CSE 321 HW5

Burak Kocausta

1901042605

Note: Explanation of the test menu is end of the report. After built-in results executed, menu will show up.

Q1) D&C Longest Common Substring

Explanation:

I designed a divide and conquer algorithm for the purpose of finding longest common substring in a string list which is quite similar to the binary search. But difference of this algorithm comes with, this is not a decrease and conquer algorithm. Reason for that is, in this algorithm, there is no elimination of problems. It divides the problem, but none of the subproblems are eliminated. Problem is solved with merging the subproblems. Every string on the string list is visited during this operation.

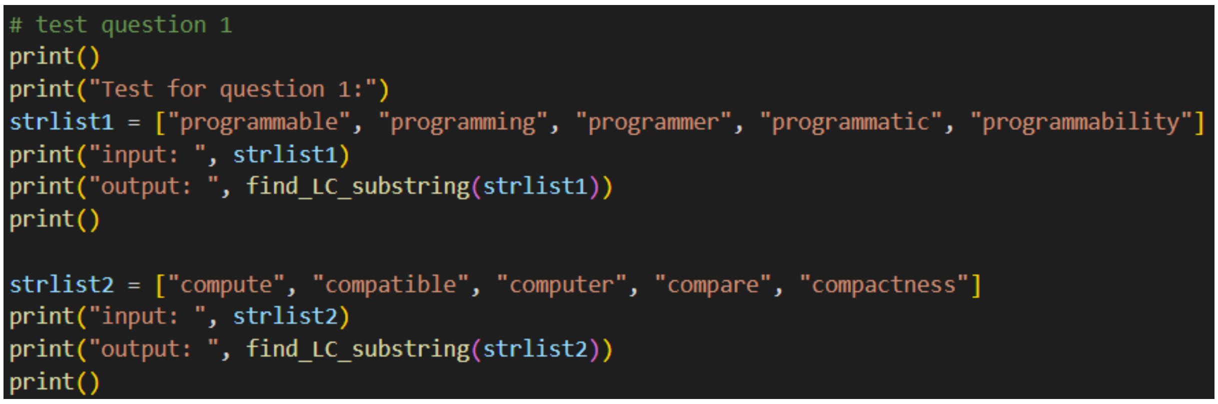
Idea is dividing the array in each step, get the common substring of left, and right then find a new common substring between this left and right, and return it. Base case is the condition when there is one element, this means return that element. Additionally, this algorithm requires extra function which is finding common substring with 2 strings. It simply checks all strings from beginning and returns the substring, but this comes with a cost to the time complexity. It is explained in analysis part. I also use sub procedures for helper functions. Main function only takes one string_list element, calls the sub procedures, and returns the result.

Analysis:

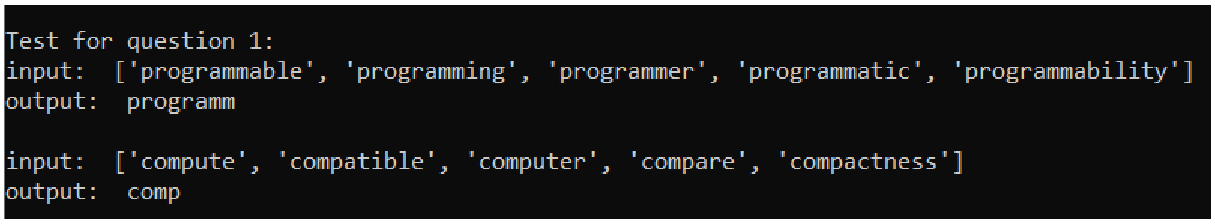
Algorithm traverses all of the strings in all cases, so it is impossible to algorithm be lesser than linear. In each merging part, algorithm takes 2 strings, and finds common substring between these 2 strings. This operation depends on the size of the common substring. Because loop is executed size of the substring times. This happens on each step. If there is no common substring, which is best case time complexity will be Θ(n). If the size of the substring is m, and n times this is executed, then it is Θ(m*n).

T(n) = O(m*n)

Test for question 1:



```python
# test question 1
print()
print("Test for question 1:")
strlist1 = ["programmable", "programming", "programmer", "programmatic", "programmability"]
print("input: ", strlist1)
print("output: ", find_LC_substring(strlist1))
print()

strlist2 = ["compute", "compatible", "computer", "compare", "compactness"]
print("input: ", strlist2)
print("output: ", find_LC_substring(strlist2))
print()
```



```
Test for question 1:
input:  ['programmable', 'programming', 'programmer', 'programmatic', 'programmability']
output:  programm

input:  ['compute', 'compatible', 'computer', 'compare', 'compactness']
output:  comp
```

Q2-a) D&C Maximize Profit

Explanation:

I designed a divide and conquer algorithm which is similar to the question 1. For every step left and right sub arrays returns 4 indices. Then with these 8 different values, algorithm merges the new result in constant time. Every sub problem produces a result, this results are the 4 indices which are the indices of maximum value, minimum value, buy value, sell value. At the end result will be buy, and sell indices. Normally, it can be thought of only buy and sell indices are worth to look, but it becomes important when merging the subproblems. Buying must always come before selling, this can lead to problem when thinking only 2 property is enough. With merging each subproblem, main problem produces the result.

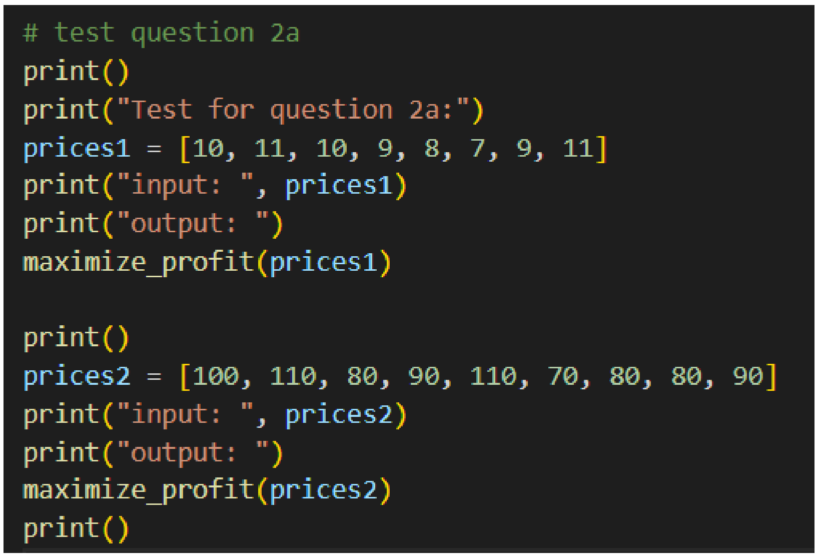
In the implementation, I used recursive function, and merging functions as a sub procedure under the main function which takes only the list of prices. Recursive function is quite similar to the binary search, but it is not decrease and conquer again like question 1, there is no elimination of subproblems, it divides the problem in every step using middle index. After getting left, and right results, I send the results to merge function, this function simply decides, what is the minimum, maximum, buy, and sell with using left and right results. Finding minimum and maximum is easy, but deciding buy and sell is a little bit tricky. While deciding that, the information of where minimum and maximum come becomes important. If both comes from left, and buy, and sell values can be left's minimum, and right's maximum, or left's buy, and left's sell. Reason of this is lefts minimum could be come after the lefts maximum, so this condition must be checked. It similar for the right-right case. For the left-right case, different profits must be calculated, then result must be determined according to these 4 profits. Last option is minimum came from left, and maximum came from right, that is easy because in this case buy is minimum of left, and sell is maximum of right. Merging operation can be a little bit confusing, but it is constant time operation, it is executed on every conquering step so being constant time is important.

Analysis:

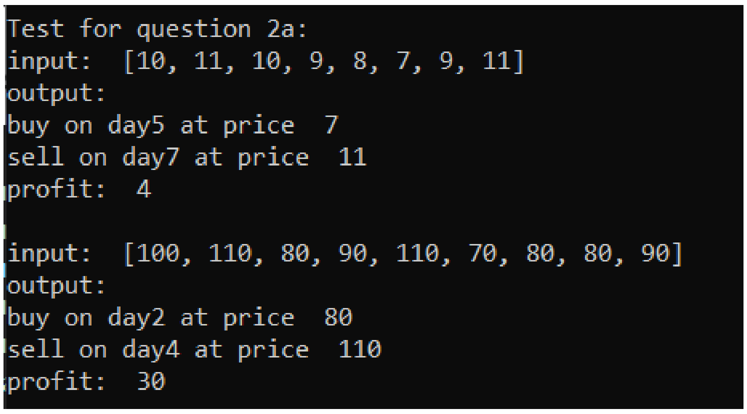
Algorithm simply visits all of the elements, then merges the results. Cost of visiting all of the elements is of course ϴ(n). In each step, merge operation is done, but merge operation is a constant time operation, so it must be linear time algorithm. Every operation in each call is constant time operation, so relation is like T(n) = 2T(n/2) + O(1), when this equation is solved with master theorem, it can be seen that algorithm is linear. Therefore, time complexity of that algorithm is ϴ(n).

T(n) = O(n)

Test for Question 2a:



```
# test question 2a
print()
print("Test for question 2a:")
prices1 = [10, 11, 10, 9, 8, 7, 9, 11]
print("input: ", prices1)
print("output: ")
maximize_profit(prices1)

print()
prices2 = [100, 110, 80, 90, 110, 70, 80, 80, 90]
print("input: ", prices2)
print("output: ")
maximize_profit(prices2)
print()
```



```
Test for question 2a:
input:  [10, 11, 10, 9, 8, 7, 9, 11]
output:
buy on day5 at price  7
sell on day7 at price  11
profit:  4

input:  [100, 110, 80, 90, 110, 70, 80, 80, 90]
output:
buy on day2 at price  80
sell on day4 at price  110
profit:  30
```

Q2-b) NON-D&C Maximize Profit

Explanation:

Question 2-b is a little bit ambiguous, because it can be interpreted like, design an algorithm which is not based on divide and conquer, and it is not linear. Other option is designing an algorithm which is not based on d&c, but algorithm is linear. Because of this ambiguity, I implemented both quadratic, and linear algorithms other than divide and conquer.

To implement the algorithm in linear time, I used dynamic programming approach to solve that problem to make it linear again, because with naïve approach, there are calculations of unnecessary profits, they can be eliminated using dynamic programming. Previous profit can be stored, and minimum value can be determined each step using the last minimum and current index. And every step, new profit can be calculated if new profit is greater than old profit, then the old values are updated. It will be linear because array is traversed only once.

Non-linear implementation is direct approach, if a new minimum (means buying day) is found, searches for the sell from current index of the array in every step. With this nested for loops, it is guaranteed that buy comes before sell, and the maximum profit is calculated.

Analysis:

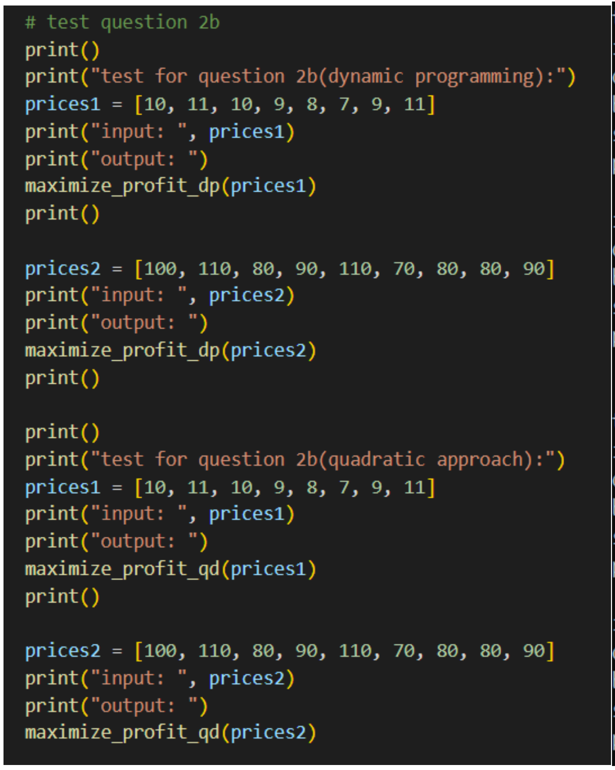
For the dynamic programming part, Algorithm simply traverses the whole array to determine minimum and maximum. There are no additional operations that are not constant time. So it is ϴ(n).

For the quadratic approach, It is an algorithm which has quadratic time complexity, because in found of minimum value, it iterates till the end of the array from that point, in the worst case it will be (n-1) + (n-2) + … + 1, which is quadratic. Best case the case that all values are same, or array is on decreasing order. In that case inner loop will never be executed, and algorithm takes linear time.

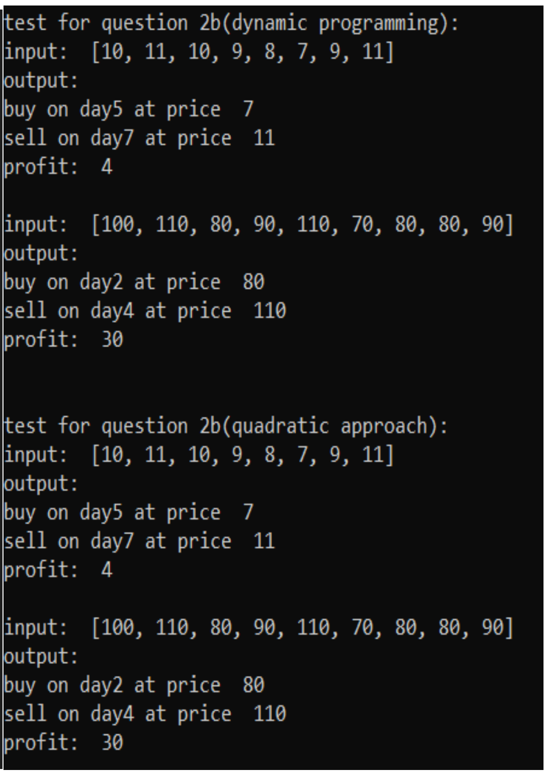
T(n) = O(n^2) (quadratic approach)

T(n) = ϴ(n) (dynamic programming)

Test for question 2b:



```
# test question 2b
print()
print("test for question 2b(dynamic programming):")
prices1 = [10, 11, 10, 9, 8, 7, 9, 11]
print("input: ", prices1)
print("output: ")
maximize_profit_dp(prices1)
print()

prices2 = [100, 110, 80, 90, 110, 70, 80, 80, 90]
print("input: ", prices2)
print("output: ")
maximize_profit_dp(prices2)
print()

print()
print("test for question 2b(quadratic approach):")
prices1 = [10, 11, 10, 9, 8, 7, 9, 11]
print("input: ", prices1)
print("output: ")
maximize_profit_qd(prices1)
print()

prices2 = [100, 110, 80, 90, 110, 70, 80, 80, 90]
print("input: ", prices2)
print("output: ")
maximize_profit_qd(prices2)
```



```
test for question 2b(dynamic programming):
input:  [10, 11, 10, 9, 8, 7, 9, 11]
output:
buy on day5 at price  7
sell on day7 at price  11
profit:  4

input:  [100, 110, 80, 90, 110, 70, 80, 80, 90]
output:
buy on day2 at price  80
sell on day4 at price  110
profit:  30


test for question 2b(quadratic approach):
input:  [10, 11, 10, 9, 8, 7, 9, 11]
output:
buy on day5 at price  7
sell on day7 at price  11
profit:  4

input:  [100, 110, 80, 90, 110, 70, 80, 80, 90]
output:
buy on day2 at price  80
sell on day4 at price  110
profit:  30
```

Q2-c) Comparison of 2a and 2b

Explanation:

Divide and conquer, and dynamic programming solutions for that problem is not dependent on any conditions, because in each problem, all elements of the array are visited. For the divide and conquer, array is divided (in place) without condition, then they are also merged without condition. For every case, all operations are done. Therefore, best, and worst case are the same for these two. On the other hand, quadratic approach can be linear when array is reversely sorted or all elements are equal, as I explained in the analysis part of 2b.

As a result:
Worst/Best Case of divide and conquer solution = O(n)

Worst/Best Case of dynamic programming solution = O(n)

Worst Case of quadratic solution = O(n^2), Best Case = O(n)

Q3) Longest Increasing Sub Array -Dynamic Programming

Explanation:

For that problem, firstly I am confused because problem is too simple to need using dynamic programming. Because, if they are consecutive, then simply traversing the array must be enough. In direct approach algorithm already not have an overlapping problem. If there is not an overlapping problem, it is impossible to compute sub problems that are computed before, this leads to an idea that dynamic programming for that problem is not needed. If there is no condition like sub arrays must not be consecutive in the main array, then dynamic program most likely be needed. But it is clearly indicated that all elements of the sub array are consecutive in the main array. Also, second example shows that the consecutiveness is a must, because if it wasn't, result would be [1,2,3,4,5]. So, I did not do the same thing as in question 2, and I implemented only one solution which finds the longest increasing sub array that has consecutive elements on the main array.

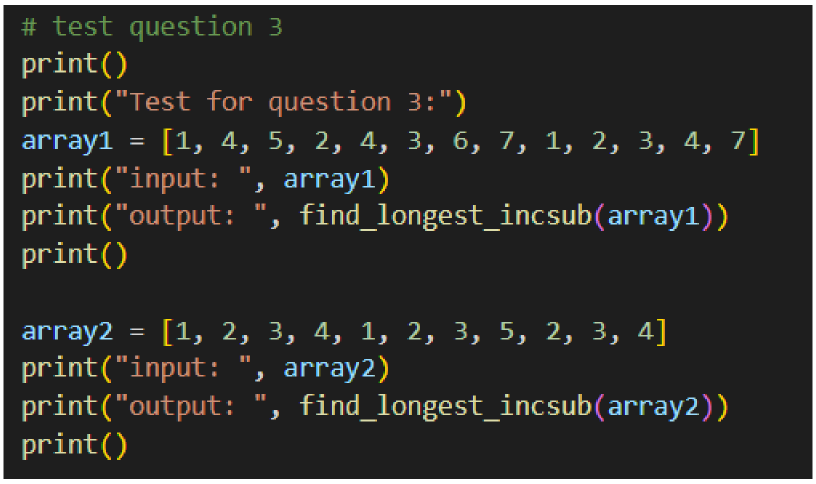
I held a dynamic table and filled all values with 0. In the algorithm, traversed the array from start to end, and in each step, I checked the increasing. If there is an increase, I incremented the counter and updated the dynamic table. This dynamic table elements are 0 or 1. They are indicating for each index, number of the increasing sub array of that size. If increasing comes to an end, then algorithm sets the counter to the 0. Then starts to increase it again. Therefore, it uses this memoization table, to check the size of the longest increasing sub array. Then returns the last 1s index.

Analysis:

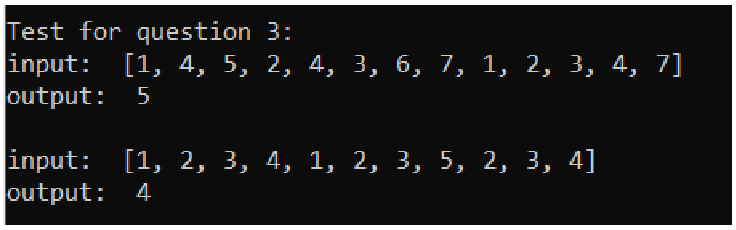
Time complexity of the algorithm is linear because it traverses the whole array only once. Table has the same size with the array, therefore it does not affect the linearity of the algorithm. There is no special condition to lead best or worst case. So theta notation can be used.

T(n) = ϴ(n)

Test for question 3:



```
# test question 3
print()
print("Test for question 3:")
array1 = [1, 4, 5, 2, 4, 3, 6, 7, 1, 2, 3, 4, 7]
print("input: ", array1)
print("output: ", find_longest_incsub(array1))
print()

array2 = [1, 2, 3, 4, 1, 2, 3, 5, 2, 3, 4]
print("input: ", array2)
print("output: ", find_longest_incsub(array2))
print()
```



```
Test for question 3:
input:  [1, 4, 5, 2, 4, 3, 6, 7, 1, 2, 3, 4, 7]
output:  5

input:  [1, 2, 3, 4, 1, 2, 3, 5, 2, 3, 4]
output:  4
```

Q4-a) Route Game Algorithm with Dynamic Programming

Explanation:

I designed the algorithm initially thinking how to eliminate overlapping operations on the brute force approach for the same problem. For that approach, it searches all possible routes, then finds the greatest one. But with doing that, same moves are done again and again. As a solution, I thought a dynamic table(n*m). This has the same size with n*m 2d map. While filling that table, if every part of that map is filled with optimal points using the points which are come from the neighbor nodes(upper, and left). Then the last element of that map will be the result. Because every step executed only once unlike brute force approach, and the optimal solution is carried to the last element.

This table starts with initial element, for every element that are on the edges; top elements gets points from left, leftmost elements get points from top. Idea is carrying the maximum point to every node of the matrix. If every node of the matrix has the maximum point to that place, it means last node will have the maximum point. For the elements that are not on the edge, decision should be made. Greatest neighbors (left or up), points must be added to the current node's point.

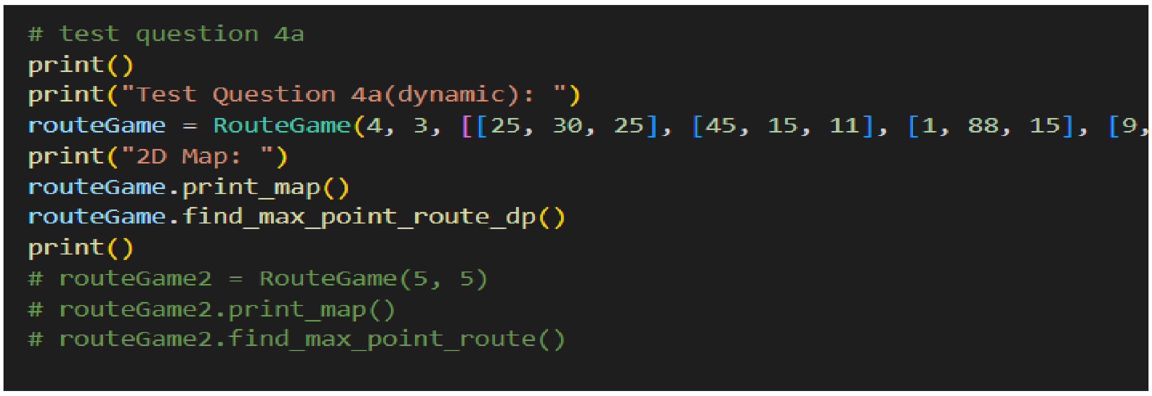
To implement that, I used nested loops to traverse the table. It starts from leftmost top, and goes right in every inner loop, at the end right bottom will be reached. Initially, node is checked for if it is edge or not, if it is edge then it means no need to decide. Last condition is getting the maximum point from 2 neighbor. And that's it for the algorithm. I also printed the dynamic table to indicate, how it works.

Analysis:

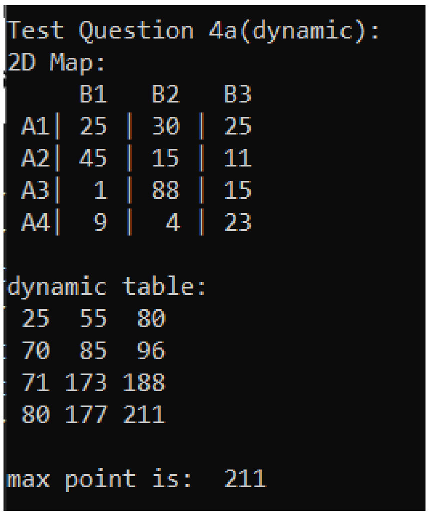
Algorithm only traverses the table, it has no condition to create worst or best case. Table has the size of m*n. So, time complexity of the algorithm is ϴ(m*n)

T(n) = ϴ(m*n)

Test for question 4a:



```
# test question 4a
print()
print("Test Question 4a(dynamic): ")
routeGame = RouteGame(4, 3, [[25, 30, 25], [45, 15, 11], [1, 88, 15], [9,
print("2D Map: ")
routeGame.print_map()
routeGame.find_max_point_route_dp()
print()
# routeGame2 = RouteGame(5, 5)
# routeGame2.print_map()
# routeGame2.find_max_point_route()
```



```
Test Question 4a(dynamic):
2D Map:
     B1   B2   B3
 A1| 25 | 30 | 25
 A2| 45 | 15 | 11
 A3|  1 | 88 | 15
 A4|  9 |  4 | 23

dynamic table:
 25  55  80
 70  85  96
 71 173 188
 80 177 211

max point is:  211
```

Q4-b) Route Game Algorithm with Greedy Algorithm

Explanation:

For the greedy algorithm, I designed the algorithm in a way that it makes every local decision according to the maximum point where it can get. It is like brute force, but only and quite important difference is, it tries only one solution. In brute force approach, there is no smart decision, but here it chooses the option where it can get the maximum point locally. This results with quite fast algorithm, but problem is **optimal solution is not guaranteed.**

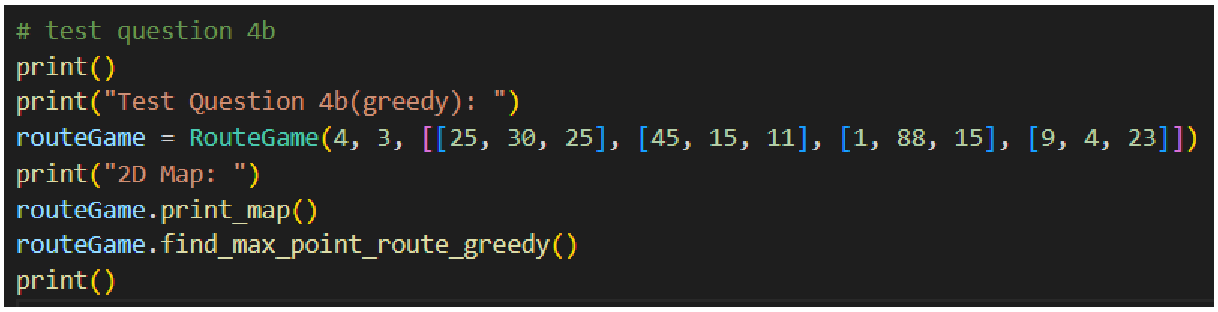
For the implementation, I used recursion, and each step of the recursion it initially checks the base case which gives the result. After that checks if bottom or rightmost is reached, because there is no decision for that conditions, It directly goes down or right. For the decision part, it checks route gives more point on that case, and chooses the greatest one. With this way route is finished.

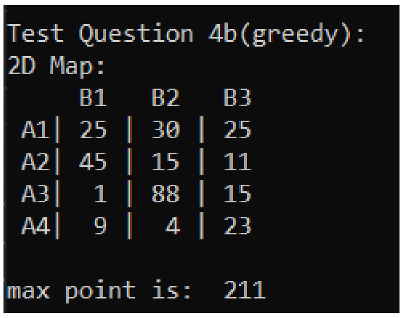
Analysis:

In this algorithm, all nodes are not visited, only the route nodes are visited. Therefore, it is even faster than dynamic programming solution, because it is linear. Number of elements in a route is the time complexity. For every matrix there is a certain amount of number in a route. It can be said that this number is n+m-1, then time complexity is ϴ(n+m).

T(n) = ϴ(n+m) – (linear)

Test for question 4b:


```
# test question 4b
print()
print("Test Question 4b(greedy): ")
routeGame = RouteGame(4, 3, [[25, 30, 25], [45, 15, 11], [1, 88, 15], [9, 4, 23]])
print("2D Map: ")
routeGame.print_map()
routeGame.find_max_point_route_greedy()
print()
```


```
Test Question 4b(greedy):
2D Map:
     B1   B2   B3
 A1| 25 | 30 | 25
 A2| 45 | 15 | 11
 A3|  1 | 88 | 15
 A4|  9 |  4 | 23

max point is:  211
```

Q4-c) Comparison of BF, DP, Greedy Route Game Algorithms

Dynamic programming algorithm is far more advantaged, than brute force algorithm, because it is polynomial, and both of them are guaranteed to get optimal solution. Brute force algorithm has no advantage over dynamic programming solution. The only advantage that can be think of is brute force algorithm does not uses extra space, but dp algorithm uses extra n*m array for the dynamic table. Even with that dynamic programming solution is much better than the brute force. Reason for that speed difference is explained, but simply brute force checks for all solutions then finds the optimal one. Dynamic programming uses the results of the subproblems without overlapping them. This reason creates the difference between these two.

To compare brute force solution and greedy algorithm; in the sense of speed, greedy algorithm is significantly faster than brute force. But in this time advantage is not certain like dp and bf comparison, because greedy algorithm has an important disadvantage which means that it can give non-optimal answers for some problems. For the above test cases, it gives optimal solution but some cases it might not be happen because it only checks for locally advantage. This tradeoff gives that approach a significant speed. One of them is exponential and other one is linear.

Last comparison is between dynamic programming, and greedy algorithm. For this problem, dynamic programming is more advantaged in my opinion because greedy algorithm does not give optimal solution for every case. It is faster, but both algorithms are polynomial time algorithms. Greedy algorithm is faster than dynamic programming solution, but dp solution guarantees the optimal result.

	Best	Worst	Average
Brute Force	O(2^(n+m))	O(2^(n+m))	O(2^(n+m))
Dynamic Programming	O(n*m)	O(n*m)	O(n*m)
Greedy	O(n+m)	O(n+m)	O(n+m)

As a result:

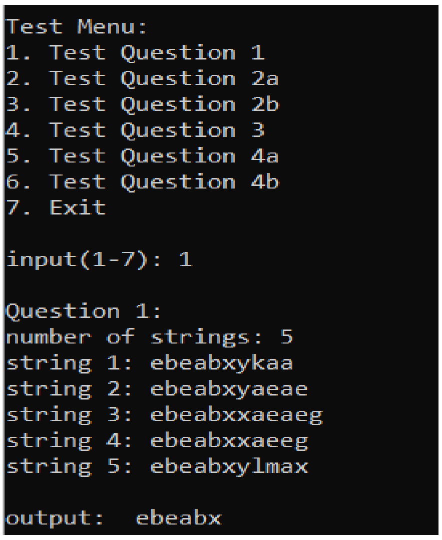
Greedy solution is the fastest, but optimal solution is not guaranteed.

Dynamic Programming solution is much more faster than brute force approach, and optimal solution is guaranteed.

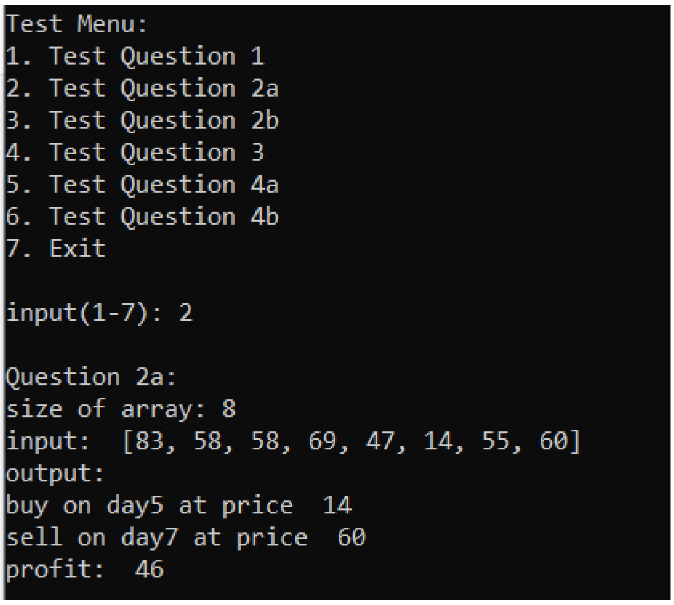
Brute Force solution guarantees the correct result, but it is too slow.

In my opinion, dynamic programming solution is the best approach for that problem to get correct result, and fast algorithm.

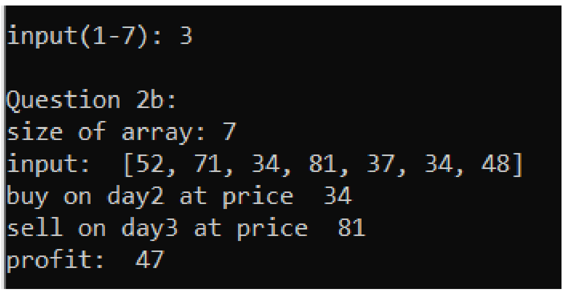
Test Menu for Questions 1-2-3-4:

Question 1: 

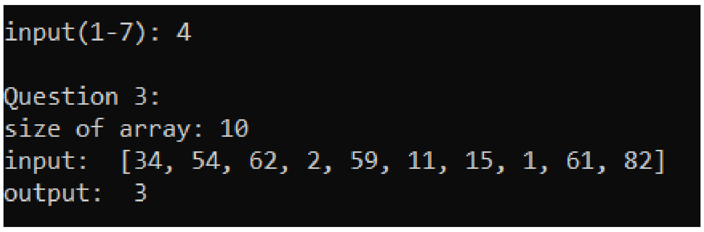
```
Test Menu:
1. Test Question 1
2. Test Question 2a
3. Test Question 2b
4. Test Question 3
5. Test Question 4a
6. Test Question 4b
7. Exit

input(1-7): 1

Question 1:
number of strings: 5
string 1: ebeabxykaa
string 2: ebeabxyaeae
string 3: ebeabxxaeaeg
string 4: ebeabxxaeeg
string 5: ebeabxylmax

output:  ebeabx
```

Question 2a:  (randomly generated)

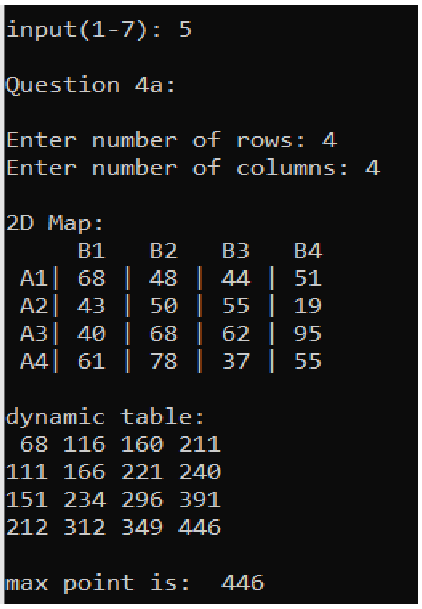
```
Test Menu:
1. Test Question 1
2. Test Question 2a
3. Test Question 2b
4. Test Question 3
5. Test Question 4a
6. Test Question 4b
7. Exit

input(1-7): 2

Question 2a:
size of array: 8
input:  [83, 58, 58, 69, 47, 14, 55, 60]
output:
buy on day5 at price  14
sell on day7 at price  60
profit:  46
```

Question 2b:  (randomly generated)

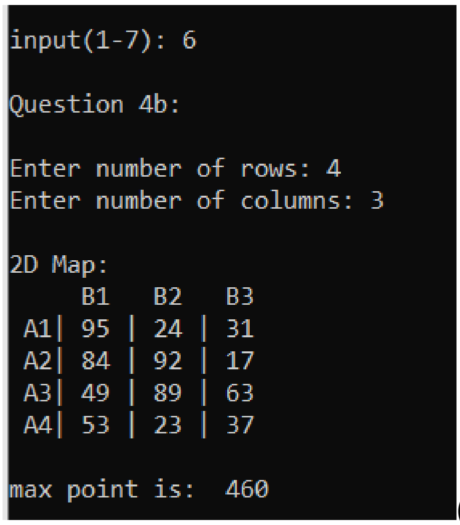
```
input(1-7): 3

Question 2b:
size of array: 7
input:  [52, 71, 34, 81, 37, 34, 48]
buy on day2 at price  34
sell on day3 at price  81
profit:  47
```

Question 3:  (randomly generated)

```
input(1-7): 4

Question 3:
size of array: 10
input:  [34, 54, 62, 2, 59, 11, 15, 1, 61, 82]
output:  3
```

Question 4a:  (dp solution, values are random)

```
input(1-7): 5

Question 4a:

Enter number of rows: 4
Enter number of columns: 4

2D Map:
     B1   B2   B3   B4
 A1| 68 | 48 | 44 | 51
 A2| 43 | 50 | 55 | 19
 A3| 40 | 68 | 62 | 95
 A4| 61 | 78 | 37 | 55

dynamic table:
 68 116 160 211
111 166 221 240
151 234 296 391
212 312 349 446

max point is:  446
```

Question 4b:  (greedy solution, values are random)

```
input(1-7): 6

Question 4b:

Enter number of rows: 4
Enter number of columns: 3

2D Map:
     B1   B2   B3
 A1| 95 | 24 | 31
 A2| 84 | 92 | 17
 A3| 49 | 89 | 63
 A4| 53 | 23 | 37

max point is:  460
```

Terminate menu with 7.