Trees : Level 2
Lesson 4

Party in Town

Geek town has N Houses numbered from 1 to N. They are connected with each other via N-1 bidirectional roads and an adjacency list is used to represent the connections. Find the house from which the distance to the farthest house is the minimum to host the optimal party.

Example 1:

Input:
N = 4
Roads = {{1,2},{2,3},{2,4}}
adj = {{2},{1,3,4},{2},{2}}

Output: 1

Your Task:

You do not need to read input or print anything. Your task is to complete the function **partyHouse()** which takes N and adj as input parameters and returns the minimum possible distance to the farthest house from the house where the party is happening.

Expected Time Complexity: O(N2)

Expected Auxiliary Space: O(1)

Constraints:

1 ≤ N ≤ 1000

```
int BFS(int v, vector<vector<int>> &adj, int N)
{
   vector<bool> visited(N+1,false);
   queue<int> q;
   q.push(v);
   visited[v]=true;
```

```
    int k=0;
    while(!q.empty())
    {
        k++;
        int l=q.size();
        while(l--)
        {
            v=q.front();
            q.pop();
            for(int i=0; i<adj[v].size(); i++)
            {
                if(!visited[adj[v][i]])
                {
                    visited[adj[v][i]]=true;
                    q.push(adj[v][i]);
                }
            }
        }
    }
    return k-1;
}
class Solution{
public:
    int partyHouse(int N, vector<vector<int>> &adj){
        int minm=INT_MAX;
        for(int i=1; i<=N; i++)
            minm=min(minm,BFS(i,adj,N));
        return minm;
    }
};
```

Firing employees

Geek is the founder of Geek Constructions. He always maintains a black-list of potential employees which can be fired at any moment.

The company has N employees (including Geek), and each employee is assigned a distinct rank (1 <= rank <= N) at the time of joining. The company has a hierarchical management such that each employee always has one immediate senior.

Geek has a strange and unfair way of evaluating an employee's performance. He sums the employee's rank and the number of seniors the employee has. If it is a prime number, the employee is put up on the black-list.

Given an array arr[] in order of the rank of company employees. For rank i, arr[i] represents the rank of the immediate senior of the employee with the ith rank. If geek's rank is i, then arr[i] is always equal to 0 as there is noone senior to him. Find out the number of Black-Listed employees.

Note: The black-list can not contain Geek's name as he is the founder of the company and he is the one that makes the list.

Example 1:

```
Input:
N = 4
arr[] = {0, 1, 1, 2}
Output: 1

Explanation:
The hierarchy is as follows
      (Geek)
      Rank 1
       /   \
 Rank 2     Rank 3
     /
Rank 4
```

Performance = rank + number of seniors
Performance for rank 1 = not considered.
Performance for rank 2 = 2+1 = 3 (prime)
Performance for rank 3 = 3+1 = 4 (not prime)
Performance for rank 4 = 4+2 = 6 (not prime)
Therefore, only employee 2 is black-listed.

Example 2:

Input:
N = 3
arr[] = {2, 3, 0}

Output: 2

Explanation:
The hierarchy is as follows

```
      (Geek)
      Rank 3
       /
  Rank 2
     /
Rank 1
```

Rank 1 and 2 are both black-listed.

Your Task:

You don't need to read input or print anything. Your task is to complete the function **firingEmployees()** which takes the array arr[] and its size N as input parameters. It returns the number of black-listed employees.

Expected Time Complexity: O(N)

Expected Auxiliary Space: O(N)

Constraints:

1 <= N <= 105

1 <= i <= N

1 <= arr[i] <= 105

```
int dfs(int mp[], vector<int> &arr, int i)
{
    if(arr[i]==0)
        mp[i]=0;
    else if(mp[i]==-1)
    {
        mp[i]=dfs(mp,arr,arr[i]-1)+1;
    }
    return mp[i];
}
vector<bool> b;
void precompute()
{
    b.resize(100010,true);
    b[0]=false;
    b[1]=false;
    for(int i=2;i<100010;i++){
        for(int j=i+i;j<100010;j=j+i){
            b[j]=false;
        }
    }
}
int firingEmployees(vector<int> &arr, int n){
    // code here
    precompute();
    int mp[n];
```

```
        for(int i=0; i<n; i++)  mp[i]=-1;
        int count=0;
        for(int i=0 ; i<n; i++)
        {
            if(arr[i]==0)   mp[i]=0;
            if(mp[i]==-1 && arr[i]!=0)
                mp[i]=dfs(mp,arr,arr[i]-1)+1;
        }
        for(int i=0 ; i<n; i++)
        {
            if(arr[i]!=0 && b[mp[i]+i+1])
                count++;
        }
        return count;
    }
};
```

Count the Number of Full Binary Trees

Given an array A[] of **N** integers, where each integer is greater than 1.The task is to find the number of Full binary tree from the given integers, such that each non leaf node value is the product of its children value.

Note: Each integer can be used multiple times in a full binary tree.

Example 1:
Input:
N = 4
A[] = {2, 3, 4, 6}

Output:
7
Explanation:
There are 7 full binary tree with the given product property.
Four trees with single nodes
2 3 4 6
Three trees with three nodes

```
   4   ,
  / \
 2   2
  6    ,
 / \
2   3

  6
 / \
3   2
```

Example 2:
Input: N = 3
A[] = {2, 4, 5}

Output: 7
Explanation: There are 4 full binary tree with the given product property.
Three trees with single nodes 2 4 5
One tree with three nodes

```
  6
 / \
 3   2
```

Your Task:

You don't need to read input or print anything. Your task is to complete the function **numoffbt()** which takes the array **A[]**and its size **N** as inputs and returns the number of Full binary tree.

Expected Time Complexity: O(N. Log(N))

Expected Auxiliary Space: O(N)

Constraints:

1 ≤ N ≤ 104

1 ≤ A[i] ≤ 104

```
int numoffbt(int arr[], int n)
{
    // Your code goes here
    int minm=INT_MAX,maxm=INT_MIN,i;
    for(i=0; i<n; i++)
    {
        minm=min(minm,arr[i]);
        maxm=max(maxm,arr[i]);
    }
    int mark[maxm+1];
```

```
        int value[maxm+1];
        memset(mark, 0, sizeof(mark));
              memset(value, 0, sizeof(value));
        for(i=0; i<n; i++)  mark[arr[i]]=value[arr[i]]=1;
        int c=0;
        for(i=minm; i<=maxm; i++)
        {
            if(mark[i])
            {
                for(int j=i+i; j<=maxm && j<=i*i; j+=i)
                {
                    if(mark[j])
                    {
                        value[j]+=value[i]*value[j/i];
                        if(i*i!=j)
                            value[j]+=value[i]*value[j/i];
                    }
                }
                c+=value[i];
            }
        }
        return c;
    }
};
```

Image Multiplication

You are given a binary tree. Your task is pretty straightforward. You have to find the sum of the product of each node and its mirror image (The mirror of a node is a node which exists at the mirror position of the node in the opposite subtree at the root.). Don’t take into account a pair more than once. The root node is the mirror image of itself.

Example 1:

Input:

```
    4
  /   \
 5     6
```

Output:

```
46
```

Explanation:

```
Sum = (4*4) + (5*6) = 46
```

Example 2:

Input:

```
                1
              /      \
             3          2
             /  \        /  \
          7     6       5    4
        /   \    \     /  \    \
      11    10    15  9    8    12
```

Output:

```
332
```

Explanation:

```
Sum = (1*1) + (3*2) + (7*4) + (6*5) + (11*12) + (15*9) = 332
```

Your Task:

You need to **complete** the **function imgMultiply()** that takes **root** as **parameter** and **returns** the **required sum.**The answer may be very large, compute the answer modulo 109 + 7.

Expected Time Complexity: O(Number of nodes).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1 <= Number of nodes <= 105

1 <= Data of a node <= 105

```
int chk(Node* root1, Node* root2)
{
  if(!root1 || !root2)    return 0;
  return (((long long)root1->data*root2->data)%1000000007 +chk(root1->left,root2->right)%1000000007+chk(root1->right,root2->left)%1000000007)%1000000007;
}
class Solution
{
  public:
  long long imgMultiply(Node *root)
  {
     // code here
     return (((long long)root->data*root->data)%1000000007 + chk(root->left,root->right)%1000000007)%1000000007;
  }
};
```

Burning Tree

Given a binary tree and a leaf node called target. Find the minimum time required to burn the complete binary tree if the target is set on fire. It is known that in 1 second all nodes connected to a given node get burned. That is, its left child, right child and parent.

Example 1:

```
Input :
        1
      /   \
    2      3
  /  \      \
 4    5      6
     / \      \
    7   8      9
                \
                10
```

Target Node = 8
Output: 7
Explanation: If leaf with the value 8 is set on fire.
After 1 sec: 5 is set on fire.
After 2 sec: 2, 7 are set to fire.
After 3 sec: 4, 1 are set to fire.
After 4 sec: 3 is set to fire.
After 5 sec: 6 is set to fire.
After 6 sec: 9 is set to fire.
After 7 sec: 10 is set to fire.
It takes 7s to burn the complete tree.

Example 2:

```
Input :
          1
        /   \
      2       3
    /  \       \
   4    5       7
  /    /
 8    10
Target Node = 10
```

Output: 5

Your Task:

You dont need to read input or print anything. Complete the function **minTime()** which takes the root of the tree and target as input parameters and returns minimum time required to burn the complete binary tree if the target is set on fire at the 0th second.

Expected Time Complexity: O(N)

Expected Auxiliary Space: O(height of tree)

Constraints:

1 ≤ N ≤ 10^4

```cpp
void findNode(Node* root, int target, Node* &targetN, unordered_map<Node*,Node*> &pa, Node* parent)
{
  if(!root)
     return;
  if(root->data==target)  targetN=root;
  pa[root]=parent;
```

```
    findNode(root->left,target,targetN,pa,root);
    findNode(root->right,target,targetN,pa,root);
}

void maxlevel(Node* root,unordered_map<Node*,Node*>pa, int &tim)
{
    set<Node*>p;
    queue<pair<Node*,int>> q;
    q.push({root,0});
    p.insert(root);
    while(q.size()>0)
    {
        root=q.front().first;
        int level=q.front().second;
        q.pop();
        tim=max(tim,level);
        if(root->left && !p.count(root->left))
        {
            q.push({root->left,level+1});
            p.insert(root->left);
        }
        if(root->right && !p.count(root->right))
        {
            q.push({root->right,level+1});
            p.insert(root->right);
        }
        if(pa[root] && p.count(pa[root])==0)
        {
            q.push({pa[root],level+1});
            p.insert(pa[root]);
        }
    }
}

int minTime(Node* root, int target)
{
    // Your code goes here
    unordered_map <Node*,Node*> pa;
    Node* targetN=NULL;
    findNode(root,target,targetN,pa,NULL);
    int tim=0;
    maxlevel(targetN,pa,tim);
    return tim;
}
```

Distribute candies in a binary tree

Given a binary tree with N nodes, in which each node value represents the number of candies present at that node. In one move, one may choose two adjacent nodes and move one candy from one node to another (the move may be from parent to child, or from child to parent.)

The task is to find the number of moves required such that every node has **exactly one** candy.

Example 1:

```
Input :     3
           /  \
          0    0
```
Output : 2
Explanation:
From the root of the tree, we move one candy to its left child, and one candy to its right child.

Example 2:

```
Input :     0
           /  \
          3    0
```
Output : 3
Explanation :
From the left child of the root, we move two candies to the root [taking two moves]. Then, we move one candy from the root of the tree to the right child.
Your task :
You don't have to read input or print anything. Your task is to complete the function **distributeCandy()** which takes the root of the tree as input and returns the number of moves required such that every node has exactly one candy.

Expected Time Complexity: O(n)
Expected Auxiliary Space: O(h)

Constraints:

1<=n<=10^4

```
int chk(Node* root, int &ans)
{
    if(!root)   return 0;
    int l=chk(root->left,ans);
    int r=chk(root->right,ans);
    ans+=abs(l)+abs(r);
    return root->key+l+r-1;
}

class Solution
{
    public:
    int distributeCandy(Node* root)
    {
        //code here
        int ans=0;
        chk(root,ans);
        return ans;
    }
};
```