Problem A. Archeologists

Author: Lucian Bicsi
Solved by: 1/11
First to solve: *KhNU_OtVinta*

This problem can be approached in many ways. Quite possibly the shortest description would be "just slope trick on prefix sums"; however, we will try to describe some various approaches.

Let's consider the following **minimum cost flow** problem, with vertices $\{0, 1, 2, \ldots, n\} \cup \{S, T\}$, and the following arcs:

- $i \to i + 1$ with capacity ∞ and cost 0, for $0 \le i < n$;
- $S \to i$ with capacity 1 and cost $p_1 + ... + p_i$, for $0 \le i \le n$;
- $i \to T$ with capacity 1 and cost $-(p_1 + ... + p_i)$, for $0 \le i \le n$.

Naturally, any valid flow corresponds to a valid solution, and the minimum cost flow corresponds with the answer (up to negation). One may solve the problem now by adapting the successive shortest paths algorithm on the particular network, by using, for example, a segment tree. The maximum flow is guaranteed to be exactly $n + 1$, due to the specifics of the network.

More easily, a minimum cost flow can be obtained by greedily going from right to left, matching the current position with some position to the right with maximum prefix sum, and then add the two residual edges that the current vertex exposed in the network. It can be shown that such a procedure never produces any negative cost cycles, therefore the flow is of minimum cost. Such solution can be easily implemented with a priority queue.

Another possible solution to the problem would be to start with the quadratic dynamic programming approach $dp(i, j)$ = maximum profit that you can obtain such that the depth at the i-th level is j, and optimize it by noticing that the function $f_i(j) = dp(i, j)$ is concave for all possible i. This fact can be proven (among other ways) by noticing that $dp(i, j)$ can also be expressed in terms of the minimum cost flow problem above, where j is the flow going along edge $(i \to i + 1)$.

Then, transitioning from $f_i(\cdot)$ to $f_{i+1}(\cdot)$ can be optimized using segment trees. This solution is, however, considerably more complex than using priority queues to simulate the minimum cost flow problem.

Bonus: An experienced coder may notice the strange connection between the priority queue solution and the implementation for a technique popularly known as "slope trick". Indeed, the two problems are tightly connected to each other (in a very non-trivial way).

One may see such a connection by formulating the current problem as a linear program:

$$\textbf{max}_x \sum_{i=1}^{n} p_i x_i \qquad \textbf{s.t.} \ \{ \ x_i - x_{i-1} \le 1 \quad x_{i+1} - x_i \le 1 \quad x_0 \le 0 \quad x \ge 0 \ \}$$

One may then dualize the linear program, and, after some substitutions, the exact linear program for the "slope trick" problem would show up:

$$\textbf{min}_{y,d} \sum_{i=1}^{n+1} d_i \qquad \textbf{s.t.} \ \{ \ y_i - s_i \le d_i \quad s_i - y_i \le d_i \quad y_i - y_{i+1} \le 0 \ \}$$

where we denode $s_i = p_1 + p_2 + \ldots + p_{i-1}$.

We leave the proof as an exercise for the (experienced) readers.

Problem B. Reverse Game

Author: Anton Trygub
Solved by: 61/87
First to solve: *Echipa Dulce*

Let's consider the number of inversions in the string s (defined as the number of pairs (i, j) such that $1 \le i < j \le n$, $s_i = \texttt{1}$, $s_j = \texttt{0}$). Suppose that currently it's I.

Note that reversing `10` decreases I by 1, and reversing any of `110`, `100`, `1010` decreases I by 2. Let's show, that if $I \ge 1$, it's possible to decrease it by 1 in 1 turn, and if $I \ge 2$, it's possible to decrease it by 2 in one turn.

Proof:

- Suppose that $I \ge 1$, there has to be a substring `10` in it, otherwise all zeros would go before all ones and there would have been no inversions. Then we can just reverse this `10`.
- Suppose that $I \ge 2$. We want to show that s must contain at least one of `110`, `100`, `1010`. Suppose that it doesn't. Then, if there is block of at least 2 consecutive zeros, it has to be in the beginning, and if there is a block of at least 2 consecutive ones, it has to be in the end. So, the string looks like: `00...0101...0101111...11`.

  Suppose that the string has a zeros in the beginning, b ones in the end and c repetitions of `10` between them. If $c = 0$, there would be no inversions at all, if $c = 1$, there would be exactly 1 inversion, so $c \ge 2$, and we can choose substring `1010`, which finishes the proof.

This shows that the game is equivalent to following:

Two players can substract 1 or 2 from I in their turns, the one who can't make a move loses. Who wins?

In this well-known game the first player wins iff I is not divisible by 3. So, to solve the problem, it's enough to find I in $O(n)$ and to determine if it's divisible by 3.

Problem C. 3-colorings

Author: Anton Trygub
Solved by: 0/0
First to solve: *N/A*

Let's start from creating nodes 1, 2, 3, and connecting them into triangle — we suppose from now on that the first node is colored in color 1, the second in color 2, third in color 3, and we need to construct a graph such that the rest is colorable in exactly k ways.

Let's create 8 nodes, connect 1st of them to 3, 2nd to 1, 3rd to 2, 4th to 3, and so on by cycle. This way, the 1st node can only be colored in one of colors $(1, 2)$, 2nd — in one of $(2, 3)$, 3rd — in one of $(3, 1)$, 4th - $(1, 2)$ and so on

Now, connect the i-th of these 8 nodes to the $(i + 1)$-st of them, in a chain. Note that this configuration has exactly 9 valid colorings: some prefix of nodes will be colored in the left color from the pair, and the remaining will be colored in the right color from the pair.

This is the end of the "template".

Now, for the i-th node let's create a_i nodes which are connected both to it and to the left color of this node (for example, for the first node, to node 1 of the starting triangle. Note that if the first k nodes are colored in their left colors, then there are exactly $2^{a_1+...+a_k}$ ways to color these new nodes. So, the total number of colorings will be $1 + 2^{a_1} + 2^{a_1+a_2} + ... + 2^{a_1+...+a_n}$.

Now the solution is easy: while k is even, divide it by 2, and add a leave hanging from node 1 (this just multiplies number of ways to color by 2). Then, let b be the number of bits which are on in k (not counting the smallest bit) — then create an edge from node $3 + b$ to the right node in its pair — this will leave only $b + 1$ ways to color. Then just add additional (highest bit) nodes, as shown above, and connect remaining nodes to 1 and 2.

Problem D. Disk Sort

Author: Lucian Bicsi
Solved by: 18/29
First to solve: *ETF/MATF Beograd - Seals*

The key approach to this problem is to try to progressively sort the colors one by one, in at most 6 operations per color. However, not all colors can immediately be solved in 6 operations. Nonetheless, we claim that there is always a color that can be sorted in 6 operations, also leaving us in a state with exactly one empty rod.

Let's define weight(i) as the sum of the depths of the three disks of color i (we consider the top row to be of depth 1, the middle row to be of depth 2, and the bottom row to be of depth 3). The sum of weights for all colors is exactly $6n$. This means that there must be some color c for which weight(c) ≤ 6. This is the color that we will choose.

There are, in fact, 6 possible cases for a configuration with weight at most 6 (we omit the situation where multiple disks are on the same rod, but they also have to be treated carefully). These cases are $[1, 1, 1]$, $[1, 1, 2]$, $[1, 1, 3]$, $[1, 2, 2]$, $[1, 2, 3]$, $[2, 2, 2]$.

Then, after doing some case analysis on paper, one might notice that all cases can be solved in at most 6 operations (including restoring to a situation where there is exactly one empty rod), except for the case $[2, 2, 2]$. However, it is not hard to see that, if we ignore all colors c for which the depths of the disks correspond to the case $[2, 2, 2]$, the sum of the remaining weights is still $6n'$ (where n' is the number of the yet unexcluded colors), therefore we can always find one of the other 5 favorable cases to solve at any point in time.

The rest of the solution is careful implementation of all cases. For example, the case $[1, 2, 2]$ can be solved in at most 6 steps as follows:

1. Move disk from rod 1 to empty rod;
2. Move disk from rod 2 to rod 1;
3. Move disk from rod 2 to empty rod;
4. Move disk from rod 3 to rod 2;
5. Move disk from rod 3 to empty rod;
6. Move disk from rod 3 to rod 2;

At the end of the six moves, the old empty rod is now sorted, and the new empty rod is rod 3. The solution for the other 4 cases are left as an exercise for the reader.

Note that at the end, all colors would be sorted and one empty rod will exist; however, if the empty rod is not $n+1$, we also have to move all disks from rod $n+1$ to the empty rod. The total number of operations cannot ever exceed $6n - 3$.

The complexity of such solution is $O(n^2)$. It can be optimized to $O(n)$, but the optimization was not required for these constraints.

Problem E. Divisible by 3

Author: Anton Trygub
Solved by: 73/96
First to solve: *KNU_ TheNextLevelPlay*

It is important to note that:

$$\sum_{1 \le i < j \le m} b_i b_j = \frac{(b_1 + b_2 + \ldots + b_m)^2 - (b_1^2 + b_2^2 + \ldots + b_m^2)}{2}$$

So, we just need to find the number of pairs (i, j), with $1 \le i \le j \le n$, for which $(a_i + a_{i+1} + \ldots + a_j)^2 - (a_i^2 + a_{i+1}^2 + \ldots + a_j^2)$ is divisible by 3. We will now describe a linear solution using prefix sums.

Let's denote:

$$S_i = a_1 + \ldots + a_i$$
$$T_i = a_1^2 + \ldots + a_i^2$$

Then, our task is to count the number of pairs (i, j), with $1 \le i \le j \le n$, for which $(S_j - S_{i-1})^2 \equiv (T_j - T_{i-1}) \bmod 3$ (here $S_0 = T_0 = 0$). Now note that we only care about all these numbers modulo 3, so in fact we can group them into at most $3 \times 3 = 9$ possible different values (x, y) with $0 \le x, y < 3$.

This gives the following $O(n)$ solution:

Let's keep an array $dp[3][3]$, where $dp[x][y]$ would be equal to the number of indices j such that $S_j \equiv x \bmod 3$ and $T_j \equiv y \bmod 3$ which we have processed so far. Initially $dp[0][0] = 1$.

Process indices from left to right, and if you are processing index i, add to answer $dp[x][y]$ for every pair (x, y) for which $(S_i - x)^2 \equiv (T_i - y) \bmod 3$, and update the corresponding entry in the dp table.

Problem F. Fence Job

Authors: Anton Trygub
Lucian Bicsi
Solved by: 10/12
First to solve: *RAF Penguins*

The set of operations might sound very daunting; let's try to find some properties about the resulting fence designs, to make it more tractable.

First of all, notice that if $h_i < h_j$ and $i < j$, then h_j can never appear on a position smaller than or equal to i. Similarly, if $h_j < h_k$ with $j < k$, then h_j can never appear on a position greater than or equal to k.

In particular, this tells us the rough structure of the resulting requences: they must be subsequences of (possibly repeating) numbers, but the unique values must appear in the same order as in the input. This is because for all $i < j$, if $h_i < h_j$ then h_j can never before any occurence of h_i, and, conversely, if $h_i > h_j$, then h_i can never appear after any occurence of h_i.

This ultimately yields a simple dynamic programming strategy: consider $dp[i][j]$ the number of output sequences b such that $b_i = h_j$. We also compute for each i two arrays:

- lf_i tells us the rightmost position $j < i$ such that $h_j < h_i$ (or 0);
- rt_i tells us the leftmost position $j > i$ such that $h_j < h_i$ (or $n + 1$).

Initially, $dp[0][0] = 1$. At each step in our $dp(i, j)$ computation, we assert that $lf_i < j < rt_i$ (otherwise, $dp[i][j] = 0$). After that, we have the following recurrence:

$$dp[i][j] = dp[i-1][0] + dp[i-1][1] + \ldots + dp[i-1][j]$$

which can be easily computed using prefix sums. Indeed, this approach works because all arrays b such that each unique value i in h corresponds to a (potentially empty) range $[l_i...r_i]$ of values in b and $lf_i \le l_i$ and $r_i \le rt_i$ can be obtained. The valid reconstruction is by cutting the planks in **decreasing** order of heights.

The complexity of this solution is $O(n^2)$.

Bonus: Solve the general version of the problem, where the plank lengths do not have to by distinct.

Problem G. Simple Hull

Author: Lucian Bicsi
Solved by: 0/1
First to solve: *N/A*

The first observation is that the area required in the statement somehow translates into the area "enclosed" by the polygonal line; however, this definition might be too informal to help us write a proper algorithm. A more standard way of thinking about this is in terms of **planar graphs**: in fact, the problem asks us to compute the **outer face** of the planar graph induced by the segments and their intersections.

Therefore, a straightforward solution is to build the planar graph, and "walk along" the edge of the outer face, in order to compute it. More specifically, after building the graph, you can start from the point with the minimum x-coordinate, and minimum y-coordinate for ties (this point is surely on the outer face) with the initial direction going up, and then start walking from vertex to vertex. When you are in some vertex, you should decide the next edge by prioritizing [turn left, go straight, turn right, turn around] in this order. This is also popularly known as the "left hand rule" in solving mazes.

Remember, though, that the vertices of the planar graph in question are intersections of segments of the original polygonal line, and there can be $\Omega(n^2)$ many. Therefore, the naive approach is too inefficient to pass the time limit.

However, one may cleverly guess that even though the planar graph is very large, the outer face can't be as big. Indeed, one may prove that the number of vertices and edges on the outer face is almost linear. It can be proven that the size of the outer face is bounded by $O(n\alpha(n))$, by noticing that the sequence of segments along the face is a **Davenport-Schinzel sequence of order 3** (consider the $2n$ oriented segments $p_i \to p_{i+1}$ and $p_{i+1} \to p_i$; consider the walk and enumerate the segment ids that you reach along the path; this enumeration can never contain an alternating subsequence $a...b...a...b...a$).

Therefore, the faster solution would be to simulate the "walking around" process faster. One may be able to do that in time $O(s \log s)$ using persistent data structures or $O(s \log^2 s)$ using 2D data structures, where s is the size of the outer face ($s \in O(n\alpha(n))$). These data structures should be able to preprocess a list of horizontal/vertical segments and answer queries of form:

"Given a point (x, y) and a direction up/left/right/down, what is the first segment that intersects the half-line extending in that direction?"

Before initializing the data structures, the overlapping horizontal and vertical segments have to be merged together. This can be done by sorting segments by one of their ends, and then using a stack.

One trick to make the implementation much easier is to consider extending the segments in both ends with a very small value. This has no effect on the answer, but it makes all the vertices of the underlying planar graph look like "+"-shaped intersections (with edges in all four directions), therefore a "left turn" is always available at any intersection. In practice, one achieve this effect without introducing floating point arithmetic by multiplying all coordinates by 3 and extending each segment by one unit to the left and to the right.

The final solution has a complexity of $O(n \log(n)\alpha(n))$. $O(n \log^2(n)\alpha(n))$ solutions should also pass, if implemented carefully.

Problem H. AND = OR

Author: Anton Trygub
Solved by: 2/4
First to solve: *RAF Penguins*

Firstly, let's understand how to solve the problem for a single query.

Let's sort the array for which we are solving the problem, let its elements be $[b_1, b_2, \ldots, b_k]$. Suppose that it's possible to split its elements into 2 groups A and B so that `OR` of the numbers from A = `AND` of the numbers from $B = x$. Then, all numbers less than x have to go to A, and all numbers larger than x have to go to B. So, some prefix of the sorted array will go to the first group, and the remaining suffix — to the second. We can check if there exists such splitting point with counting all prefix `OR` and suffix `AND`.

This already gives an $O(qn\log(n))$ solution, but this is too slow, of course. How to do better than this?

Let's dive a bit deeper. Again, suppose that it's possible to split its elements into 2 groups A and B so that `OR` of the numbers from A = `AND` of the numbers from $B = x$. Suppose that x has k bits on. Then, all numbers with less than k bits on have to get into A, and all numbers with more than k bits on have to get into B. What about the numbers with exactly k bits?

There are 2 cases: all of them go to the same group, or some go to A and some to B, let's consider the second case. Suppose that number a with k bits goes to A and number b with k bits goes to B. Note that b `AND` $a = a$, as b `AND` $x = x$ and x `OR` $a = x$. So, we must have $a = b$. But then, all numbers with k bits have to be equal.

Now, let's keep some data structure allowing us to find `AND` and `OR` of elements with exactly k bits on a segment for every k from 0 to 30: segment tree works, sqrt decomposition also works. To process the query, find `AND` and `OR` of elements with exactly k bits on this segment for every k, and precompute prefix `AND`s and suffix `OR`s. Note that the only case where we don't simply put all numbers with $\leq k$ bits to A and with $> k$ bits to B is when the prefix `OR` of all numbers with $\leq k$ bits = suffix `AND` of all numbers with $> k$ bits, and all numbers with exactly k bits are the same, and there are at least 2 of them, which is easy to check now.

Complexities for this solution can vary from $O((n+q\log n)\cdot b)$ (with segment tree) to $O(n\sqrt{n}+(q+n)\cdot b)$ (Mo's algorithm) or even $O(n\log n+(q+n)\cdot b)$ (with divide & conquer), where $b = 30$ is the number of bits.

Problem I. Modulo Permutations

Author: Anton Trygub
Solved by: 23/30
First to solve: *Insight*

Let's consider 2 arcs between the numbers 1 and 2.

First observation: $a \bmod 2$, $a \bmod 1$, $1 \bmod a$, $2 \bmod a \leq 2$ for any a, so the conditions involving 1 and 2 are automatically satisfied and we can forget about them.

Second observation: in each arc the numbers are strictly decreasing (if $2 < x < y$ on some arc, we get $x \bmod y = x > 2$, contradiction).

Now, we need to do a sieve-like dp. Let $dp[x]$ denote the number of ways to split the numbers from n to x into 2 arcs so that x goes to the first arc and $x+1$ goes to the second. We can calculate this dp for each i from n to 3 in the following manner: suppose that the numbers $x+1, \ldots, k$ go to the second arc while $k+1$ goes to the first one. Then we must have $(k+1) \bmod x \leq 2$, and the number of such k is $\leq 3\frac{n}{x}$. As the sum of $\frac{n}{x}$ over all x is $O(n \log(n))$, we get $O(n \log(n))$ solution overall.

Problem J. One Piece

Authors: Daniel Posdărăscu
Lucian Bicsi

Solved by: 0/3
First to solve: *N/A*

We'll start by handling the easy case of one treasure (the condition for this case is when there is some i for which $dist_i = 0$, where $dist_i$ is the distance shown by the detector on the i-th island.

Let's consider an arbitrary subset of treasures S. How would the distance array $dist$ look in this particular case? Well, it turns out that only two treasures matter in this case: the most distant two in S. This can be seen by analyzing the case with 3 treasures: suppose there are 3 treasures, and the subtree induced by them is formed of 3 chains which meet in some middle island. Let's suppose the length of the 3 "legs" are $a \leq b \leq c$. Then, the treasure at distance a from the middle island will never be the farthest treasure for any of the n islands.

Therefore, one may see that the distances array looks as if only two treasures are present. More specifically, by analyzing the islands i for which d_i is minimum (let's call such islands **centers**), there are two cases:

1. There is one center island;
2. There are two neighbouring center islands.

In both cases, we can notice that, in order for the distance array to be valid, two conditions have to be fulfilled (considering $d = \min(dist)$):

1. All treasure islands are at distance at most d from the central island(s);
2. There are at least two treasures at distance d from the central island(s) in different subtrees.

Now that we know the constraints that the distance array $dist$ imposes on the structure of the treasure islands, the rest is some counting/probability. Naturally, vertices with higher probability correspond to vertices which appear in many configurations. Indeed, Bayes' rule states that:

$$P(\text{treasure is in island } i \mid d) \propto P(d \mid \text{treasure is in island } i) \cdot P(\text{treasure is in island } i)$$

Here $\propto$ means "is proportional to". Note that the latter term in the right hand side is always 0.5, so it can be ignored.

However, for an island at depth greater than d the probability is always zero, and for islands at depth less than d, it is equal to 0.5, and is always (strictly) smaller than the probability for any island at depth d (every valid subset containing the treasure at depth smaller than d can be transformed into one having the treasure at depth d, but not vice-versa). The only thing that remains to calculate is how two different treasures at depth d compare to each other.

However, the number of configurations having some treasure at depth d can be calculated as $2^{M-1} - 2^{I+S}$, where M is the number of islands at depth at most d, I is the number of islands at depth less than d, and S is the number of islands at depth d in the same center's subtree as the treasure in question. But M and I are the same for all islands, so comparing two such islands reduces to comparing S (in other words, an island a has a greater probability than an island b if and only if the number of vertices at depth d in the same subtree as a is smaller than in b).

Now, the solution is simple. First, assign label(v) as being:

1. The number of treasures at depth d under the same center's subtree as v, if v is at depth d;
2. n, if v is at depth smaller than d;
3. $n + 1$, if v is at depth greater than d.

Then, just sort the islands by the pair $(\text{label}(v), v)$. Complexity is $O(n \log n)$.

Problem K. Codenames

Author: Lucian Bicsi
Solved by: 0/5
First to solve: *N/A*

Let's start by doing the following pre-computations:

1. For each string and its proper prefixes, consider the set of its characters and add it to a list
2. For each query, create a string of length 25 where the `r` characters are replaced with 1, the uppercase characters are replaced with ?, and all the other characters are replaced with 0.

The problem now becomes the following:

You have a set S of n numbers of d bits ($d = 25$) and q queries ($1 \le q \le 10^5$), consisting of binary patterns with wildcards (containing 0, 1, and ?). For each of the queries, find a number matching the pattern, or report if no such number exists.

We aim to solve the problem in $O(n + 2^d \cdot d + q \cdot 2^{d/3})$.

Let's notice that if p has at most $d/3$ question marks, then we can just brute-force check all possible configurations behind the question marks, in $O(2^{d/3})$. The more interesting case is when p contains many question marks.

In order to solve it, we will build a procedure count(p) which will return the number of matches of p in S (considering that p has at least $d/3$ question marks).

Let's suppose p contains fewer 1s than 0s. In this case, we will compute count(p) using inclusion-exclusion on the positions of 1, as follows:

- If p does not contain any 1s, then count(p) is the number of subsets of the set of ?; for example, if $p = 00??0?$, then count(p) $= \#\{x \in S \mid x \texttt{ OR } 001101_{(2)} = 001101_{(2)}\}$. This can be pre-computed for all possible patterns using the popular "sum over subsets" dynamic programming technique;
- If p contains at least one 1, then count($\ldots 1 \ldots$) = count($\ldots ? \ldots$) − count($\ldots 0 \ldots$).

This can be recognized as being an inclusion-exclusin over the number of 1s, and the complexity of the solution is $O(2^o)$, where o is the number of ones in the pattern.

We can come up with a very similar approach for the case where p contains fewer 0s than 1s. In this case:

- If p does not contain any 0s, then count(p) is the number of supersets of the set of 1s; for example, if $p = 11??1?$, then count(p) $= \#\{x \in S \mid x \texttt{ AND } 110010_{(2)} = 110010_{(2)}\}$. This can be pre-computed also using the "sum over subsets (supersets)" technique;
- If p contains at least one 0, then count($\ldots 0 \ldots$) = count($\ldots ? \ldots$) − count($\ldots 1 \ldots$).

This, again, ends up being just inclusion-exclusion on the number of 0s, and the complexity is $O(2^z)$, where z is the number of zeros in the pattern.

Choosing the best of the two approaches adaptively yields a complexity of $O(2^{\min(o,z)}) = O(2^{d/3})$ for count.

After having the above procedure, determining an actual solution is easy: just iteratively set some question mark to 0, and check if the new pattern is still matched (using count, of course); if not, change that position to 1 instead. We can do this iteratively until at most $d/3$ question marks remain, where we can transition to brute force.

At first glance such approach might seem that it works in $O(2^{d/3} \cdot d)$ per query (which might sound unpleasantly slow), but after some more thorough examining, the complexity is actually bounded by $2^1 + 2^1 + 2^2 + 2^2 + \ldots + 2^{d/3} + 2^{d/3} = O(2^{d/3})$.

Problem L. Neo-Robin Hood

Authors: Daniel Posdărăscu
Lucian Bicsi
Solved by: 14/37
First to solve: *KhNURE_Energy is not over*

Let's start by sorting the politicians decreasingly on their wealth. It is not hard to argue that, for any fixed set of k "helped" politicians, it is always optimal to rob the k richest remaining politicians. Let's binary search on the answer k, and do the following approach:

Let's consider the i-th politician (in sorted order) is the least expensive politician that we decide to rob. Naturally, it must be the case than $i \geq k$. Then, we will have to steal from k of the politicians in the left part $[1..i]$, help the other $i - k$ get political influence, and then also choose some extra $2k - i$ politicians in the right part to help, in order to cover up for all k of our thefts. But which politicians to choose?

From the right part, it's easy: just choose the $2k - i$ politicians with lowest p_j values. From the left part, we have to assess the gain from deciding to steal from a politician, instead of helping him. This gain turns out to be $p_j + m_j$ for all politicians in the left part, as Neo-Robin Hood will save m_j dollars from not helping the politician, as well as p_j dollars from stealing from it.

From here on, the solution should be clear: for all $i \geq k$, we try to pick the smallest $2k - i$ p_j values in the right part, and the biggest k $p_j + m_j$ values in the left part. If we can satisfy these politicians for some $k \leq i \leq 2k$ (the money that we steal from is assured), then we can conclude that the answer is at least k.

Dynamically keeping the largest k $p_j + m_j$ values and the largest $2k - i$ m_j values can be easily computed using two separate passes, and priority queues in $O(\log n)$. More involved solutions (using binary search on treaps/Fenwick trees) are also possible.

Total complexity is $O(n \log^2 n)$.

Problem M. Mistake

Author: Lucian Bicsi
Solved by: 92/98
First to solve: *CodeBusters*

Let's consider the job ids in order from left to right. One may argue that it is always optimal to put the current job id in the run with the smallest id that can be placed. This creates a scenario where the currently placed jobs in run i always include the currently placed jobs in run $i+1$, for all $1 \leq i < k$. Intuitively, this allows for the most opportunity for future jobs to be able to be placed somewhere, without violating the current constraints.

We'll prove the greedy using an exchange argument: suppose there exist a solution, and that our approach does not yield a valid solution. Let's choose the solution with the longest common prefix with the greedy solution. Let X be the first step where the (incorrect) greedy solution and the correct solution diverge. The correct solution will, then, not place the job in the first available run. Let's try to move it to the smallest run instead. This may create a number of conflicts, but one may prove that one may solve all conflicts with exchanges by induction on the earliest conflict.

Therefore, the solution is to place the i-th occurence of job j in the i-th run. If a solution exists, then this solution is valid. It ultimately turns out that the dependencies are not useful for solving the problem, and they can just be ignored.

It is also, quite possible, that the problem can be solved "by **Mistake**", as a lot of approaches ultimately lead to something equivalent to the solution described above.

The total complexity is $O(m + nk)$.