A

A. Aho-Parasick

You are given two trees on the same set of vertices. You have to find a dictionary, such that applying the Aho-Corasick algorithm to this dictionary will yields the trie and the suffix links equal to the given two trees.

A. Aho-Parasick

Let's start with a slow approach and then optimize it. Let's iterate of all possible roots(root is the node corresponding to the empty string). Now, in each iteration we have a known root. How to solve this problem with a fixed root?

A. Aho-Parasick

Orient all edges in the trie away from the root and all edges in the suffix links to the root. Now, each vertex has a parent in the trie and a parent in the suffix links (the direction of the vertex-parent edge is different though). In the trie each edge is assigned a letter.

It's obvious that if two non-root vertices are connected by a suffix link, the letters on their ingoing edges in the trie are the same. Let's consider those equality relation as edges between trie edges (trie edges are vertices in this graph). Trie edges which are in the same connected component obviously have the same letter on them. Lets say that each connected component corresponds to some unique letter. It will be shown later, that if this assignment of letters to edges fail any assignment will.

A. Aho-Parasick

Its obvious that one of the conforming dictionaries (if such exists) is prefixless. Now, since we know the letters on edges, we can uniquely determine the dictionary (every leave is a terminal vertex, others are not). Now, lets apply the Aho-Corasick algrotithm to this dictionary. There are two major possibilites:

1. We will obtain the required trie and the suffix links. Hooray. The problem solved.
2. Something was different. Lets list the possible reasons for that.

A. Aho-Parasick

1. The total length of the dictionary is too large.
2. Two outgoing edges from the same vertex in the trie have the same letters on them.
3. While finding the correct suffix link for some vertex V with trie parent P, the parent of the real suffix links was not on the suffix link path from P.
4. While finding the correct suffix link for some vertex V with trie parent P, with the letter l on the ingoing edge in the trie, we found an edge with an outgoing edge with letter l earlier than we should have, while traversing the suffix link path from P.

From here its easy to see that our assignment of letters to edges is optimal (i.e it's valid if some assignment is valid), because the first and the third possible problems don't care about letters on edge, and the second and the fourth favor diversity in letters.

A. Aho-Parasick

Now we have an $O(NL \log N)$ or $O(NL)$ solution, where N is the number of vertices and L is the maximum allowed total lengths of dictionary, depending on the data structure used for storing the trie edges. The sorted vector has worse complexity than unordered_map/HashMap but is faster.

Now, we have to optimize it. The part where we check the root can be optimized using a persistent segment tree, however this problem is not about this optimization. We will keep the complexity of checking a single root the same, but will lessen the number of roots we check.

A. Aho-Parasick

Let's bound the possible locations of the root. Assume that we know the answer. There is a vertex is a leave in both trees (the one which coressponds to the longest string). So, there will be one in our pair of trees. Lets just remove this vertex V from both trees. There is one problem. This vertex couldv́e been the root. However, this could happen only if there is only a single letter in the whole dictionary. This means that the trie and the suflinks must both be the same bamboo.

Let's just remove the vertices, which are leaves in both trees one by one, until we reach the babmoo state described above. The root must be one of the remaining vertices. We have bounded the possible root locations to a path, which is the same in the trie and in the suffix links.

A. Aho-Parasick

Let's call one of the endpoints of this path L and the other R. Now the vertex which closer to L is considered more "to the left". Assume that the root is vertex X which lies on the path. It can be shown that L-X path consists of edges with the same letter (assume its 'a') and X-R path consists of edges with the same letter but different from 'a' (assume its 'b').

Let's list what could go wrong again. But make some additional observations, because we know that the root lies on the L-R path.

A. Aho-Parasick

1. The total length of the dictionary large. Only one direction (left or right) will lessen the dictionary number. We will need to move the root in this direction.
2. Two outgoing edges from the same vertex in the trie have the same letters on them. It can be proven that this letter is either 'a' or 'b'. If it is 'a' we will have to move the root to the left and if it is 'b' — to the right.
3. While finding the correct suffix link for some vertex V with trie parent P, with the letter l. For each vertex v there is a unique closest vertex c_v, which lies on $L - R$ path. Let u denote the required suffix parent of v. If c_v is more left that c_u we need to move the root to the right, otherwise to the left. It can be proven, that c_v and c_u are not equal.

A. Aho-Parasick

1. While finding the correct suffix link for some vertex V with trie parent P, with the letter l on the ingoing edge in the trie, we found an edge with an outgoing edge with letter l earlier than we should have, while traversing the suffix link path from P. It can be proven that l is 'a' or 'b'. If it is 'a' we will have to move the root to the left and if it is 'b' — to the right.

All of this means that we can uniquely determine the direction to move the root to. Make the binary search on the location on the root on the $L - R$ path. Total complexity is $O(L \log^2 N)$ or $O(L \log N)$

A. Aho-Parasick

The implementation size of this problem plus the problem "Game" is greater then the sum of implementation sizes of the remaing 11 problems.

B

B. Big Numbers

Let J_v be maximum length of journey starting in vertex v and T_v be maximum length of trip starting in vertex v.

Then $J_v = \sum_{v \to to}(w_{v \to to} + T_{to})$ and $T_v = \sum_{v \to to}(w_{v \to to} + J_{to}) + \max_{v \to to}(w_{v \to to} + J_{to})$

Let's store big numbers in binary representation in std::set like structure storing positions of 1 bit.

To add a number of form 2^x to some number you can just find first position y such that $y \geqslant x$ and for all i from x to y bit with number i is set in this number.

B. Big Numbers

You can compare two number with a and b bits sets of bits in $O(min(|a|, |b|) + \log(a + b))$ time.

Let's use smaller-to-larger trick when merging values from sons of vertex.

You can easily add all required values except for maximum bit-by-bit. Totally there will be at most $O(n \cdot logn)$ bit addition, so this part works in $O(n \cdot log^2 n)$ time.

Maximum value should be added twice, so it's required to store some modificator in all values representing their bitwise shifts. This part works in $O(n \cdot log^2 n)$ time too, so total complexity of the solutiton is $O(n \cdot log^2 n)$.

C

C. Cactus Revisited

You are given a cactus. Find its optimal $a : b$ coloring.

C. Cactus Revisited

~~If some of the later tests contained only a single vertex with no edges, the answer would be 1 and a lot of teams would be like WTF is this WA 30 that would be so awesome~~ But, unfortunately $n \geqslant 2$.

The optimal ratio is obviously at least 2 and if the graph is bipartite it is 2. In that case output the 2:1 coloring.

C. Cactus Revisited

If the graph is not bipartite there exists a simple odd cycle of length $2p + 1$. We claim that the optimal ratio is at least $\frac{2p+1}{p}$.

Lets prove this statement. Ignore all other vertices and leave only this cycle.

C. Cactus Revisited

Lets turn this cycle into a chain of $2p + 2$ vertices, but add a constraint that first vertex should have the same color set as the last.

Assume that we have a better $a : b$ coloring. Let s_i be the set of colors i-th vertex recieves. Let f_t denote the size of the intersection of s_1 and s_{2t+1}. Let g_t denote the size of the intersection of s_{2t+2} and s_1s complement.

It can be shown that $g_i >= f_i$ and $f_i >= g_{i-1} + 2b - a \implies f_i >= f_{i-1} + 2b - a \implies f_p \geqslant f_0 - pa + 2pb = b - pa + 2pb = (2p + 1)b - pa \implies g_p \geqslant (2p + 1)b - pa$. Also, because $2p + 2$-th vertex is essentially the first one $f_1 = f_{2p+2}$ thus $g_p = 0$. $0 \geqslant (2p + 1)b - pa \implies \frac{a}{b} \geqslant \frac{2p+1}{p}$.

C. Cactus Revisited

Now we have the lower bound for the optimal ratio $\frac{2p+1}{p}$ where $2p + 1$ is the length of the smallest odd cycle. Let's construct a $2p + 1$:p-coloring.

Build the DFS-traversal tree of the cactus with first being the root. Each vertex will recieve some consecutive set of colors modulo $2p + 1$, i.e $2p, 2p + 1, 1, \ldots, p - 2$ is a consecutive set. The first value of this interval r_v will be called the representative color of the vertex v. For example, for the set described earlier, the representative is $2p$.

Our coloring will be valid if and only if for each edge $(u, v) : |r_u - r_v| \in \{p, p + 1\}$.

C. Cactus Revisited

For each edge in the traversal tree we will assign values d_i, such that if an edge connecting the vertices a and b, where a is the parent of b if first vertex is considered the root recieves the value d $r_b - r_a = p + d$. Initially for such edge d is equal to the height of a taken modulo 2.

Consider an odd cycle of length $2p + 2t + 1$. It contains exactly one additional edge not from the traversal tree. We need to replace exactly p d_is from 1 to 0 on the path between endpoints of the additional edge in the traversal tree.

It can proven that the resulting assignment of d_i's and accordingly r_is and accordingly color sets to the vertices is correct.

C. Cactus Revisited

The total running time is $O(n^2)$ with output being the most time consuming operation.

D

D. Decent Sequence

To solve this problem you need to find out if there is sequence under given conditions which is decent and find out if there is a sequence under given conditions which is not decent.

D. Decent Sequence

To check if there is decent sequence let's for each position find out smallest value on which incresing prefix ending in this position can end and the smallest value on which decreasing suffix beginning at this position can end.

Iterate over all possible positions of maximum value and check if you can build required increasing prefix/decreasing suffix.

D. Decent Sequence

To check if there is a decent sequence let's for for each i check if it's possible to make $a_i \leqslant a_{i+1}$ and if it's possible to make $a_i \geqslant a_{i+1}$. According to this information it's easy to check if all sequences are decent.

D. Decent Sequence

Unfortunately, we are not good at Polish, so some incorrect solutions have passed. We appologize for it.

E

E. Expected Cycle Size

Permutation pattern is a permutation with 0 as a wildcard. You are given a permutation pattern. For each index, find its expected cycle size if a random permutation conforming to the pattern is chosen.

E. Expected Cycle Size

Let's split the indices into cycles and chain. The answer for some index which lies in the cycle is the length of that cycle. Calculate the answers for the indices in the cycles and remove the cycles. Only chains are remaining.

E. Expected Cycle Size

If two indices lies in the same chain they are always in each others cycle. It can be shown, that if they are in two different chains they are in the same cycle with exactly $\frac{1}{2}$ probabilty. This means that if lengths of the chains are $a_1, a_2, \ldots, a_k$, the answer for all indices in i-th chain is $\frac{a_j + \sum_{j=1}^{k} a_j}{2}$. However the $\sum_{j=1}^{k}$ must be calulated only once.

E. Expected Cycle Size

The final complexity is $O(n)$

F

F. Folding

Given a string find its number of arithmetic valid foldings.

F. Folding

Formally we have to find number of representations of the given string as $ASS^rS \ldots SS^rB$, where S^r is a reversed string S, A is a suffix of S^r, ans B is a prefix of S or S^r (depends on parity).

Let's find all positions in which the string can be split. These are positions for which right part is an almost mirrored version of left part (except for lengths). Let's call them foldable positions (we can find them with hashes or Manacher).

If a partition consists of one string then this division corresponds to the empty set from the statement.

If a partition consists of two strings then we divide the string at foldable position.

F. Folding

Consider the case then the partition consists of three or more strings. Consider the S or S^r which goes through the middle of the string. If borders of this string are foldable positions than the partition can be completed and we will obtain a correct representation of the string.

All such choices give different representation (since we choose another string that goes through the middle).

So to get the number of such representations we can just multiply the number of foldable positions in the left part of the string by the number of foldable positions in the right part.

G

G. Game

There is a number x. n players make a single turn one by one. $i-th$ of them either does nothing or makes x equal to $(x + a_i) \mod n$. In the end x-th player wins. Players move iff they won't win if they don't move but will if they move. There are also q modifications.

Lets solve this problem without modifcations.

G. Game

The key observation is that at most one person moves. The proof is simple. All persons that move must win. There can be at most one winner. It's also obvious that 0-th player never moves, altought he may win. If some other (k-th player) wins he must have made a move, and x was equal to 0 before his turn. This means $a_k = k$. Also it can be shown, that after i-th players move $x \leqslant i$.

Lets take look at the $n-1$-th player. Let x_{n-1} be the value of x when the last player makes a turn. The only x_{n-1} at which $n-1$-th player will make a move is $p_{n-1} = n - 1 - a_{n-1}$. This means that p_{n-1} will never win and thus will never move.

Lets extend this observation.

G. Game

Let's keep the values $alive_i$ for each player. $alive_i = true$, means that i-th player will win, if after his turn x is equal to i. $alive_i = false$ means that he won't. Let's iterate the players from last to first.

Lets take a look at the player k. If k-th is not alive we simply skip him. If he is not alive and $a_k = 0$ or $a_k > k$ he will never win. Otherwise let $p_k = k - a_k$. p_k-th player will never win, so he is no more alive if he was alive, i.e assign $false$ to $alive_{p_k}$.

The final winner is the player with smallest index k, such that $alive_k = true$ and $a_k = k$. If there is no such player 0-th player is the winner.

G. Game

What about modifications?

Reprsent the whole structure of the problem by a forest, where k-th vertex has no parent if $a_k = 0$ or $a_k > k$ and has the parent $p_k = k - a_k$. We can calculate the same values $alive_i$ using simply dynamic programming on the tree. i.e, $alive_i$ is true if and only if $alive$ values of its children are all false. This doesn't speed up the solution yet, but is the step in the right direction.

G. Game

Yet again, what about modifications?

Lets split modification $a_x = y$ in two parts.

1. $a_x = \{oldvalue\} \to 0$
2. $a_x = 0 \to y$

How to handle the former?

This operation is basically cutting a subtree from its parent. Lets call the root of the subtree v and it's parent p. Now we have to update values $alive_i$. Which values could change?

The first vertex, where the alive value might change is p. If v was not alive no $alive_i$ values will change.

If v was alive, this vertex has lost an *alive* child. But if it has on more alive child it will still be dead and nothing else will change.

If it doesn't p will become *alive* and we will have to process ps parent p_2. If p_2 had an alive child, it will simply have one more alive child and it will still be dead. If it had not p_2 was alive and will become dead and will have to process p_2 parent in the same way as p.

Its fairly easy to see the pattern here. The general rule is as follows. We inverse $alive_i$ values on the path up from v until we reach a vertex which had an alive child from the different subtree (not the same on as v).

Its fairly easy to see the pattern here. The general rule is as follows. We inverse $alive_i$ values on the path up from v until we reach a vertex which had an alive child from the different subtree (not the same on as v).

The second part of the modification ($a_x = 0 \to y$) is done similarly.

All of this can be handled using Link/Cut tree data structure with boolean user data in nodes and modifications in the form of booleans inverse on a path. We should also maintain for each vertex v the number of alive children which are not in the same Link/Cut tree path as v. Also a set of alive vertices with $p_k = 0$ should be maintained.

The exact implementation and more rigorous proof is up to the readers.

The total complexity is $O((N+Q)\log^2 N)$ or $O(N+Q)\log N$ depending on the BST used inside the Link/Cut tree.

H

H. Hidden Pool

The statement can not be briefly desribed in a single slide.

H. Hidden Pool

It can be shown that the optimal strategy for streamsnipers is as follows: for each of the streamsnipers fix some moment of time. He starts searching at this moment or immediately after the delay if Arthas start searching.

Let's binary search the answer. Let k denote the current possible answer we are checking. We have to answer the following question: Is it correct that streamsnipers have a stratehy which ensures at least k streamsnipers in the same game as Arthas.

H. Hidden Pool

Sort the normal players according to t_i. Lets call a group of $p-k$ consecutive normal players a winning group (if Arthas gets in the same game with them he wins, since there will be no more than $k-1$ sniper in this game) Consider $p-k$ th normal player (p is the number of players in a single game). If the streamer gets in the same game as those $p-k$ players the streamsnipers lose. Let $t_{p-k} = Tx+y$, where $y < T$. If Arthas starts searching before t_{p-k} he obviously should do it at time Tx (still earlier that the last normal player in this winning group).

Now, there are two options.

1. $y \leqslant d$. The streamsnipers won't be able to streamsnipe (search immediately after the delay) Arthas if he starts searching at Tx. The only way for streamsnipers to avoid this is for k streamsnipers to get in the same game as the first $p-k$ players. So, they can simply start searching at $Tx+0.5$ and they will get in the game with the first $p-k$ players or instantly win if they get in the game with Arthas. We can then remove those first $p-k$ normal players with one exception: Arthas is no more able to start searching at the moment Tx. So for example if the next interesting ($2p-2k$-th) player start searching at the time$Tx+y_2$ ($y_2 < T$) we can skip the cases where arthas start searching before that player (i.e Tx). Remove those k players, while keeping track of the number of streamsnipers needed.
2. $y > d$. In that case Arthas is going to get streamsniped if he starts searching at Tx, so he is not able to it.

If Arthas wants to get in this winning group and starts searching after the last player in this group. He obviously should do so at $T(x+1)$. Let l denote the number of normal players to the left of $T(x+1)$. We know, that $l \geqslant p-k$ There are two cases yet again.

1. $l \geqslant p$. In that case first p players will get in the same game. Arthas will not inferee because he's going to get streamsniped and thus will lose. And streamsnipers have no need to interfere. Simply remove first p players.
2. $l < p$. If arthas starts searching at $T(x+1)$ and the game with those l players is not filled with streamsnipers he will win. So, $p-l$ streamsnipers have to fill this game. Remove those l players and keep track that we need another $p-l$ streamsnipers.

In any case we remove some prefix of players (in the order of sorted t_i) and maybe add some amount of required streamsnipers. Simply do this procedure until there are no more normal players left. If we need no more streamsnipers than m streamsnipers can ensure at least k of them get in the game with Arthas, otherwise they can't.

The whole thing for some fixed k can be simulated in $O(n)$ time, so the total complexity is $O(n \log(n+m))$

I

I. Intersection Of Tangents

Given polygon find a point whose tangents to the polygon intersect at a right angle

I. Intersection Of Tangents

One of the possible answers is a point whose x coordinate equals to the minimal x coordinate of polygon vertices and y coordinate equals to the minimal y coordinate. With the definition of the tangent from the statement it is obvious that this point is a correct answer (tangents are parallel to the cooridinate axis).

J

J. Just Another Edge

You are given a planar graph in which no edge can be added without losing planarity. Find the number of edges which can be added with graph being tripartite.

J. Just Another Edge

Every such graph is a triangulation of the plane with the outer face being triangle too.

Every triangulation has no more than one different division into three independent sets.

Also we can colour it into three colours using a simple algorithm:

- Colour some two vertices connected with an edge with colours 1 and 2.
- Until we colour all vertices there will always be a vertex which is uniquely colourable (there will be a triangle in which exactly two vertices are coloured) or a vertex which cannot be properly colored (then the answer is 0).

Now we can add an edge between every two vertices of different colours which weren't connected by an edge in the initial graph.

K

K. Khalin Graph

Given a halin graph find number of its 3-matchings.

K. Khalin Graph

Let's compute dp $dp[s][i][j][k]$: number of 3-matchings for subtree of vertex s such that the leftmost leaf requires i more vertices for its matching from the left, the rightmost leaf requires j more vertices for its matching from the right and the vertex s requires k more vertices for its matching from upper levels.

It is quite easy to recalculate $dp[s]$ using dps from children (during merge iterate through all possibilities of connection between matchings of rightmost leaf of left tree and rightmost leaf of right tree and through and connection between matchings of topmost vertices).

Answer would be the sum of values of dps in the root with $k = 0$ and i, j allowing connection between matchings of leftmost leaf and rightmost leaf.

L

L. Lysergic Acid Diethylamide

$f(x) = 1 + \ldots ... + x = \frac{x(x+1)}{2}$

$s_0(x) = x$

$s_k(x) = s_{k-1}(f(x) + k)$

For some amount of testcases with different p find an incorrect answer for $s_k(x)$ *modulo* p, where test cases is a tuple x, k, p. You are allowed to skip at most 20 test cases.

L. Lysergic Acid Diethylamide

$k = 0$ case is straightforward.

Consider the case where p is odd. $f(x)$ *modulo* $p = x(x+1)\frac{p+1}{2}$ *modulo* p. This means that f(x) is periodic moudlo p with period p. Also $f(0) = f(p-1) = 0$. This means that there exists r, such that $\forall_x : f(x) \neq r$ *modulo* p. This means that $s_1(x) \neq (r+1)$ *modulo* p. So, if p is odd simply find such r by iterating over all possible x s from 0 to $p-1$.

If p is divisible by some odd number p_1 find such r for p_1. $s_1(x) \neq r$ *modulo* $p_1 \rightarrow s_1(x) \neq r$ *modulo* p

If p is not divisible by any odd number p is a power of two. All ps are distinct and there are only 13 possible powers of two under given constraints. Skip those testcases by using -1 wildcard.

M

M. Moving Randomly

The score of an array is defined as follows: you are placed in a random element of an array and if

you are not in a boundary position you can randomly move to adjacent position or stop.

The score is an expected number you will stop at if you will try to maximize it.

Given an array find scores of all prefixes.

M. Moving Randomly

The only way we can improve our score is to optimally choose positions at which we stop. Suppose we chose positions $p_1, p_2, \ldots, p_i, \ldots, p_k$. If we start at position p_i then out expected score is a_{p_i}. Suppose we start at position $p_i < x < p_{i+1}$. Then we will move until we stop at point p_i or p_{i+1}. It can be shown that we will come to point p_i with probability $\frac{p_{i+1}-x}{p_{i+1}-p_i}$. Then our expected score at point x is a value of a linear function that goes through points (p_i, a_{p_i}) and $(p_{i+1}, a_{p_{i+1}})$ at point x.

M. Moving Randomly

Consider upper convex hull of points (i, a_i). Let's stop at positions for which (i, a_i) lies on the boundary of this convex hull. From previous observations it is obvious that this strategy is optimal. To calculate the score for all prefixes we just have to maintain the convex hull in a stack.

M. Moving Randomly

The phrase "You might turn into a goat or the whole game might be optimized out" isn't completely random. The first part is. The second one is a reference to this

http://en.cppreference.com/w/cpp/language/ub#UB_and_optimization