Binary search - sorted array)

```
while ( left<=right)
  //if ( a[mid] < search_element
   mid= left + (right-left)/2 ;   if( a[mid]==search_ele)
                                      return mid ;
   if( search_ele < a[mid])
           left = mid+1
  else if ( search_ele > a[mid])
           right = mid -1
```

Recursive approach

```
public static int binarySearch( int array[], int left,
        int right, int item){
   if( left<=right)
    {
         int mid = left+ ( right-left) / 2 ;
         if( array[mid] == item)
                return mid;
         if( array[mid] > item)
              return binarySearch( array , left , mid+1, ite
         else
              return binarySearch( array, mid+1, right,
                                                  item );
     }
   else
        return -1;
}
```

Time complexity

Best : O(1)

Average : O(log n)

Worst : O(log n)

Space complexity

O(1)

Sorting

in-place sorting — Not-in-place sorting

Bubble sort, Selection sort, insertion
Heap

Merge
Quick

Bubble sort

1. We will take starting two elements from the list.
2. we will compare those elements.
3. If These elements are found in unsorted order, we will sort them.
4. else we will compare next to elements.
5. Repeat until we get sorted array

pass 1

28 6 4 2 24 → 6 28 4 2 24
6 28 4 2 24 → 6 4 28 2 24
6 4 28 2 24 → 6 4 2 28 24
6 4 2 28 24 → 6 4 2 24 28

pass 2

6 4 2 24 28 → 4 6 2 24 28
4 6 2 24 28 → 4 2 6 24 28
4 2 6

Algo

```
for (int i=0; i<len-1 ; i++) {
    for (int j=0; j<len-1-i ; j++) {
        if (a[j] > a[j+1])
        {   int temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp
        }
    }
}
```

TC

Best: $O(n)$

Average = $O(n^2)$

Worst : $O(n^2)$

Space complexity

$O(1)$

In optimized bubble sort, use a boolean variable swapped, initialize with 'false' for every iteration & if it remains false at end of iteration i.e, no swapping occurs in a pass (it indicates that array is already sorted) then you can break the loop

Selection Sort - At any pt of implementation, we'll have array divided into a sorted array part on the left and unsorted array part on the right.

1. We find smallest ele on unsorted part of array
2. replace it with first item on unsorted part of array
3. sorted arr is by one element.
4.

72 50 10 44 8 20 → 8 | 50 10 44 72 20

→ 8 10 | 50 44 72 20

→ 8 10 20 | 44 72 50

→ 8 10 20 44 | 72 50

→ 8 10 20 44 50 72

Algo

```
for( i=0, i<len-1; i++){ min=i
    for( j=i+1 ; j<len; j++)
        if (a[j]<a[min])
            min=j
    swap( a[min], a[i])
```

Time complexity

$O(n^2)$

Insertion Sort

1 3 5 8 9 2 4 7 →

Ex: 3 5 1 9 8 2 4 7 → 3 5 1 9 8 2 4 7

5 3 1 9 8 2 4 7

1 3 5 9 8 2 4 7

1 3 5 9 8 2 4 7

1 3 5 8 9 2 4 7

1 2 3 5 8 9 4 7

1 2 3 4 5 8 9 7

1 2 3 4 5 7 8 9

```
for(i=1; i<len; i++)
{
    int key = arr[i]
    j=i-1
    while( j>=0  && arr[j] > key )
    {   arr[j+1]=arr[j]
        j=j-1
    }
    arr[j+1] = key
}
```

i=6 2

j=5

1 3 5 8 9 2

1 3 5 8 8 9

1 3 5 5 8 9

1 3 3 5 8 9

1 3 3 5 8 9

1 2 3 5 8 9

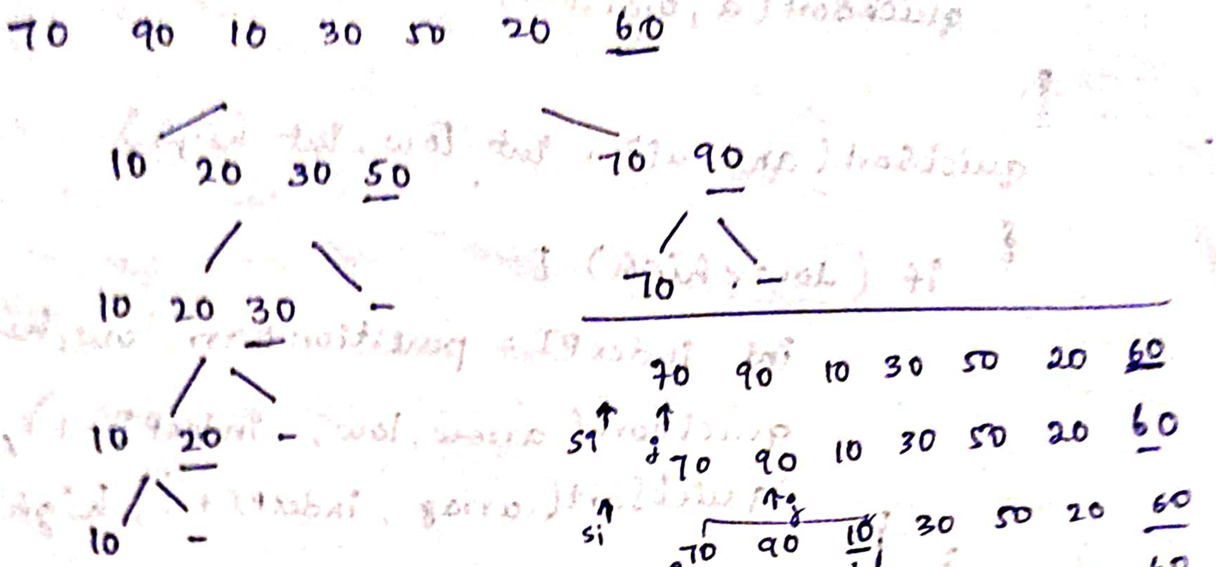
Space complexity
$O(1)$

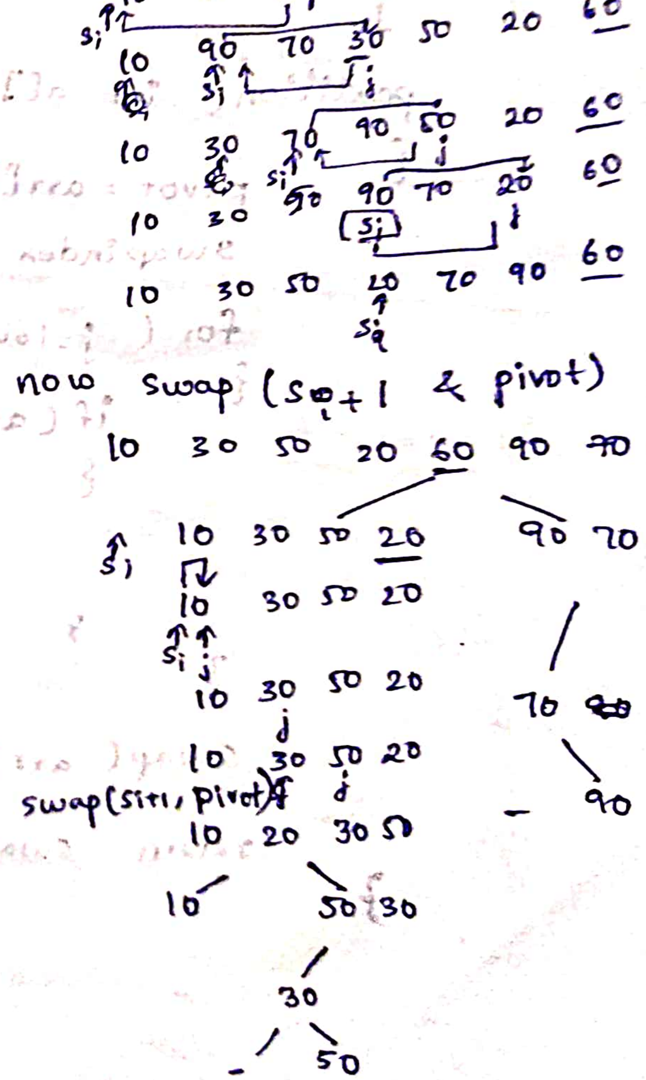
Time complexity
$O(n^2)$

Quick sort in Java

1. choose pivot element
2. move ele less than pivot in left partition
3. move ele greater than pivot to right "
4. partition index is discovered at the end

pivot = high





Time complexity

O(nlogn) ← Best, Average

O(n²) ← If pivot is chosen as first element.

Space complexity

O(1)

Auxiliary space complexity O(log n)

↑

function call stack

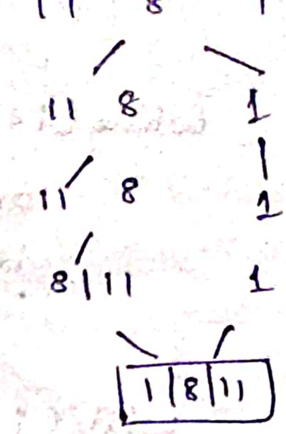
Quick sort

```
main(      )
{
    int a[] = {1,2,3,4,5,6,7,8,9};
    int size = a.length;
    quickSort(a, 0, size-1)
}

quickSort(int a[], int low, int high)
{
    if (low < high) {
        int indexPI = partition(arr, low, high)
        quickSort(arrow, low, indexPI-1),
        quickSort(array, indexPI+1, high);
    }
}

partition( int a[], int low, int high)
{
    pivot = arr[high]
    swapindex = low-1
    for ( j=low, j<=high-1, ;j++)
    {
        if(arr[j] < pivot)
        {
            swapindex++
            swapp(arr[swapindex], arr[j])
        }
    }
    swap( arr[swapindex+1], a[high] ),
    return swapindex+1;
}
```

Merge Sort



-L1

48 12

```
main( )
{
    mergeSort(a,0,size-1)
}
mergeSort( a, left ,right )
{
    int mid;
    if left< right {
        mid= left +(right-left)/2
        mergeSort(a,left, mid);
        mergeSort(a, mid+1, right);
        merge(a, left, mid, right);
    }
}
merge ( arr, left, mid, right)
{
    int i,j,k,n1,n2
    n1 = mid-left +1        // length of left subarray
    n2 = right - ~~left~~ mid
    L[n1] ,  R[n2]
    for(i=0 ;i<n1 ; i++)
            L[i]= arr[ left+i]
    for (j=0  ; j<n2; j++)
            R[j] = arr [mid+1+ j]
    i=0, j=0 // Starting index of L,R
    k = left //            merged subarray)
    while( i<n1 && j<n2) {
        if(L[i]<=R[j] {
            arr[k] = L[i]
            i++
        }
        else{ arr[k]= R[j] ; j++
        }
        k++ ;
}
```

```
while (i < n1) {
        arr[k] = L[i];
        i++
        k++
}
while (j < n2) {
        arr[k] = R[j]
        j++
        k++
}
} // merge method close
```

Time complexity

Best
Worst } $O(n \log n)$
Average

Space complexity

$O(n)$

Recursions