CSE 221

Assignment -01

Theory

Name : Mihir Das

Student ID : 22299480

Section : 22

Department : CSE

Name : Mihir Das
Student ID: 22299480
Section : 22

Part-1

Q1

ⓐ $O(n)$
ⓑ $O(\sqrt{n})$
ⓒ $O(n^2)$
ⓓ $O(\sqrt{n})$
ⓔ $O(\log n)$
ⓕ $O(n^2)$

ⓖ Big O basically indicates the upper bound/limit meaning the worst case ~~sen~~ scenario for the ~~algor~~ algorithm. And that's why the running time of algorithm A is at least $O(n^2)$ is meaningless as the statement implied $O(n^2)$ will be it's worst case.

PART-2

Q1

ⓐ Yes, it is possible.

ⓑ
- calculating the mid index
- setting the base case: if len(arr) == 1 → return arr[0]
- if mid elem > (greater than) previous elem.
  it means max elem is in right of the arr so recursive call to the right of the array.
- if mid elem. < prev. elem.
  it means max elem is in the left side of the array. so recursive call to the left.

```
def getMax(arr):
    n = len(arr)
    mid = n//2
    if n==1 : return arr[0]
    if arr[mid] > arr[mid-1]:
        arr = arr[mid:] # right side
        return getMax(arr)
    else :
        arr = arr[:mid] # left side
        return getMax(arr)
```

(c) The time complexity of this algorithm is

$$T(n) = O(\log n).$$

$\boxed{Q2}$

Here, we will use two helper functions to get the output and ~~side~~ inside of the functions binary searches logic is implemented.

- find_first function tells us which index the number came up/starts.
- find_last function tells us which index the number ends.
- So, last_index - first_index +1 gives us how many

times it occured.

```
def modified_bin_search(arr,target):
    def find_first(arr, target):
        left =0; right = len(arr) -1
        first_seen = -1
        while left<=right:
            mid =(left +right)//2
            if arr[mid] == target:
                first_seen =mid    # start index
                right = mid -1
            elif arr[mid]<target:
                left = mid +1
            else: right = mid -1
        return first_seen

    def find_last(arr, target):
        left =0; right = len(arr) -1
        last_seen = -1
        while left<=right:
            mid = (left + right) //2
            if arr[mid] == target:
                last_seen = mid  # last index
                left =mid+1
            elif arr[mid] < target:
                left =mid+1
            else:
                right =mid -1
        return last_seen
```

indentation

```
first = find_first(arr, target)
last = find_last(arr, target)
```

```
if first ==-1 or last ==-1 :
    return 0
else:
    return last - first +1
```

This shows how many times a number appeared together.

Time complexity = $O(\log n)$

But this doesnot show all the appearance of that number, for that, in the last part,

```
count =0
for i in range(first, last+1):
    if arr[i] == target:
        count +=1
return count
```

This will give us all the apperances of that number.

$\boxed{\text{Q3}}$ (a) Binary search's time complexity (worst case) is $O(\log N)$ ~~a~~ whereas ~~it~~ time complexity of linear search is $O(n)$. To search in a large number of dataset it will take more time for linear search than Binary search. In fact it will take significantly less time than linear search. So, first sorting the array and then binary ~~seach~~ search is better.

(b) To make it work with the negative integers:

```
def count_sort(arr):
    min_val = min(arr)
    max_val = max(arr)
    range = max_val - min_val + 1
    c_arr = [0] * range
    output_arr = [0] * len(arr)
    for num in arr:
        c_arr[num - min_val] += 1
    for i in range(1, range):
        c_arr[i] += c_arr[i-1]
    for num in reversed(arr):
        output_arr[c_arr[num - min_val] - 1] = num
        c_arr[num - min_val] -= 1
    return output_arr
```

(c) To make it work with the given list:

```
def count_sort(arr):          # turn num in int type
    factor = 10
    int_arr = []
    for i in arr:
        num = int(i * factor)
        int_arr.append(num)
    :
    :
    sorted_arr = []           # back to float
    for n in output_arr:
        num = n/factor
        sorted_arr.append(num)
return sorted_arr
```

(d) For memory consumption merge sort is worse than quick sort because while merge sort sorts the array in $O(n\log n)$ ~~but~~ and quicksort sorts in $O(n^2)$ (worst case) still for memory consumption quick sort is better for a large dataset. Because of in-place sorting quick sort requires less memory.

(e) In ~~an~~ a sorted array quick sort will always pick the worst case possible pivot.

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Here, quicksort will have to go to the end of array which will make the time complexity $O(n^2)$. And this is where quick sort fails to work in $O(n\log n)$.

PART-3

Q1 (a) Selection ~~Array~~ Sort : Scans the entire array.

$T(n) = O(n^2)$

(b) Merge Sort : ~~divi~~ divide the array into two parts and them sort these two parts. Divide and conquer approach.

$T(n) = O(n\log n)$

Q2 Here, even indices hold numbers in decreasing order and odd indices hold numbers in increasing order.

- From the array separate the even indexed numbers in even_arr and the odd index one's in odd_arr.
- As even_arr is in descending order, reverse the even_arr
- merge the two array together

```
def sort_list(arr):
    n=len(arr); even_arr=[]; odd_arr=[]
    for i in range(n):
        if i%2==0: even_arr.append(arr[i])
        else: odd_arr.append(arr[i])
    even_arr.sort()
    even_arr = even_arr[::-1]
    odd_arr.sort()

    even_idx=0; odd_idx=0
    for i in range(n):
        if i%2==0:
            arr[i]=even_arr[even_idx]; even_idx+=1
        else:
            arr[i]=odd_arr[odd_idx]; odd_idx+=1
    return arr
```

PART-4

Q1 (A) $T(n) = 2T(n/2) + 1/n$

using master theorem,

$T(n) = aT(n/b) + f(n)$

$a = 2, b = 2$
$f(n) = 1/n$

$\therefore f(n) = O(n^c)$ where $c = -1$

compairing $n^{\log_b a}$ with $f(n)$:

$n^{\log_2 2} = n^1$

$\therefore T(n) = O(n^{\log_b a}) = O(n)$ $\therefore \underline{T(n) = O(n)}$

(B) $T(n) = 625T(n/5) + n^5$

using master theorem,

$a = 625, b = 5$
$f(n) = n^5$

compair, $n^{\log_b a}$ with $f(n)$

$\log 625 = \log_5 5^4 = 4$

Since, $f(n) = n^5$, So, $\underline{T(n) = O(n^5)}$

(C) $T(n) = T(n/2) + T(n/5) + n$

For, $n/2$ part $\rightarrow O(n)$
$n/5$ part $\rightarrow O(n)$

$\therefore \underline{T(n) = O(n)}$

(D) $T(n) = 2T(n/4) + n^2$

using master theorem,

$a = 2, b = 4$
$f(n) = n^2$

$\log_4 2 = 1/2$

$n^{\log_4 2} = n^{1/2}$

or in other words, $T/4 \rightarrow O(n)$
$O(n) + O(n^2) = O(n^2)$

since, $f(n) = n^2$

$\therefore \underline{T(n) = O(n^2)}$

Q2

(a) Benjamin used subproblems of size $n/3$. So, by assuming n is a power of 3, let's split A and B into three parts.

$$A = A1 \times 10^{2n/3} + A_2 \times 10^{n/3} + A_3$$
$$B = B1 \times 10^{2n/3} + B2 \times 10^{n/3} + B_3$$

(b) Calculating the product of AB

$$AB = (A1 \times 10^{2n/3} + A2 \times 10^{n/3} + A3) \times (B1 \times 10^{2n/3} + B2 \times 10^{n/3} + B3)$$
$$= (A1 \times B1 \times 10^{4n/3}) + (A1 \times B2 \times 10^{3n/3}) + (A1 \times B3 \times 10^{2n/3}) +$$
$$(A2 \times B1 \times 10^{3n/3}) + (A_2 \times B_2 \times 10^{2n/3}) + (A_2 \times B_3 \times 10^{n/3}) +$$
$$(A3 \times B1 \times 10^{2n/3}) + (A3 \times B2 \times 10^{n/3}) + (A3 \times B3)$$

(c) We will have to go through this following steps:

- single digit (A or B) return AXB. (Base Case)
- size of the numbers, m = (n+2)//3
- split A and B into three parts as

  A1 → highest part
  A2 → mid part
  A3 → lowest part ..... same goes for B1, B2, B3.

We will have to return,

(A1B1 * 10 ** (4 * m)) + (((A3A1)(B3B1)) - A1B1 - A2B2) * 10 ** (3 * m)) +
(((A1A2A3)(B1B2B3) - (A3A1)(B3B1) - (A3B3)) * 10 ** (2 * m)) + (A2B2 *
10 ** m) + A3B3

(c)

```
def kar_three_parts(A,B):
    if A<10 or B<10: return A*B   # Base case
    n=max(len(str(A)), len(str(B)))
    m=(n+2)//3

    A1 = A// (10**(2*m))
    A2 =(A// (10** m))% (10** m)
    A3 = A%(10** m)

    B1 = B//(10** (2* m)
    B2 =(B// (10** m))% (10** m)
    B3 = B % (10** m)

    A3B3 =kar_three_parts(A3, B3)
    (A1A2A3)(B1B2B3) = kar_three_parts((A1+ A2+A3),( B1+B2+B3))
    A2B2 = kar_three_parts (A2,B2)
    (A3A1)(B3B1) = kar_three_parts ((A3+A1),(B3+B1))
    A1B1 = kar_three_parts(A1, B1)

    return (A1B1*10 **(4* m)) +(((A3A1)(B3B1) - A1B1- A2B2)*
        10 ** (3* m))+(((A1A2A3)(B1B2B3)-(A3A1)(B3B1) -
        A3B3)* 10 **(2*m)) + (A2B2 * 10 ** m) + A3B3
```

(d) $T(n) = 5T(n/3) + O(n)$

by using master theorem, $T(n) = aT(n/b) + f(b)$

comparing $f(n)$ with $n^{\log_b a}$,

$a = 5, b = 3$
$f(n) = O(n)$

$$\log_b a = \log_3 5 \approx 1.4649$$

$$\therefore T(n) = O(n^{\log_3 5}) \approx O(n^{1.4649})$$

Q3

ⓐ Maximum sum Subarray is a suitable algorithm for divide and conquer.

ⓑ We will follow this steps:

- Base case: if len(B)==0: return 0
- for 0 → return 1 and 1 → return 0
- mid point and then split ~~are~~ into two parts (divide and conquer approach)
- left suffix of zeros: ending of the mid point
- right suffix of zeros: starting of the mid+1 point
- combined_max = left_suffix + right_suffix
- find the max (combined_max, l_max, r_max)

P.T.O.

ⓒ Time complexity, $T(n) = O(n \log n)$

(b) ~~part of (b)~~

```
def max_con_zeros(B):
    n = len(B)
    if n ==0:
        return 0
    def d_and_c(B, left, right):
        if left==right:
            if B[left]=='0':
                return 1
            else:
                return 0
        mid = (left +right)//2
        l_max = d_and_c(B, left, mid)
        r_max = d_and_c(B, mid+1, right)

        left_suffix = 0
        if B[mid]=='0':
            left_suffix = 1
            i = mid-1
            while i >=left and B[i]=='0':
                left_suffix += 1
                i -= 1
        right_prefix =0
        if B[mid+1] == '0':
            right_prefix =1
            i = mid+2
            while i<=right and B[i]=='0':
                right_prefix +=1
                i+=1
        combined_max = left_suffix +right_suffix
        return max(l_max, r_max, combined_max)
    return d_and_c(B,0, n-1)
```