05/01/2023

CSE321

Introduction to Algorithm Design

Homework 5

Abdurrahman Bulut
1901042258

Question 1:

The code is in the python file..

The divide-and-conquer algorithm in pseudocode form:

```
function LONGEST_SUBSTRING(strings, start, end):
  if start > end:
    return ""
  middle = (start + end) / 2
  for i = 1 to length of strings:
    if strings[0][middle] != strings[i][middle]:
      return LONGEST_SUBSTRING (strings, start, middle - 1)
  return strings[0][start:middle + 1] + LONGEST_SUBSTRING (strings, middle + 1, end)
```

This algorithm takes in the array of strings “strings”, the start index “start”, and the end index “end” of the substring to be checked as input, and returns the longest common substring at the beginning of all the strings as output. It uses a binary search approach to find the longest common substring by repeatedly dividing the substring to be checked in half and checking the characters in the middle of the substring. If the characters are the same in all the strings, it continues checking the characters in the second half of the substring. If they are not the same, it continues checking the characters in the first half of the substring. It returns the longest common substring at the beginning of all the strings once it has been found.

The worst-case time complexity of the divide-and-conquer algorithm to find the longest common substring at the beginning of an array of strings is O(nlogn), where n is the length of the longest string in the array. This is because the algorithm uses a binary search approach to find the longest common substring, which has a time complexity of O(logn). The time complexity is further reduced to O(n) if all the strings in the array have the same length. In the worst case, the algorithm may have to perform the binary search on the entire length of the longest string in the array in order to find the longest common substring. This results in a time complexity of O(nlogn).

For example, consider the input ["aaa", "aab", "aac"]. In this case, the algorithm would have to perform the binary search on the entire length of the longest string "aaa" in order to find the longest common substring, which is the empty string. This would result in a time complexity of O(3log3) = O(3 * 1.58). On the other hand, if the input is ["aa", "aa", "aa"], the algorithm would only have to perform the binary search on the first character of the strings, resulting in a time complexity of O(1).

Question 2-a:

The code is in the python file..

The code in Python file is an implementation of the divide-and-conquer algorithm for finding the maximum profit that can be made by buying and selling goods. The algorithm has a time complexity of O(nlogn). This is because at each step of the recursive process, the list of prices is divided in half, and the function is called on each half. This means that the time complexity is logarithmic with respect to the length of the list of prices.

Here is a more detailed breakdown of the time complexity:

The base case (when the length of the list of prices is 0 or 1) takes O(1) time.

The list of prices is divided in half at each recursive step, which takes O(n) time.

The function is called recursively on each half of the list of prices, which takes O(log n) time (since the list is halved at each step).

The minimum and maximum prices are found in the left and right halves of the list, which takes O(n) time.

Therefore, the overall time complexity is O(nlogn).

Here is an example of how the function would work with the input [10, 11, 10, 9, 8, 7, 9, 11]:

Divide the list of prices into two halves: [10, 11, 10] and [9, 8, 7, 9, 11].

Recursively find the maximum profit by buying and selling in the left and right halves.

For the left half, divide the list into [10] and [11, 10].

For the left half, return 0 (no profit can be made).

For the right half, return 1 (the difference between 11 and 10).

For the right half, divide the list into [9, 8] and [7, 9, 11].

For the left half, return 1 (the difference between 8 and 7).

For the right half, return 4 (the difference between 11 and 7).

Find the maximum profit by buying in the left half and selling in the right half.

The minimum price in the left half is 10, and the maximum price in the right half is 11.

The maximum profit is 1 (the difference between 11 and 10).

Return the maximum of the profits from the left and right halves and the cross profit (4).

Question 2-b:

The code is in the python file..

The max_profit function first checks if the input list is empty or has only one element. If this is the case, it returns 0, since no profit can be made.

Next, it initializes the min_price and max_profit variables to the first element in the list. These variables are used to keep track of the minimum price seen so far, and the maximum profit that can be made by selling at the current price.

Then, the function iterates through the list of prices. At each step, it updates the min_price if the current price is lower than the current minimum price. It also updates the max_profit if it can be increased by selling at the current price. To do this, it compares the difference between the current price and the minimum price seen so far to the current max_profit. If the difference is greater, it updates max_profit to this difference.

Finally, the function returns the max_profit.

Initialize the minimum price and maximum profit to the first price in the list.

Time complexity: O(1)

Iterate through the list of prices.

Time complexity: O(n)

At each step, update the minimum price if the current price is lower, and update the maximum profit if it can be increased by selling at the current price.

Time complexity: O(1)

Return the maximum profit.

Time complexity: O(1)

The overall time complexity of this algorithm is O(n), since it only needs to iterate through the list of prices once. This makes it faster than the divide-and-conquer algorithm in the worst case, which has a time complexity of O(nlogn).

Here is an example of how the function would work with the input [10, 11, 10, 9, 8, 7, 9, 11]:

Initialize min_price to 10 and max_profit to 0.

Iterate through the list of prices:

min_price is unchanged at 10.

max_profit is updated to 1 (the difference between 11 and 10).

min_price is unchanged at 10.

max_profit is unchanged at 1.

min_price is updated to 8.

max_profit is updated to 3 (the difference between 8 and 5).

min_price is unchanged at 8.

max_profit is updated to 4 (the difference between 9 and 8).

Return 4 as the maximum profit.

Question 2-c:

The code is in the python file..

The first algorithm, which uses a divide-and-conquer approach, has a worst-case time complexity of O(n log n). This is because at each step of the recursive process, the list of prices is divided in half, and the function is called on each half. This means that the time complexity is logarithmic with respect to the length of the list of prices.

The second algorithm, which uses a linear approach, has a worst-case time complexity of O(n). This is because it only needs to iterate through the list of prices once, regardless of the length of the list.

The time complexity of an algorithm is a measure of how the running time of the algorithm increases as the input size increases. In the case of these two algorithms, the input size is the length of the list of prices.

In the worst case, the divide-and-conquer algorithm will take longer to run than the linear algorithm. This is because the time complexity of the divide-and-conquer algorithm is higher than that of the linear algorithm. Specifically, the divide-and-conquer algorithm has a time complexity of O(n log n), while the linear algorithm has a time complexity of O(n).

Here is a comparison of the two algorithms in terms of their worst-case time complexity:

Divide-and-conquer algorithm: O(nlogn)

Linear algorithm: O(n)

In general, algorithms with a lower time complexity will run faster in the worst case than algorithms with a higher time complexity. Therefore, the linear algorithm will be faster than the divide-and-conquer algorithm in the worst case.

Question 3:

The Code is in the python file..

First, we define a function “longest_increasing_subarray(arr)” that takes in an array “arr” and returns the longest increasing sub-array.

We declare a variable “n” which is the length of the input array. Then, we initialize two arrays dp and “start” with size “n”, and fill them with 1s and 0s respectively. The array “dp” stores the length of the longest increasing sub-array ending at each index i, while the array “start” stores the start index of the sub-array ending at index i.

We then iterate through the input array with a for loop, starting from index 1. For each element “arr[i]”, we check if it is greater than the previous element “arr[i-1]”. If it is, we update the value of “dp[i]” as “dp[i-1] + 1”, and the value of “start[i]” as “start[i-1]”. This is because if “arr[i]” is greater than “arr[i-1]”, we can extend the longest increasing sub-array ending at index “i-1” by adding “arr[i]” to it. The start index of the sub-array remains the same because we are only adding to the end of the sub-array.

If “arr[i]” is not greater than “arr[i-1]”, we set the value of “start[i]” to “i”. This is because we cannot extend the previous sub-array, so the longest increasing sub-array ending at index “i” starts at “i”.

After the for loop, we find the maximum value in the “dp” array, which is the length of the longest increasing sub-array. We then find the index of this maximum value, which is the end index of the sub-array. Using this end index and the start index stored in “start”, we can return the sub-array using list slicing: “arr[start[end]:end+1]”.

The time complexity of the modified algorithm is O(n), since we are iterating through the input array once. The time complexity of the original algorithm is also O(n), since it only has a single for loop.

In both algorithms, the time complexity is dominated by the for loop, which has a time complexity of O(n). The other operations, such as element comparison, array indexing, and assignment, have a constant time complexity and do not affect the overall time complexity.

Therefore, the time complexity of the modified algorithm to find the longest increasing sub-array and print it is O(n).

Question 4-a:

The code is in the python file..

Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems, storing the solutions to these subproblems in an array or other data structure, and using these stored solutions to solve larger subproblems.

In this case, the problem of finding the highest possible score by traversing the 2D board can be broken down into smaller subproblems of finding the maximum score for each coordinate. We store the solutions to these subproblems (i.e., the maximum score at each coordinate) in the "scores" array and use these stored solutions to solve the larger subproblem of finding the maximum score for the entire board.

The time complexity of this code is O(nm), since the algorithm has two nested loops that each iterate over all the coordinates in the board . The outer loop runs in O(n) time and the inner loop runs in O(m) time, so the total time complexity is O(nm).

This means that the running time of the algorithm is directly proportional to the size of the input board . If the board has n rows and m columns, the algorithm will take O(n*m) time to complete.

The space complexity of this code is also O(nm), since the "scores" array has n rows and m columns and takes up O(nm) space.

Question 4-b:

The code is in the python file..

The code defines a function highest_score that takes in three parameters:

n: the number of rows in the board

m: the number of columns in the board

board: a 2D array representing the board , with each element representing the score at that coordinate

The function first initializes the starting coordinate (i and j) to be the top-left corner of the board , and the initial score to be the score at this coordinate. It also initializes an empty list path to keep track of the path taken through the board .

The function then enters a loop that continues until it reaches the final coordinate (the bottom-right corner of the board). Inside the loop, it checks if moving down or right gives a higher score. If moving down gives a higher score (or if we are at the rightmost column and can only move down), it updates the row index i to move down. If moving right gives a higher score (or if we are at the bottommost row and can only move right), it updates the column index j to move right. In either case, it adds the score at the new coordinate to the total score and appends the new coordinate to the path list.

Finally, the function returns the total score after it has reached the final coordinate.

The greedy algorithm does not always find the optimal solution, as it makes decisions based on the current optimal solution without considering the consequences of these decisions on future steps. The optimal solution may depend on the path taken to reach it, so a greedy algorithm that always chooses the path with the highest immediate reward may not find the optimal solution.

The time complexity of this code is O(n+m), since the loop will run for a maximum of n+m iterations (once for each row and once for each column).

This means that the running time of the algorithm is directly proportional to the size of the input board . If the board has n rows and m columns, the algorithm will take O(n+m) time to complete.

The space complexity of this code is O(n+m), since the path list will have a maximum of n+m elements and take up O(n+m) space.

Question 4-c:

Dynamic programming solution: This solution is guaranteed to be correct, as it uses a bottom-up approach to systematically find the optimal solution by considering all possible paths. The worst-case time complexity of this solution is O(nm), since it has two nested loops that each iterate over all the coordinates in the board.

Greedy algorithm solution: This solution is not guaranteed to be correct, as it makes decisions based on the current optimal solution without considering the consequences of these decisions on future steps. The optimal solution may depend on the path taken to reach it, so a greedy algorithm that always chooses the path with the highest immediate reward may not find the optimal solution. The worst-case time complexity of this solution is O(n+m), since the loop will run for a maximum of n+m iterations (once for each row and once for each column).

Brute-force solution from hw4: This solution is guaranteed to be correct, as it tries every possible route and chooses the one with the highest score. The worst-case time complexity of this solution is $O(n^2m^2)$, since it has two nested loops that each iterate over all the coordinates in the board.

In general, the dynamic programming solution is the most efficient in terms of time complexity, followed by the greedy algorithm and then the brute-force.