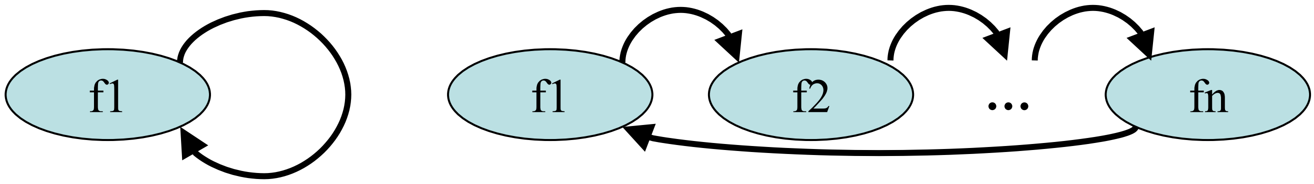
Recursion

The **recursive function** is
- a kind of function that calls itself, or
- a function that is part of a cycle in the sequence of function calls.



Let's we want to find the ***factorial*** of a number: f(n) = n! We know that

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

For example, f(5) = 1 * 2 * 3 * 4 * 5. We also know that f(4) = 1 * 2 * 3 * 4. So

$$f(5) = (1 * 2 * 3 * 4) * 5 = f(4) * 5$$

The problem of calculating f(5) is ***reduced*** to the problem of calculating f(4): in order to find f(5) we first must find f(4) and then multiply the result by 5. This process can be continues like

$$f(5) = f(4) * 5 = f(3) * 4 * 5 = f(2) * 3 * 4 * 5 = \ldots$$

How long shall we continue this process? We know that 0! = 1, but there is no sense for calculating factorial for negative numbers. The equality 0! = 1 or f(0) = 1 is called ***simple case*** or ***terminating case*** or ***base case***. When we need to find f(0), we do not continue the reduction like f(0) = f(-1) * 0 because it has no sense, but simply substitute the value of f(0) by 1. So

$$f(2) = f(1) * 2 = f(0) * 1 * 2 = 1 * 1 * 2 = 2$$

A **recursive function** consists of two types of cases:
- ***a base case(s)***
- ***a recursive case***

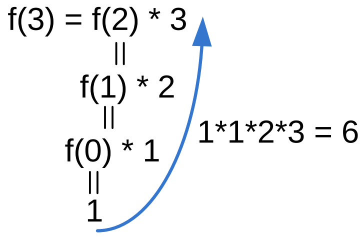
The **base case** is a small problem
- the solution to this problem should not be recursive, so that the function is guaranteed to terminate
- there can be more than one base case

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

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

If we denote 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***.



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

E-OLYMP **1658. Factorial** For the given number n find the factorial n!

► The problem can be solved with ***for*** loop, but we'll consider the recursive solution. To solve the problem, simply call a function fact(n). The value $n \le 20$, use `long long` type.

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

E-OLYMP **1603. The sum of digits** Find the sum of digits of an integer.

► Input number n can be negative. In this case we must take the absolute value of it (sum of digits for -n and n is the same).

Let ***sum***(n) be the function that returns the sum of digits of n.

- If n < 10, the sum of digits equals to the number itself: ***sum***(n) = n;
- Otherwise we add the last digit of n to ***sum***(n / 10);

We have the following recurrence relation:

$$sum(n) = \begin{cases} sum(n/10) + n\%10, n \ge 10 \\ n, n < 10 \end{cases}$$

sum(123)	=	sum(12)	+ 3	=	sum(1)	+ 2	+ 3	=	1	+ 2	+ 3	=	6

E-OLYMP **2. Digits** Find the number of digits in a nonnegative integer n.

► Let ***digits***(n) be the function that returns the number of digits of n. Note that sum of digits for n = 0 equals to 1.

- If n < 10, the number of digits equals to 1: ***digits***(n) = 1;
- Otherwise we add 1 to ***digits***(n / 10);

We have the following recurrence relation:

$$digits(n) = \begin{cases} digits(n/10) + 1, n \ge 10 \\ 1, n < 10 \end{cases}$$

Example: digits(246) = digits(24) + 1 = digits(2) + 1 + 1 = 1 + 1 + 1 = 3.

E-OLYMP 3258. Fibonacci Sequence The Fibonacci sequence is defined as follows:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_k = a_{k-1} + a_{k-2}$$

For a given value of n find the n-th element of Fibonacci sequence.

► In the problem you must find the n-th Fibonacci number. For $n \le 40$ the recursive implementation will pass time limit. The Fibonacci sequence has the following form:

i	0	1	2	3	4	5	6	7	8	9	10	...
f_i	0	1	1	2	3	5	8	13	21	34	55	...

The biggest Fibonacci number that fits into `int` type is

$$f_{46} = 1836311903$$

For $n \le 40$ its enough to use type `int`.

Let ***fib***(n) be the function that returns the n-th Fibonacci number. We have the following recurrence relation:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2), n > 1 \\ 1, n = 1 \\ 0, n = 0 \end{cases}$$

```
int fib(int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n - 2);
}
```

E-OLYMP 273. Modular exponentiation Three positive integers x, n and m are given. Find the value of x^n mod m.

► **Exponentiation** is a mathematical operation, written as x^n, involving two numbers, the base x and the exponent or power n. When n is a positive integer, exponentiation corresponds to repeated multiplication of the base: that is, x^n is the product of multiplying n bases: $x^n = x * x * \ldots * x$.

How to find x^n if x and n are given? We can use just one loop with complexity O(n). Linear time algorithm will pass the *time limit* because $n \le 10^7$.

Use `long long` type to avoid overflow.

```
scanf("%lld %lld %lld", &x, &n, &m);
res = 1;
for (i = 1; i <= n; i++)
  res = (res * x) % m;
printf("%lld\n", res);
```

E-OLYMP 4439. Exponentiation Find the value of x^n.

► How can we find x^n faster then O(n)? For example,

$$x^{10} = (x^5)^2 = (x * x^4)^2 = (x * (x^2)^2)^2$$

We can notice that $x^{2n} = (x^2)^n$, for example $x^{100} = (x^2)^{50}$.

For odd power we can use formula $x^{2n+1} = x * x^{2n}$, for example $x^{11} = x * x^{10}$.

The recurrent formula gives us the O($\log_2 n$) solution:

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

```
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);
}
```

At the iterative implementation, the case $x = 1$ and n is a large integer should be processed separately. For example, if $x = 1$ and $n = 10^{18}$, in order to calculate x^n, 10^{18} iterations should be performed and will give the ***Time Limit***.

E-OLYMP 1601. GCD of two numbers Find the GCD (greatest common divisor) of two nonnegative integers.

► 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

For example, to find GCD (1, 10^9) in the case of using *subtraction*, 10^9 operations should be performed. When using the *module* operation, one action is sufficient.

GCD of two numbers can be found using the formula:

$$\text{GCD}(a, b) = \begin{cases} a, b = 0 \\ b, a = 0 \\ \text{GCD}(a \bmod b, b), a \geq b \\ \text{GCD}(a, b \bmod a), a < b \end{cases},$$

or the same

$$\text{GCD}(a, b) = \begin{cases} a, b = 0 \\ \text{GCD}(b, a \bmod b), b \neq 0 \end{cases}$$

The loop implementation is based on the idea given in the last recurrence relation:

```
while (b > 0) :
     compute a = a % b;
     swap the variables a and b;
```

```
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);
}
```

or

```
int gcd(int a, int b)
{
  return (b) ? gcd(b,a % b) : a;
}
```

E-OLYMP **1602. LCM of two integers** Find the LCM (least common multiple) of two integers.

► The **Least Common Multiple** (LCM) of two integers a and b is the smallest positive integer that is evenly divisible by both a and b. For example, LCM(2, 3) = 6 and LCM(6, 10) = 30.

To find the least common multiple, use the formula:

$$\text{GCD}(a, b) * \text{LCM}(a, b) = a * b$$

where from

$$\text{LCM}(a, b) = a * b / \text{GCD}(a, b)$$

Since $a, b < 2 * 10^9$, then when multiplying the value $a * b$ can go beyond the type `int`. When calculating, use the type `long long`.

Consider the numbers from the sample:

$$\text{GCD}(42, 24) * \text{LCM}(42, 24) = 42 * 24,$$

where from

$$\text{LCM}(42, 24) = 42 * 24 / \text{GCD}(42, 24) = 42 * 24 / 6 = 168$$

```
long long lcm(long long a, long long b)
{
```

```
  return a / gcd(a, b) * b;
}
```

What do the next functions do (calculate):

Quiz 1

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

Quiz 2

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

Quiz 3

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

Quiz 4

```
int f(int n)
{
  if (n == 0) return 0;
  return f(n-1) + 5;
}
```

What will be printed with the next code

Quiz 5

```
#include <stdio.h>

void f(int n)
{
  if (n == 0) return;
  printf("%d ",n);
  f(n-1);
}

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

```
  f(n);
  return 0;
}
```

Quiz 6

```
#include <stdio.h>

void f(int n)
{
  if (n == 0) return;
  f(n-1);
  printf("%d ",n);
}

int main(void)
{
  int n;
  scanf("%d",&n);
  f(n);
  return 0;
}
```

Quiz 7

```
#include <stdio.h>

int f(int x, int y)
{
  if (x == 0) return y;
  return f(x-1,y) + 1;
}

int main(void)
{
  int a, b;
  scanf("%d %d",&a,&b);
  printf("%d\n",f(a,b));
  return 0;
}
```