TUTORIAL → 3

① Search (ar, n, x) {

```
if (ar[n-1] == x)
    {
    return "found";
    }
int flag = ar[n-1];
ar[n-1] = x;
for(int i=0; ; i++){
  if(ar[i] == x){
    ar[n-1] = flag;
    if(i<n-1)
      {
      return "found";
      }
    else
      return "Not found";
  }
}
```

}

② Algorithm

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$
Quick Sort	$\Omega(n\log(n))$	$\Theta(n\log n)$	$O(n^2)$
Merge Sort	$\Omega(n\log(n))$	$\Theta(n\log n)$	$O(n\log n)$

Page →

(4)

→ Bubble sort, insertion sort, selection sort are in-place sorting algorithm.

→ Bubble sort and insertion sort can be applied as stable algorithm.

→ Merge sort is stable but not in-place algorithm

→ Insertion sort is also used for online sorting.

→ Quick sort is an in-place but not stable.

→ Heap sort is in-place but not stable.

→ Selection sort is also an online sorting algorithm.

(5) Recursive

```
int binary(int arr[], int low, int high, int x)
{
    if (low > high) {
        return -1;
    }
    int mid = (low + high)/2;
    if (x == arr[mid])
        return mid;
    else if (x < arr[mid])
        return binary(arr, low, mid-1, x);
    else
        return binary(arr, mid+1, high, x);
}
```

Iterative

```
int binary(int arr[], int n, int x){
  int low=0, high=n-1;
  while(low<=high)
  {int mid=(low+high)/2;
   if (x==arr[mid])
        return mid;
   else if (x<arr[mid])
        high=mid-1;
   else
        low=mid+1;
   }
}
```

Time complexity of Linear Search → O(n)
Binary Search → θ(log n)

(b) Time complexity : Best case → O(1)
Average → O(n log n)
Worst → T(n) = T(n/2) + C

Recurrence Relation

$$T(n) = \begin{cases} C & \text{if } i>=j \\ T(n/2)+C & \text{otherwise} \end{cases}$$

(c)
```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
      a[i]+a[j]=k;
  }
}
```

(8) Quick sort is the best sorting algorithm for practical use our most widely used sorting algorithm at present

→ Has running time of $O(n^2)$ that maker it best in real-time applications.

→ Most often than not runs at $O(n \log n)$

→ High space efficiency by executing in place.

(9) Inversion count for an array indicates how far the array is form being sorted. If array is already sorted, then the inversion count is 0. But if the array is sorted in the reverse order, the inversion count is maximum.

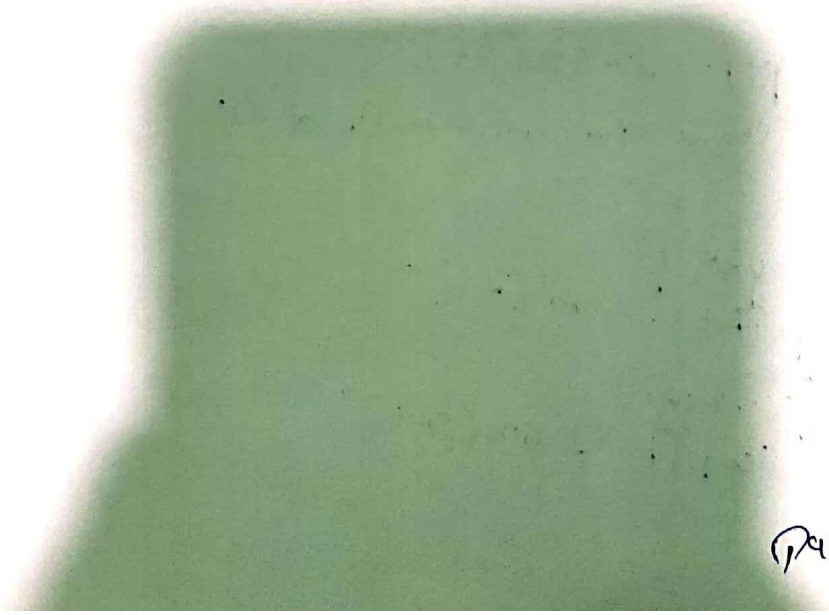
```
int merge (int arr[], int temp[], int left, int mid, int right){
  int i, j, k;
  int inv_count[];
  i = left, j = mid, k = left;
while((i<=mid-1) && (j<=right)){
  if (arr[i]<=arr[j]){
      temp[k++]=arr[i++];
  }
else {
      temp[k++]= arr[j++];
      inv_count= inv_count++ (mid-1);
}}
while(i<=mid-1)
    temp [k++] = arr [i++];
while(j<=right)
    temp[k++]=arr[j++];
```

```
for(i=left; i<=right; i++)
    arr[i]=temp[i];
  return inv_count;
}

int mergesort (int arr[], int arr_size){
 int temp[arr_size];
 return merge_count (arr, temp, 0, arr_size-1);
}
int merge_count(int arr[], int temp[], int left, int right)
{ int mid;
  if(right > left){
   mid=(right+left)/2;
   inv = merge_count (arr, temp, left, mid);
   inv = merge_count(arr, temp, mid+1, right);
   inv = merge(arr, temp, left, mid+1, right);
  }
  return mid;
}
```

(10) The worst case occurs when the Partition process always picks up greatest or smallest element as pivot.

The Best case occurs when the Partition process always picks the middle element as pivot



(17) void stable selection_sort (int a[], int n)

```
{
  for(int i=0 ; i<n-1; i++)
    {
      int min=i;
    for(int j=i+1 ; j<n; j++)
       if (a[min]>a[j])
          min=j;
        int key=a[min];
        while(min>i){
        a[min]==a[min-1];
        min--;
      }
      a[i]=key;
}
```

(18) for sorting 4GB of data using 2 GB RAM:

→ Read 2GB of data in main m/m and sort it by using quick sort.

→ Write sorted array to disk

→ Repeat it all until all data is in sorted 2GB chunks i.e. 4/2 = 2 chunks.

→ which now need to be merged in single file.

→ Perform 2 way merge and store the result in output buffer [needs 200 MB from each sorted chunk into both input and allocation for output buffer].