EE4371 - Assignment 2

EE19B094

Manvar Nisharg

This PDF contains the solutions to questions from assignment 2 and explanations to the source code submitted along.

Question 1

Order the following functions by asymptotic growth rate.

$a)$	$4nlogn + 2n$	2^{10}	2^{logn}
$b)$	$3n + 100logn$	$4n$	2^n
$c)$	$n^2 + 10n$	n^3	$nlogn$

Answer 1

We know that a function $f(x)$ is said to be $f(x) = O(g(x))$ if $\exists$ a positive real number 'c' s.t. $f(x) \leq cg(x) \ \ \forall\ x \geq x_0$ OR as $x \to \infty$.

Using this, we find the asymptotic functions of all the functions and compare them.

a)

- $f(n) = 4nlogn + 2n = O(nlogn)$
  $Taking,\ c = 6,$
  $4nlogn + 2n < 6 * nlogn$
  $\Rightarrow 2n < 2nlogn$
  $\Rightarrow 1 < logn \quad which\ is\ true\ as\ n \to \infty$

- $f(n) = 2^{10} = O(1)$
  $Taking,\ c = 2^{10} + 1,$
  $2^{10} < (2^{10} + 1) * 1$
  $\Rightarrow 0 < 1 \quad which\ is\ true\ as\ n \to \infty$

- $f(n) = 2^{logn} = O(n^{log2})$
  $Taking,\ c = 1,$
  $n^{log2} = 1 * n^{log2} \quad which\ is\ true\ as\ n \to \infty$

So , we have to compare the functions $nlogn,\ 1,\ n^{log2}$ as n → ∞,

If base of logarithm ∈ (1,2)

$$\Rightarrow log2 > 1$$
$$\Rightarrow n^{log2} > nlogn \qquad as\ n\ \to\ \infty$$

$$\therefore 1\ <\ nlogn\ <\ n^{log2}$$

Therefore, the asymptotic growth rate of functions follow the order

$$2^{10}\ <\ 4nlogn + 2n\ <\ 2^{logn} \qquad (\text{when base of log} \in (1,2))$$

Else if base of algorithm is ≥ 2

$$\Rightarrow log2 \leq 1$$
$$\Rightarrow n^{log2} < nlogn \qquad as\ n\ \to\ \infty$$

$$\therefore 1\ <\ n^{log2}\ <\ nlogn$$

Therefore, the asymptotic growth rate of functions follow the order

$$2^{10}\ <\ 2^{logn}\ <\ 4nlogn + 2n \qquad (\text{when base of log is} \geq 2)$$

b)

- $f(n) = 3n + 100logn = O(n)$
  $Taking,\ c = 103,$
  $3n + 100logn < 103 * n$
  $\Rightarrow logn < n \quad which\ is\ true\ as\ n \to \infty$

- $f(n) = 4n = O(n)$
  $Taking,\ c = 5,$
  $4n < 5 * n \quad which\ is\ true\ as\ n \to \infty$

- $f(n) = 2^n = O(2^n)$
  $Taking\ c = 2,$
  $2^n < 2 * 2^n$
  $\Rightarrow 0 < 2^n \quad which\ is\ true\ as\ n \to \infty$

So, we have to compare the functions $n,\ n,\ 2^n$ as n → ∞

$$n\ =\ n\ <\ 2^n$$

Therefore, the asymptotic growth rate of functions follow the order

$$3n + 100logn\ =\ 4n\ <\ 2^n$$

c)

- $f(x) = n^2 + 10n = O(n^2)$
  $Taking,\ c = 11,$
  $n^2 + 10n < 11 * n^2$
  $\Rightarrow n < n^2 \quad which\ is\ true\ as\ n \to \infty$

- $f(x) = n^3 = O(n^3)$
  $Taking,\ c = 2,$
  $n^3 < 2 * n^3$
  $\Rightarrow 0 < n^3 \quad which\ is\ true\ as\ n \to \infty$

- $f(x) = nlogn = O(nlogn)$
  $Taking,\ c = 2,$
  $nlogn < 2 * nlogn$
  $\Rightarrow 0 < nlogn \quad which\ is\ true\ as\ n \to \infty$

So, we have to compare the function $n^2,\ n^3,\ nlogn$ as n → ∞

$$nlogn\ <\ n^2\ <\ n^3$$

Therefore, the asymptotic growth rate of functions follow the order

$$nlogn\ <\ n^2 + 10n\ <\ n^3$$

Question 2

In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$ or both (in which case $f = \Theta(g)$). Justify your answer.

	$f(x)$	$g(x)$
(a)	$n - 100$	$n - 200$
(b)	$100n + logn$	$n + (logn)^2$
(c)	$log2n$	$log3n$
(d)	n	$nlogn$

Answer 2

We know that, a function $f(x)$ is said to be $f(x) = O(g(x))$ if $\exists$ a positive real number 'c' s.t. $f(x) \leq cg(x)\ \ \forall\ x \geq x_0$ OR as $x \to \infty$.

Also,a function $f(x)$ is said to be $f(x) = \Omega(g(x))$ if $\exists$ a positive real number 'c' s.t. $f(x) \geq cg(x)\ \ \forall\ x \geq x_0$ OR as $x \to \infty$.

a)

$f(n) = n - 100\ and\ g(n) = n - 200$

$Taking\ c = 1\ and\ n_o = 200\ we\ get,$
$n - 100\ >\ 1 * (n - 200)\ for\ all\ n > 200$
$\Rightarrow -100\ >\ -200 \quad which\ is\ true\ for\ given\ conditions.$
$Therefore,\ f = \Omega(g)$

$Similarly\ taking\ c = 2\ and\ n_o = 300\ we\ get,$
$n - 100\ <\ 2 * (n - 200)\ for\ all\ n > 300$
$\Rightarrow 300\ <\ n \quad which\ is\ true\ for\ given\ conditions.$
$Therefore,\ f = O(g)$

$Therefore, f = \Theta(g)$

b)

$f(n) = 100n + logn \; and \; g(n) = n + (logn)^2 \qquad (Assuming \; base \; to \; be \; 2)$

$Taking \; c = 1 \; and \; n_o = 1 \; we \; get,$

$100n + logn \; > \; n + (logn)^2 \;\; for \;\; all \;\; n > 1$

$\Rightarrow 99n \; > \; (logn)^2 - logn$

$As, \; n \; > \; (logn)^2 \; for \; n > 1,$

$\Rightarrow 99n \; > \; (logn)^2 - logn \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = \Omega(g)$

$Similarly, \; taking \; c = 200 \; and \; n_o = 2 \; we \; get,$

$100n + logn \; < \; 200 * (n + (logn)^2) \; for \; all \; n > 2$

$As, \; 100n \; < \; 200n \; and \; logn \; < \; (logn)^2 \; for \; n > 2$

$\Rightarrow 100n \; < \; 200n + 200(logn)^2 - logn \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = O(g)$

$Therefore \; f = \Theta(g)$

c)

$f(n) = log2n \; and \; g(n) = log3n \qquad (Assuming \; base \; to \; be \; 2)$

$Taking \; c = 0.1 \; and \; n_0 = 1 \; we \; get,$

$log(2n) > 0.1 * log(3n) \; for \; all \; n > 1$

$\Rightarrow 0.9 * logn \; > \; 0.1 * log3 - log2$

$\Rightarrow n > 0.533 \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = \Omega(g)$

$Similarly, \; taking \; c = 1.1 \; and \; n_0 = 1 \; we \; get,$

$log(2n) < 1.1 * log(3n) \; for \; all \; n > 1$

$\Rightarrow log2 - 1.1 * log3 \; < \; 0.1 * logn$

$\Rightarrow n \; > \; 0.0058 \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = O(g)$

$Therefore \; f = \Theta(g)$

d)

$f(x) = n^{1.01} \; and \; g(x) = nlog^2(n)$

$Taking \; c \; to \; be \; very \; very \; small \; say \; 0.0001 \; and \; n_0 = 2.582 \times 10^{120} \; we \; get,$

$n^{1.01} > nlog^2(n) \; for \; all \; n > n_0$

$Therefore f = \Omega(g)$

$Similarly, \; solving \; for \; f = O(g) \; we \; find \; for \; any \; value \; of \; c, \; there \; is \; always \; a \; closed \; bound \; for \; n$

$Therefore \; f \neq O(g)$

$Therefore \; f = \Omega(g)$

Question 3

Describe an efficient algorithm for finding the ten largest elements in a sequence of size n. What is the running time of your algorithm?

Answer 3

In [ ]:

```
"""
PURPOSE : Find ten greatest numbers in a sequence of numbers (Q3)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : k (the number of greatest number needed) , sequence of numbers
OUTPUT : k greatest number in the sequence
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
"""


#import header files
import sys



#Class to implement Min Heap Data structure
#Max size of heap is k, which makes all operations constant time as k is constant in our case.
class minHeap:
        def __init__(self,max_heapsize):
                self.currentsize = 0    #Currentsize of heap
                self.maxsize = max_heapsize      #Maxsize of heap
                self.heap = [0] * (max_heapsize + 1)    #Initalize all nodes of heap
                self.heap[0] = -1 * sys.maxsize         #Initalize reference node

        #Function to return index of left child of a node
        get_left_child = lambda self,index : 2*index

        #Function to return index of right child of a node
        get_right_child = lambda self,index : (2*index)+1

        #Function to return index of parent of a node
        get_parent = lambda self,index : index//2

        #Function that returns if given index node is a leaf
        is_leaf = lambda self,index : index > self.currentsize//2 and index <= self.currentsize

        #Function to swap two nodes
        def swap_nodes(self,index1,index2):
                self.heap[index1] , self.heap[index2] = self.heap[index2] , self.heap[index1]

        #Recursive function to heapify a node
        def minHeapify(self,index):
                if self.is_leaf(index):
                        return

                #Check if any child has smaller value than the node
                if self.heap[index] > self.heap[self.get_left_child(index)] or self.heap[index] > self.heap[self.get_right_child(index)]:

                        #Then swap the appropriate child and call the function again
```

```
                if self.heap[self.get_left_child(index)] < self.heap[self.get_right_child(index)]:
                        self.swap_nodes(index,self.get_left_child(index))
                        self.minHeapify(self.get_left_child(index))

                else:
                        self.swap_nodes(index,self.get_right_child(index))
                        self.minHeapify(self.get_right_child(index))

        #Heapify all the nodes
        def minHeap_whole(self):
                for index in range (self.currentsize//2-1,0,-1):
                        self.minHeapify(index)

        #Function to insert node in the tree at right index
        def insert_node(self,value):

                #Increase currentsize by 1
                self.currentsize += 1

                #Assign the value to the last node
                self.heap[self.currentsize] = value

                #Swap nodes till we satisfy the minheap property
                current_node = self.currentsize

                while self.heap[current_node] < self.heap[self.get_parent(current_node)]:
                        self.swap_nodes(current_node, self.get_parent(current_node))
                        current_node = self.get_parent(current_node)

        #Function to delete minimum element (i.e. head) of the tree and heapify the tree
        def delete_min(self):
                value = self.heap[1]
                self.heap[1] = self.heap[self.currentsize]      #Swap smallest number to the last node
                self.currentsize -= 1   #Reduce size of heap
                self.minHeapify(1)      #Heapify the tree
                return value

        #Printing the tree in sorted order
        def print_Heap(self):
                while(self.currentsize!=0):
                        print(self.delete_min(),end=" ")



#Call function to implement the algorithm
def ten_greatest_number():
        print()
        print("To find k greatest numbers in a sequence of n numbers")
        print()

        k = 0
        while(k <= 0):  #Taking k as input from user untill valid input is not received
                k = int(input("Enter k (>0) : "))
                print()


        sequence = input("Enter sequence of numbers seperated by space : ")     #Storing sequence of numbers as string (sequence)
        list_of_numbers = sequence.split()      #Splitting sequence string into individual numbers with space as delimitter in a list (list_of_numbers)


        Heap = minHeap(len(list_of_numbers)+1) #Initalize min-heap

        #Add first k numbers into the min-heap as base case
        for number in list_of_numbers[0:k]:
                Heap.insert_node(float(number))

        #Iterate through rest of the numbers and if it is bigger than the smallest number in the heap then add it into the heap and remove the smallest number
        for a in range(k,len(list_of_numbers)):
                if(Heap.heap[1]<float(list_of_numbers[a])):
                        Heap.delete_min()
                        Heap.insert_node(float(list_of_numbers[a]))


        #Printing the result
        if len(list_of_numbers) < k:
                print("\nThe %d (as input had size less than k (%d) ) greatest numbers are :"%(len(list_of_numbers),k),end=" ")
        else:
                print("\nThe %d greatest numbers are :"%k,end=" ")
        Heap.print_Heap()


#Calling the function
ten_greatest_number()
```

Outputs for some of the test-cases

- ```
  To find k greatest numbers in a sequence of n numbers

  Enter k (>0) : 10

  Enter sequence of numbers seperated by space : 12 54 83 05 38 18 75 29 47 28 47 99 102

  The 10 greatest numbers are : 28.0 29.0 38.0 47.0 47.0 54.0 75.0 83.0 99.0 102.0
  ```

- ```
  To find k greatest numbers in a sequence of n numbers

  Enter k (>0) : 10

  Enter sequence of numbers seperated by space : 10.5 23.9 61.0 23.1 42.3

  The 5 (as input had size less than k (10) ) greatest numbers are : 10.5 23.1 23.9 42.3 61.0
  ```

Algorithm

We want an algorithm such that we can get k maximum values from a sequence of n numbers.
Assumption : All the elements are float numbers.

$\underline{Data - Structure\ used}$:

The algorithm is implemented using the ‘Min - Heap’ data structure.
Min - Heap is a special case of binary tree data structure wherein the two children of a node are always greater than it. Therefore the smallest value in the heap is always at the root of the tree.
Therefore we can access the smallest element in $O(1)$ time.

$Other\ operations:$

1) $insert\_node()$: We add the new node initially at the leftmost open end of the heap (i.e. leaf in a tree) and swap it with it's parent if, it's value is smaller than it's parent so as to rectify the violated min-heap property and, traverse this upwards towards the root till we find a parent whose value is smaller than it.

As in the worst case, we traverse the tree from a leaf to it's root $insert\_node()$ is implemented in $O(logn)$ time

2) $delete\_min()$: To delete the minimum value (i.e. always the head) from the heap, we first swap the last element of the heap with the head and reduce the current size of the heap by one thus effectively cutting the last element out of the heap. Then, we heapify the head of the tree i.e starting from the head we check if any of it's children has value smaller than it, if yes swap with the respective child and continue this process till we find a position where both it's children has values greater than it or if we reach the end of tree.

As at worst case we traverse the tree from root to the end of the tree (i.e. till a leaf) $delete\_min()$ is implemented in $O(logn)$ time.

3) $print\_heap()$: This function prints the values in the heap in sorted order. It deletes the smallest value while printing it till the heap is empty.

As $delete\_min()$ takes $O(logn)$ time this will take $O(nlogn)$ time.

$Implementation:$

Firstly we add k elements into the empty heap one by one.
Now as inserting a node in a heap takes $O(logn)$ time.
This will take less than $k * O(logk)$ time to implement for sure.
Now moving forward, at any moment of time, we will have the k greatest numbers from the numbers we have iterated so far in the heap thus, from now on we keep updating the heap for every number we iterate.

Then for the remaining (n-k) elements, we iterate through them and, if the value of current number is greater than the minimum number in the heap then first we delete the minimum number from the heap and then add the current number into the heap. Thus, after every iteration we would have the k greatest numbers in the heap.

Now as, finding minimum element in the heap takes $O(1)$ time, while deleting the minimum node and adding nodes takes $O(logn)$ time,
This will take,

$$\sum_{i\ =\ n-k}^{n} (2 * O(logk) + O(1)) = (n-k) * O(logk)$$

Finally, printing the heap in sorted order takes $O(klogk)$ time

Therefore implementing the whole algorithm takes

$$\begin{aligned} & k * O(logk) + (n-k) * O(logk) + O(klogk) \\ & = O(n * logk)\ overall\ time. \end{aligned} \qquad (\text{as } k \leq n)$$

$$However,\ as\ in\ our\ case\ k\ is\ a\ constant\ i.e.\ 10,\ the\ time\ complexity\ will\ be\ \underline{O(n)}$$

Also, as we have kept max size of the heap as $k+1$,

$$so,\ the\ space\ complexity\ of\ the\ algorithm\ is\ O(k)$$
$$but\ again\ as\ in\ our\ case\ k\ is\ a\ constant\ i.e.\ 10,\ it\ will\ be\ \underline{O(1)}$$

Question 4

Use the divide and conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

Answer 4

In [ ]:
```
"""
PURPOSE : Multiply two binary numbers using 'Divide and Conquer' method (Karatsuba algorithm)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Two binary numbers
OUTPUT : Multiplication of the numbers in binary and decimal base
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
"""

#Importing header files
import math


"""
Function to make length of two binary strings same by adding zeros at the beginning of shorter string
:string1 param: First binary string
:param type: string
:string2 param: Second binary string
:param type: string
:return type: tuple of strings in same order as parameters
"""
def make_length_equal(string1 , string2):
        #Base case
        if len(string1) == len(string2):
                pass

        #Add 0s at the beggining of short string
        elif len(string1) < len(string2):
                string1 = '0' * (len(string2) - len(string1)) + string1

        else:
                string2 = '0' * (len(string1) - len(string2)) + string2

        return (string1 , string2)


"""
Function to multiply two single bit numbers
i.e. multiply LSBs of the two strings provided
:string1 param: First number string
:param type: string
:string2 param: Second number string
:param type: string
:return type: int
"""
def multiply_single_digit(string1 , string2):
        return int(string1[0]) * int(string2[0])


"""
```

```
"""
Function to add two binary strings bit by bit
:string1 param: First binary string
:string1 type: string
:string2 param: Second binary string
:string2 type: string
:return type: string
"""
def add_numbers(string1 , string2):
        result = ""
        carry = 0

        #make length of both strings same
        string1 , string2 = make_length_equal(string1 , string2)
        length = len(string1)

        #Loop through bits and add them
        for i in range(length-1 , -1 , -1):
                bit_1 = int(string1[i])
                bit_2 = int(string2[i])

                sum_bit = (bit_1 ^ bit_2 ^ carry)
                result = str(sum_bit) + result

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

        #Add bit at MSB if carry at the end of addition is 1
        if carry == 1:
                result = '1' + result

        return result


"""
Recursive function to implement Karatsuba algorithm for two binary strings
:string1 param: First binary string
:string1 type: string
:string2 param: Second binary string
:string2 type: string
:return type: int
"""
def Karatsuba(string1 , string2):
        #Make length of strings equal
        string1 , string2 = make_length_equal(string1 , string2)

        #Base Cases
        if len(string1) == 0:
                return 0
        elif len(string1) == 1:
                return multiply_single_digit(string1 , string2)


        length = int(len(string1))
        second_half = math.ceil(length/2)

        #Divide the strings into left and right half
        string1_left = string1[:math.floor(length/2)]
        string2_left = string2[:math.floor(length/2)]

        string1_right = string1[math.floor(length/2):]
        string2_right = string2[math.floor(length/2):]

        #Calculating three results required to implement the algorithm
        result1 = int(Karatsuba(string1_left , string2_left))
        result2 = int(Karatsuba(string1_right , string2_right))
        result3 = int(Karatsuba(add_numbers(string1_left , string1_right) , add_numbers(string2_left , string2_right)))

        #Caluclate and return final product
        return result1*(1<<(2*second_half)) + (result3 - result1 - result2)*(1<<second_half) + result2


"""
Call function to implement Karatsuba algorithm
:input: Two binary numbers
:output: Multiplication of the numbers in binary and decimal base
"""
def call_for_Karatsuba():
        print()
        print("To multiply two binary numbers using Karatsuba algorithm :")
        string1 = input("Enter binary number 1 : ")
        string2 = input("Enter binary number 2 : ")

        if(len(string1)==0 or len(string2)==0):
                print()
                print("Atleast one of the numbers were NULL, please enter valid input.")
                return

        answer = Karatsuba(string1 , string2)

        print("\nThe product of the two numbers in decimal is %d and in binary is %s \n"%(answer , bin(answer).replace("0b","")))


#Call function
call_for_Karatsuba()
```

Output for some of the test-case

- ```
  To multiply two binary numbers using Karatsuba algorithm :
  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
  ```

Algorithm

We have to implement Karastuba algorithm
Assumption : Given binary numbers are whole numbers therefore no sign bit.

$\underline{Karatsuba\ Algorithm}$

Let X and Y be two binary strings of length n. Then we can break them into two parts as,

$$X = X_1 * 2^{n/2} + X_2 \qquad (\text{Where X\_1 and X\_2 contain the leftmost and rightmost n/2 bits of X})$$

$$Y = Y_1 * 2^{n/2} + Y_2 \qquad (\text{Where Y\_1 and Y\_2 contain the leftmost and rightmost n/2 bits of Y})$$

Now, we can write product XY as,

$$XY = 2^n\ X_1Y_1 + 2^{n/2} * [(X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2] + X_2Y_2$$
  ```

Now, we can calculate X_1Y_1 and X_2Y_2 by again using Karastuba algorithm using recursion.
This, causes a recursion tree which follows the equation,

$$T(n) = 3T(n/2) + O(n)$$

and it's solutions comes out to be $O(n^{\log_2 3}) \approx \underline{O(n^{1.59})}$ using master theorem for divide-and-conquer recurrences.

$Functions\ and\ Implementation$

1) $make\_length\_equal()$: This function takes two binary strings as input and makes the length of shorter string equal to the longer string by adding appropriate number of '0's at the beginning of the shorter string.
This takes $O(n)$ time where n is the length of longer string because it just includes appending two strings.

2) $multiply\_single\_digit()$: Function to multiply two single digit numbers.
Therefore $O(1)$ time.

3) $add\_numbers()$: Function to add two binary strings. We add the two strings bit by bit using the formulas,

$$sum\_bit = bit\_1 \oplus bit\_2 \oplus carry$$
$$carry = (bit\_1\ \&\ bit\_2) \mid (bit\_1\ \&\ carry) \mid (bit\_2\ \&\ carry)$$

Therefore this will take $O(n)$ time where ns is the length of the binary string.

4) $Karatsuba()$: Function to implement the karatsuba algorithm. Firstly, make the length of both the strings equal, then divide the strings into two halves and calculate the three multiplications using recursion and use them to find the desired product.
As seen above this will take $O(n^{\log_2 3}) \approx \underline{O(n^{1.59})}$ to execute where n is the length of the longer string.

Question 5

You are given a unimodal array of n distinct elements, meaning that its entries are in increasing order up until its maximum elements, after which its elements are in decreasing order. Give an algorithm to compute the maximum element of a unimodal array that runs in O(log n) time.

Answer 5

In [ ]:
```
"""*****************************************************************************************************************
PURPOSE : To find the maximum element in a unimodal sequence of n distinct numbers (Assign-2 Q5)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Unimodal sequence of n distinct numbers
OUTPUT : Maximum element in the sequence
NOTE : Please use python 3.x as several functions used are only compatible with python 3 or later versions
*****************************************************************************************************************"""

"""
Modified Binary function to find the maximum number in a unimodal sequence of distinct numbers

:param sequence: List of integers
:type sequence: List

:return: Max element of the sequence
:return type: int

:input: Null
:output: Null
"""
def peak_finding(sequence):
        #Initalize required variables
        list_size = len(sequence)
        start = 0       #Index of first element of active search window
        end = list_size - 1     #Index of last element of active search window

    #Applying loop of modified binary search
        while(start <= end):
                if start==end:  #If start = end return number at start/end
                        return sequence[start]

                mid = int((start+end)/2)        #Index of middle element of active search window

                # If mid element is neither at begining or at the end of the sequence,
                # compare it with adjacent numbers and take appropriate steps
                if(mid != 0 and mid != list_size-1):
                        if(sequence[mid]>sequence[mid+1] and sequence[mid]>sequence[mid-1]): #Part 2 of list (Refer explanation)
                                return sequence[mid]

                        elif(sequence[mid]>sequence[mid+1] and sequence[mid]<sequence[mid-1]): #Part 3 of list
                                end = mid-1

                        else: #Part 1 of list
                                start = mid+1

                # Else if mid element is at the beginning of the sequence, compare with
                # it's right element and take appropriate steps
                elif(mid == 0):
                        if(sequence[mid]>sequence[mid+1]): #Part 2 of list
                                return sequence[mid]
                        else:  #Part 1 of
                                start = mid+1

                # Else if mid element is at the end of the sequence, compare with
                # it's left element and take appropriate steps
                else:
                        if(sequence[mid]>sequence[mid-1]):
                                return sequence[mid]
                        else:
                                end = mid - 1


"""
Initial call function to find the maximum element in a unimodal sequence of n distinct numbers

:return: Null
:input: Unimodal sequence of distinct numbers
:output: Max number in the sequence
"""
def max_in_unimodal_array():
        print()
        print("To find the maximum element in a unimodal array of n distinct numbers : ")

        #Asking for sequence of numbers from user till valid input is found
        sequence = input("Enter sequence of numbers seperated by space : ")
        list_of_numbers = sequence.split()      #Splitting sequence of numbers into inidividual numbers and storing in a list with space as delimiter

        if(len(list_of_numbers)==0):
                print()
                print("No numbers were input")
                return

        #Converting list of numbers stored as strings to float data type
```

```
        list_of_numbers = list(map(float, list_of_numbers))

        print()
        print("The maximum element in the unimodal array is : %f"%peak_finding(list_of_numbers)) #Calling modified binary search function and printing result
        print()

#Calling the initial call function
max_in_unimodal_array()
```

Output for some test-case

- ```
  To find the maximum element in a unimodal array of n distinct numbers :
  Enter sequence of numbers seperated by space : 1.2 3.4 4.9 5.0 6.2 2.2 -5.7

  The maximum element in the unimodal array is : 6.200000
  ```

- ```
  To find the maximum element in a unimodal array of n distinct numbers :
  Enter sequence of numbers seperated by space : 1 2 3 4 5 6 7

  The maximum element in the unimodal array is : 7.000000
  ```

- ```
  To find the maximum element in a unimodal array of n distinct numbers :
  Enter sequence of numbers seperated by space : 1

  The maximum element in the unimodal array is : 1.000000
  ```

Algorithm

The algorithm implemented is a modified version of Binary Search algorithm.
Assumption : All the elements are floating point numbers.

As the code contains only one function therefore, the algorithm and the function have the same explanation.

$\underline{peak\_finding()}$: This function takes the list of unimodal numbers as input. Now as in Binary Search we reduce the size of active search window by half in our case too but the conditions will be different. As usual let $start$ and end denote the starting and ending index of the active window of search and the index at mid of the active search window be the number under consideration. For ex. let the sequence of numbers be,

$$list = \quad 1.2 \quad 3.4 \quad 4.9 \quad 5.0 \quad 6.2 \quad 2.2 \quad -5.7$$

Now, we can see that we can divide the list into three parts.

$$\{1.2 \quad 3.4 \quad 4.9 \quad 5.0\} \quad \{6.2\} \quad \{2.2 \quad -5.7\}$$

Where the first part contains the numbers which are in the increasing order. Third part contains the numbers which are in the decreasing order. And the second part contains the highest number (i.e. the number to be found).

1. Now at any point of time if we are currently at an element in the first part then, we need to move to the right therefore move the $start$ value to $mid + 1$. We check if we are in the first part if the left neighbour is smaller and right neighbiour is greater than the current number.
2. Similarly, if at any point of time we are in the third part then, we need to move to the left therefore move the $start$ value to $mid + 1$. We check if we are in the right part if the left neighbour is greater and right neighbour is smaller than the current number.
3. And in case if are in the second part, we found the number we wanted and thus return the number.

If we go through the example till the end.

$$\begin{array}{cccccccc} 1.2 & 3.4 & 4.9 & 5.0 & 6.2 & 2.2 & -5.7 & (4.9 < 5.0 < 6.2) \\ \uparrow & & & \uparrow & & & \uparrow & \\ start & & & mid & & & end & \end{array}$$

As we can see mid is in part 1. So we move start.

$$\begin{array}{cccccccc} 1.2 & 3.4 & 4.9 & 5.0 & 6.2 & 2.2 & -5.7 & (6.2 > 2.2 > \text{-}5.7) \\ & & & & \uparrow & \uparrow & \uparrow & \\ & & & & start & mid & end & \end{array}$$

Now as mid is in part 3. So we move end.

$$\begin{array}{cccccccc} .2 & 3.4 & 4.9 & 5.0 & 6.2 & 2.2 & -5.7 & (5.0 < 6.2 > 2.2) \\ & & & & \uparrow & & & \\ & & & & start, mid, end & & & \end{array}$$

As mid is in part 2 means we found the number we wanted so we return the number.

Just like binary search, we reduce the size of our active search window by half so at max we do $O(logn)$ operations.

$$So\ the\ overall\ time\ complexity\ of\ the\ algorithm\ is\ \underline{O(logn)}$$

.

Also as no extra memory has been used which is dependent on size of input,

$$so\ the\ overall\ space\ complexity\ of\ the\ algorithm\ is\ \underline{O(1)}$$

Question 6

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in array. Show how to remove all duplicates from the array in time O(n log n).

Answer 6
  ```

NOTE: This question can be implemented in O(n) time using python dictionaries as it uses hashing but, as the question specifically asks for O(nlogn) algorithm, it has been implemented in O(nlogn).

In [ ]:

```
"""**************************************************************************************************
PURPOSE : Remove duplicates from a sequence of numbers (Assign-2 Q6)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Sequence of numbers
OUTPUT : Sequence of numbers with duplicates removed in sorted and original form
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
**************************************************************************************************"""

from array import array #Importing necessary files

"""
Function to merge two sorted subarrays into one

:param arr: The array whose two subarrays are to be merged
:param type: list
:param start: The starting index in arr from where elements are to be merged(i.e. starting index of first subarray)
:param type: int
:param mid: The midpoint index in arr (i.e. index where two subarrays are to be divided)
:param type: int
:param end: The endpoint index in arr till where elements are to be merged(i.e. ending index of second subarray)
:param type: int
"""
def merge(arr, start, mid, end):
    #size of arrays to be merged
    size_left = mid - start + 1
    size_right = end - mid


    ##Initalize temporary arrays and copy data from original arrays to temporary arrays
    temp_left = arr[start : start + size_left]
    temp_right = arr[start + size_left : end+1]


    #Pointers to current elements in all three arays
    i = 0     # Initial index of first temporary array
    j = 0     # Initial index of second temporaary array
    k = start     # Initial index of merged array

    #Comparing elements and adding them into the original array until end of any one array is reached
    while i < size_left and j < size_right :
        if temp_left[i] <= temp_right[j]:
            arr[k] = temp_left[i]
            i += 1
        else:
            arr[k] = temp_right[j]
            j += 1
        k += 1

    # Copy the remaining elements of temp_left array if any, into the original array
    while i < size_left:
        arr[k] = temp_left[i]
        i += 1
        k += 1

    # Copy the remaining elements of temp_right array if any, into the original array
    while j < size_right:
        arr[k] = temp_right[j]
        j += 1
        k += 1


"""
Function to sort a array using merger sort algorithm

:param arr: The array which is to be merged
:param type: list
:param start: The index of first element to be sorted
:param type: int
:param end: The index of last element to be sorted
:param type: int
"""
def mergeSort(arr,start,end):
    if start >= end:
        return

    mid = (start+(end-1))//2

    # Dividing array into two halves and sorting both the halves first
    mergeSort(arr, start, mid)
    mergeSort(arr, mid+1, end)
    # Merging the already sorted arrays into one
    merge(arr, start, mid, end)



"""
Function to return index of a number in a sorted 2D array using binary search algorithm (w.r.t. first coloumn)

:param arr: The arr in which number is to be searched
:param type: list
:param number: The number to be searched
:param type: float

:return: index of number in array
"""
def binary_search(arr,number):
    #Base Cases
    if(len(arr)==0):
        return
    elif(len(arr)==1):
        return 0

    #Initalize pointers to beginning and end of array
    start = 0
    end = int(len(arr)-1)

    while start <= end:

        mid = int((start + end)/2)  #Middle index of active window of search

        if(arr[mid][0] == number):     #If number at mid is equal to 'number' then return mid
            return mid
        elif(arr[mid][0] > number):  #Else if number at mid is greater than 'number' then change end pointer to mid-1
            end = mid - 1
        else:    #Else if number at mid is less than 'number' then change start pointer to mid+1
            start = mid + 1

    return -1   #If number is not found return -1


"""
Call function to remove duplicates from a sequence of numbers

:input: Sequence of numbers
:output: Sequence of numbers with duplicates removed in sorted and original form
"""
def remove_duplicates():
    print("\nTo remove duplicates from a sequence of numbers :")

    sequence = input("Enter sequence of numbers seperated by space : ")     #Storing sequence of numbers as string (sequence)
    list_of_numbers_original = sequence.split()  #Splitting sequence string into individual numbers with space as delimitter in a list (list_of_numbers_original)
```

```
    if len(list_of_numbers_original)==0:
        print()
        print("No sequence was given as input.")
        return

    list_of_numbers_original = list(map(float, list_of_numbers_original))   #Converting numbers stored as string to float
    list_of_numbers = list_of_numbers_original.copy()   # Making a copy of original list for processing

    mergeSort(list_of_numbers,0,len(list_of_numbers)-1)     # Sorting the list of numbers using merge-sort

    #Initalize an empty 2D array whose first coloumn will contain all the sorted numbers once and
    #second coloumn contains if that number has already been printed or not (0 for NO and 1 for YES)
    #so initially 0 for all
    #This array will be used later to print sequence in original order
    binary_array_for_numbers = []



    #Prining sorted list so that one number is printed only once

    print("The sequence with duplicates removed in sorted order is :")
    current = None   #Initalize variable which stores value of active number


    #Looping through list from index 1 to end
    for i in range(0,len(list_of_numbers)):
        if list_of_numbers[i]==current:    #If number is already encountered, skip
            continue
        #else the number is encountered for the first time print it and change the value of current to that value
        print(list_of_numbers[i],end = " ")
        current = list_of_numbers[i]
        #Adding number into binary_array_for_numbers array as it is encountered for first time
        temp = []
        temp.append(list_of_numbers[i])
        temp.append(0)
        binary_array_for_numbers.append(temp)

    print()
    print()


    #Printing sequence with duplicates removed in original order

    print("The sequence with duplicates removed in original order is :")

    #Looping through the numbers in the original list given by user
    for i in range(len(list_of_numbers_original)):
        index = binary_search(binary_array_for_numbers,list_of_numbers_original[i])  #Finding index of that number in binary_array_of_numbers using binary search
        if(binary_array_for_numbers[index][1] == 1):    #If the value of second coloumn is 1 for that index that means number has alrady been printed
            continue
        else:
            print(binary_array_for_numbers[index][0],end = " ")     #Else print the number and change value of second coloumn to 1
            binary_array_for_numbers[index][1] = 1


remove_duplicates() #Calling function to remove duplicates
```

Output for some test case

- ```
  To remove duplicates from a sequence of numbers :
  Enter sequence of numbers seperated by space : 2.1 -4.5 1.0 2.1 8.9 8.9 7.7

  The sequence with duplicates removed in sorted order is :
  -4.5 1.0 2.1 7.7 8.9

  The sequence with duplicates removed in original order is :
  2.1 -4.5 1.0 8.9 7.7
  ```

Algorithm

The algorithm can be divided into two parts. One is implemeted using sort by using merge sort to print the sequence in sorted order. And another is binary search to print the sequence in original order. In this we take advantage of the fact that we sorted the sequence to implement the first part.
Assumption : All the elements are floating point numbers.

$\underline{Sorted\ Order}$:

In this we use the conventional merge sort algorithm to sort the numbers first. Functions in the code to implement this are :

- **mergeSort()**: Recursive function to implement merge sort. This function divides the input array into two parts, sorts them using recursion and then finally join them using the merge() function.
- **merge()**: Function that merges two sorted arrays into one. It uses two pointer method where one pointer points to one element in each array and compare them and add the smaller number to the result array and increment that pointer untill all elements are added.

So, we first sort the list of numbers using merge sort. Now, we traverse the sorted array and print the first appearance of each number and, as the array is sorted we need to traverse only once.
Now, we already know that merge sort takes $O(nlogn)$ complexity. And then traversing the sorted array once takes $O(n)$ time. Thus, overall complexity of this part is
$O(nlogn) + O(n) = \underline{O(nlogn)}$.
Also, we know merge sort takes up $\underline{O(n)}$ space complexity.

$\underline{Original\ Order}$:

In this we use binary search on the already sorted array in the first part to implement the algorithm. Only function for this part is :

- **binary search()**: This function takes a 2D array and the number to be searched (x). It then uses conventional binary search algorithm to find the number in the first coloumn of the 2D array.
  Overview of the binary search algorithm is :
  1. Compare x with the middle element.
  2. If x matches with the middle element, we return the mid index.
  3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
  4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

  As we can see we reduce the size of array by half each time so binary search takes O(nlogn) time to execute.

Now to implement the algorithm we traverse the original sequence and print a number we encounter that number for the first time else skip.
To implement that we keep a 2D array (binary_array_for_numbers) whose first numbers contain all the numbers once in sorted order and second coloumn contains 0 if we haven't encountered the number yet and 1 if we have. We create this array while we traverse the sorted array in the previous part, every time we print a number in that part we also add the number in binary_array_for_numbers with initial encounter value as 0. Now, as we traverse the original sequence we use binary search to find that number in binary_array_for_numbers and check if we have encounter it before or not. If no print it, change it's encounter value to 1 else skip the number.
As we use binary search for every number in the original sequence the algorithm takes up $\underline{O(nlogn)}$ time as well.

Also, as we use a 2D array, space complexity of this part is $\underline{O(n)}$.
  ```

So, both the part of the codes take up $O(nlogn)$ time therefore,

$$the\ overall\ time\ complexity\ of\ the\ code\ is\ \underline{O(nlogn)}$$

Also, both the part of code takes up $O(n)$ space complexity therefore,

$$the\ overall\ space\ complexity\ of the\ code\ is\ \underline{O(n)}$$