实验8 实现最长子公共序列的动态规划算法和实现构建最优二叉搜索树算法

姓名：邵宁录　　学号：2018202195

目录

一、问题描述

1. 实现最长子公共序列的动态规划算法
    - **输入：** 两个字符串s1和s2
    - **输出：** 最长公共子序列及其长度
2. 实现构建最优二叉搜索树算法
    - **输入：** 有效节点的概率与个数，无效节点的概率
    - **输出：** 最优二叉搜索树节点情况

二、算法基本思路

1. 最长公共子序列

该算法实现思路可以用 **自底向上** ，也可以用 **带备忘的自顶向下** 来实现，两种方法的实现难度差不多。

首先是 **自底向上** 的方法。该方法从二维表的左上角开始逐行计算，自左向右。计算方法按照下式：

$$c[i,\ j] = \begin{cases} 0, & i = 0\ or j = 0 \\ c[i-1,\ j-1] + 1, & i,\ j > 0\ and\ x_i = y_i \\ max(c[i,\ j-1],\ c[i-1,\ j]), & i,\ j > 0\ and\ x_i \neq y_i \end{cases}$$

然后是 **带备忘的自顶向下** 算法。该方法与上一个方法无本质区别，仅是把二维表的赋值改为递归调用函数。

2. 最优二叉搜索树

最优二叉搜索树的实现，我选择使用了 **自底向上** 的方法。

有递推公式如下：

$$e[i,\ j] = \begin{cases} q_{i-1}, & j = i-1 \\ min_{i \le r \le j}(e[i,\ r-1] + e[r+1,\ j] + w[i,\ j]), & i \le j \end{cases}$$

其中，$e[i,\ j]$ 表示第i个有效节点到第j个有效节点形成的树的期望搜索代价，$w[i,\ j]$ 表示第i个有效节点到第j个有效节点的搜索概率和加上第i-1个无效节点到第j个无效节点的搜索概率和。

三、算法复杂度分析

1. 最长公共子序列

因为有 $\mathrm{O}(mn)$ 个子问题，且每个子问题的时间复杂度为常数 $\mathrm{O}(1)$ 。因此两种方法实现的最长公共子序列的时间复杂度为 $\mathrm{O}(mn)$ 。

2. 最优二叉搜索树

因为有 $\mathrm{O}(n^2)$ 个子问题，且每个子问题的时间复杂度为常数 $\mathrm{O}(n)$ 。因此该方法实现的最优二叉搜索树的时间复杂度为 $\mathrm{O}(n^3)$ 。

四、源码

1. 最长公共子序列

```cpp
/*
 * @Description: 算法导论第15章最长公共子序列实现
 * @Author: rainym00d
 * @Github: https://github.com/rainym00d
 * @Date: 2020-11-01 19:37:54
 * @LastEditors: rainym00d
 * @LastEditTime: 2020-11-06 11:52:25
 */
#include <iostream>
#include <cstdlib>
#include <vector>
#include <string>

using namespace std;

const int SUP = 23333333;
const int INF = -23333333;
const int N = 1024;
```

```
int c[N][N];
int b[N][N];    // 1: 左上  2: 上   3: 左

// 带备忘的自顶向下
int MemoizedLcsLengthAux(string &s1, string &s2, int i, int j)
{
    // 边界条件
    if (c[i][j] != -1)
    {
        return c[i][j];
    }
    // 状态转移方程
    // 第一种情况
    if (i == 0 || j == 0)
    {
        c[i][j] = 0;
    }
    else
    {
        // 第二种情况
        if (s1[i - 1] == s2[j - 1])
        {
            c[i][j] = MemoizedLcsLengthAux(s1, s2, i - 1, j - 1) + 1;
            b[i][j] = 1;
        }
        // 第三种情况
        else
        {
            int p = MemoizedLcsLengthAux(s1, s2, i - 1, j);
            int q = MemoizedLcsLengthAux(s1, s2, i, j - 1);
            // 若是上面的比较大
            if (p >= q)
            {
                c[i][j] = p;
                b[i][j] = 2;
            }
            // 若是左面的比较大
            else
            {
                c[i][j] = q;
                b[i][j] = 3;
            }
        }
    }
    return c[i][j];
}

int MemoizedLcsLength(string &s1, string &s2)
```

```
{
    memset(c, -1, sizeof c);

    return MemoizedLcsLengthAux(s1, s2, s1.length(), s2.length());
}

// 自底向上
void LcsLength(string &s1, string &s2)
{
    // 获取字符串长度
    int m = s1.length();
    int n = s2.length();
    // 从左上角开始遍历，自上而下
    for (int i = 0; i < m; i ++)
    {
        // 从左往右
        for (int j = 0; j < n; j ++)
        {
            // 若当前位置字符相同，则为左上角+1
            if (s1[i] == s2[j])
            {
                c[i + 1][j + 1] = c[i][j] + 1;
                b[i + 1][j + 1] = 1;
            }
            // 上面的比较大
            else if (c[i][j + 1] >= c[i + 1][j])
            {
                c[i + 1][j + 1] = c[i][j + 1];
                b[i + 1][j + 1] = 2;
            }
            // 左面的比较大
            else
            {
                c[i + 1][j + 1] = c[i + 1][j];
                b[i + 1][j + 1] = 3;
            }
        }
    }
}
// 打印函数，s务必得是第一个字符串
void PrintLcs(string &s, int i, int j)
{
    if (i == 0 || j == 0)
    {
        return;
    }
    if (b[i][j] == 1)
    {
        PrintLcs(s, i - 1, j - 1);
```

```
        cout << s[i - 1];
    }
    else if (b[i][j] == 2)
    {
        PrintLcs(s, i - 1, j);
    }
    else
    {
        PrintLcs(s, i, j -1);
    }
}

int main(int argc, char const *argv[])
{
    // 二维表初始化
    memset(c, 0, sizeof c);
    memset(b, 0, sizeof b);

    string s1 = "ABCBDAB";
    string s2 = "BDCABA";
    // 自底向上
    cout << "自底向上：" << endl;
    LcsLength(s1, s2);
    cout << "长度：" << c[s1.length()][s2.length()] << endl;
    PrintLcs(s1, s1.length(), s2.length());
    cout << endl;
    // 带备忘的自顶向下
    cout << "带备忘的自顶向下：" << endl;
    cout << "长度" << MemoizedLcsLength(s1, s2) << endl;
    PrintLcs(s1, s1.length(), s2.length());
    cout << endl;

    return 0;
}
```

2. 最优二叉搜索树

```
```

五、运行结果截图

1. 最长公共子序列

测试样例：

- s1 = "ABCBDAB"
- s2 = "BDCABA"

```
(base) ~/code/ Introduction to Algorithms/15_dynamic_programming cd "/Users/lsn/code/ Introduction to Algorithms/15_dynamic_programming/" && g++ longest_common_subse
quence.cpp -o longest_common_subsequence && "/Users/lsn/code/ Introduction to Algorithms/15_dynamic_programming/"longest_common_subsequence
自底向上：
长度： 4
BCBA
带备忘的自顶向下：
长度4
BCBA
```

2. 最优二叉搜索树

测试样例：

- n = 4
- p = {3, 3, 1, 1}
- q = {2, 3, 1, 1, 1}

```
(base) ~/code/ Introduction to Algorithms/15_dynamic_programming cd "/Users/lsn/code/ Introduction to Algorithms/15_dynamic_programming/" && g++ optimal_binary_searc
h_trees.cpp -o optimal_binary_search_trees && "/Users/lsn/code/ Introduction to Algorithms/15_dynamic_programming/"optimal_binary_search_trees
k2为根
k1为 k2的左孩子
d0为 k1的左孩子
d1为 k1的右孩子
k3为 k2的右孩子
d2为 k3的左孩子
k4为 k3的右孩子
d3为 k4的左孩子
d4为 k4的右孩子
```

六、问题与总结

本次实验完成了两个代码，分别是最长公共子序列和最优二叉搜索树。代码实现上总体都不难，算法理解上也有书和大量网上资料的帮助，因此也没有什么困难。

由于最长公共子序列的实验我用了两种方法，我发现 **带备忘的自顶向下** 的方法不会像 **自底向上** 的方法一样遍历整个二维表，只会遍历部分必要的二维表。因此可能在数据量较大的时候，它会有优势；但同时也要看考虑递归算法的堆栈消耗。所以总的来说，这两种方法解决同一个问题各有优势。