第5章

贪心算法

在求最优解问题的过程中，依据某种贪心标准，从问题的初始状态出发，直接去求每一步的最优解，通过若干次的贪心选择，最终得出整个问题的最优解，这种求解方法就是贪心算法。从贪心算法的定义可以看出，贪心算法并不是从整体上考虑问题，它所做出的选择只是在某种意义上的局部最优解，而由问题自身的特性决定了该题运用贪心算法可以得到最优解。贪心算法所做的选择可以依赖于以往所做过的选择，但既不依赖于将来的选择，也不依赖于子问题的解，因此贪心算法与其他算法相比具有一定的速度优势。如果一个问题可以同时用几种方法解决，贪心算法应该是最好的选择之一。

当一个问题具有最优子结构性质和贪心选择性质时，贪心算法通过一系列的选择来得到一个问题的解。它所做的每一个选择都是在当前状态下具有某种意义的最好选择，即贪心选择，并且每次贪心选择都能将问题化简为一个更小的与原问题具有相同形式的子问题。尽管贪心算法对许多问题不能总是产生整体最优解，但对诸如最短路径问题、最小生成树问题，以及哈夫曼编码问题等具有最优子结构和贪心选择性质的问题却可以获得整体最优解，而且所给出的算法一般比动态规划算法更加简单、直观和高效。

5.1 活动安排问题

活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

设有 n 个活动的集合 $E=\{1,2,\cdots,n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i，且 $s_i<f_i$。如果选择了活动 i，则它在半开时间区间 $[s_i,f_i)$ 内占用资源。若区间 $[s_i,f_i)$ 与区间 $[s_j,f_j)$ 不相交，则称活动 i 与活动 j 是相容的。也就是说，当 $s_i\geqslant f_j$ 或 $s_j\geqslant f_i$ 时，活动 i 与活动 j 相容。活动安排问题就是要在所给

的活动集合中选出最大的相容活动子集合。

为了编程方便,我们把活动的起始时间和结束时间定义为结构体:

```
struct action{
    int s;                                          //起始时间
    int f;                                          //结束时间
    int index;                                      //活动的编号
};
```

活动的集合 E 记为数组:

```
action a[1000];
```

然后按活动的结束时间升序排序:$a[1].f \leqslant a[2].f \leqslant \cdots \leqslant a[n].f$。排序比较因子:

```
bool cmp(const action &a, const action &b)
{
    if (a.f <= b.f) return true;
    return false;
}
```

使用标准模板库函数排序(下标 0 未用):

```
sort(a + 1, a + n + 1, cmp);
```

计算活动安排问题的贪心算法,如算法 5.1 所示。

算法 5.1 计算活动安排问题的贪心算法。

```
//形参数组 b 用来记录被选中的活动
void GreedySelector(int n, action a[], bool b[])
{
    b[1] = true;                                    //第 1 个活动是必选的
    //记录最近一次加入集合 b 中的活动
    int preEnd = 1;
    for(int i = 2; i <= n; i++)
        if (a[i].s >= a[preEnd].f)
        {
            b[i] = true;
            preEnd = i;
        }
}
```

算法实现源代码:actionArrangement.cpp。

活动 i 在集合 b 中,当且仅当 $b[i]$ 的值为 true。变量 preEnd 用来记录最近一次加入集合 b 中的活动。算法 greedySelector 首先选择活动 1,并将 preEnd 初始化为 1,然后依次检查活动 i 是否与当前已经选择的所有活动相容。若相容则将活动 i 加入已选择活动的集合 b 中,否则放弃,继续检查下一活动与集合 b 中活动的相容性。

例如有 11 个活动,其开始时间和结束时间如表 5-1 所示,并按结束时间升序排列。

表 5-1　样例数据(11 个活动)

i	1	2	3	4	5	6	7	8	9	10	11
$a[i].s$	1	3	0	5	3	5	6	8	8	2	12
$a[i].f$	4	5	6	7	8	9	10	11	12	13	14
b	1	0	0	1	0	0	0	1	0	0	1

由于输入的活动按完成时间升序排列,所以算法 greedySelector 每次总是选择具有最早完成时间的相容活动加入集合 b 中。直观上,按这种方法选择相容活动为未安排活动留下尽可能多的时间。该算法的贪心选择意义是使剩余的可安排时间段极大化,以便安排尽可能多的相容活动。

算法 5.1 的计算过程,如图 5-1 所示。

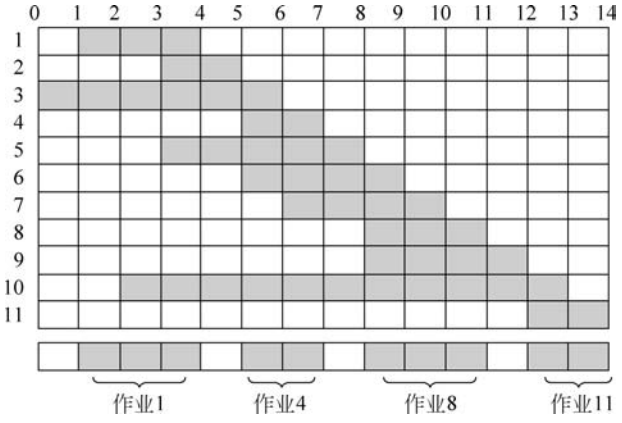

图 5-1　活动安排问题的几何意义

根据数组 b 的值,就可以输出选中的活动编号:

```
for(int i = 1; i < = n;i++)
    if(b[i]) printf(" % d ",a[i].index);
printf("\n");
```

当输入的活动已按结束时间升序排列,算法只需 $O(n)$的时间来安排 n 个活动,使最多的活动能相容地使用公共资源。如果所给出的活动未按升序排列,可用 $O(n\log_2 n)$时间重排。

贪心算法并不总能求得问题的整体最优解。但对于活动安排问题,贪心算法 greedySelector 却总能求得整体最优解,即它最终所确定的相容活动集合 b 的规模最大。这个结论可以用数学归纳法证明,请参考文献[3]。

5.2　贪心算法的理论基础

贪心算法是一种在每一步选择中都采取在当前状态下最好或最优的选择,从而希望得到结果是最好或最优的算法。贪心算法是一种能够得到某种度量意义下的最优解的分级处理方法,通过一系列的选择来得到一个问题的解,而它所做的每一次选择都是当前状态下某

种意义的最好选择,即贪心选择。即希望通过问题的局部最优解来求出整个问题的最优解。这种算法是一种很简洁的方法,对许多问题它能产生整体最优解,但不能保证总是有效,因为它不是对所有问题都能得到整体最优解,只能说其解必然是最优解的很好近似值。

利用贪心算法解题,需要解决两个问题:

(1) 该题是否适合用贪心算法求解;

(2) 如何选择贪心标准,以得到问题的最优/较优解。

5.2.1 贪心选择性质

贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择,即贪心选择来达到。这是贪心算法可行的第一个基本要素,也是贪心算法与动态规划算法的主要区别。

(1) 在动态规划算法中,每步所做的选择往往依赖于相关子问题的解,因而只有在解出相关子问题后才能做出选择。

(2) 在贪心算法中,仅在当前状态下做出最好选择,即局部最优选择。然后再解出这个选择后产生的相应的子问题。贪心算法所做的贪心选择可以依赖于以往所做的选择,但不依赖于将来所做的选择,也不依赖于子问题的解。

正是由于这种差别,动态规划算法通常以自底向上的方式解各个子问题,而贪心算法则通常以自顶向下的方式进行,以迭代的方式做出相继的贪心选择,每做一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题,要确定它是否具有贪心选择性质,必须证明每一步所做的贪心选择最终导致问题的整体最优解。首先考察问题的一个整体最优解,并证明可修改这个最优解,使其以贪心选择开始。做了贪心选择后,原问题简化为规模更小的类似子问题。然后,用数学归纳法证明,通过每一步做贪心选择,最终可得到问题的整体最优解。证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

5.2.2 最优子结构性质

当一个问题的最优解包含其子问题的最优解时,称此问题具有最优子结构性质。运用贪心算法在每一次转化时都取得了最优解。问题的最优子结构性质是该问题可用贪心算法或动态规划算法求解的关键特征。贪心算法的每一次操作都对结果产生直接影响,而动态规划算法则不是。贪心算法对每个子问题的解决方案都做出选择,不能回退;动态规划算法则会根据以前的选择结果对当前进行选择,有回退功能。动态规划算法主要运用于二维或三维问题,而贪心算法一般是一维问题。

5.2.3 贪心算法的求解过程

使用贪心算法求解问题应该考虑如下几个方面:

(1) 候选集合 A:为了构造问题的解决方案,有一个候选集合 A 作为问题的可能解,即

问题的最终解均取自于候选集合 A。

(2) 解集合 S：随着贪心选择的进行，解集合 S 不断扩展，直到构成满足问题的完整解。

(3) 解决函数 solution：检查解集合 S 是否构成问题的完整解。

(4) 选择函数 select：即贪心策略，这是贪心算法的关键，它指出哪个候选对象最有希望构成问题的解，选择函数通常和目标函数有关。

(5) 可行函数 feasible：检查解集合中加入一个候选对象是否可行，即解集合扩展后是否满足约束条件。

贪心算法的一般流程，如算法 5.2 所示。

算法 5.2　贪心算法的一般流程。

```
//A 是问题的输入集合即候选集合
Greedy(A)
{
    S = { };                                    //初始解集合为空集
    while (not solution(S))                     //集合 S 没有构成问题的一个解
    {
        x = select(A);                          //在候选集合 A 中做贪心选择
        if feasible(S, x)                       //判断集合 S 中加入 x 后的解是否可行
            S = S + {x};
            A = A - {x};
    }
    return S;
}
```

5.3　背包问题

给定一个载重量为 M 的背包，考虑 n 个物品，其中第 i 个物品的质量为 w_i，价值为 $v_i(1\leqslant i\leqslant n)$，要求把物品装满背包，且使背包内的物品价值最大。有两类背包问题(根据物品是否可以分割)，如果物品不可以分割，称为 0-1 背包问题(见 4.5 节)；如果物品可以分割，则称为背包问题，就是本节讨论的内容。

假设 x_i 是物品 i 装入背包的部分($0\leqslant x_i\leqslant 1$)，当 $x_i=0$ 时表示物品 i 没有被装入背包；当 $x_i=1$ 时表示物品 i 被全部装入背包。根据问题的要求，该问题可形式化描述为：

$$\max\sum_{i=1}^{n}p_ix_i,\quad \text{s.t.}\ \sum_{i=1}^{n}w_ix_i=M$$

其中，$0\leqslant x_i\leqslant 1$。

假设背包的容量是 50，有 3 个物品，如表 5-2 所示。

表 5-2　背包问题的 3 个物品

n	1	2	3
质量	10	20	30
价值	60	100	120
性价比	6	5	4

有 3 种方法来选取物品：

(1) 当作 0-1 背包问题，采用动态规划算法，如图 5-2(b)所示，获得最优值 220；

(2) 当作 0-1 背包问题，采用贪心算法，按性价比从高到低的顺序选取物品，获得最优值 160，如图 5-2(c)所示。由于物品不可分割，剩下的空间 20，没有相应的物品可以装入而白白浪费。

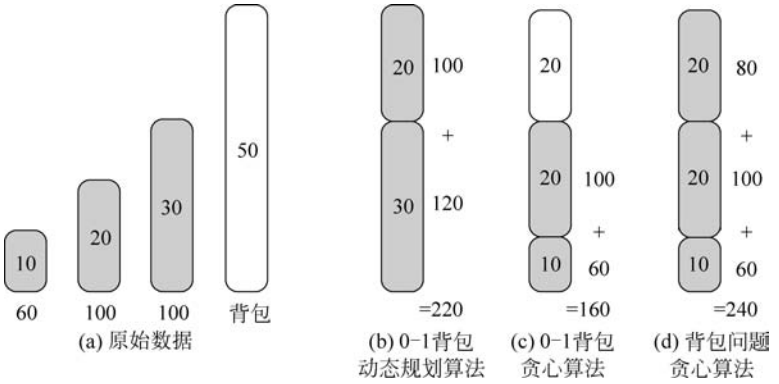

图 5-2 背包问题与 0-1 背包问题的比较

(3) 当作背包问题，采用贪心算法，按性价比从高到低的顺序选取物品，获得最优值 240，如图 5-2(d)所示。由于物品可以分割，剩下的空间 20，装入物品 3 的一部分，而获得了更好的性能。

从上面的分析可知，贪心算法对 0-1 背包问题不能得到最优解。当物品按性价比递减排序后，应用贪心算法可以得到最优解。该结论的证明，可以阅读参考文献[9]。

为了方便计算，建立如下的数据结构，表示物品的参数：

```
struct bag{
    int w;                                  //物品的质量
    int v;                                  //物品的价值
    double c;                               //性价比
}a[1001];                                   //存放物品的数组
```

排序因子(按性价比降序)：

```
bool cmp(bag a,bag b){
    if(a.c > b.c) return true;
    return false;
}
```

使用标准模板库函数排序(最好使用 stable_sort()函数，在性价比相同时保持输入的顺序)：

```
sort(a, a + n, cmp);
```

按性价比排序后，背包问题的贪心算法如算法 5.3 所示。

算法 5.3 计算背包问题的贪心算法。

```
//形参 n 是物品的数量,c 是背包的容量 M,数组 a 是按物品的性价比降序排序
```

```
double knapsack(int n, bag a[], double c)
{
    double cleft = c;                                    //背包的剩余容量
    int i = 0;
    double b = 0;                                        //获得的价值
    //当背包还能完全装入物品 i
    while(i < n && a[i].w < cleft)
    {
        cleft -= a[i].w;
        b += a[i].v;
        i++;
    }
    //装满背包的剩余空间
    if (i < n) b += 1.0 * a[i].v * cleft/a[i].w;
    return b;
}
```

算法实现源代码：Knapsack.cpp。

如果要获得解向量 $\boldsymbol{X}=\{x_1,x_2,\cdots,x_n\}$，则需要在数据结构中加入物品编号：

```
struct bag{
    int w;
    int v;
    double x;                                            //装入背包的量,0≤x≤1
    int index;                                           //物品编号
    double c;
}a[1001];
```

获得解向量时，背包问题的贪心算法如算法 5.4 所示。

算法 5.4　计算背包问题的贪心算法，同时得到解向量。

```
double knapsack(int n, bag a[], double c)
{
    double cleft = c;
    int i = 0;
    double b = 0;
    while(i < n && a[i].w <= cleft)
    {
        cleft -= a[i].w;
        b += a[i].v;
        //物品原先的序号是 a[i].index,全部装入背包
        a[a[i].index].x = 1.0;
        i++;
    }
    if (i < n)
    {
        a[a[i].index].x = 1.0 * cleft/a[i].w;
        b += a[a[i].index].x * a[i].v;
    }
    return b;
}
```

算法实现源代码：Knapsack-best.cpp。

5.4 最优装载问题

有一批集装箱要装上一艘载重量为 c 的轮船，其中集装箱 i 的重量为 w_i。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

该问题的形式化描述为：

$$\max\sum_{i=1}^{n}x_i,\quad \text{s.t.}\ \sum_{i=1}^{n}w_ix_i\leqslant c$$

其中 $x_i\in\{0,1\}$，$1\leqslant i\leqslant n$。

输入

本题有多组数据。对于每组数据，第 1 行有两个数据：第 1 个数据是轮船的载重量 c（在整数范围内），第 2 个数据是集装箱的数量 n（$1\leqslant n\leqslant 100$）。第 2 行有 n 个整数 $w_1,w_2,\cdots,w_n$，整数之间用一个空格分开，这 n 个整数依次表示这 n 个集装箱的重量（$1\leqslant w_i\leqslant 100$，$i=1,2,\cdots,n$）。

输出

对每组数据，输出轮船在装载体积不受限制的情况下，最多能装载的集装箱数以及集装箱的编号。如果一个集装箱也装不进，就输出“No answer!”。

输入样例

```
50 3
40 10 40
37 5
10 30 24 35 40
25 3
30 40 50
```

输出样例

```
2
1 2
2
1 3
No answer!
```

【算法分析】

最优装载问题可用贪心算法求解。采用重量最轻者先装的贪心选择策略，可以得到装载问题的最优解。表示集装箱的数据结构如下：

```
struct load {
    int index;                                  //集装箱编号
    int w;                                      //集装箱质量
}box[1001];
```

排序因子(按集装箱的质量升序)：

```
bool cmp(load a, load b) {
    if (a.w < b.w) return true;
    else return false;
}
```

使用标准模板库函数排序(box[0]未使用)：

```
stable_sort(box, box + n + 1, cmp);
```

这是稳定排序函数，当重量相同时，保持输入数据原来的顺序。

最优装载问题的贪心算法如算法 5.5 所示。

算法 5.5　最优装载问题的贪心算法。

```
//输入数据和数组初始化
while (scanf("%d%d", &c, &n)!= EOF)
{
    memset(box, 0, sizeof(box));
    memset(x, 0, sizeof(x));
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &box[i].w);
        box[i].index = i;
    }
    //按集装箱的质量升序排序
    stable_sort(box, box + n + 1, cmp);
    if (box[1].w > c) {
        printf("No answer!\n");
        continue;
    }
    //贪心算法的实现，质量最轻者先装载
    int i;
    for (i = 1; i <= n && box[i].w <= c; i++)
    {
        x[box[i].index] = 1;
        c -= box[i].w;
    }
    //输出装载的集装箱数量
    printf("%d\n",i - 1);
    //输出装载的集装箱编号
    for (i = 1; i <= n; i++)
        if (x[i]) printf("%d ", i);
    printf("\n");
}
```

算法实现源代码：Loading.cpp。

5.5　单源最短路径

给定带权有向图 $G=(V,E)$，其中每条边的权是非负实数。另外，还给定 V 中的一个顶点，称为源。现在要计算从源到所有其他各顶点的最短路长度，这里路的长度是指路上各

边权之和。这个问题通常称为单源最短路径问题(Single-Source Shortest Paths)。

如图 5-3 所示,就是要计算源点 v_1 到其他各个顶点的最短距离,并输出相应的路径。

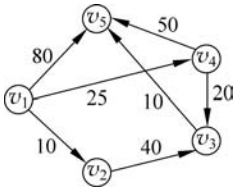

图 5-3 带权有向图 G

输入

本题有多组数据。

第 1 行有两个数据 n 和 m,其中 n 表示结点的个数,m 表示路径的数目。

接下来有 m 行,每行都有 3 个数据 s、t 和 edge,其中 s 表示路径的起点,t 表示路径的终点,edge 表示该路径的长度。

当 $n=0$,$m=0$ 时,输入数据结束。

输出

源点(统一规定为顶点 v_1,虽然其他顶点也是可以的)到所有其他各顶点的最短路径长度。

接下来有 $n-1$ 行,是从各个顶点(按升序)回到源点的路径。

输入样例

```
5 7
1 2 10
1 4 25
1 5 80
2 3 40
3 5 10
4 3 20
4 5 50
0 0
```

输出样例

```
10 45 25 55
2<--1
3<--4<--1
4<--1
5<--3<--4<--1
```

【算法分析】

Dijkstra 算法是解单源最短路径问题的一个贪心算法。其基本思想是,设置顶点集合 S 并不断地做贪心选择来扩充这个集合。一个顶点属于集合 S 当且仅当从源点到该顶点的最短路径长度已知。初始时,S 中仅含有源点。设 u 是 G 的某一个顶点,把从源点到 u 且中间只经过 S 中顶点的路径称为从源点到 u 的特殊路径,并用数组 dist 记录当前每个顶点所对应的最短特殊路径长度。Dijkstra 算法每次从 V-S 中取出具有最短特殊路径长度的顶点 u,将 u 添加到 S 中,同时对数组 dist 做必要的修改。一旦 S 包含了所有 V 中顶点,dist 就记录了从源点到所有其他顶点之间的最短路径长度。

Dijkstra 算法的实现如算法 5.6 所示。

算法 5.6 计算单源最短路径问题的 Dijkstra 算法。

```
#define NUM 100
```

```
#define maxint 10000
//顶点个数 n,源点 v,有向图的邻接矩阵为 c
//数组 dist 保存从源点 v 到每个顶点的最短特殊路径长度
//数组 prev 保存每个顶点在最短特殊路径上的前一个结点
void dijkstra(int n,int v,int dist[],int prev[],int c[][NUM])
{
    int i,j;
    bool s[NUM];                        //集合 S
    //初始化数组
    for(i=1; i<=n; i++)
    {
        dist[i] = c[v][i];
        s[i] = false;
        if (dist[i]>maxint) prev[i] = 0;
        else prev[i] = v;
    }
    //初始化源结点
    dist[v] = 0;
    s[v] = true;
    for(i=1; i<n; i++)                  //其余顶点
    {
        //在数组 dist 中寻找未处理结点的最小值
        int tmp = maxint;
        int u = v;
        for(j=1; j<=n; j++)
            if(!(s[j]) && (dist[j]<tmp))
            {
                u = j;
                tmp = dist[j];
            }
        s[u] = 1;                       //结点 u 加入 S 中
        //利用结点 u 更新数组 dist
        for(j=1; j<=n; j++)
            if(!(s[j]) && c[u][j]<maxint)
            {
                //newdist 为从源点到该点的最短特殊路径
                int newdist = dist[u]+c[u][j];
                if (newdist<dist[j])
                {
                    //修正最短距离
                    dist[j] = newdist;
                    //记录 j 的前一个结点
                    prev[j] = u;
                }
            }
    }
}
```

算法实现源代码：ShortestPaths.cpp。

如图 5-3 所示的有向图 G，邻接矩阵 c 的内容如表 5-3 所示。一对顶点(v_i,v_j)没有直

接相连的边时，$c[i][j]=\infty$。数组 dist[i]表示从源点到顶点 v_i 的最短特殊路径长度。

表 5-3　有向图 G 的邻接矩阵

行/列	1	2	3	4	5
1	∞	10	∞	25	80
2	∞	∞	40	∞	∞
3	∞	∞	∞	∞	10
4	∞	∞	20	∞	50
5	∞	∞	∞	∞	∞

对图 5-3 所示的有向图 G，应用 Dijkstra 算法计算从源顶点 v_1 到其他顶点间最短路径的过程，如表 5-4 所示。

表 5-4　从源顶点 v_1 到其他顶点的数组 dist 值和最短路径的求解过程

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初值	{1}	—	10	∞	25	80
$i=1$	{1,2}	2	10	50 (v_1,v_2,v_3)	25	80
$i=2$	{1,2,4}	4	10	45 (v_1,v_4,v_3)	25	75 (v_1,v_4,v_5)
$i=3$	{1,2,4,3}	3	10	45	25	55 (v_1,v_4,v_3,v_5)
$i=4$	{1,2,4,3,5}	5	10	45	25	55

下面分析 Dijkstra 算法的运行时间：代码的核心部分是一个双循环，所以算法时间复杂度是 $O(n^2)$。人们可能只希望找到从源点到某一特定终点的最短路径，其实这与求源点到其他所有顶点的最短路径一样复杂，其时间复杂度也是 $O(n^2)$。

根据算法 5.5 中数组 prev 保存的信息，可以输出相应的最短路径。算法中数组 prev[i]记录从源点到顶点 i 的最短路径上 i 的前一个顶点。数组 prev 的初值为 0，表示没有前一个结点。

在 Dijkstra 算法中更新最短路径长度时，只要 dist[u]+$c[u][i]$<prev[i]时，就置 prev[i]=u。当 Dijkstra 算法终止时，就可以根据数组 prev 找到从源点到顶点 i 的最短路径上每个顶点的前一个顶点，从而找到从源点到顶点 i 的最短路径。

图 5-3 中有向图的数组 prev 如表 5-5 所示。例如，要找出 v_1 到 v_5 的最短路径，根据 prev[5]=3 得到其前一个结点是 3，prev[3]=4 得到其前一个结点是 4，prev[4]=1 得到其前一个结点是 1，因此最短路径为(v_1,v_4,v_3,v_5)。

表 5-5　有向图 G 的数组 prev

结点	1	2	3	4	5
父结点	0	1	4	1	3

获得最短路径的算法如算法 5.7 所示。

算法 5.7　根据数组 prev 计算单源最短路径的算法。

```
void traceback(int v, int i, int prev[])
{
    cout << i <<"<-- ";
    i = prev[i];
    if(i!= v) traceback(v,i,prev);
    if(i == v) cout << i << endl;
}
```

如果要获得源点 v_1 到所有其他顶点的最短路径，使用迭代算法更方便，如算法 5.8 所示。

算法 5.8　根据数组 prev 计算源点 v_1 到所有其他顶点最短路径的迭代算法。

```
for(int j = 2; j <= n; j++)
{
    printf(" %d",j);
    int t = prev[j];
    while (t!= 1)
    {
        printf("<-- %d",t);
        t = prev[t];
    }
    printf("<-- 1\n");
}
```

5.6　最小生成树

设 $G=(V,E)$ 是无向连通带权图，即一个网络。对图中每一条边 (u,v) 的权为 $c[u][v]$，表示连通 u 与 v 的代价。如果 G 的子图 T 是一棵包含 G 的所有顶点的树，则称 T 为 G 的生成树。生成树上各边权的总和称为该生成树的耗费。在 G 的所有生成树中，耗费最小的生成树称为 G 的最小生成树：

$$\min\Big\{\sum_{(u,v)\in T} c[u][v]\Big\}$$

如图 5-4 所示，显示各条边的权值。粗线的边为最小生成树的边，其各边的权值之和为 37。最小生成树不是唯一的，用边 (a,h) 替代边 (b,c) 构成另外一棵最小生成树，其各边的权值之和也是 37。

网络最小生成树在实际中有广泛的应用。例如，在设计通信网络时，用图的顶点表示城市，用边 (u,v) 的权 $c[u][v]$ 表示建立城市 u 与城市 v 之间的通信线路所需要的费用，则最小生成树就给出了建立通信网络的最经济的方案。

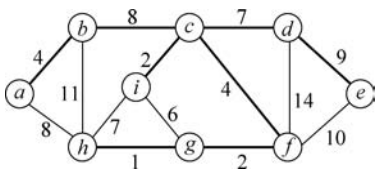

图 5-4　一个连通图的最小生成树

5.6.1　最小生成树的性质

用贪心算法设计策略可以设计出构造最小生成树的有效算法。构造最小生成树的

Prim 算法和 Kruskal 算法都是应用贪心算法设计策略的例子。这两个算法做贪心选择的方式不同,它们都利用了下面的最小生成树性质。

设 $G=(V,E)$是连通带权图,U 是 V 的真子集。如果$(u,v)\in E$,且 $u\in U$,$v\in V-U$,且在所有这样的边中,(u,v)的权 $c[u][v]$最小,那么一定存在 G 的一棵最小生成树,它以(u,v)为其中一条边。这个性质有时也称为 MST(Minimum Spanning Tree)性质。

可以用反证法证明。假设 G 的任何一棵最小生成树都不包含(u,v)。设 T 是连通图的一棵最小生成树,当将边(u,v)加入 T 中时,由生成树的定义,T 中必存在一条包含(u,v)的回路。另一方面,由于 T 是生成树,则在 T 上必存在另一条回路(u',v'),其中 $u'\in U$,$v'\in V-U$,且 u'和 u 之间,v'和 v 之间均有路径相通,如图 5-5 所示。

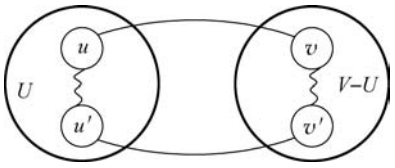

图 5-5 含边(u,v)的回路

删除边(u',v'),便可消除上述回路,同时得到另一棵生成树 T'。因为 $c[u][v]\leqslant c[u'][v']$,则 T'的代价不高于 T,于是 T'是包含(u,v)的一棵最小生成树,这与假设矛盾。

5.6.2 Prim 算法

设 $G=(V,E)$是连通带权图,$V=\{v_1,v_2,\cdots,v_n\}$。

构造 G 的最小生成树的 Prim 算法基本思想是:

首先置 $S=\{v_1\}$。重复执行下列操作:只要 S 是 V 的真子集,就做如下贪心选择——选取满足条件 $v_i\in S$,$v_j\in V-S$,且 $c[i][j]$是最小的边,将顶点 v_j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止,且选取到的所有边恰好构成 G 的一棵最小生成树,如算法 5.9 所示。

算法 5.9 构成最小生成树的 Prim 算法描述。

```
define NUM 1000
void Prim(int n,int c[][NUM])
{
    T = ϕ;                                  //生成树边的集合
    S = {v1};                               //生成树顶点的集合
    while (S!= V)
    {
        (vi,vj) = vi∈S 且 vj∈V-S 的最小权边;
        T = T∪{(vi,vj)};
        S = S∪{vj}
    }
}
```

算法结束时,T 中包含 G 的 $n-1$ 条边,S 中包含 G 的 n 个顶点。

例如,对于图 5-6(a)所示的无向连通带权图,按 Prim 算法选取边的过程如图 5-6(b)~(f)所示。

对于构成最小生成树的 Prim 算法,还应当考虑如何有效地找出满足条件 $v_i\in S$,$v_j\in$

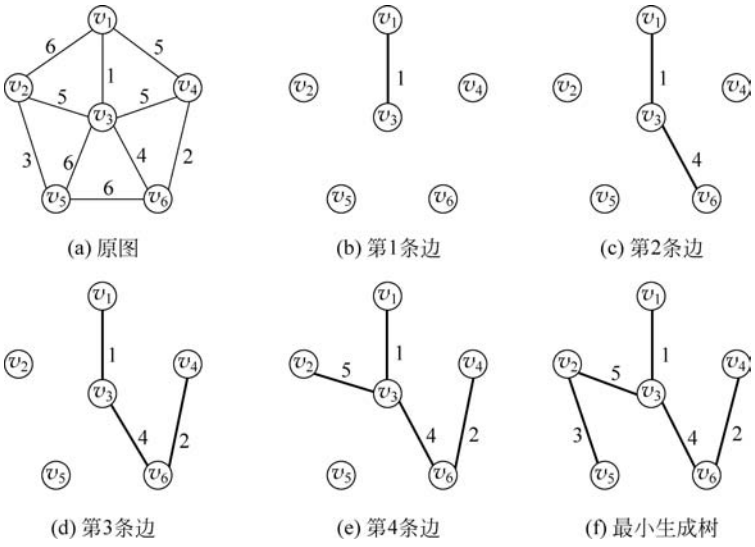

图 5-6　使用 Prim 算法构造最小生成树的过程

$V-S$，且 $c[i][j]$是最小的边(v_i, v_j)。设无向连通带权图的邻接矩阵为数组 c，一对顶点(v_i, v_j)没有直接相连的边时，$c[i][j]=\infty$。

采用两个数组 closest 和 lowcost 实现这个目标。对于每一个 $v_j \in V-S$，closest[j]是 v_j 在 S 中的邻接顶点，它与 v_j 在 S 中的其他邻接顶点 v_k 比较，有：

$$c[j][\text{closest}[j]] \leqslant c[j][k], \quad \text{lowcost}[j]=c[j][\text{closest}[j]]$$

在 Prim 算法的执行过程中，先找出 $V-S$ 中使 lowcost 值最小的顶点 v_j，然后根据数组 closest 选取边$(v_j, \text{closest}[j])$，最后将 v_j 添加到 S 中。然后根据 v_j 对 closest 和 lowcost 做必要的修改，如算法 5.10 所示。

算法 5.10　使用 Prim 算法构成最小生成树。

```
#define NUM 1000
#define maxint 10000000                    //表示∞
//形参 n 为顶点的个数,数组 c 为无向连通带权图的邻接矩阵
void Prim (int n, int c[][NUM])
{
    int lowcost[NUM];                      //类似于算法 5.6 中的 dist 数组
    int closest[NUM];                      //类似于算法 5.6 中的 prev 数组
    bool s[NUM] = {false};
    //初始化数组
    for (int i = 1; i <= n; i++)
    {
        lowcost[i] = c[1][i];
        closest[i] = 1;
        s[i] = false;
    }
    //从顶点 v1 开始
    s[1] = true;
    //处理其余 n-1 个顶点
    for (int i = 1; i < n; i++)
```

```
    {
        //在未处理的结点集中,找最小边权的结点 vj
        int min = maxint;
        int j = 1;
        for(int k = 2; k <= n; k++)
            if((lowcost[k]< min) && (!s[k]) )
            {
                min = lowcost[k];
                j = k;
            }
        //一条边(vj,closest[j])
        printf("%d %d\n", closest[j], j);
        //加入结点 vj
        s[j] = true;
        //根据结点 vj,更新数组 lowcost 和 closest
        for(int k = 2; k <= n; k++)
            if((c[j][k]< lowcost[k]) && (!s[k]))
            {
                lowcost[k] = c[j][k];
                closest[k] = j;
            }
    }
}
```

算法实现源代码：MST-Prim.cpp。

下面分析 Prim 算法的运行时间：代码的核心部分是一个双循环，所以算法的时间复杂度是 $O(n^2)$。该算法与图中的边数无关，因此适用于计算边稠密的最小生成树。而 Kruskal 算法恰恰相反，它的计算时间复杂度为 $O(e\log_2 e)$（其中 e 为图中边的数量），因此相对于 Prim 算法，Kruskal 算法更适合于计算边稀疏的最小生成树。

5.6.3 Kruskal 算法

设 $G=(V,E)$是连通带权图，$V=\{v_1,v_2,\cdots,v_n\}$。

Kruskal 算法构造 G 的最小生成树的基本思想是：首先将 G 的 n 个顶点看成 n 个孤立的连通分量，将所有的边按权从小到大排序。然后从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接两个不同的连通分量。当查看到第 i 条边(u,v)时，如果端点 u 和 v 分别是当前两个不同的连通分量 T_1 和 T_2 中的顶点时，就用边(u,v)将 T_1 和 T_2 连接成一个连通分量，然后继续查看第 $i+1$ 条边；如果端点 u 和 v 在当前的同一个连通分量中，就直接再查看第 $i+1$ 条边。这个过程一直进行到只剩下一个连通分量时为止，该连通分量就是 G 的一棵最小生成树。

例如，图 5-7(b)～(f)所示是按照 Kruskal 算法构造 G 的最小生成树的过程。开始时，每个顶点都是一棵树。

采用抽象数据类型并查集的基本运算，可以很方便地实现 Kruskal 算法。

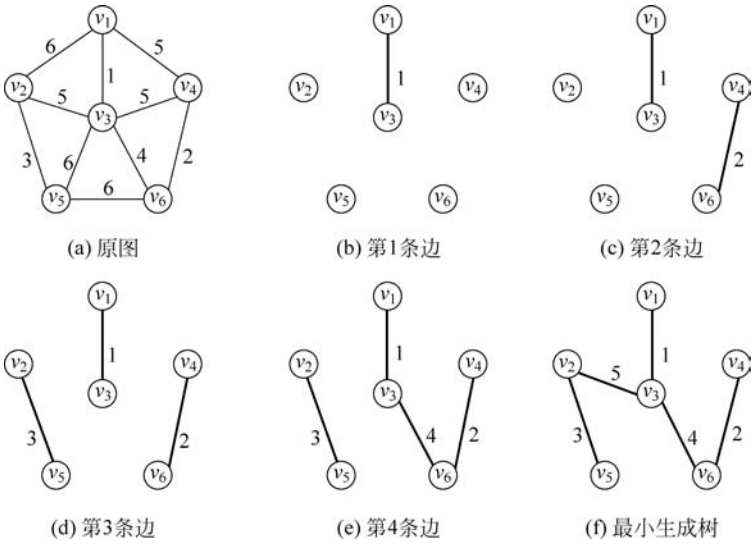

图 5-7　使用 Kruskal 算法构造最小生成树的过程

(1) 建立边的数据结构。

```
#define N 1000
struct graph
{
    int u,v,cost;                          //顶点 u 与顶点 v 的边权是 cost
    void set(int a,int b,int w){u = a,v = b,cost = w;}
};
graph d[N * (N + 1)/2];
```

对于有 n 个顶点的图，其边的数量 $e \leqslant n(n+1)/2$。

假设有 m 条边，每条边的描述是$\{v,w,\text{edge}\}$，表示顶点 v 与顶点 w 的边权是 edge，则可以按如下方式构造边集合 d：

```
int i;
int v,w,edge;
for (i = 0; i < m; i++)
{
    scanf("%d%d%d", &v, &w, &edge);
    d[i].set(v,w,edge);
}
```

(2) 按边权 cost 的递增顺序排序。

排序因子：

```
int cmp(graph x,graph y)
{
    if (x.cost < y.cost) return true;
    return false;
}
```

使用 C++标准模板库函数排序：

```
sort(d, d + m, cmp);
```

(3) 应用并查集的基本运算,实现 Kruskal 算法构造 G 的最小生成树。

并查集是一种树型的数据结构,用于处理一些不相交集合(Disjoint Sets)的合并及查询问题。常常在使用中以森林来表示。

① 查找一个元素所在的集合。查找一个元素所在的集合,只要找到这个元素所在集合的祖先。

先设置一个数组 Father[x],表示 x 的"父亲"的编号,即可转换为寻找两个元素的最久远祖先是否相同,可以采用递归实现,如算法 5.11 所示。

算法 5.11 查找一个元素所在集合的算法。

```
int father[N];
int Find(int x)
{
    if (father[x] == -1) return x;
    return father[x] = Find(father[x]);
}
```

② 合并两个不相交集合。合并两个不相交集合的方法就是,找到其中一个集合最久远的祖先,将另外一个集合最久远的祖先指向它,如算法 5.12 所示。

算法 5.12 合并两个不相交集合的算法。

```
bool Union(int x, int y)
{
    //判断两个元素是否属于同一集合,只要看它们所在集合的祖先是否相同
    x = Find(x);
    y = Find(y);
    //祖先相同,是同一个集合
    if (x == y) return false;
    //祖先不相同,合并两个集合
    if (x > y) father[x] = y;
    if (x < y) father[y] = x;
    return true;
}
```

③ 应用并查集,实现 Kruskal 算法构造最小生成树。因为已经按边权的大小升序排序,所以最先找到满足条件的边就是最小生成树中的一条边,如算法 5.13 所示。

算法 5.13 使用 Kruskal 算法构造最小生成树。

```
memset(father, -1, sizeof(father));
//最小生成树中边权的和
int sum = 0;
//已经找到的边数
int count = 0;
//查找每一条边
for(i = 0; i < n * (n + 1)/2; i++)
{
    //找到了一条满足条件的边
    if (Union(d[i].u, d[i].v))
```

```
        {
            printf("%d %d\n", d[i].u, d[i].v);
            sum += d[i].cost;
            ++count;
        }
        if (count == n - 1) break;
    }
    printf("%d\n",sum);
```

算法实现源代码：MST－Kruskal.cpp。

按边权 cost 递增排序以后，算法 5.13 使用 Kruskal 算法构造最小生成树时，count 的计数过程如表 5-6 所示。

表 5-6　使用 Kruskal 算法构造最小生成树时 count 的计数过程

i	0	1	2	3	4	5	6	7	8	9
(u,v)	(1,3)	(4,6)	(2,5)	(3,6)	(1,4)	(2,3)	(3,4)	(1,2)	(3,5)	(5,6)
cost	1	2	3	4	5	5	5	6	6	6
count	1	2	3	4		5				

对图 5-7 所示的图 G，算法 5.12 的输出结果是：

```
1 3
4 6
2 5
3 6
2 3
15
```

前 5 行刚好与图 5-7(b)～(f)对应，最后一行 15 是构成的最小生成树的边权和。与图 5-7 对应的数组 father 的值，如表 5-7 所示。其中下标表示结点，对应的数值表示该结点的父结点。

表 5-7　与图 5-7 对应的数组 father 的值

下标(结点)	1	2	3	4	5	6
父结点	－1	1	1	1	2	4

5.7　删数问题

给定 n 位正整数 a，去掉其中任意 $k \leqslant n$ 个数字后，剩下的数字按原次序排列组成一个新的正整数。对于给定的 n 位正整数 a 和正整数 k，设计一个算法，找出剩下数字组成的新数最小的删数方案。

输入

第 1 行是 1 个正整数 a，第 2 行是正整数 k。

输出

对于给定的正整数 a，编程计算删去 k 个数字后得到的最小数。

输入样例

```
178543
4
```

输出样例

```
13
```

【算法分析】

n 位数 a 可表示为 $x_1x_2\cdots x_ix_jx_k\cdots x_n$，要删去 k 位数，使得剩下的数字组成的整数最小。将该问题记为 T，其最优解 $A=\{x_{i1}x_{i2}x_{i3}\cdots x_{im}\}$，其中 $i_1<i_2<\cdots<i_m$，$m=n-k$，在删去 k 个数后剩下的数字按原次序排成的新数，其最优值记为 N。

本问题采用贪心算法求解，采用最近下降点优先的贪心策略，即 $x_1<x_2<\cdots<x_{i-1}<x_i$，如果 $x_{i+1}<x_i$（下降点），则删去 x_i，即得到一个新的数且这个数为 $n-1$ 位中最小的数 N_1，可表示为 $x_1x_2\cdots x_{i-1}x_{i+1}\cdots x_n$。显然删去 1 位数后，原问题 T 变成了对 $n-1$ 位数删去 $k-1$ 个数的新问题 T'。新问题和原问题的性质相同，只是问题规模由 n 减小为 $n-1$，删去的数字个数由 k 减少为 $k-1$。基于此种删除策略，对新问题 T'，选择最近下降点的数继续进行删除，直至删去 k 个数为止，如算法 5.14 所示。

算法 5.14 删数问题的贪心算法实现。

```
string a;                                   //n 位数 a
int k;
cin >> a >> k;
//如果 k≥n,数字被删完了
if(k >= a.size()) a.erase();
else while(k > 0)
{
    //寻找最近下降点
    int i;
    for(i = 0;(i < a.size() - 1) && (a[i] <= a[i + 1]); ++i);
    a.erase(i,1);                           //删除 xi
    k -- ;
}
//删除前导数字 0
while(a.size() > 1 && a[0] == '0')
    a.erase(0,1);
cout << a << endl;
```

算法实现源代码：DeleteNumber.cpp。

5.7.1 问题的贪心选择性质

对问题 T 删除最近下降点的数 x_i 后得到的 N_1 是 $n-1$ 位数中最小的数。

根据数的进制特点，对 a 按权展开后得到：

$$a=x_1\times10^{n-1}+x_2\times10^{n-2}+\cdots+x_{i-1}\times10^{n-i+1}+$$
$$x_i\times10^{n-i}+x_{i+1}\times10^{n-i-1}+\cdots+x_n$$

则有：

$N_1 = x_1 \times 10^{n-2} + x_2 \times 10^{n-3} + \cdots + x_{i-1} \times 10^{n-i} + x_{i+1} \times 10^{n-i-2} + \cdots + x_n$

假设删去的不是 x_i 而是后面的 x_{i+1}，则有：

$N_2 = x_1 \times 10^{n-2} + x_2 \times 10^{n-3} + \cdots + x_{i-1} \times 10^{n-i} + x_i \times 10^{n-i-1} + \cdots + x_n$

因为有 $x_1 < x_2 < \cdots < x_{i-1} < x_i$，且 $x_{i+1} < x_i$（下降点），则有 $N_1 < N_2$。

因此删数问题满足贪心选择性质。

5.7.2　问题的最优子结构性质

在进行了贪心选择后，原问题 T 就变成了对 N_1 如何删去 $k-1$ 个数的问题 T'，是原问题的子问题。若 $A=(x_i, A')$ 是原问题 T 的最优解，则 A' 是子问题 T' 的最优解，其最优值为 N_A。

采用反证法证明：假设 A' 不是子问题 T' 的最优解，其子问题的最优解为 B'，其最优值记为 N_B，则有 $N_B < N_A$。而根据 N 的定义可知：$N = N_A + x_i \times 10^{n-i}$，而 $N_B < N_A$，因此有 $N_B + x_i \times 10^{n-i} < N_A + x_i \times 10^{n-i}$。即存在一个由数 a 删去 1 位数后得到的 $n-1$ 位数比最优值 N 更小，这与 N 为问题 T 的最优值相矛盾，则 A' 是子问题 T' 的最优值。

因此，删数问题满足最优子结构性质。

5.8　多处最优服务次序问题[①]

设有 n 个顾客同时等待一项服务，顾客 i 需要的服务时间为 t_i，$1 \leqslant i \leqslant n$，共有 s 处可以提供此项服务。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小？平均等待时间是 n 个顾客等待服务时间的总和除以 n。

对于给定的 n 个顾客需要的服务时间和 s 的值，编程计算最优服务次序。

输入

第一行有 2 个正整数 n 和 s，表示有 n 个顾客且有 s 处可以提供顾客需要的服务。

接下来的 1 行中，有 n 个正整数，表示 n 个顾客需要的服务时间。

输出

最小平均等待时间，输出保留 3 位小数。

输入样例

```
10 2
56 12 1 99 1000 234 33 55 99 812
```

输出样例

```
336.000
```

【算法分析】

假设原问题为 T，并已经知道某个最优服务序列，即最优解为 $A=\{t_1, t_2, \cdots, t_n\}$，其中

① 李洪霞，于仁师，等. 用贪心算法求解最优服务次序问题[J]. 科技广场，2008(3)，18-20.

t_i 为第 i 个用户需要的服务时间，则每个用户等待时间 T_i 为：

$$T_1 = t_1$$
$$T_2 = t_1 + t_2$$
$$\vdots$$
$$T_n = t_1 + t_2 + \cdots + t_n$$

那么总的等待时间，即最优值 N 为：

$$N = nt_1 + (n-1)t_2 + (n-2)t_3 + \cdots + 2t_{n-1} + t_n$$

由于平均等待时间是 n 个顾客等待时间的总和除以 n，故本题实际上就是求使顾客等待时间的总和最小的服务次序。

设计贪心策略如下：对服务时间最短的顾客先服务。首先对需要服务时间最短的顾客进行服务，即做完第一次选择后，原问题 T 变成了需对 $n-1$ 个顾客服务的新问题 T'。新问题和原问题相同，只是问题规模由 n 减小为 $n-1$。基于此种选择策略，对新问题 T'，在 $n-1$ 个顾客中选择服务时间最短的先进行服务，如此进行下去，直至所有服务都完成为止，如算法 5.15 所示。

算法 5.15 多处最优服务次序问题的贪心算法实现。

```
//顾客等待的队列为 client,提供服务的窗口 s 个
double greedy(vector < int > client,int s)
{
    //服务窗口的顾客等待时间
    vector < int > service(s + 1,0);
    //服务窗口顾客等待时间的总和
    vector < int > sum(s + 1,0);
    //顾客的数量
    int n = client.size();
    //按顾客的服务时间升序排序
    sort(client.begin(),client.end());
    //贪心算法的实现
    int i = 0;                          //顾客的指针
    int j = 0;                          //窗口的指针
    while(i < n)
    {
        service[j] += client[i];
        sum[j] += service[j];
        ++i,++j;
        if(j == s) j = 0;
    }
    //计算所有窗口服务时间的总和
    double t = 0;
    for(i = 0; i < s; ++i) t += sum[i];
    t /= n;
    return t;
}
```

算法实现源代码：serviceProblem.cpp。

算法 5.15 中使用 Vector 创建动态数组，相关知识请参考 2.2 节。

对于样例数据，计算过程如图 5-8 所示。

排序后的数组client

i	0	1	2	3	4	5	6	7	8	9
x	1	12	33	55	56	99	99	234	812	1000

多处最优服务次序问题贪心算法的计算过程

数组	service		sum	
j	0	1	0	1
i=0, 1	1	12	1	12
i=2, 3	34	67	35	79
i=4, 5	90	166	125	245
i=6, 7	189	400	314	645
i=8, 9	1001	1400	1315	2045

图 5-8　多处最优服务次序问题贪心算法的计算过程

贪心算法实现部分是一重循环，其时间复杂度为 $O(n)$。主要时间花费是前面的排序，排序函数 sort()的时间复杂度是 $O(n\log_2 n)$，所以该算法总的时间复杂度是 $O(n\log_2 n)$。

5.8.1　问题的贪心选择性质

按顾客服务的时间升序排序，获得最优服务序列，即最优解 $A=\{t_1,t_2,\cdots,t_n\}$，其中 $t_i\leqslant t_{i+1}$，$1\leqslant i\leqslant n-1$，此时的最优值为：

$$N=nt_1+(n-1)t_2+(n-2)t_3+\cdots+2t_{n-1}+t_n$$

采用反证法证明：假设 t_1 不是最小的，不妨设 $t_1>t_i(i>1)$。

设另一服务序列 $B=\{t_i,t_2,\cdots,t_1,\cdots,t_n\}$，最优值为 N'，则：

$$N-N'=n\times(t_1-t_i)+(n+1-i)(t_i-t_1)=(1-i)(t_i-t_1)>0$$

即 $N>N'$，这与 A 是最优服务序列相矛盾，故最优服务次序问题满足贪心选择性质。

5.8.2　问题的最优子结构性质

在进行了贪心选择后，原问题 T 就变成了如何安排剩余的 $n-1$ 个顾客的服务次序的问题 T'，是原问题的子问题。若 A 是原问题 T 的最优解，则 $A'=\{t_2,t_3,\cdots,t_n\}$ 是服务次序问题子问题 T' 的最优解，其最优值为 N_A。

采用反证法证明：假设 A' 不是子问题 T' 的最优解，其子问题的最优解为 B'，相应的最优值为 N_B，则有 $N_B<N_A$。根据 N 的定义知，$N_A+t_1=N$，因此 $N_B+t_1<N_A+t_1=N$，即存在一个比最优值 N 更短的总等待时间，而这与 N 为问题 T 的最优值相矛盾。因此，A' 是子问题 T' 的最优值。

从以上贪心选择及最优子结构性质的证明，可知最优服务次序问题用贪心算法可求得最优解。平均等待时间即为 N/n。

5.9　ZOJ1012-Mainframe

【问题描述】

多纳(Ronald)先生是 ACM(Agent on Computing of Mathematics，计算数学代理商)大

型计算机的管理员。该代理商为一些公司承担在大型计算机上的计算工作,完成工作后获得报酬,因此大型计算机对这个代理商来说太重要了。多纳先生需要为这些在大型计算机上运行的作业安排顺序。一旦要运行某项作业,他就要检查运行该作业所需要的空闲资源。如果空闲资源足够的话,他就为该作业分配这些资源;否则就将该作业挂起,直至有足够的资源投入运行。

刚开始他并不熟悉这项工作,把事情搞得很乱。日积月累,他就胜任这项工作了,而且他还总结了如下一套规则:

(1) 大型计算机有 M 个 CPU 和 N 大小的内存空间可供分配。

(2) 对等待运行的作业有一个等待队列。可以假定这个队列足够长,能够存放所有等待运行的作业。

(3) 假定作业 J_i 需要 A_i 个 CPU 和 B_i 的内存空间,在时间 T_i 到达等待队列,需要在时间 U_i 之前完成。成功运行后,代理商可以获得 V_i(美元)的报酬;如果能在规定的时间之前完成,则每小时还可以额外获得 W_i(美元)的奖金;如果工期拖延,则每小时赔偿 X_i(美元)。例如,假定一项作业的报酬是 10 美元,时限 8 小时,每拖欠 1 小时罚 2 美元。如果该作业在 10 小时完成,则代理商可以获得报酬:10−(10−8)×2=6 美元。

(4) 当一项作业开始后,就独占了分配给它的 CPU 和内存空间,不能同时再分配给其他的作业。当该作业运行结束后,这些资源被释放。如果资源足够多,同时可以运行多项作业。

(5) 为了最大限度地发挥大型计算机的计算能力,每项作业在开始运行后刚好 1 小时就完成。你可以假定每项作业的运行时间就是 1 小时。

(6) 没有作业运行时,机器空闲,一直到一项作业进入作业等待队列。

(7) 如果有多项作业进入作业等待队列,则报酬高的作业优先。可以假定这些作业的报酬都不相等($V_i \neq V_j$)。

(8) 如果某项作业的空闲 CPU 或内存空间不能满足,它就要被挂起 1 小时,也不占用任何资源。一小时后,再次为该作业检查资源,而不需要考虑等待队列里的其他作业。如果资源仍不满足要求,那就继续挂起 1 小时,把资源分配给其他在等待队列里的作业;否则,该作业将独占 CPU 和存储空间并投入运行。

(9) 当多项作业挂起时,采取先来先服务的原则。

使用这些规则,多纳先生把事情安排得井井有条。现在除了日常公务外,代理商要求他根据作业列表计算收入。给定某个时间 F,计算出已经完成的作业和应该被完成的作业。对作业 J_i,如果它的时限 $U_i > F$ 并且仍未完成,就不需要统计收入。对已经完成的作业或 $U_i \leqslant F$ 的作业都要统计。如果工作没有完成,它不会给代理商带来报酬,但到这个时间 F 为止的罚款仍要计算。

他不会程序设计,又不想手工做,现在很不安。你可以帮助他解决这个问题吗?

输入

有多组测试例。每个测试例描述大型计算机的资源和作业列表,第一行是整数 F($0 \leqslant F \leqslant 10\,000$),表示时限;接下来的一行是三个整数 M,N 和 L(M,N,$L \geqslant 0$),M 是机器 CPU 的数量,N 是存储空间的大小,L 是作业等待队列中作业的数量。最多有 10 000 项作业。

后面的 L 行是作业的信息。描述作业 J_i 的数据是 7 个整数：A_i，B_i，T_i，U_i，V_i，W_i 和 X_i。A_i 和 B_i（A_i，$B_i \geqslant 0$）指出了该作业对 CPU 和内存的需求；T_i 和 U_i 表示作业的到达时间和时限（$0 \leqslant T_i \leqslant U_i$）；$V_i$，$W_i$，$X_i$ 分别是工作的报酬、奖励和罚款（V_i，W_i，$X_i \geqslant 0$）。

一个空的测试例（$F=0$）表示输入结束，该测试例无须处理。

输出

根据作业列表算总收入。对每个测试例，都输出测试例编号，一个冒号，一个空格，然后是收入。

测试例之间有一个空行。

注意：对尚未投入运行的且时限超过 F 的作业，不必统计。

输入样例

```
10
4 256 3
1 16 2 3 10 5 6
2 128 2 4 30 10 5
2 128 2 4 20 10 5
0
```

输出样例

```
Case 1: 74
```

题目来源

Asia 2001，*Shanghai*（*Mainland China*）

【算法分析】

在进行资源分配时，运用贪心策略能够获得很好的效率。

1. 样例分析

时限 $F=10$；

CPU 数量 $M=4$，存储空间 $N=256$，作业数量 $L=3$。

3 个作业的参数如表 5-8 所示。

表 5-8 样例作业的参数

作业编号	A（CPU）	B（内存空间）	T（到达时间）	U（时限）	V（报酬）	W（奖励）	X（罚款）
0	1	16	2	3	10	5	6
1	2	128	2	4	30	10	5
2	2	128	2	4	20	10	5

按规则(9)排序：先来先服务的原则，根据到达时间排序；而到达时间是一样的，按规则(7)排序：报酬高的作业优先。排序后的结果如表 5-9 所示。

表 5-9 样例作业的排序结果

作业编号	A（CPU）	B（内存空间）	T（到达时间）	U（时限）	V（报酬）	W（奖励）	X（罚款）
1	2	128	2	4	30	10	5
2	2	128	2	4	20	10	5
0	1	16	2	3	10	5	6

前面两项作业，在时间 2 到达，刚好有 4 个 CPU 和 256 个存储空间可用，立即投入运行，而且在时限 4 之前结束，既有报酬又有奖金(提前 1 小时)。

作业 1 收益：30＋10。

作业 2 收益：20＋10。

这两项作业在时间 3 运行结束，接着运行作业 3，至时间 4 结束，晚点时间 1 小时：

作业 3 收益：10－6＝4。

总收益是 74。

2. 数据结构

对每项作业，使用一个结构体表示其 7 个参数：

```
#define Max 10001
struct Node{
    int a,b;                          //分别为 CPU 和内存空间
    int t,u;                          //分别为到达时间和时限
    int v,w,x;                        //分别为报酬、奖励和罚款
};
```

对所有作业，使用数组 Job 表示：

```
Node Job[Max];
```

作业完成的状态保存到数组 Finished 中：

```
bool Finished[Max];
```

3. 作业排序

应用 sort()函数对作业进行排序：

```
sort(Job, Job + L, cmp);
```

其中比较因子 cmp()函数的实现：

```
bool cmp(Node a , Node b)
{
    //根据到达时间优先排序(升序)
    if (a.t!= b.t) return a.t < b.t;
    //到达时间相等时,按报酬数量降序排序
    else return a.v > b.v;
}
```

4. 报酬的计算

排序之后就可以实现贪心策略了。

在统计时限 F 之前到达的所有作业，能够运行的就计酬，不能运行的就罚款。

按照排好的顺序，依次将作业投入运行，资源足够的就投入运行，资源不够的作业就等待下一个小时。由于所有作业都只需要运行 1 小时，因此在每小时开始时，资源的数量都是原始数量，如算法 5.16 所示。

算法 5.16　大型计算机作业调度问题的贪心算法实现。

```
int i,j;
int iCase = 0;                                  //测试例编号
int F;                                          //统计时限
while(scanf("%d",&F) && F)
{
    iCase++;
    int Profit = 0;                             //报酬
    memset(Finished, 0, sizeof(Finished));
    //分别表示 CPU、内存空间和作业的数量
    int M,N,L;
    scanf("%d%d%d",&M, &N, &L);
    for(i = 0; i<L; i++)
        scanf("%d%d%d%d%d%d", &Job[i].a,
            &Job[i].b,&Job[i].t, &Job[i].u,
            &Job[i].v, &Job[i].w, &Job[i].x);
    //按函数 cmp()的要求排序
    sort(Job, Job+L, cmp);
    int count = 0;                              //统计完成作业的数量
    //在统计时限内完成 L 个作业
    for(i = 0; i<F && count<L; i++)
    {                                           //每一小时开始时,所有资源都是空闲的
        int cpu = M;
        int memory = N;
        //对所有的作业搜索一遍
        for(j = 0; j<L; j++)
        {
            //该时刻作业还没有到达
            if(Job[j].t>i) break;
            //如果作业还没有运行,并且资源足够
            if(!Finished[j] && cpu>=Job[j].a && memory>=Job[j].b)
            {
                cpu -= Job[j].a;                //占有资源
                memory -= Job[j].b;
                Finished[j] = 1;                //设置完成标志
                Profit += Job[j].v;             //计算报酬
                if(i+1<= Job[j].u)              //计算奖金
                    Profit += (Job[j].u-i-1)*Job[j].w;
                else                            //计算罚款
                    Profit -= (i+1-Job[j].u)*Job[j].x;
                count++;                        //完成了一项作业
            }
```

```
            }
        }
        //在统计时限 F 之前到达的作业,没有运行的就要罚款
        for(i = 0; i<L; i++)
            if(!Finished[i]&&Job[i].u<= F)
                Profit -= (F-Job[i].u)*Job[i].x;
        printf("Case %d: %d\n\n", iCase, Profit);
    }
```

算法实现源代码：zju1012.cpp。

5.10 ZOJ1025-Wooden Sticks

【问题描述】

现有 n 根木棒,已知它们的长度和质量。要用一部木工机一根一根地加工这些木棒。该机器在加工过程中需要一定的准备时间,是用于清洗机器,调整工具和模板的。

木工机需要的准备时间如下：

(1) 第一根木棒需要 1min 的准备时间；

(2) 在加工了一根长为 l,重为 w 的木棒之后,接着加工一根长为 l'($l\leqslant l'$),重为 w'($w\leqslant w'$)的木棒是不需要任何准备时间的,否则需要 1min 的准备时间。

给定 n 根木棒,你要找到最少的准备时间。例如现在有长和重分别为(4,9),(5,2),(2,1),(3,5)和(1,4)的 5 根木棒,那么所需准备时间最少为 2min,顺序为(1,4),(3,5),(4,9),(2,1),(5,2)。

输入

输入有多组测试例。输入数据的第一行是测试例的个数 T。

每个测试例两行：第一行是一个整数 n($1\leqslant n\leqslant 5000$),表示有多少根木棒；第二行包括 $n\times 2$ 个整数,表示 $l_1,w_1,l_2,w_2,l_3,w_3,\cdots,l_n,w_n$,全部不大于 10 000,其中 l_i 和 w_i 表示第 i 根木棒的长度和质量。数据由一个或多个空格分隔。

输出

输出是以分钟为单位的最少准备时间,一行一个。

输入样例

```
3
5
4 9 5 2 2 1 3 5 1 4
3
2 2 1 1 2 2
3
1 3 2 2 3 1
```

输出样例

```
2
1
3
```

题目来源

Asia 2001，*Taejon*（*South Korea*）

【算法分析】

对这种资源调度问题，贪心算法能够获得很好的效率。

但本题仅仅使用贪心算法是不够的，排序之后还要使用动态规划的算法，所以在网上会有把本题归类为贪心算法和动态规划算法。

1. 数据结构

采用结构体表示木棒的信息：

```
#define maxN 5001
struct stick
{
    int l;                                  //木棒的长度
    int w;                                  //木棒的质量
};
stick data[maxN];                           //存放所有木棒
```

2. 按木棒的长度使用贪心算法

利用 C++的标准模板库函数 sort()实现排序：

```
sort(data, data + n, cmp);
```

排序函数 cmp()的实现：

```
int cmp(stick a, stick b)
{
    //长度相等时,按质量排序
    if(a.l == b.l)
        return a.w < b.w;
    //优先按长度排序
    else if (a.l < b.l) return true;
    return false;
}
```

对于样例数据 1，排序之后的结果如表 5-10 所示。

表 5-10　样例数据 1 排序之后的结果

下标	0	1	2	3	4
长度 l	1	2	3	4	5
质量 w	4	1	5	9	2
数组 b	1	2	1	1	2

3. 使用动态规划的方法，计算质量 w 的单调递增子序列的个数

从表 5-10 直观地看到，对质量 w，4，5 和 9 是一组，1 和 2 是一组，所以答案是 2。

用数组 b 记录质量 w 的分组序号，在表 5-10 中，4，5 和 9 的组序号是 1，1 和 2 的组序号是 2。则 $a[i].w(0 \leqslant i < n)$的递增子序列的分组个数为：$\max\limits_{0 \leqslant i < n}\{b[i]\}$。

$b[i]$满足最优子结构性质,可以递归地定义为:

$$b[0]=1;$$

$$b[i]=\max_{\substack{0\leqslant i<n\\ a[i].w<a[j].w}}\{b[j]\}+1 \quad (0\leqslant j<i)$$

该算法统计质量 w 的单调递增子序列的个数,而对每个递增子序列,并不一定是最长的。例如数据:(3170),(4155),(5239),(6300),(7240),(8241),(9369),数组 b 的值为(1,2,1,1,2,2,1),对应数组 b 为 1 的 w 序列为(170,239,300,369),而最长单调递增子序列是(170,239,240,241,369)。程序实现如算法 5.17 所示。

算法 5.17 计算质量 w 的单调递增子序列个数的动态规划实现。

```
//形参 n 是木棒的数量,stick 是木棒参数的数组
int DP(int n, stick a[])
{
    //数组 b 表示木棒分组的序号
    int b[maxN];
    memset(b, 0, sizeof(b));
    int i, j, k;
    b[0] = 1;
    for (i = 1; i < n; i++)
    {
        //计算第 i 个木棒的分组序号
        k = 0;
        //在 i 的前面,比 a[i]大的单元中,找 b 的最大值,然后 k + 1
        for (j = 0; j < i; j++)
            if (a[i].w < a[j].w && k < b[j]) k = b[j];
        b[i] = k + 1;
    }
    //查找最大的分组序号(数组 b 中的最大值)
    int max = 0;
    for (i = 0; i < n; i++)
        if (b[i]> max) max = b[i];
    return max;
}
```

算法实现源代码:zju1025-sort.cpp。

既然是统计质量 w 的单调递增子序列的个数,则可以应用活动安排问题的算法,找到第一个分组;对剩下的数据继续应用活动安排问题的算法,找到第二个分组;以此类推,直到所有的质量 w 都已经安排。最后分组的数量 count 就是答案。

算法实现源代码:zju1025-greedy.cpp。

5.11 ZOJ1029-Moving Tables

【问题描述】

著名的 ACM(Advanced Computer Maker)公司租用了一层有 400 个房间的办公楼,结构如图 5-9 所示。

房间 1	房间 3	房间 5	…	房间 397	房间 399
走廊					
房间 2	房间 4	房间 6	…	房间 398	房间 400

图 5-9　ACM 公司办公楼示意图

这层楼沿着走廊南北向的两边各有 200 个房间。最近，公司要做一次装修，需要在各个办公室之间搬运办公桌。由于走廊狭窄，办公桌都很大，走廊里一次只能通过一张办公桌。必须制订计划提高搬运效率。经理制订如下计划：一张办公桌从一个房间移到另一个房间最多用 10 分钟。当从房间 i 移动一张办公桌到房间 j，两个办公室之间的走廊都会被占用。所以，每 10 分钟内，只要不是同一段走廊，都可以在房间之间移动办公桌。为了说得更清楚一些，经理举例说明哪些情况可以同时移动，哪些情况不能同时移动，如表 5-11 所示。

表 5-11　办公桌移动的状态

分类	移动办公桌	理由
可行的	(房间 30 到 50)和(房间 60 到 90)	走廊不重叠
	(房间 11 到 12)和(房间 14 到 13)	走廊不重叠
不可行的	(房间 20 到 40)和(房间 31 到 80)	房间 31 到房间 40 的走廊重叠
	(房间 1 到 4)和(房间 3 到 6)	房间 3 前面的走廊重叠
	(房间 2 到 8)和(房间 7 到 10)	房间 7 前面的走廊重叠

每个房间，只有一张办公桌进出。现在，经理想找到一种方案，使移动桌子的事情尽快完成。请编写程序解决经理的难题。

输入

输入数据有 T 组测试例，在第一行给出测试例个数 T。

每个测试例的第一行是一个整数 $N(1\leqslant N\leqslant 200)$，表示要搬运办公桌的次数。接下来的 N 行，每行都有两个正整数 s 和 t，表示一张桌子是从房间号码 s 移到房间号码 t。

有多组输入数据，输入第一行为一个表示输入数据总数的整数 N，然后是 N 组输入数据。

输出

每组输入都有一行的输出数据，为一整数 T，表示完成任务所花费的最小时间。

输入样例

```
3
4              2              3
10 20          1 3            10          100
30 40          2 200          20          80
50 60                         30          50
70 80
```

输出样例

```
10
20
30
```

题目来源

Asia 2001,*Taejon*(*South Korea*)

【算法分析】

在经理给出的说明表格中,已经明确地描述了算法:

- (从房间 30 到 50)和(从房间 60 到 90)是允许的,因为没有占用公共的走廊;
- (从房间 20 到 40)和(从房间 31 到 80)是不允许的,因为要占用公共的走廊。

我们将每个房间之间的走廊作为一个统计单位,当所有的办公桌都搬运完成之后,看看这段走廊到底需要占用多少次?然后统计所有的走廊被占用的最大值 max,这个值就是要单独安排的搬运次数,乘以 10 就是总的搬运时间。

房间与走廊的映射关系,如表 5-12 所示。

表 5-12 房间与走廊的映射关系

房间编号	1,2	3,4	5,6	7,8	…	397,398	399,400
表示走廊的数组下标	0	1	2	3	…	198	199

该算法应该属于贪心算法,因为它尽可能使搬运办公桌同时进行,以便使单独安排的搬运次数最少。程序实现如算法 5.18 所示。

算法 5.18 ACM 公司搬运办公桌的贪心算法实现。

```
int i, j;
//每个房间之间一个走廊,一共有 200 个公共走廊
int move[200];
int N;                                  //搬运次数
//每次搬运的起点和终点
int from, to;
scanf("%d", &N);
memset(move, 0, sizeof(move));
for(i = 0; i < N; i++)
{
    scanf("%d%d", &from, &to);
    //将房间号映射为走廊号
    from = (from - 1)/2;
    to = (to - 1)/2;
    //确保 from < to
    if(from > to) swap(from, to);
    }
    //占用走廊情况
    for(j = from; j <= to; j++)
        move[j]++;
}
//所有的走廊被占用的最大值
int max = 0;
for(i = 0; i < 200; i++)
    if(move[i] > max) max = move[i];
printf("%d\n", max * 10);
```

算法实现源代码:zju1029.cpp。

例如样例1：(10,20),(30,40),(50,60)和(70,80),四次搬运都不占用公共的走廊,所以可以同时进行,一次搬运成功。

5.12　ZOJ1076-Gene Assembly

【问题描述】

随着大量的基因组DNA序列数据被获得,其对于了解基因越来越重要(基因组DNA的一部分是负责蛋白质合成的)。众所周知,在基因组序列中,由于存在垃圾的DNA中断基因的编码区,真核生物(相对于原核生物)的基因链更加复杂。也就是说,一个基因被分成几个编码片段(称为外显子)。虽然在蛋白质的合成过程中,外显子的顺序是固定的,但是外显子的数量和长度可以是任意的。

大多数基因识别算法分为两步：第一步,寻找可能的外显子；第二步,通过寻找一条拥有尽可能多的外显子的基因链,尽可能大地拼接一个基因。这条链必须遵循外显子出现在基因组序列中的顺序。外显子 i 在外显子 j 的前面的条件是 i 的末尾必须在 j 开头的前面。

本题的目标：给定一组可能的外显子,找出一条拥有尽可能多的外显子链,拼接成一个基因。

输入

给出几组输入实例。每个实例的开头是基因组序列中可能的外显子数 n($0<n<1000$)。接着的 n 行,每行是一对整数,表示外显子在基因组序列中的起始和结束位置。假设基因组序列最长为50 000。当一行是0时,表示输入结束。

输出

对于每个实例,找出尽可能多的外显子链,输出链中的外显子,并占一行。假如有多条链,但外显子数相同,那么可以输出其中任意一条。

输入样例

```
6                3                0
340 500          705 773
220 470          124 337
100 300          453 665
880 943
525 556
612 776
```

输出样例

```
3 1 5 6 4
2 3 1
```

题目来源

South America 2001

【算法分析】

虽然本题有很多人使用动态规划算法,但最简单的算法是贪心算法,相当于只有一个资源的活动安排问题。其算法的详细描述,请读者阅读本书5.1节。

1. 数据结构

外显子数为 n,外显子的结构体表示:

```
struct gene {
    int start, end, pos;                    //起始和结束位置,序号
}seq[1000];                                 //存放外显子的数组
```

2. 排序算法

按外显子的结束位置升序,排序因子:

```
bool cmp(gene a, gene b)
{
    if (a.end < b.end) return true;
    return false;
}
```

程序实现如算法 5.19 所示。

算法 5.19 基因拼接问题的贪心算法实现。

```
while(scanf("%d", &n) && n)
{
    //读取外显子
    for(int i = 0; i < n; i++)
    {
        seq[i].pos = i + 1;
        scanf("%d%d", &seq[i].start, &seq[i].end);
    }
    //按外显子的结束位置升序排序
    sort(seq, seq+n, cmp);
    //贪心算法的实现,第一个总是有效的
    int preEnd = seq[0].end;
    printf("%d", seq[0].pos);
    for(int i = 0; i < n; i++)
        //第 i 个显子的起始位置必须大于前一个选中的显子的结束位置
        if(seq[i].start > preEnd)
        {
            preEnd = seq[i].end;
            printf("%d", seq[i].pos);
        }
    printf("\n");
}
```

算法实现源代码:zju1076-sort.cpp。对于样例数据 1,计算结果如表 5-13 所示。

表 5-13 样例数据 1 的计算结果

原始顺序	1	2	3	4	5	6
	340 500	220 470	100 300	880 943	525 556	612 776
拼接之后的顺序	3	1	5	6	4	
	100 300	340 500	525 556	612 776	880 943	

5.13　ZOJ1161-Gone Fishing

【问题描述】

约翰想要去钓鱼。他有 h(1≤h≤16)个小时的时间,在该地区有 n(2≤n≤25)个湖,这些湖刚好分布在一条路线上,该路线是单向的。约翰从湖 1 出发,他可以在任一个湖结束钓鱼。但他只能从一个湖到达另一个与之相邻的湖,而且不必每个湖都停留。假设湖 i(i=1,…,n−1),以 5 分钟为单位,从湖 i 到湖 i+1 需要的时间用 t_i(0<t_i≤192)表示。例如 t_3=4,是指从湖 3 到湖 4 需要花 20 分钟时间。

为了制订钓鱼计划,约翰收集了一些湖的信息。已知在最初 5 分钟,湖 i 预计钓到鱼的数量为 f_i(f_i≥0)。以后每隔 5 分钟,预计钓到鱼的数量将以常数 d_i(d_i≥0)递减。如果某个时段预计钓到鱼的数量小于或等于 d_i,那么在下一时段将钓不到鱼。为简单起见,假设没有其他的钓鱼者影响约翰的钓鱼数量。

编写一个程序,帮助约翰制订钓鱼旅行的计划,以便他尽可能多地钓到鱼。在每个湖上花费的时间必须是 5 的倍数。

本题可以包含多组测试例。多组测试例的第一行是一个整数 N,然后是一个空行,接着是 N 个输入数据块。每个数据块的格式都在问题描述中给出。每个数据块之间都有一个空行。

输出格式包括 N 个输出数据块,每个输出数据块之间都有一个空行。

输入

输入有多组测试例。对每组测试例,第一行都是 n,接下来一行是 h。下面一行是 n 个整数 f_i(1≤i≤n),然后是一行 n 个整数 d_i(1≤i≤n),最后一行是 n−1 个整数 t_i(1≤i≤n−1)。当 n=0 时,输入结束。

输出

对每个测试例,输出在每个湖上花费的时间,用逗号分隔,这是约翰要实现钓到最多的鱼的计划(必须使整个计划在同一行输出,即使超过 80 个字符)。接下来一行是钓到的鱼的数量。如果存在很多方案,尽可能选择在湖 1 钓鱼所耗费的时间,即使有些时段没有钓到鱼;如果还是无法区分,那就尽可能选择在湖 2 钓鱼所耗费的时间,以此类推。测试例之间有一个空行。

输入样例

```
1              4              4
               4              4
2              10 15 20 17    10 15 50 30
1              0 3 4 3        0 3 4 3
10 1           1 2 3          1 2 3
2 5                           0
2
```

输出样例

```
45, 5
```

```
Number of fish expected: 31
240, 0, 0, 0
Number of fish expected: 480
115, 10, 50, 35
Number of fish expected: 724
```

题目来源

East Central North America 1999;*Pacific Northwest 1999*

【算法分析】

本题在刘汝佳的《算法艺术与信息学竞赛》第一章“算法与数据结构”中有描述。在网站上也有很多资料,基本上都是使用贪心算法。

1. 数据结构

每个湖预计钓到鱼的数量,定义为数组:

```
#define NUM 30
int f[NUM];
```

每个湖预计钓到鱼的数量的递减值,定义为数组:

```
int d[NUM];
```

相邻湖之间的旅行时间,定义为数组:

```
int t[NUM];
```

钓鱼计划,定义为数组:

```
int plan[NUM];
```

湖的个数 n,用于钓鱼的时间 h,尽可能多地钓鱼数量 best。

2. 搜索,在任意一个湖结束钓鱼时的最优钓鱼计划

首先把用于钓鱼的时间 h,由小时转换为以 5 分钟为单位的时间:

$$h = h \times 60/5$$

这样把钓 5 分钟鱼的时间称为钓一次鱼。由于约翰从湖 1 出发,可以在任一个湖结束钓鱼,要得到最优解,就需要进行搜索。

设花费在路程上的时间为:

```
int  time = 0;
```

假设约翰在第 m 个湖结束钓鱼,因为路程是单向的,所以路程上的时间:

$$\text{time} = \sum_{i=1}^{m} t[i],\text{并且满足 } h - \text{time} > 0。$$

于是展开搜索:

```
for (i = 1; i <= n && h-time; ++i)
{
    greedy(i, h - time);
    time += t[i];
}
```

其中函数 greedy()为：

```
void greedy(int pos, int time);
```

约翰在第 pos 个湖结束钓鱼，用于钓鱼的时间是 time(不含路程)，即钓鱼 time 次。该函数采用贪心算法构造约翰的钓鱼计划。可以认为约翰能从一个湖“瞬间转移”到另一个湖，即在任意一个时刻都可以从湖 1 到湖 pos 中任选一个钓一次鱼。

3. 采用贪心策略，每次选择鱼最多的湖钓一次鱼

对于每个湖来说，由于在任何时候鱼的数目只和约翰在该湖里钓鱼的次数有关，和钓鱼的总次数无关，所以这个策略是最优的。一共可以钓鱼 time 次，每次在 n 个湖中选择鱼最多的一个湖钓鱼，程序实现如算法 5.20 所示。

算法 5.20　选择鱼最多的湖钓鱼的贪心算法实现。

```
//从湖 1 起到湖 pos 止，花费时间 time(不含路程)的钓鱼计划
void greedy(int pos, int time)
{
    //时间已经用完
    if (time <= 0) return;
    int i, j;
    int fish[MAXN];
    int p[MAXN];
    int t = 0;
    for (i = 0; i < pos; ++i)
        fish[i] = f[i];
    memset(p, 0, sizeof(p));
    //在时间 time 内，选择鱼最多的湖钓鱼；如果鱼都没有了，就把时间放在湖 1 上
    for (i = 0; i < time; ++i)
    {
        //鱼最多的湖中，鱼的数量
        int max = 0;
        //鱼最多的湖的编号
        int id = -1;
        //查找鱼最多的湖中，鱼的数量和湖的编号
        for (j = 0; j < pos; ++j)
            if (fish[j] > max){
                max = fish[j];
                id = j;
            }
        //找到了，进行钓鱼处理
        if (id != -1)
        {
            ++p[id];
            fish[id] -= d[id];
            t += max;
        }
        //没有找到(从湖 1 起到湖 pos 全部钓完了)，就把时间放在湖 1 上
        else ++p[0];
    }
    //处理最优方案
```

```
        if (t > best)
        {
            best = t;                          //最优值
            memset(plan, 0, sizeof(plan));
            for (i = 0; i < pos; ++i)          //最优解
                plan[i] = p[i];
        }
    }
```

输出钓鱼计划时,再把 5 乘回去,就变成实际的钓鱼时间(分钟):

```
for (i = 0; i < n - 1; ++i)
    printf("%d, ", plan[i] * 5);
printf("%d\n", plan[n - 1] * 5);
printf("Number of fish expected: %d\n", best);
```

算法实现源代码:zju1161.c。

5.14 ZOJ1171-Sorting the Photos

【问题描述】

假如你有 $n(1 \leqslant n \leqslant 10^5)$张照片。有些照片正面朝上,而有些是反面朝上。你的任务是排序,使所有的照片朝向一致。需要做的操作是,从一堆照片中把需要翻转的照片挑出来放成一堆,然后把这堆照片翻转过来,放回到原来的那堆照片上。

编写程序:采用上述的操作方法完成排序,计算最少的操作次数。

本题包含多组测试例。多组测试例的第一行都是一个整数 N,然后是一个空行,接着是 N 个输入数据块。每个数据块的格式都在问题描述中给出。每个数据块之间都有一个空行。

输出格式包括 N 个输出数据块,每个输出数据块之间都有一个空行。

输入

输入的第一行只有一个数字。然后是一组字符,这些字符在不同的行,用/或者空格隔开,字符 D 表示正面向下,U 表示正面朝上。

输出

一个整数,即给照片排序所需最少的翻转操作次数。

输入样例

```
1
5
UU D
UU
```

输出样例

```
2
```

【算法分析】

由于照片多达 10^5 张,回溯算法肯定是太慢了。必须找到一种翻转照片的方法,方法本

身就是最优的。这里面有一个技巧，从一堆照片中取照片是不计操作次数的，只有翻转时才计算操作次数。因此我们就逐张照片比较，如遇到照片不同时就翻转一次，再用当前照片的朝向接着比较。例如，样例数据 UUDUU，操作步骤如下：

(1) 先将 UU 放在一边，遇到 D，将 UU 翻转，放到原来的堆上面，成为 DDDUU；

(2) 将 DDD 放在一边，遇到 U，将 DDD 翻转，放到原来的堆上面，成为 UUUUU。

朝向就变成一样的了，操作次数是 2。

编程时，还可以简化，从第一张照片开始与下一张照片比较，如果遇到照片朝向不同，则翻转次数加 1，再用不同的那张照片朝向接着与下一张比较，直到结束。

程序实现如算法 5.21 所示。

算法 5.21 翻转照片的贪心算法实现。

```
int i;
int n;                                  //照片的张数
char p[100000];                         //存放照片
int N;                                  //数据块的数量
scanf("%d", &N);
while(N--)
{
    scanf("%d", &n);
    for(i = 0; i < n; )
    {
        scanf("%c", p + i);
        //跳过分隔符
        if(p[i] == 'U' || p[i] == 'D') i++;
    }
    //最少的翻转操作次数
    int ans = 0;
    //比较位置
    int pos = 0;
    //逐张照片比较,如遇到不同则 ans + 1,再用当前照片的朝向接着比较
    for(i = 1; i < n; i++)
        if(p[i] != p[pos])
        {
            pos = i;
            ans++;
        }
    printf("%d\n", ans);
    if(N) putchar('\n');
}
```

算法实现源代码：zju1171.c。

为什么这种方法能够得到最优解呢？实际上这种方法采用了贪心策略，不仅每一步是最优的，而且整体也是最优的，即具有最优子结构性质。

基于这种算法，代码还可以进一步简化：取消数组，在读取数据的时候就进行比较。

算法实现源代码：zju1171-noArray.c。

5.15 ZOJ2109-FatMouse' Trade

FatMouse准备了M磅的猫粮,打算与看守仓库的猫交换食品,仓库里存放着它喜爱的食物JavaBean。

仓库有n个库房,库房i存放$J[i]$磅JavaBean,需要$F[i]$磅猫粮予以交换。FatMouse不需要交换库房里所有的JavaBean,可以按比例交换。如果它支付$F[i]\times a\%$磅的猫粮,就可以换取$J[i]\times a\%$磅的JavaBean,其中a是实数。

现在明确编程任务:FatMouse最多能换取多少JavaBean。

输入

输入包含多组测试例。对每个测试例,第一行是两个非负整数M和N。接下来N行,每行两个非负整数$J[i]$和$F[i]$。最后一个测试例是两个-1,不需要处理。所有的整数都不超过1000。

输出

对每个测试例,都输出一行:是一个实数,精确到小数点后3位,表示FatMouse最多能换取的JavaBean数量。

输入样例

```
5 3          20   3
7 2          25   18
4 3          24   15
5 2          15   10
             -1  -1
```

输出样例

```
13.333
31.500
```

【算法分析】

FatMouse用M磅猫粮与猫换取它喜爱的食物JavaBean。JavaBean存储在一个仓库的N个库房中,而且每个库房的JavaBean所需要的猫粮代价是不一样的。以样例数据1为例,其代表的状态如表5-14所示。

表5-14 样例数据1所表示的状态

库房号	1	2	3
存储的JavaBean	7	4	5
所需要的猫粮代价	2	3	2

当FatMouse剩余的猫粮不够换取整个库房的JavaBean时,可以按比例换取。题目要求猫能够换取尽可能多的JavaBean。

1. 为了使猫能够换取尽可能多的猫粮,就要计算出每个库房的性价比

定义数据结构:

```
#define N 1005
struct trade
{
    int java;                    //库房中的 JavaBean
    int food;                    //换取该库房 JavaBean 所需要的猫粮
    double ratio;                //用猫粮换取 JavaBean 时的性价比
}a[N];
```

对样例数据 1,猫粮换取 JavaBean 的性价比如表 5-15 所示。

表 5-15　样例数据 1 猫粮换取 JavaBean 的性价比

存储的 JavaBean	7	4	5
所需要的猫粮代价	2	3	2
性价比	3.5	1.3333	2.5

2. 按性价比排序

应该从性价比最好的库房换取,这就需要按性价比进行降序排序。排序因子:

```
bool cmp(const info& a, const info& b)
{
    if (a.ratio > b.ratio) return true;
    return false;
}
```

使用 C++的标准模板库函数排序:

```
sort(a, a + n, cmp);
```

3. 应用贪心算法,用猫粮换取 JavaBean

当猫粮足够多时,按库房的性价比由高到低换取;

当猫粮的量不足以换取整个库房的 JavaBean 时,则按比例换取。

程序实现如算法 5.22 所示。

算法 5.22　FatMouse 用猫粮换取 JavaBean 的贪心算法实现。

```
double beans = 0;                //换取到的 JavaBean
int catfood = 0;                 //花费的猫粮
//采用贪心算法换取猫粮
//m - catfood 是当前 FatMouse 还剩下的猫粮
for (i = 0; i < n && m - catfood >= a[i].food; i++)
{
    beans += a[i].java;
    catfood += a[i].food;
}
//剩下的猫粮不够换取一个库房的 JavaBean,则按比例换取 JavaBean
if (i < n)
    beans += (m - catfood) * a[i].ratio;
printf("%.3lf\n", beans);
```

算法实现源代码:zju2109.cpp。

上机练习题

浙江大学在线题库(由于 ZOJ 题目很多,这里只列出了部分题目,仅供参考):

1117-Entropy	2425-Inversion
1161-Gone Fishing	2488-Rotten Ropes
1200-Mining	2510-Concentric Rings
1239-Hanoi Tower Troubles Again!	2521-LED Display
1307-Packets	2536-Best Balance
1360-Radar Installation	2541-Goods Transportation
1375-Pass-Muraille	2702-Unrhymable Rhymes
1409-Communication System	2710-Two Pipelines
1543-Stripies	2833-Friendship
2049-Advertisement	2921-Stock
2091-Mean of Subsequence	3116-Loan Scheduling
2229-Ride to School	3197-Google Book
2256-Mincost	3301-Make Pair
2315-New Year Bonus Grant	3410-Layton's Escape
2343-Robbers	3424-Rescue
2354-Elevator Stopping Plan	3433-Gu Jian Qi Tan
2378-Evil Straw Warts Live	3508-The War
2397-Tian Ji-The Horse Racing	

北京大学在线题库(由于 POJ 题目很多,这里只列出了部分题目,仅供参考):

1017-Packets	2240-Arbitrage
1042-Gone Fishing	2253-Frogger
1062-昂贵的聘礼	2287-Tian Ji-The Horse Racing
1065-Wooden Sticks	2313-Sequence
1125-Stockbroker Grapevine	2325-Persistent Numbers
1230-Pass-Muraille	2370-Democracy in danger
1258-Agri-Net	2393-Yogurt factory
1323-Game Prediction	2395-Out of Hay
1328-Radar Installation	2431-Expedition
1456-Supermarket	2485-Highways
1679-The Unique MST	2709-Painter
1716-Integer Intervals	3026-Borg Maze
1755-Triathlon	3228-Gold Transportation
1784-Huffman's Greed	3253-Fence Repair
1789-Truck History	3259-Wormholes
1797-Heavy Transportation	3262-Protecting the Flowers
1860-Currency Exchange	3522-Slim Span
1861-Network	3544-Journey with Pigs
1862-Stripies	3614-Sunscreen
1922-Ride to School	3617-Best Cow Line
2054-Color a Tree	3625-Building Roads
2209-The King	3723-Conscription