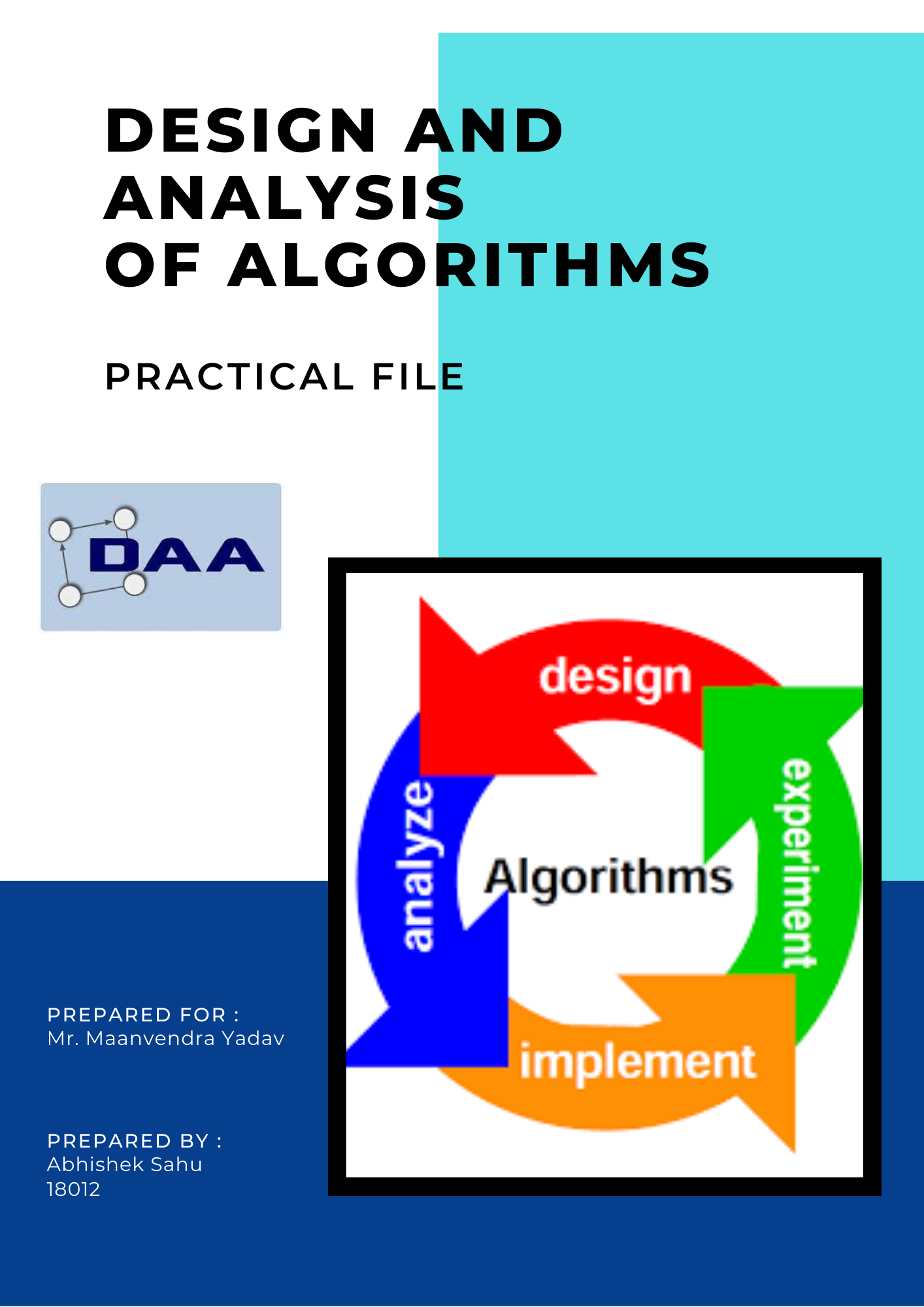
****

1. **Implement Insertion Sort**

Code:

#include <cstdlib>

#include <fstream>

#include <iomanip>

#include <iostream>

#define MIN\_SIZE 30

#define MAX\_SIZE 1000

using namespace std;

int insertionSort(int \*, int);

int main()

{

try

{

srand(time(0));

int array[MAX\_SIZE];

int size, comparisons;

ofstream fout("./results.csv");

cout << "+------------------------------------------------+\n";

cout << "| Input Size | Best Case | Avg Case | Worst Case |\n";

cout << "+------------------------------------------------+\n";

fout << "size,best,avg,worst\n";

for (int i = 0; i < 100; i++)

{

// rand() % (upperBound + 1 - lowerBound) + lowerBound

size = rand() % (MAX\_SIZE + 1 - MIN\_SIZE) + MIN\_SIZE;

// Input Size

cout << "| " << setw(10) << size;

fout << size << ",";

// Best Case

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

array[i] = i + 1;

comparisons = insertionSort(array, size);

cout << " | " << setw(9) << right << comparisons;

fout << comparisons << ",";

// Average Case

try

{

ifstream fin("./random.txt");

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

fin >> array[i];

fin.close();

comparisons = insertionSort(array, size);

cout << " | " << setw(8) << right << comparisons;

fout << comparisons << ",";

}

catch (exception e)

{

cerr << e.what();

return -1;

}

// Worst Case

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

array[i] = size - i;

comparisons = insertionSort(array, size);

cout << " | " << setw(10) << right << comparisons << " |\n";

fout << comparisons << "\n";

}

cout << "+------------------------------------------------+\n\n";

fout.close();

return 0;

}

catch (exception e)

{

cerr << e.what();

return -1;

}

}

int insertionSort(int \*array, int size)

{

int i, j, k, key, count = 0;

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

{

key = array[i];

for (j = i - 1; j >= 0; j--)

{

count++;

if (array[j] > key)

{

array[j + 1] = array[j];

}

else

{

break;

}

}

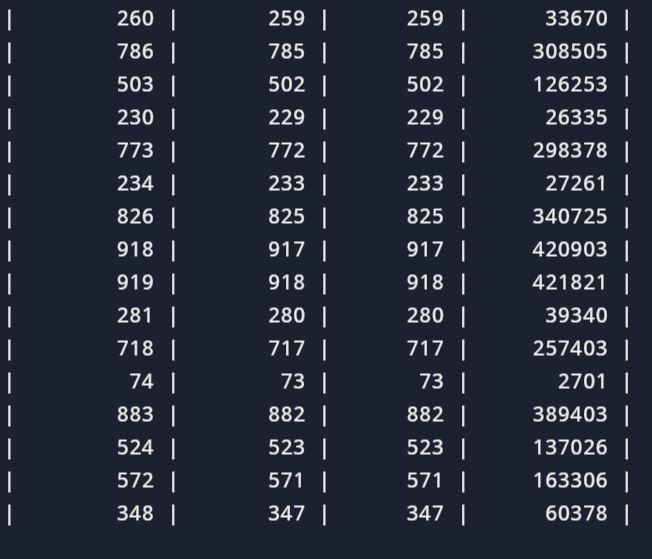
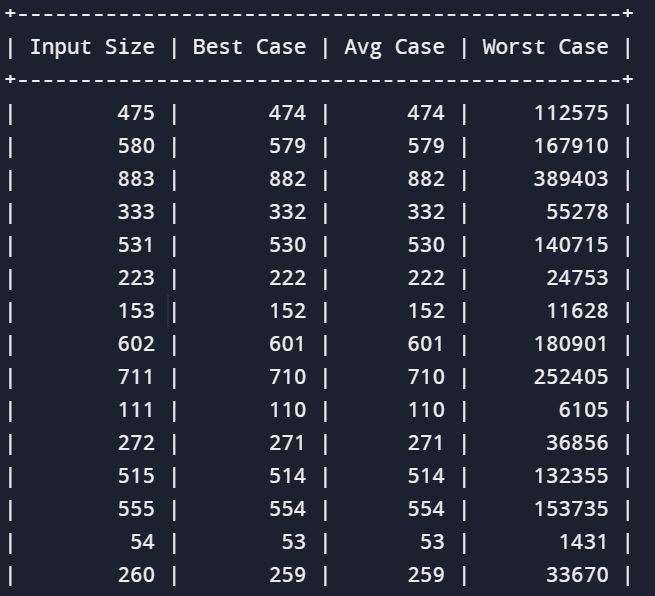
array[j + 1] = key;

}

return count;

}

Output



Plotting of graph

import math

import numpy as np

Plotting of graph

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv("results.csv")

df = df.sort\_values("size")

plt.figure(figsize=(8, 6))

plt.plot(df['size'], df['best'], 'g')

plt.plot(df['size'], df['avg'], 'b')

plt.plot(df['size'], df['worst'], 'r')

plt.plot(df['size'], df['size'] \* np.log2(df['size']), 'k--')

plt.legend(['best case', 'avg case', 'worst case', 'reference: nlogn'])

plt.title('Insertion Sort')

plt.xlabel('Input Size')

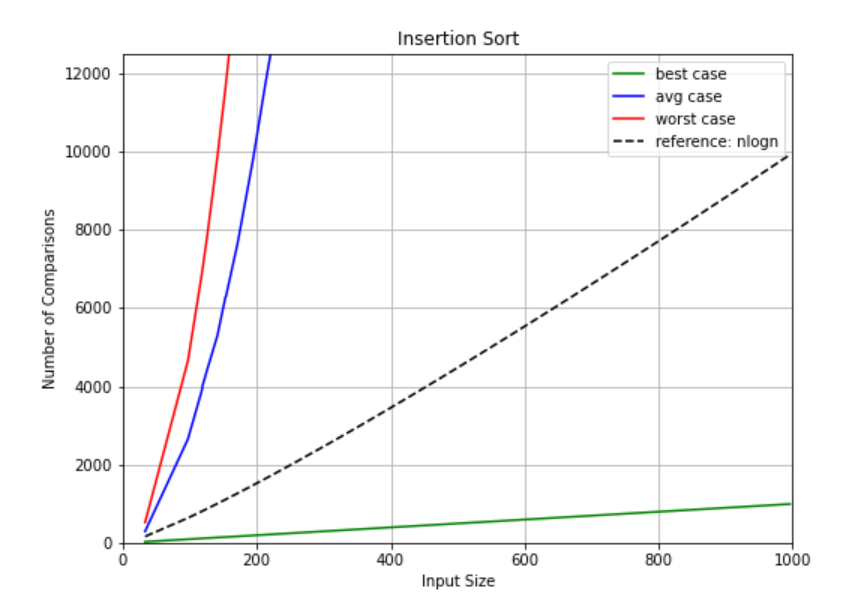
plt.ylabel('Number of Comparisons')

plt.ylim(0, 12500)

plt.xlim(0, 1000)

plt.grid()

plt.savefig('plot.png')



2. Implement Merge Sort

Code

#include <cstdlib>

#include <fstream>

#include <iomanip>

#include <iostream>

#define MIN\_SIZE 30

#define MAX\_SIZE 1000

using namespace std;

int mergeSort(int \*, int, int);

int merge(int \*, int, int, int);

int main()

{

try

{

srand(time(0));

int array[MAX\_SIZE];

int size, comparisons;

ofstream fout("./results.csv");

cout << "+------------------------------------------------+\n";

cout << "| Input Size | Best Case | Avg Case | Worst Case |\n";

cout << "+------------------------------------------------+\n";

fout << "size,best,avg,worst\n";

for (int i = 0; i < 100; i++)

{

// rand() % (upperBound + 1 - lowerBound) + lowerBound

size = rand() % (MAX\_SIZE + 1 - MIN\_SIZE) + MIN\_SIZE;

// Input Size

cout << "| " << setw(10) << size;

fout << size << ",";

// Best Case

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

array[i] = i + 1;

comparisons = mergeSort(array, 0, size - 1);

cout << " | " << setw(9) << right << comparisons;

fout << comparisons << ",";

// Average Case

try

{

ifstream fin("./random.txt");

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

fin >> array[i];

fin.close();

comparisons = mergeSort(array, 0, size - 1);

cout << " | " << setw(8) << right << comparisons;

fout << comparisons << ",";

}

catch (exception e)

{

cerr << e.what();

return -1;

}

// Worst Case

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

array[i] = size - i;

comparisons = mergeSort(array, 0, size - 1);

cout << " | " << setw(10) << right << comparisons << " |\n";

fout << comparisons << "\n";

}

cout << "+------------------------------------------------+\n\n";

fout.close();

return 0;

}

catch (exception e)

{

cerr << e.what();

return -1;

}

}

int mergeSort(int \*array, int beg, int end)

{

int comparisons = 0;

if (beg < end)

{

int mid = (beg + end) / 2;

comparisons += mergeSort(array, beg, mid);

comparisons += mergeSort(array, mid + 1, end);

comparisons += merge(array, beg, mid, end);

}

return comparisons;

}

int merge(int \*array, int beg, int mid, int end)

{

int comparisons = 0;

int n1 = mid - beg + 1;

int n2 = end - mid;

int L[n1 + 1], R[n2 + 1];

for (int i = 0; i < n1; i++)

L[i] = array[beg + i];

for (int j = 0; j < n2; j++)

R[j] = array[mid + 1 + j];

L[n1] = R[n2] = INT16\_MAX;

for (int i = 0, j = 0, k = beg; k <= end; k++)

{

if (L[i] != INT16\_MAX &&

R[j] != INT16\_MAX)

comparisons++;

if (L[i] <= R[j])

array[k] = L[i++];

else

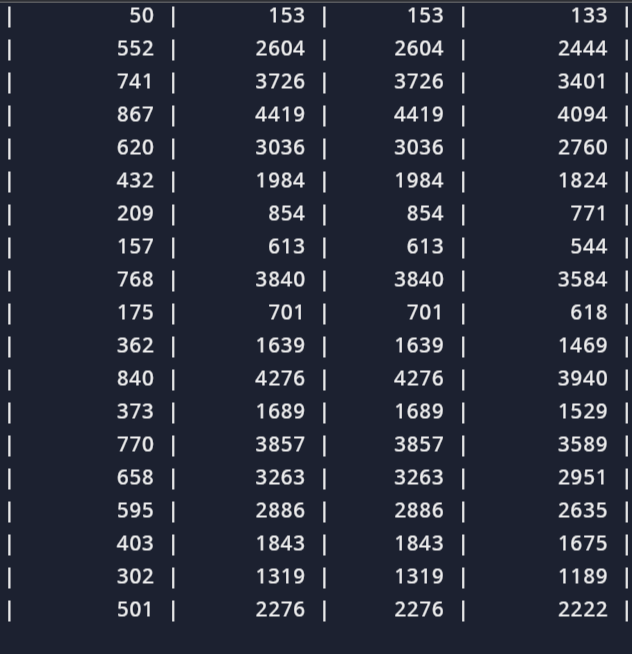
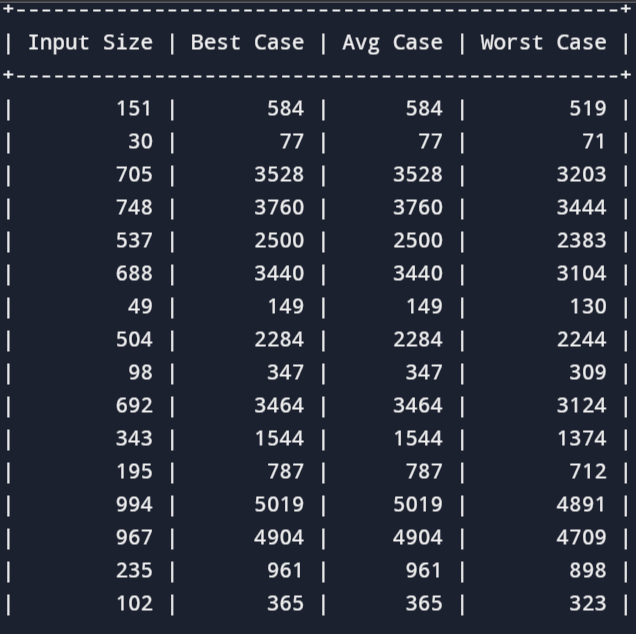
array[k] = R[j++];

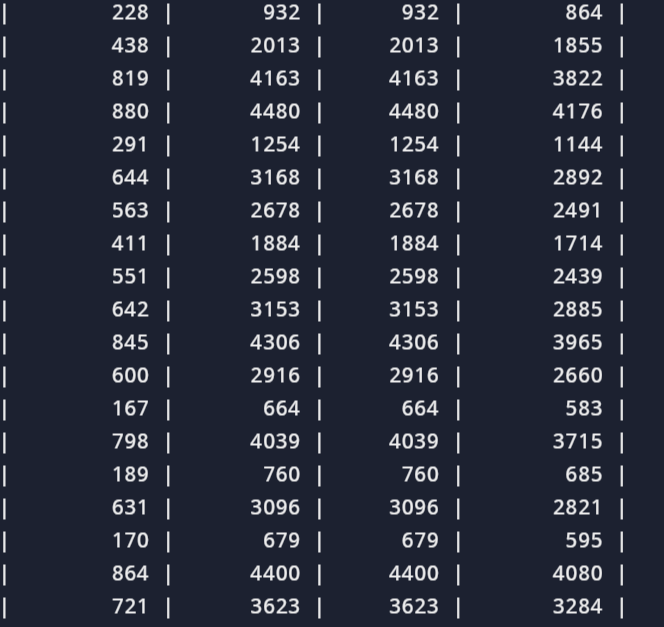
}

return comparisons;

}

Output





Plotting of graph

import math

import numpy as np

import pandas as pd

Plotting of graph

import matplotlib.pyplot as plt

df = pd.read\_csv("results.csv")

df = df.sort\_values("size")

plt.figure(figsize=(8, 6))

plt.plot(df['size'], df['best'], 'g')

plt.plot(df['size'], df['avg'], 'b')

plt.plot(df['size'], df['worst'], 'r')

plt.plot(df['size'], df['size'] \* np.log2(df['size']), 'k--')

plt.legend(['best case', 'avg case', 'worst case', 'reference: nlogn'])

plt.title('Merge Sort')

plt.xlabel('Input Size')

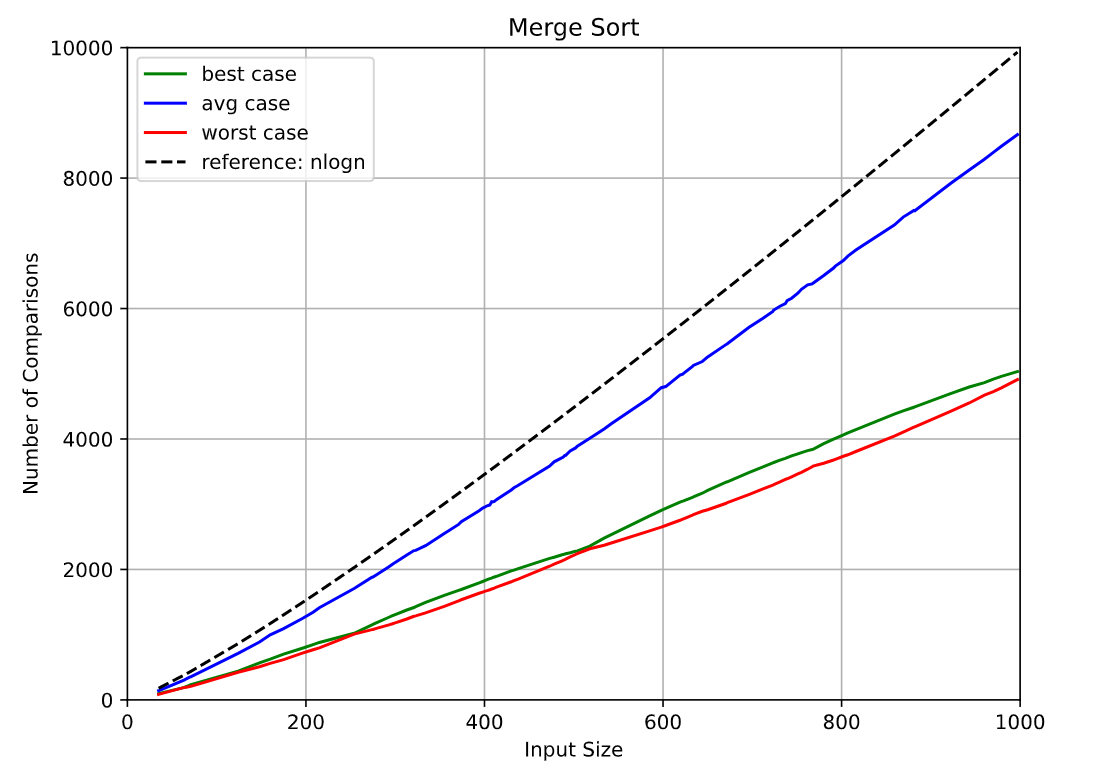
plt.ylabel('Number of Comparisons')

plt.ylim(0, 10000)

plt.xlim(0, 1000)

plt.grid()

plt.savefig('plot.png')



3. Implement Heap Sort

Code

#include <cstdlib>

#include <fstream>

#include <iomanip>

#include <iostream>

#define MIN\_SIZE 30

#define MAX\_SIZE 1000

using namespace std;

int comparisons;

int heap[MAX\_SIZE];

int parent(int i)

{

return (i - 1) / 2;

}

int left(int i)

{

return 2 \* i + 1;

}

int right(int i)

{

return 2 \* i + 2;

}

int maxHeapify(int \*&A, int n, int i)

{

int temp;

int largest;

int comparisons = 0;

int l = left(i);

int r = right(i);

if (l <= n && A[l] > A[i])

{

largest = l;

}

else

{

largest = i;

}

if (r <= n && A[r] > A[largest])

{

largest = r;

}

if (largest != i)

{

comparisons++;

temp = A[i];

A[i] = A[largest];

A[largest] = temp;

comparisons += maxHeapify(A, n, largest);

}

return comparisons;

}

int buildHeap(int A[], int n)

{

int comparisons = 0;

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

comparisons += maxHeapify(A, n, i);

return comparisons;

}

int heapSort(int A[], int n)

{

int comparisons = 0;

comparisons += buildHeap(A, n);

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

{

swap(A[0], A[i]);

comparisons += maxHeapify(A, i, 0);

}

return comparisons;

}

int main()

{

try

{

srand(time(0));

int array[MAX\_SIZE];

int size, comparisons;

ofstream fout("./results.csv");

cout << "+------------------------------------------------+\n";

cout << "| Input Size | Best Case | Avg Case | Worst Case |\n";

cout << "+------------------------------------------------+\n";

fout << "size,best,avg,worst\n";

for (int i = 0; i < 100; i++)

{

// rand() % (upperBound + 1 - lowerBound) + lowerBound

size = rand() % (MAX\_SIZE + 1 - MIN\_SIZE) + MIN\_SIZE;

// Input Size

cout << "| " << setw(10) << size;

fout << size << ",";

// Best Case

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

array[i] = i + 1;

comparisons = heapSort(array, size);

cout << " | " << setw(9) << right << comparisons;

fout << comparisons << ",";

// Average Case

try

{

ifstream fin("./random.txt");

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

fin >> array[i];

fin.close();

comparisons = heapSort(array, size);

cout << " | " << setw(8) << right << comparisons;

fout << comparisons << ",";

}

catch (exception e)

{

cerr << e.what();

return -1;

}

// Worst Case

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

array[i] = size - i;

comparisons = heapSort(array, size);

cout << " | " << setw(10) << right << comparisons << " |\n";

fout << comparisons << "\n";

}

cout << "+------------------------------------------------+\n\n";

fout.close();

return 0;

}

catch (exception e)

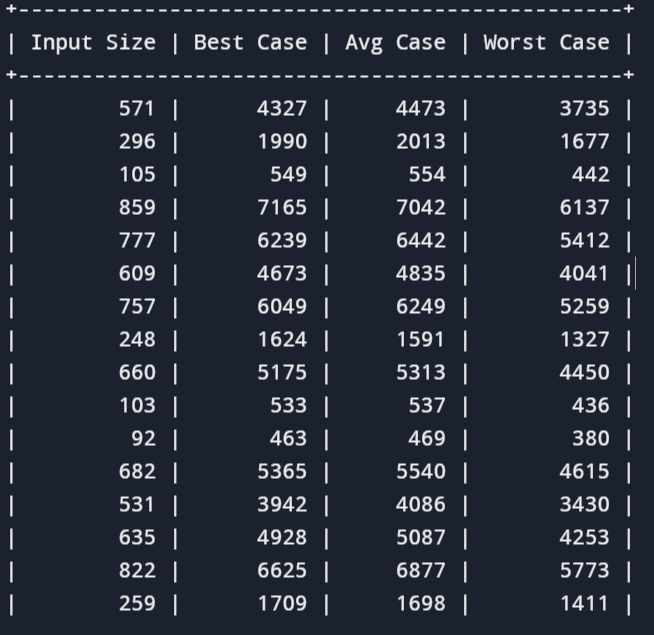
{

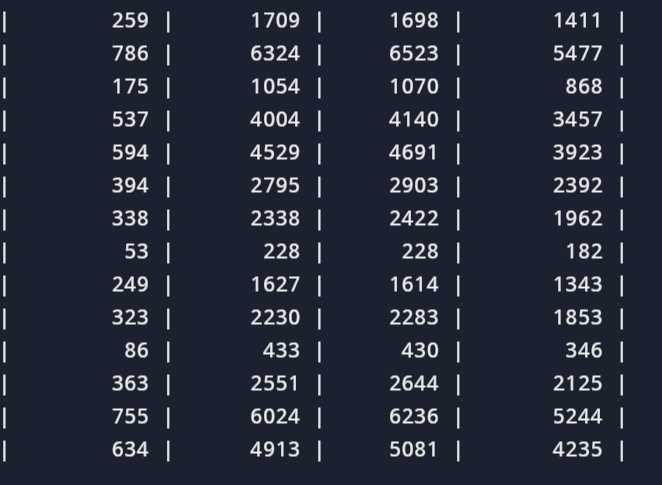
cerr << e.what();

return -1;

}

}

Output



Plotting of graph

import math

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv("results.csv")

df = df.sort\_values("size")

plt.figure(figsize=(8, 6))

plt.plot(df['size'], df['best'], 'g')

plt.plot(df['size'], df['avg'], 'b')

plt.plot(df['size'], df['worst'], 'r')

plt.plot(df['size'], df['size'] \* np.log2(df['size']), 'k--')

plt.legend(['best case', 'avg case', 'worst case', 'reference: nlogn'])

plt.title('Heap Sort')

plt.xlabel('Input Size')

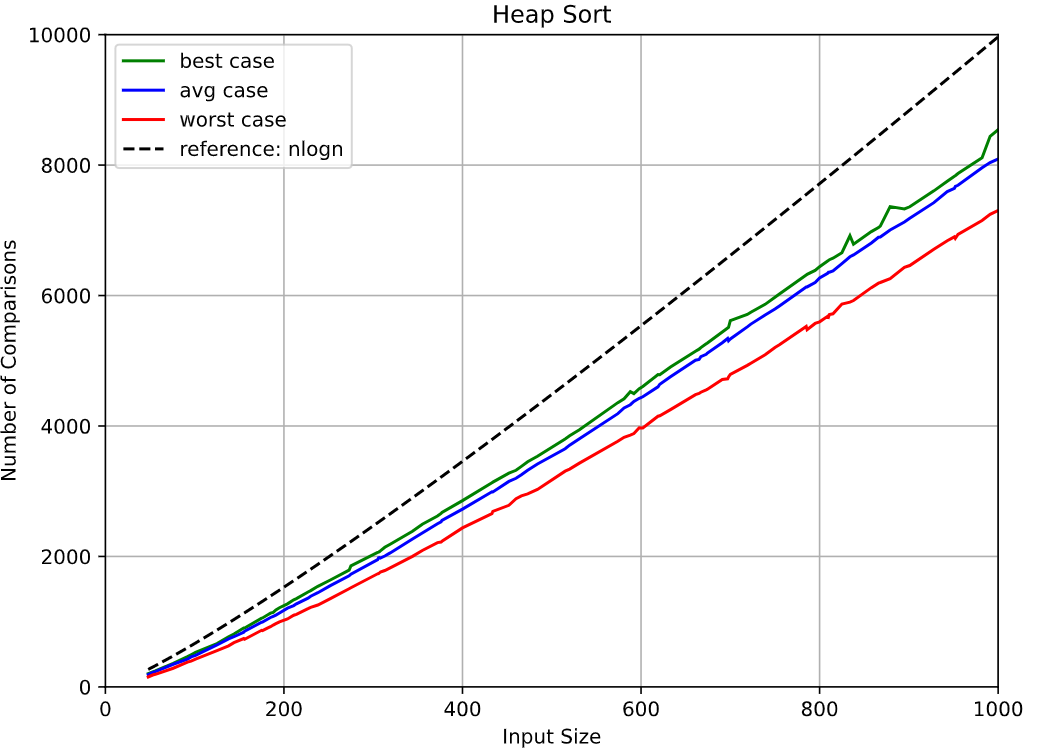
plt.ylabel('Number of Comparisons')

plt.ylim(0, 10000)

plt.xlim(0, 1000)

plt.grid()

plt.savefig('plot.png')



4. Implement Randomized Quick Sort

Code

#include <cstdlib>

#include <fstream>

#include <iomanip>

#include <iostream>

#define MIN\_SIZE 30

#define MAX\_SIZE 1000

using namespace std;

int partition(int \*&A, int p, int r, int &ctr)

{

int x = A[r];

int i = p - 1;

for (int j = p; j < r; j++)

{

ctr++;

if (A[j] <= x)

swap(A[++i], A[j]);

}

swap(A[i + 1], A[r]);

return i + 1;

}

int quickSort(int A[], int p, int r)

{

int comparisons = 0;

if (p < r)

{

int q = partition(A, p, r, comparisons);

comparisons += quickSort(A, p, q - 1);

comparisons += quickSort(A, q + 1, r);

}

return comparisons;

}

int randomPivotPartition(int A[], int p, int r, int &ctr)

{

swap(A[r], A[p + rand() % (r - p + 1)]);

return partition(A, p, r, ctr);

}

int randomizedQuickSort(int A[], int p, int r)

{

int comparisons = 0;

if (p < r)

{

int q = randomPivotPartition(A, p, r, comparisons);

comparisons += randomizedQuickSort(A, p, q - 1);

comparisons += randomizedQuickSort(A, q + 1, r);

}

return comparisons;

}

int main()

{

try

{

srand(time(0));

int array[MAX\_SIZE];

int size, comparisons;

ofstream fout("./results.csv");

cout << "+-------------------------------------------------------------+\n";

cout << "| Input Size | Best Case | Avg Case | Worst Case | Randomized |\n";

cout << "+-------------------------------------------------------------+\n";

fout << "size,best,avg,worst,randomized\n";

for (int i = 0; i < 100; i++)

{

// rand() % (upperBound + 1 - lowerBound) + lowerBound

size = rand() % (MAX\_SIZE + 1 - MIN\_SIZE) + MIN\_SIZE;

// Input Size

cout << "| " << setw(10) << size;

fout << size << ",";

// Best Case - Post Order of Balanced Tree

AVLTree tree;

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

tree.root = tree.insert(i + 1, tree.root);

int \*postArray = tree.getPostOrderArray(size);

comparisons = quickSort(postArray, 0, size - 1);

cout << " | " << setw(9) << right << comparisons;

fout << comparisons << ",";

// Average Case

try

{

ifstream fin("./random.txt");

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

fin >> array[i];

fin.close();

comparisons = quickSort(array, 0, size - 1);

cout << " | " << setw(9) << right << comparisons;

fout << comparisons << ",";

}

catch (exception e)

{

cerr << e.what();

return -1;

}

// Worst Case

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

array[i] = size - i;

comparisons = quickSort(array, 0, size - 1);

cout << " | " << setw(10) << right << comparisons;

fout << comparisons << ",";

// Randomized Quick Sort

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

array[i] = i + 1;

comparisons = randomizedQuickSort(array, 0, size - 1);

cout << " | " << setw(9) << right << comparisons << " |\n";

fout << comparisons << "\n";

}

cout << "+-------------------------------------------------------------+\n";

fout.close();

return 0;

}

catch (exception e)

{

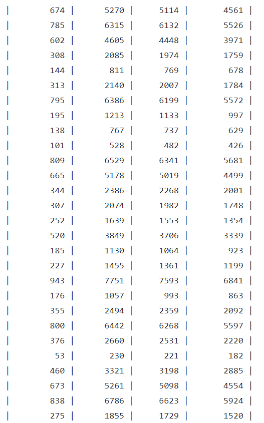
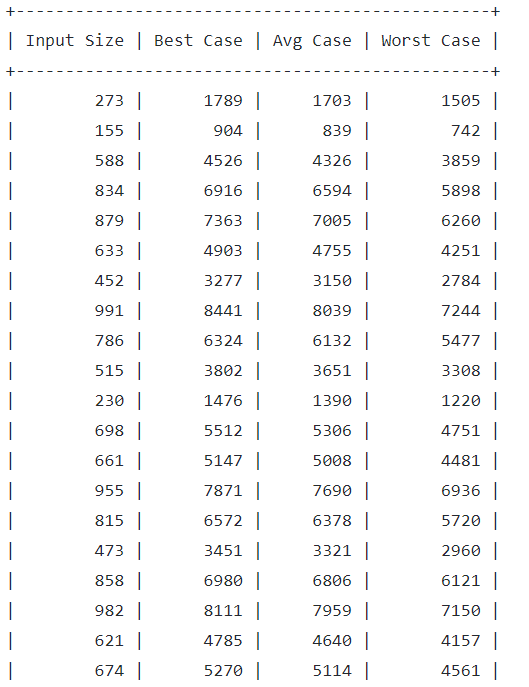
cerr << e.what();

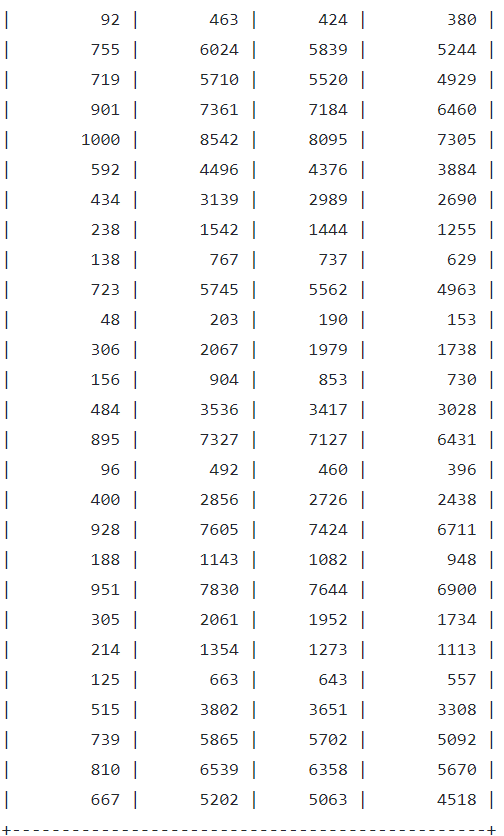
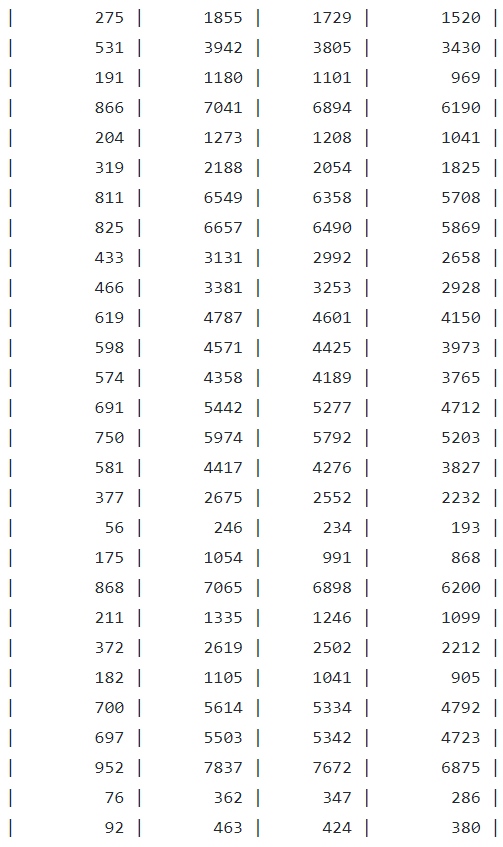
return -1;

}

}

Output





Plotting of graph

import math

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv("results.csv")

df = df.sort\_values("size")

plt.figure(figsize=(8, 6))

plt.plot(df['size'], df['best'], 'g')

plt.plot(df['size'], df['avg'], 'b')

plt.plot(df['size'], df['worst'], 'r')

plt.plot(df['size'], df['randomized'], 'r--')

plt.plot(df['size'], df['size'] \* np.log2(df['size']), 'k--')

plt.plot(df['size'], df['size'] \*\* 2, 'm--')

plt.legend(['best case', 'avg case', 'worst case', 'randomized', 'reference: nlogn', 'reference: n^2'])

plt.title('Quick Sort')

plt.xlabel('Input Size')

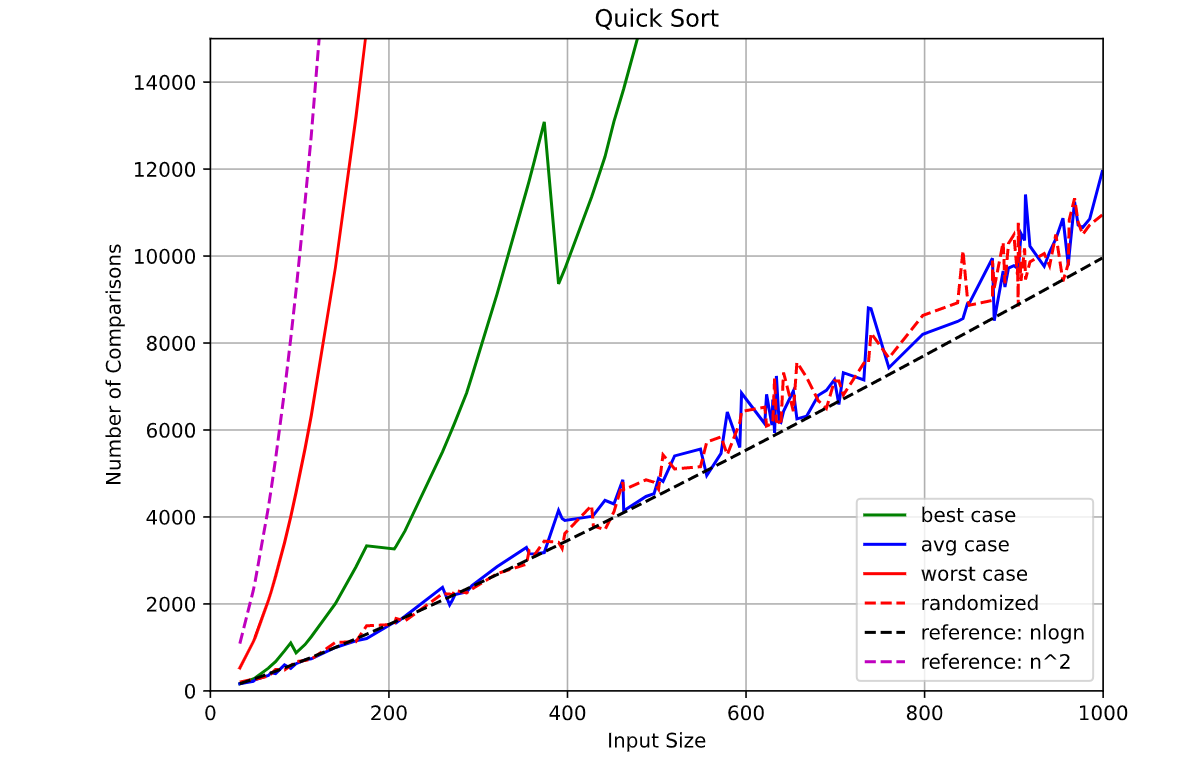
plt.ylabel('Number of Comparisons')

plt.ylim(0, 15000)

plt.xlim(0, 1000)

plt.grid()

plt.savefig('plot.png')



5. Implement Radix Sort

Code

#include <cstdlib>

#include <iomanip>

#include <iostream>

#define MAX\_SIZE 10

using namespace std;

int getMaximal(int A[], int n)

{

int m = A[0];

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

if (A[i] > m)

m = A[i];

return m;

}

void countingSort(int A[], int n, int e)

{

int i, B[n], C[10] = {0};

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

C[(A[i] / e) % 10]++;

for (i = 1; i < 10; i++)

C[i] += C[i - 1];

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

{

B[C[(A[i] / e) % 10] - 1] = A[i];

C[(A[i] / e) % 10]--;

}

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

A[i] = B[i];

}

void print(int A[], int n)

{

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

cout << A[i] << " ";

cout << endl;

}

void radixSort(int A[], int n)

{

int m = getMaximal(A, n);

for (int e = 1, count = 1; (m / e) > 0; e \*= 10, count++)

{

countingSort(A, n, e);

cout << "Pass " << count << ": ";

print(A, n);

}

}

int main()

{

try

{

srand(time(0));

int array[MAX\_SIZE];

int size = MAX\_SIZE;

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

array[i] = rand() % (1000 + 1 - 100) + 100;

cout << "Before Sorting: ";

print(array, size);

cout << "Sorting using Radix Sort...\n";

radixSort(array, size);

cout << "After Sorting: ";

print(array, size);

return 0;

}

catch (exception e)

{

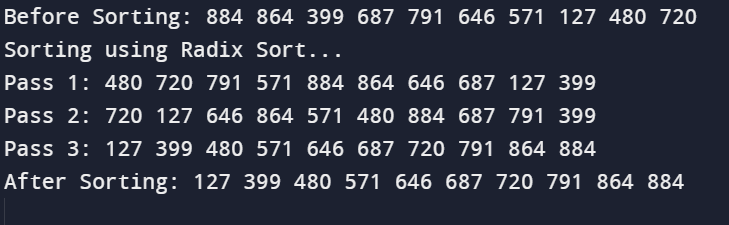
cerr << e.what();

return -1;

}

}

Output



6. Implement Bucket Sort

Code

#include <cmath>

#include <iomanip>

#include <iostream>

using namespace std;

#define MAX\_SIZE 8

#define MAX\_BUCKETS 10

#define BUCKET\_SIZE 10

template <class T>

class Node

{

public:

T info;

Node \*next;

};

template <class T>

Node<T> \*insertionSort(Node<T> \*list)

{

Node<T> \*k, \*nodeList;

if (list == nullptr || list->next == nullptr)

return list;

nodeList = list;

k = list->next;

nodeList->next = nullptr;

while (k != nullptr)

{

Node<T> \*ptr;

if (nodeList->info > k->info)

{

Node<T> \*temp = k;

k = k->next;

temp->next = nodeList;

nodeList = temp;

continue;

}

for (ptr = nodeList; ptr->next != 0; ptr = ptr->next)

{

if (ptr->next->info > k->info)

break;

}

if (ptr->next != 0)

{

Node<T> \*temp = k;

k = k->next;

temp->next = ptr->next;

ptr->next = temp;

continue;

}

else

{

ptr->next = k;

k = k->next;

ptr->next->next = nullptr;

continue;

}

}

return nodeList;

}

template <class T>

int getBucketIndex(T value)

{

return value \* BUCKET\_SIZE;

}

template <class T>

void BucketSort(T array[])

{

int i, j;

Node<T> \*\*buckets;

buckets = (Node<T> \*\*)malloc(sizeof(Node<T> \*) \* MAX\_BUCKETS);

for (i = 0; i < MAX\_BUCKETS; ++i)

buckets[i] = nullptr;

for (i = 0; i < MAX\_SIZE; ++i)

{

int pos = getBucketIndex(array[i]);

Node<T> \*current = new Node<T>();

current->info = array[i];

current->next = buckets[pos];

buckets[pos] = current;

}

cout << "Binning..." << endl;

for (i = 0; i < MAX\_BUCKETS; i++)

{

cout << "\tBucket[" << i << "]: ";

printBuckets(buckets[i]);

cout << endl;

}

for (i = 0; i < MAX\_BUCKETS; ++i)

buckets[i] = insertionSort(buckets[i]);

cout << "Sorting within bins..." << endl;

for (i = 0; i < MAX\_BUCKETS; i++)

{

cout << "\tBucket[" << i << "]: ";

printBuckets(buckets[i]);

cout << endl;

}

cout << "Concatenating buckets..." << endl;

for (j = 0, i = 0; i < MAX\_BUCKETS; ++i)

{

Node<T> \*node = buckets[i];

while (node)

{

array[j++] = node->info;

node = node->next;

}

}

for (i = 0; i < MAX\_BUCKETS; ++i)

{

Node<T> \*node = buckets[i];

while (node)

{

Node<T> \*temp = node;

node = node->next;

free(temp);

}

}

free(buckets);

return;

}

template <class T>

void print(T ar[])

{

int i;

for (i = 0; i < MAX\_SIZE; ++i)

cout << ar[i] << " ";

cout << endl;

}

template <class T>

void printBuckets(Node<T> \*list)

{

Node<T> \*cur = list;

while (cur != nullptr)

{

cout << cur->info << " ";

cur = cur->next;

}

}

int main()

{

try

{

srand(time(0));

double array[MAX\_SIZE];

int size = MAX\_SIZE;

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

array[i] = double(rand() % (1000 + 1 - 100) + 100) / double(1000);

cout << "Before Sorting: ";

print<double>(array);

cout << "Sorting using Radix Sort...\n";

BucketSort<double>(array);

cout << "After Sorting: ";

print<double>(array);

return 0;

}

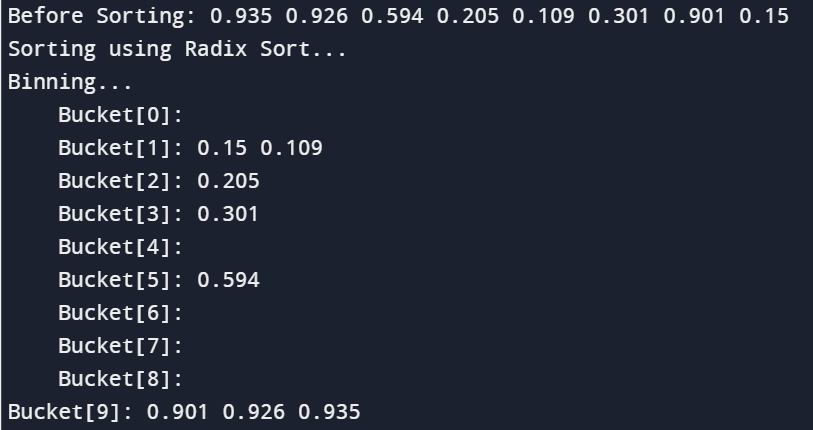
catch (exception e)

{

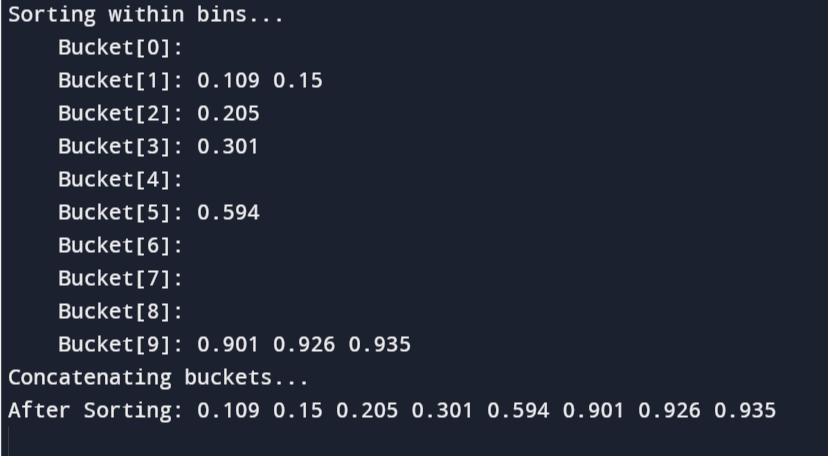
cerr << e.what();

return -1;

}

}

Output



7. Implement Randomized Select

Code

#include <bits/stdc++.h>

using namespace std;

int partition(int arr[], int l, int r)

{

int x = arr[r], i = l;

for (int j = l; j <= r - 1; j++) {

if (arr[j] <= x) {

swap(arr[i], arr[j]);

i++;

}

}

swap(arr[i], arr[r]);

return i;

}

int kthSmallest(int arr[], int l, int r, int k)

{

if (k > 0 && k <= r - l + 1) {

int index = partition(arr, l, r);

if (index - l == k - 1)

return arr[index];

if (index - l > k - 1)

return kthSmallest(arr, l, index - 1, k);

return kthSmallest(arr, index + 1, r,

k - index + l - 1);

}

return INT\_MAX;

}

int main()

{

int arr[] = { 10, 4, 5, 8, 6, 11, 26 };

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

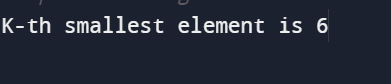
int k = 3;

cout << "K-th smallest element is "

<< kthSmallest(arr, 0, n - 1, k);

return 0;

}

Output 

8. Implement Breadth-First Search in a graph

Code

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

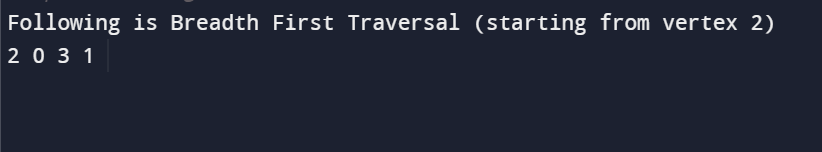
<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

Output



9. Implement Depth-First-Search in a graph

Code

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include <bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

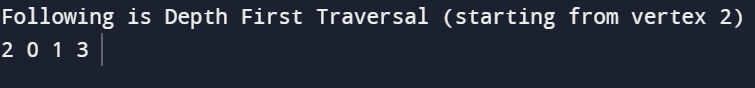
" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

Output



10. Write a program to determine the minimum spanning tree of a graph using both Prims and Kruskals algorithm.

Prims MST

Code

// A C++ program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(int key[], bool mstSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the

// constructed MST stored in parent[]

void printMST(int parent[], int graph[V][V])

{

cout<<"Edge \tWeight\n";

for (int i = 1; i < V; i++)

cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

void primMST(int graph[V][V])

{

// Array to store constructed MST

int parent[V];

// Key values used to pick minimum weight edge in cut

int key[V];

// To represent set of vertices included in MST

bool mstSet[V];

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first vertex.

key[0] = 0;

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

// Pick the minimum key vertex from the

// set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of

// the adjacent vertices of the picked vertex.

// Consider only those vertices which are not

// yet included in MST

for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

// print the constructed MST

printMST(parent, graph);

}

// Driver code

int main()

{

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

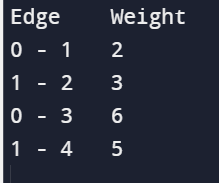
// Print the solution

primMST(graph);

return 0;

}

Output



Kruskals MST

Code

#include<bits/stdc++.h>

using namespace std;

// Creating shortcut for an integer pair

typedef pair<int, int> iPair;

// Structure to represent a graph

struct Graph

{

int V, E;

vector< pair<int, iPair> > edges;

// Constructor

Graph(int V, int E)

{

this->V = V;

this->E = E;

}

// Utility function to add an edge

void addEdge(int u, int v, int w)

{

edges.push\_back({w, {u, v}});

}

// Function to find MST using Kruskal's

// MST algorithm

int kruskalMST();

};

// To represent Disjoint Sets

struct DisjointSets

{

int \*parent, \*rnk;

int n;

// Constructor.

DisjointSets(int n)

{

// Allocate memory

this->n = n;

parent = new int[n+1];

rnk = new int[n+1];

// Initially, all vertices are in

// different sets and have rank 0.

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

{

rnk[i] = 0;

//every element is parent of itself

parent[i] = i;

}

}

// Find the parent of a node 'u'

// Path Compression

int find(int u)

{

/\* Make the parent of the nodes in the path

from u--> parent[u] point to parent[u] \*/

if (u != parent[u])

parent[u] = find(parent[u]);

return parent[u];

}

// Union by rank

void merge(int x, int y)

{

x = find(x), y = find(y);

/\* Make tree with smaller height

a subtree of the other tree \*/

if (rnk[x] > rnk[y])

parent[y] = x;

else // If rnk[x] <= rnk[y]

parent[x] = y;

if (rnk[x] == rnk[y])

rnk[y]++;

}

};

/\* Functions returns weight of the MST\*/

int Graph::kruskalMST()

{

int mst\_wt = 0; // Initialize result

// Sort edges in increasing order on basis of cost

sort(edges.begin(), edges.end());

// Create disjoint sets

DisjointSets ds(V);

// Iterate through all sorted edges

vector< pair<int, iPair> >::iterator it;

for (it=edges.begin(); it!=edges.end(); it++)

{

int u = it->second.first;

int v = it->second.second;

int set\_u = ds.find(u);

int set\_v = ds.find(v);

// Check if the selected edge is creating

// a cycle or not (Cycle is created if u

// and v belong to same set)

if (set\_u != set\_v)

{

// Current edge will be in the MST

// so print it

cout << u << " - " << v << endl;

// Update MST weight

mst\_wt += it->first;

// Merge two sets

ds.merge(set\_u, set\_v);

}

}

return mst\_wt;

}

// Driver program to test above functions

int main()

{

/\* Let us create above shown weighted

and undirected graph \*/

int V = 9, E = 14;

Graph g(V, E);

// making above shown graph

g.addEdge(0, 1, 4);

g.addEdge(0, 7, 8);

g.addEdge(1, 2, 8);

g.addEdge(1, 7, 11);

g.addEdge(2, 3, 7);

g.addEdge(2, 8, 2);

g.addEdge(2, 5, 4);

g.addEdge(3, 4, 9);

g.addEdge(3, 5, 14);

g.addEdge(4, 5, 10);

g.addEdge(5, 6, 2);

g.addEdge(6, 7, 1);

g.addEdge(6, 8, 6);

g.addEdge(7, 8, 7);

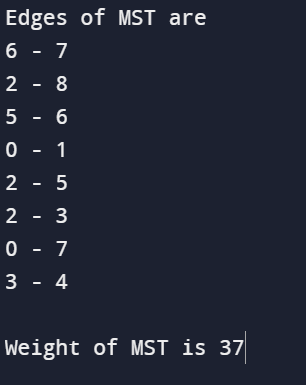
cout << "Edges of MST are \n";

int mst\_wt = g.kruskalMST();

cout << "\nWeight of MST is " << mst\_wt;

return 0;

}

Output

11. Write a program to solve the weighted interval scheduling problem.

Code

#include <iostream>

#include <algorithm>

using namespace std;

int M[20];

// Job data structure

struct Job {

int s, // Start Time

f, // Finish Time

w; //weight

};

int p(Job jobs[], int j)

{

for(int i = j - 1; i >= 0; i--)

if (jobs[i].f <= jobs[j].s)

return i;

// return the negative index if no non-conflicting job is found

return 0;

}

// Function to compare jobs used to sort them according to finish time

bool compareJob(Job j1, Job j2)

{

return (j1.f < j2.f);

}

// Function to compute optimal value using memoization

int ComputeOpt(Job jobs[], int j)

{

if (M[j] == -1)

{

M[j] = max(jobs[j].w + ComputeOpt(jobs, p(jobs, j)),

ComputeOpt(jobs, j - 1));

return M[j];

}

}

// Function to print optimal solution

void FindSolution(Job jobs[], int j)

{

if (j == 0)

cout << "";

else if ((jobs[j].w + M[p(jobs, j)]) > M[j - 1])

{

cout << j << " ";

FindSolution(jobs, p(jobs, j));

}

else

FindSolution(jobs, j - 1);

}

// Main function to find the optimal solution

void weightedIntervalScheduling(Job jobs[], int n)

{

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

{

cout << endl << p(jobs, i) << " ";

}

cout << endl;

// Sort jobs according to finish time

sort(jobs, jobs + n, compareJob);

// Find value of optimal solution

M[0] = 0;

// for(int j = 1; j < n; j++)

// M[j] = max(jobs[j].w + M[p(jobs, j)], M[j - 1]);

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

{

int index = p(jobs, i);

int incl = jobs[i].w;

if (index != -1) {

incl += M[index];

}

M[i] = max(incl, M[i-1]);

}

cout << M[n-1] << " is the optimal value.";

// Find optimal solution

cout << "\nOptimal Solution: ";

FindSolution(jobs, n);

}

/\*

Driver Code

\*/

int main()

{

int n;

cout << "\nEnter the no of jobs: ";

cin >> n;

cout << "Enter the job details:\n";

Job jobs[n];

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

{

cout << "Starting time of Job " << i+1 << ": ";

cin >> jobs[i].s;

cout << "Finishing time of Job " << i+1 << ": ";

cin >> jobs[i].f;

cout << "Weight of Job " << i+1 << ": ";

cin >> jobs[i].w;

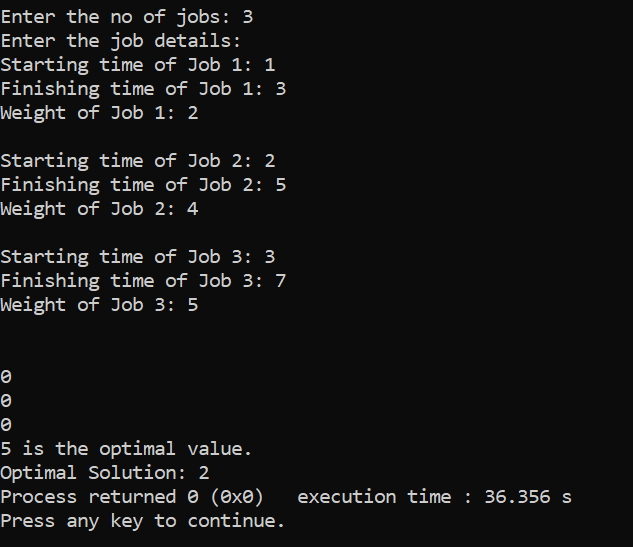
cout << endl;

}

weightedIntervalScheduling(jobs, n);

return 0;

}



Output

12. Write a program to solve the 0-1 knapsack problem.

Code

#include <iostream>

using namespace std;

int SubsetSum(int set[], int n, int W)

{

int M[n + 1][W + 1];

for(int w = 0; w <= W; w++)

M[0][w] = 0;

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

for(int w = 0; w <= W; w++)

if (w < set[i-1])

{

M[i][w] = M[i-1][w];

}

else

{

M[i][w] = max(M[i - 1][w], (set[i - 1] + M[i - 1][w - set[i - 1]]));

}

//--- Printing the array M

cout << "Final Iteration:\n";

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

{

for(int j = 0; j <= W; j++)

cout << M[i][j] << " ";

cout << endl;

}

cout << endl;

return M[n][W];

}

int main()

{

int n, W;

cout << "\nEnter the number of elements in set: ";

cin >> n;

int set[n];

cout << "Enter the set elements: ";

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

cin >> set[i];

cout << "Enter the sum: ";

cin >> W;

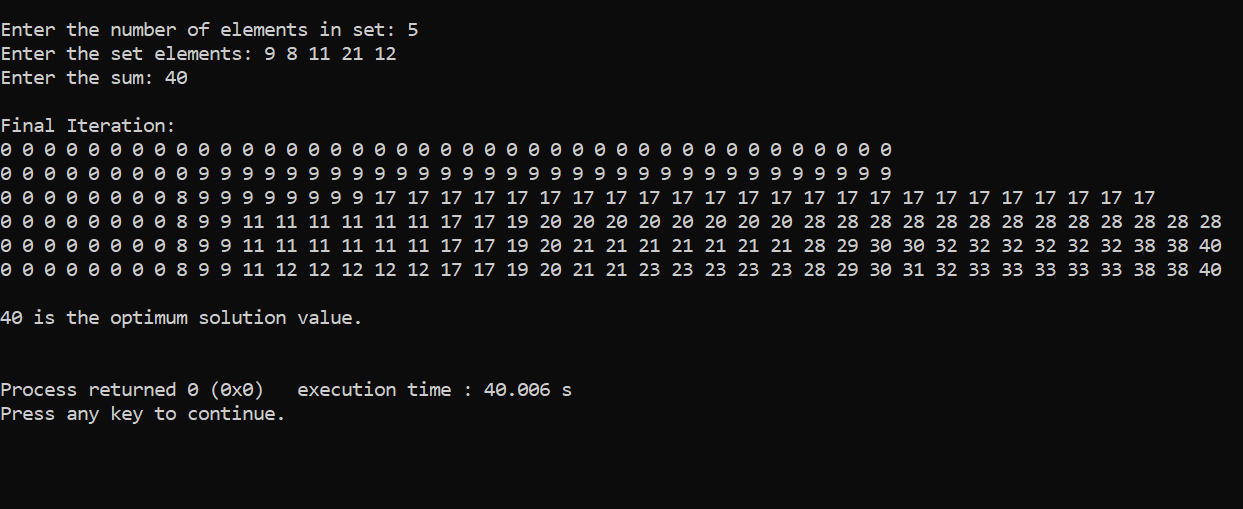
cout << endl;

cout << SubsetSum(set, n, W) << " is the optimum solution value.\n";

cout << endl;

return 0;

}

Output