Problem A. ASCII Automata Art

Problem author and developer: Roman Elizarov

This is a simple problem. You just need to implement the requirements from the problem statement. The example in the problem statement was specifically designed to cover various cases to make debugging of the problem straightforward.

This problem is solved by parsing the regular expressions and then recursively generating the corresponding boxes as specified in the problem statement. Parsing of such grammar can be implemented using recursive descent.

The only tricky part of the problem is how neighboring letters that parse to a single $\langle term \rangle$ should be grouped into a single $3 \times (4 + n)$ box. There are two ways to go about it. One is to parse the regular expression into an intermediate tree structure first. This makes it straightforward to distinguish a case of "`RC`" (where a $\langle term \rangle$ consists of two single-letter $\langle atom \rangle$ non-terminals and which renders as a single box) from the case of "`RC+`" (where a $\langle term \rangle$ consists of one letter $\langle atom \rangle$ and one $\langle atom \rangle$ that uses "`+`" operator).

Alternatively, a plain recursive descend can be modified to include a lookahead to distinguish those two cases. It allows combining parsing and generation of the resulting boxes into a single recursive algorithm without intermediate data structures.

Problem B. Button Lock

Problem author and developer: Artem Vasilyev

Let's build a directed graph on given sets of digits. There is an edge $A \to B$ iff A is a subset of B. The problem can be now restated as follows: cover all vertices by disjoint paths, the cost of the path is the size of the last set in that path plus one (let's pretend that we will reset after each path, and then remove the last "`R`").

The path cover problem is usually solved by bipartite matching: we create a bipartite graph with two parts corresponding to vertices of the original graph, and make edges $A \to B$ into an undirected edge from A_L to B_R. The maximum matching in this graph corresponds to the path cover with the minimum number of paths (equivalently, the minimum number of edges).

But in this problem, paths have cost: the size of the end vertex plus one. Each vertex that is at the end of a path corresponds to the vertex in the left part that is not covered by the matching. Thus, we need to find the matching in the bipartite graph, such that the cost of non-covered vertices in the left part is minimized, equivalently, the cost of covered vertices is maximized. This problem can be solved by a greedy algorithm: let's sort all the vertices in the order of decreasing cost, and run Kuhn's algorithm in this order. If a vertex gets a match during Kuhn's algorithm, it never gets unmatched later and vice versa. Afterward, build paths from edges in the maximum matching.

The bipartite graph contains at most 2^d vertices in each part and at most 3^d edges, so the pessimistic upper bound on the runtime is $O(6^d)$, but Kuhn's algorithm is often faster than that, especially on graphs with a special structure.

Problem C. Cactus Not Enough

Problem author: Vitaly Aksenov; problem developer: Andrei Lopatin

First of all, let's find a condition for a graph being a strong cactus. Consider all 2-vertex-connected components of the graph. If a graph is a cactus, each of them is either a cycle or a single edge. If there are two single-edge components that share a vertex, the graph isn't a strong cactus, because an edge between other vertices can be added. On the other hand, if there are no such components, no edge could be added, because a path between the ends would cover at least one edge, which already lies on cycle, and this edge

would lie on two cycles if we add the first edge. A more convenient form of the same statement is "a cactus is strong, iff after removing all edges lying on cycles, each connected component has at most one edge."

Consider a set of edges e_i we add to the graph. Let p_i be any simple path on the initial graph between ends of edges e_i. The graph should still be a cactus, so none of this path can go through any edge lying on a cycle. So, these paths are unique, and the problem is independent of each of the connected components after removing all cycles. Also, these paths must not intersect, or an edge lying on two of them will lie on two cycles. So, our problem is split into several instances of the following problem: given a tree, find the minimum number of paths after removing which no two edges would be incident.

This problem could be solved either by dynamic programming or by a greedy algorithm. Dynamic programming is quite straightforward on the state of the subtree, the prefix of sons already covered, did we cover edge from the parent, do we have a not covered edge. But implementing it (especially building the answer itself) is quite error-prone, so let's focus on the greedy algorithm.

In fact, we need to remove some matching from a tree, and after that, cover the tree with paths. The number of paths required to cover the full tree is equal to the number of vertices with an odd degree divided by two. This can be done by making each edge a single path and then joining paths in each vertex in any way until it has at most one end of a path. There would be exactly one path end at each odd-degree vertex after that. So, we need to find a matching minimizing the number of odd-degree vertices. Edges, which have at least one even-degree end can be removed from this matching because they do not decrease the number of odd-degree vertices. So we only need just to find the maximum matching on vertices with odd degree, which can be done with a greedy algorithm, because our graph is a tree.

In fact, one can just go from bottom to top, joining paths in any way, and creating a new path, only when it must be created. It can be shown, that this greedy algorithm will leave uncovered exactly this greedy-found maximum matching on odd-degree vertices, and some edges between unmatched odd-degree vertices with even-degree, which doesn't change the number of paths.

Problem D. Digits

Problem author and developer: Pavel Marvin

There are several solutions to this problem. We'll try to explain the simplest one.

First, let's notice that if d is odd, then we can ignore all even a_i. Similarly, if d is not divisible by 5, then we can ignore all a_i divisible by 5. Now let's add all the remaining numbers in the set and look at the last digit of the product. If this digit is d, then we are done. If not, let's try to remove some numbers from the set to make the last digit equal to d. It's easy to see it's never optimal to remove more than 3 numbers since there are only 4 different remainders modulo 5. Now we can use the following dynamic programming. Let $d[i][j]$ be the **minimal** possible product of numbers **not taken** into the set for the first i elements and last digit j. This DP can be easily calculated in $O(n \cdot 10)$ time.

Problem E. Equilibrium Point /\/\

Problem author and developer: Mikhail Dvorkin

First, let's definitely leave non-integer arithmetics, as it is slower and error-prone.

When working with some sequence, the state describing everything we will need for future usage is (m, x, y), where m is its mass, x is x-coordinate of its center of mass, y is y-coordinate of its center of mass.

However, x and y are not necessarily integer, although rational. To reach the only-integer world, let's rather store: $(m, xm, 3ym)$. There three numbers store the same information but are integer.

Another approach is to switch from diagonal-style diagrams to Young diagrams. This is done by a simple $45°$ rotation, and in this world for each subdiagram you need to store just (m, xm, ym), without an ugly coefficient 3.

Now when we are in the integer world, we can think of the process of generation of all bracket sequences together with their center-of-mass information.

Remember that one of the formal grammars describing bracket sequences is $S \to (S)S|\varepsilon$. So to obtain sequences of length n we can pair up in the "$(S)S$" manner all sequences of lengths $2i$ and $n-2i-2$, for all i. The new center-of-mass information is not so ugly, here's the formulas for the new triple if the left sequence has length $2i$:

```
 left sequence of length 2i
/                          \right sequence of length n-2i-2
```

- $m_{new} = m_{left} + m_{right} + 2i + 1$;
- $xm_{new} = xm_{left} + m_{left} + xm_{right} + (2i+2)m_{right} + (i+1)(2i+1)$;
- $ymTripled_{new} = ymTripled_{left} + 3m_{left} + ymTripled_{right} + 3i + 1$.

We can now generate all bracket sequences of length n and check each one if it has the desired center of mass.

This process is exponential, but as soon as we store only sequences with distinct center-of-mass information, it becomes polynomial. Indeed, each element of $(m, xm, 3ym)$ has a polynomial number of possible values, as $m \le n^2$, and $x, y \le n$.

Here's the data for the numbers of center-of-mass distinct sequences:

- length 32: 478615 distinct out of 35357670;
- length 34: 841858 distinct out of 129644790;
- length 36: 1427986 distinct out of 477638700.

The problem can be solved by doing all of the above in a neat time-saving code style: storing center-of-mass information and sequences efficiently, and doing as little non-integer arithmetic operations as possible.

Or, you can use meet-in-the-middle approach. Calculate all sequences, not necessarily with zero balance, of length $n/2$, and for each left half, try to obtain a right half that fits exactly. That is, iterate over all possible masses of the right half, and having fixed this mass, you know the x and y of the right half that you need, so you can look it up in the map quickly.

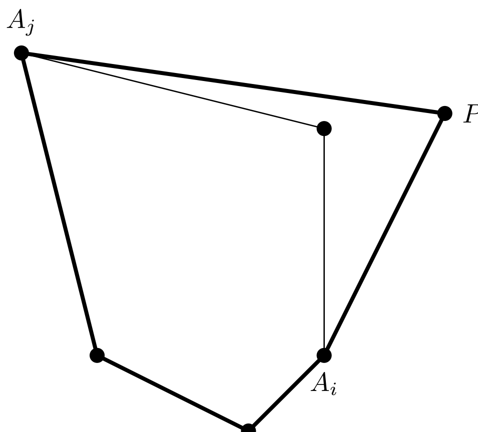
Finally, quite different working approach is:

- Select the set of column heights such that the y coordinate of the center mass is the desired one. Since we're not caring about x coordinates here, this can be done by full search. Actually, there are several appropriate sets, so for each of them:
- Find their horizontal allocation such that the x coordinate of the center mass is the desired one. Once again, the search space is not so large, so a full search will work.

Problem F. Fiber Shape

Problem author and developer: Artem Vasilyev

Consider some specific direction (for example, the positive direction of X axis), and find the extreme point P in that direction. There are two support points, A_i and A_j, that determine the shape of the curve in the neighborhood of P: for every point, the sum of distances to i and j is constant and equal to l minus the part of the polygon from A_j to A_i. This shape is an ellipse with foci A_i and A_j. The major and the minor axis can be determined from the sum of distances.



Thus, the entire curve consists of pieces of ellipses. When does the curve switch from one ellipse to another while moving counterclockwise? This happens when support points change: either A_{i+1} becomes the next support point, either A_j is no longer a support point. This happens when points A_i, A_{i+1} and P or A_j, A_{j+1} and P are on the same line. We can determine when this happens by intersecting the rays A_iA_{i+1} and $A_{j+1}A_j$ with the current ellipse and taking the closest intersection point (and changing the corresponding support point). We can go around the polygon and compute all the ellipse pieces that form the curve in $O(n)$ time. It's convenient to start the process when $i = 0$. To do that we find the furthest point of form $A_0 + \overrightarrow{A_{n-1}A_0}t$ with a binary search on t and checking the length of the perimeter.

To find the area of the curve we will use the Greene's formula $\frac{1}{2}\oint\limits_C x\,dy - y\,dx = \frac{1}{2}\oint\limits_C (x(t)y'(t) - y(t)x'(t))\,dt$.

By additivity, this integral breaks down into integrals over ellipse pieces. We found that this angular parametrization works best with this integral: $P(\alpha) = O + \overrightarrow{e_1}A\cos(\alpha) + \overrightarrow{e_2}B\sin(\alpha)$. Here, O is the center of the ellipse, A and B are the lengths of the major and minor axes, and e_1 and e_2 are unit vectors corresponding to major and minor axes. Let's simplify the formula inside the integral $x(t)y'(t) - y(t)x'(t) = (x(t), y(t)) \times (x'(t), y'(t))$, where $A \times B$ is the cross product of vectors A and B.

$$(x(t), y(t)) = O + \overrightarrow{e_1}A\cos(\alpha) + \overrightarrow{e_2}B\sin(\alpha)$$
$$(x'(t), y'(t)) = -\overrightarrow{e_1}A\sin(\alpha) + \overrightarrow{e_2}B\cos(\alpha)$$
$$(x(t), y(t)) \times (x'(t), y'(t)) = -(O \times \overrightarrow{e_1})A\sin(\alpha) + (O \times \overrightarrow{e_2})B\cos(\alpha) + AB$$

Then, $\int\limits_{\alpha_1}^{\alpha_2}(-(O \times \overrightarrow{e_1})A\sin(\alpha) + (O \times \overrightarrow{e_2})B\cos(\alpha) + AB)\,d\alpha$ can be easily calculated.

Problem G. Guide

Problem author and developer: Vitaly Aksenov

Suppose for a moment that we have to visit exactly k vertices and return to the capital. In this case, it can be proven that the minimal length of the traversed path should be $2(k-1)$. However, in our problem we do not have to return to the capital: thus, the best way is to stop the traversal at the deepest vertex possible. So, we have to find a vertex v which is the deepest and which depth does not exceed k. Then, we have to simply restore some such path. To do that, for example, we count the number of vertices on the path to v and run a depth-first search from the vertices on the path until we visit k vertices.

Problem H. Hard Optimization

Problem author and developer: Gennady Korotkevich

A *laminar* set of segments clearly forms a forest-like structure: outer segments (not contained in any other ones) are the roots of the trees, segments that are only contained in outer segments are their children, and so on, recursively.

Recovering the tree structure can be done using sorting and a stack. Go through all segments in increasing order of L_i, and keep a stack that describes the current rightmost path in the tree. For each segment, pop the segments that are fully on the left of it from the stack. After that, we know that the top of the stack is the parent of this segment (if the stack is empty, this segment is the root of a new tree), and we push the segment onto the stack.

On these trees, we are going to use dynamic programming. The basic idea is that if some segment i has children $s_1, s_2, \ldots, s_k$, its optimal subsegment position is one of the following:

1. to the left of segment s_1, possibly extending onto the prefix of segment s_1;
2. between segments s_j and s_{j+1} for some j, possibly extending onto the suffix of segment s_j and the prefix of segment s_{j+1};
3. to the right of segment s_k, possibly extending onto the suffix of segment s_k;
4. strictly inside segment s_j for some j.

We will calculate DP from bottom to top. Our DP state will have the following variables:

- integer i: current segment (subtree root);
- boolean h_l: whether there's currently a subsegment ending at l_i;
- boolean h_r: whether there's currently a subsegment starting at r_i;
- integer k: how many subsegments should be placed inside segment i's subtree.

To make transitions, we can use another DP going through segment i's children. For each of the $k+1$ areas (to the left of s_1, between s_j and s_{j+1}, to the right of s_k) we should decide whether there's a subsegment covering that area, and for each of the k segments, we should decide how many "outer" subsegments go into their subtrees.

Finally, note that k in our DP state can be limited by the size of segment i's subtree, because we can always cover the whole segment i having at least that many "outer" subsegments. Therefore, merging DP tables of two subtrees can be done in time proportional to the product of their sizes, and the overall complexity of our DP can be bounded by $O(n^2)$.

Problem I. Is It Rated?

Problem author and developer: Petr Mitrichev

See https://en.wikipedia.org/wiki/Randomized_weighted_majority_algorithm for more details.

Intuitively, we want to place more trust in the predictions of those participants that have already made fewer mistakes. The most radical version of this would be to only trust the participants with the smallest number of mistakes in any given wager; when there are multiple such participants and their predictions differ, we can, for example, choose the prediction which has the most such "best participant" votes behind it. And in case even those are tied, we can flip a coin.

However, it turns out that this solution is not good enough, and trusting just the best participants can backfire. For example, consider the following two wagers with two participants, which can be easily generalized to any even number of participants: 01 (correct 1), then 10 (correct 1). Each participant has made one mistake, while we have made 0.5 mistakes in the first wager on average, plus 1 mistake in the second wager (because the participant who was correct in the first wager makes a mistake there), so we're going to be 1.5 times worse than the worst participant if this is repeated many times, which is not good enough.

Therefore, we need to make two improvements to the aforementioned solution:

- Instead of just considering the votes of the participants that have made the smallest number of mistakes only, we will assign weight β^x to the vote of a participant that has made x mistakes, where $0 < \beta < 1$ is some value that we'll choose later.
- Instead of choosing the prediction that gets the most (weighted) votes, we will choose the prediction randomly, using the fraction of the total vote weight that favors this prediction as the probability of us choosing it.

You can check the "Analysis" section of the Wikipedia article mentioned above for the (relatively straightforward) proof that this approach is now good enough for the appropriate values of β. $\beta = \frac{3}{4}$ passes with a margin of 6 standard deviations, but any value of β between roughly $\frac{1}{2}$ and $\frac{19}{20}$ was good enough.

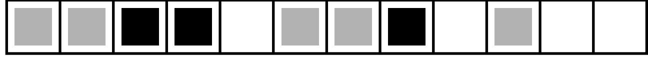
Note that in order to avoid floating-point underflow, we should use β^{x-y} instead of just β^x as the vote weights, where y is the smallest number of mistakes that any participant has made so far.

Note that the judges are not aware of any deterministic solution that passes in this problem. The approach mentioned in the beginning, where we choose the prediction that has the most weighted votes instead of picking randomly with the appropriate probabilities, ends up being almost twice as worse in the case where the votes are split $51/49$ in favor of the wrong prediction every time.

Problem J. Japanese Game

Problem author and developer: Dmitry Yakutov

One can calculate mask m of profile p with the following algorithm. Let's place sets of consecutive blocks in p as left as possible. Let k be the number of empty cells at the end of the row of n cells after the placement (maybe 0). Then we should erase k first filled cells in every set of p. If some set consists of less than k cells then erase all the cells in the set.



Example with $n = 10$. Black cells represent mask m. Grey cells represent the cells we've erased while calculating m. $k = 2$ in this case.

We are given mask m. If we know the exact profile p then m can be found by the algorithm above. Let's iterate over values of k in this algorithm. It is bounded by the following numbers:

- Number of empty cells at the beginning of m;
- Number of empty cells at the end of m;
- $a_i - 1$ where a_i is the number of empty cells between i-th and $i+1$-th sets of consecutive filled cells in mask m.

Let's try to create a profile p with mask m with the specified value of k.

Every set of consecutive filled cells in m should be extended to the left with k filled cells. k last cells of the row of n cells should be empty. Let's cut them out. Now we have some sets of consecutive empty cells at the beginning, at the end, and between sets of filled cells.

If there is exactly one empty cell at the beginning of the row, then such a value of k is not suitable. Otherwise, let's try to fill some of these cells with sets of 1 and 2 consecutive cells. Note that we can use sets of 2 cells only if $k \geq 2$.

The same algorithm can be used at the end of the row. Also, it can be used between sets of consecutive filled cells: if there is exactly one empty cell between sets, then it is already filled, and if there are exactly two empty cells, then it is impossible.

It gives us a solution with $O(n^2)$ time complexity. In the solution above, we haven't used more than 2 consecutive cells, so it can be improved as follows. If $k \geq 4$ and we have found the exact profile p then there exists a profile p' with $k' = k - 2$. All we need is to add a set of single-filled cells between every pair of adjacent sets of m and at the end of the row. It means there is no need to check values of $k \geq 4$, so the resulting complexity is $O(n)$.

Problem K. King's Task

Problem author and developer: Pavel Marvin

First, let's notice that both operations are self-inverse, i.e. if you apply the same operation twice, you get the initial permutation. So, if you make a sequence of operations, these operations must have alternating types.

Let's fix the type of the first operation, and simulate the process until we either find the sorted permutation or return to the initial permutation.

How many operations can we do until we enter the loop? If n is even, then the length of the cycle is 4. For example, the first element of the permutation moves in the following cycle: $1 \to 2 \to (n+2) \to (n+1) \to 1$.

If n is odd, then the length of the cycle is $2n$. For example, the first element of the permutation moves in the following cycle: $1 \to 2 \to (n+2) \to (n+3) \to 3 \to 4 \to \ldots \to 2n \to n \to (n+1) \to 1$.

So it's always enough to make $\max(2n, 4)$ steps to find the sorted permutation or return to the initial permutation.

The time complexity of this solution is $O(n^2)$. It is possible to implement a faster algorithm, but we decided to make this problem easier.