SAVEETHA SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CSA06 – DESIGN AND ANALYSIS OF ALGORITHMS

LAB MANUAL

1.Write a program to Print Fibonacci Series using recursion.

Aim: To write a C program to print the Fibonacci series using recursion

Algorithm:

1. Define a recursive function `fibonacci(int n)` that returns the nth Fibonacci number.
2. In the main function:
    - Take the number of terms as input from the user.
    - Print the Fibonacci series up to the specified number of terms using the recursive function.

C Code:

```
#include <stdio.h>

// Function to calculate the nth Fibonacci number using recursion
int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return (fibonacci(n - 1) + fibonacci(n - 2));
}

int main() {
    int n, i;

    // Input: Number of terms to be printed
    printf("Enter the number of terms: ");
    scanf("%d", &n);

    // Output: Fibonacci series
    printf("Fibonacci Series: ");
    for (i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of terms: 10
```

Sample Output:

```
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34
```

Result: Thus, the program has been executed successfully.

2.Write a program to check the given no is Armstrong or not.

Aim: To write a C program to check if a given number is an Armstrong number:

Algorithm:

1. Take an integer input from the user.
2. Calculate the number of digits in the input number.
3. Calculate the sum of each digit raised to the power of the number of digits.
4. Compare the sum with the original number.
5. If they are equal, the number is an Armstrong number; otherwise, it is not.

C Code:

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

// Function to count the number of digits in a number
int countDigits(int num) {
    int count = 0;
    while (num != 0) {
        num /= 10;
        count++;
    }
    return count;
}

// Function to check if a number is an Armstrong number
int isArmstrong(int num) {
    int sum = 0, temp, remainder, n = 0;

    // Calculate the number of digits
    n = countDigits(num);

    temp = num;
    while (temp != 0) {
        remainder = temp % 10;
        sum += pow(remainder, n);
        temp /= 10;
    }

    // Check if sum of digits each raised to the power of number of digits
is equal to num
    if (sum == num)
```

```
        return 1;
    else
        return 0;
}

int main() {
    int num;

    // Input: Number to be checked
    printf("Enter a number: ");
    scanf("%d", &num);

    // Output: Check if the number is an Armstrong number
    if (isArmstrong(num))
        printf("%d is an Armstrong number.\n", num);
    else
        printf("%d is not an Armstrong number.\n", num);

    return 0;
}
```

Sample Input and Output:

Sample Input 1:

```
Enter a number: 153
```

Sample Output 1:

```
153 is an Armstrong number.
```

Sample Input 2:

```
Enter a number: 123
```

Sample Output 2:

```
   123   not an Armstrong number.
```

Result: Thus, the program has been executed successfully.

3.Program to find the GCD of two numbers .

Aim: To find the Greatest Common Divisor (GCD) of two numbers, you can use the Euclidean algorithm. The Euclidean algorithm is efficient and works as follows:

1. If the second number is 0, the GCD is the first number.
2. Otherwise, the GCD of two numbers `a` and `b` is the same as the GCD of `b` and `a % b`.

Here's a C program to find the GCD of two numbers using the Euclidean algorithm:

Algorithm:

1. Take two integers as input from the user.

2. Implement the Euclidean algorithm to find the GCD.
3. Print the GCD.

C Code:

```
#include <stdio.h>

// Function to calculate GCD using the Euclidean algorithm
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

int main() {
    int num1, num2, result;

    // Input: Two numbers
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    // Calculate GCD
    result = gcd(num1, num2);

    // Output: GCD of the two numbers
    printf("GCD of %d and %d is %d.\n", num1, num2, result);

    return 0;
}
```

Sample Input and Output:

Sample Input 1:

```
Enter two integers: 56 98
```

Sample Output 1:

```
GCD of 56 and 98 is 14.
```

Sample Input 2:

```
Enter two integers: 48 18
```

Sample Output 2:

```
GCD of 48 and 18 is 6.
```

Result: Thus, the program has been executed successfully.

4. Write a program to get the largest element of an array.

Aim: To find the largest element in an array, you can iterate through the array and keep track of the largest element encountered so far.

Here is a C program to find the largest element in an array:

Algorithm:

1. Take the number of elements and the elements of the array as input from the user.
2. Initialize the largest element with the first element of the array.
3. Iterate through the array and update the largest element if a larger element is found.
4. Print the largest element.

C Code:

```
#include <stdio.h>

int main() {
    int n, i;
    int largest;

    // Input: Number of elements in the array
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];

    // Input: Elements of the array
    printf("Enter the elements of the array:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Initialize the largest element with the first element
    largest = arr[0];

    // Iterate through the array to find the largest element
    for (i = 1; i < n; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }

    // Output: Largest element in the array
    printf("The largest element in the array is %d.\n", largest);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
12 34 56 78 90
```

Sample Output:

```
The largest element in the array is 90.
```

Result: Thus, the program has been executed successfully.

5. Write a program to find the Factorial of a number

Aim: To find the factorial of a number in C, you can use either an iterative or a recursive approach. Here, I'll provide the iterative approach, which is straightforward and easy to understand.

Algorithm:

1. Take an integer input from the user.
2. Initialize a variable to store the factorial result, starting with 1.
3. Use a loop to multiply the result variable by each integer from 1 up to the input number.
4. Print the factorial result.

C Code:

```
#include <stdio.h>

// Function to calculate the factorial of a number using iteration
unsigned long long factorial(int n) {
    unsigned long long fact = 1;
    for (int i = 1; i <= n; ++i) {
        fact *= i;
    }
    return fact;
}

int main() {
    int num;

    // Input: Number to find the factorial of
    printf("Enter a number: ");
    scanf("%d", &num);

    // Check if the input number is non-negative
    if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Output: Factorial of the number
        printf("Factorial of %d is %llu.\n", num, factorial(num));
    }

    return 0;
}
```

Sample Input and Output:

Sample Input 1:

```
Enter a number: 5
```

Sample Output 1:

```
Factorial of 5 is 120.
```

Sample Input 2:

```
Enter a number: 0
```

Sample Output 2:

```
Factorial of 0 is 1.
```

Result: Thus, the program has been executed successfully.

6.Write a program to check a number is a prime number or not .

Aim: To check if a number is a prime number in C, you can use a simple algorithm that iterates from 2 to the square root of the number and checks if the number is divisible by any of these values. If it is divisible by any number other than 1 and itself, it is not a prime number.

Algorithm:

1. Take an integer input from the user.
2. If the number is less than or equal to 1, it is not a prime number.
3. Iterate from 2 to the square root of the number.
4. If the number is divisible by any of these values, it is not a prime number.
5. If no divisors are found, the number is a prime number.

C Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

// Function to check if a number is prime
bool isPrime(int num) {
    if (num <= 1) {
        return false;
    }
    for (int i = 2; i <= sqrt(num); i++) {
        if (num % i == 0) {
            return false;
        }
    }
    return true;
}

int main() {
    int num;

    // Input: Number to be checked
    printf("Enter a number: ");
```

```
    scanf("%d", &num);

    // Output: Check if the number is a prime number
    if (isPrime(num)) {
        printf("%d is a prime number.\n", num);
    } else {
        printf("%d is not a prime number.\n", num);
    }

    return 0;
}
```

Sample Input and Output:

Sample Input 1:

```
Enter a number: 17
```

Sample Output 1:

```
17 is a prime number.
```

Sample Input 2:

```
Enter a number: 20
```

Sample Output 2:

```
20 is not a prime number.
```

Result: Thus, the program has been executed successfully.

7.Write a program to perform Selection sort.

Aim : To write a 'C' program to perform selection sort

Algorithm:

1. Iterate over the array.
2. For each position in the array, assume it is the minimum.
3. Compare this minimum with the rest of the unsorted part of the array to find the actual minimum.
4. Swap the found minimum element with the first element of the unsorted part.
5. Repeat until the entire array is sorted.

C Code:

```
#include <stdio.h>

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;
```

```
    // Iterate over the array
    for (i = 0; i < n-1; i++) {
        // Assume the minimum element is the first element of the unsorted
part
        min_idx = i;

        // Find the actual minimum element in the unsorted part
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element of the
unsorted part
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

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

int main() {
    int n, i;

    // Input: Number of elements in the array
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];

    // Input: Elements of the array
    printf("Enter the elements of the array:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Perform selection sort
    selectionSort(arr, n);

    // Output: Sorted array
    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
64 25 12 22 11
```

Sample Output:

```
Sorted array:
11 12 22 25 64
```

Result: Thus, the program has been executed successfully.

8.Write a program to perform Bubble sort

Aim : To write a 'C' program to perform bubble sort

Algorithm:

1. Iterate through the array.
2. Compare each pair of adjacent elements.
3. Swap them if they are in the wrong order.
4. Repeat the process for the entire array until no swaps are needed.

C Code:

```
#include <stdio.h>

// Function to perform bubble sort
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++) {
            // Compare and swap if elements are in the wrong order
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

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

int main() {
    int n, i;

    // Input: Number of elements in the array
    printf("Enter the number of elements in the array: ");
```

```
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];

    // Input: Elements of the array
    printf("Enter the elements of the array:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Perform bubble sort
    bubbleSort(arr, n);

    // Output: Sorted array
    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
64 34 25 12 22
```

Sample Output:

```
Sorted array:
12 22 25 34 64
```

Result: Thus, the program has been executed successfully.

9.Write a program for to multiply two Matrix

Aim : To write a ‘C’ program to perform matrix multiplication

Algorithm:

1. Take input for the dimensions and elements of the two matrices.
2. Initialize a result matrix with appropriate dimensions.
3. Use nested loops to calculate the product of the two matrices.
4. Store the result in the result matrix.
5. Print the result matrix.

C Code:

```
#include <stdio.h>

// Function to multiply two matrices
void multiplyMatrices(int firstMatrix[][10], int secondMatrix[][10], int
resultMatrix[][10], int r1, int c1, int r2, int c2) {
```

```
    int i, j, k;

    // Initialize the result matrix to 0
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++) {
            resultMatrix[i][j] = 0;
        }
    }

    // Multiply the matrices
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++) {
            for (k = 0; k < c1; k++) {
                resultMatrix[i][j] += firstMatrix[i][k] *
secondMatrix[k][j];
            }
        }
    }
}

// Function to print a matrix
void printMatrix(int matrix[][10], int row, int col) {
    int i, j;
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int r1, c1, r2, c2, i, j;

    // Input: Dimensions of the first matrix
    printf("Enter the number of rows and columns of the first matrix: ");
    scanf("%d %d", &r1, &c1);

    // Input: Dimensions of the second matrix
    printf("Enter the number of rows and columns of the second matrix: ");
    scanf("%d %d", &r2, &c2);

    // Ensure the matrices can be multiplied
    if (c1 != r2) {
        printf("Error! Column of the first matrix must be equal to row of
the second matrix.\n");
        return -1;
    }

    // Declare the matrices
    int firstMatrix[10][10], secondMatrix[10][10], resultMatrix[10][10];

    // Input: Elements of the first matrix
    printf("Enter the elements of the first matrix:\n");
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c1; j++) {
            scanf("%d", &firstMatrix[i][j]);
        }
    }

    // Input: Elements of the second matrix
```

```
    printf("Enter the elements of the second matrix:\n");
    for (i = 0; i < r2; i++) {
        for (j = 0; j < c2; j++) {
            scanf("%d", &secondMatrix[i][j]);
        }
    }

    // Multiply the matrices
    multiplyMatrices(firstMatrix, secondMatrix, resultMatrix, r1, c1, r2,
c2);

    // Output: Result matrix
    printf("Resultant Matrix:\n");
    printMatrix(resultMatrix, r1, c2);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of rows and columns of the first matrix: 2 3
Enter the number of rows and columns of the second matrix: 3 2
Enter the elements of the first matrix:
1 2 3
4 5 6
Enter the elements of the second matrix:
7 8
9 10
11 12
```

Sample Output:

```
Resultant Matrix:
58 64
139 154
```

Result: Thus, the program has been executed successfully.

10.Write a program for to check whether a given String is Palindrome or not

Aim : To write a ‘C’ program to check whether a given String is Palindrome or not

Algorithm:

1. Take a string as input from the user.
2. Initialize two indices, one starting from the beginning and the other from the end of the string.
3. Compare the characters at these indices.
4. If all corresponding characters match, the string is a palindrome.
5. If any character does not match, the string is not a palindrome.

C Code:

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

// Function to check if a string is a palindrome
bool isPalindrome(char str[]) {
    int left = 0;
    int right = strlen(str) - 1;

    while (left < right) {
        if (str[left] != str[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

int main() {
    char str[100];

    // Input: String to be checked
    printf("Enter a string: ");
    scanf("%s", str);

    // Output: Check if the string is a palindrome
    if (isPalindrome(str)) {
        printf("\"%s\" is a palindrome.\n", str);
    } else {
        printf("\"%s\" is not a palindrome.\n", str);
    }

    return 0;
}
```

Sample Input and Output:

Sample Input 1:

```
Enter a string: racecar
```

Sample Output 1:

```
"racecar" is a palindrome.
```

Sample Input 2:

```
Enter a string: hello
```

Sample Output 2:

```
"hello" is not a palindrome.
```

Result: Thus, the program has been executed successfully.

11.Write a program for to copy one string to another

Aim : To write a ‘C’ program to copy one string to another

Algorithm:

1. **Input the Source String**: Read the string to be copied.
2. **Initialize Destination String**: Prepare a destination string with sufficient space to hold the copied string.
3. **Copy the String**:
    - **Using `strcpy`**: Use the standard library function to copy.
    - **Manual Copying**: Copy each character from the source to the destination, including the null terminator.
4. **Output the Result**: Print the destination string to verify the copy operation.

C Code (Using `strcpy`):

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

int main() {
    char source[100], destination[100];

    // Input: Source string
    printf("Enter the source string: ");
    fgets(source, sizeof(source), stdin);

    // Remove the newline character if present
    source[strcspn(source, "\n")] = '\0';

    // Copy the string using strcpy
    strcpy(destination, source);

    // Output: Destination string
    printf("Destination string: %s\n", destination);

    return 0;
}
```

C Code (Manual Copying):

```
#include <stdio.h>

// Function to manually copy a string
void copyString(char *source, char *destination) {
    while (*source != '\0') {
        *destination = *source;
        source++;
        destination++;
    }
    *destination = '\0'; // Null-terminate the destination string
}

int main() {
    char source[100], destination[100];
```

```
    // Input: Source string
    printf("Enter the source string: ");
    fgets(source, sizeof(source), stdin);

    // Remove the newline character if present
    source[strcspn(source, "\n")] = '\0';

    // Copy the string manually
    copyString(source, destination);

    // Output: Destination string
    printf("Destination string: %s\n", destination);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the source string: Hello, World!
```

Sample Output:

```
Destination string: Hello, World!
```

Explanation:

- **Using `strcpy` Function**:
  - `fgets` reads the source string from the user, including spaces and up to a newline character.
  - `strcpy` copies the string from `source` to `destination`, handling the null terminator automatically.
- **Manual Copying**:
  - A custom function `copyString` is used to copy each character from `source` to `destination`.
  - This method shows how copying is done character-by-character and ensures the string is null-terminated.

Both methods effectively copy a string and produce the same result. The choice of method depends on whether you prefer using standard library functions or understanding the underlying mechanics with manual copying.

Result: Thus, the program has been executed successfully.

12.Write a Program to perform binary search.
Aim: To write a 'C' program to perform binary search

Algorithm:

1. **Initialize**: Set the initial boundaries for the search (low and high).

2. **Mid Calculation**: Calculate the middle index.
3. **Comparison**:
    - If the target value is equal to the middle element, the search is successful.
    - If the target value is less than the middle element, repeat the search in the left half.
    - If the target value is greater than the middle element, repeat the search in the right half.

    - repeat the search in the left half.
    - If the target value is greater than the middle element, repeat the search in the right half.
2. **Repeat**: Continue the process until the target is found or the boundaries converge.

C Code:

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;
    int mid;

    while (low <= high) {
        mid = low + (high - low) / 2;

        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid; // Target found
        }

        // If target is smaller, ignore the right half
        if (arr[mid] > target) {
            high = mid - 1;
        }
        // If target is larger, ignore the left half
        else {
            low = mid + 1;
        }
    }

    return -1; // Target not found
}

int main() {
    int n, target, result;

    // Input: Number of elements
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];

    // Input: Elements of the array
```

```
    printf("Enter the elements of the sorted array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input: Target value to search
    printf("Enter the target value to search: ");
    scanf("%d", &target);

    // Perform binary search
    result = binarySearch(arr, n, target);

    // Output: Result of binary search
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of elements in the array: 10
Enter the elements of the sorted array:
2 4 6 8 10 12 14 16 18 20
Enter the target value to search: 14
```

Sample Output:

```
Element 14 found at index 6.
```

Explanation:

- **Input**: The program takes a sorted array and a target value from the user.
- **Binary Search**: It calculates the middle index, compares the target value, and narrows down the search interval accordingly.
- **Output**: The program reports the index of the target value if found, or indicates that the target is not in the array.

Ensure that the array is sorted before applying binary search, as it only works on sorted arrays.

Result: Thus, the program has been executed successfully.

13,Write a program to print the reverse of a string

Aim: To write a ‘C’ program to print the reverse of a string

Algorithm

1. **Input the String:**
    - Read the input string from the user.
2. **Determine the Length of the String:**
    - Find the length of the string to know how many characters to process.
3. **Reverse the String:**
    - Use a loop to print the characters of the string from the end to the beginning.
4. **Output the Reversed String:**
    - Print each character in reverse order.

Method 1: Using a Loop

C Code:

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

// Function to print the reverse of a string
void printReverse(char str[]) {
    int length = strlen(str);
    for (int i = length - 1; i >= 0; i--) {
        printf("%c", str[i]);
    }
    printf("\n");
}

int main() {
    char str[100];

    // Input: String to be reversed
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    // Remove the newline character if present
    str[strcspn(str, "\n")] = '\0';

    // Print the reverse of the string
    printf("Reversed string: ");
    printReverse(str);

    return 0;
}
```

Method 2: Using a Temporary Array

C Code:

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

// Function to reverse a string and store it in a temporary array
void reverseString(char str[], char reversed[]) {
    int length = strlen(str);
    for (int i = 0; i < length; i++) {
        reversed[i] = str[length - 1 - i];
    }
    reversed[length] = '\0'; // Null-terminate the reversed string
```

```
}

int main() {
    char str[100], reversed[100];

    // Input: String to be reversed
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    // Remove the newline character if present
    str[strcspn(str, "\n")] = '\0';

    // Reverse the string
    reverseString(str, reversed);

    // Output: Reversed string
    printf("Reversed string: %s\n", reversed);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter a string: OpenAI
```

Sample Output:

```
Reversed string: IApneO
```

Explanation:

- **Method 1 (Using a Loop)**:
  - This method directly prints the characters from the end to the beginning of the string.
- **Method 2 (Using a Temporary Array)**:
  - This method first copies the characters in reverse order into a new array and then prints the reversed string.

Both methods effectively reverse a string, and you can choose based on whether you prefer to use a temporary array or simply print in reverse order.

Result: Thus, the program has been executed successfully.

14.Write a program to find the length of a string.

Aim: To write a ‘C’ program to find the length of a string.

Algorithm

1. **Initialize a Counter:**

- Set a variable `length` to `0`. This will keep track of the number of characters in the string.

2. **Traverse the String:**
   - Iterate through each character of the string using a loop.
   - Continue incrementing the `length` counter until you reach the null terminator (`'\0'`), which marks the end of the string.
3. **Return the Length:**
   - After exiting the loop, the `length` variable will contain the total number of characters in the string.

Method 1: Using `strlen` Function

C Code:

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

int main() {
    char str[100];

    // Input: String from the user
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    // Remove the newline character if present
    str[strcspn(str, "\n")] = '\0';

    // Calculate the length of the string using strlen
    int length = strlen(str);

    // Output: Length of the string
    printf("Length of the string: %d\n", length);

    return 0;
}
```

Method 2: Using a Custom Function

C Code:

```
#include <stdio.h>

// Function to calculate the length of a string manually
int stringLength(char str[]) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}

int main() {
    char str[100];

    // Input: String from the user
    printf("Enter a string: ");
```

```
    fgets(str, sizeof(str), stdin);

    // Remove the newline character if present
    str[strcspn(str, "\n")] = '\0';

    // Calculate the length of the string using the custom function
    int length = stringLength(str);

    // Output: Length of the string
    printf("Length of the string: %d\n", length);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter a string: Hello, World!
```

Sample Output:

```
Length of the string: 13
```

Explanation:

- **Method 1 (Using `strlen`)**:
  - The `strlen` function directly provides the length of the string, excluding the null terminator. The `fgets` function is used to read the string input, and `strcspn` is used to remove any newline character that `fgets` might include.
- **Method 2 (Using a Custom Function)**:
  - This method manually counts the characters in the string until the null terminator is encountered, thus providing the length of the string.

Both methods effectively find the length of a string, and you can choose based on whether you prefer to use the built-in function or write your own.

Result: Thus, the program has been executed successfully.

15.Write a program to perform Strassen’s Matrix Multiplication.

Aim: To write a ‘C’ program to perform Strassen’s matrix multiplication

Overview of Strassen's Algorithm:

1. **Divide**: Divide each matrix into four submatrices.
2. **Conquer**: Calculate seven intermediate matrices (products) using a set of recursive multiplications.
3. **Combine**: Combine these intermediate matrices to get the final result.

Algorithm Steps:

1. **Base Case**: When the matrix size is 1x1, perform direct multiplication.
2. **Divide**: For larger matrices, split them into four submatrices.
3. **Compute**: Calculate the seven products using Strassen's formula.
4. **Combine**: Combine these products to get the final result.

C Code:

Below is the implementation of Strassen's matrix multiplication algorithm in C:

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

#define SIZE 2  // For simplicity, using a fixed size matrix. Adjust as
needed.

// Function to add two matrices
void add(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
}

// Function to subtract one matrix from another
void subtract(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE])
{
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
}

// Function to multiply matrices using Strassen's algorithm
void strassen(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE]) {
    if (SIZE == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    // Allocate memory for submatrices
    int A11[SIZE / 2][SIZE / 2], A12[SIZE / 2][SIZE / 2];
    int A21[SIZE / 2][SIZE / 2], A22[SIZE / 2][SIZE / 2];
    int B11[SIZE / 2][SIZE / 2], B12[SIZE / 2][SIZE / 2];
    int B21[SIZE / 2][SIZE / 2], B22[SIZE / 2][SIZE / 2];
    int M1[SIZE / 2][SIZE / 2], M2[SIZE / 2][SIZE / 2];
    int M3[SIZE / 2][SIZE / 2], M4[SIZE / 2][SIZE / 2];
    int M5[SIZE / 2][SIZE / 2], M6[SIZE / 2][SIZE / 2];
    int M7[SIZE / 2][SIZE / 2];
    int temp1[SIZE / 2][SIZE / 2], temp2[SIZE / 2][SIZE / 2];

    // Splitting the matrices into submatrices
    for (int i = 0; i < SIZE / 2; i++) {
        for (int j = 0; j < SIZE / 2; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + SIZE / 2];
            A21[i][j] = A[i + SIZE / 2][j];
            A22[i][j] = A[i + SIZE / 2][j + SIZE / 2];
```

```
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + SIZE / 2];
            B21[i][j] = B[i + SIZE / 2][j];
            B22[i][j] = B[i + SIZE / 2][j + SIZE / 2];
        }
    }

    // Compute M1 to M7
    add(A11, A22, temp1);
    add(B11, B22, temp2);
    strassen(temp1, temp2, M1);

    add(A21, A22, temp1);
    strassen(temp1, B11, M2);

    subtract(B12, B22, temp1);
    strassen(A11, temp1, M3);

    subtract(B21, B11, temp1);
    strassen(A22, temp1, M4);

    add(A11, A12, temp1);
    strassen(temp1, B22, M5);

    subtract(A21, A11, temp1);
    add(B11, B12, temp2);
    strassen(temp1, temp2, M6);

    subtract(A12, A22, temp1);
    add(B21, B22, temp2);
    strassen(temp1, temp2, M7);

    // Combine the results
    add(M1, M4, temp1);
    subtract(temp1, M5, temp2);
    add(temp2, M7, C);

    add(M3, M5, temp1);
    C[0][1] = temp1[0][0];

    add(M2, M4, C);
    C[1][0] = C[0][0];

    add(M1, M3, temp1);
    subtract(temp1, M6, C);
    C[1][1] = C[0][0];
}

int main() {
    int A[SIZE][SIZE] = {{1, 2}, {3, 4}};
    int B[SIZE][SIZE] = {{5, 6}, {7, 8}};
    int C[SIZE][SIZE];

    // Perform Strassen's matrix multiplication
    strassen(A, B, C);

    // Output: Result matrix
    printf("Result matrix C:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", C[i][j]);
```

```
        }
        printf("\n");
    }

    return 0;
}
```

Sample Input and Output:

Sample Output:

```
Result matrix C:
19 22
43 50
```

Explanation:

1. **Submatrices Creation**: The matrices are divided into four submatrices each.
2. **Strassen's Multiplications**: Seven products (`M1` to `M7`) are computed based on Strassen's formula.
3. **Combining Results**: The final result matrix `C` is computed using these intermediate results.

Note: Strassen's algorithm is most effective for larger matrices. For simplicity, the provided code uses 2x2 matrices, but you can extend it to larger matrices by modifying the size and managing submatrices accordingly.

Result: Thus, the program has been executed successfully.

16.Write a program to perform Merge Sort.

Aim: To write a 'C' programto perform merge sort

Algorithm: Here's a step-by-step breakdown of the Merge Sort algorithm:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Combine**: Merge the two sorted halves to produce a sorted array.

C Code for Merge Sort

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

// Function to merge two halves
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int* L = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));
```

```
    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    // Merge the temporary arrays back into arr[left..right]
    int i = 0; // Initial index of the first subarray
    int j = 0; // Initial index of the second subarray
    int k = left; // Initial index of the merged subarray

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k++] = L[i++];
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k++] = R[j++];
    }

    // Free allocated memory
    free(L);
    free(R);
}

// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort the first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

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

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Given array:\n");
    printArray(arr, size);

    mergeSort(arr, 0, size - 1);

    printf("Sorted array:\n");
    printArray(arr, size);

    return 0;
}
```

Sample Input and Output:

Sample Output:

```
Given array:
12 11 13 5 6 7
Sorted array:
5 6 7 11 12 13
```

Explanation:

1. **Merge Function**: This function merges two sorted subarrays into one sorted array. It uses temporary arrays to hold the subarrays and then merges them back into the original array.
2. **Merge Sort Function**: This function divides the array into halves and recursively sorts each half. It then merges the sorted halves using the `merge` function.
3. **Print Function**: A helper function to print the contents of the array.

This implementation of Merge Sort works efficiently for arrays of various sizes and maintains a time complexity of O(n log n), making it suitable for large datasets.

Result: Thus, the program has been executed successfully.

17.Using Divide and Conquer strategy to find Max and Min value in the list.

Aim: To write a 'C' program to find Min and Max value in the list.

Algorithm:

1. **Divide**: Split the list into two halves.
2. **Conquer**: Recursively find the maximum and minimum values in each half.
3. **Combine**: Compare the results from the two halves to get the overall maximum and minimum.

Steps:

1. **Base Case**: If the list has one element, return that element as both the maximum and minimum.
2. **Divide**: Split the list into two halves.
3. **Recursive Case**: Find the maximum and minimum for each half.

4. **Combine**: Compare the maximum values from the two halves to get the overall maximum, and similarly compare the minimum values to get the overall minimum.

C Code:

Here is a C program implementing the Divide and Conquer strategy to find the maximum and minimum values in an array:

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

// Function to find maximum and minimum using divide and conquer
void findMaxMin(int arr[], int left, int right, int *max, int *min) {
    if (left == right) {
        // Base case: only one element
        *max = arr[left];
        *min = arr[left];
    } else if (right == left + 1) {
        // Base case: two elements
        if (arr[left] > arr[right]) {
            *max = arr[left];
            *min = arr[right];
        } else {
            *max = arr[right];
            *min = arr[left];
        }
    } else {
        // Divide
        int mid = left + (right - left) / 2;
        int leftMax, leftMin, rightMax, rightMin;

        // Conquer
        findMaxMin(arr, left, mid, &leftMax, &leftMin);
        findMaxMin(arr, mid + 1, right, &rightMax, &rightMin);

        // Combine
        *max = (leftMax > rightMax) ? leftMax : rightMax;
        *min = (leftMin < rightMin) ? leftMin : rightMin;
    }
}

int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int max, min;

    // Find maximum and minimum
    findMaxMin(arr, 0, size - 1, &max, &min);

    // Output: Maximum and Minimum values
    printf("Maximum value: %d\n", max);
    printf("Minimum value: %d\n", min);

    return 0;
}
```

Sample Input and Output:

Sample Output:

```
Maximum value: 9
Minimum value: 1
```

Explanation:

1. **Divide and Conquer**:
    - The array is divided into smaller subarrays until the base cases are reached.
    - For a single element or two elements, the maximum and minimum values are directly identified.
    - For larger subarrays, recursive calls are made to process each half.
2. **Combining Results**:
    - Once the maximum and minimum values of the subarrays are known, they are combined by comparing the maximum values and minimum values of the two halves.

This method effectively reduces the problem size in each step, leading to an efficient solution with a time complexity of O(n), where n is the number of elements in the array.

Result: Thus, the program has been executed successfully

18.Write a program to generate all the prime numbers.

Aim: To write a ‘C” program to generate all the prime numbers.

Algorithm:

1. **Initialize**: Create a boolean array to keep track of prime status for each number.
2. **Sieve**:
    - Start from the first prime number (2).
    - Mark all multiples of each prime number as non-prime.
    - Move to the next number that is still marked as prime.
3. **Collect Results**: Numbers that are still marked as prime are the prime numbers.

C Code:

Here's a C program to generate all prime numbers up to a given number using the Sieve of Eratosthenes:

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

// Function to generate prime numbers using Sieve of Eratosthenes
void sieveOfEratosthenes(int n) {
    // Create a boolean array and initialize all entries as true
    bool prime[n + 1];
    for (int i = 0; i <= n; i++) {
        prime[i] = true;
    }

    // 0 and 1 are not prime numbers
```

```
    prime[0] = prime[1] = false;

    for (int p = 2; p * p <= n; p++) {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true) {
            // Update all multiples of p
            for (int i = p * p; i <= n; i += p) {
                prime[i] = false;
            }
        }
    }

    // Print all prime numbers
    printf("Prime numbers up to %d:\n", n);
    for (int p = 2; p <= n; p++) {
        if (prime[p]) {
            printf("%d ", p);
        }
    }
    printf("\n");
}

int main() {
    int n;

    // Input: Upper limit to generate prime numbers
    printf("Enter the upper limit to generate prime numbers: ");
    scanf("%d", &n);

    // Generate and print prime numbers up to n
    sieveOfEratosthenes(n);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the upper limit to generate prime numbers: 30
```

Sample Output:

```
Prime numbers up to 30:
2 3 5 7 11 13 17 19 23 29
```

Explanation:

1. **Initialization**:
    - A boolean array `prime` is initialized to `true`. Each index represents whether the number is prime or not.
2. **Sieve Process**:
    - For each number starting from 2, if it is still marked as prime, mark all its multiples as non-prime.
3. **Output**:

- After marking non-prime numbers, iterate through the boolean array to print all indices that are still marked as true.

The Sieve of Eratosthenes is efficient with a time complexity of O(nlog⁡log⁡n)O(n \log \log n)O(nloglogn) and space complexity of O(n)O(n)O(n), making it suitable for generating prime numbers up to large values.

Result: Thus, the program has been executed successfully.

19. Write a program to perform Knapsack problem using greedy techniques.

Aim: To write a 'C' program to perform Knapsack problem using greedy techniques

Algorithm:

1. **Calculate Value-to-Weight Ratio**: For each item, calculate the ratio of value to weight.
2. **Sort Items**: Sort the items based on this ratio in descending order.
3. **Pick Items**: Select items based on the sorted order. If an item can fit entirely into the knapsack, add its full value; if not, add a fraction of its value.

C Code:

Here's a C program that demonstrates the greedy solution to the Fractional Knapsack problem:

```
#include <stdio.h>

// Structure to represent an item
typedef struct {
    int value;
    int weight;
    float ratio;
} Item;

// Function to compare items based on their value-to-weight ratio
int compare(const void *a, const void *b) {
    Item *item1 = (Item *)a;
    Item *item2 = (Item *)b;
    return (item2->ratio > item1->ratio) - (item2->ratio < item1->ratio);
}

// Function to perform the fractional knapsack problem
float knapsack(Item items[], int n, int capacity) {
    // Sort items by value-to-weight ratio
    qsort(items, n, sizeof(Item), compare);

    int currentWeight = 0;
    float totalValue = 0.0;

    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            // Take the whole item
            currentWeight += items[i].weight;
            totalValue += items[i].value;
```

```
        } else {
            // Take the fractional part of the item
            int remaining = capacity - currentWeight;
            totalValue += items[i].value * ((float)remaining /
items[i].weight);
            break;
        }
    }
    return totalValue;
}

int main() {
    int n, capacity;

    // Input: Number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    Item items[n];

    // Input: Value, Weight, and Capacity
    printf("Enter the value and weight for each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d - Value: ", i + 1);
        scanf("%d", &items[i].value);
        printf("Item %d - Weight: ", i + 1);
        scanf("%d", &items[i].weight);
        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    // Compute the maximum value that can be carried
    float maxValue = knapsack(items, n, capacity);
    printf("Maximum value in the knapsack: %.2f\n", maxValue);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of items: 3
Item 1 - Value: 60
Item 1 - Weight: 10
Item 2 - Value: 100
Item 2 - Weight: 20
Item 3 - Value: 120
Item 3 - Weight: 30
Enter the capacity of the knapsack: 50
```

Sample Output:

```
Maximum value in the knapsack: 240.00
```

Explanation:

1. **Item Structure**: Each item is represented with its value, weight, and value-to-weight ratio.
2. **Sorting**: Items are sorted in descending order of their value-to-weight ratio.
3. **Greedy Selection**: The knapsack is filled with items based on their sorted order. If the full item cannot be added, a fractional part is added.

This approach works efficiently for the fractional knapsack problem and ensures an optimal solution with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn) due to sorting.

Result: Thus, the program has been executed successfully.

20.Write a program to perform MST using greedy techniques.

Aim: To write a ‘C’ program to perform MST using greedy techniques

Algorithm:

Kruskal's Algorithm:

Kruskal's algorithm is based on sorting all edges and adding them to the MST in increasing order of weight, while avoiding cycles.

Prim's Algorithm:

Prim's algorithm starts from a selected vertex and grows the MST by adding the smallest edge connecting the MST to a new vertex.

C Code for Kruskal's Algorithm

Here’s a C program to perform Kruskal’s algorithm for finding the MST:

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

#define MAX 100

// Structure to represent an edge
typedef struct {
    int src, dest, weight;
} Edge;

// Structure to represent a subset for Union-Find
typedef struct {
    int parent, rank;
} Subset;

// Function to compare two edges based on their weight
int compareEdges(const void *a, const void *b) {
    Edge *edge1 = (Edge *)a;
    Edge *edge2 = (Edge *)b;
    return edge1->weight - edge2->weight;
}
```

```
// Function to find the parent of a node using path compression
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

// Function to perform union of two subsets
void unionSubsets(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

}

// Function to perform Kruskal's algorithm
void kruskal(int vertices, Edge edges[], int e) {
    Edge result[MAX];
    Subset subsets[vertices];

    // Sort edges by weight
    qsort(edges, e, sizeof(Edge), compareEdges);

    // Initialize subsets
    for (int i = 0; i < vertices; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    int edgeIndex = 0; // Index used to store result edges
    int i = 0; // Initial index of sorted edges

    while (edgeIndex < vertices - 1 && i < e) {
        Edge nextEdge = edges[i++];

        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);

        // If including this edge does not form a cycle
        if (x != y) {
            result[edgeIndex++] = nextEdge;
            unionSubsets(subsets, x, y);
        }
    }

    // Print the resulting MST
    printf("Edges in the Minimum Spanning Tree:\n");
    int minimumCost = 0;
    for (int i = 0; i < edgeIndex; i++) {
```

```
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
result[i].weight);
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost: %d\n", minimumCost);
}

int main() {
    int vertices, edgesCount;

    // Input: Number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edgesCount);

    Edge edges[edgesCount];

    // Input: Edge details (source, destination, weight)
    printf("Enter the edges (source, destination, weight):\n");
    for (int i = 0; i < edgesCount; i++) {
        printf("Edge %d - Source: ", i + 1);
        scanf("%d", &edges[i].src);
        printf("Edge %d - Destination: ", i + 1);
        scanf("%d", &edges[i].dest);
        printf("Edge %d - Weight: ", i + 1);
        scanf("%d", &edges[i].weight);
    }

    // Perform Kruskal's algorithm
    kruskal(vertices, edges, edgesCount);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges (source, destination, weight):
Edge 1 - Source: 0
Edge 1 - Destination: 1
Edge 1 - Weight: 10
Edge 2 - Source: 0
Edge 2 - Destination: 2
Edge 2 - Weight: 6
Edge 3 - Source: 0
Edge 3 - Destination: 3
Edge 3 - Weight: 5
Edge 4 - Source: 1
Edge 4 - Destination: 3
Edge 4 - Weight: 15
Edge 5 - Source: 2
Edge 5 - Destination: 3
Edge 5 - Weight: 4
```

Sample Output:

```
Edges in the Minimum Spanning Tree:
0 -- 3 == 5
2 -- 3 == 4
0 -- 1 == 10
Minimum Cost: 19
```

Explanation:

1. **Edge Sorting**: The edges are sorted by their weight.
2. **Union-Find**: The Union-Find data structure is used to detect cycles and manage disjoint sets.
3. **Edge Selection**: Edges are added to the MST if they don't form a cycle.
4. **Result**: The MST and its cost are printed.

This implementation of Kruskal's algorithm has a time complexity of O(Elog⁡E)O(E \log E)O(ElogE), where EEE is the number of edges, due to the sorting step and union-find operations.

Result: Thus, the program has been executed successfully.

21.Using Dynamic programming concept to find out Optimal binary search tree.

Aim: To write a 'C' program to find out Optimal binary search tree using Dynamic programming concept

Problem Definition:

Given a set of `n` keys and their search probabilities, the goal is to construct a binary search tree such that the expected search cost is minimized. The expected search cost is calculated based on the depth of the nodes and their search probabilities.

Algortithm:

1. **Define the Problem:**
   - Let `p[i]` be the probability of accessing key `i`.
   - Let `w[i][j]` be the sum of probabilities from `p[i]` to `p[j]`.
   - Let `e[i][j]` be the minimum expected search cost for the subarray from key `i` to `j`.
   - Let `root[i][j]` be the root of the optimal BST for the subarray from key `i` to `j`.
2. **Recurrence Relations:**
   - For each subarray from `i` to `j`, try every key `k` as the root and compute the cost: e[i][j]=min⁡i≤k≤j(e[i][k−1]+e[k+1][j]+w[i][j])e[i][j] = \min_{i \leq k \leq j} \left(e[i][k-1] + e[k+1][j] + w[i][j]\right)e[i][j]=i≤k≤jmin(e[i][k−1]+e[k+1][j]+w[i][j])
   - Here, `w[i][j]` is the sum of probabilities `p[i]` to `p[j]`.
3. **Initialization:**
   - When `i == j`, the cost is just `p[i]` because there's only one node in the subarray.
4. **Construct the Table:**

- Fill in tables `e[i][j]`, `root[i][j]`, and `w[i][j]` using the above recurrence relations.

C Code:

Here's a C program to compute the Optimal Binary Search Tree:

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

// Function to find the cost of the optimal binary search tree
void optimalBST(float p[], int n) {
    // Tables for storing minimum cost, root and weight
    float e[n][n], w[n][n];
    int root[n][n];

    // Initialize tables
    for (int i = 0; i < n; i++) {
        e[i][i] = p[i];
        w[i][i] = p[i];
        root[i][i] = i;
    }

    // Fill tables for chains of increasing length
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            e[i][j] = INT_MAX;
            w[i][j] = w[i][j - 1] + p[j];

            // Try different roots and find the minimum cost
            for (int k = i; k <= j; k++) {
                float t = (k > i ? e[i][k - 1] : 0) + (k < j ? e[k + 1][j]
: 0) + w[i][j];
                if (t < e[i][j]) {
                    e[i][j] = t;
                    root[i][j] = k;
                }
            }
        }
    }

    // Print the minimum cost
    printf("Minimum cost of the optimal BST: %.2f\n", e[0][n - 1]);

    // Print the root table
    printf("Root table:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%2d ", root[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n;

    // Input: Number of keys
```

```
    printf("Enter the number of keys: ");
    scanf("%d", &n);

    float p[n];

    // Input: Probabilities of the keys
    printf("Enter the probabilities of the keys:\n");
    for (int i = 0; i < n; i++) {
        printf("Probability of key %d: ", i + 1);
        scanf("%f", &p[i]);
    }

    // Compute the optimal BST
    optimalBST(p, n);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of keys: 4
Enter the probabilities of the keys:
Probability of key 1: 0.15
Probability of key 2: 0.10
Probability of key 3: 0.05
Probability of key 4: 0.20
```

Sample Output:

```
Minimum cost of the optimal BST: 1.10
Root table:
 0  1  2  1
 0  1  2  2
 0  0  2  3
 0  0  0  3
```

Explanation:

1. **Initialization**:
   - The diagonal elements of `e` represent the cost of trees with a single node.
   - `w[i][j]` accumulates probabilities for subarrays.
2. **Dynamic Programming**:
   - For each subarray, calculate the minimum cost by trying each key as the root and choosing the minimum cost.
3. **Result**:
   - `e[0][n-1]` contains the minimum cost for the entire set of keys.
   - The `root` table shows the root of the optimal BST for subarrays.

This approach ensures an efficient solution with a time complexity of O(n3)O(n^3)O(n3), where nnn is the number of keys.

Result: Thus, the program has been executed successfully.

22.Using Dynamic programming techniques to find binomial coefficient of a given number

Aim: To write a 'C' program to find binomial coefficient of a given number using Dynamic programming concept

Algorithm:

1. **Define the Table:**
   - Use a 2D array `C` where `C[i][j]` will store the value of C(i,j)C(i, j)C(i,j).
2. **Initialization:**
   - Set `C[i][0] = 1` for all iii (choosing 0 items from iii items).
   - Set `C[i][i] = 1` for all iii (choosing all iii items from iii items).
3. **Fill the Table:**
   - Use the recursive relation to fill in the rest of the table: C[i][j]=C[i−1][j−1]+C[i−1][j]C[i][j] = C[i-1][j-1] + C[i-1][j]C[i][j]=C[i−1][j−1]+C[i−1][j]

C Code:

Here is a C program to compute the binomial coefficient using dynamic programming:

```
#include <stdio.h>

// Function to compute binomial coefficient C(n, k)
int binomialCoefficient(int n, int k) {
    // Create a table to store binomial coefficients
    int C[n + 1][k + 1];

    // Initialize the table
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= (i < k ? i : k); j++) {
            if (j == 0 || j == i) {
                C[i][j] = 1;  // Base cases
            } else {
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];  // Recursive
relation
            }
        }
    }

    return C[n][k];
}

int main() {
    int n, k;

    // Input: values for n and k
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("Enter the value of k: ");
    scanf("%d", &k);

    // Compute and print the binomial coefficient
    if (k > n || k < 0) {
        printf("Invalid values for n and k.\n");
```

```
    } else {
        printf("C(%d, %d) = %d\n", n, k, binomialCoefficient(n, k));
    }

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the value of n: 5
Enter the value of k: 2
```

Sample Output:

```
C(5, 2) = 10
```

Explanation:

1. **Initialization:**
   - The diagonal of the table and the first column are set to 1, as these represent the base cases.
2. **Filling the Table:**
   - The table is filled using the recursive relation, ensuring that each entry is computed based on previously computed values.
3. **Result:**
   - `C[n][k]` gives the binomial coefficient for the given n and k

This approach has a time complexity of O(n×k)O(n \times k)O(n×k) and a space complexity of O(n×k)O(n \times k)O(n×k), making it efficient for computing binomial coefficients for moderate values of n and k.

Result: Thus, the program has been executed successfully.

23.Write a program to find the reverse of a given number.

Aim: To write a ‘C’ program to find the reverse of a given number.

Algorithm:

1. **Extract Digits**: Repeatedly extract the last digit of the number using the modulus operator.
2. **Construct Reversed Number**: Construct the reversed number by appending the extracted digit to the reversed number so far.
3. **Update the Original Number**: Remove the last digit from the original number by performing integer division by 10.

C Code:

Here's a C program to reverse a given number:

```
#include <stdio.h>

// Function to reverse the digits of a number
int reverseNumber(int num) {
    int reversed = 0;
    while (num != 0) {
        int digit = num % 10;         // Extract the last digit
        reversed = reversed * 10 + digit;  // Append the digit to the
reversed number
        num /= 10;                   // Remove the last digit from the
original number
    }
    return reversed;
}

int main() {
    int number;

    // Input: the number to reverse
    printf("Enter a number: ");
    scanf("%d", &number);

    // Reverse the number and print the result
    int reversedNumber = reverseNumber(number);
    printf("Reversed number: %d\n", reversedNumber);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter a number: 12345
```

Sample Output:

```
Reversed number: 54321
```

Explanation:

1. **Extracting the Last Digit:**
   - The last digit is obtained using `num % 10`.
2. **Constructing the Reversed Number:**
   - Multiply the current reversed number by 10 and add the extracted digit to it.
3. **Updating the Original Number:**
   - Perform integer division by 10 to remove the last digit.

Result: Thus, the program has been executed successfully.

24.Write a program to find the perfect number.
Aim: To write a 'C' program to find the perfect number

Algprithm:

1. **Find Proper Divisors**: Iterate through numbers from 1 to half of the given number to find all its divisors.
2. **Sum Divisors**: Compute the sum of these divisors.
3. **Check Perfection**: Compare the sum of the divisors to the original number.

C Code:

Here is a C program to check if a given number is a perfect number:

```
#include <stdio.h>

// Function to check if a number is a perfect number
int isPerfectNumber(int num) {
    if (num <= 1) return 0;  // 0 and negative numbers are not perfect
numbers

    int sum = 0;
    // Find divisors and sum them up
    for (int i = 1; i <= num / 2; i++) {
        if (num % i == 0) {
            sum += i;
        }
    }

    return sum == num;  // Check if sum of divisors equals the original
number
}

int main() {
    int number;

    // Input: the number to check
    printf("Enter a number: ");
    scanf("%d", &number);

    // Check if the number is a perfect number and print the result
    if (isPerfectNumber(number)) {
        printf("%d is a perfect number.\n", number);
    } else {
        printf("%d is not a perfect number.\n", number);
    }

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter a number: 28
```

Sample Output:

```
28 is a perfect number.
```

Explanation:

1. **Finding Proper Divisors:**
   - Iterate from 1 to num2\frac{num}{2}2num (since any divisor greater than num2\frac{num}{2}2num cannot be a proper divisor).
   - Check if each number divides `num` evenly.
2. **Summing Divisors:**
   - Add the divisors to a running total.
3. **Checking Perfection:**
   - If the sum of the divisors equals the original number, then the number is perfect.

Result: Thus, the program has been executed successfully.

25.Write a program to perform travelling salesman problem using dynamic programming

Aim: To write a 'C' program to perform travelling salesman problem using dynamic programming

Algorithm:

1. **State Representation:**
   - Use a bitmask to represent the set of cities visited so far.
   - Use a 2D array `dp` where `dp[mask][i]` represents the minimum cost to visit all cities in the subset `mask` and end at city `i`.
2. **Initialization:**
   - Start with the base case where only the starting city is visited.
3. **Transition:**
   - For each subset of cities represented by `mask`, and for each city `i` in that subset, try to move to another city `j` not in the subset and update the cost accordingly.
4. **Result Extraction:**
   - The result is found by looking at the minimum cost to visit all cities and return to the starting city.

C Code:

Here's a C program that solves the TSP using dynamic programming:

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

#define MAX 16
#define INF INT_MAX

// Function to find the minimum cost of visiting all cities and returning
to the start
int tsp(int n, int dist[MAX][MAX]) {
    int dp[1 << MAX][MAX];

    // Initialize DP table with infinity
```

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

    // Starting from the first city
    dp[1][0] = 0;

    // Iterate over all subsets of cities
    for (int mask = 1; mask < (1 << n); mask += 2) {
        for (int u = 0; u < n; u++) {
            if (!(mask & (1 << u))) continue;
            for (int v = 0; v < n; v++) {
                if (mask & (1 << v)) continue;
                int newMask = mask | (1 << v);
                dp[newMask][v] = (dp[newMask][v] < dp[mask][u] +
dist[u][v]) ? dp[newMask][v] : dp[mask][u] + dist[u][v];
            }
        }
    }

    // Calculate the minimum cost to return to the starting city
    int answer = INF;
    for (int i = 1; i < n; i++) {
        answer = (answer < dp[(1 << n) - 1][i] + dist[i][0]) ? answer :
dp[(1 << n) - 1][i] + dist[i][0];
    }

    return answer;
}

int main() {
    int n;

    // Input: number of cities
    printf("Enter the number of cities: ");
    scanf("%d", &n);

    int dist[MAX][MAX];

    // Input: distance matrix
    printf("Enter the distance matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &dist[i][j]);
        }
    }

    // Compute and print the minimum cost
    int result = tsp(n, dist);
    printf("The minimum cost of the TSP is: %d\n", result);

    return 0;
}
```

Sample Input and Output:

Sample Input:

```
Enter the number of cities: 4
Enter the distance matrix:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
```

Sample Output:

```
The minimum cost of the TSP is: 80
```

Explanation:

1. **Initialization:**
   - Initialize the `dp` array with infinity, except for the starting city with a cost of 0.
2. **Dynamic Programming Transition:**
   - Iterate over all subsets of cities and update the `dp` array based on moving from one city to another.
3. **Result Extraction:**
   - Compute the minimum cost to complete the tour and return to the starting city.

Result: Thus, the program has been executed successfully.

26.Write a program for the given pattern If n=4

```
   1
  1 2
 1 2 3
1 2 3 4
```

Aim: To generate the given pattern where `n = 4`, you need to print a pattern of numbers in a right-aligned triangle format.

Algorithm

1. **Input the Value of `n`:**
   - Read the integer `n` which represents the number of rows in the pattern.
2. **Loop Through Each Row:**
   - For each row from `1` to `n`:
     1. **Print Leading Spaces:**
        - Calculate the number of leading spaces required to align the numbers properly.
        - Print these spaces.
     2. **Print Numbers:**
        - For each row, print numbers starting from `1` up to the row number.
3. **Move to the Next Line:**
   - After printing all numbers in the current row, move to the next line.

C Code:

Here's a C program to generate this pattern:

```
#include <stdio.h>

// Function to print the pattern
void printPattern(int n) {
    for (int i = 1; i <= n; i++) {
        // Print leading spaces
        for (int j = 0; j < n - i; j++) {
            printf(" ");
        }
        // Print numbers in the current row
        for (int k = 1; k <= i; k++) {
            printf("%d ", k);
        }
        printf("\n"); // Move to the next line
    }
}

int main() {
    int n;

    // Input: number of rows
    printf("Enter the number of rows (n): ");
    scanf("%d", &n);

    // Print the pattern
    printPattern(n);

    return 0;
}
```

Explanation:

1. **Leading Spaces:**
    - For each row `i`, print `n - i` spaces to align the numbers to the right.
2. **Print Numbers:**
    - Print numbers from 1 to `i` for the current row.
3. **Move to the Next Line:**
    - After printing numbers for the current row, print a newline character to move to the next row.

Sample Input and Output:

Sample Input:

```
Enter the number of rows (n): 4
```

Sample Output:

```
   1
  1 2
 1 2 3
1 2 3 4
```

Result: Thus, the program has been executed successfully.

27.Write a program to perform Floyd's algorithm

Aim: To write a 'C' program to perform Floyd's algorithm

Algorithm:

1. **Initialization:**
   - Create a distance matrix `dist` where `dist[i][j]` represents the shortest distance from vertex `i` to vertex `j`. Initialize `dist[i][j]` to the weight of the edge between `i` and `j` if there is an edge; otherwise, initialize it to infinity (or a large number).
2. **Algorithm:**
   - Update the distance matrix by considering whether a path through an intermediate vertex `k` provides a shorter path between any two vertices `i` and `j`.
3. **Output:**
   - After completing the algorithm, `dist[i][j]` will hold the shortest distance from vertex `i` to vertex `j`.

C Code:

Here is a C program to perform Floyd-Warshall algorithm:

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

#define MAX 100
#define INF INT_MAX

// Function to perform Floyd-Warshall algorithm
void floydWarshall(int graph[MAX][MAX], int n) {
    int dist[MAX][MAX];

    // Initialize distance matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                dist[i][j] = 0;
            } else if (graph[i][j] != 0) {
                dist[i][j] = graph[i][j];
            } else {
                dist[i][j] = INF;
            }
        }
    }

    // Floyd-Warshall algorithm
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
```

```
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][j] >
dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the distance matrix
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][j] == INF) {
                printf("INF\t");
            } else {
                printf("%d\t", dist[i][j]);
            }
        }
        printf("\n");
    }
}

int main() {
    int n;

    // Input: number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    int graph[MAX][MAX];

    // Input: adjacency matrix
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
            if (i != j && graph[i][j] == 0) {
                graph[i][j] = INF; // Treat zero as infinity for non-
diagonal elements
            }
        }
    }

    // Perform Floyd-Warshall algorithm
    floydWarshall(graph, n);

    return 0;
}
```

Explanation:

1. **Initialization:**
   - Initialize the `dist` matrix based on the input graph. Set distances between different vertices to infinity if no edge exists between them.
2. **Floyd-Warshall Algorithm:**
   - For each pair of vertices `(i, j)`, check if a path through vertex `k` offers a shorter path than the current known shortest path.
3. **Print Results:**

- Output the shortest distance matrix where each entry shows the shortest path distance between vertex `i` and vertex `j`. Use "INF" to represent infinite distances.

Sample Input and Output:

Sample Input:

```
Enter the number of vertices: 4
Enter the adjacency matrix:
0 3 0 7
8 0 2 0
5 0 0 1
2 0 0 0
```

Sample Output:

```
Shortest distances between every pair of vertices:
0       3       5       6
8       0       2       3
5       8       0       1
2       5       7       0
```

Result: Thus, the program has been executed successfully.

28.Write a program for pascal triangle.

Aim: To write a 'C' program to form pascal triangle

Algorithm

1. **Input the Number of Rows (`n`):**
   - Read the integer `n` which represents the number of rows in Pascal's Triangle.
2. **Initialize the Triangle:**
   - Create a 2D array or list to store the values of Pascal's Triangle.
3. **Fill the Triangle:**
   - **First Row:** Set the first element of the first row to `1`.
   - **Subsequent Rows:**
     - For each row `i` from `1` to `n-1`:
       - Set the first and last elements of the row to `1`.
       - For each element `j` in between (i.e., `1` to `i-1`):
         - Calculate the value using the sum of the two elements directly above it (i.e., `triangle[i-1][j-1] + triangle[i-1][j]`).
4. **Print the Triangle:**
   - Print each row of the triangle in the correct format, usually centered.

C Code:

Here is a C program to generate Pascal's Triangle for a given number of rows:

```c
#include <stdio.h>

// Function to generate Pascal's Triangle
void printPascalsTriangle(int n) {
    int triangle[n][n]; // Declare a 2D array to store Pascal's Triangle

    // Initialize the Pascal's Triangle
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            if (j == 0 || j == i) {
                triangle[i][j] = 1; // First and last element of each row
are 1
            } else {
                triangle[i][j] = triangle[i - 1][j - 1] + triangle[i -
1][j]; // Sum of two elements above
            }
        }
    }

    // Print Pascal's Triangle
    for (int i = 0; i < n; i++) {
        // Print leading spaces for alignment
        for (int j = 0; j < n - i - 1; j++) {
            printf(" ");
        }
        // Print elements of the current row
        for (int j = 0; j <= i; j++) {
            printf("%d ", triangle[i][j]);
        }
        printf("\n"); // Move to the next line
    }
}

int main() {
    int n;

    // Input: number of rows
    printf("Enter the number of rows for Pascal's Triangle: ");
    scanf("%d", &n);

    // Print Pascal's Triangle
    printPascalsTriangle(n);

    return 0;
}
```

Explanation:

1. **Initialization:**
    - Initialize a 2D array `triangle` where `triangle[i][j]` stores the value at row `i` and column `j`.
2. **Generating the Triangle:**
    - The first and last elements of each row are set to `1`.
    - Interior elements are computed as the sum of the two elements directly above from the previous row.
3. **Printing the Triangle:**
    - Print leading spaces to align the numbers correctly.
    - Print each number in the row and move to the next line after each row.

Sample Input and Output:

Sample Input:

```
Enter the number of rows for Pascal's Triangle: 5
```

Sample Output:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Result: Thus, the program has been executed successfully.

29.Write a program to find the optimal cost by using appropriate algorithm

Aim: To write a 'C' program to find the optimal cost by using appropriate algorithm

Knapsack Problem:

The goal is to maximize the total value of items placed in a knapsack without exceeding its weight capacity.

Problem Definition:

- You have a set of items, each with a weight and a value.
- You have a knapsack with a maximum weight capacity.
- Determine the maximum value that can be carried in the knapsack.

Algorithm

1. **Define the State:**
   - Use a 2D array `dp` where `dp[i][w]` represents the maximum value that can be obtained using the first `i` items and with a knapsack capacity of `w`.
2. **Initialization:**
   - Initialize `dp[i][0] = 0` for all `i` because if the capacity is `0`, the maximum value is `0`.
3. **Transition:**
   - For each item and each capacity, decide whether to include the item or not.
4. **Extract Result:**
   - The result is in `dp[n][W]` where `n` is the number of items and `W` is the maximum weight capacity.

C Code:

Here is a C program to solve the Knapsack Problem using dynamic programming:

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

```
// Function to find the maximum value in the knapsack
int knapsack(int W, int weights[], int values[], int n) {
    int dp[n + 1][W + 1];

    // Initialize the dp array
    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 (weights[i - 1] <= w) {
                dp[i][w] = (values[i - 1] + dp[i - 1][w - weights[i - 1]] >
dp[i - 1][w]) ?
                           (values[i - 1] + dp[i - 1][w - weights[i - 1]])
:
                           dp[i - 1][w];
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    // The maximum value that can be carried is in dp[n][W]
    return dp[n][W];
}

int main() {
    int n, W;

    // Input: number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    int weights[n], values[n];

    // Input: weights and values of items
    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights[i]);
    }

    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    // Input: maximum weight capacity
    printf("Enter the maximum weight capacity of the knapsack: ");
    scanf("%d", &W);

    // Compute and print the maximum value
    int result = knapsack(W, weights, values, n);
    printf("The maximum value that can be carried is: %d\n", result);

    return 0;
}
```

Explanation:

1. **Initialization:**

    - Initialize the `dp` array where `dp[i][w]` is set to `0` if either `i` or `w` is `0`.
2. **Dynamic Programming Transition:**
    - For each item and weight capacity, decide whether to include the current item based on its weight and value.
3. **Result Extraction:**
    - The maximum value obtainable with the given capacity is stored in `dp[n][W]`.

Sample Input and Output:

Sample Input:

```
Enter the number of items: 4
Enter the weights of the items:
2 3 4 5
Enter the values of the items:
3 4 5 6
Enter the maximum weight capacity of the knapsack: 5
```

Sample Output:

```
The maximum value that can be carried is: 7
```

Result: Thus, the program has been executed successfully.

30. Write a program to find the sum of digits.

Aim: To write a ‘C’ program to find the sum of digits

C Code:

Here's a C program to find the sum of digits of a given number:

```
#include <stdio.h>

// Function to calculate the sum of digits of a number
int sumOfDigits(int num) {
    int sum = 0;
    while (num != 0) {
        sum += num % 10; // Add the last digit to sum
        num /= 10;       // Remove the last digit from the number
    }
    return sum;
}

int main() {
    int number;

    // Input: the number to find the sum of digits
    printf("Enter a number: ");
    scanf("%d", &number);

    // Handle negative numbers
    if (number < 0) {
```

```
        number = -number; // Convert to positive if the number is negative
    }

    // Compute and print the sum of digits
    int result = sumOfDigits(number);
    printf("The sum of digits is: %d\n", result);

    return 0;
}
```

Explanation:

1. **Extracting and Summing Digits:**
   - Use `num % 10` to get the last digit of the number.
   - Add this digit to the `sum`.
   - Update `num` by dividing it by `10` to remove the last digit.
2. **Handling Negative Numbers:**
   - Convert the number to its positive equivalent if it is negative, as the sum of digits is typically computed for non-negative integers.

Sample Input and Output:

Sample Input:

```
Enter a number: 1234
```

Sample Output:

```
The sum of digits is: 10
```

Result: Thus, the program has been executed successfully.

31.Write a program to print a minimum and maximum value sequency for all the numbers in a list.

Aim: To write a 'C' program to print a maximum and minimum value sequence for all the numbers in a list

Algorithm

1. **Input the List:**
   - Read or initialize the list of numbers.
2. **Initialize Variables:**
   - Set `min_val` to a very large number (e.g., `INT_MAX`).
   - Set `max_val` to a very small number (e.g., `INT_MIN`).
3. **Iterate Through the List:**
   - For each number in the list:
     - Update `min_val` if the current number is smaller.
     - Update `max_val` if the current number is larger.
4. **Print Results:**

- Print the minimum and maximum values found.

C Code:

Here's a C program that finds and prints the minimum and maximum values in a list of numbers:

```
#include <stdio.h>

// Function to find the minimum and maximum values in the list
void findMinMax(int arr[], int size, int *min, int *max) {
    *min = arr[0];
    *max = arr[0];

    for (int i = 1; i < size; i++) {
        if (arr[i] < *min) {
            *min = arr[i];
        }
        if (arr[i] > *max) {
            *max = arr[i];
        }
    }
}

int main() {
    int n;

    // Input: number of elements in the list
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Input: elements of the list
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int min, max;

    // Find the minimum and maximum values
    findMinMax(arr, n, &min, &max);

    // Print the results
    printf("Minimum value: %d\n", min);
    printf("Maximum value: %d\n", max);

    return 0;
}
```

Explanation:

1. **Function `findMinMax`:**
    - This function takes an array, its size, and pointers to `min` and `max` values.
    - Initializes `min` and `max` to the first element of the array.
    - Iterates through the array to find the minimum and maximum values.

2. **Main Function:**
    - Reads the number of elements and the elements of the list.
    - Calls `findMinMax` to determine the minimum and maximum values.
    - Prints the results.

Sample Input and Output:

Sample Input:

```
Enter the number of elements: 5
Enter the elements:
3 1 4 1 5
```

Sample Output:

```
Minimum value: 1
Maximum value: 5
```

Result: Thus, the program has been executed successfully.

32.Write a program to perform n Queens problem using backtracking.

Aim: To write a 'C' program to perform n queens using backtracking

Algorithm

1. **Initialize the Board:**
    - Create a 2D array or list to represent the chessboard. Initialize all positions to indicate no queens are placed.
2. **Start Placing Queens:**
    - Begin with the first row and attempt to place a queen in each column of that row.
    - For each placement, check if it's valid (i.e., the queen does not threaten any other queen).
3. **Check Valid Placement:**
    - Ensure that placing the queen does not lead to a conflict in the current row, column, or diagonals.
4. **Recursive Call:**
    - If placing the queen is valid, make a recursive call to place queens in the next row.
5. **Backtrack:**
    - If placing the queen leads to a solution, move to the next row. If no solution is found, remove the queen (backtrack) and try the next column in the current row.
6. **Solution Found:**
    - When all queens are placed successfully, record or print the solution.
7. **Terminate:**
    - If all rows are processed, print or store the solution.

C Code:

Here's a C program to solve the N-Queens problem using backtracking:

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

#define MAX 20

// Function to print the solution
void printSolution(int board[MAX][MAX], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf(" %d ", board[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

// Function to check if a queen can be placed on board[row][col]
bool isSafe(int board[MAX][MAX], int row, int col, int N) {
    // Check this column on upper rows
    for (int i = 0; i < row; i++) {
        if (board[i][col]) {
            return false;
        }
    }

    // Check upper left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }

    // Check upper right diagonal
    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
        if (board[i][j]) {
            return false;
        }
    }

    return true;
}

// Function to solve the N-Queens problem using backtracking
bool solveNQueens(int board[MAX][MAX], int row, int N) {
    if (row >= N) {
        return true; // All queens are placed
    }

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

```
    }

    return false; // No solution found
}

int main() {
    int N;
    int board[MAX][MAX] = {0}; // Initialize board to zero

    // Input: number of queens (and size of the board)
    printf("Enter the number of queens (N): ");
    scanf("%d", &N);

    // Solve the N-Queens problem
    if (solveNQueens(board, 0, N)) {
        printf("One possible solution is:\n");
        printSolution(board, N);
    } else {
        printf("No solution exists for N = %d\n", N);
    }

    return 0;
}
```

Explanation:

1. **Function `isSafe`:**
   - Checks if placing a queen at `board[row][col]` is safe from attacks by other queens.
   - Checks the column and both diagonals for conflicts.
2. **Function `solveNQueens`:**
   - Attempts to place queens row by row.
   - Uses recursion to place queens in subsequent rows.
   - Backtracks if placing a queen leads to no solution.
3. **Function `printSolution`:**
   - Prints the board where `1` indicates the presence of a queen and `0` indicates an empty cell.

Sample Input and Output:

Sample Input:

```
Enter the number of queens (N): 4
```

Sample Output:

```
One possible solution is:
 0 0 1 0
 1 0 0 0
 0 0 0 1
 0 1 0 0
```

Result: Thus, the program has been executed successfully.

33. Write a program to inset a number in a list.

Aim: To write a ‘C’ program to insert a number in the list

Algorithm to Insert a Number at a Specific Position in a List

1. **Input the List and the Number:**
   - Read or initialize the list where the number will be inserted.
   - Read or specify the number to be inserted.
   - Read or specify the position where the number should be inserted.
2. **Validate the Position:**
   - Check if the position is within the valid range (i.e., `0 <= position <= length of list`).
3. **Insert the Number:**
   - Shift elements from the specified position to the end of the list one position to the right.
   - Place the new number at the specified position.
4. **Update the List:**
   - Ensure the list has enough space to accommodate the new number if it's a fixed-size array.
5. **Print the Updated List:**
   - Display the list after the insertion.

C Code:

```
#include <stdio.h>

#define MAX 100 // Define the maximum size of the list

// Function to insert a number into the list at a specific position
void insertNumber(int list[], int *size, int number, int position) {
    // Check if the position is valid
    if (position < 0 || position > *size) {
        printf("Invalid position!\n");
        return;
    }

    // Check if the list has enough space
    if (*size >= MAX) {
        printf("List is full!\n");
        return;
    }

    // Shift elements to the right to make space for the new number
    for (int i = *size; i > position; i--) {
        list[i] = list[i - 1];
    }

    // Insert the new number at the specified position
    list[position] = number;

    // Update the size of the list
    (*size)++;
}
```

```
// Function to print the list
void printList(int list[], int size) {
    printf("List elements are:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", list[i]);
    }
    printf("\n");
}

int main() {
    int list[MAX];
    int size = 0; // Current number of elements in the list
    int number, position;

    // Input: number of elements in the list
    printf("Enter the number of initial elements in the list: ");
    scanf("%d", &size);

    // Input: elements of the list
    printf("Enter the elements of the list:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &list[i]);
    }

    // Input: number to insert and the position
    printf("Enter the number to insert: ");
    scanf("%d", &number);
    printf("Enter the position to insert the number at (0-based index): ");
    scanf("%d", &position);

    // Insert the number into the list
    insertNumber(list, &size, number, position);

    // Print the updated list
    printList(list, size);

    return 0;
}
```

Explanation:

1. **Function `insertNumber`:**
    - Checks if the given position is valid and within bounds.
    - Checks if there is space available in the list.
    - Shifts elements to the right starting from the end of the list to the position where the new number will be inserted.
    - Inserts the new number at the specified position.
    - Updates the size of the list.
2. **Function `printList`:**
    - Prints all elements in the list.
3. **Main Function:**
    - Reads the initial elements into the list.
    - Takes input for the number to be inserted and its position.
    - Calls `insertNumber` to perform the insertion.
    - Prints the updated list.

Sample Input and Output:

Sample Input:

```
Enter the number of initial elements in the list: 5
Enter the elements of the list:
10 20 30 40 50
Enter the number to insert: 25
Enter the position to insert the number at (0-based index): 2
```

Sample Output:

```
List elements are:
10 20 25 30 40 50
```

Result: Thus, the program has been executed successfully.

34.Write a program to perform sum of subsets problem using backtracking

Aim: To write a 'C' program to perform sum of subsets using backtracking

Algorithm

1. **Input the Set and Target Sum:**
   - Read or initialize the set of numbers.
   - Specify the target sum.
2. **Initialize Variables:**
   - Create a list to hold the current subset.
   - Use a global list to store all valid subsets.
3. **Recursive Function:**
   - Define a function to recursively explore all subsets.
   - At each step, decide whether to include or exclude the current number.
   - Update the current sum and subset accordingly.
4. **Base Case:**
   - If the current sum matches the target sum, add the current subset to the list of valid subsets.
   - If the current sum exceeds the target, terminate the current branch of recursion.
5. **Backtrack:**
   - Remove the last added element and try other possibilities.
6. **Print Results:**
   - Display all subsets that sum up to the target value..

C Code:

Here's a C program to solve the Sum of Subsets problem using backtracking:

```
#include <stdio.h>

#define MAX 20
```

```
// Function to print a subset
void printSubset(int subset[], int size) {
    printf("{ ");
    for (int i = 0; i < size; i++) {
        printf("%d ", subset[i]);
    }
    printf("}\n");
}

// Recursive function to find subsets that sum up to the target
void findSubsets(int arr[], int n, int index, int target, int currentSum,
int subset[], int subsetSize) {
    // If we have reached the target sum, print the subset
    if (currentSum == target) {
        printSubset(subset, subsetSize);
        return;
    }

    // If we have processed all elements or exceeded the target sum, return
    if (index >= n || currentSum > target) {
        return;
    }

    // Include the current element in the subset
    subset[subsetSize] = arr[index];
    findSubsets(arr, n, index + 1, target, currentSum + arr[index], subset,
subsetSize + 1);

    // Exclude the current element from the subset
    findSubsets(arr, n, index + 1, target, currentSum, subset, subsetSize);
}

int main() {
    int arr[MAX], n, target;
    int subset[MAX];

    // Input: number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Input: elements of the set
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input: target sum
    printf("Enter the target sum: ");
    scanf("%d", &target);

    // Find and print all subsets that sum up to the target
    printf("Subsets that sum up to %d are:\n", target);
    findSubsets(arr, n, 0, target, 0, subset, 0);

    return 0;
}
```

Explanation:

1. **Function `printSubset`:**
    - Prints the elements of the subset in a readable format.
2. **Function `findSubsets`:**
    - Uses backtracking to explore all possible subsets.
    - Includes or excludes the current element and recursively searches for valid subsets.
    - If the current sum matches the target, it prints the subset.
    - Backtracks by removing the last added element and continues searching.
3. **Main Function:**
    - Reads input values, including the number of elements, the elements themselves, and the target sum.
    - Calls `findSubsets` to generate and print all subsets that sum up to the target.

Sample Input and Output:

Sample Input:

```
Enter the number of elements: 5
Enter the elements:
1 2 3 4 5
Enter the target sum: 5
```

Sample Output:

```
Subsets that sum up to 5 are:
{ 1 4 }
{ 2 3 }
{ 5 }
```

Result: Thus, the program has been executed successfully.

35.Write a program to perform graph coloring problem using backtracking.

Aim: To write a ‘C’ program to perform graph coloring using backtracking

Algorithm

1. **Input the Graph:**
    - Read or initialize the adjacency matrix or adjacency list representation of the graph.
    - Specify the number of colors available.
2. **Initialize Variables:**
    - Create a color array to store the color assigned to each vertex.
3. **Recursive Function:**
    - Define a function to assign colors to the vertices using backtracking.
4. **Base Case:**
    - If all vertices are colored, the solution is valid.
5. **Check Valid Color:**
    - For each vertex, check if the color can be assigned without causing a conflict with adjacent vertices.

6. **Backtrack:**
    - If assigning the color leads to a solution, move to the next vertex. If not, try another color.
7. **Print Results:**
    - Display the color assignment if a valid coloring is found.

C Code:

Here's a C program that performs graph coloring using backtracking:

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

#define MAX_VERTICES 100

// Function to check if the current color assignment is safe for vertex `v`
bool isSafe(int graph[MAX_VERTICES][MAX_VERTICES], int color[], int v, int
c, int V) {
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && color[i] == c) {
            return false;
        }
    }
    return true;
}

// Recursive function to solve the graph coloring problem
bool graphColoringUtil(int graph[MAX_VERTICES][MAX_VERTICES], int color[],
int v, int m, int V) {
    // Base case: If all vertices are assigned a color then return true
    if (v == V) {
        return true;
    }

    // Try different colors for vertex `v`
    for (int c = 1; c <= m; c++) {
        if (isSafe(graph, color, v, c, V)) {
            color[v] = c; // Assign color `c` to vertex `v`

            // Recur to assign colors to the rest of the vertices
            if (graphColoringUtil(graph, color, v + 1, m, V)) {
                return true;
            }

            // If assigning color `c` doesn't lead to a solution then
remove it
            color[v] = 0;
        }
    }
    return false;
}

// Function to solve the graph coloring problem
bool graphColoring(int graph[MAX_VERTICES][MAX_VERTICES], int V, int m) {
    int color[V];
    for (int i = 0; i < V; i++) {
        color[i] = 0; // Initialize all vertices with no color
    }
```

```
    // Call the recursive function to solve the problem
    return graphColoringUtil(graph, color, 0, m, V);
}

// Function to print the color assignment
void printSolution(int color[], int V) {
    printf("Solution:\n");
    for (int i = 0; i < V; i++) {
        printf("Vertex %d ---> Color %d\n", i, color[i]);
    }
}

int main() {
    int V, E, m;
    int graph[MAX_VERTICES][MAX_VERTICES] = {0};

    // Input: number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    // Input: number of edges
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    // Input: edges of the graph
    printf("Enter the edges (format: u v):\n");
    for (int i = 0; i < E; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        graph[u][v] = 1;
        graph[v][u] = 1;
    }

    // Input: number of colors
    printf("Enter the number of colors: ");
    scanf("%d", &m);

    // Solve the graph coloring problem
    if (graphColoring(graph, V, m)) {
        printf("Solution exists with %d colors:\n", m);
        printSolution(color, V);
    } else {
        printf("Solution does not exist with %d colors.\n", m);
    }

    return 0;
}
```

Explanation:

1. **Function `isSafe`:**
    - Checks if it is safe to assign a color to a vertex without violating the coloring constraint.
2. **Function `graphColoringUtil`:**
    - Recursively tries to color the graph.
    - For each vertex, attempts to assign each color and checks if it leads to a valid solution.
    - Backtracks if the current color assignment does not work.

3. **Function `graphColoring`:**
    - Initializes the color array and calls the recursive function to start coloring.
4. **Function `printSolution`:**
    - Prints the color assigned to each vertex.
5. **Main Function:**
    - Reads input values for the number of vertices, edges, and colors.
    - Constructs the adjacency matrix of the graph.
    - Calls `graphColoring` to find and print a valid coloring solution if it exists.

Sample Input and Output:

Sample Input:

```
Enter the number of vertices: 4
Enter the number of edges: 4
Enter the edges (format: u v):
0 1
0 2
1 2
1 3
Enter the number of colors: 3
```

Sample Output:

```
Solution exists with 3 colors:
Vertex 0 ---> Color 1
Vertex 1 ---> Color 2
Vertex 2 ---> Color 3
Vertex 3 ---> Color 1
```

Result: Thus, the program has been executed successfully.

36.Write a program to compute container loader Problem.

Aim: To write a ‘C’ program to compute container loader problem

Algorithm

One common heuristic for solving the Container Loader problem is the First-Fit Decreasing (FFD) algorithm. Here's a step-by-step breakdown:

1. **Input the List of Items:**
    - Read or initialize the list of item sizes that need to be packed.
2. **Sort Items:**
    - Sort the items in decreasing order of their sizes.
3. **Initialize Containers:**
    - Create an empty list to represent containers.
4. **Fit Items into Containers:**

- Iterate over each item and try to place it in the first container that has enough remaining capacity.
- If no container has enough space, create a new container and place the item in it.

5. **Print Results:**
   - Display the contents of each container.

C Code:

Here's a C program to solve the Container Loader Problem using the First-Fit Decreasing heuristic:

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

#define MAX_ITEMS 100
#define MAX_BINS 100

// Function to compare items for sorting (used in qsort)
int compare(const void *a, const void *b) {
    return (*(int*)b - *(int*)a); // Sort in descending order
}

// Function to perform the First-Fit Decreasing algorithm
void containerLoader(int items[], int n, int binCapacity) {
    int bins[MAX_BINS];
    int binCount = 0;
    int i, j;

    // Initialize all bins to 0
    for (i = 0; i < MAX_BINS; i++) {
        bins[i] = 0;
    }

    // Sort items in decreasing order
    qsort(items, n, sizeof(int), compare);

    // Place each item into the first bin that can accommodate it
    for (i = 0; i < n; i++) {
        int item = items[i];
        int placed = 0;

        for (j = 0; j < binCount; j++) {
            if (bins[j] + item <= binCapacity) {
                bins[j] += item;
                placed = 1;
                break;
            }
        }

        // If item was not placed in any bin, create a new bin
        if (!placed) {
            bins[binCount] = item;
            binCount++;
        }
    }
```

```
    // Print the results
    printf("Number of bins used: %d\n", binCount);
    for (i = 0; i < binCount; i++) {
        printf("Bin %d: %d\n", i + 1, bins[i]);
    }
}

int main() {
    int items[MAX_ITEMS];
    int n, binCapacity;

    // Input: number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    // Input: items
    printf("Enter the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &items[i]);
    }

    // Input: bin capacity
    printf("Enter the bin capacity: ");
    scanf("%d", &binCapacity);

    // Solve the container loader problem
    containerLoader(items, n, binCapacity);

    return 0;
}
```

Explanation:

1. **Sorting Items:**
    - Items are sorted in decreasing order to try and fit larger items first. This helps in reducing the number of bins used.
2. **First-Fit Decreasing (FFD) Algorithm:**
    - For each item, the algorithm tries to place it in the first bin that has enough remaining capacity.
    - If no such bin exists, a new bin is created.
3. **Print Results:**
    - The number of bins used and their contents are printed.

Sample Input and Output:

Sample Input:

```
Enter the number of items: 7
Enter the items:
5 3 8 6 2 7 4
Enter the bin capacity: 10
```

Sample Output:

```
Number of bins used: 4
```

```
Bin 1: 8 2
Bin 2: 7 3
Bin 3: 6 4
Bin 4: 5
```

Result: Thus, the program has been executed successfully.

37.Write a program to generate the list of all factor for n value.

- **Aim:** To write a 'C' program to generate the list of all factor for n value.
- **Algorithm:**
    1. Read the integer `n` from the user.
    2. Iterate from `1` to `n`.
    3. For each number `i` in this range, check if `i` is a factor of `n` (i.e., `n % i == 0`).
    4. Print each factor.
- **Input:**
    - An integer `n`.
- **Method:**
    0. Read `n` from the user.
    1. Use a loop to check all numbers from `1` to `n` to determine if they are factors of `n`.
    2. Print the factors.

C Code

```
#include <stdio.h>

// Function to print all factors of a number
void printFactors(int n) {
    printf("Factors of %d are:\n", n);

    // Loop through all numbers from 1 to n
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) { // If i is a factor of n
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main() {
    int n;

    // Input: number to find factors for
    printf("Enter a number: ");
    scanf("%d", &n);

    // Print all factors of the number
    printFactors(n);

    return 0;
}
```

Sample Input and Output

Sample Input:

```
Enter a number: 36
```

Sample Output:

```
Factors of 36 are:
1 2 3 4 6 9 12 18 36
```

Explanation:

1. **Function `printFactors`:**
   - This function takes an integer `n` as input and iterates from `1` to `n`.
   - For each `i`, it checks if `i` is a divisor of `n` (i.e., `n % i == 0`).
   - If it is, `i` is printed as a factor.
2. **Main Function:**
   - Prompts the user to enter a number.
   - Calls `printFactors` to display all factors of the number provided.

Result: Thus, the program has been executed successfully.

38.Write a program to perform Assignment problem using branch and bound

Aim: To write a 'C' program to perform assignment problem using branch and bound

Branch and Bound Algorithm for Assignment Problem

1. **Input:**
   - A cost matrix where `cost[i][j]` represents the cost of assigning task `i` to agent `j`.
2. **Initialization:**
   - Set up a priority queue or list to manage nodes (partial assignments).
   - Initialize the best cost to infinity.
3. **Branching:**
   - Create branches by assigning tasks to agents and expanding the search tree.
   - For each branch, calculate a lower bound to estimate the best possible cost from that branch.
4. **Bounding:**
   - Use a lower bound (often calculated by relaxing the problem to an assignment problem solved via techniques like the Hungarian algorithm) to prune branches that cannot lead to a better solution than the current best.
5. **Pruning:**
   - If a branch's lower bound is greater than or equal to the best known cost, prune (discard) that branch.
6. **Termination:**
   - Continue until all nodes have been processed or pruned.
   - The best cost and its corresponding assignment will be the optimal solution.

Here's a C program to solve the Assignment Problem using the Branch and Bound method:

C Code

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

#define N 4 // Number of tasks and workers

// Function prototypes
void assignmentProblem(int costMatrix[N][N]);
int branchAndBound(int costMatrix[N][N], int assignment[], int row, int n,
int bound, int currCost, int minCost, int visited[]);

// Helper function to calculate the lower bound for a given node
int calculateLowerBound(int costMatrix[N][N], int assignment[], int n, int
row, int visited[]);

// Helper function to find the minimum cost assignment
int findMinCost(int costMatrix[N][N], int assignment[], int n, int
currCost, int minCost, int visited[]);

int main() {
    int costMatrix[N][N] = {
        {10, 2, 8, 12},
        {9, 4, 7, 6},
        {5, 11, 13, 10},
        {7, 9, 16, 5}
    };

    assignmentProblem(costMatrix);

    return 0;
}

void assignmentProblem(int costMatrix[N][N]) {
    int assignment[N] = {-1}; // Store the assignment of tasks to workers
    int visited[N] = {0}; // Track visited nodes
    int minCost = INT_MAX; // Initialize minimum cost to a large value

    minCost = branchAndBound(costMatrix, assignment, 0, N, 0, 0, minCost,
visited);

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

int branchAndBound(int costMatrix[N][N], int assignment[], int row, int n,
int bound, int currCost, int minCost, int visited[]) {
    if (row == n) {
        // All tasks are assigned
        if (currCost < minCost) {
            minCost = currCost;
        }
        return minCost;
    }

    for (int col = 0; col < n; col++) {
        if (!visited[col]) {
            // Mark this worker as visited
            visited[col] = 1;
            assignment[row] = col;

            // Calculate the lower bound for this node
```

```
            int newBound = bound + costMatrix[row][col];
            int lowerBound = calculateLowerBound(costMatrix, assignment, n,
row + 1, visited);

            // If the lower bound is less than the minimum cost found so
far, explore further
            if (newBound + lowerBound < minCost) {
                minCost = branchAndBound(costMatrix, assignment, row + 1,
n, newBound, currCost + costMatrix[row][col], minCost, visited);
            }

            // Backtrack
            visited[col] = 0;
            assignment[row] = -1;
        }
    }

    return minCost;
}

int calculateLowerBound(int costMatrix[N][N], int assignment[], int n, int
row, int visited[]) {
    int bound = 0;

    // Calculate the lower bound using row reduction
    for (int i = row; i < n; i++) {
        int min1 = INT_MAX, min2 = INT_MAX;
        for (int j = 0; j < n; j++) {
            if (!visited[j] && costMatrix[i][j] < min1) {
                min2 = min1;
                min1 = costMatrix[i][j];
            } else if (!visited[j] && costMatrix[i][j] < min2) {
                min2 = costMatrix[i][j];
            }
        }
        bound += (min1 == INT_MAX) ? 0 : min1;
        bound += (min2 == INT_MAX) ? 0 : min2;
    }

    // Calculate the lower bound using column reduction
    for (int j = 0; j < n; j++) {
        int min1 = INT_MAX, min2 = INT_MAX;
        for (int i = row; i < n; i++) {
            if (!visited[j] && costMatrix[i][j] < min1) {
                min2 = min1;
                min1 = costMatrix[i][j];
            } else if (!visited[j] && costMatrix[i][j] < min2) {
                min2 = costMatrix[i][j];
            }
        }
        bound += (min1 == INT_MAX) ? 0 : min1;
        bound += (min2 == INT_MAX) ? 0 : min2;
    }

    return bound / 2;
}
```

Explanation:

1. **Function `assignmentProblem`:**

    - Initializes necessary variables and starts the Branch and Bound algorithm.
2. **Function `branchAndBound`:**
    - Explores possible assignments using recursive backtracking.
    - Calculates the cost for each assignment and updates the minimum cost if a better assignment is found.
    - Uses `calculateLowerBound` to prune branches that cannot lead to a better solution.
3. **Function `calculateLowerBound`:**
    - Computes the lower bound for the cost of completing the assignment.
    - Uses row and column reductions to estimate the minimum possible cost from the current state.
4. **Main Function:**
    - Defines the cost matrix and calls `assignmentProblem` to find the minimum cost.

Sample Input and Output:

Sample Input:

The cost matrix is hardcoded in the program:

```
10 2 8 12
9 4 7 6
5 11 13 10
7 9 16 5
```

Sample Output:

```
Minimum cost is 26
```

Result: Thus, the program has been executed successfully.

39.Write a program for to perform liner search.

Aim: To write a ‘C’ program to perform linear search

Algorithm for Linear Search

1. **Input:**
    - A list of elements and the target value to search for.
2. **Iterate Through the List:**
    - Start from the beginning of the list and check each element.
3. **Check for Match:**
    - If the current element matches the target value, return the index of the element.
4. **Continue Searching:**
    - If the end of the list is reached and no match is found, return a value indicating that the target value is not in the list (e.g., `-1`).
5. **Output:**

- The index of the target element if found, otherwise `-1`.

'C' code:

```
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index of the target element
        }
    }
    return -1; // Return -1 if the target element is not found
}

int main() {
    int arr[100];
    int size, target, result;

    // Input: size of the array
    printf("Enter the number of elements in the array: ");
    scanf("%d", &size);

    // Input: elements of the array
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Input: target element to search for
    printf("Enter the element to search for: ");
    scanf("%d", &target);

    // Perform linear search
    result = linearSearch(arr, size, target);

    // Output the result
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Explanation:

1. **Function `linearSearch`:**
   - Takes an array `arr`, its size `size`, and the `target` element to find.
   - Iterates through each element of the array to check if it matches the target.
   - Returns the index of the target element if found, otherwise returns `-1`.
2. **Main Function:**
   - Prompts the user to enter the size of the array and its elements.
   - Prompts the user to enter the target element to search for.

- Calls `linearSearch` to find the target element and prints the result.

Sample Input and Output:

Sample Input:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
10 20 30 40 50
Enter the element to search for: 30
```

Sample Output:

```
Element 30 found at index 2.
```

Result: Thus, the program has been executed successfully.

40.Write a program to find out Hamiltonian circuit Using backtracking method

Aim: To write a 'C' program to find out Hamiltonian circuit using backtracking method

Algorithm for Finding Hamiltonian Circuit Using Backtracking

1. **Input:**
   - An adjacency matrix representing the graph where `graph[i][j]` is `1` if there is an edge between vertex `i` and vertex `j`, and `0` otherwise.
   - A starting vertex.
2. **Initialize:**
   - Create a path array to keep track of the current path.
   - Set the starting vertex as the first element of the path.
3. **Backtracking:**
   - Try to extend the path by adding vertices that are adjacent to the last vertex in the current path and have not been visited yet.
   - Recursively attempt to complete the path.
   - If the path is complete and forms a Hamiltonian circuit (i.e., returns to the starting vertex), print the path.
4. **Pruning:**
   - If the current path cannot be extended to form a Hamiltonian circuit, backtrack and try a different vertex.
5. **Output:**
   - The Hamiltonian circuit if found; otherwise, indicate that no such circuit exists.

C Code

Here's a C program to find a Hamiltonian Circuit using the backtracking method:

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

```
#include <stdbool.h>

#define V 5 // Number of vertices in the graph

// Function to check if the vertex can be added to the Hamiltonian path
bool isSafe(int graph[V][V], int path[], int pos) {
    // Check if this vertex is an adjacent vertex of the previously added
vertex.
    if (graph[path[pos-1]][path[pos]] == 0) {
        return false;
    }

    // Check if the vertex has already been included.
    for (int i = 0; i < pos; i++) {
        if (path[i] == path[pos]) {
            return false;
        }
    }

    return true;
}

// Function to solve the Hamiltonian Circuit problem
bool hamCycleUtil(int graph[V][V], int path[], int pos) {
    // Base case: If all vertices are included in the path
    if (pos == V) {
        // And if there is an edge from the last included vertex to the
first vertex
        return graph[path[pos-1]][path[0]] == 1;
    }

    // Try different vertices as the next candidate in the path
    for (int v = 1; v < V; v++) {
        if (isSafe(graph, path, pos)) {
            path[pos] = v;

            // Recur to build the rest of the path
            if (hamCycleUtil(graph, path, pos + 1)) {
                return true;
            }

            // If adding vertex v doesn't lead to a solution, remove it
            path[pos] = -1;
        }
    }

    return false;
}

// Function to find and print the Hamiltonian Circuit
void findHamiltonianCircuit(int graph[V][V]) {
    int path[V];
    for (int i = 0; i < V; i++) {
        path[i] = -1;
    }

    // Let the first vertex in the path be 0
    path[0] = 0;

    if (hamCycleUtil(graph, path, 1) == false) {
        printf("No Hamiltonian Circuit found\n");
```

```
    } else {
        printf("Hamiltonian Circuit found:\n");
        for (int i = 0; i < V; i++) {
            printf("%d ", path[i]);
        }
        printf("%d\n", path[0]);
    }
}

int main() {
    // Graph representation using adjacency matrix
    int graph[V][V] = {
        {0, 1, 1, 1, 0},
        {1, 0, 1, 1, 1},
        {1, 1, 0, 1, 1},
        {1, 1, 1, 0, 1},
        {0, 1, 1, 1, 0}
    };

    findHamiltonianCircuit(graph);

    return 0;
}
```

Explanation:

1. **Function `isSafe`:**
    - Checks if it is safe to add a vertex to the current path:
        - The vertex must be adjacent to the last vertex in the path.
        - The vertex must not already be included in the path.
2. **Function `hamCycleUtil`:**
    - Recursively attempts to build a Hamiltonian path:
        - Tries different vertices to add to the path.
        - If a valid path is found, it returns `true`.
        - If adding the vertex leads to a dead end, it backtracks and tries another vertex.
3. **Function `findHamiltonianCircuit`:**
    - Initializes the path with the starting vertex.
    - Calls `hamCycleUtil` to find the Hamiltonian Circuit.
    - Prints the circuit if found, otherwise indicates that no circuit was found.
4. **Main Function:**
    - Defines a graph using an adjacency matrix.
    - Calls `findHamiltonianCircuit` to find and print the Hamiltonian Circuit.

Sample Input and Output:

Sample Output:

```
Hamiltonian Circuit found:
0 1 2 3 4 0
```

In this example, the program finds a Hamiltonian Circuit in the graph. The output lists the vertices in the order they are visited in the Hamiltonian Circuit, ending with the starting vertex to complete the cycle.

Result: Thus, the program has been executed successfully.