Further explanation for submitted codes

Questions 1 and 2 were solved in a handwritten manner and are attached in the same submission as a scanned pdf file.

Q3 : Find the 10 largest elements of a given array:

In [1]:
```
#For this question we can use Merge SOrt, but it will be of the order(NlogN)
#Instead, will iterate through the sequence 10 times and take out the maximum element each time
#The order of such an algorithm is 10*N which is O(N) for a large enough N
#Hence it is more efficient than a Merge Sort
#Merge Sort is more efficient in cases where we have to arrange whole array
```

In [2]:
```
#Importing required libraries
import random
import time
import math
```

In [3]:
```
def Ten_Largest_Elements(seq):              #Function defined to find 10 largest elements
    n = len(seq)                            #finding length of sequence to use in for loop
    result = [0]*10                         #Creating empty list of 10 values to store output
    for i in range(10):
        max = 0                             #Setting max to 0 before entering for loop for seq
        for j in range(n-i):
            if seq[j] >= max:
                max = seq[j]                #max is set to value of element if it exceeds previous value of max
                k = j
        result[i]=max
        seq.pop(k)                         #popping the max element from temp so that next time we find max,
                                            #the loop won't give same value as before
    return result
```

In [4]:
```
n = int(input("Please enter an integer value which you want to be the length of sequence:"))
```

```
Please enter an integer value which you want to be the length of sequence:25
```

In [5]:
```
#I will arrange the randomly generated sequence of the length specified by the user
#in decreasing order using Merge_Sort and Merge functions, output is the first 10 elements of the
#sequence arranged in decreasing order

seq=list()
for i in range(n):
    seq.append(random.randint(1,1000))              #Add a random integer in the range 1 to 1000 to this sequence
temp = [x for x in seq]                             #creating temp so pop won't alter original seq
output = Ten_Largest_Elements(temp)                 #Ten_Largest_Elements is compiled in this line
print("Input sequence is:",seq)                     #printing input sequence
print("10 largest elements are:",output)            #printing result that is the 10 largest elements of the sequenc
e
```

```
Input sequence is: [700, 525, 594, 40, 344, 76, 794, 696, 363, 54, 752, 967, 702, 484, 414, 479, 599, 734, 638, 90,
16, 905, 784, 368, 394]
10 largest elements are: [967, 905, 794, 784, 752, 734, 702, 700, 696, 638]
```

I have used a randint generator to generate a random sequence of given length. The assumption in this is that all the entries are integers, but since we are just having to arithmetically compare 2 numbers, them being integers or floats does not make a difference to the function written.

Also as explained in the code, I have iterated through the original sequence 10 times, so time complexity is 10*n which is O(n) by definition. Had I gone about using a Merge Sort algorithm,

its time complexity would have been O(nlogn) which is greater than O(n) and hence iterating through the whole sequence 10 times is a more efficient way to go about the task of finding the ten largest elements.

In general also , if we have to find k largest or smallest elements from an array where k is some constant and not a variable dependant on n, this method will be of order k*n which is again O(n) which is less than O(nlogn) and hence more efficient for a sufficiently large n.

Q4 : Find the multiplication of 2 binary numbers using divide and conquer algorithm.

In [1]:
```
#Binary Product of 2 given inputs using Karatsuba's implementations for efficient time complexity of
#O(n^log3) where log is to the base 2
#I will work in binary numbers, will define binary additon (Full Adder of sorts) and product which will be used
#in the recursion method of Karatsuba
```

In [2]:
```
import math        #Importing required libraries
```

In [3]:
```
def single_digit_product(str1,str2):            #Since input is binary, single digit product can be defined as
    return int(str1[0])*int(str2[0])            #normal product itself
```

In [4]:
```
def match_lengths(str1,str2):                   #Its easier to operate on 2 numbers of equal length while using
    if len(str1)==len(str2):                    #karatsuba's so will add zeros behind the smaller number if that
        return (str1,str2)                      #case comes up

    elif len(str1) < len(str2):
        while(len(str1)!=len(str2)):
            str1 = '0' + str1
        return (str1,str2)

    else:
        while(len(str1)!=len(str2)):
            str2 = '0' + str2
        return (str1,str2)
```

In [5]:
```
def binary_addition(str1,str2):                 #Will choose to operate on strings instead of int form
    result = ""                                 #since we are defining binary add and multiply
    carry = 0                                   #and its easier to add carry's to strings

    str1,str2 = match_lengths(str1,str2)        #Matching lengths of both inputs so addition can be implemented

    for i in range(len(str1)-1,-1,-1):          #len(str)-1 is the lowest bit and since carry may overflow -1 is
        bit_1 = int(str1[i])                    #the highest and addition must be from lowest to highest, iteration
        bit_2 = int(str2[i])                    #is backwards

        sum_bit = (bit_1 ^ bit_2 ^ carry)       #Using bitwise operations to create full adder
        result = str(sum_bit) + result          #storing sum_bit each time to result by appending on left side

        carry = (bit_1 and bit_2) or (bit_1 and carry) or (bit_2 and carry)

    if carry == 1:
        result = '1' + result

    return result
```

In [6]:
```
def Karastuba(str1,str2):
    str1,str2 = match_lengths(str1,str2)             #Matching lengths for easier use of Karatsuba

    if len(str1) == 0:                               #Defining initial conditions for recursion
        return 0
    elif len(str1) == 1:
        return single_digit_product(str1,str2)

    length = int(len(str1))                          #finding length of input to split it for recursion

    L = length//2                                    #left half length
    R = length - L                                   #right half length

    str1_left = str1[:length//2]                     #splitting inputs to use for recursion
    str2_left = str2[:length//2]

    str1_right = str1[length//2:]
    str2_right = str2[length//2:]

    K1 = int(Karastuba(str1_left,str2_left))         #K1 is first recursive use of Karatsuba
    K2 = int(Karastuba(str1_right,str2_right))       #K2 is second recursive use of Karatsuba
    K3 = int(Karastuba(binary_addition(str1_left,str1_right),binary_addition(str2_left,str2_right)))
                                                     #K3 is third recursive use of Karatsuba

    return K1*(1<<(2*R)) + (K3 - K1 - K2)*(1<<R) + K2
```

In [7]:
```
str1 = input("Enter binary number 1 : ")
str2 = input("Enter binary number 2 : ")
answer = Karastuba(str1,str2)                        #Calling Karatsuba function
print("The product of the two numbers in decimal is {0} and in binary is {1} \n".format(answer,bin(answer).replace
("0b","")))
                                                     #Converted decimal to binary to print both outputs
```

```
Enter binary number 1 : 10011011
Enter binary number 2 : 10111010
The product of the two numbers in decimal is 28830 and in binary is 111000010011110
```

Since given numbers are binary, the assumption made is that the inputs for multiplication will be binary.

Since Karatsuba's method is a faster implementation of divide and conquer, I have used that method to find the product. Since the inputs are binary, I have solved the problem by writing a binary adder and binary multiplier and used these in the Kratsuba's implementation.
Time complexity of a Karatsuba's implementation for a product of 2 numbers is O(n^(log3)) where log is to the base 2.

Q5 : Find the maximum element of a unimodal array in O(logn)

In [1]:
```
#Find the maximum element of a unimodal array using an algorithm of O(logn)
#I will go about this by usingan algorithm similar to a binary search tree
#Will start by locating the middle element of the array and check if it is >= its previous element
#If yes then will jump to right by half of n/2 to n, else will jump to left by half 0 to n/2
#n is the length of the array
```

In [2]:
```
print("Enter the elements of unimodal array.\n")
x = input()
```

```
Enter the elements of unimodal array.

2 4.5 6 7.8 9.1 12 15.6 14 12.4 8 5.2
```

In [3]:
```
A = x.split()
n = len(A)
```

In [4]:
```
print("The list is:",A)
print("The number of elements are:",n)
```

```
The list is: ['2', '4.5', '6', '7.8', '9.1', '12', '15.6', '14', '12.4', '8', '5.2']
The number of elements are: 11
```

In [5]:
```
L = 0                                       #dynamic index - left
R = n - 1                                   #dynamic index - right
while R - L > 1:            #while the left and right dynamic indices do not become close enough to pinpoint to sin
gle element
    K = (L+R)//2                           #middle index
    if float(A[K]) >= float(A[K-1]):               #condition for array to be increasing at index K (middle)
        L = K                               #take middle index as new left
    else:
        R = K                               #take middle index as new right
print("Maximum element of this unimodal array is {0} and it is present at position {1}\n".format(float(A[L]), L+1))
```

```
Maximum element of this unimodal array is 15.6 and it is present at position 7
```

Assumption made for this implementation is that all the inputs of the sequence are either float or int type.

Solved using an implementation similar to Binary search trees, we start by searching at the middle index, if the value of that element is greater than its previous element, we will search in the right half since the decreasing of values in the array has not yet started, and if it's less then we will search in the left half because that means the increasing has ended and decreasing of values in unimodal array has started.

Since we are halving the search list every time, if length of the array is n is assumed to be 2^k, then the time complexity will be k since the number of iterations in the while loop are k till we find the maximum element hence making the time complexity O(k) i.e O(logn)

Q6 : Given an array , show how to remove all duplicates in time complexity O(nlogn)

In [1]:
```
#Will write a Merge Sort code and implement it to order the sequence
#Then will iterate through the sorted sequence once to remove all the repeated entries
```

In [2]:
```
def Merge_Sort(seq):
    n=len(seq)
    if n<=1:                            #Defining initial condition for recursion
        return seq
    if n%2==0:
        n1=n//2
    else:
        n1=(n+1)//2
    left = (seq[0:n1])                  #Dividing original array into left half approximately
    right = (seq[n1:n])                 #Dividing original array into right half approximately
    L=Merge_Sort(left)                  #Recursive step on left
    R=Merge_Sort(right)                 #Recursive step on right
    return Merge(L,R)
```

In [3]:
```
def Merge(L,R):
    n1=len(L)
    n2=len(R)
    n = n1+n2
    result = [0]*n              #Creating empty list of total length to store sorted list
    i=0                         #Variable to iterate through L
    j=0                         #Variable to iterate through R
    k=0                         #Variable to iterate through result list

    while i < n1 and j < n2 :       #Merge Step
        if L[i] <= R[j]:            #To put smaller element before larger element
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
        k += 1

    while i < n1:          # Copy the remaining elements of L[], if there are any
        result[k] = L[i]
        i += 1
        k += 1

    while j < n2:          # Copy the remaining elements of R[], if there are any
        result[k] = R[j]
        j += 1
        k += 1
    return result
```

In [4]:
```
X = input("Enter the sequence from which you want to delete duplicates:")
```

```
Enter the sequence from which you want to delete duplicates:-2 3 4.5 8.1 12 10 4.5 3 12
```

In [5]:
```
temp = X.split()
seq = list()                        #Creating empty list to append to it the int for float values from input
for i in temp:                      #Since time complexity of function is O(nlogn), this will not affect run time
    if i.isnumeric():               #much, and since merge sort is written for floats we have to input a seq of
        seq.append(int(i))          #floats or ints or combined
    else:
        seq.append(float(i))
```

In [6]:
```
sorted_seq = Merge_Sort(seq)                        #Applying Merge Sort to the sequence
```

In [7]:
```
deleter = sorted_seq[0]                             #Creating deleter to check if element is repeating
result_seq = list()
result_seq.append(deleter)                          #Since first element will be there, if repeated again will
k = len(sorted_seq)                                 #not append it to list as can be seen from the for loop
for j in range(0,k):
    if j!=0:                                        #Since I have assigned deleter variable to sorted_seq[0]
        if sorted_seq[j] == deleter:                #and don't want to delete that
            continue                                #Not adding already occured element, hence continue
        else:
            deleter = sorted_seq[j]
            result_seq.append(deleter)              #Appending newly occured element
print(seq)                                          #printing original and sorted sequence
print(result_seq)
```

```
[-2.0, 3, 4.5, 8.1, 12, 10, 4.5, 3, 12]
[-2.0, 3, 4.5, 8.1, 10, 12]
```

In [8]:
```
#Faster implementation than in O(nlogn) is possible to get the same result
```

In [9]:
```
#It can be done by typecasting the sequence as a set and back to a list , this will be of O(n)
```

In [10]:
```
seq_set = set(seq)
sort_seq = list(seq_set)
print(seq)
print(sort_seq)
```

```
[-2.0, 3, 4.5, 8.1, 12, 10, 4.5, 3, 12]
[3, 4.5, 8.1, 10, 12, -2.0]
```

This problem can be solved in O(n) by simply typecasting the list as a set and then again typecasting the new set as a list. Since both of these operations are O(n), i.e they iterate through the array only once, it makes time complexity of the complete algorithm to be O(n).

Assumption made while writing this function is that all the entries are floats or ints.

Since the question asks us to solve this in O(nlogn), I have used the Merge Sort to arrange the sequence in an increasing order which is O(nlogn) and after doing this, I have deleted the repeated elements by iterating through the sorted sequence once which is O(n) which makes the time complexity of the full algorithm O(nlogn) since it dominates O(n).

However in any of the methods, it does not maintain the sequence in its original form i.e the ordering of its elements does not remain the same.