DESIGN AND ANALYSIS OF ALGORITHM

Name - Gautam
Goyal
Section - 051
Roll no - 50

Assign

Ans 1 pseudocode of linear search

```
function linearsearch( array, target):
    n= length(array)
   for i from 0 to n-1:
       if array[i] == target:
          return i
       else if array[i] > target:
          return "Element not found"

   return "Element not found"
```

Ans 2 Iterative insertion sort:

```
function iteractive insertionsort (array):
    n = length (array)
   for i from 1 to n-1:
       key = array[i]
       j=i-1
      while j>=0 and arr[j] > key:
          array[j+1] = array[j]
          j=j-1
      array[j+1] = key
return array
```

Recursive insertion sort:

```
function recursiveinsertionsort( array, n):
    if n<=1:
        return array
    recursiveinsertionsort (array, n-1)
    key = arr[n-1]
    j=n-2
    while j>=0 and array[j] > key
        array[j+1] = array[j]
        j=j-1
    array[j+1] = key
    return array
```

Insertion sort is called online sorting because it can sort a list as it recives it, without needing to have all the elements beforehand. This is particularly useful in scenarios where input data is continuously arriving, and you want to maintain a sorted list in real time.

Ans 3.

Time complexity:

1. Bubble sort : $O(n^2)$
2. Selection sort : $O(n^2)$
3. Insertion sort : $O(n^2)$ average & worst case
   $O(n)$ best case
4. merge sort : $O(n \log n)$
5. Quick sort : $O(n \log n)$ avg case time
   $O(n^2)$ worst case

Radix sort $O(d * (n+k))$ d · number of digits
n number of input number elements
k is the range of input

Count sort $O(n+k)$: n number of elements in the input array
k is range of the input

Ans 4.

	inplace	stability	online
Bubble sort	✓	✓	x
Selection sort	✓	x	x
Insertion sort	✓	✓	✓
merge sort	x	✓	x
quick sort	✓	x	x
Radix sort	x	✓	x
Count sort	x	✓	x

Ans 5.

```
function binarysearch ( array, target, low, high):
  if low > high:
      return "Element not found"
  mid = (low + high)/2
  if array[mid] == target:
      return mid;
  else if array[mid] < target:
      return binarysearch (array, target, mid+1, high)
  else:
      return binarysearch (array, target, low, mid-1)
```

Iterative binary search:

```
function binarySearch(array, target):
    low = 0
    high = length(array) - 1
    while low <= high:
        mid = (low + high) / 2
        if array[mid] == target:
            return mid
        else if array[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return "Element not found"
```

Time & space complexity:

Linear search:

Time complexity: O(n)

Space complexity: O(1)

Binary search:

Recursive Time complexity: O(log n)

Space complexity: O(log n)

Iterative Time complexity: O(log n)

Space complexity: O(1) constant space

Ans 6

Recurrence Relation for binary recursive search

Let T(n) be the time taken by binary search for a sorted array of size n. In each recursive call, the problem size is

halved.

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

Ans 7

a(n) complexity using a two-pointer approach

```
function findindexwithsum (array, k):
    left = 0
    right = length(array) - 1
    while left < right:
        current_sum = array[left] + array[right]
        if current_sum == k:
            return left, right
        else if current_sum < k:
            left = left + 1
        else:
            right = right - 1
    return "NO such pair exists"
```

Ans 8 quick sort.

The best sorting algorithm for practical use depend on various factors such as the size of the dataset, distribution of data, available memory, and whether the data is partially sorted or not.

quick sort is efficient for many choices. It has an avg TC O(logn). which is very efficient It has good cache memory due to its locality of reference.

Page 9

Inversion in an array occur when there are pairs of elements that are out of order relative to each other.

// Total 17 inversions

```
mergesort(arr):
    if len(arr) <= 1
        return arr, 0
    mid = len(arr) // 2
    left, inv_left = mergesort(arr[0:mid])
    right, inv_right = mergesort(arr[mid:])
    merged, inv_merge = merge_with_inversions(left, right)
    return merged, inv_left + inv_right + inv_merge

merge_with_inversions(left, right):
    result = []
    inversions = 0
    i = j = 0
    while i < length(left) and j < length(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i = i + 1
        else:
            result.append(right[j])
            j = j + 1
            inversions = inversions + (length(left) - i)
    result.extend(left[i:])
    result.extend(right[j:])
    return result, inversions

arr [7, 21, 31, 8, 10, 1, 20, 6, 4, 5]

sorted_arr, inversions = mergesort(arr)
```

Ans 10

Best case quick sort gives it best case time complexity $O(n \log n)$ whereas the pivot chosen in each partition steps divides the array into two nearly equal halves. This occurs when the pivot is the median element of the array or its subarray.

Worst case: Quick sort gives it worst case time complexity $O(n^2)$ when pivot selection consistently results in highly unbalanced partitions such as when the smallest or largest element is always chosen as the pivot.

Ans 11:

Merge sort:

Best case: $O(n \log n)$

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

Quick sort:

Best case: $O(n \log n)$

Worst case: $O(n^2)$

similiarities:

Both merge sort & quick sort have a best case time complexity of $O(n \log n)$ when the input is uniformly distributed or when a good pivot is chosen

Both algorithm are comparison-based sorting algorithm.

Differences:

quick sort is generally faster in practice, merge sort is more stable in terms of time complexity.

merge sort is a stable sorting algorithm. quick sort in its standard implementation is not stable.

merge sort requires additional space proportional to the size of the input array for merging. whereas quick sort is in place sorting algorithm

Ans 12 Stable selection sort

```
function stableselectionsort(array):
    n = length(array)
    for i from 0 to n-1:
        minindex = i
        for j from at 1 to n-1:
            if array[j] < array[minindex]:
                minindex = j
        minval = array[minindex]
        while minindex > i:
            array[minindex] = array[minindex-1]
            minindex = minindex - 1
        array[i] = minval
    return array
```

Ans 13

```
function bubblesort(array):
    n = length(array)
    swapped = true
    while swapped:
        swapped = false
        for i from 0 to n-2:
            if array[i] > array[i+1]:
                swap(array[i], array[i+1])
                swapped = true
        n = n - 1
    return array
```

Ans 14 Choosing sorting algorithm for large ~~data~~ on limited ram

When the data size exceeds the available RAM, external sorting algorithms are typically used.

Internal sorting: sorting algorithm that operates entirely within the computer's main memory (RAM) are called internal sorting algorithm.

External sorting: sorting algorithm designed that are too large to fit entirely into memory are called external sorting algorithms.