“十二五”普通高等教育本科国家级规划教材

国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

◎ 王晓东　编著

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“十二五”普通高等教育本科国家级规划教材
国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

王晓东 编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

图书在版编目（CIP）数据

计算机算法设计与分析习题解答/王晓东编著. —5版. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34438-1

Ⅰ. ① 计… Ⅱ. ① 王… Ⅲ. ① 电子计算机－算法设计－高等学校－题解 ② 电子计算机－算法分析－高等学校－题解 Ⅳ. ① TP301.6-44

中国版本图书馆CIP数据核字（2018）第120711号

策划编辑：章海涛
责任编辑：章海涛
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×1092　1/16　　印张：22.75　　字数：580千字
版　　次：2005年8月第1版
　　2018年10月第5版
印　　次：2018年10月第1次印刷
定　　价：56.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：192910558（QQ群）。

前　言

一些著名的计算机科学家在有关计算机科学教育的论述中认为，计算机科学是一种创造性思维活动，其教育必须面向设计。“计算机算法设计与分析”正是一门面向设计，且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究，理解掌握算法设计的主要方法，培养对算法的计算复杂性正确分析的能力，为独立设计算法和对算法进行复杂性分析奠定坚实的理论基础，对每一位从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者都是非常重要和必不可少的。课程结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，在内容选材、深度把握、系统性和可用性方面进行了精心设计，力图适合高校本科生教学对学时数和知识结构的要求。

本书是“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》（ISBN 978-7-121-34439-8）配套的辅助教材，对《计算机算法设计与分析（第5版）》一书中的全部习题做了详尽的解答，旨在让使用该书的教师更容易教，学生更容易学。为了便于对照阅读，本书的章序与《计算机算法设计与分析（第5版）》一书的章序保持一致，且一一对应。

本书的内容是对《计算机算法设计与分析（第5版）》的较深入的扩展，许多教材中无法讲述的较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力，本书将主教材中的许多习题改造成算法实现题，要求学生不仅设计出解决具体问题的算法，而且能上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设，建立了“算法设计与分析”教学网站。国家精品资源共享课地址：http://www.icourses.cn/sCourse/course_2535.html。欢迎广大读者访问作者的教学网站并提出宝贵意见。

在本书编写过程中，福州大学“211工程”计算机与信息工程重点学科实验室为本书的写作提供了优良的设备与工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤劳动，他们认真细致、一丝不苟的工作精神保证了本书的出版质量。在此，谨向每位曾经关心和支持本书编写工作的各方面人士表示衷心的谢意！

作　者

目　录

第 3 章　动态规划

算法分析题 3

3-1　最长单调递增子序列。

设计一个 $O(n^2)$时间的算法，找出由 n 个数组成的序列的最长单调递增子序列。

分析与解答：用数组 b[0:n−1]记录以 a[i]（$0 \leqslant i < n$），为结尾元素的最长递增子序列的长度。序列 a 的最长递增子序列的长度为 $\max\limits_{0 \leqslant i \leqslant n}\{b[i]\}$。易知，b[$i$]满足最优子结构性质，可以递归地定义为

$$b[0]=1,\ b[i]=\max_{\substack{0 \leqslant k \leqslant i \\ a[k] \leqslant a[i]}}\{b[k]\}+1$$

据此将计算 b[i]转化为 i 个规模更小的子问题。

按此思想设计的动态规划算法描述如下。

```
int LISdyna() {
   int  i, j, k;
   for(i=1, b[0]=1; i < n; i++) {
      for(j=0, k=0; j<i; j++)
         if(a[j] <= a[i] && k < b[j])
            k = b[j];
      b[i] = k+1;
   }
   return maxL(n);
}
int maxL(int n) {
   for (int i=0, temp=0; i < n; i++)
      if (b[i] > temp)
         temp = b[i];
   return temp;
}
```

其中，算法 LISdyna 按照递归式计算出 b[0:n−1]的值，然后由 maxL 计算出序列 a 的最长递增子序列的长度。从算法 LISdyna 的二重循环容易看出，算法所需的计算时间为 $O(n^2)$。

3-2　最长单调递增子序列的 $O(n\log n)$算法。

将算法分析题 3-1 中算法的计算时间减至 $O(n\log n)$（提示：一个长度为 i 的候选子序列的最后一个元素至少与一个长度为 i−1 的候选子序列的最后一个元素一样大。通过指向输入序列中元素的指针来维持候选子序列）。

分析与解答：可用归纳设计策略解此问题。归纳假设是：已知计算序列 a[0:i−1]（$i<n$）的最长递增子序列的长度的正确算法。归纳的初始情况是平凡的。对于长度为 n 的序列 a[0:n−1]，应设法转换为长度小于 n 的序列。

用归纳设计策略解题时，归纳假设对应于算法循环中的循环不变式。本题的循环不变式 P 是：P:k 是序列 a[0:i]（$0 \leqslant i < n$）的最长递增子序列的长度。

容易看出，在由 $i-1$ 到 i 的循环中，a[i]的值起关键作用。如果 a[i]能扩展序列 a[0:$i-1$]的最长递增子序列的长度，则 $k=k+1$，否则 k 不变。设 a[0:$i-1$]中长度为 k 的最长递增子序列的结尾元素是 a[j]（$0 \leqslant j < i-1$），则当 a[i]$\geqslant$a[j]时可以扩展，否则不能扩展。如果序列 a[0:$i-1$]中有多个长度为 k 的最长递增子序列,那么需要存储哪些信息？容易看出，只要存储序列 a[0:$i-1$]中所有长度为 k 的递增子序列中结尾元素的最小值 b[k]。因此,需要将循环不变式 P 增强为：P:（$0 \leqslant i < n$）k 是序列 a[0:i]的最长递增子序列的长度；b[k]是序列 a[0:i]中所有长度为 k 的递增子序列中的最小结尾元素值。

相应地，归纳假设也增强为：已知计算序列 a[0:$i-1$]（$i<n$）的最长递增子序列的长度 k 以及序列 a[0:$i-1$]中所有长度为 k 的递增子序列中的最小结尾元素值 b[k]的正确算法。

增强归纳假设后，在由 $i-1$ 到 i 的循环中，当 a[i]$\geqslant$b[k]时，$k=k+1$，b[k]=a[i]，否则 k 值不变。注意到当 a[i]$\geqslant$b[k]时，k 值增大，b[k]的值为 a[i]。那么，当 a[i]$<$b[k]时，b[1:k]的值应该如何改变？如果 a[i]<b[1]，则显然应该将 b[1]的值改变为 a[i]。当 b[1]$\leqslant$a[i]$\leqslant$b[k]时，注意到数组 b 是有序的，可以用二分搜索算法找到下标 j，使得 b[$j-1$]$\leqslant$a[i]<b[j]。此时，b[1:$j-1$]和 b[$j+1$:k]的值不变，b[j]的值改变为 a[i]。

按上述思想设计的算法可实现如下。

```
int LIS() {
  b[1] = a[0];
  for(int i=1, k=1; i < n; i++) {
    if(a[i] >= b[k])
      b[++k] = a[i];
    else
      b[binary(i,k)] = a[i];
  }
  return k;
}
int binary(int i, int k) {
  if (a[i] < b[1])
    return 1;
  for(int h=1, j=k; h!=j-1; ) {
    if (b[k=(h+j)/2] <= a[i])
      h = k;
    else
      j = k;
  }
  return j;
}
```

其中，binary(i, k)用二分搜索算法在 b[1:k]中找到下标 j，使得 b[$j-1$]$\leqslant$a[i]<b[j]。算法 binary(i, k)所需的计算时间显然为 $O(\log k)$。由此可见，在最坏情况下，上述算法所需的计算时间为 $O(n\log n)$。

3-3 整数线性规划问题。

考虑下面的整数线性规划问题。

$$\max\sum_{i=1}^{n}c_ix_i \qquad \begin{cases}\sum_{i=1}^{n}a_ix_i \leqslant b\\ x_i\text{为非负整数} \quad 1\leqslant i\leqslant n\end{cases}$$

试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

分析与解答：该问题是一般情况下的背包问题，具有最优子结构性质。

设所给背包问题的子问题

$$\max\sum_{k=1}^{i}c_kx_k \qquad \sum_{k=1}^{i}a_kx_k \leqslant j$$

的最优值为 $m(i,j)$，即 $m(i,j)$是背包容量为 j，可选择物品为 $1,2,\cdots,i$ 时背包问题的最优值。由背包问题的最优子结构性质，可以建立计算 $m(i,j)$的递归式如下：

$$m(i,j)=\begin{cases}\max\{m(i-1,j),m(i,j-a_i)+c_i\} & a_i\leqslant j\\ m(i-1,j) & 0\leqslant j<a_i\end{cases}$$

$$m(0,j)=m(i,0)=0;\ m(i,j)=-\infty,\ j<0$$

按此递归式计算出的 $m(n,b)$为最优值。算法所需的计算时间为 $O(nb)$。

3-4 二维 0-1 背包问题。

给定 n 种物品和一背包。物品 i 的重量是 w_i，体积是 b_i，其价值为 v_i，背包的容量为 c，容积为 d。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？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

分析与解答：该问题是二维 0-1 背包问题。问题的形式化描述是：给定 $c>0$，$d>0$，$w_i>0$，$b_i>0$，$v_i>0$（$1\leqslant i\leqslant n$），要求找出 n 元 0-1 向量$(x_1,x_2,\cdots,x_n)$，$x_i\in\{0,1\}$（$1\leqslant i\leqslant n$），使得 $\sum_{i=1}^{n}w_ix_i\leqslant c$，$\sum_{i=1}^{n}b_ix_i\leqslant d$ 而且 $\sum_{i=1}^{n}v_ix_i$ 达到最大。

因此，二维 0-1 背包问题也是一个特殊的整数规划问题。

$$\max\sum_{i=1}^{n}v_ix_i \qquad \begin{cases}\sum_{i=1}^{n}w_ix_i\leqslant c\\ \sum_{i=1}^{n}b_ix_i\leqslant d\end{cases} \qquad x_i\in\{0,1\},\ 1\leqslant i\leqslant n$$

容易证明，该问题具有最优子结构性质。

设所给二维 0-1 背包问题的子问题

$$\max\sum_{t=i}^{n}v_tx_t \qquad \begin{cases}\sum_{t=i}^{n}w_tx_t\leqslant j\\ \sum_{t=i}^{n}b_tx_t\leqslant k\end{cases} \qquad x_t\in\{0,1\},\ i\leqslant t\leqslant n$$

的最优值为 $m(i,j,k)$，即 $m(i,j,k)$是背包容量为 j，容积为 k，可选择物品为 i、$i+1$、…、n 时二维 0-1 背包问题的最优值。由二维 0-1 背包问题的最优子结构性质，可以建立计算 $m(i,j,k)$的递归式如下：

$$m(i,j,k)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i,k-b_i)+v_i\} & j\geqslant w_i\text{且}k\geqslant b_i\\ m(i+1,j) & 0\leqslant j<w_i\text{或}0\leqslant k<b_i\end{cases}$$

$$m(n,j,k)=\begin{cases}v_n & j\geqslant w_i\text{且}k\geqslant b_n\\ 0 & 0\leqslant j<w_n\text{或}0\leqslant k<b_n\end{cases}$$

按此递归式计算出的 $m(n, c, d)$为最优值。算法所需的计算时间为 $O(ncd)$。

3-5 Ackermann 函数。

Ackermann 函数 $A(m, n)$可递归定义如下：

$$A(m,n)=\begin{cases}n+1 & m=0\\ A(m-1,1) & m>0,\ n=0\\ A(m-1,A(m,n-1)) & m>0,\ n>0\end{cases}$$

试设计一个计算 $A(m, n)$的动态规划算法，该算法只占用 $O(m)$空间（提示：用两个数组 val[0:m]和 ind[0:m]，使得对任何 i 有 val[i]=A(i, ind[i])）。

分析与解答：按定义容易写出递归算法。

```
int ackermann(int m, int n) {
   if(m == 0)
      return n+1;
   if(n == 0)
      return ackermann(m-1, 1);
   else
      return ackermann(m-1, ackermann(m, n-1));
}
```

按备忘录方法的思想，可将上述算法修改为备忘录算法。

```
int ack(int m, int n) {
   if(a[m][n])
      return a[m][n];
   if(m == 0)
      return a[0][n] = n+1;
   if(n == 0)
      return a[m][0] = ack(m-1, 1);
   return a[m][n] = ack(m-1, ack(m, n-1));
}
```

另外，可用消去递归的方法，将上述算法非递归化如下。

```
int ackm(int m, int n) {
   int  top = 1;
   s[1][1] = m;
   s[1][2] = n;
   while (top > 0){
      m = s[top][1];
      n = s[top][2];
      top--;
      if (top == 0 && m == 0)
         return n+1;
      if (m == 0)
```

```
            s[top][2] = n+1;
        else if(n == 0) {
            s[++top][1] = m-1;
            s[top][2] = 1;
        }
        else {
            s[++top][1] = m-1;
            s[++top][1] = m;
            s[top][2] = n-1;
        }
    }
    return s[0][1];
}
```

实际上，还有一个稍简单的递归算法。

```
int ack(int m, int n) {
    for(int i=m; i > 0; i--) {
        if (n == 0)
            n = 1;
        else
            n = ack(i, n-1);
    }
    return n+1;
}
```

同样可将其改造为备忘录算法。

这些算法都是自顶向下的递归算法。下面考察自底向上的动态规划算法。

首先对于较小的 m 和 n，考察 Ackermann 函数 A(m, n)值的变化，如图 3-1 所示。

m \ n	0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12
2	3	5	7	9	11	13	15	17	19	21	23
3	5	13	29	61	125	253	509	1021	2045	4093	8189
4	13	65533									
5	65533										

图 3-1　Ackermann 函数 $A(m, n)$值的变化

可以看出，当 m=0 时，第 0 行对应 A(0, n)的值。当 n=0 时，第 0 列对应 A(m, 0)的值，其值恰好是第 m−1 行第 1 列的值。A(m, n)的值等于第 m−1 行第 A(m, n−1)列的值。据此，可从第 1 行的值依次递推出各行的值。为此，用 2 个数组 val[0:m]和 ind[0:m]分别记录当前第 i 行计算到第 ind[i]列的值，即已计算到 A(i, ind[i])的值，这个值存储于 val[i]中。当计算到 A(m, n)时，算法结束。按此思想设计的动态规划算法如下。

```
int ack(int m, int n) {
    int  i, *val, *ind;
    if(m == 0)
        return n+1;
```

```
    val = new int[m+1];
    ind = new int[m+1];
    for(i=0; i <= m; i++) {
       val[i]=-1;
       ind[i]=-2;
    }
    val[0] = 1;
    ind[0] = 0;
    while(ind[m] < n) {
       val[0]++;
       ind[0]++;
       for(i=0; i < m; i++){
          if(ind[i] == 1 && ind[i+1] < 0) {
             val[i+1] = val[0];
             ind[i+1] = 0;
          }
          if(val[i+1] == ind[i]) {
             val[i+1] = val[0];
             ind[i+1]++;
          }
       }
    }
    return val[m];
}
```

算法显然只占用 $O(m)$空间。

算法实现题 3

3-1　独立任务最优调度问题。

问题描述：用两台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时需要时间 a_i，若由机器 B 来处理，则需要时间 b_i。由于各作业的特点和机器的性能关系，可能对于某些 i，有 $a_i \geqslant b_i$，而对于某些 j（$j \neq i$），有 $a_j < b_j$。既不能将一个作业分开由两台机器处理，也没有一台机器能同时处理 2 个作业。设计一个动态规划算法，使得这两台机器处理完这 n 个作业的时间最短（从任何一台机器开工到最后一台机器停工的总时间）。研究一个实例：$(a_1, a_2, a_3, a_4, a_5, a_6)=(2, 5, 7, 10, 5, 2)$，$(b_1, b_2, b_3, b_4, b_5, b_6)=(3, 8, 4, 11, 3, 4)$。

算法设计：对于给定的两台处理机 A 和 B 处理 n 个作业，找出一个最优调度方案，使 2 台机器处理完这 n 个作业的时间最短。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 n，表示要处理 n 个作业。在接下来的 2 行中，每行有 n 个正整数，分别表示处理机 A 和 B 处理第 i 个作业需要的处理时间。

结果输出：将计算出的最短处理时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt

6 15

2 5 7 10 5 2

3 8 4 11 3 4

分析与解答：

（1）计算 $m=\max\{\max_{1\le i\le n}\{a_i\},\ \max_{1\le i\le n}\{b_i\}\}$。

（2）设布尔量 $p(i, j, k)$表示前 k 个作业可以在处理机 A 用时不超过 i 时间且在处理机 B 用时不超过 j 时间内完成。用动态规划算法计算

$$p(i,j,k)=p(i-a_k,j,k-1)\mid p(i,j-b_k,k-1)$$

（3）由（2）的结果可以计算最短处理时间如下：

$$\min_{0\le i\le mn,\,0\le j\le mn,\,p(i,j,n)=\text{true}}\{\max\{i,j\}\}$$

具体算法实现如下。

read()函数读入初始数据，并计算 m 的值。

```
void read(){
   cin >> n;
   m = 0;
   a = new int[n];
   b = new int[n];
   for(int i=0; i < n; i++) {
      cin >> a[i];
      if(a[i] > m)
         m=a[i];
      }
   for(i=0; i < n; i++) {
      cin>>b[i];
      if(b[i] > m)
         m = a[i];
   }
   mn = m*n;
   Make3DArray(p, mn+1, mn+1, n+1);
}
```

dyna()函数实现动态规划算法。

```
void dyna() {
   int  i, j, k;
   for(i=0; i <= mn; i++) {
      for(j=0; j <= mn; j++) {
         p[i][j][0] = true;
         for(k=1; k <= n; k++)
            p[i][j][k] = false;
      }
   }
   for(k=1; k <= n; k++) {
      for(i=0; i <= mn; i++) {
         for(j=0; j <= mn; j++) {
```

```
            if(i-a[k-1] >= 0)
              p[i][j][k] = p[i-a[k1]][j][k-1];
            if(j-b[k-1] >= 0)
              p[i][j][k] = (p[i][j][k]||p[i][j-b[k-1]][k-1]);
          }
        }
      }
      for(i=0, opt=mn; i <= mn; i++) {
        for(j=0; j <= mn; j++) {
          if(p[i][j][n]) {
            int  tmp = (i>j) ? i : j;
            if(tmp < opt)
              opt = tmp;
          }
        }
      }
      cout << opt << endl;
}
```

实现算法的主函数如下。

```
int main() {
  read();
  dyna();
  return 0;
}
```

上述算法所需的计算时间显然为 $O(m^2n^3)$。

3-2 最优批处理问题。

问题描述：在一台超级计算机上，编号为 1、2、…、n 的 n 个作业等待批处理。批处理的任务就是将这 n 个作业分成若干批，每批包含相邻的若干作业。从时刻 0 开始，分批加工这些作业。在每批作业开始前，机器需要启动时间 S，而完成这批作业所需的时间是单独完成批中各个作业需要时间的总和。单独完成第 i 个作业所需的时间是 t_i，所需的费用是它的完成时刻乘以一个费用系数 f_i。同一批作业将在同一时刻完成。例如，如果在时刻 T 开始一批作业 $x, x+1, \cdots, x+k$，则这一批作业的完成时刻均为 $T+S+(t_x+t_{x+1}+\cdots+t_{x+k})$。最优批处理问题就是要确定总费用最小的批处理方案。例如，假定有 5 个作业等待批处理，且

$$S=1,(t_1,t_2,t_3,t_4,t_5)=(1,3,4,2,1),(f_1,f_2,f_3,f_4,f_5)=(3,2,3,3,4)$$

如果采用批处理方案 $\{1,2\},\{3\},\{4,5\}$，则各作业的完成时间分别为 $(5,5,10,14,14)$，各作业的费用分别为 $(15,10,30,42,56)$，因此，这个批处理方案总费用是 153。

算法设计：对于给定的待批处理的 n 个作业，计算其总费用最小的批处理方案。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是待批处理的作业数 n，第 2 行是启动时间 S。接下来每行有 2 个数，分别为单独完成第 i 个作业所需的时间是 t_i 和所需的费用系数 f_i。

结果输出：将计算出的最小总费用输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt

```
5                                    153
1
1 3
3 2
4 3
2 3
1 4
```

分析与解答：

（1）基本动态规划算法

设 z_i 是批处理作业序列 $i, i+1, \cdots, n$ 所需的最小费用。设 $z_i(j)$ 是在批处理作业序列 i、$i+1$、…、n 中选取 i、$i+1$、…、$j-1$ 为第 1 批作业前提下所需的最小费用，即：

$$z_i(j) = z_j + (S + t_i + t_{i+1} + \cdots + t_{j-1})(w_i + w_{i+1} + \cdots + w_n)$$

由此可建立动态规划递归式如下：

$$z_i = \begin{cases} 0, & i > n \\ \min\limits_{i<j\leqslant n} z_i(j), & 1 \leqslant i \leqslant n \end{cases} \tag{3.1}$$

设

$$\begin{cases} st_i = \sum\limits_{j=i}^{n} t_j \\ sw_i = \sum\limits_{j=i}^{n} w_j \end{cases} \tag{3.2}$$

则（3.1）等价于

$$z_i = \begin{cases} 0, & i > n \\ \min\limits_{i<j\leqslant n} z_j + sw_i * (S + st_i - st_j) & 1 \leqslant i \leqslant n \end{cases} \tag{3.3}$$

显然，z_1 为所要求的最小费用。据此容易设计如下求所需的最小费用的动态规划算法。

```
int dyna() {
   z[n+1] = 0;
   for (i=n; i; i--) {
      st[i] = st[i+1]+t[i];
      sw[i] = sw[i+1]+w[i];
   }
   for(i=n; i; i--)
      for(j=i+1; j <= n+1; j++)
         z[i] = min(z[i], z[j]+sw[i]*(S+st[i]-st[j]));
   return z[1];
}
```

由于 for 循环体只需 $O(1)$ 时间，因此算法所需时间为 $O(n^2)$。

（2）动态规划算法加速

由于此问题具有一些特殊性质，解此问题的动态规划算法可以得到进一步改进。

首先，考察如下较一般的 1 维动态规划模型。

$$g(i)=\begin{cases}0, & i=0\\ a_i+\min\limits_{0\leqslant j<i}\{g(j)+b_j+c_id_j\} & 1\leqslant i\leqslant n\end{cases} \tag{3.4}$$

其中，a_i,b_j,c_i,d_j（$1\leqslant i\leqslant n$，$0\leqslant j\leqslant n-1$）是仅依赖其下标的常数。

此递推式的关键是依次对$i=1,2,\cdots,n$，快速计算

$$m_i=\min_{0\leqslant j<i}\{g(j)+b_j+c_id_j\} \tag{3.5}$$

按照基本算法计算m_i需要$O(i)$时间，因此总耗时为$O(n^2)$。

如果将$g(j)+b_j+c_id_j$看作以c_i为变量的函数

$$\begin{cases}y_j(x)=g(j)+b_j+d_jx\\ f_i(x)=\min\limits_{0\leqslant j<i}\{y_j(x)\}\end{cases} \tag{3.6}$$

则问题等价于求$f_i(c_i)=m_i$的值。显而易见，对于$0\leqslant j<i$，$y_j(x)=g(j)+b_j+d_jx$是平面上经过点$(0,g(j)+b_j)$的一条直线。对于这i条直线均有$m_i\leqslant y_j(c_i)$。换句话说，平面上的点(c_i,m_i)位于i个半平面$H_j=\{(x,y)\mid y\leqslant g(j)+b_j+d_jx\}$（$0\leqslant j<i$）的交$P_i=\bigcap\limits_{0\leqslant j<i}H_j$中。由于$H_j$（$0\leqslant j<i$）是凸集，$P_i$是一个凸多边形，因此其上凸壳对应的函数就是$f_i(x)$。它是一个分段线性函数，其中的线段就是直线$y_j(x)=g(j)+b_j+d_jx$的一段，如图3-2所示。

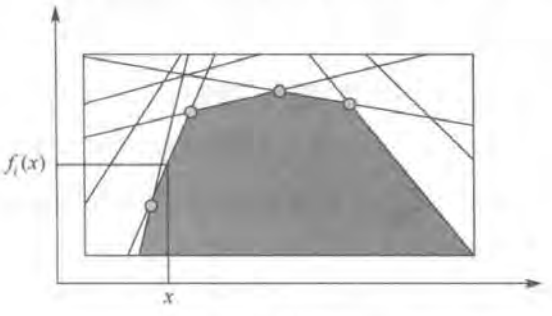

图3-2　半平面的交

由此可见，所求的动态规划问题等价于如下的计算几何问题。

对于$i=1,2,\cdots,n$，维护P_i的上凸壳$f_i(x)$，并计算$g(i)=a_i+f_i(c_i)=a_i+m_i$的值。最后，$g(n)=a_n+f_n(c_n)$即所求的最优值。解此计算几何问题的算法可描述如下。

Algorithm 1：求$g(n)$的几何算法

Output：$g(n)$

$g(0)\leftarrow 0$; $f_1(x)\leftarrow g(0)+b_0+d_0x$;

for i=1 to n **do**

　（1）根据$f_i(x)$计算出$f_i(c_i)$; $g(i)\leftarrow a_i+f_i(c_i)$;

　（2）将直线$y_i(x)=g(i)+b_i+d_ix$加入上凸壳$f_i(x)$得到$f_{i+1}(x)$;

end

return $g(n)$;

为了高效实现上述算法，需要设计一个数据结构来高效维护上凸壳$f_i(x)$。注意到上凸

壳 $f_i(x)$ 的每条直线段都属于唯一的直线 $y_j(x)$（$0\leqslant j<i$）。假设 $f_i(x)$ 由 k 个直线段 $s_1, s_2, \cdots, s_k$ 组成，它们分别属于直线 $y_{j_1}(x), y_{j_2}(x), \cdots, y_{j_k}(x)$，则上凸壳 $f_i(x)$ 上从左到右排列的 $k-1$ 个极点 $p_1=(x_1, y_1), p_2=(x_2, y_2), \cdots, p_{k-1}=(x_{k-1}, y_{k-1})$，分别为线段 s_t 与 s_{t+1} 的交点，$t=1, \cdots, k-1$。它们也是直线 $y_{j_t}(x)$ 与 $y_{j_{t+1}}(x)$ 的交点，$t=1, \cdots, k-1$。考察 $k-1$ 个极点 p_t 的横坐标 x_t（$0\leqslant t<k$），和 k 条直线 $y_{j_t}(x)$ 的斜率 d_{j_t}（$1\leqslant t\leqslant k$），它们具有如下单调性质：

$$\begin{cases} x_t < x_{t+1} & 1\leqslant t\leqslant k-2 \\ d_{j_t} > d_{j_{t+1}} & 1\leqslant t\leqslant k-1 \end{cases} \tag{3.7}$$

换句话说，x_t（$1\leqslant t<k$）从左到右单调递增，d_{j_t}（$1\leqslant t\leqslant k$）从左到右单调递减。

利用这个单调性质，将 k 条直线 $y_{j_t}(x)$ 的下标 j_t（$1\leqslant t\leqslant k$）按从左到右的序存储在一棵平衡树 T 中，就可以高效维护上凸壳 $f_i(x)$。

在算法的步骤（1）中，计算 $f_i(c_i)$ 就等价于在 T 中找到与 $x=c_i$ 相交的直线 $y_{j_t}(x)$。利用 x_t（$1\leqslant t<k$）单调性，这可以在 T 中用 $O(\log k)$ 时间实现。

在算法的步骤（2）中，要将直线 $y_i(x)=g(i)+b_i+d_i x$ 加入上凸壳 $f_i(x)$，就要计算 $y_i(x)$ 与 $f_i(x)$ 的交点。此时，需要区分交点个数为 0、1 和 2 这 3 种情形。

当 $y_i(x)$ 位于 $f_i(x)$ 上方时，交点个数为 0，此时 $f_i(x)$ 不变，即 $f_{i+1}(x)=f_i(x)$。

当 $y_i(x)$ 的斜率 d_i 大于 $f_i(x)$ 中所有直线的斜率，或 $y_i(x)$ 的斜率 d_i 小于 $f_i(x)$ 中所有直线的斜率时，交点个数为 1，$f_{i+1}(x)$ 对 $f_i(x)$ 的修改在其两端发生。其他情况下交点个数为 2，$f_{i+1}(x)$ 对 $f_i(x)$ 的修改在其中间发生。

首先，利用 d_{j_t}（$1\leqslant t\leqslant k$）在 T 中的单调性，搜索 $y_i(x)$ 在 $f_i(x)$ 中的插入位置。当 $d_i>d_{j_t}$（$1\leqslant t\leqslant k$）或 $d_i<d_{j_t}$（$1\leqslant t\leqslant k$）时，$y_i(x)$ 与 $f_i(x)$ 的交点个数为 1。否则，必存在相邻的 2 个直线斜率 d_{j_l} 和 $d_{j_{l+1}}$ 使得 $d_{j_l}>d_i\geqslant d_{j_{l+1}}$。如果这 2 个相邻线段的交点 p_l 位于直线 $y_i(x)$ 的下方，则 $y_i(x)$ 与 $f_i(x)$ 的交点个数为 0，否则交点个数为 2。

在交点个数为 2 的情形，可以依次向 p_l 的左和右方向搜索位于直线 $y_i(x)$ 的上方的最左极点 p_l 和最右极点 p_r（$l\leqslant t\leqslant r$）。p_l 是斜率为 d_{j_l} 和 $d_{j_{l+1}}$ 的两条相邻直线的交点。p_r 是斜率为 d_{j_r} 和 $d_{j_{r+1}}$ 的两条相邻直线的交点，如图 3-3 所示。将直线 $y_{j_e}(x)$（$l+1\leqslant e\leqslant r$）从 T 中删除，并将 $y_i(x)$ 插入 T 中就将 $f_i(x)$ 修改成了 $f_{i+1}(x)$。

在交点个数为 1 且 $d_i>d_{j_t}$（$1\leqslant t\leqslant k$）的情形，可以从 d_{j_1} 开始依次向右搜索位于直线 $y_i(x)$ 的上方的最右极点 p_r。p_r 是斜率为 d_{j_r} 和 $d_{j_{r+1}}$ 的两条相邻直线的交点，如图 3-4 所示。将直线 $y_{j_e}(x)$（$e\leqslant r$）从 T 中删除，并将 $y_i(x)$ 插入 T 中就将 $f_i(x)$ 修改成了 $f_{i+1}(x)$。如果 p_r 不存在，将 $y_i(x)$ 插入 T 的最左端即可。

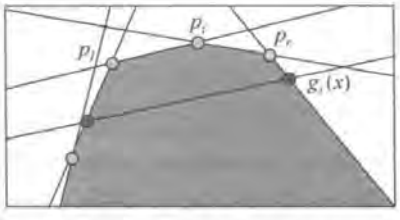

图 3-3　交点个数为 2 的情形

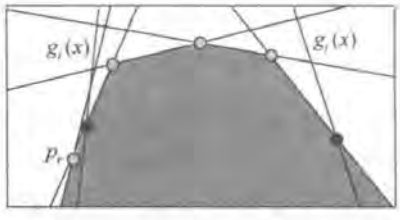

图 3-4　交点个数为 1 的情形

类似地，在交点个数为 1 且 $d_i < d_{j_t}$（$1 \leqslant t \leqslant k$）的情形，可以从 d_{j_k} 开始依次向左搜索位于直线 $y_i(x)$ 的上方的最左极点 p_l。p_l 是斜率为 d_{j_l} 和 $d_{j_{l+1}}$ 的两条相邻直线的交点。将直线 $y_{j_e}(x)$（$l+1 \leqslant e$）从 T 中删除，将 $y_i(x)$ 插入 T 中，就将 $f_l(x)$ 修改成了 $f_{l+1}(x)$。如果 p_l 不存在，将 $y_i(x)$ 插入 T 的最右端即可。

在 T 中搜索 $y_i(x)$ 在 $f_l(x)$ 中的插入位置需要 $O(\log k)$ 时间。在 T 中删除一条直线需要 $O(\log k)$ 时间。整个算法最多删除 n 条直线，因此从 T 中删除直线需要的总时间为 $O(n\log n)$。在 T 中插入一条直线需要 $O(\log k)$ 时间。整个算法最多插入 n 条直线，因此在 T 中插入直线需要的总时间为 $O(n\log n)$。

由此可见，算法的步骤（2）需要的总时间为 $O(n\log n)$。算法的步骤（1）依次需要 $O(\log k)$ 时间，因此它需要的总时间为 $O(n\log n)$。综上所述，整个算法需要的总时间为 $O(n\log n)$。

在一些重要的特殊情况下，上述算法还可以进一步改进。

① 如果 c_i（$1 \leqslant i \leqslant n$）是单调序列，则算法的步骤（1）可以在 $O(n)$ 时间内完成。事实上，假设 c_i（$1 \leqslant i \leqslant n$）是单调不减序列，则每次只要在 T 中向右寻找与 $x = c_i$ 相交的直线 $y_{j_l}(x)$，而不必从树根开始搜索。最坏情况下需要 $O(n)$ 次向右移动。

② 另一方面，如果 d_i（$1 \leqslant i \leqslant n$）是单调序列，则算法的步骤（2）可以在 $O(n)$ 时间内完成。事实上，假设 d_i（$1 \leqslant i \leqslant n$）是单调不增序列，则每次直线 $y_i(x)$ 加入上凸壳 $f_l(x)$ 所引发的直线的插入和删除都在上凸壳的右端发生。显而易见，此时可以用一个栈来存储上凸壳，而不必用平衡树。所有运算就可以在 $O(n)$ 时间内完成。

结合（1）和（2）可知，如果 c_i（$1 \leqslant i \leqslant n$）和 d_i（$1 \leqslant i \leqslant n$）均是单调序列，则问题（3.4）可以在 $O(n)$ 时间内求解。

现在回到最优批处理问题。在（3.3）中取

$$\begin{cases} g(i) = z_i \\ a_i = sw_i * (S + st_i) \\ b_i = 0 \\ c_i = -sw_i \\ d_i = st_i \end{cases} \tag{3.8}$$

则（3.3）等价于

$$g(i) = \begin{cases} 0 & i > n \\ a_i + \min\limits_{i<j\leqslant n}\{g(j) + b_j + c_i d_j\} & 1 \leqslant i \leqslant n \end{cases} \tag{3.9}$$

注意到（3.9）与（3.4）的唯一区别在于（3.4）是前向动态规划，而（3.9）是后向动态规划。它们之间没有本质差别。

因此，用 Algorithm 1 可以在 $O(n\log n)$ 时间内得到最优批处理问题的解。

从（3.2）可知，对于 $1 \leqslant i < n$ 有

$$\begin{cases} st_{i+1} < st_i \\ sw_{i+1} < sw_i \end{cases} \tag{3.10}$$

从而 c_i（$1 \leqslant i \leqslant n$）和 d_i（$1 \leqslant i \leqslant n$）均是单调序列。因此，解最优批处理问题所需的时间可以进一步减少为 $O(n)$。

事实上，此时式（3.6）变成

$$\begin{cases} y_j(x) = g(j) + st_j x \\ f_i(x) = \min\limits_{0 \le j < i} \{ y_j(x) \} \end{cases} \tag{3.11}$$

任何两条直线 $y_p(x)$ 和 $y_q(x)$ 的交点坐标 (x_{pq}, y_{pq}) 的值为

$$\begin{cases} x_{pq} = \dfrac{g(q) - g(p)}{st_p - st_q} \\ y_{pq} = \dfrac{st_p g(q) - st_q g(p)}{st_p - st_q} \end{cases} \tag{3.12}$$

在 Algorithm 1 的步骤（1）中，向右寻找与 $x = c_i = -sw_i$ 相交的直线 $y_{j_t}(x)$ 时，只要判断两直线 $y_{j_t}(x)$ 和 $y_{j_{t+1}}(x)$ 交点的 x 坐标值是否大于 c_i。

由式（3.12）即知，这等价于

$$g(j_t) - g(j_{t+1}) \geqslant sw_i(st_{j_t} - st_{j_{t+1}}) \tag{3.13}$$

确定 $x = c_i = -sw_i$ 与直线 $y_{j_t}(x)$ 相交后，可以计算出 $g(i) = a_i + y_{j_t}(x)(c_i)$，即

$$g(i) = g(j_t) + sw_i(S + st_i - st_{j_t}) \tag{3.14}$$

在 Algorithm 1 的步骤（2）中，需要判断栈顶两直线 $y_{j_t}(x)$ 和 $y_{j_{t-1}}(x)$ 的交点是否位于待插入直线 $y_i(x)$ 之上。

由式（3.12）即知，这等价于

$$\frac{st_p g(q) - st_q g(p)}{st_p - st_q} \geqslant g(i) + st_i \frac{g(q) - g(p)}{st_p - st_q} \tag{3.15}$$

其中，$p = j_{t-1}, q = j_t$。进一步简化后，这个条件等价于

$$st_p(g(i) - g(q)) + st_q(g(p) - g(i)) + st_i(g(q) - g(p)) \geqslant 0 \tag{3.16}$$

根据上面的讨论，改进后的 $O(n)$ 时间算法可描述如下。

```
int batch() {
  int  l = 0, r = 0;
   stk[r] = n+1;
   for(int i=n; i > 0; i--) {
      st[i] = st[i+1]+t[i];
      sw[i] = sw[i+1]+w[i];
   }
   for(int i=n; i>0; i--) {
      while(l < r && right(stk[l], stk[l+1], i))
         l++;                                          // move right
      g[i] = gg(stk[l], i);
      while(l < r && top(stk[r-1], stk[r], i))
         --r;                                          // pop
      stk[++r] = i;                                    // push
   }
   return g[1];
}
```

上述算法中，用一个栈 stk 来存储上凸壳中的依从左到右的次序排列的直线的下标，最

右端直线的下标在栈顶。游标 l 指向栈底，且可以依次向栈顶方向移动。每次对栈的操作显然只要 $O(1)$ 时间。其中函数 right()、gg()和 top()分别对应式（3.13）、式（3.14）和式（3.16）。

```
bool right(int p, int q, int i) {                          // 式(3.13)
   return g[p]-g[q] >= sw[i]*(st[p]-st[q]);
}
int gg(int p, int i) {                                     // 式(3.14)
   return g[p]+sw[i]*(S+st[i]-st[p]);
}
bool top(int p, int q, int i) {                            // 式(3.16)
    return st[p]*(g[i]-g[q])+st[q]*(g[p]-g[i])+st[i]*(g[q]-g[p]) >= 0;
}
```

这 3 个函数显然只要 $O(1)$ 时间。因此，从算法 batch 的主循环不难看出，整个算法需要的时间是 $O(n)$。

3-3　石子合并问题。

问题描述：在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。试设计一个算法，计算出将 n 堆石子合并成一堆的最小得分和最大得分。

算法设计：对于给定 n 堆石子，计算合并成一堆的最小得分和最大得分。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n（$1 \leqslant n \leqslant 100$），表示有 n 堆石子。第 2 行有 n 个数，分别表示每堆石子的个数。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行的数是最小得分，第 2 行中的数是最大得分。

输入文件示例	输出文件示例
input.txt	output.txt
4	43
4 4 5 9	54

分析与解答：算法实现如下。

```
int circle (int ss) {
   for (int i=1; i <= n-1; i++)
      a[n+i] = a[i];
   n = 2*n-1;
   if ((ss == 1))
      minsum(a);
   else
      maxsum(a);
   n = (n+1) / 2;
   int mm = m[1][n];
   for(i=2; i <= n; i++)
      if ((ss == 1) && (m[i][n+i-1] < mm)||(ss>1) && (m[i][n+i-1] > mm))
         mm = m[i][n+i-1];
   return(mm);
}
```

算法 circle(int ss)处理圆排列的石子合并问题。当 ss=1 时，计算最小得分；当 ss>1 时，计算最大得分。

算法 minsum 用来解直线排列的最小得分石子合并问题。

```
void minsum (vector<int> a) {
   for (int i=2; i <= n; i++)
      a[i] = a[i] + a[i-1];
   for(int r=2; r <= n; r++) {
      for(i=1; i <= n-r + 1; i++) {
         int  j = i+r-1;
         int  i1 = i+1;
         int  j1 = j;
         if((s[i][j-1] > i))
            i1 = s[i][j-1];
         if((s[i+1][j] > i))
            j1 = s[i + 1][j];
         m[i][j] = m[i][i1-1] + m[i1][j];
         s[i][j] = i1;
         for (int k = j1; k >= i1 + 1; k--) {
            int  q = m[i][k-1] + m[k][j];
            if((q < m[i][j])) {
               m[i][j] = q;
               s[i][j] = k;
            }
         }
      }
      m[i][j] = m[i][j] + a[j]-a[i-1];
   }
}
```

算法 maxsum 用来解直线排列的最大得分石子合并问题。

```
void maxsum (vector<int> a) {
   for (int i=2; i <= n; i++)
      a[i] = a[i]+a[i-1];
   for (int r=2; r <= n; r++) {
      for (i=1; i <= n-r + 1; i++) {
         int  j = i+r-1;
         if (m[i+1][j] > m[i][j-1])
            m[i][j] = m[i+1][j]+a[j]-a[i-1];
         else
             m[i][j] = m[i][j-1]+a[j]-a[i-1];
      }
   }
}
```

3-4　数字三角形问题。

问题描述：给定一个由 n 行数字组成的数字三角形，如图 3-5 所示。试设计一个算法，计算出从三角形的顶至底的一条路径，使该路径经过的数字总和最大。

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

图 3-5　数字三角形

算法设计：对于给定的由 n 行数字组成的数字三角形，计算从三角形的顶至底的路径经过的数字和的最大值。

数据输入：文件 input.txt 提供输入数据。文件的第 1 行是数字三角形的行数 n（$1 \leqslant n \leqslant 100$）。接下来的 n 行是数字三角形各行中的数字。所有数字在 0～99 之间。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行中的数是计算出的最大值。

输入文件示例	输出文件示例
input.txt	output.txt
5	30
7	
3 8	
8 1 0	
2 7 4 4	
4 5 2 6 5	

分析与解答：以自底向上的方式递归计算。

```
for(int row=n-2; row >= 0; row--) {
   for(int col=0; col <= row; col++) {
     if(triangle[row+1][col] > triangle[row+1][col+1])
        triangle[row][col] += triangle[row+1][col];
     else
        triangle[row][col] += triangle[row+1][col+1];
   }
}
```

最优值在 triangle[0][0]中。

3-5　乘法表问题。

表 3-1　∑乘法表

	a	b	C
a	b	b	A
b	c	b	A
c	a	c	c

问题描述：定义于字母表∑={a, b, c}上的乘法表如表 3-1 所示。对任一定义于∑上的字符串，适当加括号后，得到一个表达式。例如，对于字符串 x=bbbba，它的一个加括号表达式为(b(bb))(ba)。依乘法表，该表达式的值为 a。试设计一个动态规划算法，对任一定义于∑上的字符串 $x=x_1x_2\cdots x_n$，计算有多少种不同的加括号方式，使由 x 导出的加括号表达式的值为 a。

算法设计：对于给定的字符串 $x=x_1x_2\cdots x_n$，计算有多少种不同的加括号方式，使由 x 导出的加括号表达式的值为 a。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出一个字符串。

结果输出：将计算结果输出到文件 output.txt。文件的第 1 行中的数是计算出的加括号方式数。

输入文件示例	输出文件示例
input.txt	output.txt
bbbba	6

分析与解答：

```
void dyna(){
   for(int k=1; k < n; k++) {
     for(int i=0; i < nk; i++) {
        int  j = i+k;
        for(int p=i; p < j; p++){
          f[i][j][0] += f[i][p][2]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*f[p+1][j][2];
          f[i][j][1] += f[i][p][0]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*f[p+1][j][1];
```

```
            f[i][j][2] += f[i][p][1]*f[p+1][j][0]+f[i][p][2]*(f[p+1][j][1]+f[p+1][j][2]);
        }
      }
    }
}
```

最优值在 f[0][n−1][0]中。

3-6　租用游艇问题。

问题描述：长江游艇俱乐部在长江上设置了 n 个游艇出租站 1, 2, …, n。游客可在这些游艇出租站租用游艇，并在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i,j)$（$1\leqslant i<j\leqslant n$）。试设计一个算法，计算出从游艇出租站 1 到游艇出租站 n 所需的最少租金。

算法设计：对于给定的游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i,j)$（$1\leqslant i<j\leqslant n$），计算从游艇出租站 1 到游艇出租站 n 所需的最少租金。

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（$n\leqslant 200$），表示有 n 个游艇出租站。接下来的 n−1 行是 $r(i,j)$（$1\leqslant i<j\leqslant n$）。

结果输出：将计算出的从游艇出租站 1 到游艇出租站 n 所需的最少租金输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	12
5 15	
7	

分析与解答：

```
void dyna(){
   for(int k=2; k < n; k++) {
      for(int i=0; i < nk; i++) {
         int  j = i+k;
         for(int p=i+1; p < j; p++) {
            int  tmp = f[i][p]+f[p][j];
            if (f[i][j]>tmp)
               f[i][j] = tmp;
         }
      }
   }
}
```

最优值在 f[0][n−1]中。

3-7　汽车加油行驶问题。

问题描述：给定一个 $N\times N$ 的方形网格，设其左上角为起点◎，坐标为(1, 1)，X 轴向右为正，Y 轴向下为正，每个方格边长为 1。一辆汽车从起点◎出发驶向右下角终点▲，其坐标为(N, N)。在若干网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则：

（1）汽车只能沿网格边行驶，装满油后能行驶 K 条网格边。出发时汽车已装满油，在起点与终点处不设油库。

（2）当汽车行驶经过一条网格边时，若其 X 坐标或 Y 坐标减小,则应付费用 B，否则免付费用。

（3）汽车在行驶过程中遇油库则应加满油并付加油费用 A。

（4）在需要时可在网格点处增设油库，并付增设油库费用 C（不含加油费用 A）。

（5）（1）～（4）中的各数 N、K、A、B、C 均为正整数。

算法设计：求汽车从起点出发到达终点的一条所付费用最少的行驶路线。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 N、K、A、B、C 的值，$2 \leqslant N \leqslant 100$，$2 \leqslant K \leqslant 10$。第 2 行起是一个 $N \times N$ 的 0-1 方阵，每行 N 个值，至 $N+1$ 行结束。方阵的第 i 行第 j 列处的值为 1 表示在网格交叉点(i, j)处设置了一个油库，为 0 时表示未设油库。各行相邻的 2 个数以空格分隔。

结果输出：将找到的最优行驶路线所需的费用即最小费用输出到文件 output.txt。文件的第 1 行中的数是最小费用值。

输入文件示例	输出文件示例
input.txt	output.txt
9 3 2 3 6	12
0 0 0 0 1 0 0 0 0	
0 0 0 1 0 1 1 0 0	
1 0 1 0 0 0 0 1 0	
0 0 0 0 0 1 0 0 1	
1 0 0 1 0 0 1 0 0	
0 1 0 0 0 0 0 1 0	
0 0 0 0 1 0 0 0 1	
1 0 0 1 0 0 0 1 0	
0 1 0 0 0 0 0 0 0	

分析与解答：用 $f(x, y, 0)$表示汽车从网格点$(1, 1)$行驶至网格点(x, y)所需的最少费用，$f(x, y, 1)$表示汽车行驶至网格点(x, y)后还能行驶的网格边数。

建立计算 $f(x, y, 0)$和 $f(x, y, 1)$的递归式如下：

$$f(1, 1, 0)=0，\ f(1, 1, 1)=K$$

$$f(x, y, 0)=f(x, y, 0)+A，\ f(x, y, 1)=K，\ (x, y)\text{是油库}$$

$$f(x, y, 0)=f(x, y, 0)+C+A，\ f(x, y, 1)=K，\ (x, y)\text{是非油库且}f(x, y, 1)=0$$

$$f(x, y, 0)=\min_{0 \leqslant i \leqslant 4}\{f(x+s[i][0], y+s[i][1], 0)+s[i][2]\}$$

$$f(x, y, 1)=f(x+s[j][0], y+s[j][1], 1)-1$$

式中，数组 $s=\{\{-1, 0, 0\}, \{0, -1, 0\}, \{1, 0, B\}, \{0, 1, B\}\}$。

用备忘录方法递归计算。$f(n, n, 0)$为最优值。

3-8　最小 m 段和问题。

问题描述：给定 n 个整数组成的序列，现在要求将序列分割为 m 段，每段子序列中的数在原序列中连续排列。如何分割才能使这 m 段子序列的和的最大值达到最小？

算法设计：给定 n 个整数组成的序列，计算该序列的最优 m 段分割，使 m 段子序列的和的最大值达到最小。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 m。正整数 n 是序列的长度；正整数 m 是分割的段数。接下来的一行中有 n 个整数。

结果输出：将计算结果输出到文件 output.txt。文件的第 1 行中的数是计算出的 m 段子序列的和的最大值的最小值。

输入文件示例	输出文件示例
input.txt	output.txt
1 1	10
10	

分析与解答：

```
void solve(int n, int m) {
   int  i, j, k, temp, maxt;
   for(i=1; i <= n; i++)
     f[i][1] = f[i-1][1]+t[i];
   for(j=2; j <= m; j++) {
     for(i=j; i <= n; i++) {
       for(k=1, temp=INT_MAX; k < i; k++) {
         maxt=max(f[i][1]-f[k][1],f[k][j-1]);
         if(temp > maxt)
           temp = maxt;
       }
       f[i][j]=temp;
     }
   }
}
```

最优值在 f[n][m]中。

```
void main() {
   int  n, m;
   cin>> n >> m;
   if ((n < m) || (n == 0)) {
     cout << 0 << endl;
     return;
   }
   f.resize(n+1, m+1);
   t.resize(n+1);
   for(int i=1; i <= n; i++)
     cin >> t[i];
   solve(n, m);
   cout << f[n][m]<<endl;
}
```

3-9 圈乘运算问题。

问题描述：关于整数的二元圈乘运算⊗定义为

(X⊗ Y)=十进制整数 X 的各位数字之和×十进制整数 Y 的最大数字+Y 的最小数字

例如，(9⊗30)=9×3+0=27。

对于给定的十进制整数 X 和 K，由 X 和⊗运算可以组成各种不同的表达式。试设计一个算法，计算出由 X 和⊗运算组成的值为 K 的表达式最少需用多少个⊗运算。

算法设计：给定十进制整数 X 和 K（$1 \leqslant X, K \leqslant 10^{20}$），计算由 X 和⊗运算组成的值为 K 的表达式最少需用多少个⊗运算。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每行有 2 个十进制整数 X 和 K。最后一行是 0 0。

结果输出：将找到的最少⊗运算个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 12	1
0 0	

分析与解答：

（1）⊗运算一般不满足交换律和结合律。

（2）最优子结构性质：设⊗运算表达式(X⊗Y)=K 用了最少的⊗运算，则 X 和 Y 也用了最少的⊗运算。

（3）⊗运算表达式值的有限性：对于给定的 N，其十进制位数为 $m=[\log_{10}(N+1)]$。易知，当 $m>1$ 时，⊗运算表达式最大值不超过 $81\times m+9$；当 $m=1$ 时，⊗运算表达式最大值不超过 171。

对于给定的 N，⊗运算表达式的值域为 $S(N)$，则

$$S(N)\subseteq[1:L]\cup\{N\}$$

式中，

$$L=81\times\max\{2,\lceil\log_{10}(N+1)\rceil\}+9$$

因此，对于给定的 N，⊗运算表达式最大值为 $O(\log N)$量级。

（4）从 N 出发，反复用 N 和⊗运算可求得 $S(N)$中每个数。在计算过程中，用动态规划算法求 $S(N)$中每个数所用的最少⊗运算次数。

（5）数据结构：用数组 num[0:L][0:3]存储 $S(N)$中每个数的相关信息。

对于任意正整数 x，用 sum(x)、max(x)和 min(x)分别记其各位数字之和、最大数字和最小数字。num[i][0]存储 i 所需的最少⊗运算次数，num[i][1]存储 min(i)，num[i][2]存储 max(i)，num[i][3]存储 sum(i)。min(N)、max(N)和 sum(N)分别存储于 num[0][1]、num[0][2]和 num[0][3]中。

（6）算法实现。

先由 input 输入 N 和 K。计算 L，当 $K>L$ 时，明显无解。

```
int input() {
   cout << "N = ";
   cin >> s1;
   cout << "K = ";
   cin >> s2;
   if(strcmp(s1, s2) == 0) {
      out(0);
      return 0;
   }
   len = strlen(s1);
   big = 81*len+9;
   if(big < 171)
      big = 171;
   int  biglen = ceil(log10(big));
```

```
  if (strlen(s2) > biglen) {
    out(-1);
    return 0;
  }
  kk=atoi(s2);
  if(kk > big) {
    out(-1);
    return 0;
  }
  return 1;
}
```

然后，由 init()函数初始化数组 num。

```
void init() {
  Make2DArray(num, ++big, 4);
  for(int i=0; i < big; i++){
    num[i][0]=INT_MAX;
    num[i][1]=INT_MAX;
    num[i][2]=0;num[i][3]=0;
  }
  for(i=0; i < len; i++) {
    count(ctoi(s1[i]), 0);
    num[0][0] = 0;
    for(i=1; i < big; i++) {
      int  t = i;
      while(t > 0) {
        int  j = t%10;
        t /= 10;
        count(j, i);
      }
    }
  }
}
```

其中用到的函数 ctoi()、count()和 out()如下。

```
int ctoi(char x) {
  return (int)x-48;
}
void count(int i, int j) {
  if(i < num[j][1])
    num[j][1] = i;
  if(i > num[j][2])
    num[j][2] = i;
  num[j][3]+=i;
}
void out(int x) {
  if(x>=0)
    cout<<"The number of ⊗ is " << x << endl;
  else
```

```
    cout << "No answer!" << endl;
}
```

算法 compute 从 N 出发，反复用 N 和⊗运算求 $S(N)$中的每个数。在计算过程中，用动态规划算法求 $S(N)$中的每个数所用的最少⊗运算次数。

```
void compute() {
   bool  flag = true;
   while(flag) {
      flag = false;
      for(int i=0; i < big; i++) {
         if(num[i][0] < INT_MAX) {
            for(int k=0; k < big; k++) {
               if(num[k][0] < INT_MAX) {
                  int j = num[i][3]*num[k][2] + num[k][1];
                  if(num[i][0]+num[k][0]+1 < num[j][0]) {
                     num[j][0] = num[i][0]+num[k][0]+1;
                     flag = true;
                  }
               }
            }
         }
      }
   }
   if(num[kk][0] < INT_MAX)
      out(num[kk][0]);
   else
      out(-1);
}
```

主函数 main()如下。

```
int  kk, len, big, **num;
char  s1[20], s2[20];
void main() {
   if(input())
      init();
   else
      return;
   compute();
}
```

（7）算法的计算复杂性。

算法所需空间显然为 $O(\log N)$。

算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的 1 次 while 循环需要 $O(\log^2 N)$的计算时间。在最坏情况下需要 $O(\log N)$次 while 循环。因此,算法在最坏情况下所需的计算时间为 $O(\log^3 N)$。

（8）算法的改进。

设 x 和 y 是两个正整数，当 $\min(x)=\min(y)$，$\max(x)=\max(y)$且 $\text{sum}(x)=\text{sum}(y)$时，称 x 和 y 是⊗等价的。由⊗运算的定义知，当 x 和 y 是⊗等价时，对于任意正整数 z 有$(x\otimes z)=(y\otimes z)$

且$(z\otimes x)=(z\otimes y)$。由此可见，在$S(N)$中只要关注⊗等价类中⊗运算次数最少的数。

⊗等价类可按min()、max()、sum()函数进行分类。对于$S(N)$中任一数x，有

$$0\leqslant \min(x), \max(x)\leqslant 9 \qquad 1\leqslant \mathrm{sum}(x)\leqslant \mathrm{LL}$$

式中，$\mathrm{LL}=9\times\lceil\log_{10}(L+1)\rceil$。

因此，可以用两个数组leftn[1:LL]和rightn[0:9][0:]来存储$S(N)$中的⊗等价类，并由此构造出$S(N)$中所有数。

输入N和K后，数组leftn和rightn由init初始化如下。

```
void init() {
   biglen *= 9;
   Make2DArray(rightn, 10, 10);
   for(int i=0; i < 10; i++) {
      for(int j=0; j < 10; j++) {
         rightn[i][j] = INT_MAX;
         leftn = new int[biglen+1];
         for(i=1; i <= biglen; i++)
            leftn[i] = INT_MAX;
         int a = INT_MAX, b = 0;
         sum = 0;
         leftn[0] = 0;
         for(i=0; i < len; i++)
            count(ctoi(s1[i]), a, b, sum);
         rightn[a][b] = 0;
         if(sum <= biglen)
            leftn[sum] = 0;
      }
   }
}
```

其中，用函数count()和trans()来计算min(i)、max(i)和sum(i)。

```
void count(int i, int &a, int &b, int &c) {
   if(i < a)
      a = i;
   if(i > b)
      b = i;
   c+=i;
}
void trans(int t, int &a, int &b, int &c) {
   a = INT_MAX;
   b = 0;
   c = 0;
   while(t > 0) {
      int  j = t%10;
      t /= 10;
      count(j, a, b, c);
   }
}
```

用⊗等价类思想实现的动态规划算法描述如下。

```
void compute() {
   int  a, b, c, best = INT_MAX;
   bool  flag = true;
   while(flag) {
      flag = false;
      for(int i=0; i <= biglen; i++) {
         if(leftn[i] < INT_MAX) {
            for(int j=0; j < 10; j++) {
               for(int k=0; k <= j; k++) {
                  if(rightn[k][j] < INT_MAX) {
                     int  num = i>0 ? i*j+k : sum*j+k;
                     trans(num, a, b, c);
                     int  curr = leftn[i]+rightn[k][j]+1;
                     if(curr < leftn[c]) {
                        leftn[c] = curr;
                        flag = true;
                     }
                     if(curr < rightn[a][b]) {
                        rightn[a][b] = curr;
                        flag = true;
                     }
                     if(num == kk && curr < best) {
                        best = curr;
                        flag = true;
                     }
                  }
               }
            }
         }
      }
   }
   if(best < INT_MAX)
      out(best);
   else
      out(-1);
}
```

完成整个计算的主函数 main()如下。

```
int  kk, len, biglen, sum, *leftn, **rightn;
char  s1[20], s2[20];
void main() {
   if(input())
      init();
   else
      return;
   compute();
}
```

（9）改进算法的计算复杂性。

改进算法所需空间显然为 $O(\log\log N)$。

改进算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的 1 次 while 循环需要 $O(\log\log N)$的计算时间。在最坏情况下需要 $O(\log N)$次 while 循环。因此，算法在最坏情况下所需的计算时间为 $O(\log N\log\log N)$。

可以看到，新算法的计算复杂性有十分显著的改进。

3-10　最大长方体问题。

问题描述：一个长、宽、高分别为 m、n、p 的长方体被分割成 $m\times n\times p$ 个小立方体。每个小立方体内有一个整数。试设计一个算法，计算所给长方体的最大子长方体。子长方体的大小由它所含所有整数之和确定。

算法设计：对于给定的长、宽、高分别为 m、n、p 的长方体，计算最大子长方体的大小。

数据输入：文件 input.txt 提供输入数据，第 1 行是 3 个正整数 m、n、p（$1\leq m, n, p\leq 50$）。在接下来的 $m\times n$ 行中每行 p 个正整数，表示小立方体中的数。

结果输出：将计算结果输出到文件 output.txt。文件的第 1 行中的数是计算出的最大子长方体的大小。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 3	14
0 −1 2	
1 2 2	
1 1 −2	
−2 −1 −1	
−3 3 −2	
−2 −3 1	
−2 3 3	
0 1 3	
2 1 −3	

分析与解答：在最大子矩阵和问题的动态规划算法基础上，容易设计解此问题的动态规划算法如下。

```
template <class T>
T maxSum3(const vector< matrix<T> > & a) {
   T  max, sum = 0;
   int  m = a.size(), n = a[0].rows(), p = a[0].cols();
   matrix<int> b(n, p, 0);
   for(int i=0; i < m; i++) {
      for(int k=0; k < n; k++) {
         for(int t=0; t < p; t++)
            b[k][t] = 0;
         for(int j=i; j < m; j++) {
            for(int k=0; k < n; k++) {
               for(int t=0; t < p; t++) {
                  b[k][t] += a[j][k][t];
```

```
                max = maxSum2(b);
                if(max > sum)
                    sum = max;
            }
        }
    }
  }
 }
 return sum;
}
```

3-11　正则表达式匹配问题。

问题描述：许多操作系统采用正则表达式实现文件匹配功能。一种简单的正则表达式由英文字母、数字及通配符“*”和“?”组成。“?”代表任意一个字符，“*”则可以代表任意多个字符。现要用正则表达式对部分文件进行操作。

试设计一个算法，找出一个正则表达式，使其能匹配的待操作文件最多，但不能匹配任何不进行操作的文件。所找出的正则表达式的长度还应是最短的。

算法设计：对于给定的待操作文件，找出一个能匹配最多待操作文件的正则表达式。

数据输入：由文件 input.txt 提供输入数据。文件由 n（$1 \leqslant n \leqslant 250$）行组成。每行给出一个文件名。文件名由英文字母和数字组成。英文字符要区分大小写，文件名长度不超过 8 个字符。文件名后是一个空格符和一个字符“+”或“−”。“+”表示要对该行给出的文件进行操作，“−”表示不进行操作。

结果输出：将计算出的最多文件匹配数和最优正则表达式输出到文件 output.txt。文件第 1 行中的数是计算出的最多文件匹配数，第 2 行是最优正则表达式。

输入文件示例	输出文件示例
input.txt	output.txt
EXCHANGE +	3
EXTRA +	*A*
HARDWARE +	
MOUSE −	
NETWORK −	

分析与解答：设当前考察的正则表达式为 s，当前期考察的文件为 f。用 match(i, j)表示 $s[1..i]$与 $f[1..j]$的匹配情况。当 $s[1..i]$能匹配 $f[1..j]$时，match(i, j)=1，否则 match(i, j)=0。显然，可用下面的递归式计算 match(i, j)。

$$\text{match}(i,j)\begin{cases}1\begin{cases}\text{match}(i-1,j-1)=1 & s[i]='?'\\ \text{match}(i-1,j-1)=1 & s[i]=f[i]\\ \text{match}(i-1,k)=1 & s[i]='*'\end{cases}\\ 0\end{cases}$$

据此可设计求最优匹配的回溯法如下。

```
void search(int len) {
  if((currmat > maxmat || currmat == maxmat && len < minlen) && ok(len)) {
    maxmat=currmat;
```

```
      minmat = s;
      minlen = len;
    }
    len++;
    if(len == 1 || s[len1] != '*') {
      s[len]='?';
      if(check(len))
        search(len);
      s[len]='*';
      if(check(len))
        search(len);
    }
    for(int i=1; i <= p[len-1]; i++) {
      s[len] = cha[len-1][i].c;
      if(check(len))
        search(len);
    }
}
```

check()函数计算当前匹配情况，ok()函数判定是否匹配非操作文件。

```
bool check(int len) {
    int  i, j, t, k = 0;
    currmat = 0;
    for(i=1; i <= n[0]; i++) {
      memset(&match[len][i], 0, sizeof(match[len][i]));
      if(len == 1 && s[1] == '*')
        match[len][i][0] = 1;
      for(j=1; j <= f[i].length(); j++) {
        switch(s[len]) {
          case '*':
                    for(t=0; t <= j; t++) {
                      if(match[len1][i][t] == 1) {
                        match[len][i][j] = 1;
                        break;
                      }
                    }
                    break;
          case '?':
                    match[len][i][j] = match[len-1][i][j-1];
                    break;
          default:
                    if(s[len] == f[i][j-1])
                      match[len][i][j] = match[len-1][i][j-1];
                    break;
        }
        for(j=f[i].length(); j >= 1; j--) {
          if(match[len][i][j] == 1) {
            k++;
            if(j == f[i].length())
```

```
                currmat++;
              break;
            }
          }
        }
      }
      if (k<maxmat || k==maxmat && len>=minlen)
        return 0;
      p[len]=0;
      for(i=1; i <= n[0]; i++)
        for(j=1; j <= f[i].length()-1; j++)
          if (match[len][i][j] == 1)
            save(f[i][j],len);
      return 1;
}
bool ok(int len) {
      int  i, j, k, t;
      for(k=1; k <= len; k++) {
        for(i=n[0]+1; i <= n[0]+n[1]; i++) {
          memset(&match[k][i], 0, sizeof(match[k][i]));
          if(s[1] == '*' && k == 1)
            match[k][i][0] = 1;
          for(j=1; j <= f[i].length(); j++) {
            switch(s[k]) {
              case '*':
                        for (t=0; t <= j; t++) {
                          if (match[k-1][i][t] == 1) {
                            match[k][i][j] = 1;
                            break;
                          }
                        }
                        break;
              case '?':
                        match[k][i][j] = match[k-1][i][j-1];
                        break;
              default:
                        if (s[k] == f[i][j-1])
                          match[k][i][j] = match[k-1][i][j-1];
                        break;
            }
          }
        }
      }
      for(i=n[0]+1; i <= n[0]+n[1]; i++)
        if(match[len][i][f[i].length()] == 1)
          return 0;
      return 1;
}
```

算法的主函数如下。

```
int main() {
   readin();
   search(0);
   out();
   return 0;
}
```

readin()函数用于读入数据并初始化。

```
void readin() {
   string  k[MAXN+1], str;
   char  chr;
   n[0] = 0;
   n[1] = 0;
   p[0] = 0;
   while(1) {
      fin >> str >> chr;
      if(str.length() == 0)
         break;
      if (chr == '+') {
         f[++n[0]] = str;
         save(str[0], 0);
      }
      else
         k[++n[1]] = str;
   }
   for(int i=1; i <= n[1]; i++)
      f[n[0]+i] = k[i];
   memset(match, 0, sizeof(match));
   for(i=1; i <= n[0]+n[1]; i++)
      match[0][i][0] = 1;
   maxmat = 0;
   minlen = 255;
}
```

save 对操作文件名中出现的字符按出现频率排序存储，以加快搜索进程。

```
void save(char c, int len) {
   for(int i=1; i <= p[len]; i++) {
      if(cha[len][i].c == c) {
         cha[len][i].f++;
         cha[len][0] = cha[len][i];
         int  j = i;
         while(cha[len][j-1].f < cha[len][0].f) {
            cha[len][j]=cha[len][j-1];
            j--;
         }
         cha[len][j] = cha[len][0];
         return;
      }
```

```
    }
    cha[len][++p[len]].c = c;
    cha[len][p[len]].f = 1;
}
```

3-12 双调旅行售货员问题。

问题描述：欧氏旅行售货员问题是对给定的平面上 n 个点确定一条连接这 n 个点的长度最短的哈密顿回路。欧氏距离满足三角不等式，所以欧氏旅行售货员问题是一个特殊的具有三角不等式性质的旅行售货员问题，仍是一个 NP 完全问题。最短双调 TSP 回路是欧氏旅行售货员问题的特殊情况。平面上 n 个点的双调 TSP 回路是从最左点开始，严格地由左至右直到最右点，然后严格地由右至左直至最左点，且连接每个点恰好一次的一条闭合回路。

算法设计：给定平面上 n 个点，计算这 n 个点的最短双调 TSP 回路。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示给定的平面上的点数。在接下来的 n 行中，每行 2 个实数，分别表示点的 x 坐标和 y 坐标。

结果输出：将计算的最短双调 TSP 回路的长度（保留 2 位小数）输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7	25.58
0 6	
1 0	
2 3	
5 4	
6 1	
7 5	
8 2	

分析与解答：先将给定的平面上 n 个点依其 x 坐标排序。设排好序的 n 个点为 $p_i=(x_i, y_i)$（$i=1, 2, \cdots, n$）。点集 $\{p_1, p_2, \cdots, p_i\}$ 的最短双调 TSP 回路的长度记作 $t(i)$。容易证明，该问题具有最优子结构性质。$t(i)$ 满足如下动态规划递归式：

$$t(i)=\min_{1<k<i}\{t(k)+D(k,i)+d(k-1,i)-d(k-1,k)\}$$

$$t(1)=0, t(2)=2d(1,2)$$

式中，

$$d(i,j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

$$D(i,j)=\sum_{k=i+1}^{j} d(k-1,k)$$

设 $s(i)=\sum_{k=1}^{i} d(k-1,k)$，则

$$D(k,i)=s(i)-s(k), \qquad d(k-1,k)=s(k)-s(k-1)$$

上述递归式可进一步简化为

$$t(i)=\min_{1<k<i}\{t(k)+s(i)+s(k-1)-2s(k)+d(k-1,i)\}$$

按此递归式计算出的 $t(n)$ 为最优值。算法所需的计算时间为 $O(n^2)$。

```
void init() {
    fin >> n;
    point.resize(n);
```

```
    for(int i = 0; i < n; i++) {
        point[i].resize(2);
        fin >> point[i][0] >> point[i][1];
    }
    sort(point.begin(), point.end(), vless);
}
void dynamic() {
    s[1]= 0;
    for (int i=2; i <= n; i++)
        s[i] = dist(i-1, i)+s[i-1];
    t[2] = 2*s[2];
    for(i=3; i <= n; i++) {
        t[i] = t[2]+s[i]-2*s[2]+dist(i, 1);
        for(int j=2; j <= i-2; j++) {
            double  temp = t[j+1]+s[i]+s[j]-2*s[j+1]+dist(i, j);
            if(t[i] > temp)
                t[i] = temp;
        }
    }
}
```

dist()函数用于计算两点之间的距离。

```
double dist(int p, int q) {
    return sqrt(pow(point[p-1][0] - point[q-1][0], 2)+ pow(point[p-1][1] - point[q-1][1], 2));
}
```

3-13　最大 k 乘积问题。

问题描述：设 I 是一个 n 位十进制整数。如果将 I 划分为 k 段，则可得到 k 个整数。这 k 个整数的乘积称为 I 的一个 k 乘积。试设计一个算法，对于给定的 I 和 k，求出 I 的最大 k 乘积。

算法设计：对于给定的 I 和 k，计算 I 的最大 k 乘积。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 k。正整数 n 是序列的长度，正整数 k 是分割的段数。接下来的一行中是一个 n 位十进制整数（$n \leq 10$）。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行中的数是计算出的最大 k 乘积。

输入文件示例	输出文件示例
input.txt	output.txt
2 1	15
15	

分析与解答：设 $I(s, t)$ 是 I 的由从 s 位开始的 t 位数字组成的十进制数。$f(i, j)$ 表示 $I(0, I)$ 的最大 j 乘积，则 $f(i, j)$ 具有最优子结构性质。计算 $f(i, j)$ 的动态规划递归式如下：

$$f(i, j) = \max_{1 \leq k \leq i} \{f(k, j-1) * I(k, i-k)\}$$

据此可设计最大 k 乘积问题的动态规划算法如下。

```
void solve(int n, int m) {
    int  i, j, k;
    int  temp, maxt, tk;
    for(i=1; i<=n; i++)
        f[i][1] = conv(0, i);
```

```
  for(j=2; j<= m; j++) {
    for(i=j; i <= n; i++) {
      for(k=1, temp=0; k < i; k++) {
        maxt = f[k][j-1]*conv(k, i-k);
        if(temp < maxt) {
          temp = maxt;
          tk = k;
        }
      }
    }
    f[i][j] = temp;
    ka[i][j] = tk;
  }
}
```

conv 计算 $I(s, t)$的值。

```
int conv(int i,int j) {
  string  str1 = str.substr(i, j);
  return atoi(str1.c_str());
}
```

实现算法的主函数如下。

```
int main() {
  int  n, m;
  ifstream cin1("kmul.in");
  cin1 >> n >> m;
  if((n < m) || (n == 0)) {
    cout << 0 << endl;
    return;
  }
  cin1 >> str;
  if(n != str.size()) {
    cout << 0 << endl;
    return;
  }
  f.resize(n+1, m+1);
  ka.resize(n+1, m+1);
  solve(n, m);
  out(n, m);
  return 0;
}
```

out 输出计算结果。

```
void out(int n, int m) {
  ofstream cout("kmul.out");
  cout << f[n][m] << endl;
  for(int i=m, k=n; i >= 1 && k > 0;k = ka[k][i], i--)
    cout << "f[" << k << "][" << i << "]=" << f[k][i] << endl;
}
```

3-14　最少费用购物问题。

问题描述：商店中每种商品都有标价。例如，一朵花的价格是 2 元，一个花瓶的价格是 5 元。为了吸引顾客，商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组，并降价销售。例如，3 朵花的价格不是 6 元而是 5 元，2 个花瓶加 1 朵花的优惠价是 10 元。试设计一个算法，计算出某顾客所购商品应付的最少费用。

算法设计：对于给定欲购商品的价格和数量，以及优惠商品价，计算所购商品应付的最少费用。

数据输入：由文件 input.txt 提供欲购商品数据。文件的第 1 行中有 1 个整数 $B(0\leqslant B\leqslant 5)$，表示所购商品种类数。在接下来的 B 行中，每行有 3 个数 C、K 和 P。C 表示商品的编码（每种商品有唯一编码），$1\leqslant C\leqslant 999$；$K$ 表示购买该种商品总数，$1\leqslant K\leqslant 5$；P 是该种商品的正常单价（每件商品的价格），$1\leqslant P\leqslant 999$。注意，一次最多可购买 5×5=25 件商品。

由文件 offer.txt 提供优惠商品价数据。文件的第 1 行中有 1 个整数 S（$0\leqslant S\leqslant 99$），表示共有 S 种优惠商品组合。接下来的 S 行，每行的第 1 个数描述优惠商品组合中商品的种类数 j。接着是 j 个数字对（C、K），其中 C 是商品编码，$1\leqslant C\leqslant 999$；$K$ 表示该种商品在此组合中的数量，$1\leqslant K\leqslant 5$。每行最后一个数字 P（$1\leqslant P\leqslant 9999$）表示此商品组合的优惠价。

结果输出：将计算出的所购商品应付的最少费用输出到文件 output.txt。

输入文件示例		输出文件示例
input.txt	offer.txt	output.txt
2	2	14
7 3 2	1 7 3 5	
8 2 5	2 7 1 8 2 10	

分析与解答：设 cost(a, b, c, d, e)表示购买商品组合(a, b, c, d, e)需要的最少费用。$A[k]$、$B[k]$、$C[k]$、$D[k]$、$E[k]$表示第 k 种优惠方案的商品组合。offer(m)是第 m 种优惠方案的价格。如果 cost(a, b, c, d, e)使用了第 m 种优惠方案，则

$$\text{cost}(a, b, c, d, e)=\text{cost}(a-A[m], b-B[m], c-C[m], d-D[m], e-E[m])+\text{offer}(m)$$

即该问题具有最优子结构性质。按此递归式，容易设计解此问题的动态规划算法如下。

```
void minicost() {
   int  i, j, k, m, n, p, minm;
   minm = 0;
   for(i=1; i <= b; i++)
      minm += product[i]*purch[i].price;
   for(p=1; p <= s; p++) {
      i=product[1] offer[p][1];
      j=product[2] offer[p][2];
      k=product[3] offer[p][3];
      m=product[4] offer[p][4];
      n=product[5] offer[p][5];
      if(i >= 0 && j >= 0 && k >= 0 && m >= 0 && n >= 0 && (cost[i][j][k][m][n]+offer[p][0] < minm))
         minm = cost[i][j][k][m][n]+offer[p][0];
   }
   cost[product[1]][product[2]][product[3]][product[4]][product[5]] = minm;
}
```

其中，product[i]是购买第 i 种商品的数量，由 comp 迭代计算，init 进行初始化计算。

```
void comp(int i) {
    if(i > b) {
      minicost();
      return;
    }
    for(int j=0; j <= purch[i].piece; j++) {
      product[i] = j;
      comp(i+1);
    }
}
void init() {
    int  i, j, n, p, t, code;
    for(i=0; i < 100; i++)
      for(j=0; j < 6; j++)
        offer[i][j] = 0;
    for(i=0; i < 6; i++) {
      purch[i].piece = 0;
      purch[i].price = 0;
      product[i] = 0;
    }
    fin >> b;
    for(i=1; i <= b; i++) {
      fin >> code;
      fin >> purch[i].piece >> purch[i].price;
      num[code] = i;
    }
    fin1 >> s;
    for(i=1; i <= s; i++) {
      fin1 >> t;
      for(j=1; j <= t; j++) {
        fin1 >> n >> p;
        offer[i][num[n]] = p;
      }
      fin1 >> offer[i][0];
    }
}
```

实现算法的主函数如下。

```
int main() {
    init();
    comp(1);
    out();
    return 0;
}
```

3-15　收集样本问题。

问题描述：机器人 Rob 在一个有 $n \times n$ 个方格的方形区域 F 中收集样本。(i, j)方格中样本的价值为 $v(i, j)$，如图 3-6 所示。Rob 从方形区域 F 的左上角 A 点出发，向下或向右行走，

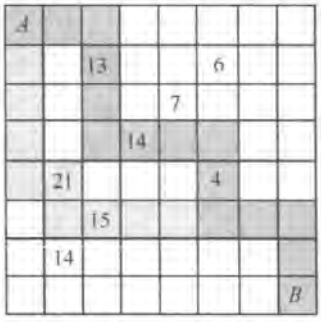

图 3-6　$n \times n$ 个方格的方形区域 F

直到右下角的 B 点，在走过的路上，收集方格中的样本。Rob 从 A 点到 B 点共走 2 次，试找出 Rob 的 2 条行走路径，使其取得的样本总价值最大。

算法设计：给定方形区域 F 中的样本分布，计算 Rob 的 2 条行走路径，使其取得的样本总价值最大。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示方形区域 F 有 $n \times n$ 个方格。接下来每行有 3 个整数，前 2 个数表示方格位置，第 3 个数为该位置样本价值。最后一行是 3 个 0。

结果输出：将计算的最大样本总价值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
8	67
2 3 13	
2 6 6	
3 5 7	
4 4 14	
5 2 21	
5 6 4	
6 3 15	
7 2 14	
0 0 0	

分析与解答：设两次行走等长距离，到达 (x_1, y_1) 和 (x_2, y_2) 处取得的最大价值为 $h[x_1][y_1][x_2][y_2]$。容易证明其具有最优子结构性质。动态规划算法如下。

```
void dyna() {
   int  x1, y1, x2, y2, s, v;
   for(int i=0; i <= n; i++)
      for(int j=0; j <= n; j++)
         for(int k=0; k <= n; k++)
            for(int l=0; l <= n; l++)
               h[i][j][k][l] = 0;
   h[1][1][1][1] = g[1][1];
   for(s=2; s <= n+n-1; s++) {
      for(x1=1; x1 <= s-1; x1++) {
         for(x2=1; x2 <= s-1; x2++) {
            y1 = s-x1;
            y2 = s-x2;
            v = h[x1][y1][x2][y2];
            val(x1+1, y1, x2+1, y2, v);
            val(x1+1, y1, x2, y2+1, v);
            val(x1, y1+1, x2+1, y2, v);
            val(x1, y1+1, x2, y2+1, v);
         }
      }
   }
```

```
}
```

其中，val()函数用于动态更新最优值。

```
void val(int x1, int y1, int x2, int y2, int v) {
  if(x1 == x2 && y1 == y2)
    h[x1][y1][x2][y2] = max(h[x1][y1][x2][y2], v+g[x1][y1]);
  else
    h[x1][y1][x2][y2] = max(h[x1][y1][x2][y2], v+g[x1][y1]+g[x2][y2]);
}
```

g[*i*][*j*]是方格(*i*, *j*)处样本的价值。

3-16　最优时间表问题。

问题描述：一台精密仪器的工作时间为 n 个时间单位。与仪器工作时间同步进行若干仪器维修程序。一旦启动维修程序，仪器必须进入维修程序。如果只有一个维修程序启动，则必须进入该维修程序。如果在同一时刻有多个维修程序，可任选进入其中的一个维修程序。维修程序必须从头开始，不能从中间插入。一个维修程序从第 s 个时间单位开始，持续 t 个时间单位，则该维修程序在第 $s+t-1$ 个时间单位结束。为了提高仪器使用率，希望安排尽可能短的维修时间。

算法设计：对于给定的维修程序时间表,计算最优时间表。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。n 表示仪器的工作时间单位，k 是维修程序数。在接下来的 k 行中，每行有 2 个表示维修程序的整数 s 和 t，该维修程序从第 s 个时间单位开始，持续 t 个时间单位。

结果输出：将计算出的最短维修时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
15 6	11
1 2	
1 6	
4 11	
8 5	
8 1	
11 5	

分析与解答：设 mt(i)表示从第 i 个时间单位开始到第 n 个时间单位结束的最短维修时间，则 mt(i)具有最优子结构性质，且满足如下递归式：

当时间单位 i 有多个维修程序时，$\mathrm{mt}(i)=\min_{s_j=i}\{\mathrm{mt}(i+t_j)\}$；

当时间单位 i 没有维修程序时，mt(i)=mt(i+1)−1。

初始值为 mt(n+1)=n。

3-17　字符串比较问题。

问题描述：对于长度相同的两个字符串 A 和 B，其距离定义为相应位置字符距离之和。两个非空格字符的距离是它们的 ASCII 编码之差的绝对值。空格与空格的距离为 0，空格与其他字符的距离为一定值 k。

在一般情况下，字符串 A 和 B 的长度不一定相同。字符串 A 的扩展是在 A 中插入若干空格字符所产生的字符串。在字符串 A 和 B 的所有长度相同的扩展中，有一对距离最小的

扩展，该距离称为字符串 A 和 B 的扩展距离。

对于给定的字符串 A 和 B，试设计一个算法，计算其扩展距离。

算法设计：对于给定的字符串 A 和 B，计算其扩展距离。

数据输入：由文件 input.txt 给出输入数据。第 1 行是字符串 A，第 2 行是字符串 B，第 3 行是空格与其他字符的距离定值 k。

结果输出：将计算出的字符串 A 和 B 的扩展距离输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
cmc	10
snmn	
2	

分析与解答：设字符串 A 和 B 的子串 $A[1..i]$和 $B[1..j]$的扩展距离为 $\mathrm{val}(i,j)$，则 $\mathrm{val}(i,j)$ 具有最优子结构性质，且满足如下递归式：

$$\mathrm{val}(i,j)=\min\{\mathrm{val}(i-1,j)+k,\ \mathrm{val}(i,j-1)+k,\ \mathrm{val}(i-1,j-1)+\mathrm{dist}(a_i,b_j)\}$$

据此容易设计动态规划算法如下。

```
int comp() {
   int  i, j, tmp, len1, len2;
   val[0][0] = 0;
   len1 = strlen(str1);
   len2 = strlen(str2);
   for(i=0; i <= len1; i++) {
      for(j=0; j <= len2; j++) {
         if(i+j) {
            val[i][j]=MAXINT;
            if((i*j) && (tmp=val[i-1][j-1]+dist(str1[i-1], str2[j-1])) < val[i][j])
               val[i][j] = tmp;
            if(i && (tmp = val[i-1][j]+dist(str1[i-1],' ')) < val[i][j])
               val[i][j] = tmp;
            if(j && (tmp = val[i][j-1]+dist(str2[j-1],' ')) < val[i][j])
               val[i][j] = tmp;
         }
      }
   }
   return val[len1][len2];
}
int main() {
   readin();
   cout<<comp() << endl;
   return 0;
}
```

从动态规划递归式可知，算法的计算时间为 $O(mn)$，m 和 n 分别是字符串 A 和 B 的长度。

3-18　有向树 k 中值问题。

问题描述：给定一棵有向树 T，树 T 中每个顶点 u 都有一个权 $w(u)$，树的每条边(u,v)也都有一个非负边长 $d(u,v)$。有向树 T 的每个顶点 u 可以看作客户，其服务需求量为 $w(u)$。

每条边(u, v)的边长$d(u, v)$可以看作运输费用。如果在顶点u处未设置服务机构，则将顶点u处的服务需求沿有向树的边(u, v)转移到顶点v处服务机构所需付出的服务转移费用为$w(u)\cdot d(u, v)$。树根处已设置了服务机构，现在要在树T中增设k处服务机构，使得整棵树T的服务转移费用最小。

算法设计：对于给定的有向树T，计算在树T中增设k处服务机构的最小服务转移费用。

数据输入：由文件input.txt给出输入数据。第1行有2个正整数n和k。n表示有向树T的边数，k是要增设的服务机构数。有向树T的顶点编号为$0, 1, \cdots, n$。根结点编号为0。在接下来的n行中，每行有表示有向树T的一条有向边的3个整数。第i+1行的3个整数w_i、v_i、d_i分别表示编号为i的顶点的权为w_i，相应的有向边为(i, v_i)，其边长为d_i。

结果输出：将计算的最小服务转移费用输出到文件output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 2	4
1 0 1	
1 1 10	
10 2 5	
1 2 3	

分析与解答：要求有向树T的k个顶点组成的集合F，使$\text{cost}(F)=\sum_{x\in T}\min_{u\in F+r} w(x)\cdot d(x,u)$达到最小。在一般情况下，有向树$T$是一棵多叉树。为便于计算，可以将$T$变换为与之等价的如下二叉树。将每个顶点的第一个儿子顶点作为其父顶点的左儿子顶点，同时增加一个0权0边长的附加顶点作为右儿子顶点。然后对于其他儿子顶点以相同方式作为新增附加顶点的左儿子顶点，一直继续下去，直至处理完所有顶点。所得到的二叉树与树T具有相同的最小耗费。在下面的讨论中，假设T是二叉树。

设T的以顶点x为根的子树为T_x，其左、右儿子顶点分别为y和z。F中距顶点x最近的顶点为u。在以顶点x为根的子树T_x中取j个F中顶点的最小耗费记为$\text{cost}(x, u, j)$。

这个问题是树型动态规划问题的典型例子。容易证明，$\text{cost}(x, u, j)$具有最优子结构性质，它满足如下递归式：

$$\text{cost}(x,u,j)=\min\begin{cases}\min\limits_{p+q=j}\{\text{cost}(y,u,p)+\text{cost}(z,u,q)\}+w(x)\cdot d(x,u)\\ \text{cost}(x,x,j-1)\end{cases}$$

式中，$d(x, u)$是从顶点x到顶点u的有向路径的长度。当x是叶结点时，$\text{cost}(x, u, j)$满足：

$$\text{cost}(x,u,j)=\begin{cases}w(x)\cdot d(x,u) & j=0\\ 0 & j>0\end{cases}$$

据此可设计动态规划算法如下。树型动态规划问题通常采用自底向上的方式计算，与树的后序遍历方式相同。可以将阶段性计算结果记录在树结点处。

二叉树的结点结构如下。

```
struct binode{
   int  w, d, dep;
   int  **cost, *dis;
   link  parent, left, right;
```

```
};
```

其中，w、d、dep 分别记录该结点的权值、边长和深度。在结点 x 处，cost(x, u, j)的值记录在 cost[j][u]中。dis[u]记录 $d(x, u)$的值。

```
typedef struct binode  *link;
link  *subt;
```

link 是指向二叉树的结点的指针。subt 是指向二叉树的结点的指针数组，用于构造与给定的树 T 等价的二叉树。读入初始数据并构造二叉树的算法如下。

```
void init() {
   int  a, b, c;
   cin >> n >> k;
   subt = new link[n+1];
   for(int i=0;i <= n; i++)
      subt[i] = newnode();
   for(i=1; i <= n; i++) {
      cin >> a >> b >> c;
      subt[i]->w = a;
      subt[i]->d = c;
      link  p = findc(subt[b]);
      link  q = newnode();
      p->left = subt[i];
      p->right = q;
      subt[i]->parent = p;
      q->parent = p;
   }
   sd(subt[0]);
}
```

其中，subt[0]是指向二叉树根结点的指针。算法 sd 以前序遍历的方式计算各结点的深度及 dist 的值。

```
void sd(link t) {
   PreOrder(adddep, t);
}
void PreOrder(void(*Visit)(link u), link t) {
   if(t) {
      Visit(t);
      PreOrder(Visit, t->left);
      PreOrder(Visit, t->right);
   }
}
void adddep(link t) {
   if(t->parent)
      t->dep = t->parent->dep+1;
   Make2DArray(t->cost, k+1, t->dep+1);
   t->dis = new int[t->dep+1];
   t->dis[0] = 0;
   if(t->parent)
```

```
    for(int i=1; i <= t->dep; i++)
      t->dis[i] = t->parent->dis[i-1]+t->d;
}
```

newnode()函数创建一个新的树结点。

```
link newnode() {
  link p=new binode;
  p->parent=0;p->left=0;p->right=0;
  p->w=0;p->d=0;p->dep=0;
  return p;
}
```

findc()函数找出新结点的插入位置。

```
link findc(link q) {
  while(q && q->right)
    q = q->right;
  return q;
}
```

在每个结点处的动态规划计算由 comp()函数完成。

```
void comp(link t) {
  if(!t->left) {
    for(int j=0; j <= t->dep; j++)
      t->cost[0][j] = t->w*t->dis[j];
    for(int i=1; i <= k; i++)
      for(int j=0; j <= t->dep; j++)
        t->cost[i][j] = 0;
  }
  else {
    for(int i=0; i <= k; i++) {
      for(int j=0; j <= t->dep; j++) {
        if(i)
          t->cost[i][j] = t->cost[i-1][0];
        else
          t->cost[i][j] = INT_MAX;
        for(int a=0; a <= i; a++) {
          int tmp = t->left->cost[a][j+1] + t->right->cost[i-a][j+1] + t->w*t->dis[j];
          if(t->cost[i][j] > tmp)
            t->cost[i][j] = tmp;
        }
      }
    }
  }
}
```

solution()函数用后序遍历的方式完成整棵树的动态规划计算。

```
int solution(){
  PostOrder(comp, subt[0]);
  return subt[0]->cost[k][0];
```

```
}
void PostOrder(void(*Visit)(link u), link t) {
   if(t) {
      PostOrder(Visit, t->left);
      PostOrder(Visit, t->right);
      Visit(t);
   }
}
```

最后整个算法由主函数完成。

```
int main() {
   init();
   cout << solution() << endl;
   return 0;
}
```

从动态规划递归式可知，算法的计算时间为 $O(k^2n^2)$。

3-19 有向树独立 k 中值问题。

问题描述：给定一棵有向树 T，树 T 中每个顶点 u 都有权值 $w(u)$，树的每条边(u, v)都有一个非负边长 $d(u, v)$。有向树 T 的每个顶点 u 可以看作客户，其服务需求量为 $w(u)$。每条边(u, v)的边长 $d(u, v)$可以看作运输费用。如果在顶点 u 处未设置服务机构，则将顶点 u 处的服务需求沿有向树的边(u, v)转移到顶点 v 处服务机构需付出的服务转移费用为 $w(u)\times d(u, v)$。树根处已设置了服务机构，现在要在树 T 中增设 k 处独立服务机构，使得整棵树 T 的服务转移费用最小。服务机构的独立性是指任何两个服务机构之间都不存在有向路经。

算法设计：对于给定的有向树 T，计算在树 T 中增设 k 处独立服务机构的最小服务转移费用。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。n 表示有向树 T 的边数；k 是要增设的服务机构数。有向树 T 的顶点编号为 $0, 1, \cdots, n$。根结点编号为 0。接下来的 n 行中，每行有表示有向树 T 的一条有向边的 3 个整数。第 i+1 行的 3 个整数 w_i、v_i、d_i 分别表示编号为 i 的顶点的权为 w_i,相应的有向边为(i, v_i),其边长为 d_i。

结果输出：将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 2	12
1 0 1	
1 1 10	
10 2 5	
1 2 3	

分析与解答：求有向树 T 的 k 个独立顶点组成的集合 F，使

$$\mathrm{cost}(F)=\sum_{x\in T}\min_{u\in F+r} w(x)\cdot d(x,u)$$

达到最小。与有向树 k 中值问题类似。首先将有向树 T 变换为与之等价的二叉树。设 T 的以顶点 x 为根的子树为 T_x，其左、右儿子顶点分别为 y 和 z。

在以顶点 x 为根的子树 T_x 中取 j 个 F 中顶点的最小耗费记为 $\mathrm{cost}(x, j)$，它具有最优子结

构性质，满足如下递归式：

$$\text{cost}(x, j) = \min \begin{cases} \min\limits_{p+q=j} \{\text{cost}(y, p) + \text{cost}(z, q)\} + w(x) \cdot d(x) \\ \sum\limits_{t \in T_x} w(t) \cdot d(t, x) \end{cases}$$

式中，$d(t, x)$是从顶点 t 到顶点 x 的有向路经的长度；$d(x)$是从顶点 x 到根结点的有向路经的长度。当 x 是叶结点时 $\text{cost}(x, j)$满足：

$$\text{cost}(x, j) = \begin{cases} w(x) \cdot d(x) & j = 0 \\ 0 & j > 0 \end{cases}$$

据此可设计动态规划算法如下。与树的后序遍历方式相同，采用自底向上的方式计算，将阶段性计算结果记录在树结点处。

二叉树的结点结构如下。

```
struct binode {
   int  w, wx, d, wd, dep;
   int  *cost;
   link  parent, left, right;
};
```

其中，wx、d、dep 分别记录该结点的权值、边长和深度。在结点 x 处 $\text{cost}(x, j)$的值记录在 cost[j]中。

```
typedef struct binode  *link;
link  *subt;
```

link 是指向二叉树的结点的指针。subt 是指向二叉树的结点的指针数组，用于构造与给定的树 T 等价的二叉树。读入初始数据并构造二叉树的算法如下。

```
void init() {
   int  a, b, c;
   cin >> n >> k;
   subt = new link[n+1];
   for(int i=0; i <= n; i++)
      subt[i] = newnode();
   for(i=1; i <= n; i++) {
      cin >> a >> b >> c;
      subt[i]->w = a;
      subt[i]->wx = a;
      subt[i]->d = c;
      link  p = findc(subt[b]);
      link  q = newnode();
      p->left = subt[i];
      p->right = q;
      subt[i]->parent = p;
      q->parent = p;
   }
   sd(subt[0]);
   sw(subt[0]);
```

```
}
```

其中，subt[0]是指向二叉树根结点的指针。算法 sd 用前序遍历的方式计算各结点的深度。

```
void sd(link t) {
   PreOrder(adddep,t);
}
void PreOrder(void(*Visit)(link u), link t) {
   if(t) {
      Visit(t);
      PreOrder(Visit, t->left);
      PreOrder(Visit, t->right);
   }
}
void adddep(link t) {
   if(t->parent)
      t->dep = t->parent->dep+1;
   t->cost = new int[k+1];
   if(t->parent)
      t->d += t->parent->d;
}
```

算法 sd 用后序遍历的方式计算各结点的 wd 的值。

```
void sw(link t) {
   PostOrder(addw, t);
}
void addw(link t) {
   t->wd = t->w*t->d;
   if(t->left) {
      t->w += t->left->w;
      t->w += t->right->w;
      t->wd += t->left->wd;
      t->wd += t->right->wd;
   }
}
```

newnode()函数创建一个新的树结点。findc()函数找出新结点的插入位置。在每个结点处动态规划计算由 comp()函数完成。

```
void comp(link t) {
   if(!t->left){
      t->cost[0] = t->wd;
      for(int i=1; i <= k; i++)
         t->cost[i] = 0;
   }
   else {
      for(int i=0; i <= k; i++) {
         t->cost[i] = t->wd-t->w*t->d;
         for(int a=0; a <= i; a++) {
            int  tmp = t->left->cost[a] + t->right->cost[i-a] + t->wx*t->d;
            if(t->cost[i] > tmp)
```

```
            t->cost[i] = tmp;
        }
      }
    }
}
```

solution()函数用后序遍历的方式完成整棵树的动态规划计算。

```
int solution() {
   PostOrder(comp, subt[0]);
   return subt[0]->cost[k];
}
void PostOrder(void(*Visit)(link u), link t) {
   if(t) {
      PostOrder(Visit, t->left);
      PostOrder(Visit, t->right);
      Visit(t);
   }
}
```

整个算法由主函数完成。

```
int main() {
   init();
   cout << solution() << endl;
   return 0;
}
```

从动态规划递归式可知，算法的计算时间为 $O(k^2n)$。

上述问题可以变换为与之等价的另一个树型动态规划问题。

设 $\text{gain}(F)=\sum_{x\in T} w(x)\cdot d(x)-\text{cost}(F)$，则 cost(F)的最小值对应于 gain(F)的最大值。设 T 的以顶点 x 为根的子树为 T_x，其左、右儿子顶点分别为 y 和 z。在以顶点 x 为根的子树 T_x 中取 j 个 F 中顶点的最大值记为 gain(x, j)，它具有最优子结构性质，满足如下递归式

$$\text{gain}(x,j)=\max\begin{cases}\max\limits_{p+q=j}\{\text{gain}(y,p)+\text{gain}(z,q)\}\\ \left(\sum\limits_{t\in T_x} w(t)\right)\cdot d(x)\end{cases}$$

式中，$d(x)$是从顶点 x 到根结点的有向路经的长度。当 x 是叶结点时 gain(x, j)满足

$$\text{gain}(x,j)=\begin{cases}0 & j=0\\ w(x)\cdot d(x) & j>0\end{cases}$$

据此可设计动态规划算法如下。与树的后序遍历方式相同，采用自底向上的方式计算，将阶段性计算结果记录在树结点处。

在每个结点处动态规划计算由 comp()函数完成。

```
void comp(link t) {
   if(!t->left) {
      t->gain[0] = 0;
      for(int i=1; i <= k; i++)
```

```
            t->gain[i] = t->w*t->d;
        }
        else {
            for(int i=0; i <= k; i++) {
                t->gain[i] = t->w * t->d;
                for(int a=0; a <= i; a++) {
                    int  tmp = t->left->gain[a] + t->right->gain[i-a];
                    if(t->gain[i] < tmp)
                        t->gain[i] = tmp;
                }
            }
        }
    }
```

solution()函数用后序遍历的方式完成整棵树的动态规划计算。

```
int solution() {
    PostOrder(comp, subt[0]);
    return tot-subt[0]->gain[k];
}
```

从动态规划递归式可知，算法的计算时间为 $O(k^2n)$。

3-20 有向直线 m 中值问题。

问题描述：给定一条有向直线 L 及 L 上的 n+1 个点 $x_0<x_1<\cdots<x_n$。有向直线 L 上的每个点 x_i 都有权值 $w(x_i)$，每条有向边(x_i, x_{i-1})都有一个非负边长 $d(x_i, x_{i-1})$。有向直线 L 上的每个点 x_i 可以看作客户，其服务需求量为 $w(x_i)$。每条边(x_i, x_{i-1})的边长 $d(x_i, x_{i-1})$可以看作运输费用。如果在点 x_i 处未设置服务机构，则将点 x_i 处的服务需求沿有向边转移到点 x_j 处服务机构需付出的服务转移费用为 $w(x_i)\times d(x_i, x_j)$。在点 x_0 处已设置了服务机构，现在要在直线 L 上增设 m 处服务机构，使得整体服务转移费用最小。

算法设计：对于给定的有向直线 L，计算在直线 L 上增设 m 处服务机构的最小服务转移费用。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示有向直线 L 上除了点 x_0，还有 n 个点 $x_0<x_1<\cdots<x_n$。接下来的 n 行中，每行有 2 个整数。第 i+1 行的 2 个整数分别表示 $w(x_{n-i-1})$和 $d(x_{n-i-1}, x_{n-i-2})$。

结果输出：将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9 2	26
1 2	
2 1	
3 3	
1 1	
3 2	
1 6	
2 1	
1 2	
1 1	

分析与解答：此题是有向树 k 中值问题在树 T 退化为有向直线的特殊情形，可以用算法实现题 3-21 的算法求解，或直接求解。

设 cost(i,j)是在前 i 个点中设置 j 个服务机构的最小费用，则 cost(i,j)具有最优子结构性质，满足如下递归式

$$\text{cost}(i,j)=\min_{1\le k\le i}\{\text{cost}(k-1,j-1)+\sum_{t=k+1}^{i} w(t)\cdot d(t,k)\}$$

式中，$d(t,k)$是从点 x_t 到点 x_k 的有向路径的长度。当 $j\ge i$ 时，显然有 cost(i,j)=0。

为了在 $O(1)$时间内计算 $\sum_{t=k+1}^{i} w(t)\cdot d(t,k)$，先用 $O(n)$时间预先计算

$$w(1,i)=\sum_{t=1}^{i} w(t)$$

$$d(1,i)=\sum_{t=1}^{i} d(x_t,x_{t-1})$$

$$\text{wd}(1,i)=\sum_{t=1}^{i} w(1,t)\cdot d(1,t)$$

readin()函数读入初始数据并进行预处理。

```
void readin() {
   int  d, w;
   dist[1] = 0;
   wt[0] = 0;
   swt[1] = 0;
   cin >> n >> m;
   cin >> wt[1] >> dist[2];
   for(int i=2; i <= n; i++) {
      cin >> w >> d;
      wt[i] = wt[i-1]+w;
      dist[i+1] = dist[i]+d;
      swt[i] = swt[i-1] + w*dist[i];
   }
}
```

$\sum_{t=i}^{j} w(t)\cdot d(t,j)$ 可在 $O(1)$时间内由 getw()函数计算如下。

```
int getw(int i, int j) {
   if(i > j)
      return 0;
    else
      return (wt[j-1] wt[i])*dist[j] - (swt[j-1]-swt[i]);
}
```

据此可设计动态规划算法如下。

```
void comp() {
   int  i, j, k, tmp;
   for(i=1; i <= n; i++)
```

```
      minco[i][0] = getw(0, i+1);
   for(i=1; i <= n; i++) {
      for(j=1; j <= m; j++) {
         if(j >= i)
            minco[i][j] = 0;
         else {
            minco[i][j] = getw(1, i+1);
            for(k=2; k <= i; k++) {
               tmp = minco[k-1][j-1] + getw(k, i+1);
               if(tmp < minco[i][j])
                  minco[i][j] = tmp;
            }
         }
      }
   }
}
```

minco[n][m]给出所求最优值。

```
int main() {
   readin();
   comp();
   cout << minco[n][m] << endl;
   return 0;
}
```

算法需要的计算时间为 $O(mn^2)$，占用空间 $O(mn)$。

注意，计算过程中只用到 minco 的一列数据，可以进一步将空间需求减少到 $O(n)$。

```
void comp() {
   int  i, j, k, tmp;
   for(i=1; i <= n; i++)
      minco[i] = getw(0, i+1);
   for(j=1; j <= m; j++) {
      for(i=n; i > j; i) {
         minco[i] = getw(1, i+1);
         for(k=2; k <= i; k++) {
            tmp = minco[k1]+getw(k, i+1);
            if(tmp < minco[i])
               minco[i] = tmp;
         }
      }
      for(i=1; i <= j; i++)
         minco[i] = 0;
   }
}
```

minco[n]给出所求最优值。

```
int main() {
   readin();
   comp();
```

```
    out << minco[n] << endl;
    return 0;
}
```

3-21　有向直线 2 中值问题。

问题描述：给定一条有向直线 L 及 L 上的 $n+1$ 个点 $x_0<x_1<\cdots<x_n$。有向直线 L 上的每个点 x_i 都有权值 $w(x_i)$，每条有向边(x_i, x_{i-1})都有一个非负边长 $d(x_i, x_{i-1})$。有向直线 L 上的每个点 x_i 可以看作客户，其服务需求量为 $w(x_i)$。每条边(x_i, x_{i-1})的边长 $d(x_i, x_{i-1})$可以看作运输费用。如果在点 x_i 处未设置服务机构，则将点 x_i 处的服务需求沿有向边转移到点 x_j 处服务机构需付出的服务转移费用为 $w(x_i)\times d(x_i, x_j)$。在点 x_0 处已设置了服务机构，现在要在直线 L 上增设 2 处服务机构，使得整体服务转移费用最小。

算法设计：对于给定的有向直线 L，计算在直线 L 上增设 2 处服务机构的最小服务转移费用。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示有向直线 L 上除了点 x_0，还有 n 个点 $x_0<x_1<\cdots<x_n$。接下来的 n 行中，每行有 2 个整数。第 $i+1$ 行的 2 个整数分别表示 $w(x_{n-i-1})$和 $d(x_{n-i-1}, x_{n-i-2})$。

结果输出：将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9	26
1 2	
2 1	
3 3	
1 1	
3 2	
1 6	
2 1	
1 2	
1 1	

分析与解答：

此题是有向直线 m 中值问题当 $m=2$ 的特殊情形，可以用算法实现题 3-20 的算法求解，可设计效率更高的算法。

设 cost(i, j)是在 x_i 和 x_j 处（$i<j$）设置服务机构的费用，则问题是求 $i<j$ 的最优位置，使 cost(i, j)达到最小。

假设在位置 i 已取定，j 的最优位置为 opt(i)，则 opt(i)具有如下性质：对任何 $i<j$，总有 opt(i)<opt(j)。

利用此性质可设计如下分治算法。

首先，计算 a=opt(n/2)。由上述性质知，当 $i<n/2$ 时，opt(i)<a；当 $i>n/2$ 时，opt(i)>a。

据此设计分治算法 comp(minp1, maxp1, minp2, maxp2)，找出 $x_i\in\{x_{minp1}, \cdots, x_{maxp1}\}$和 $x_j\in\{x_{minp2}, \cdots, x_{maxp2}\}$，使 cost($i$, j)最小。

readin()函数读入初始数据并进行预处理计算。

```
void readin() {
    int  d, w;
```

```
    cin >> n;
    dist[1] = 0;
    cin >> wt[1] >> dist[2];
    tot = wt[1]*dist[2];
    for (int i=2; i <= n; i++) {
        cin >> w >> d;
        wt[i] = wt[i-1]+w;
        dist[i+1] = dist[i]+d;
        tot = tot + wt[i]*d;
    }
}
```

getcost()函数计算 cost(i, j)的值。

```
int getcost(int i,int j) {
    if(i>j)
        return 0;
    else
        return tot - wt[i]*(dist[j]-dist[i]) - wt[j]*(dist[n+1]-dist[j]);
}
```

comp()函数递归计算最优值。

```
void comp(int minp1, int maxp1, int minp2, int maxp2) {
    int  i, j, cost, opt, optcost;
    if(minp1 >= minp2)
        minp2 = minp1+1;
    if(maxp1 >= maxp2)
        maxp1 = maxp2-1;
    if((minp1 > maxp1) || (minp2 > maxp2))
        return;
    // 计算 opt((minp2+maxp2)/2)
    optcost = MAXCOST+1;
    j = (minp2+maxp2)/2;
    for(i=minp1; i <= min(maxp1, j-1); i++) {
        cost=getcost(i,j);
        if(cost < optcost) {
            opt = i;
            optcost = cost;
        }
    }
    if(optcost < mincost)
        mincost = optcost;
    comp(minp1, opt, minp2, (minp2+maxp2)/2-1);
    comp(opt, maxp1, (minp2+maxp2)/2+1, maxp2);
}
```

comp(1, n−1, 2, n)完成整个计算。

```
int main() {
    readin();
    mincost = MAXCOST+1;
```

```
    comp(1, n-1, 2, n);
    cout << mincost << endl;
    return 0;
}
```

算法所需的计算时间为 $O(n\log n)$。

3-22 树的最大连通分支问题。

问题描述：给定一棵树 T，树中每个顶点 u 都有权值 $w(u)$，可以是负数。现在要找到树 T 的一个连通子图使该子图的权之和最大。

算法设计：对于给定的树 T，计算树 T 的最大连通分支。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示树 T 有 n 个顶点。树 T 的顶点编号为 $1, 2, \cdots, n$。第 2 行有 n 个整数，表示 n 个顶点的权值。接下来的 $n-1$ 行中，每行有表示树 T 的一条边的 2 个整数 u 和 v，表示顶点 u 与顶点 v 相连。

结果输出：将计算出的最大连通分支的权值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	4
−1 1 3 1 −1	
4 1	
1 3	
1 2	
4 5	

分析与解答：此题是树型动态规划算法。

设 sum(k)是以第 k 个顶点为根的子树中的最大连通分支的权值，则 sum(k)满足如下递归式

$$\text{sum}(k) = w(k) + \sum_{\text{sum}(i)>0} \text{sum}(i)$$

式中，顶点 i 是顶点 k 的儿子结点。所求的最大连通分支的权值为 $\max\limits_{1\le k\le n}\{\text{sum}(k)\}$。

据此可设计如下树型动态规划算法。

树结点结构如下：

```
typedef struct node *link;
struct node {
   int  s;
   struct node  *next;
};
```

init()函数读入初始数据。

```
void init() {
   int  a, b;
   cin>>n;
   for(int i=1; i<=  n; i++)
      cin >> w[i];
   for(i=1; i <= n-1; i++) {
      cin >> a >> b;
      link  p = new node;
      p->s = a;
```

```
        p->next = child[b];
        child[b] = p;
        p = new node;
        p->s = b;
        p->next=  child[a];
        child[a] = p;
    }
}
```

build()函数建立树结构。

```
void build(int k) {
    link p,q;
    vis[k]=1;
    p=child[k];
    while(p){
        int  i = p->s;
        if(vis[i]) {
            if(p == child[k]) {
                child[k] = p->next;
                delete p;
                p = child[k];
            }
            else {
                q->next = p->next;
                delete p;
                p = q->next;
            }
        }
        else {
            build(i);
            if(p == child[k])
                q = child[k];
            else
                q = q->next;
            p = p->next;
        }
    }
}
```

comp()函数实现动态规划算法。

```
void comp(int k) {
    link  p = child[k];
    while(p){
        int  i = p->s;
        comp(i);
        if(sum[i] > 0)
            sum[k] += sum[i];
        p = p->next;
    }
```

```
    sum[k] += w[k];
}
```

主函数如下。

```
int main() {
    init();
    build(1);
    comp(1);
    out();
    return 0;
}
```

out()函数输出最优值。

```
void out(){
    for(int i=1; i <= n; i++)
        if(sum[i] > maxw)
            maxw = sum[i];
    cout << maxw << endl;
}
```

算法需要的计算时间显然为 $O(n)$。

3-23 直线 k 中值问题。

问题描述：在一个按照南北方向划分成规整街区的城市里，n 个居民点分布在一条直线上的 n 个坐标点 $x_0<x_1<\cdots<x_n$ 处。居民们希望在城市中至少选择一个，但不超过 k 个居民点建立服务机构。在每个居民点 x_i 处，服务需求量为 $w_i\geq 0$，在该居民点设置服务机构的费用为 $c_i\geq 0$。假设居民点 x_i 到距其最近的服务机构的距离为 d_i，则居民点 x_i 的服务费用为 $w_i\times d_i$。

建立 k 个服务机构的总费用为 $A+B$。A 是在 k 个居民点设置服务机构的费用的总和；B 是 n 个居民点服务费用的总和。

算法设计：对于给定直线 L 上的 n 个点 $x_0<x_1<\cdots<x_n$，计算在直线 L 上最多设置 k 处服务机构的最小总费用。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。n 表示直线 L 上有 n 个点 $x_0<x_1<\cdots<x_n$；k 是服务机构总数的上限。接下来的 n 行中，每行有 3 个整数。第 $i+1$ 行的 3 个整数 x_i、w_i、c_i，分别表示相应居民点的位置坐标、服务需求量和在该点设置服务机构的费用。

结果输出：将计算的最小服务费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9 3	19
2 1 2	
3 2 1	
6 3 3	
7 1 1	
9 3 2	
15 1 6	
16 2 1	
18 1 2	

19.11

分析与解答：直线 L 上的每个点 x_i 是一个客户，其服务需求量为 $w(i)$。每个点 x_i 到服务机构 S 的距离定义为 $d(i,s)=\min_{y\in S}\{|x_i-y|\}$。服务机构 S 的总服务费用为

$$\text{cost}(S)=\sum_{x_i\in S}c(i)+\sum_{j=1}^{n}w(j)\cdot d(j,S)$$

设 opt(i,m)是在 x_1, x_2, …, x_m 中恰好设置 i 个服务机构的最小费用；popt(i,m)是在 x_1, x_2, …, x_m 中恰好设置 i 个服务机构且在 x_m 处设置了服务机构的最小费用。

$$\text{opt}(i,m)=\min_{S\subseteq V_m,|S|=i}\left(\sum_{x_i\in S}c(i)+\sum_{j=1}^{m}w(i)\cdot d(j,S)\right)$$

$$\text{popt}(i,m)=\min_{S\subseteq V_m,|s|=i,x_m\in S}\left(\sum_{x_i\in S}c(i)+\sum_{j=1}^{m}w(j)\cdot d(j,S)\right)$$

由此可知，所求的最优值为 $\text{opt}=\min_{1\le j\le k}(\text{opt}(i,n))$

opt(i,m)和 popt(i,m)具有最优子结构性质，且满足如下递归式：

$$\text{opt}(i,m)=\min_{i\le j\le m}\left(\text{popt}(i,j)+\sum_{t=j+1}^{m}w(t)\cdot d(j,t)\right)$$

$$\text{popt}(i,m)=\min_{i-1\le j\le m-1}\left(\text{opt}(i-1,j)+\sum_{t=j+1}^{m-1}w(t)\cdot d(t,m)\right)+c(m)$$

式中，$d(j,t)=x_t-x_j$ 是点 x_t 和 x_j 之间的距离。

边界条件是，当 i=1 时，$S=\{x_m\}$，

$$\text{popt}(1,m)=c(m)+\sum_{t=1}^{m-1}w(t)\cdot d(t,m)$$

在算法预处理时，用 $O(n)$时间计算出 $\text{sw}(m)=\sum_{i=1}^{m}w(i)$ 和 $\text{wd}(m)=\sum_{i=1}^{m}w(i)\cdot d(1,i)$，则可在 $O(1)$时间内计算 $\sum_{t=j+1}^{m}w(t)\cdot d(j,t)$ 和 $\sum_{t=j+1}^{m-1}w(t)\cdot d(t,m)$ 如下。

$$\sum_{t=j+1}^{m}w(t)\cdot d(j,t)=\text{wd}(m)-\text{wd}(j)-[\text{sw}(m)-\text{sw}(j)]\cdot d(1,j)$$

$$\sum_{t=j+1}^{m-1}w(t)\cdot d(t,m)=[\text{sw}(m-1)-\text{sw}(j)]\cdot d(1,m)-[\text{wd}(m-1)-\text{wd}(j)]$$

据此可设计动态规划算法如下。

readin()函数读入初始数据并进行预处理。

```
void readin() {
   int  w;
   wt[0] = 0;
   swt[0] = 0;
   cin >> n >> m;
   for(int i=1; i <= n; i++) {
```

```
    cin >> x[i] >> w >> c[i];
    ww[i] = w;
    wt[i] = wt[i-1] + w;
    dist[i] = x[i] - x[1];
    swt[i] = swt[i-1] + w*dist[i];
  }
}
```

comp()函数实现动态规划算法。

```
void comp() {
  int  i, j, k, tmp;
  // opt2[1][j]
  for(j=1; j <= n; j++)
    opt2[1][j]=c[j]+getw2(0,j);
  // opt1[1][j]
  for(j=1; j <= n; j++) {
    opt1[1][j] = opt2[1][1] + getw1(1,j);
    for(k=2; k <= j; k++){
      tmp = opt2[1][k] + getw1(k,j);
      if(opt1[1][j] > tmp)
        opt1[1][j] = tmp;
    }
  }
  for(i=2; i <= m; i++) {
    // opt2[i][j]
    for(j=i; j <= n; j++) {
      opt2[i][j] = opt1[i-1][i-1] + getw2(i-1, j);
      for(k=i; k < j; k++) {
        tmp = opt1[i-1][k] + getw2(k, j);
        if(opt2[i][j] > tmp)
          opt2[i][j] = tmp;
      }
      opt2[i][j] += c[j];
    }
    // opt1[i][j]
    for(j=i; j <= n; j++) {
      opt1[i][j] = opt2[i][i] + getw1(i, j);
      for(k=i+1; k <= j; k++) {
        tmp = opt2[i][k] + getw1(k, j);
        if(opt1[i][j] > tmp)
          opt1[i][j]=tmp;
      }
    }
  }
}
```

其中，getw1()函数和getw2()函数分别用$O(1)$时间计算$\sum_{t=j+1}^{m} w(t)\cdot d(j,t)$和$\sum_{t=j+1}^{m-1} w(t)\cdot d(t,m)$的值。

```
int getw1(int j,int m) {
   return (swt[m]-swt[j])-(wt[m]-wt[j])*dist[j];
}
int getw2(int j, int m) {
   return (wt[m-1]-wt[j])*dist[m]-(swt[m-1]-swt[j]);
}
```

算法的主函数如下。

```
int main() {
   readin();
   comp();
   cout<<solution() << endl;
   return 0;
}
```

solution()函数输出最优值。

```
int solution() {
   int  tmp = opt1[1][n];
   for(int i=2; i <= m; i++)
      if(opt1[i][n] < tmp)
         tmp = opt1[i][n];
   return tmp;
}
```

从动态规划递归式可知，算法的计算时间为 $O(kn^2)$，使用空间 $O(kn)$。

注意到动态规划计算只用到矩阵 opt1 和 opt2 的一行数据，可进一步将空间需求减少到 $O(n)$。

```
void comp() {
   int  i, j, k, tmp;
   // opt2[j]
   for(j=1; j <= n;j++)
      opt2[j] = c[j] + getw2(0,j);
   // opt1[j]
   for(j=1; j <= n; j++) {
      opt1[j] = opt2[1] + getw1(1, j);
      for(k=2; k <= j; k++) {
         tmp = opt2[k] + getw1(k, j);
         if(opt1[j] > tmp)
            opt1[j] = tmp;
      }
   }
   min = opt1[n];
   for(i=2; i <= m; i++) {
      // opt2[j]
      for(j=i; j <= n; j++) {
         opt2[j] = opt1[i1] + getw2(i1, j);
         for(k=i; k < j; k++) {
            tmp = opt1[k] + getw2(k, j);
            if(opt2[j] > tmp)
```

```
            opt2[j] = tmp;
        }
        opt2[j] += c[j];
    }
    // opt1[j]
    for(j=i; j <= n; j++) {
        opt1[j] = opt2[i] + getw1(i, j);
        for(k=i+1; k <= j; k++) {
            tmp = opt2[k] + getw1(k, j);
            if(opt1[j] > tmp)
                opt1[j] = tmp;
        }
    }
    if(opt1[n] < min)
        min = opt1[n];
  }
}
```

最优值在全局变量 min 中。

3-24 直线 k 覆盖问题。

问题描述：给定一条直线 L 上的 n 个点 $x_0<x_1<\cdots<x_n$，每个点 x_i 都有权值 $w(i)\geq 0$，以及在该点设置服务机构的费用 $c(i)\geq 0$。每个服务机构的覆盖半径为 r。直线 k 覆盖问题是要求找出 $V_n=\{x_1, x_2, \cdots, x_n\}$的一个子集 $S\subseteq V_n$，$|S|\leq k$，在点集 S 处设置服务机构，使总覆盖费用达到最小。

每个点 x_i 都是一个客户。每个点 x_i 到服务机构 S 的距离定义为 $d(i, S)=\min\limits_{y\in S}\{|x_i-y|\}$。如果客户 x_i 在 S 的服务覆盖范围内，即 $d(i, S)\leq r$，则其服务费用为 0，否则其服务费用为 $w(i)$。服务机构 S 的总覆盖费用为

$$\mathrm{cost}(S)=\sum_{x_i\in S}c(i)+\sum_{j=1}^{n}w(j)\,I(j,S)$$

式中，$I(j, S)$的定义为

$$I(j,S)=\begin{cases}0 & d(j,S)\leq r\\ 1 & d(j,S)>r\end{cases}$$

算法设计：对于给定直线 L 上的 n 个点 $x_0<x_1<\cdots<x_n$，计算在直线 L 上最多设置 k 处服务机构的最小覆盖费用。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、k 和 r。n 表示直线 L 上有 n 个点 $x_0<x_1<\cdots<x_n$；k 是服务机构总数的上限；r 是服务机构的覆盖半径。接下来的 n 行中，每行有 3 个整数。第 i+1 行的 3 个整数 x_i、w_i、c_i 分别表示 $x(i)$、$w(i)$和 $c(i)$。

结果输出：将计算的最小覆盖费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9 3 2	12
2 1 12	
3 2 11	
6 3 3	

7 1 11
9 3 12
15 1 6
16 2 11
18 1 2
19 1 11

分析与解答： 设 opt(i, m)是在 x_1, x_2, …, x_m 中设置 i 个服务机构的最小费用；popt(i, m)是在 x_1, x_2, …, x_m 中设置 i 个服务机构且在 x_m 处设置了服务机构的最小费用。

$$\mathrm{opt}(i,m)=\min_{S\subseteq V_m,|s|=i}\left(\sum_{x_j\in S}c(i)+\sum_{j=1}^{m}w(j)I(j,S)\right)$$

$$\mathrm{popt}(i,m)=\min_{S\subseteq V_m,|s|=i,x_m\in S}\left(\sum_{x_j\in S}c(i)+\sum_{j=1}^{m}w(j)I(j,S)\right)$$

由此可知，所求的最优值为

$$\mathrm{opt}=\min_{1\le i\le k}\left[\mathrm{opt}(i,n)\right]$$

opt(i, m)和 popt(i, m)具有最优子结构性质，且满足如下递归式

$$\mathrm{opt}(i,m)=\min\{w(m)+\mathrm{opt}(i,m-1),\min_{\mathrm{cov}(m)\le j\le m}\mathrm{popt}(i,j)\}$$

$$\mathrm{popt}(i,m)=c(m)+\min_{\mathrm{unc}(m)\le j\le m-1}\mathrm{opt}(i-1,j)$$

式中，cov(j)=min{i : $i\le j$, $x_j-x_i\le r$}是 x_1, x_2, …, x_j 中被 x_j 覆盖的最小下标，unc(j)=max{i : $i<j$, $x_j-x_i>r$}是 x_1, x_2, …, x_j 中未被 x_j 覆盖的最大下标。

边界条件是，当 i=1 时，$S=\{x_m\}$，$\mathrm{popt}(1,m)=c(m)+\sum_{i=1}^{\mathrm{unc}(m)}w(t)$。

据此可设计动态规划算法如下。

readin()函数读入初始数据。

```
void readin() {
   cin >> n >> m >> r;
   for(int i=1; i <= n; i++)
      cin >> x[i] >> w[i] >> c[i];
}
```

comp()函数实现动态规划算法。

```
void comp() {
   int  i, j, k, h, tmp;
   // opt2[1][j]
   for(j=1; j <= n; j++) {
      h=unc(j);
      opt2[1][j]=c[j];
      for(k=1; k <= h; k++)
         opt2[1][j] += w[k];
   }
   // opt1[1][j]
   for(j=1; j <= n; j++) {
      if(j > 1)
```

```
      opt1[1][j] = w[j] + opt1[1][j-1];
    else
      opt1[1][j] = INT_MAX;
    h = cov(j);
    tmp = INT_MAX;
    for(k=h; k <= j; k++) {
      if(opt2[1][k] < tmp)
        tmp = opt2[1][k];
      if(opt1[1][j] > tmp)
        opt1[1][j] = tmp;
    }
  }
  for(i=2; i <= m; i++) {
    // opt2[i][j]
    for(j=i; j <= n; j++) {
      h = unc(j);
      if(h<i-1)
        h=i-1;
      opt2[i][j] = INT_MAX;
      for(k=h; k < j; k++) {
        tmp = opt1[i-1][k];
        if(opt2[i][j] > tmp)
          opt2[i][j] = tmp;
      }
      opt2[i][j] += c[j];
    }
  }
  // opt1[i][j]
  for(j=i; j <= n; j++) {
    h= cov(j);
    if(h < i)
      h = i;
    tmp = INT_MAX;
    if(j > i)
      opt1[i][j] = w[j] + opt1[i][j-1];
    else
      opt1[i][j] = INT_MAX;
    for(k=h; k <= j; k++) {
      if(opt2[i][k] < tmp)
        tmp = opt2[i][k];
      if(opt1[i][j] > tmp)
        opt1[i][j] = tmp;
    }
  }
}
```

cov()函数和 unc()函数分别计算 cov(*j*)和 unc(*j*)的值。

```
int cov(int j) {
  for(int i=j; i > 0; i--)
```

```
    if(x[j]-x[i] > r)
      break;
  return i+1;
}
int unc(int j) {
  for(int i=1; i <= j; i++)
    if(x[j]-x[i] <= r)
      break;
  return i-1;
}
```

算法的主函数如下。

```
int main() {
  readin();
  comp();
  cout << solution() << endl;
  return 0;
}
```

solution()函数输出最优值。

```
int solution() {
  int  tmp = opt1[1][n];
  for(int i=2; i <= m; i++)
    if(opt1[i][n] < tmp)
      tmp = opt1[i][n];
  return tmp;
}
```

从动态规划递归式可知，算法的计算时间为 $O(kn^2)$，使用空间 $O(kn)$。

注意，动态规划计算只用到矩阵 opt1 和 opt2 的一行数据，可进一步将空间需求减小到 $O(n)$。

```
void comp() {
  int  i, j, k, h, tmp;
  // opt2[j]
  for(j=1; j <= n; j++){
    h = unc(j);
    opt2[j] = c[j];
    for(k=1; k <= h; k++)
      opt2[j] += w[k];
  }
  // opt1[j]
  for(j=1; j <= n; j++) {
    if(j > 1)
      opt1[j] = w[j] + opt1[j-1];
    else
      opt1[j] = INT_MAX;
    h = cov(j);
    tmp = INT_MAX;
    for(k=h; k <= j; k++) {
```

```
        if(opt2[k] < tmp)
          tmp = opt2[k];
        if(opt1[j] > tmp)
          opt1[j] = tmp;
      }
    }
    in = opt1[n];
    for(i=2; i <= m; i++) {
      // opt2[j]
      for(j=i; j <= n; j++) {
        h = unc(j);
        if(h < i-1)
          h=i-1;
        opt2[j] = INT_MAX;
        for(k=h; k < j; k++) {
          tmp = opt1[k];
          if(opt2[j] > tmp)
            opt2[j] = tmp;
        }
        opt2[j]+=c[j];
      }
      // opt1[j]
      for(j=i; j <= n; j++) {
        h = cov(j);
        if(h < i)
          h = i;
        tmp = INT_MAX;
        if(j > i)
          opt1[j] = w[j] + opt1[j-1];
        else
          opt1[j] = INT_MAX;
        for(k=h; k <= j; k++)
          if(opt2[k] < tmp)
            tmp = opt2[k];
        if(opt1[j] > tmp)
          opt1[j] = tmp;
      }
      if(opt1[n] < min)
        min = opt1[n];
    }
}
```

最优值在全局变量 min 中。

3-25 m 处理器问题。

问题描述：在网络通信系统中，要将 n 个数据包依次分配给 m 个处理器进行数据处理，并要求处理器负载尽可能均衡。设给定的数据包序列为 $\{\sigma_0, \sigma_1, \cdots, \sigma_{n-1}\}$。$m$ 处理器问题要求的是 $r_0=0 \leqslant r_1 \leqslant \cdots \leqslant r_{m-1} \leqslant n=r_m$，将数据包序列划分为 m 段：$\{\sigma_0, \cdots, \sigma_{r_1-1}\}$，$\{\sigma_{r_1}, \cdots, \sigma_{r_2-1}\}$，$\cdots$，

$\{\sigma_{r_{m-1}},\cdots,\sigma_{n-1}\}$，使 $\max_{i=0}^{m-1}\{f(r_i,r_{i+1})\}$ 达到最小。式中，$f(i,j)=\sqrt{\sigma_i^2+\cdots+\sigma_j^2}$ 是序列$\{\sigma_i, \cdots, \sigma_j\}$的负载量。

$\max_{i=0}^{m-1}\{f(r_i,r_{i+1})\}$ 的最小值称为数据包序列$\{\sigma_0, \sigma_1, \cdots, \sigma_{n-1}\}$的均衡负载量。

算法设计：对于给定的数据包序列$\{\sigma_0, \sigma_1, \cdots, \sigma_{n-1}\}$，计算 m 个处理器的均衡负载量。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。n 表示数据包个数，m 表示处理器数。接下来的 1 行中有 n 个整数，表示 n 个数据包的大小。

结果输出：将计算的处理器均衡负载量输出到文件 output.txt，且保留 2 位小数。

输入文件示例	输出文件示例
input.txt	output.txt
6 3	12.32
2 2 12 3 6 11	

分析与解答：设 $g(i, k)$是将$\{\sigma_i, \cdots, \sigma_{n-1}\}$划分为 k 段的均衡负载量，所求的最优值为 $g(0, m)$。

$g(i, k)$具有最优子结构性质，且满足如下递归式：

当 $2\leqslant k<n-i$ 时，$g(i,k)=\min_{i\leqslant j\leqslant n-k}\max\{f(i,j),g(j+1,k-1)\}$。

当 $n-i<k\leqslant m$ 时，$g(i, k)=g(i, n-i)$。

边界条件是：$g(i, 1)=f(i, n-1)$，$g(i, n-i)=\max_{i\leqslant j\leqslant n} f(j, j)$。

据此可设计动态规划算法如下。

```
void solve(int n,int m) {
   int  i, j, k, tmp, maxt;
   tmp=f(n-1, n-1);
   for(i=n-1; i >= 0; i--) {
      if(f(i, i) > tmp)
         tmp = f(i, i);
      g[i][1] = f(i, n-1);
      if(n-i <= m)
         g[i][n-i] = tmp;
   }
   for(i=n-1; i >= 0; i--) {
      for(k=2; k <= m; k++) {
         for(j=i, tmp=INT_MAX; j <= n-k; j++) {
            maxt = max(f(i, j), g[j+1][k-1]);
            if(tmp > maxt)
               tmp = maxt;
         }
         g[i][k] = tmp;
      }
      for(k=n-i+1; k <= m; k++)
         g[i][k] = g[i][n-i];
   }
}
```

算法所需的计算时间为 $O(mn^2)$。

上述动态规划算法适用于一般的函数 f。本题的函数 f 具有非负性和单调性，即对于 $0 \leqslant i \leqslant j \leqslant n-1$，有 $f(i,j) \geqslant 0$，$f(i+1,j) < f(i,j) < f(i,j+1)$。

利用函数的单调性可进一步将算法的计算时间降低到 $O(m(n-m))$如下。

```
void solve(int n, int m) {
   int  i, j, k;
   for(i=n-1; i >= 0; i--)
     g[i][1] = f(i, n-1);
   for(k=2; k <= m; k++) {
     g[n-k][k] = max(f(n-k, n-k), g[n-k+1][k-1]);
     j = n-k;
     for(i=n-k-1; i >= m-k; i--) {
       if(f(i,j) <= g[j+1][k-1])
         g[i][k] = g[j+1][k-1];
       else if(f(i,i) >= g[i+1][k-1]) {
         g[i][k] = f(i, i);
         j=i;
       }
       else {
         while(f(i, j-1) >= g[j][k-1])
           j--;
         g[i][k] = min(f(i, j), g[j][k-1]);
         if(g[i][k] == g[j][k-1])
           j--;
       }
     }
   }
}
```

计算机算法设计与分析习题解答（第5版）

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

提升学生“知识—能力—素质”	体现“基础—技术—应用”内容
把握教学“难度—深度—强度”	提供“教材—教辅—课件”支持

相关图书：《计算机算法设计与分析（第5版）》 ISBN 978-7-121-34439-8

策划编辑：章海涛
责任编辑：章海涛
封面设计：张 昱

ISBN 978-7-121-34438-1

定价：56.00 元