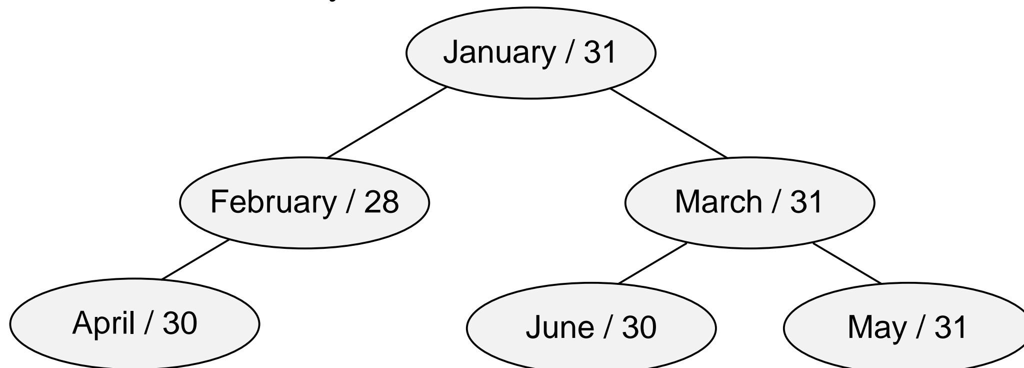
MAP

A **map** is a sorted associative container that stores a pair: a key and its corresponding value. All keys are unique.

To use the map in the program, you must include the <map> header file in it:

```
#include <map>
```

The internal structure of the map is a balanced binary tree, or rather a red – black tree that stores set of pairs. Thus, access to its elements is performed with complexity O(log *n*), where *n* is the cardinality of the set.



Declare the map and assign the number of days to the month names:

```
map<string,int> m;
```

The map can also be considered like a set of pairs (*string*, *int*).

To insert an element into the map, use one of the following methods:

```
m["July"] = 30;
```

or

```
m.insert(make_pair("July",30));
```

The ***size*** method returns the number of elements in the map:

```
number_of_elements = m.size();
```

Example. Insert the names of the months and the corresponding number of days into the map.

```
#include <cstdio>
#include <map>
#include <string>
using namespace std;

map<string,int> m;

int main(void)
{
  m["January"] = 31;  m["February"] = 28;
  m["March"] = 31;    m["April"] = 30;
  m["May"] = 31;      m["June"] = 30;

  m.insert(make_pair("July",30));
```

```
  printf("Number of months: %d\n",m.size());
  printf("February has %d days\n",m["February"]);
  return 0;
}
```

An ***iterator*** is an interface that provides access to and navigation through the elements of a collection (array or container). In the simplest case, an iterator is a pointer (a variable containing a memory address).

The map iterator *iter* is declared as follows:

```
map<string,int>::iterator iter;
```

Only two operations can be performed on iterators:

- moving forward (*iter*++), or moving to the next element of the map
- moving backward (*iter*--), or moving to the previous element.

The ***begin*** method returns the address of the first (smallest) element in the map. You can only assign addresses to iterators.

```
iter = m.begin();
```

If the iterator *iter* contains the address of a map item, then the item itself is available as **iter*. Since the **iter* object is a pair, its first element is available as (**iter*).*first* or *iter->first*, and the second as (**iter*).*second* or *iter->second*. For example, the first map element can be printed as follows:

```
iter = m.begin();
printf("%s has %d days\n",(*iter).first.c_str(),(*iter).second);
printf("%s has %d days\n",iter->first.c_str(),iter->second);
```

The first element of the map can be printed also as follows:

```
printf("%s has %d days\n",
m.begin()->first.c_str(),m.begin()->second);
```

Example. Print the months and the number of days in them. Print the months in alphabetical order.

```
#include <cstdio>
#include <map>
#include <string>
using namespace std;

map<string,int> m;
map<string,int>::iterator iter;

int main(void)
{
  m["January"] = 31;  m["February"] = 28;
  m["March"] = 31;    m["April"] = 30;
  m["May"] = 31;      m["June"] = 30;

  for(iter = m.begin(); iter !=m.end(); iter++)
    printf("%s has %d days\n",iter->first.c_str(),iter->second);
```

```
  return 0;
}
```

E-OLYMP 6940. Maximum word frequency Find the most frequent word in a document. If you get two or more results, choose one that comes later in the lexicographical order.

► Declare a data structure map<string, int> *m*, that will count how many times each word occurs. Then find the word that occurs the most times. If there are several such words, then print lexicographically the largest.

In the map *m* count how many times each word appears.

```
map<string,int> m;
```

Read and count the words.

```
scanf("%d\n",&n);
for(i = 0; i < n; i++)
{
  scanf("%s\n",s);
  m[(string)s]++;
}
```

Find the word *res* that occurs the most times *mx*.

```
mx = 0;
for(iter = m.begin(); iter != m.end(); iter++)
  if ((*iter).second >= mx)
  {
    mx = (*iter).second;
    res = (*iter).first;
  }
```

Print the word with the maximum frequency and the frequency itself.

```
printf("%s %d\n",res.c_str(),mx);
```

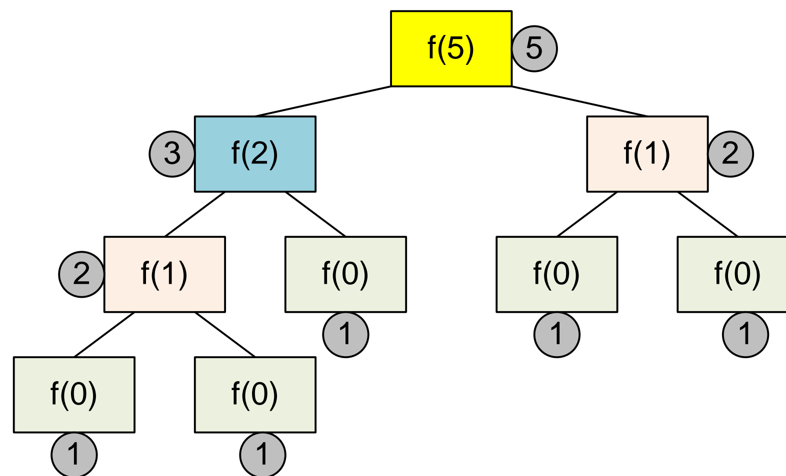
E-OLYMP 10296. Recursive function 1 Find the value of the function:

$$f(n) = \begin{cases} 1, n = 0 \\ f(n/2) + f(n/3), n > 0 \end{cases},$$

where $n \le 10^{18}$.

► Because of the limitation $n \le 10^{18}$, it is impossible to use a linear array to store the results of values of the function ***f***. For this purpose, we'll use the **map** data structure.

Consider the process of computing the function for $n = 5$:



Declare the **map**.

```
map<long long, long long> m;
```

Implement the function ***f***.

```
long long f(long long n)
{
  if (n == 0) return 1;
  if (m[n] > 0) return m[n];
  return m[n] = f(n / 2) + f(n / 3);
}
```

The main part of the program. Read the value of n. Compute and print the value of the function.

```
scanf("%lld", &n);
res = f(n);
printf("%lld\n", res);
```

E-OLYMP 7860. Judging Troubles The NWERC organisers have decided that they want to improve the automatic grading of the submissions for the contest, so they now use two systems: DOMjudge and Kattis. Each submission is judged by both systems and the grading results are compared to make sure that the systems agree. However, something went wrong in setting up the connection between the systems, and now the jury only knows all results of both systems, but not which result belongs to which submission! You are therefore asked to help them figure out how many results could have been consistent.

► In the problem one should calculate which and how many judging results were obtained by the DOMjudge and Kattis systems. If the result s was obtained by the DOMjudge system a times, and by the Kattis system b times, then the number of identical results s is equal to min(a, b).

Count in the map map<string, int> *cnt* the number of results s returned by the DOMjudge system. Further, for each result s of the Kattis system, decrease the value of cnt[s] by one (if cnt[s] > 0). Count the number of such reductions – it equals to the maximum number of results that were the same for both systems.

Count the number of results by the DOMjudge and Kattis systems in the given sample.

	DOMjudge	Kattis	min
correct	3	2	2
wronganswer	1	1	1
timelimit	1	2	1

4

4 results were identical in the systems.

Declare the map *cnt*, where cnt[*s*] stores the number of results *s* returned by the DOMjudge system.

```
map<string, int> cnt;
```

Read the number of submissions *n*.

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

Count the number of results of the DOMjudge system.

```
for (i = 0; i < n; i++)
{
  gets(s);
  cnt[s]++;
}
```

Process the results of the Kattis system.

```
res = 0;
for (i = 0; i < n; i++)
{
  gets(s);
```

If the result *s* occurred in the DOMjudge (cnt[*s*] > 0), then decrease cnt[*s*] by one and increase the counter of results *res*, which would be the same for both systems.

```
  if (cnt[s] > 0)
  {
    cnt[s]--;
    res++;
  }
}
```

Print the answer.

```
printf("%d\n", res);
```

E-OLYMP [1403. Baloo arithmetic] This happened at the time when Baloo the bear taught Mowgli the Law of the Jungle. A large and important brown bear rejoiced at the abilities of a student, because wolf cubs usually learn from the Law of the Jungle only what their Pack and tribe need. But Mowgli, like a baby cub, needed to know much more.

In class on arithmetic, Baloo came up with the next game. It was necessary to get the number n from the number 1, while allowing the current number to be either multiplied by 3, or 4 to be added to the current number. For each multiplication, Baloo gave 5 cuffs, and for each addition 2 cuffs. For example,

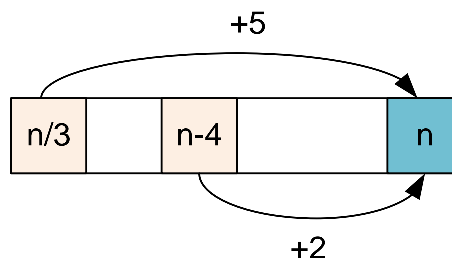
$$1 \xrightarrow{+4} 5 \xrightarrow{+4} 9 \xrightarrow{+4} 13 \xrightarrow{+4} 17 \xrightarrow{+4} 21$$

$$1 \xrightarrow{\cdot 3} 3 \xrightarrow{+4} 7 \xrightarrow{\cdot 3} 21$$

in the first case one gets 10 cuffs, in the second case 12 cuffs.

Mowgli naturally mastered arithmetic best of all and quickly figured out how to solve a problem, having received the least amount of cuffs. He also noticed that it is not always possible to complete the task of a cunning bear ...

► Let f(n) be the minimum number of cuffs that can be obtained for solving the problem.



Then:

- f(n) = min (f(n / 3) + 5, f(n – 4) + 2), if n is divisible by 3.
- f(n) = f(n – 4) + 2 , if n is not divisible by 3.

For example, for $n \leq 10^7$ compute and store the values of the function f(n) in the linear array m. For large values of n, do the memoization in the structure *map*.

Declare the structures for storing the values of the function f(n).

```
#define MAX 10000000
int m[MAX];
map<int, int> mp;
```

Function ***f*** is implemented recursively with memoization.

```
int f(int n)
{
  if (n < MAX) return m[n];
  if (mp[n] > 0) return mp[n];
  if (n % 3 == 0)
    mp[n] = f(n/3) + 5;
  else
    mp[n] = f(n-4) + 2;
  return mp[n];
```

```
}
```

The main part of the program. Let m[*i*] = - ∝, if Mowgli cannot get the value *i* by any actions. Compute the values of f(*i*) for *i* up to 10^7 and store them in m[*i*].

```
m[0] = -MAX; m[1] = 0; m[2] = -MAX; m[3] = 5; m[4] = -MAX; m[5] = 2;
for(i = 6; i < MAX; i++)
  if(i % 3 == 0)
    m[i] = min(m[i / 3] + 5, m[i - 4] + 2);
  else
    m[i] = m[i - 4] + 2;
```

Read the input value of *n*. Compute and print the answer.

```
scanf("%d",&n);
res = f(n);
```

If *res* is less than 0, then it is impossible to solve the problem, print the number 0.

```
if (res < 0) res = 0;
printf("%d\n",res);
```

E-OLYMP 9033. Greedy Aziz Do you know that Aziz loves chocolate sweets very much? However, his father does not allow him to eat a lot of chocolate because of his teeth.

Father gave him an array that contains a lot of identical numbers. During the day Aziz can eat as many candies as the maximum times the number in array is repeated.

Aziz wants to cheat to eat more candies. He can change some numbers in array, and the father will not notice it. He can increase or decrease any number in the array by 1 and only 1 time.

Aziz wants to eat the maximum number of sweets. Help him in this matter.

Find the maximum number of sweets Aziz can get.

► Count the number of times that each number occurs in the array. To do this, use the map data structure: m[*x*] will contain the number of times the number *x* occurs in the array.

For each number a[*i*] in the array, assume that it is the maximum occurring after Aziz's fraud. To do this, Aziz must increase the number (a[*i*] – 1) by 1, and decrease the number (a[*i*] + 1) by 1 (if such numbers exist in array). For example, let the array look like

5	5	6	6	7	7	7	7

The ***map*** structure contains the following data (two fives, two sixes, and four sevens):

$$m[5] = 2, m[6] = 2, m[7] = 4$$

In order for the number 6 to be the most common after fraud, it is necessary to add 1 to all numbers 5, and subtract 1 from all numbers 7:

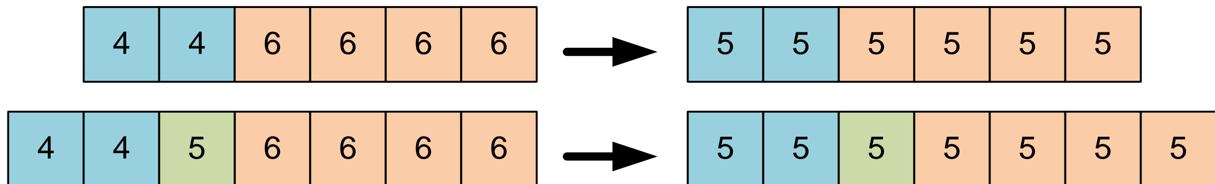
6	6	6	6	6	6	6	6

Let initially a[i] = 6. Then array initially contains m[a[i]] sixes, m[a[i] – 1] fives, and m[a[i] + 1] sevens. After cheating, the number of sixes will be

$$m[a[i] - 1] + m[a[i]] + m[a[i] + 1],$$

which will be the number of sweets that Aziz receive.

Consider the situation when the array contains two numbers a[i] = 4 and a[i] + 2 = 6 (the difference between which is 2), but the number a[i] + 1, for example, may be absent.



In this case, it would be optimal to increase all numbers a[i] by 1, and to decrease all numbers a[i] + 2 by 1. After the fraud, the amount of numbers a[i] + 1 will be

$$m[a[i]] + m[a[i] + 1] + m[a[i] + 2]$$

Declare the input array a. Declare a variable m of type ***map***.

```
map<int, int> m;
int a[100000];
```

Read the input array. Count the number of times a[i] occurs in the array a.

```
scanf("%d", &n);
for (i = 0; i < n; i++)
{
  scanf("%d", &a[i]);
  m[a[i]]++;
}
```

Compute the answer in the variable *res*.

```
int res = 0;
for (int i = 0; i < n; i++)
{
```

If the most frequently encountered number after cheating is a[i].

```
  res = max(res, m[a[i]] + m[a[i] - 1] + m[a[i] + 1]);
```

If the most frequently encountered number after cheating is a[i] + 1.

```
  res = max(res, m[a[i]] + m[a[i] + 1] + m[a[i] + 2]);
}
```

Print the answer.

```
printf("%d\n", res);
```

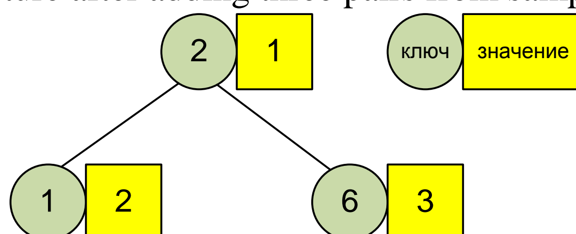
E-OLYMP 9639. Big array of Dino Once, when Dino was solving a problem related to arrays, he saw that the size of all arrays is at most 10^6. Since Dino is a dinosaur, this number seemed very small to him. Therefore, he decided to create a big array.

Dino first creates an empty array and selects n pairs of numbers: (a_1, b_1), (a_2, b_2), ..., (a_n, b_n). Then for each of these pairs he inserts into array the number b_i a_i times. For example, if the first pair is (3, 5), the number 5 will be inserted into array 3 times. After that, Dino decides to arrange this array in non-decreasing order, but since the array is very large, Dino's computer cannot perform this arrangement. He is interested in the k-th (the array is numbered starting from 1) number. Help Dino to find this number.

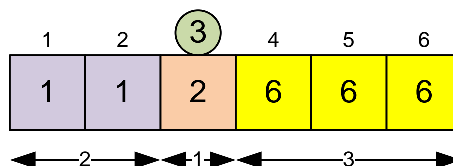
► Declare the **map** data structure. For each pair (a_i, b_i) add a_i to m[b_i]. The map keys will be the numbers of array, the associated values will be the number of times the keys occur in array.

Sum up the number of elements in array by iterating over the **map** starting from the smallest key and adding up the associated values. As soon as the sum reaches the value of k, the key (the k-th element of array) will be found.

The **map** data structure after adding three pairs from sample will look like this:



The array in the sample contains two 1, one 2, and three 6. The third number after sorting is 2.



Consider the process of finding the third element (k = 3) in the array.

1 *Iteration*. k = 3, m[1] = 2. Check: m[1] ≥ 3? No, subtract m[1] from k, k = 3 – 2 = 1.

2 *Iteration*. k = 1, m[2] = 1. Check: m[2] ≥ 1? Yes, the required element of the array is the key of the current vertex of the **map**, that is, the number 2.

Declare the **map** data structure.

```
map<int, long long> m;
```

Read the input data. For each pair (a_i, b_i) add a_i to m[b_i].

```
scanf("%d", &n);
for (i = 0; i < n; i++)
{
```

```
    scanf("%d %d", &a, &b);
    m[b] += a;
  }
```

Iterate over the **map** structure in ascending order of keys. Subtract associated values from k. As soon as the associated value for the next key is at least k, then the corresponding key is the k-th element of array.

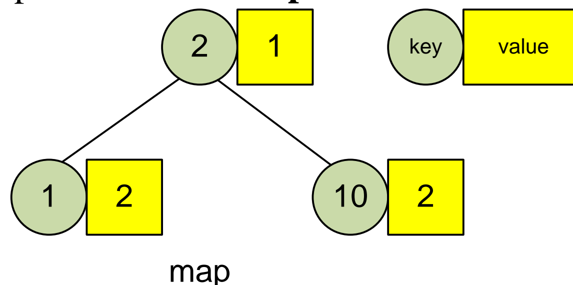
```
scanf("%lld", &k);
for (iter = m.begin(); iter != m.end(); iter++)
{
  if ((*iter).second >= k) break;
  k -= (*iter).second;
}
```

Print the answer.

```
printf("%d\n", (*iter).first);
```

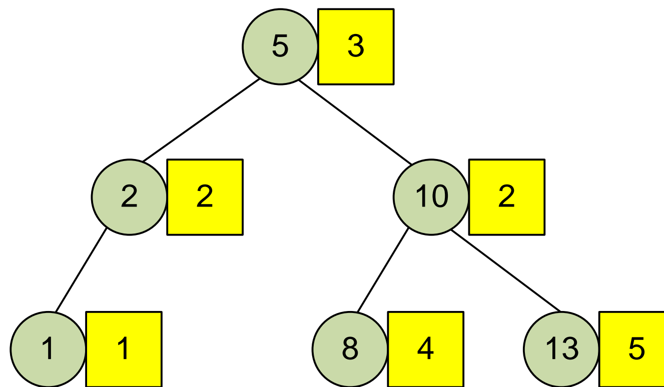
E-OLYMP [10030. SpaceX](#) Elon Musk plans to send his spaceships to k different planets. To do this, he has n spaceships. Initially, it is known where each ship will be sent. The planets are numbered from 1 to 10^9 . As SpaceX's chief space engineer, you are entitled to change the destination of any ship. For the minimum number of changes you need to make sure that all ships are sent to k different planets.

► For each destination p in the map m, count the number of ships m[p] sent there. For example, in the second test, two ships are sent to planet 1, one ship is sent to planet 2, and two ships are sent to planet 10. The **map** structure looks like this:

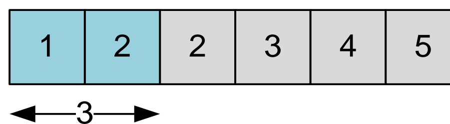


The size of the map is m.size () = 3, all ships are sent to 3 planets. The ships should be sent to exactly k = 4 different planets. If m.size () < k, then k – m.size() ships should change the course.

Consider the case where spaceships are sent to more than k planets. Let k = 4, but the **map** structure will be as follows:



17 ships were sent to 6 planets. From the two planets, ships should be redirected to any of the 4 remaining ones. Since you want to minimize the number of changes, you should find the two planets to which the least number of ships are sent. Sort the numbers – the number of ships sent to the planets. And find the sum of the two smallest ones.



Declare the data structures.

```
map<long long, long long> m;
vector<long long> v;
```

Read the input data.

```
scanf("%d %d", &n, &k);
for (i = 0; i < n; i++)
{
  scanf("%d", &x);
```

In m[x] count the number of ships sent to planet x.

```
  m[x]++;
}
```

If all spaceships are sent to less than k planets, then the minimum number of changes is $k – m$.size().

```
if (m.size() < k)
{
  printf("%d\n", k - m.size());
  return 0;
}
```

Spaceships are sent to more than k planets. Construct a vector v that contains the number of ships sent to different planets.

```
for (iter = m.begin(); iter != m.end(); iter++)
  v.push_back((*iter).second);
```

Sort the vector *v*.

```
sort(v.begin(), v.end());
```

Look for the sum of *m*.size() – *k* smallest numbers (from this number of planets the ships destinations should be changed).

```
sum = 0;
for (i = 0; i < m.size() - k; i++)
  sum += v[i];
```

Print the answer.

```
printf("%lld\n", sum);
```