ASSIGNMENT – 5

Introduction to Algorithms (CS 430)

1. Fractional Knapsack problem

```
function FractionalKnapsack(items, B):
    n = length(items)
    for i = 1 to n:
        items[i].unitValue = items[i].value / items[i].size
    sort items in descending order by unitValue
    totalValue = 0
    remainingCapacity = B
    for i = 1 to n:
        if remainingCapacity >= items[i].size:
            totalValue += items[i].value
            remainingCapacity -= items[i].size
        else:
            fraction = remainingCapacity / items[i].size
            totalValue += fraction * items[i].value
            break
    return totalValue
```

Explanation:

- We define a function *FractionalKnapsack* that takes the list of items and the knapsack capacity B as inputs.
- We calculate the unit value (value per unit size) for each item, and sort the items in descending order based on their unit values. After this, we initialize variables to keep track of the total value and remaining capacity.
- Iteration:
  - If an item can fit completely, we add its full value and reduce the remaining capacity.
  - If an item can only fit partially, we add a fraction of its value based on how much of it can fit, then end the loop.

Using this algorithm, we greedily select items with the highest value per unit size, which has been proven to yield an optimal solution for the Fractional Knapsack problem in Lecture 13.

2.

In order for Bob to take the following 10 language courses to take one language course each semester that allows Bob to satisfy all the prerequisites first we have to**:**

1. Build the digraph based on the prerequisites

2. Apply topological sorting to the digraph

Digraph

- LA15 has no prerequisites.
- LA16 has LA15 and LA17 as prerequisites.
- LA17 has LA15 as a prerequisite.
- LA22 has no prerequisites.
- LA31 has LA15 as a prerequisite.
- LA32 has LA16 and LA31 as prerequisites.
- LA126 has LA22, LA32, and LA127 as prerequisites.
- LA127 has LA16 as a prerequisite.
- LA141 has LA22 and LA169 as prerequisites.
- LA169 has LA32 as a prerequisite.

Topological Sorting

1. Identify vertices with no incoming edges (courses with no prerequisites):
    - LA15, LA22
2. Initialize a queue with these vertices:
    - Queue: [LA15, LA22]
3. Initialize an empty list for the result:
    - Result: [ ]
4. Process the queue:
    i. Remove a vertex from the queue
    ii. Add it to the result list
    iii. Remove its outgoing edges
    iv. If this creates new vertices with no incoming edges, add them to the queue

Final Topological Sorted Order:

LA 15, LA 22, LA 17, LA 31, LA 16, LA 32, LA 127, LA 169, LA 141, LA 126

3.

```
1. function hasCycle(G):
2.     visited = [false] * |V|
3.     parent = [null] * |V|
4.     for each vertex v in V:
5.         if not visited[v]:
6.             if DFS(G, v, -1):
7.                 return true
8.     return false
```

```
function DFS(G, v, parent):
    visited[v] = true
    for each neighbor u in adjacency_list[v]:
        if not visited[u]:
            if DFS(G, u, v):
                return true
        else if u != parent:
            return true
    return false
```

Time Complexity:

- In a sparse graph, |E| = O(|V|), which means the number of edges is linearly proportional to the number of vertices.
- The initialization of visited and parent arrays takes O(|V|) time.
- The outer loop iterates over all vertices, taking O(|V|) time.
- In the DFS function, each vertex is visited exactly once due to the visited array.
- For sparse graphs, the total number of edges explored across all DFS calls is O(|E|), which is O(|V|) for sparse graphs.
- ❖ Therefore, for sparse graphs, the total time complexity is O(|V|) + O(|E|) = O(|V|) + O(|V|) = O(|V|).

This algorithm correctly determines whether a sparse, undirected graph contains a cycle in O(|V|) time, meeting the required time complexity.

------ Another Solution ------

- Initialize an array called "visited" to keep track of visited vertices, initially set all values to false.
- For each vertex v in V, do the following:
    - o If the vertex v is not visited, call the DFS function starting from v.
- DFS function:
    - o Mark the current vertex as visited.
    - o For each neighbor u of the current vertex, do the following:
- If the neighbor u is not visited, recursively call the DFS function starting from u.
- If the neighbor u is visited and is not the parent of the current vertex, then a cycle exists in the graph.

Time Complexity:
The time complexity of this algorithm is O(V), as it performs a single DFS traversal of the graph, visiting each vertex exactly once. The time complexity is independent of the number of edges |E| since we are only considering the vertices. This algorithm efficiently checks for cycles in an undirected graph using a linear time complexity based on the number of vertices.

4.

```
function IsBipartite(G):
    color = {}  //Dictionary to store colors (0 or 1) for each vertex
    for each vertex v in G.V:
        if v not in color:
            if not BFS_Color(G, v, color):
                return False
    return True

function BFS_Color(G, start, color):
    queue = new Queue()
    queue.enqueue(start)
    color[start] = 0  //Assign initial color
    while queue is not empty:
        v = queue.dequeue()
        for each neighbour in G.adjacentVertices(v):
            if neighbour not in color:
                color[neighbour] = 1 - color[v]  //Assign opposite color
                queue.enqueue(neighbour)
            else if color[neighbour] == color[v]:
                return False  //Found adjacent vertices with same color
        //Explicitly handle different color case (no action needed)
            else:
                continue  //Neighbour has different color, continue
    return True
```

Time Complexity Analysis:

- Outer Loop in *IsBipartite*: Iterates through all vertices once: O(|V|) time.
- BFS in *BFS_Color*:
    - **Vertex Processing**: Each vertex is enqueued and dequeued at most once: O(|V|) operations.
    - **Edge Processing:** Each edge is examined exactly twice (once from each end): O(|E|) operations.

Therefore, the total time complexity is O(|V| + |E|), which is efficient and meets the required time complexity. This explicit handling ensures that all cases are considered and makes the algorithm more readable and understandable.