

Experiment-3 : City Database

AIM:

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

- a) Collect running time statistics for each operation in both implementations.
- b) What are your conclusions about the relative advantages and disadvantages of the two implementations?
- c) Would storing records on the list in alphabetical order by city name speed any of the operations?
- d) 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;
}
```

OUTPUT

```
***** Welcome To Prayag's MegaCity DataBase *****

Enter Number of Cities : 5
Enter First City Name :- Vadodara
Enter X-Coordinate :- 2
Enter Y-Coordinate :- 3
Enter City Name :- Ahmedabad
Enter X-Coordinate :- 3
Enter Y-Coordinate :- 4
Enter City Name :- Mumbai
Enter X-Coordinate :- 6
Enter Y-Coordinate :- 7
Enter City Name :- Rajkot
Enter X-Coordinate :- 10
Enter Y-Coordinate :- 11
Enter City Name :- Surat
Enter X-Coordinate :- 4
Enter Y-Coordinate :- 8
Enter Your Queries :
    1. Search a city with Name :
    2. Search a city with Coordinates :
    3. Delete a city with Name :
    4. Delete a city with Coordinates :
    5. Find all cities within a given distance :
    6. To View City DataBase :
Enter Your Choice :
6
Vadodara 2 3
Ahmedabad 3 4
Mumbai 6 7
Rajkot 10 11
```

CONCLUSION :-

Advantages of using a LinkedList :

1. **Dynamic size:** Linked lists have a dynamic size, making it easier to add or remove elements as the database size changes.
2. **More flexible :** Linked lists are more flexible than arrays and can perform more complex operations.
3. **Memory efficiency :** Linked lists are more memory-efficient for smaller databases, as they only require memory for the data stored.

Disadvantages of using a LinkedList :

1. **Slow access to data:** Linked lists provide slower access to data as each node requires a pointer to the next node, which can be time-consuming.
2. **Requires more memory:** Linked lists require more memory than arrays as each node requires an additional pointer.
3. **More complex implementation:** Linked lists are generally more complex to implement than arrays as they require more operations to add, remove, or search elements.

Experiment- 4 Operations using Linked List

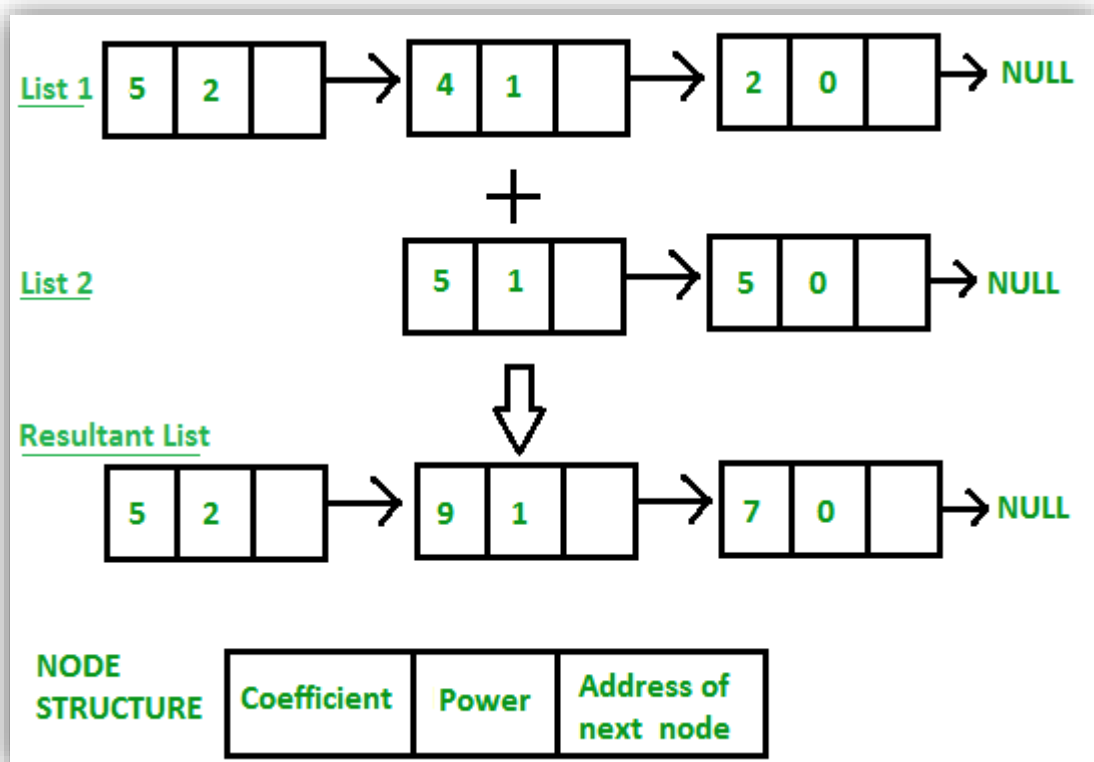
AIM:

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?

ALGORITHM

1. Traverse the two linked lists in order to add preceding zeros in case a list is having lesser digits than the other one.
2. Start from the head node of both lists and call a recursive function for the next nodes.
3. Continue it till the end of the lists.
4. Creates a node for current digits sum and returns the carry.



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;
        int sum = carry + val1 + val2;
        int 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;
}
```



```
Enter the Number of Digits for 1st Number :- 4
1000
Enter a Digit Between [ 0 to 9 ] only.
Enter again :- 1
0
0
0
You Entered the Number : 1000
Enter the Number of Digits for 2nd Number :- 4
2
5
0
0
You Entered the Number : 2500
The Addition of 1000 and 2500 is 3500
```

```
#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";  
}
```

```
9999
```

```
1189
```

```
Addition:
```

```
[8, 8, 1, 1, 1]
```

```
11188
```

```
Subtraction:
```

```
[0, 1, 8, 8]
```

```
8810
```

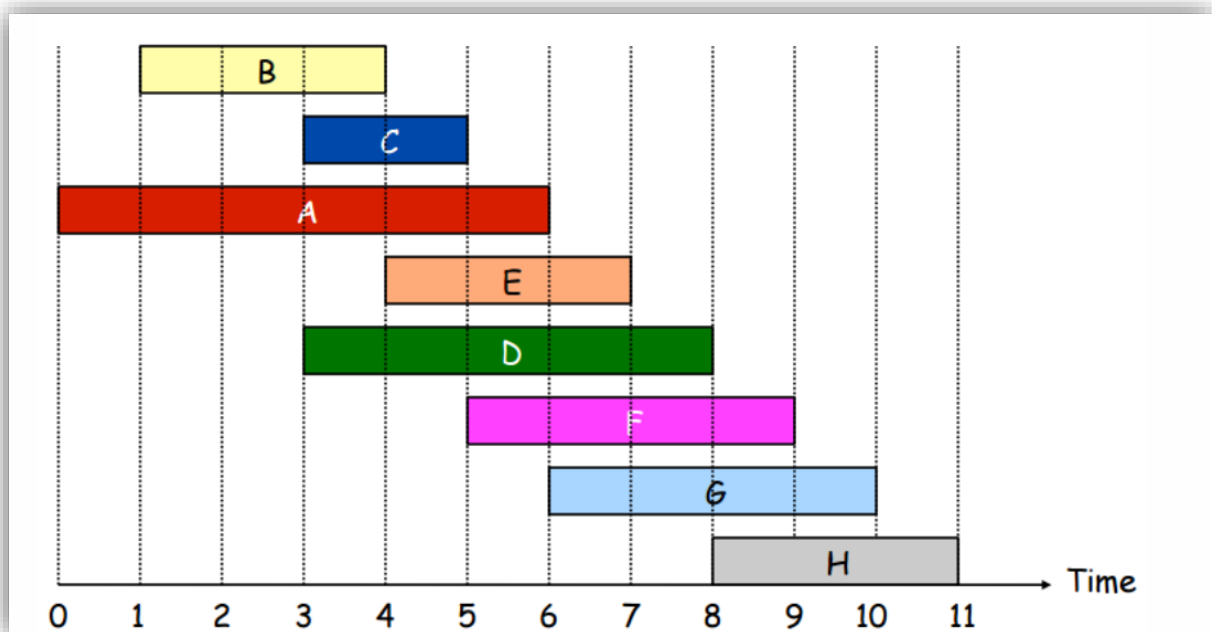
Assignment- 5 Interval Scheduling

AIM :

Implement interval scheduling algorithm. Given N 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.

ALGORITHM:

- 1) Sort the Subjects according to their End Times.
- 2) Add as much as possible subjects by checking if the next subject's Start Time is less than or equal to End Time of Previous Subject.



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";
}
}
```

```
Enter the number of Subjects : 5

Enter the subject : Maths
Enter the Start Time : 2
Enter the End Time : 3

Enter the subject : DBMS
Enter the Start Time : 3
Enter the End Time : 5

Enter the subject : TOC
Enter the Start Time : 1
Enter the End Time : 2

Enter the subject : DAA
Enter the Start Time : 3
Enter the End Time : 5

Enter the subject : OS
Enter the Start Time : 5
Enter the End Time : 6
TOC 1 2
Maths 2 3
DBMS 3 5
DAA 3 5
OS 5 6
```

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.

In conclusion, the interval scheduling algorithm is a useful tool for finding an optimal schedule that includes as many events as possible. The algorithm involves sorting the events based on their ending times and selecting the earliest ending event. Then, it checks if any other events overlap with the selected event, and if not, adds it to the schedule. This process is repeated until all events are scheduled. This algorithm has a time complexity of **$O(N \log N)$** due to the sorting step, but it can provide an optimal solution for scheduling events. It is important to note that this algorithm assumes that each event can only be selected in its entirety, which may not always be the case. Overall, the interval scheduling algorithm can be a valuable tool in various scheduling applications, such as job scheduling or course scheduling.

Time Complexity : $O(N * \log N)$

Experiment- 6 : Strassen's Multiplication

AIM :

Implement both a standard $O(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?

CODE:

```
#include <bits/stdc++.h>

using namespace std;

#define ROW_1 4
#define COL_1 4

#define ROW_2 4
#define COL_2 4

void printMatrix(vector<vector<int>> matrix)
{
    for (int i = 0; i <= matrix[0].size() - 1; i++)
    {
        for (int j = 0; j <= matrix.size() - 1; j++)
        {
            cout << matrix[i][j] << " ";
        }
        cout << "\n";
    }
    cout << "\n";
    return;
```

```
}
```

```
vector<vector<int>>
```

```
add_matrix(vector<vector<int>> first,
            vector<vector<int>> second, int half_ind,
            int multiplyWith = 1)
{
    for (int i = 0; i < half_ind; i++)
        for (int j = 0; j < half_ind; j++)
            first[i][j] = first[i][j] + (multiplyWith * second[i][j]);
    return first;
}
```

```
vector<vector<int>> multiply_matrix(vector<vector<int>> first, vector<vector<int>>
second)
```

```
{
    int col_1 = first[0].size();
    int row_1 = first.size();
    int col_2 = second[0].size();
    int row_2 = second.size();

    if (col_1 != row_2)
    {
        cout << "\nNumber of Rows in A is not equal to Number of columns in B\n";
        return {};
    }
}
```

```
vector<int> prod_matrix_row(col_2, 0);
vector<vector<int>> prod_matrix(row_1,
                                prod_matrix_row);
```

```
if (col_1 == 1)
    prod_matrix[0][0] = first[0][0] * second[0][0];
else
{
    int half_ind = col_1 / 2;

    vector<int> row_vector(half_ind, 0);

    vector<vector<int>> a00(half_ind, row_vector);
    vector<vector<int>> a01(half_ind, row_vector);
    vector<vector<int>> a10(half_ind, row_vector);
    vector<vector<int>> a11(half_ind, row_vector);
    vector<vector<int>> b00(half_ind, row_vector);
    vector<vector<int>> b01(half_ind, row_vector);
    vector<vector<int>> b10(half_ind, row_vector);
    vector<vector<int>> b11(half_ind, row_vector);

    for (int i = 0; i < half_ind; i++)
        for (int j = 0; j < half_ind; j++)
        {
            a00[i][j] = first[i][j];
            a01[i][j] = first[i][j + half_ind];
            a10[i][j] = first[half_ind + i][j];
            a11[i][j] = first[i + half_ind]
                [j + half_ind];
            b00[i][j] = second[i][j];
            b01[i][j] = second[i][j + half_ind];
            b10[i][j] = second[half_ind + i][j];
```

```
        b11[i][j] = second[i + half_ind]
            [j + half_ind];
    }

    vector<vector<int>> p(multiply_matrix(
        a00, add_matrix(b01, b11, half_ind, -1)));
    vector<vector<int>> q(multiply_matrix(
        add_matrix(a00, a01, half_ind), b11));
    vector<vector<int>> r(multiply_matrix(
        add_matrix(a10, a11, half_ind), b00));
    vector<vector<int>> s(multiply_matrix(
        a11, add_matrix(b10, b00, half_ind, -1)));
    vector<vector<int>> t(multiply_matrix(
        add_matrix(a00, a11, half_ind),
        add_matrix(b00, b11, half_ind)));
    vector<vector<int>> u(multiply_matrix(
        add_matrix(a01, a11, half_ind, -1),
        add_matrix(b10, b11, half_ind)));
    vector<vector<int>> v(multiply_matrix(
        add_matrix(a00, a10, half_ind, -1),
        add_matrix(b00, b01, half_ind)));

    vector<vector<int>> prod_matrix_00(add_matrix(
        add_matrix(add_matrix(t, s, half_ind), u,
            half_ind),
        q, half_ind, -1));
    vector<vector<int>> prod_matrix_01(
        add_matrix(p, q, half_ind));
    vector<vector<int>> prod_matrix_10(
```



```
        add_matrix(r, s, half_ind));
vector<vector<int>> prod_matrix_11(add_matrix(
    add_matrix(add_matrix(t, p, half_ind), r,
        half_ind, -1),
    v, half_ind, -1));

for (int i = 0; i < half_ind; i++)
    for (int j = 0; j < half_ind; j++)
    {
        prod_matrix[i][j] = prod_matrix_00[i][j];
        prod_matrix[i][j + half_ind] = prod_matrix_01[i][j];
        prod_matrix[half_ind + i][j] = prod_matrix_10[i][j];
        prod_matrix[i + half_ind]
            [j + half_ind] = prod_matrix_11[i][j];
    }

a00.clear();a01.clear();a10.clear();a11.clear();
b00.clear();b01.clear();b10.clear();b11.clear();

p.clear();q.clear();r.clear();s.clear();
t.clear();u.clear();v.clear();

prod_matrix_00.clear();prod_matrix_01.clear();
prod_matrix_10.clear();prod_matrix_11.clear();
}
return prod_matrix;
}

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>> prod_matrix(
        multiply_matrix(first, second));

    cout << " ----- Product Matrix ----- \n";
    printMatrix(prod_matrix);
    cout << "\n";
}}
```

OUTPUT

----- Strassen's Multiplication -----			
----- First Matrix -----			
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
----- Second Matrix -----			
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
----- Product Matrix -----			
10	20	30	40
10	20	30	40
10	20	30	40
10	20	30	40

Conclusion

Strassen's algorithm for matrix multiplication is a divide-and-conquer algorithm that can reduce the number of multiplications required to compute the product of two matrices. The algorithm recursively divides the matrices into smaller submatrices and computes the product using a set of mathematical equations. Strassen's algorithm can reduce the time complexity of matrix multiplication from **$O(n^3)$ to $O(n^{\log_2(7)}) \approx O(n^{2.81})$** . This can be significant for large matrices where the standard matrix multiplication algorithm can be prohibitively expensive.

In summary, Strassen's algorithm for matrix multiplication can be an efficient alternative to the standard algorithm for large matrices. It has a time complexity of $O(n^{\log_2(7)}) \approx O(n^{2.81})$, which is faster than the standard algorithm's time complexity of $O(n^3)$.

Time Complexity using Recursive Method : **$O(N^3)$**

Time Complexity using Strassen's Multiplication : **$O(N^{2.81})$**

Assignment 7 :- Floyd Warshall Algorithm

AIM :

Implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem. You are given a weighted diagraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw} between any node v and node w . Find the cheapest path from every node to every other node. Edges may have negative weights. Consider the following test case to check your algorithm.

THEORY :

The Floyd Warshall Algorithm is for solving all pairs of shortest-path problems. The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed Graph.

It is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm follows the dynamic programming approach to find the shortest path.

ALGORITHM :

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

CODE :

```
#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[j].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 floydWarshall ( 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 = floydWarshall(g);

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

}
```

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter source vertex: 0
Enter destination vertex: 1
Enter edge weight: 10
Enter source vertex: 0
Enter destination vertex: 2
Enter edge weight: 6
Enter source vertex: 0
Enter destination vertex: 3
Enter edge weight: 5
Enter source vertex: 1
Enter destination vertex: 3
Enter edge weight: 15
Enter source vertex: 2
Enter destination vertex: 3
Enter edge weight: 4
Added edge: 03 (weight 5)
Added edge: 02 (weight 6)
Added edge: 13 (weight 15)
Minimum spanning tree cost: 26
```


CONCLUSION

Time Complexity: $O(V^3)$

Auxiliary Space: $O(V^2)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle the maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

Assignment 8 :- N – Queens (BackTracking)

AIM :

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

THEORY :

The N Queens problem is a classic problem in computer science and mathematics. The problem is to place N chess queens on an N x N chessboard in such a way that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal.

There are various algorithms to solve the N Queens problem. One of the most common algorithms is the backtracking algorithm. The backtracking algorithm works by placing queens on the board one by one and checking if the placement is safe. If the placement is safe, then the algorithm continues with the next queen. If the placement is not safe, then the algorithm backtracks and tries a different position for the previous queen.

Algorithm

1. Initialize an empty chessboard of size $N \times N$.
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.

CODE:

```
#include <bits/stdc++.h>

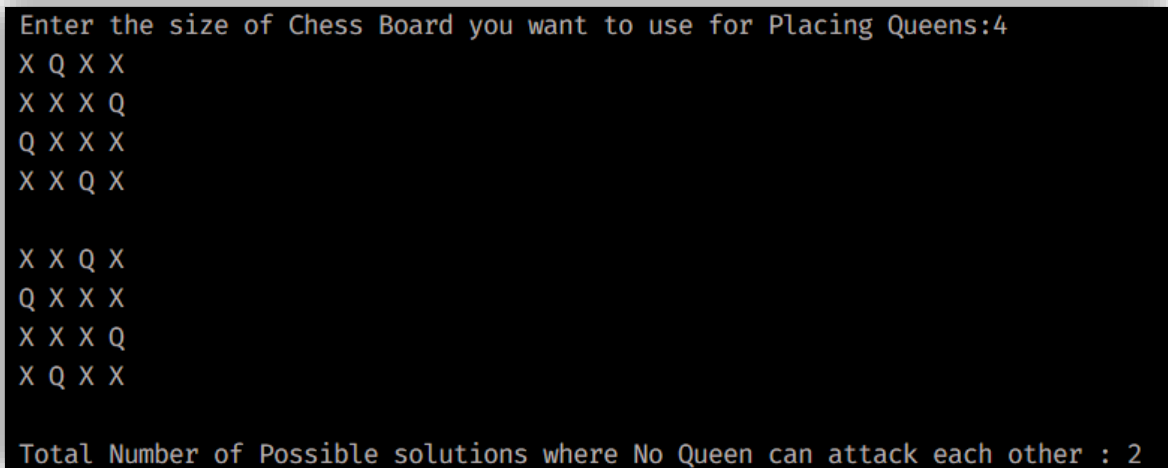
using namespace std;

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;
}
```



```
Enter the size of Chess Board you want to use for Placing Queens:4
X Q X X
X X X Q
Q X X X
X X Q X

X X Q X
Q X X X
X X X Q
X Q X X

Total Number of Possible solutions where No Queen can attack each other : 2
```

CONCLUSION:

Time complexity :

For finding a single solution where the first queen Q has been assigned the first column and can be put on N positions, the second queen has been assigned the second column and would choose from N-1 possible positions and so on; the time complexity is $O(N * (N - 1) * (N - 2) * \dots 1)$. i.e The worst-case time complexity is $O(N!)$.

Thus, for finding all the solutions to the N Queens problem the time complexity runs in **polynomial time**.

Assignment – 9 :

Travelling Salesman Problem (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.

THEORY:

Traveling salesman problem, an optimization problem in graph theory in which the nodes (cities) of a graph are connected by directed edges (routes), where the weight of an edge indicates the distance between two cities. The problem is to find a path that visits each city once, returns to the starting city, and minimizes the distance traveled. The only known general solution algorithm that guarantees the shortest path requires a solution time that grows exponentially with the problem size (i.e., the number of cities). This is an example of an NP-complete problem (from nonpolynomial), for which no known efficient (i.e., polynomial time) algorithm exists.

ALGORITHM

Algorithm

1. Travelling salesman problem takes a graph $G \{V, E\}$ as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
2. The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
3. The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
4. Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
5. Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A .

CODE:

```
#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";  
}
```

```
Minimum cost : 75  
Optimal path is as traced : 0 2 3 1 0
```

CONCLUSION:

The time complexity of the Traveling Salesman Problem (TSP) using Branch and Bound algorithm is **$O((n-1)!)$** , where n is the number of cities or nodes in the problem. However, the actual time taken to solve the problem may be much lower depending on the structure and size of the problem.

Time Complexity : $O(N!)$

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 n different pieces of cryptographic software. Due to regulations, they can only obtain 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: in particular, the cost of license j increases by a factor of $r_j > 1$ each month, where r_j is a given parameter. This means that if license j is purchased t months from now, it will cost $100 r_j^t$. We will assume that all the price growth rates are distinct; that is, $r_i \neq r_j$ for licenses $i \neq j$ (even though they start at the same price of \$100). The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible?
Give an algorithm that takes the n rates of price growth r_1, r_2, \dots, r_n , and computes an order in which to buy the licenses so that the total amount of money spent is minimized. The running time of your algorithm should be polynomial in n .

CODE:

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
struct license
```

```
{
```

```
    double growth_rate;
```

```
    int index;
```

```
};
```

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

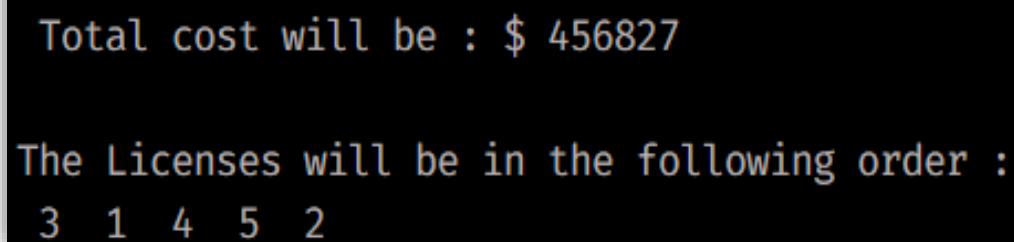
```
{
```

```
    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;  
}
```

OUTPUT :



```
Total cost will be : $ 456827  
  
The Licenses will be in the following order :  
3 1 4 5 2
```

2. Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \dots, A[n]$ is unimodal. That is, for some index p between 1 and n , the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n . (So if you were to draw a plot with the array position j on the x -axis and the value of the entry $A[j]$ on the y -axis, the plotted points would rise until x -value p , where they'd achieve their maximum value, and then fall from there on). You'd like to find the "peak entry"

p without having to read the entire array - in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A

PSEUDO-ALGORITHM :

In an unimodal array, the elements are sorted in a specific order until a certain point, after which the order is reversed. A peak element is an element in the array that is greater than or equal to its neighboring elements.

Binary search is an efficient algorithm for finding a peak element in an unimodal array. The algorithm works by comparing the middle element of the array with its neighbors. If the middle element is greater than both its neighbors, it is a peak element. If it is less than its left neighbor, the algorithm searches the left half of the array, because a peak element must exist on the left half. If it is less than its right neighbor, the algorithm searches the right half of the array.

The binary search algorithm repeatedly halves the search space until it finds the peak element. Since the search space is halved at each step, the time complexity of the binary search algorithm is $O(\log n)$, where n is the size of the array.

CODE :

```
#include<bits/stdc++.h>

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";
}
```

OUTPUT

```
Peak entry in the array : { 1 3 5 7 9 19 8 6 4 2 } is 19
```

CONCLUSION

Time Complexity : $O(\log N)$

the binary search algorithm is an efficient way to find a peak element in an unimodal array with a time complexity of **$O(\log n)$** . It is a useful algorithm in computer science and can be applied in various domains such as image processing, signal processing, and machine learning.