Mayank Joshi
1961101 BTech CSE
VC

~~Ans~~1) function LSearch (array, key ~~}~~, size)

```
function LSearch (array, key, size)
    for index = 0 to size -1
        if Key== array[index] then
            return index
        endif
        ~~else~~
    end forloop
    return -1
end.
```

Ans2) ~~Inserst~~ Iterative insertion sort

```
Function IIS (array, size)
{
    For i=1 to n-1 do
    Begin
        Key= array[i]
        j=i-1
        while j>=0 and array[j]>Key do
        Begin
            array[j+1]= arr[j]
            j=j-1;
        End while loop
        array[j+1]= Key
    End for loop
} End
```

Recursive insertion sort

```
Function RIS ( array, size)
Begin
    if size <=1 then
        return;
    endif RIS(array, size-1)
    j= size-2
    Key = array [size-1]
    while j>=0 and array[j] > Key then
    Begin
        array [j+1] = array[j];
        j = j-1
    End while loop
    array [j+1]= Key
End
```

An online algorithm is one that can process its input piece by piece (2) in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the beginning. Insersion sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is an online algorithm. Other algorithms reauires presence of all elements to be sort on the first iteration.

Ans 3:

Time complexity Best	Average	worst	space complexity	Algorithm
O(n2)	O(n2)	O(n2)	O(1)	Bubble sort
O(n2)	O(n2)	O(n2)	O(1)	Selection Sort
O(n)	O(n2)	O(n2)	O(1)	Insertion Sort
O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Merge sort
O(nlogn)	O(nlogn)	O(n2)	O(n)	Quick sort
O(nlogn)	O(nlogn)	O(nlogn)	O(1)	Heap Sort
O(n+K)	O(n+K)	O(n+K)	O(K)	counting sort

Ans 4:

Algorithm	Inplace	Stable	Online
Bubble sort	✓	✓	✗
Selection Sort	✓	✗	✗
Insertion Sort	✓	✓	✓
Merge Sort	✗	✓	✗
Quick Sort	✓	✗	✗
Heap Sort	✓	✗	✗
counting sort	✗	✓	✗

Ans 5:

```
Function BS(array, Key, first, last)
Begin
      mid= first+(last-first)/2
      if array[mid]== Key
            return mid+1
      else if array[mid]<Key
      return BS(array, mid+1, last)
      else
      return BS(array, first, mid-1)
      return -1;
End.
```

Time complexity = O(n){L}, O(logn){RB, IB}; Space complexity: O(1){L, IB}, O(n){RB}

Ans 6) $T(n) = \begin{cases} 1 & , \text{if } n = 1 \\ T(\frac{n}{2})+1 & , \text{if } n > 1 \end{cases}$

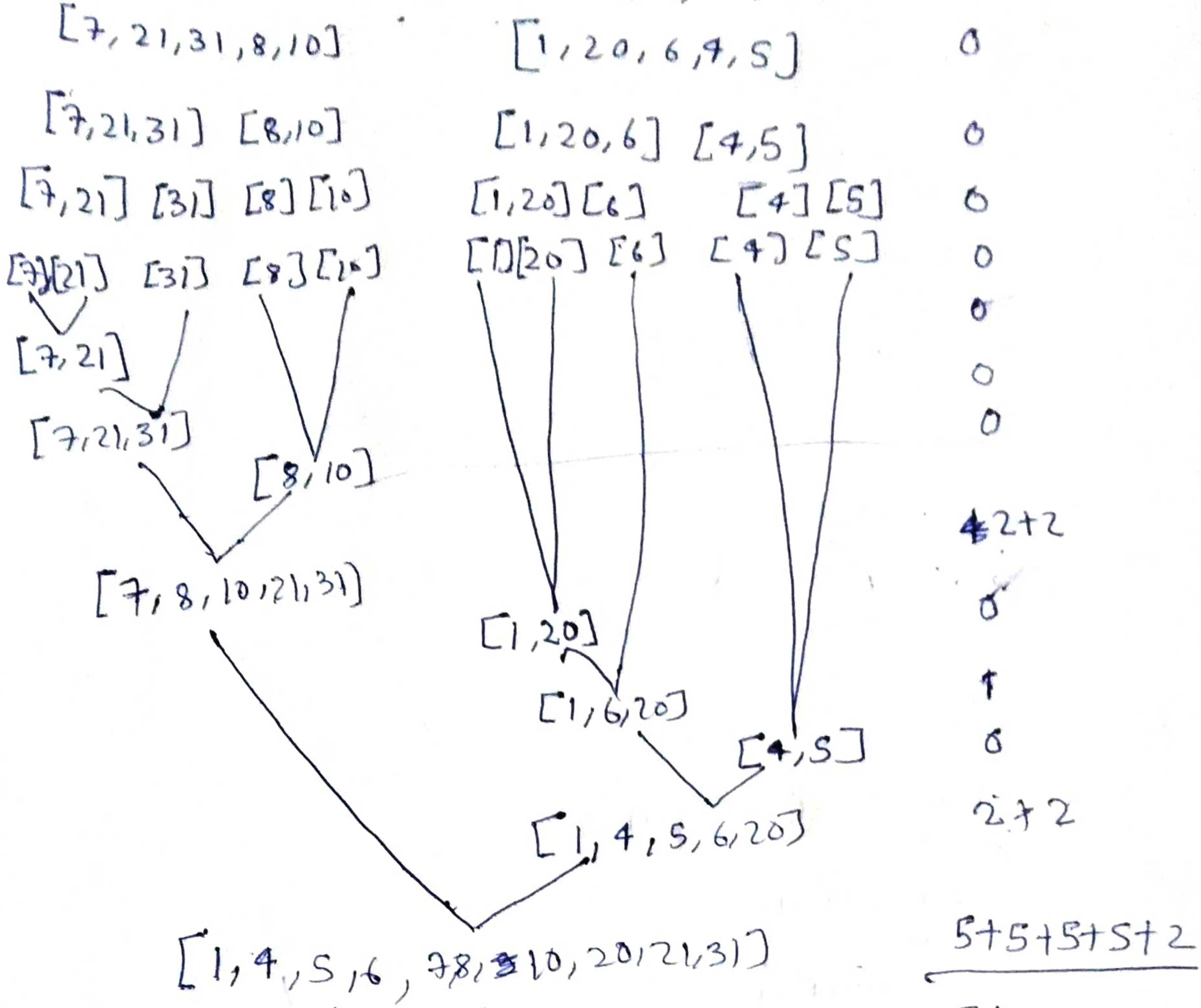
Ans 7) function fun(A, n, K)

```
Begin
    Sort (A, n)
    l=0, r = n-1
    while (l<r) do
    Begin
        if A[l]+A[r]==K then
              return (l,r)
        else if A[l]+A[r] > K then
              r=r-1
        else
              l= l+1
    End while loop
    return (-1,-1)
End.
```

Ans 8) Quicksort is preferred for practical implementation since it is faster than other $O(n\log n)$ algorithms in practice because its inner loop can be efficiently implemented on most architectures and in most real-world data, it is possible to make design choices that minimize the probability of requiring quadratic time.
Quicksort tends to make excellent usage of memory heirarchy, and uses virtual memory and available caches. so it is suitable for modern computers.

Ans 9) Inversions count for an array indicates- how far (or close) the array is from ~~beig~~ being sorted.
If the array is sorted then inversion count is 0.
inversion is when $a[i] > a[j]$ and $i < j$

array = [7, 21, 31, 8, 10, 1, 20, 6, 4, 5] count



Total no. of inversions = 31

Ans 10) when the array to be sorted is reversely sorted or already sorted then this is the worst case of Quick sort. $O(n^2)$
Best case is when partition process always pick the middle element as pivot. $O(n \log n)$

Ans 11)

	Merge sort	Quick sort
best case	$T(n) = 2T(\frac{n}{2}) + n$	$T(n) = 2T(\frac{n}{2}) + n$
worst case	$T(n) = 2T(\frac{n}{2}) + n$	$T(n) = T(n-2) + n$

The best and worst case recurrence relation of Merge Sort is same as Best case Recurrence relation of Quicksort.
Difference between the complexities is worst case of Quicksort is $O(n^2)$ and that of merge sort is $O(n \log n)$. This is because merge sort always divides the array in two halves of ~~$\frac{n}{2}$ size~~ but this is not in worst case of quicksort, which divides array in two subarrays of size (n-1) and 1.

Ans12> ~~stable~~ selection sort is as follows

```
Function SS(array, size)
Begin
    for i = 0 to n-2
    Begin
        minind = i
        for j = i+1 to n-1
        Begin
            if array[j] < array[min ind] then
                minind = j
        End for loop
        swap (array[i], array[minind])
    End for loop
End
```

now stable selection sort is as follows

```
Function SSS(array, n)
Begin
    for i = 0 to n-2
    Begin
        minind = i
        for j = i+1 to n-1
        Begin
            if array[j] < array[minind]
                minind = j
        End
        Key = array[minind]
        while min > i do
        Begin
            array[minind] = array[minind-1]
            minind = minind - 1
        End
        array[i] = Key
End
```

Ans 13) function MBS (array, n)

Begin

For i=0 to n-1 do

Begin flag=0

For j=0 ~~to~~ to n-i-1 do

Begin

if array [j+1] < arr[j]

swap array[j], array[j+1]

~~End forloop~~

flag = 1

End forloop

if flag == 0 then

break the outer loop

End outer forloop

End

Ans 14) In this case merge sort will be used as it can ~~mer~~ sort ~~chunks~~ ~~of~~ data in chunks. The 4 GB file is kept in external storage and small ~~ar~~ size of data is taken for merging. ~~and~~ The sorted subarrays are again kept into ~~the~~ external storage.

~~to Extertan~~

External sorting can be done in this case.

In internal sorting all the data to sort is stored in memory at all times while sorting is in progress.

In external sorting ~~all the~~ data is stored outside memory and only loaded into memory in small chunks