Quicksort

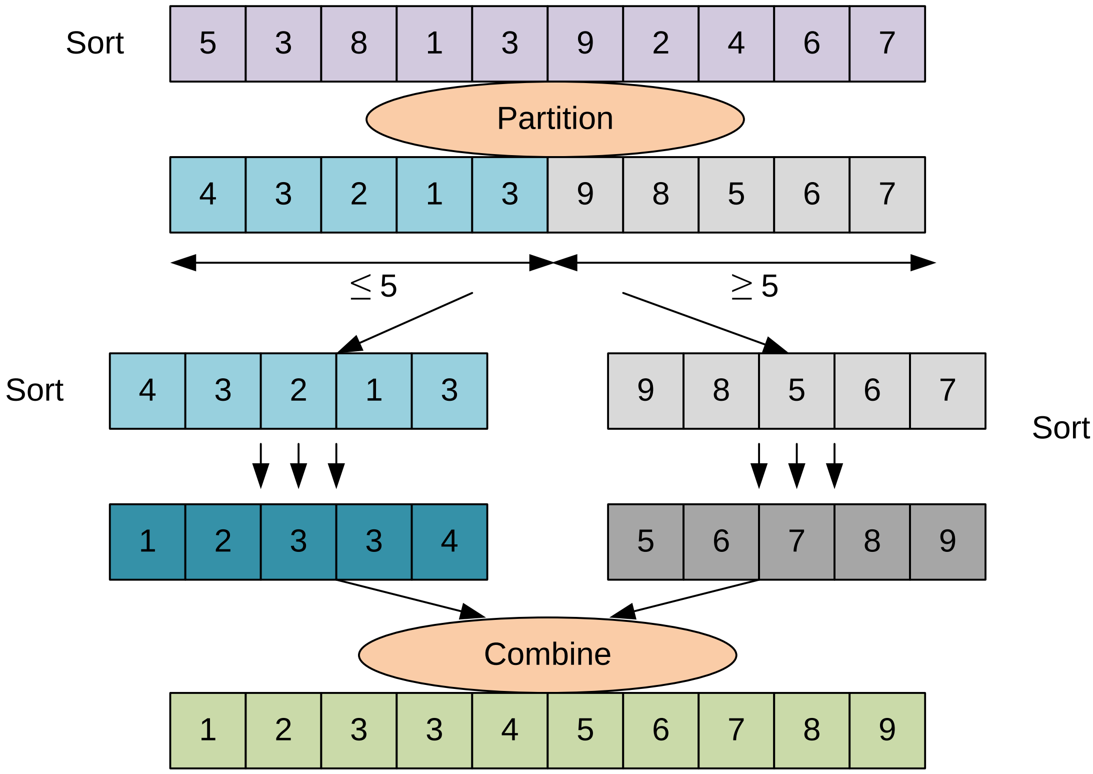
Quicksort is a ***sorting algorithm*** whose worst-case running time is O(n^2) on an input array of n numbers. In spite of this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is O(n log$n$), and the constant factors hidden in the O(n log$n$) notation are quite small.

Quicksort is based on the ***divide-and-conquer*** paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray a[l . . r]:

Divide: Partition (rearrange) the array a[l . . r] into two (possibly empty) subarrays a[l . . q] and a[q +1 . . r] such that each element of a[l . . q] is less than or equal to a[q], which is, in turn, less than or equal to each element of a[q + 1 . . r]. Compute the index q as part of this ***partitioning*** procedure.

Conquer: Sort the two subarrays A[l . . q] and A[q +1 . . r] by recursive calls to quicksort.

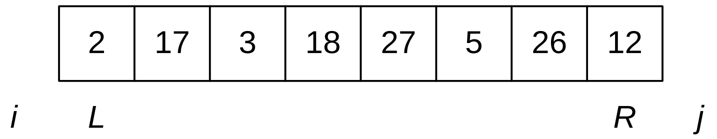
Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array a[l . . r] is now sorted.



In the worst case, the running time of the algorithm is O(n^2), although in practice its average running time is O(n *log* n).

One of the critical operations in quicksort is the selection of a ***pivot*** (the element around which the array is partitioned). The simplest algorithm for choosing a pivot is to take the first or last element of array, but in this case we can get a bad behavior on almost sorted data. Niklaus Wirth suggested to use a **middle element** to prevent this case from degrading to O(n^2) on bad inputs. The "***median of three***" selects the **median** of the first, middle and last array elements as a pivot. However, even though it works well on most inputs, it is still possible to find inputs that slow down this sorting algorithm a lot.

Here is an implementation where the array partitioning algorithm m[L .. R] was developed by ***Hoare***. x = m[L] is chosen as pivot. The idea is to accumulate elements, not greater than x, in the initial segment of the array m [L .. i], and elements, not less than x, at the end of m [j .. R]. At the beginning, both segments are empty: i = L – 1, j = R + 1.



initial state

current state

Partitioning an array is done by repeating the following steps:

Step 1. Increase i by one. Move the pointer i to the right until encountered a number that is not less than x.

Step 2. Decrease j by one. Move the pointer j to the left until encountered a number that is not greater than x.

Step 3. If in this case $i < j$ holds, then we swap the values m[i] and m[j] and go to step 1. Otherwise, the splitting algorithm ends and the array is considered divided into m[L ... j] and m[j + 1 ... R].

Upon completion of the ***Partition*** procedure, each element of subarray m[L ... j] does not exceed the values of each element of the subarray m[j + 1 ... R]. The running time of the procedure is O(n), where n = R – L + 1.

E-OLYMP **2321. Sort** Sort array of integers in nondecreasing order.

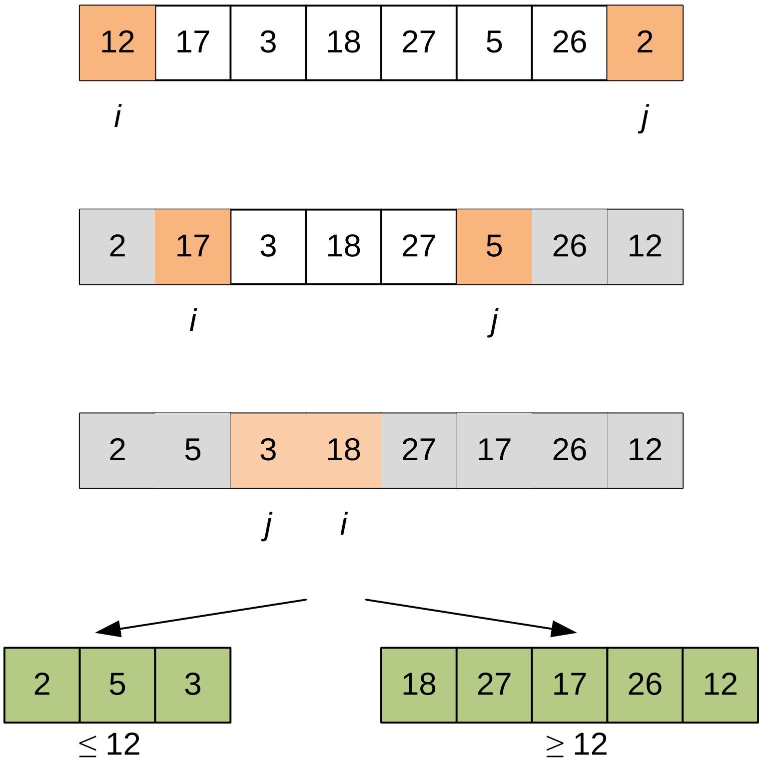
► Use **quicksort** to sort an array.

```
#include <stdio.h>

int m[1001];
int i, n;

void swap(int &i, int &j)
{
  int temp = i; i = j; j = temp;
}

int Partition(int L, int R)
{
  int x = m[L], i = L - 1, j = R + 1;
  while (1)
  {
    do j--; while (m[j] > x);
    do i++; while (m[i] < x);
    if (i < j) swap(m[i], m[j]); else return j;
  }
}

void QuickSort(int L, int R)
{
  if (L < R)
  {
    int q = Partition(L, R);
    QuickSort(L, q); QuickSort(q + 1, R);
  }
}

int main(void)
{
  scanf("%d", &n);
  for (i = 0; i < n; i++) scanf("%d", &m[i]);

  QuickSort(0, n - 1);

  for (i = 0; i < n; i++) printf("%d ", m[i]);
  printf("\n");
  return 0;
}
```

Example. Let's do the Hoare *partition* of the next array. Pivot $x = 12$.

12	17	3	18	27	5	26	2

i ⟶ ⟵ j



E-OLYMP [**972. Sorting time**]() Sort the time according to specified criteria.
► Use **QuickeSort** to sort the time structures.

Declare structure **MyTime**.

```
struct MyTime
{
  int hour, min, sec;
  MyTime() {};
  MyTime(MyTime &a) : hour(a.hour), min(a.min), sec(a.sec) {};
};
```

Declare the comparator.

```
int f(MyTime a, MyTime b)
{
  if ((a.hour == b.hour) && (a.min == b.min)) return a.sec < b.sec;
  if (a.hour == b.hour) return a.min < b.min;
  return a.hour < b.hour;
}
```

Read the input data into array of **MyTime** sturctures.

```
#define MAX 1001
MyTime lst[MAX];
```

Call **QuickeSort** to sort the data.

```
QuickSort(lst, 1, n);
```

E-OLYMP [**1953. The results of the olympiad**]() *n* Olympiad participants have unique numbers from 1 to *n*. As a result of solving problems at the Olympiad, each participant received a score (an integer from 0 to 600). It is known how many points everybody scored.

Print the list of participants in Olympiad in decreasing order of their accumulated points.

► Use **QuickSort** to sort the *Member* (participant) structures. Each participant has his own *id* and *score*.

```
struct Member
{
  int id, score;
  Member(int id = 0, int score = 0) : id(id), score(score) {};
};
```

E-OLYMP [**8637. Sort the points**]() The coordinattes of *n* points are given on a plane. Print them in increasing order of sum of coordinates. In the case of equal sum of point coordinates sort the points in increasing order of abscissa.

► Use **QuickSort** to solve the problem.

E-OLYMP [**8236. Sort evens and odds**]() Sequence of integers is given. Sort the given sequence so that first the odd numbers are arranged in ascending order, and then the even numbers are arranged in descending order.

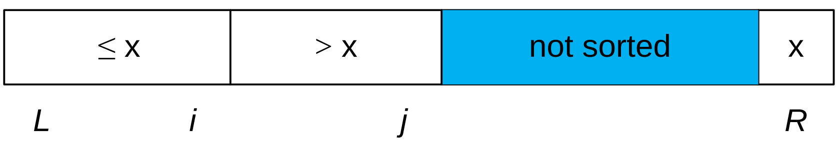
► Use **QuickSort** to solve the problem according to the following comparator f(***int*** *a*, ***int*** *b*):

- if *a* and *b* have different parity, then even numbers must come after odd numbers;
- if *a* and *b* are even, then sort them in in decreasing order;
- if *a* and *b* are odd, then sort them in in increasing order;

Note that the input numbers can be positive and negative.

Consider ***another*** algorithm for partitioning an array m[L .. R]. Let us choose x = m[R] as the **pivot** element. During operation, the algorithm of partitioning the array is divided into 4 parts:

- elements not larger than x;
- elements larger than x;
- unsorted part;
- the last element is a **pivot**;

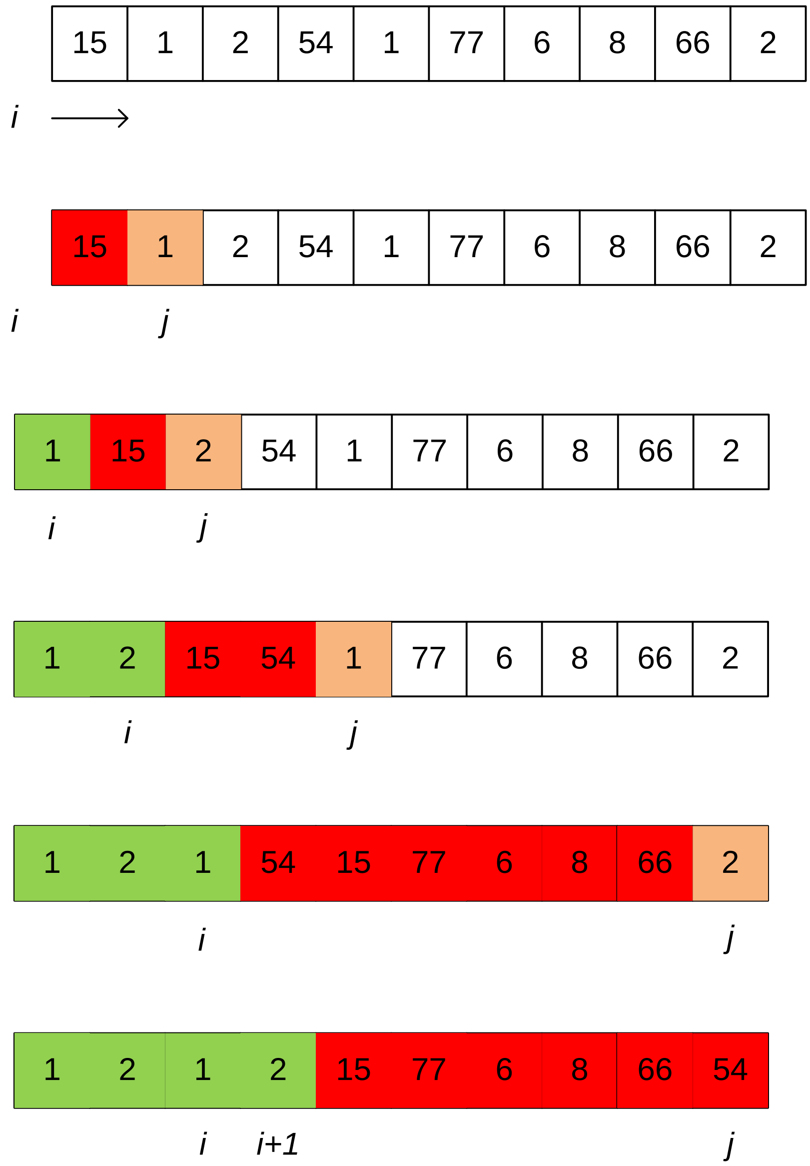


Initially set i = L – 1. Move the pointer j from L to R – 1. As soon as found an element m[j] that is not greater than x, increase i by 1 and swap m[i] and m[j]. The pivot *x* during the *j* loop remains in its place. At the end of the loop, swap m[*i* + 1] and *x*. The array will then be split into two halves by the pivot *x*.

E-OLYMP [2321. Sort](#) Sort array of integers in nondecreasing order.
► Use **quicksort** to sort an array.

```
#include <stdio.h>

int m[1001];
int i, n;

void swap(int &i, int &j)
{
  int temp = i; i = j; j = temp;
}

int Partition(int L, int R)
{
  int x = m[R], i = L - 1, j;
  for (j = L; j < R; j++)
    if (m[j] <= x)
    {
      i++;
      swap(m[i], m[j]);
    }
  swap(m[i + 1], m[R]);
  return i + 1;
}

void QuickSort(int L, int R)
{
  if (L < R)
  {
    int q = Partition(L, R);
    QuickSort(L, q - 1);
    QuickSort(q + 1, R);
  }
}

int main(void)
{
  scanf("%d", &n);
  for (i = 0; i < n; i++) scanf("%d", &m[i]);

  QuickSort(0, n - 1);

  for (i = 0; i < n; i++) printf("%d ", m[i]);
  printf("\n");
  return 0;
}
```

Example. Let's make a partition of the next array. The pivot element *x* = 2. Mark the element m[*j*] in brown, that should be swapped with m[*i* + 1]. Highlighted in green the set of already processed elements, not greater than *x*, highlighted in red the elememts larger than *x*.



Time complexity of the quicksort algorithm depends on how the array is partitioned at each step. If the partitioning occurs into approximately equal parts, then the running time is O($n\log_2 n$). If the sizes of the parts are very different, sorting process can take O(n^2) time.

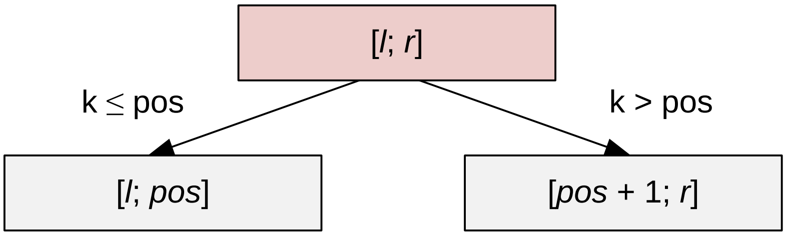
Introspective sort

Introsort, or ***introspective sort***, is a sorting algorithm proposed by David Musser in 1997. It uses **quicksort** and switches to **heapsort** when the recursion depth exceeds some predetermined level (for example, the logarithm of the number of items being sorted). This approach combines the advantages of both methods with O(n log n) worst-case performance and performance comparable to quicksort.

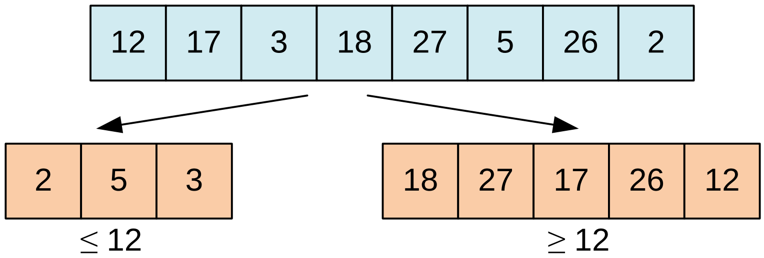
Finding the *k*-th order statistic

***k*-th order statistic** is the *k*-th smallest / largest element in array. Let us show how to compute it in linear time.

Using the procedure ***Partition***, divide the array m[*l* .. *r*] in two halves m[*l* .. *pos*] and m[*pos* + 1 .. *r*]. If *l* = *r*, then the *k*-th element is in m[*l*]. If $k \le pos$, the *k*-th element is in m[l .. *pos*]. Otherwise it should be looked for in m[*pos* + 1 .. *r*].



Example. Let we want to find *k*-th smallest element is array m = {12, 17, 3, 18, 27, 5, 26, 2}. Run *partition* and divide an array in two parts:

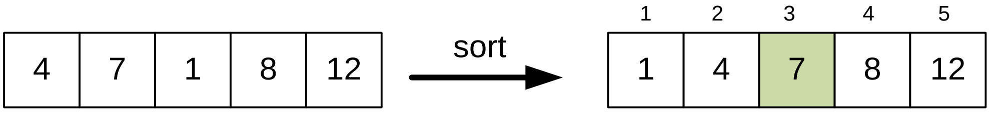


Left part m[1 .. 3] contains 3 elements, right part m[4 .. 8] contains 6 elements.

- If $k \le 3$, continue search in the left part;
- If $k \ge 4$, continue search in the right part;

E-OLYMP [9025. k-th element](#) Array *a* of *n* integers and number *k* are given. Find the *k*-th element in a sorted array a (indexing starts from 1).

► To solve the problem in O($n\log_2 n$), it is enough to ***sort*** the array and print its *k*-th element.



We can use the ***nth_element*** function, which in O(*n*) permutes the elements of the array in such a way that the *k*-th element will be in the *k*-th place, the numbers to the left of it are no more than a[*k*], and the numbers to the right of it are at least a[*k*].

The *k*-th statistic can be found in linear time using the ***partition*** function, which is used in *quicksort* algorithm. The partition function in linear time splits (does not sort) the array a[1..*n*] into two parts a[1..*pos*] and a[*pos* + 1..*n*] so that all elements of the array from the first part are no more than elements from the second part. If $k \le pos$, then we look for the *k*-th statistics in a[1..*pos*], otherwise we look for it in a[*pos* + 1..*n*].

```
#include <cstdio>
```

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

vector<int> v;
int n, k, i;

int Partition(int left, int right)
{
  int x = v[left], i = left - 1, j = right + 1;
  while (1)
  {
    do j--; while (v[j] > x);
    do i++; while (v[i] < x);
    if (i < j) swap(v[i], v[j]); else return j;
  }
}

int kth(int k, int left, int right)
{
  if (left == right) return v[left];
  int pos = Partition(left, right);
  if (k <= pos) return kth(k, left, pos);
  else return kth(k, pos + 1, right);
}

int main(void)
{
  scanf("%d %d", &n, &k);
  v.resize(n + 1);
  for (i = 1; i <= n; i++)
    scanf("%d", &v[i]);

  printf("%d\n", kth(k, 1, n));
  return 0;
}
```

E-OLYMP **5201. k-th minimum** Find the *k*-th number in array A = < a_1, a_2, ..., a_n > sorted in increasing order.

Array A is generated with the polynom P(*x*) = $132x^3 + 77x^2 + 1345x + 1577$: a_i = P(*i*) mod 1743.

► Generate array A. Use ***partition*** to solve the problem in O(*n*).

E-OLYMP **5721. Find an element** Array of *n* integers is given. Find its *k*-th element in decreasing order.

► Use ***partition*** to solve the problem in O(*n*).