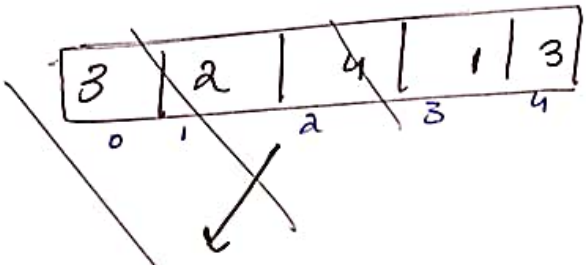
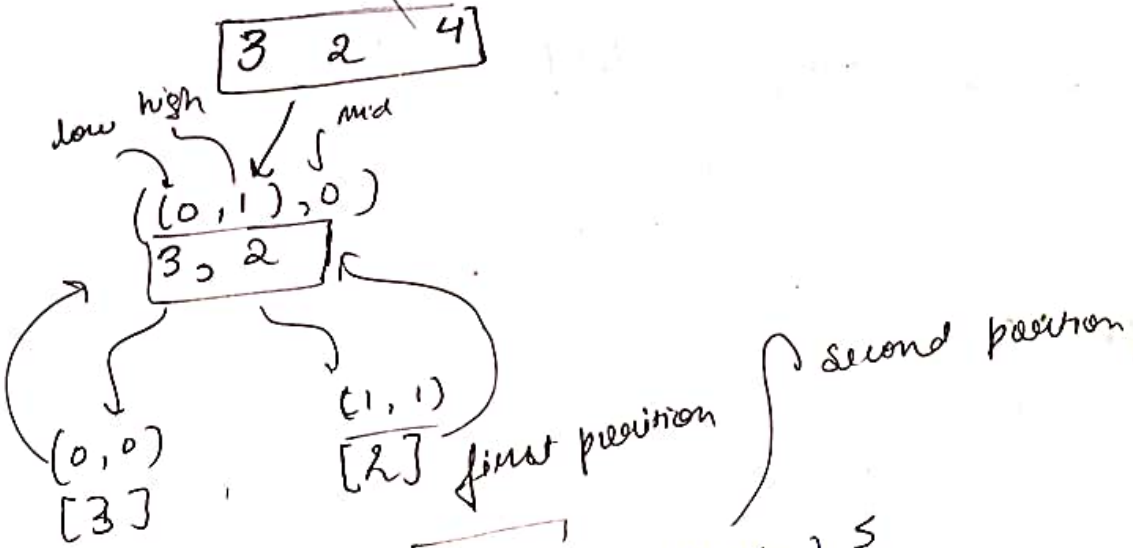
```
mergeSort (arr, low, high) {
    if (low >= high) return;
    mid = (low + high) / 2;
    mergeSort (arr, low, mid);
    mergeSort (arr, mid+1, high)
    merge (arr, low, mid, high)
}
```






```
merge (arr, low, mid, high) {
    temp
    arr → []
    left → low
    right → mid+1;
    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.add (arr[left])
            left++; }}
```

```
        else { temp.add(arr[right])
               right++
        }
    }

    while (left <= mid) {
        temp.add(arr[left]);
        left++;
    }

    while (right <= high) {
        temp.add(arr[right]);
        left++; right;
    }

    for (i = low → high) { arr[i] = temp[i - low]; }
}
```

TIME COMPLEXITY -

QUICK SORT ALGORITHM

Better in a way that it does not require a temp array for sorting like merge sort does.

- Pick a pivot element & place it in its correct place in the sorted array. (where all element on left are smaller or equ and on right > element)

- The pivot can be 1st
- last, median or any random element of the array

working.

4 6 2 5 7 9 1 3

pivot

How pivot element 4 is placed on correct position?

i → ← j

4 6 2 5 7 4 1 3

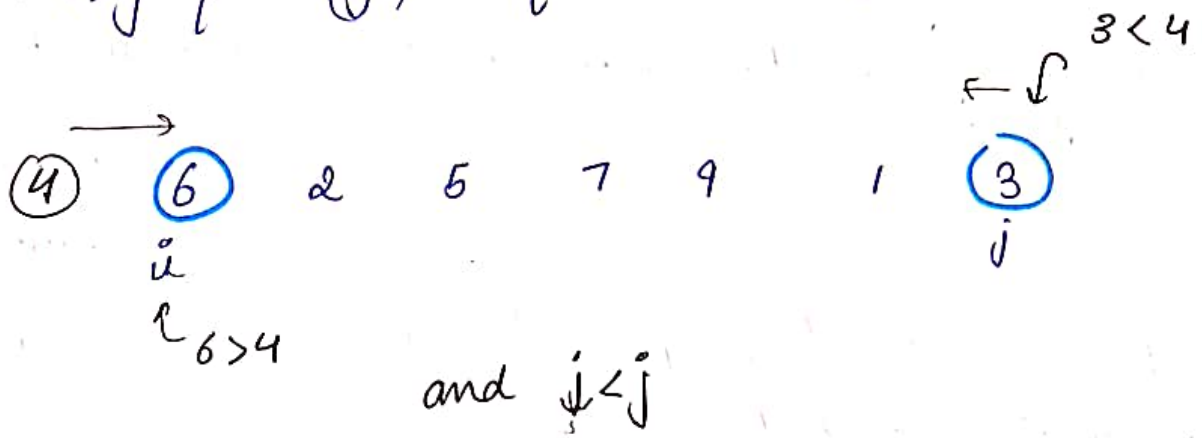
Scan the pointer from left to right till an element is found which is > pivot stop when we encounter such element

So similar thing using j from right to left

look for (element $\leq$ pivot)

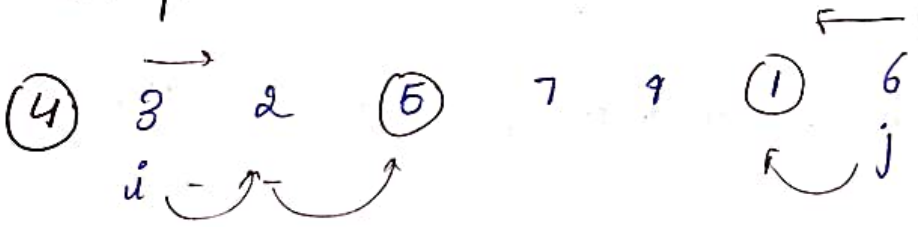
if $i < j$ swap the element

if $i > j$ then (j) is the correct position.

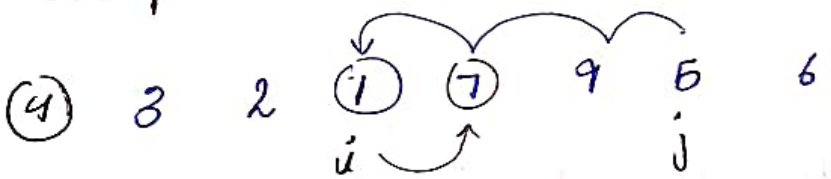


and $i < j$

∴ swap

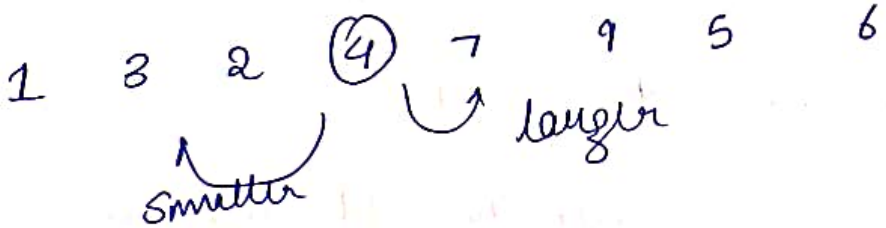


∴ swap

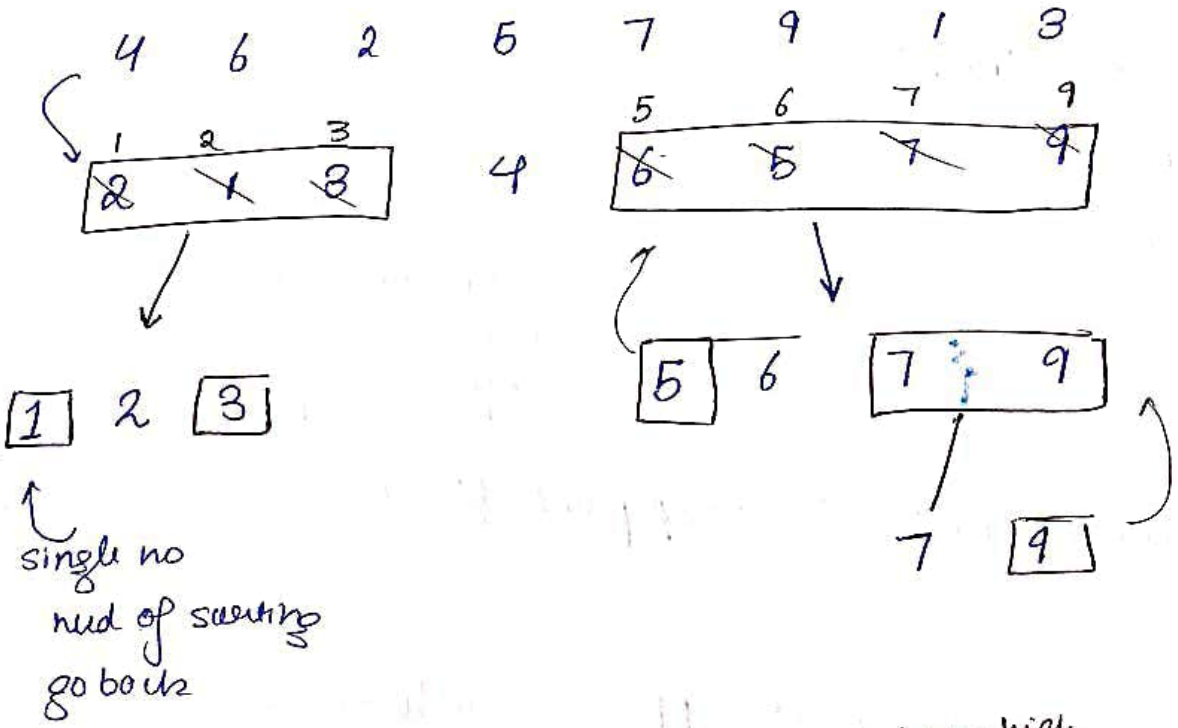


Now $i > j$ ∴ j is correct pos for 4

Now swap pivot with arr[j]



How whole array is sorted



single no
need of sorting
go back

when low = high
only one element ∴ return

PSUEDO CODE

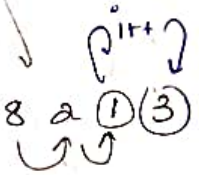
```
qs(arr, low, high){
  if(low < high){
      qs(arr,
partIndex = f(arr, low, high);
      qs(arr, low, partIndex-1);
      qs(arr, partIndex+1, high);

  }

}
```

```
f(arr, low, high){
    pivot = arr[low];
    i = low;
    j = high;
    while(i < j){
```

when on last element of array then index after loop will point to element after the last element.

~~while(arr[i] <= arr[pivot && i <= high~~

```
while(arr[i] <= arr[pivot] && i <= high-1){
        i++
    }
while(arr[j] > pivot && j >= low-1){
        j++;
    }
if(i < j){ swap(arr[j], arr[i]);
    }
}
swap(arr[j], arr[low])
return j;
}
```



means that no element found >8 ∴ last should be position

INSERTION SORT USING RECURSION

element to compare with (index) — i
size — n

```
void insertion_sort ( arr, i, n ) {

    if (n == i) return
        ← when last element is covered.

    int j = i;
    while ( j>0 && arr[j-1] > arr[j] ){
            int temp = arr[j-1];
            arr[j-1] = arr[j];
            arr[j] = temp;
            j--;
    }

    insertion_sort (arr, i+1, n );

}
```

BUBBLE SORT USING RECURSION

My code

→ void bubble(arr[], start, end) {

```
if (start >=0){
  bubble(arr, start-1, end-1);
  if(arr[start] > arr[start+1]) {
      swap
      }
  if (start+1 == end) bubble(arr, start-1, end-1)
}
```

this reduces changes the two elements under consideration

(8 (2) (5) 9)

this reduces size of array

(8 2 5) 9)
J S E

Strivers

```
bubble(arr, n) {
  if(n==1) return;
  for(j=0 → n-2) swap & compare;
  bubble(arr, n-1);
  }
```