TOPIC: Sorting Algorithms

# 1.Selection Sort

LINK: <https://practice.geeksforgeeks.org/problems/selection-sort/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=selection-sort>

## The algorithm steps are as follows:

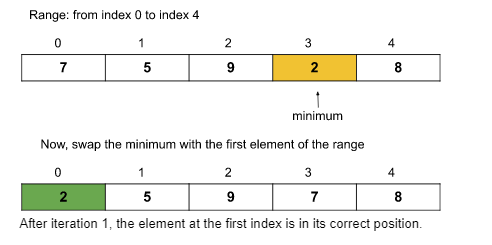
1. First, we will select the range of the unsorted array using a loop (say i) that indicates the starting index of the range.  
   The loop will run forward from 0 to n-1. The value i = 0 means the range is from 0 to n-1, and similarly, i = 1 means the range is from 1 to n-1, and so on.  
   (*Initially, the range will be the whole array starting from the first index.*)
2. Now, in each iteration, we will select the minimum element from the range of the unsorted array using an inner loop.
3. After that, we will swap the minimum element with the first element of the selected range(*in step 1*).
4. Finally, after each iteration, we will find that the array is sorted up to the first index of the range.

## Dry run:

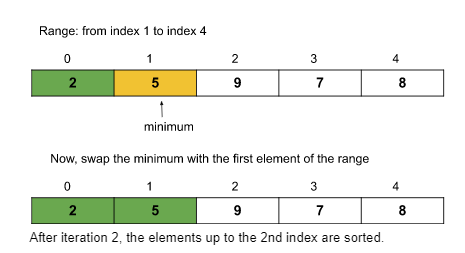
The following dry run will clarify the concepts:

Assume the given array is: {7, 5, 9, 2, 8}

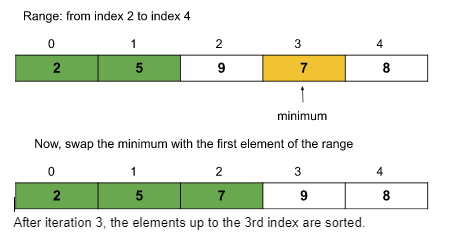
**Outer loop iteration 1:**The range will be the whole array starting from the 1st index as this is the first iteration. The minimum element of this range is 2(**found using the inner loop**).



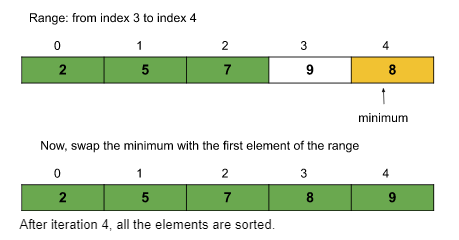
**Outer loop iteration 2:**The range will be from the **[2nd index to the last index]** as the array is sorted up to the first index. The minimum element of this range is 5(**found using the inner loop**).



**Outer loop iteration 3:**The range will be from the **[3rd index to the last index]**. The minimum element of this range is 7(**found using the inner loop**).



**Outer loop iteration 4:**The range will be from the **[4th index to the last index]**. The minimum element of this range is 8(**found using the inner loop**).



So, after 4 iterations(i.e. n-1 iterations where n = size of the array), the given array is sorted.

**Time complexity:**O(N^2)

**Space Complexity: O(1)**

## **CODE:**

void selectionSort(int arr[], int n)

{

//code here

for(int i=0;i<=n-2;i++)

{

int mini=i;

for(int j=i;j<=n-1;j++)

{

if(arr[j]<arr[mini])

mini=j;

}

swap(arr[mini],arr[i]);

}

}

# 2.Bubble Sort

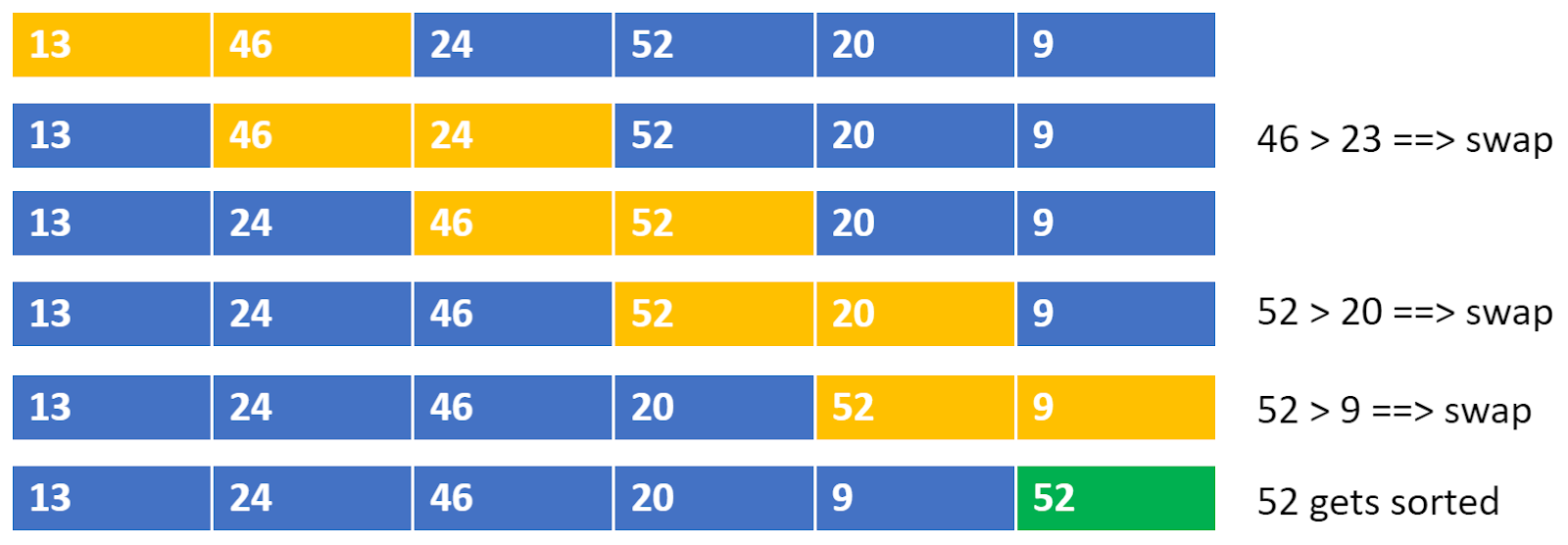
LINK: <https://practice.geeksforgeeks.org/problems/bubble-sort/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=bubble-sort>

## The algorithm steps are as follows:

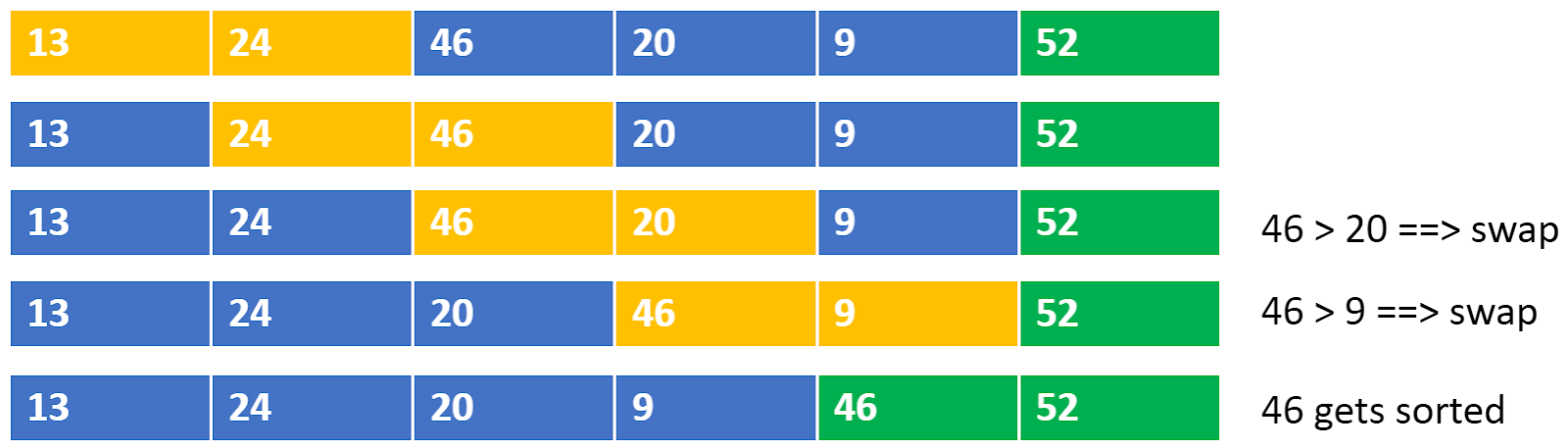
1. First, we will select the range of the unsorted array. For that, we will run a loop(say i) that will signify the last index of the selected range. The loop will run backward from index n-1 to 0(where n = size of the array). The value i = n-1 means the range is from 0 to n-1, and similarly, i = n-2 means the range is from 0 to n-2, and so on.
2. Within the loop, we will run another loop(say j, runs from 0 to i-1 though the range is from 0 to i) to push the maximum element to the last index of the selected range, by repeatedly swapping adjacent elements.  
   Basically, we will swap adjacent elements(**if arr[j] > arr[j+1]**) until the maximum element of the range reaches the end.
3. Thus, after each iteration, the last part of the array will become sorted. Like: after the first iteration, the array up to the last index will be sorted, and after the second iteration, the array up to the second last index will be sorted, and so on.
4. After (n-1) iteration, the whole array will be sorted.

## ****Dry Run:****

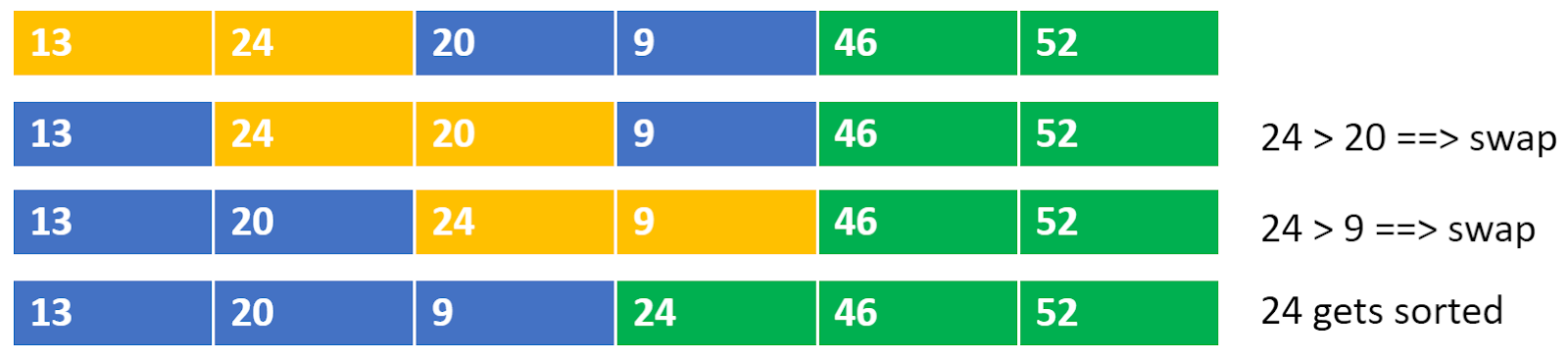
**Iteration 1:**



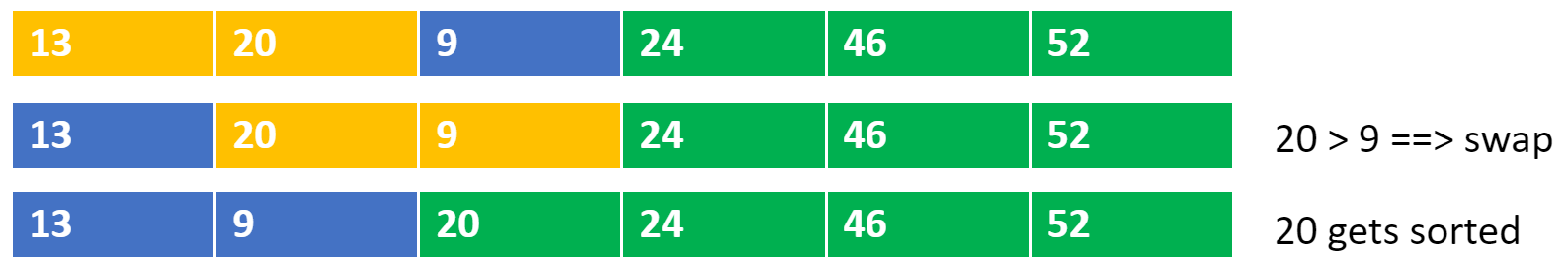
**Iteration 2:**



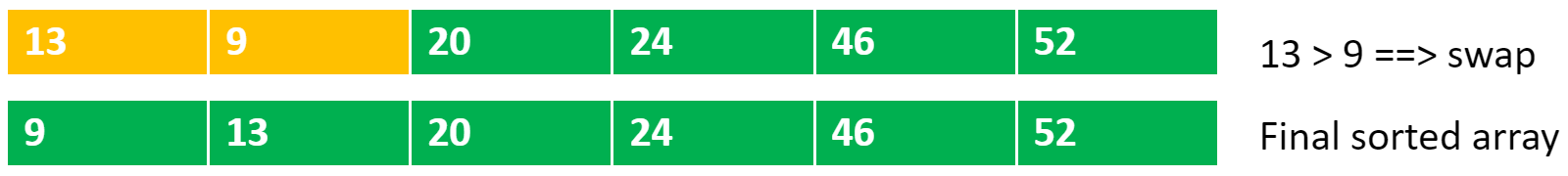
**Iteration 3:**



**Iteration 4:**



**Iteration 5:**



**Time complexity:** O(N^2)

**Space Complexity: O(1)**

## ****CODE:****

* **RECURIVE CODE:**

void Rec\_bubbleSort(int arr[],int i)

{

//Base Case

if(i==0)

return ;

bool didswap=true;

for(int j=0;j<=i-1;j++)

{

if(arr[j]>arr[j+1])

{

swap(arr[j],arr[j+1]);

didswap=false;

}

}

if(didswap)

return ;

return Rec\_bubbleSort(arr,i-1);

}

* **Iterative Code**

void bubbleSort(int arr[], int n)

{

// Your code here

for(int i=n-1;i>=0;i--)

{

bool didswap=true;

for(int j=0;j<=i-1;j++)

{

if(arr[j]>arr[j+1])

{

swap(arr[j],arr[j+1]);

didswap=false;

}

}

if(didswap)

break;

}

# 3.Insertion Sort

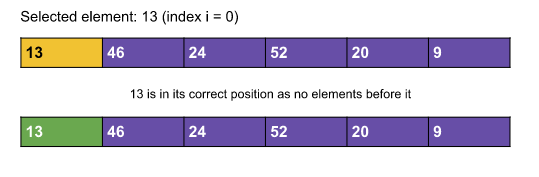
LINK: [https://practice.geeksforgeeks.org/problems/insertion-sort/0?category[]=Algorithms&page=1&query=category[]Algorithmspage1&utm\_source=youtube&utm\_medium=collab\_striver\_ytdescription&utm\_campaign=insertion-sort](https://practice.geeksforgeeks.org/problems/insertion-sort/0?category%5b%5d=Algorithms&page=1&query=category%5b%5dAlgorithmspage1&utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=insertion-sort)

## Approach:

1. Select an element in each iteration from the unsorted array(using a loop).
2. Place it in its corresponding position in the sorted part and shift the remaining elements accordingly (using an inner loop and swapping).
3. The “inner while loop” basically shifts the elements using swapping.

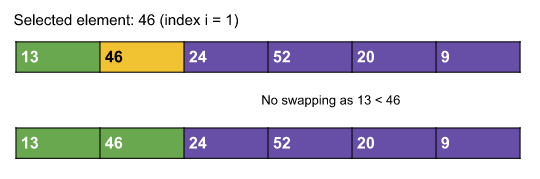
## ****Dry Run:****

**Outer loop iteration 1(selected index i = 0):** The element at index i=0 is 13 and there is no other element on the left of 13. So, currently, the subarray up to the first index is sorted as it contains only element 13.



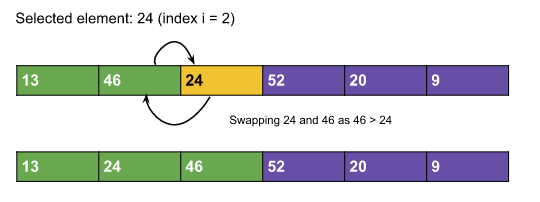
**Outer loop iteration 2(selected index i = 1):**

The selected element at index i=1 is 46. Now, we will try to move leftwards and put 46 in its correct position. Here, 46 > 13 and so 46 is already in its correct position. Now, the subarray up to the second index is sorted.



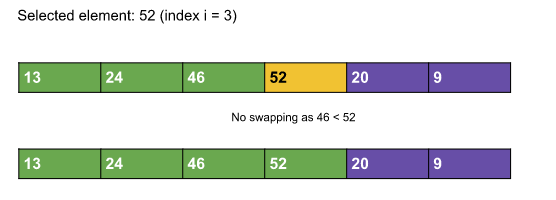
**Outer loop iteration 3(selected index i = 2):**

The selected element at index i=2 is 24. Now, we will try to move leftwards and put 24 in its correct position. Here, the correct position for 24 will be index 1. So, we will insert 24 in between 13 and 46. We will do it by swapping 24 and 46. Now, the subarray up to the third index is sorted.



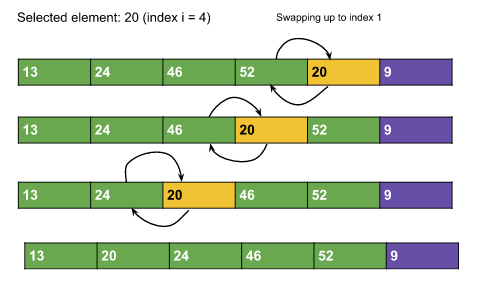
**Outer loop iteration 4(selected index i = 3):**

The selected element at index i=3 is 52. Now, we will try to move leftwards and put 52 in its correct position. Here, the correct position for 52 will be index 3. So, we need not swap anything. Now, the subarray up to the fourth index is sorted.



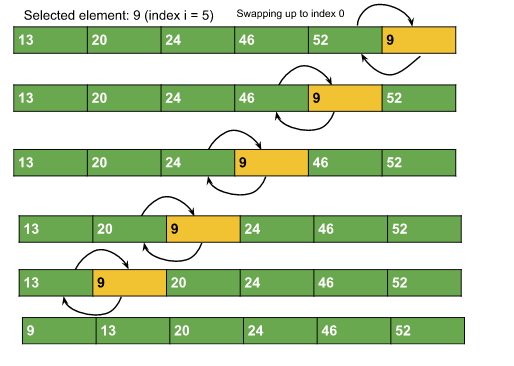
**Outer loop iteration 5(selected index i = 4):**

The selected element at index i=4 is 20. Now, we will try to move leftwards and put 20 in its correct position. Here, the correct position for 20 will be index 1. So, we need to swap adjacent elements until 20 reaches index 1. Now, the subarray up to the fifth index is sorted.



**Outer loop iteration 6(selected index i = 5):**

The selected element at index i=5 is 9. Now, we will try to move leftwards and put 9 in its correct position. Here, the correct position for 9 will be index 0. So, we need to swap adjacent elements until 9 reaches index 0. Now, the whole array is sorted.



**Time complexity:**O(N^2)

**Space Complexity: O(1)**

## ****CODE:****

* REVURSIVE CODE:

void insert(int arr[], int i,int n)

{

//code here

//Base Case

if(i==n)

return ;

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--;

}

insert(arr,i+1,n);

}

* ITERATIVE CODE:

void insertionSort(int arr[], int n)

{

//code here

for(int i=0;i<n;i++)

{

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--;

}

}

}

# 4.Merge Sort

LINK: <https://practice.geeksforgeeks.org/problems/merge-sort/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=merge-sort>

## Intuition:

1. Merge Sort is a **divide and conquers algorithm**, it divides the given array into equal parts and then merges the 2 sorted parts.
2. There are 2 main functions :  
   **merge():**This function is used to merge the 2 halves of the array. It assumes that both parts of the array are sorted and merges both of them.  
   **mergeSort():** This function divides the array into 2 parts. low to mid and mid+1 to high where,

**Approach:**

We will be creating 2 functions mergeSort() and merge().

* **mergeSort(arr[], low, high):**

in mergeSort(), we will divide the array around the middle index(*rather than creating a separate array*) by making the recursive call :  
i. mergeSort(arr,low,mid)   [*Left half of the array*]  
ii. mergeSort(arr,mid+1,high)[*Right half of the array*]  
where low = leftmost index of the array, high = rightmost index of the array, and mid = middle index of the array. Now, in order to complete the recursive function, we need to write the base case as well. We know from step 2.1, that our recursion ends when the array has only 1 element left. So, the leftmost index, low, and the rightmost index high become the same as they are pointing to a single element.  
**Base Case:**if(low >= high) return;

* **merge(arr[], low, mid, high):**

1. In the merge function, we will use a temp array to store the elements of the two sorted arrays after merging. Here, the range of the left array is low to mid and the range for the right half is mid+1 to high.
2. Now we will take two pointers left and right, where left starts from low and right starts from mid+1.
3. Using a while loop( while(left <= mid && right <= high)), we will select two elements, one from each half, and will consider the smallest one among the two. Then, we will insert the smallest element in the temp array.
4. After that, the left-out elements in both halves will be copied as it is into the temp array.
5. Now, we will just transfer the elements of the temp array to the range low to high in the original array.

**Time complexity:**O(nlogn)

Reason: At each step, we divide the whole array, for that logn and we assume n steps are taken to get sorted array, so overall time complexity will be nlogn

**Space complexity:** O(n)

Reason: We are using a temporary array to store elements in sorted order.

## CODE:

void merge(int arr[], int l, int m, int r)

{

// Your code here

int lh=l;

int rh=m+1;

vector<int>temp;

while(lh<=m && rh<=r)

{

if(arr[lh]<=arr[rh])

{

temp.push\_back(arr[lh]);

lh++;

}

else

{

temp.push\_back(arr[rh]);

rh++;

}

}

//If any Element is Left

while(lh<=m)

{

temp.push\_back(arr[lh]);

lh++;

}

while(rh<=r)

{

temp.push\_back(arr[rh]);

rh++;

}

//Putting Back The Element in Array in Sorted Order

for(int i=l;i<=r;i++)

arr[i]=temp[i-l];

}

void mergeSort(int arr[], int l, int r)

{

//code here

//Base Case

if(l>=r)

return ;

int mid=(l+r)/2;

//Recursive Call for Left Half

mergeSort(arr,l,mid);

//Recursive Call for Right Half

mergeSort(arr,mid+1,r);

//Merge Two Half

merge(arr,l,mid,r);

}

# 5.Quick Sort

LINK: <https://practice.geeksforgeeks.org/problems/quick-sort/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=quick-sort>

## ****Intuition:****

This algorithm is basically a repetition of two simple steps that are the following:

* Pick a pivot and place it in its correct place in the sorted array.
* Shift smaller elements(i.e. Smaller than the pivot) on the left of the pivot and larger ones to the right.

 in an array a pivot can be any of the following:

* The first element of the array
* The last element of the array
* Median of array
* Any Random element of the array

## The algorithm steps are the following for the ****quickSort()**** function:

1. Initially, the **low** points to the first index and the **high** points to the last index(as the range is n i.e. the size of the array).
2. After that, we will get the index(*where the pivot should be placed after sorting*) while shifting the smaller elements on the left and the larger ones on the right using a partition() function discussed later.  
   Now, this index can be called the **partition index** as it creates a partition between the left and the right unsorted subarrays.
3. After placing the pivot in the partition index(within the partition() function specified), we need to call the function quickSort() for the left and the right subarray recursively. So, **the range of the left subarray will be [low to (partition index – 1)]** and **the range of the right subarray will be [(partition index + 1) to high].**
4. This is how the recursion will continue until the range becomes 1.

**Time Complexity:**O(N\*logN), where N = size of the array.

**Reason:** At each step, we divide the whole array, for that logN and n steps are taken for the partition() function, so overall time complexity will be N\*logN.

The following recurrence relation can be written for Quick sort :

F(n) = F(k) + F(n-1-k)

Here k is the number of elements smaller or equal to the pivot and n-1-k denotes elements greater than the pivot.

There can be 2 cases :

**Worst Case** – This case occurs when the pivot is the greatest or smallest element of the array. If the partition is done and the last element is the pivot, then the worst case would be either in the increasing order of the array or in the decreasing order of the array.

**Recurrence:**  
**F(n) = F(0)+F(n-1)  or  F(n) = F(n-1) + F(0)**

**Worst Case Time complexity: O(n2)**

Best Case – This case occurs when the pivot is the middle element or near to middle element of the array.  
Recurrence :  
F(n) = 2F(n/2)  
  
Time Complexity for the best and average case: O(N\*logN)

**Space Complexity:**O(1) + O(N) auxiliary stack space.

## CODE:

void quickSort(int arr[], int low, int high)

{

// code here

//Base Case

if(low>high)

return ;

int pt=partition(arr,low,high);

quickSort(arr,low,pt-1);

quickSort(arr,pt+1,high);

}

int partition (int arr[], int low, int high)

{

// Your code here

int i=low;

int j=high;

int pivot=arr[low];

while(i<j)

{

while(arr[i]<=pivot && i<=high-1)

i++;

while(arr[j]>pivot && j>=low+1)

j--;

if(i<j)

swap(arr[i],arr[j]);

}

swap(arr[low],arr[j]);

return j;

}