Assignment – 4

Question 1:

Verbal Description:

The goal of this program is to find the kind of increasing subsequence and at every step you have to consider three consecutive numbers where the average of first two elements should be less than the third element.

What

Here, we have used a 2-d array named dynamic-array which keeps track of the maximum length of the sequence at that specific index.

In the end we output the maximum of the dynamic-array we created.

How

dynamic-array[k][l] = max(dynamic-array[k][l] , dynamic-array[m][k] +1)

This dynamic-array keeps track of the maximum length of the sequence at that specific index and it is calculated based on the condition

[(aji + aji+1) / 2aji+2]. If this condition is satisfied then the maximum of

(dynamic-array[k][l] , dynamic-array[m][k] +1) will be chosen and the dynamic-array will be updated by plus 1 because we choose to include that specific element.

We will continue to do this until we reach the end of the for-loop and the value at the end of the dynamic-array will be the final output which is the maximum length.

Where

The final output will be the last element in the dynamic-array at the last index of that array which will be returned to the main function and displayed as output.

Pseudo-Code

```
increasingSequence method

Input array - x

Max-length = 0

Initialize 2-d dynamic-array

for (k =0 till length of array-1){
```

Assignment – 4

```
        for (l =k+1 till length of array-1){
                dynamic-array[k][l] = 2
                dynamic-array = calculateLength(dynamic-array, k, l, x)
                max-length = max.length(max-length, dynamic-array[k][l])


}
}
return max-length
```

calculateLength() method

```
calculateLength(dynamic-array, k, l, x){
        for (m =0 till k){
                if(x[k]+x[m]/2 < x[l]){
                        dynamic-array[k][l] = max(dynamic-array[k][l] ,
                        dynamic-array[m][k] +1)
}
return dynamic-array
}
```

Running Time Estimate

O(n^3)

There are three for loops, the first one runs from 0 till n, the second one will run from i +1 till n and the third for-loop will run from 0 till i.

Therefore, the overall complexity of the algorithm will be O(n^3).

Brief Reasoning

There are three for loops, the first one runs from 0 till n, the second one will run from i +1 till n and the third for-loop will run from 0 till i.

Therefore, the overall complexity of the algorithm will be O(n^3).

Question 2:

Verbal Description:

The goal of the algorithm is to maximize the output between the limits of W1 and W2.

To describe the heart of the algorithm-

WHAT - `m[n][W] maximum cost of n items given bounded weights between W1…W2`

HOW - `Math.max(v[i-1]+m[i-1][j-w[i-1]], m[i-1][j]);`

```
      0 when m[n][W] = 0
```

WHERE - `m[n][W] which will hold the maximum value.`

PseudoCode:

```
for (int i to n { //for every number
if(w[i-1] <= j) {
                        m[i][j] = Math.max(v[i-1]+m[i-1][j-w[i-1]], m[i-1]
[j]);
                  }
                  else {
                        m[i][j] = m[i-1][j]; //skip
                  }
              }
            }
```

Running Time Estimate: O(nW)

Reasoning –

This algo has 2 for loops. One runs for n elements and the other runs for iterations going on till W.

It is a pseudo polynomial.

Question 3:

Verbal Description:

The goal of the program is to find the largest side k that exists in the given n x n grid such that there exists a k x k all-white square inside it.

To describe the heart of the algorithm-

Assignment – 4

WHAT – we have decided to use OPT[i][j] that represents the maximum length of the side of the square that contains all white squares inside it.

HOW – $OPT(i, j) = \min(OPT(i-1, j), OPT(i-1, j-1), OPT(i, j-1)) + 1$

For every 'w' found, our recurrence relation is as mentioned above. So we update the current location based on the best possible solution considering the element above, the element to the left and the element diagonally to the left. Once, this value is updated, we determine the maximum value using the max function.

WHERE – OPT[i][j] is the bottom right corner that contains the maximum length up till i, j.

Pseudo Code:

```
for (int i->n)
    for (j=1->n)
        if (element at arr[i-1] [j -1] is 'w')
                Calculate OPT[i][j] using min (OPT ($i-1, j$), OPT ($i-1, j-1$), OPT ($i, j-1$)) + 1
                Find max OPT store in k
return k
```

Proof of Correctness:

Consider the following matrix –

0	1	2	3	4	
b	w	w	w	w	0
w	w	w	w	w	1
b	b	w	w	b	2
b	b	b	b	b	3
w	b	b	w	w	4

If you eyeball it, we can easily make out a 2x2 all white matrix. The algorithm builds another matrix that records the number of white blocks it has seen till that particular cell of i, j. So, if you consider cell [1][3] it will contain value 2. Basically, every time a 'w' square is seen, the algorithm counts it if all its neighbors are also white. This can change for every iteration. Which is why we calculate the max value after every iteration.

Running time estimate:
$O(n^2)$

Brief Reasoning:

Assignment – 4

In the algorithm we have 2 for loops that will run n times in the worst case. The rest of the operations are O(1).
Hence the n^2 term dominates all. Hence it is $O(n^2)$.

Question 4:
Verbal Description:

The goal of the problem is to find the optimal minimal cost and the optimal parenthesizing. In order to do so, we have implemented the matrix chain multiplication algorithm we discussed in class. To print the parenthesis, we implemented another function called printParenthesis() that prints out the order of multiplication of the two matrices.

WHAT – OPT[i][j] represents minimum number of operations to multiply A1….An

HOW – OPT[i][j] = min(OPT[i][k] + OPT[k + 1][j] + A[i - 1] * A[k] * A[j]) where i<=k<=j

= 0 when i==j

WHERE – OPT[1] [n-1]

Pseudo Code:

```
matrixChainMultiplication()
for(d=1->n)
    for(i=1 to n-d)
        calculate j=i+d
        OPT[i][j] = infinity initially
        for(k=i->j-1)
        Calculate tmp as OPT[i][k] + OPT[k + 1][j] + A[i - 1] * A[k] * A[j];
        if OPT[i][j]>tmp
            Update OPT[i][j] to tmp and parenthesis[i][j] to k
Return OPT[1][n-1]
Pass parameters 1, n-1, n, parenthesis to function printParenthesis()
printParenthesis()
if there is just one matrix left i.e. i==j
    print this matrix
recursively implement from i to parenthesis[i][j]
recursively implement from parenthesis [i][j] to j
```

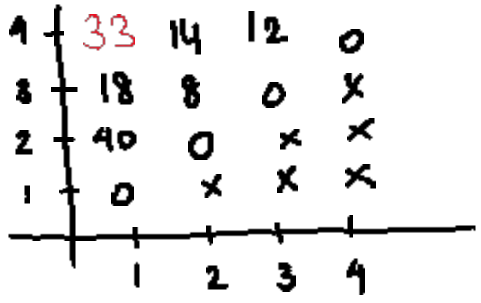
Proof of Correctness:

Assignment – 4

The algorithm discussed is correct because we find a range (i, j) for which the value is already calculated. We then return the minimum of that range. The number present at the top right corner is the final return value. Our algorithm works for the following example in the following way-

Consider arrays – A1, A2,A3,A4 each with dimensions 5x2, 2x4, 4x1, 1x3.

Our dp looks like this-



4	33	14	12	0
3	18	8	0	x
2	40	0	x	x
1	0	x	x	x
	1	2	3	4

base case

$S[K][K] = 0 \quad 1 \leq k \leq n$

Thus 33 will be outputted which is the correct answer.

Running time Estimate: $O(n^3)$

Brief Reasoning:

We have 3 for loops that run. Each for loop iterates over n items at the most. Rest of the operations are of O (1). In the printParenthesis() we have a recursive call that also occurs at most n times. So, complexity is n^3 as this term dominates the rest.