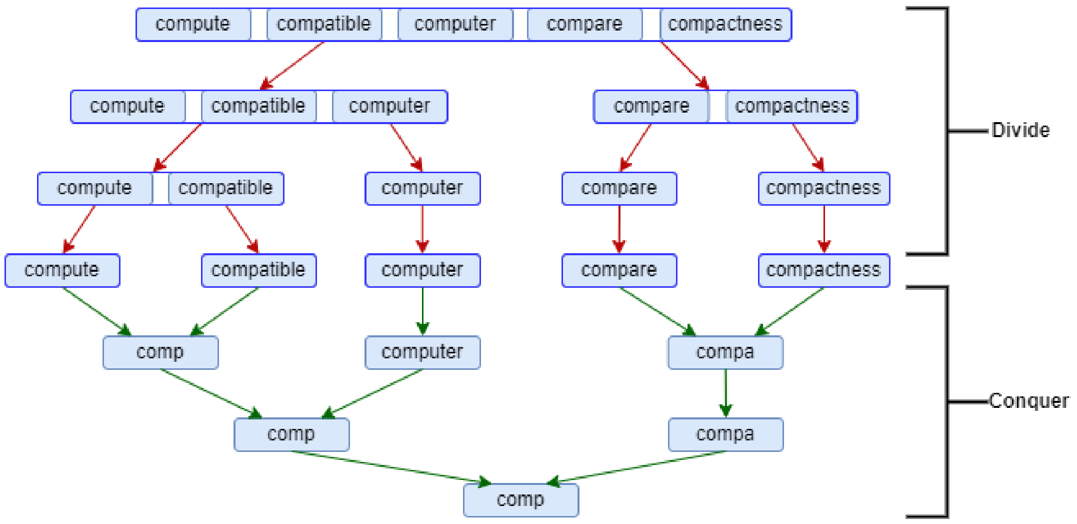
CSE321

Homework #5

Abdullah Çelik - 171044002

1)

The problem looks like mergesort. Mergesort also uses the divide and conquer approach. Here, after dividing to the smallest part (array contains a single string), each recursioncall will return the longest string that matches the string returned from the other call. The diagram below explains the solution better.



Worst Case Analysis:

Divide process will continue the same in all cases. In all scenarios, the string array will be divided into two parts at a time. The worst case is when two strings are most often matched while merging. If the two strings match exactly, the maximum comparision will occur. To express it as a recurrence relation

Divide process:

- $T_1(n) = T_1(n/2) + T_1(n/2)$
- $T_1(n) = 2T_1(n/2)$

Merge process:

- $T_2(n) = n$ (n comparision)

Overall:

$T(n) = 2T(n/2) + n$

The equation i get when i make backward substitution is: $T(n) = n\log n + n$

Conclusion: The algorithm will work in the worst case $\theta(n\log n)$.

2)

(a)

I perform divide operation until our sub lists contain only one item. Then, when our recursion call is returning(conquer step), there are three possibilities for the maximum profit value. These are:

1- Left subarray contains maximum profit
2- Right subarray contains maximum profit
3- Minimum value of the left subarray and the maximum value of the right subarray (cross profit) are the maximum profit

After selecting the maximum profit from these three possibilities, it returns the maximum profit and the index values of the days(minimum day index, maximum day index).

In the case of a subarray containing an item (base case), our profit value will be 0.

Design:

- Divide the list into two parts(divide operation)
- The most recent recursion call (array contains only one item) returns 0(profit value), minimum index(left), maximum index(right). Note: left is equal to right
- Then the recursion calls compares the items coming from the left part and the right part. After setting the minimum and maximum values(conquer operation), they return them.

(b)

Design:

Algorithm searches for maximum profit with two indexes. These are the buy and sell indexes. Buy index is 0 and sell index is 1 as initial. The sell index, on the other hand, moves linearly above the prices(array). Meanwhile, the current profit is calculated. There are two scenarios when the sell index is going end of the array linearly. These:

- Sell price is lower than the buy price. In this case the buy price is updated
- Sell price is higher than the buy price and this is the best profit up to that time. In this case, maxProfit is updated. Buy day and sell day updated accordingly

As a result, I traverse the array once and find the maximum profit and the day of these transactions (buy and sell days).

(c)

In the divide and conquer approach, the list is divided into two parts each time and the minimum and maximum items are returned in the conquer stage. In worst case, maximum profit will be came cross profit step. I will find minimum of the left subarray and maximum of the right subarray. As a result the algorithm will run in $\theta(n\log n)$ time. (The calculation was made using the reccurence relation in the first question)

In solving linear time, i will traverse the list once and i find maximum profit for best case, worst case and average case. Traverse the list means working in θ(n) time. It will take constant time to compare for profit and maximum profit. The algorithm will run in θ(n) linear time. As you can see, as the input size increases, the divide and conquer approach grows faster. O(n) < O(nlogn).

3)

Design:

The algorithm creates an array which items are 1 and size is given input array's size (called longestSequece). This is used for dynamic programming. Thus, without repeating the previous calculation, I can use the previous calculation and save the new longest number of sequence to this array, depending on whether the current item fits the sequence or not.

The elements of the longestSequece array represent: The ith item is the number of sequence that comes up to this item in an increasing way. Thus, if the i+1 th item provides increasing sequence in the given input array, it will be sufficient to perform the following operation.

- longestSequece[i] = longestSequece[i-1] + 1

Thus, we will calculate for the current item without repeating the previous calculation.
As a result of traversing the given input array once, I will fill my longestSequece array and the largest value in this array will give the longest increasing sequence.

Algorithm will traverse given input array only once in best case, worst case and average and will find the longest sequence. So the algorithm will run in θ(n) time. But for this problem, i will not actually be using the full power of dynamic programming. Because here dynamic programming did not provide an advantage in time complexity. Simple solutions with less space complexity are also available in linear time. For example, while i traverse the array, i keep the number of items that meet the given conditions in one place. If this number is greater than the variable named maximum, i update the maximum and as a result, i will work in θ(n) time by traversing the list once. The advantage of this algorithm will be on the side of space complexity. When we use dynamic programming, i will need memory space as given input array's size. But here only one variable is required.

4)

(a)

Design:

I create a two dimensional array and dimensions are input row number and input column number (called maxCost). The value of this array represents the maximum score we can reach while coming to that index. So if I want to find the maximum score for the ith row jth column I will have to check two values. These:

- maxCost[i-1][j]
- maxCost[i][j-1]

Thus, when I add the map[i][j] value with the larger one, it will give the highest score I have collected while reaching the ith row jth column. As you can see, I don't need to recalculate for i-1 th row jth column and ith row j-1 th column. This is the advantage of dynamic programming.

Algorithm fills the first row and first column costs as initial. The reason is that we can only go one way to the positions in the first row and column. It will then use previous calculations for uncalculated positions and will have found the maximum score.

I try to get index 0 a 0 starting from the last row and column to get the path. I can only move left and up to get here. Whichever of these two moves has the highest score, do that move and add its indexes to the list. Reverse this list after the process is finished, I get the path.

(b)

Design:

Algorithm makes the highest point move (moving to the right or down) until it reaches the target. It updates the maximum score variable for each move it makes and adds the indexes to the list called path.

(c)

Usign Brute Force: Brute-force algorithms try to solve a problem by trying all possible solutions and then checking which one is the correct one. These algorithms are guaranteed to be correct because they will always find the correct solution if it exists. However, their time complexity is usually high, often exponential in the worst case. In our problem worst case sceneario, time complexity is $\theta(2^{row+column})$ because of generating all subproblems.

Using Dynamic Programming: Dynamic programming algorithms also try to solve a problem by trying all possible solutions, but they store the solutions to subproblems in a table so that they can be reused (memoized) rather than recomputed. This can greatly improve the time complexity of the algorithm, often reducing it to polynomial time. These algorithms are guaranteed to be correct because they will always find the correct solution if it exists. However, not all problems can be solved using dynamic programming, and the space complexity of the algorithm may be high because of the need to store the solutions to the subproblems. In our problem worst case sceneario, time complexity is $\theta(row*column)$. We can say $\theta(n^2)$ because of traversing map.

Using Greedy: Greedy algorithms make a locally optimal choice at each step and hope that these choices lead to a globally optimal solution. These algorithms are usually easy to implement and have a good time complexity, but they are not always correct because the locally optimal choices may not lead to a globally optimal solution. In our problem worst case sceneario, time complexity is $\theta(row + column)$. We can say $\theta(n)$ because we move last row and last column.

	Time Complexity(worst case)	*Correctness(if solution exist)*
Brute Force	$\Theta(2^{row+column})$	✓
Dynamic Programming	$\Theta(row * column)$	✓
Greedy	$\Theta(row + column)$	✗ (usually)