Assignment 3

QUESTION 1: Donut.java

Verbal Description

1) The problem statement was to find the most optimal location for the Donut store possible.
2) That location will be the one that will minimize the sum of distances that the traffic police will have to travel to reach the Donut store.
3) To find the intersection which will be the optimal location of the Donut store we find the average of x-coordinates and y-coordinates.
4) X-average and Y-average will be the X-best and Y-best values, respectively.
5) Further, we will find the minimum distance possible by using the given formula:

   $$\sum_{i=1}^{n} |x_{best} - x_i| + |y_{best} - y_i|$$

   The summation will give us the minimum distance.

Pseudo-Code

1) Create a x-array of input size n.
2) Create a y-array of input size n.
3) Store the elements x,y in their respective arrays.
4) Compute the x-average and y-average value.
   x-average(x-best) = sum of all x-values / total number of input elements
   y-average(y-best) = sum of all y-values / total number of input elements
5) Call the MinDistance(x- array, y-array, x-average, y-average).
6) MinDistance (x-array, y-array, x-average, y-average) {

```
        initialize total_distance to 0
        for i=0 till length of array-1, do i++{
                compute Xi = Math.abs(x-average - xi)
                compute Yi = Math.abs(y-average - yi)
                total_distance = total_distance + Xi +Yi

}
return the total_distance computed.
```

Proof of Correctness

This algorithm will always produce the correct output since it will always find the x-best and y-best values and based on that it will use the afore-mentioned formula to compute the minimum distance. The x-best and y-best values will be computed by taking the average ofthe x-values and the y-values, respectively. For even large input size it will produce the correct output since it will always run in O(n) time complexity.

A tight running time estimate of the algorithm

This algorithm will run in **O(n)** time complexity.
Worst case time-complexity : **O(n)**

Assignment 3

A brief reasoning behind the Running time estimate

There are two for-loops in this algorithm. One for storing the values in an array and the second one to iterate n times to compute the final output.

All the other are O (1) time-complexity.

Total complexity = O(n) + O(n) +O (1)

= **O(n)**

QUESTION 2:

a) Pseudocode:

APPROACH - 1

```
i = 0
O(n) → while (i < n) {
O(n) →     l = smallest (arr, i) → returns index of smallest no.
           print (arr[l])
           i = l
       }
```

APPROACH - 2

```
max = 0
O(n) → for ( l = 1 → n) {
           i = l
O(n) →     for ( j = i+1 → n) {
O(1) →         if ( a[i] < a[j]) {
                   len ++
                   i = j                //calculate substring length
               }
O(1) →     max = max (max, len)
       }
O(1) → return max
```

b) For approach 1:

Since it contains just 1 while loop,

Time complexity = $O(n^2)$

For approach 2:

As shown above, running time complexity is $O(n^2)$

Due to the 2 for loops, the (n x n) term dominates.

Assignment 3

c) Both approaches do not work.

① consider i/p sequence → [8,2,5,7,9,1,10]

For approach 1,

[8,2,5,7,9,1,10]

1st itr : 1 ; 2nd itr : 10 ; longest subseq : [1,10]

o/p : 2

However, this isn't true.

Longest subseq. should be [2,5,7,9,10] ; o/p = 5

For approach 2,

[1, 11, 4, 7, 8, 13]

Itr 1 : i=0 arr[i] = 1
j=1 arr[j] = 11
arr[i] < arr[j]
len = 2
for rest of j's,
arr[i] < arr[j]
len = 6
But it is not increasing.

Itr 2 : i=1 arr[i]=11
j=2 arr[j] = 4
arr[i] not less than arr[j] for all j's upto j=7
j=7 arr[j] = 13
arr[i] < arr[j]
len=2
max=2

Itr 3: i=2 arr[2]=4
j=3 arr[3] = 7
arr[i] < arr[j]
len = 2
i=j
i = 3
j = 4
arr[i] < arr[j]
len = 3
⋮
len = 4
so max = 4

Here, max increasing seq. is [1,4,7,8,13]
length = 5

But the code produces seq. [4,7,8,13]
length = 4

Hence, the code fails

② For i/p [1,11,4,7,8,13]

ideal o/p via approach 2 should be [1,4,7,8,13]

③ The solution produced by the greedy approach is -
[4,7,8,13]

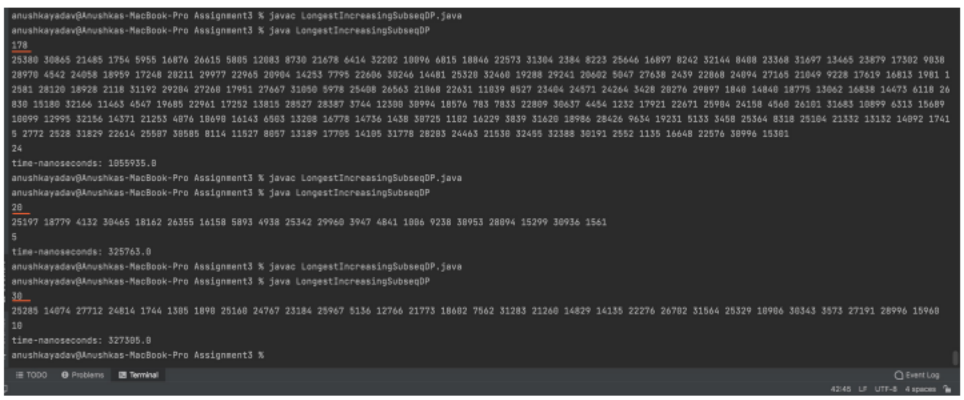
Assignment 3

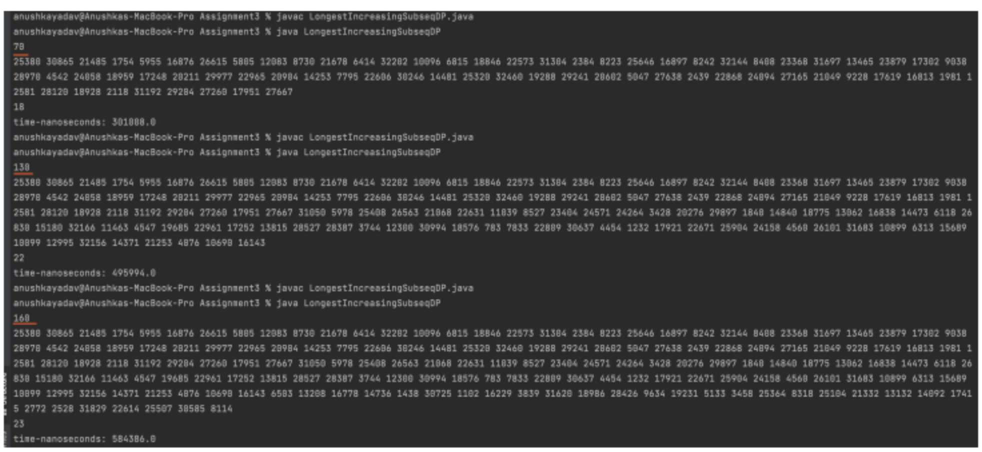
QUESTION 3: LongestIncreasingSubseqDP.java

Time complexity for recursive approach: $O(2^n)$ will be exponential. Due to this the last test case will always fail since n =10000.

c)

input-size	run time
10	360675.0
20	325763.0
30	327305.0
50	269502.0
60	301064.0
70	301008.0
90	263901.0
130	495994.0
160	584386.0
178	1055935.0





Assignment 3

```
anushkayadav@Anushkas-MacBook-Pro Assignment3 % javac LongestIncreasingSubseqDP.java
anushkayadav@Anushkas-MacBook-Pro Assignment3 % java LongestIncreasingSubseqDP
90
25388 30865 21485 1754 5955 16876 26615 5885 12083 8730 21678 6414 32202 18096 6815 18846 22573 31304 2384 8223 25646 16897 8242 32144 8408 23368 31697 13465 23879 17302 9838
28978 4542 24858 18959 17248 20211 29977 22965 20984 14253 7795 22686 30246 14481 25328 32460 19288 29241 20682 5047 27638 2439 22868 24094 27165 21849 9228 17619 16813 1981 1
2581 28120 18928 2118 31192 29204 27260 17951 27667 31050 5978 25408 26563 21068 22631 11039 8527 23404 24571 24264 3428 20276 29897 1840 14840 18775 13062 16838 14473
19
time-nanoseconds: 263901.0
anushkayadav@Anushkas-MacBook-Pro Assignment3 % javac LongestIncreasingSubseqDP.java
anushkayadav@Anushkas-MacBook-Pro Assignment3 % java LongestIncreasingSubseqDP
50
25388 30865 21485 1754 5955 16876 26615 5885 12083 8730 21678 6414 32202 18096 6815 18846 22573 31304 2384 8223 25646 16897 8242 32144 8408 23368 31697 13465 23879 17302 9838
28978 4542 24858 18959 17248 20211 29977 22965 20984 14253 7795 22686 30246 14481 25328 32460 19288 29241 20682
15
time-nanoseconds: 269502.0
anushkayadav@Anushkas-MacBook-Pro Assignment3 % javac LongestIncreasingSubseqDP.java
anushkayadav@Anushkas-MacBook-Pro Assignment3 % javac LongestIncreasingSubseqDP.java
anushkayadav@Anushkas-MacBook-Pro Assignment3 % java LongestIncreasingSubseqDP
60
25388 30865 21485 1754 5955 16876 26615 5885 12083 8730 21678 6414 32202 18096 6815 18846 22573 31304 2384 8223 25646 16897 8242 32144 8408 23368 31697 13465 23879 17302 9838
28978 4542 24858 18959 17248 20211 29977 22965 20984 14253 7795 22686 30246 14481 25328 32460 19288 29241 20682 5047 27638 2439 22868 24094 27165 21849 9228 17619 16813
16
time-nanoseconds: 381864.0
```

```
anushkayadav@Anushkas-MacBook-Pro Assignment3 % javac LongestIncreasingSubseqDP.java
anushkayadav@Anushkas-MacBook-Pro Assignment3 % java LongestIncreasingSubseqDP
10
41 18467 6334 26500 19169 15724 11478 29358 26962 24464
4
time-nanoseconds: 360675.0
```

For the DP solution the time is being computed in nano seconds. When trying to get the run time in seconds it was taking 0 seconds for the inputs that the program was run on because the algorithm will run in O(n^2) time complexity. From the given table it can be observed that for different n inputs the run-time varied a little in terms of nano seconds.

NOTE: *We tried to get the time in seconds initially but since it wasn't taking the algorithm that long to execute on the inputs given, we did it in ~~milli~~ nano seconds, instead. Also, the terminal wouldn't take more than 180 inputs so the maximum of 178 n-inputs were given for run-time.*

QUESTION 4: Hopscotch.java

For this program Dynamic Programming approach was used. A new array called OPT [] was created which kept track of all the maximum sums when starting from position i.

Here, we start calculating from the end of the original array to get the maximum possible sum of numbers by either jumping 2 steps or 3 steps.

It will continue to calculate the sum until we reach the end of the original array, therefore the maximum sum stored in the OPT [] array at the position 0 (because that this the position we will be at when we reach the end of the original array) will be the output for the given original array.

There's also a corner case that we had to take care of, i.e., for the first 3 positions we had to compute the value and store it in the OPT [] array since, initially the len-1, len-2 and (len-3 + len-1) will not exist in the OPT [] array. So, to make sure that we either consider 2ndelement or the 3rd element when computing the maximum sum, we have to store the first 3 possible values in the OPT [] array.

Assignment 3

Another corner case will be if the length of the original array is less than 3 which would mean we cannot make the required jumps to get the maximum sum because we will reach the end of the array. In this case we will just return the value at the 0th index position of the original array as the output.

Further, to compute the maximum sum at each step we use the following formula where the for-loop will begin from (len-4) position:

maximum_value = Math.max(OPT[j+2], OPT[j+3])	*to get the maximum of the two positions that we can jump to.*
OPT[j] = input_array[j] + max_val	*the sum will then be stored at the jth index in the OPT [] array.*

Therefore, the OPT [0] will give us the final output, since by the end of the loop it will have considered all the elements of the original array.

Time complexity:
The program will run in **O(n)** time since there are only 3 for-loops, one for initializing all the values to 0 for the OPT[] array and the other to traverse over the original array to compute the maximum sum. The third for-loop is used to store the user entered values in the original array.

Assignment 3

RECURRENCE RELATION

$$OPT[j] = \begin{cases} input\_array[0] & , \text{ if } input\_array.length < 3 \\ input\_array[j] + \max(OPT[j+2], OPT[j+3]) & , \text{ if } input\_array.length > 3 \end{cases}$$

The return statement is in the method MaximumSum() where the maximum possible sum is being computed.

return OPT [0];

Assignment 3