Problem Tutorial: "Attack Order"

If $n = 2$, the answer is always "`Yes`", because each minion buffs the other one, and we know their exact attacks before the fight.

If $n > 2$, the minions should be sorted in non-increasing order of their initial attacks a_i, and minions with equal a_i should be sorted in non-decreasing order of b_i.

After sorting, if $a_i + \sum_{j \neq i} b_j > a_{i-1}$ for some i, the answer is "`No`", because it could happen that every minion buffs minion i, and minion i buffs a minion other than minion $i - 1$. Otherwise, the answer is "`Yes`".

Problem Tutorial: "Browsing the Collection"

Since the current item always satisfies all the conditions in the set, the current state can always be described with two integers $(i, mask)$, where i is the current item and $mask$ is the bitmask of the active filtering conditions.

We can casework on the starting item and use BFS on these states. The "left" and "right" transitions can be easily precomputed, the "remove" transition is simple, and the only non-trivial transition is "add". You could use bitset to optimize the transitions, or optimize your code in other ways, and get accepted if you try hard enough.

However, the intended solution is to look at the process backwards, and make all transitions in reverse. Instead of caseworking on the starting item, let's casework on the finishing item and run BFS. The "left" and "right" transitions have not changed. The "add" transition is simple — we add a filter and don't move the pointer. The "remove" transition is now the trickiest.

Let $prev(i, mask)$ be the closest item to the left of item i with the same values as item i in parameters $mask$. Suppose we are "removing" the filter on parameter j. Then we can make transitions to the following states:

- $(i, mask \setminus j)$;
- $(prev(i, mask \setminus j), mask \setminus j)$;
- $(prev(prev(i, mask \setminus j), mask \setminus j), mask \setminus j)$;
- ...

and so on, until we hit $prev(i, mask)$. We can check that in each of the states above, normally, if we add a filter on parameter j with the value $a_{i,j}$, the pointer will move to item i.

It seems that we can still have $O(n)$ transitions for each state, so how does this help? Let's maintain a disjoint-set union for each $mask$. In this DSU, item i will be the representative of its own set if and only if $(i, mask)$ is not visited by the BFS yet. Whenever a state $(i, mask)$ is visited, we make a link from item i to item $prev(i, mask)$. This way, we can only traverse states that we are going to put into the queue right now, spending $O(\log n)$ time per each. The overall complexity is $O(n^2 \cdot 2^m \cdot m \cdot \log n)$.

Problem Tutorial: "Casual Dancers"

Observation: $\max(a, b, c) - \min(a, b, c) = \frac{1}{2}(|a - b| + |a - c| + |b - c|)$.

Due to the linearity of expectation, to find the expected stretch, it's enough to find the expected values of distances between every pair of points.

For two points, let their coordinates be a and b, and consider the (signed) difference $d = a - b$. How does it change after one second?

- With probability $\frac{1}{3}$, d doesn't change (because the third point was chosen).

- With probability $\frac{p}{3} + \frac{1-p}{3} = \frac{1}{3}$, d increases by 1 (with probability $\frac{p}{3}$, a increases by 1, and with probability $\frac{1-p}{3}$, b decreases by 1).
- With probability $\frac{1-p}{3} + \frac{p}{3} = \frac{1}{3}$, d decreases by 1.

Let $P(x) = \frac{1}{3}(1 + x + x^2)$, and consider $P^k(x)$. We can see that the i-th coefficient of $P^k(x)$ is the probability that d increases by exactly $i - k$ after k seconds. Thus, we can easily find the expected value of $|d|$ after k seconds.

If we use binary exponentiation and FFT, the overall time complexity is $O(k \log k)$.

An interesting observation is that the answer does not depend on p.

Problem Tutorial: "Diameter Two"

Let $k > 0$. Nodes from 1 to k must all be connected to the same node, otherwise the diameter will exceed 2. Let this node be $k+1$. Again, since the distance from node 1 to any node must not exceed 2, any node from $k+2$ to n must be connected to node $k+1$. Thus, node $k+1$ has degree $n-1$. What remains is to add extra edges between nodes from $k+2$ to n to increase their degrees. We can just connect $k+2$ and $k+3$, $k+4$ and $k+5$, and so on, and if node n is left unpaired, connect it with $n-1$.

This construction does not work when $k = n$ (obviously it's impossible to build the network then, since $n > 2$) and when $k = n - 2$ (node n can only be connected to node $n-1$, and there is no way to increase node n's degree; the answer is -1 in this case as well).

Let $k = 0$. It's tempting to try the same: let node 1 have degree $n-1$, and connect the remaining nodes in pairs. Let's try to prove this.

Let $n = 2t+1$ be odd (the case of even n is handled similarly). Then our construction method from above forms t triangles and uses exactly $3t$ edges. Now let's try to build a network with a smaller number of edges, without nodes of degree $n-1$.

Suppose there are no nodes of degree 2. Then all nodes have degree at least 3, and the number of edges is at least $\frac{3n}{2} = 3t + \frac{3}{2}$. This case is not interesting, because we're trying to use strictly less than $3t$ edges.

Now suppose there is a node z of degree 2. Let its neighbors be x and y. Each of the remaining $n-3$ nodes in the network must be connected to either x or y. Split these remaining nodes into two groups based on this condition (if a node is connected to both x and y, send it to any group). Suppose there are a nodes connected to node x, and b nodes connected to node y (not including z), where $a + b = n - 3 = 2t - 2$.

As of now, we already have $2 + a + b = n - 1 = 2t$ edges. Moreover, since the remaining $2t-2$ nodes all have degree 1 but need degree at least 2, we need to create at least $\lceil \frac{2t-2}{2} \rceil = t - 1$ new edges. Thus, the smallest number of edges we will need is $3t - 1$, which is strictly smaller than $3t$. It means we can potentially make a better network than we previously had, but only if we connect the remaining $a + b$ nodes in pairs. But can we do this?

Since we're considering odd n, $a + b$ must be even. If $a \geq 2$ and $b \geq 2$, we can see that no matter how we connect the vertices, some pair of vertices will end up at a distance greater than 2. The same happens if $a \geq 1$ and $b \geq 3$. When $a = 0$, we've already lost because now we have an extra vertex of degree 1 which we can not afford. The only remaining case is $a = 1$ and $b = 1$, but this case is actually important! It turns out we can build a network on 5 nodes using only 5 edges (as opposed to 6 edges using the previous method) if we connect them in a cycle.

Similarly to the odd n case, if we consider the even n case, we arrive at two new constructions: $a = 0$ and $b = 1$ (a cycle on 4 nodes, with just 4 edges instead of 5), and $a = 1$ and $b = 2$ (a cycle on 5 nodes plus an extra node connected to two non-adjacent nodes of the cycle; this is a network on 6 nodes with just 7 edges instead of 8).

It follows that for $n > 6$, the initial construction is optimal.

Problem Tutorial: "Escaped from NEF"

How to solve the problem for a tree? Since there are no cycles, we have a directed acyclic graph. We can use dynamic programming to find the number of vertices reachable from each vertex i: $dp_i = 1 + \sum_{ij \in E} dp_j$.

It can be seen that every vertex will be counted towards dp_i at most once, and this DP is correct.

What changes if we have cycles? First of all, we can compress strongly connected components in G. Once we do that, each vertex i now has weight w_i, and the formula changes to $dp_i = w_i + \sum_{ij \in E} dp_j$.

However, there is another issue: some vertices now may be counted twice towards dp_i. That would only happen if there is an undirected cycle consisting of two directed paths $i \to j$. In this case, all vertices reachable from j will be counted twice, and we should subtract dp_j from dp_i.

We can identify all cycles in a cactus in linear time, check if they consist of two directed paths (the easiest way to do that is to calculate vertex outdegrees, and check if there's a single vertex of outdegree 2 and a single vertex of outdegree 0), and remember all pairs (i, j) for such cycles. After that we can just run the DP algorithm. The overall time complexity is $O(n + m)$.

Problem Tutorial: "First Occurrence"

Let's use the following recursive property of the Thue-Morse sequence: if we replace every `0` with `01` and every `1` with `10` in T simultaneously, we get T again.

Let's try to apply this property in reverse. Let's consider some small example. Suppose we want to find substring `010010` in T. Since its *predecessor* substring (with respect to the described property) also appears in T, let's split the characters into pairs. We have two ways to do that: `01 00 10` and `0 10 01 0` (depending on the parity of the index of the occurrence). In the first case, since one of the pairs `00` is neither `01` nor `10`, we can conclude that this split is invalid. In the second case, we can see that the string `0 10 01 0` can be generated from `1100`. Thus, if i is the index of the first occurrence of `1100` in T, we can see that the index of the first occurrence of `010010` in T is exactly $2i + 1$ (the $+1$ comes from the fact that the first "pair" in the split has just one digit).

Now, what happens if both parities result in valid splits? For that to happen, each character in the substring must be not equal to its neighbors, so the substring should look like `010101...` or `101010...`. However, in this case, the predecessor substrings will look like `0000...` or `1111...`, and there are no substrings with more than 2 equal digits in T. It follows that the case when both parities are valid is only possible when the length of the sought substring is at most 3.

Finally, what if the substring we are looking for is actually a substring of T: $t_{l..r}$? In this case, if this substring is long enough, we already know the only valid split! The predecessor substring is $t_{\lfloor l/2 \rfloor .. \lfloor r/2 \rfloor}$, and the parity shift is $l \bmod 2$.

Thus, the answer to the problem is $f(l, r) = f(\lfloor \frac{l}{2} \rfloor, \lfloor \frac{r}{2} \rfloor) + (l \bmod 2)$ if $r - l + 1 \geq 4$. By repeatedly applying this formula, we end up searching for a substring of length at most 3, which can be done naively. The time complexity of this solution is $O(\log r)$.

Problem Tutorial: "Gross LCS"

Consider a particular value of x. Let $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ be all pairs (i, j) such that $a_i + x = b_j$. Let's order these pairs in such a way that $i_t \leq i_{t+1}$, and if $i_t = i_{t+1}$ then $j_t > j_{t+1}$. Then $\mathrm{LCS}(A + x, B) = \mathrm{LIS}(\langle j_1, j_2, \ldots, j_k \rangle)$, where LIS denotes the length of the longest increasing subsequence (basically, both $\mathrm{LCS}(A + x, b)$ and $\mathrm{LIS}(\langle j_1, j_2, \ldots, j_k \rangle)$ choose as many pairs (i, j) as possible in such a way that both i's and j's are strictly increasing).

Note that each pair (i, j) can only be used for one x, namely, $x = b_j - a_i$. A solution that uses $O(nm \log(nm))$ time and $O(nm)$ memory looks as follows:

- Sort all tuples $(x = b_j - a_i, i, -j)$ in increasing order.

- For each block of tuples with equal x, find the LIS of the values of j and add it to the answer.

To get to $O(n+m)$ memory, we need two ideas.

The first idea is how to generate tuples $(x, i, -j)$ in increasing order without storing them all in memory. We will generate these tuples one by one.

Let $p_1, p_2, \ldots, p_m$ be a permutation of $1, 2, \ldots, m$ such that $b_{p_j} \le b_{p_{j+1}}$, and if $b_{p_j} = b_{p_{j+1}}$, then $p_j > p_{j+1}$. Note that tuples $(b_{p_1} - a_i, i, -p_1), \ldots, (b_{p_m} - a_i, i, -p_m)$ are sorted in this exact order.

Consider a priority queue containing the smallest unused tuple for each i. Initially, this priority queue contains the tuple $(b_{p_1} - a_i, i, -p_1)$ for each i. We will repeat the following process:

- Pop the smallest tuple from the priority queue, let this tuple be $(b_{p_j} - a_i, i, -p_j)$.
- Add this tuple to the global sorted list of tuples.
- If $j < m$, add $(b_{p_{j+1}} - a_i, i, -p_{j+1})$ to the priority queue.

Finally, to avoid using $O(nm)$ memory, instead of adding tuples to the list, let's process them on the go. Maintain a virtual integer sequence, initially empty. Once a tuple emerges, if its x is different from the previous tuple's x, add the LIS of the sequence to the answer and "clear" the sequence. Then, "append" the tuple's value of j to the sequence (without actually appending).

How to find the LIS of an integer sequence "online", without storing the whole sequence? Both usual methods for finding LIS can be adapted:

- One can use DP with Binary Indexed/Fenwick/Segment tree. Whenever an integer j appears, find the maximum value M on the prefix $0..j-1$ in your data structure, and update element j with $M+1$. The data structure is of size $O(m)$, and we can either use "timers" to clear it (save the last time of modification for each cell, and the last time the data structure was cleared), or save the pointers to all modified cells and clear them when required.
- The other approach, using binary search, works even better. The vector in which we do the binary search always has length at most m, and it can even be cleared explicitly.

Problem Tutorial: "Hundred Thousand Points"

Basically, we want to know the n-dimensional volume formed by all points $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ leading to non-intersecting angle arrangements.

Note that the angles from every point except $(1, 0)$ and $(n, 0)$ must be fully contained inside either the upper or the lower half-plane. First, suppose that angles from $(1, 0)$ and $(n, 0)$ also satisfy this property.

Suppose that we have decided, for each angle, what half-plane it goes to. Suppose that angles of $b_1, b_2, \ldots, b_k$ degrees go to the upper half-plane. It can be shown that the k-dimensional volume of non-intersecting arrangements is $(180 - b_1 - b_2 - \ldots - b_k)^k / k!$. For the remaining $n-k$ angles of $c_1, c_2, \ldots, c_{n-k}$ going to the lower half-plane, the formula for the $(n-k)$-dimensional volume is similar: $(180 - c_1 - c_2 - \ldots - c_{n-k})^{n-k} / (n-k)!$. The overall n-dimensional volume is the product of these two values.

Now, suppose that the angle from point $(1, 0)$ contains point $(0, 0)$. Let's say that out of the remaining $n-1$ angles, k angles of $b_1, b_2, \ldots, b_k$ degrees go to the upper half-plane, and the remaining $n-k-1$ angles of $c_1, c_2, \ldots, c_{n-k-1}$ degrees go to the lower half-plane. There is a continuous segment of the $(1, 0)$-verticed angle positions, each position corresponds to a real value $x \in [0; a_1]$. The formula for the $(n-1)$-dimensional volume for a fixed value of x looks as follows:

$$(180 - x - b_1 - b_2 - \ldots - b_k)^k \cdot (180 - (a_1 - x) - c_1 - c_2 - \ldots - c_{n-k-1})^{n-k-1} / (k!(n-k-1)!).$$

Now we need to integrate this function over x from 0 to a_1 to get the n-dimensional volume. To do that, we can explicitly find the polynomial of x (of degree $n-1$), find its primitive, and substitute $x = a_1$.

The case when the angle from point $(n, 0)$ contains point $(n+1, 0)$ is symmetric.

The case when both the angle from point $(1, 0)$ goes left and the angle from point $(n, 0)$ goes right is trickier, it will result in a double integral over two variables $x \in [0; a_1]$ and $y \in [0; a_n]$. However, it's possible to substitute $z = x + y$, split the integration inteval into three parts, and integrate them separately. The full mathematical details are omitted here.

Finally, if we just choose for each angle whether it goes up or down, we will have $O(2^n)$ options. We can use a knapsack-like DP instead, since we are only interested in the number and the total degree value of the angles going up.

The time complexity of this solution is $O(M^4)$ where $M = 180$ is half the full angle, in degrees: we have $O(M^2)$ options for the pair (number of angles, their total degree), and building the polynomial for each such pair takes $O(M^2)$ time. FFT can improve the complexity to $O(M^3 \log M)$, but this was not required.

Problem Tutorial: "Implemented Incorrectly"

Suppose that for each step of the algorithm we decide whether $a_i < a_1$ or not. There are 2^{n-1} ways to decide that. For each way to decide, we can count the number of permutations corresponding to it, and add it to the answer if 1 doesn't end up at the front.

We can implement this with backtracking. Let $f(step, ptr, cnt, ways, seen[n])$ be the backtracking function with the following meaning of the arguments:

- $step$ is the current i of the algorithm from the problem statement;
- ptr is the pointer to the smallest element of the permutation seen so far;
- cnt is the number of seen elements;
- $ways$ is the number of permutations of seen elements leading to the current situation;
- $seen[]$ is the boolean array containing what elements of the permutation we have actually seen.

At the next step of the algorithm, a_{ptr} will be compared to $a_{(ptr+step) \bmod n}$. If $seen[(ptr + step) \bmod n]$ is `true`, we already know that a_{ptr} is smaller, so we just call $f(step+1, ptr, cnt, ways, seen[])$. Otherwise, we mark $seen[(ptr + step) \bmod n]$ with `true`, and then we call $f(step + 1, ptr, cnt + 1, ways \cdot cnt)$ (assuming a_{ptr} is smaller) and $f(step + 1, (ptr + step) \bmod n, cnt + 1, ways)$ (assuming a_{ptr} is bigger).

As we can already see, this backtracking works in $O(2^n)$. It turns out that because in some cases we don't branch, the actual number of backtracking calls for $n = 42$ is around 10^9, which makes this solution fast enough, at least for precomputing the answers.

Problem Tutorial: "Junk or Joy"

The given equation is equivalent to $(n-1)(n+1) = k \cdot p^m$. Note that $\mathrm{GCD}(n-1, n+1) = 1$ or 2.

First, consider the case $p > 2$. Then only one of $n-1$ and $n+1$ can be divisible by p. Thus, $n \pm 1 = x \cdot p^m$ and $n \mp 1 = \frac{k}{x}$ for some x which is an integer divisor of k. From these two equations, it follows that $p^m = \frac{\frac{k}{x} \pm 2}{x}$. We can try all divisors x of k, try both $+2$ and -2, and for every integer value of p^m we get, check if this number is actually a prime power.

When $p = 2$, things are not very different. It can happen that both $n+1$ and $n-1$ are divisible by 2, but it can't happen that both are divisible by 4. Thus, $n \pm 1 = x \cdot 2^{m-1}$ and $n \mp 1 = \frac{2k}{x}$. It follows that $2^{m-2} = \frac{\frac{k}{x} \pm 1}{x}$, which is similar to the formula from the previous paragraph, and the same algorithm applies.

Problem Tutorial: "Kilk Not"

Let's do a binary search on k, the maximum allowed length of a block of consecutive equal digits.

For a fixed value of k, let's find b_{min} and b_{max} — the smallest and the largest numbers of `1`'s in a string t obtained by replacing `?` in s and satisfying the maximum block condition.

These values can be found greedily, but the greedy algorithm is a bit tricky. We can use dynamic programming instead. Let $f_{min}(i, c)$ denote the smallest number of `1`'s in $t_{1..i}$ if $t_i = c$. Then, in general:

- $f_{min}(i, 0) = \min\limits_{j=i-k}^{i-1} f_{min}(j, 1)$;
- $f_{min}(i, 1) = \min\limits_{j=i-k}^{i-1} f_{min}(j, 0) + (i - j)$.

However, in some cases, since some characters are already known, the lower bound on j might be greater than $i - k$. In any case, this DP can be calculated in $O(n)$ using a queue supporting the global minimum query.

Suppose we have found b_{min} and b_{max}, can we replace `?`'s in s with exactly b ones and satisfy the maximum block condition too? If $b < b_{min}$ or $b > b_{max}$, the answer is obviously "no". Otherwise, if $b_{min} \le b \le b_{max}$, it can be proved that the answer is "yes".

How can we prove that? Let t_{min} and t_{max} be the strings achieving b_{min} and b_{max}, respectively. Then, due to monotonicity, there exists i such that $t = t_{min}[1..i] + t_{max}[i + 1..n]$ contains exactly b ones. We have to be a little careful though: if $t_{min}[i] = t_{max}[i + 1]$, we might get an unexpected block of equal digits. However, it turns out we can always choose such i that $t_{min}[i] \neq t_{max}[i + 1]$. The proof is left as an exercise.