1. Insertion sort can be expressed as a recursive procedure as follows. In order to sort A[1 .. n], we recursively sort A[1 .. n -1] and then insert A[n] into the sorted array A[1 .. n - 1].
1a) (10 points) Write pseudocode for this algorithm, Insertion sort, as a RECURSIVE procedure.

code:

```
Insertionsort(n,j):

    if j<n.length:

        i=j-1
        while i>=0 and key<n[i]:
            n[i+1]=n[i]
            i=i-1
        n[i+1]=key
        j=j+1
        return Insertionsort(n,j)
    else:
        return n
```

1b) (5 points) Write a recurrence for the worst case running time of this recursive version of insertion sort.

Because the worst case of Insertionsort is reverse order, which means that every two elements need to be compared and inserted opreation

so the worst case is

[n-1,n-2,n-3…..0]

so

The worst case running time is O(n^2)

1c) (8 points) Solve the recurrence equation.

T(n)=T(n-1)+T(n-2)+T(n-3)…+T(1)+Θ(1)

T(n)=T(n(n-1)/2)

T(n)=T($\frac{n^2-n}{2}$)

T(n)=O(n^2)

1d) (2 points) Is your Insertion sort stable? Why or why not?

It’s stable.

Because in the process of sorting, each element is inserted, only the position of the element is changed in order, and so the order of the same element will not be changed

2. (25 points) Each of n keys in an array may have one of the values red, white or blue. Give a linear time algorithm for rearranging the keys so that all the reds come before all the whites and all the whites come before all the blues. The only operations permitted on the keys are
- Examination of a key to find out what color it is.
- A swap (interchange of positions) of two keys specified by their indices.

code:

```
    countingsort_r_w_b(A,k):
        temp=[]
        res=[]

        for i =0 to k+1:
            temp.append(0)

        for j =0 to A.length:
            temp_address=''
            if A[j]=='red':
                temp_address=0
                temp[temp_address]=temp[temp_address]+1
            elif A[j]=='white':
                temp_address=1
                temp[temp_address]=temp[temp_address]+1
            elif A[j]=='blue':
                temp_address=2
                temp[temp_address]=temp[temp_address]+1
            else:
                pass
        for x=0 to temp.length:
             t=temp[x]
            while t>0:
                if x==0:
                    res.append(‘red')
                elif x==1:
                    res.append('white')
                elif x==2:
                    res.append('blue')
                t=t-1
        return res
```

and we can use the Test case:

```
        n=[]
        for x =0 to 10:
                n.append(random.randint(0,2))
        m=[]

        for x in n:
                 if x==0:
                        m.append('red')
                if x==1:
                        m.append('white')
                if x==2:
                        m.append('blue')
        print(m)   # show the list before the sorting
        countingsort_r_w_b(m,2)
        print(m)  # show the list sorted
```

3.(25 points) The operation HEAP-DELETE(*A*, *i*) deletes the item in node *i* from heap *A*. Give an algorithm of HEAP-DELETE that runs in *O*(lg *n*) time for an *n*-element max-heap. Explain in detail why it is *O*(lg *n*).

code:

```
    class heap_sort:


        def max_heapify(self,A,i):
                l=2*i+1
                r=2*i+2

                if l<A.length:
                        if A[l]>A[i]:
                                largest=l
                        else:
                                largest=i
                else:
                        largest=i
                if r<A.length:
                        if A[r]>A[largest]:
                                largest=r
                if largest!=i:
                        temp=A[i]
                        A[i]=A[largest]
                        A[largest]=temp
                        self.max_heapify(A,largest)
                else:
                        pass

        def bulid_max_heap(self,A):
                i=int(A.length/2)-1
                while i>=0:
                        self.max_heapify(A,i)
                        i=i-1
```

```
def HEAPDELETE(self,A,i):
        if i==A.length:
                A.pop()

        else A[i]=A[A.length-1]
                A.pop()
                self.max_heapify(A,i)
                return A
```

and we can use the Test case:

```
n=[]
for x =0 to 10:
        n.append(random.randint(0,10))

clac=heap_sort()
clac.bulid_max_heap(n)
res=clac.HEAPDELETE(n,4)
```

about f(n)=O(lgn) proof:

f2(n)=f(max_heapify)=O(lgn)

T(n)<=T(2n/3)+Θ(1)

beacause a=1 ,b=3/2 ==> $n\log_{3/2}1 = n^0 = 1$

f(n)=Θ($n^{\log_b a}$ **) ==> T(n)=Θ (lgn) ==> T(n)=O(lgn)**

4.(25 points) Given a set of n numbers (a set of arrays and each array comprises n numbers), we wish to find the ith largest number from each array and sort them using the comparison-based algorithm/data structure specified. Devise an algorithm to return an array of the ith largest numbers in sorted order, lowest to highest and use an ith largest order-statistic algorithm to find a theta bound on the worst case running time in terms of n and i.

code:

```
class qucik_sort_set:



    def partition(self,n,p,r):

        random_x=random.randint(p,r-1)
        temp=n[random_x]
        n[random_x]=n[r]
        n[r]=temp

        x=n[r]
        i=p-1
        for j in range(p,r):
            if n[j]<=x:
                i+=1
                temp=n[i]
                n[i]=n[j]
                n[j]=temp

        temp=n[i+1]
        n[i+1]=n[r]
        n[r]=temp
        return i+1

    def quicksort(self,n,p,r):
        if p<r and p<self.end:
            q=self.partition(n,p,r)
            self.quicksort(n,p,q-1)
            self.quicksort(n,q+1,r)

    def main(self,setn,i):
        res=[]
        for n in setn:
            self.end=i
            self.quicksort(n,0,n.length-1)
            res.append(n[i])
        self.end=res.length
        self.quicksort(res,0,n.length-1)
        return res
```