upes-new-logo

**CANDIDATE’S DECLARATION**

I/We hereby certify that the project work entitled **“Comparative Analysis of Sorting Algorithms”** in partial fulfillment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in OIL AND GAS INFORMATICS and submitted to the Department of Informatics at School of Computer Science, University of Petroleum & Energy Studies, Dehradun , is an authentic record of my/ our work carried out during a period from **August**, **2019** to **November**,**2019** under the supervision of **Mr. Vivek Shahare, Ass.Proffesor(CIT)**

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

**Mustafa Hakeem**

**Anurag Pant**

**Srajan Sahu**

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_2019 **Mr. Vivek Shahare**

Project Guide

**Dr. Thipendra Pal Singh**

**Professor and H.O.D**

Department of Informatics

School of Computer Science

University of Petroleum & Energy Studies

Dehradun – 248 001 (Uttarakhand)

**Contents**

**S.No. Name Page No.**

1. Introduction 06
2. Related Study 08
3. Problem Statement 12
4. Objective 13
5. Design 16
6. Implementation 17
7. Result analysis 22
8. Appendix 23

References 24

**Table of Figures**

**S.No. Name Page No.**

2.1 Bogosort 09

2.2 Shellsort 10

2.3 Iterativemerge 12

6.1 Comparision Table 19

6.2 First output 20

6.3 Second output 20

6.4 Third output 21

6.5 Fourth output 21

7.1 Result Graph 22

1. **Introduction**

Most often in programming, when an algorithm is developed to solve a particular problem it is not regarded as a complete success. Often it emerges the question of the performance analysis for the created algorithm: Complexity and Portability on different system.

Complexity is the measure by which an algorithm can be evaluated in comparison with other algorithms that solve the same problem. The aim of the complexity assess is to provide an information about the performances of an algorithm before transpose it in a programming language and without considering the particularities set of the incoming data. Evaluating the complexity of the algorithm will assume the estimation of:

* The time complexity: number of executions of a certain statement. Usually this statement is the most significant for the particular problem to solve, referred as the base statement. In sorting problems, the base statement is the comparison between items of the array;
* The space complexity: the amount of supplementary memory needed. Both time and space complexities are expressed as function in n (the input size). The time complexity is expressed in the *O* notation. For instance, an algorithm that executes the base statement *2n+3* times is a linear algorithm, having the time complexity. An algorithm that executes the base statement *(n2)+3* times is a polynomial algorithm, having the time complexity. If the algorithm executes the base statement *2n* times, it will be an exponential algorithm, having the time complexity.

The time complexity can be computed in three ways:

* the time related to the best-case scenario; sometimes, in sorting problems, this happens when the array is already ordered
* the time related to the worst-case scenario; sometimes, in sorting problems, this happens when the array is ordered descending;
* the mean time, the most significant of the three, is computed considering all the possible situations for the input data. The time complexity is a theoretical measure, usually it needs complex theories to be computed, therefore, in some cases, it is estimated using the run-time complexity. This means the algorithm is implemented in any programming language and the run-time is computed on a particular machine.

Why There r so my Algos:

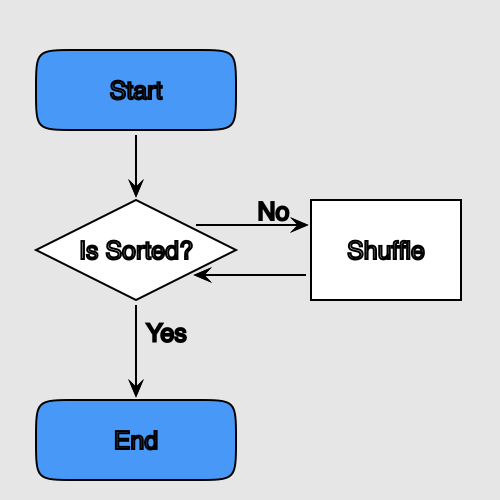
Whenever a new sorting algorithm is developed its credibility lies in sorting the given data faster. All algorithms developed till dates have their merits and demerits. Their performance depends on the type of data given as input and the implementation of the algorithm.  
Merge sort was one of the first algorithms developed to sort with a time complexity of nlogn.

Many of the algorithms work best for specific data structures and specific arrangements of inputs that is in specific situations when other algorithms fail to provide the minimum time complexity.  
Hence a programmer should be aware of all these sorting algorithms and apply the appropriate one according to the demand of the situation before him.

1. **Related Study**

**2.1 Bogo Sort[5][6]**

* if Data is unsorted
* Shuffle Data
* If data is sorted
* Return value
* Else repeate 2nd step



*Fig2.1: Bogosort example*

**Pseudo Code**

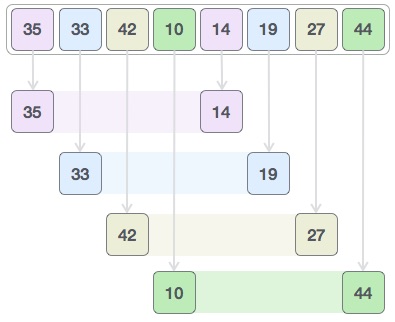
1.function bogosort(array)

2.while !ordered(array)

3.shuffle(array)

**2.2 Shell Sort[7][8]**

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.



*Fig2.2: Shell Sort Example*

**Pseudo Code**

procedure shellSort()

A : array of items

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

valueToInsert = A[outer]

inner = outer;

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

end while

A[inner] = valueToInsert

end for

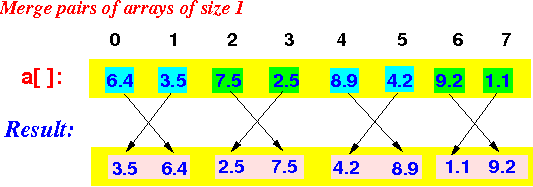
interval = (interval -1) /3;

end while

end procedure

**2.3 Iterative Merge Sort[9][10]**

The above function is recursive, so uses [function call stack](http://en.wikipedia.org/wiki/Call_stack) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution. Unlike [Iterative QuickSort](https://www.geeksforgeeks.org/iterative-quick-sort/), the iterative deepening doesn’t require explicit auxiliary stack.



*Fig2.3: Iterative Merge Example*

**Pseudo Code**

public static T[] Iterative(T[] array, IComparer<T> comparer)

{for (int i = 1; i <= array.Length / 2 + 1; i \*= 2)

{for (int j = i; j < array.Length; j += 2 \* i)

{Merge(array, j - i, j, Math.Min(j + i, array.Length), comparer);}}

return array;

}

1. **Problem Statement**

The method of organization of transaction data in the computer is the main area of the work. It defines the bank account transaction data of the computers which can be managed according to everyone’s need i.e. by time, by amount, by details etc. Also the, mismanagement of computer memory and time is another problem to look into for solution. It could be recalled that the necessary and efficiency of sorting activities cannot be over emphasized. However, this project research is aimed at taking a comprehensive look on some sorting algorithms in basic programming language to access their performance.

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.

1. **Objective**

This deals with presentation of information or data in an appreciable form and also to minimize the time used in searching for items. However, it can be seen that a well-sorted or organized file enhances easy searching of data while unsorted one will pose little or great problem to locate a given item in a large list. However, the comparison process will be based on the following

* Memory space used.
* Number of comparison made during sorting process.
* Time taken during the process.
* The input element: “Array”.

**5. Design**

**5.1 Methodology**

In this project, the following 12 algorithms are going to be compared in accordance to there best, average and worst cases. They also need to be compared on basis of space complexity or their memory usage. The stability of the methods are also need to be taken care about. The other factor is method on which the sorting algorithms works is also important for the comparison of different sorting algorithms.

The 5 sorting algorithms which are going to be compared in this following project are:

* **Merge Sort:[1][2]**

Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The *merge()* 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

* **Quick sort:[3][4]**

The key process in Quick Sort is *partition().* Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sort.ed array and put all smaller elements (smaller than *x*) before *x*, and put all greater elements (greater than *x)* after *x*. All this should be done in linear time

* **Bogo sort:[5][6]**

BogoSort also known as permutation sort, stupid sort, slow sort, shotgun sort or monkey sort is a particularly ineffective algorithm based on generate and test paradigm. The algorithm successively generates permutations of its input until it finds one that is sorted.

* **Iterative Merge Sort:[9][10]**  
  The above function is recursive, so uses [function call stack](http://en.wikipedia.org/wiki/Call_stack) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution. Unlike [Iterative QuickSort](https://www.geeksforgeeks.org/iterative-quick-sort/), the iterative MergeSort doesn’t require explicit auxiliary stack.
* **Shell Sort:[7][8]**

[ShellSort](http://en.wikipedia.org/wiki/Shellsort)is mainly a variation of [Insertion Sort](http://quiz.geeksforgeeks.org/insertion-sort/). In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of *h*.

**5.2 Algorithm of Project**

1. First Step is to take user input for which sorting algorithm user needs to process.
2. Secondly to execute the sorting algorithm which user wants to execute.
3. Next step is to display the random numbers that are needed to sort.
4. And then to display the sorted numbers of the input array.
5. And then finally display the time taken by that algorithm for sorting the numbers.
6. And then end the compilation process.

**6.Implementation**

**6.1 Pseudo Code**

* **Bogo sort**

1.function bogosort(array)

2.while !ordered(array)

3.shuffle(array)

* **Shell Sort**

procedure shellSort()

A : array of items

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

valueToInsert = A[outer]

inner = outer;

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

end while

A[inner] = valueToInsert

end for

interval = (interval -1) /3;

end while

end procedure

* **Iterative Merge Sort**

public static T[] Iterative(T[] array, IComparer<T> comparer)

{for (int i = 1; i <= array.Length / 2 + 1; i \*= 2)

{for (int j = i; j < array.Length; j += 2 \* i)

{Merge(array, j - i, j, Math.Min(j + i, array.Length), comparer);

}

}

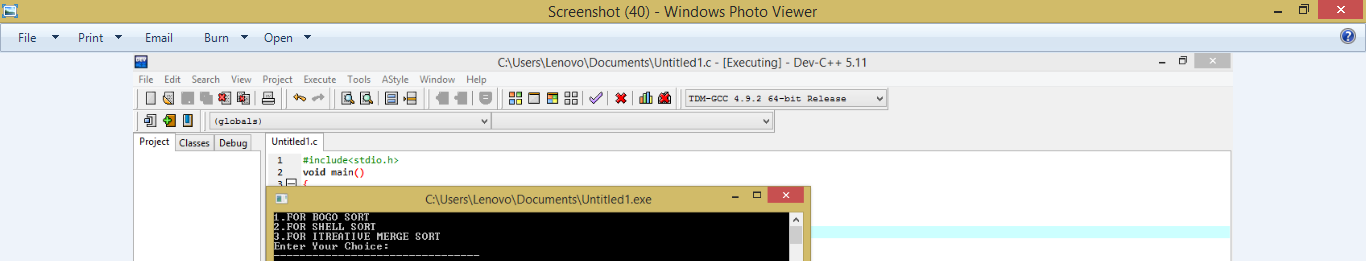
return array;

}

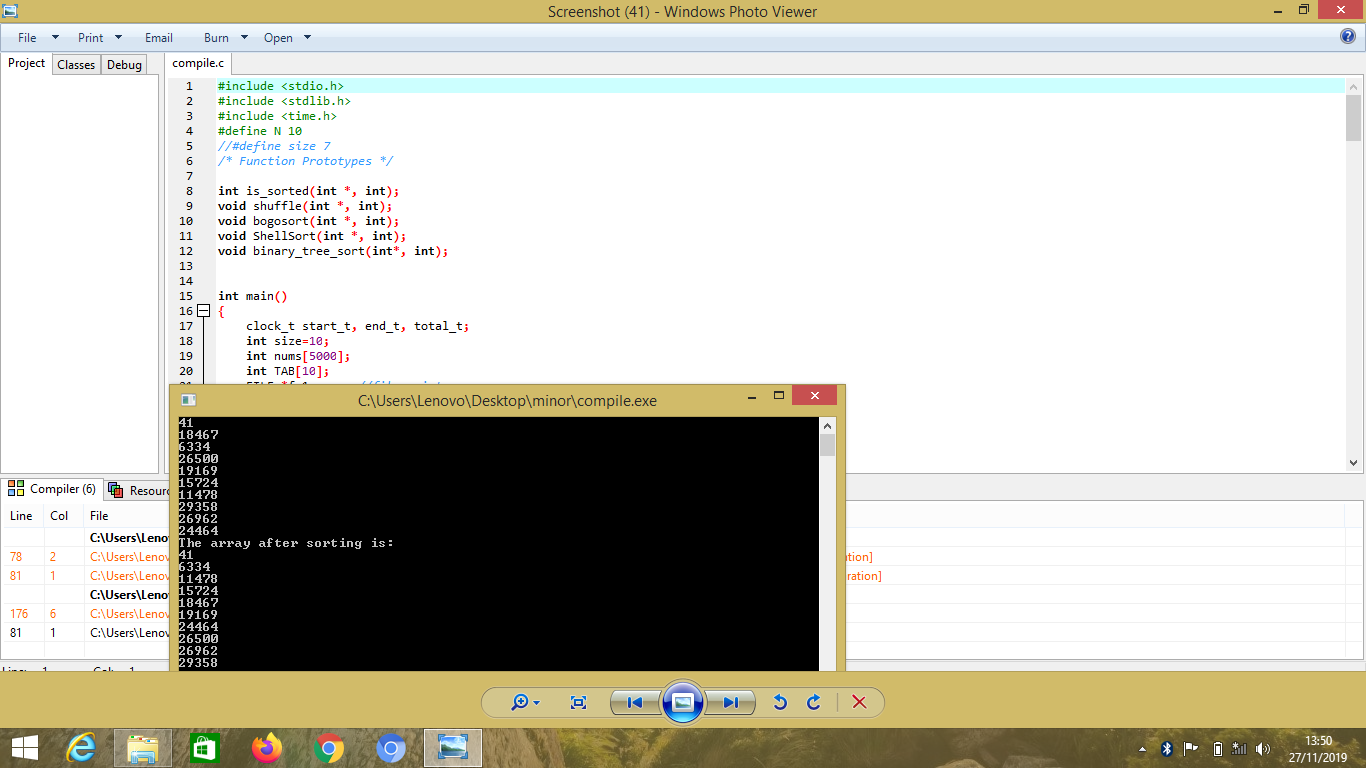
| **Name** |  | **Average** | **Worst** | **Memory** | **Stable** | **Method** |
| --- | --- | --- | --- | --- | --- | --- |
| [Merge sort](https://en.wikipedia.org/wiki/Merge_sort) | {\displaystyle n\log n} *O(n log n)* | *O(n log n){\displaystyle n\log n}* | {\displaystyle n\log n} *O(n log n)* | *n* A hybrid block merge inmemory. | Yes | Merging |
| [Quicksort](https://en.wikipedia.org/wiki/Quicksort) | *O(n log n) {\displaystyle n\log n} variation is n* | *O(n log n) {\displaystyle n\log n}* | *O(n2){\displaystyle n^{2}}* | *on average, worst case space complexity is n;* | Typical in-place sort is not stable; stable versions exist. | Partitioning |
| [Shell sort](https://en.wikipedia.org/wiki/Shell_sort) | *O(n log n){\displaystyle n\log n}* | dependsDepends on gap sequence | *O(n2)Depends on gap sequence; best known is {\displaystyle n^{4/3}}* | *1* | No | Insertion |
| [Bogosort](https://en.wikipedia.org/wiki/Bogosort) | *O(n)n* | *O((n+1)!){\displaystyle (n\times n!)}* | *O(n){\displaystyle \infty }* | *1* | No | Yes |
| [Iterative merge sort](https://en.wikipedia.org/wiki/In-place_merge_sort) | *Θ(nLogn).—* | *Θ(nLogn).—* | *Θ(nLogn).{\displaystyle n\log ^{2}n} See above, for hybrid, that is {\displaystyle n\log n}* | *1* | Yes | Merging |

*Fig6.1: Comparison table*[11]

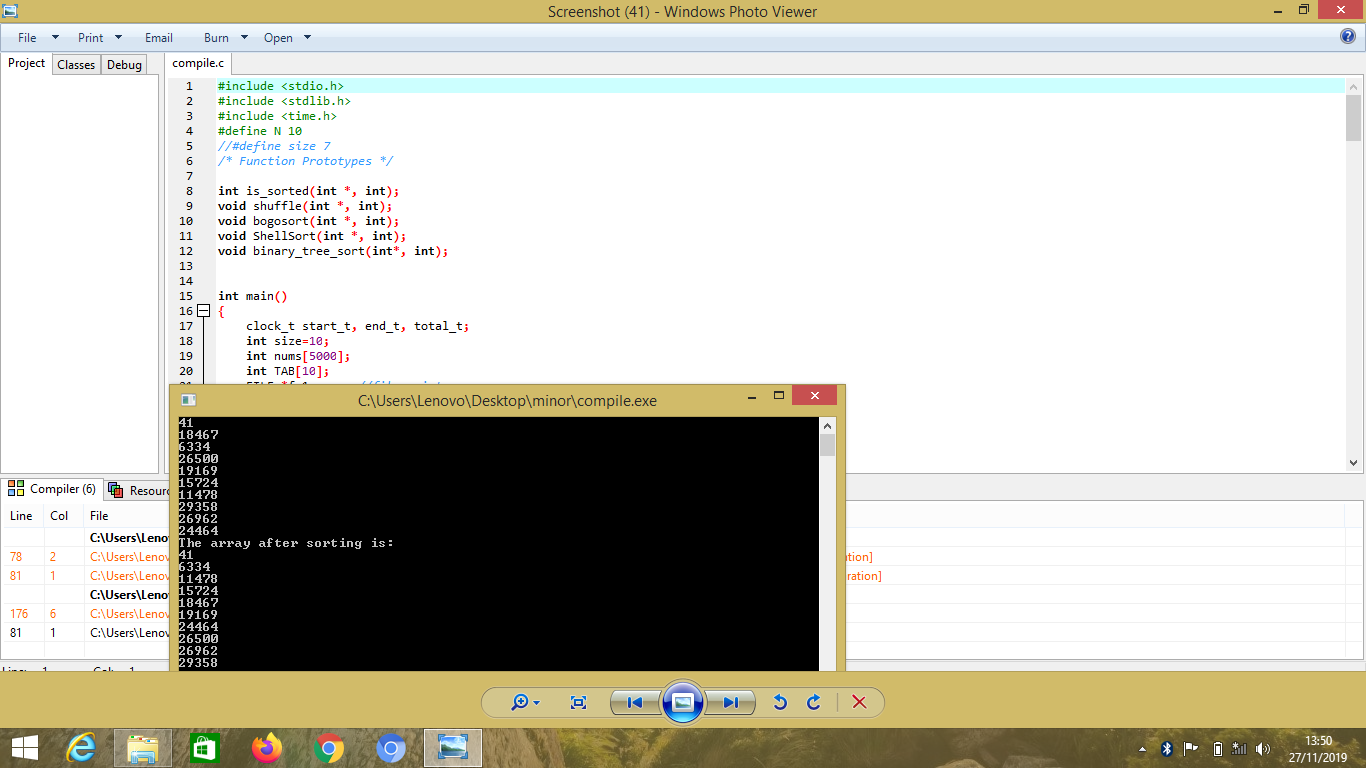
* 1. **Output Screen**



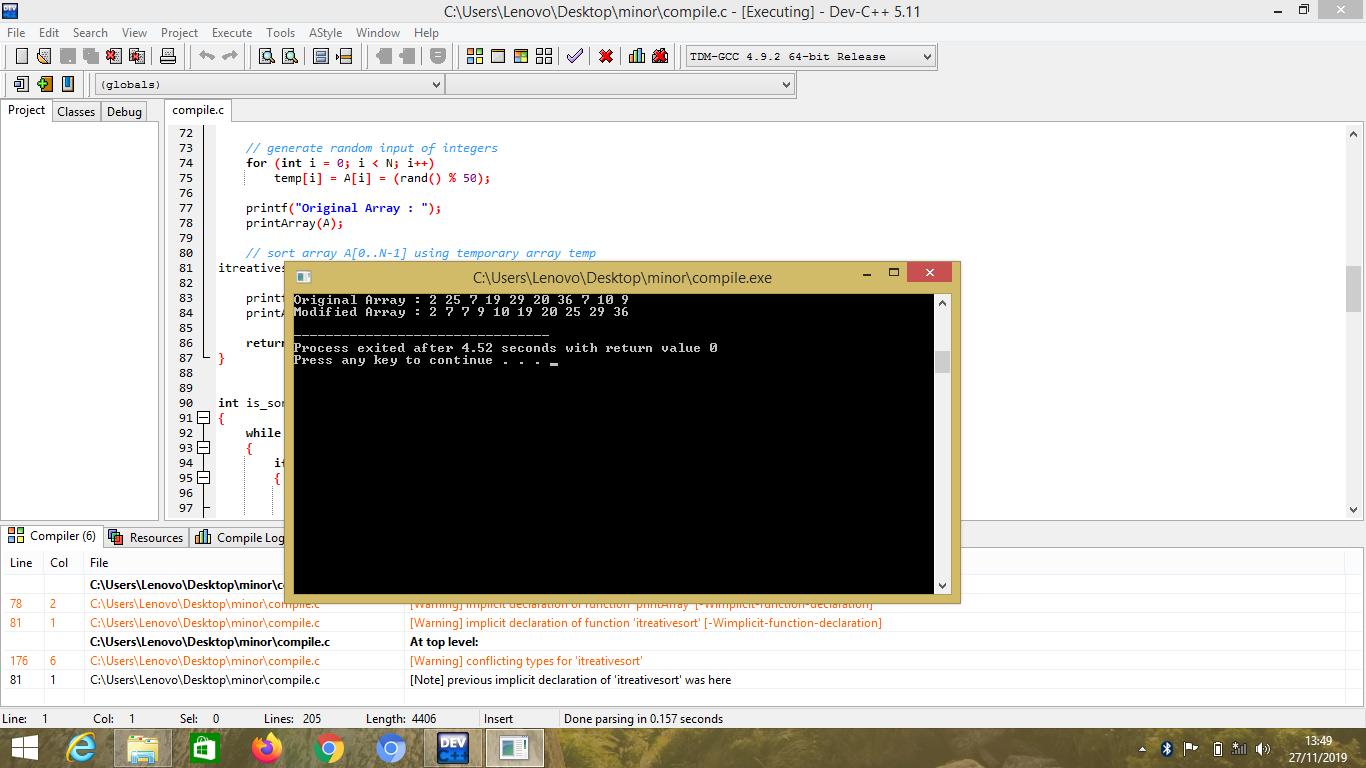
*Fig6.2: First Output*



*Fig6.3: Second Output*

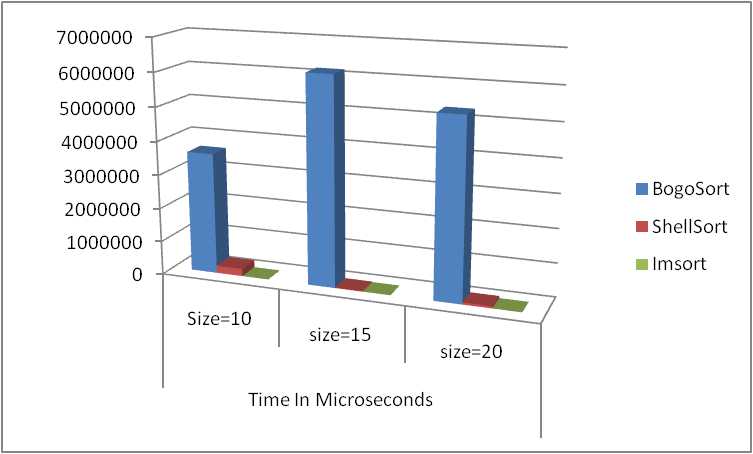


*Fig6.4: Third Output*



*Fig6.5: Fourth Output*

1. **Result analysis**

****

*Fig7.1: Result Graph*

According to our Results in this Project, we analyze that among these three sorting Algorithms

The best orders of sorting algorithms are as follows:

1.Iterative merge sort.

2.Shell Sort.

3.Bogo sort.

Therefore, we conclude that the Iterative merge sorting algorithm is best among all.

1. **Appendix**

**Project Code**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 10

//#define size 7

/\* Function Prototypes \*/

int is\_sorted(int \*, int);

void shuffle(int \*, int);

void bogosort(int \*, int);

void ShellSort(int \*, int);

void binary\_tree\_sort(int\*, int);

int main()

{

clock\_t start\_t, end\_t, total\_t;

int size=10;

int nums[5000];

int TAB[10];

FILE \*fp1; //file pointer

int i;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* code starts here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for(i=0;i<50;i++) //initialize array with 0

nums[i]=0;

i=0; //clean up and initialize LCV

fp1=fopen("file\_10.txt","r");

while((fscanf(fp1,"%d",&nums[i]))!=EOF) //scanf and check EOF

{

printf("%d\n",nums[i]);

i++;

}

fclose(fp1);

start\_t = clock();

bogosort(nums, size);

printf("The array after sorting is:\n");

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

{

printf("%d\n", nums[i]);

}

printf("\n");

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

printf("Total time taken by CPU: %f\n", total\_t );

int a[500]; //up to 50 element int array

FILE \*fp3; //file pointer

int l=10;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* code starts here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for(i=0;i<50;i++) //initialize array with 0

a[i]=0;

i=0; //clean up and initialize LCV

fp3=fopen("file\_10.txt","r");

while((fscanf(fp1,"%d",&a[i]))!=EOF) //scanf and check EOF

{

printf("%d\n",a[i]);

i++;

}

fclose(fp3);

ShellSort(a,l);

printf("The sorted elements are ::\n ");

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

printf("%d \n",a[i]);

printf("\n");

//itreative deeping

int A[N], temp[N];

srand(time(NULL));

// generate random input of integers

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

temp[i] = A[i] = (rand() % 50);

printf("Original Array : ");

printArray(A);

// sort array A[0..N-1] using temporary array temp

itreativesort(A, temp, 0, N - 1);

printf("Modified Array : ");

printArray(A);

return 0;

}

int is\_sorted(int \*a, int n)

{

while (--n >= 1)

{

if (a[n] < a[n - 1])

{

return 0;

}

}

return 1;

}

void shuffle(int \*a, int n)

{

int i, t, temp;

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

{

t = a[i];

temp = rand() % n;

a[i] = a[temp];

a[temp] = t;

}

}

void bogosort(int \*a, int n)

{

while (!is\_sorted(a, n))

{

shuffle(a, n);

}

}

void ShellSort(int a[], int n)

{

int i, j, increment, tmp;

for(increment = n/2; increment > 0; increment /= 2)

{

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

{

tmp = a[i];

for(j = i; j >= increment; j -= increment)

{

if(tmp < a[j-increment])

a[j] = a[j-increment];

else

break;

}

a[j] = tmp;

}

}

}

int min(int x, int y)

{

return (x < y) ? x : y;

}

// Merge two sorted subarrays A[from .. mid] and A[mid + 1 .. to]

void deeping(int A[], int temp[], int from, int mid, int to)

{

int k = from, i = from, j = mid + 1;

// loop till there are elements in the left and right runs

while (i <= mid && j <= to)

{

if (A[i] < A[j])

temp[k++] = A[i++];

else

temp[k++] = A[j++];

}

// Copy remaining elements

while (i < N && i <= mid)

temp[k++] = A[i++];

// Don't need to copy second half

// copy back to the original array to reflect sorted order

for (int i = from; i <= to; i++)

A[i] = temp[i];

}

// Iteratively sort array A[low..high] using temporary array

void itreativesort(int A[], int temp[], int low, int high)

{

// divide the array into blocks of size m

// m = [1, 2, 4, 8, 16...]

for (int m = 1; m <= high - low; m = 2\*m)

{

// for m = 1, i = 0, 2, 4, 6, 8

// for m = 2, i = 0, 4, 8

// for m = 4, i = 0, 8

// ...

for (int i = low; i < high; i += 2\*m)

{

int from = i;

int mid = i + m - 1;

int to = min(i + 2\*m - 1, high);

deeping(A, temp, from, mid, to);

}

}

}

// Utility function to print an given array

int printArray(int A[])

{

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

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

}

printf("\n");

}