CSE 321 INTRODUCTION TO ALGORITHM DESIGN HW5

WORST CASE ANAYLSIS

- Finding Longest Common String

```python
def longestCommonString(words):
    if(len(words) == 1):
        return words[0]

    def helper(arrays):
        n = len(arrays)

        if (n == 1):
            return arrays[0]
        mid = int(n / 2)
        left = helper(arrays[:mid])
        right = helper(arrays[mid:])

        commonString=""

        for i in range(min(len(left), len(right))):
            if left[i] == right[i]:
                commonString += left[i]

            else:
                break
        return commonString

    return helper(words)
```

This helper function is called for comparing two strings to find common substrings between them.

Helper function is called n/2 times where N is length of the input list.

Comparing to two string takes O(M) time that M= length of the string that has minimum length.

Len(string) function takes = Θ(n).

Min(x,y) takes = Θ(1).

So worst case complexiys is = O(N*M);

- Finding Maximum Profit

a- Divide and Conquer Approach

```
def findMaxProfitV1(array):
    maxProfit = -sys.maxsize -1
    minIndex = 0
    maxIndex = 0
    if(len(array) == 1):

        return 0,print(0)


    def helper(array):
        n = len(array)
        if((n == 1) | (n == 0)):
            return 0
        mid = int(n /2)
        helper(array[:mid])
        minPrice = min(array[:mid])
        nonlocal minIndex,maxIndex
        minIndex = indexOf(minPrice,array)
        helper(array[mid:])
        maxPrice = max(array[mid:])
        maxIndex = indexOf(maxPrice,array)
        nonlocal maxProfit
        profit = maxPrice - minPrice
        if(maxProfit < profit):
            maxProfit = profit
        return maxProfit
    helper(array)
    print("Output: Buy on Day{0} for {1} and sell on Day{2} for {3} profit = {4}"
    .format(minIndex,array[minIndex],maxIndex,array[maxIndex],maxProfit),end='\n')
```

Above code finds minimum buying value of left half of the array , and maximum selling value of the right half of array. Then find maximum profit by subtracting selling and buying value.

Helper Function is called n/2 times where n = length of possible sell , buy days.

Finding minimum takes O(n).

Finding maximum takes O(n).

Indexof function takes O(n).

So total **worst case complexity is O(nlogn).**

b-) Naïve Approach (Linear Time)

```
def findMaxProfitV2(array):

    n = len(array)
    if(n == 1):
        return 0 ,print(0)
    maxProfit = -sys.maxsize -1
    minIndex = 0
    maxIndex = 0

    for i in range (n):
        buy = array[i]
        for j in range(i+1 ,n):
            sell = array[j]
            profit = int(sell - buy)
            if(maxProfit <  profit):
                minIndex =i
                maxIndex = j
                maxProfit = profit

    print("Output: Buy on Day{0} for {1} and sell on Day{2} for {3} profit = {4}"
    .format(minIndex,array[minIndex],maxIndex,array[maxIndex],maxProfit),end='\n')
```

First for loop works for n times where n is length of the array.

Second for loop work for i times.

Index of works takes = O(N) times.

$equation = \sum_{i=0}^{n-1} i * n$. so it takes W(n) = (n*n+1)/2 = O(N^2).

- Length of the Longest Subarray

```
def longest_increasing_sub_array(array):
    n = len(array)
    end = n
    result = [1]*n
    for i in range (n-1 , -1 , -1):
        for j in range (i+1,end):
            if(array[i] < array[j]):
                result[i] = max(result[i], 1+result[j])
            else :

                end = i+1
                break
    return max(result)
```

This code works as ;

First we fill the result array with 1. We think result array as cache.

Then the iteration is begun in array in reverse order.

If the next input is less than previous input ,

Then the rule is = result[i] = max(result[i] , 1+result[j]).

Then return the maximum value of result array.

Filling the array with 1 takes = Θ(n).

The first loop work for n times where n is equal to length of the array.

First loop begins from n to 0 as decreasing.

Second loop work for end – i times. And i is decreasing 1 for each loop iteration.

So total wors case complexity if the array is sorted in decreasing order then T(n) = (n*(n+1))/2 = O(N^2).

- MAXIMUM PATH FINDING

V1

```python
def highestScore(matrix):
    rowNumber = len(matrix)
    colNumber = len(matrix[0])
    tempMatrix = [[0]* (colNumber+1) for i in range(rowNumber+1)]
    for i in range (1,rowNumber+1):
        for j in range(1,colNumber+1):
            tempMatrix[i][j] = matrix[i-1][j-1]

    for i in range (1, rowNumber+1):
        for j in range (1, colNumber+1):

            if(tempMatrix[i-1][j] > tempMatrix[i][j-1]):
                max = tempMatrix[i-1][j]
            else:
                max =tempMatrix[i][j-1]
            tempMatrix[i][j] = tempMatrix[i][j] + max
    return tempMatrix[rowNumber][colNumber]
```

We are using tempMatrix as cache. It stores values need to reach that position.

So firstly fill the tempMatrix with 0’s. İt’s length is one more than actual matrix.

Then fill the tempMatrix with actual matrix.

The current position depends on the upper and left adjacent position.

If the upper adjacent position gives much score then the actual position became upper adjacent position + it’s own score.

It is same as left adjacent position situation.

Then we return desired position score.

Filling array with 0’s and actual array takes O(N).

First for loop works for N times.

Second for loop also works for N times if the matrix is square.

So total W(N) = O(N*N).

V2

```python
# this version is not gives optimal solution for all the matrix
def highestScoreV2(matrix):
    rowLen = len(matrix)
    colLen = len(matrix[0])

    i = 0
    j = 0
    res = matrix[0][0]
    while((i != rowLen-1) & (j != colLen-1)):
        if(matrix[i+1][j] > matrix[i][j+1]):
            res += matrix[i+1][j]
            i += 1
        else :
            res += matrix[i][j+1]
            j += 1
    return res + matrix[rowLen-1][colLen-1]
```

This greed method does not give true solution for all the set of array.

This method finds select maximum scored position to reach desired position.

The worst case is O(N^2).

2-C

Comparing Worst Case

Divide and Conquer method takes O(nlogn) since it compares left and right half of the array to find max profit , whereas Naïve method (Linear) compares all the element with other elements. So it takes O(N^2).

4-C

Comparing Correctness:

Greedy method does not give correct solution for all possible input sets. Since it looks for more scored position to select new position. Whereas the most scored position may be located in unreached position. Thus the result will became incorrect.

On the other hand, Dynamic Programming Technique gives always true result. Since it stores all the scores that need for the positions. The next position depends on upper and left adjacent position and itself.

Comparing Worst Case :

They are equal . Both are O(N^2).