# 背景

不安全网页的黑名单包含100亿个黑名单网页，每个网页的URL最多占用64字节。现在想要实现一种网页过滤系统，可以根据网页的URL判断该网页是否在黑名单上，请设计该系统，要求该系统允许有万分之一以下的判断错误率，并且使用的额外空间不要超过30G。



对于涉及网页黑名单系统，垃圾邮件过滤系统，爬虫的网址判断重复系统，同时可以容忍一定程度的失误率，对空间要求较严格的情况，可以使用布隆过滤器。

# 位运算符

位运算符：&、|、^、~ 、<<、>>、>>>

# 布隆过滤器

## 概述

布隆过滤器可以精确地代表一个集合（不是准确代表集合，精确程度由用户的具体设计决定，做到100%的正确是不可能的），可精确（不是准确）判断某一元素是否在此集合中。

布隆过滤器的优势在于，利用很少的空间可以做到精确率较高。

## 原理







注：如果a对应的bitarray中所有位置是否为1，如果有1个不为1则一定不在集合里，如果都为1，则a在该集合中，但是可能误判。

## 参数

布隆过滤器的bitarray大小如何确定？

说明：大小为m，样本数量为n，失误率为p。

n=100亿，p=0.01%：

单个样本大小不影响布隆过滤大小，只影响了哈希函数的实现细节：

求得m=19.19n，向上取整为m=20n。

2000亿bit，约为25G。

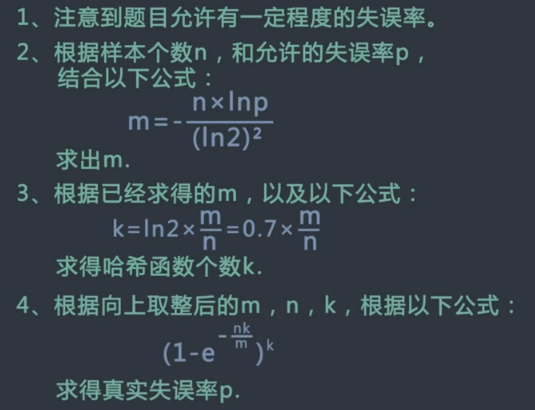
求得：k=14

综上，m=20n，k=14

求得真实失误率p=0.006%，占用大小为25G。

总结：

1. 注意到题目允许一定程度的失误率；
2. 根据样本个数n，和允许的失误率p，结合以下公式求出m：
3. 根据已经求得的m，结合以下公式求得哈希函数个数k：
4. 根据向上取整后的m，n，k，结合以下公式求得真实失误率p：



# 应用

位运算的应用主要包含：

1. 出现次数类型：只出现一次Leetcode136、Leetcode137、Leetcode260、多数元素Leetcode169、出现次数超过一半剑指offer39、缺失数字Leetcode268  
    2、二进制数计算类型：颠倒二进制位Leetcode190、位1的个数Leetcode191、2的幂Leetcode231、4的幂Leetcode342、数字范围按位与Leetcode201、比特位计数Leetcode338、两整数之和Leetcode371、整数替换Leetcode397、数字补码Leetcode476、不用加号的加法

## 只出现一次的数字

题目：给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

输入: [2,2,1]

输出: 1

示例 2:

输入: [4,1,2,1,2]

输出: 4

**分析：**

**方法一：暴力求解**

//哈希暴力求解

class Solution {

public:

int singleNumber(vector<int>& nums) {

map<int,int> m;

for(int i=0; i<nums.size();i++){

if(m.find(nums[i]) != m.end()){

m.erase(m.find(nums[i]));

}else{

m[nums[i]] = 1;

}

}

return m.begin()->first;

}

};

**方法二：异或法**

两个相等的数字经过异或后，还原为0，所以最后留下的就是只有一个的数字了

**代码：**

//异或

class Solution {

public:

int singleNumber(vector<int>& nums) {

int v=0;

for(int i=0; i<nums.size(); i++){

v ^= nums[i];

}

return v;

}

};

## 多数元素

**题目：**给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: [3,2,3]

输出: 3

示例 2:

输入: [2,2,1,1,1,2,2]

输出: 2

**分析：**

**方法一：排序+中位数**

可以将数组先排序，然后出现次数超过一半的必定在数组中间位置。

**代码：**

方法一：

class Solution {

public:

int majorityElement(vector<int>& nums) {

std::sort(nums.begin(),nums.end());

return nums.at(nums.size()/2);

}

};

## 消失数字

**题目：**数组nums包含从0到n的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在O(n)时间内完成吗？

注意：本题相对书上原题稍作改动

示例 1：

输入：[3,0,1]

输出：2

示例 2：

输入：[9,6,4,2,3,5,7,0,1]

输出：8

**分析：**

**方法一：排序+比较**

先对原来数组排序，然后对应的下标与数组中的nums[i]比较，如果不相等，则返回这个值，否则返回下一个值。

**代码：**

方法一：

class Solution {

public:

int missingNumber(vector<int>& nums) {

std::sort(nums.begin(),nums.end());

int tmp = 0;

for(int i=0;i<nums.size();i++)

{

tmp = i;

if(i != nums.at(i))

return i;

}

return tmp+1; //或者return nums.size();

}

};

方法二：位运算

思路：

利用异或的特性：res = res ^ x ^ x。对同一个值异或两次，那么结果等于它本身，所以我们对res从0-nums.length进行异或，同时对nums数组中的值进行异或，出现重复的会消失，所以最后res的值是只出现一次的数字，也就是nums数组中缺失的那个数字。

代码：

class Solution {

public int missingNumber(int[] nums) {

int res = 0;

for (int i = 0; i < nums.length; ++i) {

res ^= i;

res ^= nums[i];

}

res ^= nums.length;

return res;

}

}

## 数组中出现次数超一半的数字

**题目：**

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: [1, 2, 3, 2, 2, 2, 5, 4, 2]

输出: 2

限制：

1 <= 数组长度 <= 50000

**分析：**

1. 排序

排序后中间的元素一定是出现超过一半的数字。

1. 位运算

给出输入[1,2,3,3,3]，将这几个数字写成二进制数(整型32位，这里只写出后面4位)[0001, 0010, 0011, 0011, 0011]。在这4位二进制数字中，各自的最高位分别为[0，0，0，0]，则最高位出现最多的数字是二进制0。同理，高二位出现最多的二进制数字为0，高三位出现最多的二进制数字为1，最低位出现最多的二进制数字为1。则4位出现最多的二进制数字拼起来为0011，值为3，即结果。

注：位运算的优点在于占用内存少且运算速度快！

1. 哈希表

使用unordered\_map数据结构，扫一遍给定数组，存储每个数字出现的次数。

再扫一遍哈希表，找出出现次数超过一半的数字

**代码：**

class Solution {

public:

int majorityElement(vector<int>& nums) {

int ans = 0;

int n = nums.size();

//位运算

for(int i=0; i < 32; i++){

int cnt = 0;

for(int n: nums){

if(n & 1 << i) cnt ++;

}

if(cnt > n / 2) ans ^= 1 << i;

}

return ans;

}

};

class Solution {

public:

int majorityElement(vector<int>& nums) {

//方法1：排序后中间的元素一定是出现超过一半的数字

sort(nums.begin(),nums.end());

return nums[nums.size()/2];

//方法2：哈希表

unordered\_map<int,int>mp;

for(auto it : nums){

mp[it]++;

if(mp[it]>nums.size()/2) return it;

}

return 0;

//方法3：超过一半的数字比其他所有数字的总和次数多

int n=1;

int result=nums[0];

for(int i=1;i<nums.size();i++){

if(n==0){

result=nums[i];

n=1;

}

else if(result==nums[i])n++;

else n--;

}

return result;

}

};

## 查找奇数次数字

**题目：**给定一个整型数组arr，其中只有一个数出现了奇数次，其他的数都出现了偶数次，请打印这个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。

**分析：**

N与0异或结果为n，n与n异或结果为0

E0=0,arr=[C,B,D,A,A,B,C]

异或运算满足交换律和结合律

按照原始arr数出现的顺序异或结果，与该数组异或的结果相同：[A,A,B,B,C,C,D]

e0与上述数组异或的结果为e0=D

**拓展：**给定一个整型数组arr，其中有两个数出现了奇数次，其他的数都出现了偶数次，请打印这两个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。



注：对于数组中存在奇偶个数或者奇偶相关的查找排序操作，可以采用位运算解决。

## 二进制中1的个数

**题目：**请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

示例 1：

输入：00000000000000000000000000001011

输出：3

解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

示例 2：

输入：00000000000000000000000010000000

输出：1

解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。

示例 3：

输入：11111111111111111111111111111101

输出：31

解释：输入的二进制串 11111111111111111111111111111101 中，共有 31 位为 '1'。

**分析：**

方法一：按位与

根据与运算定义，设二进制数字n，则有：

若n & 1 =0，则n二进制最右一位为0；

若n & 1 =1，则n二进制最右一位为1。

根据以上特点，考虑以下循环判断：

判断n最右一位是否为1，根据结果计数。

将n右移一位（本题要求把数字n看作无符号数，因此使用无符号右移操作）

注：凡是计算1的个数的都是与操作，凡是计算0的个数的都是或操作。

**代码：**

class Solution {

public:

int hammingWeight(uint32\_t n) {

int num = 0;

for(int i=0;i<32;i++){

if(n & (1<<i)) //而不是i<<1

num++;

}

return num;

}

};

## 汉明距离

题目：两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。

注意：

0 ≤ x, y < 231.

示例:

输入: x = 1, y = 4

输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

上面的箭头指出了对应二进制位不同的位置。

**分析：**

1、两个数字异或之后可以得到不同二进制位的数字（异或求不同）

2、计算该数字中1的个数，即是汉明距离

计算1的个数时，有几种方法:

1、不断和左移的1进行与，判断该位是否为1；

2、n&(n-1)就是把n最右边的bit 1位去掉，看能去掉几次，就有几个1位。

**代码：**

class Solution {

public:

int hammingDistance(int x, int y) {

return countBitNumber(x^y);

}

int countBitNumber(int n)

{

int count = 0;

while(n)

{

n = (n & n-1);

++count;

}

return count;

}

};

另外一种写法：

/\*

1.先经过异或操作求得两个数不同的数

2.再判断x^y结果中含有1的个数

\*/

class Solution {

public:

int hammingDistance(int x, int y) {

int tmp = x ^ y;

int num = 0;

for(int i=0;i<32;i++)

{

if(tmp & (1<<i))

num++;

}

return num;

}

};

## 2的幂

**题目：**给定一个整数，编写一个函数来判断它是否是2的幂次方。

示例1:

输入: 1

输出: true

解释: 20 = 1

示例2:

输入: 16

输出: true

解释: 24 = 16

示例3:

输入: 218

输出: false

**分析：**

该问题将通过位运算在O(logN)的时间复杂度解决，通过使用如下的按位技巧：

如何获取二进制中最右边的1：x & (-x)。

如何将二进制中最右边的1设置为0：x & (x - 1)。

以下的两种解决方案背后思想都是一样的：2的幂在二进制中是有一个1后跟一些 0：

1 = (00000001) 2

2 = (00000010) 2

4 = (00000100) 2

8 = (00001000) 2

不是2的幂的二进制中有一个以上的1。

3 = (00000011) 2

5 = (00000101) 2

6 = (00000110) 2

7 = (00000111) 2

除了0，我们应该单独处理。

**代码：**

**方法一：**逐次求余

class Solution {

public:

bool isPowerOfTwo(int n) {

if(0 == n)

return false;

while(0==n%2)

n = n/2;

if(1==n)

return true;

return false;

}

};

**方法二：位运算**

## 找不同

题目：给定两个字符串 s 和 t，它们只包含小写字母。

字符串 t 由字符串 s 随机重排，然后在随机位置添加一个字母。

请找出在 t 中被添加的字母。

示例:

输入：

s = "abcd"

t = "abcde"

输出：

e

解释：

'e' 是那个被添加的字母。

**分析：**

**代码：**

class Solution {

public:

char findTheDifference(string s, string t)

{

sort(s.begin(), s.end());

sort(t.begin(), t.end());

int i, len = s.size();

for(i = 0; i < len; i++){

if(s[i] != t[i])

return t[i];

}

return t[i]; //不同的为t的最后一个字符

}

};

## 求较大值

**题目：**给定两个32位整数a和b，返回a和b中较大的，但是不能用任何比较判断。

**分析：**

**方法一：得到a-b的符号，根据该符号决定返回a或b**

public static int flip(int n){

return n ^ 1;

}

public static int sign(int n){

return flip((n>>31)&1);

}

public static int getMax(int a,int b){

int c = a - b;

int scA = sign(c);

int scB = flip(scA);

return a\*scA + b\*scB;

}

注：方法一可能存在问题，当a-b溢出时，会发生错误。

**方法二：**

public static int getMax2(int a,int b){

int a = a - b;

int as = sign(a); //a的符号,as==1表示a为非负,as==0表示a为负

int bs = sign(b); //b的符号,bs==1表示b为非负,bs==0表示b为负

int cs = sign(c); //a-b的符号

int difab = as ^ bs; //表示a和b是否符号相同,相同为0,不相同为1

int sameab = flip(difab); //表示aheb是否符号相同,相同为1,不同为0

int returnA = difab\*as + sameab\*cs;

int returnB = flip(returnA);

return a\*returnA + b\*returnB;

}

## 交换整数值

**题目：**如何不用任何额外变量交换两个整数的值？

分析：

对于给定整数a和b，如果采用额外申请内存的方式：

int c = a;

a = b;

b = c;

假设a=a0，b=b0：

a = a ^ b；🡪a = a0 ^ b0，b = b0

b = a ^ b；🡪a = a0 ^ b0，b = a0 ^ b0 ^ b0 = a0

a = a ^ b；🡪a = a0 ^ b0 ^ a0 = b0，b = a0

代码：

class Solution {

public:

vector<int> swapNumbers(vector<int>& numbers) {

numbers[0] = numbers[0] ^ numbers[1];

numbers[1] = numbers[0] ^ numbers[1];

numbers[0] = numbers[0] ^ numbers[1];

return numbers;

}

};

## 加密解密

异或运算可完成简单的加密与解密过程：

明文text，用户给定的密码pw，假设密文为cipher：

cipher = text ^ pw

text = cipher ^ pw = (text ^ pw) ^ pw

= text ^ (pw ^ pw) = text

如果text长度大于pw，循环使用pw与text进行按位异或。