Introduction to Algorithm – Assignment 3

1. Reinforce your understanding of the selection and the median finding problem.

a. Write an iterative version of Randomized Select

⇨ The pseudocode of an iterative version of Randomized select is as follows:

```
RANDOMIZED-SELECT(A,p,r)
  if p=r
    return A[p]
  q= RANDOMIZED-PARTITION(A,p,r)
  k=q-p+1
  if i=k
    return A[q]
  else if i<k
    for each j=p to q-1
      if j=i
        return A[j]
  else
    for each j=q+1 to r
     if j=(i-k)
       return A[j]
```

The algorithm works by finding a pivot element by using the Randomized-partition method and then checks if the element we are searching for is equal to the pivot element.

- If it is equal to the pivot element we have reached our goal and we can return the element.
- If it is less than the pivot element then we discard the second half of the array(the part after pivot element) and iterate through the elements in the first half, once we find the element we return it.
- If it is greater than the pivot element then we discard the first half of the array(the part before the pivot element) and iterate through the elements in the second half, once we find the element we return it.

The running time of the algorithm will be O(n).

b. Argue that the algorithm SELECT doesn't work in linear time.

⇨ The running time of the function SELECT is dependant on:

$$T(n) = c.n + T(n/a) + T (LL \text{ or } RL)$$

Here,
c.n is linear. It is computed by finding the medians and partition time
T(n/a) is the recursive call to median. A is depended on the group size of the array is splitted
T(LL or RL) is the time which would be computed depending upon which list- left list or right list is called in correspondence to median of medians.

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5
X1	X2	X3	X4	X5
C1	C2	C3	C4	C5
D1	D2	D3	D4	D5

X1, x2,x3,x4,x5 represent the medians of the respective group of input.
the elements highlighted in yellow represent the elements which are greater or equal to the median of medians, given by X3 in this case.

So if we count the number of highlighted elements as against the total elements, it comes out to be greater than $1/4^{th}$ of the total elements.
this means that if $1/4^{th}$ of the elements are greater than the median of medians than, less than $3/4^{th}$ of the total elements should be less than median of medians.
So taking the worst case scenario, and substituting the values in the equation,

$$T(n) = c.n + T(n/5) + T(3n/4)$$

If numbers are divided in group of 7:

$$T(n) = c.n + T(n/7) + T(3n/4)$$

This is linear time.

And if numbers are divided in group of 3:

$$T(n) = c.n + T(n/7) + T(3n/4)$$

This is not linear time.

c. Let X[1..n] and Y[1..n] be two arrays, each containing n numbers already in sorted order. Give an O(log n)-time algorithm to find the median of all 2n elements in arrays X and Y.

⇨ We have 2 arrays X[1..n] and Y[1..n], each having n elements.

The Psedocode for the algorithm is:

```
MEDIAN(X[1..n],Y[1..n])
  if n =1
    min[X(1), Y(1)]
  //Else Get the median index of all the arrays.
  if n % 2 = 0
    m = n/2
  else:
    m = (n/2) + 1
  if X[m] < Y[m]
    return MEDIAN(X [m..n], Y [1..m])
  else
    return MEDIAN[X[1..m], Y [m..n])
```

We know that we have two sorted lists X and Y.
There can be 2 scenarios, if n is odd and n is even. If n is odd then median will lie at n+1/2 and if it is even it will lie at n/2.
From here we can infer that there are m-1 elements smaller than the median element and n-m elements larger than the median element.

If X[m] > Y[m], then median is present in one of the following:
1) X[1] to X[m]
2) Y[m] Y[n]
If Y[m] > X[m], then median is present in one of the following:
1) X[m] to X[n]
2) Y[1] to Y[m]

The time complexity for MEDIAN is O(lgn) and the algorithm implemented is Divide and Conquer.

2. Purpose: reinforce your understanding of dynamic programming and the matrix-chain multiplication problem, and practice run time measurements.
a) Please implement a recursive, a dynamic programming, and a memoized version of the algorithm for solving the matrix-chain multiplication problem described in our textbook (Chapter 15), and design suitable inputs for comparing the run times, the number of recursive calls, and the number of scalar multiplications for both algorithms. Describe input data, report, graph and comment your results. Please submit your programs in three separate files

=> For dynamic programming, the list values are as follows:

The unit for **scalar multiplications** is integer.
The unit for **Time taken** is in microseconds.
The unit for **no. of recurrences** is in integer.

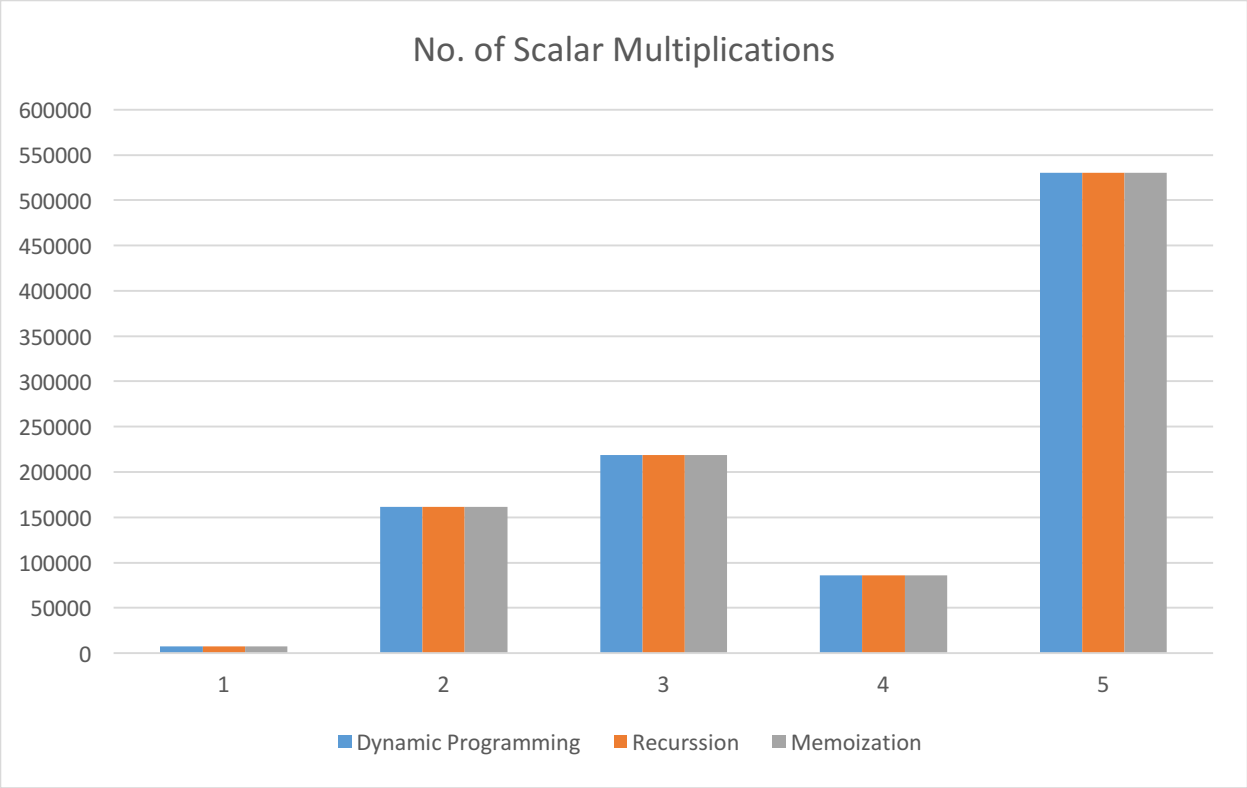
Dynamic Programming					
	p1 = 4	p2 = 8	p3 = 12	p4 = 16	p5 = 20
Scalar Multiplications	7875	161500	219000	85672	530390
Time Taken	9.824	1.364	26.577	72.889	106.155
No. of Recurrences	0	0	0	0	0

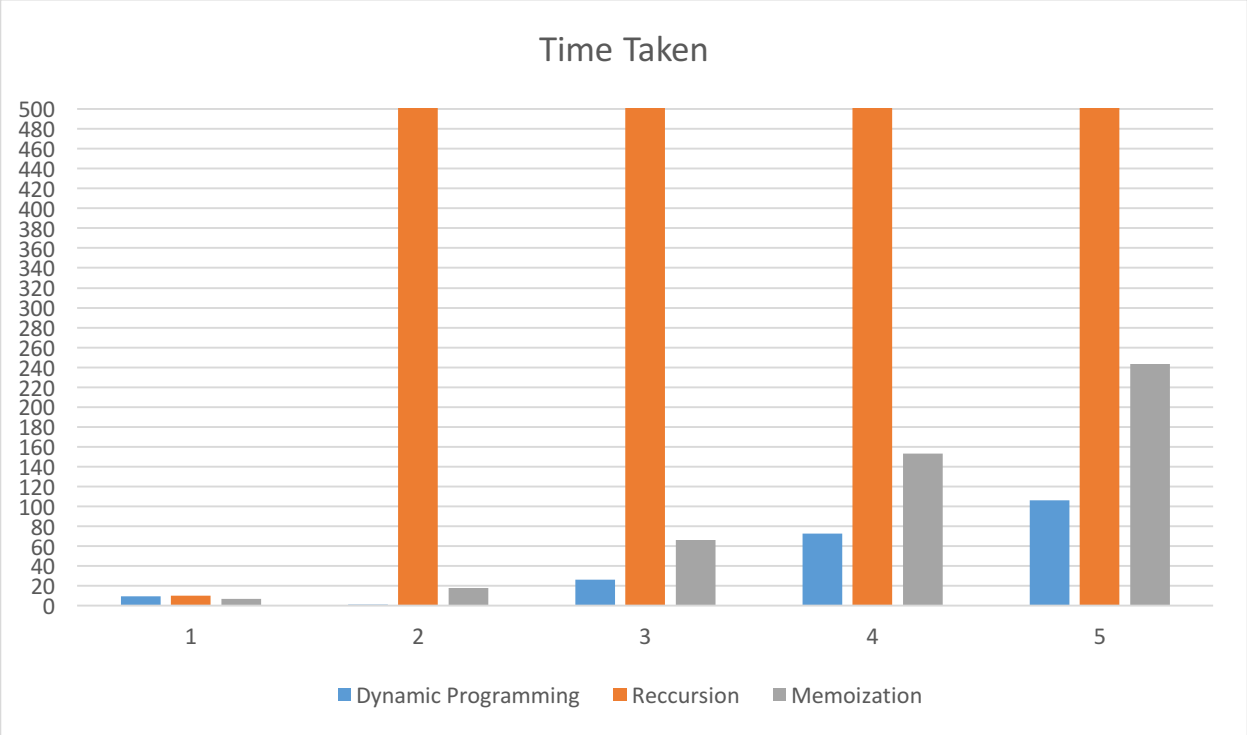
For Recurssion, the list of values are as follows:

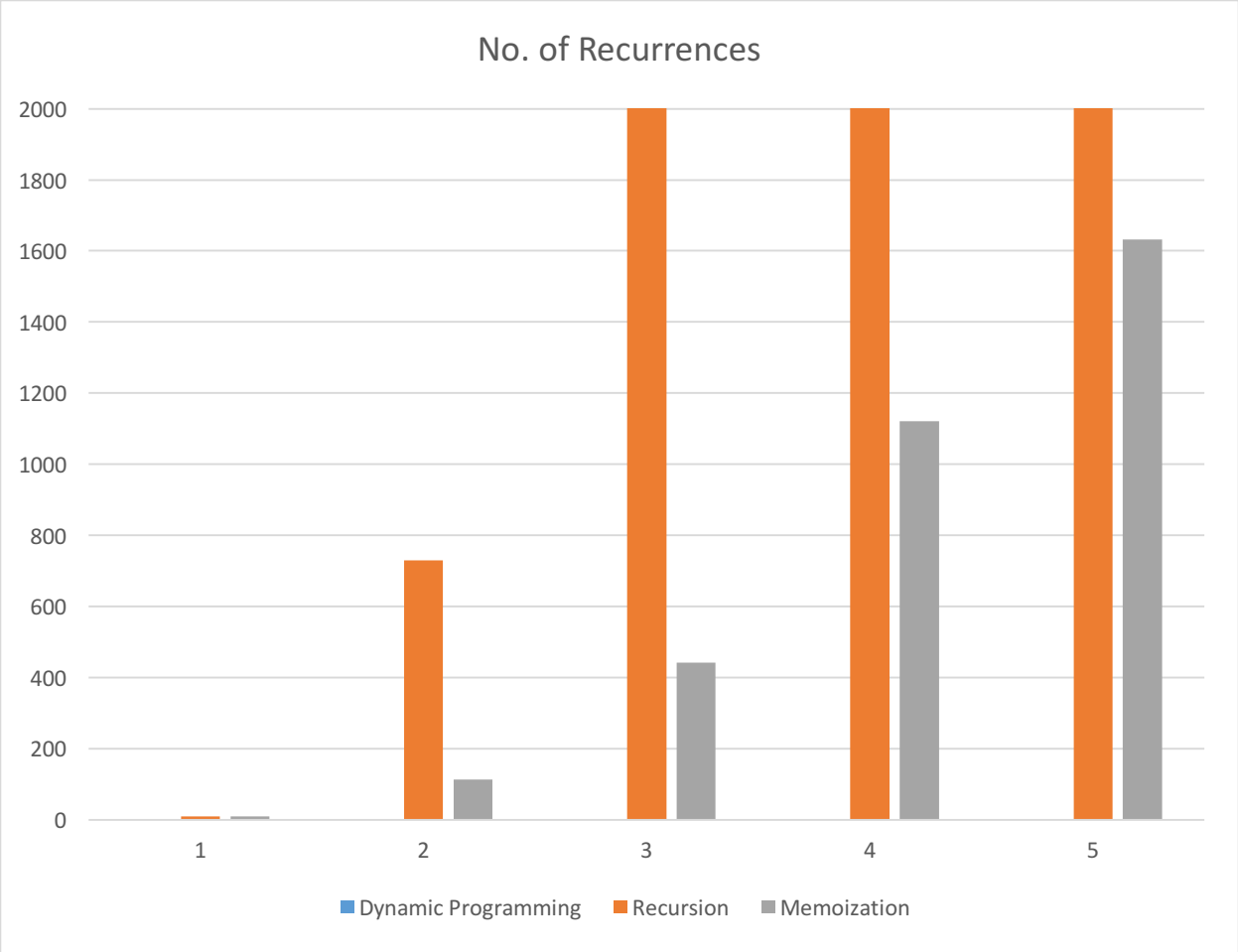
Recursive					
	p1 = 4	p2 = 8	p3 = 12	p4 = 16	p5 = 20
Scalar Multiplications	7875	161500	219000	85672	530390
Time Taken	10.357	1929.973	3885.158	90722.163	818540.736
No. of Recurrences	9	729	59049	4782969	43046721

For Memoization, the list of values are as follows:

Memoization					
	p1 = 4	p2 = 8	p3 = 12	p4 = 16	p5 = 20
Scalar Multiplications	7875	161500	219000	85672	530390
Time Taken	7.237	17.853	66.526	153	243.644
No. of Recurrences	9	113	441	1121	1633


No. of Scalar Multiplications
600000
550000
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
1
2
3
4
5
Dynamic Programming
Recurssion
Memoization


Time Taken
500
480
460
440
420
400
380
360
340
320
300
280
260
240
220
200
180
160
140
120
100
80
60
40
20
0
1
2
3
4
5
Dynamic Programming
Reccursion
Memoization



We can infer the following points from the graph:

1. The no. of scalar multiplications is not directly proportional to the length of the matrix chain. It is dependent on both the length of the matrix chain and the dimensions of the matrix in the matrix chain.
2. The running time of an algorithm depends on the length of the matrix chain and grows:
   -> Asymptotically at $O(n^3)$ for Dynamic Programming
   -> Exponentially at $O(2^n)$ for Recurssion
   -> Asymptotically at $O(n^3)$ for Memoization
3. The no. of recurrences is:
   -> 0 for Dynamic Programming
   -> Depends on length of matrix chain and the dimensions for Recurrsion.
   -> Depends on length of matrix chain and the dimensions for Recurrsion.
   -> Is smaller for Memoization than recursion because of its optimal structure

b) Find an optimal parenthesization of a matrix-chain product whose sequence of dimension is <5, 2, 4, 3, 7, 9, 7, 8, 6, 1, 3, 7, 6, 5>. How many multiplications does this parenthesization require? How many multiplications needs a naive algorithm which multiplies the input matrices in their input order?

⇨ The given input has the optimal parenthesization as follows:
((A1(A2(A3(A4(A5(A6(A7(A8A9)))))))(A10A11)A12)A13)
No. of multiplications are 399. If this was implemented in a naïve algorithm, the number of multiplications would be 1865.

3. Purpose: practice algorithm design, and dynamic programming. Consider a sequence of n distinct integers S[1], …S[n]. The longest non-decreasing subsequence problem asks you to find the longest sequence $(i_1, \ldots i_k)$ such that $i_j < i_{j+1}$ and $S[i_j] <= S[i_{j+1}]$ for $j \in \{1, \ldots, k-1\}$. For example, consider the sequence: 3, 45, 23, 9, 3, 99, 108, 76, 12, 77, 16, 18, 4. A longest non-decreasing subsequence is 3, 3, 12, 16, 18, having length 5.

a) Let S[1], …S[n] be a sequence of n integers. Denote l_i be the length of the longest non-decreasing subsequence that ends in (and includes) S[i]. Use dynamic programming to compute l_i and give an algorithm that generates the corresponding non-decreasing subsequence.

⇨ The pseudocode for finding the longest non-decreasing sub-sequence is as follows.
Let n = length of the array S.

```
Longest-non-decreasing-Subsequence(S,n)
  for i ← 1 to n
    l[i] ← 1

  for i ← 2 to n
    for j ← 1 to i – 1
      if S[j] <= S[i]
        l[i] = max(T[i], T[j]+1)
```

The above code follows the dynamic programming approach. An array ‘L’ is initialized with the same length as that of the sequence. All the elements of the array ‘L’ is initialized with 1, indicating that the longest non-decreasing subsequence can be 1 at the beginning. Next, we iterate through the array from the 2nd element to the nth element, inside the ith loop we iterate from j ← 1 to i-1. If the value of the element at j^{th} index is less than the value of element at i^{th} index, then we can assume that we have found the next element in our non-decreasing subsequence. Once we know that we have found the next element, we can increase the count for the corresponding i^{th} element in the ‘L’ array. Hence, denoting that we have the next element for our subsequence.

The time complexity for this algorithm results in $\Theta(n^2)$. Because we need atleast 2 nested loop to simulataneously iterate and compare values in the array.

b) Using $l_1 \ldots l_n$, how do you solve the longest nondecreasing subsequence problem?

⇨ Using the L array we got in the previous section, we can do the following:

```
m = index of largest_no_in(l)
sequence ← [S[m]]
for j← m-1 downto 1
  if l[j] = l[m]-1
    if S[j] <= S[m]
      sequence.add(S[j])
      m = j
```

The above code explains the way we can find the longest non-decreasing subsequence once we have the 'length' array 'L'. We start iterating from the highest number which gives the longest subsequence length. The sequence array is initialized with the last element of the longest non-decreasing subsequence which will be at position 'm'. Each iteration we try and find the element with the m-1 value, which will give us the next element for the subsequence.

Each time we find the next number for our longest non-decreasing subsequence, if the number at that index in the S array is less than the number at m index in the S array, we add it to the subsequence array. We change the value of m to the value of j for our next comparison.

c) An alternative approach: Design an algorithm that use theLongest Common Subsequence (LCS) Problem that we have discussed in class to solve the longest nondecreasing subsequence problem.

⇨ Let S be the original sequence. We need to find the longest non-decreasing subsequence by using LCS problem.

```
S = Sequence of numbers.
M = sort-in-ascending-order(S)
LCS-length(S, M)
  x = S.length
  y = M.length
```

```
  let b[1…x, 1…y] and c[9…x, 0…y] be new tables
  for i = 1 to x
    c[i,0] = 0
  for j = 0 to y
    c[0,j] = 0
  for i = 1 … x
    for j = 1 … y
      if S[i] == M[j]
        c[i,j] = c[i-1, j-1] + 1
        b[i,j] = "↖"
      elseif c[i-1,j] >= c[i,j-1]
        c[i,j] = c[i-1,j]
        b[i,j] = "↑"
      else c[i, j] – c[i, j-1]
        b[i,j] = " "
return c and b
```

To find the longest non-decreasing subsequence using LCS method, we just need to sort the sequence in ascending order and store it into another array. Once done, we can calculate the longest non-decreasing subsequence by find the longest-common-subsequence between the 2 sequences.

The total time complexity of the algorithm will be $O(x^2 + xlgx)$. The xlgx part is for sorting the original sequence of length X. And to find the LCS of 2 sequence of length m and n, we know that the time complexity is O(m.n). In this case it will be O(x.x) = $O(x^2)$. So the time complexity is denominated by $O(x^2)$.

4. Purpose: practice designing greedy algorithms. Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible…

⇨ The pseudocode for the above problem is as follows:

Let **X[1…n]** be an array of houses scattered on the road, **Y[1…k]** be the cell towers required for the n houses and **d** be the maximum distance required to maintain a reasonable signal between a house and a tower.

```
int i =1, j =1
Y[j] = X[i] + d
i ← i + 1
while (i < = n)
  if (|y[j] – x[i]| > d)
    j ← j + 1
    y[j] = x[i] + d
  i ← i + 1
```

The above algorithm works in a greedy manner. Wherein at each step it tries and find the best outcome for the problem. For this problem the objective is to minimize the number of towers, in other words the number of elements in the array 'Y'. The algorithm achieves this by checking at each step the number of houses a particular tower can cover in its vicinity while maintaining a reasonable signal. To make sure that we have the optimal solution, we assume that the first tower Y[1] is at a distance of d from the first house X[1]. Then we check the number of houses which can be covered under the signal of the given tower. Once we check that, we can iterate to the next uncovered house and then place the next tower at a distance of 'd' from there and perform the same steps iteratively.

The time complexity of the greedy algorithm will be O(n) where n is the number of houses. Each i^{th} iteration includes 3 arithmatic operations and 1 comparisons.