

实验四：动态规划

计算矩阵连乘积

问题描述

在科学计算中经常要计算矩阵的乘积。矩阵A和B可乘的条件是矩阵A的列数等于矩阵B的行数。若A是一个 $p \times q$ 的矩阵，B是一个 $q \times r$ 的矩阵，则其乘积 $C=AB$ 是一个 $p \times r$ 的矩阵。由该公式知计算 $C=AB$ 总共需要 pqr 次的数乘。其标准计算公式为：

$$C_{ij} = \sum_{k=1}^q A_{ik} B_{kj} (1 \leq i \leq p, 1 \leq j \leq r)$$

现在问题是现在的问题是，给定 n 个矩阵 $\{A_1, A_2, \dots, A_n\}$ 。其中 A_i 与 A_{i+1} 是可乘的， $i=1, 2, \dots, n-1$ 。要求计算出这 n 个矩阵的连乘积 $A_1 A_2 \dots A_n$ 。

问题分析与算法思想

设二维数组 $m[N][N]$ 表示当前矩阵的连乘次数，一维数组 $p[N]$ 表示各矩阵的维度（其中 $p[0]$ 表示第一个矩阵的行数， $p[i]$ 表示第 i 个矩阵的列数），则得到以下递推公式：

$$m[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

将 m 数组的对角线初始化为0，然后依次计算第 i 个矩阵与第 $i+1$ 个矩阵到最后一个矩阵连乘的最优解情况：依次在 $r-1$ 个分隔位置中依次检测最优分隔点：对于每个分隔点，变换依次分隔位置，再进行逐一测试，如果有更有的分隔点，就替换掉当前的分隔点。

由此，输出 $m[1][n]$ ，得到最少的连乘计算次数；记录间隔位置，可以输出计算连乘的顺序，即最佳添加括号的方式

算法设计与代码实现

```
void MatrixChain(int n)
{
    int r, i, j, k;
    for (i = 0; i <= n; i++)                // 初始化对角线
    {
        m[i][i] = 0;
    }
    for (r = 2; r <= n; r++)                // r 个矩阵连乘
    {
        for (i = 1; i <= n - r + 1; i++)    // 依次计算每r个矩阵相连乘的最优解情况
```

```

{
    j = i + r - 1;
    m[i][j] = m[i][i] + m[i + 1][j] + p[i - 1] * p[i] * p[j];
    s[i][j] = i; // 分隔位置
    for (k = i + 1; k < j; k++) // 变换分隔位置，逐一测试
    {
        int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
        if (t < m[i][j]) // 如果变换后的位置更优，则替换原来的分隔方法
        {
            m[i][j] = t;
            s[i][j] = k;
        }
    }
}
}
}
}
}

```

算法演示：

```

C:\Users\HP\Desktop\AlgorithmExperiments\program\matrixmultiple.exe
请输入矩阵的数目：6
请输入各个矩阵的维度(相邻维度只需输入一个即可):30 35 15 5 10 20 25
最佳添加括号的方式为：((p[1](p[2]p[3]))(p[4]p[5])p[6]))
最小计算量的值为：15125

-----
Process exited after 21.45 seconds with return value 0
请按任意键继续. . .

```

```

C:\Users\HP\Desktop\AlgorithmExperiments\program\matrixmultiple.exe
请输入矩阵的数目：5
请输入各个矩阵的维度(相邻维度只需输入一个即可):10 20 30 40 20 15
最佳添加括号的方式为：(((p[1]p[2])p[3])p[4])p[5])
最小计算量的值为：29000

-----
Process exited after 23.72 seconds with return value 0
请按任意键继续. . .

```

算法讨论

通过动态规划来确定矩阵连乘顺序的时间复杂度是 $O(n^3)$ ，在大规模矩阵连乘是，选择最优的连乘顺序，可以大幅度减少计算量，提高计算效率

防卫导弹

问题描述

一种新型的防卫导弹可截击多个攻击导弹。它可以向前飞行，也可以用很快的速度向下飞行，可以毫无损伤地截击进攻导弹，但不可以向后或向上飞行。但有一个缺点，尽管它发射时可以达到任意高度，但它只能截击比它上次截击导弹时所处高度低或者高度相同的导弹。现对这种新型防卫导弹进行测试，在每一次测试中，发射一系列的测试导弹（这些导弹发射的间隔时间固定，飞行速度相同），该防卫导弹所能获得的信息包括各进攻导弹的高度，以及它们发射次序。现要求编一程序，求在每次测试中，该防卫导弹最多能截击的进攻导弹数量，一个导弹能被截击应满足下列两个条件之一：

1. 它是该次测试中第一个被防卫导弹截击的导弹
2. 它是在上一次被截击导弹的发射后发射，且高度不大于上一次被截击导弹的高度的导弹

输入数据：第一行是一个整数 n ，以后的 n 各有一个整数表示导弹的高度

输出数据：截击导弹的最大数目

问题分析与算法思想

对于本题的拦截导弹问题，因为一个导弹能被截击应满足高度不大于上一次被截击导弹的高度的导弹可以将其抽象成求一个最长不上升子序列的问题。

以下是最长不上升子序列的一个算法分析：

1. 定义状态：我们可以使用一个数组 dp 来表示最长不上升子序列的长度。 $dp[i]$ 表示以第 i 个元素为结尾的最长不上升子序列的长度。
2. 初始化：将 dp 数组的所有元素初始化为1，因为每个单独的元素都可以视为一个长度为1的非递增子序列。
3. 状态转移：对于每个位置 i （从1到 $n-1$ ），我们需要考虑所有在 i 之前的位置 j （从0到 $i-1$ ）。如果 $nums[i] \leq nums[j]$ ，则可以将元素 i 添加到以 j 为结尾的非递增子序列中，从而得到以 i 为结尾的非递增子序列。因此，我们可以更新 $dp[i] = \max(dp[i], dp[j] + 1)$ 。
4. 找到最大值：遍历整个 dp 数组，找到其中的最大值，即为最长不上升子序列的长度。

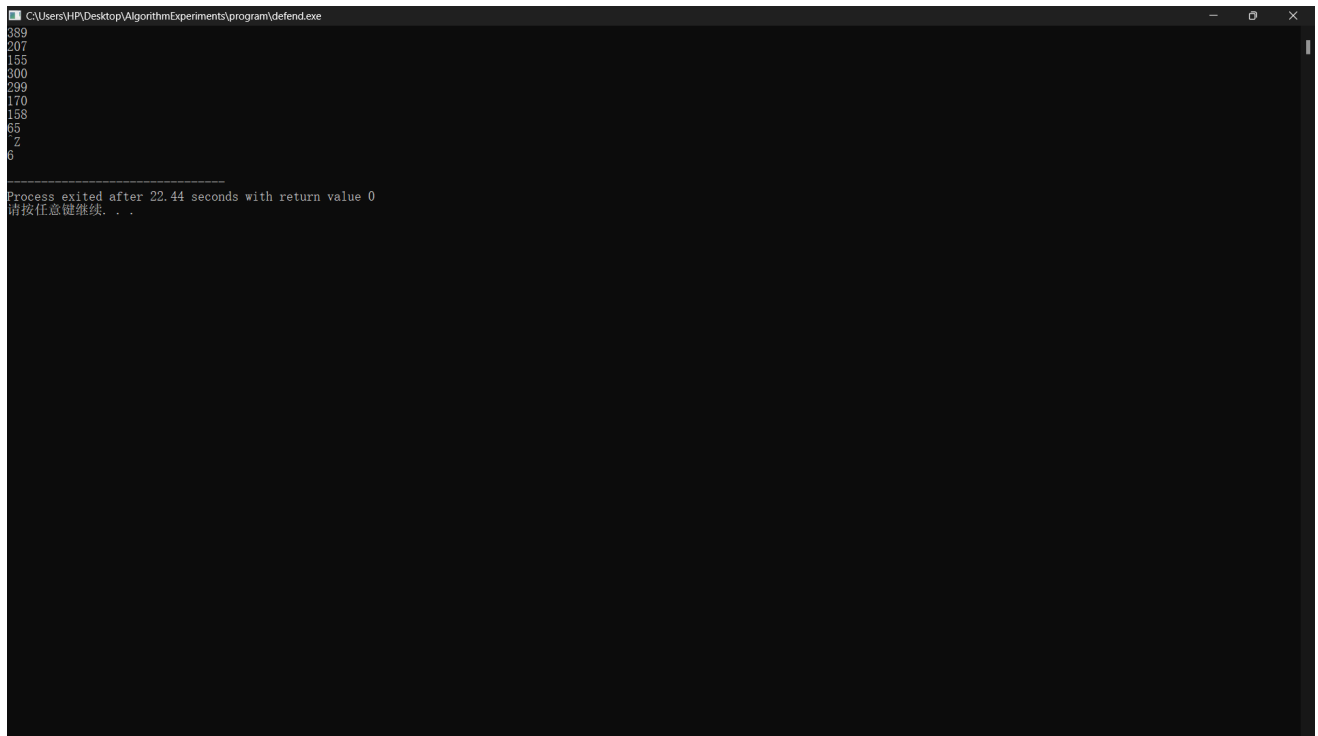
算法中的一些细节：

- 状态表示 $f[i]$ ：
 - 集合：所有以第 i 个数结尾的不升子序列；
 - 属性： Max
- 状态计算：集合划分—— $f[i]$
 - 划分依据：最后一个不同的点
 - 以上一个数的位置进行划分
 - 若 $a_i \leq a_j$ ，则 $f[i] = \max\{f[j] + 1\} (j = 0, 1, 2, \dots, i-1)$

算法设计与代码实现

```
void FindMissileNum(int n)
{
    int res = 0, cnt = 0;
    for (int i = 0; i < n; i++)
    {
        f[i] = 1;
        for (int j = 0; j < i; j++)
            if (h[i] <= h[j])
                f[i] = max(f[i], f[j] + 1);
        res = max(res, f[i]);
    }
    cout<<res<<endl;
}
```

算法演示：



```
C:\Users\HP\Desktop\AlgorithmExperiments\program\defend.exe
389
207
155
300
299
170
158
65
2
6

Process exited after 22.44 seconds with return value 0
请按任意键继续...
```

算法讨论

算法的时间复杂度是 $O(n^2)$ ，可以对遍历进行优化，进行树状数组优化或者二分优化，可以将时间复杂度优化到 $O(n\log n)$ 。

皇宫看守

问题描述

太平王世子事件后，陆小凤成了皇上特聘的御前一品侍卫。皇宫以午门为起点，直到后宫嫔妃们的寝宫，呈一棵树的形状；某些宫殿间可以互相望见。大内保卫森严，三步一岗，五步一哨，每个宫殿都要有人全天候看守，在不同的宫殿安排看守所需的费用不同。可是陆小凤手上的经费不足，无论如何也没法在每个宫殿都安置留守侍卫。

请你编程计算帮助陆小凤布置侍卫，在看守全部宫殿的前提下，使得花费的经费最少。

输入数据：数据表示一棵树，描述如下：

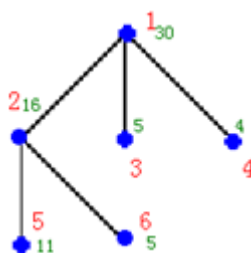
第1行 n ，表示树中结点的数目。

第2行至第 $n+1$ 行，每行描述每个宫殿结点信息，依次为：该宫殿结点标号 i ($0 < i \leq n$)，在该宫殿安置侍卫所需的经费 k ，该边的儿子数 m ，接下来 m 个数，分别是这个节点的 m 个儿子的标号 r_1, r_2, \dots, r_m 。

对于一个 n ($0 < n \leq 1500$) 个结点的树，结点标号在1到 n 之间，且标号不重复。

输出数据：输出到output.txt文件中。输出文件仅包含一个数，为所求的最少的经费。

如下示例：



sample input

```
6
1 30 3 2 3 4
2 16 2 5 6
3 5 0
4 4 0
5 11 0
6 5 0
```

sample output

```
25
```

问题分析与算法思想

这道题目是求权值最小的点支配集，要求图中每个点都能被观察到，因此有下列情况：

- 父节点**放置**哨兵，所有子节点的哨兵都可放可不放
- 父节点**不设置**哨兵，**至少有一个**子节点需要放置哨兵
- 父结点**不设置**哨兵，但其**父节点**设置哨兵观察，则子节点哨兵**可放可不放**

总结上述情况可以得到每个节点总共有三种情况：

- 被父节点看守
- 被子节点看守
- 被节点自身看守

将上述三种情况分别编为0, 1, 2

建立**状态转移函数** $f[i][3]$ ，其中：

1. $f[i][0]$ 表示第 i 个节点由父节点看守下的最小代价
2. $f[i][1]$ 表示第 i 个节点由子节点看守下的最小代价
3. $f[i][2]$ 表示第 i 个节点由自身看守下的最小代价

转移关系：

1. $f[i][0] += \min\{f[j][1], f[j][2]\}$
2. $f[i][1] = \min\{f[i][1], \text{sum} - \min(f[j][1], f[j][2]) + f[j][2]\}$
3. $f[i][2] += \min\{\min(f[j][0], f[j][1]), f[j][2]\}$

从根节点开始DFS，然后遍历所有当前节点的所有子节点，进行递归。

算法设计与代码实现

```
void dfs(int u)
{
    f[u][2] = w[u];

    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];          // 遍历所有子节点
        dfs(j);
        f[u][0] += min(f[j][1], f[j][2]);
        f[u][2] += min(min(f[j][0], f[j][1]), f[j][2]);
    }

    f[u][1] = 1e9;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        // f[u][0]为所有子节点的摆放方案代价之和，减去 min(f[j][1], f[j][2]) 即是除了j节点其余节点的
        代价之和
        f[u][1] = min(f[u][1], f[j][2] + f[u][0] - min(f[j][1], f[j][2]));
    }
}
```

算法演示：

```
CA\Users\HP\Desktop\AlgorithmExperiments\program\defendpalace.exe
6
1 30 3 2 3 4
2 16 2 5 6
3 5 0
4 4 0
5 11 0
6 5 0
25
-----
Process exited after 32.64 seconds with return value 0
请按任意键继续. . .
```

算法讨论

本题是树形DP的应用，涉及到了父节点、子节点的邻接关系。算法遍历了树中的所有节点，时间复杂度是 $O(n)$

附录

计算矩阵连乘积源代码

```
#include<iostream>
using namespace std;
const int N = 100;
int p[N];      // 矩阵规模
int m[N][N];   // 最优解
int s[N][N];

void MatrixChain(int n)
{
    int r, i, j, k;
    for (i = 0; i <= n; i++)           // 初始化对角线
    {
        m[i][i] = 0;
    }
    for (r = 2; r <= n; r++)           // r 个矩阵连乘
    {
        for (i = 1; i <= n - r + 1; i++) // 依次计算每r个矩阵相连乘的最优解情况
        {
            j = i + r - 1;
            m[i][j] = m[i][i] + m[i + 1][j] + p[i - 1] * p[i] * p[j];
            s[i][j] = i;               // 分隔位置
        }
    }
}
```

```

        for (k = i + 1; k < j; k++)        // 变换分隔位置，逐一测试
        {
            int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (t < m[i][j])                // 如果变换后的位置更优，则替换原来的分隔方法
            {
                m[i][j] = t;
                s[i][j] = k;
            }
        }
    }
}

void print(int i, int j)                // 输出连乘顺序
{
    if (i == j)
    {
        cout << "p[" << i << "]";
        return;
    }
    cout << "(";
    print(i, s[i][j]);
    print(s[i][j] + 1, j);
    cout << ")";
}

int main()
{
    int n;                // n个矩阵
    cout << "请输入矩阵的数目: ";
    cin >> n;
    int i, j;
    cout << "请输入各个矩阵的维度(相邻维度只需输入一个即可):";
    for (i = 0; i <= n; i++)
    {
        cin >> p[i];
    }
    MatrixChain(n);
    cout << "最佳添加括号的方式为: ";
    print(1, n);
    cout << "\n最小计算量的值为: " << m[1][n] << endl;
    return 0;
}

```

防卫导弹源代码

```

#include <sstream>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1000;

```



```

int h[N], f[N], q[N];          // q数组记录开好的子序列结尾的数

void FindMissileNum(int n)
{
    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        f[i] = 1;
        for (int j = 0; j < i; j ++ )
            if (h[i] <= h[j])
                f[i] = max(f[i], f[j] + 1);
        res = max(res, f[i]);
    }
    cout<<res<<endl;
}

int main()
{
    int n=0;
    while (cin >> h[n]) n ++ ;
    FindMissileNum(n);
    return 0;
}

```

皇宫看守源代码

```

#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1510;

int n;
int h[N], w[N], e[N], ne[N], idx;
int f[N][3];
bool st[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

void dfs(int u)
{
    f[u][2] = w[u];

    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];          // 遍历所有子节点
        dfs(j);
        f[u][0] += min(f[j][1], f[j][2]);
    }
}

```

```

        f[u][2] += min(min(f[j][0], f[j][1]), f[j][2]);
    }

    f[u][1] = 1e9;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        // f[u][0]为所有子节点的摆放方案代价之和，减去 min(f[j][1], f[j][2]) 即是除了j节点其余节点的
        代价之和
        f[u][1] = min(f[u][1], f[j][2] + f[u][0] - min(f[j][1], f[j][2]));
    }
}

int main()
{
    cin >> n;

    memset(h, -1, sizeof h);
    for (int i = 1; i <= n; i++)
    {
        int id, cost, cnt;
        cin >> id >> cost >> cnt;
        w[id] = cost;           // 在点上记录花费
        while (cnt--)
        {
            int ver;
            cin >> ver;
            add(id, ver);
            st[ver] = true;     // 标记不是根节点
        }
    }

    int root = 1;
    while (st[root]) root++; // 找到根节点

    dfs(root);

    cout << min(f[root][1], f[root][2]) << endl;

    return 0;
}

```