

Code explanation assistance



i have viva and practical exam
if i provide you my code then can you help me by explaining the code line by line,also what is use of the code and how it works
and if possible can you give video to understand the concept in code

Yes, I can definitely help you with explaining your code line by line, detailing its functionality, and providing insights into how it works. However, I cannot provide video content directly. What I can do is explain the concepts in depth and provide visual or step-by-step breakdowns to help you better understand how the code functions.

If you share the code, I'll go through it and break it down for you. Feel free to upload your code or provide details on any specific concepts or problems you're working on!



```
#include<iostream>
using namespace std;

int find(int, int);
void print(int, int);
int c[20][20], w[20][20], r[20][20], p[20], q[20], k, m, i, j, n;
char idtr[10][7];

int main()
{
    cout<<"\nEnter number of identifiers: ";
    cin>>n;
    for(i = 1; i <= n; i++)
    {
        cout<<"Enter Identifier "<<i<<": ";
        cin>>idtr[i];
    }
    for(i = 1; i <= n; i++)
    {
        cout<<"Enter successful probability of "<<i<<": ";
        cin>>p[i];
    }
    for(i = 0; i <= n; i++)
    {
        cout<<"Enter unsuccessful probability of "<<i<<": ";
        cin>>q[i];
    }
    for(i = 0; i <= n; i++)
    {
```

```

w[i][i] = q[i];
c[i][i] = r[i][i] = 0;
cout<<"\nW: "<<w[i][i]<<" | c: "<<c[i][i]<<" | r: "<<r[i][i];
}
for(i = 0; i < n; i++)
{
    j = i + 1;
    w[i][j] = p[j] + q[j] + q[i];
    c[i][j] = w[i][j];
    r[i][j] = j;
    cout<<"\nW: "<<w[i][j]<<" | c: "<<c[i][j]<<" | r: "<<r[i][j];
}
for(m = 2; m <= n; m++)
{
    for(i = 0; i <= n-m; i++)
    {
        j = i + m;
        w[i][j] = p[j] + q[j] + w[i][j-1];
        c[i][j] = 1000;
        for(k = i + 1; k <= j; k++)
        {
            int cost = c[i][k-1] + c[k][j];
            if(cost < c[i][j])
            {
                c[i][j] = cost;
                r[i][j] = k;
            }
        }
        c[i][j] += w[i][j];
        cout<<"\nW: "<<w[i][j]<<" | c: "<<c[i][j]<<" | r: "<<r[i][j];
    }
}
cout<<"\nFinal OBST is: ";
print(0, n);
return 0;
}

void print(int i, int j)
{
    if(i < j)
        cout<<"\n"<<idtr[r[i][j]];
    else
        return;
    print(i, r[i][j] - 1);
    print(r[i][j], j);
}

```

Let's break down the code line by line and explain how it works, as well as its purpose.

Code Overview

This C++ program is designed to implement an **Optimal Binary Search Tree (OBST)** algorithm. The goal is to construct an optimal binary search tree using a set of identifiers and associated probabilities of successful and unsuccessful searches.

The input consists of:

- Identifiers (strings).
- Successful probabilities of each identifier.
- Unsuccessful probabilities (probability of a failed search between identifiers).

The program calculates the structure of the optimal binary search tree by using dynamic programming. It minimizes the expected search cost based on the probabilities of success and failure.

Code Breakdown

cpp

```
#include<iostream>
using namespace std;
```

- This includes the necessary header file `<iostream>` to allow input and output operations. The `using namespace std;` is used to avoid repeatedly writing `std::` before standard functions like `cout` and `cin`.

cpp

```
int find(int, int);
void print(int, int);
int c[20][20], w[20][20], r[20][20], p[20], q[20], k, m, i, j, n;
char idtr[10][7];
```

- **Global variables:**
 - `c[20][20]`: A 2D array to store the minimum cost of constructing an OBST for subarrays.
 - `w[20][20]`: A 2D array to store the sum of probabilities for a range of identifiers.
 - `r[20][20]`: A 2D array to store the root of the subtree for the optimal BST.
 - `p[20]`: Array storing the successful probability of each identifier.
 - `q[20]`: Array storing the unsuccessful probability of each identifier.
 - `idtr[10][7]`: A 2D array to store the identifier names (strings).
 - Other variables like `k`, `m`, `i`, `j`, and `n` are used as loop counters and to store intermediate values.

cpp

```
int main()
{
    cout<<"\nEnter number of identifiers: ";
    cin>>n;
```

- The program starts by asking for the number of identifiers `n`.

cpp

```
for(i = 1; i <= n; i++)
{
    cout<<"Enter Identifier "<<i<<": ";
    cin>>idtr[i];
}
```

- A loop that takes input for each identifier (stored as a string in the `idtr` array).

cpp

```
for(i = 1; i <= n; i++)
{
    cout<<"Enter successful probability of "<<i<<": ";
    cin>>p[i];
}
```

- Another loop that takes input for the successful probability `p[i]` for each identifier.

cpp

```
for(i = 0; i <= n; i++)
{
    cout<<"Enter unsuccessful probability of "<<i<<": ";
    cin>>q[i];
}
```

- Loop to take input for the unsuccessful probability `q[i]` for each identifier.

cpp

```
for(i = 0; i <= n; i++)
{
    w[i][i] = q[i];
    c[i][i] = r[i][i] = 0;
    cout<<"\nW: "<<w[i][i]<<" | c: "<<c[i][i]<<" | r: "<<r[i][i];
}
```

- Initializes base cases for the DP arrays. For `i = 0`, `w[0][0]` is set to `q[0]` (unsuccessful probability). Similarly, `c[0][0]` and `r[0][0]` are set to 0, which represents no cost or root.

cpp

```
for(i = 0; i < n; i++)
{
    j = i + 1;
    w[i][j] = p[j] + q[j] + q[i];
    c[i][j] = w[i][j];
    r[i][j] = j;
    cout<<"\nW: "<<w[i][j]<<" | c: "<<c[i][j]<<" | r: "<<r[i][j];
}
```

- This part calculates `w[i][j]`, which is the sum of probabilities for a range `[i, j]`. Then it initializes `c[i][j]` to the same value (which is the cost of having only one root). The root `r[i][j]` is set to `j`, meaning the identifier at position `j` is considered the root for this range.

cpp

```
for(m = 2; m <= n; m++)
{
    for(i = 0; i <= n-m; i++)
    {
        j = i + m;
        w[i][j] = p[j] + q[j] + w[i][j-1];
        c[i][j] = 1000;
        for(k = i + 1; k <= j; k++)
        {
            int cost = c[i][k-1] + c[k][j];
            if(cost < c[i][j])
            {
                c[i][j] = cost;
                r[i][j] = k;
            }
        }
        c[i][j] += w[i][j];
        cout<<"\nW: "<<w[i][j]<<" | c: "<<c[i][j]<<" | r: "<<r[i][j];
    }
}
```

- This part contains the core of the dynamic programming solution. The outer loop iterates over the lengths of subarrays, and the inner loop computes the optimal root for each subarray `[i, j]`.
 - `w[i][j]` is updated by adding the probability of `p[j]` and `q[j]`.
 - The `c[i][j]` is initialized to a large value (`1000`), and then we check for the minimum cost by trying all possible roots `k` within the range.
 - The result is stored in `c[i][j]`, and the optimal root for this range is stored in `r[i][j]`.

cpp

```
cout<<"\nFinal OBST is: ";
print(0, n);
return 0;
}
```

- After the dynamic programming tables are filled, the function `print()` is called to print the optimal binary search tree by recursively displaying the roots.

cpp

```
void print(int i, int j)
{
    if(i < j)
        cout<<"\n"<<idtr[r[i][j]]; // Print the root for the range
    else
        return;
    print(i, r[i][j] - 1); // Recursively print the left subtree
    print(r[i][j], j); // Recursively print the right subtree
}
```

- This function recursively prints the optimal binary search tree structure using the `r` table, which holds the root for each range. It prints the root and then recursively prints the left and right subtrees.

What the Code Does

1. **Input:** The program takes n identifiers and their respective successful and unsuccessful probabilities.
2. **Dynamic Programming:** It then constructs a table to compute the optimal binary search tree by minimizing the cost of searching for each identifier. It calculates the minimum cost for each subtree and stores the optimal root for each range.
3. **Output:** Finally, it prints the structure of the optimal binary search tree by recursively displaying the roots.

Key Concepts

- **Dynamic Programming (DP):** The solution uses DP to break the problem of constructing an optimal binary search tree into smaller subproblems.
- **Optimal Binary Search Tree:** An OBST minimizes the expected search cost, given the probabilities of success and failure for each search.
- **Recursion:** The tree structure is printed using recursion to display the left and right subtrees of each root.

Let me know if you'd like further clarification or need help with any specific part of this explanation!