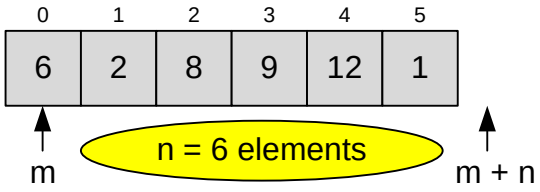
SORTING

Let m_0, m_1, …, m_{n-1} are n integers in array m (`int m[1001]`) to be sorted. You can use STL function ***sort***:

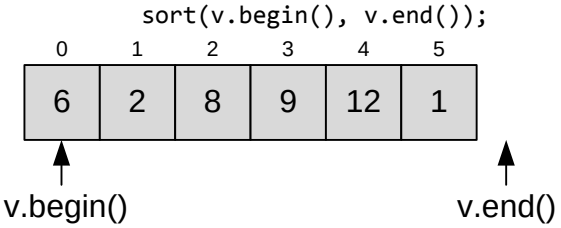
```
sort(m, m + n);
```

Here

- m = &m[0] is a pointer to the first element;
- $m + n$ = &m[n] is a pointer to the elment that is next to the last element;



If you have a vector (`vector<int> v`), use the next format:



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

► Read the data into integer array. Sort the data using STL function ***sort***. Print the data.

Store the array of integers in vector v.

```
vector<int> v;
```

Read the numbers into the array.

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

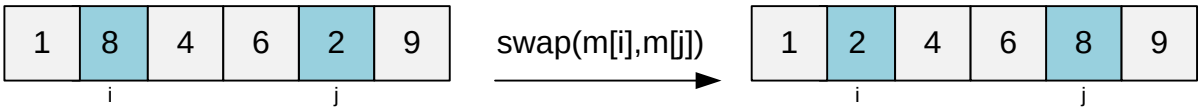
Sort it with ***sort*** function.

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

Print the elements in nondecreasing order.

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

In this problem you can use ***swap sort*** with complexity O(n^2). Since $n \le 1000$, this algorithm will pass the time limit. Iterate over all pairs (i, j), $0 \le i < j < n$, and if m[i] > m[j], swap them.




```
void sort(void)
{
  for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (m[i] > m[j])
    {
      int temp = m[i]; m[i] = m[j]; m[j] = temp;
    }
}
```

To sort the array in increasing order in Java, you can use Arrays.sort() method.

```
import java.util.*;

public class Main
{
  public static void main(String[] args)
  {
    Scanner con = new Scanner(System.in);
    int n = con.nextInt();
    int m[] = new int[n];
    for(int i = 0; i < n; i++) m[i] = con.nextInt();

    Arrays.sort(m);

    for(int i = 0; i < n; i++)
      System.out.print(m[i] + " ");
    System.out.println();
    con.close();
  }
}
```

To sort the data in increasing or decreasing order you can use **comparators**:

- `sort(m, m + n, less<int>())` – sort in increasing order;
- `sort(m, m + n, greater<int>())` – sort in decreasing order;

To sort the vector v, use

- `sort(v.begin(), v.end(), less<int>())` – sort in increasing order;
- `sort(v.begin(), v.end(), greater<int>())` – sort in decreasing order;

To sort the data in increasing order, you can omit the comparator. Default order is increasing.

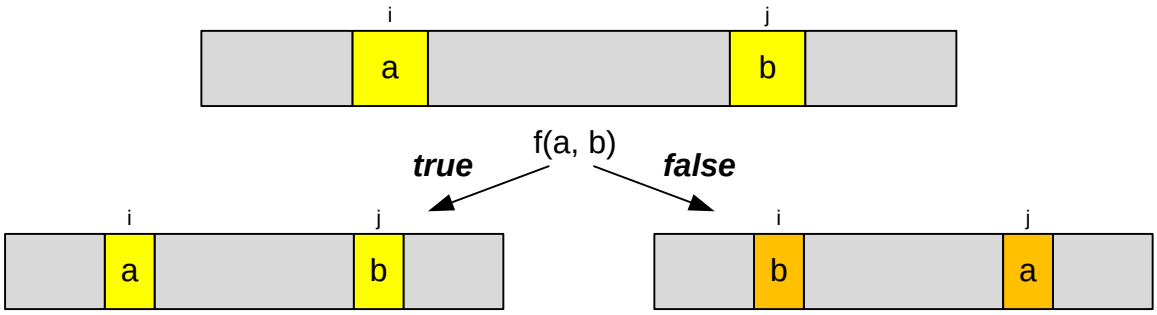
E-OLYMP 4738. Sorting Sort array of integers in non-increasing order.

► Read the data into integer array. Sort the data using STL function ***sort*** and `greater<int>()` comparator. Print the data.

You can write your own comparator. Comparator is a function of the form

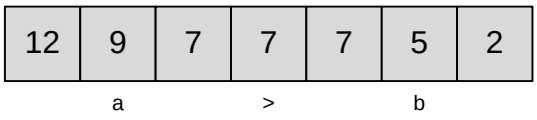
```
int f(type a, type b)
{
  . . .
}
```

that returns ***true*** (1), if elements *a* and *b* should **not** be swapped and ***false*** (0) otherwise. Consider array m and two elements: *a* = m[*i*], b = m[*j*], where *i* < *j*. If m[*i*] and m[*j*] must be swapped to preserve the sorting order, then comparator must return ***false***.



For ***sorted array*** the value of the function f(*a*, *b*) must be ***true*** for any elements *a* and *b* such that position of *a* is less than the position of *b*. The next comparator sorts array of integers in decreasing order:

```
int f(int a, int b)
{
  return a > b;
}
```



Comparator *a* < *b* is equivalent to `less<int>()`.
Comparator *a* > *b* is equivalent to `greater<int>()`.

`int f(int a, int b)`
`{`
`  return a < b;`
`}`
	`less<int>()`

`int f(int a, int b)`
`{`
`  return a > b;`
`}`
	`greater<int>()`

Let's solve this problem using our own comparator.

```
#include <cstdio>
#include <algorithm>
#define MAX 1001
using namespace std;

int m[MAX];
int i, n;

int f(int a, int b)
{
  return a > b;
```

```
}

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

  sort(m, m + n, f);

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

Below Java solution is given.

```
import java.util.*;

public class Main
{
  public static class MyFun implements Comparator<Integer>
  {
    public int compare(Integer a, Integer b)
    {
      return b - a;
    }
  }

  public static void main(String[] args)
  {
    Scanner con = new Scanner(System.in);
    int n = con.nextInt();
    Integer m[] = new Integer[n];

    for(int i = 0; i < n; i++)
      m[i] = con.nextInt();

    Arrays.sort(m, new MyFun());

    for(int i = 0; i < n; i++)
      System.out.print(m[i] + " ");

    System.out.println();

    con.close();
  }
}
```

E-OLYMP 4848. Quick sort Sort the given sequence in non-decreasing order.

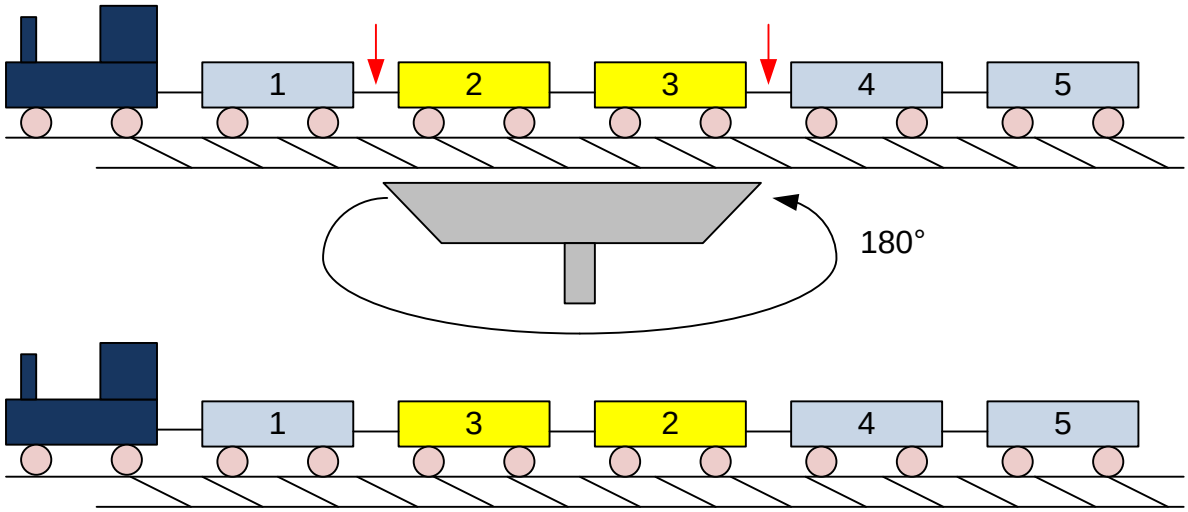
► Read the sequence of numbers into array till the *end of file* and use any sorting algorithm.

E-OLYMP 8735. Train swapping At an old railway station, you may still encounter one of the last remaining "train swappers". A train swapper is an employee of the railroad, whose sole job it is to rearrange the carriages of trains. Once the carriages are arranged in the optimal order, all the train driver has to do, is drop the carriages off, one by one, at the stations for which the load is meant.

The title "train swapper" stems from the first person who performed this task, at a station close to a railway bridge. Instead of opening up vertically, the bridge rotated around a pillar in the center of the river. After rotating the bridge 90 degrees, boats could pass left or right.

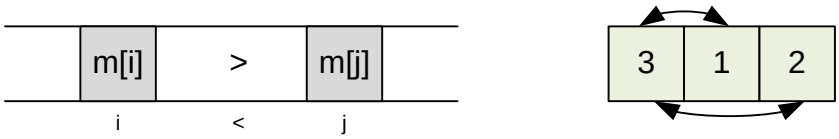
The first train swapper had discovered that the bridge could be operated with at most two carriages on it. By rotating the bridge 180 degrees, the carriages switched place, allowing him to rearrange the carriages (as a side effect, the carriages then faced the opposite direction, but train carriages can move either way, so who cares).

Now that almost all train swappers have died out, the railway company would like to automate their operation. Part of the program to be developed, is a routine which decides for a given train the least number of swaps of two adjacent carriages necessary to order the train. Your assignment is to create that routine.



► 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.

Simulate the solution for the next sample.

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

Declare the array m.

```
int m[100010];
```

Sequentially, process the test for the problem.

```
scanf("%d", &tests);

while (tests--)
{
```

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++;
```

Print the answer.

```
  printf("Optimal train swapping takes %lld swaps.\n", res);
}
```

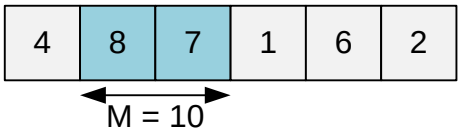
E-OLYMP 1457. “Sort” station At the “Sorting” railway station on the way there are *n* railroad cars, out of which it is necessary to form the railway train. All cars have the same length, but different cargoes are arranged at them, so the cars may have different weights. The workers of the “Sort” train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

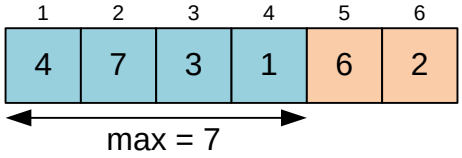
Your task is to write a program that determines whether it is possible to sort the cars on the rails with the help of experimental device in the demanding order.

► 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.



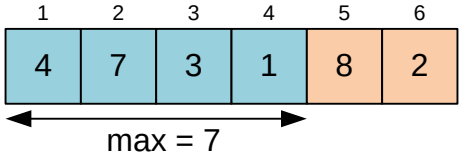
Calculate in the variable *mx* the maximum among the numbers m_1, m_2, …, m_i. If the next value m_{i+1} is less than *mx*, then the car of mass *mx* must be swapped with the car of mass m_{i+1} during the exchange sorting process. If for some i the inequality $mx + m_{i+1} > M$ holds and, in addition,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

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

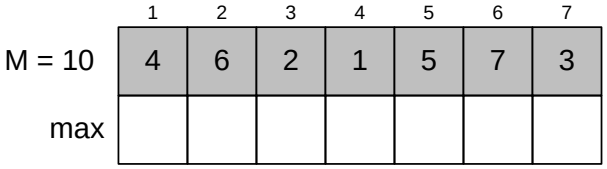


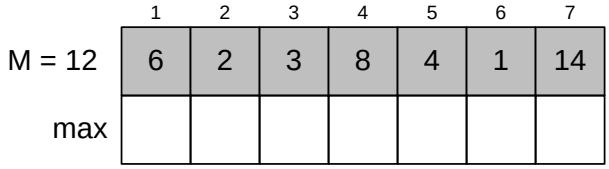
Consider the fifth element: m[5] = 6. The maximum element *mx* before it is 7 (m[2] = 7). The car with mass of 6 must stand before the car with mass of 7, therefore these cars should be interchanged. However, this is impossible, since their masses are greater than M: m[2] + m[5] > *M* or 7 + 6 > 10.

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.



Simulate the solution for the next samples.





Sort the letters / strings

E-OLYMP **8316. Sort the letters** A string consisting of lowercase Latin letters is given. Sort its letters in ascending and then in descending lexicographical order.

► Sort the strings using **sort** function from STL. Use the comparator `less<int>()` for sorting in increasing order and comparator `greater<int>()` for sorting in decreasing order.

Declare the character array.

```
#define MAX 101
char s[MAX];
```

Read the string.

```
gets(s);
```

Sort the letters in increasing order. Print the resulting string.

```
sort(s,s+strlen(s),less<int>());
puts(s);
```

Sort the letters in decreasing order. Print the resulting string.

```
sort(s,s+strlen(s),greater<int>());
puts(s);
```

Implementation using strings.

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

string s;

int main(void)
{
  cin >> s;
  sort(s.begin(),s.end(),less<int>());
  cout << s << "\n";

  sort(s.begin(),s.end(),greater<int>());
  cout << s << "\n";
  return 0;
}
```

Implementation in Java.

```
import java.util.*;

public class Main
```

```
{
  public static void main(String[] args)
  {
    Scanner con = new Scanner(System.in);
    String s[] = con.nextLine().split("");
    // s = {"q", "w", "e", "r", "t", "y"}

    Arrays.sort(s);
    System.out.println(String.join("", s));

    Arrays.sort(s,Collections.reverseOrder());
    System.out.println(String.join("", s));
    con.close();
  }
}
```

E-OLYMP 2166. Anagrams The word is an anagram of another word, if it can be obtained by rearrangement of its letters.

► Sort the letters in each word in lexicographic order. If the obtained words are the same, then they consist of the same letters and thus are anagrams.

E-OLYMP 2323. Numbers from digits Given nonnegative integer *n*. Create from all its digits the biggest and then the smallest number. Print the sum of obtained numbers.

For example, for *n* = 56002 the biggest will be 65200 and the smallest will be 256 (the leading zeros in number 00256 do not count). The resulting sum is 65200 + 256 = 65456.

► Read the number into character array. Sort the digits in descending order, get the largest number. Sort the numbers in ascending order, get the smallest number. Find and print their sum.

For example, to get the biggest number you can the next way.

```
gets(s); // read the number to har array
sort(s,s+strlen(s),greater<char>()); // sort in decreasing order
sscanf(s,"%d",&a); // get a number from char array
```

How to get a number from a string in Java.

```
// read a line, split to strings
String s[] = con.nextLine().split("");
// s = {"1", "2", "3", "4", "5", "6"}

Arrays.sort(s); // sort the chars
// Join the strings, get an integer from a string
int a = Integer.parseInt(String.join("", s));
```

E-OLYMP 2325. Two numbers Two integers are given. Print the maximum number that can be obtained from their digits.

For example, from digits of numbers 345 and 6090737 we can get the maximum number 9776543300.

► Read both numbers into a character array, thus gluing them together. Sort the numbers in decreasing order, and get the largest number.

Numbers can also be read in two strings and then concatenated. Then sort the numbers in decreasing order.

E-OLYMP 8317. Square of difference Given positive integer. Find and print the square of difference between the maximum and minimum numbers, composed from the digits of the given number.

For example, if given number is 30605, the maximum number, composed from its digits, is 65300, and minimum number is 356 (the minimum is 00356, but leading zeros are not counted). The required square of difference is (65300 – 356) * (65300 – 356) = 4217723136.

► Read the number into character array. Sort the digits in decreasing order and get the largest number *a*. Sort the digits in increasing order and get the smallest number *b*. Compute their squared difference.

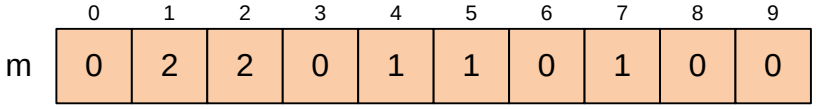
Counting sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing).

For simplicity, consider the data in the range 0 to 9.

Take a count array *m* to store the count of each unique object. For each value of *x* from input array make m[*x*]++. We have:

- two one's (m[1]++ twice)
- two two's (m[2]++ twice)
- one four, five and seven (m[4]++; m[5]++; m[7]++)



E-OLYMP 2327. Counting sort Use the magic of counting sort: sort *n* numbers in the range [0; 100000].

► Input numbers can be sorted using counting sort, as they are integers and we know their range.

Input numbers store in array m.

```
#define MAX 100010
int m[MAX];
```

Read the input data. In m[*i*] count the number of times the value *i* is encountered among the given numbers.

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

Print the resulting array. Number *i* must be printed m[*i*] times.

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

E-OLYMP 2617. Birthdays If somebody in Summer Computer School (SCS) has a birthday, the celebration is arranged. Help organizers to find out how many times they must arrange the birthday parties during SCS.August.

► Note that we are only interested in birthdays for August (one month), that has 31 days. For each day from 1 to 31, calculate how many LKSH students were born on that day. And then count the number of days when LKSH students celebrate birthdays.

E-OLYMP 354. Permutation The sequence of *n* positive integers is given. Determine whether the sequence is a permutation of the first *n* positive integers.

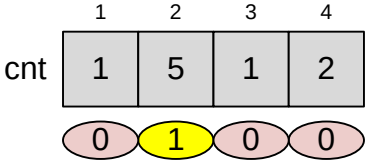
► Declare the linear array m of length $n \le 10000$. Put the amount of numbers *i* in the input sequence into cell m[*i*]. If all values m[1], m[2], …, m[*n*] are nonzero (there are exactly *n* input numbers), then all these values are equal to 1 and input seuence is a permutation. Otherwise print the smallest *i* for which m[*i*] is zero.

If some number of the input sequence is greater than *n*, then such sequence is not a permutation.

E-OLYMP 145. Squares Given the length of n segments. What is the maximum number of squares can be constructed? Each side of a square must be constructed from only one segment.

► Let we have *k* segments of the same length. Then you can make *k* / 4 squares out of them. The lengths of the segments vary from 1 to 100. Count the number of segments of length *i* ($1 \le i \le 100$) in cnt[*i*]. Then the maximum possible number of squares that can be made out of the given segments is (cnt[1] + cnt[2] + …. + cnt[100]) / 4.

Consider the state of the *cnt* array after counting sort.



From segments of length 2, you can make 5 / 4 = 1 square. It is impossible to make squares out of the remaining lengths of the line segments.

E-OLYMP 7809. Morning exercises In the morning, many students do their exercises. According to established tradition, the students dance in the branded T-shirts. During the first three days of the camp, schoolchildren and teachers noticed that a couple who dances in identical T-shirts looks more aesthetically better. Therefore before the exercises, they decided first to put pairs of children in identical T-shirts, and then the remaining ones. The excellent student Seryozha wanted to know what the largest number of aesthetic pairs can be formed from everyone who came to morning exercises.

► T-shirt color is a number from 0 to 9. Let's count the number of T-shirts of color *i* in m[*i*]. Then, out of schoolchildren wearing a T-shirt of color *i*, one can make m[*i*] / 2 aesthetic pairs. Then the total number of aesthetic pairs is

$$(m[0] + m[1] + \ldots + m[9]) / 2$$

Comparators

E-OLYMP 972. Sorting time Sort the time according to specified criteria. Print the times, sorted in nondecreasing order (time is also displayed in the form of three numbers, do not print the leading zeros).

► Declare the time structure that contains hours, minutes and seconds.

```
struct MyTime
{
  int hour, min, sec;
};

MyTime lst[100];
```

Sorting time function.

```
int f(MyTime a, MyTime b)
{
```

If hours and minutes are equal, sort the time in increasing order of seconds.

```
  if ((a.hour == b.hour) && (a.min == b.min)) return a.sec < b.sec;
```

If hours are equal (minutes are not equal), sort the time in increasing order of minutes.

```
  if (a.hour == b.hour) return a.min < b.min;
```

Otherwise sort the time in increasing order of hours.

```
  return a.hour < b.hour;
}
```

Main part of the problem. Read the input data.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
  scanf("%d %d %d",&lst[i].hour,&lst[i].min,&lst[i].sec);
```

Sort the time.

```
sort(lst,lst+n,f);
```

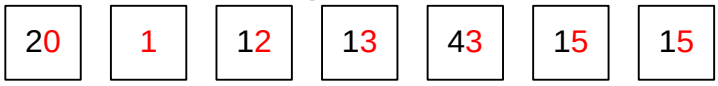
Print the time.

```
for(i = 0; i < n; i++)
  printf("%d %d %d\n",lst[i].hour,lst[i].min,lst[i].sec);
```

E-OLYMP **1462. Tricky sorting** The sequence of numbers is given. Arrange them in non-decreasing order of the last digit, and in the case of equality of last digits – arrange the numbers in non-decreasing order.

► Implement the comparator to sort the integers.

The array of integers is sorted in ascending order of the last digit. Numbers with the same last digit are sorted in ascending order.



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.

► Sort the numbers 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.

Declare the comparator ***f***.

```
int f(int a, int b)
{
```

If *a* and *b* have different parity, then even numbers must come after odd numbers.

```
  if (abs(a % 2) != abs(b % 2)) return abs(a % 2) > abs(b % 2);
```

If *a* and *b* are even, then sort them in in decreasing order.

```
  if (a % 2 == 0) return a > b;
```

If *a* and *b* are odd, then sort them in in increasing order.

```
  if (abs(a % 2) == 1) return a < b;
}
```

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.

► Declare a **Point** structure containing the *x* abscissa and *y* ordinate of the point. Implement the comparator according to the problem statement. Sort and print the points.

Declare the point structure **Point**.

```
struct Point
{
public:
  int x, y;
  Point(int x, int y) : x(x), y(y) {}
};
```

Declare the vector of points.

```
vector<Point> p;
```

Function *f* is a comparator. Sort the points in ascending order of sums of coordinates. If the points' sum of coordinates are equal, sort them in ascending order of abscissa.

```
int f(Point a, Point b)
{
  if (a.x + a.y == b.x + b.y) return a.x < b.x;
  return a.x + a.y < b.x + b.y;
}
```

The main part of the program. Read the input points.

```
while (scanf("%d %d", &x, &y) == 2)
  p.push_back(Point(x, y));
```

Sort the points.

```
sort(p.begin(), p.end(), f);
```

Print the sorted points.

```
for (i = 0; i < p.size(); i++)
  printf("%d %d\n", p[i].x, p[i].y);
```