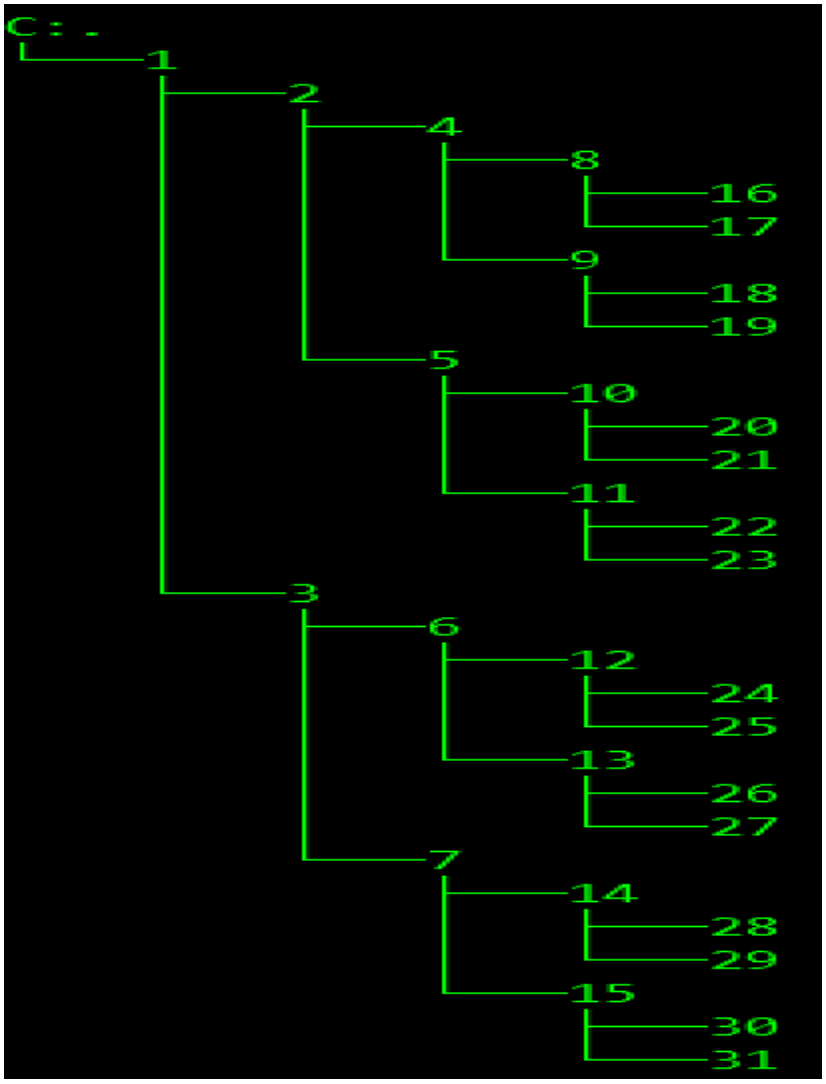
Problem Description:

Let's assume we are working on software which tells the user that how many minimum click required to reach the current directory to destination directory in drive. So initially we don't know about the size of directory so we assume that size of directory can be infinite (i.e. Depth of tree). And name of directories is indexed with integer (for easy to implementation) which has root directory is 1. And 1 has two sub-directories 2 and 3 and so on. Now our task is to write a program which takes two input X (as a current directory) and Y (as a destination directory) and returns the minimum number of click for which user can reach from directory X to Y. For more clarity refer image-



Tree view of directory 1

Solution of given problem:

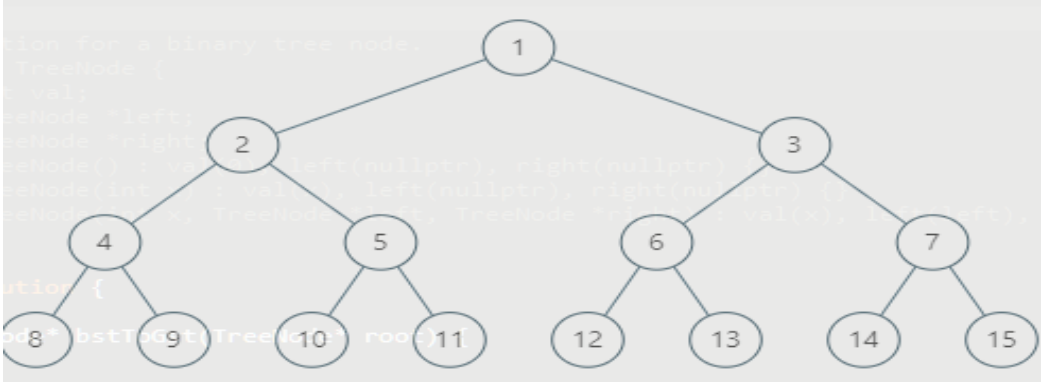
Brute force approach:

One brute force approach to solve this problem is that, just create a tree and find the path from root node (1) to X and path from root node to Y and find the length of common ancestor. And finally minimum distance of X to Y will be

distance=pathX.size() + pathY.size()-2*lenCommonAncestor;

For example:

Tree



X=10 , y=11

So path of X={1,2,5,10} and path of Y={1,2,5,11}

So common ancestor {1,2,5} is included two times once in path of X and other in Path Y.

That's why

distance=pathX.size() + pathY.size()-2*lenCommonAncestor;

1) Steps to implement:

a) Tree structure:

We are going to use array as a tree for easy to build the tree.

And root node will be index 1 and we will keep 0 index as dummy node,

0-th node as dummy because if want to go left then index of root node of left subtree would be 2*i and for right. 2*i+1 so if parent would be 0 then 2*i will

always 0. Or other way we can start with 0 then for index of root of left subtree would be 2*i+1 and for right subtree 2*i+2.

b) Finding the path:

For finding the path we are going to use depth first search and whenever we found the tree value as X then we stored it and return as a true for which while backtracking I would able to store exact path from root to node.

i) call dfs on root node==1
ii) whenever tree[index]==x store and return true
iii) call on left side with index=index*2 and if left subtree has path then store tree[index] else call on right subtree with index =index*2+1.

c) Finding distance:

distance=pathX.size() + pathY.size()-2*lenCommonAncestor;

2) Complexity Analysis:

a) Time complexity:

Time complexity of this algorithm would be O(max(X,Y)) because we are spending time to do dfs for finding path. So for getting path we need to check all the node.

b) Space complexity:

Space complexity of this algorithm would be O(max(X,Y)) . Since we are building the tree and for inserting node X and Y we have to insert at least max(X,y) element in the tree.

Optimize level I:

In brute force approach we are constructing the tree which took O(max(X ,Y)) space complexity and doing dfs which took O(max(X,Y)) time complexity. But if we analyze, we don't need to construct the complete tree because *Tree could be of height of infinite length* so we are not allowed to construct the tree. Instead of constructing the tree and getting the path we could have store direct ancestor.

If current node is X then it's child would be 2*x and 2*x+1 it sure. Then look the reverse of this statement: -

If current node is X then it's parent would be **floor(x/2)** as we did in heap to apply **Heapify** for finding the parent of current node.

Steps to implement:

1. **Get Ancestor**
   I. Make a recursive (or iterative) function
   II. If value is 0 return
   III. Else call f(value/2)

IV. store parent while backtracking because we want to path as increasing and later we apply on binary search Otherwise have to reverse the path or have to change binary search algorithm (because then we have path in decreasing order)

2. **Find Last Common Ancestor**
   I. so find first common ancestor, so start from end because in beginning probabilities of getting number of common ancestor is high.
   II. for example: ancestorX {1,2,5} and ancestorY {1,2,4} so getting first common ancestor we have to binary search on 1 and 2 and then when we call on 5 or 4 we get mismatched ancestor and get 2 as a last common ancestor but if we start from end then after calling at 2 we get first matched ancestor.
   III. Do binary search, if current ancestor of x is present in y then just return the distance. And distance would be number of non-common ancestor in ancestor of X(ancestorX.size()-1-i) + number of non-common ancestor in ancestor of Y(ancestorY.size()-1-x)).
   IV. distance=(ancestorX.size()-1-i)+(ancestorY.size()-1-x); where x is the index of current ancestor of X into Y.

Complexity Analysis:

I. **Time complexity:**
We are storing the ancestor which would take *O(log(max(x,y) * log(max(x,y)))* because each time we are going one level up so at most *height* would be log(max(X,y)), But finding the first common ancestor will take *log (max(X, Y) *log(max(X,Y))* because we are just doing binary search for all x in worst case.

II. **Space complexity:**
Space complexity would be O(log(max(X, Y))) because we are storing just ancestors only and each time node is divide by 2.

Optimize level II:

In previous Optimize Level I implementation we have stored all the ancestors of x and y so space complexity was O (log (max (X, Y))) but if we analyses that solution so we are just finding the last common ancestor i.e. *any value k such that it is ancestor of both and no such value exist Z which is greater than k and also an common ancestor* so we can do this like as we did in heap while **Heapify** we are going to parent by getting flor(x/2).

so we will keep dividing by 2 to max (X, Y) until it become equal (x==y).

Steps to implement:

I. So first make x always as great value i.e. if x<y then swap
II. Each time distance will increase by one
III. If x>y then x/=2
IV. If y>x then y/=2 and increment once again distance because we found one more node which is not a common ancestor
V. Repeat step II to IV until x and y not become same.
VI. First time when it would be same the distance would be minimum distance among original x and y. Return distance.

Complexity Analysis:

I. Time complexity:

Time complexity would be O (log (max (X, Y)) because at most we can reach log (max (X, Y)) for getting common ancestor and each time X and Y decrease by 2.

II. Space complexity:

Since we are not using any extra space as we did in optimize level I. So space complexity would be O (1).