课程编号： B080201110

算法设计与分析
实验报告

姓　　名	薛旗	学　　号	20155362
班　　级	软信-1503	指　导　教　师	马毅
实　验　名　称	算法设计与分析		
开　设　学　期	2017-2018 第一学期		
开　设　时　间	第 1 周 —— 第 8 周		
报　告　日　期	2017.11.04		
评　定　成　绩		评　定　人	马毅
		评　定　日　期	

东北大学软件学院

实验一：分治法的应用

实验目的：

（1）理解分治法的思想

（2）掌握用分治法解决问题

实验内容：

中位数问题

★ **问题描述**

设 X[0 : n - 1]和 Y[0 : n - 1]为两个数组，每个数组中含有 n 个已排好序的数。找出 X 和 Y 的 2n 个数的中位数。

★ **编程任务**

利用分治策略试设计一个 O (log n)时间的算法求出这 2n 个数的中位数。

★ **数据输入**

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n(n<=200)，表示每个数组有 n 个数。接下来的两行分别是 X，Y 数组的元素。

★ **结果输出**

程序运行结束时，将计算出的中位数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3 5 15 18 3 14 21	14

实验思路：

1.对于数组 X[0 : *n* - 1]和 Y[0 : *n* - 1]，先分别找出 X 和 Y 的中位数 m1，m2。算法如下：

```
1  double median(int arr[], int n) {  //计算中位数
2      if (n % 2 == 0)
3          return (arr[n / 2] + arr[n / 2 - 1]) / 2.0;
4      else
5          return arr[n / 2];
6  }
```

对于已经排序好的数组：若 n 为奇数，则数组下标为(n-1)/2 处的数即为中位数，若 n 为偶数，则取(n-1)/2 向下取整和向上取整这两个位置的数的平均值作为中位数。

2.比较两个序列的中位数大小：

(1)若 m1 = m2,返回结果，该数为整个 2n 个数据的中位数；

(2)若 m1 < m2，则表明中位数在(m1，m2)之间,即整个 2n 的数的中位数在 X 数组的后一半和 Y 数组的前一半中；

(3)若 m1 > m2,则表明则中位数在(m2，m1)之间，即整个 2n 的数的中位数在 X 数组的前一半和 Y 数组的后一半中；

若出现 m1 = m2 的情况，算法结束，否则通过比较，分别减少两个序列的查找范围，确定查找的起止位置，继续查找，直至数组分割至左右两部数组只有一个数字的情况，求其平均值即为中位数。

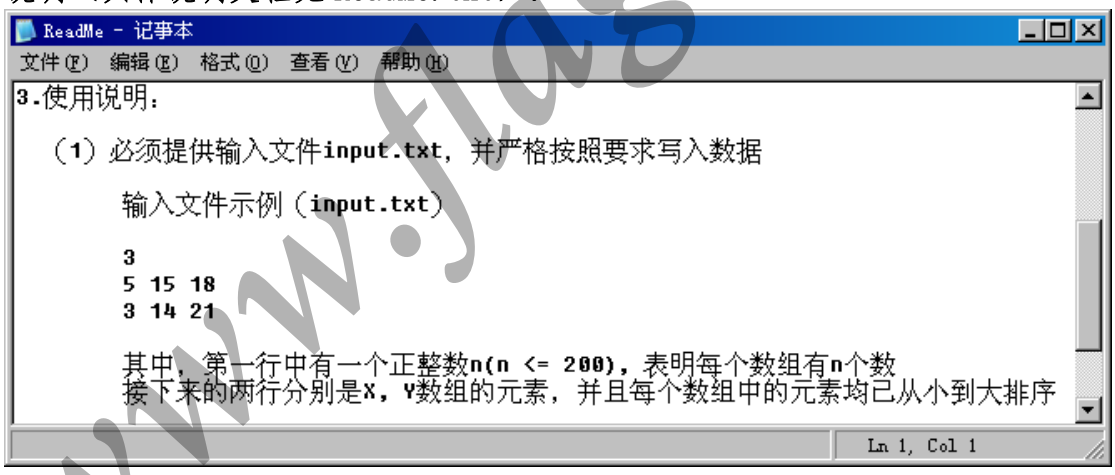
具体实现代码见附录。

实验步骤：

步骤	说明
1	创建 input.txt 文件并按实验要求写入数据
2	从 input.txt 文件读取预先写入的数据,将待计算数据存入数组 X[],Y[]
3	判断递归出口条件，若 n = 1 或 n = 2，结束递归调用并返回计算值，否则继续
4	计算数组 X 和数组 Y 的中位数 m1，m2
5	若 m1 = m2，返回结果即为所求
6	若 m1 < m2，则丢弃数组 X 的前一半和数组 Y 的后一半，若 n 为奇数，则 n = n - n /2，若 n 为偶数，则 n = n - n/2 + 1，递归
7	若 m1 > m2，则丢弃数组 X 的后一半和数组 Y 的前一半，若 n 为奇数，则 n = n - n /2，若 n 为偶数，则 n = n - n/2 + 1，递归
8	保存中位数结果到 output.txt 文件中

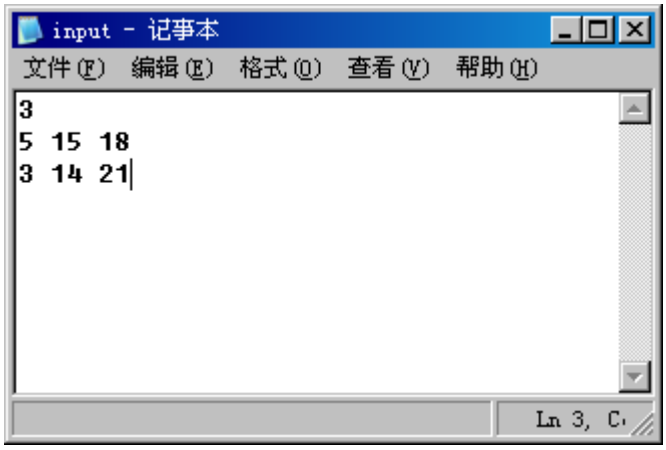
实验结果：

说明（具体说明文档见 ReadMe.txt）：



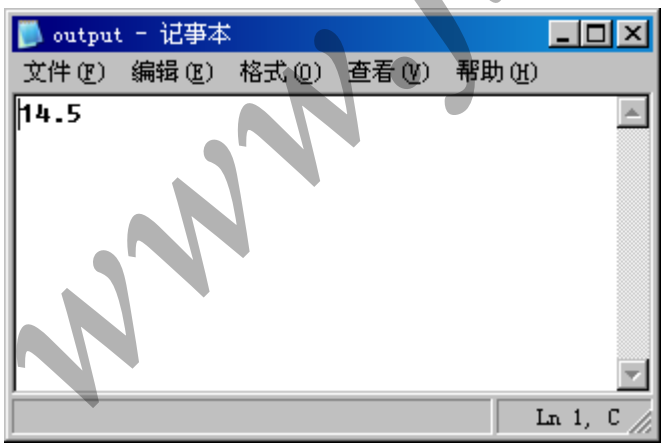
test1:

输入文件数据(input.txt):



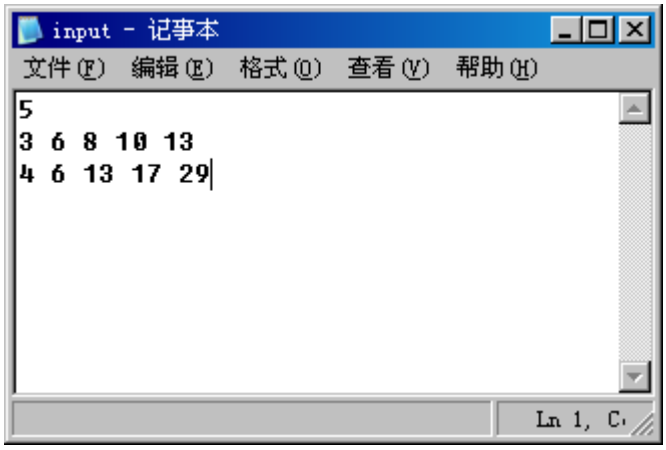
运行：

输出文件数据(output.txt)：

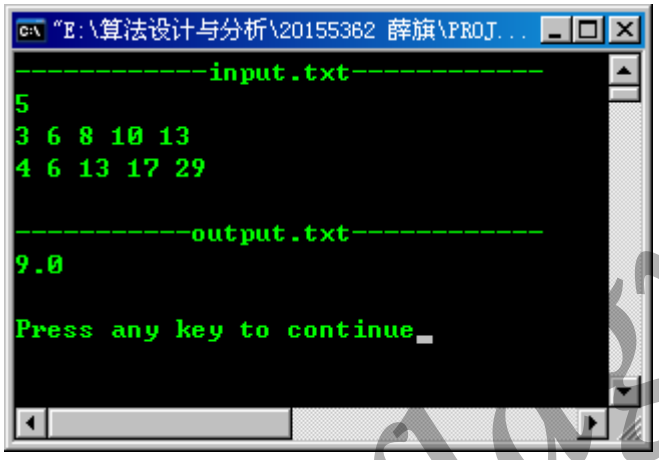


test2:

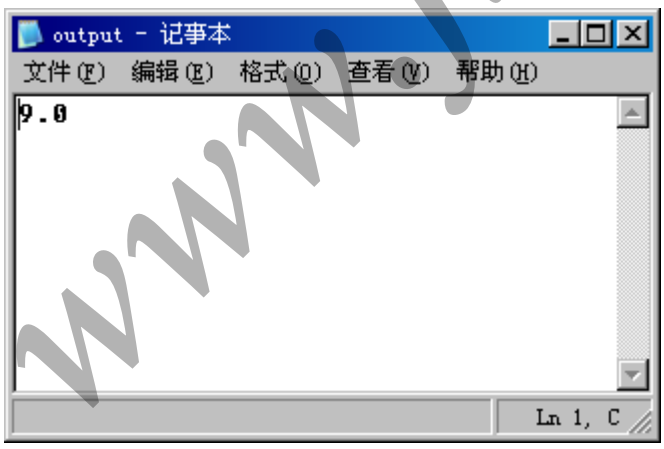
输入文件数据(input.txt)：



运行：



输出文件数据(output.txt)：



实验总结：

通过本次分治策略的应用实践，我加深了对分治法的理解。

分治法所能解决的问题一般具有以下几个特征：

1) 该问题的规模缩小到一定的程度就可以容易地解决

2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

3) 利用该问题分解出的子问题的解可以合并为该问题的解；

4) 该问题所分解出的各个子问题是相互独立的,即子问题之间不包含公共的子子问题。

分治法在每一层递归上都有三个步骤：
step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题
step3 合并：将各个子问题的解合并为原问题的解。

实验二：动态规划

实验目的：

（1）熟练掌握动态规划思想及教材中相关经典算法。
（2）掌握用动态规划解题的基本步骤，能够用动态规划解决一些问题。

实验内容：

找零钱问题

★ 问题描述

设有 n 种不同面值的硬币，各硬币的面值存于数组 T[1:n]中。现要用这些面值的硬币来找钱，可以实用的各种面值的硬币个数不限。当只用硬币面值 T[1],T[2],…,T[i]时，可找出钱数 j 的最少硬币个数记为 C(i,j)。若只用这些硬币面值,找不出钱数 j 时,记 C(i,j)=∞。

★ 编程任务

设计一个动态规划算法，对 1≤j≤L，计算出所有的 C(n,j)。算法中只允许实用一个长度为 L 的数组。用 L 和 n 作为变量来表示算法的计算时间复杂性

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n（n<=13），表示有 n 种硬币可选。接下来的一行是每种硬币的面值。由用户输入待找钱数 j。

★ 结果输出

程序运行结束时，将计算出的所需最少硬币个数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3	3
1 2 5	
9	

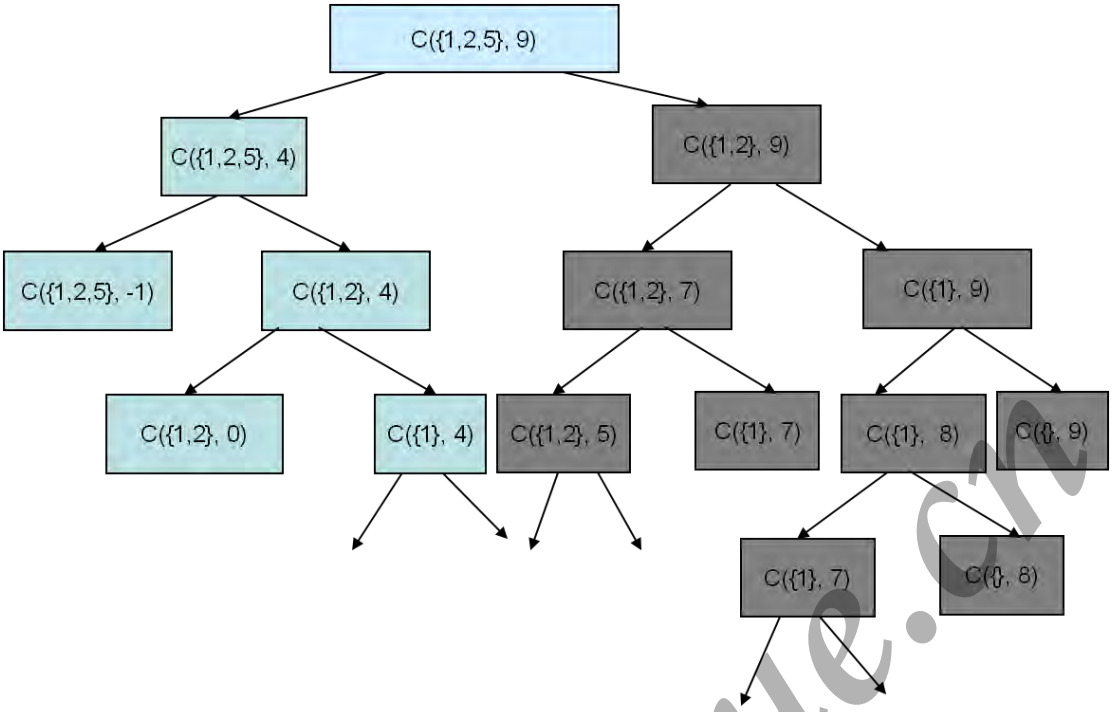
实验思路及步骤：

要求所需最少硬币，我们需要遍历所有可行的情况。而要求所有可行解，我们可以把一整套方案分成两组来解决,即解决方案不包含第 m 钟硬币或解决方案包含至少一个第 m 种硬币。假设对于 i = 1...N-1， 所需最少的硬币数 Count(i) 已知， 那么对于 N，所需的硬币数为 Min(Count(i) + Count(N-i)) ， i=1...N-1。这个问题是具有最优子结构性质的问

题。本题一个直观的方法是用递归计算。在本次实验中，遵循以上提到的递归结构，我写了一个递归算法如下：

```
1  int get_coins(int n, int T[], int j) {
2      int moneys = 0;
3      if (j <= 0)
4          return moneys;
5
6      if (n == 1) {
7          if (j % T[0] == 0)
8              moneys += j / T[0];
9          else
10             moneys += 999999999;
11         return moneys;
12     }
13     if (j >= 0 && j < T[n - 1]) {
14         n--;
15         moneys += get_coins(n, T, j);
16         return  moneys;
17     }
18     else {
19         int b = get_coins(n, T, j - T[n - 1]) + 1;
20         n--;
21         int a = get_coins(n, T, j);
22         if (a > b) {
23             moneys += b;
24             return moneys;
25         }
26         else {
27             moneys += a;
28             return moneys;
29         }
30     }
31 }
```

但是，上述函数反复计算相同的子问题，见如下递归树（以实验指导书示例为例）



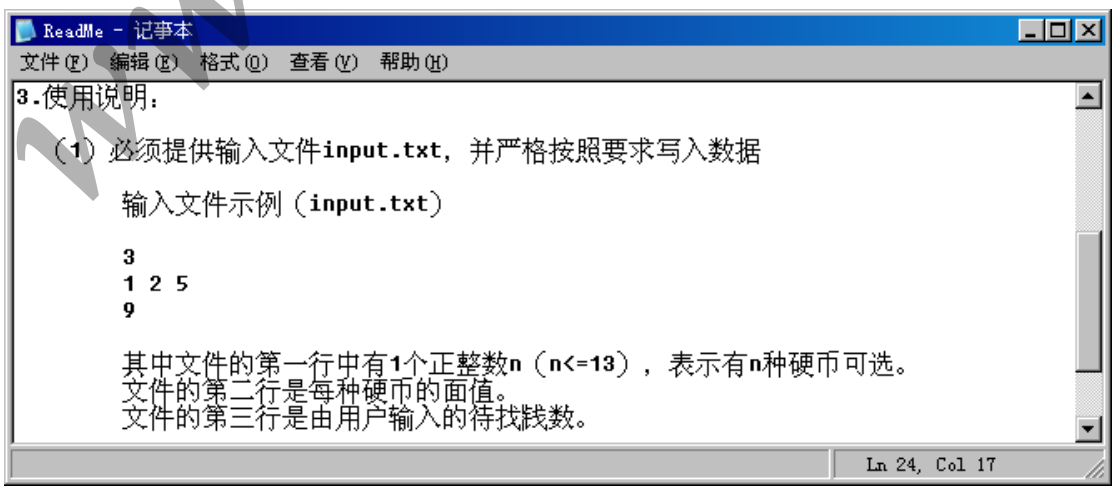
图中 C({1}, 7)被调用多次。如果将完整的树绘制出来，将会发现有许多子问题被多次调用。

我们发现，递归过程中，每次计算 Count(i)，都会重复计算 Count(1)....Count(i-1)；这样时间复杂度就是 O(N^2)。

由以上分析可知，硬币找零问题符合动态规划的两个重要属性，及满足最优子结构及重叠子问题。我们可以从 1 开始记录下每个钱数所需要的硬币枚数，避免重复计算。为了能够输出硬币序列，我们还需要记录下每次新加入的硬币。通过动态规划求解编程实现，相对上述递归程序而言效率有极大的提升。动态规划算法见附录。

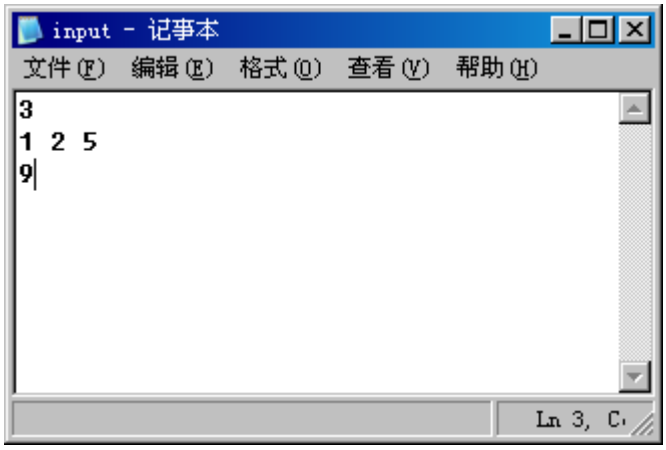
实验结果：

说明（具体说明文档见 ReadMe.txt）：

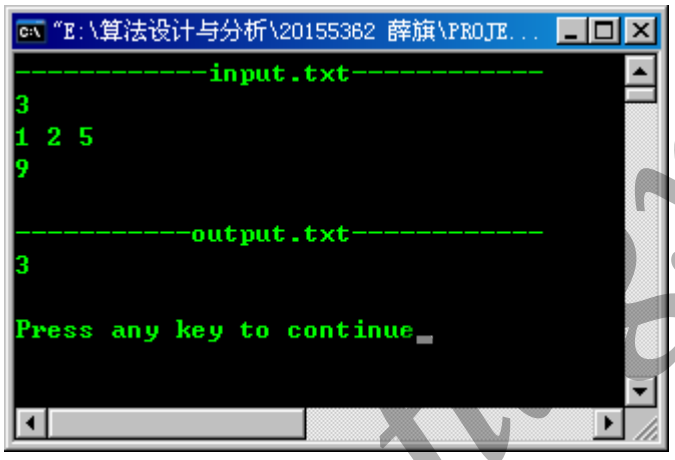


test1:

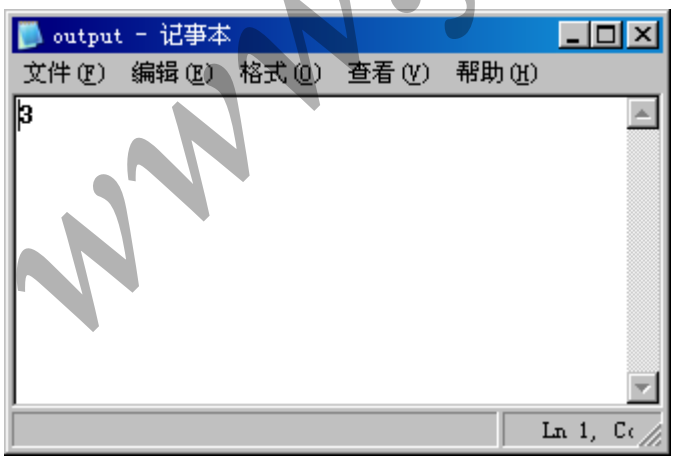
输入文件数据(input.txt):



运行:

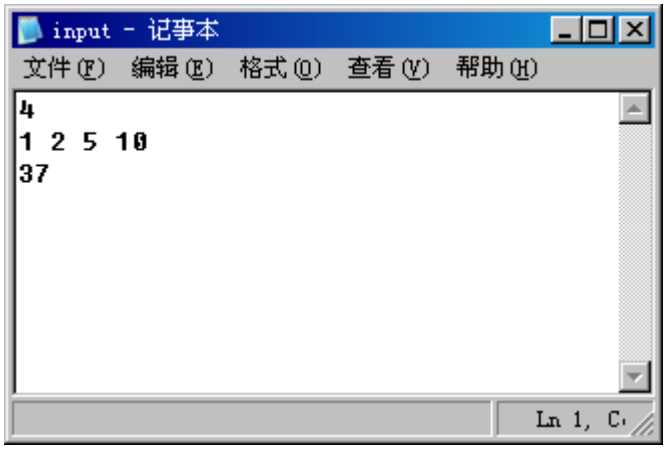


输出文件数据(output.txt):

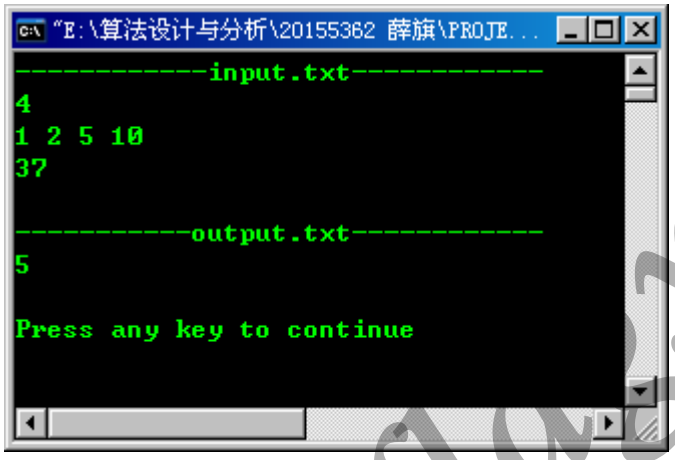


test2:

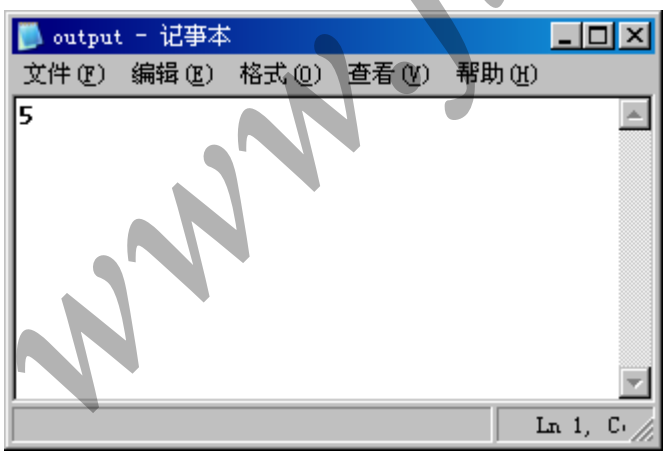
输入文件数据(input.txt):



运行：



输出文件数据(output.txt)：



实验总结：

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

动态规划的基本思想与分治法类似，也是将待求解的问题分解为若干个子问题(阶段)，按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一

子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

能采用动态规划求解的问题的一般要具有 3 个性质：

(1)最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2)无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3)有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

实验三：回溯法

实验目的：

（1）理解回溯法的思想。

（2）掌握一些经典的问题解决方法。

实验内容：

0-1 背包问题

★ **问题描述**

给定 n 种物品和一背包。物品 i 的重量是 wi>0，其价值为 vi>0，背包的容量为 c。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？

★ **编程任务**

利用回溯法试设计一个算法求出 0-1 背包问题的解，也就是求出一个解向量 x i（x i = 0 或 1，x i = 0 表示物体 i 不放入背包，x i =1 表示把物体 i 放入背包），使得尽量多的价值装入背包。

★ **数据输入**

由文件 input.txt 提供输入数据 n，c，及每个物品的重量 w[ ]和价值 v[ ]。

★ **结果输出**

程序运行结束时，将最优解输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4	1 1 0 1
5	

```
2 1 3 2
12 10 20 15
```

实验步骤：

(1)确定搜索空间：n 件物品的取舍数字化为：“取”标记为 1，“不取”标记为 0。则搜索的空间为 n 元一维数组$(x_1,x_2,x_3,...,x_n,)$，其值从

（0,0,0,...,0,0）,(0,0,0,...,1,0),...,(1,1,1,...,1,1)。这就是一棵子集树。

(2)确定约束条件。题目中的约束条件很明确：就是所取物品的重量和不超过 m。只有取当前物品时才需要判断所取物品的重量和不超过 m,若不取当前物品时,就无须进行判断，只要进一步进行深度搜索。

还有一个约束条件或者说是停止条件就是搜索完一个分支，也就是说完成一组$(x_1,x_2,x_3,...,x_n,)$的枚举。

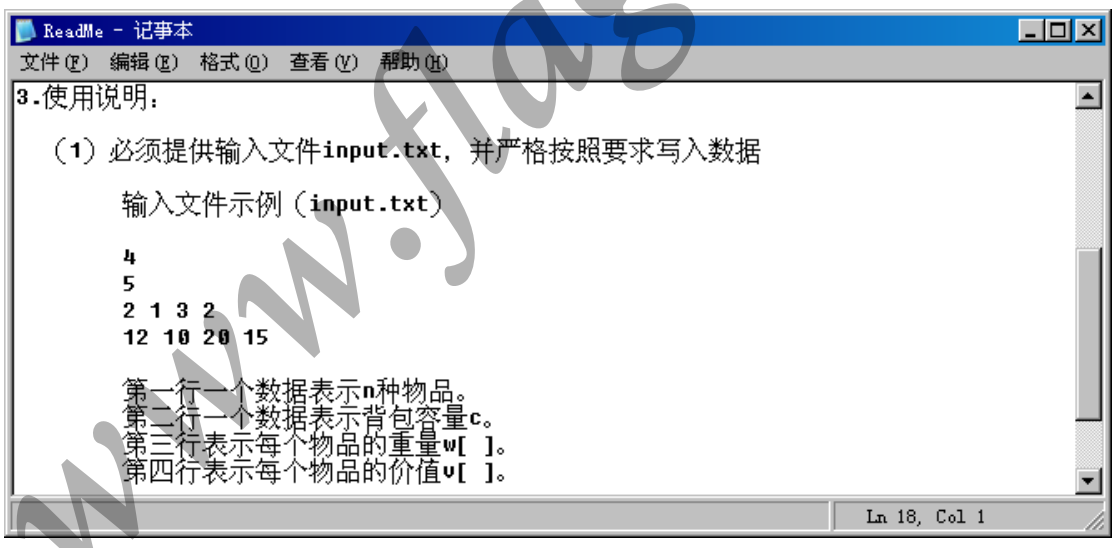
(3)搜索过程中一要累加所取物品的重量，回溯时还要做现场清理，也就是将当前物品置为不取状态，且从累加重量中减去当前物品的重量。

每搜索完一个分支就要计算该分支的获利情况，并记录当前的最大获利情况。

具体实现代码见附录。

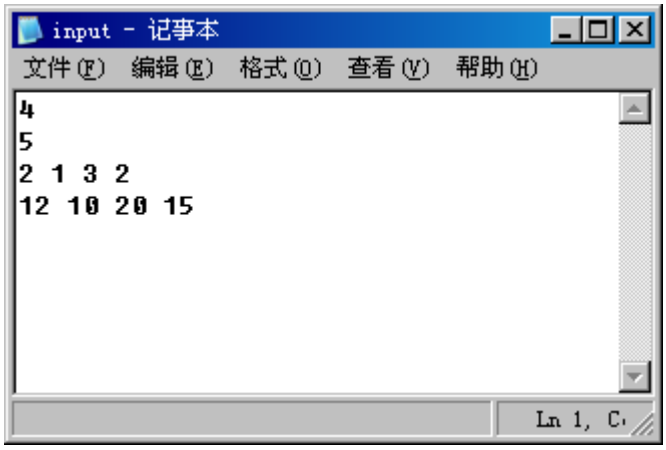
实验结果：

说明（具体说明文档见 ReadMe.txt）：

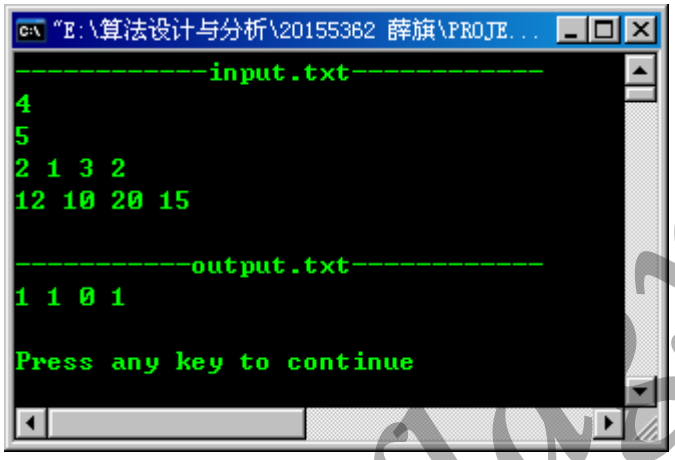


test1:

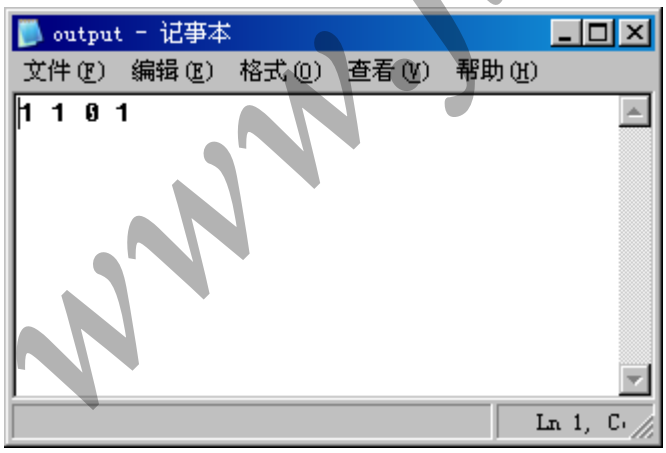
输入文件数据(input.txt)：



运行：

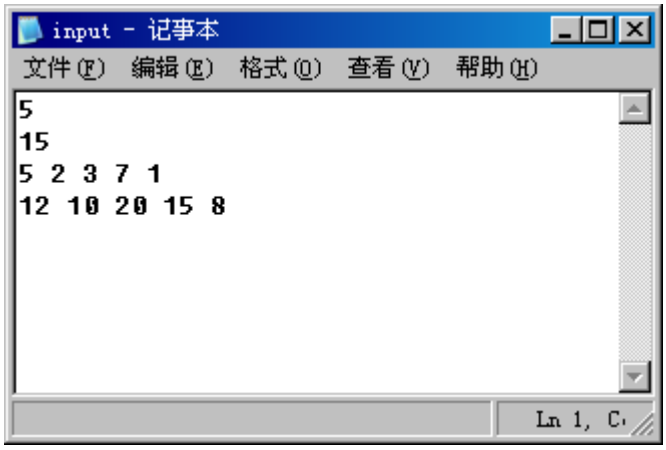


输出文件数据(output.txt)：

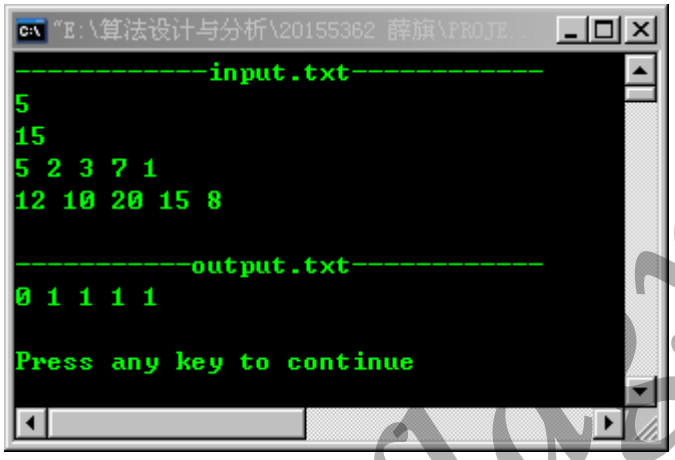


test2:

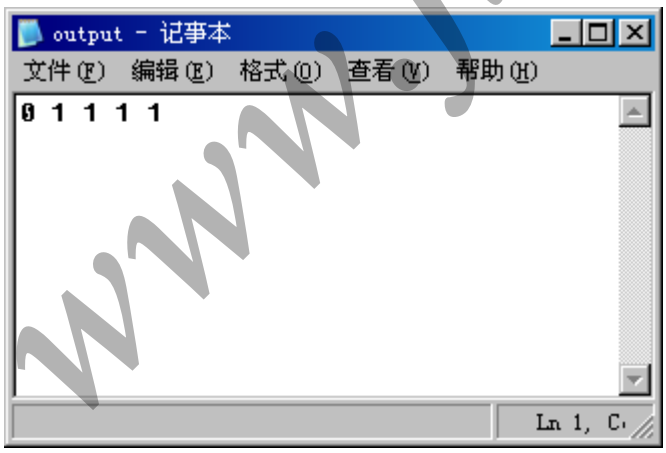
输入文件数据(input.txt)：



运行：



输出文件数据(output.txt)：



实验总结：

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择。

在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。

若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。

而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

用回溯法解题的一般步骤：

（1）针对所给问题，确定问题的解空间：

首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。

（2）确定结点的扩展搜索规则

（3）以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

附录：

实验一：

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <windows.h>  //min(),max()
using namespace std;
const int maxNum = 200;


double median(int arr[], int n) {  //计算中位数
    if (n % 2 == 0)
        return (arr[n / 2] + arr[n / 2 - 1]) / 2.0;
    else
        return arr[n / 2];
}

double getMedian(int X[], int Y[], int n) { //计算两个数组中的中位数
    double m1;   //数组X的中位数
    double m2;   //数组Y的中位数

    if (n <= 0)
        return -1;   //非法输入检测
    if (n == 1)
        return (X[0] + Y[0]) / 2.0;
    if (n == 2)
        return (max(X[0], Y[0]) + min(X[1], Y[1])) / 2.0;

    m1 = median(X, n);    //计算数组X中的中位数
    m2 = median(Y, n);    //计算数组Y中的中位数

```

```
30
31      if (m1 == m2)//如果两组计算得到相同的中位数，则返回结果
32          return m1;
33
34      if (m1 < m2) {   //如果m1 < m2, 则表明中位数在(m1, m2)之间
35          if (n % 2 == 0)
36              return getMedian(X + n / 2 - 1, Y, n - n / 2 + 1);
37          else
38              return getMedian(X + n / 2, Y, n - n / 2);
39      }
40
41      else {   //如果m1 > m2, 则中位数在(m2, m1)之间
42          if (n % 2 == 0)
43              return getMedian(Y + n / 2 - 1, X, n - n / 2 + 1);
44          else
45              return getMedian(Y + n / 2, X, n - n / 2);
46      }
47  }
48
49  int main() {
50      system("color 0a");
51      int i, n;
52      double mid;  //中位数
53      int X[maxNum] = { 0 };
54      int Y[maxNum] = { 0 };
55
56      //从文件读取数据
57      FILE *fp;
58      fp = fopen("input.txt", "r");
59      fscanf(fp, "%d", &n);
60      for (i = 0; i < n; i++)
61          fscanf(fp, "%d", &X[i]);
62      for (i = 0; i < n; i++)
63          fscanf(fp, "%d", &Y[i]);
64      fclose(fp);
65
66      //打印从文件读取的数据到屏幕
67      cout << "------------input.txt------------" << endl;
68      cout << n << endl;
69      for (i = 0; i < n; i++)
70          cout << X[i] << " ";
71      cout << endl;
72      for (i = 0; i < n; i++)
73          cout << Y[i] << " ";
```

```
    cout << endl;

    mid = getMedian(X, Y, n);

    //将计算得到的中位数保存到文件中
    fp = fopen("output.txt", "w+");
    fprintf(fp, "%.1f", mid);
    fclose(fp);

    //输出结果到屏幕
    cout << endl << "----------output.txt----------" << endl;
    printf("%.1f\n\n", mid);


    return 0;
}
```

实验二：

```
#include <iostream>
#include <windows.h>   //min()

using namespace std;
const int maxNum = 13;  //有13种硬币可选

int get_coins(int n, int T[], int j) {
    int i, k;
    if (n <= 0 || T == NULL || j < 1) {     //非法输入检测，对不合理输入给出提示
        cout << "非法输入!请确保input.txt文件内数据格式正确!" << endl;
        return -1;
    }
    int *coinNum = new int[j + 1]; //对钱数为1-j找零，最少需要硬币个数
    int *coinValue = new int[j + 1];    //判断是否可找零
    coinNum[0] = 0;
    for (i = 1; i <= j; i++) {
        int minNum = i;        //i面值，最少需要硬币个数
        int usedMoney = 0;
        for (k = 0; k < n; k++) {
            if (T[k] <= i) { //找零钱数大于硬币面值
                int temp = coinNum[i - T[k]] + 1;
                if (temp < minNum && (i == T[k] || coinValue[i - T[k]] != 0)) {
                    //如若零钱能找开，则更新最少硬币个数
                    minNum = temp;
```

```
                        usedMoney = T[k];
                    }
                }
            }
            coinNum[i] = minNum;
            coinValue[i] = usedMoney;
        }
        int counts = coinNum[j];
        if (coinValue[k] == 0) {
            return 9999999;   //找不出钱数j时，记C(i,j)=∞
        }
        return counts;
}

void main() {
        system("color 0a");
        int n;  //有n种硬币可选
        int T[maxNum] = { 0 };    //硬币面值存储于数组T中
        int i;
        int j;  //待找钱数

                //从文件读取数据
        FILE *fp;
        fp = fopen("input.txt", "r");
        fscanf(fp, "%d", &n);
        for (i = 0; i < n; i++)
            fscanf(fp, "%d", &T[i]);
        fscanf(fp, "%d", &j);

        //打印从文件读取的数据到屏幕
        cout << "------------input.txt------------" << endl;
        cout << n << endl;
        for (i = 0; i < n; i++)
            cout << T[i] << " ";
        cout << endl;
        cout << j << endl;

        //递归计算
        int result = get_coins(n, T, j);

        //将计算得到的中位数保存到文件中
        fp = fopen("output.txt", "w+");
        fprintf(fp, "%d", result);
        fclose(fp);
```

```
69
70      //输出结果到屏幕
71      cout << endl << "----------output.txt----------" << endl;
72      cout << result << endl << endl;
73  }
```

实验三：

```
1   /*
2   n:物品种类数
3   c:背包容量
4   w[]:每个物品的重量
5   v[]:每个物品的价值
6   x1[]:解向量，即取舍情况数组(取值为0或1)
7   total:累加索取物品的重量
8   sum:累加当前分支的获利值变量
9   max:存储当前最大获利值的变量
10  x[]:当前最大获利分支数组，即取舍情况数组(取值为0或1)
11  */
12  #include <iostream>
13  #include <windows.h>
14  using namespace std;
15  const int maxNum = 10;//物品最多的种类数
16  int x1[maxNum] = { 0 };
17  int x[maxNum] = { 0 };
18  int total = 0;
19  int max = 0;
20
21  void knap(int n, int c, int w[], int v[], int i) {   //i为递归深度
22      int j;
23      int sum = 0;
24      if (i == n) {//到达叶节点
25          for (j = 0; j < n; j++)
26              sum = sum + x1[j] * v[j];
27          if (sum > max) {
28              max = sum;
29              for (j = 0; j < n; j++)
30                  x[j] = x1[j];
31          }
32          return;
33      }
34      x1[i] = 0;    //不装入第(i+1)件物品的情形
35      knap(n, c, w, v, i + 1);
```

```
36      if (total + w[i] <= c) {   //可装入第(i+1)件物品的情形
37          x1[i] = 1;    //装入第(i+1)件物品的情形
38          total += w[i];
39          knap(n, c, w, v, i + 1);
40          x1[i] = 0;
41          total -= w[i];    //回溯之前清理现场
42      }
43  }
44
45  int main() {
46
47      int i, n, c, s;
48      s = 0;
49      int w[maxNum] = { 0 };
50      int v[maxNum] = { 0 };
51
52      system("color 0a");
53      //从文件读取数据
54      FILE *fp;
55      fp = fopen("input.txt", "r");
56      fscanf(fp, "%d", &n);
57      fscanf(fp, "%d", &c);
58      for (i = 0; i < n; i++) {
59          fscanf(fp, "%d", &w[i]);
60          s = s + w[i];
61      }
62      for (i = 0; i < n; i++)
63          fscanf(fp, "%d", &v[i]);
64
65      //打印从文件读取的数据到屏幕
66      cout << "------------input.txt------------" << endl;
67      cout << n << endl;
68      cout << c << endl;
69      for (i = 0; i < n; i++)
70          cout << w[i] << " ";
71      cout << endl;
72      for (i = 0; i < n; i++)
73          cout << v[i] << " ";
74      cout << endl;
75
76      if (s <= c) {
77          cout << endl << "Whole choose" << endl;
78          return 0;
79      }
```

```
80
81      cout << endl;
82
83      //递归计算
84      knap(n, c, w, v, 0);
85
86      //将计算得到的中位数保存到文件中
87      fp = fopen("output.txt", "w+");
88      for (i = 0; i < n; i++)
89          fprintf(fp, "%d ", x[i]);
90      fclose(fp);
91
92      //输出结果到屏幕
93      cout << "----------output.txt------------" << endl;
94      for (i = 0; i < n; i++)
95          cout << x[i] << " ";
96      cout << endl << endl;
97      return 0;
98  }
```

教师评语或**评价表格：**

评价表格示例：（考核标准与教学大纲中的实验考核标准一致）

考核标准	得分
（1）正确理解和掌握实验所涉及的概念和原理（10%）；	
（2）按实验要求合理设计数据结构和程序结构（20%）；	
（3）能设计测试用例，运行结果正确（20%）；	
（4）认真记录实验数据，原理及实验结果分析准确（40%）；	
（5）实验报告规范（10%）。	