Jadavpur University

Department of Electronics and Telecommunication Engineering,
Faculty of Engineering & Technology

DSA LAB REPORT
2nd Year First Semester 2020

Name : RAHUL SAHA
Roll: 001910701009

Group 1

SORTING AND SEARCHING ALGORITHMS

Date Of Submission: 29/04/2021

Q1. Compare the performance of following five sorting algorithms on different sets of data:

Bubble Sort, Quick Sort, Selection Sort Insertion Sort, Merge Sort.
By different sets of data we mean i) completely unsorted dataset, ii) nearly sorted data etc.

Introduction

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

In order to demarcate between these algorithms on the basis of their performance, we need large amounts of testing data. We generated these datas by executing it through file handling. We created the files of various datasets like *integers_10000.txt* and saved them in the desired directory.
One of the code completely random numbers by executing it through *rand()* function and other by partially unordered dataset which is implemented by swapping a few values within the dataset.

Unsorted Number Generator

This code will generate 6 files with 10000, 15000, 25000, 50000, 75000 and 100000 data respectively. Data is completely unordered.

Source Code:

```
#include <stdio.h>
#include<time.h>
#include<stdlib.h>
int main(void) {

  FILE *fptr1,*fptr2,*fptr3,*fptr4,*fptr5,*fptr6;// creating a FILE
variable
  long int i,num;
  fptr1 = fopen("integers_10000.txt", "w");// opening the file in write
mode
  if (fptr1 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
```

```
  fptr2 = fopen("integers_15000.txt", "w");// opening the file in write
mode
  if (fptr2 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr3 = fopen("integers_25000.txt", "w");// opening the file in write
mode
  if (fptr3 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr4 = fopen("integers_50000.txt", "w");// opening the file in write
mode
  if (fptr4 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr5 = fopen("integers_75000.txt", "w");// opening the file in write
mode
  if (fptr5 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr6 = fopen("integers_100000.txt", "w");// opening the file in write
mode
  if (fptr6 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  srand(time(0));
  //10000 elements
```

```
  for(i=0;i<10000;i++){
    num=rand();
    fprintf(fptr1,"%d\n",num);
  }
  fclose(fptr1);// close connection

  //15000 elements
  for(i=0;i<15000;i++){
    num=rand();
    fprintf(fptr2,"%d\n",num);
  }
  fclose(fptr2);// close connection

  //25000 elements
  for(i=0;i<25000;i++){
    num=rand();
    fprintf(fptr3,"%d\n",num);
  }
  fclose(fptr3);// close connection


  //50000 elements
  for(i=0;i<50000;i++){
    num=rand();
    fprintf(fptr4,"%d\n",num);
  }
  fclose(fptr4);// close connection

  //75000 elements
  for(i=0;i<75000;i++){
    num=rand();
    fprintf(fptr5,"%d\n",num);
  }
  fclose(fptr5);// close connection

  //100000 elements
  for(i=0;i<100000;i++){
    num=rand();
    fprintf(fptr6,"%d\n",num);
  }
  fclose(fptr6);// close connection
  return 0;
}
```

Nearly Sorted Data Set Generator:

This code will generate 6 files with 10000, 15000, 25000, 50000, 75000 and 100000 data respectively. Data is almost ordered.

Source Code:

```
#include <stdio.h>
#include<time.h>
#include<stdlib.h>
#define max 2000000
int indices[50];
int array[max];

void indices_select(int last){
    int i;
    for (i = 0; i < 10; i++) {
      int rand_num = (rand() % (1 - last + 1)) + 1;
      indices[i]=rand_num;
   }
}

void array_swap(){
    int i,temp;
    for(i=0;i<50;i++){
       temp=array[indices[i]];
       array[indices[i]]=array[indices[i]-1];
       array[indices[i]-1]=temp;
   }
}

int main() {
  FILE *fptr1,*fptr2,*fptr3,*fptr4,*fptr5,*fptr6;// creating a FILE
variable
  long int i,num;
  fptr1 = fopen("partial_integers_10000.txt", "w");// opening the file in
write mode
  if (fptr1 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr2 = fopen("partial_integers_15000.txt", "w");// opening the file in
write mode
```

```
  if (fptr2 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr3 = fopen("partial_integers_25000.txt", "w");// opening the file in
write mode
  if (fptr3 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr4 = fopen("partial_integers_50000.txt", "w");// opening the file in
write mode
  if (fptr4 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr5 = fopen("partial_integers_75000.txt", "w");// opening the file in
write mode
  if (fptr5 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr6 = fopen("partial_integers_100000.txt", "w");// opening the file in
write mode
  if (fptr6 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }

  //10000 elements
  for(i=0;i<10000;i++){
    array[i]=i+1;
```

```
}
indices_select(250);
array_swap();
for(i=0;i<10000;i++){
  num=array[i];
  fprintf(fptr1,"%d\n",num);
}
fclose(fptr1);// close connection

//15000 elements
for(i=0;i<15000;i++){
  array[i]=i+1;
}
indices_select(250);
array_swap();
for(i=0;i<15000;i++){
  num=array[i];
  fprintf(fptr2,"%d\n",num);
}
fclose(fptr2);// close connection

//25000 elements
for(i=0;i<25000;i++){
  array[i]=i+1;
}
indices_select(250);
array_swap();
for(i=0;i<25000;i++){
  num=array[i];
  fprintf(fptr3,"%d\n",num);
}
fclose(fptr3);// close connection


//50000 elements
for(i=0;i<50000;i++){
  array[i]=i+1;
}
indices_select(250);
array_swap();
for(i=0;i<50000;i++){
  num=array[i];
  fprintf(fptr4,"%d\n",num);
}
fclose(fptr4);// close connection

//75000 elements
```

```
  for(i=0;i<75000;i++){
    array[i]=i+1;
  }
  indices_select(250);
  array_swap();
  for(i=0;i<75000;i++){
    num=array[i];
    fprintf(fptr5,"%d\n",num);
  }
  fclose(fptr5);// close connection

  //100000 elements
  for(i=0;i<100000;i++){
    array[i]=i+1;
  }
  indices_select(250);
  array_swap();
  for(i=0;i<100000;i++){
    num=array[i];
    fprintf(fptr6,"%d\n",num);
  }
  fclose(fptr6);// close connection
  return 0;
}
```

1. Bubble Sort

Description

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Algorithm

1. Start from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element. After doing this, the largest element is present at the end.
   Repeat the above steps but process array elements [0, n-2] because the last one, i.e., a[n-1], is present at its correct position. After this step, the largest two elements are present at the end.
5. Repeat this process n-1 times.

Time Complexities

Average Case	$O(n^2)$
Best Case	O(n)
Worst Case	$O(n^2)$

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 200000
int array[MAX];

//sorts the data
void bubblesort(int array[],long int size){
  for (long int j = 0; j < size - 1; ++j) {//access each array element
    int swapped = 0;
    for (long int i = 0; i < size - j - 1; ++i) {//compare array elements
```

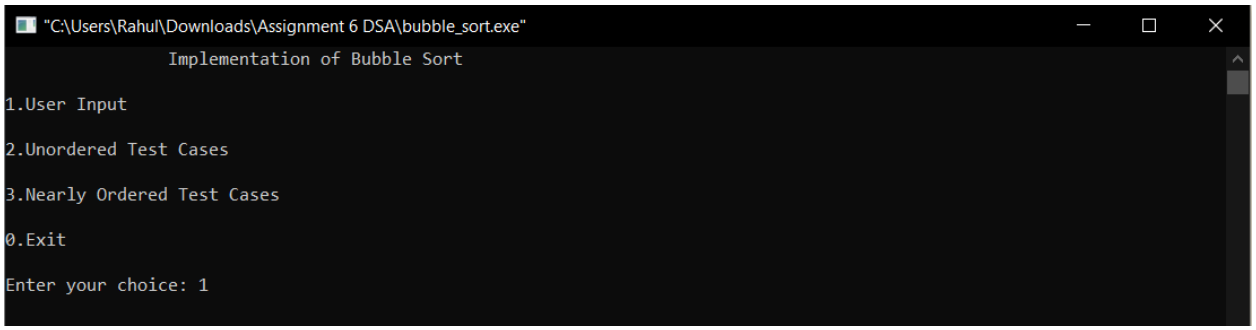
```
      if (array[i] > array[i + 1]) {//compare two adjacent elements
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;//elements getting swapped
        swapped = 1;
      }
    }
  }
}
//prints the data
void print_array(int array[],long int size) {
  for (long int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

//user input
void user_data(){
  system("cls");
  clock_t t;
  int i,value,num;
  printf("\t\tSorting User Inputed Data");
  printf("\n\nEnter the number of data you want to enter:");
  scanf("%d",&num);
  printf("\n\nEnter the datas: \n");
  for(i=0;i<num;i++){
      scanf("%d",&array[i]);
  }
  printf("UnSorted Array:\n");
  print_array(array, num);
  t = clock();
  bubblesort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("Sorted Array in Ascending Order:\n");
  print_array(array, num);
  printf("\nExecution time taken: %e  \n", time_taken);
  getch();
  return ;
}

//unordered test case
void unordered_data(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined unordered Dataset");
```

```
  FILE *fptr,*bin_ptr;
  int i=0,value;
  long int num=0;
  if ((fptr = fopen("integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  bubblesort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

//partially ordered test case
void partially_ordered(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined nearly sorted Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("partial_integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
```

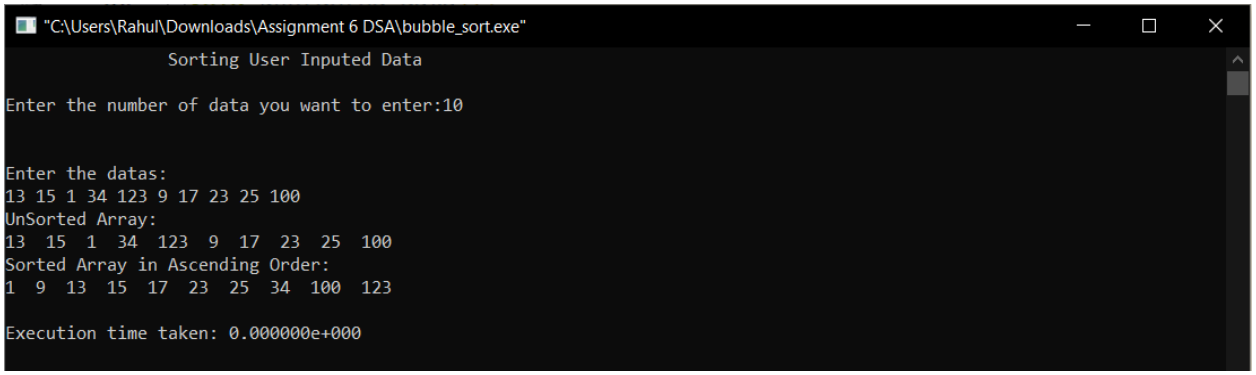
```
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  bubblesort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test_2.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}
//driver function
int main(){
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Bubble Sort");
      printf("\n\n1.User Input ");
      printf("\n\n2.Unordered Test Cases");
      printf("\n\n3.Nearly Ordered Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: unordered_data();
                break;
            case 3: partially_ordered();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

User Drive

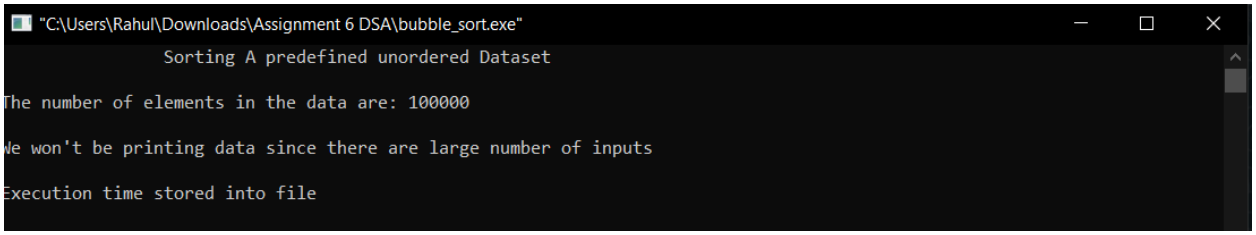


Sorting using small set of inputs

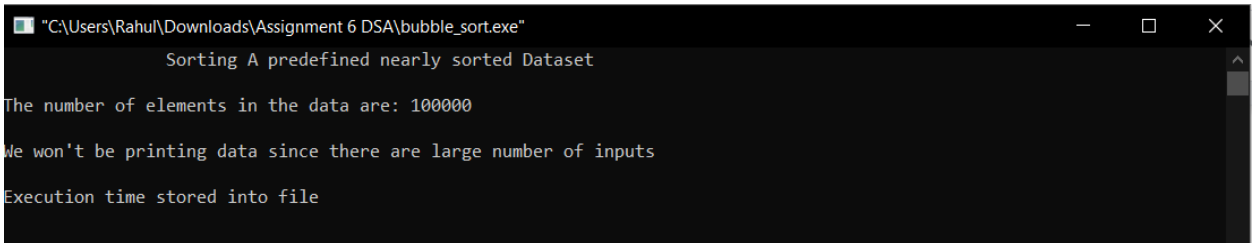


As visible, the number of inputs provided is 10. So execution time is not at all prominent as the operation is very fast due to the small number of inputs. To make it prominent we will use large datasets that we previously obtained from the file handling codes.

Sorting a completely unordered dataset



Sorting a partially ordered dataset



Final execution times for all dataset

*save_test - Notepad

File Edit Format View Help

0.235000	10000
0.554000	15000
1.682000	25000
6.828000	50000
15.659000	75000
27.494000	100000

Unsorted

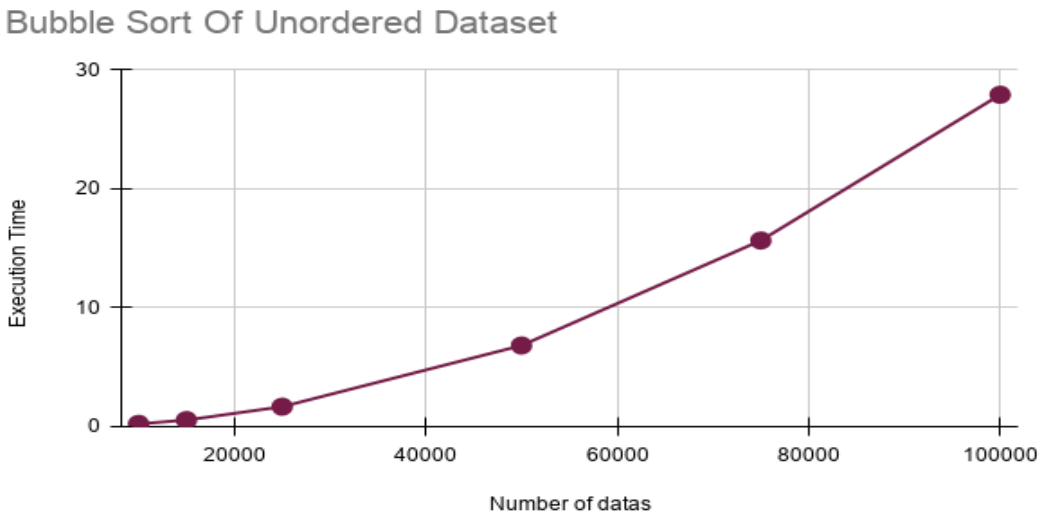
*save_test_2 - Notepad

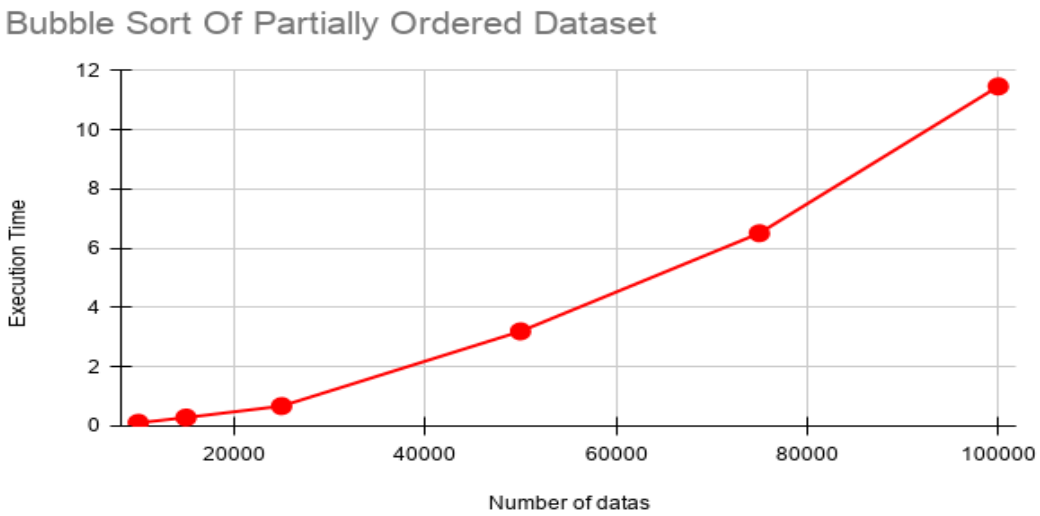
File Edit Format View Help

0.110000	10000
0.288000	15000
0.675000	25000
3.198000	50000
6.512000	75000
11.470000	100000

Nearly Sorted

Graphs according to the datas obtained





2. Selection Sort

Description

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Algorithm

```
1.Set min_idx to index 0 of the array.
2.Search the minimum element in the list.
3.Swap with value at index 0.
4.Increment min_idx to point to the next element.
5.Repeat until the list is sorted.
```

Time Complexities:

Average Case	$O(n^2)$
Best Case	$O(n^2)$
Worst Case	$O(n^2)$

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 100000

int array[MAX];

//swap two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

//selection sort
```

```
void selectionsort(int array[], long int size) {
  for (long int j = 0; j < size - 1; j++) {
    int min_idx = j;
    for (long int i = j + 1; i < size; i++) {
      if (array[i] < array[min_idx])//Selecting the minimum element in each
loop.
        min_idx = i;
    }
    swap(&array[min_idx], &array[j]);// put min at the correct position
  }
}

// function to print an array
void print_array(int array[], long int size) {
  for (long int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
//user input
void user_data(){
  system("cls");
  clock_t t;
  int i,value,num;
  printf("\t\tSorting User Inputed Data");
  printf("\n\nEnter the number of data you want to enter:");
  scanf("%d",&num);
  printf("\n\nEnter the datas: \n");
  for(i=0;i<num;i++){
      scanf("%d",&array[i]);
  }
  printf("UnSorted Array:\n");
  print_array(array, num);
  t = clock();
  selectionsort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("Sorted Array in Ascending Order:\n");
  print_array(array, num);
  printf("\nExecution time taken: %e  \n", time_taken);
  getch();
  return ;
}

//unordered test case
void unordered_data(){
  system("cls");
```

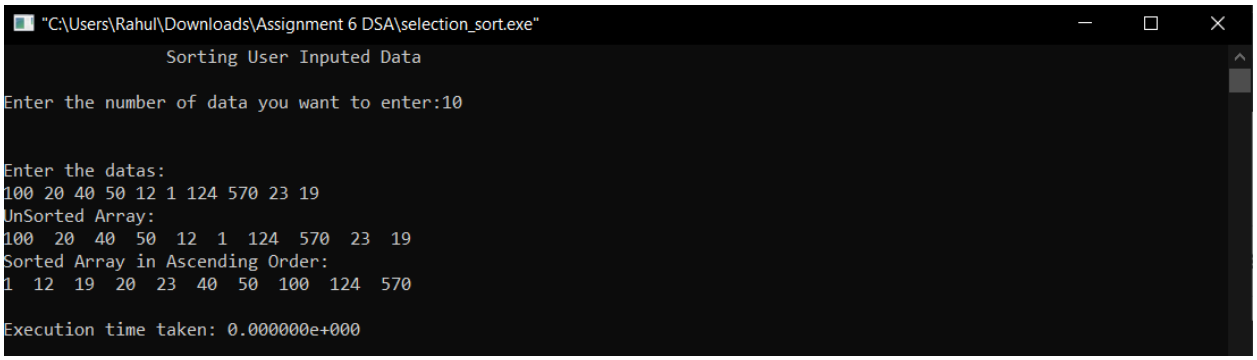
```
  clock_t t;
  printf("\t\tSorting A predefined unordered Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
   while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  selectionsort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

//partially ordered test case
void partially_ordered(){
  system("cls");
  clock_t t;
   printf("\t\tSorting A predefined nearly sorted Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("partial_integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
   while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
```

```
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  selectionsort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test_2.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

int main(){
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Selection Sort");
      printf("\n\n1.User Input ");
      printf("\n\n2.Unordered Test Cases");
      printf("\n\n3.Nearly Ordered Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: unordered_data();
                break;
            case 3: partially_ordered();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

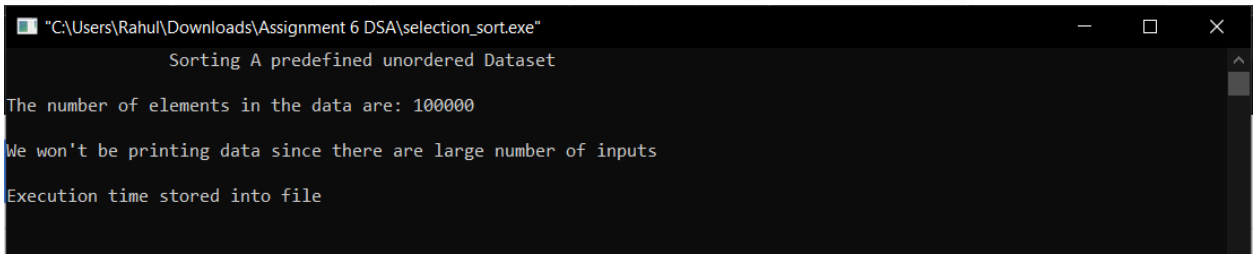
User Drive

Sorting using small set of inputs

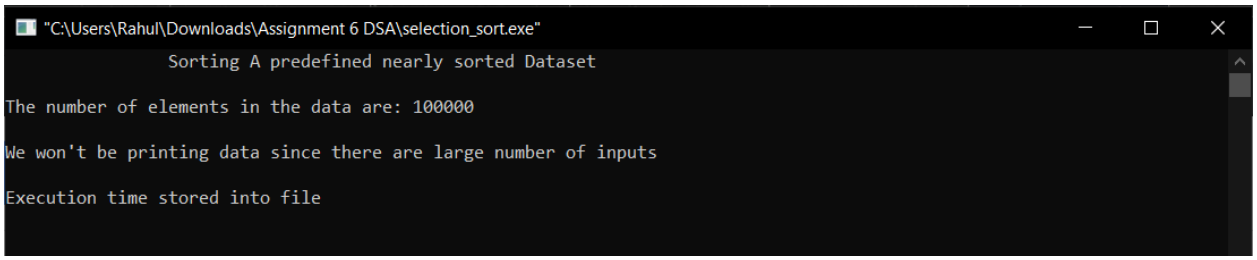


As visible, the number of inputs provided is 10. So execution time is not at all prominent as the operation is very fast due to the small number of inputs. To make it prominent we will use large datasets that we previously obtained from the file handling codes.

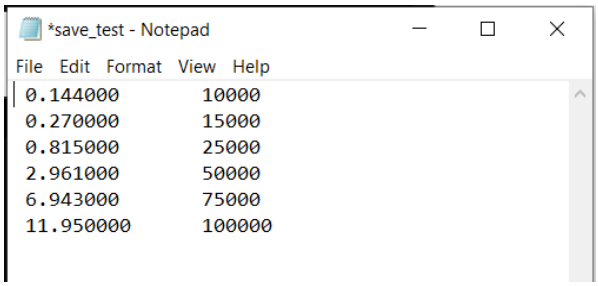
Sorting a completely unordered set



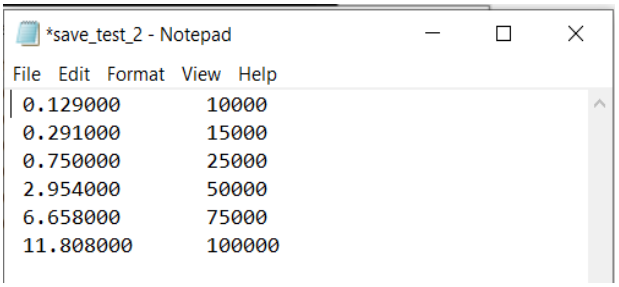
Sorting a partially ordered set



Final execution time for all datasets

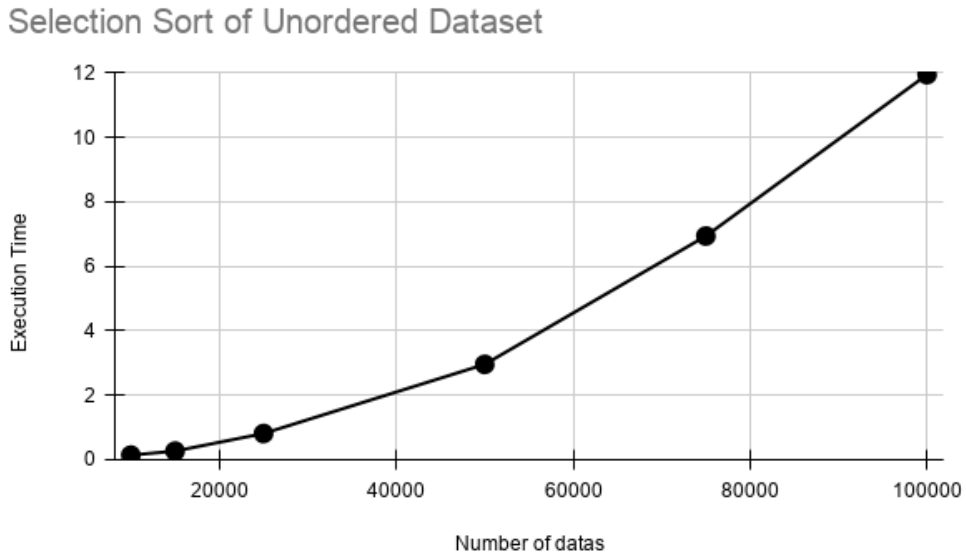


Unordered



Partially Ordered

Graphs according to the datas obtained



3. Insertion Sort

Description:-

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Algorithm:

```
1.Iterate from arr[1] to arr[n] over the array.
2.Compare the current element (key) to its predecessor.
3.If the key element is smaller than its predecessor, compare it to the
elements before. Move the greater elements one position up to make space
for the swapped element.
```

Time Complexities:

Average Case	$O(n^2)$
Best Case	O(n)
Worst Case	$O(n^2)$

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 200000
int array[MAX];

//insertion sort
void insertionsort(int array[], long int size) {
  for (long int j = 1; j < size; j++) {
    int key = array[j];//Selecting the key
    int p = j - 1;
    while (key < array[p] && p >= 0) {//comparing the key with elements on
left
      array[p + 1] = array[p];
      --p;
```

```
    }
    array[p + 1] = key;//changing key when a smaller element  is found
  }
}

//prints the array
void print_array(int array[], long int size) {
  for (long int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

//user input
void user_data(){
  system("cls");
  clock_t t;
  int i,value,num;
  printf("\t\tSorting User Inputed Data");
  printf("\n\nEnter the number of data you want to enter:");
  scanf("%d",&num);
  printf("\n\nEnter the datas: \n");
  for(i=0;i<num;i++){
      scanf("%d",&array[i]);
  }
  printf("UnSorted Array:\n");
  print_array(array, num);
  t = clock();
  insertionsort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("Sorted Array in Ascending Order:\n");
  print_array(array, num);
  printf("\nExecution time taken: %e  \n", time_taken);
  getch();
  return ;
}

//unordered test case
void unordered_data(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined unordered Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("integers_100000.txt","r")) == NULL){
```

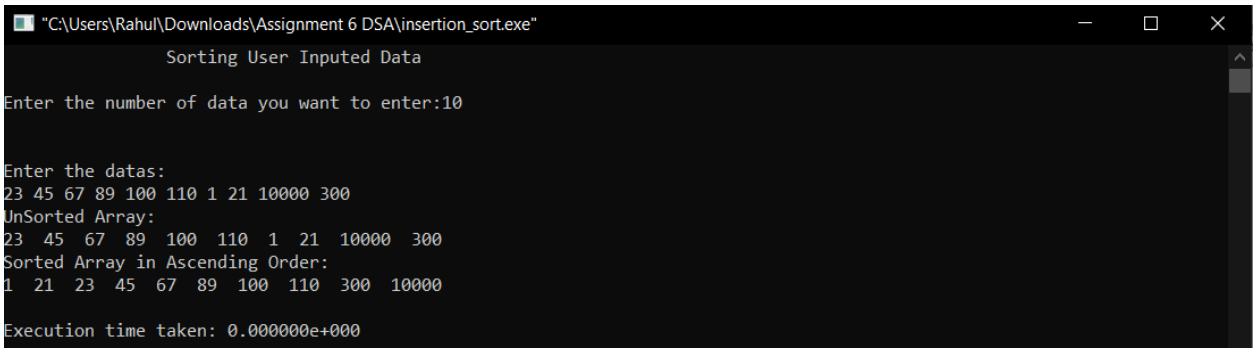
```
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  insertionsort(array,num);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

//partially ordered test case
void partially_ordered(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined nearly sorted Dataset");
  FILE *fptr,*bin_ptr;
  int value,temp;
  long int num=0,i=0;
  if ((fptr = fopen("partial_integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  insertionsort(array,num);
```

```
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test_2.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}
//driver function
int main(){
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Insertion Sort");
      printf("\n\n1.User Input ");
      printf("\n\n2.Unordered Test Cases");
      printf("\n\n3.Nearly Ordered Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: unordered_data();
                break;
            case 3: partially_ordered();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

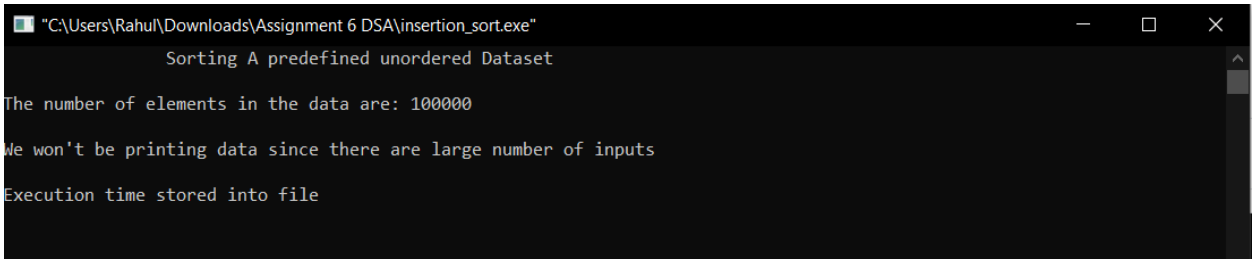
User drive

Sorting small set of inputs

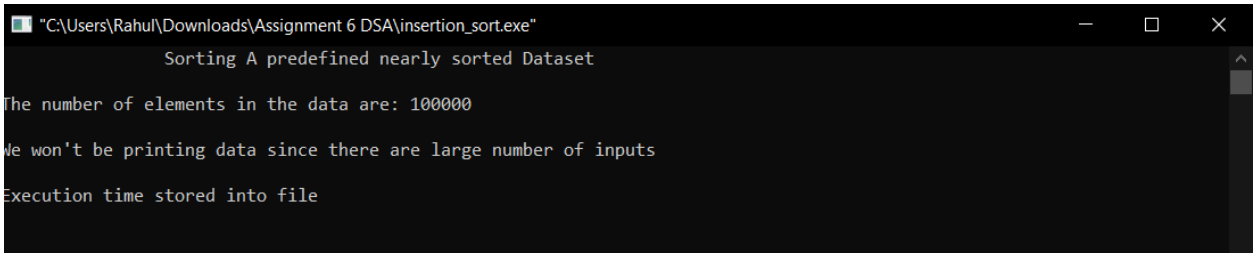


As visible, the number of inputs provided is 10. So execution time is not at all prominent as the operation is very fast due to the small number of inputs. To make it prominent we will use large datasets that we previously obtained from the file handling codes.

Sorting an unordered dataset



Sorting a partially ordered dataset



Final execution time of all the datasets

*save_test - Notepad

File Edit Format View Help

0.084000	10000
0.141000	15000
0.390000	25000
1.619000	50000
3.623000	75000
6.365000	100000

Unordered

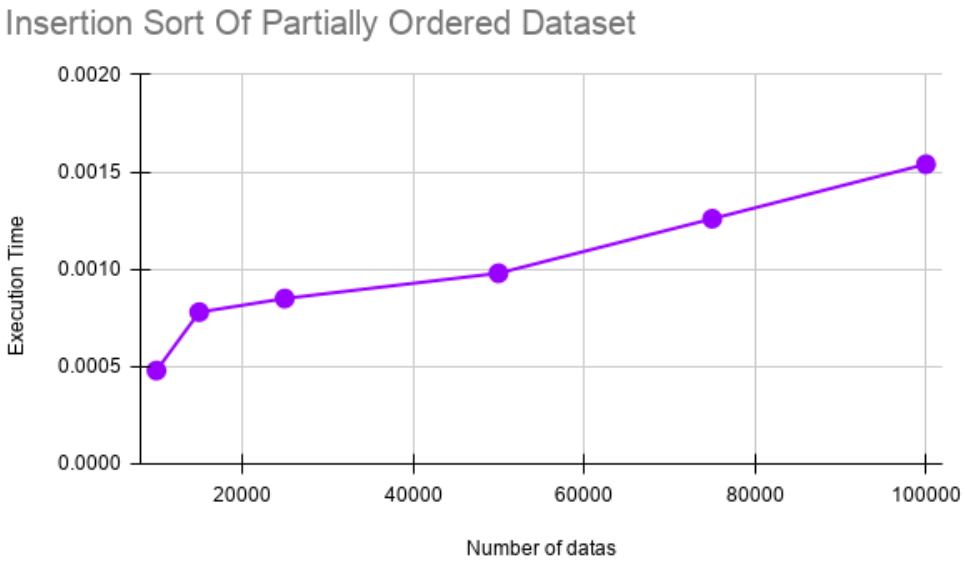
*save_test_2 - Notepad

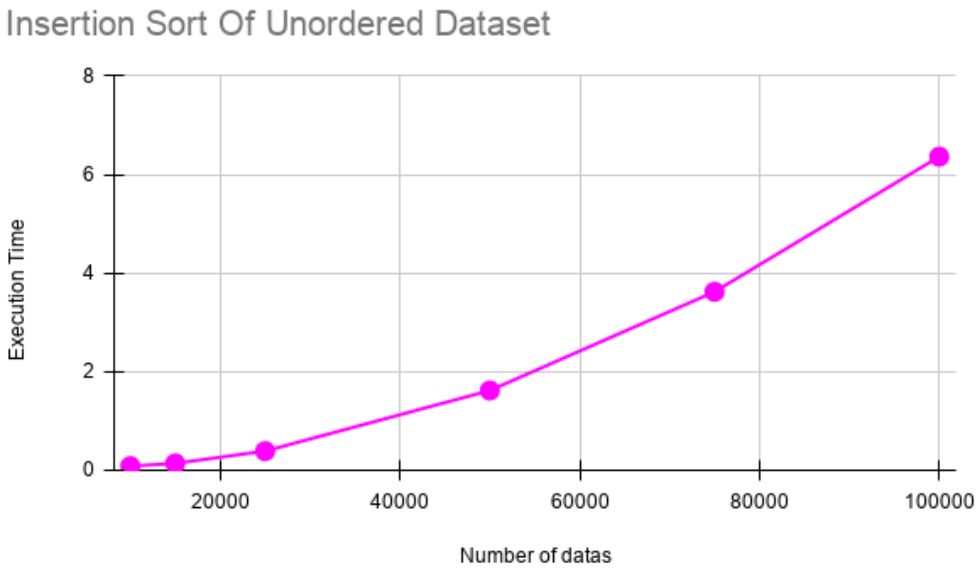
File Edit Format View Help

0.00048	10000
0.00078	15000
0.00085	25000
0.00098	50000
0.00126	75000
0.00154	100000

Partially Ordered

Graphs according to the dataset





4. Merge Sort

Description:

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several subarrays until each subarray consists of a single element and merging those subarrays in a manner that results into a sorted list.

Algorithm:

```
1.Divide the unsorted array into N subarrays , each containing 1 element.
2.Take adjacent pairs of two singleton arrays and merge them to form an
array of 2 elements. N will now convert into N/2 lists of size 2.
3.Repeat the process till a single sorted list of obtained..
```

Time Complexity:

Average Case	O(n*log n)
Best Case	O(n*log n)
Worst Case	O(n*log n)

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 100000

int array[MAX];

//merging the subarrays
void merge(int arr[], long int l, long int m,long  int r) {
  long int n1 = m - l + 1;
  long int n2 = r - m;
  long int i,j,k;
  int L[n1], M[n2];
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
```

```
    M[j] = arr[m + 1 + j];

  i=0;
  j=0;
  k = l;

  while (i < n1 && j < n2) {//placing elements in L and M in correct
position at arr[p,,r]
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }
  while (i < n1) {//If elements in L or M run out we pick up the remaining
elements and put in arr[p..r]
    arr[k] = L[i];
    i++;
    k++;
  }
  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// mergesort
void mergesort(int arr[], long int left, long int right) {
  if (left < right) {
    long int middle = left + (right - left) / 2;//dividing in to two
subarrays by position m
    mergesort(arr, left, middle);
    mergesort(arr, middle + 1, right);
    merge(arr, left, middle, right);//merging the subarrays
  }
}

// function to print an array
void print_array(int array[], long int size) {
  for (long int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
```

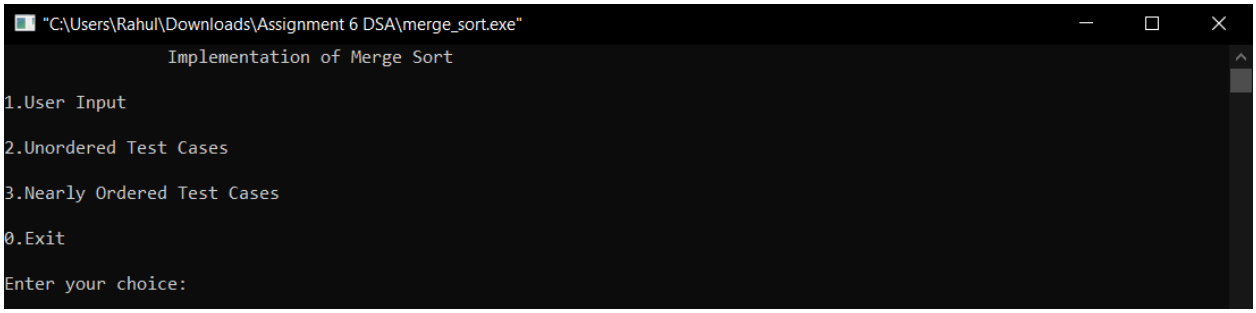
```
}

//user input
void user_data(){
  system("cls");
  clock_t t;
  int i,value,num;
  printf("\t\tSorting User Inputed Data");
  printf("\n\nEnter the number of data you want to enter:");
  scanf("%d",&num);
  printf("\n\nEnter the datas: \n");
  for(i=0;i<num;i++){
      scanf("%d",&array[i]);
  }
  printf("UnSorted Array:\n");
  print_array(array, num);
  t = clock();
  mergesort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("Sorted Array in Ascending Order:\n");
  print_array(array, num);
  printf("\nExecution time taken: %e  \n", time_taken);
  system("pause");
  return ;
}

//unordered test case
void unordered_data(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined unordered Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %ld",num);
  printf("\n\nWe won't be printing data since there are large number of
```

```
inputs\n\n");
  t = clock();
  mergesort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("\nExecution time taken: %e  \n", time_taken);
  bin_ptr = fopen("save_test.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

//partially ordered test case
void partially_ordered(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined nearly sorted Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("partial_integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %ld",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  mergesort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("\nExecution time taken: %e  \n", time_taken);
  bin_ptr = fopen("save_test_2.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}
```

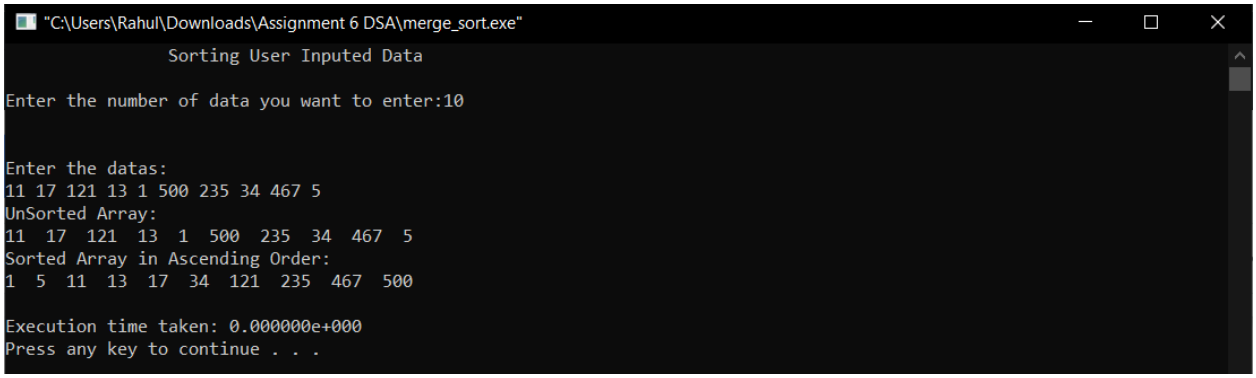
```
//driver function
int main(){
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Merge Sort");
      printf("\n\n1.User Input ");
      printf("\n\n2.Unordered Test Cases");
      printf("\n\n3.Nearly Ordered Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: unordered_data();
                break;
            case 3: partially_ordered();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

Userdrive

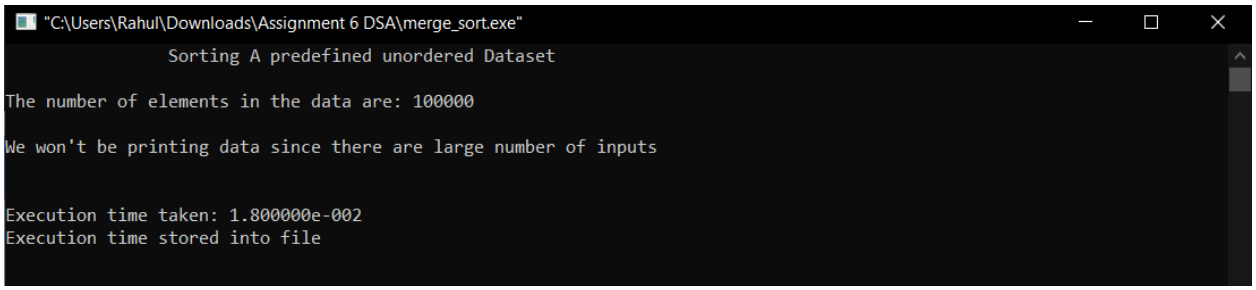


Sorting a small set of data

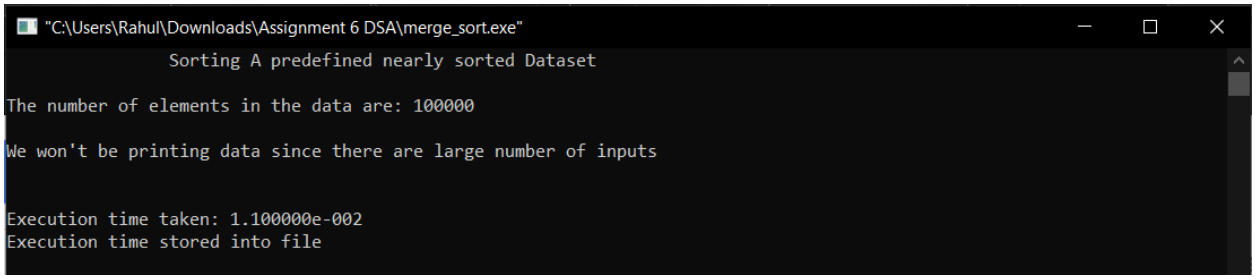


As visible, the number of inputs provided is 10. So execution time is not at all prominent as the operation is very fast due to the small number of inputs. To make it prominent we will use large datasets that we previously obtained from the file handling codes.

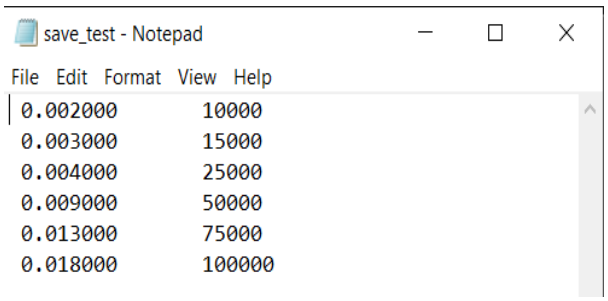
Sorting a completely unordered dataset



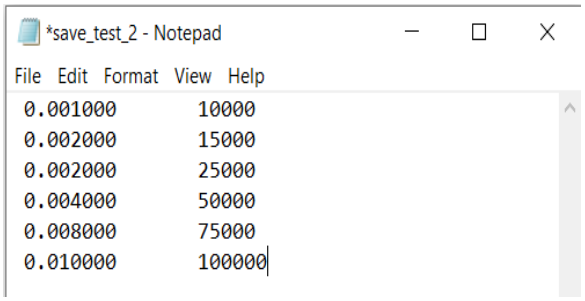
Sorting partially ordered dataset



Execution time for all datasets

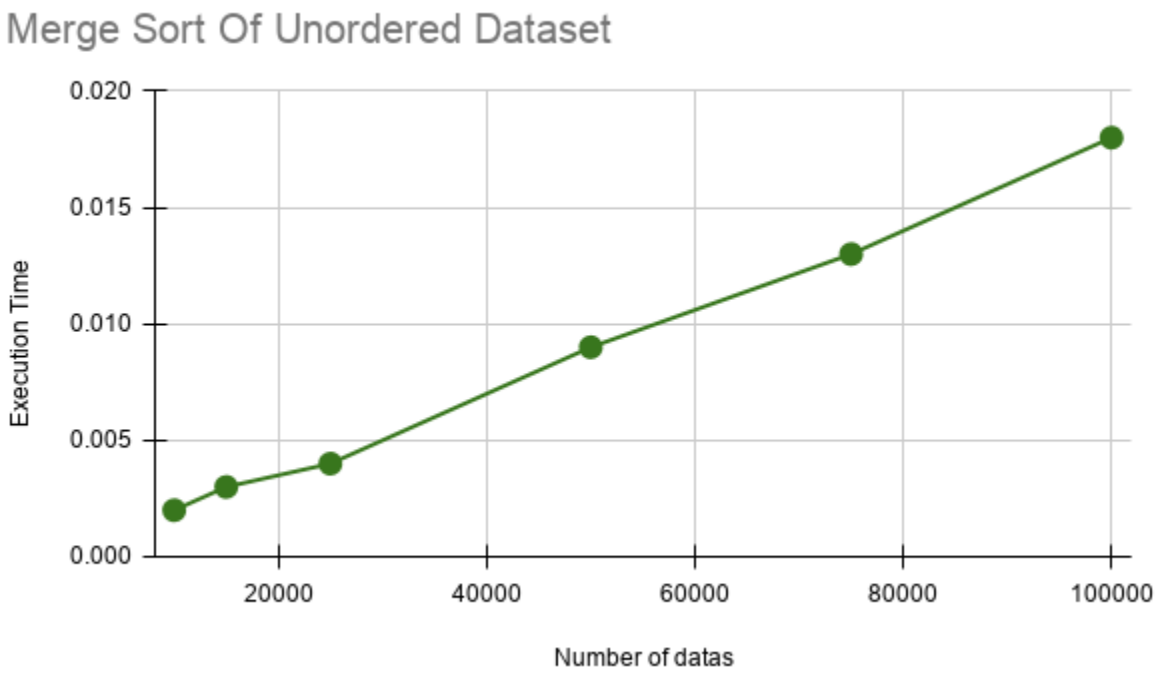


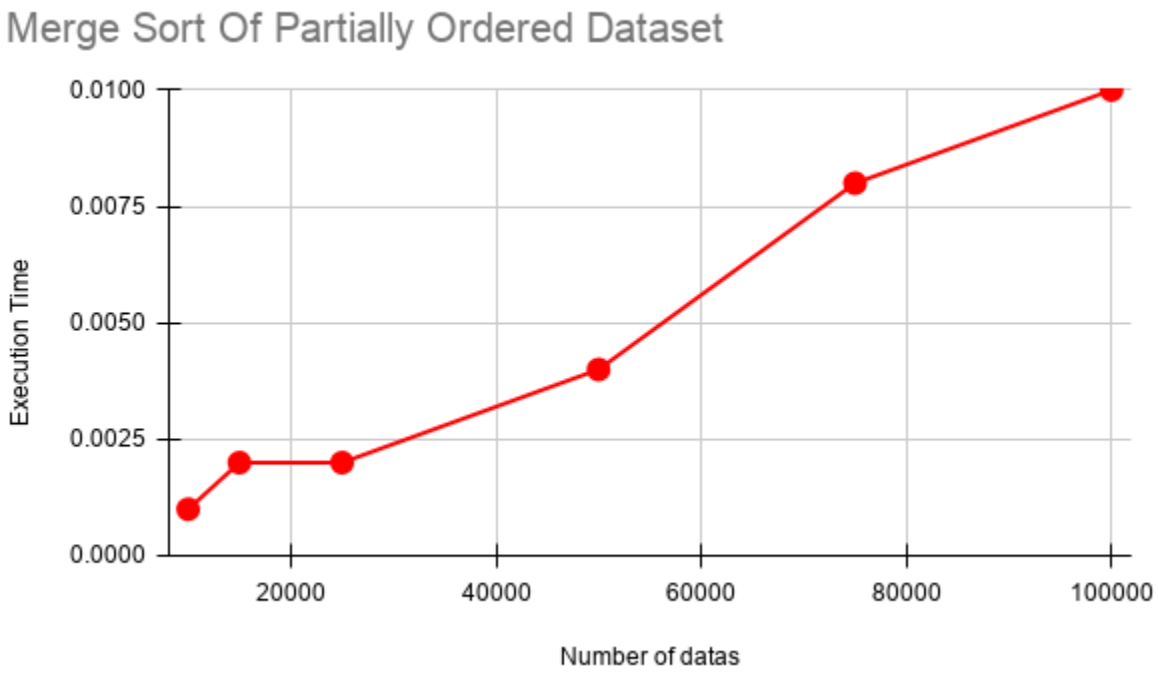
Unordered



Partially Ordered

Graphs for the obtained datasets





Quick Sort

Description

Quick Sort is based on the Divide and Conquer algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Algorithm:

```
1.Choose the highest index value as pivot.
2.Partition the array based on the pivot element. Elements that are smaller
than the pivot are put on the left and the elements greater than the pivot
are put on the right.
3.Apply a quick sort on the left partition recursively.
4.Apply a quick sort on the right partition recursively.
5.The subarrays are divided until each subarray is formed of a single
element. At this point, the array is already sorted.
```

Time Complexity:

Average Case	O(n*log n)
Best Case	O(n*log n)
Worst Case	$O(n^2)$

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 100000
int array[MAX];

void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
```

```
// function to find the partition position
//The element is such that all the elements to the left of pivot are
smaller than it
//All the elements to the right of pivot are greater than it
int partition(int array[], long int low, long int high) {
  long int pivot = array[high];//pivot is the rightmost element
  long int i = (low - 1);//greater element pointer
  for (long int j = low; j < high; j++) {
    if (array[j] <= pivot) {//comparing with pivot
      i++;//greater element pointer shifts
      swap(&array[i], &array[j]);//swaps with the greater element
    }
  }
  swap(&array[i + 1], &array[high]);
  return (i + 1);//partition point
}

//quick sort
void quicksort(int array[], long int lower, long int higher) {
  if (lower < higher) {

    long int pivot = partition(array, lower, higher);
    quicksort(array, lower, pivot - 1);//selecting pivot for the left part
    quicksort(array, pivot + 1, higher);//selecting pivot for the right
part
  }
}

// function to print an array
void print_array(int array[], long int size) {
  for (long int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

//user input
void user_data(){
  system("cls");
  clock_t t;
  int i,value,num;
  printf("\t\tSorting User Inputed Data");
  printf("\n\nEnter the number of data you want to enter:");
  scanf("%d",&num);
  printf("\n\nEnter the datas: \n");
  for(i=0;i<num;i++){
      scanf("%d",&array[i]);
```

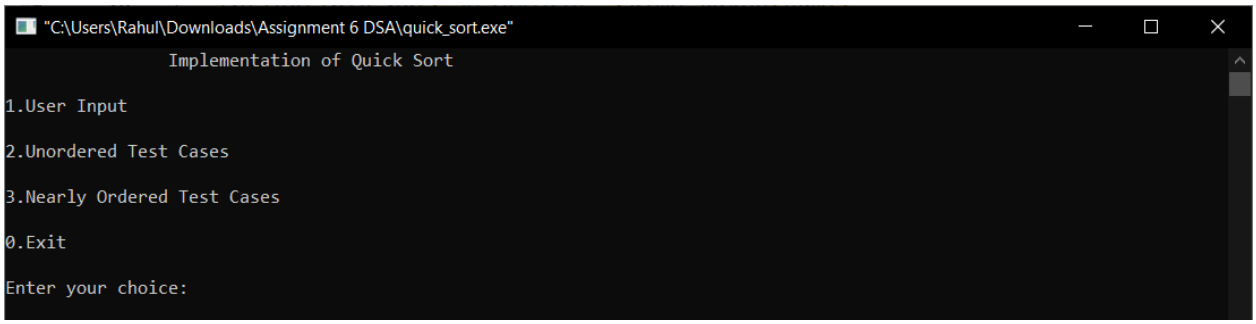
```
  }
  printf("UnSorted Array:\n");
  print_array(array, num);
  t = clock();
  quicksort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  printf("Sorted Array in Ascending Order:\n");
  print_array(array, num);
  printf("\nExecution time taken: %e  \n", time_taken);
  getch();
  return ;
}

//unordered test case
void unordered_data(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined unordered Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("integers_100000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  quicksort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds;
  bin_ptr = fopen("save_test.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}
```

```
//partially ordered test case
void partially_ordered(){
  system("cls");
  clock_t t;
  printf("\t\tSorting A predefined nearly sorted Dataset");
  FILE *fptr,*bin_ptr;
  int i=0,value,temp;
  long int num=0;
  if ((fptr = fopen("partial_integers_50000.txt","r")) == NULL){
    printf("Error! opening file");
    exit(1);
  }
  while(fscanf(fptr,"%d\n",&value)==1){
    array[i]=value;
    i++;
    num++;
  }
  fclose(fptr);
  printf("\n\nThe number of elements in the data are: %d",num);
  printf("\n\nWe won't be printing data since there are large number of
inputs\n\n");
  t = clock();
  quicksort(array,0,num-1);
  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
  bin_ptr = fopen("save_test_2.txt", "a+");
  fprintf(bin_ptr,"%f\t%ld\n ",time_taken,num);
  printf("Execution time stored into file");
  fclose(bin_ptr);
  getch();
  return;
}

//driver function
int main(){
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Quick Sort");
      printf("\n\n1.User Input ");
      printf("\n\n2.Unordered Test Cases");
      printf("\n\n3.Nearly Ordered Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
```

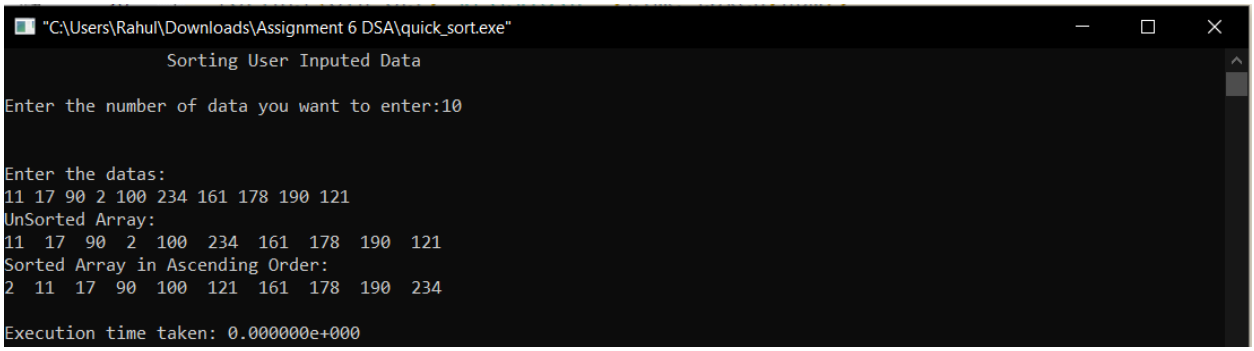
```
                break;
            case 2: unordered_data();
                break;
            case 3: partially_ordered();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

Userdrive

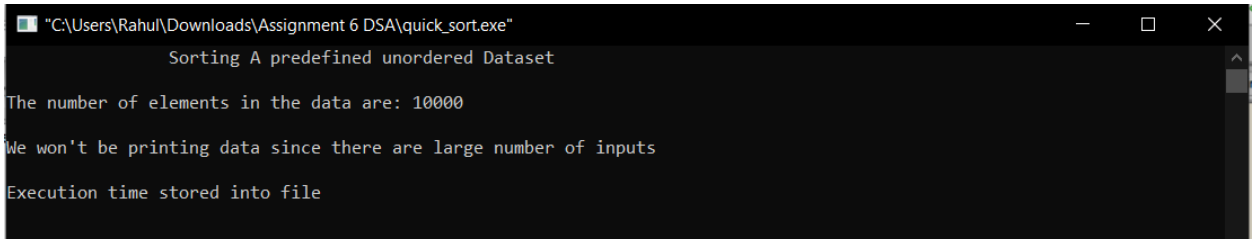


Sorting a small set of integers

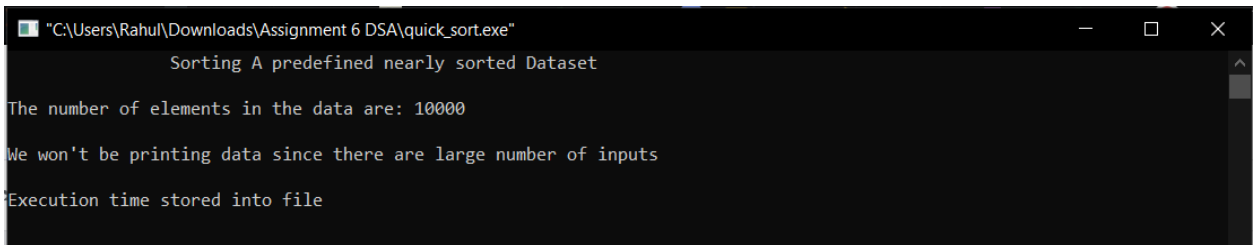


As visible, the number of inputs provided is 10. So execution time is not at all prominent as the operation is very fast due to the small number of inputs. To make it prominent we will use large datasets that we previously obtained from the file handling codes.

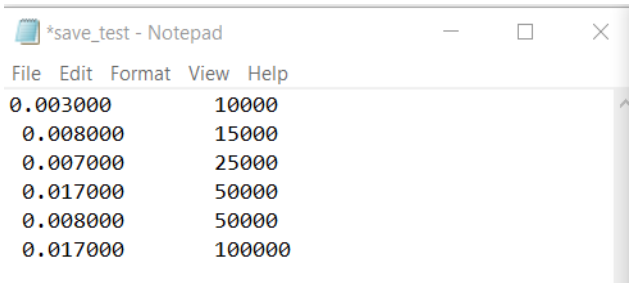
Sorting an unordered dataset



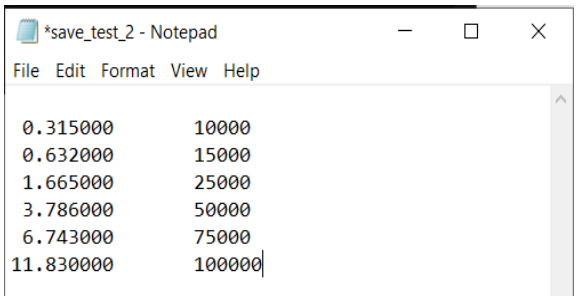
Sorting partially ordered dataset



Execution time for all the datasets

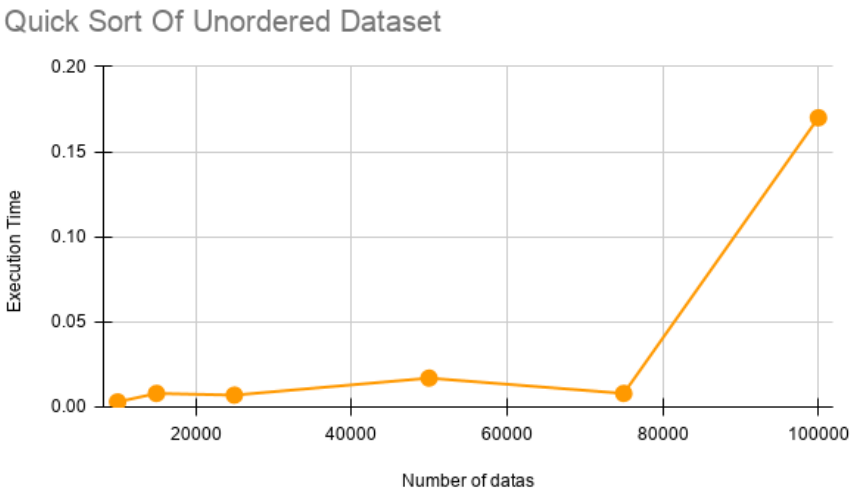


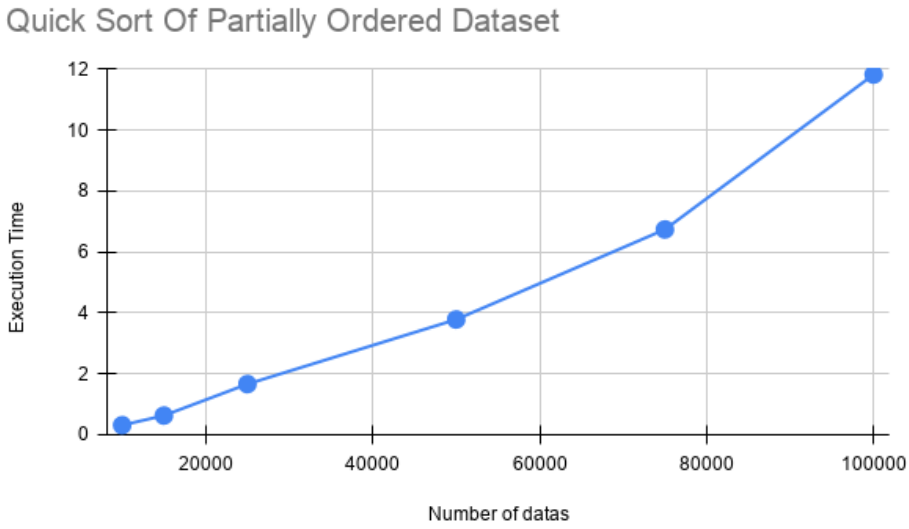
Unordered



Partially Ordered

Graphs from the obtained datas

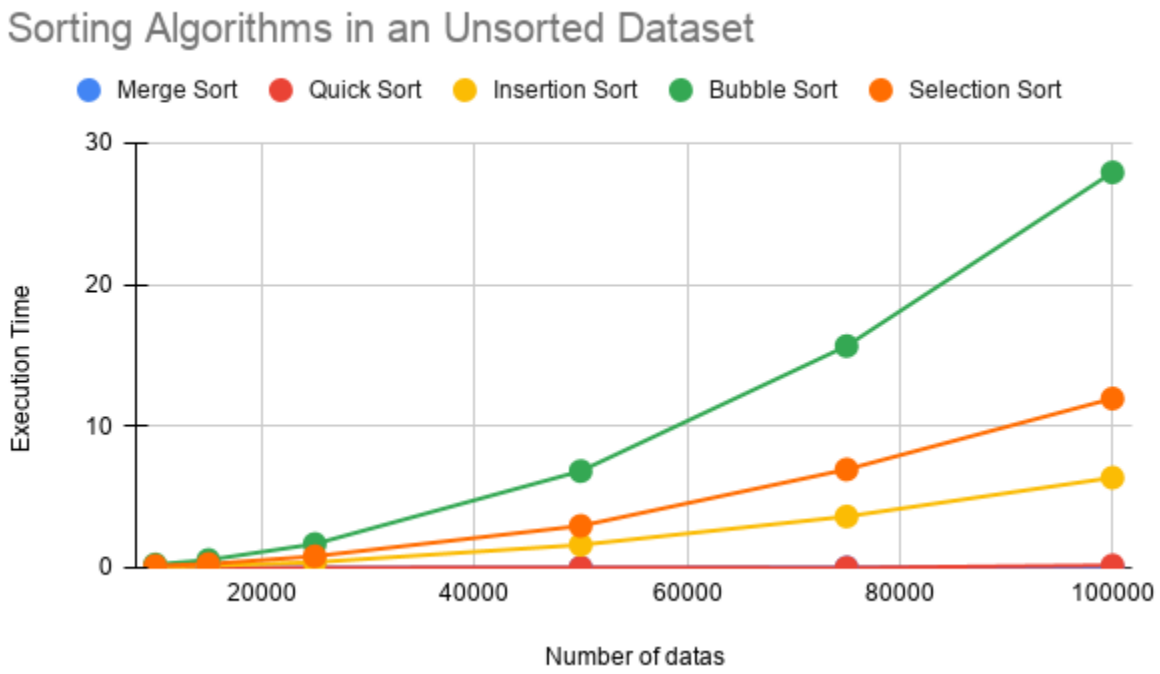




Comparison Of All Algorithms

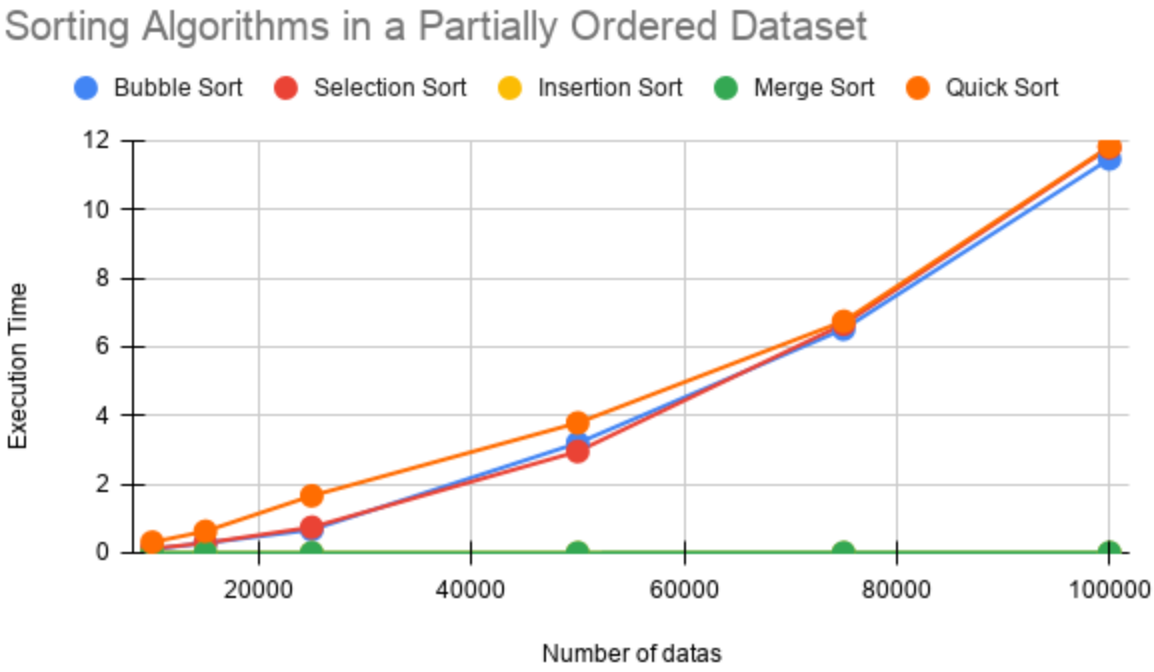
1. Unsorted Dataset

Unordered Data Set					
Number of datas	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
10000	0.235	0.144	0.084	0.002	0.003
15000	0.554	0.27	0.141	0.003	0.008
25000	1.682	0.815	0.39	0.004	0.007
50000	6.828	2.961	1.619	0.009	0.017
75000	15.659	6.943	3.623	0.013	0.008
100000	27.921	11.95	6.365	0.018	0.17



2. Partially Sorted Dataset

Partially Ordered Dataset					
Number of datas	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
10000	0.11	0.129	0.00048	0.001	0.315
15000	0.288	0.291	0.00078	0.002	0.632
25000	0.675	0.75	0.00085	0.002	1.665
50000	3.198	2.954	0.00098	0.004	3.786
75000	6.512	6.658	0.00126	0.008	6.743
100000	11.47	11.8	0.00154	0.01	11.83


Sorting Algorithms in a Partially Ordered Dataset
Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
Quick Sort
Execution Time
0
2
4
6
8
10
12
20000
40000
60000
80000
100000
Number of datas

Q2. Compare the performance of linear and binary search

Introduction

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Searching may be sequential or not. If the data in the dataset are random, then we need to use sequential searching. Otherwise we can use other different techniques to reduce the complexity.

In order to get prominent execution time and demarcate between the two algorithms, we have used a large number of datasets and executed it through file handling.

Number Generator

This code will generate 11 files with 10000, 20000, 30000, 400000, 50000, 100000, 200000, 350000, 500000,750000 and 1000000 data respectively.

Source Code:

```
#include <stdio.h>
#include<time.h>
#include<stdlib.h>
int main() {

  FILE
*fptr1,*fptr2,*fptr3,*fptr4,*fptr5,*fptr6,*fptr7,*fptr8,*fptr9,*fptr10;//
creating a FILE variable
  long int i,num;
  fptr1 = fopen("high_integers_10000.txt", "w");// opening the file in
write mode
  if (fptr1 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr2 = fopen("high_integers_20000.txt", "w");// opening the file in
write mode
  if (fptr2 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
```

```
  }
  fptr3 = fopen("high_integers_30000.txt", "w");// opening the file in
write mode
  if (fptr3 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr4 = fopen("high_integers_40000.txt", "w");// opening the file in
write mode
  if (fptr4 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr5 = fopen("high_integers_50000.txt", "w");// opening the file in
write mode
  if (fptr5 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr6 = fopen("high_integers_100000.txt", "w");// opening the file in
write mode
  if (fptr6 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr7 = fopen("high_integers_200000.txt", "w");// opening the file in
write mode
  if (fptr7 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr8 = fopen("high_integers_350000.txt", "w");// opening the file in
```

```
write mode
  if (fptr8 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr9 = fopen("high_integers_500000.txt", "w");// opening the file in
write mode
  if (fptr9 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  fptr10 = fopen("high_integers_1000000.txt", "w");// opening the file in
write mode
  if (fptr10 != NULL) {
    printf("File created successfully!\n");
  }
  else {
    printf("Failed to create the file.\n");
    return -1;
  }
  srand(time(0));
  for(i=0;i<10000;i++){
    num=i+1;
    fprintf(fptr1,"%ld ",num);
  }
  fclose(fptr1);// close connection

  for(i=0;i<20000;i++){
    num=i+1;
    fprintf(fptr2,"%ld ",num);
  }
  fclose(fptr2);// close connection

  for(i=0;i<30000;i++){
    num=i+1;
    fprintf(fptr3,"%ld ",num);
  }
  fclose(fptr3);// close connection


  for(i=0;i<40000;i++){
```

```
    num=i+1;
    fprintf(fptr4,"%ld ",num);
  }
  fclose(fptr4);// close connection

  for(i=0;i<50000;i++){
    num=i+1;
    fprintf(fptr5,"%ld ",num);
  }
  fclose(fptr5);// close connection

  for(i=0;i<100000;i++){
    num=i+1;
    fprintf(fptr6,"%ld ",num);
  }
  fclose(fptr6);// close connection

  for(i=0;i<200000;i++){
    num=i+1;
    fprintf(fptr7,"%ld ",num);
  }
  fclose(fptr7);// close connection

  for(i=0;i<350000;i++){
    num=i+1;
    fprintf(fptr8,"%ld ",num);
  }
  fclose(fptr8);// close connection

  for(i=0;i<500000;i++){
    num=i+1;
    fprintf(fptr9,"%ld ",num);
  }
  fclose(fptr9);// close connection


  for(i=0;i<1000000;i++){
    num=i+1;
    fprintf(fptr10,"%ld ",num);
  }
  fclose(fptr10);// close connection
  return 0;
}
```

1. Linear Search

Description:-

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

```
1.We start from the first element(k) in the array and compare k with each
element x.
2.If x == k, we return the index.
3.Else we return -1
```

Source Code:

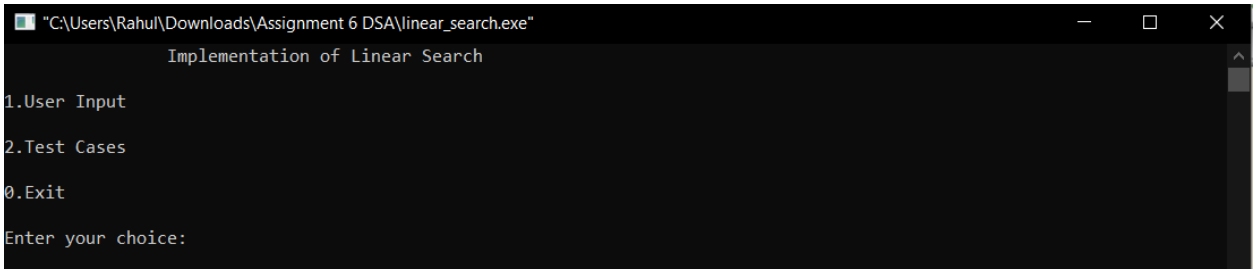
```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 10000000

int array[MAX];

long int linear_search(int array[], long int n, int x) {
    for (long int i = 0; i < n; i++){
        if (array[i] == x){
            return i;
        }
    }
    return -1;
}

void print_data(long int result,int data){
    if(result!=-1){
        printf("\n\nThe element %d is found at index %ld",data,result);
    }
    else{
        printf("\n\nThe element %d was not found in the array",data);
    }
}

void test_data(){
```

```
    system("cls");
    clock_t t;
    printf("\t\tSearching data in a predefined test case");
    FILE *fptr,*bin_ptr;
    int i=0,value,data;
    long int num=0;
    if ((fptr = fopen("high_integers_100000.txt","r")) == NULL){
        printf("Error! opening file");
        exit(1);
    }
    while(fscanf(fptr,"%d\n",&value)==1){
        array[i]=value;
        i++;
        num++;
    }
    fclose(fptr);
    printf("\n\nThe number of elements in the dataset are: %d",num);
    printf("\n\nEnter the element to be searched: ");
    scanf("%d",&data);
    t = clock();
    long int result=linear_search(array,num,data);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    print_data(result,data);
    bin_ptr = fopen("save_test.txt", "a+");
    fprintf(bin_ptr,"%f\n ",time_taken);
    printf("\n\nExecution Time saved in File");
    fclose(bin_ptr);
    getch();
    return;
}

void user_data(){
    system("cls");
    int i,num,data;
    clock_t t;
    printf("\t\tUser Input\n\n");
    printf("Enter the number of data: ");
    scanf("%d",&num);
    printf("\n\nEnter The data: ");
    for(i=0;i<num;i++){
        scanf("%d",&array[i]);
    }
    printf("\n\n Enter the data you want to search: ");
    scanf("%d",&data);
    t = clock();
    long int result=linear_search(array,num,data);
```

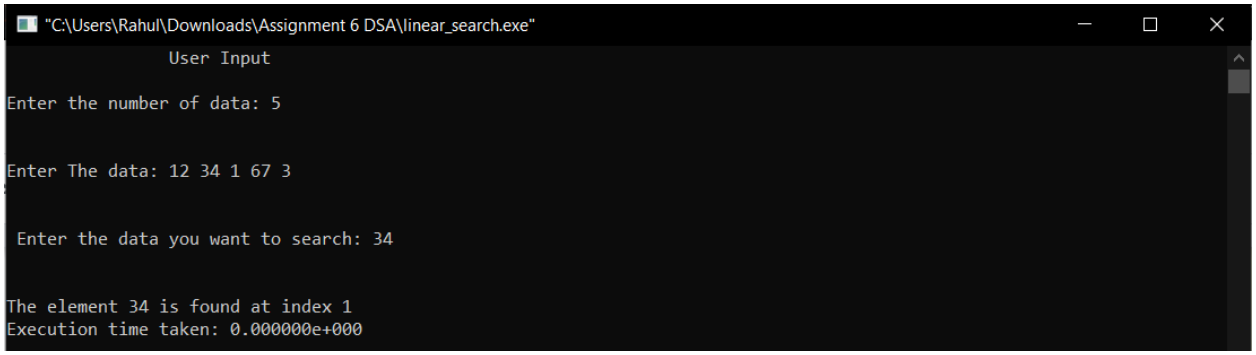
```
    t = clock() - t;
    print_data(result,data);
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("\nExecution time taken: %e  \n", time_taken);
    getch();
    return ;
}
int main() {
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Linear Search");
      printf("\n\n1.User Input ");
      printf("\n\n2.Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: test_data();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

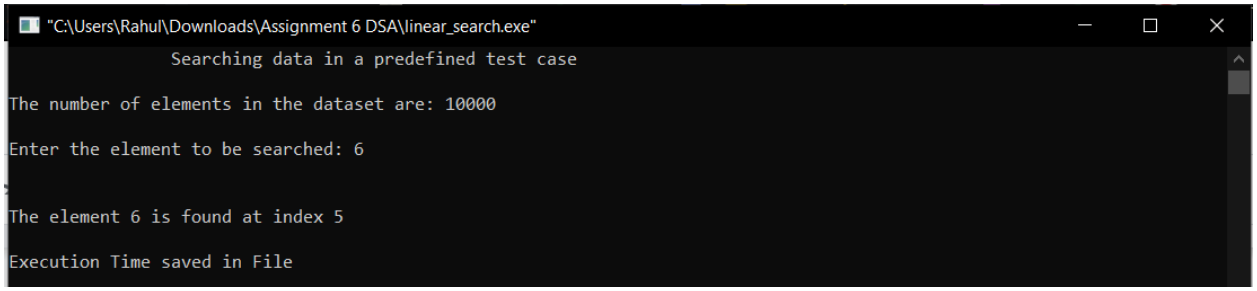
Userdrive



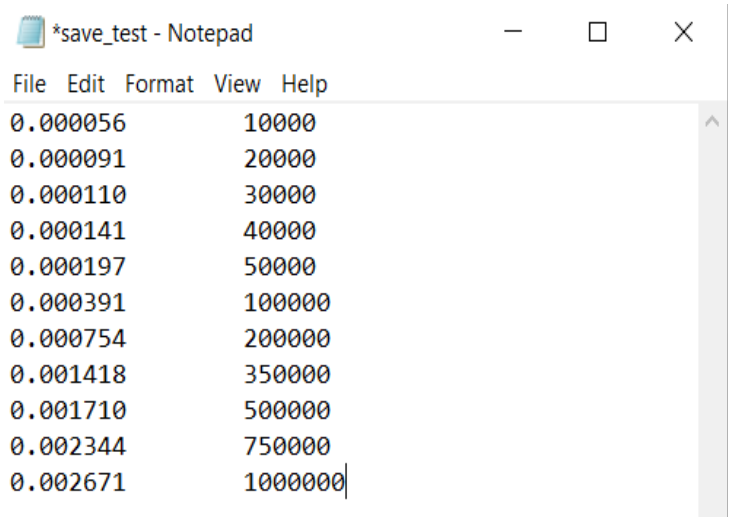
Searching small set of inputs



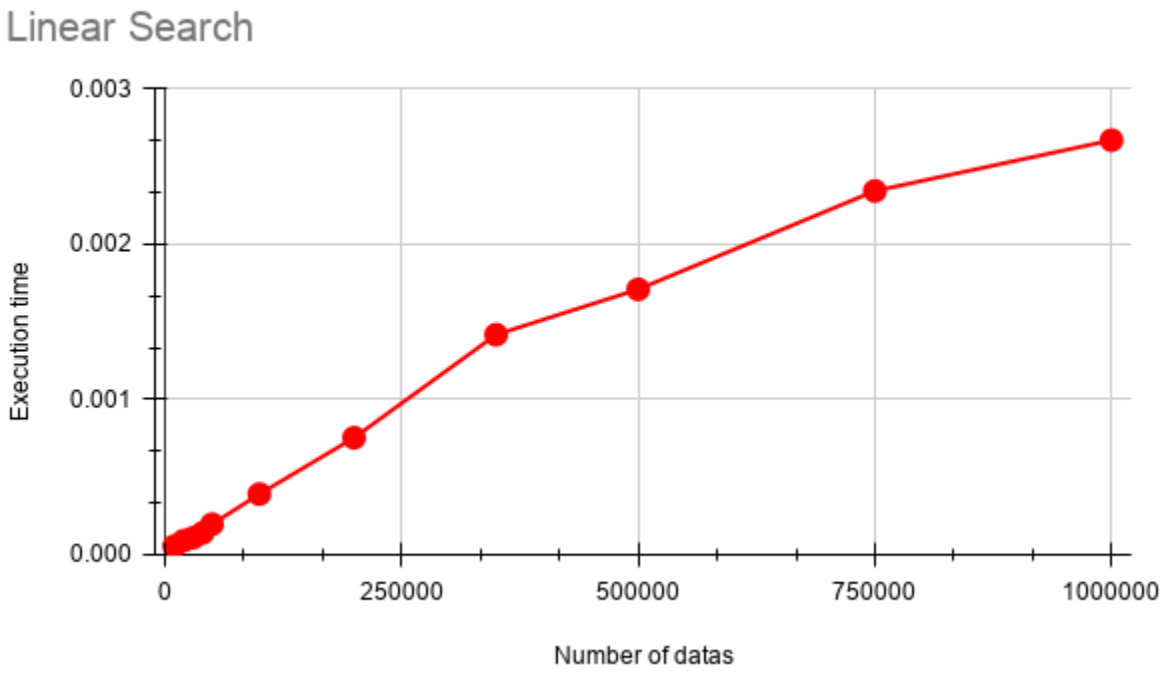
Next we will run the code for predefined test files. The Execution time output for every file input is stored in another file.



Execution time for all datasets:



Graph from the obtained data



2. Binary Search

Description:

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array.The array has to be sorted to apply Binary search.

Algorithm

1. Let the element to be found is x. Two pointers low and high are set at the lowest and the highest positions respectively.
2. Find the middle element mid(m) of the array ie. arr[(low + high)/2]
3. If x == mid, then return mid.Else, compare the element to be searched with m.
4. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
5. Else (x is smaller) recur for the left half.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
#include<time.h>
#define MAX 10000000

int array[MAX];

long int binary_search(int array[], int x, long int low,long int high) {
  if (high >= low) {
    long int mid = low + (high - low) / 2;
    if (array[mid] == x)//if found at middle returning it
      return mid;
    if (array[mid] > x)//searching in the left half
      return binary_search(array, x, low, mid - 1);
    return binary_search(array, x, mid + 1, high);//else searching in right
half
  }

  return -1;
}
void print_data(long int result,int data){
```

```
    if(result!=-1){
        printf("\n\nThe element %d is found at index %ld",data,result);
    }
    else{
        printf("\n\nThe element %d was not found in the array",data);
    }
}

void test_data(){
    system("cls");
    clock_t t;
    printf("\t\tSearching data in a predefined test case");
    FILE *fptr,*bin_ptr;
    int value,data;
    long int num=0,i=0;
    if ((fptr = fopen("high_integers_100000.txt","r")) == NULL){
        printf("Error! opening file");
        exit(1);
    }
    while(fscanf(fptr,"%d\n",&value)==1){
        array[i]=value;
        i++;
        num++;
    }
    fclose(fptr);
    printf("\n\nThe number of elements in the dataset are: %ld",num);
    printf("\n\nEnter the element to be searched: ");
    scanf("%d",&data);
    t = clock();
    long int result=binary_search(array,data,0,num-1);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    print_data(result,data);
    bin_ptr = fopen("save_test.txt", "a+");
    fprintf(bin_ptr,"%f\t%d\n",time_taken,num);
    printf("\n\nExecution Time saved in File");
    fclose(bin_ptr);
    getch();
    return;
}

void user_data(){
    system("cls");
    int i,num,data;
    clock_t t;
    printf("\t\tUser Input\n\n");
    printf("Enter the number of data(in sorted order): ");
```

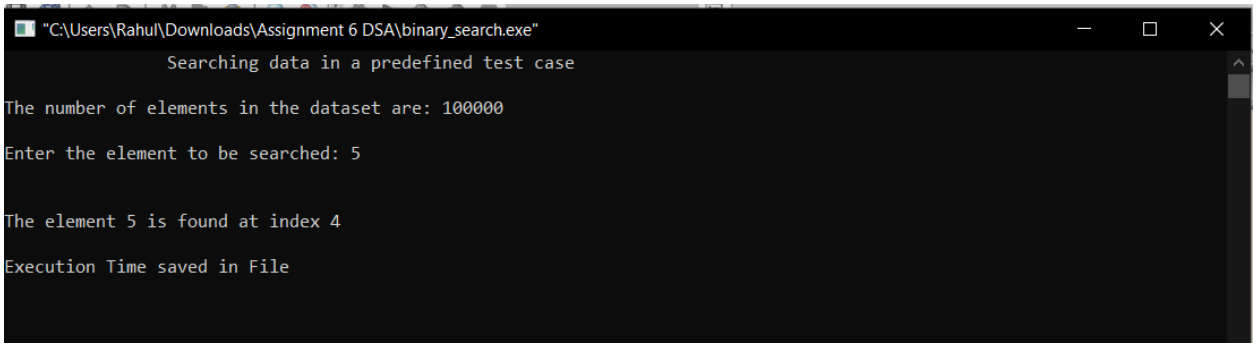
```
    scanf("%d",&num);
    printf("\n\nEnter The data: ");
    for(i=0;i<num;i++){
        scanf("%d",&array[i]);
    }
    printf("\n\n Enter the data you want to search: ");
    scanf("%d",&data);
    t = clock();
    long int result=binary_search(array,data,0,num-1);
    t = clock() - t;
    print_data(result,data);
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("\nExecution time taken: %e  \n", time_taken);
    getch();
    return ;
}
int main() {
    int ch;
    do{
      system("cls");
      printf("\t\tImplementation of Binary Search");
      printf("\n\n1.User Input ");
      printf("\n\n2.Test Cases");
      printf("\n\n0.Exit");
      printf("\n\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch){
            case 1: user_data();
                break;
            case 2: test_data();
                break;
            case 0: exit(0);
            default: printf("Invalid Choice");
                break;
        }
    }while(1);
    return 0;
}
```

Output Console:

User drive

Searching from small number of inputs

Next we will run the code for predefined test files. The Execution time output for every file input is stored in another file.



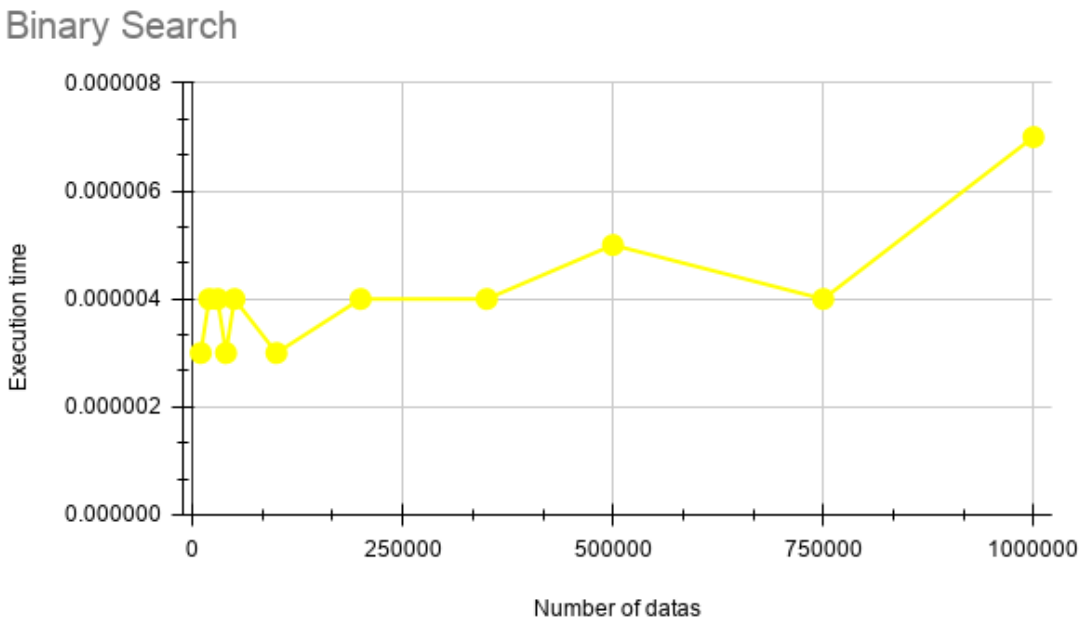
Execution time for all the datasets

*save_test - Notepad

File Edit Format View Help

0.000003	10000
0.000004	20000
0.000004	30000
0.000003	40000
0.000004	50000
0.000003	100000
0.000004	200000
0.000004	350000
0.000005	500000
0.000004	750000
0.000007	1000000

Graph from the obtained datas



Searching Algorithm Comparison

Number of datas	Linear Search	Binary Search
10000	0.000056	0.000003
20000	0.000091	0.000004
30000	0.00011	0.000004
40000	0.000141	0.000003
50000	0.000197	0.000004
100000	0.000391	0.000003
200000	0.000754	0.000004
350000	0.001418	0.000004
500000	0.00171	0.000005
750000	0.002344	0.000004
1000000	0.002671	0.000007