Assignment Number [2]: Algorithms,2020

Nikilesh B
EE17B112

1. The order of the functions by asymptotic growth rate will be :

   2^10 **<** 3n+100*logn*, 4n, 2^*logn* **<** n*logn*, 4n*logn* + 2n **<** n^2 +10n **<** n^3 **<** 2^n

   Reasons :
   - **a.** b^n exceeds n^a for any a >0 and b>1.
   - **b.** Biggest term suppresses the other terms i.e., n^2 + 10n is nothing but O(n^2) and similarly other functions.
   - **c.** Also n*logn* increases greater than n and here 2^logn is equal to O(n).
   - **d.** The last function will be the constant function.

2.

   A. **f(n) = n - 100 g(n) = n - 200 Ω**
   Here f = Ω(g) , because there exists some constants c and N, so that for all n >= N , f(n) >= c*g(n).

   B. **f(n) = 100n + logn g(n) = n+logn^2**
   Here f=Θ(g) , because for larger values of n both tends to O(n).

   C. **f(n) = log(2n) g(n) = log(3n)**
   Here f =Θ(g) , because there is just a constant difference between the functions.

   D. **f(n) = n^1.01 g(n) = n*log^2(n)**
   Here f = **O**(g) , because there exists some constants c and N, so that for all n >= N , f(n) <= c*g(n).

3. The efficient algorithm will be to first sort the sequence and return the last 10 numbers in the sequence. Here in the code below the sort() function of python is used for which the average time complexity is o(nLogn).

```
 def find10largest(arr):
    n = len(arr)
    arr = sorted(arr) # avg. case Time complexity = O(nLogn)

    dupli_check = 0
    large_count = 1

    for i in range(1, n + 1):
        if(large_count < 11):
            if(dupli_check != arr[n - i]):

                # to handle duplicate values
                print(arr[n - i], end = " ")
                dupli_check = arr[n - i]
                large_count += 1
        else:
            break

#Sample test input and output
arr = [3,5,7,6,9,4,6,1,2,9,10,45,65,34,32,12,67]
find10largest(arr, n)

67 65 45 34 32 12 10 9 7 6
```

4.

Here we use divide and conquer algorithm or also known as Karatsuba Algorithm to multiply integers.

For multiplying binary integers 10011011 and 10111010 , we can just convert them to integer numbers by a function defined in code block and pass them on to the karatsuba function.

And the result can be converted back to binary also by the function `decimalToBinary`

```python
lookuptable = {
  (-1,-1): 1, (-1,0): 0, (-1, 1): -1,
  (0,-1): 0, (0,0):0, (0,1): 0,
  (1,-1): -1, (1,0): 0, (1,1): 1}


def karatsuba(x,y):
  n = max(x.bit_length(), y.bit_length())

  # trivial
  if n <= 1:
      return lookuptable[(x, y)]

  # divide
  k = (n + 1) >> 1
  x1 = x >> k
  x0 = x - (x1 << k)
  y1 = y >> k
  y0 = y - (y1 << k)

  # recursions
  a = karatsuba(x1, y1)
  c = karatsuba(x0, y0)
  p = karatsuba(x0 + x1, y0 + y1)
  b = p - a - c

  # conquer
  return (a << (n + (n & 1))) + (b << k) + c

 def binaryToDecimal(n):
      return int(n,2)

 def decimalToBinary(n):
      return int(bin(n).replace("0b", ""))
```

5. We can do this with a modified binary search algorithm.

i) If the mid element is greater than both of its adjacent elements, then mid is the maximum.
ii) If the mid element is greater than its next element and smaller than the previous element then maximum lies on the left side of mid.
iii) If the mid element is smaller than its next element and greater than the previous element then the maximum lies on the right side of mid.

```
def findMaximum(arr, low, high):
      n = len(arr)
      low = 0; high = n-1;
      # Base Case: Only one element is present in arr[low..high]*/
      if low == high:
      return arr[low]

      # If there are two elements and first is greater then
      # the first element is maximum
      if high == low + 1 and arr[low] >= arr[high]:
      return arr[low];

      # If there are two elements and second is greater then
      # the second element is maximum
      if high == low + 1 and arr[low] < arr[high]:
      return arr[high]

      mid = (low + high)//2   #low + (high - low)/2;*/

      # If arr[mid] is greater than both
      # arr[mid-1] and arr[mid+1], then arr[mid]
      # is the maximum element
      if arr[mid] > arr[mid + 1] and arr[mid] > arr[mid - 1]:
      return arr[mid]

      # If arr[mid] is greater than the next element and smaller than the
previous
      # element then maximum lies on left side of mid
      if arr[mid] > arr[mid + 1] and arr[mid] < arr[mid - 1]:
      return findMaximum(arr, low, mid-1)
      else: # when arr[mid] is greater than arr[mid-1] and smaller than
arr[mid+1]
      return findMaximum(arr, mid + 1, high)
```

6. For removing all the duplicates from the list first we will sort the list and pass it to the function which returns the size of the new array with no duplicates.

```
def removeDuplicates(arr, n):
      if n == 0 or n == 1:
      return n

      # To store index of next unique element
      j = 0

      # traversing through the array
      for i in range(0, n-1):
      if arr[i] != arr[i+1]:
            arr[j] = arr[i]
            j += 1

      arr[j] = arr[n-1]
      j += 1
      return j

arr = [1, 3, 2, 4, 5, 2, 4, 5, 4] #example input
arr.sort() #to sort the array
n = len(arr)

# removeDuplicates() returns
# new size of array.
n = removeDuplicates(arr, n)

# Print updated array
for i in range(0, n):
      print (" %d "%(arr[i]), end = " ")

#example output
1 2 3 4 5
```