1.位与运算（&或者and）

and运算通常用于二进制的取[位操作](https://baike.baidu.com/item/%E4%BD%8D%E6%93%8D%E4%BD%9C" \t "_blank)，例如一个数 and 1的结果就是取[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6" \t "_blank)的最末位。这可以用来判断一个整数的奇偶，二进制的最末位为0表示该数为[偶数](https://baike.baidu.com/item/%E5%81%B6%E6%95%B0" \t "_blank)，最末位为1表示该数为奇数。

相同位的两个数字都为1，则为1；若有一个不为1，则为0。（不考虑进位）

00101

11100

----------------

00100

**2.异或运算（^）**

如果a、b两个值不相同，则异或结果为1。如果a、b两个值相同，异或结果为0。在二进制运算中结果如下（不考虑进位）

00101

11100

       ----------------

       11001

**3.shl运算 （<<）**

a shl b就表示把a转为二进制后左移b位（在后面添b个0）。例如100的二进制为1100100，而110010000转成十进制是400，那么100 shl 2 = 400。可以看出，a shl b的值实际上就是a乘以2的b次方，因为在二进制数后添一个0就相当于该数乘以2。

通常认为a shl 1比a \* 2更快，因为前者是更底层一些的操作。因此程序中乘以2的操作请尽量用左移一位来代替。

**运算思路**

一.将二进制运算分解成两部分 a+b=a^b+(a&b)<<1

1.异或运算 各个权值相等的位进行相加  此部分是没有进位的部分，进位的部分用0代替

例     0110   （十进制 6）

         0010   （十进制 2）

-----------------------

         0100    （十进制 4）

2 位与运算  通过位与运算得到两数相加中哪些权位进行了进位  在二进制运算中 向下一权位进1，即为原2进制的权位×2，

例     0110   （十进制 6）

         0010   （十进制 2）

**[cpp]** [view plain](https://blog.csdn.net/DlnuRicardo/article/details/79136819) [copy](https://blog.csdn.net/DlnuRicardo/article/details/79136819)

1. #include<iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a = 6, b = 2;
6. **int** c = a ^ b;
7. **int** d = a & b;
8. cout<<"初始c d 的值"<<endl;
9. cout<<"c="<<c<<endl;
10. cout<<"d="<<d<<endl;
11. **while**(d) //这里以进不进位作为循环的终结
12. {
13. **int** a = c;
14. **int** b = d << 1;
15. cout<<"循环中 a b 变化"<<endl;
16. cout<<"a="<<a<<endl;
17. cout<<"b="<<b<<endl;
18. c = a ^ b;
19. d = a & b;
20. cout<<"c="<<c<<endl;
21. cout<<"d="<<d<<endl;
22. }
23. printf("res = %d\n", c);
24. **return** 0;
25. ;
26. }

-----------------------

                   0010     （十进制2）

代表权值为2的位产生了进位，产生的进位的数值为0100 （用0010<<1得到）

将这两部分相加可得到第一步的结果

二.

将产生的两部分运算结果分别与输入的结果a，b进行替换，进行重复的运算操作，直到没有进位为止，得到最终结果

4.[计算n阶乘中尾部零的个数](https://blog.csdn.net/surp2011/article/details/51168272)

这个算法真的是感触很深，对平时很多习以为常的公式、道理有了非常直观的认识，因此对自己的冲击很大，也促进了思考的进步。

提交算法2的代码，发现前面的简单测试都能通过，但是数值5555550000000测试失败。特别是实现了时间复杂度O(logN)的算法3之后，才发现两者时间开销差别真的是很大。

#### 重新分析

1、2、3、4、5、6、7、8、9、10、11、...

* 1

1、分析上面的数列可知，每5个数中会出现一个可以产生结果中0的数字。把这些数字抽取出来是：

...、5、...、10、...、15、...、20、...、25、...

* 1

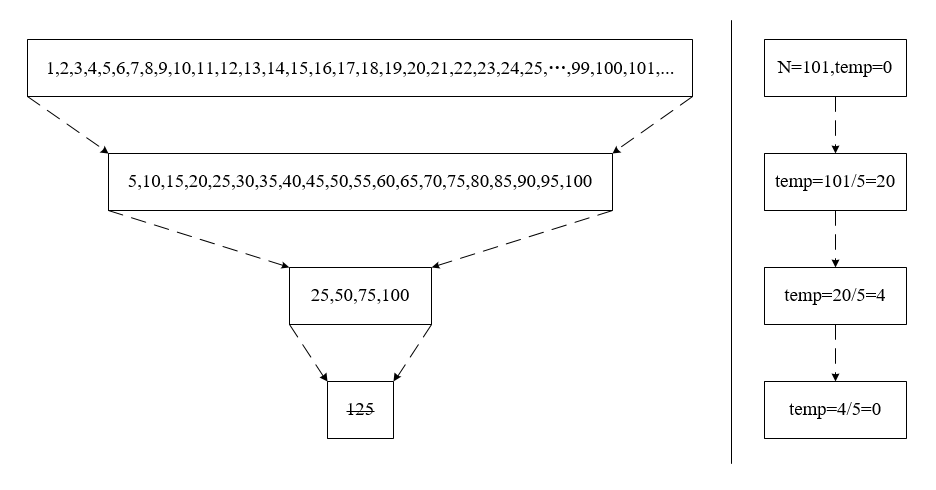
这些数字其实是都能满足5\*k的数字，是5的倍数。统计一下他们的数量：n1=N/5。比如如果是101，则101之前应该是5,10,15,20,...,95,100共101/5=20个数字满足要求。

整除操作满足上面的数量统计要求。

2、将1中的这些数字化成5\*(1、2、3、4、5、...)的形式，内部的1、2、3、4、5、...又满足上面的分析：每5个数字有一个是5的倍数。抽取为：

...、25、...、50、...、75、...、100、...、125、...

* 1

而这些数字都是25的倍数（5的2次幂的倍数），自然也都满足5\*k的要求。   
这些数字是25、50、75、100、125、...=5\*(5、10、15、20、25、...)=5\*5\*(1、2、3、4、5、...)，内部的1、2、3、4、5、...又满足上面的分析，因此后续的操作重复上述步骤即可。   
统计一下第二次中满足条件的数字数量：n2=N/5/5，101/25=(101/5)/5=4。   
因为25、50、75、100、125、...它们都满足相乘后产生至少两个0，在第一次5\*k分析中已经统计过一次。对于N=101，是20。因此此处的5\*5\*k只要统计一次4即可，不需要根据25是5的二次幂统计两次。   
后面的125,250,...等乘积为1000的可以为结果贡献3个0的数字，只要在5\*5\*k的基础上再统计一次n3=((N/5)/5)/5即可。   


3、第三次   
其实到这里已经不用再写，规律已经很清楚了。对于例子N=101，只要根据规律进行101/125=((101/5)/5)/5=4/5=0，退出统计。因此最终结果是20+4=24。计算结束。

#### 算法3代码

下面编写打码实现上面的思想。

public class Solution {

/\*

\* param n: As desciption return: An integer, denote the number of trailing

\* zeros in n!

\*/

public long trailingZeros(long n) {

// write your code here

long count = 0;

long temp=n/5;

while (temp!=0) {

count+=temp;

temp/=5;

}

return count;

}

}

算法中每次循环均有除以5的操作，也就是每次都会将所要处理的数据量缩小至上一次的1/5，容易推知时间复杂度为O(logN)。

至此，问题解决。