VPCPC 2014

problems and sample solutions

The first ever Visegrad Programming Contests Preparation Camp took place in Danišovce, Slovakia, between June 28th and July 6th, 2014. The camp was attended by delegations from all four Visegrad countries: Czech Republic, Hungary, Poland, and Slovakia. The camp was partially supported by the Visegrad Fund Small Grant 11340003.

Table of Contents

Part 0: About the camp 4

Preface 4

Results 5

Statistics 6

Part I: Problem statements 7

Day 1: Slovak problems 7
Hyperways 7
Dynamic memory allocation 8
Shades of the town 9

Day 2: Czech problems 11
Bus lines 11
Cutting of a birthday cake 12
Investigation 14

Day 3: Mixed problems 16
New Tree 16
Cubic Art 17
Universities 20
Wall 22

Day 4: Hungarian problems 24
Critical Projects 24
Connect Highways 26
Next Permutation 28

Day 5: Polish problems 29
Game 29
Posters 30
Sorting 32

Day 6: Mixed problems 33
Mission 33
Newspapers 35
An inexperienced slalomer 36
Tickets 37

 licensed under CC BY-SA 3.0

Part II: Solutions **38**

Day 1: Slovak problems **38**
Hyperways 38
Dynamic memory allocation 40
Shades of the town 42

Day 2: Czech problems **44**
Bus lines 44
Cutting 45
Investigation 46

Day 3: Mixed problems **49**
New Tree 49
Cubic Art 52
Universities 54
Wall 56

Day 4: Hungarian problems **58**
Critical Projects 58
Connect Highways 60
Next Permutation 63

Day 5: Polish problems **65**
Game 65
Posters 66
Sorting 70

Day 6: Mixed problems **71**
Mission 71
Newspapers 72
An inexperienced slalomer 73
Tickets 74

licensed under CC BY-SA 3.0

Preface

Hello, dear reader!

This booklet contains all tasks and solution writeups from the first ever Visegrad Programming Contest Preparation Camp (VPCPC). The VPCPC is continuing and extending a long tradition of international algorithmic preparation camps in central Europe. The predecessor of this camp was the Czech-Polish-Slovak Preparation Camp (CPSPC). This camp has been organized once per year by one of the three participating countries. The first ever CPSPC took place in the summer of 1999 in Belušské Slatiny, Slovakia. After starting that tradition, we are now hoping that we were able to start a new, even better one. We would love to see and attend many more VPCPCs in the years to come!

Anyway, let's get back to the tasks. The participants invited to the camp are secondary school students who are among the best in algorithmic problem solving. Most of them are future participants in the International Olympiad in Informatics (IOI) and similar contests. The 20 tasks used at VPCPC 2014 reflect the skill level of these contestants. (Read: most of the tasks are quite difficult.)

The test data for all the tasks is available at `https://github.com/trojsten/vpcpc`. All the materials from the camp are released under the Creative Commons Attribution-ShareAlike (CC BY-SA) 3.0 license.

Michal Forišek
Bratislava, July 2014

 licensed under CC BY-SA 3.0

Results

Below are the results of VPCPC 2014. Each problem was worth 100 points. Hence, the theoretical maximum was 2000 points: contestants could score at most 300 points on national days and at most 400 points on mixed days (day 3 and day 6). Note that contestant names are printed according to national customs – i.e., the family name is printed first for Hungarian contestants and last for contestants from the other three countries.

rank	contestant (country)	total	day 1	day 2	day 3	day 4	day 5	day 6
1.	Eduard Batmendijn (SK)	1709.50	280.00	259.50	340.00	300.00	300.00	230.00
2.	Jarosław Kwiecień (PL)	1456.37	260.00	196.37	300.00	200.00	200.00	300.00
3.	Stanisław Barzowski (PL)	1337.21	200.00	184.21	240.00	300.00	180.00	233.00
4.	Michał Glapa (PL)	1284.98	200.00	194.98	200.00	200.00	200.00	290.00
5.	Maciej Holubowicz (PL)	1152.02	280.00	164.52	200.00	140.00	167.50	200.00
6.	Jan-Sebastian Fabík (CZ)	1013.64	200.00	164.14	150.00	156.50	180.00	163.00
7.	Martin Raszyk (CZ)	976.50	240.00	120.00	43.00	173.00	237.50	163.00
8.	Jan Tabaszewski (PL)	969.09	0.00	153.59	173.00	200.00	212.50	230.00
9.	Albert Citko (PL)	939.20	40.00	151.70	200.00	200.00	237.50	110.00
10.	Michal Korbela (SK)	933.76	140.00	148.26	100.00	240.00	142.50	163.00
11.	Weisz Ambrus (HU)	917.59	160.00	152.59	83.00	249.50	142.50	130.00
12.	Erdős Márton (HU)	715.04	60.00	129.54	133.00	200.00	82.50	110.00
13.	Somogyvári Kristóf (HU)	684.40	140.00	146.90	175.00	40.00	82.50	100.00
14.	Pavel Madaj (SK)	584.96	140.00	134.46	140.00	20.00	87.50	63.00
15.	Ondřej Hübsch (CZ)	575.69	180.00	143.19	33.00	56.50	100.00	63.00
16.	Michal Sládeček (SK)	568.49	120.00	46.99	63.00	73.00	102.50	163.00
17.	Székely Szilveszter (HU)	565.80	80.00	80.30	50.00	143.00	112.50	100.00
18.	Mernyei Péter (HU)	557.24	100.00	81.74	50.00	153.00	52.50	120.00
19.	Zarándy Álmos (HU)	549.49	140.00	46.99	50.00	120.00	142.50	50.00
20.	Peter Ralbovský (SK)	503.57	100.00	102.57	73.00	0.00	65.00	163.00
21.	Václav Rozhoň (CZ)	479.75	140.00	126.75	33.00	40.00	110.00	30.00
22.	Matěj Konečný (CZ)	436.00	60.00	120.00	0.00	173.00	20.00	63.00
23.	Dominik Smrž (CZ)	402.50	140.00	20.00	10.00	120.00	112.50	0.00
24.	Alan Marko (SK)	339.90	0.00	106.90	33.00	0.00	70.00	130.00

 licensed under CC BY-SA 3.0

Statistics

The following table contains some statistics for the individual tasks. The columns of the table contain the following information:

- The column "full" is the number of contestants who achieved a perfect score (i.e., 100 points).
- The column "mean" is the arithmetic mean (average) of all scores.
- The column "median" is the median score – more precisely, the average of the scores of the 12th best and the 13th best solver of that task.

The task day2-cutting was an open-data task with relative scoring. The best score awarded to a contestant was 84.21.

task id	full	mean	median
day1-hyperways	3	33.33	40.00
day1-malloc	8	52.50	40.00
day1-shades	5	55.83	60.00
day2-buslines	18	84.16	100.00
day2-cutting	0	24.84	16.95
day2-investigation	2	23.33	20.00
day3-newtree	1	20.00	0.00
day3-rubik	4	21.88	0.00
day3-universities	7	41.54	33.00
day3-wall	8	36.25	10.00
day4-critical	8	52.50	40.00
day4-networks	4	29.06	8.25
day4-nextperm	15	64.16	100.00
day5-game	8	48.75	30.00
day5-posters	1	22.92	12.50
day5-sorting	12	67.50	85.00
day6-mission	4	29.04	33.00
day6-newspaper	3	36.25	30.00
day6-slalom	0	0.00	0.00
day6-tickets	17	75.00	100.00

 licensed under CC BY-SA 3.0

Hyperways

task: `hyperways`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `3000 ms`	memory limit: `1 GB`

The intergalactic company Oods&co is going to build a network of hyperways[1] that will connect the planets of our galaxy. They have already prepared a construction plan, i.e., a sequential order of building the hyperways. Each hyperway will be a bidirectional corridor that will connect two (not necessarily distinct) planets.

Task

A hyperway is *safe* if it is not the only hyperway that connects (directly or indirectly) some pair of planets. In other words, a hyperway H is *unsafe* if there are two planets A and B such that when traveling from A to B we have to use H. (Alternately, note that safe hyperways are the ones that lie on some cycle.)

You will be given the order in which hyperways will be constructed. After each construction there may be some hyperways which have just become safe. (Including, possibly but not necessarily, the hyperway that was just built.) Count those hyperways.

Note that if a hyperway is already safe, it will remain safe for the rest of the construction.

Input

On the first line of input are two integers n and m: n is the number of planets in the plan, m is the number of hyperways. The planets are numbered $1 \ldots n$.

Each of next m lines consists of two space separated integers – the ids of planets next hyperway will join. There can be a hyperway connecting a planet with itself. There can be a multiple hyperways between two planets.

In all test cases, $n \leq 10^6$ and $m \leq 2 \cdot 10^6$.

In 40% of test cases, $n \leq 1000$ and $m \leq 2000$.

Output

For each hyperway in the input, output a single line with a single integer: the number of hyperways that just became safe.

Samples

input

```
5 8
1 2
3 3
4 5
2 3
4 5
3 4
4 1
5 2
```

output

```
0
1
0
0
2
0
4
1
```

[1] hyperspace highways, also called hyhi

 licensed under CC BY-SA 3.0

Dynamic memory allocation

task: `malloc`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1200 ms`	memory limit: `1 GB`

Kamila is once again devising a new amazing programming language that will be superior to all other languages in every aspect. Writing code will become even simpler than ABC, so little children will learn how to program before they will know how to write comments. This is not a bug,[2] as comments are absolutely unnecessary in Kamila's dreamt up language: Due to its intuitive syntax, the purpose of any block of code is immediately crystally clear. Also, the compiler will bake you a cake for your birthday.

There is still a handful of outstanding issues though. For example, the language constructs for memory management do not work yet. Kamila prepared a precise specification of the memory allocation and freeing procedures, but she did not find enough time to implement them. Help her, and achieve instant fame for contributing to such an important project!

Task

The available memory is an array of n bytes numbered 0 through $n-1$. At the beginning, all the bytes are free (i.e., not allocated). Then the memory management system allocates and frees the bytes according to a sequence of queries.

An *allocation* query is specified by an integer ℓ. The system finds a block of ℓ consecutive free bytes, allocates them, and returns the position of the first byte in this block. If there are multiple such blocks available, the one starting at the position with the smallest number is chosen. If there is no such block available, the query is rejected and the system returns -1.

A *freeing* query is specified by two integers x and ℓ. The system marks the block of ℓ consecutive bytes starting at the position x as free, and returns the number of actually freed bytes (i.e., the number of bytes in this block that were not free before the query).

Input

The first line of the input consists of two integers n and q – the number of bytes in the available memory and the number of queries ($1 \leq n, q \leq 3 \cdot 10^5$).

Each of the following q lines describes a query. The first integer in a line determines the type of the query: 1 is for allocation and 2 for freeing. A line with an allocation query then continues with one additional integer ℓ ($1 \leq \ell \leq n$). Similarly, a line with a freeing query continues with two integers x and ℓ ($0 \leq x \leq n-1$, $1 \leq \ell \leq n-x$).

Output

For every query in the given sequence, output one line with the value returned by the system.

Samples

input

```
5 4
1 3
1 3
2 1 3
1 4
```

output

```
0
-1
2
1
```

[2]By the way, it will be impossible to introduce bugs into a program – Kamila's brilliant compiler will optimize them out.

 licensed under CC BY-SA 3.0

Shades of the town

task: `shades`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1000 ms`	memory limit: `1 GB`

A long long time ago in a galaxy far away, there was a town. Well, we assume that it was a town. But it disappeared and almost nothing was left. Just a shade. A dark, cold shade. A shade which doesn't move. The shade of destiny.

We assume that all the buildings in the town stood in a single line and were equally spaced. All buildings had the same width, only their heights differed. The buildings are now gone, only their shadows remain. Note that the lengths of shadows don't have to be the same as the heights of the original buildings: they can all be scaled by the same constant factor.

We are interested in the architecture of the culture that occupied the planet. We have multiple sequences of building heights (called "patterns"). For each such pattern, we would like to find all possible occurrences in the original sequence of buildings.

Task

You are given a sequence of positive integers: the lenghts of the preserved shadows. You are also given several queries. In each query we give you one pattern. The pattern is a sequence of positive integers: the heights of some buildings. We say that a pattern occurs in the shade if there is a contiguous subsequence of the shadows that is the same as the pattern, scaled by a positive real factor.

For each pattern, find the number of its occurrences in the original sequence. (The occurrences are allowed to overlap.)

Input

On the first line of input is a single integer n – the number of paterns.

Each of next n lines describes one pattern. It starts with an integer l_i – the patern length. Then l_i space separated positive integers follow.

The last line describes the shade. It starts with an integer m – the shade length. Then m space separated positive integers follow.

Constraints:

- $1 \leq m \leq 3 \cdot 10^5$
- $1 \leq l_i$
- $\sum_{i=1}^{n} l_i \leq m$
- All heights and lengths are between 1 and 10 000, inclusive.

In 40% of testcases we have $m \leq 1000$.

Output

Output a single line with a single integer – the total number of occurrences of all patterns in the shade.

Samples

 licensed under CC BY-SA 3.0

input

```
4
1 47
2 21 42
2 34 17
3 1 2 1
7 3 6 3 6 12 6 3
```

output

```
15
```

The first patterns can be scaled to any height and so it occurs 7 times. The second and the third pattern each occur 3 times. The fourth pattern occurs 2 times.

 licensed under CC BY-SA 3.0

Bus lines

task: `buslines`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `3000 ms`	memory limit: `1 GB`

In the Czech city called Kocourkov they have a spectacular public transportation system. It consists of N bus stops and $N-1$ bidirectional roads, each road connecting two bus stops. It is possible to get from each bus stop to every other using a sequence of roads.

Every morning, for each pair of distinct bus stops a and b there is exactly one bus that starts at a and goes to b (along the only direct path). That is, there are a total of $N(N-1)$ buses. Each bus stops at all bus stops it visits along the way.

At every bus stop there must be a timetable listing all the buses that stop there (including buses that start or end their journey there). You are now wondering how many buses are listed on each timetable.

Task

You are given the description of the traffic system in Kocourkov. For every bus stop in the city calculate the number of buses that stop on that particular stop.

Input

First line contains an integer N, the number of bus stops in the city (stops are numbered from 1 to N). The following $N-1$ lines describe the roads in the city. Each line contains two different integers $1 \leq x, y \leq N$ meaning that there is a road connecting bus stops x and y.

It holds $1 \leq N \leq 10^6$.
In the 20% of testcases $N \leq 100$.
In the 40% of testcases $N \leq 1000$.

Output

The output consists of N lines. The i-th line should contain a signle integer, the number of buses that stop on the i-th bus stop.

Samples

input

```
6
1 2
2 3
3 4
4 5
5 6
```

output

```
10
18
22
22
18
10
```

input

```
5
4 5
2 1
3 2
2 5
```

output

```
8
18
8
8
14
```

 licensed under CC BY-SA 3.0

Cutting of a birthday cake

task: `cutting`	input file: `text files`	output file: `text files`
points: `100`	time limit: `-`	memory limit: `-`

Mimino is celebrating his birthday! Although an incredible programmer, he is only N years old. Kamila's compiler (remember from yesterday?) cooked a huge[3] chocolate cake for his birthday party and placed N candles on the top of it. Mimino and his friends are all getting really hungry and want to cut the cake. They want to cut the cake with the straight cuts, such that there will be at most one candle on each slice.

Mimino is now wondering, what is the minimum number of cuts needed to satisfy above constraints.

Task

Having a plane and some points in it, give a set of lines that separate the plane into subplanes so that each subplane can contain at most 1 point. The set should be as small as possible.

Point coordinates are all integers. Lines are given by two different points (with integer coordinates). Lines have a direction from their first point to their second point; this is used to determine that when a line intersects an input point, the input point is considered to be on the right side of the line.

Lines can only be horizontal, vertical or diagonal.

Input

The first line contains a single integer N, the number of points. Following N lines describe the points. The $(i+1)$-th line contains a pair of space separated integer coordinates x and y of the i-th point.

Output

The first line should contain L, the number of lines. Following L lines describe the lines.

The $(i+1)$-th line contains four space separated integers X_1, Y_1, X_2, Y_2. Points (X_1, Y_1) and (X_2, Y_2) must be different, the line goes through both of them. At least one of the following conditions must held:

- $X_1 = X_2$ (vertical line)
- $Y_1 = Y_2$ (horizontal line)
- $X_1 - X_2 = Y_1 - Y_2$
- $X_1 - X_2 = Y_2 - Y_1$

Your solution should output at most 10 000 lines and the absolute value of all coordinates must be less than or equal to 1 000 000.

Scoring

This is an open data problem. You can download all 10 testcases from the submission system. You are only required to submit your output files.

If your output doesn't follow the output format or doesn't separate the points correctly, your score will be zero for the testcase.

[3] The cake is so big, it can be represented as an infinite plane.

 licensed under CC BY-SA 3.0

Otherwise, your score for the testcase is equal to $10 \cdot \left(1 - \sqrt{1 - L_{min}/L}\right)$, where L is the number of lines in your output and L_{min} is the best submission made by any contestant during the contest. Note that this score is calculated after the contest.

Samples

input

```
4
3 1
4 5
6 6
8 4
```

output

```
2
3 2 8 7
2 8 8 2
```

 licensed under CC BY-SA 3.0

Investigation

task: `investigation`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `2500 ms`	memory limit: `1 GB`

There was a robbery in a city of Bytelandia. The thief successfully escaped and hid somewhere in the city. You are investigating this crime. Your goal is to find and arrest the thief.

The city consists of N houses and $N-1$ roads, connecting some of the houses in such a way, that there exists a unique path between any two houses (i.e. the city forms a tree structure). The thief is hiding in one of the houses.

To locate the thief, you can choose a house h and search it. If it was the house where the thief was hiding, you arrest him. Otherwise you interrogate the inhabitants of the house and they provide you with the following information: "If you picture the city as a rooted tree, with house h as its root and houses $c_1, c_2 \dots c_m$ as the children of h, then the thief is hiding in one of the houses of the subtree rooted at c_i (for some i, $1 \le i \le m$)."

You have to keep searching the houses until you find and arrest the thief. You can suppose that the thief stays hidden in the same house during the whole investigation process (i.e. he doesn't change the location).

Obviously, the order in which you search the houses matters, because even if you don't find the thief in a house, with the provided piece of information you can highly reduce the number of possible houses where the thief can be hiding. So you need to come up with an optimal strategy that minimizes the number of searched houses in the worst possible scenario.

Task

You are given the description of the city. Come up with a strategy for searching the houses, that minimizes the number of houses you need to search in the worst possible scenario.

Input

First line contains a single integer N, the number of houses in the city (houses are numbered from 0 to $N-1$).

Second line contains $N-1$ space separated integers, $v_1 v_2 \dots v_{N-1}$. Integer v_i ($1 \le i \le N-1$) means there is a road connecting the houses with numbers v_i and i ($v_i < i$).

It holds $2 \le N \le 10^5$.
In the 20% of testcases $N \le 10$.
In the 40% of testcases $N \le 20$.
In the 60% of testcases $N \le 1000$.

Output

Output exactly one integer, the number of houses you need to search in the worst possible scenario, when searching using the optimal strategy.

Samples

input

```
5
0 1 1 1
```

The city looks like a star, with house 1 in the middle.

output

```
2
```

First search the house 1. If the thief wasn't there, after interrogation you will know in which house is the thief hiding.

 licensed under CC BY-SA 3.0

input	output
8 0 1 2 1 3 5 6	3

 licensed under CC BY-SA 3.0

New Tree

task: `newtree`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `200 ms`	memory limit: `1 GB`

A new tree has been planted in the city park and the gardener wants to protect it. To do so, he creates a protected area around the new tree by selecting three of the old trees, and encircling them with a band. The new tree must be strictly inside the protected area but no other tree is allowed to be there. The gardener has already selected one of the old trees. Help him find the other two.

Task

The task is to compute two old trees that form a valid protected area together with the already selected old tree.

Input

The first line of the input contains two integers, N and A. N ($3 \leq N \leq 100000$) is the number of old trees and A ($1 \leq A \leq N$) is the identifier of the preselected old tree. The trees are identified by the numbers $1, \ldots, N$. The second line contains two integers x and y, the x-and y-coordinates of the new tree. Each of the next N lines contains two integers x and y, ($-1000000 \leq x, y \leq 1000000$) the coordinates of an old tree.

In 40% of the testcases $N \leq 5000$ also holds.

Output

The first line of the output must contain two integers B and C separated by a single space, where B and C are old tree identifiers with the following property: if A is the identifier of the preselected old tree, then the triangle with nodes A, B and C (in counterclockwise order) forms a valid protected area. That is, there are no trees on the sides of the triangle other than A, B and C, and the only tree strictly inside the triangle is the new tree. If there is no solution then the output must be 0 0. If there are multiple solutions, your program should output only one; it does not matter which one.

Samples

input

```
7 1
9 3
3 1
8 7
9 5
11 5
12 4
9 1
13 6
```

output

```
6 4
```

 licensed under CC BY-SA 3.0

Cubic Art

task: rubik	input file: stdin	output file: stdout
points: 100	time limit: 1200 ms	memory limit: 1 GB

Modern art is unpredictable. When Bob was tidying his room he found his old Rubik's cube. Then the moment came. He closed his eyes, listened to his inner voice, made a few moves (up to 65 000) and the masterpiece was nearly complete. But the final state was not to his liking. He realized he did some of the moves incorrectly. If only he could go back in time and change them!

All he now needs to do is a few changes (again, up to 65 000 of them). Each of the changes consists of replacing one move with some other move. Bob would like to see what each change does. But it is annoying to repeat the entire sequence of moves again and again.

Task

You are given the initial state of Bob's Rubik's cube. (The cube is not necessarily solved in its initial state.) You are also given the original sequence of moves Bob performed.

Finally, you are given a sequence of changes. Each change is of the form "change the k-th move into this new move". For each change, output the state of the cube *at the end* of the entire sequence of moves.

Note that the changes are permanent – for example, the second change should be applied to the sequence of moves with the first change, not to the original sequence.

Cube's I/O

Let the cube's colors be A,B,C,D,E,F. When you are playing with the cube, the middle squares of its faces do not move. Therefore, we will always use A as the color of the center of the top face, B, C, D, E as the centers of the side faces (in order), and F as the center of the bottom face. The surface of the cube can then be unfolded into the following form:

```
???
?A?
???
????????????
?B??C??D??E?
????????????
???
?F?
???
```

Input

First 9 lines of the input contain the description of the starting state of the cube. The description is given in the above form. You may assume that the centers of the six faces are labeled as shown above.[4]

Then there is a line with two integers n and m: n is the number of moves and m is number of subsequent changes.

Next n lines are describing Bob's original moves. They have the form "C_i d_i", where C_i is the color of the center of the rotated side and d_i is -1 in case of a clockwise move and 1 in case of a counterclockwise move.

[4] You may also assume that the starting state is a valid configuration that can be obtained from the solved state. However, this is actually irrelevant in our problem.

 licensed under CC BY-SA 3.0

The last m lines describe the changes, in order. Each one has the form "a_j C_j d_j", where a_j is the (1-based) index of the move that is being replaced and C_j d_j describes the new move.

Constraints

In all test cases, $n, m \leq 65\,000$. In 50% of testcases $n, m \leq 1000$.

 licensed under CC BY-SA 3.0

Output

Let S_i be the sequence of moves obtained from the initial sequence by applying the first i changes.

For each i between 1 and m, inclusive, output 9 lines: the final state of the cube obtained by starting in the initial configuration and performing the sequence of moves S_i. Use the same format as in the input.

Samples

input

```
AAA
AAA
AAA
BBBCCCDDDEEE
BBBCCCDDDEEE
BBBCCCDDDEEE
FFF
FFF
FFF
8 4
E 1
E -1
F 1
F -1
B 1
B -1
E 1
E -1
8 C -1
2 C -1
6 D -1
4 A -1
```

The original moves cancel each other out. At the end of the original sequence of moves, the cube is back in its initial state.
After we make the four changes described in the input, we obtain a sequence of moves that flips everything except for the centers of all sides.

output

```
BAB
BAB
BAB
FBFCCCADAEEE
FBFCCCADAEEE
FBFCCCADAEEE
DFD
DFD
DFD
FAF
FAF
FAF
DBDCCCBDBEEE
DBDCCCBDBEEE
DBDCCCBDBEEE
AFA
AFA
AFA
FCF
BAB
FCF
EFEDFDCACBAB
DBDCCCBDBEEE
EFEDFDCACBAB
AEA
DFD
AEA
CCC
CAC
CCC
FFFDDDAAABBB
FBFDCDADABEB
FFFDDDAAABBB
EEE
EFE
EEE
```

 licensed under CC BY-SA 3.0

Universities

task: `universities`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1000 ms`	memory limit: `1 GB`

You have just graduated from the high school and are looking for a university to enroll in. There are N magic universities in Bytelandia. Each university teaches either black magic or white magic. There are $N-1$ bidirectional roads between the universities, each road connecting two different universities. Universities are connected in such a way, that there exists a unique path between any two universities.

You plan to visit some of the universities. Each university has a happiness factor, by which your overall happiness increases when you visit the university. Note that if the happiness factor is negative, your overall happiness decreases.

To plan your trip, you choose two different universities, the departure and the destination university. You will visit all the universities on the path between the departure and destination universities, both of them including. To keep things in balance, you must visit the same number of white magic universities as black magic universities.

You are now wondering, what is the optimal trip that maximizes the happiness of the trip, i.e. sum of the happiness factors of the universities you visit.

Task

You are given the description of universities and connections between them. Find the optimal trip around the universities, in which you visit the same number of black and white magic universities and the happiness of the trip is as large as possible.

Input

Input consists of four lines.

First line contains a single integer N ($2 \leq N \leq 10^5$), number of universities (universities are numbered from 1 to N).

Second line contains a string of length N, consisting of “B” and “W” characters. If the i-th character is “B”, then the i-th university is teaching black magic. If the i-th character is “W”, then the i-th university is teaching white magic. There will be at least one university teaching white magic and at least one university teaching black magic.

Third line contains N space-separated integers $h_1 h_2 \ldots h_N$ ($-10^5 \leq h_i \leq 10^5$). Integer h_i is the happiness factor of the i-th university.

Fourth line contains $N-1$ space-separated integers $v_1 v_2 \ldots v_{N-1}$. Integer v_i means there is a road connecting the universities with numbers v_i and $(i+1)$ ($1 \leq v_i \leq i$).

Output

Output exactly one integer, maximum overall happiness of the trip, in which you visit the same number of black and white magic universities.

Samples

input

```
6
BWBBBW
6 0 3 -2 100 5
1 2 2 4 4
```

output

```
9
```

In optimal trip you visit the universities 1,2,4,6.

 licensed under CC BY-SA 3.0

input	output
3 WBW 1 -10 5 1 2	-5

Because you have to visit some universities, sometimes the answer can be negative.

 licensed under CC BY-SA 3.0

Wall

task: `wall`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1000 ms`	memory limit: `1 GB`

Mirek is a conservator. His job is to maintain monuments, keeping them in good condition. Mirek's today task is to repair the defense wall of an old fortress. The wall is about to fall to pieces, so he must hurry. He searched through the Internet and found a robot designed to repair such walls extremely fast. After purchasing the robot, he tried to make the optimal repairment plan, but this was way too difficult for him.

The wall can be considered as a straight line. Mirek wrote down all points on the wall, which require repairing. For each point he knows, what would be the cost C_i of repairing it, and a coefficient D_i – how would the cost increase if it wasn't repaired immediately. If the i-th point will be repaired after time t, then the cost of repairing it would be equal to:

$$C_i + t \cdot D_i$$

Task

Given the coordinates of all points on the wall and the initial position of the robot, and knowing that transportation of the robot from point x_1 to point x_2 would take $|x_1 - x_2|$ time[5], calculate the minimum cost of repairing all spoiled points. You can assume that repairing a single point takes no time.

Input

On the first line of input there are two integers N and P ($1 \leq N \leq 2\,000$, $0 \leq P \leq 10^9$) – the number of points on the wall, which should be repaired, and the initial position of the robot. Then, N lines follow, i-th of these lines describes i-th point on the wall and contains three integers X_i, C_i and D_i ($0 \leq X_i \leq 10^9$, $0 \leq C_i, D_i \leq 10^6$, $X_i \neq P$) – the position of the point and cost coefficients. There are no two points with equal positions.

Output

Output a single line with integer C, where C is the minimal cost of repairing all points on the wall.

Sample

[5] Here, x_1 and x_2 are the coordinates of these two points (not their indices).

 licensed under CC BY-SA 3.0

input

```
3 7
10 32 1
3 5 1
14 0 2
```

output

```
72
```

The optimal plan of repairing points is:

- *Transport robot from point* 7 *to point* 10 *and repair the first point after time* 3*.*
- *Transport robot from point* 10 *to point* 14 *and repair the third point after time* $3 + 4 = 7$*.*
- *Transport robot from point* 14 *to point* 3 *and repair the second point after time* $3+4+11 = 18$*.*

Thus, the total cost of repairing the wall is $5 + 18 \cdot 1 + 32 + 3 \cdot 1 + 0 + 7 \cdot 2 = 72$*.*

 licensed under CC BY-SA 3.0

Critical Projects

task: `critical`	input file: `stdin`	output file: `stdout`
points: 100	time limit: 600 ms	memory limit: 32 MiB

A large project is subdivided into N different subprojects. The manager of the project established precedence relations among the subprojects. This means that there are pairs of subprojects u and v such that the completion of the subproject u must be finished before the start of the subproject v. In this case we say that u directly precedes v. We say that u precedes v if u directly precedes v or there is a subproject z such that u precedes z and z precedes v. Any subproject u is considered critical if for each subproject v (other than u) either v precedes u or u precedes v. It is known that the whole project can be completed, e.i., there is no subproject u such that u precedes itself.

Task

Write a program that computes all the critical subprojects.

Input

The first line of the input contains two integers, N and M. N ($1 \leq N \leq 100000$) is the number of the subprojects and M ($0 \leq M \leq 1000000$) is the number of the direct precedence pairs. Subprojects are identified by the numbers $1, \ldots, N$. Each of the next M lines contains two integers u and v, ($1 \leq u \neq v \leq N$) a direct precedence pair, that is u directly precedes v.
In 40% of the testcases $N \leq 5000$ and $M \leq 30000$ also hold.

Output

The first line of the output must contain the number of critical subprojects. The second line contains the identifiers of the critical subprojects in ascending order. The numbers must be separated by a single space. If there is no critical subproject then the first and only line contains the number 0.

Samples

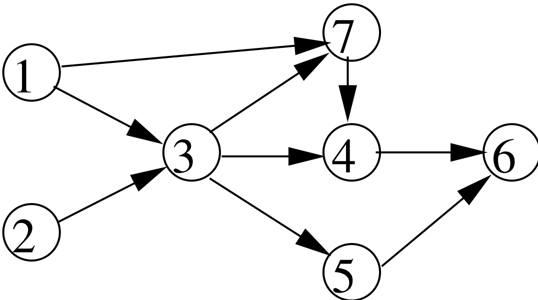
input

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

output

```
2
3 6
```


licensed under CC BY-SA 3.0



 licensed under CC BY-SA 3.0

Connect Highways

task: `networks`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `400 ms`	memory limit: `32 MiB`

In Byteland, there are two highway networks operated by two companies: Red and Blue. Both networks consist of junction points and straight lines connecting pairs of junction points, called segments. Any two segments are non-crossing, meaning that they can only touch at junction points. Both networks are connected, that is any two junction points are connected through a series of consecutive segments. Moreover, the two systems are disjoint, i.e., no junction point appears in both networks. The two companies have now decided to fuse into a single one, and they want to connect their networks by building a straight line segment between two junction points, one in each network. The new segment cannot cross any existing segment.

Task

Write a program that computes a suitable connecting segment.

Input

The input contains the description of the Red, followed by the description of the Blue network. The first line of the description contains two integers N ($2 \leq N \leq 200000$) and M ($1 \leq M \leq 700000$). N is the number of the junction points and M is the number of the segments. Each of the following N lines contains two integers x and y ($-1000000 \leq x, y \leq 1000000$), which are the coordinates of a junction point. Each of the following M lines contains two integers p and q ($1 \leq p \neq q \leq N$), the endpoints of a segment. Junction points are identified by the numbers $1, \ldots, N$ in the order of their appearance in the input.

In the 30% of the testcases the number of the junction point and the number of the segments are not larger than 3000 in both networks.

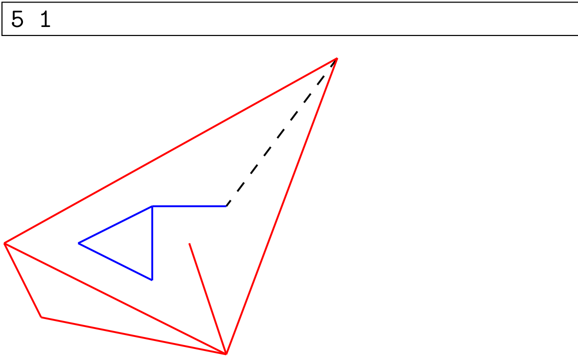
Output

The first and only line of the output contains two integers u and v, the endpoints of a connecting segment. That is, u is junction point of the Red, v is a junction point of the Blue network and the line segment with endpoints u and v crosses no segment of any of the networks. If there are multiple solutions, your program should output only one; it does not matter which one.

Samples

 licensed under CC BY-SA 3.0

input	output
5 6 0 3 1 1 6 0 5 3 9 8 1 2 1 3 4 3 3 5 1 5 2 3 4 4 6 4 4 4 4 2 2 3 1 2 4 2 2 3 3 4	5 1



 licensed under CC BY-SA 3.0

Next Permutation

task: `nextperm`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `100 ms`	memory limit: `32 MiB`

Permutations are intensively studied in mathematics and computer science. Pattern avoiding permutations are of special interest. A permutation $p_1, p_2, \ldots, p_n$ of the natural numbers $1, \ldots, n$ is called 3-1-2 pattern avoiding if there are no three indices $1 \leq i < j < k \leq n$ such that $p_i > p_j$, $p_i > p_k$ and $p_j < p_k$.

Task

Write a program that computes for a given 3-1-2 pattern avoiding permutation the next 3-1-2 pattern avoiding permutation according to the lexicographic ordering.

Input

The first line of the input contains one integer n ($3 \leq n \leq 10000$). The second line contains n positive integers separated by single spaces, a 3-1-2 pattern avoiding permutation of the natural numbers $1, \ldots, n$. The input is not the decreasing sequence $n, n-1, \ldots, 1$.

Constraints

In the 40% of the testcases $n \leq 1000$ also holds.

Output

The first line of the output must contain the 3-1-2 pattern avoiding permutation that follows the input permutation in the lexicographic ordering. The numbers must be separated by a single space.

Samples

input	output
`5` `2 4 5 3 1`	`2 5 4 3 1`

 licensed under CC BY-SA 3.0

Game

task: game	input file: stdin	output file: stdout
points: 100	time limit: 100 ms	memory limit: 1 GB

Mirek really likes playing with numbers. Together with his friend, Kamil, he plays a following game. At the beginning, there are two non-negative integers – A and B. Let's say $A \leqslant B$. The players can perform one of two moves in turns:

- Replace B with $B - A^K$. Number K can be any integer chosen by the player, considering the limitations that $K > 0$ and $B - A^K \geq 0$.
- Replace B with $B \; mod \; A$.

If $B \leqslant A$, similar moves are possible. The player who changes any number into 0, wins. Mirek always starts. He likes this game, but he likes winning much more. Help him determine who will win, if both of them play optimally.

Task

You are given the description of games played by Mirek and Kamil. For every game determine who will win, Mirek or Kamil.

Input

First line contains an integer T ($1 \leq T \leq 10^4$), the number of games played by boys. In the next T lines, there are descriptions of those games. Every such line contains two integers A, B ($1 \leq A, B \leq 10^{18}$)

In the 30% of testcases $A, B \leq 1000$.

Output

Output T lines. The i-th line should contain the name of the player who wins the i-th game, `Mirek` or `Kamil`.

Samples

input

```
4
1 1
12 4
4 6
15 31
```

output

```
Mirek
Mirek
Kamil
Mirek
```

 licensed under CC BY-SA 3.0

Posters

task: `posters`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `2000 ms`	memory limit: `1 GB`

Mirek is a devoted fan of his favorite music band. He attends every concert and collects their posters. Each time, when he gets a new poster, he hangs it on the wall, above his bed. After many years of collecting posters, almost the whole wall has been covered with them and now Mirek cannot find space for the new ones. He just got some new posters to hang and he needs your help to find the best place for them on the wall. For each poster and its placement, Mirek would like to know how much this poster would cover other posters.

Task

You are given the coordinates of the posters which already hang on the wall and the coordinates of the posters which do not hang yet, but Mirek considers hanging them. For each new poster, find the area of the parts of hanging posters which would be covered directly by this poster.

The posters on the wall may overlap, and if their intersection is covered, you shouldn't count its area twice.

Input

On the first line of input there is one integer N ($1 \leq N \leq 100\,000$) – the number of posters which hang on the wall. In the next N lines, there are descriptions of those posters. In $N + 2$ line there is one integer M ($1 \leq M \leq 100\,000$) – the number of new posters which Mirek would like to hang on the wall. In the next M lines, there are descriptions of those posters.

Each poster is a rectangle with edges parallel to axis. It is described by four integers x_1, y_1, x_2, y_2 ($0 \leq x_1 < x_2 \leq 10^9$, $0 \leq y_1 < y_2 \leq 10^9$), denoting the coordinates of the bottom left corner and the top right corner.

In 12.5% of testcases $n, m \leq 10$, the coordinates are not greater than 100.

In 25% of testcases $n, m \leq 50$.

In 50% of testcases $n, m \leq 1000$.

In 50% of testcases the coordinates are not greater than 30 000.

Output

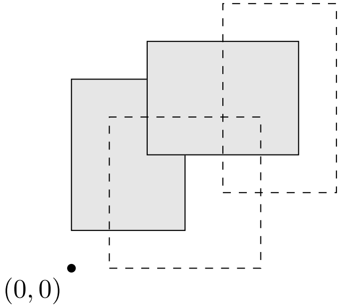
For each new poster output one line containing an integer – the answer for the Mirek's problem. The answers should be printed in the same order as the posters were given in the input.

 licensed under CC BY-SA 3.0

Sample

input	output
2 0 1 3 5 2 3 6 6 2 1 0 5 4 4 2 7 7	8 6

Mirek's wall is presented on the picture below. Dashed rectangles are the new posters, and filled rectangles are the posters which have been already hanged.



 licensed under CC BY-SA 3.0

Sorting

task: `sorting`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1000 ms`	memory limit: `1 GB`

Your friend, Mirek, had some files containing integers. He had to sort integers in each file in ascending order. Mirek is an IT specialist so, of course, he tried to find a command line tool that would do his task. The name of a tool wasn't hard to guess, but it didn't work as Mirek expected – after sorting the files, he realized that this tool was treating every integer as a string and it sorted them lexicographically. He knew that such a thing could happen, but he was surprised anyway – these files were still sorted in ascending order.

Now, Mirek wonders how lucky he was and how was even possible that integers from these files could have had same lexicographical and numerical order. Help him satisfy his curiosity.

Task

Given a range of integers $[A, B]$, determine the number of subsets of those integers, that their lexicographical and numerical orders are equal.

Input

On the first and only line of input there are two integers A and B ($1 \leq A \leq B \leq 10^{18}$, $B - A \leq 10^5$).

Output

Output a single line with integer M, where M is the number of subsets of set $\{A, A+1, \ldots, B\}$, which keep specified condition. As the answer may be really big, output it modulo $10^9 + 7$.

Sample

input	output
`98 101`	`7`

Those subsets are: $\emptyset$, $\{98\}$, $\{99\}$, $\{100\}$, $\{101\}$, $\{98, 99\}$, $\{100, 101\}$.

licensed under CC BY-SA 3.0

Mission

task: `mission`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `1000 ms`	memory limit: `1 GB`

Attention soldier!

I have a special task for you. We have detected an enemy base and it needs to be destroyed. You will be given a map and enough bombs to blow it up. After the action a helicopter will be waiting for you in the forest nearby.

Sounds easy, doesn't it? Find the fastest way to achieve the goal and make sure you don't visit any place twice, otherwise you will be detected.

Is everything clear? Very well... get ready, because you are leaving in 10 minutes!

I wish you good luck, don't get killed and see you at the dinner.

Task

You are given an undirected graph and its three different vertices: your base, enemy base and the place where helicopter is waiting. Find the shortest path in the graph from your base to the helicopter's place. The path must go through the enemy base and can't visit any vertex twice.

Input

First line of the input contains five space separated integers N, M, B, E, H ($1 \leq B, E, H \leq N$, $B \neq E \neq H \neq B$).

The graph contains N vertices numbered 1 to N, your home base is at vertex B, enemy base at vertex E and the helicopter is waiting at vertex H.

Following M lines describe the edges of the graph. Each line contains three space separated integers v, w and t ($1 \leq v, w \leq N$, $v \neq w$, $1 \leq t \leq 1\,000\,000$). It means there is an undirected edge connecting vertices v and w and it costs t units of time to traverse the edge.

No two vertices are connected by more than one edge.

It holds $3 \leq N \leq 1\,000$ and $0 \leq M \leq 1\,000$. In at least 30% testcases $N \leq 20$.

Output

Output a single line with a single integer, the least amount of time needed to complete the mission. If it is impossible to complete the mission, output -1 instead.

Samples

input

```
3 2 1 2 3
1 2 10
2 3 20
```

output

```
30
```

input

```
3 0 2 1 3
```

output

```
-1
```

 licensed under CC BY-SA 3.0

input	output
4 4 3 2 4 2 3 5 3 1 1 1 4 1 2 4 100	105

 licensed under CC BY-SA 3.0

Newspapers

task: `newspapers`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `4000 ms`	memory limit: `1 GB`

In the beautiful Polish city called Mirkow, there are n intersections which are connected by $n-1$ bidirectional streets. It is possible to travel between every pair of intersections using these streets.

Mirek works in Mirkow as a paper boy and he delivers newspapers to the villagers. For every street it is known how many people live there (it is also the number of newspapers he should deliver on that street). Each day he chooses two intersections and visits every house on the shortest path between them. Mirek's daily salary is proportional to the average number of newspapers delivered on a single road (the number of delivered newspapers divided by the number of traversed roads).

At first, Mirek tried to be smart by delivering newspapers only on the road with the highest number of habitants, but the boss found out what his strategy was and tried to stop it, because too many people were not getting their newspapers. The boss gave Mirek an additional constraint: he had to choose a route containing not less than k roads.

Help Mirek and find the optimal path for him.

Task

Given the description of Mirkow, find a path containing k streets or more with the highest number of delivered newspapers per road.

Input

On the first line of input there are two integers n and k ($1 \leq n \leq 50\,000$, $1 \leq k \leq n-1$) – the number of intersections in Mirkow and the minimal length of Mirek's path. On the next $n-1$ lines there are descriptions of streets. Each description consists of three integers a, b and c ($1 \leq a < b \leq n$, $0 \leq c \leq 10^6$) – they mean that there is a street connecting intersections a and b and on this road there live c people.

You can assume that there exists a path of length k.

In tests worth 30 points: $n \leq 1\,000$.

Output

Output one number – the average number of delivered newspapers on a single road in the optimal route. The output will be considered correct if the difference between it and the correct answer is less than 10^{-6}.

Sample

input

```
5 2
1 2 4
2 3 1
3 4 3
3 5 3
```

output

```
3.00000000
```

 licensed under CC BY-SA 3.0

An inexperienced slalomer

task: `slalom`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `300 ms`	memory limit: `1 GB`

It is Hubert's first slalom race, so naturally he feels very nervous. Moreover, it is also his first day on skis and he has not learnt how to make turns yet – he can only slide along a straight line. But nothing is lost yet. Maybe the designer of this particular course was not careful enough, so it is actually possible to pass through every gate without taking a single turn.

Task

You are given the description of a slalom course with n gates. The course runs from left to right. Each gate is represented by a vertical line segment between two poles. From a bird's-eye view, Hubert looks like a disk with diameter d ($d \geq 0$) and the trajectory of his center follows a straight line. He can choose his start point anywhere to the left of the leftmost gate, and his finish point anywhere to the right of the rightmost gate. To complete the course, Hubert must pass with his entire body between the poles of all gates. Touching the poles is allowed.

Find the largest diameter d such that there is a trajectory enabling Hubert to complete the course.

Input

The first line of the input contains a single integer n – the number of gates ($1 \leq n \leq 10^5$). Each of the n following lines describes a gate and consists of three space-separated integers x, y_1, y_2 ($0 \leq x \leq 10^9$, $0 \leq y_1 \leq y_2 \leq 10^9$). The described gate is the vertical line segment with endpoints $[x, y_1]$ and $[x, y_2]$. No two gates have the same x-coordinate.

Output

Output a single line with the largest d such that Hubert can complete the course. We will accept answers with absolute or relative error less than 10^{-9}. In C++, you can output the answer using `printf("%.10lf\n", d);`

If there is no such non-negative value of d, output a single line with word `Impossible`.

Samples

input

```
3
4 3 7
6 6 9
1 5 10
```

output

```
1.3728129460
```

input

```
2
3 7 9
10 4 4
```

output

```
0.0000000000
```

input

```
3
0 4 7
2 0 3
4 4 7
```

output

```
Impossible
```

 licensed under CC BY-SA 3.0

Tickets

task: `tickets`	input file: `stdin`	output file: `stdout`
points: `100`	time limit: `800 ms`	memory limit: `32 MiB`

The match of the year will be played next week. There are N seats in the stadium numbered by the integers 1 to N. Each fan can request one ticket and can specify the range of seats where he would be willing to sit. The range is specified by two integers F and L, the first and last seat in the range, respectively. This means that the fan accepts any seat S such that $F \leq S \leq L$ holds. The ticket office has already received M requests from football fans and wants to select the maximal number of requests that can be simultaneously satisfied.

Task

Write a program that computes the maximal number of fans that each can obtain a ticket with a suitable seat, and gives an adequate seat assignment. No two fans can get the same seat.

Input

The first line of the input contains two integers N ($1 \leq N \leq 100000$), the number of seats, and M ($1 \leq M \leq 1000000$), the number of requests. The seats are numbered by $1, \ldots, N$. Each of the next M lines contains two integers F and L ($1 \leq F \leq L \leq N$), a request specification. Requests are identified by the numbers $1, \ldots, M$ in the order of their appearance in the input.

Output

The first line of the output must contain one integer K, the maximal number of the selected requests. Each of the next K lines contains two integers S R, a seat assignment, where S is a seat number and R is the number of the request obtaining the seat S. The seat assignments can be given in any order. If there are multiple solutions, your program should output only one; it does not matter which one.

Samples

input

```
10 9
1 3
2 4
5 7
2 6
1 5
3 7
4 8
7 9
3 8
```

output

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

 licensed under CC BY-SA 3.0

Hyperways

Problem author: Marián Horňák
Task preparation: Marián Horňák
Solution writeup: Michal Forišek, Marián Horňák

Formally, planets are graph vertices, hyperways are edges and unsafe hyperways are bridges in the graph. Graph is not directed but can contain multi-edges and self-loops. Our task is to output how many edges are not more bridges after each edge addition.

Slow solutions

Basic idea of slow solutions is to recalculate number of bridges after each edge addition. You can do it in quadratic time deleting each edge and running BFS / DFS to find out whether it has changed the number of components. (The edge is bridge if and only if you split the graph deleting it.) This solution runs in $O(m^3)$ time.

To improve it, you can find all the bridges running a single DFS. You need to remember the "time" you first visited each vertex. It is called visit time. Time can be for example the number of vertices you visited before. When returning from vertex v back to vertex u using some edge e, consider the earliest visit time you have met since you entered e. Lets call it return time. Return time can be calculated as the minimum of visit time of v and the return times of all the other edges incident to v. If this wasn't first visit of v, we returned directly and it is just v's visit time.

If the return time of e is higher than (or equal to) the first visit time of u, vertices u and v can be connected through the vertex this time belongs to and thus e is not a bridge. Otherwise, part of graph we visited has no other connection to the rest and e is a bridge. Therefore, we can easily decide whether e is a bridge or not. Solution works in $O(m^2)$ time.

Union Find algorithm

Union Find algorithm is a prerequisite to the optimal solution Since it is well known algorithm, I just mention it briefly.

Imagine you have a number of disjoint sets of elements. Union Find allows you to perform two operation on these sets: join two sets into one and find out the set given element is in. It works in amortized time complexity $O(k \cdot \alpha(k))$, where k is the number of operations and α is the inverse of extremely fast-growing Ackermann function.

Optimal solution

Union Find can be used to determine the component vertex belongs to. When adding an edge that connects two components, this edge must be a bridge and thus the number of edges that are no more bridges is 0.

Problem is with edges that connects vertices from the same component. Suppose, the first such edge was added. Since, it is the first one, graph was a forest before and this edge creates a cycle c. All the edges in the cycle are no more bridges, so we can output cycle size.

Imagine we have added few new edges and there is at least one edge e incident to some vertex of the cycle c. Now we move e to the another vertex of the cycle c. This operation will not create or remove any new cycles, because all the cycles that may be affected intersect with c before as well as after the

 licensed under CC BY-SA 3.0

operation. Therefore, this operation does not bridgeness of any edge. This allows us to do not care about which cycle we are using and thus merge cycle c into one vertex.

Since c is the only cycle in the time of its creation, merge will make our graph the forest again. Repeating this, we can process all the vertices.

The only question is, how to merge the cycle.

Cycle merging

Union Find can be used to merge the vertices. The problem is to calculate the cycle.

Consider the spanning forest of the complete graph, created by by skipping all the edges, that would create a cycle. This can be easily precalculated (using Union Find to check whether next edge will be bridge). Then one can select root in every tree of the forest and use DFS to direct the edges to the root.

When the first non-bridge edge comes, all edges in the real graph are bridges. Therefore, they are subset of the edges in the precalculated directed spanning forest and the created cycle is the same in both graphs. In the directed graph we are able to find the lowest common ancestor (LCA) of connected vertices which is the top vertex of the cycle.

LCA can be found by repeatedly moving deeper vertex one generation up (depths can be also precalculated).

After the cycle is found, vertices are merged also in precalculated graph. This does not affect the the structure of the graph and we can suppose they were never split so the above argumentation can be used again.

Every merge decreases the number of mergeable elements by one. There are three levels of merges: precalculation, components and vertex merges. The number of Union find requests is linear to the number of edges and the number of merges. DFS runs in linear time. Finding LCA visits each vertex once (then is merged). Therefore, the time complexity is $O(m \cdot \alpha(m))$.

 licensed under CC BY-SA 3.0

Dynamic memory allocation

Problem author: Peter 'Bob' Fulla
Task preparation: Peter 'Bob' Fulla
Solution writeup: Peter 'Bob' Fulla

It is quite obvious how to process a query of either type in time $O(n)$, but such a solution would certainly fail to achieve the perfect score for this task. We will describe two approaches that lead to algorithms with time complexity $O(\log n)$ per query.

Solution based on segment trees

We build a segment tree on top of the array of bytes. In each node, we store the following information about the segment represented by the node:

- the number of allocated bytes in the segment,
- the length of the longest continuous block of free bytes in the segment,
- the number of consecutive free bytes at the beginning of the segment,
- the number of consecutive free bytes at the end of the segment.

Note that we can easily compute these values for a node from the values stored in its two children.

To process an allocation query, we first check whether the longest block of free bytes in the entire array would be long enough to satisfy the query. If it is too short, we return -1. Otherwise, we need to find the leftmost block of length at least ℓ. This can be done using recursion: If the longest block in the left-hand half of the array is of length ℓ or more, we continue looking for the answer there. If this is not the case, we check whether we obtain a sufficient block of free bytes by concatenating the block at the end of the left-hand half of the array with the block at the beginning of the right-hand half. If this option also fails, the sought block must lie completely in the right-hand half of the array.

After we find the position of the block to allocate, we need to change the state of bytes in it and make necessary updates to the values in the segment tree. However, the number of bytes allocated in a query may reach $\Theta(n)$, so we must employ a technique called lazy propagation: Whenever we want to allocate all bytes in the segment corresponding to a node of the segment tree, we simply mark the node with a flag meaning "all bytes below are allocated", update the values stored in it, and do not descend to its subtree at all. When processing a subsequent query, we may enter a marked node. Only at that time we unmark the node and propagate the flag to its two children.

Freeing a block of bytes is very similar to allocating, we just need a different flag meaning "all bytes below are free". To compute the number of allocated bytes before a query, we simply add up the corresponding values stored in nodes of the segment tree.

The lazy propagation technique guarantees that we visit at most $2 \log n$ nodes when updating the tree. Therefore, the time complexity is $O(\log n)$ per query.

Solution based on self-balancing binary search trees

This approach requires us to implement a customized self-balancing binary search tree, for example a treap. The nodes of the tree will represent maximal blocks of consecutive free bytes and they will be ordered by their positions in the array. In addition, we need to maintain the following values in every node:

 licensed under CC BY-SA 3.0

- the sum of lengths of blocks in the node's subtree,
- the maximum of these lengths.

Using the precomputed maxima of subtrees, we can easily find the leftmost block with length ℓ or more in time $O(\log n)$. To completely process an allocation query, we just need to update the found block (or remove it, if its length equals exactly ℓ).

Dealing with freeing queries is now a bit more complicated, as they are not necessarily aligned with blocks of free bytes. We split the tree before the first block and after the last block contained by the freed interval, update the adjacent blocks if necessary, and merge the parts of the tree together. These operations can be implemented with time complexity $O(\log n)$.

 licensed under CC BY-SA 3.0

Shades of the town

Problem author: FIXME Problem Setter
Task preparation: FIXME Testdata Developers
Solution writeup: Michal 'misof' Forišek

We will first discuss a simple brute-force solution. Then, we will show how to reduce this problem to a standard string matching problem, and then we will solve that problem.

Solution using brute force

Given a pattern of length ℓ_i, we can simply try all $m - \ell_i + 1$ possible offsets between the pattern and the sequence of shadows. For a fixed offset, we simply verify whether all pairs (building,shadow) give the same scaling factor. This can be done in $O(\ell_i)$.

Thus, the total time complexity of the brute force solution is $O(m \cdot \sum \ell_i)$.

Reduction to string matching

The main trick in solving this task is getting rid of the scaling. To do that, we can make the following observation: the only thing that matters are ratios of consecutive building heights / shadow lengths.

Here's the same thing formally.

Notation: Given a sequence $A = (a_1, \ldots, a_k)$ of positive reals, the sequence $\rho(A)$ (read "ratios of A") is defined as follows: $(a_2/a_1,\ a_3/a_2,\ \ldots,\ a_n/a_{n-1})$.

Lemma: Given two sequences $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$ of positive reals, the sequence B can be obtained by scaling A if and only if $\rho(A) = \rho(B)$.

Proof of the lemma should be obvious.

Thus, instead of solving the original problem, we can compute the ratios of all patterns and look for those in the ratios of the shadow lengths. Thus we get a standard string matching problem: given a long string ("haystack") and a collection of short strings ("needles"), count the number of occurrences of all needles in the haystack. Of course, the *alphabet* we are using are actually *fractions*: each "letter" of our strings is actually a rational number.

This standard string matching problem can be solved optimally using the Aho-Corasick algorithm The time complexity of this solution is $O(m + \sum \ell_i)$, that is, linear in the input size.

A slightly slower but slightly simpler solution is to use an $O(m + \ell_i)$ string matching algorithm (such as KMP or Rabin-Karp hashing) separately for each pattern. With n patterns, this gives us the total time complexity $O(nm + \sum \ell_i)$.

The Aho-Corasick algorithm

The Aho-Corasick algorithm can be seen as a generalization of the Knuth-Morris-Pratt algorithm for a single pattern. In Aho-Corasick we insert all patterns into a trie, and then build a set of back-links that corresponds to the failure function of the KMP algorithm. Formally, for each node v in the trie (other than the root) we have a single back-link that points to the deepest node w that is less deep than v, and has the property that the string that corresponds to w is a suffix of the string that corresponds to v. (I.e., if, after processing some letters of the haystack, we can be in the node v, we can also be in the node w.)

 licensed under CC BY-SA 3.0

Some care must be taken when counting the matches. For example, if we are looking for the needles `abcd` and `bc` in the haystack `pqrabcuvw`, after processing the letters `pqrabc` we will be in the trie node that corresponds to the string `abc` – however, at this moment we should count one occurrence of `bc`. In a full implementation of Aho-Corasick this is handled using a second type of back-links ("output links"), but in our case we can simply precompute this information in linear time once we have the trie with back-links ready.

 licensed under CC BY-SA 3.0

Bus lines

Problem author: Filip Hlásek
Task preparation: Filip Hlásek
Solution writeup: Filip Hlásek

We are given an undirected connected graph with N vertices and $N-1$ edges, therefore a tree.

Let's try to count the number of paths that go through a vertex x. Imagine that the tree is rooted at x. It has k children, each of them is a root of the subtree of size $s_1, s_2, \ldots, s_k$, respectively. The paths that go through x have either x as one of the endpoints or have the endpoints in two different subtrees of x's children. The total count equals:

$$\begin{aligned} numPaths(x) &= 2(N-1) + 2\left(s_1 s_2 + s_1 s_3 + \ldots s_{k-1} s_k\right) \\ &= 2(N-1) + 2 \sum_{1 \le i < j \le k} s_i s_j \\ &= 2(N-1) + \left(\sum_{i=1}^{k} s_i\right)^2 - \sum_{i=1}^{k} s_i^2 \\ &= 2(N-1) + (N-1)^2 - \sum_{i=1}^{k} s_i^2 \\ &= N^2 - 1 - \sum_{i=1}^{k} s_i^2 \end{aligned}$$

We can obtain the same result by considering all $N(N-1)$ paths from which $s_1(s_1-1)+\cdots+s_k(s_k-1)$ don't go through the vertex x.

So far we have counted the number of paths that go through some chosen vertex x. Now we have to do it for every vertex in the tree, but with better time complexity than $O(N^2)$.

Let's root the tree at an arbitrary vertex. We will run a DFS algorithm from the root. For each vertex x it calculates the number of paths that go through x, store it in some global variable, and return the size of the subtree rooted at x. Please note, that to successfully use the above $numPaths(x)$ formula, for each vertex x we have to know the sizes of all the child subtrees if the whole tree was actually rooted at x. In our DFS traversal we are missing the size of one subtree for each non-root vertex. To calculate it, we can use the fact that $\sum_{i=1}^{k} s_i = N-1$.

The complete algorithm is illustrated by the following peseudocode:

```
function DFS(x):
  ans = N * N - 1
  subtree_size = 1
  for v descendant of x:
    s = DFS(v)
    subtree_size += s
    ans -= s * s
  s = N - subtree_size - 1
  ans -= s * s
  // ans contains the number of busses that stops at x
  return subtree_size
```

The time complexity of the algoritm is $O(N)$

 licensed under CC BY-SA 3.0

Cutting

Problem author:	Filip Hlásek
Task preparation:	Challenge24
Solution writeup:	Michal 'Mimino' Danilák

This problem was originally used in Challenge24 competition (year 2013, problem Dissection).

Usually the first thing to do with open-data problems is to visualize the data. The reason for doing it is that most of the time the given data are not random and their structure can be exploited in some ways. This was also true for this problem and small inputs could be solved optimally even by hand. I recommend converting the input data into some simple image file format, like PPM [6], which is pure ASCII format, and can be opened in GIMP.

Our reference solution for this problem is just a basic hill-climbing algorithm:

1. Until the points are not completely separated, add the line which separates most of the points from each other.
2. It can happen that some lines become redundant during the process. So for each line that can be removed, while keeping all the points separated, remove it.
3. Now we have a valid solution. Output the best solution found so far.
4. For each line, remove it with the probability P (in our case 0.05).
5. Go to step 1.

[6] http://en.wikipedia.org/wiki/Netpbm_format

 licensed under CC BY-SA 3.0

Investigation

Problem author: Michal 'Mimino' Danilák
Task preparation: Michal 'Mimino' Danilák
Solution writeup: Michal 'Mimino' Danilák

Terminology

In the following text I will refer to the problem in the terms from graph theory, where the houses are vertices and the roads are edges of the graph, which is also a tree.

By *querying a vertex* we mean to choose some house and search it.

By a *strategy* we mean a correct decision tree of queries leading to discovery of the thief. By an *optimal strategy* we mean the decision tree of the lowest possible height. In the following text we will consider only the reasonable strategies, where we will never query a vertex for which the answer can be infered from the previous queries and answers.

What doesn't work

During the contest we saw a plenty of submissions trying to solve the problem with various incorrect strategies:

- choose the center [7] of the tree,
- choose a vertex, which minimizes the size of the biggest subtree,
- choose a vertex, which minimizes the length of the longest path in any of the subtrees.

Even though every such strategy returns an asymptotically optimal solution, requiring $O(\log N)$ queries, for each of them there exists a counter-example where the strategy returns the answer worse by a constant factor.

Brute-force solution

In each step we remember the set of vertices where the thief can possibly be hiding. Then we try all the possible ways to choose the next vertex, solve the problem recursively and return the best answer.

Clever implementation with bitmasks and memoization has time complexity $O(N \cdot 2^N)$. Implementing this solution first would not only give you 40 points, but it would also help you find the counter-examples for the incorrect strategies you might come up with and help you better understand the devil in this problem.

Optimal solution

Imagine you already have a strategy which in the worst case requires X queries. We will now define a *strategy function* $f : V \to \mathbb{Z}_{>0}$ which maps the vertices of the tree into positive integers. For each vertex v, $f(v) = |\text{height of the node in the strategy's decision tree in which you query } v|$. In other words, if v_1 is in the root of the decision tree (i.e. you first query v_1), then $f(v_1) = X$. If vertices $v_2, v_3 ... v_i$ are on the second level of the decision tree (i.e. based on the first answer you would query one of the $v_2, v_3 ... v_i$),

[7] http://en.wikipedia.org/wiki/Graph_center

 licensed under CC BY-SA 3.0

then $f(v_2) = f(v_3)... = f(v_i) = X - 1$. And so on, until you assign 1 to the vertices deepest in the decision tree. Because decision trees of reasonable strategies have for each vertex v exactly one node in which you query it, the function is always well-defined.

Lemma (*path property*): For any strategy function holds that for each pair of distinct vertices $v_1, v_2 \in V$ if $f(v_1) = f(v_2)$, then on the simple path from v_1 to v_2 there is a vertex v_3 such that $f(v_3) > f(v_1)$.

Proof: It should be obvious from the construction of strategy function.

We will now show that actually every function $f : V \to \mathbb{Z}_{>0}$ satisfying the *path property* represents a valid strategy which in the worst case requires no more than M_f queries, where M_f is the maximum value of $f(v)$. We will prove it by constructing a valid strategy from such function:

First vertex to query, v, will be the one with value M_f. There should be only one such vertex in the tree, otherwise the *path property* would not hold. The second vertex to query in each of the subtrees separated by v is the vertex with the maximum value in that subtree. Again, from the *path property* we know there should be only one such vertex in each subtree. And so on.

So to determine the optimal strategy it suffices to compute a strategy function with the lowest maximum value M_f.

We arbitrarily root the tree and compute $f(v)$ for each vertex in the bottom-up order. For each v, $T(v)$ represents the subtree rooted at v. Also we say that vertex $w \in T(v)$ is visible from v if on the path from v to w there is no vertex y such that $f(y) > f(w)$. If w is visible from v, we also say that value $f(w)$ is visible from v. Actually for each vertex we can store the bitmask of all the values visible from v in $T(v)$. For example if values $\{5, 4, 2, 0\}$ are visible, we will store $(110101)_2$. We will call it a *visibility mask* a write it $S(v)$.

For the bottommost vertices we set $f(v) = 1$.

For every other vertex v, $f(v)$ will be set to the lowest possible value, while taking the visible values in $T(v)$ into account. For a vertex v value X is forbidden in any of the two conditions hold:

- value X is visible from v,
- value Y is visible from v in at least two different subtrees and $Y \geq X$.

From the visibility masks $S(v_1), S(v_2)...S(v_k)$ of the children of v and the value $f(v)$ we can compute the value of $S(v)$ using the bit operators as follows:

$$S_f = 1 << f(v) S(v) \qquad = (S(v_1)|S(v_2)...|S(v_k)|S_f) \& \ (S_f - 1). \tag{1.1}$$

What the complicated-looking formula does, is basically a bit OR of all the children's visibility masks with the mask of $f(v)$ and then erasing all the values lower than $f(v)$ from the mask, which are not visible anymore.

Finally the answer to the original problem is M_f, the maximum value $f(v)$ assigned to any of the vertices.

Depending on how fast can we calculate the value of $f(v)$ the run time of the algorithm can be $O(N)$ if we use the clever bit operations machinery and calculate it in constant time. Otherwise we can simply sequentially find the lowest possible value for $f(v)$, resulting in the total run time $O(N \cdot \log N)$ which would also score 100 points.

Proof of correctness

Now we will prove that the above algorithm always calculates the optimal strategy, even though rooting the same tree by the different vertex can result into a possibly different strategy, which is nevertheless optimal.

 licensed under CC BY-SA 3.0

The function calculating the visibility mask $S(v)$ (i.e. the ugly bitwise expression above) is *monotone* and *minimizing*. By *monotone* we mean that by increasing the value of the visibility masks of some of its children, we do not decrease the value of the visibility mask of their parent. By *minimizing* we mean that for the given visibility masks of the children, we calculate the $S(v)$ with the minimum possible value.

Lemma: If the function calculating $S(v)$ is *monotone* and *minimizing*, then the bottom-up algorithm calculates a strategy function $f(v)$ for which M_f is minimal.

Proof: First we will show that the computed visibility mask of the root is minimal. We will prove it by induction on the height of the tree. Suppose that the masks computed for the children of v are minimal. It means that they are not greater than corresponding masks for any other strategy function. Since $S(v)$ is *monotone*, it wouldn't help if any of the children values would be bigger. And because $S(v)$ is also *minimizing*, there is no smaller visibility mask $S(v)$ for any strategy function.

Because the visibility mask at the root is minimal, so is M_f, because the visibility mask from the root always contains M_f and no greater values.

 licensed under CC BY-SA 3.0

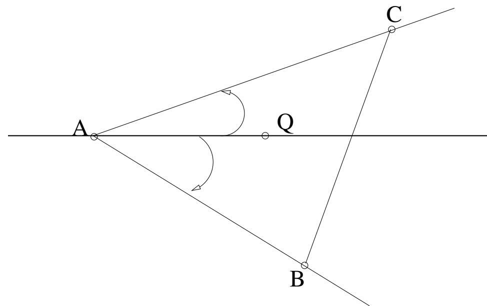
New Tree

Problem author: Gyula Horváth
Task preparation: Gyula Horváth
Solution writeup: Gyula Horváth

Trees are point in the plane. Denote the new tree by Q.
We distinguish two cases.

Case 1

The case when there is no point U with the following property: U is collinear with A, Q and Q is between A and U.
Consider the point B on the right side of the line $\overrightarrow{A,Q}$ which has the smallest angle with $\overrightarrow{A,Q}$. If there are more such points, chose the one which is closet to A.
Similarly, consider the point C on the let side of the line $\overrightarrow{A,Q}$ which has the smallest angle with $\overrightarrow{A,Q}$. If there are more such points, chose the one which is closet to A.
If there is no such B or C then there is no solution.
It is clear that there is a solution if and only if the pair of points B and C is a solution. That is, the triangle with nodes A, B and C strictly contains the point Q.

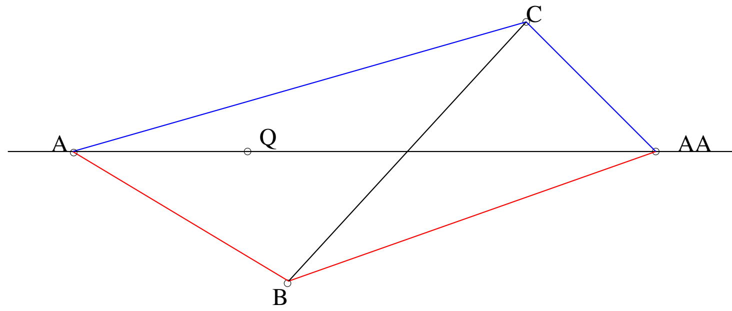


Case 2

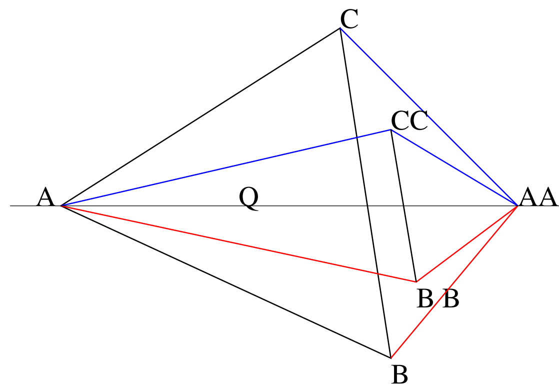
There is a point AA with the following property: AA is collinear with A, Q and Q is between A and AA. If there are several such points, choose the one which is closest to Q.

In this case a solution exists if and only if there is a point B and there is a point C such that there is no point inside the triangle with nodes A,B and AA and the triangle with nodes A,AA and C and the pair B C is a solution.

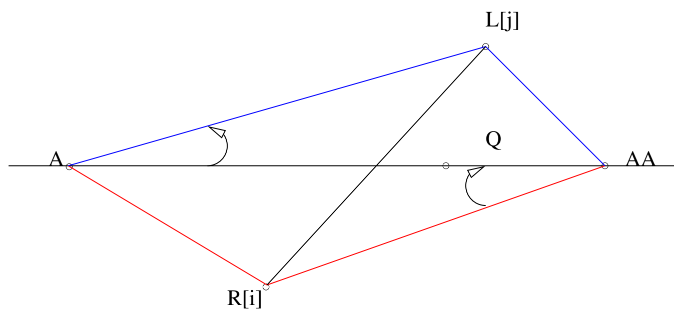
 licensed under CC BY-SA 3.0



Indeed, if there is a point BB in the triangle A, B, AA then replace B with BB. Similarly, if there is a point CC in the triangle A, AA, C then replace C with CC.

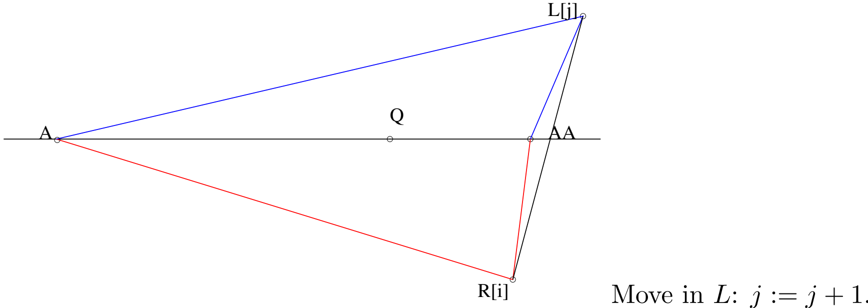


Consider the set R of all point U on the right side of the line $\overrightarrow{A, Q}$ with the property that there is no point inside the triangle $\triangleleft(A, U, AA)$. Sort the set R by the angle at AA. Similarly, let L be the set of all point V on the left side of the line $\overrightarrow{A, Q}$ with the property that there is no point inside the triangle $\triangleleft(A, AA, V)$. Sort the set L by the angle at A.
We can select solution pair by alternately moving either in R or in L until solution found or exhaust one of the sets.



Move in R: $i := i + 1$.

 licensed under CC BY-SA 3.0



Move in L: $j := j + 1$.

The set of points R and L can be constructed by sorting the points by angle.
Running time of the algorithm: $O(n \log n)$

 licensed under CC BY-SA 3.0

Cubic Art

Problem author: Marián Horňák
Task preparation: Marián Horňák
Solution writeup: Marián Horňák

It is not hard to realize that the simple simulation would work in quadratic time, which was enough to score 50% points! The problem is to write a bugless code that simulates the cube. The key to do it are the permutations.

Permutations

Although I am sure the reader had met the permutations before, I am going to explain basics. In the following discussion I will use array indexing starting at 1.

Permutation is an array P containing each of the numbers $1 \dots k$ once. It is understood to be a rearrangement of the array. $P[i]$ is the index i-th array element should be moved to. Applying permutation P to the array A creates an array $B = A \circ P$ such that $\forall i : B[i] = (A \circ P)[i] = A[P[i]]$. Permutation application can be implemented using simple loop.

It is easy to see that for array A and permutations P, Q holds:

$$\forall i : ((A \circ P) \circ Q)[i] = (A \circ P)[Q[i]] = A[P[Q[i]]] = A[(P \circ Q)[i]] = (A \circ (P \circ Q))[i]$$

and thus $(A \circ P) \circ Q = A \circ (P \circ Q)$. What does this mean? Applying P to the A and then Q to the result will have the same effect as applying Q to the P and the the result to the A. In other words, we can join P and Q to a single permutation before we use them. Permutation joining is just an application of the first one to the second. The above equation also says, that the joining of the permutations is associative (the order does not matter)

Permutation I that satisfies $\forall i : I[i] = i$ is called identity and does nothing. If $Q \circ P = I$ holds, Q is called inverse of P. Applying one permutation and then another is same as doing nothing. Therefore, the permutations must have opposite effects. The equation $Q[P[i]] = I[i] = i$ can be used to calculate Q using P.

Cube simulation

The simplest way to simulate the Rubik's cube, is to represent any sequence of moves as a permutation of small color squares. To keep it as simple as possible, we can include all 54 $(6 \cdot 3 \cdot 3)$ squares in the order they are in the input (and will be in the output.)

Since the middles don't move, we can precalculate the permutation for every elementary move. One way is to do it all by hand. One can calculate rotating to the other side using inverse. The other possibility is to write the foursomes that swaps in the cycle and parse them to the permutation.

Now it's quite easy to make a simple simulation. Read the moves and store corresponding permutations in the array. After each update just rejoin whole array and apply result to the original cube. This works in $O(n \cdot m)$ time.

Optimal solution

The only repetitive work we are doing is rejoining. To make it effective we can use simple interval tree.

 licensed under CC BY-SA 3.0

Let's enlarge the array of moves to the next power of 2 using identity permutation. This twices the array in the worst, so it does not affect the time complexity. Now build a complete binary tree using the array as the leafs. Then let each node to be a join of its children. Since the permutation joining is associative, the root will now contain the join of the whole array.

After updating a move and its permutation in the leaf, it is sufficient to recalculate the path to the root. This takes approximately $log(n)$ steps. Therefore, the time complexity is $O(n + m \cdot log(n))$.

 licensed under CC BY-SA 3.0

Universities

Problem author: Michal 'Mimino' Danilák
Task preparation: Michal 'Mimino' Danilák
Solution writeup: Michal 'Mimino' Danilák

In the following text I will refer to the problem in the terms from graph theory, where the universities are vertices and the roads are edges of the graph, which is also a tree. Each vertex is either black or white and has a cost associated with it. We say that the path in the tree is *balanced* if it contains the same number of black and white vertices. We are looking for the cost of the balanced path for which the sum of costs of its vertices is maximized.

First I will explain the solution to the easier subproblem. Then I will apply that solution to the original problem.

Subproblem

You are given a vertex v, find the optimal balanced path that goes through v.

We root the tree by the vertex v. For each other vertex w we calculate the path balance $B[w] = \sum_{x \in path_{w->v}} balance_x$, where $balance_x$ is 1 if the vertex is black and -1 otherwise. Obviously if $B[w] = 0$, then the path from w to v is balanced. For each vertex we also calculate the path cost $C[w] = \sum_{x \in path_{w->v}} cost_x$.

Now we have two choices on how the optimal path can look like:

1. v is one of the endpoints on the path,
2. v is one of the inner vertices on the path.

In the first case we can simply look at the vertices w with $B[w] = 0$ and store the best value $C[w]$ among them.

In the second case, let $c_1, c_2, ...c_k$ be the children of v, and $T(c_i)$ represent the subtree rooted at c_i. Then the endpoints of the optimal path must lie in two different subtrees. For each c_i we will calculate the array $BEST_{c_i}[b] = \max_{w \in T(c_i), B[w]=b} C[w]$, i.e. for each balance b we will store the cost of the best path from w to v with balance b. Because the indexes to this array can be negative, in C++ either use *map* or the following trick:

```
// now you can index array best even with negative indexes with absolute value up to MAX_BALANCE
int _best[2*MAX_BALANCE], * best = _best + MAX_BALANCE;
```

If we want to find the optimal path with endpoints in subtrees $T(c_1)$ and $T(c_2)$, we can iterate through all the values stored in $BEST_{c_2}$ and for each balance b we can get the cost of the balanced path starting in $T(c_2)$ and ending in $T(c_1)$ as $BEST_{c_2}[b] + BEST_{c_1}[balance_v - b] - cost_v$ and update the best value found so far. For simplicity the missing values in $BEST$ arrays have value $-\infty$.

After we processed the subtrees $T(c_1)$ and $T(c_2)$, we can merge their $BEST$ arrays into one $BEST_{c_12}[b] = \max(BEST_{c_1}[b], BEST_{c_2}[b])$. In the next step we will process the arrays $BEST_{c_12}$ and $BEST_{c_3}$ in the same way as described above.

Time complexity of this solution is linear in the number of vertices of the tree.

Original problem

Back to the original problem, where the optimal path can go through any vertex. We will define the strategy function previously introduced in the solution of *Investigation* task.

 licensed under CC BY-SA 3.0

Function $f : V \to \mathbb{Z}_{>0}$, mapping the vertices into positive integers, is a *strategy function* if for each pair of distinct vertices $v_1, v_2 \in V$ with $f(v_1) = f(v_2)$ holds that on the simple path from v_1 to v_2 there is a vertex v_3 such that $f(v_3) > f(v_1)$.

We can either construct an optimal strategy function as described in the solution of *Investigation* task or for the purposes of this problem any such function with the maximum value of $O(\log N)$ would be sufficient. So you could also use any strategy described in *What doesn't work* section of the *Investigation* solution.

For each vertex v we define $T_f(v) = \{w | w \in V, \forall y \in path_{w->v} : f(y) \leq f(v)\}$. Basically it is the subtree consisting of vertices reachable from v by traversing only the vertices for which $f(w) \leq f(v)$.

Lemma: For each pair of distinct vertices $v_1, v_2 \in V$ with $f(v_1) = f(v_2)$ holds that the subtrees $T_f(v_1)$ and $T_f(v_2)$ are disjoint.

Proof: Follows from the property of strategy function.

What will we do now is for each vertex v we will 'guess' that the optimal path goes through v and solve the subproblem with the algorithm described above. But for the subproblem we will only consider the vertices of the subtree $T_f(v)$, not the whole tree.

Imagine the optimal path in the graph. Let $M_path = \max\{f(v) | v \in path\}$. There is exactly one vertex v for which $f(v) = M_path$, for which holds that $path \subseteq T_f(v)$. Therefore we would discover this optimal path when solving the subproblem for the vertex v.

Time complexity of the whole algorithm is $O(N * \log N)$ - there are $O(\log N)$ different values of $f(v)$ and from the lemma we know, that the subtrees $\{T_f(v) | v \in V, f(v) = x\}$ are pairwise disjoint, therefore solving all the subtrees with the same root value takes in total $O(N)$.

 licensed under CC BY-SA 3.0

Wall

Problem author: ACM Regionals Central Europe 2004
Task preparation: Błażej Magnowski & Marek Sommer
Solution writeup: Marek Sommer

For the sake of simplicity let's create an additional point on the wall in which the robot starts repairment, and set its cost coefficients to 0 0 (so that it won't change the result). Let's name all the points (including the false one) in order they appear on the wall: W_1, W_2, W_3, ..., W_n. Let's say that the robot starts in point W_k.

First of all, we can see that the first cost coefficient C_i does not really affect the order of visiting the points on the wall. No matter what that order would be, these coefficients are constant and always should be added to the final cost. We must count the sum of C_i coefficients, add them to the result, and then we can ignore them to the rest of our algorithm, assuming, that from now on there is only one cost coefficient – D_i.

Let's compute the prefix and suffix sums of points coefficients, which will help us later. To be more precise, let's construct two arrays P and S such that for each $i \in \{1, 2, \dots, n\}$:

$$P[i] = \sum_{j=1}^{i} D_j$$

$$S[i] = \sum_{j=i}^{n} D_j$$

Also, let's assume that for $i \notin \{1, 2, \dots, n\}$, $P[i] = S[i] = 0$ (that will help us deal with special cases when $i = 0$ or $i = n + 1$).

After applying above three preparation steps, we can proceed to the algorithm. We will use dynamic programming. Let's compute the answers for some subsets of our points, which will lead us to the answer for all the points. Let's create an array $dp[i][j][f]$ with the following meaning:

- If $f = 0$, $dp[i][j][f]$ is equal to the minimal cost of repairing points $W_1, W_2, \dots, W_i, W_j, W_{j+1}, \dots, W_n$ with the robot starting at position W_i.
- If $f = 1$, $dp[i][j][f]$ is equal to the minimal cost of repairing points $W_1, W_2, \dots, W_i, W_j, W_{j+1}, \dots, W_n$ with the robot starting at position W_j.

In short, $dp[i][j][f]$ is equal to the cost of repairing points from only a prefix $1..n$ and a suffix $j..n$, starting from position i or j (which depends on f – if it is 0 or 1). Not all the fields in this array will be used. Only those which meet conditions: $0 \le i < j \le n + 1$, $f \in \{0, 1\}$. For simplicity, let's say that prefix $1..0$ is empty and suffix $(n + 1)..n$ is also empty.

If $i = 0$ and $j = n + 1$, there are no points to repair, so:

$$dp[0][n + 1][0] = dp[0][n + 1][1] = 0$$

In some more general situations (let's assume that $i \neq 0$ and $j \neq n + 1$):

$$dp[0][j][0] = \text{(let's leave it undefined)}$$

$$dp[0][j][1] = |X_j - X_{j+1}| \cdot S[j + 1] + dp[0][j + 1][1]$$

 licensed under CC BY-SA 3.0

$$dp[i][n+1][0] = |X_{i-1} - X_i| \cdot P[i-1] + dp[i-1][n+1][0]$$
$$dp[i][n+1][1] = \text{(let's leave it undefined)}$$

In the most general situations (when $i \neq 0$ and $j \neq n+1$):

$$dp[i][j][0] = \min\left(|X_{i-1} - X_i| \cdot (P[i-1] + S[j]) + dp[i-1][j][0], |X_i - X_j| \cdot (P[i-1] + S[j]) + dp[i-1][j][1]\right)$$

$$dp[i][j][1] = \min\left(|X_j - X_{j+1}| \cdot (P[i] + S[j+1]) + dp[i][j+1][1], |X_i - X_j| \cdot (P[i] + S[j+1]) + dp[i][j+1][0]\right)$$

Two equations above jest check two possibilities to go left or to go right and they choose the one, with smaller cost.

All theses equations together allow to compute any value $dp[i][j][f]$. The only difficulty left, is to process them in a correct order (to assure that every value on the right side of equation, is computed before the value on the left side). We can notice that every equation uses answers for a smaller number of points, so we can process the dynamic states in order from the one with 0 points to the ones with n points (the order of states within the same number of points does not matter).

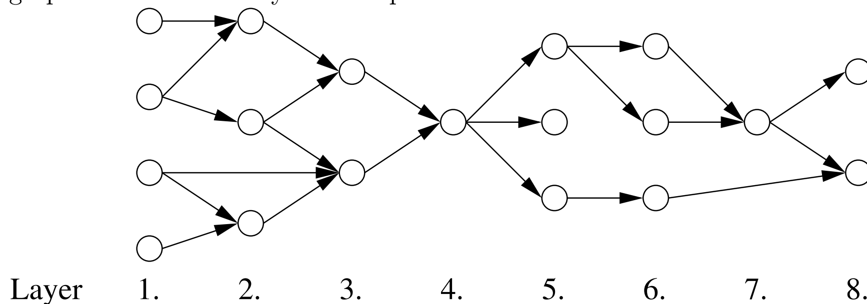
After computing answer for every state, the answer to the problem will be held under $dp[k][k+1][0]$ or $dp[k-1][k][1]$, where k is the initial position of the robot (the false point we made at the beginning).

 licensed under CC BY-SA 3.0

Critical Projects

Problem author: Gyula Horváth
Task preparation: Gyula Horváth
Solution writeup: Gyula Horváth

Consider the graph model of the problem: a directed graph $G = (V, E)$ where V is the set of the subprojects, $V = \{1, \dots, N\}$ and the edges of G are pairs (u, v) if u precedes v. G is obviously acyclic graph. Consider the layer decomposition of the node set V.



The first layer contains all nodes of indegree 0. Delete all nodes of the first layer, then all nodes of 0 indegree in this graph are the members of the second layer, and so on. Denote the layer number of node p by $L(p)$. Therefore the following holds.

$$L(p) = \begin{cases} 1 & \text{if } indegree(p) = 0 \\ \max\{L(q) : q \to p \in E\} + 1 & \text{if } indegree(p) > 0 \end{cases} \tag{1.2}$$

We define the function F as follows.

$$F(p) = \begin{cases} \infty & \text{if } outdegree(p) = 0 \\ \min\{L(q) : p \to q \in E\} & \text{otherwise} \end{cases} \tag{1.3}$$

Define the function $Ln(x)$ as the number of nodes in the layer x.

$$Ln(x) = |\{p : L(p) = x\}|$$

Statement

A node p is critical if and only if the following two conditions hold.

$$Ln(L(p)) = 1$$

$$(\forall q)(L(q) < L(p) \Rightarrow F(q) \leq L(p))$$

Assume that the conditions hold for a node p. We will show that for all node q if $L(p) < L(q)$ then there is path from p to q and if $L(q) < L(p)$ then there is path from q to p.
From the first condition and the definition of the function $L()$ it follows that for all nodes q with $L(p) < L(q)$ there is path from p to q: $p \rightsquigarrow q$.
Let $L(q) < L(p)$. We prove by induction on $L(q)$ that there is path from q to p. If $L(p) = L(q) + 1$ then by the definition of L and the second condition it follows that there is an edge from q to p. Assume that

 licensed under CC BY-SA 3.0

$L(q) = k$ and for all nodes s such that $k < L(s) \leq L(p)$ there is a path from s to p. It follows from the second condition that there is a node s such that $L(s) \leq L(p)$ and there is an edge from q to s. Therefore there is path $q \rightarrow s \rightsquigarrow p$.

Conversely, assume that p is critical. This implies that p must be the only node in its layer, that is the first condition holds. Assume that $L(q) < L(p)$ and $F(q) > L(p)$. This is a contradiction, because there is a path $q \rightsquigarrow p$.

Therefore if we compute the function L and F we can determine all critical nodes.
In order for checking the second condition, we compute a topological order $T = p - 1, \ldots, p_n$ with the following property that for all nodes p and q if $L(p) < L(q)$ it follows that p precedes q in the topological order. This can be done in $\Theta(n + m)$ time when n id the number of nodes and m is the number of edges of the graph.
Running time of the algorithm: $O(n + m)$

 licensed under CC BY-SA 3.0

Connect Highways

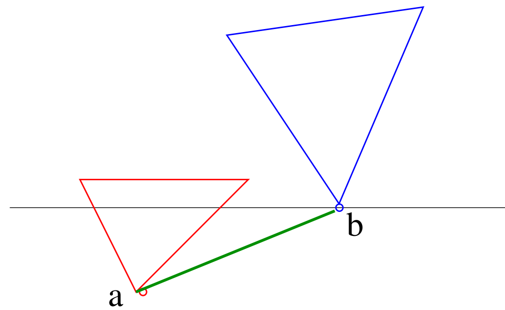
Problem author: Gyula Horváth
Task preparation: Gyula Horváth
Solution writeup: Gyula Horváth

Naive solution

Check for all pairs of points a in Red and b in Blue whether any segment of Red or Blue crosses the segment $\overline{(a,b)}$. The running time of this algorithm is $O(n_1\, n_2\, (m_1 + m_2))$ where n_1 and n_2 are the number of points, m_1 and m_2 are the number of segments, respectively.

Linear time solution

Let a and b be the points with least y-coordinates in Red and Blue networks, respectively:

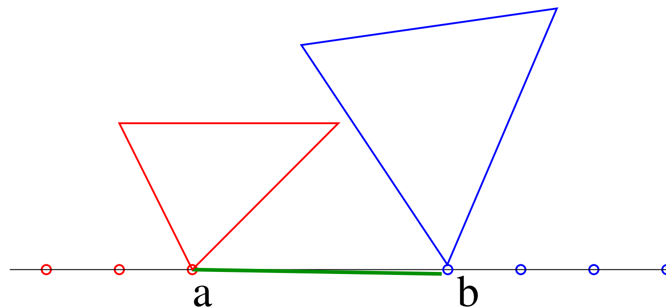


$$a : (\forall p \in Red)(a.y \le p.y)$$

$$b : (\forall p \in Blue)(b.y \le p.y)$$

Without loss of the generality we assume that $a.y \le b.y$.
We distinguish three cases.

Case 1: $a.y = b.y$

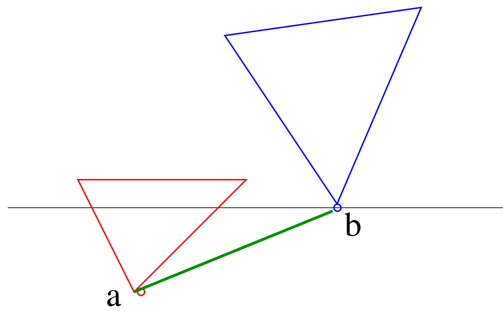


 licensed under CC BY-SA 3.0

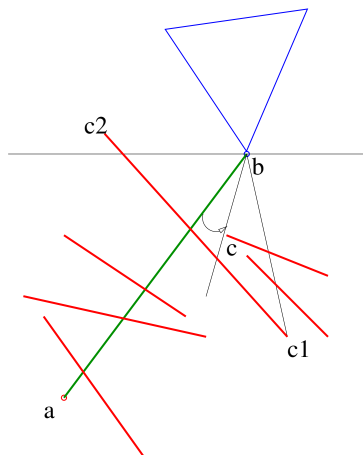
In this case let a and b be the points such that there is no point between a and b. In this case the pair $a\,b$ is a solution.

Case 2: $a.y < b.y$ **and there is no segment crossing the segment** $\overline{(a, b)}$

In this case the pair $a\,b$ is a solution.



Case 3: $a.y < b.y$ **and there is a segment crossing the segment** $\overline{(a, b)}$

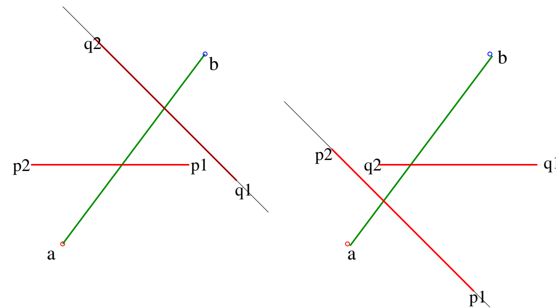


Let $c1$ and $c2$ the endpoints of the such that the intersection of the segment (a, b) and the segment $(c1, c2)$ is closest to the point b. Assume that $c1.y < b.y$. Then the pair $c1\,b$ is a candidate solution. It is a solution if there is no segment crossing the segment $(c1, b)$. Otherwise take the point c which is the endpoint of a crossing segment and there is no endpoint of crossing segment inside the triangle (b, a, c). In this case the pair $c\,b$ is a solution.
In order to compute the segment $\overline{(c1, c2)}$ we define a linear ordering relation on the set of segments crossing the segment $\overline{(a, b)}$:

$$\overline{(p_1, p_2)} < \overline{(q_1, q_2)}$$

if and only if either of the case depicted in the following figure hold.

 licensed under CC BY-SA 3.0



It is clear, that segment $\overline{(c1, c2)}$ is the maximal element of the set of segments crossing the segment $\overline{(a, b)}$ according to the relation defined above.
For the implementation we need the following three basic geometry operations.

```
int Turn(Point P0,Point P1,Point P2){
//Output:
//-1 iff P0-P1-P2 clockwise turn
// 0 iff P0,P1,P2 collinear
// 1 iff P0-P1-P2 counter-clockwise turn
   int64 crossp=(P1.x-P0.x)*(P2.y-P0.y)-(P2.x-P0.x)*(P1.y-P0.y);
   if (crossp<0)
      return -1;
   else if (crossp>0)
      return 1;
   else
      return 0;
}

bool Sless(Point p1, Point p2, Point q1, Point q2){
//Input: segment (p1,p2) and (q1,q2) crosses segment (a,b)
//Output: true iff intersection of (a,b) and (p1,p2) is closer to a then the intersection of (a,b) and (q1,q2)
   int pq1=Turn(p1,p2,q1);
   int pq2=Turn(p1,p2,q2);
   int qp1=Turn(q1,q2,p1);
   int qp2=Turn(q1,q2,p2);
   return pq1<=0 && pq2<=0 || qp1>=0 && qp2>=0;
}

bool Between(Point p1,Point p2, Point p3){
//Input: p1-p2-p3 collinear
//Output: true iff p3 is between p1 and p2
   return   (abs(p1.x-p3.x)<=abs(p2.x-p1.x)) &&
            (abs(p2.x-p3.x)<=abs(p2.x-p1.x)) &&
            (abs(p1.y-p3.y)<=abs(p2.y-p1.y)) &&
            (abs(p2.y-p3.y)<=abs(p2.y-p1.y)) ;
}
```

Running time of the algorithm: $O(m_1 + m_2)$ where m_1 and m_2 are the number of segments of Red and Blue networks, respectively.

 licensed under CC BY-SA 3.0

Next Permutation

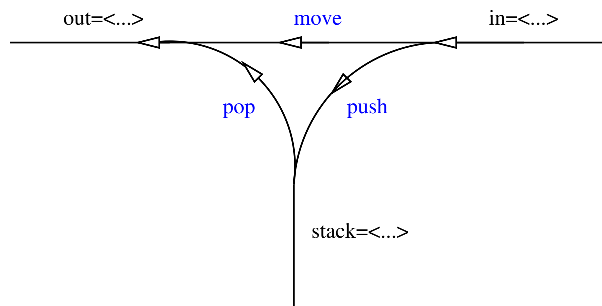
Problem author: Gyula Horváth
Task preparation: Gyula Horváth
Solution writeup: Gyula Horváth

Naive solution

Algorithm: Take next permutation until 3-1-2 avoiding permutation found.
The running time of this algorithm is exponential in worth case, since there is a 3-1-2 avoiding permutation π of n elements such that there are $(n-2)!-1$ permutations between π and the next 3-1-2 avoiding permutation. For example, if $\pi = \langle n-1, n, n-2, \ldots, 2, 1\rangle$ then the next 3-1-2 avoiding permutation is $\langle n, n-1, n-2, \ldots, 2, 1\rangle$.

Linear time solution

First we show that 3-1-2 avoiding permutations can be represented by stack generation. Consider the following stack operations



I/O specification of the stack operations

$$\{in = \langle a, \alpha\rangle \wedge stack = \langle \beta\rangle \wedge out = \langle \gamma\rangle\} \quad move \quad \{in = \langle \alpha\rangle \wedge stack = \langle \beta\rangle \wedge out = \langle \gamma, a\rangle\} \tag{1.4}$$
$$\{in = \langle a, \alpha\rangle \wedge stack = \langle \beta\rangle \wedge out = \langle \gamma\rangle\} \quad push \quad \{in = \langle \alpha\rangle stack = \langle \beta, a\rangle \wedge out = \langle \gamma\rangle\} \tag{1.5}$$
$$\{in = \langle \alpha\rangle \wedge stack = \langle \beta, b\rangle \wedge out = \langle \gamma\rangle\} \quad pop \quad \{in = \langle \alpha\rangle \wedge stack = \langle \beta\rangle \wedge out = \langle \gamma, b\rangle\} \tag{1.6}$$

We say that a permutation π can be generated by stack if there is a sequence of stack operations that results π from the input sequence $\langle 1, 2, \ldots, n\rangle$.

Statement

A permutation π is a 3-1-2 pattern avoiding permutation if and only if π can be generated by stack.

Proof

Assume that π is generated by stack. Consider the moment when an element x moved to the output (by move or pop operation). In this moment every element of the input sequence larger then x, moreover,

 licensed under CC BY-SA 3.0

the stack content is always decreasing sequence. Therefore there is no 3-1-2 pattern in π with x as the first element of the pattern.

Conversely, we prove by induction on the length of the permutation that every 3-1-2 pattern avoiding permutation can be generated by stack. It is obvious that every permutation of length 1 or 2 can be generated by stack. Consider a 3-1-2 pattern avoiding permutation π of the elements $1, \dots, n$ and decompose it as $\pi = \langle \alpha, 1, \beta \rangle$. Both α and β are 3-1-2 pattern avoiding sequence. Moreover, due to the 3-1-2 pattern avoiding property, if k is the largest element of α then α contains all numbers from 2 to k. Consequently, β contains all numbers from $k+1$ to n. By induction, α can be generated by stack with a sequence of stack operations (α) from the input sequence $2, \dots, k$. Similarly, β can be generated with a sequence of stack operations (β) from the sequence $k+1, \dots, n$. Therefore, the sequence of stack operations $push, (\alpha), pop, (\beta)$ generates π from the input sequence $\langle 1, 2, \dots, k, k+1, \dots, n \rangle$.

Notice that the generating sequence of stack operations is unique if *push*; *pop* not allowed (give *move* instead).
Let π be a 3-1-2 pattern avoiding permutation of length n. Decompose π according to the position of the number n in π: $\pi = \langle \pi_1, u, n, \pi_2 \rangle$. The position of u is the largest position such that there is a larger element than u in the sequence whose position is larger than the position of u. Consider the moment of the stack generation of π, when u moved (by *move* or *pop* operation) to the output:

$$out = \langle \alpha, u \rangle \text{ and } in = \langle \beta \rangle \text{ and } stack = \langle \gamma \rangle$$

, where $\beta = v, v+1, \dots, n$ for some v. Then

$$\pi = \langle \alpha, u, \mathrm{mirror}(\beta), \mathrm{mirror}(\gamma) \rangle$$

The permutation

$$\rho = \langle \alpha, v, u, \mathrm{mirror}(\gamma), v+1, \dots, n \rangle$$

is a 3-1-2 pattern avoiding permutation because ρ can be generated by stack, and ρ follows π according to the lexicographic ordering since $u < v$. Moreover, all elements of γ are less than u and v is the smallest in β implies that there is no 3-1-2 pattern avoiding permutation between π and ρ.

Using this observation we can develop $O(n)$ running time algorithm which computes the next 3-1-2 avoiding permutation.

 licensed under CC BY-SA 3.0

Game

Problem author: Codeforces
Task preparation: Błażej Magnowski & Marek Sommer
Solution writeup: Błażej Magnowski

Lets try to find a way to determine whether the first player can win when the starting numbers are A and B. In other words, we want to know if (A, B) is a winning state. Lets assume $A \geqslant B$. We have two types of moves. The first type of move is to move to state $(A \bmod B, B)$ and the second type of move is to go to state $(A - B^k, B)$ where k is a positive integer.

If one of the numbers is equal to zero, we know it is a winning state. So from now on, lets assume that $A \geqslant B > 0$. We know we can make a move of the first type to a state $(A \bmod B, B)$. If this is a losing state then (A, B) is a winning state. Lets assume $(A \bmod B, B)$ is a winning state. So the other possible moves are the moves of the second type to a state $(A - B^k, B)$. If $A - B^k < B$ then $A - B^k = (A \bmod B)$ and we reached state $(A \bmod B, B)$ which we know is a winning state. If $A - B^k \geqslant B$ then from state $(A - B^k, B)$ by making a move of the first type we will reach state $(A \bmod B, A)$ because $A - B^k \equiv A (\bmod B)$. So from now on none of the players will make a move of the first type because it will lead their opponent to a winning state.

So we reduced our game to a subgame: we start from state (A, B) where $A \geqslant B$ and we can only use moves of the second type. Whoever moves to state $(A \bmod B, B)$, loses. We can redefine this game. Let $X = \lfloor \frac{A}{B} \rfloor$. The possible moves are to subtract B^k from X where k is a non-negative integer. Whoever will turn X to zero loses. If B is an odd number then $B^k \equiv 1 (\bmod 2)$ for any k so the value $X \bmod 2$ determines who wins. If B is an even number then lets look at the value of $X \bmod (B + 1)$. We will prove that if and only if this value is even then this is a winning state. Let's assume that $X \bmod (B+1)$ is even. If $X \bmod (B + 1) \neq 0$ then we can subtract $B^0 = 1$ from X and then $(X - 1) \bmod (B + 1)$ will be odd. If $X \bmod (B + 1) = 0$ then either $X = 0$ and that means that our opponent reached zero so we won or $X \geqslant B$. In the second case we can subtract $B^1 = B$ from X and now $(X - B) \bmod (B + 1) = 1$. Now let's assume that $X \bmod (B + 1)$ is odd. We can see that $B^k \equiv (-1)^k (\bmod (B + 1))$ so no matter what move we make, we can either decrease or increase the value of $X \bmod (B+1)$ by 1. So if the value of $X \bmod (B + 1)$ is odd then no matter what move we will make then $(X - B^k) \bmod (B + 1)$ will be even. Also if the value of $X \bmod (B+1)$ is even we either already won or we can make such a move that $(X - B^k) \bmod (B + 1)$ will be even. Therefore we proved that for given numbers X and B (B is even) this state is a winning state if and only if $X \bmod (B + 1)$ is even.

In short, to check if (A, B) is a winning state $(A \geqslant B > 0)$ we have to find out if $(A \bmod B, B)$ is a winning state. After that in constant time we can tell if (A, B) is a winning state. There will be at most $O(\log A)$ states to check, so the time complexity of determining if (A, B) is a winning state is $O(\log A)$.

 licensed under CC BY-SA 3.0

Posters

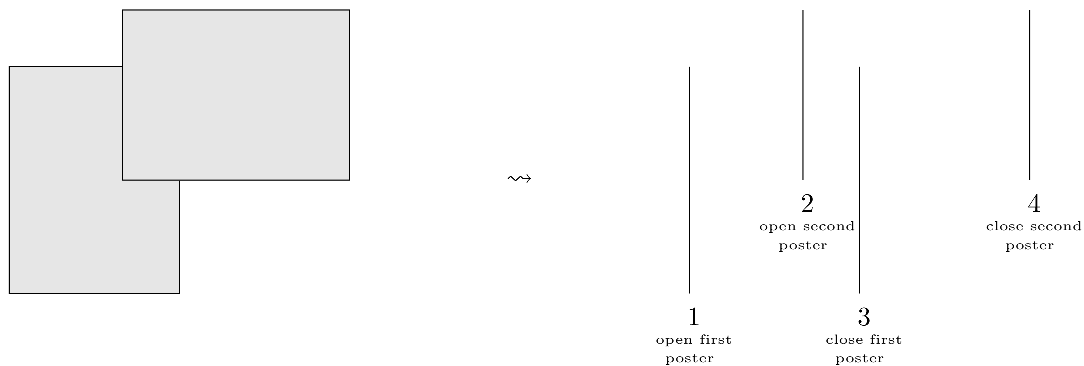
Problem author: Marek Sommer
Task preparation: Marek Sommer
Solution writeup: Marek Sommer

Subtask solution

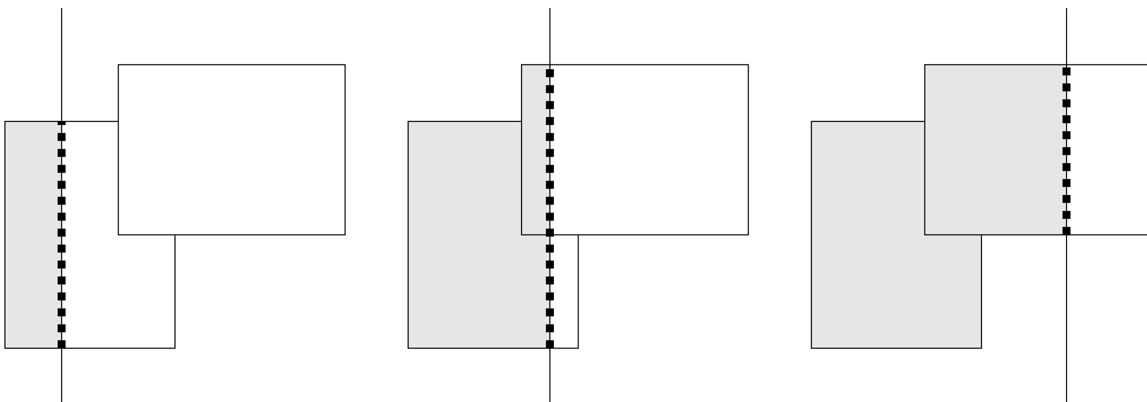
The task statement specified that in 50% of testcases, there holds $n, m \leq 1000$. This subtask can be solved by considering each new poster separately. So let's fix one of the new posters (and name it P), and let's find the answer for that poster.

For each poster, which hangs on the wall, we can substitute it with its intersection with P. Those operations cannot change the final answer, because the parts of posters, which lay outside the poster P, wouldn't be counted anyway. After these operations, the answer for poster P is the total area covered by the changed posters. From now on poster P is useless.

To compute the answer, we will use the sweep line algorithm. We will split each poster into two events – the opening (left) and the closing (right) edge of the poster, and then we will process the events in order from left to right.

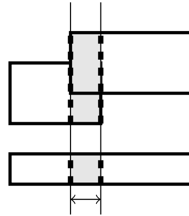


After every event, we would like to know the length of the intersection of opened posters with vertical line from sweeping. On the figures below, those intersections are marked with thick, dashed lines.

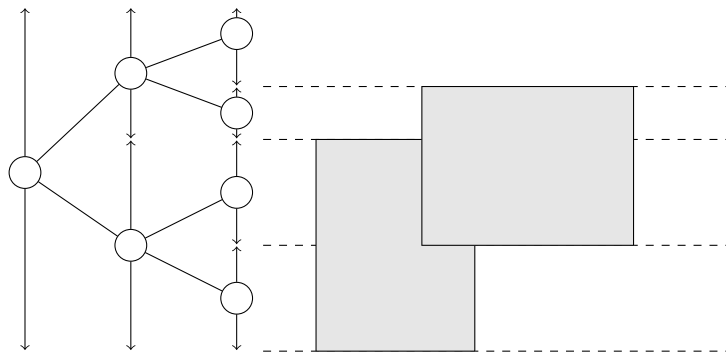


 licensed under CC BY-SA 3.0

If we would be able to compute this value, then we could easily find the area covered by the posters, between two adjacent vertical lines. To find this area, we could simply multiply the length of the intersection, by the distance between these lines.



We will build a segment tree, which will count the length of this intersection. The leafs in the segment tree will represent the spaces between y coordinates, which belong to any poster. The other vertices will represent the spaces, which belong to their sons.



In each vertex we should hold some information about the range it represents.

First of all, the length of a range should be known. It is easy to find such a value for every leaf, and for other vertices we could compute it, by adding the lengths of both sons. Let's call this value `length`.

The second value stored in a vertex is an indicator whether the whole range (which is represented by this vertex) lies inside any opened rectangle. If this value is grater than zero, then it means that there is such a rectangle, but if the value is equal to zero, than it does not necessarily mean that there is no such rectangle, because this value could be grater than zero in any ancestor of that vertex.

The third value would be the one we are looking for – the length of the intersection of opened rectangles with the sweeping line. We will call it a `covered` value.

Updating this values on our segment tree is easy. If we meet an opening (closing) edge of a rectangle, then we should find the vertices, whose ranges cover the whole edge of that rectangle. Then we should increase (decrease) the second value in each of these vertices, and then we should update the `covered` value in every node on the path from each of those vertices to the root. If we try to update the `covered` value in vertex v, than we should consider some cases:

- If v is a leaf:
  - If second value in v is equal to 0, then the `covered` value is equal to 0.
  - If second value in v is grater than 0, then the `covered` value is equal to the length of the range represented by v.
- If v is not a leaf:

 licensed under CC BY-SA 3.0

- If second value in v is equal to 0, then the `covered` value is equal to the sum of `covered` values of both sons.
- If second value in v is grater than 0, then the `covered` value is equal to the length of the range represented by v.

This will not give us a correct `covered` value in every node, but the `covered` value in root will always be valid, after applying these updates. We can also see that in next steps of our algorithm only the value from root will be needed.

That way, each tree update after opening or closing a rectangle is done in logarithmic time (in size of a tree). Thus we can find the total area covered by n rectangles in $O(n \log n)$ time. If we consider each query separately, than we will get the $O(mn \log n)$ solution, which will give us 50% of points.

Final solution

To get 100% points, we need to slightly modify our segment tree. We would like to implement some new functionalities, which would allow us to consider all of the queries at the same time.

In the following description we will use a term "time" to name the x coordinates. For example, if we say "now", we will think about the current position of the sweeping line.

In our modified tree we would like to hold the total sum of the area covered by rectangles since the beginning of time till now (till the current sweep line position). Of course, we will hold such a value in every vertex, and each vertex will hold information only about the area covered inside the horizontal strip determined by the vertex range.

We would like to be able to read the sum of these areas on the given range. That would help us with queries, because if we split each query into the opening and closing edge, than we should only read the covered area from the beginning of time till the closing edge, and the covered area from the beginning of time till the opening edge, and then subtract those values to get the final answer.

Let's implement this total area feature. In a vertex v we should store two additional variables: `total_area` and `last_update`. These values will mean that the last time we visited vertex v, was when the sweeping line was at the position `last_update`, and at that time, the total area (from the beginning of time) was equal to `total_area`.

Let's say we now have time T, and we enter a vertex v. We know that the current covered area since the beginning of time is equal to[8]:

$$v.\texttt{total_area} + (T - v.\texttt{last_update}) \cdot v.\texttt{covered}$$

So, when we would like to enter any vertex in order to do something there (doesn't matter if we met an opening, or closing edge of a poster, or a query), we should increase the `total_area` by the difference from the last update to now, and then we should set `last_update` value to T. We cannot leave this update for later, because the `covered` value might change in this vertex and then we won't be able to find the difference so easily.

There's still a small problem with this method. Let's say that we update v at time t_1, and v.`covered` is equal to 0 (because there are no opened rectangles there). Then, at time t_2, we open a rectangle, and during that operation we update some vertices **above** v. The range represented by v is covered by this rectangle, but we never reach v, because we stopped in any ancestor of v. Then we close this rectangle at time t_3, and we still don't reach v. If we finally visit v at time t_4 we would like to update v.`total_area`, but with the previous formula we would add $(t_4 - t_1) \cdot 0$, because the `covered` value is equal to 0. We see that this value is not correct, because since t_2, until t_3, there was a rectangle covering v's range. Thus, we should add $(t_3 - t_2) \cdot v.\texttt{length}$ instead.

[8] This value is not totally valid, but we would like it to work like that. We will correct it later.

 licensed under CC BY-SA 3.0

In each vertex we should hold an information about the total time, when the range was fully covered since the last update. We can call this value `fully_covered`. Let's say that we enter the vertex v at time T and we try to update it. We should increase the `total_area` value by:

$$(T - v.\texttt{last_update} - v.\texttt{fully_covered}) \cdot v.\texttt{covered} + v.\texttt{fully_covered} \cdot v.\texttt{length}$$

And then, we should set `last_update` to T and `fully_covered` to 0.

To maintain the `fully_covered` value, we should also increase vertex's both sons' `fully_covered` value with v.`fully_covered`, because if v was covered, then all of its descendants were also covered during that time. If the "second value" in v is grater than 0 we should also increase the sons' `fully_covered` value by the time that passed from v.`last_update` until T, without counting the v.`fully_covered` time.

Such updating allows us to maintain the total covered area in **every** vertex. This leads us to an $O((n+m)\log(n+m))$ solution.

 licensed under CC BY-SA 3.0

Sorting

Problem author: JAG Practice Contest, Tokyo, 2011-11-06
Task preparation: Marek Sommer & Błażej Magnowski
Solution writeup: Marek Sommer

Let's see what would happen if we would put the numbers A, $A+1$, …, $B-1$, B into an array, and then sort them lexicographically (sorting may be done just by converting each number into a string, and then comparing two strings letter by letter). We may call this sorted sequence, a sequence S. Let n be the length of the sequence S (and also the size of set $\{A, A+1, \ldots, B-1, B\}$).

If we would look at any ascending subsequence of S, and if we would consider a set containing numbers from this subsequence, then this set would meet conditions stated in the task. That's because this subsequence is in lexicographical order (as a subsequence of a sequence with lexicographical order), and also it is in numerical order (because it's ascending).

From the other side, if we look at any subset of $\{A, A+1, \ldots, B-1, B\}$, with the given property, and we would sort it lexicographically, then it would form a subsequence of sequence S (because S contains all elements of this subsequence and these elements are correctly ordered). Moreover, this sequence would be ascending, because the set has the property, that lexicographical and numerical orders are the same. Hence, this subset would be formed from an ascending subsequence of sequence S.

So, the number of subsets of $\{A, A+1, \ldots, B-1, B\}$ with the given property is equal to the number of ascending subsequences of sequence S.

To find the number of such subsequences we will use an algorithm very similar to the algorithm which computes the longest ascending subsequence. We will use dynamic programming and for each element of the sequence S, we will compute the number of ascending subsequences, ending exactly on this element. Let's call this value p_i for i-th element. If we would be able to compute such values, then the answer would be the sum of them (as each ascending subsequence should end in one of those elements), plus one (we must add empty subsequence).

We will compute these values starting from the first element and ending on the last one. The value p_1 is always 1 (no matter what the first element is). If we try to compute the answer for the i-th element (S_i), then we must find the sum of all p_j such that $j < i$ and $S_j < S_i$. This sum is the number of all ascending subsequences, ending in i-th element, of length greater than one. Therefore we must increase this sum by one to obtain the number of all subsequences ending in the i-th element.

At this point we already have an $O(n^2)$ solution. We can make it better by finding the sums of p_j more efficiently then by checking each possible j.

Let's create an array of length n. First element of this array would represent A, second element would represent $A+1$, and so on. After we compute a value p_i, we should store it in the array in the element representing S_i. If we would like to compute the sum of p_j with $j < i$ and $S_j < S_i$, then we should simply calculate the sum of array's elements from the one representing A to the one representing $S_i - 1$. This continuous part of the array contains p_j values such that $S_j < S_i$. The second constraint that $j < i$ is simply hold thanks to the order in which we compute the values p_i (from $i = 1$, to $i = n$), so in the array there are only p_j such that $j < i$.

We reduced the problem to construction of an array, on which we could make two queries: setting one element, and computing a sum of elements on a range of elements. This could be done with a segment tree, or a Fenwick tree, giving us an $O(n \log n)$ algorithm to count the number of ascending subsequences. The overall solution is a bit slower, because the lexicographical sorting was done in $O(n \log n \log B)$ time.

 licensed under CC BY-SA 3.0

Mission

Problem author: Filip Hlásek & Michal 'Mimino' Danilák
Task preparation: Filip Hlásek & Michal 'Mimino' Danilák
Solution writeup: Filip Hlásek

Every path from B to E to H is equivalent to a pair of vertex disjoint paths – one from E to B and the other from E to B. We can even add a new vertex X and join it by edges of the same length with B and E respectively. Now the problem is to find two vertex disjoint paths from E to X.

We have reduced the task to a standard *Minimum-cost flow problem*[9] [10]. Usually there are edges with capacities, but in our case vertices have capacities equal to 1.

Vertex capacities

We must ensure that no vertex is visited more than once. We can do it by splitting every vertex into two – one is called input and the other output. A directed edge (a, b) from the original network goes from the output part of vertex a to the input part of vertex b in the new network. Additionally there is a directed edge from the input part to the output part of a vertex and its capacity is equal to the vertex's capacity. The resulting network is equivalent to the given network (in sense of flow), but only edges have capacities. The size of network was only linearly increased.

Min-cost max-flow

There are many approaches to this problem and all of them should be sufficient to solve the task. We will present one of the easiest without the proof of correctness.

1. Find the shortest path from E to X using Bellman-Ford algorithm (or any other).
2. For every edge on the path, decrease it's capacity by 1 and add an edge in the opposite direction with capacity equal to 1 and opposite cost.
3. Find the shortest path from E to X using Bellman-Ford algorithm (or any other which can deal with negative cost edges).
4. Sum of cost of the two paths is the required answer.

Overall complexity of the given algorithm is $O(NM)$.

[9] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem
[10] http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=minimumCostFlow1

 licensed under CC BY-SA 3.0

Newspapers

Problem author: Błażej Magnowski
Task preparation: Błażej Magnowski & Marek Sommer
Solution writeup: Błażej Magnowski

We will use binary search to solve this problem. So now, we want to know if for a given value S, there exists in our tree a path of length at least K, such that the arithmetic average of its edges is equal to or greater then S. Let's say that such a path exists and its length is equal to N. Let's denote the values of the edges that lay on this path as $V_1, V_2, \ldots, V_N$. Let's say that the value of the path is the sum of the values of the edges. We know that:

$$\frac{V_1 + V_2 + \ldots + V_N}{N} \geqslant S \qquad | \cdot N$$

$$V_1 + V_2 + \ldots + V_N \geqslant NS \qquad | - NS$$

$$(V_1 - S) + (V_2 - S) + \ldots + (V_N - S) \geqslant 0$$

By subtracting S from the values of edges we were able to change our to problem to finding out if the maximal value of the paths of length at least K is equal to or greater then zero. To solve this, we will use the same method that was used in task Universities. We choose some vertex and find the maximal value of the paths of length at least K which go through this vertex. After that, we erase that vertex from our tree and continue our procedure in the subtrees. The vertex we choose for the given subtree can be its centroid. Due to that, all the subtrees created from removing that vertex will be at least two times smaller than the starting subtree. If we can find the maximal value of the paths of length at least K which go through a vertex in a given subtree in linear time, the complexity of this part of the algorithm will be $O(N \log N)$.

Now, we need to know how to find the maximal value of the paths of length at least K which go through a given vertex. Let's name the given vertex X and let's make this tree rooted in X. Also let's call the path we are looking for the maximal path, and let's call the paths that start in X tails. At first, we need to compute the values of all the tails. The maximal path consist of at most two tails. Let's take any tail and assume that this path is one of the parts of the maximal path. Let's say that L is the length of this tail. Then, the other part of the maximal path will be a path of length at least $K - L$ which starts in X and goes into a different subtree then the first tail. Of course, the value of the other tail has to be maximal possible. Let's say the length of the longest tail is equal to M. We will construct an array P and for every i from 0 to M, $P[i]$ will hold two maximal values of tails of length i, which go into different subtrees. With this array, for every i from 0 to M, we can compute two maximal values of tails of length at least i, which go into different subtrees. Now, for every tail, we can assume that it is one of the tails of the maximal path, and add to it the most fitting tail in constant time. So we will find the value of the maximal path in linear time. Overall, the complexity of our solution is $O(N \log N \log D)$ where D is the maximal value of an edge, divided by the relative or absolute error that is accepted.

 licensed under CC BY-SA 3.0

An inexperienced slalomer

Problem author:	Peter 'Bob' Fulla
Task preparation:	Peter 'Bob' Fulla, Marián 'Syseľ' Horňák
Solution writeup:	Peter 'Bob' Fulla

A gate has two endpoints; we will call the one with smaller y-coordinate the *bottom* endpoint and the other one the *top* endpoint. In order to pass through each gate, Hubert must ski above every bottom endpoint and below every top endpoint. He moves along a straight line, therefore we may replace this requirement by "Hubert must ski above the upper convex hull of bottom endpoints and below the lower convex hull of top endpoints", which is equivalent to it.

If the intersection of the two convex hulls has a positive area, the answer will be `Impossible`, as no straight line will separate the hulls. Otherwise, let us choose a point P belonging to one of the hulls and a point Q belonging to the other hull such that their distance is minimal among all such pairs of points; we will denote their distance by d. In other words, d equals the distance between the two convex hulls. We claim that d is also the answer to our problem. Clearly, a disk of a diameter strictly greater than d cannot pass through the line segment PQ, therefore its trajectory cannot separate the two convex hulls. On the other hand, there is a valid trajectory for a disk of diameter exactly d: Its center will move along the perpendicular bisector of the line segment PQ. One can show that this trajectory does not intersect the two convex hulls (though it touches them) – if it did, there would be a pair of points P', Q' with a smaller distance than d.

The intersection of the two convex hulls has a positive area iff a vertex of one hull lies strictly inside the other hull. We can check this by a simple sweep line algorithm. The distance of the convex hulls can be computed as follows: Without loss of generality, we may assume that at least one of the points P, Q is a vertex of the convex hull it belongs to; the other may also be a vertex or it may lie on a line segment on the hull's border. For any fixed vertex V on a convex hull, we can compute its distance to the other hull H in time $O(n)$ simply by iterating through all vertices and line segments of H and taking the minimum of obtained distances. This would lead to a solution with time complexity $O(n^2)$. Realizing that the sequence of distances from a fixed vertex V is at first decreasing and then increasing, we can find its minimum using a ternary search in time $O(\log n)$. Another option is to use the two pointers technique to find minima for all vertices V in time $O(n)$. As we have to sort the gates at the beginning, the overall time complexity is in both cases $O(n \log n)$.

 licensed under CC BY-SA 3.0

Tickets

Problem author: Gyula Horváth
Task preparation: Gyula Horváth
Solution writeup: Gyula Horváth

Solution 1: request to seat selection

Let $C(x)$ is the set of candidate requests for seat x:

$$C(x) = \{r : r.first \leq x \leq r.last \text{ and r is not satisfied yet}\}$$

Greedy choice:
Choose the request with least last value.

```
for (x=1;x<=N;){
    insert r into C where r.first=x;
    if (!C.empty(){
        r=C.top; C.pop();
        A[x]=r;
    }
    while (C.top().last==x) C.pop();
}
```

The running time of this algorithm is $O(M \log M)$ using priority queue for the implementation of the greedy choice, where M is the number of the requests.

Solution 2: seat to request selection

Greedy choice:
Take the requests in decreasing order of their first value and if there is a free seat in between the request's first and last value, choose the largest. Let $R[x]$ be the set of requests whose first seat is x. The operation $FindEmpty(a, b)$ returns the largest empty seat d, and if $a \leq d$ then mark d as not free.

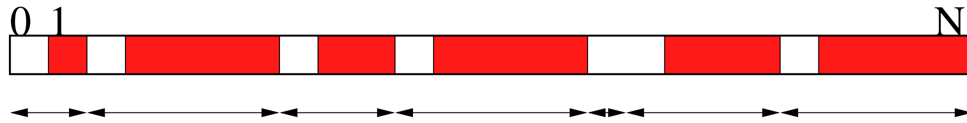
```
int sol=0;
for(int x=n;x>0;x--){
   for(Req y:R[x]){
      int d=FindEmpty(x, y.last);
      if(x<=d){
         sol++;
         A[d]=y.id;
      }
   }
}
```

Proof of correctness.
A problem instance is a par of sets (S, R), where S is the set of available seats and R is the set of requests. A solution of a problem instance (S, R) is a partial function $A : S \to R$, the seat assignment. Assume that A is an optimal solution. We show that the optimal solution can be modified such that it begins with the greedy choice. Let $r0$ be the greedy choice, i.e., $r0.first$ is largest among the requests. For the greedy solution Ag $Ag[r0.last] = r0$ holds. If $A[r0.last]$ is not assigned, then set $A[r0.last] = r0$ and if for some s $A[s] = r0$ delete this assignment. Assume that $A[r0.last] = r$ and $r \neq r0$. Therefore $r.first \leq r0.first$ and $r0.last \leq r.last$. If there is no seat s such that $A[s] = r0$ then set $A[r0.last] = r0$. Otherwise change assignment by setting $A[s] = r$ and $A[r0.last] = r0$. The modified optimal solution includes the greedy choice. Denote the modified assignment by $\overline{A}$.
Consider the reduced problem instance $(\overline{S}, \overline{R})$ where $\overline{S} = S - \{r0.last\}$ and $\overline{R} = R - \{r0\}$. The reduced

 licensed under CC BY-SA 3.0

assignment $A : \overline{S} \to \overline{R}$ is an optimal solution of the reduces problem instance, therefore $\overline{A}$ is an optimal solution of (S, R)

Implementation of the FindEmpty operation by Union-Find



Consider the disjoint intervals of the set of seats, such that the least element is a free seat and this is the only free seat in the interval.

The sets of request $R[x]$ can be represented by list, therefore the creation of the sets $R[x]$ requires $\Theta(M)$ time.
The running time of this algorithm is $O(M\,\alpha(N, M))$.

 licensed under CC BY-SA 3.0