Romerico David, COSC483.001, Midterm
Professor Nguyen
10/16/2024

Question 1

Step-by-Step Thought Process:

1. Create an array temp[] of size arr[].length to store the indices of all valid k-centers.
2. Start from index k and loop until arr.length – k – 1 which ensures you don't go out of bounds when checking elements k positions before and after index i.
3. For each index i, compare the element k positions before (arr[i - k]) and k positions after (arr[i + k]) the current index i.
4. If arr[i−k]==arr[i+k], store the current index i in the temp[] array, and increment the j pointer to keep track of valid indices.
5. Once all valid k-centers are identified and their indices are stored in temp[], initialize two variables:
    a. max: Stores the maximum value found at those positions.
    b. maxIdx: Stores the index of the element with the maximum value in temp[].
6. Loop through the temp[] array and determine the max value and maxIdx of arr[]
7. Return the index of the maximum k-center

Brute Force/Main Solution:

```
public static int getMaxKCenter(int[] arr, int k) {
        if (arr.length == 0) return -1;
    if (arr.length < 2 * k + 1) return -1;

        int[] temp = new int[arr.length];
        for (int i = k, j = 0; i < arr.length - k - 1; i++) {
            int minusKHop = arr[i - k];
            int plusKHop = arr[i + k];

            if (minusKHop == plusKHop) temp[j++] = i;
        }

        int max = temp[0];
        int maxIdx = 0;
        for (int i = 1; i < temp.length; i++) {
            if (arr[temp[i]] > max) {
```

```
            max = arr[temp[i]];
            maxIdx = i;
        }
    }

    return temp[maxIdx];
}
```

Proof of Correctness:

The algorithm identifies valid k-centers in the array, where a k-center at index i satisfies the condition arr[i−k]==arr[i+k]. It starts from index i=k and runs until i=arr.length−k−1, ensuring no out-of-bounds issues when checking elements at positions i−k and i+k. If these values are equal, the index i is stored in a temporary array, temp[], which holds all indices of valid k-centers.

After identifying the k-centers, the algorithm finds the k-center with the maximum value by comparing the values at other k-centers stored in temp[] with the current max value and its index and updating it respectively.

Big-O Analysis:

The first loop iterates from index k to arr.length−k−1, covering n−2k elements, where n is the length of the input array arr[]. In each iteration, the algorithm performs a constant-time comparison between two elements. Thus, the total time complexity of the first loop is O(n−2k) which simplifies to O(n).

The second loop iterates through the temp[] array, which holds the indices of all valid k-centers. In the worst case, if every element in the array is a valid k-center, temp[] could contain up to n−2k elements. For each element in temp[], the algorithm performs a constant-time comparison to find the maximum value, resulting in a total time complexity of O(n−2k) which again simplifies to O(n).

The overall time complexity of the algorithm is O(n)+O(n)=O(n), where n is the length of the input array.

In terms of space complexity, the temporary array temp[], which is used to store the indices of valid k-centers, can hold up to n−2k elements in the worst case, meaning it also requires O(n) space. Additionally, the algorithm uses a few constant-space variables such as i, j, max, and maxIdx, which require O(1) space. Therefore, the overall space complexity of the algorithm is O(n).

Question 2

Step-by-Step Thought Process:

Brute Force Approach: You would generate all possible combinations of weights while considering the maximum selection limits S[i]. For each combination, calculate the total weight and check if it is equal to or as close to the target capacity C as possible, while minimizing the number of selected weights.

1. For each weight W[i], generate all valid selections from 0 to S[i] times.
2. Check all combinations of weights that sum up to a value less than or equal to C.
3. For each valid combination, count the number of selected weights.
4. Choose the combination that meets C or is closest to it with the fewest number of weights.

Dynamic Programming Approach: In the dynamic programming approach, we aim to minimize the number of weights selected while trying to achieve a sum close to C.

1. Initialize the dp array which stores the min number of weights to reach each capacity from 0 to C. Set dp[0] = 0 and all other dp[j] values to Integer.MAX_VALUE.
2. Create combination array where each element is an empty ArrayList to store the combination of weights used for each capacity.
3. Iterate over each weight in the array W and for each weight W[i], iterate over capacities j from C down to W[i]. For each capacity j, try selecting the weight from 1 to S[i] times.
4. For each valid number of selections k (from 1 to S[i]), check if j >= k * W[i]. If true, check whether selecting the weight improves the current solution by comparing dp[j] with dp[j - k * W[i]] + k. If it improves, update dp[j] with the new minimum.
5. Update the combination[j] list by copying the combination from combination[j - k * W[i]] and adding W[i], k times to this list.
6. After all weights are processed, if dp[C] is still Integer.MAX_VALUE, no valid combination was found. Otherwise, return the combination[C] list, which contains the optimal combination of weights that sum up to C.

Brute Force Solution:

```
public static ArrayList<Integer> bruteForceKnapsack(int[] W, int[] S, int C) {
        ArrayList<Integer> bestCombination = new ArrayList<>();
        int[] minWeights = { Integer.MAX_VALUE };

        bruteForce(new ArrayList<>(), 0, 0, W, S, C, bestCombination, minWeights);

        return bestCombination;
    }

    private static void bruteForce(ArrayList<Integer> currentCombination, int currentSum, int idx, int[] W, int[] S, int C,
                                   ArrayList<Integer> bestCombination, int[] minWeights) {
        if (currentSum > C) return;

        if (currentSum == C) {
            if (currentCombination.size() < minWeights[0]) {
                minWeights[0] = currentCombination.size();
                bestCombination.clear();
                bestCombination.addAll(currentCombination);
            }
            return;
        }

        if (idx == W.length) {
            return;
        }

        for (int count = 0; count <= S[idx]; count++) {
            ArrayList<Integer> newCombination = new ArrayList<>(currentCombination);
            for (int i = 0; i < count; i++) {
                newCombination.add(W[idx]);
            }
            bruteForce(newCombination, currentSum + W[idx] * count, idx + 1, W, S, C, bestCombination, minWeights);
        }
    }
```

Dynamic Programming Solution:

```
public static ArrayList<Integer> dpKnapsack(int[] W, int[] S, int C) {
        int[] dp = new int[C + 1];
```

```
    for (int i = 1; i < dp.length; i++) {
      dp[i] = Integer.MAX_VALUE;
    }

    ArrayList<Integer>[] combination = (ArrayList<Integer>[]) new ArrayList[C + 1];
    for (int i = 0; i <= C; i++) {
      combination[i] = new ArrayList<>();
    }

    for (int i = 0; i < W.length; i++) {
      for (int j = C; j >= 0; j--) {
        for (int k = 1; k <= S[i]; k++) {
          if (j >= k * W[i] && dp[j - k * W[i]] != Integer.MAX_VALUE) {
            if (dp[j] > dp[j - k * W[i]] + k) {
              dp[j] = dp[j - k * W[i]] + k;

              combination[j] = new ArrayList<>(combination[j - k * W[i]]);
              for (int count = 0; count < k; count++) {
                combination[j].add(W[i]);
              }
            }
          }
        }
      }
    }

    return dp[C] == Integer.MAX_VALUE ? new ArrayList<>() : combination[C];
  }
```

Proof of Correctness:

The brute force solution tries every possible combination of the given weights. The proof of correctness hinges on the following:

1. Exhaustive Search: The brute force solution generates every possible subset of the weights that can be selected. For each weight W[i], the algorithm explores all options from using the weight 0 times to S[i] times. By exploring all combinations, it ensures that every valid way to form the target capacity C is considered.

2. For each valid combination that sums up to a value less than or equal to C, the algorithm checks whether it uses the fewest number of weights. It keeps track of the combination with the minimum number of selected weights that either equals or is closest to C. This ensures the final result is the solution with the fewest number of weights.

The proof of correctness for the dynamic programming solution hinges on the following:

1. Initialization: The base case is initialized correctly with dp[0] = 0, meaning no weights are needed to reach a capacity of 0. All other capacities are initially set to Integer.MAX_VALUE representing unreachable capacities
2. For each weight W[i] and each capacity j, the DP solution checks how many times the weight can be used (from 1 to S[i]). If selecting the weight improves the solution for capacity j i.e. dp[j] = min(dp[j], dp[j - k × W[i]] + k) where k is the number of times the weight W[i] is selected), the solution is updated. This relation ensures that each dp[j] is computed optimally by considering all possible ways of achieving the capacity j using the current weight.
3. The DP table builds the solution for capacity C by using the solutions for smaller capacities. For example, to compute dp[C], the algorithm relies on the solutions for capacities C – k × W[i]. This ensures that if dp[j] is updated, it has the minimum number of weights needed to reach capacity j because of the optimal solutions of smaller capacities.
4. After processing all weights, the value dp[C] holds the minimum number of weights required to achieve the capacity C. If dp[C] = Integer.MAX_VALUE, it means no valid combination of weights can sum up to C, which is handled by returning an empty result.

By maintaining and updating optimal solutions for smaller subproblems and correctly building up to the final solution, the DP approach guarantees the correct minimum number of weights that sum up to C.

Big-O Analysis:

The brute force approach explores all possible combinations of weights, taking into account how many times each weight can be selected. For each weight W[i], the algorithm tries every option from selecting it 0 times to S[i] times. This results in trying all combinations, which leads to an exponential number of possibilities. For n weights, if each weight can be selected up to S[i] times, the total number of combinations to check is on the order of $O(S1 \times S2 \times \cdots \times Sn)$. The space complexity is O(n) because the algorithm stores the current combination being explored and the best combination found so far.

The time complexity of the dynamic programming approach is O(n×C×Smax), where n is the number of weights, C is the target capacity, and Smax is the maximum number of times any weight can be selected. For each weight W[i], the algorithm iterates over all capacities j from C down to 0 and tries selecting the weight up to S[i] times, resulting in the O(n×C×Smax) time complexity. The space complexity is O(C) because the algorithm stores the minimum number of weights required for each capacity from 0 to C, along with the combinations that achieve those capacities.

Question 4

Step-by-Step Thought Process:

When I first approached the problem of removing duplicates from an array, I would iterate through each element in the array and compare it against the elements I had already seen. If it wasn't a duplicate, I would store it in a temporary array. To implement this, I created a new array temp where I would store unique values. For each element in the original array, I checked if it was already in temp by looping through the array I had built so far. If it wasn't there, I added it and kept track of how many unique elements I had stored. Once I had processed all elements, I copied the non-duplicate values from temp into a result array of the correct size. This solution worked but was inefficient

Then, I thought about using a Set, which by design only allows unique elements and avoid the need for manual duplicate checking entirely. By switching to a LinkedHashSet, I knew I could maintain the order of elements as well. This meant I could simply iterate through the array once, adding elements directly into the Set and it would automatically discard duplicates for me and maintain the first occurrences of each duplicate.

Brute Force Solution:

```
public static double[] removeDuplicatesBruteForce(double arr[]) {
        if (arr.length == 0) return new double[0];

        double[] temp = new double[arr.length];
        int size = 0;

        for (int i = 0; i < arr.length; i++) {
            boolean isDuplicate = false;
            for (int j = 0; j < size; j++) {
                if (arr[i] == temp[j]) {
                    isDuplicate = true;
                    break;
                }
            }

            if (!isDuplicate) temp[size++] = arr[i];

        }

        double[] result = new double[size];
        for (int i = 0; i < size; i++) result[i] = temp[i];

        return result;
    }
```

Main Solution:

```
public static double[] removeDuplicatesOptimalApproach(double arr[]) {
    if (arr.length == 0) return new double[0];

    LinkedHashSet<Double> temp = new LinkedHashSet<Double>();
    for (int i = 0; i < arr.length; i++) temp.add(arr[i]);

    double[] result = new double[temp.size()];
    int i = 0;
    for (double val : temp) result[i++] = val;

    return result;
  }
```

Proof of Correctness:

The brute-force solution iterates through each element in the input array and checks if it has already been added to the temp array. If the element is not a duplicate, it is added to temp. Once all elements have been processed, the unique elements are transferred to the result array. Since every element is compared to previously encountered elements and only unique values are stored, this approach correctly removes duplicates.

The optimized solution uses a LinkedHashSet to store unique elements. The Set data structure inherently ensures that no duplicates are stored and the LinkedHashSet preserves the order of elements. The final result array is constructed from this Set and guarantees that all elements are unique and in their original order.

Big-O Analysis:

The brute-force solution involves a nested loop. For each element in the array, the algorithm checks whether the element already exists in the temp array. This leads to a worst-case of $O(n^2)$, where n is the number of elements in the input array. This is because for each element, the inner loop may need to iterate over all previously stored elements. The space complexity is O(n) because the algorithm creates an extra temp array temp to store unique elements, which can grow to the size of the input array.

The optimized solution uses a Set which allows for average constant-time insertions. The time complexity of this solution is O(n) because each element in the array is inserted into the Set in constant time. After that, copying the elements from the Set to the result array takes O(n) time, so the overall time complexity remains O(n). The space complexity is O(n) since the LinkedHashSet stores up to n unique elements and the result array also requires O(n) space.

Question 5

Step-by-Step Thought Process:

Brute Force Approach:

In the brute force solution, we explore all possible paths from the top row to the bottom row using recursion. From each cell, we recursively explore the three possible moves (down, down-left, and down-right) and calculate the total number of coins collected for each path. And we keep track of the path taken to collect those coins. We choose the path that gives the maximum coins and return the corresponding path.

1. Start at each cell in the first row.
2. For each cell, recursively move down, down-left, or down-right, and collect the maximum number of coins by exploring all paths.
3. Keep track of the path taken along each recursive call.
4. Return the maximum coins collected and the path that results in collecting the most coins.

Dynamic Programming Approach:

We create a 2D DP table where dp[i][j] represents the maximum coins the pirate can collect starting from cell (i,j) and reaching any cell in the last row. Additionally, we maintain a path table to keep track of the column from which the maximum coins were collected for each cell in order to reconstruct the path after filling the DP table.

1. Initialize the DP table with the coin values from the first row of the grid and the Path table.
2. For each cell (i,j), calculate the maximum coins that can be collected by coming from the three possible directions in the previous row (up-left, up, and up-right). Update both the dp table and the path table simultaneously.
3. Use the DP table to store the maximum coins collected so far for each cell, and the path table to track the previous column that led to this cell.
4. Find the maximum value in the last row for the total coins collected and trace back the path using the path table.
5. Return the path by backtracking through the path table.

Brute Force Solution:

```
public static ArrayList<Integer> findMaxCoinsBruteForce(int[][] grid) {
    int N = grid.length;
```

```java
    int M = grid[0].length;
    ArrayList<Integer> maxPath = new ArrayList<>();
    int maxCoins = 0;

    for (int col = 0; col < M; col++) {
        ArrayList<Integer> path = new ArrayList<>();
        int resultCoins = maxCoinsBruteForce(grid, 0, col, N, M, path);
        if (resultCoins > maxCoins) {
            maxCoins = resultCoins;
            maxPath = path;
        }
    }

    return maxPath;
}

public static int maxCoinsBruteForce(int[][] grid, int row, int col, int N, int M, ArrayList<Integer> path) {
    if (row == N) return 0;

    int currentCoins = grid[row][col];
    ArrayList<Integer> bestPath = new ArrayList<>();
    int maxCoins = 0;

    int downCoins = maxCoinsBruteForce(grid, row + 1, col, N, M, bestPath);
    if (downCoins > maxCoins) {
        maxCoins = downCoins;
        path.clear();
        path.addAll(bestPath);
    }

    if (col - 1 >= 0) {
        ArrayList<Integer> downLeftPath = new ArrayList<>();
        int downLeftCoins = maxCoinsBruteForce(grid, row + 1, col - 1, N, M, downLeftPath);
        if (downLeftCoins > maxCoins) {
            maxCoins = downLeftCoins;
            path.clear();
            path.addAll(downLeftPath);
        }
    }

    if (col + 1 < M) {
        ArrayList<Integer> downRightPath = new ArrayList<>();
        int downRightCoins = maxCoinsBruteForce(grid, row + 1, col + 1, N, M, downRightPath);
        if (downRightCoins > maxCoins) {
            maxCoins = downRightCoins;
            path.clear();
            path.addAll(downRightPath);
        }
    }

    path.add(0, currentCoins);
    return currentCoins + maxCoins;
}
```

Dynamic Programming Solution:

```
public static ArrayList<Integer> findMaxCoinsDP(int[][] grid) {
        int N = grid.length;
        int M = grid[0].length;

        int[][] dp = new int[N][M];
        int[][] pathTable = new int[N][M];

        for (int col = 0; col < M; col++) {
            dp[0][col] = grid[0][col];
            pathTable[0][col] = -1;
        }

        for (int row = 1; row < N; row++) {
            for (int col = 0; col < M; col++) {
                int maxFromAbove = dp[row - 1][col];
                int fromCol = col;

                if (col > 0 && dp[row - 1][col - 1] > maxFromAbove) {
                    maxFromAbove = dp[row - 1][col - 1];
                    fromCol = col - 1;
                }

                if (col < M - 1 && dp[row - 1][col + 1] > maxFromAbove) {
                    maxFromAbove = dp[row - 1][col + 1];
                    fromCol = col + 1;
                }

                dp[row][col] = grid[row][col] + maxFromAbove;
                pathTable[row][col] = fromCol;
            }
        }

        int lastCol = 0;
        for (int col = 0; col < M; col++) {
            if (dp[N - 1][col] > dp[N - 1][lastCol]) lastCol = col;
        }

        ArrayList<Integer> path = new ArrayList<>();
        int currentCol = lastCol;
        for (int row = N - 1; row >= 0; row--) {
            path.add(0, grid[row][currentCol]);
            if (row > 0) currentCol = pathTable[row][currentCol];
        }

        return path;
    }
```

Proof of Correctness:

The brute force solution exhaustively searches all possible paths from any cell in the first row to any cell in the last row. For each cell, the algorithm considers every possible valid move (down, down-left, and down-right) and recursively calculates both the total number of coins collected and the path taken along all these paths. This guarantees that the path yielding the maximum number of coins will be found.

For the dynamic programming solution, at each cell dp[i][j], the maximum number of coins collected up to that point is determined by considering the best possible move from the previous row (either coming straight down, from the left, or from the right).

By storing these optimal values in the DP table, we ensure that each subproblem is solved optimally. And by maintaining a path table to track where the maximum coins were collected from, we ensure that the correct path can be reconstructed. Thus, the solution guarantees that the path with the most coins is selected.

Big-O Analysis:

The time complexity of the brute force solution is $O(3^{N \times M})$, where N is the number of rows and M is the number of columns. This is because for each cell in the first row, we recursively explore up to three possible moves (down, down-left, and down-right) for each subsequent row, resulting in an exponential number of possible paths. Specifically, each recursive call generates up to three more calls for the next row, leading to $3^{N \times M}$ paths to explore for each column in the first row, and with M columns, the total time complexity becomes $O(3^{N \times M})$. The space complexity is O(N) because of the depth of the recursion stack. The recursive calls go N levels deep (one for each row in the grid) and is independent of the number of columns.

The time complexity of the dynamic programming solution is $O(N \times M)$. As we iterate through all the cells in the grid exactly once, for each cell, we compute the maximum coins that is collectable by considering the previous row’s values from three possible directions (straight up, up-left, and up-right). There is constant work for each cell and finally results in a total time complexity of $O(N \times M)$. The space complexity is $O(N \times M)$ because we maintain a DP and Path table of size $N \times M$ to store the maximum coins for each cell.