A) Binary search algorithm

→ Searching algorithm used to search in a sorted array by dividing the search interval in half. The time complexity is reduced to O(log n).

Basic steps:

- Sort the array in ascending order.
- Set the low ~~order~~ index to first element of the array and high index to the last element.
- Set the middle index to the average of the low and high indices.
- if element is at middle index, then return the middle index.
- if target element is greater than middle element, set low index to mid + 1.
- if target element is lesser than middle element, set high index to mid - 1.
- Repeat steps 3-6, until the element is found or it is clear that element is not present in the array.

1) Iterative method

```
binarysearch (arr, x, l, h) {
    repeat till l == h {
    mid = (l+h)/2
        if (x == arr[mid])
            return mid
        else if (x > arr[mid])
            l = mid + 1
        else  h = mid - 1 }
```

Date .....

2) recursive method

```
binarysearch(arr, n, l, h) {
    if low > high
        return 0
    else
        mid = (l+h)/2
        if (n == arr[mid]
            return mid
        else if (n > arr[mid]
            return binarysearch(arr, n, mid+1, h)
        else
            return binarysearch(arr, n, l, mid-1)
    }
```

Time complexity: $O(\log n)$

recurrence relation: $T(n) = T(n/2) + 1$

where, $T(n)$ is time required for binary search in an array of size n.

3) Merge sort algorithm

→ Merge sort is sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray and then merging the ~~sub~~ sorted subarrays back together to form the final sorted array.

→ In simple terms, divide the array into two halves, sort each half and then merge sorted halves back together, until the whole array is sorted.

Spiral

Date .....

Complexities

Time complexity :
Best case : O(n log n)
Avrage case : O(n log n)
Worst case : O(n log n)

space complexity :
✓ stable, O(n)

algorithm :

arr = array
s = start of array
e = end of array

```
mergesort(arr, s, e){
  if (s >= e)
      return
  mid = (s+e)/2
  mergesort(arr, s, mid)
  mergesort(arr, mid+1, e)
   merge(arr, s, e)
}

merge(arr, s, e){
      mid = (s+e)/2
      l1 = mid - s + 1
      l2 = e - mid
int a1 as array of length l1
int a2 a: array of length l2
//copy elements from main array to left and
    right array
  k = s ; //index of main array
for(i=0 to (1){
```

Spiral

Date .....

```
a1[i] ← arr[k++];
}
k = mid + 1;
for(i = 0 to l2){
    a2[i] ← arr[k++];
}
// merge 2 sorted arrays
index1 = 0
index2 = 0
k = s
while(index1 < l1 && index2 < l2){
    if(a1[index1] < a2[index2]){
        arr[k++] = a1[index1++];
    else{
        arr[k++] = a2[index2++];
    }
}
while (index1 < l1){ //for remaining of a1
    arr[k++] = a1[index1++];
}
while (index2 < l2){
    arr[k++] = a2[index2++];
}
}//end
```

recurrence relation:

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

Spiral

2122/CO/417

Date .....

1) Quicksort

```
int Partition (int Arr, L, h){
    Pivot = Arr[L]
    i=L, j=h;
    do{
        do{i++;} while (A[i] <= pivot);
        do{j--;} while (A[j] > pivot);
        if(i<j){
            swap(A[i], A[j]);
        } while(i<j);
    swap(A[L], A[j]);
    return j;
    }
```

time complexity:

Best case: O(nlogn)
worst case: O(n²)
average case: O(nlogn)

```
void quicksort(Arr, L, h){
    if(L<h){
        int j = partition(Arr, L, h);
        quicksort(Arr, L, j);
        quicksort(Arr, j+1, h);
        }
    }
```

Spiral