

Fundamental Algorithm Techniques

Problem Set #6

Problem 1:

1. Class Definition

```
1 class TreeNode:
2     def __init__(self, weight, n_children):
3         self.weight = weight
4         self.n_children = n_children
5         self.children = []
6
7     def add_child(self, child):
8         self.children.append(child)
```

2. Tree Generation (Depth = 3)

```
1 def create_tree(depth, n, parent_weight=1.0):
2     if depth == 0:
3         return None
4
5     node = TreeNode(parent_weight, n)
6
7     if depth > 1:
8         for _ in range(n):
9             child = create_tree(depth - 1, n, parent_weight / n)
10            node.add_child(child)
11
12    return node
13
14 tree = create_tree(3, 2)
```

3. Depth-First Search

```
1 def dfs_sum(node):
2     if node is None:
3         return 0
4     total = node.weight
5     for child in node.children:
```

```

6         total += dfs_sum(child)
7     return total
8
9 # Test
10 for n in [2, 3, 4]:
11     t = create_tree(3, n)
12     print(f"n={n}: {dfs_sum(t)}")

```

4. Breadth-First Search

```

1 from collections import deque
2
3 def bfs_sum(root):
4     if root is None:
5         return 0
6     total = 0
7     q = deque([root])
8     while q:
9         node = q.popleft()
10        total += node.weight
11        for child in node.children:
12            q.append(child)
13    return total
14
15 # Test
16 for n in [2, 3, 4]:
17     t = create_tree(3, n)
18     print(f"n={n}: {bfs_sum(t)}")

```

5. Sign Flipping on Revisit

```

1 def dfs_flip(node, visited=None):
2     if visited is None:
3         visited = {}
4     if node is None:
5         return 0
6
7     nid = id(node)
8     visited[nid] = visited.get(nid, 0) + 1
9     sign = 1 if visited[nid] % 2 == 1 else -1
10
11    total = sign * node.weight
12    for child in node.children:
13        total += dfs_flip(child, visited)
14    return total
15
16 def bfs_flip(root):
17     if root is None:
18         return 0
19     visited = {}
20     total = 0

```

```

21     q = deque([root])
22     while q:
23         node = q.popleft()
24         nid = id(node)
25         visited[nid] = visited.get(nid, 0) + 1
26         sign = 1 if visited[nid] % 2 == 1 else -1
27         total += sign * node.weight
28         for child in node.children:
29             q.append(child)
30     return total
31
32 # Test with n=2
33 t1 = create_tree(3, 2)
34 print(f"First\u2022pass\u2022DFS:\u2022{dfs_sum(t1)}")
35 t2 = create_tree(3, 2)
36 print(f"Sign\u2022flip\u2022DFS:\u2022{dfs_flip(t2)}")
37 print(f"Sign\u2022flip\u2022BFS:\u2022{bfs_flip(create_tree(3,\u00b22))}")

```

6. Recursive vs Non-Recursive BFS

```

1 # Recursive BFS (level-by-level)
2 def bfs_recursive(nodes):
3     if not nodes:
4         return 0
5     total = sum(n.weight for n in nodes)
6     next_level = []
7     for n in nodes:
8         next_level.extend(n.children)
9     return total + bfs_recursive(next_level)
10
11 # Non-Recursive BFS (with queue)
12 def bfs_iterative(root):
13     if root is None:
14         return 0
15     total = 0
16     q = deque([root])
17     while q:
18         node = q.popleft()
19         total += node.weight
20         for child in node.children:
21             q.append(child)
22     return total
23
24 # Compare
25 t1 = create_tree(3, 2)
26 t2 = create_tree(3, 2)
27 print(f"Recursive:\u2022{bfs_recursive([t1])}")
28 print(f"Iterative:\u2022{bfs_iterative(t2)}")

```

7. Why Recursive BFS is Not Recommended

Key Issues:

1. **Stack Overflow:** Each recursive call adds to the call stack. For deep or wide trees, this exhausts memory quickly.
2. **Inefficient:** Recursion creates function frames for every level. Queue-based iteration reuses the same loop context.
3. **Against BFS Nature:** BFS processes level-by-level, which is a natural queue operation. Recursion goes depth-first by design.
4. **Hard to Follow:** The recursive approach requires collecting all nodes at each level, which is indirect and confusing.
5. **Performance:** Function call overhead makes recursive BFS slower than iterative with queue.

Conclusion: Use **recursive** for DFS, use **iterative queue** for BFS.

8. Complete Solution

```

1 if __name__ == "__main__":
2     print("==TreeTraversalTests==\n")
3
4     # DFS test
5     print("DFS\u2225Results:")
6     for n in [2, 3, 4, 5]:
7         t = create_tree(3, n)
8         print(f"n={n}:{dfs_sum(t):.6f}")
9
10    # BFS test
11    print("\nBFS\u2225Results:")
12    for n in [2, 3, 4, 5]:
13        t = create_tree(3, n)
14        print(f"n={n}:{ bfs_sum(t):.6f}")
15
16    # Sign flip test
17    print("\nSign\u2225Flip\u2225(n=2):")
18    print(f"DFS:{dfs_flip(create_tree(3,2)):.6f}")
19    print(f"BFS:{ bfs_flip(create_tree(3,2)):.6f}")
20
21    # Recursive vs Iterative
22    print("\nRecursive\u2225vs\u2225IterativeBFS:")
23    t1 = create_tree(3, 2)
24    t2 = create_tree(3, 2)
25    print(f"Recursive:{ bfs_recursive([t1]):.6f}")
26    print(f"Iterative:{ bfs_iterative(t2):.6f}")

```