《算法设计与分析》习题6作业

学　号：1004191211　　　　　　姓　名：郎文鹏

日　期：2021/10/28　　　　　　得　分：

问题五：给定模式“$grammer$”和文本“$grameer$”，写出动态规划法求解K-近似匹配的过程。

【原理】

设样本为$patt = p_1p_2...p_m$，文本为$tex = t_1t_2...t_n$，对于一个非整数K，样本$patt$和文本tex的K个差别匹配无疑是下列三种之一：

1. 修改：$patt$和tex中对应字符不同。
2. 删去：$patt$中含有一个未出现在$patt$中的字符。
3. 插入：tex中不含有出现在$patt$中的一个字符。

容易证明该问题满足最优解原理，样本$patt$在文本tex的某一位置上有最优解则$patt$的任意一个子串在文本tex的对应也是最优解。定义问题为$dp[m][n]$表示样本$patt$前m个字符与文本tex前n个字符的最小差别数，那么其子问题的边界显然满足：

$$\begin{cases} dp[0][j] = j & 1 \le j \le n \\ dp[i][0] = i & 1 \le i \le m \end{cases}$$

考虑前缀子串$p_1...p_i$与文本前缀子串$t_1...t_j$之间的最小差别数$dp[i][j]$，则此时$dp[i-1][j-1]$、$dp[i-1][j]$与$dp[i][j-1]$的答案已经提前求出，对于p_i和$\mathrm{t_j}$两个字符则有三种对齐方式：

1. 字符p_i和t_j相对应，此时若是$\mathrm{p_i} = \mathrm{t_j}$则总差别数为$dp[i-1][j-1]$，反之则需要进行一次修改总差别数为$dp[i-1][j-1]+1$
2. 字符p_i为多余，即字符p_i肯定被舍去总差别数有$dp[i-1][j]+1$
3. 字符t_j为多余，即字符t_j肯定被舍去总差别数有$dp[i][j-1]+1$

由此得到所有情况的动态规划转移方程式：

$$
dp[i][j] = \begin{cases} dp[0][j] = j & i = 0, j \geq 0 \\ dp[i][0] = j & i \geq 0, j = 0 \\ min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) & i > 0, j > 0, p_i = t_i \\ min(dp[i-1][j-1] + 1, dp[i-1][j], dp[i][j-1]) & i > 0, j > 0, p_i \neq t_i \end{cases}
$$

【变式】

我们将字符比较的两种情况进行标记，分析每种状态下转移的过程：先看对应字符是否相同，然后再从左上方的三个状态选择最小值作为最优解转移过来作为当前状态的最优解。

【调试】

为了看清每个状态的最优解，我们输出其求解过程的矩阵。

➢第一列为模式串，第一行为文本串用于我们对比每个状态的字符是否相同。

➢将字符相同的状态打上绿色背景标记，转移由方程第三个转移而来。

➢从最终答案状态向前回溯，分析出差别产生的位置与进行的操作。

修改

	tex	g	r	a	m	e	e	r
patt	0	1	2	3	4	5	6	7
g	1	0	1	2	3	4	5	6
r	2	1	0	1	2	3	4	5
a	3	2	1	0	1	2	3	4
m	4	3	2	1	0	1	2	3
m	5	4	3	2	1	1	2	3
e	6	5	4	3	2	1	1	2
r	7	6	5	4	3	2	2	1

图 1 K近似匹配统计表

可以看到两个串的前四个字符不用做任何修改，在第五个字符位置处出现不同的情况，进行了修改操作（用$patt$中的字符m替换tex中的字符e），进行一次该操作后后两个字符也对应不做任何修改。

【源码】

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 15;

int dp[maxn][maxn];
string patt = "grammer", tex = "grameer";

void match() {
    for (int j = 1; j <= tex.size(); ++j) dp[0][j] = j;
```

```
    for (int i = 1; i <= patt.size(); ++i) dp[i][0] = i;
    for (int j = 1; j <= tex.size(); ++j)
        for (int i = 1; i <= patt.size(); ++i) {
            if (patt[i - 1] == tex[j - 1])
                dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j] + 1,
dp[i][j - 1] + 1});
            else
                dp[i][j] = min({dp[i - 1][j - 1] + 1, dp[i - 1][j] + 1,
dp[i][j - 1] + 1});
        }
}

void print() {
    cout << "    ";
    for (int i = 0; i < tex.size(); ++i) cout << tex[i] << ' ';
    cout << endl;
    for (int i = 0; i <= patt.size(); ++i) {
        if (i)
            cout << patt[i - 1] << ' ';
        else
            cout << "  ";
        for (int j = 0; j <= tex.size(); ++j)
            cout << dp[i][j] << ' ';
        cout << endl;
    }
}

signed main() {
    freopen("IO//out.txt", "w", stdout);
    match();
    print();
    cout << dp[patt.size()][tex.size()] << endl;
    return 0;
}
```

问题六：对于最优二叉查找树的动态规划算法，设计一个线性时间算法，从二维表 R 中生成最优二叉查找树。

【原理】

核心是依据最优二叉查找树满足动态规划的最优性原理。其算法中用到的平均比较次数矩阵C和根节点矩阵R同样满足最优性原理，我们通过查表将整体问题分解成更小的子问题，由于子问题同样满足最优性原理所以仍然可以看做上述

问题进行查表求解。

【复杂度问题】

每查以此表可以确定一个结点的位置，没有重复最终确定所有节点的位置即查表n次，所以复杂度为$O(n)$。

【伪源码】

```
void add_edge(int to, int from) {
    from->to;    //建立有向边
}

int dfs(int l, int r) {
    if (l == r) return l;           //到达叶节点
    if (l > r) return -1;           //越界访问
    int rt = R[l][r];              //查表确定当前区间根节点
    add_edge(dfs(l, rt - 1), rt);  //将左子区间根节点作为当前区间根节点的左孩子
    add_edge(dfs(rt + 1, r), rt);  //将右子区间根节点作为当前区间根节点的右孩子
    return rt;
}
```

【举例】

以教材中讲解的图 6.20 为例进行求解。根节点矩阵R如下图所示。

	0	1	2	3	4	
1		1	2	3	3	d=3
2			2	3	3	d=2
3				3	3	d=1
4					4	
5						

图 1 根节点矩阵R图

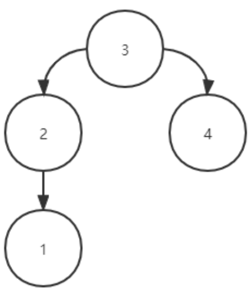

图 2 正确答案

经验证，算法输出结果与期望一致，时间复杂度为$O(n)$。

```
#include <bits/stdc++.h>
using namespace std;

int p = 0;
int root;
int table[5][5];
int G[5][2];

int build(int l, int r) {
    if (l == r) return table[l][l];
    if (l > r) return -1;
    int rt = table[l][r];
    memset(G, -1, sizeof G);
    G[rt][0] = build(l, rt - 1);
    G[rt][1] = build(rt + 1, r);
    return rt;
}

void visit(int rt) {    //输出前序遍历
    if (rt == -1) return;
    cout << rt << endl;
    visit(G[rt][0]);
    visit(G[rt][1]);
}

signed main() {
    freopen("IO//out.txt", "w", stdout);
    table[1][1] = 1;
    table[1][2] = table[2][2] = 2;
    table[1][3] = table[2][3] = table[3][3] = table[1][4] = table[2][4] = table[3][4] = 3;
    table[4][4] = 4;
    root = table[1][4];
    build(1, 4);
    visit(root);
    return 0;
}
/**
 * 3
 * 2
 * 1
 * 4
```

```
*/
```