随机化算法

随机化算法是这样的一种算法，即在算法中使用了随机函数（指 random 函数，其返回值为[0,n-1]中的某个整数，且返回每个整数都是等概率的），且随机函数的返回值直接或间接地影响了算法的流程或执行结果。

根据这个定义，可以看出，并不是所有使用了随机函数的算法都可以称为随机算法。例如，某个算法包含 i:=random(n)，但变量 i 除了在这里被赋予一个随机值之外，在其他地方从未出现过。显然，如果这个算法没有在其他地方用过随机函数，上面这条语句就无法影响执行的流程或结果，这个算法就不能称为随机化算法。

另一方面，若一个算法是随机化算法，则它执行的流程或结果就会受其使用的随机函数的影响。根据这一影响的性质和程度，可以分为下列三种情况：

（1）随机不影响执行结果。这时，随机必然影响了执行的流程，其效应多表现为算法的时间效率的波动；

（2）随机影响执行结果的**正确性**。在这种情况中，原问题要求我们求出某个可行解，或者，原问题为判定性问题，随机效应表现为程序执行后得到正确解的概率；

（3）随机影响执行结果的**优劣**。这时，随机效应表现为实际执行结果与理论上的最优解或期望结果的差异。

由于随机化算法的执行情况受到随机函数这种不确定因素的支配，因此，即使同一个算法在多次执行中用同样的输入，其执行情况也会不同，至少略有差异。差异可能表现为出解速度快慢、解的正确与否、解的优劣等。例如，一个随机化算法可能在两次执行中，前一次得到的解较优，后一次得到的解较劣。

现在的问题是：在绝大数情况下，尤其是在各种竞赛中，对于同样的输入，只允许程序运行一次，根据运行结果判定算法的好坏。如此一来，我们就会把出劣解的一次运行归咎为运气不佳，反之亦然。然而，比赛比的是谁的算法更有效，而不是谁的运气更好。

既然我们使用了随机函数，我们就无法摆脱运气的影响，所以，我们的目标是尽量将运气的影响降到最低。也就是说，我们必须使算法的执行情况较为稳定。因此，在接下来的对算法的分析中，我们将从以下四个方面分析随机化算法的性能：

（1）时间效率；

（2）解的正确性；

（3）解的优劣程度（解与最优解的接近程度）；

（4）稳定性，即算法对同样的输入数据执行后输出结果的变化，变化越小则越稳定。非随机化算法的稳定性为 100%，随机化算法的稳定性属于区间[0%,100%]。稳定性可以是算法的平均时间复杂度，也可以是执行算法得到正确解的概率，还可以是实际解达到某一优劣程度的概率。稳定性是评判随机化算法好坏的一个重要指标。

下面，我们按照随机化函数对程序结果和流程影响的性质和程度来分别讨论。

一、执行结果确定的随机化算法

我们以常用的“快速排序算法”为例为进行说明。

亦即，每次划分时，从 A[lo..hi]中随机地选择一个数作为 x 对 A[lo..hi]进行划

分，这只需对原算法稍作修改即可。

算法改动以后，算法的最坏、平均、最佳时间复杂度依然分别为 $O(n^2)$、$O(n\log_2 n)$、$O(n\log_2 n)$，只不过最坏、最佳情况不再由输入所决定，而是由随机函数所决定。

现在，我们来分析随机化快速排序算法的稳定性。

在“各种排列的出现是等概率”的假设下（该假设不一定成立），快速排序遇到最坏情况的可能性为 1/n!。假设 random(n)产生 n 个数的概率都相同（该假设几乎一定成立），则随机化快速排序遇到最坏情况的可能性也为 1/n!。

如果 n 足够大，我们就有多于 99%的可能性会“交好运”。也就是说，随机化的快速排序算法有很高的稳定性。下表是一般快排和随机化后的快排的性能对比表。

表 3-2：暂略，见 P51

从以上分析可以看出，执行结果确定的随机化算法原理是：

用随机函数全部或部分地抵消最坏输入的作用，使算法的时间效率不完全依赖于输入的好坏，通过对输入的适当控制，使得执行结果相对稳定，这是设计这一类随机化算法的常用方法。

（例如，在随机化快排算法中，随机的作用就相当于将数打乱；又如，在建立二叉排序树时，可以先随机地将待插入的关键字顺序打乱，再依次插入树中，以获得较为平衡的二叉排序树，提高以后查找关键字的效率。

二、执行结果可能偏离正确解的随机化算法

某些随机化算法有时甚至会输出错误的结果，但它在某些场合依然是很有效的。下面我们以判定素数的算法为例，来试验一下这种随机化算法。

一般的素数判定算法是这样的：

对于较小的 n，可以用“筛法”来判定 n 是否为素数。对于稍大一点的 n，我们只能用 2~trunc(sqrt(n))来试除 n。若能除尽，则说明 n 是合数，否则为素数。

实现时，我们可以先判断 n 是否为偶数，然后用 3、5、7、9、…、来逐个试除 n，以加快程序的运行速度。

尽管如此，当遇到较大的素数时，这一算法还是显得效率低下，其最坏情况时间复杂度为 O(sqrt(n))，而且，这种方法要借助于大数组。如果 n 很大，则空间开销也会很大。

下面换一种素数判定的方法：

由 Fermat 小定理可知：

若 n 是素数，而 a 不能整除 n，则 a^{n-1} mod n=1 必然成立。

我们可以将这个定理**改写成**：若 n 是素数，对于 a=1,2,…,n-1，有 a^{n-1} mod n=1。所以，若存在整数 a∈[1,n-1]，使得 a^{n-1} mod n<>1，则 n 必为合数。

这样，我们**随机地选取若干个（假设为 s 个）a 值**，并检查 a^{n-1} mod n=1 是否成立。若发现某个 a 使得该式不成立，则算法判定“n 是合数”；若选取的 a 都使该式成立，则算法没有找到反例，即认为“n 是素数”——**这就是随机化的素数判定算法**。

这个算法只产生一种错误，即选取的 s 个 a 值均满足 a^{n-1} mod n=1，而实际上 n 是合数，这时，算法会误认为 n 是合数的证据不足，判其为素数。

那么，如何求 a^{n-1} mod n 呢？方法如下：

设 $n-1=b_k 2^k+b_{k-1}2^{k-1}+\cdots+b_0 2^0$，$b_k b_{k-1}\cdots b_0$ 为 n-1 的二进制表示，则

k=trunc(log$_2$(n-1))。这样，$a^{n-1} \bmod n = a^{(\cdots(((b_k)\square 2+b_{k-1})\square 2+b_{k-1})\square 2+b_{k-2})\square 2+\cdots+b_1)\square 2+b_0} \bmod n$。

我们不妨用<a>表示 a mod n，则上式可以改写为：

$$<<<\cdots<<<<<< a^{b_k} >^2>< a^{b_{k-1}} >>^2 \cdots >^2>< a^{b_0} >>$$

亦即，用 trunc(log$_2$(n-1))次循环，就可以求出 $a^{n-1} \bmod n$ 的值。 {**快速幂方法**}

随机化算法判定素数算法的稳定性取决于：对于合数 n，在[1..n-1]内满足 $a^{n-1} \bmod n=1$ 的 a 值的个数，记该个数为 H(n)，则该算法判定 n 的稳定性为 $(1-H(n)/n)^s$。事实上，大多数 n 的 H(n)/n 都可以达到 98%，几乎所有 n 的 H(n)/n 都不小于 50%。在 10000 以内，H(n)/n 小于 50%的不超过 10 个。所以，我们有理由相信，只要 s 取适当的值，就能使算法有很高的稳定性。

实践证明，当 s=50 时，该算法对所有 n 的稳定性至少为 $1-2^{-50}>99\%$。即使取较小的 s(如 s=5)，算法也未必会得到错误的结果。

随机化的素数判定算法在最坏情况下的时间复杂度为 $O(s\log_2 n)$，其中 s 为一个常数（随机抽取的 a 值的个数）。该算法的时间效率比一般的素数判定算法要高许多，这可以从它们各自的时间复杂度中看出。

在实际运用中，取 s=50，n=761838257287（是素数），将两种算法对应的程序各运行 100 次，后者需要的时间是前者的 5 倍左右。此外，素数判定通常都是作为一个子程序被调用的，其实际使用次数可能还不止 100 次，因此，可以看出，这两种算法的差距还是很明显的。

执行结果可能偏离正确解的随机化算法基于这样的原理：一个近似正确的算法的近似程度受某个参数的影响。当该参数取某个值时，算法能得到正确解。因此，只要将算法执行多次，每次参数取指定范围内的随机值，算法就可能得到正确结果。

通常我们可以这样来设计此类随机化算法：**先选一个近似算法**（这里用了以 Fermat 定理为基础的判定算法），然后**在算法中加入随机化控制（这里用了 a）**，最后**加外循环控制，多次执行近似算法（这里用了 s）**，如此构成随机化算法。另外，通过重复执行近似算法，也提高了算法的稳定性，使之达到了期望的水平。

三、执行结果有优劣变化的随机化算法

这种随机化算法大多针对要求较优解的问题，例如 NOI 1998 中的“并行计算”问题。

例 1：并行计算

【问题描述】

运算器（ALU）是计算机中的重要部件，它的功能是进行数学运算。下图是运算器的工作简图。

运算器的一次运算操作过程为：运算器在控制器的控制下，从指定的存储器（MEMORY）存储单元中读出待运算的两个源操作数 A 和 B，经过一定时间的计算后，得到运算结果 C，并将它写入指定的存储器存储单元中。

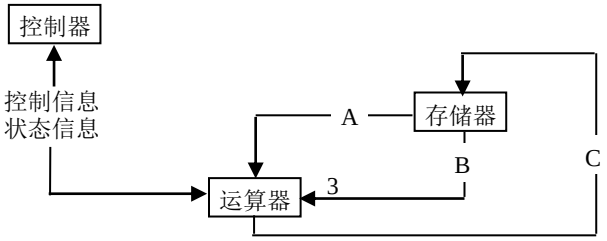

图 1

运算器能完成的运算种类及所需时间如下表所示：

运算种类	运算操作	所需运算时间
1	C = A +B	Tadd
2	C = A - B	Tsub
3	C = A * B	Tmul
4	C = A / B	Tdiv

在计算复杂的四则混合运算表达式时，运算器就变成了高速计算的“瓶颈”。为了得到更快的运算速度，计算机科学家们设计了一种有两个运算器的并行计算机，它的结构结构简图如下图所示：

图暂略

由于两个运算器可以同时进行运算，因此大大提高了整机的运算速度。

该并行计算机有以下两种控制指令：

运算指令：

OP Time Alu_no Operate_no Address1 Address2 Address3

其中 OP 为运算指令标识符，其后有六个参数：

Time 表示执行该指令的开始时间；

Alu_no 表示承担该次运算的运算器的编号（1 号或 2 号）；

Operate_no 表示该次运算的运算种类（Operate_no∈{1,2,3,4}；

Address1、Address2 表示待运算的两个源操作数的存储单元地址；

Address3 表示该次运算结束后存放运算结果的存储单元地址。

结束指令：

END Time Address

其中 END 表示结束指令标识符，其后有两个参数：

Time 表示整个控制程序的结束时间；

Address 表示存放最终结果的存储单元地址。

每个运算器同一时刻可以执行一条运算指令，结束指令表示控制程序结束。

现在需要你编写一个程序，对给定的待计算的表达式，自动生成一段控制程序，使上图所示的并行计算机能够正确计算该表达式的值，并使总的运算时间尽量小。

输入：

输入文件的第一行为四个不超过 1000 的正整数，依次为 Tadd、Tsub、Tmul、Tdiv。

输入文件的第二行为待计算的四则混合运算表达式（长度不超过 255 个字符），表达式中的变量用大写英文字母表示，各变量的初始值依照变量的字母顺序依次存放在地址为 1、2、…的存储单元中。

输入文件中同一行相邻两项之间用一个或多个空格隔开。

输出：

输出文件为完成该表达式计算的最优控制指令段。指令根据其开始时间先后依次输出（对于开始执行时间相同的两条指令，输出先后次序不限），每条指令占一行。

输出文件中同一行相邻两项之间用一个空格隔开。

【问题约定】

控制程序初始执行时间从 0 时刻开始；问题中所涉及的各种时间量的单位相同；存储器的存储单元最多有 1K 个；由于数据读写时间同运算时间相比较小，可忽略不计；如果两个运算器同时对同一个存储单元进行操作，则运算器 1 先操作，运算器 2 后操作。

【评分标准】

程序得分将取决于其所能找到的最优解与标准最优解相比较的优劣程度。

【样例输入】

```
2 2 4 12
C+(A+B)*C-E/F+F
```

【样例输出】

```
OP 0 1 1 1 2 6
OP 0 2 4 4 5 8
OP 2 1 1 3 5 7
OP 4 1 3 6 3 10
OP 8 1 1 10 7 11
OP 12 1 2 11 8 12
END 14 12
```

【问题分析】

如果用搜索算法来解决这个问题，则每次扩展搜索树必须确定当前的操作数、运算符和运算器，搜索量大得惊人。

由于问题并未要求我们求最优解，因此，我们可以用贪心法求较优解。

至于采用哪种贪心策略，可以有多种选择，如每次先选取耗时长的运算符，也可以每次选取最早空闲的运算器等。

我们目前尚未找到一种普遍适用的贪心策略，且找到这种贪心策略的可能性不大。另一方面，现在的每一种策略都只可能在一定程度上对某些输入取得较好的结果，命题者完全可以对各种策略分别设计出不符合其贪心方式的输入，使它输出不理想的结果。

囿于上述限制，用纯粹的贪心算法无法有效地解决问题。

一种解决的方法是：由于贪心算法有很高的时间效率，我们可以在同一个程序中，将各种贪心策略全部都试一次，但这样无疑极大地增加了编程复杂度。

下面的随机化贪心算法给出了一个较好的解决方法。

随机化贪心算法的基本思想是：

设置贪心程度 rate%(rate∈{0..100})，选择一种较好的贪心策略为基础，每次求局部最优解的过程改为每次求在该贪心标准下贪心程度不小于 rate%的某个局部较优解。也就是说，存在输入，使算法在不是每次都选局部最优解的情况下，得到的解比每次都选局部最优解所得到的解更优。上述随机化贪心算法在 rate=100 时得到的结果就是原来未加修改的贪心算法的结果，所以，上述随机化贪心算法至少不比一般的贪心算法差。

上述思想较简单，实现起来也不困难，而且贪心算法都有着很高的时间效率，多次贪心消耗的时间也不会很长。但是，由于问题对解的约束不多，解的种类和个数就可能很多，因此，要从理论上分析随机化贪心算法的性能较为困难。不过，我们可以从实际效果验证出，随机化贪心算法对大部分输入都能得到与标准答案同样优的结果，甚至对某些输入，能得到比标准答案更优的结果。

同时，该算法也有着较高的稳定性（注意，这里的稳定性是指获得某一范围内的解的概率，如得到与标准答案同样优的结果的概率，又如得到比标准答案稍好一点的结果的概率）。

执行结果有优劣变化的随机化算法的原理为：一个近似算法的近似程度受某个参数的影响，当参数变化时，算法的执行结果会有优劣变化。只要将该算法执行多次，每次参数取指定范围内的随机值，并在执行后及时更新当前最优解，当前解就能不断逼近理论最优解。

通过对以上三种随机化算法的分析，我们总结如下：随机化算法的基本原理是：当某个决策中有多个选择、但又无法确定哪一个是好的选择，或确定好的选择需要付出较大的代价时，如果大多数选择是好的，那么随机选一个往往是一种有效的策略。**通常，当一个算法需要做出多个决策，或需要多次执行一个算法时，这一点表现得尤为明显。**

这个原理使得随机化算法有一个共同的性质：没有一个特别的输入会使算法执行出现最坏情况。最坏情况可以表现在执行流程中（主要是时间效率），也可以表现在执行结果中。

根据这个原理，我们设计随机化算法时，通常以某个算法为母版，加入随机因素，使得算法在难以做出决策时随机地选择。在设计执行结果受随机影响的算法时，我们还可以多次执行算法，使算法不断逼近正确解或最优解。

以上只是设计随机化算法的基本方式，实际应用中，随机化算法的设计没有公式可套，必须具体问题具体分析，深入研究，设计出好的随机化算法。同时，在设计随机化算法时，还需要特别注意一个问题，就是随机化算法的适用范围和有效性。（思考、讨论、实例）

一个问题对算法的所有要求除了问题本身的描述之外，还有诸如空间限制、时间限制、稳定性限制等要求。如果一个问题对算法不强求 100%地稳定，即对于同样的输入，不必每次运行的情况都相同（不然这种不稳定性不能太大），同时作为补偿，又要求算法在其他某些方面有较好的性能（如出解迅速），而这些性能是一般非随机化算法无法达到的，那么此时随机化算法可能就是能有效解决问题的候选算法之一；否则，随机化算法便不适用。当一个随机化算法适用于解决某个问题，且该算法有着较高的稳定性，同时在其他某些方面有突出表现（如速度快、代码短等），能比一般非随机化算法做得更为出色时，那么，这个随机化算法就是一个行之有效的算法。

例 2：火力网

【问题描述】

在一个 n*n 的阵地中，有若干炮火不可摧毁的石墙，现在要在这个阵地中的空地上布置一些碉堡。假设碉堡只能向上下左右四个方向开火，由于建筑碉堡的材料挡不住炮火，所以任意一个碉堡都不能落在其他碉堡的火力范围内，请问至多可以建造几座碉堡？

【输入】

输入文件名为 fire.in，其中第一行是一个整数 n（n<=10）。

接下来有 n 行，每行为一个由 n 个字符构成的字符串，表示阵地的布局，包括空地（'.'）和石墙（'X'）。

【输入】

输出文件名为 fire.out，其中包含一个整数，即最多可以建造的碉堡数。

【样例输入】

```
4
.X..
....
XX..
....
```

【样例输出】

```
5
```

【问题分析】

本题由于数据较小，可以考虑用随机化方法多次模拟求最优解。

具体来说，对于每一遍求解，就是不断地随机选取可以建碉堡的空格来建碉堡，直到没有可建碉堡的空地为止，此时将碉堡数与当前最优解进行比较，若优于当前最优解，则更新当前最优解。

为了方便处理，程序中将平面上的格点逐行排列成一维的格点，并对它们进行编号（从 1 号开始），用数组 state 记录每个方格的状态，值为-1 表示石墙，值为 try 表示可在此格建碉堡，值为 try+1 表示此点在某碉堡的火力范围内。

link 数组对每个格点记录它能火力控制到的所有其他格点的编号。随机的方法是每次任取一格，若此格可以建碉堡，则选定此格建碉堡，否则，从此格向前后扫描，寻找附近的可建碉堡的格点来建碉堡。若无格点可供建碉堡，则得到一组解。反复这一过程，直至时限到期。

【参考程序】

```
const
  maxn = 10;
  limit = 4;
type
  arrtype = array[1..maxn*maxn,0..2*maxn] of longint;
var
  c, count, i, n, time, max, try, x, y: longint;
  found: boolean;
  row: array[1..maxn] of string;
  link: arrtype;
  state: array[0..maxn*maxn+1] of longint;

procedure init;
var i, j, k, no, x, y: longint;
begin
  state[0] := -1;   state[n*n+1] := -1;
  for no := 1 to n*n do
  begin
    i := (no - 1) div n + 1;
    j := (no - 1) mod n + 1;
```

```
      link[no, 0] := 0;
      state[no] := -1;
      if row[i,j] = '.' Then
      begin
        state[no] := 0;
        k := 0;
        y := j;
        while (y<n) and (row[i,y+1]='.') do
        begin
          link[no,0]:=link[no,0]+1;
          y := y + 1;
          k := k + 1;
          link[no,k] := (i-1)*n + y;
        end;
        y := j;
        while (y>1) and (row[i,y-1]='.') do
        begin
          link[no,0]:=link[no,0]+1;
          y := y - 1;
          k := k + 1;
          link[no,k] := (i-1)*n + y;
        end;
        x := i;
        while (x<n) and (row[x+1,j]='.') do
        begin
          link[no,0]:=link[no,0]+1;
          x := x + 1;
          k := k + 1;
          link[no,k] := (x-1)*n + j;
        end;
        x := i;
        while (x>1) and (row[x-1,j]='.') do
        begin
          link[no,0]:=link[no,0]+1;
          x := x - 1;
          k := k + 1;
          link[no,k] := (x-1)*n + j;
        end;
      end;
    end;
  end;

  begin  { main }
    assign(input, 'fire.in');    reset(input);
```

```
  assign(output, 'fire.out');   rewrite(output);
  time := mem1[$40:$6c];
  readln(n);
  for i := 1 to n do   readln(row[i]);

  init;

  max := 0;
  try := 0;

  repeat
    count := 0;
    repeat
      found := true;
      x := random(n*n) + 1;   {要加 randomize}
      c := x;
      while (x<n*n) and (state[x]<>try) do x := x + 1;
      if state[x] <> try then
      begin
        x := c;
        while (x>0) and (state[x]<>try) do x := x - 1;
      end;
      if state[x] = try then
         begin
           count := count + 1;
           for i :=1 to link[x,0] do
              if state[link[x,i]] = try then
                 inc(state[link[x,i]]);
           inc(state[x]);
         end
      else
         found := false;
    until not(found);
    if count > max then max := count;
    try := try + 1;
  until meml[$40:$6c] - time > 18.2 * limit;

  writeln( max );
end.
```

例 3：棉花糖超好吃(delicious.pas/cc/cpp)

【问题描述】

大家在 NOIP 前打赌，谁考了最低分就要请大家吃东西，于是吐血毫无悬念地中彩了。

msh：“棉花糖超好吃。”
虚伪：“我不吃辣的。”
will：“我要喝奶茶。”
吐血：“只要国产的就行。”
wx：“只要不是黑的就好。”
lwc：“只要你掏钱我都 ok！”
众：……

这附近共有 m 种食物出售，并且已从 1 到 m 编号。大家转了一圈后发现，因为消费群体特殊的缘故，这里和苏州中学周围一样，所有食物都是成对出售的。这不仅再次刺激了吐血，也难倒了大家。每个人都有自己喜欢的若干种食物，因为晚上还有聚餐，所以每人现在只愿意吃一份自己喜欢的食物，多余的只好丢掉，但是这样吐血会很伤心。

现在请你给出一种消费方案，将吐血的伤心程度降到最低。其实吐血是个很大方的人，所以他只会对浪费的部分感到伤心。

【输入】

输入文件第一行有两个整数 n 和 m，分别表示人数和食物种类数，食物已经按照 1 到 m 编号。

第二行有 m 个正整数 c1、c2、….、cm，ci 表示编号为 i 的食物的价格。

接下来 n 行，每行第一个整数 ki，表示第 i 个人喜欢的食物种类数，紧接着的 ki 个整数，是他喜欢的食物编号。

【输出】

输出文件仅有一个整数，即最小的浪费金额。

【输入样例】

```
6 10
3 5 7 11 6 9 50 7 1 11
1 9
5 1 2 4 8 9
1 3
4 1 7 9 10
2 5 6
10 1 2 3 4 5 6 7 8 9 10
```

【输出样例】

```
7
```

【数据规模】

1<=n<=35　　1<=m<=50　　1<=ki<=m

输入数据保证所有计算都在长整型范围内。

【参考程序】

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

const int maxn=500;
int n, m, c[100], a[50][100], h[50], sum, ans, s[100], ma,ss;
bool f;
```

```
int main()
{
    freopen("delicious.in","r",stdin);
    freopen("delicious.out","w",stdout);

    scanf("%d%d",&n,&m);

    ma=0;

    for (int i=1;i<=m;i++)  // 读入 m 样食物的价格
    {
        scanf("%d",&c[i]);
        if (c[i]>ma)          // 找出其中的最大价格
            ma=c[i];
    }

    for (int i=1;i<=n;i++)   // 读入每个人喜欢的食物编号
    {
        scanf("%d",&a[i][0]);
        for (int j=1;j<=a[i][0];j++)
        {
            scanf("%d",&a[i][j]);
        }
    }

    ans=ma*n;
    for (int z=1;z<=maxn;z++)      // 运行 500 次
    {
        memset(s,0,sizeof(s));
        for (int i=1;i<=n;i++)
        {
            h[i]=a[i][rand() % a[i][0]+1];   // 每个人随机挑一样食物
            s[h[i]]++;                       // 则喜欢该食物的人数加 1
        }
        sum=0;
        for (int i=1;i<=m;i++)  //计算零头的、浪费的食物的金额
        {
            sum+=(s[i] % 2)*c[i];
        }
        if (sum < ans)     // 必要的话，更新 ans
            ans = sum;
        f=true;
        while (f)
```

```
        {
            f=false;
            for (int i=1;i<=n;i++)
            {
                for (int j=1;j<=a[i][0];j++)  // 检查一下其他食物
                {                             // 需要的话再调整、优化
                    if (h[i]==a[i][j]) continue;
                    ss=sum;
                    if (s[h[i]] % 2==0)  //把h[i]换成a[i,j]试试
                        ss+=c[h[i]];
                    else
                        ss-=c[h[i]];
                    if (s[a[i][j]] % 2==0)
                        ss+=c[a[i][j]];
                    else
                        ss-=c[a[i][j]];
                    if (ss<sum)
                    {   sum=ss;
                        f=true;
                        s[h[i]]--;
                        s[a[i][j]]++;
                        h[i]=a[i][j];
                        break;
                    }
                }
                if (f) break;    // 找到了一种比随机方案更优的方案
            }    // 再试其他的食物，看是否需要更换方案
            if (sum < ans)   ans=sum;
        }
    }

    printf("%d\n",ans);

    return 0;
}     // 讨论：此处随机化算法的特点？为什么管用？
```

【参考程序（Pascal）】{ QMD }

```
program delicious;
var
  n,m:integer;
  ans:longint;
  like:array[1..35,1..50]of boolean;
  g:array[1..35,0..50]of integer;
  price:array[1..50]of longint;
  hash:array[1..33554432]of longint;
```

```
function f(status:longint):longint;
  var
    i,j,x:integer;
    backup,last,min,temp:longint;
  begin
    backup:=status;
    if hash[status]<>0 then f:=hash[status]
    else
    if status=0 then f:=0
               else
                begin
                  last:=status and(status xor(status-1));
                  status:=status-last;
                  x:=1;
                  while last>1 do
                    begin
                      last:=last div 2;
                      x:=x+1;
                    end;
                  min:=maxlongint;
                  for i:=1 to g[x,0] do
                    begin
                      temp:=f(status)+price[g[x,i]];
                      if temp<min then min:=temp;
                      for j:=1 to n do
                        if (status and(1 shl (j-1)))<>0 then
                          if like[j,g[x,i]] then
                            begin
                              temp:=f(status-1 shl(j-1));
                              if temp<min then min:=temp;
                            end;
                    end;
                  hash[backup]:=min;
                  f:=min;
                end;
  end;
procedure init;
  var
    i,j:integer;
  begin
    assign(input,'delicious.in');
    reset(input);
    readln(n,m);
    for i:=1 to m do
```

```
      read(price[i]);
    readln;
    fillchar(like,sizeof(like),false);
    for i:=1 to n do
      begin
        read(g[i,0]);
        for j:=1 to g[i,0] do
          begin
            read(g[i,j]);
            like[i,g[i,j]]:=true;
          end;
        readln;
      end;
    close(input);
  end;
procedure work;
  var
    all:longint;
  begin
    fillchar(hash,sizeof(hash),0);
    all:=1 shl n-1;
    ans:=f(all);
  end;
procedure print;
  begin
    assign(output,'delicious.out');
    rewrite(output);
    writeln(ans);
    close(output);
  end;
begin
  init;
  work;
  print;
end.
```