PROGRAM -1

AIM:-Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

source code:-

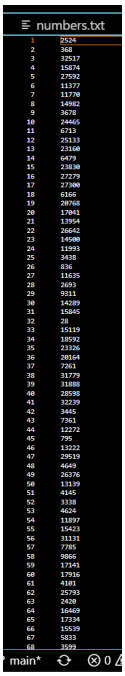
```
#include<iostream>
using std::cout,std::endl;
#include<vector>
using std::vector;
#include<algorithm>
#include<utility>
#include<fstream>
using std::ifstream,std::ofstream;
#include<iomanip>
#include <chrono>
#include <iterator>

template<typename comparable>
const comparable & median3(vector<comparable> & values, int left, int right){
   int center = (left+right)/2;
   if (values[right]<values[left]){
      std::swap(values[left],values[right]);
   }
   if(values[center]<values[left]){
      std::swap(values[center],values[left]);
   }
   if(values[right]<values[center]){
      std::swap(values[center],values[right]);
   }

   std::swap(values[center],values[right-1]);
   return(values[right-1]);
}

template<typename comparable>
void helper_isort(vector<comparable> & values, int left, int right){
   for(int p= left+1; p<=right;++p){
      comparable temp=std::move(values[p]);
      int j;
      for (j=p;j>left&& temp<values[j-1]; --j){
         values[j] = std::move(values[j-1]);
      }
      values[j]=std::move(temp);
   }
}

template<typename comparable>
void quicksort(vector<comparable> & values,int left, int right){
```

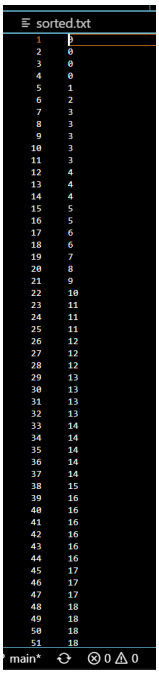
```
    if (left +10<= right){
        const comparable & pivot = median3(values,left,right);

        int i=left, j=right-1;
        for(;;){
            while(values[++i]<pivot){}
            while (pivot<values[--j]){}
            if(i<j){ std::swap(values[i],values[j]);}
            else{
                break;
            }

        }
        std::swap(values[i],values[right-1]) ;
         quicksort(values, left, i-1);
         quicksort (values, i+1,right);
    }
    else{
        helper_isort (values,left,right);
    }
}

template<typename comparable>
vector<comparable>& createVec(comparable range, vector<comparable> & values){
    ifstream input ("numbers.txt");
    if (input){
    cout<<"input file open success!\n";
    std::istream_iterator<comparable> start(input);
    std::copy_n(start,range,std::back_inserter(values));
    input.close();
    }
    else{
        std::cerr<<"oops! something wrong with input file opening!";
    }
    return values;
}

template<typename comparable>
void quicksort(vector<comparable> & values){

    quicksort(values,0, values.size()-1);
}


int main(){
   vector<unsigned long> values; //change the data type here.
   cout<<"enter the number of values to take from the file";
   unsigned long num;
   std::cin>>num;
   values.reserve(num);
   createVec(num,values);
```

```
   auto start = std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio(false);

    quicksort(values);

   auto end= std::chrono::high_resolution_clock::now();
    double time_taken=std::chrono::duration_cast<std::chrono::nanoseconds>(end-start).count();
    time_taken*=1e-9;
    cout<<"time taken by quicksort algorithm
is:"<<std::fixed<<std::setprecision(9)<<time_taken<<"sec"<<endl;


   ofstream output("sorted.txt");
    if(output){
        cout<<"output file opened successfully!\n";
        for (const auto & i: values){
            output<<i<<'\n';
        }
        output.close();
    }
    else{
        cout<<"oops! something went wrong while opening output file!";
    }
}
```

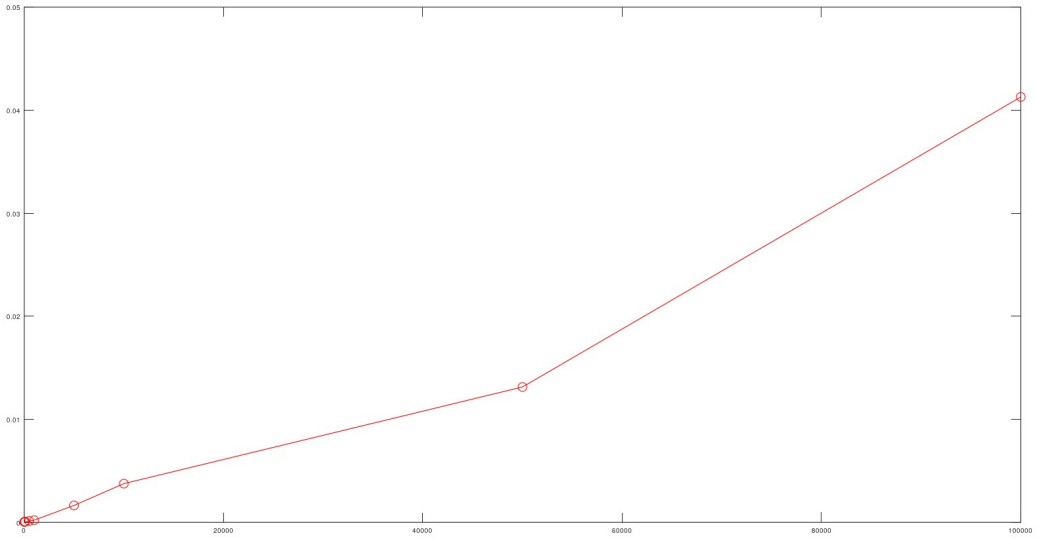
INPUT AND OUTPUT FILES:-

numbers.txt **sorted.txt**





TIME vs VALUES PLOT



PROGRAM -2

AIM:-Implement a parallelized Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

SOURCE CODE:-

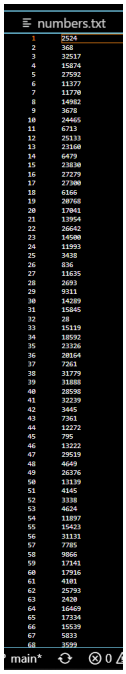
```
#include <iostream>
using std::cout,std::endl;
#include <vector>
using std::vector;
#include <thread>
using std::thread;
#include<fstream>
using std::ifstream , std::ofstream;
#include <algorithm>
#include<utility>
#include <iterator>
#include<iomanip>
#include <chrono>

template<typename T>
void merge(vector<T>& v, int start, int mid, int end)
{
   vector<T> temp(end - start + 1);
   int i = start, j = mid + 1, k = 0;
   while (i <= mid && j <= end) {
      if (v[i] <= v[j]) {
         temp[k++] = v[i++];
      } else {
         temp[k++] = v[j++];
      }
   }
   while (i <= mid) {
      temp[k++] = v[i++];
   }
   while (j <= end) {
      temp[k++] = v[j++];
   }
   for (int i = start, k = 0; i <= end; i++, k++) {
      v[i] = temp[k];
   }
}

template<typename T>
void mergesort(vector<T>& v, int start, int end)
{
   if (start >= end) {
      return;
   }
```

```
    int mid = start + (end - start) / 2;
    // Parallelize the recursive calls to mergesort
    thread left_thread(mergesort<T>, std::ref(v), start, mid);
    thread right_thread(mergesort<T>, std::ref(v), mid + 1, end);
    left_thread.join();
    right_thread.join();
    merge<T>(v, start, mid, end);
}

template<typename T>
vector<T>& createVec(T range, vector<T> & values){
    ifstream input ("numbers.txt");
    if (input){
    cout<<"input file open success!\n";
    std::istream_iterator<T> start(input);
    std::copy_n(start,range,std::back_inserter(values));
    input.close();
    }
    else{
        std::cerr<<"oops! something wrong with input file opening!";
    }
    return values;
}



int main()
{   vector<unsigned> values;
    cout<<"enter the number of values to take from the file";
    unsigned num;
    std::cin>>num;
    values.reserve(num);
    createVec(num,values);

    auto start = std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio(false);

    mergesort<unsigned>(values, 0, values.size() - 1);

    auto end= std::chrono::high_resolution_clock::now();
    double time_taken=std::chrono::duration_cast<std::chrono::nanoseconds>(end-start).count();
    time_taken*=1e-9;
    cout<<"time taken by parallel mergesort algorithm
is:"<<std::fixed<<std::setprecision(9)<<time_taken<<"sec"<<endl;


    ofstream output("merge-sorted.txt");
    if(output){
        cout<<"output file opened successfully!\n";
        for (const auto & i: values){
            output<<i<<'\n';
        }
```

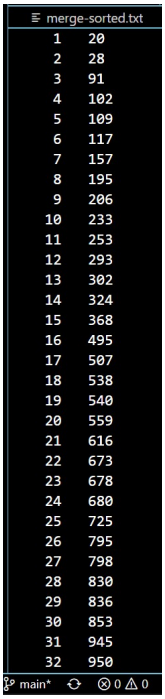
```
        output.close();
    }
    else{
        cout<<"oops! something went wrong while opening output file!";
    }
}
```

OUTPUT AND INPUT FILES:

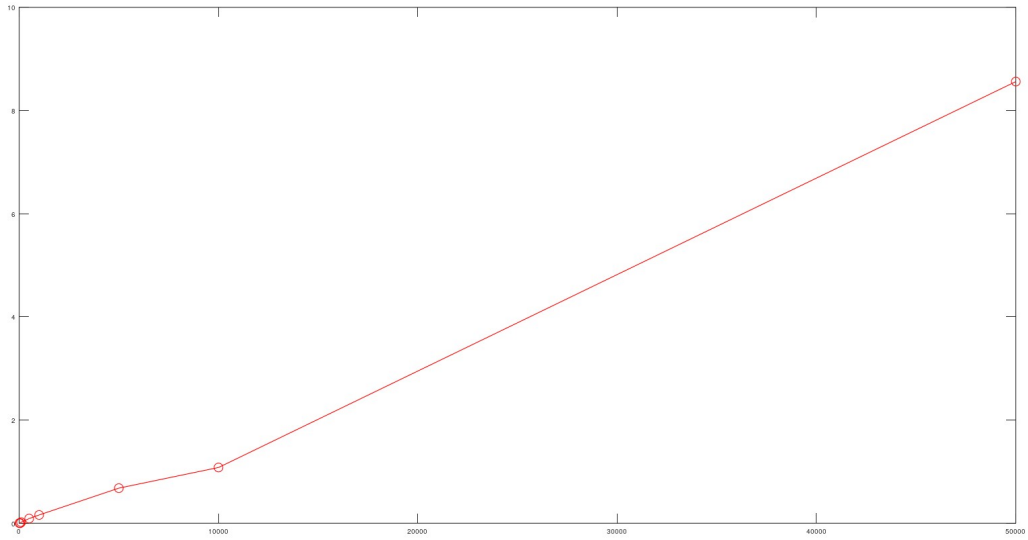
numbers.txt



merge-sorted.txt



TIME vs VALUES PLOT



PROGRAM -3(a)

AIM:-Obtain the Topological ordering of vertices in a given digraph.

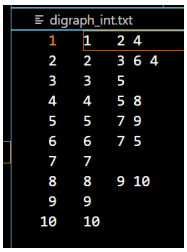
SOURCE CODE:-

```
#include <iostream>
using std::cout,std::endl,std::cerr;
#include<unordered_map>
using std::unordered_map;
#include <string>
using std::string;
#include<sstream>
using std::stringstream;
#include<fstream>
using std::ifstream;
#include <vector>
using std::vector;
#include<queue>
using std::queue;
#include <utility>
using std::move;

template <typename T>
using graph=unordered_map<T,vector<T>>;

template <typename T>
const vector<T> & topological_sort(const graph<T>& digraph, vector<T>&result){

    //initialising the indegrees
    unordered_map<T,unsigned int> indegrees;
    for(const auto & pair:digraph){
        indegrees[pair.first]=0;
    }
    for(const auto & pair:digraph){
        for(const auto& adj_vertices:pair.second){
            ++indegrees[adj_vertices];
        }
    }

    //queue to store all vertices with zero degrees.
    queue<T> zero_degrees;
    for (const auto & pair:indegrees){
        if (pair.second==0){
            zero_degrees.push(std::move(pair.first));
        }
    }

    while(!zero_degrees.empty()){
        const T& remove= zero_degrees.front();

        for(const auto& adj_vertex:(*digraph.find(remove)).second){
            --indegrees[adj_vertex];
```

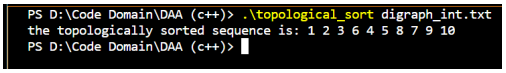
```
            if(indegrees[adj_vertex]==0){
                zero_degrees.push(adj_vertex);
            }
        }
        result.push_back(std::move(remove));
        zero_degrees.pop();

    }
    return result;
}

template<typename T>
const graph<T>& create_graph(ifstream& input_file,graph<T>& digraph){
    string line;
    T vertex,neighbour;
    while (std::getline(input_file,line))
    {
        stringstream vertices(line);
       if(!( vertices>>vertex)){
        cerr<<"Error parsing line:"<<line<<'\n';
        };
        digraph.insert({vertex,vector<T>(0)});
        while(vertices>>neighbour){
            digraph[vertex].push_back(neighbour);
        }

    }
    return digraph;

}

int main(int argc, char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    ifstream input_file(argv[1]);
    if (!input_file){
        cerr<<"error opening this file";
        return -1;
    }
    graph<int> digraph;                //you can change data type here
    vector<int> result;
    create_graph(input_file,digraph);
    input_file.close();

    const auto & topological_sequence=topological_sort(digraph,result);
    cout<<"the topologically sorted sequence is: ";
    for(const auto& vertices:topological_sequence){
        cout<<vertices<<" ";
    }
}
```

INPUT AND OUTPUT:-

digraph_int.txt



terminal output:-



PROGRAM-4

AIM:-Implement 0/1 Knapsack problem using Dynamic Programming.

SOURCE CODE:-

```
#include <iostream>
using std::cout,std::cin,std::cerr,std::endl;
#include<vector>
using std::vector;
#include<utility>
using std::pair;
#include<fstream>
using std::ifstream;
#include<string>
using std::string,std::stod;
#include<sstream>
using std::stringstream;
#include<cmath>
using std::pow;
#include<iomanip>

template<typename NameType,typename WeightType,typename ValueType>
using item_vector= vector<pair<NameType,pair<WeightType,ValueType>>>;

item_vector<string,double,double> data;  // change the data type here

void InitialiseData(ifstream & input){
   string line,name,weight,value;
   while (std::getline(input,line))
   {
      stringstream tokens(line);
      tokens>>name>>weight>>value;
      data.push_back({name,{stod(weight),stod(value)}});
   }
}

vector<vector<double>> memo_table;
unsigned decimal_places=0;

void InitialiseMemoTable(const double & capacity){
   //counting number of columns
   double min_weight=data[0].second.first;
   for(const auto &i:data){
      if (i.second.first<min_weight){
         min_weight=i.second.first;
         }
   }

   decimal_places=0;
   while(min_weight<1){
      min_weight*=10;
      ++decimal_places;
   }
```

```
    unsigned columns=(capacity*pow(10,decimal_places))+1;


    //initialsing memoization table
    memo_table= vector<vector<double>>(data.size()+1,vector<double>(columns,-1));
}



double knapsackRecursive(double remaining_capacity,  std::size_t ItemIndex){
    //base case
    if(ItemIndex==0 || remaining_capacity==0){
        memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)]=0;
        return 0;
     }
    //if the value is already computed
    if (memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)] != -1) {
        return memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)];
    }
    // If the weight of the current item is more than the remaining capacity,
    // skip the item and recursively call with the next item index
    if (data[ItemIndex].second.first > remaining_capacity) {
        memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)] =
knapsackRecursive(remaining_capacity, ItemIndex-1);
        return memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)];
    }
    // Calculate the maximum value by either including or excluding the current item
    double includeItem = data[ItemIndex].second.second + knapsackRecursive(remaining_capacity-
data[ItemIndex].second.first, ItemIndex - 1);
    double excludeItem = knapsackRecursive(remaining_capacity, ItemIndex-1);

    // Store the maximum value in the memoization table
    memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)] = std::max(includeItem,
excludeItem);
    return memo_table[ItemIndex][remaining_capacity*pow(10,decimal_places)];

 }

 void Show_knapsack(double remaining_capacity, std::size_t ItemIndex){
    vector<bool> items(data.size(),false);
    cout<<"The maximum value of items in knapsack
is :"<<std::fixed<<std::setprecision(decimal_places)<<memo_table[data.size()]
[remaining_capacity*pow(10,decimal_places)]<< " rupees"<<endl;
    cout<<"The contents of the Knapsack are:- \n";
    double max_value=1;
    unsigned max_row=0;
    double remaining_value=1;
    std::size_t current_limit=memo_table.size();
    while(remaining_value>0){
    max_row=0;
    auto current_column=remaining_capacity*(pow(10,decimal_places));
```

```
    max_value= memo_table[max_row][current_column];
    for (std::size_t current_row=0; current_row<current_limit;++current_row){
        if (memo_table[current_row][current_column]>max_value && items[current_row]==false){
            max_value=memo_table[current_row][current_column];
            max_row=current_row;
        }
    }
    remaining_value=max_value-data[max_row].second.second;
    items[max_row]=true;
    current_limit=max_row;
    remaining_capacity -= data[max_row].second.first;
    cout<<data[max_row].first<<'\t'<<data[max_row].second.first<<"grams
"<<data[max_row].second.second<<"rupees\n"<<endl;
    }
}

 void knapsack(const double & capacity){
    InitialiseMemoTable(capacity);
    //calling the recursive function
    std::size_t ItemIndex=data.size();
    knapsackRecursive(capacity,ItemIndex);
    Show_knapsack(capacity,ItemIndex);
}

int main(int argc, char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    double capacity;
    cout<<"Pleae Enter the capacity(positive!) of the kanapsack"<<endl;
    cin>>capacity;
    if (capacity<0){
        cerr<<"I told you to enter non- negative values!!";
        return -1;
    }
    else{

    ifstream input(argv[1]);
    if (!input){
        cerr<<"error opening this file";
        return -1;}
    InitialiseData(input);
    input.close();
    knapsack(capacity);
    }
    }
}
```

INPUT AND OUTPUT:-

knapsack.txt

```
guitar 1500 4000
mobile 200 30000
laptop 2000 60000
metal_ore 1000000 100000000
pencil_box 50 60
diamond_ring 5 5000000
feather 1 10
gold_bar 40000 50000000
pen 5 2
plastic_spoon 1 1
aluminium_ignot 1000 500000
feather_boa 3 8000
sapphire_bracelet 4 150000
paper_clip 0.01 0.01
platinum_ring 1 200000
kiwi_crate 50000 20000
plastic_bottle 0.5 1
ruby_earrings 3 120000
pebble 0.1 0.05
bronze_medal 10 1000
feather 0.02 0.05
glass_marble 0.5 5
emerald_pendant 20 180000
toothpick 0.1 0.01
gold_coin 0.9 100000
microchip 0.01 1000000
feather_duster 2 5000
plastic_fork 0.5 0.1
silver_bracelet 15 25000
eraser 5 1
headphones 150 2000
watch 100 2500
water_bottle 200 100
camera 800 5000
shoes 600 3000
umbrella 400 150
wallet 50 1000
socks 50 200
hat 100 300
sunglasses 100 700
towel 300 400
backpack 500 800
tent 2000 10000
sleeping_bag 1500 5000
binoculars 500 2500
flashlight 100 300
map 50 100
sunscreen 100 200
```

terminal output

```
PS D:\Code Domain\DAA (c++)> .\knapsack knapsack.txt
Pleae Enter the capacity(positive!) of the kanapsack
5000
The maximum value of items in knapsack is :7392720.22 rupees
The contents of the Knapsack are:-
binoculars      500.00grams  2500.00rupees

sunglasses      100.00grams  700.00rupees

wallet  50.00grams  1000.00rupees

camera  800.00grams  5000.00rupees

watch   100.00grams  2500.00rupees

headphones      150.00grams  2000.00rupees

eraser  5.00grams  1.00rupees

silver_bracelet 15.00grams  25000.00rupees

plastic_fork    0.50grams  0.10rupees

feather_duster  2.00grams  5000.00rupees

microchip       0.01grams  1000000.00rupees

gold_coin       0.90grams  100000.00rupees

toothpick       0.10grams  0.01rupees

emerald_pendant 20.00grams  180000.00rupees

glass_marble    0.50grams  5.00rupees

feather 0.02grams  0.05rupees

bronze_medal    10.00grams  1000.00rupees

pebble  0.10grams  0.05rupees
```

```
gold_coin       0.90grams  100000.00rupees

toothpick       0.10grams  0.01rupees

emerald_pendant 20.00grams  180000.00rupees

glass_marble    0.50grams  5.00rupees

feather 0.02grams  0.05rupees

bronze_medal    10.00grams  1000.00rupees

pebble  0.10grams  0.05rupees

ruby_earrings   3.00grams  120000.00rupees

plastic_bottle  0.50grams  1.00rupees

platinum_ring   1.00grams  200000.00rupees

paper_clip      0.01grams  0.01rupees

sapphire_bracelet       4.00grams  150000.00rupees

feather_boa     3.00grams  8000.00rupees

aluminium_ignot 1000.00grams  500000.00rupees

plastic_spoon   1.00grams  1.00rupees

pen     5.00grams  2.00rupees

feather 1.00grams  10.00rupees

diamond_ring    5.00grams  5000000.00rupees

laptop  2000.00grams  60000.00rupees

mobile  200.00grams  30000.00rupees

PS D:\Code Domain\DAA (c++)>
```