Input File: `therace.in` **100 Points**
Output File: `therace.out` **Time limit:** 8 s
Source File: `therace.pas/.c/.cpp` **Memory limit:** 16 MB

Solution

This task comprises two subtasks which are virtually independent from each other.

The first task is to calculate the number of passes. As all velocities are constant there is a point in time after which no more passes will occur. Then the space crafts are ordered by their speed. The groups of space crafts flying with the same speed are ordered by their starting position. This reminds us of a stable sorting algorithm where only adjacent elements are exchanged in one step: Bubblesort. In fact, the first subtask of our problem is equal to calculating the number of swap-operations that the Bubblesort algorithm will perform on the given sequence of speeds.

There are two simple algorithms to calculate the number of passes. The simplest one loops over all ships from the course head to the course tail, updating the number of ships behind the current position and counting how many of these will overtake the current ship. This solution runs in $O(N \cdot V_{max})$. It works as follows:

```
int numPasses=0, numWithSpeed[MAX_SPEED+1];
for (int i=MAX_SPEED;i>0;i--) numWithSpeed[i]=0;
for (int i=0; i < N; i++) numWithSpeed[ship[i].speed]++;
for (int i=N-1;i>=0;i--){          // for all ships (first to last)
  numWithSpeed[ship[i].speed]--;
  // count passes of ships that are behind position i
  for (int v=ship[i].speed+1;v<=MAX_SPEED;v++)
    numPasses = (numPasses + numWithSpeed[v]) % 1~000~000;
}
```

There is also a *Divide and Conquer* solution that runs in $O((N+V_{max}) \cdot \log N)$ which is implemented in the Pascal sample solution. Whenever this solution combines two equally sized smaller solutions, it only regards the overtakes where ships from the left half overtake ships of the right half. It counts the number of ships for each speed in the left half and in the right half and uses this information to count the overtakes in $O(N+V_{max})$. The additional logarithmic factor is determined by the recursion depth of the *Divide and Conquer* approach.

The second task is to print the first 10 000 passes in chronological order.

In worst case there are $O(N^2)$ passes. Comparing all pairs of space crafts, determining their possible passes, and sorting them would need $O(N^2 \log N)$. This cannot be done within the given timelimit.

The following algorithm leads to a runtime of $O(N + c \log N)$ where c denotes the number of passes to print.

A space craft i can only pass another space craft j, if j is flying directly in front of i. The idea is to maintain a list of space crafts and a heap which stores for every space craft i which space craft j is currently in front of i as well as the time when i will pass j. If i does not pass j, we do not store an entry. The top element of the heap is the event that will occur next.

We simulate the first 10 000 passes: We remove the first event, like space craft i passes j, from the queue and print it. Next, we exchange i and j in the list which results in a new predecessor for i and a new successor for j. This means that we have to update our heap:
If i passes another space craft in the future, we will insert a new event into the heap. If the successor of j passes i, it will pass j as well – In fact, it will pass j before i, so we have to update its event in the heap. This is easy, as we just have to move it upwards.

The algorithm looks like this:

```
sort space_crafts;
for i = 1 .. N–1
    if space_crafts[i] passes space_crafts[i+1]
        store event in heap;
for i = 1 .. 10~000
    if heap empty: break;
    event = heap.pop();
    print event;
    space_crafts.exchange(event.i,event.j);
    if space_crafts[event.i] passes space_crafts[event.i+1]
        store event in heap;
    if space_crafts[event.j–1] passes space_crafts[event.j]
        update event of spacecraft j;
```