贪婪的动态规划

——浅谈贪心思想在动态规划中的应用

浙江省绍兴县柯桥中学 黄劲松

【关键字】

贪心法，动态规划，状态，时间复杂度

【摘要】

贪心法和动态规划是信息学竞赛中的两种常用算法，本文着重讨论了贪心的思想是如何巧妙的运用到动态规划的解题中的。全文分三个部分，首先讨论了贪心思想运用到动态规划解题中的可行性和必要性，然后就贪心思想在动态规划中的两种基本应用分别做了举例说明，最后总结全文。

【正文】

引言

贪心法和动态规划是信息学竞赛中的常用经典算法，而当某些问题的模型过于复杂的时候，由于状态过于庞大、转移困难等一系列的问题，常规的动态规划难于甚至无从下手。而在这个时候，巧妙的使用贪心思想，将其融入到动态规划的解题中，动态规划便焕发出了新的光彩。

1、贪心思想运用到动态规划中的必要性和可行性

动态规划的原理是：在求解问题的过程中，通过处理位于当前位置和所达目标之间的中间点来找到整个问题的解。整个过程是递归的，每到一个新的中间点

都是已访问过的点的一个函数。适合于动态规划法的标准问题必须具有下列特点：

- 1、整个问题的求解可以划分为若干个阶段的一系列决策过程。
- 2、每个阶段有若干可能状态。
- 3、一个决策将你从一个阶段的一种状态带到下一个阶段的某种状态。
- 4、在任一个阶段，最佳的决策序列和该阶段以后的决策无关。
- 5、各阶段状态之间的转换有明确定义的费用

在实际的动态规划的解题中，面临着两大困难：一是不知道题目是否可以用动态规划求解；二是即使能够想到用动态规划来求解，但是因为种种因素，算法的效率并不乐观。这个时候，使用贪心思想分析问题，可以让你在山穷水尽疑无路的时候，柳暗花明又一村。

在运用贪心思想的时候，主要是分析出问题的一些本质，或者分析出低效算法的一些冗余。当然，我们要根据题目的特殊信息，合理的运用好贪心思想，才能帮助动态规划发挥其强大的功效。

下文就贪心思想如何解决动态规划面临着的这两大困难分别做了举例说明。

2、贪心思想在动态规划中的应用一： 确立状态

动态规划当中，状态的确立是重点，而在实际的解题过程中，状态的信息往往是隐含的，这个时候，合理的运用贪心思想，可以迅速的从繁芜丛杂的问题背景中巧妙地抽象出状态。我们通过下面的例子来看一看，贪心思想是如何帮助动态规划确立状态的。

例题一　青蛙的烦恼[1]

题目大意：

池塘里有 *n* 片荷叶（1≤*n*≤1000），它们正好形成一个凸多边形。按照顺时针方向将这 *n* 片荷叶顺次编号为 1，2，…，*n*。

有一只小青蛙站在 1 号荷叶上，它想跳过每片荷叶一次且仅一次（它可以从所站的荷叶跳到任意一片荷叶上）。同时，它又希望跳过的总距离最短。

请你编程帮小青蛙设计一条路线。

算法分析：

问题似乎是一个 N 个点的 TSP 问题[2]，但却又是一个特殊的 TSP 问题——题目中说明，N 个点围成了一个凸多边形！

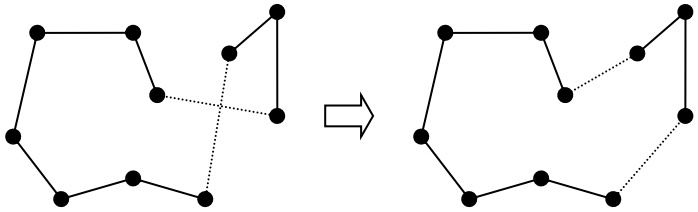
如何合理的运用这个特殊的条件成为了解题的关键。

原始的问题模型很容易使人放弃动态规划，因为状态难以抽象，状态转移也无从着手。但如果从问题特征出发，灵活运用贪心策略，却能巧妙的设计出一种高效的动态规划算法！

先介绍一下 TSP 问题的一种局部搜索算法——2-最优算法。它从随机的一种可选路径出发（称为路径 T），试图去改善它。我们在 T 中寻找出 2 条不相交的边，移动这两条边，如果新产生的路径比原来的路径优秀，那么就移动这两条边。这种移动称为 2-交换。如图所示：

[1] 经典问题

[2] 经典的旅行商问题，NP 问题



可以根据这个 TSP 问题的局部最优算法容易想到，原问题的最短路线中显然不存在相交的边，否则一定可以将这条路线“改进”成一条更优秀的路线。

根据上述结论可以知道：从 1 出发的第一步只能到 2 或者 *n*，否则产生的路线一定不会是最优的。

因此，原问题的模型变成了：寻找以 1 为起点，遍历凸多边形{1..N}中的顶点一次且一次的最短路线。根据上述结论可以知道，如果离开 1 到达 2，接下来的任务是寻找以 2 为起点，遍历凸多边形{2..N}中的顶点一次且一次的最短路线；如果离开 1 到达 N，接下来的任务是寻找以 N 为起点，遍历凸多边形{2..N}中的顶点一次且一次的最短路线。这是一个递归的过程！

因此，状态可以这样表示：*f*[*i*,*L*,0]表示从 *i* 出发，遍历{*i*..*i*+*L*-1}中的顶点一次且一次的最短距离；*f*[*i*,*L*,1]表示从 *i*+*L*-1 出发，遍历{*i*..*i*+*L*-1}中的顶点一次且一次的最短距离。状态转移方程是：

$$f[i,L,0] = min\{dist(i,i+1)+f[i+1,L-1,0], dist(i,i+L-1)+f[i+1,L-1,1]\}$$
$$f[i,L,1] = min\{dist(i+L-1,i+L-2)+f[i,L-1,1], dist(i+L-1,i)+f[i,L-1,0]\}$$

$$f[i,1,0] = 0,\ f[i,1,1] = 0$$

其中 *dist*(*a*,*b*)表示第 *a* 个结点和第 *b* 个结点之间的欧几里德距离。问题的答案是 *f*[1,*n*,0]

时间复杂度 O(n^2)。

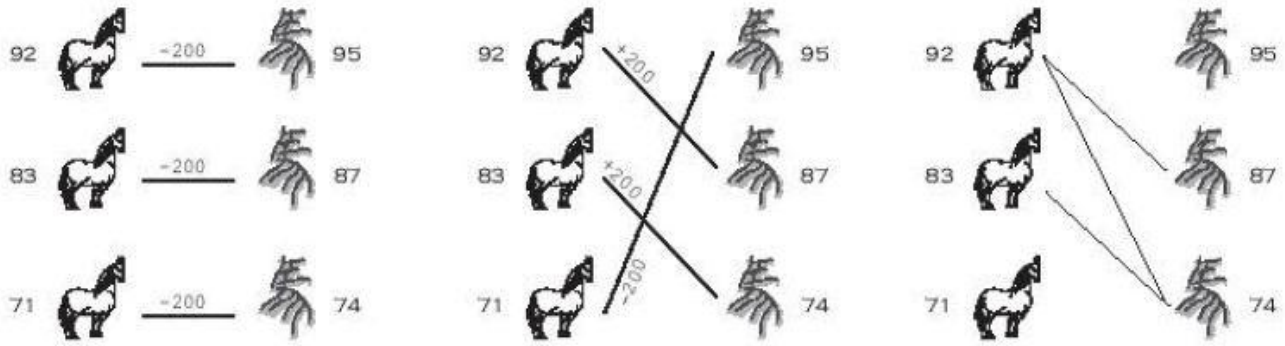
本题小结：

本题中运用贪心思想合理的分析出了问题最优解的一些特点，从而能够根据问题中的特殊条件确立出动态规划的状态，把看似 NP 的问题，用动态规划巧妙的解决了。

例题二　The Horse Racing [3]

题目大意：

中国古代的历史故事“田忌赛马”是为大家所熟知的。话说齐王和田忌又要赛马了，他们各派出 N 匹马（N≤2000），每场比赛，输的一方将要给赢的一方 200 两黄金，如果是平局的话，双方都不必拿出钱。现在每匹马的速度值是固定而且已知的，而齐王出马也不管田忌的出马顺序。请问田忌该如何安排自己的马去对抗齐王的马，才能赢最多的钱？



算法分析：

这个问题很显然可以转化成一个二分图最佳匹配的问题。把田忌的马放左边，把齐王的马放右边。田忌的马 *A* 和齐王的 *B* 之间，如果田忌的马胜，则连

[3] 上海赛区 2004 年 ACM 比赛试题

一条权为 200 的边；如果平局，则连一条权为 0 的边；如果输，则连一条权为－200 的边。

然而我们知道，二分图的最佳匹配算法的复杂度很高，无法满足 N = 2000 的要求。

我们不妨用贪心思想来分析一下问题。因为田忌掌握有比赛的“主动权”，他总是根据齐王所出的马来分配自己的马，所以这里不妨认为齐王的出马顺序是按马的速度从高到低出的。由这样的假设，我们归纳出如下贪心策略：

- 1、如果田忌剩下的马中最强的马都赢不了齐王剩下的最强的马，那么应该用最差的一匹马去输给齐王最强的马。
- 2、如果田忌剩下的马中最强的马可以赢齐王剩下的最强的马，那就用这匹马去赢齐王剩下的最强的马。
- 3、如果田忌剩下的马中最强的马和齐王剩下的最强的马打平的话，可以选择打平或者用最差的马输掉比赛。

第一个贪心策略的证明：

此时田忌的所有马都赢不了齐王的马，所以无论用最慢马去输还是用最快的马去输都同样是输，而处于贪心的思想，我们应该保留相比之下更强的马，因此用最慢的马去输一定不会比用别的马去输来得劣，所以这是最优策略。

证毕。

第二个贪心策略的证明：

假设现在齐王剩下的最强的马是 *A*，田忌剩下的最强的马是 *B*，如果存在一种更优的比赛策略，让 *B* 的对手不是 *A*，而使得田忌赢更多的钱的话，那么设此

时 *A* 的对手是 *b*，*B* 的对手是 *a*：

1、 若 *b*>*A*，则有 *B*>*a*，*b*>*A*。这个结果和 *B*>*A*，*b*>*a* 是相同的。

2、 若 a<*b*≤*A*，则有 *B*>*a*，*b*≤*A*。这个结果不如 *B*>*A*，*b*>*a* 来得优秀。

3、 若 *b*≤*a*≤*A*，则有 *B*>*a*，*b*≤*A*。这个结果和 *B*>*A*，*b*≤*a* 是相同的。

由此可知，交换各自对手后，一定不会使得结果变劣，那么假设是不成立的。

证毕。

第三个贪心策略的证明：

因为田忌最快的马也只是和齐王的马打平，那么田忌只能选择平或输，选择平的话，当然只能用最快的马去平了；选择输的话当时是用最慢的马去输来得值得，这和第一个贪心策略的思路是一样的。

证毕。

我们发现，第三个贪心策略出现了一个分支：打平或输掉。如果穷举所有的情况，算法的复杂度将比求二分图最佳匹配还要高；如果一概而论的选择让最强的马去打平比赛或者是让最差的马去输掉比赛，则存在反例：

- ✓ 光是打平的话，如果齐王马的速度分别是 1 2 3，田忌马的速度也是 1 2 3，每次选择打平的话，田忌一分钱也得不到，而如果选择先用速度为 1 的马输给速度为 3 的马的话，可以赢得 200 两黄金。
- ✓ 光是输掉的话，如果齐王马的速度分别是 1 3，田忌马的速度分别是 2 3，田忌一胜一负，仍然一分钱也拿不到。而如果先用速度为 3 的马去打平的话，可以赢得 200 两黄金。

虽然因为第三个贪心出现了分支，我们不能直接的按照这种方法来设计出一

个完全贪心的方法，但是通过上述的三种贪心策略，我们可以发现，如果齐王的马是按速度排序之后，从高到低被派出的话，田忌一定是将他马按速度排序之后，从两头取马去和齐王的马比赛。有了这个信息之后，动态规划的模型也就出来了！

设 *f[i,j]*表示齐王按从强到弱的顺序出马和田忌进行了 *i* 场比赛之后，从“头”取了 *j* 匹较强的马，从“尾”取了 *i-j* 匹较弱的马，所能够得到的最大盈利。

状态转移方程如下：

$$f[i,j]=\max\{f[i-1,j]+g[n-(i-j)+1,i],f[i-1,j-1]+g[j,i]\}$$

其中 *g[i,j]*表示田忌的马和齐王的马分别按照由强到弱的顺序排序之后，田忌的第 *i* 匹马和齐王的第 *j* 匹马赛跑所能取得的盈利，胜为 200，输为－200，平为 0。

本题小结：

虽然本题存在直接贪心的方法，不过它可以作为一个例子告诉大家，合理的运用贪心策略，分析出问题的一些本质之后，一些看似不能用动态规划做的题目便可以巧妙的确立出状态，继而可以用动态规划来求解。

3、贪心思想在动态规划中的应用二： 优化算法

一些动态规划的题目虽然容易确立出正确的状态以及轻松的写出状态转移方程。但是直序的算法往往效率不高。而贪心历来都是与“高效”一词分不开的。所以，合理的在动态规划中运用贪心思想，能够让原本效率低下的算法获得“重生”！

例题三 石子归并[4]

题目大意：

[4] 经典问题

在一个操场上摆放着一排 n（n≤1000）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花费。

试编程求出将 n 堆石子合并成一堆的最小花费。

算法分析：

这是一道经典的动态规划问题，用 $m[i,j]$表示第 i 堆石子合并到第 j 堆石子所需的最小花费，$w[i,j]$表示第 i 堆石子到第 j 堆石子的石子总数。状态转移方程如下：

$$m[i,j]=\min_{i<k\le j}\{m[i,k-1]+m[k,j]+w[i,j]\}$$

上述算法的状态总数为 $O(n^2)$，每个状态转移的状态数为 $O(n)$，每次状态转移的时间为 O(1)，所以总的时间复杂度为 $O(n^3)$。

显然，这个简单易懂的算法的过于低效了，我们需要进行优化。

我们尝试寻找动态规划中的冗余。不难发现，在枚举决策 k 的时候，上述算法实在是太盲目了，因为当有一些 k 如果作为决策点，肯定得不到最优解，我们可以不去枚举他们。

这是一种贪心的思想！

令 $s[i,j]$表示使得 $m[i,j]$取到最小时的 k。可以用四边形不等式[5]证明出

$$s[i,j]\le s[i,j+1]\le s[i+1,j+1],\qquad i\le j$$

这说明 $m[i,j]$的决策单调。于是状态转移方程可以变成如下的形式：

$$m[i,j]=\min_{s[i,j-1]\le k\le s[i+1,j]}\{m[i,k-1]+m[k,j]\}+w[i,j]$$

[5] 当函数 $w[i,j]$满足 $w[i,j]+w[i',j']\le w[i',j]+w[i,j'],\ i\le i'\le j\le j'$ 时，称 w 满足四边形不等式。

整体的时间复杂度则变为了：

$$O\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1+s[i+1,j]-s[i,j-1])\right)=O\left(\sum_{i=1}^{n-1}(n-i+s[i+1,n]-s[1,n-i])\right)=O\left(n^2\right)$$

本题的相关证明详见参考文献[3]

本题小结：

决策单调的动态规划是一类很常见的问题，例如 NOI2004 的 hut 就是一道不错的根据决策单调的性质来优化动态规划的题目。根据决策单调的性质，我们避免了一些无用的决策枚举，从而使得算法的复杂度降了级，这是贪心思想的经典体现。

例题四　The Lost House[6]

题目大意：

蜗牛的房子遗失在了一棵树的某个叶子结点上，它要从根结点出发开始寻找它的房子。有一些中间结点可能会住着一些虫子，这些虫子会告诉蜗牛它的房子是否在以这个中间结点为根的子树上，这样蜗牛就不用白跑路了。当然，如果有些结点没有住着虫子的话，那么可怜的蜗牛只有靠自己决定访问顺序来探索了。假设蜗牛走过一条边的耗费都是 1，且房子遗失在每个叶子结点的概率都是相等的，那么请问蜗牛找到他的房子的最小数学期望值？

我们约定，树上的结点数 n 最多为 1000，每个结点的分叉数 k 最多为 8。

例如在下面的这棵树当中：

[6] 北京赛区 2004 年 ACM 比赛试题

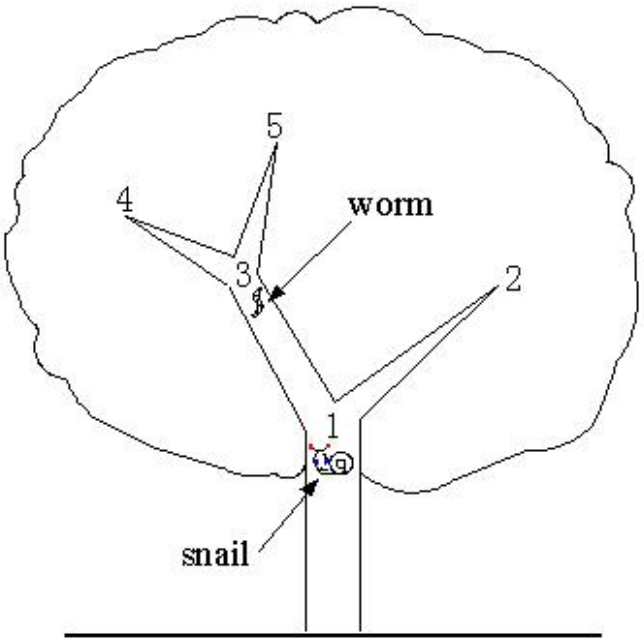


Figure-1

蜗牛从根结点 1 出发开始寻找它的房子，它的房子可能遗失在 2、4、5。在结点 3 上住着一只虫子，它会告诉蜗牛，以 3 为根的子树上是否有蜗牛的房子。蜗牛有两种走法。蜗牛可以先访问 2，如果它在那儿不能找到房子，那么它要回到根结点 1，再通过 3 来访问结点 4 (或 5)，如果还是不能找到它的房子，那么它又要回到结点 3，再去访问结点 5 (或 4)。在这种走法中，当房子分别位于 2、4、5 的时候，蜗牛需要走的步数分别是 1、4、6，期望值是(1+4+6)/3=11/3。显然，这种走法没有充分发挥虫子在这里起到的作用。在另一种走法中，蜗牛先访问结点 3,它可以从住在 3 上的虫子那里得知它的房子是否存在于 4 或 5 的信息。在这种走法中，当房子分别位于 2、4、5 的时候，蜗牛需要走的步数分别是 3、2、4，期望值是(3+2+4)/3=3。这种走法合理的利用了虫子提供的信息，得到了更优的数学期望值。

算法分析：

不难分析出本题的大体模型是用树的动态规划来求解。用 *Fa*[*i*]表示蜗牛的 House 在 i 为根的子树上的期望和，用 *Fb*[*i*]表示蜗牛的 House 不在 i 为根的子树上的时候遍历该子树需要的时间，用 *Leaves*[*i*] 表示 i 为根的子树上叶子节点的数目。问题的解答就是 *Fa*[根结点]/*Leaves*[根结点]。

如果结点 *u* 有 *k* 个儿子,我们按照 *S*[1]..*S*[*k*]进行访问，*Fa*[*u*]的计算方式是：

```
Fa[u] = 0; Fb[u] = 0;
for i = 1 to k do
  begin
    Fa[u] = Fa[u] + (Fb[u] + 1)×Leaves[S[i]] + Fa[S[i]];
    Fb[u] = Fb[u] + Fb[S[i]] + 2;
  end;
```

用公式的形式可以表示为：

$$Fa[u]=\sum_{i=1}^{k}(\sum_{j=1}^{i-1}((Fb[S_j]+2)+1)\times Leaves[S_i]+Fa[S_i]) \qquad ①$$

现在的问题就是如何决定访问儿子的顺序，不同的访问顺序会产生不同的 *Fa*[*u*]。我们要使得 *Fa*[*u*]尽量的小。

一种直观的方法是 *k* !枚举访问顺序，总体复杂度是 O(*nk*!)，实在是很低效。用状态压缩的动态规划进行二次动规的话，可以将复杂度降为 O($n2^k k$)，勉强可以接受了。（注：由于状态压缩的动态规划不是本文的重点，故这里不做展开）

但是我们的研究并没有因此停止!

我们尝试用贪心的思想来分析问题：考虑一种访问顺序中的两个相邻元素 *v* 和 *v*+1，如果交换 *v* 和 *v*+1 之后得到的值不如交换前的值，那么 *v* 一定在 *v*+1 的前面了。

具体证明如下：

我们对公式①来进行一些处理。

$$① = \sum_{i=1}^{k} Fa[S_i] + \sum_{i=1}^{k} Leaves[S_i] + \sum_{i=1}^{k} \sum_{j=1}^{i-1} (Fb[S_j] + 2) \times Leaves[S_i]$$

可以看出，交换 *v* 和 *v*+1 之后，对 $\sum_{i=1}^{k} Fa[Si]$ 和 $\sum_{i=1}^{k} Leaves[Si]$ 是不会产生任何影响的，关键是看 $\sum_{i=1}^{k} \sum_{j=1}^{i-1} (Fb[Sj] + 2) \times Leaves[Si]$ 是增大了还是减小了。把交换前和交换后的值做差：

$$Fa[u] - Fa[u]' = (Fb[S_v] + 2) \times Leaves[S_{v+1}] - (Fb[S_{v+1}] + 2) \times Leaves[S_v]$$

最后得到的 $(Fb[S_v] + 2) \times Leaves[S_{v+1}] - (Fb[S_{v+1}] + 2) \times Leaves[S_v]$ 则只跟元素 *v* 和 *v*+1 的信息有关，于别的元素的排列情况无关，所以元素 *v* 和 *v*+1 是可比的。

证毕。

另外，根据这个结果，可以得出另一个结论：如果 *A* 一定放在 *B* 前，*B* 一定放在 *C* 前，可以推导出 *A* 一定放在 *C* 前。

证明：

$$(Fb[S_A] + 2) \times Leaves[S_B] \le (Fb[S_B] + 2) \times Leaves[S_A]$$
$$(Fb[S_B] + 2) \times Leaves[S_C] \le (Fb[S_C] + 2) \times Leaves[S_B]$$

两式相乘得到：

$$(Fb[S_A] + 2) \times Leaves[S_C] \le (Fb[S_C] + 2) \times Leaves[S_A]$$

证毕。

上面这个结论说明，本题中的“可比性”是可以传递的，因此可以根据这个性质确定出一个全序关系，因而省去了枚举排列的部分，只需要对所有元素进行一次排序即可。时间复杂度为 O($nk\log_2 k$)，非常优秀。

本题小结：

从原始的动态规划模型入手，分析出算法的大体框架，巧妙的运用贪心思想来除去原始算法中的冗余，进而达到优化算法的目的。

4、贪心思想在动态规划中的应用总结

本文通过四个例题简单的介绍了贪心思想在动态规划中的两种简单应用——**确立状态**和**优化算法**。贪心思想运用于动态规划时的奇妙之处在于它合理的利用了问题中隐含的一些***特殊信息***，因而可以使得看似不能动态规划的题目确立出动态规划的状态，或者除去算法中的冗余，提高动态规划的效率。

“贪婪的动态规划”并不是一种算法，而是一种思想，要灵活的在动态规划中运用好贪心思想，关键是在于对问题的深入理解和推敲。这要求我们具备“勇于探索”、“大胆创新”、“举一反三”的能力。

【感谢】

NOI2005 浙江代表队全体成员的支持

浙江省绍兴县柯桥中学吴建峰老师的技术指导

【参考文献】

[1]刘汝佳，黄亮《算法艺术与信息学竞赛》

[2](美)Zbigniew Michalewicz David B.Fogel《如何解决问题——现代启发式方法》

[3]IOI2001 集训队论文——《动态规划算法的优化技巧》毛子青

【附录】

[参考程序]

[例题二]参考程序:


Horse.pas

[例题四]参考程序:


House.pas

[例二原题]

Tian Ji -- The Horse Racing

Description

Here is a famous story in Chinese history.

That was about 2300 years ago. General Tian Ji was a high official in the country Qi. He likes to play horse racing with the king and others.

Both of Tian and the king have three horses in different classes, namely, regular, plus, and super. The rule is to have three rounds in a match; each of the horses must be used in one round. The winner of a single round takes two hundred silver dollars from the loser.

Being the most powerful man in the country, the king has so nice horses that in each class his horse is better than Tian's. As a result, each time the king takes six hundred silver dollars from Tian.

Tian Ji was not happy about that, until he met Sun Bin, one of the most famous generals in Chinese history. Using a little trick due to Sun, Tian Ji brought home two hundred silver dollars and such a grace in the next match.

It was a rather simple trick. Using his regular class horse race against the super class from the king, they will certainly lose that round. But then his plus beat the king's regular, and his super beat the king's plus. What a simple trick. And how do you think of Tian Ji, the high ranked official in China?
Were Tian Ji lives in nowadays, he will certainly laugh at himself. Even more, were he sitting in the ACM contest right now, he may discover that the horse racing problem can be simply viewed as finding the maximum matching in a bipartite graph. Draw Tian's horses on one side, and the king's horses on the other. Whenever one of Tian's horses can beat one from the king, we draw an edge between them, meaning we wish to establish this pair. Then, the problem of winning as many rounds as possible is just to find the maximum matching in this graph. If there are ties, the problem becomes more complicated, he needs to assign weights 0, 1, or -1 to all the possible edges, and find a maximum weighted perfect matching...

However, the horse racing problem is a very special case of bipartite matching. The graph is decided by the speed of the horses -- a vertex of higher speed always beat a vertex of lower speed. In this case, the weighted bipartite matching algorithm is a too advanced tool to deal with the problem.

In this problem, you are asked to write a program to solve this special case of matching problem.

Input

The input consists of up to 50 test cases. Each case starts with a positive integer n (n<=1000) on the first line, which is the number of horses on each side. The next n integers on the second line are the speeds of Tian's horses.

Then the next n integers on the third line are the speeds of the king's horses. The input ends with a line that has a single `0' after the last test case.

Output

For each input case, output a line containing a single number, which is the maximum money Tian Ji will get, in silver dollars.

Sample Input

3

92 83 71

95 87 74

2

20 20

20 20

2

20 19

22 18

0

Sample Output

200

0

0

[例四原题]

The Lost House

Description

One day a snail climbed up to a big tree and finally came to the end of a branch. What a different feeling to look down from such a high place he had never been to before! However, he was very tired due to the long time of climbing, and fell asleep. An unbelievable thing happened when he woke up ---- he found himself lying in a meadow and his house originally on his back disappeared! Immediately he realized that he fell off the branch when he was sleeping! He was sure that his house must still be on the branch he had been sleeping on. The snail began to climb the tree again, since he could not live without his house.

When reaching the first fork of the tree, he sadly found that he could not remember the route that he climbed before. In order to find his lovely house, the snail decided to go to the end of every branch. It was dangerous to walk without the protection of the house, so he wished to search the tree in the best way.

Fortunately, there lived many warm-hearted worms in the tree that could accurately tell the snail whether he had ever passed their places or not before he fell off.

Now our job is to help the snail. We pay most of our attention to two parts of the tree ---- the forks of the branches and the ends of the branches, which we call them key points because key events always happen there, such as choosing a path, getting the help from a worm and arriving at the house he is searching for.

Assume all worms live at key points, and all the branches between two neighboring key points have the same distance of 1. The snail is now at the first fork of the tree.

Our purpose is to find a proper route along which he can find his house as soon as possible, through the analysis of the structure of the tree and the locations of the worms. The only restriction on the route is that he must not go down from a fork until he has reached all the ends grown from this fork.

The house may be left at the end of any branches in an equal probability. We focus on the mathematical expectation of the distance the snail has to cover before arriving his house. We wish the value to be as small as possible.

As illustrated in Figure-1, the snail is at the key point 1 and his house is at a certain point among 2, 4 and 5. A worm lives at point 3, who can tell the snail whether his house is at one of point 4 and 5 or not. Therefore, the snail can choose two strategies. He can go to point 2 first. If he cannot find the house there, he should go back to point 1, and then reaches point 4 (or 5) by point 3. If still not, he has to return point 3, then go to point 5 (or 4), where he will undoubtedly find his house. In this choice, the snail covers distances of 1, 4, 6 corresponding to the circumstances under which the house is located at point 2, 4 (or 5), 5 (or 4) respectively. So the expectation value is (1 + 4 + 6) / 3 = 11 / 3. Obviously, this strategy does not make full use of the information from the worm. If the snail goes to point 3 and gets useful information from the worm first, and then chooses to go back to point 1 then towards point 2, or go to point 4 or 5 to take his chance, the distances he covers will be 2, 3, 4 corresponding to the different locations of the house. In such a strategy, the mathematical expectation will be (2 + 3 + 4) / 3 = 3, and it is the very route along which the snail should search the tree.

Input

The input contains several sets of test data. Each set begins with a line containing one integer N, no more than 1000, which indicates the number of key points in the tree. Then follow N lines describing the N key points. For convenience, we number all the key points from 1 to N. The key point numbered with 1 is always the first fork of the tree. Other numbers may be any key points in the tree except the first fork. The i-th line in these N lines describes the key point with number i. Each line consists of one integer and one uppercase character 'Y' or 'N' separated by a single space, which represents the number of the previous key point and whether there lives a worm ('Y' means lives and 'N' means not). The previous key point means the neighboring key point in the shortest path between this key point and the key point numbered 1. In the above illustration, the previous key point of point 2 or 3 is point 1, while the previous key point of point 4 or 5 is point 3. This integer is -1 for the key point 1, means it has no previous key point. You can assume a fork has at most eight branches. The first set in the sample input describes the above illustration.

A test case of N = 0 indicates the end of input, and should not be processed.

Output

Output one line for each set of input data. The line contains one float number with exactly four digits after the decimal point, which is the mathematical expectation value.

Sample Input

```
5
-1 N
1 N
1 Y
3 N
3 N
10
-1 N
1 Y
1 N
2 N
2 N
2 N
3 N
3 Y
8 N
8 N
6
-1 N
1 N
1 Y
```

1 N

3 N

3 N

0

Sample Output

3.0000

5.0000

3.5000