CS 514

Quiz 3 – Greedy and Dynamic Programming Algorithms

Q1.

Answer-

Code –

```
def max_satisfied_dogs(h_level, b_size):
    # Sort the arrays in descending order
    h_level.sort(reverse=True)
    b_size.sort(reverse=True)
    # Initialize pointers and count
    i, j, satisfied_dogs = 0, 0, 0
    # Iterate through hunger levels and biscuit sizes
    while i < len(h_level) and j < len(b_size):
        # Check if the current biscuit can satisfy the
current hungry dog
        if b_size[j] >= h_level[i]:
            # Dog is satisfied, mark the biscuit as used
            satisfied_dogs += 1
            j += 1
        i += 1
    # Return the count of satisfied dogs
    return satisfied_dogs


hunger_level = [2, 3]
biscuit_size = [3, 3]
print(max_satisfied_dogs(hunger_level, biscuit_size))
```

a. Describe a greedy algorithm to solve this problem.

As we know, in greedy algorithms the solution is constructed through a sequence of steps and at each decision point, the choice that seems the best out of all alternatives is chosen.

In this problem, we are given an array of hunger levels of dogs (hunger_level [1…n]) and another array of biscuit sizes (biscuit_size [1...m]). Each dog can have its hunger satisfied by only one biscuit of its hunger level or of size greater than its hunger level.

Optimization problem -

Given a set of hungry dogs with specific hunger levels and a set of biscuits with varying sizes, the goal is to maximize the number of satisfied dogs. Each dog can only be satisfied by a biscuit of size greater than or equal to its hunger level, and each biscuit can be given to at most one dog.

Greedy approach -

The greedy approach taken to solve this problem is to make locally optimal choices at each step to achieve a globally optimal solution. In this context, the greedy problem is to choose the largest available biscuit for each hungry dog, starting with the hungriest ones. By prioritizing the largest hunger levels and selecting the largest available biscuits, the algorithm seeks to maximize the overall satisfaction of dogs.

Describing my code –

Step 1. Sort the input arrays in Descending Order -This is important as it ensures that the greedy approach starts by considering the hungriest dogs and the largest biscuits first.

Step 2. Initializing the pointers and variables - The two pointers (i for hunger levels and j for biscuit sizes) move independently through their respective sorted arrays. The algorithm considers the next hungry dog (i) and tries to find the largest available biscuit (j) that can satisfy its hunger. The variable "satisfied_dogs" which keeps track of number of satisfied dogs is assigned to 0.

Step 3. Iteration – The choice of selecting the largest available biscuit for each hungry dog is implicit in the sorting of arrays. The while loop processes the hungry dogs (i) and available biscuits (j) in descending order, making the greedy choice to address the largest hunger levels first.

- The condition b_size[j] >= h_level[i] checks whether the current biscuit size is greater than or equal to the current hunger level.
- If this condition is true, it means the dog can be satisfied, and the corresponding biscuit is marked as used.

Once a biscuit is found to satisfy a dog (when the condition is true), the variable j is incremented to move to the next available biscuit. This ensures that each dog receives at most one biscuit.
The variable "satisfied_dogs" is incremented when a suitable biscuit is found for a hungry dog, effectively counting the number of satisfied dogs.

The iteration efficiently addresses the hungriest dogs first due to the descending order sorting.
The greedy choice is evident in the selection of the largest available biscuit for each hungry dog. This choice aims to optimize the overall satisfaction of dogs by addressing the largest hunger levels first.

b. Analyze the time complexity of your approach.

We go through the code, line by line to check the time complexity.

1. Sorting an array takes O(N log N) time, where N is the length of the input arrays. Here, as we are sorting arrays h_level and b_size, the combined time complexity will be **O(nlogn + mlogm)**, where n and m are the lengths of these two arrays, respectively.
2. The while loop iterates through the sorted arrays once. In the worst case, both arrays are fully traversed. Each iteration of the loop performs constant-time operations (comparisons and increments). In this step, the combined time complexity will be **O(n + m).**

Hence, the overall time complexity of the max_satisfied_dogs function is dominated by the sorting step, which is **O(nlogn + mlogm).**

So, considering the worst-case scenario, the time complexity is O(NlogN) where N is max(len(hunger_level), len(biscuit_size)).

2.

a. **Dynamic Programming paradigm (using bottom up approach)–**

// Function to find the number of distinct ways to arrange blocks of length 1 and 2 to get a total length of N
function countWaysDP(N):
// Create an array to store the number of ways for each length from 0 to N
dp = new Array(N + 1)

// Base cases
dp[0] =0
dp[1] = 1
dp[2] = 2

// Filling the array using bottom-up approach
for i from 3 to N:
dp[i] = dp[i - 1] + dp[i - 2]

// The final result is stored in dp[N]
return dp[N]

Python code –

```
def count_ways_dp(N):
    # Create an array to store the number of ways for
each length from 0 to N
    dp = [0] * (N + 1)
    # Base cases
    if N == 0:
        return 0
    elif (N == 1):
        return 1
    elif (N == 2):
        return 2
    else:
        dp[0] = 1
        dp[1] = 1
        dp[2] = 2
        # Fill the dp array using bottom-up approach
        for i in range(3, N + 1):
            dp[i] = dp[i - 1] + dp[i - 2]
```

```
    # The final result is stored in dp[N]
    return dp[N]
```

In the dynamic programming approach, we use a bottom-up approach to fill in a 1D array (dp) of size N. The time complexity is determined by the loop that iterates from 3 to N, where each iteration involves constant-time operations (addition and assignment). Therefore, the time complexity is O(N). This is because we perform a constant amount of work for each unit increase in N.

b. Brute Force Approach

```
// Function to find the number of distinct ways to arrange blocks of length 1 and 2 to get a total length of N
function countWays_BF (N):
   // Base case: if N is 0 or negative, there is only one way (no blocks)
   if N == 0:
      return 0
  else if N == 1:
     return 1
  else if N == 2:
     return 2
 else:
    // Recursive case: count ways by considering both 1-unit and 2-unit blocks
    return countWays_BF(N - 1) + countWays_BF(N - 2)
```

In the brute force approach, we use recursion to calculate the number of ways to arrange blocks. The recursion tree expands exponentially, as each function call spawns two more calls until we reach the base cases. The number of function calls grows in a manner like the Fibonacci sequence.

Let's denote T(N) as the number of function calls for a given N. The recurrence relation is T(N) = T(N−1) + T(N−2). This is analogous to the Fibonacci sequence, and the time complexity is exponential, O(2^N).
The reason for the exponential time complexity is that, at each level of the recursion tree, we have two recursive calls, leading to an exponential growth in the number of function calls. The repeated calculations of the same subproblems make this approach inefficient.

c. Time Complexity Comparison –

The dynamic programming approach has a time complexity of O(N), where N is the input length. This is because we fill in a 1D array of size N once.
The brute force approach, on the other hand, has an exponential time complexity of O(2^N). This is because for each length, it recursively considers two possibilities (1-unit or 2-unit blocks) until it reaches the base case.

d. Recurrence Formula –

The recurrence formula for the problem can be expressed as follows:
dp[i] = dp[i−1] + dp[i−2]

This formula states that the number of ways to form a length of i using 1-unit and 2-unit blocks is the sum of the number of ways to form lengths (i-1) and (i-2).