Problem A. King of String Comparison

Author: Utkarsh Gupta
Solved by: 112/118
First to solve: *UBB_Zalau00*

Note that to compare two strings, we just have to know the first position where they differ, and which string at that position has a larger character.

Let's go from right to left, keeping the last met position where s and t are different — $last$, and a boolean variable $less$, which is $true$ if $s[last] < t[last]$ and $false$ otherwise. Initially there is no such position, so set $last$ to some undefined position (for example, -1).

When we are at position l, we update $last$, $less$ if $s[l] \neq t[l]$ first. After that, if $last$ is defined and $less$ is $true$, we add $n + 1 - last$ to the answer: for all pairs (l, r) with $r \geq last$, $s_l s_{l+1} s_{l+2} \dots s_r$ will be **lexicographically smaller** than $t_l t_{l+1} t_{l+2} \dots t_r$, and for smaller r they will be equal.

Total complexity is $O(n)$.

Problem B. New Queries On Segment Deluxe

Author: Alex Danilyuk
Solved by: 3/5
First to solve: *KNU_Duplee*

The problem quite clearly asks to implement a persistent segment tree with lazy propagation, except that the operations are a bit weird. For a segment tree to work, you need to merge answers from sons (no problem, just take minimum) and quickly change the answer for a vertex when applying the operation on the whole segment. And while it is not an issue for `+=`, for `:=` the answer can change quite unpredictably: you need to kinda forget about this row in a matrix and take minimum if we sum all the rows except this one...

After some considerations, it should be clear that we need to expand what is "the answer": we want to store the minimum not only for the sum of all rows but also for all subsets of rows. The details of the implementation are left as an exercise :)

Time and memory complexities are $O((n + q \log n)2^k)$.

Problem C. Werewolves

Author: Andrei Constantinescu
Solved by: 12/19
First to solve: *KhNURE_Energy is not over*

The majority color, if it exists, is unique, because otherwise, the total number of nodes with one of the majority colors would be greater than the total number of nodes in the subtree. Let's iterate through the n possible colors and calculate the number of subtrees with this color being a majority color.

Given a fixed candidate majority color, for all vertices set $s_v = +1$ if it is of this color, and $s_v = -1$ otherwise. The problem is now to find the number of subtrees whose sum of s values is strictly positive. This can be done with a standard $dp[v][sum]$, meaning the number of subtrees of T restricted to the subtree of v which contain node v and have their sum of s values equal to sum. It seems like this works in $O(n^4)$, but let's look at it more closely.

Let's say that the number of nodes with color c is k_c. Then, when we calculate the dp for this color, $sum \leq k_c$. Additionally, note that we only care about states with $sum > -k_c$, because we won't be able

to reach a positive *sum* from smaller values. Also $|sum| \le sz_v$, where sz_v is the size of the subtree of node v. Thus, we can only calculate dp for $|sum| \le \min(sz_v, k_c)$. It is well-known that such dp works in $O(n \cdot k_c)$. Therefore, the total time complexity is $\sum_c O(n \cdot k_c) = O\left(n \cdot \sum_c k_c\right) = O(n^2)$.

Problem D. Many LCS

Author: Anton Trygub and Alex Danilyuk
Solved by: 0/0
First to solve: -

As the problem is purely constructive and there are a lot of possible directions, we would not be surprised if there are solutions that are different from our solution in every regard. Having said that, we are not aware of any such solutions. There is no logical reasoning behind the general construction, we just tried a lot of things and stumbled on the one that works, although at least three people came up with this construction independently, so it doesn't seem like pure luck. Let's jump into the general construction.

$S = 0^{k_1}\,01\,0^{k_2}\,01\,\ldots\,0^{k_t}\,01$,

$T = (01)^{(n+t)}$,

where $n, t, k_1, k_2, \ldots, k_t$ — some non-negative integers (there might be more concise constructions with the same idea, I personally find this one the clearest). We will also require $k_1 + k_2 + \ldots + k_t \ge n$.

I state that it is possible to take all 1 from S and all 0 from T, which means that length of $LCS(S,T) = n + 2t$, and the only possibility to reach this is to be of the form $0^{a_1}\,01\,0^{a_2}\,01\,\ldots\,0^{a_t}\,01$, where $0 \le a_i \le k_i$, $\sum_{i=1}^{t} a_i = n$.

Now we want to choose $n, t, k_1, k_2, \ldots, k_t$ such that the number of ways to choose $a_1, a_2, \ldots, a_t$ is exactly K.

Let's first understand what is the number of ways to choose $a_1, a_2, \ldots, a_t$ for the given $n, t, k_1, k_2, \ldots, k_t$. It is a standard combinatorics problem:

If there were no limitations $a_i \le k_i$, the answer would be equal to $\binom{n+t-1}{t-1}$. Let's now apply inclusion-exclusion principle, fix the set I of $a_i > k_i$ (that is, the condition $a_i \le k_i$ is violated). The answer will be equal to $\sum_{I \in 2^{[1..t]}} (-1)^{|I|} \binom{n-\sum_{i \in I}(k_i+1)+t-1}{t-1}$.

This doesn't look good, because we can control only $t+2$ parameters, while the number of summands is 2^t. Let's make most of them be equal to zeroes! Let's choose k_i close to n, more formally, we want $k_i + k_j \ge n$, so that we can't violate two or more conditions at once. This way the answer will be $\binom{n+t-1}{t-1} - \sum_{i=1}^{t} \binom{n-k_i-1+t-1}{t-1}$.

With the answer in this form the idea for the construction is clear: fix t, choose some n such that $\binom{n+t-1}{t-1} \ge K$, represent $\binom{n+t-1}{t-1} - K$ as the sum of t binomial coefficients.

With $t = 3$ we are dealing with triangular numbers, and there is actually a theorem that states that every number is representable by a sum of at most three triangular numbers. Unfortunately, $t = 3$ means that we will need $n \approx \sqrt{2K}$ to get big enough value, and we can't afford that.

$t = 4$ would resolve this issue, we will be good with $n \approx \sqrt[3]{6K}$. Now we are dealing with tetrahedral numbers and... they are *good enough*. There is an unproven conjecture that says that every number is representable by a sum of at most *five* tetrahedral numbers. Fortunately for us, this conjecture is checked for all numbers up to 10^{10}, moreover, there are only 241 (known) numbers that actually require five summands, the largest being 343 867. In case we are unfortunate to encounter such a test, we can just choose a larger n.

The (almost) last issue is that we are not satisfied with just knowing that the representation in a form of such sum exists, we actually need to construct one. The largest value of n we might want to use is less than 1850, which means the numbers we want to represent in this form are bounded by

$\binom{1850}{2} + 343\,867 < 2\,100\,000$. Thus we can use only the first 240 tetrahedral numbers (as $\binom{240}{3} > 2\,100\,000$). We can implement DP in $240 \cdot 2\,100\,000$ to construct the representation, and if for some reason it is slow, use bitsets to speed it up.

The actual last issue is that we have to ensure that $k_i + k_j \geq n$ actually holds. It will be true automatically for large values of K, because $\binom{n-k_i-1+t-1}{t-1} \leq 2\binom{n+t-1}{t-2}$, and that means that for large enough n $n - k_i < n/2$. You might need to use some simpler approach for small values of K, for example, the solution above with $t = 2$ or $t = 3$ ($t = 2$ should be enough).

Problem E. Replace Sort

Author: Daniel Posdarascu
Solved by: 12/20
First to solve: *LNU Bulldogs*

Solution 1:

Sort elements in B because the order does not matter.

$O(N^2)$ solution:

$dpA[i]$ = the minimum cost in order to solve the first i elements in A and the element on position i is $A[i]$ (the initial value) $dpB[i][j]$ = the minimum cost in order to solve the first i elements in A and the element on position i is $B[j]$ (it was replaced)

For $dpA[i]$ we have two cases:

1. The previous element has also kept his initial value. This can be applied only if $A[i-1] < A[i]$.
2. The previous element was replaced with something from B, so any $B[j] < A[i]$.

Therefore, $dpA[i] = min(dpA[i-1]$ (if it's the case), $dpB[i][j])$ for all j with $B[j] < A[i]$. This can easily be done with partial min.

For $dpB[i][j]$ we have two cases:

1. The previous element kept its initial value. This can be applied only if $A[i-1] < B[j]$.
2. The previous element was also replaced by another element in B. The key observation is that if position i in A was replaced by element j in B, then position $i-1$ will be replaced by element $j-1$ in B (or there is no reason to take any other element from B).

Therefore, $dpB[i][j] = min(dpA[i-1] + 1$ (if it's the case), $dp[i-1][j-1] + 1)$.

$O(N \cdot \log N)$ optimization:

We will iterate only through the first dimension (iterate only i) and we will update all the values of $dpB[i][j]$ using a data structure. We can see that we only need minimum on prefix, update with minimum on suffix and update +1 on elements. This can be done using lazy update on a segment tree for example.

Solution 2:

We want to replace as few elements of A as possible, which is the same as not touching as many elements of A as possible. Let's mentally fix the elements of A we won't touch, can we achieve that? Obviously, the untouched elements should be increasing left-to-right, but also we should have enough elements in B to fill in the gaps. The good news is since the untouched elements are increasing, the elements of B we can use to fill in the gaps are different for each gap, so we can check if we can fill in each gap independently.

That leads to a polynomial solution: we don't need to choose all the elements we won't touch, we can choose them one-by-one left-to-right, each time checking that we have enough elements in B to fill in the gap. Do `DP[v]` — maximum number of untouched elements if the last one is on position `v`. To make a transition, iterate over the next untouched element `u`, check that $A_v < A_u$ and that we have at least

$(u - v - 1)$ elements in B between A_v and A_u. The obvious implementation is $O(N^2M)$, we need to do better than that. (For convenience, add elements $A_0 = -\infty$ and $A_{N+1} = \infty$, so that we can always take them)

Can we do the transition faster? Well, yes, we only need to check if the number of elements in B is enough, we can do that with a couple of `lower_bound`s if we sort B in advance. Moreover, we can right away define p_v — position of first element in B which is greater than A_v, then the transition is possible if and only if $A_v < A_u$ and $u - v - 1 \leq p_u - p_v$. Unless $u = v + 1$, which we can check separately, the second condition implies the first, so we only need to check the second one. We can rewrite it as $p_v - v - 1 \leq p_u - u$, which is good, because the left side depends only on v and the right side depends only on u. Therefore, to calculate `DP[u]`, we only need to take maximum over `DP[v]` for v with $p_v - v$ less than some value.

It can be done with Segment Tree, but there is a way that requires only std::set. Smaller values of $p_v - v$ are better — higher chances to be less than some value. Therefore, if there are two states v and u with $p_v - v \leq p_u - u$ and $DP[v] \geq DP[u]$, we don't need u, we can just forget about it. So, if we only store the values which are useful, a higher value of $p_v - v$ will mean a higher value of $DP[v]$. Let's store the pairs $(p_v - v, DP[v])$ in std::set, they will be sorted by the first parameter, which also means sorted by the second. When we insert the new value, erase all the old ones that are worse. To calculate the new DP value you only need one `lower_bound`. Since we can insert each element at most once, we can erase it at most once, thus the amortized complexity is $O(N \log N)$.

Accounting for the sorting of B and `lower_bound`s needed to calculate p_v, total complexity is $O((N + M)(\log N + \log M))$.

Problem F. to Pay Respects

Author: Daniel Posdarascu
Solved by: 77/99
First to solve: *KhNU_ OtVinta*

Observations:

If we decide to cast a spell that will only add a poison stack on the enemy, do it as soon as possible. If we cast spells in X such cases, it will be on the first X moments that do not have a regeneration.

If we decide to cast a spell that will both add a poison stack and remove a regeneration one , again, do it as soon as possible when the regeneration is added. If we cast spells in Y such cases, it will be on the first Y moments that have a regeneration.

Obviously, we do prefer to cast a spell in case 2 such that we will eliminate a regeneration while we add a poison, but some of those regenerations may be placed too late in the timeline, so it might not be optimal. In conclusion, the spells that we cast form a prefix in the timeline (first Z spells at moments from 1 to Z), plus some spells added at moments where regenerations are added.

Solution 1: compute for each moment x from 1 to N the cost of improvement if you cast a spell at that time. If the moment has no regeneration spell added, then the cost of improvement is $(N - x + 1) \cdot P$ because you only add a poison stack. If the moment does have a regeneration spell added at that time, the cost of improvement is $(N - x + 1) \cdot (P + R)$ because you add a poison stack and also remove a regeneration one. Sort those N options and select the best K.

Solution 2: initially consider the full prefix of K spells dealt at moments from 1 to K and then always take the last spell in case 1 and see if it's better to change it with the first spell from case 2. This solution works in time proportional to the number of regeneration spells which is bounded by $O(N)$. This solution works for even higher constraints.

Problem G. Max Pair Matching

Author: Anton Trygub
Solved by: 52/79
First to solve: *KhNURE_Energy is not over*

Solution 1. Let's assume that $a_i \le b_i$ in each pair.

Clearly, from each of the $2n$ pairs, we have to choose one number, with a plus or minus sign, so that exactly n numbers are chosen with a plus and exactly n with a minus, and we have to maximize the sum of these chosen numbers (with corresponding signs). That means, we have to choose n pairs from which we will choose b_i, and n pairs from which we will choose $-a_i$.

Now, let's consider $S = \sum_{i=1}^{2n} b_i$. Each pair from which we choose $-a_i$ instead of b_i, decreases this sum by $a_i + b_i$. So, it's optimal to choose $-a_i$ from the n pairs with the smallest sum $a_i + b_i$.

The overall algorithm would be to first make sure that $a_i \le b_i$, to sort by $a_i + b_i$, and take $-a_i$ from the first n pairs, and b_i from the last n.

Solution 2. Let's draw a point $c_i = \frac{a_i+b_i}{2}$ for each pair. Then $w_{ij} = \frac{|a_i-b_i|}{2} + \frac{|a_j-b_j|}{2} + |c_i - c_j|$.

Then we just have to find the partition into pairs for which the sum of $|c_i - c_j|$ over pairs (i, j) is the largest possible, and this is a well-known problem: just put match largest n points with smallest n points.

Problem H. Colourful Permutation Sorting

Author: Alex Danilyuk
Solved by: 1/3
First to solve: *KNU_0_GB_RAM*

Let's show that it only makes sense to do shuffle for each colour at most once. Let's say that elements of permutation are written on cards, and those cards lie in numbered boxes, each box has the colour of the corresponding position. When we swap two elements, we will swap the cards and record the ids of boxes they were in. When we do the shuffle for a specific colour, we will shuffle the boxes themselves. In such an interpretation swaps and shuffles are totally independent, thus we can do all the swaps first, and then all the shuffles, now it's obvious that we don't need to shuffle the same colour twice.

If we don't use shuffles, we have to make $(n - cyc)$ swaps, where cyc is the number of cycles in the permutation. When we shuffle the colour, we basically say that all the positions of this colour are the same. In terms of the graph, we merge all the vertices of this colour.

Let's try all 2^k ways to choose which colours we will shuffle. Construct the graph of the permutation, merge the vertices of the chosen colours. The cost will be `[cost for shuffles]` $+S(n - cyc)$, where cyc is the maximum number of cycles we can split the edges of the graphs into. We can contract the vertices with $indegree = outdegree = 1$, so we will only have the mega-vertices corresponding to chosen colours.

Thus we have reduced our problem to the problem "Given a directed graph with at most 5 vertices and at most n edges, for each vertex $indegree = outdegree$, what is the maximum number of the cycles we can split the edges into?" We are aware of 2 working methods for this problem:

Common observations:

We can remove cycles one by one, the property $indegree = outdegree$ will remain. But the order of removal can be important.

It is always optimal to take cycles of length 1. It is also always optimal to take cycles of length 2. Suppose the opposite, then in optimal solution they are in different cycles. Let's take these 2 edges out, make a cycle out of them, and also make a cycle out of two remaining chains. It is also 2 cycles, so it's not worse.

From here we assume that we don't have self-loops and edges in the opposite directions (if they appear during the algorithm, we remove them).

Method 1.

If there is a vertex from which there are edges to only one other vertex then all the cycles coming through this vertex will have to go through that edge, so we can contract this vertex, by glueing incoming edges with the only outcoming.

If we have at most $k = 4$ vertices in the graph, we can always find such a vertex, thus we have a $O(k^3)$ solution.

For $k = 5$ vertices it is possible that each vertex has 2 incoming and 2 outcoming edges. Here we can remember that the total number of edges is at most n, and for one edge we can bruteforce all the possibilities where cycles that go through it will go next. Then we can remove this edge and finish as before, as now there will always be a vertex we can contract.

Method 2.

It turns out that for $k \leq 5$ it is not only optimal to take cycles of length 1 and 2, but also of length 3, 4 and 5 in this order.

The reasoning is that when we are talking about cycles of length L, there are already no cycles of smaller lengths in the graph. Let's recall our proof that we can take cycles of length 2: we supposed the opposite, took the optimal solution, took the cycles that contain the edges we want, took those edges out to form a cycle, and formed a cycle out of the chains that were remaining. We get only 2 cycles this way, which can be less than L. But let's say that $L = 3$ and the 3 edges were all in different cycles in the optimal solution, otherwise we are good. Those cycles should all have a length of at least 3 because there are no shorter cycles in the graph now. Thus the length of our "leftover"cycle is at least 6, which is greater than the number of vertices, thus by pigeonhole principle there will be a repeated vertex, thus we can split this cycle in at least 2. This is the main idea of the proof, the casework for $L = 4$ is left to the reader.

Technically the second method is $O(n)$, because it just doesn't work for $k > 5$, but in the limitations of this problem the first method is faster, but both should pass without trouble.

Problem I. Flood Fill

Author: Lucian Bicsi
Solved by: 11/13
First to solve: *KhNURE_Energy is not over*

Let's look at each initial connected component in picture A. For each component, we can either leave the colour the same, or change it. Let's choose for each component what we'll do. Which situations are impossible? Well, it's not hard to see that two neighbouring components cannot both change colour, because once one of them changes colour, they will have the same colour, and we can't change that in the future.

Let's build a graph: the vertices are connected components and they are connected if they have a common border. Based on previous observation, the set of components in which you want to change the colour should form an independent set in this graph. And it's clearly possible for any independent set: just use the operation on the components you want, since they are not neighbours they don't affect each other (so the sample explanation is a red herring actually, you never have to apply the operation on the same component more than once).

For each component we can calculate the score if we change the colour in it and if we don't, now we want to find weighted maximum independent set in the graph we constructed. Our graph is bipartite since two neighbouring components have different colours. It's a standard problem, which is reduced to max flow.

Problem J. ABC Legacy

Author: Anton Trygub
Solved by: 67/93
First to solve: *KNU_Duplee*

Suppose that we will form X subsequences "AB", Y subsequences "AC", Z subsequences "BC", and let cnt_A, cnt_B, cnt_C be the numbers of times "A", "B", "C" appear in S, correspondingly. Then $cnt_A = X + Y$, $cnt_B = X + Z$, $cnt_C = Y + Z$, so we get $X = \frac{cnt_A + cnt_B - cnt_C}{2}$, $Y = \frac{cnt_A - cnt_B + cnt_C}{2}$, $Z = \frac{-cnt_A + cnt_B + cnt_C}{2}$. If these values are negative/aren't integers, there is clearly no way to split the string in the desired way.

Otherwise, we found our X, Y, Z. Look at the occurences of "B": it's going to appear X times in "AB", as the second letter, and Z times in "BC", as the first letter. Clearly, it's optimal to use first Z occurrences of "B" in subsequences "BC", and last X in subsequences "AB". Also, we want to take as "right" "A"s as possible in "AB" and as "left" "C" as possible in "BC", as we will have to form subsequences "AC" after.

So, for each of the first Z occurrences of "B", we will match it with the first yet unused occurrence of "C" to the right of it, if there exists any, and for each of the last X occurrences of "B", we will match it with the last yet unused occurrence of "A" to the left of it, if there exists any. As for remaining characters, we just have to check if the given string of Y "A"s and Y "C" can be split into subsequences "AC". If for some i the i-th occurrence of "A" goes after the i-th occurrence of "C", there is no solution, otherwise match i-th occurrence of "A" with the i-th occurrence of "C" for each i.

Problem K. Amazing Tree

Author: Andrei Constantinescu
Solved by: 17/32
First to solve: *LNU Bulldogs*

As always, when we want to find the lexicographically minimal order of something, we should do it greedily in a left-to-right fashion. It's clear that the first element will be one of the leaves, and for any leaf we can construct a post-order with this leaf as the first element. Let v be the leaf with the minimum id, and u be its only neighbour. To construct a post-order with v as the first element, we have to go from the (yet unknown) root straight to v, and the last vertex we will visit before v is u (v itself can't be the root). After visiting v we will return to u, and there can be two possibilities:

- u itself is the root. Then we have to visit all other neighbours of u first, and finish in u. It's clear that we need to visit the neighbours in increasing order of the minimum leaf in them.
- u is not the root. Then we have to visit all neighbours of u, except v and the one leading to the root, then write down u in the post-order, and then go up.

Since we can choose where the root will be, we can reformulate the greedy as follows:

- We have to visit all neighbours of u, except v and the one with the largest min leaf.
- Then, either traverse the remaining subtree or put u in the order and continue in the remaining neighbour.

We did it only on the first step, but it is not hard to see that it works like that in the future also: we have arrived at u from v, that's all that is important.

The implementation can be made much easier if you root the tree in the min leaf, because now you will always go down the tree.

Problem L. Jason ABC

Author: Roman Bilyi
Solved by: 41/52
First to solve: *KNU_0_GB_RAM*

Let's show that 2 operations are always enough:

Let p be the minimum prefix length that has exactly n same characters. Without loss of generality, it's character `A`. So there are n characters `A`, $b_p < n$ characters `B` and $c_p < n$ characters `C` in prefix of length p. Now we can replace segment $[p+1, p+n-b_p]$ by `B` and segment $[p+n-b_p+1, 3 \cdot n]$ by `C`. After that there are exactly n characters `A`, `B` and `C`.

We should still check if it's possible to achieve the goal with 0 or 1 operation.

Zero operations are possible only if there are the same number of every character initially and we could check that.

Let's check if it's possible to use only one operation. Iterate over character which is used in this operation. Let this character be `A`. Let b be the number of `B` and c be the number of `C` in S. We need to find such segment $[l, r]$ that there are exactly n characters `B` and n characters `C` among $S[1..l-1] + S[r+1..3n]$, where $+$ means concatenation of the strings. This means that there should be $b-n$ characters `B` and $c-n$ characters `C` in $S[l..r]$. Let's find values b_i and c_i — number of characters `B` and `C` respectively in prefix of length i. Now segment $[l, r]$ is good iff $b_r - b_{l-1} = b-n$ and $c_r - c_{l-1} = c-n$. We can find if such segment exists using two pointers or set.

Complexity: $\mathcal{O}(n)$.

Problem M. Counting Phenomenal Arrays

Author: Anton Trygub
Solved by: 8/14
First to solve: *UAIC O(n^e)*

For an array $b_1, b_2, \ldots, b_k$ of integers ≥ 2, define its **imbalance** as $b_1 + b_2 + \ldots + b_k - b_1 b_2 \ldots b_k$.

Lemma 1: For any integers $b_1, b_2, \ldots, b_k, b_{k+1}$, all of which are at least 2, the imbalance of the array $[b_1, b_2, \ldots, b_{k+1}]$ is smaller or equal to the imbalance of the array $[b_1, b_2, \ldots, b_k]$.

Proof: We have to show that $b_1 + b_2 + \ldots + b_k + b_{k+1} - b_1 b_2 \ldots b_k b_{k+1} \leq b_1 + b_2 + \ldots + b_k - b_1 b_2 \ldots b_k$, or, equivalently, that $b_{k+1}(b_1 b_2 \ldots b_k - 1) \geq b_1 b_2 \ldots b_k$. This follows as $b_{k+1}(b_1 b_2 \ldots b_k - 1) \geq 2(b_1 b_2 \ldots b_k - 1) = b_1 b_2 \ldots b_k + (b_1 b_2 \ldots b_k - 2) \geq b_1 b_2 \ldots b_k$.

Lemma 2: All elements of any phenomenal array of size n don't exceed n.

Proof: If there is only one element not equal to 1, clearly the sum will exceed the product. So, there are at least two such elements, consider two largest of them — x and y. Suppose that $x \geq n+1$, $y \geq 2$, then the balance of $[x, y]$ is $x + y - xy = x + y(1-x) \leq x + 2(1-x) = 2 - x \leq -(n-1)$, so we would have to add at least $n-1$ ones to it to make the balance 0 (adding elements larger than 1 decreases the balance even more, and adding 1 increases it by one), but we have only $n-2$ elements remaining.

Note that for $k = 1$, the balance is always 0. So, such balance is always nonpositive.

Consider now any phenomenal array of size n, and look at its elements which are not equal to 1 — let them be $b_1, b_2, \ldots, b_k$. It means that the remaining $n-k$ elements are 1s. This means that the balance of $[b_1, b_2, \ldots, b_k]$ has to be precisely $-(n-k)$. Moreover, each such array $b_1, b_2, \ldots, b_k$ corresponds to some precise number of ones that you have to add to it to make it phenomenal.

So in theory, what we would have to do is to find all arrays $b_1, b_2, \ldots, b_k$ of integers ≥ 2, to calculate their balance bal, and to add $\binom{k+(-bal)}{(-bal)}$ to the $f(k+(-bal))$ for each such array. But how would we consider all such arrays?

First, note that we don't have to consider arrays for which $k + (-bal) > n$. Second, note that we only care about the product and the sum of $b_1, b_2, \ldots, b_k$.

So, let's implement some recursive algorithm

```
recurse(int sum, int product, int size, int ways, int iter)
```

We will keep the sum of the currently chosen elements, their product, the number of chosen elements, and the number of ways to rearrange them, and we will start from `recurse(0, 1, 0, 1, 2)`. For every *iter* from 2 to n, we will try to add a few elements equal to *iter* to the current array, without breaking the condition $size + (-bal) \leq n$. One option is to not add any element equal to *iter* at all, from which we will recurse to

`recurse(sum, product, size, ways, iter+1)`.

Another option is to add *cnt* instances of number i, which would lead us to

`recurse(sum+iter·cnt, product·` $iter^{cnt}$`, size+cnt, ways·`$\binom{size+cnt}{cnt}$`, iter+1)`. Iterate *cnt* until the $bal + (size + cnt)$ exceeds n.

At each step of recursion, add $ways \cdot \binom{size+(-bal)}{size}$ to $f(size + (-bal))$.

This recursion runs very fast, as the product increases much faster than the sum does. It runs fast enough even for $n \leq 10^6$ (in under 2 seconds), so you shouldn't have experienced any TL issues if you tried any recursive approach.

Problem N. A-series

Author: Lucian Bicsi
Solved by: 109/112
First to solve: *KhNURE_Lacrimosa*

Let's look at the smallest size, AN. What if $b_N < a_N$? Clearly, the condition on the number of pieces of size AN is satisfied, and there is no need to cut any $A(N-1)$ piece to obtain additional ANs. As we can't transform smaller pieces into larger ones, we can just forget about size AN.

What if $b_N > a_N$? It means that we have to cut piece $A(N-1)$ at least $need = \lceil \frac{b_N - a_N}{2} \rceil$ times. And, similarly to the first paragraph, it's meaningless to make more such cuts.

So we have to add *need* to the answer and subtract *need* from a_{N-1}, denoting that we cut *need* of those (even if a_{N-1} becomes negative), and forget about size AN after that.

Let's continue doing this. For each i from $N-1$ to 0, find how many cuts of piece Ai do we need to make to get enough $A(i+1)$ pieces, add it to the answer, and adjust the number of pieces of paper Ai accordingly.

In the end, we will have to consider only the pieces of size $A0$. If a_0 will be at least as big as b_0, output the answer we were summing up. Otherwise, output -1.