Name : Rahul Joshi

Sec : ML

Roll No : 44

Subject : DAA (TCS-401)

Assignment -2

A-1 Linear search (arr, target, length arr)

```
{
    ~~n = length~~

    for (i = 0 ; i < n ; i++)
    {
        if (arr[i] == target)
            return i;
        else if (arr[i] > target)
            break
    }
    return -1;
}
```

A 2 Iterative insertion sort

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

```
    arr [j+1] = key ;
  }
}
```

Recursion Insertion sort:

```
void insertionr (int arr[], int n)
{
    if (n <= 1)
        return ;

    insertionr(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;
}
```

An online sorting algorithm works by processing elements one at a time, while keeping the sequence sorted as more elements are added. Insertion sort is an online algorithm because it works from left to right & doesn't need to see the entire array.

Insertion sort is a basic sorting algorithm that builds a final sorted array one element at a time. The array is split into a sorted & unsorted part. Value from the unsorted part are picked and placed at the correct posion sort in the sorted part.

<u>Other online sorting</u>

1. Insertion sort - ✓
2. Merge sort - ✗
3. Heap sort - ✗
4. Quick sort - ✗
5. Radix sort - ✗
6. Count sort - ✗
7. Bubble sort - ✗
8. Selection sort - ✗

3. Sorting technique / Time complexity / Space complexity

Sorting technique	Best	Avg	Worst	Space complexity
1) Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
2) Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
3) Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
4) Count sort	$O(n)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
5) Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$
6) Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
7) Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$
8) Radix sort	$O(d(n+k))$	$O(d(n+k))$	$O(d(n+k))$	$O(n+k)$

4.

Inplace	Stable	Online
Bubble Sort	Bubble sort	—
Selection sort	—	—
Insertion sort	Insertion sort	Insertion sort
Quick sort	—	—
Heap sort	—	—
—	Merge sort	—
—	Count sort	—
—	Radix sort	—

5. Recursive Binary Search

```
int bs (int arr[], int l, int r, int key)
{
    if (r >= l)
    {
        int mid = l + (r-1)/2
        if (arr[mid] == key)
            return mid;

        if (arr[mid] > l)
            return bs (arr, l, mid-1, key)
```

```
return bs(arr, mid+1, r, key);

}
return -1;
}
```

Iterative Binary Search

```
int bs (int arr[], int l, int r, int key)
{
    while (l<=r)
    {
        int m = l+(r-l)/2;

        if(arr[m] == key)
            return m;

        if (arr[m] < key)
            l = m+1;

        else
            r = m-1;
    }
    return -1;
}
```

Search	Time complexity Best	avg	worst	Space complexity
linear	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary (iterative)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Binary (recursive)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log_2 n)$

6. Recurrence relation for binary ~~___~~ recursive search :

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

where : $T(n)$ = time taken to search within array of size n

$T(n/2)$ = time taken to search within half of array

$O(1)$ = Const time take for comparison & other operation.

7. pseudocode

```
Search (A[], n, key)
{
    ms (A, n) // Merge sort call
    i=0;
    j = n-1;
    while (i<j)
    {
        if (a[i] + A[j]) = = key)
        {
        } return i, j;
```

```
else if((A[i] + A[j]) < key)
        i++;
else
        j++;
}
return -1, -1;
}
```

8- Quicksort - It is fast & efficient:
advantages: Fast average case performance, good cache performance and it is an in place sorting algorithm. It is efficient for sorting large datasets as it has an average time complexity of O(n log n). Its divide & conquer strategy allows it to quickly sort the data by recursively dividing the array into smaller partitions & sorting them individually

9. The inversion count for an array is the number of steps it will take for the array to be sorted, or how far away any array is from being sorted

```
#include&lt;iostream&gt;
using namespace std;

int mergesort (int arr[], int temp[],
               , int left, int right);
int merge (int arr[], int temp[], int left
           , ~~int right~~, int mid, int right);
```

```
int mergesort (int arr[], int array_size)
 {
   int temp [array_size];
   return mergesort (arr, temp, 0, array_size
                                        - 1);
 }

int mergesort (int arr[], int temp [], int left,
        int right)
   {
      int mid, inv_count = 0;
      if (right > left)
       {
         mid = (right + left) / 2
       inv_count += mergesort (arr, temp
                               , left
                               , mid)
       inv_count += mergesort (arr, temp
                               , mid + 1, right)
       inv_count += merge (arr, temp, left
                          , mid + 1, right);
      }
    return inv_count;
 }

int merge (int arr [], int temp [], int left,
          int mid, int right)
   {
     int i, j, k;
     int inv_count = 0;
```

```
i = left;
j = mid;
k = left;
while ((i <= mid-1) && (j <= right))
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];

    else
        {
            temp[k++] = arr[j++]
            inv_count = inv_count + (mid-i);
        }
}
while (i <= mid-1)
        temp[k++] = arr[i++];

while (j <= right)
        temp[k++] = arr[j++];

for (i = left; i <= right; i++)
        arr[i] = temp[i];

return inv_count;
}
int main()
  {
     int arr[] = {7,21,3,1,8,10,1,20,6,4,5};
     int n = sizeof(arr)/sizeof(arr[0]);
     int ans = mergeSort(arr, n)
     Cout << "inversions =" << ans;
     return 0;
```

Output

inversions = 31

10. Quick sort's best case time complexity is $O(n \log n)$. This occurs when the pivot is chosen as the median element in array on each ~~recursive~~ recursive call. This results in roughly equal-sized partitions on each recursive call.

Worst

Quick sort worst case time complexity is $O(n^2)$. This occurs when ~~the~~ the partition sizes are unbalanced. For example, if the pivot ~~chosen~~ chosen by the partition function is always either the smallest or the largest element in the subarray. This results in one partition having all the elements & the other partition having none of the elements. Worst ~~case~~ case can also occur when the array is sorted and the smallest or largest ~~~~ indexed element is selected as the pivot.

11. Merge sort

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

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

~~Merge Sort~~ :

Quick sort:

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

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

Similarity:

1. Divide & conquer approach

Both merge & quick sort follow the divide and conquer paradigm. The recursively break down a problem into smaller subproblems & combine the solutions to the subproblems to solve the original problem.

In best case both have same recurrence relation

$\therefore T(n) = 2T(n/2) + O(n)$

Best case T.C = $O(n \log n)$

Difference

1. They have difference in worst time complexity. Merge sort has worst T.C $O(n \log n)$ & Quick sort has $O(n^2)$

Quick sort relies heavily on the choice of pivot element. In worst case poor pivot selection can lead to Quadratic time complexity $O(n^2)$. Merge sort on the other hand divides the array into two halves, regardless of input distribution.

12.
```
void stable(int a[], int n)
{   int i;
 for(i=0; i<n-1; i++)
   {
    ~~int min=i~~
    int min = i;
   for (int j = i+1; j<n; j++)
       if (a [min] > a[j])
           min = j;

    int key = a [min], k;
    for (k = min; k>i; k--)
        a[k] = a[k-1] // Shifting elements

    a[i] = a[key]; // Storing key at right
                      position

}
```

13.

Date / /
Page

13. void bubble (int arr[], int size)

```
{
    for (int i = 0 ; i < size - 1; i++)
    {
        int flag = 0
        for (j = 0; j < size - i - 1; j++)
        {
            if (arr[j] > arr[j + 1];
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                flag = 1;
            }
        }
        if (flag == 0)
            break;
    }
}
```

13. We will ~~use~~ use k-way merge sorting algorithm for this purpose

1. Divide the array:
Divide the 4GB array into k chunks where each chunk can fit into the available 2GB RAM. Those ~~chunks~~ chunks can be loaded into memory one at a time ~~for~~ for sorting.

2. Sort Each chunk -
~~App~~ Apply an in-memory sorting algorithm to sort each individual chunk.

3. <u>k-way merging</u> - use the ~~the~~ k-way merge sort algorithm to merge the sorted chunks. This involves merging k sorted list at a time until you obtain a single sorted list.

<u>Internal sorting</u>:

Internal sorting refers to the process of sorting data that fits entirely within the computer's main memory (RAM). In this scenario, the entire dataset can be loaded into memory and sorting algorithms can operate efficiently without the need for external storage. eg quicksort, merge sort, heapsort

<u>External sorting</u>

External sorting deals with sorting datasets that are too large to fit entirely into the computer's main memory.
when the dataset exceeds the available RAM, external storage (hard drives or SSDs) is used to store parts of dataset temporarily.
External sorting algorithms are designed to minimize the no of disk I/O operations, as reading & writing to external storage is ~~significantly~~ significantly slower than operations within RAM.

eg: External Merge sort.