A

A. Arrange the Vertices

Arrange vertices of a graph on a circle of radius R so that no two vertices are at distance smaller than 1, and no two edges cross.

A. Arrange the Vertices

If the vertices are spaced evenly on the circle, the distance between adjacent ones is $2R \sin \frac{\pi}{n}$. If this is at least 1, than just place vertices evenly in their order along the circle, otherwise no placement is possible. We now only have to think about ordering of the vertices.

If the graph is disconnected, consider all its connected components separately. If the (connected) graph is a tree, then it is trivial to place the vertices accordingly (preorder of the tree works, for instance). Otherwise, locate any cycle c in the graph. Vertices of c have to be located in the same order along the circle (say, clockwise).

A. Arrange the Vertices

We can now check if edges connecting the vertices of c do not intersect (if they do, there is no answer). All other vertices have to be located in gaps between adjacent cycle vertices.

To define which vertex goes into which gap, we determine which vertices of c are reachable from any vertex v (without going through vertices of c). If two non-adjacent vertices of c are reachable from the same vertex v, then there is no answer.

Otherwise, for each v there is either one reachable vertex of c, of two adjacent ones. Vertices satisfying the first option form biconnected components adjacent to an articulation point u lying on c. We can process each component recursively, and place them next to u on the circle in arbitrary order.

A. Arrange the Vertices

For each gap between vertices u, w of c, let $C_{u,w}$ be the subgraph of vertices v such that both u and w are reachable from v. We process $C_{u,w} \cup \{u, w\}$ recursively as well while merging u and w into a single vertex.

A recursive call for a graph of size m we perform $O(m)$ operations (for different DFS'es) and make further calls for graphs of total size $\leqslant m$. This implies that the total complexity of the algorithm is $O(m^2)$.

B

B. Bad String, Good String

There is a string s, with some positions marked as bad. Count the number of distinct substrings with at least one occurence containing at most one bad position (and not starting or ending with a bad position).

B. Bad String, Good String

Consider maximal substrings $t_1, \ldots, t_k$ of s containing at most one bad position. For example, if $s =$**abcdef**g (where bold letters mean bad positions), the maximal substrings are `a`bc, `cde`, `efg`. Note that the total length of these substrings does not exceed $2|s|$.

A substring is good if it is a substring of at least one of $t_1, \ldots, t_k$, unless all of its occurences in $t_1, \ldots, t_k$ start or end at bad position (hence `a`b is not a good substring of **abca**b).

Construct a suffix automaton of $t_1 \# \ldots \# t_k$. We can now easily count the number of distinct substrings of $t_1, \ldots, t_k$ as the number of paths from the root without # characters. Now we have to count those of them that do not always start or end at bad positions.

B. Bad String, Good String

We will do that by explicitly substracting those bad occurences. Recall that each state of the automaton corresponds to a range of suffixes of a substring u of s; in addition to the number of occurences of u, we are going to store a collection of queries "decrease the number of good occurences in a subrange by 1".

To subtract occurences that start at a bad position of t_i, process them by increasing of length by traversing the automaton. Each of these decreases the number of occurences for a single string in a corresponding state.

The occurences that end in a bad position are suffixes of each other. Locate the largest of them, and decrease the number for this occurence and all smallest in the same state; after that, traverse the suffix links and process all smaller bad occurences in the same way. Note that since the total length of t_i is $O(|s|)$, this will take linear time.

B. Bad String, Good String

Now, for a state with k occurences, count the number of positions that are covered with less then k queries. Note that substrings with # characters in them have to be ignored (even if they are in the same state with actual substrings).

The above can be done with a scanline in each of the states. Since the number of queries is $O(|s|)$, this solution has complexity $O(|s| \log |s|)$, but can be optimized to $O(|s|)$ since all queries are either concerned with a single character or with a prefix of the range.

C

C. Circle

You are given a triangle ABC and a circle of radius R inscribed in the angle $\angle BAC$. In a single

operation, replace the circle with the one having a common tangent from one of the vertices and inscribed into the corresponding angle. Find the radius of the circle after n operations.

C. Circle

This operation is periodical with period 6. Indeed, note that affine transformations of the picture preserve the operations. Find an affine transformation of the triangle ABC with an equilateral one.

The first operation will move the circle from angle A to angle B, and change its radius from R to R'. The next operation will move it from angle B to angle C, and change the radius back to R by symmetry, and so on. Clearly, after six operations we arrive at the original circle, hence the same thing must happen in the original triangle as well.

Perform n mod 6 operations manually. Complexity $O(1)$.

D

D. Decoration

Construct the shortest chain of rings going between left and right walls and avoiding a set of rectangular pictures.

D. Decoration

Construct the Minkwoski sum of each picture with a circle of radius 0.5 (it's a to be a rectangle with quarter-circle at each corner). Now we want to find a sequence of points at distance 1 from each other that doesn't enter the "rounded" rectangles.

Now (roughly) try all ways to construct tangents between rectangles and choose the best; we can do it in almost any way since then number of rectangles is very small. The hardest part is optimizing the sequence near the corner circles.

Consider the last point on the horizontal side of the left rectangle (not on the quarter circle). We can successively obtain loci of i-th point from there. The first locus is a circle arc (trimmed at the rectangle), and the subsequent loci can be unions of circle arcs. We stop this process when one the arcs reaches the desired side of the right rectangle. Complexity $O(???)$.

E

E. Explosion

Find the shortest distance from point A to a point not visible from a point B in presence of polygonal obstacles.

E. Explosion

The hiding point is either infinitely close to a polygon vertex, or lies on a tangent from B to a polygon vertex. A polygon vertex can be a hiding spot if one of the adjacent sides is not visible from B.

Note that each segment of a route either connects two vertices of polygons, or is a perpendicular to a tangent from B to the polygon. Thus, we construct a graph on all polygon vertices (as well as point A). A pair of vertices is connected if it is possible to go from one to another without meeting obstacles.

Compute shortest path from A to all vertices, then try all segments from a vertex to tangent. The complexity is $O(S^3)$ per test, where S is the total number of vertices.

F

F. Favorite Restaraunts

There is a directed graph on n vertices. We can start at any vertex, and move from a vertex v to any vertex reachable from v (probably, in more then one step) without repetition. When we arrive to u from v, if v is reachable from u, we pay x_u, otherwise we pay y_u. For each k, find the smallest cost to visit exactly k restaraunts.

F. Favorite Restaraunts

Construct the condensation of the graph. The conditions imply that we pay x_u whenever we stay in the same SCC (strongly connected component), and y_u otherwise.

For a single SCC i, compute $s_{i,k}$ — the answer if we are restricted to stay in the SCC i. To compute $s_{i,k}$ we try all options for first vertex v (which will contribute y_v) summed up with $k-1$ smallest values of x_u among vertices u in the same CSS.

Now, compute $d_{i,k}$ — the complete answer if we have to start in the CSS i. Clearly, $d_{i,k} = \min(c_{i,k}, \min_{i \to j, 1 \leqslant k' < k} c_{i,k'} + d_{j,k-k'})$, where the summation is over all CSS j reachable from i.

The total complexity is $O(n^2)$.

G

G. Groups and Teams

We have a set of programmers and mathematicians, split into equal number of groups. Students in each group are familiar with each other and with all students of the corresponding group. Also, heads of all groups are mutually familiar. We don't know the groups, but only which pairs of students are familiar. Split the students into teams of three mutually familiar students so that each team has at least one programmer and at least one mathematician.

G. Groups and Teams

If there is only one group of each type, then the graph is complete and we can solve with a greedy algorithm.

Otherwise, let's determine which students are heads of their groups. To do that, for every edge e of the graph, try to delete e and check if the graph is connected. The conditions imply that the graph becomes disconnected if and only if the endpoints of e are heads of two corresponding groups.

Further, by looking at the disconnected components we can determine the group distribution as well.

G. Groups and Teams

The (pairs of) groups are almost independent since between groups only heads can share a team. More specifically, let p and m be the number of programmers and mathematicians in a corresponding pair of groups. Then:

- $(p + m) \equiv 0 \pmod 3$: all students (including heads) of this pair of groups must be divided into teams. This is possible iff $p \leqslant 2m$ and $m \leqslant 2p$ (again, with a greedy algorithm).
- $(p + m) \equiv 1 \pmod 3$: one of the heads must not be included into teams of this pair of groups. If $p > 2m + 1$ or $m > 2p + 1$, then there is no answer. If $p = 2m + 1$ ($m = 2p + 1$), then this must a head of the programmer (mathematician) group, otherwise the choice is arbitrary.
- $(p + m) \equiv 2 \pmod 3$: neither of the heads can be included into teams of this pair of groups. This is possible iff $p - 1 \leqslant 2(m - 1)$ and $m - 1 \leqslant 2(p - 1)$.

G. Groups and Teams

Hence, we have heads of several groups that are left out no matter out, and in some groups we can choose the type of the head arbitrarily. We have to make the decisions so that the greedy algorithm works (that is, neither $p > 2m$ nor $m > 2p$). After that, reconstructing the answer is straightforward (provided one exists).

The complexity is $O(|E|^2)$.

H

H. Hacker Neo

Perform several operations with a string. An operation is "change the right half of a substring so that it becomes a palindrome"

H. Hacker Neo

Use persistent treap to store the string. To make an operation, we effectively have to insert a mirrored copy of a substring in place of another substring. Since persistent treap allows for split-

ting/mergine, copying parts (= reusing the same vertices) and group modifications, this is a straightforward application.

It is possible to have memory issues with this solution, any standard approach (garbage collection/rebuilding the structure from scratch every k steps) can help.

The complexity is $O(n + m \log n)$ time and memory.

I

I. Interesting Game

Two players play a game on a graph. There are peaceful and aggressive vertices, and there is a certain number of knights in each vertex. On each turn we can either redistribute knigts in a certain way between adjacent peaceful vertices, or attack a peaceful vertex from an adjacent aggressive vertex (destroying knights there). Among all 2^n distributions of vertices into peaceful and aggressive, find the number of wins for the first player and the second player.

I. Interesting Game

First, note that if there are two adjacent peaceful vertices, then the game is a draw. Indeed, any of the players can maintain so that both of these vertices have a positive numbers of knights by redistributing, hence the game will never end.

If no two peaceful vertices are adjacent, then the game can be considered a direct sum of individual peaceful vertices. We can compute Grundy values for each vertex (assuming all its neighbours are aggressive). The values will never exceed 40.

The problem now is to compute the number of independent sets in the graph, and out of these count the number of sets with total Grundy value 0.

I. Interesting Game

First, let us simply count the independent subsets. Divide the vertices into two parts A and B. Try all independent subsets I of A directly (in $O^*(2^{|A|})$); each of them confines vertices in B to a subset B_I. The answer is $\sum_I i(B_I)$, where $i(S)$ is the number of independent sets confined to S. This can be computed in $O(|A|2^{|A|} + |B|2^{|B|})$ with fast subset convolution (or, less efficiently, in $O(|A|2^{|A|} + 3^{|B|})$).

To account for the Grundy values we simply apply the previous approach for independent sets $I_A \subseteq A, I_B \subseteq B$ with $gr(I_A) = gr(I_B) = x$ for each x. Note that x does not exceed 63.

The resulting complexity is $O^*(2^{n/2})$ (or, alternatively, $O^*(1.529...^n)$).

J

J. Jatlin

There a program with n functions; some functions must be declared before others. The cost of swapping two functions is the product of their lengths. Given an initial order of functions, find the smallest cost of swapping them to obtain a valid order.

J. Jatlin

Let the initial order be $1, \ldots, n$. Note that the total cost for a particular order of functions is the number of inversions between the lines (that is, an inversion between functions $j > i$ is counted with the coefficient $M_j \cdot M_i$).

Now, apply subset DP. For a subset S and a function i that is allowed to be placed immediately after S (that is, all prerequisites of i belong to S), make a transition from S to $S + i$ with the cost of the transition being $M_i \sum_{j \in S, i \prec j} M_j$, where $i \prec j$ denotes dependence between functions i and j.

The complexity is $O(2^n n)$.

K

K. Kings and Rooks

Find the number of way to place three mutually non-attacking rooks on an $n \times m$ chessboard with k obstacles.

K. Kings and Rooks

We apply inclusion-exclusion principle. The total number of ways to place three rooks is $\binom{nm-k}{3}$.

Next, substract the placements where rooks i and j are attacking for $1 \leqslant i < j \leqslant 3$ (here, we ignore the third rook for the purpose of checking whether the rooks are attacking). This number is equal to $(nm - k - 2) \sum_l \binom{l}{2}$, where the sum ranges over lengths of all maximal horizonal and vertical segments. We can construct all the maximal segments by sorting obstacles with the same coordinate together.

Now we need to add back configurations that were subtracted too many times. There are two types of such configurations: three rooks in the same horizonal/vertical segment, or three attacking rooks placed in an L shape (in one of four orientations).

K. Kings and Rooks

The configurations with all rooks in the same segment are currently accounted for with multiplicity $1 - 3 = -2$. Counting these is done the same way as before: $\sum_l \binom{l}{3}$. Add this back with coefficient 2.

The L-shapes are accounted for with multiplicity $1 - 2 = -1$. We will use scanline approach to count them. Let's count the number of L's with the horizontal segment going left (then do the same in the other direction).

K. Kings and Rooks

For each row containing at least one obstacle store the x-coordinate of the last encountered obstacle; also store the number of empty rows between them. For each empty column i containing two of the rooks the answer is $(i-1)n$ minus the sum of x's of last obstacles (place two rooks in the same row), multiplied by $n-1$ (place the third rook). We need to group empty columns together, and the answer for a group is proportional to the arithmetic progression sum over i.

If the column i does contain obstacles, we can obtain the answer the same way for each maximal vertical segment with range sum queries.

The total complexity is $O(k \log k)$.

L

L. Logical Expression

Count the number of satisfying variable substitutions for a 2SAT instance such that no variable (either itself or with negation) appears in at most two clauses.

L. Logical Expression

Construct a graph on variables where two variables are adjacent iff they share a clause (possibly, with negations). This graph has all degrees $\leqslant 2$, hence all its connected components are paths and cycles. Note that the problem is independent for the components, hence we consider them one by one and multiply the answers.

For a path we can apply straightforward DP $dp_{i,j}$ — the number of satisfying substitutions for a prefix of length i s.t. the last variable has value j. To make the transitions, try both options for the next variable and check for conflicts in the unique common clause.

A cycle is handled similarly, except we will have to remember the value for the first specified variable.

The complexity is $O(n)$.

M

M. Matrix

Given a large matrix, perform queries "flip a submatrix with respect to horizontal/vertical axis".

M. Matrix

We are going to maintain a "two-dimensional" doubly linked list. For each element of the matrix, store links to two vertically and two horizontally adjacent elements (if one of these doesn't exist, store -1 instead).

The links parallel to the same dimenstion are not going to have a particular direction (that is, they are unordered); note that we can still traverse any row and column starting from the border of the matrix.

Additionally maintain a pointer to the top left element.

M. Matrix

To flip a submatrix we are going to "cut it out" and then glue it back in another orientation. First, traverse links from the top left corner to arrive at the top left corner of the submatrix.

Then, severe all the links connecting the submatrix with the rest part. For instance, to cut the top links leaving the top row r_1 of the submatrix, traverse the rows r_1 and $r_1 - 1$ in parallel and disconnect elements if they fall in the horizontal range of the submatrix. Do this for all four borders (provided they are not adjacent to the border of the whole matrix).

Finally, to glue the submatrix back, do the same parallel traverse, except we need to start from the opposite side of the submatrix. The whole manipulation takes $O(n + m)$ operations in total, for the total complexity $O(k(n + m))$.