2025 Team Formation Test

Task Overview

ID	Name	Time Limit	Memory Limit	Subtasks
T251	Moving Marvellous Marbles	2.000 s	1024 MB	3 + 5 + 10 + 19 + 8 + 15 + 21 + 18 + 1
T252	Sage's Shopping Spree	1.000 s	1024 MB	9 + 14 + 15 + 19 + 10 + 9 + 8 + 16
T253	Peaceful Pirate Pairs	1.000 s	1024 MB	15 + 9 + 20 + 11 + 21 + 24
T254	Robo's Recursive Realm	1.500 s	1024 MB	8 + 15 + 37 + 14 + 10 + 16

Notice:

- All tasks are divided into subtasks. You need to pass all test cases in a subtask to get points.
- There is an attachment package that you can download from the contest system, containing sample graders, sample implementations, example test cases, and compile and run scripts.
- When testing your programs with the sample grader, your input should match the format and constraints from the task statement, otherwise, unspecified behaviors may occur.
- In sample grader inputs, every two consecutive tokens on a line are separated by a single space, unless another format is explicitly specified.
- The task statements specify signatures using generic type names `void`, `bool`, `int`, `int64`, `int[]` (array) and `int[][]` (array of array).
- In C++, the graders use appropriate data types or implementations, as listed below:

void	bool	string	int	int64
`void`	`bool`	`string`	`int`	`long long`

int[]	int[][]	length of array a
`std::vector<int>`	`std::vector<std::vector<int>>`	`a.size()`

T251 - MOVING MARVELLOUS MARBLES

Time Limit: 2.000 s / Memory Limit: 1024 MB

Once upon a time, there was a place called Marblea. Marblea consists of N islands, where the islands are labelled island 0 to island $N-1$. The N islands in Marblea are connected by $N-1$ bidirectional bridges, where the i-th ($0 \leq i < N-1$) bridge connects island U_i and V_i ($0 \leq U_i, V_i < N$). There is exactly one way to go from an island to another island by bridges. In other words, it resembles a tree structure.

You have a budget to build at most K new bidirectional tunnels between any pair of islands. Since the budget is very limited, K is either 0 or 1, which means you are either not allowed to build any tunnels or may build at most one tunnel.

Initially, there are A_i marbles on island i ($0 \leq i < N$). Your target is to move marbles around so that island i ($0 \leq i < N$) ends up with B_i marbles. It is guaranteed that the total number of initial marbles equals the total number of target marbles. You are the only one who is moving the marbles. Every second, you may roll exactly one marble across any single bridge or tunnel in either direction. Note that when you are not carrying a marble, you can teleport anywhere instantly.

Your goal is to find the minimum number of seconds needed to reach the target distribution of marbles in Marblea, including optimally picking the endpoints of the new tunnel if applicable.

IMPLEMENTATION DETAILS

You should implement the following procedure:

```
int64 redistribute(int N, int K, int[] A, int[] B, int[] U, int[] V)
```

- This procedure is called exactly once per run.
- N: The number of islands in Marblea.
- K: The maximum number of tunnels that you can build.
- A: An array of size N, representing the initial number of marbles on each island.
- B: An array of size N, representing the target number of marbles on each island.
- U, V: Arrays of size $N-1$, representing the bridges of Marblea. The i-th bridge ($0 \leq i < N-1$) connects island U_i and island V_i.
- The procedure should return an integer, the minimum number of seconds needed to redistribute the marbles to achieve the target distribution.

SAMPLE GRADER

The sample grader reads input in the following format:

- The first line contains two integers, N and K.
- The second line contains N integers, $A_0, A_1, \ldots, A_{N-1}$.
- The third line contains N integers, $B_0, B_1, \ldots, B_{N-1}$.
- The $(i+1)$-th of the next $N-1$ lines contains two integers, U_i and V_i.

The sample grader outputs a single integer, the returned integer of the procedure.

EXAMPLE

Consider the following call:

```
redistribute(5, 0, [1, 0, 0, 0, 1], [0, 1, 1, 0, 0], [0, 1, 2, 3], [1, 2, 3, 4])
```

One possible method is to move a marble from island 4 to island 3 via bridge 3, move a marble from island 3 to island 2 via bridge 2, and move a marble from 0 to 1 via bridge 0 with total time 3. It can be proven that this is the minimum time achievable. Hence, your procedure should return 3.

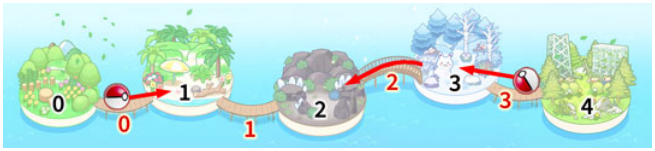


Figure 1: Illustration of Sample 1. Image from Pokemon Sleep.

Consider the following call:

```
redistribute(5, 1, [1, 0, 0, 0, 1], [0, 1, 1, 0, 0], [0, 1, 2, 3], [1, 2, 3, 4])
```

One possible method is to build a tunnel between island 2 and island 4, then move a marble from 4 to 2 via the newly added tunnel and move a marble from 0 to 1 via bridge 0 with total time 2. It can be proven that this is the minimum time achievable. Hence, your procedure should return 2.

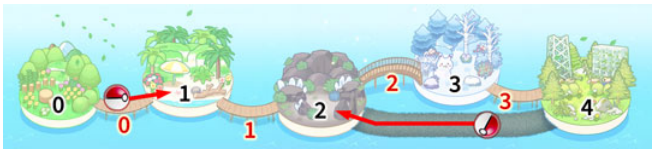


Figure 2: Illustration of Sample 2. Image from Pokemon Sleep.

Consider the following call:

```
redistribute(6, 1, [4, 6, 2, 1, 4, 3], [1, 3, 7, 1, 4, 4], [4, 3, 5, 1, 2], [1, 0, 2, 0, 1])
```

One possible method is to build a tunnel between island 0 and island 2, then move the marbles with total time 7. It can be proven that this is the minimum time achievable. Hence, your procedure should return 7.

SAMPLE TESTS

	Input	Output
1	5 0 1 0 0 0 1 0 1 1 0 0 0 1 1 2 2 3 3 4	3
2	5 1 1 0 0 0 1 0 1 1 0 0 0 1 1 2 2 3 3 4	2

	Input	Output
3	6 1 4 6 2 1 4 3 1 3 7 1 4 4 4 1 3 0 5 2 1 0 2 1	7

SUBTASKS

For all cases:
$2 \leq N \leq 50000$
$K = 0$ or $K = 1$
$0 \leq A_i, B_i \leq 10^9$ for all $0 \leq i < N$
$0 \leq U_i, V_i < N$ for all $0 \leq i < N - 1$
$A_0 + A_1 + \ldots + A_{N-1} = B_0 + B_1 + \ldots + B_{N-1}$
There is exactly one way to go from an island to another island by bridges

	Points	Constraints
1	3	$K = 0$ $U_i = i$ and $V_i = i + 1$ for all $0 \leq i < N - 1$
2	5	$K = 0$
3	10	$K = 1$ $N \leq 30$ $A_0 + A_1 + \cdots + A_{N-1} \leq 30$
4	19	$K = 1$ $N \leq 300$
5	8	$K = 1$ $N \leq 1500$ $U_i = i$ and $V_i = i + 1$ for all $0 \leq i < N - 1$
6	15	$K = 1$ $N \leq 1500$
7	21	$K = 1$ $N \leq 5000$
8	18	$K = 1$ $N \leq 20000$ $U_i = \lfloor \frac{i}{2} \rfloor$ and $V_i = i + 1$ for all $0 \leq i < N - 1$
9	1	$K = 1$

T252 - SAGE'S SHOPPING SPREE

Time Limit: 1.000 s / Memory Limit: 1024 MB

It's been a long, exhausting term. Sage has just returned from studying overseas, and now she has to complete her first task at home: grocery shopping at the local supermarket with her mother. The supermarket is modelled as a tree, a set of N nodes and $N-1$ edges. Nodes are numbered from 0 to $N-1$ and edges are numbered from 0 to $N-2$. Each edge connects two distinct nodes of the tree and has an integer parameter associated with it. Specifically, edge j $(0 \leq j \leq N-2)$ connects nodes $U[j]$ and $V[j]$ with a parameter $W[j]$.

Each node i (for $0 \leq i \leq N-1$) stores a single item of size $A[i]$, where all $A[i]$ are **distinct** integers from 1 to N (inclusive). Sage's goal is to help her mother collect **all** N distinct items from the supermarket. To do so, she has a shopping basket with unlimited capacity. However, when she pushes the basket across edge j (connecting nodes $U[j]$ and $V[j]$), the basket must satisfy the following condition: the size of **each item** currently in the basket must be **at most** $W[j]$.

Once an item is placed in the basket, it remains there until she decides to empty it. She may not remove items mid-trip. A **trip** is defined as follows:

- She can choose any node to begin a trip. At the start of each trip, her shopping basket is empty.
- She may traverse the tree visiting any number of nodes (possibly visiting a node more than once), collecting items along the way.
- During a trip, she must pick up at least one item overall. However, when visiting a specific node, she can choose not to pick up the item there (even if it hasn't been collected yet).
- She can choose any node to end the trip, at which point she empties the basket.
- Merely passing through a node designated as an ending node does not conclude the trip unless she empties the basket there.

Different trips may start and end at different nodes. Since time is running short, Sage's task is to decide whether it is possible to collect all N distinct items using **at most** K **trips**. If it is possible, she should also determine the **minimum number of trips** needed. If it is impossible, she should report that as well. Help Sage complete the task!

IMPLEMENTATION DETAILS

You should implement the following procedure:

```
int minimum_trips(int N, int K, int[] A, int[] U, int[] V, int[] W)
```

- This procedure is called exactly once per run.
- N: The number of nodes in the tree.
- K: The maximum allowed number of trips.
- A: An array of size N where $A[i]$ is the size of the item at node i (for $0 \leq i \leq N-1$).
- U, V, W: Arrays of size $N-1$ representing the edges of the tree. There is an edge between nodes $U[j]$ and $V[j]$ with parameter $W[j]$ (for $0 \leq j \leq N-2$).
- The procedure should return the minimum number of trips needed to collect all items if it is possible to do so using at most K trips. If it is impossible to collect all items within K trips, return -1.

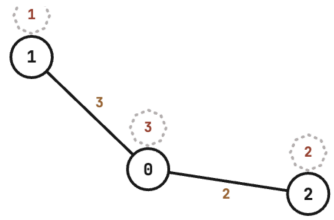
EXAMPLE

Consider the following call:

```
minimum_trips(3, 1, [3, 1, 2], [0, 0], [1, 2], [3, 2])
```

Here, the supermarket is represented as a tree with $N = 3$ nodes and at most $K = 1$ trip is allowed. Node 0 stores an item of size 3, node 1 stores an item of size 1, and node 2 stores an item of size 2. The tree edges are:

- Edge between node 0 and node 1 with parameter 3.
- Edge between node 0 and node 2 with parameter 2.

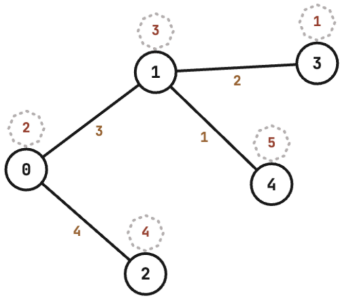


One valid strategy is:

- Start at node 2, picking up an item of size 2.
- Then travel to node 0. When moving from node 2 to node 0 the basket has a single item of size 2, which does not exceed the edge parameter $W[j] = 2$.
- Pick up an item of size 3 at node 0.
- Then travel to node 1. When moving from node 0 to node 1 the basket has two items of sizes 2 and 3 respectively, both of which do not exceed the edge parameter $W[j] = 3$.
- Pick up an item of size 1 at node 1.
- End the trip at node 1, emptying the basket.

Thus, all items can be collected in a single trip, which does not exceed the allowed maximum of $K = 1$. The procedure should return 1.
Consider the following call:

```
minimum_trips(5, 2, [2, 3, 4, 1, 5], [0, 0, 1, 1], [1, 2, 3, 4], [3, 4, 2, 1])
```

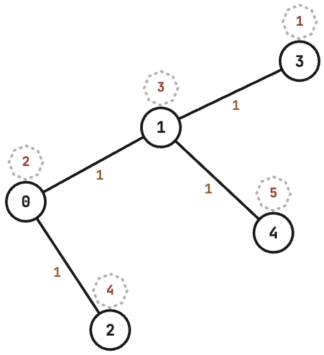


Here, a possible way to use 2 trips to collect all the items is as follows:

- Start at node 1, collect items on node 1, then node 0, then node 2 on the first trip.
- Start at node 3, collect items on node 3, then node 4 (passing through node 1) on the second trip.

It can be proven that there is no way to collect all the items in 1 trip. Hence, the procedure should return 2.
Consider the following call:

```
minimum_trips(5, 2, [2, 3, 4, 1, 5], [0, 0, 1, 1], [1, 2, 3, 4], [1, 1, 1, 1])
```



It can be proven that there is no way to use at most $K = 2$ trips to collect all the items. Hence, the procedure should return -1
.

SAMPLE GRADER

The sample grader reads input in the following format:

- The first line contains two integers N and K.
- The second line contains N integers $A[0]$, $A[1]$, ..., $A[N-1]$.
- The following $N-1$ lines each contain three integers: $U[j]$, $V[j]$ and $W[j]$.

The sample grader outputs a single integer: the value returned by the procedure.

SAMPLE TESTS

	Input	Output
1	3 1 3 1 2 0 1 3 0 2 2	1
2	5 2 2 3 4 1 5 0 1 3 0 2 4 1 3 2 1 4 1	2
3	5 2 2 3 4 1 5 0 1 1 0 2 1 1 3 1 1 4 1	-1

SUBTASKS

For all cases:
$1 \le N \le 10^5$
$1 \le K \le N$
$1 \le A[i] \le N$ for $0 \le i \le N-1$
All $A[i]$ are distinct
$0 \le U[j], V[j] \le N-1$ for $0 \le j \le N-2$
$1 \le W[j] \le N$ for $0 \le j \le N-2$
The edges $(U[j], V[j])$ form a tree

	Points	Constraints
1	9	$K = 1$ $N = 3$ $U[j] = 0, V[j] = j+1$ for $0 \le j \le N-2$
2	14	$K = 1$ $1 \le N \le 200$ $U[j] = 0, V[j] = j+1$ for $0 \le j \le N-2$
3	15	$K = 1$ $1 \le N \le 200$
4	19	$K = 1$
5	10	$K = 2$ $3 \le N \le 200$ There exists at least two edges such that $W[j] = 1$
6	9	$1 \le N \le 200$
7	8	$1 \le N \le 2000$
8	16	No additional constraints

T253 - PEACEFUL PIRATE PAIRS

Time Limit: 1.000 s / Memory Limit: 1024 MB

A square stretch of ocean is modelled as an $N \times N$ grid, where the rows are numbered $0, 1, \ldots, N-1$ from north to south, and the columns are numbered from west to east as $0, 1, \ldots, N-1$. We denote cell (X, Y) as the cell on the X-th row and the Y-th column $(0 \leq X, Y < N)$.

Each cell is either **open sea** or occupied by an **impassable island**. There are only K cells which are initially islands, the i-th of which $(0 \leq i < K)$ is at cell $(A[i], B[i])$.

There are M pirate ships located on the open sea (i.e. non-island cells), each occupying a single cell. The j-th of them $(0 \leq j < M)$ occupies cell $(X[j], Y[j])$. It is guaranteed that no two ships share a cell. Two pirate ships are said to **be in conflict** if and only if they lie on the same row and no island lies strictly between them on that row. We say the ocean is **peaceful** if no pair of pirate ships is in conflict.

The following figures are examples of peaceful oceans with $N = 4$.

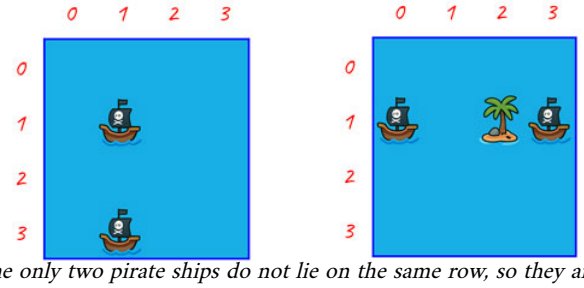


Figure 1 (Left): The only two pirate ships do not lie on the same row, so they are NOT in conflict.
Figure 2 (Right): An island is at $(1, 2)$, which is strictly between the only two pirate ships at $(1, 0)$ and $(1, 3)$, so they are also NOT in conflict.

On the other hand, the two pirate ships in both figures below lie on the same row, and no island lies strictly between them on that row. Hence, the ocean is not peaceful in either case.

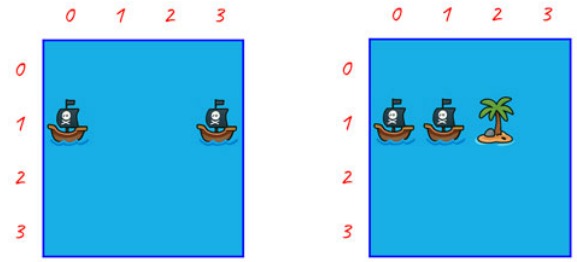


Figure 3 (Left): The two pirate ships lie on the same row with no islands and no ships in between, so they are in conflict.
Figure 4 (Right): Note that two ships are considered in conflict when they are immediately next to each other in the same row.

Initially, the ocean is NOT peaceful. To maintain the peace of the ocean, **barriers** may be applied on columns **without pirate ships** to stop pirate ships from being in conflict. Formally, for some column c, where $0 \leq c < N$, if column c contains no pirate ship, it is possible to apply a barrier on column c, which turns **every** cell of column c into an **island** cell.

At the beginning, no barriers are applied. There are Q updates, each doing one of the following:

- **Type 1**: Apply a new barrier on column c, where $0 \leq c < N$. It is guaranteed that column c has no pirate ship, and there is currently no barrier on this column.
- **Type 2**: Remove the existing barrier on column c, where $0 \leq c < N$. It is guaranteed that before this update, there is currently a barrier on column c. Note that, removing a barrier on a column c does not remove the original island cells located on column c.

You must answer: After each update, is the ocean currently peaceful or not?

IMPLEMENTATION DETAILS

You should implement the following procedure:

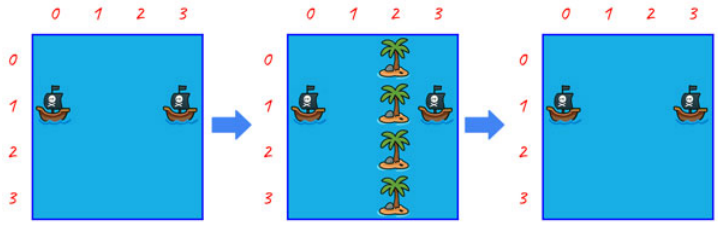
`bool[] solve(int N, int K, int[] A, int[] B, int M, int[] X, int[] Y, int Q, int[] T, int[] C)`

- This procedure is called exactly once per run.
- N: The number of rows and columns of the grid.
- K: The number of island cells.
- A, B: Two arrays of length K, representing the locations of the island cells. For each i $(0 \leq i < K)$, an island is on cell $(A[i], B[i])$. All locations of island cells are distinct.
- M: The number of pirate ships.
- X, Y: Two arrays of length M, representing the locations of the pirate ships. For each i $(0 \leq i < M)$, a pirate ship is placed on cell $(X[i], Y[i])$. No pirate ship is placed on a island cell, and all locations of pirate ships are distinct.
- Q: The number of updates.
- T, C: Two arrays of length Q, representing the updates. For each i $(0 \leq i < Q)$, the i-th update is of type $T[i]$ and parameter $C[i]$.
- This procedure should return an array S of length Q. For each i $(0 \leq i < Q)$, $S[i]$ should be `true` if the ocean is peaceful after the i-th update. Otherwise, $S[i]$ should be `false`.

EXAMPLE

Consider the following call:

```
solve(
    4,
    0, [], [],
    2, [1, 1], [3, 0],
    2, [1, 2], [2, 2]
)
```



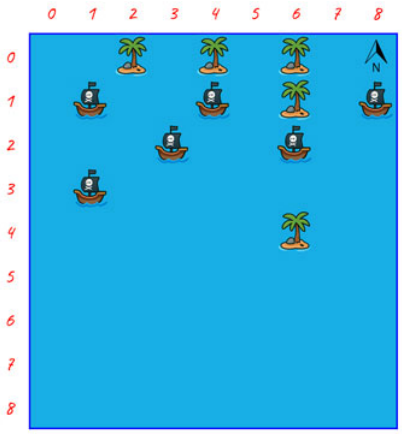
There are initially $K = 0$ islands. There are $M = 2$ pirate ships located at $(1, 3), (1, 0)$. Initially, pirate ships 0 and 1 are in conflict. Then two updates are performed, applying a barrier on column 2, and removing it afterwards.

- After the 1-st update, pirate ships 0 and 1 are no longer in conflict. Hence, the ocean is peaceful after this update.
- After the 2-nd update, pirate ships 0 and 1 are in conflict again. Hence, the ocean is NOT peaceful after this update.

Hence, the procedure should return the array `[true, false]`.

Consider the following call:

```
solve(
    9,
    5, [0, 0, 0, 1, 4], [2, 4, 6, 6, 6],
    6, [1, 1, 1, 2, 2, 3], [1, 4, 8, 3, 6, 1],
    6, [1, 1, 1, 2, 1, 2], [0, 2, 5, 0, 7, 2]
)
```

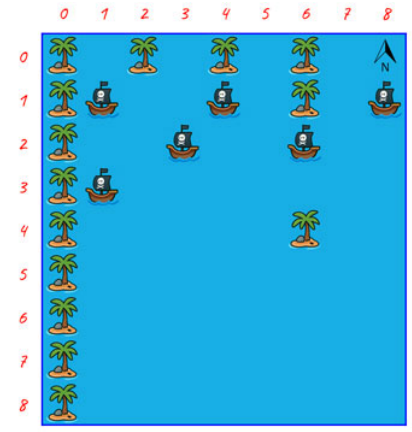
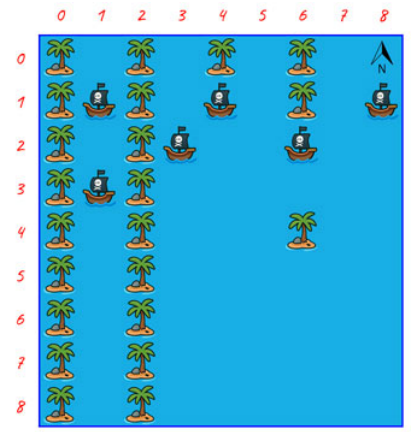
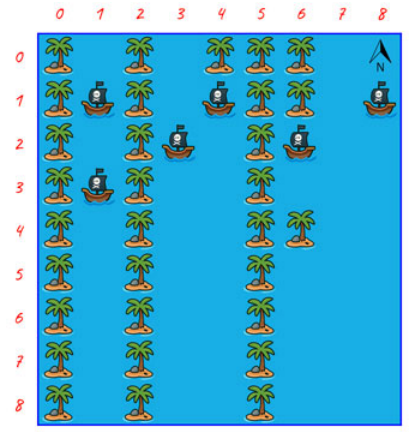
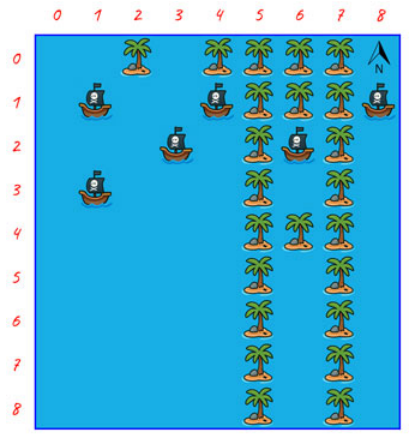


There are initially $K = 5$ islands located at $(0, 2), (0, 4), (0, 6), (1, 6), (4, 6)$. There are $M = 6$ pirate ships located at $(1, 1), (1, 4), (1, 8), (2, 3), (2, 6), (3, 1)$. Hence, the initial layout of the ocean can be illustrated in the following diagram:

You can see that the following pairs of pirate ships are currently in conflict:

1. Pirate ships 0 and 1.
2. Pirate ships 3 and 4.

The following is what happens after each update:

1. Apply a barrier on column 0.	2. Apply a barrier on column 2.	3. Apply a barrier on column 5.
		
4. Remove the barrier on column 0.	**5. Apply a barrier on column 7.**	**6. Remove the barrier on column 2.**
		

- After the 1-st and 2-nd updates, pirate ships 3 and 4 are still in conflict. Hence, the ocean is not peaceful after these updates.
- After the 3-rd, 4-th, and 5-th updates, no pair of pirate ships are in conflict. Hence, the ocean is peaceful after these updates.
- After the 6-th update, pirate ships 0 and 1 become in conflict. Hence, the ocean is not peaceful after this update. Note that although we remove the barrier on column 2, the cell $(0, 2)$ remains an island cell since it is one of the K existing island cells.

Hence, the procedure should return the array `[false, false, true, true, true, false]`.

Consider the following call:

```
solve(
    5,
    1, [0], [1],
    3, [0, 0, 0], [0, 2, 4],
    3, [1, 1, 2], [1, 3, 1]
)
```

There is initially $K = 1$ island located at $(0, 1)$. There are $M = 3$ pirate ships located at $(0, 0), (0, 2), (0, 4)$. Pirate ships 1 and 2 are in conflict.

- After the 1-st update, pirate ships 1 and 2 are still in conflict. Hence, the ocean is not peaceful after this update.
- After the 2-nd update, no pair of pirate ships are in conflict. Hence, the ocean is peaceful after this update.
- After the 3-rd update, no pair of pirate ships are in conflict. Hence, the ocean is peaceful after this update. Note that removing the barrier on column 1 does not affect the cell $(0, 1)$, which remains an island because it was there initially.

Hence, the procedure should return the array `[false, true, true]`.

SAMPLE GRADER

The sample grader reads the input in the following format:

- The first line contains a single integer N.
- The second line contains a single integer K.
- The next K lines describe the locations of the island cells. The $(i + 1)$-th line among them $(0 \leq i < K)$ contains two integers, $A[i]$ and $B[i]$.
- The next line contains a single integer M.
- The next M lines describe the locations of the pirate ships. The $(i + 1)$-th line among them $(0 \leq i < M)$ contains two integers, $X[i]$ and $Y[i]$.
- The next line contains a single integer Q.
- The next Q lines describe the updates. The $(i + 1)$-th line among them $(0 \leq i < Q)$ contains two integers, $T[i]$ and $C[i]$.

The output of the sample grader is in the following format:

- It outputs `Yes` for `true` and `No` for `false` in the contents of the array returned, on separate lines.

SAMPLE TESTS

	Input	Output
1	4 0 2 1 3 1 0 2 1 2 2 2	Yes No
2	9 5 0 2 0 4 0 6 1 6 4 6 6 1 1 1 4 1 8 2 3 2 6 3 1 6 1 0 1 2 1 5 2 0 1 7 2 2	No No Yes Yes Yes No
3	5 1 0 1 3 0 0 0 2 0 4 3 1 1 1 3 2 1	No Yes Yes

SUBTASKS

For all cases:
$2 \leq N \leq 2 \times 10^5$
$1 \leq Q \leq 2 \times 10^5$
$0 \leq K \leq \min(N^2, 2 \times 10^5)$
$2 \leq M \leq \min(N^2, 2 \times 10^5)$
$0 \leq A[i], B[i] < N$ for $0 \leq i < K$
$0 \leq X[i], Y[i] < N$ for $0 \leq i < M$
$1 \leq T[i] \leq 2$ for $0 \leq i < Q$
$0 \leq C[i] < N$ for $0 \leq i < Q$
Initially, the ocean is not peaceful

	Points	Constraints
1	15	$N, M \leq 30$ $Q \leq 10$ $T[i] = 1$ for $0 \leq i < Q$
2	9	$N \leq 2000$ $Q \leq 10$ $T[i] = 1$ for $0 \leq i < Q$
3	20	$Q \leq 10$ $T[i] = 1$ for $0 \leq i < Q$
4	11	$T[i] = 1$ for $0 \leq i < Q$
5	21	$X[i] = 0$ for $0 \leq i < M$
6	24	No additional constraints

T254 - ROBO'S RECURSIVE REALM

Time Limit: 1.500 s / Memory Limit: 1024 MB

Robo the robot has entered a mysterious realm! The realm can be modelled as a grid with N rows and N columns, where N is an integer known to Robo. The cell on the r-th row and the c-th column is denoted as (r, c). Robo is initially located at $(1, 1)$, facing down.

The grid is surrounded by walls, except for a single exit which is located at one of the edges (top, bottom, left or right). It is guaranteed that the exit is **not next to a corner** (i.e. $(1, 1)$, $(1, N)$, $(N, 1)$ or (N, N)). Therefore, we can denote the location of the exit using the cell closest to it. Besides, the grid contains a single box initially located at cell (r, c). The box is **not located at the sides**, i.e. $2 \leq r, c \leq N - 1$. Unfortunately, Robo does not know the location of the exit and the box.

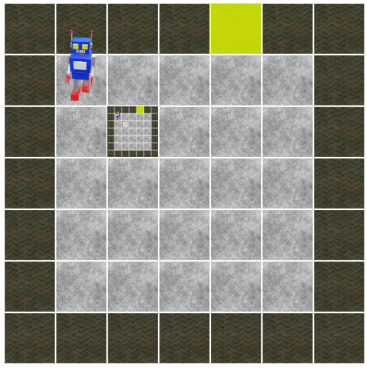


Figure 1: Initial configuration of the realm with $N = 5$. The box is located at $(2, 2)$ and the exit is located at $(1, 4)$. All dark cells are walls.

Robo's goal is to push the box to the exit (without accidentally exiting the realm itself). To achieve this, you can program Robo using a sequence of **instructions**. Each instruction has a **type** and is possibly marked with a unique **label**. The available instruction types are as follows:

Type	Name	Description
`F`	Forward	Robo moves one cell forward towards the direction that Robo is currently facing. The box is pushed forward if it blocks Robo. If Robo or the box is blocked by the wall, Robo's location remains unchanged. Note that it's possible for Robo or the box to move onto the exit cell, but the 3 adjacent edges of the exit would be blocked by walls.
`L`	Turn Left	Robo turns 90 degrees to the left (anti-clockwise).
`R`	Turn Right	Robo turns 90 degrees to the right (clockwise).
`BB label`	Branch if Box	If the box is immediately in front of Robo currently, jump to the instruction marked by `label`. Otherwise, do nothing.
`BW label`	Branch if Wall	If the wall is immediately in front of Robo currently, jump to the instruction marked by `label`. Otherwise, do nothing. Note that the exit is not considered a wall.
`J label`	Jump	Jump to the instruction marked by `label`.
`DIE`	Die	Terminates the program immediately, reporting that it is **impossible to push the box to the exit** under the **initial configuration**. This does **not** depend on the current state of the realm.

Robo executes the sequence of instructions sequentially, except if it encounters a branch or jump instruction. For example, if the program only consists of three instructions `F`, `L` and `F` respectively, Robo will move as shown in the following figure:

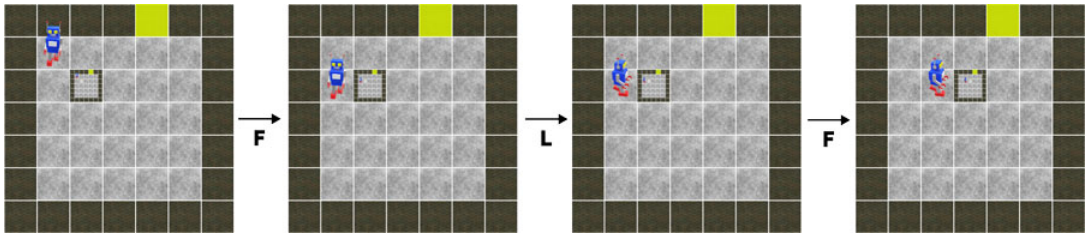


Figure 2: Result of executing the program.

Consider inserting a Branch if Box instruction and a label `[end]` to the program, so that the program now consists of the instructions `F`, `L`, `BB [end]`, `F` and `[end]`. Then, the `BB` instruction would divert the control flow and the execution jumps to `[end]` directly, skipping the `F` instruction.

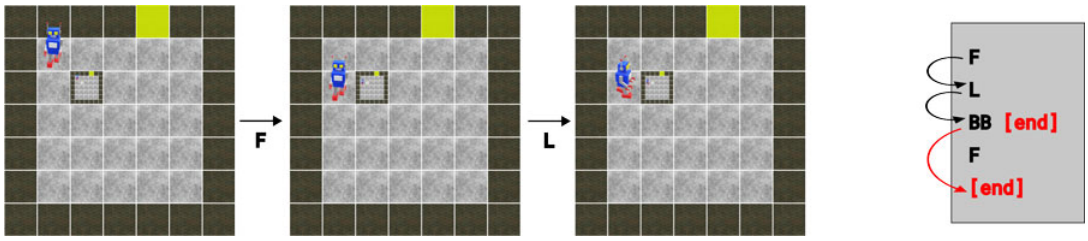


Figure 3: Result of executing the modified program with the branch instruction.

If we instead use a Branch if Wall instruction instead of the Branch if Box instruction, then the branch will not be triggered (as Robo is not blocked by the wall) and the fourth instruction (`F`) will be executed as normal.

However, there is a caveat! As you might have observed, the box is a replica of the entire game state. Most of the time, the box is a *normal box*; it can be normally pushed from the four directions. However, sometimes, the box is a *recurbox* instead, which allows Robo to "zoom into" the game state recursively!

The recurbox comes with a *blackhole*. Similar to the exit, the location of the blackhole is described by the cell closest to it. This cell is located near one of the edges (top, bottom, left or right), except any of the four corners. **The blackhole and the exit possibly share the same location**. The edge containing the blackhole is associated with the corresponding edge of the recurbox. Once the recurbox is pushed along that edge in a forward move, Robo goes through the blackhole and arrives at the cell closest to the blackhole. The location of the box remains unchanged.

For example, consider the realm shown in the following figure, where the blackhole is located at the left edge $(3, 1)$. If Robo attempts to push the box to the right (along its left edge), Robo enters the box and its location changes to $(3, 1)$ instead.

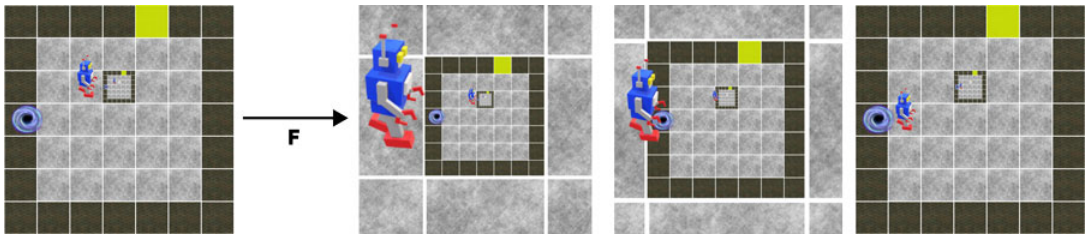


Figure 4: Teleportation effect when pushing a recurbox along the edge of the blackhole.

Note that the teleportation only works **one-way** - the teleportation only happens when Robo attempts to push the box. Nothing happens when Robo attempts to walk into the blackhole on the wall. Moreover, the Branch if Wall instruction treats the wall with the blackhole like any other wall, i.e. jump to label is taken if the wall is not the exit.

You are given the type of the box (either normal box or recurbox) as well as N, the number of rows and columns in the grid. Given this, construct a sequence of instructions such that Robo pushes the box to the exit or correctly reports that it is impossible to push the box to the exit, regardless of the initial location of the box and/or the blackhole (if any). In addition, Robo must terminate after executing at most 2×10^6 instructions (excluding labels).

IMPLEMENTATION DETAILS

You should implement the following procedure:

```
void construct_instructions(int R, int N)
```

- R: the type of the box. If $R = 0$, the box is a normal box. If $R = 1$, the box is a recurbox.
- N: the number of rows and columns of the grid.
- This procedure is called exactly once and should construct a sequence of instructions to perform the required task.

This procedure should call one or more of the following procedures to construct a sequence of instructions:

```
void append_F()
void append_L()
void append_R()
void append_J(string label)
void append_BB(string label)
void append_BW(string label)
void append_DIE()
```

- Each procedure appends a `F`, `L`, `R`, `J`, `BB`, `BW`, `DIE` instruction to the program, respectively.
- For all relevant instructions, `label` must be an existing label created using the procedure `append_label`. The label can be created before or after it is used.
- These procedures must be called at most 10^6 times **in total**.
- In addition to the previous constraint, the procedures `append_J`, `append_BB` and `append_BW` must be called at most 10^5 times **in total**.

The following procedure should be called to insert a label between instructions:

```
void append_label(string label)
```

- `label` should be a string of at most 30 characters. There are no restrictions in the contents of the string.
- `label` should be unique across all calls to `append_label`.
- Any call to this procedure does not add to the number of constructed instructions. However, this procedure must be called at most 10^5 times.

After appending the last instruction, `construct_instructions` should return. The program will then be evaluated on a number of test cases. See the "Sample Grader" section for more details.

Note that in the actual judging, the grader may consume up to 0.3 seconds of your runtime and consume up to 128 MB of memory.

SCORING

If your program is incorrect, the score of your solution will be 0 in the subtask. Otherwise, let K be the number of instructions in your program. Then, your score (in percent) for the subtask will be calculated according to the respective table for $R = 0$ or $R = 1$:

$R = 0$:

Condition	Score
$60 < K \leq 10^6$	$(184.5 - 60 \ln \ln K)\%$
$K \leq 60$	100%

$R = 1$:

Condition	Score
$100 < K \leq 10^6$	$(191.5 - 60 \ln \ln K)\%$
$K \leq 100$	100%

EXAMPLE

Suppose it is guaranteed that $R = 0$, $N = 5$, the box is initially located at $(2, 2)$ and the exit is initially located at either $(1, 2)$ or $(1, 3)$. Then, a possible solution is to make the following calls:

```
append_L() append_F() append_L()
append_BW("exit-right")
append_L() append_F() append_L() append_F() append_F() append_L() append_F()
append_L() append_F() append_F() append_J("end")
append_label("exit-right")
append_L() append_F() append_L() append_F() append_L() append_F() append_R()
append_F() append_L() append_F() append_L() append_F() append_F()
append_label("end")
```

The implementation first controls Robo to reach $(1, 2)$, facing up. Then, it uses the Branch if Wall instruction to detect whether the exit is located at $(1, 2)$. In either case, it then executes a sequence of F, L and R instructions to push the box to the exit.

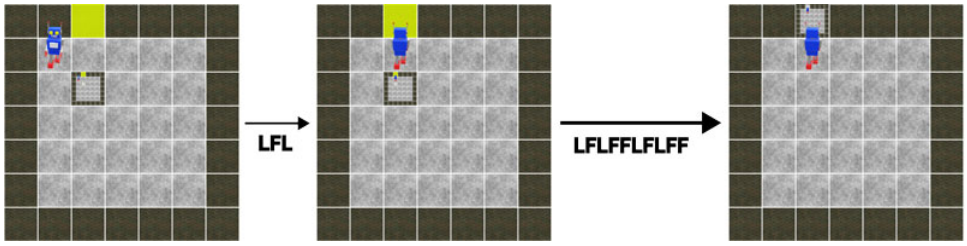


Figure 5: The outcome of the above sequence of instructions when the exit is located at $(1, 2)$.

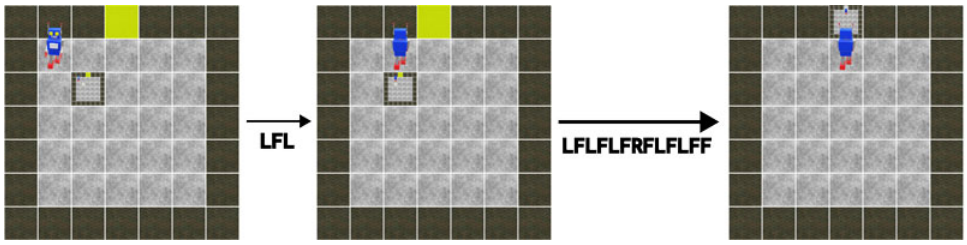


Figure 6: The outcome of the above sequence of instructions when the exit is located at $(1, 3)$.

Note that the box always stays at the exit once it is pushed there. Therefore, one can append more auxiliary instructions (e.g. F) without affecting its correctness, as long as the program still terminates eventually.

Next, suppose it is guaranteed that $R = 1$, $N = 5$, the box is initially located at $(2, 2)$ and the exit is initially located at $(2, 5)$. However, the black hole is located at $(3, 1)$. As the box cannot be pushed to the right, it is impossible to push the box to the exit. A possible solution is to make the following calls:

```
append_F() append_L() append_F() append_DIE() append_F()
```

The instruction DIE is successfully executed, reporting that it is impossible to push the box to the exit under the initial configuration. The F instruction following DIE is not executed.

SAMPLE GRADER

The sample grader reads input from standard input in the following format:

- The first line consists of two integers R and N, the arguments passed to `construct_instructions`.
- The second line consists of a single integer Q, the number of test cases.
- The next Q lines each contains a test case. If $R = 0$, then a test case is represented by four integers r_B, c_B, r_E and c_E, such that the box is initially located at (r_B, c_B) and the exit is located at (r_E, c_E). If $R = 1$, then two **additional integers** r_T and c_T follow, where (r_T, c_T) is the location of the blackhole.

If the sequence of instructions is incorrect, or it gives an incorrect behavior for any of the test cases, the sample grader outputs `Wrong Answer: Reason`. Otherwise, it outputs whether Robo pushes the box to the exit, or reports `DIE`. At the end, it outputs the total number of instructions used.

If an invalid input is supplied to the sample grader, the behaviour of the grader is undefined.

The sample grader also generates a file `program.robo` that can be loaded into the display tool (see the "Display Tool" section).

SAMPLE TESTS

	Input	Output

1	`0 5`
	`2`
	`2 2 1 4`
	`2 4 1 2`
		`Test 1: Accepted: Box is at exit`
		`Test 2: Accepted: Box is at exit`
		`1000000 instructions used`

2	`1 5`
	`3`
	`2 3 1 4 1 4`
	`2 3 1 2 3 1`
	`2 3 1 2 5 4`
		`Test 1: Accepted: Box is at exit`
		`Test 2: Accepted: Box is at exit`
		`Test 3: Reported DIE`
		`1000000 instructions used`

The number of instructions used could be less than 1000000 (a placeholder value). However, Robo must correctly push the box to the exit in the first 2 test cases and report that the third test case is impossible.

SUBTASKS

For all cases:
$5 \leq N \leq 100$
R is either 0 or 1

	Points	Constraints
1	8	$R = 0$ $N = 5$ The box is initially located on the second row (i.e. $r = 2$) The exit is located on the top edge (i.e. $r = 1$)
2	15	$R = 0$ The box is initially located on the second row (i.e. $r = 2$) The exit is located on the top edge (i.e. $r = 1$)
3	37	$R = 0$
4	14	$R = 1$ The box is initially located on the second row (i.e. $r = 2$) The exit is located on the top edge (i.e. $r = 1$) It is always possible to push the box to the exit
5	10	$R = 1$ The box is initially located on the second row (i.e. $r = 2$) The exit is located on the top edge (i.e. $r = 1$)
6	16	$R = 1$

DISPLAY TOOL

The attachment package `robo-visualizer.zip` contains a file named `display.html`. When opened in a browser, a graphical interface shows up and you can visualize Robo's execution under different initial configurations. The main features are as follows:

- You can observe the status of the realm, including the latest locations of Robo and the box. You can also observe which instruction Robo is currently executing.
- You can browse through the steps of Robo by clicking the left and right buttons. You can also jump to a specific step.
- You can play the simulation automatically by pressing the play button. The speed of execution can be adjusted using the slide bar in the bottom left corner.
- You can limit the number of steps simulated by imposing a step limit (**including labels**). Its default value is 10^5.

You should load in the sequence of instructions by selecting the file `program.robo`. The values of R and N are defined inside the file and are fixed. Alternatively, you may load the two files provided, `sample1.robo` and `sample2.robo`. They correspond to the two examples shown above.

Note that simulating with large grids and large step limits might cause your browser to hang. You are advised to simulate with the default step limit of 10^5 and on small grids.