

**《算法分析与技术》实验报告**

专业班级： 计科2105

学 号： 8202191123

学生姓名： 谭哲文

指导老师： 李 敏

2023年06月

**实验一 递归与分治**

**一、实验目的：**

理解递归算法的思想和递归程序的执行过程，并能熟练编写递归程序。

掌握分治算法的思想，对给定的问题能设计出分治算法予以解决。

实验预习内容

编程实现讲过的例题：二分搜索、合并排序、快速排序。

对本实验中的问题，设计出算法并编程实现。

1. **算法定义：**

二分搜索采用递归调用的方法，可先将数组用快速排序排好，再进行搜索

1. **代码实现：**

**快速排序：**

void quickSort(int a[], int low, int high)

{

    if (low < high) // 判断是否满足排序条件，递归的终止条件

    {

        int i = low, j = high; // 把待排序数组元素的第一个和最后一个下标分别赋值给i,j，使用i,j进行排序；

        int x = a[low];        // 将待排序数组的第一个元素作为哨兵，将数组划分为大于哨兵以及小于哨兵的两部分

        while (i < j)

        {

            while (i < j && a[j] >= x)

                j--; // 从最右侧元素开始，如果比哨兵大，那么它的位置就正确，然后判断前一个元素，直到不满足条件

            if (i < j)

                a[i++] = a[j]; // 把不满足位次条件的那个元素值赋值给第一个元素，（也即是哨兵元素，此时哨兵已经保存在x中，不会丢失）并把i的加1

            while (i < j && a[i] <= x)

                i++; // 换成左侧下标为i的元素开始与哨兵比较大小，比其小，那么它所处的位置就正确，然后判断后一个，直到不满足条件

            if (i < j)

                a[j--] = a[i]; // 把不满足位次条件的那个元素值赋值给下标为j的元素，（下标为j的元素已经保存到前面，不会丢失）并把j的加1

        }

        a[i] = x;                 // 完成一次排序，把哨兵赋值到下标为i的位置，即前面的都比它小，后面的都比它大

        quickSort(a, low, i - 1); // 递归进行哨兵前后两部分元素排序 ， low,high的值不发生变化，i处于中间

        quickSort(a, i + 1, high);

    }

}

**递归搜索：**

int binarySearch(int a[], int x, int left, int right)

{

    int mid = (left + right) / 2;

    if (a[mid] == x)

        return mid ;

    else if (left == right)

        return -1; // 表示没有找到

    else if (a[mid] < x)

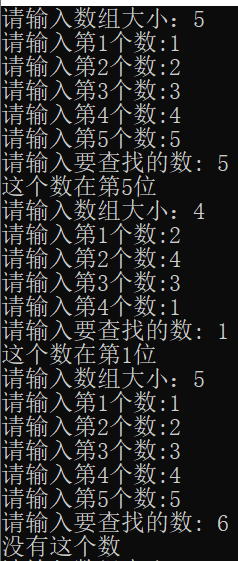
        binarySearch(a, x, mid + 1, right);

    else if (a[mid > x])

        binarySearch(a, x, left, mid - 1);

}

**运行截图：**



1. **总结：**

**二分查找需要在一个已排序的结构中查找。**

**实验四 动态规划**

**一、实验目的：**

理解动态规划的基本思想，理解动态规划算法的两个基本要素最优子结构性质和子问题的重叠性质。熟练掌握典型的动态规划问题。掌握动态规划思想分析问题的一般方法，对较简单的问题能正确分析，设计出动态规划算法，并能快速编程实现。

**实验 4.1 最长公共子序列**

**（1）实验内容**

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说，若给定序列

X=<x1, x2,…, xm>，则另一序列 Z=<z1, z2,…, zk>是 X 的子序列是指存在一个严格递增

的下标序列 <i1, i2,…, ik>，使得对于所有 j=1,2,…,k 有

易证最长公共子序列问题也有最优子结构性质

设序列 X=<x1, x2, …, xm>和 Y=<y1, y2, …, yn>的一个最长公共子序列 Z=<z1, z2, …,

zk>，则：

i. 若 xm=yn，则 zk=xm=yn 且 Zk-1 是 Xm-1 和 Yn-1 的最长公共子序列；

ii. 若 xm≠yn 且 zk≠xm ，则 Z 是 Xm-1 和 Y 的最长公共子序列；

iii.若 xm≠yn 且 zk≠yn ，则 Z 是 X 和 Yn-1 的最长公共子序列。

其中 Xm-1=<x1, x2, …, xm-1>，Yn-1=<y1, y2, …, yn-1>，Zk-1=<z1, z2, …, zk-

1. 。所以最长公共子序列问题具有最优子结构性质。

**（2）问题描述分析与算法设计思想**

蛮力法求解最长公共子序列：

需要遍历出所有的可能，时间复杂度是O(n³)，太慢了

动态规划求解最长公共子序列：

分析规律：

设X=<x1,x2,x3,x4...,xm>，Y=<y1,y2,y3,y4...,yn>为两个序列，Z=<z1,z2,z3,z4...,zk>是他们的任意公共子序列

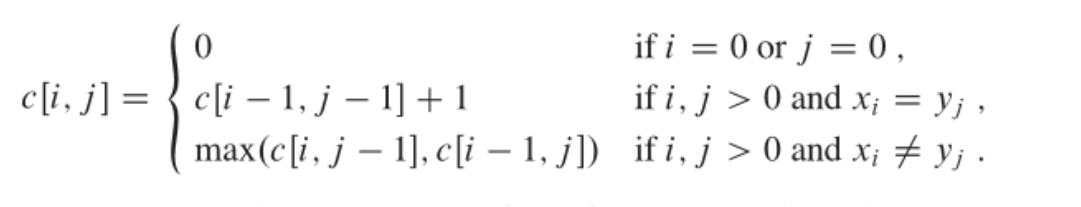
经过分析，我们可以知道：

1、如果xm = yn，则zk = xm = yn 且 Zk-1是Xm-1和Yn-1的一个LCS

2、如果xm != yn 且 zk != xm，则Z是Xm-1和Y的一个LCS

3、如果xm != yn 且 zk != yn，则Z是X和Yn-1的一个LCS

所以如果用一个二维数组c表示字符串X和Y中对应的前i，前j个字符的LCS的长度话，可以得到以下公式：



文字意思就是：

设

p1表示X的前 i-1 个字符和Y的前 j 个字符的LCS的长度

p2表示X的前 i 个字符和Y的前 j-1 个字符的LCS的长度

p表示X的前 i-1 个字符和Y的前 j-1 个字符的LCS的长度

p0表示X的前 i 个字符和Y的前 j 个字符的LCS的长度

如果X的第 i 个字符和Y的第 j 个字符相等，则p0 = p + 1

如果X的第 i 个字符和Y的第 j 个字符不相等，则p0 = max(p1,p2)

做法：

因此，我们只需要从c[0][0]开始填表，填到c[m-1][n-1]，所得到的c[m-1][n-1]就是LCS的长度

但是，我们怎么得到LCS本身而非LCS的长度呢？

也是用一个二维数组b来表示：

在对应字符相等的时候，用↖标记

在p1 >= p2的时候，用↑标记

在p1 < p2的时候，用←标记

若想得到LCS，则再遍历一次b数组就好了，从最后一个位置开始往前遍历：

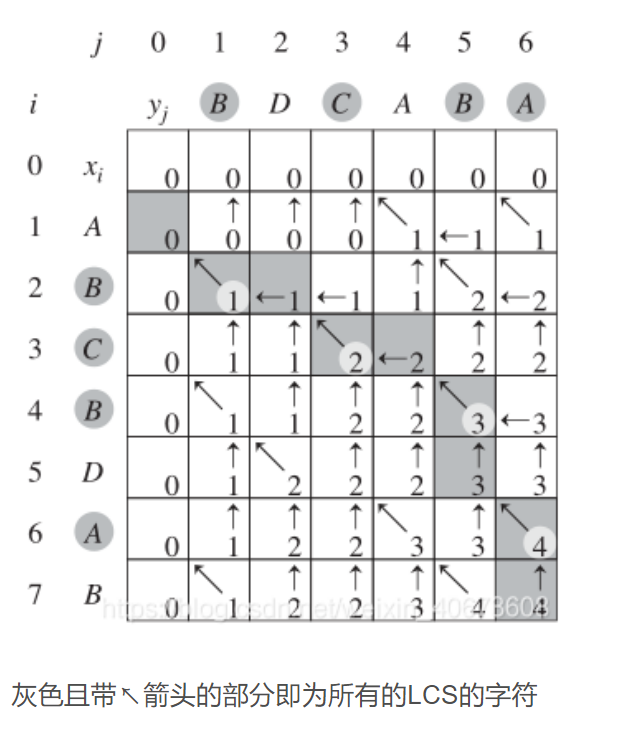
如果箭头是↖，则代表这个字符是LCS的一员，存下来后 i-- , j--

如果箭头是←，则代表这个字符不是LCS的一员，j--

如果箭头是↑ ，也代表这个字符不是LCS的一员，i--

如此直到i = 0或者j = 0时停止，最后存下来的字符就是所有的LCS字符

比如说求ABCBDAB和BDCABA的LCS:



**（3）代码实现**

#include <iostream>

#include <string>

#include <stack>

using namespace std;

void LCS(string s1, string s2)

{

    int m = s1.length() + 1;

    int n = s2.length() + 1;

    int\*\* c;

    int\*\* b;

    c = new int\* [m];

    b = new int\* [m];

    for (int i = 0; i < m; i++)

    {

        c[i] = new int[n];

        b[i] = new int[n];

        for (int j = 0; j < n; j++)

            b[i][j] = 0;

    }

    for (int i = 0; i < m; i++)

        c[i][0] = 0;

    for (int i = 0; i < n; i++)

        c[0][i] = 0;

    for (int i = 0; i < m - 1; i++)

    {

        for (int j = 0; j < n - 1; j++)

        {

            if (s1[i] == s2[j])

            {

                c[i + 1][j + 1] = c[i][j] + 1;

                b[i + 1][j + 1] = 1;          //1表示箭头为  左上

            }

            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;          //2表示箭头向  上

            }

            else

            {

                c[i + 1][j + 1] = c[i + 1][j];

                b[i + 1][j + 1] = 3;          //3表示箭头向  左

            }

        }

    }

    for (int i = 0; i < m; i++)                //输出c数组

    {

        for (int j = 0; j < n; j++)

        {

            cout << c[i][j] << ' ';

        }

        cout << endl;

    }

    stack<char> same;                   //存LCS字符

    stack<int> same1, same2;             //存LCS字符在字符串1和字符串2中对应的下标，方便显示出来

    for (int i = m - 1, j = n - 1; i >= 0 && j >= 0; )

    {

        if (b[i][j] == 1)

        {

            i--;

            j--;

            same.push(s1[i]);

            same1.push(i);

            same2.push(j);

        }

        else if (b[i][j] == 2)

            i--;

        else

            j--;

    }

    cout << s1 << endl;                     //输出字符串1

    for (int i = 0; i < m && !same1.empty(); i++)      //输出字符串1的标记

    {

        if (i == same1.top())

        {

            cout << 1;

            same1.pop();

        }

        else

            cout << ' ';

    }

    cout << endl << s2 << endl;                //输出字符串2

    for (int i = 0; i < n && !same2.empty(); i++)      //输出字符串2的标记

    {

        if (i == same2.top())

        {

            cout << 1;

            same2.pop();

        }

        else

            cout << ' ';

    }

    cout << endl << "最长公共子序列为：";

    while (!same.empty())

    {

        cout << same.top();

        same.pop();

    }

    cout << endl << "长度为：" << c[m - 1][n - 1] << endl;

    for (int i = 0; i < m; i++)

    {

        delete[] c[i];

        delete[] b[i];

    }

    delete[]c;

    delete[]b;

}

int main()

{

    string s1 = "ABCPDSFJGODIHJOFDIUSHGD";

    string s2 = "OSDIHGKODGHBLKSJBHKAGHI";

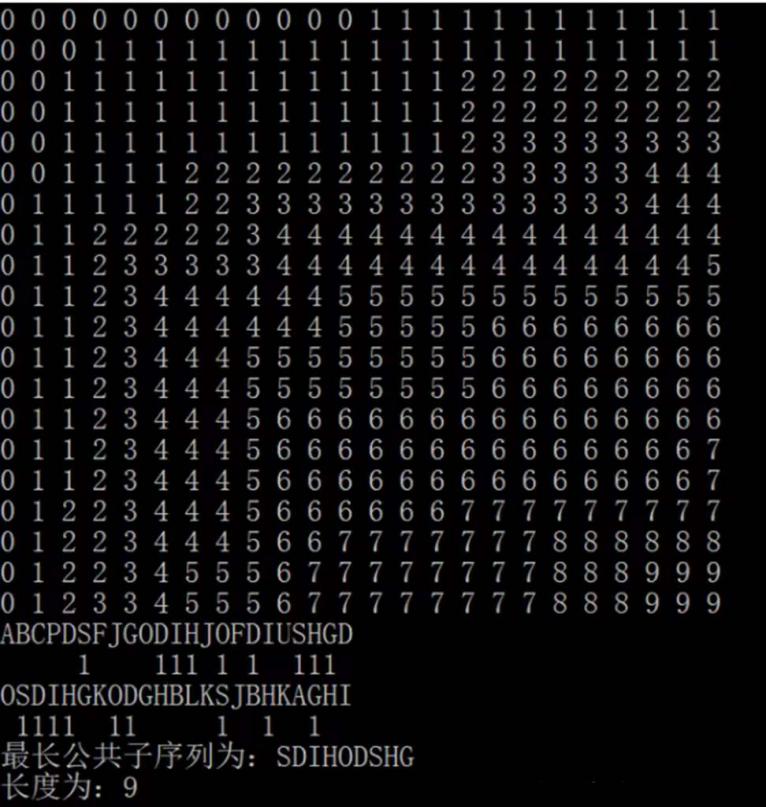
    LCS(s1, s2);

    system("pause");

    return 0;

}

运行结果：



1. **算法及问题分析**

### ****计算最优值：****

由于在所考虑的子问题空间中，总共只有θ(m\*n)个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法LCS\_LENGTH(X,Y)以序列X=<x1, x2, …, xm>和Y=<y1, y2, …, yn>作为输入。输出两个数组c[0..m ,0..n]和b[1..m ,1..n]。其中c[i,j]存储Xi与Yj的最长公共子序列的长度，b[i,j]记录指示c[i,j]的值是由哪一个子问题的解达到的，这在构造最长公共子序列时要用到。最后，X和Y的最长公共子序列的长度记录于c[m,n]中。

### ****时间复杂度：****

由于只需要填一个m行n列的二维数组，其中m代表第一个字符串长度，n代表第二个字符串长度

所以时间复杂度为O（m\*n）

**实验 4.9 皇宫看守**

**（1）实验内容**

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：输入数据由文件名为intput.txt的文本文件提供。输入文件中数据表示一棵树，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<=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r1，r2，...，rm。

对于一个n（0 < n <= 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。

**（2）问题描述分析与算法设计思想**

首先这道题的题意是：给定一棵树，要在一些节点上放置守卫，每个守卫可以看护当前节点以及与此节点连通的节点，在不同节点放置守卫的代价不同，如何选取节点使代价最小，这是个典型的树形DP问题，显然每个节点有放置守卫和不放置守卫两种，但是从计算的过程看，不放置守卫的状态由两种，一种是有其父节点上的守卫看护，一种是由其子节点的守卫看护，因此可将每个节点的看护情况分为三种：

1.该节点由父节点处放置的守卫看护

2.该节点由子节点处放置的守护看护

3.该节点由在该节点放置的守卫看护

下面考虑状态转移的过程，建立数组f[i][3]，其中

1. f[i][0]表示第i个节点由父节点处放置的守卫看护下的最小代价
2. f[i][1]表示第i个节点由子节点处放置的守卫看护下的最小代价
3. f[i][2]表示第i个节点由在该节点放置的守卫看护下的最小代价

那么可以写出转移关系：

1. f[i][0] += min(f[j][1], f[j][2]);
2. f[i][1] = min(f[i][1], 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]);

其中j为i的子节点，sum为所有子节点j的min(f[j][1], f[j][2])的和，1和3的意义很明显，2的意义代表，如果第i个节点由子节点守卫，那么所有子节点都不能由父节点守卫，并且每个子节点都得到了守卫，且至少有一个子节点处放置了守卫

**（3）代码实现**

#include <iostream>

#include <cstring>

#include <algorithm>

#include <cmath>

using namespace std;

const int N = 1510;

int n;

int h[N], e[N], ne[N], idx, w[N];

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];

    int sum = 0;

    for (int i = h[u]; ~i; i = ne[i])

    {

        int j = e[i];

        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]);

        sum += min(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][1] = min(f[u][1], sum - min(f[j][1], f[j][2]) + f[j][2]);

    }

}

int main()

{

    cout << "Sample Input:" << endl;

    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;

        while (cnt--)

        {

            int ver;

            cin >> ver;

            add(id, ver);

            st[ver] = true;

        }

    }

    int root = 1;

    while (st[root])

        root++;

    dfs(root);

    cout << "Sample Out:" << endl;

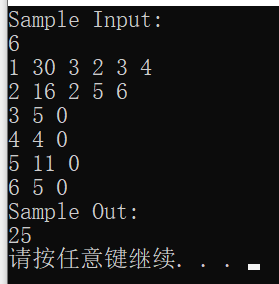
    cout << min(f[root][1], f[root][2]) << endl;

    system("pause");

    return 0;

}

运行结果：



**实验五 贪心算法**

1. **概述：**

**贪心算法：**

(１) 贪心性质：整体的最优解可通过一系列局部最优解达到，并且每次的选择可以依赖以前做出的选择，但不能依赖于以后的选择。

(２) 最优子结构：问题的整体最优解包含着它的子问题的最优解。

基本步骤：

(１) 分解：将原问题分解为若干相互独立的阶段。

(２) 解决：对于每一个阶段求局部的最优解。

(３) 合并：将各个阶段的解合并为原问题的解。

**实验5.1 背包问题**

**（1）实验内容**

A.问题描述

给定几组数据，利用贪心算法的思想，将物品装入背包并使得其价值最大。

B.实验步骤

① 计算每种物品单位重量的价值 Vi / Wi 。

② 依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。

③ 若将这种物品全部装入背包后，背包内的物品总重量未超过 C, 则选择单位重

量价值次高的物品并尽可能多地装入背包。

④ 依此策略一直地进行下去，直到背包装满为止。

**（2）问题描述分析与算法设计思想**

利用贪心算法求解背包问题，输出相应结果，并计算出程序运行所需要的时间。背包

问题简介如下：假设有 n 个物品，每个物品 i 的价值为 vi，⼤⼩为 si。包的容量为 S， 要求从这 n 个物品中挑选若⼲物品装进包中，在所有装进包中物品的⼤⼩于等于包容量 S 的

前提下，包中物品总价值最⼤。

（分数）背包问题与 0-1 背包问题的区别在于，完全背包问题可以只将物品的一部分

装入背包；那么对于背包问题思路很明确，优先将单位重量价值最高的物品放入，最终一

定能得到装满背包所能达到的最大的价值；

首先，给定几种数据。固定几个随机数组；然后，求解背包问题，使背包装满并且使得其价值最大。

贪心算法：

① 计算每种物品单位重量的价值 Vi / Wi 。

存储在数组中，便于调用计算。

② 依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。

通过排序算法，将存有物品单位重量的价值的数组进行排序，从值最大的

物品开始选择。

③ 若将这种物品全部装入背包后，背包内的物品总重量未超过 C, 则选择单位重

量价值次高的物品并尽可能多地装入背包。

判断比较此时包的容纳能力和物品重量，选择是否装入物品。

④ 依此策略一直地进行下去，直到背包装满为止。

循环以上三步，直到背包装满。

**（3）代码实现**

对于任一物品，拥有重量、价值（、编号（便于统计）），将其作为一个整体（排序）

① 计算每种物品单位重量的价值 Vi / Wi 。存储在数组中，便于调用计算。

② 依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。通过排序

算法，将存有物品单位重量的价值的数组进行排序，从值最大的物品开始选择。

③ 若将这种物品全部装入背包后，背包内的物品总重量未超过 C, 则选择单位重

量价值次高的物品并尽可能多地装入背包。判断比较此时包的容纳能力和物品重量，选

择是否装入物品。

④ 依此策略一直地进行下去，直到背包装满为止。

循环以上三步，直到背包装满。

**算法部分：**

#include <iostream>

using namespace std;

// 按照单位重量的价值量大小降序排列

void Sort(int n, float \*w, float \*v)

{

    int i, j;

    float temp1, temp2;

    for (i = 1; i <= n; i++)

        for (j = 1; j <= n - i; j++) // 冒泡排序

        {

            temp1 = v[j] / w[j];

            temp2 = v[j + 1] / w[j + 1];

            if (temp1 < temp2)

            {

                swap(w[j], w[j + 1]);

                swap(v[j], v[j + 1]);

            }

        }

}

int main()

{

    float w[101]; // 用来表示每个物品的重量

    float v[101]; // 用来表示每个物品的价值量

    float x[101]; // 表示最后放入背包的比例

    int n;        // 物品数

    float M;      // 背包最大容纳重量

    cout << "请输入物品数：" << endl;

    cin >> n;

    cout << "请输入最大容量：" << endl;

    cin >> M;

    // 依次输入每件物品的重量和价值量

    cout << "请依次输入每件物品的重量和价值量：" << endl;

    for (int i = 1; i <= n; i++)

        cin >> w[i] >> v[i];

    // 按照单位重量的价值量大小降序排列

    Sort(n, w, v);

    int i;

    for (i = 1; i <= n; i++)

        x[i] = 0; // 初始值，未装入背包，x[i]=0

    float c = M;  // 更新背包容纳量

    for (i = 1; i <= n; i++)

    {

        if (c < w[i])

            break; // 不能完全装下

        x[i] = 1;

        c = c - w[i];

    }

    if (i <= n)

        x[i] = c / w[i];

    // 输出

    double sum = 0;

    for (int i = 1; i <= n; i++)

    {

        cout << "重量为" << w[i] << "价值量为" << v[i] << "的物品"

             << "放入的比例为" << x[i] << endl;

        sum += x[i] \* v[i];

    }

    cout << "总价值为：" << endl

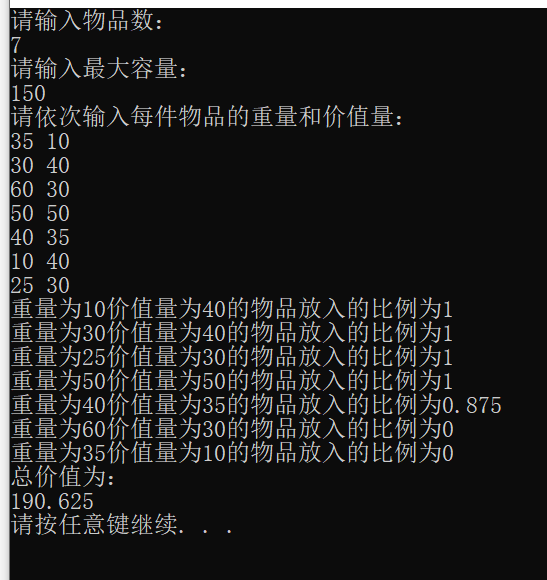
         << sum<<endl;

    system("pause");

    return 0;

}

**运行结果：**



搬桌子问题

**（1）实验内容**

某教学大楼一层有n个教室，从左到右依次编号为1、2、…、n。现在要把一些课桌从某些教室搬到另外一些教室，每张桌子都是从编号较小的教室搬到编号较大的教室，每一趟，都是从左到右走，搬完一张课桌后，可以继续从当前位置或往右走搬另一张桌子。输入数据：先输入n、m，然后紧接着m行输入这m张要搬课桌的起始教室和目标教室。输出数据：最少需要跑几趟。

**Sample Input**

10 5

1 3

3 9

4 6

6 10

7 8

**Sample Output**

3

分析：贪心算法，把课桌按起点从小到大排序，每次都是搬离当前位置最近的课桌。

**代码分析**

#include <stdio.h>

#include<iostream>

using namespace std;

int main()

{

    struct

    {

        int start;

        int end;

    } a[100];

    int i, j;

    int n, m, min, num, temp, used[100] = {0};

    cout<<"Sample Input:\n";

    cin>>m>>n;

    for (i = 0; i < n; i++)

        cin>>a[i].start>>a[i].end;

    min = 0;

    num = 0;

    while (num < n)

    {

        temp = 0;

        for (i = 0; i < n; i++)

            if (used[i] == 0 && a[i].start >= temp)

            {

                temp = a[i].end;

                used[i] = 1;

                num++;

            }

        min++;

    }

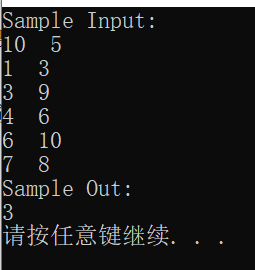
    cout<<"Sample Out:\n";

    cout<<min<<endl;

    system("pause");

}

**运行结果：**



**贪心算法及问题分析**

一般来说，贪心算法的证明围绕着：整个问题的最优解一定由在贪心策略中存在的子问题的最优解得来的。

对于背包问题中的3种贪心策略，都是无法成立(无法被证明)的，解释如下：

贪心策略：选取价值最大者。反例：

W=30

物品：A B C

重量：28 12 12

价值：30 20 20

根据策略，首先选取物品A，接下来就无法再选取了，可是，选取B、C则更好。

(2)贪心策略：选取重量最小。它的反例与第一种策略的反例差不多。

(3)贪心策略：选取单位重量价值最大的物品。反例：

W=30

物品：A B C

重量：28 20 10

价值：28 20 10

根据策略，三种物品单位重量价值一样，程序无法依据现有策略作出判断，如果选择A，则答案错误。