CIS 675 Homework 4

Design and Analysis of Algorithms

Saket Kiran Thombre

SU ID: 899913802

NetID: sthombre

Email: sthombre@syr.edu

Collaborators for Q1, Q2:

1. Saurav Shashank Narvekar

   NetID: sanarvek
2. Chandan Pothumarthi

   NetID: cpothuma

Problem 1 Stock ticker prices (Point 5)

Given a list of stock ticker prices p1, p2, . . . , pn, devise an O(n 2) algorithm that finds the longest (not necessarily consecutive) streak of prices that increase or stays the same. For example, given the prices 2, 5, 2, 6, 3, 3, 6, 7, 4, 5, there is the streak 2, 5, 6, 6, 7 of prices that increase or stay the same, but an even longer streak is 2, 2, 3, 3, 4, 5. Similarly, given the prices 1, 9, 2, 4, 3, 5, 8, 7, 7, 9, there can be multiple longest streaks: 1, 2, 3, 5, 7, 7, 9 or 1, 2, 4, 5, 7, 7, 9. In your solution, finding one longest streak is enough. Extra Credit Challenge: by using both dynamic programming and binary search, you can solve this problem in O(n log n) time. (The deadline of this extra-credit problem is same as assignment/homework 4. This extra credit will be counted toward your total assignment points and will credited more points compared to the extra-credit problem set problems.)

Solution:

I have developed an algorithm using dynamic programming that will yield the optimal answer, which is the longest list by utilizing them to make judgments and adding them to the current subproblem.

All the algorithm lines are numbered.

We create a function to construct and print the longest streak of prices that will increase of remain same.

```
1. def constructPrintLIS(arr: list, n: int)
```

We now create an array longestSequence to store the longest increasing number. The range for this array is till i.

```
2.     longestSequence = [[] for i in range(n)]
```

The [0]th element of both arrays longestSequence and tempArray are equal. Now we append these two arrays.

```
3.     longestSequence[0].append(tempArray[0])
```

Now we start from the first index postion and travel till n.

```
4.     for i in range(1, n)
```

We now traverse for all j less than i.

```
5.         for j in range(i)
```

We now copy the "j"th longest sequence into "i" by using the following condition. longestSequence must be equal to the max element in j and tempArray i only when j < i and tempArray[j] < tempArray[i].

```
6.             if tempArray[i] > tempArray[j] and (len(longestSequence[i]) <
len(longestSequence[j]) + 1):
```

```
7.                  longestSequence[i] = longestSequence[j].copy()
```

```
Appending tempArray[i] to longest sequence as it ends with the "i"th element of
tempArray.
8.          longestSequence[i].append(tempArray[i])
```

```
longestSequence array will store all the values from tempArray[0....i] since they
end in tempArray[i]
9.      maxElement = longestSequence[0]
```

```
Now we traverse the array longestSequence and compare if the length of m is
greater than any element. If yes we add it to maxElement.
10.     for m in longestSequence
11.          if len(m) > len(maxElement)
12.              maxElement = m
```

```
Now we print maxElement
13.      print(maxElement)
```

We can say that the time complexity for this algorithm is $O(n^3)$.

We have 2 nested for loops and one for loop for copying and traversing elements from longestSequence to maxElement.

Extra-Credit Solution:

```
All the algorithm lines are numbered.
```

```
1. def LongestStreakPrice(tempArray, n)
```

```
We add boundary cases for when n is 0.
```

```
initiliazing array tailElement with 0
2.      tailElement =[0 for i in range(n + 1)]
```

```
initilalizing prevElement with -1
3.      prevElement =[-1 for i in range(n + 1)]
```

```
4.      tempLen = 1
```

Traversing the for loop from 1 to n with the condition to find the smallest value element.

```
5.     for i in range(1, n)

6.         if (tempArray[i] < tempArray[tailElement[0]])
```

We now get the smallest value into tailElement

```
7.             tailElement[0] = i
```

Else we check if tempArray elements are greater than tailElement length -1

```
8.         elif (tempArray[i] > tempArray[tailElement[tempLen-1]]):
```

We now extend the largest number sequence from tempArray

```
9.             prevElement[i] = tailElement[tempLen-1]
10.            tailElement[tempLen] = i
11.            tempLen += 1

12.         else
```

tempArraywill now check if it has a potential candidate for subsequent replace operation. If yes, it will replace it with the the top most ID.

```
13.             tempPosition = topElementID(tempArray, tailElement, -1,
14.                                         tempLen-1, tempArray[i])

15.             prevElement[i] = tailElement[tempPosition-1]
16.             tailElement[tempPosition] = i

17. print longest sequence index number
18.     i = tailElement[tempLen-1]
19.     while(i less than or equal to 0)
20.         print(tempArray[i])
21.         i = prevElement[i]

22.     return tempLen
```

The algorithm will have the time complexity of O(Nlog(n)) since we are using one for loop and a recursive call to traverse through the array.

Problem 2 Hershey's bar (Point 5)

You are given an n × m Hershey's bar. Your goal is to devise an algorithm A that takes as input (n, m) and returns the minimal number of cuts needed to divide the bar into perfect squares of either 1x1, 2x2, 3x3, . . ., jxj. With each cut, you can split the bar either horizontally or vertically. The first cut will split the Hershey's bar into two sub-bars; after that each cut splits one remaining sub-bar into two. For example, A(2, 3) = 2 because 2x3 → (2x2, 1x2) → (2x2, 1x1, 1x1).

1. Hint: Notice that no matter the rectangle input n × m, it is always possible to make a perfect square in the first cut. But this strategy will fail. It is possible to find an example input size for which the strategy of picking the cut which creates the largest square leads to extra cuts in total.
2. You cannot have fraction, after each cut, the generated rectangles will have integer dimensions (width and height are integer values).
3. Devise a DP algorithm which determines the minimal number of cuts.
4. If your solution has run-time θ(n 3), your solution is acceptable. Discuss the runtime of your algorithm, and show that run-time is equal or below θ(n 3).

Solution:

All the algorithm lines are numbered.

```
1. def hersheysCut(m,n)
```

If the given Hershey's chocolate bar is already a square, then we will return the same answer. For This we are comparing "n" and "m" with each other.

```
2.      if n == m
3.          return 1
```

We will now cut the Hershey's bar vertically into 2 smaller bars and recursively repeat the process till we get a minimum value of 1. We are doing int(n/2)+1 because when we cross the half mark, we will start repeating the same values.

```
4.      hersheyVertical = [n * m]
5.      for i in range (1, int(n/2)+1)
```

Now we will calculate minimum parts for the Hershey's bar by cutting the bar vertically till we reach the minimum value 1. This will further get appended to hersheyVertical.

```
6.          hersheyVertical.append(hersheysCut(m,i) + hersheysCut(m,n-i))
```

Now we cut the Hershey's bar horizontally into 2 smaller bars and recursively repeat the process till we get a minimum value of 1. We are doing int(n/2)+1 because when we cross the half mark, we will start repeating the same values.

```
7.      hersheyHorizontal = [n * m]
8.      for i in range (1, int(n/2)+1)
9.          hersheyHorizontal.append(hersheysCut(i,n) + hersheysCut(m-i, n))
```

Since we change the orientation of cutting the Hershey's bar, now we try to find the best possible solution which is the solution with the least amount of cuts.

Then we return the minimum values of horizontal and vertical results.

```
10.     return min(min(hersheyHorizontal), min(hersheyVertical))
```

The time complexity for the following algorithm is O(n^2).
We come to this complexity since we have 2 for loops and 2 recursive calls which are traversing the half list.

Example of a Hershey's bar A(2,3) using the same algorithm.