



A vibrant neon-themed graphic design on a dark background. The central focus is the text "PARKING PLANNING" in a bold, white, sans-serif font, enclosed within a white neon-outlined tag-like shape. Surrounding this central element are various glowing neon shapes in pink and cyan, including triangles, lines, and a large arrow pointing towards the text. To the right of the central tag is a cyan-outlined speech bubble containing a heart symbol. The overall aesthetic is modern and digital.

PARKING PLANNING

TABLE OF CONTENTS

01

MOTIVATION

Why we choose
Parking Planning ?

03

PSEUDOCODE AND ANALYSIS

Discussing the
Pseudocode and
analysing the
algorithm

02

PROBLEM DEFINITION

Formal description
of problem
statement

04

IMPLEMENTATION

Executing the
algorithm in C++
programming
language



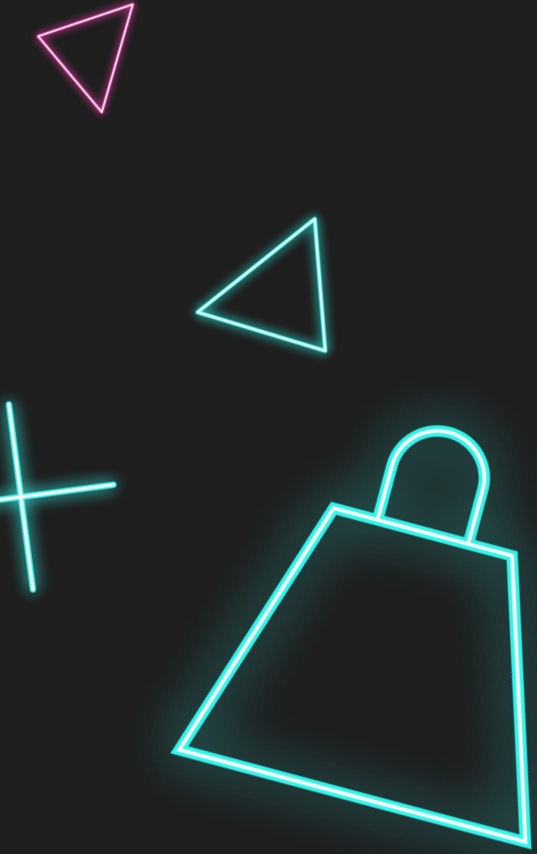
MOTIVATION



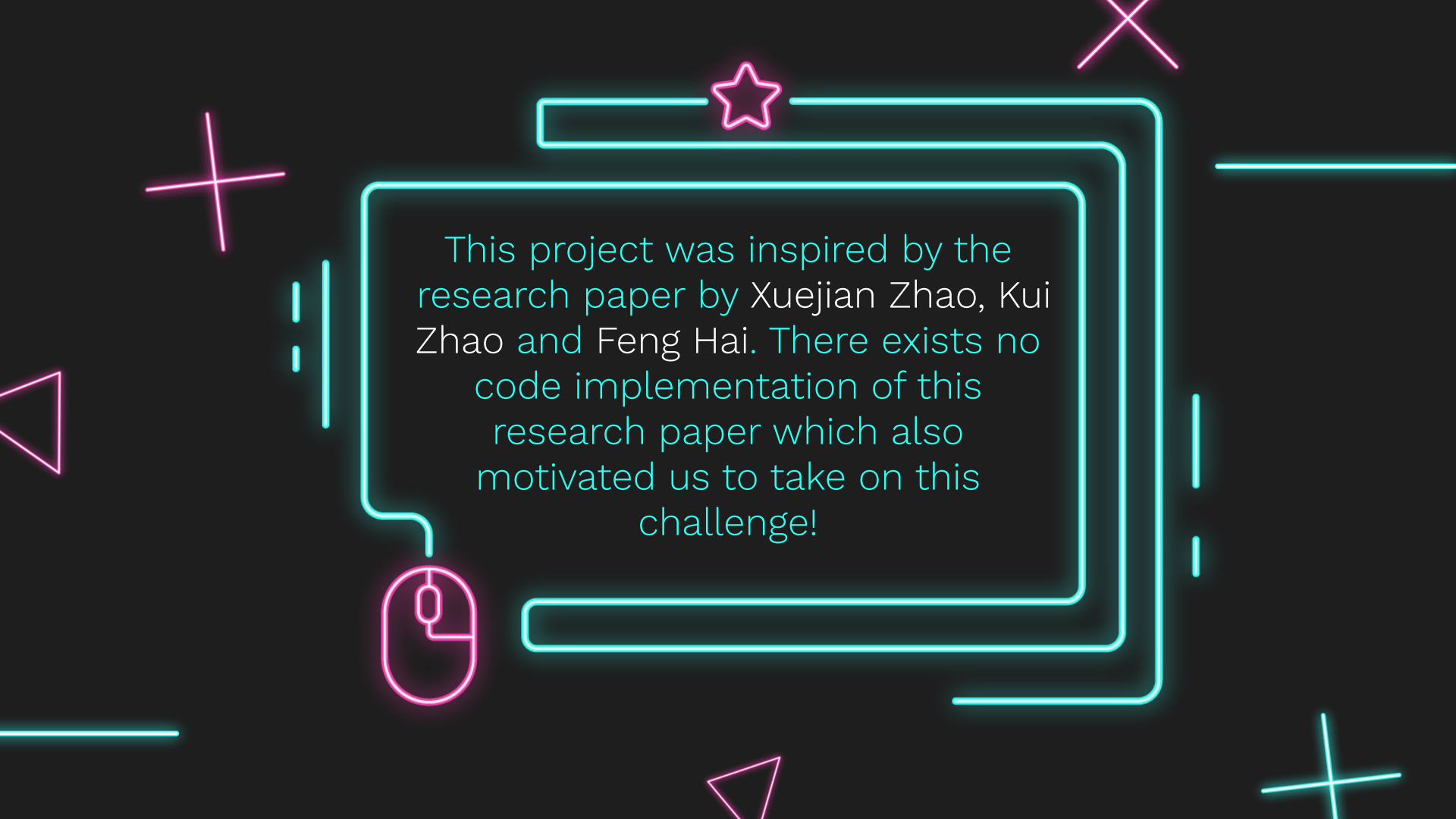
MOTIVATION

- The increasing number of vehicles leading to parking problems
- Introducing Smart Parking to ease off parking problems
- Smart Parking requires Parking Planning to make best possible use of resources
- Preliminary research revealed that a real life feasible code doesn't exist



A collection of glowing geometric shapes on the left side of the slide: a small magenta triangle at the top left, a cyan triangle below it, a cyan plus sign to the left of the shopping bag, and a large cyan shopping bag at the bottom left.

The project is an attempt to solve the parking planning program to give timely and efficient guide information to vehicles for a real time smart parking system.



This project was inspired by the research paper by Xuejian Zhao, Kui Zhao and Feng Hai. There exists no code implementation of this research paper which also motivated us to take on this challenge!



PROBLEM DEFINITION



PROBLEM DEFINITION

We transform the parking planning problem into a kind of
LINEAR ASSIGNMENT problem.

Vehicles \leftrightarrow JOBS

Parking spaces \leftrightarrow AGENTS

Distances between
vehicles and parking spaces \leftrightarrow COSTS FOR
AGENTS DOING JOBS

PROBLEM DEFINITION

INPUT

- P , denoting the set of all vehicles having parking query in the queue (Size : M)
- S , denoting the set of all available parking spaces included in the smart parking system (Size : N)
- D ; denoting the set of d_{ij} , and d_{ij} is the distance between the vehicle p_i ($p_i \in P$) and the parking space s_j ($s_j \in S$) (Size : $M*N$)

- Best possible assignment
- Total cost for that assignment

OUTPUT

PROBLEM DEFINITION

ADDITIONAL INFORMATION

- The distances between vehicle and parking spaces are already known. Real life implementation will require the use of GPS for finding this data
- Size of $M \lll N$
where M, number of vehicles in queue is in hundreds and N is total parking spots in millions
- If number of vehicles in the system is more than the queue then we will take first M vehicles on a first come first serve basis. Our algorithm works for each queue of M size which is also the size of the total vehicles in input.



PSEUDOCODE AND ANALYSIS



GREEDY APPROACH

- In this method we guide the vehicle, which is querying parking space, to the nearest available parking space.
- This approach fails miserably but is the quickest. This is actually real-time allotment.
- This gives us local optimization but fails for global optimization and is gives really high cost for our problem
- Our Modern approach will queue a fixed no of cars on first come first serve basis and apply a modified version of hungarian algorithm or linear problem assignment algorithm which is faster than the classical one.
- This way we can improve the time complexity of classical Hungarian Algorithm and optimize the cost for our greedy approach

THE HUNGARIAN ALGORITHM

- The Hungarian Algorithm consists of FOUR steps
- Steps 1 and 2 are executed once only
- Steps 3 and 4 are repeated until an optimal assignment is found
- INPUT : (N X N) Square Matrix with non-negative elements only

THE HUNGARIAN ALGORITHM

(classical approach)

For each row, find the lowest element and subtract it from each element in that row

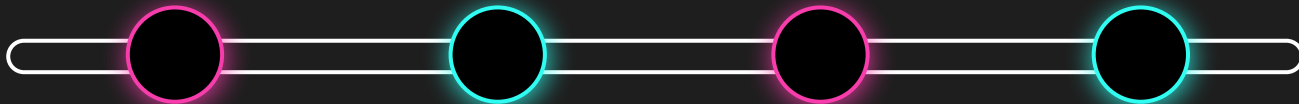
SUBTRACT ROW
MINIMA

02

If n lines are required, an optimal assignment exists and algorithm stops; else step 4

COVER ALL ZEROS
WITH A MINIMUM
NUMBER OF LINES

04



01

SUBTRACT COLUMN
MINIMA

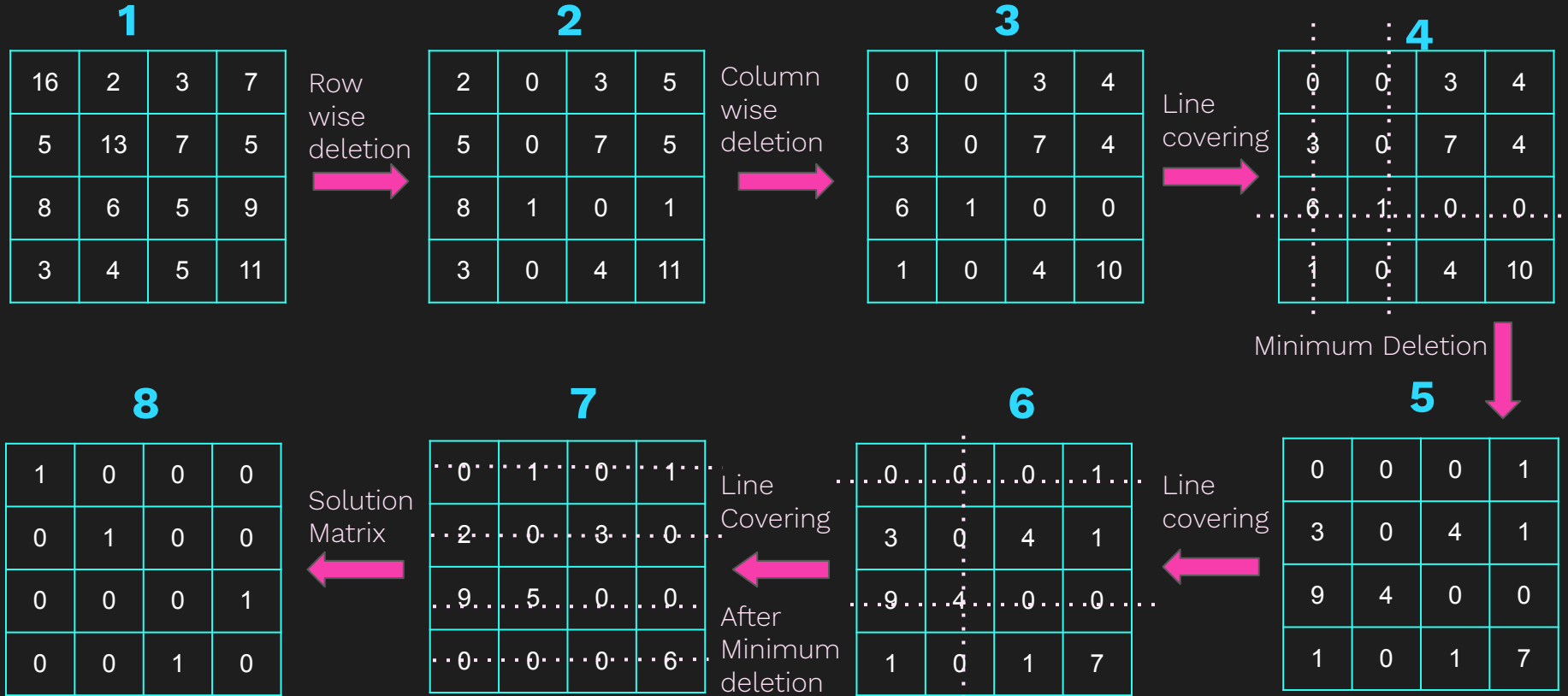
For each column, find the lowest element and subtract it from each element in that column

03

CREATE ADDITIONAL
ZEROS

Find smallest element, k not covered by a line in Step 3, Subtract k from uncovered elements and add k to elements covered twice.

EXAMPLE



MODERN APPROACH

- This is achieved by selecting only the first M lowest cost spots out of N spots, corresponding to each vehicle P
- Now, we sum up all these spots and create an array S where each spot is present only one time and its size is n which at worst case theoretically is M^2 but practically is $M \leq n \ll M^2$ (is a constant multiplication of M in real life)
- We then create a square matrix D of size $n \times n$, where n columns corresponds to S and n rows corresponding to m P vehicles and $n-m$ P virtual vehicles whose distance from all spots is 0.

MODERN APPROACH

- This method is based on classical Hungarian algorithm
- Hungarian Algorithms time complexity is $O(N^3)$ which makes it impossible to solve a real life scenarios especially for parking allotment problem for a city, where N =no of spots can be in Millions
- Our Modern algorithm based on the research paper solves this problem in $O(n^3)$ where $n \ll N$ (of order hundreds in real life). So, Time Complexity is drastically reduced and is therefore practically implementable
- Final solution doesn't give the optimal cost rather is close to the optimal cost.

PSEUDOCODE

Our Execution will start from main function

```
Main(){
  N = Input for number of parking spots
  M = Input for number of vehicles included in our vehicle queue
  For i = 0 to M-1 {
    For j = 0 to N-1 {
      D[i][j] = Corresponding input for distance between ith vehicle and jth parking spot
    }
  }
  Matrix_answer = Cost(D,M,N)
}
```

//This is for take input for distance between vehicle and parking spots

//This Matrix_answer is our final solution matrix

Below function is for calling hungarian and other function

```
Cost(D,M,N){
  res = hungarian(Dnew(D,M,Combine(D,M,N)))
  return res;
}
```

//Dnew and Combine functions declared after this function

PSEUDOCODE

This function is for develop a matrix which will have only those parking spots with vehicles which can take its solution

```
Combine(D,M,N){
```

```
  S = [ ]
```

```
  Si = [ ]
```

```
  For i = 0 to M-1 {
```

```
    si = subi(D,i,M)
```

```
    For j = 0 to M-1 {
```

```
      if S is not empty {
```

```
        if Si[j] is not present in S {
```

```
          S += Si[j]
```

```
        }
```

```
      }
```

```
      else {
```

```
        S += Si[j]
```

```
      }
```

```
    }
```

```
  }
```

```
  return S;
```

```
}
```

//subi function declared after this function

//find if this particular element is present in S or not

PSEUDOCODE

This function is for find the best M parking spots for a particular vehicle

```
Subi(D,i,M){
```

```
  si = [ ] (1d matrix)
```

```
  For j = 0 to M-1 {
```

```
    Q = find index of parking spot which has jth min distance in D[i]
```

```
    Si += Q
```

```
  }
```

```
  return Si;
```

```
}
```

PSEUDOCODE

This function is for make a new matrix which will have a square matrix with virtual vehicle included

```
Dnew(D,M,S){
D1 = [ ][ ] (2d matrix)

For i = 0 to size of S - 1 {
    For j = 0 to M-1 {
        D1[j][i] = D[j][S[i]]
    }
    if(M<size of S){
        For i = 0 to size od S -1 {
            For j = M to size of S - 1 {
                D1[j][i] = 0
            }
        }
    }
}
return D1;
}
```



IMPLEMENTATION



16x3 (matrix input)

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16
P1	5	3	10	11	12	14	20	18	19	100	25	72	65	78	79	55
P2	6	2	9	15	4	4	56	49	47	74	56	78	45	46	48	45
P3	9	8	9	6	6	3	56	47	78	27	26	23	45	45	56	78

Best 3 possible assignment for each vehicle

P1	S2	S1	S3	→	SUB1
P2	S2	S5	S6	→	SUB2
P3	S6	S4	S5	→	SUB3

Now, from the submatrix, SUB1, SUB2 and SUB3, we are going to make a matrix **S** having only unique elements

S	S1	S2	S3	S4	S5	S6
----------	----	----	----	----	----	----

Our new **D'** matrix.

	S1	S2	S3	S4	S5	S6
P1	5	3	10	11	12	14
P2	6	2	9	15	4	4
P3	9	8	9	6	6	3
P4	0	0	0	0	0	0
P5	0	0	0	0	0	0
P6	0	0	0	0	0	0

Virtual
Vehicles

Now we will pass our D' to the Hungarian algorithm to generate X

	S1	S2	S3	S4	S5	S6
P1	1	0	0	0	0	0
P2	0	1	0	0	0	0
P3	0	0	0	0	0	1
P4	0	0	1	0	0	0
P5	0	0	0	1	0	0
P6	0	0	0	0	1	0

Virtual
Vehicles

$$X_{ij} = \begin{cases} 1, & \text{if } P_i \text{ is guided to } S_j \\ 0, & \text{otherwise} \end{cases}$$

$$\text{TOTAL COST} = \sum \sum (D'_{ij} \times X_{ij})$$

TOTAL COST=10 (for above problem)

Classical Hungarian approach

$O(16^3)$

Modern approach

$O(6^3)$



THE CODE



MAIN FUNCTION

```
int main()
{
    int N, M; // M=no of parking cars and N=no of parking spots

    cout<<"Enter no of parking spots: ";
    cin>>N;
    cout<<"Enter no of cars: ";
    cin>>M;
    vector<vector<int>> D (M, vector<int> (N)); // Distance (Dij) between parking car- i and parking spot- j
    cout<<endl;
    for(int i=0;i<M;i++) // Taking distance between each car and parking spot
        for(int j=0;j<N;j++)
            {
                cout<<"Enter distance between car P"<<i+1<<" and Spot"<<j+1<<": ";
                cin>>D[i][j];
            }
    cout<<endl;
    cout<<"Input matrix-\n";
    print(D);
    cout<<".....";
    cout<<"\nTotal Cost: "<<cost(D,M,N);
    cout<<"\n.....";
    return 0;
}
```

COST FUNCTION

```
int cost(vector<vector<int>> D,int M, int N) //return total cost
{
    //print(Dnew(D,M,combine(D,M,N)));
    auto res=hungarian(Dnew(D,M,combine(D,M,N)));
    return res;
}
```

COMBINE FUNCTION

```
vector<int> combine(vector<vector<int>> D, int M, int N) // combine all Si with unique elements
{
    vector<int> S;
    vector<int> Si;
    for(int i=0;i<M;i++)
    {
        Si=subi(D,i,M);

        for(int j=0;j<M;j++)
        {
            if(S.size()!=0)
            {
                auto k=find(S.begin(),S.end(),Si[j]);
                if(k==S.end())
                    S.push_back(Si[j]);
            }
            else
                S.push_back(Si[j]);
        }
    }

    return S;
}
```

SUBI FUNCTION

```
vector<int> subi(vector<vector<int>> D,int i, int M) // create
{
    vector<int> temp=D[i];
    sort(temp.begin(),temp.end());
    vector<int> Si;

    for(int j=0;j<M;j++)
    {
        auto k=find(D[i].begin(),D[i].end(),temp[j]);
        Si.push_back(k-D[i].begin());
    }

    return Si;
}
```

DNEW FUNCTION

```
vector<vector<int>> Dnew(vector<vector<int>> D, int M, vector<int> S)
{
    vector<vector<int>> D1 (S.size(), vector<int> (S.size()));

    for(int i=0;i<S.size();i++)
        for(int j=0;j<M;j++)
            D1[j][i]=D[j][S[i]];

    if(M < S.size()) //since M is less than size of S we need to add v
    {
        for (int i=0;i<S.size();i++)
            for(int j=M;j<S.size() ;j++)
                D1[j][i]=0;
    }

    return D1;
}
```


PRINT FUNCTION

```
void print(vector<vector<int>> D1)
{
    for(int i=0; i< D1.size(); i++)
    {
        cout<<"P"<<i+1<<"->";
        for(int j=0;j<D1[i].size();j++)
        {
            cout<<" | "<<D1[i][j];
        }
        cout<<" \n";
    }
}
```

INPUT

Enter no of parking spots: 16

Enter no of cars: 3

Enter distance between car P1 and Spot1: 5

Enter distance between car P1 and Spot2: 3

Enter distance between car P1 and Spot3: 10

Enter distance between car P1 and Spot4: 11

Enter distance between car P1 and Spot5: 12

Enter distance between car P1 and Spot6: 14

Enter distance between car P1 and Spot7: 20

Enter distance between car P1 and Spot8: 18

Enter distance between car P1 and Spot9: 19

Enter distance between car P1 and Spot10: 100

Enter distance between car P1 and Spot11: 25

Enter distance between car P1 and Spot12: 72

Enter distance between car P1 and Spot13: 65

Enter distance between car P1 and Spot14: 78

Enter distance between car P1 and Spot15: 79

Enter distance between car P1 and Spot16: 55

Enter distance between car P2 and Spot1: 6

Enter distance between car P2 and Spot2: 2

Enter distance between car P2 and Spot3: 9

Enter distance between car P2 and Spot4: 15

Enter distance between car P2 and Spot5: 4

Enter distance between car P2 and Spot6: 4

Enter distance between car P2 and Spot7: 56

Enter distance between car P2 and Spot8: 49

Enter distance between car P2 and Spot9: 47

Enter distance between car P2 and Spot10: 74

Enter distance between car P2 and Spot11: 56

Enter distance between car P2 and Spot12: 78

Enter distance between car P2 and Spot13: 45

Enter distance between car P2 and Spot14: 46

Enter distance between car P2 and Spot15: 48

Enter distance between car P2 and Spot16: 45

Enter distance between car P3 and Spot1: 9

Enter distance between car P3 and Spot2: 8

Enter distance between car P3 and Spot3: 9

Enter distance between car P3 and Spot4: 6

Enter distance between car P3 and Spot5: 6

Enter distance between car P3 and Spot6: 3

Enter distance between car P3 and Spot7: 56

INPUT

```
Enter distance between car P2 and Spot8: 49
Enter distance between car P2 and Spot9: 47
Enter distance between car P2 and Spot10: 74
Enter distance between car P2 and Spot11: 56
Enter distance between car P2 and Spot12: 78
Enter distance between car P2 and Spot13: 45
Enter distance between car P2 and Spot14: 46
Enter distance between car P2 and Spot15: 48
Enter distance between car P2 and Spot16: 45
Enter distance between car P3 and Spot1: 9
Enter distance between car P3 and Spot2: 8
Enter distance between car P3 and Spot3: 9
Enter distance between car P3 and Spot4: 6
Enter distance between car P3 and Spot5: 6
Enter distance between car P3 and Spot6: 3
Enter distance between car P3 and Spot7: 56
Enter distance between car P3 and Spot8: 47
Enter distance between car P3 and Spot9: 78
Enter distance between car P3 and Spot10: 27
Enter distance between car P3 and Spot11: 26
Enter distance between car P3 and Spot12: 23
Enter distance between car P3 and Spot13: 45
Enter distance between car P3 and Spot14: 45
Enter distance between car P3 and Spot15: 56
Enter distance between car P3 and Spot16: 78
```

OUTPUT

Input martix-

P1->	5	3	10	11	12	14	20	18	19	100	25	72	65	78	79	55
P2->	6	2	9	15	4	4	56	49	47	74	56	78	45	46	48	45
P3->	9	8	9	6	6	3	56	47	78	27	26	23	45	45	56	78

.....

Best spots Assignment using this algorithm

For car P2 assignment on spot with cost= 2

For car P1 assignment on spot with cost= 5

For car P3 assignment on spot with cost= 3

Total Cost: 10

.....

PS C:\Users\Adxto\Desktop\Project Spartan\Smart Parking System> █