BRAC University

Assignment No: 01

Course Code: CSE221

Semester: Summer 2023

Student Name: Sumaiya Azad Isha

ID: 22101503

Section: 15

Date of Submission: 7th July 2023

Subject :

Date :
Time :

Q.1

a. The given code snippet has undetermind time complexity. The inner loop will run infinite times. So, the time complexity is infinity.

b. The time complexity of this code is also infinity. Because the 2nd part of the code will run infinte time.

c. The statement is meaningless because doesn't give any details about the algorithm's performance. The work 'at least' means the lower bound of the algorithm. Big-oh notation means the upper bound. So the statement doesn't make any sense.

Subject :

Date :

Time :

Q.2

a. user_id = 'x'

details = [[id, name], ..., n]

for i → 0 to n+1:

if user_id == details[i][1]

return True

return False.

b. countsort can be used here.

Algorithm:

CountSort (A, Output,

C = [0]* max[A] → new array ; output = [0]* len(A)

for i → 0 to len(A):

C[A[i]] += 1

for j → 0 to len(C):

C[j] += C[j-1]

for K → len(A)-1 to -1:

output [C[A[K]]-1] = A[K]

C[A[K]] = C[A[K]-1]

return output

Friendship

Subject :

Date :
Time :

c. Yes, the crash was a result of the sorting algorithm. It is possible that the sorting ~~algori~~ algorithm was running out of memory if the maximum number of postcount is very big. Due to insufficient memory allocation there will a crash.

d. Merge Sort has a time complexity of nlogn in best, worst and average case.

```
def mergesort (A, l, h):
    if (l>h):
        mid = (l+h)//2
        ~~mergesort(l~~
        lA = mergesort (~~l, mid~~ A, l, mid)
        rA = mergesort (A, mid + 1, h)
        merge (lA, rA, n, m):
        ~~nA = [0] * (m+n)~~
        ~~while (~~
```

Friendship

```
def merge(lA, rA, n, m):
    mA[0]*(m+n)
    i, j, K = 0
    while (i<n and j<m):
        if lA[i] < rA[j]:
            mA[k] = lA[i]
            i+=1
        else:
            mA[k] = rA[j]
        k+=1
    while i,<n:
        mA[k], = lA[i]
        i+=1, k+=1
    while j<m:
        mA[k] = rA[j]
        j+=1, k+=1
    return mA
```

Subject :

Date :

Time :

Subject :

Date :
Time :

Q. The searching algorithm is Binary Search.

Algorithm:

```
Def BinarySearch(A, X): # X is the target value
    l = 0
    h = len(A)-1
    while (l<=h):
        mid = (l+h)//2
        # if x is greater, ignore the left half
        if A[mid] < x:
            l = mid+1
        # if x is smaller, ignore the right half
        elif A[mid] > x:
            h = mid-1
        else:
            return "Found"

    return "Not found"
    # if the element is not present in the list.
```

Friendship

Subject :

Date :
Time :

Steps to search for post count 9

Since the list is already sorted, the mid element is 5.

→ 9 is greater than 5, so we will do the searching to the right half ([8, 9, 12, 14, 17])

→ The mid element of the right half is 12

→ 12 is greater than 9, so we will search in left half : [8, 9]

→ The mid element is 8, so we will search in the right half : [9]

→ The mid element is 9. We found the target.

Subject :

Date :
Time :

Q.3

1. The modified ~~algorithm~~ algorithm is :

```
def b-search (array, ele):
    l=0 , h= len(array)-1
    while (l <=h) :
        mid= (l+h)//2
        if array[mid] < ele:
            l=mid+1
        elif array [mid] > ele:
            h= mid-1
        else:
            return mid

    return -1
```

2. To return the first index if there are duplicates.

```
def b-search (array, ele) :
    l=0, h= len(array)-1 , index = -1
    while (l≤h):
        mid=(l+h)//2
        if array [mid ] < ele :
            l= mid+1
        elif array [mid] >ele:
            h= mid-1
```

Friendship

Subject :

Date :
Time :

```
        else:
            indx = mid
            h = mid-1
    return indx
```

3. To return index and first appearance:

```
def b_search(array, ele):
    l=0, h = len(array)-1, f_index = -1
    while (l<=h):
        mid = (l+h)//2
        if array[mid] < ele:
            l = mid+1
        elif array[mid] > ele:
            h = mid-1
        else:
            f_index = mid
            h = mid-1
    temp = f_index
    while (temp < len(array) and array[temp] == ele):
        temp += 1
    count = temp - f_index
```

Friendship

Subject :

Date :
Time :

```
    if f_index == -1:
        return False

    return (f_index, count)
```

4. To find minimum element wave pattern:

```
def find_min_ele (array):
    l=0, h=len(array)-1
    while (l<=h):
        mid = (l+h)//2
        if (mid==0 or array[mid-1] > array[mid])
            and (mid == len(array)-1 or array[mid+1] > array
                                                [mid]):
            return array[mid]
        elif (mid > 0) and (array[mid-1] < array[mid]):
            h = mid - 1
        else:
            l = mid + 1
    return array[mid]
```

Friendship

Subject :

Date :
Time :

Q.4 :

For the given list if we search for T=2 the steps will be:

middle	L_indx	R_indx
	0	7
3	0	2
2		

↳ found T

@ As we can see the list is not sorted. Coincidently we have found the value T=2 but if the value was ~~23 or 19 or any~~ 1 or anything we would not be able to find it because the array/list is not sorted.

Subject :

Date :
Time :

Q.5.

a Using Binary Search we can solve it.

```
def b_search_find_max (array):
    l=0, h= len(array)-1
    while (l<=h):
        mid= (l+h)//2
        if (array[mid]> array[mid-1]) and (array[mid]>
                                          array[mid+1]):
            return array[mid]
        elif (array[mid]> array[mid-1]) and (array[mid]<
                                          array[mid+1]):
            l= mid +1
        else:
            h= mid-1
    return max (array[l], array[h])
```

b. The time complexity of this code is O(logN).

Friendship

Subject :

Date :
Time :

Q.6.

a. Sorting an array before binary search is useful ~~because~~ when we need to search for multiple times. Sorting the array takes O(NlogN) time (only once). Then we can search for the element in O(logN) time. But if we do linear search it will take O(N) time for each search which is not efficient way to do the searching.

b. Modified countsort algorithm:

```
def countsort(array):
    #find the max, min of the array
    min_ele = min(array), max_ele = max(array)
    count = [0]* (max_ele - min_ele + 1)
    for val in array:
        count[val - min_ele] += 1
```

Subject :

Date :
Time :

```
    for i in range(1, max_ele - min_ele + 1):
        count[i] += count[i-1]

    output = [0] * len(array)

    for j in range(len(array)-1, -1, -1):
        output[count[array[j] - min_ele] - 1] = array[j]
        count[array[j] - min_ele] -= 1

    return output
```

c. Same as the code ~~correct~~ in (b).
Here we'll just take the int value using the int() function.

```
def count_sort(array):
    max = int(max(array)), min = int(min(array))
    count = [0] * (max - min + 1)
    for val in array:
        count[int(val) - min] += 1
```

iendship

Subject :

Date :
Time :

```
for i in range (1, max-min+1):
    count[i] += count[i-1]
output = [0]* len (array)
for j in range (len(array)-1, -1, -1):
    output[count[int(array[j])-min]-1] =
                                    array[j]
    count [int(array[j])-min] -= 1
return output
```

d. QuickSort is faster than mergesort when the pivot is chosen wisely. If the the pivot is chosen poorly Quicksort will have the worst-case time complexity of $O(n^2)$. Mergesort work in $O(N \log N)$ time complexity in every case. If we want to run the code at $O(n \log n)$ time for any case then mergesort is a better option.

Friendship