TUTORIAL - 3

Question 1

```
function length_search(array, element):
    n = length of array
    for i from 0 to n-1
        if array[i] == element:
            return i
        else if array[i] > element:
            break
    return "element not found"
```

Question 2

Iterative code

```
function insertion_sort(array):
    n = length of array
    for i from 1 to n-1:
        key = array[i]
        j = i - 1
        while j >= 0 && array[j] > key:
            array[j+1] = array[j]
            j = j - 1
        array[j+1] = key
    return array
```

Recursive code

```
function recursive(array, n)
    if n <= 1
        return
    recursive(array, n-1)
    key = array[n-1]
```

```
j = n-2
while j >= 0 and array[j] > key :
    array[j+1] = array[j]
    j = j-1
array[j+] = key
```

Insertion sort doesn't need to know anything about what values it will sort & the information is requested while the algo is running.

3- (i) Selection Sort
TC - Best Case : $O(n^2)$: Worst Case = $O(n^2)$
SC = (O(1))

(ii) Insertion sort.
TC - Best case = O(n) Worst case = $O(n^2)$
SC = O(1)

(iii) Merge Sort
TC = Best Case = O(nlogn) Worst case = O(nlogn)
SC = O(n)

(iv) Quick Sort.
TC = Best Case O(nlogn) Worst Case = $O(n^2)$
SC = = O(n)

(v) Heap Sort
TC = Best Case - O(nlogn) Worst Case = O(nlogn)
SC = O(1)

(vi) Bubble Sort -
TC - Best Case - $O(n^2)$ Worst Case - $O(n^2)$
SC = O(1)

4)

Sorting	Inplace	Stable	Online
Selection	✓		
Insertion	✓	✓	✓
Merge		✓	
Quick	✓		
Heap	✓		
Bubble	✓	✓	

5) Iterative code

```
int bin_search (int ar[], int l, int r, int x)
{
  while (l<=r)
  {
    int m = (l+r)/2;
    if (ar[m] = x)
      return m;
    if (ar[m]<x)
      l = m+1;
    else
      r = m-1;
  }
  return -1;
}
```

T.C

Best case = O(1)
Avg case = O(log n)
Worst case = O(log n)

Recursive code

```
int bin_search (int ar[], int l, int r, int x)
{
  if (r>=1)
  {
    int mid = (l+r)/2;
    if (ar[mid] = x)
      return mid;
    else if (ar[mid] > x)
      return bin_search (ar, l, mid-1, x);
    else
      return bin_search (ar, mid+1, r, x)
  }
  return -1;
}
```

TC

B.C = O(1)
AVG = O(log n)
WORST = O(log n)

6) Recurrence Relation

$T(n) = T(n/2) + 1$

(8) Quick Sort is the fastest general purpose sort used when you need an in-place sorting algorithm with good average-case performance especially for large datasets where memory usage is a concern

(9) How far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if array is sorted in reverse order, the inversion count is max

Eg - For the given array Total inversions are 18
(7,8), (21,31) (8,10) (1,20) (1,6) (1,4) (1,5)
(20,6) (20,4) (20,5)

10) The worst case time complexity of quick sort is $O(n^2)$ the worse case occurs when the pivot is always on the extreme element. This happens when input array is sorted or reverse sorted & either first or last element is picked as pivot
Best case - Pivot is a mean element

11) Recurrence Relation -
(a) merge → $T(n) = 2T(n/2) + n$
(b) Quick → $T(n) = 2T(n/2) + n$

merge sort is more efficient & works faster than quick sort.
Worst case Complexity Quick Sort = $O(n^2)$
merge sort = $O(n \log n)$

12)

```
void stab_selsort(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;
   }
}
```

13) External sorting - If the input data is su
that it cannot adjust in the memory entirel
at once, it needs to be stored in a hard disk. et

- Internal sorting - If input data is such tha
it can adjust in the main memory at
once