1 的数目

给定一个十进制正整数 *N* , 写下从 1 开始 , 到 *N* 的所有整数 , 然后数一下其中出现的所有“1”的个数。

例如 :

N= 2 , 写下 1 , 2。这样只出现了 1 个“1”。

N= 12 , 我们会写下 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12。这样 , 1 的个数是 5。

问题是 :

1. 写一个函数*f*(*N*) ,返回1到*N*之间出现的“1”的个数,比如*f*(12) =5。
2. 在32位整数范围内 ,满足条件“*f*(*N*)= *N*”的最大的*N*是多少 ?

分析与解法

【问题 1 的解法一】

这个问题看上去并不是一个困难的问题，因为不需要太多的思考，我想大家都能找到一个最简单的方法来计算 $f(N)$，那就是从 1 开始遍历到 N，将其中每一个数中含有“1”的个数加起来，自然就得到了从 1 到 N 所有“1”的个数的和。写成程序如下：

代码清单 2-9

```
ULONGLONG Count1InAInteger(ULONGLONG n)
{
   ULONGLONG iNum = 0;
   while(n != 0)
   {
      iNum += (n % 10 == 1) ? 1 : 0;
      n /= 10;
   }

   return iNum;
   }

ULONGLONG f(ULONGLONG n)
{
   ULONGLONG iCount = 0;
   for (ULONGLONG i = 1; i <= n; i++)
   {
      iCount += Count1InAInteger(i);
   }
```

```
    return iCount;
}
```

这个方法很简单，只要学过一点编程知识的人都能想到，实现也很简单，容易理解。但是这个算法的致命问题是效率，它的时间复杂度是

$O(N)$×计算一个整数数字里面“1”的个数的复杂度 $= O(N * \log_2 N)$

如果给定的 N 比较大，则需要很长的运算时间才能得到计算结果。比如在笔者的机器上，如果给定 N=100 000 000，则算出 $f(N)$ 大概需要 40 秒的时间，计算时间会随着 N 的增大而线性增长。

看起来要计算从 1 到 N 的数字中所有 1 的和，至少也得遍历 1 到 N 之间所有的数字才能得到。那么能不能找到快一点的方法来解决这个问题呢？要提高效率，必须摈弃这种遍历 1 到 N 所有数字来计算 $f(N)$ 的方法，而应采用另外的思路来解决这个问题。

【问题 1 的解法二】

仔细分析这个问题，给定了 N，似乎就可以通过分析“小于 N 的数在每一位上可能出现 1 的次数”之和来得到这个结果。让我们来分析一下对于一个特定的 N，如何得到一个规律来分析在每一位上所有出现 1 的可能性，并求和得到最后的 $f(N)$。

先从一些简单的情况开始观察，看看能不能总结出什么规律。

先看 1 位数的情况。

如果 $N = 3$，那么从 1 到 3 的所有数字：1、2、3，只有个位数字上可能出现 1，而且只出现 1 次，进一步可以发现如果 N 是个位数，如果 $N>=1$，那么 $f(N)$ 都等于 1，如果 $N=0$，则 $f(N)$ 为 0。

再看 2 位数的情况。

如果 $N=13$，那么从 1 到 13 的所有数字：1、2、3、4、5、6、7、8、9、10、11、12、13，个位和十位的数字上都可能有 1，我们可以将它们分开来考虑，个位出现 1 的次数有两次：1 和 11，十位出现 1 的次数有 4 次：10、11、12 和 13，所以 $f(N)=2+4=6$。要注意的是 11 这个数字在十位和个位都出现了 1，但是 11 恰好在个位为 1 和十位为 1 中被计算了两次，所以不用特殊处理，是对的。再考虑 $N=23$ 的情况，它和 $N=13$ 有点不同，十位出现 1 的次数为 10 次，从 10 到 19，个位出现 1 的次数为 1、11 和 21，所以 $f(N)=3+10=13$。通过对两位数进行分析，我们发现，个位数出现 1 的次数不仅和个位数字有关，还和十位数有关：如果 N 的个位数大于等于 1，则个位出现 1 的次数为十位数的数字加 1；如果 N 的个位数为 0，则个位出现 1 的次数等于十位数的数字。而十位数上出现 1 的次数不仅和十位数有关，还和个位数有关：如果十位数字等于 1，则十位数上出现 1 的次数为个位数的数字加 1；如果十位数大于 1，则十位数上出现 1 的次数为 10。

```
f(13) = 个位出现1的个数 + 十位出现1的个数 = 2 + 4 = 6;
f(23) = 个位出现1的个数 + 十位出现1的个数 = 3 + 10 = 13;
f(33) = 个位出现1的个数 + 十位出现1的个数 = 4 + 10 = 14;
…
f(93) = 个位出现1的个数 + 十位出现1的个数 = 10 + 10 = 20;
```

接着分析 3 位数。

如果 $N = 123$：

个位出现 1 的个数为 13：1, 11, 21, …, 91, 101, 111, 121

十位出现 1 的个数为 20：10～19, 110～119

百位出现 1 的个数为 24：100～123

$f(23)$ = 个位出现 1 的个数 + 十位出现 1 的个数 + 百位出现 1 的次数 = 13 + 20 + 24 = 57；

同理我们可以再分析 4 位数、5 位数。读者朋友们可以写一写，总结一下各种情况有什么不同。

根据上面的一些尝试，下面我们推导出一般情况下，从 N 得到 $f(N)$ 的计算方法：

假设 $N=abcde$，这里 a、b、c、d、e 分别是十进制数 N 的各个数位上的数字。如果要计算百位上出现 1 的次数，它将会受到三个因素的影响：百位上的数字，百位以下（低位）的数字，百位（更高位）以上的数字。

如果百位上的数字为 0，则可以知道，百位上可能出现 1 的次数由更高位决定，比如 12 013，则可以知道百位出现 1 的情况可能是 100～199，1 100～1 199，2 100～2 199，…，11 100~11 199，一共有 1 200 个。也就是由更高位数字（12）决定，并且等于更高位数字（12）×当前位数（100）。

如果百位上的数字为 1，则可以知道，百位上可能出现 1 的次数不仅受更高位影响，还受低位影响，也就是由更高位和低位共同决定。例如对于 12 113，受更高位影响，百位出现 1 的情况是 100～199，1 100～1 199，2 100～2 199，…，11 100~11 199，一共 1 200 个，和上面第一种情况一样，等于更高位数字（12）×当前位数（100）。但是它还受低位影响，百位出现 1 的情况是 12 100～12 113，一共 114 个，等于低位数字（123）+1。

如果百位上数字大于 1（即为 2~9），则百位上可能出现 1 的次数也仅由更高位决定，比如 12 213，则百位出现 1 的可能性为：100～199，1 100～1 199，2 100～2 199，…，11 100～11 199，12 100～12 199，一共有 1 300 个，并且等于更高位数字+1（12+1）×当前位数（100）。

通过上面的归纳和总结，我们可以写出如下的更高效算法来计算 $f(N)$：

代码清单 2-10

```
LONGLONG Sum1s(ULONGLONG n)
{
   ULONGLONG iCount = 0;

   ULONGLONG iFactor = 1;

   ULONGLONG iLowerNum = 0;
   ULONGLONG iCurrNum = 0;
```

```
   ULONGLONG iHigherNum = 0;

   while(n / iFactor != 0)
   {
      iLowerNum = n - (n / iFactor) * iFactor;
      iCurrNum = (n / iFactor) % 10;
      iHigherNum = n / (iFactor * 10);

      switch(iCurrNum)
      {
      case 0:
         iCount += iHigherNum * iFactor;
         break;
      case 1:
         iCount += iHigherNum * iFactor + iLowerNum
+ 1;
         break;
      default:
         iCount += (iHigherNum + 1) * iFactor;
         break;
      }

      iFactor *= 10;
   }

   return iCount;
}
```

这个方法只要分析 N 就可以得到 $f(N)$，避开了从 1 到 N 的遍历，输入长度为 Len 的数字 N 的时间复杂度为 $O(Len)$，即为 $O(\ln(n)/\ln(10)+1)$。在笔者的计算机上，计算 N=100 000 000，

相对于第一种方法的 40 秒时间，这种算法不到 1 毫秒就可以返回结果，速度至少提高了 40 000 倍。

【问题 2 的解法】

要确定最大的数 N，满足 $f(N)=N$。我们通过简单的分析可以知道（仿照上面给出的方法来分析）：

9 以下为：1 个；

99 以下为：1×10+10×1=20 个；

999 以下为：1×100+10×20=300 个；

9999 以下为：1×1000+10×300=4000 个；

…

999999999 以下为：900000000 个；

9999999999 以下为：10000000000 个。

容易从上面的式子归纳出：$f(10^{n-1}) = n * 10^{n-1}$。通过这个递推式，很容易看到，当 $n = 9$ 时候，$f(n)$ 的开始值大于 n，所以我们可以猜想，当 n 大于某一个数 N 时，$f(n)$ 会始终比 n 大，也就是说，最大满足条件在 0～N 之间，亦即 N 是最大满足条件 $f(n) = n$ 的一个上界。如果能估计出这个 N，那么只要让 n 从 N 往 0 递减，每个分别检查是否有 $f(n) = n$，第一个满足条件的数

就是我们要求的整数。

因此，问题转化为如何证明上界 N 确实存在，并估计出这个上界 N。

证明满足条件 $f(n) = n$ 的数存在一个上界

首先，用类似数学归纳法的思路来推理这个问题。很容易得到下面这些结论（读者朋友可以自己试着列举验证一下）：

当 n 增加 10 时，$f(n)$ 至少增加 1；

当 n 增加 100 时，$f(n)$ 至少增加 20；

当 n 增加 1 000 时，$f(n)$ 至少增加 300；

当 n 增加 10 000 时，$f(n)$ 至少增加 4 000；

……

当 n 增加 10^k 时，$f(n)$ 至少增加 $k*10^{k-1}$。

首先，当 $k>=10$ 时，$k*10^{k-1} > 10^k$，所以 $f(n)$ 的增加量大于 n 的增加量。

其次，$f(10^{10-1}) = 10^{10} > 10^{10-1}$。如果存在 N，当 $n = N$ 时，$f(N) - N > 10^{10-1}$ 成立时，此时不管 n 增加多少，$f(n)$ 的值将始终大于 n。

具体来说，设 n 的增加量为 m：当 m 小于 10^{10-1} 时，由于 $f(N) - N > 10^{10-1}$，因此有 $f(N+m) > f(N) > N + 10^{10-1} > N + m$，即 $f(n)$ 的值仍然比 n 的值大；当 m 大于等于 10^{10-1} 时，$f(n)$

的增量始终比 n 的增量大，即 $f(N+m)-f(N)>(N+m)-N$，也就是 $f(N+m)>f(N)+m>N+10^{10-1}+m>N+m$，即 $f(n)$ 的值仍然比 n 的值大。

因此，对于满足 $f(N)-N>10^{10-1}$ 成立的 N 一定是所求该数的一个上界。

求出上界 N

又由于 $f(10^{10-1})=n*10^{10-1}$，不妨设 $N=10^{K-1}$，有 $f(10^{K-1})-(10^{K-1})>10^{10-1}$，即 $K*10^{K-1}-(10^{K-1})>10^{10-1}$，易得 $K>=11$ 时候均满足。所以，当 $K=11$ 时，$N=10^{11-1}$ 即为最小一个上界。

计算这个最大数 n

令 $N=10^{11-1}$=99 999 999 999，让 n 从 N 往 0 递减，每个分别检查是否有 $f(n)=n$，第一满足条件的就是我们要求的整数。很容易解出 n = 1 111 111 110 是满足 $f(n)=n$ 的最大整数。

扩展问题

对于其他进制表达方式，也可以试一试，看看有什么规律。例如二进制：

$f(1)=1$

$f(10)=10$（因为 01, 10 有两个 1）

$f(11)=100$（因为 01, 10, 11 有四个 1）

读者朋友可以模仿我们的分析方法，给出相应的解答。

NIM“拈”游戏分析

问题

有 N 块石头和两个玩家 A 和 B，玩家 A 先将石头分成若干堆，然后按照 BABA……的顺序不断轮流取石头，能将剩下的石头一次取光的玩家获胜。每次取石头时，每个玩家只能从若干堆石头中任选一堆，取这一堆石头中任意数目（大于 1）个石头。

请问：玩家 A 有必胜策略吗？要怎么分配和取石头才能保证自己有把握取胜？

解法与分析

据说，该游戏起源于中国，英文名字叫做“NIM”，是由广东话“拈”（取物之意）音译而来，经由当年到美洲打工的华人流传出去，这个游戏一个常见的变种是将十二枚硬币分三列排成 [3，4，5] 再开始玩 。我们这里讨论的是一般意义上的“拈”游戏。

言归正传，在面试者咄咄逼人的目光下，你要如何着手解决这个问题？

在面试中，面试者考察的重点不是“what”——能否记住某道题目的解法，某件历史事件发生的确切年代，C++语言中关于类的继承的某个规则的分支等。面试者很想知道的是“how”——应聘者是如何思考和学习的。

所以，应聘者得展现自己的思路。解答这类问题应从最基本的特例开始分析。我们用 N 表示石头的堆数，M 表示总的石头数目。

当 N=1 时，即只有一堆石头——显然无论你放多少石头，你的对手都能一次全拿光，你不能这样摆。

当 N=2 时，即有两堆石头，最简单的情况是每堆石头中各有一个石子（1，1）——先让对手拿，无论怎样你都可以获胜。我们把这种在双方理性走法下，你一定能够赢的

局面叫作安全局面。

当 N = 2，M > 2 时，既然（1, 1）是安全局面，那么（1, X）都不是安全局面，因为对手只要经过一次转换，就能把（1, X）变成（1, 1），然后该你走，你就输了。既然（1, X）不安全，那么（2, 2）如何？经过分析，（2，2）是安全的，因为它不能一步变成（1，1）这样的安全局面。这样我们似乎可以推理（3, 3）、（4, 4），一直到（X, X）都是安全局面。

于是我们初步总结，如果石头的数目是偶数，就把它们分为两堆，每堆有同样多的数目。这样无论对手如何取，你只要保证你取之后是安全局面（X, X），你就能赢。

好，如果石头数目是奇数个呢？

当 M=3 的时候，有两种情况，（2, 1）、（1, 1, 1），这两种情况都会是先拿者赢。

当 M=5 的时候，和 M=3 类似。无论你怎么摆，都会是先拿者赢。

若 M=7 呢？情况多起来了，头有些晕了，好像也是先拿者赢。

我们在这里得到一个很重要的阶段性结论：

当摆放方法为（1, 1,…, 1）的时候，如果 1 的个数是奇数个，则先拿者赢；如果 1 的个数是偶数个，则先拿者必输。

当摆放方法为（1, 1,…, 1, X）（多个 1，加上一个大于 1 的 X）的时候，先拿者必赢。因为：

如果 1 有奇数个，先拿者可以从（X）这一堆中一次拿走 X-1 个，剩下偶数个 1——接下来动手的人必输。

如果有偶数个 1，加上一个 X，先拿者可以一次把 X 都拿光，剩下偶数个 1——接下来动手的人也必输。

当然，游戏是两个人玩的，还有其他的各种摆法，例如当 M = 9 的时候，我们可以摆为（2, 3, 4）、（1, 4, 4）、（1, 2, 6），等等，这么多堆石头，它们既互相独立，又互相牵制，那如何分析得出致胜策略呢？关键是找到在这一系列变化过程中有没有一个特性始终决定着输赢。这个时候，就得考验一下真功夫了，我们要想想大学一年级数理逻辑课上学的异或（XOR）运算。异或运算规则如下：

XOR（0, 0）= 0
XOR（1, 0）= 1
XOR（1, 1）= 0

首先我们看整个游戏过程，我们从N堆石头（$M_1, M_2, \cdots, M_n$）开始，双方斗智斗勇，石头一直递减到全部为零（0, 0,…, 0）。

当 M 为偶数的时候，我们的取胜策略是把 M 分成相同的两份，这样就能取胜。

开始：（M_1, M_1）　　它们异或的结果是XOR（M_1, M_1）= 0

中途：（M_1, M_2）　　对手无论怎样从这堆石头中取，XOR（M_1, M_2）!= 0

我方：（M_2, M_2）　　我方还是把两堆变相等。XOR（M_2, M_2）= 0

…

最后：（M_2, M_2）　　我方取胜

类似的，若M为奇数，我们把石头分成（1, 1, …,1）奇数堆的时候，XOR（1, 1,…,1）$_{[奇数个]}$!=0。而这时候，对方可以取走一整堆，XOR（1, 1,…, 1）$_{[偶数个]}$=0，如此下去，我方必输。

我们推广到 M 为奇数，但是每堆石头的数目不限于 1 的情况，看看 XOR 值的规律：

开始：（$M_1, M_2, \cdots, M_n$）　　XOR（$M_1, M_2, \ldots M_n$）=?

中途：（$M_1', M_2', \cdots M_n'$）　　XOR（$M_1', M_2', \ldots M_n'$）=?

最后：（0, 0, …, 0）　　XOR（0，0，… 0）=0

不幸的是，可以看出，当有奇数个石头时，无论你如何分堆，XOR（$M_1, M_2, \ldots M_n$）总是不等于 0！因为必然会有奇数堆有奇数个石头（二进制表示最低位为 1），异或的结果最低位肯定为 1。　　[结论 1]

再不幸的是，还可以证明，当XOR（$M_1, M_2, \ldots M_n$）!= 0 时，我们总是只需要改变一个M_i的值，就可以让XOR（$M_1, M_2, \ldots M_i', \ldots M_n$）= 0。　　[结论 2]

更不幸的是，又可以证明，当XOR（$M_1, M_2, \ldots M_n$）= 0 时，对任何一个M值的改变（取走石头），都会让XOR（$M_1, M_2, \ldots M_i', \ldots M_n$）! = 0。　　[结论 3]

有了这三个“不幸”的结论，我们不得不承认，当 M 为奇数时，无论怎样分堆，总是先动手的人赢。

还不信？那我们试试看：当 M=9，随机分堆为（1，2，6）

开始：（1，2，6）

1=0 0 1
2=0 1 0
6=1 1 0
XOR=1 0 1　　即 XOR（1，2，6）!=0

B 先手：（1, 2, 3），即从第三堆取走三个，得到（1，2，3）

1=0 0 1
2=0 1 0
3=0 1 1
XOR=0 0 0　　所以，XOR（1，2，3）=0

A 方：（1, 2, 2）XOR（1, 2, 2）!=0。

B 方：（0, 2, 2）XOR （0, 2, 2）=0

……A 方继续顽抗……

B 方最后：（0, 0, 0），XOR（0, 0, 0）= 0

好了，通过以上的分析，我们不但知道了这类问题的答案，还知道了游戏的规律，以及如何才能赢。XOR，这个我们很早就学过的运算，在这里帮了大忙[1]。我们应该对XOR说Orz才对！

有兴趣的读者可以写一个程序，返回当输入为（$M_1, M_2, \cdots, M_n$）的时候，到底如何取石头，才能有赢的可能。比如，当输入为（3, 4, 5）的时候[2]，程序返回（1, 4, 5）——这样就转败为胜了！

扩展问题

1．如果规定相反，取光所有石头的人输，又该如何控制局面？

[1] 温馨提示：你还记得教我们XOR运算的老师么？这门课一定比较枯燥吧，如果当时能玩NIM这个游戏就好了。

[2] 提一句，这是一个不明智的分堆办法，不如分为（6, 6），这样必赢无疑。

2．如果每次可以挑选任意K堆，并从中任意取石头，又该如何找到必胜策略呢？

《编程之美——微软技术面试心得》

《编程之美——微软技术面试心得》（http://www.china-pub.com/38070）是微软亚洲研究院技术创新组研发主管邹欣继《移山之道——VSTS 软件开发指南》后的最新力作。它传达给读者：微软重视什么样的能力，需要什么样的人才。但它更深层的意义在于引导读者思考，提倡一种发现问题、解决问题的思维方式，充分挖掘编程的乐趣，展示编程之美。本书 3 月份上市。网上讨论和解答在：www.msra.cn/bop

题目《让 CPU 占用率曲线听你指挥》

问题

写一个程序，让用户来决定 Windows 任务管理器（Task Manager）的 CPU 占用率。程序越精简越好，计算机语言不限。例如，可以实现下面三种情况：

1. CPU的占用率固定在50%，为一条直线；

2. CPU的占用率为一条直线，但是具体占用率由命令行参数决定（参数范围1~ 100）；

3. CPU的占用率状态是一个正弦曲线。

分析与解法[1]

有一名学生写了如下的代码：

```
while (true)
{
   if (busy)
      i++;
   else

}
```

然后她就陷入了苦苦思索：else 干什么呢？怎么才能让电脑不做事情呢？CPU 使用率为 0 的时候，到底是什么东西在用 CPU？另一名学生花了很多时间构想如何“深入内核，以控制 CPU 占用率”——可是事情真的有这么复杂么？

MSRA TTG（Microsoft Research Asia, Technology Transfer Group）的一些实习生写了各种解法，他们写的简单程序可以达到如图 1-1 所示的效果。

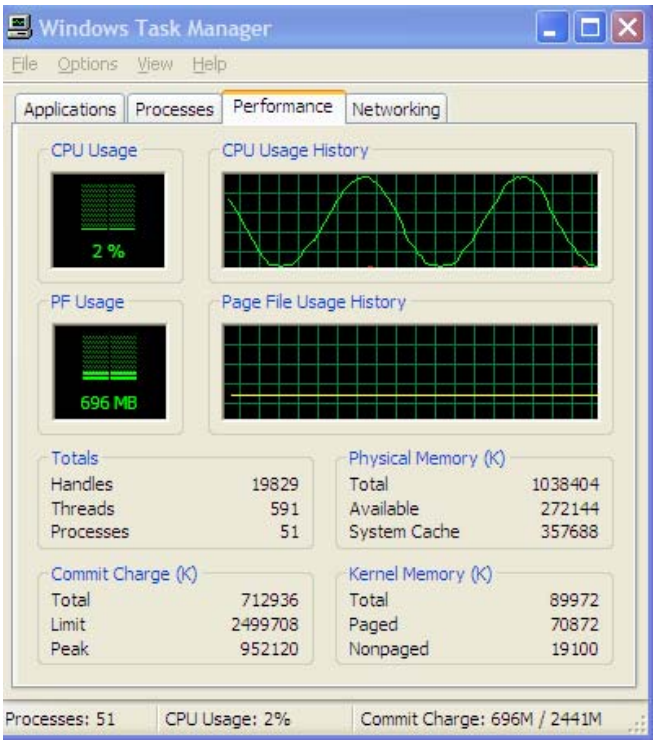


图 1-1 编码控制 CPU 占用率呈现正弦曲线形态

看来这并不是不可能完成的任务。让我们仔细地回想一下写程序时曾经碰到的问题，如

[1] 作者注：当面试的同学听到这个问题的时候，很多人都有点意外。我把我的笔记本电脑交给他们说，这是开卷考试，你可以上网查资料，干什么都可以。大部分面试者在电脑上的第一个动作就是上网搜索“CPU 控制 50%”这样的关键字，当然没有找到什么直接的结果。不过这本书出版以后，情况可能就不一样了。

果我们不小心写了一个死循环，CPU 占用率就会跳到最高，并且一直保持 100%。我们也可以打开任务管理器[2]，实际观测一下它是怎样变动的。凭肉眼观察，它大约是 1 秒钟更新一次。一般情况下，CPU 使用率会很低。但是，当用户运行一个程序，执行一些复杂操作的时候，CPU 的使用率会急剧升高。当用户晃动鼠标时，CPU 的使用率也有小幅度的变化。

那当任务管理器报告 CPU 使用率为 0 的时候，谁在使用 CPU 呢？通过任务管理器的“进程（Process）”一栏可以看到，System Idle Process 占用了 CPU 空闲的时间——这时候大家该回忆起在“操作系统原理”这门课上学到的一些知识了吧。系统中有那么多进程，它们什么时候能“闲下来”呢？答案很简单，这些程序或者在等待用户的输入，或者在等待某些事件的发生（`WaitForSingleObject()`），或者进入休眠状态（通过 `Sleep()`来实现）。

在任务管理器的一个刷新周期内，CPU 忙（执行应用程序）的时间和刷新周期总时间的比率，就是 CPU 的占用率，也就是说，任务管理器中显示的是每个刷新周期内 CPU 占用率的统计平均值。因此，我们写一个程序，让它在任务管理器的刷新期间内一会儿忙，一会儿闲，然后通过调节忙/闲的比例，就可以控制任务管理器中显示的 CPU 占用率。

【解法一】简单的解法

步骤 1 要操纵 CPU 的 usage 曲线，就需要使 CPU 在一段时间内（根据 Task Manager 的采样率）跑 busy 和 idle 两个不同的 loop，从而通过不同的时间比例，来获得调节 CPU Usage 的效果。

步骤 2 Busy loop 可以通过执行空循环来实现，idle 可以通过 `Sleep()`来实现。

问题的关键在于如何控制两个 loop 的时间，方法有二：

Sleep一段时间，然后以for循环n次，估算n的值。

那么对于一个空循环 `for(i = 0; i < n; i++)`；又该如何来估算这个最合适的 n 值呢？我们都知道 CPU 执行的是机器指令，而最接近于机器指令的语言是汇编语言，所以我们可以先把这个空循环简单地写成如下汇编代码后再进行分析：

```
loop:
mov dx i        ;将i置入dx寄存器
inc dx          ;将dx寄存器加1
mov i dx        ;将dx中的值赋回i
cmp i n         ;比较i和n
jl loop         ;i 小于 n 时则重复循环
```

假设这段代码要运行的 CPU 是 P4 2.4Ghz（2.4 * 10 的 9 次方个时钟周期每秒）。现代 CPU 每个时钟周期可以执行两条以上的代码，那么我们就取平均值两条，于是让（2 400 000 000 * 2）/5=960 000 000（循环/秒），也就是说 CPU 1 秒钟可以运行这个空循环 960 000 000 次。不过我们还是不能简单地将 n = 60 000 000，然后 `Sleep(1000)`了事。如果我们让 CPU 工

[2] 如果应聘者从来没有琢磨过任务管理器，那还是不要在简历上说“精通 Windows”为好。

作 1 秒钟 ,然后休息 1 秒钟 ,波形很有可能就是锯齿状的——先达到一个峰值(大于>50%) , 然后跌到一个很低的占用率。

我们尝试着降低两个数量级，令 $n = 9\ 600\ 000$，而睡眠时间相应改为 10 毫秒（Sleep(10)）。用 10 毫秒是因为它不大也不小，比较接近 Windows 的调度时间片。如果选得太小（比如 1 毫秒），则会造成线程频繁地被唤醒和挂起，无形中又增加了内核时间的不确定性影响。最后我们可以得到如下代码：

代码清单 1-1

```
int main()
{
   for(;;)
   {
      for(int i = 0; i < 9600000; i++);
      Sleep(10);
   }
   return 0;
}
```

在不断调整 9 600 000 的参数后，我们就可以在一台指定的机器上获得一条大致稳定的 50% CPU 占用率直线。

使用这种方法要注意两点影响：

1. 尽量减少sleep/awake的频率，如果频繁发生，影响则会很大，因为此时优先级更高的操作系统内核调度程序会占用很多CPU运算时间。

2. 尽量不要调用system call（比如I/O这些privilege instruction），因为它也会导致很多不可控的内核运行时间。

该方法的缺点也很明显：不能适应机器差异性。一旦换了一个 CPU，我们又得重新估算 n 值。有没有办法动态地了解 CPU 的运算能力，然后自动调节忙/闲的时间比呢？请看下一个解法。

【解法二】使用 GetTickCount()和 Sleep()

我们知道 GetTickCount()可以得到“系统启动到现在”的毫秒值，最多能够统计到 49.7 天。另外，利用 Sleep()函数，最多也只能精确到 1 毫秒。因此，可以在“毫秒”这个量级做操作和比较。具体如下：

利用 GetTickCount()来实现 busy loop 的循环，用 Sleep()实现 idle loop。伪代码如下：

代码清单 1-2

```
int busyTime = 10;  //10 ms
int idleTime = busyTime;  //same ratio will lead to 50% cpu usage
```

```
Int64 startTime = 0;
while (true)
{
    startTime = GetTickCount();
    // busy loop的循环
    while ((GetTickCount() - startTime) <= busyTime) ;

    //idle loop
    Sleep(idleTime);
}
```

这两种解法都是假设目前系统上只有当前程序在运行，但实际上，操作系统中有很多程序都会在不同时间执行各种各样的任务，如果此刻其他进程使用了 10% 的 CPU，那我们的程序应该只能使用 40%的 CPU（而不是机械地占用 50%），这样可达到 50%的效果。

怎么做呢？

我们得知道"当前 CPU 占用率是多少"，这就要用到另一个工具来帮忙——Perfmon.exe。

Perfmon 是从 Windows NT 开始就包含在 Windows 服务器和台式机操作系统的管理工具组中的专业监视工具之一（如图 1-2 所示）。Perfmon 可监视各类系统计数器，获取有关操作系统、应用程序和硬件的统计数字。Perfmon 的用法相当直接，只要选择您所要监视的对象（比如：处理器、RAM 或硬盘），然后选择所要监视的计数器（比如监视物理磁盘对象时的平均队列长度）即可。还可以选择所要监视的实例，比如面对一台多 CPU 服务器时，可以选择监视特定的处理器。

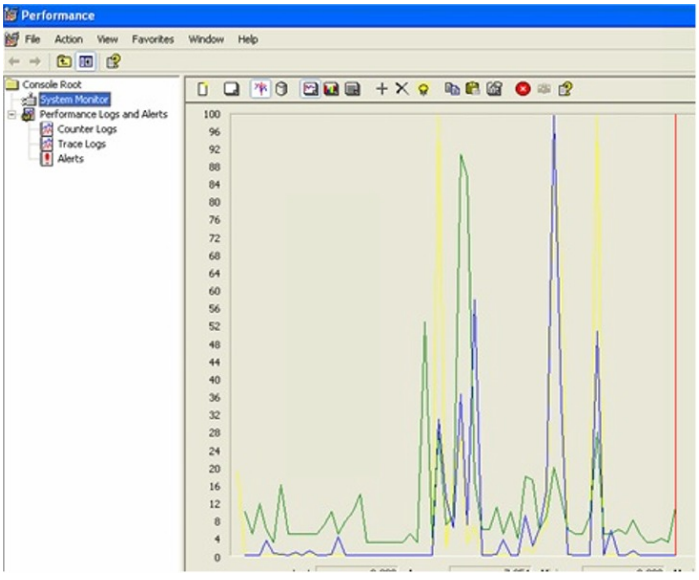


图 1-2 系统监视器（Perfmon）

我们可以写程序来查询 Perfmon 的值，Microsoft .Net Framework 提供了 PerformanceCounter()这一类型，从而可以方便地拿到当前各种计算机性能数据，包括 CPU 的使用率。例如下面这个程序——

【解法三】能动态适应的解法

代码清单 1-3

```
//C# code
static void MakeUsage(float level)
{
   PerformanceCounter p = new PerformanceCounter("Processor", "% Processor Time",
     "_Total");

   while (true)
   {
      if (p.NextValue() > level)
         System.Threading.Thread.Sleep(10);
    }
}
```

可以看到，上面的解法能方便地处理各种 CPU 使用率参数。这个程序可以解答前面提到的问题 2。

有了前面的积累，我们应该可以让任务管理器画出优美的正弦曲线了，见下面的代码。

【解法四】正弦曲线

代码清单 1-4

```
//C++ code  to make task manager generate sine graph
#include "Windows.h"
#include "stdlib.h"
#include "math.h"

const double SPLIT = 0.01;
const int COUNT = 200;
const double PI = 3.14159265;
const int INTERVAL = 300;

int _tmain(int argc, _TCHAR* argv[])
{
    DWORD busySpan[COUNT];  //array of busy times
    DWORD idleSpan[COUNT];  //array of idle times
    int half = INTERVAL / 2;
    double radian = 0.0;
    for(int i = 0; i < COUNT; i++)
    {
        busySpan[i] = (DWORD)(half + (sin(PI * radian) * half));
        idleSpan[i] = INTERVAL - busySpan[i];
        radian += SPLIT;
    }

    DWORD startTime = 0;
    int j = 0;
    while (true)
    {
        j = j % COUNT;
        startTime = GetTickCount();
        while ((GetTickCount() - startTime) <= busySpan[j]) ;
        Sleep(idleSpan[j]);
        j++;
    }
    return 0;
}
```

讨论

如果机器是多 CPU，上面的程序会出现什么结果？如何在多个 CPU 时显示同样的状态？例如，在双核的机器上，如果让一个单线程的程序死循环，能让两个 CPU 的使用率达到 50%的水平么？为什么？

多 CPU 的问题首先需要获得系统的 CPU 信息。可以使用 GetProcessorInfo()获得多处理器的信息,然后指定进程在哪一个处理器上运行。其中指定运行使用的是 SetThreadAffinityMask()函数。

另外，还可以使用 RDTSC 指令获取当前 CPU 核心运行周期数。

在 x86 平台上定义函数：

```
inline __int64 GetCPUTickCount()
{
    __asm
    {
       rdtsc;
    }
}
```

在 x64 平台上定义：

```
#define GetCPUTickCount() __rdtsc()
```

使用 CallNtPowerInformation API 得到 CPU 频率，从而将周期数转化为毫秒数，例如：

代码清单 1-5

```
_PROCESSOR_POWER_INFORMATION info;

CallNTPowerInformation(11,   //query processor power information
   NULL,               //no input buffer
   0,                 //input buffer size is zero
   &info,             //output buffer
   Sizeof(info));    //outbuf size

   __int64 t_begin = GetCPUTickCount();

   //do something

   __int64 t_end = GetCPUTickCount();
   double millisec = ((double)t_end –
      (double)t_begin)/(double)info.CurrentMhz;
```

RDTSC 指令读取当前 CPU 的周期数，在多 CPU 系统中，这个周期数在不同的 CPU 之间基数不同，频率也有可能不同。用从两个不同的 CPU 得到的周期数作计算会得出没有意义的值。如果线程在运行中被调度到了不同的 CPU，就会出现上述情况。可用 SetThreadAffinityMask 避免线程迁移。另外 ,CPU 的频率会随系统供电及负荷情况有所调整。

总结

能帮助你了解当前线程/进程/系统效能的 API 大致有以下这些：

1. Sleep()——这个方法能让当前线程“停”下来。
2. WaitForSingleObject()——自己停下来，等待某个事件发生
3. GetTickCount()——有人把 Tick 翻译成“嘀嗒”，很形象。
4. QueryPerformanceFrequency()、QueryPerformanceCounter()——让你访问到精度更高的 CPU 数据。
5. timeGetSystemTime()——是另一个得到高精度时间的方法。
6. PerformanceCounter——效能计数器。
7. GetProcessorInfo()/SetThreadAffinityMask()。遇到多核的问题怎么办呢？这两个方法能够帮你更好地控制 CPU。
8. GetCPUTickCount()。想拿到 CPU 核心运行周期数吗？用用这个方法吧。

了解并应用了上面的 API，就可以考虑在简历中写上“精通 Windows”了。

2.2 不要被阶乘吓倒 ★★

阶乘（Factorial）是个很有意思的函数，但是不少人都比较怕它，我们来看看两个与阶乘相关的问题：

1. 给定一个整数N，那么N的阶乘N！末尾有多少个0呢？例如：$N=10$，$N!=3\,628\,800$，N！的末尾有两个0。

2. 求N！的二进制表示中最低位1的位置。

分析与解法

有些人碰到这样的题目会想：是不是要完整计算出 N！的值？如果溢出怎么办？事实上，如果我们从“哪些数相乘能得到 10”这个角度来考虑，问题就变得简单了。

首先考虑，如果 $N! = K \times 10^M$，且 K 不能被 10 整除，那么 $N!$ 末尾有 M 个 0。再考虑对 $N!$ 进行质因数分解，$N! = (2^x) \times (3^y) \times (5^z) \cdots$，由于 $10 = 2 \times 5$，所以 M 只跟 X 和 Z 相关，每一对 2 和 5 相乘可以得到一个 10，于是 $M = \min(X, Z)$。不难看出 X 大于等于 Z，因为能被 2 整除的数出现的频率比能被 5 整除的数高得多，所以把公式简化为 $M = Z$。

根据上面的分析，只要计算出 Z 的值，就可以得到 $N!$ 末尾 0 的个数。

【问题 1 的解法一】

要计算 Z，最直接的方法，就是计算 i（$i = 1, 2, \cdots, N$）的因式分解中 5 的指数，然后求和：

代码清单 2-6

```
ret = 0;
for(i = 1; i <= N; i++)
{
    j = i;
    while(j % 5 ==0)
    {
        ret++;
        j /= 5;
    }
}
```

【问题 1 的解法二】

公式：$Z = [N/5] + [N/5^2] + [N/5^3] + \cdots$（不用担心这会是一个无穷的运算，因为总存在一个 K，使得 $5^K > N$，$[N/5^K]=0$。）

公式中，$[N/5]$表示不大于 N 的数中 5 的倍数贡献一个 5，$[N/5^2]$表示不大于 N 的数中 5^2 的倍数再贡献一个 5，……代码如下：

```
ret = 0;
while(N)
{
    ret += N / 5;
    N /= 5;
}
```

问题 2 要求的是 $N!$ 的二进制表示中最低位 1 的位置。给定一个整数 N，求 $N!$ 二进制表示的最低位 1 在第几位？例如：给定 $N = 3$，$N! = 6$，那么 $N!$ 的二进制表示（1 010）的最低位 1 在第二位。

为了得到更好的解法，首先要对题目进行一下转化。

首先来看一下一个二进制数除以 2 的计算过程和结果是怎样的。

把一个二进制数除以 2，实际过程如下：

判断最后一个二进制位是否为 0，若为 0，则将此二进制数右移一位，即为商值（为什么）；反之，若为 1，则说明这个二进制数是奇数，无法被 2 整除（这又是为什么）。

所以，这个问题实际上等同于求 *N*！含有质因数 2 的个数。即答案等于 *N*！含有质因数 2 的个数加 1。

【问题 2 的解法一】

由于 *N*! 中含有质因数 2 的个数，等于 $N/2 + N/4 + N/8 + N/16 + \cdots$[1]，

根据上述分析，得到具体算法，如下所示：

代码清单 2-7

```
int lowestOne(int N)
{
    int Ret = 0;
    while(N)
    {
        N >>= 1;
        Ret += N;
    }
    return Ret;
}
```

【问题 2 的解法二】

N！含有质因数 2 的个数，还等于 *N* 减去 *N* 的二进制表示中 1 的数目。我们还可以通过这个规律来求解。

下面对这个规律进行举例说明，假设 $N = 11011$，那么 *N*!中含有质因数 2 的个数为 $N/2 + N/4 + N/8 + N/16 + \cdots$

即：
$$
\begin{aligned}
& 1101 + 110 + 11 + 1 \\
&= (1000 + 100 + 1) \\
&+ (100 + 10) \\
&+ (10 + 1) \\
&+ 1 \\
&= (1000 + 100 + 10 + 1) + (100 + 10 + 1) + 1 \\
&= 1111 + 111 + 1 \\
&= (10000 - 1) + (1000 - 1) + (10 - 1) + (1 - 1)
\end{aligned}
$$

[1] 这个规律请读者自己证明（提示 N/k，等于 $1, 2, 3, \cdots, N$ 中能被 k 整除的数的个数）。

= 11011−*N* 二进制表示中 1 的个数

小结

任意一个长度为 *m* 的二进制数 *N* 可以表示为 $N = b[1] + b[2] * 2 + b[3] * 2^2 + \cdots + b[m] * 2^{(m-1)}$，其中 *b* [*i*]表示此二进制数第 *i* 位上的数字（1 或 0）。所以，若最低位 *b*[1]为 1，则说明 *N* 为奇数；反之为偶数，将其除以 2，即等于将整个二进制数向低位移一位。

相关题目

给定整数 *n*，判断它是否为 2 的方幂（解答提示：*n*>0&&（（*n*&（*n*−1））==0））。

1.2 中国象棋将帅问题

★★★

下过中国象棋的朋友都知道，双方的“将”和“帅”相隔遥远，并且它们不能照面。在象棋残局中，许多高手能利用这一规则走出精妙的杀招。假设棋盘上只有“将”和“帅”二子（如图1-3所示）（为了下面叙述方便，我们约定用 A 表示“将”，B 表示“帅”）：

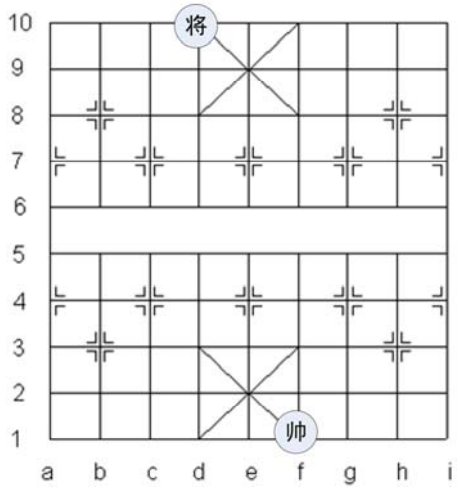


图 1-3

A、B 二子被限制在己方 3×3 的格子里运动。例如,在如上的表格里，A 被正方形$\{d_{10}, f_{10}, d_8, f_8\}$包围，而 B 被正方形$\{d_3, f_3, d_1, f_1\}$包围。每一步，A、B 分别可以横向或纵向移动一格，但不能沿对角线移动。另外，A 不能面对 B，也就是说，A 和 B 不能处于同一纵向直线上（比如 A 在 d_{10} 的位置，那么 B 就不能在 d_1、d_2 以及 d_3）。

请写出一个程序，输出 A、B 所有合法位置。要求在代码中只能使用一个变量。

分析与解法

问题的本身并不复杂，只要把所有 *A*、*B* 互相排斥的条件列举出来就可以完成本题的要求。由于本题要求只能使用一个变量，所以必须首先想清楚在写代码的时候，有哪些信息需要存储，并且尽量高效率地存储信息。稍微思考一下，可以知道这个程序的大体框架是：

```
遍历A的位置
    遍历B的位置
        判断A、B的位置组合是否满足要求。
        如果满足，则输出。
```

因此，需要存储的是 *A*、*B* 的位置信息，并且每次循环都要更新。为了能够进行判断，首先需要创建一个逻辑的坐标系统，以便检测 *A* 何时会面对 *B*。这里我们想到的方法是用 1~9 的数字，按照行优先的顺序来表示每个格点的位置（如图 1-4 所示）。这样，只需要用模余运算就可以得到当前的列号，从而判断 *A*、*B* 是否互斥。

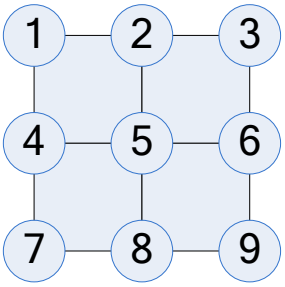


图 1-4　用 1~9 的数字表示 A、B 的坐标

第二，题目要求只用一个变量，但是我们却要存储 *A* 和 *B* 两个子的位置信息，该怎么办呢？

可以先把已知变量类型列举一下，然后做些分析。

对于 bool 类型，估计没有办法做任何扩展了，因为它只能表示 true 和 false 两个值；而 byte 或者 int 类型，它们能够表达的信息则更多。事实上，对本题来说，每个子都只需要 9 个数字就可以表达它的全部位置。

一个 8 位的 byte 类型能够表达 2^8=256 个值，所以用它来表示 *A*、*B* 的位置信息绰绰有余，因此可以把这个字节的变量（设为 *b*）分成两部分。用前面的 4 bit 表示 *A* 的位置，用后面的 4 bit 表示 *B* 的位置，那么 4 个 bit 可以表示 16 个数，这已经足够了。

问题在于：如何使用 bit 级的运算将数据从这一 byte 变量的左边和右边分别存入和读出。

下面是做法：

■ 将 byte *b*（10100101）的右边 4 bit（0101）设为 *n*（0011）：

首先清除 *b* 右边的 bits，同时保持左边的 bits:

11110000（LMASK）

& 10100101（*b*）

10100000

然后将上一步得到的结果与 *n* 做或运算

10100000（LMASK & *b*）

^ 00000011（*n*）

10100011

■ 将 byte *b*（10100101）左边的 4 bit（1010）设为 *n*（0011）：

首先，清除 *b* 左边的 bits，同时保持右边的 bits:

00001111（RMASK）

& 10100101（*b*）

00000101

现在，把 *n* 移动到 byte 数据的左边

n << 4 = 00110000

然后对以上两步得到的结果做或运算，从而得到最终结果。

00000101（RMASK & *b*）

^ 00110000（*n* << 4）

00110101

■ 得到 byte 数据的右边 4 bits 或左边 4 bits（e.g. 10100101 中的 1010 以及 0101）：

清除 *b* 左边的 bits，同时保持右边的 bits

```
    00001111 (RMASK)
&   10100101 (b)
    -----------
    00000101
```

清除 *b* 的右边的 bits，同时保持左边的 bits

```
    11110000 (LMASK)
&   10100101 (b)
    -----------
    10100000
```

将结果右移 4 bits

10100000 >> 4 = 00000101

最后的挑战是如何在不声明其他变量约束的前提下创建一个 for 循环。可以重复利用 1byte 的存储单元，把它作为循环计数器并用前面提到的存取和读入技术进行操作。还可以用宏来抽象化代码，例如：

```
for (LSET(b, 1); LGET(b) <= GRIDW * GRIDW; LSET(b, (LGET(b) + 1)))
```

【解法一】

代码清单 1-6

```
#define HALF_BITS_LENGTH 4
// 这个值是记忆存储单元长度的一半，在这道题里是4bit
#define FULLMASK 255
// 这个数字表示一个全部bit的mask，在二进制表示中，它是11111111。
#define LMASK (FULLMASK << HALF_BITS_LENGTH)
// 这个宏表示左bits的mask，在二进制表示中，它是11110000。
#define RMASK (FULLMASK >> HALF_BITS_LENGTH)
// 这个数字表示右bits的mask，在二进制表示中，它表示00001111。
#define RSET(b, n) (b = ((LMASK & b) ^ n))
// 这个宏，将b的右边设置成n
#define LSET(b, n)  (b = ((RMASK & b) ^ (n << HALF_BITS_LENGTH)))
// 这个宏，将b的左边设置成n
#define RGET(b) (RMASK & b)
// 这个宏得到b的右边的值
#define LGET(b) ((LMASK & b) >> HALF_BITS_LENGTH)
// 这个宏得到b的左边的值
#define GRIDW 3
// 这个数字表示将帅移动范围的行宽度。
#include <stdio.h>
#define HALF_BITS_LENGTH 4
#define FULLMASK 255
#define LMASK (FULLMASK << HALF_BITS_LENGTH)
#define RMASK (FULLMASK >> HALF_BITS_LENGTH)
#define RSET(b, n) (b = ((LMASK & b) ^ n))
#define LSET(b, n) (b = ((RMASK & b) ^ (n << HALF_BITS_LENGTH)))
#define RGET(b) (RMASK & b)
#define LGET(b) ((LMASK & b) >> HALF_BITS_LENGTH)
#define GRIDW 3
```

```
int main()
{
    unsigned char b;
    for(LSET(b, 1); LGET(b) <= GRIDW * GRIDW; LSET(b, (LGET(b) + 1)))
        for(RSET(b, 1); RGET(b) <= GRIDW * GRIDW; RSET(b, (RGET(b) + 1)))
            if(LGET(b) % GRIDW != RGET(b) % GRIDW)
                printf("A = %d, B = %d\n", LGET(b), RGET(b));

return 0;
}
```

【输出】

格子的位置用 N 来表示，$N = 1, 2, \cdots, 8, 9$，依照行优先的顺序，如图 1-5 所示：

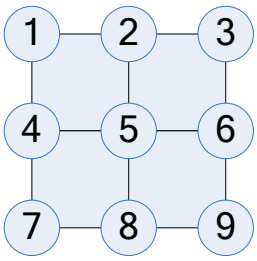
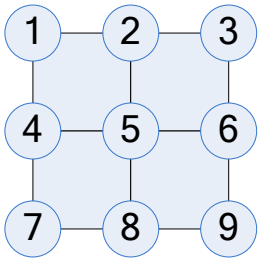
“将”（A）的格子	
“帅”（B）的格子	

图 1-5

$A = 1, B = 2$	$A = 4, B = 2$	$A = 7, B = 2$
$A = 1, B = 3$	$A = 4, B = 3$	$A = 7, B = 3$
$A = 1, B = 5$	$A = 4, B = 5$	$A = 7, B = 5$
$A = 1, B = 6$	$A = 4, B = 6$	$A = 7, B = 6$
$A = 1, B = 8$	$A = 4, B = 8$	$A = 7, B = 8$
$A = 1, B = 9$	$A = 4, B = 9$	$A = 7, B = 9$
$A = 2, B = 1$	$A = 5, B = 1$	$A = 8, B = 1$
$A = 2, B = 3$	$A = 5, B = 3$	$A = 8, B = 3$
$A = 2, B = 4$	$A = 5, B = 4$	$A = 8, B = 4$
$A = 2, B = 6$	$A = 5, B = 6$	$A = 8, B = 6$
$A = 2, B = 7$	$A = 5, B = 7$	$A = 8, B = 7$
$A = 2, B = 9$	$A = 5, B = 9$	$A = 8, B = 9$
$A = 3, B = 1$	$A = 6, B = 1$	$A = 9, B = 1$
$A = 3, B = 2$	$A = 6, B = 2$	$A = 9, B = 2$
$A = 3, B = 4$	$A = 6, B = 4$	$A = 9, B = 4$
$A = 3, B = 5$	$A = 6, B = 5$	$A = 9, B = 5$
$A = 3, B = 7$	$A = 6, B = 7$	$A = 9, B = 7$
$A = 3, B = 8$	$A = 6, B = 8$	$A = 9, B = 8$

考虑了这么多因素，总算得到了本题的一个解法，但是 MSRA 里却有人说，下面的一小段代码也能达到同样的目的：

```
BYTE i = 81;
while(i--)
{
    if(i / 9 % 3 == i % 9 % 3)
        continue;
    printf(“A = %d, B = %d\n”, i / 9 + 1, i % 9 + 1);
}
```

但是很快又有另一个人说他的解法才是效率最高的：

代码清单 1-7

```
struct {
    unsigned char a:4;
    unsigned char b:4;
} i;

for(i.a = 1; i.a <= 9; i.a++)
for(i.b = 1; i.b <= 9; i.b++)
    if(i.a % 3 == i.b % 3)
            printf("A = %d, B = %d\n", i.a, i.b);
```

读者能自己证明一下么？[1]

[1] 这一题目由微软亚洲研究院工程师 Matt Scott 提供，他在学习中国象棋的时候想出了这个题目，后来一位应聘者给出了比他的“正解”简明很多的答案，他们现在成了同事。

子数组的最大乘积

给定一个长度为 N 的整数数组，只允许用乘法，不能用除法，计算任意($N-1$)个数的组合乘积中最大的一组，并写出算法的时间复杂度。

我们把所有可能的（$N-1$）个数的组合找出来，分别计算它们的乘积，并比较大小。由于总共有 N 个（$N-1$）个数的组合，总的时间复杂度为 $O(N^2)$，但显然这不是最好的解法。

分析与解法

【解法一】

在计算机科学中，时间和空间往往是一对矛盾体，不过，这里有一个优化的折中方法。可以通过“空间换时间”或“时间换空间”的策略来达到优化某一方面的效果。在这里，是否可以通过“空间换时间”来降低时间复杂度呢？

计算（$N-1$）个数的组合乘积，假设第 i 个（$0\le i\le N-1$）元素被排除在乘积之外（如图 2-13 所示）。

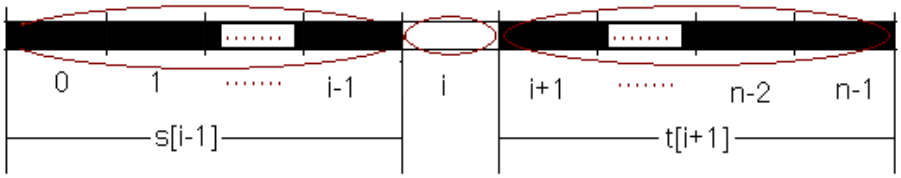


图 2-13　组合示意图

设 $array[]$为初始数组，$s[i]$表示数组前 i 个元素的乘积$s[i]=\prod_{1}^{i} array[i-1]$，其中 $1\le i\le N$，$s[0]=1$（边界条件），那么 $s[i]=s[i-1]\times array[i-1]$，其中 $i=1, 2, \cdots, N-1, N$；

设 $t[i]$表示数组后（$N-i$）个元素的乘积$t[i]=\prod_{i}^{n} array[i]$，其中 $1\le i\le N$，$t[N+1]=1$（边界条件），那么 $t[i]=t[i+1]\times array[i]$，其中 $i=1, 2, \cdots, N-1, N$；

那么设 $p[i]$为数组除第 i 个元素外,其他 $N-1$ 个元素的乘积，即有：

$p[i]=s[i-1]\times t[i+1]$。

由于只需要从头至尾，和从尾至头扫描数组两次即可得到数组 $s[]$和 $t[]$，进而线性时间可以得到 $p[]$。所以，很容易就可以得到 $p[]$的最大值（只需遍历 $p[]$一次）。总的时间复杂度等于计算数组 $s[]$、$t[]$、$p[]$的时间复杂度加上查找 $p[]$最大值的时间复杂度等于 $O(N)$。

【解法二】

其实，还可以通过分析，进一步减少解答问题的计算量。假设 N 个整数的乘积为 P，针对 P 的正负性进行如下分析（其中，A_{N-1} 表示 $N-1$ 个数的组合，P_{N-1} 表示 $N-1$ 个数的组合的乘积）：

1. P为0

   那么，数组中至少包含有一个0。假设除去一个0之外，其他$N-1$个数的乘积为Q，根据Q的正负性进行讨论：

   Q为0

   说明数组中至少有两个0，那么$N-1$个数的乘积只能为0，返回0；

   Q为正数

   返回Q，因为如果以0替换此时A_{N-1}中的任一个数，所得到的P_{N-1}为0，必然小于Q；

   Q为负数

   如果以0替换此时A_{N-1}中的任一个数，所得到的P_{N-1}为0，大于Q，乘积最大值为0。

2. P为负数

   根据“负负得正”的乘法性质，自然想到从N个整数中去掉一个负数，使得P_{N-1}为一个正数。而要使这个正数最大，这个被去掉的负数的绝对值必须是数组中最小的。我们只需要扫描一遍数组，把绝对值最小的负数给去掉就可以了。

3. P为正数

类似P为负数的情况，应该去掉一个绝对值最小的正数值，这样得到的P_{N-1}就是最大的。

上面的解法采用了直接求 N 个整数的乘积 P，进而判断 P 的正负性的办法，但是直接求乘积在编译环境下往往会有溢出的危险(这也就是本题要求不使用除法的潜在用意☺)，事实上可做一个小的转变，不需要直接求乘积，而是求出数组中正数（+）、负数（－）和 0 的个数，从而判断 P 的正负性，其余部分与以上面的解法相同。

在时间复杂度方面，由于只需要遍历数组一次，在遍历数组的同时就可得到数组中正数（+）、负数（－）和 0 的个数，以及数组中绝对值最小的正数和负数，时间复杂度为 $O(N)$。

寻找发帖“水王”

★

Tango 是微软亚洲研究院的一个试验项目。研究院的员工和实习生们都很喜欢在 Tango 上面交流灌水。传说，Tango 有一大“水王”，他不但喜欢发贴，还会回复其他 ID 发的每个帖子。坊间风闻该“水王”发帖数目超过了帖子总数的一半。如果你有一个当前论坛上所有帖子（包括回帖）的列表，其中帖子作者的 ID 也在表中，你能快速找出这个传说中的 Tango 水王吗？

分析与解法

首先想到的是一个最直接的方法，我们可以对所有 ID 进行排序。然后再扫描一遍排好序的 ID 列表，统计各个 ID 出现的次数。如果某个 ID 出现的次数超过总数的一半，那么就输出这个 ID。这个算法的时间复杂度为 $O(N * \log_2 N + N)$。

如果 ID 列表已经是有序的，还需要扫描一遍整个列表来统计各个 ID 出现的次数吗？

如果一个 ID 出现的次数超过总数 N 的一半。那么，无论水王的 ID 是什么，这个有序的 ID 列表中的第 $N/2$ 项（从 0 开始编号）一定会是这个 ID（读者可以试着证明一下）。省去重新扫描一遍列表，可以节省一点算法耗费的时间。如果能够迅速定位到列表的某一项（比如使用数组来存储列表），除去排序的时间复杂度，后处理需要的时间为 $O(1)$。

但上面两种方法都需要先对 ID 列表进行排序，时间复杂度方面没有本质的改进。能否避免排序呢？

如果每次删除两个不同的 ID（不管是否包含“水王”的 ID），那么，在剩下的 ID 列表中，“水王”ID 出现的次数仍然超过总数的一半。看到这一点之后，就可以通过不断重复这个过程，把 ID 列表中的 ID 总数降低（转化为更小的问题），从而得到问题的答案。新的思路，避免了排序这个耗时的步骤，总的时间复杂度只有 $O(N)$，且只需要常数的额外内存。伪代码如下：

代码清单 2-8

```
Type Find(Type* ID, int N)
{
    Type candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = ID[i], nTimes = 1;
        }
        else
        {
            if(candidate == ID[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    return candidate;
}
```

在这个题目中，有一个计算机科学中很普遍的思想，就是如何把一个问题转化为规模较小的若干个问题。分治、递推和贪心等都是基于这样的思路。在转化过程中，小的问题跟原问题本质上一致。这样，我们可以通过同样的方式将小问题转化为更小的问题。因此，转化过程是很重要的。像上面这个题目，我们保证了问题的解在小问题中仍然具有与原问题相同

的性质：水王的 ID 在 ID 列表中的次数超过一半。转化本身计算的效率越高，转化之后问题规模缩小得越快，则整体算法的时间复杂度越低。

扩展问题

随着 Tango 的发展，管理员发现，“超级水王”没有了。统计结果表明，有 3 个发帖很多的 ID，他们的发帖数目都超过了帖子总数目 *N* 的 1/4。你能从发帖 ID 列表中快速找出他们的 ID 吗？

寻找最大的 K 个数

在面试中，有下面的问答：

问：有很多个无序的数，我们姑且假定它们各不相等，怎么选出其中最大的若干个数呢？

答：可以这样写：int array[100] ……

问：好，如果有更多的元素呢？

答：那可以改为：int array[1000] ……

问：如果我们有很多元素，例如 1 亿个浮点数，怎么办？

答：个，十，百，千，万……那可以写：float array [100 000 000] ……

问：这样的程序能编译运行么？

答：嗯……我从来没写过这么多的 0 ……

分析与解法

【解法一】

当学生们信笔写下 float array [10000000]，他们往往没有想到这个数据结构要如何在电脑上实现，是从当前程序的栈（Stack）中分配，还是堆（Heap），还是电脑的内存也许放不下这么大的东西？

我们先假设元素的数量不大，例如在几千个左右，在这种情况下，那我们就排序一下吧。在这里，快速排序或堆排序都是不错的选择，他们的平均时间复杂度都是 $O(N * \log_2 N)$。然后取出前 K 个，$O(K)$。总时间复杂度 $O(N * \log_2 N) + O(K) = O(N * \log_2 N)$。

你一定注意到了，当 K=1 时，上面的算法也是 $O(N * \log_2 N)$ 的复杂度，而显然我们可以通过 $N-1$ 次的比较和交换得到结果。上面的算法把整个数组都进行了排序，而原题目只要求最大的 K 个数，并不需要前 K 个数有序，也不需要后 $N-K$ 个数有序。

怎么能够避免做后 $N-K$ 个数的排序呢？我们需要部分排序的算法，选择排序和交换排序都是不错的选择。把 N 个数中的前 K 大个数排序出来，复杂度是 $O(N * K)$。

那一个更好呢？$O(N * \log_2 N)$ 还是 $O(N * K)$？这取决于 K 的大小，这是你需要在面试者那里弄清楚的问题。在 $K(K <= \log_2 N)$ 较小的情况下，可以选择部分排序。

在下一个解法中，我们会通过避免对前 K 个数排序来得到更好的性能。

【解法二】

回忆一下快速排序，快排中的每一步，都是将待排数据分做两组，其中一组的数据的任何一个数都比另一组中的任何一个大，然后再对两组分别做类似的操作，然后继续下去……

在本问题中，假设 N 个数存储在数组 S 中，我们从数组 S 中随机找出一个元素 X，把数组分为两部分 S_a 和 S_b。S_a 中的元素大于等于 X，S_b 中元素小于 X。

这时，有两种可能性：

1. S_a中元素的个数小于K，S_a中所有的数和S_b中最大的$K-|S_a|$个元素（$|S_a|$指S_a中元素的个数）就是数组S中最大的K个数。

2. S_a中元素的个数大于或等于K，则需要返回S_a中最大的K个元素。

这样递归下去，不断把问题分解成更小的问题，平均时间复杂度 $O(N * \log_2 K)$。伪代码如下：

代码清单 2-11

```
Kbig(S, k):
   if(k <= 0):
      return []      // 返回空数组
   if(length S <= k):
      return S
   (Sa, Sb) = Partition(S)
   return Kbig(Sa, k).Append(Kbig(Sb, k – length Sa)

Partition(S):
   Sa = []           // 初始化为空数组
   Sb = []           // 初始化为空数组
                     // 随机选择一个数作为分组标准，以避免特殊数据下的算法退化
                     // 也可以通过对整个数据进行洗牌预处理实现这个目的
                     // Swap(S[1], S[Random() % length S])
p = S[1]
   for i in [2: length S]:
      S[i] > p ? Sa.Append(S[i]) : Sb.Append(S[i])
                     // 将p加入较小的组，可以避免分组失败，也使分组更均匀，提高效率
length Sa < length Sb ? Sa.Append(p) : Sb.Append(p)
return (Sa, Sb)
```

【解法三】

寻找 N 个数中最大的 K 个数 ,本质上就是寻找最大的 K 个数中最小的那个，也就是第 K 大的数。可以使用二分搜索的策略来寻找 N 个数中的第 K 大的数。对于一个给定的数 p，可以在 $O(N)$ 的时间复杂度内找出所有不小于 p 的数。假如 N 个数中最大的数为 V_{max}，最小的数为 V_{min}，那么这 N 个数中的第 K 大数一定在区间$[V_{min}, V_{max}]$之间。那么，可以在这个区间内二分搜索 N 个数中的第 K 大数 p。伪代码如下：

代码清单 2-12

```
while(Vmax – Vmin > delta)
{
   Vmid = Vmin + (Vmax - Vmin) * 0.5;
   if(f(arr, N, Vmid) >= K)
      Vmin = Vmid;
   else
      Vmax = Vmid;
   }
```

伪代码中 $f(arr, N, V_{mid})$ 返回数组 $arr[0, \ldots, N-1]$中大于等于 V_{mid} 的数的个数。

上述伪代码中，delta 的取值要比所有 N 个数中的任意两个不相等的元素差值之最小值小。如果所有元素都是整数，delta 可以取值 0.5。循环运行之后，得到一个区间（V_{min}, V_{max}），这个区间仅包含一个元素（或者多个相等的元素）。这个元素就是第 K 大的元素。整个算法的时间复杂度为 $O(N * \log_2(|V_{max} - V_{min}|/delta))$。由于 delta 的取值要比所有 N 个数中的任意两个不相等的元素差值之最小值小，因此时间复杂度跟数据分布相关。在数据分布平均的情况下，时间复杂度为 $O(N * \log_2(N))$。

在整数的情况下，可以从另一个角度来看这个算法。假设所有整数的大小都在$[0, 2^{m-1}]$之间，也就是说所有整数在二进制中都可以用 m bit 来表示（从低位到高位,分别用 $0, 1, \ldots, m-1$ 标记）。我们可以先考察在二进制位的第（$m-1$）位，将 N 个整数按该位为 1 或者 0 分成两个部分。也就是将整数分成取值为$[0, 2^{m-1}-1]$和$[2^{m-1}, 2^m-1]$两个区间。前一个区间中的整数第（$m-1$）位为 0，后一个区间中的整数第（$m-1$）位为 1。如果该位为 1 的整数个数 A 大于等于 K，那么，在所有该位为 1 的整数中继续寻找最大的 K 个。否则，在该位为 0 的整数中寻找最大的 $K-A$ 个。接着考虑二进制位第（$m-2$）位，以此类推。思路跟上面的浮点数的情况本质上一样。

对于上面两个方法，我们都需要遍历一遍整个集合，统计在该集合中大于等于某一个数的整数有多少个。不需要做随机访问操作，如果全部数据不能载入内存，可以每次都遍历一遍文件。经过统计，更新解所在的区间之后，再遍历一次文件，把在新的区间中的元素存入新的文件。下一次操作的时候，不再需要遍历全部的元素。每次需要两次文件遍历，最坏情况下，总共需要遍历文件的次数为 $2 * \log_2(|V_{max} - V_{min}|/delta)$。由于每次更新解所在区间之后，元素数目会减少。当所有元素能够全部载入内存之后，就可以不再通过读写文件的方式来操作了。

此外，寻找 N 个数中的第 K 大数，是一个经典问题。理论上，这个问题存在线性算法。不过这个线性算法的常数项比较大，在实际应用中效果有时并不好。

【解法四】

我们已经得到了三个解法，不过这三个解法有个共同的地方，就是需要对数据访问多次，那么就有下一个问题，如果 N 很大呢，100 亿？（更多的情况下，是面试者问你这个问题）。这个时候数据不能全部装入内存（不过也很难说，说知道以后会不会 1T 内存比 1 斤白菜还便宜），所以要求尽可能少的遍历所有数据。

不妨设 $N > K$，前 K 个数中的最大 K 个数是一个退化的情况，所有 K 个数就是最大的 K 个数。如果考虑第 K+1 个数 X 呢？如果 X 比最大的 K 个数中的最小的数 Y 小，那么最大的 K 个数还是保持不变。如果 X 比 Y 大，那么最大的 K 个数应该去掉 Y，而包含 X。如果用一个数组来存储最大的 K 个数，每新加入一个数 X，就扫描一遍数组，得到数组中最小的数 Y。用 X 替代 Y，或者保持原数组不变。这样的方法，所耗费的时间为 $O(N * K)$。

进一步，可以用容量为 K 的最小堆来存储最大的 K 个数。最小堆的堆顶元素就是最大 K 个数中最小的一个。每次新考虑一个数 X，如果 X 比堆顶的元素 Y 小，则不需要改变原来的堆，因为这个元素比最大的 K 个数小。如果 X 比堆顶元素大，那么用 X 替换堆顶的元素 Y。在 X 替换堆顶元素 Y 之后，X 可能破坏最小堆的结构（每个结点都比它的父亲结点大），需要更新堆来维持堆的性质。更新过程花费的时间复杂度为 $O(\log_2 K)$。

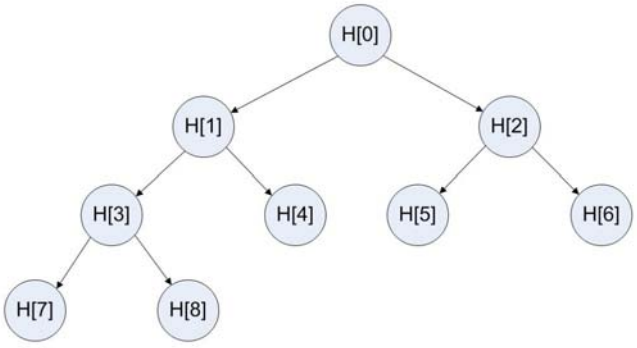


图 2-1

图 2-1 是一个堆，用一个数组 $h[]$表示。每个元素 $h[i]$，它的父亲结点是 $h[i/2]$，儿子结点是 $h[2 * i + 1]$和 $h[2 * i + 2]$。每新考虑一个数 X，需要进行的更新操作伪代码如下:

代码清单 2-13

```
if(X > h[0])
{
   h[0] = X;
   p = 0;
   while(p < K)
   {
      q = 2 * p + 1;
      if(q >= K)
         break;
      if((q < K – 1) && (h[q + 1] < h[q]))
         q = q + 1;
      if(h[q] < h[p])
      {
         t = h[p];
         h[p] = h[q];
         h[q] = t;
         p = q;
      }
      else
         break;
   }
}
```

因此，算法只需要扫描所有的数据一次，时间复杂度为 $O(N * \log_2 K)$。这实际上是部分执行了堆排序的算法。在空间方面，由于这个算法只扫描所有的数据一次，因此我们只需要存储一个容量为 K 的堆。大多数情况下，堆可以全部载入内存。如果 K 仍然很大，我们可以尝试先找最大的 K'个元素，然后找第 $K'+1$ 个到第 $2 * K'$个元素，如此类推（其中容量 K'的堆可以完全载入内存）。不过这样，我们需要扫描所有数据 ceil[1]（K/K'）次。

【解法五】

上面类快速排序的方法平均时间复杂度是线性的。能否有确定的线性算法呢？是否可以通过改进计数排序、基数排序等来得到一个更高效的算法呢？答案是肯定的。但算法的适用范围会受到一定的限制。

[1] ceil（ceiling，天花板之意）表示大于等于一个浮点数的最小整数。

如果所有 N 个数都是正整数，且它们的取值范围不太大，可以考虑申请空间，记录每个整数出现的次数，然后再从大到小取最大的 K 个。比如，所有整数都在（0, MAXN）区间中的话，利用一个数组 count[MAXN]来记录每个整数出现的个数（count[i]表示整数 i 在所有整数中出现的个数）。我们只需要扫描一遍就可以得到 count 数组。然后，寻找第 K 大的元素：

代码清单 2-14

```
for(sumCount = 0, v = MAXN – 1; v >= 0; v--)
{
   sumCount += count[v];
   if(sumCount >= K)
      break;
}
return v;
```

极端情况下，如果 N 个整数各不相同，我们甚至只需要一个 bit 来存储这个整数是否存在。

当实际情况下，并不一定能保证所有元素都是正整数，且取值范围不太大。上面的方法仍然可以推广适用。如果 N 个数中最大的数为 V_{max} ,最小的数为 V_{min}，我们可以把这个区间$[V_{min}, V_{max}]$分成 M 块，每个小区间的跨度为 $d = (V_{max} - V_{min})/M$，即 $[V_{min}, V_{min}+d], [V_{min} + d, V_{min} + 2d]$,……然后，扫描一遍所有元素，统计各个小区间中的元素个数，跟上面方法类似地，我们可以知道第 K 大的元素在哪一个小区间。然后，再对那个小区间，继续进行分块处理。这个方法介于解法三和类计数排序方法之间，不能保证线性。跟解法三类似地，时间复杂度为 $O((N+M) * \log_2 M(|V_{max} - V_{min}|/\text{delta}))$。遍历文件的次数为 $2 * \log_2 M(|V_{max} - V_{min}|/\text{delta})$。当然，我们需要找一个尽量大的 M，但 M 取值要受内存限制。

在这道题中，我们根据 K 和 N 的相对大小，设计了不同的算法。在实际面试中，如果一个面试者能针对一个问题，说出多种不同的方法，并且分析它们各自适用的情况，那一定会给人留下深刻印象。

注：本题目的解答中用到了多种排序算法，这些算法在大部分的算法书籍中都有讲解。掌握排序算法对工作也会很有帮助。

扩展问题

3. 如果需要找出N个数中最大的K个不同的浮点数呢？比如，含有10个浮点数的数组（1.5, 1.5, 2.5, 2.5, 3.5, 3.5, 5, 0, −1.5, 3.5）中最大的3个不同的浮点数是（5, 3.5, 2.5）。

4. 如果是找第k到m（$0 < k <= m <= n$）大的数呢？

5. 在搜索引擎中，网络上的每个网页都有“权威性”权重，如page rank。如果我们需要寻找权重最大的K个网页，而网页的权重会不断地更新，那么算法要如何变动以达到快速更新（incremental update）并及时返回权重最大的K个网页？

   提示：堆排序？当每一个网页权重更新的时候，更新堆。还有更好的方法吗？

6. 在实际应用中，还有一个“精确度”的问题。我们可能并不需要返回严格意义上的最大的K个元素，在边界位置允许出现一些误差。当用户输入一个query的时候，对于每一个文档d 来说，它跟这个query之间都有一个相关性衡量权重f （query, d）。搜索引擎需要返回给用户的就是相关性权重最大的K个网页。如果每页10个网页，用户不会关心第1000页开外搜索结果的“精确度”，稍有误差是可以接受的。比如我们可以返回相关性第10 001大的网页，而不是第9999大的。在这种情况下，算法该如何改进才能更快更有效率呢？网页的数目可能大到一台机器无法容纳得下，这时怎么办呢？

   提示：归并排序？如果每台机器都返回最相关的 K 个文档，那么所有机器上最相关 K 个文档的并集肯定包含全集中最相关的 K 个文档。由于边界情况并不需要非常精确，如果每台机器返回最好的 K' 个文档，那么 K' 应该如何取值，以达到我们返回最相关的 90%*K 个文档是完全精确的，或者最终返回的最相关的 K 个文档精确度超过 90%（最相关的 K 个文档中

写书评，赢取《编程之美--微软技术面试心得》www.ieee.org.cn/BCZM.asp

90%以上在全集中相关性的确排在前 K），或者最终返回的最相关的 K 个文档最差的相关性排序没有超出 110%*K。

7. 如第4点所说，对于每个文档d，相对于不同的关键字q_1, q_2, …, q_m，分别有相关性权重$f(d, q_1)$，$f(d, q_2)$，…, $f(d, q_m)$。如果用户输入关键字q_i之后，我们已经获得了最相关的K个文档，而已知关键字q_j跟关键字q_i相似，文档跟这两个关键字的权重大小比较靠近，那么关键字q_i的最相关的K个文档，对寻找q_j最相关的K个文档有没有帮助呢？

求二叉树中节点的最大距离

如果我们把二叉树看成一个图，父子节点之间的连线看成是双向的，我们姑且定义“距离”为两个节点之间边的个数。

写一个程序求一棵二叉树中相距最远的两个节点之间的距离。

如图 3-11 所示，粗箭头的边表示最长距离：

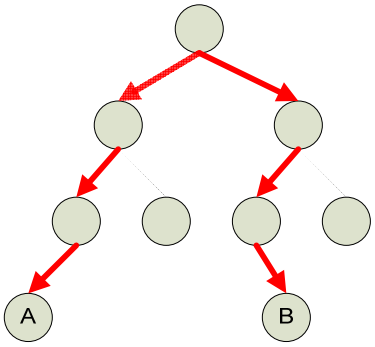


图 3-11　树中相距最远的两个节点 A，B

分析与解法

我们先画几个不同形状的二叉树，（如图 3-12 所示），看看能否得到一些启示。

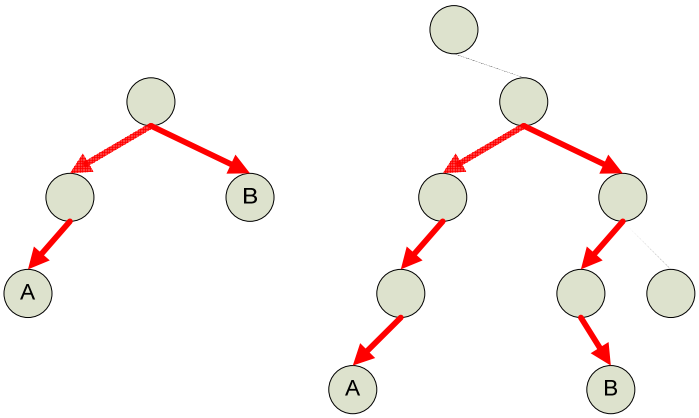


图 3-12　几个例子

从例子中可以看出，相距最远的两个节点，一定是两个叶子节点，或者是一个叶子节点到它的根节点。（为什么？）

【解法一】

根据相距最远的两个节点一定是叶子节点这个规律，我们可以进一步讨论。

对于任意一个节点，以该节点为根，假设这个根有 K 个孩子节点，那么相距最远的两个节点 U 和 V 之间的路径与这个根节点的关系有两种情况：

1. 若路径经过根Root，则U和V是属于不同子树的，且它们都是该子树中到根节点最远的节点，否则跟它们的距离最远相矛盾。这种情况如图3-13所示：

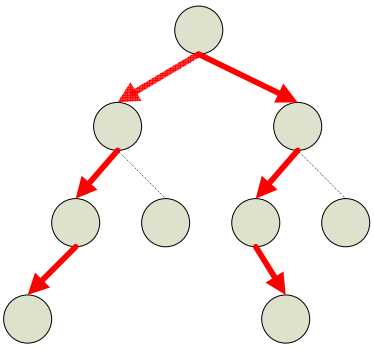


图 3-13　相距最远的节点在左右最长的子树中

2. 如果路径不经过Root，那么它们一定属于根的K个子树之一。并且它们也是该子树中相距最远的两个顶点。如图3-14中的节点A：

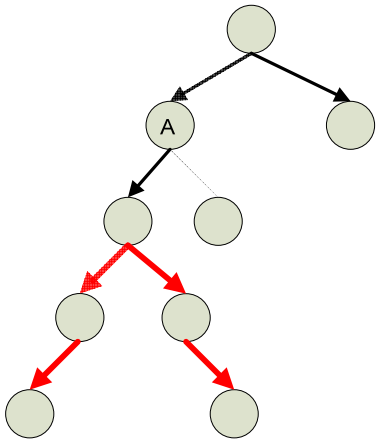


图 3-14　相距最远的节点在某个子树下

因此，问题就可以转化为在子树上的解，从而能够利用动态规划来解决。

设第 K 棵子树中相距最远的两个节点：U_k 和 V_k，其距离定义为 $d(U_k, V_k)$，那么节点 U_k 或 V_k 即为子树 K 到根节点 R_k 距离最长的节点。不失一般性，我们设 U_k 为子树 K 中到根节点 R_k 距离最长的节点，其到根节点的距离定义为 $d(U_k, R)$。取 $d(U_i, R)$（$1 \le i \le k$）中最大的两个值 max1 和 max2，那么经过根节点 R 的最长路径为 max1+max2+2，所以树 R 中相距最远的两个点的距离为：$\max\{d(U_1, V_1), \cdots, d(U_k, V_k)$，max1+max2+2$\}$。

采用深度优先搜索如图 3-15，只需要遍历所有的节点一次，时间复杂度为 $O(|E|) = O(|V|-1)$，其中 V 为点的集合，E 为边的集合。

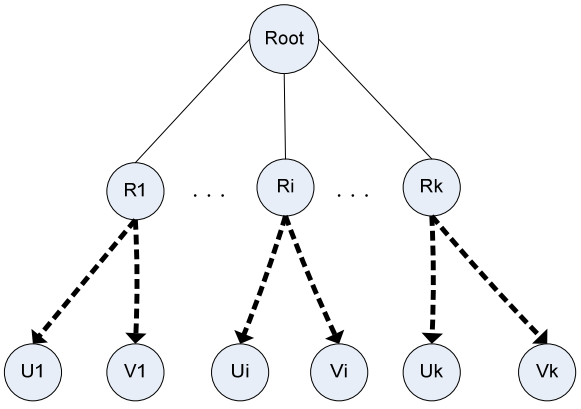


图 3-15　深度遍历示意图

示例代码如下，我们使用二叉树来实现该算法。

代码清单 3-11

```
// 数据结构定义
```

```
struct NODE
{
    NODE* pLeft;        // 左孩子
    NODE* pRight;       // 右孩子
    int nMaxLeft;       // 左子树中的最长距离
    int nMaxRight;      // 右子树中的最长距离
    char chValue;       // 该节点的值
};

int nMaxLen = 0;

// 寻找树中最长的两段距离
void FindMaxLen(NODE* pRoot)
{
    // 遍历到叶子节点，返回
    if(pRoot == NULL)
    {
        return;
    }

    // 如果左子树为空，那么该节点的左边最长距离为0
    if(pRoot -> pLeft == NULL)
    {
        pRoot -> nMaxLeft = 0;
    }

    // 如果右子树为空，那么该节点的右边最长距离为0
    if(pRoot -> pRight == NULL)
    {
        pRoot -> nMaxRight = 0;
    }

    // 如果左子树不为空，递归寻找左子树最长距离
    if(pRoot -> pLeft != NULL)
    {
        FindMaxLen(pRoot -> pLeft);
    }

    // 如果右子树不为空，递归寻找右子树最长距离
    if(pRoot -> pRight != NULL)
    {
        FindMaxLen(pRoot -> pRight);
    }

    // 计算左子树最长节点距离
    if(pRoot -> pLeft != NULL)
    {
        int nTempMax = 0;
        if(pRoot -> pLeft -> nMaxLeft > pRoot -> pLeft -> nMaxRight)
        {
            nTempMax = pRoot -> pLeft -> nMaxLeft;
        }
        else
        {
            nTempMax = pRoot -> pLeft -> nMaxRight;
        }
        pRoot -> nMaxLeft = nTempMax + 1;
    }

    // 计算右子树最长节点距离
    if(pRoot -> pRight != NULL)
```

```
    {
        int nTempMax = 0;
        if(pRoot -> pRight -> nMaxLeft > pRoot -> pRight -> nMaxRight)
        {
            nTempMax = pRoot -> pRight -> nMaxLeft;
        }
        else
        {
            nTempMax = pRoot -> pRight -> nMaxRight;
        }
        pRoot -> nMaxRight = nTempMax + 1;
    }

    // 更新最长距离
    if(pRoot -> nMaxLeft + pRoot -> nMaxRight > nMaxLen)
    {
        nMaxLen = pRoot -> nMaxLeft + pRoot -> nMaxRight;
    }
}
```

扩展问题

在代码中，我们使用了递归的办法来完成问题的求解。那么是否有非递归的算法来解决这个问题呢？

总结

对于递归问题的分析，笔者有一些小小的体会：

1. 先弄清楚递归的顺序。在递归的实现中，往往需要假设后续的调用已经完成，在此基础之上，才实现递归的逻辑。在该题中，我们就是假设已经把后面的长度计算出来了，然后继续考虑后面的逻辑；

2. 分析清楚递归体的逻辑，然后写出来。比如在上面的问题中，递归体的逻辑就是如何计算两边最长的距离；

3. 考虑清楚递归退出的边界条件。也就说，哪些地方应该写return。

注意到以上 3 点，在面对递归问题的时候，我们将总是有章可循。

求二进制数中 1 的个数

对于一个字节（8bit）的变量，求其二进制表示中“1”的个数，要求算法的执行效率尽可能地高。

分析与解法

大多数的读者都会有这样的反应：这个题目也太简单了吧，解法似乎也相当地单一，不会有太多的曲折分析或者峰回路转之处。那么面试者到底能用这个题目考察我们什么呢？事实上，在编写程序的过程中，根据实际应用的不同，对存储空间或效率的要求也不一样。比如在 PC 上的程序编写与在嵌入式设备上的程序编写就有很大的差别。我们可以仔细思索一下如何才能使效率尽可能地“高”。

【解法一】

可以举一个八位的二进制例子来进行分析。对于二进制操作，我们知道，除以一个 2，原来的数字将会减少一个 0。如果除的过程中有余，那么就表示当前位置有一个 1。

以 10 100 010 为例；

第一次除以 2 时，商为 1 010 001，余为 0。

第二次除以 2 时，商为 101 000，余为 1。

因此，可以考虑利用整型数据除法的特点，通过相除和判断余数的值来进行分析。于是有了如下的代码。

代码清单 2-1

```
int Count(int v)
{
int num = 0;
while(v)
    {
        if(v % 2 == 1)
    {
        num++;
    }
        v = v/ 2;
    }
return num;
}
```

【解法二】使用位操作

前面的代码看起来比较复杂。我们知道，向右移位操作同样也可以达到相除的目的。唯一不同之处在于，移位之后如何来判断是否有 1 存在。对于这个问题，再来看看一个八位的数字：10 100 001。

在向右移位的过程中，我们会把最后一位直接丢弃。因此，需要判断最后一位是否为 1，而“与”操作可以达到目的。可以把这个八位的数字与 00000001 进行“与”操作。如果结果为 1，则表示当前八位数的最后一位为 1，否则为 0。代码如下：

代码清单 2-2

```
int Count(int v)
{
    int num = 0;
    While(v)
    {
        num += v &0x01;
        v >>= 1;
    }
    return num;
}
```

【解法三】

位操作比除、余操作的效率高了很多。但是，即使采用位操作，时间复杂度仍为 O（$\log_2 v$），$\log_2 v$ 为二进制数的位数。那么，还能不能再降低一些复杂度呢？如果有办法让算法的复杂度只与“1”的个数有关，复杂度不就能进一步降低了吗？

同样用 10 100 001 来举例。如果只考虑和 1 的个数相关，那么，我们是否能够在每次判断中，仅与 1 来进行判断呢？

为了简化这个问题，我们考虑只有一个 1 的情况。例如：01 000 000。

如何判断给定的二进制数里面有且仅有一个 1 呢？可以通过判断这个数是否是 2 的整数次幂来实现。另外，如果只和这一个“1”进行判断，如何设计操作呢？我们知道的是，如果进行这个操作，结果为 0 或为 1，就可以得到结论。

如果希望操作后的结果为 0，01 000 000 可以和 00 111 111 进行“与”操作。

这样，要进行的操作就是 01 000 000 & (01 000 000 – 00 000 001) = 01 000 000 & 00 111 111 = 0。

因此就有了解法三的代码：

代码清单 2-3

```
int Count(int v)
{
    int num = 0;
    while(v)
    {
        v &= (v-1);
        num++;
    }
    return num;
}
```

【解法四】使用分支操作

解法三的复杂度降低到 $O(M)$，其中 M 是 v 中 1 的个数，可能会有人已经很满足了，只用计算 1 的位数，这样应该够快了吧。然而我们说既然只有八位数据，索性直接把 0~255 的情况都罗列出来，并使用分支操作，可以得到答案，代码如下：

代码清单 2-4

```
int Count(int v)
{
    int num = 0;
    switch (v)
    {
        case 0x0:
            num = 0;
            break;
        case 0x1:
        case 0x2:
        case 0x4:
        case 0x8:
        case 0x10:
        case 0x20:
        case 0x40:
        case 0x80:
            num = 1;
            break;
        case 0x3:
        case 0x6:
        case 0xc:
        case 0x18:
        case 0x30:
        case 0x60:
        case 0xc0:
            num = 2;
            break;
            //...
    }
    return num;
}
```

解法四看似很直接，但实际执行效率可能会低于解法二和解法三，因为分支语句的执行情况要看具体字节的值，如果 a =0，那自然在第 1 个 case 就得出了答案，但是如果 a =255，则要在最后一个 case 才得出答案，即在进行了 255 次比较操作之后！

看来，解法四不可取！但是解法四提供了一个思路，就是采用空间换时间的方法，罗列并直接给出值。如果需要快速地得到结果，可以利用空间或利用已知结论。这就好比已经知道计算 1+2+ … +N 的公式，在程序实现中就可以利用公式得到结论。

最后，得到解法五：算法中不需要进行任何的比较便可直接返回答案，这个解法在时间复杂度上应该能够让人高山仰止了。

【解法五】查表法

代码清单 2-5

```
/* 预定义的结果表 */
int countTable[256] =
{
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3,
       3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3,
       4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4,
       3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3,
       4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6,
```

```
        6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4,
        5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3,
        4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4,
        4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
        7, 6, 7, 7, 8
};
int Count(int v)
{
    //check parameter
    return countTable[v];
}
```

这是个典型的空间换时间的算法，把 0~255 中“1”的个数直接存储在数组中，*v* 作为数组的下标，countTable[*v*]就是 *v* 中“1”的个数。算法的时间复杂度仅为 *O*（1）。

在一个需要频繁使用这个算法的应用中，通过“空间换时间”来获取高的时间效率是一个常用的方法，具体的算法还应针对不同应用进行优化。

扩展问题

1. 如果变量是32位的DWORD，你会使用上述的哪一个算法，或者改进哪一个算法？

2. 另一个相关的问题，给定两个正整数（二进制形式表示）*A*和*B*，问把*A*变为*B*需要改变多少位（bit）？也就是说，整数*A* 和*B* 的二进制表示中有多少位是不同的？

《编程之美—微软技术面试心得》

IT从业人员“面试真题”。倡导： 思考力等于竞争力。

揭露微软面试内幕：从平常的问题入手，深入挖掘，考察动手能力和独到的思路。 解答面试疑惑： IT 企业重视什么样的能力、需要什么样的技术人才、如何甄别人才。

《编程之美—微软技术面试心得》文章节选：

一摞烙饼的排序

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？

分析与解法

这个排序问题非常有意思，首先我们要弄清楚解决问题的关键操作——“单手每次抓几块饼，全部颠倒”。

具体参看图 1-6：

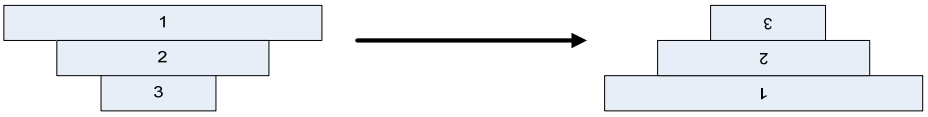


图 1-6　烙饼的翻转过程

每次我们只能选择最上方的一堆饼，一起翻转。而不能一张张地直接抽出来，然后进行插入，也不能交换任意两块饼子。这说明基本的排序办法都不太好用。那么怎么把这 *n* 个烙饼排好序呢？

由于每次操作都是针对最上面的饼，如果最底层的饼已经排序，那我们只用处理上面的 *n*−1 个烙饼。这样，我们可以再简化为 *n*−2、*n*−3，直到最上面的两个饼排好序。

【解法一】

我们用图 1-7 演示一下，为了把最大的烙饼摆在最下面，我们先把最上面的烙饼和最大的烙饼之间的烙饼翻转（1~4 之间），这样，最大的烙饼就在最上面了。接着，我们把所有烙饼翻转（4~5 之间），最大的烙饼就摆在最下面了。

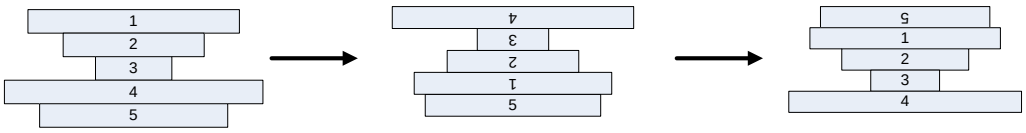


图 1-7　两次翻转烙饼，调整最大的烙饼到最底端

之后，我们对上面 $n-1$、$n-2$ 个饼重复这个过程就可以了。

那么，我们一共需要多少次翻转才能把这些烙饼给翻转过来呢？

首先，经过两次翻转可以把最大的烙饼翻转到最下面。因此，最多需要把上面的 $n-1$ 个烙饼依次翻转两次。那么，我们至多需要 $2(n-1)$ 次翻转就可以把所有烙饼排好序（因为第二小的烙饼排好的时候，最小的烙饼已经在最上面了）。

这样看来，单手翻转的想法是肯定可以实现的。我们进一步想想怎么减少翻转烙饼的次数吧。

怎样才能通过程序来搜索到一个最优的方案呢？

首先，通过每次找出最大的烙饼进行翻转是一个可行的解决方案。那么，这个方案是最好的一个吗？考虑这样一种情况，假如这堆烙饼中有好几个不同的部分相对有序，凭直觉来猜想，我们可以先把小一些的烙饼进行翻转，让其有序。这样会比每次翻转最大的烙饼要更快。

既然如此，有类似的方案可以达到目的吗？比如说，考虑每次翻转的时候，把两个本来应该相邻在烙饼尽可能地换到一起。这样，当所有的烙饼都换到一起之后，实际上就是完成排序了。（从这个意义上来说，每次翻最大烙饼的方案实质上就是每次把最大的和次大的交换到一起。）

在这样的基础之上，本能的一个想法就是穷举。只要穷举出所有可能的交换方案，那么，我们一定能够找到一个最优的方案。

沿着这个思路去考虑，我们自然就会使用动态规划或者递归的方法来进行实现了。可以从不同的翻转策略开始，比如说第一次先翻最小的，然后递归把所有的可能全部翻转一遍。这样，最终肯定是可以找到一个解的。

但是，既然是递归就一定有退出的条件。在这个过程中，第一个退出的条件肯定是所有的烙饼已经排好序。那么，有其他的吗？如果大家仔细想想就会发现到，既然 $2(n-1)$ 是一个最多的翻转次数。如果在算法中，需要翻转的次数多于 $2(n-1)$，那么，我们就应该放弃这个翻转算法，直接退出。这样，就能够减少翻转的次数。

从另外一个层面上来说，既然这是一个排序问题。我们也应该利用到排序的信息来进行处理。同样，在翻转的过程中，我们可以看看当前的烙饼数组的排序情况如何，然后利用这些信息来帮助减少翻转次数的判断过程。

下面是在前面讨论的基础之上形成的一个粗略的搜索最优方案的程序：

代码清单 1-8

```
#include <stdio.h>
/*********************************************************************/
//
// 烙饼排序实现
//
/*********************************************************************/
class CPrefixSorting
{
public:

    CPrefixSorting()
    {
        m_nCakeCnt = 0;
        m_nMaxSwap = 0;
    }

    //
    // 计算烙饼翻转信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Run(int* pCakeArray, int nCakeCnt)
    {
        Init(pCakeArray, nCakeCnt);

        m_nSearch = 0;
        Search(0);
    }

    //
    // 输出烙饼具体翻转的次数
    //
void Output()
{
    for(int i = 0; i < m_nMaxSwap; i++)
    {
        printf("%d ", m_arrSwap[i]);
    }

    printf("\n |Search Times| : %d\n", m_nSearch);
    printf("Total Swap times = %d\n", m_nMaxSwap);
}

private:

    //
    // 初始化数组信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Init(int* pCakeArray, int nCakeCnt)
    {
        Assert(pCakeArray != NULL);
        Assert(nCakeCnt > 0);

        m_nCakeCnt = n;
```

```
// 初始化烙饼数组
    m_CakeArray = new int[m_nCakeCnt];
    Assert(m_CakeArray != NULL);
    for(int i = 0; i < m_nCakeCnt; i++)
    {
        m_CakeArray[i] = pCakeArray[i];
    }

    // 设置最多交换次数信息
    m_nMaxSwap = UpBound(m_nCakeCnt);

    // 初始化交换结果数组
    m_SwapArray = new int[m_nMaxSwap];
    Assert(m_SwapArray != NULL);

    // 初始化中间交换结果信息
    m_ReverseCakeArray = new int[m_nCakeCnt];
    for(i = 0; i < m_nCakeCnt; i++)
    {
        m_ReverseCakeArray[i] = m_CakeArray[i];
    }
    m_ReverseCakeArraySwap = new int[m_nMaxSwap];
}


//
// 寻找当前翻转的上界
//
//
int UpBound(int nCakeCnt)
{
    return nCakeCnt*2;
}

//
// 寻找当前翻转的下界
//
//
int LowerBound(int* pCakeArray, int nCakeCnt)
{
    int t, ret = 0;

    // 根据当前数组的排序信息情况来判断最少需要交换多少次
    for(int i = 1; i < nCakeCnt; i++)
    {
        // 判断位置相邻的两个烙饼，是否为尺寸排序上相邻的
        t = pCakeArray[i] - pCakeArray[i-1];
        if((t == 1) || (t == -1))
        {
        }
        else
        {
            ret++;
        }
    }
    return ret;
}

// 排序的主函数
```

```
void Search(int step)
{
    int i, nEstimate;

    m_nSearch++;

    // 估算这次搜索所需要的最小交换次数
    nEstimate = LowerBound(m_ReverseCakeArray, m_nCakeCnt);
    if(step + nEstimate > m_nMaxSwap)
        return;

    // 如果已经排好序，即翻转完成，输出结果
    if(IsSorted(m_ReverseCakeArray, m_nCakeCnt))
    {
        if(step < m_nMaxSwap)
        {
            m_nMaxSwap = step;
            for(i = 0; i < m_nMaxSwap; i++)
                m_arrSwap[i] = m_ReverseCakeArraySwap[i];
        }
        return;
    }

    // 递归进行翻转
    for(i = 1; i < m_nCakeCnt; i++)
    {
        Revert(0, i);
        m_ReverseCakeArraySwap[step] = i;
        Search(step + 1);
        Revert(0, i);
    }
}

//
// true : 已经排好序
// false : 未排序
//
bool IsSorted(int* pCakeArray, int nCakeCnt)
{
    for(int i = 1; i < nCakeCnt; i++)
    {
        if(pCakeArray[i-1] > pCakeArray[i])
        {
            return false;
        }
    }
    return true;
}

//
// 翻转烙饼信息
//
void Revert(int nBegin, int nEnd)
{
Assert(nEnd > nBegin);
    int i, j, t;

    // 翻转烙饼信息
    for(i = nBegin, j = nEnd; i < j; i++, j--)
    {
```

```
            t = m_ReverseCakeArray[i];
            m_ReverseCakeArray[i] = m_ReverseCakeArray[j];
            m_ReverseCakeArray[j] = t;
        }
    }

private:

    int* m_CakeArray;   // 烙饼信息数组
    int m_nCakeCnt;     // 烙饼个数
    int m_nMaxSwap;     // 最多交换次数。根据前面的推断，这里最多为m_nCakeCnt * 2
    int* m_SwapArray;   // 交换结果数组

    int* m_ReverseCakeArray;        // 当前翻转烙饼信息数组
    int* m_ReverseCakeArraySwap;    // 当前翻转烙饼交换结果数组
    int m_nSearch;                  // 当前搜索次数信息
};
```

当烙饼不多的时候，我们已经可以很快地找出最优的翻转方案。

我们还有办法来对这个程序进行优化，使得能更快地找出最优方案吗？

我们已经知道怎么构造一个可行的翻转方案，所以最优的方案肯定不会比这个差。这个就是我们程序中的上界（UpperBound），就是说，我们感兴趣的最优方案最差也就是我们刚构造出来的方案了。如果我们能够找到一个更好的构造方案，我们的搜索空间就会继续缩小，因为我们一开始就设 m_nMinSwap 为 UpperBound，而程序中有一个剪枝：

```
nEstimate = LowerBound(m_tArr, m_n);
if(step + nEstimate > m_nMinSwap)
    return;
```

m_nMinSwap 越小，那么这个剪枝条件就越容易满足，更多的情况就不需要再去搜索。当然，程序也就能更快地找出最优方案。

仔细分析上面的剪枝条件，在到达 m_tArr 状态之前，我们已经翻转了 step 次，nEstimate 是在当前这个状态我们至少还要翻转多少次才能成功的次数。如果 step+nEstimate 大于 m_nMinSwap，也就说明从当前状态继续下去，m_nMinSwap 次我们也不能排好所有烙饼。那么，当然就没有必要再继续了。因为继续下去得到的方案不可能比我们已经找到的好。

显然，如果 nEstimate 越大，剪枝条件越容易被满足。而这正是我们希望的。

结合上面两点，我们希望 UpperBound 越小越好，而下界（LowerBound）越大越好。假设如果有神仙指点，你只要告诉神仙你当前的状态，他就能告诉你最少需要多少次翻转。这样的话，我们可以花费 $O(N^2)$ 的时间得到最优的方案。但是，现实中，没有这样的神仙。我们只能尽可能地减小 UpperBound，增加 LowerBound，从而减少需要搜索的空间。

利用上面的程序，做一个简单的比较。

对于一个输入，10 个烙饼，从上到下，烙饼半径分别为 3, 2, 1, 6, 5, 4, 9, 8, 7, 0。对应上面程序的输入为：

```
10
3 2 1 6 5 4 9 8 7 0
```

如果 LowerBound 在任何状态都为 0，也就是我们太懒了，不想考虑那么多。当然任意状态下，你至少需要 0 次翻转才能排好序。这样，上面的程序 Search 函数被调用了 575 225 200 次。

但是如果把 LowerBound 稍微改进一下（如上面程序中所计算的方法估计），程序则只需要调用 172 126 次 Search 函数便可以得到最优方案：

```
6
4 8 6 8 4 9
```

程序中的下界是怎么估计出来的呢？

每一次翻转我们最多使得一个烙饼与大小跟它相邻的烙饼排到一起。如果当前状态 *n* 个烙饼中，有 *m* 对相邻的烙饼它们的半径不相邻，那么我们至少需要 *m* 次才能排好序。

从上面的例子，大家都会发现改进上界和下界，好处可不少。我想不用多说，大家肯定想继续优化上界和下界的估计吧。

除了上界和下界的改进，还有什么办法可以提高搜索效率吗？如果我们翻了若干次之后，又回到一个已经出现过的状态，我们还值得继续从这个状态开始搜索吗？我们怎样去检测一个状态是否出现过呢？

读者也许不相信，比尔盖茨在上大学的时候也研究过这个问题，并且发表过论文。你不妨跟盖茨的结果[1]比比吧。

[1] Gates, W. and Papadimitriou, C. "Bounds for Sorting by Prefix Reversal." Discrete Mathematics. 27, 47~57, 1979. 据说这是 Bill Gates 发表的唯一学术论文。

轻松一刻：看了《编程之美》后面试微软的经历

今天我去面试，前面答得还不错，最后面试官问：看了《编程之美》了么？

我回答：看了。

问：怪不得，书带来了么？

我从书包里拿出皱巴巴的书。

问：为什么书这么皱？你在上面乱画了什么？好像还被水泡过。。。

我想起来面试的时候要诚实，就鼓起勇气说：我有一次做题的时候趴在上面睡着了，然后流了很多口水。。。

面试官想了想，说：比尔开始写程序的时候，也是趴在电脑上睡着了。。。 你明天就来上班吧。

好书推荐《移山之道——VSTS 软件开发指南》

《编程之美——微软技术面试心得》作者之一邹欣 力作。

经典的项目管理之道，讲述在中国的产业背景下，

中小型 IT 企业项目开发和管理的实际案例

不管你用不用 VSTS，这本书都对你很有用！

《移山之道——VSTS 软件开发指南》社区网站 http://yishan.cc/

China-pub 地址：http://www.china-pub.com/35373

《移山之道——VSTS 软件开发指南》读者评论：

全书内容写的很不错 ,以一种别具匠心的风格展示软件生命周期各个阶段管理以及应用。

内容也很简洁没有那么多的理论 ,比较适合要步入 PM 或者其他技术人员学习参考 , 尤其是里面的一些看似很短的对话却体现了一个很深刻的道理。希望作者能多写些这方面的著作。支持原创作品。　　读者： thirston_bill

其实里面讲 VSTS 部分并不比我们公司内部讲师讲的详细 ,但作者说明了为什么要这么用，另外作者让我弄明白了什么是 MSF，也让我有马上将自己项目移至 VSTS 服务器上去的冲动。

目前项目还是处于很松弛的管理管理状态 , 代码管理也不正规 , 而该书还提出了一个在微软敏捷 4 之上的新的方法——移山方法。我就打算用这个管理我自己的项目。　　读者： kiciro

因为在书店中看了一看这书, 感觉很不错, 所以买了它。

当然我买这本书的目的并不是在学习 VSTS 的用法， 因为我觉得在这本书中有很多的技巧与道理， 出于学习这些技巧与道理的角度，我觉得不管你用不用 VSTS 都可以来看看这本书。　　读者：ooyuan

经典的一本项目管理之道： 故事化的情节，实战型的经历，让你不知不觉中了解整个项目开发的过程 , 就当看小说一样 , 只要你能翻到书的最后一页 , 你就受益非浅。　　豆瓣读者

瓷砖覆盖地板

某年夏天，位于希格玛大厦四层的微软亚洲研究院对办公楼的天井进行了一次大规模的装修。原来的地板铺有 $N \times M$ 块正方形瓷砖，这些瓷砖都已经破损老化了，需要予以更新。装修工人们在前往商店选购新的瓷砖时，发现商店目前只供应长方形的瓷砖，现在的一块长方形瓷砖相当于原来的两块正方形瓷砖，工人们拿不定主意该买多少了，读者朋友们请帮忙分析一下：能否用 1×2 的瓷砖去覆盖 $N \times M$ 的地板呢？

分析与解法

$N \times M$ 的地板有如下几种可能：

1. 如果$N = 1$，M为偶数的话，显然，1×2的瓷砖可以覆盖1×M的地板，在这种情况下，共需要$M/2$块瓷砖。

2. 如果$N \times M$为奇数，也就是N和M都为奇数，则肯定不能用1×2的瓷砖去覆盖它。

   证明：假设能够用k 块1×2的瓷砖去覆盖$N \times M$（N、M都为奇数）的地板，设每块瓷砖的面积为1×2，那么总的地板面积就为$2k$——必为偶数，又因为N、M都为奇数，也就是$N \times M$的地板面积肯定为奇数，与1×2的瓷砖所能覆盖的面积相矛盾，所以肯定不能用1×2的瓷砖去覆盖它。

3. N和M中至少有一个为偶数，不妨设M为偶数，那么既然我们可以用1×2的地板覆盖1×M的地板，也就可以简单地重复N次覆盖1×M的地板的做法，即可覆盖$N \times M$的地板。

扩展问题

求用1×2的瓷砖覆盖2×M的地板有几种方式？

设用 1×2 的瓷砖覆盖 2×M 的地板有 $F(M)$ 种方式，其中 F 为 M 的函数，那么第一块瓷砖的放法如图 4-1、图 4-2 所示：

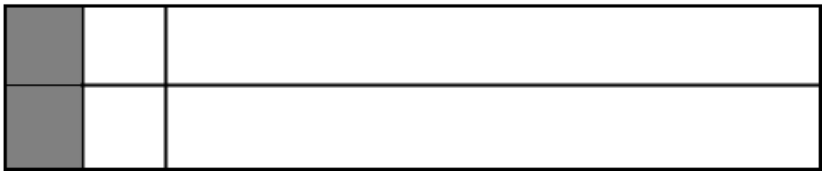


图 4-1　第一块瓷砖竖着放，如图中阴影部分所示

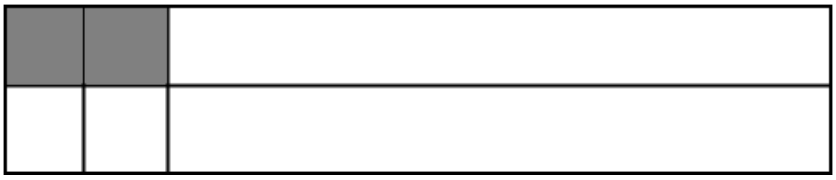


图 4-2　第一块瓷砖横着放，如图中阴影部分所示

通过对图 4-1、图 4-2 的简单分析，我们知道第一块瓷砖的放法，要么是竖着放，要么是横着放。

当第一块瓷砖竖着放的时候，问题转换成求用 1×2 的瓷砖覆盖剩下的 2×（M−1）的方式，即 F（M−1）。

当第一块瓷砖横着放的时候（必有另一块瓷砖放在其正下方，如图中阴影所示），问题转换成求用 1×2 的瓷砖覆盖剩下的 2×（M−2）的方式，即 F（M−2）。

在求 F（M−1）和 F（M−2）时，由于第一列地板的覆盖方式已经不同，F（M−1）种覆盖方式和 F（M−2）中覆盖方式没有重叠，故 F（M）= F（M−1）+F（M−2），其中，F（1）=1，F（2）=2。

读者朋友们不妨思考一下这个问题的推广形式又该如何解答：

4. 用1×2的瓷砖覆盖8×8的地板，有多少种方式呢？如果是N×M的地板呢？

5. 用p×q的瓷砖能够覆盖M×N的地板吗？

精确表达浮点数

在计算机中，使用 float 或者 double 来存储小数是不能得到精确值的。如果你希望得到精确计算结果，最好是用分数形式来表示小数。有限小数或者无限循环小数都可以转化为分数。比如：

0.9 = 9/10

0.333（3）= 1/3（括号中的数字表示是循环节）

当然一个小数可以用好几种分数形式来表示。如：

0.333（3）= 1/3 = 3/9

给定一个有限小数或者无限循环小数，你能否以分母最小的分数形式来返回这个小数呢？如果输入为循环小数，循环节用括号标记出来。下面是一些可能的输入数据，如 0.3、0.30、0.3（000）、0.3333（3333）、……

分析与解法

拿到这样一个问题，我们往往会从最简单的情况入手，因为所有的小数都可以分解成一个整数和一个纯小数之和，不妨只考虑大于 0，小于 1 的纯小数，且暂时不考虑分子和分母的约分，先设法将其表示为分数形式，然后再进行约分。题目中输入的小数，要么为有限小数 $X=0.a_1a_2\cdots a_n$，要么为无限循环小数 $X=0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$，$X$ 表示式中的字母 $a_1a_2\cdots a_n$，$b_1b_2\cdots b_m$ 都是 0~9 的数字，括号部分（$b_1b_2\cdots b_m$）表示循环节，我们需要处理的就是以上两种情况。

对于有限小数 $X=0.a_1a_2\cdots a_n$ 来说，这个问题比较简单，X 就等于（$a_1a_2\cdots a_n$）$/10^n$。

对于无限循环小数 $X=0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$ 来说，其复杂部分在于小数点后同时有非循环部分和循环部分，我们可以做如下的转换：

$X = 0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$

$\Rightarrow 10^n * X = a_1a_2\cdots a_n.(b_1b_2\cdots b_m)$

$\Rightarrow 10^n * X = a_1a_2\cdots a_n+0.(b_1b_2\cdots b_m)$

$\Rightarrow X = (a_1a_2\cdots a_n+0.(b_1b_2\cdots b_m))/10^n$

对于整数部分 $a_1a_2\cdots a_n$，不需要做额外处理，只需要把小数部分转化为分数形式再加上这个整数即可。对于后面的无限循环部分，可以采用如下方式进行处理：

令 $Y=0.b_1b_2\cdots b_m$，那么

$10^m * Y = b_1b_2\cdots b_m.(b_1b_2\cdots b_m)$

$\Rightarrow 10^m * Y = b_1b_2\cdots b_m+0.(b_1b_2\cdots b_m)$

$\Rightarrow 10^m * Y - Y = b_1b_2\cdots b_m$

$\Rightarrow Y = b_1b_2\cdots b_m / (10^m-1)$

将 Y 代入前面的 X 的等式可得：

$X = (a_1a_2\cdots a_n+Y)/10^n$

$= (a_1a_2\cdots a_n + b_1b_2\cdots b_m/(10^m-1))/10^n$

$= ((a_1a_2\cdots a_n)*(10^m-1) + (b_1b_2\cdots b_m))/((10^m-1)*10^n)$

至此，便可以得到任意一个有限小数或无限循环小数的分数表示，但是此时分母未必是最简的，接下来的任务就是让分母最小，即对分子和分母进行约分，

这个相对比较简单。对于任意一个分数 *A*/*B*，可以简化为（*A* /Gcd（*A*, *B*））/（*B* /Gcd（*A*, *B*）），其中 Gcd 函数为求 *A* 和 *B* 的最大公约数，这就涉及本书中的算法（2.7 节“最大公约数问题”），其中有很巧妙的解法，请读者阅读具体的章节，这里就不再赘述。

综上所述，先求得小数的分数表示方式，再对其分子分母进行约分，便能够得到分母最小的分数表现形式。

例如，对于小数 0.3（33），根据上述方法，可以转化为分数：

0.3（33）
=（3 *（10^2−1）+ 33）/（（10^2−1）*10）
=（3*99+33）/990
= 1 / 3

对于小数 0. 285714（285714），我们也可以算出：
0. 285714（285714）
=（285714 *（10^6−1）+ 285714）/（（10^6−1）*10^6）
=（285714*999999 +285714）/ 999999000000
= 285714 / 999999
= 2/7

《编程之美——微软技术面试心得》

《编程之美——微软技术面试心得》（http://www.china-pub.com/38070）是微软亚洲研究院技术创新组研发主管邹欣继《移山之道——VSTS 软件开发指南》后的最新力作。它传达给读者：微软重视什么样的能力，需要什么样的人才。但它更深层的意义在于引导读者思考，提倡一种发现问题、解决问题的思维方式，充分挖掘编程的乐趣，展示编程之美。本书 3 月份上市。网上讨论和解答在：www.msra.cn/bop

题目《让 CPU 占用率曲线听你指挥》

问题

写一个程序，让用户来决定 Windows 任务管理器（Task Manager）的 CPU 占用率。程序越精简越好，计算机语言不限。例如，可以实现下面三种情况：

1. CPU的占用率固定在50%，为一条直线；

2. CPU的占用率为一条直线，但是具体占用率由命令行参数决定（参数范围1~ 100）；

3. CPU的占用率状态是一个正弦曲线。

分析与解法[1]

有一名学生写了如下的代码：

```
while (true)
{
   if (busy)
      i++;
   else

}
```

然后她就陷入了苦苦思索：else 干什么呢？怎么才能让电脑不做事情呢？CPU 使用率为 0 的时候，到底是什么东西在用 CPU？另一名学生花了很多时间构想如何“深入内核，以控制 CPU 占用率”——可是事情真的有这么复杂么？

MSRA TTG（Microsoft Research Asia, Technology Transfer Group）的一些实习生写了各种解法，他们写的简单程序可以达到如图 1-1 所示的效果。

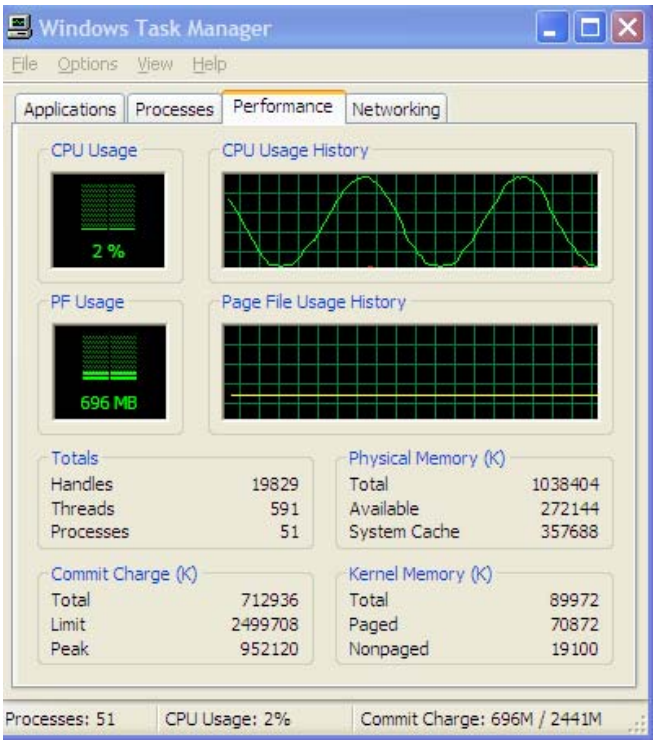


图 1-1 编码控制 CPU 占用率呈现正弦曲线形态

看来这并不是不可能完成的任务。让我们仔细地回想一下写程序时曾经碰到的问题，如

[1] 作者注：当面试的同学听到这个问题的时候，很多人都有点意外。我把我的笔记本电脑交给他们说，这是开卷考试，你可以上网查资料，干什么都可以。大部分面试者在电脑上的第一个动作就是上网搜索“CPU 控制 50%”这样的关键字，当然没有找到什么直接的结果。不过这本书出版以后，情况可能就不一样了。

果我们不小心写了一个死循环，CPU 占用率就会跳到最高，并且一直保持 100%。我们也可以打开任务管理器[2]，实际观测一下它是怎样变动的。凭肉眼观察，它大约是 1 秒钟更新一次。一般情况下，CPU 使用率会很低。但是，当用户运行一个程序，执行一些复杂操作的时候，CPU 的使用率会急剧升高。当用户晃动鼠标时，CPU 的使用率也有小幅度的变化。

那当任务管理器报告 CPU 使用率为 0 的时候，谁在使用 CPU 呢？通过任务管理器的“进程（Process）”一栏可以看到，System Idle Process 占用了 CPU 空闲的时间——这时候大家该回忆起在“操作系统原理”这门课上学到的一些知识了吧。系统中有那么多进程，它们什么时候能“闲下来”呢？答案很简单，这些程序或者在等待用户的输入，或者在等待某些事件的发生（`WaitForSingleObject()`），或者进入休眠状态（通过 `Sleep()`来实现）。

在任务管理器的一个刷新周期内，CPU 忙（执行应用程序）的时间和刷新周期总时间的比率，就是 CPU 的占用率，也就是说，任务管理器中显示的是每个刷新周期内 CPU 占用率的统计平均值。因此，我们写一个程序，让它在任务管理器的刷新期间内一会儿忙，一会儿闲，然后通过调节忙/闲的比例，就可以控制任务管理器中显示的 CPU 占用率。

【解法一】简单的解法

步骤 1 要操纵 CPU 的 usage 曲线，就需要使 CPU 在一段时间内（根据 Task Manager 的采样率）跑 busy 和 idle 两个不同的 loop，从而通过不同的时间比例，来获得调节 CPU Usage 的效果。

步骤 2 Busy loop 可以通过执行空循环来实现，idle 可以通过 `Sleep()`来实现。

问题的关键在于如何控制两个 loop 的时间，方法有二：

Sleep一段时间，然后以for循环n次，估算n的值。

那么对于一个空循环 `for(i = 0; i < n; i++)`；又该如何来估算这个最合适的 n 值呢？我们都知道 CPU 执行的是机器指令，而最接近于机器指令的语言是汇编语言，所以我们可以先把这个空循环简单地写成如下汇编代码后再进行分析：

```
loop:
mov dx i        ;将i置入dx寄存器
inc dx          ;将dx寄存器加1
mov i dx        ;将dx中的值赋回i
cmp i n         ;比较i和n
jl loop         ;i 小于 n 时则重复循环
```

假设这段代码要运行的 CPU 是 P4 2.4Ghz（2.4 * 10 的 9 次方个时钟周期每秒）。现代 CPU 每个时钟周期可以执行两条以上的代码，那么我们就取平均值两条，于是让（2 400 000 000 * 2）/5=960 000 000（循环/秒），也就是说 CPU 1 秒钟可以运行这个空循环 960 000 000 次。不过我们还是不能简单地将 n = 60 000 000，然后 `Sleep(1000)`了事。如果我们让 CPU 工

[2] 如果应聘者从来没有琢磨过任务管理器，那还是不要在简历上说“精通 Windows”为好。

作 1 秒钟 ,然后休息 1 秒钟 ,波形很有可能就是锯齿状的——先达到一个峰值(大于>50%) , 然后跌到一个很低的占用率。

我们尝试着降低两个数量级，令 $n = 9\ 600\ 000$，而睡眠时间相应改为 10 毫秒（Sleep(10)）。用 10 毫秒是因为它不大也不小，比较接近 Windows 的调度时间片。如果选得太小（比如 1 毫秒），则会造成线程频繁地被唤醒和挂起，无形中又增加了内核时间的不确定性影响。最后我们可以得到如下代码：

代码清单 1-1

```
int main()
{
   for(;;)
   {
      for(int i = 0; i < 9600000; i++);
      Sleep(10);
   }
   return 0;
}
```

在不断调整 9 600 000 的参数后，我们就可以在一台指定的机器上获得一条大致稳定的 50% CPU 占用率直线。

使用这种方法要注意两点影响：

1. 尽量减少sleep/awake的频率，如果频繁发生，影响则会很大，因为此时优先级更高的操作系统内核调度程序会占用很多CPU运算时间。

2. 尽量不要调用system call（比如I/O这些privilege instruction），因为它也会导致很多不可控的内核运行时间。

该方法的缺点也很明显：不能适应机器差异性。一旦换了一个 CPU，我们又得重新估算 n 值。有没有办法动态地了解 CPU 的运算能力，然后自动调节忙/闲的时间比呢？请看下一个解法。

【解法二】使用 GetTickCount()和 Sleep()

我们知道 GetTickCount()可以得到“系统启动到现在”的毫秒值，最多能够统计到 49.7 天。另外，利用 Sleep()函数，最多也只能精确到 1 毫秒。因此，可以在“毫秒”这个量级做操作和比较。具体如下：

利用 GetTickCount()来实现 busy loop 的循环，用 Sleep()实现 idle loop。伪代码如下：

代码清单 1-2

```
int busyTime = 10;  //10 ms
int idleTime = busyTime;  //same ratio will lead to 50% cpu usage
```

```
Int64 startTime = 0;
while (true)
{
    startTime = GetTickCount();
    // busy loop的循环
    while ((GetTickCount() - startTime) <= busyTime) ;

    //idle loop
    Sleep(idleTime);
}
```

这两种解法都是假设目前系统上只有当前程序在运行，但实际上，操作系统中有很多程序都会在不同时间执行各种各样的任务，如果此刻其他进程使用了 10% 的 CPU，那我们的程序应该只能使用 40%的 CPU（而不是机械地占用 50%），这样可达到 50%的效果。

怎么做呢？

我们得知道"当前 CPU 占用率是多少"，这就要用到另一个工具来帮忙——Perfmon.exe。

Perfmon 是从 Windows NT 开始就包含在 Windows 服务器和台式机操作系统的管理工具组中的专业监视工具之一（如图 1-2 所示）。Perfmon 可监视各类系统计数器，获取有关操作系统、应用程序和硬件的统计数字。Perfmon 的用法相当直接，只要选择您所要监视的对象（比如：处理器、RAM 或硬盘），然后选择所要监视的计数器（比如监视物理磁盘对象时的平均队列长度）即可。还可以选择所要监视的实例，比如面对一台多 CPU 服务器时，可以选择监视特定的处理器。

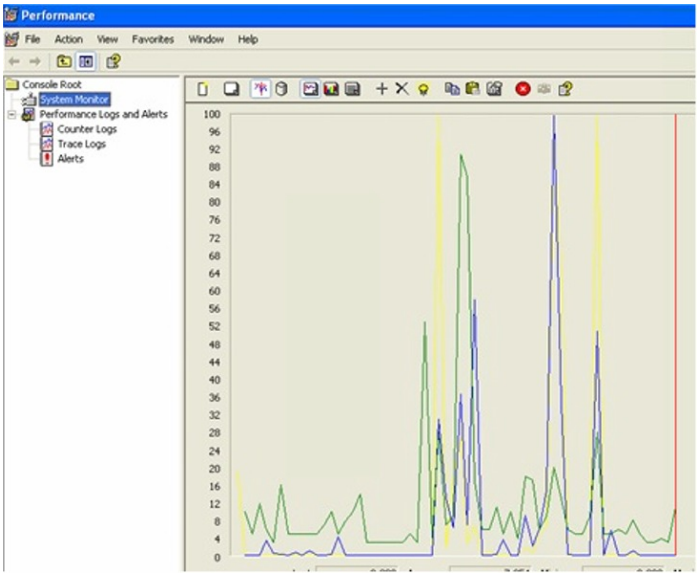


图 1-2 系统监视器（Perfmon）

我们可以写程序来查询 Perfmon 的值，Microsoft .Net Framework 提供了 PerformanceCounter()这一类型，从而可以方便地拿到当前各种计算机性能数据，包括 CPU 的使用率。例如下面这个程序——

【解法三】能动态适应的解法

代码清单 1-3

```
//C# code
static void MakeUsage(float level)
{
   PerformanceCounter p = new PerformanceCounter("Processor", "% Processor Time",
     "_Total");

   while (true)
   {
      if (p.NextValue() > level)
         System.Threading.Thread.Sleep(10);
    }
}
```

可以看到，上面的解法能方便地处理各种 CPU 使用率参数。这个程序可以解答前面提到的问题 2。

有了前面的积累，我们应该可以让任务管理器画出优美的正弦曲线了，见下面的代码。

【解法四】正弦曲线

代码清单 1-4

```
//C++ code  to make task manager generate sine graph
#include "Windows.h"
#include "stdlib.h"
#include "math.h"

const double SPLIT = 0.01;
const int COUNT = 200;
const double PI = 3.14159265;
const int INTERVAL = 300;

int _tmain(int argc, _TCHAR* argv[])
{
    DWORD busySpan[COUNT];  //array of busy times
    DWORD idleSpan[COUNT];  //array of idle times
    int half = INTERVAL / 2;
    double radian = 0.0;
    for(int i = 0; i < COUNT; i++)
    {
        busySpan[i] = (DWORD)(half + (sin(PI * radian) * half));
        idleSpan[i] = INTERVAL - busySpan[i];
        radian += SPLIT;
    }

    DWORD startTime = 0;
    int j = 0;
    while (true)
    {
        j = j % COUNT;
        startTime = GetTickCount();
        while ((GetTickCount() - startTime) <= busySpan[j]) ;
        Sleep(idleSpan[j]);
        j++;
    }
    return 0;
}
```

讨论

如果机器是多 CPU，上面的程序会出现什么结果？如何在多个 CPU 时显示同样的状态？例如，在双核的机器上，如果让一个单线程的程序死循环，能让两个 CPU 的使用率达到 50%的水平么？为什么？

多 CPU 的问题首先需要获得系统的 CPU 信息。可以使用 GetProcessorInfo()获得多处理器的信息,然后指定进程在哪一个处理器上运行。其中指定运行使用的是 SetThreadAffinityMask()函数。

另外，还可以使用 RDTSC 指令获取当前 CPU 核心运行周期数。

在 x86 平台上定义函数：

```
inline __int64 GetCPUTickCount()
{
    __asm
    {
       rdtsc;
    }
}
```

在 x64 平台上定义：

```
#define GetCPUTickCount() __rdtsc()
```

使用 CallNtPowerInformation API 得到 CPU 频率，从而将周期数转化为毫秒数，例如：

代码清单 1-5

```
_PROCESSOR_POWER_INFORMATION info;

CallNTPowerInformation(11,   //query processor power information
   NULL,               //no input buffer
   0,                 //input buffer size is zero
   &info,             //output buffer
   Sizeof(info));    //outbuf size

   __int64 t_begin = GetCPUTickCount();

   //do something

   __int64 t_end = GetCPUTickCount();
   double millisec = ((double)t_end –
      (double)t_begin)/(double)info.CurrentMhz;
```

RDTSC 指令读取当前 CPU 的周期数，在多 CPU 系统中，这个周期数在不同的 CPU 之间基数不同，频率也有可能不同。用从两个不同的 CPU 得到的周期数作计算会得出没有意义的值。如果线程在运行中被调度到了不同的 CPU，就会出现上述情况。可用 SetThreadAffinityMask 避免线程迁移。另外 ,CPU 的频率会随系统供电及负荷情况有所调整。

总结

能帮助你了解当前线程/进程/系统效能的 API 大致有以下这些：

1. Sleep()——这个方法能让当前线程“停”下来。
2. WaitForSingleObject()——自己停下来，等待某个事件发生
3. GetTickCount()——有人把 Tick 翻译成“嘀嗒”，很形象。
4. QueryPerformanceFrequency()、QueryPerformanceCounter()——让你访问到精度更高的 CPU 数据。
5. timeGetSystemTime()——是另一个得到高精度时间的方法。
6. PerformanceCounter——效能计数器。
7. GetProcessorInfo()/SetThreadAffinityMask()。遇到多核的问题怎么办呢？这两个方法能够帮你更好地控制 CPU。
8. GetCPUTickCount()。想拿到 CPU 核心运行周期数吗？用用这个方法吧。

了解并应用了上面的 API，就可以考虑在简历中写上“精通 Windows”了。

编程判断两个链表是否相交

给出两个单向链表的头指针（如图 3-8 所示），比如 h_1、h_2，判断这两个链表是否相交。这里为了简化问题，我们假设两个链表均不带环。

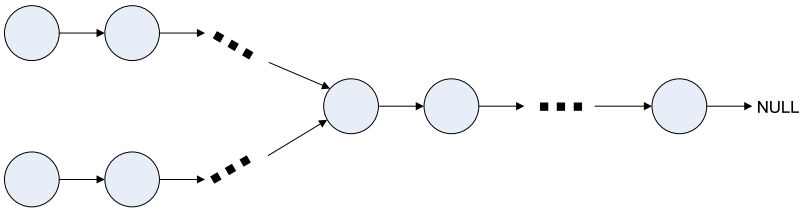


图 3-8　链表相交示意图

分析与解法

这样的一个问题，也许我们平时很少考虑。但在一个大的系统中，如果出现两个链表相交的情况，而且释放了其中一个链表的所有节点，那样就会造成信息的丢失，并且另一个与之相交的链表也会受到影响，这是我们不希望看到的。在特殊的情况下，的确需要出现相交的两个链表，我们希望在释放一个链表之前知道是否有其他链表跟当前这个链表相交。

【解法一】直观的想法

看到这个问题，我们的第一个想法估计都是，“不管三七二十一”，先判断第一个链表的每个节点是否在第二个链表中。这种方法的时间复杂度为 $O(\text{Length}(h_1) * \text{Length}(h_2))$。可见，这种方法很耗时间。

【解法二】利用计数的方法

很容易想到，如果两个链表相交，那么这两个链表就会有共同的节点。而节点地址又是节点的唯一标识。所以，如果我们能够判断两个链表中是否存在地址一致的节点，就可以知道这两个链表是否相交。一个简单的做法是对第一个链表的节点地址进行 hash 排序，建立 hash 表，然后针对第二个链表的每个节点的地址查询 hash 表，如果它在 hash 表中出现，那么说明第二个链表和第一个链表有共同的节点。这个方法的时间复杂度为 $O(\max(\text{Length}(h_1) + \text{Length}(h_2)))$。但是它同时需要附加 $O(\text{Length}(h_1))$的存储空间，以存储哈希表。虽然这样做减少了时间复杂度，但是是以增加存储空间为代价的。是否还有更好的方法呢，既能够以线性时间复杂度解决问题，又能减少存储空间？

【解法三】

由于两个链表都没有环，我们可以把第二个链表接在第一个链表后面，如果得到的链表有环，则说明这两个链表相交。否则，这两个链表不相交（如图 3-9 所示）。这样我们就把问题转化为判断一个链表是否有环。

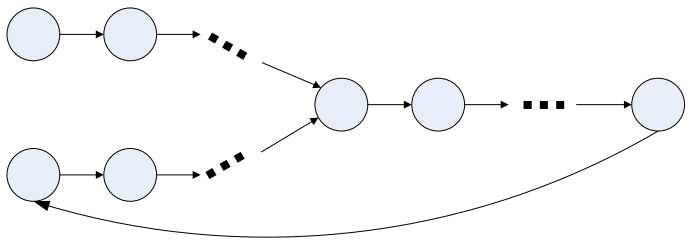


图 3-9　链表有环的情况

判断一个链表是否有环，也不是一个简单的问题，但是需要注意的是，在这里如果有环，则第二个链表的表头一定在环上，我们只需要从第二个链表开始遍历，看是否会回到起始点就可以判断出来。最后，当然可别忘了恢复原来的状态，去掉从第一个链表到第二个链表表头的指向。

这个方法总的时间复杂度也是线性的，但只需要常数的空间。

【解法四】

仔细观察题目中的图示，如果两个没有环的链表相交于某一节点的话，那么在这个节点之后的所有节点都是两个链表所共有的。那么我们能否利用这个特点简化我们的解法呢？困难在于我们并不知道哪个节点必定是两个链表共有的节点（如果它们相交的话）。进一步考虑“如果两个没有环的链表相交于某一节点的话，那么在这个节点之后的所有节点都是两个链表共有的”这个特点，我们可以知道，如果它们相交，则最后一个节点一定是共有的。而我们很容易能得到链表的最后一个节点，所以这成了我们简化解法的一个主要突破口。

先遍历第一个链表，记住最后一个节点。然后遍历第二个链表，到最后一个节点时和第一个链表的最后一个节点做比较，如果相同，则相交，否则，不相交。这样我们就得到了一个时间复杂度，它为 $O((\text{Length}(h_1) + \text{Length}(h_2))$，而且只用了一个额外的指针来存储最后一个节点。这个方法比解法三更胜一筹。

扩展问题

1. 如果链表可能有环呢?上面的方法需要怎么调整？

2. 如果我们需要求出两个链表相交的第一个节点呢？

计算字符串的相似度

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度。我们定义了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，都仅需要一次操作。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出它们的相似度呢？

分析与解法

不难看出，两个字符串的距离肯定不超过它们的长度之和（我们可以通过删除操作把两个串都转化为空串）。虽然这个结论对结果没有帮助，但至少可以知道，任意两个字符串的距离都是有限的。

我们还是应该集中考虑如何才能把这个问题转化成规模较小的同样的问题。如果有两个串 *A=xabcdae* 和 *B=xfdfa* ,它们的第一个字符是相同的 ,只要计算 *A*[2, …, 7] = *abcdae* 和 *B*[2, …, 5] = *fdfa* 的距离就可以了。但是如果两个串的第一个字符不相同，那么可以进行如下的操作（len*A* 和 len*B* 分别是 *A* 串和 *B* 串的长度）：

1．删除*A*串的第一个字符，然后计算*A*[2, …, len*A*]和*B*[1, …, len*B*]的距离。

2．删除*B*串的第一个字符，然后计算*A*[1, …, len*A*]和*B*[2, …, len*B*]的距离。

3．修改*A*串的第一个字符为*B*串的第一个字符，然后计算*A*[2, …, len*A*]和*B*[2, …, len*B*]的距离。

4．修改*B*串的第一个字符为*A*串的第一个字符，然后计算*A*[2, …, len*A*]和*B*[2, …, len*B*]的距离。

5．增加*B*串的第一个字符到*A*串的第一个字符之前，然后计算*A*[1, …, len*A*]和*B*[2, …, len*B*]的距离。

6．增加*A*串的第一个字符到*B*串的第一个字符之前，然后计算*A*[2, …, len*A*]和*B*[1, …, len*B*]的距离。

在这个题目中，我们并不在乎两个字符串变得相等之后的字符串是怎样的。所以，可以将上面 6 个操作合并为：

1. 一步操作之后，再将$A[2, \cdots, \text{len}A]$和$B[1, \cdots, \text{len}B]$变成相同字符串。

2. 一步操作之后，再将$A[1, \cdots, \text{len}A]$和$B[2, \cdots, \text{len}B]$变成相同字符串。

3. 一步操作之后，再将$A[2, \cdots, \text{len}A]$和$B[2, \cdots, \text{len}B]$变成相同字符串。

这样，很快就可以完成一个递归程序：

代码清单 3-6

```
Int CalculateStringDistance(string strA, int pABegin, int pAEnd, string strB,
  int pBBegin, int pBEnd)
{
    if(pABegin > pAEnd)
    {
        if(pBBegin > pBEnd)
            return 0;
        else
            return pBEnd – pBBegin + 1;
    }

    if(pBBegin > pBEnd)
    {
        if(pABegin > pAEnd)
            return 0;
        else
            return pAEnd – pABegin + 1;
    }

    if(strA[pABegin] == strB[pBBegin])
    {
        return CalculateStringDistance(strA, pABegin + 1, pAEnd, strB,
         pBBegin + 1, pBEnd);
    }
    else
    {
        int t1 = CalculateStringDistance(strA, pABegin + 1, pAEnd, strB,
         pBBegin + 2, pBEnd);
        int t2 = CalculateStringDistance(strA, pABegin + 2, pAEnd, strB,
         pBBegin + 1, pBEnd);
        int t3 = CalculateStringDistance(strA, pABegin + 2, pAEnd, strB,
         pBBegin + 2, pBEnd);
        return minValue(t1,t2,t3) + 1;
    }
}
```

上面的递归程序，有什么地方需要改进呢？在递归的过程中，有些数据被重复计算了。比如，如果开始我们调用 `CalculateStringDistance(strA, 1, 2, strB, 1, 2)`，图 3-4 是部分展开的递归调用：

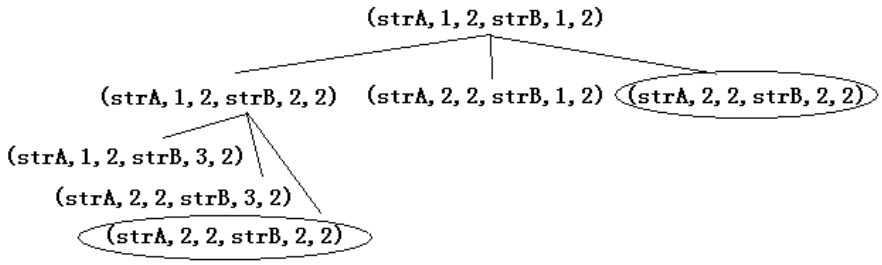


图 3-4

可以看到，圈中的两个子问题被重复计算了。为了避免这种不必要的重复计算，可以把子问题计算后的解存储起来。如何修改递归程序呢？这个问题就留给读者自己完成吧！

连连看游戏设计

连连看是一种很受大家欢迎的小游戏。微软亚洲研究院的实习生们就曾经开发过一个类似的游戏——Microsoft Link-up。

图 1-17　连连看游戏示意图

图 1-17 为 Microsoft Link-up 的一个截图。如果用户可以把两个同样的图用线（连线拐的弯不能多于两个）连到一起，那么这两个头像就会消掉，当所有的头像全部消掉的时候，游戏成功结束。游戏头像有珍稀动物、京剧脸谱等。Microsoft Link-up 还支持用户输入的图像库，微软的同事们曾经把新员工的漫画头像加到这个游戏中，让大家在游戏之余也互相熟悉起来。

假如让你来设计一个连连看游戏的算法，你会怎么做呢？要求说明：

1. 怎样用简单的计算机模型来描述这个问题？

2. 怎样判断两个图形能否相消？

3. 怎样求出相同图形之间的最短路径（转弯数最少，路径经过的格子数目最少）。

4. 怎样确定目前是处于死锁状态，如何设计算法来解除死锁？

分析与解法

连连看游戏的设计，最主要包含游戏局面的状态描述，以及游戏规则的描述。而游戏规则的描述就对应着状态的合法转移（在某一个状态，有哪些操作是满足规则的，经过这些满足规则的操作，会到达哪些状态）。所以，自动机模型适合用来描述游戏设计。

下面是一个参考的连连看游戏的伪代码：

代码清单 1-22

```
生成游戏初始局面
Grid preClick = NULL, curClick = NULL;
while(游戏没有结束)
{
   监听用户动作
   if(用户点击格子(x, y)，且格子(x, y)为非空格子)
   {
      preClick = curClick;
      curClick.Pos = (x, y);
   }
   if(preClick != NULL && curClick != NULL
   && preClick.Pic == curClick.Pic
   && FindPath(preClick, curClick) != NULL)
   {
      显示两个格子之间的消去路径
      消去格子preClick, curClick;
      preClick = curClick = NULL;
   }
}
```

从上面的整体框架可以看到，完成连连看游戏需要解决下面几个问题：

1. 生成游戏初始局面。

2. 每次用户选择两个图形，如果图形满足一定条件（两个图形一样，且这两个图形之间存在少于3个弯的路径），则两个图形都

能消掉。给定具有相同图形的任意两个格子，我们需要寻找这两个格子之间在转弯最少的情况下，经过格子数目最少的路径。如果这个最优路径的转弯数目少于3，则这两个格子可以消去。

3. 判断游戏是否结束。如果所有图形全部消去，游戏结束。

4. 判断死锁，当游戏玩家不可能再消去任意两个图像的时候，游戏进入“死锁”状态。如图1-18，该局面中已经不存在两个相同的图片相连的路径转弯数目小于3的情况。

在死锁的情况下，我们也可以暂时不终止游戏，而是随机打乱局面，打破“死锁”局面。

图 1-18　连连看死锁的情况

首先思考问题：怎样判断两个图形能否相消？在前面的分析中，我们已经知道，两个图形能够相消的充分必要条件是这两个图形相同，且它们之间存在转弯数目小于 3 的路径。因此，需要解决的主要问题是，怎样求出相同图形之间的最短路径。首先需要保证最短路径的转弯数目最少。在转弯数目最少的情况下，经过的格子数目也要尽可能地少。

在经典的最短路径问题中，需要求出经过格子数目最少的路径。而这里，为了保证转弯数目最少，需要把最短路径问题的目标函数修改为从一个点到另一个点的转弯次数。虽然目标函数修改了，但算法的框架仍然可以保持不变。广度优先搜索是解决经典最短路问题的一个思路。我们看看在新的目标函数（转弯数目最少）下，如何用广度优先搜索来解决图形 A（x_1，y_1）和图形 B（x_2，y_2）之间的最短路径问题。

首先把图形 A（x_1，y_1）压入队列。

然后扩展图形 A（x_1，y_1）可以直线到达的格子（即图形 A（x_1，y_1）

可以通过转弯数目为 0 的路径（直线）到达这些格子）。假设这些格子为集合 S_0，S_0 = Find（x_1, y_1）。如果图形 B（x_2, y_2）在集合 S_0 中，则结束搜索，图形 A 和 B 可以用直线连接。

否则，对于所有 S_0 集合中的空格子（没有图形），分别找到它们可以直线到达的格子。假设这个集合为 S_1。S_1 = {Find（p）| $p \in S_0$}。S_1 包含了 S_0，我们令 S_1’ = S_1-S_0，则 S_1’中的格子和图形 A（x_1, y_1）可以通过转弯数目为 1 的路径连起来。如果图形 B（x_2, y_2）在 S_1’中，则图形 A 和 B 可以用转弯数目为 1 的路径连接，结束搜索。

否则，我们继续对所有 S_1’集合中的空格子（没有图形），分别找出它们可以直线到达的格子，假设这个集合为 S_2，S_2 = Find{ Find（p）| $p \in S_1$’}。S_2 包含了 S_0 和 S_1，我们令 S_2’ = S_2 - S_0 - S_1 = S_2 - S_0 - S_1’。集合 S_2’是图形 A（x_1, y_1）可以通过转弯数目为 2 的路径到达的格子。如果图形 B（x_2, y_2）在集合 S_2’中，则图形 A 和 B 可以用转弯数目为 2 的路径连接，否则图形 A 和 B 不能通过转弯小于 3 的路径连接。

在扩展的过程中，只要记下每个格子是从哪个格子连过来的（也就是转弯的位置），最后图形 A 和 B 之间的路径就可以绘制出来。

在上面的广度优先搜索过程中，有两步操作：S_1’ = S_1 - S_0 和 S_2’ = S_2 - S_0 -S_1。它们可以通过记录从图形 A（x_1, y_1）到该格子（x, y）的转弯数目来实现。开始，将所有格子（x, y）和格子 A（x_1, y_1）之间路径的最少转弯数目 MinCrossing（x, y）初始化为无穷大。然后，令 MinCrossing（A）= MinCrossing（x_1, y_1）= 0，格子 A 到自身当然不需要任何转弯。第一步扩展之后，所有 S_0 集合中的格子的 MinCrossing 值为 0。在 S_0 集合继续扩展得到的 S_1 集合中，格子 X 和格子 A 之间至少有转弯为 1 的路径，如果格子 X 本身已经在 S_0 中，那么，MinCrossing（X）= 0。这时，我们保留转弯数目少的路径，也就是 MinCrossing（X）= MinValue（MinCrossing（X）, 1）= 0。这个过程，就实现了上面伪代码中的 S_1’ = S_1 - S_0。S_2’ = S_2 - S_0 - S_1 的扩展过程也类似。

经过上面的分析，我们知道，每一个格子 X（x, y），都有一个状态值 MinCrossing（X）。它记录下了该格子和起始格子 A 之间的最优路径的转弯数目。广度优先搜索，就是每次优先扩展状态值最少的格子。如果要保证在转弯数目最少的情况下，还要保持路径长度尽可能地短，则需要对每一个格子 X 保存两个状态值 MinCrossing（X）和 MinDistance（X）。从格子 X 扩展到格子 Y 的

过程，可以用下面的伪代码实现：

```
if((MinCrossing(X) + 1 < MinCrossing(Y)) ||
((MinCrossing(X) + 1 == MinCrossing(Y) && (MinDistance(X) +
   Dist(X,Y) < MinDistance(Y)))
{
   MinCrossing(Y) = MinCrossing(X) + 1;
   MinDistance(Y) = MinDistance(X) + Dist(X, Y);
}
```

也就是说，如果发现从格子 *X* 过来的路径改进了转弯数目或者路径的长度，则更新格子 *Y*。

“死锁”问题本质上还是判断两个格子是否可以消去的问题。最直接的方法就是，对于游戏中尚未消去的格子，都两两计算一下它们是否可以消去。此外，从上面的广度优先搜索可以看出，我们每次都是扩展出起始格子 $A(x_1, y_1)$ 能够到达的格子。也就是说，对于每一个格子，可以调用一次上面的扩展过程，得到所有可以到达的格子，如果这些格子中有任意一个格子的图形跟起始格子一致，则它们可以消去，目前游戏还不是“死锁”状态。

扩展问题：

1. 在连连看游戏设计中，是否可以通过维护任意两个格子之间的最短路径来实现快速搜索？在每一次消去两个格子之后，更新我们需要维护的数据（任意两个格子之间的最短路径）。这样的思路有哪些优缺点，如何实现呢？
2. 在围棋或象棋游戏中，经过若干步操作之后，可能出现一个曾经出现过的状态（例如，围棋中的打劫）。如何在围棋、象棋游戏设计中检测这个状态呢？

金刚坐飞机问题

国外有一个谚语：

问：体重 800 磅的大猩猩在什么地方坐？

答：它爱在哪儿坐就在哪儿坐。

这句谚语一般用来形容一些“强人”并不遵守大家公认的规则，所以要对其行为保持警惕。

现在有一班飞机将要起飞，乘客们正准备按机票号码（1, 2, 3, …N）依次排队登机。突然来了一只大猩猩（对，他叫金刚）。他也有飞机票，但是他插队第一个登上了飞机，然后随意地选了一个座位坐下了[1]。根据社会的和谐程度，其他的乘客有两种反应：

1. 乘客们都义愤填膺，“既然金刚同志不遵守规定，为什么我要遵守？”他们也随意地找位置坐下，并且坚决不让座给其他乘客。

2. 乘客们虽然感到愤怒，但还是以“和谐”为重，如果自己的位置没有被占领，就赶紧坐下，如果自己的位置已经被别人（或者金刚同志）占了，就随机地选择另一个位置坐下，并开始闭目养神，不再挪动位置。

那么，在这两种情况下，第 i 个乘客（除去金刚同志之外）坐到自己原机票位置的概率分别是多少？

[1] 金刚的口头禅是——我是金刚，我怕谁？大家在旅途中可能看见过类似的事儿。

分析与解法

这两个问题之间有一处小小的区别，这个区别是如何影响最后的概率的呢？

【问题 1 的解法】

我们可以用 $F(i)$ 来表示第 i 个乘客坐到自己原机票位置的概率。

第 i 个乘客坐到自己位置（概率为 $F(i)$），则前 $i-1$ 个乘客都不坐在第 i 个位置（设概率为 $P(i-1)$），并且在这种情况下第 i 个乘客随即选择位置的时候选择了自己的位置（设概率为 $G(i)$）。

而 $P(i-1)$ 可以分解为前 $i-2$ 个乘客都不坐在第 i 个位置的概率 $P(i-2)$，和在前 $i-2$ 个乘客都不坐在第 i 个位置的条件下第 $i-1$ 个乘客也不坐在第 i 个位置上的概率 $Q(i-1)$。

于是得到如下的公式（合并结果）：

$$F(i)=G(i)*P(i-1)=G(i)*Q(i-1)*P(i-2)=G(i)*Q(i-1)\cdots Q(2)*P(1)$$

容易知道 $Q(i)=(N-i)/(N-i+1)$，$P(1)=(N-1)/N$，$G(i)=1/(N-i+1)$

代入公式得到，$F(i)=1/N$。

【问题 2 的解法】

可以按照金刚坐的位置来分解问题，把原问题从“第 i 个乘客坐在自己位置上的概率是多少”变为“如果金刚坐在第 n 个位置上，那么第 i 个乘客坐在自己位置上的概率是多少”（设这个概率为 $f(n)$）。

现在金刚坐在了 n 号位置上。如果 $n=1$ 或 $n>i$，那么第 i 个乘客坐在自己位置上的概率是 1（因为大家会尽量坐到自己的位子上，2 号乘客将选择坐到 2 号位置上……）。如果 $n=i$，那么第 i 个乘客是没希望坐到自己的位置上了（他还不至于敢和金刚 PK）。如果 $1<n<i$，那么问题似乎并没有太直接的求解方式。我们来继续分解问题。当金刚坐在了第 n（$1<n<i$）个位置上的时候，第 2, 3, …, $n-1$ 号的乘客都可以坐到自己的座位上，于是我们可以按照第 n 个乘客坐的位置来继续分解这个问题。如果第 n 个乘客，选了金刚的座位，那么第 i 个乘客一定坐在自己的位置上；而如果第 n 个乘客坐在第 j（$n<j<=N$）个座位上，就相当于金刚坐了第 j 个座位。

把问题分解到这一步，应该可以进行问题解答的合并了。一般来讲，合并这个步骤，有可能很简单，也可能很复杂，这主要取决于问题分解的结果。在这道题目里，合并问题的主要工具是全概率公式，也即$P(M)=\sum_{i=1}^{M}P(i)*P(M|i)$，这里 i 表示各种不同的情况，$P(i)$ 表示这种情况发生的概率，$P(M|i)$ 表示在这种情况下事件 M 发生的概率。

首先求解 $f(n)$（$1<n<i$），由前面的分析可知：

$$f(n)=\sum_{j}\frac{1}{N-n+1}*f(j),(j=1,n+1,n+2,\cdots,N)$$

其中 j 表示第 n 个乘客坐的位置。

所以

$$f(n)=1/(N-n+1)*(1+f(n+1)+...+f(N))(1<n<i) \qquad (式1)$$

由此递推式，可得 $f(n)=f(n+1)(1<n<i-1)$

将 n=2 代入（式 1），再利用 $f(x)=1(x>i), f(x)=0(x=i)$，另 $1<n, n+1<i-1$ 可得：

$f(n)=\frac{N-i+1}{N-i+2}$，所以

$$f(n)=\begin{cases}1 & (n=1\text{ 或}n>i)\\ \frac{N-i+1}{N-i+2} & (1<n<i)\\ 0 & (n=i)\end{cases}$$

则$\sum_{n=1}^{N}\frac{1}{N}*f(n)=\frac{(N-i+1)}{(N-i+2)}$就是第 i 个乘客坐在自己位置上的概率。

回顾

有些问题看起来规模太大而无从下手。这时我们可以采用分而治之的方法，这个方法有两个核心步骤：

1. 分解问题，得到局部问题的答案。

2. 合并问题的解答。

扩展问题

在这个问题假设所有乘客是按照机票座位的次序（1，2，3，…）登机的，在现实生活中，乘客登机并没有一定的次序。如果在金刚抢先入座之后，所有乘客以随机次序登机，并且有原来题目所描述的两种行为，那第 i 个乘客坐到自己原机票位置的概率分别是多少？

1.6 饮料供货

★★★

在微软亚洲研究院上班，大家早上来的第一件事是干啥呢？查看邮件？No，是去水房拿饮料：酸奶，豆浆，绿茶、王老吉、咖啡、可口可乐……（当然，还是有很多同事把拿饮料当做第二件事）。

管理水房的阿姨们每天都会准备很多的饮料给大家，为了提高服务质量，她们会统计大家对每种饮料的满意度。一段时间后，阿姨们已经有了大批的数据。某天早上，当实习生小飞第一个冲进水房并一次拿了五瓶酸奶、四瓶王老吉、三瓶鲜橙多时，阿姨们逮住了他，要他帮忙。

从阿姨们统计的数据中，小飞可以知道大家对每一种饮料的满意度。阿姨们还告诉小飞，STC（Smart Tea Corp.）负责给研究院供应饮料，每天总量为 V。STC 很神奇，他们提供的每种饮料之单个容量都是 2 的方幂，比如王老吉，都是 2^3=8 升的，可乐都是 2^5=32 升的。当然 STC 的存货也是有限的，这会是每种饮料购买量的上限。统计数据中用饮料名字、容量、数量、满意度描述每一种饮料。

那么，小飞如何完成这个任务，求出保证最大满意度的购买量呢？

分析与解法

【解法一】

我们先把这个问题“数学化”一下吧。

假设 STC 共提供 n 种饮料，用（S_i、V_i、C_i、H_i、B_i）（对应的是饮料名字、容量、可能的最大数量、满意度、实际购买量）来表示第 i 种饮料（$i = 0, 1, \cdots, n-1$），其中可能的最大数量指如果仅买某种饮料的最大可能数量，比如对于第 i 中饮料 $C_i=V/V_i$。

基于如上公式：

饮料总容量为$\sum_{i=0}^{n-1}(V_i * B_i)$；

总满意度为$\sum_{i=0}^{n-1}(H_i * B_i)$；

那么题目的要求就是，在满足条件$\sum_{i=0}^{n-1}(V_i * B_i)=V$ 的基础上，求解 $\max\{\sum_{i=0}^{n-1}(H_i * B_i)\}$。

对于求最优化的问题，我们来看看动态规划能否解决。用 Opt（V', i）表示从第 i, $i+1$, $i+2$, …, $n-1$, n 种饮料中，算出总量为 V'的方案中满意度之和的最大值。

因此，Opt（V, n）就是我们要求的值。

那么，我们可以列出如下的推导公式：Opt（V', i）= max { $k* H_i$ + Opt（$V' - V_i * k$, $i-1$）}（$k = 0, 1, \cdots, C_i$, $i = 0, 1, \cdots, n-1$）。

即：最优化的结果 = 选择第 k 种饮料×满意度+减去第 k 种饮料×容量的最优化结果根据这样的推导公式，我们列出如下的初始边界条件：

Opt（0, n）= 0，即容量为 0 的情况下，最优化结果为 0。

Opt（x, n）= −INF（x != 0）（－INF 为负无穷大），即在容量不为 0 的情况下，把最优化结果设为负无穷大，并把它作为初值。

那么，根据以上的推导公式，就不难列出动态规划求解代码，如下所示：

代码清单 1-9

```
int Cal(int V, int type)
{
    opt[0][T] = 0;// 边界条件
    for(int i = 1; i <= V; i++)// 边界条件
    {
        opt[i][T] = -INF;
    }
    for(int j = T - 1; j >= 0; j--)
    {
        for(int i = 0; i <= V; i++)
        {
```

```
            opt[i][j] = -INF;
            for(int k = 0; k <= C[j]; k++)     // 遍历第j种饮料选取数量k
            {
                if(i <= k * V[j])
                {
                    break;
                }
                int x = opt[i - k * V[j]][j + 1];
                if(x != -INF)
                {
                    x += H[j] * k;
                    if(x > opt[i][j])
                    {
                        opt[i][j] = x;
                    }
                }
            }
        }
    }
    return opt[V][0];
}
```

在上面的算法中，空间复杂度为 $O(V*N)$，时间复杂度约为 $O(V*N*\text{Max}(C_i))$。

因为我们只需要得到最大的满意度，则计算 opt[i][j]的时候不需要 opt[i][j+2]，只需要 opt[i][j]和 opt[i][j+1]，所以空间复杂度可以降为 $O(v)$。

【解法二】

应用上面的动态规划法可以得到结果，那么是否有可能进一步地提高效率呢？我们知道动态规划算法的一个变形是备忘录法，备忘录法也是用一个表格来保存已解决的子问题的答案，并通过记忆化搜索来避免计算一些不可能到达的状态。具体的实现方法是为每个子问题建立一个记录项。初始化时，该纪录项存入一个特殊的值，表示该子问题尚未求解。在求解的过程中，对每个待求解的子问题，首先查看其相应的纪录项。若记录项中存储的是初始化时存入的特殊值，则表示该子问题是第一次遇到，此时计算出该子问题的解，并保存在其相应的记录项中。若记录项中存储的已不是初始化时存入的初始值，则表示该子问题已经被计算过，其相应的记录项中存储的是该子问题的解答。此时只需要从记录项中取出该子问题的解答即可。

因此，我们可以应用备忘录法来进一步提高算法的效率。

代码清单 1-10

```
int[V + 1][T + 1] opt;      // 子问题的记录项表，假设从 i 到 T 种饮料中，
                            // 找出容量总和为V’ 的一个方案，快乐指数最多能够达到
                            // opt（V'，i，T-1），存储于opt[V’][i]，
                            // 初始化时opt中存储值为-1，表示该子问题尚未求解。
int Cal(int V, int type)
{
    if(type == T)
    {
        if(V == 0)
            return 0;
        else
            return -INF;
    }
    if(V < 0)
        return -INF;
    else if(V == 0)
        return 0;
    else if(opt[V][type] != -1)
        return opt[V][type];    // 该子问题已求解，则直接返回子问题的解;
                                // 子问题尚未求解，则求解该子问题
    int ret = -INF;
    for(int i = 0; i <= C[type]; i++)
    {
        int temp = Cal(V – i * C[type], type + 1);
        if(temp != -INF)
        {
            temp += H[type] * i;
            if(temp > ret)
            ret = temp;
        }

    }
    return opt[V][type] = ret;
}
```

【解法三】

请注意这个题目的限制条件，看看它能否给我们一些特殊的提示。

我们把信息重新整理一下，按饮料的容量（单位为 L）排序：

Volume	TotalCount	Happiness
20L	TC_00	H_00
20L	TC_01	H_01
...	...	...
21L	TC_10	H_10
...	...	...
2000L	TC_M0	H_M0
...	...	...

假设最大容量为 2 000L。一开始，如果 V%（21）非零，那么，我们肯定需要购买 20L 容量的饮料，至少一瓶。在这里可以使用贪心规则，购买快乐指数最高的一瓶。除去这个，我们只要再购买总量（V−20）L 的饮料就可以了。这时，如果我们要购买 21L 容量的饮料怎么办呢？除了 21L 容量里面快乐指数最高的，我们还应该考虑，两个容量为 20L 的饮料组合的情况。其实我们可以把剩下的容量为 20L 的饮料之快乐指数从大到小排列，并用最大的两个快乐指数组合出一个新的“容量为 2L”[1]的饮料。不断地这样使用贪心原则，即得解。这是不是就简单了很多呢？

[1] 如果各种饮料数量都无限的话，这种方法是很简单。但是如果饮料有个数限制，复杂度可能达到指数级，您有更好的办法么？