Combinatorics

E-OLYMP 318. Binomial coefficients 1 Let n be a non-negative integer. Let

$$n! = 1 * 2 * ... * n \ (0! = 1),$$

$$C_n^k = \frac{n!}{k!(n-k)!}$$

You are given the numbers n and k. Calculate C_n^k.

► Calculations will be performed in 64-bit unsigned integers (unsigned long long). Its obvious that

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{n\cdot(n-1)\cdot(n-2)\cdot...\cdot(n-k+1)}{1\cdot2\cdot3\cdot...\cdot k} = \frac{n}{1}\cdot\frac{n-1}{2}\cdot\frac{n-2}{3}\cdot...\cdot\frac{n-k+1}{k}$$

Let's assign the variable *res* to the value 1. Then multiply it by $\frac{n-i+1}{i}$ for all i from 1 to k. Each time the division by i will give integer result, but multiplication can give overflow. Let d = GCD(*res* , i). Then let's rewrite the operation

$$res = res * (n - i + 1) / i$$

as

$$res = (res / d) * ((n - i + 1) / (i / d))$$

In this implementation we'll avoid the overflow (the answer is 64--bit unsigned integer). Note that first we need to perform the division $(n - i + 1) / (i / d)$, and then multiply *res* / d by the resulting quotient.

To compute C_n^k, we must run k iterations. But what if we need to find $C_{2000000000}^{1999999999}$? The answer is no more than 2^{64}, so such input values are possible. As long as $C_n^k = C_n^{n-k}$, then for $n - k < k$ we should assign $k = n - k$.

Consider the next sample:

$$C_6^3 = \frac{6}{1}\cdot\frac{5}{2}\cdot\frac{4}{3} = 15\cdot\frac{4}{3}$$

Let *res* = 15, and we need to make a multiplication $res * \frac{4}{3} = 15 * \frac{4}{3}$. Compute d = GCD(15 , 3) = 3. So $15 * \frac{4}{3} = (15 / 3) * \frac{4}{(3/3)} = 5 * \frac{4}{1} = 20$.

E-OLYMP 3260. How many? Once preparing for the exam, Peter put in front of him n different cheating papers of his "favorite" subject "Calculus". And as during the semester Peter did not learn properly, there were so many cribs that they all can not fit into any pocket. Peter then found the maximum number of cribs, that he can take with him to the exam, and suddenly thought: how many ways are there to choose the right number of cribs?

► To implement the function ***Cnk*** for calculating the *binomial coefficient*, we'll use the relation:

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{n\cdot(n-1)\cdot(n-2)\cdot...\cdot(n-k+1)}{1\cdot2\cdot3\cdot...\cdot k}$$

Let's declare a variable *res*, initialize it with 1. Then multiply it by *n* and divide by 1. Next multiply it by *n* – 1 and divide by 2. The process of multiplication and division will be continued *k* times (the numerator and denominator of C_n^k after simplifying contains *k* factors).

For recursive implementation of the binomial coefficient, use the recurrence relation:

$$C_n^k = \begin{cases} C_{n-1}^{k-1} + C_{n-1}^k, n > 0 \\ 1, k = n \text{ or } k = 0 \end{cases}$$

```
int Cnk(int n, int k)
{
  if (n == k) return 1;
  if (k == 0) return 1;
  return Cnk(n - 1, k - 1) + Cnk(n - 1, k);
}
```

E-OLYMP **9892. C0n +… + Cnn** Given non-negative integer *n*, find the sum of binomial coefficients

$$C_n^0 + C_n^1 + ... + C_n^n$$

► The Newton's binomial formula has the form:

$$(a+b)^n = \sum_{i=0}^{n} C_n^i a^i b^{n-i}$$

If we assign $a = b = 1$, we get the relation:

$$(1+1)^n = \sum_{i=0}^{n} C_n^i 1^i 1^{n-i}$$

or

$$2^n = \sum_{i=0}^{n} C_n^i = C_n^0 + C_n^1 + ... + C_n^n$$

Thus, the indicated sum is 2^n.

If $n = 1$, then $C_1^0 + C_1^1 = 1 + 1 = 2$;
If $n = 2$, then $C_2^0 + C_2^1 + C_2^2 = 1 + 2 + 1 = 4$;
If $n = 3$, then $C_3^0 + C_3^1 + C_3^2 + C_3^3 = 1 + 3 + 3 + 1 = 8$.

E-OLYMP **5329. Party** In how many ways can we choose among *n* students exactly *k* of them, who will get yogurt? Print the answer modulo 9929.

► The answer to the problem is $C_n^k \bmod 9929$. Since it is necessary to find the binomial coefficient by modulo, you must avoid division during calculations. To do this, use the relation $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$, $C_n^0 = 1$.

For example, here is an iterative implementation:

```
#define MAX 502
#define MOD 9929
int cnk[MAX][MAX];
```

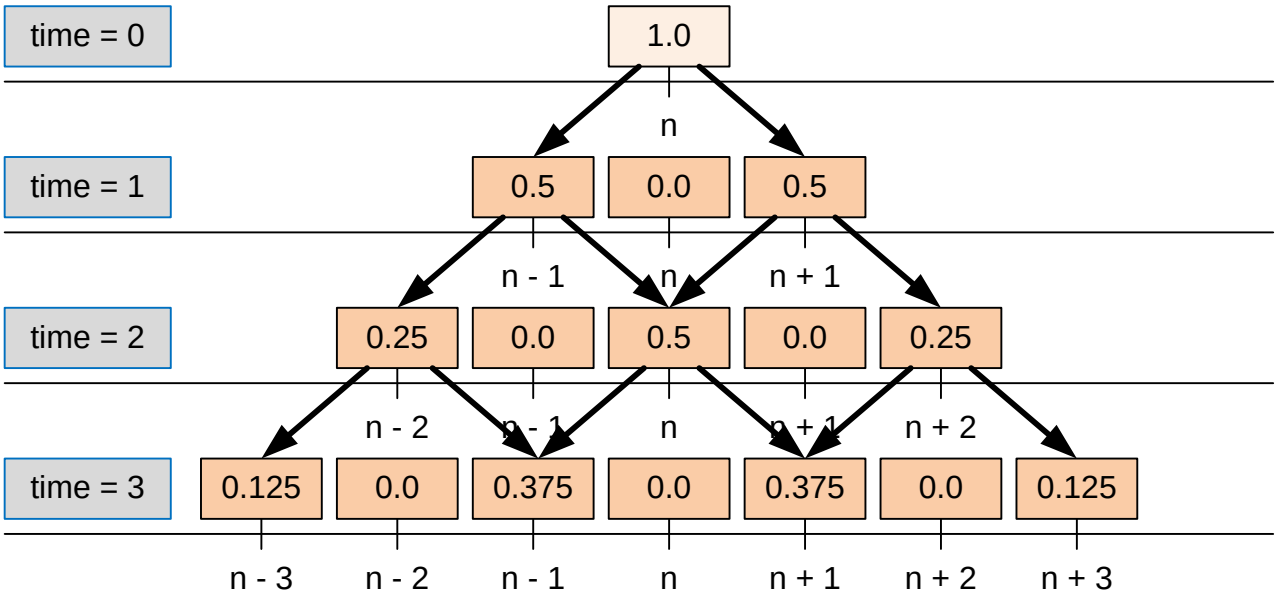
```
void FillCnk(void)
{
  int n, k;
  memset(cnk,0,sizeof(cnk));
  for(n = 0; n < MAX; n++) cnk[n][0] = 1;

  for(n = 1; n < MAX; n++)
  for(k = 1; k <= MAX; k++)
    cnk[n][k] = (cnk[n-1][k] + cnk[n-1][k-1]) % MOD;
}
```

E-OLYMP **7261. Difficult path** Alex drank a lot this night and now when he reached his street, he has completely lost the sense of direction. Since he can not remember where his house is, he chooses direction randomly. Moreover, at every crossroads there is a 50% chance that he will keep going forward or turn around and go back. He so lost his touch with reality that he can even walk past his own house and not notice it!

Having passed *n* blocks, Alex falls asleep right on the street. When he wakes up, he wonders what were the chances that he slept near his house? From the crossroads, where he started his way to the crossroads near his house there are m blocks. Help him.

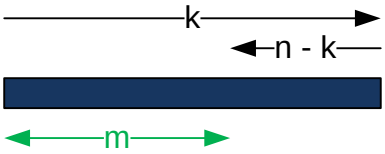
► Let d be a two dimensional array where d[*time*][*x*] is a probability of being at the point with abscissa *x* at time *time*. Let initially (at time $t = 0$) Alex is located at the point with the abscissa $x = n$. Then d[0][*n*] = 1.



Let's evaluate d[*i*][*j*] – the probability that Alex at time *i* will be at position *j*. For this Alex must be located at time $i - 1$ either at position $j - 1$, or at position $j + 1$. Then at time *i* he can come from them to position *j* with probability 50%. So

$$d[i][j] = (d[i - 1][j - 1] + d[i - 1][j + 1]) / 2$$

Let's consider the mathematical solution of the problem. We encode the Alex' path with sequence of 0 and 1. Let 1 means move to the right and 0 means move to the left. Let among *n* Alex' steps *k* steps he did right. Then $n - k$ steps he did left.



We are interested to find the probability that Alex moved himself in one of the sides (for example to the right) m blocks. Then we must have: $m + n - k = k$, where $k = (m + n) / 2$. The number of sequences of length n with k ones equals to C_n^k. Since Alex made n movements, he has 2^n different ways to choose the path. So the probability that Alex pass to the right m blocks equals to $C_n^k / 2^n$, where $k = (m + n) / 2$. Note that desired probability equals to 0, if $m + n$ is even. In this case Alex is not able to reach home ($m + n = 2k$ is even).

Let Alex makes $n = 3$ steps.
If $m = 1$, then $k = (3 + 1) / 2 = 2$ and probability equals to $C_3^2 / 2^3 = 3 / 8 = 0.375$.
If $m = 3$, then $k = (3 + 3) / 2 = 3$ and probability equals to $C_3^3 / 2^3 = 1 / 8 = 0.125$.
Let Alex makes $n = 4$ steps.
If $m = 0$, then $k = (4 + 0) / 2 = 2$ and probability equals to $C_4^2 / 2^4 = 6 / 16 = 0.375$.
If $m = 2$, then $k = (4 + 2) / 2 = 3$ and probability equals to $C_4^3 / 2^4 = 4 / 16 = 0.25$.
If $m = 4$, then $k = (4 + 4) / 2 = 4$ and probability equals to $C_4^4 / 2^4 = 1 / 16 = 0.0625$.

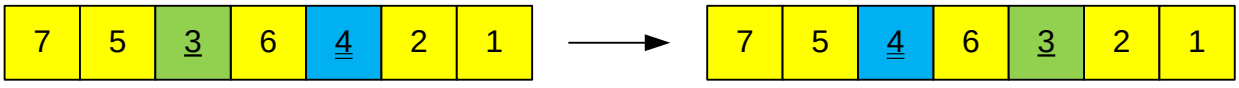
Permutations

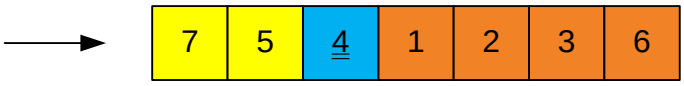
E-OLYMP 2386. The next permutation Find the next permutation. Assume that permutation (n, $n - 1$, ..., 2, 1) is followed by the identity (1, 2, ..., $n - 1$, n).

► To generate the next permutation, we'll use the ***next_permutation*** function. For the lexicographically largest permutation (n, $n - 1$, …, 2, 1) the next is the lexicographically smallest one (1, 2, ..., $n - 1$, n).

Iterate through the current permutation from right to left until the next number is greater than the previous one. Stop when the rule is broken. Mark (underline) this position: (5, 6, 7, 4, 3). Again iterate the traversed path (from right to left) until we reach the first number that is greater than the marked one. The place of the second stop is marked with double underlining: (5, 6, 7, 4, 3). Swap the marked numbers: (5, 7, 6, 4, 3). Now sort all the numbers to the right of the double underlined integer in ascending order. Since they have so far been ordered in descending order, it is enough to reverse the indicated segment. We get Q = (5, 7, 3, 4, 6). This permutation is next after P.

Find the permutation following P = (7, 5, 3, 6, 4, 2, 1).

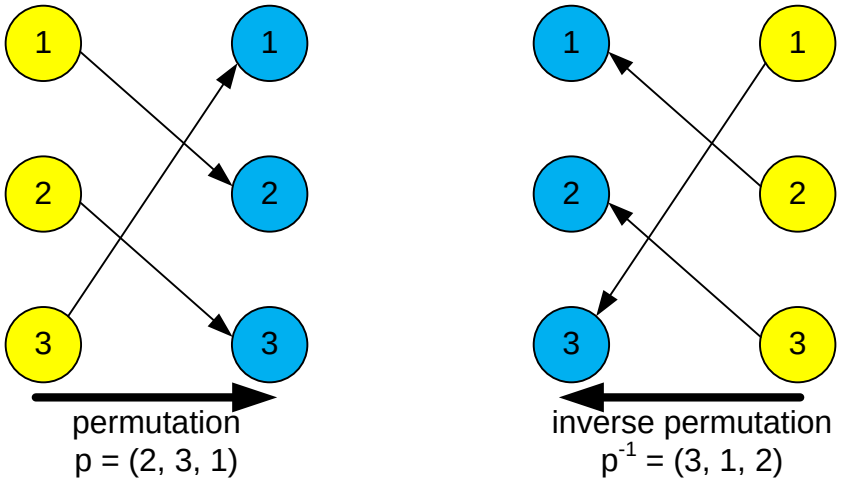




E-OLYMP 1788. Inverse permutation Given a permutation *p*, find the inverse p^{-1}.

► Let *p* be a given permutation. It means that element from position *i* is moved to position p[*i*], that is, the transformation $i \rightarrow$ p[*i*] takes place. The inverse for *p* is such a permutation *pi* for which the inverse transformation p[*i*] $\rightarrow i$ takes place. That is, in the *pi* array at the p[*i*]-th place there must be number *i* (pi[p[*i*]] = *i*).

Consider the permutation $p = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$ and its inverse $p^{-1} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$. In the inverse permutation the edges are oriented in the other direction.



Generate permutations

E-OLYMP 2169. Permutations Given a positive integer *n*, print all permutations of the integers from 1 to *n* in lexicographical order.

► In the problem you should generate all permutations of numbers from 1 to *n*. This can be done, for example, using the ***next_permutation*** function.

Use array m to generate permutations.

```
int m[10];
```

Read the value of *n*. Initialize the array m with the initial permutation 1 2 3…. *n* starting from the first index.

```
scanf("%d",&n);
for(i = 1; i <= n; i++) m[i] = i;
```

Using the ***next_permutation*** function, generate all permutations: from the lexicographically smallest to the lexicographically largest.

```
do
{
```

Print the next permutation on a separate line.

```
  for(i = 1; i <= n; i++)
    printf("%d ",m[i]);
  printf("\n");
} while(next_permutation(m+1,m+n+1));
```

E-OLYMP **1533. Anagram generation** You are to write a program that has to generate all possible words from a given set of letters.

Example: Given the word "*abc*", your program should – by exploring all different combination of the three letters – output the words "*abc*", "*acb*", "*bac*", "*bca*", "*cab*" and "*cba*".

In the word taken from the input file, some letters may appear more than once. For a given word, your program should not produce the same word more than once, and the words should be output in alphabetically ascending order.

► 1. What is the difference between *lexicographic* and *alphabetical* sorting?

2. Consider the string zAZaaZ. Sort the letters in alphabetical and lexicographic order.

3. What STL function can be used to generate permutations of all letters in a word?

4. The function ***sort***, by default, sorts the letters in a word in lexicographic order. How to implement with it an alphabetical sorting?

5. Implement a comparator for alphabetical sort `int lt(char a, char b)`, which is passed as the third argument to the ***sort*** function.

6. How to find the alphabetically smallest permutation of letters in the string *s*?

Sort the characters of the input string in ascending order. Use the built-in ***next_permutation*** function to generate all permutations. However, you should write your own function to compare the characters. In standard (lexicographic) comparison, any uppercase letter is less than any lowercase letter. That is, when sorting letters a, A, z, Z, r, R, we get the word ARZarz. In this problem you should sort (and generate permutations) in accordance with the alphabetical order AaBbCc… Zz, so it is necessary to obtain AaRrZz from the letters a, A, z, Z, r, R.

For the string **aAb** (1-st test case) the smallest permutation would be **Aab**, and the biggest would be **baA**.

The function ***lt*** will be used for sorting and generating permutations. It compares two characters according to alphabetical order AaBbCc ... Zz.

```
int lt(char a, char b)
{
  if (toupper(a) != toupper(b)) return (toupper(a) < toupper(b));
  return (a < b);
}
```

Read the input string, calculate its length, and sort the characters alphabetically.

```
scanf("%s", &s);len = strlen(s);
sort(s,s+len,lt);
```

Print the current anagram (permutation of characters) and generate the next one until its possible.

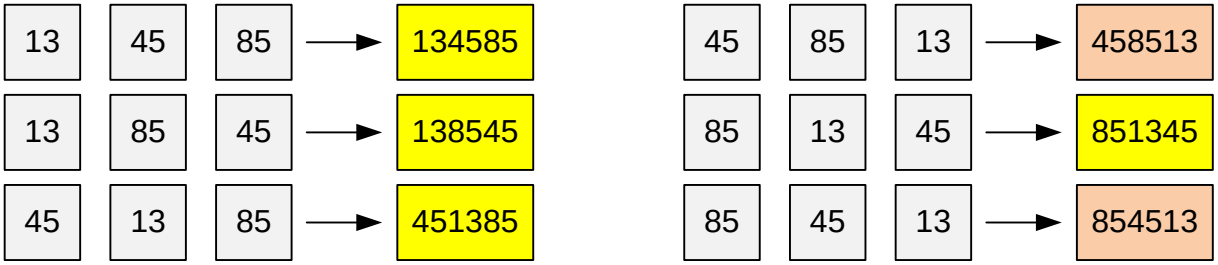
```
do {
  printf("%s\n",s);
} while(next_permutation(s,s+len,lt));
```

E-OLYMP 50. The cut number Vasylko wrote a number which is multiple of *d* on a scrap of paper. His smaller brother Dmytro cut the number into *k* parts. Vasylko decided to restore the number that he wrote. He remembered only number *d*. But there was a problem. It is possible to make many numbers which are multiple of *d*.

How many numbers multiple of *d* can make Vasylko? He must use all parts to make a number.

► Put the numbers of all parts into the array. Consider all possible gluing of the available parts. Such full search is possible, since $k < 9$ and there will be no more than 9! different gluings. For each number obtained by gluing, check is it divisible by *d*.

Consider all possible gluings of three parts 13, 85 and 45. For each number obtained, check whether it is divisible by $d = 5$.



Count the objects

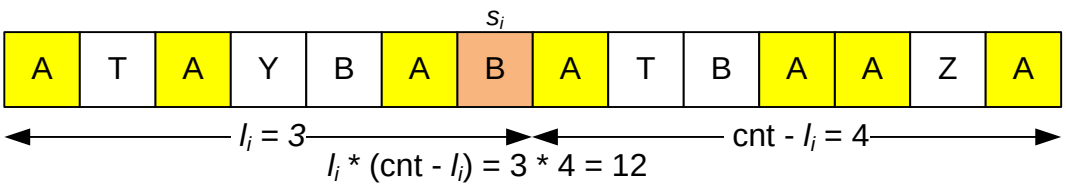
E-OLYMP 9890. Balls and boxes Find the number of ways to put *n* different balls into *k* boxes. You can put into each box any number of balls (including zero).

► Take the first ball. It can be placed in one of the *k* boxes available. That is, there are *k* ways for putting the first ball into the box. Similarly, each next ball can also be put into one of *k* boxes (there are *k* ways for each ball).

Therefore, there are $k * k * \ldots * k = k^n$ ways to arrange all the balls in the boxes.

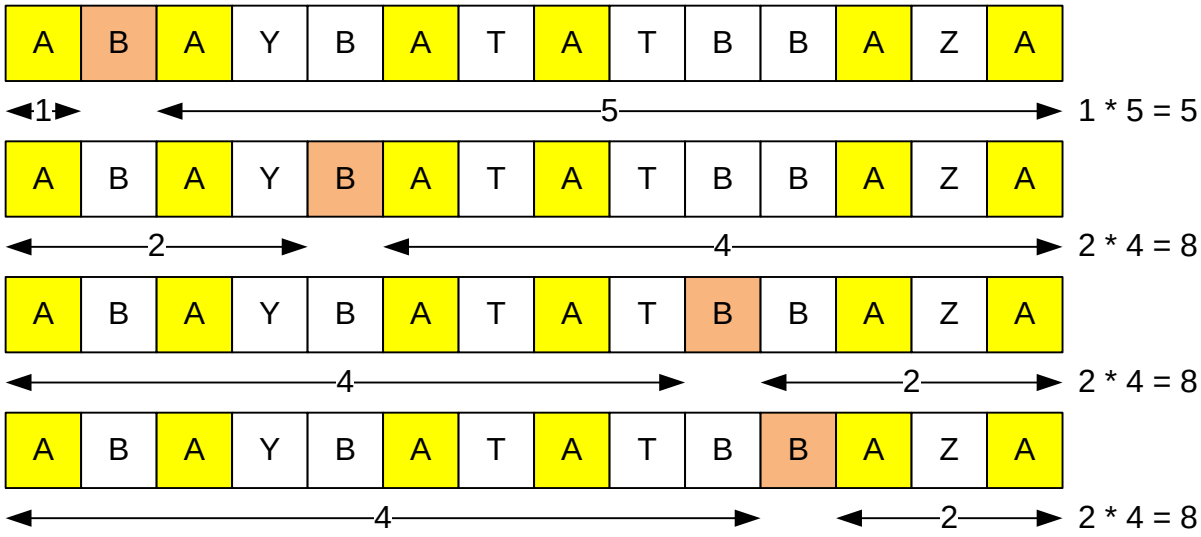
E-OLYMP 9594. ABA The string of letters is given. How many subsequences "ABA" does it contain? The letters "ABA" don't have to be consecutive. But the order of letters should be the same.

► Let $s_0 s_1 \ldots s_{n-1}$ be the input string. Let l[*i*] contains the number of letters A in positions from zero to *i*-th inclusive. Let *cnt* be the number of letters A in the word. Then to the right of position *i* there are *cnt* – l[*i*] letters A.



If the letter B is in the *i*-th position, then to the left of it there are l[*i*] letters A, and to the right (*cnt* – l[*i*]) letters A. The number of subsequences "ABA" in which the letter B is in the *i*-th position, equals to l[*i*] * (*cnt* – l[*i*]).

Consider the second test case.

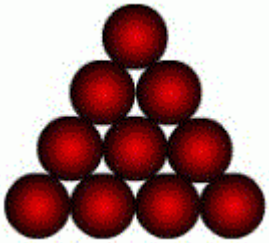


The number of subsequences "ABA" is

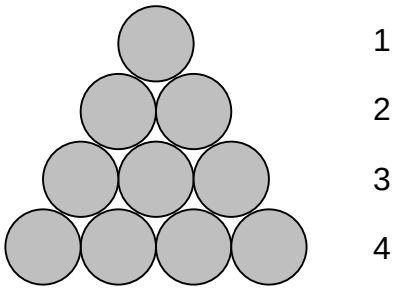
$$1 * 5 + 2 * 4 + 2 * 4 + 2 * 4 = 5 + 8 + 8 + 8 = 29$$

E-OLYMP 4538. Bob and balls Recently Bob learned that balls can be played in a very entertaining game. In this game you want to stack the balls in the form of various geometric shapes. Just now Bob is engaged in laying the balls in the form of an equilateral triangle. But here's the thing: sometimes Bob do not have enough balls, and he wants to know, what is the greatest side of the triangle for which it is enough balls? Help Bob to count the value of *n* – the length of equilateral triangle for given number of balls *k*.

Below given the example of placing the balls in the form of an equilateral triangle:



► If *n* is the length of the side of triangle, then for its complete packing it is required $1 + 2 + \ldots + n = n * (n + 1) / 2$ balls.



1 + 2 + 3 + 4 = 10

There are k balls available. Solve the equation $n * (n + 1) / 2 = k$ and round the non-negative root down to the nearest integer.

Solve the quadratic equation: $n^2 + n - 2k = 0$, $d = 1 + 8k$, $n = \left(-1+\sqrt{1+8k}\right)/2$.

The answer is the value $\left\lfloor\left(-1+\sqrt{1+8k}\right)/2\right\rfloor$.

E-OLYMP 1548. Diagonal The number of diagonals of an n-gon is not less than N. What is the minimum possible value of n?

► Each point of the polygon is connected by segments with $n - 1$ other points. These line segments form 2 sides and $n - 3$ diagonals. Since there are n points in the polygon, and $n - 3$ diagonals get out from each point, the number of diagonals of a convex n - gon is $n * (n - 3) / 2$ (each diagonal is counted twice).

If $n * (n - 3) / 2 = N$, then the value of n can be found from the quadratic equation

$$n^2 - 3 * n - 2 * N = 0$$

The positive root of the equation is

$$n = \frac{3+\sqrt{9+8N}}{2}$$

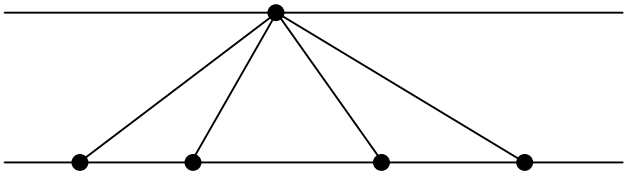
It remains to round up the computed value. Since $N \le 10^{15}$, the calculations should be performed using ***long long*** data type.

Consider the second test case. For N = 100 we have

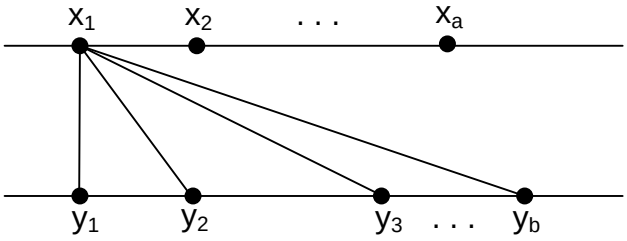
$$n = \left\lceil \frac{3+\sqrt{9+8\cdot 100}}{2} \right\rceil = 16$$

E-OLYMP 1539. How many points of intersection? We have two rows. There are a dots on the top row and b dots on the bottom row. We draw line segments connecting every dot on the top row with every dot on the bottom row. The dots are arranged in such a way that the number of internal intersections among the line segments is maximized. To achieve this goal we must not allow more than two line segments to intersect in a point. The intersection points on the top row and the bottom are not included in our count; we can allow more than two line segments to intersect on those two rows. Given the value of a and b, your task is to compute P(a, b), the number of intersections in between the two rows. For example, in the following figure $a = 2$ and $b = 3$. This figure illustrates that P(2, 3) = 3.

► Let f(a, b) be the required number of intersection points. Obviously, f(1, b) = 0, since for a = 1 no two segments intersect.



Consider the general case. Let x_1, x_2, …, x_a be points on the first line, y_1, y_2, …, y_b be points on the second line. Connect point x_1 with poinys y_1, y_2, …, y_b. There will be no intersection points on the segment x_1y_1. The segment x_1y_2 will contain the points of intersection with the segments y_1x_2, y_1x_3, …, y_1x_a (a – 1 points in total). The segment x_1y_j will contain the points of intersection with the segments y_ix_k, where $i < j$, $2 \le k \le a$ ((j – 1) * (a – 1) points in total). The number of intersection points that lie on the segments outgoing from x_1 is (0 + 1 + 2 + … + (b – 1)) * (a – 1) = b * (b – 1) / 2 * (a – 1).



So, out of f(a, b) points b * (b – 1) / 2 * (a – 1) points lie on segments outgoing from x_1, and the rest of the points lie on segments with ends at x_2, …, x_a. We have a recurrent relation:

$$f(a, b) = b * (b - 1) / 2 * (a - 1) + f(a - 1, b)$$

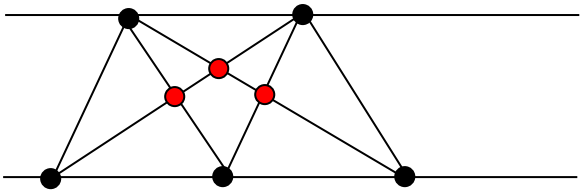
Expanding it, we get:

$$f(a, b) = b * (b - 1) / 2 * (a - 1) + f(a - 1, b) =$$
$$b * (b - 1) / 2 * (a - 1) + b * (b - 1) / 2 * (a - 2) + f(a - 2, b) = ... =$$
$$b * (b - 1) / 2 * (a - 1) + b * (b - 1) / 2 * (a - 2) + ... + b * (b - 1) / 2 * 1 =$$
$$b * (b - 1) / 2 * ((a - 1) + (a - 2) + ... + 1) =$$
$$b * (b - 1) / 2 * a * (a - 1) / 2$$

Thus, the maximum number of intersection points is

$$\frac{a \cdot (a-1)}{2} * \frac{b \cdot (b-1)}{2}$$

Consider the second test case, where a = 2, b = 3. The maximum possible number of intersection points among the segments is 3 and this is shown in the figure:



Generate the sequences

E-OLYMP **4835. Without two consecutive ones** Given positive integer n, print all binary sequences of length n without consecutive ones, in lexicographical order.

► Let x be an integer. In binary representation number x contains two ones in a row if x & (x << 1) is not zero.

Iterate in the variable x over the numbers from 0 to $2^n - 1$. If number x in the binary representation does not contain two ones in a row, then output such binary representation.

Read value of n.

```
scanf("%d",&n);
```

Iterate over the numbers from 0 to $2^n - 1$.

```
for(x = 0; x < (1 << n); x++)
{
```

If number x contains two ones in a row in its binary representation, then skip it.

```
  if (x & (x << 1)) continue;
```

In one line print the binary code of the number x from left to right as a sequence of 0 and 1.

```
  for(i = n - 1; i >= 0; i--)
    printf("%d ",(x >> i) & 1);
  printf("\n");
}
```

Consider the recursive solution.

```
#include <stdio.h>

int n;
int m[22];

void gen(int n, int pos)
{
  if (n <= 0)
  {
    for(int i = 0; i < pos + n; i++)
      printf("%d ",m[i]);
    printf("\n");
    return;
  }
  m[pos] = 0; gen(n-1,pos+1);
  m[pos] = 1; m[pos+1] = 0; gen(n-2,pos+2);
}

int main(void)
{
```

```
  scanf("%d",&n);
  gen(n,0);
  return 0;
}
```

E-OLYMP **1663. Bracket sequences** Positive integer n ($1 \le n \le 10$) is given. Print in alphabetical order all correct bracket sequences of length $2n$, assuming that the character '(' comes before ')' in the alphabet.

A correct bracket sequence is either an empty string, or a string like (S), where S is a correct bracket sequence, or a string like S_1S_2, where S_1 and S_2 are correct bracket sequences.

► Solve the problem by exhaustive search. Let *s* be a string where we'll generate the required sequences. It is initially empty. Let the variables *left* and *right* contain the number of unused open and closed parentheses, respectively (initially *left* = *right* = *n*). Then:

- If *left* > 0, we can append '(' to the string *s*.
- If *right* > 0, we can append ')' to the string *s*.

In this case, the number of already used open parentheses should not be less than the number of closed ones. When *left* = *right* = 0, print the next correct bracket sequence.

Function ***gen*** generates the sequences. Part of the already generated bracket sequence is contained in string *s*. The variables *left* and *right* contain the number of unused open and closed parentheses, respectively.

```
void gen(string s, int left, int right)
{
  if(left > right) return;
  if(left == 0 && right == 0)
  {
    printf("%s\n",s.c_str());
    return;
  }

  if(left > 0)  gen(s + "(", left - 1, right);
  if(right > 0) gen(s + ")", left, right - 1);
}
```

The main part of the program. Read the value of *n* and start generating the bracket sequences.

```
scanf("%d",&n);
gen("",n,n);
```

Generate the subsets

E-OLYMP **4106. Subsets generation** The set *s* of cardinality *n* is given. It contains all the elements in the range [1 .. *n*]. Generate all its subsets.

► In this problem it is necessary to generate all subsets of the given set. To do this, we iterate over (in a loop of i) all numbers from 1 to $2^n - 1$. Represent the number i in binary notation and consider its last n bits (possibly with leading zeros). A subset corresponds to such a binary representation: if there is **one** in its k-th place, then the number k ($1 \le k \le n$) is included in the subset. For example, if $n = 3$ and $i = 2$, then its bit representation is 010 and the set {2} corresponds to it.

Consider the subsets for $n = 3$.

	3	2	1	
0	0	0	0	{ }
1	0	0	1	{ 1 }
2	0	1	0	{ 2 }
3	0	1	1	{ 2, 1 }

	3	2	1	
4	1	0	0	{ 3 }
5	1	0	1	{ 3, 1 }
6	1	1	0	{ 3, 2 }
7	1	1	1	{ 3, 2, 1 }