CEOI Poland 2011 Gdynia

Tasks and Solutions

Gdynia, 2011

Authors:

Mateusz Baranowski
Dawid Dąbrowski
Adam Karczmarz
Tomasz Kociumaka
Marcin Kubica
Tomasz Kulczyński
Jakub Łącki
Mirosław Michalski
Jakub Pachocki
Karol Pokorski
Jakub Radoszewski
Tomasz Waleń

Editors:

Marcin Kubica
Jakub Radoszewski

Proofreaders:

Tomasz Kociumaka
Jakub Łącki

Contents

Preface

Central European Olympiad in Informatics (CEOI) gathers the best teen-age programmers from several European countries. The 18-th CEOI was held at Pomeranian Science and Technology Park (PPNT) in Gdynia, Poland between July 7 and July 12, 2011. It was organized by Polish Olympiad in Informatics and Association *Talent*, together with: Polish Ministry of National Education, Foundation for Information Technology Development and Institute of Informatics, University of Warsaw. City of Gdynia and Polish Foundation for Computer Education have also supported organization of this event.

Nine countries took part in the competition: Croatia, Czech Republic, Germany, Hungary, Poland, Romania, Slovakia, Slovenia and Switzerland. All of the countries, except Poland, were represented by 4 contestants. Poland was represented by the first team and the second team, each consisting of 4 contestants. More information about the rules of the competition can be found at `http://ceoi2011.mimuw.edu.pl`. The medalists of CEOI 2011 are as follows:

Gold medalists:
Krzysztof Pszeniczny (Poland)
Jan Kanty Milczek (Poland)
Ivan Katanic (Croatia)
Piotr Bejda (Poland)
Matej Vecerik (Slovakia)

Silver medalists:
Wiktor Kuropatwa (Poland)
Krzysztof Leszczyński (Poland)
Matija Milisic (Croatia)
Adrian Budău (Romania)
Marian Hornak (Slovakia)
Klemen Kloboves (Slovenia)
Radu Ştefan Voroneanu (Romania)

Bronze medalists:
Alexandru George Cazacu (Romania)

4 *Preface*

> Matjaž Leonardis (Slovenia)
> Łukasz Jocz (Poland)
> Tobias Lenz (Germany)
> Vlad Alexandru Gavrilă (Romania)
> Marcin Smulewicz (Poland)
> Mateusz Kopeć (Poland)
> Zoltán Szenczi (Hungary)
> Gellért Weisz (Hungary)
> Martin Zikmund (Czech Republic)
> Marco Keller (Switzerland)
> Patrick Klitzke (Germany)
> Andrej Maris (Slovakia)

In parallel to the onsite competition, an online contest with exactly the same tasks and test data was held. It gathered about 200 competitors. The top 3 places in the contest were taken by Gennady Korotkevich (who got a perfect score), Erjin Zhou and Yiqin Lu.

The contest consisted of three sessions: a practice session followed by two competition sessions. During the practice session contestants had an occasion to become familiar with the software environment and to solve a practice task *Similarity*. During each of the competition sessions they had to solve three tasks within five hours of time: *Balloons*, *Matching* and *Treasure Hunt* during the first session, and *Hotel*, *Teams* and *Traffic* during the second one.

This booklet presents tasks from CEOI'2011 together with the discussion of their solutions. It was prepared with hope that it will help participants of various programming contests in their training. More materials, including test data used during the evaluation, can be found at `http://ceoi2011.mimuw.edu.pl`. Automated evaluation of solutions of the tasks from the contest is available at `http://main.edu.pl/en/archive/ceoi/2011`.

<div align="center">

Marcin Kubica | Jakub Radoszewski
chair of CEOI'2011 | *chair of CEOI'2011*
Steering Committee | *Scientific Committee*

</div>

Similarity

Byteman works in a computational biology research team in Gdynia. He is a computer scientist, though, and his work is mainly concentrated on designing algorithms related to strings, patterns, texts etc. His current assignment is to prepare a tool for computing the **similarity** *of a pattern and a text.*

Given a pattern and a text, one can align them in many different ways, so that each letter of the pattern has a corresponding letter in the text. Here we only consider alignments without holes, in which the pattern is matched against a consecutive part of the text of length equal to the length of the pattern. For any such alignment, one can count the positions where the letter of the pattern is the same as the corresponding letter of the text. The sum of such numbers is called the **similarity** *of the pattern and the text. The table below illustrates the computation of the similarity between an example pattern* `abaab` *and the text* `aababacab`.

		Matched letters:	
Text:	`aababacab`	`aababacab`	
Pattern:	`abaab`	`a..ab`	*(3)*
	`abaab`	`aba..`	*(3)*
	`abaab`	`...a.`	*(1)*
	`abaab`	`aba..`	*(3)*
	`abaab`	`...ab`	*(2)*
	Similarity:		*12*

Byteman has already managed to implement the graphical interface of the tool. Could you help him in writing the piece of software responsible for computing the similarity?

Input

The standard input consists of two lines. The first line contains a non-empty string composed of small English letters — the pattern. The second line contains a non-empty string composed of small English letters — the text. You may assume that the length of the pattern does not exceed the length of the text. The text contains no more than 2 000 000 letters.

Additionally, in test cases worth 30 points the length of the text does not exceed 5 000.

Output

The only line of the standard output should contain the similarity of the given pattern and the text.

Example

For the input data:	the correct result is:
abaab	12
aababacab	

Explanation. *The example above is the same as in the task description.*

Solution

In this task we are to compute the *similarity* of given patterns and texts efficiently. What is the similarity? Assume that p is the pattern and t is the text. Let n be the length of the pattern and m be the length of the text. There are $m - n + 1$ possible correct alignments of the pattern with the text, so that the pattern fits within the text. Assume that the first letter of the pattern is aligned with the i-th letter of the text. Then we compute the number of such $j = 1, 2, \ldots, n$ that $p[j] = t[i + j - 1]$. The similarity equals the sum of such numbers for each $i = 1, 2, \ldots, m - n + 1$.

Naive solution

The first approach is to apply the above definition directly. We test all possible values of i and for each of them consider all the possible values of j. If the letters are the same, we simply increment the result by one.

This solution runs in $O(n \cdot m)$ time and is worth 30 points, as stated in the task statement.

Model solution

To improve the time complexity of the solution, one should look at this problem from a different perspective. If we were to compute all the partial values, that is, the numbers of equal letters for each of the possible $m - n + 1$ alignments, this problem would be much harder to solve[1]. Since we are requested only to find the sum of such values, we can change the order of the sum.

Instead of counting corresponding equal letters for each alignment of the pattern and the text, we will count, for each position i of the text, the number of alignments in which the i-th letter of the text matches the corresponding letter of the pattern. If not for the condition that for each alignment the pattern must fit within the text, this value would be equal to the number of occurrences of the letter $t[i]$ in p. To take the aforementioned condition into account, for the i-th letter of the text, $i = 1, 2, \ldots, n - 1$, we may only consider the first i positions of the pattern. Similarly, for the $(m - i + 1)$-th letter of the text, $i = 1, 2, \ldots, n - 1$, we may only consider the last i positions of the pattern.

To implement this approach efficiently, we consider the letters of the text from left to right and use an auxiliary array a, in which for each letter of the alphabet $'\mathsf{a}'..'\mathsf{z}'$ we store the count of this letter at *significant* positions of the pattern. The whole algorithm can be written as follows:

1: $t['\mathsf{a}'..'\mathsf{z}'] := (0, 0, \ldots, 0)$;
2: *similarity* := 0;
3: **for** $i := 1$ **to** m **do begin**
4:    **if** $i \leqslant n$ **then** $a[p[i]] := a[p[i]] + 1$;
5:    *similarity* := *similarity* + $a[t[i]]$;
6:    **if** $i \geqslant m - n + 1$ **then** $a[p[n + i - m]] := a[p[n + i - m]] - 1$;
7: **end**
8: **return** *similarity*;

---

[1]One could use the so called fast Fourier transform (FFT) to solve this variant of the task in $O(m \log m \cdot A)$ time, where A is the size of the alphabet. We omit the details.

8   *Similarity*

This solution runs in $O(m)$ time. It could also be implemented less carefully in $O(m \log m)$ time or $O(m \cdot A)$ time, where A is the size of the alphabet (here $A = 26$).

Possible errors

One of the possible errors one could make in the above solution is to use a 32-bit integer type to store the result. Note that the result could be quite big (consider a pattern consisting of one million letters `a` and a text consisting of two million letters `a`), a 64-bit integer is required to store it.

Balloons

The organizers of CEOI 2011 are planning to hold a party with lots of balloons. There will be n balloons, all sphere-shaped and lying in a line on the floor.

The balloons are yet to be inflated, and each of them initially has zero radius. Additionally, the i-th balloon is permanently attached to the floor at coordinate x_i. They are going to be inflated sequentially, from left to right. When a balloon is inflated, its radius is increased continuously until it reaches the upper bound for the balloon, r_i, or the balloon touches one of the previously inflated balloons.

Fig. 1: The balloons from the example test, after being fully inflated.

The organizers would like to estimate how much air will be needed to inflate all the balloons. You are to find the final radius for each balloon.

10 Balloons

Input

The first line of the standard input contains one integer n ($1 \leqslant n \leqslant 200\,000$) — the number of balloons. The next n lines describe the balloons. The i-th of these lines contains two integers x_i and r_i ($0 \leqslant x_i \leqslant 10^9$, $1 \leqslant r_i \leqslant 10^9$). You may assume that the balloons are given in a strictly increasing order of the x coordinate.

In test data worth 40 points an additional inequality $n \leqslant 2\,000$ holds.

Output

Your program should output exactly n lines, with the i-th line containing exactly one number — the radius of the i-th balloon after inflating. Your answer will be accepted if it differs from the correct one by no more than 0.001 for each number in the output.

Example

For the input data:	the correct result is:
3	9.000
0 9	1.000
8 1	4.694
13 7	

Hint: *To output a* `long double` *with three decimal places in C/C++ you may use* `printf("%.3Lf\n", a);` *where* `a` *is the* `long double` *to be printed. In C++ with streams, you may use* `cout << fixed << setprecision(3);` *before printing with* `cout << a << "\n";` *(and please remember to include the* `iomanip` *header file). In Pascal, you may use* `writeln(a:0:3);` *. You are advised to use the* `long double` *type in C/C++ or the* `extended` *type in Pascal, this is due to the greater precision of these types. In particular, in every considered correct algorithm no rounding errors occur when using these types.*

Solution

The solution to this problem is to simply simulate the process of inflating the balloons from left to right. There are two concerns here:

the first one is being able to accurately determine how much can a balloon be inflated before it touches another given one, and the second one is the time complexity of our solution.

Assume we have an already inflated balloon at position x_1, with radius r_1, and we want to inflate another one, at position x. We need to determine the maximum radius r at which the second balloon does not intersect the first one (or, equivalently, the only radius at which the two balloons just touch). This can be done with a simple binary search, or, alternatively, we can derive an exact expression for r. As the center of the first balloon is at (x_1, r_1) and the center of the second one is at (x, r), the distance between the centers is $\sqrt{(x_1 - x)^2 + (r_1 - r)^2}$. The balloons touch if and only if this distance is equal to $r_1 + r$. Therefore, we must have:

$$
\begin{aligned}
(x_1 - x)^2 + (r_1 - r)^2 &= (r_1 + r)^2 \\
(x_1 - x)^2 &= 4r_1 r \\
r &= \frac{(x_1 - x)^2}{4r_1}
\end{aligned}
$$

With geometry accounted for, we can already implement a naive $O(n^2)$ time solution: when inflating a balloon, check all previously inflated ones to determine which one will be touched first (take the minimum of r over all possible (x_1, r_1) in (1)). The following observation is the key to arriving at the model $O(n)$ time solution:

Observation 1. When (x_1, r_1) and (x_2, r_2) are the centers of two previously inflated balloons, with $x_2 > x_1$ and $r_2 \geqslant r_1$, then no balloon to the right of x_2 may touch the first one before touching the second one.

Proof: As $x_2 > x_1$, for any $x > x_2$ we have $(x_1 - x)^2 > (x_2 - x)^2$. This, along with the condition $r_1 \leqslant r_2$, concludes that:

$$
\frac{(x_1 - x)^2}{4r_1} > \frac{(x_2 - x)^2}{4r_2}
$$

so the second balloon always implies a stronger bound on r in (1). ∎

Therefore, when inflating a balloon we only need to consider the previously inflated balloons that had no bigger balloon inflated after them. Keeping them on a stack enables us to use the following simple

algorithm to inflate a balloon at position x, with maximum allowable radius r:

```
 1: function InflateBalloon(x, r)
 2: begin
 3:     while not stack.empty() do begin
 4:         (x₁, r₁) := stack.top();
 5:         r := min(r, (x₁ − x)²/(4r₁));
 6:         if r ⩾ r₁ then stack.pop()
 7:         else break;
 8:     end
 9:     stack.push((x, r));
10:     return r;
11: end
```

As the balloons are given in an increasing order by the x coordinate, we can simply invoke this function for every subsequent balloon while reading the input and print its return value, which is the final radius for the given balloon. As every balloon will be pushed to the stack exactly once, the total amortized running time of this function is $O(1)$, and $O(n)$ for the whole program.

Matching

As a part of a new advertising campaign, a big company in Gdynia wants to put its logo somewhere in the city. The company is going to spend the whole advertising budget for this year on the logo, so it has to be really huge. One of the managers decided to use whole buildings as parts of the logo.

The logo consists of n vertical stripes of different heights. The stripes are numbered from 1 to n from left to right. The logo is described by a permutation $(s_1, s_2, \ldots, s_n)$ of numbers $1, 2, \ldots, n$. The stripe number s_1 is the shortest one, the stripe number s_2 is the second shortest etc., finally the stripe s_n is the tallest one. The actual heights of the stripes do not really matter.

There are m buildings along the main street in Gdynia. To your surprise, the heights of the buildings are distinct. The problem is to find all positions where the logo matches the buildings.

Help the company and find all contiguous parts of the sequence of buildings which match the logo. A contiguous sequence of buildings matches the logo if the building number s_1 within this sequence is the shortest one, the building number s_2 is the second shortest, etc. For example a sequence of buildings of heights 5, 10, 4 matches a logo described by a permutation $(3, 1, 2)$, since the building number 3 (of height 4) is the shortest one, the building number 1 is the second shortest and the building number 2 is the tallest.

Input

The first line of the standard input contains two integers n and m ($2 \leqslant n \leqslant m \leqslant 1\,000\,000$). The second line contains n integers s_i, forming a permutation of the numbers $1, 2, \ldots, n$. That is, $1 \leqslant s_i \leqslant n$ and $s_i \neq s_j$ for $i \neq j$. The third line contains m integers h_i — the heights of the buildings ($1 \leqslant h_i \leqslant 10^9$ for $1 \leqslant i \leqslant m$). All the numbers h_i are different. In each line the integers are separated by single spaces.

Additionally, in test cases worth at least 35 points, $n \leqslant 5\,000$ and $m \leqslant 20\,000$, whereas in test cases worth at least 60 points, $n \leqslant 50\,000$ and $m \leqslant 200\,000$.

Output

The first line of the standard output should contain an integer k, the number of matches. The second line should contain k integers — 1-based indices of buildings which correspond to the stripe number 1 from the logo in a proper match. The numbers should be listed in an increasing order and separated by single spaces. If k = 0, your program should print an empty second line.

Example

For the input data:
```
5 10
2 1 5 3 4
5 6 3 8 12 7 1 10 11 9
```
the correct result is:
```
2
2 6
```
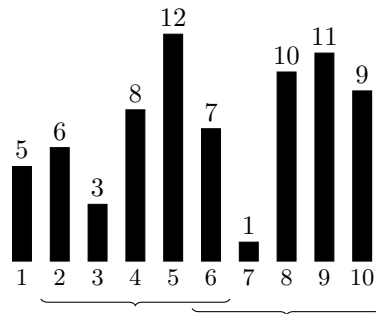

Explanation: *Both the sequences 6, 3, 8, 12, 7 and 7, 1, 10, 11, 9 match the logo described by the permutation (2, 1, 5, 3, 4). In particular, in the first sequence the building number 2 (of height 3) is the shortest one, the building number 1 (of height 6) is the second shortest, the building number 5 (of height 7) is the third shortest, and so on.*

Solution

The problem described in the task statement is a variant of the string matching problem. We are given two strings: a *pattern* $p[1..n]$ and a *text* $t[1..m]$. The task is to find all positions $1 \leqslant j \leqslant m - n + 1$, such that the pattern matches the text at position j. In our problem, however, both the pattern and the text are sequences of distinct integers.

Moreover, the pattern $p[1..n]$ is not given explicitly. Instead we are given a sequence (s_i), that describes $p[1..n]$: s_1 is the index of the smallest element in $p[1..n]$, s_2 is the second smallest, and so on. Any

$p[1..n]$ which is consistent with the input is equivalent, so from now on we assume that $p[1..n]$ consists of distinct integers from the range $[1, n]$. This representation can be easily computed from (s_i) in $O(n)$ time.

In the most typical pattern matching problem, the pattern matches the text at position j if p is equal to $t[j..j+n-1]$. This problem gives a different definition of a match, in which the pattern matches the text if it is *isomorphic* to $t[j..j+n-1]$. We call two sequences a and b of length k isomorphic if:

$$a[i] < a[j] \Leftrightarrow b[i] < b[j] \text{ for all } 1 \leqslant i, j \leqslant k.$$

To shorten the notation, we write $a \approx b$ to denote that a is isomorphic to b. In the following we use two simple, but important observations. Given three sequences a, b and c of length k:

- If $a \approx b$, then all their corresponding subwords are isomorphic, that is $a[x..y] \approx b[x..y]$ for $(1 \leqslant x \leqslant y \leqslant k)$.

- If $a \approx b$ and $b \approx c$, then $a \approx c$.

To solve the problem, we adapt the Knuth-Morris-Pratt (KMP) string matching algorithm to our needs. In the following we assume that the reader is familiar with it.

The failure function

We define a *border* of a sequence $a[1..k]$ as a suffix of $a[1..k]$ of length t which is isomorphic to $a[1..t]$. As in the KMP algorithm, we begin by computing a *failure function* f. For each $1 \leqslant i \leqslant n$, we want to know what is the longest border of $p[1..i]$ (excluding the trivial border of length i):

$$f[i] = \max_{0 \leqslant k < i} p[1..k] \approx p[i-k+1..i].$$

Additionally, we set $f[0] = 0$.

We compute the values $f[i]$ one by one, for increasing values of i. The longest border of $p[1..i]$ consists of some border of $p[1..i-1]$ and the letter $p[i]$. Hence, we iterate through all the borders of $p[1..i]$, starting from the longest one, and for each border we check if it can be extended with $p[i]$ to form a border of $p[1..i]$.

To iterate through all the borders we use the following lemma; its proof is given later.

Lemma 1. The lengths of all the borders of $p[1..i]$ are given by $f[i], f[f[i]], f[f[f[i]]], \ldots$

Note that since $0 \leqslant f[i] < i$, from some point the above sequence consists of zeroes only.

It remains to show how to check whether some border of $p[1..i-1]$ extended with $p[i]$ forms a border of $p[1..i]$. In other words, given two sequences $a[1..k]$ and $b[1..k]$ (the former corresponds to a prefix of the pattern, while the latter to its subword), we want to check if they are isomorphic, knowing that $a[1..k-1] \approx b[1..k-1]$. Observe that it suffices to check whether:

$$a[q] < a[k] \Leftrightarrow b[q] < b[k] \text{ for all } 1 \leqslant q < k.$$

This can be rewritten in the following way (recall that all elements in each of the sequences a, b are distinct):

Property 1. For some $1 \leqslant r \leqslant k$, $a[k]$ is the r-th biggest number among $a[1..k]$ and $b[k]$ is the r-th biggest number among $b[1..k]$.

We now describe a way of checking if the above holds.

Let $a[u]$ be the greatest integer among $a[1..k-1]$ which is smaller than $a[k]$ and $a[w]$ be the smallest integer among $a[1..k-1]$ which is greater than $a[k]$. Here we assume that both these elements exist, the other cases are similar. By definition, $a[u] < a[k] < a[w]$. We claim that checking if $b[u] < b[k] < b[w]$ is equivalent to property 1. This follows from the fact that $a[1..k-1]$ is isomorphic to $b[1..k-1]$ so the number of elements smaller than $a[u]$ in $a[1..k-1]$ is the same as the number of elements smaller than $b[u]$ in $b[1..k-1]$. Similarly, the number of elements greater than $a[w]$ in $a[1..k-1]$ is the same as the number of elements greater than $b[w]$ in $b[1..k-1]$. Hence, this check is indeed equivalent to property 1.

We now show how to compute the indices u and w. For each $1 \leqslant i \leqslant n$, we need to find in $p[1..i]$ the element which is the greatest among the elements less than $p[i]$, call its index $g[i]$. We also need to know the index of the smallest number greater than $p[i]$, call its index $h[i]$ (this is a symmetric problem).

Recall than $p[1..n]$ consists of distinct integers from the range $[1, n]$. We compute $g[i]$ for decreasing values of i. We maintain a doubly linked list of all elements of $p[1..i]$ in an increasing order. Initially this is just a list of all numbers from 1 to n. In every step one element is deleted. With each element we associate its position in $p[1..n]$ and for each position in $p[1..n]$ we store a link to its corresponding element in the list. This list allows us for each i to obtain the two elements which are closest to $p[i]$ in constant time — they are simply the neighbors of $p[i]$ in the list after $n - i$ elements of the list are removed.

This gives an algorithm for computing the failure function. After precomputation of the array $g[1..n]$ and its symmetric counterpart $h[1..n]$, which runs in $O(n)$ time, the algorithm is completely analogous to the computation of failure function in KMP. It follows that the overall running time is $O(n)$.

Finding the matches

The main operation performed in the matching phase of KMP is, roughly speaking, the following:

Given a partial match of the pattern in the text, try to extend it with one character. If this is not possible, use the failure function to get a shorter partial match, further to the right in the text.

This is what we also do in our problem. Using the ideas described above, we can check if a partial match can be extended with one character in constant time. The proof of correctness is simple, the general idea is that if we skip some correct match (i.e., we move the partial match too much to the right with the failure function), we immediately get a contradiction with the definition of the failure function.

Again, since the matching phase is a slightly modified KMP, it runs in $O(n+m)$ time. Hence, the whole algorithm requires only linear time.

Proof of lemma 1: We show that if $p[1..i]$ has a border of length t, then the longest border which is shorter than t has length $f[t]$. If $f[t] = 0$ then the border of length t is the shortest one. From the definition of a border, we have $p[1..t] \approx p[i - t + 1..i]$.

We first show that $p[1..i]$ has a border of length $f[t]$. First, observe that from the isomorphism $p[1..t] \approx p[i-t+1..i]$ it follows that for any $1 \leqslant s < t$, $p[1..t]$ has a border of length s if and only if $p[i-t+1..i]$ has a border of length s. Moreover, any border of $p[1..t]$ is isomorphic to a border of $p[i-t+1..i]$. Hence, the border of $p[1..t]$ of length $f[t]$ is isomorphic to a border of $p[i-t+1..i]$ of length $f[t]$. This gives that $p[1..f[t]] \approx p[i-f[t]+1..i]$.

To complete the proof, we have to show that there is no border of length strictly between $f[t]$ and t. In order to do that, we show that any border of such length is also a border of $p[1..t]$, hence it would contradict the definition of $f[t]$. Let $f[t] < u < t$ and let $p[1..u] \approx p[i-u+1..i]$, that is, there is a border of $p[1..i]$ of length u. This means that a prefix of length u of $p[1..t]$ is isomorphic to a suffix of $p[i-t+1..t]$ of equal length. We already know that $p[1..t]$ and $p[i-t+1..t]$ are isomorphic, so the suffix of $p[i-t+1..t]$ of length u is isomorphic to a suffix of $p[1..t]$ of length u. As a result a suffix of $p[1..t]$ of length u is isomorphic to $p[1..u]$, a contradiction. ∎

Tomasz Kulczyński
Task idea and formulation

Jakub Radoszewski
Analysis

Treasure Hunt

Ahoy! Have you ever heard about pirates and their treasures? Bytie has found an old bottle while having a walk along the beach in Gdynia. The letter inside gives instructions on how to find a hidden treasure, but it is quite difficult to decipher. One thing Bytie knows for sure is that he needs to find two special points in the park nearby and the treasure will be in the middle of the path connecting them.

There are several trails in the park. Apart from that, the forest in the park is very dense, so only positions on the trails are reachable for human beings. The structure of the trails has an interesting property: for any two points lying on the trails there is a unique path connecting them. The path may lead along multiple trails, but it never visits any point more than once.

Bytie asked his friends for help in exploring the park. They will start the treasure hunt in some point of the park, located on one of the trails. They will explore the park in phases. In each phase, one of the friends chooses one point that was already explored and walks a number of steps from that point along a trail, visiting only points which were never reached by any of the friends before.

During the exploration, Bytie will be analysing the structure of the park carefully. From time to time, Bytie may guess the two special points which determine the location of the treasure. For each such guess, he wants to know the point located in the middle of the only path connecting them. Your task is to help Bytie in determining these middle points.

Communication

You should write a library which interacts with the grading program. The library should contain the following three functions which will be called by the grader (and any more functions if you like):

- init — *this function will be called exactly once, in the beginning of the execution. It is for you to initialize your data structures etc.*

  - *C/C++:* `void init();`
  - *Pascal:* `procedure init();`

> When this function is called, you should assume that there is exactly one point already explored in the park, marked with the number 1.

- `path` — *stating that one of the friends explored a new path in the park. This function is for you to build your data structures representing trails.*

  - *C/C++:* `void path(int a, int s);`
  - *Pascal:* `procedure path(a, s: longint);`

  The path starts in the point number a (which was already explored) and takes s steps along a trail (s > 0). After each step, the current point is assigned a new number: the smallest positive integer not yet used for this purpose. This function will be called at least once.

- `dig` — *asking where to dig for the treasure.*

  - *C/C++:* `int dig(int a, int b);`
  - *Pascal:* `function dig(a, b: longint): longint;`

  It should return the number assigned to the point located in the middle of the path connecting the points marked with the numbers a and b. You can assume that the points a and b have already been explored and that a ≠ b. If the middle of the path is not a point with an assigned number (because the path has an odd length), the function should return the number assigned to one of the two middle points of the path — the one that is closer to a (see also the example on the next page). This function will be called at least once.

Your library **may not** *read anything, neither from the standard input nor from any file, and* **may not** *write anything, neither to the standard output nor to any file. Your library* **may** *write to the standard error stream/file (*`stderr`*). You should be aware, however, that this consumes time during the execution of the program.*

If you are writing in C/C++, your library **may not** *contain the* `main` *function. If you are using Pascal, you have to provide a unit (see the sample programs on your disk).*

Compilation

Your library — `tre.c`*,* `tre.cpp`*, or* `tre.pas` *— will be compiled with the grader using the following commands:*

- *C:* `gcc -O2 -static -lm tre.c tregrader.c -o tre`

- *C++:* `g++ -O2 -static -lm tre.cpp tregrader.cpp -o tre`

- *Pascal:*
  ```
  ppc386 -O2 -XS -Xt tre.pas
  ppc386 -O2 -XS -Xt tregrader.pas
  mv tregrader tre
  ```

Sample execution

The table below shows a sample sequence of calls to the functions and the correct results of the `dig` *calls. The structure of the trails in the park corresponding to this example run is shown in the figure.*
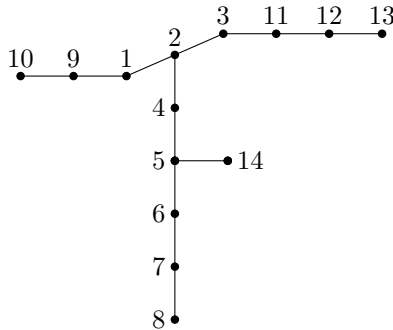
Function call	Result	New points added
`init();`		1
`path(1, 2);`		2, 3
`dig(1, 3);`	2	
`path(2, 5);`		4, 5, 6, 7, 8
`dig(7, 3);`	5	
`dig(3, 7);`	4	
`path(1, 2);`		9, 10
`path(3, 3);`		11, 12, 13
`dig(10, 11);`	1	
`path(5, 1);`		14
`dig(14, 8);`	6	
`dig(2, 4);`	2	

Constraints

- *There will be at most $400\,000$ calls to the functions (*init*, `path`, and `dig`) and at most $1\,000\,000\,000$ points explored by Bytie's friends.*

- *In test cases worth 50 points in total, there will be no more than $400\,000$ points explored.*

- *In test cases worth 20 points in total, there will be no more than $5\,000$ points explored and at most $5\,000$ calls to the functions `init`, `path`, and `dig`.*

Experimentation

To let you experiment with your solution, you are given a sample grading program in the file:

> `tregrader.c`, `tregrader.cpp`, *and* `tregrader.pas`

located in the directory `/home/zawodnik/tre/` on your machine. To experiment with the grading program, you should put your solution in the file:

> `tre.c`, `tre.cpp`, *or* `tre.pas`

in the respective directory (`c`, `cpp`, or `pas`). In the beginning of the contest in each of these files you can find a sample incorrect solution to the problem. You can compile your solution using the command:

> `make tre`

which works exactly as described in the Compilation section, provided that you compile your program in the respective directory. The C/C++ compilation requires the file `treinc.h`, which is also located in the respective directories.

The resulting binary reads a list of function names and arguments from the standard input, calls the corresponding functions from your solution and writes the results of the calls of the `dig` function to the standard output. The list of functions in the input should be formatted as follows: the first line contains the number of instructions, q. Then q lines follow, each containing a character `i`, `p`, or `d` followed by two non-negative integers. The character determines which function should be called: `i` for `init`, `p` for `path`, and `d` for `dig`. The integers denote the arguments of the function: a and s for `path`, a

and b for `dig`. If the character is `i`, both integers are equal to 0. Note that the sample grader **does not** check if the input is formatted correctly or if it satisfies the requirements listed in the Communication & Constraints sections.

You are given the file `tre0.in` which represents the sample execution described above:

```
12
i 0 0
p 1 2
d 1 3
p 2 5
d 7 3
d 3 7
p 1 2
p 3 3
d 10 11
p 5 1
d 14 8
d 2 4
```

To read the data from this file, use the following command:

```
./tre < tre0.in
```

The results of the `dig` calls returned by your solution will be written to the standard output. The correct result for the aforementioned input, written also in the file `tre0.out`, is:

```
2
5
4
1
6
2
```

You can verify if the output of your solution to the sample test case is correct by submitting your solution to the grading system.

Solution

The structure of the trails in the park can be represented as an undirected graph. The vertices of the graph represent the points which were assigned a number by one of Bytie's friends, and the edges show the subsequent steps made by the friends. Each pair of vertices is connected by exactly one simple path, so the graph is a *tree*, that is, a connected acyclic graph. Let us denote by N the total number of vertices (nodes) of the tree.

The structure of the tree is unfolded in phases, each of which reveals a path composed of a number of new nodes (the `path` operation). We are to answer multiple queries related to the tree. In each query, our task is to find the middle point of the path connecting a given pair of nodes a, b of the tree, or one of the middle points, if the path has an odd length (the `dig` operation). We denote the resulting node by $dig(a, b)$. Let n be the total number of calls to the functions.

Simple $O(Nn)$ time solutions

The most straightforward solution can be obtained by using one of the classical graph searching algorithms — Breadth First Search or Depth First Search — to find the path connecting the nodes a and b. Afterwards we simply return the middle point of the path. Each call to the searching algorithm takes $O(N)$ time, so the whole algorithm has $O(Nn)$ time complexity.

There is also another $O(Nn)$ time solution. Let us root the tree in the node number 1. Now, for every node v we can define its *depth*, equal to the distance between v and the root, and its *parent*, being the immediate predecessor of v in the path from v to the root. To simplify the algorithms, we assume that the root is the parent of itself. Using these notions, one can find the path connecting the nodes a and b in two steps. In the first step we equalize the depths of the nodes, advancing the lower of the nodes to the depth of the upper one. The second step consists in traversing the tree upwards, simultaneously for both nodes, until the two nodes meet. The resulting node is called the *lowest common ancestor* of a and b, and is denoted by $\text{LCA}(a, b)$. Now we can combine the two parts of the path, corresponding to the nodes a and b, and find the node $dig(a, b)$ as in the first algorithm.

As an example, consider the rooted tree obtained from the example in the problem statement (fig. 1) and assume we process the `dig` query for the nodes 12 and 8. We have $depth[12] = 4$ and $depth[8] = 6$, so we start by going two levels upwards from 8, arriving at the node 6 of depth 4 (indicated in the figure by an arrow). Finally, we ascend level by level from the nodes 12 and 6 simultaneously until we reach the node 2 of depth 1. Here we have $LCA(12, 8) = 2$. Note that the answer to the query is: $dig(12, 8) = 4$.
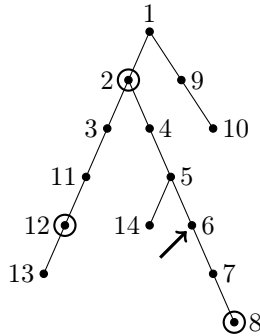

Fig. 1: The rooted tree obtained from the example in the problem statement. Here $LCA(12, 8) = 2$ and $dig(12, 8) = 4$.

This solution could be expected to work faster than the previous one, since the total number of steps in a single query, equal to $depth[a] + depth[b] - 2 \cdot depth[LCA(a, b)]$, might be $o(N)$. Nevertheless, the worst-case time complexity is still $O(Nn)$.

Both $O(Nn)$ time solutions score about 20 points, as the partial scoring conditions may suggest.

$O((N + n) \log N)$ time solution — faster LCA

We obtain a solution with a better time complexity by computing the lowest common ancestors of pairs of nodes more efficiently. Knowing $LCA(a, b)$, one can compute $dig(a, b)$ using a single auxiliary function $go\text{-}up(a, h)$, which advances h levels starting from the node a. More precisely, comparing the depths of the nodes a, b and $LCA(a, b)$, one

can easily find out if the node $dig(a, b)$ lays within the path from a to $\text{LCA}(a, b)$ or within the path from b to $\text{LCA}(a, b)$, and then advance from the appropriate node by the number of levels equal to the half of the length of the path connecting a and b. See the following pseudocode for details.

```
 1: function  dig(a, b)
 2: begin
 3:    v := LCA(a, b);
 4:    len := depth[a] + depth[b] − 2 · depth[v];
 5:    pos := ⌊len/2⌋;
 6:    if pos ⩽ depth[a] − depth[v] then return go-up(a, pos)
 7:    else return go-up(b, len − pos);
 8: end
```

Let us first figure out how to implement the function *go-up* efficiently. The main idea is switching the numeration system to binary — we develop an algorithm similar to computing powers by repeated squaring. For each node of the tree we will not only store its parent, but also its ancestors lying $2, 4, 8 \ldots$ levels above it. We use an array $anc[1 .. N][0 .. m]$ (where $m = \lceil \log N \rceil$), in which $anc[v][i]$ is the ancestor of v located 2^i levels above v, or $anc[v][i] = 1$ if no such ancestor exists. Note that this is consistent with a convention that $parent[1] = 1$. The values $anc[v][*]$ are computed when the node v is created in the `path` call; below by $curr$ we denote the number of the last created node of the tree.

```
 1: function  path(a, s)
 2: begin
 3:    prev := a;
 4:    for j := 0 to s − 1 do begin
 5:       v := curr + 1;
 6:       depth[v] := depth[prev] + 1;   parent[v] := prev;
 7:       anc[v][0] := parent[v];
 8:       for i := 1 to m do anc[a][i] := anc[anc[a][i − 1]][i − 1];
 9:       prev := curr;   curr := v;
10:    end
11: end
```

Using the array *anc*, one can easily implement the *go-up*(a, h) function. If the i-th bit in the binary representation of h is equal to 1, we advance 2^i levels upwards from a. Clearly, the implementation below works in $O(m) = O(\log N)$ time.

```
1: function go-up(a, h)
2: begin
3:    for i := 0 to m do begin
4:       if h mod 2 = 1 then
5:          a := anc[a][i];
6:       h := ⌊h/2⌋;
7:    end
8: end
```

Alternatively, this could be written as follows (this approach is used later on, in the following section).

```
1:  function go-up2(a, h)
2:  begin
3:     i := 0;
4:     while 2^(i+1) ≤ h do i := i + 1;
5:     while i ≥ 0 do begin
6:        if 2^i ≤ h then begin
7:           a := anc[a][i];
8:           h := h − 2^i;
9:        end
10:       i := i − 1;
11:    end
12: end
```

Now we can finally proceed to computing the LCAs of nodes. Note that the first step of finding $\text{LCA}(a, b)$ — equalizing the depths of a and b — can be implemented by making just a single call to the *go-up* function. The second step is also similar. If after the first step the nodes are already equal, we do nothing. Otherwise we start by finding the smallest integer value of i such that advancing 2^i levels from a and b we get to the same node. This is an upper bound for the number of levels we need to ascend to reach $\text{LCA}(a, b)$. Now we are going to find the level of the nodes located just one level below $\text{LCA}(a, b)$. For this,

we try to go 2^{i-1}, 2^{i-2}, ..., 2^0 levels up from a and b, if only after such a step we arrive at different nodes. We end up with a pair of nodes, whose LCA is their parent.

```
1: function LCA(a, b)
2: begin
3:    { First step }
4:    if depth[a] < depth[b] then a := go-up(a, depth[a] − depth[b])
5:    else b := go-up(b, depth[b] − depth[a]);
6:    if a = b then return a;
7:    { Second step }
8:    i := 0;
9:    while anc[a][i] ≠ anc[b][i] do i := i + 1;
10:   for j := i − 1 downto 0 do
11:      if anc[a][j] ≠ anc[b][j] then begin
12:         a := anc[a][j];
13:         b := anc[b][j];
14:      end
15:   return parent[a];
16: end
```

Now, let us analyse the complexity of the solution. The *anc* array requires $O(Nm) = O(N \log N)$ space and all other arrays have their sizes linear in terms of N. Hence the memory complexity increases to $O(N \log N)$ (compared to the previous approaches). As for the time complexity, each of the functions *go-up*, LCA and *dig* performs a constant number of binary ascents, each in $O(\log N)$ time, and the *path* function computes the whole array *anc*, which requires $O(N \log N)$ time in total. The time complexity of the solution is, therefore, $O((N + n) \log N)$. As one could expect having read the problem statements, such a solution scores 50 points.

$O(n \log N \log n)$ time solution — using the compacted tree

It is clear that solutions with time complexity $\Omega(N)$ have no chances for a perfect score. We need to try to solve the problem dealing with

compacted edges. Now we will show how to adapt the previous solution to such conditions.

A standard approach to such problems is dividing all the nodes of the tree into *explicit* and *implicit*. The nodes having more than one child or no children (leaves), together with the root of the tree, belong to the former class, while all other to the latter one. Indeed, this sounds like a decent idea: e.g., for any pair of nodes a and b, $\mathrm{LCA}(a, b)$ is either one of the nodes a, b or is an explicit node of the tree. On the other hand, a problem of detecting which nodes are the explicit ones arises. If we were given the `path` and `dig` operations in an offline manner, that is, if we knew all the operations in advance, we could easily detect the explicit nodes. This is not the case in this problem, though — there is no way of predicting which nodes will become explicit.

Instead of trying to foresee the future, we can modify the definitions of explicit and implicit nodes. A natural choice could be to define as explicit nodes the first and the last node created within each `path` call (and the node number 1 as well), see fig. 2. Now, for each node we know whether it is explicit at the very moment it is revealed. Furthermore, one could see the structure of the connections between the explicit nodes as a tree: that is, for every explicit node (different from the root) one can choose its parent as the first explicit node located in the path connecting it with the root. For example, for the tree in fig. 2 we would have $parent(14) = parent(8) = 4$ and $parent(4) = 2$. One can also define the depths of the explicit nodes, in two ways: either as the *real-depths*, that is, the depths in the uncompacted tree, or as the *depths* in the tree formed only by the explicit nodes. That is, $depth(v)$ equals the number of explicit nodes in the path from v to the root (not counting v itself). This immediately lets us compute the *anc* values for the explicit nodes; this time $anc(v, i)$ represents both the reference to the node located 2^i explicit levels above v and the difference of real-depths of those two nodes. The only problem is that, compared to the original definition of the explicit nodes, we loose the nice property of the LCA operation: this time $\mathrm{LCA}(a, b)$ can be an implicit node different from a and b, e.g., $\mathrm{LCA}(14, 8) = 5$ is an implicit node in the figure.

In the following solution we cope with this problem by simplifying the way the LCA values are computed. Note that regardless of the method used here, we still need to implement the function $go\text{-}up(a, h)$, which advances h levels upwards (with respect to the real-depths) from
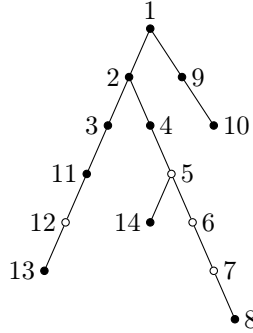
Fig. 2:   The tree from figure 1 with explicit nodes represented as filled
circles and implicit nodes as empty circles.

an explicit or implicit node a. It appears that using this method a
number of times, one can implement the $\mathrm{LCA}(a, b)$ function easier than
previously.

Assume that a is located deeper in the tree than b. We start by
advancing $real\text{-}depth(a) - real\text{-}depth(b)$ levels from a, which is done using
a single *go-up* call as previously. We arrive at an (explicit or implicit)
node a'. Assume that $a' \neq b$, otherwise we are already done. Now we
can find $\mathrm{LCA}(a', b) = \mathrm{LCA}(a, b)$ using binary search. Indeed, we know
that $\mathrm{LCA}(a', b)$ is located at least one and at most $real\text{-}depth(b)$ nodes
above a' and b, and that all the ancestors of a' and b' of real-depth not
exceeding $real\text{-}depth(\mathrm{LCA}(a', b))$ are same ancestors.

```
1: function LCA2(a, b)
2: begin
3:    { First step }
4:    if real-depth(a) < real-depth(b) then swap(a, b);
5:    a := go-up(a, real-depth(a) − real-depth(b));
6:    if a = b then return a;
7:    { Second step }
8:    low := 1;   high := real-depth(b);
9:    while low < high do begin
10:      med := ⌊(low + high)/2⌋;
11:      if go-up(a, med) = go-up(b, med) then high := med
12:      else low := med + 1;
```

13:     **end**
14:     **return** *go-up*(*a, low*);
15: **end**

Now it all reduces to the implementation of the *go-up*(*a, h*) function. It should work as follows:

1. Advance from *a* to its closest explicit ancestor. Let *d* be the distance traversed. If $d \geqslant h$, we have the result. Otherwise, we need to advance $h - d$ levels from the resulting explicit node a'.

2. Perform a binary ascent from a', similar to the one in the pseudocode of the *go-up2* function. More precisely, we find the lowest explicit ancestor of a' of real-depth at least *real-depth*(a')$-(h-d)$. Instead of comparing 2^i or 2^{i+1} with h as in the original pseudocode, we compare with h the differences of depths of a' and $anc(a', i)$ ($anc(a', i+1)$) respectively.

3. Let a'' be the resulting explicit node. If *real-depth*(a'') is equal to *real-depth*(a') $- (h - d)$ then we are done, we have found the requested node.

4. Otherwise we need to advance from the node a'' by exactly *real-depth*(a'')$-$ *real-depth*(a')$+(h-d)$ levels to find the resulting implicit node. We have two cases: either a'' is the lower end of a compacted edge, and then the implicit node is located within the same edge, or a'' is the upper end of a compacted edge. In the latter case we need to find the implicit node it is attached to in the tree and go up along the edge containing this implicit node.

To be able to implement all the aforementioned steps, we need to decide on a convenient representation of implicit and explicit nodes of the tree. The explicit nodes will be assigned numbers as they are added to the tree, starting from the root with the number 1 (note that we omit implicit nodes in this numeration). We call these the *new numbers*, in contrast with the *old numbers* assigned as in the problem statement. This enables storing the values *parent*, *depth*, *real-depth* and *anc* for the explicit nodes in arrays indexed by their new numbers. Now an implicit node can be uniquely represented as a pair: (the new number of its closest explicit ancestor, the distance from the ancestor), note that

the closest explicit ancestor is always located in the same compacted edge. To be able to map the old numbers into the new numbers, we keep an array $expl[1..2n]$ containing the old numbers of all explicit nodes. This array is also indexed by the new numbers of the explicit nodes in a natural way. To obtain the representation of an implicit node v, it suffices to find the largest element in the array $expl$ which does not exceed the old number of v; this element corresponds to the lowest explicit ancestor of v. We can perform this operation in $O(\log n)$ time using binary search (in C++ one can also use the `lower_bound` routine from the STL). Finally, we need an additional array $attached[1..2n]$ to store the representations of (implicit of explicit) nodes to which each new explicit node is attached (this could be required in the fourth step of the above algorithm).

This way we have concluded all the technical considerations. The reader is encouraged to check that now the *go-up* function can be implemented in $O(\log n)$ time.

We are ready to summarize the solution and find its complexity. We use several arrays of size at most $2n$: *parent*, *depth*, *real-depth*, *expl* and *attached*, and the array *anc* of size $O(n \log n)$. All those arrays are computed during the `path` calls. Each `dig` call is performed as in the pseudocode in the previous solution, just with *real-depths* instead of *depths*. It performs one *go-up* call and one LCA call, it also requires finding representations (the new numbers) of the nodes being its arguments, which is done in $O(\log n)$ time. Each LCA call performs $O(\log N)$ calls to the *go-up* function, and the latter function works in $O(\log n)$ time. In conclusion, both the LCA function and the *dig* function work in $O(\log N \log n)$ time. The whole solution works in $O(n \log N \log n)$ time and $O(n \log n)$ space. Depending on the implementation, it scores between 80 and 100 points.

One way of improving this solution by a constant factor is to make the lowest nodes created by each *path* call implicit. Indeed, it is easy to see that is was not necessary for them to be explicit in the above solution and we made them so only by an analogy with the classical notion of explicit nodes.

$O(n \log n)$ time solution — the model solution

The model solution for this task has both time and memory complexity $O(n \log n)$. This slight improvement over the previous solution is

obtained by coming back to the original idea of implementing LCA.

Let us recall the first implementation of the LCA(a, b) function, in which we first equalize the depths of the nodes a and b, and then climb up the tree using the array *anc*. We can apply the same procedure in the case of a compacted tree, if only we modify it *really carefully*. In the first step we start by moving from a and b to their explicit ancestors. If those are equal, then one of the nodes a, b is the LCA and we can easily deal with this case. Otherwise, we can equalize the *depths* of the explicit nodes. Note that we do mean the *depths* of the nodes, not the real-depths, so a *different go-up* procedure is required than in the previous section. As the depths of the nodes are equal (note that the real-depths might be different here!), we perform a binary climb using the *anc* array, finishing in a pair of explicit nodes lying just below their LCA. Unfortunately, the LCA of the two explicit nodes may be explicit or implicit, and this gives us several cases we need to consider to return the result — to see the potential difficulties here, recall figure 2, and imagine this figure with an additional path of length 1 attached to the node 6 and a query LCA$(14, 15)$, and then another path of length 1 attached at the node 5 and a query LCA$(14, 16)$. Note that similar special cases arise in what was previously written in line 6 of the LCA algorithm as a simple equality check. Also the very first step of moving from a and b to their explicit ancestors requires some attention, since the topmost of the nodes a, b could be their LCA even though their explicit ancestors could be different.

After getting through all the technicalities from the previous section, the reader should be able to fill in this description with all the missing details. However, it is easy to see that this solution is much less pleasant to implement than the previous one, and it is faster only by a logarithmic factor. Having implemented an algorithm of that kind, to be more certain of the correctness of the program, it is highly advisable to test it against a brute-force algorithm on a large number of random test cases.

Hotel

Your friend owns a hotel at the seaside in Gdynia. The summer season is just starting and he is overwhelmed by the number of offers from potential customers. He asked you for help in preparing a reservation system for the hotel.

There are n rooms for rent in the hotel, the i-th room costs your friend c_i zlotys of upkeep (only if it is rented) and has a capacity of p_i people. You may assume that the upkeep of a room is never cheaper than the upkeep of any smaller room, that is, of any room which can hold a smaller number of people.

The reservation system will be receiving multiple offers, each of them specifying the amount of zlotys for rental of any room (v_j) for one particular day and the minimal capacity of the requested room (d_j). For each offer, only a single room can be rented. And conversely: each room can accommodate only a single offer. Your friend has decided not to accept more than o offers.

Knowing you are a skilled programmer, your friend asked you to implement the part of the system which finds the maximum profit (total income from renting out rooms minus their upkeep) he can make by accepting some of the offers.

Input

The first line of the standard input contains three integers n, m, and o ($1 \leqslant n, m \leqslant 500\,000$, $1 \leqslant o \leqslant min(m, n)$), denoting the number of rooms in the hotel, the number of offers received and the maximum number of offers your friend is willing to accept. The next n lines describe the rooms, with the i-th of these lines containing two integers c_i, p_i representing the upkeep of the room in zlotys and the capacity of the room ($1 \leqslant c_i, p_i \leqslant 10^9$). The next m lines describe the offers, with the j-th of these lines containing two integers v_j, d_j representing the offered rental price in zlotys and the minimal capacity of the requested room ($1 \leqslant v_j, d_j \leqslant 10^9$).

You may assume that in test cases worth 40 points in total an additional inequality $n, m \leqslant 100$ holds.

Output

The first and only line of the standard output should contain one integer equal to the maximum profit your friend can achieve accepting at most o of the offers. Note that the profit might get big.

Example

For the input data:	the correct result is:
3 2 2	400
150 2	
400 3	
100 2	
200 1	
700 3	

Explanation of the example: *Your friend can accept both offers, renting out rooms number 2 and 3.*

Solution

In this task we are asked to select at most o offers from the given set and pair them up with the rooms. We cannot pair offers with rooms smaller than requested. Under these constraints, we want to earn as much as we can.

Basic idea

Our solution is based on a greedy approach. We start off by sorting the offers by their offered price values (v_j) in a non-increasing order. Then for each offer we find the smallest room which is not yet paired with any other offer and is large enough to suffice the currently considered offer (i.e., $p_i \geqslant d_j$). If there is no such room, we discard the offer. Note that, by a corresponding condition from the problem statement, the selected room has the smallest upkeep among all rooms satisfying the requirements of the offer. We mark this offer and this room as *paired*.

After we finish pairing, we basically choose o best pairs as the result. More precisely, we order the pairs with respect to their profit, i.e., the

price offered decreased by the upkeep of the room. If the profit is negative (it is actually a loss), we discard the corresponding pair. As the result we take the top o pairs in the order of profit, or we take a smaller number of pairs if there are less than o pairs available.

Why does it work?

The above solution works in two phases: pairing the offers with the rooms and choosing the best pairs. Note that if the pairs created are fixed, then the second phase clearly gives the optimal result. So we only need to prove that we create the best pairing possible.

Let P_{ALG} be the set of pairs (offer, room) created by our algorithm and let P_{OPT} be the set of pairs created by a hypothetical optimal algorithm, that is, such a set of pairs for which the total profit generated is maximized. For a pairing P, we denote by $W(P)$ the profit this pairing generates, that is, the sum of profits of at most o most profitable pairs in P. If $P_{ALG} = P_{OPT}$ then our algorithm is certainly optimal. Otherwise, we will prove that P_{OPT} can be transformed into P_{ALG} without decreasing $W(P_{OPT})$.

Note that some offers and some rooms may not be included in each of the pairings. To simplify the analysis, we assume that there is a sufficient number of rooms with an extremely large capacity and also an extremely large upkeep, which can be paired with any of the offers. Similarly, we can add artificial offers with both the offered price and requested capacity equal to 0. From now on we can assume that each offer and each room is paired, possibly with an artificial room or offer.

Let j_1 be the first offer (in the order of the offered price, as in the solution description) which is paired with a different room by ALG and OPT. Assume that it is paired with the room i_1 by ALG and with the room i_2 by OPT. Due to the greedy manner in which the offers are paired with the rooms in our solution, we have $c_{i_2} \geqslant c_{i_1}$ and also $p_{i_2} \geqslant p_{i_1}$. Let j_2 be the offer which is paired with the room i_1 in OPT. We have:
$$(j_1, i_1) \in P_{ALG}, \quad (j_1, i_2), \ (j_2, i_1) \in P_{OPT}.$$

We will prove that by changing the pairs (j_1, i_2), (j_2, i_1) (*the old pairs*) into (j_1, i_1), (j_2, i_2) (*the new pairs*) we obtain a solution P'_{OPT} for which $W(P'_{OPT}) \geqslant W(P_{OPT})$. This will conclude the whole proof.

First of all, we need to prove that the offer j_2 can possibly be paired with the room i_2, i.e., that $p_{i_2} \geqslant d_{j_2}$. As we have already noticed, $p_{i_2} \geqslant p_{i_1}$, and since $(j_2, i_1) \in P_{OPT}$, we have $p_{i_1} \geqslant d_{j_2}$. Combining these two inequalities, we obtain that $p_{i_2} \geqslant d_{j_2}$, as requested.

Clearly, the sum of profits in the two old pairs and the two new pairs is exactly the same. However, the profits of the new pairs are more polarized:

$$v_{j_1} - c_{i_1} \;\geqslant\; v_{j_2} - c_{i_1}, \; v_{j_1} - c_{i_2} \;\geqslant\; v_{j_2} - c_{i_2}.$$

From this we can conclude that the swap of the rooms does not decrease $W(P_{OPT})$. Indeed, if any of the old pairs was present among the final o pairs (that is, included in $W(P_{OPT})$), then the pair (j_1, i_1) (or a pair with equal profit generated) will also be. If both old pairs were included, then either both new pairs will be included or (j_1, i_1) (or other profit-equivalent pair) and another pair not worse than (j_2, i_2) will be. Finally, if none of the old pairs was included in $W(P_{OPT})$, then the pair (j_1, i_1), with non-smaller profit, could only increase the total profit if it gets selected to $W(P_{OPT})$. In all the cases the profit stays the same or increases, which concludes the proof.

Implementation details

The second part of our solution (finding the o best pairs) can be implemented using any efficient sorting algorithm in time complexity $O((n + m) \cdot \log{(n + m)})$. In the first part of the solution we need a data structure which allows us to perform operations *lower_bound* (finding the smallest element in the structure above a given threshold) and *remove* (removing a given element) efficiently. We can choose one of the well-known dictionary data structures: AVL tree, Splay, or Red-Black tree (which is implemented as `set` or `map` container in the STL in C++). Then each of the requested operations is performed in logarithmic time and the structure can initially be constructed in linear-logarithmic time. The total running time of the algorithm is $O((n + m) \cdot \log{(n + m)})$.

What if you do not like STL?

Well... if you fancy Pascal and do not feel like implementing AVL/Red-Black trees by your own, you could either try to use a static data structure (like a segment tree) instead, or take a slightly more Pascal-friendly approach proposed in this section. The rough idea is to find a different data structure which can perform the operations *remove*, *lower_bound* in a decent amortized time.

Let us consider the following solution, which uses the *Find-Union* data structure. We first sort all the offers by their price values in a non-increasing order and the rooms by their sizes (equivalently, upkeep) in a non-decreasing order. Then for each offer we use binary search to find the first room satisfying the offer. If we decide to create a pair consisting of the offer and the room, we mark the room as paired and union it with the set containing the next room on the list. After the binary search, if we hit a room which is marked as paired, we consider the furthest room it is in the same set with instead. This way we will ignore all already used rooms. In total we perform $O(n + m)$ Find-Union operations in addition to $O(n + m)$ binary searches, thus the time complexity of this solution is also $O((n + m) \cdot \log{(n + m)})$.

Other approaches

One could try to visualize this task as a minimum cost maximum flow problem in a bipartite graph. Using any of the classical efficient algorithms for this problem, one would get 40 points for this task.

If a linear time implementation of the *remove* and *lower_bound* operations is used instead of the efficient algorithms described above, an $O(nm)$ time algorithm can be obtained. Such a solution would receive about 50 points.

Adam Karczmarz   **Dawid Dąbrowski, Jakub Łącki**

Task idea and formulation   Analysis

Teams

A Sports Day is being held in a primary school in Gdynia. The most important part of the event is the Annual Football Cup.

Several children gathered at the football pitch, where teams were to be formed. As everyone wanted to belong to the best team, the players could not reach an agreement. Some of them threatened not to play, others started to cry and now nobody is sure if the tournament will take place at all.

Byteman, a sports teacher, is in charge of organizing the tournament. He decided to split the children into teams himself, so that no player would be unhappy with her team. The i-th of the n children on the pitch (numbered 1 through n) said that she will be unhappy with her team if the team consists of less than a_i players.

Apart from satisfying the children's requirements, Byteman would like to maximize the total number of teams. If there are still many possibilities, he requests the size of the largest team to be as small as possible. As it turned out to be quite a difficult task, Byteman asked you for help.

Input

In the first line of the standard input there is one integer n ($1 \leqslant n \leqslant 10^6$). Then, n lines follow. The i-th of these lines contains a single integer a_i ($1 \leqslant a_i \leqslant n$), that denotes the minimum team size that satisfies the child number i.

Additionally, in test cases worth at least 50 points, n will not exceed 5 000.

Output

In the first line of the standard output your program should write a single integer t equal to the maximum possible number of teams. Then, t lines containing a description of the teams should follow. The i-th of these lines should contain an integer s_i ($1 \leqslant s_i \leqslant n$) denoting the size of the i-th team, and then s_i integers $k_1, k_2, \ldots, k_{s_i}$ ($1 \leqslant k_j \leqslant n$ for $j = 1, 2, \ldots, s_i$), denoting the numbers of children belonging to the team i. If there are many possible answers, you can output any of the solutions which minimize the size of the largest team (among all the solutions consisting of exactly t teams).

Example

<table>
<tr><td>For the input data:</td><td>the correct result is:</td></tr>
<tr><td>5</td><td>2</td></tr>
<tr><td>2</td><td>2 4 2</td></tr>
<tr><td>1</td><td>3 5 1 3</td></tr>
<tr><td>2</td><td></td></tr>
<tr><td>2</td><td></td></tr>
<tr><td>3</td><td></td></tr>
</table>

Solution

In this task we are to divide children into teams. Let us say that a division is *valid* whenever all children's demands are fulfilled. We call every valid division a *solution*. A solution is *optimal* if and only if it consists of the smallest possible number of teams and, among all such solutions, the size of the largest team is minimized.

First, let us investigate when a group of children may form a team. Observe that a team may consist of children numbered $i_1, i_2, \ldots, i_k$ if and only if $\max(a_{i_1}, a_{i_2}, \ldots, a_{i_k}) \leqslant k$, that is, the size of the team is at least as big as the largest among the requirements. Clearly, if this condition holds, then all the demands are satisfied. Otherwise an inequality $a_{i_l} > k$ must hold for some l, which means a demand is not satisfied.

Before we proceed with further observations, let us assume that the children are ordered in such a way, that their requirements form a non-decreasing sequence, i.e., $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. This can be achieved in linear time by counting sort, as the numbers are not greater than n.

Now, we can only consider solutions in which all teams consist of consecutive children, as there exists at least one optimal solution of this form. The proof of this statement is simple. Let us fix two teams of sizes $t_1 \leqslant t_2$, such that at least one of them does not consist of consecutive children. We can take all children from both teams and split them in a pair of new teams, still of sizes t_1 and t_2. Namely, t_1 children with smallest requirements will form the first team, and the remaining t_2 children — the second team. It is easy to observe that in both teams all the demands are met, since their sizes remained unchanged and the biggest requirement in any team could only decrease. In addition to that, the greatest index of a child in both teams does not increase and

in at least one of the teams it *decreases.* This means that we can repeat this process until all teams consist of consecutive children. From now on (until the final part of the last section of the task analysis) we only consider solutions in which all teams consist of consecutive children.

An $O\left(n^2\right)$ time solution

The observations made so far lead to a simple quadratic time solution based on dynamic programming. For each i from 1 to n we calculate two values: $\text{TEAMS}[i]$ — the greatest number of teams in the optimal solution for the children $1, 2, \ldots, i$, and $\text{MAXTEAM}[i]$ — the size of the largest team in that solution. If for some i the children cannot be partitioned in a valid way, we set $\text{TEAMS}[i] = -\infty$. This happens, for example, when $a_i > i$. For simplicity we assume that $\text{TEAMS}[0] = 0$ and $\text{MAXTEAM}[0] = -\infty$.

Suppose we have already computed TEAMS and MAXTEAM for $1, 2, \ldots, i-1$ and we want to know $\text{TEAMS}[i]$ and $\text{MAXTEAM}[i]$. According to the observations, we assume that the i-th child belongs to the team consisting of children $j, j+1, \ldots, i$ for some $1 \leqslant j \leqslant i$. Hence, $\text{TEAMS}[i] = \text{TEAMS}[j-1] + 1$ and $\text{MAXTEAM}[i] = \max\{i - j + 1, \text{MAXTEAM}[j-1]\}$.

We can search for j which maximizes $\text{TEAMS}[i]$ and among all such j's pick the one which minimizes $\text{MAXTEAM}[i]$. Clearly, the resulting solution works in $O(n^2)$ time. It should score 50 points.

More efficient solutions

We implement an auxiliary function $\text{MAXTEAMS}(limit)$, which computes the division of children into a maximal number of teams of size at most *limit*. We first call $\text{MAXTEAMS}(n)$ to find the maximal possible number of teams T and then do a binary search to find the lowest M, for which $\text{MAXTEAMS}(M)$ computes the division into T teams.

We now describe how to implement $\text{MAXTEAMS}(limit)$, so that it runs in $O(n \log n)$ or $O(n)$ time. We compute an array TEAMSL such that $\text{TEAMSL}[i]$ is equal to the greatest number of teams of size at most *limit* that can be formed from children $1, 2, \ldots, i$. Again, we only consider teams consisting of consecutive children.

The basic observation is that the team containing child i has size x, which satisfies $a_i \leqslant x \leqslant limit$. Hence, we can find such s ($a_i \leqslant s \leqslant limit$) that $\text{TEAMSL}[i-s]$ is maximal and form a new team from children $i-s+1, i-s+2, \ldots, i$, thus creating $\text{TEAMSL}[i-s]+1$ teams in total. Finding the right value for s can be achieved in $O(\log n)$ time, using an efficient data structure for range minimum queries in the TEAMSL array, for example, a segment tree. In total, we call MAXTEAMS $O(\log n)$ times, because of binary search, and since each call takes $O(n \log n)$ time, we obtain an $O(n \log^2 n)$ time solution. Such a solution scores about 70 points.

This can also be done more efficiently. If for some $i < n$ there exists $j < i$ such that the maximal possible number of teams that can be formed from children $1, 2, \ldots, i$ is smaller than $\text{TEAMSL}[j]$, we set $\text{TEAMSL}[i] = -\infty$. This is because in any optimal solution (with each team consisting of consecutive children) there cannot be a team in which the child with the biggest requirement is i.

To prove it, assume that there is an optimal solution which contains a team $j+1, j+2, \ldots, i$. Then, we can alter the solution by building $\text{TEAMSL}[j]$ teams from children $1, 2, \ldots, j$ and adding children $j+1, j+2, \ldots, i$ to the group containing the n-th child. As a result, we increase the number of groups, a contradiction.

From the observation, it follows that the sequence of all $\text{TEAMSL}[i]$ which are not equal to $-\infty$ is non-decreasing. Hence, to find the greatest value of $\text{TEAMSL}[i]$ in a given interval, it suffices to find the last non-negative value. This can be done in constant time if in the dynamic programming for every i we also compute the biggest $i' \leqslant i$ such that $\text{TEAMSL}[i']$ is non-negative.

This algorithm, running in $O(n \log n)$ time, scores 100 points.

Model $O(n)$ time solution

The model solution is quite tricky. We reverse the order of the children and assume that they are sorted in non-ascending order, i.e., $a_1 \geqslant a_2 \geqslant \ldots \geqslant a_n$. Let $\text{TEAMSR}[i]$ and $\text{MAXTEAMR}[i]$ denote the maximum number of teams that can be formed from children $1, 2, \ldots, i$ and the minimum possible size of the biggest team, respectively. We append the letter R to the names of the arrays, as the children are now sorted in reverse order. The group containing child number 1 has to

contain at least a_1 children. Hence, for $i < a_1$, $\textsc{TeamsR}[i] = -\infty$. For $i = a_1$ we have $\textsc{TeamsR}[i] = 1$ and $\textsc{MaxTeamR}[i] = i$.

We compute both $\textsc{TeamsR}$ and $\textsc{MaxTeamR}$ again using dynamic programming. Assume that we want to compute $\textsc{TeamsR}[i]$ and $\textsc{MaxTeamR}[i]$, where $i > a_1$.

Let us take a look at the solution for $i - 1$. We can add the i-th child to the team which contains child $i - 1$. This obviously does not increase the total number of teams. On the other hand, it is easy to see that from the children $1, 2, \ldots, i$ we can form at most one more team than from children $1, 2, \ldots, i - 1$. Hence, either $\textsc{TeamsR}[i] = \textsc{TeamsR}[i - 1]$ or $\textsc{TeamsR}[i] = \textsc{TeamsR}[i - 1] + 1$.

How do we check whether $\textsc{TeamsR}[i] = \textsc{TeamsR}[i - 1] + 1$? This is the case when for some $1 \leqslant j \leqslant i$ we have $\textsc{TeamsR}[j - 1] = \textsc{TeamsR}[i - 1]$ and we can form a valid team from the children numbered $j, j+1, \ldots, i$. The second condition is equivalent to checking if $a_j \leqslant i - j + 1$. If we search for such j iterating through all possibilities, we will end up with another $O(n^2)$ solution. In fact it is very similar to the one explained above.

To accelerate it, we observe that once we find $1 \leqslant j \leqslant i$, such that $\textsc{TeamsR}[j - 1] = \textsc{TeamsR}[i - 1]$ and $a_j \leqslant i - j + 1$, then in fact $a_j = i - j + 1$. Otherwise, we would have $a_j < i - j + 1$ so we could create $\textsc{TeamsR}[j - 1]$ teams from children $1, 2, \ldots, j - 1$ and another team from those numbered $j, j + 1, \ldots, i - 1$. As a result, we would get $\textsc{TeamsR}[j-1]+1 = \textsc{TeamsR}[i-1]+1$ teams from children $1, 2, \ldots, i - 1$, which contradicts the definition of $\textsc{TeamsR}[i - 1]$.

Hence, we can search only for j's satisfying $a_j = i - j + 1$, which can be rewritten as $a_j + j = i + 1$. This means that for every j there is at most one i, such that $a_j + j = i + 1$. Thus, for each i one can list all possible values of j and then, while computing $\textsc{TeamsR}[i]$, iterate through those j's.

It remains to find $\textsc{MaxTeamR}[i]$, that is, the size of the largest team in the optimal solution for children $1, 2, \ldots, i$. If $\textsc{TeamsR}[i] = \textsc{TeamsR}[i-1]+1$, it can be computed along with searching for the size of the team containing the i-th child. The case when $\textsc{TeamsR}[i] = \textsc{TeamsR}[i - 1]$ is more complex. Let us consider how an optimal solution for children $1, 2, \ldots, i$ can be formed in this case. There are two possible ways of forming the teams.

1. If removing the i-th child does not violate the requirement of any of her teammates, we can take an optimal division of

children $1, 2, \ldots, i - 1$ into $\text{TEAMSR}[i - 1]$ teams and add the child i to one of those teams. The optimal way is to add her to the smallest among those teams[1]. Observe that the requirement of the child i is satisfied, as she joins a team, in which requirements are greater than or equal to a_i. If $i - 1 = \text{TEAMSR}[i - 1] \cdot \text{MAXTEAMR}[i - 1]$ then all teams are of equal size, so after adding the child i, one team consists of $\text{MAXTEAMR}[i-1]+1$ children. Otherwise, we keep the maximum team size intact and set $\text{MAXTEAMR}[i - 1] = \text{MAXTEAMR}[i]$.

2. We can also form a team from children numbered $j, j + 1, \ldots, i$ for some j, such that $\text{TEAMSR}[j - 1] = \text{TEAMSR}[i] - 1$. This means that $a_j + j = i + 1$. As before, we can iterate over all j's satisfying the equation and find the one that minimizes the largest team size.

The above ideas allow us to compute both arrays in $O(n)$ total time.

---

[1]Note that this may result in creating teams that do not consist of consecutive children, but, as we argued before, any solution can be transformed to such a form.

Jakub Łącki

Task idea and formulation

Mateusz Baranowski, Tomasz Kociumaka

Analysis

Traffic

The center of Gdynia is located on an island in the middle of the Kacza river. Every morning thousands of cars drive through this island from the residential districts on the western bank of the river (using bridge connections to junctions on the western side of the island) to the industrial areas on the eastern bank (using bridge connections from junctions on the eastern side of the island).

The island resembles a rectangle, whose sides are parallel to the cardinal directions. Hence, we view it as an $A \times B$ rectangle in a Cartesian coordinate system, whose opposite corners are in points $(0,0)$ and (A,B).

On the island, there are n junctions numbered from 1 to n. The junction number i has coordinates (x_i, y_i). If a junction has coordinates of the form $(0,y)$, it lies on the western side of the island. Similarly, junctions with the coordinates (A,y) lie on the eastern side. Junctions are connected by streets. Each street is a line segment connecting two junctions. Streets can be either unidirectional or bidirectional. No two streets may have a common point (except for, possibly, a common end in a junction). There are are no bridges or tunnels. You should not assume anything else about the shape of the road network. In particular, there can be streets going along the river bank or junctions with no incoming or outgoing streets.

Because of the growing traffic density, the city mayor has hired you to check whether the current road network on the island is sufficient. He asked you to write a program which determines how many junctions on the eastern side of the island are reachable from each junction on the western side.

Input

The first line of the standard input contains four integers n, m, A and B $(1 \leqslant n \leqslant 300\,000,\ 0 \leqslant m \leqslant 900\,000,\ 1 \leqslant A, B \leqslant 10^9)$. They denote the number of junctions in the center of Gdynia, the number of streets and dimensions of the island, respectively.

In each of the following n lines there are two integers x_i, y_i $(0 \leqslant x_i \leqslant A,\ 0 \leqslant y_i \leqslant B)$ describing the coordinates of the junction number i. No two junctions can have the same coordinates.

The next m lines describe the streets. Each street is represented in a single line by three integers c_i, d_i, k_i $(1 \leqslant c_i, d_i \leqslant n,\ c_i \neq d_i,\ k_i \in \{1, 2\})$. Their

48   *Traffic*

meaning is that junctions c_i and d_i are connected with a street. If $k_i = 1$, then this is a unidirectional street from c_i to d_i. Otherwise, the street can be driven in both directions. Each unordered pair $\{c_i, d_i\}$ can appear in the input at most once.

You can assume that there is at least one junction on the western side of the island from which it is possible to reach some junction on the eastern side of the island.

Additionally, in test cases worth at least 30 points, $n, m \leqslant 6\,000$.

Output

Your program should write to the standard output one line for each junction on the western side of the island. This line should contain the number of junctions on the eastern side that are reachable from that junction. The output should be ordered according to **decreasing** *y-coordinates of the junctions.*

Example

For the input data:	the correct result is:
5 3 1 3	2
0 0	0
0 1	2
0 2	
1 0	
1 1	
1 4 1	
1 5 2	
3 5 2	

Whereas for the input data:

```
12 13 7 9
0 1
0 3
2 2
5 2
7 1
7 4
7 6
7 7
3 5
0 5
0 9
3 9
1 3 2
3 2 1
3 4 1
4 5 1
5 6 1
9 3 1
9 4 1
9 7 1
9 12 2
10 9 1
11 12 1
12 8 1
12 10 1
```
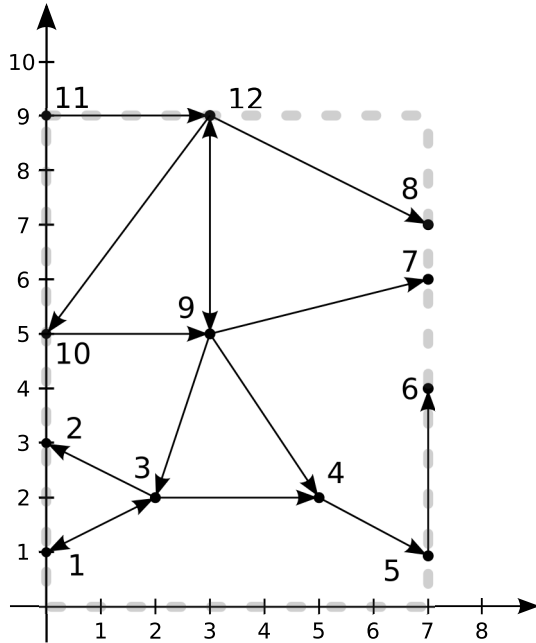
the correct result is:

```
4
4
0
2
```

Solution

The road network in the center of Gdynia is described with a directed graph. The graph is drawn in a rectangle, in such a way that no two edges intersect. In particular, the graph is *planar*.

The goal is to determine the number of vertices drawn on the right side of the rectangle reachable from each vertex on the left side.

A simple solution

The problem can easily be solved with a graph searching algorithm. For every junction on the western side, we can compute all reachable junctions on the eastern side in linear time, using, for example, the BFS algorithm. It is a well-known fact that planar graphs with n vertices have $O(n)$ edges, so such algorithm runs in $O(n^2)$ total time, and scores about 30 points.

A few observations

The following solutions, unlike the previous one, take advantage of planarity of the graph. In the description, we write $u \rightsquigarrow v$ to denote that there exists a directed path from a vertex u to v.

The model solution starts by eliminating unnecessary vertices from the graph. We remove all junctions which cannot be reached from any western junction and all western junctions, from which it is not possible to reach any eastern junction. This can be achieved in linear time using BFS: first we run the graph search algorithm starting in all the western junctions and discard all eastern junctions which are not visited; then we apply the same procedure for all the eastern junctions in the graph with all edges reversed. We need to consider these removed vertices while printing the output, but, for the sake of simplicity, in the following we assume that in the input such junctions have already been removed.

Let $w_1, w_2, \ldots, w_c$ be the junctions on the western side and let $e_1, e_2, \ldots, e_d$ denote the junctions on the eastern side of the island. In both lists, the junctions are sorted in ascending order of second coordinates.

The removal of some junctions allows us to formulate the key observation.

Observation 1. Let $1 \leqslant x < y \leqslant d$ and let v be an arbitrary vertex. If $v \rightsquigarrow e_x$ and $v \rightsquigarrow e_y$, then for all z, such that $x < z < y$, $v \rightsquigarrow e_z$.

Proof: This property follows directly from the structure of the network. Fix some $x < z < y$. There exists some western junction w_q, such that $w_q \rightsquigarrow e_z$. Because the graph is planar, the path from w_q to e_z has to intersect one of the paths $v \rightsquigarrow e_x$, $v \rightsquigarrow e_y$ in some junction u. Hence, there exists a path $v \rightsquigarrow u \rightsquigarrow e_z$. ∎

The observation implies that the set of junctions reachable from any western junction is a sequence of *consecutive* junctions on the eastern side. We define $\texttt{north}(w_i)$ to be the index of northernmost eastern junction reachable from w_i and $\texttt{south}(w_i)$ to be the index of southernmost eastern junction reachable from w_i. Thus, the number of eastern junctions reachable from w_i is $\texttt{north}(w_i) - \texttt{south}(w_i) + 1$.

Model solution

How can we compute the functions $\texttt{north}$ and $\texttt{south}$? We concentrate on the former one, since the latter can be computed in a similar manner.

Observe that the $\texttt{north}$ function is non-decreasing, that is, for $i < j$, $\texttt{north}(w_i) \leqslant \texttt{north}(w_j)$. To prove it, let us assume the opposite, that is, $\texttt{north}(w_j) = z < \texttt{north}(w_i)$. We exploit the planarity of the graph and the facts that $i < j$ and $z < \texttt{north}(w_i)$. The path $w_j \rightsquigarrow e_z$ has to intersect the path $w_i \rightsquigarrow e_{\texttt{north}(w_i)}$. Hence, there exists a path $w_j \rightsquigarrow e_{\texttt{north}(w_i)}$. This contradicts the assumption that $\texttt{north}(w_j) < \texttt{north}(w_i)$.

We can now give a simple algorithm. For consecutive values of i, starting from 1, we compute $\texttt{north}(w_i)$ using DFS or BFS. Due to monotonicity, we do not need to visit nodes which were already visited in one of the previous searches. If during the search originating in w_i we visit some eastern junctions, then we know that $\texttt{north}(w_i)$ is the greatest among all indices of visited eastern junction. Otherwise, $\texttt{north}(w_i) = \texttt{north}(w_{i-1})$.

Since we visit every junction exactly once, the time and space complexity is linear. The $\texttt{south}$ function is computed similarly.

Because we initially sort all eastern and western junctions, the overall time complexity is $O(n \log n)$.

Alternative solution

One can also take a very different approach to computing `north` and `south` functions. It uses the concept of strongly connected components.

Let us naturally extend the `north` and `south` functions to all junctions. Let $\mathcal{N}(v)$ be the set of vertices directly reachable from v. It is clear that for any junction v which is not on the eastern side $\text{north}(v) = \max_{w \in \mathcal{N}(v)} \text{north}(w)$, whereas for any $1 \leqslant i \leqslant d$, $\text{north}(e_i)$ is the maximum of $\max_{w \in \mathcal{N}(e_i)} \text{north}(w)$ and i.

A similar formula holds for the `south` function. Unfortunately, our graph can contain cycles, so the formulas might lead to recursive dependencies.

To deal with that, we find strongly connected components in the input graph. This can be done in linear time using Tarjan's algorithm or Kosaraju's algorithm. After that, we construct a directed acyclic graph G by contracting all edges within each strongly connected component. For any two vertices u and v belonging to the same strongly connected component, $\text{north}(u) = \text{north}(v)$ and $\text{south}(u) = \text{south}(v)$. Therefore it suffices to compute the values of both functions in the graph G. We first compute them for vertices v such that $\mathcal{N}(v)$ is empty and then go through all the remaining vertices in reverse topological order and use the aforementioned formulas. As a result, we obtain another solution working in $O(n \log n)$ time.

CEOI'2011 has been supported by:

OFEK

OGÓLNOPOLSKA FUNDACJA
EDUKACJI KOMPUTEROWEJ

GDYNIA
moje miasto