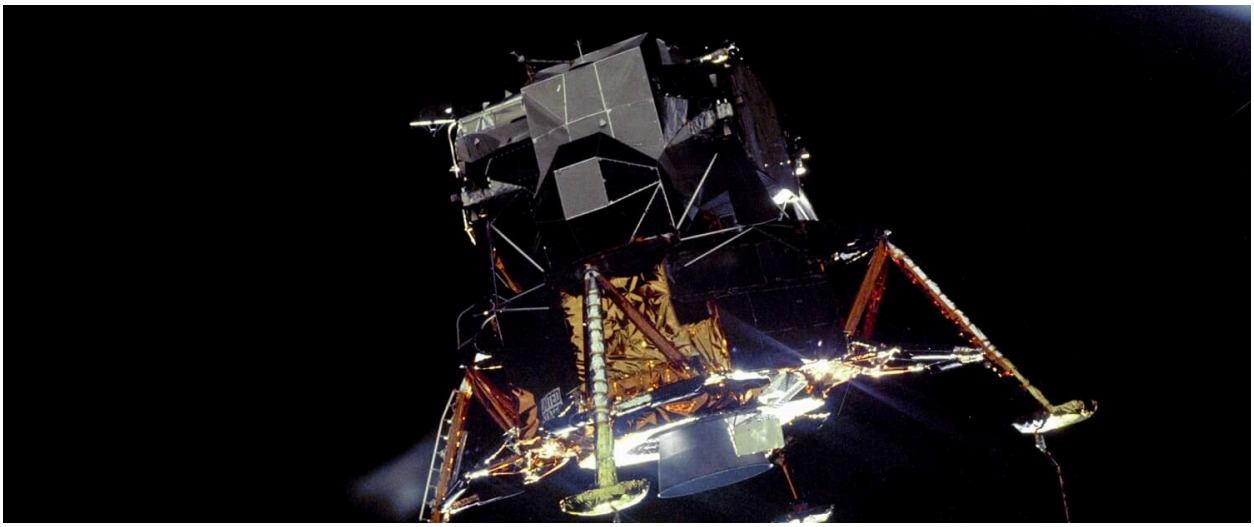


DSA (Data structure algorithm)-Arrays and Sorting Algorithms

```
datatype[] variable name= new datatype[size];

int rollno1=123;// sequence to define an array
```

DSA means storing and storing data in memory

there are two types of arrays (index of array always start from 0 to n).(new keyword is used to create an object).

-Arrays are fixed size and depends on the time of creation. in order to increase the size of array a new array must be created and data must be copied from old array to new.

-Operations

Create=Add/Insert(C)

Search= View/Update(R/U)

Delete=(D)

-Unsorted(Insert: Fastest, Search: slowest, Delete: Fastest) -Sorted (Insert: Slowest, Search: Fastest, Delete: Slowest)

-all search method and function on arrays should return the index of the item if found and -1 otherwise.

-there are two types of search linear and binary search.

*Linear Search

also known as sequential search in which every element is checked.

```
-int LinerarSearch (int[] arr, int numItems, int Key){
for (int i; i < numItems; i++) {
if (arr[i] == key){
return i;
}
return -1;
}
return numItems;
}
```

```
ex))
int arr[] = { 2, 3, 4, 10, 40 };
int x = 10;
// Function call
int result = search(arr, arr.length, x);
if (result == -1)
  System.out.print("Element is not present in array");
else
  System.out.print("Element is present at index " + result);
```

2.Binary Search: also known as interval search. these algorithm structures are more efficient than linear search as they target the center of the search structure and divide the search space in half. the data in Binary search must be sorted. and only single dimensional array must be in use, it is very fast and complex process.

-for binary search the array must be sorted

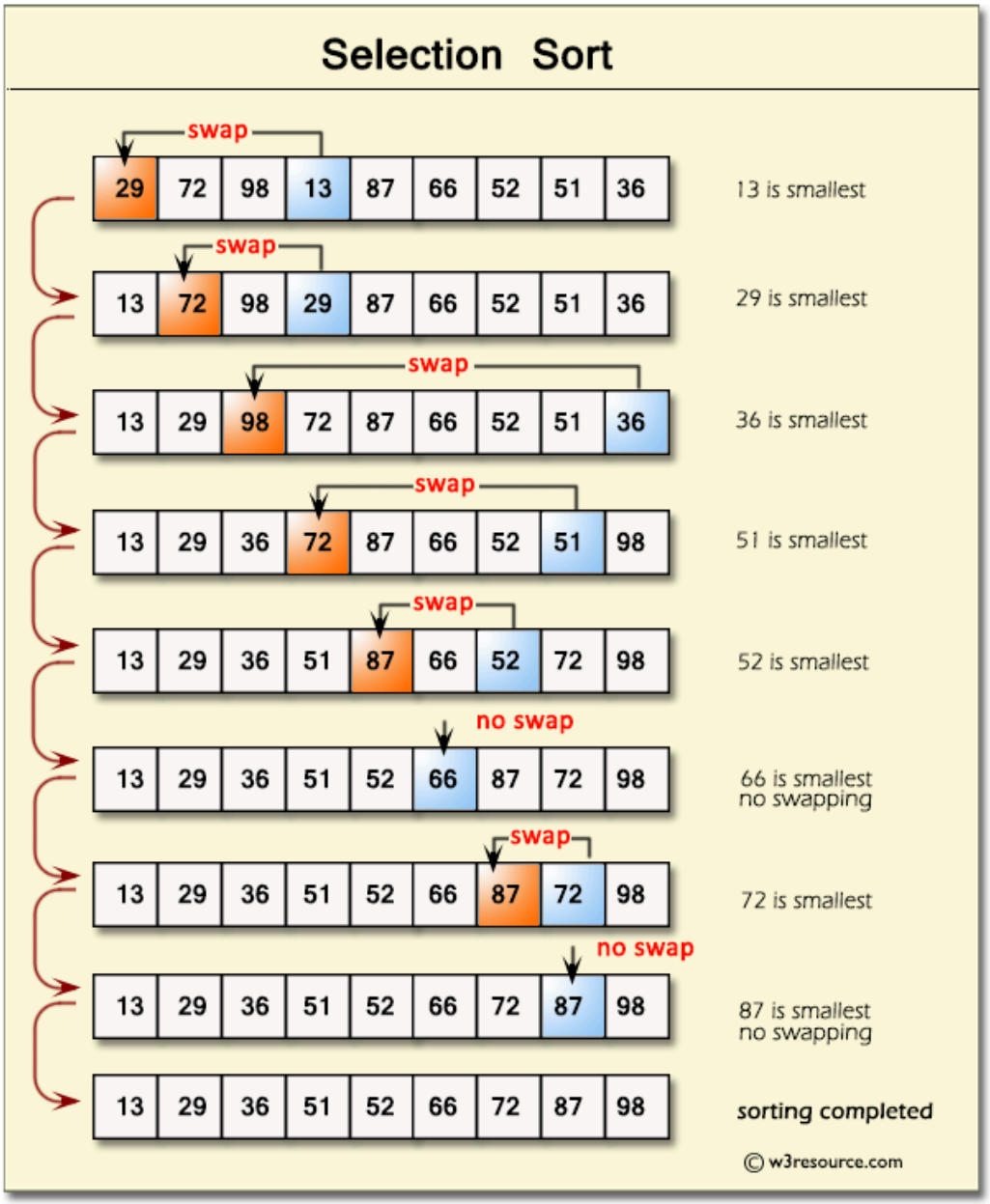
```
public int binarySearch (int key){
int lo=0, hi = numElements-1, mid;
while (lo <= hi) {
   mid = (lo + hi) / 2;
if (m_array[mid]== key)
return mid;
if (key < m_array[mid]) {
hi= mid-1;
}
else {lo=mid+1;
}
}
return -1;
}
}
//example
public static void main(String args[])
   {
      BinarySearch ob = new BinarySearch();
      int arr[] = { 2, 3, 4, 10, 40 };
      int n = arr.length;
      int x = 10;
      int result = ob.binarySearch(arr, x);
      if (result == -1)
         System.out.println(
            "Element is not present in array");
      else
         System.out.println("Element is present at "
                            + "index " + result);
   }
}
```

- There are two types of Binary Search

- Iterative Binary Search Algorithm-*Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.*
- Recursive Binary Search Algorithm-*Create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.*

Sorts

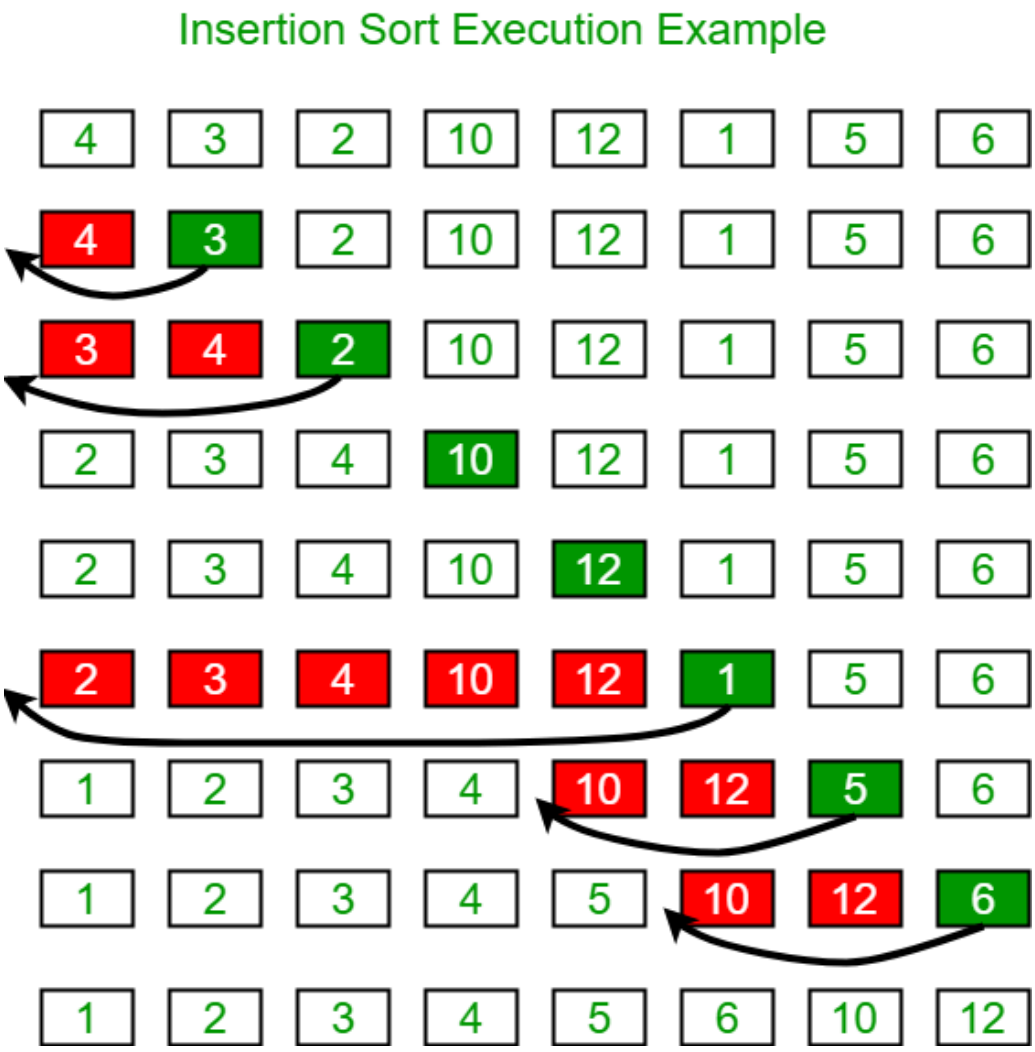
1. Selection Sort-




```
public void selectionsort(){
for (int start=0; start<numElements-1; start++){
int locsmall=start;
for(int start+1=loc; loc<numElements;  loc++){
if (m_array[loc]<m_array[locsmall]){
locsmall=loc;
}
}
```

```
int temp= m_array[start];
m_array[start]=m_array[locsmall];
m_array[locsmall]=temp;
}
}
```

2.Insertion Sort-




```
public void insertionSortAsc(){
        // generate starting positions
        for (int start=1; start <numElements; start++){
            int temp = m_array[start]; // save the value "pull it up"
```

```
        //insert it in order with respect to the items before it.
        int presPos =start-1;
        while (presPos>=0 && m_array[presPos]>temp){
            m_array[presPos+1]=m_array[presPos];
            presPos--;
        }
        m_array[presPos+1]=temp;
    }
}
```

Code to Create an Unordered array and Sort and list array

```
public class UnorderedArray {
private final int[] m_array;
private final int maxSize;
private int numElements;

public UnorderedArray(int size){
    numElements=0;
    maxSize=size;
    m_array= new int [maxSize];
}

public Boolean addLast (int item){
if (numElements<maxSize){
m_array[numElements]=item;
numElements++;
return true;
}
return false;
}

public String ListItems(){
```

```
String s="";
for (int x =0; x<numElements; x++){
s += m_array[x]+ " ";
}
return s;
}
public boolean efficientRemoveItem(int index){

  if (index>=0 && index<numElements){
          m_array[index] =m_array[numElements-1];
          numElements--;
          return true;
       }
       return false;
    }

    public boolean removeItem(int key){
       int loc = linearSearch(key);
       if (loc!=-1){
          return efficientRemoveItem(loc);
       }
       return false;
    }
```

Multidimensional Arrays

```
int [ ][ ] arr= new int [ ][ ]
example)
int [][] arr= new int [{1,2,3},{4,5,6},{7,8,9}];
arr[1]={4,5,6}
but arr[1][0]=4;
```

-individual size of each array vary.

Quick Sort

1. Quick Sort: In this sort the array is unsorted and we are going to pick any pivot from array.

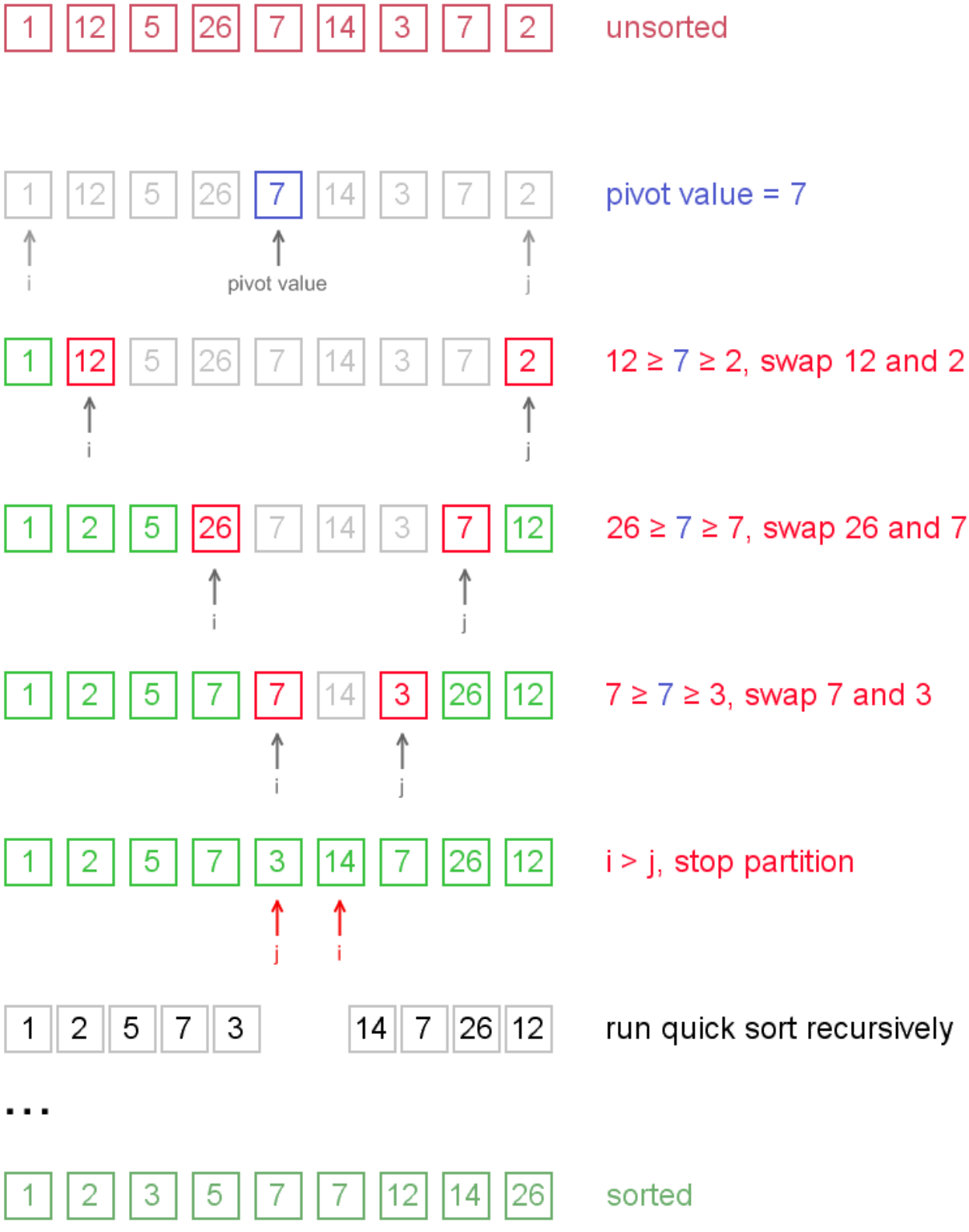
Terminology:

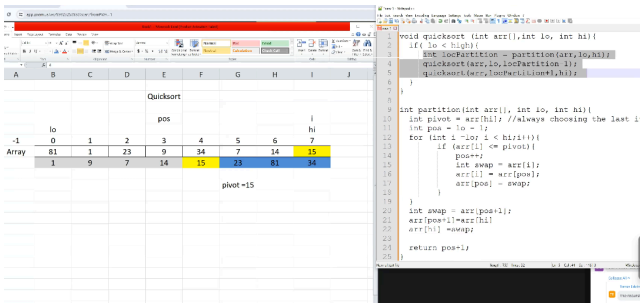
1) choose a pivot.

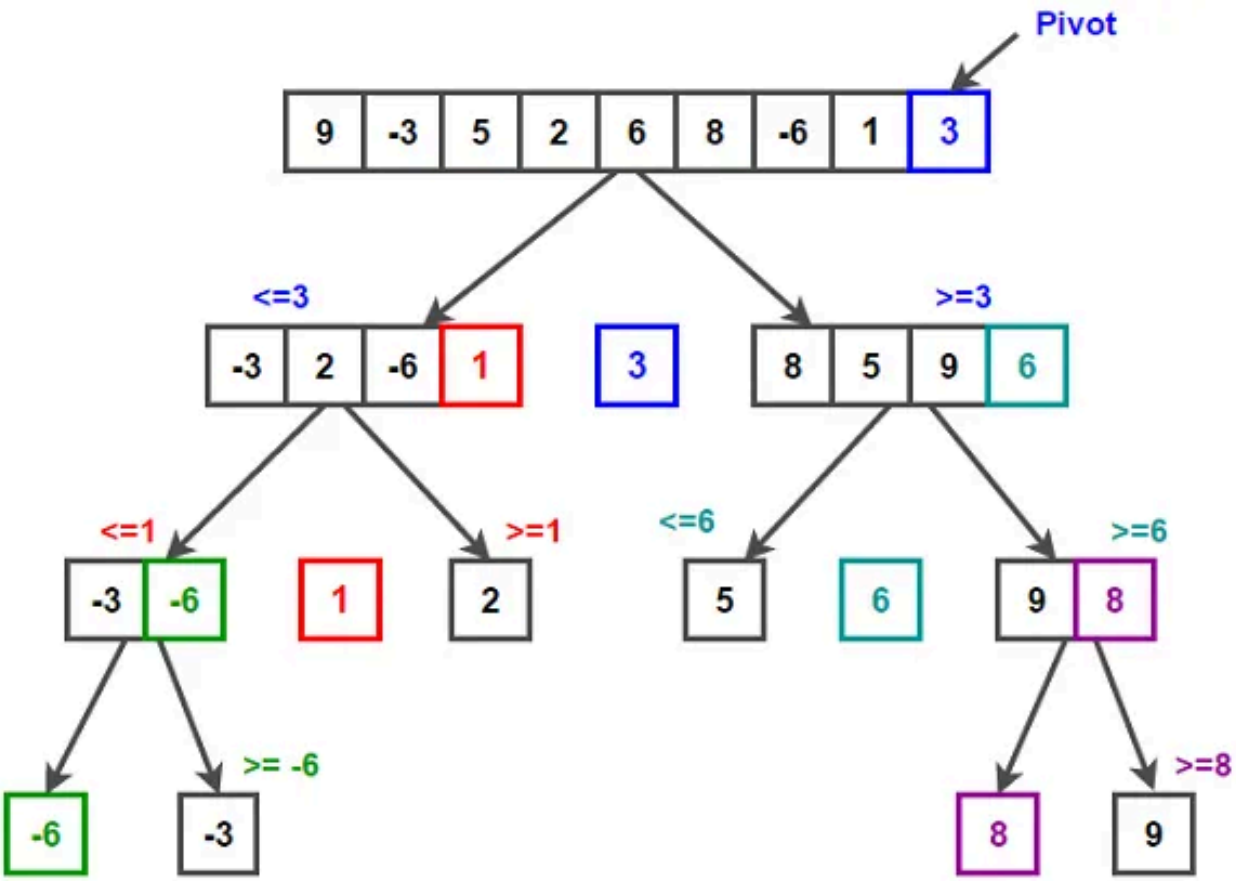
2) group the items less than together and greater than together.

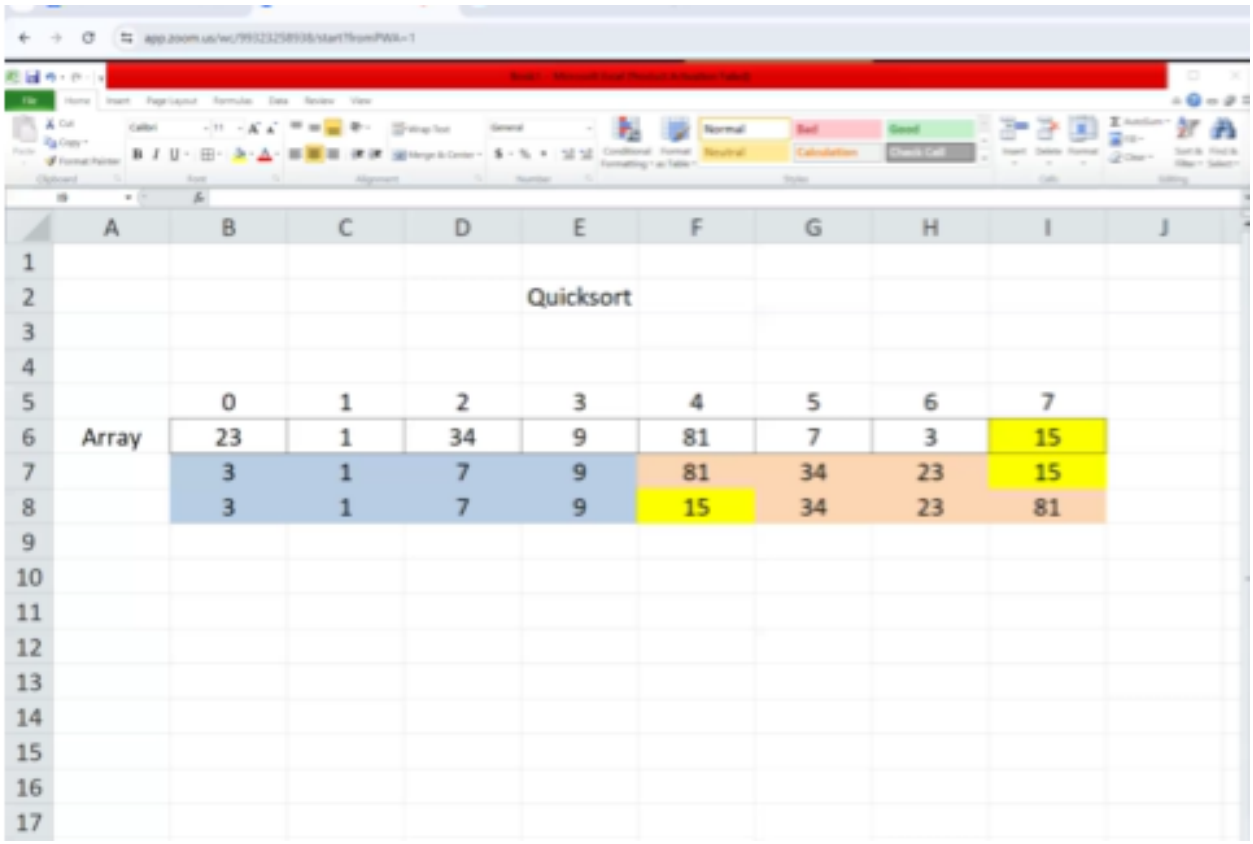
3)Swap the pivot item with the first item in the “greater than “ group (if sorting in ascending order).

4)


1 12 5 26 7 14 3 7 2
unsorted
1 12 5 26 7 14 3 7 2
i
pivot value
j
pivot value = 7
1 12 5 26 7 14 3 7 2
i
j
12 ≥ 7 ≥ 2, swap 12 and 2
1 2 5 26 7 14 3 7 12
i
j
26 ≥ 7 ≥ 7, swap 26 and 7
1 2 5 7 7 14 3 26 12
i
j
7 ≥ 7 ≥ 3, swap 7 and 3
1 2 5 7 3 14 7 26 12
j
i
i > j, stop partition
1 2 5 7 3
14 7 26 12
run quick sort recursively
...
1 2 3 5 7 7 12 14 26
sorted

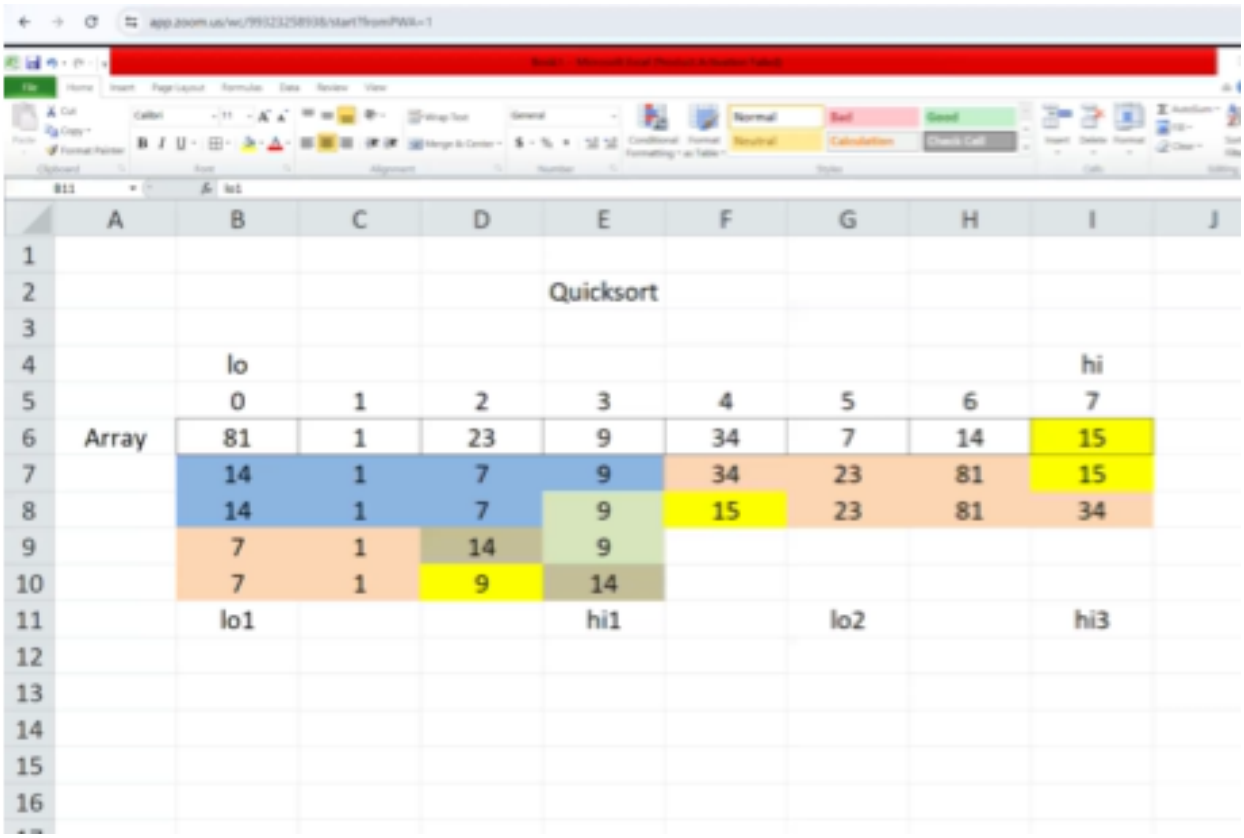

Quicksort
pos
lo
hi
i
Array
pivot =15
void quicksort (int arr[],int lo, int hi){
if(lo < high){
int locPartition = partition(arr,lo,hi);
quicksort(arr,lo,locPartition-1);
quicksort(arr,locPartition+1,hi);
}
}
int partition(int arr[], int lo, int hi){
int pivot = arr[hi]; //always choosing the last i
int pos = lo - 1;
for (int i =lo; i < hi;i++){
if (arr[i] <= pivot){
pos++;
int swap = arr[i];
arr[i] = arr[pos];
arr[pos] = swap;
}
}
int swap = arr[pos+1];
arr[pos+1]=arr[hi]
arr[hi] =swap;
return pos+1;
}


Pivot
9 -3 5 2 6 8 -6 1 3
<=3
>=3
-3 2 -6 1
3
8 5 9 6
<=1
>=1
<=6
>=6
-3 -6
1
2
5
6
9 8
>= -6
>=8
-6
-3
8
9



Quicksort

		0	1	2	3	4	5	6	7
Array		23	1	34	9	81	7	3	15
		3	1	7	9	81	34	23	15
		3	1	7	9	15	34	23	81



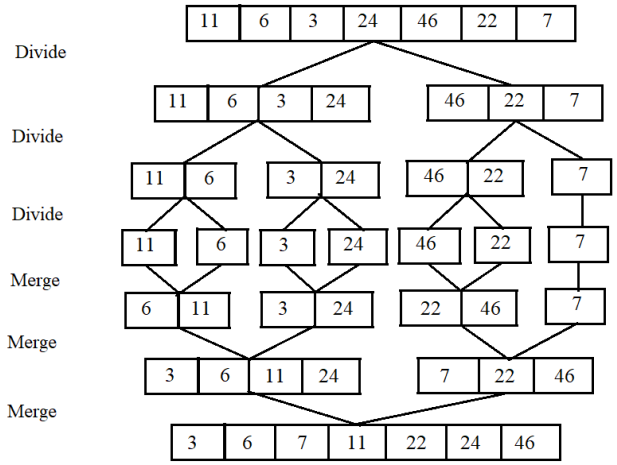
Quicksort

		lo							hi
		0	1	2	3	4	5	6	7
Array		81	1	23	9	34	7	14	15
		14	1	7	9	34	23	81	15
		14	1	7	9	15	23	81	34
		7	1	14	9				
		7	1	9	14				
		lo1			hi1		lo2		hi3

```
public void quickSortAsc(){
        quickSortWorker(0,numElements-1);
    }

    private void quickSortWorker(int lo,int hi){
        if (lo < hi){
            int pivotPoint = partition(lo,hi);
            quickSortWorker(lo,pivotPoint-1);
            quickSortWorker(pivotPoint+1,hi);
        }
    }

    private void swap(int location1,int location2){
        int temp =m_array[location1];
        m_array[location1] = m_array[location2];
        m_array[location2 ]= temp;
    }

    private int partition (int lo, int hi){
        int pivot = m_array[hi]; // always picks hi as the pivot
        int marker= lo-1,temp; // This marks the last item in the left partition
        for (int presPos=lo;presPos<hi;presPos++){
            if (m_array[presPos] < pivot){  // only moves when an item that is smaller
                marker++;
                //swap(marker,presPos);
                temp =m_array[marker];
                m_array[marker] = m_array[presPos];
                m_array[presPos ]= temp;
            }
        }
        temp =m_array[marker+1];
        m_array[marker+1] = m_array[hi];
        m_array[hi ]= temp;
        return marker+1;
    }
```

```
}
```

2. Merge Sort

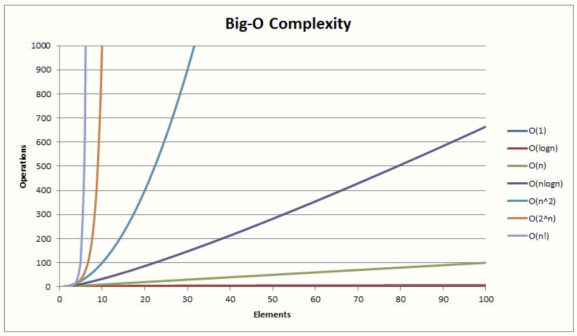



```
public void mergeSortAsc(){
      mergeSortWorker(0,numElements-1);
   }

   public void mergeSortWorker (int left, int right){
       // base case stop when we have 1 item
      if (left<right){
         int mid = (left+right)/2;
         mergeSortWorker(left,mid);
         mergeSortWorker(mid+1,right);
         merge(left,mid,right);
      }
   }

   public void merge(int left,int mid, int right){
      int[] leftArray = new int[mid-left+1];
      int[] rightArray = new int[right-mid];
```

```
    for (int x=0;x<leftArray.length;x++){
        leftArray[x]=m_array[left+x];
    }

    for (int x=0;x<rightArray.length;x++){
        rightArray[x]=m_array[mid+1+x];
    }
    // now merge
    int p1=0,p2=0,pM=left;

    while (p1<leftArray.length && p2 <rightArray.length){
        if (leftArray[p1]<rightArray[p2]){
            m_array[pM]=leftArray[p1];
            p1++;
        }else{
            m_array[pM]=rightArray[p2];
            p2++;
        }
        pM++;
    }

    while (p1<leftArray.length ){
        m_array[pM]=leftArray[p1];
        p1++;
        pM++;
    }

    while (p2<rightArray.length ){
        m_array[pM]=rightArray[p2];
        p2++;
        pM++;
    }
}
```

Time complexity of all sorts algorithm

BigO Graph



*Correction:- Best time complexity for TIM SORT is O(nlogn)

> *SORTING ALGORITHMS RANKED FROM SLOWEST TO FASTEST*

- Bubble sort
- Revised bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Merge sort

Time complexity Cheat Sheet

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

BigO Graph