Name- Siddharth Sonal
Section- CE
Roll No- 08

M T W T F S S
Page No
Date
YOUVA

1.
```
linearSearch (vector<int> v, int target)
    i = 0;
    size = v.size();
    j = size-1, flag = 0;
    while (i < size && j > -1)
    {
        if (v[i] < target && v[j] > target)
            i++;
            j--;

        else if ( v[i] == target || v[j] == target)
            flag = 1;
            break;

        else
            break;

    if ( flag == 0)
    print "Not Found";

    else
    print "Found";
```

2.
```
linear ( vector<int> &nums)
    size = nums.size();
    i, j, temp;
    for ( i=1; i< size ; i++)
            j = i-1;
            temp = nums[i];
            while ( j > -1 && temp < nums[j])
                    nums[j+1] = nums[j];
                    j--;
```

```
nums[j+1] = temp;

recursive (vector<int> &nums, int size)
    if (size<=1)
    return;
    recursive (nums, size-1);
    temp = size - 2;
    while (j>=-1 && nums[j]>temp)
        nums[j+1]= nums[j]
        j--
    nums[j+1] = temp
```

Insertion sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is an online algorithm.

While the other sorting algorithms know all about it's input data the moment it is invoked. Thus they are called offline sorting algorithm.

3 =

	Best	Worst
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n\log n)$	$O(n\log n)$
Quick Sort	$O(n\log n)$	$O(n^2)$

Counting Sort	O(n+k)	O(n+k)
Heap Sort	O(n log n)	O(n log n)

4 =

	Inplace	Stable	Online
Selection	✓	X	X
Bubble	✓	✓	X
Insertion	✓	✓	✓
Merge	X	✓	X
Quick	✓	X	X
Count	X	✓	X
Heap	✓	X	X

5 =

```
linear (vector<int> &nums, int target
    size = nums.size()
    i = 0, j = size - 1
    mid, flag = 0
    while (i <= j)
        mid = (i+j)/2
        if (nums[mid] == target)
            flag = 1
            break
        if (nums[mid] < target)
            i = mid+1
```

```
        else
            j = mid - 1

    if (flag)
        print "Found"

    else
        print "NOT Found"

recursive (vector<int> nums, int i, int j, int target)
    mid;
    if(i<=j)
        mid = (i+j)/2;
        if(nums[mid] == target)
            return 1
        else if (nums[mid] >= target)
            return recursive(nums, i, mid-1,
                                target);

        else
            return recursive (nums, mid+1, j,
                                target);

    return 0
```

Time Complexity of linear binary search = $O(n^2)$

Time Complexity of ~~binary~~ recursive binary search = $O(\log n)$

6 $$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + 1 & n > 1 \end{cases}$$

7
```
void findIndex(vector<int>& nums, int k)
{
    int size = nums.size();
    int i=0, j=size-1;
    int sum=0;
    while(i<=j)
    {
        sum = nums[i]+nums[j];
        if(sum<k)
        i++;
        else if(sum>k)
        j--;
        else if(sum==k)
        {
            cout<<i<<" AND "<<j<<endl;
            i++;
            j--;
        }
    }
}
```

8 For ~~pac~~ practical uses, quicksort is widely used sorting algorithm that has an average time complexity of O(n log n) & is often faster than other popular sorting algorithms. Quicksort is particularly efficient for large datasets & can be easily implemented in place to save memory.

However, it's worst case time complexity is $O(n^2)$ which can occur when the input data is already sorted.

10 Best case → The pivot element chosen should be the median of the array. If the pivot is chosen as the median at each step, then the partitioning step will divide the array into two sub arrays of equal size, resulting in balanced tree of recursive calls. In this case, the time complexity of quick sort is $O(n \log n)$.

Worst case → In worst case, the pivot element chosen at each step is either largest or smallest element in the sub-array.

Note: Worst case occurs when the array is already sorted or reverse sorted. Time complexity will be $O(n^2)$

11 Recurrence Relation for merge sort

Best case → $T(n) = 2T(n/2) + O(n)$
Worst case → $T(n) = 2T(n/2) + O(n \log n)$

Recurrence Relation for quick sort

Best case → $T(n) = 2T(n/2) + O(n)$
Worst case → $T(n) = T(n-1) + O(n)$

Similarities between the two algorithms is that they have same average & best case time complexity i.e. O(nlogn)

The difference b/w the time complexities of merge & quick sort are that merge sort has worst case time complexity of O(nlogn) & Quick sort has the worst case time complexity of $O(n^2)$.

Both merge & quick sort have the same space complexity i.e. O(n) as they need to create the temporary array.

12 Yes, it is possible to implement the stable version of selection sort.

```
void selectionSort (int arr[], int n)
{
    for (int i = 0; i < n-1; i++){
        int min = i;
        for (int j = i+1; j < n; j++){
            if (arr[j] < arr[min]){
                min = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```

13 Yes, we can modify the bubble sort algorithm to optimize it so that it does not scan the entire array once it is already sorted.

```
void bubbleSort(int arr[], int n){
    bool swapped;
    for (int i=0; i<n; i++){
        swapped = false;
        for (int j=0; j<n-1-i; j++){
            swap(arr[j], arr[j+1]);
            swapped = true;
        }
    }
    if(!swapped){
        break;
    }
}
}
```

14 It is not possible to load the entire array into the memory for sorting using internal sorting. In this case we would need to use external sorting algorithms that operates on disk instead of memory.
External sorting is the technique used to sort large data sets that can not be held in memory at once. It involves a combination of internal & external sorting techniques.

The most commonly used external sorting algorithm is external merge sort. In this algorithm the

data is split into smaller parts that can fit into memory & each chunk is sorted using an internal sorting, such as ~~device sort~~ quick sort or heap sort. The sorted chunks are then merged together in a series of passes where the data is read from the disk, merged & written back to disk.