ESO207 Theoretical Assignment 3

P1 All or None

(a).
If only the sequence of nodes is given (without backtracking), it is impossible to recover the graph. Even if it is given after backtracking, still cannot recover all the original edges of graph, same with DFS tree. Thus not possible to find a tour with given condition.
For example, same sequence can correspond to two graphs (cyclic, acyclic).

(b).
If we are again given the sequence of BFS traversal (even with backtracking), it is not possible to recover all edges. This is because the last edge in a cycle simply wont get traversed.
Since, all edges in BFS tree are in the original graph, but such edges which complete a cycle in original graph may or may not be present in the BFS tree.
Thus, not possible to find (an Euler) tour.

(c).
We claim that it is possible to traverse each edge exactly once and return back to the starting city iff every vertex has an even degree.

("Only if" side is trivial since odd number of edges would mean, we either entered that vertex or left that vertex odd number of times, which is not possible for the source vertex, nor for the intermediate vertices.)

Start with some vertex v, we choose an edge and move further (this is always possible since we have even degree i.e. we have a different edge to leave once we enter a vertex). The number of edges being finite, we arrive back at vertex v. Let this be a cycle C. If all edges are covered then we have found the required tour.
If not, then let the subgraph excluding edges of C be S. All the connected components of S will be connected to cycle C (since the graph is connected). Now, we can invoke induction on the number of edges and thus all the connected components of S will also have such tours. Thus, the original graph will also admit such a tour since on the cycle C, if we reach a connection point (at which another connected component of S is connected), we complete that tour first and then further complete the cycle C.

(d).
We modify our DFS a bit and explore all unvisited edges in each traversal and mark those which are visited, and keep appending those in our (Euler) "tour". Thus, total time complexity is $O(|E|)$ or $O(|V| + |E|)$ if we intend to check if the graph is Eulerian or not.

`Pseudocode:`

```
function DFS(u, G, visited_edge, path)                ▷ O(|E|)
    tour.Append(path)
    for each v in G[u] do
        if !visited_edge[u][v] then
            Set visited_edge[u][v] and visited_edge[v][u] to true
            Call DFS(v, G, visited_edge, path)
        end if
    end for
```

```
        return path
    function GetDegree(G)                                          ▷ O(|V| + |E|)
        odd_deg_count ← 0
        odd_node ← −1
        for vertex i in G do
            if the degree of node i in G is odd then
                Increment odd_deg_count by 1
                Set odd_node to i
            end if
        end for
        if odd_deg_count is 0 then
            return (1, odd_node)
        else if odd_deg_count is 2 then
            return (0, odd_node)
        end if
    function CheckEuler(G)
        visited_edge ← [false]_{n×n}
        check, odd_node ← GetDegree(G)
        if check is 1 then
            Print("Graph has an Euler tour")
        else
            Print("Graph has no such tour")
        end if
        path ← DFS(start_node, G, visited_edge)
        Print(path)
```

P2 Chaotic Dino

(a).
We traverse the graph using BFS for a particular vertex upto *power* $\leftarrow x$ levels, and then recursively BFS again from those vertices which have towers. Now, if we somehow reach the destination in the process, we are done. We are not actually interested in getting the actual path from source to destination, just to check if that is a possibility.
Each BFS traversal takes $O(|V| + |E|)$ time and we'd do these traversals max for $n = |V|$ times. Thus, overall time complexity being $O(|V|(|V| + |E|))$.

`Pseudocode:`

```
visited = [false]_n
master_queue ← Queue()
function TOWER-RESEARCH(G, S, x, D)
    power ← x
    queue ← Queue()
    visited[S] ← True
    queue.enqueue(S)
    while !Q.isEmpty() and power ≥ 0 do
        node ← Q.dequeue()
        power ← power − 1
        for each neighbour in G[node] do
            if neighbour == D then
                PRINT("Possible")
                return true
            end if
            if !visited[neighbour] and HASTOWER(neighbour) then
                master_queue.enqueue(neighbour)
            end if
        end for
    end while
end function

function GET-TOWER(G, S, x, D)
    while !master_queue.isEmpty() do
        v ← master_queue.dequeue()
        if TOWER-RESEARCH(G, v, x, D) then
            return true
        end if
    end while
    return false
```

(b).
We can employ binary search on the value of power, x. The minimum and maximum possible values of x are 0 and the length of longest path from S to D respectively (we may take that as n as well).
Since again, the TOWER-RESEARCH function acts as a predicate function on the power x i.e. becomes true after a certain minimum power. Thus, we are able to apply binary search. Total time complexity is $O(\log n \cdot |V|(|V| + |E|))$ (log n times that of TOWER-RESEARCH)

`Pseudocode:`

```
function BIN-SEARCH(lo, hi, x, n)
    if !TOWER-reSEARCH(G, S, n, D) then
        return -1
    end if
    lo ← 0, hi ← n
    while lo ≤ hi do
        mid ← ⌊(lo+hi)/2⌋
        if !TOWER-reSEARCH(G, S, mid, D) then
            hi ← mid − 1
        else
            lo ← mid + 1
        end if
    end while
    return hi
```

P3 Room Colours

We create a segment tree with each node on it storing the colour for that particular segment. Nodes are created by dividing the original segment into two halves recursively.

Range-Assignment takes $O(h)$ where, h is the height of tree where $O(h) = O(\log n)$ (since we are recursively dividing segments into halves). Also, lookup takes $O(\log n)$ time.

Thus, for m updations (m bombings) and n lookups (for n rooms), we have total time complexity $O((m+n)\log n)$.

`Pseudocode:`

```
function PushDown(id)
    if painted[id] then
        t[2id] = t[2id + 1] = t[id]
        painted[2id] = painted[2id + 1] = true
        painted[id] = false
    end if
end function

function Update(id, x, y, l, r, c)                              ▷ O(log n)
    if l > r then
        return
    end if
    if l == x and y == r then
        t[id] = c
        painted[id] = true
    else
        PushDown(id)
        mid = (x+y)/2
        Update(2id, x, mid, l, min(r, mid), c)
        Update(2id + 1, mid + 1, y, max(l, mid + 1), r, c)
    end if
end function

function WhichColour(id, x, y, pos)                             ▷ O(log n)
    if x == y then
        return t[id]
    end if
    PushDown(id)
    mid = (x+y)/2
    if pos ≤ mid then
        return WhichColour(2id, x, mid, pos)
    else
        return WhichColour(2id + 1, mid + 1, y, pos)
    end if
end function

function Bomb-n-Final                                           ▷ O((m + n) log n)
    painted ← [false]_4n
    t ← [0]_4n
    for i in range(0, m) do
        l, r, c ← input
```

```
    UPDATE(0, 0, n − 1, l, r, c)        ▷ labelling root node as 0
end for
colour = []_n
for i in range(0, n) do
    colour[i] = WHICHCOLOUR(0, 0, n − 1, l, r, c)
end for
```

P4 Fest Fever

We can keep track of the cost of the sweets in an array and thus a segment tree implemented on it stores the range-sums.

Range-Updation takes $O(h)$ where, h is the height of tree, and since we are dividing our segments into halves recursively $O(h) = O(\log n)$
Thus, for n queries, the total time complexity is $O(n \log n)$

`Pseudocode:`

```
function BuildTree(prices, idx, l, r)                    ▷ O(4n) = O(n)
    if l == r then
        tree[idx] ← prices[l]
    else
        mid ← (l+r)/2
        BuildTree(prices, 2idx + 1, l, mid)
        BuildTree(prices, 2idx + 2, mid + 1, r)
        tree[idx] ← tree[2idx + 1] + tree[2idx + 2]
    end if
end function

function Sum-Query(idx, l, r, x, y)                       ▷ O(log n)
    if x ≤ l and r ≤ y then
        return tree[idx]
    end if
    if r < x or l > y then
        return 0
    end if
    mid ← (l+r)/2
    l_sum ← Sum-Query(2idx + 1, l, mid, x, y)
    r_sum ← Sum-Query(2idx + 2, mid + 1, r, x, y)
    return l_sum + r_sum
end function

function Update(idx, l, r, target, val)                   ▷ O(log n)
    if l = r = target then
        tree[idx] ← val
    else
        mid ← (l+r)/2
        if target ≤ mid then
            Update(2idx + 1, l, mid, target, val)
        else
            Update(2idx + 2, mid + 1, r, target, val)
        end if
        tree[idx] ← tree[2idx + 1] + tree[2idx + 2]
    end if
end function

function Handle-Query(a, b, c)                            ▷ O(log n)
    if a == 1 then
        result ← Sum-Query(0, 0, n − 1, b, c)
        if result ≤ M then
            Print("YES")
```

```
        else
            PRINT("NO")
        end if
    else if a == 2 then
        UPDATE(0, 0, n − 1, b, c)
    end if
end function

function EAT-SWEETS(n)                                    ▷ O(n log n)
    prices ← [0]_{4n}
    BUILDTREE(prices, 0, 0, n − 1)
    for i in range(0, n) do
        a, b, c ← input
        HANDLE-QUERY(a, b, c)
    end for
```

P5 Edible Sequence

We first run a modified BFS search on the graph and store the parent child relationships in a $n \times n$ matrix, as well as store the child count of a particular node. Now, we keep two pointers p and q where q points to the (probable) child of p and both of them are iterated over the given sequence and are checked if they are infact parent-child (q is iterated as much times as the child count of p), if not, we return "Not Edible" else "Edible".

We are basically checking nodes from consecutive levels in the BFS tree i.e. V_i, V_{i+1} and whether they occur in order i.e. If $2, 3$ occur in this particular order in sequence, then children of 2 should precede that of 3 as well.

`Pseudocode:`

```
function isSequenceEdible(T, sequence)
    n ← length(sequence)
    parent ← {}
    children ← {0}_n
    q ← 1
    visited ← {}
    queue ← Queue()
    start_node ← sequence[0]
    visited[start_node] ← True
    queue.enqueue(start_node)

    while !queue.isEmpty() do                 ▷ (BFS) O(|V| + |E|) = O(n + n − 1) = O(n)
        node ← queue.dequeue()
        for neighbour in T[node] do
            if neighbour not in visited then
                visited[neighbour] ← True
                queue.enqueue(neighbour)
                parent[node][neighbour] ← 1
                children[node] ← children[node] + 1
            end if
        end for
    end while

    for p in range(0, n) do                   ▷ O(n)
        for i in range(0, children[sequence[p]]) do
            if parent[sequence[p]][sequence[q]] ≠ 1 then
                return "Not Edible"
            else
                q ← q + 1
            end if
        end for
    end for

    return "Edible"
```