Sqrt (or Square Root) Decomposition | Set 2 (LCA of Tree in O(sqrt(height)) time)

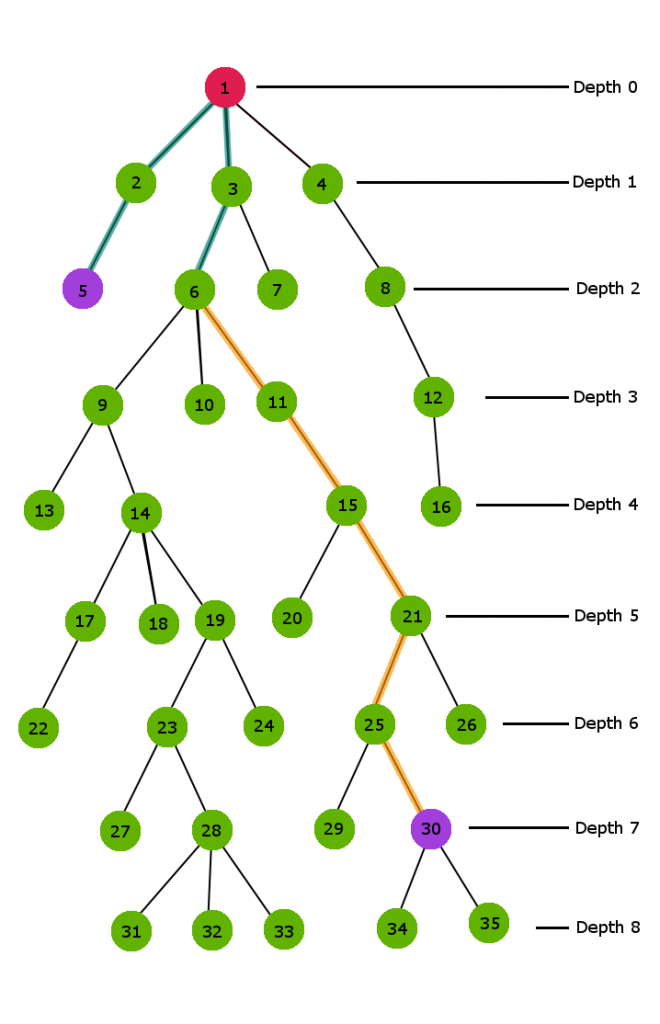
Last Updated: 24-01-2020

Prerequisite : [Introduction](https://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/) and [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)

The task is to find LCA of two given nodes in a tree (not necessarily a Binary Tree). In previous posts, we have seen how to calculate [LCA using Sparse Matrix DP approach](https://www.geeksforgeeks.org/lca-for-general-or-n-ary-trees-sparse-matrix-dp-approach-onlogn-ologn/). In this post, we will see an optimization done on Naive method by sqrt decomposition technique that works well over the Naive Approach.

**Naive Approach**  
To calculate the LCA of two nodes first of all we will bring both the nodes to same height by making the node with greater depth jump one parent up the tree till both the nodes are at same height. Once, both the nodes are at same height we can then start jumping one parent up for both the nodes simultaneously till both the nodes become equal and that node will be the LCA of the two originally given nodes.

Consider the below n-ary Tree with depth 9 and lets examine how naive approach works for this sample tree.



**Here in the above Tree we need to calculate the LCA of node 6 and node 30**  
Clearly node 30 has greater depth than node 6. So first of all we start jumping one parent above for node 30 till we reach the depth value of node 6 i.e at depth 2.

The **orange colored path** in the above figure demonstrates the jumping sequence to reach the depth 2. In this procedure we just simply jump one parent above the current node.

Now both nodes are at same depth 2.Therefore, now both the nodes will jump one parent up till both the nodes become equal. This end node at which both the nodes become equal for the first time is our LCA.

The **blue color path** in the above figure shows the jumping route for both the nodes

**Code for the above implementation:-**

C++

|  |
| --- |
| // Naive C++ implementation to find LCA in a tree  #include<bits/stdc++.h>  using namespace std;  #define MAXN 1001    int depth[MAXN];           // stores depth for each node  int parent[MAXN];          // stores first parent for each node    vector < int > adj[MAXN];    void addEdge(int u,int v)  {      adj[u].push\_back(v);      adj[v].push\_back(u);  }    void dfs(int cur, int prev)  {      // marking parent for each node      parent[cur] = prev;        // marking depth for each node      depth[cur] = depth[prev] + 1;        // propogating marking down the tree      for (int i=0; i<adj[cur].size(); i++)          if (adj[cur][i] != prev)              dfs(adj[cur][i],cur);  }    void preprocess()  {      // a dummy node      depth[0] = -1;        // precalclating 1)depth.  2)parent.      // for each node      dfs(1,0);  }    // Time Complexity : O(Height of tree)  // recursively jumps one node above  // till both the nodes become equal  int LCANaive(int u,int v)  {      if (u == v)  return u;      if (depth[u] > depth[v])          swap(u, v);      v = parent[v];      return LCANaive(u,v);  }    // Driver function to call the above functions  int main(int argc, char const \*argv[])  {      // adding edges to the tree      addEdge(1,2);      addEdge(1,3);      addEdge(1,4);      addEdge(2,5);      addEdge(2,6);      addEdge(3,7);      addEdge(4,8);      addEdge(4,9);      addEdge(9,10);      addEdge(9,11);      addEdge(7,12);      addEdge(7,13);        preprocess();        cout << "LCA(11,8) : " << LCANaive(11,8) << endl;      cout << "LCA(3,13) : " << LCANaive(3,13) << endl;        return 0;  } |

**Output:**

LCA(11,8) : 4

LCA(3,13) : 3

**Time Complexity** : We pre-calculate the depth for each node using one **DFS traversal in O(n)**. Now in worst case, the two nodes will be two bottom most node on the tree in different child branches of the root node. Therefore, in this case the root will be the LCA of both the nodes. Hence, both the nodes will have to jump exactly h height above, where h is the height of the tree. So, to answer each **LCA query Time Complexity will be O(h)**.

**The Sqrt Decomposition Trick :**  
We categorize nodes of the tree into different groups according to their depth. Assuming the depth of the tree h is a perfect square. So once again like the [general sqrt decomposition approach](https://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/) we will be having sqrt(h) blocks or groups. Nodes from depth*0 to depth sqrt(h) – 1*lie in first group; then nodes having depth*sqrt(H) to 2\*sqrt(h)-1*lie in second group and so on till last node.

We keep track of the corresponding group number for every node and also depth of every node. This can be done by one single dfs on the tree (see the code for better understanding).

**Sqrt trick**:- In naive approach we were jumping one parent up the tree till both nodes aren’t on the same depth. But here we perform group wise jump. To perform this group wise jump, we need two parameter associated with each node : 1) parent and 2) jump parent  
Here **parent** for each node is defined as the first node above the current node that is directly connected to it, where as **jump\_parent** for each node is the node that is the first ancestor of the current node in the group just above the current node.

So, now we need to maintain 3 parameters for each node :  
**1) depth**  
**2) parent**  
**3) jump\_parent**

All these three parameters can be maintained in one dfs(refer to the code for better understanding)

**Pseudo code for optimization process**

LCAsqrt(u, v){

*// assuming v is at greater depth*

while (jump\_parent[u]!=jump\_parent[v]){

v = jump\_parent[v];

}

*// now both nodes are in same group*

*// and have same jump\_parent*

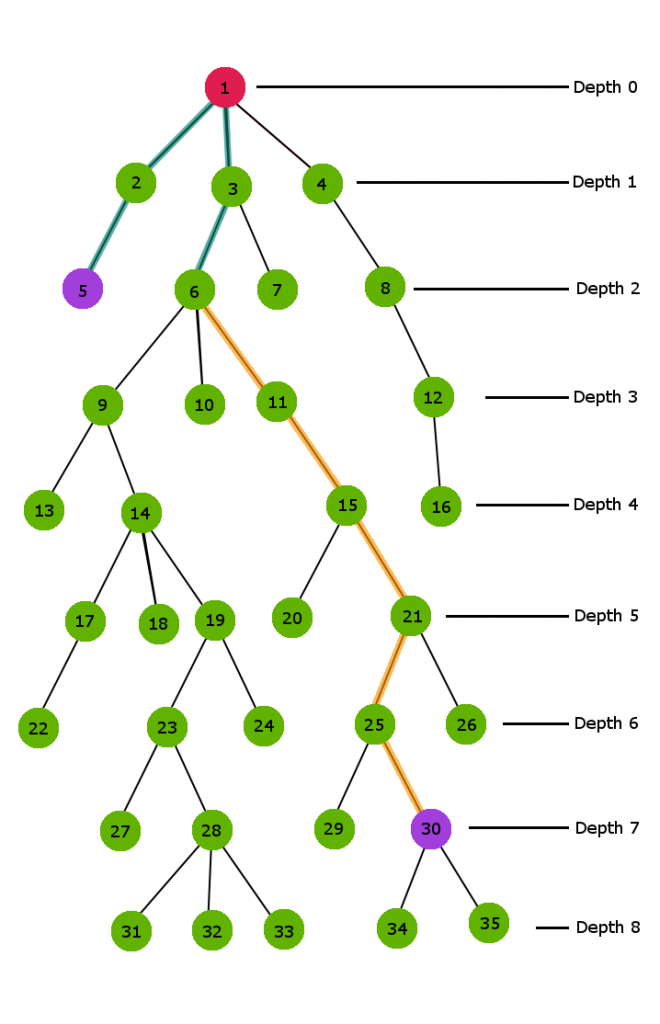
return LCAnaive(u,v);

}

The key concept here is that first we bring both the nodes in same group and having same jump\_parent by climbing decomposed blocks above the tree one by one and then when both the nodes are in same group and have same jump\_parent we use our naive approach to find LCA of the nodes.

This optimized group jumping technique reduces the iterating space by a factor of **sqrt(h)** and hence reduces the Time Complexity(refer below for better time complexity analysis)

Lets decompose the above tree in sqrt(h) groups (h = 9) and calculate LCA for node 6 and 30.



In the above decomposed tree

**Jump\_parent[6] = 0 parent[6] = 3**

**Jump\_parent[5] = 0 parent[5] = 2**

**Jump\_parent[1] = 0 parent[1] = 0**

**Jump\_parent[11] = 6 parent[11] = 6**

**Jump\_parent[15] = 6 parent[15] = 11**

**Jump\_parent[21] = 6 parent[21] = 15**

**Jump\_parent[25] = 21 parent[25] = 21**

**Jump\_parent[26] = 21 parent[26] = 21**

**Jump\_parent[30] = 21 parent[30] = 25**

Now at this stage Jump\_parent for node 30 is 21 and Jump\_parent for node 5 is 0, So we will climp to jump\_parent[30] i.e to node 21

Now once again Jump\_parent of node 21 is not equal to Jump\_parent of node 5, So once again we will climb to jump\_parent[21] i.e node 6

At this stage jump\_parent[6] == jump\_parent[5], So now we will use our naive climbing approach and climb one parent above for both the nodes till it reach node 1 and that will be the required LCA .

**Blue path** in the above figure describes jumping path sequence for node 6 and node 5.

|  |
| --- |
| // C++ program to find LCA using Sqrt decomposition  #include "iostream"  #include "vector"  #include "math.h"  using namespace std;  #define MAXN 1001    int block\_sz;          // block size = sqrt(height)  int depth[MAXN];       // stores depth for each node  int parent[MAXN];      // stores first parent for                         // each node  int jump\_parent[MAXN]; // stores first ancestor in                         // previous block    vector < int > adj[MAXN];    void addEdge(int u,int v)  {      adj[u].push\_back(v);      adj[v].push\_back(u);  }    int LCANaive(int u,int v)  {      if (u == v)  return u;      if (depth[u] > depth[v])          swap(u,v);      v = parent[v];      return LCANaive(u,v);  }    // precalculating the required parameters  // associated with every node  void dfs(int cur, int prev)  {      // marking depth of cur node      depth[cur] = depth[prev] + 1;        // marking parent of cur node      parent[cur] = prev;        // making jump\_parent of cur node      if (depth[cur] % block\_sz == 0)            /\* if it is first node of the block             then its jump\_parent is its cur parent \*/          jump\_parent[cur] = parent[cur];        else            /\* if it is not the first node of this block             then its jump\_parent is jump\_parent of             its parent \*/          jump\_parent[cur] = jump\_parent[prev];          // propogating the marking down the subtree      for (int i = 0; i<adj[cur].size(); ++i)          if (adj[cur][i] != prev)              dfs(adj[cur][i], cur);  }      // using sqrt decomposition trick  int LCASQRT(int u, int v)  {      while (jump\_parent[u] != jump\_parent[v])      {          if (depth[u] > depth[v])                // maintaining depth[v] > depth[u]              swap(u,v);            // climb to its jump parent          v = jump\_parent[v];      }        // u and v have same jump\_parent      return LCANaive(u,v);  }    void preprocess(int height)  {      block\_sz = sqrt(height);      depth[0] = -1;        // precalclating 1)depth.  2)parent.  3)jump\_parent      // for each node      dfs(1, 0);  }    // Driver function to call the above functions  int main(int argc, char const \*argv[])  {      // adding edges to the tree      addEdge(1,2);      addEdge(1,3);      addEdge(1,4);      addEdge(2,5);      addEdge(2,6);      addEdge(3,7);      addEdge(4,8);      addEdge(4,9);      addEdge(9,10);      addEdge(9,11);      addEdge(7,12);      addEdge(7,13);        // here we are directly taking height = 4      // according to the given tree but we can      // pre-calculate height = max depth      // in one more dfs      int height = 4;      preprocess(height);        cout << "LCA(11,8) : " << LCASQRT(11,8) << endl;      cout << "LCA(3,13) : " << LCASQRT(3,13) << endl;        return 0;  } |

Output:

LCA(11,8) : 4

LCA(3,13) : 3

**Note :** The above code works even if height is not perfect square.

Now Lets see how the Time Complexity is changed by this simple grouping technique :

**Time Complexity Analysis:**  
We have divided the tree into sqrt(h) groups according to their depth and each group contain nodes having max difference in their depth equal to sqrt(h). Now once again take an example of worst case, let’s say the first node ‘u’ is in first group and the node ‘v’ is in sqrt(h)th group(last group). So, first we will make group jumps(single group jumps) till we reach group 1 from last group; This will take exactly sqrt(h) – 1 iterations or jumps. So, till this step the Time Complexity is **O(sqrt(h))**.

Now once we are in same group, we call the LCAnaive function. The Time complexity for LCA\_Naive is O(sqrt(h’)), where h’ is the height of the tree. Now, in our case value of h’ will be sqrt(h), because each group has a subtree of at max sqrt(h) height. So the complexity for this step is also O(sqrt(h)).  
Hence, the total Time Complexity will be **O(sqrt(h) + sqrt(h)) ~ O(sqrt(h))**.