**PROJECT TITLE:**

OPTIMAL SORTING ALGORITHM SELECTOR

**ABSTRACT:**

The problem of sorting is a problem that arises frequently in computer programming. Many sorting algorithms have been developed and improved to make sorting fast. As a measure of performance mainly the average number of the operations or the average execution time of these algorithms have been investigated and compared. There is no one sorting method that is best for every situation. Some of the factors to be considered in choosing a sorting algorithm include the size of the list to be sorted, the programming effort, no of words of main memory available, the size of the disk, the extent to which the list is already ordered.

Purpose of this project is to make a custom library, which will suggest best Sorting algorithm for the certain type of the data based on its time complexity.

**INTRODUCTION:**

Purpose of this project is to make a custom library, which will suggest best Sorting algorithm for the certain type of the data based on its time complexity.

Although there is no “best” sorting algorithm, we will, nevertheless, define the best sorting algorithm as the one whose weighted sum of the number of comparison, moves and exchange is minimal. Many algorithms that have the same efficiency do not have the same speed on the same input. Indeed, all the sorting algorithms are problem specific means they do well on special kind of problems like some are useful for small number element, some are for large list, some are suitable for the repeated value, some are useful for float value and some are useful for integer value.

Sorting techniques can be of various types, differentiated by their efficiency and space requirements. Here are some sorting techniques which are used in the project are:

* Insertion Sort
* Selection Sort
* Quick Sort
* Merge Sort
* Heap Sort
* Bubble sort

So, in this project we demonstrate these sorting algorithms and compared them and suggest the optimal sorting algorithm for the set of elements entered by the user.

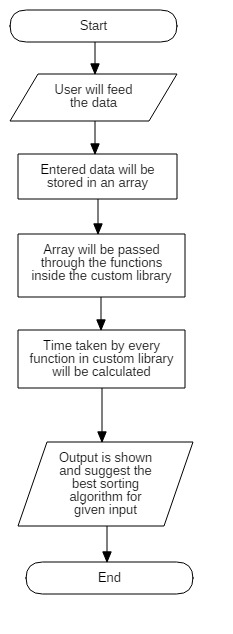
**OBJECTIVE ACHIEVED:**

Suggesting the best sorting algorithm for the set of elements entered by the user along with the time taken by each of the sorting algorithm.

**METHODOLOGY:**

**Techniques:**

**flowchart**

****

**Pseudocode:**

bestofall()

{

initialize array to store time

initialize a variable to store shortest time

intialize variable of type clock\_t to store ticks

initialize 2D array to store name of sorting

//MERGE SORT

start<-clock()

run mergesort function (called from custom library)

end<-clock()

calculate time //end-start

name[]<-"Merge Sort"

print time taken

//QUICK SORT

start<-clock()

run quicksort function (called from custom library)

end<-clock()

calculate time

name[]<-"Quick Sort"

print time taken

//INSERTION SORT

start<-clock()

run insertionsort function (called from custom library)

end<-clock()

calculate time

name[]<-"Insertion Sort"

print time taken

//SELECTION SORT

start<-clock()

run selectionsort function (called from custom library)

end<-clock()

calculate time

name[]<-"Selection Sort"

print time taken

//BUBBLE SORT

start<-clock()

run bubblesort function (called from custom library)

end<-clock()

calculate time

name[]<-"Bubble Sort"

print time taken

//HEAP SORT

start<-clock()

run heapsort function (called from custom library)

end<-clock()

calculate time

name[]<-"Heap Sort"

print time taken

while time\_taken has values

if time\_taken < x

put x =time\_taken

endif

endwhile

print the sorting algo name with smallest time taken.

}

main()

{

initialize n to store number of elements

take number of elements and the data for sorting

call bestofall function and pass array and array size

}

**Running code:**

1. **Function files**

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* BUBBLE SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void swap\_a (float a, float b)

{

float t = a;

a = b;

b = t;

}

// A function to implement bubble sort

void bubbleSort (float arr[], float n)

{

int i, j;

for (i = 0; i < n-1; i++)

// Last i elements are already in place

for (j = 0; j < n-i-1; j++)

if (arr[j] > arr[j+1])

swap\_a(arr[j], arr[j+1]);

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* HEAP SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void swap\_b(float a, float b)

{

float t = a;

a = b;

b = t;

}

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(float arr[], 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 && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i)

{

swap\_b(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// main function to do heap sort

void heapSort(float arr[], int n)

{

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

for (int i=n-1; i>=0; i--)

{

// Move current root to end

swap\_b(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*INSERTION SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void insertionSort(float arr[], int n)

{

int i, j;

float key;

for (i = 1; i < n; 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;

}

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MERGE SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(float arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

float 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 \*/

double mergeSort(float 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 swap\_c(float a, float b)

{

float t = a;

a = b;

b = t;

}

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

int partition (float arr[], int low, int high)

{

float pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap\_c(arr[i], arr[j]);

}

}

swap\_c(arr[i + 1], arr[high]);

return (i + 1);

}

/\* The main function that implements QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

void quickSort(float arr[], int low, int high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

int pi = partition(arr, low, high);

// Separately sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SELECTION SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void swap\_d(float a, float b)

{

float t = a;

a = b;

b = t;

}

void selectionSort(float arr[], int n)

{

int i, j, min\_idx;

// One by one move boundary of unsorted subarray

for (i = 0; i < n-1; i++)

\* {

// Find the minimum element in unsorted array

min\_idx = i;

for (j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first element

swap\_d(arr[min\_idx], arr[i]);

}

}

**b)Header file**

void mergeSort(float arr[], int , int);

void quickSort(float arr[], int , int);

void insertionSort(float arr[], int);

void selectionSort(float arr[], int);

void bubbleSort(float arr[], int);

void heapSort(float arr[], int);

**c) main file**

#include<stdlib.h>

#include<stdio.h>

#include<time.h>

#include<string.h>

#include"sorttime.h"

void bestofall(float arr[], int arr\_size)

{

double time\_taken[6], x;

clock\_t start,end;

char name[6][20];

//MERGE SORT

start=clock();

mergeSort(arr, 0, arr\_size - 1);

end=clock();

time\_taken[0]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[0],"MERGE SORT");

printf("\n Merge Sort: %f", time\_taken[0]);

//QUICK SORT

start=clock();

quickSort(arr, 0, arr\_size - 1);

end=clock();

time\_taken[1]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[1],"QUICK SORT");

printf("\n Quick Sort: %f",time\_taken[1]);

//INSERTION SORT

start=clock();

insertionSort(arr, arr\_size-1);

end=clock();

time\_taken[2]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[2],"INSERTION SORT");

printf("\n Insertion Sort: %f",time\_taken[2]);

//SELECTION SORT

start=clock();

selectionSort(arr, arr\_size-1);

end=clock();

time\_taken[3]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[3],"SELECTION SORT");

printf("\n Selection Sort: %f",time\_taken[3]);

//BUBBLE SORT

start=clock();

bubbleSort(arr, arr\_size-1);

end=clock();

time\_taken[4]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[4],"BUBBLE SORT");

printf("\n Bubble Sort: %f",time\_taken[4]);

//HEAP SORT

start=clock();

heapSort(arr, arr\_size-1);

end=clock();

time\_taken[5]=((double)end-start)/(CLOCKS\_PER\_SEC/1000);

strcpy(name[5],"HEAP SORT");

printf("\n Heap Sort: %f",time\_taken[5]);

printf("\n");

x=time\_taken[0];

for(int i=0; i<6; i++)

{

if(time\_taken[i] <= x)

x = time\_taken[i];

}

printf("\n Following are the sorting technique(s) optimal for the given input: \n");

for(int i=0; i<6; i++)

if(x == time\_taken[i])

{

printf(" >> %s \n",name[i]);

}

printf("\n");

}

int main()

{

int n;

printf("\n Enter number of elements: ");

scanf("%d",&n);

float ar[n];

printf("\n Enter the data: ");

for(int i=0; i<n; i++)

{

scanf("%f",&ar[i]);

}

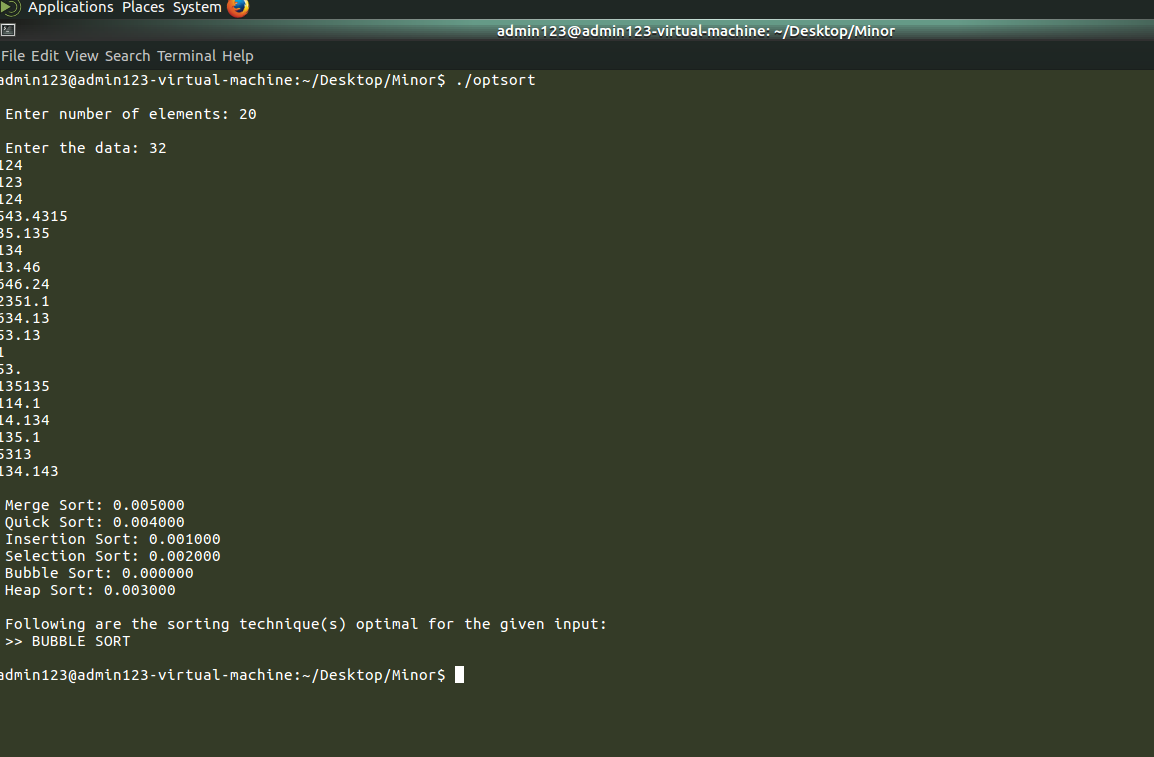
bestofall(ar,n);

getchar();

return 0;

}

**Execution result**

****

**REFERENCES:**

[1] Miraj Gul, Noorul Amin, M. Suliman.Deptt of IT "An Analytical Comparison of

Different Sorting Algorithms in Data Structure"(2015).

[2] Ms. Nidhi Chhajed, Mr. Imran Uddin, Mr. Simarjeet Singh Bhatia "A Comparison Based Analysis of Four Different Types of Sorting Algorithms in Data Structures with Their Performances" (2013).

[3] Miss. Pooja K. Chhatwani, Miss. Jayashree S. Somani, "Comparative Analysis &Performance of Different Sorting Algorithm in Data Structure"(Volume 3, Issue 11, November 2013.