CS590 Algorithms - Project 2

Yiyang Gao, Ao Wang, John Rizzo

Due: April 22, 2025

Question 1: How you can break down a large problem instance into one or more smaller instances?

The problem can be modeled as a graph traversal with scoring constraints. Each node is connected to other nodes via neighbors, and a valid tour must consist of strictly increasing numeric values.

To apply dynamic programming, we define the subproblem as:

```
dp[i] = maximum score achievable starting from space i
```

Key ideas:

- The score from space i depends on the best scores from its neighbors with larger values ($value[j] > value[i]$).
- We can process nodes in increasing order of value to ensure acyclic transitions.
- Compute: `dp[j]` $= \max($`dp[i]` $+p(j))$ for all i such that $value[i] < value[j]$ and $i \to j$.

Why:

- Smaller subproblems build up to the full solution (optimal substructure)
- Recomputing the same nodes wastes time (hence memoization)
- The value rules create a natural 'upward only' path direction.

Question 2: What are the base cases of this problem, and what are their solutions?

A node with no neighbors that has a higher value cannot extend the tour. Thus, the base case is:

`dp[i] = p(i)`, where no neighbor j satisfies $value[j] > value[i]$.

Solution:

1. Set the node's base score:

   ```
   memo[nodeId] = nodes[nodeId].point; (Just its own value)
   ```

2. Return immediately:

   ```
   return nodes[nodeId].point; (No further recursion)
   ```

Question 3: What data structure would you use to store the partial solutions to this problem?

- `dp[i]`: an array to store the maximum score from space i.
- `prev[i]`: to store the predecessor of i in the optimal path.

These arrays provide $O(1)$ access and support fast updates and backtracking.

Question 4: Give pseudocode for an algorithm that uses memoization to compute the maximum score.

```
int[] memo = new int[n + 1];
Arrays.fill(memo, -1);

int maxScore(int nodeId) {
  if (memo[nodeId] != -1) return memo[nodeId];

  int maxVal = point[nodeId];
  for (int neighbor : neighbors[nodeId]) {
    if (value[nodeId] > value[neighbor]) {
      int current = maxScore(neighbor) + point[nodeId];
      if (current > maxVal) {
        maxVal = current;
        prev[nodeId] = neighbor;
      }
    }
  }

  memo[nodeId] = maxVal;
  return maxVal;
}
```

Question 5: What is the time complexity of your memoized algorithm?

- Each node is visited once: $O(n)$
- Each edge is checked once: $O(m)$
- Total: $\boxed{O(n+m)}$

Question 6: Give pseudocode for an iterative algorithm for this problem.

```
int[] dp = new int[n + 1];
int[] prev = new int[n + 1];
Arrays.fill(dp, 0);
Arrays.fill(prev, -1);

List<Integer> order = getSortedNodeIdsByValue();

for (int nodeId : order) {
    Node current = nodeMap.get(nodeId);
    if (dp[nodeId] == 0) dp[nodeId] = current.point;

    for (int neighborId : current.neighbors) {
        Node neighbor = nodeMap.get(neighborId);
        if (neighbor.value > current.value) {
            int newScore = dp[nodeId] + neighbor.point;
            if (newScore > dp[neighborId]) {
                dp[neighborId] = newScore;
                prev[neighborId] = nodeId;
            }
        }
    }
}
```

Question 7: Describe how you could modify your algorithm to identify the maximum-scoring tour, not just the maximum possible score.

To reconstruct the maximum-scoring tour, we maintain a `prev[]` array to store the predecessor of each node. After calculating the scores, we backtrack from the node with the maximum score to recover the full tour.

```
int maxScore = 0;
int maxNodeId = 0;

for (int i = 1; i <= n; i++) {
    if (dp[i] > maxScore) {
        maxScore = dp[i];
        maxNodeId = i;
    }
}

List<Integer> path = new ArrayList<>();
while (maxNodeId != -1) {
    path.add(maxNodeId);
    maxNodeId = prev[maxNodeId];
}
Collections.reverse(path);
```

The variable `maxScore` stores the final score, and `path` holds the optimal tour in order.

Bonus: Describe (briefly) how you would modify your algorithm to account for adjacent equal values and wildcards.

- **Equal values:** Allow movement to neighbors with equal values ($value[j] \geq value[i]$).
- **Wildcards:** Treat as compatible with any number; allow transitions from or to any value.

Challenges:

- Permits cycles: must track visited nodes to avoid infinite recursion.
- Graph may become dense due to wildcard transitions.

Approach:

- Extend DP to allow $value[j] \geq value[i]$.
- Preprocess wildcard nodes to connect to all neighbors.
- To avoid going in circles, we probably need to keep track of visited nodes - maybe using a HashSet to mark nodes we've already seen during the search.

This extension increases the time complexity and may lead to exponential behavior in the worst case.