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？