Boris Bespartochnyy, bpb56, Written Assignment 4

Question 1:

1a

Start at index 3, with the value of 2, then compare with the children at indices 7 and 8. no swap is needed.

Move to index 2, with a value of 8, then compare with the children at indices 5 and 6. Swap with 1.

Swap to index 1, with a value of 7, then compare with the children at indices 3 and 4. Swap with 2.

Move to index 0, with a value of 3, then compare with the children at indices 1 and 2, and swap with 1.

Final Array: $[1, 2, 3, 7, 5, 8, 3, 9, 10]$

1b

We remove 3, leaving us with the heap $[7, 4, 7, 9, 4, 8, 12]$.

After heapifying, we get $[4, 7, 4, 7, 9, 12, 8]$.

1c

Add 6 to the heap: $[1, 3, 2, 3, 4, 6]$.

No swaps are needed.

1d

We remove 6 from the heap: $[7, 13, 15, 25, 42, 14]$.

After heapifying, we get $[7, 14, 13, 15, 25, 42]$.

1e

Given the new heap $[3, 16, 3, 8, 12, 4, 3, 17, 10]$, compare 16 with its children, 8 and 12. Swap with 8.

Compare 16 with its new children, 17 and 10, then swap with 10.

After heapifying, we get $[3, 8, 3, 10, 12, 4, 3, 17, 16]$.

Recurrence Expansion

First, we expand the recurrence:

$$\begin{aligned} T(n) &= T(n-3) + 2n \\ &= T(n-6) + 2(n-3) + 2n \\ &= T(n-9) + 2(n-6) + 2(n-3) + 2n \end{aligned}$$

From the expansion, we see that the recurrence continues until $n - 3k \leq 0$, where k is the number of steps.

$$T(n) = c + \sum_{i=1}^{k} 2(n - 3(i-1))$$

Expanding the sum:

$$\begin{aligned} \sum_{i=1}^{k} 2(n - 3(i-1)) &= 2\sum_{i=1}^{k}(n - 3i + 3) \\ &= 2\left(kn - 3\frac{k(k-1)}{2} + 3k\right) \end{aligned}$$

Since $k \approx \frac{n}{3}$, we substitute k and simplify:

$$T(n) \approx c + \frac{(n^2 + 9n)}{3}$$

Graph Theory Problems

3a

The minimum possible size of S is 1. This happens because the graph is strongly connected, meaning every node can reach every other node. As a result, there is only one strongly connected component.

3b

The maximum possible size of S is V. This occurs when each node is its own strongly connected component, meaning there are no cycles in the graph. In this case, no node can reach any other node except itself.

4a

Yes, it is possible for the size of S to remain unchanged. Adding an edge that does not create new cycles will not alter the number of strongly connected components. For example, if we add an edge from strongly connected component A to B, where A and B are already in the same component, the number of strongly connected components remains the same.

4b

Yes, the size of S can decrease by $V-1$. This happens when an edge is added that connects all previously disconnected components into a single strongly connected component, reducing the number of strongly connected components by $V-1$. This occurs when the newly added edge forms a cycle that includes all nodes.

5

```
def updateMST(G, T, q, r, newWeight):
    G.updateEdgeWeight(q, r, newWeight)
    T.addEdge(q, r)
    cycle = findCycle(T, q, r)
    maxEdge = cycleMaxEdge(T, cycle)
    T.removeEdge(maxEdge)
    return T

def findCycle(T, q, r):
    visited = set()
    parent = {}
    stack = [(q, None)]
    while stack:
        node, p = stack.pop()
        if node in visited:
            cycle = []
            current = node
            while current != p:
                cycle.append(current)
                current = parent[current]
            cycle.append(p)
            return cycle
        visited.add(node)
        parent[node] = p
        for neighbor in T.adjacentNodes(node):
            if neighbor != p:
                stack.append((neighbor, node))
    return []

def cycleMaxEdge(T, cycle):
    maxWeight = -1
    maxEdge = None
    for i in range(len(cycle)):
        u = cycle[i]
        v = cycle[(i + 1) % len(cycle)]
        weight = T.getEdgeWeight(u, v)
        if weight > maxWeight:
            maxWeight = weight
            maxEdge = (u, v)
    return maxEdge
```

The algorithm functions by creating a cycle when the edge (q, r) is added to the MST. By removing the highest-weight edge in this cycle, the tree remains a minimum spanning tree.

The algorithm has a time complexity of $O(V)$ because the most computationally expensive operations are a depth-first search (DFS) to detect the cycle and a traversal to identify the maximum-weight edge, both of which run in linear time.

6a

G_1 and G_2 will not have the same shortest paths if 2 is added to all of G_1's edge weights. This is because adding a constant to every edge weight alters the relative differences between path lengths. For example, if one path consists of two edges, each with a weight of 1, and another path consists of a single edge with a weight of 3, the first path ($2 < 3$) would be shorter. However, after adding 2 to each edge, the first path's total weight becomes 6 ($3 + 3$), while the second path's weight becomes 5, making the second path the shorter option instead.

6b

G_1 and G_2 will have the same shortest paths if all of G_1's edge weights are multiplied by 2. This is because scaling all edges by the same positive factor preserves the relative proportions of path lengths, ensuring that the shortest paths remain unchanged.

7

```
def shortestPathForAll(graph):
    V = graph.vertices()
    distance = [[float('inf')] * V for _ in range(V)]
    for v in range(V):
        queue = []
        queue.append(v)
        distance[v][v] = 0
        while queue:
            u = queue.pop(0)
            for neighbor in graph.adjacentVertices(u):
                if distance[v][neighbor] == float('inf'):
                    distance[v][neighbor] = distance[v][u] + 2
                    queue.append(neighbor)
    return distance
```

Breadth-First Search (BFS) explores all nodes level by level, ensuring that when we reach a node, we have found the shortest path to it in terms of the number of edges. Since all edges have the same weight of 2, the number of edges directly corresponds to the path length. Therefore, BFS correctly finds the shortest paths.

The BFS for each vertex runs in $O(V + E)$ time. Since BFS is performed V times, the total runtime is $O(V(V + E))$.

8

```
def longestPathWeighted(G):
n = len(G.vertices())
    distance = [float('-inf')] * n
    distance[0] = 0
    path = [[] for _ in range(n)]
    path[0] = [v_1]
    for i in range(n):
        if i + 1 < n:
            if distance[i] + G.getEdgeWeight(v_i, v_{i+1}) > distance[i + 1]:
                distance[i + 1] = distance[i] + G.getEdgeWeight(v_i, v_{i+1})
                path[i + 1] = path[i] + [v_{i+1}]
        if i + 2 < n:
            if distance[i] + G.getEdgeWeight(v_i, v_{i+2}) > distance[i + 2]:
                distance[i + 2] = distance[i] + G.getEdgeWeight(v_i, v_{i+2})
                path[i + 2] = path[i] + [v_{i+2}]
    return path[n - 1]
```

This method works because the longest path to v_i can be constructed from the longest paths of v_{i-1} and v_{i-2}. After initializing the distance array, all distances are set to negative infinity, except for v_1, which is set to 0. A path array is also initialized.

We iterate through each vertex and update the distance and path arrays if a longer path is found. The runtime is $O(n)$ because the distance and path arrays are initialized in $O(n)$ time. The loop iterates over each vertex once, and for each vertex, we check and update the distances in constant time. Since we perform $O(1)$ work for each of the n vertices, the overall runtime is $O(n)$.