GTU Department of Computer Engineering
CSE 321 - Fall 2022
Homework 5 Report

Emirkan Burak Yılmaz
1901042659

1. Longest Common Prefix

   lo: low position inside the array to indicate start index.

   hi: high position inside the array to indicate end index

   - If the lo is equal to hi, then just return the string in the lo index.
   - If the lo is smaller than the hi, apply the below processes.
     - First divide the array into halves and recursively find the common prefix of halves.
     - Compare the two common prefixes of halves and return longest common prefix between them.

   In worst case, algorithm requires to check all the characters of all the strings. For that reason, the time complexity of the algorithm is O(N*M) where N is the number of strings in the array and M length of each string (same for worst case).

2. Max Profit of Merchant
   a. Divide-and-Conquer Approach

      prices: array that contains day by day market prices.
      N: length of prices array.
      lo: low position inside the prices array to indicate start index.
      hi: high position inside the prices array to indicate end index.

      - Basically, algorithm divides array into two parts, the max profit could come from left sub array, right sub array or mixture of them (max price in left - min price in right).
      - For the base case, check if lo is lower than hi. If it's not, then return lo for all return values.
      - Split the array into two parts and recursively find the max profit of left and right parts.
      - For the left part, function call ended up four return value lbuy, lsell, lmin and lmax values. The values lbuy and lsell indicate buying and selling indices for max profit on the left part and lmin and lmax for the minimum and maximum encountered prices on the left part. And similarly, we have, rbuy, rsell, rmin, rmax for the right part.
      - We split the problem into two subproblem and now it's time to combine the result. The max profit could come from left sub array, right sub array or lmax - rmin (max price in left - min price in right). So, there are three cases that need to be considered.
      - Compare the three cases and set variables buy and sell to proper values.
      - Set variables min and max for the maximum and minimum value of the whole array by comparing lmax, rmax and rmin, lmin.
      - Return buy, sell, min and max values.

This algorithm divides the current problem into two subproblems and then combines them in constant time and this can be represented as the below recurrence relation.

$T(N) = 2T(N/2) + O(1)$

So, by applying master theorem, time complexity of this algorithm can be found as O(N). Since the algorithm contains recursive calls the space complexity is S(logN).

b. Non-Divide-and-Conquer Approach

prices: array that contains day by day market prices.
N: length of the prices array.

- Keep four variables buy and sell for the most profitable days, mini for the index of encountered minimum price and maxProfit to keep track of maximum profit.
- Set buy and mini to 0 and sell 1.
- Set maxProfit as prices[1] – prices[0]
- Loop from i = 1 to N - 1. Inside the loop, there are two condition that needs to be consider.
    - If the difference of prices[i] and the prices[mini] is larger than maxProfit which means that more profitable sell and buy day found, then set buy to mini and sell to i.
    - Else if prices[i] is less than prices[mini] which indicates that a new minimum price is found, then set mini to i. With this way, more profitable days can be found with the first case.
- After execution of the loop, buy and sell gives the buying and selling days for the maximum profit.

With this implementation, a single loop through prices is enough to find the most profitable buy and sell days. So, the time complexity of this algorithm is O(N).

c. Comparison
Both algorithms have linear time complexity. However, the algorithm that follows the divide-and-conquer approach contains more calculations and this makes it slower than the other algorithm. On the other hand, it also requires additional space of stack. So, for this situation non-divide-and-conquer algorithm seems better.

3. Longest Increasing Sub-Array

   N: number of elements in the given array

   - Set an array size N named length initialized with 1. This array will be used to keep the length of the increasing sub-array that finishes at that index.
   - Set variable named maxLength with value of 0 to keep the maximum length of the sub-array.
   - Loop over the array from index 1 to last index.
   - Inside loop check if the previous element is smaller than the current element.
   - If it is smaller, then set the sub-array length of the current element to previous sub-array length + 1 and update maxLenght, if the length of the current subarray becomes larger than maxLength.
   - Return maxLength.

   Algorithm loops over all the elements in the array, for that reason this is a linear algorithm. So, the time complexity of this algorithm is O(N). Since dynamic programming approach used in this algorithm, it also requires additional space of N. So, the space complexity is S(N).

4. Max Points

   a. Dynamic Programming Approach

      R = number of rows
      C = number of columns
      grid = game grid

      - Set a RxC 2D array named dp and initialize it with 0. This 2D array will be used to keep the maximum points for that location.
      - Set dp[0][0] to grid[0][0].
      - For the first row and first column there is only one path available. So maximum point at that locations are can be easily found.
      - Fill the first row by dp[i][0] = dp[i - 1][0] + grid[i][0] where 0 < i < R.
      - Fill the column by dp[0][i] = dp[0][i] + grid[0][i] where 0 < i < C.
      - For the remaining cells there are two options. For example, grid[i][j] can be reached from grid[i - 1][j] or grid[i][j - 1]. To collect maximum points on the path, the one gives more points should be selected.
      - Create a nested loop i from 1 to R and and j from 1 to C and inside loop apply dp[i][j] = grid[i][j] + max(dp[i - 1][j], dp[i][j - 1]).
      - After filling dp by making decision in each cell, the maximum score is found at dp[R – 1][C - 1].
      - The path can be easily found by starting from the end (dp[R - 1][C - 1]) and following the larger score either moving left or up to the start position.

In this algorithm, each grid cell visited once to create dp array. Then creating the path is just found by applying R + C - 2 movement. So, the running time of this algorithm is $O(N^2)$ where N is the max(R, C).

b. Greedy Approach

R = number of rows
C = number of columns

- Start from position A_1B_1.
- Select greedily the cell that has larger points and continue with that cell recursively by either right or down movement.
- Apply the above logic by considering the boundaries of the game map until A_RB_C is reached.

In this algorithm, only R + C - 2 decisions are made. For that reason, only one path is created. For that reason, time complexity is O(N) where N is R + C.

c. Comparison

So far, we have three algorithms which are implemented by considering different approaches. For sake of simplicity, let's call Algo1 as brute force approach, Algo2 as dynamic programming approach and Algo3 for the greedy approach. When we consider the time complexity of those algorithms, Algo3 is the quickest and Algo1 is the slowest one. However, Algo3 does not necessarily give the right answer every time as the nature of greedy algorithms. So greedy approach is not reliable for this problem.

When we eliminate Algo3 and consider only Algo1 and Algo2, Algo1 is slower than Algo2 as we said earlier. However, this gain of speed has an additional space cost. The space complexity of Algo1 is constant and its quadratic for Algo2. So, there is a trade-off in space and time. For speed considerations, Algo2 which follows dynamic programming approach will be more suitable. On the other hand, for space considerations Algo1 which follows brute force approach will be more suitable