Ans 1(a): **No**, as DFS traversal doesn't visits any node twice so we can't tell us if it returns to city it started or not.
Ans 1(b): **No**,as BFS traversal doesn't visits any node twice so we can't tell us if it returns to city it started or not.
Ans 1(c): All vertices have even degree and are connected.
Ans 1(d): If the graph meets the conditions mentioned in part c, we can follow this procedure to get such a path:
Pseudo-Code:

```
V<- any vertex in graph to start from
Use the recursive function below.
Find_Path(V):
  1. find any edge coming out of V:
     remove this edge from the graph,
     and call Find_Path( ) from the second end of this edge;
  2. add vertex V to the answer.
```

The complexity of this algorithm is obviously **O(E)** as, in each recursive call we delete an edge and we would stop when all edges are exhausted.

Ans 2(a):

Approach & Logic :

- If a city has tower, can send signal to cities whose shortest distance from the tower is at most x. We can equivalently say that neighbors of a tower of power x are acting as tower of power x-1 (unless they are tower themselves).
- Make a power array and initialize with -1 denoting inactive state of all cities initially.
- Create an empty queue and enqueue source (activating it), power[s]= tower's power(x)
- Now, apply a modified BFS, as whenever dequeue a node, visit its child only if its not the node with zero power(power[v]=0) i.e. at x distance from the nearest tower.
- Else, enqueue neigbor if it is either inactive or has power < power of current node -1
- After this BFS completes check if destination is active or not and return True/False.

Pseudo-Code:

```
n <- no. of cities
int s,d; // source and destination
tower_cities(n) // tower_cities[i]=1 if it has a tower else 0
g<- map/graph of state // adjacency list

check_power(x){
        if(s==d) return true;
        vector<int> power(n,-1);

        CreateEmptyQueue(Q);
        Enqueue(s,Q);
        power[s]=x;
        While(Not IsEmptyQueue(Q)){
                v= Dequeue(Q);
                if(power[v]==0) continue;
                For each neighbor w of v: {
                        if(power[w]< power[v]-1){
                                power[w]= power[v]-1;
                                if(tower_cities[w]) power[w]=x;
                                Enqueue(w,Q);
                        }
                }
        }

        if(power[d]!=-1)return true;
        else return false;
}
```

Ans 2(b):

Approach & Logic: Apply binary search for min_power required and check whether the power of x is capable or not using *check_power()* function.

Pseudo-Code:

```
min_power(){
        int lo=0, hi=n;  // there could be n-1 distance at max.in graph, so hi=n
        int mid= (lo+hi)>>1;
        int ans=n;

        while(lo<=hi){
                if(check_power(mid)){
                        ans=mid;
                        hi=mid-1;
                }
                else lo=mid+1;
                mid= (lo+hi)>>1;
        }
        return ans;
}
```

Ans 3:

Approach:

- Use complete binary tree to store the effect of bombings where leaf nodes as rooms.
- Value of node represents the index of bombing through which it (room or set of rooms in its subtree) was colored.
- For each bombing:
  1.Store corresponding color c in the *color_order* array.
  2. Let u and v be the leaf nodes corresponding to l-th and r-th room.
  3. Keep repeating the following step as long as u < v :
    - If u is right child of its parent, update value of current node to index of the bombing and mark the node right next to the current node as u.
    - If v is left child of its parent, update value of current node to index of the bombing and mark node left next to the current node as v.
    - Move up by one step simultaneously from u and v
  4. If both u and v are now same node, update value of current node to index of bombing.
- For each room, move from the corresponding leaf node to root and take maximum of values at nodes in the path. This will be the index of bombing whose color will be on the room. **final_color** array stores the final color of rooms.

Pseudo-Code:

```
n <- no. of rooms,  m <-no. of bombings
int len,st,l,r,c;
If( n & (n-1) ) st= n; // if n is power of 2
Else st= 1<<(log2(n) + 1);
len= 2*st - 1;
vector<int> A(l,-1);
vector<int> color_order(m);

For (k=0 to m-1):{
        cin>> l >> r >> c;
        color_order[k]= c;

        i= l-1; j= r-1; //bring in 0-based indexing
        i = n-1 + i; //start node position in tree
        j = n-1 + j; //end node position in tree

        if(i==j) A[i]=k;
        else if(j>i){
                while( j>i ){
                        if(i%2==0){
                                A[i]=k;
                                i=i+1;
                        }
                        if(j%2==1){
```

```
                                A[j]=k;
                                j=j-1;
                        }

                        i= (i-1)/2;
                        j= (j-1)/2;
                }
                if(i==j) A[i]=k;
        }
}

vector<int> final_color(n); //stores the final colors of the rooms (0-based indexing)

For (k=0 to n-1):{
        i = n-1 + k;
        last_color_index= -1;
        while(i>=0){
                last_color_index= max(last_color_index, A[i]);
                i = (i-1)/2;
        }
        final_color[k]= color_order[last_color_index];
}
```

Time Complexity:

- Initialization of tree takes O(n) as we have 2n-1 nodes.
- In first for loop, each iteration would take O(log(n)) as it moves up by one level of tree in each iteration of while loop with constant no. of operations. So the first for loop would take O(m log(n)) in total.
- In next for loop, each iteration would take O(log(n)) as it will move from leaf to root node, which is equal to height no. of steps. So this for loop would take O(n log(n)) in total.
- Hence, Overall time complexity of algorithm becomes **O((m+n) log(n))**

Ans 4:

Approach:

- Use binary tree to respond to queries. Initialise a complete binary tree with leaf nodes as initial price of sweets.
- Internal(root also) nodes will store sum of prices of interval of sweets array which come under its subtree.
- **Request Query:** When a request (l,r) comes, we calculate sum of prices of interval (l,r) of sweets array as follows:
  1. Let u and v be the leaf nodes corresponding to sweet[l] and sweet[r].
  2. Keep repeating the following step as long as u < v :
     - If u is right child of its parent, add the value of node u to the sum and mark node right next to the current node as u.
     - If v is left child of its parent, add the value of node u to the sum and mark node left next to the current node as v.
     - Move up by one step simultaneously from u and v
  3. If both u and v are now same node, add the value of node to the sum.

  Output YES/NO on the basis of sum<=M or sum>M
- **Update Query:** When a request (i,x) comes, store the difference in new and old price in a variable *diff.* Update node corresponding to the i-th sweet. Now keep moving up towards root node and add the diff to each internal node on the path.

Pseudo-Code:

```
M <-Money; n <-no. of sweets;
vector<int> sweets; // contain initial prices of sweets( 0-based indexing)
int l,st;
If( n & (n-1) ) st= n; // if n is power of 2
Else st= 1<<(log2(n) + 1);
l= 2*st - 1;
vector<int> A(l,0);

for(int i=0;i<n; i++)  A[st+i-1]= sweets[i];   // leaf nodes initialization
for(int i =st-2;i>=0;i--)  A[i]= A[2*i+1] + A[2*i+2];  // internal nodes initialization
```

```
update(i,x){
        i = i-1; //bring in 0-based indexing
        i = n-1 + i; //node position in tree
        int diff= x - A[i]; // amount of change
        A[i]=x; // update
        i = (i-1)/2;
        while(i>=0){
                A[i]+= diff;
                i = (i-1)/2;
        }
}
```

```
request(l,r){
        int i= l-1; j= r-1; //bring in 0-based indexing
        i = n-1 + i; //start node position in tree
        j = n-1 + j; //end node position in tree

        int sum=0; //stores purchase price of l to r
        if(i==j) sum= A[i]; //trivial case
        Else if(j>i){
                while( j>i ){
                        if(i%2==0){
                                sum+= A[i];
                                i=i+1;
                        }
                        if(j%2==1){
                                sum+= A[j];
                                j=j-1;
                        }

                        i= (i-1)/2;
                        j= (j-1)/2;
                }
                if(i==j) sum+= A[i];
        }

        if(sum<=M) cout<<"YES\n";
        else cout<<"NO\n";
}
```

Time Complexity:

- Initialization of tree takes O(n) as we have 2n-1 nodes.
- Update function take O(log(n)) per query as it will move from leaf to root node, which is equal to height no. of steps.
- Request function would also take O(log(n)) per query as it also moves up by one level in each iteration with constant no. of operations.
- Since total no. of queries is n, overall time complexity of algorithm becomes **O(n log(n))**

Ans 5:

Approach:

- Create a parent array and using bfs() store the parents of each node.
- If first element of the required sequence is not 1 then it's not the valid BFS order.
- Create an empty queue Q and put first element of the sequence in it. And then sequentially for each element in the sequence:
    - Dequeue Q until front(Q) is parent of the element or the Q becomes empty.
    - If Q becomes empty, means either some non-sibling node has come in wrong order or parent was not in traversal before it, in short not a valid BFS order hence return false.
    - Else enqueue current element.
- If above steps complete without returning false means its a valid BFS order.

Pseudo-Code:

```
seq(n) //given sequence of size n
bfs(parent){
        CreateEmptyQueue(Q);
        Enqueue(1,Q);
        parent[1]=-1;

        While(Not IsEmptyQueue(Q)){
                v= Dequeue(Q);
                For each neighbor w of v {
                        if(w!=parent[v]){
                                parent[w]=v;Enqueue(w,Q);
                        }
                }
        }
}
Is_Edible(){
        parent(n+1); //stores parent of nodes in tree
        bfs(parent);
        if(seq[0]!=1)return false;

        CreateEmptyQueue(Q);
        Enqueue(1,Q);
        for( i=1 to n-1){
                while(Not IsEmptyQueue(Q) && parent[seq[i]] != Front(Q)){
                        Dequeue(Q);
                }
                if(IsEmptyQueue(Q)) return false;
                else Enqueue(seq[i],Q);
        }
        return true;
}
```

Time Complexity:

- Initializing parent array takes O(n) as Bfs algorithm works in O(n).
- While loop inside the for loop will iterate at most n times in overall, as at most n elements will be enqueued, hence at most n elements can be dequeued. So for loop and while loop combinedly take O(n).
- Hence time complexity for the algorithm is **O(n)**.