2020

PROJECT REPORT

WRITTEN BY-

MAYANK SRIVASTAVA (MS19BQ)

COT5405 | [Company address]

Contents

INTRODUCTION

This is the project report for the course COT5405. This project was written in C++ 11 programming language and was tested successfully in linprog.cs.fsu.edu.

I have implemented all the sorting algorithms in this project, Insertion Sort, Merge Sort, Heap Sort, and Quick Sort. I have also implemented a Binary Search tree(almost all the operations that can be performed, including height and insertion) and Red-Black Tree insertion, deletion, and height.

In the "Implementation" section of this report, Implementations of all the algorithms used in the projects are discussed. The next section consists of "Data representation," where the output achieved from each algorithm is shown and states the initial analysis on the production. In this section, whether the sorting algorithm is in-place or not and Binary_search_tree and Red-Black trees, node representation is discussed. In the last section, "Empirical Evaluation," things like analysis on the output generated from the Sorting algorithms and tree are discussed. Also, calculate the standard deviation and mean for each sorting algorithm on 20 randomly permutated instances of each size n. Here the detailed analysis of the height of the trees generated from the Red-Black and Binary search is also discussed.

INPUT

Sorting Algorithm:

Since it was a requirement of this project to run all the sorting algorithms on 20 randomly permuted instances each of size n, Python programming language is used to implement the input program.

A program that will ask for the user input and generate 20 lines (signifying the instance), and each instance(line) will have n no elements. The Python program is written as such that it will generate random no between 0 to 2^32-1. The result generated is stored in five separate files for each n size. Here in this project, the size of the n = {10,10^2,10^3,10^4,10^5} as requested in the requirements.

Filename used in input	Size of n
sample1.txt	10
sample2.txt	100
sample3.txt	1000
sample4.txt	10000

Sample5.txt	100000

Trees Algorithm:

The python program used here to generate the input file is a modified version of the previously discussed python program. Instead of having 20 instances, the program will generate one sample of size n between 1 to 1000(inclusive).

The filename used as input is: "sample_tree.txt."

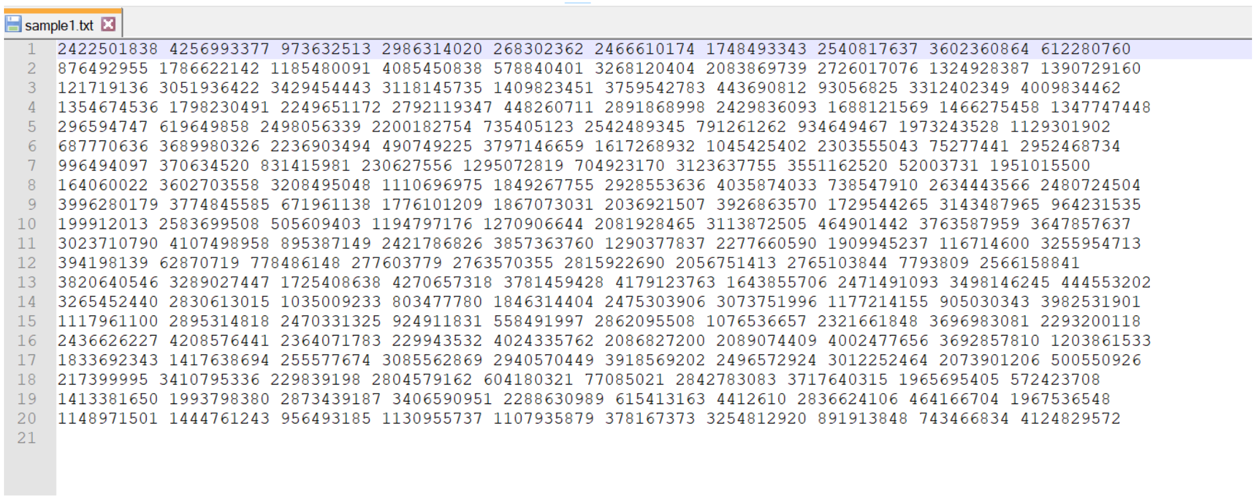
CODE ORGANIZATION AND EXECUTION

- Language used: C++ 11

The project is written in C++ programming language(C++ 11). All the programs are standalone and accepts command line argument as an input. Here the input passed using the command-line argument is the name of the file.

Syntax: ./<Program_executable> <file_name>

Example: ./Heap_Sort sample1.txt

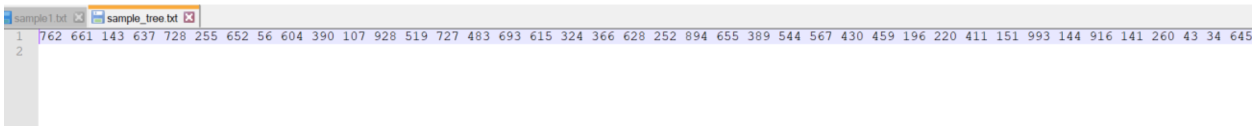


sample1.txt

```
2422501838 4256993377 973632513 2986314020 268302362 2466610174 1748493343 2540817637 3602360864 612280760
876492955 1786622142 1185480091 4085450838 578840401 3268120404 2083869739 2726017076 1324928387 1390729160
121719136 3051936422 3429454443 3118145735 1409823451 3759542783 443690812 93056825 3312402349 4009834462
1354674536 1798230491 2249651172 2792119347 448260711 2891868998 2429836093 1688121569 1466275458 1347747448
296594747 619649858 2498056339 2200182754 735405123 2542489345 791261262 934649467 1973243528 1129301902
687770636 3689980326 2236903494 490749225 3797146659 1617268932 1045425402 2303555043 75277441 2952468734
996494097 370634520 831415981 230627556 1295072819 704923170 3123637755 3551162520 52003731 1951015500
164060022 3602703558 3208495048 1110696975 1849267755 2928553636 4035874033 738547910 2634443566 2480724504
3996280179 3774845585 671961138 1776101209 1867073031 2036921507 3926863570 1729544265 3143487965 964231535
199912013 2583699508 505609403 1194797176 1270906644 2081928465 3113872505 464901442 3763587959 3647857637
3023710790 4107498958 895387149 2421786826 3857363760 1290377837 2277660590 1909945237 116714600 3255954713
394198139 62870719 778486148 277603779 2763570355 2815922690 2056751413 2765103844 7793809 2566158841
3820640546 3289027447 1725408638 4270657318 3781459428 4179123763 1643855706 2471491093 3498146245 444553202
3265452440 2830613015 1035009233 803477780 1846314404 2475303906 3073751996 1177214155 905030343 3982531901
1117961100 2895314818 2470331325 924911831 558491997 2862095508 1076536657 2321661848 3696983081 2293200118
2436626227 4208576441 2364071783 229943532 4024335762 2086827200 2089074409 4002477656 3692857810 1203861533
1833692343 1417638694 255577674 3085562869 2940570449 3918569202 2496572924 3012252464 2073901206 500550926
217399995 3410795336 229839198 2804579162 604180321 77085021 2842783083 3717640315 1965695405 572423708
1413381650 1993798380 2873439187 3406590951 2288630989 615413163 4412610 2836624106 464166704 1967536548
1148971501 1444761243 956493185 1130955737 1107935879 378167373 3254812920 891913848 743466834 4124829572
```

Fig 1: Showing sample1.txt consist of size n=10 (20 instances)

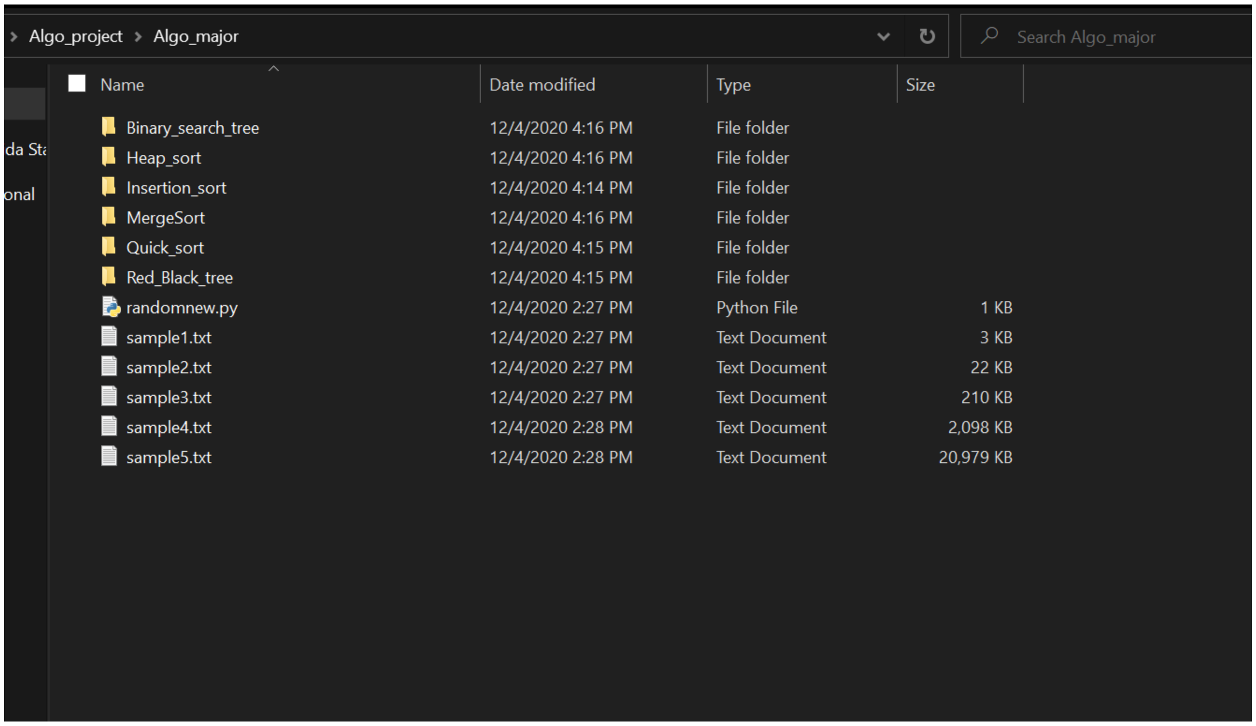
To run the Tree Algorithms(Red-Black or Binary_search), pass "sample_tree.txt" in the command line argument.



```
762 661 143 637 728 255 652 56 604 390 107 928 519 727 483 693 615 324 366 628 252 894 655 389 544 567 430 459 196 220 411 151 993 144 916 141 260 43 34 645
```

Fid: Showing some input of "sample_tree.txt" total elements in the file is 200.

All the programs are present in separate folders to minimize the effect of file overwriting. Since the height, time, and sorting results are stored in the file. It is because printing data to the terminal will result in difficulty to understand.



Name	Date modified	Type	Size
Binary_search_tree	12/4/2020 4:16 PM	File folder	
Heap_sort	12/4/2020 4:16 PM	File folder	
Insertion_sort	12/4/2020 4:14 PM	File folder	
MergeSort	12/4/2020 4:16 PM	File folder	
Quick_sort	12/4/2020 4:15 PM	File folder	
Red_Black_tree	12/4/2020 4:15 PM	File folder	
randomnew.py	12/4/2020 2:27 PM	Python File	1 KB
sample1.txt	12/4/2020 2:27 PM	Text Document	3 KB
sample2.txt	12/4/2020 2:27 PM	Text Document	22 KB
sample3.txt	12/4/2020 2:27 PM	Text Document	210 KB
sample4.txt	12/4/2020 2:28 PM	Text Document	2,098 KB
sample5.txt	12/4/2020 2:28 PM	Text Document	20,979 KB

Fig: showing the file organization

Binary_search_tree folder contains "Binary_search_tree.cpp" and similarly, all other folders contain its ".cpp" files.

Each sorting program will generate two files. 1) "Result_sorted.txt" containing all the sorted requests of the input using the algorithm. 2) "Result_time.txt" storing time in microseconds the algorithm took to run a file.

COMPILATION

Since the program is written in C++ programming language, to compile the code following commands are required.

- Compiler: g++
- Enabling C++ 11: "-std=c++11" (required to run in linprog)

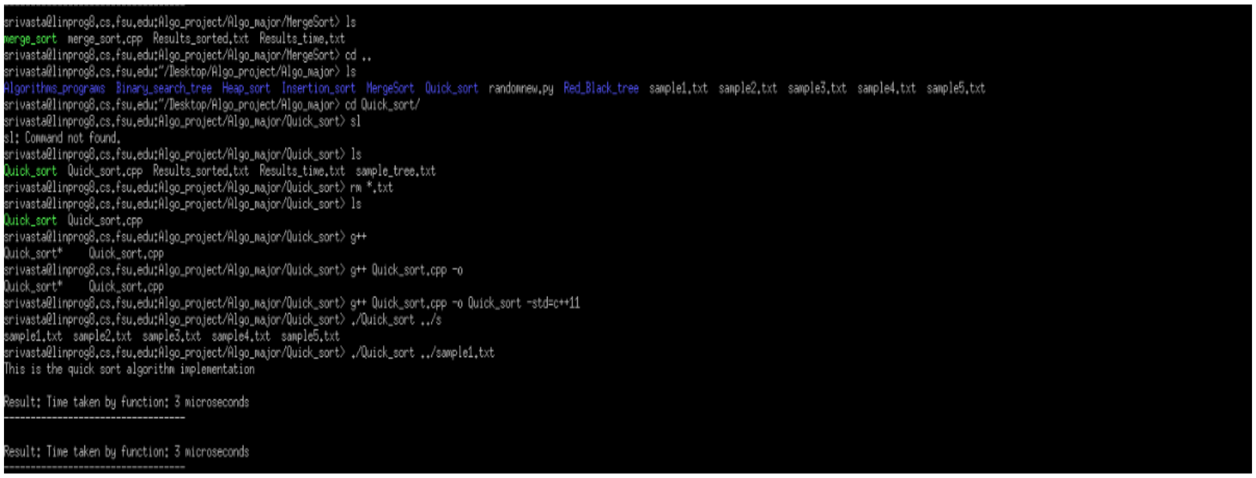
Note: No inbuilt C++ functions are used in this program; the most general cause is using timer(C++11) and file handling operations.

Syntax to compile the program:

```
"g++ <filename.cpp> -o <executable_file_name> -std=c++11
```

To run the program, use syntax:

```
"./<Program_executable>  <file_name>"
```



```
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/MergeSort> ls
merge_sort  merge_sort.cpp  Results_sorted.txt  Results_time.txt
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/MergeSort> cd ..
srivasta@linprog8.cs.fsu.edu:~/Desktop/Algo_project/Algo_major> ls
Algorithms_programs  Binary_search_tree  Heap_sort  Insertion_sort  MergeSort  Quick_sort  randomnew.py  Red_Black_tree  sample1.txt  sample2.txt  sample3.txt  sample4.txt  sample5.txt
srivasta@linprog8.cs.fsu.edu:~/Desktop/Algo_project/Algo_major> cd Quick_sort/
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> sl
sl: Command not found.
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> ls
Quick_sort  Quick_sort.cpp  Results_sorted.txt  Results_time.txt  sample_tree.txt
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> rm *.txt
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> ls
Quick_sort  Quick_sort.cpp
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> g++
Quick_sort*    Quick_sort.cpp
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> g++ Quick_sort.cpp -o
Quick_sort*    Quick_sort.cpp
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> g++ Quick_sort.cpp -o Quick_sort -std=c++11
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> ./Quick_sort ../s
sample1.txt  sample2.txt  sample3.txt  sample4.txt  sample5.txt
srivasta@linprog8.cs.fsu.edu:Algo_project/Algo_major/Quick_sort> ./Quick_sort ../sample1.txt
This is the quick sort algorithm implementation

Result: Time taken by function: 3 microseconds
----------------------------------


Result: Time taken by function: 3 microseconds
----------------------------------
```

Fig Showing the compilation of Quick Sort in linprog.

IMPLEMENTATION

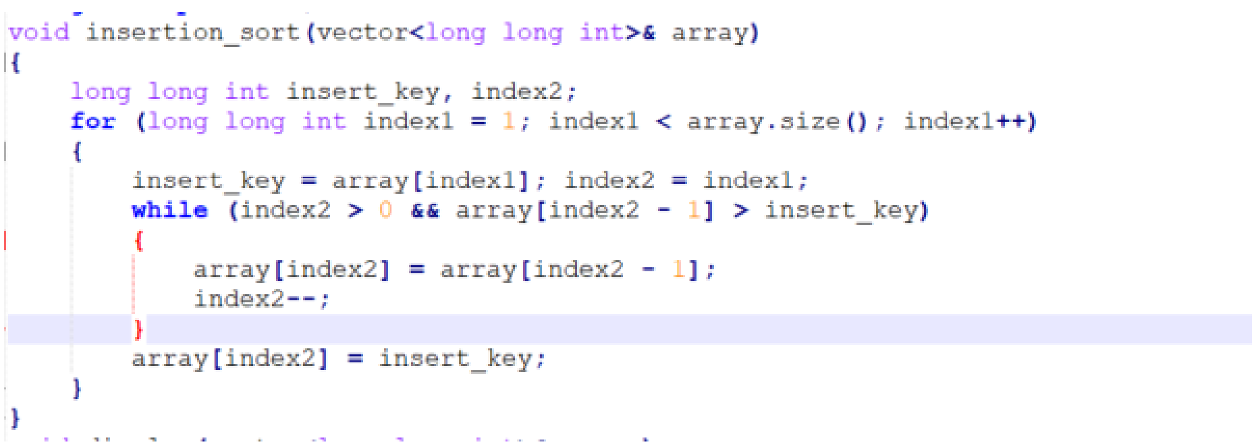
INSERTION SORT

The algorithm is the same as playing a card game. The logic is supposed the array is split into two half; one is sorted, and the other is unsorted. The algorithm's task is to pick elements one by one from the unsorted part and put them in the correct position in the sorted part of the array.

In this Project vector from C++, STL is used to store the elements since the vector is a dynamic array. Here in the program, "long long int" is used since the input is between (0 to 2^32-1). The size of "int" in C++ is 2^31-1.

The algorithm is written in function and will pass the reference of vector to reduce the run time.

Firstly the program will divide the array into two parts; initially, it will start its loop from 1 to do so. And it will swap elements to the right to create the correct position(using while loop in the program). Finally, will insert to the correct position



```
void insertion_sort(vector<long long int>& array)
{
    long long int insert_key, index2;
    for (long long int index1 = 1; index1 < array.size(); index1++)
    {
        insert_key = array[index1]; index2 = index1;
        while (index2 > 0 && array[index2 - 1] > insert_key)
        {
            array[index2] = array[index2 - 1];
            index2--;
        }
        array[index2] = insert_key;
    }
}
```

Fig: Showing insertion sort function

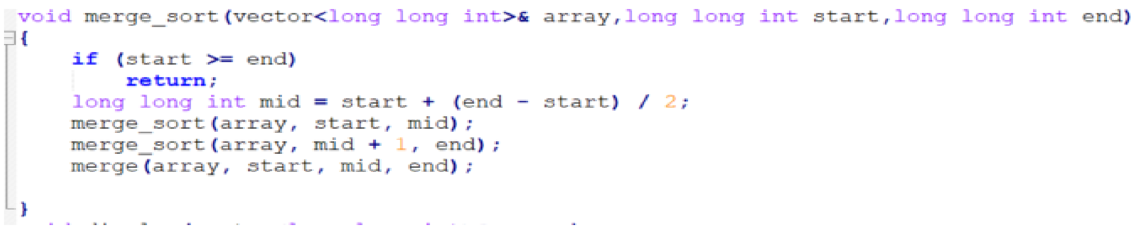
Worst case complexity = O(n^2)

MERGE SORT

This sorting is based on the divide and conquers algorithm approach. Here the input is divided into two half to merge once halves are sorted. The merge sort requires extra space to sort the elements; hence it is not an in-place sorting algorithm.

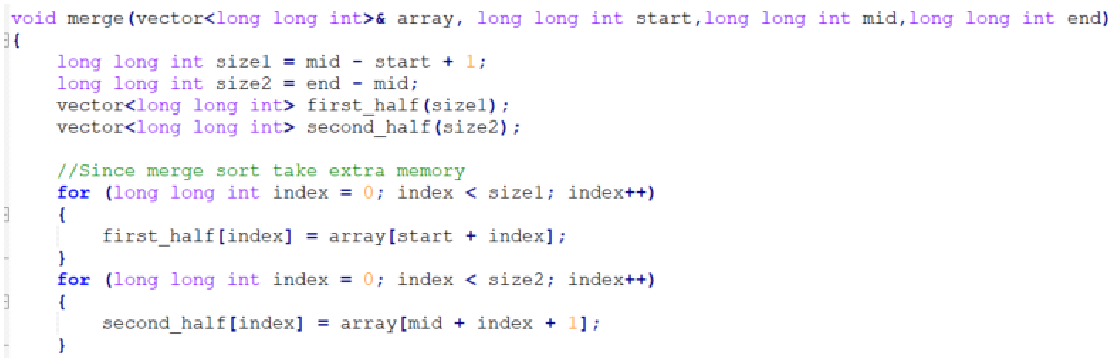
In the project, the task of the "merge_sort" function is to divide the array into two parts and later pass to the "merge" function to merge into an array.

In the "merge" function, first, the two halves are stored in a separate new space, then data is copied from those new arrays to the existing array using comparison.



```
void merge_sort(vector<long long int>& array,long long int start,long long int end)
{
    if (start >= end)
        return;
    long long int mid = start + (end - start) / 2;
    merge_sort(array, start, mid);
    merge_sort(array, mid + 1, end);
    merge(array, start, mid, end);
}
```

Fig: Showing the merge sort function



```
void merge(vector<long long int>& array, long long int start,long long int mid,long long int end)
{
    long long int size1 = mid - start + 1;
    long long int size2 = end - mid;
    vector<long long int> first_half(size1);
    vector<long long int> second_half(size2);

    //Since merge sort take extra memory
    for (long long int index = 0; index < size1; index++)
    {
        first_half[index] = array[start + index];
    }
    for (long long int index = 0; index < size2; index++)
    {
        second_half[index] = array[mid + index + 1];
    }
```

Fig: showing the extra space used in the merge sort algorithm.

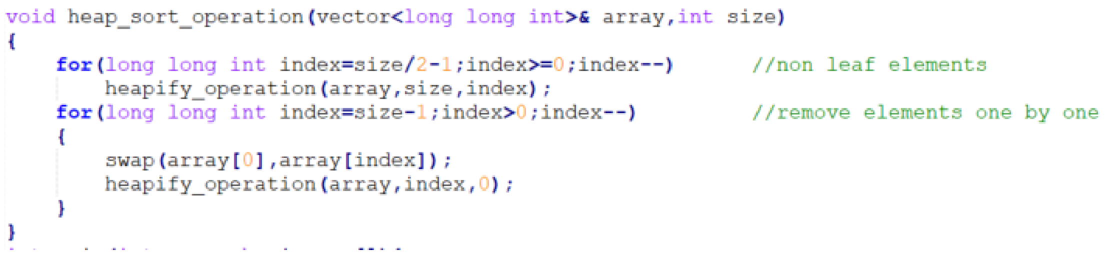
Worst-case Time complexity: O(N log N)

Space complexity: O(n) "Extra space used in the "merge" function.

HEAP SORT

It is a comparison-based algorithm. In this algorithm, a heap is needed to be built. Since elements are sorted in ascending order in this project, "max heap" must be created.

This "max heap" will contain the largest element in its root. This will be replaced with the last element present in a heap, reducing the heap's size by 1. Then "Heapify" the root. This process will be continued till the size of the heap is greater than one.



```
void heap_sort_operation(vector<long long int>& array,int size)
{
    for(long long int index=size/2-1;index>=0;index--)      //non leaf elements
        heapify_operation(array,size,index);
    for(long long int index=size-1;index>0;index--)         //remove elements one by one
    {
        swap(array[0],array[index]);
        heapify_operation(array,index,0);
    }
}
```

Fig: showing the heap sort function

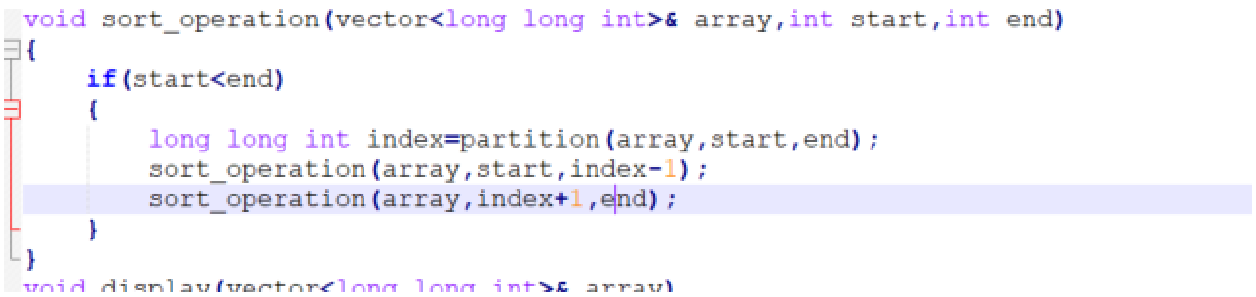
In this function, firstly heap is built, then, one by one, elements are extracted from the heap, and at last, the "heapify_operation" function is called to reduce the size of the heap.

The "heapify_sorting" function will set root as the largest value, find the left and right nodes, check if the left or right child is greater than the root. If found that root is not greater, then swap the values with the greatest.

Worst case complexity= O(N log(N)).

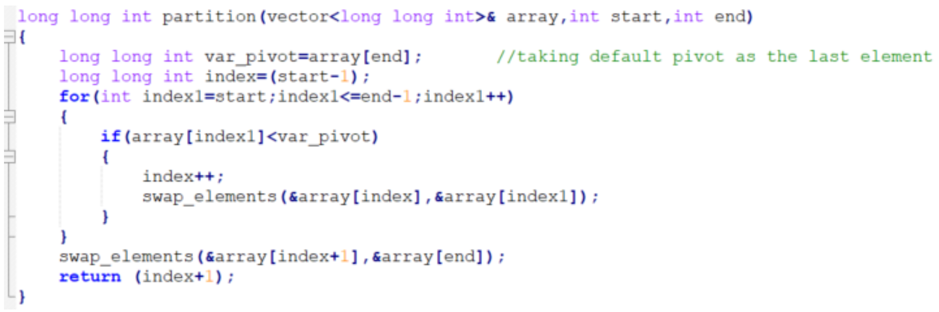
Quick Sort

This algorithm is also based on the divide and conquer approach; in this algorithm, the primary task is to pick the pivot element(the last element is selected in the project). The array is divided based on the pivot. Elements less than the pivot are put before, elements greater than the pivot are put after.



```
void sort_operation(vector<long long int>& array,int start,int end)
{
    if(start<end)
    {
        long long int index=partition(array,start,end);
        sort_operation(array,start,index-1);
        sort_operation(array,index+1,end);
    }
}
void display(vector<long long int>& array)
```

Fig: showing that the Quicksort also used the divide and conquered approach



```
long long int partition(vector<long long int>& array,int start,int end)
{
    long long int var_pivot=array[end];        //taking default pivot as the last element
    long long int index=(start-1);
    for(int index1=start;index1<=end-1;index1++)
    {
        if(array[index1]<var_pivot)
        {
            index++;
            swap_elements(&array[index],&array[index1]);
        }
    }
    swap_elements(&array[index+1],&array[end]);
    return (index+1);
}
```

Fig: partition function

The above function shows that the pivot is selected as the last position in an array. Then pivot is being used for comparison and swapping elements.

Worst time complexity: O(n^2)

Average and Best time complexity: O(n log (n))

BINARY SEARCH TREE

The property of the Binary search tree is

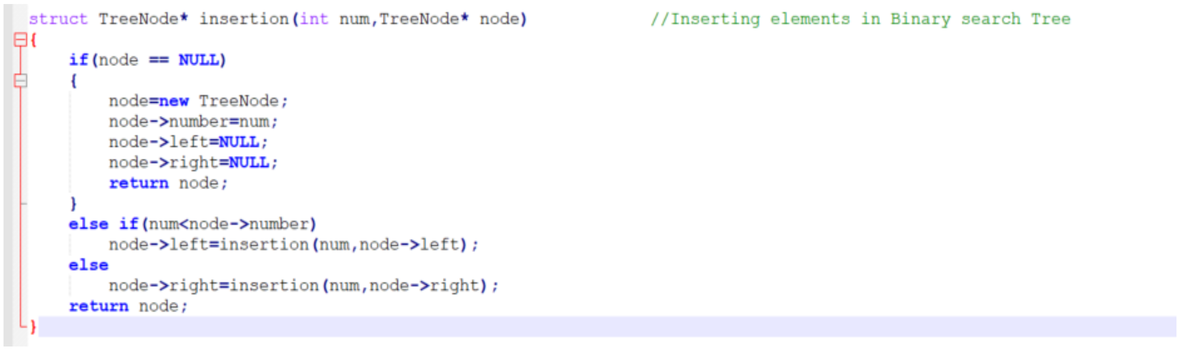
1. Elements in the left subtree are less than the node key.
2. Elements in the right subtree are greater than the node key.
3. The left and right subtree are also BST.
4. There is no duplicate element in the BST.

Following operation are performed in the binary search tree:

- ### INSERTION

  As a recursive function, that first checks if the node is null, if the node is null, there is no element present in the tree, or it reached to leaf.

  As stated, the subtree is also BST; move to the tree's left or right to find a suitable position.



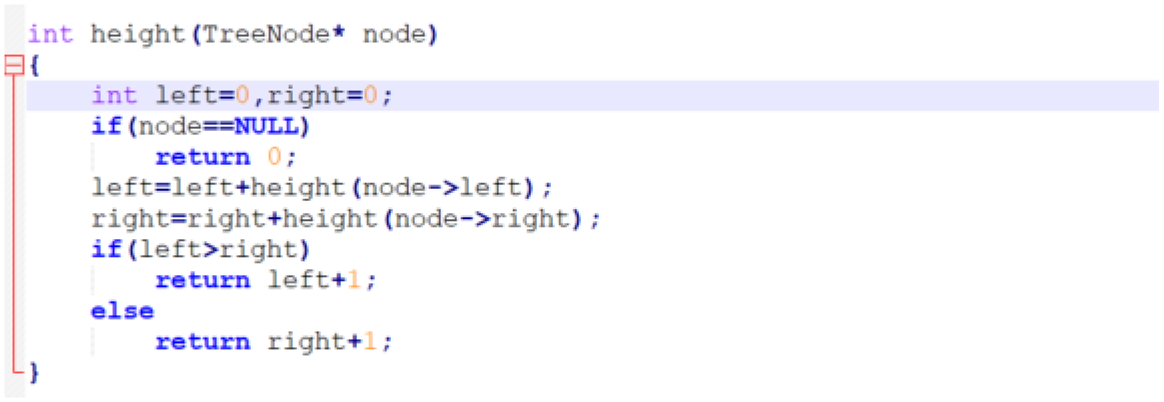
```
struct TreeNode* insertion(int num,TreeNode* node)            //Inserting elements in Binary search Tree
{
    if(node == NULL)
    {
        node=new TreeNode;
        node->number=num;
        node->left=NULL;
        node->right=NULL;
        return node;
    }
    else if(num<node->number)
        node->left=insertion(num,node->left);
    else
        node->right=insertion(num,node->right);
    return node;
}
```

Fig: showing BST insertion

Here node is the root of the tree, and "num" is the value to be inserted. If the root is null, it directly creates a node; otherwise, it moves to the BST left or right using node(root).

- HEIGHT

  This function will use two variables to calculate the left and right sub-tree height as a recursive function. The logic of calculating the height is adding 1 to the max variable(left or right) to get the tree's complete height.



```
int height(TreeNode* node)
{
    int left=0,right=0;
    if(node==NULL)
        return 0;
    left=left+height(node->left);
    right=right+height(node->right);
    if(left>right)
        return left+1;
    else
        return right+1;
}
```

Fig: showing the height function

  This function will take root as input and recursively reach the left and right subtree's maximum depth. Then get the maximum depth of left and right and add one to it.

NOTE: Along with this, Inorder traversal, postorder traversal, preorder traversal, finding maximum and minimum element in BST, searching, node deletion, and level order traversal program are also written in a program in the form of functions.

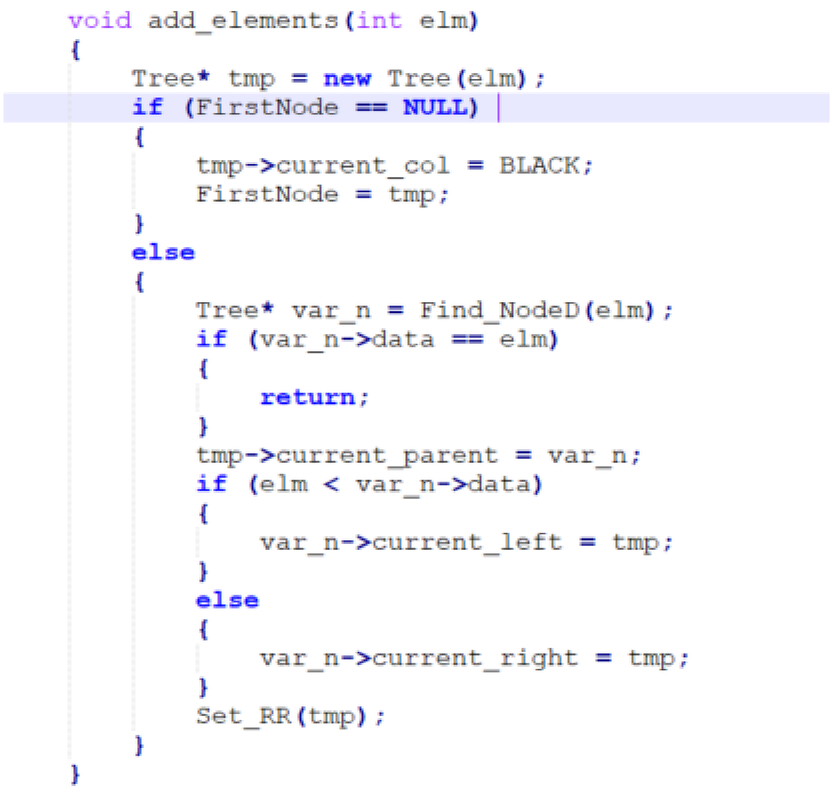
RED-BLACK TREE

- INSERTION

  To insert the element in the RBT(RED-BLACK Tree). The insertion has to be performed like BST(binary search tree), but here node also has a color that should be RED.

If the tree is null, then the color of the node will be BLACK. If the tree is not empty, then check the color of the parent node. If the color is black, then there is no need to change anything else; match the uncle node's color and if it is RED, then perform recoloring. If the uncle code color is black, then you have to do the rotation. LL, LR, RR, RL (L is left and R is right).

Below is the insert "add_element" function, where the value is passed as "elm." First, it is checked if the tree is null, then set the color to black and create a tree. Otherwise, use BST property and set to RR and check uncle color property.



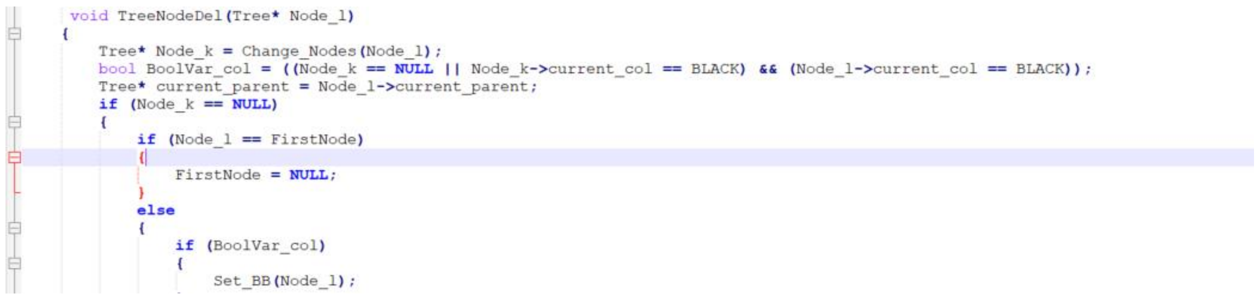
```
void add_elements(int elm)
{
    Tree* tmp = new Tree(elm);
    if (FirstNode == NULL)
    {
        tmp->current_col = BLACK;
        FirstNode = tmp;
    }
    else
    {
        Tree* var_n = Find_NodeD(elm);
        if (var_n->data == elm)
        {
            return;
        }
        tmp->current_parent = var_n;
        if (elm < var_n->data)
        {
            var_n->current_left = tmp;
        }
        else
        {
            var_n->current_right = tmp;
        }
        Set_RR(tmp);
    }
}
```

Fig: Insert Function

- DELETION
  In deletion, sibling color has to be checked to perform the operations.

1. Do Binary search tree deletion
2. If either node x1 or x2 is black, then the replaced child has to be marked as black.

3. If both the nodes x1 and x2 are black, recoloring has to be done before deletion.
4. If one of the sibling colors is red, then rotation has to be performed. There are multiple cases based on the condition like (left-left case, right-right case) depend on that coloring has to be done.



```
void TreeNodeDel(Tree* Node_l)
{
    Tree* Node_k = Change_Nodes(Node_l);
    bool BoolVar_col = ((Node_k == NULL || Node_k->current_col == BLACK) && (Node_l->current_col == BLACK));
    Tree* current_parent = Node_l->current_parent;
    if (Node_k == NULL)
    {
        if (Node_l == FirstNode)
        {
            FirstNode = NULL;
        }
        else
        {
            if (BoolVar_col)
            {
                Set_BB(Node_l);
```

Fig: showing the initial deletion function.

- HEIGHT
  This function is the same as discussed in the Binary search tree section

DATA REPRESENTATION

The file input used is already discussed in the "INPUT" section of this report. This project was created using C++ 11 programming language, using STL<vector as data structure>. Using a vector is because of its size increases and decreases based on the current input (dynamic). Not like an array where size is fixed and will result in the wastage or less allocated memory.

SORTING

Vector is also used in the sorting algorithms to store the data from the input and perform the sorting operation. Every sorting algorithm has different strategies; some used extra memory, and some not. Based on these concepts, we call the "in-place" or "not-in place" sorting algorithm.

- Insertion sort: "In place."
- Heap Sort: "In-place"

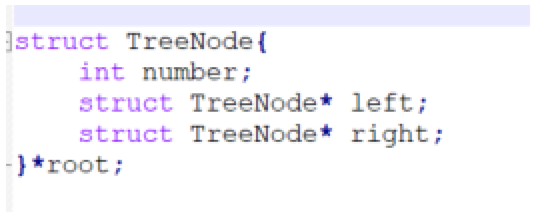
- Merge Sort: "not in place," as discussed in the merge sort section, requires extra memory to sort elements O(n).
- Quick Sort: "In-place" since extra space used is for the recursive call.

Tree

To create a node in the tree, both struct and class are used, as C++ is an object-oriented programming language.

- ### BINARY SEARCH

  Contain "int number" to store the number(data) and two pointers to point to the left and right child.



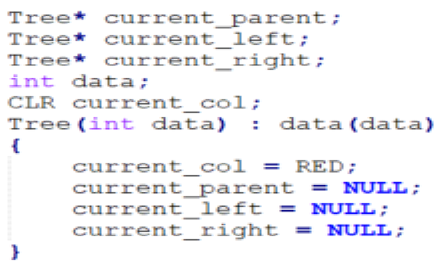
```
struct TreeNode{
    int number;
    struct TreeNode* left;
    struct TreeNode* right;
}*root;
```

Fig: showing BST node structure

- ### RED-BLACK TREE

  Nodes structure in the RBT is consist of "current_parent," current_left," "current_right" pointers to store to point the parent, left, and right child of the current node.

  Also, a "current_col" variable stores the color of the node RED or BLACK (set using enum). At the last variable, "data" stores the data(number) of the node.



```
Tree* current_parent;
Tree* current_left;
Tree* current_right;
int data;
CLR current_col;
Tree(int data) : data(data)
{
    current_col = RED;
    current_parent = NULL;
    current_left = NULL;
    current_right = NULL;
}
```

Fig: showing the structure of RBT and also the initial data.

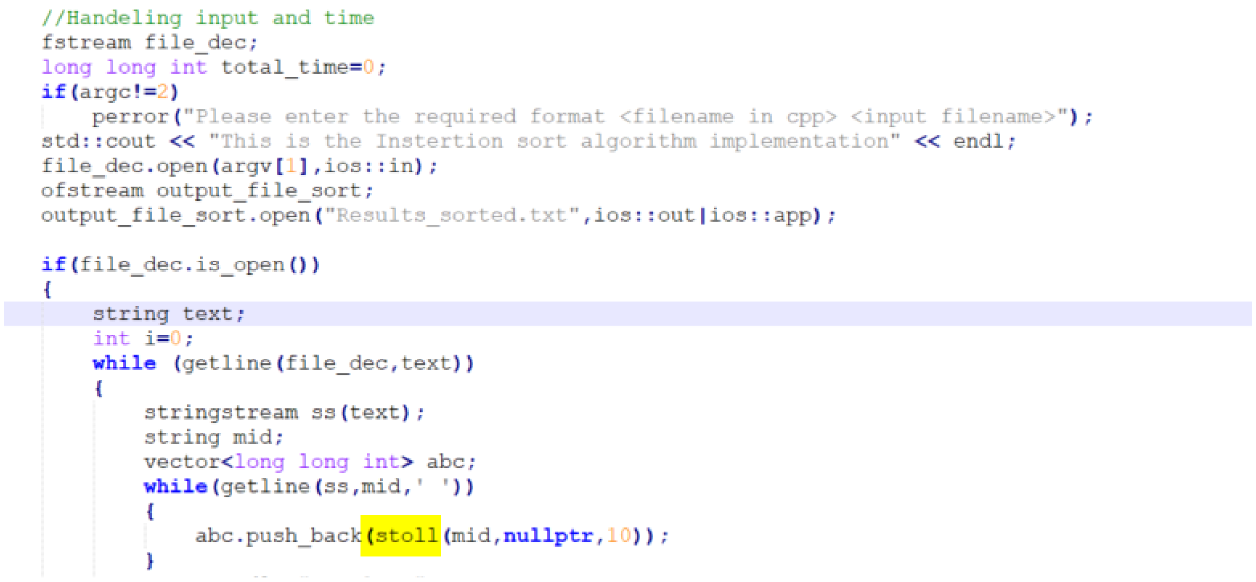
UTILITY PROCEDURES

FILE HANDLING

C++ file handling operations are used to make more clean and elegant code. Using <sstream>, <fstream>, <string> libraries to successfully fetch data from the input file.

The logic is as the input data is present in the string format, so first, open the file and read it using "stoll" function to convert string to "long long int." This way, our input data will be in "long long int" format required in all the sorting programs.

Similarly, in the tree algorithm, the same operations are done to get the data in integer format and store it into a vector.



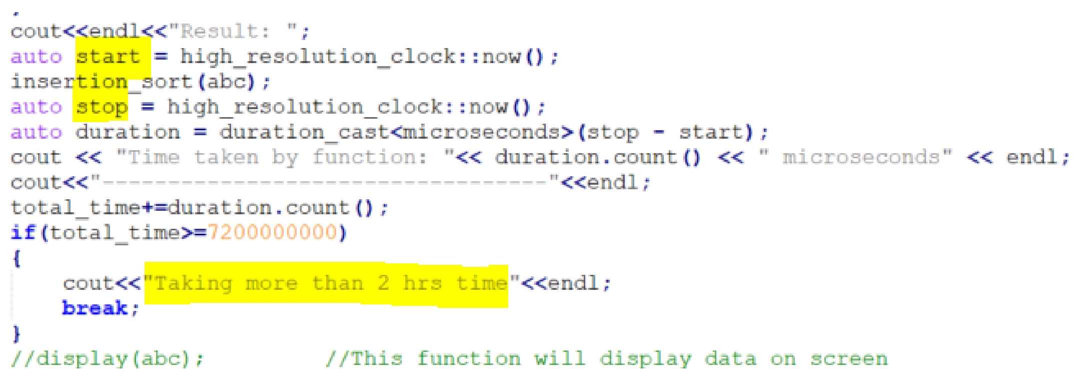
```
//Handeling input and time
fstream file_dec;
long long int total_time=0;
if(argc!=2)
    perror("Please enter the required format <filename in cpp> <input filename>");
std::cout << "This is the Instertion sort algorithm implementation" << endl;
file_dec.open(argv[1],ios::in);
ofstream output_file_sort;
output_file_sort.open("Results_sorted.txt",ios::out|ios::app);

if(file_dec.is_open())
{
    string text;
    int i=0;
    while (getline(file_dec,text))
    {
        stringstream ss(text);
        string mid;
        vector<long long int> abc;
        while(getline(ss,mid,' '))
        {
            abc.push_back(stoll(mid,nullptr,10));
        }
```

Fig: showing the use of file handling to sore data in vector

TIMER

The time required to run the sorting operation is done using the C++ <chrono> library. This will first store the starting time(just before the sorting) and end time(soon when sorting is done) for all instances. The final result is added together to know the algorithm's full time to operate on size n input of 20 instances in microseconds. Also, if the algorithm takes more than 2 hours to work, the condition is implemented.



```
cout<<endl<<"Result: ";
auto start = high_resolution_clock::now();
insertion_sort(abc);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;
cout<<"----------------------------------"<<endl;
total_time+=duration.count();
if(total_time>=7200000000)
{
    cout<<"Taking more than 2 hrs time"<<endl;
    break;
}
//display(abc);         //This function will display data on screen
```

Fig: showing the timer start, end, and 2 hours condition to break

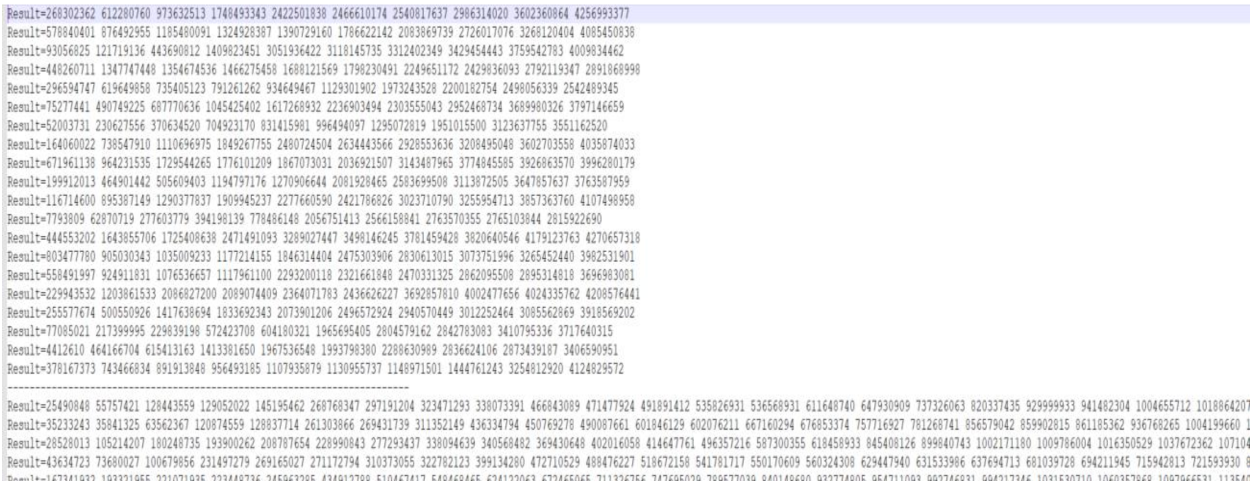
EMPIRICAL EVALUATION

SORTING

As stated, the sorting algorithm was tested and analyzed on the input of size n ={10, 100,1000,10000,100000}, each input is between 0 to 2^32-1, and operation is done on the 20 randomly permutated input of size n.

INSERTION SORT

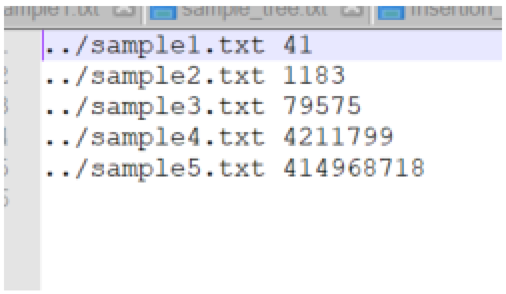
The below image shows the correctness of the algorithm, sorted in the correct order. The picture shows all sorted results of size ten and some of size 100.



```
Result=268302362 612280760 973632513 1748493343 2422501838 2466610174 2540817637 2986314020 3602360864 4256993377
Result=578840401 876492955 1185480091 1324928387 1390729160 1786622142 2083869739 2726017076 3268120404 4085450838
Result=93056825 121719136 443690812 1409823451 3051936422 3118145735 3312402349 3429454443 3759542783 4009834462
Result=448260711 1347747448 1354674536 1466275458 1688121569 1798230491 2249651172 2429836093 2792119347 2891868998
Result=296594747 619649858 735405123 791261262 934649467 1129301902 1973243528 2200182754 2498056339 2542489345
Result=75277441 490749225 687770636 1045425402 1617268932 2236903494 2303555043 2952468734 3689980326 3797146659
Result=52003731 230627556 370634520 704923170 831415981 996494097 1295072819 1951015500 3123637755 3551162520
Result=164060022 738547910 1110696975 1849267755 2480724504 2634443566 2928553636 3208495048 3602703558 4035874033
Result=671961138 964231535 1729544265 1776101209 1867073031 2036921507 3143487965 3774845585 3926863570 3996280179
Result=199912013 464901442 505609403 1194797176 1270906644 2081928465 2583699508 3113872505 3647857637 3763587959
Result=116714600 895387149 1290377837 1909945237 2277660590 2421786826 3023710790 3255954713 3857363760 4107498958
Result=7793809 62870719 277603779 394198139 778486148 2056751413 2566158841 2763570355 2765103844 2815922690
Result=444553202 1643855706 1725408638 2471491093 3289027447 3498146245 3781459428 3820640546 4179123763 4270657318
Result=803477780 905030343 1035009233 1177214155 1846314404 2475303906 2830613015 3073751996 3265452440 3982531901
Result=558491997 924911831 1076536657 1117961100 2293200118 2321661848 2470331325 2862095508 2895314818 3696983081
Result=229943532 1203861533 2086827200 2089074409 2364071783 2436626227 3692857810 4002477656 4024335762 4208576441
Result=255577674 500550926 1417638694 1833692343 2073901206 2496572924 2940570449 3012252464 3085562869 3918569202
Result=77085021 217399995 229839198 572423708 604180321 1965695405 2804579162 2842783083 3410795336 3717640315
Result=4412610 464166704 615413163 1413381650 1967536548 1993798380 2288630989 2836624106 2873439187 3406590951
Result=378167373 743466834 891913848 956493185 1107935879 1130955737 1148971501 1444761243 3254812920 4124829572
-------------------------------------------------------------------------
Result=25490848 55757421 128443559 129052022 145195462 268768347 297191204 323471293 338073391 466843089 471477924 491891412 535826931 536568931 611648740 647930909 737326063 820337435 929999933 941482304 1004655712 1018864207
Result=35233243 35841325 63562367 120874559 128837714 261303866 269431739 311352149 436334794 450769278 490087661 601846129 602076211 667160294 676853374 757716927 781268741 856579042 859902815 861185362 936768265 1004199660 1
Result=28528013 105214207 180248735 193900262 208787654 228990843 277293437 338094639 340568482 369430648 402016058 414647761 496357216 587300355 618458933 845408126 899840743 1002171180 1009786004 1016350529 1037672362 107104
Result=43634723 73680027 100679856 231497279 269165027 271172794 310373055 322782123 399134280 472710529 488476227 518672158 541781717 550170609 560324308 629447940 631533986 637694713 681039728 694211945 715942813 721593930 8
```

Fig: showing the correctness of the algorithm(sorted in the correct order)

On running through all the input of size n required. The below image shows the result generated in a microsecond(time taken by an algorithm to perform the sorting operation).



```
../sample1.txt 41
../sample2.txt 1183
../sample3.txt 79575
../sample4.txt 4211799
../sample5.txt 414968718
```

Fig: Picture from the insertion sort result generated from linprog.

Filename used in input	Size of n
sample1.txt	10
sample2.txt	100
sample3.txt	1000
sample4.txt	10000
Sample5.txt	100000

Following the same table, match the time based on the input of size n.

The average time is taken for 20 randomly permutated instance of each size n:

- N=10, (41/20)= 2.05 microseconds
- N=100, ((1183-41)/20)= 57.1 microseconds

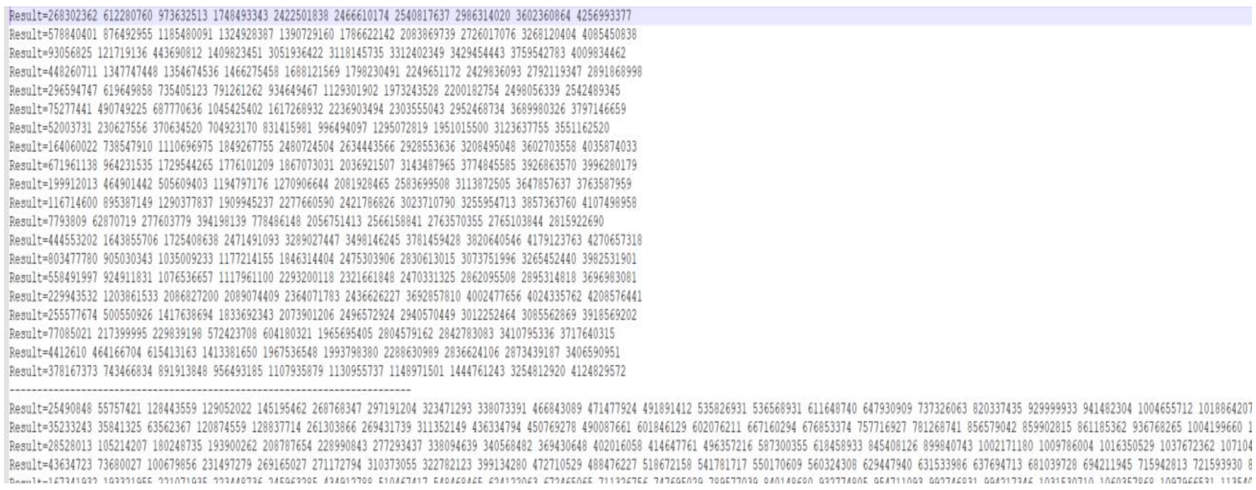
- N=1000, ((79575-1183)/20)= 3919.6 microseconds
- N=10000, ((4211799-79575)/20)= 206611.2 microseconds
- N=100000, ((414968718-4211799)/20)=20537845.95 microseconds

The time in microseconds is calculated (added) in the program to know the overall total time.

The standard deviation on all inputs of size n: 6.63 sec(approx.)

MERGE SORT

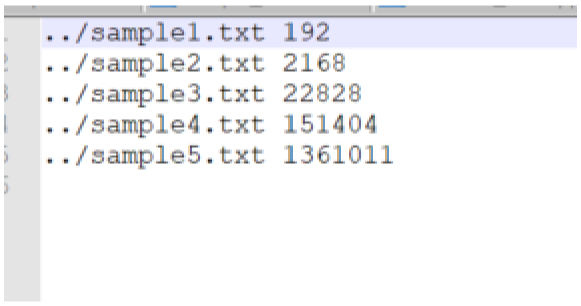
The below image shows the correctness of the algorithm, sorted in the correct order. The picture shows all sorted results of size ten and some of size 100.



```
Result=268302362 612280760 973632513 1748493343 2422501838 2466610174 2540817637 2986314020 3602360864 4256993377
Result=578840401 876492955 1185480091 1324928387 1390729160 1786622142 2083869739 2726017076 3268120404 4085450838
Result=93056825 121719136 443690812 1409823451 3051936422 3118145735 3312402349 3429454443 3759542783 4009834462
Result=448260711 1347747448 1354674536 1466275458 1688121569 1798230491 2249651172 2429836093 2792119347 2891868998
Result=296594747 619649858 735405123 791261262 934649467 1129301902 1973243528 2200182754 2498056339 2542489345
Result=75277441 490749225 687770636 1045425402 1617268932 2236903494 2303555043 2952468734 3689980326 3797146659
Result=52003731 230627556 370634520 704923170 831415981 996494097 1295072819 1951015500 3123637755 3551162520
Result=164060022 738547910 1110696975 1849267755 2480724504 2634443566 2928553636 3208495048 3602703558 4035874033
Result=671961138 964231535 1729544265 1776101209 1867073031 2036921507 3143487965 3774845585 3926863570 3996280179
Result=199912013 464901442 505609403 1194797176 1270906644 2081928465 2583699508 3113872505 3647857637 3763587959
Result=116714600 895387149 1290377837 1909945237 2277660590 2421786826 3023710790 3255954713 3857363760 4107498958
Result=7793809 62870719 277603779 394198139 778486148 2056751413 2566158841 2763570355 2765103844 2815922690
Result=444553202 1643855706 1725408638 2471491093 3289027447 3498146245 3781459428 3820640546 4179123763 4270657318
Result=803477780 905030343 1035009233 1177214155 1846314404 2475303906 2830613015 3073751996 3265452440 3982531901
Result=558491997 924911831 1076536657 1117961100 2293200118 2321661848 2470331325 2862095508 2895314818 3696983081
Result=229943532 1203861533 2086827200 2089074409 2364071783 2436626227 3692857810 4002477656 4024335762 4208576441
Result=255577674 500550926 1417638694 1833692343 2073901206 2496572924 2940570449 3012252464 3085562869 3918569202
Result=77085021 217399995 229839198 572423708 604180321 1965695405 2804579162 2842783083 3410795336 3717640315
Result=4412610 464166704 615413163 1413381650 1967536548 1993798380 2288630989 2836624106 2873439187 3406590951
Result=378167373 743466834 891913848 956493185 1107935879 1130955737 1148971501 1444761243 3254812920 4124829572
--------------------------------------------------------------------------
Result=25490848 55757421 128443559 129052022 145195462 268768347 297191204 323471293 338073391 466843089 471477924 491891412 535826931 536568931 611648740 647930909 737326063 820337435 929999933 941482304 1004655712 1018864207
Result=35233243 35841325 63562367 120874559 128837714 261303866 269431739 311352149 436334794 450769278 490087661 601846129 602076211 667160294 676853374 757716927 781268741 856579042 859902815 861185362 936768265 1004199660 1
Result=28528013 105214207 180248735 193900262 208787654 228990843 277293437 338094639 340568482 369430648 402016058 414647761 496357216 587300355 618458933 845408126 899840743 1002171180 1009786004 1016350529 1037672362 107104
Result=43634723 73680027 100679856 231497279 269165027 271172794 310373055 322782123 399134280 472710529 488476227 518672158 541781717 550170609 560324308 629447940 631533986 637694713 681039728 694211945 715942813 721593930 8
Result=167341932 193321955 221071935 223448736 245963285 434912788 510467417 548468465 624122063 672465065 711326756 747695029 789577039 840148680 932774805 954711093 992746831 994217346 1031530710 1060357868 1097966531 11354
```

Fig: showing the correctness of the algorithm(sorted in the correct order)

On running through all the input of size n required. The below image shows the result generated in a microsecond(time taken by an algorithm to perform the sorting operation).



```
../sample1.txt 192
../sample2.txt 2168
../sample3.txt 22828
../sample4.txt 151404
../sample5.txt 1361011
```

Fig: Picture from the merge sort result generated from linprog.

The average time is taken for 20 randomly permutated instance of each size n:

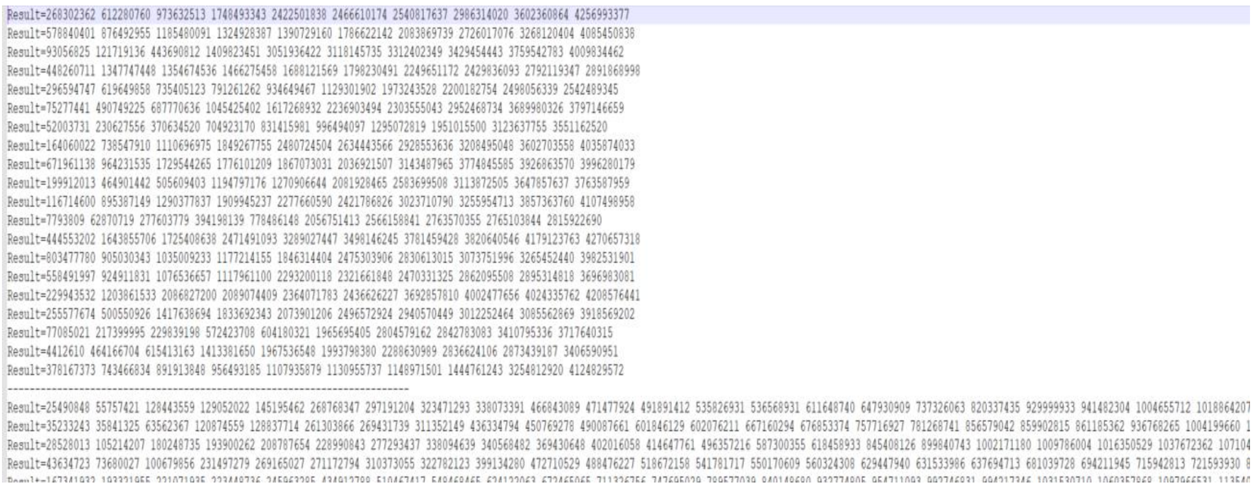
- N=10, (192/20)= 9.6 microseconds
- N=100, ((2168-192)/20)= 98.8 microseconds
- N=1000, ((22828-2168)/20)= 1034.5 microseconds
- N=10000, ((151404-22828)/20)= 6428.8 microseconds
- N=100000, ((1361011-151504)/20)=60475.35 microseconds

The time in microseconds is calculated (added) in the program to know the overall total time.

The standard deviation on all inputs of size n: 0.77 sec(approx.)

HEAP SORT

The below image shows the correctness of the algorithm, sorted in the correct order. The picture shows all sorted results of size ten and some of size 100.

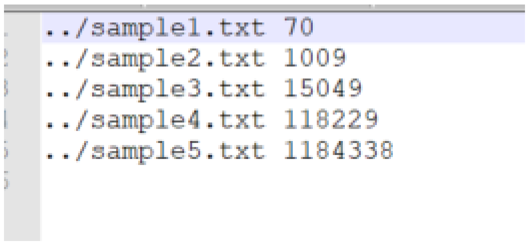


```
Result=268302362 612280760 973632513 1748493343 2422501838 2466610174 2540817637 2986314020 3602360864 4256993377
Result=578840401 876492955 1185480091 1324928387 1390729160 1786622142 2083869739 2726017076 3268120404 4085450838
Result=93056825 121719136 443690812 1409823451 3051936422 3118145735 3312402349 3429454443 3759542783 4009834462
Result=448260711 1347747448 1354674536 1466275458 1688121569 1798230491 2249651172 2429836093 2792119347 2891868998
Result=296594747 619649858 735405123 791261262 934649467 1129301902 1973243528 2200182754 2498056339 2542489345
Result=75277441 490749225 687770636 1045425402 1617268932 2236903494 2303555043 2952468734 3689980326 3797146659
Result=52003731 230627556 370634520 704923170 831415981 996494097 1295072819 1951015500 3123637755 3551162520
Result=164060022 738547910 1110696975 1849267755 2480724504 2634443566 2928553636 3208495048 3602703558 4035874033
Result=671961138 964231535 1729544265 1776101209 1867073031 2036921507 3143487965 3774845585 3926863570 3996280179
Result=199912013 464901442 505609403 1194797176 1270906644 2081928465 2583699508 3113872505 3647857637 3763587959
Result=116714600 895387149 1290377837 1909945237 2277660590 2421786826 3023710790 3255954713 3857363760 4107498958
Result=7793809 62870719 277603779 394198139 778486148 2056751413 2566158841 2763570355 2765103844 2815922690
Result=444553202 1643855706 1725408638 2471491093 3289027447 3498146245 3781459428 3820640546 4179123763 4270657318
Result=803477780 905030343 1035009233 1177214155 1846314404 2475303906 2830613015 3073751996 3265452440 3982531901
Result=558491997 924911831 1076536657 1117961100 2293200118 2321661848 2470331325 2862095508 2895314818 3696983081
Result=229943532 1203861533 2086827200 2089074409 2364071783 2436626227 3692857810 4002477656 4024335762 4208576441
Result=255577674 500550926 1417638694 1833692343 2073901206 2496572924 2940570449 3012252464 3085562869 3918569202
Result=77085021 217399995 229839198 572423708 604180321 1965695405 2804579162 2842783083 3410795336 3717640315
Result=4412610 464166704 615413163 1413381650 1967536548 1993798380 2288630989 2836624106 2873439187 3406590951
Result=378167373 743466834 891913848 956493185 1107935879 1130955737 1148971501 1444761243 3254812920 4124829572
--------------------------------------------------------------------------

Result=25490848 55757421 128443559 129052022 145195462 268768347 297191204 323471293 338073391 466843089 471477924 491891412 535826931 536568931 611648740 647930909 737326063 820337435 929999933 941482304 1004655712 1018864207
Result=35233243 35841325 63562367 120874559 128837714 261303866 269431739 311352149 436334794 450769278 490087661 601846129 602076211 667160294 676853374 757716927 781268741 856579042 859902815 861185362 936768265 1004199660 1
Result=28528013 105214207 180248735 193900262 208787654 228990843 277293437 338094639 340568482 369430648 402016058 414647761 496357216 587300355 618458933 845408126 899840743 1002171180 1009786004 1016350529 1037672362 107104
Result=43634723 73680027 100679856 231497279 269165027 271172794 310373055 322782123 399134280 472710529 488476227 518672158 541781717 550170609 560324308 629447940 631533986 637694713 681039728 694211945 715942813 721593930 8
Result=167341932 193321955 221071935 223448736 245963285 434912788 510467417 548468465 624122063 672465065 711326756 747695029 789577039 840148680 932774805 954711093 992746831 994217346 1031530710 1060357868 1097966531 11354
```

Fig: showing the correctness of the algorithm(sorted in the correct order)

On running through all the input of size n required. The below image shows the result generated in a microsecond(time taken by an algorithm to perform the sorting operation).



```
../sample1.txt 70
../sample2.txt 1009
../sample3.txt 15049
../sample4.txt 118229
../sample5.txt 1184338
```

Fig: Picture from the heap sort result generated from linprog

The average time is taken for 20 randomly permutated instance of each size n:

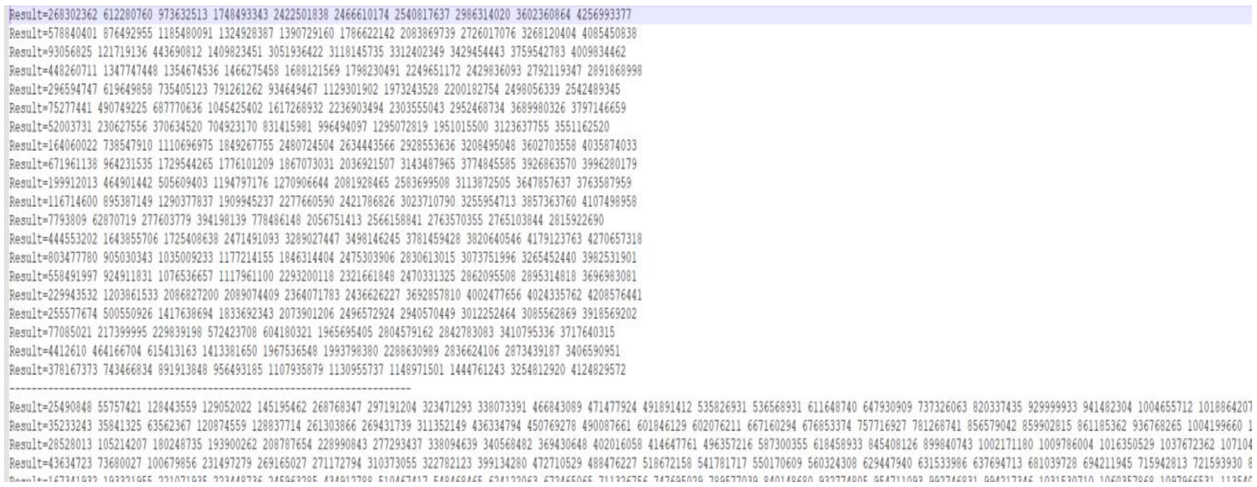
- N=10, (70/20)= 3.5 microseconds
- N=100, ((1009-70)/20)= 46.95 microseconds
- N=1000, ((15049-1009)/20)= 702.5 microseconds
- N=10000, ((118229-15049)/20)= 5159 microseconds
- N=100000, ((1184338-118229)/20)=50305.35 microseconds

The time in microseconds is calculated (added) in the program to know the overall total time.

The standard deviation on all inputs of size n: 0.50 sec(approx.)

QUICK SORT

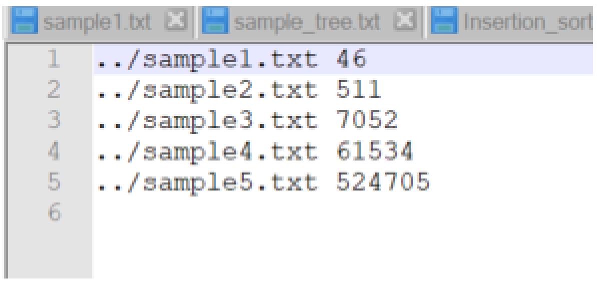
The below image shows the correctness of the algorithm, sorted in the correct order. The picture shows all sorted results of size ten and some of size 100.



```
Result=268302362 612280760 973632513 1748493343 2422501838 2466610174 2540817637 2986314020 3602360864 4256993377
Result=578840401 876492955 1185480091 1324928387 1390729160 1786622142 2083869739 2726017076 3268120404 4085450838
Result=93056825 121719136 443690812 1409823451 3051936422 3118145735 3312402349 3429454443 3759542783 4009834462
Result=448260711 1347747448 1354674536 1466275458 1688121569 1798230491 2249651172 2429836093 2792119347 2891868998
Result=296594747 619649858 735405123 791261262 934649467 1129301902 1973243528 2200182754 2498056339 2542489345
Result=75277441 490749225 687770636 1045425402 1617268932 2236903494 2303555043 2952468734 3689980326 3797146659
Result=52003731 230627556 370634520 704923170 831415981 996494097 1295072819 1951015500 3123637755 3551162520
Result=164060022 738547910 1110696975 1849267755 2480724504 2634443566 2928553636 3208495048 3602703558 4035874033
Result=671961138 964231535 1729544265 1776101209 1867073031 2036921507 3143487965 3774845585 3926863570 3996280179
Result=199912013 464901442 505609403 1194797176 1270906644 2081928465 2583699508 3113872505 3647857637 3763587959
Result=116714600 895387149 1290377837 1909945237 2277660590 2421786826 3023710790 3255954713 3857363760 4107498958
Result=7793809 62870719 277603779 394198139 778486148 2056751413 2566158841 2763570355 2765103844 2815922690
Result=444553202 1643855706 1725408638 2471491093 3289027447 3498146245 3781459428 3820640546 4179123763 4270657318
Result=803477780 905030343 1035009233 1177214155 1846314404 2475303906 2830613015 3073751996 3265452440 3982531901
Result=558491997 924911831 1076536657 1117961100 2293200118 2321661848 2470331325 2862095508 2895314818 3696983081
Result=229943532 1203861533 2086827200 2089074409 2364071783 2436626227 3692857810 4002477656 4024335762 4208576441
Result=255577674 500550926 1417638694 1833692343 2073901206 2496572924 2940570449 3012252464 3085562869 3918569202
Result=77085021 217399995 229839198 572423708 604180321 1965695405 2804579162 2842783083 3410795336 3717640315
Result=4412610 464166704 615413163 1413381650 1967536548 1993798380 2288630989 2836624106 2873439187 3406590951
Result=378167373 743466834 891913848 956493185 1107935879 1130955737 1148971501 1444761243 3254812920 4124829572
---------------------------------------------------------------------------
Result=25490848 55757421 128443559 129052022 145195462 268768347 297191204 323471293 338073391 466843089 471477924 491891412 535826931 536568931 611648740 647930909 737326063 820337435 929999933 941482304 1004655712 1018864207
Result=35233243 35841325 63562367 120874559 128837714 261303866 269431739 311352149 436334794 450769278 490087661 601846129 602076211 667160294 676853374 757716927 781268741 856579042 859902815 861185362 936768265 1004199660 1
Result=28528013 105214207 180248735 193900262 208787654 228990843 277293437 338094639 340568482 369430648 402016058 414647761 496357216 587300355 618458933 845408126 899840743 1002171180 1009786004 1016350529 1037672362 107104
Result=43634723 73680027 100679856 231497279 269165027 271172794 310373055 322782123 399134280 472710529 488476227 518672158 541781717 550170609 560324308 629447940 631533986 637694713 681039728 694211945 715942813 721593930 8
```

Fig: showing the correctness of the algorithm(sorted in the correct order)

On running through all the input of size n required. The below image shows the result generated in a microsecond(time taken by an algorithm to perform the sorting operation).



```
../sample1.txt 46
../sample2.txt 511
../sample3.txt 7052
../sample4.txt 61534
../sample5.txt 524705
```

Fig: Picture from the quick sort result generated from linprog

The average time is taken for 20 randomly permutated instance of each size n:

- N=10, (46/20)= 2.3 microseconds
- N=100, ((511-46)/20)= 23.25 microseconds
- N=1000, ((7052-511)/20)= 327.05 microseconds
- N=10000, ((61534-7052)/20)= 2724.1 microseconds
- N=100000, ((524705-61534)/20)=23158.55 microseconds

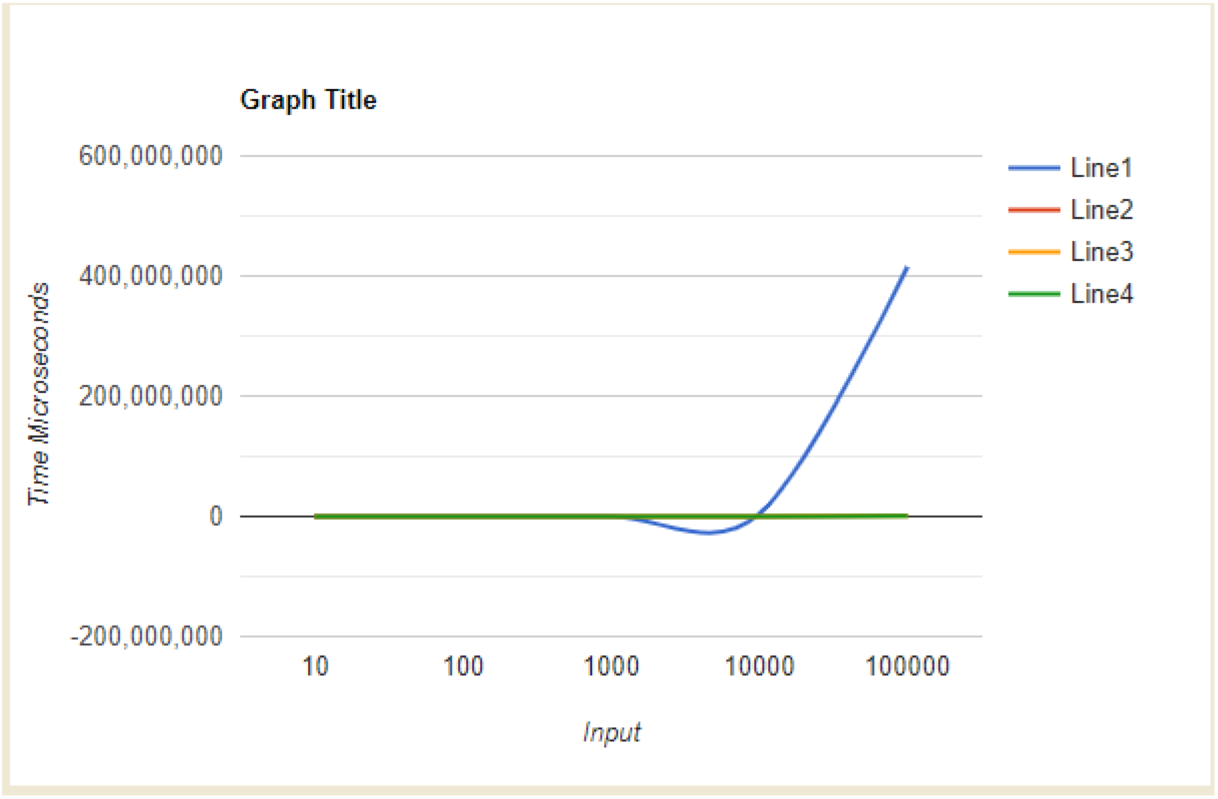
The time in microseconds is calculated (added) in the program to know the overall total time.

Standard deviation on all inputs of size n: 0.20 sec(approx.)

ANALYSIS ON SORTING

Below is the graph plotted from the value generated for each input size on linprog

(Blue line is: Insertion sort, Red is Merge Sort, Yellow is Heap sort, and Green is Quick Sort)



Since the time complexity of Insertion sort is O(n^2), it is visible how its graph(computation time) is growing with an increase in the input size. The rest sorting algorithm results are on the same line because all take O(N log N) time. In the worst case, Quick Sort is O(n^2). Maybe that is why seeing a slight deviation in between its graph, but the deviation is minimal. Therefore Quick Sort is so popular as it is too fast.

Also in C++ STL, sort() function used quick sort.

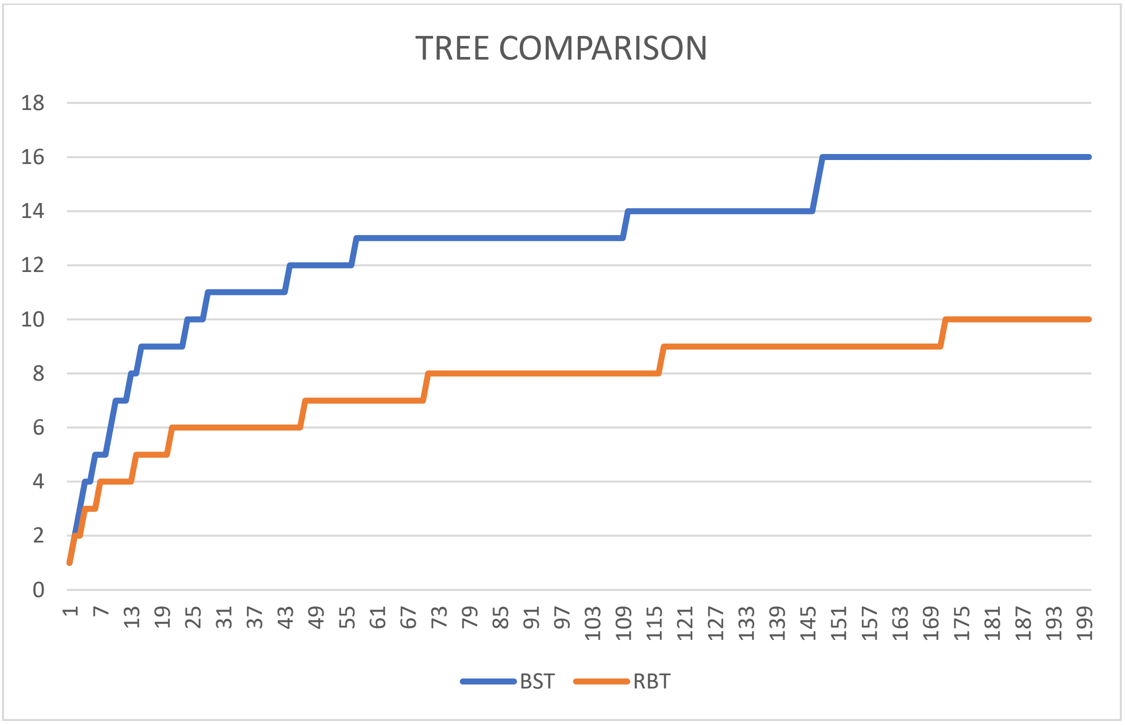
TREES

Both the Binary search tree and Red-black tree are height are calculated on the random input between o to 1000(size of n=200 in testing environment).

The Height function in both the tree was called to see the change in the tree's height.

Result: It is observed that the height of the BST max is 16 and the height of RBT max is 10

ANALYSIS ON TREES



It is visible that the height of the BST(Binary Search Tree) is increasing fast than the RBT(Red-black tree). It is because the Red-Black Tree is a self-balancing tree. This analysis means that the RBT height is O(log n) always(no matter input arrangements).

CONCLUSION

This project helped to understand that theories are derived from evidence. Our leaning to calculate Big-O notation theoretically was correct, as it matches with the data generated.

It is now clear why Quicksort is one of the most popular sorting algorithms, and many programming languages have used it to implement the sort() function. It is also easy to accurately decide which sorting algorithm fits based on the input(scenario).

The Red-Black tree's use gives the drawback of the Binary search tree that if the input is in ascending order, then the BST height is O(n), while RBT is O(log n). Although the implementation of RBT is difficult than BST, if implemented, then RBT is a better solution than BST in various scenarios like (searching).

This project(study) helped to understand the real purpose of computation time and algorithm design.