A. AppendAppendAppend

Author: Daniel Posdărăscu
Solved by: 87/123
First to solve: *KhNURE_MMXXII*

We can consider each character of t at a time and compute the minimum length l of the answer, as well as a pointer p to the current match in s (initially, $l = p = 0$).

When a new character c arrives, we look into the list of positions of c in s, and find the next occurence greater than the current position greater than p (using binary search).

If that position exists, set p as that position.

If there is no next position, we increment l and set p as the first occurence of c in s.

Total complexity is $O(|s| + |t| \log |s|)$.

B. Birthday Cake

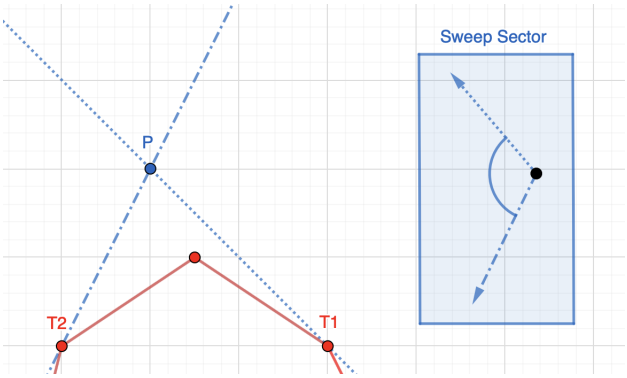
Author: *Anonymous*
Solved by: 9/33
First to solve: *LNU Stallions*

Let's denote the position of the chocolate chips as **blue** points, and the positions of the strawberries as **red** points. Then, the problem asks us to find a halfplane containing as many **blue** points as possible, and none of the **red** points.

Let's fix a given angle vector α, and consider cutting the region that is to the right of α. Imagine sliding the cut as much as possible towards the left of α (trying to cut as much of the cake as possible). When can we not cut anymore? Naturally, when we find the first **blue** point. This point is the extremal point for the angle $\alpha + \frac{\pi}{2}$.

However, it is not hard to see that an extremal point in any direction must lie on the convex hull. Let's build the convex hull of all red points. For reasons stated above, it makes sense to consider cuts that are tangent to the convex hull.

We'll go back and imagine sweeping radially, cutting tangentially to the convex hull of the red points (with a rotating caliper). Let's consider a blue point p. If p is inside the convex hull of the red points, then it will never be part of any cut. Otherwise, p will contribute to the answer precisely for a sector of angles in this sweeping procedure, given by the two tangents of p to the convex hull of the red points. See the below image for more detail.



Once we find all the n sectors for each of the blue points, we reduce to the classical problem of maximum intersections of n sectors in a circle, which we can solve with a sweep with $+1$ and -1 events. In order to treat the collinearity cases, we should consider each sector as being **open** (more specifically, consider the ending events before the starting events, for all angles). Angles should also be kept as vectors and sorted using the cross product, to avoid precision issues.

Finding upper/lower tangents from a point to a polygon and checking if a point is inside a polygon can easily be done in $O(m)$.

The total complexity is $O(n \log n + nm)$.

Note: Finding tangents to a convex polygon and checking if points are inside a convex polygon can also be done in $O(\log m)$ *using binary search, therefore* $O((n+m)\log(n+m))$ *is also possible. However, we decided not to require this harder version in the contest. We leave it as an exercise for the experienced World Finals aspirers.*

C. COVID

Author: Lucian Bicsi
Solved by: 6/15
First to solve: *UoB R-Shuf*

Let's use the tip given in the statement and compute for person i the number of **in**valid scenarios where that person is positive.

Naturally, this depends on the subset of tests where person i did not take part (the tests where person i takes part are automatically valid).

We use inclusion-exclusion to count the answer. For a given set of tests T, the number of subsets in which all tests $t \in T$ are invalid is $2^{n-1-sz(T)}$, where $sz(T)$ is the number of people in at least one of the sets (any people that belong to at least one of the sets must be healthy, person i must be sick, and all other people may be in each of the two states). We will pre-compute $sz(T)$ for all possible subsets T in complexity $O(2^m n)$

The true answer for a given person i is then a sum of numbers of form 2^{n-1-s}. We can count the contribution of each subset T in a frequency array in complexity $O(2^m)$, and then convert these arrays to binary bignums (boolean arrays) of at most n bits, using a simple left-to-right pass (carry propagation) at the end.

Afterwards, sort **decreasingly** by comparing with usual lexicographic comparator.

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

Note that solutions using floating point arithmetic require far too big precision ($O(n)$ bits) to be able to correctly sort the people.

D. Divisible by 4 Spanning Tree

Author: Anton Trygub
Solved by: 0/2
First to solve: *N/A*

Note that the number of nodes with an odd degree is always even.

First, find any spanning tree. If the number of nodes with an odd degree in it is divisible by 4, return `YES`. Otherwise, we have to determine whether it's possible to "switch"this number mod4.

First, find all bridges; they will be in any spanning tree. Consider any biconnected component. We have to determine if it's possible to "switch"the number of nodes with an odd degree (mod4) separately for each of them.

First, consider any cycle $v_1, v_2, \ldots, v_k$. Let's choose all except $k-1$ edges of the spanning tree, such that all v_i are in different components. Now, we have a choice: we have to choose all edges except some particular edge. It follows that if for some i, parities of degrees of nodes v_{i-1} and v_{i+1} are different, then we can "switch"in this cycle (by excluding edge (v_{i-1}, v_i) or (v_i, v_{i+1})).

Now, consider any biconnected component. If it's just one cycle, then just check the condition above. Otherwise, it has to contain two simple cycles that have common nodes. It's easy to see that it has to contain two simple cycles such that the common nodes in them form a segment in both of them.

If the intersection is just one node, say v, then note that in the first cycle, by excluding the appropriate edge, we can change the parity of degree of v. So, we would make the condition not hold for the second cycle.

Now, the intersection is at least of 2 nodes. In fact, this looks as follows: we have some two different nodes, u and v, and three disjoint paths between them. If the length of at least one of these paths is at least 3, then, by excluding the appropriate edge in this path, we can vary the parity of degree of u without changing the parity of degrees of other nodes in the remaining two paths, so we would be able to "switch"here.

We are left with two cases: lengths are $(2, 2, 2)$, or $(1, 2, 2)$. We can chow that we can "switch"in lengths $(1, 2, 2)$. Otherwise, we get that if two cycles intersect (by their common segment), then the corresponding lengths are $2, 2, 2$; it's easy to show that the entire component is just a graph on 5 nodes then, with edges $(1, 2), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)$). In this component, we can show that we won't be able to switch the parity only if the parities of degrees of $2, 3, 4$ are all equal and of $1, 5$ are opposite.

Then the entire algorithm would be to find all biconnected components and check whether each of them is a cycle in which nodes at a distance 2 have the same parity of degree or a particular graph on 5 nodes and 6 edges with some conditions on parities of degrees.

Final complexity is $O(n + m)$.

E. Exercise

Author: Anton Trygub
Solved by: 38/68
First to solve: *Infinity*

Let's sort the students so that $c_1 \le c_2 \le \ldots \le c_{2n}$, and remember which n pairs are prohibited.

If we had no prohibited pairs, the optimal solution would be just to form pairs $(c_1, c_2), (c_3, c_4), \ldots, (c_{2n-1}, c_{2n})$.

Now, consider any optimal solution. If there are several optimal solutions, consider the one in which the sum of $(x-y)^2$ over pairs (c_x, c_y) is minimized.

Now, suppose that we have some pairs (c_x, c_y) and (c_z, c_t) for some $x < y$, $z < t$. Then:

- $x < z < t < y$ is impossible.

  Proof: This means that we can't form pairs $(c_x, c_z), (c_t, c_y)$ and $(c_x, c_t), (c_z, c_y)$. This is impossible. For example, if pair (c_x, c_z) is banned, then both pairs $(c_x, c_t), (c_z, c_y)$ aren't banned.

- If (c_x, c_y) is a pair, then $|y - x| \le 3$.

  Proof: Wlog $x < y$. Consider any c_z for $z \in [x+1, y-1]$. It forms a pair with some c_t, where t is outside of this range. If $t < x$, then one of pairs $(c_t, c_x), (c_z, c_y)$ must be banned, else one of pairs $(c_x, c_z), (c_y, c_t)$ must be banned.

  It follows that there are at least $y - x - 1$ banned pairs for c_x and c_y in total, but there are, of course, only 2 such pairs, q.e.d.

So, there is an optimal solution in which all pairs are at a distance at most 3. Then there is the following solution: go from left to right, and keep *dp*: what's the smallest possible sum of differences in the given prefix, if we paired all elements except certain *mask* among the last 3 elements. Updating this *dp* is easy: consider new element, and pair it with someone unpaired in the previous 3 elements, or mark it as yet unpaired.

The final complexity is $O(n)$.

F. Fortune over Sportsmanship

Author: Daniel Posdărăscu
Solved by: 46/53
First to solve: *Giocanul*

It turns out that the solution equals exactly the **maximum spanning tree** of the complete graph where we add edges (i, j) of weight $P_{i,j}$ for all $1 \leq i < j \leq n$.

First, for any candidate tournament, the pairs (i, j) that describe the popularity $P_{i,j}$ of each of the n matches should form a tree. This can be seen inductively, as each new match (a, b) yields some $P_{i,j}$ where i is one of the contestants that were (directly or indirectly) beaten by the first player a, and j is one of the contestants that were (directly or indirectly) beaten by the second player b. Then, the virtual edge (i, j) would have to connect different connected components in the imagined complete graph.

This proves that the pairs (i, j) form a tree; now all it remains is to show that we can construct the "best" tree (the MST). This can be proven by construction: run Kruskal's algorithm, and for each tree edge (i, j) output a match between the players (a, b), where a is the minimum index in the connected component in which i resides, and b is the minimum index in the component where j resides.

Another, more intuitive way to view this transformation, is by first noticing that the fact that the smallest indexed player wins in a match between i and j is not important; one could think of a "superplayer" which encompasses both i and j, and replace both of them with this "superplayer". This immediately shows that, even though one of i and j would lose, their popularity can be re-used nonetheless for other matches (if it's profitable). Therefore, any configuration where the matches form a tree can be achieved via either the original players, or some "superplayers" that come from one or more matches where the original players took place.

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

Note: The problem can also be solved in $O(n^2)$ by using Prim's MST algorithm. However, this was not required to solve the problem.

G. Gears

Author: Lucian Bicsi
Solved by: 35/63
First to solve: *UoB R-Shuf*

The key observation to solving this problem is noticing that, once we place the first gear, the others are uniquely determined. This gives us a rather straightforward $O(n^2)$ solution, which we need to further optimize to fit for the big constraints.

Let $r_1 = X$ be the radius of the first gear in the correct placement. Then, the radius of the second gear is $r_2 = (x_2 - x_1) - r_1 = (x_2 - x_1) - X$. Further, the radius of the third gear is $r_3 = (x_3 - x_2) - r_2 = (x_3 - x_2) - (x_2 - x_1) + X$.

The important thing to note is that we can express all other $n - 1$ radii as either $X + a_i$ or $a_i - X$ (depending on the parity of i) for some a_i. How to compute the terms a_i is left as an exercise for the reader.

Notice that the gear with the minimum radius must be placed either on the position with the minimum of all odd a_i's or the minimum of all even a_i's. Either case deduces the value of X and, consequently, of all radii. So try both, and simply check which case outputs a correct solution.

Complexity is $O(n)$ or $O(n \log n)$, depending on how you implement the checker.

Note: Solutions based on hashing are also possible.

H. Hanoi

Author: Alexandru Lungu
Solved by: 72/88
First to solve: *Infinity*

There are many techniques which lead to at most $2 \cdot n^2$ moves. We will present only one of them:

Consider we already have placed the top-most $p-1$ disks from the first rod to the third rod and we are going to move the p-th disk now. First move, this p-th disk on the auxiliary rod (the second rod). As long as we can't move this p-th disk on the third rod, move disks from the third rod on the first, as there, the order doesn't matter anyways. Move the p-th disk on the third rod and revert back the disks moved to the first rod in the last step. At the end of this process, the top-most p disks from the first rod are correctly placed on the third rod. This process takes at most $2 + 2 \cdot (p-1)$ moves.

Repeating the previous process for all disks, solves the problem. The total number of moves is $n \cdot (n+1)$ which is always less than $2 \cdot n^2$ for any $1 \leq n$.

I. Inadequate Operation

Author: Anton Trygub
Solved by: 15/43
First to solve: *UzhNU_OLDS*

Let's call an operation k-operation, if $\max(a_i, a_{i+1})$ in it is k.

Consider any $k > 0$. Consider all elements which are $\geq k$; they form some consecutive segments of lengths, say, $l_1, l_2, \ldots, l_t$.

Claim: We have to perform at least $\lceil \frac{l_1}{2} \rceil + \lceil \frac{l_2}{2} \rceil + \ldots + \lceil \frac{l_t}{2} \rceil$ k-operations.

Proof: Each element can decrease at most by 1 in an operation, so every element $\geq k$ will have to be involved in at least one k-operation, q.e.d..

It turns out that making exactly this number of k-operations for every k is achievable. Let M be the largest number. Consider any segment of consecutive Ms; let it have length l.

If $l \geq 2$, then we can trivially make all the elements in this segment equal to $M - 1$ in $\lceil \frac{l}{2} \rceil$ queries.

If $l = 1$, consider its neighbors. If it has only one neighbor, so, wlog, $a_1 = M$, then just make an operation with a_1, a_2. Note that while a_2 might increase, the number of required k-operations for all $k < M$ will remain the same (as for $k \leq a_2$ nothing changes, and for $k > a_2$ a_1 was forming a separate segment anyways, so we might as well make a_2 equal to $M - 1$, and make a segment $[a_1, a_2]$ instead).

Now suppose that $a_i = M$ with $2 \leq i \leq n - 1$. Wlog $a_{i-1} \leq a_{i+1} < M$. Then make an operation with (a_i, a_{i+1}). The argument is the same: for $k \leq a_{i+1}$ nothing changes, and for $k > a_{i+1}$ a_i was forming a separate segment anyways, so we might as well make a_{i+1} equal to $M - 1$, and make a segment $[a_i, a_{i+1}]$ instead.

How to find these values for all k? Well, just go over numbers from large to small, and keep track of consecutive segments (the easiest way to do so is using doubly-linked lists). Every time you merge two segments, recalculate the sum. The answers for k_1 and k_2 are the same if there is no a_i in $[k_1, k_2)$, and here are $O(n)$ distinct values in the array, so we can calculate all the places where the number of k-operations changes (and by how much) in $O(n \log n)$ (from sorting).

Note: The sorting can be completely removed, yielding a simpler $O(n)$ solution using a doubly-linked list and a queue. However, this was not required to solve the problem.

J. Joyful Death

Author: Daniel Posdărăscu
Solved by: 5/9
First to solve: *Infinity*

Solving just one instance of the problem is a rather straightforward problem, and can be done by the sweep line technique, using a priority queue to choose the best available dish at all times. However, it is hard to optimize straight-away to handle incrementally considering each elf. Nonetheless, we will see that this algorithm is crucial to build intuition for the harder problem.

Let's look at the naive algorithm in more detail. More specifically, let's consider how the algorithm operates for a set S of elves, and then imagine that one of the elves $x \in S$ is removed. It is not hard to prove that the dishes that would be chosen by running the algorithm without some elf $S \setminus \{x\}$ would always be a subset of the dishes chosen by the algorithm with all elves S. Therefore, it is correct to assume that, whenever the new elf arrives, all the already assigned dishes would still be assigned (possibly to different elves), and at most one extra dish will be chosen.

Note: The above observation is, in fact, valid for all instances of bipartite weighted matching, and it is a key observation of the incremental Hungarian Algorithm implementation.

However, one problem still remains: how to choose the best new dish? Intuitively, to answer this question, we have to analyze which new dishes are valid to be picked given the new elf (after possible re-assignments), and, thereafter, which sets of dishes are valid.

Let's go back to the algorithm and consider k elves and k dishes. In order for the assignment to be feasible, each of the k elves must have at least one dish in the queue when they get processed in the algorithm. Therefore, if we think of dishes as open brackets "(" and elves as closing brackets ")", then the sequence of opening and closing brackets must be **balanced** (in essence, for each prefix there must be at least as many open brackets – dishes – as closed ones – elves).

Note: One may invoke Hall's Marriage Theorem and arrive to the same fact.

Thinking of the process in terms of brackets simplifies the work by a lot. Each new elf corresponds to a closed bracket, and at each step we should find the best open bracket to add to the sequence to keep it balanced. Note that if the open bracket comes before the closed one, it will always be valid. This is also backed by intuition: we can always assign a dish to the current elf, and keep the other reassignments.

Interestingly, however, some open brackets that come after the current closed one are also valid. In essence, the valid open brackets form a prefix; this prefix ends at the smallest spot where there are just as many open brackets as closed, greater than the spot of the current closed bracket. The details of why this is true is left as an exercise to the reader.

In terms of implementation, finding out this prefix can be solved by binary searching the first zero prefix sum after a given position where we add $+1$ for each open bracket and -1 for each closed one, using a segment tree. Finding and removing the best non-chosen open bracket in that prefix is, again, a standard segment tree query. Extra care should be taken for when there is no such open bracket (dish) to assign, where no updates happen.

Final complexity is $O((n+m)\log(n+m))$.

Note: $O(n \log^2 n)$ or $O(n\sqrt{n})$ solutions may also pass, if implemented carefully.

K. Knowledge Testing Problem

Author: Lucian Bicsi
Daniel Posdărăscu
Solved by: 2/7
First to solve: *Echipa Sarata*

Let $d(=10)$ be the maximum distance between vertices allowed.

First relax the graph such that each shortest walk doesn't have any "u-turns". You should obtain that with two passes (one forward one backward), and some Roy-Floyd-like approach:

```
for i in 1..N:
  for j in i+1..i+D:
    for k in i+1..i+D:
       dist[j][k] = min(dist[j][k], dist[i][j] + dist[i][k])
```

Note: the above is just the forward pass

This essentially lets us view the graph as being directed (all edges be arcs from left to right).

Let's group the n nodes of this graph into blocks of size d, and build a segment tree, where each node in the tree stores a $d \times d$ matrix $M_{i,j} = \text{dist}(l+i, r+j)$. The complexity of building such segment tree is $O(n/d \cdot d^3) = O(nd^2)$.

We solve a query (u, v) in such a segment tree in $O(d^2 \log n)$, as follows:

Consider an array ans where $ans[i]$ is the minimum distance to vertex number i in some block of size d. Initially, we set $ans[u\%d] = 0$, and for the other values, $ans[i] = \infty$.

For each of the $2 \log n$ nodes visited by a query inside the segment tree, we update the ans array using the following procedure:

```
for i in 0..D-1:
  ans'[i] = INF

for i in 0..D-1:
  for j in 0..D-1:
    ans'[j] = min(ans'[j], M[i][j] + ans[i])
```

It is not hard to see that the new answer ans' contains the required distances to each of the new d vertices.

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

Note: The segment tree build and the query can be seen as tropical matrix-matrix and matrix-vector multiplication. Alternative solutions using Divide and Conquer are also possible. Solutions of complexity $O(nd^2 \log n)$ (using Dijkstra instead of Roy-Floyd-like relaxation) can also pass, given a careful implementation.

L. Level Up

Author: Alexandru Lungu
Solved by: 0/0
First to solve: *N/A*

There are some preliminary observations:

- In the i-th realm, the character can achieve between 1 and k_i level ups.
- In order to achieve j levels in the i-th realm, the character should choose to engage exactly the weakest j mobs from the i-th realm.
- It is optimal to kill the most powerful engaged mob in each turn.
- Engaging j mobs in the i-th realm damages the character $d_{i,j}$ hit points. Considering that $ps_{i,j}$ is the partial sum of the mob strengths in the i-th realm, $d_{i,j} = \sum_{k=1}^{j} ps_{i,k}$.
- When already at level m it is ideal to engage only the weakest mob.

This basically means that in each realm i we can choose to level up j times ($1 \leq j \leq k_i$) at the cost of $d_{i,j}$.

For determining the lowest starting HP, we can apply dynamic programming. Let $dp_{i,j}$ be the lowest HP required if the character just reached realm i and has level j. The recurrence is $dp_{i,j} = \min_{k=j+1}^{\min(j+k_i,m)}(\max(1, dp_{i+1,k} - h_k) + d_{i,k-j})$. Specially handle the case where $j = m$. The build-up should be done bottom-up and the answer is found in $dp_{1,1}$

Using this dynamic programming right away results into a $O(n \cdot m^2)$ solution. To reduce this, notice that the optimal k for computing $dp_{i,j}$ is non-decreasing. Therefore, we can use Divide and Conquer optimization to achieve a better complexity.

Final complexity is $O(nm \log m)$.

Note: Better complexity $O(nm)$ can be achieved using a technique similar to Convex Hull Trick, but it wasn't required for this problem.

M. Mousetrap

Author: Lucian Bicsi
Solved by: 3/15
First to solve: *SuteAlbastre*

First, let's denote the path from 1 to n in the tree as $1 = v_0, v_1, v_2, \ldots, v_{k-1}, v_k = n$. It is easy to see that it makes sense to add cheese only on chambers $v_1, v_2, \ldots, v_k$.

We can express the probability of exiting the network as a product $P = \prod_{i=1}^{k} \frac{a_i}{b_i}$, where $1 \le a_i \le b_i \le 10^9$ can be computed from the input. We leave the implementation details for computing these values as an exercise to the reader.

Let's analyze what happens when one adds a piece of cheese to some chamber v_i. It turns out that the probability becomes $P' = P \cdot \frac{b_i}{a_i} \cdot \frac{a_i+1}{b_i+1}$. Furthermore, note that adding more cheese yields an increasingly smaller benefit to the answer (a "diminishing returns" scenario); in other words, all k "benefit" functions are concave.

This means that we can deduce a correct $\tilde{O}(x)$ algorithm: using a priority queue, add cheese to the chamber with the biggest gain $g_i = \frac{b_i}{a_i} \cdot \frac{a_i+1}{b_i+1}$. However, this is too slow for this problem's constraint on x.

One way to optimize is to start with a "close enough" solution. One good candidate is the solution for the fractional problem, where one drops the constraint that the added cheese should be integer, and optimizes instead the log-product:

$$\textbf{max} \quad \sum_{i=1}^{k} (\log(a_i + x) - \log(b_i + x))$$

$$\textbf{s.t.} \quad x_i \ge 0 \quad \sum_{i=1}^{k} x_i = x$$

by binary searching on the derivative $\delta = \frac{d}{dx}(\log(a_i + x) - \log(b_i + x))$ (up to some precision ϵ). For a given derivative δ, the cheese added to each chamber can be computed by solving a quadratic equation.

One can prove that the solution for the fractional problem is $O(n)$ away from the integer solution, therefore the adjustment from the floating point solution to the integer solution can be done in $O(n \log n)$ using the above-described priority-queue method.

Comparing fractions using 128-bit integers yields a solution that suffers from no precision issues; however, using long doubles might also work.

Final complexity is $O(n \log n + n \log \epsilon^{-1})$.

N. Nusret Gökçe

Author: Daniel Posdărăscu
Solved by: 110/121
First to solve: *CodeBusters*

The constraints $|s_i - s_{i+1}| \leq m$ get split into two types of constraints:

1. $s_{i+1} \geq s_i - m$ $(1 \leq i \leq n-1)$
2. $s_{i-1} \geq s_i - m$ $(2 \leq i \leq n)$

We can solve all the constraints of the first type in a simple forward (left-to-right) pass, setting $s_{i+1} = \max\{s_{i+1}, s_i - m\}$. Similarly, we can solve all the constraints of the second type in a backward (right-to-left) pass, setting $s_{i-1} = \max\{s_{i-1}, s_i - m\}$.

One can easily see that each decision is "forced". Moreover, the backward pass will never introduce extra constraints of type 1, therefore the solution is valid.

One may also alternatively view this problem as a "system of difference constraints", and the two-pass approach yields the same result as applying the Bellman-Ford algorithm on the line graph with undirected edges $(i, i+1)$ of cost m (as each relaxation can go along a left-to-right path or along a right-to-left path).

Total complexity is $O(n)$.

Note: Alternatively, one may observe that the maximum value in s will never change. Therefore, one may propagate the constraints given by the maximum value left and right, erase it, and repeat. By using a priority queue to simulate the process, one can achieve $O(n \log n)$ complexity.