DAA Assignment-2

Naman Deol
4th Sem CE
Rollno: 27(2019571)

Ans-1 => Pseudo code for ~~Insertion sort~~ linear search

```
#include <iostream>
using namespace std;
int search( int array[], int n, int key)
{
    int i=0;
    while( i<n && array[i] <= key) // condition for minimum comparisons
    {   if(array[i]==key)
        return i;
    i++;
    }
    return -1;
}
```

Ans-2 => i) Iterative function

```
void InsertionSort( int arr[], int n)
{   for(int i=0; i<n; i++)
    {   int key = arr[i];
        int j = i-1;
        while (j>=0 && arr[j]>key){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

ii) Recursive method

```
void recursiveinsertion ( int arr[], int n)
{
    If (n<=1)
       return;
    recursiveinsertion ( arr, n-1);
    int last = arr[n-1];
    int j = n-2;
    while (j>=0 && arr[j]> last) {
            arr[j+1]= arr [j];
            j--;
    }
    arr[j+1]= last;
}
```

Insertion sort is called an online sorting algorithm because it can sort a list as it receives it, without waiting for the entire list to be received. This is because insertion sort builds the sorted list one element at a time, by inserting each new element in its proper place in the already - sorted part of the list.

Other sorting algorithms that have been discussed in lectures are such as bubble sort, selection sort, merge sort, quick sort, and heap sort, are not online sorting algorithms.

Ans-3 =>

Time Complexity → / Sorting techniques ↓	Best case	Worst case	Average case	Space complexity
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Ans-4 => In-place sorting algorithms: They sort the input array in-place without requiring any additional memory beyond the original array.

- Bubble Sort
- Selection Sort
- Insertion Sort

Stable sorting algorithms: Algorithms that maintain the relative order of equal elements in the sorted array.

- Bubble sort
- Insertion sort
- Merge sort

Online sorting algorithms: Algorithms that can sort a list as it receives it, without waiting for the entire list to be received.

- Insertion sort.

Binary Search

Ans-5 => i) Recursive method

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

ii) Iterative method

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

	Best Case	Avg Case	Worst case	Space Complexity
Binary Search (Recursive)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$
Binary Search (Iterative)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Linear Search (Recursive)	$O(1)$	$O(n/2)$	$O(n)$	$O(n)$
Linear Search (Iterative)	$O(1)$	$O(n/2)$	$O(n)$	$O(1)$

Ans-6 =) $T(n) = T(n/2) + O(1)$

$T(n)$ => Time taken to search for an element in an array of size n.

$O(1)$ => Constant time taken to perform each comparison / check at each recursive step.

Ans-7 =>

```
vector<int> find_k( vector<int> vec, int k){
    sort(vec.begin(), vec.end());
    vector<int> ret;
    int i = 0, j = vec.size() - 1;
    while ( i<j)
    {
        if (vec[i]+vec[j] == k)
            return {i, j};
        else if (vec[i]+vec[j] < k)
            i++;
        else
            j--;
    }
    return { };
}
```

Ans-8 => In general, for small arrays (e.g. less than 20 elements), simple sorting algorithms such as insertion sort or selection sort can perform well due to their simplicity and low overhead. However, for larger arrays, more efficient algorithms such as quicksort, mergesort, or heapsort may be preferred due to their faster average case performance.

In terms of stability, insertion sort, merge sort and radix sort are stable sorting techniques which means that they preserve the relative order of equal elements in the input array. On the other hand, quicksort and heapsort are not stable sorting techniques.

If memory usage is concern, then an in-place sorting algorithm such as heap or quick sort may be preferred, as they do not require additional memory to store intermidate results.

Ans-9 => Inversion in an array is defined as a pair of elements in the array such that the elements are not in their sorted order. More formally, if $i < j$ and $arr[i] > arr[j]$, then (i, j) is said to be an inversion in the array.

Ans-10 => Best case :- Quick sort will have a time complexity of $O(n \log n)$ when the partition process always divides the array into two equal halves.

Worst case : Quick sort will have a time complexity of $O(n^2)$ when the partition process always divides the array into one subarray of size 0 and another subarray of size $(n-1)$. This happens when the pivot selected is either the smallest or largest element in the array, and the array is already sorted in ascending or descending order.

Ans-11 => Recurrence relation for merge sort:

Best case : $T(n) = 2T(n/2) + O(n)$

Worst case : $T(n) = 2T(n/2) + O(n)$

Recurrence relation for quick sort

Best case : $T(n) = 2T(n/2) + O(n)$

Worst case : $T(n) = T(n-1) + O(n)$

The similarities between the complexity of Merge sort and Quick sort are that their best case time complexity is $O(n \log n)$ and they both use divide-and-conquer approach.

The difference in worst case time complexity between Merge sort and Quick sort is due to the partioning method used in Quick sort, which can lead to an unbalanced partition and hence more comparisons in the worst case. Merge sort, on the other hand, always divides the array into two equal halves, ensuring that the worst-case time complexity is still $O(n \log n)$.

Ans-12 Yes, it is possible to write a stable version of Selection sort.

Pseudo code for stable version of selection sort:

```
void stableselection ( int arr[], int n)
{   for( int i=0; i<n-1 ; i++) {
      int min = i;
      for ( int j = i+1; j<n; j++) {
            if( arr[j] < arr[min])
                  min = j;
      }

      int key = arr[min];
      while ( min > i) {
            arr[min] = arr[min-1];
            min = min-1;
      }

      arr[i] = key;
   }
}
```

Ans-13 ⇒ Yes, we can modify the bubble sort to stop scanning the array once it is sorted.

```
void bubblesort( int arr[], int n) {
      bool swap;
      for ( int i=0; i<n-1 ; i++) {
            swap = false;
            for( int j=0; j<n-i-1 ; j++) {
                  if(arr[j] > arr[j+1] ) {
                        swap (arr[j], arr[j+1]);
                        swap = true;
                  }   }
            if(!swap)
                  break;
      }
}
```

Ans-14 => Since the array to be sorted is larger than the available RAM, we need to use external sorting. External sorting is a technique used to sort large datasets that cannot fit entirely in memory (RAM). It involves dividing the dataset into smaller chunks that can fit into memory, sorting them in memory, and then merging them back into a single sorted dataset.

One popular algorithm for external sorting is the external merge sort. It works by dividing the dataset into smaller chunks that can fit into memory, sorting them in a memory using an ~~to~~ internal sorting algorithm such as quicksort or heapsort, and then merging the sorted chunks back into a single sorted dataset using a merge operation. This process is repeated until the entire dataset is sorted.

In the given scenario, since the available RAM is only 2GB, we can divide the 4GB dataset into two 2GB chunks, sort each chunk using an ~~external merge sort. This approach~~ internal sorting, and then merge the sorted chunks using external merge sort. This approach enable us to sort the entire dataset while utilizing the available memory efficiently.