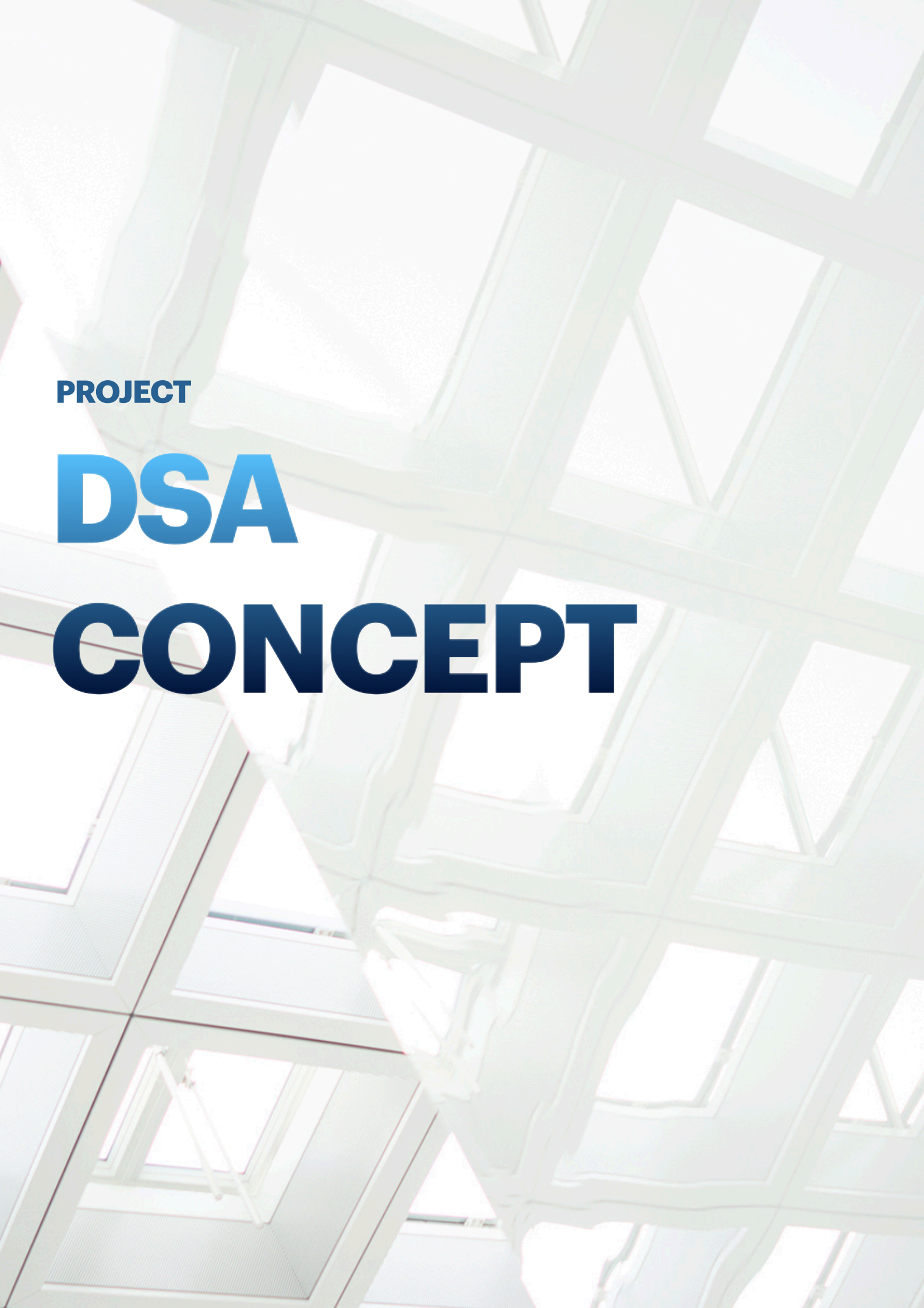

PROJECT

DSA CONCEPT

What is map navigator(Dijkstra algorithm)?

Dijkstra's Algorithm is a **greedy algorithm** used to find the **shortest path** between a source node and all other nodes in a weighted graph. It's widely used in map navigation systems to calculate the most efficient route.

Algorithm Overview …

Graph Representation

- Use an **adjacency list** to represent the graph.
- Each node points to a list of pairs: (neighbor, weight)

Initialize Distance Array

- Set the source node's distance to `0`, others to ∞.

Use a Priority Queue (Min-Heap)

- Always process the node with the smallest current distance.

Relaxation Step

- For each neighbor of the current node:
  - If `current_distance + edge_weight < known_distance`, update it.

Repeat until all nodes are visited.

What does it teaches?

1. **Graph Data Structures**
2. **Greedy Algorithm Design**
3. **Priority Queues / Min-Heaps**
4. **Shortest Path Algorithms**
5. **Time and Space Complexity Awareness**
6. **Real-World Applications**
7. **Problem Solving and Abstraction**

CODE…

```
#include<bits/stdc++.h>
using namespace std;
#define INT_MAX 100000
```

```
//input be like n number of nodes
//enter the adj matrix

void relax(priority_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > >&
p_q,int node,int distance)
{
   vector<pair<int,int>> vec;
   while(p_q.top().second != node){
      pair<int,int> ans=p_q.top();
      p_q.pop();
      vec.push_back(ans);
   }
   pair<int,int>ans=p_q.top();
   p_q.pop();
   if(distance<ans.first){
      ans=make_pair(distance,node);
   p_q.push(ans);
   }
   else
   p_q.push(ans);
   for(int i=0;i<vec.size();i++)
   p_q.push(vec[i]);
   return ;
}

void Dijkstra(vector<vector<int> >& adj,vector<int>& answer,int node)
{
   priority_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > > p_q;
   pair<int,int> a=make_pair(0,0);
   p_q.push(a);
   for(int i=1;i<adj.size();i++){
      pair<int,int> ans=make_pair(INT_MAX,i);
      p_q.push(ans);
   }

   vector<bool> visited(adj.size(),0);
   while(!p_q.empty())
   {
      int nd=p_q.top().second;
      int distance=p_q.top().first;
      p_q.pop();
      if(visited[nd]==true)continue;
      visited[nd]=true;
      answer[nd]=distance;
      for(int i=0;i<adj[nd].size();i++)
      {
      if(adj[nd][i]!=0 && !visited[i]){
         relax(p_q,i,distance+adj[nd][i]);
      }
      }
   }
   return;
```

```
}


int main(){

    int n;

    cin>>n;
    vector<vector<int> > adj(n,vector<int>(n,0));
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cin>>adj[i][j];
        }
    }
    vector<int> answer(adj.size(),0);
    Dijkstra(adj,answer,0);
    for(int i=0;i<adj.size();i++)
    cout<<answer[i]<<" “;
 return 0;
}
```

Time complexity->O((E+V)log v)
Space complexity->O(E*V)

What is file zipper(Huffman coding)?

Huffman Coding is a **lossless compression algorithm** that reduces file size by assigning shorter binary codes to more frequent characters and longer codes to less frequent ones. It builds a binary tree (Huffman Tree) based on character frequencies.

Algorithm Overview …

Frequency Analysis

- Count the frequency of each character in the input file.

Build a Min-Heap

- Insert each character as a node with its frequency into a priority queue (min-heap).

Construct Huffman Tree

- While there are more than one nodes:
  - Remove two nodes with the smallest frequency.
  - Combine them into a new internal node (sum of frequencies).
  - Re-insert the combined node into the heap.

Generate Huffman Codes

- Traverse the Huffman Tree:
  - Left edge = `0`, right edge = `1`.
  - Leaf nodes receive binary codes.

Encode the Input File

- Replace each character with its Huffman binary code.

Save Encoded File & Tree

- Save the binary file (compressed) and the tree (for decompression).

What Huffman teaches …

Greedy Algorithm application (optimal substructure).

Priority Queue / Min Heap usage for efficient tree construction.

Tree Data Structure for encoding & decoding.

Map and String Manipulation for frequency and encoding.

File Handling for reading/writing compressed output.

CODE…

```
#include<bits/stdc++.h>
using namespace std;
//input:->
//first line give the string to compress str
//then give a string of 0's and 1's to decode it s

//structure of node
struct node{
    //character value
    char data;
    //frequency of that character
    int fre;
    //it's left and right child
    node* right,* left;
    //constructor
    node(char data,int fre){
        this->data=data;
        this->fre=fre;
        this->right=NULL;
        this->left=NULL;
    }
};

//functor for making priority queue minimum
struct compare{
    bool operator()(node* a,node* b){
        return a->fre>b->fre;
    }
};

//printing the prefixfree code of all the characters
void print_code(node* root,string str){
    //base case
    //if a character print the code
    if(root->data != '$'){
        cout<<root->data<<"-->"<<str<<endl;
        return;
    }
    //on left str+0
    print_code(root->left,str+"0");
    //on right str+1
    print_code(root->right,str+"1");
    return;
}

//decode string of 0's and 1's into actual stirng of character
void decode_string(node* root,string str){
    node*temp=root;
    for(int i=0;i<str.length();i++){
        //if it is 1 goto right
        if(str[i]=='1'){
            temp=temp->right;
            //if character is not $ print character
            if(temp->data != '$'){
                cout<<temp->data;
                temp=root;
            }
        }
```

```
        //else go to the left for 0
        else{
            temp=temp->left;
            //print data if not equal to $
            if(temp->data != '$'){
                cout<<temp->data;
                temp=root;
            }
        }
    }
    return;
}

//huffman code to make the tree
void HuffmanCoding(vector<pair<char,int> >&freq,string str){
    //making priority queue
    priority_queue<node*,vector<node*>,compare> p_q;
    //inserting all the values
    for(int i=0;i<freq.size();i++){
        p_q.push(new node(freq[i].first,freq[i].second));
    }
    //if size is one the only node is the root node
    while(p_q.size()!=1){
        //pop the top 2 least frequency character
        node* n1=p_q.top();
        p_q.pop();
        node* n2=p_q.top();
        p_q.pop();
        //make a newnode with this two nodes as child of it and freq=freq1+freq2
        node* newnode=new node('$',n1->fre+n2->fre);
        newnode->left=n1;
        newnode->right=n2;
        //push the newnode with new freq into priority queue
        p_q.push(newnode);
    }
    //the only node is root node
    node* root=p_q.top();
    //print code for each character
    print_code(root,"");
    //decode the given string in actual characters
    decode_string(root,str);
}

//decode the string into characters and there frequency
vector<pair<char, int> > decode(string str,unordered_map<char,int>& freq){
    int length=str.length();
    vector<pair<char ,int> >ans;
    //increasing the freq of each character present in it using map
    for(int i=0;i<length;i++){
       freq[str[i]]++;
    }
    //store this value in an array
    unordered_map<char,int>::iterator it;
    for(it=freq.begin();it!=freq.end();it++){
        ans.push_back(make_pair(it->first,it->second));
    }
    //return array
    return ans;
}
```

```
int main(){
    string str;
    cin>>str;
    string to_decode;
    cin>>to_decode;
    //map
    unordered_map<char,int>freq;
    //array
    vector<pair<char,int> >frequency;
    frequency=decode(str,freq);
    //calling huffman codeing function
    HuffmanCoding(frequency,to_decode);
    return 0;
}

//n size of input and k is number of unique character
Time complexity->O(n+klogk)
//for the compressed string n is size of input
Space compexity->O(n)
```

What is Sudoku Solver?

Given a **9x9 grid** partially filled with numbers, the solver finds a way to:

- Fill all **empty cells**
- So that **each row, each column, and each 3×3 subgrid** contains the digits `1 to 9` **without repetition**

How does it works..

Most Sudoku solvers use **backtracking**, a type of recursion that:

1. Tries a number in an empty cell
2. Moves to the next empty cell
3. If a conflict arises, it **backtracks** and tries a different number

---

Approach…

A Sudoku solver uses backtracking to fill a 9×9 grid such that every row, column, and 3×3 subgrid contains digits from 1 to 9 without repetition. The algorithm scans the grid to find the first empty cell and tries placing numbers 1 through 9 in it. For each number, it checks whether the placement is valid by ensuring the number doesn't already exist in the same row, column, or subgrid. If valid, the number is temporarily placed, and the algorithm recursively attempts to solve the rest of the board. If a dead end is reached, it backtracks by removing the last placed number and tries the next one. This process continues until the entire grid is filled correctly.

What does it teaches you..

- Backtracking – Learn how to try solutions step-by-step and undo choices when needed.
- Recursion – Practice calling a function within itself to explore different possibilities.
- Constraint Checking – Understand how to validate rules (rows, columns, boxes) before making a move.
- Grid Traversal – Improve working with 2D arrays and coordinate systems.
- Problem Solving – Boost logical thinking by breaking down a complex puzzle into solvable parts.
- Optimization Thinking – Think about reducing unnecessary checks and speeding up the algorithm.
- Debugging Recursive Logic – Sharpen your skills in tracing backtracking errors and call stacks.

CODE

```
#include <bits/stdc++.h>
using namespace std;
int N=9;
//input will be :-
//first line row and column
//next n line are rows

bool is_valid(vector<vector<int>>& board,int row,int column,int num)
{
    //cheching in the row and column
    //if same number is present then invalid
    for (int x = 0; x < N; x++) {
        if (board[row][x] == num || board[x][col] == num)
            return false;
    }

    int s_row = row - row % 3;
    int s_col = col - col % 3;

    //cheking in 3 cross 3 block for same number
    //if present return false
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (board[s_row + i][s_col + j] == num)
                return false;

    //if any of the case is not present return true
    //possible
    return true;
}

//sudoku function
bool sudoku_func(int board[N][N]) {
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            //looking for empty cell
            if (board[row][col] == 0) {
                //if found empty cell check for all number 1 to 9
                for (int num = 1; num <= 9; num++) {
                    if (is_valid(board, row, col, num)) {
                        board[row][col] = num;

                        //recursivly checking for the board
                        if (sudoku_func(board))
                            return true;

                        //backtracking
                        board[row][col] = 0;
                    }
                }
                //if no valid number found
                return false;
            }
        }
    }
    //all cells are filled return true
    return true;
}
```

```
void print_board(vector<vector<int>>& board,int row,int col)
{
    for( int i=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            cout<<board[i][j]<<" ";
        }
        cout<<endl;
    }
    return;
}

int main() {
            //input the number of rows and column
            int row,column;
            cin>>row>>column;

            vector<vector<int>> board(row,vector<int>(column));

            //input the sudoku board
            for(int i=0;i<row;i++)
            {
                for(int j=0;j<column;j++)
                {
                    cin>>board[i][j];
                }
            }

            //solving the sudoku board
            //if possible printing it
    if(sudoku_func(board)){
        cout<<"possible";
        print_board(board,row,column);
    }
    else{
        cout<<"Not possible";
    }
    return 0;
}

//m is number of empty blocks
Time complexity->O(9^m)
Space complexity->O(m)
```

What is cash flow minimizer?

It's a problem where a group of people have borrowed money from each other, and we want to minimize the **number of transactions** required to settle all debts.

Approach…

1. **Calculate net amount** for each person (money to receive - money to pay).
2. **Find the person with the maximum credit** and one with **maximum debit**.
3. **Settle minimum of their absolute values**.
4. Repeat recursively until all balances are zero.

What it teaches you..

Understanding **graph-based debt relationships**

Applying **greedy strategy** to minimize transactions

Using **net balances** to simplify multi-party debt

Practicing **recursion and greedy pairing**

Helps understand **real-world optimization** scenarios

CODE…

```
#include <bits/stdc++.h>
using namespace std;

//input be like first line n number of people
//next n rows are cash n rows

int min_val(vector<int>& amount){
    int ans=0;
    for( int i=1;i<amount.size();i++){
        if(amount[i]<amount[ans])
        ans=i;
    }
    return ans;
}

int max_val(vector<int>& amount){
    int ans=0;
    for( int i=1;i<amount.size();i++){
        if(amount[i]>amount[ans])
        ans=i;
    }
    return ans;
}

void minimize_cash_flow(vector<int>& amount){
    //finding the min and max value index
    int minval=min_val(amount);
```

```
    int maxval=max_val(amount);

    //base condition
    if(amount[maxval]==0 && amount[minval]==0)return ;

    //decide what amount one can maximaly give or take
    int val_to_reduce=min((-1*amount[minval]),amount[maxval]);
    amount[maxval]-=val_to_reduce;
    amount[minval]+=val_to_reduce;

    cout<<"person "<<minval<<" pays "<<val_to_reduce<<" to "<<maxval<<endl;

    //recursive call
    minimize_cash_flow(amount);
    return;

}

int main() {
        int n;
        cin>>n;

        vector<vector<int>> cash(n,vector<int>(n));

        //who has to pay whom and how much
        //i has to pay to y
        for (int i=0;i<n;i++)
        {
            for (int j=0;j<n;j++){
        cin>>cash[i][j];
            }
        }

        //what is the absolute cash credit-debit of a person
        vector<int> amount(n);
    for(int i=0;i<n;i++){
        for (int j=0;j<n;j++)
        {
            amount[i]+=(cash[j][i]-cash[i][j]);
        }
    }

    minimize_cash_flow(amount);
    return 0;
}
```

Example input
3
0 2000 3000
0 0 5000
0 0 0

Output is
Person 0 pays 5000 to 2
Person 1 pays 3000 to 2

Time complexity->O(n^2)
Space complexity->O(n^2)

What is Snack array?

A **snake array** (or snake pattern matrix) refers to filling a 2D matrix such that the elements are arranged in a zig-zag (snake-like) pattern. It often alternates the direction of filling row by row.

1 2 3 4 5
10 9 8 7 6
11 12 13 14 15

Notice how:

- Row 0 is filled **left to right**
- Row 1 is filled **right to left**
- Row 2 is filled **left to right**
- ...and so on (alternating)

Explanation...

Initialise a 2D array of size `rows x cols`

Use a counter starting at 1

Loop through each row:

- If the row is **even**, fill **left to right**
- If the row is **odd**, fill **right to left**

What Snack Array problem teaches?

1.Control flow and conditional logic.

2.Looping technique.

3.Pattern printing logic.

4.Algorithmic thinking.

5.Time and space complexity.

CODE

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

using namespace std;


//input is like:-

//only 1 line

//first line input is n and m dimension of graphs


//function to get the snack order

vector<vector<int>> func(int row,int column)

{

    vector<vector<int>> ans(row,vector<int>(column,0));


    //flag for direction

    bool flag=true;

    int count=1;


    //outer loop for rows

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

    {

        //inner loop for colums

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

        {

            //true then left to right

            if(flag){

                ans[i][j]=count;

                count++;

            }
```

```
            else{
                ans[i][column-j-1]=count;
                count++;
            }
        }
        flag=(!flag);
    }
    return ans;
}

int main() {
    //input symbols
    int row,column;
    cin>>row>>column;

    //calling function for snack array
    vector<vector<int>> Snack_array=func(row,column);

    //printing array
    for( int i=0;i<row;i++){
        for( int j=0;j<column;j++){
            cout<<Snack_array[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Time complexity->O(n*m)

Space complexity->O(n*m)