Fundamental Algorithm Techniques

Problem Set #6

Problem 1:

1. Class Definition

```python
class TreeNode:
    def __init__(self, weight, n_children):
        self.weight = weight
        self.n_children = n_children
        self.children = []

    def add_child(self, child):
        self.children.append(child)
```

2. Tree Generation (Depth = 3)

```python
def create_tree(depth, n, parent_weight=1.0):
    if depth == 0:
        return None

    node = TreeNode(parent_weight, n)

    if depth > 1:
        for _ in range(n):
            child = create_tree(depth - 1, n, parent_weight / n)
            node.add_child(child)

    return node

tree = create_tree(3, 2)
```

3. Depth-First Search

```python
def dfs_sum(node):
    if node is None:
        return 0
    total = node.weight
    for child in node.children:
```

```
        total += dfs_sum(child)
    return total

# Test
for n in [2, 3, 4]:
    t = create_tree(3, n)
    print(f"n={n}: {dfs_sum(t)}")
```

4. Breadth-First Search

```
from collections import deque

def bfs_sum(root):
    if root is None:
        return 0
    total = 0
    q = deque([root])
    while q:
        node = q.popleft()
        total += node.weight
        for child in node.children:
            q.append(child)
    return total

# Test
for n in [2, 3, 4]:
    t = create_tree(3, n)
    print(f"n={n}: {bfs_sum(t)}")
```

5. Sign Flipping on Revisit

```
def dfs_flip(node, visited=None):
    if visited is None:
        visited = {}
    if node is None:
        return 0

    nid = id(node)
    visited[nid] = visited.get(nid, 0) + 1
    sign = 1 if visited[nid] % 2 == 1 else -1

    total = sign * node.weight
    for child in node.children:
        total += dfs_flip(child, visited)
    return total

def bfs_flip(root):
    if root is None:
        return 0
    visited = {}
    total = 0
```

```
    q = deque([root])
    while q:
        node = q.popleft()
        nid = id(node)
        visited[nid] = visited.get(nid, 0) + 1
        sign = 1 if visited[nid] % 2 == 1 else -1
        total += sign * node.weight
        for child in node.children:
            q.append(child)
    return total

# Test with n=2
t1 = create_tree(3, 2)
print(f"First pass DFS: {dfs_sum(t1)}")
t2 = create_tree(3, 2)
print(f"Sign flip DFS: {dfs_flip(t2)}")
print(f"Sign flip BFS: {bfs_flip(create_tree(3, 2))}")
```

6. Recursive vs Non-Recursive BFS

```
# Recursive BFS (level-by-level)
def bfs_recursive(nodes):
    if not nodes:
        return 0
    total = sum(n.weight for n in nodes)
    next_level = []
    for n in nodes:
        next_level.extend(n.children)
    return total + bfs_recursive(next_level)

# Non-Recursive BFS (with queue)
def bfs_iterative(root):
    if root is None:
        return 0
    total = 0
    q = deque([root])
    while q:
        node = q.popleft()
        total += node.weight
        for child in node.children:
            q.append(child)
    return total

# Compare
t1 = create_tree(3, 2)
t2 = create_tree(3, 2)
print(f"Recursive: {bfs_recursive([t1])}")
print(f"Iterative: {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

```
if __name__ == "__main__":
    print("===TreeTraversalTests===\n")

    # DFS test
    print("DFS Results:")
    for n in [2, 3, 4, 5]:
        t = create_tree(3, n)
        print(f"n={n}:{dfs_sum(t):.6f}")

    # BFS test
    print("\nBFS Results:")
    for n in [2, 3, 4, 5]:
        t = create_tree(3, n)
        print(f"n={n}:{bfs_sum(t):.6f}")

    # Sign flip test
    print("\nSign Flip (n=2):")
    print(f"DFS:{dfs_flip(create_tree(3,2)):.6f}")
    print(f"BFS:{bfs_flip(create_tree(3,2)):.6f}")

    # Recursive vs Iterative
    print("\nRecursive vs IterativeBFS:")
    t1 = create_tree(3, 2)
    t2 = create_tree(3, 2)
    print(f"Recursive:{bfs_recursive([t1]):.6f}")
    print(f"Iterative:{bfs_iterative(t2):.6f}")
```