Recursion

Recursion:

A *recursive algorithm* is an algorithm that calls itself on “smaller” input (smaller in size or values or both).

A **recursive function** consists of two types of cases:

- ***a base case(s)***
- ***a recursive case***

The **base case** is a small problem

- the solution to this problem should not be recursive, so that the function is guaranteed to terminate;
- there can be more than one base case;

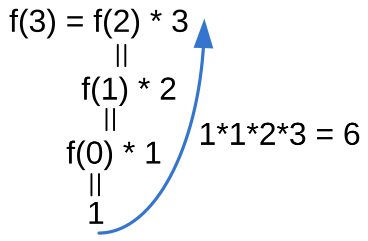
The **recursive case** defines the problem in terms of a smaller problem of the same type

- the recursive case includes a recursive function call
- there can be more than one recursive case

From the definition of factorial we can conclude that

$$n! = (1 * 2 * 3 * \dots * (n-1)) * n = (n-1)! * n$$

Or if we denoted f(n) = n! then f(n) = f(n – 1) * n. This is called ***recursive case***. We continue the recursive process till n = 0, when 0! = 1. So f(0) = 1. This is called the ***base case***.



$$\begin{cases} f(n) = f(n-1) * n & \textit{recursive case} \\ f(0) = 1 & \textit{base case} \end{cases}$$

```
int fact(int n)
{
  if (n == 0) return 1;
  return fact(n-1) * n;
}
```

Analysis of Algorithms:

Time complexity T(n): number of operations in the algorithm, as a function of the input size;

Space complexity S(n): number of memory words needed by the algorithm;

Since memory has become very cheap and abundant, we rarely care about space complexity. ***Time***, however, is always a premium even if computers are always increasing in speed.

Since speed slows down for very large input sizes, the time estimate can focus more on large input sizes n, and we thus should be more concerned about the "order of growth" of the time function T(n), or as typically called, the *asymptotic behavior* of the T(n). Since computers vary in speed from model to model and from generation to generation, and the variation is by a constant factor (with respect to input size), we can (and should) ignore constant factors in time estimations, and focus again on the order of growth rather than the precise time in micro/nano-seconds.

Asymptotics and Big-O Notation

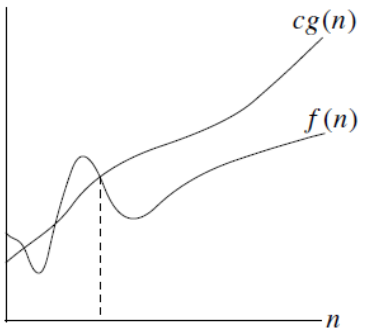
Big-O notation

Definition. Let f(n) and g(n) be two functions of n (n is usually the input size in algorithm analysis). We say that

$$f(n) = O(g(n))$$

if $\exists\ n_0 \in N$ and constant $c > 0$ such that $|f(n)| \leq c|g(n)|\ \forall n \geq n_0$.

O-notation gives an ***upper bound*** for a function to within a constant factor.



$f(n) = O(g(n))$

Example. $3n + 1 = O(n^2)$ since $3n + 1 \leq 3n^2\ \forall n \geq 2$. $n_0 = 2$, $c = 3$.
Example. $3n + 6 = O(n)$ because $3n + 6 \leq 4n\ \forall\ n \geq 6$. $n_0 = 6$, $c = 4$.
Example. $an + b = O(n^k)$ for any $a > 0$, $k \geq 1$.

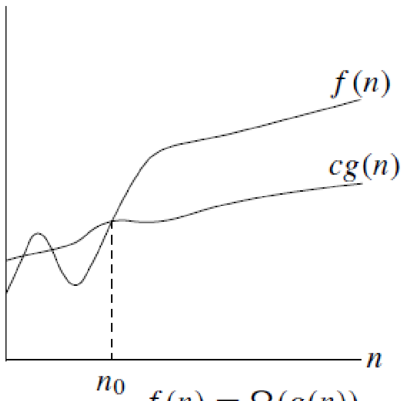
Big Omega **(Ω)**

Definition. Let f(n) and g(n) as above. We say that

$$f(n) = \Omega(g(n))$$

if $\exists n_0 \in N$ and constant $c > 0$ such that $|f(n)| \geq c|g(n)|\ \forall n \geq n_0$.

Ω-notation gives a ***lower bound*** for a function to within a constant factor.



$f(n) = \Omega(g(n))$

Example. $\frac{n^2}{3} = \Omega(n)$ because $\frac{1}{3}n^2 \geq n \ \ \forall n \geq 3$. $n_0 = 3$, $c = 1$.
Example. $3n + 6 = \Omega(n)$ because $3n + 6 \geq 3n \ \ \forall n \geq 1$. $n_0 = 1$, $c = 3$.
Example. $an^{10} + b = \Omega(n^k)$ *for any* $a > 0$, $0 \leq k \leq 10$.

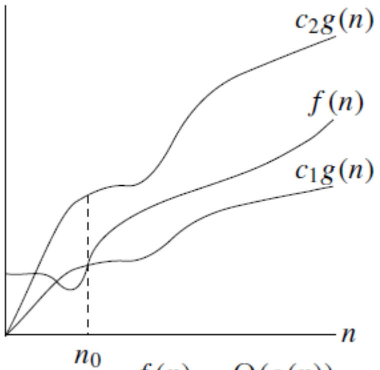
Big Theta (Θ)
Definition. Let f(n) and g(n) as above. We say that

$$f(n) = \Theta(g(n))$$

if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. That is,
if $\exists n_0 \in N$ and two positive constants $c_1 > 0$ and $c_2 > 0$ such that

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \ \forall n \geq n_0$$

Θ-notation ***bounds*** a function to within constant factors.



$f(n) = \Theta(g(n))$

Example. $3n + 6 = \Theta(n)$ because $3n + 6 = O(n)$ and $3n + 6 = \Omega(n)$.

Example. $3n^2 + 2n - 6 = \Theta(n^2)$ because $n^2 \leq 3n^2 + 2n - 6 \leq 100n^2$ ($c_1 = 1$, $c_2 = 100$) starting from $n \geq n_0 = 10$.

Theorem. Let $f(n) = a_m n^m + a_{m-1}n^{m-1} + \ldots + a_1 n + a_0$ be a polynomial (in n) of degree m, where m is a positive constant integer, and $a_m, a_{m-1}, \ldots, a_1, a_0$ are constants. Then $f(n) = O(n^m)$.

Proof. $|f(n)| \leq |a_m|n^m + |a_{m-1}|n^{m-1} + \ldots + |a_1|n + |a_0| \leq$

$$\leq |a_m|n^m + |a_{m-1}|n^m + \ldots + |a_1|n^m + |a_0|\, n^m \leq$$
$$\leq (|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|)\, n^m \leq cn^m,$$

where $c = |a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|$ and $n \geq 1$. Therefore, by definition, $f(n) = O(n^m)$.

In general, if the time T(n) is a sum of a constant number of terms, you can keep the largest-order term and drop all the other terms, and drop the constant factor of the largest order term, to get a simple Big-O form for T(n).

Example. If T(n) = $3n^{2.7} + n\sqrt{n} + 7n\log n$, then T($n$) = $O\left(n^{2.7}\right)$.

Master theorem. Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = a * T(n / b) + f(n),$$

where we interpret n / b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) can be bounded asymptotically as follows:

1. If f(n) = $O\left(n^{\log_b a-\varepsilon}\right)$ for some constant $\varepsilon > 0$, then T(n) = $\Theta\left(n^{\log_b a}\right)$.
2. If f(n) = $\Theta\left(n^{\log_b a}\right)$, then T($n$) = $\Theta\left(n^{\log_b a}\log n\right)$.
3. If f(n) = $\Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant $\varepsilon > 0$, and if a * f(n / b) $\leq c$ * f(n) for some constant $c < 1$ and all sufficiently large n, then T(n) = Θ(f(n)).

Simplified form of Master theorem. To apply the master's theorem, we must calculate the value of p(n) = $n^{\log_b a}$.

1. If f(n) < p(n), then T(n) = $\Theta\left(n^{\log_b a}\right)$ = Θ(p(n)).
2. If f(n) = p(n), then T(n) = $\Theta\left(n^{\log_b a}\log n\right) = \Theta\left(p(n)\log n\right)$.
3. If f(n) > p(n), then T(n) = Θ(f(n)).

Example. T(n) = 9T(n / 3) + n.
Here a = 9, b = 3, f(n) = n. p(n) = $n^{\log_b a}$ = $n^{\log_3 9}$ = n^2.
f(n) < p(n) because $n < n^2$, so T(n) = Θ(n^2).

We have $n^{\log_b a}$ = $n^{\log_3 9}$ = Θ(n^2). Since f(n) = O($n^{\log_3 9-\varepsilon}$), where ε = 1, we can apply case 1 of the master theorem and conclude that the solution is T(n) = Θ(n^2).

Example. T(n) = T(2n / 3) + 1.
Here a = 1, b = 3 / 2, f (n) = 1, p(n) = $n^{\log_b a}$ = $n^{\log_{3/2} 1}$ = n^0 = 1.
f(n) = p(n) because 1 = 1, so T(n) = Θ(log n).

We have $n^{\log_b a}$ = $n^{\log_{3/2} 1}$ = n^0 = Θ(1). Case 2 applies, since f(n) = Θ ($n^{\log_b a}$) = Θ(1), and thus the solution to the recurrence is T(n) = Θ(log n).

Example. T(n) = 3T(n / 4) + $n$log$n$.
Here a = 3, b = 4, f(n) = $n$log$n$, p(n) = $n^{\log_b a}$ = $n^{\log_4 3}$ = $n^{0.793}$.
f(n) > p(n) because $n$log$n$ > $n^{0.793}$, so T(n) = Θ($n$log$n$).

We have $n^{\log_b a}$ = $n^{\log_4 3}$ = Θ($n^{0.793}$). Since f(n) = $\Omega\left(n^{\log_4 3+\varepsilon}\right)$, where ε ≈ 0.2, case 3 applies if we can show that the regularity condition holds for f(n). For sufficiently large n, a * f (n / b) = 3(n / 4) lg(n / 4) ≤ (3 / 4) $n$lg$n$ = c * f(n) for c = 3 / 4. Consequently, by case 3, the solution to the recurrence is T(n) = Θ($n$log$n$).

The master theorem ***cannot*** be used if:

- T(n) is not monotone, for example T(n) = sin n;
- f(n) is not polynomial, for example f(n) = 2^n;
- a is not a constant, for example $a = 2n$;

Example. *Master theorem solver*:
https://www.nayuki.io/page/master-theorem-solver-javascript

Problems. Solve the recurrence relations:
$T(n) = 4T(n / 2) + n$;
$T(n) = 4T(n / 2) + n^2$;
$T(n) = 4T(n / 2) + n^3$;
$T(n) = 2T(n / 4) + \sqrt{n}$;
$T(n) = 6T(n / 3) + n$;
$T(n) = 6T(n / 3) + n^2$;
$T(n) = 6T(n / 3) + n\sqrt{n}$;
$T(n) = 9T(n / 3) + n^2$;

E-OLYMP 2860. Sum of integers on the interval Find the sum of all integers from a to b. Integers are no more than 10^9 by absolute value.

► Let's solve the problem with ***for*** loop:

```
res = 0;
for(i = a; i <= b; i++)
  res = res + i;
```

Number of iterations is proportional to amount of numbers on the interval [a..b]. Let $n = b - a + 1$ be the size of the interval. To run a program, we must make n iterations in the ***for*** loop. For example, if $n = 10^9$, we must make 10^9 iterations. Number of operations increase linearly with the value of n. So time complexity is T(n) = O(n).

The speed of nowadays computers is approximately 10^9 operations per 2 seconds. So we can also estimate the running time of our programs in seconds.

Time limit for the problem **2860 (Sum of integers on the interval)** is 1 second. So *for loop* solution will give **Time Limit Exceeded** (TLE) on some test cases. We must find an algorithm faster than O(n).

We can notice that numbers from a to b form an arithmetic progression with difference $d = 1$. And their sum according to the formula equals to

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

Solution to the problem can be just one line:

```
res = (a + b) * (b - a + 1) / 2;
```

This formula has 5 arithmetic operations regardless the value of *n*. So complexity of this solution is O(1) and it is accepted in 1 second.

Example. Consider the next triple loop with complexity O(n^3).

time_t represents the system time and date as some sort of integer. Function time(0) or time(NULL) returns number of seconds since January 1, 1970.

Change the value of *n* and estimate the running time of the program.

```
#include <stdio.h>
#include <ctime>

int i, j, k, n;
long long cnt;

int main(void)
{
  // Number of sec since January 1,1970
  time_t start = time(0);
  printf("Number of seconds started: %lld\n", start);

  n = 1000; // 10^9 operations per 2 seconds, CORE i5
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
  for (k = 1; k <= n; k++)
    cnt++;

  printf("Counter = %lld\n", cnt);
  time_t finish = time(0);
  printf("Number of seconds finished: %lld\n", finish);

  printf("Running time of the program in seconds: %lld\n", finish - 
start);
  return 0;
}
```

Using the function **clock**(), you can estimate the running time in milliseconds. The C library function **clock**(**void**) returns the number of clock ticks elapsed since the program was launched. To get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC.

Try to run the program with *n* = 1000 and *n* = 2000.

```
#include <stdio.h>
#include <ctime>

int i, j, k, n;
long long cnt;

int main(void)
{
  clock_t start = clock();

  n = 1000;
  for (i = 1; i <= n; i++)
```

```
   for (j = 1; j <= n; j++)
   for (k = 1; k <= n; k++)
    cnt++;

   printf("Counter = %lld\n", cnt);
   clock_t finish = clock();
   // now you can see running time milliseconds
   printf("Running   time   of   the   program   in   seconds:   %f\n",
(float)(finish - start) / CLOCKS_PER_SEC);
   return 0;
  }
```

E-OLYMP 1616. Prime number? Check if the given number n is prime. The number is ***prime*** if it has no more than two divisors: 1 and the number itself.

► If number n is composite, it has a divisor not greater than $\lfloor \sqrt{n} \rfloor$. To check if n is prime, we must check its divisibility by 2, 3, …, $\lfloor \sqrt{n} \rfloor$. Complexity $O(\sqrt{n})$.

```
int IsPrime(int n)
{
  for (int i = 2; i <= sqrt(1.0*n); i++)
    if (n % i == 0) return 0;
  return 1;
}
```

E-OLYMP 8669. All divisors Find all divisors of positive integer n ($n \le 10^9$).

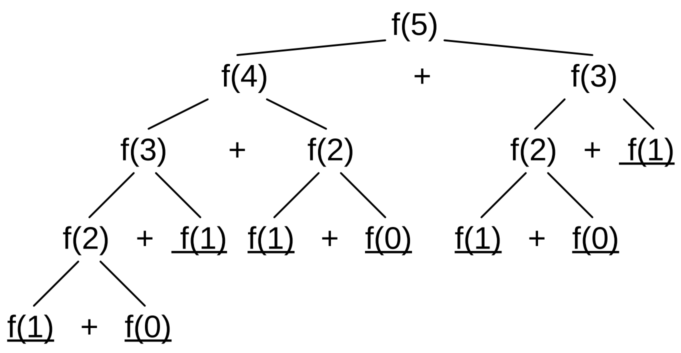
► If d is a divisor of n, then n / d is also a divisor of n. Find divisors d such that $1 \le d \le \lfloor \sqrt{n} \rfloor$, and corresponding to them divisors n / d. Sort and print divisors. Be careful if $d = \lfloor \sqrt{n} \rfloor$, then n / d is the same divisor, do not print it twice. Complexity $O(\sqrt{n})$.

E-OLYMP 4730. Fibonacci ***Fibonacci numbers*** is a sequence of numbers F(n), given by the formula:

$$F(0) = 1,$$
$$F(1) = 1,$$
$$F(n) = F(n-1) + F(n-2)$$

Given the value of n, print the n-th Fibonacci number.

► First let's consider the direct implementation of recursion.

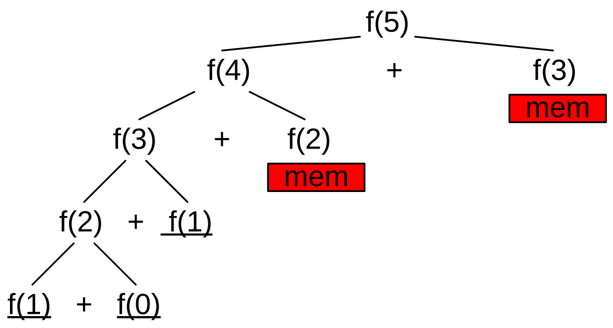


```
int f(int n)
{
  if (n == 0) return 1;
  if (n == 1) return 1;
```

```
    return f(n - 1) + f(n - 2);
  }
```

This solution has complexity O(2^n) because execution tree has a form of binary tree, and complete binary tree has no more than 2^n nodes. For n = 45 we must execute 2^{45} operations, that is very big for 1 second (time limit for this problem)

We can notice that some calculations done multiple times. For example, after finding f(3), we can store this value in fib[3] (let's declare integer array `int fib[46]`), and when again we need to find f(3), we can take this value out of fib[3] (and not to run all calculations again). This technique is called **memorization**. Complexity of recursion with memorization is O(n) because each value f(n) is calculated only once.



```
  #include <stdio.h>
  #include <string.h>

  int n, fib[46];

  int f(int n)
  {
    // base case
    if (n == 0) return 1;
    if (n == 1) return 1;

    // if the value fib[n] is ALREADY found, just return it
    if (fib[n] != -1) return fib[n];

    // if the value fib[n] is not found, calculate and memorize it
    return fib[n] = f(n-1) + f(n - 2);
  }

  int main(void)
  {
    scanf("%d",&n);

    // fib[i] = -1 means that this value is not calculated yet
    memset(fib,-1,sizeof(fib));

    printf("%d\n",f(n));
    return 0;
  }
```

Memorization of Fibonacci numbers can also be done with just one for loop, O(n) complexity:

```
fib[0] = 1; fib[1] = 1;
for (int i = 2; i < MAX; i++)
  fib[i] = fib[i - 1] + fib[i - 2];
```

Let's again look at Fibonacci recurrence and try to estimate its grows.

$$F(n) = F(n-1) + F(n-2)$$

Since $F(n-1) > F(n-2)$, we have $F(n) > 2 * F(n-2)$. From this inequality we have:

$$F(n) > 2 * F(n-2) > 4 * F(n-4) > \ldots > 2^{n/2} = \sqrt{2}^n = 1.4142^n$$

From the other hand, $F(n) = F(n-1) + F(n-2) < 2 * F(n-1)$. From this inequality we have:

$$F(n) < 2 * F(n-1) < 4 * F(n-2) < 2^n$$

So, Fibonacci numbers satisfy the inequality:

$$1.4142^n < F(n) < 2^n$$

Fibonacci numbers (F_n) are related to the golden ratio φ and to its conjugate $\overline{\varphi}$, which are given by the following formulas:

$$\varphi = \left(1+\sqrt{5}\right)/2 \approx 1.61803\ldots,$$

$$\overline{\varphi} = \left(1-\sqrt{5}\right)/2 \approx -0.61803\ldots,$$

$$F_n = \frac{\varphi^n - \overline{\varphi}^n}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Since $|\overline{\varphi}| < 1$, we have $|\overline{\varphi}|^n/\sqrt{5} < 1/\sqrt{5} < 1/2$, so that the ith Fibonacci number F_n is equal to $\varphi^n/\sqrt{5}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially, time complexity is $O(1.61^n)$.

The **greatest common divisor** (gcd) of two integers is the largest positive integer that divides each of the integers. For example, gcd(8, 12) = 4.

It is also known that gcd(0, x) = $|x|$ (absolute value of x) because $|x|$ is the biggest integer that divides 0 and x. For example, gcd(-6, 0) = 6, gcd(0, 5) = 5.

To find gcd of two numbers, we can use iterative algorithm: subtract smaller number from the bigger one. When one of the numbers becomes 0, the other equals to gcd. For example, gcd(10, 24) = gcd(10, 14) = gcd(10, 4) = gcd(6, 4) = gcd(2, 4) = gcd(2, 2) = gcd(2, 0) = 2.

If instead of "minus" operation we'll use "mod" operation, calculations will go faster.

a	b
10	24
10	14
10	4
6	4
2	4
2	2
2	0

a	b
2	9
2	7
2	5
2	3
2	1
1	1
1	0

9 mod 2 = 1

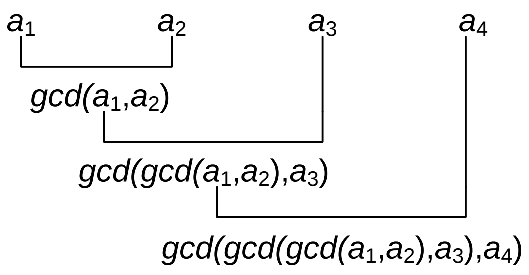
Greater Common Divisor: GCD(a, b) = $\begin{cases} GCD(a \bmod b, b), a \geq b \\ GCD(a, b \bmod a), a < b \\ a, b = 0 \\ b, a = 0 \end{cases}$,

```
int gcd(int a, int b)
{
  if (a == 0) return b;
  if (b == 0) return a;
  if (a >= b) return gcd(a % b, b);
  return gcd(a, b % a);
}
```

Complexity $O(\log_2(a+b))$.

E-OLYMP 137. GCD Find the Greatest Common Divisor of n numbers.

► Use function gcd of two arguments to find gcd of n integers.



Least Common Multiple (lcm) can be found from the formula:

$$\gcd(a, b) * \text{lcm}(a, b) = a * b$$

E-OLYMP 9643. LCM of n numbers Given list of integers a_1, a_2, ..., a_n. Find the value of LCM($a_1 * a_1$, $a_2 * a_2$, ..., $a_n * a_n$) mod m, where LCM means Least Common Multiple.

► Use function lcm of two arguments to find lcm of n integers.

Power x^n. How to find this value if x and n are given? We can use just simple loop with complexity O(n) like

```
res = 1;
for (i = 1; i <= n; i++)
  res = res * x;
```

Can we do it faster? For example, $x^{10} = (x^5)^2 = (x * x^4)^2 = (x * (x^2)^2)^2$.
We can notice that $x^{2n} = (x^2)^n$, or $x^{100} = (x^2)^{50}$.

$$x^n = \begin{cases} (x^2)^{n/2}, n \text{ is even} \\ x \cdot x^{n-1}, n \text{ is odd} \\ 1, n = 0 \end{cases}$$

```
int f(int x, int n)
{
  if (n == 0) return 1;
  if (n % 2 == 0) return f(x * x, n / 2);
  return x * f(x, n - 1);
}
```

Complexity O($\log_2 n$).

E-OLYMP 5198. Modular Exponentiaion Find the value of a^b mod m.
► Use function f(a, b, m) = a^b mod m.

E-OLYMP 9557. Bins and balls There are n bins in a row. There is also an infinite supply of balls of n distinct colors. Place exactly one ball into each bin, with the restriction that adjacent bins cannot contain balls of the same color. How many valid configurations of balls in bins are there?

► Any of n balls can be put into the first box. The color of the ball in the second box must not match the color of the ball in the first box. Therefore, you can put any ball of n – 1 colors in the second box. In the i-th box, you can put a ball of any color that does not match the color of the ball in the (i – 1)-th box.

1	2	3	4		n
n	n-1	n-1	n-1	...	n-1

Thus, the number of different arrangements of balls in the boxes equals to

$$n * (n-1)^{n-1} \bmod 10^9 + 7$$

Binomial coefficient: $C_n^k = \begin{cases} C_{n-1}^{k-1} + C_{n-1}^k, n > 0 \\ 1, k = n \\ 1, k = 0 \end{cases}$, where $C_n^k = \dfrac{n!}{k!(n-k)!}$

Proof. $C_{n-1}^{k-1} + C_{n-1}^k = \dfrac{(n-1)!}{(k-1)!(n-k)!} + \dfrac{(n-1)!}{k!(n-k-1)!} =$

$$\frac{(n-1)!(k+n-k)}{k!(n-k)!} = \frac{n!}{k!(n-k)!}$$

```
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 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 is $C_n^k \bmod 9929$. Use formula given above plus memorization and modular operation.

Stirling formula. $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

$$C_{2n}^n = \frac{(2n)!}{n!n!} \approx \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^2} = \frac{4^n}{\sqrt{\pi n}}$$

Problems / Solutions. Solve the recurrence relations:

- $T(n) = 4T(n / 2) + n$;

$a = 4$, $b = 2$, $f(n) = n$, $p(n) = n^{\log_b a} = n^{\log_2 4} = n^2$.
$f(n) < p(n)$ because $n < n^2$, so $T(n) = \Theta(n^2)$.

We have $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$.
$f(n) = n = O(n^{\log_2 4-\varepsilon})$ for $\varepsilon = 1$, $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$.

- $T(n) = 4T(n / 2) + n^2$;

$a = 4$, $b = 2$, $f(n) = n^2$, $p(n) = n^{\log_b a} = n^{\log_2 4} = n^2$.
$f(n) = p(n)$ because $n^2 = n^2$, so $T(n) = \Theta\left(n^2 \log n\right)$.

We have $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$.
$f(n) = n = \Theta(n^{\log_2 4}) = \Theta(n^2)$, $T(n) = \Theta\left(n^{\log_b a}\log n\right) = \Theta\left(n^{\log_2 4}\log n\right) = \Theta\left(n^2 \log n\right)$.

- $T(n) = 4T(n / 2) + n^3$;

$a = 4$, $b = 2$, $f(n) = n^3$, $p(n) = n^{\log_b a} = n^{\log_2 4} = n^2$.
$f(n) > p(n)$ because $n^3 > n^2$, so $T(n) = \Theta(n^3)$.

We have $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$.
$f(n) = n^3 = \Omega\left(n^{\log_2 4+\varepsilon}\right)$ for $\varepsilon = 1$, $T(n) = \Theta(n^3)$.

- $T(n) = 2T(n / 4) + \sqrt{n}$;

$a = 2$, $b = 4$, $f(n) = \sqrt{n}$, $p(n) = n^{\log_b a} = n^{\log_4 2} = n^{1/2} = \sqrt{n}$.
$f(n) = p(n)$ because $\sqrt{n} = \sqrt{n}$, so $T(n) = \Theta\left(\sqrt{n} \log n\right)$.

We have $n^{\log_b a} = n^{\log_4 2} = \Theta(n^{1/2}) = \Theta(\sqrt{n})$.
$f(n) = \sqrt{n} = \Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$, $T(n) = \Theta\left(n^{\log_b a} \log n\right) = \Theta\left(\sqrt{n} \log n\right)$.

- $T(n) = 6T(n / 3) + n$;

$a = 6$, $b = 3$, $f(n) = n$, $p(n) = n^{\log_b a} = n^{\log_3 6} = n^{1.631}$.
$f(n) < p(n)$ because $n < n^{1.631}$, so $T(n) = \Theta(n^{1.631})$.

We have $n^{\log_b a} = n^{\log_3 6} = \Theta(n^{1.631})$.
$f(n) = n = O(n^{\log_3 6 - \varepsilon})$, $\varepsilon \approx 0.631$, $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_3 6}\right) = \Theta(n^{1.631})$.

- $T(n) = 6T(n / 3) + n^2$;

$a = 6$, $b = 3$, $f(n) = n^2$, $p(n) = n^{\log_b a} = n^{\log_3 6} = n^{1.631}$.
$f(n) > p(n)$ because $n^2 > n^{1.631}$, so $T(n) = \Theta(n^2)$.

We have $n^{\log_b a} = n^{\log_3 6} = \Theta(n^{1.631})$.
$f(n) = n^2 = \Omega\left(n^{\log_3 6 + \varepsilon}\right)$, $\varepsilon = 0.2$, $T(n) = \Theta(n^2)$.

- $T(n) = 6T(n / 3) + n\sqrt{n}$;

$a = 6$, $b = 3$, $f(n) = n\sqrt{n} = n^{1.5}$, $p(n) = n^{\log_b a} = n^{\log_3 6} = n^{1.631}$.
$f(n) < p(n)$ because $n^{1.5} < n^{1.631}$, so $T(n) = (n^{1.631})$.

We have $n^{\log_b a} = n^{\log_3 6} = \Theta(n^{1.631})$.
$f(n) = n^{1.5} = O(n^{\log_3 6 - \varepsilon})$, $\varepsilon = 0.1$, $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_3 6}\right) = \Theta(n^{1.631})$.

- $T(n) = 9T(n / 3) + n^2$;

$a = 9$, $b = 3$, $f(n) = n^2$, $p(n) = n^{\log_b a} = n^{\log_3 9} = n^2$.
$f(n) = p(n)$ because $n^2 = n^2$, so $T(n) = (n^2 \log n)$.

We have $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.
$f(n) = n^2 = \Theta\left(n^{\log_3 9}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right) = \Theta\left(n^{\log_3 9} \log n\right) = \Theta(n^2 \log n)$.