Problem Tutorial: "Buffaloes"

This problem can be reduced to "find the number of permutations of size N with LIS (Longest Increasing Subsequence) not 3 or 4", as the lines are permutations, and a possible family is a decreasing subsequence (which is analogous to an increasing one).

A well-known algorithm to find the LIS of a sequence is to determine, for each i the smallest number that can be the ith number of an increasing subsequence. Notice that i does not exceed the size of the LIS. Here the idea is similar, but we will have those numbers as the state in a dynamic programming. This will be used to determine the number of permutation with LIS 3 or 4, and the answer will be to subtract that number from $N!$.

The dp will create the permutation number by number. Let B_i be the smallest selected number that can be the ith number of an incrasing subsequence. Consider `solve(a1, a2, a3, a4)` where `a1` is the amount of unselected numbers of the permutation that are smaller than B_1 (that is, the smallest number selected so far). We have `a2` is the amount of unselected numbers between B_1 and B_2, and `a3` and `a4` is defined analogously. Notice that we don't need `a5` because we won't work with IS's of size more than 4.

At each step, we must choose an unselected number and add it to our permutation. If we choose a number smaller than B_1 (there are `a1` possibilities), it will be the new B_1, and the sizes of `a1` and `a2` need to be changed accordingly, depending on each number was chosen. The same goes for numbers in $[B_1, B_2]$ and $[B_2, B_3]$. For numbers in $[B_3, B_4]$ we can only choose the largest of these numbers, otherwise the largest one would be chosen in some other iteration and eventually we would have a permutation with LIS 5. (That is why we don't need `a5`)

It may be necessary to use a boolean value in order to avoid counting permutations with LIS 1 and 2.

We need to test `a1 + a2 + a3 + 1` numbers and we have `a1 + a2 + a3 + a4` $\leq N$, so the complexity is $\mathcal{O}(N^5)$, with a really small constant (around $\frac{1}{12}$).

Problem Tutorial: "Cahokia ruins"

Let L and R be integer arrays representing the width of the west and east wall, respectively. Note that, if we fix the row i, the time it takes for the walls to collide in this line is $\frac{1}{2} \cdot (W - L[i] - R[i])$. Then, we need to find

$$\frac{1}{2} \cdot \min\{W - L[i] - R[i] : 1 \leq i \leq H\}.$$

Clearly, we can find the answer in time $\mathcal{O}(H)$.

Problem Tutorial: "A Word to Trump All"

To solve this, you need to use Aho-Corasick (there are many great references on the web and books), a trie for a set of strings that has fail links, a link that points, for a node v, to the largest proper suffix of v that is a prefix of some string of the set, considering the name of the vertex is its associated prefix. It can be preprocessed so that, from any vertex u, given a letter c, you can change in time $\mathcal{O}(1)$ to the vertex v such that it is the largest suffix of uc that is represented in the trie (that is, it is a prefix of some of the subsets' strings).

A path $u_1, \ldots, u_k$ in that digraph (it is no longer a tree) represents a string $c_1 c_2 \ldots c_k$, and at the step g the node u_g is the largest prefix of some string of the set that is a suffix of $c_1 c_2 \ldots c_g$.

We will create this digraph for all $N + M$ strings. What the problem wants is a path of smallest size such that all N horrible words occur and none of the M kind words do. This will be done using BFS. Since N is very small, we will use a bitmask to identify which horrible strings we have already created. For each node u we will precalculate a value b_i that is the bitmask of the set of all horrible strings that are sufixes of this node (this can be done using the fail links).

The state of the BFS is (u, b) where u is the node of the digraph and b is the bitmask of already completed strings. Whenever we reach a new node v we update the value of b by doing a bitwise OR with b_v. We

mark nodes u such that u has a kind string, and never visit them (which guarantees our word does not have a kind string as substring), this happens if any node of a kind word can be reached via fail links.

The BFS will determine the minimum path from $(\text{root}, 0)$ to $(\text{any node}, 2^N - 1)$, because $2^N - 1$ is the full bitmask, meaning we created all horrible strings. The complexity is $\mathcal{O}(2^N * \sum_{i=1}^{N+M} |S_i|)$, where S_i is the $i$rh string in the input. This sum is guaranteed to not exceed 300.

Problem Tutorial: "The Declaration of Independence"

What the problem asks is to implement a persistent queue, that is, a queue that can be changed to a previous version.

There are two main ways to do this, a nice online way and a faster offline one.

If we can answer these queries offline, as we could do in this problem, we can create a version tree, that is, we create a node for each version, and if we apply query i to a node v, we create a new node i and connect it to node v, labeling the edge with the query number.

After doing this, we can process all the answers with a DFS, each edge "going down" (away from the root), means we need to do the operation in that edge, when we "go up" again in the same edge we must undo that operation. That means, when we are examining node v, only the operations that lead from the root to v are done, and that is exactly what was the version v of the queue.

It is necessary to use a deque to be able to undo queue operations. Doing this DFS, we ever only need to undo the last operation done, like in a stack. This solution is offline and has complexity $\mathcal{O}(Q)$.

The online solution is more complicated. We will also create a tree, but it is not the same tree, this tree will contain all values added to the queue so far. For each query i so far we will store two nodes s_i and e_i, the start and end of the queue in version i, and we can change to the correct version of the queue in each query.

If the query i is to add x to version v to the end of the queue, we just need to create a new node connected to e_v and update $s_i = s_v$ and $e_i =$ new node . If the query i is to pop the version v, we need to move s_v one step closer to e_v. But since this is a tree and not just a path, it is not so easy. One solution is to use sparse tables in each node to be able to jump from e_v to the next node after s_v in the path to e_v, similar to a Level Ancestor query.

This solution is online and has complexity $\mathcal{O}(Q \cdot \log(Q))$.

Problem Tutorial: "Pop Divas"

The first observation is that the square root on the glauberian mean makes no difference, and we can consider only the product of the numbers. As the limits are very small, we can calculate the mean of all $2^N - 1$ possible albums for Taylor and all $2^M - 1$ values for Katy, and then use some structure (`set` for example) to find quickly for each Katy album if there is one by Taylor with the same value.

The problem is the product of all songs may be very big (up to 10^{144}), we can't store them in a normal integer. There are three main solutions:

- Use BigInteger from Java or some other bignum class. That works, but if you don't use Java it may be complicated.
- Use hashing, that is, take the result of the product modulo a big prime (in the order of 10^9). Just doing this won't work because we're checking $\approx 2^{32}$ pairs, so the chace of collision using just one prime is big. The solution is to use 2 primes.
- Before multiplication, factor all numbers, and store their prime decomposition. When multiplying two numbers, add their primes. The size of this representation will be around log of the size of the numbers, which is fine. I recommend using `map<int, int>` to represent each number, for each prime storing its power.

This can be implemented in time $\mathcal{O}(2^K \cdot K)$ (hash) or $\mathcal{O}(2^K \cdot K \cdot P)$, where $K = \max(N, M)$ and P is the number of primes in all of the numbers decomposition, this will be very small.

Problem Tutorial: "Protecting the Central Park"

Let $G = (V, E)$ be an undirected graph such that V represents the places we want to protect and E represents the roads that connect those places. The problem asks for a partition of E into paths of length 2, in other words, we want a collection of paths

$$\mathcal{C} = \{P_1, P_2, \ldots, P_k\}$$

such that $|E(P_i)| = 2$, for $i = 1, \ldots, k$, $E(P_i) \cap E(P_j) = \emptyset$ if $i \neq j$, and $\cup_{i=1}^{k} E(P_i) = E$. Observe that the statement guarantees the existence of such solution.

In first place, note that if try to apply a recursive algorithm to find such partition and if we remove any pair of adjacent edges in G, the resulting graph can be disconnected, and we would need to deal with each component separately, also being careful to leave each component with an even number of edges. Thus, we desire to create an algorithm that find this partition in a such a way that, after removing a pair of adjacent edges, the resulting graph is connected.

Let T be a spanning tree of G. Observe that if we remove a leaf of T, the resulting graph is a tree and, therefore, connected. So, our strategy will be to process the vertices in G following a post-order of T. Let $\delta_G(v)$ be the edges incident to v in G. Note that, if $|\delta(v)|$ is even, we can pair arbitrarily the edges in $\delta(v)$ to obtain paths of length 2. Furthermore, the graph $G' = G - v$ is connected and has even number of edges. In the other hand, if $|\delta(v)|$ is odd, we can pair the edges in $\delta(v) - uv$ to obtain such paths. Also, the graph $G' = G - \delta(v) + uv$ is connected and has even number of edges. In this way, we can process each vertex in V following a post-order of T and obtain the answer. This algorithm can be implemented to run in $\mathcal{O}(|V| + |E|)$.

Problem Tutorial: "King of Tokyo"

The first thing to notice is that the order of the dice does not matter, only the quantity, and all the unnumbered faces can be treated equally. This means instead of considering 6^6 different dice rolls, we consider only the $\binom{9}{3} = 84$ different rolls (warning: these rolls are no longer equiprobable).

We will solve the problem with dynamic programming, we will simulate all possible plays and results. The dp `solve(dice, k)` will tell what is the expected number of points, playing optimally, when you rolled `dice` (one of the 84 different rolls), given you can reroll a subset `k` times.

To do this, you should try all of the possible subsets of `dice` (there are $\binom{10}{4} = 210$ essentially different subsets of all dice, and this is an upper bound for the subsets of `dice`). After rerroling those, say, G dice, each of the 6^G rolls are equiprobable, but there are only $\binom{G+3}{3} \leq 84$ essentially different, so we can precalculate the probability that each of them happens, and use this to call de dp recursively and determine the expected value doing this reroll. We then get the maximum among all possible rerolls.

You should use the same dp for all T cases, without recalculating it. The number of operations this code will do is around $K \cdot 84 \cdot 210 \cdot 84$.

Problem Tutorial: "Mount Rushmore and Birthdays"

Considering a year with N days, the probability that i people in a room have all different birthdays is $P(i) = \frac{N}{N} \cdot \frac{N-1}{N} \cdots \frac{N-i+1}{N}$. All needed is to determine the smallest i such that $1 - P(i) \geq 0.5$, that is, $P(i) \leq 0.5$.

Problem Tutorial: "The Knapsack problem"

A (usually not so smart) way to solve the knapsack problem with repetitions is to use dynamic programming the following way: $f(S)$ is the the best way to solve the problem with at most S weight. The recurrence has three possible cases:

1. You choose no items, in this case $f(S) = 0$.
2. You choose 1 item, in this case the answer is the largest value of a single object with weight at most S.
3. You choose 2 or more items, in this case, the chosen set of items can be partitioned in two non-empty sets of size not exceeding X and $S - X$, and then $f(S) = f(X) + f(S - X)$.

The problem with case 3 is that we do not know the size of the set X, so the straightforward approach is to calculate $f(S) = \min_{1 \le X < S}(f(X) + f(S - X))$. Notice that this will work for **any** correct partitioning of the set of size S, and in this problem since $w_i \le 10^3$ we can assume $|X - (S - X)| \le 10^3$ and therefore $\frac{S}{2} - 500 \le X \le \frac{S}{2} + 500$.

We want to calculate only $f(S)$, and for that we need to have calculated the interval $[\frac{S}{2} - W, \frac{S}{2} + W]$, where $W = 500$. To calculate that interval we need to have the interval $[\frac{S}{4} - \frac{3W}{2}, \frac{S}{4} + \frac{3W}{2}]$ and so on. It can be proven that the constant that multiplies W will never exceed 2 (exercise, it is easy :P), so the "level" K will be no larger than $[\frac{S}{2^K} - 1000, \frac{S}{2^K} + 1000]$.

It is clear that we need to determine only $\approx \log_2(S)$ levels, so there are only about $4 \cdot W \cdot \log_2(S)$ states of the dp that need to be calculated, and each of them takes time $2 \cdot W$. So this problem can be solved in time $\mathcal{O}(W^2 \log(S))$.