CSE 321 - Introduction to Algorithm Design

Homework 5 – Report

Ömer Faruk Bitikçioğlu

161044010

1) This function finds the longest common substring present in a list of strings. It does this by using a divide-and-conquer approach: it divides the strings in half and recursively finds the longest common substring present in each half. If the longest common substring found in the left half of the strings has a length equal to the length of the first half of the strings, then the longest common substring must also be present in the right half, and the function concatenates the two substrings to form the complete longest common substring. If the longest common substring found in the left half is shorter than the length of the first half of the strings, then it is not present in the right half and the function returns the substring found in the left half.

The function first checks for the base cases where the input list is empty or has only one element. If the input list has more than one element, it then checks if all the strings in the list are equal. If they are, it returns the first string as the longest common substring. If the strings are not all equal, the function divides the strings in half and recursively finds the longest common substring in each half. It then concatenates the two substrings if the longest common substring found in the left half has a length equal to the length of the first half of the strings. If the longest common substring found in the left half is shorter, it returns that substring.

The worst-case time complexity of the longest_common_substring function is O(n * m), where n is the number of strings in arr_strings and m is the maximum length of the strings in arr_strings. This is because the function performs a linear search through arr_strings at each level of recursion and divides the strings in arr_strings in half at each level, leading to a total of m levels of recursion.

In the worst case, the function will divide the strings in arr_strings in half until they are all single characters, and then it will perform a linear search through arr_strings at each level of recursion to check if they are all the same. This results in a time complexity of O(n * m).

2.a) The find_max_profit_dac_helper() method is a recursive function that takes as input the prices list and the indexes list (which represent the indices of the prices list). It has two base cases:

- If the indexes list has only one element, the method returns without doing anything.
- If the indexes list has two elements, the method compares the two elements and updates the buy_index, sell_index, and max_profit attributes if necessary.

For all other cases, the method divides the prices and indexes lists in half and calls itself recursively on each half. It then compares the minimum value in the left half of the prices list with the maximum value in the right half and updates the buy_index, sell_index, and max_profit attributes if necessary. Finally, it returns without doing anything.

This divide-and-conquer approach allows the algorithm to find the maximum profit in a relatively efficient manner, by dividing the input list into smaller and smaller pieces and finding the maximum profit in each piece, until the base cases are reached.

2.b) This method is used to find the maximum profit that the merchant can make by buying goods at a low price and selling them at a higher price, using a linear-time algorithm.

The method first resets the buy_index, sell_index, and max_profit attributes to their default values. If the input prices list has fewer than two elements, the method returns without doing anything. Otherwise, it initializes two variables idx_buy and idx_sell to 0 and the last index of the prices list, respectively. It then enters a loop that continues until idx_buy and idx_sell are equal (which means that the prices list has been fully traversed). Inside the loop, the method compares the element at index idx_buy with the element at index idx_sell. If the element at idx_buy is less than or equal to the element at idx_sell, it calculates the profit as the difference between the two elements and updates the buy_index, sell_index, and max_profit attributes if necessary. It then increments idx_buy by 1. If the element at idx_buy is greater than the element at idx_sell, it decrements idx_sell by 1.

This continues until idx_buy and idx_sell are equal, at which point the loop exits. When the loop exits, the find_max_profit_linear() method returns without doing anything.

2.c) The divide-and-conquer algorithm has a worst-case time complexity of O(n log n), where n is the length of the input list. This is because the algorithm divides the input list in half at each recursive step, and it performs O(n) work on each sublist before combining the results. In the worst case, the input list is divided into sublists of size 1 at the bottom of the recursion, resulting in log n recursive steps.

The linear-time algorithm has a worst-case time complexity of O(n), because it iterates through the input list once, performing a constant amount of work on each element. This approach making it more efficient than the divide-and-conquer algorithm in the worst case. However, it may not be as efficient as the divide-and-conquer algorithm in the average case, depending on the specific input data.

3) The dp array is used to store the length of the longest increasing sub-array ending at each index i in arr_nums. The for loop updates the dp array by comparing the element at index i to the element at index i-1. If the element at index i is greater than the element at index i-1, it updates the element at index i in the dp array to be the element at index i-1 plus 1. This way, the dp array stores the length of the longest increasing sub-array ending at each index i in arr_nums.

After the for loop finishes, the function finds the maximum value in the dp array and stores it in a variable called max_len. This represents the length of the longest increasing sub-array in arr_nums. It then finds the index of this maximum value in the dp array and stores it in a variable called idx. This represents the last element of the longest increasing sub-array in arr_nums. Finally, it creates a sub-array called sub_array which is a slice of arr_nums starting from idx + 1 - max_len and ending at idx + 1. This represents the longest increasing sub-array in arr_nums.

The function then returns a tuple containing max_len and sub_array. This allows the caller of the function to both get the length and the sub-array of the longest increasing sub-array in arr_nums.

The worst-case time complexity of the function longest_increasing_subarray is O(n). This is because the function has a single for loop that iterates over the elements of the input array arr_nums, which has a length of n. Therefore, the time complexity of the function is linear with respect to the size of the input.

In the worst case, the input array is in non-increasing order, meaning that the for loop will have to iterate over all n elements of the array before completing. Therefore, the time complexity of the function in the worst case is O(n).

4.a) This code is an implementation of a dynamic programming (DP) algorithm to find the path with the maximum number of points in a 2D game map. The algorithm works by storing the maximum number of points at each coordinate in a 2D array dp. The algorithm first initializes the dp array with all elements set to 0. It then iterates through the game map and fills the dp array with the maximum number of points that can be gained at each coordinate.

The algorithm checks if the current element is in the first row or column of the map. If it is, it sets the value of the element in the dp array to the value in the map plus the value of the element in the dp array to the left or above it (depending on whether the element is in the first row or column). If the element is not in the first row or column, the algorithm sets the value of the element in the dp array to the maximum of the elements to the left and above it in the dp array, plus the value of the element in the map.

After the dp array has been filled, the algorithm sets the max_points variable to the value of the last element in the dp array, which represents the maximum number of points that can be gained in the game. It then initializes the path list with the last element in the map and iterates through the dp array in reverse, starting from the last element. The algorithm checks if the current element is in the first row or column of the map. If it is, the algorithm breaks out of the loop. Otherwise, it checks which element to the left or above has the higher value in the dp array and adds the corresponding element in the map to the start of the path list. The algorithm continues this process until it reaches the first element in the map.

4.b) This code is an implementation of a greedy algorithm to find the path with the maximum number of points in a 2D game map. The algorithm works by starting from the first element in the map and choosing the element to the right or down with the highest value at each step. The algorithm initializes the path list with the first element in the map and the max_points variable with the value of the first element in the map. It then initializes the current row and column indices, i and j, with 0.

The algorithm then enters a loop that continues until it reaches the last row and column of the map. Inside the loop, the algorithm checks if the element to the right or down has a higher value. If both elements have the same value, the algorithm chooses the element to the right. If the element to the right or down has a higher value, the algorithm chooses that element and increments the corresponding index by 1. The algorithm then adds the element to the end of the path list and updates the max_points variable with the value of the element in the map.

The algorithm continues this process until it reaches the last row and column of the map. At that point, it returns the path list and the max_points variable, which represent the path with the maximum number of points and the maximum number of points, respectively.

4.c) The brute-force solution for finding the path with the maximum number of points in a 2D game map involves generating all possible paths and comparing the number of points gained by each path. This solution is guaranteed to find the correct path with the maximum number of points, as it considers all possible paths. However, the time complexity of this solution is exponential, as the number of paths grows exponentially with the size of the map. This makes the brute-force solution impractical for large maps, as it takes a long time to complete.

The dynamic programming (DP) solution for this problem involves storing the maximum number of points at each coordinate in a 2D array and iterating through the map to fill the array. This solution is also guaranteed to find the correct path with the maximum number of points, as it considers all possible paths and chooses the path with the maximum number of points at each step. The time complexity of this solution is linear in the size of the map, as the number of elements in the map is equal to the number of elements in the dp array. This makes the DP solution more efficient than the brute-force solution, as it takes much less time to complete for large maps.

The greedy solution for this problem involves starting from the first element in the map and choosing the element to the right or down with the highest value at each step. This solution is not guaranteed to find the correct path with the maximum number of points, as it only considers the best choice at each step without considering the overall effect on the path. The time complexity of this solution is also linear in the size of the map, as the algorithm only needs to iterate through the map once. However, the greedy solution may be less efficient than the DP solution in certain cases, as it may choose a suboptimal path that does not lead to the maximum number of points.