卡特蓝数：96. 不同的二叉搜索树

这段代码使用动态规划来计算所有可能的二叉搜索树的数量。

1. 定义动态规划数组 dp

cpp 复制

```cpp
vector<int> dp(n + 1);
dp[0] = 1;
```

- dp[i] 表示节点数量为 i 时，可以构造出的不同二叉搜索树的数量。
- dp[0] = 1 表示当没有节点时，只有一种空树的情况（即 1 种情况）。

2. 外层循环 i

cpp 复制

```cpp
for (int i = 1; i <= n; i++)
```

- i 表示当前正在考虑有 i 个节点的情况下可以构造出的二叉搜索树的数量。

3. 内层循环 j

cpp 复制

```cpp
for (int j = 1; j <= i; j++) {
    dp[i] += dp[j-1] * dp[i-j];
}
```

- j 表示将节点 j 作为根节点。对于每一个 j：
  - 左子树包含 j-1 个节点，这部分有 dp[j-1] 种构造方式。
  - 右子树包含 i-j 个节点，这部分有 dp[i-j] 种构造方式。
  - 将左子树和右子树的构造方式组合起来，总的构造方式就是 dp[j-1] * dp[i-j]。
  - dp[i] 通过将所有可能的 j 作为根节点的构造方式累加得出。

4. 返回结果

cpp 复制

```cpp
return dp[n];
```

- 最终返回 dp[n]，它表示可以构造 n 个节点的不同二叉搜索树的数量。

举例说明

假设 n = 3，我们计算有 3 个节点时可以构造的不同二叉搜索树的数量。

初始状态

- 初始化 dp 数组：

plaintext 复制

```
dp = [1, 0, 0, 0]  // dp[0] = 1 表示空树的情况。
```

第一步：计算 dp[1]

- 当 i = 1 时：
  - j = 1：左子树有 dp[0] 种构造方式，右子树有 dp[0] 种构造方式。
  - 所以 dp[1] = dp[0] * dp[0] = 1。

dp 数组更新为：

plaintext 复制

```
dp = [1, 1, 0, 0]
```

第二步：计算 dp[2]

- 当 i = 2 时：
  - j = 1：左子树有 dp[0] 种构造方式，右子树有 dp[1] 种构造方式。

- j = 2：左子树有 dp[1] 种构造方式，右子树有 dp[0] 种构造方式。
- 所以 dp[2] = dp[0] * dp[1] + dp[1] * dp[0] = 1 * 1 + 1 * 1 = 2。

dp 数组更新为：

plaintext 复制

```
dp = [1, 1, 2, 0]
```

第三步：计算 dp[3]

- 当 i = 3 时：

```cpp
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n+1);
       dp[0] = 1;
        for(int i = 1;i <= n;i++)
        {
            for(int j = 1;j <= i;j++)
            {
                dp[i] += dp[j-1]*dp[i-j];
            }
        }
        return dp[n];
    }
};
```

多源BFS： 542. 01 矩阵

```cpp
class Solution {
    int dx[4] = {1, -1, 0, 0};
    int dy[4] = {0, 0, 1, -1};
    int m, n;

public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
        m = mat.size();
        n = mat[0].size();
        vector<vector<int>> dist(m, vector<int>(n, -1));
        queue<pair<int, int>> q;

        // 1.
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (mat[i][j] == 0) {
                    q.push({i, j});
                    dist[i][j] = 0;
                }
            }
        }

        // 2.
        while (q.size()) {
            auto [a, b] = q.front();
            q.pop();
            for (int i = 0; i < 4; i++) {
                int x = a + dx[i];
                int y = b + dy[i];
                if (x >= 0 && x < m && y >= 0 && y < n && dist[x][y] == -1) {
                    dist[x][y] = dist[a][b] + 1;
                    q.push({x, y});
                }
            }
        }
        return dist;
    }
};
```