

Dhruv Varia

210303105821

DAA Lab Manual

**Practical : 1**

*Implementation and Time analysis of Bubble, Selection and Insertion sorting algorithms for best case, average case & worst case.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Swap utility

void swap(long int\* a, long int\* b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

// Bubble sort

void bubbleSort(long int a[], long int n)

{

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

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

if (a[j] > a[j + 1]) {

swap(&a[j], &a[j + 1]);

}

}

}

}

// Insertion sort

void insertionSort(long int arr[], long int n)

{

long int i, key, j;

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;

}

}

// Selection sort

void selectionSort(long int arr[], long int n)

{

long int i, j, midx;

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

// Find the minimum element in unsorted array

midx = i;

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

if (arr[j] < arr[j])

midx = j;

// Swap the found minimum element

// with the first element

swap(&arr[midx], &arr[i]);

}

}

// Driver code

int main()

{

long int n = 10000;

int it = 0;

// Arrays to store time duration

// of sorting algorithms

double tim1[10], tim2[10], tim3[10];

printf("A\_size, Bubble, Insertion, Selection\n");

// Performs 10 iterations

while (it++ < 10) {

long int a[n], b[n], c[n];

// generating n random numbers

// storing them in arrays a, b, c

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

long int no = rand() % n + 1;

a[i] = no;

b[i] = no;

c[i] = no;

}

// using clock\_t to store time

clock\_t start, end;

// Bubble sort

start = clock();

bubbleSort(a, n);

end = clock();

tim1[it] = ((double)(end - start));

// Insertion sort

start = clock();

insertionSort(b, n);

end = clock();

tim2[it] = ((double)(end - start));

// Selection sort

start = clock();

selectionSort(c, n);

end = clock();

tim3[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li, %li, %li, %li\n",

n,

(long int)tim1[it],

(long int)tim2[it],

(long int)tim3[it]);

FILE \*fp=NULL;

fp=fopen("complexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

fp=fopen("complexity1.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim2[it]);

fp=fopen("complexity2.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim3[it]);

// increases the size of array by 10000

n += 10000;

}

// Code for auto run and load data into GNU Plot for graph….

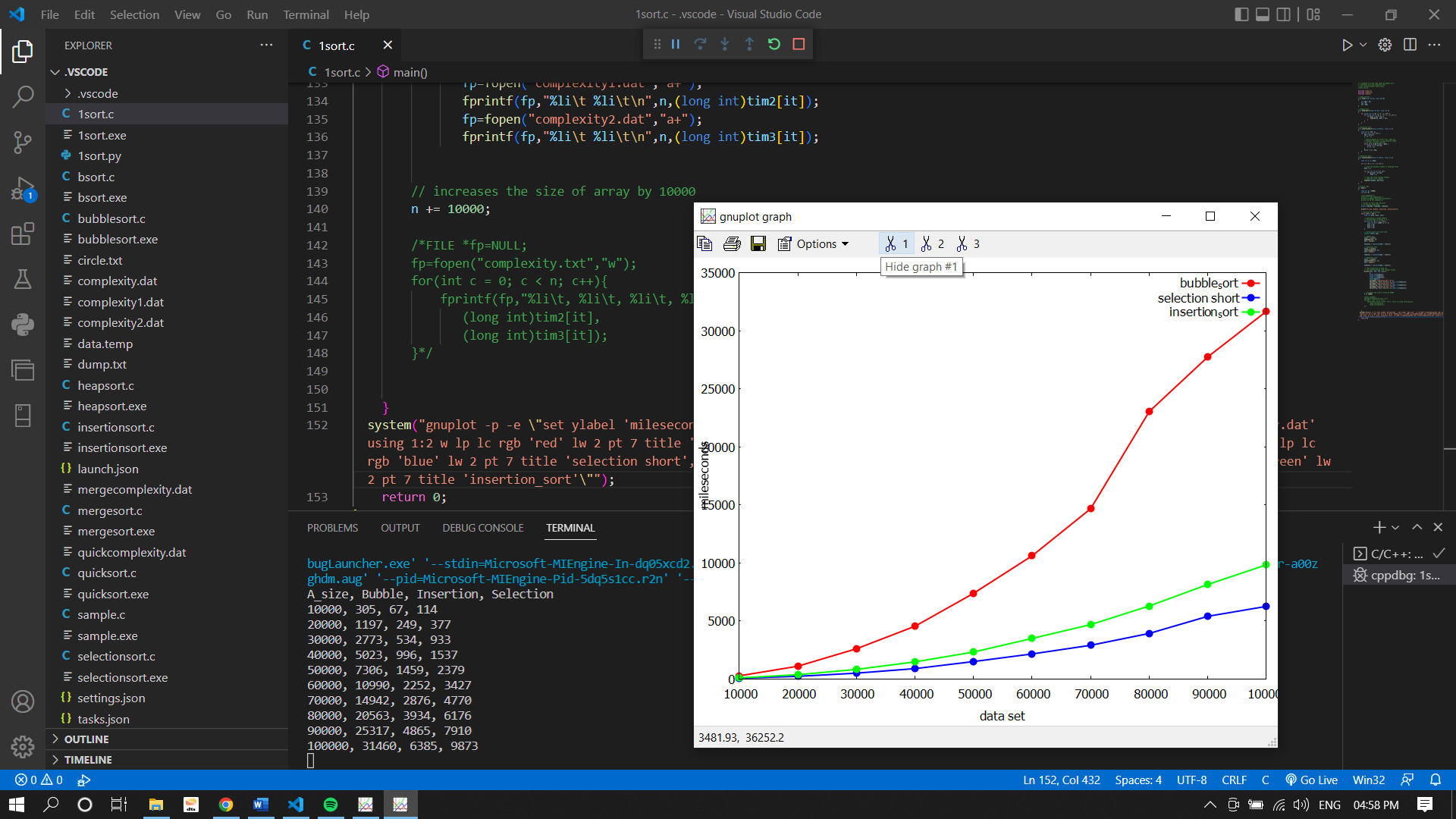
system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/complexity.dat' using 1:2 w lp lc rgb 'red' lw 2 pt 7 title 'bubble\_sort','D:/SEM 5 programming/DAA Lab/.vscode/complexity1.dat' using 1:2 w lp lc rgb 'blue' lw 2 pt 7 title 'selection short','D:/SEM 5 programming/DAA Lab/.vscode/complexity2.dat' using 1:2 w lp lc rgb 'green' lw 2 pt 7 title 'insertion\_sort'\"");

return 0;

}

**Output :**

**Avg Case:**

****

**Code 2 : With User input array**

#include<stdio.h>

#include<stdlib.h>

void display(int a[],int n);

void bubble\_sort(int a[],int n);

void selection\_sort(int a[],int n);

void insertion\_sort(int a[],int n);

void display(int arr[],int n)

{

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

{

printf(" %d ",arr[i]);

}

}

void bubble\_sort(int arr[],int n)

{

int i,j,temp;

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

{

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

{

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

{

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

printf("After Bubble sort Elements are : ");

display(arr,n);

}

void selection\_sort(int arr[],int n)

{

int i,j,temp;

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

{

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

{

if(arr[i]>arr[j])

{

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

}

printf("After Selection sort Elements are : ");

display(arr,n);

}

void insertion\_sort(int arr[],int n)

{

int i,j,min;

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

{

min=arr[i];

j=i-1;

while(min<arr[j] && j>=0)

{

arr[j+1]=arr[j];

j=j-1;

}

arr[j+1]=min;

}

printf("After Insertion sort Elements are : ");

display(arr,n);

}

int main()

{

int n,choice,i;

char ch[20];

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

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

{

scanf("%d",&arr[i]);

}

printf("\nPlease select any option Given Below for Sorting : \n");

while(1)

{

printf("\n1. Bubble Sort\n2. Selection Sort\n3. Insertion Sort\n4. Display Array.\n5. Exit the Program.@\_dv99\_,Enroll:821\n");

printf("\nEnter your Choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

bubble\_sort(arr,n);

break;

case 2:

selection\_sort(arr,n);

break;

case 3:

insertion\_sort(arr,n);

break;

case 4:

display(arr,n);

break;

case 5:

return 0;

default:

printf("\nPlease Select only 1-5 option ----\n");

}

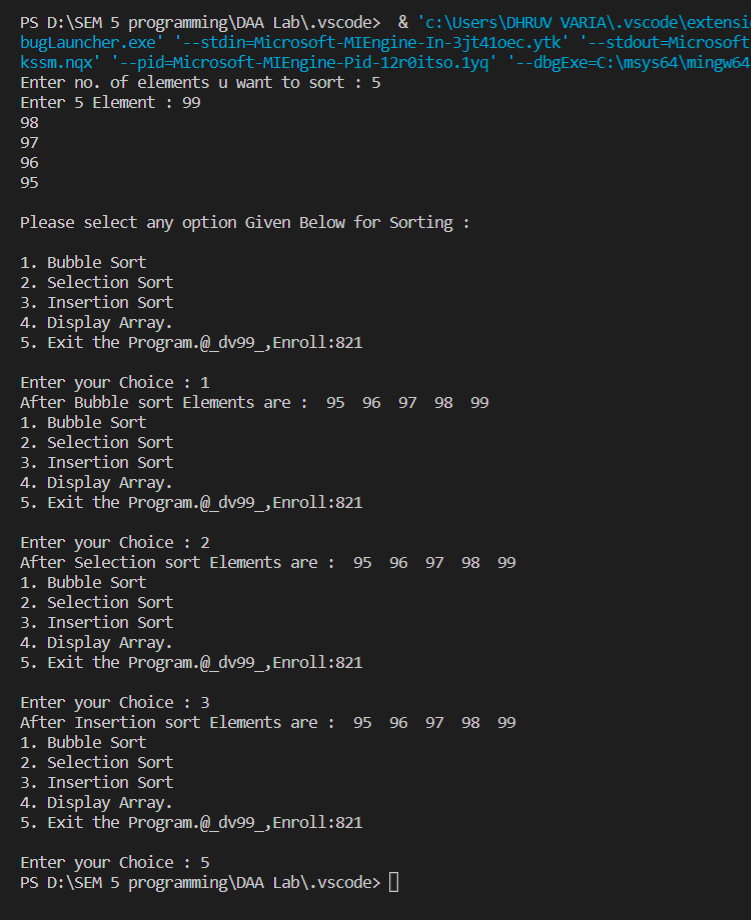
}

return 0;

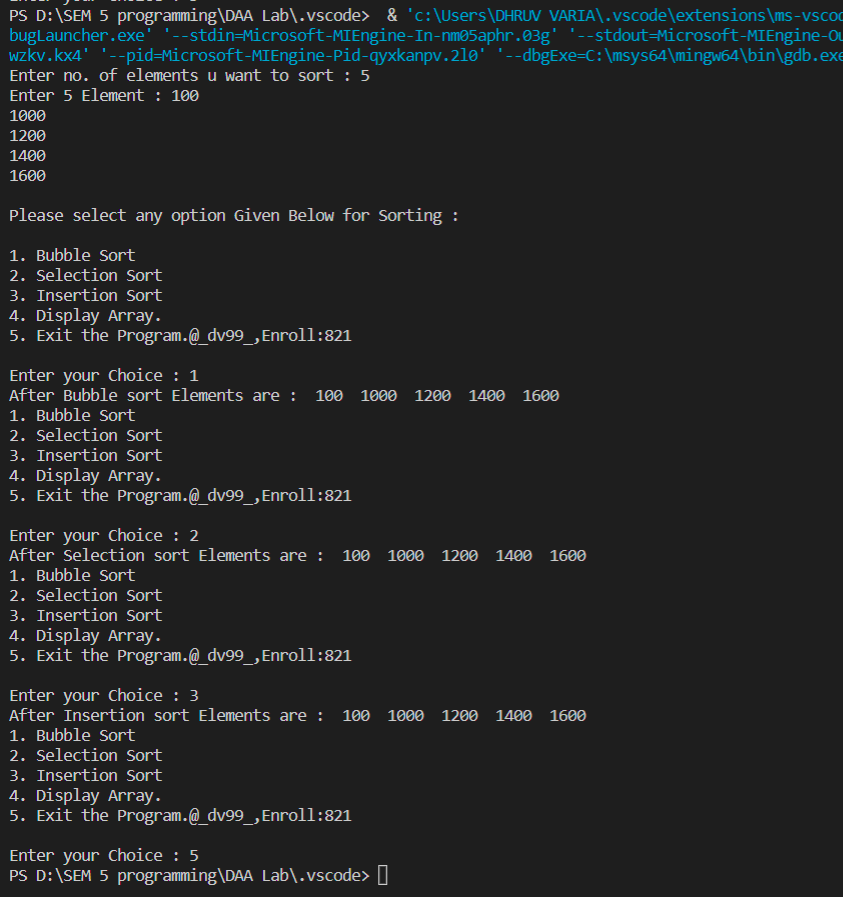
}

**Output :**

**Best Case**

****

**Worst Case**



**Practical : 2**

*Implementation and Time analysis of Max-Heap sort algorithm.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

/\* function to heapify a subtree. Here 'i' is the

index of root node in array a[], and 'n' is the size of heap. \*/

void heapify(int a[], int n, int i)

{

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left child

int right = 2 \* i + 2; // right child

// If left child is larger than root

if (left < n && a[left] > a[largest])

largest = left;

// If right child is larger than root

if (right < n && a[right] > a[largest])

largest = right;

// If root is not largest

if (largest != i) {

// swap a[i] with a[largest]

int temp = a[i];

a[i] = a[largest];

a[largest] = temp;

heapify(a, n, largest);

}

}

void heapSort(int a[], int n)

{

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

heapify(a, n, i);

// One by one extract an element from heap

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

/\* Move current root element to end\*/

// swap a[0] with a[i]

int temp = a[0];

a[0] = a[i];

a[i] = temp;

heapify(a, i, 0);

}

}

void printArr(int arr[], int n)

{

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

{

printf("%d", arr[i]);

printf(" ");

}

}

int main()

{

// using clock\_t to store time

clock\_t start, end;

start = clock();

int h=15;

int it = 0;

double tim1[10];

int a[h];

//int a[] = {7,6,5,4,3,2,1};

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

{

long int no = rand() % h + 1;

a[i] = no;

}

int n = sizeof(a) / sizeof(a[0]);

printf("Before sorting array elements are - \n");

printArr(a, n);

heapSort(a, n);

printf("\nAfter sorting array elements are - \n");

printArr(a, n);

end = clock();

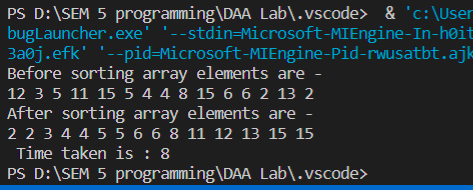
tim1[it] = ((double)(end - start));

printf("\n Time taken is : %li\n",(long int)tim1[it]);

return 0;

}

**Output :**



**Complexity :**

Time Complexity of this operation is O(Log n) because we insert the value at the end of the tree and traverse up to remove violated property of min/max heap.

1. Heapify()

In the heapify() function, we walk through the tree from top to bottom. The height of a binary tree (the root not being counted) of size n is log2 n at most.

The complexity for the heapify() function is accordingly O(log n).

1. Heapsort()

To initially build the heap, the heapify() method is called for each parent node – backward, starting with the last node and ending at the tree root.

A heap of size n has n/2 (rounded down) parent nodes:

Since the complexity of the heapify() method is O(log n) as shown above, the complexity for the heapsort() method is, therefore, maximum\* O(n log n).

The time complexity of Heapsort is : O(n log n)

**Practical : 3**

*Implementation and Time analysis of Merge Sort algorithms for Best case, Average case &Worst-case using Divide and Conquer.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// 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);

}

}

void printArray(int A[], int size)

{

int i;

for (i = 0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

int main()

{

long int n = 10000;

int it = 0;

double tim1[10];

while (it++ < 10) {

long int arr[n];

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

{

long int no = rand() % n + 1;

arr[i] = no;

}

clock\_t start, end;

int arr\_size = sizeof(arr) / sizeof(arr[0]);

start = clock();

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

end = clock();

tim1[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li %li \n", n,(long int)tim1[it]);

//For Saving in file. #Dhruv

FILE \*fp=NULL;

fp=fopen("mergecomplexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

n += 10000;

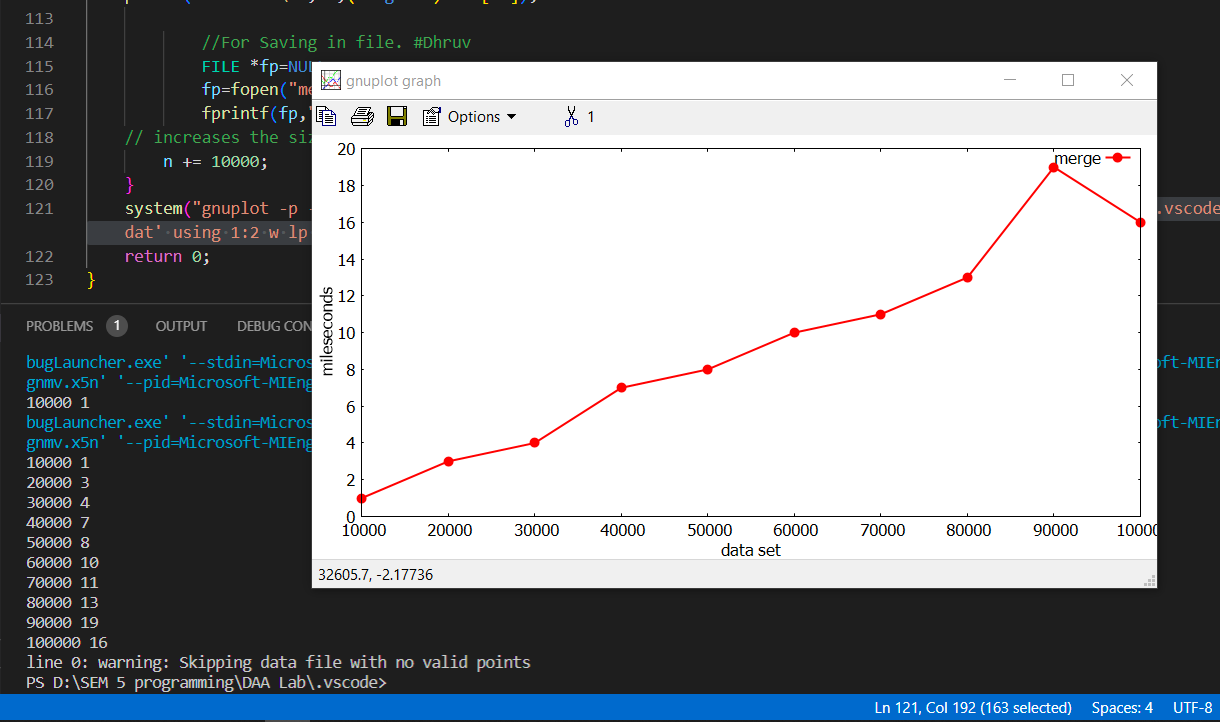
}

system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/mergecomplexity.dat' using 1:2 w lp lc rgb 'red' lw 2 pt 7 title 'merge'\"");

return 0;

}

**Output :**



**Code 2 : With User input array**

**/\* Functions are same as Code 1 \*/**

int main()

{

int i,n;

int it = 0;

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

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

{

scanf("%d",&arr[i]);

}

printf("\n Your array is : ");

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

{

printf(", %d ",arr[i]);

}

int arr\_size = sizeof(arr) / sizeof(arr[0]);

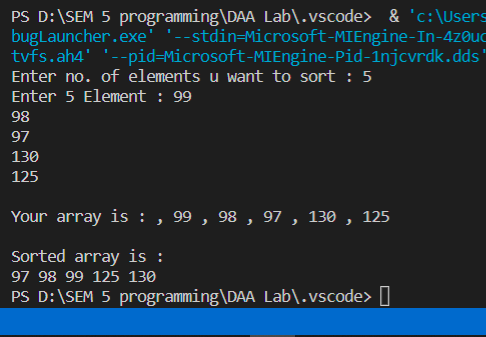
mergeSort(arr, 0, arr\_size - 1);

printf("\n\n Sorted array is : \n");

printArray(arr, arr\_size);

}

**Output :**

****

**Complexity :**

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. T(n) = 2T(n/2) + θ(n)

The solution of the above recurrence is O(nLogn). The list of size N is divided into a max of Logn parts, and the merging of all sublists into a single list takes O(N) time, the worst-case run time of this algorithm is O(nLogn)

Best Case Time Complexity: O(n\*log n)

Worst Case Time Complexity: O(n\*log n)

Average Time Complexity: O(n\*log n)

The time complexity of MergeSort is O(n\*Log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

**Practical : 4**

*Implementation and Time analysis of Quick Sort algorithms for Best case, Average case & Worst-case using Divide and Conquer.*

**Code 1 : With Random Numbers**

#include<stdio.h>

#include <stdlib.h>

#include <time.h>

void quicksort(int arr[],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(arr[i]<=arr[pivot]&&i<last)

i++;

while(arr[j]>arr[pivot])

j--;

if(i<j){

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

temp=arr[pivot];

arr[pivot]=arr[j];

arr[j]=temp;

quicksort(arr,first,j-1);

quicksort(arr,j+1,last);

}

}

int main()

{

long int n = 10000;

int it = 0;

double tim1[10];

while (it++ < 10) {

long int arr[n];

// generating n random numbers

// storing them in arrays a, b, c

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

{

long int no = rand() % n + 1;

arr[i] = no;

}

clock\_t start, end;

int arr\_size = sizeof(arr) / sizeof(arr[0]);

start = clock();

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

end = clock();

tim1[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li %li \n", n,(long int)tim1[it]);

//For Saving in file. #Dhruv

FILE \*fp=NULL;

fp=fopen("quickcomplexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

n += 10000;

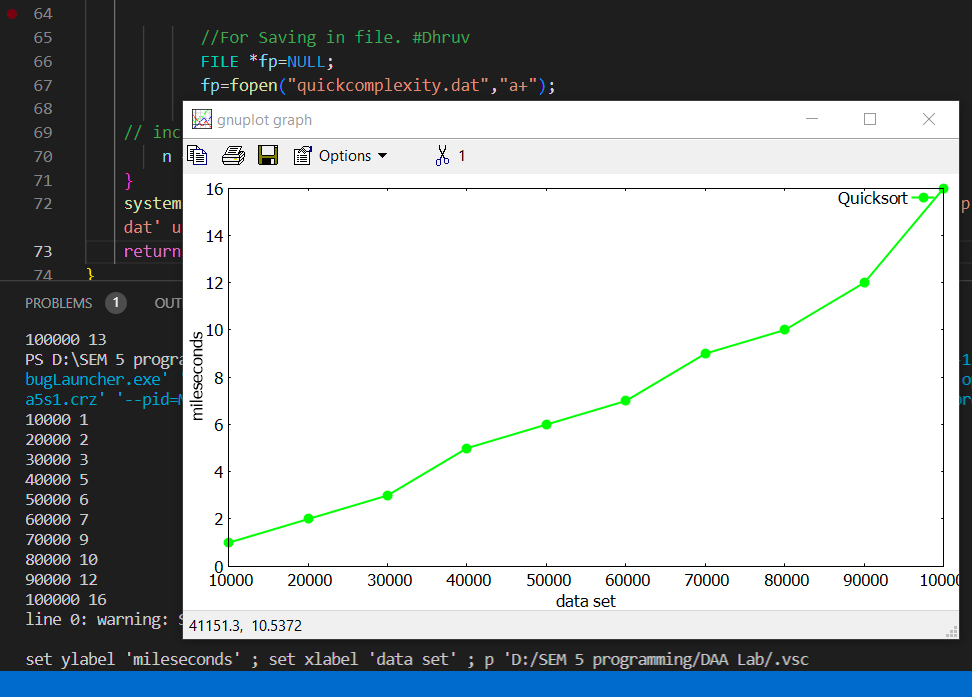
}

system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/quickcomplexity.dat' using 1:2 w lp lc rgb 'green' lw 2 pt 7 title 'Quicksort'\"");

return 0;

}

**Output :**

****

**Code 2 : With User input array**

**/\* Functions are same as Code 1 \*/**

int main()

{

int i,n;

int it = 0;

double tim1[10];

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

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

{

scanf("%d",&arr[i]);

}

printf("\nYour array is : ");

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

{

printf(", %d ",arr[i]);

}

int arr\_size = sizeof(arr) / sizeof(arr[0]);

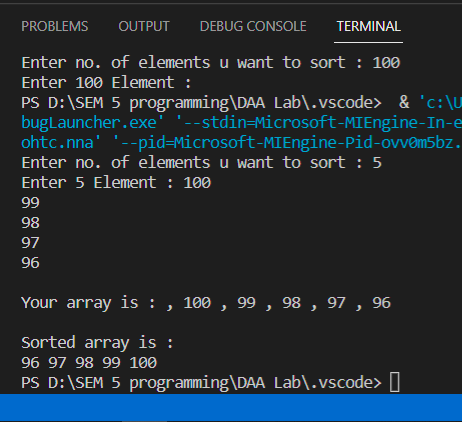
quicksort(arr,0,arr\_size-1);

printf("\n\nSorted array is : \n");

printArray(arr, arr\_size);

}

**Output :**



**Complexity :**

* **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n\*logn)**.
* **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n2)**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.