K.RAMAKRISHNAN

COLLEGE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

SAMAYAPURAM, TRICHY-621 112

Practical Record Note

Name	:	PRAGATHEESHWARAN E
Register Number	:	2303811710421118
Subject code/name	:	CGB1211 - Design and Analysis of Algorithms Laboratory
Programme	:	B.E - Computer Science and Engineering

K.RAMAKRISHNAN

COLLEGE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

SAMAYAPURAM, TRICHY-621 112

CERTIFICATE

Certified that this is a bonafide record of work done by **PRAGATHEESHWARAN E** of **III** Semester in **CGB1211 - Design and Analysis of Algorithms Laboratory** during the academic year 2024-2025

His/Her University Register Number is **2303811710421118**

Staff Incharge **Head of the Department**

Submitted for the Practical exam held on: 05.12.2024

Internal Examiner
Date: 05.12.2024

External Examiner
Date: 05.12.2024

INDEX

S.No.	Date	PROGRAM NAME
1	07-Aug-2024	Factorial using recursion
2	07-Aug-2024	Factorial using non recursive approach
3	14-Aug-2024	String matching algorithm
4	14-Aug-2024	Sort E-Books based on IDs using Quick Sort Algorithm
5	28-Aug-2024	Knapsack
6	28-Oct-2024	Dijkstra's Shortest Path Algorithm
7	28-Oct-2024	Huffman Encoding
8	25-Nov-2024	N Queen problem using backtracking
9	28-Oct-2024	Job assignment
10	28-Oct-2024	Travelling Salesman Problem
11	06-Nov-2024	Graph Colouring

Exp No:	ExpName: *Factorial using recursion*	Date:

Aim:

Write a recursive function factorial that accepts an integer n as a parameter and returns the factorial of n, or $n!$

A factorial of an integer is defined as the product of all integers from 1 through that integer inclusive. For example, the call of factorial(4) should return 1 * 2 * 3 * 4, or 24. The factorial of 0 and 1 are defined to be 1.

You may assume that the value passed is non-negative and that its factorial can fit in the range of type int.

Input format:

The first line is the integer that represents the number of test cases.
Each test case will contain a single integer n where $n >= 0$.

Output format:

For each input case, generate and print the factorial of the integer.

Program:

factorialcalc.c

```
#include <stdio.h>
int factorial(int n)
{
        if(n<0)
                return -1;
        if (n == 0 || n ==1)
           return 1;
        else
           return n*factorial(n-1);
}
int main()
{
   int T, no;
   scanf("%d",&T);
   while(T--)
   {
     scanf("%d",&no);
     printf("%d\n",factorial(no));
   }
 return 0;
}
```

Output:

Test case - 1
User Output
6
4
24
3
6
11

39916800
7
5040
1
1
0
1

Test case - 2
User Output
4
5
120
8
40320
9
362880
3
6

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: *Factorial using non recursive approach*	Date:

Aim:

Write a C program to calculate the factorial of small positive integers using non recursive approach

Input Format:

Single line of input contains an integer N representing the value to find the factorial

Output Format:

Print the factorial result of N

Program:

factorial.c

```
#include<stdio.h>
int main(){
        int n,i;
        long long factorial=1;
        scanf("%d",&n);
                for(i=1;i<=n;++i){
                        factorial *= i;
                }
                printf("%lld",factorial);

        return 0;
        }
```

Output:

Test case - 1
User Output
2
2

Test case - 2
User Output
3
6

Test case - 3
User Output
4
24

Test case - 4

User Output
5
120

Test case - 5
User Output
1
1

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: *String matching algorithm*	Date:

Aim:

Write a program to implement a Naive string-matching algorithm

The program takes two strings as input: a text string and a pattern string. The program should search for occurrences of the pattern string within the text string and print the starting index of each occurrence. If the pattern is not found in the text, the program should print **"Not Found"**.

Program:

stringMatching.c

```
#include <stdio.h>
#include <string.h>


void search( const char *pat, const char *txt)
{
        int n = strlen(txt);   // Length of the text
    int m = strlen(pat);   // Length of the pattern
    int found = 0;         // Flag to check if the pattern is found

    // Loop through the text
    for (int i = 0; i <= n - m; i++) {
        // Check for a match
        int j;
        for (j = 0; j < m; j++) {
            if (txt[i + j] != pat[j]) {
                break; // Mismatch found
            }
        }
        // If the pattern is found
        if (j == m) {
            printf("Pattern found at index %d\n", i);
            found = 1; // Set found flag
        }
    }

    // If no match was found, print "Not Found"
    if (!found) {
        printf("Not Found\n");
        }
}

int main()
{       char txt[50], pat[50];
        scanf("%s", txt);
        scanf("%s", pat);
        search(pat, txt);
        return 0;
}
```

Output:

Test case - 1
User Output
AABAACAADAABAAABAA
AABA
`Pattern found at index 0`
`Pattern found at index 9`
`Pattern found at index 13`

Test case - 2
User Output
IamSoLuckyToLearnProgramming
lucky
`Not Found`

Test case - 3
User Output
hjyhfdrtiufgghfvrdvgfbhgcgvvjbhgvbygfvbhg
fvbhg
`Pattern found at index 36`

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Sort E-Books based on IDs using Quick Sort Algorithm***	Date:

Aim:

You are developing a program for a digital library to manage its collection of e-books. Each e-book is identified by a unique Book ID. To optimize the process of listing e-books, you need to implement a sorting algorithm that arranges the e-books based on their Book ID's.

Your task is to design a C program to sort the e-books based on their Book ID's using the quick sort algorithm.

Constraints:

- The digital library manages a maximum of 1000 different e-books.

Input Format:

- The first line contains an integer, representing the total number of e-books.
- The next line contains space-separated integers, representing the Book ID's of the respective e-books.

Output Format:

- The first line should display the original array of Book ID's.
- The second line should display the sorted array of Book ID's.

Program:

`BookCollection.c`

```c
#include <stdio.h>
#include <stdlib.h>

// Quicksort partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
    int i = (low - 1);     // Index of the smaller element

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i + 1] and arr[high] (or pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Quicksort function
void quickSort(int arr[], int low, int high) {
 if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}

int main() {
    int n;
    printf("no of e-books: ");
    scanf("%d", &n);

    int BID[n];
    printf("Book ID's of e-books: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &BID[i]);
    }

    printf("Original Book ID's: ");
    printArray(BID, n);
```

```
    printf("\n");

    // Sort the e-books based on their BIDs using quicksort
    quickSort(BID, 0, n - 1);

    printf("Sorted Book ID's: ");
    printArray(BID, n);
    printf("\n");

    return 0;
}
```

Output:

Test case - 1
User Output
no of e-books:
5
Book ID's of e-books:
15 69 58 47 65
Original Book ID's: 15 69 58 47 65
Sorted Book ID's: 15 47 58 65 69

Test case - 2
User Output
no of e-books:
6
Book ID's of e-books:
98 48 57 56 32 15
Original Book ID's: 98 48 57 56 32 15
Sorted Book ID's: 15 32 48 56 57 98

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Knapsack***	Date:

Aim:

You are given weights and values of**N**items, put these items in a knapsack of capacity**W**to get the maximum total value in the knapsack. Note that we have only**one quantity of each item**.

In other words, given two integer arrays**val[0..N-1]**and**wt[0..N-1]**which represent values and weights associated with**N**items respectively. Also given an integer **W** which represents knapsack capacity, find out the maximum value subset of**val[]**such that the sum of the weights of this subset is smaller than or equal to**W.**

You cannot break an item,**either pick the complete item or don't pick it (0-1 property)**.

Constraints:

1 ≤ **N** ≤ 1000
1 ≤ **W** ≤ 1000
1 ≤ **wt[i]** ≤ 1000
1 ≤ **v[i]** ≤ 1000

Input format:

- The first input line reads a positive integer representing **N**.
- The second input line reads a positive integer representing **W**.
- The third input line reads **N** space-separated positive integers representing values of **N** items.
- The fourth input line reads **N** space-separated positive integers representing weights of **N** items.

Output format:

- The output is an integer representing the maximum total value in knapsack.

Sample test case:

Input:

3 //N
4 // W - Capacity of the Knapsack
1 2 6 //Values of N items
4 5 1 //Weights of N items

Output:

6

Explanation:

In this example, we have 3 items with values [1, 2, 6] and weights [4, 5, 1]. The knapsack capacity is 4. The optimal solution is to choose the third item, which has a weight of 1 and a value of 6. This is the maximum value we can achieve while keeping the total weight less than or equal to the knapsack capacity of 4.

Instruction: To run your custom test cases strictly map your input and output layout with the visible test cases.

Program:

CTC13069.c

```c
#include <stdio.h>

// Function to solve the 0/1 Knapsack Problem
int knapsack(int N, int W, int val[], int wt[]) {
    // Initialize a 2D array to store the maximum value
    int dp[N + 1][W + 1];

    // Initialize the first row and first column to 0
    for (int i = 0; i <= N; i++) {
        dp[i][0] = 0;
    }
    for (int w = 0; w <= W; w++) {
        dp[0][w] = 0;
    }

    // Fill the 2D array
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (wt[i - 1] > w) {
                // If the weight of the current item is greater than the current
capacity, skip this item
                dp[i][w] = dp[i - 1][w];
            } else {
                // Choose the maximum value between including the current item and
excluding the current item
                dp[i][w] = (val[i - 1] + dp[i - 1][w - wt[i - 1]] > dp[i - 1][w]) ?
val[i - 1] + dp[i - 1][w - wt[i - 1]] : dp[i - 1][w];
            }
        }
    }

    // Return the maximum value that can be obtained with the given capacity
    return dp[N][W];
}

int main() {
    int N, W;

    scanf("%d", &N);


    scanf("%d", &W);

    int val[N], wt[N];


    for (int i = 0; i < N; i++) {
        scanf("%d", &val[i]);
    }


    for (int i = 0; i < N; i++) {
        scanf("%d", &wt[i]);
    }

    // Calculate and print the maximum total value
    int max_value = knapsack(N, W, val, wt);
```

```
    printf("%d\n", max_value);

    return 0;
}
```

Output:

Test case - 1
User Output
3
3
1 2 3
4 5 6
0

Test case - 2
User Output
3
4
1 2 6
4 5 1
6

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Dijkstra's Shortest Path Algorithm***	Date:

Aim:

You're a transportation engineer at a city planning department tasked with optimizing public transportation routes for a growing urban population. Your team is exploring graph-based algorithms to find the shortest routes between key locations in the city. Your manager, Alex, provides the context:

As our city's population continues to grow, it's crucial to optimize our public transportation routes to ensure efficient and reliable travel for residents. One approach is to model our transportation network as a graph, where each vertex represents a key location in the city, and edges represent the transportation connections between locations. We need to find the shortest routes between a source vertex, such as a major transit hub, and each vertex in the graph to improve commuter accessibility and reduce travel times.

Input Layout:

- The first line contains an integer V represents the number of vertices in the graph.
- Next line onwards is the graph's adjacency matrix of size V x V.

Constraints to be followed:

- 1<= V <= 1000

Sample test case:

Input:

9
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0

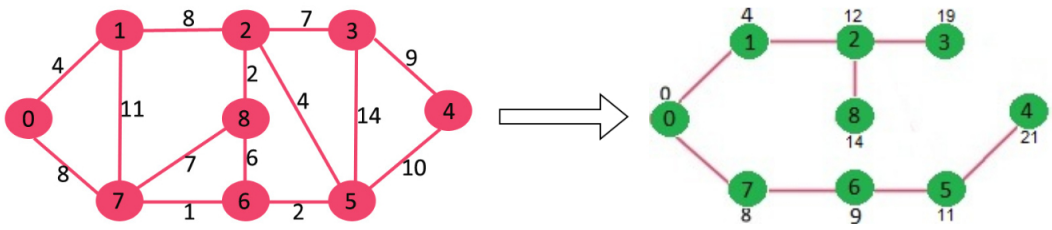
Output:

VertexDistance from Source

00
14
212
319
421
511
69
78
814

Explanation:

Consider the graph (fig -1)with src=0 (Distance of source vertex from itself is always 0) and we get the following Shortest Path Tree-SPT (refer fig-2).



- The distance from 0 to 1 = 4.
- The minimum distance from 0 to 2 = 12. 0->1->2
- The minimum distance from 0 to 3 = 19. 0->1->2->3
- The minimum distance from 0 to 4 = 21. 0->7->6->5->4
- The minimum distance from 0 to 5 = 11. 0->7->6->5
- The minimum distance from 0 to 6 = 9. 0->7->6
- The minimum distance from 0 to 7 = 8. 0->7
- The minimum distance from 0 to 8 = 14. 0->1->2->8

Program:

`DijkstrasShortestPath.c`

```
//Write your code here
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define INF INT_MAX

// Function to find the vertex with the minimum distance
int minDistance(int dist[], int sptSet[], int V) {
    int min = INF, min_index;
    for (int v = 0; v < V; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
// Dijkstra's algorithm to find the shortest paths from a source vertex
void dijkstra(int** graph, int src, int dist[], int V) {
    int sptSet[V]; // Shortest Path Tree Set

    // Initialize all distances as INFINITE and sptSet[] as false
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = 0;
    }
    dist[src] = 0; // Distance from source to itself is always 0

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed
        int u = minDistance(dist, sptSet, V);

        // Mark the picked vertex as processed
        sptSet[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // Update dist[v] if and only if it is not in sptSet, there is an edge from
u to v, and the total weight of path from source to v through u is smaller than the
current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] <
dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
}
int main() {
    int V;
    scanf("%d", &V);

    int** graph = (int**)malloc(V * sizeof(int*));
    for (int i = 0; i < V; i++) {
        graph[i] = (int*)malloc(V * sizeof(int));
    }
```

```
    // Reading the adjacency matrix
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    int* dist = (int*)malloc(V * sizeof(int));
    dijkstra(graph, 0, dist, V); // Find shortest paths from source vertex 0

    // Printing the result
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t\t\t\t %d\n", i, dist[i] == INF ? -1 : dist[i]);
    }

    // Free dynamically allocated memory
    for (int i = 0; i < V; i++) {
        free(graph[i]);
    }
    free(graph);
    free(dist);

    return 0;
}
```

Output:

Test case - 1	
User Output	
9	
0 4 0 0 0 0 0 8 0	
4 0 8 0 0 0 0 11 0	
0 8 0 7 0 4 0 0 2	
0 0 7 0 9 14 0 0 0	
0 0 0 9 0 10 0 0 0	
0 0 4 14 10 0 2 0 0	
0 0 0 0 0 2 0 1 6	
8 11 0 0 0 0 1 0 7	
0 0 2 0 0 0 6 7 0	
Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9

7	8
8	14

Test case - 2	
User Output	
3	
2 1 1	
2 3 1	
3 4 1	
Vertex	Distance from Source
0	0
1	1
2	1

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Huffman Encoding***	Date:

Aim:

Given a string **S** of distinct character of size **N** and their corresponding frequency **f[]** i.e. character **S[i]** has **f[i]** frequency. Your task is to build the Huffman tree and print all the Huffman codes in the preorder traversal of the tree.

Note: While merging if two nodes have the same value, then the node that occurs at first will be taken on the left of Binary Tree and the other one to the right, otherwise Node with less value will be taken to the left of the subtree and other one to the right.

Example 1:

Input:

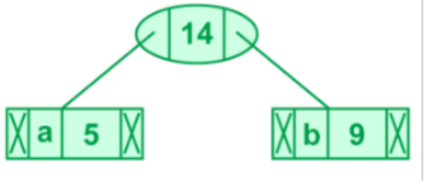
S = "abcdef"

f[ ] = {5, 9, 12, 13, 16, 45}

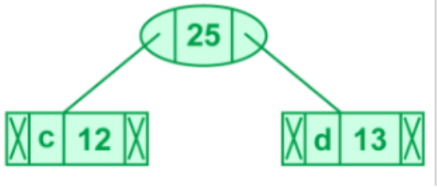
Output:

0 100 101 1100 1101 111

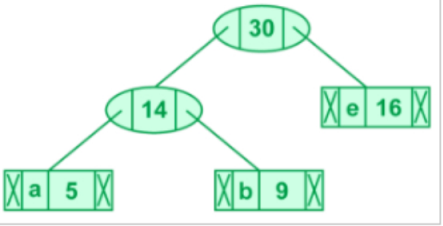
Explanation:



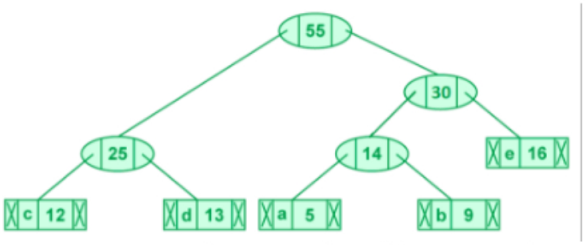
Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



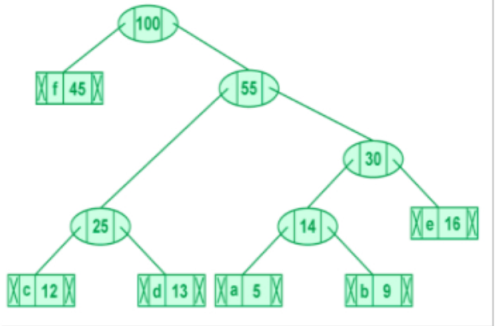
Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



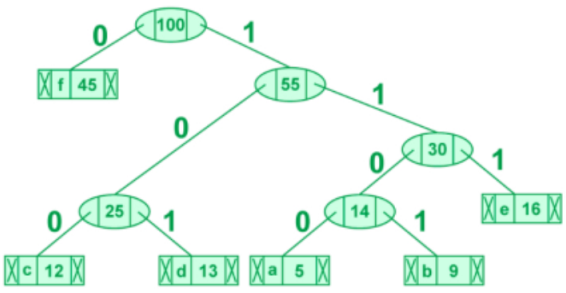
Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array.

Huffman codes will be:

f : 0

c : 100

d : 101

a : 1100

b : 1101

e : 111

Hence printing them in the PreOrder of BinaryTree.

Input Format:

- Enter the characters as a single string **S**.
- Enter the frequencies of the characters as a space-separated list on the next line.

Output Format:

- The program prints the Huffman codes (separated by space) for each character in **S**, in the order they appear.

Constraints:

$1 \leq N \leq 26$

Program:

CTC31048.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A Huffman tree node
struct MinHeapNode {
    char data;  // The character
    int freq;   // The frequency of the character
    struct MinHeapNode *left, *right;  // Left and right child
};

// A MinHeap: Collection of Huffman tree nodes
struct MinHeap {
    int size;    // Current size of the heap
    int capacity; // Capacity of min heap
    struct MinHeapNode **array; // Array of minheap node pointers
};

// A utility function to create a new min heap node
struct MinHeapNode* newNode(char data, int freq) {
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(int capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->size = 0; // Current size is 0
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));

    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// Standard minHeapify function
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]-
>freq)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]-
>freq)
```

```
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
// A utility function to check if size of heap is one
int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

// Extract minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}
// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
  minHeap->array[i] = minHeapNode;
}

// A utility function to build a min heap
void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1;
    for (int i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right);
}

// Creates a min heap of capacity equal to size and inserts all character of data[]
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that builds Huffman tree
```

```
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;

    // Create a min heap & inserts all characters of data[]
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {
        // Extract the two minimum freq items from heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Create a new internal node with frequency equal to the sum of the two nodes
        // and add it back to the min heap
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    // The remaining node is the root node
    return extractMin(minHeap);
}
// Print Huffman codes using the Huffman tree built
void printCodes(struct MinHeapNode* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    if (isLeaf(root)) {
        //printf("%c: ", root->data);
        for (int i = 0; i < top; ++i)
            printf("%d", arr[i]);
        printf(" ");
        }
}

// The main function that builds a Huffman Tree and prints codes
void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);

    int arr[100], top = 0;
    printCodes(root, arr, top);
}

int main() {
    // Input: characters and their frequencies
    char S[100];
    int freq[100];
    int N;
```

```
    // Reading the input
    //printf("Enter the characters: ");
    scanf("%s", S);
    N = strlen(S);

    //printf("Enter the frequencies: ");
    for (int i = 0; i < N; ++i) {
        scanf("%d", &freq[i]);
    }

    // Generate and print Huffman Codes
    HuffmanCodes(S, freq, N);

    return 0;
}
```

Output:

Test case - 1
User Output
abcdef
5 9 12 13 16 45
0 100 101 1100 1101 111

Test case - 2
User Output
abcd
10 20 30 40
0 10 110 111

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: *N Queen problem using backtracking*	Date:

Aim:

N-Queens using Backtracking:

You're a software developer at a chess club you have been tasked to design and implement an algorithm to solve the N-Queens problem using backtracking, providing a step-by-step explanation of how the algorithm works and how it ensures that no queens attack each other on the chessboard.

Here is an algorithm for solving the N-Queens problem using backtracking:

1. Start with an empty chessboard of size N x N.
2. Place the first queen in the leftmost column of the first row.
3. Move to the next row and try to place a queen in an empty square in that row that is not attacked by any of the queens already placed on the board.
4. If a queen can be placed in the current row, move to the next row and repeat step 3.
5. If a queen cannot be placed in the current row, backtrack to the previous row and try a different square in that row. If all squares in the previous row have been tried and none of them worked, backtrack again to the previous row.
6. Repeat steps 3-5 until all N queens have been placed on the board or it is determined that no solution exists.

The key to this algorithm is the "not attacked" rule. To determine whether a queen is being attacked by any other queen on the board, we need to check whether any other queen is in the same row, column, or diagonal as the current queen.

Write the code to implement the N Queen problem using backtracking

Given an Array, you need to place a queen in the array such that no queen can strike down any other queen. A queen has the possibility to attack horizontally, vertically, or diagonally. You need to check whether the queen is placed in the correct position or not.

Input Format:

It contains the number of queens in an Array.

Output Format:

An array in which the queens are placed and print a*****symbol when there is no queen in that particular row or column. You need to print all the possible solutions based on the input and display the Total no of solutions. If there are no solutions then print the solution as 0.

Constraints:

1 <= N <= 6

Program:

nQueen.c

```
/*
int main() {
    int N;
    scanf("%d", &N);

    int board[N];
    for (int i = 0; i < N; i++) {
        board[i] = -1; // Initialize the board
    }

    int totalSolutions = solveNQueens(board, N, 0);
    printf("Total solutions:%d\n", totalSolutions);

    return 0;
}
*/

#include <stdio.h>
#include <stdbool.h>
#define MAX 6
int board[MAX][MAX];
int solutions = 0;
void printSolution(int N) {
        printf("Solution #%d:\n",++solutions);
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (board[i][j] == 1)
                                printf("Q\t");
                        else
                                printf("*\t");
                }
                printf("\n");
        }
}
bool isSafe(int row, int col, int N) {
        int i, j;
        for (i = 0; i < row; i++)
                if (board[i][col])
                        return false;
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
                if (board[i][j])
                        return false;
        for (i = row, j = col; i >= 0 && j < N; i--, j++)
                if (board[i][j])
                        return false;
        return true;
}
bool solveNQueens(int row, int N) {
        if (row == N) {
                printSolution(N);
                return true;
        }
        bool hasSolution = false;
        for (int col = 0; col < N; col++) {
                if (isSafe(row, col, N)) {
                        board[row][col] = 1;
                        hasSolution = solveNQueens(row + 1, N) || hasSolution;
```

```
                        board[row][col] = 0;
                }
        }
        return hasSolution;
}
int main() {
        int N;
        scanf("%d", &N);
        if (N < 1 || N > MAX) {
                printf("Invalid input! N must be between 1 and 6.\n");
                return 1;
        }
        for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                        board[i][j] = 0;
        if (!solveNQueens(0, N)) {
                printf("Total solutions:0\n");
        } else {
                printf("Total solutions:%d\n", solutions);
        }
        return 0;
}
```

Output:

Test case - 1
User Output
4
Solution #1:
* Q * *
* * * Q
Q * * *
* * Q *
Solution #2:
* * Q *
Q * * *
* * * Q
* Q * *
Total solutions:2

Test case - 2
User Output
3
Total solutions:0

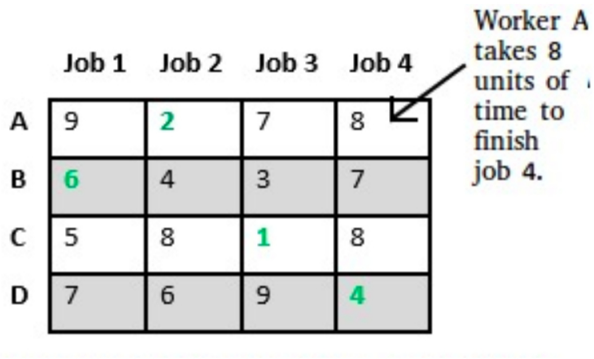
Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Job assignment***	Date:

Aim:

Let there be **N** workers and **N** jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



Green values show optimal job assignment that is A-Job 2, B-Job 1, C-Job 3, and D-Job 4.

Problem Statement:

You are given:

- An **N X N** cost matrix **C**, where **C[i][j]** represents the cost of assigning task **j** to agent **i**.

Objective: Assign each agent to exactly one task and each task to exactly one worker to minimize the total assignment cost.

Write a program for solving the assignment problem by the branch-and-bound algorithm.

Input format:

- The number of agents/tasks **N**.
- An **N X N** cost matrix represents the cost of assigning each task to each agent.

Output Format:

- The optimal assignment cost.
- The optimal assignment of tasks to workers.

Program:

`jobAssignment.c`

```
/*#include <stdio.h>

int findMinCost(  ) {

        // write the code..

}
// Driver code
int main() {
    int N;
    scanf("%d", &N);

    int** costMatrix = (int**)malloc(N * sizeof(int*));
    for (int i = 0; i < N; i++) {
        costMatrix[i] = (int*)malloc(N * sizeof(int));
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &costMatrix[i][j]);
        }
    }

    printf("Optimal Cost:%d\n", findMinCost(costMatrix, N));

    // Free dynamically allocated memory
    for (int i = 0; i < N; i++) {
        free(costMatrix[i]);
    }
    free(costMatrix);

    return 0;
}
*/
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX_N 10 // Maximum number of workers/jobs

// Function to compute the minimum cost using branch and bound
int findMinCost(int** costMatrix, int N);

// Helper function to calculate the cost of a given assignment
int calculateCost(int* assignment, int** costMatrix, int N) {
    int cost = 0;
    for (int i = 0; i < N; i++) {
        cost += costMatrix[i][assignment[i]];
    }
    return cost;
}
void branchAndBound(int** costMatrix, int* assignment, int* visited, int depth, int N,
int* minCost, int* bestAssignment, int currentCost) {
    if (depth == N) {
        // Check if current assignment is better than the best one found so far
        if (currentCost < *minCost) {
            *minCost = currentCost;
```

```
            for (int i = 0; i < N; i++) {
                bestAssignment[i] = assignment[i];
            }
        }
        return;
    }

    for (int job = 0; job < N; job++) {
        if (!visited[job]) {
            visited[job] = 1;
            assignment[depth] = job;
            branchAndBound(costMatrix, assignment, visited, depth + 1, N, minCost,
bestAssignment, currentCost + costMatrix[depth][job]);
            visited[job] = 0; // Backtrack
        }
    }
}
int findMinCost(int** costMatrix, int N) {
    int* assignment = (int*)malloc(N * sizeof(int));
    int* visited = (int*)malloc(N * sizeof(int));
    int* bestAssignment = (int*)malloc(N * sizeof(int));

    for (int i = 0; i < N; i++) {
        visited[i] = 0;
        assignment[i] = -1;
        bestAssignment[i] = -1;
    }

    int minCost = INT_MAX;

    branchAndBound(costMatrix, assignment, visited, 0, N, &minCost, bestAssignment, 0);
    for (int i = 0; i < N; i++) {
        printf("Worker %c - Job %d\n", 'A' + i, bestAssignment[i]);
    }

    free(assignment);
    free(visited);
    free(bestAssignment);

    return minCost;
}
int main() {
    int N;
    scanf("%d", &N);

    int** costMatrix = (int**)malloc(N * sizeof(int*));
    for (int i = 0; i < N; i++) {
        costMatrix[i] = (int*)malloc(N * sizeof(int));
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &costMatrix[i][j]);
        }
    }

    printf("Optimal Cost:%d\n", findMinCost(costMatrix, N));
```

```
    // Free dynamically allocated memory
    for (int i = 0; i < N; i++) {
        free(costMatrix[i]);
    }
    free(costMatrix);

    return 0;
}
```

Output:

Test case - 1
User Output
4
9 2 7 8
6 4 3 7
5 8 1 8
7 6 9 4
Worker A - Job 1
Worker B - Job 0
Worker C - Job 2
Worker D - Job 3
Optimal Cost:13

Test case - 2
User Output
3
2500 4000 3500
4000 6000 3500
2000 4000 2500
Worker A - Job 1
Worker B - Job 2
Worker C - Job 0
Optimal Cost:9500

Result:

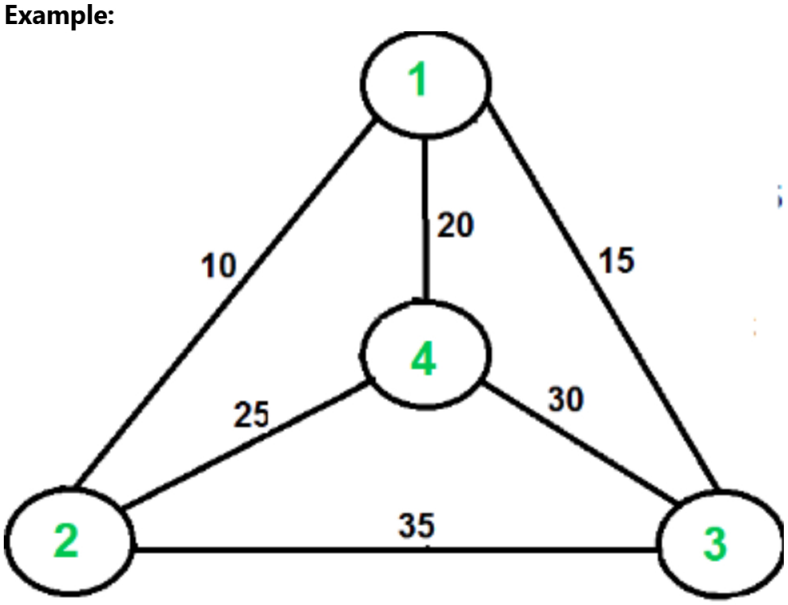
Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: *Travelling Salesman Problem*	Date:

Aim:

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

Example:



A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80

Input format:

- The first line contains an integer **n**, representing the number of cities.
- The following **n** lines contain the cost matrix of size **n x n**. Each line contains **n** integers separated by spaces, representing the cost of traveling from one city to another. If there is no direct connection between two cities, -1 is used to represent infinity.

Output format:

- The output is a single integer representing the optimal cost of the traveling salesman problem.

Program:

```
CTC30966.c
```

```
#include <stdio.h>
#include <limits.h>

#define INF 10000000

int tsp(int n, int cost[n][n]) {
    int dp[1 << n][n];

    for (int mask = 0; mask < (1 << n); mask++) {
        for (int i = 0; i < n; i++) {
            dp[mask][i] = INF;
        }
        }
    dp[1][0] = 0;

    for (int mask = 1; mask < (1 << n); mask++) {
        for (int u = 0; u < n; u++) {
            if (mask & (1 << u)) {
                for (int v = 0; v < n; v++) {
                    if ((mask & (1 << v)) == 0 && cost[u][v] != -1) {
                        dp[mask | (1 << v)][v] = dp[mask | (1 << v)][v] < dp[mask][u] +
cost[u][v] ? dp[mask | (1 << v)][v] : dp[mask][u] + cost[u][v];
                    }
                }
            }
        }
    }


    int result = INF;
    for (int i = 1; i < n; i++) {
        if (cost[i][0] != -1) {
            result = result < dp[(1 << n) - 1][i] + cost[i][0] ? result : dp[(1 << n) -
1][i] + cost[i][0];
        }
    }

    return result == INF ? -1 : result;
}

int main() {
    int n;
    scanf("%d", &n);
    int cost[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == -1) {
                cost[i][j] = INF;
            }
        }
        }
    int result = tsp(n, cost);
    printf("%d\n", result);

    return 0;
}
```

Output:

Test case - 1
User Output
3
-1 2 3
4 -1 6
1 2 -2
9

Test case - 2
User Output
4
-1 2 3 4
1 -1 1 3
4 5 -1 1
2 3 6 -1
6

Result:

Thus the above program is executed successfully and the output has been verified

Exp No:	ExpName: ***Graph Colouring***	Date:

Aim:

Given an undirected graph **G** with **n** vertices and **e** edges, the goal is to assign colors to the vertices such that no two adjacent vertices (vertices connected by an edge) have the same color, using the minimum number of colors possible.

Write a program to color the graph using the greedy method and print the result as shown in the example. Fill in the missing code in the below program.

Input Format:

- The input starts with two lines containing the number of vertices **n** and the number of edges **e**, respectively.
- The next **e** lines contain pairs of vertex indices **k** and **l**, representing an edge between vertices **k** and **l**. Note that the vertices are 0-indexed.

Output Format:

- The output should display the minimum number of colors i.e. chromatic number.

Example:

```
4 ---------------> No of vertices
5 -----------------> No of edges
0 1 --------> Each edge (u, v) u-->v and v--->u represents undirected graph
1 2
1 3
2 3
3 0
3 ------------> output representing chromatic number of the graph
```

Program:

CTC30967.c

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

// Function to find the minimum number of colors needed to color the graph
int greedy_coloring(int n, int graph[MAX_VERTICES][MAX_VERTICES]) {
    // Array to store the color assigned to each vertex
    int result[MAX_VERTICES];

    // Initialize all vertices as uncolored
    for (int i = 0; i < n; i++) {
        result[i] = -1; // -1 means no color assigned
    }

    // Assign the first color to the first vertex
    result[0] = 0;

    // Assign colors to the remaining vertices
    for (int u = 1; u < n; u++) {
        // Mark all colors as available
        int available[MAX_VERTICES] = {0};

        // Check colors of adjacent vertices and mark them as unavailable
        for (int v = 0; v < n; v++) {
            if (graph[u][v] == 1 && result[v] != -1) {
                available[result[v]] = 1; // Mark color as unavailable
            }
        }

        // Find the first available color
        for (int color = 0; color < n; color++) {
            if (!available[color]) {
                result[u] = color;
                break;
            }
        }
    }

    // The chromatic number is the maximum color assigned + 1
    int chromatic_number = 0;
    for (int i = 0; i < n; i++) {
        if (result[i] > chromatic_number) {
            chromatic_number = result[i];
        }
    }

    return chromatic_number + 1; // Add 1 to get the total number of colors used
}

int main() {
    int n, e;
    int graph[MAX_VERTICES][MAX_VERTICES] = {0}; // Adjacency matrix

    // Read number of vertices and edges
    scanf("%d", &n);
    scanf("%d", &e);
```

```
    // Read each edge and populate the adjacency matrix
    for (int i = 0; i < e; i++) {
        int k, l;
        scanf("%d %d", &k, &l);
        graph[k][l] = 1; // Undirected graph
        graph[l][k] = 1; // Undirected graph
    }

    // Get the chromatic number using the greedy coloring function
    int chromatic_number = greedy_coloring(n, graph);

    // Print the result
    printf("%d\n", chromatic_number);

    return 0;
}
```

Output:

Test case - 1
User Output
4
5
0 1
1 2
1 3
2 3
3 0
3

Test case - 2
User Output
6
7
0 1
0 2
1 2
1 3
2 3
3 4
4 5
3

Test case - 3
User Output

2
1
0 1
2

Result:

Thus the above program is executed successfully and the output has been verified