PREREQUISITES:

Dynamic programming

PROBLEM:

JJ has an array A and an integer X. At most once, he can choose a subsequence of A and add X to all its elements.

What is the maximum possible value of $\sum_{i=2}^{N}(A_i \oplus A_{i-1})$ that he can obtain?

EXPLANATION:

This task can be solved fairly easily with the help of dynamic programming.

Suppose we define a new array dp of length N, where dp_i is the maximum possible answer for the first i elements. Our aim is to compute dp_N.

Each element has two choices for it: it either remains A_i, or becomes $A_i + X$. This information can be encapsulated into the state for dynamic programming by simply holding two values for each index: $dp_{i,1}$ denotes the maximum answer for the first i elements if A_i isn't changed, and $dp_{i,2}$ denotes the maximum answer for the first i elements if A_i is changed to $A_i + X$.

All that remains are the base cases and transitions.

- The base case is $i = 1$, where we have $dp_{1,1} = dp_{1,2} = 0$
- To compute $dp_{i,1}$, we have 2 choices for the previous element: either it changed, or it did not.
  - If it did not change, we get the value $dp_{i-1,1} + (A_i \oplus A_{i-1})$
  - If it did change, we get the value $dp_{i-1,2} + (A_i \oplus (A_{i-1} + X))$
  - $dp_{i,1}$ is the maximum of these two values.
- Similarly, $dp_{i,2}$ is the maximum of $dp_{i-1,1} + ((A_i + X) \oplus A_{i-1})$ and $dp_{i-1,2} + ((A_i + X) \oplus (A_{i-1} + X))$.

These values can be computed iteratively, simply iterating i from 2 to N. The final answer is the maximum of $dp_{N,1}$ and $dp_{N,2}$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Simple observations, dp

PROBLEM:

An array is called alternating if any two consecutive numbers in it have opposite signs (i.e. one of them should be negative, whereas the other should be positive).

You are given an array A of size N consisting of **non-zero** elements. For each index i from 1 to N, you have to find the length of longest subarray starting at i.

QUICK EXPLANATION:

We can observe that if an array is alternating, then all of its subarrays are also alternating.

So, we can divide the array into maximal alternating subarrays.
For doing that, we will iterate over array from left to right, if the sign of current number is different than previous number, then this number can be used to extend previous alternating subarray, Otherwise we will have to start constructing a new maximal alternating subarray.

In this way, we can partition the array into various maximal alternating subarrays.

After this, finding length of longest subarray starting at index i is quite easy, as it can be done easily by finding the position of i in the corresponding maximal alternating subarray. If p be position and L be the length of the maximal subarray, then $L - p + 1$ will be the length of longest subarray starting at index i.

EXPLANATION:

Observation 1:
Values of the number don't matter, only their signs do.

So, we can change the array A such that it consists of -1 and 1's only.

Observation 2:
If an array A is alternating, then all of it's subarrays are alternating.

Let us take an example to understand how we can apply the above observation to solve the problem.
Let $A = [1, 1, -1, 1, 1, -1, -1]$
So, we start from A_1 and note that A_1 is equal A_2.
So the maximal subarray starting from 1 will be $[1]$ itself.

Now, we go to A_2, we can see that A_2, A_3, A_4 have different signs, and A_4 has same sign as A_5.
So the maximal subarray starting from index 2 will be [1, -1, 1].

So, we break the array into several parts such that each part is maximal alternating subarray. In our example, the parts will be
[1] [1, -1, 1] [1, -1], [-1]

We can formalize the above idea to write a pseudo code to break an array into various parts of maximal alternating subarrays.

```
vector<vector<int> > parts;
vector<int> curPart;
subpart.push_back(a[0]);
for (int i = 1; i < n; i++) {
    // If signs of current and previous number are different,
    // then it means that we can extend the current part.
    if (a[i] * a[i - 1] == -1) {
```

```
        curPart.push_back(a[i]);
    } else {
        // We add the curPart into parts.
        parts.push_back(curPart);
    }
}
// Check at the end whether the curPart has non-zero length or not.
// If it has, then add curPart into parts.
if (curPart.size() > 0) {
    parts.push_back(subpart);
}
```

Now, let us find the length of longest alternating subarray ending at each index i for our example subarray A.
We get [1] [3, 2, 1], [2, 1], [1]

So, this means that for an maximal alternating subarray of length L, the answers (length of longest alternating subarray start from that index) will be $L, L-1, \ldots, 1$.

We can use this idea to solve our problem.

```
// i denotes the current index of array at which we currently are.
int i = 1;
for (vector<int> curPart : parts) {
    int L = curPart.size();
    while (L > 0) {
        answer[i] = L;
        // increment i
        i++;
        // decrement L
        L--;
    }
}
// Note the fact that we didn't use the explicit values of curPart, only its size matter.
```

Dynamic programming based Solution

You can also solve the problem by a very simple dp.

Let $len[i]$ denote the maximum length of alternating subarray starting at position i.
We can observer that if $a[i]$ and $a[i+1]$ has opposite signs, then $len[i]$ will be 1 more than $len[i+1]$.
Otherwise in the case when they have same sign, then $len[i]$ will be just 1.

```
len[N] = 1;
for (int i = N - 1; i >= 1; i--) {
    // a[i] and a[i + 1] have different signs.
    // Note that the a[i] can go upto 10^9,
    // So if a is stored in int data type, then the a[i] * a[i + 1] might not fit in int.
    // So, we cast it to long long
    if (a[i] * (long long) a[i + 1]< 0) {
        len[i] = len[i + 1] + 1;
    } else {
        len[i] = 1;
    }
}
```

Time Complexity:

As we have seen in both the solutions we have to iterate over the array A only once or constant number of times. So, time complexity of the algorithm will be $\mathrm{O}(N)$ which will easily fit in time with $N = 10^5$ and $T = 10$.

Pre-requisites:

Dynamic Programming

Problem:

Given a sequences of **N** positive numbers **A[0], A[1], …, A[N-1]**, find how many subsequences of the integers are possible, for which the following procedure terminates in all integers equal to 1. The procedure is to take two unequal numbers, and replace the larger with the difference of the two. The procedure ends when all numbers become equal.

Explanation:

Given a particular sequence of integers, the final number you end up with is the GCD of the numbers. This is because, at each step, the GCD remains invariant, and when all numbers are equal, their GCD will be the number itself.

The question then reduces to finding the number of subsequences of the sequence A, whose gcd is 1.

For this, we go through the numbers A[0], A[1], …, in order, and at each step make a choice as to whether to pick it, or not. We then have to also ask if at the end, the GCD will be 1, and so we recursively pass on the information of our current gcd.

Thus, we can define the function:

f(pos, current_gcd) = f(pos+1, current_gcd) + f(pos+1, gcd(current_gcd, A[pos]))

with the base cases as

f(N, 1) = 1, f(N, k) = 0 for k > 1,

f(pos, 1) = 2^(N-pos) to speed up calculation.

Finally, the answer we require is sum(pos = 0 to N-1) f(pos+1, A[pos]).

The above procedure needs to be memoized in order to work in time. Since A[i] <= 10^4, we get that current_gcd <= 10^4. In particular, the current_gcd will be a divisor of some A[i], hence the total number of states will be atmost N * (total number of divisors of A[i]). This is small enough to work in time, (certainly less than N * 10^4 per testcase).

PREREQUISITES:

Trees, Dynamic Programming

PROBLEM:

Chef is given a rooted tree with N nodes numbered 1 to N where 1 is the root node.
Each node is assigned a lowercase english alphabet between a to z (both included).

Given two nodes A and B:

- Let $LCA(A, B)$ denote the **lowest common ancestor** of nodes A and B.
- Let $DIS(A, B)$ denote the string containing all the characters associated with the nodes in the same **order** as the shortest path from node A to node B (**excluding** the character associated to node B). Note that $DIS(A, A)$ is an **empty** string.

Chef needs to answer Q queries. Each query is of the type:

- $U\ V$: Given two integers U and V $(1 \le U, V \le N)$, determine if there exist a concatenation of a non-empty **subsequence** of $DIS(U, LCA(U, V))$ and a non-empty subsequence of $DIS(V, LCA(U, V))$ such that it is a **palindrome**.

EXPLANATION:

The first thing that needs to be done is to pre-calculate lowest common ancestors for all nodes using binary lifting so that we can find the lowest common ancestor for any two nodes in $log(N)$ time. More details regarding that can be found **here**

Along with that, we also define another variable $cnt[node][x]$ that stores frequency of character x from $root$ till $node$. This can be again done using DFS.

Once we are done with both these preprocessing, we can solve for each query in $log(N)$ time. Say for a query we are given U and V, we can calculate their lowest common ancestor from the algorithm above. Let that be lca. Now for concatenation of a particular subsequence of $DIS(U, lca)$ and $DIS(V, lca)$ to be a palindrome they both need to have at least one character common, i.e $DIS(U, lca)$ and $DIS(V, lca)$ should have one character in common say a since then our final string can be aa which is a palindrome.
Thus for each query we loop through all characters from a to z and say for a particular character j if

$$cnt[U][j] - cnt[lca][j] > 0\ AND\ cnt[V][j] - cnt[lca][j] > 0$$

then that implies they both have character j common and so we can form a palindrome string.

TIME COMPLEXITY:

$O(Qlog(N))$, for each test case.

PREREQUISITES:

Dynamic programming, prefix sums, basic probability

PROBLEM:

You're given an array A and a string S which contains only L and R.
For each element of the string in order:

- If the array has size 1, do nothing.
- Randomly split A into two non-empty subarrays, each split being equiprobable.
- If S_i is L, discard the right part; otherwise discard the left part.

Find the expected sum of the final array.

EXPLANATION:

Every move, we cut out either a prefix or a suffix of the remaining array.
That means in the end, we'll be left with some subarray of A.
In fact, at any stage of the process we'll have a subarray of A.

Let $p(L, R)$ be the probability that we're left with subarray $[A_L, A_{L+1}, \ldots, A_R]$ after all the moves.
Then, by definition of expectation, the final answer is just $\sum P(L, R) \cdot (A_L + A_{L+1} + \ldots + A_R)$ across all $1 \le L \le R \le N$.

So, it's enough for us to find all the $p(L, R)$ values.

The remaining subarray changes after each move, so we also need information about which move we're on.
Let $p(L, R, k)$ be the probability that after k moves, we have subarray $A[L : R]$ with us.
Initially, we have $p(1, N, 0) = 1$ and $p(L, R, 0) = 0$ for all other (L, R).

For $k \ge 1$, let's look at all possible moves.

- Suppose $S_k = \mathrm{L}$, meaning we must keep the left subarray.
  Then, subarray $[L, R]$ can be obtained by cutting $[L, x]$ for some $R < x$. So,

$$p(L, R, k) = \sum_{x=R+1}^{N} p(L, x, k-1) \cdot \frac{1}{x - L}$$

- If $L = R$, we'll also have an extra $p(L, L, k-1)$ term to account for size-1 subarrays not being splittable.
- If $S_k = \mathrm{R}$, transitions are similar: consider all subarrays $[x, R]$ for $1 \le x < L$.

Notice that we have $\mathrm{O}(N^2 K)$ states, and $\mathrm{O}(N)$ transitions for each one.
Memoizing states thus allows for a $\mathrm{O}(N^3 K)$ dynamic programming solution, which is enough to pass subtask 1.

To optimize this solution further, it seems our only hope is to make transitions faster: all the states seem necessary.

One way to do this is to look at transitions from the opposite side.
Instead of looking at $p(L, R, k)$ in terms of sums of $p(L, x, k-1)$, let's look at how $p(L, R, k)$ *contributes* to $p(L, x, k+1)$

So, suppose $[L, R]$ is fixed, and $S_{k+1} = \mathrm{L}$, i.e, we keep the left part of the subarray.
There's a $\frac{1}{R-L}$ chance of obtaining each of $[L, L], [L, L+1], \ldots, [L, R-1]$.
So, we add $p(L, R, k) \cdot \frac{1}{R-L}$ to each of $p(L, L, k+1), p(L, L+1, k+1), \ldots, p(L, R-1, k+1)$.

Notice that L is fixed for all of these, so we're essentially just adding a constant value to a range!

Processing Q range-add updates on an array of length N can be done in $\mathrm{O}(N+Q)$ time using prefix sums.

How?

Suppose we want to process Q updates on the array B.

Create an auxiliary array C of range N, initially filled with zeros.
Then, for each update (L, R, x) (meaning we want to add x to the range $[L, R]$ of B),

- Increase C_L by x
- Decrease C_{R+1} by x

Finally, take the prefix sums of array C.
The i-th prefix sum of C is the amount by which B_i will change.

A slightly more detailed explanation of why this works can be found in the "How?" section **here** .

In our case we perform one update for each (L, R), for about N^2 updates in total.
So, for each move, we're now down to $\mathrm{O}(N^2)$ operations in total!

This optimization makes the solution $\mathrm{O}(N^2K)$.

Note that once again, a little care is needed when dealing with $L = R$.

TIME COMPLEXITY

$\mathrm{O}(KN^2)$ per testcase.

PROBLEM:

You're given an array A of N integers. You need to find the minimum cost of creating another array B of N integers with the following properties

- $B_i \geq 0$ for each $1 \leq i \leq N$
- The GCD of adjacent elements of B is equal to 1, i.e, $\gcd(B_i, B_{i+1}) = 1$ for each $1 \leq i < N$

The cost of creating B is defined as follows:

$$\sum_{i=1}^{N} 2^{|A_i - B_i|}$$

Find the **minimum** possible cost to create the array B. Since the answer can be huge print it modulo $10^9 + 7$

Note: You need to minimize the value of total cost of creating the array B, and then print this minimum value modulo $10^9 + 7$. For example, suppose there is a way to create the required B with a cost of 500, and another way to do it with a cost of $10^9 + 7$ (which is $0 \pmod{10^9 + 7}$). The output in this case would be 500 and not 0.

EXPLANATION:

Observation 1: Let us take k as the $max(|A_i - B_i|),\ 1 \leq i \leq N$. We would note that k is small(less than 20).

Now using the above observation let us solve this problem using DP.
Let us define our dp $dp[i][j]$, where $1 \leq i \leq n$ and $-k \leq j \leq k$.
Here $dp[i][j]$ stores the minimum cost and denotes that we have processed till i_{th} index and the element at the i_{th} index is modified to $(A[i] + j)$. Now let us compute $dp[i][j]$.
Clearly $dp[i][j]$ depends on $dp[i-1][j^{'}],\ where\ -k \leq j^{'} \leq k$. As we are computing $dp[i][j]$, we know that the value of i_{th} index is $A[i] + j$, so $dp[i][j]$ depends on $dp[i-1][j^{'}]$ if and only if $gcd(A[i] + j, A[i-1] + j^{'})$ is 1. In that case:

$$dp[i][j] = min(dp[i][j], dp[i-1][j^{'}] + 2^{|j|})$$

So we compute this dp and our answer would be $min(dp[n][j]), where\ -k \leq j \leq k$

TIME COMPLEXITY:

$O(Nk^2)$, for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM:

Choose K disjoint distinct subarrays each of length L from an array of size N such that their sum is maximum. Also, you are allowed to choose at most C elements twice in a subarray.

EXPLANATION:

Let's choose K subarrays one by one. For choosing xth subarray suppose we already have best possible answer for 0 common element, 1 common element $\ldots C$ common elements for each array from $[0, i]$ for $x - 1$ subarrays. Now if we want to choose xth subarray from $[i + 1, i + L]$ then:

For 0 common elements with subarray $[i + 1, i + L]$ best possible ans is: $max($ ans for x-1th subarray till i such that C elements are common, ans for x-1th subarray till i such that $C - 1$ elements are common…, ans for x-1th subarray till i such that 0 elements are common) $+$ sum of subarray $[i + 1, i + L]$.

Similarly, for 1 common elements with subarray $[i + 1, i + L]$ best possible ans is: $max($ ans for x-1th subarray till $i + 1$ such that $C - 1$ elements are common, ans for x-1th subarray till $i + 1$ such that $C - 2$ elements are common…, ans for $(x - 1)th$ subarray till $i + 1$ such that 0 elements are common) $+$ sum of subarray $[i + 1, i + L]$.

Similarly, we can choose best answer with subarray $[i + 1, i + L]$ for 2 commons,3 commons…C commons.

Now, in the same way, we can find best solution for K subarrays.

For formally,
$Dp[Nth][last_index_of_subarray][Cth_common] = max(Dp[(N - 1)th][last_index_of_subarray - L + Cth_common][C - Cth_common] + sum_of_subarray[last_index_of_subarray - L + 1, last_index_of_subarray], X)$

where,

- $Dp[i][j][k]$ denotes best answer for choosing ith subarray($0 \le i \le K$) till jth index and for k commons.
- $sum_of_subarray[i,j]$ is the sum of subarray in range $[i,j]$ $(0 \le i \le j \le N - 1)$
- $X = max(Dp[Nth][last_index_of_subarray - 1][Cth_common], Dp[Nth][last_index_of_subarray][Cth_common - 1])$

TIME COMPLEXITY:

$O(N * K * C)$

PREREQUISITES:

Dynamic programming

PROBLEM:

Given a binary string S, you can replace 01 with a, 10 with b, 010 with ab, and 101 with ba.

Given a string of a's and b's, how many initial strings could possibly have created this string?

EXPLANATION:

We will use dynamic programming to find the answer.
Let A denote the input string.

Let dp_i denote the number of binary strings such that after replacement, they can form the first i characters of A. The answer we want is, of course, dp_N.

As a base case, we have $dp_0 = 1$, since we can only form the empty string from the empty string. Now, for $i \geq 1$:

- Suppose $A_i = a$. Then, we have two choices for a binary string that forms $A_1 A_2 \ldots A_i$
  - If its last two characters were 01, we could replace them with a, then form the other $i - 1$ characters from the remaining part of the string. There are dp_{i-1} ways to form the rest of the string.
  - If its last three characters were 101 and $A_{i-1} = b$, we could replace 101 with ba and then form the remaining $i - 2$ characters in dp_{i-2} ways. Note that this is only applicable when $i \geq 2$.
  - There is no other way to obtain a string ending with a.
- The analysis for $A_i = b$ will give you the exact same transitions.

So, our dp is as follows:

- $dp_0 = 1$
- For $i \geq 1$, $dp_i = dp_{i-1}$
- Further, if $i \geq 2$ and $A_i = A_{i-1}$, add dp_{i-2} to dp_i.

Finally, print dp_N.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Dynamic programming, expected value, the inclusion-exclusion principle

PROBLEM:

You're given an integer N.
For an integer $1 \le x \le N$, consider the following process:

- Pick a random integer y between 1 and N
- Replace x by $\gcd(x, y)$.

Let $\mathrm{E}[x]$ be the expected number of moves this process takes till x reaches 1.
For each $1 \le x \le N$, compute $\mathrm{E}[x]$.

EXPLANATION:

Writing out the expected value for a single x, we see that:

$$\mathrm{E}[x] = \sum_{y=1}^{N} \frac{1}{N} \cdot (1 + \mathrm{E}[\gcd(x, y)]) = 1 + \frac{1}{N} \sum_{y=1}^{N} \mathrm{E}[\gcd(x, y)]$$

This lends itself to a rather natural (if slow) recursive approach, because $\gcd(x, y) \le x$.
The only issue is those y for which $\gcd(x, y) = x$, since they lead to 'loops'.

However, such y lead to an $\mathrm{E}[x]$ term anyway, so they can be taken care of with a bit of algebra.
Suppose there are k such y. Then,

$$\mathrm{E}[x] = 1 + \frac{k}{N}\mathrm{E}[x] + \frac{1}{N} \sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$$
$$\mathrm{E}[x] \cdot (1 - \frac{k}{N}) = 1 + \frac{1}{N} \sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$$

The right-hand side here now depends only on values strictly less than x, so a dynamic programing approach can compute it.
Once the RHS is known, $\mathrm{E}[x]$ is easily found.
However, it's still $\mathrm{O}(N^2)$, so we need to speed it up.

Let's look at the expression we want to compute.
The expression we want to compute is $\sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$.

Let's fix $g = \gcd(x, y)$ and try to find the number of $1 \le y \le N$ such that $\gcd(x, y) = g$.
If we know this, we can add this count multiplied by $\mathrm{E}[\gcd(x, y)]$ to the overall sum.
g has to be a divisor of x, which greatly limits the number of g we need to consider.

Writing $x = g \cdot s$ and $y = g \cdot t$, we have $\gcd(g \cdot s, g \cdot t) = g$, which means $\gcd(s, t) = 1$.
Further, $g \cdot t \le N$, meaning $t \le \frac{N}{g}$

So, we really just want to compute the number of integers in the range $[1, \frac{N}{g}]$ that are coprime to $s = \frac{x}{g}$.

That can be done using inclusion-exclusion on the prime factors of s.

Details

Let $M = \frac{N}{g}$ be our upper bound.
Let the distinct prime factors of s be $p_1, p_2, \ldots, p_k$.

Then, we have the following:

- Initially, pretend everything in $[1, M]$ is coprime to s.
- Anything not coprime to s must have some p_i as a factor.
  So, subtract all multiples of $p_1, p_2, \ldots, p_k$ from the count.
  These are $\lfloor \frac{M}{p_1} \rfloor, \lfloor \frac{M}{p_2} \rfloor, \ldots, \lfloor \frac{M}{p_k} \rfloor$ respectively.
- However, now multiples of, for example, $p_1 p_2$ were counted twice.
  So, for each pairwise product of primes, add back in the count of its multiples.
- Then, you'll need to subtract the count of multiples of all triplet products, and so on.

In general, we go through each of the 2^k possibilities of products of prime factors, and either add or subtract the count of its multiples depending on how many primes are involved in the product.

A slightly more detailed explanation can be found **here**.

Notice that if s has k distinct prime factors, this approach takes 2^k or $2^k \cdot k$ steps, depending on implementation.
Since $N \leq 5 \cdot 10^5$, any integer $\leq N$ has at most 6 distinct prime factors, so $k \leq 6$ here, which is pretty small.

We do the 2^k operation for each pair of (number, divisor) below N.
It's well-known that there are $\mathrm{O}(N \log N)$ such pairs, so a simple upper bound on complexity is $\mathrm{O}(2^6 \cdot N \log N)$.

However, in practice most integers have far fewer than 6 prime factors, so the constant factor is a lot better and your code should run fast.
In particular, even with a $2^k \cdot k$ algorithm that goes through bitmasks, the number of operations is about 10^8 which is not an issue for C++.

A 2^k algorithm for each pair uses less than $4 \cdot 10^7$ operations.

TIME COMPLEXITY

$\mathrm{O}(2^6 \cdot N \log N)$ per testcase, but with fairly low constant factor.

PREREQUISITES:

DP, **Longest Common Subsequence**

PROBLEM:

Given two arrays of pairwise distinct strings find the maximum numbers of strings remaining after deleting some (or possibly none) strings from either of the arrays.

QUICK EXPLANATION:

Common and in same order hints for LCS, it's a direct question of longest common subsequence but it fails for the n^2 approach if n>1000 so for n being around 10^5 we need to find an efficient algorithm.

EXPLANATION:

This can be converted to a lis(longest increasing subsequence) question by maintaining an array which contains the index of the common elements in one array and then find its lis to find the answers. As the strings in an array are pairwise distinct we can use binary search for finding the index of common strings. Hence reducing the overall complexity to O(n*log(n)). This can be either implemented through vector of pair of map.

PREREQUISITES:

Dynamic programming

PROBLEM:

You are given an array A of size N.

A partitioning of the array A is the splitting of A into one or more non-empty contiguous subarrays such that each element of A belongs to exactly one of these subarrays.

Find the number of ways to partition A such that the parity of the sum of elements within the subarrays is alternating. In other words, if S_i denotes the sum of the elements in the i-th subarray, then either

- S_1 is odd, S_2 is even, S_3 is odd and so on.
- or S_1 is even, S_2 is odd, S_3 is even and so on.

For example if $A = [1, 2, 3, 3, 5]$. One way to partition A is $[1, 2][3, 3][5]$. Another way to partition A is $[1][2][3][3, 5]$. Note that there exists more ways to partition this array.

Since the answer may be large, output it modulo 998244353.

QUICK EXPLANATION

Let $dp[i][parity]$ represent the number of valid partitions of Prefix P_i such that the last subarray chosen has parity of the sum as $parity$. Let $sum[i]$ represent the sum of first i elements of the A.
Initialize $dp[0][0] = 1$ and $dp[0][1] = 1$

Case 1: $sum[i]$ **is even.**

- $dp[i][0] = \sum_{\substack{j=0 \\ sum[j] | 2}}^{i-1} dp[j][1]$
- $dp[i][1] = \sum_{\substack{j=0 \\ sum[j] \nmid 2}}^{i-1} dp[j][0]$

Case 2: $sum[i]$ **is odd.**

- $dp[i][1] = \sum_{\substack{j=0 \\ sum[j] | 2}}^{i-1} dp[j][0]$
- $dp[i][0] = \sum_{\substack{j=0 \\ sum[j] \nmid 2}}^{i-1} dp[j][1]$

EXPLANATION:

Let us calculate the number of valid partitions of A when a fixed prefix P_i (array formed by first i elements of A) is the first subarray of the partition. Let the sum of elements of P_i be even. Then the problem is reduced to finding the number of valid partitions of array B formed by the remaining $N - i$ elements (other than P_i) of array A without changing the order of the elements and starting with the first subarray of the partition having odd sum.

The case when the sum of elements of P_i is odd is similar except now the problem is reduced to finding the number of valid partitions of array B starting with the first subarray of the partition having even sum.

This problem has both optimal substructure property and overlapping subproblems. These kind of problems can be solved using dynamic programming.

Let $dp[i][parity]$ represent the number of valid partitions of Prefix P_i such that the last subarray chosen has parity of the sum as $parity$. Let $sum[i]$ represent the sum of first i elements of the A.
Initialize $dp[0][0] = 1$ and $dp[0][1] = 1$

Case 1: $sum[i]$ **is even.**

- $dp[i][0] = \sum_{\substack{j=1 \\ sum[j] \mid 2}}^{i-1} dp[j][1]$ $i.e.$ sum of all valid partitions till now such that the last chosen subarray of these partitions has odd sum and the sum of all elements in the partition is even.
- $dp[i][1] = \sum_{\substack{j=1 \\ sum[j] \nmid 2}}^{i-1} dp[j][0]$: $i.e.$ sum of all valid partitions till now such that the last chosen subarray of these partitions has even sum and the sum of all elements in the partition is odd.

Case 2: $sum[i]$ **is odd.**

- $dp[i][1] = \sum_{\substack{j=1 \\ sum[j] \mid 2}}^{i-1} dp[j][0]$ $i.e.$ sum of all valid partitions till now such that the last chosen subarray of these partitions has even sum and the sum of all elements in the partition is even.
- $dp[i][0] = \sum_{\substack{j=1 \\ sum[j] \nmid 2}}^{i-1} dp[j][1]$: $i.e.$ sum of all valid partitions till now such that the last chosen subarray of these partitions has odd sum and the sum of all elements in the partition is odd.

These sums can be maintained using just four variables. The answer would be $dp[N][0] + dp[N][1]$ For details of the implementation see the code attached below.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

DP, Bitwise Operations

PROBLEM:

Given an array A of N integers ranging from 0 to 2^{30}, partition this array into K consecutive sub-arrays such that the sum of Bitwise ORs of these sub-arrays is maximized.

Naive Approach

The simplest approach is to try out all possible partitions using recursion and take the maximum. This approach will try out at least $\binom{N}{K-1}$ possibilities. It is clearly inefficient and will time-out for all sub-tasks.

An $\mathrm{O}(N^2K)$ solution:

We can use Dynamic Programming. Let us maintain a 2-D array $DP[][]$ in which $DP[i][j]$ stores the maximum sum that you can get by partitioning the segment $1 \ldots i$ into j sub-segments. Evidently, this is an optimal overlapping sub-structure of the main problem.

To calculate $DP[i][j]$, we can iterate from $p = i - 1 \ldots 1$ and use the relation:

$$DP[i][j] = max(DP[i][j], DP[p][j-1] + BitwiseOR(A[p+1], A[p+2], \ldots, A[i])).$$

Since $N, K \geq 1000$, this algorithm will time out for the last two sub-tasks.

Optimising the DP to $\mathrm{O}(NK\log(A_{max}))$:

We make two observations:

- $DP[p][j] \leq DP[i][j]$ for $p \leq i$. This isn't difficult to infer since Bitwise OR operation on two numbers never results in a number smaller than either of the two.
- Let us consider an array C where $C[i] = BitwiseOR(A[1], A[2], \ldots, A[i])$. For $i > 30$, we claim that there are only 30 distinct values in the array C. The reason is that cumulative Bitwise OR operation accumulates '1' bits. Since the maximum number of '1' bits in the values stored in array C can be 30, thus there can only be 30 distinct values in array C.

Now, using the two aforementioned observations, we can optimize our DP. While calculating $DP[i][1 \ldots K]$, we were iterating from $p = 1 \ldots i-1$ and checking $DP[p][1 \ldots K]$. But we can now say that we only need to iterate from $1 \ldots K$ for those p where $BitwiseOR(A[p], A[p+1], \ldots, A[i])$ and $BitwiseOR(A[p+1], A[p+2], \ldots, A[i])$ are different. Since, there can at max be 30 such p, therefore, the net complexity comes down to $\mathrm{O}(NK\log(A_{max}))$. This will easily finish in under 10 seconds.

An $\mathrm{O}(NK)$ solution :

The $\mathrm{O}(N^2K)$ DP solution can directly be optimised to $\mathrm{O}(NK)$ DP using a widely known optimisation method called "Knuth Optimisation". Though this was not required to solve the problem for 100 points, we suggest that you go through the concept:
Quora
Codeforces

PREREQUISITES:

Basic mathematics, matrix, array.

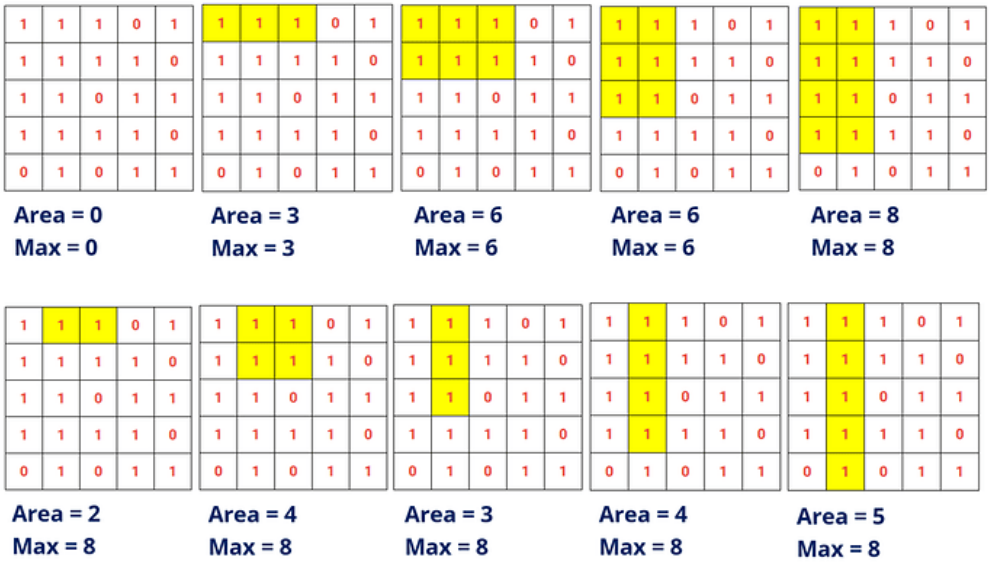
PROBLEM:

For a table in a restaurant which divided by n x m cells, the waiter has randomly placed Ice-cream and gulab jamun on each cell of the table. Find out the largest rectangle which can be formed by Ice-creams only such that there is no gulab jamun within the rectangle. Note: Ice-cream is represented by char values as ‘1’ and gulab jamun by ‘0’. The input is given in the form of n x m Matrix containing zeros and ones.

EXPLANATION

Instead of forming every rectangle, then checking validity of rectangle, we can optimize the brute-force by only considering valid rectangles. For this, we can start from every cell and consider valid rectangle starting from that cell.

- Let the current cell be at `(i, j)`.
- We first consider `i`th row and find maximum column length of 1s starting from `M[i][j]`.
- Then move to `i+1`th row and find maximum column length of 1s starting from `M[i+1][j]`. Take minimum of all lengths and find the area and keep updating max area.
- Continue similar process till you reach last row and then repeat the process for all other cells as well.
- Finally return maximum area found from all valid rectangles.



TIME COMPLEXITY

Time complexity is O((mn)^2)

PREREQUISITES

Simple Math, Repeated Squaring

PROBLEM

You are given three integers **d, m and N** in the following relation

$x^d + y^d + z^d \equiv m$, modulo N

Find the **number of solutions** of the above equation where

$0 \le x, y, z \le U$

EXPLANATION

The first observation to be made is that the value of **N** is very small. We already know that

$a^b = (a \% N)^b$, modulo N

Despite the fact that **U** is a large number, it is only relevant for us to find the solutions **(X, Y, Z)** which satisfy

$0 \le X, Y, Z < N$

and for each solution **(X, Y, Z)**, calculate the number of values in the range

$0 \le x, y, z \le U$

such that

- **x = X, modulo N**
- **y = Y, modulo N**
- **z = Z, modulo N**

Let us say that

- **A[t] = t^d % N**
- **$0 \le t < N$**

Now, we can iterate through the list of unique solutions **(X, Y, Z)**

```
for X = 0 to N-1
        for Y = 0 to N-1
                for Z = 0 to N-1
                        if (A[X] + A[Y] + A[Z]) % N = m
                                process(X, Y, Z)
```

The complexity of calculating **A[]** is **O(N log d)** if we use **repeated squaring**.

In the worst case, the number of times **process()** is called is **O(N^3)**.

For each solution **(X, Y, Z)** we wish add the solutions **(x, y, z)** from the complete range. The number of values of **x**, satisfying

- **x = X, modulo N**
- **$0 \le x \le U$**

is

```
number-of-sols(X) =

floor(U / N) + {
        1, if (U % N > 0 AND X ≤ U % N),
```

```
        0, otherwise
}
```

Now, the number of solutions **(x, y, z)** for **(X, Y, Z)** is simply

```
answer = 0

process(X, Y, Z) =
        answer = answer +
                number-of-sols(X) *
                number-of-sols(Y) *
                number-of-sols(Z)
```

Thus, the answer can be found in **O(N^3)** time.

CODING COMMENTARY

The answer has to be found modulo **$10^{9</sup+7}$**. This leaves a lot of scope for **integer overflow**. Make sure that your implementation uses **64-bit integers** for all intermediate results, at the least.

64-bit integers can handle the overflow of multiplying **two** 32-bit integers, but not **three**. You must find the **modular result after each product**.

PREREQUISITES:

Iterating through submasks, dynamic programming

PROBLEM:

An array A is said to be good if $A_i \mid A_j = (A_i \oplus A_j) + A_j$ for every $i < j$.

Given a permutation P, find the number of its good subsequences.

EXPLANATION:

The condition for an array to be good is a bit weird, so let's analyze it a bit.

It can be seen that the given condition reduces to "A_j is a submask of A_i when both are written in binary".

How?

Let's look at the relation between $A_i \mid A_j$ and $A_i \oplus A_j$, bit by bit.

- If A_i and A_j both have a bit set, $A_i \mid A_j$ has it set but $A_i \oplus A_j$ doesn't.
- If one of A_i and A_j has a bit set and the other doesn't, both $A_i \mid A_j$ and $A_i \oplus A_j$ have it set
- If both A_i and A_j have a bit unset, $A_i \mid A_j$ and $A_i \oplus A_j$ both have it unset.

Together, these should tell you that $A_i \mid A_j = (A_i \oplus A_j) + (A_i \& A_j)$ where $\&$ denotes bitwise AND.

Now, if $A_i \mid A_j = (A_i \oplus A_j) + A_j$, then along with the above equation we know that $A_j = A_i \& A_j$, i.e, A_j is a submask of A_i.

Also notice that the submask relation is transitive, that is, if A_j is a submask of A_i and A_k is a submask of A_j, then A_k is a submask of A_i.
So, we only need to ensure that each pair of adjacent elements in the subsequence satisfies the submask relation: this automatically satisfies it for every pair.

With that in mind, let's move to a full solution for our permutation P.
I'll use the notation $P_i \subseteq P_j$ to denote P_i being a submask of P_j.

Let dp_i denote the number of good subsequences *ending at* position i.
As noted above, we only need to care about adjacent elements, so suppose we fix the previous element of the subsequence as position j. Then we can take any subsequence ending at P_j and append P_i to it to obtain a subsequence ending at position i.

This gives us the relation:

$$dp_i = 1 + \sum_{\substack{1 \le j < i \\ P_j \subseteq P_i}} dp_j$$

(we add 1 to account for the 1-length subsequence $[P_i]$)

This is a solution in $\mathrm{O}(N^2)$, which is too slow for the given constraints.

To speed it up, notice that we're doing some wasteful iteration: we don't really care about all $j < i$, we only care about those j for which $P_i \subseteq P_j$; i.e supermasks of P_i.

So, let's instead iterate across all supermasks of P_i and add their dp values to dp_i.

The complexity of doing this is $O(3^{\log N})$ when summed up across all indices.

How?

Notice that the link in the prerequisites details how to iterate across all *submasks*, while we care about *supermasks*.
Modifying the approach given there to iterate supermasks instead of submasks is not hard and left as an exercise 🙂

TIME COMPLEXITY:

$O(3^{\log N})$ per testcase.

PREREQUISITES:

Depth-First Search, Dynamic Programming on Trees

PROBLEM:

Given a tree with N vertices numbered from 0 to $N-1$ and $N-1$ edges, output an array where the ith element of the array denotes the sum of distances of the vertex i to all other vertices in the tree.

Note : Here the distance between two vertices connected through an edge is 1 units.

QUICK EXPLANATION:

Let, $nodesCount[v]$ be no. of nodes which are in the subtree of node v (including v) and $ans[v]$ be the sum of distances of the node v to the remaining nodes.

Then, $ans[v] = ans[u] + N - 2 * nodesCount[v]$ (where, u is the parent of v)

EXPLANATION:

Throughout the editorial consider u to be parent of v.

Suppose, $down[v]$ denotes sum of distances to all nodes in the subtree of node v and let us define $nodesCount[v]$, $ans[v]$ same as above.

Calculation of $nodesCount[u]$:

$$nodesCount[u] = 1 + \sum_{for\ all\ children\ v\ of\ u} nodesCount[v]$$

Calculation of $down[u]$:

$$down[u] = \sum_{for\ all\ children\ v\ of\ u} (nodesCount[v] + down[v])$$

To get the sum of distances from u to all nodes in the subtree of v we need to add 1 unit extra distance for each node in subtree of v. Since, there are $nodesCount[v]$ many nodes hence, we add this to $down[v]$ to get the sum of distances from u to all nodes in subtree of v and take the sum over all children v of u.

Now, if we calculate $up[v]$ i.e. sum of distances to all nodes from v that are **not** a part of subtree of v then,

$$ans[v] = down[v] + up[v]$$

Calculation of $up[u]$:

$$up[v] = ans[u] - (nodesCount[v] + down[v]) + (N - nodesCount[v])$$

Here, we assume that we already know $ans[u]$.

When we subtract $(nodesCount[v] + down[v])$ from $ans[u]$ we get sum of distances to all nodes from u to the nodes which are **not** in the subtree of v. Now, to make it the sum of distances from v we add the number of nodes which are not in subtree of v (This is similar to calculating $down[u]$ from $down[v]$ by adding $nodesCount[v]$)

Calculation of $ans[u]$:

$ans[v] = down[v] + up[v]$

$\implies ans[v] = down[v] + ans[u] - (nodesCount[v] + down[v]) + (N - nodesCount[v])$

$\implies ans[v] = ans[u] - N - 2 * nodesCount[v]$

We had assumed that we already know $ans[u]$ while calculating $up[v]$. We can incrementally calculate $ans[u]$ and use it for $up[v]$ calculation using another Depth-First Search. Here, the base case is $ans[root] = down[root]$, since for root all nodes lie in its subtree and $up[root] = 0$.

PREREQUISITES:

Dynamic Programming

PROBLEM:

Consider an array A with N positive integers.

Let B be defined as an array of N integers such that for each $1 \leq i \leq N$:

- $B_i = A_i - 1$, if A_i is even.
- $B_i = A_i + 1$, if A_i is odd.

You are given an array C of size $2 \cdot N$ which is formed by taking all the elements of the arrays A and B and rearranging them.
You need to find the number of distinct arrays A whose sum of elements is K, from which C can be obtained.

Note: Two arrays X and Y are considered to be distinct if there exists no rearrangement of X which is the same as Y.

Since the final answer can be very large, print the answer modulo $10^9 + 7$.

EXPLANATION:

Some Observations:

- If an odd number x occurs m times in array C, then $x + 1$ should also occur m times, (otherwise the answer will be zero), because if x occurred in array A, $x + 1$ must occur in array B and vice versa.
- Similarly an even number y occurs m times in array C, then $y - 1$ should also occur m times in array C
- Number of odd elements is equal to Number of even elements = N.
- Minimum possible sum of array A will be the sum of all the odd numbers of array C
- Maximum possible sum of array A will be the sum of all the even numbers of array C.

This implies that if $K <$ sum of all odd numbers in array C OR $K >$ sum of all even numbers in array C, the answer is zero.
Now assume D = $K-$ sum of all odd numbers in array C.
Consider a possible array A with all the odd elements of C. We need to select any D even elements from C and replace each of them with their adjacent smaller odd number to increase the sum by D.
We denote the number of distinct even numbers in array C by M and the count of each even number by $P_1, P_2, ..., P_m$. Thus we need to find the solution of the equation $X_1 + X_2 + ... + X_m = D$, such that $0 \leq X_1 \leq P_1, 0 \leq X_2 \leq P_2, ...0 \leq X_m \leq P_m$.

The solution to the above equation can be obtained by standard dynamic programming under given constraints.

TIME COMPLEXITY:

$O(N^2)$ for each test case.

PREREQUISITES

Dynamic Programming, Combinatorics

PROBLEM

Consider an array A of length N where $0 \leq A_i \leq 10^5$. Construct an array B of length $N-1$ as $B_i = min(A_i, A_{i+1})$.

You are given the array B, find the number of arrays A of length N, which can be used to construct array B by the above process.

EXPLANATION

This is a DP combinatorics problem, So I recommend pen and paper as well. Also, using $M = 10^5$ throughout the editorial.

What we can see here is that, Either $B_i = A_i$ or $B_i = A_{i+1}$. Let's consider first element A_1.
Either

- $A_1 > B_1$ OR
  In this case, it doesn't matter which value A_N takes, as long as $B_1 < A_1 \leq M$, So there are $(M - B_1)$ ways to choose the last value. Also, A_2 must be equal to B_1 in this case.
- $A_1 = B_1$
  In this case, A_2 can take any value in range $[max(B_1, B_2), M]$.

We can see that we only care about whether $B_i = A_i$ or $B_i = A_{i+1}$ and not the actual values of A_i and A_{i+1}.

This suggests a kind of DP where we maintain one state for keeping track of index, and a flag to determine whether $B_i = A_i$ or $B_i = A_{i+1}$

Let's denote $f(i)$ as the number of ways to select first i elements of A in valid way such that $A_i = B_i$
Also, let's denote $g(i)$ as the number of ways to select first $i+1$ elements of A such that $A_{i+1} = B_i$ and $A_i > A_{i+1}$. This second condition is added, to avoid double counting of the case where $A_i = A_{i+1} = B_i$

The final answer would be $f(N-1) \times (M - B_{N-1} + 1) + g(N-1)$, as this covers both the cases when $A_{N-1} = B_{N-1}$ (first term) and $A_{N-1} > B_{N-1}$ (second term).

For simplicity, let's denote $w(x) = M - x + 1$, we we need to use it quite frequently. Required answer becomes $f(N-1) * w(B_{N-1}) + g(N-1)$. $w(B_{N-1})$ denotes the number of ways to select last element when first $N-1$ elements are already chosen.

Now, let's see how base cases and transitions work.

Base cases

We can see that $f(1) = 1$, as $A_1 = B_1$ is the only way.
For $g(1)$, we choose $A_2 = B_1$ and $A_1 > B_1$ which can be done in $(M - B_1) = w(B_1 + 1)$ ways.

Hence, $f(1) = 1$ and $g(1) = w(B_1 + 1)$ are the base cases.

Transitions

We need to find formula for $f(i)$ and $g(i)$ represented by terms $f(i-1)$ and $g(i-1)$.

For the computation of $f(i)$, the following cases need to be considered.

- Contribution of $f(i-1)$ in $f(i)$.
  If we have $B_i \geq B_{i-1}$, then we can choose $A_i = B_i$, which can be done in $f(i-1)$ ways.
- Contribution of $g(i-1)$ in $f(i)$
  If an array A is included in both $f(i)$ and $g(i-1)$, it implies $A_i = B_i$ and $A_i = B_{i-1}$. This can only happen when $B_i = B_{i-1}$. Hence, if $B_i = B_{i-1}$, each way included in $g(i-1)$ contributes one way in $f(i).

Similarly, for computation of $g(i)$, the following cases need to be considered.

- Contribution of $f(i-1)$ in $g(i)$
  $f(i-1) sets $A_{i-1} = B_{i-1}$ and $g(i)$ sets $A_{i+1} = B_i$. So we only need $A_i \geq max(B_{i-1}, B_i + 1)$. Hence, each way in $f(i-1)$ contributes $w(max(B_{i-1}, B_i + 1)$ ways in $g(i)$.
- Contribution of $g(i-1)$ in $g(i)$
  Only if $B_{i-1} > B_i$ that each way in $g(i-1)$ contributes one way in $g(i)$

I know the above details are quite hard to understand from the pure text. Let's assume [condition] evaluates to 1 if condition is true, and 0 if false. We can write the above recurrences as follows.

$$f(i) = [B_i \geq B_{i-1}]f(i-1) + [B_i = B_{i-1}]g(i-1)$$
$$g(i) = f(i-1) * w(max(B_{i-1}, B_i + 1)) + [B_{i-1} > B_i]g(i-1)$$

Hence, we can compute the values of $f(i)$ and $g(i)$ one by one in increasing order, and compute the final answer as $f(N-1) * w(B_{N-1}) + g(N-1)$

Questions you may have

- Why choosing these weird functions?
  Functions are a really nice representation of how dynamic programming works. Referring my code, ways[i][0] represents $f(i) and ways[i][1] represents $g(i)$.
- How do I debug this kind of problem?
  The best way would be to write a brute force solution. For this problem, I'd recommend writing brute force for $N = 4$ and $M = 10$, generating random array B and running 4 nested loops to compute the correct answer in $O(M^4)$. I have added such a snippet for your reference.

I hope you guys found it understandable and clear. Refer my code, written on same lines.

TIME COMPLEXITY

PREREQUISITES:

Dynamic programming, **next greater element**

PROBLEM:

An array is said to be good if the prefix upto its maximum element is sorted.

Given a permutation P of length N, find the number of ways to partition it into good subarrays.

EXPLANATION:

This is a counting problem, and most of the time that means the solution is going to involve either combinatorics or dynamic programming.
Combinatorics doesn't seem to be very helpful here, so let's try DP.

Let $P[L:R]$ denote the subarray of P starting at L and ending at R.

Let's try to make a couple of observations. Consider a subarray $P[L:R]$. Then,

- If $L = R$, then $P[L:R]$ is a good subarray.
- If $P[L:R]$ is a good subarray and $L < R$, then $P[L:R-1]$ is also a good subarray.
- If $P[L:R]$ is not a good subarray, then $P[L:R+1]$ is also not a good subarray.

All three facts should be fairly obvious to see.
Notice that this gives us a useful piece of information: if we fix the left endpoint L, then the set of R such that $P[L:R]$ is a good subarray form a contiguous range starting at L.
That is, the good subarrays starting at L are $P[L:L], P[L:L+1], P[L:L+2], \ldots, P[L:L+K]$ for some $K \geq 0$.

This immediately gives us an idea for a (slow) dynamic programming solution.

Initial idea

Let dp_i denote the number of ways to partition the suffix starting from i into good subarrays. Our final answer is dp_1, and the base case is $dp_N = 1$.

By fixing the length of the good subarray starting at i, it's easy to see that our transitions are simply

$$dp_i = dp_{i+1} + dp_{i+2} + \ldots + dp_{j+1}$$

where j is the largest integer such that $P[i:j]$ is good.

However, there are two problems with this solution:

- The first issue is that we need to find the right endpoint j (and find it quickly, at that)
- The second is that this is too slow. Each index can have a potentially $\mathrm{O}(N)$ transition, in which case this approach has a runtime of $\mathrm{O}(N^2)$.

Let's fix the problems individually.

Speeding up the DP

The speed issue is easy to fix using suffix sums.
Suppose we maintained the suffix sum array suf, where $suf_i = dp_i + dp_{i+1} + \ldots + dp_N$.
Then,

- $dp_i = dp_{i+1} + dp_{i+2} + \ldots + dp_{j+1} = suf_{i+1} - suf_{j+2}$
- $suf_i = suf_{i+1} + dp_i$

So, suf can be computed as we keep computing dp, and this allows us to optimize transitions to $\mathrm{O}(1)$.

Finding the right endpoint

Suppose you know the value of j for a left endpoint i. Can you find its value for left endpoint $i-1$?
It turns out, we can! Here's how:

- Suppose $P_{i-1} < P_i$. Then, the exact same value of j works for $i-1$ as well.
- Otherwise, $P_{i-1} > P_i$. In this case, let $x > i-1$ be the smallest integer such that $P_x > P_{i-1}$. Then, we choose $j = x-1$.

Proof

Both cases are not hard to prove.

- If $P_{i-1} < P_i$, and good subarray starting at $i-1$ can be turned into a good subarray starting at i by deleting the first element. Conversely, any good subarray starting at i still remains good when we insert P_{i-1} in the beginning. This implies that they have the same value of j.
- If $P_{i-1} > P_i$, the only sorted prefix is the one with length 1. So, the subarray starting here can only be good if P_{i-1} itself is the maximum element. So, the instant we hit something larger than P_{i-1}, the subarray is no longer good.

Dealing with the first case is trivial. However, dealing with the second needs us to find, for a given element, the first element to its right that is greater than it.

In fact, it is possible to find this next greater element for *every* index of the array in $\mathrm{O}(N)$, using a stack. The method of doing this is linked in the prerequisites.

Putting together the computation to find the right endpoint for each index, along with the DP speedup using suffix sums, gives us a solution in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Dynamic Programming, Strings

PROBLEM:

Gotham City is the home of Batman, and Batman likes to keep his people together.

There are N houses in the city. The resident of the ith house is of type A_i. It is known that people of the same type are friends and people of different types are not friends.

To maintain peace, Batman can do the following type of operation:

- Choose indices i and j $(1 \leq i \leq j \leq N)$ and reverse the substring $A[i,j]$.

Determine the length of the **longest** range of friends after applying **at most** k operations on the string for all $0 \leq k \leq N$.

EXPLANATION:

First let's see how we can increase the length of substrings having same character by the reversal operation as given in the question.
For a string say, s and for character say, x, we first take the longest and 2nd longest substrings having all characters as x. We are taking the longest and 2nd longest since we want to maximize the length of the substring formed in the given operation. Say they are at position i_1 and i_2 with length l_1 and l_2. Assuming i_1 is less than i_2, we can just reverse the substring $s[i_1....i_2 - 1]$ to get substring of length $(l_1 + l_2)$ having all x. We can keep repeating this operation till we get the longest substring that contains all the x that are present in string s. Also we would observe that the number of operations required to get the above longest substring would be the number of substrings of x in the original string which would be always less than n, i.e the length of the string s.

Now coming back to this question, what we can do is repeat the operations as defined in the above paragraph for each character from a to z and for each operation take the length of the maximum possible substring that can be formed among a to z as our answer for that operation.

TIME COMPLEXITY:

$O(NlogN)$, for each test case.

PREREQUISITES:

Math or dynamic programming

PROBLEM:

You're given a string S containing the characters `4, 0, *` only.
Each `*` can be replaced with either `0` or `4`.

Find the number of times `404` appears as a subsequence of S, across all possible replacements.

EXPLANATION:

There are two different ways to solve this problem: either somewhat directly using math, or using dynamic programming.

Solution 1 (Math)

Let's start with a slow solution first.

Instead of fixing a way of replacing the `*`s and counting the number of `404` subsequences, let's instead fix a `404` subsequence and figure out the number of replacements it occurs in.

So, suppose we fix indices $i < j < k$ to form the `404` subsequence. Then,

- Each of S_i and S_k should be either `4` or `*`.
  Further, if they're `*`, their replacement is unique (since we want them to be `4`).
- Similarly, S_j should be either `0` or `*`, and if it is `*` its replacement is unique.
- If the above conditions are satisfied, then any `*` that's not at positions i, j, k can be freely replaced by either `0` or `4`.
  So, if there are m such indices, the count is 2^m.

Doing this for each (i, j, k) triplet gives us a solution in $\mathrm{O}(N^3)$.

To optimize this, let's try to only fix the position of j, i.e, where the `0` in the `404` subsequence will be.
For this, we can choose any position such that S_j is either `0` or `*`.

Let the remaining number of `*`s be m.

Once `0` is fixed, let's try to fix i and k, i.e, the `4`s.
There are four options here:

- **Case 1:** We can choose a `4` before j and a `4` after j.
  For any such choice, the number of valid replacements is 2^m.
  The number of such choices equals the number of `4`s before j, multiplied by the number of `4`s after j.
- **Case 2:** We can choose a `4` before j and a `*` after j; and replace the `*` with `4`.
  The number of valid replacements is now 2^{m-1}, since we used up one `*`.
  The number of such choices equals the number of `4`s before j, multiplied by the number of `*`s after j.
- **Case 3:** Choose a `*` before j and a `4` after j.
  This is symmetric to case 2, just with 'before' and 'after' swapped.
- **Case 4:** Choose a `*` before j and a `*` after j.
  Here, there are 2^{m-2} valid replacements.

Notice that each case can be calculated in $\mathrm{O}(1)$ time if we know the number of `4`s before/after index j, and the number of `*`s before/after index j.

These counts are easy to find quickly: for example, you can precompute them using prefix or suffix sums; maintain them as you iterate; or even binary search on a list of positions.

In particular, each index can be processed in $\mathrm{O}(1)$ or $\mathrm{O}(\log N)$ time, which is fast enough.

Solution 2 (Dynamic programming)

Let $\mathrm{dp}[i][404]$ denote the number of occurrences of `404` as a subsequence of the first i characters, across all replacements of `*` in them.
Similarly define $\mathrm{dp}[i][40]$ and $\mathrm{dp}[i][4]$.

Then, we have the following transitions for each $1 \le i \le N$.

- First, for each $x \in \{4, 40, 404\}$, set $\mathrm{dp}[i][x] = \mathrm{dp}[i-1][x]$.
  This is because existing instances of each subsequence will continue to exist.
  Now, let's look at new instances of each one we can create using the i-th character.
- If S_i is `4`, then:
  - Add 2^m to $\mathrm{dp}[i][4]$, where m is the number of `*` before index i.
    This is because the `4` at this index will contribute one occurrence for every possible replacement so far, and there are 2^m such replacements.
  - Add $\mathrm{dp}[i-1][40]$ to $\mathrm{dp}[i][404]$.
    This is because we can create a new instance of `404` from a previously existing instance of `40`.
- If S_i is `0`, add $\mathrm{dp}[i-1][4]$ to $\mathrm{dp}[i][40]$.
  This is because we can create a new instance of `40` from an already existing instance of `4`.
- If S_i is `*`, perform both transitions.
  Note that in this case, you should initially set $\mathrm{dp}[i][x] = 2 \cdot \mathrm{dp}[i-1][x]$; once for each replacement.

The final answer is $\mathrm{dp}[N][404]$.

For ease of implementation, the states can be renamed into $\mathrm{dp}[i][0], \mathrm{dp}[i][1], \mathrm{dp}[i][2]$ corresponding to $4, 40, 404$ respectively.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

Prerequisites:

Dynamic Programming (DP)

Quick Explanation

We can use dynamic programming with states (Number of flowers seen so far, number of flowers chosen so far, number of modifications modified so far)

Explanation

Define $\mathrm{dp}[i][k][d]$ as, the maximum Hanna power possible if we choose the ith flower and we have chosen k flowers so far and have performed d modifications.

Define $\mathrm{RMQ[i][j]}$ as the range maximum query in the interval $[i..j]$ (inclusive)
Define $\mathrm{rmq}[i][j]$ as the range minimum query in the interval $[i..j]$ (inclusive)

Define $\mathrm{util}[i][j][c]$ as the maximum value of $\mathrm{RMQ}[i][j] - \mathrm{rmq}[i][j]$ by making exactly c modifications in the interval $[i..j]$.

Then,

$\mathrm{RMQ}[i][j] = \max_{i \le k \le j} a_k$
$\mathrm{rmq}[i][j] = \min_{i \le k \le j} a_k$

$\mathrm{dp}[i][k][d] = \max_{0 \le j < i} \max_{0 \le c \le i-j} (\mathrm{dp}[j][k-1][d-c] + \mathrm{util}[j+1][i][c])$

considering 0-based indexing.

Calculating $dp[i][k][d]$ by considering all values of c will certainly not fit in the time limit. So how do we optimize it?

We can make the observation that in any interval $[i..j]$, to maximize the value of $\mathrm{RMQ}[i][j] - \mathrm{rmq}[i][j]$, we will at most need 2 modifications. This is because with 1 modification, an element a_k may become the maximum in the interval, and with another modification, an element may become the minimum in the interval. Hence we should consider only the values 0, 1 and 2 for c.

We can pre-compute $\mathrm{RMQ}[i][j]$ in $O(n^2)$ time and $\mathrm{rmq}[i][j]$ in $O(n^2)$ time, using which we can pre-compute, $\mathrm{util[i][j][0]}$ in $O(n^2)$ time, $\mathrm{util[i][j][1]}$ in $O(n^3)$ time and $\mathrm{util}[i][j][2]$ in $O(n^4)$ time.

Using all these pre-computations, we can find the dp values in $O(n^4)$ time, which is enough to fit in the time limit.

The final answer is now simply $\max_{1 \le i \le N, 0 \le d \le D} \mathrm{dp}[i][K][d]$

PREREQUISITES:

The **minimax** algorithm.

PROBLEM:

There is a strip of N cells, some empty and some blocked.
Alice and Bob play a game, with Alice starting first.
On their turn, a player does the following:

- Choose K contiguous empty cells on the strip, and place dominoes on these K cells.
- A domino placed on turn T will disappear at the start of turn $T + D$.

A player who cannot make a move loses.
Find out the winner with optimal play; or whether the game will go on infinitely.

EXPLANATION:

As a first observation, note that if the game hasn't finished by the end of the D-th turn (under optimal play), it will go on forever.
This is because, on the $(D + i)$-th turn, the dominos placed on the i-th turn will disappear; so the current player will always have a move to make.

This means we can discard the 'disappearing' part entirely, and just work with placing normal dominoes.
The 'normal' game will, of course, have a unique winner — and obviously, this person is the only one who has a chance of winning the disappearing game as well.
We only need to check at the end whether the winner can force a win in $< D$ moves; if they can't, it's a draw.

This also tells us what each player's strategy is.
Suppose Alice wins the normal game. Then,

- Alice will make a move that allows her to force a victory in the minimum number of moves.
- Bob will make a move that allows him to prolong the game as much as possible; in an attempt to reach a draw.

If Bob wins, the roles reverse.

Together, this tells us the two pieces of information we need to know:

- Who the winner of the normal game is; and
- The least number of moves they need to obtain victory.

Both of these can be found using dynamic programming.

Each state of the strip can be represented using N-bit bitmasks.

We will compute the following pieces of information for each bitmask:

- $\text{winner}[mask]$, which is a boolean value.
  A value of 1 or `True` means that the person who plays first on this mask will win it; while a value of 0 or `False` means that the person going second wins.
  A mask will be called a *winning state* if $\text{winner}[mask] = 1$, and a *losing state* otherwise.
- $\text{time}[mask]$, which is the 'optimal' number of turns for this mask for the first player.
  That means the following:
  - If $\text{winner}[mask] = 1$, the $\text{time}[mask]$ is the *minimum* time needed for victory.
  - Otherwise, $\text{time}[mask]$ is the *maximum* amount of time the game can be prolonged for.

These values can be computed as follows:

- If no moves can be made from $mask$, then $\text{winner}[mask] = 0$ and $\text{time}[mask] = 0$.
- Otherwise, let S_{mask} be the set of states that can be reached from $mask$ by making exactly one move.
  If S_{mask} contains any losing state, then $\text{winner}[mask] = 1$; otherwise $\text{winner}[mask] = 0$.
- If $\text{winner}[mask] = 0$, then $\text{time}[mask] = 1 + \max(\text{time}[nxt])$ across all $nxt \in S_{mask}$.
  That is, if the current state is losing, then the player will choose to make a move such that the opponent needs as many possible moves to win.
- If $\text{winner}[mask] = 1$, then $\text{time}[mask] = 1 + \min(\text{time}[nxt])$ across all $nxt \in S_{mask}$, such that $\text{winner}[nxt] = 0$.
  That is, if the current state is winning, the player will choose to make a move such that they can win in as little time as possible; while also ensuring that they move to a losing state before passing the turn.

If the winner and time values are memoized, performing the above computations recursively is enough to solve the problem.

However, a little care is needed with implementation.
In particular, you might notice that there are technically 2^N possible bitmasks; and for $N = 30$ we certainly can't compute the answers for them all quickly enough.

However, in most cases, the set of bitmasks that are actually reachable is generally much smaller because we place multiple dominoes at the same time.
For example, with $N = 30, K = 2$ and a completely empty starting strip, the real number of reachable states is only slightly more than 10^6 which is very manageable.

The only outlier is if $K = 1$, in which case every mask can indeed be reached since we place dominoes one at a time.
However, $K = 1$ can be handled separately: if there are $\geq D$ empty spaces, the game is a draw; otherwise the winner is decided uniquely by the parity of the number of empty cells (Alice if odd, Bob if even).

For $K \geq 2$, recursively processing all reachable bitmasks and using a hashmap for memoization is good enough to get AC.

TIME COMPLEXITY

About 10^7 hashset/hashmap operations, in the worst case.

PREREQUISITES:

Factorization, Sieve of Eratosthenes, Dynamic programming, Binary search/2-pointers

PROBLEM:

Given an array A, in one move you can divide any element of it by one of its prime factors.
Find the minimum number of moves to make the array non-decreasing.

EXPLANATION:

A few greedy strategies might come to mind, but none of them really work.

Instead, let's try to use dynamic programming.
Let $dp_{i,x}$ denote the minimum number of moves to make $A_1 \le A_2 \le \ldots \le A_i$, such that $A_i = x$.
Transitions are fairly easy to see:

$$dp_{i,x} = cost(A_i, x) + \min_{1 \le y \le x} dp_{i-1,y}$$

where $cost(a, b)$ denotes the smallest number of moves to convert a to b when dividing by primes.

There are a couple of things to do now:

- Figure out how to compute $cost(a, b)$
- Optimize the dp computation to work enough.

Computing $cost$

Note that $cost(a, b)$ depends solely on the prime factorizations of a and b. In particular,

- b must be a factor of a
- Exactly one prime can be removed from a at each step
- So, if $p(a)$ denotes the number of (not necessarily distinct) primes in the factorization of a, then $cost(a, b) = p(a) - p(b)$.
- Alternately, you can see that we remove exactly $\frac{a}{b}$ using primes, so this also equals $p(\frac{a}{b})$.

So, all that needs to be done is to compute $p(n)$ for every $1 \le n \le 5 \cdot 10^5$.
This is a standard computation that can be done with the help of a **sieve** .

Optimizing dp

There are two things to optimize: the number of states, and performing the transitions.

Note that $dp_{i,x}$ only matters for x that is a factor of A_i since no other x can be reached via division.
So, all other states can be ignored. This gives an easy upper bound of about $\mathrm{O}(N\sqrt{M})$ states (where $M = 5 \cdot 10^5$), but can be in fact bounded by $200 \cdot N$, since no number in the given range has more than 200 factors.

This also makes our transitions take ≤ 200 steps each (since we only need to iterate over divisors of A_{i-1}, making the entire solution something like $\mathrm{O}(N \cdot d^2)$ where $d = 200$.

Unfortunately, this is likely too slow, and we need to optimize it a bit. There are a couple ways to do this.

- One way is to use binary search along with prefix minimums. Note that $dp_{i,x}$ depends exactly on some prefix minimum of dp_{i-1}. So, if the prefix minimums of dp_{i-1} are computed, the appropriate one can be

found using binary search on the factors of A_{i-1}. This gives a solution in $\mathrm{O}(N \cdot d \log d)$.
- Instead of binary search, one can also use two pointers to achieve $\mathrm{O}(N \cdot d)$.

Finally, there is the issue of actually finding the factors. Once again, there are two ways to do this:

- Directly factorize each A_i in $\mathrm{O}(\sqrt{M})$.
- Or, once again use a sieve to precompute the divisors of all numbers $\leq M$.

The second method is faster and is what is intended, however the first method can also pass if written well. Note that you might TLE if you use both sqrt factorization *and* binary search, but optimizing any one of them should be enough to get AC.

TIME COMPLEXITY

$\mathrm{O}(M \log M)$ precomputation followed by $\mathrm{O}(N \cdot D)$ per test case where $M = 5 \cdot 10^5$ and $D \approx 200$ is the maximum number of factors a number $\leq M$ can have.

PREREQUISITES:

Dynamic Programming, Probability and Expected values.

PROBLEM:

Given an array of n ranges, where the ith range is denoted by a pair of two integers X_i and Y_i. For a range of valid indices in the array, we visit each index i within the range and pick a number $X_i \leq V \leq Y_i$ which is added to our score. If our score is less than M, we don't gain a profit, else we gain a profit equal to the score collected. Find the expected value of the score if V was chosen uniformly randomly.

QUICK EXPLANATION:

We will first calculate the expected value of our loss, i.e. the value of our score if it is **less** than M.

For each range, create a dynamic programming table $dp[i][j]$ denoting the probability of getting a loss of j from the first i ranges in the query range. Once we get the expected value of our loss, we can reduce it from the overall expected value of our score if the constraint of M was not there to find our expected profit.

EXPLANATION:

We will first target our expected loss. If our score is less than M, we don't receive any profit, in other words the score is counted as a loss. Hence, we aim to find the expected value of our score if it is allowed to be **less** than M only. The expected profit would be the overall expected value (our expected score if there is not restriction of M) reduced by our expected loss.

To calculate the expected loss, we first have to observe that our loss can never be greater than or equal to M (because if it was, it won't be a loss). Let us only focus on the given index range during the query, let say there are L ranges.

Define a dynamic programming $dp[L+1][M]$, where $dp[i][j]$ means the probability of getting a loss of j from the first i ranges. Here $dp[0][0] = 1$ and $dp[0][x] = 0$ for $x > 0$ should be the base cases since you will always get 0 loss from 0 ranges.

Transitions are trivial : for $i \geq 1,\ dp[i][j] = \sum_{k=X_i}^{k \leq min(Y_i, j)} \frac{dp[i-1][j-k]}{Y_i - X_i + 1}$

TIME COMPLEXITY:

$O(Q \cdot K \cdot M \cdot C)$. Where C is the maximum bound on the value of Y_i.

PREREQUISITES
Dynamic Programming

EXPLANATION

Some Observations:

- The trick here is to see that since the sum of the two players' scores is the sum of the input list, Player 1 tries to maximize score1−score2, while Player 2 tries to minimize it.
- Let A_i denote the value of the i_{th} number. Let $dp[i][j]$ denote the difference between the score of the first player and the second player, score1 - score2 if they play the game between the indices i and j.
- For every interval (i,j) there is a fixed person who is going to be playing on that interval. $dp[i][j]$ stores the difference between the scores of the person who is playing on that interval and then person who plays next if they are only allowed to use that interval. Since the person playing can only take the ends of the intervals, only i and j are available.
- Call the person taking currently A. If he takes i, then the interval becomes $(i+1,j)$. $dp[i+1][j]$ stores the value of $B-A$, so you subtract it from $a[i]$ to get A + $a[i]$ - B which is the value for the current segment if A takes i. It is very similar for if A takes j, and you compare them to get $dp[i][j]$. $dp[i][j]$ = $max(a[i]-dp[i+1][j], a[j]-dp[i][j-1])$.

For better explanation refer: [Explanation]

PREREQUISITES

Dynamic Programming, Segment Tree

PROBLEM

You have an array A containing all the natural numbers from 1 to N twice. Find the number of **ordered pairs** of **disjoint** subsequences (P, Q) of A such that they are equal.

EXPLANATION

Let,

- $dp_i :-$ denote the number of **ordered pairs** of subsequences (P, Q) that can be formed taking elements from index 0 to i.
- $f_i, s_i :-$ denote the indices of the first and the second occurrence of A_i.
- $t_i :-$ denote the number of **unordered pairs** of subsequences (P, Q) ending at indices f_i and s_i respectively.

Now, let's say we have already computed dp_{i-1}.

Case 1: If $i = f_i$,

No additional pair of (P, Q) can be formed by including the element at index i. So we can simply say, $dp_i = dp_{i-1}$.

Case 2: If $i = s_i$,

We have to add all the ordered pairs of subsequences (P, Q) ending at f_i and i to dp_{i-1}. So we can say,

$dp_i = dp_{i-1} + 2 \cdot t_i$.

Let's try to find t_i now. This comprises of 3 parts:

- **Part 1:** We can add A_{f_i} and A_i respectively at the end of all the **ordered pairs** of subsequences (P, Q) formed from indices 0 to $f_i - 1$.
- **Part 2:** We can add A_{f_i} and A_i at the end of all the **unordered pairs** of subsequences (P, Q) which end at f_x and s_x repectively, such that, $0 \le f_x < f_i < s_x < i$.
- **Part 3:** An additional subsequence pair $([A_{f_i}], [A_{s_i}])$.

The **first part** is simply dp_{f_i-1} and as f_i is the first occurrence of A_i, we can also say $dp_{f_i-1} = dp_{f_i}$ *(Case 1)*.

Now, for the **second part**, you have f_i and i as fixed and you need to sum all the values of t_x where, $f_x < f_i < s_x < i$. So, let's say we have maintained an array T (say) till index $i - 1$ with the value t_x at the index of the first occurrence and $-t_x$ at the index of the second occurrence of all the elements having $s_x \le i - 1$.

The sum of this array T from $j = 0$ to $j = f_i - 1$ will give us the value of the second part because in case of $s_x < f_i$, the values t_i and $-t_i$ are simply going to cancel each other out.

For the condition of $s_x < i$ to be fulfilled at all times, we will update the array T only as we iterate over i.
We **cannot** pre-compute the values of T, otherwise, the summation will also include values where $f_x < f_i$ but

$s_x > i$.

Mathematically,

$$T \begin{cases} T_{f_x} = T_{s_x} = 0, & \text{if } i - 1 < s_x \\ T_{f_x} = t_x \text{ and } T_{s_x} = -t_x, & \text{if } i - 1 >= s_x \end{cases}$$

and, $t_i = dp_{f_i} + \sum_{j=0}^{j=f_i-1} T_j + 1$.

Finally, we can update the array T as follows:
$T_{f_i} = t_i$ and $T_i = -t_i$

These range sum and point update queries in T can be handled efficiently using a Segment Tree.

Note: No updates in T were required in **Case 1** as $i < s_i$.

TIME COMPLEXITY

The time complexity is $O(N \cdot \log(N))$.

PREREQUISITES:

Simple dynamic programming, basic data structure knowledge — for example `sets`

PROBLEM:

N people order takeout, the i-th person's order will be ready at time X_i and he will arrive to collect it at time Y_i.
$X_i < Y_i$ is known.
Each person will:

- Take his order if it's present.
- If his order is not present but some others are, take whichever present order has the lowest *index*.
- Do nothing if no orders are present.

An *evil person* is one who takes an order that isn't his.
A devil can steal exactly one person's order. What's the maximum possible number of evil people?

EXPLANATION:

The main observation required to solve this problem is the following: at any moment of time, at most one order (that should be present at this time) will be missing.

Proof

Suppose the devil steals person i's order that's supposed to be ready at time X_i; and the person arrives at time Y_i.
Then,

- At any time before X_i, everything is as it should be
- At times in the range $[X_i, Y_i - 1]$, *almost* everything is as it should be: the only difference is that person i's order is missing.
- At time Y_i:
  - If there are no orders present, nothing happens and person i leaves.
    Note that if the devil hadn't stolen his order, then in the normal process there would be no orders remaining at the end of this time anyway; which means the process will go on as usual after Y_i and no more orders will be missing.
  - If there is an order present, person i will take it. Suppose he takes order j.
    Then, at times in the range $[Y_i + 1, Y_j - 1]$, person j's order will be the only one missing (everything else is accounted for, and person i's order would normally have been taken at time Y_i anyway)
- Continue this process to see that at any instant, at most one order is missing.

Note that in the last case in the above proof, the process during time $[Y_i + 1, Y_j - 1]$ proceeds exactly the same as if the devil had stolen person j's order initially!
This observation leads us to a solution.

Let $\text{take}[i]$ denote the index of the takeout that person i will take, if his own is missing (and $\text{take}[i] = -1$ if no such person exists).
Note that since at most one order is missing at any time, $\text{take}[i]$ is uniquely defined.

Now, if the devil steals person i's takeout, it's easy to see that the people who become angry form a chain, going as $i \to \text{take}[i] \to \text{take}[\text{take}[i]] \to \ldots$, as long as the next person exists (i.e doesn't equal -1).

This tells us that we need to compute two things: the $\text{take}[i]$ values for each i, and then the longest chain in them.

Computing take

Let's attempt to find $\text{take}[i]$.
If person i's takeout is missing, then at time Y_i we know that all remaining takeouts will be of persons j such that $X_j < Y_i < Y_j$, i.e, takeouts that were prepared before Y_i but weren't taken yet.
We want to find the smallest j such that this condition is satisfied, and do so for *every* i.
If no such j exists, then $\text{take}[i] = -1$ of course.

Note that the problem constraints guarantee that all the X_i and Y_j are distinct between 1 and $2N$, which means something (in fact, exactly one thing) happens at each time instant.
Let's use that fact to simulate the process nicely:

- Iterate across time, from 1 to $2N$. Also maintain a set of takeouts that haven't been taken yet, say S.
- At time t,
  - If there's an i such that $X_i = t$, insert i into S.
  - If there's an i such that $Y_i = t$, delete i from S.
    Then, (if S is non-empty) $\text{take}[i]$ will be the smallest element of S.

At each time instant, we do only a couple of operations at most.
So, if S were a data structure that allowed us to quickly insert and erase elements, as well as get the smallest element, we'd be done.

There is indeed such a data structure: a `set` (`std::set` in C++, `TreeSet` in Java, Python unfortunately doesn't have one built-in).
This allows us to insert/delete elements in $\text{O}(\log N)$, and get the minimum element in $\text{O}(1)$.

So, make S a set and simulate the process as mentioned above, for $\text{O}(N \log N)$ runtime.
Finding the longest chain

Once the $\text{take}[i]$ values are known, finding the longest chain is fairly straightforward.

Let $\text{dp}[i]$ be the length of the longest chain if the devil steals person i's takeout.
Then,

- If $\text{take}[i] = -1$, we have $\text{dp}[i] = 0$
- Otherwise, $\text{take}[i] = 1 + \text{dp}[\text{take}[i]]$

As long as memoization is used, computing this for every i takes $\text{O}(N)$ time.
The answer is just the maximum of the dp array.

TIME COMPLEXITY

$\text{O}(N \log N)$ per test case.

PREREQUISITES:

Fast Fourier transform, Dynamic Programming, Divide and Conquer, Combinatorics, Modular multiplication

PROBLEM:

Alice initially had X stones. She divides the X stones into N piles where the ith pile contains A_i stones.

Now, she randomly selects a subset of piles S such that every subset has an **equal** probability of getting selected.
After selecting the subset, for every $1 \le i \le N$,

- If $i \in S$, Alice colours all the stones in the ith pile RED.
- If $i \notin S$, Alice colours all the stones in the ith pile BLUE.

Let the total number of RED coloured stones be R and the number of BLUE coloured stones be B.
Alice now wonders what the expected value of $|R - B|$ is. Can you help Alice? Output the answer modulo 998244353.

Note that, $|x|$ denotes the absolute value of x. For example, $|-4| = 4$ and $|7| = 7$.

EXPLANATION:

The problem can be rephrased in simple words as:
Given an array A of length N, we randomly choose a subsequence P and add all the remaining elements to a second subsequence Q. Find expected value of $|sum_P - sum_Q|$ (where sum_X for a subsequence X represents the sum of all the elements of X and $|i|$ represents the absolute value of a number i). Find the answer modulo 998244353.

Let S be the sum of all elements of A (i.e. $S = \sum_{i=1}^{N} A_i$). Since each subsequence has equal probability of getting selected therefore for each subsequence we calculate $|S - 2 \cdot sum_P|$ and in the end divide the sum by 2^N to get the final answer.
The solution in steps:

Method 1: Brute force approach - Iterate over all subsequences P of array A and for each subsequence P formed calculate sum_P and $sum_Q = S - sum_P$ .The time complexity of this approach is $O(2^N)$ which is too slow for such large N.

Method 2: Dynamic Programing

The dp state is defined as follows: dp_i represents the number of ways to get sum equal to i by picking some subset of elements of array A. This dp can be calculated in $O(N * S)$ time using standard knapsack approach.
Example of calculating this dp

Then the final answer can be calculated as follows: $ans += dp_i * |S - 2 \cdot i|$ for all $0 \le i \le S$
But since N and S are both large, this is not fast enough to pass the test cases for the given constraints.
Sub-task for Method 2: FFT
We can use FFT to speed up the computation of the dp states. FFT is an algorithm that allows us to multiply two polynomials of length N in $O(NlogN)$ time,
Let the polynomial $P(x) = (1 + x^{A_1}) * (1 + x^{A_2}) * (1 + x^{A_3}) * ... * (1 + x^{A_n})$. We can compute P by multiplying the polynomials in a divide and conquer fashion using FFT. The coefficient of x^i in P represents the number of ways to get sum equal to i.
Time complexity of this solution is $O(Slog(N)log(S))$ which is fast enough to pass the test cases.

For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(Xlog(X)log(N))$ for each test case.

PREREQUISITES:

Prefix Sums, Dynamic Programming

PROBLEM:

Given two matrices A and B, of size $N * M$. The value in each cell denotes the count of students currently in that cell.

- Students represented by matrix A want to reach the left-edge of the field moving only leftwards.
- Students represented by matrix B want to reach the top-edge of the field moving only upwards.

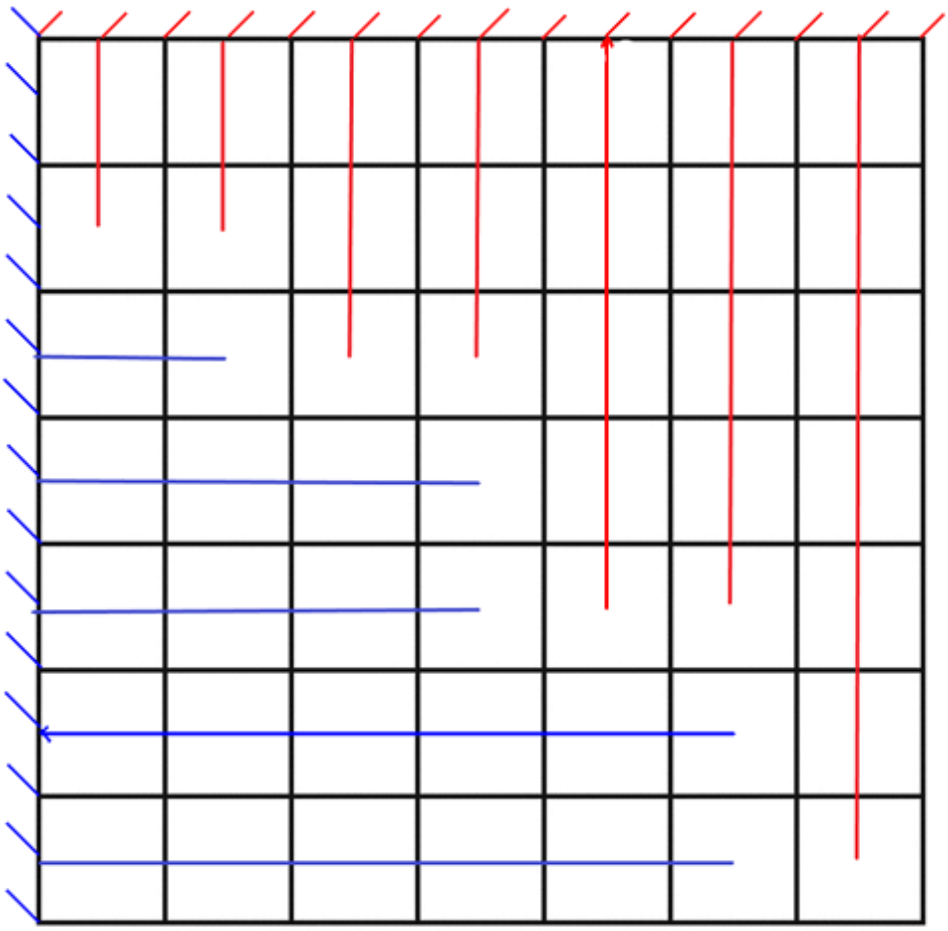
The paths followed by two students going to different hostels should not intersect.

Calculate the maximum number of students which can reach their hostels.

EXPLANATION:

The given problem can be solved by using two-dimensional dp speeded up with prefix sums with $O(M * N)$ time complexity and $O(m * n)$ extra space for the dp matrix.

Now we need to make the partition of the field to maximize the number of students, so this would be step-like structure starting from the top-left end towards the bottom-right end.



Preprocessing: For both matrices A and B, for every column calculate the prefix sums, such that:

- $A[i][j] = A[i][j] + A[i+1][j] ... + A[n-1][j]$
- $B[i][j] = B[0][j] + B[1][j] + ... + B[i][j]$.

Now we would start building our $dp[m+1][n+1]$ matrix from the last column and from bottom to top for every column, and the formula for $dp[row][col]$ is

- $dp[row][col] = max(dp[row+1][col], dp[row][col+1] + A[row-1][col] + B[row][col])$
  (Note: elements going out of the matrix would be equal to 0)

$dp[row][col]$ denote the maximum number of students who could reach their hostels when the submatrix $(row...n) * (col...m)$ is considered.

Now, The maximum number of students that would reach their hostels would be equal to $dp[0][0]$.

TIME COMPLEXITY:

$O(N * M)$ for each test case.

PREREQUISITES:

Depth First Search, Dynamic programming, GCD, Trees

PROBLEM:

There is a rooted tree of N vertices rooted at vertex 1. Each vertex v has a value A_v associated with it.

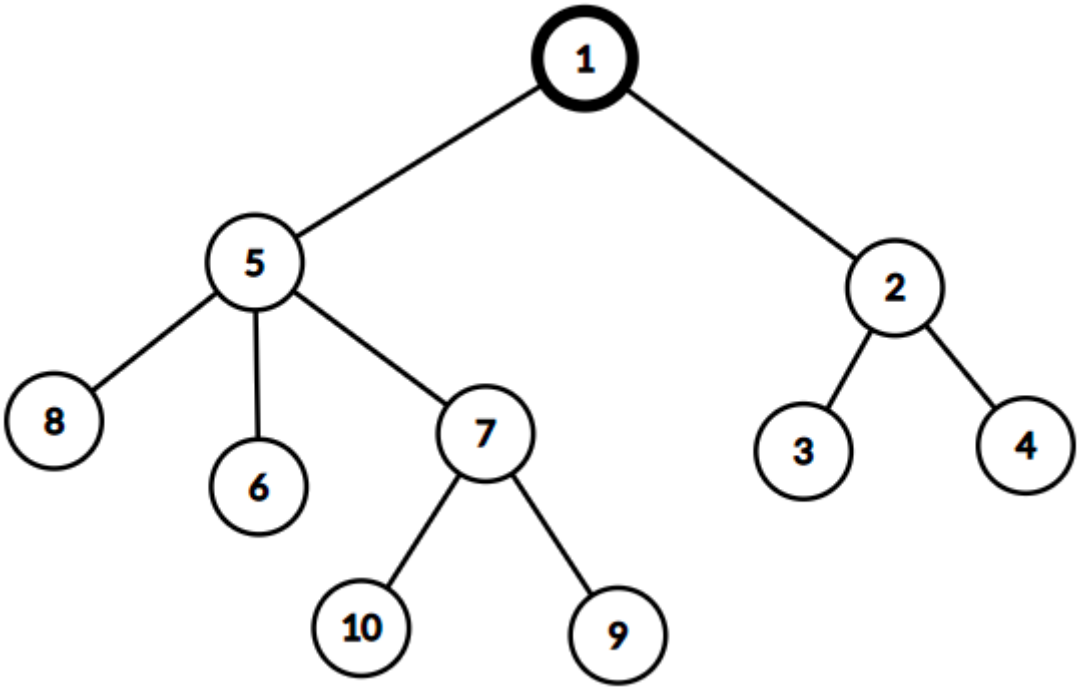
You choose a vertex v (possibly the root) from the tree and remove all vertices on the path from the root to the vertex v, also including v. This will result in a forest of zero or more connected components.

The beauty of a connected component is the GCD of the values of all vertices in the component. Find the maximum value of the sum of beauties of the obtained connected components for any choice of v.

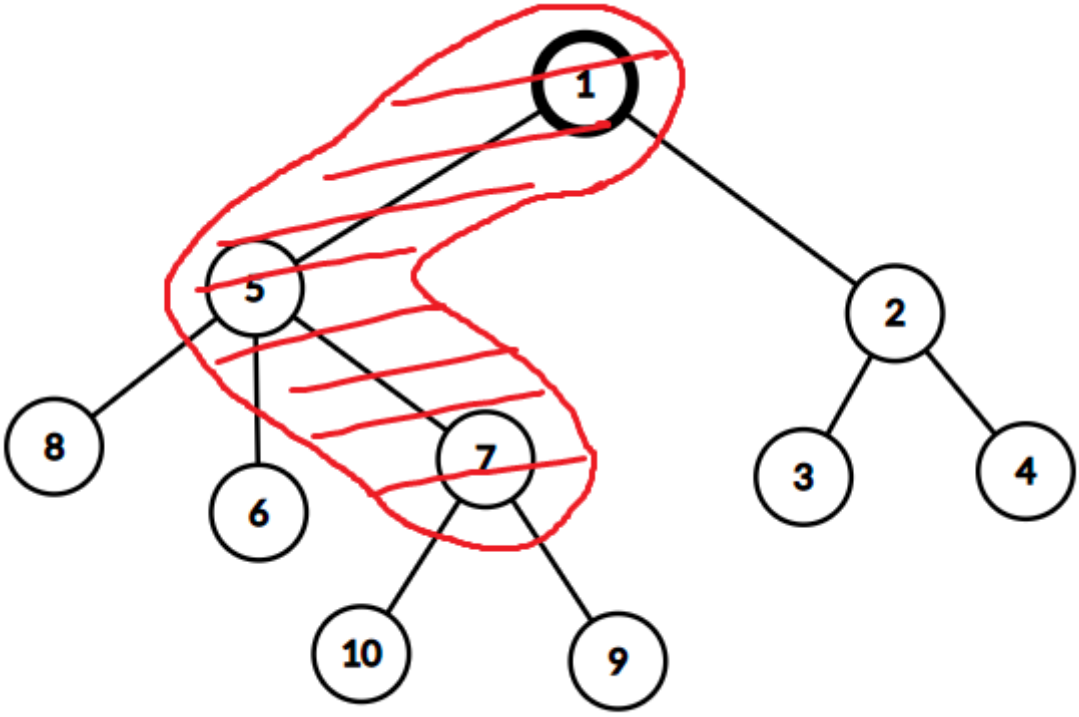
Here, GCD stands for Greatest Common Divisor.

EXPLANATION:

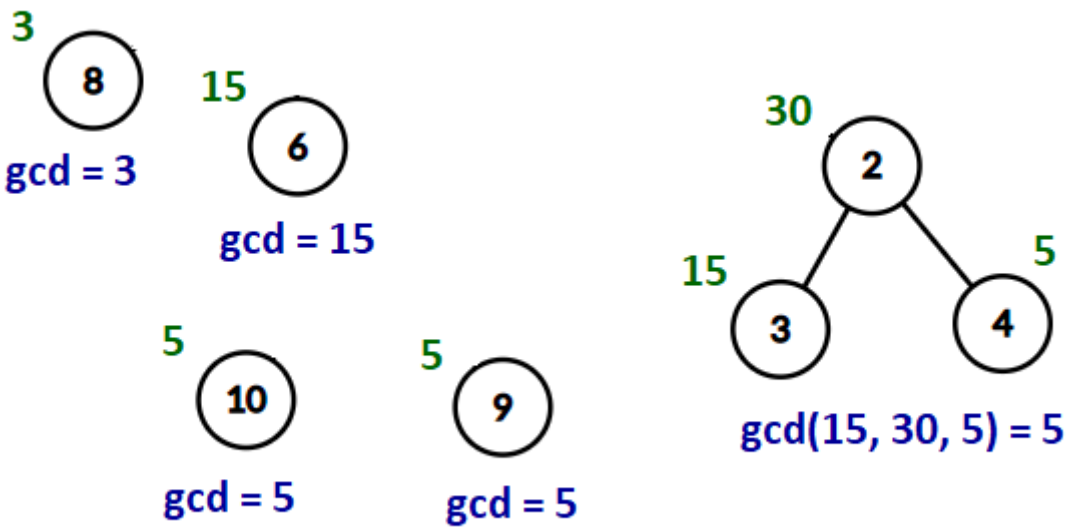
Let us look at what happens when we remove the path from a node to the root of the tree (node 1) . Let initially the tree look like what is shown below.



And lets say we decided to remove path from node 7 to node 1 from this tree. All the nodes on the path are removed. The Remaining nodes still form a forest of zero or more connected components (trees). All the children of node removed (7 here) are disconnected from each other and form different connected components (subtrees of node 10 and 9 here). For each node on the path till root (ancestors) all children of that node other than the child which is included in the path from node u to node 1 form different connected components.



Therefore for every node u in the tree we need to find the sum of beauties of subtrees of all its children and sum of beauties of subtrees of all the children, except u, of all ancestors (all nodes other than u that lie on the path to the root) of node u. The maximum of this value over all nodes of the tree is the answer to the problem.



Removing every node and its path to the root and calculating the answer naively takes $O(N^2)$ time which is not fast enough to pass the test cases for large values of N.
To improve the runtime of the solution, let us perform some calculations first,
Let G_u represent the gcd of all node values A_v present in the subtree of u and $SumofG_u$ represent the sum of G_x for all children x of node u
Formally, $G_u = gcd(all(A_v)),\ \ v \in$ Subtree of u.
$SumofG_u = \sum G_x$ where x is a child of node u.

Both of these parameters for all nodes N of the tree can be done in $O(Nlog(max(A_v)))$ by a single dfs traversal of the tree starting from root node and basic dynamic programming.

Calculation of the parameters defined above for all nodes

```
void dfsforgcd(int u, int p = -1)
{
  g[u] = a[u];
  sumofg[u] = 0;
  for (auto x : v[u])
  {
    if (x == p)
      continue;
    dfsforgcd(x, u);
    g[u] = gcd(g[u], g[x]);
    sumofg[u] += g[x];
  }
}
```

Sum of beauties of connected components we get after removing the path from node u to node 1 = $SumofG_u$+ $\sum(SumofG_p - g_p) - g_u$ where p are all the nodes on the path from root node 1 to node u (except u).
The maximum of this value over all nodes of the tree is the answer to the problem.

TIME COMPLEXITY:

$O(N \cdot log(max(A_i)))$ for each test case. $log(max(A_i))$ factor is added due to the calculations of the gcd which takes logarithmic time.

PREREQUISITES:

Longest increasing subsequence

PROBLEM:

There's an $N \times M$ grid with K special cells.
From a cell (x, y):

- If it's a special cell, Chef can move to any one of the 8 neighboring cells
- Otherwise, Chef can move to one of the 4 horizontally/vertically adjacent cells

Find the minimum number of moves to reach (N, M) from $(1, 1)$.

EXPLANATION:

Clearly, in an optimal sequence of moves every move we make is going to be one of the following:

- $(x, y) \to (x, y+1)$
- $(x, y) \to (x+1, y)$
- $(x, y) \to (x+1, y+1)$ (only possible from a special cell)

We need to make $N-1$ moves along x and $M-1$ moves along y, and a special cell allows us to at best combine two of these moves into one (thus saving one step). Going in the 'wrong direction' doesn't decrease the number of steps we need to make, so it's suboptimal.

This also means that any special cells of the form (x, M) or (N, y) don't matter at all, since we can't use them to make the third type of move. Let's discard such special cells first.

Now, notice that if we touch s special cells along our path, the number of moves will be exactly $N + M - 2 - s$. This follows from our observation above: we make $N - 1 + M - 1$ moves, and each special cell allows us to combine two of these into one.

To minimize this, we'd like to find the largest possible value of s, i.e, the largest number of special cells that we can visit along our path.
Note that if (x_1, y_1) and (x_2, y_2) are two special cells, we can visit both of them along our optimal path if and only if $x_1 < x_2$ *and* $y_1 < y_2$.

In particular, if we sort the special cells in ascending order of their x-coordinates, the y-coordinates must form an increasing sequence.
In other words, what we're looking for is just the longest increasing subsequence of the y-coordinates!
This can be computed in $\mathrm{O}(K \log K)$ time in a variety of ways: see here for a couple of them.

Note that there is one extra condition here: we need to ensure that we also pick distinct x-coordinates.
One simple way to ensure this is to sort the cells in increasing order of x, and break ties by *decreasing* order of y.

TIME COMPLEXITY:

$\mathrm{O}(K \log K)$ per testcase.

PREREQUISITES:

Greedy, DP etc.

PROBLEM:

There is going to be a major fair in **Government College Of Engineering Nagpur – GCOEN** Named as **Adhyaaya** . In which there are many events going to happen. There are nn numbers of events you can attend. You know its starting and ending days and the amount of money you could win in the event.But the condition is that you can attend only one event in a day. Tell the maximum amount of money you will win.

EXPLANATION:

In the given test case, There are **4** events in which you can participate

- first event will start on day 2 and end on day 4
- second event will start on day 3 and end on day 6
- Third event will start on day 6 and end on day 8
- Fourth event will start on day 5 and end on day 7

So, first event and third , first event and fourth event are the only 2 events whose days are not co-inciding. [2,4] && [6,8] and [2,4] && [5,7] respectively.
if we add prizes of 2 events , we will get prize of (4+2 = 6) in first condition and (4+3 =7) in second condition, so we will print max of them i.e **7** .

PREREQUISITES :

Basic Dynamic Programming

PROBLEM :

You are given a sequence of integers of length N, consisting of ≈ 1000 distinct values. Now, you need to find the number of subsequence of length K of this sequence, such that each pair of elements in the selected subsequence is pair-wise distinct.

QUICK EXPLANATION :

A quick look at the constraints, and we can prove that the given array $A[]$ can consist of at max 1007 distinct values. So, K shall always be ≤ 1007. Then, we can group numbers with equal values together. Then, the task comes down to finding the number of ways to select K numbers, such that we pick at most 1 number from each group. To accomplish this, we can do some dynamic programming, which runs in approx $O(X^2)$ time, where $X \approx 1000$

EXPLANATION :

Let's begin by having a look at the constraints. It states that each $A_i \leq 8000$, and that the numbers A_i are prime. Now, we know that prime numbers occur less frequently than their composite counter parts.

In fact, the number of prime numbers ≤ 8000 is exactly 1007. So, we know that the array given in the input shall also consist of at max 1007 distinct values of A_i. We initially needed to find the number of subsequences consisting of at most K distinct values, where $K \leq 100,000$.

But since the maximum number of distinct values in any subsequence of this array is always ≤ 1007, so we can take $K = min(K, 1007)$. Now, we need to solve the original problem using these modified constraints.

Another observation we can make is that the ordering of elements in a subsequence is not important. For example, if we pick some valid subsequence $a_{k_1}, a_{k_2} ... a_{k_z}, \quad z \leq K$, $a_i \neq a_j$ if $k_i \neq k_j$, and $k_1 < k_2 ... < k_z$, then in what ever order we arrange the elements a_{k_i}, it does not make a difference, they do not break the pairwise distinctness property.

So, the original problem now boils down to finding the number of subsets $\{k_1, k_2 ... k_z\} \quad z \leq K$, of the set $\{1, 2, ... N\}$, such that $a[k_i] \neq a[k_j]$ if $k_i \neq k_j$

We can do a dynamic programming for the following. Let the number of distinct values appearing in the array be ctr. Then we can replace every distinct value appearing in the array by an integer in the range $[1, ctr]$ and our answer remains unchanged.

Also, for each distinct value, let's maintain a variable $cnt[i]$ that states the number of times i appears in the given array a. Then we can do the following dynamic programming, that is in a way extremely similar to how to construct pascal triangle for binomial coefficients but with a small change :

$dp[i][0] = 1 \quad \forall \quad i \geq 0$

$dp[i][j] = dp[i-1][j] + dp[i-1][j-1] \cdot cnt[i]$

The number we are looking for is $\sum_{i=0}^{K} dp[ctr][i]$

Here, $dp[i][j]$ indicates the number of valid subsets of size j we can pick considering only array values $\leq i$

For a fixed (i,j) , the term $dp[i-1][j]$ considers the valid subsets using array values $< i$ and the second one ensures we pick a single instance of i, and add it to valid subsets of size $j-1$ consisting of array values $< i$.

Since we have $cnt[i]$ instances of element i in the array, so we can pick exactly one among them in $cnt[i]$ ways.

On a side note :

Assume we want to solve this problem without the additional constraint of $A_i \leq 8000$ and A_i being prime. Then, N and K can be as large 10^5. The answer is then given by :

$[x^k] \prod_{i=1}^{ctr} (1 + cnt[i] \cdot x)$

That is, the answer is given by the coefficient of $[x^k]$ in the polynomial :

$(1 + cnt[1] \cdot x) \cdot (1 + cnt[2] \cdot x) \cdot ..... \cdot (1 + cnt[ctr] \cdot x)$

We can multiply all these smaller polynomials in $O(N \cdot log^2 N)$ time using high precision FFT Modulo $10^9 + 7$

That's it, Thank you !

Your comments are welcome !

COMPLEXITY ANALYSIS :

Time Complexity : $O(X^2)$, where $X \approx 1000$

Space Complexity : $O(X^2)$, where $X \approx 1000$

PREREQUISITES:

Observation, dynamic programming

PROBLEM:

You're given N intervals $[L_i, R_i]$ where $L_i < R_i$.

A *good operation* is as follows:
Pick M of these intervals, and $K_i > 1$ distinct integers $Y_{i,1}, Y_{i,2}, \ldots, Y_{i,K_i}$ from the i-th chosen interval such that:

- The product $K_1 \times K_2 \times \ldots \times K_M$ doesn't exceed N.
- M is as large as possible.

The score of a good operation is the product of all chosen $Y_{i,j}$, i.e,

$$\prod_{i=1}^{M} \prod_{j=1}^{K_i} Y_{i,j}$$

Find the sum of scores across all possible good operations.

EXPLANATION:

There are a lot of parts to this problem, so let's analyze them one at a time.

Good operations

First, let's look at when exactly an operation is good.
We pick M intervals and at least 2 elements from each chosen interval.
The product of the numbers of chosen elements shouldn't exceed N, which already tells us that M can't be very large since this product is at least 2^M.
In particular, this gives us the upper bound $M \leq \lfloor log_2 N \rfloor$.

Since M should be maximum, we can always choose $M = \lfloor log_2 N \rfloor$ (we can just pick 2 elements from each interval).
So, M is fixed.

Now, let's look at the K_i values.
Clearly most of them should be 2, but if $2^M < N$ we have *some* leeway.
For example, if $N = 6$ we have $M = 2$, and we have $2 \cdot 2 \leq 6$ and $2 \cdot 3 \leq 6$, which means we can have one of the K_i equal to 3 if we like.

However, despite this leeway, the restrictions are in fact fairly tight.
Specifically, a little analysis should tell you the following:

- Every K_i will be ≤ 3.
- At most one K_i can be 3.

Proof

Suppose $K_i \geq 4$.
Then, we can instead pick 2 elements each from 2 other intervals for a larger M but not larger product, which is surely not optimal.

Similarly, if we have 2 intervals with $K_i = 3$, their product is 9 and we can instead pick 3 intervals and choose 2 from each for larger M and lower product, which once again isn't optimal.

This puts a rather strict structure on what a good operation looks like:

- We always choose exactly $M = \lfloor log_2 N \rfloor$ intervals.
- Then, either we pick two elements from all M of these, or (product allowing) we pick three elements from one of them and two from all the rest.

This allows us to compute the score using dynamic programming.

Computing the sum of scores

There are two cases here: either we pick 2 elements from every interval, or we pick two elements from all but one and 3 from the last one.
Dealing with both is similar, so let's see how to answer the first case (i.e, all twos) first.

To get an idea of how to do do this, let's look at a simple case first: there are two intervals $[L_1, R_1]$ and $[L_2, R_2]$, and we pick exactly two elements from each one.
How would we compute the sum of scores in this case?

Well, suppose we pick the elements x_1, x_2 from the first interval and x_3, x_4 from the second. They add $x_1 x_2 x_3 x_4$ to the sum.
Summing this across all possible values, we want

$$\begin{aligned}
&\sum_{L_1 \le x_1 < x_2 \le R_1} \sum_{L_2 \le x_3 < x_4 \le R_2} (x_1 x_2 x_3 x_4) \\
&= \sum_{(x_1, x_2)} \left(x_1 x_2 \sum_{(x_3, x_4)} x_3 x_4 \right) \\
&= \left(\sum_{(x_1, x_2)} x_1 x_2 \right) \left(\sum_{(x_3, x_4)} x_3 x_4 \right)
\end{aligned}$$

That is, if we know the sum of pairwise products of each the two intervals, all we'd need to do is multiply them.

It's easy to see that this generalizes to M intervals as well: if P_i is the sum of pairwise products of the i-th interval, then what we're looking for is the sum of products of the P_i values taken M at a time.

First, let's compute the P_i values. That can be done with a bit of math.
For an interval $[L_i, R_i]$, the corresponding P_i value is

$$\frac{(L_i + L_{i+1} + \ldots + R_i)^2 - (L_i^2 + L_{i+1}^2 + \ldots + R_i^2)}{2}$$

because:

- $(L_i + L_{i+1} + \ldots + R_i)^2$ is the product of all pairs of values in the interval, disregarding order or distinctness.
- We subtract $(L_i^2 + L_{i+1}^2 + \ldots + R_i^2)$ to account for choosing two of the same element.
- Now the remaining value didn't account for order, i.e, $x_1 x_2$ and $x_2 x_1$ were both counted as different products.
  To account for this, divide by 2.

Once all the P_i values are computed, we want to compute their product taken M at a time.
That can be done in $\mathrm{O}(N \cdot M)$ with a fairly simple DP.

Let $\mathrm{dp}[i][j]$ denote the sum of products all j-tuples of the first i values. Then,
$\mathrm{dp}[i][j] = \mathrm{dp}[i-1][j] + P_i \cdot \mathrm{dp}[i-1][j-1]$
The value we're looking for is, of course, $\mathrm{dp}[N][M]$.

As a small side note, the sum of products of the elements taken M at a time can be computed for *every* M from 1 to N in $\mathrm{O}(N \log^2 N)$ using NTT, but that's extreme overkill for this problem because we're only interested in a single value of M, and the $\mathrm{O}(N \cdot M)$ method is enough because $M = \lfloor \log_2 N \rfloor$.

This almost solves the problem: the only remaining case is when we take exactly three elements from one interval.
You might've noticed that this is not so different from the above case: indeed, once again we only really need to know, for each interval, the sum of products of all distinct triples of elements in it.

This can be computed almost exactly the same way we computed P_i, by using inclusion-exclusion.

Details

Just as before, we start with $(L_i + \ldots + R_i)^3$, disregarding both distinctness and order.

From this, subtract $3 \cdot (L_i + \ldots + R_i) \cdot (L_i^2 + \ldots + R_i^2)$, which is the sum of all triplets such that two elements are equal (fix the equal pair in $\binom{3}{2} = 3$ ways, then it's the sum of all elements multiplied by the sum of squares of all elements).

Next, add $2 \cdot (L_i^3 + \ldots + R_i^3)$, to account for the case when all three elements are equal.
Such cases were subtracted thrice above but should've been subtracted only once; hence the multiplier of 2.

Now we're left with the sum of products of triplets, all of whose elements are distinct.
To account for order, divide this by 6 to get the final answer.

Note that in intermediate steps, we needed to compute the sum of first/second/third powers of a range of integers; there are well-known formulas for these that allow for $\mathrm{O}(1)$ computation.

Once these values are computed for each interval (let's call them Q_i), the earlier dp can be easily modified to account for them.
Let $\mathrm{dp}[i][j][0/1]$ denote the sum of scores for the first i intervals if we've chosen exactly j so far, and the third dimension denotes whether we have/haven't chosen 3 from an interval yet.
Then,

- $\mathrm{dp}[i][j][0] = \mathrm{dp}[i-1][j][0] + \mathrm{dp}[i-1][j-1][0] \cdot P_i$
- $\mathrm{dp}[i][j][1] = \mathrm{dp}[i-1][j][1] + \mathrm{dp}[i-1][j-1][1] \cdot P_i + \mathrm{dp}[i-1][j-1][0] \cdot Q_i$

Once again, this takes $\mathrm{O}(N \log N)$ time.
It can also be implemented with $\mathrm{O}(\log N)$ memory, though that's likely unnecessary to get AC.

As for the final answer,

- If $3 \cdot 2^{M-1} \le N$, we can have an interval with $K_i = 3$. In this case, the answer is $\mathrm{dp}[i][M][0] + \mathrm{dp}[i][M][1]$.
- Otherwise, all intervals must have $K_i = 2$ and the answer is just $\mathrm{dp}[i][M][0]$.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

There is a grid consisting of distinct integers. Two players A and B play a game on it, taking turns moving a token from $(1, 1)$ to (N, M) one step right/down at a time. B moves first.

B can make any available move, while A will always move to whichever neighbor has the higher value.
Let K be the maximum value encountered on the path. If B plays optimally, what is the minimum possible value of K?

EXPLANATION:

First, note that which cell of the grid we are on uniquely defines whose turn it is. For cell (i,j), if $(i+j)$ is even then it is Becky's turn, and if $(i+j)$ is odd it is Anya's turn.

With this in hand, the task can be solved using dynamic programming.

Let $dp(i,j)$ denote the minimum possible value of K if the token starts at cell (i,j) and moves towards (N, M). Our answer is $dp(1, 1)$.

Then,

- If it is Anya's turn, $dp(i,j) = \max(A_{i,j}, dp(next))$, where $next$ denotes the unique cell Anya will move to from (i,j), i.e,
  - $next = (i+1, j)$ if $A_{i+1,j} > A_{i,j+1}$
  - $next = (i, j+1)$ otherwise.
- If it is Becky's turn, $dp(i,j) = \max(A_{i,j}, \min(dp(i+1,j), dp(i,j+1)))$

The reasons for these transitions should be obvious:

- On Anya's turn, she will definitely move to $next$. The maximum of the path from (i,j) is thus either $A_{i,j}$ or the maximum of the path from $next$.
- On Becky's turn, the maximum of the path is obviously at least $A_{i,j}$. Beyond that, she will choose whichever move minimizes the maximum. There are only two possible moves so we just take their minimum.

This can either be implemented as a recursive function with memoization (with the base case being $dp(N, M) = A_{N,M}$, or an iterative dp where you simply iterate across the grid from bottom to top, left to right. Either way, the complexity is $\mathrm{O}(N \cdot M)$.

TIME COMPLEXITY

$\mathrm{O}(N \cdot M)$ per test case.

PREREQUISITES:

Greedy, dynamic programming

PROBLEM:

There are two lanes L_1 and L_2, each containing N blocks, and L_1 is on top of L_2. Chef starts running from the beginning of (any) one of the lanes and must reach the end of any lane. To do so, Chef can use the following jumps (assuming he's currently at the x th block of some lane):

- Go to the $(x+1)$th block of the same lane.
- Switch gravity and go to the x th block of the other lane.
- Switch gravity and go to the $(x+1)$th block of the other lane.

Some of the blocks are dirty, which means Chef cannot step over them.

You need to know whether Chef can reach the end and if so, what is the minimum number of **gravity switches** necessary.

QUICK EXPLANATION:

The second kind of jump is not helpful, so we only use the first and third. It's also not helpful to switch lanes unless there is an obstacle to avoid. With this in mind, there is one obvious optimal path that we should consider (i.e. switch lanes only if necessary):

- There is no path to the end path if and only if there is a column where both lanes contain dirty blocks.
- If there is a lane containing no dirty blocks, then the minimum number of gravity switches is 0.
- Otherwise, start at the lane whose first dirty block appears later. Only switch lanes if the next block in the current lane is dirty. The number of steps taken is the answer.

There's also a dynamic programming approach: let $D_1(x)$ and $D_2(x)$ be the fastest way to reach the x th block of lanes L_1 and L_2, respectively. Then:

- The answer is $\min(D_1(N), D_2(N))$
- We have the recurrences:

$$D_1(x) = \begin{cases} \infty & \text{if } L_1(x) \text{ is dirty} \\ \min(D_1(x-1), D_2(x-1)+1) & \text{otherwise} \end{cases}$$

$$D_2(x) = \begin{cases} \infty & \text{if } L_2(x) \text{ is dirty} \\ \min(D_2(x-1), D_1(x-1)+1) & \text{otherwise} \end{cases}$$

We define $D_1(0) = D_2(0) = 0$ as base cases.

By making two arrays D_1 and D_2 of length N, we can compute these values by increasing x in $O(N)$ time.

EXPLANATION:

Let's give some names to the different kinds of jumps.

- **Forward**: Go to the $(x+1)$th block of the same lane.
- **Quick switch**: Switch gravity and go to the x th block of the other lane.
- **Slow switch**: Switch gravity and go to the $(x+1)$th block of the other lane.

The first thing to notice is that “quick switch” is not helpful. Let’s see why this is so, by considering the various possible jumps immediately after doing a quick switch. Let’s assume that Chef is at position $L_1(x)$ (the case $L_2(x)$ is essentially the same).

- If the next jump is a “forward”, then you went from $L_1(x)$ to $L_2(x+1)$ with one gravity switch. But this is just like doing a “slow switch”! If we do a single “slow switch” instead, the number of gravity switches stays the same, and we skipped passing through the block $L_2(x)$ (which is actually better in case $L_2(x)$ is dirty).
- If the next jump is a “slow switch”, then you went from $L_1(x)$ to $L_1(x+1)$ with two gravity switches. But we can instead do a single “forward” move without doing any gravity switches! (and we skipped passing through the block $L_2(x)$ which is helpful as explained in the previous bullet)
- If the next jump is another “quick switch”, then you just went back to your original position while incurring two gravity switches. Clearly, doing two quick switches in a row is just a waste of effort.

Thus, we have shown that forwards and slow switches are the only kinds of jumps we have to consider.

A greedy approach

The remaining kinds of jumps have the property that each jump takes Chef one column to the right. In fact, we can derive a few more properties of optimal paths:

- It’s not optimal to slow switch if there is no obstacle to avoid. More specifically, if you did a slow switch, then a sequence of k forwards, and another slow switch, but there weren’t any dirty blocks that were avoided in the process, then you can just replace that sequence with a sequence of $k+2$ forward switches. This saves you two gravity switches 🙂
- It’s always better to start at the lane whose first dirty block appears later, or doesn’t appear at all. This is because if you start at the other lane, you are forced to do a slow switch which you could have avoided by simply starting at the other lane.

So we now have the following **greedy** solution to the problem:

- There is no path to the end path if and only if there is a column where both lanes contain dirty blocks.
- If there is a lane containing no dirty blocks, then the minimum number of gravity switches is 0.
- Otherwise, start at the lane whose first dirty block appears later, and simulate the path by trying to do only “forward” steps. Only use “slow switch” if the next block is dirty. The number of steps taken is the answer.

A dynamic programming approach

Another standard way to approach the problem is to use the ever-powerful **dynamic programming**. For some people this is simpler, because it involves less thinking about the shape of the optimal path.

Let’s define the **distance** of a block to be the minimum number of gravity switches needed to reach that block (in case the block is unreachable, we say the distance is ∞). Then let $D_1(x)$ and $D_2(x)$ be the distances of blocks $L_1(x)$ and $L_2(x)$, respectively.

Notice that the final answer is simply $\min(D_1(N), D_2(N))$, because we want to reach either $L_1(N)$ or $L_2(N)$ as fast as possible. Now, let’s focus on $D_1(x)$.

If $L_1(x)$ is dirty, then of course there’s no way to reach that cell (you can’t even step on it!), so we immediately know that $D_1(x) = \infty$. Otherwise, the last move must have been either a “forward” or a “slow switch”.

- If it was a forward, then you arrived at $L_1(x-1)$ to get to $L_1(x)$. The minimum number of gravity switches required for this is $D_1(x-1)$.
- If it was a slow switch, then you arrived at $L_2(x-1)$ to get to $L_1(x)$. The minimum number of gravity switches required for this is $D_2(x-1)+1$ (the $+1$ is due to the last move which uses one gravity switch).

Therefore, we have the following:

$$D_1(x) = \begin{cases} \infty & \text{if } L_1(x) \text{ is dirty} \\ \min(D_1(x-1), D_2(x-1)+1) & \text{otherwise} \end{cases}$$

The case is very similar for $D_2(x)$, i.e.:

$$D_2(x) = \begin{cases} \infty & \text{if } L_2(x) \text{ is dirty} \\ \min(D_2(x-1), D_1(x-1)+1) & \text{otherwise} \end{cases}$$

These formulas enable us to compute $D_1(x)$ and $D_2(x)$ from $D_1(x-1)$ and $D_2(x-1)$ recursively!

But what about the base cases? We can compute $D_1(1)$ and $D_2(1)$ easily by considering that Chef can start at either $L_1(1)$ or $L_2(1)$ without doing any move (as long as the block is not dirty of course!). Thus:

$$D_1(1) = \begin{cases} \infty & \text{if } L_1(1) \text{ is dirty} \\ 0 & \text{otherwise} \end{cases}$$

$$D_2(1) = \begin{cases} \infty & \text{if } L_2(1) \text{ is dirty} \\ 0 & \text{otherwise} \end{cases}$$

But we can make the base case simpler by **adding a clean block at the beginning of both lanes**! This doesn't worsen the solution, because you can just start at the same lane you would have started originally, and do a "forward" move, without incurring additional gravity switches. Thus, we define two new blocks $L_1(0)$ and $L_2(0)$ as clean blocks, and define $D_1(0) = D_2(0) = 0$ as the base cases!

We now have all the ingredients for our solution. First, we build two (0-indexed) arrays, D_1 and D_2, each of length $N+1$. Then, set $D_1[0] = D_2[0] = 0$. Next, compute $D_1[x]$ and $D_2[x]$ for $1 \le x \le N$ in increasing order, based on the recurrences above. Finally, the answer is now $\min(D_1[N], D_2[N])$! (If this value is infinite, then the answer is `No`)

Implementation details / optimizations

Handling ∞

Notice that the value ∞ is used in our arrays D_1 and D_2. But most builtin types (such as `int` or `long`) do not have infinite values. Here are possible ways to handle that:

- Use a dummy value such as "-1" in place of infinity. I don't recommend this though, because you'll have to check for the value "-1" all the time (for example, you need to modify the `min` function because infinity should be larger than all other elements).
- Define "infinity" to be some large value. Some possibilities are 10^9, 2^{30}, 2^{60} or the data type's upper limit. This has the advantage that comparisons of infinity against normal values are correct. However, one should be careful in comparing "infinite" values against each other. Also, be careful in doing arithmetic with this "infinity", because you might incur overflow!
- Use the "float" or "double" data type which have infinite values.

I prefer the second solution.

Memory-efficient DP

The solution above requires us to create two arrays, D_1 and D_2. However, notice that when computing $D_1[x]$ and $D_2[x]$, we only need the values of $D_1[x-1]$ and $D_2[x-1]$. Therefore, we only need to store the current

and previous values of D_1 and D_2, reducing the memory requirements from $O(N)$ to $O(1)$ (disregarding the memory where the input is stored)!

Sample implementations

Finally, here we provide some implementations of the solution. The DP solutions use a large number for “infinity”, and also use $O(1)$ additional memory.

Python (Greedy approach):

```
INF = 1<<30 # some really large number
for cas in xrange(input()):
    L1 = raw_input()
    L2 = raw_input()
    ans = 0
    curr = 0
    for L1x, L2x in zip(L1, L2):
        if L1x == L2x == '#':
            ans = INF
            break

        if L1x == '#':
            if curr == 1:
                ans += 1
            curr = 2
        if L2x == '#':
            if curr == 2:
                ans += 1
            curr = 1

    if ans >= INF:
        print "No"
    else:
        print "Yes"
        print ans
```

C++ (Greedy approach):

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define LIM 200011
#define INF LIM<<3

char words[2][LIM];
int main() {
    int z;
    scanf("%d", &z);
    while (z--) {
        scanf("%s%s", words[0], words[1]);
        int n = strlen(words[0]);
        int curr = -1, ans = 0;
        for (int i = 0; i < n; i++) {
            bool dirty0 = words[0][i] == '#';
            bool dirty1 = words[1][i] == '#';
            if (dirty0 && dirty1) {
                ans = INF;
                break;
            }
```

```
            if (dirty0) {
                if (curr == 0) ans++;
                curr = 1;
            }
            if (dirty1) {
                if (curr == 1) ans++;
                curr = 0;
            }
        }
        if (ans >= INF) {
            printf("No\n");
        } else {
            printf("Yes\n%d\n", ans);
        }
    }
}
```

Python (DP approach):

```
INF = 1<<30 # some really large number
for cas in xrange(input()):
    L1 = raw_input()
    L2 = raw_input()
    D1 = D2 = 0
    for L1x, L2x in zip(L1, L2):
        D1, D2 = (
            INF if L1x == '#' else min(D1, D2 + 1),
            INF if L2x == '#' else min(D2, D1 + 1),
        )
    ans = min(D1, D2)
    if ans >= INF:
        print "No"
    else:
        print "Yes"
        print ans
```

C++ (DP approach):

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define LIM 200011
#define INF (LIM<<3)

char L[2][LIM];
int D[2];
int nD[2];
int main() {
    int z;
    scanf("%d", &z);
    while (z--) {
        scanf("%s%s", L[0], L[1]);
        int n = strlen(L[0]);
        D[0] = D[1] = 0;
        for (int i = 0; i < n; i++) {
            nD[0] = L[0][i] == '#' ? INF : min(D[0], D[1] + 1);
            nD[1] = L[1][i] == '#' ? INF : min(D[1], D[0] + 1);
            D[0] = nD[0];
            D[1] = nD[1];
```

```
        }
        int ans = min(D[0], D[1]);
        if (ans >= INF) {
            printf("No\n");
        } else {
            printf("Yes\n%d\n", ans);
        }
    }
}
```

Time Complexity:

$O(N)$

PREREQUISITES:

Prefix sums, dynamic programming

PROBLEM:

Given a binary array A, find the longest *good* sequence $x_1, x_2, \ldots, x_k$ such that it satisfies:

- $1 \le x_1 < x_2 < \ldots < x_k \le N + 1$
- $A[x_i : x_j - 1]$ contains an equal number of zeros and ones for each $i < j$.

EXPLANATION:

First, note that it's enough to ensure that $A[x_i : x_{i+1} - 1]$ (i.e, the subarray between consecutive elements) satisfies the condition, i.e, contains an extra one.

Proof

This is not hard to see.
The subarray $A[x_i : x_j - 1]$ can be thought of as the concatenation of $A[x_i : x_{i+1} - 1], A[x_{i+1} : x_{i+2} - 1], \ldots, A[x_{j-1}, x_j - 1]$. Note that there are $j - i$ such subarrays.

If each of these contains an extra one, of course all together the number of extra ones will be exactly $j - i$.

Now, let's use a small trick: replace every 0 in the array with -1. Let's call the array obtained this way as B.
For example, if $A = [0, 1, 1, 0, 1]$ then $B = [-1, 1, 1, -1, 1]$.
Notice that "$A[L : R]$ contains an extra one" translates to "$B[L : R]$ has a sum of 1" in the new array, which is much easier to deal with!

Since we're dealing with subarray sums, it's natural to think of prefix sums.
So, let $P_i = B_1 + B_2 + \ldots + B_i$ (with $P_0 = 0$) denote the prefix sum array of B.

Now, for $B[L : R]$ to have a sum of 1, we must have $P_R - P_{L-1} = 1$, or $P_R = P_{L-1} + 1$.

Applying this to the definition of a good sequence, we see that we must have $P_{x_{i+1}-1} = P_{x_i - 1} + 1$ for every $1 \le i < k$.
In other words, we need to choose a sequence of indices whose prefix sums are $k, k+1, k+2, \ldots$ for some value of k.

The longest good sequence thus corresponds to the longest sequence of prefix sums of this form.

This can be computed using dynamic programming.

Let dp_i denote the length of the longest sequence of prefix sums ending at position i that satisfies the above condition.
Transitions for a given index are quite easy to compute in $\mathrm{O}(N)$. We simply have $dp_i = 1 + \max(dp_j)$ across all $j < i$ such that $P_j + 1 = P_i$.

However, doing this $\mathrm{O}(N)$ computation for each index is too slow, we need to speed it up a bit.
This can be done by noting that we only care about those indices whose P_j values equal $P_i - 1$; in fact, we only care about the maximum dp_j value among these indices.

So, let's keep another array mx, where $\mathrm{mx}[x]$ is the largest value of dp_i across all indices i such that $P_i = x$.
Initially, initialize it to all zeros.
Then, for each i from 0 to N,

- $dp_i = 1 + \mathrm{mx}[P_i]$

- $\text{mx}[P_i] = \max(dp_i, \text{mx}[P_i])$

The maximum element of dp is the final answer.
Reconstructing a valid sequence is not hard, when performing the dp keep a link to the previous element in the sequence, then follow these in reverse at the end.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PREREQUISITES:

DP on trees

PROBLEM:

Given a tree T on N vertices, find the minimum number of edges that need to be added to it so that the resulting graph contains a Hamiltonian path.

EXPLANATION:

Suppose we add some edges to T so that it has a Hamiltonian path.
Notice that the tree edges used in this Hamiltonian path form a set of vertex-disjoint paths in the original tree.

Let's call a set of vertex-disjoint paths a *path forest*.

Of course, minimizing the number of extra edges we use is the same thing as maximizing the number of tree edges forming the complementary path forest; so it seems reasonable that we compute the maximum size of a path forest that can be obtained from the given tree.

Proof of equivalence

Let's call a graph *good* if it contains a Hamiltonian path.
Our claim is that keeping the largest number of edges in a path forest is equivalent to minimizing the number of edges added to form a good graph.
Alternately, the minimum number of edges that need to be *removed* to obtain a path forest equals the minimum number of edges added to form a good graph; which is what we'll prove.

Let P be one minimum (by size) set of edges that, upon removal, form a path forest.
Let H be one minimum (by size) set of edges that, upon addition to T, make it good.

Note that the path forest can be connected into a single long path (thus making the resulting graph good) using $|P|$ additional edges (since removing $|P|$ edges from a tree will create exactly $|P| + 1$ connected components).
So, $|H| \le |P|$.

Conversely, suppose we add the edges in H.
Consider one Hamiltonian path obtained by this addition.
Suppose there are k tree edges on this path. Then, $|H| = N - 1 - k$ (because H wouldn't be minimal otherwise).
Deleting the $N - 1 - k$ *tree* edges that are not on this path gives us a path forest, so $|S| \le N - 1 - k = |H|$.

This shows that $|S| = |H|$, and we're done.

Now, notice that in a path forest, every vertex will have degree ≤ 2. This allows us to compute the largest size of a path forest using dynamic programming.

Root the tree at vertex 1.
Then, let $dp(u, d)$ denote the largest size of a path forest in the subtree of vertex u, such that u has degree $\le d$ in this forest.
Transitions are not too hard:

- $dp(u, 0)$ is simply the sum of $dp(v, 2)$ across all children v of u.
- $dp(u, 1)$ requires us to connect u to one of its children.
  - Suppose we connect it to child c. Then, the size of the path forest is $1 + dp(c, 1) + \sum_{v \neq c} dp(v, 2)$, which also equals $1 + dp(c, 1) - dp(c, 2) + \sum_v dp(v, 2)$.
  - Maximizing this quantity is the same as maximizing $dp(c, 1) - dp(c, 2)$, so take the maximum value across all c and we're done.
- $dp(u, 2)$ requires us to connect u to two of its children.

  - Once again, it's not hard to see that this requires us to take the two largest values of $dp(c, 1) - dp(c, 2)$ across all c; and add them to $dp(u, 0) + 2$.
- Finally because our states were defined to use degree $\leq d$, set $dp(u, 1) = \max(dp(u, 0), dp(u, 1))$ and $dp(u, 2) = \max(dp(u, 2), dp(u, 1), dp(u, 0))$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Dynamic programming

PROBLEM:

N students are in a circular hostel with M rooms.
Initially, student i is in room A_i.
For each i and j, you also know the value $H_{i,j}$: the happiness of student i if he were to stay in room j.

You can perform the following operation:

- Choose two students i and j.
  Move i clockwise by X rooms and j counterclockwise by X rooms.

Find the maximum possible total happiness of the students after performing some moves.

EXPLANATION:

Working with pairs of students is somewhat hard.
Observe that we can instead work with single students and fix how many moves each one makes: it doesn't really matter who is paired with who as long as the total number of counterclockwise moves equals the total number of clockwise moves — the end result will be the same and that's all that matters.

This gives us something to start with.
Let $\text{dp}[i][j]$ denote the maximum possible happiness if we've considered students $1, 2, \ldots, i$; and there are j 'unpaired' moves.
For convenience, if $j > 0$ we say that j anticlockwise moves must be made, while $j < 0$ means $-j$ clockwise moves must be made.

Then,

- If student i makes k clockwise moves, he will end up at room $(A_i + k \cdot X) \bmod M$.
  Let h be his happiness at ending up in this room.
  Then the value we obtain here is $\text{dp}[i-1][j-k] + h$.
- If student i makes k anticlockwise moves, he will end up at room $(A_i - k \cdot X) \bmod M$.
  Again, if h is the happiness of this, the overall value is $\text{dp}[i-1][j+k] + h$.

By considering all possible values of k, $\text{dp}[i][j]$ can be computed as the maximum among the above.
The final answer is $\text{dp}[N][0]$: the best value after all students have been considered, and no excess moves remain.

While this formulation is correct, it has a large flaw: it's too slow.
In fact, the number of moves that a single student can make might have to be very large, so we have to constrain this somehow.

For that, we use the cyclicity property we have.
We can make the following observations:

- First, it's enough to consider $\text{dp}[i][j]$ values for $-M < j < M$.
  This is because, if $j \geq M$, student i can just make M counterclockwise moves to reduce j by M while still remaining at his initial position. Repeat this till $j < M$.
- Secondly, for similar reasons, it's enough to consider k (the number of moves made) to also be from 0 to $M-1$; since if we make more, the count can be reduced by M without affecting the result.

This immediately gives us a more tractable solution.
There are $\text{O}(N \cdot M)$ states of $\text{dp}[i][j]$, and for each one we perform M transitions for $\text{O}(NM^2)$ in total, which

should already be enough to solve the problem.

This is enough to get AC if implemented well, though further optimizations can be made to improve runtime:

- Instead of $-M < j < M$, it's enough to consider $0 \le j < M$.
  This is because j clockwise moves can also be simulated using $M - j$ anticlockwise moves.
- One nice property of repeatedly adding X on cycles like this is that we in fact obtain $\gcd(M, X)$ disjoint cycles, each of length $\frac{M}{\gcd(M,X)}$ and rather equally spaced out.
  Each element, when X is added/subtracted to it, only moves within its own cycle.
  This allows us to bring the complexity down to $\mathrm{O}\left(N \cdot \left(\frac{M}{\gcd(M,X)}\right)^2\right)$, though the worst case complexity (when $\gcd(M, X) = 1$) remains the same.

TIME COMPLEXITY

$\mathrm{O}(NM^2)$ per testcase.

PREREQUISITES:

Dynamic Programming

PROBLEM:

in every century, there is a chosen one. Therefore, we do not live in a society
–Last paragraphs of the IZhO 2022 Editorial

K0Kalaru47 was out with his friends and he started to tell them about a CP Task. Because the statement was informal, his friends quickly parried with "Who Asked", so here is the formal statement:

Given arrays P and C of size $(N+K)$ each, the *beauty* of an array B of size $(N+K)$ is defined as the sum of C_i for all $1 \le i \le (N+K)$ such that $B_i = P_i$.

More formally, the *beauty* of an array B is equal to $\sum_{i=1}^{|B|} C_i \, [B_i = P_i]$

Given an array A of size N such that $(1 \le A_i \le M)$.
Your task is to insert K elements into the array such that:

- Each inserted element lies in the range $[1, M]$.
- The *beauty* of the final array is **maximized**.

Note that, an element can be inserted at any index of the array.

You should find two values:

- The **maximum** beauty that can be obtained across all arrays of size $(N+K)$.
- The **number of distinct arrays** of size $(N+K)$ having the maximum beauty. Note that, two arrays X and Y are said to be different if there exists some index i such that $X_i \neq Y_i$. Since the number of distinct arrays can be huge, print the value modulo MOD.

QUICK EXPLANATION:

We can say some final vector B can be obtained from the initial vector if and only if

- All number in B are $[1, M]$.
- Initial vector A is a subset of final vector B.

We can check if the initial vector is a subset of final vector using the following greedy approach.
FIrst pick a pointer $p = 1$. Loop through the array, if at any i, $B_i = A_p$, increment p by 1. If at the end $p = |A| + 1$, then and only then is A a subset of B. This is also known as the Subset Search Algorithm
One important thing to note here is that the subset it finds is always unique. Using this we can define our $dp[i][j]$ to store the maximum beauty and the number of arrays to have that beauty.

One more thing we can notice is that we will form array B by only using the following values:

- P_i
- A_j
- A_{j+1}
- Any other value other than above three.

Thus we can only try these values and have $O(1)$ transitions. Our final answer would be then $dp[n+k][n]$.

EXPLANATION:

Let us construct our $dp[i][j]$ and define it to store two values as:

- The maximum beauty of an array that has length i and the subset search algorithm found the first j values of vector A.
- The number of ways to have that beauty for an array that has length i and the subset search algorithm found the first j values of vector A.

Now let us define the transitions of our dp as follows:

- if $p[i] = a[j]\ and\ a[j] = a[j+1]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j-1].first + C[i], dp[i-1][j-1].second\}, \{dp[i-1][j].first, dp[i$$

- Else if $a[j] \neq a[j+1] \& a[j] \neq p[i] \& a[j+1] \neq p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first + C[i], dp[i-1][j].second\}, \{dp[i-1][j].first, dp[i-1][j].$$

- Else if $a[j] = a[j+1]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first + C[i], dp[i-1][j].second\}, \{dp[i-1][j].first, dp[i-1][j].$$

- Else if $a[j] = p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j-1].first + C[i], dp[i-1][j-1].second\}, \{dp[i-1][j].first + C[$$

- Else if $a[j+1] = p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first, dp[i-1][j].second * (m-2)\}, dp[i-1][j], dp[i-1][j-1]$$

TIME COMPLEXITY:

$O((N+K) \cdot N)$ for each test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

A permutation is K-beautiful if it can be partitioned into K subarrays such that each subarray consists of elements of the same parity, and cannot be partitioned into $< K$ subarrays satisfying this condition.

You are given a permutation with some of the elements replaced by 0's. How many ways exist to replace the 0's such that the resulting sequence is a K-beautiful permutation?

EXPLANATION:

The main observation here is that every odd number is essentially the same since we only care about parity. That is, it doesn't matter whether we replace a 0 with a 1 or a 3 — only whether we replace it with an even number or an odd number. So, first compute E and O: the total number of unused even and odd integers respectively.

This leads to a dynamic programming solution: we need to keep track of where we are, how many subarrays we have created so far, how many odd/even numbers have been used to replace zeros with so far, and what the parity of the previous element is (to know whether its subarray can be extended or not).

Let $f(pos, ct, ev, od, par)$ denote the number of ways to replace zeros in the first pos positions using od odd integers and ev even integers, such that exactly ct maximal subarrays have been created so far, and the parity of the integer placed at pos is par.

This gives us a somewhat simple recurrence:

- First, suppose $P_{pos} \neq 0$. Then, the parity at this position is fixed, and we don't do any replacements so ev and od do not change. So,
  - If $par \neq parity(P_{pos})$, then $f(pos, ct, ev, od, par) = 0$.
  - Otherwise, $f(pos, ct, ev, od, par) = f(pos-1, ct, ev, od, par) + f(pos-1, ct-1, ev, od, 1-par)$. That is, if the previous position had the same parity then the number of maximal subarrays (ct) doesn't change, and if it had a different parity the number of subarrays increases by 1.
- Now, suppose $P_{pos} = 0$. There are two cases to consider: replacing this position with an even integer, and replacing it with an odd integer. I'll discuss the odd case, the even one follows similarly.
  - First, note that we have $\max(0, O - od + 1)$ choices for which odd integer to place here (since $od - 1$ were used in previous positions).
  - Then, the recurrence looks similar to the above case: either the parity of the previous position is the same (in which case ct doesn't change), or it is different (in which case ct increases by 1).
  - So, $f(pos, ct, ev, od, 1) = \max(0, O - od + 1) \cdot (f(pos-1, ct, ev, od-1, 1) + f(pos-1, ct-1, ev, od-1, 0))$

Our final answer is, of course, $f(N, K, E, O, 0) + f(N, K, E, O, 1)$.

Now, note that we have around $2N^4$ states in our dp. Even with memoization to make transitions $\mathrm{O}(1)$, this results in a $\mathrm{O}(N^4)$ time solution overall, which is too slow.

To speed it up, we can use a somewhat common dp trick: reduce the number of states by looking for a relation between them.

In our case, ev and od are related: note that any relevant state must have $ev + od$ equal to the number of zeros in positions $\leq pos$ (since we *must* replace every zero).

The number of zeros in a prefix is fixed, and thus knowing one of ev or od immediately tells us the other one. So, instead of having both ev and od in our state, it is enough to have only one of them.

This reduces our number of states to $2N^3$, and transitions are still $\mathrm{O}(1)$ so this is fast enough to pass.

TIME COMPLEXITY

$\mathrm{O}(N^3)$ per test case.

PREREQUISITES:

Dynamic Programming, Knowledge of inversions and permutations.

PROBLEM:

We are given Q queries of the following type:
Print the Kth lexicographically smallest permutation of $[1, N]$ with exactly R inversions.
$N \leq 500$
$R \leq 200,000$
$K \leq 10^{18}$
$Q \leq 2000$

QUICK EXPLANATION:

Initially, build a table $dp[N_{max}][R_{max}]$, where $dp[i][j]$ means number of permutations of length i with exactly j inversions. Reconstruct the solution for each query: greedily take the smallest possible value for each index, as long as enough number of permutations can be constructed while considering the rest of the constraints.

DETAILED EXPLANATION:

I will elaborate on the solution in terms of step-by-step observations.

Initial thoughts about complexity

Since N is just 500 and even R is atmost $2 * 10^5$, nothing much can be guessed about the expected complexity. However, we can say that K will definitely not be in our complexity analysis.

First thoughts

Often, a good way to approach such problems, where we have to print K^{th} lexicographically smallest something, is to think if we can keep greedily fixing elements, as long as the reduced parameters allow for at least X more permutations. Maybe we can do so here?

Observation 1

A curious property is that we can calculate how many permutations of length M $(M < N)$ are possible with R_m $(R_m < R)$ inversions, without needing to know what the first $N - M$ values are.

More explanation

What do I mean by that? Let's say we are dealing with the numbers $\{1, 2, 3, 4, 5\}$ and want to see how many permutations with y inversions are possible. Lets say we fix the first number as 3. Now we are left with $\{1, 2, 4, 5\}$. However, note that the gap is not important really in our calculation of how many permutations with (let's say) x inversions we can get from this set. Our set is essentially the same as $\{1, 2, 3, 4\}$ for all purposes since while counting inversions, we only need to know greater than or less than values. Hence for our purposes, any 4 distinct numbers would have given the same number of permutations.

this hints towards some precomputation. In particular, maybe if we calculate a table $dp[N_{max}][R_{max}]$, where $dp[i][j]$ means number of permutations of length i with exactly j inversions, perhaps it will help us later. Note that we will be calculating this table as a precomputation step, without processing any queries. We don't need to recompute anything during the queries since for a particular i and j, $dp[i][j]$ is always constant, irrespective of the values already taken.

Building the DP

The DP table can be built on the following recurrence:

$$dp[n][r] = \sum_{k=1}^{min(n,r+1)} dp[n-1][r-(k-1)]$$

The rationale behind arriving at this $dp[.][.]$ recurrence is that we are essentially trying to place k as the first element in our permutation, and considering the rest of the premutation of $n-1$ elements with $r-(k-1)$ inversions.

Full Solution
Hint:

Try to recreate the Kth smallest permutation, using the $dp[.][.]$ table.

Recreating: Details

Let's say we have a function `getSmallest(int N, int R)` which returns the optimal number that should be placed at the start of the N length permutation. Let's say the number placed is x. Observe, this contributes exactly $x-1$ to the total number of inversions R. Hence, $R_{new} = R-(x-1)$. Now we call `getSmallest(N-1,Rnew)`, and continue.

Now, also observe that the optimal x is also the smallest value possible which ensures that $dp[N-1][R_{new}] \geq K$. (unless obviously N=1, in which case it doesn't matter). This means that had we put some value less than x, then we would have had $< k$ permutations formable using the given constraints.

Implementation 1

A quick thing to note is that the number of permutations will be very VERY large. Hence a trick will be to store any value larger than 10^{18} as -1, as the maximum value of K is 10^{18}, and hence any value larger than that is all the same since we just have to compare it to K while restoring. This means that while computing the $dp[.][.]$, we cannot use prefix sums to make the transitions faster. Instead, we just add the values one by one. This makes for a $O(N)$ transition, and since we have $O(N^3)$ states, the total time taken should be $O(N^4)$. However, if we break out of the loop the first time we encounter a -1, the number of iterations reduces *drastically*, and it works really fast. Intuition behind this is that the number of permutations with a even say 20 inversions and $N=50$ (say), is really *really* large. Hence a large portion of the $dp[.][.]$ table is essentially filled with -1.

Time Complexity

We have a precomputation step of $O(N^4)$, which is effectively $O(N^3)$ amortised due to the break statements. During the queries, we, in worst case, will be looping over all possible numbers in `getSmallest(...)`. Since we call it $O(N)$ times, our time complexity becomes $O(Q*N^2)$. However,

again note that in most cases, most of the first elements of the permutation will just be $1, 2, 3, 4, ...$ for larger values of N. Hence, it runs *much* faster in practice.

Challenge:

Do the queries in $O(NlogN)$ per query.

QUICK REMINDERS:

Keep using those **long long ints**.

PREREQUISITES:

Dynamic programming

PROBLEM:

You're given a string S consisting of lowercase Latin characters.
Find a bracket sequence T of the same length as S such that:

- T is a regular bracket sequence
- If $S_i = S_{i+1}$, then $T_i = T_{i+1}$

EXPLANATION:

The condition $S_i = S_{i+1} \implies T_i = T_{i+1}$ means that each contiguous block of equal characters in S must correspond to a contiguous block of either `(` or `)`, of the same length, in T.

So, let's find all the maximal contiguous blocks of equal characters in S.
Suppose there are k such blocks, and their lengths are $L_1, L_2, \ldots, L_k$ in order.
For example, if $S = \text{aabacccbbddd}$ then $L = [2, 1, 1, 3, 2, 3]$.

Now, for each of these blocks, we need to decide whether it corresponds to `(` or `)`.
There are 2^k possibilities, and since k can be as large as N, checking each one is out of the question.

Instead, note that T is a regular bracket sequence iff:

- Each prefix of T has a non-negative balance.
- T as a whole has balance 0.

Here, the balance of a bracket sequence equals the number of opening parentheses it contains, minus the number of its closing parentheses.
For example, the balance of $()$ and $)($ are both 0, while $)(()($ has balance $3 - 2 = 1$.

This allows us to only have to deal with prefixes; in particular, we only really care about the balance of each prefix.
This reduction lets us use dynamic programming.

Let $\text{dp}[i][j]$ be a boolean value: `True` if there is a way to assign `(` and `)` to the first i blocks such that resulting balance is exactly j; and false otherwise.
Note that we only care about $j \geq 0$, since the balance cannot be allowed to go negative.

With this definition, transitions are in fact quite straightforward.
There are two choices for the i-th block: `(` or `)`. Trying each one,

- If the i-th block is turned into `(`, it will increase the balance by exactly A_i.
  So, the part before it must have a balance of $j - A_i$, and we can check whether this is possible by looking at $\text{dp}[i][j - A_i]$
- Similarly, if the i-th block is turned into `)`, we need to look at $\text{dp}[i-1][j + A_i]$.
- So, we have $\text{dp}[i][j] = \text{dp}[i-1][j - A_i] \mid \text{dp}[i-1][j + A_i]$, where $\mid$ denotes the bitwise OR operator.

Of course, the final answer is `Yes` if $\text{dp}[k][0]$ is `True`, and `No` otherwise; because the overall balance after all k blocks are assigned, should be 0.

If a solution exists, reconstructing it is not hard: move from the last block to the first, and each time the stored dp values will tell you which transition you took; and hence which type of bracket to assign to the current block.

There are upto $\mathrm{O}(N^2)$ states and $\mathrm{O}(1)$ transitions for each, for $\mathrm{O}(N^2)$ time overall.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per testcase.

PREREQUISITES:

Dynamic programming

PROBLEM:

Given a tree on N nodes where node i has value A_i, for each u from 1 to N compute

$$\sum_{i=1} \lfloor \frac{A_i}{2^{d(u,i)}} \rfloor$$

EXPLANATION:

The main observation here is as follows: if $d(u,i) > 20$, then $\lfloor \frac{A_i}{2^{d(u,i)}} \rfloor = 0$ no matter what the value of A_i is, since $A_i \leq 10^6$.

This means that we only care about vertices at distances ≤ 20 from a given u.
Of course, this doesn't directly solve the problem, but it's a start.

Let's root the tree at some node, say 1.
With this root, let $p^i(u)$ denote the i-th ancestor of u. In particular, $p^0(u) = u$ and $p^1(u)$ is the parent of u.

Note that u contributes a value of $\lfloor \frac{A_u}{2^i} \rfloor$ to $p^i(u)$, and vice versa.

In particular, this allows us to, at least, compute the answer for every u when only considering values that lie in its subtree: for each u, add $\lfloor \frac{A_u}{2^i} \rfloor$ to the answer of $p^i(u)$ for each i from 0 to 20.
This takes $\mathrm{O}(20N)$ time.

Now, let's look at a specific u. We've already computed the contribution of things in its subtree, so we need to look outside.
So, let's look at $p^1(u)$. Consider some node v in the subtree of $p^1(u)$, that is *not* in the subtree of u.
If $d(v, p^1(u)) = k$, then $d(v,u) = k+1$. Can we use this in some way?

Yes, we can!
Let's compute a 3D dynamic programming table: $dp[u][k][x]$ stores the following:

- Consider the subtree of vertex u, and all nodes at a distance of k from u in this subtree.
- $dp[u][k][x]$ holds the contribution of such nodes to a node at a distance of x from u.

So, coming back to our earlier discussion, the contribution of nodes in the subtree of $p^1(u)$ to the answer of u can be contributed using $dp[p^1(u)][k][1]$ across all k.
Note that this will also include some values in the subtree of u, which shouldn't be counted: their contribution can be subtracted out separately using the appropriate cell in the dp table.

Note that this allows us to visit every relevant ancestor of u and do the same thing. That is, for each $0 \leq i \leq 20$, visit $p^i(u)$ and add the values of $dp[p^i(u)][k][i]$ across all k, while also subtracting appropriate dp values to ensure that nothing is double-counted.
This will cover every node that is at a distance of ≤ 20 from u, which is exactly what we wanted.

The algorithm given above takes $\mathrm{O}(20^2 \cdot N)$ time and space.
It's possible to optimize the space to $\mathrm{O}(20N)$, but this optimization was unnecessary to get AC.

TIME COMPLEXITY

$\mathrm{O}(20^2 \cdot N)$ per test case.

PREREQUISITES:

Dynamic Programming , Permutations and Combinations.

PROBLEM:

You are given an integer array A of size N and a positive integer M. This array contains a hidden message which can be decoded by the following algorithm:

```
int ox[N+1], oy[N+1];
for(int i = 1; i ≤ N; i++) {
    ox[i] = A[i] / M;
    oy[i] = A[i] % M;
}
```

There are N landmines on the xy-plane, such that the ith landmine $(1 \le i \le N)$ is located at (ox_i, oy_i).

Initially, you are at point $(0, 0)$ and you must go to point (X, Y). At each step, you can either move to the right by 1 unit or move up by 1 unit.
Formally, if you are standing at (x, y), then you can either move to $(x + 1, y)$ or to $(x, y + 1)$.

A path is called *safe* if there aren't any landmines on that path. Count the number of *safe* paths which you can take to reach the destination. As the answer can be quite large, output it modulo $10^9 + 7$.

EXPLANATION:

Suppose there are no obstacles then number of ways to move from point $(0, 0)$ to say point (x, y) will be $\binom{x+y}{x}$. You can read more about it **here**.

Thus we know the total number of ways to reach point (x, y) which is $\binom{x+y}{x}$. Now we will subtract all the invalid ways, i.e ways that go through a landmine.

To calculate number of invalid ways we go through each cell having a landmine and calculate number of ways to reach at that cell and multiply it with number of ways to reach (x, y) from that cell which would give all the number of ways to reach (x, y) passing through that mined cell. We would add all the invalid ways for each such cells to get the total number of invalid paths. We would simply subtract this from total number of paths to get our answer.

Now in order to calculate number of ways to reach a mined cell, we can use $dp[i][j]$, which tells us the number of ways to reach point (i, j) without stepping on any landmine.

$$dp[i][j] = (dp[i-1][j] + dp[i][j-1]) \ \% \ mod$$

Now if there is a landmine on point (i, j), then

$$invalids \mathrel{+}= (dp[i][j] \times \binom{x-i+y-j}{x-i}) \ \% \ mod$$
$$dp[i][j] = 0$$

Thus finally our answer would be:

$$\binom{x+y}{x} - invalids$$

TIME COMPLEXITY:

$O(Mlog(X+Y)),$

$W = max(A[i]/M) + 1,\ 1 \le i \le N$
$H = max(A[i]\ mod\ M) + 1,\ 1 \le i \le N$
So the following inequality is always true :
$W \le (1000000 + M)/M$ and $H \le M$
which gives
$W \times H \le 1000000 + M$

PREREQUISITES:

Dynamic programming, observation

PROBLEM:

There are N shops; the i-th shop sells items with value V_i.
Each shop has an infinite supply of items, and each item costs exactly one dollar.

You can buy items worth at most K dollars. However, the following constraints must be followed:

- Suppose you but num_i items from the i-th shop.
- Then:
  - $\text{num}_1 = 0$ or $\text{num}_i = 1$
  - For $i > 1$, $\text{num}_i \leq \text{num}_{i-1} + 1$

Find the maximum total value of items you can buy.

EXPLANATION:

With bounds on the number of items you can take and an objective of maximizing values, this problem seems somewhat similar to the knapsack problem.
And indeed, this task can indeed be solved with dynamic programming, with each subtask requiring a slightly higher level of optimization.

Subtask 1

Let's start with the easiest one.

A very natural dp state should encapsulate:

- Which shop we're at, say i.
- The total number of items we've bought so far, say x.
- num_i, the number of items we're buying from this shop.

So, let $dp(i, x, y)$ be the maximum value we can obtain if we bought exactly x items from the first i shops, and of them, bought exactly y items from the i-th shop.

Buying y items from the i-th shop gives us a value of $y \cdot V_i$.
Further, we must've bought exactly $x - y$ items from the previous shops; and in also *at least* $y - 1$ items from shop $i - 1$.

This gives us the transitions:

$$dp(i, x, y) = y \cdot V_i + \max(dp(i - 1, x - y, z))$$

where the maximum is across all $y - 1 \leq z \leq K$.

There are $\text{O}(NK^2)$ dp states here (i is upto N, x and y are upto K)2 and the transition from each one can be done in $\text{O}(K)$ time for an overall $\text{O}(NK^3)$ solution.

Subtask 2

Now N and K are a bit higher, so the previous solution is too slow.

Let's look at the dp transitions.
We have

$$dp(i,x,y) = y \cdot V_i + \max(dp(i-1, x-y, z))$$

across $y - 1 \le z$.
The slow part is computing the maximum across all z.

Note that when i, x, y are fixed, what we're looking for is a *suffix maximum* of the $dp(i-1, x-y, \cdot)$ table.
So, let's just calculate these suffix maximums as well!

Define $\text{suff}(i,x,y) = \max(dp(i,x,y), dp(i,x,y+1), dp(i,x,y+2), \ldots, dp(i,x,K))$.
Then, $\text{suff}(i,x,y) = \max(dp(i,x,y), \text{suff}(i,x,y+1))$; so these suffix maximums can be computed as and when we compute the dp values.

With this in hand, we have $dp(i,x,y) = y \cdot V_i + \text{suff}(i-1, x-y, y-1)$.

So, there are now $\mathrm{O}(NK^2)$ states, and an $\mathrm{O}(1)$ transition from each.
This results in a $\mathrm{O}(NK^2)$ solution, which is good enough for this subtask.
Subtask 3

N and K are now large enough that we can't even have NK^2 states anymore; it'll take too much memory and generally be too slow.

Let's try to cut it down a bit.
Suppose we buy y items from a shop.
Since we can increase the maximum by only a step each time, this means we must've bought $1, 2, 3, \ldots, y-1$ items from other shops before this.

In particular, if we buy y items from a shop, we've bought a total of *at least* $\frac{y \cdot (y+1)}{2}$ items in total.
We're only allowed to buy K items, so we don't really care about very large y.
In particular, we don't care about any y such that $\frac{y \cdot (y+1)}{2} > K$; which limits us to about $y \le \sqrt{2K}$.

In other words, we only care about $dp(i,x,y)$ for about $\mathrm{O}(NK\sqrt{K})$ states; each of which has a $\mathrm{O}(1)$ transition using the suffix trick from the previous subtask.

Implementing this optimization is enough to get AC.

Note that depending on your implementation, a solution with correct time complexity for subtasks 2 and 3 might be inexplicably much slower than other similar solutions.
This is most likely to be a memory-related issue.

You can optimize the memory requirement down from $\mathrm{O}(NK^2)$ or $\mathrm{O}(NK\sqrt{K})$ down to $\mathrm{O}(K^2)$ or $\mathrm{O}(K\sqrt{K})$ (or even $\mathrm{O}(K)$ using a slightly different approach) by noting that $dp(i, \ldots)$ depends only on $dp(i-1, \ldots)$ — so, it's enough to keep only the previous row of the dp table, instead of all of them.

This is (maybe) not *required* to get AC, though you should likely see a major speedup on using it.
In slower languages like Python, it might be necessary to get AC.

TIME COMPLEXITY

$\mathrm{O}(NK\sqrt{K})$ per test case.

PREREQUISITES:

Depth first search, dynamic programming, sieve of Eratosthenes/some fast prime factorization method.

PROBLEM:

You have an integer N, that you want to make 1. You also have M pairs of the form (a, b), where a is prime. In one move, you can:

- Pick a prime p that divides N
- Divide N by p
- Multiply N by b for all pairs of the form (p, b).

Find out whether N can be made 1, and if it can, the minimum number of moves to do so.

EXPLANATION:

First, note that we can think of this problem entirely in terms of primes. A pair of the form (a, b) essentially removes a from the prime factorization, and adds in all the prime factors of b.

Next, note that there are not actually any choices to be made in terms of number of moves. So, if there is a way to reduce N to 1, then no matter how it is done the cost will be the same.

Thus, we have to find two things:

- Is it possible to reduce N to 1?
- If it is, find the cost of doing so.

Let's look at a specific prime, say p. What is the cost of removing p from the prime factorization?
Well, based on the pairs we have, removing one copy of p will add in some other primes to the factorization. Let these primes be $p_1, p_2, \ldots, p_k$. Then, each of these primes must be removed, and so on.

So, if $cost_p$ denotes the minimum number of moves to remove p from the factorization, we have

$$cost_p = 1 + \sum_{i=1}^{k} cost_{p_i}$$

This can be implemented to run quickly (in $\mathrm{O}(V + E)$ where V is the number of vertices and is $\leq 10^6$, and E is the number of edges in the graph) with the help of dynamic programming, since each $cost_p$ only needs to be computed once. The final answer is the sum of $cost_p$ for all primes in the prime factorization of N. For example, for $N = 12$ the answer would be $cost_2 + cost_2 + cost_3$.

The only time we run into issues is if we run into a cycle - that is, if removing p from N ends up with adding p back to N. In this case, bringing N down to 1 is of course impossible, making the answer -1.

Note that you have to be careful about one thing: it is possible that the graph has a cycle somewhere in it, but this cycle is not reachable from any of the primes in the factorization of N. In this case, the answer won't be -1. A trivial example of this is when $N = 1$ and the graph can be whatever, with the answer being 0.

Note that the above solution requires computing the prime factorization of a number quickly. This can be done with the help of the sieve of Eratosthenes as follows:

- Let spf_i denote the smallest prime factor of i
- spf_i can be computed with a sieve
- To factorize x, do the following:

- If $x = 1$, stop
- Otherwise, let $p = spf_x$: p is one factor of x.
- Divide x by p and continue.

N has at most $\log N$ prime factors, so the above process takes $\mathrm{O}(\log N)$ time to prime factorize N. Note that this also gives us a bound on the number of edges on in the graph: each of the M pairs adds at most $\log(10^6) = 20$ edges to the graph, so there are $20 \cdot M$ edges at worst.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case, where $N = 10^6$.

PREREQUISITES:

Basic dynamic programming

PROBLEM:

You're given two arrays A and B of length N each.
Use them to create an array C of length $2N$, where A_i and B_i must be placed at positions C_{2i-1} and C_{2i} (in some order).

Alice first chooses a single i, and fixes the order of A_i and B_i in C.
Bob then fixes the order for the other $N-1$ indices.
Alice wants to minimize the maximum subarray sum of the resulting array; while Bob wants to maximize it.
Find the final value of the maximum subarray sum.

EXPLANATION:

Alice has $2N$ possible starting moves: she chooses one index x out of N, then chooses which of C_{2x-1} and C_{2x} should be A_x and which should be B_x.
Suppose Alice's move is fixed. Let's see what Bob will do.

Bob wants to maximize $F(C)$, the maximum subarray sum of C.
Recall that the classical maximum subarray sum problem can be solved in **linear time using dynamic programming.** .
Our problem can be solved by modifying that algorithm slightly, as follows:

- Let dp_i be the maximum sum of a subarray **ending at index** i of array C, if Bob makes his choices optimally.
- Given that there are two choices for where A_i and B_i go, we have the following considerations:
  - At index $2i-1$, Bob can choose either A_i or B_i; along with a subarray ending at index $2i-2$.
    So, $dp_{2i-1} = \max(0, \max(A_i, B_i) + dp_{2i-2})$
  - At index $2i$, Bob must choose the order of A_i and B_i as well.
    If both A_i and B_i are included in the maximum subarray sum, then there'll be a subarray ending at index $2i-2$ as well.
    If only one of A_i and B_i is included, it's best to take the larger one.
    So, $dp_{2i} = \max\{0, A_i, B_i, A_i + B_i + dp_{2i-2}\}$
- Of course, for indices $2x-1$ and $2x$ Alice has already fixed their values; so for these two indices alone the transition will be slightly different: Bob will not have a choice of A_i and B_i and must use the fixed values.
- $F(C)$ will be the maximum value in the dp array.

Together, this gives us a solution in $\mathrm{O}(N^2)$: for each of the $2N$ possible starting moves of Alice, we can compute Bob's best $F(C)$ in $\mathrm{O}(N)$ time.
Taking the min across all these choices gives the final answer.

However, this is too slow for the given constraints.
To optimize it, notice that we can in fact reuse a lot of the information we need without having to recompute it each time.

Let's first assume that Alice hasn't fixed anything, and run the $\mathrm{O}(N)$ algorithm for Bob as described above to compute all dp_i values.
Let's also do the same thing, but in *reverse* (that is, compute rdp_i to be the largest subarray sum starting from index i).

Now, suppose Alice fixes index x, and the values of C_{2x-1} and C_{2x}. Then, considering all possible subarrays:

- First, there are subarrays that end before index $2x-1$.
  For these, the best value is $\max(dp_i)$ across $1 \le i < 2x-1$.
  As a prefix max of the dp array, this can be precomputed and hence found in $\mathrm{O}(1)$.
- Then, there are subarrays that start after index $2x$.
  Once again, this is $\max(rdp_i)$ across $2x < i \le 2N$, and as a suffix max can also be precomputed.
- Finally, we have subarrays that include indices $2x-1$ and/or $2x$. Again, there are three types here:
  - Subarrays including only $2x-1$: the best Bob can do is $dp_{2x-2} + C_{2x-1}$, i.e, take the best sum ending at $2x-2$ and add C_{2x-1} to it.
  - Subarrays including only $2x$: similarly, the value here is $C_{2x} + rdp_{2x+1}$.
  - Subarrays including both: the largest value is $C_{2x-1} + C_{2x} + dp_{2x-2} + rdp_{2x+1}$.
- Bob's best value is the maximum among everything computed here.

Notice that after precomputation, each value we wanted could be obtained in $\mathrm{O}(1)$ time.
So, we can try all $2N$ possible starting positions for Alice, find their $F(C)$ values in $\mathrm{O}(1)$ time, and take the minimum across them all.
This solves the problem in $\mathrm{O}(N)$ time.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Dynamic programming

PROBLEM:

There is a string S, on which Alice and Bob play a game. Alice starts first and then they alternate moves.

- On her move, Alice picks a prefix of S, appends it to her string, and then removes this prefix from S
- On his move, Bob picks a suffix of S, appends its reverse to his string, and then removes this suffix from S

If they play optimally, what is the maximum possible value of the longest common subsequence of Alice's and Bob's strings?

EXPLANATION:

The main observation to be made here is that, no matter how the moves are performed, Alice's final string will be some prefix of S and Bob's string will be the (reversed) remaining suffix.

That is, Alice's string will be $S[1 \dots i]$ and Bob's string will be $rev(S[i+1 \dots N])$ for some $1 \le i \le N$. It's not hard to see that every such pair of strings is achievable.

For convenience, let R denote the reverse of S.
Our task is then to compute the maximum value of $LCS(S[1 \dots i], R[1 \dots j])$ across all (i,j) such that $i+j = N$.

This is a rather standard problem that is solved using dynamic programming; in fact, we can simply use the exact same states as the classical longest common subsequence problem.

Let `dp[i][j]` denote the longest common subsequence of the strings $S[1 \dots i]$ and $R[1 \dots j]$.
Then, there are three possible cases:

- S_i is not a part of the LCS; in which case the answer is `dp[i-1][j]`
- R_j is not a part of the LCS; in which case the answer is `dp[i][j-1]`
- S_i and R_j are both part of the LCS; in which case the answer is `1 + dp[i-1][j-1]`
  - Note that $S_i = R_j$ must hold for this case to be applicable.
- `dp[i][j]` is the maximum of the above three values.

The base cases are `dp[i][0] = 0` and `dp[0][j] = 0`.

Dynamic programming allows us to compute `dp[i][j]` for every $0 \le i,j \le N$ in $\mathrm{O}(N^2)$, after which the final answer is simply the maximum value of `dp[i][j]` across all (i,j) such that $i+j = N$.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ per testcase.

PREREQUISITES:

Dynamic Programming, MEX

PROBLEM:

A function F is defined on an array A of length N as follows:

$F(A) = \sum_{i=1}^{N} MEX(A_1 + A_2 + \cdots + A_i)$

For example, if $A = [2, 0, 1]$, then $F(A) = MEX([2]) + MEX([2, 0]) + MEX([2, 0, 1]) = 0 + 1 + 3 = 4$.

You are given two integers N and K. For each $i = N, N+1, \ldots, K$ in order, find the number of **permutations** P of integers $[0, 1, 2, \ldots, N-1]$ such that $F(P) = i$. Since the answer can be very large, find it modulo $(10^9 + 7)$.

EXPLANATION:

Hint

Try to put the elements $0, 1, 2, \ldots, N-1$ in order and see how the value of $F(A)$ changes for putting a new element in the permutation

Solution

Let's say Pre_j denotes the prefix of length j of the permutation P and Mex_j denotes the mex of Pre_j. Thus $F(A) = Mex_1 + Mex_2 + \cdots + Mex_N$.

Initially consider that the permutation P is empty. Hence $Mex_j = 0$ for each $1 \le j \le N$, as the mex of an empty array is 0. Thus $F(A) = 0$. Now we put the elements from 0 to $N-1$ in order.

Say we put the element 0 on index id_0. Now for all indices $j = id_0, id_0 + 1, \ldots, N$, Pre_j will contain the element 0 making $Mex_j = 1$, for the rest indices Mex_j remains 0. So putting the element 0 on index id_0 increase the value of $F(A)$ by $N - id_0 + 1$.

Now we put the element 1 on index $id_1 (id_1 \neq id_0)$. Here two cases occur:

- $id_1 < id_0$: For all indices $j = id_0, id_0 + 1, \ldots, N$, Pre_j will contain the element $0, 1$ making $Mex_j = 2$. So putting the element 1 on index id_1 increases the value of $F(A)$ by $N - id_0 + 1$, because for each $id_0 \le j \le N$ the value of Mex_j becomes 2 from 1, Mex_j remains unchanged for the remaining indices.
- $id_1 > id_0$: For all indices $j = id_1, id_1 + 1, \ldots, N$, Pre_j will contain the element $0, 1$ making $Mex_j = 2$. So putting the element 1 on index id_1 increases the value of $F(A)$ by $N - id_1 + 1$ because for each $id_1 \le j \le N$ the value of Mex_j becomes 2 from 1, Mex_j remains unchanged for the remaining indices.

Let' say $Mx_1 = \max(id_0, id_1)$. Then putting 1 on index id_1 increases the value of $F(A)$ by $N - MX_1 + 1$.

If we put the element 2 on index id_2 and $Mx_2 = \max(id_0, id_1, id_2)$, it will increase the value of $F(A)$ by $N - MX_2 + 1$.

Let, $Dp[i][Mx_i][S]$ = number of ways to put the elements $0, 1, \ldots, i$ in a permutation of length N such that maximum position of these elements is Mx_i and the value of $F(A)$ we get is S. So $Dp[0][i][N - i + 1] = 1$ for each $1 \le i \le N$.

Say, we have put the elements $0, 1, 2, \ldots, i-1$ on positions $id_0, id_1, \ldots, id_{i-1}$ respectively, the current value of $F(A)$ is S. Now we put the element i on index id_i and $Mx_i = \max(id_0, id_1, \ldots, id_i)$. It will increase the value

of $F(A)$ by $N - mx[i] + 1$. Here two cases may occur:

- $id_i = Mx_i$ (i.e . we put the element i on index Mx_i): The maximum position among $0, 1, \ldots, i-1$ (i.e. Mx_{i-1}) can lie in the range $[1, Mx_i - 1]$. Thus we will do: $Dp[i][Mx_i][S + N - Mx_i + 1] += \sum_{j=1}^{Mx_i - 1} Dp[i-1][j][S]$.
- $id_i < Mx_i$ (i.e one of $0, 1, \ldots, i-1$ on position Mx_i): We have put $0, 1, \ldots, i-1$ among the first Mx_i positions from left. Hence we can put the element i on one of the $Mx_i - i$ positions. Thus we will do: $Dp[i][Mx_i][S + N - Mx_i + 1] += Dp[i-1][Mx_i][S] * (Mx_i - i)$.

So the number of permutations P of length N with $F(A) = i$ will be $Dp[N-1][N][i]$.
.

TIME COMPLEXITY:

$O(N^2 \cdot K)$ for each test case.

PREREQUISITES:

Depth-first Search, Dynamic Programing

PROBLEM:

You are a given a tree of N vertices. The i-th vertex has two things assigned to it — a value A_i and a range $[L_i, R_i]$. It is guaranteed that $L_i \le A_i \le R_i$.

You perform the following operation **exactly once**:

- Pick a (possibly empty) subset S of vertices of the tree. S must satisfy the additional condition that no two vertices in S are directly connected by an edge, i.e, it is an **independent set**.
- Then, for each $x \in S$, you can set A_x to any value in the range $[L_x, R_x]$ (endpoints included).

Your task is to minimize the value of $\sum |A_u - A_v|$ over all **unordered** pairs (u, v) such that u and v are connected by an edge.

EXPLANATION:

Observation : For a particular node u if we are going to change its value, we will either make it equal to the median of all A_i values of its neighbors or either it would be equal to L_u or R_u depending upon which is the nearest value to the median.

Median minimizes the sum of absolute deviations

Given a set of values S
We're basically after:
arg min(x) $\sum_{i=1}^{N} |si - x|$

One should notice that $d|x|/dx$=$sign(x)$
Hence, deriving the sum above yields $\sum_{i=1}^{N} sign|si - x|$
This equals to zero only when the number of positive items equals the number of negative which happens when $x = median[s_1, s_2, \cdots, s_N]$.

If median is less than L then we will set A_u to L: Suppose we set A_u to some value $L^{'} > L$. Reducing $L^{'}$ by 1 does not increase the absolute sum of deviations(number of values $\ge L^{'}$ is less than or equal to number of values $<$ than $L^{'}$) and thus we can keep on decreasing the value of $L^{'}$ until we reach L.
Similarly, if median is greater than R we will set it to R.

Now this problem can be solved using dynamic programing.
Let us root the tree at node 1. For each node of the tree calculate the optimal value after a change by sorting all the values of the neighbors and using the above observation and let this value for node i be stored in $optimal_i$.
Let $dp[u][1]$ represent the minimum value that can be obtained in the subtree of node u by selecting some or more nodes in set S while also changing the value of node u and dp[u][0] represents the minimum value that can be obtained in its subtree by selecting some or more nodes in set S while not changing the value of node u. If we change the value of node u we cannot change the value of any of its neighbors. Initialize all the dp states with 0.

start a depth first search from node 1, Now
$dp[u][1] = \sum dp[x][0] + abs(A[x] - optimal[u])$ for every child x of node u
$dp[u][0] = \sum min(dp[x][0] + abs(A[x] - A[u]), dp[x][1] + abs(optimal[x] - A[u]))$ for every child x of node u

The answer to the problem is $min(dp[1][0], dp[1][1])$

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PREREQUISITES:

Subset sum dp

PROBLEM:

You're given an array A containing N integers.
In one move, you can pick $1 \le i < j \le N$ and subtract $\min(A_i, A_j)$ from both A_i and A_j.

Find out whether it's possible to make all the array elements 0.
If it is, also find a way to achieve this using at most N operations.

EXPLANATION:

First, let's see how to decide whether the answer is `Yes` or `No`.

Clearly, performing an operation when $\min(A_i, A_j) = 0$ is pointless, so let's not perform such operations.

Suppose a valid sequence of moves exists. For each i, let's match it with the j that sets it to zero.
In other words, let's create a graph on N vertices, where for each move (i, j) that we make, we add the edge $i \leftrightarrow j$.

A bit of observation should tell you that this graph cannot contain cycles, i.e, it's a forest.

Proof

Let's build the graph edge-by-edge.
Suppose we're adding the edge (i, j). This sets at least one of A_i and A_j to zero.

Claim: At any stage, every connected component of our graph contains at most one node with non-zero value.
Proof: The proof is by induction on the number of edges added.
The claim is clearly true for when there are no edges, since each component is a singleton.
Now, look at what happens when we add an edge (i, j).

- If A_i and A_j are in the same component, one of them is zero so we don't make this move anyway.
- If they're in different components, these two components join together. However, at least one of A_i and A_j becomes zero, so the new combined component also has at most one non-zero vertex.

This shows that every edge joins two different components together, and hence cannot create a cycle.

This graph being a forest specifically means that it's **bipartite**.
Let (X, Y) be two sets forming a bipartition of this graph.

Also, let $S_X = \sum_{x \in X} A_x$ and $S_Y = \sum_{y \in Y} A_y$ be the sums of the respective sets.

Notice that any edge connects some element of X to some element of Y; in other words, every operation subtracts the same value from both S_X and S_Y.
So, $S_X = S_Y$ must hold.

In particular, this means A can be partitioned into two subsets of equal sum.
Of course, this condition is necessary for the answer to be `Yes`. Is it also sufficient?

Yes, it is!

Suppose X and Y partition A, such that $S_X = S_Y$.

it's quite easy to construct a sequence of operations: pick some non-zero element from X, pick some non-zero element from Y, then perform the operation on them.

Since their sums are equal, both sets will become empty at the same time.

So, all we really need to do is to check whether A can be partitioned into two subsets with equal sum.
This is, in fact, an instance of the subset-sum problem. We're really just looking for a subset of A whose sum equals $sum(A)/2$.
Notice that if $sum(A)$ is odd then we can never find such a subset.

The subset-sum problem can be solved in $\mathrm{O}(N \cdot sum(A))$ using dynamic programming.

How?

Let $dp(i, x)$ be True if we can get a subset with sum x from the first i elements, and False otherwise.
Then, $dp(i, x)$ is True if and only if $dp(i-1, x)$ is True, or if $dp(i-1, x-A_i)$ is True; otherwise it's False.

This gives us $\mathrm{O}(N \cdot sum(A))$ states and $\mathrm{O}(1)$ transitions each.

Once a bipartition has been found, reconstructing the operations is trivial; as mentioned above, simply keep picking one non-zero element each from the two sets.
This takes at most $N-1$ steps, since every operation makes one element 0.

This instance of the subset sum problem can be solved even faster: using bitsets for a factor 64 speedup, or even in $\mathrm{O}(N \cdot \max(A))$ (see **this blog** if you're interested).
However, neither are needed to get AC in this task.

TIME COMPLEXITY:

$\mathrm{O}(N \cdot sum(A))$ per testcase.

PREREQUISITES:

Dynamic programming

PROBLEM:

You are given integers N and M.
An $N \times M$ tables containing 0 and 1 is said to be *special* if, for each (i,j), $A_{i,j} = 1$ if and only if the sum of row i equals the sum of column j.

Find the minimum sum of values of a special table with N rows and M columns.

EXPLANATION:

First off, what does a special table even look like?

Suppose the row sums are $r_1, r_2, \ldots, r_N$ and the column sums are $c_1, c_2, \ldots, c_M$.
Say $r_1 = k$. What information does that give you?

Answer

There are exactly k ones in the first row.
This means that there are exactly k columns whose sum is k.

First, let's assume $k \neq 0$.
Applying the same argument to one of the columns with sum k tells us that there are also exactly k rows whose sum is k.

So we have k rows and k columns, such that:

- The intersection of any one of these rows and columns contains a 1
- All other cells in these rows/columns are 0

This gives us k^2 ones in the grid.

Notice that without loss of generality, we can assume $r_1 = r_2 = \ldots = r_k = c_1 = c_2 = \ldots = c_k = k$, then delete these rows and columns and apply the same argument to the rest of the grid.

Now, what happens if $k = 0$?
Well, we still have exactly k columns with sum k, i.e, there are no columns with sum 0.
This means we can simply delete the first row and consider the remaining part of the grid instead.

The above discussion in fact tells us the following:

- First, the analysis of $k = 0$ tells us that the grid can contain *either* empty rows or empty columns, but not both. So, we can safely delete empty rows/columns and solve for the reduced N or M appropriately.
- Second, let the *distinct* non-zero row (or column) sums be $x_1 < x_2 < \ldots < x_d$.
  Then, the grid contains exactly $x_1^2 + x_2^2 + \ldots + x_d^2$ ones.
- Finally, notice that the number of non-zero rows will equal the number of non-zero columns. Combining this with the first point, it's enough to deal with only square grids, i.e, solving for an $N \times M$ grid is the same as solving for a $\min(N, M) \times \min(N, M)$ grid.

Now let's use the above observations to formulate a solution.
From now on, I'll assume $N = M$ since we established that that's the only case that matters.

As in the second point above, let $x_1, x_2, \ldots, x_d$ be the distinct row-sums of the grid.
Then, note that we must have $x_1 + x_2 + \ldots + x_d = N$.

So, our objective is to minimize $x_1^2 + x_2^2 + \ldots + x_d^2$ subject to the above constraint. Note that the x_i are all distinct here.

Since $N \leq 5000$, solving this in $\mathrm{O}(N^2)$ is still ok: and that is exactly what we will do using dynamic programming.

Let $dp(i,j)$ denote the minimum possible value of the sum of squares, such that:

- The sum of the elements chosen is i; and
- We have picked elements from $\{1, 2, \ldots, j\}$.

Under this formulation, the answer is simply $dp(N, N)$.
Computing $dp(i,j)$ is not hard: we either choose j or we don't, so

$$dp(i,j) = \min(dp(i,j-1), dp(i-j,j-1) + j^2)$$

This allows us to solve a single testcase in $\mathrm{O}(N^2)$.

However, note that there are a large number of testcases, and so running this $\mathrm{O}(N^2)$ algorithm separately for each is not quite enough.

However, the dp states themselves don't actually change across testcases: the values computed are the exact same.
So, simply precompute the dp table for all possible values before even processing any testcase: then, a testcase can be answered in $\mathrm{O}(1)$.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ precomputation followed by $\mathrm{O}(1)$ per test case

PREREQUISITES:

Dynamic Programming

PROBLEM:

You are given an array $A = [A_1, A_2, \ldots, A_N]$ containing **distinct positive** integers.

Let B be a **permutation** of A. Define the *value* of B to be

$\sum_{i=1}^{N}(B_i \bmod B_{i+1})$

where B_{N+1} is treated to be B_1.

Find the **maximum** *value* across all permutations of A.

EXPLANATION:

For any permutation B, we can see that $B_i \mod B_{i+1} \le B_i$, and $B_i \mod B_{i+1} < B_i$ if and only if $B_i > B_{i+1}$. We say that the pair (B_i, B_{i+1}) is *bad*, and it reduces the value of the permutation by $B_i - B_i \mod B_{i+1}$; at the end, the value of the permutation B is just sum of all elements in B minus the reduced values of all bad pairs in B. Therefore, our goal now switches to finding a permutation such that the sum of values of bad pairs is as small as possible.

Sort A so that $A_1 < A_2 < \cdots < A_N$. We see that any permutation must have the bad pairs satisfying these conditions:

- A_N is in exactly one bad pair as the front element. This is because since the array is cyclic, A_N always precedes some element.
- A_1 is in exactly one bad pair as the back element.
- There must be a sequence of bad pairs $[A_1, A_{R_1}], [A_{L_2}, A_{R_2}], \ldots, [A_{L_k}, A_1]$ where $R_i \le L_{i+1}$ for all $1 \le i < k$. This is because $A_{L_{i+1}}$ always goes after some element, so we either need $A_{L_{i+1}}$ to be the back element of some bad pair (which means $A_{R_i} = A_{L_{i+1}}$, or there is another bad pair covering $A_{L_{i+1}}$ (i.e. $A_{R_i} < A_{L_{i+1}}$).

We can also prove that from any set of bad pairs satisfying such conditions, we can create a permutation with only those bad pairs. Therefore the problem now becomes finding such a set of bad pairs that satisfies the said conditions. We can do a dynamic programming solution as follow:

Let dp_i be the smallest sum of values of a set of bad pairs $[A_1, A_{R_1}], [A_{L_2}, A_{R_2}], \ldots, [A_{L_k}, A_i]$ that satisfies the condition, then

$dp_i = \min_{j=i+1}^{n}(\min_{k=i+1}^{j} dp_k) + (A_j - A_j \mod A_i)$

where $dp_N = 0$. We can implement this formula easily in $O(N^2)$.

TIME COMPLEXITY:

Time complexity is $O(N^2)$ per test case.

PREREQUISITES:

Dynamic Programming, Binary Search, Range Minimum Queries, Coordinate Compression

PROBLEM:

Given array A and integer K. Determine the largest value s such that, it is possible to partition A into K contiguous blocks, where the sum of elements of each block is $\geq s$.

EXPLANATION:

Let s be the largest integer such that it is possible to partition A into K contiguous blocks, where the sum of elements of each block is $\geq s$. This value is the answer we are looking for.

Step 1: Use binary searching to find the value of s.

Why? How?

Let $solve(d)$ represent whether it is possible to partition A into K blocks, where the sum of elements of each block is $\geq d$. Let $solve(d)$ equal 1 if it's possible to partition, and 0 otherwise.

Observation 1: If $solve(d) = 1$, then $solve(d') = 1$ for all $d \geq d'$.
(The proof is trivial and left to the reader as an exercise)

Observation 2: If $solve(d) = 0$, then $solve(d') = 0$ for all $d' \geq d$.

Proof

We prove by contradiction.
Say there exists $d' \geq d$ such that $solve(d) = 0$ and $solve(d') = 1$.

By the previous observation, $solve(d') = 1$ implies $solve(x) = 1$ for all $d' \geq x$. But $solve(d) = 0$ - a contradiction and we are done.

The two observations imply the monotonic structure of the function $solve$, sufficiently allowing us to binary search to find the answer.

Let d be the currently considered value in the binary search. Now, we want to check if it's possible to partition A into K blocks, each whose sum $\geq d$.

Definition: *Call an array of elements good, if the sum of its elements $\geq d$.*

Let dp_i be the set of all positive integers k, such that it is possible to partition the first i elements of A into k good subarrays. I claim that dp_i is a (possibly empty) set of consecutive integers, for all i.

Proof

Since the proof has a lot in common with the logic of the solution, reading the rest of the editorial first will make the proof much more clearer.

Let $pref(x) = A_1 + A_2 + \cdots + A_x$.

Computing dp_i is straightforward, and can be described mathematically as:

$$dp_i = \{x + 1 \mid x \in dp_j \ \forall\, j < i \text{ and } pref(j) \leq pref(i) - d\}$$

Let's define $G_i[v]$ as the set of integers x, such that it is possible to partition the first j elements of A into x good subarrays, where $j < i$ and $pref(j) = v$.

$$G_{i+1}[v] = \begin{cases} G_i[v] \cup dp_i, & \text{if } pref(i) = v \\ G_i[v], & \text{otherwise} \end{cases}$$

This recurrence is derived from a simple (yet important!) observation, the proof of which is left to the reader as an exercise.

Finally (the last definition, bear with me), let's define $H_i[v] = G_i[v] \cup H_i[v-1]$. That is, $H_i[v]$ is the union of all sets $G_i[v']$ with $v' \leq v$. It is easy to deduce that $H_i[v']$ is always subset of $H_i[v]$.

Informally, $H_i[v]$ can be described as the set of integers x, such that it is possible to partition the first j elements of A into x good subarrays, where $j < i$ and $pref(j) \leq v$.

Then, we can rewrite the set relation of dp_i (mentioned at the start) as:

$$dp_i = \{x + 1 \mid x \in H_i[pref(i) - d]\}$$

Thus, if we prove that $H_i[v]$ is a (possibly empty) set of consecutive integers for all v, then dp_i is also a (possibly empty) set of consecutive integers, completing the proof.

We prove by induction.
Assume for a particular n, $H_n[v]$ is a set of consecutive integers, for all v. Let us prove by inducting on n.

$$\begin{aligned} H_{n+1}[v] &= G_{n+1}[v] \cup G_{n+1}[v-1] \cup \dots \\ &= \begin{cases} (G_n[v] \cup G_n[v-1] \cup \dots) \cup dp_n, & \text{if } pref(n) \leq v \\ (G_n[v] \cup G_n[v-1] \cup \dots), & \text{otherwise} \end{cases} \\ &= \begin{cases} H_n[v] \cup dp_n, & \text{if } pref(n) \leq v \\ H_n[v], & \text{otherwise} \end{cases} \end{aligned}$$

$H_{n+1}[v] = H_n[v]$ implies our proposition holds for $H_{n+1}[v]$ (since by our assumption, it holds for $H_n[v]$). So, lets only focus on the former case, which can be expanded as:

$$H_{n+1}[v] = H_n[v] \cup \{x + 1 \mid x \in H_n[pref(n) - d]\}$$

One of $H_{n+1}[v], H_n[pref(n) - d]$ is a subset of the other. A bit of case work shows that, the union of two overlapping sets of consecutive integers, after shifting all values of one set by 1, is still a set of consecutive integers. Therefore, the proposition holds true for $H_{n+1}[v]$, establishing the inductive step.

The trivial case of $n = 0$ has $H_0[v] = \emptyset$ for all v, which matches our proposition.

Thus proved.

Why is this claim important? The claim implies that the set of values of dp_i is made up of all integers in some range $[L_i, R_i]$. Then, A can partitioned into K good subarrays only if $L_N \leq K \leq R_N$.

All that remains now is to calculate L_i and R_i efficiently.

Suboptimal solution

A straightforward DP recurrence relation gives:

$$L_i = 1 + \min L_j$$
$$R_i = 1 + \max R_j$$

where $j < i$ and $\sum_{x=j+1}^{i} A_x \geq d$.

This can be calculated in $O(N^2)$, enough for subtask 2, but not viable for the original constraints.
Optimised solution

Let $pref(x) = A_1 + A_2 + \cdots + A_x$.

The suboptimal solution can be rewritten as:

$$L_i = 1 + \min L_j$$
$$R_i = 1 + \max R_j$$

where $j < i$ and $pref(i) - pref(j) \geq d \implies pref(j) \leq pref(i) - d$.

Let $mini_x$ represent the set of all L_j such that, $j < i$ and $pref(j) = x$. Similarly, let $maxi_x$ represent the set of all R_j such that, $j < i$ and $pref(j) = x$.
Then, from the above stated recurrence constraint, it is easy to see

$$L_i = 1 + \min_{x \leq pref(i) - d} (\min mini_x)$$
$$R_i = 1 + \max_{x \leq pref(i) - d} (\max maxi_x)$$

This can be done easily using range minimum queries on $mini$ and $maxi$, typically using a Segment Tree/Fenwick Tree with coordinate compression (since the range of x in $mini_x$/$maxi_x$ is $[-1e14, +1e14]$).

The time complexity of this approach is $O(N \log N)$, which is efficient enough to AC for the given constraints!

To summarise, binary search for the answer - let the midpoint value we are currently trying be d. For this particular d, calculate $[L_i, R_i]$ for all i, in subquadratic time. Then, A can be partitioned into K subarrays, each with sum of elements $\geq d$, only if $L_N \leq K \leq R_N$. Return the result and continue the binary search until it converges.

TIME COMPLEXITY:

Binary searching for the answer takes $O(\log S)$, where S is the size of the answer space ($S \leq 2 \times 10^{14}$).
Calculating $[L_i, R_i]$ for each d (using the memory efficient method) takes $O(N \log N)$.
The total time complexity is therefore

$$O(N \log N \log S)$$

per test case.

PROBLEM:

You will be given an array **A** of **K** types of radioactive decays where each decay is a pair of values a_i and b_i. In each radioactive decay, mass number of a atom decreases by a_i and atomic number decreases by b_i. Then you will be given **Q** queries, and for each query you will get mass and atomic numbers of the initial atom M_1, Z_1 and desired(special) mass and atomic number of special atom M_2, Z_2. You have to print the number of ways to obtain special element from initial element via the **K** given radioactive decays. Since the answer can be very large, print it by taking modulo $10^9 + 7$.

NOTE- Order of the $Decay$ doesn't matter.
$decay_1 \rightarrow decay_2 \rightarrow decay_3$ then all possible permutations of this chain reaction will be considered same, i.e., $decay_2 \rightarrow decay_1 \rightarrow decay_3$, $decay_1 \rightarrow decay_3 \rightarrow decay_2$, $decay_2 \rightarrow decay_3 \rightarrow decay_1$, $decay_3 \rightarrow decay_1 \rightarrow decay_2$ and $decay_3 \rightarrow decay_2 \rightarrow decay_1$ all are same.

Prerequisites:

- Dynamic Programming.
- Modular arithmetic
- Pinch of common sense 🙂

Hint:

Hint 1

Try thinking of a recursive solution by taking difference of M_1 , M_2 and Z_1 , Z_2 and reducing it to zero.

Hint 2

dp [$difference\ of\ M_1\ and\ M_2$] [$difference\ of\ Z_1\ and\ Z_2$] [$current\ index\ in\ decay\ array$] = $No.\ of\ ways\ of\ making\ that\ difference\ zero.$

QUICK EXPLANATION:

A simple approach that comes across our mind is to use all types of $decays$ available to convert M_1 to M_2 and Z_1 to Z_2. This can be done easily by taking there differences and making it zero.

EXPLANATION:

Since it is a counting number of ways type of a problem, we will return be returning 1 whenever our variable reaches desired base case and will return 0 in other base cases. It will be a *knapsack* type of approach where we have to either use the given $decay[i][0..1]$ and decrement the difference M and Z accordingly and recurse for the same index i in $decay[\ ][\ ]$ array or simply leave the

difference as it is and recurse for $i-1$.
Let $decay[0..k][2]$ be a 2-D array containing k types of decays.
Firstly we have to define a $dp[\ ]$ array and determine its state and transition.

State

$$dp[\ difference\ of\ M_1\ and\ M_2][\ difference\ of\ Z_1\ and\ Z_2][\ Current\ decay\ index\ ] = No.\ of\ ways\ to\ make\ the\ difference\ zero$$

Transition

$$dp[M][Z][i] = dp[(M - decay[i][0])\ ]\ [(Z - decay[i][1])\ ]\ [i]\ +\ dp[M][Z][i-1]$$

Base cases for the recursive calls will be when the difference M and Z is zero we have to return 1.
If the difference M or Z or the decay index i is less than zero ($M < 0$ || $Z < 0$ || $i < 0$) then return 0.
Also don't forget taking mod of number of ways by $1e^9 + 7$.

SOLUTIONS:

Setter's Solution
Tester's Solution

Quote

PREREQUISITES:

Dynamic Programming, Suffix Sum, Fenwick Tree

PROBLEM:

Given a Directed-Acyclic-Graph (DAG) G = (V, E) in which node i has edges to nodes in [i + 1, i + N[i]], find how many paths are there between S[i] and T.

EXPLANATION:

This DAG is really special and the order of 1 … V is exactly same as its topo order in which edges are only existed from previous nodes to their later ones.

Use F[i] to state the number of different paths starting from node i to node T.

```
Initially F[T] = 1, F[others] = 0.
```

The transmission can be described as following:

```
For i = T - 1 downto 1
    F[i] = \sum_{v = i + 1} ^ {i + N[i]}
```

To speed up this transmission procedure, we can use a Fenwick Tree to get the sum. But we can achieve it in a simpler way as following, using suffix sum.

```
suffixSum[] = 0;
suffixSum[T] = 1;
For i = T - 1 downto 1
    F[i] = G[i + N[i]];
    G[i] = G[i + 1] + F[i]
```

To answer each query, just directly output the F[S[i]]. Therefore, the time complexity is O(N + Q) in total.

AUTHOR’S AND TESTER’S SOLUTIONS:

Author’s solution can be found **here** .
Tester’s solution can be found **here** .

PREREQUISITES:

Dynamic Programming

PROBLEM:

You are given an array A consisting of N integers. Calculate the number of ways to divide this array into subsegments, such that the sequence formed by taking bitwise AND in each segment of the partition is a palindrome.

More formally, consider a partition of the array into segments $[L_1, R_1]$, $[L_2, R_2]$, $[L_3, R_3]$, $\ldots$, $[L_k, R_k]$ such that $L_1 = 1, L_2 = R_1 + 1, L_3 = R_2 + 1, \ldots, L_k = R_{k-1} + 1, R_k = N$. Let's define B_i as AND on i-th segment: $B_i = A_{L_i} \wedge \ldots \wedge A_{R_i}$, where $x_1 \wedge x_2 \wedge \ldots \wedge x_t$, denotes bitwise AND of numbers $x_1, \ldots, x_t$.

A partition is *palindromic* if $B_i = B_{k+1-i}$ for every $1 \le i \le k$. Your task is to calculate the number of palindromic partitions. Since this number can be large, calculate it modulo 998244353.

EXPLANATION:

We have a pretty straightforward dynamic programming solution: Let $dp_{l,r}$ be the number of palindromic partitions of the subarray $A_{[l,r]}$, then:

$$dp_{l,r} = 1 + \sum_{i=l}^{r-1} \sum_{j=i+1}^{r} dp_{i+1,j-1} \cdot (f(l,i) == f(j,r))$$

where $f(l,r) = A_l \wedge A_{l+1} \wedge \cdots \wedge A_r$.

However, this solution obviously will not pass, since the complexity is $O(N^4)$. We have a few observations to speed up this algorithm:

- Notice that if we fix l, there are at most $\log(\max(A))$ values of $f(l,i)$. To prove this, we can imagine $f(l,i)$ as the "prefix AND" starting from l, where $f(l,i) = f(l,i-1) \wedge A_i$. Notice that $f(l,i)$ is always a submask of $f(l,i-1)$. Therefore, if we look at the number of bits of $f(l,i)$, it is always non-decreasing, which means there are only $\log(\max(A))$ values of the number of bits, which proves our observation.
- Also notice that $f(l,i)$ is also non-increasing, which can be proven directly from the fact that $f(l,i)$ is a submask of $f(l,i-1)$.

Therefore, we can group $f(l,i)$ into at most $\log(\max(A))$ ranges, where each range contains the indices j where $f(l,j)$ have the same value. Similarly, when we fix r, we can group $f(j,r)$ into at most $\log(\max(A))$ ranges. Finally, using the monotonic property, we can use two-pointers to loop over these ranges and pick out the two ranges where $f(l,i)$ have the same value as $f(j,r)$.

This alone doesn't solve the problem, since we still need to iterate i and j over these ranges; more specifically, we still need to calculate $\sum_{i=a}^{b-1} \sum_{j=c+1}^{d} dp_{i+1}j-1$, where $[a,b)$ and $(c,d]$ are the mentioned ranges. Luckily, we can easily optimize this operation: let $s_{i,j}$ be sum of $dp_{l,r}$ where $i \le l \le r \le j$ (basically, $s_{i,j}$ sums over all dp within the range $[i,j]$). We can easily calculate $s_{i,j}$ after calculating any $dp_{i,j}$ ($s_{i,j} = dp_{i,j} + s_{i,j-1} + s_{i+1,j} - s_{i+1,j-1}$), and the previous calculation is just $s_{a+1,d-1} - s_{a+1,c-1} - s_{b+1,d-1} + s_{b+1,c-1}$.

TIME COMPLEXITY:

Time complexity is $O(N^2 \log(\max(A)))$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

Chef has N numbers written on his board. He wants only one number to remain on the board and so he devises a special tactic to do so.

He randomly picks up 2 numbers written on the board say A and B respectively. He erases both of them from the board and writes a new number on the board which equals -

- $A - B$ if parity of $A =$ parity of B
- $A + B$ if parity of $A \neq$ parity of B

Here parity refers to remainder obtained when a number is divided by 2.

Notice that the total numbers on the board reduced by 1 after 1 such operation. After a finite number of operations there would only be 1 number remaining on the board.

Chef is interested in knowing how many different final numbers he could end up with.

EXPLANATION:

This problem requires a little bit of case work and observations. In the end the resulting number is a signed(positive or negative) sum of all numbers of the array but there are some constraints on the sign of the numbers.

Observation 1

Let x be the count of odd numbers in the array then $floor(x/2)$ odd numbers will have a '-' sign before them. After applying operations some numbers will have positive sign in result and some negative let us show no matter how you apply operations you always end up with always $floor(n/2)$ negatives (here n = count of odds) (say this function as F_n)
For $n = 1$ => negative = 0
For $n = 2$ => negative = 1
Now when computing for higher n we split it considering before the last operation one side had processed x_1 numbers and other x_2 such that
$x_1 + x_2 = n$ then result after this operation will be F_{x_1} + $(x_2 - F_{x_2})$ iff both x_1 and x_2 have same parity else it will be $F_{x_1} + F_{x_2}$
For $n = 3$ => only splitting possible $(2, 1)$ or $(1, 2)$ => $(2, 1) = 1 + 0 = 1$ and $(1, 2) = 0 + 1 = 1$; Hence, 1;
for $n = 4$ => $(2, 2)(3, 1)(1, 3)$ => $(2, 2) = 1 + (2 - 1) = 2$;$(3, 1)$ => $1 + (1 - 0) = 2$; $(1, 3) = 0 + (3 - 1) = 2$;
Now lets generalize by induction $(F_n = n/2)$
For odd n we will always get different parity numbers hence F_{x_1} + F_{x_2} = $x_1/2$ + $x_2/2$ = $(x_1 + x_2)/2 = n/2$
For even n we will always get same parity numbers hence $F_{x_1} + (x_2 - F_{x_2})$
now for odd numbers $floor(n/2)$ = $(n - 1)/2$
same parity - (even, even) - $x_1/2$ + $(x_2 - x_2/2)$ for even numbers $x_2 - x_2/2$ = $x_2/2$ hence $(x_1 + x_2)/2 = n/2$
(odd, odd) - $x_1/2 + (x_2 - x_2/2)$ lets substitute $(x_1 - 1)/2 + (x_2 - (x_2 - 1)/2) = (x_1 + x_2)/2 = n/2$
Hence, we always end up with $floor(n/2)$ odd negatives no matter how we procced.

Observation 2

If the count of odd numbers is greater than or equal to 2 then the sign of any even number can be assigned to be + or -.
Construction: Select an odd number, add all even numbers to it in which you want a + sign. Select another odd

number , add all even numbers to it in which you want a $-$ sign. Both of these resulting numbers are odd. Now subtract the second number from the first number. The resulting number is the only even number and rest all are odd. Keep on doing the operations until there is only one number left. The result number either has the configuration of '+' and '-' as we need or the reverse of it(+ instead of $-$ and vice versa). If we have achieved reverse configuration just swap all even numbers in the beginning to get the desired number.

Observation 3

If there is No odd number in the array we cannot assign $-$ sign or + sign to all the even numbers. Just look at the first operation the sign of the elements in the first operation is swapped and will always be opposite to each other no matter what the operations are. These means there is at least 1 even number with + sign and one with $-$ sign.
Construction: Subtract all even numbers in front of which you want a + sign except one from the even number(in front of which you want a negative sign). Now subtract the even number formed by these operations from the one which was excluded (in front of which we need a + sign). Now all the even elements in front of which we want a + sign have a + sign, Keep on subtracting even numbers from this number till you reach a single number.

Observation 4

If the count of odd numbers in the array is exactly 1 then we cannot assign '-' sign to all even numbers. If there is only one even number this already true. Let there be more than 2 even numbers. The count of odd numbers is always going to be 1 in the array and '-' sign can only be introduced when two even numbers are subtracted from each other in this case. Now these two even numbers cannot have the same sign till the end. To create a number in which all even numbers have positive sign just keep adding all of them to the odd number.
Construction: Add all ,except one, even numbers in front of which you want a positive sign to the odd number and keep subtracting other even numbers in front of which you want a negative sign from the even number excluded earlier. Now add these two numbers to get the desired number.

After these operations this problem reduces to compute all possible sums given these restrictions on the sign of even and odd numbers in various cases. This can be solved using standard dynamic programing approach.
For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(N^3 \cdot A_{MAX})$ for each test case.

PREREQUISITES:

Dynamic programming, stacks

PROBLEM:

You have an array A.
The score of a partition of A into k subarrays $B_1, B_2, \ldots, B_k$ is

$$\sum_{i=1}^{k} (1 - \max(B_i))$$

Find the maximum score across all partitions of A into subarrays.

EXPLANATION:

Given that we're splitting into subarrays, there's a natural dynamic programming setup.
Let dp_i be the maximum score obtainable by partitioning the first i elements of A, with $dp_0 = 0$ being the base case for the empty subarray.

Then, by considering all possible split points, we have

$$dp_i = 1 + \max_{0 \le j < i} (dp_j - \max(A[j+1 : i]))$$

This is easy to compute in $\mathrm{O}(N^2)$, but we need to optimize it further.

Let's look at the structure of the subarrays we consider.
Suppose we fix an index i. Then, as we move the left endpoint j from i to 1, the value of $\max(A[j : i])$ will only increase — after all, we're only adding elements.

So, we have several distinct maximums to consider, and for each maximum, a *range* of elements where it's the maximum.
Note that for a fixed maximum M, with corresponding range $[L, R]$, we only really care about the maximum dp_j value for $L - 1 \le j \le R - 1$.

Let's keep the following information:

- The current list of maximums, in descending order from left to right.
  Notice that this can be represented using a stack.
- The maximum dp value corresponding to the range of each maximum.
- The set of $(max(dp) - M)$ values across all maximums.
  Let this set be S.

Then, when moving from index $i - 1$ to index i:

- Push A_i onto the stack, as a new maximum.
  The singular dp value corresponding to it is dp_{i-1}.
- Then, while the top element of the stack is not less then the previous element, their ranges can be combined into a single one (with A_i being the maximum value of the new range).
  When combining segments, the new maximum dp value of the segment equals the larger of the previous ones.
- Finally, we keep the set S updated.
  Each time, two elements are removed from the stack and one is added to it.

This corresponds to deleting two elements from S and inserting one; which can be done quickly using a `multiset`.
- Once the above process is finished, we simply have $dp_i = 1 + \max(S)$.

It's easy to see that $\mathrm{O}(N)$ push/pop operations are done on the stack in total; and each corresponds to one multiset operation for $\mathrm{O}(N \log N)$ in total.

There are also ways to implement this using other data structures, such as a segment tree. A couple of these implementations are linked below; in the author's and tester's code.
The base idea of maintaining a stack of maximums remains the same, however.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Dynamic Programming

PROBLEM:

You are given two positive integers N and K.

- Consider a set S of segments, initially empty. For each i ($1 \le i \le N-1$) such that $P_i < P_{i+1}$, insert the segment $[P_i, P_{i+1}]$ into S. Then, if you pick any subset of K segments of S, their intersection is empty.

Count the number of *good* permutations $P_1, P_2, \ldots, P_N$. As the result can be very large, you should print it modulo $998\,244\,353$.

EXPLANATION:

We classify each element P_i in the permutation in one of four categories: start point of some segment (i.e. $P_i < P_{i+1}$ and $P_i < P_{i-1}$), end point of some segment ($P_i > P_{i-1}$ and $P_i > P_{i+1}$), both start point and end point ($P_{i-1} < P_i < P_{i+1}$), and none ($P_{i-1} > P_i > P_{i+1}$). To help with the analysis, let $P_0 = N+1$ and $P_{N+1} = 0$. Notice that the structure of the permutation is as follow:

- Some none elements
- One start element
- Some both elements
- One end element
- Some none elements

and repeat.

Let's construct the permutation as follow: we iterate values from 1 to N. For each value, we first choose its type (either start, end, both, or none), then insert it into the appropriate position in the current construction. Note that the construction doesn't necessarily follow the previously mentioned structure of a final permutation (in particular, some start element might be "unclosed", meaning that while it may be followed by some both elements, it is yet to be followed by the matching end element), but at the end the final construction should obey the structure. Because of this, we store an additional information on the current construction: the amount of unclosed start element. This has an added benefit: when we iterate by increasing values v, this amount of unclosed start element is the amount of segments in S that covers $v-1$, so we can impose a restriction on this value being no more than K.

All of these ideas point to a dynamic programming solution: let $f_{i,j}$ be the number of ways to construct the first i values (from 1 to i) such that there are j unclosed start elements. Our final answer should be $f_{N,0}$. Iterating on the previous state $f_{i-1,j}$, there are four ways to choose the role of the value i to be inserted:

- i is a none element. We can only insert i in $j+1$ positions: either the very beginning of the permutation, or right after some unclosed start elements. The transition is $f_{i,j} \mathrel{+}= f_{i-1,j} \cdot (j+1)$.
- i is a start element. Similarly, we can only insert i in $j+1$ positions like the previous case, but j has to be less than k. The transition is $f_{i,j+1} \mathrel{+}= f_{i-1,j} \cdot (j+1)$ if $j < k$.
- i is a end element. We can only insert i in j positions (since i close off unclosed start elements). The transition is $f_{i,j-1} \mathrel{+}= f_{i-1,j} \cdot j$ if $j > 0$.
- i is a both element. We can only insert i in j positions (after some unclosed start elements), but j needs to be less than k. The transition is $f_{i,j} \mathrel{+}= f_{i-1,j} \cdot j$ if $j < k$.

TIME COMPLEXITY:

Time complexity is $O(N^2)$ for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM:

There are **N** questions. The **i**th question needs **T[i]** minutes to learn and will give you **C[i] × P[i]** points in the tests. You have **W** minutes to learn the questions. Your job is to find which questions to learn to maximise your score.

EXPLANATION:

We need to choose a sub-set of **N** questions so that the total learning time does not exceed **W** and the total socore is
as large as possible. With the constraint **N ≤ 100** we cannot try out all possible sub-sets since the number of sub-sets can be **2** ^{100} which is extremely large.

We need to use dynamic programming in this problem. Let **F[i][j]** be the maximal score you can get if you spend no more than **j**
minutes to learn some questions amongst the first **i** questions. Now, there are two cases:

- If you choose to learn the **i**th question then you get maximally **F[i - 1][j - T[i]] + C[i] × P[i]** points. Be careful with the case where **j < T[i]**.
- If you do not learn the **i**th question then you get maximally **F[i-1][j]** points.

The complexity of this solution is **O(N×W)**. For the initialization you need to set **F[0][j] = 0** for all **0 ≤ j ≤ W**.

AUTHOR'S AND TESTER'S SOLUTIONS:

Author's solution can be found **here** .
Tester's solution can be found **here** .

In the solution of problem setter and the tester an optimization is used to reduce the memory complexity.
With a little modification of the algorithm above we can solve this problem using a one-dimensional array of **W** elements.
The pseudo code is given below:

```
FOR i -> 0 to W
        G[i] = 0
ENDFOR

(*)FOR i -> 1 to N
        FOR j -> W downto 0
                G[j] = max(G[j + T[i]], G[j] + C[i] × P[i])
ENDFOR
```

After the **i**thiteration of the for loop (*), the value in G array is exactly as in the **i**th
row of the F array in the original algorithm. Notice that in the second For loop we need to consider the value of j in the decreasing order
so that each time we use **G[j]** to update **G[j + T[i]]** we know that the value in **G[j]** is the optimized value of using **j** minutes and the first **(i - 1)** questions only.

PREREQUISITES:

Bitwise Operations, Dynamic Programming

PROBLEM:

Chef has an array A such that $A_i = i$.

A *transform* on the array is performed by replacing each element of the array with the **bitwise XOR** of the prefix till that element. For example, if B denotes the array after performing *transform* on array A, then, $B_i = A_1 \oplus A_2 \oplus \ldots \oplus A_i$. Similarly, if C denotes the array after performing two *transforms* on array A, then, $C_i = B_1 \oplus B_2 \oplus \ldots \oplus B_i$.

Let f^K be the array obtained by performing K *transforms* on the array A. Thus, f_X^K denotes the Xth element of the Kth transform of the array A.

Formally,

- $f_i^0 = A_i$
- $f_i^1 = A_1 \oplus A_2 \oplus \ldots \oplus A_i$
- $f_i^2 = f_1^1 \oplus f_2^1 \oplus \ldots \oplus f_i^1$
- $f_i^k = f_1^{k-1} \oplus f_2^{k-1} \oplus \ldots \oplus f_i^{k-1}$. Here, $\oplus$ denotes the bitwise XOR operation.

Given K and X, determine the value f_X^K.

EXPLANATION:

Basically in this problem, we need to calculate $f_k[i]$. By definition

$$f_k[i] = f_{k-1}[0] \oplus f_{k-1}[1] \oplus \ldots .. f_{k-1}[i]$$

Using simple combinatorics and induction, we can prove that $f_k[i]$ is just XOR of all $1 \le j \le i$ such that $\binom{k+i-j}{k}$ is odd.
This is a well known fact as a corollary of Lucas Theorem that $\binom{k+i-j}{k}$ is odd if and only if k is a submask of $(k + i - j)$.

Now we need to find XOR of all j such that $1 \le j \le i$ and $(k + i - j)$ is a supermask of k.

We can do it using $dp[bit][carry][0/1]$ which defines the XOR of all j if we processed the first bit bits, $carry$ is a carry bit for $(k + i - j)$ that we have after processing bit bits, and the last parameter is 0 if $j \le i$ or 1 otherwise.

Now we go through bits, if the current bit of k is 0, we can try to make a bit of $(k + i - j)$ equals to 0 or 1, and if the current bit of k is 1, then the bit of $(k + i - j)$ should also be 1.

TIME COMPLEXITY:

$O(logK)$, for each test case

PREREQUISITES:

Knapsack-style dynamic programming

PROBLEM:

You have two arrays R and B, both of length N. At each index, you can choose either R_i or B_i. Let X denote the sum of all chosen R_i and Y denote the sum of all chosen B_i. Maximize $\min(X, Y)$.

EXPLANATION:

The limits on N and the values are small, so a natural knapsack-style dynamic programming solution should strike you, something along the following lines:

Let $f(i, x, y)$ be a boolean function, where $f(i, x, y)$ is true if and only if you can make choices among the first i elements such that the sum of reds is exactly x and the sum of blues is exactly y.
Transitions are extremely easy: $f(i, x, y) = f(i-1, x-R_i, y) \vee f(i-1, x, y-B_i)$ ($\vee$ denotes logical OR), and memoization naturally makes transitions $\mathrm{O}(1)$.
The final answer is the maximum value of $\min(x, y)$ across all (x, y) such that $f(N, x, y)$ is true.

While this is correct, it is also too slow. x and y can be as large as $200 \times N$, so we have $200^2 \times N^3$ states in our dp, which is way too much.

Note that the constraints do allow a solution in $\mathrm{O}(200 \times N^2)$, i.e, kicking out one state of our dp.
We can achieve that by a relatively common trick: turn the removed state into the value of the dp!

Consider a function $f(i, x)$ which denotes the maximum sum of blues from the first i elements, given that the sum of reds is x.
Transitions for this function are as follows:

- If we choose R_i, the sum of blues is $f(i-1, x-R_i)$
- Otherwise, the sum of blues is $f(i-1, x) + B_i$
- So, $f(i, x) = \max(f(i-1, x) + B_i, f(i-1, x-R_i))$

Once again, by memoizing $f(i, x)$ values, transitions are $\mathrm{O}(1)$, so both our time and space complexity are fine.

The final answer is the maximum of $\min(x, f(N, x))$ across all $0 \le x \le 200 \cdot N$.

TIME COMPLEXITY

$\mathrm{O}(N \cdot S)$ per test case, where $S = 200 \times N$.

PREREQUISITES:

Sorting, sweep line and/or Dijkstra's algorithm

PROBLEM:

There are k points on the 2-D plane. The i-th point has a cost of c_i and energy e_i. You start at $(0, 0)$ and want to reach (N, M).

Moving one step up or right reduces energy by 1, while moving left or down increases it by 1. At the i-th of the the N points above, you can also choose to pay c_i and reset your energy to e_i.

Find the minimum cost to reach (N, M) while ensuring that your energy never becomes negative.

EXPLANATION:

Let us define the *height* of the point (x, y) to be $x + y$.
Note that moving from a point with height h_1 to a point with height h_2 will change your energy by exactly $h_2 - h_1$. In particular, if $h_2 < h_1$ it is always possible to make this movement without your energy falling negative.

A simple interpretation of the statement is to treat the points and costs as a weighted graph: create a graph with $k + 1$ vertices (where the first k are the given points and the $k + 1$-th is the destination), and edges as follows:

- Let h_i denote the height of the i-th point, as defined above.
- For $1 \le u, v \le k + 1$, if $h_v \le h_u + e_u$, create an edge from u to v with weight c_u. Essentially, this says that we can move directly from u to v without having to reset energy at a different point.

The answer is now the shortest path from $(0, 0)$ to (N, M) in this graph.

However, this graph can have $\Omega(k^2)$ edges, which is too many. We need to optimize the above solution a bit.

Let $dist_i$ denote the shortest distance to point i. Let us also sort the points in increasing height.
We have the following observation:

- Under the above sort, $dist_i \le dist_j$ for $i \le j$. This follows immediately from the very first point made, since we can always go to a lower height for no cost.
- Next, let's look at how dijkstra would work on this graph. When at i, we only need to update some vertices greater than i. But, since vertices are sorted by height, we in fact update a contiguous range of indices, starting from $i + 1$ - and all these updates are of the same value (namely, $dist_i + c_i$).

Updates of this form can be done with the help of a sweepline algorithm.
When we are at index i, we need to do the following:

- Find the largest index j such that $h_j \le h_i + e_i$. This can be done with binary search.
- Now, for each $i + 1 \le k \le j$, we need to set $dist_k \leftarrow \min(dist_k, dist_i + c_i)$.
- Looking at it differently, when we are at index i, $dist_i$ is the minimum value over all updates that cover index i.
- So, we can maintain a set of the current updates and the expiry time of the update (i.e, the last position it is valid for, j in the first step). At position i, we do the following:
  - Remove all updates from the set that have already expired before i.
  - Set $dist_i$ to be the minimum value among the updates set.
  - Compute j as described in the first step.
  - Insert $dist_i + c_i$ to the updates set, and set it to expire at $j + 1$

This solves the problem in $\mathrm{O}(k \log k)$.

TIME COMPLEXITY

$O(k \log k)$ per test case.

PREREQUISITES:

Dynamic programming, range-max queries

PROBLEM:

You're given an array S. For Q queries of (L, R), answer the following:

- For a fixed value of D, you start from L with a score of D and move towards R. At each index, $S_i \leq D$ must hold. Further, if $S_i = D$, reduce D by 1.
  Find the smallest value of D that allows you to reach R from L.

EXPLANATION:

Let $M = \max(S_L, S_{L+1}, \ldots, S_R)$.
Clearly, the answer to query (L, R) must be at least M, since we will encounter an M along our way and need to be $\geq M$ when we do.

Further, it's also easy to see that $D = M + 1$ always allows us to reach R, since it's strictly larger than anything in this range and hence will never reduce.
So, our task reduces to checking whether $D = M$ allows us to reach R or not.

Let's see how this process would go.

- Let $i_1 \geq L$ be the first index such that $S_{i_1} = M$. We will definitely reach this index, after which we'd be left with $D = M - 1$.
- From here, we move right till we
  - Reach R, at which point we're done
  - Reach an occurrence of something $> D$, at which point we can't continue
  - Reach another occurrence of D, at which point we move to $D - 1$; and then this process repeats.

This is easy to simulate in $\mathrm{O}(N)$, but we need to do better.

Notice that once we reach the first position where $S_i = M$, the process is rather uniquely defined, because we can pretend we start there with $D = M$.

So, let's find for each index i, the furthest we can travel if we start from i with value S_i.
Let this be dp_i.

dp_i can be computed as follows:

- Let $j > i$ be the first index such that $S_j \geq S_i$
- Let $k > i$ be the first index such that $S_k = S_i - 1$.
- Then, $dp_i = \min(j - 1, dp_k)$, because we can't cross index j no matter what; and if we reach k we're essentially starting from there instead, so we can't go beyond dp_k either.

dp_i can be precomputed quite easily for all indices by iterating i from N down to 1. The only things that need to be taken care of are quickly computing indices j and k.

- Computing k is easy. Since we're iterating in reverse anyway, simply keep a record (using, say, a map) of the last occurrence of every element. Let this be $\text{last}[x]$; then $k = \text{last}[S_i - 1]$.
  Note that after processing index i, you should set $\text{last}[S_i] = i$.
- Computing j needs a bit more effort, but in the case of this specific problem, can be actually done easier.
  Note that rather than finding the next element that's $\geq A_i$, it suffices to find the next element that's equal to

S_i, i.e, $\text{last}[S_i]$ itself. Do you see why?

At any rate, by maintaining the last map, all the dp values can be computed quickly.

After this, for a query (L, R) we need to:

- Find the maximum M on the range $[L, R]$, which is a standard data structure task (use a segment tree/sparse table/whatever).
- Find the leftmost occurrence of this maximum, which is also not too hard: for example, the segment tree/sparse table merge function can be modified slightly to return this, by maintaining pairs of (value, index).
- Let i be the index we found. Then, if $dp_i \geq R$ the answer is M, otherwise the answer is $M + 1$.

TIME COMPLEXITY

$\text{O}((N + Q) \log N)$ per test case.

PREREQUISITES:
Graph Theory

PROBLEM:

Define a beautiful walk in a directed graph as a walk that contains all the vertices atleast once and has $\leq \frac{n^2}{4}$ edges. Given a directed graph G, find a beautiful walk in it or claim that no such walk exists.

EXPLANATION:

Define a covering walk as one that contains all vertices at least once.

Consider the SCC DAG (each strongly connected component condensed into a single vertex) for this directed graph.

In the topological ordering, there must be an edge from component i to component $i+1$ for each i, otherwise one can not come back to $i+1$ after they have left component i.

We claim that this condition is also sufficient for there to exist a beautiful walk.

First, say the graph was strongly connected. Consider a walk which first traverses its diameter and then visits each unvisited vertex one by one. Let d be the diameter. It takes d edges to traverse the diameter, and then $\leq d$ edges each for the unvisited nodes. Since there are $n-1-d$ unvisited vertices, the total number of edges in this walk is $\leq d + (n-1-d)d = d(n-d) \leq \frac{n^2}{4}$.

Let the walk found above be $w_1, w_2, \ldots, w_l$.

Now, say we had to find a covering walk in an SCC that starts at a given node s. We claim that this can be done in $\leq \frac{(n+1)^2}{4} - 1$ edges. If there is a diameter starting at s, we can use that to find a walk with $\leq \frac{n^2}{4}$ edges.
Else, there is a path from s to w_1 with $< d$ edges. So, prepending this path to our walk $w_1, \ldots w_l$, we have the total length $\leq d - 1 + d + (n-1-d)d = d(n+1-d) - 1 \leq \frac{(n+1)^2}{4} - 1$ edges.

Similarly, if we have to find a covering walk ending at a given node t, we can also do that in $\leq \frac{(n+1)^2}{4} - 1$ edges.

If we have to find a covering walk starting at a given node s and ending at a given node t, we can do it in $\leq \frac{(n+2)^2}{4} - 2$ using a similar logic(Prepend $s - w_1$ walk and append $w_l - t$ walk. If anyone of these has a length d, we can use that instead of original walk, else we have $\leq 2(d-1) + d + (n-1-d)d \leq \frac{(n+2)^2}{4} - 2$ edges).

Now, say we had $k \geq 2$ SCCs in the graph, in a chain, having sizes $x_1, x_2, \ldots x_k$.
Find any starting and ending nodes in the middle SCCs. Clearly, the first and the last SCC have a constraint on one end of the walk whereas the middle SCCs have a constraint on both endpoints. So, from the previous conclusions, our walk has a maximum of :

$$\frac{(x_1+1)^2}{4} - 1 + \frac{(x_k+1)^2}{4} - 1 + \sum_{i=2}^{k-1} (\frac{(x_i+2)^2}{4} - 2) + k - 1$$

edges.

We know that the sum of squares is maximized when the sum is concentrated around one variable. So, the worst possible cases we need to consider are $x = (n-1, 1)$ and $x = (1, n-2, 1)$. In both of these cases, the above value comes out to be exactly $\lfloor \frac{n^2}{4} \rfloor$.

The overall complexity is $O(nm)$ as we do a bfs in every SCC from all of its nodes to find the required paths.

AUTHOR'S SOLUTION:

Author's solution can be found **here**

PREREQUISITES:

Dynamic programming

PROBLEM:

You have an array A. In one move, you can choose two indices i and j with $i \leq j$ and $A_i = A_j$, and set every element in the segment $[i,j]$ to A_i.

Find the minimum number of moves required to sort A, or report that it isn't possible.

EXPLANATION:

This problem can be solved with the help of dynamic programming.

Let dp_i denote the minimum number of moves requred to sort the first i elements of A. We want to compute dp_N .

The transitions are as follows:

Transitions

The above transitions give us a solution in $\mathrm{O}(N^2)$, which is too slow. However, note that:

- The first case is always $\mathrm{O}(1)$ and doesn't really need to be optimized
- The second case is the one that might take $\mathrm{O}(N)$. However, we are constrained to look at only those indices $j < i$ such that $A_j = A_i$, and want the minimum dp_j across these indices.
- So, we can simply maintain an array mn, where mn_x denotes the minimum value of dp_j across all indices with $A_j = x$ that we have *seen so far*.
- Updating mn is simple: once we have computed dp_i for an index i, simply set $mn_{A_i} \leftarrow \min(mn_{A_i}, dp_i)$

With this optimization, our solution is now $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

BFS, Dynamic programming

PROBLEM:

You have N integers. In one move, you can choose A_i and set it to either A_i^2 or $\lfloor \sqrt{A_i} \rfloor$.

Find the minimum number of moves required to sort A.

EXPLANATION:

The single most important observation here is that a given index can't take too many values: for the given limits, there are ≤ 40 possibilities for each index.

Proof

We can make a couple of observations for any positive integer x.

- $\lfloor \sqrt{x^2} \rfloor = x$. This means that any sequence of operations on x can be replaced by an equivalent sequence of operations in which *every square root operation is before every squaring operation*.
- If we don't use the square root operation, the integers we can reach are $x, x^2, x^4, x^8, \ldots$, i.e, integers of the form x^{2^k} for some $k \geq 0$.

If $x = 1$ the only integer that can be reached is 1, so we only look at $x > 1$.
If $x > 1$, then note that $x^{2^6} = x^{64} > 10^{18}$ no matter what x is. So, repeated squaring only allows us to reach at most 6 values: $x, x^2, x^4, x^8, x^{16}, x^{32}$.

Similar reasoning should tell you that repeatedly taking the square root of x will give you at most 6 distinct values before you hit 1.
Since any valid integer can be created by taking the square root some times and then squaring some times, this gives us an upper bound of $6 \times 6 = 36$ reachable values.

Further, while it isn't needed, a slight modification of the argument allows us to prove a tighter bound: $6 + 5 + 4 + 3 + 2 + 1 = 21$, which along with $x = 1$ gives us 22. Do you see how?

With this in mind, let's first find, for each $1 \leq i \leq N$, the possible values that can occur at this position (and the minimum number of steps needed to reach each value).
Let S_i denote the set of values that can occur at position i, and $cost(i, x)$ denote the number of steps needed to turn A_i into x.

How to compute these?

We use bfs!

More specifically, consider the (infinite) directed graph on the set of positive integers, where there's an edge from x to y if and only if $y = x^2$ or $y = \lfloor \sqrt{x} \rfloor$.

Then, note that S_i is exactly the set of nodes that can be reached from A_i in this graph, and $cost(i, x)$ is simply the shortest path from A_i to x.
So, simply start a bfs from A_i to compute distances: while the graph itself is impossible to hold in memory, we only care about those vertices that can be reached from A_i, which as noted above is pretty small. Edges are defined implicitly and can be computed as we do the bfs.

This bfs will take $\mathrm{O}(|S_i|)$ time for a given index i, and $|S_i| \leq 40$ so this is $\leq 40N$ vertices visited in total, which is perfectly fine.

Once we have this information, the rest of the problem turns into a relatively simply dp.

Let $dp_{i,x}$ denote the minimum number of moves needed to sort the first i elements of the array, such that the value at position i is x. The final answer is the minimum value of $dp_{N,x}$ across all $x \in S_N$.

The transitions are simple:

$$dp_{i,x} = cost(i,x) + \min_{\substack{y \in S_{i-1} \\ y \le x}} (dp_{i-1,y})$$

Or in simpler words: the minimum cost to sort the first i elements and have x end up at position i equals the cost of turning A_i into x, plus the minimum cost of sorting the first $i-1$ elements such that position $i-1$ contains an element not larger than x.
The first part is $cost(i,x)$ which we precomputed, and the second part is exactly what our dp contains so we obtain a recursion.

Let M be the maximum number of possibilities for a single index. Note that $M \le 22$.
The algorithm above, if implemented directly as described, uses $\mathrm{O}(NM)$ space and has a worst-case complexity of $\mathrm{O}(NM^2)$.

It is possible to improve both of these:

- Note that computing $dp_{i,x}$ depends on only the values of dp_{i-1}. So, we can discard all previous rows of the dp table, which brings the memory down to $\mathrm{O}(N+M)$.
- Computing the transitions itself can be improved to $\mathrm{O}(NM)$ by sorting the sets S_i and iterating in increasing order to compute dp_i, and then maintaining two pointers (or with binary search for an additional $\log$ factor, which should still be fast enough).

The second step is explained more in-depth in the editorial of **DIVSORT** (which is a pretty similar problem, and has essentially the same dp).

If your solution matches the above and you're getting WA, it might be due to use of the `sqrt` function: read through **this blog** to see how to fix it.

TIME COMPLEXITY

$\mathrm{O}(NM)$ or $\mathrm{O}(NM \log M)$ per test case, where $M \le 22$ is the maximum possible number of values an index can have.

PREREQUISITES:

Counting, Dynamic Programming, Recursion.

PROBLEM:

Given strings X and Y, count the number of strings of length n which do not contain string $S(X \leq S \leq Y)$ of length $m(\leq n)$ as a substring.

EXPLANATION:

In this problem it is asked to count the number of strings of length n which do not contain string $S(X \leq S \leq Y)$ of length $m(\leq n)$ as a substring. Let H be a set of strings which are lexicographically in between X and Y inclusively that is $H = \{S : |S| = m \wedge X \leq S \leq Y\}$. Now define $\mathrm{dp}[i]$ as the number of strings of length exactly equal to i which do not contain string(s) which belong(s) to H as a substring. It is clear that if $i < m$ then $\mathrm{dp}[i] = 26^i$.

For $i \geq m$, define a recursive function $f(S, n, i)$ which takes input string s, ending index n and index i which is to be filled. This function returns the number of valid strings(which are not hated) whose last $n - i$ letter are already fixed as S. For example $f(AB, 5, 3)$ will return number of valid strings of length 6(assuming zero indexing and $n = 5$) whose last 2 letters are fixed as “AB”. Also define a function $V(S, len)$ which return 1 if string S of length len is valid else it returns 0. Note that if $len < m$ then the string is trivially valid else we have to check $X \leq S \leq Y$. Further len will always be less than or equal to m by virtue of following definition of $f(S, n, i)$ where a value greater than m is never passed to $V(S, len)$

Now $f(S, n, i) = 24\mathrm{dp}[i] + f(AS, n, i-1) \times \mathrm{V}(AS, \min(m, \mathrm{len}(AS)) + f(BS, n, i-1) \times \mathrm{V}(BS, \min(m, \mathrm{len}(BS))$ because the ith index can be C-Z, in which case we are left with string of length i (index 0 to $i-1$) thats why the term $24\mathrm{dp}[i]$; or ith index can be ‘A’ in that case we have to check whether the string from ith index to min(i+m-1, n) is valid or not thats why the term $f(AS, n, i-1) \times \mathrm{V}(AS, \min(m, \mathrm{len}(AS))$ and similar expression when ith index is ‘B’. Base case will be when $i = -1$ in that case return 1.

Now, for $i \geq m$, $\mathrm{dp}[i] = f(\text{empty string}, i-1, i-1)$. So, the final answer to the question will be $\mathrm{dp}[n]$. Refer to the editorialist code in the solution section for more detail. The worst case time complexity of this algorithm is $\mathrm{O}(n2^n)$.

Solution:

Setter’s solution can be found **here**
Tester’s solution can be found **here**
Editorialist’s solution can be found **here**

PREREQUISITES:

Dynamic programming, preprocessing, binary search, cumulative sums

PROBLEM:

You are given a string S of length N consisting only of 0 s and 1 s. You are also given an integer K.

You have to answer Q queries. In the i th query, two integers L and R are given. Then you should print the number of substrings of $S[L, R]$ which contain at most K 0 s and at most K 1 s where $S[L, R]$ denotes the substring from L th to R th characters of the string S.

In other words, you have to count number of pairs (i, j) of integers such that $L \leq i \leq j \leq R$ such that no character in substring $S[i, j]$ occurs more than K times.

QUICK EXPLANATION:

Let $\text{far}[i]$ be the last index j such that $S[i, j]$ has at most K 0 s and at most K 1 s. Then for a given query (L, R), the number of *valid* strings starting at index i is $\min(R, \text{far}[i]) - i + 1$ (for $L \leq i \leq R$). Therefore, the answer for the query (L, R) is the following:

$$\sum_{i=L}^{R} (\min(R, \text{far}[i]) - i + 1)$$

Note that $\text{far}[i]$ never decreases, so we can use binary search to find the maximum index k such that $\text{far}[i] \leq R$. The sum then becomes:

$$\sum_{i=L}^{k} (\text{far}[i] - i + 1) + \sum_{i=k+1}^{R} (R - i + 1)$$

Each of these is solvable in closed form, except possibly for range sums of $\text{far}[i]$. But for that, we can simply compute *cumulative sums* of $\text{far}[i]$.

The algorithm runs in $O(N + Q \log N)$, but can be sped up to $O(N + Q)$ by getting rid of binary search. We'll describe this below.

EXPLANATION:

We will begin by describing a slow solution, and then work on improving it incrementally. We will call a string **valid** if there are at most K 0 s and at most K 1 s.

Slow solution

First, we can simply test each substring and count all those that are valid. This can be accomplished, for example, with the following:

```
def answer_query(L, R):
    answer = 0
    for i in L...R
        for j in i...R
            # try the substring S[i, j]
            count0 = count1 = 0
```

```
            for k in i...j
                if S[k] == '1'
                    count1 += 1
                else
                    count0 += 1

            if count0 <= K and count1 <= K
                answer += 1

    return answer
```

How fast does this go? This runs proportionally to the sum of the length of all substrings of $S[L, R]$, and one can easily compute it to be approximately $\frac{(R-L+1)^3}{6}$. Since $R - L + 1$ is at most N, in the worst case the running time is approximately $\frac{N^3}{6}$ steps. This won't pass any of the subtasks!

Breaking early

The above brute-force solution can be optimized to pass subtask 2, by the observation that valid strings are at most $2K$ in length. Therefore, we can simply check all substrings that are at most $2K$ in length, and there are $O(NK)$ of them:

```
def answer_query(L, R):
    answer = 0
    for i in L...R
        # process only strings up to min(2K-1,R)
        for j in i...min(i+2*K-1, R)
            # try the substring S[i, j]
            count0 = count1 = 0
            for k in i...j
                if S[k] == '1'
                    count1 += 1
                else
                    count0 += 1

                if count0 > K or count1 > K:
                    break

            if count0 <= K and count1 <= K
                answer += 1

    return answer
```

Notice that aside from reducing the limit of j to $\min(2K + 1, R)$, we have also added a `break` statement, because when either `count0` or `count1` exceeds K, then there is no more point in proceeding to inspect the rest of the letters, since we already know that the string wouldn't be valid.

Thus the algorithm now runs in $O(NK \min(N, K))$ time per query and $O(QNK \min(N, K))$ time overall, and passes subtask 2!

Dynamic programming

In fact, we can extend the argument we used about the `break` statement! Notice that the substring of a valid string is also valid, and a superstring of any invalid string is also invalid! Therefore, we can break out of the j loop once we encounter an invalid string! However, this alone won't improve the code in the worst case, because it might be the case that we don't encounter an invalid string, or we encounter such a string late enough.

However, we can exploit even more properties of the substrings we are inspecting! Notice that to check whether a substring $S[i,j]$ is valid or not, we only need to count the 0 s and 1 s in it. However, this can be computed easily once we know these counts for the substring $S[i,j-1]$! Specifically, as we process the substrings $S[i,j]$ for a fixed i and increasing j, we only need to increment `count0` or `count1` depending on $S[i]$:

```
def answer_query(L, R):
    answer = 0
    for i in L...R
        count0 = count1 = 0
        for j in i...min(i+2*K-1, R)
            # try the substring S[i, j]
            if S[j] == '1'
                count1 += 1
            else
                count0 += 1

            if count0 <= K and count1 <= K
                answer += 1
            else
                break

    return answer
```

Now, this runs in $O(N\min(N,K))$ time per query, and $O(QN\min(N,K))$ time overall!

However, using this might still not pass subtask 1, because there are up to 10^5 queries. Thankfully, in subtask 1, N is at most 100, so there are only at most $N(N-1)/2$ substrings $S[L,R]$. Thus, we can simply precompute the answer for all those substrings, and then answer the Q queries with a simple lookup. This approach should run in $O(N^3\min(N,K)+Q)$ and should be able to pass subtask 1!

Walking algorithm

There is still a way to optimize the above! Remember what we said above, that the substring of a valid string is also valid, and a superstring of any invalid string is also invalid. Therefore, if $S[i,j]$ is valid, then $S[i+1,j]$ is also valid!. Therefore, when we process the next i, we don't have to start j from i any more, because we already know many strings are valid from the previous i. Specifically, if j' is the last j such that $S[i,j]$ is valid, then $S[i+1,j']$ is also valid, so we can simply start iterating j from $j'+1$ onwards. What's more, we can compute `count0` and `count1` of $S[i+1,j'+1]$ from $S[i,j'+1]$ by simply decrementing one of them depending on $S[i]$! This is illustrated in the following code:

```
def answer_query(L, R):
    answer = 0
    j = L
    count0 = count1 = 0
    if S[L] == '1'
        count1 += 1
    else
        count0 += 1
    for i in L...R
        while j <= R and count0 <= K and count1 <= K:
            j += 1
            if j > R
                break
            if S[j] == '1'
                count1 += 1
            else
                count0 += 1
```

```
        # at this point, we know S[i, j-1] is valid but S[i, j] is invalid
        answer += j - i

        # decrement
        if S[i] == '1'
            count1 -= 1
        else
            count0 -= 1

    return answer
```

Now, how fast does this go? Notice that there are still nested loops. However, every time the inner loop iterates, j increases by 1. Therefore, the inner loop runs in at most $R - L$ steps, or $O(N)$. Therefore, the whole algorithm runs in $O(N)$ time per query, and $O(QN)$ time overall! This should be able to pass subtask 1, 2 and 3.

Preprocessing and binary search

The previous algorithm is too slow for subtask 4, because it still takes $O(N)$ time per query. In fact, this is probably the fastest we can do without some sort of *preprocessing*, because at the very least we have to *read* the string $S[L, R]$ to compute the answer, and this already takes $O(N)$ time. Thus, we will try to speed up the algorithm by preprocessing.

First, note that the crucial part of the previous solution is finding, for each i, the first j such that $S[i,j]$ is invalid, or $j > R$. However, by ignoring first the "or $j > R$" part, we see that the first such j for each i only depends on the string S! For a given i, let's denote that j by $\text{far}[i]$ (if you read the quick explanation above, note that this far is different from the far there. Specifically, this one is larger by exactly 1.). Thus, we can try to precompute far at the beginning:

```
far[1...N]

def precompute():
    count0 = count1 = 0
    j = 1
    if S[1] == '1'
        count1 += 1
    else
        count0 += 1
    for i in 1...N
        while j <= N and count0 <= K and count1 <= K:
            j += 1
            if j > N
                break
            if S[j] == '1'
                count1 += 1
            else
                count0 += 1

        # at this point, we know S[i, j-1] is valid but S[i, j] is invalid
        far[i] = j

        # decrement
        if S[i] == '1'
            count1 -= 1
        else
            count0 -= 1
```

and use that for our queries:

```
def answer_query(L, R):
    answer = 0
    for i in L...R
        j = min(far[i], R+1) # R+1 is the first j such that j > R
        answer += j - i
    return answer
```

The precomputation works similarly to the previous code and runs in $O(N)$, but the queries still takes $O(N)$ time each. But now, the `answer_query` code is much simpler, and in fact can be expressed by the following mathematical expression

$$\sum_{i=L}^{R} (\min(\text{far}[i], R+1) - i)$$

We can now compute such a sum using some simple manipulations:

$$\begin{aligned} &= \sum_{i=L}^{R} \min(\text{far}[i], R+1) - \sum_{i=L}^{R} i \\ &= \sum_{i=L}^{R} \min(\text{far}[i], R+1) - (\frac{R(R+1)}{2} - \frac{L(L-1)}{2}) \end{aligned}$$

However, we still got this nasty $\min$ term which we need to take care of. Thankfully, we can use the fact that $\text{far}[i]$ **never decreases**, to know that $\text{far}[i]$ will be $\leq R$ at the beginning, and then as i increases it will eventually exceed R, and once it does, it stays greater than R. Thus, it makes sense to find the last k such that $\text{far}[k] \leq R$, so the above expression becomes

$$\begin{aligned} &= \sum_{i=L}^{k} \text{far}[i] + \sum_{i=k+1}^{R} (R+1) - (\frac{R(R+1)}{2} - \frac{L(L-1)}{2}) \\ &= \sum_{i=L}^{k} \text{far}[i] + (R-k)(R+1) - (\frac{R(R+1)}{2} - \frac{L(L-1)}{2}) \end{aligned}$$

Now, we are almost down to $O(1)$ computation, aside from two things: finding the k and a range sum for $\text{far}[i]$. But these are simple. First, k can be computed with binary search, as the index such that $\text{far}[k] \leq R < \text{far}[k+1]$, because $\text{far}[i]$ is monotonically nondecreasing. Also, to compute range sums for $\text{far}[i]$, one can simply use *cumulative sums* or *prefix sums*: Let $\text{sumfar}[i]$ be the sum of the fars until the i th index. Then the sum $\text{far}[i] + \text{far}[i+1] + \cdots + \text{far}[j]$ is simply $\text{sumfar}[j] - \text{sumfar}[i-1]$!

These are illustrated in the following code:

```
far[1...N]
sumfar[0...N]

def precompute():

    # precompute far
    count0 = count1 = 0
    j = 1
    if S[1] == '1'
        count1 += 1
    else
```

```
        count0 += 1
    for i in 1...N
        while j <= N and count0 <= K and count1 <= K:
            j += 1
            if j > N
                break
            if S[j] == '1'
                count1 += 1
            else
                count0 += 1

        far[i] = j

        # decrement
        if S[i] == '1'
            count1 -= 1
        else
            count0 -= 1

    # precompute sumfar
    sumfar[0] = 0
    for i in 1...N
        sumfar[i] = sumfar[i-1] + far[i]

def answer_query(L, R):

    # binary search to find k such that far[k] <= R < far[k+1]
    # we maintain the invariant far[k1] <= R < far[k2]
    k1 = L-1
    k2 = R+1
    while k2 - k1 > 1:
        km = (k1 + k2) / 2 # here, "/" floor division
        if far[km] <= R
            k1 = km
        else
            k2 = km

    k = k1 # k is now equal to k1 because k2 - k1 = 1 and far[k1] <= R < far[k2]

    answer = sumfar[k] - sumfar[L-1] + (R-k)*(R+1) - (R*(R+1)/2 - L*(L-1)/2)

    return answer
```

Using this, one can see that precomputation still runs in $O(N)$ time, and queries now run in $O(\log N)$ time each, due to the binary search. Thus, the overall algorithm runs in $O(N + Q \log N)$ time, which comfortably passes all the subtasks!

Be careful with overflows! Use the right data type for this.

Bonus: $\text{far}[i]$ and $\text{raf}[i]$

The above solution already works, but we will introduce a final optimization here. Specifically, we will try to improve our algorithm to compute the k in each query.

Note that k is the largest index such that $\text{far}[k] \leq R$ or $k < L$. The key idea is that, by ignoring first the "or $k < L$" part, we see that k is only dependent on R! Thus it would be nice if we are able to precompute the k s for all possible R s, and in fact it is easy to do so.

Let's define a similar array raf, where $\text{raf}[R]$ is the smallest index i such that $\text{far}[i] > R$. Using this array, one can compute k simply as $\max(L, \text{raf}[R]) - 1$ (we'll leave this to the reader to see why). Now, how do we compute all the raf s? The idea is that raf is essentially the *reverse* of far, only that the direction is to the left rather than to the right. Thus, a similar $O(N)$ time walking algorithm can be used to compute it. We will leave this as an exercise for the reader, however, because we will show here a different way to compute it in $O(N)$ time, by exploiting the relationships between far and raf. See the following pseudocode for details:

```
far[1...N]
raf[1...N]
sumfar[0...N]

def precompute():

    # precompute far
    .....

    # precompute sumfar
    ....

    # precompute raf
    # initialize
    for i in 1...N
        raf[i] = -1

    # we know that far[i]-1 < far[i], so if j = far[i]-1,
    # then raf[j] must be <= i, (because raf[j] is the least such i)
    # we process each i in decreasing order to guarantee that we assign the least such i
    for i in N...1 by -1
        raf[far[i]-1] = i

    # for all the raf[i]'s that we didn't encounter, set its value to raf[i+1]
    # because raf[i] <= raf[i+1]
    for i in N...1 by -1
        if raf[i] == -1
            raf[i] = raf[i+1]

def answer_query(L, R):

    k = max(L, raf[R])-1
    answer = sumfar[k] - sumfar[L-1] + (R-k)*(R+1) - (R*(R+1)/2 - L*(L-1)/2)
    return answer
```

Finally, one can now see that it runs in $O(1)$ time per query, and $O(N + Q)$ time overall!

Time Complexity:

$O(N + Q)$

PREREQUISITES:

Dynamic programming

PROBLEM:

Kulyash has given you an array A of size N.

He defines the *subsequence-number* of a **non-empty** subsequence S of array A as the number formed by the **concatenation** of all the elements of the subsequence S.

Find the count of non-empty subsequences of A having their *subsequence-numbers* **divisible** by 7. Since the answer can be huge, output it modulo $10^9 + 7$.

For example: Consider $A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. A subsequence S of A is $[2, 5, 7, 10]$.
The *subsequence-number* of this subsequence is 25710.

QUICK EXPLANATION:

Let $dp[i][j]$ represent the number of subsequences in the prefix P_i of the array A which leave a remainder j when there subsequence-number is divided by 7.
Let D be the number of digits in the ith element.
Initialize $dp[0][0] = 1$ and rest as 0 (the empty subsequence, this will be subtracted from the answer later).
The dp transition is : for each j from 0 to 6
$dp[i][j * 10^D + A_i] = dp[i-1][j] + dp[i-1][j * 10^D + A_i]$

The answer is $dp[N][0] - 1$

EXPLANATION:

For each value of remainder, rem = 0 to 6, let us calculate the number of subsequences in the prefix P_i (first i elements of the array A) which leave a remainder rem when their subsequence-number is divided by 7 . The subsequences till a particular index i leaving remainder rem include the subsequences calculated for prefix P_{i-1} for the same remainder rem.
Let D be the number of digits in the ith element. Let the number of subsequences in P_{i-1} which leave a remainder R when their subsequence-number is divided by 7 be X, Then the remainder when the subsequence-number formed by combining the ith element with an earlier subsequence from these X subsequences is divided by 7 is $(R * 10^D + A_i)\%7$. Hence, we need to add X to the number of subsequences in P_i which leave a remainder $(R * 10^D + A_i)\%7$ when their subsequence-number is divided by 7.

This problem has both optimal substructure property and overlapping subproblems. These kind of problems can be solved using dynamic programming.

Let $dp[i][j]$ represent the number of subsequences in the prefix P_i of the array A which leave a remainder j when there subsequence-number is divided by 7.
Initialize $dp[0][0] = 1$ and rest as 0 (the empty subsequence, this will be subtracted from the answer later).
The dp transition is : for each j from 0 to 6
$dp[i][j * 10^D + A_i] = dp[i-1][j] + dp[i-1][j * 10^D + A_i]$

The answer is $dp[N][0] - 1$

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Greedy and/or dynamic programming

PROBLEM:

You have an array A. In one move, you can choose an index i, subtract 1 from A_i, and subtract 2 from A_{i+1}. Find the minimum possible value of $|A_1| + |A_2| + \ldots + |A_N|$ after performing some operations.

EXPLANATION:

This problem can be solved with dynamic programming, though a greedy solution exists as well.

Both solutions use the fact that it is natural to think of iterating from the back to the front (i.e, from N to 1, since reducing some A_i by 2 is 'better' than reducing it by 1)

Dynamic programming

First, there's a fairly simple slow solution that comes to mind:

Let $dp_{i,x}$ denote the minimum possible sum of $|A_i| + |A_{i+1}| + \ldots + |A_N|$, assuming that exactly x operations are performed on index i (i.e, subtract x from A_i and $2x$ from A_{i+1}).

$dp_{i,x}$ can be easily computed from $dp_{i+1,y}$, since we know the values of $|A_i|$ and $|A_{i+1}|$ after knowing x and y (and the contribution of elements at positions greater than $i+1$ will be contained in $dp_{i+1,y}$. Of course, this solution is much too slow.

To optimize it, note the following when we are at position i:

- If $A_{i+1} \leq 0$, it is better to not perform any operation on position i, since our 'profit' for each operation performed is always going to be negative
- If $A_{i+1} > 0$, it is optimal to reduce it to either $1, 0$, or -1. This is because our profit for these moves is always non-negative

So, for each $dp_{i+1,y}$, there are at most 3 values of x whose $dp_{i,x}$ needs to be considered. In fact, it is at most two values: if y is even, only $x = y/2$ needs to be considered, and if y is odd, only $x = \lfloor \frac{y}{2} \rfloor$ and $\lceil \frac{y}{2} \rceil$ need to be considered (bringing y down to 1 and -1, respectively).

This ensures that the total number of states is linear, and transitions are now essentially constant time. So, the whole process runs in $\mathrm{O}(N)$ (or $\mathrm{O}(N \log N)$ if implemented with maps).

Greedy

The following greedy also passes all the tests (though I do not have a formal proof of its correctness, other than the fact that 'it seems reasonable').

- Iterate i from N to 2.
  - If $A_i \leq 0$, do nothing
  - If $A_i > 0$, bring it down to either 1 or 0 (depending on parity) by performing the appropriate number of moves on position $i-1$.
- Once the above is done, once again iterate i from N to 2
  - If $A_i = 1$ and $A_{i-1} > 0$, use one operation on position $i-1$

Now print $|A_1| + |A_2| + \ldots + |A_N|$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

LCM and Dynamic Programming

PROBLEM:

Given an array A1,A2…AN you have to print the size of the largest contiguous subarray such that LCM of all integers in that subarray is equal to the product of all integers in that subarray.

Solution:

Problem Idea: Calculating efficiently longest subarray which ends at i for each i. LCM of all integers in that subarray is equal to the product of all integers in that subarray.

Let Dp[i] denotes the length of the longest subarray ending at i and satisfying the property that the product of all integers of the subarray is LCM of the subarray.

If we know Dp[i-1] , can we calculate Dp[i] ?

Yes , Dp[i] = min (Dp[i-1]+1 , i - Cal(i)) (Array indexing starts from 1) , where Cal(i) calculates the last index encountered which is not co-prime to A[i] while travelling left from i. If no such index is found , then 0 is returned.

Let A[] = 2 3 5 4 9 5 10 , then Cal(i) for i=1,…7 will be 0,0,0,1,2,3,6.

Dp[] = 1 2 3 3 3 3 1

Ans=max(Dp)=3

How to calculate the Cal(i) efficiently ?

A number is not co-prime to another number if there exist a common gcd [or a common prime factor].

Ok, then we need to factorise A[i+1] , let say we wrote A[i+1] = $p_1^{e_1} * p_2^{e_2} * p_3^{e_3} * \ldots * p_k^{e_k}$, where p_i is prime number.

Now we need to pick up the latest index where any of these factors came.

But how can we do it efficiently ?

Look at the constraints and you will notice that A[i] <= 10^6 , so each prime is less than 10^6

We can use an array of size 10^6 {let's call it P[]} to find out the latest position. Look at the pseudo code for Cal(i).

Pseudo Code

```
int Cal(int idx)
    int ret_val = 0
    Factors = Factorise(A[idx]) //Factorise will only return prime factors
    for(int i = 0 ;i< Factors.size() ;i++ ) //iterating through all factors
        ret_val = max( ret_val , P[Factors[i]] )  //P is an array of size 10<sup>6</su
        P[Factors[i]] = idx         //latest index for prime number Factors[i] is upda
    return ret_val
```

What will be our answer ?

Max (dp[i]) for all i

Some PitFalls

You need to precompute factors of each numbers [from 1 to 10^6]. Reason : Calculating it in run time may lead to TLE because :
For factoring any number , you need to iterate over all prime numbers less than sqrt(number). In worst case this number is close to 150. Hence total number of steps will be approximately 150 * 100000 for each Test case. So, for 50 test cases this goes to 75 * 10^7 which will time out.

Another Way of understanding the Solution
Actually you can also interpret the solution as an application of two pointer method.

Two pointer method
Assume that [i, j] is the your current valid segment/ window. If you go from i to i + 1, and the segment [i + 1, j] still remains a valid segment, then you can apply this method.
So you can consider that when going from left to right, if you maintain a pointer for the right end of the good segment, then the right end of the segment always keeps on increasing.

Basically in other words, you can treat the segment as a window representing a valid solution. You need to support fast addition and deletion of elements from the window. But you will notice that
if your right pointer always increases then you will never need to delete the right end of the segment. You will always delete the left end and will always add the right end of the segment.

It is very easy to prove that complexity of this method is O(n), as that right pointer is always increasing and hence can make at most n iterations, So overall there could be at most n + n iterations over both left and right ends of a good segment which amounts to total O(n) operations. This method is called window two pointer method.

Sometimes this method is called sliding window algorithm too.

How to apply it in our problem
You can notice that if the segment [i, j] is a valid segment (ie. a segment containing numbers having their lcm equal to their product), then the segment [i + 1, j] is also valid.
When we want to extend the right end of our current segment (ie. we are looking at validity of segment [i, j + 1] considering that we know that segment [i, j] is valid), we need to check whether the current number has any common divisor (prime divisor will also work) with any number in the window. This check can be done easily by using the Cal array as explained above.

Overall complexity of this method is O(N).

PREREQUISITES:

Dynamic Programming, Game

PROBLEM:

There is a line consisting of N piles of coins on the table. For each i ($1 \le i \le N$), the i-th pile contains C_i coins, each with value V_i. There is also a bin under the table.

Alice and Bob play a game, moving alternately. A move consists of two steps:

- Throw a prefix of the piles in the bin (the prefix can be empty). Then, if there are no coins left on the table, the game ends. Note that in this step, it is not allowed to throw away only some coins in a pile — the entire pile must be thrown into the bin.
- Take exactly one coin from the first (remaining) pile. Then, if there are no coins left on the table, the game ends.
  Let A and B be the sum of the values of the coins taken by Alice and Bob, respectively.

The score of the game is $A - B$. Alice wants to maximize the score; Bob wants to minimize it.

Alice moves first. What's the score of the game if Alice and Bob play optimally?

EXPLANATION:

The problem has this equivalent (and hopefully slightly easier) perspective:

We have a token that is initially placed at position 1. At each turn, each player does the following:

- Place a token onto some position greater or equals to the current position (possibly placing it at $N + 1$ which ends the game, or not moving the token).
- Take a coin from the pile at the current position of the token (if the pile is non-empty).

We have these two observations:

- Alice never wants to place the token at a position where C_i is even. This is because Bob can retaliate this play by simply not moving the token and take the coin at the same position (and doing it again and again until Alice can't take any coin from this position anymore).
- Similarly, when it's Bob's turn, he also doesn't want to place the token at a position where C_i is even.

From these two observations, we have the following two conclusions:

- We only care about piles with C_i odd in this game.
- It is always optimal for each player to move the token (since after the other player's previous move, the pile at the token's position has an even amount of coins).

From this, we arrive at a natural dynamic programming solution: let f_i be the maximal score the current player can achieve if the token is current at position i. Initially we have $f_{N+1} = 0$. There are two cases:

- The current player decides to move the token away from i first before choosing their coin. The answer in this case is f_{i+1}.
- The current player decides to pick a coin from the i-th pile, forcing the other player to move the token away. The answer in this case is $V_i - f_{i+1}$.
  Therefore $f_i = \max\{V_i - f_{i+1}, f_{i+1}\}$.

TIME COMPLEXITY:

Time complexity is $O(N)$ for each test case.

PREREQUISITES:

Catalan Number, Dynamic Programing, Difference Array, Longest Increasing subsequence

PROBLEM:

This problem has two subtasks worth 50 points each.

You are given an integer N. You also have N empty sets $S_1, S_2, \ldots, S_N$.

Consider all the arrays A of length N such that $0 \le A_i \le 1$ for each $1 \le i \le N$ (there are 2^N such arrays in total).

For each of these 2^N arrays, we compute an array B of length N where $B_i =$ length of longest non-decreasing subsequence for the prefix of length i of array A. (For example: if $N = 5$ and $A = [0, 1, 0, 1, 1]$, then $B = [1, 2, 2, 3, 4]$.) Next we add the array B to the set S_{B_N} (For example, $B = [1, 2, 2, 3, 4]$ would be added to S_4).

Output the size of sets $S_1, S_2, \ldots, S_N$ as N space-separated integers. Since the size of the sets might be large, output the values modulo $10^9 + 7$.

Note that the S_i are *sets*, and hence do not contain duplicate elements. Please look at the sample cases below for an explained example.

EXPLANATION:

Difference array: D_i of a given array A is defined as D_i = $A_i - A_{i-1}$ (for $0 < i < N$) and $D_0 = A_0$ considering 0 based indexing.
Observation 1: For any array B (The array formed by computing the length of longest non-decreasing subsequence for the prefix of length i of array A) its difference array consists of only 0 and 1 as the length of longest non decreasing subsequence cannot increase by more than 1 in any case.
Observation 2: For any array B, each B_i is at least $ceil(i/2)$. Simply take the maximum of count of zeroes or the count of ones in the prefix.
Observation 3: In the difference array formed by any array B, For any prefix length, the number of ones is greater than or equal to the number of zeroes in it.
Since each B_i is greater than $ceil(i/2)$ and B_i is same as number of ones in the prefix of length i of difference array. Let C_1 be the count of ones and C_0 be the count of zeroes in the prefix of length i of difference array.
$\therefore$ $C_1 \ge ceil(i/2)$ Let i be even then $C_1 \ge i/2 \implies C_1 >= (C_1 + C_2)/2 \implies (C_1 >= C_2)$. Similar analysis can be carried out when i is odd.
Observation 4: Every such difference array is valid. An example of the array which produces the same difference array is the difference array itself. Example Let difference array be $[1, 0, 1, 0]$ and let A be $[1, 0, 1, 0]$, Then array B is $[1, 1, 2, 2]$ difference array is $[1, 0, 1, 0]$. Each difference array corresponds to a different array B.
Thus we need to find difference arrays of size N which have total number of ones (same as B_N) in them as i for each $1 <= i <= N$

Dp solution
Let $dp[i][j]$ represent we are at index i of the array and there are j ones in the prefix of the array making sure there are no more zeroes than ones in the prefix.Initialize $dp[0][0]$ as 1.
Then either the element at i^{th} index is 1 or it is 0. Consider the dp states for both these cases as:
Then $dp[i][j] = dp[i-1][j-1] + (i - j \le j)?dp[i-1][j] : 0$
The answer to the problem is $dp[n][i]$ for all i from 1 to N.
But this is and $O(N^2)$ solution and not enough to pass all the test cases.

Combinatorics Solution

Let ${}^{N}C_{K}$ denote the usual binomial coefficient, i.e. number of ways to select K objects from set of N objects). Then number of such difference arrays having i ones in total comes out to be ${}^{N}C_{N-i} - {}^{N}C_{N-i-1}$ if $2 * i > N$ else 0.

Explanation: Let $f(K, N)$ be the number of $B - arrays$ of length N such that the last element is K. We want to find $f(1, N)$,$f(2, N)$, ...,$f(N, N)$. $f(K, N) = 0$ when $K < N/2$ or when $K > N$, and $f(0, 0) = 1$. We have the recurrence $f(K, N) = f(K - 1, N - 1) + f(K, N - 1)$. Think of this as lattice paths on the xy-plane. We start at $(0, 0)$ and reach (K, N) using two types of moves:

1. Move one step up
2. Move diagonally up-right

The only restriction is that we aren't allowed to cross the line $y = 2 \cdot x$. Now let's bring this to a more well-known form. Suppose that, instead of moving diagonally up-right, we only moved right. What happens then? If we would normally reach (K, N), now we would reach $(K, N - K)$ instead because the number of times we moved right stays the same, and the number of times we moved up decreases by K. What about the restriction that we don't cross $y = 2x$? We can see that in the new model this translates to the restriction that we don't cross the line $y = x$

Proof: Consider a path that crosses the line $y = 2 \cdot x$ in the first model. Let the first time this happens be at the point $(s, 2s + 1)$ To reach $(s, 2s + 1)$, we must have made $2s + 1$ moves of which s were diagonal and $s + 1$ were up. In the new model, this translates to s right moves and $s + 1$ up moves, hence reaching $(s, s + 1)$ Similarly, reaching $(s, s + 1)$ in the new model means reaching $(s, 2s + 1)$ in the old model. So, all we need to do is count the number of standard lattice paths from $(0, 0)$ to $(K, N - K)$ that don't cross the line $y = x$. This can be calculated by standard reflection principle stuff. The total number of paths without any constraints is just $C(K + N - K, N - K) = C(N, N - K)$ via reflecting bad paths about $y = x + 1$, We can derive that the total number of bad paths equals the total number of unconstrained paths to $(N - K - 1, K + 1)$, which is $C(N - K - 1 + K + 1, K + 1)$ = $C(N, K + 1)$. The answer is then their difference

We're looking for the number of arrays of $0s$ and $1s$ such that $count(1) \geq count(0)$ for every prefix. Starting at $(0, 0)$, consider adding a 1 to the array as moving right and adding a 0 as moving up. Then, for length N and K ones, you reach the point $(K, N - K)$ without crossing the line $y = x$
Please refer to this article for refection principle: **Reflection principle**

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

You have two counters a and b and a score S, all initially 0.
For N minutes, you need to increase a or b by 1.
There are also M events, each of one of two types:

- A type 1 event increases S by 1 if $a > b$, and sets $a = b = 0$ otherwise.
- A type 2 event increases S by 1 if $a < b$, and sets $a = b = 0$ otherwise.

What's the maximum possible final score you can achieve?

EXPLANATION:

Let $d = a - b$ denote the difference between a and b.
Note that each event checks whether the difference is positive or negative, then either increases the score by 1 or resets d to 0.

Also, in each minute, we can either increase or decrease d by 1.

Now, ideally we'd like to keep d somewhat close to 0 so we can quickly flip its sign if needed for an event.
In fact, it's enough to keep $-2 \le d \le 2$ at all times.

Proof

Consider some optimal sequence of moves that has $|d| > 2$ at some point. Without loss of generality, let's say d reaches 3. We can construct a new sequence of operations as follows:

Consider the first time we make the move $2 \to 3$.
Instead, let's make the move $2 \to 1$ in this step. Note that this essentially subtracts 2 from all later values of d.

Now,

- If the original sequence of moves never has the move $3 \to 2$ after this point, then we don't modify anything more.
  - The original sequence and new sequence both remain always positive after the change, and are the same before the change, so the new sequence has the same score as the original.
- If the original sequence of moves has the move $3 \to 2$ at some point, it corresponds to the move $1 \to 0$ in the new sequence.
  - Replace the first instance of such a move with the move $1 \to 2$ in the new sequence. Once again, it's not hard to see that the score doesn't change.

By continually repeating this process, we can ensure the entire sequence has $|d| \le 2$ at all times while still remaining optimal.

Armed with this knowledge, let's actually solve the problem.

We'll use dynamic programming: let $dp(i, x)$ denote the maximum score after i minutes if $d = x$.
Transitions are as follows:

- First, we need to either increase or decrease d by 1. That is, $dp(i, x) = \max(dp(i-1, x-1), dp(i-1, x+1))$.
- Then, we need to process an event at this time point, if there is one.
- Suppose there is a type 1 event. That changes the values as follows:

  - If $x > 0$, then add 1 to $dp(i, x)$
  - $dp(i, 0) = \max(dp(i, 0), dp(i, -1), dp(i, -2))$ since negative values are forced to 0
  - $dp(i, x) = -\infty$ for $x < 0$, since it's impossible to have a negative value of d after this minute.
- A type 2 event can be handled similarly.

The base cases for this function are $dp(0, 0) = 0$ and $dp(0, x) = -\infty$ for $x \neq 0$.
The final answer is the maximum value of $dp(N, x)$ for some x.

Since only $-2 \leq x \leq 2$ matters, we have $5N$ states and $\mathrm{O}(1)$ transitions for each, leading to a $\mathrm{O}(N)$ solution.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Dynamic programming on trees, **rerooting**

PROBLEM:

You're given a tree on N vertices whose edges are weighted.
The distance between vertices u and v is defined as follows:

- Let the edges on the (unique) $u \to v$ path be $w_1, w_2, \ldots, w_k$ in order.
- Then, $d(u, v) = f(w_1, 0) \oplus f(w_2, 1) \oplus f(w_3, 2) \oplus \ldots \oplus f(w_k, k-1)$
  Here, $f(x, b)$ refers to circular-shifting the integer x to the left, b times.
  All integers are considered to have 32 bits.

For each vertex v, compute $\sum_{i=1}^{N} f(v, i)$.

EXPLANATION:

Very often, problems like this one, which require you to calculate something for every vertex of a tree, can be solved by the technique known as *rerooting*.
The name comes from the fact that the problem is first solved for a single root, and then we keep enough information to be able to quickly recompute the answer while shifting the root from a vertex to its neighbor.

As such, rerooting is generally done in two parts:

- First, root the tree at an arbitrary vertex, say 1. Then, for each vertex u, compute the answer for the subtree of u, usually using a DFS.
- Once this is done, a second DFS allows us to reroot, and solve for all vertices that are *not* in the subtree of u. This takes care of all cases.

So, let's look at each of those parts individually for our problem.

Let's root the tree at vertex 1.
First for each u, we need to compute $\sum_v d(u, v)$ across all v that lie in the subtree of u.
That can be done with DFS, and storing relevant information corresponding to each bit.

Details

Let's perform our DFS, and suppose we're at vertex u. We want to find $\text{val}[u]$, the answer for u within its subtree.
Let c be a child of u, and w be the weight of the edge joining them. First, recursively solve for c to compute $\text{val}[c]$.

Now, let's consider all paths from u going into the subtree of c.
Each such path corresponds to a path from c going into its subtree.
Specifically, we can take any path starting at c and going down, circular rotate every edge weight on this path once (since they'll all be one step further from u than c), and then xor with w (the weight of the $u \to c$ edge).

However, it's not immediately obvious how to quickly do this for even a single path, let alone all paths.

Instead, let's look at what happens bit-by-bit.
Let $\text{ct}[u][b]$ be the number of paths from u into its subtree that have bit b set.
If we are able to calculate $\text{ct}[u][b]$ for each $0 \le b < 32$, then we'll simply have $\text{val}[u] = \sum_{b=0}^{31} \text{ct}[u][b] \cdot 2^b$.

Let's recursively compute $\text{ct}[c][b]$ for the child c of u.
Then, notice that bit b in a $c \to v$ path corresponds to bit $(b+1)$ (modulo 32) in the $u \to v$ path, since the

distance will increase by one.
In particular,

- If w has bit b *unset*, then $\text{ct}[u][b]$ increases by $\text{ct}[c][b-1]$.
- if w has bit b *set*, then $\text{ct}[u][b]$ increases by $\text{sz}[c] - \text{ct}[c][b-1]$, where $\text{sz}[c]$ denotes the subtree size of c.
  This is because we're XOR-ing with w, so any path with the bit set till c will have it unset when moving to u, and vice versa.

In this way, we can compute the contribution of child c to vertex u in $\text{O}(32)$.

This results in an algorithm in $\text{O}(32 \cdot N)$ in total.

Now that we've computed the subtree answer for each u, we need to reroot.

The rerooting process follows basically the same procedure.
When moving from u to its child c, we have the following:

- The rerooting procedure has already computed the path counts for each bit for paths starting at u that are outside it. We want to extend this to path counts for c.
- First, we need to account for all the children of u that are not c, since when moving to c, they'll be outside its subtree.
  This is not hard to do: after all, our initial DFS already explicitly computed the path counts from u to each of these other children, so all we need to do is take those values and remove the ones corresponding to child c, i.e, reverse the process for child c.
- Now, we know the path counts for everything from u. We want to shift this to path counts to everything outside c via edge weight w, which is exactly the same algorithm as we used when moving from child to parent, and hence is done in $\text{O}(32)$.

We DFS twice, each with a complexity of $\text{O}(32N)$, which is fast enough for the given constraints.

TIME COMPLEXITY

$\text{O}(32 \cdot N)$ per test case.

PREREQUISITES:

Depth first search, Dynamic programming, Trees

PROBLEM:

Arun has a rooted tree of N vertices rooted at vertex 1. Each vertex can either be coloured black or white.

Initially, the vertices are coloured $A_1, A_2, \ldots A_N$, where $A_i \in \{0, 1\}$ denotes the colour of the i-th vertex (here 0 represents white and 1 represents black). He wants to perform some operations to change the colouring of the vertices to $B_1, B_2, \ldots B_N$ respectively.

Arun can perform the following operation any number of times. In one operation, he can choose any subtree and either paint all its vertices white or all its vertices black.

Help Arun find the minimum number of operations required to change the colouring of the vertices to $B_1, B_2, \ldots B_N$ respectively.

EXPLANATION:

This problem can be solved using **dynamic programming** as it has optimal substructure and overlapping subproblems.
The tree is rooted at node 1.
Let $black_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u when the complete subtree of node u is painted black with an operation.
Let $white_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u when the complete subtree of node u is painted white with an operation.
Let $none_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u without doing any operation on u.
Initialize each of the value $black_u, white_u\ and\ none_u$ for all u where $1 \le u \le N$ with 0

Start the dfs at node 1. Let us suppose we are at some node u during the dfs traversal then for this node we have three values :
$none_u += ((A_u != B_u)?INF\ :\ min(1 + black_x, 1 + white_x, none_x)$ over all children x of node u
We can choose to colour any child's subtree either black, white or leave the child untouched whichever gives minimum number of operations we add it to $none_u$.
$black_u += ((!B_u)?1 + \sum white_x\ :\ \sum black_x)$ over all children x of node u
If in the end we need the colour of node u as white we need to first colour the whole subtree of u white otherwise we can just leave the node u untouched and calculate the answer for $black_x$ for all children x of node u and add them to $black_u$.
$white_u += ((B_u)?1 + \sum black_x\ :\ \sum white_x)$ over all children x of node u
If in the end we need the colour of node u as black we need to first colour the whole subtree of u black otherwise we can just leave the node u untouched and calculate the answer for $white_x$ for all children x of node u and add them to $white_u$.
The answer to the problem is $min(none_1, 1 + black_1, 1 + white_1)$;

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES

Math, Distribution, DP, Precomputation

PROBLEM

You have been given a weight of N kilos. The measure will be accurate only if 3 calibration weights are used. You need to put 3 weights such that the sum equals N **kilos**, given that the calibration weights are in multiples of K.

QUICK EXPLANATION

- We have to divide N into 3 multiples of k (ka, kb, kc) such that:

  $ka + kb + kc = N \implies a + b + c = \frac{N}{k}$

- The problem can be related to a common problem, distributing $\frac{N}{k}$ identical balls into 3 identical boxes such that no box should be empty.

- Let $f(N, m)$ be a function to distribute N identical balls into m identical boxes. In this particular problem:

  $f(N, 3) = f(N - 3, 1) + f(N - 3, 2) + f(N - 3, 3)$

  This recurrence relation is used to solve the problem.

EXPLANATION

In the following problem, we have to split N into 3 multiples of k. None of the multiples should be 0.

If N is not divisible by K then the answer will simply be 0. Otherwise, we need to distribute $\frac{N}{k}$ identical balls among 3 identical boxes, such that each should get at least one.

$f(N, m)$ is a function to divide N into m multiples or to put N balls into m boxes such that no box is empty.

We give 1 ball to each of the 3 identical boxes and now we are left with $N - 3$ balls to distribute without any restriction.

Now, we can either give all the $N - 3$ balls to 1 box, or we can give the $N - 3$ balls to 2 boxes, or we can give $N - 3$ balls to all the 3 boxes which is nothing but $f(N - 3, 3)$, and similar to our initial condition that we started with.

Thus, $f(N, 3)$ has 3 cases:

$f(N, 3) = f(N - 3, 1) + f(N - 3, 2) + f(N - 3, 3)$

Case 1

$N - 3$ is given to just 1 box.

This can be done only in 1 way.

$\implies f(N - 3, 1) = 1 for all N >= 3$

Case 2

$N-3$ is given to just 2 boxes.

For $N=3$:
$f(0,2)=0$

For $N=4$:
$f(1,2)=0$

For $N=5$:
$f(2,2)=1[1,1]$

For $N=6$:
$f(3,2)=1[2,1]$

For $N=7$:
$f(4,2)=2[3,1],[2,2]$

For $N=8$:
$f(5,2)=2[4,1],[3,2]$

For $N=9$:
$f(6,2)=3[5,1],[4,2],[3,3]$

For $N=10$:
$f(7,2)=3[6,1],[5,2],[4,3]$

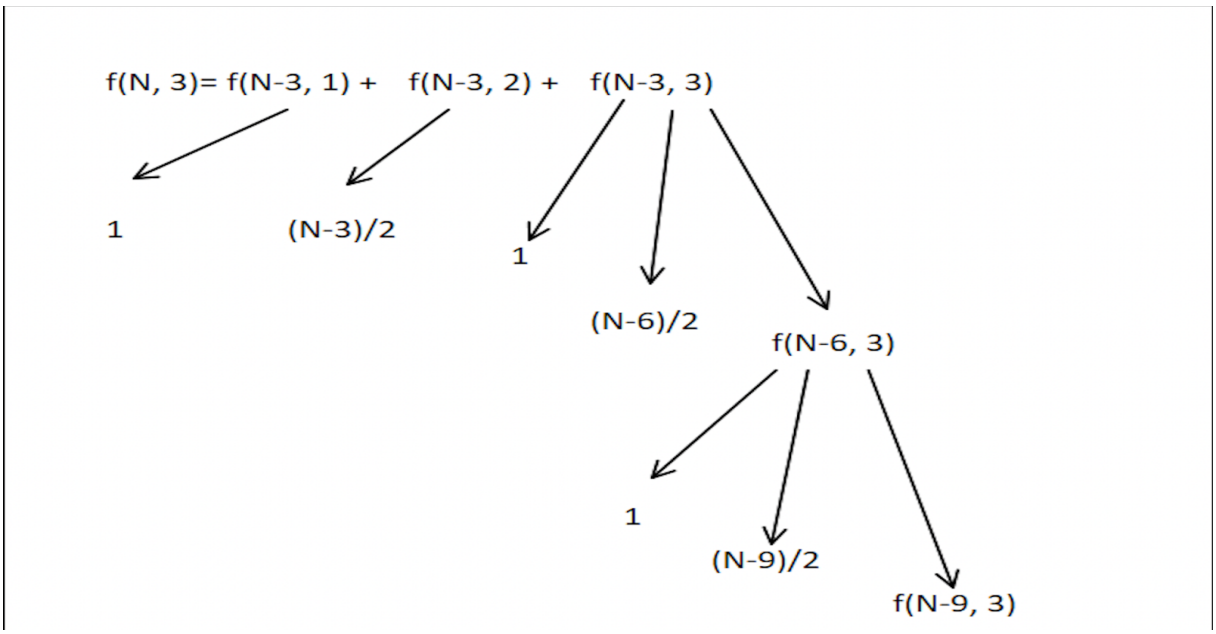
By recognizing the simple logic, or understanding the pattern we can say that this can be done in $\frac{N-3}{2}$ ways.

$\implies f(N-3,2)=\frac{N-3}{2}\ \forall N\geq 3$

Case 3 (Actual case)

$N-3$ is given to all 3 multiples.
This can further be broken into 3 cases.
$f(N-3,3)=f(N-6,1)+f(N-6,2)+f(N-6,3)$
This is the initial case that we started with.

$f(N,3)=1+\frac{N-3}{2}+f(N-3,3)$



Approach 1:

We can call the recurrence relation for each value of N. But in this approach we will have to calculate the value of $f(N-r,3)$ multiple times, where $0 < r < N$. This approach is acceptable for smaller values of N and K.

Time complexity:
$O(T \cdot N)$ where T is the number of test cases.

Approach 2:

We can use precomputation and store the final answer for each N in an array; the ith index of the array gives the answer for $N = i$
Time complexity:
$O(T+N)$ where T is the number of test cases.

PREREQUISITES:

(Optional) Dynamic programming

PROBLEM:

You have a permutation P of $\{1, 2, \ldots, N\}$. In one move, you can negate any of its elements.
A subarray is called *weird* if it can be sorted in ascending order by applying this move several times. Count the number of weird subarrays in P.

EXPLANATION:

In tasks like this which require you to count objects satisfying a certain property, it's useful to get a simpler criterion of that property.

Let's look at a weird array $A = [A_1, A_2, \ldots, A_k]$. What does this tell us about the elements of A?
One obvious fact is that in the final array, all the negative elements *must* come before all the positive elements — otherwise, there's no way for the final array to be sorted.
So, there is some index p such that $A_1, A_2, \ldots, A_p$ are negated, and $A_{p+1}, \ldots, A_k$ are not. Looking at each of those parts individually,

- We want $-A_1 < -A_2 < \ldots < -A_p$, i.e, $A_1 > A_2 > \ldots > A_p$
- We also want $A_{p+1} < A_{p+2} < \ldots < A_k$

In other words, A must look like a 'valley': first decreasing, then increasing (purely increasing/decreasing is also ok). It's also obvious that any such array is *weird*, since it can be sorted by negating the first part.

So, we just need to count the number of valleys in P. There are several ways to do this, a couple will be mentioned below:

- Let's call an index i a *hill* if $P_{i-1} < P_i > P_{i+1}$. Note that a subarray is a valley if and only if it doesn't contain a hill index along with both of its neighbors.
  - Suppose we find all hill indices. Then, we can simply count the number of weird indices between each adjacent pair of them to obtain our answer.
  - Counting the number of subarrays between two hills is simple combinatorics.
  - You may refer to the setter's code (below) for this approach.
- Another method is to directly count valleys.
  - Suppose we fix the deepest point of the valley to be index i. Let's count the number of valleys like this.
  - The left of i should be some decreasing sequence, and the right should be increasing.
  - Let $left_i$ denote the longest decreasing sequence ending at i, and $right_i$ denote the longest increasing sequence starting at i. These two can be found with dynamic programming in $\mathrm{O}(N)$.
  - Then, we add $left_i \times right_i$ to the answer, to account for all choices of subarrays with i as the deepest point (also including purely increasing/decreasing subarrays)
  - Adding up $left_i \times right_i$ across all $1 \le i \le N$ gives us the answer
  - You may refer to the editorialist's code (below) for this approach.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

You have two arrays A and B of length N. For each K from 0 to $N-1$, find the following:

- Let C be an array with $0 \le C_i \le B_i$
- At least K elements of C must be 0
- $sum(A) = sum(C)$
- Under these constraints, find the minimum value of $\sum_{i=1}^{N} |A_i - C_i|$.

EXPLANATION:

Let's fix a value of K and see what happens to C.

There are at most $N-K$ non-zero indices; let S denote the set of these non-zero indices. Then, we have the following:

- Let $B_S = \sum_{i \in S} B_i$. We must have $B_S \ge sum(A)$, otherwise it's impossible for $sum(C) = sum(A)$ to hold. For now, let's assume this condition holds.
- We can assign C using the following strategy:
  - If $i \notin S$, assign $C_i := 0$.
  - If $i \in S$ and $B_i \le A_i$, assign $C_i := B_i$. This is the best we can do at this index. Such indices contribute a value of $A_i - B_i$ to the result.
  - If $i \in S$ and $B_i > A_i$, assign $C_i := A_i$ initially.
  - Now, we need to add some values to indices of the third type to ensure $sum(C) = sum(A)$. Note that it doesn't really matter which index a value is added to: adding x to any such index increases the result by x.
  - Since $B_S \ge sum(A)$, it's always possible to achieve $sum(C) = sum(A)$, so we only need to find out how much is added — that will tell us the result.
- Finding out how much is added is simple: we have to add A_i for each $i \notin S$, and then $A_i - B_i$ for each $i \in S$ with $A_i \ge B_i$.

So, for the S we chose, the result can be computed as follows:

- Let B_S be as denoted above. Similarly, let $A_S = \sum_{i \in S} A_i$.
- Further, let $D = \sum_{i \in S} \max(0, A_i - B_i)$.
- Then,
  - The total contribution of indices $i \in S$ with $B_i \le A_i$ is D
  - The total contribution of indices $i \in S$ with $B_i > A_i$ is $sum(A) - A_S + D$
  - The total contribution of indices $i \notin S$ is $sum(A) - A_S$
- Adding all these together, the result for this set S is $2 \cdot (D + sum(A) - A_S)$

We would like to choose S in such a way that this is minimized. $sum(A)$ is a constant, so this is equivalent to minimizing $D - A_S$.
Note that a given index contributes $(\max(0, A_i - B_i) - A_i)$ to $D - A_S$ if it is chosen in S.

We also need to keep track of B_S, since we need to ensure it is greater than $sum(A)$.

This allows us to write the following knapsack-style dynamic programming solution:

- Let $dp_{i,k,x}$ denote the minimum possible value of $D - A_S$ for a set of size k chosen from indices $1, 2, \ldots, i$, whose sum of B_i values is x.
- At position i, we have two choices: either take index i into S, or don't
  - If i is not taken, we simply have $dp_{i-1,k,x}$
  - If i is taken, we have $dp_{i-1,k-1,x-B_i} + \max(0, A_i - B_i) - A_i$
  - $dp_{i,k,x}$ is the minimum of these two values.

This gives us a solution in $\mathrm{O}(N^4)$ with a pretty low constant factor (more specifically, it is $\mathrm{O}(N^2 \cdot sum(B))$, but $sum(B) \leq 10^4$ anyway).

However, it currently uses $\mathrm{O}(N^4)$ memory, which might be too much. Note that by iterating from left to right, when we are at i, we only care about the dp values at $i - 1$. This allows us to cut down the memory to $\mathrm{O}(N^3)$.

TIME COMPLEXITY

$\mathrm{O}(N^4)$ per test case

PREREQUISITES:

Dynamic programming, subarray sums, prefix sums, segment trees

PROBLEM:

There is a grid of $N \times N$ cells. The N blocks of each column are made up of soil, except for a contiguous sequence of cells: from the l_ith cell to the h_ith cell (starting from the bottom, 0-indexing). Cells can be cleared of soil.

We need to create a subgrid of length N and height H containing no soil. What is the minimum number of cells needed to be cleared of soil?

QUICK EXPLANATION:

There are only $N - H + 1$ possible $N \times H$ subgrids. The answer is the the minimum number of soil cells among all such subgrids. We can preprocess our grid so we can compute the sum of each subgrid in constant time.

To preprocess:

- We need to know the number of soil cells in each row (denoted by r_i for $0 \le i < N$). r_i is equal to the number of j s such that $0 \le j < N$ and $i < l_j$ or $i > h_j$. This is also equal to the following expression:

$$N - \#\{j : 0 \le j < N, i \le h_j\} + \#\{j : 0 \le j < N, i < l_j\}$$

Thus the r_is can be computed quickly by considering the l_js and h_js in sorted order.

- We also need to know the number of soil cells in each prefix of rows, i.e. let s_i be the sum $r_0 + r_1 + \cdots + r_{i-1}$.

Then the number of soil cells in each $N \times H$ subgrid can be computed as $s_i - s_{i-H}$ for $H \le i \le N$.

EXPLANATION:

Clearly, we need to find the $N \times H$ subgrid with the minimum number of soil cells in it. There are only $N - H + 1$ such subgrids: the following illustrates the case $N = 7$ and $H = 3$ (there are $N - H + 1 = 5$ subgrids):

```
.......          #######   .......   .......   .......   .......
.......          #######   #######   .......   .......   .......
.......          #######   #######   #######   .......   .......
....... -------> .......   #######   #######   #######   .......
.......          .......   .......   #######   #######   #######
.......          .......   .......   .......   #######   #######
.......          .......   .......   .......   .......   #######
```

We can simply construct the $N \times N$ grid, and compute the number of soil cells in these subgrids. Since there are $N - H + 1$ subgrids and each one has NH cells to check, this algorithm runs in $O((N - H + 1)NH)$ time. The worst case is when H is around half of N (which makes the running time $O(N^3)$), so unfortunately this algorithm is only good for the first subtask. For the second subtask, you can't even store the whole grid due to the memory requirements!

To answer the second subtask, we need a way to sum up these subgrids without constructing the whole grid. The first thing we notice is that the only information we need from its row is the number of soil cells in it, i.e. we don't need to know their positions in the row. Let's say the i th row ($0 \le i < N$) contains r_i soil cells. Then the number of soil cells in each of the $N - H + 1$ subgrids are the following:

- $r_0 + r_1 + r_2 + \cdots + r_{H-1}$
- $r_1 + r_2 + r_3 + \cdots + r_H$
- $r_2 + r_3 + r_4 + \cdots + r_{H+1}$
- $r_3 + r_4 + r_5 + \cdots + r_{H+2}$
- $\ldots$
- $r_{N-H} + r_{N-H+1} + r_{N-H+3} + \cdots + r_{N-1}$

The smallest of these is the answer! Thus, it would be very helpful if we can compute the sequence $r_0, r_1, \ldots, r_{N-1}$ quickly, without constructing the whole grid!

To do so, we use the following observation: r_i is equal to the number of j s such that $0 \le j < N$ and $i < l_j$ or $i > h_j$. Now, counting all such j s this way is still not fast enough, so we do some manipulations first. For a statement $\phi(j)$, let $C_{\phi(j)}$ be the number of j s such that $0 \le j < N$ and $\phi(j)$ is true. To familiarize yourself with this notation, we give a few basic facts (we invite you to verify each one):

- $C_{\text{true}} = N$
- $C_{\text{false}} = 0$
- $C_{\phi(j)} + C_{\text{not } \phi(j)} = C_{\text{true}} = N$
- $C_{\phi(j)} = C_{\text{true}} - C_{\text{not } \phi(j)} = N - C_{\text{not } \phi(j)}$
- $C_{\phi_1(j) \text{ or } \phi_2(j)} = C_{\phi_1(j)} + C_{\phi_2(j)} - C_{\phi_1(j) \text{ and } \phi_2(j)}$
- $C_{c<f(j)} + C_{c=f(j)} + C_{c>f(j)} = N$ (trichotomy)
- $C_{c<f(j)} + C_{c=f(j)} = C_{c\le f(j)}$
- If $f_1(j) \le f_2(j)$ for all j, then $C_{f_1(j)\le c\le f_2(j)} = C_{c\le f_2(j)} - C_{c<f_1(j)}$

Now, back to r_i. We have the following (using some of the facts above:

$$\begin{aligned} r_i &= C_{i<l_j \text{ or } i>h_j} \\ &= N - C_{\text{not } (i<l_j \text{ or } i>h_j)} \\ &= N - C_{i\ge l_j \text{ and } i\le h_j} \\ &= N - C_{l_j\le i\le h_j} \\ &= N - (C_{i\le h_j} - C_{i<l_j}) \end{aligned}$$

The last one is true because $l_j \le h_j$. Now, we have:

$$r_i = N - C_{i\le h_j} + C_{i<l_j}$$

To compute the r_is, we just need to compute the quantity $-C_{i\le h_j} + C_{i<l_j}$ for all $0 \le i < N$.

The key to this is to notice that r_i can be computed by a simple adjustment from r_{i-1}! In other words, we can just calculate the difference $r_i - r_{i-1}$, and if we have already computed r_{i-1}, then we can calculate r_i by adding this difference. In more detail, let's try to compute $r_i - r_{i-1}$:

$$\begin{aligned} r_i - r_{i-1} &= (N - C_{i\le h_j} + C_{i<l_j}) - (N - C_{i-1\le h_j} + C_{i-1<l_j}) \\ &= N - C_{i\le h_j} + C_{i\le l_j-1} - N + C_{i\le h_j+1} - C_{i\le l_j} \\ &= C_{i\le h_j+1} - C_{i\le h_j} + C_{i\le l_j-1} - C_{i\le l_j} \\ &= (C_{i\le h_j+1} - C_{i\le h_j}) + (C_{i\le l_j-1} - C_{i\le l_j}) \\ &= C_{i=h_j+1} - C_{i=l_j} \end{aligned}$$

But we can compute the values $C_{i=h_j+1}$ and $C_{i=l_j}$ for $0 \le i < N$ quickly, via a linear pass of all pairs (l_j, h_j) for $0 \le j < N$! The following pseudocode does it:

```
# arrays are initialized with zeroes
Ch[0...N]   # Ch[i] will contain C[i = h_j + 1]
Cl[0...N]   # Ch[i] will contain C[i = l_j]
for j = 0...N-1:
    Ch[h_j + 1] += 1
    Cl[l_j] += 1
```

Now that all the $C_{i=h_j+1}$ and $C_{i=l_j}$ are computed, we can now compute all the r_is using the following recurrence:

$$r_{-1} = N$$

$$r_i = r_{i-1} + C_{i=h_j+1} - C_{i=l_j}$$

The following pseudocode does it:

```
# array is initialized with zeroes
r[0...N]
curr = N
for i in 0...N-1:
    curr += Ch[i] - Cl[i]
    r[i] = curr
```

Clearly, these pseudocodes run in $O(N)$ time!

Finally, to compute the answer, we need to know the following sums:

- $r_0 + r_1 + r_2 + \cdots + r_{H-1}$
- $r_1 + r_2 + r_3 + \cdots + r_H$
- $r_2 + r_3 + r_4 + \cdots + r_{H+1}$
- $r_3 + r_4 + r_5 + \cdots + r_{H+2}$
- $\ldots$
- $r_{N-H} + r_{N-H+1} + r_{N-H+3} + \cdots + r_{N-1}$

and then compute the minimum among them. But this is easy! Notice that $r_i + r_{i+1} + \cdots + r_j$ is simply $(r_0 + \cdots + r_j) - (r_0 + \cdots + r_{i-1})$, so we can first try computing the **prefix sums**. Let s_i be the sum $r_0 + r_1 + \cdots + r_{i-1}$. Then $r_i + r_{i+1} + \cdots + r_j$ is simply $s_{j+1} - s_i$. The s_is can be computed in $O(N)$ too, because $s_i = s_{i-1} + r_{i-1}$, with the base case $s_0 = 0$. Afterwards, the sums we need are simply:

- $s_H - s_0$
- $s_{H+1} - s_1$
- $s_{H+2} - s_2$
- $\ldots$
- $s_N - s_{N-H}$

Since all the steps of this algorithm runs is $O(N)$, the answer can thus be computed in $O(N)$ time in total!

The following is a sample implementation in Python:

```
for cas in xrange(input()):
    n, h = map(int, raw_input().strip().split())
    row = [0]*(n+2)
```

```
for i in xrange(n):
    a, b = map(int, raw_input().strip().split())
    row[a+1] -= 1
    row[b+2] += 1
row[0] = n
for i in xrange(n): row[i+1] += row[i]
for i in xrange(n): row[i+1] += row[i]
print min(row[i] - row[i-h] for i in xrange(h,n+1))
```

A few things to notice about this implementation:

- The pairs (l_j, h_j) are never stored in an array: they are obtained from the input on the fly, processed, and thrown away immediately.
- Instead of having two arrays for $Ch[i]$ and $Cl[i]$, we only use a single array containing $Ch[i] - Cl[i]$.
- We reuse the same array `row` to contain the values $Ch[i] - Cl[i]$, r_i and s_i. Furthermore, r_i is stored in index $i + 1$ of `row`.

Time Complexity:

$O(N)$

PREREQUISITES:

Degree sequences, dynamic programming

PROBLEM:

Given a sequence $D_1, D_2, \ldots, D_N$, construct a connected simple graph on N vertices such that the degree of i is at most D_i and the XOR of all degrees is 0.

EXPLANATION:

Let's sort the sequence D in non-increasing order. Then, the degree sequence $d_1, d_2, \ldots, d_N$ is best made in non-increasing order too - any other valid sequence can be sorted by repeatedly swapping d_x and d_y ($x < y$, $d_x < d_y$) which keeps the sequence valid.

One necessary condition for a valid degree sequence is that $\sum d_i$ must be even, known as the *handshaking lemma*. As long as the XOR of all d_i has bit 0 equal to zero, this is immediately fulfilled.

Connectedness

The obvious necessary condition here is $d_i > 0$ for each i and $\sum d_i \geq 2(N-1)$. If we have a simple non-connected graph with a degree sequence satisfying this condition, we can make it connected while keeping degrees unchanged.

Since the graph is non-connected and contains at least $N-1$ edges, there must be a cycle in some connected component (CC) c_1. Let's remove one edge $u - v$ from this cycle, which doesn't break its CC. There must be an edge outside this CC since there are no isolated vertices; let's remove one such edge $w - s$, which might break a CC different from c_1 into two, denoted by c_2 and c_3. We add edges $a - w$ and $b - s$, which connects CCs c_2 and c_3 to c_1. The graph remains simple since $c_1 \neq c_2, c_3$.

If $c_2 = c_3$, the second added edge creates a cycle and we just connected two CCs into one. Otherwise we broke a CC and then connected three CCs into one. Either way, the number of CCs decreases by one and we can repeat the process until there's just one CC. In fact, we can use it in construction - find a not necessarily connected solution satisfying these inequalities and convert it in $O(NM)$ time to a connected solution.

Simpleness: Erdős-Gallai theorem

The necessary and sufficient condition for existence of a simple graph with our degree sequence $d_1, d_2, \ldots, d_N$ with even sum is that for each k ($1 \leq k \leq N$),

$$\sum_{i=1}^{k} d_i = L_k \leq R_k = k(k-1) + \sum_{i=k+1}^{N} \min(k, d_i)\,.$$

(If we were allowed to make parallel edges but no loops, $d_1 \leq \sum_{i=2}^{N} d_i$ would be the only condition. It's also immediately fulfilled by XOR zero since $a \oplus b \geq a - b$.)

When we move from k to $k+1$, the left side increases by d_{k+1}, the first term on the right side increases by $2k$ and the second term decreases at most by d_{k+1}. Once $d_{k+1} \leq k$ holds for the first time, it'll hold for any larger k and the difference $R_k - L_k$ will be non-decreasing, so we only need to check the conditions up to this k and can ignore greater values of k. This helps a lot in graphs that have very few high-degree vertices.

The condition for $k = 1$ simply says $d_1 \leq N - 1$, so we can ignore it. The condition for $k = 2$ is $d_1 + d_2 \leq 2N - 2 - n_1$ if $n_1 \leq N - 2$. For $k = 3$ and $n_1 + n_2 \leq N - 3$, the condition's $d_1 + d_2 + d_3 \leq 3N - 3 - 2n_1 - n_2$. With the observation above, it'll turn out that we won't need to directly evaluate the conditions for any higher k, but we only need the guarantee that when they're all fulfilled, a solution exists.

Simpleness + construction: Havel–Hakimi algorithm

In short, we construct a simple graph with degree sequence $d_1, d_2, \ldots, d_N$ from a graph with $N-1$ vertices and degree sequence $d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_N$ ($d_1 \le N-1$) by connecting vertex 1 to vertices $2, \ldots, d_1 + 1$. The only catch is that the smaller degree sequence needs to be sorted in non-increasing order again, but if the original sequence was sorted, then re-sorting is easy. This construction takes $O(N^2)$ time.

Looking for a sparse graph

If we have a valid degree sequence and subtract the same number s from some two degrees d_i and d_j, such that the sequence doesn't need to get reordered, then Erdos-Gallai conditions for $k \ge \min(i,j)$ will still hold, because L_k decreases by s and R_k decreases by at most s. Specifically, for $i = 1$, we obtain a valid graph.

Intuition should tell us that we want to make the resulting graph quite sparse, especially by decreasing large degrees. The only lower limit comes from connectedness, otherwise we can even see that when the sum of degrees is $O(N)$, then Erdos-Gallai conditions for $k \gg O(\sqrt{N})$ are automatically fulfilled.

Consider any (not necessarily valid) degree sequence with XOR zero. To avoid reordering the sequence, let's try subtracting 1 from some elements greater than 1; if there are multiple elements with the same value, we subtract from the last one. Can we do it while keeping the zero XOR?

- If the sum of degrees is $\le 2(N-2)$, then we stop. In particular, this includes $d_2 = 1$, so $d_2 \ge 2$ from now on.
- If there are at least two odd degrees greater than 1, we subtract 1 from each.
- If there are only even degrees greater than 1, we find the lowest set bit (LSB) in each of them. Consider the lowest value of this LSB. Since the XOR was zero, there must be an even number of degrees with this lowest LSB; degrees 1 don't change anything since it's LSB of even numbers, and all degrees with greater LSB have that bit set to 0. Therefore, we can find two degrees with bit $b > 0$ as the LSB and subtract 1 from each. This doesn't change the XOR.
- Finally, if there's exactly one odd degree greater than 1, we can subtract 1 from that (changing XOR to 1), do the same thing as above, which creates two odd numbers, and subtract 1 from either of them to fix the XOR.

We're always only subtracting 2 or 4. That's great news - from an even sum of degrees $\ge 2(N-2)$, we can always reach exactly $2(N-2)$ or $2N$, all while keeping the conditions for XOR and connectedness! What about Erdos-Gallai conditions?

- Let's assume that a condition for some k is violated. Then $L_k > k(k-1) + N - k$ and at the same time from sum of degrees, $L_k \le 2N - (N-k)$, so $N + k > k^2 - 2k + N$ must hold. This is obviously impossible for $k \ge 3$, we already know that $k = 1$ is never violated and for $k = 2$, it's only possible if $L_k = 2N - (N-k)$, which means sum of degrees $2N$ with $d_3 = 1$.
- In addition, if d_1 and d_2 are even, we can still subtract 2. Since XOR is zero, we only need to deal with the case $d_1 = 2a + 1$, $d_2 = 2a$ and $2N = (2a+1) + (2a) + (N-2)$, so $a = (N+1)/4$.
- If $D_4 \ge 2$, we can change d_3 and d_4 to 2; this doesn't change XOR, fixes the condition for $k = 2$, and since $d_3 \le 2$, we already know that conditions for greater k will also hold.

Note that when we were subtracting 4, this wouldn't be allowed if we were changing $(3,2) \to (2,2) \to (1,1) \to (1,0)$, but in such a case, there wouldn't be any other degrees ≥ 2 (their LSB would have to be ≥ 2 so we'd be able to find two with the same LSB) and there are no such sequences with XOR zero and sum $> 2N$.

The special case

The only case we haven't solved is $D_4 = 1$. However, there are only $O(N^2)$ possible degree sequences there, since D_3 uniquely determined by D_1 and D_2, and we don't need to check conditions for $k \ge 4$ if those for $k = 2$ and $k = 3$ hold. We can deal with it using brute force - find one valid degree sequence or determine that none exist.

Initial sequence

In all other cases, we described a way to find a valid degree sequence. We just need to find one that satisfies the conditions for connectedness and XOR. (If there are no such sequences, there's obviously no solution.)

Let's employ dynamic programming. If we picked $d_1, \ldots, d_i$ and got XOR x, the largest possible sum is $dp_{i,x}$. We try all values $d_{i+1} \le D_{i+1}$, all values of x and update $dp_{i+1,\cdot}$.

Since the largest bit in x is (at most) the largest bit in $N-1$, we can't have $x \ge 2(N-1)$. Still, the time complexity is $O(N^2 + N\sum D_i) = O(N^3)$, so we need to speed it up when $\sum D_i \gg N$.

- In such a case, let's start by using only the highest set bit from each D_i in d_i.
- Then for each bit $b > 0$ from highest to lowest, if there's an odd number of degrees equal to 2^b, we change one of them to 2^{b-1}. Once we're done, all bits except bit 0 occur an even number of times, so the XOR is 0 or 1.
- If the XOR is 1 and there's an even degree $d_i < D_i$, we can add 1 to d_i.
- If $D_1 = d_1$, then $D_1 = 2^b$. Otherwise $d_1 = 1$ since all degrees created so far were powers of 2.
- With $D_1 = d_1 = 2^b$ and $b > 1$, we have $d_2 = d_1$ because of XOR, and we can decrease the last two degrees equal to 2^b by 1 and 2 respectively. This fixes the XOR from 1 to 0.
- The only failure cases we have here are with $d_1 \le 2$, so $\sum d_i \le 2N$.

In the first step, we have $2d_i > D_i$, so $\sum d_i \ge (N + \sum D_i)/2$. In the second step, for each b such that $1 < 2^b < N$, we subtract at most 2^{b-1} from $\sum d_i$, which is at most $N-2$ in total. Finally we subtract at most 3. We end up with $\sum d_i \ge (\sum D_i - N - 2)/2$. If $\sum D_i \ge 5N + 4$, then $\sum d_i > 2N$, which means we don't reach either failure case and found a sequence we can use as our initial sequence.

If $\sum D_i < 5N + 4$, we just use the DP approach, which is now fast enough.

Decreasing degrees quickly

Getting a degree sequence with sum $\le 2N$ has been described above, but the straightforward approach decreases the sum by 2 or 4 in $O(N)$ time. We could customise the lookup for large $\sum D_i$ (which was already quite custom) to produce a degree sequence with reasonably small sum, but we can also implement it efficiently in general.

We're always picking two degrees > 1 based on their LSB. Let's group those elements based on LSB - there are $O(\log N)$ groups. We always pick two elements with the same LSB and change them, or change an odd element. This way, we decrease the sum of degrees by 2 or 4 in $O(\log N)$ time, just because we need to look up an LSB with two elements. New LSB can be precalculated $O(N \log N)$.

The next speedup comes from the observation that we don't need to always subtract 1 but if the LSB of any two elements is the b-th bit, then subtracting anything $\le 2^b$ still keeps XOR unchanged; we only used the lowest LSB to find two elements. We get three possible situations if we find two such elements:

- we can't subtract exactly 2^b because the sum of degrees would drop below $2(N-1)$; in this case, we just subtract what we can and we're done
- we can't subtract exactly 2^b because it'd create degrees 0, so we subtract $2^b - 1$, creating two elements equal to 1
- we subtract exactly 2^b from both chosen elements; this just sets their b-th bits to zero

The first situation can only happen once, the second only $O(N)$ times and the last one decreases the sum by 2^{b+1} after $O(b)$ LSB lookup. Since $O(2^b) \ge O(b)$, the LSB lookup can only take $O(\sum D_i)$ amortised time. Once there are no two elements > 1 with the same LSB, there are $O(\log N)$ elements > 1 so $\sum d_i$ is $O(N \log N)$ and the rest of the subtracting always takes at most $O(N \log^2 N)$ time.

We didn't bother with keeping the degree sequence non-increasing here, but since we already know that this'll give a valid sequence, and unlike Erdos-Gallai theorem or graph construction, it doesn't directly rely on

sortedness, we can just sort the resulting valid sequence.

Summary

1. If $D_4 = 1$, use brute force to find a degree sequence for a valid solution - or report that there's no solution.
2. Otherwise, find a degree sequence with positive elements, XOR zero and sum $\geq 2(N-2)$ - or report that there's no solution. Then convert it into a degree sequence for a valid solution by subtracting from degrees.
3. Construct a graph with this valid degree sequence.
4. Make this graph connected.

TIME COMPLEXITY:

Since we always construct a sparse graph, each of the key steps can be done in $O(N^2)$.

PREREQUISITES:

Digit DP

PROBLEM:

The rating of a number is defined to be the number of trailing zeros in the product of its digits (and 1 if this product is 0).

Given two large numbers A and B, find the sum of the ratings of all numbers from $A+1$ to B.

EXPLANATION:

A and B are rather large numbers: they can have upto 1000 digits each, which definitely doesn't allow for them to be read as integers in C++.

A lot of the time, when dealing with large numbers and you need to calculate some function for all numbers $\leq n$, the solution tends to be digit DP. You can read an introduction to the technique **here** . Some familiarity with the technique will be assumed below.

In our case, let's define the function $f(N)$ to be the sum of the ratings of all numbers from 1 to N. Then, our answer is $f(B)-f(A)$, so we just need to be compute $f(N)$ at a given argument N fast enough.

Note that the rating of a number depends only on two things:

- The number of twos and fives present as factors of its digits.
- Whether the number contains 0 or not. If it does, the rating is always going to be 1.

In particular, for a number that doesn't contain a zero, its rating is simply the minimum of the number of factors of 2 and 5 present among its digits.
For now, let's consider only numbers that don't contain 0 — the ones that do can be taken care of separately later, and it isn't hard to do so.

There's a fairly simple function that comes to mind immediately:
(1-based indexing is used below)
Let $g(i,x,y)$ denote the answer if we are going to place the i-th digit of the number, and there are x powers of 2 and y powers of 5 so far.
Transitions are simple: iterate over which digit d from 1 to 9 is going to be placed here while ensuring that you don't exceed N (recall that we are ignoring 0), calculate the new values of x and y from d (call them x' and y' respectively), and then recurse to $g(i+1,x',y')$ and add it to the answer.
The base case is of course $g(L+1,x,y)=\min(x,y)$ where L is the length of the string N, and the final answer is $g(1,0,0)$.

By memoizing the values of $g(i,x,y)$ (with perhaps extra boolean states to denote whether the current inequality with N is tight or not, and whether the current prefix is non-zero), this solution runs in $\mathrm{O}(L^3\cdot 10)$.
However, $L=10^3$ for us, so this is too slow.

To speed it up, we use a somewhat common DP optimization: reduce the number of states using existing information.
In particular, it is unnecessary to know both x and y — we only really need to know whether there are more 2-s or more 5-s, and how many more there are. This information can be kept with a boolean and an integer, so we've reduced the number of states by a factor of L.

However, making this change also necessitates a change in the definition of our g, since the parameters are different now.

It's somewhat hard to explain properly with words, and I recommend looking at the setter's code linked below for what to do. However, the gist is as follows:

- $g(state)$ now stores a pair: the answer for that state, and the number of ways to reach that state. Let's denote these by $ans(state)$ and $ways(state)$ respectively.
- If the current state has more 2-s than 5-s, and the current digit adds x more 2-s (where $x > 0$), nothing much needs to be done: the difference between the number of twos and fives grows by x.
- A similar condition applies to the current state having more 5-s and the current digit being 5
- Now, suppose the state has more 2-s and we add a 5. Then,
  - The difference decreases by 1 (unless it was already zero)
  - Let $state_2$ be the state we recurse into. Then, we need to further add $ways(state_2)$ to $ans(state)$ to account for this extra five.
- Something similar is to be done for the case when currently there are more 5-s and the current digit contains a 2. In particular, $\{2, 6\}, \{4\}, \{8\}$ all need to be treated as separate cases since they add $1/2/3$ twos respectively.
- $ways(state)$ is the sum of $ways(state_2)$ across all states that are recursed into.

This brings us down to something like $\mathrm{O}(L^2 \cdot 8)$ states, and 10 transitions from each, which is fast enough for $L = 1000$.

Now, all that remains is to count the contribution of numbers that contain a zero.
Note that this is essentially the same as counting the numbers $\leq N$ that contain a zero, which is not very hard to do.
For example, you can do the following:

- For each i from 1 to L, suppose the first time you go below N is at position i. So, the first $i - 1$ digits match those of N, and the last $L - i$ digits are completely free.
- Suppose you place digit d at this position.
  - If $d = 0$ or the first $i - 1$ digits contain a zero, the remaining digits can be absolutely anything. So, add 10^{L-i} to the answer
  - Otherwise, you need to count the number of strings of length $L - i$ that contain a zero. This can be precomputed with another dp.

Special care must be taken when placing $d = 0$ at $i = 1$, since the above discussion only applies when the number has a non-zero prefix.
However, placing 0 at $i = 1$ is the same as counting all numbers of length $\leq N - 1$ that contain a zero, which can again be precomputed.

Note that this is not the only way to solve the problem: I noticed that many contestants (and even the setters) have many different solutions, some perhaps easier than the one outlined here.
If you would like to share your solution in the comments below, feel free to do so!

TIME COMPLEXITY

$\mathrm{O}(L^2 \cdot 10)$ per test case, with a constant factor of around 8.

PREREQUISITES:

Longest Increasing Subsequence, Linear Recurrences, Berlekamp-Massey Algorithm

PROBLEM:

For any array, A, define $f(A)$ to be equal to the length of the longest strictly increasing subsequence of A. Given N, M and P, compute the sum of $f(A)^P$ over all arrays A of length N taking values in $\{1, 2 \ldots M\}$ modulo 754974721.

EXPLANATION:

Given an array A of length N, we can compute the length of the longest strictly increasing subsequence in $\mathrm{O}(N \log(N))$. See **here** for reference. Consider the vector that we maintain in this algorithm. Since we are considering strictly increasing subsequence, the values in this vector will always be distinct. Also, the length of LIS is equal to the length of this vector at the end of the algorithm. So we can use the following algorithm to compute LIS (which is just a slight modification in the implementation) from **here** .

```
set<int> S;
for(int i = 0; i < N; i++){
    auto it = S.lower_bound(A[i]);
    if(it != S.end())S.erase(it);
    S.insert(A[i]);
}
cout << S.size() << '\n';
```

Notice that at each iteration, all that matters is the elements that are present in the set S. And since we have the constraint $M \leq 14$, we can use a bitmask to represent the set S.

Now create a graph with nodes being all possible sets S i.e. all 2^M subsets of $\{1, 2 \ldots M\}$. For any bitmask u and for any $1 \leq j \leq M$, we can find the bitmask v that we get if the current set is u and the next element in the sequence j. Add an edge from u to v with label j. We get a directed graph with 2^M vertices and $M \times 2^M$ directed edges.

Clearly, there is a bijection between the set of all sequences of length N taking values in $\{1, 2 \ldots M\}$ and the set of all paths of length N starting from node 0 in our graph. Also, for any array, A, the final node in the corresponding path is equal to the set that we get by running the above algorithm on A and $f(A)$ is equal to the number of bits set in the final node of the corresponding path. So the problem boils down to computing the number of paths of length N from node 0 to node u and then multiplying this with $(\text{number of bits set in } u)^P$ and then summing this up over the sets (bitmasks) u i.e.

$$\sum_{u=0}^{2^M-1} (\text{Number of paths from } 0 \text{ to } u) \times (\text{Number of bits set in } u)^P$$

Let A be the adjacency matrix of the graph we constructed. Clearly, the number of paths of length N from node i to node j is equal to the (i,j)-th entry in the matrix A^N. So, the number of paths from node 0 to node i is equal to i-th entry in the column vector $A^N e_0$ where e_0 is the column vector with 1 at 0-th index and 0 everywhere else. Now define the column vector c of length 2^M with c_u being equal to $(\text{number of bits set in } u)^P$. Then, the required answer is equal to $c^T A^N e_0$, where c^T is the transpose of the vector c.

Since A is a square matrix of size 2^M, it will exceed the time limit if we use matrix exponentiation to compute $c^T A^N e_0$.

Let $p_n = c^T A^N e_0$. From the Cayley-Hamiltonian theorem, the matrix A satisfies its characteristic polynomial i.e. if p is the characteristic polynomial of A, then $p(A) = 0$. And we have that the degree of polynomial p is equal to 2^M. Suppose

$$p(x) = a_0 + a_1 x + \ldots + a_L x^L$$

where $L = 2^M$. Since $p(A) = 0$, it implies that

$$a_L A^L + \ldots + a_1 A + a_0 I = 0$$

So, for any $N \geq L$, we have

$$a_L A^N + \ldots + a_1 A^{N-L+1} + a_0 A^{N-L} = 0$$

This implies that for any $N \geq L$, we have

$$a_L c^T A^N e_0 + \ldots + a_1 c^T A^{N-L+1} e_0 + a_0 c^T A^{N-L} e_0 = 0$$

i.e. $a_L d_N + \ldots + a_1 d_{N-L+1} + a_0 d_{N-L} = 0$. So, the sequence $\{d_N\}$ satisfies a linear recurrence relation. It's very difficult to come up with an explicit theoretical upper bound on the degree of linear recurrence corresponding to this sequence, but it can be very easily verified using the Berlekamp-Massey algorithm that for any fixed M and P, the degree of the linear recurrence is less than 500 due to the special structure of this graph.

So we compute the values of d_N for the first $2 \cdot 500 = 1000$ values of N and then use the Berlekamp-Massey algorithm to come up with coefficients of the linear recurrence relation. Then, answering a query takes $\mathrm{O}(d)$ run time where d is the degree of the linear recurrence which is less than 500. The value of d_N for the first 1000 values can be computed using dynamic programming. Define $dp[u][k]$ to be the number of paths from 0 to u of length k. Then

$$dp[v][k+1] = \sum_{(u,v) \in E} dp[u][k]$$

And since there are exactly $M \times 2^M$ edges, this transition step has a time complexity of $\mathrm{O}(M \times 2^M)$.

Let $A = 500$. So, given M and P, we precompute d_n for $n \leq 2A$ in $\mathrm{O}(A \cdot M \cdot 2^M)$. And then use the Berlekamp-Mersey algorithm to compute the coefficients of linear recurrence in $\mathrm{O}(A^2)$. Now for a given N, we can compute d_N using the algorithm described **here** in $\mathrm{O}(A^2 \log(N))$ or $\mathrm{O}(A \log(A) \log(N))$.

TIME COMPLEXITY:

$\mathrm{O}(A \cdot M \cdot 2^M + A^2 + Q \cdot A \log(A) \log(N))$

Problem statement :

There are some flat areas and some slopes connecting two flat areas . A slope area has some length d and some maximum speed s in which we can go in that slope with that maximum speed. Going on the slope takes some energy as defined below :

If s <= 50 → energy e = d*(60 - s)

If s > 50 → energy e = d*(s - 40)

Everytime you use some slope to get to the other flat area , you add the energy required to go in that slope to your total energy. You are in flat 1 and you want to reach flat area n , find the minimum total energy from which we can reach flat n

Idea :

First for each slope , we find the minimum energy required if we travel on this slope and then once we do this for all the slopes , as the graph is topologically sorted , we do DP to find the minimum energy required to reach each flat area and then we would find our answer 🙂

Subtask A :

Number of flat areas : 7

Number of slopes : 9

Slopes :

No. →	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Fro →	1	1	1	2	3	3	4	5	6
To →	2	3	4	5	5	6	6	7	7
s →	60	40	20	40	60	30	20	30	30
d →	15	10	5	20	30	10	5	10	10
e →									

NOTE : Here s is maximum speed , so we can go in any speed between 1 and s both inclusive

Let us denote E(x,y) as the energy required to use the slope which connects flat areas x and y

E(1,2) :

Here , s > 50 , therefore energy required is (s - 40)*d* = *20*15 = 300 if we go in 60 speed

Now what if we go at 51 speed ?

Here also , s > 50 , therefore energy required is (s - 40)*d* = *11*15 = 165 if we go in 51 speed

But what if we go at 50 speed ?

Here , s <= 50 , therefore energy required is (60 - s)*d* = *10*15 = 150 if we go in 50 speed which is lesser than both of the above

What if we go at a speed less than 50 ?

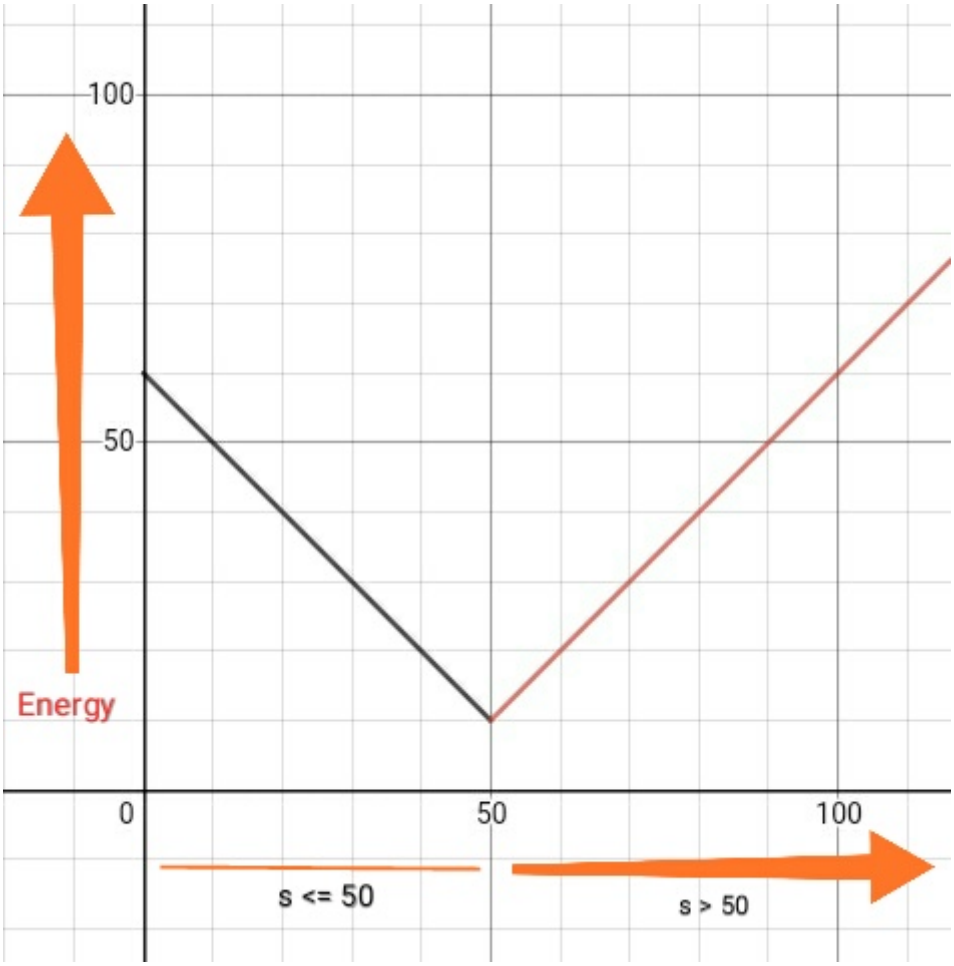
Clearly , 60 - s would increase , hence increasing the energy required

Therefore, the formula for finding the minimum energy required is

Minimum energy = d * (60 - min(50 , s))

Here we are doing min (50 , s) because if s > 50 , then it takes as 50 making the energy minimum

Let us plot it in graph and see it graphically that taking 50 if s > 50 is better :



Here we see that , taking s = 50 (when s > 50) is better than taking s > 50 or taking s < 50

So now , let us calculate the energy required for all the slopes :

E(1, 2) = d * (60 - min(50 , s)) = 15*10 = 150

E(1, 3) = d * (60 - min(50 , s)) = 10*20 = 200

E(1, 4) = d * (60 - min(50 , s)) = 5*40 = 200

E(2, 5) = d * (60 - min(50 , s)) = 20*20 = 400

E(3, 5) = d * (60 - min(50 , s)) = 30*10 = 300

E(3, 6) = d * (60 - min(50 , s)) = 10*30 = 300

E(4, 6) = d * (60 - min(50 , s)) = 5*40 = 200

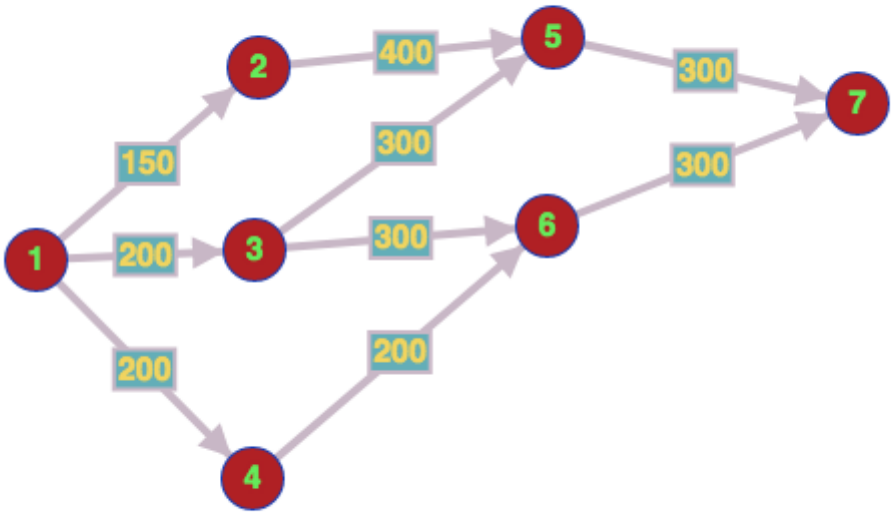
E(5, 7) = d * (60 - min(50 , s)) = 10*30 = 300

E(6, 7) = d * (60 - min(50 , s)) = 10*30 = 300

The table would look like this :

No. →	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
From →	1	1	1	2	3	3	4	5	6
To →	2	3	4	5	5	6	6	7	7
s →	60	40	20	40	60	30	20	30	30
d →	15	10	5	20	30	10	5	10	10
e →	150	200	200	400	300	300	200	300	300

The graph would look something like this :



Let us do DP now to find our answer :

Before doing DP , we have to know if dp works here ,

The answer is 'Yes' as the graph is **topologically sorted** and there are no cycles (the graph is **DAG** i.e Directed Acyclic Graph)

So now , after making sure that DP works , let us see how to do DP :

Dp[i] = minimum energy required to reach the i'th flat area

Base case : dp[1] = 0 because we start at the first flat area and reaching the first flat area requires no energy , therefore this is the base case

Relation : dp[i] = min (dp[j] + E (i,j)) for all j from which we can come to i'th flat area using some slope

Answer : dp[n] because we have to find the minimum energy required to reach the n'th flat area

So now , let us start filling the dp table :

Base case : dp[1] = 0

Relation :

Dp[2] = dp [1] + E(1,2) = 0 + 150 = 150 (as we can only come from 1)

Dp[3] = dp [1] + E(1,3) = 0 + 200 = 200 (as we can only come from 1)

Dp[4] = dp [1] + E(1,4) = 0 + 200 = 200 (as we can only come from 1)

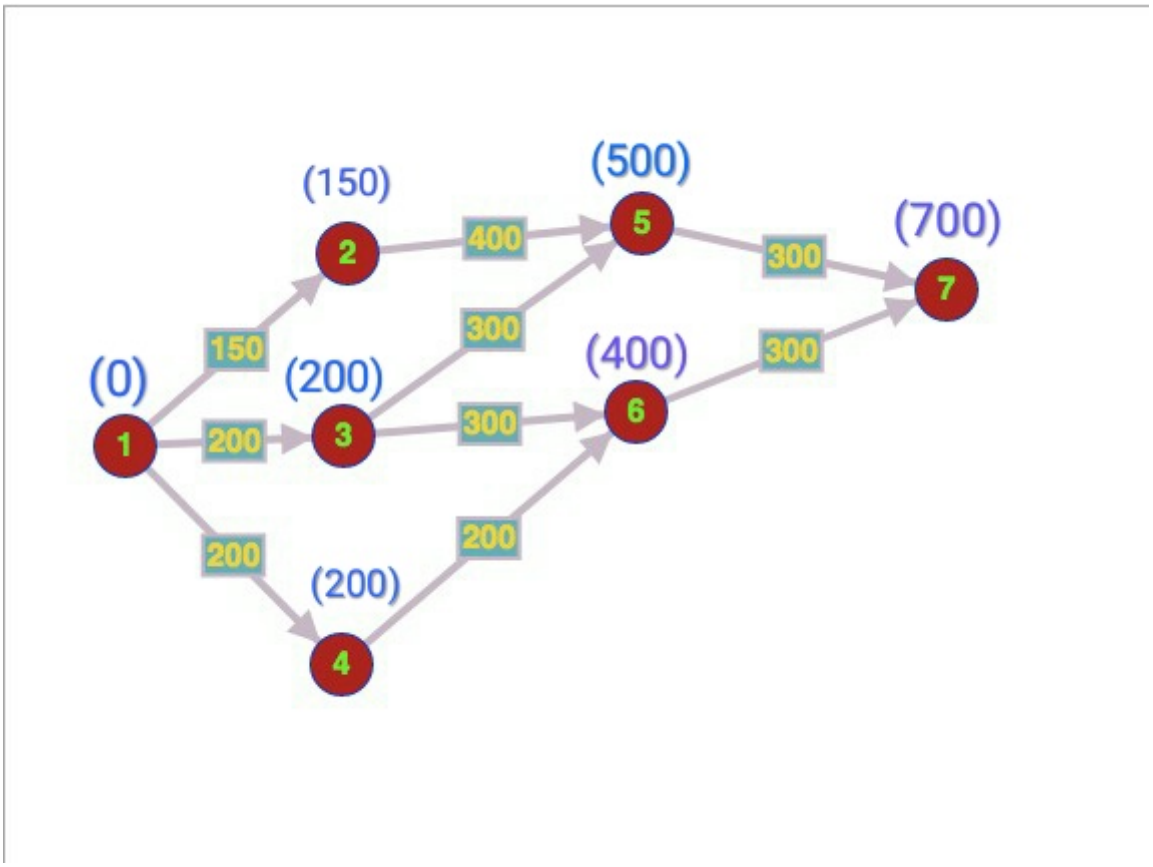
Dp[5] = min (dp[2] + 400 , dp[3] + 300) = min (700 , 800) = 700

Dp[6] = min (dp[3] + 300 , dp[4] + 200) = min (500 , 400) = 400

Dp[6] = min (dp[5] + 300 , dp[6] + 300) = min (1000 , 700) = 700

Answer : dp[7] = 700

Dp values for all the points :



Answer : 700

Subtask B :

Number of flat areas : 9

Number of slopes : 12

Slopes :

Now , let us calculate the energy for all the slopes :

E(1, 2) = d * (60 - min(50 , s)) = 15*10 = 150

E(1, 3) = d * (60 - min(50 , s)) = 25*20 = 500

E(2, 4) = d * (60 - min(50 , s)) = 30*20 = 600

E(2, 5) = d * (60 - min(50 , s)) = 25*10 = 250

E(3, 5) = d * (60 - min(50 , s)) = 20*10 = 200

E(3, 6) = d * (60 - min(50 , s)) = 10*20 = 200

E(4, 7) = d * (60 - min(50 , s)) = 10*30 = 300

E(5, 7) = d * (60 - min(50 , s)) = 10*10 = 100

E(5, 8) = d * (60 - min(50 , s)) = 5*20 = 100

E(6, 8) = d * (60 - min(50 , s)) = 20*20 = 400

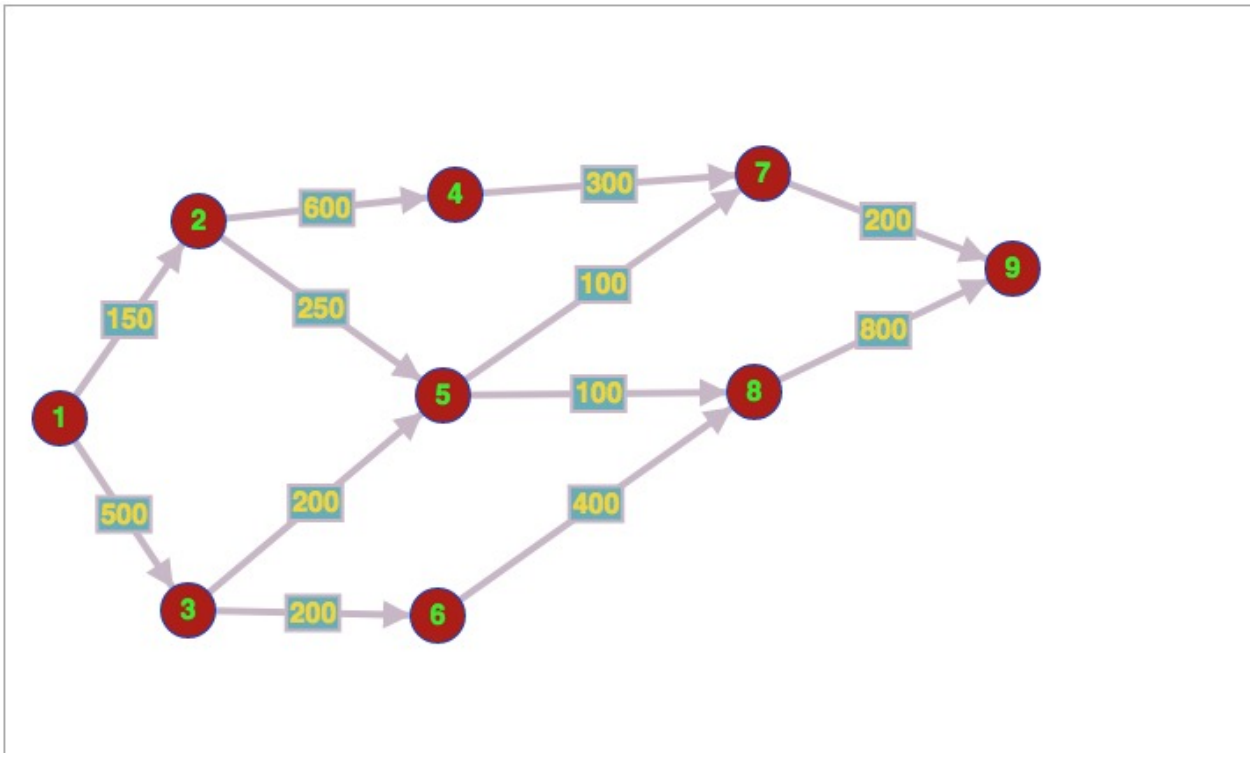
E(7, 9) = d * (60 - min(50 , s)) = 10*20 = 200

E(8, 9) = d * (60 - min(50 , s)) = 20*40 = 800

The table would look like :

No. →	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
From →	1	1	2	2	3	3	4	5	5	6	7	8
To →	2	3	4	5	5	6	7	7	8	8	9	9
s →	50	40	40	50	50	40	30	60	40	40	40	20
d →	15	25	30	25	20	10	10	10	5	20	10	20
E →	150	500	600	250	200	200	300	100	100	400	200	800

The graph would look something like this :



So now , let us start filling the dp table :

Base case : dp[1] = 0

Relation :

Dp[2] = dp[1] + 150 = 150 (as we can only come from 1)

DP[3] = dp[1] + 500 = 500 (as we can only come from 1)

Dp[4] = dp[2] + 600 = 750 (as we can only come from 2)

Dp[5] = min (dp[2] + 250 , dp[3] + 200) = 400

Dp[6] = dp[3] + 200 = 700 (as we can only come from 3)

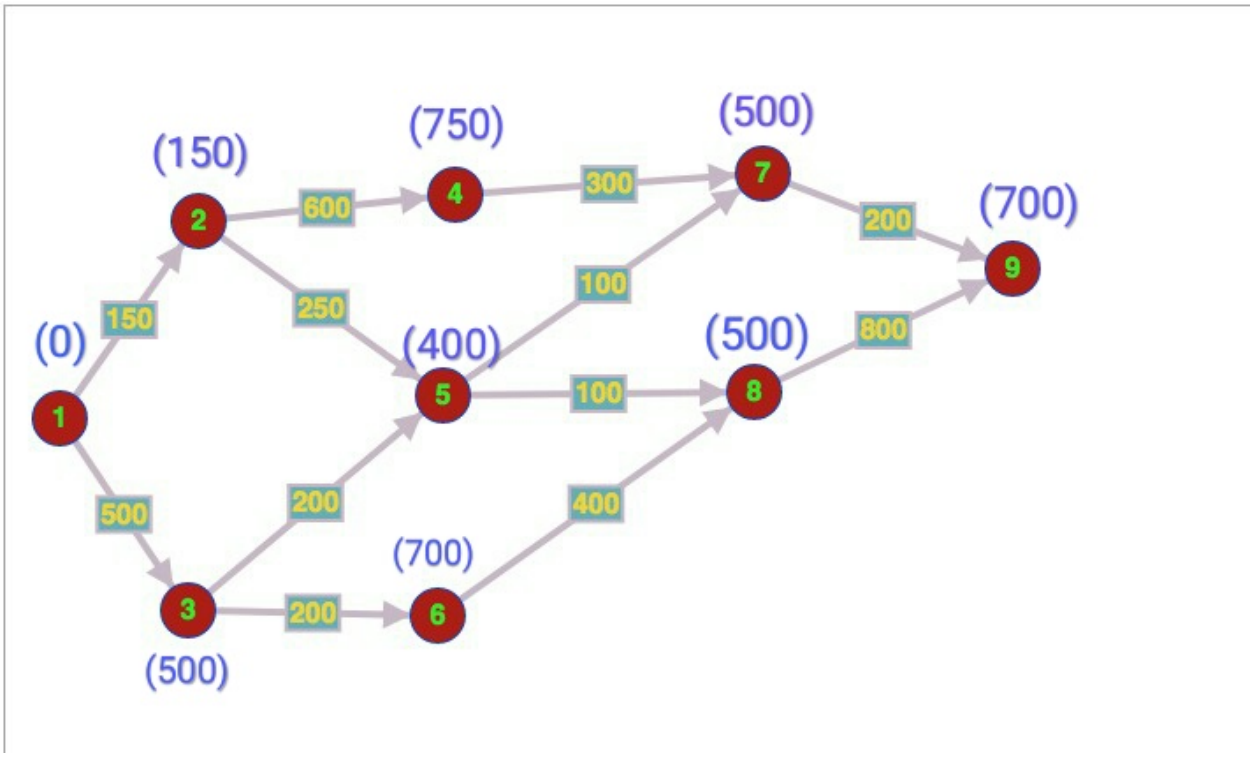
Dp[7] = min (dp[4] + 300 , dp[5] + 100) = 400

Dp[8] = min (dp[5] + 100 , dp[6] + 400) = 500

Dp[9] = min (dp[7] + 200 , dp[8] + 800) = 700

Answer : dp[9] = 700

The dp values :



Answer : 700

Subtask C :

Number of flat areas : 11

Number of slopes : 16

Slopes :

So now , let us calculate the energy required for each slope :

E(1, 2) = d * (60 - min(50 , s)) = 10*10 = 100

E(1, 3) = d * (60 - min(50 , s)) = 20*30 = 600

E(2, 4) = d * (60 - min(50 , s)) = 20*10 = 200

E(2, 5) = d * (60 - min(50 , s)) = 10*10 = 100

E(3, 5) = d * (60 - min(50 , s)) = 10*10 = 100

E(3, 6) = d * (60 - min(50 , s)) = 10*20 = 200

E(4, 7) = d * (60 - min(50 , s)) = 10*30 = 300

E(4, 8) = d * (60 - min(50 , s)) = 30*10 = 300

E(5, 8) = d * (60 - min(50 , s)) = 30*10 = 300

E(5, 9) = d * (60 - min(50 , s)) = 20*20 = 400

E(6, 9) = d * (60 - min(50 , s)) = 20*20 = 400

E(6, 10) = d * (60 - min(50 , s)) = 40*10 = 400

E(7, 11) = d * (60 - min(50 , s)) = 20*20 = 400

E(8, 11) = d * (60 - min(50 , s)) = 10*40 = 400

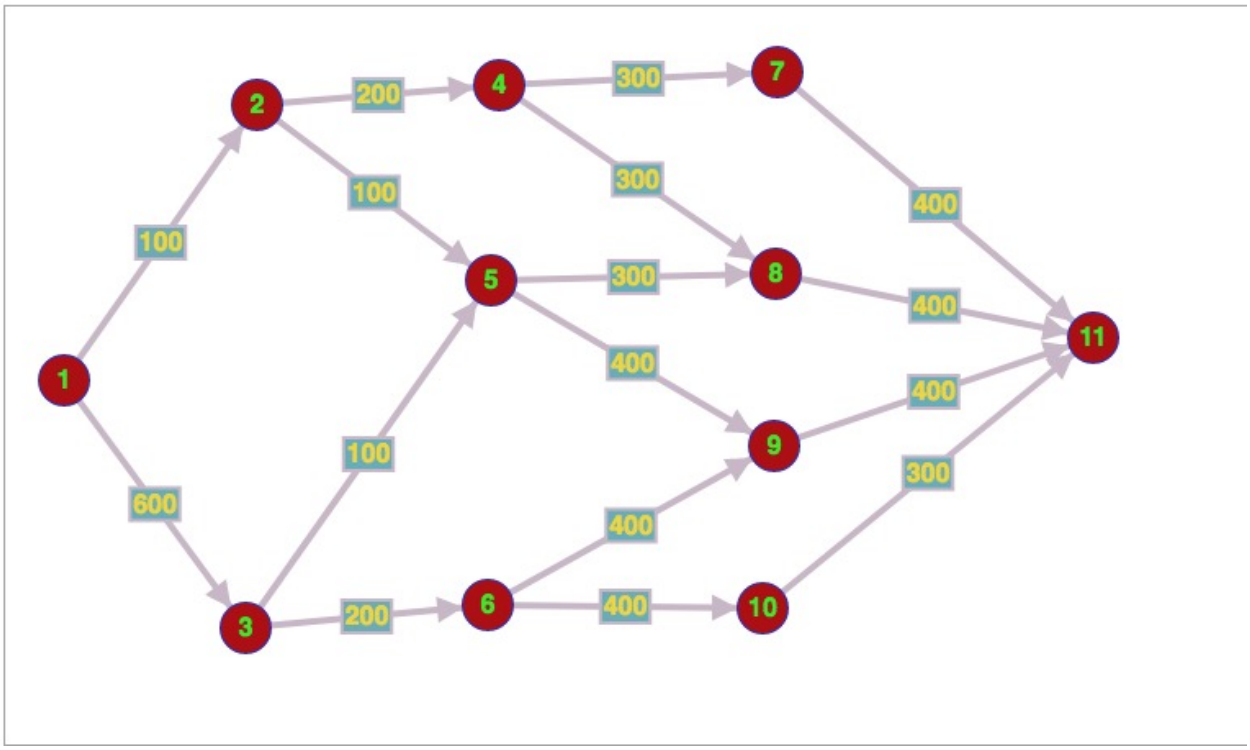
E(9, 11) = d * (60 - min(50 , s)) = 20*20 = 400

E(10, 11) = d * (60 - min(50 , s)) = 10*30 = 300

The table looks like this :

No	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.	1	1	2	2	3	3	4	4	5	5	6	6	7	8	9	10
To	2	3	4	5	5	6	7	8	8	9	9	10	11	11	11	11
s	60	30	50	60	50	40	30	50	60	40	40	50	40	20	40	30
d	10	20	20	10	10	10	10	30	30	20	20	40	20	10	20	10
e	100	600	200	100	100	200	300	300	300	400	400	400	400	400	400	300

The graph would look something like this :



So now , let us start filling the dp table :

Base case : dp[1] = 0

Relation :

Dp[2] = dp[1] + 100 = 100 (as we can only come from 1)

DP[3] = dp[1] + 600 = 600 (as we can only come from 1)

Dp[4] = dp[2] + 200 = 300 (as we can only come from 2)

Dp[5] = min (dp[2] + 100 , dp[3] + 100) = 200

Dp[6] = dp[3] + 200 = 600 (as we can only come from 3)

Dp[7] = dp[4] + 300 = 600 (as we can only come from 4)

Dp[8] = min (dp[4] + 300 , dp[5] + 300) = 500

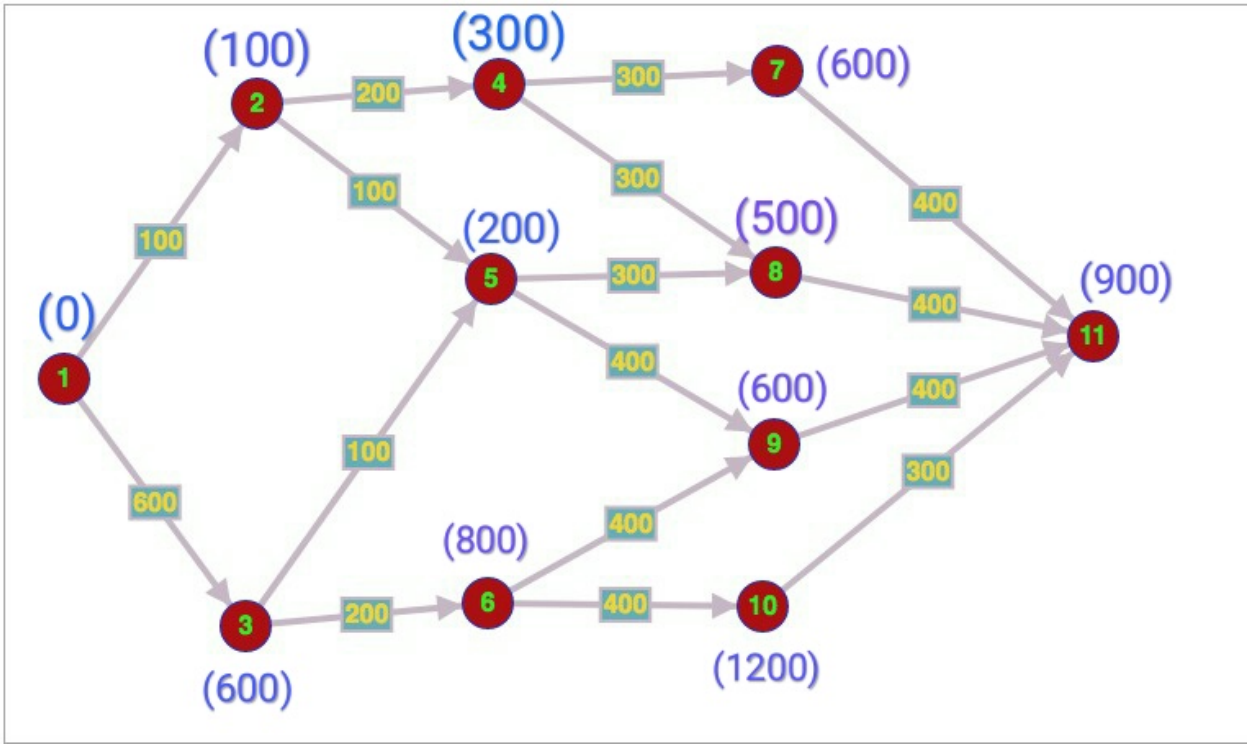
Dp[9] = min (dp[5] + 400 , dp[6] + 400) = 600

Dp[10] = dp[6] + 400 = 1200 (as we can only come from 10)

Dp[11] = min (dp[7] + 400 , dp[8] + 400 , dp[9] + 400 , dp[10] + 300) = 900

Answer : dp[11] = 900

The dp values would look something like this :



Answer : 900

Code in C++ → https://github.com/DheeruYS/ZIO-Codes/blob/main/2005/ZIO%202005%20P5.cpp

Hope you understood 🙂

PREREQUISITES:

1. Dynamic Programming
2. Basic Arithmetic

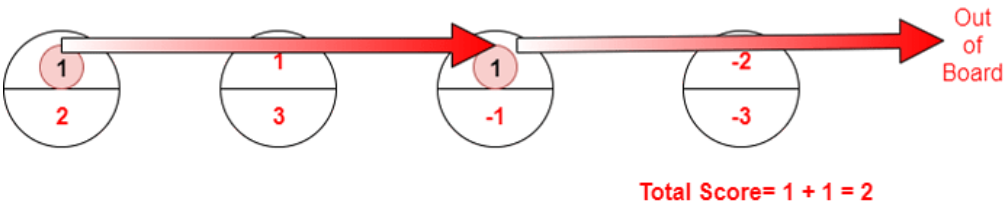
Problem in Short:

There is a game containing sequence of tiles each labelled with two numbers. You start at the first tile in the sequence and choose one number from each tile that you stop at, according to the following rules:

• At tile i, if you pick up the smaller number, you move on to the next tile, i+1, in the sequence.
• At tile i, if you pick up the larger number, you skip the next tile and move to tile i+2 in the sequence.

The game ends when your next move takes you beyond the end of the sequence. Your score is the sum of all the numbers you have picked up. Your goal is to maximize your score.

Example:



Explaination:

1. The first thing to note in the question is that you have to start from the first tile which means you have to select one number from the first tile. Similarly, note that game will only end when you cross the sequence of tiles which means there will be two cases when game ends:
   Suppose there are n tiles (n>1) numbered from 1 to n.

   a. If you select the large number in the tile (n-1).
   b. If you select any of the two numbers in the last tile (n).

2. To arrive at any tile there can be two possible moves. Suppose currently you are at tile (i)'.

   a. You can pick the smaller number in tile (i-1) and come at tile (i).
   b. You can pick the larger number in tile (i-2) and come at tile (i).

3. So just maintain two variables on each tile ("Small" and "Large") , suppose you are on tile (i) then
   "Small(i)" denotes what is the maximum score till tile (i) if you pick the smaller number at tile (i)
   "Large(i)" denotes what is the maximum score till tile (i) if you pick the larger number at tile (i).

4. Now the question arises how the value of Small and Large will be calculated at each step :
   Suppose at any tile (i) the smaller number is represented as tileSmall[i] and larger number is represented as tileLarge[i].
   Suppose currently you are at any tile(i) then
   Small(i) = tileSmall[i] + max(Small(i-1) , Large(i-2))
   Large(i) = tileLarge[i] + max(Small(i-1) , Large(i-2))

   Explanation of the above recurrence:
   Suppose you are standing at tile (i) currently so think of past that what are the possible ways to arrive at this tile (i) ,
   You can choose the smaller number at tile (i) and come to tile (i) = Small(i-1)
   You can choose the larger number at tile (i) and come to tile (i) = Large(i-2)
   So you have to choose the maximum of the two.

5. Calculate Small(i) and Large (i) for each tile in the sequence and the final answer will be:
   max(Small(n),Large(n),Large(n-1))
   Explanation of the above recurrence:
   In the problem it is given that game will end only when you cross the sequence which is possible by three

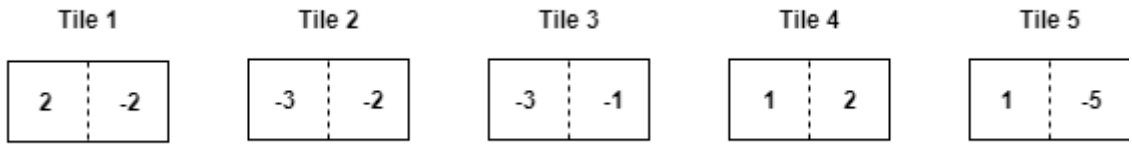
moves:

a. Pick the smaller number at last tile and get out of board. = Small(n)
b. Pick the larger number at last tile and get out of board. = Large(n)
c. Pick the larger number at second last tile and get out of board. = Large(n-1)

6. Now there are few base cases to think of:
   You know that you have to start from the first tile which means:
   Small(1) = tileSmall(1)
   Large(1) = tileLarge(1)
   To arrive at the second tile from start you have only one possible way that is to choose the smaller number in first tile because if you choose the larger number at first tile you will directly go to third tile.
   Small(2) = tileSmall(2) + Small(1)
   Large(2) = tileLarge(2) + Small(1)

Example:

Consider a sequence of 5 tiles.



Solution:

Tile 1:
tileSmall(1)= -2
tileLarge(1)= 2
Small(1)= -2 You have to start from the first tile.
Large(1)= 2 You have to start from the first tile.

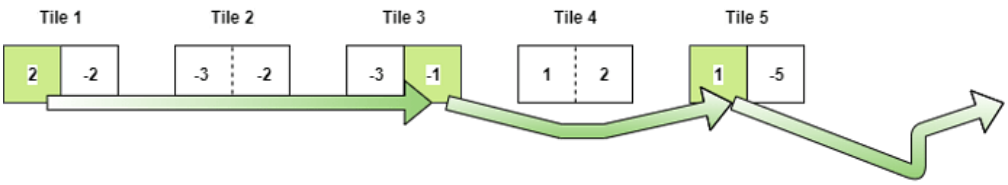
Tile 2:
There is only one way to arrive at any number on tile 2 that is to have a jump from tile 1 (small no.)
tileSmall(2)= -3
tileLarge(2)= -2
Small(2) = tileSmall(2) + Small(1) = -5
Large(2) = tileLarge(2) + Small(1) = -4

Tile 3:
tileSmall(3) = -3
tileLarge(3) = -1
Small(3) = tileSmall(3) + max(Small(2) , Large(1)) = -1
Large(3) = tileLarge(3) + max(Small(2) , Large(1)) = 1

Tile 4:
tileSmall(4) = 1
tileLarge(4) = 2
Small(4) = tileSmall(4) + max(Small(3),Large(2)) = 0
Large(4) = tileLarge(4) + max(Small(3),Large(2)) = 1

Tile 5:
tileSmall(5) = -5
tileLarge(5) = 1
Small(5) = tileSmall(5) + max(Small(4),Large(3)) = -4
Large(5) = tileLarge(5) + max(Small(4),Large(3)) = 2

```
Final Answer is :   max(Small(5),Large(5),Large(4)) = 2
```



Example 1:

Tiles	tileSmall	tileLarge
1	-2	2
2	-3	-2
3	-3	-1
4	1	2
5	-5	1
6	-2	4
7	-5	-4
8	-5	4
9	-5	-2
10	4	5

Solution:

Tiles	Small	Large
1	Small(1)=tileSmall(1)=-2	Large(1)=tileLarge(1)=2
2	Small(2)=tileSmall(2)+Small(1) = -5	Large(2)=tileLarge(2)+Small(1) = -4
3	Small(3)=tileSmall(3)+max(Small(2),Large(1)) = -1	Large(3)=tileLargel(3)+max(Small(2),Large(1)) = 1
4	Small(4)=tileSmall(4)+max(Small(3),Large(2)) = 0	Large(4)=tileLarge(4)+max(Small(3),Large(2)) = 1
5	Small(5)=tileSmall(5)+max(Small(4),Large(3)) = -4	Largel(5)=tileLarge(5)+max(Small(4),Large(3)) = 2
6	Small(6)=tileSmall(6)+max(Small(5),Large(4)) = -1	Large(6)=tileLarge(6)+max(Small(5),Large(4)) = 5
7	Small(7)=tileSmall(7)+max(Small(6),Large(5)= -3	Large(7)=tileLarge(7)+max(Small(6),Large(5)) = -2
8	Small(8)=tileSmall(8)+max(Small(7),Large(6)) = 0	Large(8)=tileLarge(8)+max(Small(7),Large(6)) = 9
9	Small(9)=tileSmall(9)+max(Small(8),Large(7)) = -5	Large(9)=tileLarge(9)+max(Small(8),Large(7)) = -2
10	Small(10)=tileSmall(10)+max(Small(9),Large(8)) = 13	Large(10)=tileLarge(10)+max(Small(9),Large(8)) = 14

Final Answer will be:max(Small(10),Large(10),Large(9)) = 14

Example 2:

Tiles	tileSmall	tileLarge
1	-1	1
2	-3	1
3	-1	4
4	-3	4
5	1	2
6	4	3
7	-4	3
8	1	4
9	2	4
10	-1	1
11	-2	-1
12	-4	-3

Solution:

Tiles	Small	Large
1	Small(1) = tileSmall(1) = -1	Large(1) = tileLarge(1) = 1
2	Small(2)=tileSmall(2)+Small(1) = -4	Large(2)=tileLarge(2) + Small(1) = 0
3	Small(3) = tileSmall(3) + max(Small(2),Large(1)) = 0	Large(3)=tileLarge(3) + max(Small(2),Large(1)) = 5
4	Small(4) = tileSmall(4) + max(Small(3),Large(2)) = -3	Large(4)=tileLarge(4) + max(Small(3),Large(2)) = 4
5	Small(5) = tileSmall(5) + max(Small(4),Large(3)) = 6	Large(5)=tileLarge(5) + max(Small(4),Large(3)) = 7
6	Small(6) = tileSmall(6) + max(Small(5),Large(4)) = 9	Large(6)=tileLarge(6) + max(Small(5),Large(4)) = 10
7	Small(7) = tileSmall(7) + max(Small(6),Large(5)) = 5	Large(7)=tileLarge(7) + max(Small(6),Large(5)) = 12
8	Small(8) = tileSmall(8) + max(Small(7),Large(6)) = 11	Large(8)=tileLarge(8) + max(Small(7),Large(6)) = 14
9	Small(9) = tileSmall(9) + max(Small(8),Large(7)) = 14	Large(9)=tileLarge(9) + max(Small(8),Large(7)) = 16
10	Small(10) = tileSmall(10) + max(Small(9),Large(8)) = 13	Large(10)=tileLarge(10) + max(Small(9),Large(8)) = 15
11	Small(11) = tileSmall(11) + max(Small(10),Large(9)) = 14	Large(11)=tileLarge(11) + max(Small(10),Large(9)) = 15
12	Small(12) = tileSmall(12) + max(Small(11),Large(10)) = 11	Large(12)=tileLarge(12) + max(Small(11),Large(10)) = 12

Final answer will be: max(Small(12),Large(12),Large(11)) = 15

Example 3:

Tiles	tileSmall	tileLarge
1	-5	-1
2	-1	1
3	-3	5
4	-5	2
5	-4	-2
6	-4	4
7	-2	5
8	-3	-2
9	-3	4
10	-5	3
11	-6	3
12	-1	5
13	-1	1
14	3	4

Screen… 936×628 5.78 KB

Solution:

Tiles	Small	Large
1	Small(1) = tileSmall(1) = -5	Large(1) = tileLarge(1) = -1
2	Small(2) = tileSmall(2) + Small(1) = -6	Large(2) = tileLarge(2) + Small(1) = -4
3	Small(3) = tileSmall(3) + max(Small(2),Large(1)) = -4	Large(3) = tileLarge(3) + max(Small(2),Large(1)) = 4
4	Small(4) = tileSmall(4) + max(Small(3),Large(2)) = -9	Large(4) = tileLarge(4) + max(Small(3),Large(2)) = -2
5	Small(5) = tileSmall(5) + max(Small(4),Large(3)) = 0	Large(5) = tileLarge(5) + max(Small(4),Large(3)) = 2
6	Small(6) = tileSmall(6) + max(Small(5),Large(4)) = -4	Large(6) = tileLarge(6) + max(Small(5),Large(4)) = 4
7	Small(7) = tileSmall(7) + max(Small(6),Large(5)) = 0	Large(7) = tileLarge(7) + max(Small(6),Large(5)) = 7
8	Small(8) = tileSmall(8) + max(Small(7),Large(6)) = 1	Large(8) = tileLarge(8) + max(Small(7),Large(6)) = 2
9	Small(9) = tileSmall(9) + max(Small(8),Large(7)) = 4	Large(9) = tileLarge(9) + max(Small(8),Large(7)) = 11
10	Small(10) = tileSmall(10) + max(Small(9),Large(8)) = -1	Large(10) = tileLarge(10) + max(Small(9),Large(8)) = 7
11	Small(11) = tileSmall(11) + max(Small(10),Large(9)) = 6	Large(11) = tileLarge(11) + max(Small(10),Large(9)) = 14
12	Small(12) = tileSmall(12) + max(Small(11),Large(10)) = 6	Large(12) = tileLarge(12) + max(Small(11),Large(10)) = 12
13	Small(13) = tileSmall(13) + max(Small(12),Large(11)) = 13	Large(13) = tileLarge(13) + max(Small(12),Large(11)) = 15
14	Small(14) = tileSmall(14) + max(Small(13),Large(12)) = 16	Large(14) = tileLarge(14) + max(Small(13),Large(12)) = 17

Final answer will be: max(Small(14),Large(14),Large(13)) = 17

PROBLEM:

Find the number of valid sequences of given length starting with a given digit from a 3x3 keypad. A sequence is considered to be valid if and only if every two consecutive digits in that sequence are adjacent in the keypad.

EXPLANATION:

Let **f(x,y,z)** be the number of valid codes of length **z** starting with the digit **x** and ending with the digit **y**, (1 <= x,y <= 9) and $\mathbf{S_y}$ be the set of the possible digits that can be there before **y**.

Present Digit(y)	Previous Digit can be one of these (S_y)
1	2, 4
2	1, 3, 5
3	2, 6
4	1, 5, 7
5	2, 4, 6, 8
6	3, 5, 9
7	4 ,8
8	5, 7, 6
9	6, 8

Table 1 : Summarises the set of possible digits there can be before a given digit

Note that, f(x, y, z) = $\sum_{s_i belongs\ to\ S_y}$ f(x, s_i, z-1)…(1)

And, answer to our subtasks are = ($\sum_{y=1}^{9}$ f(x,y,z))modulo 100…(2)

Let's take the example in the question, to understand the above expressions clearly.

Say that I want to calculate the value of f(4,1,3). We know from table 1 that the digit 1 can only be placed after the digits {2,4}. So on appending a 1 at the end of all the sequences of length 2 ending in a 2 or a 4, we can get all the sequences of length 3 ending in a 1, i.e, f(4,1,3) = f(4,2,2) + f(4,2,4).

Again, the value of f(1,9,2)=0, since it is impossible for a sequence of length 2 to start with the digit 1 and end with the digit 9!

To find out all the sequences of length 3, starting with the digit 4, we have to calculate f(4,1,3) + f(4,2,3) + f(4,3,3) +…+f(4,9,3) = sum{f(4,3,y), 1<=y<=9}

Solving the subtasks:

Recurrences:

f(x, y, z) = $\sum_{s_i belongs\ to\ S_y}$ f(x, s_i, z-1) …(1)

And, required answer = ($\sum_{y=1}^{9}$ f(x,y,z))modulo 100 …(2)

Where, x = Starting Digit of the sequence

y = Ending digit of the sequences

z = Length of sequence

Sy = Set of all possible digits that can occur before y .

Base Cases:

f(x, y, 1) = 1 ; when x=y (Since for length of sequence = 1, the starting and the ending digit must be the same.

In other words, the number of sequences of length 1, where the starting and ending digits are same is 1)

f(x, y, 1) = 0 ; when x not equals y(It's impossible to have a sequence of length 1, where the starting and the ending digits are different. Hence the answer is 0)

We construct a 2D-DP table of (y * z) to compute the values of recurrence (1) and then add them up to calculate the answer according to relation (2).

Subtask 1:

x =2, z = 8

Step 1: Fill in the base case

f(x,y,1) = x ; when x=y

f(x,y,1) = 0 ; when x not equals y

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0							
2	1							
3	0							
4	0							
5	0							
6	0							
7	0							
8	0							
9	0							

Step 2: Fill in the rest of the table according to recurrence (1).

Note that we are required to calculate the answer modulo 100, hence we only need the last two digits of the answer. So to ease our calculation we fill in the table with only the last two digits of the values obtained.

For e.g: f(2,2,7) = f(2,1,6) + f(2,3,6) + f(2,5,6) = 34 + 34 + 64 = 132. Instead of filling the cell with 132, we've directly filled it with 32, to ease further calculations!

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	1	0	5	0	34	0	60
2	1	0	3	0	18	0	32	0
3	0	1	0	5	0	34	0	60
4	0	0	2	0	16	0	28	0
5	0	1	0	8	0	64	0	12
6	0	0	2	0	16	0	28	0
7	0	0	0	3	0	30	0	52
8	0	0	1	0	14	0	24	0
9	0	0	0	3	0	30	0	52

Step 3: Sum up the last column and take modulo 100

Answer = (60+60+12+12+52) mod 100 = 36

Subtask 2:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	2	0	16	0	28	0	24	0

	1	2	3	4	5	6	7	8	9	10
2	0	1	0	8	0	64	0	12	0	96
3	0	0	2	0	16	0	28	0	24	0
4	0	1	0	8	0	64	0	12	0	96
5	1	0	4	0	32	0	56	0	48	0
6	0	1	0	8	0	64	0	12	0	96
7	0	0	2	0	16	0	28	0	24	0
8	0	1	0	8	0	64	0	12	0	96
9	0	0	2	0	16	0	28	0	24	0

Answer = (96 + 96 + 96 + 96) mod 100 = 84

Subtask 3:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	6	0	60	0	4	0	80	0	36
2	0	0	0	3	0	30	0	52	0	40	0	68	0
3	0	0	1	0	8	0	64	0	12	0	96	0	68
4	0	0	0	3	0	30	0	52	0	40	0	68	0
5	0	0	2	0	16	0	28	0	24	0	92	0	36
6	0	1	0	5	0	34	0	60	0	56	0	0	0
7	0	0	1	0	8	0	64	0	12	0	96	0	68
8	0	1	0	5	0	34	0	60	0	56	0	0	0
9	1	0	2	0	10	0	68	0	20	0	12	0	0

Answer = (36 + 68 + 36 + 68) mod 100 = 08

PREREQUISITES:

Basic Dynamic Programming

PROBLEM:

A binary string is considered as 'Good' when the pattern 0011 is concatenated k times (k≥1). Given a binary string B, find the number of 'Good' subsequences present in the string B. A subsequence is formed by dropping some characters from the initial string and keeping
the relative order of the remaining characters same.

EXPLANATION:

Let **m** = Length of String B,
n = Length of String A.
Let *f(m,n)* be the number of times the string A occurs as a subsequence in String B.

Case 1:

Consider the following example:
B = 001001101
A = 0011
Note that a subsequences '0011' in string B can be formed by appending a '1' at the end of each of the subsequences '001' in the string '00100110'.
So, if the last character of both the strings are the same, we can match them and move on to calculate for the rest of string B and string A, i.e. calculate *f(m-1, n-1).*
We can also ignore the last character of string B (i.e. delete it!) and search for the subsequence A in the rest of the string B, i.e, we can recur for *f(m-1, n)*.
So, the number of subsequence '0011' present in the substring '00100110'(here we have ignored the last character of B) = *f(m-1,n)*
Hence, the last characters being the same, we get,
The total number of subsequences of the string '0011' in B = *f(m ,n) = f(m-1, n-1) + f(m-1, n)*

Case 2:

If the last characters of string A and B are different, we can ignore the last character of B and proceed to calculate the number of subsequence of A in the remaining portion of the string B i.e, recur for *f(m-1,n)*
Combining both the cases, the final recurrences for this problem is:
f(m,n) = f(m-1,n-1) + f(m-1,n); Last characters are same
and
f(m,n) = f(m-1, n); Last characters are different

Base Case for the above recurrence:

f(m, 0) = 1 ; Since the number of empty subsequence in a given string is 1 (for **m** ≥ 0)
f(0, n) = 0 ; Number of subsequences of A in an empty string is 0 (for **n** ≥ 1)
Now in this question we have been asked to calculate the value of ***f*** for $\mathbf{A^k}$ in B for all **k** ≥ 1.
So to proceed we need to first find the maximum possible value of **k** for $\mathbf{A^k}$ to be a subsequence in B.
Then the answer is total number of good subsequences = number of subsequences as **A** + number of subsequences as $\mathbf{A^2}$ + number of subsequences as $\mathbf{A^3}$ +…+ number of subsequences as $\mathbf{A^k}$ = *f(m, 4) + f(m,8) + f(m,12) + f(m,16) + ….+f(m,4k)*

Subtask 1:

B = 001001101; m = 9

Since m = 9, the maximum length of a good subsequence in B is 8 and k = 2.

i.e, the longest possible good subsequence in B is 00110011;

So, A = 00110011; n = 8, k=2

Step 1: Fill in the base cases

	0('_')	1('0')	2('0')	3('1')	4('1')	5('0')	6('0')	7('1')	8('1')
0('_')	1	0	0	0	0	0	0	0	0
1('0')	1								
2('0')	1								
3('1')	1								
4('0')	1								
5('0')	1								
6('1')	1								
7('1')	1								
8('0')	1								
9('1')	1								

Step 2: Fill according the recurrence derived

	0('_')	1('0')	2('0')	3('1')	4('1')	5('0')	6('0')	7('1')	8('1')
0('_')	1	0	0	0	0	0	0	0	0
1('0')	1	1	0	0	0	0	0	0	0
2('0')	1	2	1	0	0	0	0	0	0
3('1')	1	2	1	1	0	0	0	0	0
4('0')	1	3	3	1	0	0	0	0	0
5('0')	1	4	6	1	0	0	0	0	0
6('1')	1	4	6	7	1	0	0	0	0
7('1')	1	4	6	13	8	0	0	0	0
8('0')	1	4	6	13	8	8	0	0	0
9('1')	1	5	10	23	21	8	0	0	0

Step 3:

Answer = Number of subsequences '0011' + Number of subsequences '00110011'

= f(9,4)+f(9,8) = 21 + 0 = **21**

Subtask 2:

B = 100110101010011

A = 001100110011

	0('_')	1('0')	2('0')	3('1')	4('1')	5('0')	6('0')	7('1')	8('1')	9('0')	10('0')	11('1')
0('_')	1	0	0	0	0	0	0	0	0	0	0	0
1('1')	1	0	0	0	0	0	0	0	0	0	0	0
2('0')	1	1	0	0	0	0	0	0	0	0	0	0
3('0')	1	2	1	0	0	0	0	0	0	0	0	0
4('1')	1	2	1	1	0	0	0	0	0	0	0	0
5('1')	1	2	1	2	1	0	0	0	0	0	0	0
6('0')	1	3	3	2	1	1	0	0	0	0	0	0
7('1')	1	3	3	5	3	1	0	0	0	0	0	0
8('0')	1	4	6	5	3	4	1	0	0	0	0	0
9('1')	1	4	6	11	8	4	1	1	0	0	0	0

	0(‘_’)	1(‘0’)	2(‘0’)	3(‘1’)	4(‘1’)	5(‘0’)	6(‘0’)	7(‘1’)	8(‘1’)	9(‘0’)	10(‘0’)	11(‘1’)
10(‘0’)	1	5	10	11	8	12	5	1	0	0	0	0
11(‘1’)	1	5	10	21	19	12	5	6	1	0	0	0
12(‘0’)	1	6	15	21	19	31	17	6	1	1	0	0
13(‘0’)	1	7	21	21	19	50	48	6	1	2	1	0
14(‘1’)	1	7	21	42	40	50	48	54	7	2	1	1
15(‘1’)	1	7	21	63	82	50	48	102	61	2	1	2

Answer = f(15,4) + f(15, 8) + f(15, 12) = 82 + 61 + 1 = **144**

Subtask 3 :

B = 01011000101001001

A = 0011001100110011

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0(‘_’)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1(‘0’)	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2(‘1’)	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3(‘0’)	1	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4(‘1’)	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5(‘1’)	1	2	1	2	1	0	0	0	0	0	0	0	0	0	0	0	0
6(‘0’)	1	3	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0
7(‘0’)	1	4	6	2	1	2	1	0	0	0	0	0	0	0	0	0	0
8(‘0’)	1	5	10	2	1	3	3	0	0	0	0	0	0	0	0	0	0
9(‘1’)	1	5	10	12	3	3	3	3	0	0	0	0	0	0	0	0	0
10(‘0’)	1	6	15	12	3	6	6	3	0	0	0	0	0	0	0	0	0
11(‘1’)	1	6	15	27	15	6	6	9	3	0	0	0	0	0	0	0	0
12(‘0’)	1	7	21	27	15	21	12	9	3	3	0	0	0	0	0	0	0
13(‘0’)	1	8	28	27	15	36	33	9	3	6	3	0	0	0	0	0	0
14(‘1’)	1	8	28	55	42	36	33	42	12	6	3	3	0	0	0	0	0
15(‘0’)	1	9	36	55	42	78	69	42	12	18	9	3	0	0	0	0	0
16(‘0’)	1	10	45	55	42	120	147	42	12	30	27	3	0	0	0	0	0
17(‘1’)	1	10	45	100	97	120	147	189	54	30	27	30	3	0	0	0	0

Answer = f(17,4)+f(17,8)+f(17,12)+f(17,16) = 97 + 54 + 3 + 0 = **154**