**Experiment-3 : City Database**

**AIM:**

Implement the database using an array-based list implementation, and then a linked list implementation. Perform following analysis:

1. Collect running time statistics for each operation in both implementations.
2. What are your conclusions about the relative advantages and disadvantages of the two implementations?
3. Would storing records on the list in alphabetical order by city name speed any of the operations?
4. Would keeping the list in alphabetical order slow any of the operations?\

**CODE:**

#include<bits/stdc++.h>

using namespace std;

int **numOfCities**;

class **cityLinklist**

{

public:

string cityName;

int x;

int y;

cityLinklist \*nextCity;

cityLinklist(string cityName, int x, int y)

{

this->cityName = cityName;

this->x = x;

this->y = y;

}

} \*firstCity = NULL, \*lastCity = NULL;

void **insertAtEnd**(cityLinklist \*&lastCity, string newCity, int x, int y)

{

cityLinklist \*last = new cityLinklist(newCity, x, y);

lastCity->nextCity = last;

lastCity = last;

}

void **printDB()**

{

cityLinklist \*temp = firstCity;

while (temp != NULL)

{

cout << temp->cityName << " " << temp->x << " " << temp->y << "\n";

temp = temp->nextCity;

}

cout << "\n";

}

void **searchCityByName**(string city)

{

cityLinklist \*temp = firstCity;

while (temp->cityName != city && temp != NULL)

{

temp = temp->nextCity;

}

temp == NULL ? cout << "NOT FOUND \n" : cout << "CITY FOUND\n";

}

void **searchCityByCoordinates**(int x, int y)

{

bool found = false;

cityLinklist \*temp = firstCity;

while (temp != NULL)

{

if (temp->x == x && temp->y == y)

{

found = true;

break;

}

temp = temp->nextCity;

}

found == false ? cout << "NOT FOUND \n" : cout << "CITY FOUND\n";

}

void **deleteCityByName**(string city)

{

cityLinklist \*temp = firstCity;

cityLinklist \*header = firstCity;

cityLinklist \*prev = firstCity;

int pos = 0;

cout << city << " is successfully Deleted \n";

if (temp->cityName == city)

{

firstCity = firstCity->nextCity;

return;

}

while (temp->cityName != city)

{

pos++;

temp = temp->nextCity;

}

while (pos--)

{

prev = header;

header = header->nextCity;

}

prev->nextCity = header->nextCity;

header->nextCity = NULL;

}

void **deleteCityByCoordinates**(int x, int y)

{

cityLinklist \*temp = firstCity;

cityLinklist \*header = firstCity;

cityLinklist \*prev = firstCity;

int pos = 0;

cout << "City with " << x << " and " << y << " Coordinates is successfully Deleted \n";

if (temp->x == x && temp->y == y)

{

firstCity = firstCity->nextCity;

return;

}

while (temp->x != x && temp->y != y)

{

pos++;

temp = temp->nextCity;

}

while (pos--)

{

prev = header;

header = header->nextCity;

}

prev->nextCity = header->nextCity;

header->nextCity = NULL;

}

vector<string> **availableCitiesWithinRadius**(int x, int y, int dist)

{

bool found = false;

vector<string> cities;

cityLinklist \*start = firstCity;

while (start != NULL)

{

int x\_dist = x - start->x;

int y\_dist = x - start->y;

int radius = pow(x\_dist, 2) + pow(y\_dist, 2);

double distance = sqrt(radius);

if (distance == 0)

continue;

if (distance <= dist)

{

found = true;

string name = start->cityName;

cities.push\_back(name);

}

start = start->nextCity;

}

if (cities.size() == 0)

{ return {};}

else return cities;

}

**int main()**

{

**cout << "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Welcome To Prayag's MegaCity DataBase \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \n\n";**

cout << "Enter Number of Cities : ";

cin >> numOfCities;

int x, y;

string **cityName**;

cout << "Enter First City Name :- ";

cin >> **cityName**;

cout << "Enter X-Coordinate :- ";

cin >> **x**;

cout << "Enter Y-Coordinate :- ";

cin >> **y**;

lastCity = new cityLinklist(cityName, x, y);

firstCity = lastCity;

for (int i = 1; i < numOfCities; i++) {

cout << "Enter City Name :- ";

cin >> cityName;

cout << "Enter X-Coordinate :- ";

cin >> x;

cout << "Enter Y-Coordinate :- ";

cin >> y;

insertAtEnd(lastCity, cityName, x, y);

}

doYouWantToContinue:

cout << "Enter Your Queries : \n";

cout << " 1. Search a city with Name :\n";

cout << " 2. Search a city with Coordinates :\n";

cout << " 3. Delete a city with Name :\n";

cout << " 4. Delete a city with Coordinates :\n";

cout << " 5. Find all cities within a given distance : \n";

cout << " 6. To View City DataBase : \n";

cout << "Enter Your Choice :\n";

char **choice**;

cin >> **choice**;

switch (choice)

{

case '1': {

cout << "Enter the name of the city you want to search : ";

string cityQuery;

cin >> cityQuery;

searchCityByName(cityQuery);

break;

}

case '2': {

cout << "Enter the Coordinates you want to Find : ";

int xCoordinate, yCoordinate;

cin >> xCoordinate >> yCoordinate;

searchCityByCoordinates(xCoordinate, yCoordinate);

break;

}

case '3': {

cout << "Enter the name of the city you want to delete : ";

string delCity;

cin >> delCity;

deleteCityByName(delCity);

printDB();

break;

}

case '4': {

cout << "Enter the Coordinates of city you want to delete :\n";

int x, y;

cout << "Enter X - Coordinate : ";

cin >> x;

cout << "Enter Y - Coordinate : ";

cin >> y;

deleteCityByCoordinates(x, y);

printDB();

break;

}

case '5': {

int x, y, radius;

cout << "Enter your X-Coordiante : ";

cin >> x;

cout << "Enter your Y-Coordiante : ";

cin >> y;

cout << "Enter the Radius : ";

cin >> radius;

vector<string> citiesWithinRadius = availableCitiesWithinRadius(x, y, radius);

cout << "The Cities within Radius " << radius << " from X = " << x << " and Y = " << y << " are as follows : \n";

for (auto &i : citiesWithinRadius)

{

cout << i << "\n";

}

break;

}

case '6': {

printDB();

break; }

}

cout << "To Quit :- Press 'q' .\n To Search more Queries : Press Any Other Key, \n";

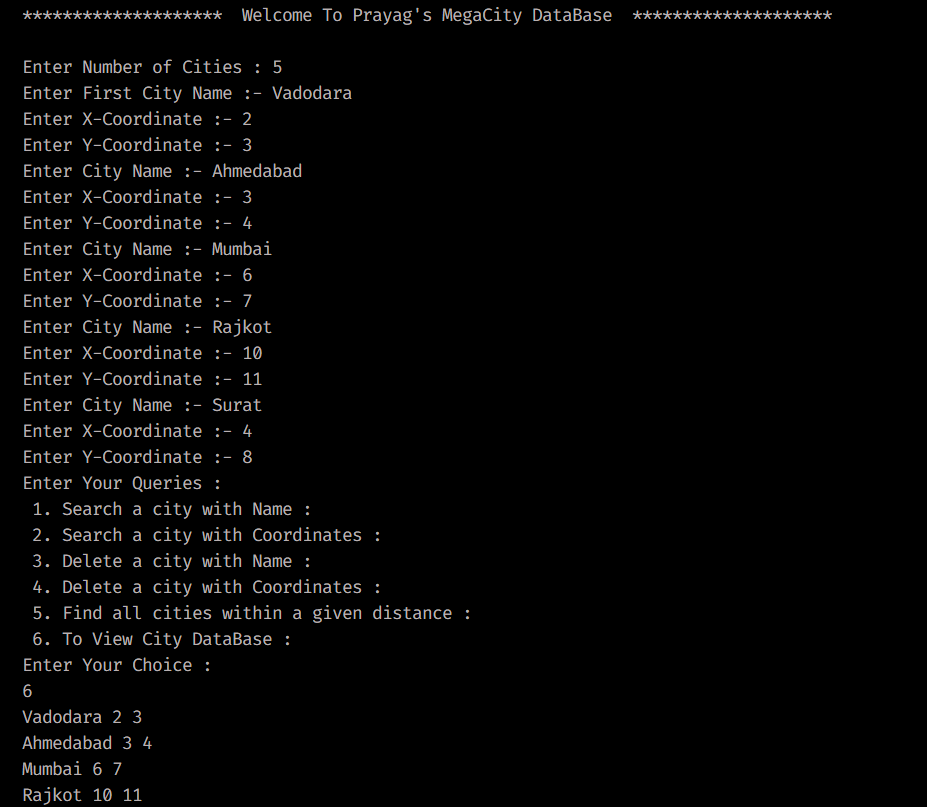
char check;

cin >> check;

if (check != 'q')

goto doYouWantToContinue;

}



**Experiment- 4 Operations using Linked List**

**AIM :**

Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.

What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

**CODE:**

#include <bits/stdc++.h>

using namespace std;

#define ll long long int

string **ans\_str**;

class **Node**

{

public:

int data;

Node \*next;

Node(int data)

{

this->data = data;

this->next = NULL;

}

};

void **insertAtBeginning**(Node \*&head, int data)

{

Node \*temp = new Node(data);

temp->next = head;

head = temp;

}

void **printList**(Node \*&head)

{

Node \*temp = head;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << "\n";

}

Node **\*addition**(Node \*first, Node \*second)

{

int carry = 0;

Node \*ansHead = NULL, \*ansTail = NULL;

while (first != NULL || second != NULL || carry != 0)

{

int val1 = 0, val2 = 0;

if (first != NULL)

val1 = first->data;

if (second != NULL)

val2 = second->data;

ll sum = carry + val1 + val2;

ll dig = sum % 10;

**insertAtBeginning**(ansHead, dig);

ans\_str += to\_string(dig);

carry = sum / 10;

if (first != NULL)

{

first = first->next;

}

if (second != NULL)

{

second = second->next;

}

}

return ansHead;

}

int **main**()

{

cout << "Enter the Number of Digits for 1st Number :- ";

ll **numOfDigitsOfNum1**;

cin >> **numOfDigitsOfNum1**;

Node \*first = NULL, \*second = NULL;

string num1, num2;

enterAgain:

ll curr\_dig;

cin >> curr\_dig;

if (curr\_dig >= 0 && curr\_dig <= 9)

{

first = new Node(curr\_dig);

num1 += to\_string(curr\_dig);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :- ";

goto enterAgain;

}

numOfDigitsOfNum1--;

do

{

cin >> curr\_dig;

if (curr\_dig >= 0 && curr\_dig <= 9)

{

insertAtBeginning(first, curr\_dig);

numOfDigitsOfNum1--;

num1 += to\_string(curr\_dig);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :- ";

}

} while (numOfDigitsOfNum1);

cout << "You Entered the Number : " << num1 << "\n";

cout << "Enter the Number of Digits for 2nd Number :- ";

ll **numOfDigitsOfNum2**, **curr\_dig2**;

enterAgain2:

cin >> **numOfDigitsOfNum2**;

cin >> **curr\_dig2**;

if (curr\_dig2 >= 0 && curr\_dig2 <= 9)

{

second = new Node(curr\_dig2);

num2 += to\_string(curr\_dig2);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :- ";

goto enterAgain2;

}

numOfDigitsOfNum2--;

do

{

cin >> curr\_dig2;

if (curr\_dig2 >= 0 && curr\_dig2 <= 9)

{

**insertAtBeginning**(second, curr\_dig2);

numOfDigitsOfNum2--;

num2 += to\_string(curr\_dig2);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :- ";

}

} while (numOfDigitsOfNum2);

cout << "You Entered the Number : " << num2 << "\n";

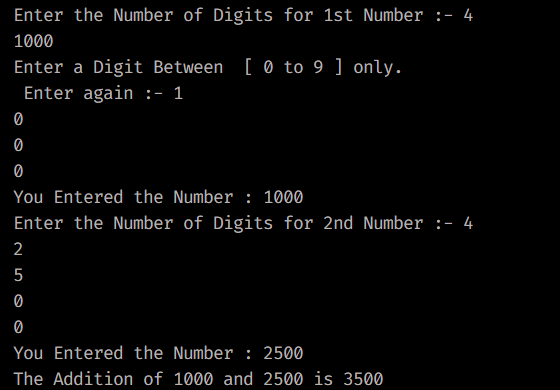
**Node \*ans = addition(first, second);**

reverse(ans\_str.begin(), ans\_str.end());

cout << "The Addition of " << num1 << " and "

<< num2 << " is " << ans\_str;

}



#include <bits/stdc++.h>

using namespace std;

#define ll long long int

string ans\_str;

ll **numOfDigitsOfNum1**, **numOfDigitsOfNum2**, **temp1**, **temp2**;

class **Node**

{

public:

int data;

Node \*next;

Node(int data)

{

this->data = data;

}

};

void **insertAtBeginning**(Node \*&head, int data)

{

Node \*temp = new Node(data);

temp->next = head;

head = temp;

}

void **printList** ( Node \*&head)

{

Node \*temp = head;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << "\n";

}

Node \***getBiggerList** ( Node \*&first, Node \*&second)

{

Node \*temp1 = first;

Node \*temp2 = second;

while (temp1 != NULL) {

if (temp1->data > temp2->data) {

return first;

} else if (temp1->data < temp2->data) {

return second;

}

temp1 = temp1->next;

temp2 = temp2->next;

}

return first;

}

Node \***subtraction**(Node \*first, Node \*second){

Node head1 = \*first;

Node head2 = \*second;

Node \*ansHead = NULL;

if ((numOfDigitsOfNum1 < numOfDigitsOfNum2) || (numOfDigitsOfNum1 == numOfDigitsOfNum2 && second == getBiggerList(first, second))) {

swap(first, second);

}

int diff = 0;

bool borrow = false;

while (first != NULL || second != NULL)

{

int val1 = 0, val2 = 0;

if (first != NULL)

val1 = first->data;

if (second != NULL)

val2 = second->data;

if (borrow)

{

val1 -= 1;

borrow = false;

}

if (first != NULL && second != NULL && val1 < val2)

{

val1 += 10;

borrow = true;

}

diff = (first != NULL ? val1 : 0) - (second != NULL ? val2 : 0);

**insertAtBeginning**(ansHead, diff);

ans\_str += to\_string(diff);

if (first != NULL)

{

first = first->next;

}

if (second != NULL)

{

second = second->next;

}

}

return ansHead;

}

int **main()**

{

cout << "Enter the Number of Digits for 1st Number :- ";

cin >> **numOfDigitsOfNum1**;

temp1 = **numOfDigitsOfNum1**;

Node \*first = NULL, \*second = NULL;

string **num1**, **num2**;

enterAgain:

ll curr\_dig;

cin >> curr\_dig;

if (curr\_dig >= 0 && curr\_dig <= 9)

{

first = new Node(curr\_dig);

num1 += to\_string(curr\_dig);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :-\n ";

goto enterAgain;

}

temp1--;

do

{

cin >> curr\_dig;

if (curr\_dig >= 0 && curr\_dig <= 9)

{

insertAtBeginning(first, curr\_dig);

temp1--;

num1 += to\_string(curr\_dig);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :- \n";

}

} while (temp1);

cout << "You Entered the Number : " << num1 << "\n";

cout << "Enter the Number of Digits for 2nd Number :- ";

ll curr\_dig2;

enterAgain2:

cin >> **temp2**;

cin >> **curr\_dig2;**

if (curr\_dig2 >= 0 && curr\_dig2 <= 9)

{

second = new Node(curr\_dig2);

num2 += to\_string(curr\_dig2);

}

else

{

cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :-\n ";

goto enterAgain2;

}

temp2--;

do

{ cin >> curr\_dig2;

if (curr\_dig2 >= 0 && curr\_dig2 <= 9)

{ insertAtBeginning(second, curr\_dig2);

temp2--;

num2 += to\_string(curr\_dig2);

}

else

{ cout << "Enter a Digit Between [ 0 to 9 ] only.\n Enter again :-\n ";

}

} while (temp2);

cout << "You Entered the Number : " << num2 << "\n";

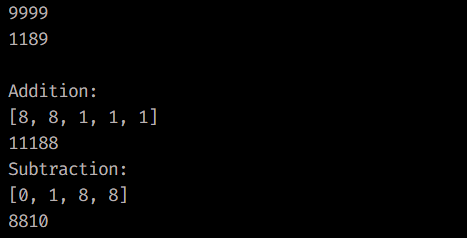
Node \*ans = subtraction(first, second);

reverse(ans\_str.begin(), ans\_str.end());

cout << "The Subtraction of " << num1 << " and " << num2 << " gives "

<< ans\_str << "\n";

}



**Assignment- 5 Interval Scheduling**

**AIM :**

Implement interval scheduling algorithm.Given events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially.

**CODE:**

#include <bits/stdc++.h>

#define ll long long int

using namespace std;

class **TimeTable**

{

public:

string **subject**;

int **startTime**;

int **endTime**;

**TimeTable(string sub, int start, int end)**

{

this->subject = sub;

this->startTime = start;

this->endTime = end;

}

};

bool **sortByEndTime** ( const TimeTable &a , const TimeTable &b )

{

return a.endTime < b.endTime;

}

int **main()**

{

cout << "Enter the number of Subjects : ";

int **numberOfSubjects**;

cin >> **numberOfSubjects**;

**vector<TimeTable> tt , ans;**

**do**

{

enterAgain:

ll **start\_time**, **end\_time**;

string **subject**;

cout << "\nEnter the subject : ";

cin >> **subject**;

cout << "Enter the Start Time : ";

cin >> **start\_time**;

cout << "Enter the End Time : ";

cin >> **end\_time**;

if (end\_time < start\_time)

{

cout << " !! Invalid End Time !! \n It cannot be lesser than start time !! ";

goto enterAgain;

}

TimeTable lecture(subject, start\_time, end\_time);

tt.push\_back(lecture);

numberOfSubjects--;

} **while (numberOfSubjects);**

**sort(tt.begin(), tt.end(), sortByEndTime);**

int cnt = 1;

int subjects = 0;

for (auto &i : tt)

{

if (cnt == 1)

{

ans.push\_back(i);

cnt = 0;

continue;

}

auto lec = tt.begin() + 1;

auto prevIter = prev(lec);

TimeTable prevLecture = \*prevIter;

if (i.startTime >= prevLecture.endTime)

{

subjects++;

ans.push\_back(i);

}

}

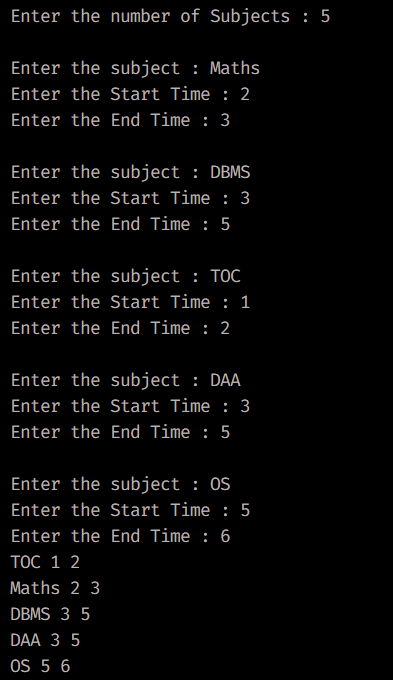
for (auto &i : ans)

{

cout << i.subject << " " << i.startTime << " " << i.endTime << "\n";

}

}



**OBSERVATIONS AND CONCLUSION**

The main aim of this algorithm is to sort according to the End Time of each Period , which gives the Maximum Number of Lectures available.

**Experiment-6 : Stresson’s Multiplication**

**AIM :**

Implement both a standard matrix multiplication algorithm and Strassen’s matrixmultiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtimeequations of the two algorithms. How big must be before Strassen’s algorithm becomes more efficient than the standard algorithm?

**CODE:**

#include <bits/stdc++.h>

using namespace std;

void **printMatrix** ( vector<vector<int>> matrix )

{

for (int row = 0; row <= matrix.size() - 1; row++) {

for (int col = 0; col <= matrix[0].size() - 1; col++) {

cout << " " << matrix[row][col] << " ";

}

cout << "\n";

}

return;

}

void **addMatrix** ( vector<vector<int>> first, vector<vector<int>> second, vector<vector<int>> &product, int index)

{

for (int row = 0; row < index; row++){

for (int col = 0; col < index; col++){

product[row][col] = first[row][col] + second[row][col];

}

}

}

vector<vector<int>> **multiplyMatrix** (vector<vector<int>> first, vector<vector<int>> second)

{

int col1 = first[0].size(), row1 = first.size();

int col2 = second[0].size(), row2 = second.size();

vector<int> rowProduct(col2, 0);

vector<vector<int>> product(row1, rowProduct);

if (col1 == 1)

product[0][0] = first[0][0] \* second[0][0];

else

{

int index = col1 / 2;

vector<int> row(index, 0);

vector<vector<int>> **product\_00(index, row);**

vector<vector<int**>> product\_01(index, row);**

vector<vector<int>> **product\_10(index, row);**

vector<vector<int>> **product\_11(index, row);**

vector<vector<int>> **a00(index, row);**

vector<vector<int>> **a01(index, row);**

vector<vector<int>> **a10(index, row);**

vector<vector<int>> **a11(index, row);**

vector<vector<int>> **b00(index, row);**

vector<vector<int>> **b01(index, row);**

vector<vector<int>> **b10(index, row);**

vector<vector<int>> **b11(index, row);**

for (int row = 0; row < index; row++)

for (int col = 0; col < index; col++)

{

a00[row][col] = first[row][col];

a01[row][col] = first[row][col + index];

a10[row][col] = first[index + row][col];

a11[row][col] = first[row + index][col + index];

b00[row][col] = second[row][col];

b01[row][col] = second[row][col + index];

b10[row][col] = second[index + row][col];

b11[row][col] = second[row + index][col + index];

}

addMatrix ( multiplyMatrix(a00, b00), multiplyMatrix(a01, b10), product\_00, index);

addMatrix ( multiplyMatrix(a00, b01), multiplyMatrix(a01, b11), product\_01, index);

addMatrix ( multiplyMatrix(a10, b00), multiplyMatrix(a11, b10), product\_10, index);

addMatrix ( multiplyMatrix(a10, b01), multiplyMatrix(a11, b11), product\_11, index);

for (int row = 0; row < index; row++)

for (int col = 0; col < index; col++)

{

product[row][col] = product\_00[row][col];

product[row][col + index] = product\_01[row][col];

product[index + row][col] = product\_10[row][col];

product[row + index][col + index] = product\_11[row][col];

}

}

return product;

}

**int main()**

{

**cout << " \n\n ------------------------- Stresson's Multiplcation ------------------------- \n";**

vector<vector<int>> first = {{1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}};

**cout << " ------------------------------- First Matrix ------------------------------- \n";**

printMatrix(first);

cout << "\n"

vector<vector<int>> second = {{1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}};

**cout << " ------------------------------ Second Matrix ------------------------------- \n";**

printMatrix(second);

cout << "\n";

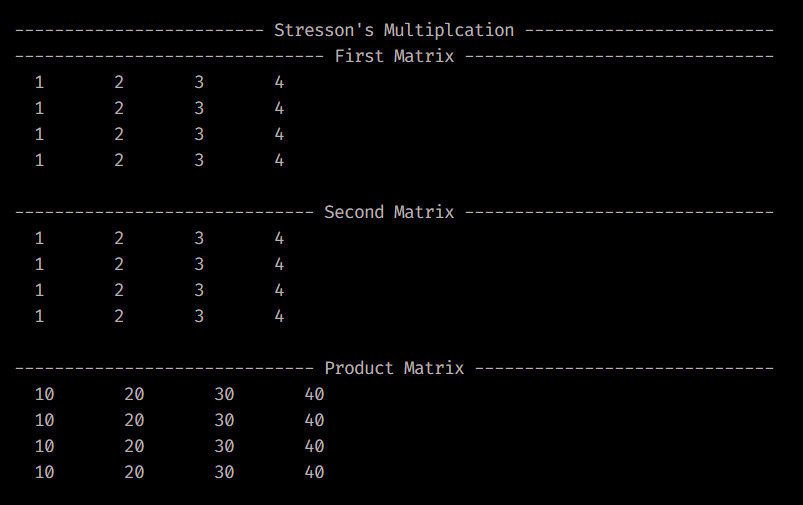
vector<vector<int>> product(multiplyMatrix(first, second));

**cout << " ------------------------------ Product Matrix ------------------------------\n";**

printMatrix(product);

cout << "\n";

}



**Assignment 7 :- Kruskal’s Algorithm**

#include <bits/stdc++.h>

using namespace std;

class **Edge**

{

public:

int src;

int dest;

int wt;

};

class **Graph**

{

public:

int vertices;

int edges;

Edge \*edge;

};

struct **subset**

{

int parent;

int rank;

};

int **find**(subset \*sub, int x)

{

if (sub[x].parent != x)

{

sub[x].parent = find(sub, sub[x].parent);

}

return sub[x].parent;

}

void **Union**(subset \*sub, int x, int y)

{

int i = find(sub, x);

int j = find(sub, y);

if (sub[i].rank < sub[j].rank)

{

sub[i].parent = j;

}

else if (sub[i].rank > sub[i].rank)

{

sub[j].parent = i;

}

else

{

sub[j].parent = i;

sub[i].rank++;

}

}

bool **cmp**(Edge a, Edge b)

{

return a.wt < b.wt;

}

int **Kruskal**(Graph g)

{

int V = g.vertices;

int E = g.edges;

Edge result[V];

int e = 0;

int i = 0;

subset \*sub = new subset[V];

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

{

sub[v].parent = v;

sub[v].rank = 0;

}

while (e < V - 1 && i < E)

{

Edge next\_edge = g.edge[i++];

int x = find(sub, next\_edge.src);

int y = find(sub, next\_edge.dest);

if (x != y)

{

result[e++] = next\_edge;

Union(sub, x, y);

cout << "Added edge: " << next\_edge.src << next\_edge.dest << " (weight " << next\_edge.wt << ")" << endl;

}

}

int cost = 0;

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

{

cost += result[i].wt;

}

cout << "Minimum spanning tree cost: " << cost << endl;

return cost;

}

**int main()**

{

int V, E;

cout << "Enter the number of vertices: ";

cin >> V;

cout << "Enter the number of edges: ";

cin >> E;

**Graph g;**

g.vertices = V;

g.edges = E;

g.edge = new Edge[E];

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

{

int s, d, wt;

cout << "Enter source vertex: ";

cin >> s;

cout << "Enter destination vertex: ";

cin >> d;

cout << "Enter edge weight: ";

cin >> wt;

sort(g.edge, g.edge + E, cmp);

g.edge[i].src = s;

g.edge[i].dest = d;

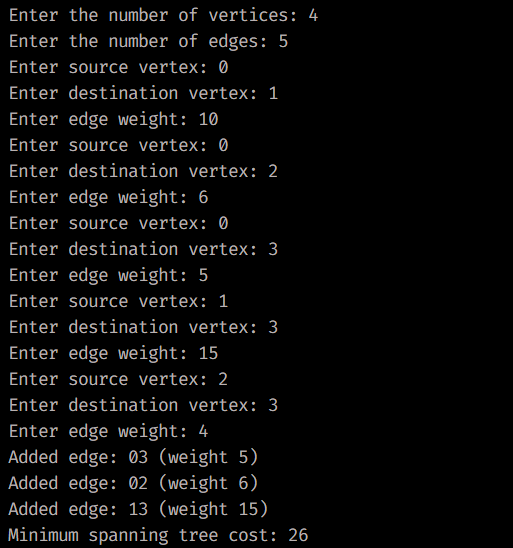
g.edge[i].wt = wt;

}

**int cost = Kruskal(g);**

cout << "The cost of the Minimum Spanning Tree: " << cost << endl;

}



**Assignment 8 :- N – Queens ( BackTracking )**

**AIM :**

Solve the queens’ problem using backtracking. Here, the task is to place chess queens on an x board so that no two queens attack each other. For example, following is a solution for the 4 Queen’ problem.

**CODE:**

#include <bits/stdc++.h>

using namespace std;

bool **isSafe**( vector<vector<bool>> &board, int row, int col) {

int n = board.size();

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

{

if (board[i][col])

return false;

}

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

{

if (board[i][j])

return false;

}

for (int i = row, j = col; i >= 0 && j < n; i--, j++)

{

if (board[i][j])

return false;

}

return true;

}

void **display** ( vector<vector<bool>> &board)

{

int n = board.size();

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

{

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

{

if ( board[i][j] )

**cout << "Q ";**

else

**cout << "X ";**

}

cout << endl;

}

cout << endl;

}

int **queens**(vector<vector<bool>> &board, int row)

{

int n = board.size();

int count = 0;

if (row == n)

{

display(board);

return 1;

}

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

{

if (isSafe(board, row, col))

{

board[row][col] = true;

count += queens(board, row + 1);

board[row][col] = false;

}

}

return count;

}

int main()

{

int n;

cout << "Enter the size of Chess Board you want to use for Placing Queens:";

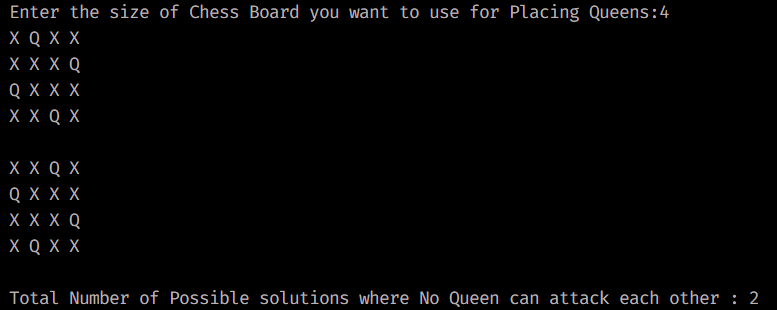
cin >> n;

vector<vector<bool>> board(n, vector<bool>(n, false));

int solutions = queens(board, 0);

cout << "Total Number of Possible solutions where No Queen can attack each other : " << solutions << endl;

}



**Assignment – 9 : Branch and Bound**

**AIM:**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

Solve this problem using branch and bound technique.

#include <bits/stdc++.h>

using namespace std;

const int N = 4;

int **finalPath[N + 1];**

bool **visited[N];**

int **finalResult** = **INT\_MAX;**

void **copyToFinal** ( int currPath[] )

{

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

{

finalPath[i] = currPath[i];

}

finalPath[N] = currPath[0];

}

int **firstMin**(int adj[N][N], int i)

{

int min = INT\_MAX;

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

{

if (adj[i][j] < min && i != j)

{

min = adj[i][j];

}

}

return min;

}

int **secondMin**(int adj[N][N], int i)

{

int first = INT\_MAX;

int second = INT\_MAX;

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

{

if (i == j)

{

continue;

}

if (adj[i][j] <= first)

{

second = first;

first = adj[i][j];

}

else if (adj[i][j] <= second && adj[i][j] != first)

{

second = adj[i][j];

}

}

return second;

}

void **TSPRec** ( int adj[N][N], int lowerbound, int currWeight, int level, int currPath[] )

{

if (level == N)

{

if (adj[currPath[level - 1]][currPath[0]] != 0)

{

int currResult = currWeight + adj[currPath[level - 1]][currPath[0]];

if (currResult < finalResult)

{

copyToFinal(currPath);

finalResult = currResult;

}

}

return;

}

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

{

if ( adj[currPath[ level – 1 ]][ i ] != 0 && visited[i] == false)

{

int temp = lowerbound;

currWeight += adj[currPath[level - 1]][i];

if (level == 1) {

lowerbound -= ((firstMin(adj, currPath[level - 1]) + firstMin(adj, i)) / 2);

}

else{

lowerbound -= ((secondMin(adj, currPath[level - 1]) + firstMin(adj, i)) / 2);

}

if (lowerbound + currWeight < finalResult) {

currPath[level] = i;

visited[i] = true;

TSPRec(adj, lowerbound, currWeight, level + 1, currPath);

}

currWeight -= adj[currPath[level - 1]][i];

lowerbound = temp;

**memset(visited, false, sizeof(visited));**

for (int j = 0; j <= level - 1; j++)

{

visited[currPath[j]] = true;

}

}

}

}

void **TSP** ( int adj[ N ][ N ] )

{

int currPath[N + 1];

int lowerbound = 0;

memset(currPath, -1, sizeof(currPath));

memset(visited, 0, sizeof(visited));

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

{

lowerbound += (firstMin(adj, i)) + secondMin(adj, i);

}

lowerbound = (lowerbound & 1) ? lowerbound / 2 + 1 : lowerbound / 2;

visited[0] = true;

currPath[0] = 0;

TSPRec(adj, lowerbound, 0, 1, currPath);

}

**int main()**

{

int adj[4][4] =

{

{ 2, 23, 15, 21 },

{ 10, 1, 35, 23 },

{ 15, 39, 1, 25 },

{ 18, 25, 30, 4 }

};

TSP(adj);

cout << "\n\nMinimum cost : " << finalResult << endl;

cout << "Optimal path is as traced : ";

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

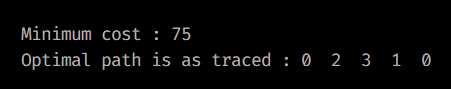
{

cout << finalPath[i] << " ";

}

cout << "\n\n";

}



**Assignment – 10 : General Problems**

**AIM :**

To design and solve given problems using different algorithmic approaches and analyze their complexity.

1. Your friends are starting a security company that needs to obtain licenses for different pieces of cryptographic software. Due to regulations, they can onlyobtain these licenses at the rate of at most one per month.Each license is currently selling for a price of $100. However, they are all becoming more expensive according to exponential growth curves: inparticular, the cost of license increases by a factor of each month, where is a given parameter. This means that if license is purchased months from now, it will cost. We will assume that all the price growth rates aredistinct; that is, for licenses (even though they start at the sameprice of $100).

The question is: Given that the company can only buy at most one licensea month, in which order should it buy the licenses so that the total amount ofmoney it spends is as small as possible?

Give an algorithm that takes the rates of price growth , andcomputes an order in which to buy the licenses so that the total amount ofmoney spent is minimized. The running time of your algorithm should bepolynomial in .

**CODE:**

#include <bits/stdc++.h>

using namespace std;

struct **license**

{

double **growth\_rate**;

int **index**;

};

bool **compareLicenses**(const license &license1, const license &license2)

{

return license1.growth\_rate < license2.growth\_rate;

}

vector<int> **findLicenseOrder**(const vector<double> &growth\_rates)

{

int n = growth\_rates.size();

vector<license> licenses(n);

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

{

licenses[i].growth\_rate = growth\_rates[i];

licenses[i].index = i + 1;

}

**sort ( licenses.begin(), licenses.end(), compareLicenses );**

vector<int> **order(n);**

double total\_cost = 0.0;

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

{

order[i] = licenses[i].index;

total\_cost += 100.0 \* licenses[i].growth\_rate;

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

{

licenses[j].growth\_rate \*= licenses[i].growth\_rate;

}

}

cout << "\n Total cost will be : $ " << total\_cost << endl;

return order;

}

**int main()**

{

vector<double> growth\_rates = {1.7, 2.7, 1.5, 1.8, 2.4};

vector<int> order = findLicenseOrder(growth\_rates);

cout << " \nThe Licenses will be in the following order : \n ";

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

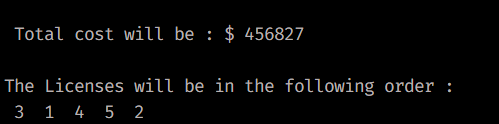
{

cout << order[i] << " ";

}

cout << endl;

}



1. Suppose you are given an array with entries, with each entry holding adistinct number. You are told that the sequence of values is unimodal.That is, for some index between and , the values in the array entries increase up to position in and then decrease the remainder of the way until position . (So if you were to draw a plot with the array position on the-axis and the value of the entry on the -axis, the plotted points wouldrise until -value , where they’d achieve their maximum value, and then fall fromthere on). You’d like to find the “peak entry” without having to read the entirearray - in fact, by reading as few entries of as possible. Show how to find the entry by reading at most entries of .

**CODE :**

#include <iostream>

#include <vector>

using namespace std;

int **find\_peak\_entry**(vector<int> &A)

{

int n = A.size();

int left = 0, right = n - 1;

while (left < right)

{

int mid = left + (right - left) / 2;

if (A[mid] < A[mid + 1])

{

left = mid + 1;

}

else

{

right = mid;

}

}

return left;

}

void **printArr**(vector<int> &A)

{

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

{

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

}

}

**int main()**

{

vector<int> A = {1, 3, 5, 7, 9, 19, 8, 6, 4, 2};

int peak\_entry = find\_peak\_entry(A);

cout << " \nPeak entry in the array : { ";

printArr(A);

cout << " } is "

<< A[peak\_entry] << "\n\n";

}

