Design and Analysis of Algorithms

Name - Chandni Mehta
Roll no - 38
University Rno - 2021676

Ans (1)

```
#include <iostream>
#include <vector>
using namespace std;
int linear_search_sorted (const vector<int> &arr, int target)
{ int n= arr.size();
  int index = 0;
  while (index <n && arr[index]<= target)
  { if (arr [index] == target)
    { return index;
    }
   index++;
  }
  return -1;
}
```

Ans - (2) Iterative Insertion sort →

```
Void iterative_insertion_sort ( vect <int> &arr)
{ int n= arr.size();
  for (int i= 1; i<n ; i++)
    { int key = arr[i];
      int j = i-1;
      while (j>=0 && arr[j] >key)
        { arr[j+1]=arr[j];
          j=j-1;
        }
      arr [j+1] = key;
    }
}
```

Recursive insertion sort -

```
void recursive_insertion_sort(vector<int> &arr, int n)
{ if(n<=1)
  {return;
  }
  recursive_insertion_sort(arr, n-1);
  int key = arr[n-1];
  int j = n-2;
  while (j>=0 && arr[j] > key)
  { arr[j+1] = arr[j];
    j = j-1;
  }
  arr[j+1] = key;
}
```

→ Insertion sort is considered an online sorting Algorithm.

① Incremental sorting - As new elements arrive, they can be inserted into their correct positions in sorted array. It doesn't need entire input data to be present in sorted array, takes one element at a time.

② Adaptive nature - Handle different input scenarios by making minimal comparisons & movements of elements.

③ No need for Additional storage - space complexity : O(1)

④ Efficient for small Inputs.

→ Online sorting Algorithm

* Bubble sort - ✗
* Selection sort - ✗
* Count sort - ✗
* Quick sort - ✗
* Merge sort - ✗
* Heap sort - ✗
* Radix sort - ✗

[Only Insertion sort is an Online sorting Algorithm]

Ans (3) Complexity table :

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
* 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	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
* Count sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$
* Quick sort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(n)$
* Merge sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
* Heap sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
* Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

Ans- 4)

Algorithm	Inplace sorting	Stable sorting	Online sorting
* Bubble sort	✓	✓	×
* Selection sort	✓	×	×
* Insertion sort	✓	✓	✓
* Count sort	×	✓	×
* Quick sort	✓ (with optimized implem)	×	×
* Merge sort	×	✓	×
* Heap sort	✓ (with optimized implem)	×	×
* Radix sort.	×	×	×

Ans- 5)

Recursive code for Binary search-

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

Iterative code for Binary search-

```
int binary-search_iteractive (vector <int > & arr, int target)
{ int low =0;
  int high = arr.size ()-1;
  while (low <= high)
      { int mid =low+(high -low) /2;
        if (arr[mid] == target)
          { return mid;
          }
        else if (arr[mid ] <target)
        { low= mid+1;
        }
        else
         { high= mid-1;}
   }
```

return -1;
}

→ Time and space complexity -

	Avg case	Worst case	space complexity
* linear Search:	O(n)	O(n)	O(1)
* Binary Search: (Recursive)	O(logn)	O(logn)	due to Recursive calls O(logn)
* Binary search: (Iterative)	O(logn)	O(logn)	O(1)

Ans - (6) let T(n) → time complexity of Binary Search,
n → size of input array.

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Problem size is reduced by half in each recursive call (i.e, searching in the left or right half of the array) and the constant time operation (O(1)) corresponds to comparison and index calculations performed at each step of Binary Search.
→ leading to a logarithmic time complexity of O(logn)

$$\therefore \boxed{T(n) = T\left(\frac{n}{2}\right) + O(1)}$$

Ans (7)

```
pair<int,int> findIndexesWithSum(vector<int>& A, int K)
{ Unordered_map<int, int> valueToIndexMap;
 for (int i=0; i< A.size(); ++i)
  { int requiredValue = K - A[i];
   if (valueToIndexMap.find(requiredValue) != valueToIndexMap
                                              .end())
    {
     return {valueToIndexMap[requiredValue], i};
    }
   else
    {
     valueToIndexMap[A[i]] = i;
    }
  }
 return {-1,-1};
}
```

Time complexity → O(n)
Space complexity → O(n)

Ans – (8) The choice of sorting algorithm depends on the specific requirements and characteristics of the dataset. Quick sort and Merge sort are generally preferred for large datasets due to their efficiency, while Insertion Sort may be more suitable for small datasets or nearly sorted data.
Additionally, the stability and memory usage of the sorting algorithm are also important considerations in practical applications.

* Quick sort →

① It is often considered the best general purpose sorting algorithm for large datasets.
② Avg-case time complexity : O(n log n), making it very efficient.
③ It is an in place sorting algorithm, requiring only O(log n) stack space for recursion, memory efficient.
④ Efficient and ease at implementation.

Ans → (9)

* No. of inversions refers to the count of pairs of elements that are out of their desired order.
* In an Array A, if there are 2 indices i and j such that i<j but A[i]>A[j], then pair (i, j) is considered an inversion.

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

→ Merge sort -

```
long long merge(vector<int> &arr, int low, int mid, int high)
{ long long inversionCount=0;
  int l_size = mid-low+1;
  int r_size = high-mid;
  Vector <int> leftArray (l_size);
  Vector <int> rightArray (r_size);
  for(int i=0; i<l_size; ++i)
    { leftArray[i] = arr[low+i];
    }
  for (int j=0; j< r_size; ++j)
    { rightArray[j] = arr[mid+1+j];
    }
```

```
int i=0, j=0, k=low;
while (i< l_size && j<r_size)
   { if (leftArray [i] <= rightArray [j])
       { arr[k++]=leftArray [i++];
       }
     else
       { arr [k++]= rightArray [j++];
         inversionCount += l_size - i;
       }
   }
while (i<l_size)
 { arr[k++] = leftArray [i++]
 }
while (j <r_size)
 { arr[k++] = rightArray [j++];
 }
return inversionCount;
}

long long mergeSort (vector<int> &arr, int low, int high)
  { long long inversionCount=0;
    if( low < high)
     { int mid = low + (high-low) /2;
       inversionCount += mergesort(arr, low, mid);
       inversionCount += megesort (arr, mid+1, high);
       inversionCount += merge. (arr, low, mid, high);
     }
  return inversionCount;
}
```

O/P → No of inversions : 22

Ans - (10)

* Best Case Time Complexity - $O(n \log n)$.

(1) when pivot element is chosen such that it consistently partitions the array into nearly equal halves.

(2) pivot element is the median of the array or when array contains many duplicates.

* Worst Case Time Complexity - $O(n^2)$

(1) when pivot element is chosen poorly, leading to highly unbalanced partitions.

(2) When pivot element is the smallest or largest element in the array, or when array is already sorted or nearly sorted.

Ans - (11)

* Merge sort :

(1) Best case → $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

* It consistently divides array into two equal halves at each recursive step.
* $O(n)$ represents time taken for merging two sorted halves of the array.

(2) Worst case → same in both worst and best cases.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

* Quick sort :

(1) Best case → $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

* consistently partitions the array into two equal halves at each recursive step.
* $O(n)$ represents the time taken for partitioning the array.

② Worst case: $T(n) = T(n-1) + O(n)$

* When pivot selection is not optimal, leading to highly unbalanced partitions.
* At each recursive step, only one element is removed from the array, leading to quadratic time complexity.

✡ Similarities b/w Merge and Quick sort complexities →

① Divide and Conquer Approach.
② Recurrence Relations. (in best case)

✡ Differences b/w Merge and Quick sort complexities →

① Worst case Time complexity:

Quick sort: Best: $O(n \log n)$
Worst: $O(n^2)$

Merge sort
Best: $O(n \log n)$
Worst: $O(n \log n)$

② Pivot selection - In Quick sort, selection of pivot element effects a lot in performance.
In Merge, not affected by choice of pivot.

③ Stability - Merge sort → A stable Sorting Algorithm.
Quick sort → Not A stable sorting Algorithm.

✡ Reasons for Similarities and Differences →

① Because they both employ the same divide and conquer strategy, dividing the problem into smaller subproblems and then combining the solutions. (Similar)

② Differences Because of Pivot selection process of Quick sort. Poor pivot selection lead to highly unbalanced partitions, resulting in worst case complexity of $O(n^2)$. While, Merge sort divides the array into equal halves

regardless of the input, leading to consistent O(n log n) time complexity.

Ans → (12)

```
void stableSelectionSort(vector<int>& arr)
{int n=arr.size();
for(int i=0 ; i<n-1 ;++i)
{ int minIndex=i;
  for(int j=i+1 ;j<n ;++j)
   {
     if(arr[j]<arr[minIndex])
     { minIndex=j;
     }
   }
int minValue=ar[minIndex];
while(minIndex>i)
   { arr[minIndex]=arr[minIndex-1];
     minIndex--;
   }
  arr[i]=minValue;
}
}
```

Ans-(13)

```
Void. optimized BubbleSort(vector<int>&arr)
{ int n=arr.size();
  bool swapped;
  for(int i=0; i<n-1 ;++i)
   { swapped= false;
     for(int j=0; j<n-i-1; +tj)
      { if(arr[j]> arr[j+1])
         { int temp = arr[j];
           arr[j] = arr[j+1];
           arr[j+1] = temp;
           swapped = true;
} }
```

```
    if (!swapped)
      {break;
      }
  }
}
```

Ans (14) We need to resort to External Sorting Algorithms.

(In this scenario, with a 4GB array and only 2GB of physical memory Available, we must use external sorting techniques.)

Choice of Algorithm -

(1) ~~External~~ Merge Sort : * optimized version of Merge sort designed explicitly for external sorting scenarios.
* efficiently handles large datasets that cannot fit entirely into memory.
* It divides the dataset into smaller chunks that fit into memory, sorts these chunks internally, and then merges them together using disk-based operations.

Internal sorting	External Sorting
* Refers to sorting algorithms that can handle datasets entirely within the available main memory (RAM).	* Deals with datasets that are too large to fit entirely into memory and must be stored on external storage devices like hard disks.
* Eg- Quick, Merge, Insertion	* Eg- External merge sort, Polyphase sort.
* Entire dataset fits into RAM, allowing fast random access to elements during sorting operations	* efficiently manage the sorting of large datasets by minimizing the no of disk I/O operations.
* Used when dataset is small enough to fit into memory without causing memory overflow issues.	* Requires careful consideration of disk I/O operations & minimizing the movement of data b/w main memory & external storage.