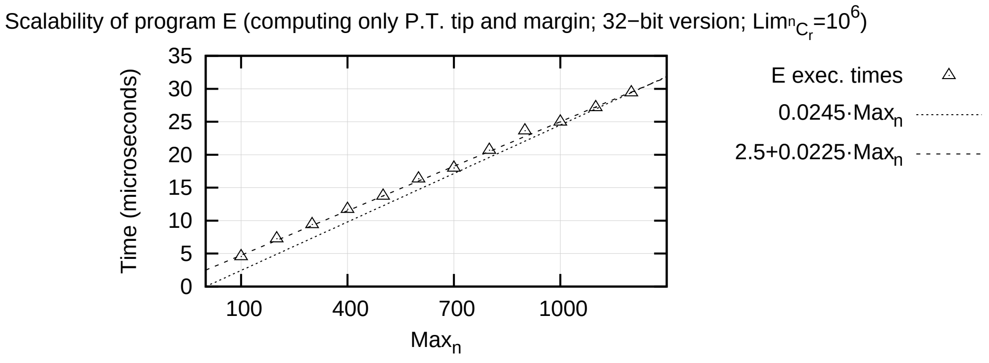
Program E
```
count = 0
C[0..$Max_n$] = { 1, 1, the rest can be uninitialized }
for n = 2 to $Max_n$ do:
    C[n-1] = 1
    r = n - 1
    while r >= 1 do:
        C[r] = C[r] + C[r-1]
        if C[r] > $Lim_{{}^nC_r}$ do:
            count = count + r - (n-r) + 1
            C[r] = $Lim_{{}^nC_r}$
            r = 0
        else do:
            r = r - 1
print count
```

The solution described in program E was posted to the thread of the problem by **hk** on 2008. Along with the code, he proposed the challenge of computing the solution for $Max_n = 10^5$ and $Lim_{{}^nC_r} = 10^{16}$.

The execution times of programs C and D depended mainly on Max_n. Changing $Lim_{{}^nC_r}$ could affect them too, but only by a small proportion. We can say that their execution time is proportional to Max_n^2. Put formally, they take $O(Max_n^2)$ time.

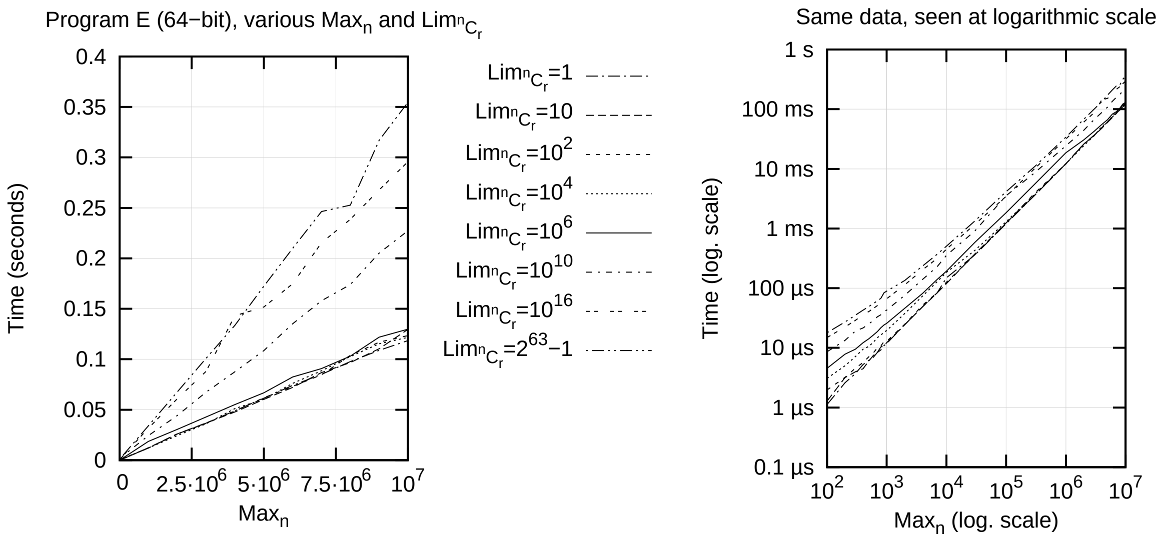
The case of program E is different. If $Lim_{{}^nC_r}$ would never be reached, program E would behave like program D. But the values of ${}^nC_r$ grow so rapidly as n is increased that they will eventually reach any practical $Lim_{{}^nC_r}$. Note that ${}^{35}C_{17} > 2^{32}$ and ${}^{68}C_{34} > 2^{64}$. After $Lim_{{}^nC_r}$ is reached, program E makes less and less iterations of the inner loop as n continues growing. Hence, we can say that program E takes $O(Max_n)$ time for most practical purposes.

The next graphic shows the behaviour of program E under the same conditions as the previous two programs —fixed $Lim_{^nC_r}$ of one-million, values of Max_n ranging from 100 to 1,200 and 32-bit variables for ^nC_r values—:



Since $Lim_{^nC_r}$ was kept constant, the tip of the triangle —where ^nC_r values were computed for every possible r— always had the same area. From $n = 23$ and on, only the margin was computed, and it got thinner and thinner as n grew. Thus, the time spent in computing the tip was better amortized with larger values of Max_n.

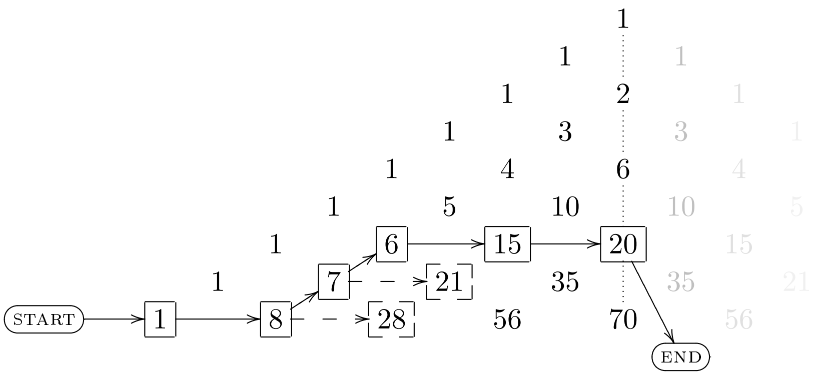
The next graphics show the behaviour of program E, adapted to 64-bits, under extreme conditions:



Note that not only Max_n, but also $Lim_{^nC_r}$ affects the execution times. The left graphic shows that high values of $Lim_{^nC_r}$ slow down the program by some proportion. The right graphic shows, at low values of Max_n, the penalty of calculating the tip of the triangle, whose area depends on $Lim_{^nC_r}$.

We will try now a different approach. Instead of computing the tip of the triangle and the margin at the right of the area where $^nC_r > Lim_{^nC_r}$, we can directly compute only the values of its boundary. The area where $^nC_r > Lim_{^nC_r}$ starts at some point in the middle of the triangle —at $^{23}C_{10}$ in the original problem— and it gets wider as we move down through the triangle. Therefore, we can traverse its border, calculating only some ^nC_r values.

The next figure shows an example where we search —or rather count— the values of ${}^nC_r$ greater than 20 where $1 \le n \le 8$:



If the value at the right is greater than the limit, we have to move up-right. Otherwise, we have to move right. Before we move up-right, we must count the values greater than the limit that we leave at the right in the current row. If the position before moving up-right is (n, r) then there are $n-2r-1$ such values.

Now we need to know how to compute, given the current position (n, r) and its value ${}^nC_r$ the value at the right and the value at the up-right position.

During the development of equation (4) we found that:

$$ {}^{n-1}C_r = \frac{(n-1)!}{r!\,(n-1-r)!} = \frac{n!/n}{r!\,(n-r)!/\,(n-r)} = \frac{n-r}{n} \cdot \frac{n!}{r!\,(n-r)!} = \frac{n-r}{n} \cdot {}^nC_r \tag{5} $$

Intuitively, if we have the combinations of r elements out of n, we can reduce them to be combinations of r elements out of $n-1$ by removing one of the n elements and, consequently, all those combinations that include it. Since every original combination excludes $n-r$ elements of n, the proportion of combinations that exclude the removed element is $\frac{n-r}{n}$.

Similarly, we can move to the right across Pascal's Triangle with:

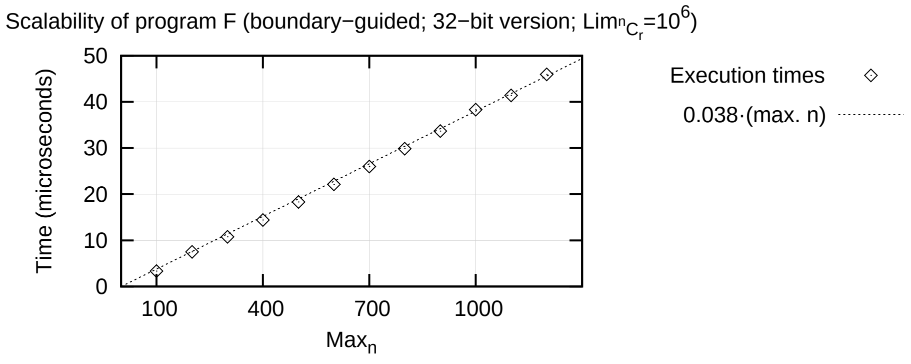
$$ {}^nC_{r+1} = \frac{n!}{(r+1)!\;(n-(r+1))!} = \frac{n!}{(r+1)\,r!\;\frac{(n-r)!}{n-r}} = \frac{n-r}{r+1} \cdot \frac{n!}{r!\,(n-r)!} = \frac{n-r}{r+1} \cdot {}^nC_r \tag{6} $$

Intuitively, if we have the combinations of r elements out of n, we can expand them to be combinations of $r+1$ elements by adding one of the remaining $n-r$ elements. Since we have $n-r$ different options, the number of combinations is multiplied by $n-r$. Though, while expanding the combinations in this way, we generated many repeated combinations. For all the combinations of r out of some $r+1$ elements there are now $r+1$ identical combinations containing these $r+1$ elements. That's why we have to divide by $r+1$.

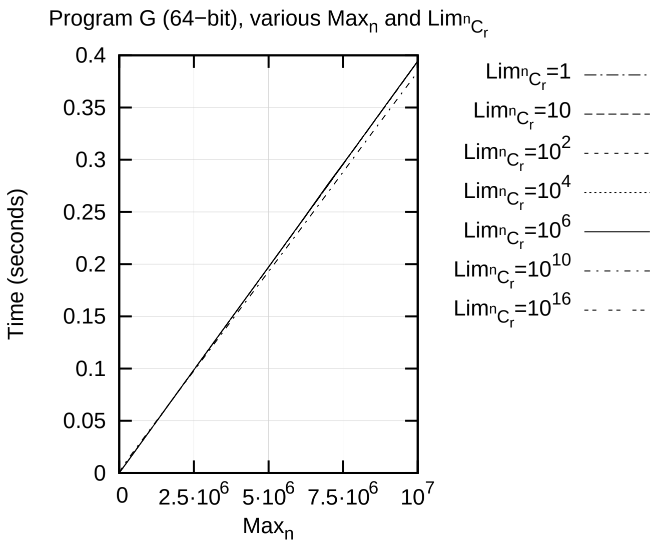
Thus, we can move right with ${}^nC_{r+1} = \frac{n-r}{r+1} \cdot {}^nC_r$ and up-right with ${}^{n-1}C_r = \frac{n-r}{n} \cdot {}^nC_r$.

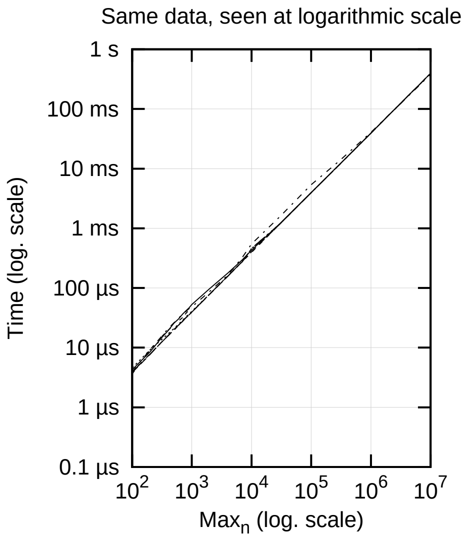
```
count = 0                                                  Program F
r = 0
n = $Max_n$
nCr = 1
while r < n/2 do:
    Cright = nCr * (n-r) / (r+1)
    if Cright <= $Lim_{^nC_r}$ do:
        r = r + 1
        nCr = Cright
    else do:
        Cupright = nCr * (n-r) / n
        count = count + n - 2*r - 1
        n = n - 1
        nCr = Cupright
print count
```

The next graphic shows the behaviour of this last program when Max_n is increased:



The execution time grows linearly, as expected. The next graphics show the behaviour of program F, adapted to 64-bits, under extreme conditions:





The case of $Lim_{^nC_r} = 2^{63} - 1$ was not tested, and the case of $Lim_{^nC_r} = 10^{16}$ was tested only up to $Max_n = 1000$. The reason is that the multiplications impose more strict limitations due to possible overflows. Program E only required $2 \cdot Lim_{^nC_r} < 2^{64}$ in order to work properly. Program F, instead, requires $Lim_{^nC_r} \cdot Max_n < 2^{64}$.

The execution time of program F seems to depend only on Max_n. Program F is a bit faster on some very particular cases —small Max_n, large $Lim_{^nC_r}$—. In general, program E is quite faster because it only makes additions, while program F makes multiplications and divisions.

Another important difference is that program F does not require any array. Put formally, program F requires just $O(1)$ additional space while program E requires $O(Max_n)$ additional space.

Program F computes less values of ^nC_r than program E. At first sight, program E looks slower because it covers an area —the tip and the right margin of the triangle—, while program F covers only a border of that area. Even though additions are faster than multiplications and divisions, one might expect the growth of the area to overcome this advantage at some point, slowing down program E. Nevertheless, program E is faster precisely with large values of Max_n. The reason is that the margin of the triangle, where $^nC_r \leq Lim_{^nC_r}$, is quite thin, getting narrower as Max_n grows.

We will rely on that fact to optimize[5] the last algorithm. When $Max_n = 10^4$ and $Lim_{^nC_r} = 10^6$, for example, program F follows the next path across Pascal's Triangle:

$^{10000}C_0$ $\mathbf{1}\!\rightarrow$ $^{10000}C_1$ $\mathbf{8586}\!\nearrow$ $^{1414}C_1$ $\mathbf{1}\!\rightarrow$ $^{1414}C_2$ $\mathbf{1232}\!\nearrow$ $^{182}C_2$ $\mathbf{1}\!\rightarrow$ $^{182}C_3$ $\mathbf{111}\!\nearrow$ $^{71}C_3$ $\mathbf{1}\!\rightarrow$ $^{71}C_4$ $\mathbf{28}\!\nearrow$ $^{43}C_4$ $\mathbf{1}\!\rightarrow$ $^{43}C_5$ $\mathbf{11}\!\nearrow$ $^{32}C_5$ $\mathbf{1}\!\rightarrow$ $^{32}C_6$ $\mathbf{5}\!\nearrow$ $^{27}C_6$ $\mathbf{1}\!\rightarrow$ $^{27}C_7$ $\mathbf{3}\!\nearrow$ $^{24}C_7$ $\mathbf{1}\!\rightarrow$ $^{24}C_8$ $\mathbf{1}\!\nearrow$ $^{23}C_8$ $\mathbf{1}\!\rightarrow$ $^{23}C_9$ $\mathbf{1}\!\nearrow$ $^{22}C_9$ $\mathbf{2}\!\rightarrow$ $^{22}C_{11}$

Most moves are up-right ($\nearrow$) and, more importantly, most of them are grouped in a few long runs at the beginning of the path. We can save many up-right moves by estimating the n value at which the next right move will take place.

By substituting terms in the numerator of equation (2) we can define an upper bound and a lower bound for ^nC_r:

$$^nC_r = \frac{n \cdot (n-1) \cdots (r+1)}{(n-r)!}, \qquad \text{therefore:} \quad \frac{(r+1)^{n-r}}{(n-r)!} \leq {^nC_r} \leq \frac{n^{n-r}}{(n-r)!} \tag{7}$$

These bounds are too loose for small values of r, but they are tight enough for high values of r. Taking benefit of the symmetry proven in equation (3), for low values of r:

$$^nC_r = {^nC_{n-r}} = \frac{n \cdot (n-1) \cdots (n-r+1)}{(\cancel{n} - (\cancel{n} - r))!} = \frac{n \cdot (n-1) \cdots (n-r+1)}{r!},$$

$$\text{therefore:} \quad \frac{(n-r+1)^r}{r!} \leq {^nC_r} \leq \frac{n^r}{r!} \tag{8}$$

Focusing on the lower bound, we can isolate n as follows:

$$\frac{(n-r+1)^r}{r!} \leq {^nC_r} \quad \leftrightarrow \quad (n-r+1)^r \leq {^nC_r} \cdot r! \quad \leftrightarrow \quad n \leq \sqrt[r]{^nC_r \cdot r!} + r - 1 \tag{9}$$

[5] This optimization was suggested by **hk** during the elaboration of this document.

During the first stage of the travel, every time we have to move up-right after moving right, we will check if we can save time by jumping to the row where the next move to the right might take place. In such situations, if the current position is (n, r), and ${}^nC_r \leq Lim_{{}^nC_r}$ but ${}^nC_{r+1} > Lim_{{}^nC_r}$, we will consider jumping to row n' where:

$$n' = \left\lceil \sqrt[r+1]{(Lim_{{}^nC_r} + 1) \cdot (r+1)!} \right\rceil + r \tag{10}$$

Jumping to row n' implies starting to compute values from ${}^{n'}C_0 = 1$, moving to the right until ${}^{n'}C_r$ is reached. Since this requires r steps, we will only do it if the number of steps saved is greater. That is, if $n - n' > r$.

Otherwise, if the *shortcut* is not shorter than the normal path, we will continue with the normal algorithm until the next move to the right. Furthermore, after the first estimation, if any estimation does not lead to a proper shortcut, no more estimations will be made.

In addition, before taking a shortcut, we will compute ${}^{n'}C_{r+1}$ too, in order to verify that it is greater than $Lim_{{}^nC_r}$. If it were not greater, then jumping to row n' might not be safe.

The pseudocode for program G, which implements this algorithm, is divided in three stages:

1. `initialise`: We walk through row Max_n horizontally searching for the first r where ${}^nC_r > Lim_{{}^nC_r}$. This usually takes only a few steps, because the ${}^nC_r$ values grow very rapidly.
2. `climb`: We try to take shortcuts. Every time we should move up-right after a move to the right, we estimate the row number n' in order to jump there, saving a lot of up-right moves. In general, if an estimation does not lead to a shortcut worth the effort, we will proceed to the final stage. The only exception is the first estimation, which might be useless just because the initial n happened to be near a critical point where we have to move right in our ascension.
3. `finalise`: “Shortcuts” are not *short* any more, so we continue with the normal algorithm, as in the previous program.

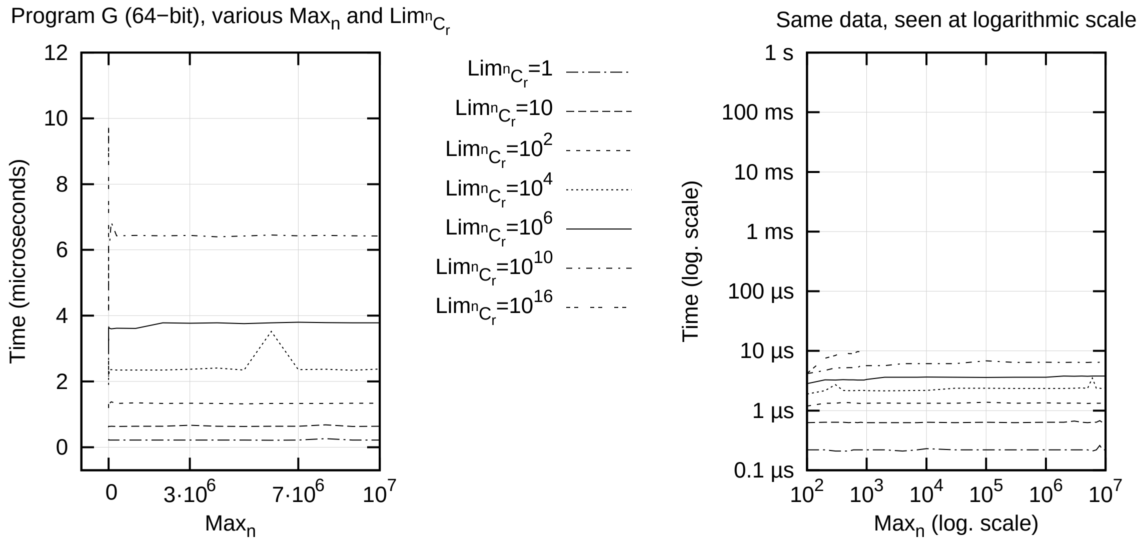
Program G

```
count = 0
n = $Max_n$
r = 0
nCr = 1
rFact = 1

procedure initialise
    while r < n/2 do:
        Cright = nCr * (n-r) / (r+1)
        if Cright <= $Lim_{{}^nC_r}$ do:
            nCr = Cright
            r = r + 1
            rFact = rFact * r
        else do:
            return
```

```
procedure climb
    firstTime = true
    while r < n/2 do:
        n′ = ceil ( power ( rFact*(r+1)*($Lim_{^nC_r}$+1), 1/(r+1) ) ) + r
        advantage = n - n′ - r
        if advantage <= 0 do:
            if not firstTime then return
        else do:
            r′ = 0
            nCr′ = 1
            while r′ < r do:
                nCr′ = nCr′ * (n′-r′) / (r′+1)
                r′ = r′ + 1
            if nCr′*(n′-r′)/(r′+1) <= $Lim_{^nC_r}$ do:
                print “The estimation was wrong! Falling back to slow mode...”
                return
            else do:
                count = count + (n-n′) * (n+n′-4*r-1) / 2
                n = n′
                nCr = nCr′
        while r < n/2 do:
            Cright = nCr * (n-r) / (r+1)
            if Cright > $Lim_{^nC_r}$
                count = count + n - 2*r - 1
                nCr = nCr * (n-r) / n
                n = n - 1
            else do:
                break out of inner “while” loop
        while r < n/2 do:
            Cright = nCr * (n-r) / (r+1)
            if Cright <= $Lim_{^nC_r}$ do:
                nCr = Cright
                r = r + 1
                rFact = rFact * r
            else do:
                break out of inner “while” loop
        firstTime = false

procedure finalise
    while r < n/2 do:
        Cright = nCr * (n-r) / (r+1)
        if Cright <= $Lim_{^nC_r}$ do:
            r = r + 1
            nCr = Cright
        else do:
            count = count + n - 2*r - 1
            nCr = nCr * (n-r) / n
            n = n - 1
```

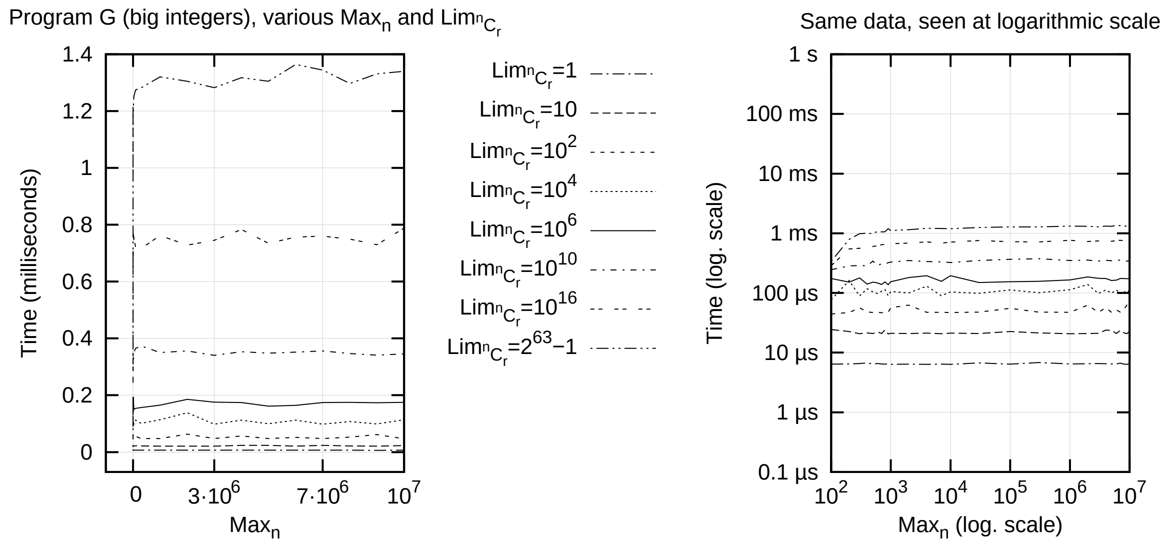
```
initialise
climb
finalise
print count
```

The next graphics show the behaviour of program G, adapted to 64-bits, under extreme conditions:



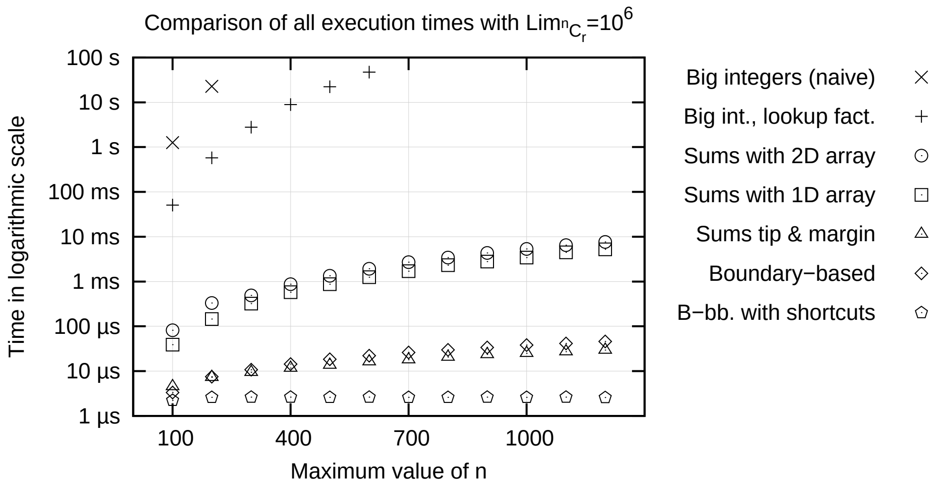
The execution time seems to depend mainly on $Lim_{{}^nC_r}$. Program G clearly outperforms the previous ones. Though, it suffers the same limitations as program F due to possible overflows.

The next graphics show the behaviour of program G implemented with arbitrarily long integers:



The execution times grow considerably with the use of big integers, but they are still interesting in many cases, compared to those of the previous programs.

To summarize, the next graphic depicts a comparison of all the previous programs, in their 32-bit versions, with $Lim_{^nC_r} = 10^6$. Note that the time scale is logarithmic, ranging from 1 microsecond to 100 seconds:



Returning to the line of thought abandoned at page 3, in light of the above findings, a good way to compute a single value of ^nC_r might be:

$$^nC_0 = 1 \quad \text{and} \quad ^nC_r \cdot \frac{n-r}{r+1} = {^nC_{r+1}} \quad \text{therefore} \quad ^nC_r = \prod_{0 \le i < r} \frac{n-i}{i+1} \tag{11}$$

In pseudocode, with a little optimization based on the symmetry of Pascal's Triangle:

```
function combinations (n, r)
    if (r > n-r)
        r = n-r
    res = 1
    for i = 0 to r-1 do:
        res = res * (n-i) / (i+1)
    retrun res
```

In any case, this function is of little use in the problem at hand. If we applied it directly, calling it from nested loops like those of the first programs, then we'd get rid of the burden of big integers. Though, we would have some limitations due to overflows, and execution times would still grow proportional to Max_n^3. Put formally, the computational complexity would be $O(Max_n^3)$.

The intermediate results of the function are other values ^nC_r —with smaller values of r—, which are useful in this problem. If we took benefit from that, then the resulting program would only take $O(Max_n^2)$ time, but it would still be slower than the sum-based programs by a fixed proportion. On the plus side, it wouldn't require any array.