ASSIGNMENT-2: Asymptotic Analysis and Divide and Conquer:
Name: Kalakonda Rohini
Roll number: EE21B066
Question1:
Assuming log^n as log_2^n
Asymptotic runtime is expressed by BIG-O notation
<u>Mathematical definition of Big-O</u>:
T(n) = O(f(n)) if and only if there exists positive constants c and n_0 such that *for all* $n \geq n_0$, $T(n) \leq c \cdot f(n)$
The Big O representations of the given functions are:

1. 4nlogn+2n - O(nlogn)
2. 2^{10} - O(1)
3. 2^{logn} - O(n)
4. $3n + 100logn$-O(n)
5. $4n$ - O(n)
6. 2^n - O(2^n)
7. n^2+10n - O(n^2)
8. n^3 - O(n^3)
9. $nlogn$ - O($nlogn$)

In general functions increase in running time in the following order:
Constant, Linear, $nlogn$, Quadratic, Polynomial, Exponential.
So, the final order of functions by asymptotic growth is :
2<3=4=5<1=9<7<8<6
Question2:

1. f(n) = n-100
   g(n) = n-200
   Claim: f =θ(g)
   To prove: f = O(g), we have to show that f(n)<=cg(n) for all n>=n_0 , where c and n_0 are positive constants.
   300<=n
   300+n<=2n
   n - 100 <= 2n – 400
   n - 100<=2(n-200)
   Therefore, f = O(g). (c=2,n_0=300)
   To prove f =Ω(g), we have to prove that f(n)>=cg(n) for all n>=n_0 , where c and n_0 are positive constants.
   For n_0=$\frac{200}{3}$
   n>=$\frac{200}{3}$
   $\frac{3n}{4}$>= 50
   n>= $\frac{n}{4}$+ 50
   n - 100>=$\frac{n}{4}$ - 50
   n - 100>= $\frac{n-200}{4}$ (where c=0.25)
   Therefore, f = Ω(g).
   Thus, we can conclude that f = θ(g).
2. f(n) = log(2n)

g(n) = log(3n)
Claim: f = θ(g)
To prove f = O(g), we have to prove f(n) <= cg(n) for all n>= n_0, where c and n_0 are positive constants.
Assume: n_0= 2
2<=n
log(2) <= log(n)
log(2) <=2log(3) + log(n)
log(2) + log(n) <=log(n) + 2log(3) + log(n)
log(2) + log(n) <=2(log(n) + log(3))
log(2n) <=2log(3n) (where c=2)
Thus, from the definition of Big-O, f=$O(g)$
To prove f = Ω(g), we have to prove that f(n)>=cg(n) for all n>=n_0 , where c and $n_0$are positive constants.
Assume: n_0= 3
n>=3
log(n) >= log(3)
2log(2) + log(n) >=log(3)
2log(2) + log(n) + log(n) >=log(3) + log(n)
2(log(2) + log(n)) >=log(3) + log(n)
2log(2n)>=log(3n)
log(2n) >=$\frac{\log(3n)}{2}$ (where c=0.5)
Therefore f = Ω(g)
Thus, we can conclude that f = θ(g).

3. f(n) = $n^{0.1}$
g(n) = $(logn)^{10}$
Claim : f = θ(g)
To prove f = Ω(g), we have to prove that f(n)>=cg(n) for all n>= n_0, where c and n_0 are positive constants
Let c = 1 and n_0 = 10.
$\frac{f(10)}{g(10)}$>1
$\frac{f(n)}{g(n)}$ is monotonically increasing because (let n=10^k)
$(10^k)^{0.1}$>=$log^{10}10^k$
$10^{0.1k}$>$log^{10}10^k$
Therefore, we should prove $10^{0.1k}$>k (for all n>10)
All n can be obtained from the base case by LHS*$(10^{0.1})$ and RHS+1
Therefore, we have to prove x*1.23>x+1
x+x*0.23>x+1
x*0.23>1 which is true for n>10
As it is monotonically increasing and greater than 1, f(n)>=cg(n) is true for all n>10.
Therefore, f =Ω(g)
To prove f≠O(g)
we have to prove that there are no possible values of positive constants c and n_0 such that f(n)>=cg(n) for all n>= n_0.
$clogn^{10}$-$n^{0.1}$>=0 for all n>= n_0.

c>=$\frac{n^{0.1}}{logn^{10}}$

i.e., the function h(n)= $\frac{n^{0.1}}{logn^{10}}$ should have an upper bound c

Consider:

$\lim_{n\to+\infty} h(n)=\infty$

This shows that h(n) doesn't have an upper bound. Hence c cannot be an upper bound of h(n), which contradicts our assumption.

Therefore, f≠O(g)

4. f(n)=n2^n

g(n)= 3^n

Claim: f=O(g)

To prove f = O(g), we have to prove f(n) <= cg(n) for all n>= n_0, where c and n_0 are positive constants.

Let us assume c=1 and n_0=2

Clearly,

n2^n<=3^n

We have to prove:

n2^n<=3^n

n<=$(3/2)^n$ for all n>2

Can be proved for any n from the base case of n=2 using induction. This can be done by adding 1 to LHS and multiplying 3/2 to RHS.

So if we can prove x+1<=$\frac{3x}{2}$

x+1<=x+$\frac{x}{2}$

1<=$\frac{x}{2}$

This is true for all x>2

Thus, from the definition of Big-O, f=O(g)

For c=1/3 and n_0=2 , f(n)>=cg(n) for all n>=2

Therefore, f =Ω(g)

Thus, we can conclude that f = θ(g).

Question3:

We could do this using MERGESORT and choose the largest 10 elements from the sorted array. But the time complexity of this would be O(nlogn).

We could also use linear search to find the ten largest elements in the given sequence. For this, we first declare an array (maxarr) of 10 elements initialised to a very small value. Each element in the sequence is compared with the elements of maxarr. If the current element in the sequence happens to be greater than the current element in maxarr (maxarr[i]) , then update happens. maxarr[i+1] takes the value of maxarr[i] and so on till the end of maxarr array is reached. If it happens to be less than all values in maxarr, then no update happens. Thus, the elements of maxarr are updated on each iteration and by the end of all iterations, maxarr will store the largest ten elements of the given sequence in descending order. This algorithm has a time complexity of O(n) since there will be at most 10 comparisons for all elements of the sequence.

Program:

```
def find_largest_ten(arr):
    if len(arr)<10:
        return "Enter atleast ten numbers"
    elif(len(arr)==10):
        return arr
    else:
        maxarr=[-float('inf')]*10
#-float('inf') represents the smallest possible float value

        for j in arr:
            i=0
            while i<10:
                if j>maxarr[i]:
                    temp=maxarr[i]
                    maxarr[i]=j
                    break
                i+=1
# if there was some i < 10 for which j > maxarr[i] due to the break statement,
#we have to update the values of maxarr at all indices > i
# this cycle is repeated till the end of maaxarr is reached
            if i!=10:
                while i<9:
                    temp1=temp
                    temp=maxarr[i+1]
                    maxarr[i+1]=temp1
                    i+=1
        return maxarr

arr = [float(x) for x in input("Enter the elements of the array as space separated values: ").split()]
print("The array of the largest 10 elements (in descending order): ",find_largest_ten(arr))
```

INPUT:
```
Enter the elements of the array as space separated values: 1 2 45
67 8 9 0.34 78 2 67 12  567 89 0 6 9 5 4.3
```
OUTPUT:
```
The array of the largest 10 elements (in descending order):
[567.0, 89.0, 78.0, 67.0, 67.0, 45.0, 12.0, 9.0, 9.0, 8.0]
```

QUESTION4:

We are required to implement binary multiplication using the Divide and Conquer algorithm. Here I have used Karatsuba's algorithm. Since the implementation is to be carried out for binary numbers, we use bitwise shift operators, the right shift operator(>>) and the left shift operator(<<). Doing a right shift by n is equivalent to dividing the number by 2^n and doing a left shift is equivalent to multiplying the number by 2^n.

The base case in the karatsuba function is checking if the numbers are single digit or not. If they are, then the multiplication is done using the multiplication operator. If not, then the function is recurred for both the halves (left and right half) of the number. This splitting happens till there is only a single digit multiplication remaining. After these trivial multiplications are carried out, they are combined. The output of the function is then converted from decimal into binary format using bin().

Program:
```
def karatsuba_mul(x,y):
    maxdig=max(x.bit_length(),y.bit_length())
    if maxdig<=1:
        return x*y
    m2=(maxdig+1)>>1
    a=x>>(m2)
    b=x-(a<<m2)
    c=y>>(m2)
    d=y-(c<<m2)
    k0=karatsuba_mul(b,d)
    k1=karatsuba_mul((a+b),(c+d))
    k2=karatsuba_mul(a,c)
    return (k2<<(maxdig+(maxdig&1)))+((k1-k2-k0)<<(m2))+k0
print("Product in binary:", bin(karatsuba_mul(0b10011011, 0b10111010)))
print("Product in decimal:", karatsuba_mul(0b10011011, 0b10111010))
```

OUTPUT:

```
Product in binary: 0b111000010011110
Product in decimal: 28830
```

QUESTION5:

We are required to describe an algorithm which finds the largest element of a unimodal array in $O(logn)$. In a unimodal array, the largest element is the only element in the entire array that is larger than both its preceding value and its succeeding value.

The algorithm has been implemented using Divide and Conquer Paradigm. In each iteration of the algorithm, the algorithm compares the value at the middle of the array with its preceding and succeeding value. If the preceding value is larger and the succeeding value is smaller, this is the descending part of the array and the largest element is to the left of the middle value. On the other hand, if the preceding value is smaller and the succeeding value is larger, then clearly this is the ascending part of the array and the largest element is to the right of the middle value. In either of these cases, the part of the array containing the largest element is passed on for the next iteration of the function. This is repeated till the largest element is found.

Program :

```
def unimodalsearch(l,h,arr):

  if l==h:

    return arr[l]

  elif h==l+1 and arr[l]<arr[h]:

    return arr[h]

  elif h==l+1 and arr[l]>arr[h]:

    return arr[l]

  mid=(l+h)//2

  if arr[mid]>arr[mid-1] and arr[mid]>arr[mid+1]:

    return arr[mid]
    elif arr[mid]>arr[mid-1] and arr[mid]<arr[mid+1]:

        return unimodalsearch(mid+1,h,arr)

    else:

        return unimodalsearch(l,mid-1,arr)


arr = [float(x) for x in input("Enter the elements of the unimodal array as space separated values: ").split()]

print("The largest element in the unimodal array is :" ,unimodalsearch(0, len(arr)-1, arr))
```

Input :

```
Enter the elements of the unimodal array as space separated values: 34 67
9847 02 95430 76 19 0 76
```

Output :

```
The largest element in the unimodal array is : 95430.0
```

Question 6 :

Towers of Hanoi :

Here there are three rods (A, B, C) and n disks. We need to transfer the n disks from source to destination without disobeying the rules as given in the problem statement. Here, we use a recursive method. The base case of the function is when n=1, and the disk can be directly transferred from source to destination. In other cases, first the n-1 disks are transferred to the auxiliary disk and then to the final destination. Thus there are two recursions inside the function.

Program :

```
def TowerOfHanoi(n , source, destination, auxiliary):

#the base case is when n=1, the disk is moved from source to  destination.

    if n==1:

        print ("Move disk 1 from source",source,"to destination",destination)

        return

#we then recur for n-1 disks. The disks are first placed at auxiliary position from the source.

    TowerOfHanoi(n-1, source, auxiliary, destination)

    print ("Move disk",n,"from source",source,"to destination",destination)

#then we recur for n-1 disks, where the disks are moved from auxiliary to destination.
```

```
    TowerOfHanoi(n-1, auxiliary, destination, source)
```

Example Input :

```
n=3

TowerOfHanoi(n,'A','B','C')
```

Output :

```
Move disk 1 from source A to destination B
Move disk 2 from source A to destination C
Move disk 1 from source B to destination C
Move disk 3 from source A to destination B
Move disk 1 from source C to destination A
Move disk 2 from source C to destination B
Move disk 1 from source A to destination B
```