A. Festival Organization

It is easy to prove that the answer for the query is $\sum_{n=l}^{r}\binom{F_{n+2}}{k}=\sum_{n=0}^{r}\binom{F_{n+2}}{k}-\sum_{n=0}^{l-1}\binom{F_{n+2}}{k}$, where F_n are Fibonacci numbers. Let's notice, that $\binom{x}{k}$ is a polynomial for x and can be expressed in the form $c_0+c_1x+\ldots+c_kx^k$. Knowing this we can express the answer in different way $\sum_{n=0}^{r}\binom{F_{n+2}}{k}=\sum_{n=0}^{r}\sum_{m=0}^{k}c_mF_{n+2}^m=\sum_{m=0}^{k}c_m\sum_{n=0}^{r}F_{n+2}^m$. Thus we reduced our problem to computing the sum of mth powers of Fibonacci numbers. To do so we will refer to Binet's formula

$F_n=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$ or $F_n=\frac{\sqrt{5}}{5}\left(\phi^n-\psi^n\right).$

$$F_n^m=\left(\frac{\sqrt{5}}{5}\right)^m\left(\phi^n-\psi^n\right)^m=\left(\frac{\sqrt{5}}{5}\right)^m\sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\phi^{nj}\psi^{n(m-j)}$$

$$\sum_{n=0}^{r}F_n^m=\left(\frac{\sqrt{5}}{5}\right)^m\sum_{n=0}^{r}\sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\left(\phi^j\psi^{(m-j)}\right)^n=\left(\frac{\sqrt{5}}{5}\right)^m\sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\sum_{n=0}^{r}\left(\phi^j\psi^{(m-j)}\right)^n$$

The inner sum is almost always a geometric progression with $b_0=1$ and $q=\phi^j\psi^{(m-j)}$, except for the cases when $\phi^j\psi^{(m-j)}=1$, but we may avoid any special cases by computing it in a way similar to binary exponentiation.

Indeed, in order to compute $\sum_{n=0}^{r}q^n$, we may start with computing $\sum_{n=0}^{2^k-1}q^n$ and q^{2^k}. Two sums with m1 and m2 elements can be merged together in the following way: $\sum_{n=0}^{m1+m2-1}q^n=\sum_{n=0}^{m1-1}q^n+q^{m1}\sum_{n=0}^{m1+m2-1}q^n$.

Thus we can compute the sum of mth powers only with additions and multiplications. The only difficulty is that there is $\sqrt{5}$ present in these formulas (in ϕ and ψ) and there is no such number modulo 10^9+7. The solution to this is simple: let's never specify an exact value for it. This way we will always work with numbers of the form $a+\sqrt{5}b$.The addition and multiplication of these numbers is fairly easy:

$$\left(a+\sqrt{5}b\right)+\left(c+\sqrt{5}d\right)=(a+c)+\sqrt{5}(b+d),$$

$$\left(a+\sqrt{5}b\right)\cdot\left(c+\sqrt{5}d\right)=(ac+5bd)+\sqrt{5}(ad+bc).$$

These are the only operations that we required. The sum of Fibonacci numbers (which are integers) is also integer, so the final result will never contain any $\sqrt{5}$. Thus we have solved this problem.

B. R3D3's Summer Adventure

Basically the problem can be formulated in the following way: Let's build a binary tree with edges labeled '0' and '1'with N leaves so that sum of costs of paths to all leaves is minimal. This tree is also called 'Varn code tree'*. Varn code tree is generated as follows. Start with a tree consisting of a root node from which descend 2 leaf nodes, the costs associated with the corresponding code symbols. Select the lowest cost node, let c be its cost, and let descend from it 2 leaf nodes c + c(0) and c + c(1). Continue, by selecting the lowest cost node from the new tree, until N leaf nodes have been created.

This greedy approach can be done in O(NlogN) if we actually construct a tree. Basically we do the following thing N times: out of all codes we have select the lowest, delete it and create 2 new codes by adding '0' and '1' to the one we deleted. If done with some standard data structure this is O(NlogN).

If we actually don't need the tree and the coding itself, just the final cost, we can improve to O(log^2(N)). Let's define an array F to represent this tree, where F[i] will be number of paths to leaves in this tree with cost equal to i. While sum of all values in F is lower than N we do the following procedure: find an element with lowest 'i' where F[i] > 0. Then, F[i+c(0)] += F[i]; F[i+c(1)] += F[i]; F[i] = 0. Basically we did the same thing as in previous algorithm, just instead of adding 2 edges to the leaf with lowest cost we did that to all leafs that have lowest cost in a tree. Numbers in this array rise at least as fast as Fibonacci numbers, just the array will be sparse. So if instead of array we use some data structure like map, we get O(log^2(N)) complexity.

* B. F. Varn. "Optimal Variable Length Codes (Arbitrary Symbol Cost and Equal Code Word Probabilities)." Inform. Contr. 19 (1971):289-301.

C. Potions Homework

Everything is pretty simple here. What should be done here is to give the laziest student the easiest task, because it is always optimal to do so. If we do that continuously, it becomes obvious that solution is to sort the given array and get the following formula:

$$\sum_{i=1}^{N} A[i] * A[N-i-1]$$

D. Dexterina's Lab

Dynamic programming – $\mathcal{O}(xn^2)$

A well-known result from game theory states that the winning player of a game of Nim is determined only by the bitwise XOR of the piles' sizes. The first player has a winning strategy if (and only if) this value is not zero.

Let P_i be the probability that a pile contains i objects. Define $d_{m,x}$ as the probability that the bitwise XOR of m random sizes is equal to x. The values of d can be expressed with the recurrence relation

$$d_{m,x} = \begin{cases} 1 \text{ if } m = 0 \wedge x = 0 \\ 0 \text{ if } m = 0 \wedge x \neq 0 \\ \sum_{i=0}^{N} d_{m-1,i} P_{i \oplus x} \quad \text{otherwise} \end{cases}$$

where N is the maximal possible value of the XOR (one less than the first power of 2 greater than the maximal pile size n) and $\oplus$ the bitwise XOR operator.

The probability that the second player wins is $d_{n,0}$. Since the first player wins iff the second does not, we can calculate the values of d in $\mathcal{O}(xn^2)$ and output $1 - d_{n,0}$.

Matrix exponentiation – $\mathcal{O}(x^3 \log n)$

Let D_i be the vector $(d_{i,0} \quad d_{i,1} \quad \cdots \quad d_{i,N})^T$. The transformation that produces D_{i+1} from D_i is linear, and can therefore be expressed as $D_{i+1} = MD_i$, where M is a matrix given by $M_{i,j} = P_{i \oplus j}$.

Since matrix multiplication is associative, $D_n = M^n D_0$. We can calculate M^n using $\mathcal{O}(\log n)$ matrix multiplications, for a total complexity to calculate D_n (which contains the solution $d_{n,0}$) of $\mathcal{O}(x^3 \log n)$.

This problem can also be solved in:

Vector exponentiation – $\mathcal{O}(x^2 \log n)$

Faster multiplication – $\mathcal{O}(x \log x \log n)$

Even faster multiplication – $\mathcal{O}(x \log x + x \log n)$

But O(x^3 logN) is enough to get AC

E. Paint in really, really dark grey

Root the tree at node 1. Now notice that if we had a function F such that F(**n**) colors the whole subtree of node **n** black (except maybe n itself) and returns us to **n** in the process, we can easily solve the problem. Let **p** be the parent of **n**. Then after entering **n** do F(**n**). If **n** is black, go to **p** and never return to that subtree again. Otherwise, go from n to p then to n then to **p**. Now n is black and we can continue to do the same for other children of **p** and so on.

Now it is easy to see that all that weneed to do is to make DFS-like tour of the tree and upon returning from a node to corresponding parent we check the color of child node. If it is black, continue normally, otherwise visit it again and again return to parent and then continue normally. The only exception is the root node as it has no parent. After finishing the tour and ending up in node 1, if it is black, we are done. If not, it is the only white one and we can select any child **c** of 1 and do 1 - c - 1 - c. Now all nodes are black.

It is recommended to do the implementation with stack. Complexity is O(N).

F. Heroes of Making Magic III

For queries of type 2, we are only interested in the elements A_k with n in the interval $[i, j]$, and nothing else. For the sake of convenience, label those elements as $a_0, a_1, \ldots, a_n$, where $n = j - i + 1$.

It can be easily seen that in order to clear a segment, we can just move back and forth between positions 0 and 1 until a_0 becomes 0, move to 1 and alternate between 1 and 2, etc – until we zero out the last element. If we want this procedure to succeed, several (in)equalities must hold:

- $a_0 \geq 1$ (since we have to decrease the first element of the segment in the first step)
- $a_1 - a_0 \geq 0$ (after following this procedure, a_1 is decreased exactly by a_0, and we end up on a_1)
- $a_2 - (a_1 - a_0 + 1) = a_2 - a_1 + a_0 - 1 \geq 0$, or, $a_2 - a_1 + a_0 \geq 1$
- $a_3 - a_2 + a_1 - a_0 \geq 0$, ...
- In general, $a_m - a_{m-1} + a_{m-2} - \cdots + (-1)^m a_0$ has to be greater or equal than 0, if m is odd, and 1, if m is even

Equivalently, if we define a new sequence d', such that $d'_0 = a_0$, and $d'_m = a_m - d'_{m-1}$, these inequalities are equivalent to stating that $d'_0, d'_2, d'_4, \ldots \geq 1$ and $d'_1, d'_3, d'_5, \ldots \geq 0$.

Of course, we do not have to store the d array for each pair of indices i, j, but it is enough to calculate it once, for the entire initial array A. Then, we can calculate the appropriate values of d'_k for any interval $[i, j]$ (k is the zero-based relative position of an element inside the segment): Let $c = d_{i-1}$. Then,

- For even values of k, $d'_k = c + d_{i+k}$
- For odd values of k, $d'_k = d_{i+k} - c$

Now we only need a way to maintain the array d after updates of the form „1 i j v“. Fortunately, we can see that this array is updated in a very regular way:

- Elements on even positions inside the interval $[i, j]$ are increased by v
- If $j - i$ is even, elements on positions $j + 1, j + 3, j + 5, \ldots$ are decreased by v, and elements on positions $j + 2, j + 4, j + 6, \ldots$ are increased by v

All of this can be derived from our definition of d.

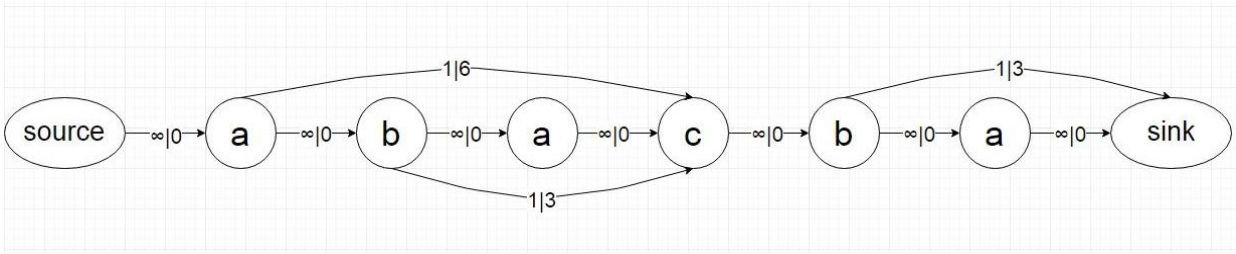
Both of these queries can be efficiently implemented using a lazy-propagation segment tree, where each internal node stores two numbers, the minima of its odd- and even-indexed leaves.

Using the approach described above, we arrive at a solution with the complexity of $O(q \log n)$.

G. Underfail

In the basic case (when X=1) we have a common DP problem. Unfortunately, for larger X it is much more complicated, so the basic case cannot be generalized.

Instead, we can look at this problem as a graph problem. We represent the crossword as directed weighted graph, where edges have both a cost and a capacity. First, we represent every letter from crossword as a vertex, and add an extra source and sink node. Then, we connect each letter's vertex with the next one with an edge of capacity ∞ and cost 0. Also, we connect the source node with the first letter and the last letter with the sink node using the same kind of edge. Finally, for every position where one of the given words matches a substring, we connect first letter of the match with letter just after the end of the match with an edge of capacity 1 and a cost equal to the score we get for the word. By doing it for example input we get this graph:



Any flow with total capacity X is equivalent to a set of selected words that satisfies the given constraint, since we can have at most X "overlapping" 1-capacity edges. The problem now reduces to maximizing the cost of that flow, which can be done by adapting an algorithm that solves the min-cost flow problem (by using negative values). Using an adaptation of Bellman-Ford algorithm we can get a complexity of $O(E^2VloqV)$.

H. Pokermon League challenge

The key idea is to first put players into conferences and then assign them teams. We go through all players and put then into conferences such that first condition is satisfied. We take them one by one and select a conference for each player such that he hates more (or equally many) players in the other conference that the one we put him in. That means that at any point of this process, there will be more hate edges connecting players in different conferences than those connecting players in same conferences. Thus, in the end, we have satisfied first condition. Complexity is O(N + E) for this part.

Now we try to satisfy second condition by assigning teams to players. Let **S** be the set of all teams that appear on at least one wish-list. Let's select some subset **A** of **S**. We select it in a way that every element from **S** will be thrown in it with 50% chance. Now, we will assign teams from **A** to players in conference 1 and teams from $\mathbf{A^c}$ to players in conference 2. If such assigning is posible while satisfying second condition, we will be done. Let's see what is the chance that a particular player in conference 1 can't be assigned a valid team from **A**. That means none of at least 16 teams on his wish-list are in **A**. The chance for that is $\frac{1}{2^{16}}$. Same goes for all players in conference 1 and equivalenty for conference 2. So, the chance that at least one player can't be assigned a team is at most $N \cdot \frac{1}{2^{16}} \leq 0.77$. This means that there is at least 23% cnance that this method will give us the final solution which fulfills both problem conditions. This means that after a several steps, we will very likely solve the problem. Complexity is O(NL) for this part. For example if we do 30 steps, the probability that we will find a correct solution is $1 - 0.77^{30} = 0.9996$.

I. Cowboy Beblop at his computer