COMS 311 - Final Exam - ARYAN RAO

1. (i) $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$

$= 2^2 T\left(\frac{n}{2^2}\right) + 2n \Rightarrow 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n \ldots 2^k T\left(\frac{n}{2^k}\right) + kn$

$\Rightarrow \frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$\Rightarrow \log n = \log 2^k$

$\log n = k.1$

$2^k T(1) + kn$

$n.1 + n.\log n$

$= n + n\log n$ or $O(n \log n)$

(ii) $O(V+E)$

(iii) '2'

$\{A\}, \{B, C\}$

(iv) TRUE

(v) FALSE

2. Algorithm: MaximizeDifference(A, l, r)

Input:
- → A: array of integers with $a_1, a_2 \ldots a_n$
- → l: left index of subarray
- → r: right index of subarray

Output:
- → i: left index of max difference pair (a_i)
- → j: right index of max difference pair (a_j)

Pseudocode:

```
MaximizeDifference(A, l, r):

    if (l == r)
        return(l, r);

    mid = (l + r)/2;

    left_idx, left_max_diff = MaximizeDifference(A, l, mid)
            // recursively find max difference pair in left half

    right_idx, right_max_diff = MaximizeDifference(A, mid+1, r)
            // similarly right half
```

```
// Calculate max difference for subarray A[l:r+1]
left_min = min(A[l:mid+1])
right_max = max(A[mid+1:r+1])
diff = right_max - left_min

if diff > left_max_diff and diff > right_max_diff
    return (l, r)

else if left_max_diff > right_max_diff
    return(left_idx, left_max_diff)

else
    return (right_idx, right_max_diff)

// Call function with entire array A
n = len(A)
i, j = MaximizeDifference(A, 0, n-1)
```

Runtime:

The runtime of the algorithm can be analyzed using a recurrence relation:

$$T(n) = 2 * T(n/2) + O(n)$$

Here $T(n)$ represents time taken to find max diff in the array of size 'n'. '$2 * T(n/2)$' corresponds to the 2 recursive calls for the left and right halves. '$O(n)$' accounts for linear time taken to calculate max diff for subarray $A(l, r+1)$.

Using Masters Theorem, above recurrence falls into Case 1, where $a=2$, $b=2$ & $f(n)=O(n)$. Since $\log_b a = \log 2 = 1$, runtime:

$$T(n) = O(n \cdot \log n)$$

Justification:

The algorithm divides the array into two halves and then combines the results to find max diff. In each recursive step, subproblems are independent & equal in size.

Linear time complexity does not dominate the runtime as recursive calls take precedence. ~~Thus~~, Therfore, algorithm's runtime is efficiently bounded by $O(n \cdot \log n)$

M T W T F S S

3. The idea will be to make sure we travel the maximum distance possible before stopping for gas at each station.

Algorithm: FewestGasStop(d, m, D)

Input:
→ d: list of distances of gas stations from Metropolis
→ m: max distance the car can travel on full ~~tank~~ tank.
→ D: total distance from Metropolis - Gotham.

Output:
List of gas stations to stop at, in order to achieve the objective.

Pseudocode:

```
FewestGasStop(d, m, D):
    Stops = [];
    current_pos = 0;
    last_postop = 0;

for pos in d
    distance_to_next = pos - current_pos

if distance_to_next > m
    current_pos = last_stop
    Stops.append(last_stop)
```

```
last_stop = pos

if current_pos < 0
    stops.append(last_stop)

return stops
```

Runtime:

The algorithm iterates through the list 'd' of stations once. Each iteration takes constant time since the calculations involve basic arithmetic operations. Hence, runtime of the algorithm is 'O(n)', where n is number of gas stations.

Justification:

The algorithm takes advantage of given conditions and the greedy approach to make as few stops as possible. At each station, it checks whether the car can reach the next station while having gas. If not, it stops at the last stations & fills the tank. By doing so, the algorithm ensures the car covers max distance before each stop. Linear runtime complexity makes it efficient for practical purposes.

4. Independent set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set S of vertices such that for every two vertices in S, there's no edge connecting the two.

In a line graphs, each vertex is only connected to its previous & next vertex. This reduces the problem to a simpler form.

Dynamic Programming solution to this problem uses a recurrence relation that is based on optimal ~~sub~~ structure:

Single node, max weight independent set is the weight of this node itself.

Two nodes, max weight independent set is the max of the weights of the two.

Three or more, max weight independent set $M[n]$ is max of:
→ max weight independent set of first $n-1$ nodes $M[n-1]$.
→ " " " " : " $n-2$ nodes plus weight of the nth node $M[n-2] + w[n]$.

Recurrence Relation:

$$M[n] = \max(M[n-1], M[n-2] + w[n])$$

Algorithm: ~~MIS~~ MWIS (w)
↓
weights

Pseudocode:

```
MWIS(w):
    n = len(w)
    M = array of size n+1 with 0's
    M[1] = w[0]

    for i from 2-n:
        M[i] = max(M[i-1], M[i-2] + w[i-1])

    return M[n]
```

Runtime:

Time complexity of the algorithm is O(n), where n is the no. of vertices in the graph. We are iterating over the vertices once. Complexity for storing the M array is also O(n), therefore it is efficient.

Justification:

In the algorithm, w is the array of weights of nodes. It initializes M with base cases and then iteratively computes the max weight independent set for each number of ~~vertices~~ nodes from 2-n using recurrence.

5. We can use a DFS based algo, from each vertex in graph and mark all vertices that are reachable from that vertex. Once we have marked all reachable vertices for each starting vertex, we can identify non-dominating vertices as the ones that remain unmarked.

Algorithm: DetectNDVertex(G)

Input:
→ G: directed graph represented as an adjacency list/matrix.

Output:
→ boolean value indicating whether a non-dominating vertex exists.

Pseudocode:

```
dfs(graph, v, visited):
    visited[v] = true
    for u in graph[v]
        if not visited[u]
            dfs(graph, u, visited)

DetectNDVertex(G):
    V = len(G);
dominated = set()

for v in range(V):
    if v not in dominated:
        visited = [False]*V
        dfs(G, v, visited)
        dominated.update(i for i in range(V) if visited[i])
```

```
for v in range(V)
    if v not in dominated
        return True // Can also return vertices as well

return False
```

Runtime:

The algorithm performs a DFS traversal starting from each vertex in graph. In worst case, it visits all vertices & edges once. Therefore, time complexity of the algorithm is '$O(V+E)$', where 'V' is no. of vertices & 'E' no. of edges.

Justification:

The algorithm efficiently detects existence of a non-dominating vertex by performing a DFS traversal. By checking if there's any vertex left unmarked, we can identify non-dominating vertices. Linear complexity makes it efficient.

6. This can be done using Minimum Spanning Tree (MST) where MST of the new graph G' can be computed by simply checking whether the edge e whose weight can be included in the original MST 'T' without forming a cycle.

Pseudocode:

```
MST(G, T, e, S):
    add e to T to form T' (graph)
    Find cycle C in T'
    mEdge = findMaximumEdgeW(C)

    if mEdge != e
        remove mEdge from T'

    else
        remove e from T'
    Return T'
```

Runtime:

Time complexity of this algorithm is mainly determined by the time to detect a cycle & find the max edge in the cycle. Detecting a cycle is done in $O(V+E)$, where V is no. of vertices & E is no. of edges.

M T W T F S S

Justification:

The key insight in the algorithm is that when the weight of an edge decreases, the MST can be updated by removing an edge that was part of the cycle formed by adding the new edge. By identifying the edge with the max weight in the cycle & comparing it with the new edge, you can determine whether the new edge should be included in the MST or not.

7. If there is a certificate or evidence for each instance of 'yes' that can be verified as valid in polynomial time, then the situation is in NP (Nondeterministic Polynomial) time.

The certificate might be a legitimate color assignment for graph G in the Graph Colouring Problem. This can be checked by: (this will take constant amt of time)

→ Verifying that the colours of each vertex in the graph is distinct from the colours of all of its neighbours. Due to the need to check each edge once, this is 'O(n)'; n is the no. of vertices.

→ Verifying that there are no more than $\lceil \log n \rceil$ unique colours used in colouring. Counting unique colours & comparing them to $(\log n)$ is also a polynomial operation, as we go through each vertex once.

Due to the polynomial nature of certificate verification, the Graph Colouring problem is in NP.