TASKS AND SOLUTIONS

CREDITS

Host Scientific Committee:

Bojan Antolović
Zvonimir Bujanović
Ante Đerek
Luka Kalinovčić
Lovro Pužar
Ivan Sikirić
Frane Šarić

Special thanks to:

Cesar Cepeda
Gordon V. Cormack
Michal Forišek
Derek Kisman
Martin Mareš
Pavel Pankov

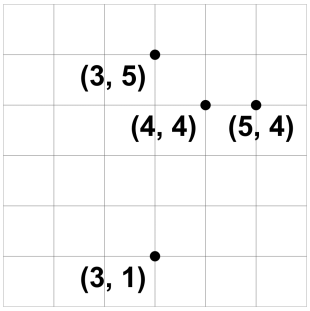
CONTENTS

CHIP

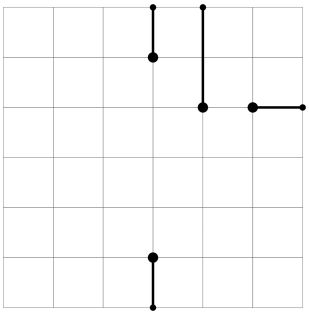
A chip is being produced on a square silicon plate.

The chip contains a number of power junctions, each described with a pair of integer coordinates. The first coordinate increases from left to right, while the second increases bottom up. The lower left corner of the chip is marked (0, 0).

For the chip to function properly, each power junction must be **connected to one of the four sides** of the chip using a **single straight** horizontal or vertical wire segment. Additionally, no two wires may overlap, intersect, or even touch.



Sample 6x6 chip with 4 power junctions


Connecting the power junctions to the chip's sides using 5 units of wire

TASK

You will be given the length of the sides of the chip and the locations of all power junctions. Find a way to connect the junctions to the sides, so that the **total length of wire used is the smallest possible**.

This is an output-only task. You will be given 10 input files and only need to produce the matching output files. You may download the input files from the contest system, on the page labeled "Tasks".

You need to submit each output file separately on the contest system. When submitting, the contest system will check the format of your output file. If the format is valid, the output file will be graded; otherwise, the contest system will report an error.

INPUT

The first line of input contains a single integer A ($2 \leq A \leq 30$), the length of the side of the chip.

The second line contains an integer N ($1 \leq N \leq 50$), the number of power junctions.

Each of the following N lines contains two integers X and Y ($1 \leq X, Y \leq A-1$), the coordinates of a power junction. No two power junctions will occupy the same position.

You may assume that there exists a solution for each input file.

OUTPUT

The first line of the output should contain the total length of wire used.

The following N lines should describe the connections. For each power junction, in the order in which they were given in the input, output one of `"up"`, `"down"`, `"left"` or `"right"`, the direction in which wire runs from that power junction.

If there is more than one optimal solution, output any one of them.

EXAMPLES

input

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

output

```
5
down
up
right
up
```

input

```
10
4
5 1
5 2
4 3
6 3
```

output

```
13
down
right
down
right
```

SOLUTION

Because the task is output-only, a well-implemented backtracking solution will be fast enough to run locally and solve all test cases. The task was originally batch, with a time limit of 1 second. The remainder of this text shows how to efficiently solve the problem.

First, sort the power junctions in increasing order by the x-coordinate. We will connect junctions to sides in that order. Now, suppose we've already connected the first K junctions and consider our options for connecting the junction K+1:

- Connecting to the right: Since all K junctions already connected have x-coordinate less than or equal to this junction, we can always do this.
- Connecting to the left: Some of the previous junctions that are connected up or down might get in our way. Of all junctions connected up we only have to check the one with the lowest y-coordinate, and of all junctions connected down we only have to check the one with the highest y-coordinate. No other junction is relevant in determining if we can connect junction K+1 to the left.
- Connecting up or down: Some previous junctions that are connected right might get in our way. Of all such junctions we have to check only the one with the highest y-coordinate if we want to connect junction K+1 up, and the one with the lowest y-coordinate if we are connecting junction K+1 down. No other junction is relevant in determining if we can connect this junction up or down.

These five parameters (K, lowest y-coordinate of all junctions connected up, highest y-coordinate of all junctions connected down, lowest y-coordinate containing of all junctions connected to the right, highest y-coordinate of all junctions connected to the right) define all possible states we can encounter in the process of connecting junctions.

These states, along with valid transitions between them, form a directed acyclic graph. We can calculate the optimal configuration using dynamic programming. The time complexity of this algorithm is $O(N \cdot A^4)$.

FISH

In a small coastal country, all towns are situated on a long coastline (which we will model as a straight line). A long straight road runs along the coast, connecting the towns. The position of each town can be described by a single non-negative integer – the distance (in kilometers) from the start of the road.

Most of the citizens are fishermen, and they catch great amounts of fish. After the fishing season is over and before the tourist season starts, the fish can be **transported** between different towns. A town can accommodate X tourists if it has X tons of fish available. The goal is to accommodate the **largest possible number of tourists** while **distributing them evenly** between towns. In other words, we want to find the **largest integer Y** for which it is possible to distribute fish so that each town can accommodate **at least Y tourists**.

In one shipment, an **integral** number of tons of fish is sent from one town to another. During transportation, **one ton of fish per kilometer traveled** is **lost** to hungry pillagers descending from the mountains. More formally, if a town ships F tons of fish to another town that is D kilometers away, then F-D tons will arrive at the destination; if F is less than D, then the entire shipment is lost.

It is possible to arbitrarily repackage and combine shipments in intermediate towns. For example, we can send shipments from towns A and B to town C, combine half of the remaining fish from both shipments with the fish originating in C and send it in a single large shipment from town C to town D.

TASK

Write a program that, given the positions of all towns and the amount of fish each town produces, determines the largest number of tourists that can be accommodated by each city after the fish has been distributed.

INPUT

The first line of input contains an integer N, $1 \leq N \leq 100\,000$, the number of towns.

Each of the following N lines contains two integers P and F, $0 \leq P, F \leq 10^{12}$, the position of a town (in kilometers) and the amount of fish it produces (in tons). The towns will be sorted in ascending order of position. The positions of all towns will be distinct.

OUTPUT

The first and only line of output should contain the largest number of tourists Y from the task description.

GRADING CRITERIA

In 50% of all test cases, N will be at most 100 and each town will produce at most 100 tons of fish.

DETAILED FEEDBACK WHEN SUBMITTING

Your first 10 submissions for this task will be evaluated during the contest (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLES

input

```
3
1 0
2 21
4 0
```

output

```
6
```

input

```
3
5 70
15 100
1200 20
```

output

```
20
```

input

```
4
20 300
40 400
340 700
360 600
```

output

```
415
```

SOLUTION

The key observation is that we can use a greedy algorithm to check if the towns can accomodate X tourists each. The algorithm considers towns from left to right. If a town has more than X units of food, it can send all of its excess food to the town directly to its right. If it has less than X units of food, it needs to receive food from the town to its right (note that, unless it is the leftmost town, it may have already implicitly received or sent out food to the cities to its left). If, after processing all towns, the rightmost town has enough food then X is a candidate solution.

Now note that if X is a candidate solution, X−1 is most certainly also a candidate solution. Similarly, if X is not a candidate solution, X+1 also cannot be a solution. With that in mind, we can use binary search to find the largest X that is a candidate solution. The time complexity of this algorithm is O(N log MAX), where MAX is any obvious upper bound on the solution.

ROBOT

Dave wishes to plunder an empty house. In order to discover the exact layout of the target house, he has teleported a remote-controlled robot inside.

The house is modeled as a grid composed of unit squares. Some squares are empty while others represent walls. The robot is initially **somewhere inside the house**, in an **empty square**.

Dave can **move the robot** by sending a simple command – the direction to move in (up, down, left or right). The robot can only move into an empty square; if it tries to move into a square occupied by a wall it will remain at its original location. After each command, it **reports back** whether move was successful or not.

TASK

Write a program that, given the ability to communicate with the robot, **determines the area of the room** into which the robot was teleported. The area of the room is the number of empty squares reachable from the robot's initial position, including the starting square. The area will be at most 1000 and you are allowed to use at most 5000 move commands.

INTERACTION

This is an interactive task. Your program sends commands to the robot using the standard output, and receives feedback from the robot by reading from the standard input.

To move the robot, you should output a single line containing one of the following commands: `"up"`, `"down"`, `"left"` or `"right"` (without the quotation marks), the direction in which the robot should move.

The robot will respond with a single line containing the word `"ok"` if the move was successful or the word `"fail"` otherwise.

When your program has found the solution, it should output a single line containing the area of the room and terminate its execution.

In order to interact properly with the grader, your program needs to flush the standard output after every write operation; the provided code samples show how to do this.

CODE SAMPLES

Code samples in all three languages are available for download on the "Tasks" page of the contest system. The code samples assume the room is rectangular and separately calculate the width and height of the room. This is not the correct solution and it will not score all points; the purpose of the samples is to show how to interact with the robot.

EXAMPLE

In the following example, commands are given in the left column, row by row. Feedback is given in the second column of the corresponding row.

output (command)	input (feedback)
up	fail
left	fail
right	ok
up	fail
right	ok
down	fail
up	fail
right	fail
left	ok
down	ok
down	fail
left	ok
left	fail
down	fail
5	

TESTING

You can use the TEST facility of the grading system to automatically run your solution with custom test cases. A single test case should be formatted as follows.

The first line of the test case should contain two integers R and C ($1 \leq R, C \leq 500$), the number of rows and columns, respectively.

The following R lines should contain C characters each. The character '.' represents an empty square, the character '#' represents a wall and the uppercase letter 'R' represents the starting location of the robot. There must be exactly one 'R' character on the map, and it needs to be inside of a room (i.e. it should be impossible for the robot to reach the edge of the map using valid moves).

Here is an example of a valid input file (corresponds to the example above). Notice that the robot is inside a room and that there is no way for it to move off the map.

```
6 8
........
.######..
.#R..#..
.#..####
.###....
..#.....
```

Valid input for test facility

The grading system will provide you with a detailed log of the execution.

SOLUTION

The room represents an undirected graph, where the vertices are empty squares, and edges exist between adjacent squares. We need to determine an algorithm for traversing the room which guarantees that:

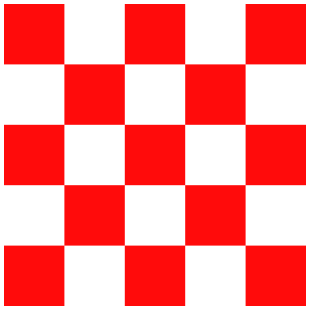
- The robot doesn't run into a loop;
- All squares in the room are found;
- No more than 5000 moves are used for a room of area 1000.

The two main algorithms for searching a graph are depth-first and breadth-first search, which both satisfy the first two constraints. Breadth-first is not applicable for this problem because it would require the robot to "jump around", and would use too many moves.

Depth-first search is the correct choice. The idea is to always move into any square which hasn't yet been visited, but retrace steps when there are no more such squares. This way we traverse each edge in the graph either zero or two times (once when entering a vertex and once when exiting). The number of edges visited twice is exactly one less than the number of vertices (because the vertices and edges used form a search tree). Adding the moves needed to explore the surrounding blocked squares, this still adds up to less than 5000 moves.

ALIENS

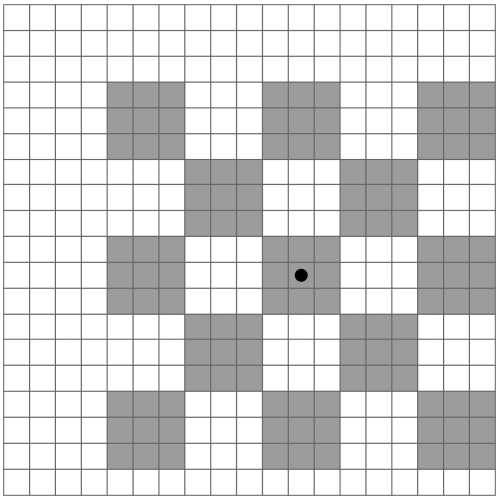
Mirko is a big fan of crop circles, geometrical formations of flattened crops that are supposedly of alien origin. One summer night he decided to make his own formation on his grandmother's meadow. The great patriot that he is, Mirko decided to make a crop formation that would have the shape of the shield part of the Croatian coat of arms, which is a 5×5 chessboard with 13 red squares and 12 white squares.



The chessboard part of the Croatian coat of arms.

Grandma's meadow is a square divided into N×N cells. The cell in the lower left corner of the meadow is represented by the coordinates (1, 1) and the cell in the upper right corner is represented by (N, N).

Mirko decided to flatten only the grass belonging to red squares in the chessboard, leaving the rest of the grass intact. He picked an **odd integer** $M \geq 3$ and flattened the grass so that each square of the chessboard comprises M×M cells in the meadow, and the chessboard completely fits inside the meadow.



Example meadow and Mirko's crop formation, with N=19 and M=3.
Cells with flattened grass are shown in gray.
The center of the formation is at (12, 9) and is marked with a black point.

After Mirko went to sleep, his peculiar creation drew the attention of real aliens! They are floating high above the meadow in their spaceship and examining Mirko's crop formation with a simple device. This device can only **determine whether the grass in a particular cell is flattened or not**.

The aliens have found **one cell with flattened grass** and now they want to find the **center cell** of Mirko's masterpiece, so that they may marvel at its beauty. They **do not know the size M** of each square in Mirko's formation.

TASK

Write a program that, given the size N ($15 \leq N \leq 2\,000\,000\,000$) of the meadow, the coordinates (X_0, Y_0) of one cell with flattened grass, and the ability to interact with the alien device, finds the coordinates of the center cell of Mirko's crop formation.

The device may be used at most 300 times in one test run.

INTERACTION

This is an interactive task. Your program sends commands to the alien device using the standard output, and receives feedback from the device by reading from the standard input.

- At the beginning of your program, you should read three integers N, X_0 and Y_0 from the standard input, separated by single spaces. The number N is the size of the meadow, while (X_0, Y_0) are the coordinates of one cell with flattened grass.
- To examine the grass in the cell (X, Y) using the alien device, you should output a line of the form `"examine X Y"` to the standard output. If the coordinates (X, Y) are not inside the meadow (the conditions $1 \leq X \leq N$ and $1 \leq Y \leq N$ are not satisfied), or if you use this facility more than 300 times, your program will receive a score of zero on that test run.
- The alien device will respond with a single line containing the word `"true"` if the grass in cell (X, Y) is flattened and the word `"false"` otherwise.
- When your program has found the center cell, it should output a line of the form `"solution` X_C Y_C`"` to the standard output, where (X_C, Y_C) are the coordinates of the center cell. The execution of your program will be automatically terminated once your program outputs a solution.

In order to interact properly with the grader, your program needs to **flush the standard output** after every write operation; the provided code samples show how to do this.

CODE SAMPLES

Code samples in all three programming languages are available for download on the "Tasks" page of the contest system. The purpose of the samples is only to show how to interact with the alien device; these are not the correct solutions and will not score all points.

GRADING

In test cases worth a total of 40 points, the size M of each of Mirko's squares will be at most 100.

Each test run will have a unique correct answer that will not depend on the questions asked by your program.

EXAMPLE

In the following example, commands are given in the left column, row by row. Feedback from the alien device is given in the second column of the corresponding row.

output (command)	input (feedback)
	19 7 4
examine 11 2	true
examine 2 5	false
examine 9 14	false
examine 18 3	true
solution 12 9	

TESTING

During the contest, there are three ways to test your solutions.

The first way is for you to simulate the alien device manually and interact with your program.

The second way is to write a program which will simulate the alien device. To connect your solution and the device you wrote, you may use a utility called `"connect"`, available for download on the contest system. To use the utility issue a command such as `"./connect ./solution ./device"` from the console (substituting `"solution"` and `"device"` with the names of your two programs). Any additional command-line parameters will be passed on to the device program.

The third way is to use the TEST facility of the grading system to automatically run your solution with a custom test case. When using the facility, the size of the meadow N is limited to 100.

A test case should contain three lines:

- The first line contains the size N of the meadow and the size M of a square in the chessboard;
- The second line contains the coordinates X_0 and Y_0 of one cell of the meadow with flattened grass, which will be given to your program;
- The third line contains the coordinates X_C and Y_C of the center cell of the chessboard.

The grading system will provide you with a detailed log of the execution, including error messages if:

- N doesn't satisfy the constraints;
- M is not an odd integer greater than or equal to 3;
- The crop formation doesn't fit in the meadow;
- The grass in cell (X_0, Y_0) is not flattened.

Here is an example of a valid input file for the test facility. The example corresponds to the figure on the first page.

```
19 3
7 4
12 9
```

Valid input for the test facility.

SOLUTION

We'll use the terms **red cells** and **red squares** for cells and squares with flattened grass. The other cells and squares are white. Using the alien device at some coordinates is now equivalent to checking the color of a cell.

The process of finding the solution consists of two stages. The first stage is to determine the side length M of one square of the chessboard and the center (X_C, Y_C) of the red square F containing the given red cell (X_0, Y_0). The second stage is to move from (X_C, Y_C) to the center of the entire crop formation (using shifts of length M or 2M along each coordinate).

The key part of this task is efficiently solving the first stage. In order to find M, we should find:

- X_R – the rightmost x-coordinate of cells in F,
- X_L – the leftmost x-coordinate of cells in F, and
- Y_B – the bottom y-coordinate of cells in F.

We describe several approaches for finding X_R; it is trivial to adapt them for finding X_L and Y_B.

The most obvious algorithm checks the colors of cells (X_0+1, Y_0), (X_0+2, Y_0), (X_0+3, Y_0) etc., stopping at the first white cell (X_R+1, Y_0) and using M queries in the worst case. This approach would score 40 points, as described in the "Grading" section of the task description.

To cut down on the number of queries, let's instead check the color of cells (X_0+1, Y_0), (X_0+2, Y_0), (X_0+4, Y_0), ..., (X_0+2^k, Y_0), ..., stopping at the cell we call C_{last}. This is the first white cell of the sequence or the first cell outside the meadow boundaries (we must be careful not to query the device in the latter case). It can be shown that all red cells left of C_{last} must belong to the square F and not to some other red square. Let $C_{previous}$ be the rightmost red cell left of C_{last}. We have used at most $\log_2 M$ queries so far.

Now we could reset $X_0 = C_{previous}$ and repeat the procedure described in the last paragraph until [$C_{previous}$, C_{last}] contains only 2 cells, after which $C_{previous}$ = (X_R, Y_0). However, here we have used $(\log_2 M)^2$ queries in the worst case, which is only good enough to score 70 points.

To get the full score, use binary search on the segment [$C_{previous}$, C_{last}] to find X_R: (X_R, Y_0) is the only red cell in that segment that has a neighboring white cell to the right. Using this algorithm we can find all of X_R, X_L and Y_B using at most 6 $\log_2 M$ queries total. Now, $M=X_R-X_L+1$, $X_C=X_L+(M-1)/2$, $Y_C=Y_B+(M-1)/2$.

This concludes the description of the first stage.

The second stage uses the alien device only a few times to detect which of 13 possible squares has (X_C, Y_C) as the center. Just checking the color of several cells having X_C, $X_C \pm M$, $X_C \pm 2M$ as x-coordinates and Y_C, $Y_C \pm M$, $Y_C \pm 2M$ as y-coordinates will do the trick.

Background

This problem is a reformulation of the task "SQUARES" externally submitted by Pavel S. Pankov from Kyrgyzstan.

FLOOD

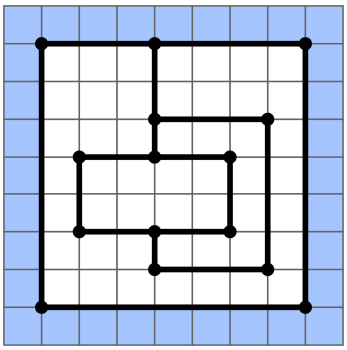
In 1964 a catastrophic flood struck the city of Zagreb. Many buildings were completely destroyed when the water struck their walls. In this task, you are given a simplified model of the city before the flood and you should determine which of the walls are left intact after the flood.

The model consists of N points in the coordinate plane and W walls. **Each wall connects a pair of points and does not go through any other points**. The model has the following additional properties:

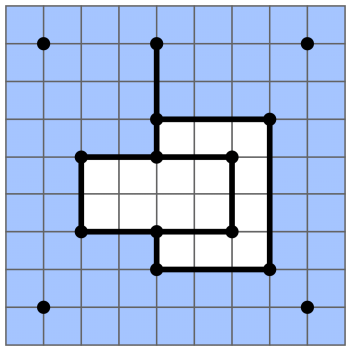
- No two walls intersect or overlap, but they may touch at endpoints;
- Each wall is parallel to either the horizontal or the vertical coordinate axis.

Initially, the entire coordinate plane is dry. At time zero, water instantly floods the exterior (the space not bounded by walls). After exactly one hour, every wall with water on one side and air on the other breaks under the pressure of water. Water then floods the new area not bounded by any standing walls. Now, there may be new walls having water on one side and air on the other. After another hour, these walls also break down and water floods further. This procedure repeats until water has flooded the entire area.

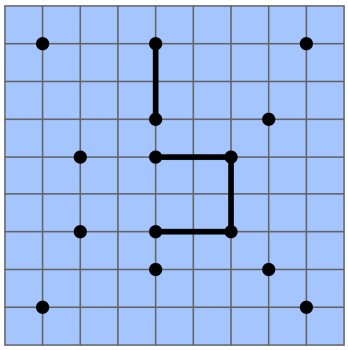
An example of the process is shown in the following figure.



The state at time zero. Shaded cells represent the flooded area, while white cells represent dry area (air).



The state after one hour.



The state after two hours. Water has flooded the entire area and the 4 remaining walls cannot be broken down.

TASK

Write a program that, given the coordinates of the N points, and the descriptions of W walls connecting these points, determines which of the walls are left standing after the flood.

INPUT

The first line of input contains an integer N ($2 \leq N \leq 100\,000$), the number of points in the plane.

Each of the following N lines contains two integers X and Y (both between 0 and 1 000 000, inclusive), the coordinates of one point. The points are numbered 1 to N in the order in which they are given. No two points will be located at the same coordinates.

The following line contains an integer W ($1 \leq W \leq 2N$), the number of walls.

Each of the following W lines contains two different integers A and B ($1 \leq A \leq N$, $1 \leq B \leq N$), meaning that, before the flood, there was a wall connecting points A and B. The walls are numbered 1 to W in the order in which they are given.

OUTPUT

The first line of output should contain a single integer K, the number of walls left standing after the flood.

The following K lines should contain the indices of the walls that are still standing, one wall per line. The indices may be output in any order.

GRADING

In test cases worth a total of 40 points, all coordinates will be at most 500.

In those same cases, and cases worth another 15 points, the number of points will be at most 500.

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 10 submissions for this task to be evaluated (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLE

input

```
15
1 1
8 1
4 2
7 2
2 3
4 3
6 3
2 5
4 5
6 5
4 6
7 6
1 8
4 8
8 8
17
1 2
2 15
15 14
14 13
13 1
14 11
11 12
12 4
4 3
3 6
6 5
5 8
8 9
9 11
9 10
10 7
7 6
```

output

```
4
6
15
16
17
```

This example corresponds to the figure on the previous page.

SOLUTION

Simulation

The simple solution is to "draw" the walls onto a two-dimensional array and use a 0-1 BFS algorithm (starting from any outer cell) to calculate the flooding time for each cell. A wall is left standing after the flood if the times needed to reach its two sides are equal.

The 0-1 BFS algorithm is used to find the shortest path in graphs in which edge has weight exactly 0 or 1 (using a double-ended queue and inserting elements at the front when expanding through a 0-edge). In our model, each pair of two adjacent cells is connected by an edge; weight of the edge is 1 if there is a wall between the cells, and 0 otherwise.

Since the size of the graph depends on the coordinates of the points, the number of cells we need to consider is too big in general. This solution was awarded approximately 40 points.

Simulation with coordinate compression

To improve the above algorithm, we can observe that the exact coordinates of points are irrelevant – only the relative order of points matters for the purpose of counting the walls standing after the flood. Therefore, if the x-coordinates take A different values and the y-coordinates take B different values, we can map them to sets {1, 2, ..., A} and {1, 2, ..., B}, respectively. The number of cells we need to consider while searching is now $O(A \cdot B)$.

As both A and B are bounded by N (the number of points), this approach has total time complexity $O(N^2)$. This solution was awarded approximately 55 points.

Model solution (dual-graph)

We represent each region (a maximal set of connected cells) as a node in a graph and we add edges between pairs of nodes if the corresponding regions share a wall. We also add the outer region as a node in the graph and connect it with all regions immediately exposed to water.

Notice that we do not need to find the exact flooding time for each region – we are only interested if two regions sharing a wall are equally distant from the outer region or not. If one connected set of regions is completely contained inside another region we can consider it separately while generating the above graph and connect it directly to the outer region.

Running a simple BFS algorithm on the above graph starting from outer region gives us enough data to solve the task. The hardest part of the solution is generating the graph.

One possible strategy is to imagine each wall as two one-way roads (going in opposite directions). Now, we pick any road and drive along, turning right at each intersection – if we can turn right we do so, else if we can go forward we do so, else if we can turn left we do so, else we turn backwards. For each starting road we will make one lap around one region and end up where we started. By successively choosing new roads and traversing them, we discover all possible regions. Exactly one traversal will give the an outer region. If for a particular wall, the two roads yield different regions, we connect those regions by an edge.

This algorithm can be implemented with time complexity O(N+W).

Alternative model solution (Outer wall traversal)

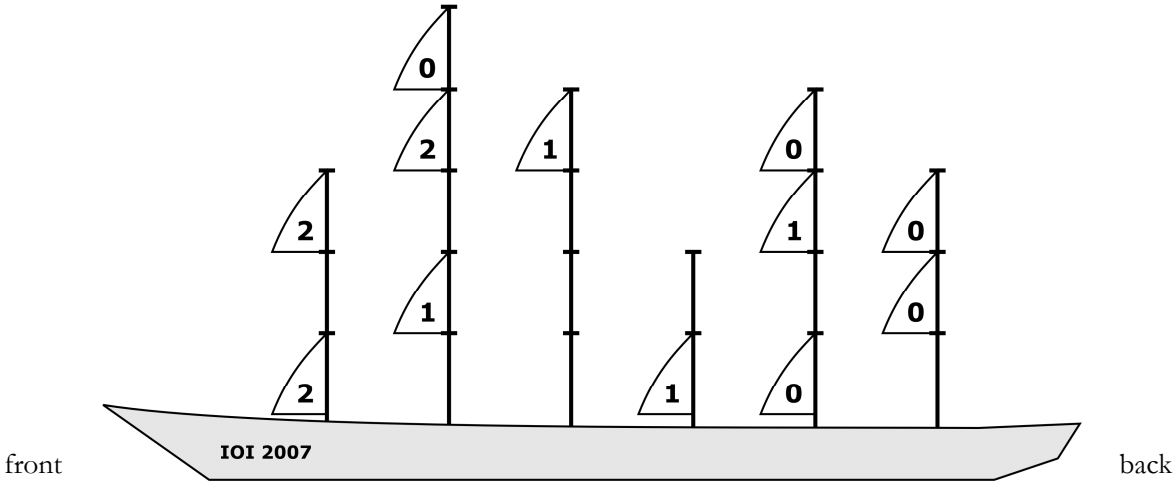
Another, somewhat different, approach to the problem is to continuously walk around the components and figure out which of the outer walls are going to collapse. We start by selecting a leftmost outer wall. Now, imagine we put a right hand on the wall and start walking around the walls keeping the hand on the wall until we return to the starting position. We observe that, of all the walls we touched, only those that we touched on both sides will survive the flood. Next, we remove all the walls we touched, and repeat the process until all walls are removed. This algorithm can be implemented in O(N log N + W) and also yields full points.

SAILS

A new pirate sailing ship is being built. The ship has N masts (poles) divided into unit sized segments – the height of a mast is equal to the number of its segments. Each mast is fitted with a number of sails and each sail exactly fits into one segment. Sails on one mast can be arbitrarily distributed among different segments, but each segment can be fitted with at most one sail.

Different configurations of sails generate different amounts of thrust when exposed to the wind. Sails in front of other sails at the same height get less wind and contribute less thrust. For each sail we define its **inefficiency** as the total number of sails that are **behind** this sail and **at the same height**. Note that "in front of" and "behind" relate to the orientation of the ship: in the figure below, "in front of" means to the left, and "behind" means to the right.

The **total inefficiency** of a configuration is the sum of the inefficiencies of all individual sails.



This ship has 6 masts, of heights 3, 5, 4, 2, 4 and 3 from front (left side of image) to back.
This distribution of sails gives a total inefficiency of 10. The individual inefficiency of each sail is written inside the sail.

TASK

Write a program that, given the height and the number of sails on each of the N masts, determines the **smallest** possible total inefficiency.

INPUT

The first line of input contains an integer N ($2 \leq N \leq 100\,000$), the number of masts on the ship.

Each of the following N lines contains two integers H and K ($1 \leq H \leq 100\,000$, $1 \leq K \leq H$), the height and the number of sails on the corresponding mast. Masts are given in order from the front to the back of the ship.

OUTPUT

Output should consist of a single integer, the smallest possible total inefficiency.

Note: use a 64-bit integer type to calculate and output the result (`long long` in C/C++, `int64` in Pascal).

GRADING

In test cases worth a total of 25 points, the total number of ways to arrange the sails will be at most 1 000 000.

EXAMPLE

input

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

output

```
10
```

This example corresponds to the figure on the previous page.

SOLUTION

Each segment is positioned at a particular **level** – a mast of height H has segments at levels 1 through H. Notice that the ordering of masts is irrelevant when counting the total inefficiency; the final result depends only on the number of sails at each level.

Let us assume that the masts are sorted in ascending order by height. Consider the following greedy algorithm:

- Process masts left to right keeping track of the total number of sails at each level.
- For each new mast of height H with K sails, do the following: out of the H current levels, choose K levels with the lowest number of sails and place the new sails there.
- Add up the values $S_X \cdot (S_X - 1) / 2$ for each level X, where S_X is the number of sails placed at level X. Report the sum as the solution.

We will omit the full proof of correctness for the algorithm. Intuitively, our goal is to keep the number of sails at different levels as close as possible, therefore it makes sense to place new sails at levels with the lowest numbers of sails so far.

In order to obtain a formal proof, show that any other choice is no better than the one this algorithm makes. Assume that A and B are different levels such that the number of sails placed on level A so far is less than the number of sails placed on level B so far, and argue that any configuration C that places a sail on B but not on A can be transformed into a configuration C' that does the opposite and has lower or equal inefficiency.

Suboptimal solutions

Although the algorithm is conceptually simple, it is not easy to find an implementation efficient enough for the given input constraints.

The above algorithm can be directly simulated. We can explicitly maintain the total number of sails for each level and update it at each step. If we use an array, we obtain an algorithm of time complexity O(total_number_of_sails · max_height). Such solutions were awarded approximately 30 points.

Notice that, since the masts are sorted by height, we can only keep track of the sail counts, and ignore the particular levels where they are obtained (that is, we can keep the histogram sorted). Hence, we can use a priority queue to maintain the histogram and efficiently find the K lowest sail counts at each step. This approach can be implemented with the time complexity of O(total_number_of_sails · log(max_height)). Such solutions were awarded 40 points.

Keeping the histogram in a sorted array gives another suboptimal solution. At each step, we can find the K lowest sail counts and update the histogram in one sweep. We just need to ensure that, among the levels with same sail count, we pick and update earlier levels first (assuming the histogram is sorted by descending sail counts). This approach can be implemented with the time complexity O(N · max_height). Such solutions were awarded 50 points.

Model solution

The model solution also uses the idea that the histogram can be kept sorted, but, instead of keeping the sail counts, it only maintains the differences between two successive sail counts. For example if, in the current state, the height is 6 and the sail counts are 5, 3, 3, 3, 2 and 2, the difference array, **delta**, is 5, -2, 0, 0, -1, 0 and -2.

The essence of the model solution is a subroutine that transforms the difference array as we process new masts. When processing a new mast of height H with K sails, we first add new entries to the end of the difference array (corresponding to the highest levels that are currently empty), and then we add one sail to each of the lowest K levels.

In order to add one sail to each level in the interval [A, B], in most cases we can simply increase delta[A] and decrease delta[B]. When delta[A] is zero (i.e. when the left part of the interval is in the middle of a group of equal sail counts), then the procedure results in an array that is not sorted. However, whenever delta[A] is not zero the above described procedure results in a difference array corresponding to a sorted array.

Let **level group** be a group of levels with same sail count. Observe that level groups can be identified in the difference array as a sequence of zeros bounded by nonzero values. In other words, to identify the level group containing the level H, we need to find first nonzero value in delta sequence before and after H.

When updating the lowest K levels (interval [H−K+1, H]), the model solution works as follows:

If the left part of the interval (H−K+1) is in the middle of a level group [A, B] then update the intervals [A, A+B−H+K−1] and [B+1, H]. In order to keep the array sorted, we have moved the new sails from the right part to the left part of the level group.

Otherwise, update the interval [H−K+1, H].

In order to obtain an efficient solution we need to be able to quickly find the level group corresponding to a particular level. There are a number of solutions based on this or similar ideas that score between 70 and 100 points, depending on the efficiency of the data structure used to identify the level group.

The model solution uses the interval tree data structure to answer the needed queries. Processing one mast will take time O(log(H)), where H is the height of the mast. Therefore, the total time complexity of this solution is O(N · log(max_height)).

MINERS

There are **two** coal mines, each employing a group of miners. Mining coal is hard work, so miners need food to keep at it. Every time a shipment of food arrives at their mine, the miners produce some amount of coal. There are three types of food shipments: meat shipments, fish shipments and bread shipments.

Miners like variety in their diet and they will be more productive if their food supply is kept varied. More precisely, every time a new shipment arrives to their mine, they will **consider the new shipment and the previous two shipments** (or fewer if there haven't been that many) and then:

- If all shipments were of the same type, they will produce one unit of coal.
- If there were two different types of food among the shipments, they will produce two units of coal.
- If there were three different types of food, they will produce three units of coal.

We know in advance the types of food shipments and the order in which they will be sent. It is possible to influence the amount of coal that is produced by determining which shipment should go to which mine. Shipments cannot be divided; each shipment must be sent to one mine or the other in its entirety.

The two mines don't necessarily have to receive the same number of shipments (in fact, it is permitted to send all shipments to one mine).

TASK

Your program will be given the types of food shipments, in the order in which they are to be sent. Write a program that finds the **largest total amount of coal** that can be produced (in both mines) by deciding which shipments should be sent to mine 1 and which shipments should be sent to mine 2.

INPUT

The first line of input contains an integer N ($1 \leq N \leq 100\,000$), the number of food shipments.

The second line contains a string consisting of N characters, the types of shipments in the order in which they are to be distributed. Each character will be one of the uppercase letters 'M' (for meat), 'F' (for fish) or 'B' (for bread).

OUTPUT

Output a single integer, the largest total amount of coal that can be produced.

GRADING

In test cases worth a total of 45 points, the number of shipments N will be at most 20.

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 10 submissions for this task to be evaluated (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLES

input	**input**
6 MBMFFB	16 MMBMBBBBMMMMMBMB
output	**output**
12	29

In the left sample, by distributing the shipments in this order: mine 1, mine 1, mine 2, mine 2, mine 1, mine 2, the shipments will result in 1, 2, 1, 2, 3 and 3 units of coal produced in that order, for a total of 12 units. There are other ways to achieve this largest amount.

SOLUTION

The problem is solved using dynamic programming. Let M(n, state1, state2) be the largest amount of coal that can be produced after n−1 food shipments have already been distributed; state1 describes which shipments have gone to mine 1 so far, and state2 describes which shipments have gone to mine 1 so far. Furthermore, let value(a, b) denote the amount of coal produced when food of type b arrives to a mine in state a. The following recursive formula holds:

M(n, state1, state2) = max {

M(n+1, newstate1, state2) + value(state1, type of shipment n),

M(n+1, state1, newstate2) + value(state2, type of shipment n)

}

One could write a simple recursion based on the above formula (calculate M(0, empty, empty)). Such an algorithm has $O(2^N)$ complexity and would score 45 points.

Note that, for calculating the value of M, it suffices to describe the state of a mine by the last two shipments only. This is because any shipment, other than the last two, cannot influence the amount of coal produced. The total number of different states is thus reduced to only 3*3+1=10 per mine (3 types of food in each of the last two shipments, and a special state describing an empty mine).

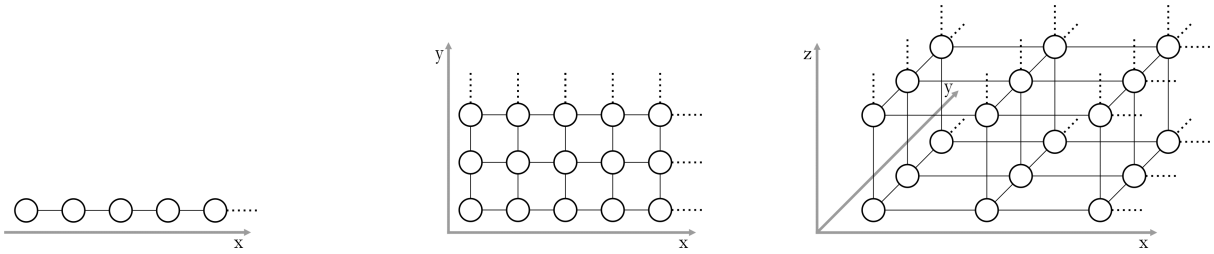
This allows us to drastically reduce the total number of configurations: N shipments × 10 states (for the first mine) × 10 states (for the second mine). For each of the configurations, we can calculate the value of the configuration once and store it for reuse.

However, with N as high as 100 000, using so much memory (100N integers) for storing the values of the configurations would break the memory limit and score between 70 and 85 points.

To get the full score, we note that, when calculating M(n, *, *), we only use M(n+1, *, *). If we calculate M in decreasing order of n, we need only 2·100 integers.

PAIRS

Mirko and Slavko are playing with toy animals. First, they choose one of three boards given in the figure below. Each board consists of cells (shown as circles in the figure) arranged into a one, two or three dimensional grid.



Board 1 Board 2 Board 3

Mirko then places N little **toy animals** into the cells.

The **distance** between two cells is the smallest number of moves that an animal would need in order to reach one cell from the other. In one move, the animal may step into one of the adjacent cells (connected by line segments in the figure).

Two animals can hear each other if the distance between their cells is **at most D**. Slavko's task is to calculate how many **pairs of animals** there are such that one animal can hear the other.

TASK

Write a program that, given the board type, the locations of all animals, and the number D, finds the desired number of pairs.

INPUT

The first line of input contains four integers in this order:

- The board type B ($1 \leq B \leq 3$);
- The number of animals N ($1 \leq N \leq 100\,000$);
- The largest distance D at which two animals can hear each other ($1 \leq D \leq 100\,000\,000$);
- The size of the board M (the largest coordinate allowed to appear in the input):
  - When B=1, M will be at most 75 000 000.
  - When B=2, M will be at most 75 000.
  - When B=3, M will be at most 75.

Each of the following N lines contains B integers separated by single spaces, the coordinates of one toy animal. Each coordinate will be between 1 and M (inclusive).

More than one animal may occupy the same cell.

OUTPUT

Output should consist of a single integer, the number of pairs of animals that can hear each other.

Note: use a 64-bit integer type to calculate and output the result (`long long` in C/C++, `int64` in Pascal).

GRADING

In test cases worth a total of 30 points, the number of animals N will be at most 1 000.

Furthermore, for each of the three board types, a solution that correctly solves all test cases of that type will be awarded at least 30 points.

EXAMPLES

input

```
1 6 5 100
25
50
50
10
20
23
```

output

```
4
```

input

```
2 5 4 10
5 2
7 2
8 4
6 5
4 4
```

output

```
8
```

input

```
3 8 10 20
10 10 10
10 10 20
10 20 10
10 20 20
20 10 10
20 10 20
20 20 10
20 20 20
```

output

```
12
```

Clarification for the leftmost example. Suppose the animals are numbered 1 through 6 in the order in which they are given. The four pairs are:

- 1-5 (distance 5)
- 1-6 (distance 2)
- 2-3 (distance 0)
- 5-6 (distance 3)

Clarification for the middle example. The eight pairs are:

- 1-2 (distance 2)
- 1-4 (distance 4)
- 1-5 (distance 3)
- 2-3 (distance 3)
- 2-4 (distance 4)
- 3-4 (distance 3)
- 3-5 (distance 4)
- 4-5 (distance 3)

SOLUTIONS

Let N be the number of animals, M the size of the board and D the largest distance.

Solution for 1D board

To solve the 1D board, we first sort the coordinates and then perform a sweep-line algorithm. We keep track of two pointers: **head** and **tail**. When the head is pointing to an element with coordinate x, the tail is pointing to the first element with coordinate greater than or equal to x−D.

For each position of head we add head−tail to the total result. As we advance head to the next element, we can easily adjust tail to point to the required element.

The time complexity of this algorithm is O(N log N) for sorting, and O(N) for sweeping.

Solution for 2D board

To solve the 2D version of the problem, we first consider the distance formula:

$$dist(P, Q) = | P.x - Q.x | + | P.y - Q.y |$$

The formula above will resolve to one of the following four formulas:

$$dist(P, Q) = P.x - Q.x + P.y - Q.y = (P.x + P.y) - (Q.x + Q.y)$$

$$dist(P, Q) = P.x - Q.x - P.y + Q.y = (P.x - P.y) - (Q.x - Q.y)$$

$$dist(P, Q) = -P.x + Q.x + P.y - Q.y = (Q.x - Q.y) - (P.x - P.y)$$

$$dist(P, Q) = -P.x + Q.x - P.y + Q.y = (Q.x + Q.y) - (P.x + P.y)$$

It is easy to see that the distance is always equal to the largest of these four values. As P.x + P.y and P.x − P.y represent the "diagonal coordinates" of point P we substitute:

$$P.d1 := P.x + P.y \text{ and } P.d2 := P.x - P.y.$$

Now, we can rewrite the distance formula in terms of d1 and d2:

$$dist(P, Q) = \max\{ P.d1 - Q.d1, P.d2 - Q.d2, Q.d2 - P.d2, Q.d1 - P.d1 \},$$

or shorter:

$$dist(P, Q) = \max\{ |P.d1 - Q.d1|, |P.d2 - Q.d2| \}$$

After substitution, we sort all the points by the first coordinate (d1) increasingly and perform a sweep-line algorithm similar to the one-dimensional case. Since each point P between head and tail satisfies the inequality $head.d1 - P.d1 \leq D$, we only need to find out for how many of them the inequality $|head.d2 - P.d2| \leq D$ is satisfied as well. To calculate that value, we keep all points (their d2 coordinates) between head and tail in either an **interval tree** or a **binary indexed tree** [1] data structure.

The time complexity of the algorithm implemented with a binary indexed tree data structure is O(N log N) for sorting, and O(N log M) for sweeping, where M is the upper bound on the coordinates.

Solution for 3D board

Inspired by our 2D solution we start with the distance formula once again and obtain:

$$dist(P, Q) = \max\{ |P.f1 - Q.f1|, |P.f2 - Q.f2|, |P.f3 - Q.f3|, |P.f4 - Q.f4| \}, \text{ where}$$

$$P.f1 := P.x + P.y + P.z$$

$$P.f2 := P.x + P.y - P.z$$

$$P.f3 := P.x - P.y + P.z$$

P.f4 := P.x − P.y − P.z

Again, we perform a sweep-line algorithm on the f1 coordinate while keeping all of the points between **head** and **tail** in a 3D binary indexed tree in order to count the number of points P satisfying inequalities $|head.f2 - P.f2| \leq D$, $|head.f3 - P.f3| \leq D$ and $|head.f4 - P.f4| \leq D$.

The time complexity of the algorithm is $O(N \log N)$ for sorting, and $O(N \log^3 M)$ for sweeping.

It is worth mentioning that we can use this solution to solve all types of boards. We just assign any constant value (1 for example) to each of the missing coordinates and implement a 3D binary indexed tree using either dynamic memory allocation, or using a one-dimensional array and manually mapping the 3-dimensional space to elements of the array.

References

[1] P. M. Fenwick, A new data structure for cumulative frequency tables, **Software - Practice and Experience** 24, 3 (1994), 327-336, 1994.

TRAINING

Mirko and Slavko are training hard for the annual tandem cycling marathon taking place in Croatia. They need to choose a route to train on.

There are N cities and M roads in their country. Every road connects two cities and can be traversed in both directions. Exactly N−1 of those roads are **paved**, while the rest of the roads are unpaved trails. Fortunately, the network of roads was designed so that each pair of cities is connected by a path consisting of paved roads. In other words, the N cities and the N−1 **paved roads form a tree structure**.

Additionally, each city is an endpoint for **at most 10 roads total**.

A training route starts in some city, follows some roads and ends in the same city it started in. Mirko and Slavko like to see new places, so they made a rule **never to go through the same city** nor **travel the same road twice**. The training route may start in any city and does not need to visit every city.

Riding in the back seat is easier, since the rider is shielded from the wind by the rider in the front. Because of this, Mirko and Slavko change seats in every city. To ensure that they get the same amount of training, they must choose a route with an **even number of roads**.

Mirko and Slavko's competitors decided to **block** some of the unpaved roads, making it **impossible** for them to find a training route satisfying the above requirements. For each unpaved road there is a **cost** (a **positive** integer) associated with blocking the road. It is impossible to block paved roads.

TASK

Write a program that, given the description of the network of cities and roads, finds the **smallest total cost** needed to block the roads so that **no training route exists** satisfying the above requirements.

INPUT

The first line of input contains two integers N and M ($2 \leq N \leq 1\,000$, $N-1 \leq M \leq 5\,000$), the number of cities and the total number of roads.

Each of the following M lines contains three integers A, B and C ($1 \leq A \leq N$, $1 \leq B \leq N$, $0 \leq C \leq 10\,000$), describing one road. The numbers A and B are different and they represent the cities directly connected by the road. If C=0, the road is paved; otherwise, the road is unpaved and C represents the cost of blocking it.

Each city is an endpoint for at most 10 roads. There will never be more than one road directly connecting a single pair of cities.

OUTPUT

Output should consist of a single integer, the smallest total cost as described in the problem statement.

GRADING

In test cases worth a total of 30 points, the paved roads will form a chain (that is, no city will be an endpoint for three or more paved roads).

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 10 submissions for this task to be evaluated (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLES

input

```
5 8
2 1 0
3 2 0
4 3 0
5 4 0
1 3 2
3 5 2
2 4 5
2 5 1
```

output

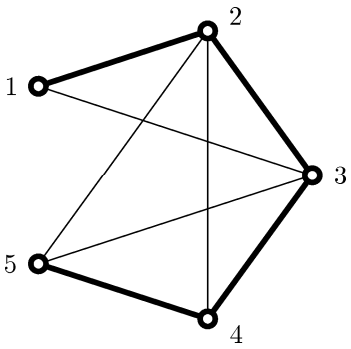
```
5
```

input

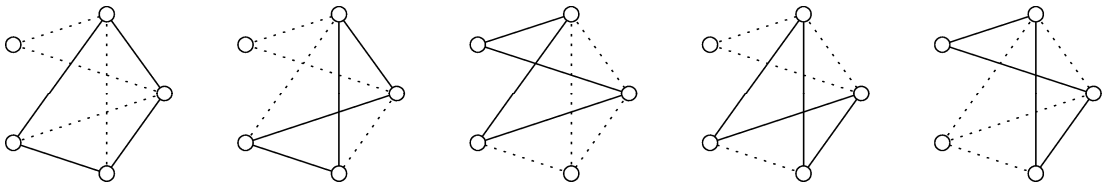
```
9 14
1 2 0
1 3 0
2 3 14
2 6 15
3 4 0
3 5 0
3 6 12
3 7 13
4 6 10
5 6 0
5 7 0
5 8 0
6 9 11
8 9 0
```

output

```
48
```



The layout of the roads and cities in the first example. Paved roads are shown in bold.



There are five possible routes for Mirko and Slavko. If the roads 1-3, 3-5 and 2-5 are blocked, then Mirko and Slavko cannot use any of the five routes. The cost of blocking these three roads is 5.

It is also possible to block just two roads, 2-4 and 2-5, but this would result in a higher cost of 6.

SOLUTION

Detecting an odd cycle in a graph is a well-known problem. A graph does not contain an odd cycle if and only if it is bipartite. On the other hand, the problem of detecting an even cycle in a graph is not widely known.

We are given a graph consisting of N vertices and M edges. Exactly N−1 edges are marked as **tree edges** and they form a tree. An edge that is not a tree edge will be called a **non-tree edge**. Every non-tree edge e has a weight w(e) associated with it.

The task asks us to find a minimum-weighted set of non-tree edges whose removal results in a graph that does not contain a cycle of even length. We will call such a cycle an **even cycle**. Reasoning backwards, starting from a graph containing tree edges only, we have to find a maximum-weighted set of non-tree edges that can be added to the graph without forming any even cycles.

In order to describe the model solution, we first need to make a few observations about the structure of the graph we are working with.

Even and odd edges

Consider a non-tree edge e={A, B}. We define the **tree path of the edge e** to be the unique path from A to B consisting of tree edges only. If the length of the tree path is even, we say that e is an **even edge**; otherwise we say that e is an **odd edge.** We will use TP(e) to denote the tree path of an edge e.

Obviously, any odd edge present in the graph together with its tree path forms an even cycle. Therefore, we can never include an odd edge in our graph and we can completely ignore them.

Relation between two even edges

Individual even edges may exist in the graph. However, if we include several even edges, an even cycle might be formed. More precisely, if e1 and e2 are even edges such that TP(e1) and TP(e2) share a common tree edge, then adding both e1 and e2 to the graph necessarily creates an even cycle.

In order to sketch the proof of this claim, consider the two odd cycles created by e1 and e2 together with their respective tree paths. If we remove all common tree edges from those cycles we get two paths P1 and P2. The parity of P1 is equal to the parity of the P2 since we removed the same number of edges from the two initial odd cycles. As P1 and P2 also have the same endpoints, we can merge them into one big even cycle.

Tree edges contained in odd cycles

As a direct consequence of the previous claim, we can conclude that **every tree edge** may be contained in **at most one odd cycle**.

Conversely, if we add only even edges to the tree in such a way that every tree edge is contained in at most one odd cycle, then we couldn't have formed any even cycles. We briefly sketch the proof of this claim here. If an even cycle existed, it would have to contain one or more non-tree edges. Informally, if it contains exactly one non-tree edge we have a contradiction with the assumption that only even edges are added; if it contains two or more non-tree edges then we will arrive at a contradiction with the second assumption.

Model solution

Now, we can use our observations to develop a dynamic programming solution for the problem. A **state** is a subtree of the given tree. For each state we calculate the weight of the maximum-weighted set of even edges that can be added to the subtree while maintaining the property that each tree edge is contained in at most one odd cycle. The solution for the task is the weight associated with the state representing the initial tree.

To obtain a recursive relation, we consider all even edges with tree paths passing through the root of the tree. We can choose to do one of the following:

(1) We do not add any even edge whose tree path passes through the root of the tree. In this case, we can delete the root and proceed to calculate the optimal solution for each of the subtrees obtained after deleting the root node.

(2) We choose an even edge e whose tree path passes through the root of the tree and add it to the tree. Next, we delete all tree edges along TP(e) (since, now, they are contained in one odd cycle), and, finally, we proceed to calculate the optimal solution for each of the subtrees obtained after deleting the tree path. Add w(e) to the total sum.

We will use the tree in figure 1 as an example. Figure 2 shows case (1) in the recursive relation (we choose not to include an edge whose tree path passes through the root). Figure 3 shows case (2) in the recursive relation, when we include the even edge e={7, 9} in the graph.

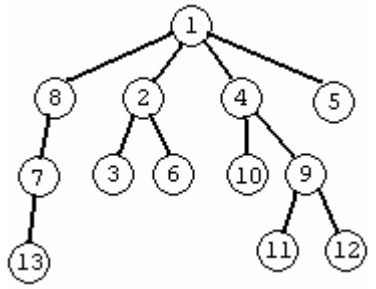


Figure 1

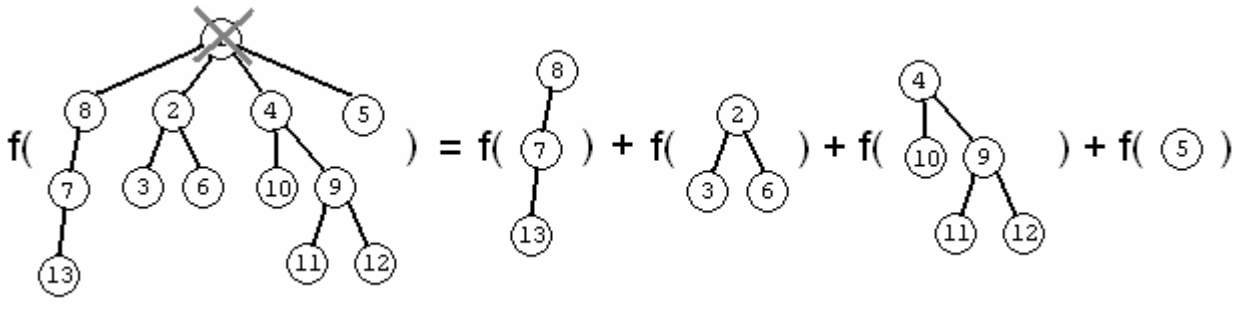


Figure 2

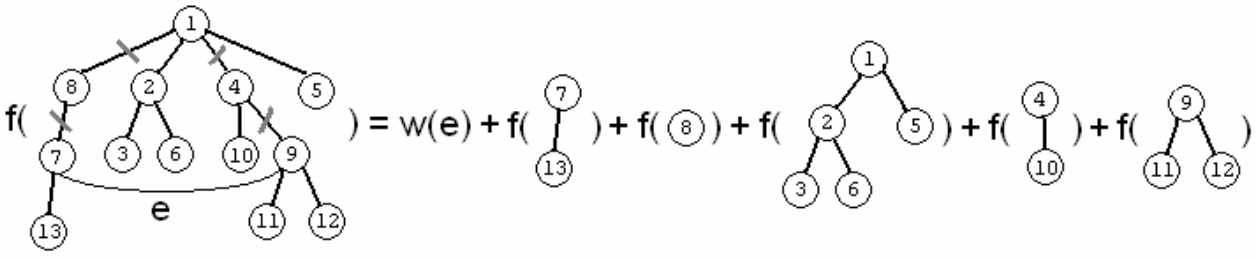


Figure 3

Because of the way the trees are decomposed, all subtrees that appear as subproblems can be represented with an integer and a bit mask. The integer represents the index of the subtree's root node, while the bit mask represents which of the root node's children are removed from the subtree.

The total number of possible states is, therefore, bounded by $N \cdot 2^K$ where K is the maximum degree of a node.

Depending on the implementation details, the time complexity of the algorithm can vary. The official implementation has time complexity $O(M \log M + MN + M \cdot 2^K)$.