BOI 2007

Tasks and Solutions

Contents

1 Introduction **3**

2 Task 1: Escape **5**
2.1 Task description 5
2.2 Solution description 7

3 Task 2: Sorting **9**
3.1 Task description 9
3.2 Solution description 10

4 Task 3: Sound **13**
4.1 Task description 13
4.2 Solution description 14

5 Task 4: Fence **18**
5.1 Task description 18
5.2 Solution description 19

6 Task 5: Points **21**
6.1 Task description 21
6.2 Solution description 22

7 Task 6: Sequence **27**
7.1 Task description 27
7.2 Solution description 28

The source code of the sample programs as well as the test data and this document in PDF format are available online at `http://www.boi2007.de`.

Introduction

The 13th Baltic Olympiad in Informatics took place in Güstrow, Germany, from April 24 to April 28, 2007. Every year, countries around the Baltic Sea send delegations of up to six of their most talented high school computer scientists to represent their countries. This year, the organizers were happy to host delegations from Denmark, Estonia, Finland, Germany, Latvia, Lithuania, Norway, Poland and Sweden. A delegation from the hosting state Mecklenburg West-Pomerania completed the field.

The contest comprises two sessions, in both of which participants are required to solve three tasks. For each task, a contestant should submit a program that correctly and efficiently calculates the answer to the given problem. It is a tradition of the Baltic Olympiad in Informatics that each country submits a task proposal. From these proposals two times three tasks were selected that are challenging and cover a big variety of problem types. This booklet contains these tasks and describes their solutions.

Special thanks go to the task authors and other contributors to this booklet:

Day 1

Task: escape (LTU)
 Task author: Vilius Naudžiūnas
 Solution: Linas Petrauskas, Thomas Fersch

Task: sorting (GER)
 Task author: Adrian Kügel
 Solution: Adrian Kügel

Task: sound (EST)
 Task author: Ahto Truu
 Solution: Ahto Truu, Jakub Radoszewski, Erik Panzer

Day 2

Task: fence (SWE)
Task author: Andreas Björklund
Solution: Jimmy Mårdell

Task: points (LAT)
Task author: Rihards Opmanis
Solution: Alexander Hullmann

Task: sequence (POL)
Task author: Jakub Radoszewski
Solution: Jakub Radoszewski

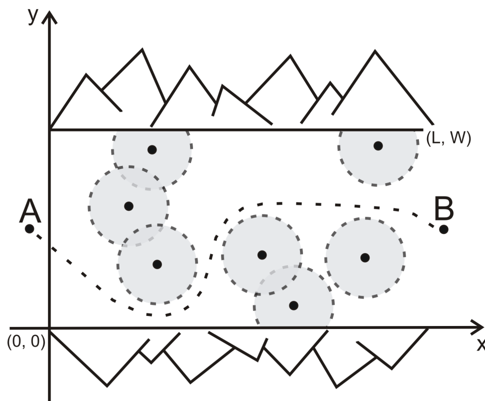
The scientific committee was represented by Felix Arends, Dominic Battré, Hans-Christian Ebke, Thomas Fersch, Alexander Hullmann, Adrian Kügel, Erik Panzer, and Tobias Polley. Many of their efforts went into writing sample solutions and creating and validating test cases.

Dominic Battré

Task 1: Escape

2.1 Task description

A group of war prisoners are trying to escape from a prison. They have thoroughly planned the escape from the prison itself, and after that they hope to find shelter in a nearby village. However, the village (marked as B, see picture below) and the prison (marked as A) are separated by a canyon which is also guarded by soldiers. These soldiers sit in their pickets and rarely walk; the range of view of each soldier is limited to exactly 100 meters. Thus, depending on the locations of soldiers, it may be possible to pass the canyon safely, keeping the distance to the closest soldier *strictly larger* than 100 meters at any moment.



You are to write a program which, given the width and the length of the canyon and the coordinates of every soldier in the canyon, and assuming that soldiers do not change their locations, first determines whether prisoners can pass the canyon unnoticed. If this is impossible then the prisoners (having seen enough violence) would like to know the minimum number of soldiers that have to be eliminated in order to pass the canyon safely. A soldier may be eliminated regardless of whether he is visible to any other soldier or not.

Input

The input is read from a text file named `escape.in`. The first line contains three integers L, W, and N – the length and the width of the canyon, and the number of soldiers, respectively. Each of the following N lines contains a pair of integers X_i and Y_i – the coordinates of i-th soldier in the canyon ($0 \leq X_i \leq L$, $0 \leq Y_i \leq W$). The coordinates are given in meters, relative to the canyon: the southwestern corner of the canyon has coordinates $(0, 0)$, and the northeastern corner of the canyon has coordinates (L, W), as seen in the picture above.

Note that passing the canyon may start at coordinate $(0, y_s)$ for any $0 \leq y_s \leq W$ and end at coordinate (L, y_e) for any $0 \leq y_e \leq W$. Neither y_s nor y_e need to be integer.

Output

The output is written into a text file named `escape.out`. In the first and only line of the output file the program should print the minimum number of soldiers that have to be eliminated in order for the prisoners to pass the canyon safely. If the prisoners can escape without any elimination, the program should print 0 (zero).

Example

escape.in	escape.out
130 340 5 10 50 130 130 70 170 0 180 60 260	1

Constraints

$1 \leq W \leq 50{,}000 \quad 1 \leq L \leq 50{,}000 \quad 1 \leq N \leq 250$

Grading

Your program will receive partial credits if it can only determine whether the prisoners need to eliminate any guard at all in order to escape. For this, several test runs will be grouped to one test group. You will receive 30% of a test groups's credits in case you determine for each test run correctly whether any guards need to be eliminated (0 means no guards need to be eliminated, any integer > 0 means that any number of guards need to be eliminated). You will receive 100% of a test group's credits in case you determine for each test run correctly how many guards need to be eliminated for the prisoners' escape.

2.2 Solution description

The soldiers and the canyon can be modeled as an undirected graph G: Let each soldier be represented by one vertex, where there is an edge between two vertices if and only if areas visible by the two corresponding soldiers intersect or touch, i.e. the distance between the two soldiers is less than or equal to 200. Let there be two additional vertices s and t, where s represents the northern side of the canyon and t the southern side of the canyon. Let these vertices be connected to all vertices representing soldiers who can see the respective side of the canyon, i.e. whose distance from the side is at most 100 meters.

Now it is fairly easy to decide if the canyon can be passed safely. Just perform either depth-first-search or breadth-first-search to check whether there is a path between s and t in G. If that is the case, then there exists a continuous chain of areas visible by some soldier that connects the northern and southern sides of the canyon, thus keeping the prisoners from passing it. Vice versa, if there is no such chain of visible areas, the canyon can clearly be passed safely.

Deciding how many soldiers must be eliminated in order to pass the canyon safely is a bit more complicated. Since the canyon can be passed safely if and only if there is a path between s and t in G, one has to find $(s-t)$-vertex-connectivity of G, i.e. how many vertices in G have to be deleted in order to disconnect s and t. This is essentially a minimum cut problem. It can be solved by setting edge capacities to ∞, vertex capacities to 1 and finding the maximum flow from s to t (contrary to the standard minimum cut problem for edges, where vertex capacities are set to ∞ and edge capacities to 1). To make standard maximum flow algorithm applicable, vertex capacities have to be eliminated, though. This can be done relatively easy by constructing a directed graph G', where each vertex of G except s and t is replaced by two

vertices, in-vertex and out-vertex. If edges in G' are set according to figure 2.1 below, then the maximum flow from s to t in G will be equal to the maximum flow from s to t in G'. Since there are no vertex capacities in G', this maximum flow and hence $(s-t)$-vertex-connectivity of G can now be found using standard maximum flow algorithm.

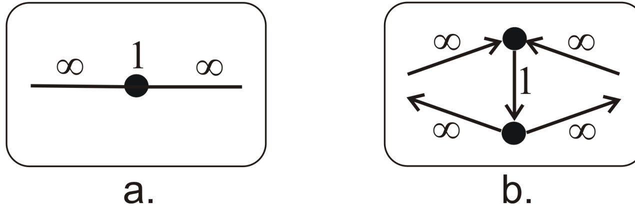


Figure 2.1: Converting a vertex capacity (a) to an edge capacity (b)

Task 2: Sorting

3.1 Task description

You are given the scores of several players in a competition. Your task is to create a ranklist of the players, sorted in decreasing order by score.

Unfortunately, the data structure used for the list of players supports only one operation, which moves a player from position i to position j without changing the relative order of other players. If $i > j$, the positions of players at positions between j and $i-1$ increase by 1, otherwise if $i < j$ the positions of players at positions between $i+1$ and j decrease by 1.

This operation takes i steps to locate the player to be moved, and j steps to locate the position where he or she is moved to, so the overall cost of moving a player from position i to position j is $i + j$. Here, positions are numbered starting with 1.

Determine a sequence of moves to create the ranklist such that the sum of the costs of the moves is minimized.

Input

The input is read from a text file named `sorting.in` . The first line contains n ($2 \leq n \leq 1000$), the number of players. Each of the following n lines contains one non-negative integer s_i ($0 \leq s_i \leq 1{,}000{,}000$), the scores of the players in the current order. You may assume that all scores are distinct.

Output

The output is written into a text file named `sorting.out` . In the first line of the output print the number of moves used to create the ranklist. The following lines should specify the moves in the order in which they are applied. Each move should be described by a line containing two integers i and j, which means that the player at position i is moved to position j. The numbers i and j must be separated by a single space.

Example

sorting.in	sorting.out
5 20 30 5 15 10	2 2 1 3 5

Grading

30% of the test cases have values of $n \leq 10$

3.2 Solution description

Important observations

When looking at the description of what happens when moving a player, we can see that if we move a player towards the front of the list, the moving cost of some other players increases, whereas if we move a player towards the end of the list, the moving cost of some other players decreases. So, it might be helpful to move players first which have to be moved towards the end of the list to decrease the moving costs of future moves. This observation may lead to following two ideas:

1. Each player is moved at most once.

2. The players who are moved are moved in ascending order according to their score.

A proof that these two ideas are correct will follow later.

Brute force solution

Using the two ideas mentioned above, a brute force solution would try all selections of players to be moved, and then move them to their required position in ascending order according to their score. The complexity of this solution would be $O(n^2 \cdot 2^n)$.

Dynamic programming solution

Let the players be numbered from 1 to n according to their final position in the ranklist. With dynamic programming, we want to determine the minimum cost to get the players $i \dots n$ in their correct relative order with the first of these players at original position j. It is tricky to determine the current position of the next player to be moved. It would be far easier if we knew that all players $(i+1)\dots n$ already moved towards the end of the list; in that case, in order to determine the current position of player i, we could count how many players with bigger scores exist which have an original index position smaller than player i. Obviously, if some of the players with the $n-i$ smallest scores didn't move, the current position for player i determined by this method can be too small.

However, if we assign a moving cost to each player who doesn't move we can take care of the underestimated moving cost for players with bigger score. Let's see what happens if we didn't move player j. This means that players with bigger score will be moved before him, so when determining the position of a player i which appears after player j in the list, the determined rank would be too small by $j - i$. So the moving cost for a player who doesn't move is the sum of $j - p_i$ for all players $p_i < j$ which appear between player j and the position of player $j + 1$.

It is possible to implement a dynamic programming solution using this idea with complexity $O(n^2)$ (see `sorting.cpp`). With the dynamic programming we can determine the minimal overall moving costs and which players were moved. Now, we simply need the second part of the brute force solution to generate the actual moves performed.

Proof

Here is the proof why the two ideas mentioned above are correct:
Lets look at the player $p_{\min}$ with the smallest score, who has to end up at position n.

Claim 1: If $p_{\min}$ is moved at all, it is optimal to move him directly to the end of the list.

Proof: Obviously, moving him towards the front will only increase the costs of other players to be moved. So assume he will be moved towards the end to a position $i < n$ (thus saving $n - i$ steps). Now there are two possibilities: either, the $n-i$ players after him in the list have to be moved and their costs

are increased by 1 (as opposed to that $p_{\min}$ was moved to the end), or $p_{\min}$ has to be moved again, which clearly is not optimal.

Claim 2: If $p_{\min}$ is moved at all, we can do this in the first move.

Proof: Following from Claim 1, we already know the second part of the moving costs is fixed. So the only reason not to move $p_{\min}$ first is that his current position decreases caused by moving other players. But these players have their moving costs increased by 1 because $p_{\min}$ wasn't moved to the end.

If $p_{\min}$ is moved, we have reduced our problem to the same problem with $n-1$ players. If $p_{\min}$ is not moved, we have a similar problem with $n-1$ players with the additional restriction that all players after $p_{\min}$ have to be moved before $p_{\min}$.

Let $p'_{\min}$ be the player with the smallest score among the remaining $n-1$ players. We can see that it does not help to move $p'_{\min}$ after $p_{\min}$, because that would mean we have to move him again, and we gain nothing from moving him past other players still to be moved (for each of them, we decreased the moving cost by 1, and increased the moving cost of $p'_{\min}$ by at least one). If $p'_{\min}$ is currently placed before $p_{\min}$ in the list, by similar arguments as for Claim 1 we can argue that if $p'_{\min}$ is moved at all, it is optimal to move him directly before $p_{\min}$. Also, Claim 2 still works. So we are left with the case that $p'_{\min}$ is currently placed after $p_{\min}$ and has to be moved towards the front. If there are players between $p_{\min}$ and $p'_{\min}$, we could consider moving them first. If we move $p'_{\min}$ first, the start positions of the players in between are increased by 1. But if we move some player in between first, the end position where $p'_{\min}$ has to be moved to increases by 1, so it can't be bad if we move $p'_{\min}$ first.

So, we can see that by using induction and the arguments presented above we can prove that each player is moved at most once, and the players can be moved in ascending order according to their score.

Task 3: Sound

4.1 Task description

In digital recording, sound is described by a sequence of numbers representing the air pressure, measured at a rapid rate with a fixed time interval between successive measurements. Each value in the sequence is called a *sample*.

An important step in many voice-processing tasks is breaking the recorded sound into chunks of non-silence separated by silence. To avoid accidentally breaking the recording into too few or too many pieces, the silence is often defined as a sequence of m samples where the difference between the lowest and the highest value does not exceed a certain treshold c.

Write a program to detect silence in a given recording of n samples according to the given parameter values m and c.

Input

The input is read from a text file named `sound.in`.

The first line of the file contains three integers: n ($1 \leq n \leq 1{,}000{,}000$), the number of samples in the recording; m ($1 \leq m \leq 10{,}000$), the required length of the silence; and c ($0 \leq c \leq 10{,}000$), the maximal noise level allowed within silence.

The second line of the file contains n integers a_i ($0 \leq a_i \leq 1{,}000{,}000$ for $1 \leq i \leq n$), separated by single spaces: the samples in the recording.

Output

The output is written into a text file named `sound.out`.

The file should list all values of i such that $\max(a[i \ldots i+m-1]) - \min(a[i \ldots i+m-1]) \leq c$. The values should be listed in increasing order, each on a separate line.

If there is no silence in the input file, write `NONE` on the first and only line of the output file.

Example

sound.in	sound.out
7 2 0 0 1 1 2 3 2 2	2 6

4.2 Solution description

There are quite a few possible solutions in this task.

The simplest one is to keep a "sliding window" buffer of the last m values seen and scan it after each new sample to determine the lowest and the highest value in the buffer. This obviously runs in $O(nm)$ time, which is too slow for any larger test cases. Such a solution is given in the file `sound1a.pas` and is expected to score about 40%.

An easy improvement over the naïve solution is to cache the minimum and maximum and only scan the full buffer when a current extreme value is pushed out. While this still runs in $O(nm)$ time in the worst case, it is much faster in many cases. Such a solution is given in the file `sound1b.pas` and is expected to score about 50%, as the larger test cases are engineered against this heuristic.

An asymptotic improvement is to build a binary tree that keeps the last m values in the leaves and the minimum and maximum of each subtree in the internal nodes. Then the global minimum and maximum can be updated in $O(\log m)$ steps by replacing the oldest value in the buffer with the new one and updating the minima and maxima on the path from the updated leaf to the root of the tree and the overall running time is $O(n \log m)$. Such a solution is given in the file `sound1c.pas` and is expected to score full points.

Another solution is to keep a "last seen" index for each possible sample value and scan this table after each new sample to determine the lowest and highest value seen no more than m samples ago. This obviously runs in $O(na)$ time, where a is the size of the range of sample values. Such a solution is given in the file `sound2a.pas` and is expected to score about 20%.

Like in the "sliding window" case, we can cache the minimal and maximal values seen recently and use this information to reduce the scanning range. This reduces the worst-case running time to $O(na/m)$ and proves to be surprisingly efficient on the average. Such a solution is given in the file `sound2b.pas` and is expected to score about 50%, as the larger test cases are engineered against this heuristic.

The "last seen"-based solution can also be augmented with a binary tree.

This time we need to keep only the maximum of each subtree, as this is sufficient to reduce the search for the highest and lowest value seen among the last m samples to $O(\log a)$ steps and thus the total running time to $O(n \log a)$. Such a solution is given in the file `sound2c.pas` and is expected to score about 70%.

Yet another solution is to keep a "number of times seen among the last m" counter for each possible sample value in addition to the m-element "sliding window" buffer. Using the buffer, we can update the counters in constant time when a new sample arrives and an old one retires. Using the counters and the most recent sample a_i, we can check in $O(c)$ steps if the values $a_i+j-c \ldots a_i+j$ account for all m samples for some $j \in 0 \ldots c$. The total running time of this solution is $O(nc)$. Such a solution is given in the file `sound3.pas` and is expected to score abut 60%.

An $O(n \log n)$ solution could be written using a segment tree on the whole input sequence. Such a solution would be expected to score about 80%.

An optimal solution can be developed considering the prefixes and suffixes of m-element blocks of the input sequence. For each block, we can sweep the block from left to right and then from right to left to compute the minimal values for each prefix and suffix of the block, as shown in table 4.1 for the case $n = 16$, $m = 4$. An arbitrary window consists of a suffix of one block and a prefix of the following one (shown in bold in the first row of the table). The minimal value of the window can be computed in constant time by taking the minimum of the min-suffix and the min-prefix corresponding to the suffix and prefix that make up the window (shown in bold in the second and third row). The maximal value can be computed similarly, which yields a solution that runs in $O(n)$ time and $O(n)$ space. Such a solution is given in the file `sound5.pas` and is expected to score full points. Actually, it would be sufficient to keep only two adjacent blocks in the memory at any time, and thus reduce the memory requirement to $O(m)$.

Sequence	2	8	4	1	7	2	**7**	**4**	**3**	**9**	4	7	2	3	5	8
Min-Suffix	1	1	1	1	2	2	**4**	4	3	4	4	7	2	3	5	8
Min-Prefix	2	2	2	1	7	2	2	2	3	**3**	3	3	2	2	2	2

Table 4.1: Computing the minimal value in a window

Another optimal solution can be derived from the observation that after encountering a value x, any previously encountered value $y \leq x$ will not affect computing the maximum any more. Therefore we can replace the naïve "sliding

window" with a list of indices $j_1 < j_2 < \ldots < j_k$ such that $j_1 > i - m$ and $a_{j_1} > a_{j_2} > \ldots > a_{j_k}$. When a new value a_i arrives, the list can be updated as follows:

1. If $j_1 = i - m$, remove j_1 from the front of the list.
2. While $k > 0$ and $a_{j_k} \leq a_i$, remove j_k from the back of the list.
3. Append i to the back of the list.

It should be obvious that the preceding steps quarantee that at any time the value a_{j_1} is the maximum among the last m samples. While the algorithm can spend $O(m)$ operations to process the arrival of a sample a_i, the total cost of processing all n samples can't exceed $O(n)$, as each index enters and leaves the list only once. Maintaining the minimal value among the last m samples can be done similarly, or just by keeping another list that is updated based on the value $-a_i$ for each sample a_i. Such a solution using $O(n)$ time and $O(m)$ memory is given in the file `sound6.pas` and is expected to score full points.

The last solution has the additional benefit that it is easily modifiable for the case when silence is defined as at least m samples where the noise level does not exceed the threshold. Also, with a bit more careful programming, we could get away using only $O(c)$ memory. This could be useful when analyzing long recordings containing potentially long sections of silence on a device with limited memory.

Testcases

1. $n = 18$, $m = 6$, $c = 1$, $1 \leq a_i \leq 9$.
   A small straightforward case.

2. $n = 150$, $m = 6$, $c = 6$, $0 \leq a_i \leq 15$.
   Some overlapping periods of silence.

3. $n = 1{,}500$, $m = 6$, $c = 6$, $0 \leq a_i \leq 100$.
   Boundary conditions: both the first and the last sample belong to silence. Also big enough that a naïve $O(na)$ solution should time out.

4. $n = 131{,}072$, $m = 8{,}192$, $c = 64$, $0 \leq a_i \leq 100{,}000$.
   Special case: n, m, c have form 2^i, which could throw off a sloppily implemented binary tree. Also big enough that a naïve $O(nm)$ solution should time out.

5. $n = 131{,}071$, $m = 8{,}191$, $c = 63$, $0 \leq a_i \leq 100{,}000$.
   Special case: n, m, c have form $2^i - 1$, which could throw off a sloppily implemented binary tree. Also proves that the optimized $O(nm)$ solution is still $O(nm)$ in the worst case.

6. $n = 131{,}073$, $m = 17$, $c = 65$, $0 \leq a_i \leq 100{,}000$.
   Special case: n, m, c have form $2^i + 1$, which could throw off a sloppily implemented binary tree. Also proves that the optimized $O(na/m)$ solution is still $O(na)$ if m is small enough.

7. $n = 500{,}000$, $m = 750$, $c = 500$, $0 \leq a_i \leq 1{,}000{,}000$.
   Big enough that an $O(nc)$ solution should time out.

8. $n = 800{,}000$, $m = 750$, $c = 800$, $0 \leq a_i \leq 1{,}000{,}000$.

9. $n = 900{,}000$, $m = 750$, $c = 900$, $0 \leq a_i \leq 1{,}000{,}000$.

10. $n = 1{,}000{,}000$, $m = 750$, $c = 1000$, $0 \leq a_i \leq 1{,}000{,}000$.

11. $n = 1{,}500$, $m = 750$, $c = 6$, $0 \leq a_i \leq 15$
    Correct output is NONE.

12. $n = 1{,}000{,}000$, $m = 10{,}000$, $c = 4859$.
    Test case with maximum n and m, very few periods of silence.

13. $n = 1{,}000{,}000$, $m = 5{,}000$, $c = 2{,}600$.
    Many periods of silence, only linear time complexity solutions should pass.

Task 4: Fence

5.1 Task description

Leopold is indeed a lucky fellow. He just won a huge estate in the lottery. The estate contains several grand buildings in addition to the main mansion, in which he intends to live from now on. However, the estate lacks a fence protecting the premises from trespassers, which concerns Leopold to a great extent. He decides to build a fence, but unfortunately he cannot afford to put it round all of his newly acquired land. After some thinking, he decides it is sufficient to have a fence that encloses the main mansion, except for one important restriction: the fence must not lie too close to any of the buildings. To be precise, seen from above, each building is enclosed in a surrounding forbidden rectangle within which no part of the fence may lie. The rectangles' sides are parallel to the x- and y-axis. Each part of the fence must also be parallel either to the x-axis or the y-axis.

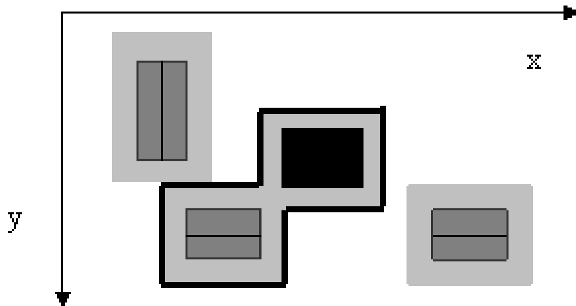


Figure 5.1: The main mansion (black) and three other buildings with surrounding forbidden rectangles. The thick black line shows a shortest allowed fence enclosing the main mansion.

Input

The input is read from a text file named `fence.in`. The first line of the input file contains a positive integer m ($1 \le m \le 100$), the number of buildings of the estate. Then follow m lines each describing a forbidden rectangle enclosing a building. Each row contains four space-separated integers tx, ty, bx, and by, where (tx, ty) are the coordinates of the upper left corner and (bx, by) the coordinates of the bottom right corner of the rectangle. All coordinates obey $0 \le tx < bx \le 10{,}000$ and $0 \le ty < by \le 10{,}000$. The first rectangle is the forbidden rectangle enclosing the main mansion.

Output

The output is written into a text file named `fence.out`. It contains one line with a single positive integer equal to the minimum length of any allowed fence enclosing the main mansion.

Example

fence.in	fence.out
4 8 4 13 8 2 1 6 7 4 7 9 11 14 7 19 11	32

Grading

In 30% of the testcases $m \le 10$ holds.

5.2 Solution description

We can build a graph G from the input data by considering all rectangle corners that do not lie strictly inside any other rectangle as a vertex. Since there are at most 100 rectangles, the graph G will have at most 400 vertices. We add an edge between two pairs of vertices x, y in G if the rectangular area between the corners they represent is empty. The weight (length) of this edge will be the Manhattan distance between x and y. A closed path in G will correspond

to a fence where the length of the path is the length of the fence. It is easy to see that an optimal fence can always be built according to the edges in G (non-horizontal/vertical edges can be split into one horizontal and one vertical component).

We must now make sure that the path goes around the main mansion. To understand the idea outlined below, remember that to determine if a point lies inside a polygon (which the fence can be conceived as), one can draw a straight line from the point to some known place outside the polygon and count the parity of the number of line intersections.

Let X, Y be the coordinates of the top left corner of the mansion. The idea is to extend the graph G into two layers, corresponding to an even or an odd number of revolutions. Call this new graph H. When traversing an edge that corresponds to the movement $(x_1, y_1) \rightarrow (x_2, y_2)$ where $x_1 \leq X$ and $x_2 > X$ and $y_1, y_2 \leq Y$, the number of revolutions traversed increases by 1. If we go the other direction, it decreases. In both cases the parity of the number of revolutions changes. Now the problem becomes to find the shortest path between some node and the same node but in the other layer of the graph H. This can be solved using standard Dijkstra algorithm (simplified version is enough), once for each node in the first layer in H.

Task 5: Points

6.1 Task description

Consider a regular grid of $3 \times N$ points. Every point in the grid has up to eight neighboring points (see Fig. 6.1).

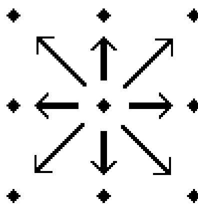


Figure 6.1: Neighboring points (marked by arrows).

We are interested in counting the number of different ways to connect the points of the grid by polygons that fulfill the following conditions:

1. The set of vertices of the polygon consists of all $3 \times N$ points.
2. Adjacent vertices of the polygon are neighboring points in the grid.
3. Each polygon is simple, i.e. there must not be any self-intersections.

Two possible polygons for $N = 6$ are given in the Fig. 6.2.

Write a program that calculates for a given N the number of possible ways to connect the points as described modulo 1,000,000,000.

Input

The input is read from a text file named `points.in`. The first and only line contains one positive integer N ($N \leq 1{,}000{,}000{,}000$).

Figure 6.2: Two possible connections of points for $N = 6$.

Output

The output is written into a text file named `points.out`. The only line to be written contains the remainder of the number of ways to connect the points modulo 1,000,000,000.

Example

points.in	points.out
4	40

Grading

- 30% of the test cases have values of $N \leq 200$.
- 60% of the test cases have values of $N \leq 100{,}000$.

6.2 Solution description

Firstly, if you try to draw a polygon fulfilling the necessary conditions, you are strongly limited in the ways you can move. When starting from the lower left point of the grid in an anti-clockwise direction, you have to get to the lower right point first before you can move to any point in the top line of the grid. This is due to the fact that your polygon would otherwise split the grid in two parts, although you still had to get to the lower right and to the upper left point. Since this is impossible, we can deduce that there is a bottom part of the polygon never connecting points in the top line of the grid and a top part of the polygon never connecting points in the bottom line in the grid.

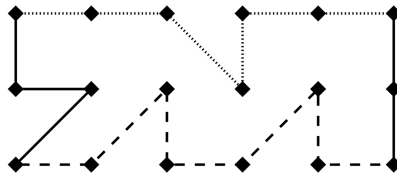


Figure 6.3: Dashed line: Bottom part. Pointed line: Top part.

Secondly, that means that when drawing the top or the bottom part of the polygon from left to right, only two lines of the grid may be used which means you cannot move more than one column in the opposite direction. Otherwise the way to the rightmost column could not be completed.

With these observations, we can take advantage of a left-right-ordering: We cut a valid polygon at a specified column vertically in two parts and look at the left part of it. There will be two open endings (first observation) at which we could continue to draw the polygon. Apart from the last column the shape of the left part of the polygon does not influence the number of ways we can continue to draw it, since it is not possible to go more than one column backwards (second observation).

Now we have to identify the situations or states that occur at the specified column where we made our vertical cut. With each state we associate how many possibilities we had to draw the left part of the polygon. When we find a recursive formula to get from the states of column i to the states of column $i+1$ we can construct all possibilities from left to right, and we have basically solved the problem.

The states you choose to identify are arbitrary, as long as they are correct. A very short solution is to discard any vertical connections in the column we look at, and to associate with the upper, the middle and the lower point of the column how many connections they have to the column to the left.

State 1	State 2	State 3	State 4	State 5	State 6
2	1	1	1	1	0
1	1	2	1	0	1
1	2	1	0	1	1

One example for each state is shown in fig. 6.4. If we are in column i we do not need any other information than what state the left part of the polygon

is in and how many possibilities we had to draw it so far. Let us denote that number with $s_i(state)$.

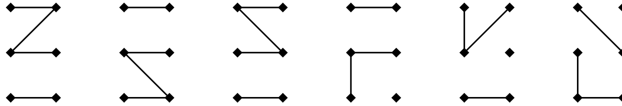


Figure 6.4: Examples for states 1, 2, 3, 4, 5 and 6.

Now we have to find the recurrence relationship for our s_i:

$$
\begin{aligned}
s_{i+1}(1) &= s_i(5) \\
s_{i+1}(2) &= s_i(5) \\
s_{i+1}(3) &= s_i(4) + s_i(6) \\
s_{i+1}(4) &= s_i(1) + s_i(2) + s_i(3) + s_i(4) + 2s_i(5) + s_i(6) \\
s_{i+1}(5) &= s_i(1) + s_i(2) + s_i(3) + s_i(4) + 2s_i(5) + s_i(6) \\
s_{i+1}(6) &= s_i(1) + s_i(2) + s_i(3) + s_i(4) + 2s_i(5) + s_i(6)
\end{aligned}
$$

In fig. 6.5 all possibilities to construct state 6 out of the column to the left are being shown.

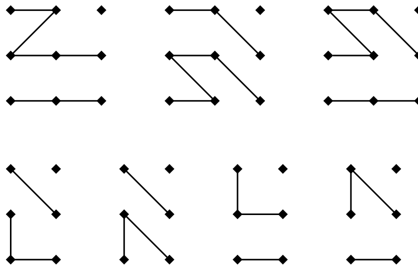


Figure 6.5: Recurrence relationships for state 6.

Let us now take a look at the possible starts, i.e. possible connections between the top and bottom part of the polygon.

Obviously states 1 and 2 as well as states 4, 5 and 6 have the same recurrence relationships, and their starting values are equal, too. Therefore it

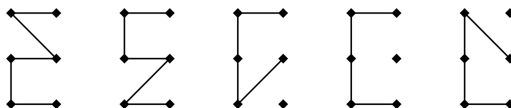


Figure 6.6: Possible starting configurations are states 2 (twice) and states 4, 5 and 6.

suffices to calculate states 1, 3 and 4:

$$
\begin{aligned}
s_{i+1}(1) &= s_i(4) \\
s_{i+1}(3) &= 2s_i(4) \\
s_{i+1}(4) &= 2s_i(1) + s_i(3) + 4s_i(4)
\end{aligned}
$$

In column $i = N - 1$ we have to close our polygon. Then the final result is $s_{N-1}(1) + s_{N-1}(2) + s_{N-1}(5)$, i.e. $2s_{N-1}(1) + s_{N-1}(4)$. See fig. 6.7 for reference.

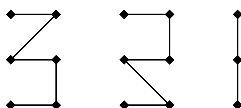


Figure 6.7: Possible closing configurations are states 1, 2 and 5.

Since the state change from column i to $i+1$ is a linear mapping, we can do it with a matrix multiplication. In order to do that, we combine states 1, 3 and 4 in a vector with three components, and rewrite the recurrence relationships above as a matrix:

$$
M = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 2 & 1 & 4 \end{pmatrix} \quad v_{start} = \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix} \quad v_{end} = \begin{pmatrix} 2 & 0 & 1 \end{pmatrix}
$$

Now, the problem is being solved by calculating the expression $v_{end}M^{N-1}v_{start}$.

We could simply do $N - 1$ matrix-vector multiplications to get the result, but this is definitely too slow for $N = 1{,}000{,}000{,}000$. As these multiplications are associative, we can instead calculate M^{N-1}. This can be done rather quickly, namely in $O(\log N)$ steps:

```
calculateMpowerE(M, e):
   Result := IdentityMatrix;
   while (e > 0):
      if (e mod 2 == 1):
         Result = Result * M;
      M = M * M;
      e = e div 2;
   return Result;
```

Some math or a different state representation allows you to reduce the matrix to a 2×2 matrix but this was not expected.

The last problem we have is that the numbers can get really big. Since only the number of possibilities modulo 10^9 needs to be calculated, we can do this modulo-operation on all intermediate results. They will always fit in a 64-bit integer, so there is no more trouble in that respect.

A different solution to get full score is to use a linear approach but precompute values for multiples of 1,000,000 and start the search from there.

Task 6: Sequence

7.1 Task description

We are given a sequence $a_1, ..., a_n$. We can manipulate this sequence using the operation $reduce(i)$, which replaces elements a_i and a_{i+1} with a single element $max(a_i, a_{i+1})$, resulting in a new shorter sequence. The cost of this operation is $max(a_i, a_{i+1})$. After $n-1$ operations *reduce*, we obtain a sequence of length 1. Our task is to compute the cost of the optimal reducing scheme, i.e. the sequence of *reduce* operations with minimal cost leading to a sequence of length 1.

Input

The input is read from a text file named `sequence.in`. The first line contains n ($1 \leq n \leq 1{,}000{,}000$), the length of the sequence. The following n lines contain one integer a_i, the elements of the sequence ($0 \leq a_i \leq 1{,}000{,}000{,}000$).

Output

The output is written into a text file named `sequence.out`. In the first and only line of the output print the minimal cost of reducing the sequence to a single element.

Example

sequence.in	sequence.out
3 1 2 3	5

Grading

In 30% of the test cases $n \leq 500$ holds.
In 50% of the test cases $n \leq 20{,}000$ holds.

7.2 Solution description

Simple observations

The definition of the *reduce* operation in the problem statement could be rephrased in the following way. We have n points on a line (corresponding to the elements of the sequence), so there are $n-1$ line segments connecting pairs of adjacent points. In the beginning all $n-1$ line segments are erased, so each point forms a single set. A single reduction is equivalent to drawing back one of these line segments, what results in joining the sets that contain the points corresponding to the ends of this line segment; the cost of this operation is equal to the maximal element of the resulting set. This interpretation of the *reduce* operation will simplify some of the following solutions' descriptions (as in this version the sequence never becomes shorter).

$O(n^3)$ dynamic programming solution

A *segment* of a sequence is its contiguous subsequence, i.e. $a_l, a_{l+1}, \ldots, a_r$. In the simplest solution for each segment of the sequence we count the minimal cost of reducing it into a single element (let us denote this value by $tab[l,r]$ for the segment $a_l, \ldots, a_r$). Obviously, for every $i \in \{1, \ldots, n\}$ we have $tab[i,i] = 0$. If $l < r$ then the last operation in the process of reducing this segment to a single element always costs $M = \max(a_l, \ldots, a_r)$; in our solution we can only choose the position of this reduction, which is in other words the position of the line segment drawn during this reduction. That is why for $l < r$ formula $tab[l,r] = \min(tab[l,i] + tab[i+1,r] + M \ : \ l \leq i < r)$ holds. Using this formula we can compute all values of array tab from the shortest to the longest segments. The total time complexity of this solution is $O(n^3)$, because there are n^2 fields of array tab and computing each of them requires $O(n)$ time. This solution — implemented in file `seqs1.cpp` — scores 30 points out of 100.

A greedy observation

To achieve a better time complexity of our solution we need to introduce some greediness. Let us assume for simplicity that $a_0 = a_{n+1} = \infty$. We claim that performing $n - 1$ repetitions of the following step:

```
find i such that a_{i-1} ≥ a_i ≤ a_{i+1}
if (a_{i-1} < a_{i+1}) then reduce(i − 1) else reduce(i)
```

gives us an optimal reduction scheme for our sequence.

Proof: Firstly we need to prove that in each of the $n - 1$ steps finding a required value of i is possible. In each step let us start with $j = 1$ and while the condition $a_{j-1} > a_j$ holds let us increase the value of j by one. This process will certainly terminate because $a_n < a_{n+1} = \infty$ (since $n \geq 1$). The value of $i = j - 1$ in the moment of termination is an example of an index that we were looking for.

Now let us find out why this reduction scheme is optimal. We will prove this by induction on the value of n. If $n = 1$ then this solution is obviously optimal with cost 0. If $n > 1$ then we need to prove that there exists an optimal reduction scheme of our sequence in which the reduction determined by our greedy step is done in the beginning. Let us assume that $a_{i-1} < a_{i+1}$ (the other case is analogical). Let S be any optimal reduction scheme of our sequence in which the reduction between a_{i-1} and a_i is not the first reduction (if no such scheme exists, then we are done). Let us consider two reductions: the one defined by the line segment between a_{i-1} and a_i (we will call it r_1) and the one between a_i and a_{i+1} (called r_2). If none of the reductions r_1 and r_2 is first in S then we can move the first of them that appears in S (let r_i be this reduction) to the beginning of S. After performing this operation, the cost of S does not increase. Why? Firstly, the cost of r_i can only decrease or stay the same (the sooner a reduction is performed, the lower is its cost). Secondly, performing this reduction earlier will not change the cost of any reductions performed in S before the second of reductions r_1 and r_2, because inserting a_i to a set containing either a_{i-1} or a_{i+1} does not change the maximal element of this set and only reductions involving this set could change their costs. On the other hand, after both r_1 and r_2 were performed, the costs of all following reductions will not change (all three elements a_{i-1}, a_i and a_{i+1} will be in a single set since then).

If $r_i = r_1$ then we are done now. In the opposite case we will prove that swapping r_2 (which is now in the beginning of S) and r_1 decreases the cost of S (what will give a contradiction with optimality of S). Let us notice that after

this swap the cost of all reductions apart from r_1 and r_2 in S is exactly the same as in the beginning (the argument is similar to the one from the previous fragment of the proof). The second of those reductions in both cases (before and after the swap) costs exactly the same, as its resulting set is the same. So the change of cost of S is $cost(r_1) - cost(r_2) = a_{i-1} - a_{i+1} < 0$, what gives us the desired contradiction.

Implementations of the greedy method

A straightforward implementation of the greedy approach (performing simply one step after another on a sequence represented as an array of integers) leads to an $O(n^2)$ solution. This solution — implemented in file `seqs2.cpp` — scores 50 points out of 100. There is a way of implementing this solution which has complexity $O(n^2)$ only in some special cases (files `seqs3.cpp` and `seqs4.cpp`). These solutions are very similar to the model solution (see below) and behave very well on random test data. They may score up to 70 out of 100 points.

However, an implementation of this algorithm with complexity $O(n)$ can also be obtained. In this solution the sequence will be analyzed from a_0 to a_{n+1}. Throughout the algorithm *a stack* containing a decreasing subsequence of a is maintained. In the beginning the stack contains only values a_0 and a_1. In each of the following steps (for $i = 2, 3, \ldots, n+1$), if a_i is smaller than the element on the top of the stack, then a_i is simply pushed on the stack, and in the opposite case all possible reductions are made. In other words, if the element from the top of the stack is not greater than a_i, then this element is reduced with smaller of the elements: a_i and the element second from the top of the stack. This operation pops the element from the top of the stack and the process continues. This loop stops at one of the conditions:

- if $i = n + 1$ and the stack contains only two elements, one of which must be $a_0 = \infty$ — this terminates the algorithm, the sequence was reduced to a single element,
- if a_i is smaller than the element on the top of the stack (in this case a_i is pushed on the stack) — this terminates the loop in the case $i < n + 1$; there will surely be such a moment in the loop for every $i < n + 1$, because $a_0 > a_i$.

Even though a single step of the algorithm (for a given i) may take $O(n)$ operations, the whole algorithm has complexity $O(n)$. This is true because

the number of iterations of the inner loop of the algorithm is bounded by the total number of elements that are pushed on the stack, and the latter is $O(n)$ because in every step only one element is pushed on the stack.

This solution is implemented in file `seq.cpp`.

Another nice way of solving the problem is summing up the maximum value of all neighbouring numbers in the sequence.

To see that this is true, consider the following reduction strategy: we pick the maximum element, reduce the sequence to the left of this element, then the sequence to the right of this element, and finally we use $reduce(0)$ twice to reduce the whole sequence to just the maximum element. We can now prove the correctness by induction over the length of the sequence. Suppose that our sequence has k elements and the above algorithm works for any sequence with less than k elements. We are now left with three cases:

1. The maximum element is the first element of the sequence. In this case, we first reduce the rest of the sequence, and the last step reduces the two remaining elments (with the same cost as the value of the first element). By induction hypothesis, the algorithm give the correct cost for reducing the rest of the sequence, and since the cost increases by the value of the first element, we can see that the algorithm still works in this case.

2. The maximum element is the last element of the sequence. This case is similar to the first case.

3. The maximum element, say m, is neither the first nor the last element. Let m_l denote the cost of reducing the left side and m_r denote the cost of reducing the right side. The cost of reducing the complete sequence is then equal to $m_r + m_l + 2 \cdot m$. We can see that this is the same value which the algorithm would return.

Finally, the induction hypothesis holds for single elements, because the algorithm always yields 0 in this case.