算法设计与分析
最优二叉搜索树实验报告

编译环境：Microsoft Visual Studio 2019

Windows SDK **版本：**10.0

平台工具集：Visual Studio 2019（v142）

语言：C++

一、基本原理：

动态规划：

使用动态规划需要两个条件

① 具有优化子结构：

对于一棵最优二叉搜索树 T，其任意一棵子树 T1 一定是一棵最优二叉搜索树。可以用反证法证明，T 是一棵最优二叉搜索树，如果存在一棵子树 T1 不是最优二叉搜索树，那么将此子树替换为包含关键字相同但是形态不相同的另外一棵子树 T2，就很有可能使的整棵树 T 的期望搜索代价更低，这就不满足 T 是一棵最优二分搜索树的前提，因此 T 如果是一棵最优二叉搜索树，其任意一棵子树 T1 一定是一棵最优二叉搜索树

② 具有重叠子问题

根据上述的分析，可以知道最优二叉搜索树问题是存在重叠子问题的，因为如果我们要构造一棵最优二叉搜索树，必须优先构造其子树，构造其子树的时候要构造其子树的子树，如此下去，就存在了重叠的子问题。

因此最优二分搜索树问题是可以使用动态规划求解的。

二、问题描述：

最优二叉搜索树问题：

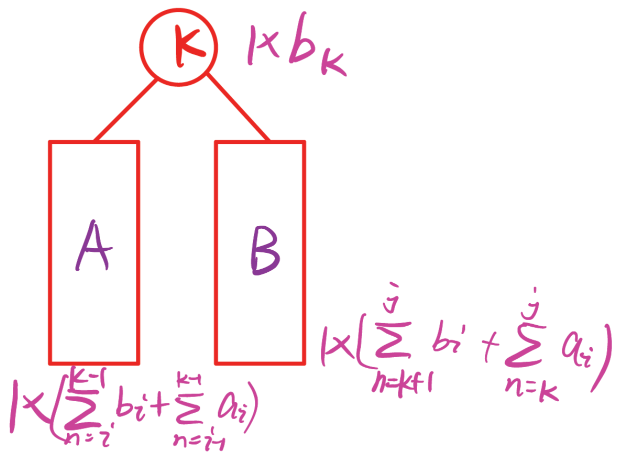
给定一个 n 个不同关键字已排序的序列 ：

K=<k1,k2,...,kn> (k1<k2<...<kn)，

我们希望用这些关键字构造一棵二叉搜索树。对每个关键字 ki，都有一个概率 bi 表示其搜索频率。有些要搜索的值可能不在 K 中，因此我们还有 n+1 个“伪关键字”d0,d1,d2...dn 表示不在 K 中的值。d0 表示所有小于 k1 的值， dn 表示所有大于 kn 的值，对 i=1,2,...,n−1，伪关键字 di 表示所有在 ki 和 ki+1 之间的值。对每个伪关键字 di，也都有一个概率 ai 表示对应的搜索频率。

三、算法分析与实现：

Ps：图和公式不好写，我在 ipad 上手写了部分，然后截图到了 word 中



A,B为两棵最优二叉搜索树，如果把A,B作为节点K的左右子树，平均路长多了 $P_K=b_K$，对A B的每个关键字和伪关键字路上都多了1，也就是A,B,K共多了

$$1\times\left(\sum_{n=i}^{k-1} b_i+\sum_{n=i-1}^{k-1} a_i\right)+1\times\left(\sum_{n=k+1}^{j} b_i+\sum_{n=k}^{j} a_i\right)+b_K$$

$$=\sum_{n=i}^{j} b_i+\sum_{n=i-1}^{j} a_i$$

$$=a_{i-1}+b_i+\cdots+b_j+a_j \quad (1\le i\le j\le n)$$

设为 w_{ij}

最优二叉搜索树 T(i, j)的平均路长为 m(i, j)，则所求最优解为：m(1, n)。根据最优子结构的性质可以建立计算 m(i, j)的递归式：

$$m(i,j)=w_{ij}+\min_{i\le k\le j}\{\underbrace{m(i,k-1)+m(k+1,j)}_{\text{左右子树原来的平均路长}}\}$$

以k为根节点时增加的路长 $i\le j$

对于 $j=i-1$ 的情况. eg.

$m(1,1)=\min(m(1,0)+m(2,1))+w_{1,1}$

显然 $m(1,1)=w_{1,1}$ ∴ $m(1,0)=m(2,1)=0$

同理可得 $m(i,i-1)=0 \quad (1\le i\le n)$

算法效率：

算法主要计算量在于计算

$\min\limits_{i\le k\le j}\{m(i,k-1)+m(k+1,j)\}$，时间复杂度为

$$\sum_{l=0}^{n-1}\sum_{i=1}^{n-l} O(l+1)=O(n^3)$$

但事实上述算法可以证明：

$\min\limits_{i\le k\le j}\{m(i,k-1)+m(k+1,j)\}=$

$\min\limits_{s[i][j-1]\le k\le s[i+1][j]}\{m(i,k-1)+m(k+1,j)\}$

因为可以证明

$Root(i,j-1)\le Root(i,j)\le Root(i+1,j)$

在上式中 $s[i][j]$ 就是 $Root(i,j)$，代表从 i 到 j 这棵最优二叉搜索树的根节点.

这样就可以实现在 0(n^2)的时间内计算。

四、测试样例与实验总结：

总结：

实用动态规划核心是判断该问题是否满足动态规划的两个条件：具有优化子结构，具有重叠子问题。满足动态规划的条件后，只需找出递推关系式，根据递

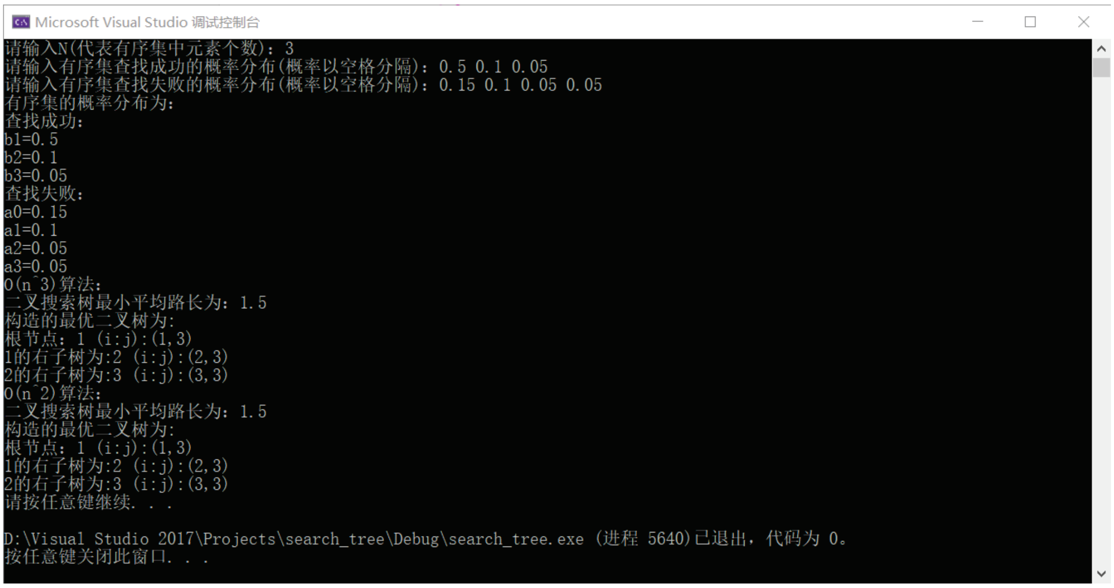
推关系式编写代码即可。

对于少量查找概率发生小的变化时最佳搜索树的调整算法，如果概率变大，则要将该节点上调，但此时不再满足二叉搜索树的要求，因此使用二叉搜索树的调整算法调整至满足二叉搜索树的要求即可，同理如果概率变小，则要将该节点下调，用同样的方法即可。

输入：

3
0.5 0.1 0.05
0.15 0.1 0.05 0.05

输出：



五、附录：

源码：

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

void OBST1(double a[], double b[], int n, double** m, int** s, double** w)//s[i][j]保存最优子树T(i,j)的根节点中元素
{
    //初始化构造无内部节点的情况
    for (int i = 0; i <= n; i++)
    {
        w[i + 1][i] = a[i];
        m[i + 1][i] = 0;
    }

    for (int r = 0; r < n; r++)//r代表起止下标的差
```

```
    {
        for (int i = 1; i <= n - r; i++)//i为起始元素下标
        {
            int j = i + r;//j为终止元素下标

            //构造T[i][j]，填写w[i][j],m[i][j],s[i][j]
            //首选i作为根，其左子树为空，右子树为节点
            w[i][j] = w[i][j - 1] + a[j] + b[j];
            m[i][j] = m[i + 1][j];
            s[i][j] = i;

            //不选i作为根，设k为其根，则k=i+1，……j
            //左子树为节点：i,i+1……k-1,右子树为节点：k+1,k+2,……j
            for (int k = i + 1; k <= j; k++)
            {
                double t = m[i][k - 1] + m[k + 1][j];

                if (t < m[i][j])
                {
                    m[i][j] = t;
                    s[i][j] = k;//根节点元素
                }
            }
            m[i][j] += w[i][j];
        }
    }
}

void OBST2(double a[], double b[], int n, double** m, int** s, double** w)
{
    //初始化构造无内部节点的情况
    for (int i = 0; i <= n; i++)
    {
        w[i + 1][i] = a[i];
        m[i + 1][i] = 0;
        s[i + 1][i] = 0;
    }

    for (int r = 0; r < n; r++)//r代表起止下标的差
    {
        for (int i = 1; i <= n - r; i++)//i为起始元素下标
        {
            int j = i + r;//j为终止元素下标
            int i1 = s[i][j - 1] > i ? s[i][j - 1] : i;
```

```
                int j1 = s[i + 1][j] > i ? s[i + 1][j] : j;

                //构造T[i][j]，填写w[i][j],m[i][j],s[i][j]
                //首选i作为根，其左子树为空，右子树为节点
                w[i][j] = w[i][j - 1] + a[j] + b[j];
                m[i][j] = m[i][i1 - 1] + m[i1 + 1][j];
                s[i][j] = i1;

                //不选i作为根，设k为其根，则k=i+1，……j
                //左子树为节点：i,i+1……k-1,右子树为节点：k+1,k+2,……j
                for (int k = i1 + 1; k <= j1; k++)
                {
                    double t = m[i][k - 1] + m[k + 1][j];

                    if (t <= m[i][j])
                    {
                        m[i][j] = t;
                        s[i][j] = k;//根节点元素
                    }
                }
                m[i][j] += w[i][j];
            }
        }
}

void traceback(int n, int i, int j, int** s, int f, char ch)//回溯输出最优二叉搜索树
{
    int k = s[i][j];
    if (k > 0)
    {
        if (f == 0)
        {
            //根
            cout << "根节点：" << k << " (i:j):(" << i << "," << j << ")" << endl;
        }
        else
        {
            //子树
            if (ch=='L')
            {
                cout << f <<"的左子树为:"<< k << " (i:j):(" << i << "," << j << ")"
<< endl;
            }
            else if (ch == 'R')
```

```
                {
                    cout << f << "的右子树为:" << k << " (i:j):(" << i << "," << j << ")"
<< endl;
                }
            }

            int t = k - 1;
            if (t >= i && t <= n)
            {
                //回溯左子树
                traceback(n, i, t, s, k, 'L');
            }
            t = k + 1;
            if (t <= j)
            {
                //回溯右子树
                traceback(n, t, j, s, k, 'R');
            }
        }
}

int main()
{
    int N;
    cout << "请输入N(代表有序集中元素个数): ";
    cin >> N;
    double* a = new double[N + 1];
    double* b = new double[N + 1];
    cout << "请输入有序集查找成功的概率分布(概率以空格分隔): ";
    b[0] = 0.00;
    for (int i = 1; i < N+1; i++)
    {
        cin >> b[i];
    }
    cout << "请输入有序集查找失败的概率分布(概率以空格分隔): ";
    for (int i = 0; i < N + 1; i++)
    {
        cin >> a[i];
    }
    cout << "有序集的概率分布为: " << endl;
    cout << "查找成功: " << endl;
    for (int i = 1; i < N+1; i++)
    {
        cout << "b" << i << "=" << b[i] << endl;
```

```
    }
    cout << "查找失败：" << endl;
    for (int i = 0; i < N + 1; i++)
    {
        cout << "a" << i << "=" << a[i] << endl;
    }
    double** m = new double* [N + 2];
    int** s = new int* [N + 2];
    double** w = new double* [N + 2];

    for (int i = 0;i < N + 2;i++)
    {
        m[i] = new double[N + 2];
        s[i] = new int[N + 2];
        w[i] = new double[N + 2];
    }

    OBST1(a, b, N, m, s, w);
    cout << "O(n^3)算法：" << endl;
    cout << "二叉搜索树最小平均路长为：" << m[1][N] << endl;
    cout << "构造的最优二叉树为:" << endl;
    traceback(N, 1, N, s, 0, '0');
    OBST2(a, b, N, m, s, w);
    cout << "O(n^2)算法：" << endl;
    cout << "二叉搜索树最小平均路长为：" << m[1][N] << endl;
    cout << "构造的最优二叉树为:" << endl;
    traceback(N, 1, N, s, 0, '0');

    for (int i = 0;i < N + 2;i++)
    {
        delete m[i];
        delete s[i];
        delete w[i];
    }
    delete[] m;
    delete[] s;
    delete[] w;
    system("pause");
    return 0;
}
```