# Сортиране

Функцията **sort**, дефинирана в библиотеката **<algorithm>**, е удобна за използване и по същество представлява имплементация на алгоритъма **quick sort**. Параметрите, които подаваме, са указател към началото на редицата, която сортираме, и указател към елемента след края на редицата.

Знаем, че в средния случай, този алгоритъм има сложност **Θ(*N**log *N*)**, а в най-лошия - **Θ($N^2$)**. Знаем също така, че най-лошият случай се получавам, когато елементите, които сортираме, са подредени първоначално в обратен ред. Въпреки това ние можем да очакваме *sort* да работи със сложност **Θ(*N**log *N*)** във всеки един случай, дори когато подаваме елементи, които са подредени в обратен ред.

```
int a[5] = { 4, 2, 4, 1, 5 };
sort(a, a + 5);
for(int i = 0;i < 5;i++)
	printf("%d ", a[i]); // 1 2 4 4 5
printf("\n");
```

Функцията **sort** нормално използва оператора **<**, за да определи дали един елемент е по-малък от друг, в процеса на сортирането. Вместо да използваме естествената подредба на елементите, които сравняваме, можем и **да подадем като трети параметър на *sort* указател към функция**, която имплементира някаква наша логика за сравнение. Контрактът на тази функция е да връща резултат *true*, ако първият й аргумент е по-малък от втория.

```
#include <stdio.h>
#include <algorithm>
using namespace std;

bool cmp(const int &a, const int &b) { return a > b; }

int main()
{
	sort(a, a + 5, cmp);
	for(int i = 0;i < 5;i++)
		printf("%d ", a[i]); // 1 2 4 4 5
	printf("\n");

	return 0;
}
```

Важно е да се отбележи, че в **pair** са предефинирани операторите **<**, ==, **>**, както и операторът за присвояване **=**. Сравнението се извършва първо по полето **first** и ако стойностите в него са равни - по полето **second**. Това е демонстрирано в следващия пример, където първоначално сортираме вектора без да му подаваме сравняваща функция. За следващото сортиране използваме версията на функцията sort, която като трети параметър приема указател към функция, която имплементира наша специфична логика за сравнение. В нашия случай това е функцията ***cmpVector***, в която сравняваме **pair** обектите **единствено по второто им поле**:

```
#include <stdio.h>
#include <algorithm>
#include <vector>
```

```
using namespace std;

bool cmpVector(const pair<int, double> a, const pair<int, double> b)
{
	return a.second != b.second ?
		a.second - b.second < 0:
		a.first - b.first < 0;
}

vector< pair <int, double > > v;

void printVector()
{
	vector< pair <int, double > > :: iterator it;
	for(it = v.begin(); it != v.end(); it++)
		printf("%d %.7f\n", it->first, it->second);
	printf("\n");
}

int main()
{
	v.push_back(make_pair(3, 0.000001));
	v.push_back(make_pair(2, 3.1415));
	v.push_back(make_pair(1, 2.1));
	v.push_back(make_pair(1, 1.2));
	v.push_back(make_pair(2, 4.9999999));
	v.push_back(make_pair(2, 5));

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

	//1 1.2000000
	//1 2.1000000
	//2 3.1415000
	//2 4.9999999
	//2 5.0000000
	//3 0.0000010

	sort(v.begin(), v.end(), cmpVector);
	printVector();

	//3 0.0000010
	//1 1.2000000
	//1 2.1000000
	//2 3.1415000
	//2 4.9999999
	//2 5.0000000

	return 0;
}
```

# Търсене в сортирани последователности

## Шаблони за двоично търсене

### Двоично търсене с обновяване на отговора за дискретни функции

```
int binarySearch(int x)
{
    int left = 0, right = 100000, ans = -1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (can(x))
            left = mid + 1;
        else
            right = mid - 1, ans = mid;
    }

    return ans;
}
```

### Непрекъснат интервал (сравнение на границите)

```
#define EPS 1e-9

bool can(double f)
{
	bool ok = false;

	//simulate and set ok to true or false
	//....

	return ok;
}

void solve()
{
	double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
	while(fabs(hi - lo) > EPS)
	{
		mid = (hi + lo) / 2.0;
		if(can(mid))
		{
			ans = mid;
			hi = mid;
		}
		else
			lo = mid;
	}
}
```

## Непрекъснат интервал (предварително зададен брой стъпки)

```
double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
for(int i = 0;i < 100;i++)
{
	mid = (hi + lo) / 2.0;
	if(can(mid))
	{
		ans = mid;
		hi = mid;
	}
	else
		lo = mid;
}
```

## Двоично търсене с STL

Следните функции **работят коректно само върху сортирани последователности** и имат логаритмична сложност.

- ***lower_bound*** - намира първия елемент в дадена сортирана последователност, който има стойност **по-малка или равна** от зададената.
- ***upper_bound*** - намира първия елемент в дадена сортирана последователност, който има стойност **по-голяма** от зададената.
- ***binary_search*** **-** двоично търсене, което проверява дали елемента е в сортираната последователност. Връща `true`, ако го има и `false`, ако го няма.

```
#include <algorithm>
#include <vector>
using namespace std;

bool mygreater (int i,int j) { return (i > j); }

int main()
{
	int a[] = {10, 20, 30, 30, 20, 10, 10, 20};
	vector<int> v(a, a + 8);          // 10 20 30 30 20 10 10 20
	vector<int>::iterator low, up;

	sort (v.begin(), v.end());       // 10 10 10 20 20 20 30 30

	//Returns an iterator pointing to the first element in the sorted range
	//[first,last) which does not compare less than value
	low = lower_bound (v.begin(), v.end(), 20);

	//Returns an iterator pointing to the first element in the sorted range
	//[first,last) which compares greater than value
	up = upper_bound (v.begin(), v.end(), 20);
```

```
        // 10 10 10 20 20 20 30 30
        //          ^         ^
        //               low         up
        printf("lower_bound at position %d\n", low - v.begin()); // 3
        printf("upper_bound at position %d\n", up - v.begin());  // 6

        //Returns true if an element in the range [first,last) is equivalent
        //to value, and false otherwise.
        bool found = binary_search(v.begin(), v.end(), 33); //false

        pair<vector<int>::iterator, vector<int>::iterator> bounds;
        bounds = equal_range (v.begin(), v.end(), 20);
        // 10 10 10 20 20 20 30 30
        //          ^        ^
        printf("bounds at positions %d and %d\n",
               bounds.first - v.begin() ,
               bounds.second - v.begin());

        // using "mygreater" as comp:
        sort (v.begin(), v.end(), mygreater);
        bounds = equal_range (v.begin(), v.end(), 20, mygreater);
        // 30 30 20 20 20 10 10 10
        //       ^        ^
        printf("bounds at positions %d and %d\n",
               bounds.first - v.begin() ,
               bounds.second - v.begin());

        return 0;
}
```

## Полезни ресурси

Сортиране

Двоично търсене

### Задачи за упражнение:

- Задачите дадени в края на двете теми по-горе.
- Spoj0
- A2 Online Judge
- Algoogle