1.)

```
75      def swap(self):
76          if self.Header is None or self.Current.Next is None:
77              return -1
78          else:
79              x = self.Current.Data
80              self.Current.Data = self.Current.Next.Data
81              self.removeCurrentNext()
82              self.insertCurrentNext(x)
83
84              return 0
```

2.)

a.) A Python Dictionary is an Array object.

b.) dic.update({key,value})

c.) dic[“key”] = “val2”

d.) The for loop will be executed for however many key:value pairs are in the dictionary

3.)

```
def inOrderRank(self, num):
    def create_rank_list(self):
        rank_list = []
        def __visit__(n):
            if (n != None):
                __visit__(n.Left)
                rank_list.append(n.Data)
                __visit__(n.Right)

        __visit__(self.Root)
        return rank_list

    x = create_rank_list()

    for i in range(len(x)):
        if x[i] == num:
            return i
    return -1
```

4.)

a.)

```
180  def compare_arrays(A, B):
181      def check(arr1, arr2):
182          ans = []
183          for i in range(len(arr1)):
184              count = 0
185              for j in range(len(arr2)):
186                  if arr1[i] == arr2[j]:
187                      count = count + 1
188              if count == 0:
189                  ans.append(arr1[i])
190          return ans
191
192      C1 = check(A,B)
193      C2 = check(B,A)
194      return C1 + C2
195
196  A = [20, 40, 70, 30, 10, 80, 50, 90, 60]
197  B = [35, 45, 55, 60, 50, 40]
198
199  print(compare_arrays(A,B))
```

b.) C1:

i = 0 and ans = [20]

i = 1 and ans = [20]

i = 2 and ans = [20, 70]

i = 3 and ans = [20, 70, 30]

i = 4 and ans = [20, 70, 30, 10]

i = 5 and ans = [20, 70, 30, 10, 80]

i = 6 and ans = [20, 70, 30, 10, 80]

i = 7 and ans = [20, 70, 30, 10, 80, 90]

i = 8 and ans = [20, 70, 30, 10, 80, 90]

C2:

i = 0 and ans = [35]

i = 1 and ans = [35, 45]

i = 2 and ans = [35, 45, 55]

i = 3 and ans = [35, 45, 55]

i = 4 and ans = [35, 45, 55]

i = 5 and ans = [35, 45, 55]

C1 + C2:

[20, 70, 30, 10, 80, 90, 35, 45, 55]

c.)

180

181

182 c1

183 c2 * n-1

184 c3 * n-1

185 c4 * (m-1)*m/2

186 c5 * (m-1)*m/2

187 c6 * (m-1)*m/2

188 c7 *n-1

189 c8 * n-1

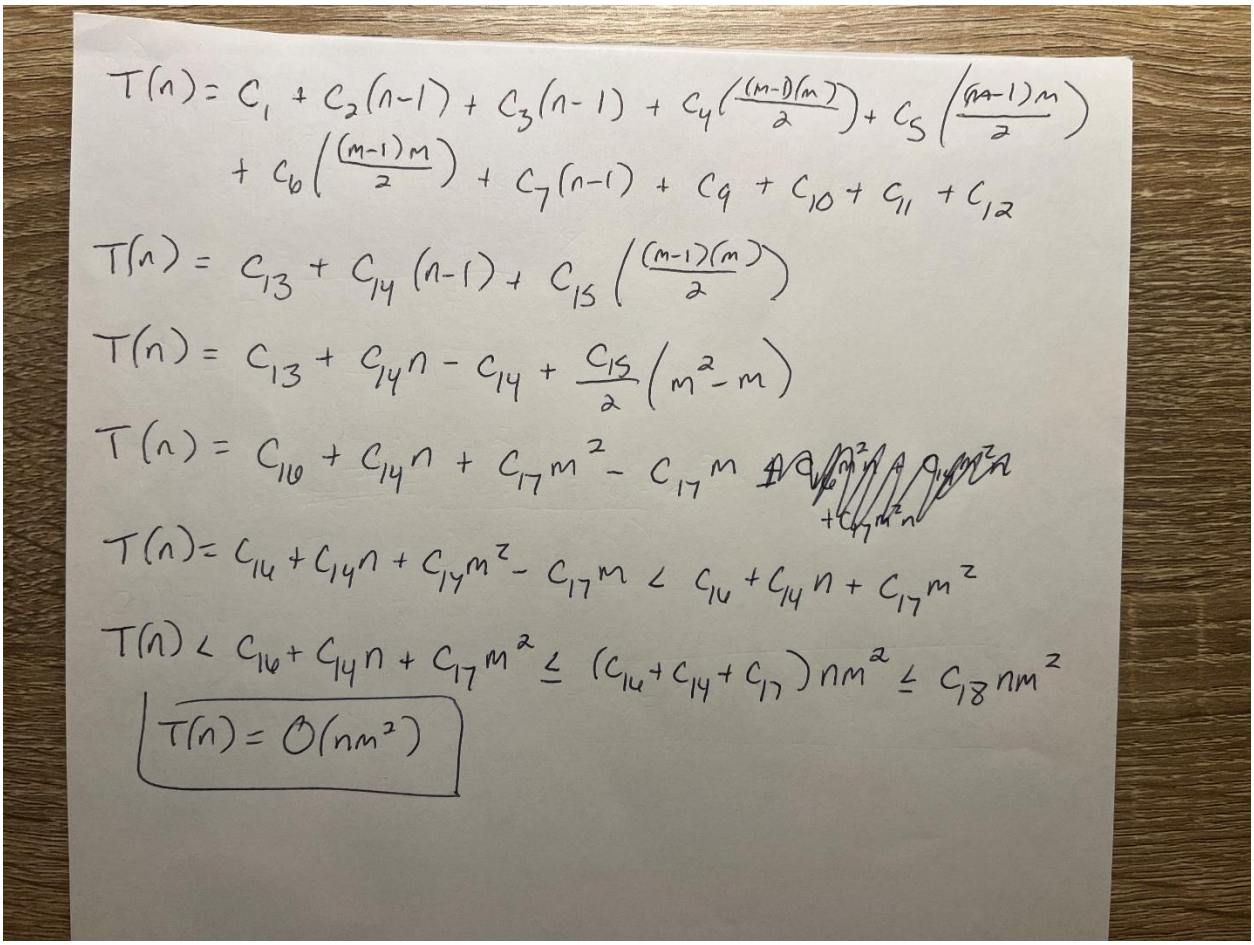
190 c9

191

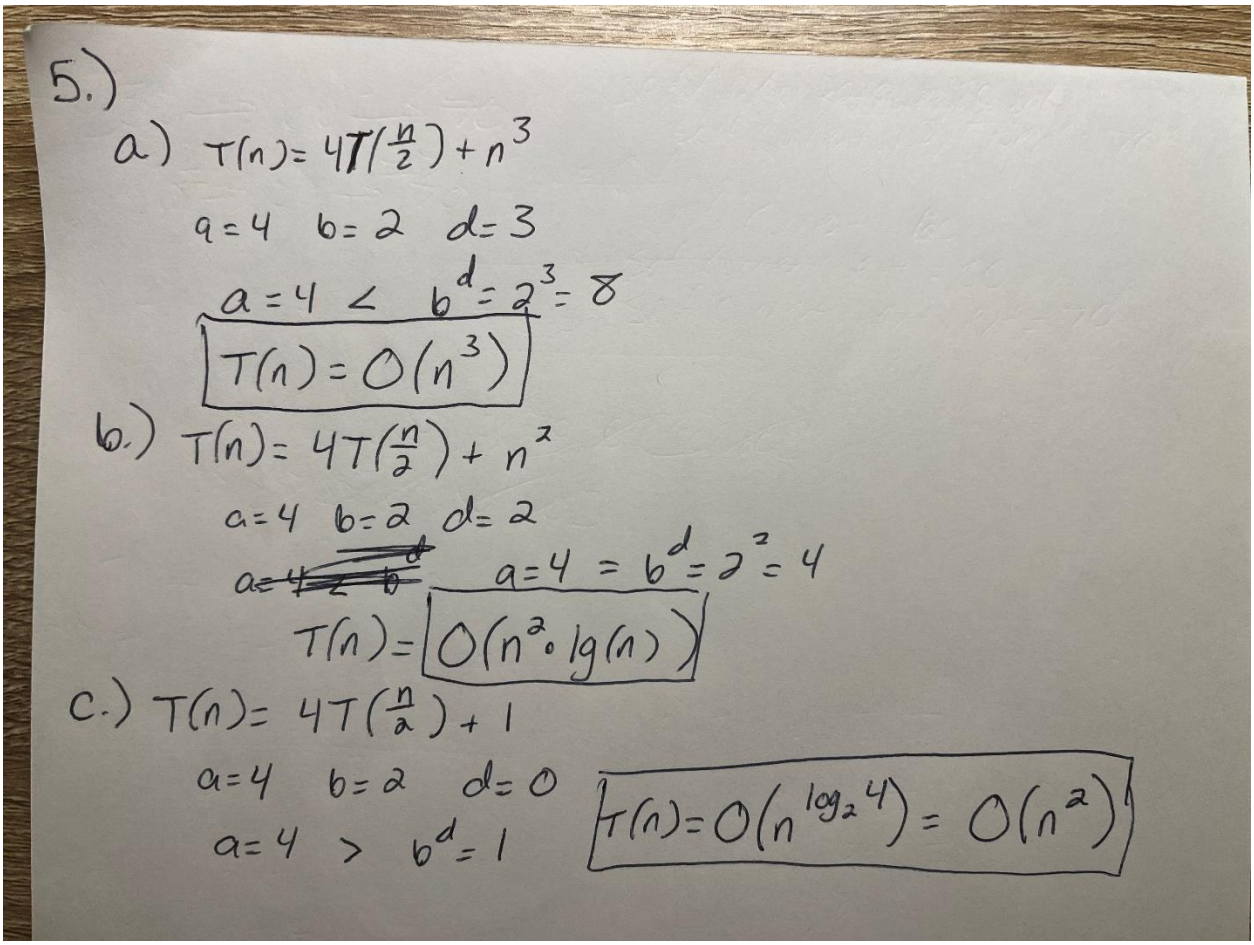
192 c10

193 c11

193 c12


$T(n) = C_1 + C_2(n-1) + C_3(n-1) + C_4\left(\frac{(m-1)(m)}{2}\right) + C_5\left(\frac{(n-1)m}{2}\right)$
$+ C_6\left(\frac{(m-1)m}{2}\right) + C_7(n-1) + C_9 + C_{10} + C_{11} + C_{12}$
$T(n) = C_{13} + C_{14}(n-1) + C_{15}\left(\frac{(m-1)(m)}{2}\right)$
$T(n) = C_{13} + C_{14}n - C_{14} + \frac{C_{15}}{2}(m^2 - m)$
$T(n) = C_{16} + C_{14}n + C_{17}m^2 - C_{17}m$
$T(n) \leq C_{16} + C_{14}n + C_{17}m^2 \leq (C_{16} + C_{14} + C_{17})nm^2 \leq C_{18}nm^2$
$T(n) = O(nm^2)$

5.)


5.)
a) $T(n) = 4T(\frac{n}{2}) + n^3$
$a=4 \quad b=2 \quad d=3$
$a=4 < b^d = 2^3 = 8$
$T(n) = O(n^3)$
b.) $T(n) = 4T(\frac{n}{2}) + n^2$
$a=4 \quad b=2 \quad d=2$
$a=4 = b^d = 2^2 = 4$
$T(n) = O(n^2 \cdot \lg(n))$
c.) $T(n) = 4T(\frac{n}{2}) + 1$
$a=4 \quad b=2 \quad d=0$
$a=4 > b^d = 1$
$T(n) = O(n^{\log_2 4}) = O(n^2)$

6.)

6.) a)

```
graph = [[0,1,0,1,0,0,0]
         [1,0,1,0,0,0,1]
         [1,1,0,0,0,0,0]
         [0,0,0,0,1,1,0]
         [0,0,0,0,0,1,0]
         [0,1,0,0,0,0,0]
         [0,0,1,0,0,1,0] ]
```

b)

Adjacency List

A	BD
B	ACG
C	AB
D	EF
E	F
F	B
G	CF

c.) Queue

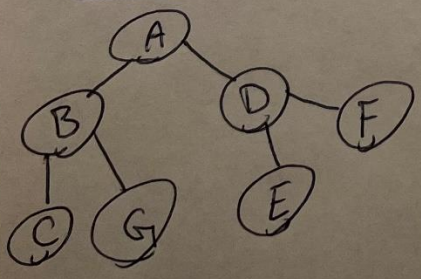
```
[]
[A]
[AB]
[ABD]
[BD]
[BDC]
[BDCG]
[DCG]
[DCGE]
[DCGEF]
```

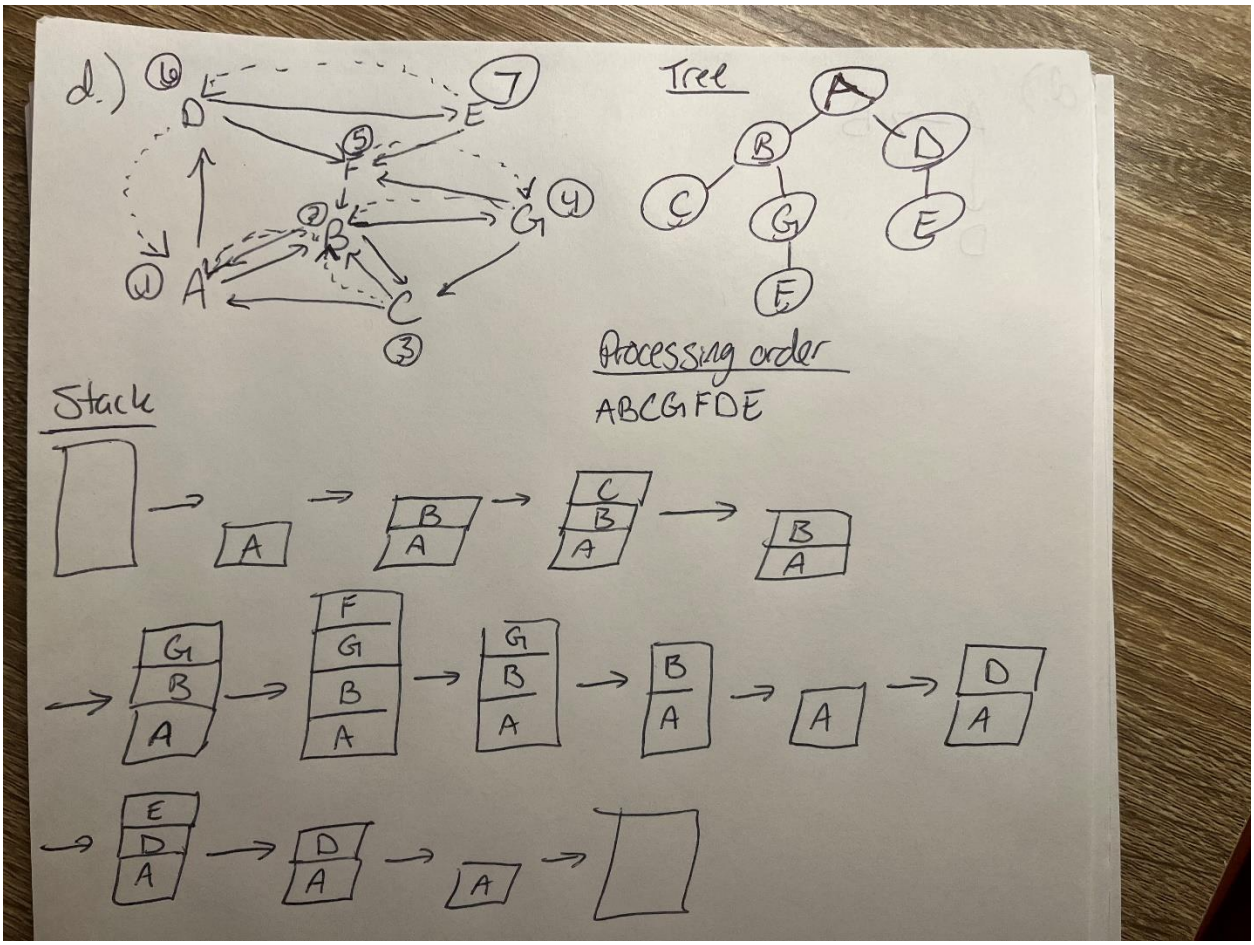
```
[CGEF]
[GEF]
[EF]
[F]
[]
```

Processing List

ABDCGEF

Tree





7.)

a.)

```
205  def find_max(A, right):
206      print("Params: ", A, right)
207      if right == 0:
208          print("Return: ", A[right])
209          return A[right]
210      else:
211          x = find_max(A, right - 1)
212          print(A[right], " compared to ", x)
213          if A[right] > x:
214              print("Return: ", A[right])
215              return A[right]
216          else:
217              print("Return: ", x)
218              return x
219
220  A =  [17, 62, 49, 73, 26, 51]
221  find_max(A, 5)
```

b.)

Params: [17, 62, 49, 73, 26, 51] 5

Params: [17, 62, 49, 73, 26, 51] 4

Params: [17, 62, 49, 73, 26, 51] 3

Params: [17, 62, 49, 73, 26, 51] 2

Params: [17, 62, 49, 73, 26, 51] 1

Params: [17, 62, 49, 73, 26, 51] 0

Return: 17

62 compared to 17

Return: 62

49 compared to 62

Return: 62

73 compared to 62

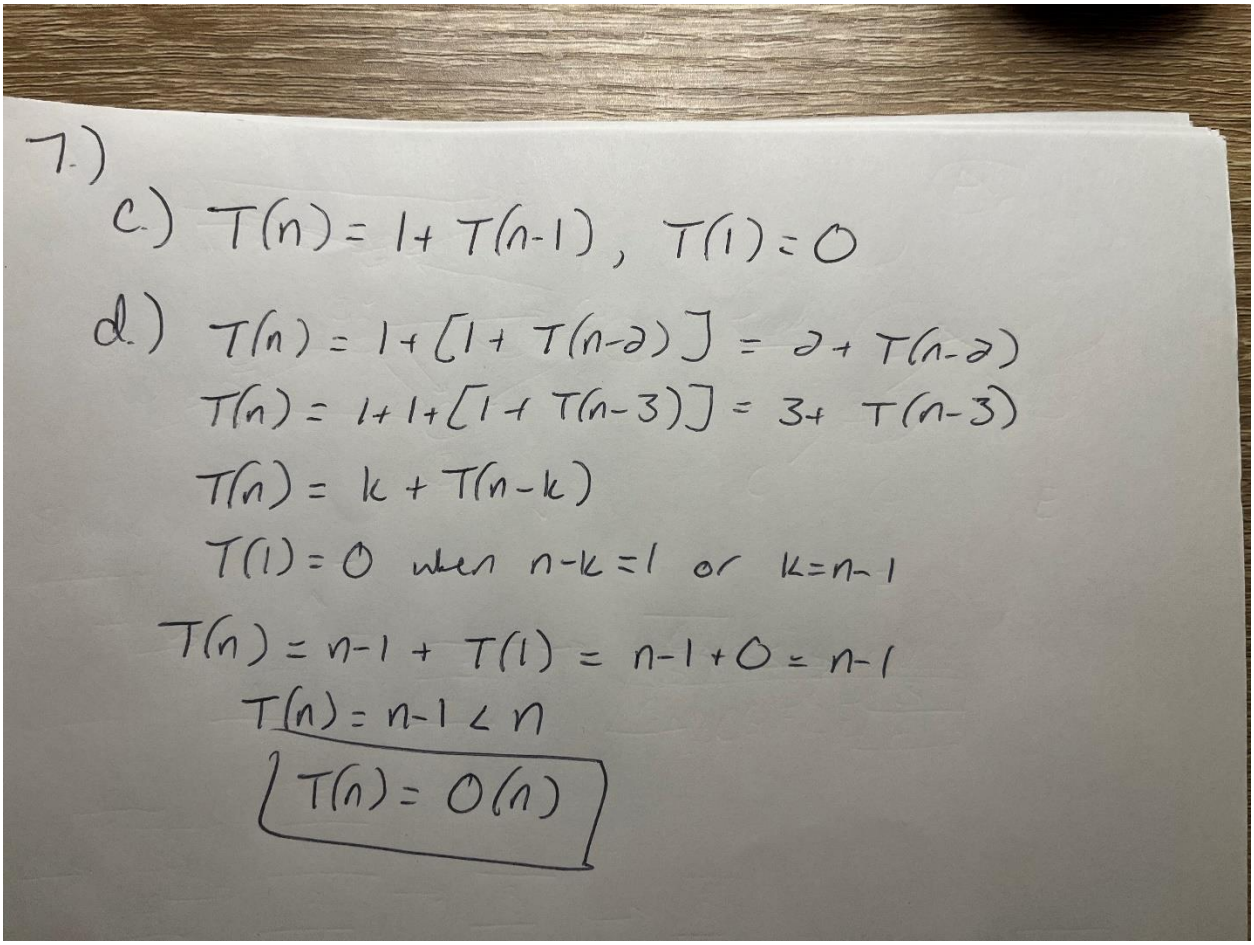
Return: 73

26 compared to 73

Return: 73

51 compared to 73

Return: 73



8.)

a.) MergeSort is Big-Oh class n*lg(n) because the fundamental step of the algorithm is the comparisons in the Merge function. In the worst case the cost of Merge is n-1 and Merge will be called however many times it takes to combine all subarrays into one final array by combining two at a time. QuickSort is in Big-Oh class n^2 because in the worst case when the pivot is always on at the left or right index, the cost of the partition function can be n-1(n)/2.

b.) MergeSort average case is n*lg(n) because no algorithm will do more or less than that amount of work. QuickSort average case is n*lg(n) because the best case is n*lg(n) when the partition algorithm does n amount of work. The worst case for QuickSort is very unlikely, so the average case is best estimated using the best case.

c.) No answer. Ran out of time ☹

d.) No answer. Ran out of time ☹