Ahmed Shafique, Haiqa Qaiser 2021-EE-313, 2022-EE-255

Data Structures and Algorithms

Lab 10 &11: Sorting

Q1): Write a program that implements the following functions:

Functions:

a. Selection_sort().

```
void selectionSort(int arr[], int lenght)
{
    int min_idx;

    for (int i = 0; i < lenght - 1; i++) {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (int j = i + 1; j < lenght; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element
        // with the first element
        if (min_idx != i)
            swap(arr[min_idx], arr[i]);
    }
```

b. Bubble_sort().

```
 int bubbleSort(int arr[], int length)
{
    int temp;
    for (int j = 0; j < length; j++)
    {
        int swap = 0;
        for (int i = 0; i < length-1; i++)
        {
            if (arr[i] > arr[i+1])
            {
                temp = arr[i];
                arr[i]= arr[i+1];
                arr[i+1]= temp;
                swap = swap + 1;
            }
        }
    }
}
```

c. Insertion_sort().

```
void insertionSort(int arr[], int length)
{
    int i, j, key;
    for (i = 1; i < length; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
```

```
            // that are greater than key,
            // to one position ahead of their
            // current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

d. Merge_sort().

```
  void mergesort(int arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int* left_half = new int[n1];
    int* right_half = new int[n2];

    // Copy data to temp arrays left_half[] and right_half[]
    for (int i = 0; i < n1; i++)
        left_half[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        right_half[j] = arr[mid + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (left_half[i] <= right_half[j])
        {
            arr[k] = left_half[i];
            i++;
        }
        else
        {
            arr[k] = right_half[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of left_half[], if any
    while (i < n1)
    {
        arr[k] = left_half[i];
        i++;
        k++;
    }

    // Copy the remaining elements of right_half[], if any
    while (j < n2)
    {
        arr[k] = right_half[j];
        j++;
        k++;
    }

    delete[] left_half;
    delete[] right_half;
}

// for recursively call merge sort because its recursive sort
```

```
// cutting array recursively
void merge_sort(int arr[], int left, int right)
{
    if (left<right)
    {
        int mid = left+(right-left)/2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        mergesort(arr,left,mid,right);
    }

}
```

e. Quick_sort().

```
int partition(int arr[], int start, int end)
{
    int pivot = arr[start];
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= 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] <= pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}

void quickSort(int 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);
}
```

f. Heap_sort()

```
void heapify(int arr[], int length, int index) {
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < length && arr[left] > arr[largest]) {
        largest = left;
```

```
    }

    if (right < length && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != index) {
        swap(arr[index], arr[largest]);
        heapify(arr, length, largest);
    }
}

void heapsort(int arr[], int length) {
    // Build heap (rearrange array)
    for (int i = length / 2 - 1; i >= 0; i--) {
        heapify(arr, length, i);
    }

    // One by one extract an element from heap
    for (int i = length - 1; i >= 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

g. Radix_sort() & Count_sort()

```
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int arr[], int length, int exp)
{
    // output array
    int output[length];
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < length; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = length - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < length; i++)
        arr[i] = output[i];
}
```

```
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int length)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, length);
    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, length, exp);
}
```

h. Bucket_sort()

```
void bucketsort(float arr[], int length) {
const int bucketSize = 10; // Define the number of buckets
float maxVal = *max_element(arr, arr + length);
// Find the maximum value in the array
float minVal = *min_element(arr, arr + length);
// Find the minimum value in the array
// Create an array of buckets
float buckets[bucketSize][length];
// Initialize buckets
for (int i = 0; i < bucketSize; i++) {
    fill_n(buckets[i], length, -1);
// Initialize each bucket with -1
}
// Scatter the array elements into buckets
for (int i = 0; i < length; i++) {
    int index = (int)((arr[i] - minVal) / (maxVal -     minVal) *
(bucketSize - 1));
    int j = 0;
    while (buckets[index][j] != -1) {
        j++;
    }
    buckets[index][j] = arr[i];
}


    // Sort individual buckets and merge them back into the original
  array
    int index = 0;
    for (int i = 0; i < bucketSize; i++) {
        InsertionSort(buckets[i], length);
        for (int j = 0; j < length; j++) {
```

```
                if (buckets[i][j] != -1) {
                    arr[index++] = buckets[i][j];
                }
            }
        }
    }
```

Q2) Main Function:

```
    int main()
    {
        int n, store[500];
        const char* file_open = "array_data.txt";
        readdata(file_open, store, n);
        printing(store, n);



        cout<<"HeapSort Applying................"<<endl;
        // Measure run-time for HeapSort
        auto start = high_resolution_clock::now();

        heapsort(store, n);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(stop - start);
        cout << "HeapSort Time: " << duration.count() << " milliseconds"
    << endl;
        printing_sorted_arrays(store,n);



        cout<<"CountSort Applying................"<<endl;
        start = high_resolution_clock::now();
        CountSort(store, n);
        stop = high_resolution_clock::now();
        duration = duration_cast<milliseconds>(stop - start);
        cout << "CountSort Time: " << duration.count() << " milliseconds"
    << endl;
        printing_sorted_arrays(store,n);



        cout<<"RadixSort Applying................"<<endl;
```

```
    start = high_resolution_clock::now();
    radixsort(store, n);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "RadixSort Time: " << duration.count() << " milliseconds"
<< endl;
    printing_sorted_arrays(store,n);



    cout<<"BubbleSort Applying................"<<endl;
    start = high_resolution_clock::now();
    bubbleSort(store, n);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "BubbleSort Time: " << duration.count() << "
milliseconds" << endl;
    printing_sorted_arrays(store,n);



    cout<<"QuickSort Applying................"<<endl;
    start = high_resolution_clock::now();
    quickSort(store, 0, n - 1);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "QuickSort Time: " << duration.count() << " milliseconds"
<< endl;
    printing_sorted_arrays(store,n);



    cout<<"BucketSort Applying................"<<endl;
    start = high_resolution_clock::now();
    int len = 9;
    float array[len] = {8.48, 5.27, 9.10, 7.89, 3.01, 2.48, 6.32,
1.95, 1.27};
    bucketsort(array, n);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "BucketSort Time: " << duration.count() << "
milliseconds" << endl;
    fprinting_sorted_arrays(array,len);
```

```
    cout<<"InsertionSort Applying................"<<endl;
    start = high_resolution_clock::now();
    insertionSort(store, n);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "InsertionSort Time: " << duration.count() << "
milliseconds" << endl;
    printing_sorted_arrays(store,n);


    cout<<"MergeSort Applying................"<<endl;
    start = high_resolution_clock::now();
    merge_sort(store,0,n-1);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "MergeSort Time: " << duration.count() << " milliseconds"
<< endl;
    printing_sorted_arrays(store,n);


    cout<<"SelectionSort Applying................"<<endl;
    start = high_resolution_clock::now();
    selectionSort(store, n);
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "SelectionSort Time: " << duration.count() << "
milliseconds" << endl;
    printing_sorted_arrays(store,n);
    return 0;
}
```

Methodology:

Certainly! Our code executes various sorting algorithms on an array of integers read from a file, measuring their execution times using `std::chrono`. After sorting, it prints the sorted arrays along with the time taken by each algorithm. To enhance readability and efficiency, you can use concise naming conventions, organize the code into functions for each sorting algorithm, and optimize where possible. For visualizing the results, you can plot execution time against the number of entries using a logarithmic scale for the x-axis, with each algorithm represented by a different color. This approach helps in comparing the performance of sorting algorithms and understanding their efficiency for different input sizes.

Header files:

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <chrono>
#include <time.h>
#include <random>
using namespace std;
using namespace std::chrono;
```

Random Data Generator:

```
void generateRandomData(int arr[], int size) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 1000);
    for (int i = 0; i < size; ++i) {
        arr[i] = dis(gen);
    }
}
```

Saving time data in CSV file:

```
void saveTimingData(const string& filename, const string& algorithm, int
size, long long duration) {
    ofstream file(filename, ios::app);
    if (file.is_open()) {
        file << algorithm << "," << size << "," << duration << endl;
        file.close();
    } else {
        cout << "Unable to open file: " << filename << endl;
    }
}
```

Results:

Outputs on terminal window of vscode.

```
[0]: 8
[1]: 7
[2]: 6
[3]: 8
[4]: 5
[5]: 10
[6]: 27
[7]: 68
[8]: 55
[9]: 99
HeapSort Applying................
HeapSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
CountSort Applying................
CountSort Time: 0 milliseconds
```

Fig:01

```
CountSort Applying................
CountSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
RadixSort Applying................
RadixSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
BubbleSort Applying................
BubbleSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
QuickSort Applying................
QuickSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
BucketSort Applying................
BucketSort Time: 0 milliseconds
[0]: 5.89505e-039[1]: 1.27[2]: 1.95[3]: 2.48[4]: 3.01[5]: 5.27[6]: 6.32[7]: 7.89[8]: 8.48
```

Fig:02

```
InsertionSort Applying................
InsertionSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
MergeSort Applying................
MergeSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
SelectionSort Applying................
SelectionSort Time: 0 milliseconds
[0]: 5[1]: 6[2]: 7[3]: 8[4]: 8[5]: 10[6]: 27[7]: 55[8]: 68[9]: 99
PS D:\6th Semester\DSA\Lab1\DSA_lab>
```

Fig:03

Python Graph: