LCS:

Sure! Let's go through the code line by line and explain its functionality with an example.

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

#define MAX_LENGTH 100

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

The code begins with the inclusion of necessary header files (`stdio.h` and `string.h`). Then, a macro `MAX_LENGTH` is defined with a value of 100, representing the maximum length of the input sequences.

The function `max(int a, int b)` is a utility function that returns the maximum of two integers.

```c
void printLCS(char lcs[MAX_LENGTH][MAX_LENGTH], char* X, int m, int n) {
    if (m == 0 || n == 0) {
        return;
    }

    if (lcs[m][n] == 'd') {
        printLCS(lcs, X, m - 1, n - 1);
        printf("%c", X[m - 1]);
    } else if (lcs[m][n] == 'u') {
        printLCS(lcs, X, m - 1, n);
    } else {
        printLCS(lcs, X, m, n - 1);
    }
}
```

The function `printLCS` is a recursive function that prints the Longest Common Subsequence (LCS) by traversing the `lcs` matrix. It takes four arguments:
- `lcs[MAX_LENGTH][MAX_LENGTH]`: A 2D array that stores the direction of arrows ('d', 'u', 'l') to reconstruct the LCS.
- `X`: The first input sequence.
- `m`: The current index in sequence `X`.
- `n`: The current index in sequence `Y`.

If either `m` or `n` becomes 0, it means we have reached the end of either sequence, so we return from the function.

If the direction stored in `lcs[m][n]` is 'd' (diagonal arrow), we recursively call `printLCS` with `m-1` and `n-1`, and print the character `X[m-1]`, as it belongs to the LCS.

If the direction is 'u' (upward arrow), we recursively call `printLCS` with `m-1` and `n`. This means we move upward in the `lcs` matrix.

If the direction is 'l' (leftward arrow), we recursively call `printLCS` with `m` and `n-1`. This means we move leftward in the `lcs` matrix.

```c
void findLCS(char* X, char* Y) {
    int m = strlen(X);
    int n = strlen(Y);

    int L[m + 1][n + 1];
    char lcs[MAX_LENGTH][MAX_LENGTH];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                L[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
                lcs[i][j] = 'd'; // diagonal arrow
            } else {
                if (L[i - 1][j] >= L[i][j - 1]) {
                    L[i][j] = L[i - 1][j];


                    lcs[i][j] = 'u'; // upward arrow
                } else {
                    L[i][j] = L[i][j - 1];
                    lcs[i][j] = 'l'; // leftward arrow
                }
            }
        }
    }

    printf("\nLongest Common Subsequence (LCS): ");
    printLCS(lcs, X, m, n);
```

```
    printf("\nLength of LCS: %d\n", L[m][n]);
}
```
The `findLCS` function takes two input sequences (`X` and `Y`) and finds the Longest Common Subsequence (LCS) between them. It uses the dynamic programming approach to solve this problem.

First, the lengths of sequences `X` and `Y` are calculated using the `strlen` function.

Next, two matrices are declared:
- `L[m + 1][n + 1]`: A 2D matrix that stores the lengths of the common subsequences. `L[i][j]` represents the length of LCS for the prefixes `X[0...i-1]` and `Y[0...j-1]`.
- `lcs[MAX_LENGTH][MAX_LENGTH]`: A 2D matrix that stores the directions of the arrows ('d', 'u', 'l') to reconstruct the LCS.

The nested `for` loops iterate over the `L` and `lcs` matrices. For each `i` and `j`:
- If either `i` or `j` is 0, it means we have an empty sequence, so `L[i][j]` is set to 0.
- If the characters `X[i - 1]` and `Y[j - 1]` are equal, it means we have found a match. So, the length of LCS increases by 1, and the direction 'd' (diagonal) is stored in `lcs[i][j]`.
- If the characters are not equal, we compare the lengths of the subsequences by moving either upward or leftward in the `L` matrix. The maximum length is chosen, and the corresponding direction ('u' or 'l') is stored in `lcs[i][j]`.

After filling the `L` and `lcs` matrices, the function prints the LCS and its length by calling the `printLCS` function and printing `L[m][n]`.

```c
int main() {
    char X[MAX_LENGTH];
    char Y[MAX_LENGTH];

    printf("Enter the first sequence: ");
    scanf("%s", X);

    printf("Enter the second sequence: ");
    scanf("%s", Y);

    findLCS(X, Y);

    return 0;
}
```
In the `main` function, two character arrays `X` and `Y` are declared with a maximum length of `MAX_LENGTH` (100).

The user is prompted to enter the first sequence and the second sequence using `printf` and `scanf` functions.

Finally, the `findLCS` function is called with the input sequences `X` and `Y`.

That's the end of the code! It takes two sequences as input from the user and finds the Longest Common Subsequence (LCS) using dynamic programming. The LCS is then printed along with its length.

LIS:

Certainly! Let's go through the code line by line and explain its functionality with an example.

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

The code begins with the inclusion of necessary header files (`stdio.h` and `stdlib.h`).

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

The function `max(int a, int b)` is a utility function that returns the maximum of two integers.

```c
void printLIS(int* arr, int* lis, int n) {
    int maxLength = lis[0];
    int maxIndex = 0;
    for (int i = 1; i < n; i++) {
        if (lis[i] > maxLength) {
            maxLength = lis[i];
            maxIndex = i;
        }
    }

    int* lisSeq = (int*)malloc(sizeof(int) * maxLength);
    lisSeq[maxLength - 1] = arr[maxIndex];
    int j = maxLength - 1;

    for (int i = maxIndex - 1; i >= 0; i--) {
        if (arr[i] < arr[maxIndex] && lis[i] == lis[maxIndex] - 1) {
            lisSeq[--j] = arr[i];
```

```
            maxIndex = i;
        }
    }

    printf("\nLongest Increasing Subsequence (LIS): ");
    for (int i = 0; i < maxLength; i++) {
        printf("%d ", lisSeq[i]);
    }
    printf("\n");

    printf("Length of LIS: %d\n", maxLength);

    free(lisSeq);
}
```
The function `printLIS` takes three arguments:
- `arr`: A pointer to the array containing the input sequence.
- `lis`: A pointer to the array containing the lengths of the increasing subsequences.
- `n`: The length of the input sequence.

This function is responsible for printing the Longest Increasing Subsequence (LIS) and its length.

It first finds the maximum length in the `lis` array and its corresponding index. Then, it allocates memory for `lisSeq` array to store the LIS sequence.

Starting from the maximum index, it iterates backwards and checks if the element at index `i` is smaller than the element at the maximum index and if the length of LIS at index `i` is one less than the length at the maximum index. If both conditions are true, it means that the element at index `i` is a part of the LIS. It assigns the element at index `i` to `lisSeq` and decrements the index `j`.

Finally, it prints the LIS sequence and its length, and frees the memory allocated for `lisSeq`.

```c
void findLIS(int* arr, int n) {
    int* lis = (int*)malloc(sizeof(int) * n);

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

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
```

```
            if (arr[i] > arr[j]) {
                lis[i] = max(lis[i], lis[j] + 1);
            }
        }
    }

    printLIS(arr, lis, n);

    free(lis);
}
```
The function `findLIS` takes two arguments:
- `arr`: A pointer to the array containing the input sequence.
- `n`: The length of the input sequence.

This function is responsible for finding the Longest Increasing Subsequence (

LIS) of the input sequence.

It first allocates memory for the `lis` array, which will store the lengths of increasing subsequences for each element of the input sequence. Initially, all elements of `lis` are set to 1, as each element itself is a valid subsequence of length 1.

Then, it uses nested loops to compare each element of the input sequence with the previous elements. If the current element is greater than the previous element, it means that it can be a part of a longer subsequence. The `lis` value for the current element is updated to the maximum of its current value and the `lis` value of the previous element plus one.

After finding the `lis` array, it calls the `printLIS` function to print the LIS and its length, passing the input sequence, `lis` array, and the length of the sequence.

Finally, the memory allocated for `lis` is freed.

```c
int main() {
    int n;
    printf("Enter the number of elements in the sequence: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(sizeof(int) * n);

    printf("Enter the elements of the sequence:\n");
    for (int i = 0; i < n; i++) {
        printf("Element-%d: ",i+1);
```

```
        scanf("%d", &arr[i]);
    }

    findLIS(arr, n);

    free(arr);

    return 0;
}
```

The `main` function is where the program execution starts.

It first prompts the user to enter the number of elements in the sequence and reads the value into the variable `n`.

Memory is then allocated for the `arr` array to store the input sequence.

The user is prompted to enter each element of the sequence, and the elements are stored in the `arr` array using a loop.

After that, the `findLIS` function is called with the input sequence `arr` and its length `n` to find and print the Longest Increasing Subsequence (LIS).

Finally, the memory allocated for `arr` is freed, and the program terminates by returning 0.

That's the end of the code! It takes a sequence of numbers as input from the user and finds the Longest Increasing Subsequence (LIS) using dynamic programming. The LIS and its length are then printed.

Insertion Sort:

Certainly! Let's go through the code line by line and explain its functionality with an example.

```c
#include<stdio.h>

int main(){
    int i, j, n, temp;

    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int a[n];

    printf("Enter the elements of the array:\n");
    for(i = 0; i < n; i++){
```

```
        printf("Index-%d: ", i);
        scanf("%d", &a[i]);
    }

    for(i = 1; i < n; i++){
        temp = a[i];
        j = i - 1;
        while(j >= 0 && a[j] > temp){
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = temp;
    }

    printf("\nSorted array in ascending order is:\n");
    for(i = 0; i < n; i++){
        printf("Index-%d: %d\n", i, a[i]);
    }

    return 0;
}
```

This code implements the insertion sort algorithm to sort an array of integers in ascending order.

Let's understand it with an example:
Suppose we have an array `a` with elements `[5, 2, 8, 3, 1]`.

```c
int i, j, n, temp;

printf("Enter the size of the array: ");
scanf("%d", &n);
int a[n];
```
The code prompts the user to enter the size of the array and reads it into the variable `n`. It then declares an array `a` of size `n` to store the elements.

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

```
The user is prompted to enter the elements of the array one by one. The elements are read into the array `a`.

```c
for(i = 1; i < n; i++){
    temp = a[i];
    j = i - 1;
    while(j >= 0 && a[j] > temp){
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = temp;
}
```
The insertion sort algorithm is implemented here. It iterates over the array starting from the second element (`i = 1`).

In each iteration, the current element (`a[i]`) is stored in the variable `temp`. The variable `j` is set to `i - 1`.

A while loop is executed as long as `j` is greater than or equal to 0 and the element at index `j` is greater than `temp`. In each iteration of the while loop, the element at index `j` is shifted to the right by assigning it to `a[j + 1]`. The value of `j` is decremented.

After coming out of the while loop, the correct position for `temp` is found, and it is placed at `a[j + 1]`.

The process continues until all elements are in their correct sorted positions.

```c
printf("\nSorted array in ascending order is:\n");
for(i = 0; i < n; i++){
    printf("Index-%d: %d\n", i, a[i]);
}
```
Finally, the sorted array is printed in ascending order, displaying the elements along with their respective indices.

That's the end of the code explanation. If you have any further questions, feel free to ask!

Merge sort:

Certainly! Let's go through the code line by line and explain its functionality with an example.
```

```c
#include <stdio.h>
#include <stdlib.h>
```
The code begins with the inclusion of necessary header files (`stdio.h` and `stdlib.h`).

```c
void merge(int arr[], int left[], int left_size, int right[], int right_size) {
    int i = 0, j = 0, k = 0;

    while (i < left_size && j < right_size) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    while (i < left_size) {
        arr[k++] = left[i++];
    }

    while (j < right_size) {
        arr[k++] = right[j++];
    }
}
```
The `merge` function takes five arguments:
- `arr`: The array to be merged (resulting merged array will be stored in this array).
- `left`: The left subarray.
- `left_size`: The size of the left subarray.
- `right`: The right subarray.
- `right_size`: The size of the right subarray.

This function performs the merging of two sorted subarrays (`left` and `right`) into a single sorted array (`arr`). It compares elements from both subarrays and places them in the correct order in the merged array.

The function uses three pointers (`i`, `j`, and `k`) to keep track of the current positions in the `left`, `right`, and `arr` arrays, respectively.

The first `while` loop continues until there are elements remaining in both `left` and `right` subarrays. It compares the current elements from both subarrays and selects the smaller one to

be placed in the `arr` array. The corresponding pointer (`i` or `j`) and `k` are incremented accordingly.

The next two `while` loops handle the case where there are remaining elements in either the `left` or the `right` subarray. They copy any remaining elements to the `arr` array.

```c
void merge_sort(int arr[], int size) {
    if (size < 2) {
        return;
    }
    int mid = size / 2;
    int left[mid], right[size - mid];
    for (int i = 0; i < mid; i++) {
        left[i] = arr[i];
    }
    for (int i = mid; i < size; i++) {
        right[i - mid] = arr[i];
    }
    merge_sort(left, mid);
    merge_sort(right, size - mid);
    merge(arr, left, mid, right, size - mid);
}
```
The `merge_sort` function takes two arguments:
- `arr`: The array to be sorted.
- `size`: The size of the array.

This function implements the merge sort algorithm to sort the array in ascending order.

The base case is when the `size` is less than 2, which means the array is already sorted (0 or 1 element), so it returns.

Otherwise, it calculates the midpoint (`mid`) of the array and creates two subarrays: `left` and `right`. The `left` subarray contains elements from index 0 to `mid-1`, and the `right` subarray contains elements from index `mid` to `size-1`.

It recursively calls `merge_sort` on both subarrays to sort them individually.

Finally, it calls the `merge` function to merge the sorted `left` and `right` subarrays into the original `arr` array.

```
```

```c
int main() {
    int n, i;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int a[n];
    printf("\nEnter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        printf("Index-%d: ", i);
        scanf("%d", &a[i]);
    }

    // Call merge sort
    merge_sort(a, n);

    printf("\nSorted array in ascending:\n");
    for (int i = 0; i < n; i++) {
        printf("Index-%d: %d\n", i, a[i]);
    }
    printf("\n");

    return 0;
}
```

The `main` function is where the program execution starts.

It prompts the user to enter the size of the array (`n`) and reads the value into the variable `n`.

It then declares an array `a` of size `n` to store the elements of the array.

Next, it prompts the user to enter each element of the array, and the elements are stored in the `a` array using a loop.

After that, it calls the `merge_sort` function, passing the array `a` and its size `n` to sort the array in ascending order.

Finally, it prints the sorted array and its indices.

That's the end of the code! It takes an array of integers as input from the user and sorts it using the merge sort algorithm. The sorted array is then printed in ascending order.

Quick Sort:

Certainly! Let's go through the code line by line and explain its functionality with an example.

```c
#include<stdio.h>
```
The code begins with the inclusion of the necessary header file `stdio.h` for input/output operations.

```c
void quicksort(int number[], int first, int last)
{
   int i, j, pivot, temp;
   if (first < last)
   {
      pivot = first;
      i = first;
      j = last;
      while (i < j)
      {
         while (number[i] <= number[pivot] && i < last)
            i++;
         while (number[j] > number[pivot])
            j--;
         if (i < j)
         {
            temp = number[i];
            number[i] = number[j];
            number[j] = temp;
         }
      }
      temp = number[pivot];
      number[pivot] = number[j];
      number[j] = temp;
      quicksort(number, first, j - 1);
      quicksort(number, j + 1, last);
   }
}
```
The `quicksort` function is an implementation of the Quick Sort algorithm to sort an array of integers in ascending order. It takes three arguments:
- `number`: The array to be sorted.
- `first`: The index of the first element of the subarray to be sorted.
- `last`: The index of the last element of the subarray to be sorted.

Inside the function, it first checks if `first` is less than `last`. If not, it means there is only one element in the subarray, and no sorting is required, so it returns.

If there are multiple elements in the subarray, it selects a pivot element (`number[first]` in this case) and initializes two pointers `i` and `j` to the first and last elements of the subarray, respectively.

It enters a while loop that continues until `i` is less than `j`. Within the loop, it finds elements from the left side greater than the pivot and elements from the right side smaller than or equal to the pivot. If such elements are found, it swaps them.

After the while loop ends, it swaps the pivot element with the element at index `j`, placing the pivot in its correct sorted position.

Then, it recursively calls `quicksort` on the subarrays to the left and right of the pivot index `j` to sort them separately.

```c
int main()
{
   int i, n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d integers:\n", n);
   for (i = 0; i < n; i++)
   {
      printf("Index-%d: ", i);
      scanf("%d", &arr[i]);
   }
   quicksort(arr, 0, n - 1);
   printf("\nOrder of Sorted elements in ascending:\n");
   for (i = 0; i < n; i++)
   {
      printf("Index-%d: %d\n", i, arr[i]);
   }
   return 0;
}
```

The `main` function is where the program execution starts.

It prompts the user to enter the number of elements in the array and reads the value into the variable `n`.

It then declares an array `arr` of size `n` to store the elements of the array.

Next, it prompts the user to enter each element of the array, and the elements are stored in the `arr` array using a loop.

After that, it calls the `quicksort` function, passing the array `arr`, the index of

the first element (`0`), and the index of the last element (`n - 1`) to sort the entire array.

Finally, it prints the sorted array with its indices.

Here's an example run of the program:

```
Enter the number of elements: 6
Enter 6 integers:
Index-0: 8
Index-1: 3
Index-2: 1
Index-3: 6
Index-4: 2
Index-5: 9

Order of Sorted elements in ascending:
Index-0: 1
Index-1: 2
Index-2: 3
Index-3: 6
Index-4: 8
Index-5: 9
```

In this example, we enter 6 integers (8, 3, 1, 6, 2, 9), and the program sorts them in ascending order using the Quick Sort algorithm. The sorted array is then printed with their respective indices.

Selection sort:

Certainly! Let's go through the code line by line and explain its functionality with an example.

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

The code begins with the inclusion of the necessary header file `stdio.h` for input/output operations.

```c
```

```
int main()
{
   int i, j, n, temp, position;

   printf("Enter the size of the array: ");
   scanf("%d", &n);
   int a[n]; // Array initialization

   printf("\nEnter the elements of the array:\n", n);
   for (i = 0; i < n; i++)
   {
      printf("Index-%d: ", i);
      scanf("%d", &a[i]);
   }
```

The `main` function is where the program execution starts.

It declares variables `i` and `j` for loop iterations, `n` to store the size of the array, `temp` for temporary value storage, and `position` to keep track of the position of the maximum element.

It prompts the user to enter the size of the array and reads the value into the variable `n`.

Then, an array `a` of size `n` is declared to store the elements of the array.

Next, it prompts the user to enter each element of the array, and the elements are stored in the `a` array using a loop.

```c
for (i = 0; i < n - 1; i++)
{
   position = i;
   for (j = i + 1; j < n; j++)
   {
      if (a[j] > a[position])
      {
         position = j;
      }
   }

   if (position != i)
   {
      temp = a[i];
      a[i] = a[position];
      a[position] = temp;
```

```
    }
}
```
This code block performs selection sort in descending order. It uses nested loops to iterate over the array and find the position of the maximum element.

The outer loop runs from `i = 0` to `i < n - 1` and selects the `i`th element as the current maximum.

The inner loop starts from `j = i + 1` and compares each element from `a[i+1]` to `a[n-1]` with the current maximum element at `a[position]`. If a larger element is found, the position is updated to `j`.

After the inner loop completes, it checks if the position of the maximum element is not equal to `i`. If it's not equal, it means a larger element was found, so it swaps the elements at positions `i` and `position`.

This process continues until the entire array is sorted in descending order.

```c
printf("\nSorted array in ascending order:\n");
for (i = 0; i < n; i++)
{
    printf("Index-%d: %d\n", i, a[i]);
}

return 0;
}
```
Finally, the sorted array is printed in ascending order. It uses a loop to iterate over the array and prints each element along with its index.

Here's an example run of the program:

```
Enter the size of the array: 6
Enter the elements of the array:
Index-0: 8
Index-1: 3
Index-2: 1
Index-3: 6
Index-4: 2
Index-5: 9
```

Sorted array in ascending order:
Index-0: 1
Index-1: 2
Index-2: 3
Index-3: 6
Index-4: 8
Index-5: 9
```

In this example,

the user enters an array of size 6 with elements [8, 3, 1, 6, 2, 9]. The program sorts the array in descending order using selection sort and then prints the sorted array in ascending order.

## Coin change:

Certainly! Let's go through the code line by line and explain its functionality with an example.

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

The code begins with the inclusion of the necessary header file `stdio.h` for input/output operations.

```c
int minCoins(int coins[], int numCoins, int target) {
    int dp[target + 1];
    dp[0] = 0;

    for (int i = 1; i <= target; i++) {
        dp[i] = target + 1;
    }

    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < numCoins; j++) {
            if (coins[j] <= i) {
                int subResult = dp[i - coins[j]];
                if (subResult != target + 1 && subResult + 1 < dp[i]) {
                    dp[i] = subResult + 1;
                }
            }
        }
    }

    return dp[target];
```
```

}
```

This code defines a function `minCoins` that calculates the minimum number of coins required to make change for a given target value.

The function takes an array `coins[]` representing the values of different coins, `numCoins` representing the number of coins available, and `target` representing the target value for which change needs to be made.

The function uses dynamic programming to find the minimum number of coins required. It initializes an array `dp[]` of size `target + 1` to store the minimum number of coins for each value from 0 to the target.

The outer loop iterates from 1 to the target value, and for each value, the inner loop iterates over the coins array. It checks if the coin value is less than or equal to the current value `i`.

If the condition is true, it calculates the minimum number of coins required for the remaining value by accessing `dp[i - coins[j]]`. If this value is not equal to `target + 1` (indicating a valid combination) and if the current number of coins required plus one is less than the current value of `dp[i]`, it updates `dp[i]` with the new minimum value.

Finally, the function returns `dp[target]`, which represents the minimum number of coins required to make change for the target value.

```c
int maxCoins(int coins[], int numCoins, int target) {
    int dp[target + 1];
    dp[0] = 0;

    for (int i = 1; i <= target; i++) {
        dp[i] = -1;
    }

    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < numCoins; j++) {
            if (coins[j] <= i) {
                int subResult = dp[i - coins[j]];
                if (subResult != -1 && subResult + 1 > dp[i]) {
                    dp[i] = subResult + 1;
                }
            }
        }
    }
```
```

```
    return dp[target];
}
```
This code defines a similar function `maxCoins` that calculates the maximum number of coins required to make change for a given target value.

The logic is similar to `minCoins`, but instead of initializing `dp[]` with `target + 1`, it is initialized with `-1`.

The condition to update `dp[i]` checks if `subResult` is not equal to `-1` (indicating a valid combination) and if the current number of coins required plus one is greater than the current value of `dp[i]`, it updates `dp[i]` with the new maximum value.

Finally

, the function returns `dp[target]`, which represents the maximum number of coins required to make change for the target value.

```c
int main() {
    int numCoins;
    printf("Enter the number of coins: ");
    scanf("%d", &numCoins);

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

    int target;
    printf("Enter the target value: ");
    scanf("%d", &target);

    int minCount = minCoins(coins, numCoins, target);
    int maxCount = maxCoins(coins, numCoins, target);

    printf("Minimum number of coins required: %d\n", minCount);
    printf("Maximum number of coins required: %d\n", maxCount);

    return 0;
}
```
The `main` function serves as the entry point of the program.

It prompts the user to enter the number of coins and reads it into the variable `numCoins`. Then, it initializes an array `coins[]` of size `numCoins` to store the values of the coins. The user is prompted to enter the values of the coins one by one, and they are stored in the array.

Next, the user is prompted to enter the target value for which change needs to be made, and it is stored in the variable `target`.

The program calls the `minCoins` function with the `coins[]`, `numCoins`, and `target` to calculate the minimum number of coins required and stores the result in `minCount`.

Similarly, it calls the `maxCoins` function to calculate the maximum number of coins required and stores the result in `maxCount`.

Finally, the program prints the minimum and maximum number of coins required to make change for the target value.

That's the end of the code explanation. If you have any further questions, feel free to ask!

BFS:

Certainly! Let's go through the code line by line and explain its functionality with an example.

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

#define MAX_SIZE 100
```

The code begins with the inclusion of necessary header files `stdio.h` for input/output operations and `stdbool.h` for using the `bool` type. It also defines a constant `MAX_SIZE` as 100.

```c
typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;
```

Here, a structure `Queue` is defined to represent a queue. It contains an array `items` to store the elements of the queue and two integer variables `front` and `rear` to keep track of the front and rear indices of the queue.

```c
void enqueue(Queue* queue, int item) {
```

```
    if (queue->rear == MAX_SIZE - 1) {
        printf("Queue is full\n");
    } else {
        if (queue->front == -1) {
            queue->front = 0;
        }
        queue->rear++;
        queue->items[queue->rear] = item;
    }
}
```

The function `enqueue` is used to insert an element into the queue. It takes a pointer to a `Queue` structure and an integer item as parameters. If the rear index of the queue is at the maximum size, it prints a message indicating that the queue is full. Otherwise, it increments the rear index and inserts the item into the `items` array at the updated rear index.

```c
int dequeue(Queue* queue) {
    int item;
    if (queue->front == -1 || queue->front > queue->rear) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
        return item;
    }
}
```

The function `dequeue` is used to remove and retrieve an element from the front of the queue. It takes a pointer to a `Queue` structure as a parameter. If the front index of the queue is -1 or greater than the rear index, it prints a message indicating that the queue is empty and returns -1. Otherwise, it retrieves the item from the `items` array at the front index, increments the front index, and checks if the front index has exceeded the rear index. If so, it updates both the front and rear indices to -1.

```c
bool isEmpty(Queue* queue) {
    return queue->front == -1;
}
```

The function `isEmpty` is used to check if the queue is empty. It takes a pointer to a `Queue` structure as a parameter and returns `true` if the front index of the queue is -1, indicating an empty queue, and `false` otherwise.

```c
void BFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int startVertex) {
    bool visited[MAX_SIZE] = { false };

    Queue queue;
    queue.front = -1;
    queue.rear = -1;

    visited[startVertex] = true;
    enqueue(&queue, startVertex);

    printf("BFS traversal: ");

    while (!isEmpty(&queue)) {
        int currentVertex = dequeue(&queue);
        printf("%d ", currentVertex);

        for (int i = 0; i < vertices; ++i) {
            if (adjacencyMatrix[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = true;
                enqueue(&queue, i

);
            }
        }
    }

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

The function `BFS` performs a breadth-first search traversal on a graph represented by an adjacency matrix. It takes the adjacency matrix, the number of vertices, and the starting vertex as parameters.

Within the function, an array `visited` of size `MAX_SIZE` is created and initialized to `false`. It represents the visited status of each vertex during the traversal.

A queue is also initialized using the `Queue` structure. The front and rear indices of the queue are set to -1 initially.

The start vertex is marked as visited, and it is enqueued into the queue using the `enqueue` function.

The BFS traversal starts with a message indicating the traversal process: "BFS traversal: ".

A loop is executed until the queue becomes empty. In each iteration, the front vertex of the queue is dequeued using the `dequeue` function and stored in the variable `currentVertex`. The current vertex is then printed.

The adjacent vertices of the current vertex are examined in the adjacency matrix. If an adjacent vertex is connected to the current vertex (value 1 in the adjacency matrix) and it has not been visited before, it is marked as visited, and it is enqueued into the queue.

Once the traversal is completed, a new line is printed.

```c
int main() {
    int vertices;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

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

    int startVertex;
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    BFS(adjacencyMatrix, vertices, startVertex);

    return 0;
}
```

The `main` function serves as the entry point of the program. It prompts the user to enter the number of vertices in the graph and reads it into the variable `vertices`.

An adjacency matrix of size `MAX_SIZE` is declared to store the connectivity information of the graph.

The user is prompted to enter the adjacency matrix, row by row, representing the edges between the vertices. The matrix elements are read into the `adjacencyMatrix` array.

The user is also prompted to enter the starting vertex for the BFS traversal, which is stored in the variable `startVertex`.

The `BFS` function is called with the adjacency matrix, the number of vertices, and the starting vertex to perform the BFS traversal.

Finally, the program returns 0, indicating successful execution.

That's the end of the code explanation. If you have any further questions, feel free to ask!

DFS:

Certainly! Let's go through the code line by line and explain its functionality with an example.

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

#define MAX_SIZE 100
```

The code begins with the inclusion of necessary header files `stdio.h` for input/output operations and `stdbool.h` for using the `bool` type. It also defines a constant `MAX_SIZE` as 100.

```c
void DFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int startVertex, bool visited[]) {
    visited[startVertex] = true;

    for (int i = 0; i < vertices; i++) {
        if (adjacencyMatrix[startVertex][i] == 1 && !visited[i]) {
            DFS(adjacencyMatrix, vertices, i, visited);
        }
    }

    printf("%d ", startVertex); // Print the vertex after traversing its left and right subtrees
}
```

The function `DFS` performs a depth-first search traversal on a graph represented by an adjacency matrix. It takes the adjacency matrix, the number of vertices, the starting vertex, and a boolean array `visited` as parameters.

Within the function, the `visited` array is updated to mark the current `startVertex` as visited.

A loop is executed for each vertex in the graph. If the current vertex is connected to the `startVertex` (indicated by a value of 1 in the adjacency matrix) and it has not been visited before, a recursive call to `DFS` is made with the current vertex as the new `startVertex`.

After traversing all the adjacent vertices, the `startVertex` is printed. This print statement is responsible for the "Inorder traversal" of the graph.

```c
int main() {
    int vertices;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

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

    int startVertex;
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    bool visited[MAX_SIZE] = { false };

    printf("Inorder traversal: ");
    DFS(adjacencyMatrix, vertices, startVertex, visited);
    printf("\n");

    return 0;
}
```

The `main` function serves as the entry point of the program. It prompts the user to enter the number of vertices in the graph and reads it into the variable `vertices`.

An adjacency matrix of size `MAX_SIZE` is declared to store the connectivity information of the graph.

The user is prompted to enter the adjacency matrix, row by row, representing the edges between the vertices. The matrix elements are read into the `adjacencyMatrix` array.

The user is also prompted to enter the starting vertex for the DFS traversal, which is stored in the variable `startVertex`.

A boolean array `visited` of size `MAX_SIZE` is declared and initialized to `false`. It is used to track the visited status of each vertex during the traversal.

A message "Inorder traversal: " is printed.

The `DFS` function is called with the adjacency matrix, the number of vertices, the starting vertex, and the `visited` array to perform the DFS traversal.

Finally, a new line is printed, and the program returns 0, indicating successful execution.

That's the end of the code explanation. If you have any further questions, feel free to ask!