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)，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)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)，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

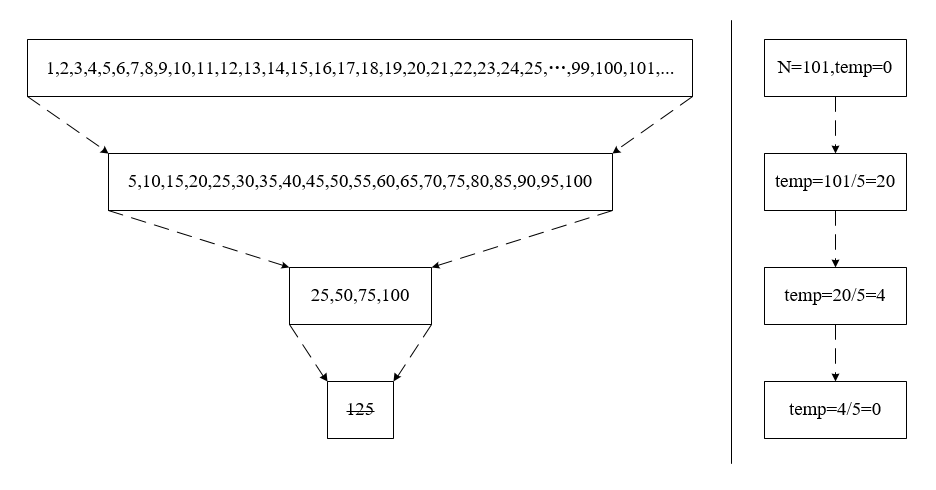
这些数字其实是都能满足5\*k的数字，是5的倍数。统计一下他们的数量：n1=N/5。比如如果是101，则101之前应该是5,10,15,20,...,95,100共101/5=20个数字满足要求。

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。

## 题目描述：

计算数字k在0到n中的出现的次数，k可能是0~9的一个值   
例如n=12，在 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]中，我们发现1出现了5次 (1, 10, 11, 12)   
**思路一：**我们可以用最简单的办法先尝试一下，遍历1到n中间的每个整数，对每个整数从低位到高位依次检查，如果有k出现则计数器自加。   
思路一最大的问题就是效率，当n非常大时，就需要很长的运行时间。想要提高效率，就要避开暴力法，从数字中找出规律。   
**思路二：**来自《编程之美》   
假设有一个5位数N=ABCDE，我们现在来考虑百位上出现2的次数，即：从0到ABCDE的数中，有多少个数的百位上是2。分析完它，就可以用同样的方法去计算个位，十位，千位，万位等各个位上出现2的次数。   
**第一种情况：当百位上的数C小于2时：**   
1）当百位c为0时，比如说12013，0到12013中哪些数的百位会出现2？我们从小的数起， 200~299, 1200~1299, 2200~2299, … , 11200~11299, 也就是固定低3位为200~299，然后高位依次从0到11，共12个。再往下12200~12299 已经大于12013，因此不再往下。所以，当百位为0时，百位出现2的次数只由更高位决定，等于更高位数字(12)x当前位数(100)=1200个。   
2）当百位C为1时，比如说12113。分析同上，并且和上面的情况一模一样。最大也只能到11200~11299，所以百位出现2的次数也是1200个。   
上面两步综合起来，可以得到以下结论：   
—>当某一位的数字小于2时，那么该位出现2的次数为：更高位数字x当前位数   
**第二种情况：当百位上的数C等于2时：**   
当百位C为2时，比如说12213。那么，我们还是有200~299, 1200~1299, 2200~2299, … , 11200~11299这1200个数，他们的百位为2。但同时，还有一部分12200~12213，共14个(低位数字+1)。所以，当百位数字为2时，百位出现2的次数既受高位影响也受低位影响，结论如下：   
—>当某一位的数字等于2时，那么该位出现2的次数为：更高位数字x当前位数+低位数字+1   
**第三种情况：当百位上的数C大于2时：**   
当百位C大于2时，比如说12313，那么固定低3位为200~299，高位依次可以从0到12，这一次就把12200~12299也包含了，同时也没低位什么事情。因此出现2的次数是： (更高位数字+1)x当前位数。结论如下：   
—>当某一位的数字大于2时，那么该位出现2的次数为：(更高位数字+1)x当前位数   
通过上述分析，我们可以得到以下规律：   
λ 当某一位的数字小于i时，那么该位出现i的次数为：更高位数字x当前位数   
λ 当某一位的数字等于i时，那么该位出现i的次数为：更高位数字x当前位数+低位数字+1   
λ 当某一位的数字大于i时，那么该位出现i的次数为：(更高位数字+1)x当前位数

class Solution {

public:

/\*

\* @param : An integer

\* @param : An integer

\* @return: An integer denote the count of digit k in 1..n

\*/

int digitCounts(int k, int n) {

// write your code here

if(n == 0 && k == 0)

return 1; // 特殊情况

int temp = n, cnt = 0, pow = 1;

while(temp != 0) {

int digit = temp % 10; // 根据当前位置数和k的大小关系，可以算出当前位置出现过k的次数

if(digit < k)

cnt += (temp / 10) \* pow;

else if(digit == k)

cnt += (temp / 10) \* pow + (n - temp \* pow + 1);

else {

if(!(k == 0 && temp / 10 == 0)) // 排除没有更高位时，寻找的数为0的情况

cnt += (temp / 10 + 1) \* pow;

}

temp /= 10;

pow \*= 10;

}

return cnt;

}

};

## 题目描述：

设计一个算法，找出只含素因子2，3，5 的第 n 大的数。   
符合条件的数如：1, 2, 3, 4, 5, 6, 8, 9, 10, 12…   
**注意：1也是丑数**   
例如：输入9，则输出10

## 思路一：暴力解法

从1开始依次向后判断各自然数是否为丑数，一直判断到第n个丑数，返回第n个丑数的值。   
第一步：先判断每一个数m是否为丑数：可以试着用2、3、5不断整除m，当m不能再被2、3、5整除时，判断m是否等于1，等于1则指定的数字是丑数（返回真），否则不是（返回假）。   
**判断是否为丑数的方法：**对一个数重复除以2，3，5直到这个数不存在2，3，5这三个因子为止，若此时得到的结果不为1，则说明该数不是丑数。

bool isUgly(int num)

{

if(num< 1) return false;

if(num == 1) return true;

vector<int> num\_vect;

num\_vect.push\_back(2);

num\_vect.push\_back(3);

num\_vect.push\_back(5);

for(size\_t i=0; i!=num\_vect.size();++i)

{

while(num % num\_vect[i] == 0)

{

num = num/num\_vect[i];

}

}

if(num == 1)

returntrue;

else

returnfalse;

}

class Solution

{

public:

int nthUglyNumber(int n)

{

if(n<= 1)

return 1;

int count = 0;

for(int i = 1; ; i++)

{

if(isUgly(i))

{

count++;

if(count == n)

return i;

}

}

}

};

# [利用快排思想找出数组中第k大的元素](https://blog.csdn.net/bladeandmaster88/article/details/52438577)

1. //用快排的思想：例如找49个元素里面第24大的元素，那么按如下步骤：
2. //1.进行一次快排（将大的元素放在前半段，小的元素放在后半段）, 假设得到的中轴为p
3. //2.判断 k -1==p - low，如果成立，直接输出a[p]，（因为前半段有k - 1个大于a[p]的元素，故a[p]为第K大的元素）
4. //3.如果 k -1 < p - low， 则第k大的元素在前半段，此时更新high = p - 1，继续进行步骤1
5. //4.如果 k -1 > p - low， 则第k大的元素在后半段，此时更新low = p + 1, 且 k = k - (p - low + 1)，继续步骤1.
6. //由于常规快排要得到整体有序的数组，而此方法每次可以去掉“一半”的元素，故实际的复杂度不是o(nlgn), 而是o(n)。
7. **class** Finder
8. {
9. **public**:
10. **int** partition(vector<**int**>&a, **int** low, **int** high)//找枢纽
11. {
12. **int** first = low;
13. **int** last = high;
14. **int** key = a[first];//用字表的第一个记录作为枢轴
16. **while** (first < last)
17. {
18. **while** (a[last] >= key && first < last)
19. --last;
20. swap(a[first], a[last]);
22. **while** (a[first] <= key && first < last)
23. ++first;
24. swap(a[first], a[last]);
25. }
26. **return** first;//返回一个枢纽
27. }
29. **int** findKth(vector<**int**>& a, **int** low, **int** high, **int** k)
30. {
31. **int** p = partition(a, low, high);
32. **if** (k == p - low + 1)
33. **return** a[p];
35. **else** **if** (k - 1 < p - low)//则第k大的元素在前半段
36. **return** findKth(a, low, p - 1, k);
38. **else** //则第k大的元素在后半段
39. **return** findKth(a, p + 1, high, k - p + low - 1);
40. }
42. **int** findKth(vector<**int**> a, **int** n, **int** K)
43. {
44. **return** findKth(a, 0, n - 1, K);
45. }
47. };

**题目：**

合并两个排序的整数数组A和B变成一个新的数组。**我采用的比较常规的思路，用的就是归并排序的方法。   
首先开辟一个新的空间，其大小是两个数组的总和，来存放合并排序之后的数组。  
然后就是归并的常规思路了。**

class Solution {

public:

/\*\*

\* @param A: sorted integer array A

\* @param B: sorted integer array B

\* @return: A new sorted integer array

\*/

vector<int> mergeSortedArray(vector<int> &A, vector<int> &B) {

// write your code here

vector<int> result(A.size()+B.size(), 0);

int i = 0, j = 0;

int k = 0;

while (i < A.size() && j < B.size())

{

if (A[i] > B[j])

{

result[k++] = B[j++];

} else

{

result[k++] = A[i++];

}

}

while (i < A.size())

{

result[k++] = A[i++];

}

while (j < B.size())

{

result[k++] = B[j++];

}

return result;

}

};

设计一个算法，并编写代码来序列化和反序列化二叉树。将树写入一个文件被称为“序列化”，读取文件后重建同样的二叉树被称为“反序列化”。

如何反序列化或序列化二叉树是没有限制的，你只需要确保可以将二叉树序列化为一个字符串，并且可以将字符串反序列化为原来的树结构。

给出一个测试数据样例， 二叉树{3,9,20,#,#,15,7}，表示如下的树结构：

3

/  \

9   20

    /    \

  15    7

数据是按照BFS遍历得到的，BFS遍历也叫广度优先搜索，以上面的为例子，就是一层一层地遍历，第一层，遍历得到3，第二层遍历得到9和10，第三层遍历得到#，#，15,7。

下面是实现的代码：

1. /\*\*
2. \* Definition of TreeNode:
3. \* public class TreeNode {
4. \*     public int val;//节点的值
5. \*     public TreeNode left, right;//左右节点
6. \*     public TreeNode(int val) {
7. \*         this.val = val;//初始化节点
8. \*         this.left = this.right = null;//将左右节点置为0
9. \*     }
10. \* }
11. \*/
12. **class** Solution {
13. /\*\*
14. \* This method will be invoked first, you should design your own algorithm
15. \* to serialize a binary tree which denote by a root node to a string which
16. \* can be easily deserialized by your own "deserialize" method later.
17. \*/
18. **public** **static** Queue<TreeNode> queue = **new** LinkedList<>();//用于保存将树结构序列化成字符串时使用来保存
19. **public** **static** Queue<TreeNode> dataqueue = **new** LinkedList<>();//用于将字符串结构反序列化成树时使用来保存节点
20. **public** String string = "";
22. **public** String serialize(TreeNode root) {
23. // write your code here
24. **if** (root == **null**) {//如果根节点为空，整棵树为空，返回“#”
25. **return** "#";
26. } **else** {
27. queue.add(root);、、根节点不为空，加入队列
28. **while** (!queue.isEmpty()) {//直到队列为空，才跳出循环
29. TreeNode treeNode = queue.poll();//每次取出队列的第一的节点
30. **if** (treeNode == **null**)//如果为空，那么直接输出“#”，不再把该点的左右子节点加入队列
31. string = string + "#" + ",";
32. **else** {//该节点不为空，则把该点的值加上在字符串后面
33. string = string + treeNode.val + ",";
34. queue.add(treeNode.left);//该节点不为空，那么它的左右子节点都要加进去，可能为空，可能不为空，加进去不判断
35. queue.add(treeNode.right);//从队列中取出来才判断是不是为空，为空则没有下一层了
36. }
37. }
38. }
39. string = string.substring(0, string.length() - 2);//将后面的“，”去掉
40. **return** string;
41. }
43. /\*\*
44. \* This method will be invoked second, the argument data is what exactly you
45. \* serialized at method "serialize", that means the data is not given by
46. \* system, it's given by your own serialize method. So the format of data is
47. \* designed by yourself, and deserialize it here as you serialize it in
48. \* "serialize" method.
49. \*/
50. **public** TreeNode deserialize(String data) {
51. // write your code here
52. String[] strs = data.split(",");//根据“，”来分隔成为字符串数组
53. **if** (strs.length == 0||data.equals("#")) {//判断字符串数组是等于#或者为空
54. TreeNode rooTreeNode = **null**;
55. **return** rooTreeNode;//直接返回空根节点
56. } **else** {//数组中至少有一个元素
57. TreeNode rootNode = **new** TreeNode(Integer.parseInt(strs[0]));//拿到数组的第一个元素
58. dataqueue.add(rootNode);//把根节点加进去队列中（思路是把已经创建好的节点加进去队列中）
59. **int** i = 1;下次取出第二个元素
60. **while** (!dataqueue.isEmpty() && i < strs.length) {//直到队列为空，或者数组中元素被取完了
61. TreeNode treeNode = dataqueue.poll();//每次弹出队列的第一个节点，分别构造左右节点
62. **if** (strs[i].equals("#")) {//如果等于“#”,那么取出来的这个节点的左节点为空
63. treeNode.left = **null**;
64. i++;
65. } **else** {//如果不是空节点，那么我们要把这个节点加入队列中，这意味着这个节点可能还有子节点
66. treeNode.left = **new** TreeNode(Integer.parseInt(strs[i]));
67. dataqueue.add(treeNode.left);
68. i++;
69. }//下面接着判断右节点，注意，有可能数组被取光了，也就是没有右节点了
70. **if** (i < strs.length&&strs[i].equals("#") ) {
71. treeNode.right = **null**;
72. i++;
73. } **else** **if**(i < strs.length&& !strs[i].equals("#")){
74. treeNode.right = **new** TreeNode(Integer.parseInt(strs[i]));
75. dataqueue.add(treeNode.right);
76. i++;
77. }
79. }
80. **return** rootNode;
81. }
83. }
84. }//在上面，最主要的是使用了队列的数据结构，然后把第一个节点加进去（不为空），再每次从队列中取出节点，直到队列中的节点被取完，当然，在循环中，如果符合条件（节点不为空，
85. //就会往队列中加入节点），这样就实现了，同一层的都是挨着加入队列的，出来的时候，也是紧挨着的，而不是不断递归到最底层，优先忽略了同一层的其他节点。