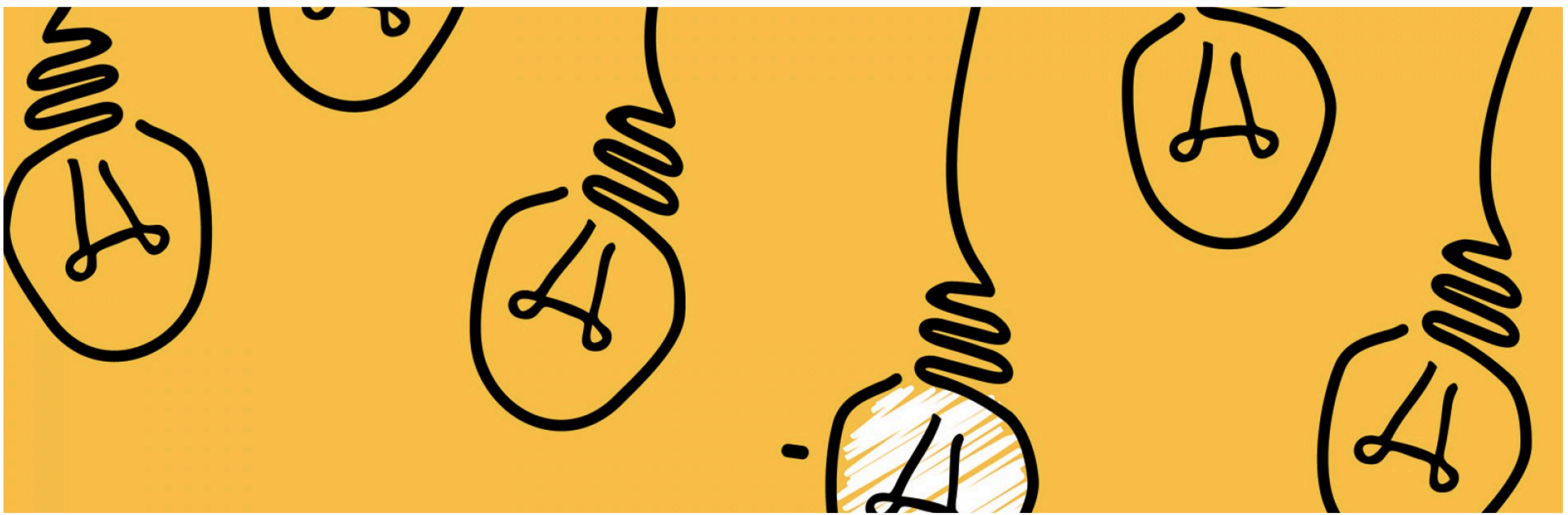


Codeforces

Important Question

Ajay Kumar

Problem

You are given an array A of length N.

Find the **maximum length** of any subarray of A, such that, the bitwise OR of all the elements of the subarray is equal to the bitwise OR of the remaining elements of the array.

Report -1 if no such subarray exists.

Note:

- A subarray is formed by deleting some (possibly zero) elements from the beginning of the array and some (possibly zero) elements from the end of the array.
- The bitwise OR of an empty array is considered to be 0.

Input Format

- The first line of input contains a single integer T, denoting the number of test cases.
- Each test case consists of two lines of input.
  - The first line of each test case contains an integer N — the length of the array A.
  - The second line of each test case contains N space-separated integers $A_1, A_2, \ldots, A_N$ representing the array A.

Output Format

For each test case, output on a new line, the **length** of **longest** subarray such that the bitwise OR of all elements of the subarray is equal to the bitwise OR of the remaining elements of the array, or output -1 if no such subarray exists.

Longest Subarray Problem

This is a very good problem on prefix sum
\\this will take (30N)

Also can be solved using the sliding window algorithm and binary search
Which is nlogn

EXPLANATION:

Let $M = A_1 \mid \ldots \mid A_N$ be the bitwise OR of all the elements.

Let's call subarray $[l, r]$ *valid* if its bitwise OR equals the bitwise OR of the rest of the array.

Note that the bitwise OR of any valid subarray *must* be M.

▶ Why?

To phrase this differently, a subarray is valid if and only if, for each bit b that occurs in M,

- $A[l, r]$ contains an element with the b-th bit set; **and**
- $A[1, l-1] \cup A[r+1, N]$ contains an element with the b-th bit set.

Let's use this fact.
Suppose we fix a position l. Let's try to find the largest r such that $[l, r]$ is valid: if we can do this for every l, the answer is the maximum value of $r - l + 1$ among them all.

Now that l is fixed, let's look at a specific bit b that is present in M.

- If b is present somewhere in the subarray $[1, l-1]$, then the 'outside' is good with respect to this bit.
- Otherwise, we need to ensure that b is present **after** r.
- In particular, if $last_b$ is the last position that contains b, then we must have $r < last_b$.

Each b that is not present in the prefix gives us one such constraint on r. Apply all of them to obtain the maximum possible r we can get if we start at l.

Now, the outside is taken care of, so we need to check the inside.
That is, for each b, we need to check whether it appears in the range $[l, r]$ or not.

Putting everything together, we require the following:

- Given l and b, check whether $[0, l-1]$ contains b
- Given l, r, b, check whether $[l, r]$ contains b
- Given b, find $last_b$, the last index at which b appears

The $last$ array in the third part can be precomputed by iterating across the array and just looking at each bit.

The first and second parts can be solved with the help of prefix sums.
Let $P(i, b)$ denote the number of times b appears in $[1, i]$. Note that if we knew every $P(i, b)$, then:

- Checking whether b lies in $[0, i-1]$ is the same as checking whether $P(i-1, b) > 0$
- Checking whether b lies in $[l, r]$ is the same as checking whether $P(r, b) - P(l-1, b) > 0$

$P(i, b)$ can be calculated as follows:

- If A_i contains the bit b, then $P(i, b) = P(i-1, b) + 1$
- Otherwise, $P(i, b) = P(i-1, b)$

This allows us to compute every $P(i, b)$ in $\mathcal{O}(30N)$ time, and with this the rest of the problem can also be solved in $\mathcal{O}(30N)$ as described above.

C. Count Binary Strings

time limit per test: 2 seconds

memory limit per test: 512 megabytes

input: standard input

output: standard output

You are given an integer n. You have to calculate the number of binary (consisting of characters `0` and/or `1`) strings s meeting the following constraints.

For every pair of integers (i, j) such that $1 \le i \le j \le n$, an integer $a_{i,j}$ is given. It imposes the following constraint on the string $s_i s_{i+1} s_{i+2} \dots s_j$:

- if $a_{i,j} = 1$, all characters in $s_i s_{i+1} s_{i+2} \dots s_j$ should be the same;
- if $a_{i,j} = 2$, there should be at least two different characters in $s_i s_{i+1} s_{i+2} \dots s_j$;
- if $a_{i,j} = 0$, there are no additional constraints on the string $s_i s_{i+1} s_{i+2} \dots s_j$.

Count the number of binary strings s of length n meeting the aforementioned constraints. Since the answer can be large, print it modulo 998244353.

Input

The first line contains one integer n ($2 \le n \le 100$).

Then n lines follow. The i-th of them contains $n - i + 1$ integers $a_{i,i}, a_{i,i+1}, a_{i,i+2}, \dots, a_{i,n}$ ($0 \le a_{i,j} \le 2$).

Output

Print one integer — the number of strings meeting the constraints, taken modulo 998244353.

In the first example, the strings meeting the constraints are `001`, `010`, `011`, `100`, `101`, `110`.

Examples

input

Copy

```
3
1 0 2
1 0
1
```

output

Copy

```
6
```

- You cannot delete the element of map during the map traversal
- if you have to delete the element of the map do this Make an array of no which has to be deleted from the map.Then what you have to do is traverse the vector and serially delete the element

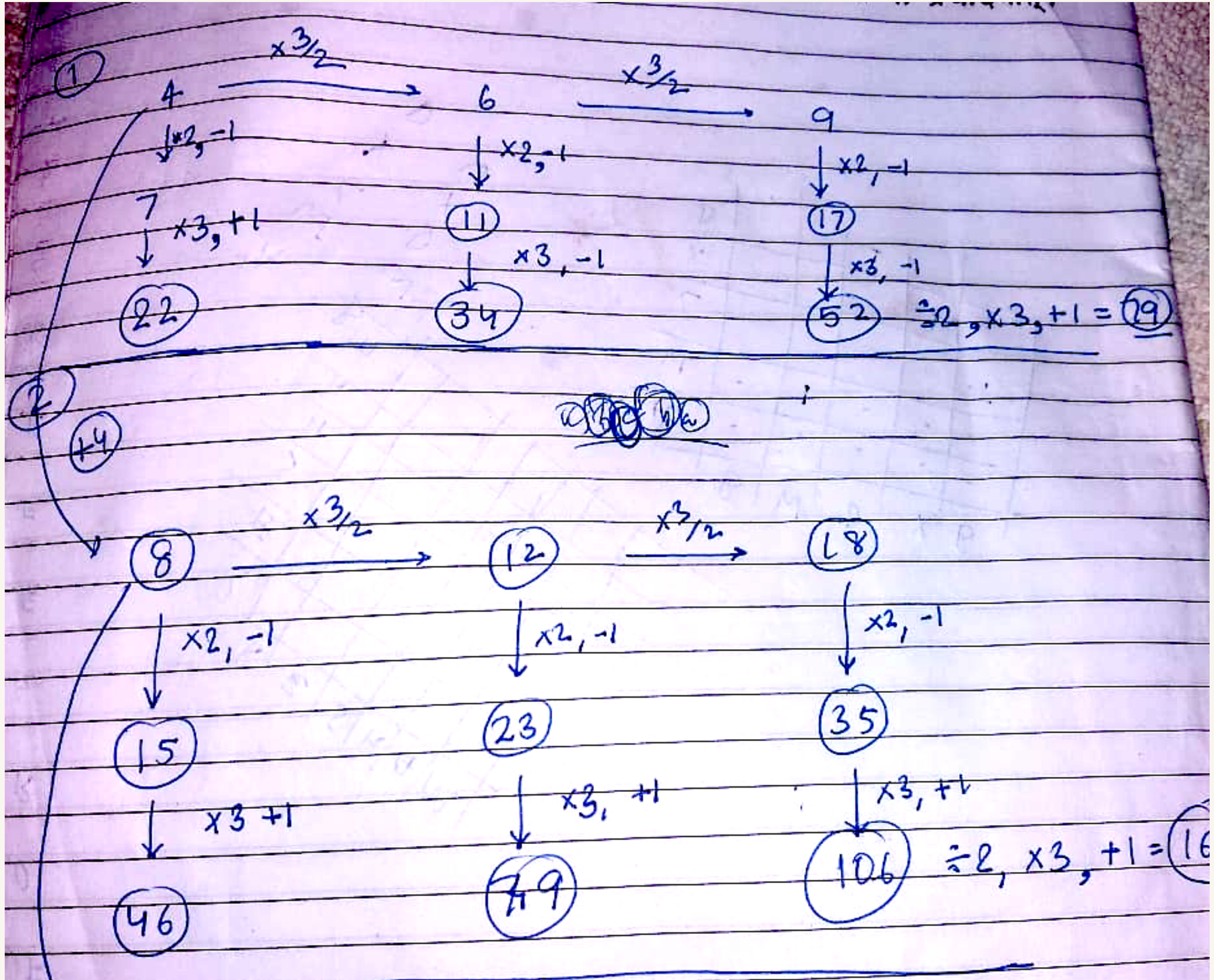
```
Sort(all(v));
vector<int>::iterator ip;
ip=unique(all(v));
v.resize(distance(v.begin(),ip));
```

```cpp
#include <bits/stdc++.h>
using namespace std;

const int mod = 998244353;

void Saraff(){
    int n;
    cin >> n;

    vector <int> dp(n + 1); // dp[i] = number of valid binary strings of length i
    vector <vector <int>> rule(n + 1, vector <int>(n + 1));

    // rule[i][j] == 0, no condition
    // rule[i][j] == 1, all same
    // rule [i][j] == 2, atleast 2 different

    int ok = 1;

    for(int i=1; i<=n; i++){
        for(int j=i; j<=n; j++){
            cin >> rule[i][j];

            if(rule[i][i] == 2){
                ok = 0;
            }
        }
    }

    if(ok == 0){
        cout << "0\n";
        return;
    }

    dp[0] = 2
```

```
for(int i=1; i<=n; i++){
        for(int j=1; j<=i; j++){
            // check [j, i] has all the elements equal
            int ok = 1;
            for(int x=j; x<=i && ok == 1; x++){
                for(int y=x; y<=i && ok == 1; y++){
                    if(rule[x][y] == 2){
                        ok = 0;
                    }
                }
            }

            for(int x=1; x<j && ok == 1; x++){
                for(int y=j; y<=i && ok == 1; y++){
                    if(rule[x][y] == 1){
                        ok = 0;
                    }
                }
            }

            if(ok == 1){
                dp[i] += dp[j - 1];
                dp[i] %= mod;
            }
        }
    }

    cout << dp[n] << "\n";
}
```

- Three friends are on travel one night they make some littis and slept. In night one friend wake up and he divide littis into three equal part with one litti extral so he gave to the dog and eat his one part and sleep. and after some time another friend wake up and he also divide the littis into three equal part and one litti left so he gave to dog and he eat his one of three part & slept, again the third friend also wake up and he does the same. Again in the morning the all three friend wake up and divide into three part but one left so he eat thier part and give one to the dog.Now no litti is left. Guess the no of littis

- Try to solve this brainstorming puzzle, my grandfather has asked to me


1
4
×3/2
6
×3/2
9
×2, -1
×2, -1
×2, -1
7
11
17
×3, +1
×3, -1
×3, -1
22
34
52
÷2, ×3, +1 = 29
2
+4
8
×3/2
12
×3/2
18
×2, -1
×2, -1
×2, -1
15
23
35
×3 +1
×3, +1
×3, +1
46
79
106
÷2, ×3, +1 = 16

M. Sleeping Schedule

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Vova had a pretty weird sleeping schedule. There are h hours in a day. Vova will sleep exactly n times. The i-th time he will sleep exactly after a_i hours from the time he woke up. You can assume that Vova woke up exactly at the beginning of this story (the initial time is 0). Each time Vova sleeps **exactly one day** (in other words, h hours).

Vova thinks that the i-th sleeping time is **good** if he starts to sleep between hours l and r inclusive.

Vova can control himself and before the i-th time can choose between two options: go to sleep after a_i hours or after $a_i - 1$ hours.

Your task is to say the maximum number of **good** sleeping times Vova can obtain if he acts optimally.

Input

The first line of the input contains four integers n, h, l and r ($1 \le n \le 2000, 3 \le h \le 2000, 0 \le l \le r < h$) — the number of times Vova goes to sleep, the number of hours in a day and the segment of the **good** sleeping time.

The second line of the input contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i < h$), where a_i is the number of hours after which Vova goes to sleep the i-th time.

Output

Print one integer — the maximum number of **good** sleeping times Vova can obtain if he acts optimally.

- // recursive code for this question is

```
//recursive code
t=time
dp[I][t]=good(t)+max(fun(i+1,t+a[i]),fun(i+1,t+a[i]-1));
// we can write down the memoised code easily
```

Here we can see the ith state depend upon the i+1 th state so its hard to fill the table
because previously we are filling the table based on previously calculated value
but in this case opposite is there or
May be I have first observed during solving this type of question so
But see how beautifully ones the calculated the value.

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define ar array

const int mxN=2e3;
int n, h, l, r, a[mxN], dp[mxN+1][mxN];

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    cin >> n >> h >> l >> r;
    memset(dp, 0xc0, sizeof(dp));
    dp[0][0]=0;
    for(int i=0; i<n; ++i) {
        cin >> a[i];
        for(int j=0; j<h; ++j) {
            int nj=(j+a[i]-1)%h;
            dp[i+1][nj]=max(dp[i][j]+(l<=nj&&nj<=r), dp[i+1][nj]);
            nj=(j+a[i])%h;
            dp[i+1][nj]=max(dp[i][j]+(l<=nj&&nj<=r), dp[i+1][nj]);
        }
    }
    int ans=0;
    for(int i=0; i<h; ++i)
        ans=max(dp[n][i], ans);
    cout << ans;
}
```

D. Different Arrays

time limit per test: 2 seconds
memory limit per test: 512 megabytes
input: standard input
output: standard output

You are given an array a consisting of n integers.

You **have to** perform the sequence of $n - 2$ operations on this array:

- during the first operation, you either add a_2 to a_1 and subtract a_2 from a_3, or add a_2 to a_3 and subtract a_2 from a_1;
- during the second operation, you either add a_3 to a_2 and subtract a_3 from a_4, or add a_3 to a_4 and subtract a_3 from a_2;
- ...
- during the last operation, you either add a_{n-1} to a_{n-2} and subtract a_{n-1} from a_n, or add a_{n-1} to a_n and subtract a_{n-1} from a_{n-2}.

So, during the i-th operation, you add the value of a_{i+1} to one of its neighbors, and subtract it from the other neighbor.

For example, if you have the array $[1, 2, 3, 4, 5]$, one of the possible sequences of operations is:

- subtract 2 from a_3 and add it to a_1, so the array becomes $[3, 2, 1, 4, 5]$;
- subtract 1 from a_2 and add it to a_4, so the array becomes $[3, 1, 1, 5, 5]$;
- subtract 5 from a_3 and add it to a_5, so the array becomes $[3, 1, -4, 5, 10]$.

So, the resulting array is $[3, 1, -4, 5, 10]$.

An array is *reachable* if it can be obtained by performing the aforementioned sequence of operations on a. You have to calculate the number of reachable arrays, and print it modulo 998244353.

One of the key observations to this problem is that, after the first i operations, the first i elements of the array are fixed and cannot be changed afterwards. Also, after the i-th operation, the elements on positions from $i+3$ to n are the same as they were before applying the operations.

This allows us to write the following dynamic programming: $dp_{i,x,y}$ — the number of different prefixes our array can have, if we have performed i operations, the $(i+1)$-th element is x, and the $(i+2)$-th element is y. The elements after $i+2$ are the same as in the original array, and the elements before $i+1$ won't be changed anymore, so we are interested only in these two elements.

Let's analyze the transitions in this dynamic programming. We apply the operation $i+1$ to the elements a_{i+1}, a_{i+2} and a_{i+3}. If we add a_{i+2} to a_{i+1}, then we subtract it from a_{i+3}, so we transition into state $dp_{i+1,y,a_{i+3}-y}$. Otherwise, we transition into state $dp_{i+1,y,a_{i+3}+y}$. The element we leave behind is either $x-y$ or $x+y$, and if $y \neq 0$, these two transitions give us different prefixes. But if $y=0$, we need to make only one of these transitions, because adding or subtracting 0 actually makes no difference.

Okay, now we've got a solution with dynamic programming in $O(n^3A^2)$, where n is up to 300 and A is up to 300. This is too slow. But we can notice that the value of a_{i+1} actually does not affect our transitions at all; we can just discard it, so our dynamic programming becomes $O(n^2A)$, which easily fits into TL.

Small implementation note: elements can become negative, and in order to store dynamic programming with negative states in an array, we need to do something about that. I don't recommend using maps (neither ordered nor unordered): you either get an extra log factor, or make your solution susceptible to hacking. Instead, let's say that the value of $dp_{i,y}$, where y can be a negative number, will be stored as $dp[i][y+M]$ in the array, where M is some constant which is greater than the maximum possible $|y|$ (for example, 10^5 in this problem). That way, all array indices will be non-negative.

Solution complexity: $O(n^2A)$.

Important Links

- A beautiful technique to solve the xor related Problem codeforces