Problem Tutorial: "Absenteeism"

Approach 1. Let's draw a coordinate plane with axes x and y, where x represents the moment of time when the main character comes to work, and y — when he leaves work. Every possible answer is an integer point on this plane.

Each employee prohibits some areas on this plane. Some of them are rectangles, and some are triangles, but all of them can be extended to rectangles by also prohibiting areas with $x > y$ which make no sense in the problem and therefore don't affect the answer. For example, employee (a, b) prohibits all (x, y) such that $a \le x \le y \le b$. The fair prohibited area is a right triangle built on points (a, a), (a, b) and (b, b), but we can extend it to a rectangle with bottom-left point (a, a) and top-right point (b, b). Other conditions lead to other prohibited areas.

After all rectangles are built, the task transforms to just finding a point which is not covered by any rectangle and having minimal $y - x$. It is very standard task solvable by sweep line and segment tree. One more thing, you should check if the answer is "`-1 -1`" before doing the main part of the solution.

There are other solutions based on the same approach. First of them is binary searching the answer, and then doing some events processing. Events are built similar to rectangles in the previous solution. Some implementations don't fit in time limit, so you must be careful writing this solution.

Approach 2. There's a more constructive way to find a solution. First, if there are no employees (a, b) that we have to meet because of the last condition (employee doesn't see you at all $\implies$ you worked less than k hours), we don't have to go to work at all. Otherwise, any such employee introduces a constraint "$y \ge a$ and $x \le b$". Together, all of these constraints become "$y \ge y_0$ and $x \le x_0$", where, say, $y_0 = \max a$ among all such employees. As long as these are met, we can forget about the last condition.

If there are two employees (a, b) and (a', b'), and $a \le a' \le b' \le b$, we can ignore the employee (a', b'). Indeed, any constraint imposed by (a', b') is also imposed by (a, b). This corresponds to removing segments nested in other segments. The remaining segments (a_i, b_i) can be ordered such that $a_i < a_{i+1}$ and $b_i < b_{i+1}$ for all i.

If the answer (x, y) is contained in any of those segments, we must have $y - x \ge k$, which does not improve on the trivial answer. Thus, we now consider segments (x, y) not contained in any (a_i, b_i). Let i be the largest index such that $b_i < y$ (we can put $b_0 = -\infty$). Then, we must have $x < a_{i+1}$, otherwise we are contained in (a_{i+1}, b_{i+1}) (put $a_{n+1} = +\infty$).

For the chosen i, start with $x = a_{i+1} - 1, y = b_i + 1$. Apply $y \ge y_0, x \le x_0$ if necessary. This is a valid solition, unless, say x is contained in any (a_j, b_j), and $x > m - k$. In this case, move x to the left until either $x = m - k$, or x is not contained in any (a_j, b_j). The same can be done with y (moving to the right). The resulting segment (x, y) is clearly the smallest possible solution under $x < a_{i+1}, y > b_i$. Try all options of i, and choose the best answer.

To speed up, say, moving x to the left, for all segments (a_i, b_i) compute l_i — the result of moving $x > m - k$, provided it starts inside (a_i, b_i). If $m - k \ge a_i$, then $l_i = m - k$, else if (a_i, b_i) intersects with (a_{i-1}, b_{i-1}), then $l_i = l_{i-1}$. Otherwise, $l_i = a_i - 1$ since we stop just outside (a_i, b_i).

Altogether, the solution involves a single sort for the initial segments sweep-line, and some linear-time work after that. The complexity is still $O(n \log n)$, however, the constant factor is far smaller than the previous solution.

Problem Tutorial: "Fakes and Shidget"

Let's binary search the answer. So we have a speed x and we must say if we can earn the gold with the speed $\le x$.

Let's reformulate the task. We will say that every second we lose x gold and we must determine if our balance remains positive at infinity. So the first quest will bring b gold but for a seconds we will lose $a \cdot x$ gold, and the second quest will bring d gold but we will lose $c \cdot x$ gold. Now it's obvious we should just compare $b - ax$ and $d - cx$ and choose the corresponding quest. All that remains is to sum up the chosen values for all quests and compare the sum with 0.

Problem Tutorial: "Cyclically Shifted Maze"

Let's try all possible column shifts and forget about them. Now we should say if some row shift gives us a connected maze.

For each prefix of rows $[0, r]$ and each suffix $[r, n-1]$ we will calculate two things: the number of connected components in the corresponding part of the maze, and the index of the connected component for each empty cell in the first (for prefix) / last (for suffix) row. Let's show how to that for prefixes. We will maintain the DSU for the prefix-maze consisting of rows $[0, r]$. When we attach a new row $r+1$ to it, we will do $O(m)$ merges in DSU, connecting a $(r+1)$-th row to the prefix part. After these merges, the number of connected components may change, and the indices of the connected components of the 0-th row may change too. Remember them and process the next row, and so on.

Now let's try all row shifts. When we try row r, we know the indices of the connected components of the 0-th row in $[0, r]$ part and the indices of the connected components of the $(n-1)$-th row in $[r+1, n-1]$ part. We also know the number of connected components in these parts. We have precalculated all these things in the previous part of solution. Now connect the 0-th row from the prefix and the $(n-1)$-th row from the suffix together, changing the number of connected components. If this number becomes 1 after merges, this row shift is the answer.

Problem Tutorial: "Two Pirates - 2"

Order treasures by non-decreasing of cost. Consider the process in reverse, that we can describe as follows: there is a sequence of black and white balls, that represent treasures that go to sober and drunk pirates respectively. Initially the sequence is empty, and pirates insert balls of their color in reverse order of their moves. The sober pirate adds a black ball to the end of the sequence (since he's taking the last treasure). The drunk pirate adds a white ball randomly in one of positions: before the first ball, between the first and second ball, ..., after the last ball. In the end, the sober pirate gets a treasure i if i-th leftmost ball is black.

Let $p_{i,j}$ be the probability that the j-th leftmost ball is black when there are i balls in the sequence. Consider going from i to $i+1$. If the sober pirate moves, then $p_{i+1,j} = p_{i,j}$ for $j = 1, \ldots, i$, and $p_{i+1,i+1} = 1$. If the drunk pirate moves, then $p_{i+1,j} = \frac{j-1}{i+1} p_{i,j-1} + \frac{i+1-j}{i+1} p_{i,j}$, with the two summands corresponding to whether the white ball is inserted to the left or to the right of the ball that arrives at position j. The final answer (for the sober pirate) can be found as $\sum_{j=1}^{n} p_{n,j} a_j$.

This results in an $O(n^2)$ DP solution. We can only store the current row of $p_{i,j}$, thus we only need $O(n)$ memory.

Extra challenge: when n is even, $p_{n,j}$ obey a simple formula, and the problem can be solved in linear time. Can we do faster than n^2 when n is odd?

Problem Tutorial: "Powerless Mage"

Let's sort all spells by the z-coordinate and process them in this order. If (X, Y, Z) is the answer, spells with $z > Z$ cannot be casted, but they are not processed yet, and we can forget about them, also for each processed spell (x, y) there must be either $x > X$ or $y > Y$.

Let's store (x, y) of all processed spells in some data structure. This data structure must be able to add point, rebuilding itself, and to say what is the optimal answer for all contained points. It turns out that such data structure consists of `map<int, int>` for points and of `multiset<int>` for answers.

The map must hold an invariant: if $x_1 < x_2$, then $y_1 > y_2$. When we insert a new point (x, y), we also erase all neighbours to the left and to the right until invariant becomes true. Also we don't forget to maintain the set of possible answers: neighbour points (x_1, y_1) and (x_2, y_2) in the map form a candidate answer $x_2 + y_1$.

Problem Tutorial: "Exactly One Point"

Let's do a simple dp: dp_x is the parent coordinate if we can place a point to x, and -1 if we can't. We

will calculate this dp forward. So for the current x, if we place the point to x, we must say where we can place the next point.

To do that, we will precalculate two values: $maxRight(x)$ is the maximal r among all segments having $l \le x$, and $next(x)$ is the segment with the minimal r having $l > x$. Then, for some coordinate x, if we place a point to x, the next point must have coordinate in the segment $[maxRight(x) + 1, next(x).r]$. It cannot be less or equal than $maxRight(x)$ because we already placed a point to x and the farthest segment containing x ends at $maxRight(x)$. And it cannot be greater than $next(x).r$ because segment $next(x)$ will be left without a point.

Setting a value to an interval can be done with segment tree. Then just restore the answer.

Problem Tutorial: "Lexicographically Minimal Subsequence"

Let's build the answer greedily. At each step, we know the position of last letter added to the answer, let it be pos. Try all letters from "a" to "z". Let's say we are trying a letter c. Find the first position $next$ such that $s_{next} = c$ and $next > pos$. If we use this letter in the answer, we must be sure we can build a tail: the total number of letters to the right of $next$ must be greater or equal than k minus length of the current answer with the added letter c. If it is, we cannot fail in the future — we have enough letters in the remaining part of s — and at this step the letter c is optimal.

Problem Tutorial: "Video Reviews - 2"

The problem has an obvious solution with the binary search, but the constraints were chosen to cut this solution.

First solve it in $O(n)$ if we have enough memory for storing the array a. In the end we will have m videos. Let's consider the last man. If he has $a_i < m$, he will record a video and we can solve the same problem for $m - 1$ and $a[0 \dots n - 2]$. If he has $a_i \ge m$, we must force him to record a video if $i + 1 = m$, because otherwise we simply don't have enough men on the prefix $a[0 \dots i]$ to record m videos. Continue doing this until we process all men.

The $O(n)$ memory variant still had a problem with a possibility of squeezing binary search, so the memory was cut. It is not a big deal, because all z_j are prime, so the generators are inversible. We can just save the last number produced by all generators, and then play them back.

Problem Tutorial: "Chess Tournament"

This problem has appeared in real life, so the author's solution is not some beautiful construction, it's just a greedy that passes all possible tests. But beautiful solutions exist.

The most simple greedy you can think of is to keep a list of all pairs and try to unite k of them in a round, keeping track of used players. If you sort this list or random-shuffle it, it will not work, but surprisingly, if you generate some schedule for $k = \frac{n}{2}$ and sort the pairs by round in this schedule, the greedy solution passes all tests.

One of the constructive solutions is the following. First of all, there is an algorithm for building a schedule for $k = \frac{n}{2}$. Let's say the first round is a_1 vs b_1, a_2 vs b_2, $\dots a_k$ vs b_k. We will fix a_1 and for all the next rounds rotate every other player clockwise, so the next round will be: a_1 vs a_2, a_3 vs b_1, a_4 vs b_2, $\dots$, a_k vs b_{k-2}, b_k vs b_{k-1}. Do it $(n - 2)$ times and get a correct schedule.

What is good in this schedule, for each player and every two consecutive rounds, the positions of pairs with this player differ at most by 1. So we can list these pairs and use every chunk of k pairs as a round. It is always correct because if we lack at least one board ($k < \frac{n}{2}$), the distance between pairs with the same player is at least $\frac{n}{2} - 1$, so the borders between rounds will always separate pairs with the same player, leaving them in different rounds.

Problem Tutorial: "Lost Island"

The solution for $a = b = 1$ is described in the notes. Let's see what happens if we have a color c with

$a = 2, b = 1$, and all other colors have $b = 0$.

If you are a person with eye color c (you don't know it), you can see only one person with eye color c in front of you. But, suddenly, they don't die after the first day. How can it be? The only possible way is that you also have eye color c. The other person thinks the same way, and you both die after the second day.

Continue this logic, and we will have that for each group of people with the same color and $b_i > 0$, they will die at the day $d_i = a_i - b_i + 1$. If there are at least two $b_i = 0$, that's only that happens, but there are some additional cases:

- if there is exactly one $b_i = 0$, this group will die after the last group having $b_i > 0$ dies, because they will know only one color remains,
- if all $b_i > 0$, the situation from the previous case may appear earlier than day d_i comes for the last group, so the answer is $min(d_0, d_1 + 1)$ if we sort d in decreasing order.

Problem Tutorial: "Bloodseeker"

This problem was intended to be the second easiest task in the problemset, but somehow people had troubles with it.

Let's split the enemies into two groups: which have $t_i \le h_i$ and $t_i > h_i$. It's obvious that we should kill enemies from the first group first.

To prevent overheal, we will hit the enemies from the first group until they have 1 hp, and move them to "1 hp pool". Each time we will have 1 hp in the future, we will for a bit pause doing our main job, kill someone from 1 hp pool to restore some hp, and continue.

In what order should we process enemies from the second group? The right answer is by decreasing h_i values. If you process two enemies i and j with $h_i < h_j$, you better swap them, processing j first, because your hp will be higher after the first regeneration (after j), and be the same after the second regeneration (after i). If you don't swap them, the order i, j may fail you because you may not regenerate enough hp after killing i.

Problem Tutorial: "Not the Longest Increasing Subsequence"

Let's call element a_i reachable if there is a sequence of values $1, 2, \ldots$ ending with a_i. For all values x, there can't be a situation where we deleted some reachable element $a_i = x$, but didn't delete reachable element $a_j = x$ with $j < i$.

Now let's calculate a dp: dp_i is the minimal number of deletions if:

- a_i is the leftmost reachable element equal to a_i,
- we deleted some elements less than a_i or equal to a_i to the left of it.

We will calculate it forward. There are two cases:

- if we delete a_i, the next reachable position is the minimal position $j > i$ having $a_j = a_i$ — relax dp_j with $dp_i + 1$,
- if we don't delete a_i, we can reach the next value — go to the minimal $j > i$ having $a_j = a_i + 1$ and relax dp_j with dp_i.

Also keep track of parents. Then restore the answer.

Problem Tutorial: "Binary Search Tree"

For most of the vertices, it's possible to correctly determine their parent, left child and right child:

- if a vertex x has three neighbours a, b, c, and x is the second value in the sorted list $[a, x, b, c]$, then c is parent, a is the left child, b is the right child,
- if a vertex x has three neighbours a, b, c, and x is the third value in the sorted list $[a, b, x, c]$, then a is parent, b is the left child, c is the right child,
- if a vertex x has two neighbours a, b, and x is the smallest value in the sorted list $[x, a, b]$, then b is parent, a is the right child,
- if a vertex x has two neighbours a, b, and x is the largest value in the sorted list $[a, b, x]$, then a is parent, a is the left child.

For all such vertices, push them to a queue and do a BFS. When you move along the edge, you have some information about the previous vertex and are able to restore more and more vertices. This way almost all vertices, including all leaves, will know their parent, left child and right child.

All than can remain after the BFS is some chain of vertices $v_1, v_2, \ldots, v_k$, in which $v_1+1 = v2, v2+1 = v3$ and so on.

If any of these vertices is a correct root, all of them are correct roots. Just check any of them with a naive algorithm.

Problem Tutorial: "Premove Checkmate"

This problem was expected to be solved entirely with a pen and paper, so if you were able to use a computer — congratulations!

There are lots of solutions here. All of them consist of moving white queen one square top or one square right, being sure that the black king is still cut in the top-right part. Sometimes (not in all solutions) you can even allow black king to escape top-right part of the board, but knowing its exact location gives you a possibility to give a forced checkmate.

In general, the technique that is the most commonly used is to try to place a white king on some square, then move white queen to this square or further, and depending on was it successful or not, get some information about the black king position, divide the problem into two and solve them independently.