K.RAMAKRISHNAN
COLLEGE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
SAMAYAPURAM, TRICHY-621 112

Practical Record Note

Name	:	HARIPRASHATH B
Register Number	:	2303811724321033
Subject code/name	:	CGB1211 - Design and Analysis of Algorithms Laboratory
Programme	:	B.Tech - Artifical Intelligence and Data Science

CERTIFICATE

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

His/Her University Register Number is **2303811724321033**

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	14-Aug-2024	Factorial using recursion
2	14-Aug-2024	Factorial using non recursive approach
3	21-Sep-2024	String matching algorithm
4	14-Aug-2024	Sort E-Books based on IDs using Quick Sort Algorithm
5	14-Aug-2024	Knapsack
6	22-Oct-2024	Dijkstra’s Shortest Path Algorithm
7	10-Nov-2024	Huffman Encoding
8	10-Nov-2024	N Queen problem using backtracking
9	09-Oct-2024	Job assignment
10	16-Nov-2024	Travelling Salesman Problem
11	17-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 || n == 1) {
        return 1;
    }
    // Recursive case: n * factorial of (n-1)
    else {
        return n * factorial(n - 1);
        }
    // Write your code here

}
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;
    unsigned long long factorial = 1;

    // Read the integer N

    scanf("%d", &N);

    // Ensure the input is a non-negative integer
    if (N < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Calculate factorial using a loop
        for (int i = 1; i <= N; ++i) {
            factorial *= i;
        }

        // Print the result
        printf("%llu",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
```

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

void search( )
{

        // write the code..


}

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

// Function to perform the Naive string-matching algorithm
void search(char pat[], char txt[]) {
    int M = strlen(pat);  // Length of the pattern
    int N = strlen(txt);  // Length of the text

    int found = 0;

    // Slide the pattern over the text one by one
    for (int i = 0; i <= N - M; i++) {
        int j;

        // For current position i, check the pattern matches
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j]) {
                break;
            }
        }

        // If the pattern is found at index i
        if (j == M) {
            printf("Pattern found at index %d\n", i);
            found = 1;
        }
    }

    // If the pattern is not found in the text
    if (!found) {
        printf("Not Found\n");
    }
}

int main() {
    char txt[50], pat[50];

    // Read the text and pattern strings
    scanf("%s", txt);
```

```
    scanf("%s", pat);

    // Call the search function
    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];  // Pivot element
    int i = (low - 1);      // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++;    // Increment index of smaller element
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
                }
}
    // Swap the pivot element with the element at index (i+1)
    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) {
        // pi is the partitioning index, arr[pi] is now at the right place
        int pi = partition(arr, low, high);

        // Recursively sort elements before partition 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 return the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the 0-1 Knapsack problem
int knapsack(int W, int wt[], int val[], int N) {
    int dp[N+1][W+1];

    // Building the DP table
    for (int i = 0; i <= N; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (wt[i-1] <= w) {
                dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w]);
            } else {
                dp[i][w] = dp[i-1][w];
            }
        }
    }

    // The maximum value that can be achieved with the given constraints
    return dp[N][W];
}

int main() {
    int N, W;

    // Read the number of items
    scanf("%d", &N);

    // Read the knapsack capacity
    scanf("%d", &W);

    int val[N], wt[N];

    // Read the values of the items
    for (int i = 0; i < N; i++) {
        scanf("%d", &val[i]);
    }

    // Read the weights of the items
    for (int i = 0; i < N; i++) {
        scanf("%d", &wt[i]);
    }

    // Calculate the maximum value that can be achieved
    int maxValue = knapsack(W, wt, val, N);

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

    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

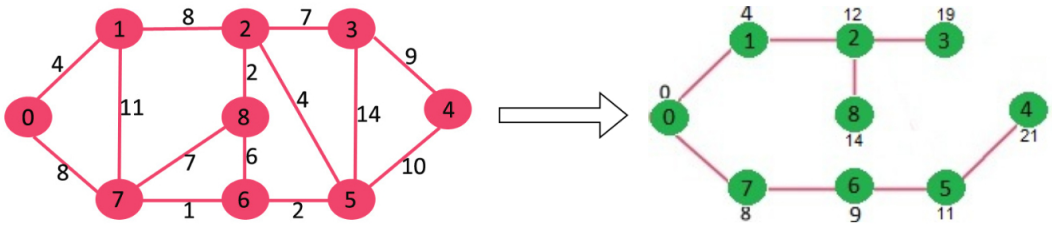
Output:

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 <limits.h>

#define MAX_VERTICES 100

int minDistance(int dist[], int visited[], int V) {
                int min = INT_MAX, min_index;
                for (int v = 0; v < V; v++) {
                        if (visited[v] == 0 && dist[v] <= min) {

                                min = dist[v], min_index = v;

                        }

                }

                return min_index;

}

void printDistances (int dist[], int V) {

        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]);

        }
}

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int V) {

        int dist[V];

        int visited[V];

        for (int i = 0; i < V; i++) {
                dist[i] = INT_MAX;
                visited[i] = 0;
        }
        dist[src] = 0;
        for (int count = 0; count < V - 1; count++) {
                int u = minDistance(dist, visited, V);
                visited[u] = 1;


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

                                if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] + graph[u][v] < dist[v]) {

                                        dist[v] = dist[u]+ graph[u][v];

                                }
```

```
                        }
        }
printDistances(dist, V);
}
int main() {

        int V;

        printf("");
        scanf("%d", &V);

        if (V > MAX_VERTICES) {

                printf("The number of vertices exceeds the maximum allowed limit
(%d).\n", MAX_VERTICES);

                        return 1;

        }


        int graph[MAX_VERTICES] [MAX_VERTICES];

        printf("");

        for (int i = 0; i< V; i++) {

                for (int j = 0; j < V; j++) {

                        scanf("%d", &graph [i][j]);

                }

        }

        int src = 0;

        dijkstra(graph, src, V);

        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:

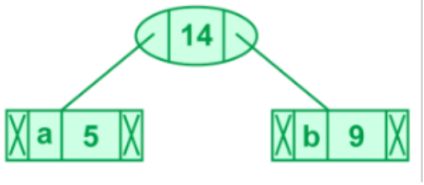
S = "abcdef"

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

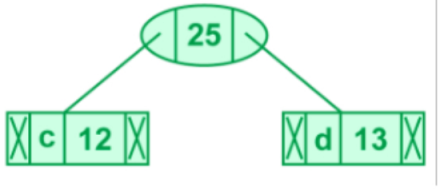
Output:

0 100 101 1100 1101 111

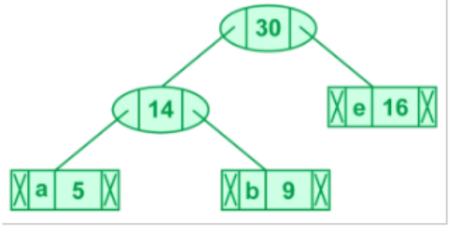
Explanation:



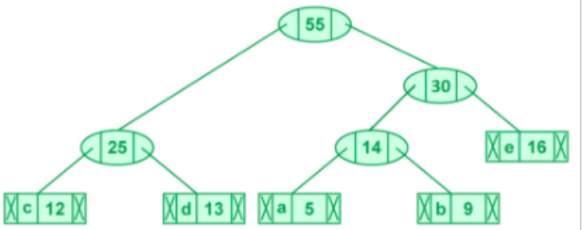
Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



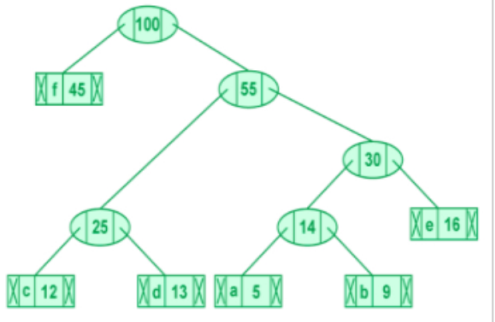
Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



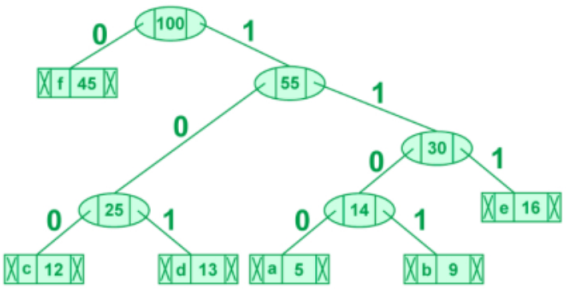
Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



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>

// Node structure for the Huffman Tree
struct Node {
    char ch;
    int freq;
    struct Node *left, *right;
};

// Create a new node
struct Node* createNode(char ch, int freq) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->ch = ch;
    newNode->freq = freq;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Swap two nodes in the array (helper function for heap)
void swap(struct Node** a, struct Node** b) {
    struct Node* t = *a;
    *a = *b;
    *b = t;
}

// Heapify function to maintain the heap property
void heapify(struct Node** heapArray, int size, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

    if (right < size && heapArray[right]->freq < heapArray[smallest]->freq)
        smallest = right;

    if (smallest != i) {
        swap(&heapArray[i], &heapArray[smallest]);
        heapify(heapArray, size, smallest);
    }
}

// Extract the node with the minimum frequency
struct Node* extractMin(struct Node** heapArray, int* size) {
    struct Node* temp = heapArray[0];
    heapArray[0] = heapArray[*size - 1];
    (*size)--;
    heapify(heapArray, *size, 0);
    return temp;
}

// Insert a new node into the heap
void insertHeap(struct Node** heapArray, int* size, struct Node* node) {
    (*size)++;
```

```
    int i = *size - 1;
    heapArray[i] = node;

    while (i && heapArray[i]->freq < heapArray[(i - 1) / 2]->freq) {
        swap(&heapArray[i], &heapArray[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

// Build the Huffman Tree
struct Node* buildHuffmanTree(char* S, int* freq, int n) {
    struct Node* left, *right, *top;

    struct Node** heapArray = (struct Node**)malloc(n * sizeof(struct Node*));
    int heapSize = 0;

    // Step 1: Build a min heap
    for (int i = 0; i < n; ++i)
        heapArray[i] = createNode(S[i], freq[i]);

    heapSize = n;
    for (int i = (heapSize - 1) / 2; i >= 0; i--)
        heapify(heapArray, heapSize, i);

    // Step 2: Extract two minimum nodes and merge them until heap contains only one node
    while (heapSize > 1) {
        left = extractMin(heapArray, &heapSize);
        right = extractMin(heapArray, &heapSize);

        top = createNode('#', left->freq + right->freq);
        top->left = left;
        top->right = right;

        insertHeap(heapArray, &heapSize, top);
    }

    return extractMin(heapArray, &heapSize);
}

// Function to print the Huffman codes using preorder traversal
void printCodes(struct Node* 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 (!root->left && !root->right) {
        for (int i = 0; i < top; i++)
            printf("%d", arr[i]);
        printf(" ");
    }
}
```

```
int main() {
    char S[27];
    int freq[26], n;

    // Input the string and frequency array
    scanf("%s", S);
    n = strlen(S);
    for (int i = 0; i < n; i++)
        scanf("%d", &freq[i]);

    // Build Huffman Tree
    struct Node* root = buildHuffmanTree(S, freq, n);

    // Print Huffman codes in preorder traversal
    int arr[100], top = 0;
    printCodes(root, arr, top);
    printf("\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

```
// Type Content here...


/*
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  // As per the problem constraint, N can be at most 6

// Global variable to track the solution number
int solutionCount = 0;

// Function to print the board
void printBoard(int board[], int N) {
    solutionCount++;
    printf("Solution #%d:\n", solutionCount);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i] == j) {
                                                printf("Q       ");
            } else {
                                        printf("*       ");
            }
        }
        printf("\n");
    }

}

// Function to check if a queen can be placed on the board at row 'row' and column 'col'
bool isSafe(int board[], int row, int col, int N) {
    for (int i = 0; i < row; i++) {
        // Check the same column and the two diagonals
        if (board[i] == col || (board[i] - i == col - row) || (board[i] + i == col +
row)) {
            return false;
        }
    }
    return true;
}

// Backtracking function to solve the N-Queens problem
```

```
int solveNQueens(int board[], int N, int row) {
    if (row == N) {
        printBoard(board, N);  // Print the solution
        return 1;  // A solution is found
    }

    int solutions = 0;
    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {
            board[row] = col;  // Place the queen
            solutions += solveNQueens(board, N, row + 1);  // Recur to place the next
queen
            board[row] = -1;  // Backtrack
        }
    }

    return solutions;  // Return the total number of solutions found
}

int main() {
    int N;

    scanf("%d", &N);

    if (N < 1 || N > MAX) {
        printf("N must be between 1 and 6.\n");
        return 1;
    }

    int board[N];
    for (int i = 0; i < N; i++) {
        board[i] = -1;  // Initialize the board with -1 indicating no queen placed in
that row
    }

    int totalSolutions = solveNQueens(board, N, 0);  // Start solving from the first row
    printf("Total solutions:%d\n", totalSolutions);

    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(  ) {


}
// 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 <stdbool.h>
#include <limits.h>

#define INF INT_MAX

// Global variables to track optimal assignments
int* optimalAssignment;
int minCost;

// Function to find the minimum cost assignment using Branch and Bound
void solve(int** costMatrix, bool* taskTaken, int* workerAssigned, int worker, int
currentCost, int N) {
    if (worker == N) {  // All workers have been assigned tasks
        if (currentCost < minCost) {
            minCost = currentCost;
            // Update the optimal assignment
            for (int i = 0; i < N; i++) {
                optimalAssignment[i] = workerAssigned[i];
            }
        }
        return;
```

```
    }

    for (int task = 0; task < N; task++) {
        if (!taskTaken[task]) {
            taskTaken[task] = true;
            workerAssigned[worker] = task;

            // Recursive call for next worker
            solve(costMatrix, taskTaken, workerAssigned, worker + 1, currentCost +
costMatrix[worker][task], N);

            taskTaken[task] = false;  // Backtrack
        }
    }
}

// Function to initialize the branch and bound process
int findMinCost(int** costMatrix, int N) {
    int* workerAssigned = (int*)malloc(N * sizeof(int));  // To store task assigned to
each worker
    bool* taskTaken = (bool*)malloc(N * sizeof(bool));    // To track assigned tasks

    // Initialize taskTaken array
    for (int i = 0; i < N; i++) {
        taskTaken[i] = false;
    }

    // Initialize minCost to a large value
    minCost = INF;

    // Start solving from worker 0 with initial cost 0
    solve(costMatrix, taskTaken, workerAssigned, 0, 0, N);

    // Cleanup
    free(workerAssigned);
    free(taskTaken);

    return minCost;
}

// 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));
    }

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

```
    // Allocate memory to store the optimal assignment
    optimalAssignment = (int*)malloc(N * sizeof(int));

    // Find the minimum cost
    int optimalCost = findMinCost(costMatrix, N);

    // Print the optimal assignment
    for (int i = 0; i < N; i++) {
        printf("Worker %c - Job %d\n", 'A' + i, optimalAssignment[i]);
    }

    // Print the optimal cost
    printf("Optimal Cost:%d\n", optimalCost);

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

    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:

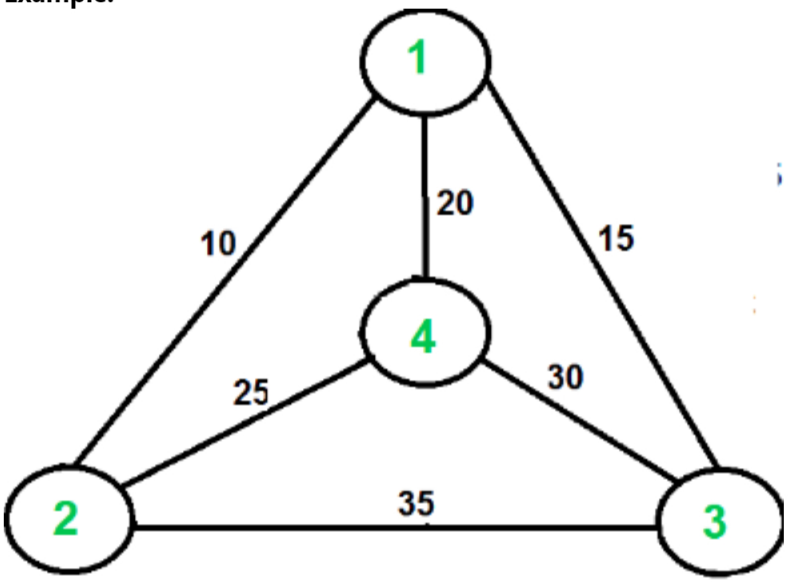
Thus the above program is executed successfully and the output has been verified

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

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

#define INF INT_MAX

// Function to calculate the total cost of a specific tour
int calculateTourCost(int costMatrix[][100], int tour[], int n) {
    int totalCost = 0;
    for (int i = 0; i < n - 1; i++) {
        int fromCity = tour[i];
        int toCity = tour[i + 1];
        if (costMatrix[fromCity][toCity] == -1) { // No direct connection
            return INF;
        }
        totalCost += costMatrix[fromCity][toCity];
    }

    // Add the cost of returning to the start city
    if (costMatrix[tour[n - 1]][tour[0]] == -1) { // No direct connection
        return INF;
    }
    totalCost += costMatrix[tour[n - 1]][tour[0]];
    return totalCost;
}

// Function to generate permutations and solve TSP
void solveTSP(int n, int costMatrix[][100]) {
    int cities[100];
    for (int i = 0; i < n; i++) {
        cities[i] = i;
    }

    int minCost = INF;

    // Generate all permutations using next_permutation-like logic
    do {
        int currentCost = calculateTourCost(costMatrix, cities, n);
        if (currentCost < minCost) {
            minCost = currentCost;
        }

        // Generate the next permutation
        int i = n - 1;
        while (i > 0 && cities[i - 1] >= cities[i]) i--;
        if (i == 0) break;

        int j = n - 1;
        while (cities[j] <= cities[i - 1]) j--;

        int temp = cities[i - 1];
        cities[i - 1] = cities[j];
        cities[j] = temp;

        for (int k = i, l = n - 1; k < l; k++, l--) {
            temp = cities[k];
            cities[k] = cities[l];
```

```
            cities[l] = temp;
        }

    } while (true);

    printf("%d\n", minCost);
}

int main() {
    int n;
    scanf("%d", &n);

    int costMatrix[100][100];

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

    solveTSP(n, costMatrix);

    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>
#include <stdbool.h>

// Function to find the chromatic number using greedy coloring
int greedy_coloring(int n, int** edges, int* degrees) {
    // Initialize all vertices as uncolored (-1)
    int result[n];
    for (int i = 0; i < n; i++) {
        result[i] = -1;
    }

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

    // A temporary array to store the availability of colors
    bool available[n];
    for (int i = 0; i < n; i++) {
        available[i] = false;
    }

    // Assign colors to the remaining vertices
    for (int u = 1; u < n; u++) {
        // Process all adjacent vertices and mark their colors as unavailable
        for (int i = 0; i < degrees[u]; i++) {
            int adj = edges[u][i];
            if (result[adj] != -1) {
                available[result[adj]] = true;
            }
        }

        // Find the first available color
        int color = 0;
        while (color < n && available[color]) {
            color++;
        }

        // Assign the found color to the vertex u
        result[u] = color;

        // Reset the availability of colors for the next iteration
        for (int i = 0; i < degrees[u]; i++) {
            int adj = edges[u][i];
            if (result[adj] != -1) {
                available[result[adj]] = false;
            }
        }
    }

    // The chromatic number is the maximum color used + 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;
}

int main() {
    // Input number of vertices and edges
    int n, e;

    scanf("%d", &n);

    scanf("%d", &e);

    // Create an adjacency list for the graph
    int* degrees = (int*)malloc(n * sizeof(int));
    int** edges = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        degrees[i] = 0;
        edges[i] = (int*)malloc(n * sizeof(int));  // Maximum possible degree is n-1
    }

    // Input the edges
    int u, v;
    for (int i = 0; i < e; i++) {
        scanf("%d %d", &u, &v);
        edges[u][degrees[u]++] = v;
        edges[v][degrees[v]++] = u;
    }

    // Get the chromatic number using the greedy coloring method
    int chromatic_number = greedy_coloring(n, edges, degrees);

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

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

    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