**A REPORT OF SUMMER**

**TRAINING**

**AT**



*ON*

**MULTIPLE STRING SEARCHING**

**IN MULTIPLE FILE(S)**

**SUBMITTED BY:**

**Name:** *PRITHVI RAJ CHAUHAN*

**B.Ttech** *Information Technology*

*(2015- 2019)*

**Roll No.** *02896403115*

**ACKNOWLEDGEMENT**

The objective of training is to work in a firm with a professional work driven environment where I can utilize and apply my knowledge, skills which would enable me as a fresh graduate to grow while fulfilling organizational goals.

I would like to express my gratitude to *Mr. Kasi Viswanath Kethineni*, my training coordinator for his constant cooperation. He was always there with his competent guidance and valuable suggestion throughout the pursuance of my training project.

I would also like to express my gratitude to sincere *Mr*. *Valsa Raj* *Uchamballi*, TechnicalDirector and *Ms*. *Ratnaboli Ghorai Dinda*, Senior Technical Director.

I would also like to thank all my teachers and *Mr. M L Sharma* , HOD IT dept. from Maharaja Agrasen Institute Of Technology for giving me this opportunity to work under an organization and develop my skills to boost my career aspects.

**DECLARATION**

I **Prithvi Raj Chauhan**, Student of **B.Tech 5Th Semester,** studying at **Maharaja Agrasen Institute of Technology(MAIT)**, New Delhi hereby declare that summer training on “**MULTIPLE STRING SEARCHING IN MULTIPLE FILE(s)**” submitted to “**NATIONAL INFORMATICS CENTRE(NIC)**”, Delhi in partial fulfilment of Degree of **Bachelors of Technology** is the original work conducted by me.

The information and data given in this report is authentic to the best of my knowledge.

This Summer training report is not being submitted to any other University for award of any other Degree, Diploma, Fellowship.

**PREFACE**

Multiple searches are required in day today activities of logs and string processing activities. This project is based of searching of multiple strings in multiple files at the same time. All the given patterns or strings are searched against given multiple files/folders at the same time through parallel searching. The result is stored in a separate file after Sorting. All the searching and sorting algorithms that can be used are briefly described in this report.

**ORGANIZATION INTRODUCTION**

**National Informatics Centre (NIC)** was established in **1976**, and has since emerged as a "prime builder" of e-Government / e-Governance applications up to the grassroots level as well as a promoter of digital opportunities for sustainable development. NIC, through its ICT Network, "NICNET", has institutional linkages with all the Ministries /Departments of the Central Government, 36 State Governments/ Union Territories, and about 688 District administrations of India. NIC has been instrumental in steering e-Government/e-Governance applications in government ministries/departments at the Centre, States, Districts and Blocks, facilitating improvement in government services, wider transparency, promoting decentralized planning and management, resulting in better efficiency and accountability to the people of India.

"Informatics-led-development" programme of the government has been spearheaded by NIC to derive competitive advantage by implementing ICT applications in social & public administration. The following major activities are being undertaken:

* Setting up of ICT Infrastructure
* Implementation of National and State Level e-Governance Projects
* Products and Services
* Consultancy to the government departments
* Research and Development
* Capacity Building

NIC provides Nationwide Common ICT Infrastructure to support e-Governance services to the citizen, Products and Solutions designed to address e-Governance Initiatives, Major e-Governance Projects, State/UT Informatics Support and district level services rendered.

NIC has set up state-of-the-art ICT infrastructure consisting of National and state Data Centres to manage the information systems and websites of Central Ministries/Departments, Disaster Recovery Centres, Network Operations facility to manage heterogeneous networks spread across Bhawans, States and Districts, Certifying Authority, Video-Conferencing and capacity building across the country. National Knowledge Network (NKN) has been set up to connect institutions/organizations carrying out research and development, Higher Education and Governance with speed of the order of multi Gigabits per second. Further, State Government secretariats are connected to the Central Government by very high speed links on Optical Fibre Cable (OFC).

**TABLE OF CONTENTS**

1. PROJECT DETAILS ……………………….……………………………… 1
2. SEARCHING ALGORITHMS ……………..……………………………… 4
   1. LINEAR SEARCH …………………..……………………………… 4
   2. BINARY SEARCH …………………...…………………………….. 5
   3. JUMP SEARCH ……………………….……………………………. 7
   4. INTERPOLATION SEARCH ………………………………………. 9
   5. EXPONENTIAL SEARCH ………………………………………… 11
   6. TERNARY SEARCH ………………………………………………. 12
3. SEARCHING WITH TIME COMPARISION ……………………………. 14

1. SORITNG ALGORITHMS …………………..………………………… 17
   1. SELECTION SORT ………………….………………………….... 17
   2. BUBBLE SORT ………………….……………………………….. 19
   3. INSERTION SORT ……………………..……………………….... 21
   4. MERGE SORT ……………………………………………………. 22
   5. HEAP SORT ………………………………………………………. 25
   6. QUICK SORT ……………………….…………………………….. 28
   7. RADIX SORT ……………………….…………………………….. 30
   8. BUCKET SORT …………………….……………………………. 33
   9. SHELL SORT ……………………….……………………………. 35
   10. COMB SORT ……………………….……………………………. 37
   11. PIGEONHOLE SORT ……………………………………………. 39
   12. CYCLE SORT ……………………………………………………. 41
2. SORTING ALGORITHM WITH EXECUTION TIME …….……………... 44
3. CONCLUSION ……………………………………………………………... 47
4. REFERENCES …………………………………………………………….. 48

**TABLE OF FIGURES**

1. Searching Program ……………………………………………………… .. (2-3)

2. Sorting Program …………………………………………………………. . (44-46)

3. Comparison Table …………………………………………………………. 47

**PROJECT DETAILS**

This project comprises of searching and sorting parts of large files mainly logs files and database files.

**Problem Statement**

Mainly while working on large files a user may face problem of searching multiple string in multiple files. And while searching in huge files the time required for execution of searching is quite high. So this project makes use various searching algorithms for searching at the same time minimizing time of execution for multiple search format.

**TOOLS USED**

**LANGUAGE:** C++

**PLATFORM:** LINUX

**FILE TYPES AND CONTENT**

All the files of same data types like .txt, .log, .docx etc. will all be opened at once. The content of all files will be shown separately and user will be notified further.

**SEARCHING MULTIPLE STRINGS**

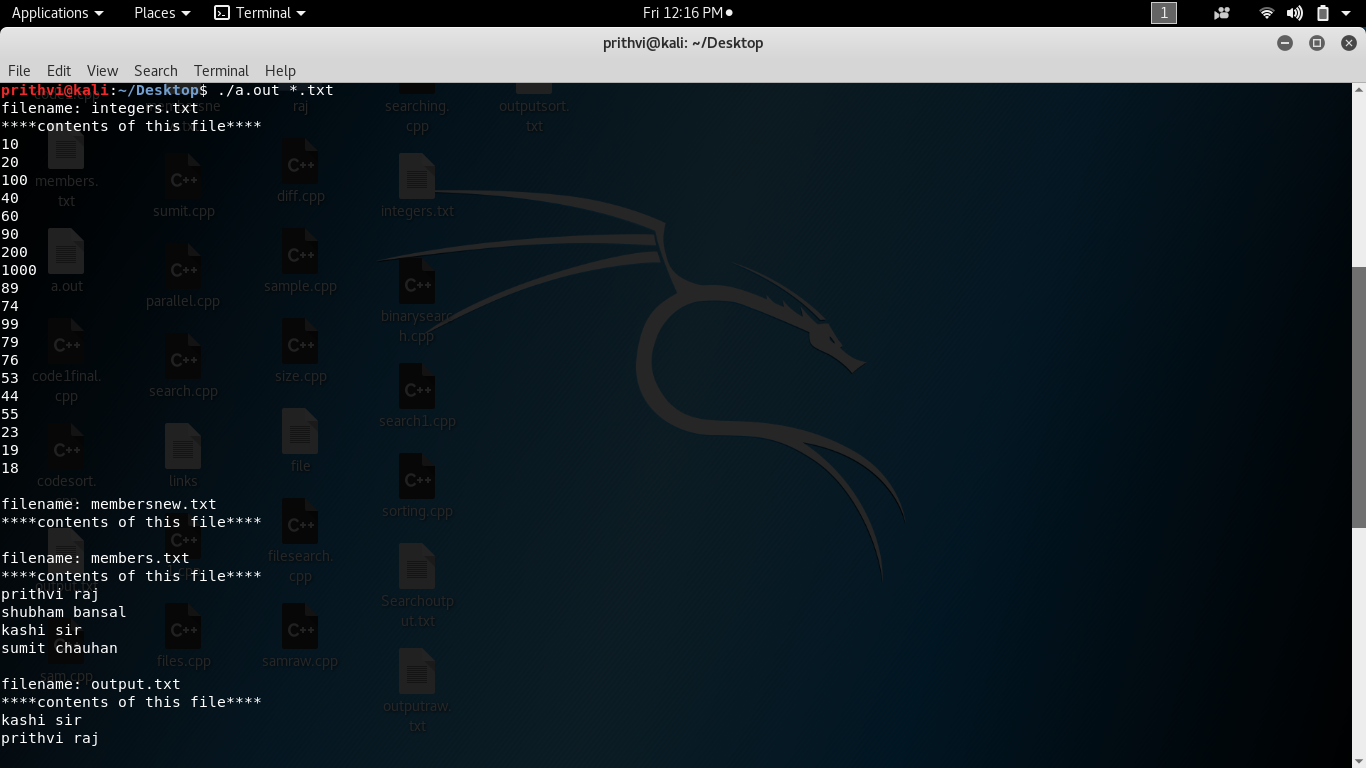
User will enter collection of random strings one by one. All entered strings will now be searched from all files and will be displayed as per as file name which contains it and line number which contains it. A separate output file will also be created for future reference.

**SORTING AND OUTPUT**

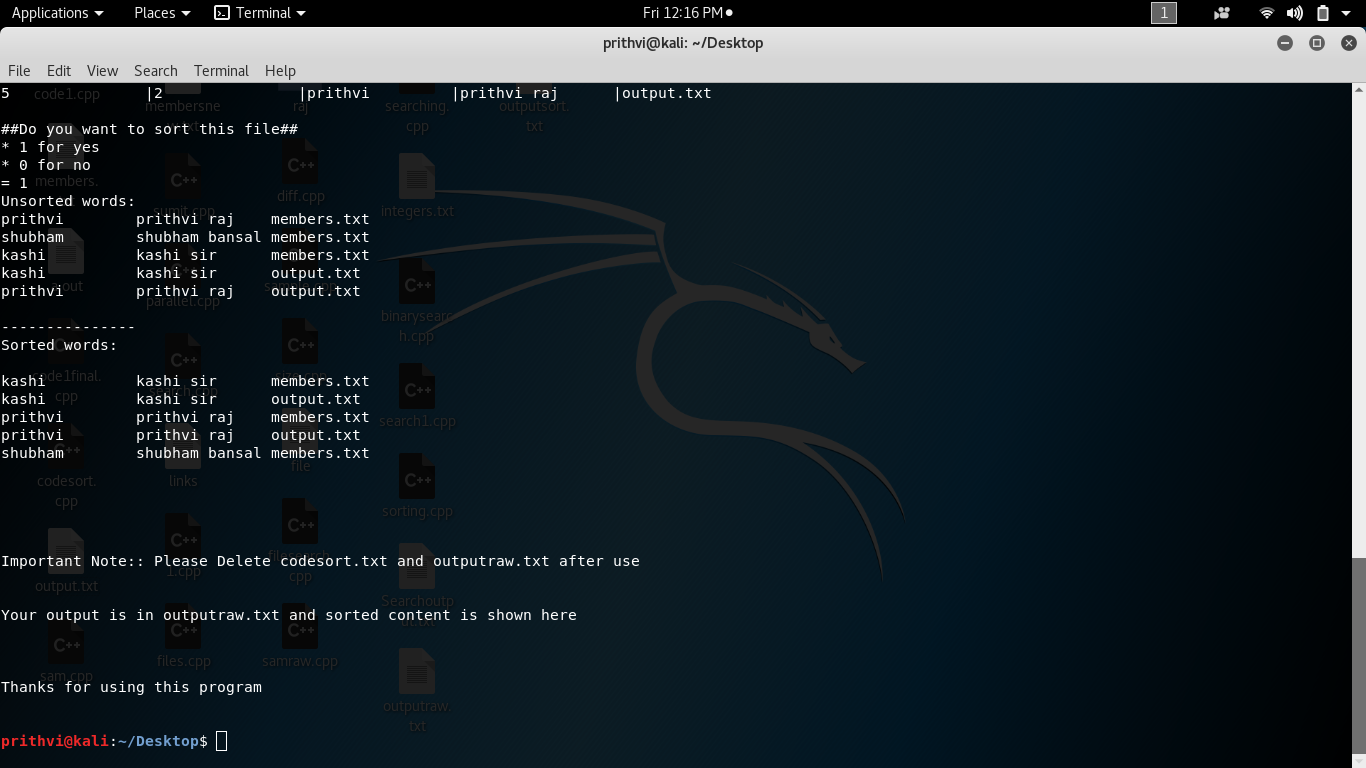
User will be notified if whether he wants a sorted output to his searched strings or not. If he selects yes the output to search will be sorted and displayed in the program.

**ABSTRACTION**

**OUTPUT OF SEARCHING PROGRAM**







**EXPLORATION PART**

Various searching and sorting algorithms are studied and their time complexity in various cases are compared. All details are mentioned as per as object.

**SEARCHING ALGORITHMS**

**LINEAR SEARCH**

A simple approach is to do **linear search**, i.e.

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
* If x matches with an element, return the index.
* If x doesn’t match with any of elements, return -1.

**SOURCE CODE**

// Linearly search x in arr[].  If x is present then return its

// location,  otherwise return -1

int search(int arr[], int n, int x)

{

    int i;

    for (i=0; i<n; i++)

        if (arr[i] == x)

         return i;

    return -1;

}

The **time complexity** of above algorithm is *O(n*).

**BINARY SEARCH**

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

**SOURCE CODE**

#include <stdio.h>

// A recursive binary search function. It returns location of x in

// given array arr[l..r] is present, otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

   if (r >= l)

   {

        int mid = l + (r - l)/2;

        // If the element is present at the middle itself

        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present

        // in left subarray

        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray

        return binarySearch(arr, mid+1, r, x);

   }

   // We reach here when element is not present in array

   return -1;

}

int main(void)

{

   int arr[] = {2, 3, 4, 10, 40};

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

   int x = 10;

   int result = binarySearch(arr, 0, n-1, x);

   (result == -1)? printf("Element is not present in array")

                 : printf("Element is present at index %d", result);

   return 0;

}

THE **TIME COMPLEXITY** OF BINARY SEARCH IS ***O(LOG(N)).***

**JUMP SEARCH**

Like [Binary Search](http://geeksquiz.com/binary-search/), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

Let’s consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.  
STEP 1: Jump from index 0 to index 4;  
STEP 2: Jump from index 4 to index 8;  
STEP 3: Jump from index 8 to index 16;  
STEP 4: Since the element at index 16 is greater than 55 we will jump back a step to come to index 9.  
STEP 5: Perform linear search from index 9 to get the element 55.

**SOURCE CODE**

int find\_element(int element)

{

   int jump\_step,prev=0;

   jump\_step=floor(sqrt(n));

   /\* Finding block in which element lies, if it is present \*/

   while(arr[prev]<element)

   {

       if(arr[jump\_step]>element || jump\_step>=n)

       {

            break;

       }

       else

       {

                prev=jump\_step;

                jump\_step=jump\_step+floor(sqrt(n));

       }

   }

   /\*Finding the element in the identified block \*/

   while(arr[prev]<element)

   {

        prev++;

   }

   if(arr[prev]==element)

   {

        return prev+1;

   }

   else

   {

        return -1;

   }

}

**Time Complexity** : O(√n)

**INTERPOLATION SEARCH**

The Interpolation Search is an improvement over [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to middle element to check. On the other hand interpolation search may go to different locations according the value of key being searched. For example if the value of key is closer to the last element, interpolation search is likely to start search toward the end side.

**SOURCE CODE**

int interpolationSearch(int arr[], int n, int x)

{

    // Find indexes of two corners

    int lo = 0, hi = (n - 1);

    // Since array is sorted, an element present

    // in array must be in range defined by corner

    while (lo <= hi && x >= arr[lo] && x <= arr[hi])

    {

        // Probing the position with keeping

        // uniform distribution in mind.

        int pos = lo + (((double)(hi-lo) /

              (arr[hi]-arr[lo]))\*(x - arr[lo]));

        // Condition of target found

        if (arr[pos] == x)

            return pos;

        // If x is larger, x is in upper part

        if (arr[pos] < x)

            lo = pos + 1;

        // If x is smaller, x is in lower part

        else

            hi = pos - 1;

    }

    return -1;

}

**Time Complexity** : If elements are uniformly distributed, then ***O (log log n))****.* In worst case it can take upto ***O(n).***

**EXPONENTIAL SEARCH**

Exponential search involves two steps:

1. Find range where element is present
2. Do Binary Search in above found range.

**How to find the range where element may be present?**  
The idea is to start with subarray size 1 compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.  
Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration).

**SOURCE CODE**

int exponentialSearch(int arr[], int n, int x)

{

    // If x is present at firt location itself

    if (arr[0] == x)

        return 0;

    // Find range for binary search by

    // repeated doubling

    int i = 1;

    while (i < n && arr[i] <= x)

        i = i\*2;

    //  Call binary search for the found range.

    return binarySearch(arr, i/2, min(i, n), x);//SAME AS FOR BINARY SEARCH

}

**Time Complexity :**O(Log n)

**TERNARY SEARCH**

Like linear search and binary search, ternary search is a searching technique that is used to determine the position of a specific value in an array. In binary search, the sorted array is divided into two parts while in ternary search, it is divided into 33 parts and then you determine in which part the element exists.

Ternary search, like binary search, is a divide-and-conquer algorithm. It is mandatory for the array (in which you will search for an element) to be sorted before you begin the search. In this search, after each iteration it neglects ⅓⅓part of the array and repeats the same operations on the remaining ⅔.

**SOURCE CODE**

int ternarySearch(int arr[], int l, int r, int x)

{

   if (r >= l)

   {

        int mid1 = l + (r - l)/3;

        int mid2 = mid1 + (r - l)/3;

        // If x is present at the mid1

        if (arr[mid1] == x)  return mid1;

        // If x is present at the mid2

        if (arr[mid2] == x)  return mid2;

        // If x is present in left one-third

        if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

        // If x is present in right one-third

        if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

        // If x is present in middle one-third

        return ternarySearch(arr, mid1+1, mid2-1, x);

   }

   // We reach here when element is not present in array

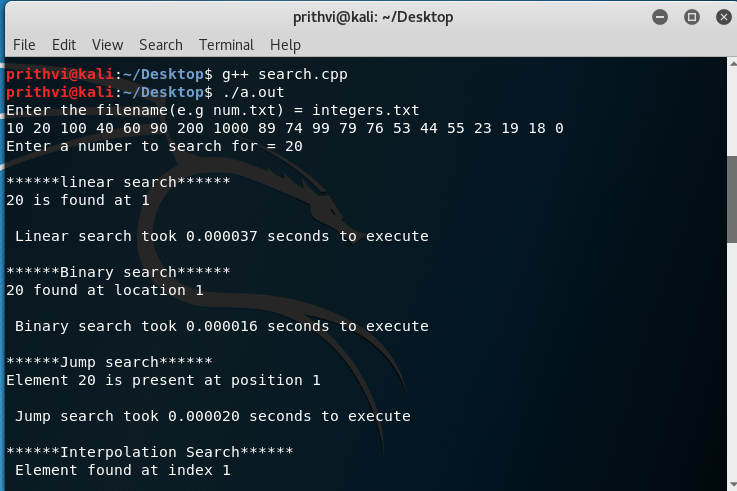
   return -1;

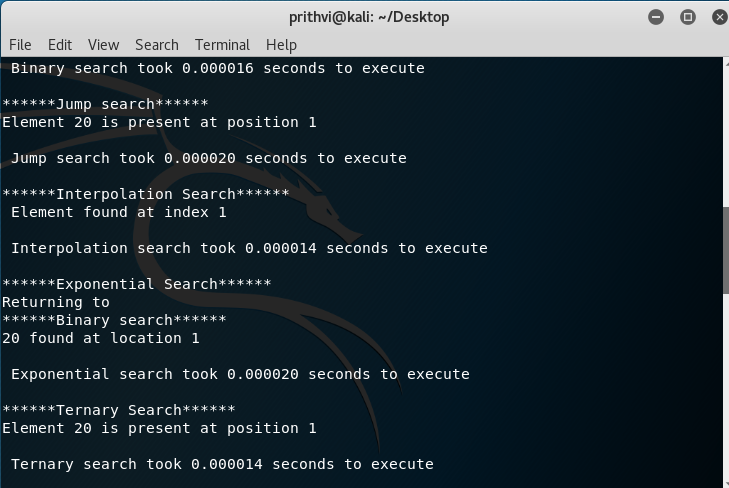
}

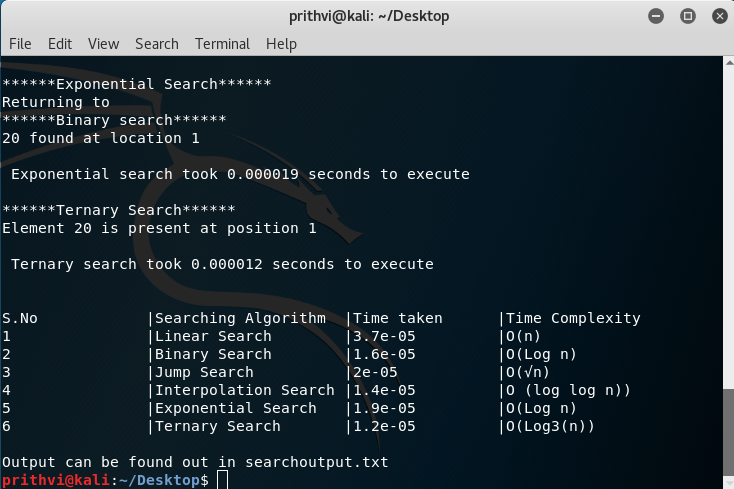
Time Complexity for Ternary search = *4clog3n + O(1)*

**SEARCHING WITH TIME COMPARISION**

**OUTPUTS TO MY CODE:**

****

****

****

****

**SORTING ALGORITHMS**

**SELECTION SORT**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

**SOURCE CODE**

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    // One by one move boundary of unsorted subarray

    for (i = 0; i < n-1; i++)

    {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i+1; j < n; j++)

          if (arr[j] < arr[min\_idx])

            min\_idx = j;

        // Swap the found minimum element with the first element

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

    }

}

**Time Complexity:**  *O(n2)* as there are two nested loops.

**BUBBLE SORT**

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

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

**SOURCE CODE**

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

   int i, j;

   for (i = 0; i < n-1; i++)

       // Last i elements are already in place

       for (j = 0; j < n-i-1; j++)

           if (arr[j] > arr[j+1])

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

}

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**INSERTION SORT**

The**insertion sort**, unlike the other sorts, **passes through the array only once.**  The insertion sort is commonly compared to organizing a handful of playing cards.  You pick up the random cards one at a time.  As you pick up each card, you insert it into its correct position in your hand of organized cards.    
  
The insertion sort splits an array into two sub-arrays. The first sub-array (such as the cards in your hand) is sorted and increases in size as the sort continues. The second sub-array (such as the cards to be picked up) is unsorted, contains all the elements to yet be inserted into the first sub-array, and decreases in size as the sort continues.

**SOURCE CODE**

void insertionSort(int arr[], int n)

{

   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;

   }

}

**Time Complexity:**  O(n\*n)

**MERGE SORT**

Like [QuickSort](http://quiz.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/) 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.

**SOURCE CODE**

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);

    }

}

**Time 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) +  *O(NLOG(N))*  
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is*O(NLOG(N))*.  
Time complexity of Merge Sort is *O(NLOG(N))* in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

**HEAP SORT**

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

**What is**[**Binary Heap**](http://geeksquiz.com/binary-heap/)**?**  
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))

A [Binary Heap](http://geeksquiz.com/binary-heap/) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

**Why array based representation for Binary Heap?**  
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).

**Heap Sort Algorithm for sorting in increasing order:**  
**1.** Build a max heap from the input data.  
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
**3.** Repeat above steps while size of heap is greater than 1.

**How to build the heap?**  
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

**SOURCE CODE**

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

{

    int largest = i;  // Initialize largest as root

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

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

    // If left child is larger than root

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

        largest = l;

    // If right child is larger than largest so far

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

        largest = r;

    // If largest is not root

    if (largest != i)

    {

        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree

        heapify(arr, n, largest);

    }

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

    // Build heap (rearrange array)

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

        heapify(arr, n, i);

    // One by one extract an element from heap

    for (int i=n-1; i>=0; i--)

    {

        // Move current root to end

        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

**Time Complexity:**Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

**QUICK SORT**

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted 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.

**SOURCE CODE**

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

{

    int pivot = arr[high];    // pivot

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

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

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

/\* The main function that implements QuickSort

 arr[] --> Array to be sorted,

  low  --> Starting index,

  high  --> Ending index \*/

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

{

    if (low < high)

    {

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

           at right place \*/

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

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

**Time taken** by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

**RADIX SORT**

The [lower bound for Comparison based sorting algorithm](http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/) (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

[Counting sort](http://www.geeksforgeeks.org/counting-sort/) is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

*What if the elements are in range from 1 to n2?*  
We can’t use counting sort because counting sort will take O(n2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?  
[Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

*The Radix Sort Algorithm*  
1) Do following for each digit i where i varies from least significant digit to the most significant digit.  
………….a) Sort input array using counting sort (or any stable sort) according to the i’th digit.

**SOURCE CODE**

int getMax(int arr[], int n)

{

    int mx = arr[0];

    for (int i = 1; i < n; i++)

        if (arr[i] > mx)

            mx = arr[i];

    return mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

void countSort(int arr[], int n, int exp)

{

    int output[n]; // output array

    int i, count[10] = {0};

    // Store count of occurrences in count[]

    for (i = 0; i < n; i++)

        count[ (arr[i]/exp)%10 ]++;

    // Change count[i] so that count[i] now contains actual

    //  position of this digit in output[]

    for (i = 1; i < 10; i++)

        count[i] += count[i - 1];

    // Build the output array

    for (i = n - 1; i >= 0; i--)

    {

        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

        count[ (arr[i]/exp)%10 ]--;

    }

    // Copy the output array to arr[], so that arr[] now

    // contains sorted numbers according to current digit

    for (i = 0; i < n; i++)

        arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using

// Radix Sort

void radixsort(int arr[], int n)

{

    // Find the maximum number to know number of digits

    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead

    // of passing digit number, exp is passed. exp is 10^i

    // where i is current digit number

    for (int exp = 1; m/exp > 0; exp \*= 10)

        countSort(arr, n, exp);

}

***What is the running time of Radix Sort?***  
Let there be d digits in input integers. Radix Sort takes O(d\*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) \* logb(k)). Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let k <= nc where c is a constant. In that case, the complexity becomes O(nLogb(n)). But it still doesn’t beat comparison based sorting algorithms.  
What if we make value of b larger?. What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with range from 1 to nc if the numbers are represented in base n (or every digit takes log2(n) bits).

**BUCKET SORT**

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.   
*Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?*

A simple way is to apply a comparison based sorting algorithm. The [lower bound for Comparison based sorting algorithm](http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/) (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.  
Can we sort the array in linear time? [Counting sort](http://www.geeksforgeeks.org/counting-sort/) can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

**SOURCE CODE**

void bucketSort(float arr[], int n)

{

    // 1) Create n empty buckets

    vector<float> b[n];

    // 2) Put array elements in different buckets

    for (int i=0; i<n; i++)

    {

       int bi = n\*arr[i]; // Index in bucket

       b[bi].push\_back(arr[i]);

    }

    // 3) Sort individual buckets

    for (int i=0; i<n; i++)

       sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]

    int index = 0;

    for (int i = 0; i < n; i++)

        for (int j = 0; j < b[i].size(); j++)

          arr[index++] = b[i][j];

}

**Time Complexity:** If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time as there will be n items in all buckets.  
The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed.

**SHELL SORT**

[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. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element is sorted.

**SOURCE CODE**

int shellSort(int arr[], int n)

{

    // Start with a big gap, then reduce the gap

    for (int gap = n/2; gap > 0; gap /= 2)

    {

        // Do a gapped insertion sort for this gap size.

        // The first gap elements a[0..gap-1] are already in gapped order

        // keep adding one more element until the entire array is

        // gap sorted

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

        {

            // add a[i] to the elements that have been gap sorted

            // save a[i] in temp and make a hole at position i

            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct

            // location for a[i] is found

            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location

            arr[j] = temp;

        }

    }

    return 0;

}

**Time Complexity:** Time complexity of above implementation of shellsort is O(n2). In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity.

**COMB SORT**

Comb Sort is mainly an improvement over Bubble Sort. Bubble sort always compares adjacent values. So all [inversions](http://www.geeksforgeeks.org/counting-inversions/) are removed one by one. Comb Sort improves on Bubble Sort by using gap of size more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus Comb Sort removes more than one [inversion counts](http://www.geeksforgeeks.org/counting-inversions/) with one swap and performs better than Bublle Sort.

The shrink factor has been empirically found to be 1.3 (by testing Combsort on over 200,000 random lists) .

Although, it works better than Bubble Sort on average, worst case remains O(n2).

**SOURCE CODE**

int getNextGap(int gap)

{

    // Shrink gap by Shrink factor

    gap = (gap\*10)/13;

    if (gap < 1)

        return 1;

    return gap;

}

// Function to sort a[0..n-1] using Comb Sort

void combSort(int a[], int n)

{

    // Initialize gap

    int gap = n;

    // Initialize swapped as true to make sure that

    // loop runs

    bool swapped = true;

    // Keep running while gap is more than 1 and last

    // iteration caused a swap

    while (gap != 1 || swapped == true)

    {

        // Find next gap

        gap = getNextGap(gap);

        // Initialize swapped as false so that we can

        // check if swap happened or not

        swapped = false;

        // Compare all elements with current gap

        for (int i=0; i<n-gap; i++)

        {

            if (a[i] > a[i+gap])

            {

                swap(a[i], a[i+gap]);

                swapped = true;

            }

        }

    }

}

**Time Complexity :**Worst case complexity of this algorithm is O(n2) and the Best Case complexity is O(n).

**PIGEONHOLE SORT**

[Pigeonhole sorting](https://en.wikipedia.org/wiki/Pigeonhole_sort) is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same.  
It requires O(*n* + *Range*) time where n is number of elements in input array and ‘Range’ is number of possible values in array.

**Working of Algorithm :**

1. Find minimum and maximum values in array. Let the minimum and maximum values be ‘min’ and ‘max’ respectively. Also find range as ‘max-min-1’.
2. Set up an array of initially empty “pigeonholes” the same size as of the range.
3. Visit each element of the array and then put each element in its pigeonhole. An element arr[i] is put in hole at index arr[i] – min.
4. Start the loop all over the pigeonhole array in order and put the elements from non- empty holes back into the original array.

**SOURCE CODE**

void pigeonholeSort(int arr[], int n)

{

    // Find minimum and maximum values in arr[]

    int min = arr[0], max = arr[0];

    for (int i = 1; i < n; i++)

    {

        if (arr[i] < min)

            min = arr[i];

        if (arr[i] > max)

            max = arr[i];

    }

    int range = max - min + 1; // Find range

    // Create an array of vectors. Size of array

    // range. Each vector represents a hole that

    // is going to contain matching elements.

    vector<int> holes[range];

    // Traverse through input array and put every

    // element in its respective hole

    for (int i = 0; i < n; i++)

        holes[arr[i]-min].push\_back(arr[i]);

    // Traverse through all holes one by one. For

    // every hole, take its elements and put in

    // array.

    int index = 0;  // index in sorted array

    for (int i = 0; i < range; i++)

    {

       vector<int>::iterator it;

       for (it = holes[i].begin(); it != holes[i].end(); ++it)

            arr[index++]  = \*it;

    }

}

**Time Complexity:** O(n+N)

**CYCLE SORT**

Cycle sort is an in-place sorting Algorithm, [unstable sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability), a comparison sort that is theoretically optimal in terms of the total number of writes to the original array.

* It is optimal in terms of number of memory writes. It [minimizes the number of memory writes](http://www.geeksforgeeks.org/which-sorting-algorithm-makes-minimum-number-of-writes/) to sort (Each value is either written zero times, if it’s already in its correct position, or written one time to its correct position.)
* It is based on the idea that array to be sorted can be divided into cycles. Cycles can be visualized as a graph. We have n nodes and an edge directed from node i to node j if the element at i-th index must be present at j-th index in the sorted array.

**SOURCE CODE**

void cycleSort (int arr[], int n)

{

    // count number of memory writes

    int writes = 0;

    // traverse array elements and put it to on

    // the right place

    for (int cycle\_start=0; cycle\_start<=n-2; cycle\_start++)

    {

        // initialize item as starting point

        int item = arr[cycle\_start];

        // Find position where we put the item. We basically

        // count all smaller elements on right side of item.

        int pos = cycle\_start;

        for (int i = cycle\_start+1; i<n; i++)

            if (arr[i] < item)

                pos++;

        // If item is already in correct position

        if (pos == cycle\_start)

            continue;

        // ignore all duplicate  elements

        while (item == arr[pos])

            pos += 1;

        // put the item to it's right position

        if (pos != cycle\_start)

        {

            swap(item, arr[pos]);

            writes++;

        }

        // Rotate rest of the cycle

        while (pos != cycle\_start)

        {

            pos = cycle\_start;

            // Find position where we put the element

            for (int i = cycle\_start+1; i<n; i++)

                if (arr[i] < item)

                    pos += 1;

            // ignore all duplicate  elements

            while (item == arr[pos])

                pos += 1;

            // put the item to it's right position

            if (item != arr[pos])

            {

                swap(item, arr[pos]);

                writes++;

            }

        }

    }

    // Number of memory writes or swaps

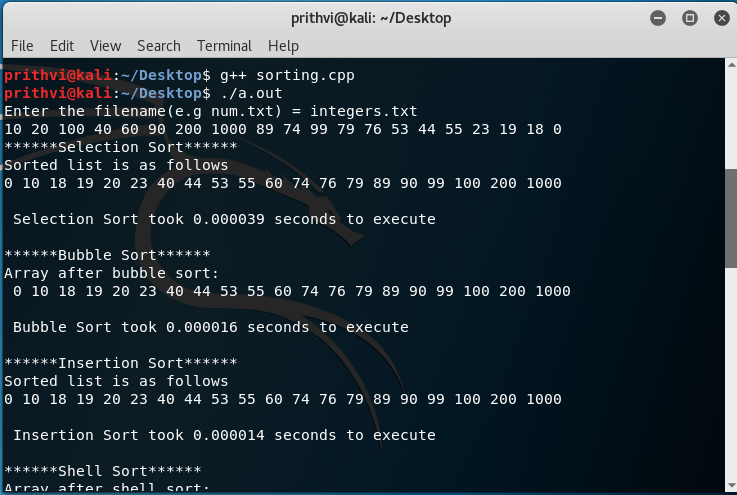
    // cout << writes << endl ;

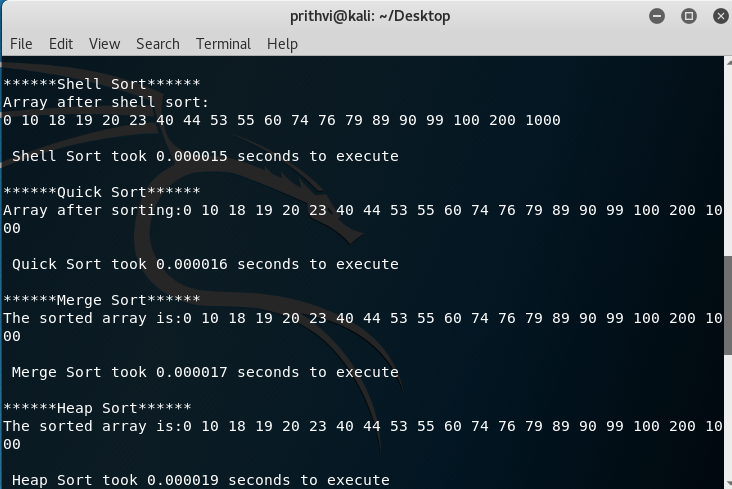
}

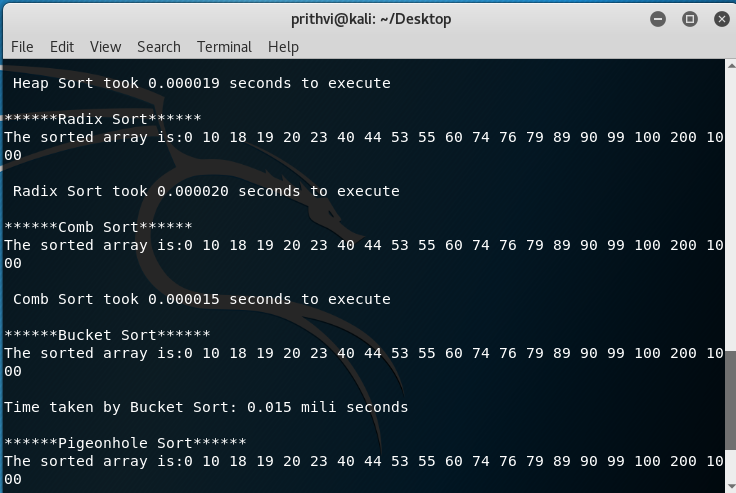
**Time Complexity** : O(n2)  
Worst Case : O(n2)  
Average Case: O(n2)  
Best Case : O(n2)

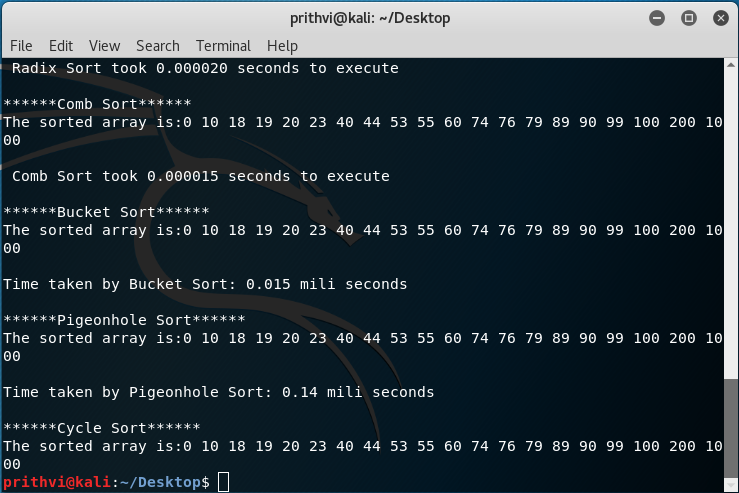
**SORTING ALGORITHMS WITH EXECUTION TIME**

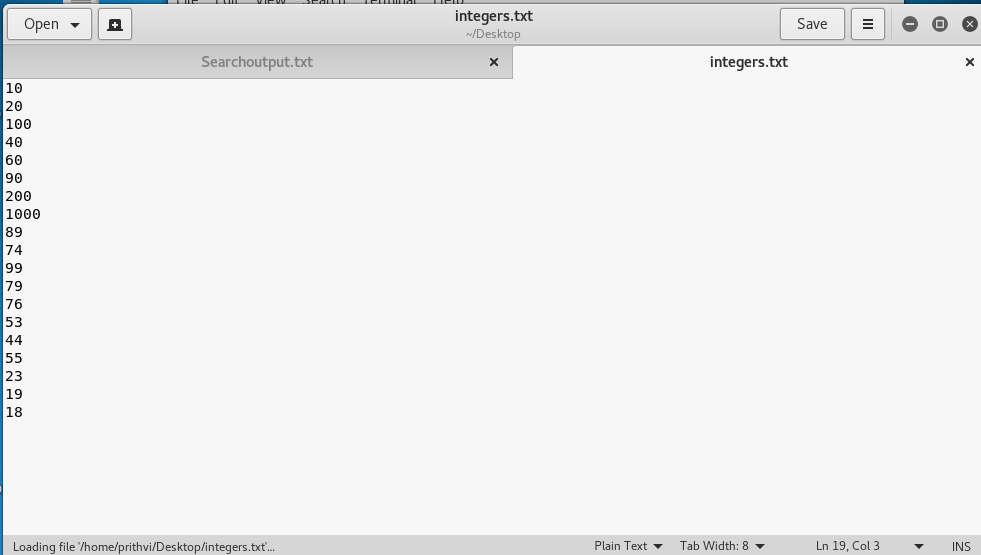
**OUTPUT:**

****

****

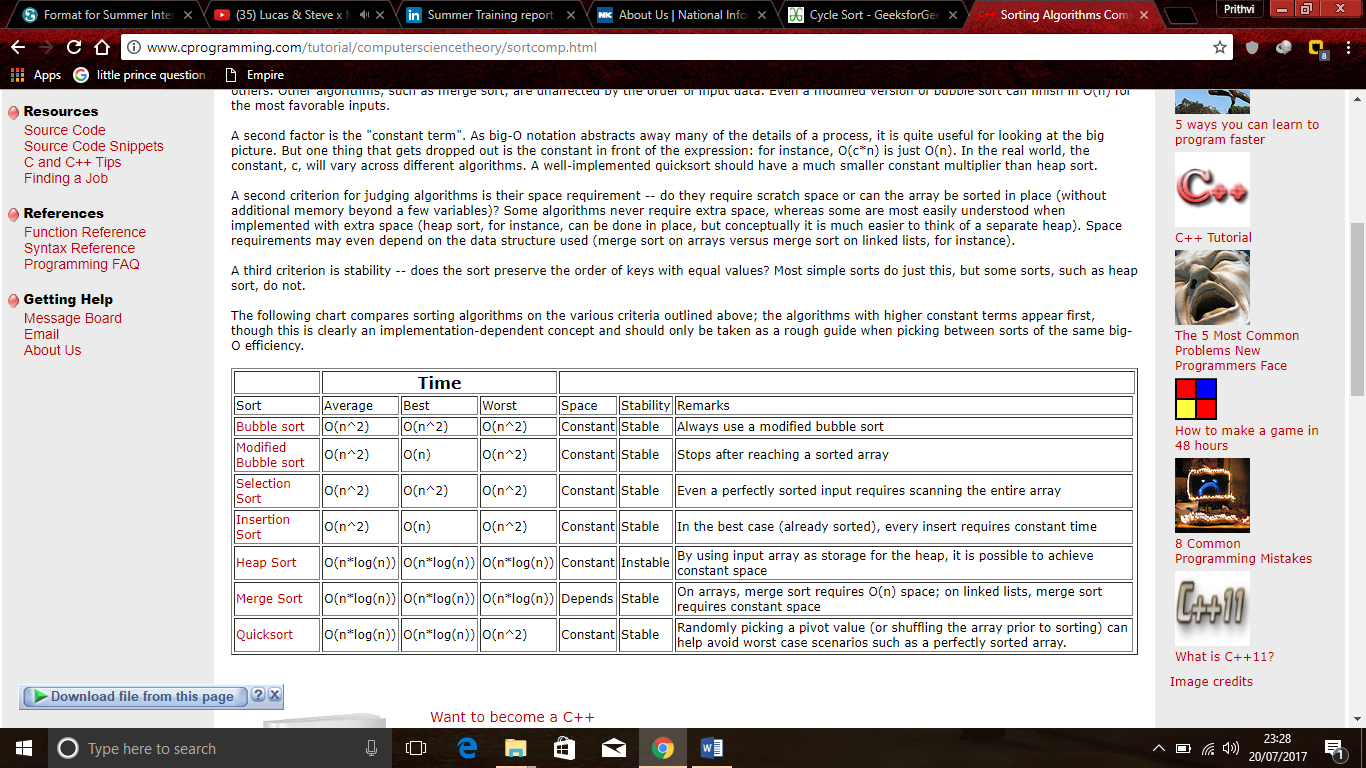






**CONCLUSION**

**Each algorithm has its own benefits in their suitable cases. All possibilities favourable to a certain algorithm is give below: -**



**Fig2. Table for comparison of different sort**

**By using different algorithm, we can speed up our research work. In the program of multiple string searching in multiple file(s) these concepts help in refining our research work and boost up the things for us and result in less execution time.**

**REFERENCES**

<http://www.geeksforgeeks.org/fundamentals-of-algorithms/>

<http://www.cplusplus.com/forum/>

<http://www.cprogramming.com/>

<http://www.sanfoundry.com>

<http://www.thecrazyprogrammer.com>

<https://stackoverflow.com>