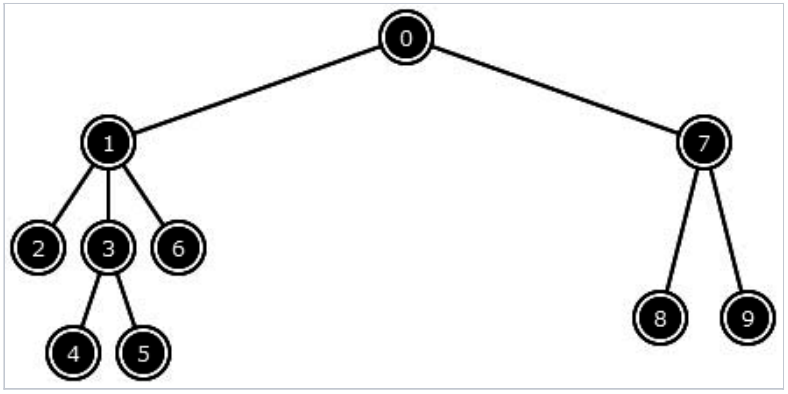
Lowest Common Ancestor

Authored by HackerRank

In Graph Theory and Computer Science, the Lowest Common Ancestor(LCA) of two nodes u and v in a tree or a directed acyclic graph(DAG) is the lowest node that has both u and v as descendants,where we define each node to be a descendant of itself . Thus , The LCA of u and v is the ancestor of u and v such that it is located farthest from the root .Computation of LCA has many applications such as finding distance between pair of nodes in a tree , Usage with Suffix trees for string processing and Computational Biology.

In a tree data structure where each node points to its parent, the lowest common ancestor can be easily determined by finding the first intersection of the paths from u and v to the root . This is the main idea behind the LCA-finding techniques that would be discussed in this article . The techniques discussed would be : *Naive algorithm for finding the LCA* , *Finding LCA using RMQ* , *Finding LCA with the use of Sparse Table/Dynamic Programming* , *Finding LCA using Square Root Decomposition of Tree*.

Go to Top



The above tree is an example of a rooted tree with 10 vertices and rooted at node number . You may verify manually that , , , , and

Throughout this article we will assume to denote the total number of nodes in the tree.

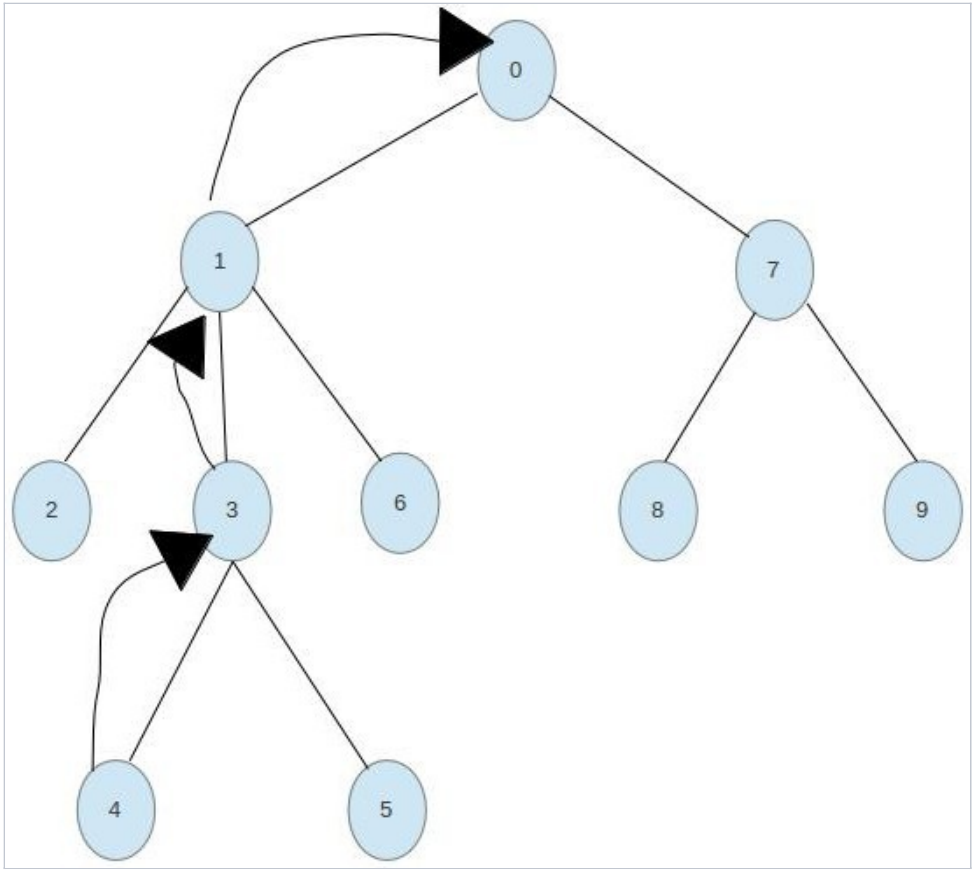
1)Naive Algorithm for finding the LCA:

Time Complexity Per Query:

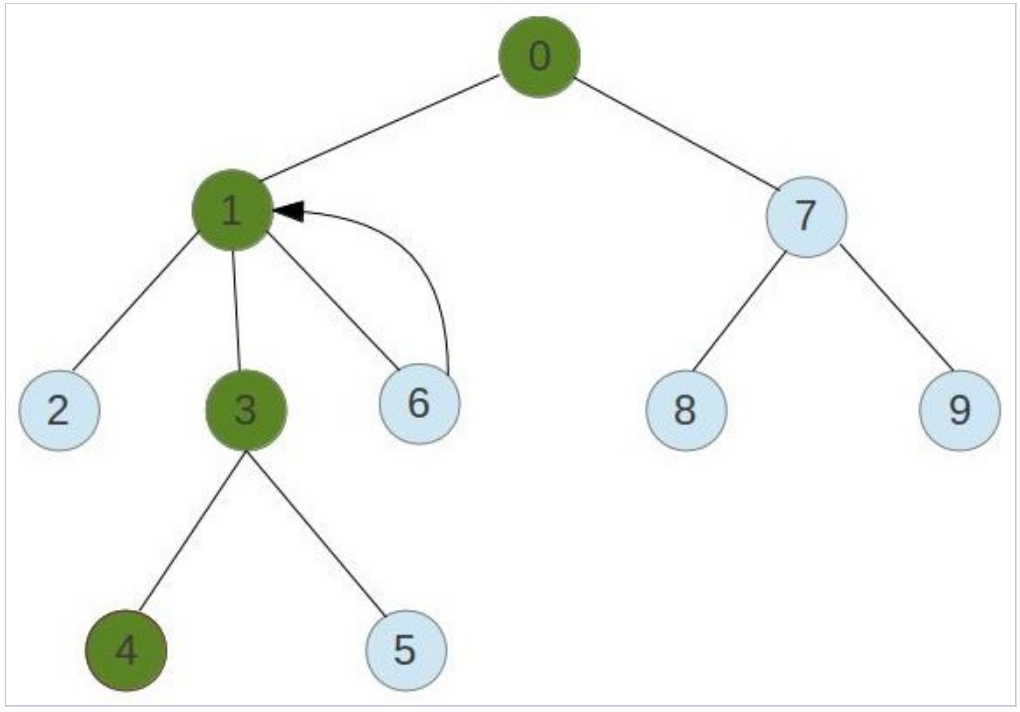
A naive solution consists of two steps: From the first vertex , we go all the way up to the of the tree and record all the vertices traversed along the way . From the second vertex , we also go all the way up to the root of the tree , but this time we stop if we encounter a common vertex for the first time . This common vertex is the *LCA*. For example, if we want to compute of the tree in the figure of the tree given above in this article using this naive solution, we will first traverse the path

and mark these 4 vertices as visited.

_Step 1: _



Step 2:



Nodes visited in Step 1 are marked in green

Then we start our traversal from the second vertex of the the query and traverse along the path

. We stop our traversal at node because this is in the path from the first vertex to the root . Hence , the paths of to and to intersect for the first time at node .As the traversal of the nodes is such that the distance of the nodes from the keeps decreasing as the traversal continues , We can say node is the node farthest from the that is an ancestor of nodes and . Hence ,

This algorithm is easier to code than other algorithms however the first and second steps of this naive algorithm take time in the worst case where is the maximum distance of the from a . In case the tree is very unbalanced such that every node of the tree has exactly child , The structure of the tree becomes *linear* and hence where is the total number of *nodes* in the tree making this algorithm . Hence , this algorithm is very slow if the question asks us to find *LCA* of two input nodes many times . In such a case the time complexity of this algorithm is where Q is the number of queries which usually gets a *TLE* verdict on Online Judges .

Here is a code snippet:

```
int parent[MAX_NODES];                  /*Keeps track of the parent of every vertex in the tree
*/
bool visited[MAX_NODES];
vector<vector<int> > tree[MAX_NODES];   /*The tree is stored as an undirected graph using an ad
jacency list*/



/*GetParents() function traverses the tree and computes the parent array such that
The pre-order traversal begins by calling GetParents(root_node,-1) */
void GetParents(int node , int par){
    for(int i = 0 ; i < tree[node].size() ; ++i){
        /*As this is a pre-order traversal of the tree the parent of the current    node has al
ready been processed*/
        Thus it should not be processed again*/
        if(tree[node][i] != par){
            parent[tree[node][i]] = node ;
            GetParents(tree[node][i] , node) ;
        }
    }
}


/*Computes the LCA of nodes u and v . */
int LCA(int u , int v){
    /*traverse from node u uptil root node and mark the vertices encountered along the path . *
/
    int lca ;
    while(1){
        visited[u] = true ;
        if(u == root_node){
            break ;
        }
        u = parent[u] ;
    }

    /*Now traverse from node v and keep going up untill we dont hit a node that is in the path
of node u to root node*/
    while(1){
        if(visited[v]){ /*Intersection of paths found at this node.*/
            lca = v;
            break ;
        }
        v = parent[v] ;
    }
    return lca ;
```

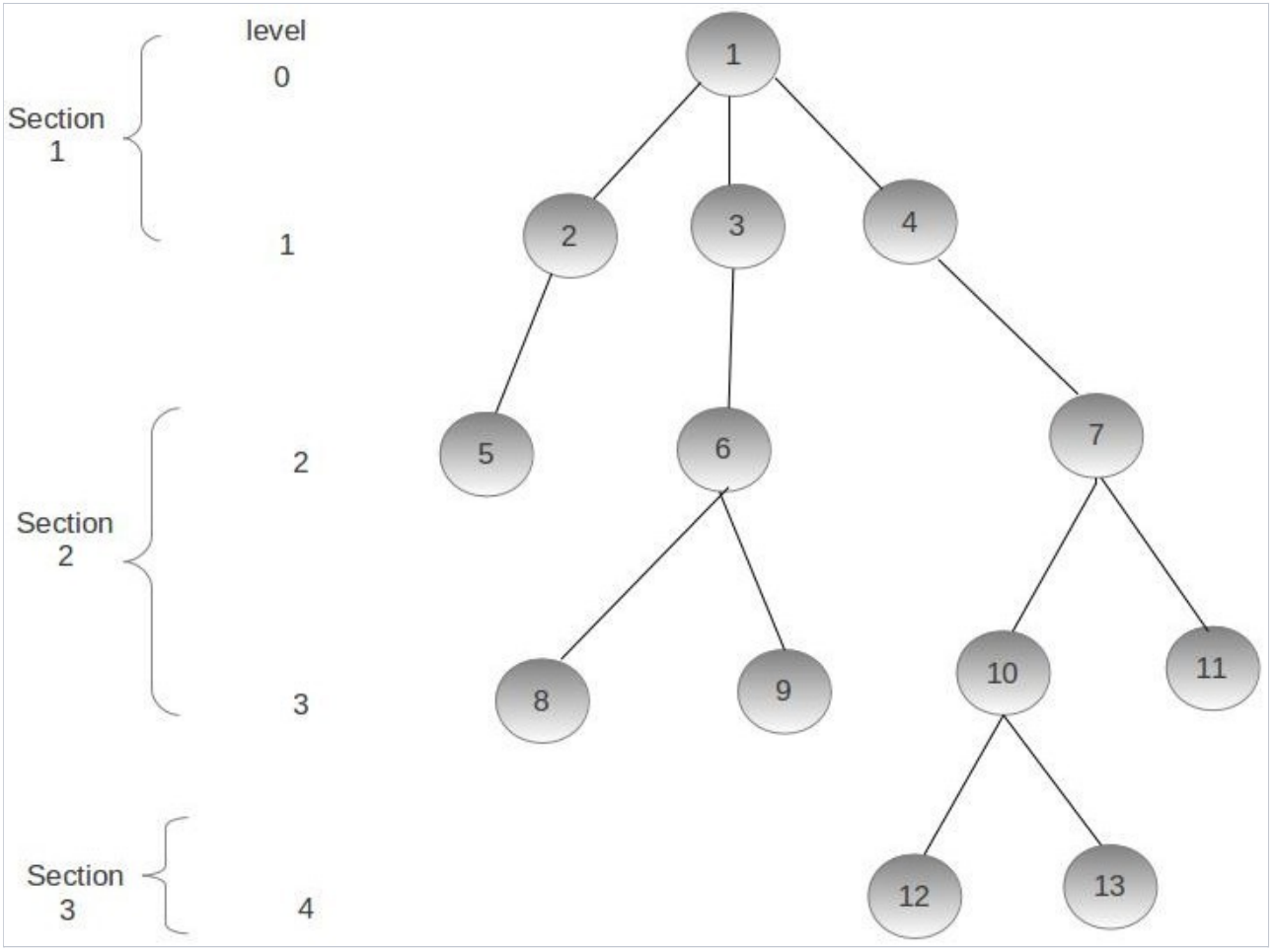
```
}
```

2)LCA Using Square-Root Decomposition:

Time Complexity for Preprocessing:

Time Complexity for answering each query:

In this method we would be dividing our tree into equal-sized parts to make the process of answering queries faster.The idea is to split the tree into parts, where is the height of the tree. Thus t he section would contain all the nodes having level from level to level . Here is how the tree would look like after the decomposition:



So each node is alloted a section such that:

Section of node =

Now, in the preprocessing step for each node , we store a node such that it is an *ancestor* of the *current node* and it is located at the *bottom-most level* of the *previous/upper section* of the *current node* . In our discussion , we would be storing this information for every node in an integer array . For the tree in the figure above here is how the table would look like:

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]	P[9]	P[10]	P[11]	P[12]	P[13]
1	1	1	1	2	3	4	6	6	7	7	10	10

For the nodes that are at the *topmost* level in their respective section .

For the sake of simplicity:

```
int parent[MAX_NODES] , level[MAX_NODES] , P[MAX_NODES];
vector<vector<int> > tree[MAX_NODES];

void traverse(int node , int head ,int prev_section){
    /*head stores the node that is in the same level as the current node
    And is its ancestor*/
    int current_section =  sqrt(level[node]) + 1;
    if(current_section == 1){
        P[node] = 1 ;
    }
    else{
        if(current_section == prev_section + 1){
            P[node] = parent[node] ;
            head = node ;
        }
        else{
            P[node] = parent[head] ;
        }
    }
    for(int i = 0 ; i < tree[node].size() ; ++i){
        if(tree[node][i] != parent[node]){
            traverse(tree[node][i] ,head ,current_section) ;
        }
    }
}
```

After this precomputation coding of the *LCA* function becomes pretty easy. Suppose we are finding the *LCA* for and . At Each step two cases may occur:

 . In this case we pick the node at a greater depth and update its value to and then carry the process further. Each such movement takes time because of the precomputation that we made earlier.

 . In this case it is guaranteed that the *LCA* of the two nodes is either or some node at a lower level/greater depth than . Thus in such a case we can keep picking the node at the greater depth at each step, and move it one unit upwards i.e. . As the distance of from either nodes is less than or equal to , this process takes time in the worst case.

```
void LCA(int u , int v){
    while(P[u] != P[v]){
        if(level[u] > level[v]){
            u = P[u];
        }
        else{
            v = P[v];
        }
    }
    while(u != v){
        if(level[u] > level[v]){
            u = parent[u] ;
        }
        else{
            v = parent[v] ;
        }
    }
    return u ; //Either u or v store the lca.
}
```

Worst case happens when the two nodes are two leaves at the maximum depth and their *LCA* is the *Root* node. Hence in this case a total of steps are taken.Thus the complexity for answering each query is .

3) LCA using Sparse Table:

In this method the precomputation step would consist of calculating the ancestor for all valid for each . To do this we

would be using a *sparse* table. This technique to find the *kth* ancestor for a node in a tree is also known as *binary raise* technique. We would be precomputing a *2D* table such that stores the ancestor for node . Here is a well-commented code snippet for the pre-computation step.

```
//P[i][j] stores the 2^j th ancestor of node i
int P[MAX_NODES][MAX_LOG] , parent[MAX_NODES];

void preprocess(){
    //Every element in P[][] is -1 initially;
    for(int i = 1 ; i <= N ; ++i){
        for(int j = 0 ; (1<<j) < N ; ++i){
            P[i][j] = -1;
        }
    }

    //The first ancestor(2^0 th ancestor)
    //for every node is parent[node]
    for(int i = 1 ; i <= N ; ++i){
        P[i][0] = parent[i] ;
    }

    for(int j = 1; (1<<j) < N ; ++j){
        for(int i = 1 ; i <= N ; ++i){
            //If a node does not have a (2^(j-1))th ancestor
            //Then it does not have a (2^j)th ancestor
            //and hence P[i][j] should remain -1
            //Else the (2^j)th ancestor of node i
            //is the (2^(j-1))th ancestor of ((2^(j-1))th ancestor of node i)
            if(P[i][j-1] != -1){
                P[i][j] = P[P[i][j-1]][j-1] ;
            }
        }
    }
}
```

The above code clearly takes time and space .

Now we discuss the *LCA* function in this technique. Let us assume that we are given two nodes and such that . Hence their *LCA* would be at the same distance, let's say above them. Hence . We can use the property that every number can be expressed as a *sum* of distinct *powers* of . Hence we may use our table to keep raising our nodes by distinct powers of *two* to find their *LCA*. In this explanation we assumed and to be at the same *level* but in other cases we may easily raise the node at a *greater depth/lower level* to be at the same *level* as the other node and then use the technique described above.

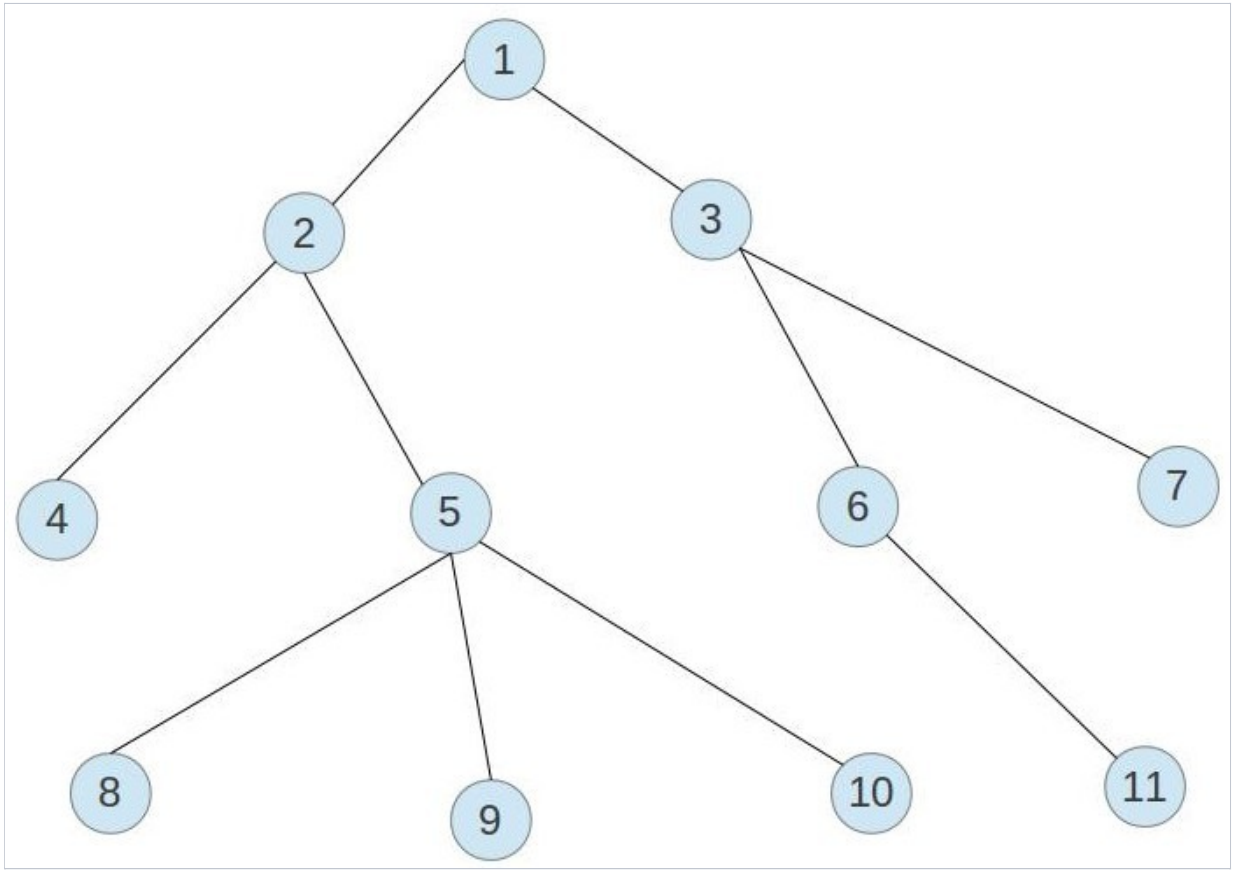
Here is a code snippet:

```
int LCA(int u , int v){
    if(level[u] < level[v]){
        swap(u,v) ;
    }
    //u is the node at a greater depth/lower level
    //So we have to raise u to be at the same
    //level as v.
    int dist = level[u] - level[v] ;
    while(dist > 0){
        int raise_by = log2(dist);
        u = P[u][raise_by];
        dist -= (1<<raise_by) ;
    }

    if(u == v){
        return u ;
    }

    //Now we keep raising the two nodes by equal amount
    //Untill each node has been raised uptill its (k-1)th ancestor
    //Such that the (k)th ancestor is the lca.
    //So to get the lca we just return the parent of (k-1)th ancestor
    //of each node

    for(int j = MAXLOG ; j >= 0 ; --j){
        //Checking P[u][j] != P[v][j] because if P[u][j] == P[v][j]
        //P[u][j] would be the Lth ancestor such that (L >= k)
        //where kth ancestor is the LCA
        //But we want the (k-1)th ancestor.
        if((P[u][j] != -1) and (P[u][j] != P[v][j])){
            u = P[u][j] ;
            v = P[v][j] ;
        }
    }
    return parent[u] ; //or parent[v]
}
```

In the worst case this function takes at most operations where . In the worst case, . Hence this functon is in the worst case.

4)Reduction from LCA to RMQ:

Before proceeding through this section, you are adviced to go through the techniques discussed in the *RMQ* and *Euler Tour* articles. In this section we would use the properties of *Euler tour* of a tree to reduce the problem of finding the *LCA* of two nodes to a problem of *RMQ*. Let us consider the following tree and build the *euler tour* array for the tree:



Here is the *euler tour* array for the tree:

1	2	4	2	5	8	5	9	5	10	5	2	1	3	6	11	6	3	7	3	1

Notice that is the node located at the uppermost level/least depth_ in the path between and .

The size of an *Euler Tour* array for a tree with nodes is . Let us denote the *euler tour* array by . Let be the index of the *first* occurance of in the *euler tour* array. Suppose we have a query to find . Let us assume is such that (otherwise we can swap and). Notice that to consists of all the *nodes* encountered in the path between and . Using the fact that is the *node* with the *least depth* in this path, We can do a *Range Minimum Query* from to such that returns the *index*

of an element in the *euler tour array* such that it lies between and and its *depth* is *minimum* . Suppose

Then

.

The *Time* and *Memory* complexity for this method depends on the method used to perform .

Using a *Sparse Table* :

Using a *Segment Tree*:

```
int LCA(int u , int v){
    //E[] is the euler tour array
    //F[i] stores the index of the
    //first occurance of i in the euler tour array.
    return E[RMQ(F[u] , F[v])];
}
```