# A DETAILED EDITORIAL FOR ATCODER - FREEE PROGRAMMING CONTEST 2023 （ATCODER BEGINNER CONTEST 310）

Date of Contest: 15-07-2023

**Editorial by**
**Dr. Dinesh Kumar Anguraj, Asso. Prof., Dept. of CSE.**

**Website Link: https://atcoder.jp/**

**Contest URL: https://atcoder.jp/contests/abc310**

**Mode: Rated & Unrated**

**Number of Contest Tasks: 7 Tasks**

**Number of Extra Tasks: 1 Task**

**Note & Credits: Problem statement and Hints from Atcoder.jp by a user "en_translator"**

# CONTEST PROBLEM STATEMENTS

# A - Order Something Else

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $100$ points

## Problem Statement

Takahashi wants a beverage called AtCoder Drink in a restaurant. It can be ordered at a regular price of $P$ yen.

He also has a discount coupon that allows him to order it at a lower price of $Q$ yen. However, he must additionally order one of the restaurant's $N$ dishes to use that coupon. For each $i = 1, 2, \ldots, N$, the price of the $i$-th dish is $D_i$ yen.

Print the minimum total amount of money that he must pay to get the drink.

## Constraints

- $1 \leq N \leq 100$
- $1 \leq Q < P \leq 10^5$
- $1 \leq D_i \leq 10^5$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

$$\begin{array}{l} N \quad P \quad Q \\ D_1 \quad D_2 \quad \ldots \quad D_N \end{array}$$

## Output

Print the answer.

## Sample Input 1

```
3 100 50
60 20 40
```

## Sample Output 1

```
70
```

If he uses the coupon and orders the second dish, he can get the drink by paying $50$ yen for it and $20$ yen for the dish, for a total of $70$ yen, which is the minimum total payment needed.

## Sample Input 2

```
3 100 50
60000 20000 40000
```

## Sample Output 2

```
100
```

The total payment will be minimized by not using the coupon and paying the regular price of $100$ yen.

# B - Strictly Superior

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $200$ points

## Problem Statement

AtCoder Shop has $N$ products. The price of the $i$-th product $(1 \leq i \leq N)$ is $P_i$. The $i$-th product $(1 \leq i \leq N)$ has $C_i$ functions. The $j$-th function $(1 \leq j \leq C_i)$ of the $i$-th product $(1 \leq i \leq N)$ is represented as an integer $F_{i,j}$ between $1$ and $M$, inclusive.

Takahashi wonders whether there is a product that is strictly superior to another. If there are $i$ and $j$ $(1 \leq i, j \leq N)$ such that the $i$-th and $j$-th products satisfy all of the following conditions, print `Yes`; otherwise, print `No`.

- $P_i \geq P_j$.
- The $j$-th product has all functions of the $i$-th product.
- $P_i > P_j$, or the $j$-th product has one or more functions that the $i$-th product lacks.

## Constraints

- $2 \leq N \leq 100$
- $1 \leq M \leq 100$
- $1 \leq P_i \leq 10^5$ $(1 \leq i \leq N)$
- $1 \leq C_i \leq M$ $(1 \leq i \leq N)$
- $1 \leq F_{i,1} < F_{i,2} < \cdots < F_{i,C_i} \leq M$ $(1 \leq i \leq N)$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

$$\begin{array}{l}
N \quad M \\
P_1 \quad C_1 \quad F_{1,1} \quad F_{1,2} \quad \ldots \quad F_{1,C_1} \\
P_2 \quad C_2 \quad F_{2,1} \quad F_{2,2} \quad \ldots \quad F_{2,C_2} \\
\vdots \\
P_N \quad C_N \quad F_{N,1} \quad F_{N,2} \quad \ldots \quad F_{N,C_N}
\end{array}$$

## Output

Print the answer in a single line.

## Sample Input 1

```
5 6
10000 2 1 3
15000 3 1 2 4
30000 3 1 3 5
35000 2 1 5
100000 6 1 2 3 4 5 6
```

## Sample Output 1

```
Yes
```

$(i, j) = (4, 3)$ satisfies all of the conditions.

No other pair satisfies them. For instance, for $(i, j) = (4, 5)$, the $j$-th product has all functions of the $i$-th one, but $P_i < P_j$, so it is not strictly superior.

## Sample Input 2

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

## Sample Output 2

```
No
```

Multiple products may have the same price and functions.

## Sample Input 3

```
20 10
72036 3 3 4 9
7716 4 1 2 3 6
54093 5 1 6 7 8 10
25517 7 3 4 5 6 7 9 10
96930 8 2 3 4 6 7 8 9 10
47774 6 2 4 5 6 7 9
36959 5 1 3 4 5 8
46622 7 1 2 3 5 6 8 10
34315 9 1 3 4 5 6 7 8 9 10
54129 7 1 3 4 6 7 8 9
4274 5 2 4 7 9 10
16578 5 2 3 6 7 9
61809 4 1 2 4 5
1659 5 3 5 6 9 10
59183 5 1 2 3 4 9
22186 4 3 5 6 8
98282 4 1 4 7 10
72865 8 1 2 3 4 6 8 9 10
33796 6 1 3 5 7 9 10
74670 4 1 2 6 8
```

## Sample Output 3

```
Yes
```

# C - Reversible

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $300$ points

## Problem Statement

There are $N$ sticks with several balls stuck onto them. Each ball has a lowercase English letter written on it.

For each $i = 1, 2, \ldots, N$, the letters written on the balls stuck onto the $i$-th stick are represented by a string $S_i$. Specifically, the number of balls stuck onto the $i$-th stick is the length $|S_i|$ of the string $S_i$, and $S_i$ is the sequence of letters on the balls starting from one end of the stick.

Two sticks are considered the same when the sequence of letters on the balls starting from one end of one stick is equal to the sequence of letters starting from one end of the other stick. More formally, for integers $i$ and $j$ between $1$ and $N$, inclusive, the $i$-th and $j$-th sticks are considered the same if and only if $S_i$ equals $S_j$ or its reversal.

Print the number of different sticks among the $N$ sticks.

## Constraints

- $N$ is an integer.
- $2 \leq N \leq 2 \times 10^5$
- $S_i$ is a string consisting of lowercase English letters.
- $|S_i| \geq 1$
- $\sum_{i=1}^{N} |S_i| \leq 2 \times 10^5$

## Input

The input is given from Standard Input in the following format:

$$N$$
$$S_1$$
$$S_2$$
$$\vdots$$
$$S_N$$

## Output

Print the answer.

## Sample Input 1

```
6
a
abc
de
cba
de
abc
```

## Sample Output 1

```
3
```

- $S_2$ = `abc` equals the reversal of $S_4$ = `cba`, so the second and fourth sticks are considered the same.
- $S_2$ = `abc` equals $S_6$ = `abc`, so the second and sixth sticks are considered the same.
- $S_3$ = `de` equals $S_5$ = `de`, so the third and fifth sticks are considered the same.

Therefore, there are three different sticks among the six: the first, second (same as the fourth and sixth), and third (same as the fifth).

# D - Peaceful Teams

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $400$ points

## Problem Statement

There are $N$ sports players.

Among them, there are $M$ incompatible pairs. The $i$-th incompatible pair $(1 \leq i \leq M)$ is the $A_i$-th and $B_i$-th players.

You will divide the players into $T$ teams. Every player must belong to exactly one team, and every team must have one or more players. Additionally, for each $i = 1, 2, \ldots, M$, the $A_i$-th and $B_i$-th players must not belong to the same team.

Find the number of ways to satisfy these conditions. Here, two divisions are considered different when there are two players who belong to the same team in one division and different teams in the other.

## Constraints

- $1 \leq T \leq N \leq 10$
- $0 \leq M \leq \dfrac{N(N-1)}{2}$
- $1 \leq A_i < B_i \leq N \ (1 \leq i \leq M)$
- $(A_i, B_i) \neq (A_j, B_j) \ (1 \leq i < j \leq M)$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

$$
\begin{array}{l}
N \quad T \quad M \\
A_1 \quad B_1 \\
A_2 \quad B_2 \\
\vdots \\
A_M \quad B_M
\end{array}
$$

## Output

Print the answer in a single line.

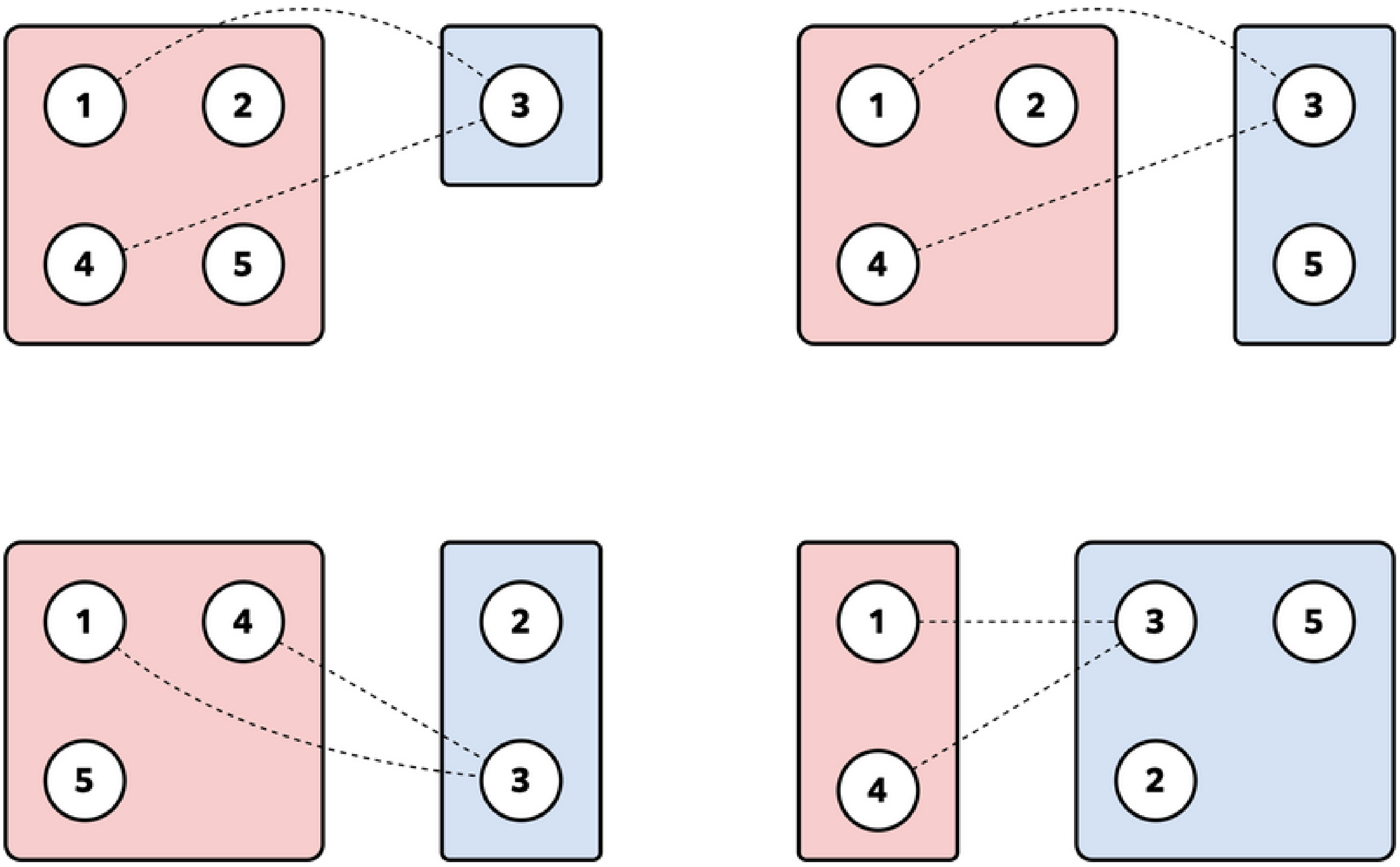
## Sample Input 1

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

## Sample Output 1

```
4
```

The following four divisions satisfy the conditions.



No other division satisfies them, so print $4$.

## Sample Input 2

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

## Sample Output 2

```
0
```

There may be no division that satisfies the conditions.

## Sample Input 3

```
6 4 0
```

## Sample Output 3

```
65
```

There may be no incompatible pair.

## Sample Input 4

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

## Sample Output 4

```
8001
```

# E - NAND repeatedly

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $450$ points

## Problem Statement

You are given a string $S$ of length $N$ consisting of `0` and `1`. It describes a length-$N$ sequence $A = (A_1, A_2, \ldots, A_N)$. If the $i$-th character of $S$ $(1 \leq i \leq N)$ is `0`, then $A_i = 0$; if it is `1`, then $A_i = 1$.

Find the following:

$$\sum_{1 \leq i \leq j \leq N} (\cdots((A_i \barwedge A_{i+1}) \barwedge A_{i+2}) \barwedge \cdots \barwedge A_j)$$

More formally, find $\displaystyle\sum_{i=1}^{N} \sum_{j=i}^{N} f(i,j)$ for $f(i,j)$ $(1 \leq i \leq j \leq N)$ defined as follows:

$$f(i,j) = \begin{cases} A_i & (i = j) \\ f(i, j-1) \barwedge A_j & (i < j) \end{cases}$$

Here, $\barwedge$, NAND, is a binary operator satisfying the following:

$$0 \barwedge 0 = 1, 0 \barwedge 1 = 1, 1 \barwedge 0 = 1, 1 \barwedge 1 = 0.$$

## Constraints

- $1 \leq N \leq 10^6$
- $S$ is a string of length $N$ consisting of `0` and `1`.
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

```
N
S
```

## Output

Print the answer in a single line.

## Sample Input 1

```
5
00110
```

## Sample Output 1

```
9
```

Here are the values of $f(i, j)$ for the pairs $(i, j)$ such that $1 \leq i \leq j \leq N$:

- $f(1,1) = 0 = 0$
- $f(1,2) = 0 \barwedge 0 = 1$
- $f(1,3) = (0 \barwedge 0) \barwedge 1 = 0$
- $f(1,4) = ((0 \barwedge 0) \barwedge 1) \barwedge 1 = 1$
- $f(1,5) = (((0 \barwedge 0) \barwedge 1) \barwedge 1) \barwedge 0 = 1$
- $f(2,2) = 0 = 0$
- $f(2,3) = 0 \barwedge 1 = 1$
- $f(2,4) = (0 \barwedge 1) \barwedge 1 = 0$
- $f(2,5) = ((0 \barwedge 1) \barwedge 1) \barwedge 0 = 1$
- $f(3,3) = 1 = 1$
- $f(3,4) = 1 \barwedge 1 = 0$
- $f(3,5) = (1 \barwedge 1) \barwedge 0 = 1$
- $f(4,4) = 1 = 1$
- $f(4,5) = 1 \barwedge 0 = 1$
- $f(5,5) = 0 = 0$

Their sum is $0 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 + 1 + 0 + 1 + 1 + 1 + 0 = 9$, so print $9$.

Note that $\barwedge$ does not satisfy the associative property. For instance, $(1 \barwedge 1) \barwedge 0 = 0 \barwedge 0 = 1 \neq 0 = 1 \barwedge 1 = 1 \barwedge (1 \barwedge 0)$.

## Sample Input 2

```
30
101010000100101011010011000010
```

## Sample Output 2

```
326
```

## F - Make 10 Again

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $500$ points

## Problem Statement

We have $N$ dice. For each $i = 1, 2, \ldots, N$, when the $i$-th die is thrown, it shows a random integer between $1$ and $A_i$, inclusive, with equal probability.

Find the probability, modulo $998244353$, that the following condition is satisfied when the $N$ dice are thrown simultaneously.

> There is a way to choose some (possibly all) of the $N$ dice so that the sum of their results is $10$.

▶ How to find a probability modulo $998244353$

## Constraints

- $1 \le N \le 100$
- $1 \le A_i \le 10^6$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

$$N \\ A_1 \quad A_2 \quad \ldots \quad A_N$$

## Output

Print the answer.

## Sample Input 1

```
4
1 7 2 9
```

## Sample Output 1

```
942786334
```

For instance, if the first, second, third, and fourth dice show $1, 3, 2$, and $7$, respectively, these results satisfy the condition. In fact, if the second and fourth dice are chosen, the sum of their results is $3 + 7 = 10$. Alternatively, if the first, third, and fourth dice are chosen, the sum of their results is $1 + 2 + 7 = 10$.

On the other hand, if the first, second, third, and fourth dice show $1, 6, 1$, and $5$, respectively, there is no way to choose some of them so that the sum of their results is $10$, so the condition is not satisfied.

In this sample input, the probability of the results of the $N$ dice satisfying the condition is $\frac{11}{18}$. Thus, print this value modulo $998244353$, that is, $942786334$.

## Sample Input 2

```
7
1 10 100 1000 10000 100000 1000000
```

## Sample Output 2

```
996117877
```

# G - Takahashi And Pass-The-Ball Game

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : $550$ points

## Problem Statement

There are $N$ Takahashi.

The $i$-th Takahashi has an integer $A_i$ and $B_i$ balls.

An integer $x$ between $1$ and $K$, inclusive, will be chosen uniformly at random, and they will repeat the following operation $x$ times.

- For every $i$, the $i$-th Takahashi gives all his balls to the $A_i$-th Takahashi.

Beware that all $N$ Takahashi simultaneously perform this operation.

For each $i = 1, 2, \ldots, N$, find the expected value, modulo $998244353$, of the number of balls the $i$-th Takahashi has at the end of the operations.

▶ How to find a expected value modulo $998244353$

## Constraints

- $1 \leq N \leq 2 \times 10^5$
- $1 \leq K \leq 10^{18}$
- $K$ is not a multiple of $998244353$.
- $1 \leq A_i \leq N \ (1 \leq i \leq N)$
- $0 \leq B_i < 998244353 \ (1 \leq i \leq N)$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

$$\begin{array}{llll} N & K & & \\ A_1 & A_2 & \cdots & A_N \\ B_1 & B_2 & \cdots & B_N \end{array}$$

## Output

Print the expected value of the number of balls the $i$-th Takahashi has at the end of the operations for $i = 1, 2, \ldots, N$, separated by spaces, in a single line.

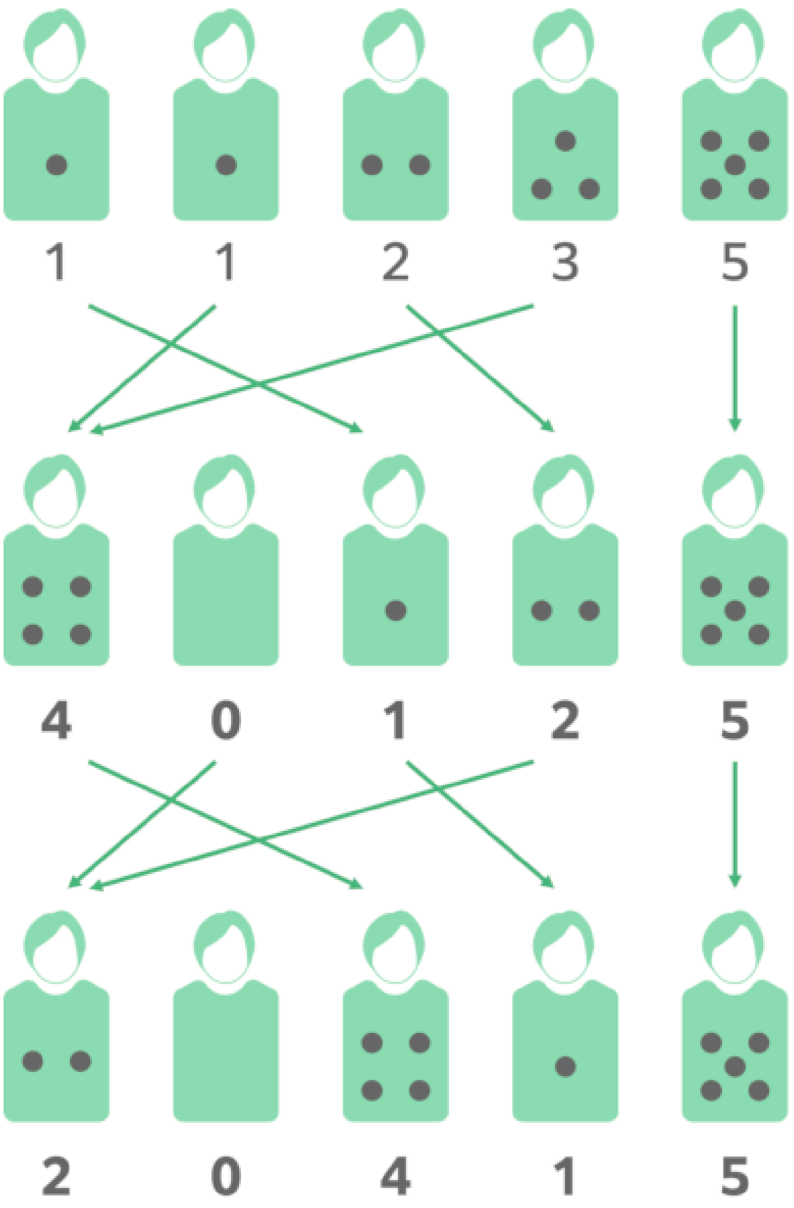
## Sample Input 1

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

## Sample Output 1

```
3 0 499122179 499122178 5
```

During two operations, the five Takahashi have the following number of balls.



If $x = 1$ is chosen, the five Takahashi have $4, 0, 1, 2, 5$ balls.
If $x = 2$ is chosen, the five Takahashi have $2, 0, 4, 1, 5$ balls.

Thus, the sought expected values are $3, 0, \frac{5}{2}, \frac{3}{2}, 5$. Print these values modulo $998244353$, that is, $3, 0, 499122179, 499122178, 5$, separated by spaces.

## Sample Input 2

```
3 1000
1 1 1
1 10 100
```

## Sample Output 2

```
111 0 0
```

After one or more operations, the first Takahashi gets all balls.

## Sample Input 3

```
16 1000007
16 12 6 12 1 8 14 14 5 7 6 5 9 6 10 9
719092922 77021920 539975779 254719514 967592487 476893866 368936979 465399362 342544824 540338192
42663741 165480608 616996494 16552706 590788849 221462860
```

## Sample Output 3

```
817852305 0 0 0 711863206 253280203 896552049 935714838 409506220 592088114 0 413190742 0 363914270
0 14254803
```

## Sample Input 4

```
24 100000000007
19 10 19 15 1 20 13 15 8 23 22 16 19 22 2 20 12 19 17 20 16 8 23 6
944071276 364842194 5376942 671161415 477159272 339665353 176192797 2729865 676292280 249875565 259
803120 103398285 466932147 775082441 720192643 535473742 263795756 898670859 476980306 12045411 620
291602 593937486 761132791 746546443
```

## Sample Output 4

```
918566373 436241503 0 0 0 455245534 0 356196743 0 906000633 0 268983266 21918337 0 733763572 173816
039 754920403 0 273067118 205350062 0 566217111 80141532 0
```

# Ex - Negative Cost

Time Limit: 3 sec / Memory Limit: 1024 MB

Score : $625$ points

## Problem Statement

A monster with health $H$ has appeared right in front of you. Your magic power is now $0$.

You can use $N$ moves called move $1$, move $2, \ldots$, move $N$, any number of times in any order.

For each $i = 1, 2, \ldots, N$, move $i$ can only be used when your magic power is at least $C_i$, and its use will decrease your magic power by $C_i$ and the monster's health by $D_i$. Here, if $C_i$ is negative, decreasing your magic power by $C_i$ means increasing it by $-C_i$.

Find the minimum possible number of times you use moves before the monster's health is $0$ or lower. The constraints of this problem guarantee that a finite number of uses of moves can make it $0$ or lower (see below).

## Constraints

- $1 \leq N \leq 300$
- $1 \leq H \leq 10^{18}$
- $-300 \leq C_i \leq 300$
- $C_i \leq 0$ for some $1 \leq i \leq N$.
- $1 \leq D_i \leq 10^9$
- All input values are integers.

## Input

The input is given from Standard Input in the following format:

```
N H
C_1 D_1
C_2 D_2
⋮
C_N D_N
```

## Output

Print the answer.

## Sample Input 1

```
3 48
3 20
-4 2
1 5
```

## Sample Output 1

```
5
```

From the initial state where your magic power is $0$ and the monster's health is $48$, consider using moves as follows.

- Use move $2$. Now, your magic power is $4$, and the monster's health is $46$.
- Use move $3$. Now, your magic power is $3$, and the monster's health is $41$.
- Use move $1$. Now, your magic power is $0$, and the monster's health is $21$.
- Use move $2$. Now, your magic power is $4$, and the monster's health is $19$.
- Use move $1$. Now, your magic power is $1$, and the monster's health is $-1$.

Here, you use moves five times before the monster's health is not greater than $0$, which is the minimum possible number.

## Sample Input 2

```
20 583988303060450752
-64 273760634
-238 960719353
-114 191410838
-250 357733867
232 304621362
-286 644706927
210 37849132
-230 556412112
-142 136397527
101 380675202
-140 152300688
190 442931589
-187 940659077
-12 312523039
32 126515475
-143 979861204
105 488280613
240 664922712
290 732741849
69 541282303
```

## Sample Output 2

```
595990842
```

# HINTS, ALGORITHMS AND SAMPLE IMPLEMENTATIONS FOR EACH PROBLEM STATEMENTS

**A – ORDER SOMETHING ELSE**

**Algorithm:**

Step 1: Read input values

1.1. Read the integer `n` - the number of dishes available.

1.2. Read the integer `p` - the price when the coupon is not used.

1.3. Read the integer `q` - the price when the coupon is used and the cheapest dish is ordered.

Step 2: Initialize variables

2.1. Initialize the variable `d_min` to a large value, e.g., 1000000000, to ensure that the first dish's price will be smaller than `d_min`.

2.2. Initialize a temporary variable `d` to store the price of each dish while inspecting them.

Step 3: Find the price of the cheapest dish

3.1. Start a loop from `i = 1` to `n` (inclusive) using a for loop, to inspect each dish one by one.

3.2. Within the loop, read the price of the current dish and store it in the variable `d`.

3.3. Update the value of `d_min` to be the minimum between the current `d` and the current value of `d_min`. This will help keep track of the cheapest dish price encountered so far.

Step 4: Compute and print the result

4.1. After the loop, compute the minimum price between `p` and `q + d_min`.

4.2. Print the result obtained in step 4.1.

Step 5: End of the algorithm

5.1. Exit the program.

The final C++ code using the algorithm above:

```
#include <iostream>
using namespace std;

int main(void)
{
   int n, p, q;
   cin >> n >> p >> q;
   int d_min = 1000000000, d;

   for (int i = 1; i <= n; i++)
   {
      cin >> d;
      d_min = min(d, d_min);
   }

   cout << min(p, q + d_min) << endl;
   return 0;
}
```

This algorithm will help you find and print the cheapest price between ordering without a coupon (`p`) and ordering with a coupon for the cheapest dish (`q + d_min`).

## B – STRICTLY SUPERIOR

**Algorithm:**

Step 1: Read input values

1.1. Read the integers `N` and `M` from the input - `N` represents the number of products, and `M` represents the number of features each product can have.

Step 2: Read product information

2.1. Create a vector of pairs called `products`, where each pair consists of an integer `price` and a vector of integers `feature`.

2.2. Iterate `N` times to read the product information for each product.

2.3. For each product, read the `price` first.

2.4. Read the integer `K` - the number of features for the current product.

2.5. Create a vector of integers called `feature` with size `K`.

2.6. Read `K` integers and store them in the `feature` vector to represent the features of the current product.

Step 3: Define `is_strictly_superior` function

3.1. Create a function called `is_strictly_superior`, which takes two pairs of `int` and `vector<int>` as input parameters representing two products - `product_i` and `product_j`.

3.2. Extract `price_i` and `feature_i` from `product_i`, and `price_j` and `feature_j` from `product_j`.

3.3. Check if the price of `product_i` is greater than or equal to the price of `product_j`.

3.4. Use the `std::includes` function to check if all the features of `product_i` are contained in the features of `product_j`.

3.5. Also, check if the price of `product_i` is strictly greater than the price of `product_j`, or if the prices are the same but the features of `product_j` are not contained in the features of `product_i`.

Step 4: Define `has_superior_pair` function

4.1. Create a function called `has_superior_pair`, which takes the `products` vector as an input parameter.

4.2. Iterate through each product in the `products` vector using `any_of` function to check if there exists any product (`product_i`) that has a superior pair.

4.3. For each `product_i`, again use `any_of` function to check if there exists any product (`product_j`) in the `products` vector such that `product_j` is strictly superior to `product_i` using the `is_strictly_superior` function.

};

Step 5: Check if a superior pair exists

5.1. Call the `has_superior_pair` function with the `products` vector as an argument.

5.2. If the function returns `true`, output "Yes"; otherwise, output "No".

Step 6: End of the algorithm

6.1. Exit the program.

The final C++ code using the algorithm above:

```
#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;


bool is_strictly_superior(const pair<int, vector<int>>& product_i, const pair<int, vector<int>>& product_j) {

   const auto& [price_i, feature_i] = product_i;

   const auto& [price_j, feature_j] = product_j;

   return price_i >= price_j &&

         includes(feature_j.begin(), feature_j.end(), feature_i.begin(), feature_i.end()) &&

         (price_i > price_j || !includes(feature_i.begin(), feature_i.end(), feature_j.begin(), feature_j.end()));

}


bool has_superior_pair(const vector<pair<int, vector<int>>>& products) {

   return any_of(products.begin(), products.end(), [&products](const auto& product_i) {

      return any_of(products.begin(), products.end(), [&product_i](const auto& product_j) {

         return is_strictly_superior(product_i, product_j);

      });

   });

}
```

```cpp
int main() {
    int N, M;
    cin >> N >> M;
    vector<pair<int, vector<int>>> products(N);
    for (auto& [price, feature] : products) {
        cin >> price;
        int K;
        cin >> K;
        feature = vector<int>(K);
        for (auto& f : feature)
            cin >> f;
    }
    cout << (has_superior_pair(products) ? "Yes" : "No") << endl;
    return 0;
}
```

This algorithm will determine if there exists a pair of products in the given list such that one product is strictly superior to the other, based on their prices and features.

**C-Reversible**

**Algorithm:**

Sure! Here's a step-by-step algorithm for the provided C++ code:

Step 1: Read input value

1.1. Read the integer `n` from the input, representing the number of sticks.

Step 2: Initialize variables

2.1. Create a variable `ans` and set it to 0. This variable will keep track of the number of distinct sticks encountered so far.

2.2. Create an empty `set<string>` called `T`. This set will store the strings of the sticks inspected so far, along with their reversals.

Step 3: Inspect sticks

3.1. Start a loop from `i = 1` to `n` (inclusive) using a for loop, to inspect each stick one by one.

3.2. Within the loop, read the string of the current stick and store it in the variable `s`.

3.3. Check if the string `s` is not present in the set `T` by using the `count()` function. If it is not present, increment the value of `ans` by 1, as it indicates that this stick is a distinct one.

3.4. Insert the string `s` into the set `T` using `insert()` to keep track of it.

3.5. Reverse the string `s`.

3.6. Check if the reversed string of `s` is not present in the set `T`. If it is not present, increment the value of `ans` by 1, as the reversed string is also distinct.

3.7. Insert the reversed string of `s` into the set `T` to keep track of it.

Step 4: Print the result

4.1. After the loop, print the value of `ans`, which represents the number of distinct sticks encountered.

Step 5: End of the algorithm

5.1. Exit the program.

The final C++ code using the algorithm above:

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main(void)
{
    int n;
    cin >> n;
    int ans = 0;
    set<string> T;
    string s;

    for (int i = 1; i <= n; i++)
    {
        cin >> s;
        if (T.count(s) == 0)
            ans++;
        T.insert(s);
        reverse(s.begin(), s.end());
        T.insert(s);
    }

    cout << ans << endl;
    return 0;
}
```

This algorithm will efficiently count the number of distinct sticks by using a balanced binary tree (implemented as a `std::set`) to keep track of the strings of the sticks and their reversals inspected so far. The algorithm ensures that it finishes within the execution time limit by using the set's fast lookup operation.

**<u>D – Peaceful Teams:</u>**

**Algorithm:**

The problem can be solved by adding players to a team one by one with certain constraints on player incompatibility. The main idea is to use a recursive function to explore all possible combinations of dividing players into teams while ensuring that no incompatible players belong to the same team.

Here's the step-by-step algorithm for the given code:

Step 1: Read input values

1.1. Read the integers `N`, `T`, and `M` from the input.

- `N` represents the number of players.
- `T` represents the desired number of teams.
- `M` represents the number of pairs of players who are incompatible with each other.

Step 2: Read incompatibility data

2.1. Create a vector called `hate` of size `N`, where each element is an unsigned integer.

2.2. Iterate `M` times to read the incompatibility pairs and store them in the `hate` vector.

- Each pair consists of integers `a` and `b`, where `a` and `b` are player indices (0-indexed) who are incompatible with each other.
- Set the corresponding bits in the `hate` vector to indicate player incompatibility.

Step 3: Define a recursive function `dfs`

3.1. Create a lambda function called `dfs`, which takes the following parameters:

- `auto&& f`: A reference to itself for recursion.
- `vector<unsigned>& teams`: A reference to the vector that holds the teams' bitmasks.
- `unsigned now`: The index of the player currently being considered.

3.2. The `dfs` function returns the number of valid combinations to divide players into teams.

Step 4: Implement the recursive function `dfs`

4.1. The `dfs` function performs the following steps recursively:

- If all `N` players are considered (`now == N`), check if the number of teams formed is equal to the desired number of teams `T`. If yes, return 1; otherwise, return 0.

- If the current player `now` can be added to an existing team without causing any incompatibility issues, try adding the player to each existing team. Recursively call `dfs` for the next player (`now + 1`), considering all possible combinations.

- If there are fewer teams formed than the desired number of teams `T`, try creating a new team with the current player. Recursively call `dfs` for the next player (`now + 1`), considering all possible combinations.

Step 5: Call the recursive function and print the result

5.1. Call the `dfs` function with an initial empty vector for teams and the starting player index `0`.

5.2. Print the result obtained from the `dfs` function, which represents the number of valid combinations to divide players into teams without any incompatible players in the same team.

Step 6: End of the algorithm

6.1. Exit the program.

The final C++ code using the algorithm above:

```cpp
#include <iostream>
#include <vector>

int main() {
    using namespace std;
    unsigned N, T, M;
    cin >> N >> T >> M;

    // j-th bit of hate[i] is 1 ⟹ i-th and j-th players are incompatible (0-indexed)
    vector<unsigned> hate(N);
    for(unsigned i{}, a, b; i < M; ++i){
        cin >> a >> b;
        hate[--b] |= 1U << --a;
    }
```

```
// Define the recursive function dfs
auto dfs{[N, T, &hate](auto&& f, vector<unsigned>& teams, unsigned now) -> unsigned {
  // OK if there are T teams so far when all the N players are inspected
  if(now == N)
    return size(teams) == T;

  unsigned ans{};

  // Add the now-th player to an existing team
  for(auto&& team : teams)
    // If nobody in the team is incompatible with the now-th player
    if(!(team & hate[now])){
      team ^= 1U << now;
      ans += f(f, teams, now + 1);
      team ^= 1U << now;
    }

  // If a new team can be made, make a new one
  if(size(teams) < T){
    teams.emplace_back(1U << now);
    ans += f(f, teams, now + 1);
    teams.pop_back();
  }

  return ans;
}};

// Print the result of the recursive function
cout << [dfs{[N, T, &hate](auto&& f, vector<unsigned>& teams, unsigned now) -> unsigned {
  if(now == N)
    return size(teams) == T;
```

```
        unsigned ans{};
        for(auto&& team : teams)
            if(!(team & hate[now])){
                team ^= 1U << now;
                ans += f(f, teams, now + 1);
                team ^= 1U << now;
            }
        if(size(teams) < T){
            teams.emplace_back(1U << now);
            ans += f(f, teams, now + 1);
            teams.pop_back();
        }
        return ans;
    }}, T]{
        vector<unsigned> team;
        team.reserve(T);
        return dfs(dfs, team, 0);
    }() << endl;

    return 0;
}
```

This algorithm efficiently explores all possible combinations of dividing players into teams while ensuring that no incompatible players belong to the same team. It uses a recursive approach and bitwise operations to achieve a time complexity of O(N * N!) with a reasonable number of players.

## E – NAND repeatedly

**Algorithm**

We can efficiently solve the problem by managing the counts of '0' and '1' in the input string 'A'. The algorithm has a time complexity of Θ(N) since it only iterates through the input string once, and it does not require any nested loops or expensive operations.

Here's the step-by-step algorithm for the given solution:

Step 1: Read input values

1.1. Read the integer `N` from the input, representing the length of the string.

1.2. Read the string `A` from the input, which consists of '0's and '1's.

Step 2: Initialize variables

2.1. Create two variables `zero` and `one`, both initialized to 0. These variables will keep track of the counts of '0's and '1's in the processed part of the string.

2.2. Create a variable `ans` and set it to 0. This variable will keep track of the total sum of '1's encountered so far.

Step 3: Iterate through the string

3.1. Iterate through each character `a` in the string `A`.

3.2. Inside the loop, update the counts of '0's and '1's based on the current character `a`.

- If `a` is '0', update `zero` to 1 and `one` to the sum of their previous values (`zero + one`).
- If `a` is '1', update `zero` to the sum of their previous values (`zero + one`) and `one` to the sum of their previous values plus 1 (`zero + one + 1`).

3.3. Add the value of `one` to the `ans` variable to keep track of the total sum of '1's.

Step 4: Print the result

4.1. After the loop, print the value of `ans`, which represents the sum of '1's obtained by processing the string `A`.

Step 5: End of the algorithm

5.1. Exit the program.

The final Python and C++ codes using the algorithm above:

**Python:**

```python
N = int(input())
A = input()

zero, one = 0, 0
ans = 0

for a in A:
    if a == '0':
        zero, one = 1, zero + one
    else:
        zero, one = one, zero + 1
    ans += one

print(ans)
```

**C++:**

```cpp
#include <vector>
#include <iostream>
#include <utility>

int main() {
    using namespace std;
    unsigned long N;
    cin >> N;
    vector<char> A(N);
    for (auto &&a : A) cin >> a;

    unsigned long ans = 0;
    using count_t = pair<unsigned long, unsigned long>;
    count_t counter;
    auto&& [zero, one] = counter;

    for (const auto a : A) {
        counter = a == '1' ? count_t{one, zero + 1} : count_t{1, one + zero};
        ans += one;
    }

    cout << ans << endl;
    return 0;
}
```

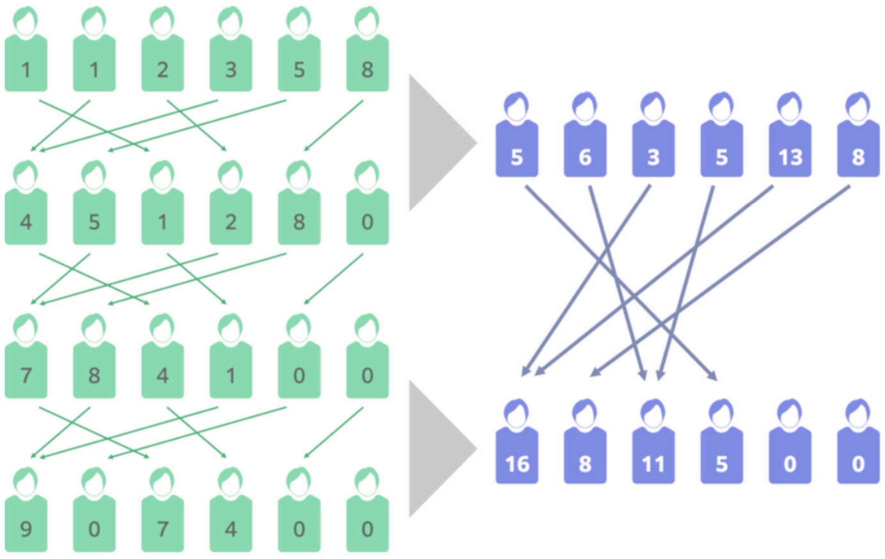
Both the Python and C++ implementations efficiently solve the problem and produce the correct output in linear time, making the algorithm much more efficient than the initial Θ(N^2)-time solution.

## G - Takahashi And Pass-The-Ball Game

## HINTS:

First, consider the state where the operation is actually performed once, to assume that between operation is performed between $0$ and $K-1$ more times. We seek for the answer multiplied by $K$ and finally divide it by $K$. Then, this problem asks to perform the operation $K$ times and find for each Takahashi the sum of the number of balls after each operation.

By adding the number of balls, two consecutive operations can be considered at once, so this problem can be solved by the doubling technique, solving it for the halved $K$. If $K$ is odd, we can consider the initial state and the state after the operation is performed once, and do the same process.



▼ Mathematical explanation

Define the following operations for length-$N$ sequences $S=(S_1,S_2,\dots,S_N)$ and $T=(T_1,T_2,\dots,T_N)$ consisting of $\{1,2,\dots,N\}$, length-$N$ sequences $\boldsymbol{x}=(x_1,x_2,\dots,x_N)$ and $\boldsymbol{y}=(y_1,y_2,\dots,y_N)$ of $\mathbb{F}_p$, and a non-negative integer $n$:

$$\boldsymbol{x}+\boldsymbol{y}=(x_1+y_1,x_2+y_2,\dots,x_N+y_N)$$

$$S\times\boldsymbol{x}=\left(\sum_{S_j=i}x_j\right)_{i=1,2,\dots,N}$$

$$S\times T=(S_{T_1},S_{T_2},\dots,S_{T_N})$$

$$S^n=\begin{cases}(1,2,\dots,N) & (n=0)\\ S^{n-1}\times S & (n>0)\end{cases}$$

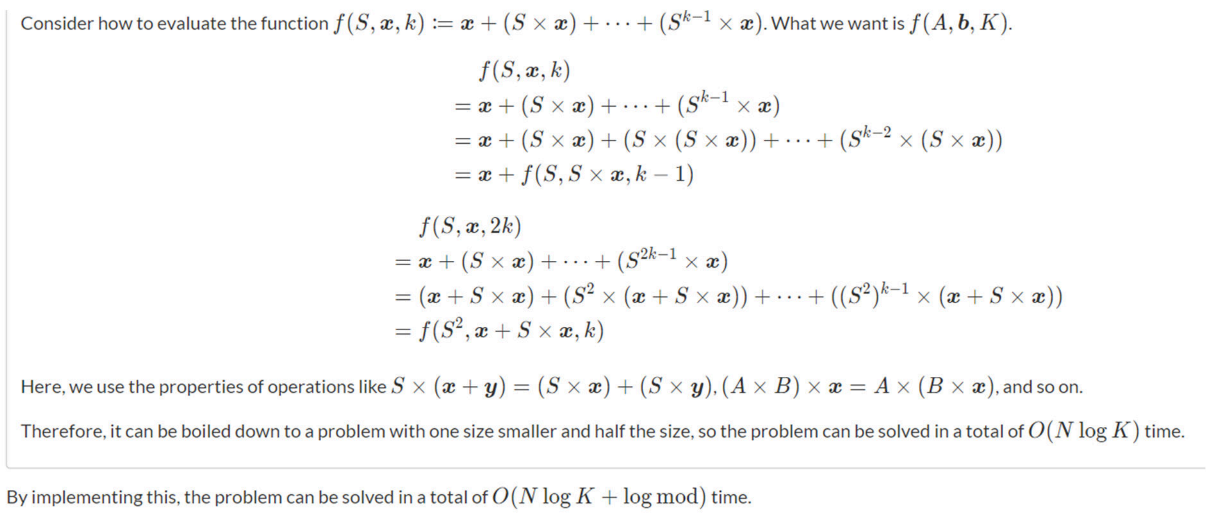
The first operation find the sum of two states of balls. The second operation represents "the state of balls right after passing balls as $i\to S_i$ when the $i$-th Takahashi has $x_i$ balls." The third operation represents "after passing balls as $i\to T_i$ and then $i\to S_i$, the $i$-th Takahashi's balls go to which Takahashi?" The fourth operation represents "after passing balls as $i\to S_i$ $n$ times, the $i$-th Takahashi's balls go to which Takahashi?"

The first three operations can be computed in a total of $O(N)$ time.

Using these operations, the answer can be expressed as follows:

$$\boldsymbol{b}+(A^1\times\boldsymbol{b})+\cdots+(A^{K-1}\times\boldsymbol{b})$$

Here, $A$ is $(A_i)$ given in the input, and $\boldsymbol{b}=(b_i)_i$ it the number of balls that Takahashi has after one operation.



Consider how to evaluate the function $f(S, \boldsymbol{x}, k) := \boldsymbol{x} + (S \times \boldsymbol{x}) + \cdots + (S^{k-1} \times \boldsymbol{x})$. What we want is $f(A, \boldsymbol{b}, K)$.

$$\begin{aligned} & f(S, \boldsymbol{x}, k) \\ &= \boldsymbol{x} + (S \times \boldsymbol{x}) + \cdots + (S^{k-1} \times \boldsymbol{x}) \\ &= \boldsymbol{x} + (S \times \boldsymbol{x}) + (S \times (S \times \boldsymbol{x})) + \cdots + (S^{k-2} \times (S \times \boldsymbol{x})) \\ &= \boldsymbol{x} + f(S, S \times \boldsymbol{x}, k-1) \end{aligned}$$

$$\begin{aligned} & f(S, \boldsymbol{x}, 2k) \\ &= \boldsymbol{x} + (S \times \boldsymbol{x}) + \cdots + (S^{2k-1} \times \boldsymbol{x}) \\ &= (\boldsymbol{x} + S \times \boldsymbol{x}) + (S^2 \times (\boldsymbol{x} + S \times \boldsymbol{x})) + \cdots + ((S^2)^{k-1} \times (\boldsymbol{x} + S \times \boldsymbol{x})) \\ &= f(S^2, \boldsymbol{x} + S \times \boldsymbol{x}, k) \end{aligned}$$

Here, we use the properties of operations like $S \times (\boldsymbol{x} + \boldsymbol{y}) = (S \times \boldsymbol{x}) + (S \times \boldsymbol{y})$, $(A \times B) \times \boldsymbol{x} = A \times (B \times \boldsymbol{x})$, and so on.

Therefore, it can be boiled down to a problem with one size smaller and half the size, so the problem can be solved in a total of $O(N \log K)$ time.

By implementing this, the problem can be solved in a total of $O(N \log K + \log \mathrm{mod})$ time.

**Algorithm:**

With this we can efficiently solves the "Takahashi And Pass-The-Ball Game" problem using the doubling technique and modular arithmetic. The problem requires finding the number of balls each player will have after performing the pass-the-ball operation K times.

Here's the step-by-step algorithm for the given solution:

Step 1: Read input values

1.1. Read the integers `N` and `K` from the input.

- `N` represents the number of players.
- `K` represents the number of times the pass-the-ball operation is performed.

Step 2: Precompute K_inv (K inverse modulo 998244353)

2.1. Compute `K_inv` using the `atcoder::static_modint` library provided by AtCoder. This step is necessary for dividing the answer by K later.

Step 3: Read player information

3.1. Create a vector called `A` of size `N`, which will store the target player to whom each player will pass the ball.

3.2. Create a vector called `ball` of size `N`, which will store the number of balls each player currently has.

Step 4: Perform the pass-the-ball operation once

4.1. Use the `apply` function to perform the pass-the-ball operation once based on the player information in vector `A`.

Step 5: Implement doubling technique

5.1. Use the `while` loop to perform the doubling technique to compute the final result efficiently.

5.2. Within the loop, if K is odd, consider the first operation and the remaining operations separately.

5.3. Put the two operations together using the `add` function and apply the current operation to the `ball` vector.

5.4. Update the `A` vector using the `compose` function to represent the composition of operations i → B[i] → A[B[i]].

Step 6: Divide by K and print the result

6.1. After the loop, multiply the result vector `ans` by `K_inv` to find the answer.

6.2. Print each element of the `ans` vector separated by spaces to represent the number of balls each player has after K operations.

Step 7: End of the algorithm

7.1. Exit the program.

The final C++ code using the algorithm above:

```
#include <bits/extc++.h>

#include <atcoder/modint>


template<int m>

std::ostream& operator<<(std::ostream& os, const atcoder::static_modint<m>& mint) {

   os << mint.val();

   return os;

}


int main() {

   using namespace std;
   using modint = atcoder::static_modint<998244353>;

   unsigned long N, K;
   cin >> N >> K;
   const auto K_inv{modint{K}.inv()};

   vector<unsigned long> A(N);
   for (auto&& a : A) {
      cin >> a;
      --a;
   }

   vector<modint> ball(N);
   for (auto&& b : ball) {
      int x;
      cin >> x;
      b = modint::raw(x);
   }

   // Composition of operations i → B[i] → A[B[i]]
   const auto compose{[N](auto&& A, const auto& B) {
      vector<unsigned long> result(N);
      for (unsigned long i{}; i < N; ++i) result[i] = A[B[i]];
      A = result;
   }};

   // Move the balls
   const auto apply{[N](const auto& A, const auto& x) {
      vector<modint> result(N);
      for (unsigned long i{}; i < N; ++i) result[A[i]] += x[i];
      return result;
```

```
    }};

    const auto add{[N](auto&& x, const auto& y) {
        for (unsigned long i{}; i < N; ++i) x[i] += y[i];
    }};

    ball = apply(A, ball); // Perform the operation once

    vector<modint> ans(N);

    while (K) {
        if (K & 1) { // If K is odd, consider the first and the remaining
            add(ans, ball);
            ball = apply(A, ball);
        }

        // Put the two together
        add(ball, apply(A, ball));
        compose(A, A);
        K /= 2;
    }

    // Multiply by 1/K to find the answer
    for (const auto& x : ans) cout << x * K_inv << " ";
    cout << endl;

    return 0;
}
```

This algorithm efficiently solves the problem using the doubling technique and modular arithmetic, making it suitable for large values of N and K. The time complexity is O(NlogK + logmod), where mod is the modular value (998244353 in this case).

**Note & Credits: Problem statement and Hints from Atcoder.jp by an user en_translator**

**Thank you.**