IOI 2008 Tasks and Solutions
(Version 1.0)

Credits

Host Scientific Committee

Mostafa ElBatsh
Ossama Ismail
Ahmed Naguib
Richard Peng
Mohamed Taha

International Scientific Committee

Cesar Cepeda
Michal Forišek
Marcin Kubica
Martin Mareš
Lovro Puzar
Velin Tzanov

Special Thanks To

Khaled Hafez
Rostislav Rumenov
Filip Wolski

Contents

BUSES

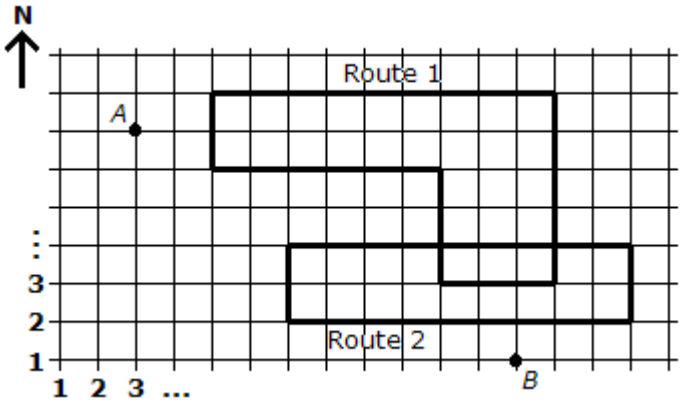
Portsaid is a north-eastern Egyptian city that lies near the Suez Canal. All streets in Portsaid are perfect straight lines oriented either north-south or east-west. Note that an intersection is a place where two streets cross each other. The public transport system of Portsaid consists of R bus routes. A route is a predefined cycle inside the city that is followed by a bus stopping at every intersection on that route. Each route has exactly one bus assigned to it that follows its path. Every bus has a fee fi , once paid you can take the bus as long as you want, but once you get off, you'll have to pay the bus fee again for another ride.

Since not every intersection in the city lies on the path of a route, sometimes you have to walk to be able to move from some intersection to another. When walking from one intersection to another, you can only use the streets.

Today you are in Portsaid for vacation and you want to go from intersection A to intersection B. Since you are here for vacation, you decided not to walk more than D blocks (A block is a distance that separates two consecutive intersections along a street). Also, you want to spend as little money as you can.

TASK

You are to write a program that is given the map of the bus routes and their fees, computes the minimum amount of money you have to spend to go from A to B walking at most D blocks.



To describe the city we will number North-South streets starting from the west and moving east, and East-West streets starting from the South and moving North as shown in figure. Intersections will be described as a pair (x, y) where x represents the North-South (vertical) street and y represents the East-West (horizontal) street.

INPUT

- 1st line consists of an integer D (0 <= D <= 300).
- 2nd line consists of two integers separated by a single space that represent intersection A.
- 3rd line consists of two integers separated by a single space that represent intersection B.
- Both coordinates of A and B range from 1 to 100,000,000 inclusive.
- A and B will never represent the same intersection.
- 4th line consists of an integer R (1 <= R <= 100), the number of bus routes in the city.
- Each of the next R lines will describe a bus route; each route will be described as a sequence of integers separated by single spaces as follows:

- The first integer on the line represents Ni (4 <= Ni <= 50), the number of intersections used to describe this route.
- The second integer on the line represents fi , the fee associated with that route. (0 <= fi <= 1,000,000).
- The Ni pairs of integers follow, each represents an intersection. The bus starts moving from the first intersection and drives in a straight line to the second; it turns 90 degrees and drives in a straight line to the third, etc. It keeps moving that way until it reaches the last intersection on the description; from there it turns 90 degrees and drives to the first one to start again. Note, that the ith route has Ni line segments which never overlap and never intersect except in the intersection point between any 2 consecutive segments. All coordinates will range from 1 to 100,000,000 inclusive.

OUTPUT

The output must be one integer representing the minimum amount of money you need to spend to get from A to B walking at most D blocks. If there is no possible way to achieve that, your program should output –1.

GRADING

In some cases worth 30 points:

- D does not exceed 100.
- A, B and the polygons' coordinates do not exceed 100.
- R does not exceed 25.
- Ni does not exceed 10.
- fi does not exceed 10.
- Any line segment in any polygon does not have more than 40 lattice points including the starting and ending points of that line segment.

DETAILED FEEDBACK

During the contest, your submissions for this task will be evaluated on part of the official test data, showing you a summary of the results.

EXAMPLE

Sample Input 1	Sample Output 1
4 3 7 13 1 2 6 2 14 8 5 8 5 6 11 6 11 3 14 3 4 5 16 4 7 4 7 2 16 2	2

This sample input corresponds to the shown figure. Note that the best way here is to move for 2 blocks, then use the first bus route, then walk for 2 blocks to reach B.

Sample Input 2	Sample Output 2
2 1 5 10 7 3 4 10 1 4 5 4 5 6 1 6 4 10 5 5 5 7 7 7 7 5 4 20 9 5 9 1 7 1 7 5	-1

SOLUTION

The Buses task is clearly a graph-theoretic task that is asking for the shortest (or, in this case, cheapest) path, given some constraints. In such tasks, it is usually the best approach to divide the solution into two conceptual steps: first build the graph, then run the appropriate algorithm.

In our task, there were multiple ways how to model the graph. The more natural one is to treat every grid point as a vertex. However, in such a graph it is hard to represent the bus lines: note that once we pay the fee for a bus, we would get to traverse multiple edges in such a graph. Also, it might be possible to get to a given spot in two ways, one of them cheaper, the other shorter, and one can not easily decide which of those ways will be used in the optimal path. Both of these issues show that the naive representation might not be the most suitable one. To find a better representation, we will first analyze the problem to see what we really care about.

Clearly, we will only use each bus line at most once. And obviously, if there is a solution, there is an optimal solution with the following property: Whenever we walk from a bus line X to a bus line Y, we take the shortest path available. (Take any valid cheapest solution. Write down the sequence of bus lines used. If we now construct a new path that uses the same bus lines in the same order, and always uses shortest walks, it will have the same cost and at most the same distance walked.)

Let $D_{X,Y}$ be the shortest distance one has to walk to get from (some spot on) the bus line X to (some spot on) the bus line Y. In our solution, we will start by computing the values $D_{X,Y}$ for each pair of lines X,Y. We will now explain a simple way how to compute these values.

Clearly, each bus line can be seen as a union of axis-parallel segments. If X and Y are two lines, then the distance $D_{X,Y}$ can be computed as the minimum of all distances between a segment P in X and a segment Q in Y. Computing the distance of two segments is a simpler task. If they intersect, the distance is zero. If not, the distance is either the distance between some of their endpoints, or the distance from the endpoint of one segment to its orthogonal projection on the other segment. Using this observation, one can compute the distance between two segments in O(1). Vector and scalar products are a good tool to implement this function in a nice, human readable way. If each of the bus lines has at most N segments, one can compute a single value $D_{X,Y}$ in $O(N^2)$, and thus all the shortest distances in $O(R^2N^2)$.

Having computed the values $D_{X,Y}$, it is easily seen that the exact topology of the bus lines does not concern us any more. And in this way we just found a more suitable graph representation: the vertices of the graph will correspond to the bus lines, the edges to walks between them. Each edge will have an associated length (the length of the walk) and a cost (the cost of buying a ticket for the bus line at its end). For edges entering the goal the cost will obviously be zero.

Now imagine that we would like to implement a complete search that examines all valid paths in our graph. At any moment we would need to know not only the vertex we are at (the bus line we are currently using), but also the total distance we walked so far.

Using this observation, we can finally formulate our task as a shortest path problem. Consider a graph with (R+2) times (D+1) vertices, where each vertex represents a bus line and the distance we had to walk to reach it. From each vertex, we have at most R+1 edges that correspond to walking to other lines and

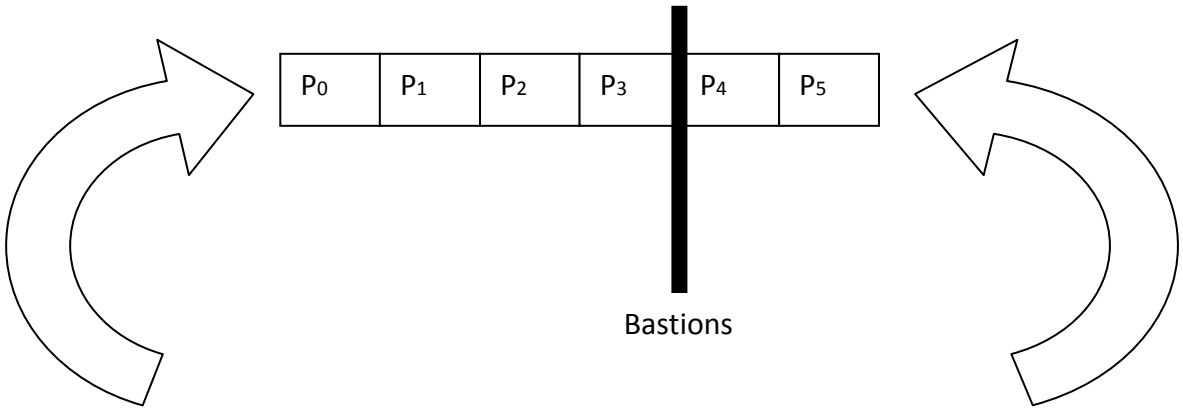
updating the total distance walked. For example, if the walking distance between line X and line Y is 3, then for each valid d we will have an edge from [X,d] to [Y,d+3].

In such a graph, we want to find the cheapest path from the vertex [A,0] to any vertex [B,*]. To do this, we can use Dijkstra's algorithm. If implemented with a priority queue that works in logarithmic time, the time complexity of searching for the path is $O(R^2D \log RD)$. The total time complexity of our solution is therefore $O(R^2N^2 + R^2D \log RD)$.

Both parts of this solution can still be improved. For the first part, sweeping can be used for a more effective way of computing the distances between the bus lines, reducing the time complexity for the first part to $O(R^2N \log N)$. For the second part, one can note that the graph is almost acyclic (acyclic except for cycles created by intersecting bus routes), and thus we can compute cheapest paths simply in $O(R^2D)$. Neither improvement was necessary to obtain a full score.

GAME

You are playing a computer game. Initially you have control of all **N** cities in the game. The cities can be seen as a sequence of adjacent squares that are numbered from **0** to **N-1** as shown in figure. By the end of each year, you gain some profit (**P_i**) from each city that is still under your control. Note, that the profits associated with each city are not the same since the resources of cities are different. Your kingdom withstood many attacks in the past, but now your enemy is preparing an enormous army that will crush your whole kingdom. As your defeat is certain, your aim is to have the maximum possible accumulated profit before losing everything.



Each year you choose to build your bastions between 2 adjacent cities that are still under your control. Then, the enemy chooses to attack you either from the west taking control of all cities before the bastions or from the east taking control of all cities after your bastions. By the end of that year, you get the profits associated only with the cities that are still under your control after the attack. Also, by the end of the year, your enemy will succeed to destroy the bastions but he will stop fighting until reinforcements arrive for the next year. In each of the next years, the same scenario happens again taking into consideration only cities that are still under your control.

TASK

This is an interactive task. Your task is to have the maximum possible profit before losing control of all cities. You should achieve this task by carefully choosing where to put your bastions each year keeping in mind that your enemy is playing optimally concerning his choice of whether attacking from the east or from the west of the bastions (he is trying to minimize your final profit).

LIBRARY

Your program must use a special library to play the game. The library consists of the files: `pgamelib.pas` (pascal), `cgamelib.h` and `cgamelib.c` (C/C++). The library provides the following functionalities:

- `procedure initialize()/void initialize()` – which must be called only once by your program. This call must be performed before any other calls for any of the following 3 functions.
- `function getN: longint/int getN()` – which returns the number of cities **N** (3 <= **N** <= 2000).

- `function getValue(city: longint):longint/int getValue (int city)` – which is given a city index (0 <= `city` < **N**) returns the profit associated with that city. The profit ranges between 1 and 100,000 inclusive. Calling this function with bad city index results in failure for your program for that test case. Note, that city **0** is the leftmost and city **N-1** is the rightmost.
- `function move(city: longint):longint/int move(int city)` – You call this function to specify the index of the city that you wish to have your bastions built right after it. It returns either 1 or 0 indicating whether the enemy will attack from the west (left) or the east (right) respectively. The valid values for the parameter `city` are only the indices of the cities remaining in your kingdom (except the right-most one of them). Calling this function with bad city index results in failure for your program for that test case.

Termination of your program is an automatic process when only one city is under your control. So, your program should keep making moves as long as there are still valid moves.

Your program must not read or write any files, it must not use standard input/output, and it must not try to access any memory outside your program. Violating any of these rules may result in disqualification.

COMPILATION

If your program is written in Pascal, then you must include '`uses pgamelib;`' statement in your source code. To compile your program, use the following command: `ppc386 -O2 -XS game.pas`

If your program is written in C or C++, then you must include '`#include "cgamelib.h"` ' statement in your source code. To compile your program, use one of the following commands:

```
gcc -O2 -static game.c cgamelib.c -lm
g++ -O2 -static game.cpp cgamelib.c –lm
```

You are provided with two simple programs illustrating usage of the above libraries: `cgame.c` and `pgame.pas`. (Please remember, that these programs are not correct solutions)

TESTING

To let you experiment with the library, you are given example opponent libraries: their sources are in `pgamelib.pas`, `cgamelib.h` and `cgamelib.c` files. They implement a simple strategy. When you run your program, it will be playing against these simple opponents. Feel free to modify them, and test your program against a better opponent. However, during the evaluation, your program will be playing against a different opponent that is playing optimally. When you submit your program using the TEST interface it will be compiled with the unmodified example opponent library. The submitted input file will be given to your program standard input. The input file should consist of **N+1** lines. The first line contains **N** and the next **N** lines contain the sequence of integers specifying the profits of the cities in order from city **0** to city **N-1**. These values are read by the example opponent library.

If you modify the implementation part of the `pgamelib.pas` library, please recompile it using the following command: `ppc386 -O2 pgamelib.pas`
This command produces files `pgamelib.o` and `pgamelib.ppu`. These files are needed to compile your program, and should be placed in the directory, where your program is located. Please do not modify the interface part of the `pgamelib.pas` library.

If you modify the `cgamelib.c` library, please remember to place it (together with `cgamelib.h`) in the directory, where your program is located — they are needed to compile it. Please do not modify the `cgamelib.h` file.

GRADING

If you get the maximum possible profit you receive full credit for this test case. Otherwise, you receive 0 points. Note that it is guaranteed that your opponent is playing optimally.
Also, note that for some tests worth 50 points, N will not exceed 500.

SAMPLE INTERACTION

Your Program Calls	Return Value	Comments
`initialize()`		
`getN()`	5	5 cities
`getValue(0)`	8	$P_0 = 8$
`getValue(1)`	6	$P_1 = 6$
`getValue(2)`	2	$P_2 = 2$
`getValue(3)`	4	$P_3 = 4$
`getValue(4)`	2	$P_4 = 2$
`move(1)`	1	The 2 intervals are [0,1] and [2,4]. Your opponent takes control of [0,1] and you still have control of [2,4], so your score after this turn is $P_2+P_3+P_4$.
`move(2)`	0	Note that the allowed moves here were only 2 and 3. After you choose 2, the intervals are [2] and [3,4]. Your opponent takes control of [3,4] and you now have control of only one city so, you add P_2 to your score and the game ends.

Your final score with this interaction is 8+2 = 10.

SOLUTION

This is quite obviously a task where dynamic programming shall be used – there are exponentially many ways the game can be played, but only polynomially many states in which the game can be. More precisely, at any moment the current state of the game is a contiguous subsequence of the original sequence of cities. In other words, each state can be uniquely described by the smallest index A of a city you have, and the smallest index B>A of a city you don't have. We will denote the state represented by A and B as [A,B).

Your goal at any moment is the same: maximize your profit in the game played from the current state. Let $S_{A,B}$ be the sum of profits for cities in [A,B) and $P_{A,B}$ be the best profit you can get from the game played on [A,B). In our solution, we will compute all the values $P_{A,B}$, and for each of them the optimal first move. Using this information we can then play the game optimally.
Note that our opponent can compute the same values and use them for his optimal play as well – when presented with a choice, he will always pick the option that will give us a lower profit. We can use this observation to write a recurrence relation for the values $P_{A,B}$:
$P_{A,A} = P_{A,A+1} = 0$
$P_{A,B} = \max_C(\min(S_{A,C}+P_{A,C}, S_{C,B}+P_{C,B}))$
The second equality holds for B>A+1, where the maximum is taken over all C such that A<=C<=B.
An explanation in words: When we pick the place C where to place the bastion, the opponent will examine the two possibilities and leave us with the worse one. Knowing this, we can compute the profit for each possible choice of C, and pick the best one.

By using the above recurrence, one can easily compute a single value $P_{A,B}$ in O(N), and as there are $O(N^2)$ pairs A,B, the total time complexity of the precomputation is $O(N^3)$. We can then make moves in O(1) each.

We can easily precompute the values $S_{A,B}$ in time $O(N^2)$. However, note that the values $S_{A,B}$ can be precomputed in O(N) time with O(N) memory – it is enough to store the values $S_{0,X}$, as $S_{A,B}=S_{0,B}-S_{0,A}$.

To get a better time complexity, we need to find a way to limit the set of places C we need to examine when computing $P_{A,B}$. We will use the following observation:
(*) Suppose that for the segment A,B the optimal split is C. Then for the segment A,B+1 the optimal split is at least C.
We will prove this later.

Once we trust that (*) holds, we can compute the values $P_{A,B}$ and $C_{A,B}$ in the following order:
[0,1) [1,2) ... [N-2,N-1) [N-1,N)
[0,2) [1,3) ... [N-2,N)
...
[0,N-1) [1,N)
[0,N)
When computing $P_{A,B}$ and $C_{A,B}$, we know from (*) and from symmetry that it is enough to consider $C_{A,B}$ between $C_{A,B-1}$ and $C_{A+1,B}$, inclusive.

For each row of our table above, the total number of possibilities we have to try when computing it in this way is at most 2N. If this is not clear to you, consider the following example:
The best split for [0,7) is between the best split for [0,6) and the best split for **[1,7)**.
The best split for [1,8) is between the best split for **[1,7)** and the best split for [2,8).
The best split for [2,9) is between the best split for [2,8) and ...

Thus we get a solution with both time and memory complexity $O(N^2)$.

Lemma 1:
For all A, B we have $P_{A,B+1} >= P_{A,B}$.
This is obvious. A formal proof by induction is possible, based on the fact that making the same split for [A,B+1) as for [A,B) leads at least to the same profit.

Proof of the observation (*):
Let C be an optimal split for [A,B). We will show that for [A,B+1) there is some optimal split >= C.
Consider the values $X = S_{A,C}+P_{A,C}$ and $Y = S_{C,B}+P_{C,B}$.
We have $P_{A,B} = \min(X,Y)$, and from Lemma 1 we have $P_{A,B+1} >= \min(X,Y)$.
Case 1: X <= Y.
In this case $P_{A,B} = X$. Let D < C. Clearly $S_{A,D} < S_{A,C}$. From Lemma 1, $P_{A,D} <= P_{A,C}$. Therefore $S_{A,D}+P_{A,D} < S_{A,C}+P_{A,C}$. If we make a split at D, our opponent will leave us with the part [A,D), and thus our total profit will be less than X. Thus such D can not be an optimal split, as $P_{A,B+1}$ must be at least X.
Case 2: X > Y.
In this case $P_{A,B} = Y$. Let D < C. The split at C was optimal for [A,B), therefore the split at D was equal or worse. This means that $\min(S_{A,D}+P_{A,D}, S_{D,B}+P_{D,B}) <= Y$. Clearly, from D<C it follows that $S_{D,B}+P_{D,B} > S_{C,B}+P_{C,B}=Y$ for the same reasons as in the previous case. Thus we must have $S_{A,D}+P_{A,D} <= Y$.
This means that if we split [A,B+1) at D, our opponent can leave us with [A,D), which will give us total profit at most Y. However, the split at C will give us total profit more than Y, therefore again D can not be an optimal split.

PYRAMID

The Grand Egyptian Museum is under construction two kilometers away from Giza pyramids and it is planned to have the opening in 2010. Some of the Egyptian antiquities were already moved there like the Statue of Ramses II which was moved there in 2006. In Egypt there are more than 100 pyramids and for the purpose of this problem we are planning to move one of those near the entrance of the new museum.

The pyramid is of course incredibly heavy, and cannot be moved as a single block. Instead, it will be carefully sliced into horizontal sections (slices). An enormous crane will be used to move one slice at a time to the new location. Of course, the crane can only hold one slice at a time, and the slices must be reassembled in the proper order, so some temporary space is needed for keeping slices out of the way. Unfortunately, there is very limited space, so in total there can be at most three stacks of slices at any time: one at the original location, one at the new location, and one at the temporary space. The crane can only pick up the slice at the top of one stack and place it on the top of a different stack.

The engineers have measured each slice and determined both its weight and its strength. The strength of a slice is the maximum combined weight of slices that may be placed on top of it. Of course, every slice is strong enough to hold the slices that were originally on top of it, otherwise the pyramid would have collapsed long time ago.

TASK

Your task is to plan how to move the slices from the original location to the new location, in an appropriate order. Your goal is to complete the job using as few moves as possible (a move consists of picking up a slice from the top of one stack and placing it on the top of a different stack).

This is an output-only task. You are given 10 input files named "pyrX.in" where X varies from 1 to 10 with no leading zeros and you are to produce the corresponding output files "pyrX.out".

INPUT

The input files are formatted as follows:

- 1st line: 1 integer **N** (2 <= **N** <= 20), the number of horizontal slices.
- The next **N** lines describe the **N** horizontal slices in order from the top to the bottom of the pyramid. Each slice description consists of 2 integers that are the weight then the strength of this slice separated by single space. The weight will range from 1 to 100,000,000 inclusive and the strength will range from 0 to 100,000,000 inclusive.

OUTPUT

The output must list the moves, one per line. Each line contains two integers, the source then the destination stack, separated by single space. Note that stack 1 is the original location, stack 2 is the temporary space and stack 3 is the new location. The maximum allowed number of moves in any of your output files is 3,000,000 moves.

TESTING

You will be provided with a checker that can be used to test the validity of your input and output files. When you execute the checker, it will prompt you to enter the name of both the input and the output files. Then, the checker will test for validity and will show you the appropriate message.

GRADING

For each of your output files, you will get:

- 0 points if the output violates the rules specified above.
- 10 points if your number of moves is the fewest for that case among the contestants.
- **2 + (6*A)/B** points where **A** is the fewest number of moves for that case among the contestants and **B** is your number of moves. The value will be rounded to the nearest integer.

EXAMPLE

Sample Input	Sample Output 1	Sample Output 2
4 3 4 2 3 3 6 2 10	1 3 1 3 1 2 3 2 3 2 1 3 2 1 2 1 2 3 1 3 1 3	1 2 1 2 1 3 1 2 3 1 2 3 1 3 2 3 2 3

For the sample input, output 1 solves it in 11 moves while output 2 solves it in 9 moves. Assuming that 9 moves is the best solution for that case, then output 2 will receive 10 points and output 1 will receive 2 + (6*9)/11 rounded to the nearest integer which is 7 points.

SOLUTION

The Pyramid problem was a generalization of the well-known Towers of Hanoi problem, and in this solution we will use the traditional terminology – the slices we move will be called discs, and the three locations are pegs.

Note that we get the original problem for example in the case when all weights are equal to 1, and strengths are 0, 1, ..., N-1.

The original problem can be solved recursively: To transfer the entire tower from peg 1 to peg 3, we have to move the largest disc at some point. This is only possible if nothing is on top of it, and if peg 3 is free. This means that to transfer all N discs from peg 1 to peg 3, we first have to (1) transfer N-1 discs to peg 2, then (2) the largest disc from 1 to 3, and finally (3) transfer the N-1 discs from peg 2 to peg 3. Steps (1) and (3) are done recursively, that is, by applying the same algorithm. It can easily be seen that we need 2^N-1 moves to transfer the entire tower.

In our more general problem, this is always an upper bound on the optimal solution – as in the original tower each disc can hold all discs above it, we can always number them 1 to N in the order in which they appear in the input, and apply the original algorithm.

However, in many cases such a solution can be far from the optimum. For example, in a test case where the strength of each disc exceeds the sum of the others' weights, just N moves are enough to transfer the entire tower.

In general, the problem is hard, and no optimal solution in polynomial time is known to the authors. We will now explain several possible strategies how to approach this problem.

The total number of reachable states can be bounded from above by (N+2)!/2. The reasoning behind this is that each state can be written in the form “contents of peg1 | peg2 | peg3”, which is essentially a permutation of N numbers and two separator symbols. This means that roughly up to N=9 or even N=10 a complete state space search is possible, and one can best implement it using breadth first search.

For larger inputs such an approach would only work for inputs that resemble the original problem. Note that in the original Towers of Hanoi problem there are only 3^N valid configurations, as the configuration is uniquely specified by the peg numbers of the discs, in order. This means that for similar inputs the number of reachable states will actually be much lower than (N+2)!/2.

Further speedup can be achieved using good implementation techniques, such as packing the state into a few integers – with the complete state space search memory consumption starts to be an issue pretty soon.

For larger inputs, a viable alternative to the breadth first search is the A* search algorithm, using a suitable heuristic function h(S) that gives us a lower bound on the number of steps we need to make from state S to reach the goal. One simple function is to try all possibilities which of the incorrectly placed discs will be the next one on peg 3, and then counting the moves supposing that the discs have infinite

strengths. Such a heuristic function helps to cut the search space significantly, while keeping the solution correct – as soon as we reach the goal, we can be sure that the number of moves made is optimal.

For test cases that even the A* search is not able to solve in a few minutes, one has to use some heuristic approach. Some ideas are listed below:

- Try to find clusters of interchangeable discs and treat them as one new disc. Afterwards, if you managed to reduce the number of discs to a reasonably low number, apply the optimal algorithm.
- Apply the optimal algorithm to the top K discs, for some K. From now on, consider them as a single disc and move them all at once, using the optimal way you found. Recursively handle the remaining problem. Try various values of K and pick the best one.
- Use dynamic programming to find the best way to split the input into smaller clusters for the above approach.
- Just use dynamic programming to find the optimal way to split the input into chunks such that you can keep each chunk together and use the naïve algorithm to move them.
- Each time peg 3 is “free” (only contains discs that will not move any more) and you can move a disc that can hold all others, it is optimal to be greedy and move it to peg 3.

TYPE PRINTER

You need to print ***N*** words on a movable type printer. Movable type printers are those old printers that require you to place small metal pieces (each containing a letter) in order to form words. A piece of paper is then pressed against them to print the word. The printer you have allows you to do any of the following operations:

- Add a letter to the end of the word currently in the printer.
- Remove the last letter from the end of the word currently in the printer. You are only allowed to do this if there is at least one letter currently in the printer.
- Print the word currently in the printer.

Initially, the printer is empty; it contains no metal pieces with letters. At the end of printing, you are allowed to leave some letters in the printer. Also, you are allowed to print the words in any order you like.

As every operation requires time, you want to minimize the total number of operations.

TASK

You must write a program that, given the ***N*** words you want to print, finds the minimum number of operations needed to print all the words in any order, and outputs one such sequence of operations.

CONSTRAINTS

1 <= ***N*** <= 25,000 The number of words you need to print.

INPUT

Your program must read from the standard input the following data:

- Line 1 contains the integer ***N***, the number of words you need to print.
- Each of the next ***N*** lines contains a word. Each word consists of lower case letters (‘a’ – ‘z’) only and has length between 1 and 20, inclusive.
  All words will be distinct.

OUTPUT

Your program must write to the standard output the following data:

- Line 1 must contain an integer ***M*** that represents the minimum number of operations required to print the ***N*** words.
- Each of the next ***M*** lines must contain one character each. These characters describe the sequence of operations done. Each operation must be described as follows:
  - Adding a letter is represented by the letter itself in lowercase
  - Removing the last letter is represented by the character ‘-‘ (minus, ASCII code 45)
  - Printing the current word is represented by the character ‘P’ (uppercase letter P)

GRADING

For a number of tests, worth a total of 40 points, N will not exceed 18.

DETAILED FEEDBACK

During the contest, your submissions for this task will be evaluated on some of the official test data, showing you a summary of the results.

EXAMPLE

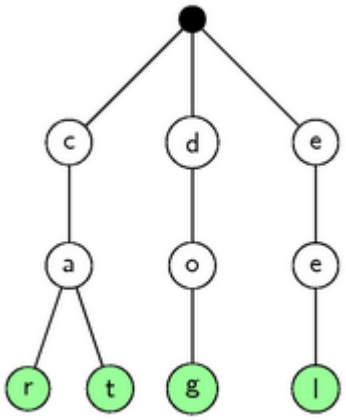
Sample input	Sample output
3 print the poem	20 t h e P - - - p o e m P - - - r i n t P

SOLUTION

Since there will be exactly N instructions to print a word, we may ignore these for the discussion of this problem. We define a word W to appear on the type printer if at some moment the printer contains exactly the word W. Our goal now is to make each of the words appear on the type printer.

Given a set of words, we can store them in a simple tree structure, known as a trie. A trie is a rooted tree where each edge is assigned a letter. Reading down from the root, these letters form the words stored in the trie.

Below you can find an example of a trie for the set of words {cat, car, do, dog, eel}. In the picture, the root node is the one at the top.



An empty type printer corresponds to the root node of the trie. Each command corresponds to moving one step in the trie: Adding a letter at the end corresponds to moving one level down in the trie, and deleting a letter corresponds to returning one level up. Try to trace this sequence of commands in the trie pictured above: c, a, t, -, r. You should end in the node that corresponds to the word 'car'.

Now consider a slightly different version of our task. Suppose that we want to find the minimal number of operations that prints all words, and after printing the last word returns the type printer to the empty state.

In our graphical representation this task can be rephrased as follows: find the shortest walk that starts in the root node, visits each node of our trie at least once, and ends in the root node again. Clearly, for each edge in our trie, at some point we have to walk down that edge to reach the nodes in the subtree below. And as the trie is a tree, we have no other way but to use the same edge later to get back from that subtree.

This means that each edge of our trie will be used at least twice. Therefore a solution that uses each edge exactly twice must be optimal. We obtain such a solution simply by running a depth-first search from the root node of the trie.

Now consider the original problem, where you can leave some letters in the printer at the end. The only thing that will change is that now we can save a few 'minus' commands – we can stop immediately after the last word appears, we do not have to delete it. And the longer the last word, the more 'minus' commands we save, and thus the shorter our solution will be. Therefore our goal is to maximize the length of the last word that appears on the printer.

It is always possible to end with the longest word. One way how to construct such a sequence of commands is to modify the depth-first search as follows: First, pick the longest word you want to print last. Mark the nodes of the trie that we visit when reading this word. Now, run the depth-first search with the additional constraint: 'if I am in a marked node that is not a leaf, first process all children that are not marked, and only at the end process the marked child node'. This will force the depth-first search to enter the marked branch at the end, and finish in the leaf node we wanted it to.

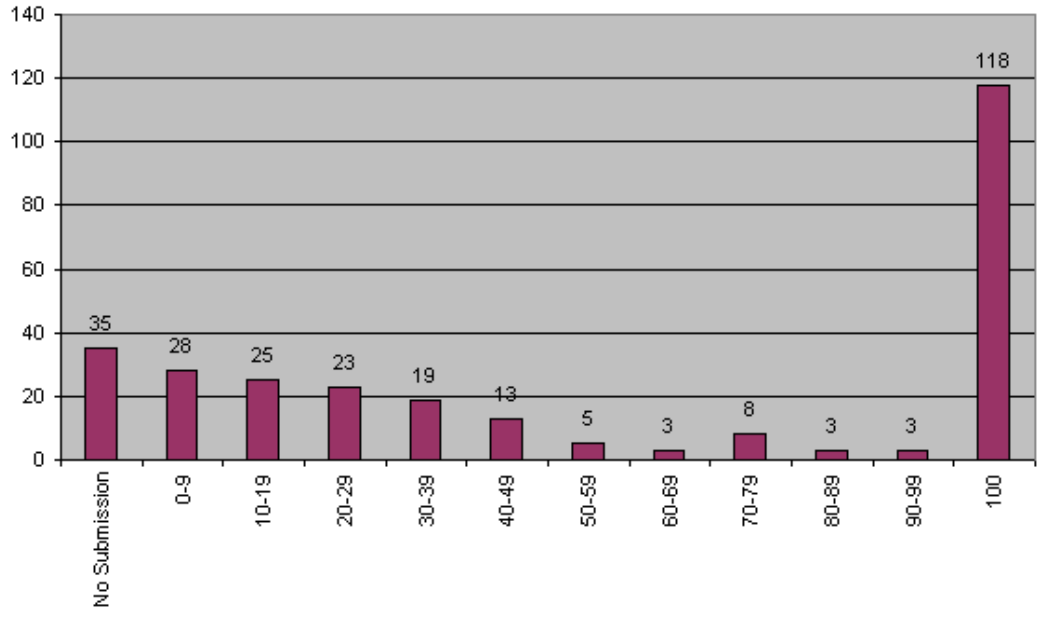
Note that it is possible to solve this task without actually implementing the trie. One possible solution that is really simple to implement:

Suppose that we pick one longest word W and then sort all the words in the input, using the length of the common prefix with W as the primary key, and the actual word as the secondary key. In other words, we sort words according to the number of first letters they share with W, breaking ties alphabetically. It can easily be proved that we will get one possible optimal order in which to print the words.

To see why this is the case, note that we are actually doing exactly the same thing as in the previous approach: first we print all the words that have the first letter different from the first letter of W, then all the words that share exactly the first letter with W, and so on. The alphabetical order within each group is there to ensure that words with a common prefix are always processed at the same time, as in the depth-first search solution above.

The solution with a trie is linear in the input size, the one with sorting has an additional logarithmic factor if you use a library function to do the sorting. Both approaches were expected to score 100 points.

SCORE DISTRIBUTION AMONG CONTESTANTS



Islands

You are visiting a park which has ***N*** islands. From each island ***i***, exactly one bridge was constructed. The length of that bridge is denoted by L_i. The total number of bridges in the park is ***N***. Although each bridge was built from one island to another, now every bridge can be traversed in both directions. Also, for each pair of islands, there is a unique ferry that travels back and forth between them.

Since you like walking better than riding ferries, you want to maximize the sum of the lengths of the bridges you cross, subject to the constraints below.

- You can start your visit at an island of your choice.
- You may not visit any island more than once.
- At any time you may move from your current island ***S*** to another island ***D*** that you have **not** visited before. You can go from ***S*** to ***D*** either by:
    - Walking: Only possible if there is a bridge between the two islands. With this option the length of the bridge is added to the total distance you have walked, or
    - Ferry: You can choose this option only if ***D*** is not reachable from ***S*** using any combination of bridges and/or previously used ferries. (When checking whether it is reachable or not, you consider all paths, including paths passing through islands that you have already visited.)

Note that you do not have to visit all the islands, and it may be impossible to cross all the bridges.

TASK
Write a program that, given the ***N*** bridges along with their lengths, computes the maximum distance you can walk over the bridges obeying the rules described above.

CONSTRAINTS

$2 <= N <= 1{,}000{,}000$	The number of islands in the park.
$1 <= L_i <= 100{,}000{,}000$	The length of bridge ***i***.

INPUT
Your program must read from the standard input the following data:

- Line 1 contains the integer ***N***, the number of islands in the park. Islands are numbered from 1 to ***N***, inclusive.
- Each of the next ***N*** lines describes a bridge. The i^{th} of these lines describes the bridge constructed from island ***i*** using two integers separated by a single space. The first integer represents the island at the other endpoint of the bridge, the second integer represents the length L_i of the bridge. You may assume that for each bridge, its endpoints are always on two different islands.

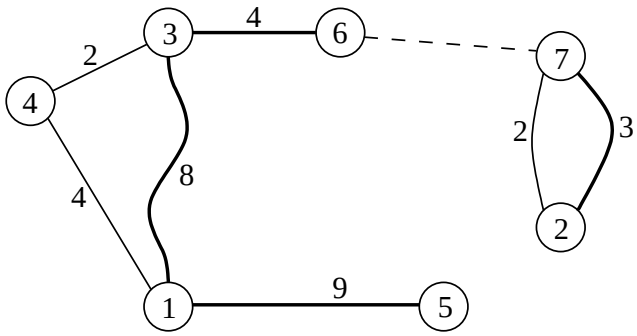
OUTPUT
Your program must write to the standard output a single line containing one integer, the maximum possible walking distance.

NOTE 1: For some of the test cases the answer will not fit in a 32-bit integer, you might need int64 in Pascal or long long in C/C++ to score full points on this problem.
NOTE 2: When running Pascal programs in the contest environment, it is significantly slower to read in 64-bit data types than 32-bit data types from standard input even when the values being read in fit in 32 bits. We recommend that you read the input data into 32-bit data types.

GRADING
For some cases worth 40 points, ***N*** will not exceed 4,000.

EXAMPLE

Sample input	Sample output
7 3 8 7 2 4 2 1 4 1 9 3 4 2 3	24



The ***N=7*** bridges in the sample are (1-3), (2-7), (3-4), (4-1), (5-1), (6-3) and (7-2). Note that there are two different bridges connecting islands 2 and 7.
One way how you can achieve maximum walking distance follows:

- Start on island 5.
- Walk the bridge of length 9 to reach island 1.
- Walk the bridge of length 8 to reach island 3.
- Walk the bridge of length 4 to reach island 6.
- Take the ferry from island 6 to island 7.
- Walk the bridge of length 3 to reach island 2.

By the end you are on island 2 and your total walking distance is 9+8+4+3 = 24.
The only island that was not visited is island 4. Note that at the end of the trip described above you can not visit this island any more. More precisely:

- You are not able to visit it by walking, because there is no bridge connecting island 2 (where you currently stand) and island 4.
- You are not able to visit it using a ferry, because island 4 is reachable from island 2, where you currently stand. A way to reach it: use the bridge (2-7), then use a ferry you already used to get from island 7 to island 6, then the bridge (6-3), and finally the bridge (3-4).

SOLUTION

We rephrase the problem in terms of graph theory, treating the islands as vertices and bridges as edges. Then the condition of taking a ferry becomes that you cannot add (and immediately traverse) an edge to a vertex that is already connected to your current vertex.

Consider the connected components of the graph. Since you cannot use ferry to jump within a connected component, your track through the component must form a simple path. And since you can start and end at any vertex of the connected component, the problem reduces to finding the longest weighted path in each connected component. The sum of these values over all connected components gives the desired answer.

This becomes the longest path problem, which is NP-complete for general graphs. It can be done using dynamic programming in O(2^E) time. To do better, we need to exploit the structure of the graph. As we are dealing with connected components only, we may assume the graph is connected.

It is quite intuitive to see, and not all that difficult to prove the following lemma:
(Several methods exist, with induction being the easiest and ugliest way to go.)
For any graph, any two of the following three statements imply the remaining one:

1. the graph has no cycles
2. the graph is connected
3. the number of edges is 1 less than the number of vertices in the graph.

(And in all three cases, the graph in question is a tree.)

Let the number of vertices in the connected component be N, then it must also have N edges, one associated with each vertex. From the lemma we get that it must contain a cycle. However, if we remove any edge on the cycle, we are not removing any connectivity, as any walk using that edge can go the 'other way' along the cycle. Thus after we remove the edge, we get a connected graph with N vertices and N-1 edges. By the lemma, there are no cycles left in the graph. Therefore the graph has exactly one cycle. Let C be the number of vertices on the cycle.

Note that this observation immediately yields a O(NC) solution for the component: No path can contain all edges of the cycle. Thus for each edge of the cycle, we can try to remove it and calculate the diameter of the resulting tree. The diameter of a tree on N vertices can be calculated in O(N). One tricky way to do it is to start from any vertex A, find the furthest vertex B from it, then find the furthest vertex C from B, and return the distance of B and C. (The proof of correctness of this algorithm is somewhat tricky.)

We will now show how to get a sub-quadratic solution. Assume that we label the vertices on the cycle V_1 to V_C, in order. Then the edges of the cycle are $(V_1,V_2),(V_2,V_3)....(V_{C-1},V_C)$ and finally (V_C,V_1). If we remove these edges, we get a cyclic sequence of rooted trees, one at each of the vertices. (Some of the trees can just be single vertices.)

There are 2 cases for the optimal path: either it lies entirely within one of these trees, or it crosses 2 trees by taking a section of the cycle. We deal with these cases separately.

Case 1: Within the same tree.

This reduces to the problem of finding the longest path in a tree. It can be done in O(N) time total by using the algorithm described earlier on each of the trees.

Case 2: Suppose the two trees involved are the ones rooted at V_i and V_j (where i<j).

Then the path within the trees should be as long as possible. So for each of the cycle vertices, we will compute the length of the longest path from it to some leaf in its tree. We will denote the length of the longest such path from V_k as $maxLen_k$.

There are 2 ways of traveling from V_i to V_j: (V_i, V_{i+1}, ..., V_j) and (V_i, V_{i-1}, ..., V_1, V_C, V_{C-1}, ..., V_{j+1}, V_j). If the edge from V_i to V_{i+1} has length L_i, the cost of these 2 paths are L_i+...+L_{j-1} and L_j+...+L_C+L_1+L_2+...+L_{i-1}, respectively.

Note this is almost identical to partial sums. We will track the partial sums of the sequence L_i using SL_i defined as follows: SL_1=0, SL_{i+1}=SL_i+L_i. Now if we set S=L_1+....+L_C, then the two above sums become SL_j-SL_i and S-(SL_j-SL_i), respectively.

If we iterate over all pairs of V_i and V_j, we get a O(C^2) solution for our problem. However, note that we are simply looking for the following value:

$max_{i<j}$ { $maxLen_i$ + $maxLen_j$ + max(SL_j-SL_i,S-SL_j-SL_i) }

Using some algebra, this can be rewritten as follows:

max($max_{i<j}$ { ($maxLen_i$-SL_i) + ($maxLen_j$+SL_j) },
S+$max_{i<j}$ { ($maxLen_i$+SL_i) + ($maxLen_j$-SL_j) })

Consider the first half,

$max_{i<j}$ { ($maxLen_i$-SL_i) + ($maxLen_j$+SL_j) }.

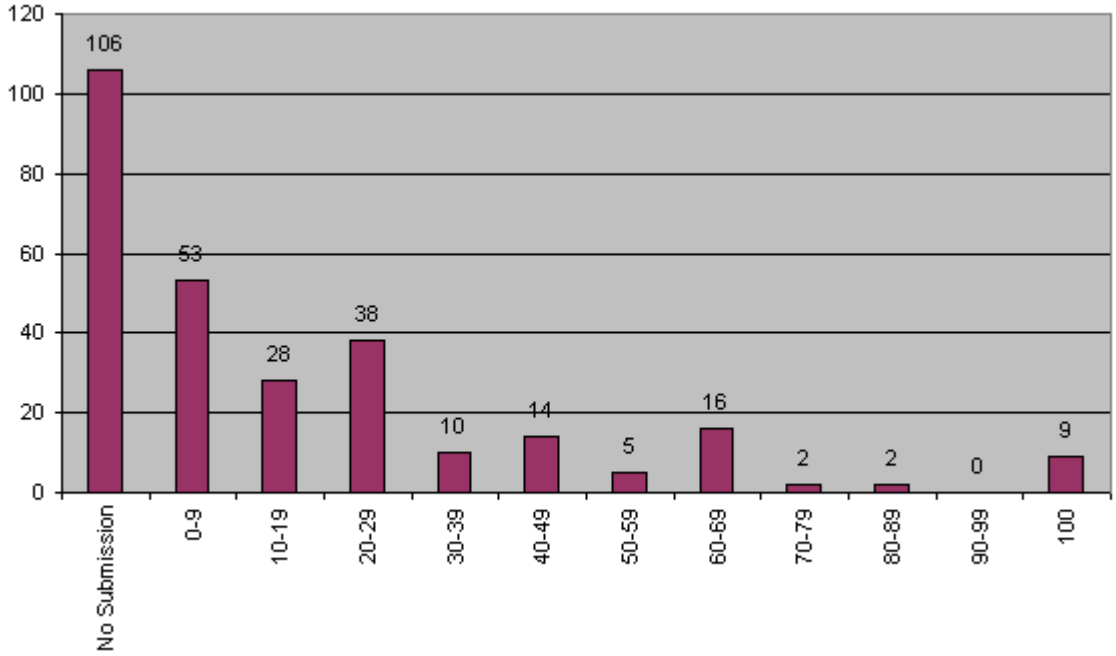
We can further manipulate this into:

max_j { $maxLen_j$ + SL_j + $max_{i \text{ such that } i<j}$ { $maxLen_i$-SL_i } }.

So as we loop upwards on V_j, the set of V_is that we need to consider is only increasing. So the value of $max_{i<}$j { $maxLen_i$-SL_i } can be updated in O(1) time as j increases. Hence, this transition can be computed in O(C) time, where C is the length of the cycle. Combining all pieces, we get a O(N) solution, which is expected to receive full points.

It is also possible to do the transition without decomposing the two maxes. This can be done by using a 'sliding window' along the cycle. (Note the ordering condition of the two vertices would no longer hold in this setup.) In this situation, we need to remove vertices from consideration as well. This can be implemented for example in O(N log N) by using a heap, or even in O(N) by implementing a fixed-range range minimum query using a double-ended queue. See also the solution for Pyramids, IOI 2006, for more ideas on a similar problem.

SCORE DISTRIBUTION AMONG CONTESTANTS



FISH

It was told by Scheherazade that far away, in the middle of the desert, there is a lake. Originally this lake had ***F*** fish in it. ***K*** different kinds of gemstones were chosen among the most valuable on Earth, and to each of the ***F*** fish exactly one gem was given for it to swallow. Note, that since ***K*** might be less than ***F***, two or more fish might swallow gems of the same kind.

As time went by, some fish ate some of the other fish. One fish can eat another if and only if it is at least twice as long (fish ***A*** can eat fish ***B*** if and only if $L_A >= 2 * L_B$). There is no rule as to when a fish decides to eat. One fish might decide to eat several smaller fish one after another, while some fish may decide not to eat any fish, even if they can. When a fish eats a smaller one, its length doesn't change, but the gems in the stomach of the smaller fish end up undamaged in the stomach of the larger fish.

Scheherazade has said that if you are able to find the lake, you will be allowed to take out one fish and keep all the gems in its stomach for yourself. You are willing to try your luck, but before you head out on the long journey, you want to know how many different combinations of gems you could obtain by catching a single fish.

TASK

Write a program that given the length of each fish and the kind of gemstone originally swallowed by each fish, finds **the number of different combinations of gems that can end up in the stomach of any fish, modulo some given integer *M*.** A combination is defined only by the number of gems from each of the ***K*** kinds. There is no notion of order between gems, and any two gems of the same kind are indistinguishable.

CONSTRAINTS

1 <= ***F*** <= 500,000	The original number of fish in the lake.
1 <= ***K*** <= ***F***	The number of different gemstone kinds.
2 <=***M*** <= 30,000	
1 <= L_X <= 1,000,000,000	The length of fish *X*.

INPUT

Your program must read from the standard input the following data:

- Line 1 contains the integer ***F***, the original number of fish in the lake.
- Line 2 contains the integer ***K***, the number of kinds of gemstones.

The kinds of gemstones are represented by integers 1 to ***K***, inclusive.

- Line 3 contains the integer ***M***.
- Each of the following ***F*** lines describes one fish using 2 integers separated by a single space: the length of the fish followed by the kind of gemstone originally swallowed by that fish.

NOTE: For all test cases used for evaluation, it is guaranteed that there is at least one gemstone from each of the ***K*** kinds.

OUTPUT

Your program must write to the standard output a single line containing one integer between 0 and ***M-1*** (inclusive): the number of different possible combinations of gemstones **modulo *M***.
Note that for solving the task, the value of ***M*** has no importance other than simplifying computations.

GRADING

For a number of tests, worth a total of 70 points, ***K*** will not exceed 7,000.
Also, for some of these tests, worth a total of 25 points, ***K*** will not exceed 20.

DETAILED FEEDBACK

During the contest, your submissions for this task will be evaluated on some of the official test data showing you a summary of the results.

EXAMPLE

Sample Input	Sample Output
5 3 7 2 2 5 1 8 3 4 1 2 3	4

There are 11 possible combinations so you should output 11 modulo 7 which is 4.

The possible combinations are: [1] [1,2] [1,2,3] [1,2,3,3] [1,3] [1,3,3] [2] [2,3] [2,3,3] [3] and [3,3].
(For each combination, we list the gems it contains. For example, [2,3,3] is a combination that consists of one gem of kind 2, and two gems of kind 3.)
These combinations can be achieved in the following ways:

- [1]: It is possible that you catch the second (or the fourth) fish before it eats any other fish.
- [1,2]: If the second fish eats the first fish, then it would have a gemstone of kind 1 (the one it originally swallowed) and a gemstone of kind 2 (from the stomach of the first fish).
- [1,2,3]: One possible way of reaching this combination: the fourth fish eats the first fish, and then the third fish eats the fourth fish. If you now catch the third fish, it will have one gemstone of each kind in its stomach.
- [1,2,3,3]: Fourth eats first, third eats fourth, third eats fifth, you catch the third one.
- [1,3]: Third eats fourth, you catch it.
- [1,3,3]: Third eats fifth, third eats fourth, you catch it.
- [2]: You catch the first fish.
- [2,3]: Third eats first, you catch it.
- [2,3,3]: Third eats first, third eats fifth, you catch it.
- [3]: You catch the third fish.
- [3,3]: Third eats fifth, you catch it.

SOLUTION

Observe that the order in which the fish eat each other is irrelevant. All that matters is whether the jewels of a given set of fish can be "united" into one fish (through the eating process described in the problem statement). We can arrive at the following lemma:

Lemma 1

The jewels that you get might be those of a given set of fish X if and only if the longest fish in X is at least twice as long as the second longest fish in X.

Proof

If the longest fish in X cannot eat the second longest, their jewels can never be united unless a fish not in X is involved. If the longest fish is at least twice as long as the second longest, then the longest one can eat everyone else in X successively.

The tricky part of this problem is to avoid counting some possible combination of jewels more than once. One way to avoid this is to map every possible combination of jewels to the longest fish that can have that combination in its stomach. We then attempt to count, for each fish, the number of combinations mapped to it.

Note: For simplicity we will say that a fish, which was originally given a jewel of kind X, is itself "a fish of kind X".

Lemma 2

Unless a fish is the longest one of its kind, it will have no combinations mapped to it.

Proof

If we have a fish A of kind J and a longer fish B also of kind J, then whatever can be eaten by A can also be eaten by B. So any jewel set that's mapped to A can also involve the longer fish B instead of A, which is a contradiction with the mapping method defined above.

Lemma 3

If the longest fish A of kind J_1 can eat N fish of kind J_1 and another fish B of kind J_2 can eat more than N fish of kind J_1, then no combinations that contain any jewel J_2 would be mapped to the longest fish of kind J_1. (Observe that in such a case the fish B must necessarily be longer than the fish A.)

Proof

This is a similar case to the one above. Any such set that's mapped to the longest fish of kind J_1 can also be found in the stomach of the longer fish of kind J_2, leading to another contradiction.

Lemma 4

If the longest fish of kind J_1 can eat N fish of kind J_1 and another longer fish of kind J_2 can also eat N, but not N+1 fish of kind J_1, then the only combinations that can potentially be mapped to the longest fish of kind J_1 would be ones which either have N+1 jewels J_1 or no jewels J_2.

Proof

Again, using the superset principle we find out that if a combination has at most N jewels of kind J_1 and at least one jewel of kind J_2, then it can be found in the stomach of the longer fish of kind J_2 and thus cannot be mapped to a fish of kind J_1.

Knowing this, we can build an algorithm to tell us which combinations are mapped to a given fish.

Note: For simplicity we will denote by $E(J_1, J_2)$ the number of fish of kind J_2 that can be eaten by the longest fish of kind J_1.

Following lemma 2, we're only interested in the longest fish of their respective kind. For each such fish F_1 of kind J_1, we count two types of combinations that are mapped to it. First, combinations that have the maximum number of jewels of kind J_1 (which is $E(J_1, J_1)$ plus one). These are called the "full" combinations. Second, all other combinations mapped to F_1. These are called the "partial" combinations.

Now, for every other kind J_2, with longest fish F_2, we count how many jewels of kind J_2 can be part of a "full" or a "partial" combination mapped to F_1. The above lemmas give rise to three scenarios:

* If $E(J_2, J_1)$ is more than $E(J_1, J_1) + 1$, meaning that F_2 can eat more fish of kind J_1 than F_1, then no jewels of kind J_2 can be part of any combination mapped to F_1.

* If F_2 is longer than F_1, but doesn't fall into the above category, there can be no jewels of kind J_2 in the "partial" category of F_1, but there can be anywhere between 0 and $E(J_1, J_2)$ jewels in the "full" category.

* If F_2 is shorter than F_1, then any number between 0 and $E(J_1, J_2)$ of fish of kind J_2 can participate in either "full" or "partial" combinations of F_1.

Except for J_1, the number of jewels of any two colors are independent of each other (i.e., the first count doesn't influence the feasibility of the second count in any way and vice versa).

A naïve implementation of this algorithm gives O(K*F) running time since for each longest fish of its kind, we look through all other fish and determine the E values. This should be sufficient for 56 points.

We can improve the naïve implementation if we realize that we need only one loop over all the individual fish, as long as the fish are sorted by length. Then we can go from smallest to longest and count how many fish of each kind we have seen so far. We will use $H(J_X)$ to denote how many fish of kind J_X we have seen. Then when we reach the largest fish that can be eaten by a fish of kind J, we have the values for $E(J, J_X)$ in the current values of $H(J_X)$. Implementing this properly yields a $O(F*\log F + K^2)$ algorithm with the O(F*log F) bottleneck being the sorting of the F fish and the $O(K^2)$ bottleneck being the evaluation of the products of the E numbers for each jewel kind. This solution is sufficient for 70~75 points.

We now work towards an O(F*log F) solution, which gets 100 points for this problem. The key observation here is that if we have the kinds sorted by the length of their respective longest fish, when we compute the number of combinations mapped to a given kind J_1, all we do is adding together a few numbers, each of which is a product of a some consecutive $E(J_1,J_X)$ numbers (where "consecutive" refers to J_X).

This means that we can achieve an O(F*log F) solution by keeping the array $H(J_X)$ (which at given times becomes $E(J_1, J_X)$ for each kind J_1) in a data structure that allows us to modify the array in O(log F) time and to extract the product of a continuous section of the array in O(log F) time as well. This can be done using a binary tree data structure with the leaves of the tree storing the $H(J_X)$ numbers and each node in the tree storing the product of the numbers in its sub-tree. A primitive illustration of this (with the leaves on top) would be as follows:

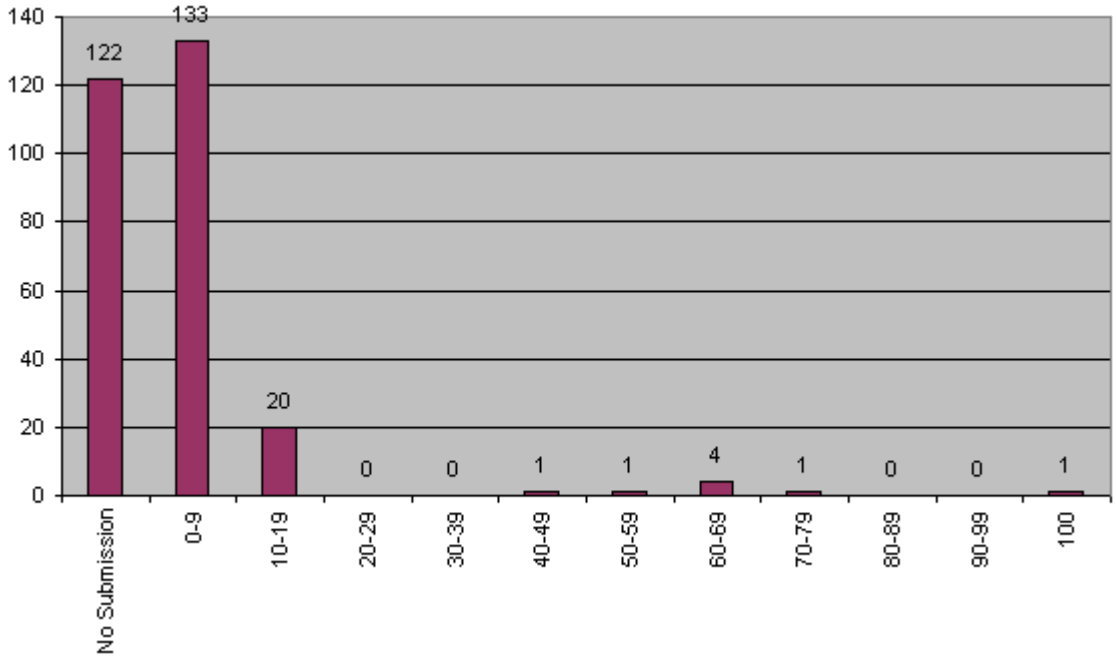
Note: [a, b] indicates that the node is keeping the product of H(a), H(a+1), H(a+2), …, H(b).

[1,1], [2,2], [3,3], …
[1,2], [3,4], [5,6]...
[1,4], [5,8], [9,12]...
[1,8], [9,16], [17,24]...
...
$[1,2^k]$, $[2^k+1,2*2^k]$, $[2*2^k+1,3*2^k]$...

Clearly each change in the array affects the value stored in O(log F) of these nodes, hence any updates to the data structure can be achieved in O(log F) time by combining the values of the node's children in every affected node, going from the affected leaf to the root. It can also be shown that any interval of the array can be decomposed into at most 2*log F of these intervals, so the product of values in any interval can be calculated in O(log F) time as well.

Final Note: The author also developed an extended version of this problem in which you catch 2 fish, rather than just 1, and the jewels from the two fish are counted together. It is also doable in polynomial time, although it is much more complicated. If you are enthusiastic about solving this version, you should feel free to send your solutions to Velin.Tzanov@deshaw.com.

SCORE DISTRIBUTION AMONG CONTESTANTS



LINEAR GARDEN

Ramesses II has just returned victorious from battle. To commemorate his victory, he has decided to build a majestic garden. The garden will contain a long line of plants that will run all the way from his palace at Luxor to the temple of Karnak. It will consist only of lotus plants and papyrus plants, since they symbolize Upper and Lower Egypt respectively.

The garden must contain exactly ***N*** plants. Also, it must be balanced: in any contiguous section of the garden, the numbers of lotus and papyrus plants must not differ by more than 2.

A garden can be represented as a string of letters 'L' (lotus) and 'P' (papyrus). For example, for ***N=5*** there are 14 possible balanced gardens. In alphabetical order, these are: LLPLP, LLPPL, LPLLP, LPLPL, LPLPP, LPPLL, LPPLP, PLLPL, PLLPP, PLPLL, PLPLP, PLPPL, PPLLP, and PPLPL.

The possible balanced gardens of a certain length can be ordered alphabetically, and then numbered starting from 1. For example, for ***N=5***, garden number 12 is the garden PLPPL.

TASK
Write a program that, given the number of plants ***N*** and a string that represents a balanced garden, calculates the number assigned to this garden **modulo some given integer *M*.**
Note that for solving the task, the value of ***M*** has no importance other than simplifying computations.

CONSTRAINTS
1 <= ***N*** <= 1,000,000
7 <= ***M*** <= 10,000,000

GRADING
In inputs worth a total of 40 points, ***N*** will not exceed 40.

INPUT
Your program must read from the standard input the following data:
- Line 1 contains the integer ***N***, the number of plants in the garden.
- Line 2 contains the integer ***M***.
- Line 3 contains a string of ***N*** characters 'L' (lotus) or 'P' (papyrus) that represents a balanced garden.

OUTPUT
Your program must write to the standard output a single line containing one integer between 0 and ***M-1*** (inclusive), the number assigned to the garden described in the input, **modulo *M***.

EXAMPLE

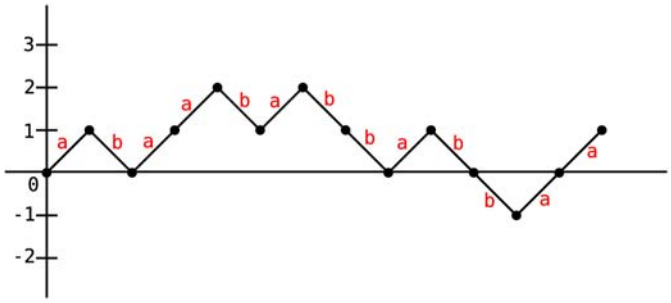
Sample input 1	Sample output 1
5 7 PLPPL	5

The actual number assigned to PLPPL is **12**. So, the output is **12** modulo **7,** which is **5**.

Sample input 2	Sample output 2
12 10000 LPLLPLPPLPLL	39

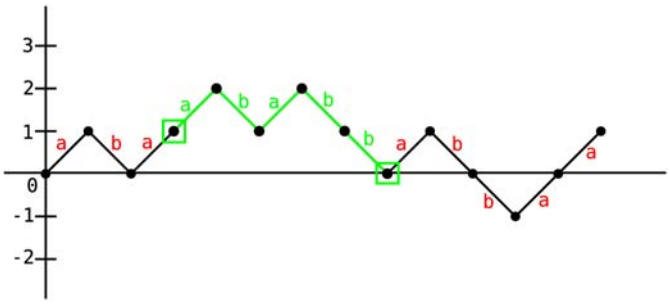
SOLUTION

For simplicity, we will consider balanced string consisting of lowercase *a*s and *b*s in this solution. We will start by analyzing what makes a string balanced.



The graph above shows a graphical representation of the string *abaababbabbaa*.
Formally, let A(S) and B(S) be the counts of letters *a* and *b* in the string S. Let Diff(S) = A(S) – B(S). The graph shows the values Diff(T) for each prefix T of the string S = *abaababbabbaa*.

Using the values shown in the graph, we can easily compute the difference between the number of *a*s and *b*s in any substring of S. More precisely, let S be the concatenation of strings T, U, and V. Then Diff(U) = Diff(TU) – Diff(T). In other words, we just need to look at the graph and read the values at the beginning and at the end of the substring in question.



In the graph above, the highlighted substring U = *ababb* starts at “height” 1, ends at “height” 0, and thus Diff(U) = 0–1 = -1.

Using this graphical representation, we can give a new definition of balanced strings. A balanced string is a string such that its graph lies entirely inside a horizontal strip of height 2.

We will now prove this. Suppose that for the string S the graph contains two vertices whose heights differ by x>=3. Then these two vertices determine a substring U where |Diff(U)|=x, and thus S is not balanced. On the other hand, if for any two vertices of the graph the height difference is at most 2, then for any substring U we have |Diff(U)|<=2, meaning that S is balanced.

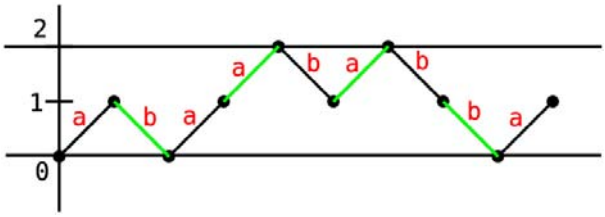
Using this knowledge, we can now try to solve the given task. To find the number assigned to the given string, we have to find an efficient way to count all balanced strings that are less or equal to the given string.

One possible way is to use dynamic programming. When generating a balanced string left to right, all we need to know at any given moment are three numbers: the lowest Diff value L in the graph so far, the highest Diff value H in the graph so far, and the current Diff value C. All of these numbers come from the range -2 to 2. Thus we can define sub-problems as follows: Let Count(K,L,H,C) be the number of ways in which we can fill in the last K letters if we are in a state described by L, H, and C. In this way we get 5^3*N sub-problems (in fact, less, not all triples L,H,C are valid), and each of them can be solved or reduced to smaller sub-problems in O(1).

Using the values Count(K,L,H,C) we can count all balanced strings less than the input string in linear time. For example, if the input string is S=*abaabab*, we have to count all the strings of the form *aa?????*, *abaaa??*, and *abaabaa*.

There is an even simpler solution, based on the fact that using one more observation we can count those sets of balanced strings directly.

First, consider a slightly easier problem: What is the count of all balanced strings of length N?
We will first count all balanced strings whose graphs lie in the strip between 0 and 2. Take a look at the following graph:



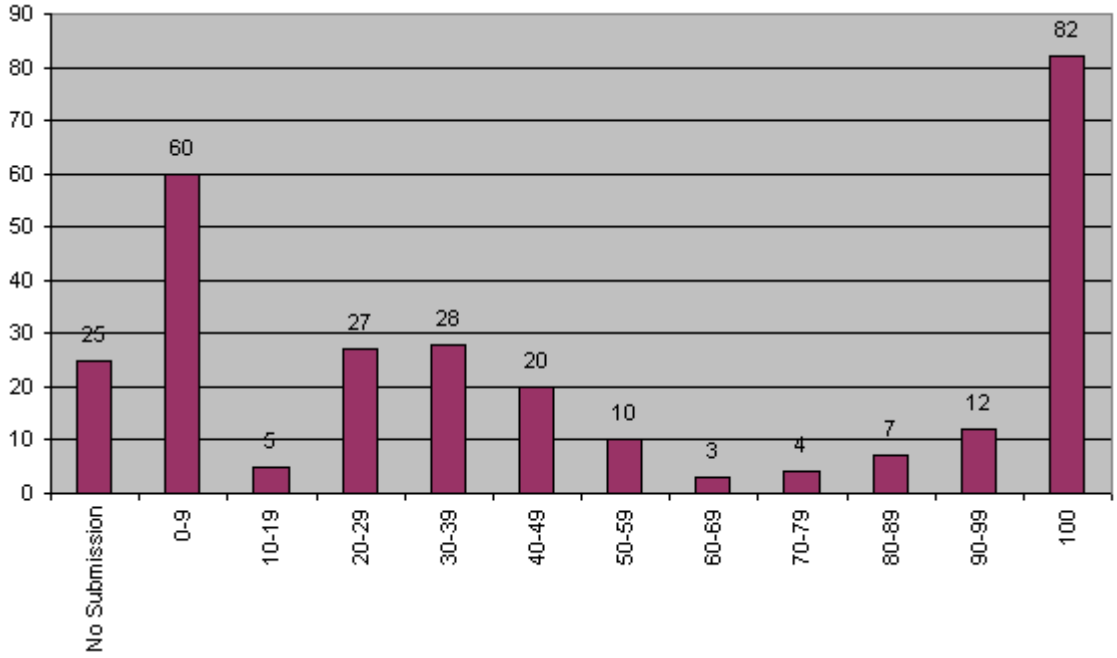
It is easy to realize that odd steps are always uniquely determined, and in even steps we always get to make a choice whether to add an *a* or a *b*. Thus the total count of such balanced strings is $2^{floor(N/2)}$. For strip -2 to 0 the count is the same from symmetry. For the strip -1 to 1, the odd steps are free, thus the count is $2^{ceil(N/2)}$. Of course, we counted two strings twice: the string *ababa...* and the string *babab...* Therefore the answer is $2^{floor(N/2)}+2^{floor(N/2)}+2^{ceil(N/2)}-2$.

Using a similar reasoning we can compute the number of ways how to finish any string to keep it balanced. Let the string so far be U, and let there be K remaining letters to add. There are three possible cases:

1. If the graph of U takes more than 2 rows, the answer is 0.
2. If it takes 2 rows, the answer is either $2^{ceil(K/2)}$ or $2^{floor(K/2)}$, depending on whether we are currently in the middle of the strip or not. For example, there are 2^3=8 balanced strings of the form *aab?????*.
3. If it takes 1 row, we have to count two types of strings. For example, if we are counting balanced strings of the form *aba?????*, we have to count all such strings in the strip 0 to 2, and all such strings in the strip -1 to 1. And obviously, in doing so we will count the string *ababa...* twice.
   Thus the answer in this case is always $2^{floor(K/2)}+2^{ceil(K/2)}-1$. For example, there are 4+8-1=11 balanced strings of the form *aba?????*.

In this way we get a very simple solution with time complexity O(N).

SCORE DISTRIBUTION AMONG CONTESTANTS



TELEPORTERS

You are participating in a competition that involves crossing Egypt from west to east along a straight line segment. Initially you are located at the westmost point of the segment. It is a rule of the competition that you must always move along the segment, and always eastward.

There are ***N*** teleporters on the segment. A teleporter has two endpoints. Whenever you reach one of the endpoints, the teleporter immediately teleports you to the other endpoint. (Note that, depending on which endpoint of the teleporter you reach, teleportation can transport you either eastward or westward of your current position.) After being teleported, you must continue to move eastward along the segment; you can never avoid a teleporter endpoint that is on your way. There will never be two teleporter endpoints at the same position. Endpoints will be strictly between the start and the end of the segment.

Every time you get teleported, you earn 1 point. The objective of the competition is to earn as many points as possible. In order to maximize the points you earn, you are allowed to add up to ***M*** new teleporters to the segment before you start your journey. You also earn points for using the new teleporters.

You can set the endpoints of the new teleporters wherever you want (even at non-integer coordinates) as long as they do not occupy a position already occupied by another endpoint. That is, the positions of the endpoints of all teleporters must be unique. Also, endpoints of new teleporters must lie strictly between the start and the end of the segment.

Note that it is guaranteed that no matter how you add the teleporters, you can always reach the end of the segment.

TASK

Write a program that, given the position of the endpoints of the ***N*** teleporters, and the number ***M*** of new teleporters that you can add, computes the maximum number of points you can earn.

CONSTRAINTS

$1 <= N <= 1{,}000{,}000$	The number of teleporters initially on the segment.
$1 <= M <= 1{,}000{,}000$	The maximum number of new teleporters you can add.
$1 <= W_X < E_X <= 2{,}000{,}000$	The distances from the beginning of the segment to the western and eastern endpoints of teleporter X.

INPUT

Your program must read from the standard input the following data:

- Line 1 contains the integer ***N***, the number of teleporters initially on the segment.
- Line 2 contains the integer ***M***, the maximum number of new teleporters that you can add.
- Each of the next ***N*** lines describes one teleporter. The i^{th} of these lines describes the i^{th} teleporter. Each line consists of 2 integers: W_i and E_i separated by a space. They represent respectively the

distances from the beginning of the segment to the western and eastern endpoints of the teleporter.

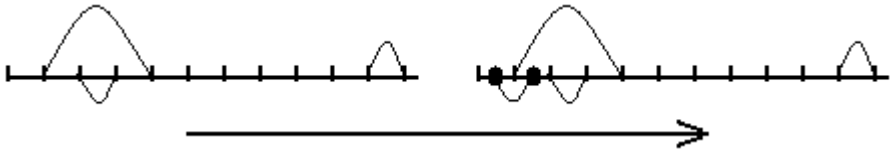
No two endpoints of the given teleporters share the same position. The segment that you will be travelling on starts at position 0 and ends at position 2,000,001.

OUTPUT
Your program must write to the standard output a single line containing one integer, the maximum number of points you can earn.

GRADING
In test data worth 30 points, ***N*** <= 500 and ***M*** <= 500.

EXAMPLE

Sample input 1	Sample output 1
3 1 10 11 1 4 2 3	6



The first figure shows a segment with the three original teleporters. The second figure shows the same segment after adding a new teleporter with endpoints at 0.5 and at 1.5.

After adding the new teleporter as shown in the figure, your travel would be the following:

- You start at position 0, moving eastward.
- You reach the endpoint at position 0.5 and get teleported to position 1.5 (you earn 1 point).
- You continue to move east and reach endpoint at position 2; you get teleported to position 3 (you have 2 points).
- You reach endpoint at position 4, and get teleported to 1 (you have 3 points).
- You reach endpoint at 1.5, and get teleported to 0.5 (you have 4 points).
- You reach endpoint at 1, and get teleported to 4 (you have 5 points).
- You reach endpoint at 10, and get teleported to 11 (you have 6 points).
- You continue until you reach the end of the segment finishing with a total score of 6 points.

Sample input 2	Sample output 2
3 3 5 7 6 10 1999999 2000000	12

SOLUTION

A series of N teleporters with 2N distinct endpoints divides the line into 2N+1 intervals. Since nothing interesting can happen to you while traversing one of these intervals, traversing one can be considered instantaneous.

From each of these intervals except the last one, we can draw an arrow to the next interval we would visit after traversing this interval and getting teleported. A few quick observations can now be made:

- Each arrow corresponds to an end of a teleport.
- There are exactly 2N arrows.
- Each interval except the first and last has exactly one arrow going out of it and one coming into it.
- The first interval only has an arrow going out of it.
- The last one only has one coming into it.

Consider the graph where intervals are vertices and our arrows are edges. This graph may consist of several components. It is obvious that one of them is always a path (and includes the first and the last interval). The other components, if present, must always be cycles.

When we add a new teleporter, we are essentially cutting up some intervals and 'rewiring' the edges. There are 3 cases to consider when adding the teleporter, and it's not very difficult to verify by casework what happens in each of these cases:

Case 1: The endpoints of the new teleporter are in two intervals that are not in the same component.
These two components are merged into one. The number of edges in the new component is two more than the total number of edges in both original components.

Case 2: The endpoints of the new teleporter are in two different intervals on the same component C.
Let X be the length of the path from the first to the second interval, i.e., the number of jumps we currently have to make in order to reach the second interval from the first one. By adding the new teleporter, we will remove these X jumps from the component C and replace them by a single jump using the new teleporter. Thus the number of edges in C decreases by X-1. Additionally, we get a new cycle with X+1 edges.

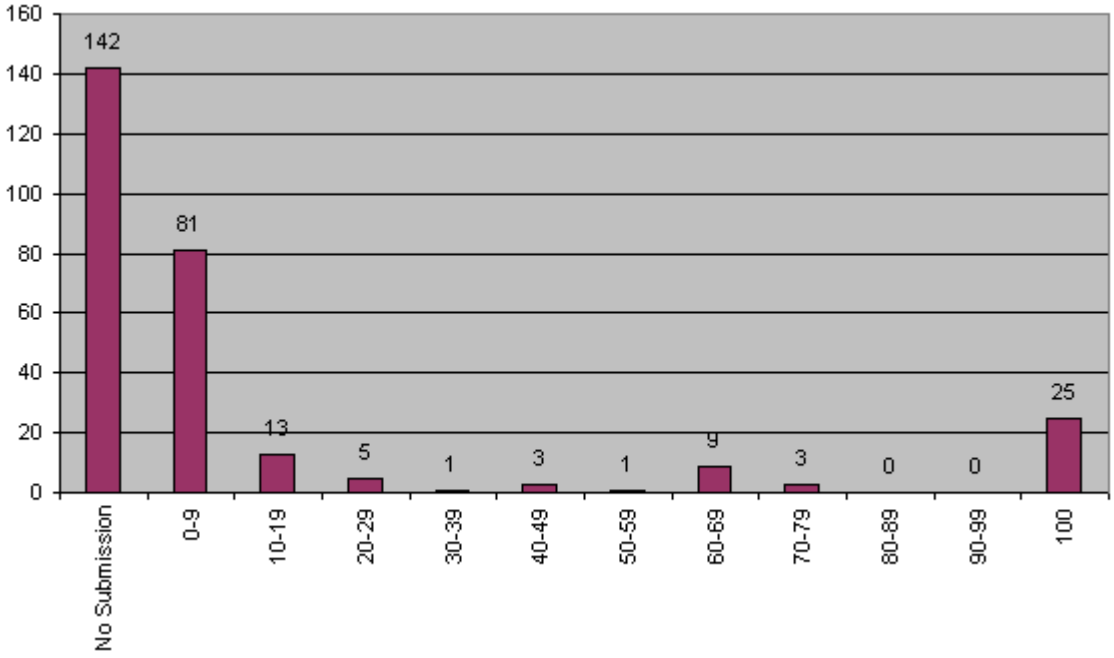
Case 3: The endpoints of the new teleporter are in the same interval.
The current component has its number of edges increased by 1, and another cycle of length 1 is created.

We can now use a greedy approach to find the best placement for the M new teleporters. If there are at least M cycles, it is clearly optimal to take the M largest cycles and connect them to the path (by placing the teleporters according to case 1 above). If we get to the situation where no cycles remain, we have to place the next teleporter according to case 3. In this way, we get a new cycle of length 1.

The components can be detected by a simple breadth first search. Then this process can be simulated in O(N log N + M), for example by sorting the cycle sizes, or by using a priority queue. It is possible to

improve the time complexity to O(N log N) by handling the case when no more cycles are left in constant time. Such a solution was expected to achieve the full score. Furthermore, counting sort can be used to sort cycle sizes to lower the time complexity to O(N).

SCORE DISTRIBUTION AMONG CONTESTANTS



PYRAMID BASE

You have been asked to find the largest affordable location for constructing a new pyramid. In order to help you decide, you have been provided with a survey of the available land which has been conveniently divided into an ***M*** by ***N*** grid of square cells. The base of the pyramid must be a square with sides parallel to those of the grid.

The survey has identified a set of ***P*** possibly overlapping obstacles, which are described as rectangles in the grid with sides parallel to those of the grid. In order to build the pyramid, all the cells covered by its base must be cleared of any obstacles. Removing the i^{th} obstacle has a cost C_i. Whenever an obstacle is removed, it must be removed completely, that is, you cannot remove only part of an obstacle. Also, please note that removing an obstacle does not affect any other obstacles that overlap it.

TASK

Write a program that, given the dimensions ***M*** and ***N*** of the survey, the description of the ***P*** obstacles, the cost of removing each of the obstacles, and the budget ***B*** you have, finds the maximum possible side length of the base of the pyramid such that the total cost of removing obstacles does not exceed ***B***.

CONSTRAINTS AND GRADING

Your program will be graded on three disjoint sets of tests. For all of them, the following constraints apply:

$1 \le M, N \le 1{,}000{,}000$	The dimensions of the grid.
$1 \le C_i \le 7{,}000$	The cost of removing the i^{th} obstacle.
$1 \le X_{i1} \le X_{i2} \le M$	X coordinates of the leftmost and the rightmost cells of the i^{th} obstacle.
$1 \le Y_{i1} \le Y_{i2} \le N$	Y coordinates of the bottommost and the topmost cells of the i^{th} obstacle.

<u>In the first set of tests worth 35 points:</u>

$B = 0$	The budget you have. (You cannot remove any obstacles.)
$1 \le P \le 1{,}000$	The number of obstacles in the grid.

<u>In the second set of tests worth 35 points:</u>

$0 < B \le 2{,}000{,}000{,}000$	The budget you have.
$1 \le P \le 30{,}000$	The number of obstacles in the grid.

<u>In the third set of tests worth 30 points:</u>

$B = 0$	The budget you have. (You cannot remove any obstacles.)
$1 \le P \le 400{,}000$	The number of obstacles in the grid.

INPUT

Your program must read from the standard input the following data:

- Line 1 contains two integers separated by a single space that represent ***M*** and ***N*** respectively.
- Line 2 contains the integer ***B***, the maximum cost you can afford (i.e., your budget).
- Line 3 contains the integer ***P***, the number of obstacles found in the survey.

- Each of the next ***P*** lines describes an obstacle. The i^{th} of these lines describes the i^{th} obstacle. Each line consists of 5 integers: X_{i1}, Y_{i1}, X_{i2}, Y_{i2}, and C_i separated by single spaces. They represent respectively the coordinates of the bottommost leftmost cell of the obstacle, the coordinates of the topmost rightmost cell of the obstacle, and the cost of removing the obstacle. The bottommost leftmost cell on the grid has coordinates (1, 1) and the topmost rightmost cell has coordinates (***M***, ***N***).

OUTPUT

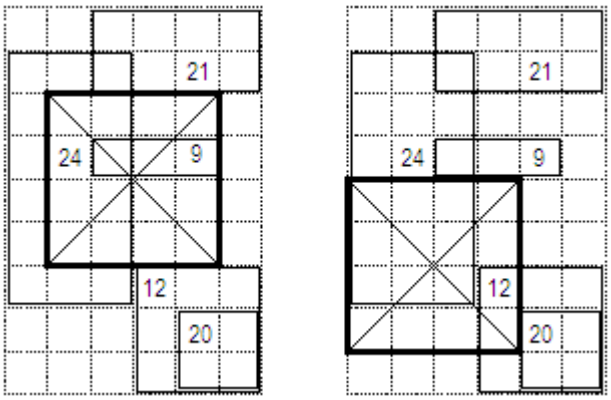
Your program must write to the standard output a single line containing one integer, the maximum possible side length of the base of the pyramid that can be prepared. If it is not possible to build any pyramid, your program should output the number 0.

DETAILED FEEDBACK

During the contest, your submissions for this task will be evaluated on some of the official test data showing you a summary of the results.

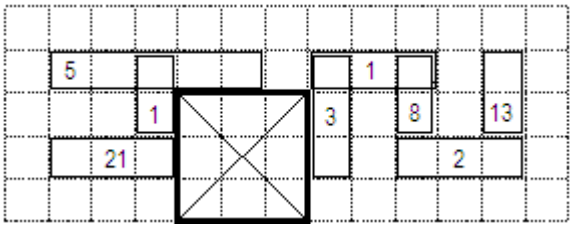
EXAMPLE

Sample input 1	Sample output 1
6 9 42 5 4 1 6 3 12 3 6 5 6 9 1 3 3 8 24 3 8 6 9 21 5 1 6 2 20	4



The figure shows two possible locations for the pyramid's base, both having a side of length 4.

Sample input 2	Sample output 2
13 5 0 8 8 4 10 4 1 4 3 4 4 1 10 2 12 2 2 8 2 8 4 3 2 4 6 4 5 10 3 10 4 8 12 3 12 4 13 2 2 4 2 21	3



The figure shows the only possible location for the pyramid's base having a side of length 3.

SOLUTION

We start off with a few observations regarding the placement of the optimal square:

Lemma 1

It suffices to consider squares whose left side touches either the left side of the field, or the right boundary of some obstacle.

Proof

We can always shift the optimal square leftwards until it hits such a boundary, without increasing the cost of placing it.

As x and y dimensions are independent in terms of translation, we also have:

Lemma 2

It suffices to consider squares whose bottom side touches either the bottom side of the field, or the upper boundary of some obstacle.

Proof

Similar to lemma 1, except we shift the optimal square downwards instead.

This observation alone yields a $O(P^3 \log P)$ solution since we could now enumerate the possible locations of the bottom left corner of the square in $O(P^2)$ time, then for each such location try increasing the square size and keep track of the obstacles we hit. It is not difficult to optimize this approach into a $O(P^3)$ solution.

We now try to get a faster solution by turning the problem into a decision problem.
Suppose that we have a square we can afford. If we now shrink it, while keeping its lower left corner in place, we will get a smaller square we can afford. This means that if X is the size of the optimal square, then for all Y<=X there is a square of size Y we can afford, and there is no such square for any Y>X. As a result, we can binary search on the side length of the square and then solve the following problem: 'Given a field with a set of rectangle obstacles, each with a given cost, find the square of side length Y such that the sum of costs of rectangles it intersects is minimum.'

It is possible to solve this problem directly by sweeping and using some clever data structures. However, one simple observation can give us a solution that is pretty easy to implement. The observation is that all we actually need is the location of the bottom left corner of this square. Note that for each point P we can easily compute which rectangles intersect a square of size Y with the bottom left corner in P – these are precisely the same rectangles that contain the point P, if we extend them by Y downwards and by Y to the left.

In other words, to solve our decision problem, we can extend each rectangle by Y both leftwards and downwards, and then solve a simpler problem: 'Given a field with a set of rectangles, each with a given cost, find the point covered by rectangles of minimum total cost.'

This problem can be solved by a range sweep going from left to right. As we encounter a rectangle's left x value, we increment the y range corresponding to it by its value, and when we pass a rectangle's right x value, we decrement the y range accordingly. So we need the following data structure:

Given an array of integers, support:

- Increasing/decreasing a section by a value.
- Query for the minimum value in the array.

Note that by lemma 2, it suffices to only consider y values that are right above a top edge of a rectangle, so we have O(P) of these entries in the array. There are 3 basic data structures that support these requirements:

A plain array, where each operation takes O(P).
A 2-level B-tree with each operation in O(sqrt(P)).
A range tree with each operation in O(log P).

More details on the construction of a range tree can be found in the solution for problem Fish.
The solution with the range tree runs in O(P * log P * log min(M,N)), the last factor being the binary search for the square size. This solution was expected to receive 70 points, the one with the B-tree 55, and the one with the array 35 points.

The remaining 30 points were awarded for large cases with zero allowed cost. We will now outline one approach to solve these cases. This approach will originate from a different approach to the general problem.

Let B be the largest cost we can afford. Define $f(x_1,x_2)$ as the maximum vertical size of a rectangle with cost at most B that has left edge on x_1 and right edge on x_2. Then we can prove the following about the function f:

Lemma 3
$f(x_1,x_2) >= f(x_1-1,x_2)$ and also $f(x_1,x_2) >= f(x_1,x_2+1)$.
Proof
Take any rectangle A with left edge at x_1-1 and right edge at x_2. If we throw away the leftmost column, we get an equally tall rectangle B with left edge at x_1 and right edge at x_2. Obviously, the cost of the new rectangle is at most the cost of the old rectangle. Thus whenever we can afford rectangle A, we can also afford rectangle B (and we may even be able to make rectangle B taller within our budget).

In terms of f, the goal of the problem can be rephrased as finding x_1 and x_2 such that $\min(x_2-x_l+1, f(x_1,x_2))$ is maximized. Another way of stating the same goal is that we try to maximize x_2-x_l+1 over all pairs (x_1,x_2) for which $f(x_1,x_2) >= x_2-x_l+1$.

To find this maximal value, for each x_1 we can find the largest x_2 such that $f(x_1,x_2) >= x_2-x_1+1$. We will use the notation $g(x_1)$ for the largest such x_2 to the given x_1.

Lemma 4
$g(x_1+1) >= g(x_1)$.

Proof

In words, if there is a valid square starting at x_1 and extending all the way to x_2, there is also such a square starting at x_1+1.

Formally, let $x_2=g(x_1)$. From the definition of $g(x_1)$, we have $f(x_1,x_2) >= x_2-x_1+1$. From Lemma 3 it follows that $f(x_1+1,x_2) >= f(x_1,x_2)$. Combining these, we get $f(x_1+1,x_2) > x_2-(x_1+1)+1$. This means that $g(x_1+1)$ is at least x_2, which is exactly what we needed.

By Lemma 4 we know that we can calculate g(x) for all x values by looping left to right on them, and incrementing the g value for the current x. This can be visualized as a double sliding window, as we insert rectangles as the right side of the window hits them and remove them as the left side leaves them. Clearly, each rectangle is inserted once and deleted once. The problem once again reduces to a data structure one:

We need a data structure that will maintain a set of weighted intervals and support:

- Insertion.
- Deletion.
- Finding the longest interval whose total cost is at most B.
  Note that this longest interval can be an arbitrary interval within the bounds of the field, and its cost is the sum of the costs of stored intervals it intersects.

In the general case where B > 0 this structure is quite difficult to maintain and the host committee was not able to find a reasonable data structure that does it better than in O(P) time per operation.

When B=0, the last requirement simplifies to finding the longest empty interval.
This is maintainable using a range tree, by tracking, for each segment S:

- The length of the empty segment that starts on the left end of S.
- The length of the empty segment that ends on the right end of S.
- The maximum length of an empty segment contained in S.

These values then propagate nicely up the tree and each operation takes O(log P) time. As in the previous solution, maintenance in O(sqrt(P)) time is also possible.

The cost for moving the left and right boundary during the sweep takes O(M) time if implemented naively. However, Lemma 1 along with the fact that the function f can only change when the right boundary of the sweep reaches the left boundary of some rectangle means that we can move the boundaries in jumps, and process O(P) interesting events only.

The motivation for creating this problem came from the maximum axes-parallel empty rectangle problem, which in its simplest form can be phrased as follows: Given a set of P point obstacles (having zero area, contrary to this problem) and a bounding rectangle, find the rectangle of maximum area that does not contain any of those points in its interior. Using the earlier lemmas, one could derive that all sides of the rectangle touch some of the points and obtain a $O(P^2)$ solution. Sub-quadratic solutions are possible; one possibility is to use repeated divide-and-conquer followed by 3-dimensional half-space queries. None of the known solutions were suitable for the IOI, as already the O(P log P)

implementations of 3D convex hulls are far from trivial. As a final note, Agarwal and Suri gave a O(P $\log^2$ P) solution in 1989.

[1] A. Aggarwal, S. Suri, Fast algorithms for computing the largest empty rectangle, Proceedings of the third annual symposium on Computational geometry, p.278-290, June 08-10, 1987, Waterloo, Ontario, Canada.

SCORE DISTRIBUTION AMONG CONTESTANTS