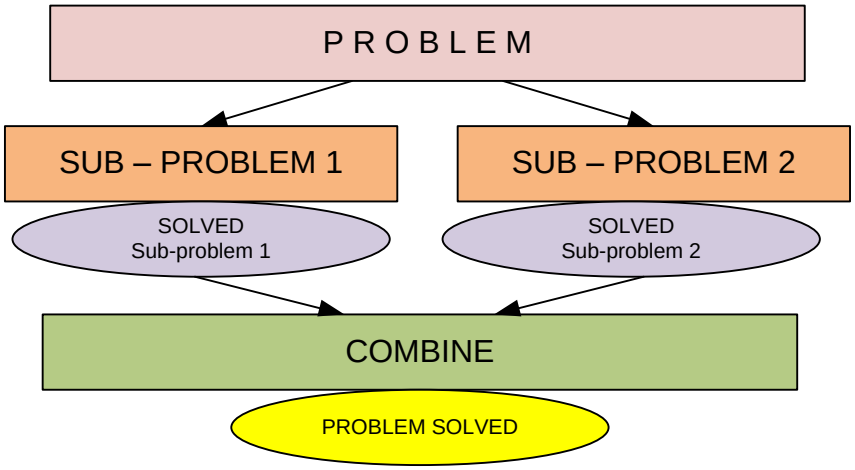
Divide and Conquer

Divide-and-conquer is an *algorithm design paradigm* based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into ***two or more*** sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

- **Divide**: divide the problem into some sub problems;
- **Conquer**: Call each sub-problem recursively until it will be solved.
- **Combine**: The sub-problems are solved so we can get the solution of entire problem.




```
divide&conquer(input I)
begin
  if (size of input is small enough) then
    solve directly;
    return;
  endif

  divide I into two or more parts I1, I2,...;

  call divide&conquer(I1) to get a subsolution S1;
  call divide&conquer(I2) to get a subsolution S2;
  ...

  Merge the subsolutions S1, S2,...into a global solution S;
end
```

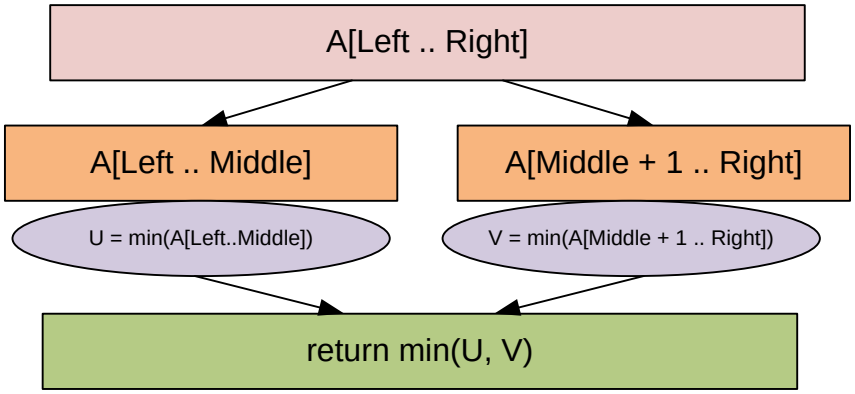
Minimum and maximum

E-OLYMP **5328. Find minmum** Array *a* of *n* integers is given. Find minimum in array using divide-and-conquer technique.

► Let FindMin(*left*, *right*) be a function that finds minimum on subarray a[*left* .. *right*]. To find minimum, divide it into two parts: a[*left* .. *mid*] and a[*mid* + 1.. *right*]

(here *mid* = (*left* + *right*) / 2, integer division). Find recursively minimum in these parts. Return minimum between them.

To find minimum in array a[1 .. *n*] we must call FindMin(1, *n*).




```
#include <stdio.h>

int i, n;
int m[1001];

int FindMin(int left, int right)
{
  if (left == right) return m[left];
  int middle = (left + right) / 2;
  int u = FindMin(left, middle);
  int v = FindMin(middle + 1, right);
  return (u < v) ? u : v;
}

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

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

E-OLYMP 928. The sum of the largest and the smallest Array of *n* integers is given. Find the sum of minimum and maximum in array using divide-and-conquer technique.

► Implement the functions FindMin(*left*, *right*) and FindMax(*left*, *right*).

To solve the problem, print the value of FindMin(1, *n*) + FindMax(1, *n*).

Merge operation

E-OLYMP 9593. Merge the sequences Two sequences are given. Merge them.

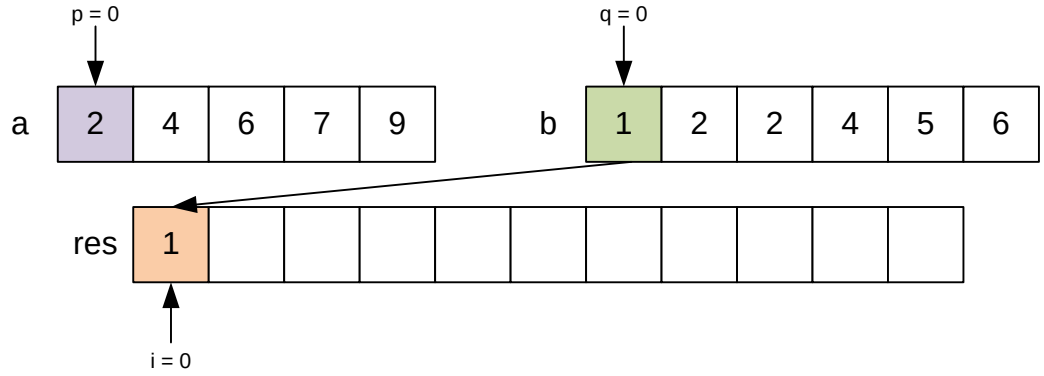
► We can read two sequences into one array and ***sort*** it. It takes $O(n\log_2 n)$ time, where *n* is the sum of sizes of two sequences.

If a and b are input arrays, res is resulting array, we can use STL function ***merge***, it takes O(*n*) time.

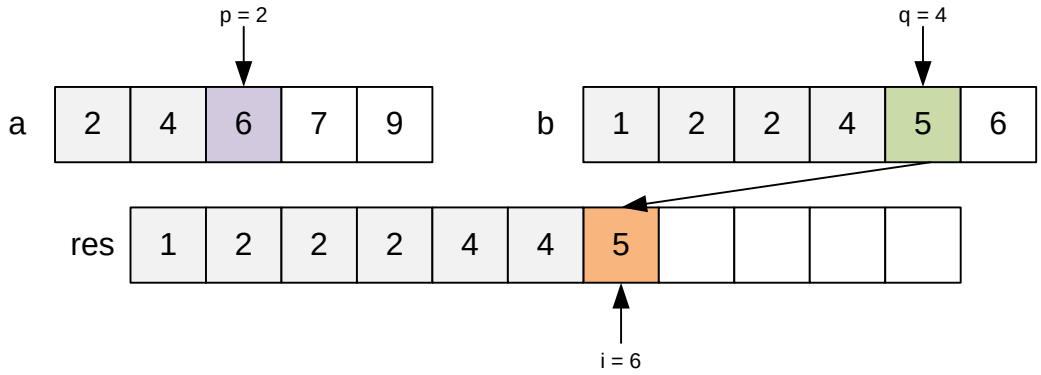
```
merge(a.begin(), a.end(), b.begin(), b.end(), res.begin());
```

But let's consider the process of merging two sequences a and b more thoroughly. Declare three variables – pointers *p*, *q*, *i* and set up them:

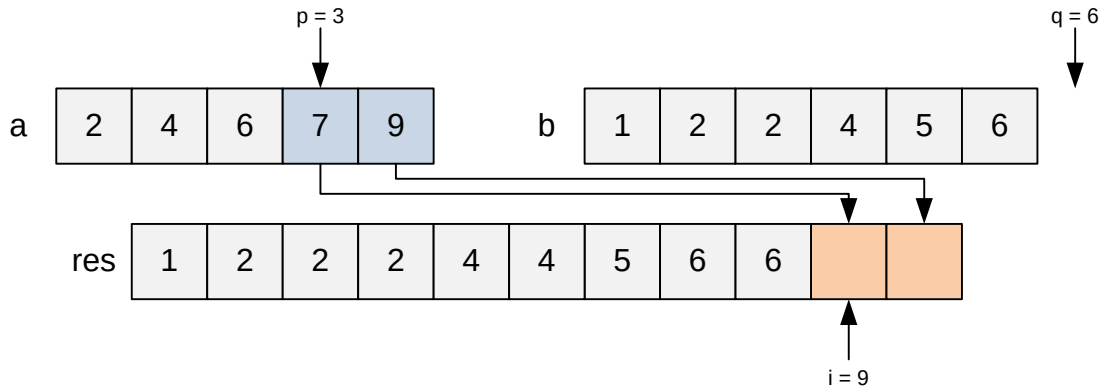
- *p* = 0, points to the start of the first array a;
- *q* = 0, points to the start of the second array b;
- *i* = 0, points to the start of the resulting array res;



At each step (iteration) compare the values a[*p*] and b[*q*]. The least of these values assign to res[*i*], and increase by 1 the pointer *i* and pointer to the minimum element (*p* or *q*). For example, after some steps we can get next state of arrays:



When the pointer in one of arrays comes to the end, the rest of the second array should be copied to the resulting array.



Decalre the working arrays.

```
vector<int> a, b, res;
```

Read two sorted sequences.

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

Set the size of the resulting sequence – it is equal to $n + m$.

```
res.resize(n + m);
```

Initialize the pointers.

```
p = q = 0;
```

While none of pointers has reached the end of array, assign res[i] = min(a[p], b[q]) and move the corresponding pointers forward by one.

```
for (i = 0; p < n && q < m; i++)
{
  if (a[p] <= b[q]) res[i] = a[p], p++;
  else res[i] = b[q], q++;
}
```

One of the pointers reached the end of array. Copy the remainder of the second array into the resulting one. If at the moment $p = n$, then only the second ***while*** will run. If $q = m$ at the moment, then only the first ***while*** will run.

```
while (p < n) res[i++] = a[p++];
while (q < m) res[i++] = b[q++];
```

Print the resulting sequence.

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

E-OLYMP **9605. Merge three sequences** Three sequences are given. Merge them.

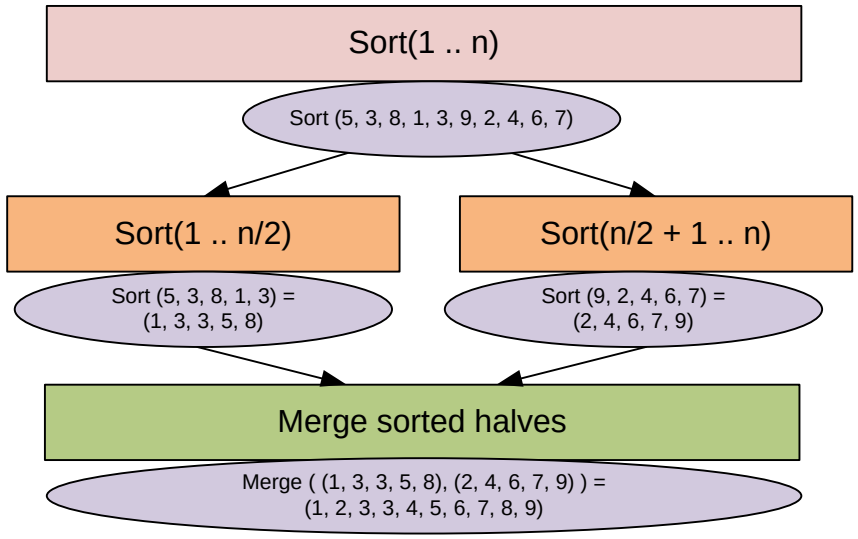
► Merge three sequences in O($n + m + k$) time, where n, m, k are the sizes of input sequences.

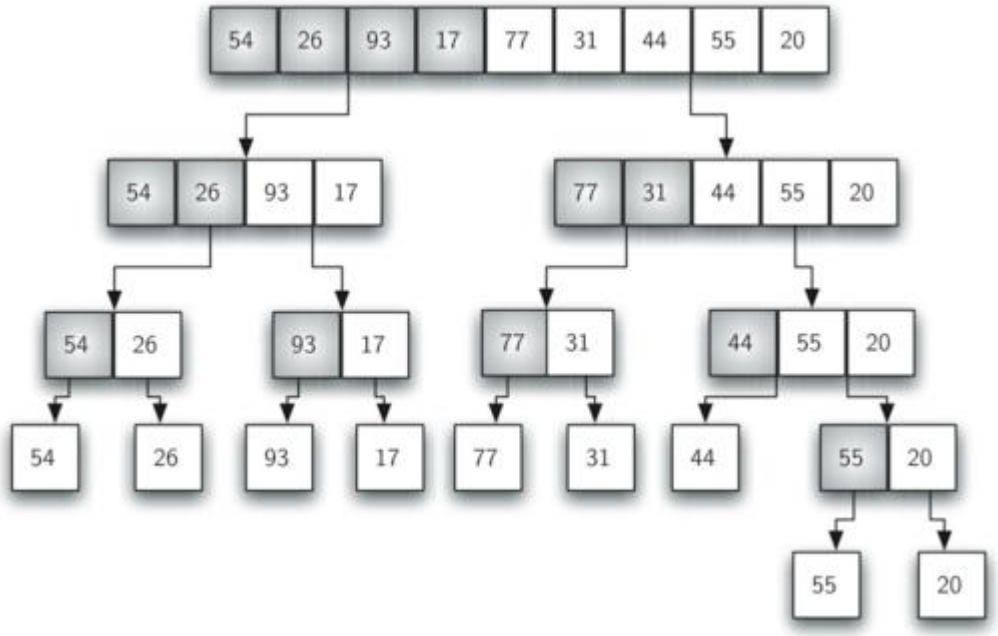
Merge sort

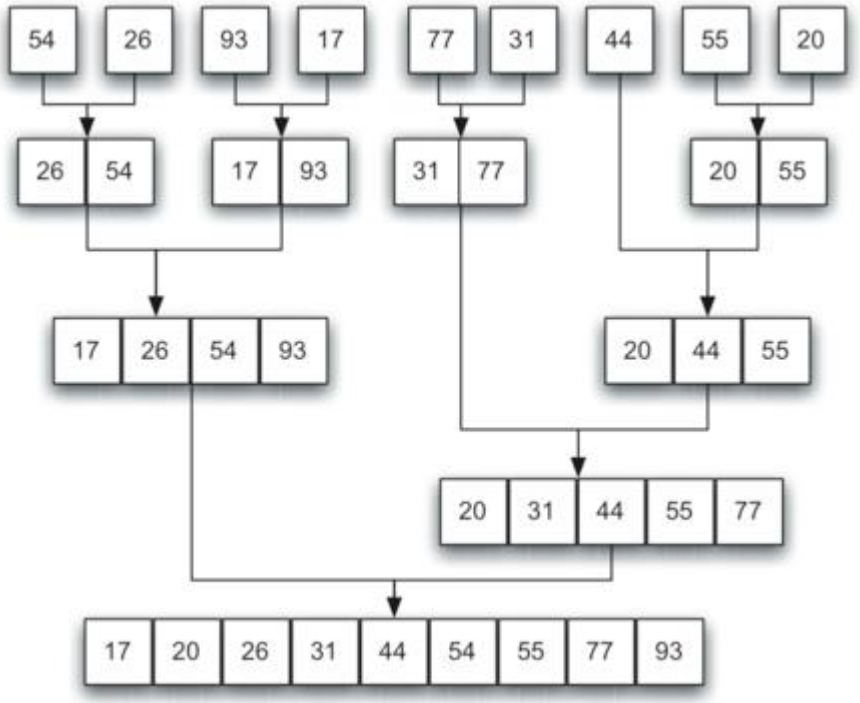
Merge sort divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

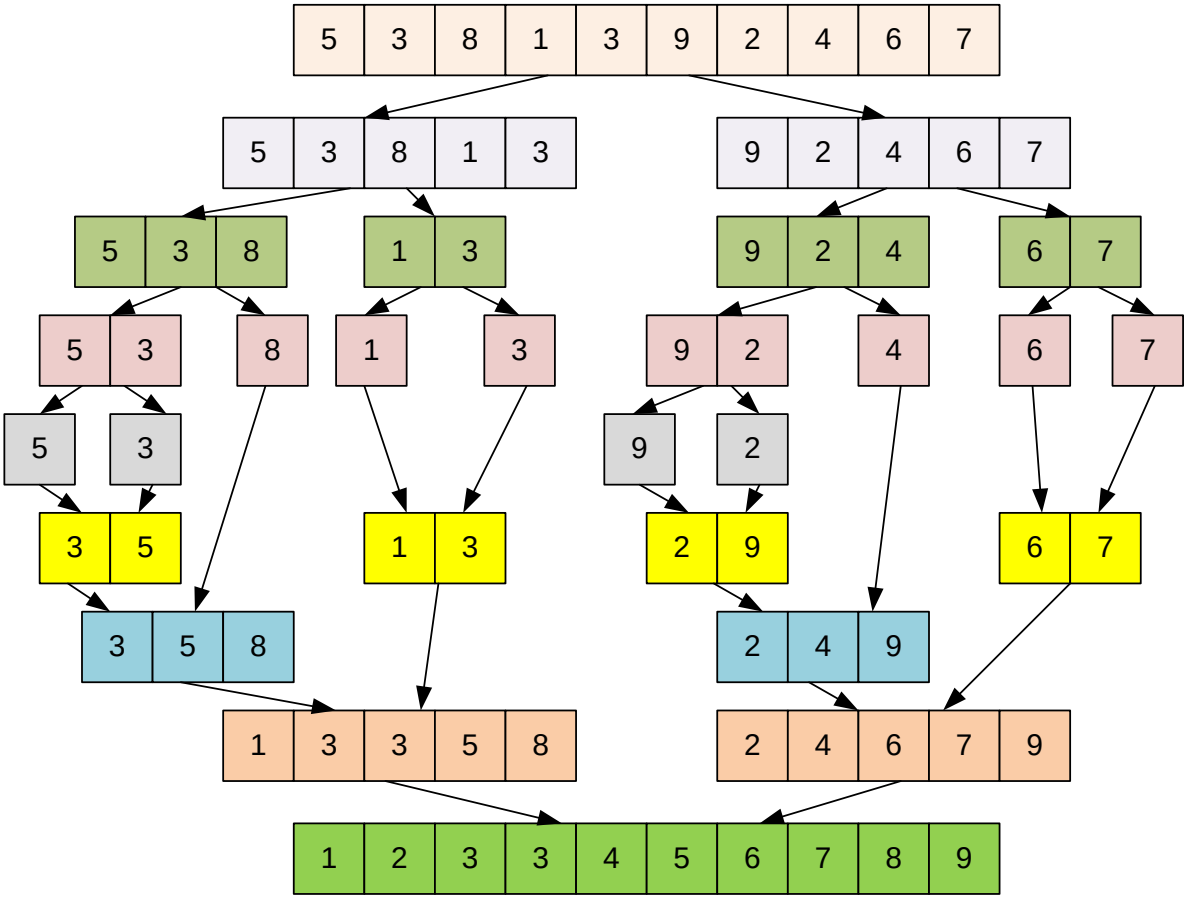
```
MergeSort(arr a[], left, right)
begin
```

```
  if (left < right) then
  begin
    Find the middle point to divide the array into two halves:
      middle = (left + right) / 2;

    Call MergeSort for the first half:
      MergeSort(arr a[], left, middle);

    Call MergeSort for the second half:
      MergeSort(arr a[], middle, right);

    Merge the two sorted halves
      Merge(arr a[], left, middle, right);
  end
end
```






54 26 93 17 77 31 44 55 20
26 54 17 93 31 77 20 55
17 26 54 93 20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93


5 3 8 1 3 9 2 4 6 7
5 3 8 1 3 9 2 4 6 7
5 3 8 1 3 9 2 4 6 7
5 3 8 1 3 9 2 4 6 7
5 3 8 1 3 9 2 4 6 7
5 3 1 3 2 9 6 7
3 5 8 2 4 9
1 3 3 5 8 2 4 6 7 9
1 2 3 3 4 5 6 7 8 9

E-OLYMP 2321. Sort Sort array of integers in nondecreasing order.
► Use ***merge sort*** for sorting.

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

void merge(int *a, int bleft, int bright, int cleft, int cright)
{
  // a[bleft..bright] and a[cleft..cright]
  // are merged into a[bleft..cright]
  int i, left = bleft, len = cright - bleft + 1;
  int *res = new int[len];
  for (i = 0; i < len; i++)
  {
    if ((bleft > bright) || (cleft > cright)) break;
    if (a[bleft] <= a[cleft]) res[i] = a[bleft], bleft++;
    else res[i] = a[cleft], cleft++;
  }

  while (bleft <= bright) res[i++] = a[bleft++];
  while (cleft <= cright) res[i++] = a[cleft++];

  for (i = left; i < left + len; i++) a[i] = res[i - left];
  delete[] res;
}

void mergeSort(int *a, int left, int right)
{
  if (left >= right) return;
  int middle = (left + right) / 2;
  mergeSort(a, left, middle);
  mergeSort(a, middle + 1, right);
  merge(a, left, middle, middle + 1, right);
}

int m[1001], i, n;

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

  mergeSort(m, 1, n);

  for (i = 1; i <= n; i++)
    printf("%d ", m[i]);
  printf("\n");
  return 0;
}
```

E-OLYMP 972. Sorting time Sort the time according to specified criteria.
► Use **MergeSort** 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 **MergeSort** to sort the data.

```
MergeSort(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 **MergeSort** to sort the *Member* (participant) structures. Each participant has his own *id* and *score*.

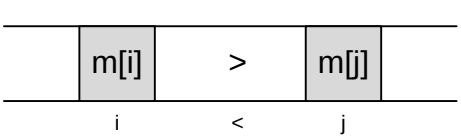
```
struct Member
{
  int id, score;
  Member(int id = 0, int score = 0) : id(id), score(score) {};
};
```

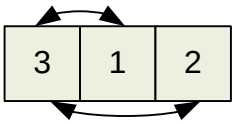
Count the inversions

E-OLYMP **8735. Train swapping** Find the number of inversions in array. Use $O(n^2)$ solution.

► Let the array m contains the input permutation – the current order of the carriages. The required minimum number of permutations equals to the number of inversions in the permutation.

An ***inversion*** is a pair of numbers (m_i, m_j) such that $i < j$ but $m_i > m_j$. That is, a pair of numbers forms an inverse if they are not in the correct order.





For example, the array m = {3, 1, 2} has two inversions: (3, 1) and (3, 2). The pair (1, 2) does not form an inversion, since the numbers 1 and 2 stand in the correct order relative to each other.

The number of inversions in the array of length *n* can be calculated using a double loop: we iterate over all possible pairs (*i*, *j*) for which $1 \le i < j \le n$, and if $m_i > m_j$, then we have an inversion.

Consider an example of counting inversions in a permutation. Under each number we write down the number of inversions that it forms with the elements to the right of it. Let inv[*i*] contains the number of *j* such that *i* < *j* and m[*i*] > m[*j*].

m[i]	4	8	2	6	5	1	7	3	
inv[i]	3	6	1	3	2	0	1	0	16

The total number of inversions is 16.

Exercise. Simulate the solution for the next sample.

m[i]	7	2	5	3	6	1	8	4	
inv[i]									

Read the input order of the carriages into the array m.

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

The minimum number of permutations for putting the train in order is calculated in the variable *res*.

```
res = 0;
for (i = 0; i < n - 1; i++)
for (j = i + 1; j < n; j++)
  if (m[i] > m[j]) res++;
```

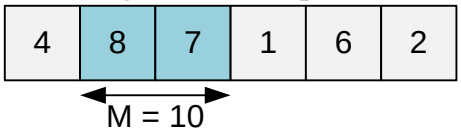
E-OLYMP 1457. “Sort” station At the “Sorting” railway station on the way there are *n* railroad cars, out of which it is necessary to form the railway train. All cars have the same length, but different cargoes are arranged at them, so the cars may have different weights. The workers of the “Sort” train station should order the cars in ascending weight, then the train is allowed to go.

Usually for such purposes the so-called shunting locomotives and electric locomotives are used, but at this station the experimental device for sorting cars is used. It is assumed that it will significantly reduce the time required for forming trains.

This device is moved on an air cushion above the cars, its length is slightly greater than the length of the two cars. It can hang on two adjacent cars, pick them both in the air and interchange. However, the device lifting capacity is limited: the operation can be carried out if the total mass of the two cars does not exceed M.

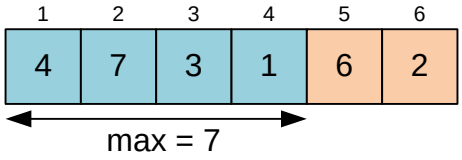
Your task is to write a program that determines whether it is possible to sort the cars on the rails with the help of experimental device in the demanding order.

► To sort the cars, we will use exchange sorting. However, the task requires not to sort the cars, but to determine whether it is possible. For this, for each adjacent pair of cars for which $m_i > m_{i+1}$, swap the cars with numbers i and $i + 1$. If there is an i such that $m_i > m_{i+1}$ and $m_i + m_{i+1} > M$, then sorting cannot be performed.



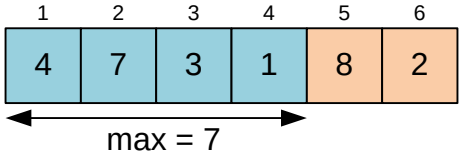
Calculate in the variable *mx* the maximum among the numbers m_1, m_2, …, m_i. If the next value m_{i+1} is less than *mx*, then the car of mass *mx* must be swapped with the car of mass m_{i+1} during the exchange sorting process. If for some i the inequality $mx + m_{i+1} > M$ holds and, in addition, $mx > m_{i+1}$, then sorting is impossible. Otherwise, the cars can be sorted in increasing order of their masses.

Let $M = 10$. Consider the next sample.



Consider the fifth element: m[5] = 6. The maximum element *mx* before it is 7 (m[2] = 7). The car with mass of 6 must stand before the car with mass of 7, therefore these cars should be interchanged. However, this is impossible, since their masses are greater than M: m[2] + m[5] > *M* or 7 + 6 > 10.

If, for example, a car with mass of 8 is in the fifth position, then the desired rearrangement is possible, since the cars with masses of 7 and 8 do not need to be swapped.



Exercise. Simulate the solution for the next samples.

	1	2	3	4	5	6	7
M = 10	4	6	2	1	5	7	3
max							

	1	2	3	4	5	6	7
M = 12	6	2	3	8	4	1	14
max							

E-OLYMP **1303. Ultra-Quick Sort** In this problem you have to analyze a particular sorting algorithm. The algorithm processes a sequence of n distinct integers

by swapping two adjacent sequence elements until the sequence is sorted in ascending order. For the input sequence

9 1 0 5 4

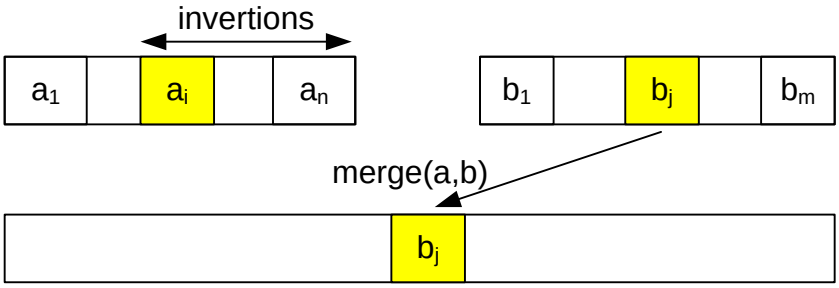
Ultra-QuickSort produces the output

0 1 4 5 9

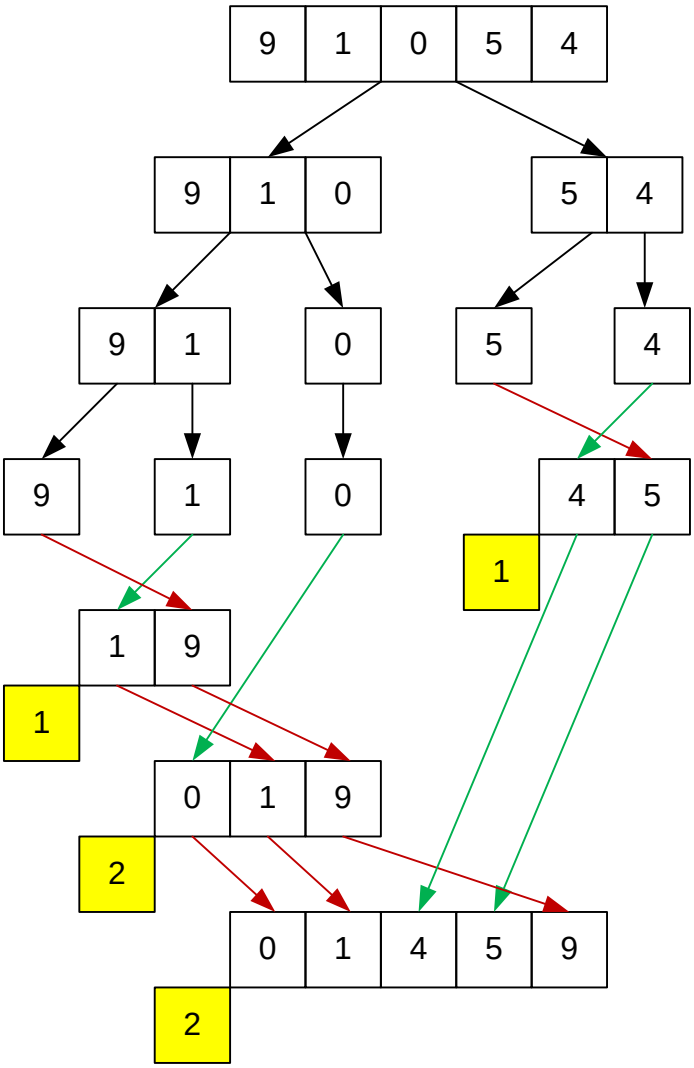
Your task is to determine how many swap operations Ultra-QuickSort needs to perform in order to sort a given input sequence.

► The minimum number of swaps required to sort the given sequence equals to the number of inversions in the input sequence. The desired number of inversions can be found in time $O(n\log_2 n)$ simulating the merge sort.

Consider the merging process of two arrays $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$. Let the parts $A = (a_1, \ldots, a_{i-1})$ and $B = (b_1, \ldots, b_{j-1})$ are already merged, and currently we compare the numbers a_i and b_j. Let $a_i > b_j$, and the element b_j is moved to the merged array. Then we claim that b_j makes inversions only with elements $a_i, \ldots, a_n$, and the number of such inversions is exactly $n - i + 1$.



Let's simulate the merge sort from the first example. Inversions are counted during the merge operation. Near each sequence obtained after merge operation, we give the number of inversions formed by the elements of the merged subsequences.


9 1 0 5 4
9 1 0
5 4
9 1
0
5
4
9
1
0
4 5
1
1 9
1
0 1 9
2
0 1 4 5 9
2