CSE321 – HW5

Bahadir Etka Kilinc
1901042701

Question 1.

One approach to solving this problem using a divide-and-conquer algorithm is as follows:

- If the input array is empty, return an empty string.
- If the input array has only one element, return that element.
- Divide the input array into two equal-sized subarrays.
- Find the longest common substring at the beginning of each subarray using this divide-and-conquer algorithm.
- Find the longest common substring at the beginning of all strings in the input array by comparing the substrings found in steps 4 and 5.
- Return the longest common substring found in step 5.
  This algorithm has a time complexity of O(n log n), where n is the number of strings in the input array.

To implement this algorithm in code, we can use a recursive function that takes the input array and the start and end indices of the subarray to process as arguments. The base cases would be when the start index is greater than or equal to the end index (return an empty string) or when the start index is equal to the end index (return the string at that index). The recursive case would involve dividing the subarray into two equal-sized subarrays, finding the longest common substring at the beginning of each subarray, and then finding the longest common substring at the beginning of all strings in the input array by comparing the substrings found in the two subarrays.

Here is some sample code in Python that demonstrates this approach:

```
def longest_common_substring(strings, start, end):
    if start >= end:
        return ""
    elif start == end - 1:
        return strings[start]
    else:
        mid = (start + end) // 2
        left = longest_common_substring(strings, start, mid)
        right = longest_common_substring(strings, mid, end)
        result = ""
        for i in range(min(len(left), len(right))):
            if left[i] == right[i]:
                result += left[i]
            else:
                break
        return result

strings = ["programmable", "programming", "programmer", "programmatic",
"programmability"]
print(longest_common_substring(strings, 0, len(strings)))
```

Question 2.

(a) Divide-and-conquer algorithm:

1 If the input array has length 0 or 1, return 0.
2 Divide the input array into two halves.
3 Recursively find the maximum profit that can be made from buying and selling in the left half of the array.
4 Recursively find the maximum profit that can be made from buying and selling in the right half of the array.
5 Find the maximum profit that can be made by buying in the left half and selling in the right half. This can be done by finding the minimum value in the left half and the maximum value in the right half and taking the difference.
6 Return the maximum of the profits from steps 3, 4, and 5.

(b) Linear time algorithm:

1 Initialize variables min_price and max_profit to the first element of the input array.
2 Iterate through the input array, starting at the second element. For each element, do the following:

- Update min_price to the minimum of min_price and the current element.
- Update max_profit to the maximum of max_profit and the difference between the current element and min_price.

3. Return max_profit.

(c) Comparison of the two algorithms:

The divide-and-conquer algorithm has a worst-case time complexity of O(n log n), as it divides the input array in half at each recursive step and therefore has logarithmic time complexity. The linear time algorithm has a worst-case time complexity of O(n), as it processes each element of the input array once. Therefore, the linear time algorithm is faster in the worst case.

However, the divide-and-conquer algorithm may be more suitable for certain scenarios where the input array can be divided and conquered more efficiently, such as when the elements of the array are already partially sorted. The linear time algorithm is generally faster and more efficient, but it requires a single pass through the entire input array, which may not be practical in some cases.

In terms of worst-case time complexity, the divide-and-conquer algorithm has a time complexity of O(n log n) because it divides the input array in half at each recursive step and therefore has logarithmic time complexity. The linear time algorithm has a time complexity of O(n) because it processes each element of the input array once. Therefore, the linear time algorithm is faster in the worst case.

Question 3.

One approach to designing a dynamic programming algorithm for this problem is to use a one-dimensional array dp to store the length of the longest increasing sub-array ending at each index of the input array.

We can initialize the elements of dp to 1, since every element can form a sub-array of length 1 on its own. Then, we can iterate through the input array and for each element, we can compute the length of the longest increasing sub-array ending at that element as follows:

- Set max_length to the current value of the element in the dp array.
- Iterate through the elements of the dp array from the end of the array to the start. For each element, if the element in the input array at the current index is greater than the element at the previous index, update max_length to the maximum of max_length and the value at the previous index in the dp array plus 1.
- Set the value at the current index in the dp array to max_length.

Finally, we can return the maximum value in the dp array as the result.

The worst-case time complexity of this algorithm is O(n^2), since we iterate through the input array and the dp array in two nested loops.

Here is some sample code for the algorithm in Python:

```python
def longest_increasing_subarray(arr):
    n = len(arr)
    dp = [1] * n
    for i in range(1, n):
        max_length = dp[i]
        for j in range(i - 1, -1, -1):
            if arr[i] > arr[j]:
                max_length = max(max_length, dp[j] + 1)
        dp[i] = max_length
    return max(dp)
```