

R. V. COLLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagavi)
Approved By AICTE, New Delhi, Accredited By NBA, New Delhi
R.V. Vidyanikethan, Bengaluru- 560059

Department of Master of Computer Applications



I Semester
Assignment Report
On
Data Structures Using C
18MCA13

Different sorting technique's functions in a Header File.

Submitted by

Aditya Raj
1RD18MCA01

January -2019

R. V. COLLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagavi)
Approved By AICTE, New Delhi, Accredited By NBA, New Delhi
R.V. Vidyanikethan, Bengaluru- 560059

Department of Master of Computer Applications



CERTIFICATE

This is to certify that **Mr. Aditya Raj (1RD18MCA01)** have successfully completed the Assignment on **Data Structures Using C (18MCA13)** on **Different sorting technique's functions in a Header File** in a partial fulfillment of I semester MCA during the academic year 2018-19.

Max Mark	Marks Obtained
30	

Mr. Krishnan.R
Assistant Professor
Department of MCA
R.V. College of Engineering
Bengaluru - 560059

Dr. Andhe Dharani
Professor and Director
Department of MCA
R.V. College of Engineering
Bengaluru - 560059

INDEX

Sl No	Chapter	Page No
1.	Introduction	
2.	Literature Survey	
3.	Project Description	
4.	Code Snippet	
5.	Screen shots	
6.	Test Result and Conclusion	

(Report Contents)

Front Page

Certificate Page

Acknowledgement

Chapter – 1 Introduction (2 – 3 Pages)

Chapter – 2 Literature Survey (8 – 10 Pages)

Chapter – 3 Project Description (3 – 5 Pages)

Chapter – 4 Code Snippet (3 – 4 Pages)

Chapter – 5 Screen Shots (3 – 4 Pages)

[don't put black color screen shots]

Chapter – 6 Test Result and Conclusion (1 – 3 Pages)

/* USE TIME ROMAN, font 12 for text, font 14 for headings, font 12 bold for subheadings, line spacing 1.5, justify */

Introduction

Abstract

Sorting is a basic task in many types of computer applications. Especially when large amounts of data are to be sorted, efficiency becomes a major issue. There are many different sorting algorithms and even more ways in which they can be implemented. The efficiency of real implementations is often at least as important as the theoretical efficiency of the abstract algorithm. For example, Quicksort is well-known to perform very well in most practical situations, regardless of the fact that many other sorting algorithms have a better worst-case behavior. The goal of this master thesis is to make a survey of sorting algorithms and discuss and compare the differences in both theory and practice. There are several features that interests in this thesis such as finding possible implementations of each algorithm and discuss their properties and administer considerable experiments in order to obtain empirical data about their practical efficiency in different situations. Finally, we present the comparison of different sorting algorithms with their practical efficiency and conclude the theoretical findings and the knowledge gained from this study.

Need of Sorting

One of the basic areas of the computer science is Data Structure. Sorting is an important issue in Data Structure which creates the sequence of the list of items. Although numbers of sorting algorithms are available, it is all the more necessary to select the best sorting algorithm. Therefore, sorting problem has attracted a great deal of research as sorting technique is very often used in a large variety of important applications so as to arrange the data in ascending or descending order. This paper presents that sorting is an important area of study in computer science. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed.

Sorting is important in programming for the same reason it is important in everyday life. It is easier and faster to locate items in a sorted list than unsorted. Sorting algorithms can be used in a program to sort an array for later searching or writing out to an ordered file or report. In computer science, arranging in an ordered sequence is called "sorting". Sorting is a common operation in many applications, and efficient algorithms to perform it have been developed. The most common uses of sorted sequences are: making lookup or search efficient; making merging of sequences efficient.

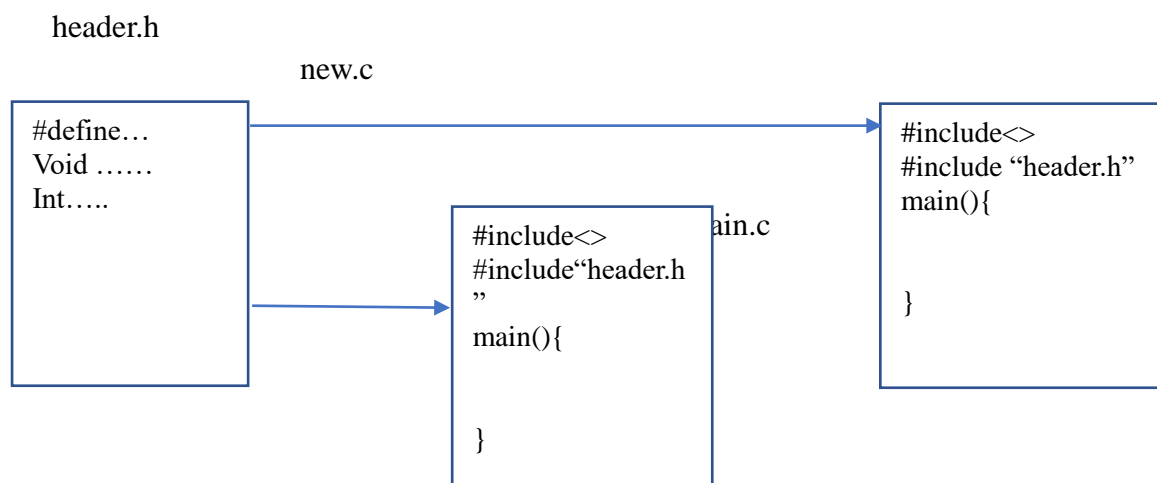
Analysis and Review of Sorting Algorithms

Header files

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

The application programming interface (API) of the C standard library is declared in a number of header files. Each header file contains one or more function declarations, data type definitions, and macros.

Header Files are included in the beginning of the program. It is a predefined code and we have to simply include the file. Thus, writing less is an advantage. It has .h extension. It contains C declarations and macro definition.



What should be in the header files for a complex project?

In C, the contents of a module consist of structure type (struct) declarations, global variables, and functions. The functions themselves are normally defined in a source file (a “.c” file). Except for the main module, each source (.c) file has a header file (a “.h” file) associated with it that provides the declarations needed by other modules to make use of this module.

The idea is that other modules can access the functionality in module X simply by `#include "X.h"` for the header file, and the linker will do the rest. The code in **X.c** needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X's code into the

final executable without needing to recompile it, which enables the Unix make utility and IDEs to work very efficiently.

A well-organized C program has a good choice of modules, and properly constructed header files that make it easy to understand and access the functionality in a module. They also help ensure that the program is using the same declarations and definitions of all of the program components. This is important because compilers and linkers need help in enforcing the One Definition Rule.

Furthermore, well-designed header files reduce the need to recompile the source files for components whenever changes to other components are made. The trick is reduce the amount of “coupling” between components by minimizing the number of header files that a module’s header file itself #includes. On very large projects, minimizing coupling can make a huge difference in “build time” as well as simplifying the code organization and debugging.

Sorting Algorithms

1. Selection Sort

This algorithm works by **selecting** the **smallest unsorted** item and then **swapping** it with the item in the **next position** to be filled.

The selection sort works as follows: you look through the entire array for the **smallest** element, once you find it you **swap** it (the smallest element) with the **first** element of the array. Then you look for the **smallest** element in the remaining array (an array without the first element) and swap it with the **second** element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

```
void selectionSort(int[] ar){  
  
    for (int i = 0; i < ar.length-1; i++)  
    {  
        int min = i;  
  
        for (int j = i+1; j < ar.length; j++)  
            if (ar[j] < ar[min]) {  
                min = j;  
  
                int temp = ar[i];  
                ar[i] = ar[min];  
                ar[min] = temp;  
            }  
    }  
}
```

```
Step 1 - Set MIN to location 0  
Step 2 - Search the minimum element in  
the list  
Step 3 - Swap with value at location  
MIN  
Step 4 - Increment MIN to point to next  
element  
Step 5 - Repeat until list is sorted
```

}} }

Example.

29, 64, 73, 34, 20,
20, 64, 73, 34, 29,
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73

This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where n is the number of items.

2. Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position. The resulting array after k iterations has the property where the first $k + 1$ entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position.

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73

Step 1 - If it is the first element, it is already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:

we need 0 comparisons to insert the first element
we need 1 comparison to insert the second element
we need 2 comparisons to insert the third element

...

we need (N-1) comparisons (at most) to insert the last element

Totally,

$$1 + 2 + 3 + \dots + (N-1) = O(n^2)$$

2 Best Case Time Complexity [Big-omega]: **$O(n)$**

3 Average Time Complexity [Big-theta]: **$O(n^2)$**

4 Space Complexity: **$O(1)$**

3. Bubble Sort

In this task, the goal is to sort an array of elements using the bubble sort algorithm. This uses the divide and conquer method of algorithm. The elements must have a total order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers. The bubble sort is generally considered to be the simplest sorting algorithm. Because of its simplicity and ease of visualization, it is often taught in introductory computer science courses. Because of its abysmal $O(n^2)$ performance, it is not used often for large (or even medium-sized) datasets.

```
void BubbleSort(int a[])
{
    int i,j;
    for (i=MAXLENGTH; --i >=0;) {
        swapped = 0;
        for (j=0; j<i;j++) {
            if (a[j]>a[j+1]) {
                Swap[a[j],a[j+1]]; swapped=1;
            }
        }
        if (!swapped) return; } }
```

```
begin BubbleSort(list)
    for all elements of list
        if list[i] > list[i+1]
            swap(list[i],
list[i+1])
        end if
    end for
    return list
end BubbleSort
```

4. Quick Sort

Quick sort is a comparison sort developed by Tony Hoare. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set. Quick sort algorithm can be depicted as follows:

QUICKSORT (A)

```
1:   step ← m;
2:   while step > 0
3:     for (i ← 0 to n with increment 1)
4:       do temp ← 0;
5:       do j ← i;
6:       for (k ← j + step to n with increment step)
7:         do temp ← A[k];
8:         do j ← k - step;
9:         while (j ≥ 0 && A[j] > temp)
10:          do A[j + step] = A[j];
11:          do j ← j - step;
12:          do Array[j] + step ← temp;
13:          do step ← step / 2;
```

5. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merg() function** is used for merging two halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

Complexity of Mergesort

Suppose $T(n)$ is the number of comparisons needed to sort an array of n elements by the Merge Sort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely $n/2$. Each part can be sort in $T(n/2)$. Finally, on the last step we perform $n-1$ comparisons to merge these two parts in one. All together, we have the following equation

$$T(n) = 2 * T(n/2) + n - 1$$

The solution to this equation is beyond the scope of this course. However, I will give you a reasoning using a binary tree. We visualize the merge sort dividing process as a tree.

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c
end procedure
```



6. Heap Sort

Heapsort is a comparison-based sorting algorithm. Heapsort is part of the selection sort family; it improves on the basic selection sort by using a logarithmic-time priority queue rather than a linear-time search. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case $O(n \log n)$ runtime. Heapsort is an in-place algorithm, but it is not a stable sort. It was invented by J. W. J. Williams in 1964.

	Worst Case	Average Case	Best Case
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Figure 2: Comparison of sorting algorithms

Code Snippet

Header File :-

File Name :- "sort.h"

This is a header file which contains all the function and business logics related to all sorting techniques.

// bubble sort function

```
void bubbleSort(int arr[], int size)
{
    int i, j, temp;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    // print the sorted array
    printf("\nSorted Array using Bubble sort: \n");
    for(i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

// selection sort

```
void SelectionSort(int arr[], int size)
{
    int i;
    for(i=0; i<size-1; i++)
    {
        int Imin = i;
```

```

        for(int j=i+1; j<size; j++)
        {
            if( arr[j] < arr[Imin] )
            {
                Imin = j;
            }
        }
        int temp = arr[Imin];
        arr[Imin] = arr[i];
        arr[i] = temp;
    }

    printf("\nSorted Array using Selection sort: \n");
    for(int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
}

// Insertion sort

void InsertionSort(int arr[], int size)
{
    int i,hole,value;
    for( i=1; i<size; i++)
    {
        value = arr[i];
        hole = i;
        while( hole>0 && arr[hole-1]>value)
        {
            arr[hole] = arr[hole-1];
            hole--;
        }
        arr[hole] = value ;
    }
    printf("\nSorted Array using Insertion sort: \n");
    for(int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
}

// Merge sort

// Merges two subarrays of arr[.

```

```

// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

```

```

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

```

```

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

```

// quick sort

```

void quicksort(int list[], int low, int high)
{
    int pivot, i, j, temp;
    if (low < high)
    {
        pivot = low;
        i = low;
        j = high;
        while (i < j)
        {
            while (list[i] <= list[pivot] && i <= high)
            {
                i++;
            }

```



```

    }
    while (list[j] > list[pivot] && j >= low)
    {
        j--;
    }
    if (i < j)
    {
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
}
temp = list[j];
list[j] = list[pivot];
list[pivot] = temp;
quicksort(list, low, j - 1);
quicksort(list, j + 1, high);
}
}

```

// heap sort

```

void heapsort(int a[], int size)
{
    int i, t;
    heapify(a, size);
    for (i = size - 1; i > 0; i--)
    {
        t = a[0];
        a[0] = a[i];
        a[i] = t;
        adjust(a, i);
    }
    for (i = 0; i < size; i++)
        printf("\t%d", a[i]);
    printf("\n");
}

void heapify(int a[], int n)
{
    int k, i, j, item;
    for (k = 1; k < n; k++)
    {
        item = a[k];

```

```

        i = k;
        j = (i - 1) / 2;
        while ((i > 0) && (item > a[j]))
        {
            a[i] = a[j];
            i = j;
            j = (i - 1) / 2;
        }
        a[i] = item;
    }
}

void adjust(int a[], int na)
{
    int i, j, item;
    j = 0;
    item = a[j];
    i = 2 * j + 1;
    while (i <= na - 1)
    {
        if (i + 1 <= na - 1)
            if (a[i] < a[i + 1])
                i++;
        if (item < a[i])
        {
            a[j] = a[i];
            j = i;
            i = 2 * j + 1;
        }
        else
            break;
    }
    a[j] = item;
}

```

Interface File:-

File Name:- **“interface.c”**

```

#include<stdio.h>
#include<stdlib.h>
#include"sort.h"
#define MAX 100

```

```

void main(int argc, char const *argv[])
{
    int arr[MAX], i, step, temp;
    // atoi() is used to convert string to integer
    int limit = atoi(argv[1]);

    if (argc > 2)
    {
        printf("No more arguments. Only two argument is allowed.\n");
        exit(1);
    }

    // ask user for number of elements to be sorted
    printf("\n*****\n");
    printf("Enter the number of elements to be sorted: ");
    //scanf("%d", &limit);
    printf("\n*****\n");
    // input elements if the array
    for (i = 0; i < limit; i++)
    {
        printf("Element %d is %d \n", i + 1, arr[i] = rand() % 100 + 1);
        //arr[i] = rand() % 100 + 1;
    }
    printf("\n*****\n");
    printf("Array elements are :- \n");
    for(i = 0; i < limit; i++)
    {
        printf("\t%d", arr[i]);
    }

    printf("\n*****\n");

    while (1) {
        int ch;
        printf("\n*****\n");
        printf("\n1-Bubble Sort \n2-Selection Sort \n3-Insertion Sort \n4-Merge Sort \n5-Quick Sort \n6-Heap Sort \n7-Exit\n");
        printf("\n*****\n");
        printf("Enter your choice\n");
        scanf("%d",&ch);

        printf("\n*****\n");

```

```

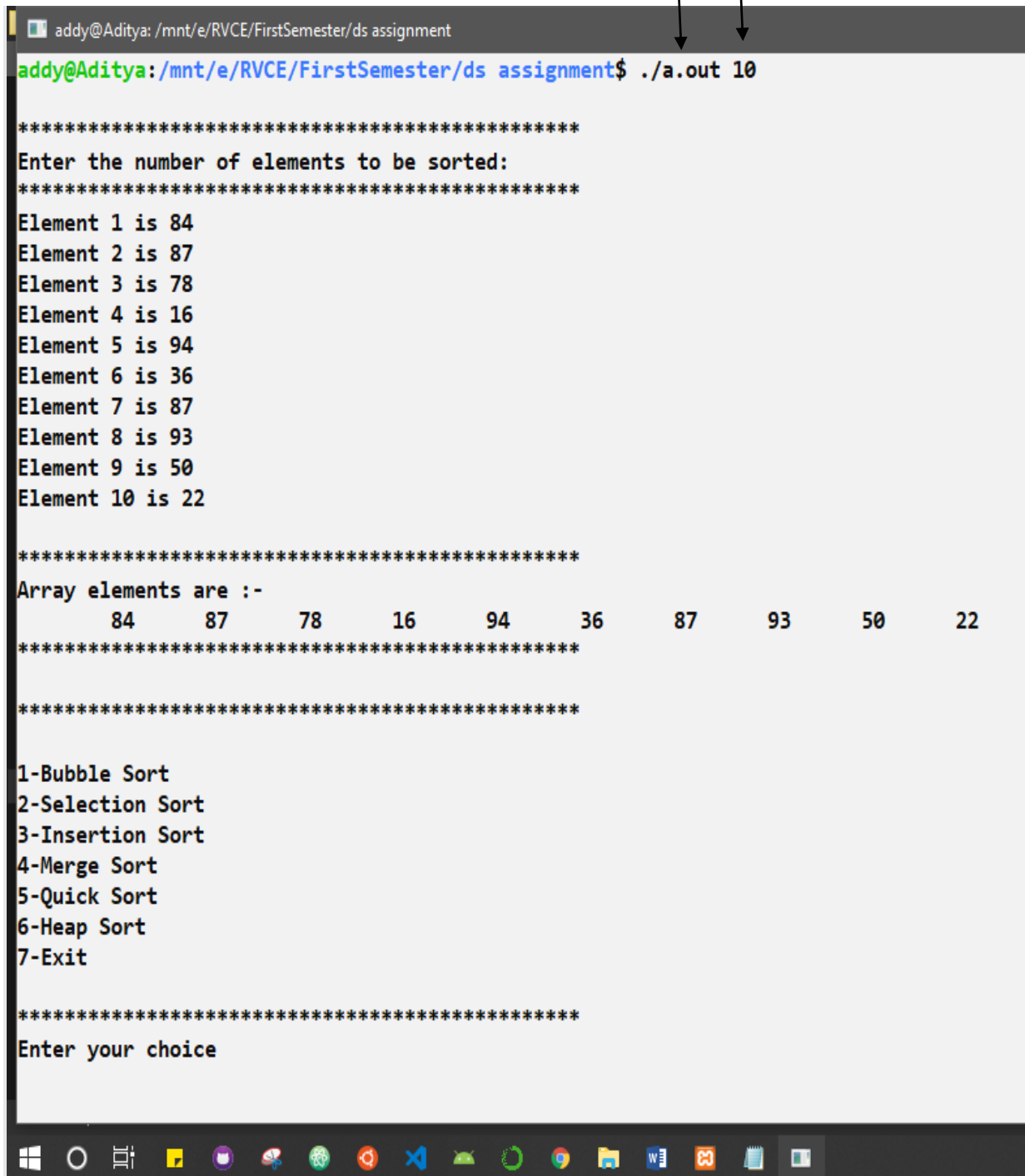
switch (ch) {
case 1:
    printf("\n***** Bubble sort *****\n");
    bubbleSort(arr,limit);
    printf("\n*****\n");
    break;
case 2:
    printf("\n***** Selection sort *****\n");
    SelectionSort(arr,limit);
    printf("\n*****\n");
    break;
case 3:
    printf("\n***** Insertion sort *****\n");
    InsertionSort(arr,limit);
    printf("\n*****\n");
    break;
case 4:
    printf("\n***** Merge sort *****\n");
    mergeSort(arr, 0, limit - 1);
    printf("\nSorted array using merge sort \n");
    for (i=0; i < limit; i++)
        printf("%d ", arr[i]);
    printf("\n*****\n");
    break;
case 5:
    printf("\n***** Quick sort *****\n");
    quicksort(arr, 0, limit - 1);
    for (i=0; i < limit; i++)
        printf("%d ", arr[i]);
    printf("\n*****\n");
    break;
case 6:
    printf("\n***** Heap sort *****\n");
    heapsort(arr, limit);
    break;
case 7:
    exit(0);
} }

```

Screen Shots

Executable file

Array size



```
addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment
addy@Aditya:/mnt/e/RVCE/FirstSemester/ds assignment$ ./a.out 10

*****
Enter the number of elements to be sorted:
*****
Element 1 is 84
Element 2 is 87
Element 3 is 78
Element 4 is 16
Element 5 is 94
Element 6 is 36
Element 7 is 87
Element 8 is 93
Element 9 is 50
Element 10 is 22

*****
Array elements are :-
      84      87      78      16      94      36      87      93      50      22
*****

*****
1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

*****
Enter your choice
```

```
addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment
*****

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

*****

Enter your choice
1

*****

***** Bubble sort *****

Sorted Array using Bubble sort:
16 22 36 50 78 84 87 87 93 94
*****

*****

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

*****

Enter your choice
_
```

addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice

3

***** Insertion sort *****

Sorted Array using Insertion sort:

16 22 36 50 78 84 87 87 93 94

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice

■



addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice

4

***** Merge sort *****

Sorted array using merge sort

16 22 36 50 78 84 87 87 93 94

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice



addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice

5

***** Quick sort *****

16 22 36 50 78 84 87 87 93 94

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice



addy@Aditya: /mnt/e/RVCE/FirstSemester/ds assignment

16 22 36 50 78 84 87 87 93 94

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice

6

***** Heap sort *****

16 22 36 50 78 84 87 87 93 94

1-Bubble Sort
2-Selection Sort
3-Insertion Sort
4-Merge Sort
5-Quick Sort
6-Heap Sort
7-Exit

Enter your choice



- **Data structure:**

Sorting of data uses Array data structure. Because for sorting we basically search the elements linearly from the first index element to the nth element. So Array Data Structure is best suitable for sorting of elements.

- **Functional test cases:**

1. Input array is empty
2. Input array contains only one element
3. Input array is already sorted in order
4. Input array is sorted in reversed order
5. Input array contains repeated element

- **Conclusion**

This paper discusses comparison-based sorting algorithms function using user defined **Header File**. It analyses the performance of these algorithms for the same number of elements. It then concludes that selection sort shows better performance than Quick sort but being the simple structure selection sort is more preferred and widely used. It is clear that both sorting techniques are not that popular for the large arrays because as the arrays size increases both timing is slower for integer and string, but generally the study indicate that integer array have faster CPU time than string arrays. Both have the upper bound running time $O(n^2)$. Bubble Sort is a very simple algorithm that is only suitable for small lists. There are lots of alternative sorting algorithms that are more performant than Bubble Sort. Bubble Sort is widely considered to be the least performant of all the established sorting algorithms.

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ Mergesort is optimal. However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice. Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

- **References**

1. Introduction to Algorithms is a book by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. C Header File Guidelines David Kieras, EECS Dept., University of Michigan
December 19, 2012.
3. https://www.tutorialspoint.com/data_structures_algorithms/index.htm
4. International Journal of Scientific Engineering and Research (IJSER)