

TUTORIAL - 3

Question 1
function length-search(array, element):

$n = \text{length of array}$

for i from 0 to $n-1$

if $\text{array}[i] == \text{element}$:

return i

else if $\text{array}[i] > \text{element}$:

break

return "Element not found"

Question 2

Iterative Code

function insertion-sort(array):

$n = \text{length of array}$

for i from 1 to $n-1$:

key = $\text{array}[i]$

$j = i - 1$

while $j \geq 0$ && $\text{array}[j] > \text{key}$:

$\text{array}[j+1] = \text{array}[j]$

$j = j - 1$

$\text{array}[j+1] = \text{key}$

return ~~array~~

Recursive Code

function recursive(array, n):

if $n \leq 1$

return

recursive(array, $n-1$)

key = $\text{array}[n-1]$

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 = $O(\log_2 n)$

Worst case = $O(\log_2 n)$

Recursive code

```

int bin-search(int ar[], int l, int r, int x)
{
    if (r >= l)
    {
        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;
}

```

T.C

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.

For the given array Total inversions are 18
eg - (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 worst 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 median element.

11) Recurrence Relation -

(a) merge $\rightarrow T(n) = 2T(n/2) + n$

(b) Quick $\rightarrow T(n) = 2T(n/2) + n$

merge sort is more efficient & works faster than quick sort.

Worst case Complexity Quick Sort = $O(n^2)$
merge Sort = $O(n \log n)$

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

$j = j - 1$
 $array[j+1] = key$

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 = $O(n)$

SC = $O(1)$

Worst case = $O(n^2)$

(iii) Merge Sort

TC = Best Case = $O(n \log n)$

SC = $O(n)$

Worst case = $O(n \log n)$

(iv) Quick Sort.

TC = Best Case = $O(n \log n)$

SC = $O(n)$

Worst case = $O(n^2)$

(v) Heap Sort

TC = Best Case = $O(n \log n)$

SC = $O(1)$

Worst case = $O(n \log n)$

(vi) Bubble Sort -

TC - Best Case = $O(n^2)$

SC = $O(1)$

Worst case = $O(n^2)$

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

12) void stab_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 > 0)
        {
            a[min] = a[min - 1];
            min--;
        }
        a[i] = key;
    }
}
```

13) External Sorting - If the input data is such that it cannot adjust in the memory entirely at once, it needs to be stored in a hard disk etc.

- Internal Sorting - If input data is such that it can adjust in the main memory at once.