Trees (+ Successor Graphs)

Definition & Terms

- An acyclic connected and undirected graph
- It contains N nodes and $N-1$ edges
- Every pair of nodes has exactly one simple path between them
- **Forest** : A graph such that each connected component is a tree
- A star graph has two common definitions:
    1. Only one node has degree greater than 1
    2. Only one node has degree greater than 2
- The subtree of a node n are the set of nodes that have n as an ancestor. A node is typically considered to be in its own subtree

For graphs in general : If a component of a graph contains c nodes and no cycle, it must contain exactly c-1 edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

Tree traversal & DP

General Approach for Tree+DP

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. eg:

```
void dfs(int s, int e)
{
// process node s
for (auto u : adj[s])
    if (u != e)
        dfs(u, s);

}
```

The function is given two parameters: the current node s and the previous node e. Unlike DFS on graphs, here you don't require a visited array bcoz it is not possible to reach a node from multiple directions.

- Decide what info you wish to store for each node (DP state)
- Root the tree at node 1 (usually)
- Identify the base case -> usually of form dp[leaf] = something
- Identify the recurrence (transition of states)
- Implementation is generally via DFS -> compute for children and combine to get state of current node

General template

```
bool isLeaf(ll u, vl adj[])
{
    if (u != 1 && adj[u].size() == 1)
```

```
        return true;
    return false;
}

void dfs(ll s, ll e, vl adj[], vl &dp)
{
    if (isLeaf(s, adj))
    { // leaf base case
      // dp[s] = something;
        return;
    }
    for (auto u : adj[s])
    { // computing for children
        if (u == e)
            continue;
        dfs(u, s, adj, dp);
    }
    for (auto u : adj[s])
    { // combining for current node
        if (u == e)
            continue;
        // recurrence relation here
    }
}

void solve()
{
    ll n, a, b;
    vl v;
    cin >> n;
    vl adj[n + 1];
    for (ll i = 2; i <= n; i++)
    {
        cin >> a >> b;
        adj[a].emplace_back(b);
        adj[b].emplace_back(a);
    }
    vl dp(n + 1, 0);
    dfs(1, 0, adj, dp);
}
```

Subordinates:

Code Saved.

We use DP on the recursive DFS fn to get number of nodes in subtree of every node (subtree includes the node itself).

No of subordinates is this count excluding the node itself, hence -1.

T.C : $O(n)$

CF - 1528A

Can See : Nice hard question about identifying DP states and transition.

Diameter:

For graph: Diameter refers to longest of the shortest distances between any pair of nodes. => O(nˆ3) by Floyd Warshall

Difficult to do more efficiently for a general graph

However, it can be found for trees in O(n)

Code saved for both approaches.

The diameter of a tree is the maximum length of a path between two nodes in the tree. (May or may not pass through root)

Obviously, ends of a diameter are always leaves. There may be several maximum-length paths.

Method-1 : DP :

States:

dp[u][0] = max distance b/w any 2 nodes in the subtree of u such that the path b/w those 2 nodes contains u and *has an endpoint at u*

dp[u][1] = max distance b/w any 2 nodes in the subtree of u such that the path b/w those 2 nodes contains u and *doesn't have an endpoint at u*

Base case:

If u is a leaf node, subtree does not have 2 nodes => dp[u][0] = dp[u][1] = 0

Recurrence:

dp[u][0] = 1 + max(dp[v][0]), v belongs to children of u

dp[u][1] : Maximum distance between two distinct children of u, let them be v1 and v2.

then ans = dp[v1][0] + dp[v2][0] + 2

=> dp[u][1] = max(dp[i][0] + dp[j][0]) + 2, where i and j are children of u and i not equal to j

Since we need largest of dp[v][0] of children in both cases, it is a good idea to put dp[v][0] in a vector and sort in descending order.

Final ans = max(max(dp[u][0], dp[u][1])) for all vertices u in the tree.

T.C : Sorting makes it O(n*lgn) though you can also work without sorting in O(n)

Method-2 : 2 BFS :

First, we choose an arbitrary node a in the tree and find the farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

T.C : O(n)

CF - 1083A

Nice hard question based on diameter DP approach, **look only if time**.

Editorial : Adhish K DP on Trees playlist

Binary Tree Points:

- Postorder + Inorder and Preorder + Inorder **uniquely determine** a tree
- *But Postorder + Preorder don't*

Tree Queries:

kth Ancestor:

The kth ancestor of a node x in a rooted tree is the node that we will reach if we move k levels up from x.

Let ancestor(x,k) denote the kth ancestor of a node x (or 0 if there is no such an ancestor).

Naive : O(k) for each query

Binary lifting

Any value of ancestor(x,k) can be efficiently calculated in O(logk) time after preprocessing.

The idea is to precalculate all values ancestor(x,k) where k <= n is a power of two. The preprocessing takes O(n*logn) time, because O(logn) values are calculated for each node.

We use this recurrence to preprocess values via DP :

```
dp[i][k] = dp[dp[i][k-1]][k-1]

where we go 2^k levels up.
```

Clearly, going 2^k levels up is same as going 2^(k-1) levels and again 2^(k-1) levels from where you reach.

After this, any value of ancestor(x,k) can be calculated in O(log k) time by representing k as a sum where each term is a power of two, since such summands are O(log k). If k is even on the order of n as in the "worst case" tree (linked list), we can still answer q queries in this way in O(n log(n)) preprocessing and then O(log(n)) per query for q queries, for a total of O((n+q)*log n)

Company Queries I

Exact idea of binary lifting for kth ancestor. T.C : O((n+q)*log n)

Read code for clarity.

Subtree and path queries

Tree flattening (or Euler Tour)

- Puts nodes into an array based on DFS traversal of tree.

Code Snippet:

```
void Traversal(ll node)
{ // euler tour
    start[node] = timer;
    timer++;    // --> placing here gives subtree as [start, end-1]
    for (auto x : children[node])
    {
        if (x == parent[node])
            continue;
        // timer++; placing here instead gives subtree as [start, end]
        parent[x] = node;
        Traversal(x);
    }
    ending[node] = timer;
}
```

TC : O(n)

- Every node has a unique start value according to which the nodes (or their values) can be sorted.
- For every node i, nodes in its subtree occupy adjacent indices in the new array.
- The subtree nodes occupy range : start[i] -> end[i]-1 (see above code) with root of subtree at start[i] => size of subtree = end[i] - start[i]
- Any queries on array ranges are equivalent to subtree queries and can be supported similarly using different techniques like fenwick, segtree, lazy propagation etc

Subtree Queries

- In the traversal array : we need to support subtree sum => range sum and node update => point update
- That is, we need Dynamic RSQ -> one possible way is segtree
- See code for clarity. TC : O(q* lgn)

LCA

The lowest common ancestor of two nodes of a rooted tree is the lowest node whose subtree contains both the nodes

Method-1 : Binary Lifting

- Two pointers to the desired nodes. Move one pointer above using *kth ancestor idea* so that both pointers point to nodes at the same level.
- If both point to same node => ie LCA, return it
- Else keep moving the pointers up by largest powers of 2 until the nodes have different ancestors
- You will end in a situation where the parent of the nodes pointers currently point to is the LCA, return it
- Each query in O(lg n)

Method-2 : Euler Tour

- We will use DFS to produce a different kind of traversal array.
- We add each node to the array *always* when the depth-first search walks through the node, and not only at the first visit.
- Hence, a node that has k children appears k+1 times in the array and there are a total of 2n-1 nodes in the array.
- We will also store depths of each node in the array alongside.
- LCA(a,b) = Node with min depth between nodes a and b in the array
- That is if no updates are performed, the LCA problem transforms into a *Static RMQ* problem.
- T.C for Static RMQ :
  - Segment Tree : O(n + q*lg n) (preprocessing + queries)
  - Sparse Table : O(n*lg n + q) (preprocessing + queries)

Method-3 : Optimal

- Other methods with O(n) preprocessing and O(1) per query exist but not needed in general
- Can refer cp-algo or CF blogs

Company Queries II

Finding LCA for given nodes.

See code for clarity. T.C : O((n+q)*log n)

Distance Queries

Find LCA and dist = level[a] + level[b] - 2 * level[c]

See code for clarity. T.C : O((n+q)*log n)

Successor / Functional Graphs:

Directed graphs with every node having **outdegree = exactly 1**

succ(x) : where x is a node is a function (every x has a single successor) => hence k/a functional graphs

succ(x,k) : Node that we will reach if we begin at node x and walk k steps forward

Computing succ(x, k):

Naive - O(k) per query

Binary Lifting - O (lg k) per query with O(n lg u) preprocessing where u is the maximum steps queried

Planet Queries I

Computing succ(x, k). See code for clarity.

Vectors give TLE. T.C : O((n+q)*log u), u = 1e9 here