2019 Multi-University Training, Round 8, Tutorial

A. Acesrc and Cube Hypernet

First, we may compute the size of the cube from the number of '#'s in the input.

Pick an aribtrary '#'. For every cell of a face of the cube, try aligning the '#' with that cell, and run flood fill on both the cube and the input. If any mismatch is found during flood fill, the answer is no.

B. Acesrc and Girlfriend

The problem asks the following question: given an edge-weighted undirected graph. For each query, you are given two vertices u, v, among all simple paths between u, v, you should find the minimum possible weight of the second heaviest edge.

Note that to find the heaviest not the second heaviest edge, a minimum spanning tree simply works. Here, we may do something similar: for each query u, v, we add the edge from the lightest one to the heaviest one and merge the connected components, until u, v are adjacent in the original graph. The weight of the last added edge is the answer.

Now we solve this problem algorithmically. The main technique we use is the so-called *small-to-large trick*. Note that, we have a set of edges $E = \{u_i v_i\}_{i=1}^{m}$ and a set of queries $Q = \{u_i v_i\}_{i=1}^{q}$. Define $A_u = \{v_i : uv_i \in E\}$, and $Q_u = \{v_i : uv_i \in Q\}$. Also we define the weight of vertex u as $w_u = |A_u| + |Q_u|$. What we should do is to perform a series of merge operations like Kruskal's MST. To merge vertex u and v $(w_u \le w_v)$, we always insert all elements in A_u and Q_u into A_v and Q_v, respectively. Merging two vertices may yield answers to some queries. How do we detect these queries? For each $x \in A_u$ if $x \in Q_v$, then we obtain the answer to query (x, v), and for each $x \in Q_u$ if $x \in A_v$, then we obtain the answer to (u, x). Due to the small-to-large property every element is visited at most $O(\log n)$ times.

The last thing to do is to map all these abstract data structures to concrete ones. If we use BBST we achieve $O(n \log^2 n)$ time and if we use hash table we may achieve $O(n \log n)$. When implementing, you should pay special attention to the indices in your data structures: keep in mind whether they are the original indices of the vertices or the indices of the representatives of union-find sets.

C. Acesrc and Good Numbers

Let $w(d, x)$ denote the number of digit d in decimal representation of x.

Lemma: every d-good number is less than 10^{11}.

Proof: let $g(d, k) = f(d, k) - k$. A d-good number is a zero point of $g(d, k)$. For $d = 1, 2, \ldots, 9$, we have $g(d, 10^{11} - 1) = 10^{10} + 1$. When $k \ge 10^{11} - 1$, $f(d, k + 10^{10}) - f(d, k) \ge 10^{10}(1 + w(d, \frac{k}{10^{10}})) \ge 10^{10}$, so $g(d, k) \ge 1$ for $k \ge 10^{11} - 1$.

The key observation is, the distribution of good numbers is very sparse. Actually, there are only 661 good numbers. The most brute force way is to compute $g(d, k)$ for every $k \le 10^{11}$, but this takes too much time.

To cope with this problem, one may consider pruning. For $k \in [1, 10^6]$, compute $g(d, 10^5 k - 1)$, where $g(d, 10^5(k+1) - 1) = g(d, 10^5 k - 1) + 10^5(w(d,k) - 0.5)$.

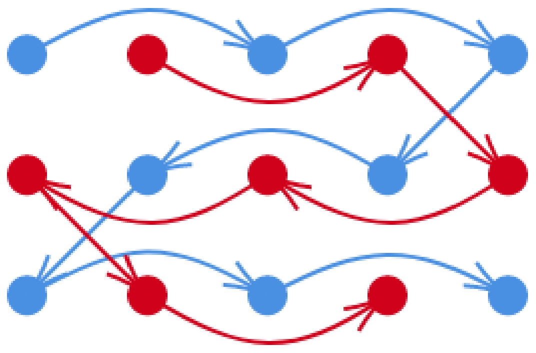
Note that:

- when $w(d,k) > 0$, $g(d,x)$ is nondecreasing in $[10^5 k, 10^5(k+1)]$, and there is a zero point only if $g(d, 10^5 k) \cdot g(d, 10^5(k+1)) \le 0$;
- when $w(d,k) = 0$, it can be verified that there is a zero point in $[10^5 k, 10^5(k+1)]$ only if $0 \le g(d, 10^5 k) \le 10^5$.

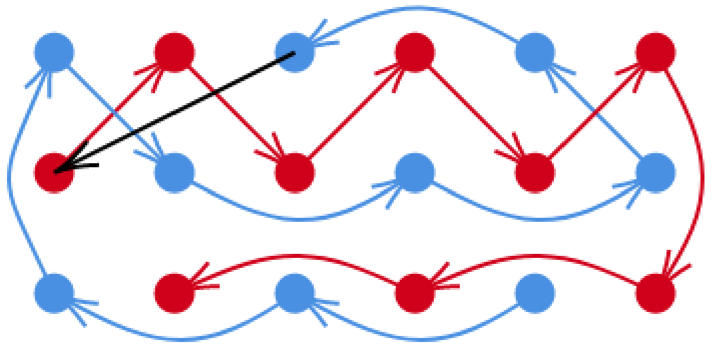
D. Acesrc and Hunting

The only cases where the answer is NO are $1 \times m, n \times 1$ $(n, m > 1)$ and 2×2. Despite the corner case 1×1 which is trivially feasible, we only present one possible construction method for other cases. Assume $n \ge 2, m \ge 3$ wlog.

We color the traps with blue and red alternately. We can easily connect all blue traps and all red traps in zigzag order respectively.



But how can we join the two paths? Just play some tricks on the first two lines.



E. Acesrc and String Theory

The problem asks the number of index pair (l, r), such that $S[l..r] = A^k$ for some nonempty string A.

We enumerate the length of A. For length x, we may find out all maximal substrings $S_x(1), S_x(2), \cdots$ in S such that x is a period of these strings. For example, if `S = abcabcabdabd`, then there are two such substrings for $x = 3$: `abcabcab` and `abdabd`. Every substring of $S_x(i)$ of length kx is splendid.

Also note that the overlapping part of every two such maximal substrings is shorter than x. We may divide the original string into segments of length x. For every two adjacent segments, if they are identical, then they belong to the same maximal substrings; otherwise, they belong to different maximal substrings. In the latter case, one shall further compute their longest common prefix and suffix to decide how long the substrings can extend, both forward and backward. One may use suffix array with sparse table to achieve overall $O(n \log n)$ time complexity.

Note that the case $k = 1$ may require special treatment.

F. Acesrc and Travel

We may weight each spot with the difference of their satisfactory values since only the difference matters.

This is a straightforward tree dynamic programming. Let $f(u, p, f)$ denote the state that they are in spot u, the previously visited spot is p (or 0 if u is the first spot), and f makes the next decision. The state transition can be made as follows:

$$f(u,p,0) = \max_{uv \in E, v \neq p} f(v,u,1) + w_u,$$
$$f(u,p,1) = \min_{uv \in E, v \neq p} f(v,u,0) + w_u.$$

Root the tree arbitrarily. We may compute the values of $f(u, p, f)$ where p is the parent of u with a depth-first traversal. All other DP values can be computed by running another depth-first traversal, where you might need to query the minimum/maximum of a set of numbers except one number. This can be solved by either maintaining prefix/suffix maximum, or just recording the top 2 maximum values. The total time is linear.

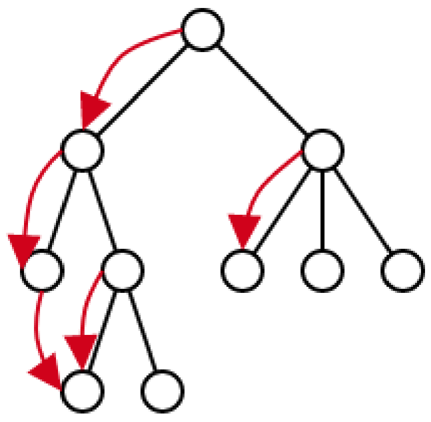
G. Andy and Data Structure

This problem can be solved in $O(n^{1.5})$, though some $O(n^{1.5}\sqrt{\log n})$ or $O(n^{1.5} \log n)$ solutions might also be acceptable. We only describe an $O(n^{1.5})$ solution here.

To achieve $O(n^{1.5})$ time complexity, we break operation 1 into two cases: $x > S$ or $x \leq S$, where $S = \Theta(\sqrt{n})$.

1. When $x > S$, the problem can be done by performing additions on n/x intervals, if we relabel the vertices in BFS order, such that the indices for vertices of the same depth are contiguous. Note that the time budget for each operation if $O(\sqrt{n})$, so we need a data structure (which is left as an exercise) that supports range addition in $O(1)$ time and querying the value of an element in $O(\sqrt{n})$ time. But, how to find these n/x intervals? We may build a forest F according to the following rule: for each vertex u, connect it up to such a vertex v, if exists:
    - v is one level deeper than u;
    - the DFS order of v is minimum, but greater than u;

   Note that the leaves in the original tree are the roots in F. For example, in the following tree, the red arrows form the forest F.



The leftmost k level descendants of u is simply the k level ancestor of u in F. We can use heavy-light decomposition to find the $O(\sqrt{n})$ k level ancestors of a vertex, because the additional $O(\log n)$ cost is dominated by $O(\sqrt{n})$. The rightmost k level descendant can be found likewise.

2. When $x \leq S$, for each modulus x and remainder y, store the vertices whose depth is y modulo x in DFS order. Hence each operation can be done by a single range addition. When querying the weight of a vertex, because there might be $O(\sqrt{n})$ moduli, we need to perform single element queries in the $O(\sqrt{n})$ sequences, each taking constant time. The design of such data structure is again left as an exercise.

Exercise:

1. Design a data structure maintaining an array of elements that supports $O(1)$ time range addition and $O(\sqrt{n})$ time single element query.
2. Design a data structure maintaining an array of elements that supports $O(\sqrt{n})$ time range addition and $O(1)$ time single element query.

H. Andy and Maze

The problem asks to find a shortest simple path with k vertices. We may use a technique called *color coding* to attack this problem.

We color the vertices of the graph with k colors uniformly and independently at random. Assume the optimal answer is path $u_1 u_2 \cdots u_k$. We expect that the colors of these k vertices are distinct. In such case, we may use dynamic programming to find the optimal answer: let $f(u, S)$ denote the longest simple path ending at vertex u, and the set of colors used in the path is S. The transition is as follows:

$$f(v, S) = \min_{u:uv \in E, C(u) \in S} f(u, S/\{C(u)\}) + w_{uv}.$$

This dynamic program runs in $O(m2^k)$.

The last problem: what is the probability that the colors of all vertices in the optimal path are distinct? This is $\frac{k!}{k^k}$. So just try $T\frac{k^k}{k!}$ iterations of the aforementioned algorithm, and you will find the optimal answer with high probability. The total time complexity is $O(T\frac{(2k)^k}{k!}m)$.

I. Calabash and Landlord

Extend the edges into infinite straight lines and remove duplicated ones. The plane is thus divided into at most 5×5 connected regions. For two adjacent regions, if there was originally no edge between them, then unite these two components (you may use union-find). The answer is the number of sets in union-find.

J. Quailty and CCPC

There will be such team if and only if d is odd and $n \bmod 10 = 5$. In such case, the rank of the team is $\frac{d}{10}(n+5)$.

K. Roundgod and Milk Tea

Consider an imaginary bipartite graph of size $(\sum a, \sum b)$, where each left vertex represents a student and each right vertex represents a cup of milk tea, and there is an edge between a student and a cup of milk tea if the milk tea is not made by the student's class. The answer is the size of the maximum matching of the bipartite graph.

Recall the Hall's marriage theorem: for bipartite graph $(U+V, E)$, the size of the maximum matching is

$$|M| = |U| - \max_{S \subset U}(|S| - |N(S)|)$$

where $N(S)$ denotes the neighborhood of S. Let $S' = \arg\max_{S \subset U}(|S| - |N(S)|)$. There are only three possibilities for S':

1. $S' = \varnothing$; in this case, $S - N(S) = 0$, and $|M| = |U|$;
2. all students in S' come from the same class; in this case, S' must contain all students from that class, for otherwise, adding students not in S' but in the same class strictly increases the size of S' without changing its neighborhood;
3. students in S' come from more than one class; in this case, $S' = U$ for the same reason as in case 2, and $|M| = |V|$.

All three cases can be computed in linear time.