Sorting Algorithms

Session: 2021 – 2025

Submitted by:

Fatiq Hussnain 2021-CS-140

Supervised by:

Ms.Maida Shahid

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Table of Contents

DESCRIPTION OF PROJECT

This is a CLI and GUI based project in which we test many sorting algorithms by printing record of different employees on their index base. In CLI and GUI we use 9 algorithms as follows.

The 9 sorting Algorithms .

a. Bubble Sort

b. Selection Sort

c. Insertion Sort

d. Merge Sort

e. Quick Sort

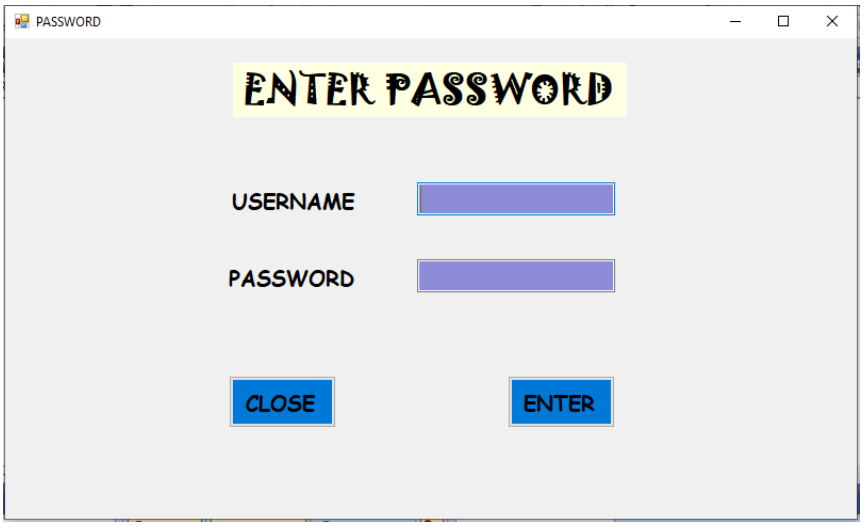
f. Heap Sort

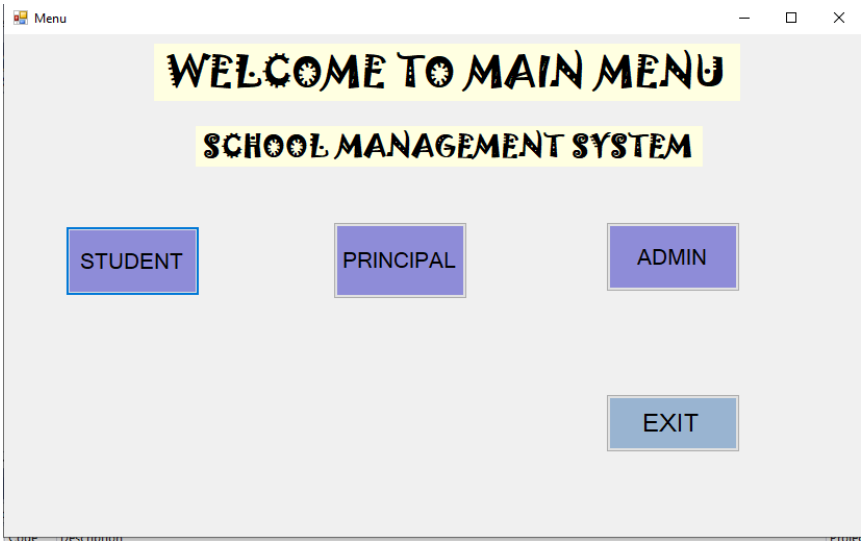
g. Counting Sort

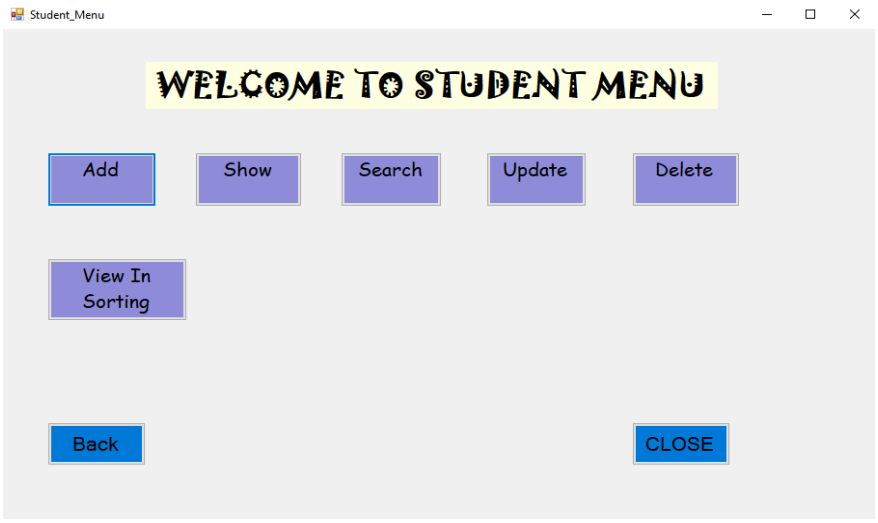
h. Radix Sort

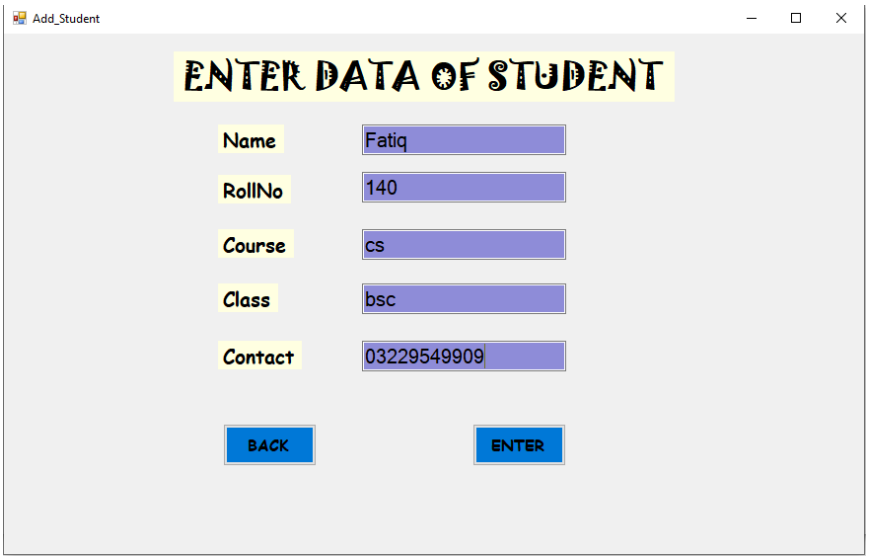
i. Bucket Sort

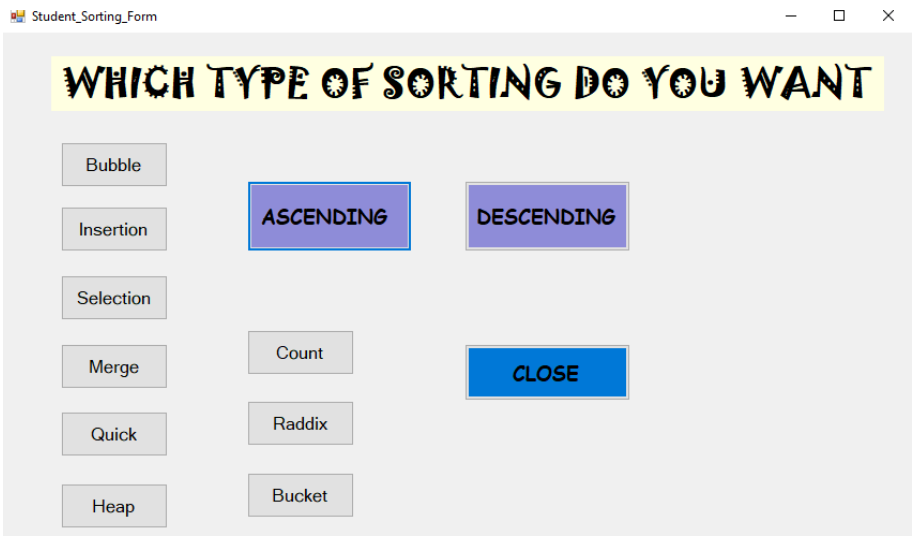
WIRE FRAMES OF GUI



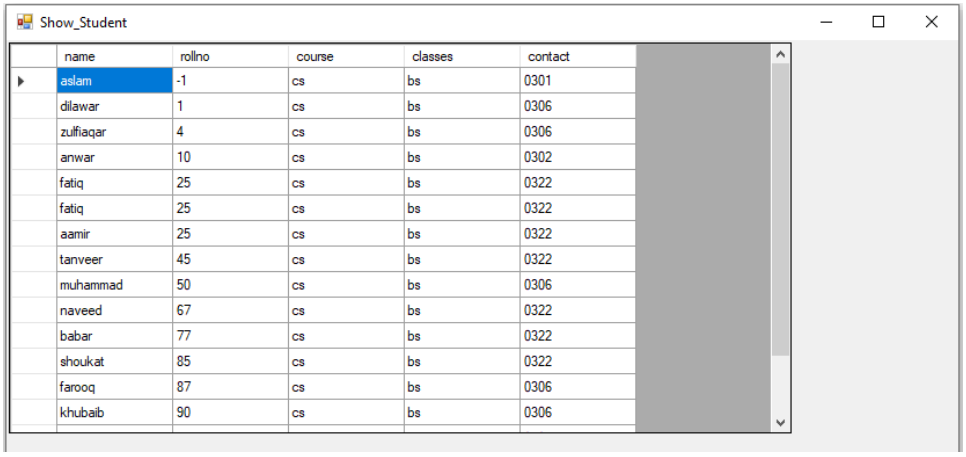



Student_Menu
WELCOME TO STUDENT MENU
Add
Show
Search
Update
Delete
View In Sorting
Back
CLOSE


Add_Student
ENTER DATA OF STUDENT
Name
Fatiq
RollNo
140
Course
cs
Class
bsc
Contact
03229549909
BACK
ENTER

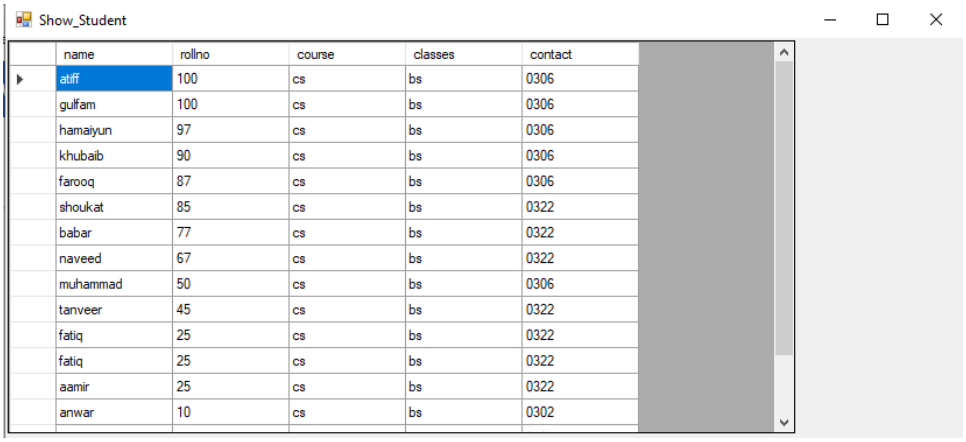

Student_Sorting_Form
WHICH TYPE OF SORTING DO YOU WANT
Bubble
Insertion
Selection
Merge
Quick
Heap
ASCENDING
DESCENDING
Count
Raddix
Bucket
CLOSE

ASCENDING SORTING...



name	rollno	course	classes	contact
aslam	-1	cs	bs	0301
dilawar	1	cs	bs	0306
zulfiaqar	4	cs	bs	0306
anwar	10	cs	bs	0302
fatiq	25	cs	bs	0322
fatiq	25	cs	bs	0322
aamir	25	cs	bs	0322
tanveer	45	cs	bs	0322
muhammad	50	cs	bs	0306
naveed	67	cs	bs	0322
babar	77	cs	bs	0322
shoukat	85	cs	bs	0322
farooq	87	cs	bs	0306
khubaib	90	cs	bs	0306

DESCENDING SORTING



name	rollno	course	classes	contact
atiff	100	cs	bs	0306
gulfam	100	cs	bs	0306
hamaiyun	97	cs	bs	0306
khubaib	90	cs	bs	0306
farooq	87	cs	bs	0306
shoukat	85	cs	bs	0322
babar	77	cs	bs	0322
naveed	67	cs	bs	0322
muhammad	50	cs	bs	0306
tanveer	45	cs	bs	0322
fatiq	25	cs	bs	0322
fatiq	25	cs	bs	0322
aamir	25	cs	bs	0322
anwar	10	cs	bs	0302

EXECUTION TIME FOR DIFFERENT SORTING ALGORITHMS.

1.BUBBLE SORT

100	1000	10000	100000	500000
0	0.0499	4.186453	540.751352	

DISCUSSION:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed.

TIME COMPLECITY

Worst complexity : n^2

Average complexity : n^2

Best complexity : n

Space complexity : 1

2.INSERTION SORT

100	1000	10000	100000	500000
0	0.002993	0.182696	63.981107	

DISCUSSION:

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time by comparisons. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

TIME COMPLECITY

Worst complexity : n^2

Average complexity : n^2

Best complexity : n

Space complexity : 1

3.SELECTION SORT

100	1000	10000	100000	500000
0	0.006981	0.381639	118.245373	

DISCUSSION:

In computer science, selection sort is an in-place comparison sorting algorithm. It has an $O(n^2)$ time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort.

TIME COMPLECITY

Worst complexity : n^2

Average complexity : n^2

Best complexity : n^2

Space complexity : 1

4.MERGE SORT

100	1000	10000	100000	500000
0	0.002031	0.088847	1.065888	6.460477

DISCUSSION:

In computer science, merge sort is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.

TIME COMPLECITY

Worst complexity : n*log(n)

Average complexity : n*log(n)

Best complexity : n*log(n)

Space complexity : n

5.QUICK SORT

100	1000	10000	100000	500000
0	0.000962	0.032959	0.108757	0.563864

DISCUSSION:

Quicksort is an in-place sorting algorithm. Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.

TIME COMPLECITY

Worst complexity : n^2

Average complexity : n*log(n)

Best complexity : n*log(n)

Space complexity : O(nlog*n).

6.HEAP SORT

100	1000	10000	100000	500000
0	0.002031	0.024177	0.337062	2.011909

DISCUSSION:

In computer science, heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted.

TIME COMPLECITY

Worst complexity : n*log(n)

Average complexity : n*log(n)

Best complexity : n*log(n)

Space complexity : O(1).

7.COUNT SORT

100	1000	10000	100000	500000
0	0.002409	0.013661	0.135549	0.715564

DISCUSSION:

In computer science, counting sort is an algorithm for sorting a collection of objects according to keys that are small positive integers; that is, it is an integer sorting algorithm.

TIME COMPLECITY

Worst complexity : (n + r)

Average complexity : (n + r)

Best complexity : (n + r)

Space complexity : (n + r).

8.RADDIX SORT

100	1000	10000	100000	500000
0	0.002991	0.032371	0.43651	2.289381

DISCUSSION:

In computer science, radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.

TIME COMPLECITY

Worst complexity : (n*k/d).

Average complexity : (n*k/d).

Best complexity : (n + r)

Space complexity : n+2^d.

9.BUCKET SORT

100	1000	10000	100000	500000
0	0.001000	0.000996	0.006021	0.040343

DISCUSSION:

Bucket sort's best case occurs when the data being sorted can be distributed between the buckets perfectly. If the values are sparsely allocated over the possible value range, a larger bucket size is better since the buckets will likely be more evenly distributed.

TIME COMPLECITY

Worst complexity : O(n^2).

Average complexity : O(n+k)

Best complexity : O(n+k)

Space complexity : O(n+k).

FINAL RESULTS.

WE concluded that for a big data like 500k we should use Heap Sort, Quick Sort and Merge Sort because their sorting time is good then others. And for small data we should use Insertion Sort then Bubble and Selection Sort because their time complexity is good for small data in the range like 1k to 10k.

Besides this Bucket Sort is also good for large data. But its time complexity is not good. So both Heap and Quick Sorts are best for large data.

FULL CODE OF CLI PROJECT.

```
#include <iostream>
#include <queue>
#include <vector>
#include <stdio.h>
#include <conio.h>
#include <chrono>
#include <sstream>
#include <fstream>
#include <algorithm>
// #include <bits/stdc++.h>
using namespace std;
using namespace ::chrono;


void home();
void readRecord();
void bigData();
struct Record
```

```
{
    int index;
    string organization_Id;
    string name;
    string website;
    string country;
    string description;
    int founded;
    string industry;
    int no_of_employees;
};
vector<Record> v;

void insert(Record company)
{
    v.push_back(company);
}
void printEmployeeIndexes()
{
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i].no_of_employees << "\t";
    }
}
void swap(int &a, int &b)
{
    int temp;
    temp = a;
```

```
    a = b;
    b = temp;
}
//////////////////Bubble Sort///////////////
void bubbleSort() // class ki waja sa ham na function ko parameters pass ni kiya.
{
    for (int i = 0; i < v.size() - 1; i++)
    {
        bool isSwapped = false;
        for (int j = 0; j < v.size() - 1 - i; j++)
        {
            if (v[j].no_of_employees > v[j + 1].no_of_employees)
            {
                swap(v[j], v[j + 1]);
                isSwapped = true;
            }
        }
        // In case if the vector list is already sorted
        if (isSwapped == false)
        {
            break;
        }
    }
}
//////////////////Insertion Sort///////////////
void insertionSort()
{
    Record temp;
```

```
    for (int x = 1; x < v.size(); x++)
    {
        temp = v[x];
        int y = x - 1;
        while (y >= 0 && temp.no_of_employees < v[y - 1].no_of_employees) //
asending order.
        {
            v[y].no_of_employees = v[y - 1].no_of_employees;
            y--;
        }
        v[y].no_of_employees = temp.no_of_employees;
    }
}
//////////////////Selection Sort//////////////////////////
void selectionSort()
{
    int minIdx;
    for (int i = 0; i < v.size() - 1; i++)
    {
        Record min = v[i];
        minIdx = i;
        for (int j = i; j < v.size(); j++)
        {
            if (v[minIdx].no_of_employees > v[j].no_of_employees)
            {
                minIdx = j;
                min = v[j];
            }
        }
```

```
            swap(v[i], v[minIdx]);
        }
    }
//////////////////Merge Sort//////////////////////////
void merge(vector<Record> &DataBase, int start, int mid, int end)
{
    int i = start;
    int j = mid + 1;
    queue<Record> tempData;
    while (i <= mid && j <= end)
    {
        if (DataBase[i].no_of_employees < DataBase[j].no_of_employees)
        {
            tempData.push(DataBase[i]);
            i++;
        }
        else
        {
            tempData.push(DataBase[j]);
            j++;
        }
    }
    while (i <= mid)
    {
        tempData.push(DataBase[i]);
        i++;
    }
    while (j <= end)
```

```
    {
        tempData.push(DataBase[j]);
        j++;
    }
    for (int x = start; x <= end; x++)
    {
        DataBase[x] = tempData.front();
        tempData.pop();
    }
}
void mergeSort(vector<Record> &DataBase, int start, int end)
{
    if (start < end)
    {
        int mid = (start + end) / 2;
        mergeSort(DataBase, start, mid);
        mergeSort(DataBase, mid + 1, end);
        merge(DataBase, start, mid, end);
    }
}
//////////////////Quick Sort///////////////
int partition(vector<Record> &arr, int start, int end)
{
    int pivot = arr[start].no_of_employees;
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i].no_of_employees <= pivot)
            count++;
```

```
    }
    // Giving pivot element its correct position
    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);
    // Sorting left and right parts of the pivot element
    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {
        while (arr[i].no_of_employees <= pivot) {
            i++;
        }
        while (arr[j].no_of_employees > pivot) {
            j--;
        }
        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}
void quickSort(vector<Record>&arr, int start, int end)
{
    // base case
    if (start >= end)
        return;
    // partitioning the array
    int p = partition(arr, start, end);
    // Sorting the left part
    quickSort(arr, start, p - 1);
```

```
    // Sorting the right part
    quickSort(arr, p + 1, end);
}
//////////////////Heap Sort////////////////////////////////
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(vector<Record> &v, int n, int i)
{
    int largest = i;   // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2
    // If left child is larger than root
    if (l < n && v[l].no_of_employees > v[largest].no_of_employees)
        largest = l;
    // If right child is larger than largest so far
    if (r < n && v[r].no_of_employees > v[largest].no_of_employees)
        largest = r;
    // If largest is not root
    if (largest != i)
    {
        swap(v[i], v[largest]);
        // Recursively heapify the affected sub-tree
        heapify(v, n, largest);
    }
}
// main function to do heap sort
void heapSort(vector<Record> &v, int n)
{
```

```
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(v, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--)
    {
        // Move current root to end
        swap(v[0], v[i]);
        // call max heapify on the reduced heap
        heapify(v, i, 0);
    }
}
///////////////Count Sort//////////////////////////
void countingSort(vector<Record> &arr)
{
    // Record max = *max_element(arr.begin(), arr.end());
    int max = arr[0].no_of_employees;
    for (int i = 0; i < arr.size(); i++)
    {
        if (arr[i].no_of_employees > max)
        {
            max = arr[i].no_of_employees;
        }
    }
    vector<Record> count(max + 1);
    vector<Record> output(arr.size());
    for (int x = 0; x < arr.size(); x++)
    {
```

```
        count[arr[x].no_of_employees].no_of_employees++;
    }
    for (int x = 1; x < count.size(); x++)
    {
        count[x].no_of_employees = count[x - 1].no_of_employees +
count[x].no_of_employees;
    }
    for (int x = arr.size() - 1; x >= 0; x--)
    {
        int index = count[arr[x].no_of_employees].no_of_employees - 1;
        count[arr[x].no_of_employees].no_of_employees--;
        output[index] = arr[x];
    }
    for (int x = 0; x < output.size(); x++)
    {
        arr[x] = output[x];
    }
}
///////////////Raddix Sort/////////////////////////////////
void Raddix(vector<Record> &arr, int place)
{
    vector<int> count(100);
    vector<int> output(arr.size());
    for (int x = 0; x < arr.size(); x++)
    {
        count[(arr[x].no_of_employees / place) % 10]++;
    }
    for (int x = 1; x < count.size(); x++)
    {
```

```
        count[x] = count[x - 1] + count[x];
    }
    for (int x = arr.size() - 1; x >= 0; x--)
    {
        int index = count[(arr[x].no_of_employees / place) % 10] - 1;
        count[(arr[x].no_of_employees / place) % 10]--;
        output[index] = arr[x].no_of_employees;
    }
    for (int x = 0; x < output.size(); x++)
    {
        arr[x].no_of_employees = output[x];
    }
}
void countingSort(vector<Record> &arr, int place)
{
    vector<Record> count(10);
    vector<Record> output(arr.size());
    for (int x = 0; x < arr.size(); x++)
    {
        count[(arr[x].no_of_employees / place) % 10].no_of_employees++;
    }
    for (int x = 1; x < count.size(); x++)
    {
        count[x].no_of_employees = count[x - 1].no_of_employees +
count[x].no_of_employees;
    }
    for (int x = arr.size() - 1; x >= 0; x--)
    {
```

```
        int index = count[(arr[x].no_of_employees / place) % 10].no_of_employees
- 1;
        count[(arr[x].no_of_employees / place) % 10].no_of_employees--;
        output[index] = arr[x];
    }
    for (int x = 0; x < output.size(); x++)
    {
        arr[x] = output[x];
    }
}
void radixSort(vector<Record> &arr)
{
    // Record max = *max_element(arr.begin(), arr.end());
    int max = arr[0].no_of_employees;
    for (int i = 0; i < arr.size(); i++)
    {
        if (arr[i].no_of_employees > max)
        {
            max = arr[i].no_of_employees;
        }
    }
    int place = 1;
    while ((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
////////////////Buckket Sort////////////
```

```
int getMax(vector<Record> &a, int n) // function to get maximum element from the
given array
{
    int max = a[0].no_of_employees;
    for (int i = 1; i < n; i++)
        if (a[i].no_of_employees > max)
            max = a[i].no_of_employees;
    return max;
}
void bucketSort(vector<Record> &a , int n) // function to implement bucket sort
{
    int max = getMax(a, n); // max is the maximum element of array
    int bucket[max], i;
    for (int i = 0; i <= max; i++)
    {
        bucket[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        bucket[a[i].no_of_employees]++;
    }
    for (int i = 0, j = 0; i <= max; i++)
    {
        while (bucket[i] > 0)
        {
            a[j++].no_of_employees = i;
            bucket[i]--;
        }
    }
```

```
}
string parseRecord(string line, int index)
{
    string temp = "";
    int count = 0;
    for (int i = 0; i < line.length(); i++)
    {
        if (line[i] == ',')
        {
            count++;
        }
        else if (count == index)
        {
            temp += line[i];
        }
    }
    return temp;
}
bool readRecord(int index)
{
    string record = "";
    fstream file;
    if (index == 100)
    {
        file.open("organizations-100.csv", ios ::in);
    }
    else if (index == 1000)
    {
        file.open("organizations-1000.csv", ios ::in);
```

```
    }
    else if (index == 10000)
    {
        file.open("organizations-10000.csv", ios ::in);
    }
    else if (index == 100000)
    {
        file.open("organizations-100000.csv", ios ::in);
    }
    else if (index == 500000)
    {
        file.open("organizations-500000.csv", ios ::in);
    }
    else
    {
        cout << "wrong input.";
    }
    int counter = 0;
    Record rec;
    while (!file.eof())
    {
        getline(file, record);
        if (counter == 0) // to skip first line of file.
        {
            counter++;
            continue;
        }
        if (counter != 0)
```

```
        {
            int var = 0;
            stringstream(parseRecord(record, 0)) >> var;
            rec.index = var;
            rec.organization_Id = parseRecord(record, 1);
            rec.name = parseRecord(record, 2);
            rec.website = parseRecord(record, 3);
            rec.country = parseRecord(record, 4);
            rec.description = parseRecord(record, 5);
            stringstream(parseRecord(record, 6)) >> var;
            rec.founded = var;
            rec.industry = parseRecord(record, 7);
            stringstream(parseRecord(record, 8)) >> var;
            rec.no_of_employees = var;
        }
        counter++;
        insert(rec);
    }
    cout << index << "  Records added successfully.";
    file.close();
    return true;
}
////////////////////////////////////////////////////////////////////
int main()
{
 while (true)
    {
        // Record rec;
```

```
        system("cls");
    xy:
        bigData();
        char option;
        cout << "\nenter your option: ";
        cin >> option;
        if (option == '1') // 100 records
        {
        a:
            system("cls");
            readRecord(100);
            home();
            // inner options
            char op;
            cout << "\nenter your option...";
            cin >> op;
            if (op == '1')
            {
                system("cls");
                auto start = high_resolution_clock::now();
                bubbleSort();
                auto stop = high_resolution_clock::now();
                printEmployeeIndexes();
                auto duration = duration_cast<microseconds>(stop - start);
                cout << "\n\nTime taken by function: "
                     << duration.count() << " microseconds" << endl;
                getch();
                cout << "Press any key to continue" << endl;
```

```
        char temp;
        cin >> temp;
    }
    else if (op == '2')
    {
        system("cls");
        auto start = high_resolution_clock::now();
        insertionSort();
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
             << duration.count() << " microseconds" << endl;
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '3')
    {
        system("cls");
        auto start = high_resolution_clock::now();
        selectionSort();
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
             << duration.count() << " microseconds" << endl;
```

```
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '4')
    {
        system("cls");
        auto start = high_resolution_clock::now();
        mergeSort(v, 0, v.size() - 1);
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
            getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '5')
    {
        system("cls");
        auto start = high_resolution_clock::now();
        quickSort(v,0, v.size() - 1);
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
```

```
            << duration.count() << " microseconds" << endl;
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '6')
    {
        cout << "Testing heap sort start" << endl;
        system("cls");
        auto start = high_resolution_clock::now();
        heapSort(v, v.size() - 1);
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
            << duration.count() << " microseconds" << endl;
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '7')
    {
        // cout << "Testing heap sort start" << endl;
        system("cls");
        auto start = high_resolution_clock::now();
        countingSort(v);
```

```
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
             << duration.count() << " microseconds" << endl;
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '8')
    {
        // cout << "Testing heap sort start" << endl;
        system("cls");
        auto start = high_resolution_clock::now();
        radixSort(v);
        auto stop = high_resolution_clock::now();
        printEmployeeIndexes();
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "\n\nTime taken by function: "
             << duration.count() << " microseconds" << endl;
        getch();
        cout << "Press any key to continue" << endl;
        char temp;
        cin >> temp;
    }
    else if (op == '9')
    {
```

```
                system("cls");
                auto start = high_resolution_clock::now();
                bucketSort(v,v.size());
                auto stop = high_resolution_clock::now();
                printEmployeeIndexes();
                auto duration = duration_cast<microseconds>(stop - start);
                 cout << "\n\nTime taken by function: "
                     << duration.count() << " microseconds" << endl;
                getch();
                cout << "Press any key to continue" << endl;
                char temp;
                cin >> temp;
            }
            else
            {
                // cout << "wrong input" << endl;
                cout << "press any key to go back...";
                getch();
                goto a;
            }
        }
 void home()
{
    cout << "\n=========================";
    cout << "\n   Sorting Algorithms";
    cout << "\n1.Bubble Sort";
    cout << "\n2.Insertion Sort";
    cout << "\n3.Selection Sort";
```

```
    cout << "\n4.Merge Sort";
    cout << "\n5.Quick Sort";
    cout << "\n6.Heap Sort";
    cout << "\n7.Count Sort";
    cout << "\n8.Raddix Sort";
    cout << "\n9.Bucket Sort";
}
void bigData()
{
    cout << "\n1.Load 100 records" << endl;
    cout << "2.Load 1000 records" << endl;
    cout << "3.Load 10000 records" << endl;
    cout << "4.Load 100000 records" << endl;
    cout << "5.Load 500000 records" << endl;
    cout << "6.Back" << endl;
}
```