Problem A. Adding Integers

Define $a_0 = n$ and $a_{q+1} = 0$.

Define $b_i = a_{i-1} - a_i \geq 0$.

Note that $\sum\limits_{i=1}^{q+1} b_i = n$.

Look at $\binom{n}{a_1} \cdot \binom{a_1}{a_2} \cdot \ldots \cdot \binom{a_{q-1}}{a_q} = \binom{b_1+b_2+\ldots+b_{q+1}}{b_2+\ldots+b_{q+1}} \cdot \binom{b_2+b_3+\ldots+b_{q+1}}{b_3+\ldots+b_{q+1}} \cdot \ldots \cdot \binom{b_q+b_{q+1}}{b_{q+1}}$. This is the number of ways to color elements in $q+1$ colors such that there are b_i elements of color i.

Turns out that $f(q)$ is simply the number of ways to color n elements into $q+1$ colors.

So $f(q) = (q+1)^n$.

Problem B. Bottles

We will consider these permutations of all bottles $\frac{(e+p+w)!}{e!w!}$, since the order of poisonous bottles matters. We will count how many of those result in the elf being alive.

Consider we have some sequence of poisonous bottles and bottles with water. For each such sequence, there is the first "poison" event that happens after some time x. It means that if we try all ways to add elixir bottles to that sequence, the ways that result in at least one elixir bottle being among the first x bottles in the sequence keep the elf alive.

So if we count for each x, $f(x)$ — how many sequences of bottles out of $\frac{(p+w)!}{w!}$ consisting of poisonous and water bottles that have the first event happening right after element x. Then the answer is $\frac{(e+p+w)!}{e!w!} - \sum_x f(x) \times \frac{(e+p+w-x)!}{e!}$, where $\frac{(e+p+w-x)!}{e!}$ — the number of ways to place all elixir bottles in the sequence after position x, hence killing the elf.

How to calculate $f(x)$. Let's calculate $g(x)$ — the number of sequences of poisonous and water bottles, such that the event is **not** happening before x. It means $f(x) = g(x) - g(x+1)$. Consider all poisonous bottles that can poison before x: such i, so that $1 + i + t - 0.5 < x$, so $i \leq x - t - 1$. So there are $m = \min(x-t-1, p)$ such bottles. Other bottles can be in any order. Bottle 1 has to be at positions from $x - t$ up to the end $p + w$, so there are $p + w - (x-t) + 1$ ways. Bottle 2 has to be at positions from $x - t - 1$ up to the end, but one of the positions is taken by bottle 1, so also $p + w - (x-t) + 1$ ways, and so on. So there are $(p + w - (x-t) + 1)^m$ ways to place these m bottles, and $\frac{(p+w-m)!}{w!}$ to place others. So $g(x) = (p + w - (x-t) + 1)^m \frac{(p+w-m)!}{w!}$

Problem C. Counting Orthogonal Pairs

The angle of a regular polygon in degrees is $\frac{(n-2)\cdot 180}{n}$.

Consider all diagonals; there are $n-3$ diagonals from the vertex, and it divides into $n-2$ angles. Each angle is equal and of $\frac{(n-2)\cdot 180}{n(n-2)} = \frac{180}{n}$ degrees. The angle between two line segments coming from the vertex is $180 \cdot \frac{k}{n}$ for some $1 \leq k \leq n-2$. $180 \cdot \frac{k}{n} = 90$ only when $\frac{k}{n} = \frac{1}{2}$, and for integer k and n, this only happens when n is even, so $k = \frac{n}{2}$, and there are $\frac{n}{2} - 1$ ways to choose a pair of segments.

When n is odd, the answer is 0.

When n is even, the answer is $n \cdot \left(\frac{n}{2} - 1\right)$.

Problem D. Divine Tree

Count the number of G and B coins, let them be g and b, respectively.

Note that because the size is odd, after the type 2 operation, we will definitely know which of the two trees has G coins and which has B coins because their sizes would differ, and their sizes have to be g and b, respectively.

Also note that if we have chosen the edge for the type 2 operation, the number of times for each edge to be used in the type 1 operation is fixed; we can just count it. For each edge, we can see how many G and B coins are at each side of the edge and how many G and B coins there have to be at each side of the edge. This imbalance is the number of times we need to use the edge. You can prove that you can always make the type 1 operations so that every time you use an edge, the imbalance decreases for it, and finally get 0 imbalance for all edges.

Make a tree rooted at some vertex and get all subtrees of size g and all subtrees of size b. For every edge that you can use in the type 2 operation, either the subtree of size g or the subtree of size b has this edge coming from its root. Store information about subtrees of size b and g independently and in a similar way, and make updates independently. When getting the query answer, get for both and take the minimum.

Now consider only one case: we have subtrees of size g, and we want for each of them to store and update the cost to make type 1 operations, to afterwards make type 2 operations on the edge coming from its root. Note that these subtrees are disjoint.

Let's start with all weights equal to 0 and make an update operation for an edge weight. Consider some edge uv is updated, d is added to its weight, and v is more distant from the root than u. And consider a single subtree A of size g. There are three cases: if an edge is in A, if A is in the subtree of v, and otherwise.

1. If an edge uv is in A, then A's value will change by the number of B vertices in v's subtree times d;
2. If A is in the subtree of v, then A's value will change by the number of G vertices outside v's subtree times d;
3. Otherwise, A's value will change by the number of G vertices in v's subtree times d.

We can update these values for all trees simultaneously by ordering them in the order of DFS visiting them. There is only one subtree of size g for case 1, and cases 1 and 2 can't happen at the same time. Create a data structure that supports range addition and range minimum, and make an update in $\log n$ time.

Problem E. Experiments With Divine Trees

Iterate over s the number of G coins. There are a total of $\binom{n}{s}$ ways to put coins. If you don't have a division into s and $n-s$ vertices, then there is no way to make s G coins; just skip this s.

Consider a case when you can't remove any leaf. Look at the division into s and $n-s$. If you have a leaf in the s subtree with G, then you can remove it. If you have a leaf outside s with B, then you can remove it. It means the only way when you probably won't be able to remove is when all leaves in s are B, and all leaves in $n-s$ are G. The number of ways to make that is $\binom{n-l(s)-l(n-s)}{s-l(n-s)}$, where $l(s)$ and $l(n-s)$ are the number of leaves in each part.

Consider three cases; if any of them holds, all the $\binom{n}{s}$ ways should be counted:

1. You have at least two divisions into s and $n-s$ vertices.
2. Consider an edge uv that divides into s and $n-s$ (v at the side of s, and u at the side of $n-s$), and u is either a leaf, or it has at most one other edge coming out of it. This actually means there is a division into $s+1$ and $n-s-1$ that $s+1$ subtree contains a subtree of s, that is taken from the division into s and $n-s$.
3. You have at least one division into $s-1$ and $n-s+1$ vertices.

You can prove those by either removing any leaf inside s or outside all considered subtrees, and consider both cases when they are G or B, ending up that there is always a division after removal.

It turns out if these three cases don't hold, then a bad case happens; we can prove it the same way by trying to remove each type of leaf, and it happens that the cases above cover all ways to divide. So in this case, we take $\binom{n}{s} - \binom{n-l(s)-l(n-s)}{s-l(n-s)}$ ways.

Problem F. Fruit Tea

Note that if $\sum_{i=0}^{n} \max(0, a_{i+1} - a_i) = k$, then $\sum_{i=0}^{n} \min(0, a_{i+1} - a_i) = -k$, because the number of increases is equal to the number of decreases, since $a_0 = a_{n+1} = 0$. So we have a total of k increases and a total of k decreases. Let's look at these changes as $+1$ and -1. So there are k $+1$s, and some of them are grouped in the same $a_{i+1} - a_i$, and there are k -1s, and some -1s are grouped, but $+1$ and -1 can not be in the same group.

Consider we already have a valid (every prefix sum is non-negative) sequence of $+1$ and -1; then let's calculate the number of different ways to express it in a. Let's get a sequence of $+1$ and -1 and divide it into blocks of equal ones; it looks like `((+1)(+1)...(+1))((-1)(-1)...(-1))...((-1)(-1)...(-1))`. Note that there is an even number of blocks.

If we express it in terms of a_i, each a_i is some prefix sum of this sequence, with the only rule that between $+1$ and -1 there is at least one a_i. So there has to be at least one a_i between two blocks; a_0 is before the first block, and a_{n+1} is after the last block; other a_i can be anywhere. So we have $(n+2)$ elements to insert; we already inserted $2x+1$ of them — in the beginning and after each of the $2x$ blocks. Other elements can be inserted in any position, so there are $2k+1$ positions, and there are $(n+2-2x-1)$ elements to insert; the number of ways is $\binom{n+2-2x-1+(2k+1)-1}{(2k+1)-1}$.

Let $N(k, x)$ be the number of sequences of $+1$ and -1 such that:

- there are $2x$ blocks of them;
- there are k $+1$s;
- there are k -1s;
- each prefix sum is non-negative.

Then the answer will be calculated as $\sum_{x=1}^{k} N(k,x) \cdot \binom{n+1-2x+2k}{2k}$.

$N(k, x)$ can be calculated by the following formula: $N(k,x) = \binom{k-1}{x-1} \cdot \binom{k}{x-1} \cdot \frac{1}{x}$, or $N(k,x) = \binom{k}{x-1} \cdot \binom{k}{x} \cdot \frac{1}{k}$. See Narayana numbers, Catalan triangle, Dyck paths with k peaks.

Problem G. Gold Coins

The key observation is that the valid configuration always have specific form.

Let's remove all empty rows and columns, let's say we have board $n \times m$ now. Then in the remaining board there should be coordinates $x, y \ge 1$ such that the rectangle $[1..x][1..y]$ is full, and the rectangle $[x+1..n][y+1..m]$ is empty.

We can solve the problem using DP: calculate $d[i, j]$ as the answer for the rectangle $[i..n][j..m]$. If the rectangle is empty, the answer is 0, if not, iterate over pairs $(x, y) : x \ge i, y \ge j$, try to split the current rectangle into four, count number of nonempty rows and columns, and update the result.

Using some standard optimizations this can be done in $O(n^3 \log n)$ time.

Problem H. Heroes and Illusions

Let's say positions of real heroes are $a_1, a_2, ..., a_{m-1}$, also add $a_0 = 0$ and $a_m = n+1$. Consider values $b_i = a_i - a_{i-1}$, the lengths of segments between consecutive heroes. The segment $[l, r]$ contains odd number of heroes, if its ends belongs to segments with different parity, so the number of such segments is

$(a_1+a_3+a_5+...)\cdot(a_2+a_4+a_6+...) = k$. Let's say $x = (a_1+a_3+a_5+...)$, then we have $x\cdot(n+1-x) = k$. From this equation we can find x, and then we have standard problem of counting the number of partitions of x and $n+1-x$.

Problem I. Interesting Permutations

The whole statement is a bit misleading, and it's just about making a permutation s, of 1 to n, such that each new s_i has an absolute difference smaller than or equal to k with some previous s_j, where $j < i$.

You can notice that when you know s_1, all the numbers smaller than s_1 and those greater than it form independent problems. A subproblem is of the form: you start with $s_0 = 0$, and afterward, we need $s_1, \ldots, s_V$ to be a permutation of length V with the same absolute difference constraint. Let's call the number of ways $f(V)$. This number of ways can be found using some dynamic programming, prefix sums, and factorials in linear time for all relevant V.

Let's call those two subproblems $f(L)$ and $f(R)$, corresponding to the left and right sides of s_1, respectively. Then, if we fix s_1, the total number of ways is calculated as:

$$f(L) \times f(R) \times \binom{L+R}{L}$$

If we take $g(L) = \frac{f(L)}{L!}$ instead, this is kind of like the convolution of two arrays.

Now we get queries where we need to answer for $l \le s_1 \le r$, and the length is n, for some n. We can reduce this to only queries with $1 \le s_1 \le r$, and the length is n.

Because for a fixed r, we can calculate the answer for all n by convolving arrays $g(1 \ldots r)$ and $g(1 \ldots \mathrm{maxN})$, so we can do square root decomposition. Precalculate the answer for all $r = k \times \sqrt{N}$, and all n using Number Theoretic Transform (NTT).

Then, to answer queries, take the largest $k\sqrt{N}$ underneath the queried r, and brute force the last $\le \sqrt{N}$ terms. To balance the square root decomposition, you should rather take a larger block size. The complexity becomes $O(n\sqrt{n \log n})$.

Problem J. Jumping Game

Consider a more general problem, where we have the undirected graph G and a token initially placed in vertex s. Players take turns moving the token, and the token cannot visit the same vertex twice. If a player cannot move, they lose.

This is a well-known problem, connected to maximal matchings. In particular, it's easy to prove that the first player loses iff there is a maximal matching that doesn't cover vertex s.

Now, in our problem, we have a very specific graph. The intuition suggests that for the large board there is always some perfect (or almost perfect, if the size is odd) matching. That's actually true. We just need to solve some small cases manually. Namely, cases $1 \times x$, $2 \times x$, 3×3, and 3×5.

Problem K. Kangaroo On Graph

Make dynamic programming, calculate for each edge the value $d[uv]$, the minimal cost of the path ending with edge uv. To calculate this value, let's look at all possible previous edges xu. We need to find the minimum $d[xu]$ over all x, except the ones that form a forbidden triplet. We can store all the forbidden x for each edge uv, and just iterate all edges xu in increasing order by $d[xu]$, and pick the first one that is not forbidden. Time $O(n + m + k \log n)$

Problem L. Low Cost Set

What can we do better than just take all segments to set s? It's easy to see that the only way to improve this solution is to change the pair of segments (a, c) and (b, d), such that $a < b < c < d$, with three segments (a, b), (b, c), (c, d). If we do this, we will decrease the total cost by $c - b$. So the task is to build pairs of segments, maximizing the total improvement. This can be done using a (non-bipartite) maximal weighted matching algorithm.