Microsoft

十年以来，热爱着微软奇迹...

十年以后，能否和微软一起创造更多的奇迹？

编程之美 II

千里之行 始于足下

高鹏

09

目录

1 序

1.1 为何编写此稿?

阅读微软《编程之美》，颇受启发。寄希望能加盟微软，加入《编程之美》小组创作其 2 版。说不如做，因此开始动手！

深知此为万里长征，但已迈出第一步。

希望本稿能证明我的兴趣与实力，助我 Join MS！

1.2 此稿包含哪些内容?

学习《编程之美》、研究网上一些算法题，通过苦思冥想 + 猛实验，有了很多新的思路：

1. 效率更高，或者
2. 形式更简洁，或者
3. 方法的解释更易懂（相同算法不同思想）

的一些解决方法，当然是相比我所能搜集到的现有方法而言。由于只有 1 个月左右的时间编写，粗糙之处敬请谅解。

1.3 Why C#?

本稿采用 C#。

曾经是 C++程序员，对 C#一直报以怀疑。然而在查阅各种文献的时候发现整个.Net 的焦点都聚在 C#身上了，狐疑中随意一试—— a new story begins...!

面对诸如："C++效率更高，越底层越厉害，C#是微软想把大家变笨的诡计（只会搭积木）"的争议，我有一些想法，在发表之前，首先提及"大师"的几句话：

1. 当年 C/C++出现的时候，存在"汇编效率更高，越底层越厉害，应该用汇编"的争议...
2. 如果说伟大的科学是"站在巨人肩膀上的巨人"，那么在有些地方软件行业有点像是"站在侏儒脚上的侏儒..."
3. 一部伟大的文学著作，是以复杂的语法炫耀文学功底？还是以简单的语言表达深刻的思想？
4. 计算机语言发展的主线是："从贴近机器的思路向贴近人的思路的转变"，增强人们操控机器的能力。

我的想法是：

1. 没有谁更好，需求说了算，在面对"逻辑密集型"的需求的时候，C#简单高效；在面对"程序任务密集型"的需求的时候，C/C++更底层而强大的语法能提高运行效率。如：操作系统内核对效率要求极高的地方还得用汇编。
2. 在面对信息爆炸的时代，"逻辑密集型"的需求将越来越多。
3. 诚然，理解计算机底层运作机制是必要的，否则在使用高层语言的时候会写出很"费"的代码。但是学习和理解底层，并不代表应该"刀耕火种"。

2 分级组合（排列）法

在琢磨一些算法题的过程中，发现很多题目其实可以用同一种思路去解决：假设有 n 个元素集合，挨个取出所有元素，计算当取出 1，2...n 个元素的情况下，集合中任意 1，2...n 元素所能组合（或排列）出的情况（或个数）。

例如：求 1，2，3 这几个数求和能得到的数字：

0级:0

1级:1，2，3

2级:3，4，5

3级:6

在此，可将由 k 个元素组合（或排列）出的情况称为“k 级”的情况。对于不同的题目，“组合（或排列）”的“规则”可能不同，在这里，规则是“求和”。

思路很简单，却能解决很多问题（本章全部问题）。此法核心在于：看上去象是在构建母集合的幂集（共有 2^n 个），但其过程中常常可以排除大量重复情况，甚至只需要记录元素的个数，从而提高效率。

2.1　卡特兰数(Catalan)

问题

《编程之美》中提到了“买票找零”问题，查阅了下资料，此问题和卡特兰数 C_n 有关，其定义如下：

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for } n \geq 0.$$

卡特兰数真是一个神奇的数字，很多组合问题的数量都和它有关系，例如：

1. C_n = 长度为 2n 的 Dyck words 的数量。Dyck words 是由 n 个 X 和 n 个 Y 组成的字符串，并且从左往右数，Y 的数量不超过 X，例如长度为 6 的 Dyck words 为：

   XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY

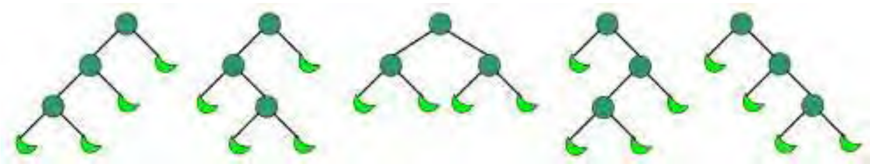
2. C_n = n 对括号正确匹配组成的字符串数，例如 3 对括号能够组成：

   ((())) ()(()) ()()() (())() (()())

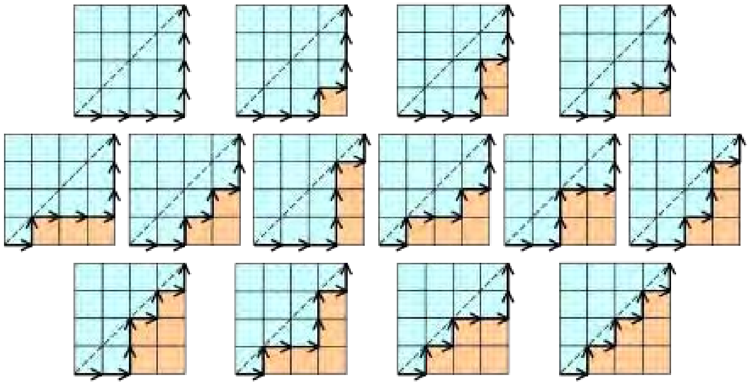
3. C_n = n+1 个数相乘，所有的括号方案数。例如，4 个数相乘的括号方案为：

   ((ab)c)d (a(bc))d (ab)(cd) a((bc)d) a(b(cd))

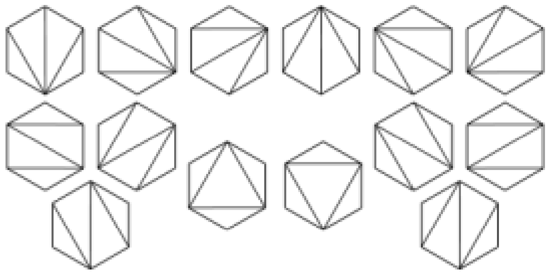
4. C_n = 拥有 n+1 个叶子节点的二叉树的数量。例如 4 个叶子节点的所有二叉树形态：

5. C_n = n*n 的方格地图中，从一个角到另外一个角，不跨越对角线的路径数，例如，4×4 方格地图中的路径有：



6. C_n = n+2 条边的多边形，能被分割成三角形的方案数，例如 6 边型的分割方案有：



7. C_n = 圆桌周围有 2n 个人，他们两两握手，但没有交叉的方案数。

在《Enumerative Combinatorics》一书中，竟然提到了多达 66 种组合问题和卡特兰数有关。

分析

“卡特兰数”除了可以使用公式计算，也可以采用“分级排列法”来求解。以 n 对括弧的合法匹配为例，对于一个序列 (() 而言，有两个左括弧，和一个右括弧，可以看成“抵消了一对括弧，还剩下一个左括弧等待抵消”，那么说明还可以在末尾增加一个右括弧，或者一个左括弧，没有左括弧剩余的时候，不能添加右括弧。

由此，问题可以理解为，总共 2n 个括弧，求 1～2n 级的情况，第 i 级保存所有剩余 i 个左括号的排列方案数。1～8 级的计算过程如

下表：

	0	1	2	3	4	5	6	7	8
1		1							
2	1		1						
3		2		1					
4	2		3		1				
5		5		4		1			
6	5		9		5		1		
7		14		14		6		1	
8	14		28		20		7		1

计算过程解释如下：

1级：只能放1个“(”;

2级：可以在一级末尾增加一个“)”或者一个“(”

以后每级计算时，如果遇到剩余 n>0 个“(”的方案，可以在末尾增加一个“(”或者“)”进入下一级；遇到剩余 n=0 个“(”的方案，可以在末尾增加一个“(”进入下一级。

奇数级只能包含剩余奇数个“(”的排列方案

偶数级只能包含剩余偶数个“(”的排列方案

从表中可以看出，灰色部分可以不用计算。

解法

关键代码为：

```
double Catalan(int n)
{
    if (n == 0) return 1;
    for (int i = 2; i <= 2 * n; i++)
    {
        var m = i <= n ? i : 2 * n + 1 - i;
        for (int j = (i - 1) & 1; j <= m; j += 2)
        {
            if (j > 0) arr[j - 1] += arr[j];
            if (j < n) arr[j + 1] += arr[j];
```

```
                    arr[j] = 0;
                }
            }
            return arr[0];
        }
```

其中：

n 为 C_n 中的 n；

arr = new double[n + 1];//arr[i]代表有 k 个括弧的时候，剩余"("个数为 i 的排列方案个数 arr[1] = 1;

讨论

算法复杂度为 $O(\frac{(n+2)(n+1)}{2} - 1)$，空间复杂度为 O(n+1)。相对于利用公式计算而言，此方法的优势在于——没有乘除法，只有加法。

2.2 序列 ABAB 对应字符串集合

问题

本题为 Google “Top Coder” 850 分例题。假设有这样一种字符串，它们的长度不大于 26 ，而且若一个这样的字符串其长度为 m ，则这个字符串必定由 a, b, c ... z 中的前 m 个字母构成，同时保证每个字母出现且仅出现一次。比方说某个字符串长度为 5 ，那么它一定是由 a, b, c, d, e 这 5 个字母构成。一旦长度确定，这个字符串中有哪些字母也就确定了，唯一的区别就是这些字母的前后顺序而已。

现在我们用一个由大写字母 A 和 B 构成的序列来描述这类字符串里各个字母的前后顺序： 如果字母 b 在字母 a 的后面，那么序列的第一个字母就是 A （After），否则序列的第一个字母就是 B（Before）；如果字母 c 在字母 b 的后面，那么序列的第二个字母就是 A ，否则就是 B； 如果字母 d 在字母 c 的后面，那么 ……

不用多说了吧？直到这个字符串的结束。

这规则甚是简单，不过有个问题就是同一个 AB 序列，可能有多个字符串都与之相符，比方说序列“ABA”，就有“acdb”、“cadb”等等好几种可能性。说的专业一点，这一个序列实际上对应了一个字符串集合。

那么现在问题来了：给你一个这样的 AB 序列，问你究竟有多少个不同的字符串能够与之相符？或者说这个序列对应的字符串集合有多大？注意，只要求个数，不要求枚举所有的字符串。

分析

在给定的“ABABB...”序列中，计算 1～n 级的情况(n=序列长度)，规则为“映射”。在本题目中，我们并不需要关心每级中包含元素是什么，而在于元素的个数。

假设现需要计算序列“ABBAA”，0 级只映射一个字符串“a”，并且 a 在 0 位置；1 级的时候，序列字符为‘A’，说明 b 在 a 后面，只能映射一个字符串“ab”，b 在 1 位置；2 级的时候，序列字符为‘B’，说明 c 在 b 前面，能够映射“cab”,“acb”，分别在 0、1 位置。以此类推，每级映射字符串数量计算如下表

级	序列	0	1	2	3	4	5	说明
0		1	0	0	0	0	0	字母 a 在位置 0 的字符串有 1 个
1	A	0	1	0	0	0	0	字母 b 在位置 1 的字符串有 1 个
2	B	1	1	0	0	0	0	字母 c 在位置 0、1 的字符串分别有 1 个
3	B	2	1	0	0	0	0	字母 d 在位置 0、1 的字符串分别有 2、1 个
4	A	0	2	3	3	3	0	字母 e 在位置 1、2、3、4 的字符串分别有 2、3、3、3 个
5	A	0	0	2	5	8	11	字母 f 在位置 2、3、4、5 的字符串分别有 2、5、8、11 个

由上表可以看出，对于每一级的情况，我们需要记录两个信息：

1. 本级能够影射的字符串数量；
2. 最大字符所在位置（因为后序 AB 序列只关心最大字符）；

综合起来，就是需要按照最后字符所在位置的不同，分别记录映射字符串的数量。

解法

按照分析思路，可以设置两个哈希表，一个记录 I 级数据，一个用来保存 I+1 级数据，计算一次后交换，代码如下：

```
for (int i = 1; i < n; i++)
{
    foreach (var kvp in dic1)
    {
        int m = 0, M = i;
        if (ab[i - 1] == 'A') m = kvp.Key + 1;
        else M = kvp.Key;
        for (int j = m; j <= M; j++)
        {
            if (!dic2.ContainsKey(j)) dic2.Add(j, 0);
            dic2[j] += kvp.Value;
        }
    }
    var temp = dic1; dic1 = dic2; dic2 = temp;
    dic2.Clear();//交换两个字典
}
其中：
string ab = ABAB 序列
 n = ab.Length + 1;//映射的字符串长度为"ABAB..."长度 +1
 dic1 = new Dictionary<int, double>(n);//key=字符出现的位置 value=字符串数量
 dic2 = new Dictionary<int, double>(n);
 初始化 dic1.Add(0, 1);
```

讨论

在本题目中，由于只需要记录每级元素的数量，“分级组合（排列）法”非常有效。后续应用中，我们还将看到类似例子。

给定序列长度	所有排列最大映射字符串数
3	5
4	16
5	61
6	272
7	1385
8	7936
9	50521
10	353792
11	2702765
12	22368256
13	199360981
14	1903757312
15	19391512145
16	209865342976

2.3 数组分割

问题

一个 2n 的整数数组，分成 n 元素两组，保证两组和之差的绝对值最小，设计算法，保证两组和之差的绝对值最小。

分析

依次拿出 2n 个整数，计算 1～n 级情况，第 i 级保存 i 数相加能得到的和。

解法一

使用 n 个哈希表来保存每级的情况，以排除重复：

```
for (int k = 1; k <= 2 * n; k++)
{
    var m = Math.Min(k - 1, n - 1);//只需要计算 1～n 级
    for (int l = m; l >= 0; l--)//倒序计算，避免重复
        foreach (var v in heap[l])//遍历每一级
        {
            var key = v + arr[k - 1]; //从 l 级扩充 l+1 级
            if (key <= halfSum) heap[l + 1].TryAdd(key);
```

```
        }
    }
```
其中：

arr 为保存整数的数组 int[]

```
n = arr.Length / 2;
halfSum = arr.Sum() / 2;

heap = new Dictionary<int>[n + 1] [1]; //heap[l]保存第 l 级的元素
```
初始化 heap[0]中只有一个元素 0

解法二

《编程之美》中提出另外一个思路：在从 l 级扩充 l+1 级的时候，也可以倒过来考虑，遍历所有可能的和，减去当然元素后看 l 级中是否有这个数，有则扩充。

这里可做一个改进：遍历所有可能的和的时候，并不需要从 1 到 halfSum，否则存在大量浪费。

将 arr[]排序，便可以很容易得到 j（j=1,2,..m）个元素所能求和的最大值 maxs[j]和最小值 mins[j]

```
for (int k = 1; k <= 2 * n; k++)
{
    var m = Math.Min(k, n);
    for (int l = m; l > 0; l--)
        for (int s = mins[l - 1]; s <= maxs[l - 1]; s++)
        {
            var key = s - arr[k - 1];
            if (heap[l - 1].ContainsKey(key)) heap[l].TryAdd(s);
            call_cnt2++;
        }
}
```

[1] Dictionary<int>是修改.net 源代码得到的单参数泛型哈希表

讨论

对于本题而言，可以看作是“相同代码，不同解释”的一种情况。在解法二中，对原书解法进行了改进。若考虑两种情况：

1. 元素数目多、值都比较小、重复元素多；
2. 元素数目少、值都比较大、重复元素少；

解法二初衷是为了适应第 1 种情况，在第 2 种情况下效率偏低。然而解法一使用了哈希表，能高效排除重复。通过测试表明大部分情况下都是解法一效率高。

需要说明的是，在元素个数多（n>100）、重复又较少的时候，两种解法效率都很低，因为规则是“求和”，所以能够排除重复不多。

在这样的情况下，可以采用“非精确求解”的一些方法，如《编程之美》的解法一、启发式搜索、遗传算法等。

2.4 最长递增子序列

问题

设 L=<a1,a2,…,an>是 n 个不同的整数的序列，L 的递增子序列是这样一个子序列 Lin=<aK1,ak2,…,akm>，其中 k1<k2<…<km 且 aK1<ak2<…<akm。求最大的 m 值。

分析

在给定的 L 序列中，计算 1～n 级所能构成的最长递增子序列。在计算过程中，实际上只需要保存每级最长的递增子序列中，尾数最小的那个，算法复杂度为 O(nlogn)。

解法一

```
for (int k = 0; k < arr.Length; k++)
{
    int index = -1;
    index = BinarySearch.FindFirstGreater<int>(ms, arr[k]) [2];
    if (ms[index - 1] == arr[k]) index = -1; //排除 arr 中重复元素
    if (index >= 0)
    {
        ms[index] = arr[k];//则替换 i+1 级
        maxLength = Math.Max(index, maxLength);
    }
}
return maxLength;
其中：
arr 为给定的整数序列
ms = new int[arr.Length + 1];
//ms[i]保存了长度为 i 的递增子序列中，尾数最小的序列的尾数
for (int i = 1; i < ms.Length; i++) ms[i] = int.MaxValue;
ms[0] = int.MinValue;
```

解法二

此问题还可以转化为“最长公共子序列”问题（参见下一小节）：将原序列 L 排序，并排除重复元素，得到新序列 L’，那么最长递增子序列就是 L 和 L’的最长公共子序列。

讨论

由于 ms[i]保存了长度为 i 的递增子序列中，尾数最小的序列的尾数，因此 ms 数组是严格递增的，可以利用二分查找将算法复杂度降至 O(nlogn)

证明：若 ms[i]≤ms[i-1]，则说明长度为 i-1 的序列尾数不是最小，与前提矛盾例如：

[2] BinarySearch.FindFirstGreater<int> 是自行开发的辅助类库中的一个，此函数试图找到刚好比给定元素“大”的元素的位置，找不到返回-1

长度为 4 的子序列：1，4，5，7

长度为 5 的子序列：1，3，4，5，6

那么长度为 4 的子序列存在 1，4，5，6，尾数比上例小，因此和前提矛盾。

2.5 最长公共子序列

问题

给定两个序列

X = { x1 , x2 , ... , xm }　　Y = { y1 , y2 , ... , yn }

求 X 和 Y 的一个最长公共子序列（LCS，Longest Common Sequence）

分析

一般典型的解法是利用动态规划，利用如下性质：

1. 若 xm = yn ，则 zk = xm = yn，且 Z[k-1] 是 X[m-1] 和 Y[n-1] 的最长公共子序列
2. 若 xm != yn ，且 zk != xm，则 Z 是 X[m-1] 和 Y 的最长公共子序列
3. 若 xm != yn，且 zk != yn，则 Z 是 Y[n-1] 和 X 的最长公共子序列

在一个二维表格上递推：

当 i = 0 , j = 0 时 , c[i][j] = 0
当 i , j > 0 ; xi = yi 时 , c[i][j] = c[i-1][j-1] + 1
当 i , j > 0 ; xi != yi 时 , c[i][j] = max { c[i][j-1] , c[i-1][j] }

算法复杂度为 O(mn)，辅助空间也需要 O(mn)。后来也又不少改进算法，基本思想是通过发现一些规律，减少在表格上的递推次数。

解法一

这个问题其实也可以利用“分级排列法”求解，算法复杂度接近

O(mlogn)，辅助空间 O(m+n)。基本思路和本文“最长递增子序列”的解法类似：

在给定的 X 序列中，求 1～m 级的情况，每一级中保存与 Y 的所有 LCS 中，末尾元素（在 Y 中位置）最低的那个。

假设 X=4，2，3，2，5，1，7，8，4，Y=1，2，3，4，8，2，4，5，6，8，每级计算过程如下表：

	0	1	2	3	4	5	6	7	8	9
Y	1	2	3	4	8	2	4	5	6	8
X	4	2	3	2	5	1	7	8	4	
	-1	-	-	-	-	-	-	-	-	-
4	-1	3	-	-	-	-	-	-	-	-
2	-1	1	5	-	-	-	-	-	-	-
3	-1	1	2	-	-	-	-	-	-	-
2	-1	1	2	5	-	-	-	-	-	-
5	-1	1	2	5	7	-	-	-	-	-
1	-1	0	2	5	7	-	-	-	-	-
7	-1	0	2	5	7	-	-	-	-	-
8	-1	0	2	4	7	9	-	-	-	-
4	-1	0	2	3	6	9	-	-	-	-

那么计算过程可以解释为：

1. 先从 X 中取出第一个元素 4，通过 yDic 发现 Y 中包含 4，位置为 3，6，那么长度为 1 的公共子序列，末尾元素在 Y 中最低位置为 3；
2. 从 X 中取出第二个元素 2，通过 yDic 发现 Y 中包含 2，位置为 1，5，那么长度为 1 的公共子序列，末尾元素在 Y 中最低位置更新为 1。长度为 2 的公共子序列，末尾元素在 Y 中最低位置为 5；
3. 以此类推，遇到 Y 中没有的元素，则直接略过。

关键代码如下：

```
int maxL = 0;
for (int i = 0; i < X.Count; i++)//处理 X 中每一个元素
{
        var x = X[i];
        if (yDic.ContainsKey(x))//如果 Y 中有相同元素 x
        {
                var pos = yDic[x];//获取 x 在 Y 中所有位置信息
                for (int j = pos.Count - 1; j >= 0; j--)//循环所有位置
                {
                        var g = ~BinarySearch.Find<int>(lcss, pos[j], 0, maxL + 1) 3;
                        if (g < 0) continue; // lcss 中已经有此元素
                        lcss[g] = pos[j];
                        maxL = Math.Max(maxL, g);
                }
    }
}
```

其中：

lcss = new int[X.Count + 1]，lcss[i]记录了长度为 i 的 LCS 的末尾元素在 X 中的最低位置。

```
lcss[0] = -1;//初始化为最小值
for (int i = 1; i < lcss.Length; i++) lcss[i] = Y.Count;//初始化为最大值
```

yDic 是 Y 元素位置的索引：它是一个哈希表，key 为 Y 中的元素，value 保存此元素在 Y 中出现位置（从小到大），建立的时候可以采用二分查找插入位置。

解法二

从上述解法中可以看出，在循环元素 x 在 Y 中位置信息的时候存在浪费——先从高位计算，低位置可能覆盖高位计算结果。

特别是在 Y 中存在大量重复元素的时候，程序效率会受到比较大的影响。

那么，我们可以做如下改进：从低位置和高位置两头同时计算，如果发现两者拟更 lcss 改同一个位置，那么以低位置为准，并跳出循

[3] BinarySearch.Find<T>：二分查找某元素位置，如果找不到，返回刚好比待查找元素大的元素位置求补。

环——因为以后的计算都将针对这个位置，所以没必要计算（想想为什么？）。

另外，yDic[x]中已经使用了的位置信息也不需要再进行计算。

关键代码如下：

```
renewDic.Clear();
var xLs = yDic[x];//获取 x 在 Y 中所有位置信息
for (int h = xLs.First, r = xLs.Last; h <= r; h = xLs.Next(h), r = xLs.Prev(r))
{
    var lo = ~BinarySearch.Find<int>(lcss, xLs[h], 0, maxL + 1);
    var hi = lo;
    if (h != r) hi = ~BinarySearch.Find<int>(lcss, xLs[r], 0, maxL + 1);
    maxL = Math.Max(maxL, hi);

    SetRenew(lo, h);//设置一个更新
    if (lo == hi) break;
    SetRenew(hi, r); //设置一个更新
}
foreach (var kvp in renewDic) RenewLcss(kvp.Key, xLs, kvp.Value, Y);//最后一并更新
```

其中：

xLs 中保存了元素 x 在 Y 中的位置信息（从小到大），xLs.Next 和 xLs.Prev 方法可以跳过已经使的位置信息。

如果边循环边更新 lcss 将导致后续位置更新出错，所以这里先保存所有更新信息，最后一并更新——renewDic 是更新 lcss 的信息哈希表，key = 在 lcss 中要更新的位置，value = 待更新值在 xLs 中的位置。

讨论

此算法实现了接近 O(mlogn)的时间复杂度，O(m+n)的空间复杂度，适用于比较长的（>10）的 LCS 求解情况——对于长度上百万的两个字符串，求解时间都能够控制在秒级。

然而因为需要建立索引，对于比较短的（<10）的 LCS 求解情况（如英文字典应用），也可以采用动态规划的改进算法，如：时间复杂度为 O(p(m-p))，空间复杂度为 O(m+n)的算法（p 是 LCS 长度）。

2.6　计算字符串的相似度

问题

许多程序会大量使用字符串。对于不同的字符串，我们希望能够有办法判断其相似程序。我们定义一套操作方法来把两个不相同的字符串变得相同，具体的操作方法为：

1. 修改一个字符（如把“a”替换为“b”）；
2. 增加一个字符（如把“abdd”变为“aebdd”）；
3. 删除一个字符（如把“travelling”变为“traveling”）；

比如，对于“abcdefg”和“abcdef”两个字符串来说，我们认为可以通过增加/减少一个“g”的方式来达到目的。上面的两种方案，都仅需要一 次 。把这个操作所需要的次数定义为两个字符串的距离，而相似度等于“距离+1”的倒数。也就是说，“abcdefg”和“abcdef”的距离为 1，相似度为 1/2=0.5。给定任意两个字符串，你是否能写出一个算法来计算它们的相似度呢？

分析

《编程之美》中采用递了归方法，其运算量是指数级增长的，例如：

strA = "appidimiologist";

strB = "epidemiologist";

运算时间特别长。当然可以通过排除搜索树中的重复，来使得算法接近自底向上的动态规划方法。

实际上这个问题可以采用“最长公共子序列”来解决。

解法

```
int len = LonComSeq<char>.GetLength(strA.ToCharArray(), strB.ToCharArray()) [4];
return Math.Max(strA.Length - len, strB.Length - len);
```

讨论

对较长字符串，利用 2.5 节中的算法，能够以快好几个数量级的速度，得到相同的结果。如前文提到，LonComSeq<T> GetLength 能够轻松应对长度上百万的字符串。

2.7 游戏 24 点

问题

给 4 个 1～13 之间数：n_1 n_2 n_3 n_4，是否能只使用四则运算使这 4 个数运算得到 24？

分析

依次拿出给定的 4 个数字，计算 1～4 级的情况，规则为“四则运算”。

需要说明的是，这道题和前面题目每级发展方法有点不同：前面都是拿出一个元素，将所有 i 级扩展到 i+1 级；然而在本例中，存在计算优先级(括号方案)的问题，例如：照原来的方法，可以得到 $n_1 * (n_2 + n_3 + n_4)$的情况，但是无法得到$(n_1 + n_2) * (n_3 + n_4)$的情况。

因此，发展方法改为：先将 n 个数放入 1 级(1 个数不需要计算)，

[4] LonComSeq<T> GetLength 是自习开发的辅助类库中的一个，能够找出 LCS 的长度。

然后执行 n 步计算：

1 级和自己发展到 2 级，1 级和 2 级发展 3 级...

2 级和自己发展 4 级，2 级和 3 级发展 5 级...

以此类推...

解法

首先设计一个表达式类 Expression，用以表示 n 个数四则运算所能得到的表达式，然后设计一个表达式集合类 ExpreSet，用来装相同数字所有组成的所有表达式，关键代码如下：

```
for (int i = 0; i < nExpres.Length; i++)
    for (int j = i; j < nExpres.Length - 1; j++)//以 i 级为基准向上发展
    {
        if (i + j + 2 > nExpres.Length) break;//大于输入数字的数量，无需再往上发展

        for (int ii = 0; ii < nExpres[i].Count; ii++)
            for (int jj = (i == j ? ii + 1 : 0); jj < nExpres[j].Count; jj++)
            {                                    //第 i 代和第 j 代两两组合
                var ex1 = nExpres[i][ii];//较短表达式
                var ex2 = nExpres[j][jj];//较长表达式

                if (ex1.IsOverlap(ex2)) continue;//排除已包含相同数字的表达式

                nExpres[i + j + 1].TryAdd(ex2, ex1);//发展下一级
            }
    }
```

其中：

nums 为输入的整数数组；

nExpres = new ExpreSet[nums.Length];// nExpres [n]中装了所有 n 个数字所能组合出的表达式；

nExpres [0]中装了 1 个数字能够组成的所有表达式。

讨论

运行程序，任意输入 4 个数字：5　9　12　3，得到：

1. (5-(9/3))*12 = 24
2. (9-5)*3+12 = 24
3. 12-((5-9)*3) = 24

4. (12-9+5)*3 = 24
5. (5-(9-12))*3 = 24
6. (12+5-9)*3 = 24
7. (12-(9-5))*3 = 24
8. (12-9)*(3+5) = 24

在 ExpreSet.TryAdd(exp1,exp2)方法中，已经考虑到了加法和乘法的交换率问题，所以排除了一些不必要的解。本程序实际上能够解决任意多个数字运算的情况。

2.8 寻找符合条件的整数

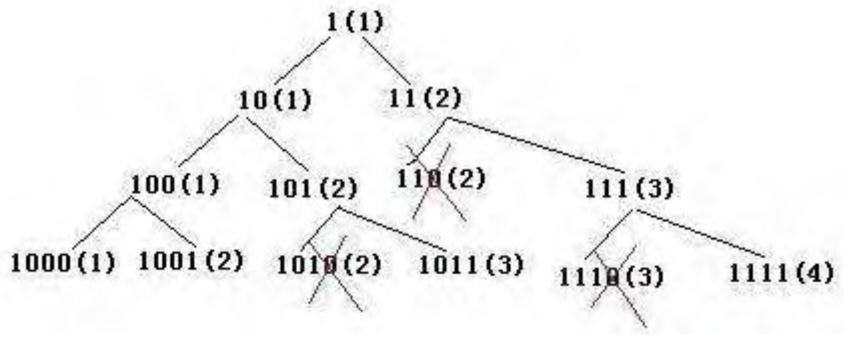
问题

任意给定一个正整数 N，求一个最小的正整数 M（M >1），使得 N * M 的十进制表示形式里只含有 1 和 0，例如：

1 * 1 = 1　　2 * 5 = 10
3 * 37 = 111　　4 * 25 = 100
5 * 2 = 10　　6 * 185 = 1,110
7 * 143 = 1,001　　8 * 125 = 1,000
9 * 12,345,679 = 111,111,111

解法一

既然乘积是类似 101011...的形式，那么可以使用二叉树去遍历所有乘积，从小到大，直到找到满足条件的。例如，下图是 N=3 时搜索的情况：



需要说明的是，在搜索二叉树的同一层中，相同余数的节点以后

发展出来的节点余数也是相同的（若 X≡Y (mod N)，则 X*10≡Y *10 (mod N) X*10+1≡Y *10+1 (mod N)），所以只需要保留一个节点即可。

关键代码如下：

```
var remainders = new int[n]; //设置一个数组来保存每层的余数（N 的余数个数≤N）
var queue = new Queue<BigInt 5>(); //设置一个队列来实现分层搜索二叉树
queue.Enqueue((BigInt)1);
while (queue.Count > 0)
{
    var v = queue.Dequeue();
    var r = v % n;
    if (r == 0)
    {
        Console.WriteLine("{0} * {1} = {2}", n, v / n, v);
        break;
    }
    if (remainders[r] == v.Digit) continue;
    else remainders[r] = v.Digit;
    v *= 10; queue.Enqueue(v);
    queue.Enqueue(v + 1);
}
```

解法二

我们也可以反过来考虑，去遍历乘数，例如，当 N=13 时，采用“试乘”的办法去得到 M——13 的个位数是 3，能够让 3 变成 1 的数只有 7（3×7=21），依次类推：

x					1	3
y			8	5	4	7
z					9	1
z				5	2	
z			6	5		
z	1	0	4			
z	1	1	1	1	1	1

需要注意的是，个位不可以“试乘”0，因为这样得到的 M 不是最小的（还可以缩小 10 倍），其它高位都可以“试乘 0。

[5] BigInt 是自习开发辅助类库中的个，用于计算非常大的整数。

另外，还可能出现无限循环的“试乘”：

x							1	2
y				3	3	4	2	5
z							6	0
z						3	0	
z					5	1		
z				4	1			
z			4	0				
z		4	0					
z	4	0						

当 N 为 d 位数的时候，“试乘”可能会出现 d 位循环出现，需要排除，关键代码如下：

```
Find_01_Format(BigInt x, BigInt y, BigInt z, int digit, Dictionary<string> cycleDic)
{
    for (int m = digit > 0 ? 0 : 1; m < 10; m++)
    {
        var sum = m * x[0] + z[digit];
        var u = sum % 10;
        if (u == 1 || u == 0)//找到一个能将最后乘积 digit 位变为 0 或 1 的数
        {
            y[digit] = m;
            var p = (x * m).LeftShift(digit).Add(z);
            if (min.Digit > 0 && p >= min) continue;//大于当前最小值，剪枝
            <检查是否找到结果><检查是否出现循环>
            Find_01_Format(x, y.Clone(), p, digit + 1, copyDic);
        }
    }
}
```

其中：

x 是被乘数，y 是乘数，z 是乘积

digit 是当前“试乘”的位数

cycleDic 是用于判断是否出现循环的哈希表

BigInt. LeftShift(k)是将 10 进制数左移 k 位，等价于×10^k

BigInt.Add(n)是将本数字加上 n 后返回自身。

min 保存的当前搜索到的最小的 z

解法三

当 N 比较大的时候，两种方法都比较慢，例如 N=987654 时：

987654 * 1,113,861,738,118,815 = 1,100,110,001,100,000,110,010

能否在秒级解决 10 万以下的数呢？

可以这样考虑，我们要得到的乘积只包含 0 和 1，即：

$z = b_1 10^0 + b_2 10^1 + b_3 10^2 + ... + b_k 10^{k-1}$

其中，$b_i = 0$ 或 1，$i = 1$，2，3，...，k

考虑无穷数列：$10^0 \% N$，$10^1 \% N$，$10^2 \% N$，...，由于 N 的余数总是< N，所以这个数列必将存在循环节，例如，当 N = 13 时，此数列为：

1, 10, 9, 12, 3, 4, 1, 10, 9, 12, 3, 4, 1, 10, 9, 12, 3, 4, 1, 10...

现在问题就可以转化为：在这个无穷数列中，挑选一些数出来，使得他们的和能够整除 N，一种挑选方案实际上对应于 $b_1 b_2 b_3$...的一种真值指派（每一位选中为 1，否则为 0）。

到此我们已经发现，这个问题可以采用“分级组合法”：

给定无穷序列 $10^0 \% N$，$10^1 \% N$，$10^2 \% N$，...求 1～n 级的情况。第 i 级保存序列中 i 个数相加所能得到和，直到找到最小的一个和能够整除 N。

代码如下：

```
var heap = new List<SumSet>(); // SumSet 类保存数列中 n 个数相加所能得到的和
heap.Add(new SumSet());
heap[0].Add(0);

int d = 1, b = 1, s = 0, cycleStart = -1;
SumSet.Sum min = null;
while (true)
{
    var r = b % n;
    //找到循环节开始的那个元素
    if (cycleStart < 0 && heap.Count > 1 && heap[1].ContainsKey(r)) cycleStart = r;
    if (r == cycleStart)   s++; //进入下一个循环周期，前 s 级不需要再扩展
```

```
        bool found = false;
        heap.Add(new SumSet());
        for (int lvl = d - 1; lvl >= s; lvl--)
            foreach (var sum in heap[lvl])
            {
                var newSum = sum.Add(r, d - 1);
                if (newSum.value % n == 0)
                {
                    found = true;
                    if (min == null || newSum < min) min = newSum;
                }
                heap[lvl + 1].Renew(newSum);
            }
        if (found) break;
        b *= 10 % n;
        b %= n;
        d++;
}
```

其中：

n 是给定的数字 N

d 表示当前计算到无穷数列的哪一位

b 依次是 10^0，10^1，10^2，10^3…，为了防止越界，计算过程中先求余，结果不变。

讨论

通过“分级组合法”基本能够秒级解决 10 万以下的数字了，逐数测试时，相比前面两法“卡卡”，真可谓酣畅淋漓！

由于从下往上看每级元素数量列呈现“橄榄型”——中间多两头少，在面对更大的数字的时候，未能到达此数字之前，就被困扰大量计算中，类似“广度优先搜索”。那么在需要计算这么大的数字之前，还可以改进此算法，利用“深度优先搜索”、“启发式搜索”，或者遗传算法给出近似解。

2.9 连续子数组和的最大值

问题

给定一整数数组，求连续的子数组和的最大值，例如：

1， -2， 3， 5， -3， 2 最大值为 8

0， -2， 3， 5， -1， 2 最大值为 9

分析

《编程之美》中给出的算法很精炼，然而解释却比较复杂，如果从“分级组合”的角度去理解要方便很多。

解法

设置两个整数变量：cur 和 sum，从给定数组中依次取出所有元素，加到 cur 上去，当 cur<0 时候重置 cur。sum 记录 cur 出现过的最大值：

```
var cur = array[0];
var sum = cur;
for (int i = 1; i < array.Length; i++)
{
    cur = cur < 0 ? array[i] : cur + array[i];
    if (sum < cur) sum = cur;
}
return sum;
```

扩展问题二解法

如果要求得到最大和的连续子数组的起始位置，那么通过以上思路就更加容易写出代码：

```
int start1 = 0, start2 = 0, end = 0;
var cur = array[0];
var sum = cur;

for (int i = 1; i < array.Length; i++)
```

```
{
    if (cur < 0)
    {
        cur = array[i];
        start2 = i;
    }
    else cur += array[i];
    if (sum < cur)
    {
        sum = cur;
        end = i;
        start1 = start2;
    }
}
//返回 [start1, end]
```

本例中，我们用 start1，start2 来记录起始位置，end 来记录终止位置。start2 相当于“工作变量”，start1 保存历史最大和连续子数组的起始位置。

扩展问题一解法

如果给定的是数组是首尾相连的循环数组，如何求解？

首先设计一个函数，求解给定数组中，从 from 开始的，最多到 to 结束的最大和连续子数组的末尾位置，思路和前面解法类似。

```
int MaxSum_Position(int[] array, int from, int to)
{
    int step = Math.Sign(to - from);
    int end = from;
    var cur = array[from];
    var sum = cur;
    for (int i = from + step; i != to + step; i += step)
    {
        cur += array[i];
        if (sum < cur)
        {
            sum = cur;
            end = i;
        }
    }
```

```
        return end;
    }
```

有了这个函数以后，可以将循环数组中的问题分成两种情况：一种是连续子数组跨越了 0 位置的，一种是没有跨越的：

```
    int sum1 = MaxSum(array);//不跨越 0 位置的最大和

    int i = MaxSum_Position(array, 0, array.Length - 1);
    int j = MaxSum_Position(array, array.Length - 1, 0);

    int sum2 = 0;
    if (i > j) j = i + 1;
    for (int k = 0; k < array.Length; k++)
    {
        sum2 += array[k];
        if (k == i) k = j - 1; //跳过中间空缺部分
    }
    return Math.Max(sum1, sum2);
```

讨论

本题是“相同算法不同思想”的例子，通过另外一个角度去看算法，不但更加容易理解，并且求解扩展问题也更容易。

3 数字之魅

3.1 数组循环移位

问题

设计一个算法，把一个含有 N 个元素的数组循环右移 K 位，要求时间复杂度为 O（N）。

解法一

看到这个题目，第一个想法是：如果程序中真遇到这个需求，以其去移动具体元素，不如改变数组访问方式，这样能实现 O(1)的复杂度。

设计一个类 CircleArray<T>，其中：

```
List<T> items; //用来保存给定的数组；
int benchmark = 0; //访问基准
public T this[int i] //然后构造索引器，改变数组访问方式
{
    get
    {//省略边界检查
        var j = (benchmark + i) % Count;
        return items[j];
    }
    set
    {//省略边界检查
        var j = (benchmark + i) % Count;
        items[j] = value;
    }
}

public void LeftShift(int k) //循环左移
{
    if (Count <= 0) return;
    benchmark = (benchmark + k) % Count;
}
```

```
public void RightShift(int k) //循环右移
{
    if (Count <= 0) return;
    k = k % Count;
    benchmark -= k;
    if (benchmark < 0) benchmark += Count;
}
```

解法二

那么有读者可能会问，对于“非移动不可”的情况呢？《编程之美》中给出的解法可谓相当简练：

```
RightShift(int* arr, int N, int k)
{
    K %= N;
    Reverse(arr, 0, N – K - 1);
    Reverse(arr, N - K, N - 1);
    Reverse(arr, 0, N - 1);
}
```

然而也存在浪费的移动，那么是否有算法能够实现所有元素“一次到位”？而又不用辅助空间？

假设有数组：1 2 3 4 5 6

左移 2 位得：3 4 5 6 1 2

右移 4 位得：3 4 5 6 1 2

由此可以看成，左移 k 位 = 右移 n-k 位

再继续分析：如果我们想“一步到位”，那么对于左移 3 位：可以将 4 和 1 交换，5 和 2 交换，6 和 3 交换，得到：

4 5 6 1 2 3

仅交换了 3 次就得到结果，相比 Reverse 法少了 2 次，但是不是所有情况都这么简单？

再继续分析：对于序列 1 2 3 4 5 6 7，我们需要左移 3 位，按照

前面的“一步到位”法可以得到：

4561237，而我们期望得到的是 4567123

由于 7%3=1，所以最后还剩了一个 7，所以出现了错误...那么是不是“一步到位”法行不通？

再仔细观察下，其实剩下的 7 可以递归地用统一方法去解决：

将序列的最后 3 位局部循环右移 1 位：

4561237->4567123

由此，我们可以写出一个循环左移、循环右移互相调用的递归算法，实现“一步到位”，并且还提供了局部循环移动的能力。另外，由于循环左移、循环右移可以相互等价，这个算法还能自动寻找短的路走。

利用 C#中的扩展方法，我们可以对 IList<T>接口进行扩展，将来所有实现此接口的集合（Array，List 等）的实例都可直接调用，非常方便。

```
public static class CircleShift
{
    public static void RightShift<T>(this IList<T> target, int k)
    {
        if (target.Count == 0 || k <= 0) return;
        k %= target.Count;
        RightShift(target, k, 0, target.Count);
    }
    public static void RightShift<T>(this IList<T> target, int k, int s, int n)
    {
        if (target.Count == 0 || k <= 0) return;
        var r = n % k;
        for (int i = s + n - 1 - k; i >= s + r; i--) Swap(i, i + k, target);
        if (r > 0) LeftShift(target, r, s, r + k);
    }
    public static void LeftShift<T>(this IList<T> target, int k)
```

```
        {
            if (target.Count == 0 || k <= 0) return;
            k %= target.Count;
            LeftShift(target, k, 0, target.Count);
        }
        public static void LeftShift<T>(this IList<T> target, int k, int s, int n)
        {
            if (target.Count == 0 || k <= 0) return;
            var r = n % k;
            for (int i = s + k; i < s + n - r; i++) Swap(i, i - k, target);
            if (r > 0) RightShift(target, r, s + n - r - k, r + k);
        }
    }
```

其中：

k 是要移位的位数

对于局部循环位移的函数，s 是起始号，n 是长度。

每次“一步到位”交换以后，再将余数长度的“尾巴”反向循环移动即可。

讨论

在实际应用中的大多数情况下，解法一的 O(1)算法是最好的选择。解法二 O(n)可谓实现了“效率高、实用好、形式简洁”的统一。

3.2 斐波那契数列

问题

《编程之美》中“斐波那契数列”一节中有一个扩展问题：设 A(0)=1, A(1)=2, A(2)=2, 当 n>2 时, 有 A(k)= A(k-1)+ A(k-2)+ A(k-3)。

那么如何求 A(n) % m？ (m<100000)

解法一

这里可以运用 3.1 中数组循环移位的解法一：

```
if (n <= 0) return 1;
var a = new CircleArray<double>(4);
a[3] = 2; a[2] = 2; a[1] = 1;
for (int i = 2; i < n; i++)
{
```

```
        a.LeftShift(1);
        a[3] = a[2] + a[1] + a[0];
        a[3] %= m;
}
return a[3];
```

使用 O(1)循环移位的类，实现了比较简洁的代码，然而此例只能说明一个应用(亦可作为工具检验其它解法)，对于 n 很大的情况，还得用矩阵。

解法二

$$\begin{bmatrix} A(1) \\ A(2) \\ A(3) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} A(k-2) \\ A(k-1) \\ A(k) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^{k-2} \times \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$$

主要代码如下：

```
if (n <= 0) return 1;
Matrix f = Matrix.Identity(3, 3);
Matrix sum = new Matrix(new double[] { 0, 1, 0, 0, 0, 1, 1, 1, 1 }, 3);
var vector = new Matrix(new double[] { 1, 2, 2 }, 3);

if (n < 3) return vector[(int)n, 0];
for (n -= 2; n > 0; n >>= 1)
{
        if ((n & 1) > 0)
        {
                f *= sum;    f %= m;
        }
        sum *= sum;
        sum %= m;
}
f *= vector;
return f[2, 0] % m;
```

说明：

假设 n=5，其二进制表示为 101

令 $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

则 $A^5 = A^{2^0} \times A^{2^2}$

在计算过程中,sum 依次为 $A^{2^0}, A^{2^1}, A^{2^2}, A^{2^2} \ldots$,如果 n 的二进制表示中遇到 1,则将 sum 累乘到 f 上去。

因“先乘最后求余”等价于“每次求余后再乘最后再求余”所以在计算的过程中不断求余，防止越界。

讨论

利用矩阵相乘的方法，能够将复杂度降到 O(logn)，在计算 $A(2^{60})\%m(m<100000)$的时候，效果特别明显。例如 $A(2^{60})\%12345=7307$，计算时间<<1s。

3.3 找重复数字

问题

假设你有一个用 1001 个整数组成的数组，这些整数是任意排列的，但是你知道所有的整数都在 1 到 1000（包括 1000）之间。此外，除一个数字出现两次外，其他所有数字只出现一次。假设你只能对这个数组做一次处理，用一种算法找出重复的那个数字。如果你在运算中使用了辅助的存储方式，那么你能找到不用这种方式的算法吗？

解法一

求和整个数组，然后减去 1～1000 的和，就可以得到这个数。然而直接这样做容易越界，所以我们可以边求和边减：

```
int v = 0;
for (int i = 0; i < array.Count; i++)
    v += array[i] - i;
return v;
```

其中：

array 是给定数组

解法二

这类问题大家一定不会忘记经典的异或了：

```
int v = 0;
for (int i = 0; i < array.Count; i++)
        v ^= array[i] ^ i;
return v;
```

解法三

C# 3.0 起提供了 LINQ，慢慢开始改变传统数据处理的线性思维，这是语言发展越来越趋向于人的例子。LINQ 的基础之一是扩展方法，我们也可以直接使用这些扩展方法，将前面的解法一、解法二分别写成：

1. array.Select((i, j) => i - j).Sum();
2. array.Select((i, j) => i ^ j).Aggregate((i, j) => i ^ j);

讨论

如前文序言所述，信息时代，“逻辑密集型”的计算需求将会越来越多，好的语言不但能够给我们提供强大的工具，并且会慢慢改变人们思考求解问题的方式。

3.4 区间重合判断

问题

给定一个源区间[x,y]($x \leqslant y$)和 N 个无序目标区间$[x_1,y_1]$ $[x_2,y_2]$ $[x_3,y_3]$...，判断源区间是不是包含在目标区间的并集内？

例如：区间[1,6]是否包含在[2,3][1,2][3,9]内？

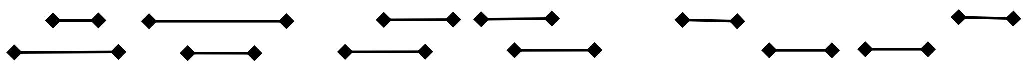
解法

《编程之美》中给出了 O(n*logn + n+ k*logn)的解法。然而和源区间不相关的目标区间其实不需要去排序或者合并，排除这些区间，并在过程中缩小源区间，我们能将复杂度降低到<O(nlog)。首先我们定义一个区间类：

```
class Section
{
    public int lo; //下界
    public int hi; //上界

    public static Section operator *(Section s1, Section s2)
    {
        return new Section { lo = Math.Max(s1.lo, s2.lo), hi = Math.Min(s1.hi, s2.hi) };
    }
}
```

其中，我们重载了操作符*，意义为区间求交集。

我们来考虑两个区间相减会出现的情况：



如图，在上面去区间为 s1，下面的区间为 s2，那么两个区间有“包含”、“相交”、“不相交”总共 6 种情况。那么：

若s1 ⊆ s2，则 s1-s2=空集；

若s2 ⊆ s1，则 s1-s2=两个区间，相减失败

若 s1，s2 相交，则 s1-s2 = s1 排除交集部分

若 s1，s2 不相交，则不变化。

由此，我们可以定义区间的减法：

```
public static Section operator -(Section s1, Section s2)
{
    var i = s1 * s2;

    if (i == s1) return Section.Empty;//s1 被 s2 包含，相减为空集
```

```
        else if (i == s2)//s1 包含 s2，减成两个区间
        {
            return null;//相减失败
        }
        else if (i.Length > 0)//交集非空，证明 s1，s2 相交
        {
            var s3 = new Section(s1);
            if (s1.hi >= s2.hi) s3.lo = s2.hi;
            else s3.hi = s2.lo;
            return s3;
        }
        return s1;//不相交，相减不变
    }
```

再定义一个区间的加法运算，意义为求并集：

```
    public static Section operator +(Section s1, Section s2)
    {
        var i = s1 * s2;
        if (i.Length < 0) return null;//交集为空 无法合并
        var u = new Section();
        u.lo = Math.Min(s1.lo, s2.lo);
        u.hi = Math.Max(s1.hi, s2.hi);
        return u;
    }
```

打好基础后，我们来看算法：依次拿出一个目标区间，用源区间 source 减去它，如果 source 被“减光”，则返回 true。

如果相减失败(源区间会被减成两个区间)，则加入排序二叉堆，按照区间下届保持有序，以后处理。

```
    bool IsContained_Subtract(Section source, List<Section> list)
    {
        var leftover = new BinHeap<Section>()[6];
        foreach (var t in list)
        {
            var d = source - t;
            if (source == d) continue;//相减不变（不相交的情况）
            if (d == null) leftover.Push(t);//相减失败，留作以后处理
            else if (d.Length < 0) return true;//源区间被减空，返回 true
            else source = d;//相减成功
```

[6] BinHeap<T>为自行开发辅助类库中的泛型二叉堆，请参见 4.1 节：动态有序集合。

```
        }
        return IsContained_Combine(source, leftover);//处理遗留的目标区间
    }
```

其中：

source 为源区间，list 为目标区间的列表。接下来处理相减失败的情况：

```
bool IsContained_Combine(Section source, BinHeap<Section> secs)
{
    Section last = null;
    var cnt = secs.Count;
    for (int i = 0; i < cnt; i++)
    {
        var t = secs.Pop();//获取当前目标区间
        if (last != null)//合并上次剩下的区间
        {
            var u = t + last; //求并集
            if (u != null) t = u;
            last = null;
        }
        var r = source * t;
        if (r == source) return true;
        else if (r.Length > 0) last = t;//源区间和当前目标区间有交集，留作下次合并
    }
    return false;
}
```

讨论

本节提出的解法并没有直接对目标区间进行排序、合并，而是在这两个环节中积极筛选和排除，从而提高了效率，复杂度<O(nlog)。

对于《编程之美》中提到的二维情况，可以采用“相减”的思路。

3.5 天平称球

问题

12 个球一个天平，现知道只有一个和其它的重量不同，问怎样称才能用三次就找到那个球。13 个呢？（注意此题并未说明那个球的重量是轻是重，所以需要仔细考虑）

分析

初次遇到这种问题，容易分析晕，如果能够找到一种思维工具，解决起来会清晰很多。

让我们从“信息”的角度来对这个问题进行分析：将球分成 4 种：“黑球”、“白球”、“轻球”、“重球”：

1. 在称之前，所有球都不知道是否有问题，都是“黑球”（用“*”表示）；
2. 排除嫌疑的球是“白球”（用“-”表示）。将“黑球”变白需要获取“信息”，方式为“用天平称”；
3. 天平称了以后，被怀疑偏轻或重的球是“轻球”（用“↑”表示）或“重球”（用“↓”表示），我们要做的事情就是让“轻球”再次被怀疑“偏重”，则它是白球，反之亦然。这也很好证明：举例说明，若 x≥8 且 x≤8，则 x＝8。
4. 用“V”表示天平，例如：** V *- 表示天平左边两黑球，右边一白一黑。

求解

有了思维工具了，求解就方便了：

1. 初始时候我们有 12 个黑球，拿出 8 个来称：**** V ****
   a) 若平，8 个球变白。还剩 4 个黑球，则称 ** V *-
      i. 若平，则 3 个黑球变白，剩下一个黑球，称* V -，即得出结果。
      ii. 若左轻，即：↑↑ V ↓ - ，那么再称 ↑ V ↑，即得结果。

右轻情况以此类推。

b) 若左轻，即：↑↑↑↑ V ↓↓↓↓，那么再称↑↑↓ V↑↓-

i. 若平，剩下↓↓↑，再称 ↓V↓，即得结果。

ii. 若左轻，剩下↑↑↓，再称 ↑V↑，即得结果。

iii. 若右轻，剩下↑↓，再称 ↑V-，即得结果。

c) 若右轻，以此类推。

讨论

在我高中的时候，同桌非常聪明，做题画的乱七八糟的居然都能算对，很多同学思维能力强，记忆力好，对这种作法引以为荣。

然而在现实生活中，个人英雄是不行的，需要团队的合作。如何找到“思维工具”，将问题清晰的表述出来，比去解决一个具体问题更为重要。有了工具大家都能用，若所有人都提供很多工具，工具之上又可构建更强工具，那么“巨人肩膀上的巨人”就出现了。

3.6 猜数字

问题

有三个六位数，分别是 ABCDEF、CDEFAB、EFABCD。

A、B、C、D、E、F 分别代表一位数，可能是 1~9 之间的任何一个，但是他们都是不同的数。

已知这三个六位数满足下列条件：

ABCDEF*2=CDEFAB　　CDEFAB *2=EFABCD

问 A=？、B=？、C=？、D=？、E=？、F=？

分析

这是微软一道笔试题，在时间压力下，没有工具“硬推”，不但容易出错，而且弄晕头脑还影响后面的题目。

首先我们来看，既然这个数字每次乘 2 都循环左移 2 位，那么 2 位是一直在一起的，可以看作一个数字，令 AB = X, CD = Y, EF = Z；则 100 进制数满足：

1. 2XYZ = YZX
2. 2YZX = ZXY

现在最麻烦的事情就是不知道相乘以后是否进位，那么何不将其全部列出？

求解

根据进位与否，式 1 可得 4 种可能性：

1.	2Z=X	2Y = Z	2X = Y
2.	2Z=X	2Y=Z+100	2X+1=Y
3.	2Z=X+100	2Y+1=Z	2X = Y
4.	2Z=X+100	2Y+1=Z+100	2X+1=Y

式 2 可得 4 种可能性：

1.	2X=Y	2Z = X	2Y = Z
2.	2X=Y	2Z=X+100	2Y+1=Z
3.	2X=Y+100	2Z+1=X	2Y = Z
4.	2X=Y+100	2Z+1=X+100	2Y+1=Z

那么就只可能：2X=Y　　2Z=X+100　　2Y+1=Z

联立求解得到：X=14　Y=28　Z=57

讨论

这道题目再次说明了，找到方法比找到答案更加重要。

4 结构之法

4.1　动态有序集合

问题

在离散型优化问题（如路径搜索、整数规划等）中，常常需要一种能够高效管理动态有序集合的数据结构：不断加入无序的数据，实时取出有序数据（如从优到劣），并且可能频繁地需要进行筛选、删除、更新（改变元素的 key）、查找等操作。

本节将介绍.Net 提供的 2 个数据结构，和自行设计的 2 个数据结构，并设计实验对比它们在不同情况下的性能。

首先定义一个接口 IDySet<T>，便于利用统一的测试类进行测试，如左图所示，函数解释如下：

1. Push：放入一个元素；
2. Pop：取出集合中“最小”元素；
3. 其它略

排序顺序表（Sorted List）

.Net 类库中，System.Collections.Generic 名称空间下提供了，SortedList，其原理是在需要插入新元素的时候，利用二分查找找到应该插入的位置。

然而其顺序的存储结构，使得插入和删除操作非常昂贵。所以称其为“静态有序集合”更加合理——适用于一次性填充，计算过程中

少有改变的情况，它的优点是提供了 O(1)的下标访问。

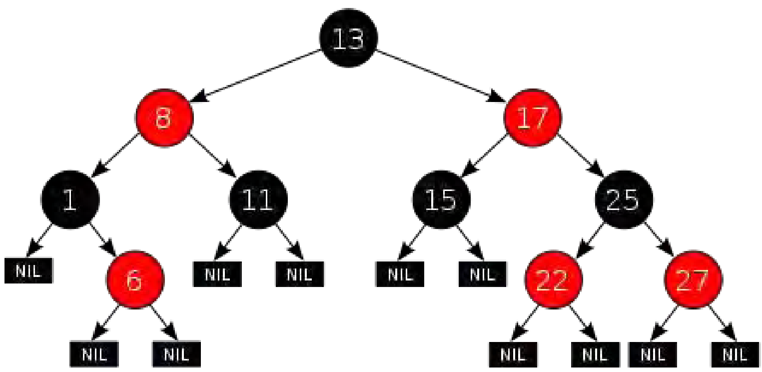
为了便于对比测试，对它进行改造，Push 时将元素从大到小排序，Pop 的时候将末尾元素删除并返回。

红黑树（Red-black Tree）

“二分查找”是诱人的，然而顺序存储不利于集合修改。人们自然想到了用链式存储——二叉排序树(BST)，遗憾的是，链式结构容易“退化”，为了解决这个问题，有了依靠“旋转节点”的平衡二叉树（AVL）。然而由于 AVL 要求严格平衡（左右子树深度差不超过 1），后来又出现了“红黑树”，只要求大致平衡，性能较好，得到了广泛的应用。

红黑树满足以下性质：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是 NIL 节点）。
4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



同样的，System.Collections.Generic 名称空间下提供了 SortedDictionary，其中 TreeSet<T>类型的数据成员实现了红黑树。

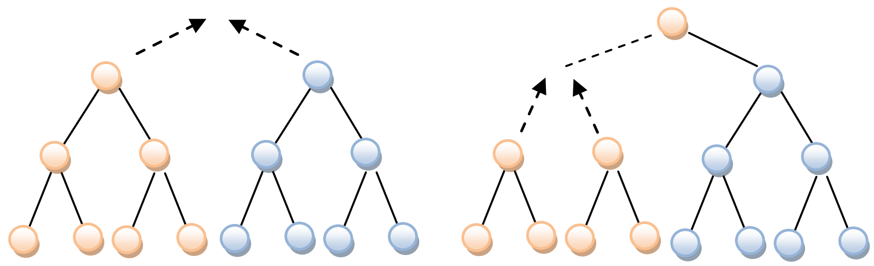
为了便于对比测试，在 TreeSet 中增加 Pop 操作：找到树中最“小”

节点（最左子树的左孩子），删除节点并返回。

链式二叉堆（Linked Binary Heap）

二叉堆想必大家都不陌生，然而常见的都是顺序存储的。是否可以定义链式二叉堆 LBH 呢？首先分别来看几个操作：

1. Push：设有待插入节点 k，准备加入某 LBH。若根其空，直接做根，否则看 k 是否优于根，优于则交换；若有根有一个空孩子，则填其位；若根为叶子，则随机做其左右；若根为二度点，则随机插入左右子树。
   从此操作可以看成，LBH 是自动保持平衡的，无需额外考虑。
2. Pop：取根节点（它是最小点）。这时我们得到了两棵 LBH，选择根小的作为新根，这时又递归出现同样问题，如图：



3. Contains：可以在 LBH 内部维护一个哈希表来实现：Dictionary<T, Node>，T 为泛型参数，Node 为内部使用节点。

LBH 的优点是计算过程只修改节点指针，不需要分配节点内存，在给定元素的搜索问题中比较适合，例如：在地图上搜索路径，所有节点是一直存在的。

二叉堆（Binary Heap）

二叉堆是顺序存储的完全二叉树，顺序存储结构会涉及到空间扩充时候的拷贝浪费，那么相比链式结构，其效率如何呢？实验之前，没想它是冠军...

首先，我们利用列表 item（List<T>）来保存元素，哈希表 dic（Dictionary<T, int>）来做索引，其 value 保存 T 在 item 中的位置。并设置一个私有变量 hashing（通过构造函数赋值），表示是否建立索

引。

1. Push：主要思想为，提拔优秀的节点

```
public void Push(T value)
{
    if (value == null) return;
    if (hashing && ContainsKey(value))
        throw new Exception("Adding Duplicate!");
    items.Add(value);//将元素添加到数组尾
    if (hashing) dic.Add(value, Count - 1);
    TryPromote(Count);
}
```

其中：

```
void TryPromote(int i)// 将较优的节点“往上升职”
{
    if (i == 1) return;//如果只有一个元素，不需要提升
    int f = i / 2;
    var temp = items[i - 1];
    while (Compare(items[f - 1], temp) > 0) //如果父节点劣
    {
        SetPosition(i - 1, items[f - 1]);
        i = f;   f = i / 2;
        if (f == 0) break;
    }
    SetPosition(i - 1, temp);
}
void SetPosition(int i, T value)//更新值和索引
{
    items[i] = value;
    if (hashing) dic[items[i]] = i;
}
```

2. Pop：主要思想为，降级劣质节点。

```
public T Pop()
{
    if (Count == 0) throw new Exception("Heap is empty!");
    var top = items[0];
    RemoveAt(0);
    return top;
}
```

其中：

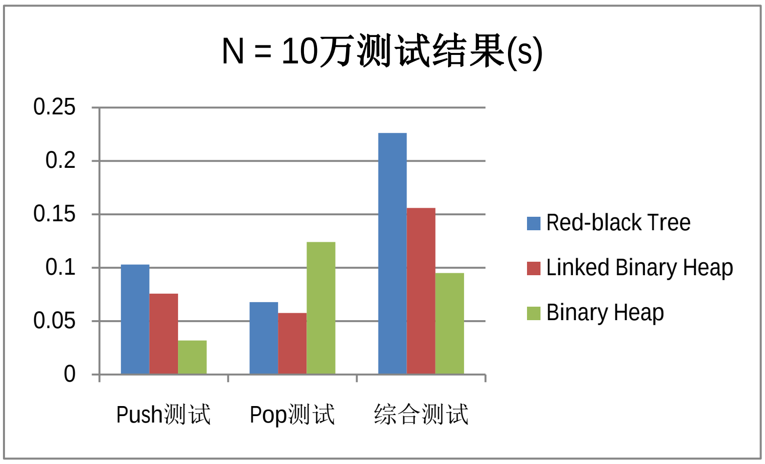
```
public void RemoveAt(int i)//移除节点
{
    if (i < 0 || i >= Count) return;
```

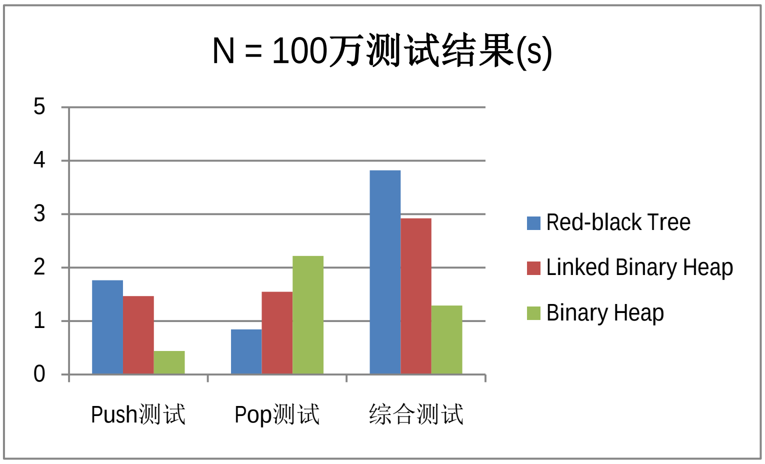
```
        if (hashing) dic.Remove(items[i]);
        if (i < Count - 1) items[i] = items[Count - 1];//把最后一个元素提拔来
        items.RemoveAt(Count - 1);
        if (Count > 0 && i < Count)
        if (!TryDegrade(i + 1)) TryPromote(i + 1);//考虑提拔或者降级
    }

    bool TryDegrade(int i)//降级劣质节点
    {
        bool degraded = false;
        var temp = items[i - 1];
        for (int s = i * 2; s <= Count; s *= 2)
        {
          if (s < Count && Compare(items[s - 1], items[s]) > 0) s++;
          //查看右孩子是否更优
          if (Compare(temp, items[s - 1]) < 0) break;//优于子孙，不需要降级
          SetPosition(i - 1, items[s - 1]);
          i = s;
          degraded = true;
        }
        SetPosition(i - 1, temp);
        return degraded;
     }
```

性能测试

为了对比这些数据结构在不同情况下的性能，设计测试类（针对 IDySet<T>接口），重复做 20 次如下实验（20 次结果已趋于稳定），计算总时间（每次计时前垃圾回收），最后取 20 次平均值：

1. Push 测试：随机生成 N 个 0～n 的整数 r，若 Contains(r) = false 则 Push(r)。
2. Pop 测试：将已加入到集合中的元素全部 Pop 出来。
3. 综合测试：随机生成 N 个 0～n 的整数 r，若 Contains(r) = true 则 Remove(r)；Push(r)；若 r 的序号能整除 8，则 Pop()。





**注: Sorted List 拥有最快的 Pop 速度，然而其它性能差距却 10 倍之多，故不再列出。*

讨论

在解决很多离散型优化问题（如路径搜索、整数规划等）时，可以采用启发是搜索算法（通过估计避免大量不必要的搜索），其中需要要用动态有序集合：

每次迭代 Pop 出最优节点，进行某些处理后（可能涉及查找、删除操作），再发展 n 个节点 Push 入集合中（如：路径搜索中的 A*算法每次扩展周围 8 个格子），找到结果后返回（可参见 4.3 “一摞饼的排序”）。

由此可以看出在求解过程中，Push 需求远大于 Pop，因此 Binary Heap 获得冠军!。自行设计的 Linked Binary Heap 和 Binary Heap 结

构都比较简单，而其性能却都超越了号称“最复杂数据结构”之一的红黑树。

4.2 寻找最大的 k 个数

问题

搜索引擎需要在大量网页中找到相关性最高的前 k 个返回给用户，并且网页的权重可能会实时更新。如何解决这个问题?

分析

通过 4.1 节实验分析可以看出，自行设计的 BinHeap 拥有极高的插入速度。并且，如果网页权重更新，只需要将其“升级”或者“降级”即可。

解法

在 Binary Heap 中增加一个 Push 重载，即可得到最大的 k 个数。

```
public void Push(T value, int countLimit)
{
    Push(value);
    if (Count > countLimit) Pop();
}
```

讨论

动态有序集合应用非常广泛，以后我们还将看到更多例子。

4.3 一摞饼的排序

问题

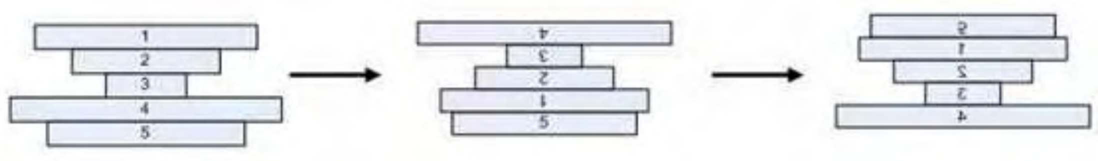
星期五的晚上，一帮同事在希格玛大厦附近的"硬盘酒吧"多喝了几杯。程序员多喝了几杯之后谈什么呢？自然是算法问题。有个同事

说：

"我以前在餐馆打工，顾客经常点非常多的烙饼。店里的饼大小不一，我习惯在到达顾客饭桌前，把一摞饼按照大小次序摆好--小的在上面，大的在下面。由于我一只手托着盘子，只好用另一只手，一次抓住最上面的几块饼，把它们上下颠倒个个儿，反复几次之后，这摞烙饼就排好序了。

我后来想，这实际上是个有趣的排序问题：假设有 n 块大小不一的烙饼，那最少要翻几次，才能达到最后大小有序的结果呢？

你能否写出一个程序，对于 n 块大小不一的烙饼，输出最优化的翻饼过程呢？



如图所示，两次翻转可以将最大的饼弄到最下面去。

分析

此问题是 4.1 节中提到的“离散型优化问题”，我们可以采用启发式搜索来解决。

启发式搜索

搜索部分代码如下：

```
node = new Anode(array);
open.Push(node);
while (open.Count > 0)
{
    node = open.Pop();//取出当前最优节点
    if (node.H == 0) break;//找到解
    if (close.Contains(node)) continue;//排除重复
```

```
        for (int i = 1; i < node.ItemCount; i++)
        {
            var child = new ANode(node, i);//发展后代
            if (!Exists(child)) open.Push(child);
        }
        close.TryAdd(node);
    }
```

其中:

open 类型是 BinHeap<ANode>，前文提到的二叉堆；

ANode 类用来表示解，其中一个数组表示饼的顺序，一个整形变量保存了是如何从“父节点”翻转来的；

ANode 中 2 个变量 G，H 体现启发搜索的重要信息：

启发函数 F = G （实际翻转次数）+ H（估计还需翻转次数）

open 中，节点按照 F 保持有序。

close 类型是 Dictionary<ANode>用来保存已经找到过的状态，避免重复。

```
bool Exists(ANode node) //用来检查当前发展的后代：
{
    if (close.ContainsKey(node)) return true;//是否已经有过
    int i = open.IndexOf(node);
    if (i >= 0)
    {
        var old = open[i];
        if (old.G > node.G)//是否有更好的翻转办法到达这个状态
        {
            old.Renew(node);
            open.TryPromote(i + 1);//更新节点位置
        }
        return true;
    }
    return false;
}
```

实验

我们来看《编程之美》提到的例子：3, 2, 1, 6, 5, 4, 9, 8, 7, 0

```
1:   3,2,1,6,5,4,9,8,7,0    Turn Over at: 4    Heuristic = 3
2:   5,6,1,2,3,4,9,8,7,0    Turn Over at: 8    Heuristic = 3
3:   7,8,9,4,3,2,1,6,5,0    Turn Over at: 6    Heuristic = 3
4:   1,2,3,4,9,8,7,6,5,0    Turn Over at: 8    Heuristic = 2
5:   5,6,7,8,9,4,3,2,1,0    Turn Over at: 4    Heuristic = 1
6:   9,8,7,6,5,4,3,2,1,0    Turn Over at: 9    Heuristic = 1
Result: 0,1,2,3,4,5,6,7,8,9
```

程序总共只花费 42 次迭代找到了最优解，相比 172126 次有不小的改进，由此说明，启发函数估计越准确，求解速度越快。

对于比较长的如：6, 15, 3, 9, 0, 1, 7, 5, 11,16, 2, 12, 8, 13, 4, 10, 14 也能在几秒得到最优解：

9,7,13,15,2,11,4,9,7,16,14,6,10,5,11,12,8

有关启发函数

如前面实验分析，启发函数是否估计得好，对求解速度有着至关重要的影响。我们先来看 2 个启发函数：

H1：主要反映数组中相邻数字是否相差超过 1，每次翻转可以让应该相邻的两个数组挨在一起，因此拿它做估计是比较合理的，可以保证 F 值≤真实翻转次数，从而保证最优解。

```
int iv = 0;
for (int i = 1; i < mArray.Length; i++)
    if (Math.Abs(mArray[i] - mArray[i - 1]) > 1) iv++;
if (iv == 0 && mArray[0] != 0) iv++;
return iv;
```

H2：在“线性代数”中我们曾学到过排列的“逆序数”，反映了它与标准排列的差异，因此我们可以这样来估计：

```
int iv1 = 0, iv2 = 0;
for (int i = 0; i < mArray.Length; i++)
    for (int j = 0; j < mArray.Length; j++)
    {
        if (i == j) continue;
        if (mArray[j] < mArray[i])
        {
            if (j < i) iv1++;
            else iv2++;
        }
    }
return Math.Min(iv1, iv2 + 1);
```

有了两个启发函数，可以做实验了，我们先在节点的比较函数里面写到：

```
int order = F.CompareTo(other.F); //F = G + H1
return order;
```

运行程序，测试 n=7 时的全排列，总共花费 345 秒。

我们再对比较函数稍做改动：

```
int order = F.CompareTo(other.F); //F = G + H1
if (order == 0) order = F2.CompareTo(other.F1); //F = G + H2
return order;
```

再运行程序，此时总共花费时间为 151 秒。当然，如果我们还能够找到更好的启发函数，或许还可以提高求解速度。

任意顺序最小翻转次数

这个问题未能解决，这里简单讨论下。对于任意顺序的最小翻转次数问题，估计不能通过测试全排列来搞定 15 个以上的饼。

我们先将这些“最难排”的顺序列出来看看是否有规律可循：

N = 1	2	3	4	5	6	7	8
0	1,0	0,2,1	1,3,0,2 2,0,3,1 3,1,2,0	0,3,2,4,1 1,3,0,4,2 2,0,4,3,1 3,1,4,0,2 4,1,3,0,2	4,2,5,0,3,1 3,5,1,4,0,2	0,4,1,6,3,5,2 1,5,3,6,0,4,2 2,5,0,3,6,1,4 3,1,5,2,6,0,4 4,1,3,6,0,5,2 5,1,3,0,6,2,4 6,2,0,4,1,5,3 ...	0,3,6,2,5,7,1,4 1,3,6,0,7,4,2,5 2,6,1,4,7,5,0,3 3,1,5,7,2,6,4,0 4,1,6,2,7,0,5,3 5,1,3,7,2,4,0,6 6,1,3,7,0,4,2,5 7,1,3,5,2,6,4,0 ...

可以大致看出有几个规律（特别是数字比较大了以后）：

1. 相邻数字一般差在 1 以上；
2. 大多按照减、增、减...的方式交替出现；

因此，可以依照这两个规律，随机产生大量排列去测试，得到了 n =14 的时候，最少需要翻转 15 次的一些排列：

0,5,3,9,1,11,7,13,10,12,4,8,2,6
11,13,6,9,5,8,0,2,7,3,12,1,4,10
7,1,9,5,12,4,13,10,3,8,2,11,6,0
2,12,3,8,5,9,1,13,11,7,10,0,4,6
1,5,3,8,0,13,10,12,2,6,4,7,11,9
0,2,9,5,12,10,13,4,7,1,8,3,6,11
10,2,4,1,12,6,11,8,13,9,5,7,0,3
4,1,12,2,9,6,13,8,10,3,5,0,11,7

然而其规律仍然难以总结，令人比较费解的是 n=6 时为何最难排序的只有 2 个。您能否找到其规律呢？

4.4 离散优化问题搜索框架

启发式搜索算法 A*和遗传算法 GA 分别是精确搜索和近似搜索，两者原理完全不同，但是仔细研究可发现，它们却又在数据结构上非常相似，几乎可以一一对应。

前者原理是在一棵搜索树上，通过估价函数“直捣黄龙”，并尽量“砍枝”；后者原理是“养活”一群物种，通过适应度函数“物竞天择”，尽量保持多样化，让它们交叉，子代尽量继承父代优良基因，并通过变异来跳出局部最优。

表 4-1 两种搜索算法的比较

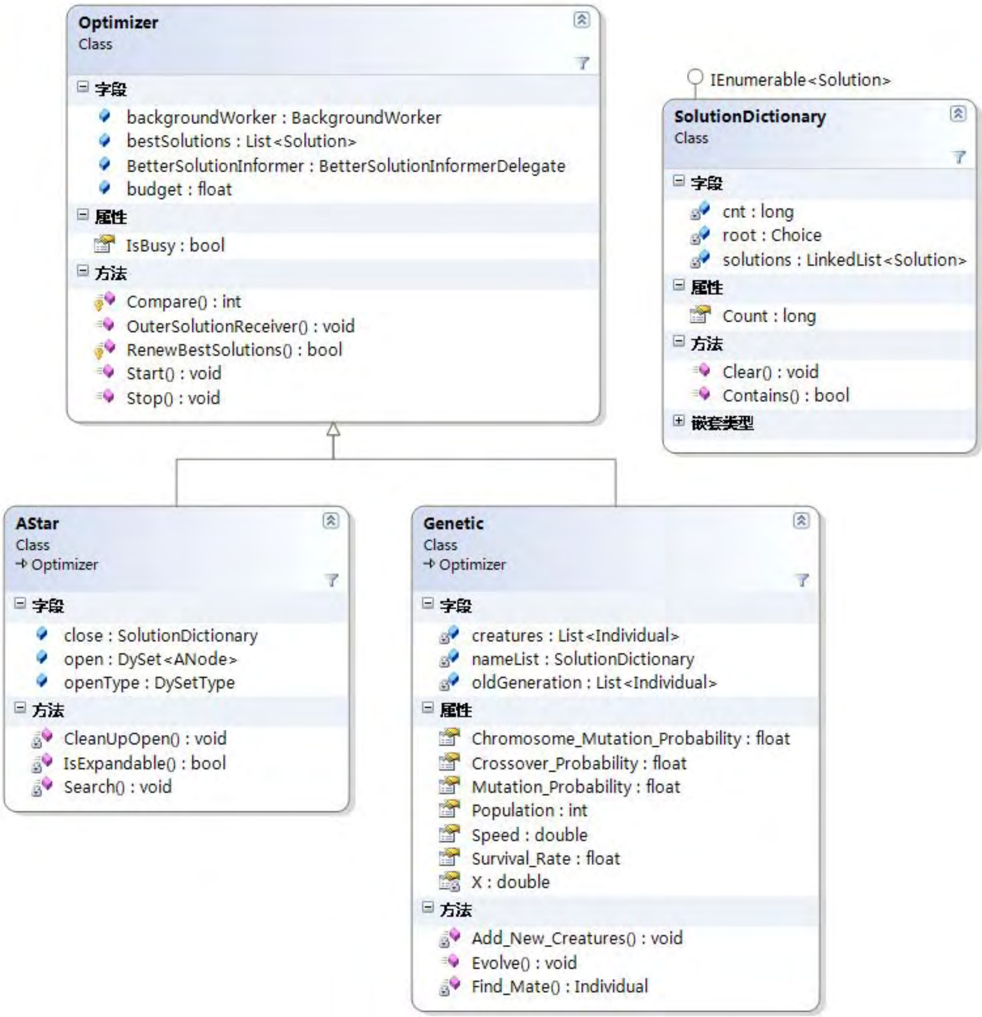
	A*算法	遗传算法	数据结构
求解方式	搜索	进化	都可放入父类 Background Worker 运行
工作环境	Open 表	种群	可采用动态有序集合
排除重复	Close 表	花名册	可采用哈希表，或树
优劣评估	启发函数	适应度函数	可采用同一种评估方式
解的表示	树节点 ANode	个体 Individual	都继承于 Solution 类
求解过程	当前节点发展子代	交叉，变异	自行扩展差异

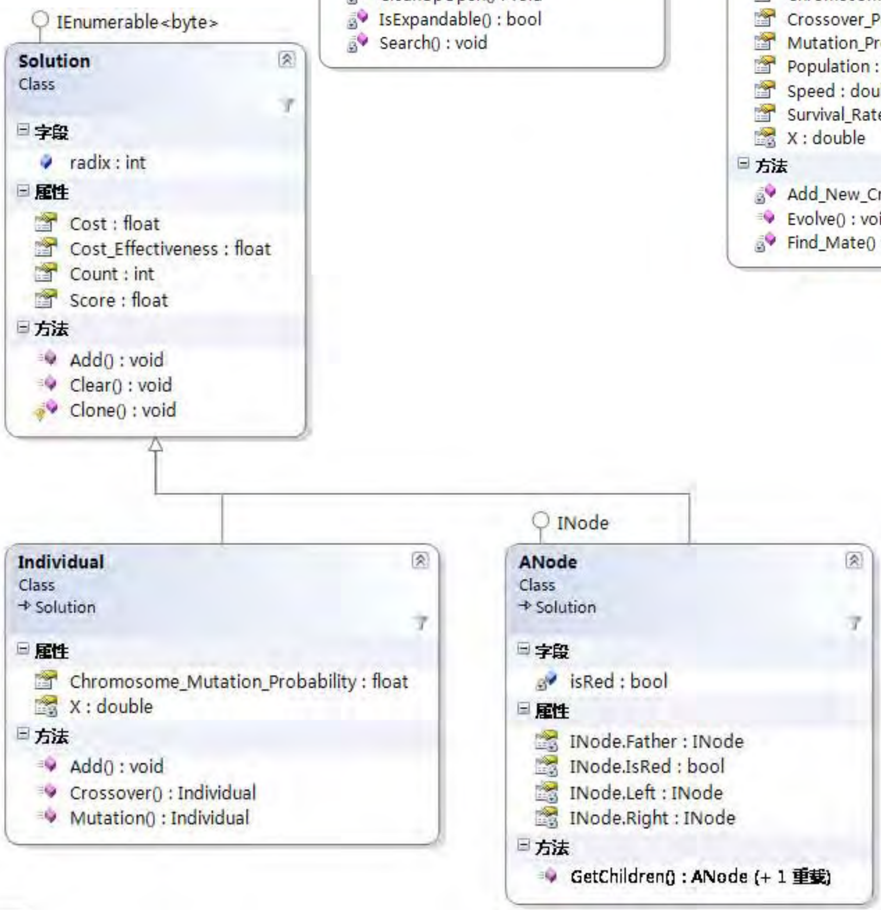
若将两个算法并行求解，找到好的解通知对方：GA 可以用来繁殖出更好的解；A*可以提高“砍枝条件”。GA 虽然是近似求解，但是拥有的变异机制常常可以跳出局部最优，两种算法结合可以优势互补。

在实践中发现，很多问题都可以采用这个框架去求解。

如果一个搜索问题的解空间可以有办法列举，那么就可以采用 A*。

如果解空间过于庞大，可以联合两种算法并行求解。





然而也存在某些问题的解空间难以列举的情况，例如 VRP(Vehicle Routing Problem)等 NP 问题。此时就可以用 GA 单独求解，但不能保证全局最优。