Educational Codeforces Round 168 (Rated for Div. 2)

A. Strong Password

2 seconds, 512 megabytes

Monocarp's current password on Codeforces is a string s, consisting of lowercase Latin letters. Monocarp thinks that his current password is too weak, so he wants to **insert exactly one lowercase Latin letter** into the password to make it stronger. Monocarp can choose any letter and insert it anywhere, even before the first character or after the last character.

Monocarp thinks that the password's strength is proportional to the time it takes him to type the password. The time it takes to type the password is calculated as follows:

- the time to type the first character is 2 seconds;
- for each character other than the first, the time it takes to type it is 1 second if it is the same as the previous character, or 2 seconds otherwise.

For example, the time it takes to type the password `abacaba` is 14; the time it takes to type the password `a` is 2; the time it takes to type the password `aaabacc` is 11.

You have to help Monocarp — insert a lowercase Latin letter into his password so that the resulting password takes the maximum possible amount of time to type.

Input

The first line contains one integer t ($1 \le t \le 1000$) — the number of test cases.

Each test case consists of one line containing the string s ($1 \le |s| \le 10$), consisting of lowercase Latin letters.

Output

For each test case, print one line containing the new password — a string which can be obtained from s by inserting one lowercase Latin letter. The string you print should have the maximum possible required time to type it. If there are multiple answers, print any of them.

input

```
4
a
aaa
abb
password
```

output

```
wa
aada
abcb
pastsword
```

The problem statement has recently been changed. View the changes.

×

B. Make Three Regions

2 seconds, 256 megabytes

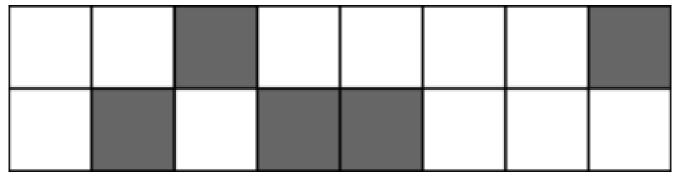
There is a grid, consisting of 2 rows and n columns. Each cell of the grid is either free or blocked.

A free cell y is reachable from a free cell x if at least one of these conditions holds:

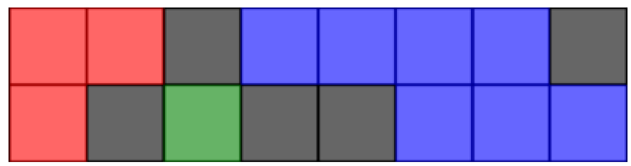
- x and y share a side;
- there exists a free cell z such that z is reachable from x and y is reachable from z.

A connected region is a set of free cells of the grid such that all cells in it are reachable from one another, but adding any other free cell to the set violates this rule.

For example, consider the following layout, where white cells are free, and dark grey cells are blocked:



There are 3 regions in it, denoted with red, green and blue color respectively:



The given grid contains at most 1 connected region. Your task is to calculate the number of free cells meeting the following constraint:

- if this cell is blocked, the number of connected regions becomes exactly 3.

Input

The first line contains a single integer t ($1 \le t \le 10^4$) — the number of test cases.

The first line of each test case contains a single integer n ($1 \le n \le 2 \cdot 10^5$) — the number of columns.

The i-th of the next two lines contains a description of the i-th row of the grid — the string s_i, consisting of n characters. Each character is either `.` (denoting a free cell) or `x` (denoting a blocked cell).

Additional constraint on the input:

- the given grid contains at most 1 connected region;
- the sum of n over all test cases doesn't exceed $2 \cdot 10^5$.

Output

For each test case, print a single integer — the number of cells such that the number of connected regions becomes 3 if this cell is blocked.

input

```
4
8
.......x
.x.xx...
2
..
..
3
xxx
xxx
9
..x.x.x.x
x.......x
```

output

```
1
0
0
2
```

In the first test case, if the cell $(1, 3)$ is blocked, the number of connected regions becomes 3 (as shown in the picture from the statement).

C. Even Positions

2 seconds, 256 megabytes

Monocarp had a regular bracket sequence s of length n (n is even). He even came up with his own way to calculate its cost.

He knows that in a regular bracket sequence (RBS), each opening bracket is paired up with the corresponding closing bracket. So he decided to calculate the *cost* of RBS as the sum of distances between pairs of corresponding bracket pairs.

For example, let's look at RBS `(())()`. It has three pairs of brackets:

- `(__)__`: the distance between brackets at position 1 and at 4 is $4 - 1 = 3$;
- `_()___`: the distance is $3 - 2 = 1$;
- `____()`: the distance is $6 - 5 = 1$.

So the cost of `(())()` is $3 + 1 + 1 = 5$.

Unfortunately, due to data corruption, Monocarp lost all characters on odd positions $s_1, s_3, \dots, s_{n-1}$. Only characters on even positions ($s_2, s_4, \dots, s_n$) remain. For example, `(())()` turned to `_(_)_)`.

Monocarp wants to restore his RBS by placing brackets on the odd positions. But since the restored RBS may not be unique, he wants to choose one with **minimum cost**. It's too hard to do for Monocarp alone, so can you help him?

Reminder: A *regular bracket sequence* is a string consisting of only brackets, such that this sequence, when inserted `1`-s and `+`-s, gives a valid mathematical expression. For example, `()`, `(())` or `(()())()` are RBS, while `)`, `()(` or `())(()` are not.

Input

The first line contains a single integer t ($1 \le t \le 5000$) — the number of test cases. Next t cases follow.

The first line of each test case contains a single integer n ($2 \le n \le 2 \cdot 10^5$; n is even) — the length of string s.

The second line of each test case contains a string s of length n, where all characters on the odd positions are '_' and all characters on the even positions are either '(' or ')'.

Additional constraints:

- s can be restored to at least one regular bracket sequence;
- the total sum of n over all test cases doesn't exceed $2 \cdot 10^5$.

Output

For each test case, print one integer — the minimum cost of the regular bracket sequence that can be obtained from s by replacing '_'-s with brackets.

input

```
4
6
_(_)_)
2
_)
8
_)_)_)_)
8
_(_)_(_)
```

output

```
5
1
4
8
```

In the first test case, it's optimal to make s equal to `(())()`. The cost of s will be equal to $3 + 1 + 1 = 5$.

In the second test case, the only option is to make s equal to `()` with cost 1.

In the third test case, the only possible RBS is `()()()()` with cost $1 + 1 + 1 + 1 = 4$.

In the fourth test case, it's optimal to make s equal to `(())(())` with cost $3 + 1 + 3 + 1 = 8$.

D. Maximize the Root

3 seconds, 256 megabytes

You are given a rooted tree, consisting of n vertices. The vertices in the tree are numbered from 1 to n, and the root is the vertex 1. The value a_i is written at the i-th vertex.

You can perform the following operation any number of times (possibly zero): choose a vertex v **which has at least one child**; increase a_v by 1; and decrease a_u by 1 for all vertices u that are in the subtree of v (except v itself). However, after each operation, the values on all vertices should be non-negative.

Your task is to calculate the maximum possible value written at the root using the aforementioned operation.

Input

The first line contains a single integer t ($1 \le t \le 10^4$) — the number of test cases.

The first line of each test case contains a single integer n ($2 \le n \le 2 \cdot 10^5$) — the number of vertices in the tree.

The second line contains n integers $a_1, a_2, \dots, a_n$ ($0 \le a_i \le 10^9$) — the initial values written at vertices.

The third line contains $n - 1$ integers $p_2, p_3, \dots, p_n$ ($1 \le p_i \le n$), where p_i is the parent of the i-th vertex in the tree. Vertex 1 is the root.

Additional constraint on the input: the sum of n over all test cases doesn't exceed $2 \cdot 10^5$.

Output

For each test case, print a single integer — the maximum possible value written at the root using the aforementioned operation.

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

In the first test case, the following sequence of operations is possible:

- perform the operation on $v = 3$, then the values on the vertices will be $[0, 1, 1, 1]$;
- perform the operation on $v = 1$, then the values on the vertices will be $[1, 0, 0, 0]$.

E. Level Up

4 seconds, 512 megabytes

Monocarp is playing a computer game. He starts the game being level 1. He is about to fight n monsters, in order from 1 to n. The level of the i-th monster is a_i.

For each monster in the given order, Monocarp's encounter goes as follows:

- if Monocarp's level is strictly higher than the monster's level, the monster flees (runs away);
- otherwise, Monocarp fights the monster.

After every k-th fight with a monster (**fleeing monsters do not count**), Monocarp's level increases by 1. So, his level becomes 2 after k monsters he fights, 3 after $2k$ monsters, 4 after $3k$ monsters, and so on.

You need to process q queries of the following form:

- $i\ x$: will Monocarp fight the i-th monster (or will this monster flee) if the parameter k is equal to x?

Input

The first line contains two integers n and q ($1 \le n, q \le 2 \cdot 10^5$) — the number of monsters and the number of queries.

The second line contains n integers $a_1, a_2, \dots, a_n$ ($1 \le a_i \le 2 \cdot 10^5$) — the levels of the monsters.

In the j-th of the following q lines, two integers i and x ($1 \le i, x \le n$) — the index of the monster and the number of fights required for a level up in the j-th query.

Output

For each query, output "YES", if Monocarp will fight the i-th monster in this query, and "NO", if the i-th monster flees.

input
```
4 16
2 1 2 1
1 1
2 1
3 1
4 1
1 2
2 2
3 2
4 2
1 3
2 3
3 3
4 3
1 4
2 4
3 4
4 4
```
output
```
YES
NO
YES
NO
YES
YES
YES
NO
YES
YES
YES
NO
YES
YES
YES
YES
```

input
```
7 15
1 1 2 1 1 1 1
5 3
2 2
2 2
1 6
5 1
5 5
7 7
3 5
7 4
4 3
2 5
1 2
5 6
4 1
6 1
```
output
```
NO
YES
YES
YES
NO
YES
YES
YES
NO
NO
YES
YES
YES
NO
NO
```

F. Chips on a Line

5 seconds, 512 megabytes

You have n chips, and you are going to place all of them in one of x points, numbered from 1 to x. There can be multiple chips in each point.

After placing the chips, you can perform the following four operations (in any order, any number of times):

- choose a chip in point $i \geq 3$, remove it and place two chips: one in $i-1$, one in $i-2$;
- choose two chips in adjacent points i and $i+1$, remove them and place a new chip in $i+2$;
- choose a chip in point 1 and move it to 2;
- choose a chip in point 2 and move it to 1.

Note that the coordinates of the chips you place during the operations cannot be less than 1, but can be greater than x.

Denote the *cost* of chip placement as the **minimum** number of chips which can be present on the line after you perform these operations, starting from the placement you've chosen.

For example, the *cost* of placing two chips in points 3 and one chip in point 5 is 2, because you can reduce the number of chips to 2 as follows:

- choose a chip in point 3, remove it, place a chip in 1 and another chip in 2;
- choose the chips in points 2 and 3, remove them and place a chip in 4;
- choose the chips in points 4 and 5, remove them and place a chip in 6.

You are given three integers n, x and m. Calculate the number of placements of exactly n chips in points from 1 to x having cost equal to m, and print it modulo 998244353. Two placements are considered different if the number of chips in some point differs between these placements.

Input

The only line contains three integers n, x and m ($1 \leq m \leq n \leq 1000$; $2 \leq x \leq 10$).

Output

Print one integer — the number of placements with cost equal to m, taken modulo 998244353.

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

input
```
42 10 5
```
output
```
902673363
```

input
```
1000 10 8
```
output
```
187821763
```

In the first example, there are five ways to place 2 chips in points from 1 to 3 so that the cost is 1:

- $(1,1)$;
- $(1,2)$;
- $(1,3)$;
- $(2,2)$;
- $(2,3)$.