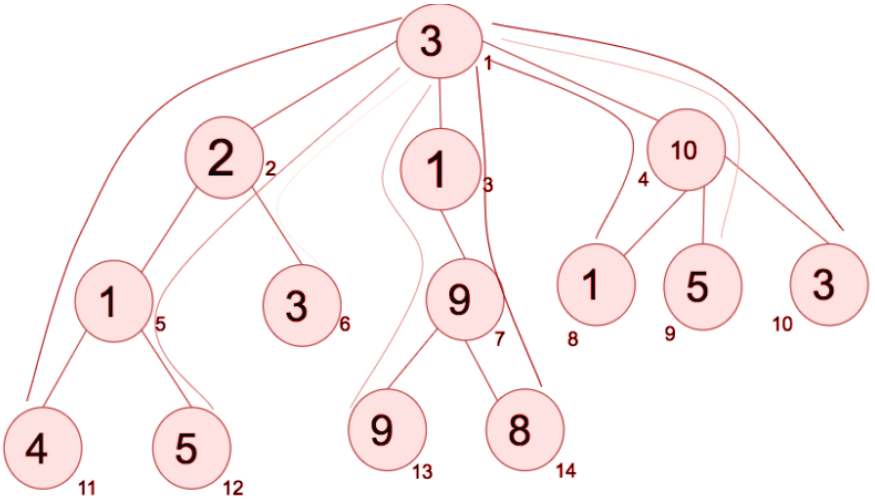
Maximum Sum path in the tree

Problem Statement: Given a tree with N nodes, rooted at node 1, where each node has a positive value associated with it a_u, find the maximum sum path from any node to any other node in the tree.



The maximum sum path in the given tree is (13-7-3-1-4-9) with a total sum of 9 + 9 + 1 + 3 + 10 + 5 = 37.

Naive approach: We can brute force on one end of the path and for each end u, find the end v with the maximum current sum. This can be done by considering each node as a root and maintaining a variable to store the sum from root to the current node and updating the answer each time.
The code for the above approach is as follows:

```
/*
        recursive dfs function that takes a node and current sum from root
        and updates the answer which is passed by reference
*/

function dfs(node , parent, curSum, ref(ans))

        curSum += a[node]
        ans = max(ans, curSum)

        for v in adjacency[node] such that v != parent
```

```
                //   recursively call dfs for child
                dfs(v, node, curSum, ans)

        return

/*
        a function that takes in graph G, Array A and
        returns the maximum sum path in the given graph
*/


function subtree(G, A)

        ans = 0
        for i from 1 to n
                /*
                        with node, i as root call the dfs function to
                        compute the answer for this node as root
                */
                dfs(i, i, 0, ans)

        return ans
```

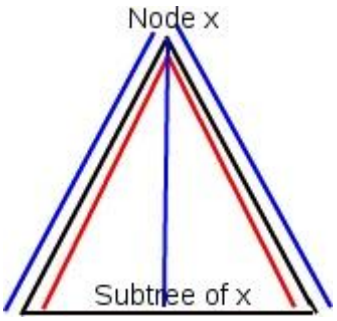
Time Complexity: The time complexity of the above algorithm is **$O(n^2)$**. Since we are making n dfs calls and each dfs call will visit all the n nodes, so we get the quadratic complexity.

Space Complexity: The space complexity of the above way is **O(n)** since we are making dfs calls and the maximum possible recursion depth can become O(n) in the worst case.

Dynamic Programming approach

First, let's root tree at node 1. Now, we need to observe that there would exist a node ***x*** such that:

- The largest sum path starts from node *x* and goes into its subtree(denoted by blue lines in the image). Let's define by *f*(*x*) this path length.
- The largest sum path starts in the subtree of *x*, passes through *x* and ends in the subtree of *x*(denoted by the red line in the image). Let's define by *g*(*x*) this path length.



If for all nodes *x*, we take a maximum of *f*(*x*), *g*(*x*), then we can get the desired path. But first, we need to see how we can calculate the maximum path length in both cases.

Now, let's say a node V which has the node value as a[V], has *n* children *v*1, *v*2, ..., *vn*. We have defined *f*(*i*) as the value of the best path that starts at node *i* and ends in the subtree of *i* (Case 1). We can recursively define *f*(*V*) as :

$$f(V) = a[V] + max(f(v1), f(v2),... f(vn))$$

because we are looking at the maximum path length possible from children of V and we take the maximum one. So, the optimal substructure is being followed here. Now, note that this is quite similar to DP except that now we are defining functions for nodes and defining recursion based on values of children. This is what DP on trees is.

Now, for case 2, a path can originate from subtree of node *vi*, and pass-through node V and end in the subtree of *vj*, where $i \neq j$. Since we want this path length to be maximum, we'll choose two children *vi* and *vj* such that *f*(*vi*) and *f*(*vj*) is maximum. We say that

$$g(V) = a[V] + \text{sum of 2 maximum values of } \{f(v1), f(v2),... f(vn)\}$$

For implementing this, we note that for calculating *f*(*V*), we need *f* to be calculated for all children of V. So, we do a DFS and we calculate these values on the go.

The code for the above approach is as follows:

```
/*
```

```
        recursive dfs function that takes a node and f and g arrays
        and updates the answer each time where the answer is passed by reference
*/

function dfs(node , f, g, ref(answer), G, a)

        f[node] = a[node]

        //   array to store the f values of children
        fValues = []

        for v in children[node]

                //   recursively call dfs for child
                dfs(v, f, g, answer, G, a)

                fValues.add(f[v])



        //   get the f[node]
        if !fValues.empty
                f[node] += fValues.max_element

        //   update the g[node] using 2 largest values
        if fValues.size >=2
                g[node] = a[node] + fValues.max_element  + fValues.second_max_element

        //   update the global answer
        answer = max(answer, g[node], f[node])

        return

/*
```

```
        function that takes in graph G, Array A and
        returns the maximum sum path in the given graph
*/



function subtree(G, A)



        answer = 0

        //   arrays to store the f and g function as defined above
        f = array[n]
        g = array[n]

        //   dfs call to calculate f and g
        dfs(1, f, g, answer, G, A)

        //   return  the max answer globally
        return answer
```

Time Complexity: The time complexity of the above algorithm is **O(n)**. Since we are making dfs calls in such a way that each node is visited exactly once and inside each visit a constant amount of work is done. Here we assume that we can get the maximum and the second maximum element of an array of size n can be found in O(n) time complexity.

Space Complexity: The space complexity of the above way is **O(n**) since we are making a dfs call and the maximum possible recursion depth can become O(n) in the worst case. Also, we are maintaining a dp array of size n so the total space will be O(n) + O(n) = O(n).