Nadine Mohamed Mostafa 20107088

Algorithm assignment

For each of the following algorithms:

1- Insertion sort
2- Quick sort
3- Knapsack brute force
4- Bubble sort
5- Multiplication of large numbers (divide and conquer)

You should provide:

1- The tracing for the algorithm using your id for example if your id is 20103421 than the input array should be 2,0,1,0,3,4,2,1 as for the Knapsack problem the weights and values should also be deduced from your id weights for item 1 to 4 should be 2,0,1,0 and values should be 3,4,2,1. Moreover, Multiplication of large numbers the first number should be 2010 and second number should 3421.
2- The pseudo code and the mathematical analysis to compute the time complexity then define the class efficiency θ. **Providing the final answer without the providing the mathematical analysis will not be acceptable.**

Insertion Sort

```
function insertionSort(arr):
   for i from 1 to length(arr) do:
      key = arr[i]
      j = i - 1
      while j >= 0 and arr[j] > key do:
         arr[j+1] = arr[j]
         j = j - 1
      arr[j+1] = key
   return arr
```

Best Case: O(n) - This occurs when the input array is already sorted, and each element is compared with its previous element only once.
Worst Case: O(n^2) - This occurs when the input array is sorted in reverse order. In this case, each element needs to be compared with all the previous elements, resulting in a quadratic time complexity.

Trace:
Initial array: [2, 0, 1, 0, 7, 0, 8, 8]

Iteration 1:
Key element: 0
Compare 0 with elements to its left: [2]
Insert 0 before 2: [0, 2, 1, 0, 7, 0, 8, 8]

Iteration 2:

Key element: 1
Compare 1 with elements to its left: [0, 2]
Insert 1 before 2: [0, 1, 2, 0, 7, 0, 8, 8]
Iteration 3:
Key element: 0
Compare 0 with elements to its left: [0, 1, 2]
Insert 0 before 1: [0, 0, 1, 2, 7, 0, 8, 8]

Iteration 4:

Key element: 7
Compare 7 with elements to its left: [0, 0, 1, 2]
Insert 7 after 2: [0, 0, 1, 2, 7, 0, 8, 8]

Iteration 5:

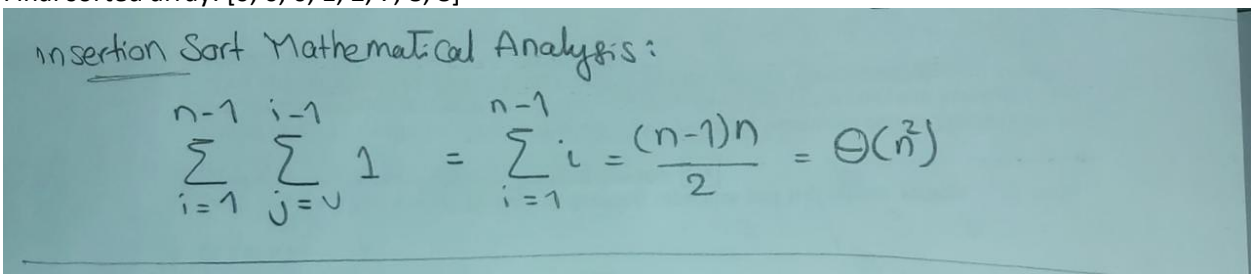
Key element: 0
Compare 0 with elements to its left: [0, 0, 1, 2, 7]
Insert 0 before 1: [0, 0, 0, 1, 2, 7, 8, 8]

Iteration 6:

Key element: 8
Compare 8 with elements to its left: [0, 0, 0, 1, 2, 7]
Insert 8 after 7: [0, 0, 0, 1, 2, 7, 8, 8]

Iteration 7:

Key element: 8
Compare 8 with elements to its left: [0, 0, 0, 1, 2, 7, 8]
Insert 8 after 8: [0, 0, 0, 1, 2, 7, 8, 8]
Final sorted array: [0, 0, 0, 1, 2, 7, 8, 8]



Quick Sort

Hoare Partition (A[l..r])
P← A[l]
i ← l; j ← r +1
repeat
repeat i ← i + 1 until A[i]≥ p
repeat j ← j − 1 until A[j]≤ p
swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j]) //undo last swap when i > j swap(A[I], A[j])
return j

Initial array: [2, 0, 1, 0, 7, 0, 8, 8]

Iteration 1:
Pivot: 2
Partitioned array: [0, 0, 1, 0, 2, 7, 8, 8]

Iteration 2:
Pivot: 0
Partitioned array: [0, 0, 1, 0, 2, 7, 8, 8]

Iteration 3:
Pivot: 1
Partitioned array: [0, 0, 1, 0, 2, 7, 8, 8]

Iteration 4:
Pivot: 0
Partitioned array: [0, 0, 0, 1, 2, 7, 8, 8]

Iteration 5:
Pivot: 7
Partitioned array: [0, 0, 0, 1, 2, 7, 8, 8]
Iteration 6:

Pivot: 8

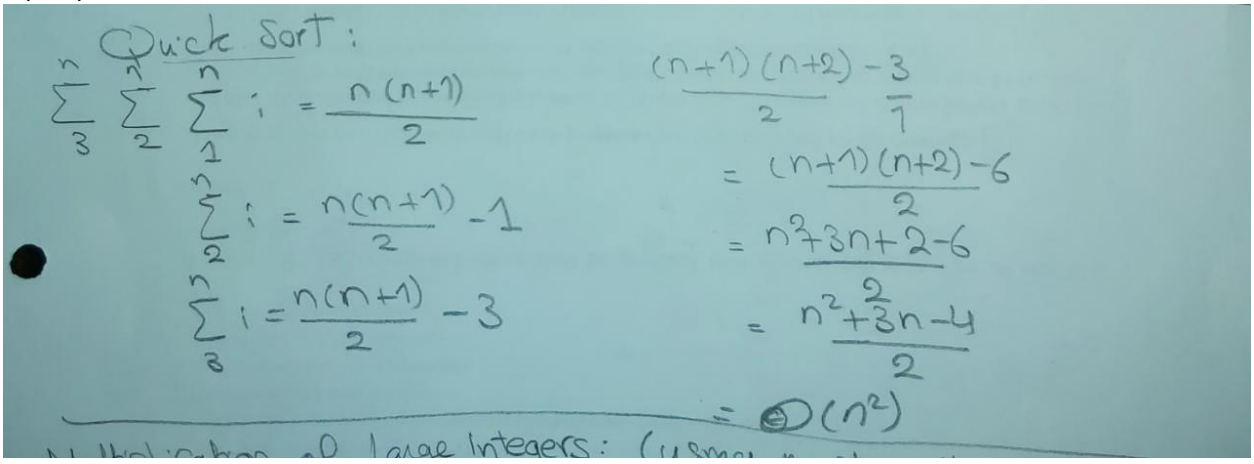
Partitioned array: [0, 0, 0, 1, 2, 7, 8, 8]

Iteration 7:
Pivot: 8
Partitioned array: [0, 0, 0, 1, 2, 7, 8, 8]
No more iterations are needed since the array is already sorted.

Final sorted array: [0, 0, 0, 1, 2, 7, 8, 8]
The sub-array is already sorted.

```
[2, 0, 1, 0, 7, 0, 8, 8]
[0, 0, 1, 0, 2, 7, 8, 8]
[0, 0, 0, 1, 2, 7, 8, 8]
```

Best Case:
time complexity is O(n log n)
Worst Case:
O(n^2)



Knapsack brute force

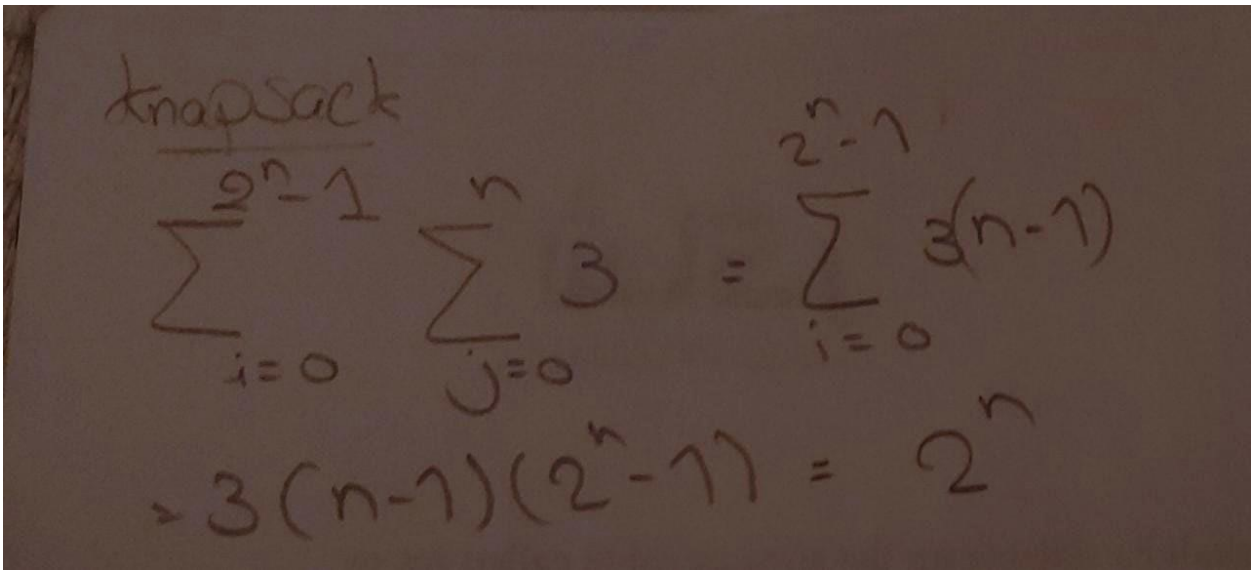
```
function knapsackBruteForce(values[], weights[], maxWeight):
    n = length(values)
    maxProfit = 0
    bestSubset = []

    for i = 0 to (2^n) - 1:
        currentWeight = 0
        currentProfit = 0
        subset = []

        for j = 0 to n - 1:
            if ith bit of i is set:
                currentWeight += weights[j]
                currentProfit += values[j]
                subset.append(j)
```

```
    if currentWeight <= maxWeight and currentProfit > maxProfit:
        maxProfit = currentProfit
        bestSubset = subset
  return maxProfit, bestSubset
```

Iteration 1:	Subset: []	Current weight: 0	Current profit: 0
Iteration 2:	Subset: [Item 1]	Current weight: 7	Current profit: 2
Iteration 3:	Subset: [Item 2]	Current weight: 0	Current profit: 0
Iteration 4:	Subset: [Item 1, Item 2]	Current weight: 7	Current profit: 2
Iteration 5:	Subset: [Item 3]	Current weight: 8	Current profit: 1
Iteration 6:	Subset: [Item 1, Item 3]	Current weight: 15	Current profit: 3
Iteration 7:	Subset: [Item 2, Item 3]	Current weight: 8	Current profit: 1
Iteration 8:	Subset: [Item 1, Item 2, Item 3]	Current weight: 15	Current profit: 3
Iteration 9:	Subset: [Item 4]	Current weight: 8	Current profit: 0
Iteration 10:	Subset: [Item 1, Item 4]	Current weight: 15	Current profit: 2
Iteration 11:	Subset: [Item 2, Item 4]	Current weight: 8	Current profit: 0
Iteration 12:	Subset: [Item 1, Item 2, Item 4]	Current weight: 15	Current profit: 2
Iteration 13:	Subset: [Item 3, Item 4]	Current weight: 16	Current profit: 1
Iteration 14:	Subset: [Item 1, Item 3, Item 4]	Current profit: 3	Current weight: 23
Iteration 15:	Subset: [Item 2, Item 3, Item 4]	Current profit: 1	Current weight: 16
Iteration 16	Subset: [Item 1, Item 2, Item 3, Item 4]	Current profit: 3	Current weight: 23



Bubble sort

```
function bubbleSort(arr):
   n = length(arr)
   for i from 0 to n-1 do:
      for j from 0 to n-i-1 do:
         if arr[j] > arr[j+1] do:
            swap(arr[j], arr[j+1])
   return arr
```

Iteration 1:
Start with the given array [2, 0, 1, 0, 7, 0, 8, 8].
Compare the first element (2) with the second element (0). Since 2 is greater than 0, swap them.
The array becomes [0, 2, 1, 0, 7, 0, 8, 8].
Compare the second element (2) with the third element (1). Since 2 is greater than 1, swap them.
The array becomes [0, 1, 2, 0, 7, 0, 8, 8], etc...
Iteration 2:
Compare the third element (2) with the fourth element (0). Since 2 is greater than 0, swap them.
The array becomes [0, 1, 0, 2, 7, 0, 8, 8].
Compare the fourth element (2) with the fifth element (7). Since 2 is smaller than 7, no swap is needed,
Compare the fourth element (2) with the sixth element (0). Since 0 is smaller than 2, swap them,
[0, 0, 1, 0, 2, 7, 8, 8]

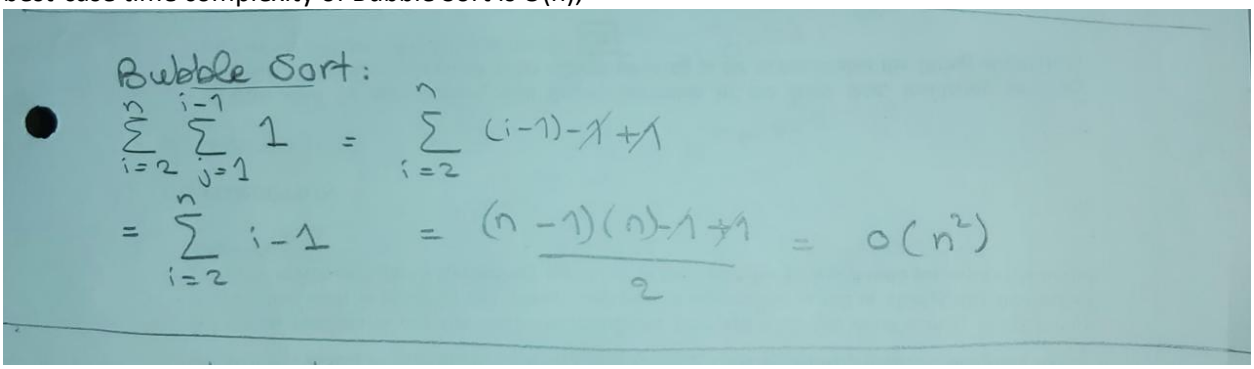
Iteration 3:
Compare the fifth element (7) with the sixth element (0). Since 7 is greater than 0, swap them.
The array becomes [0, 0, 0, 1, 2, 7, 8, 8].
Compare the sixth element (7) with the seventh element (8). Since 7 is smaller than 8, no swap is needed.

Iteration 4:

Compare the seventh element (8) with the eighth element (8). Since they are equal, no swap is needed.

Iteration 5:
No more swaps are needed as the array is already sorted.
After applying the Bubble Sort algorithm, the sorted array is [0, 0, 0, 1, 2, 7, 8, 8]

The worst-case time complexity of Bubble Sort is O(n^2).
best-case time complexity of Bubble Sort is O(n),



Multiplication of large numbers (divide and conquer)

```
function multiply(num1, num2):
   n1 = length(num1)
   n2 = length(num2)
   if n1 == 0 or n2 == 0 do:
      return "0"
   if n1 == 1 and n2 == 1 do:
      return Ans(num1) * Ans(num2)
   else do:
      m = max(n1, n2)
      m2 = m / 2
      high1, low1 = splitNum(num1, m2)
      high2, low2 = splitNum(num2, m2)
      z0 = multiply(low1, low2)
      z1 = multiply((low1 + high1), (low2 + high2))
      z2 = multiply(high1, high2)
      return (z2 * 10^(2*m2)) + ((z1 - z2 - z0) * 10^m2) + z0

function splitNum(num, mid):
   n = length(num)
   if n <= mid do:
      high = "0"
      low = num
   else do:
      high = num[1 : n-mid]
      low = num[n-mid+1 : n]
```

```
    return high, low
answer = Ans
```

Iteration 1:
multiply("2010", "7088")

Iteration 2:
m = max(4, 4) = 4
m2 = 4 / 2 = 2

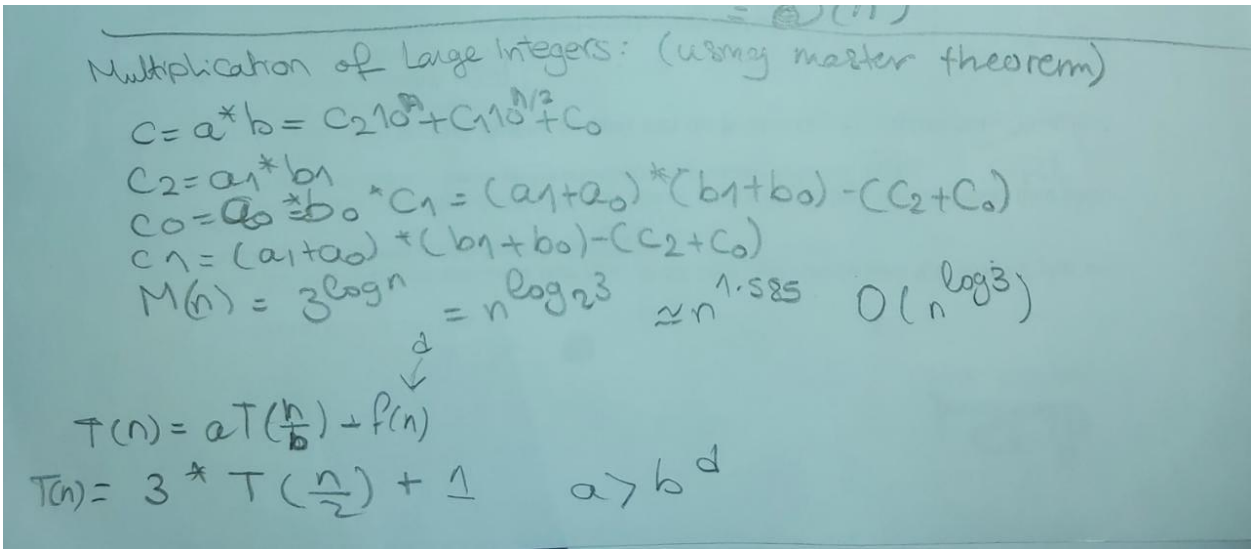
Splitting num1:
high1 = "20"
low1 = "10"

Splitting num2:
high2 = "70"
low2 = "88"

Iteration 3:
z0 = multiply("10", "88") = "880"
z1 = multiply("20" + "10", "70" + "88") = multiply("30", "158") = "4740"
z2 = multiply("20", "70") = "1400"

Iteration 4:
z2 * 10^(2m2) = "1400" * 10^(22) = "140000"
z1 - z2 - z0 = "4740" - "1400" - "880" = "2460"
(z1 - z2 - z0) * 10^m2 = "2460" * 10^2 = "246000"
z0 = "880"

Iteration 5:
Returning the final result:
Result = "140000" + "246000" + "880" = "387880"



best case : O(n^log3)
worst case: O(n^2)