1 General tips

To avoid TLE even with correct complexity, use arrays of structs instead of vectors/sets/maps of tuples.

1.1 Naming Conventions

- maxN - max value of a particular variable
- local variables in functions and other global variables need to be lowercase words, uppercase letters if just one character, or camelcased if there's multiple words (including input variables like N, M)
- arrays can be a single lowercase character or follow the same naming convention as local variables
- functions are snake cased

2 Data Structures

2.1 Unordered map

This is comparable to an array with unlimited indices, and operations take amortized $O(1)$ time

It is implemented as a hash table in which indices are mapped to a fixed size array, and they are associated with values.

2.2 Map

Implemented as a self-balancing tree where keys are ordered by some total order, and then they are associated with values. Thus, operations take $O(\log n)$ time.

2.3 Unordered set

It has the same implementation as unordered map but without the values (only keys are stored).

2.4 Set

It has the same implementation as map but without the values (only keys are stored).

2.5 Vector

Implemented as a dynamically-resizing array in ram. Basically, the CPU allocates a certain amount of extra space so that the vector always takes up a contiguous block of memory. Any overflows result in the whole vector being copied to a new location, and so operations like `push_back` take amortized $O(1)$ time.

3 Algorithms

3.1 Divide and conquer

Use this if you find a way to solve ad-hoc problems by seeing that the recursive breakdown of the problem combined with the computation of subproblems (usually involving optimization with a data structure) is faster than the naive way.

Usually involves processing the input a certain way as to allow you to use a data structure like a frequency array, set, or map.

3.2 Graphs

Length of a path: sum of the number of edges.

Shortest distance between two nodes: length of shortest path between them. (for a tree, there's exactly one path between any two nodes)

Depth of a node in a tree: its distance from the root.

3.3 BFS/DFS

A flood fill BFS/DFS (where you perform BFS/DFS on each component) takes $O(V + E)$ time because in total, you are looping through each edge twice, and you are accessing each node once. The same thing applies with one BFS/DFS.

3.4 Center + diameter

Any tree always has a diameter pair (pair of vertices with the longest path) and a center.

When a tree is roooted arbitrarily, all nodes that are furthest from the root are endpoints of a longest path, which allows you to find a diameter pair by doing two DFSes to find a furthest node, then finding the node furthest from this endpoint, which will be a diameter pair.

Center: node with the minimum height (max depth) of the tree when it's rooted.

Radius: minimum height of the tree.

Given a diameter pair, the center is the middle node. Proof: suppose that another node was the center, but it was not the middle node on this diameter. Then consider the tree laid out so that the diameter is a line and every node (except the endpoints) can have a subtree connected to it. Any other node on the diameter is clearly not a center because there exists a path longer than the current radius. The same thing happens with nodes which are not on the diameter: they must connect to a node on the diameter, which has a path that has length $\geq$ the radius, and you are adding a positive value to this length.

If you find a diameter pair (x, y), then every node's maximum distance is equal to the maximum of the distance from that node to x and the distance from that node to y.

Proof: let the arbitrary vertices be u, v. consider the tree with the diameter as a line. You will find that either $d(u, v) \leq d(u, x)$ or $d(u, v) \leq d(u, y)$ due to the fact that if the path from u to v

passes through the diameter, any path from u to a node b on the diameter is less than or equal to $d(b, x)$ and $d(b, y)$.

The minimum number of edges you need to traverse to visit all nodes in a tree is `2*E - diameter`. This is because any efficient tour is made up of a path between two nodes in which you visit all nodes in the subtrees for the nodes in this special path. It follows that the set of lengths of all efficient tours is equal to the set of all $2 * E - l$ where l is the distance between two nodes, and so the minimum total number of edges is `2*E-diameter`.

3.5 Centroid decomposition

Allows you to 'sort' the graph by finding an optimal node to start your algorithm from If the problem can be broken down into an $O(n)$ computation on the current graph combined with the same problem applied to the subtrees, then you can use the centroid to ensure that you only do $O(\log n)$ DFSes on the original graph.

Computing a centroid takes O(size of component) time since you find the size of the subtree with an arbitrary root, then DFS to find the centroid. At each iteration, you don't care about the previous parts of the graph when you find the subtree with $> s/2$ nodes, since the total number of nodes in the previous parts of the graph will be less than $s/2$.

Every graph algorithm is done with respect to a blocked array. We never search into a blocked node, and we set the current centroid node to blocked before searching into the subtrees. This way, the recursion will terminate once you encounter subtrees with only the centroid in the subtree.

Key ideas:

- You can maintain a data structure or precompute values on the centroids so that the centroid tree only has $O(\log(n))$ height.
- Break down path queries to turn them into problems related to subtrees. There are always $O(n^2)$ distinct paths in a tree which is equivalent to the pairs of nodes.
- This strategy also goes hand-in-hand with DP problems that can be broken down into identical problems about the subtrees. The key idea is that normal tree DP without centroid decomposition can only solve recurrences that take O(number of children of the root) time to compute for each call to solve a subproblem, where centroids can solve recurrences that take O(size of current graph) to compute for each subproblem.

3.6 Tree DP techniques

- Try to break a problem into equivalent problems on the subtrees of a node, given an arbitrary root.
- Up-down tree DP technique: use this if your dp array needs to represent data about the entire graph for every node. For example, consider the case where you need to calculate the height of the tree when it's rooted at any node. This problem can be broken into standard tree dp in which you calculate the height of each node's subtree with a fixed root, but you also need to calculate the max distance to a node outside of the subtree `up[i]`, which requires a separate dp array that is accumulated from the parent of a node (`up[i] = max(2+max distance`

`from the parent to a node in the parent's subtree which is not in the original node's subtree, 1+up[parent]`)

3.7 Tree DS techniques

Suppose you have a value associated with each node. Here's how to solve various problems.

- Subtree query, point update: use a tree traversal array to map each node to an index in the array such that all subtrees are represented by contiguous subsequences. Then use a BIT to perform point updates and range queries.
- Subtree update, point query: same idea as above but use a BIT difference array for range updates and point queries
- Point update, path query: precompute the combination of values from the root to any particular node, then use the `ans[a]+ans[b]-2*ans[lca(a, b)]` trick.
- Path (to the root) update, point query: notice how when a node is updated, this also implies that all of its parent nodes should be updated. When the initial value of each node is zero, this means that **you can simply assign the true value of each node to the sum of its subtree**. This translates to point updates and subtree queries.

3.8 Powers of 2 data structure

If you need to perform an associative operation n times and obtain the result, or need to perform an iterated function n times, you can maintain an $O(n \log n)$ data structure that captures the 'function' composed with itself 2^k times. Then, you can query $f^{(n)}(x)$ in $\log n$ time, or even $O(1)$ time if you're dealing with a sparse table.

Examples: Modular exponentiation, matrix exponentiation (X^n, or $\prod_{i=l}^{r} X_i$), lowest common ancestor, successor graph.

3.9 Minimum spanning tree

3.10 Lambdas + Call by reference

In C++, everything except arrays are called by value. Remember not to pass vectors; only pass structs and variables (only things of $O(1)$ size).

We can declare anonymous functions, and then functions can take a lambda as an argument. (Lambdas take $O(1)$ size as well, as they are objects that contain a function that's not executed; it simply takes up a fixed size).

https://medium.com/@winwardo/c-lambdas-arent-magic-part-2-ce0b48934809

Use the capture list to get access to a copy of a particular variable (it is passed by value.)