**Subarray Sum Closest**

**二分搜索**

**模板一 – lower bound**

定义 lower bound 为在给定升序数组中大于等于目标值的最小索引

这里的有序数组是nums

public static int lowerBound(int[] nums, int target) {

if (nums == null || nums.length == 0) return -1;

int lb = -1, ub = nums.length;

while (lb + 1 < ub) {

int mid = lb + (ub - lb) / 2;

if (nums[mid] < target) {

lb = mid;

} else {

ub = mid;

}

}

return lb + 1;

}

**模板二 - 最优解**

我们还可以利用二分搜索求最优解（最大值/最小值），通常这种题中只是隐含了『有序数组』，需要我们自己构造。

这里构造的有序数组是 int i = 0; i < 100;

有  *N* 条绳子，它们的长度分别为 *L*​*i*​​. 如果从它们中切割出 *K* 条长度相同的绳子的话，这  *K* 条绳子每条最长能有多长？答案保留到小数点后两位。

public static double solve(double[] nums, int K) {

double lb = 0.00, ub = 10e5 + 0.01;

// while (lb + 0.001 < ub) {

for (int i = 0; i < 100; i++) {

double mid = lb + (ub - lb) / 2;

if (C(nums, mid, K)) {

lb = mid;

} else {

ub = mid;

}

}

return lb;

}

public static boolean C(double[] nums, double seg, int k) {

int count = 0;

for (double num : nums) { //每一个num代表1整条绳子

count += Math.floor(num / seg);

}

//count为所有绳子切每段seg 的整数和

return count >= k; //判断是否大于k

}

}

方法C 只做一件事，给定数组nums, 判断是否能切割出K 条长度均为seg 的绳子。while 循环中使用lb + 0.001 < ub, 不能使用0.01, 因为计算mid 时有均值的计算，对于double 型数值否则会有较大误差。

**样例**

将binary search 问题转化成：寻找第一个或者最后一个，该target元素出现的位置的问题，,

def binary\_search(self, array, target):

if not array:

return -1

start, end = 0, len(array) - 1

while start + 1 < end:

mid = (start + end) / 2

if array[mid] == target:

start = mid

elif array[mid] < target:

start = mid

else:

end = mid

if array[start] == target:

return start

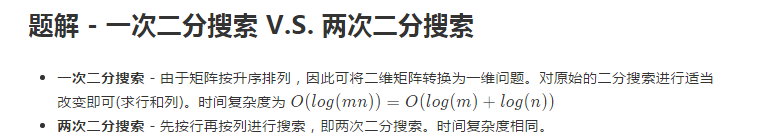
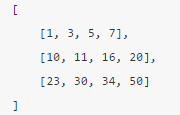
if array[end] == target:

return end

return -1

**Search a 2D Matrix**

顺序数组:



def search\_matrix(self, matrix, target):

# Find the first position of target

if not matrix or not matrix[0]:

return False

m, n = len(matrix), len(matrix[0])

st, ed = 0, m \* n - 1

while st + 1 < ed:

mid = (st + ed) / 2

if matrix[mid / n][mid % n] == target:

return True

elif matrix[mid / n][mid % n] < target:

st = mid

else:

ed = mid

return matrix[st / n][st % n] == target or \

matrix[ed / n][ed % n] == target

[Search in Rotated Sorted Array 在旋转有序数组中搜索](http://www.cnblogs.com/grandyang/p/4325648.html)

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

这道题让在旋转数组中搜索一个给定值，若存在返回坐标，若不存在返回-1。我们还是考虑二分搜索法，但是这道题的难点在于我们不知道原数组在哪旋转了，我们还是用题目中给的例子来分析，对于数组[0 1 2 4 5 6 7] 共有下列七种旋转方法：

0　　1　　2　　 **4　　5　　6　　7**

7　　0　　1　　 **2　　4　　5　　6**

6　　7　　0　　 **1　　2　　4　　5**

5　　6　　7　　 **0　　1　　2　　4**

**4　　5　　6　　7**　　0　　1　　2

**2　　4　　5　　6**　　7　　0　　1

**1　　2　　4　　5**　　6　　7　　0

二分搜索法的关键在于获得了中间数后，判断下面要搜索左半段还是右半段，我们观察上面红色的数字都是升序的，由此我们可以观察出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了，代码如下：

[复制代码](javascript:void(0);)

class Solution {

public:

int search(int A[], int n, int target) {

if (n == 0) return -1;

int left = 0, right = n - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (A[mid] == target) return mid;

else if (A[mid] < A[right]) {

if (A[mid] < target && A[right] >= target)

left = mid + 1;

else right = mid - 1;

} else {

if (A[left] <= target && A[mid] > target) right = mid - 1;

else left = mid + 1;

}

}

return -1;

}

};

实现sqrt也可也用二分搜索。

lb, ub = 1, x

while lb + 1 < ub:

mid = (lb + ub) / 2

if mid\*\*2 == x:

return mid

elif mid\*\*2 < x:

lb = mid

else:

ub = mid

return lb

**最大公约数**

常用的方法为辗转相除法，也称为欧几里得算法。

a 6 b 4

gcd(4, 2) = gcd(2, 0)

**Java**

public static long gcd(long a, long b) {

return (b == 0) ? a : gcd(b, a % b);

}

更优的洗牌算法**Knuth-Durstenfeld Shuffle**

**每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字**。

该算法类似于插入排序

从数组的最后一个数（下标为i）开始，进行随机取余（除数为i+1,确保下标不过界）

将得到的下标对应的元素和最后一个数交换

将最后一个数不在视为数组中的元素，继续循环

直到只剩下第一个元素为止

def shuffle(lis):

for i in range(len(lis) - 1, 0, -1):

p = random.randrange(0, i + 1)

lis[i], lis[p] = lis[p], lis[i]

return lis

r = shuffle([1, 2, 2, 3, 3, 4, 5, 10])

print(r)

判断两个字符串是否互为变位词

Given s = "abcd", t = "dcab", return true.  
Given s = "ab", t = "ab", return true.  
Given s = "ab", t = "ac", return false.

**题解1 - hashmap 统计字频**

def anagram(self, s, t):

return collections.Counter(s) == collections.Counter(t)

另一直接的解法是对字符串先排序，若排序后的字符串内容相同，则其互为变位词。

def anagram(self, s, t):

return sorted(s) == sorted(t)

变形

问B中的所有字符是否都在A中：比如B="AABC"包含两个「A」，而A="ABCD"只包含一个「A」，故返回false.

For A = "ABCD", B = "ACD", return true.

For A = "ABCD", B = "AABC", return false.

def compareStrings(self, A, B):

letters = collections.defaultdict(int)

for a in A:

letters[a] += 1

for b in B:

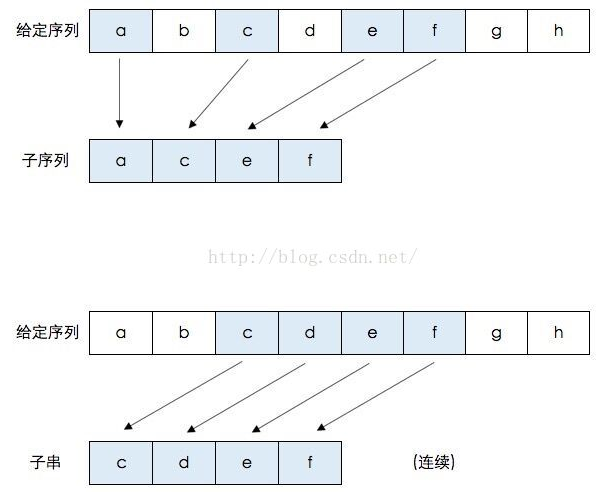
letters[b] -= 1

if b not in letters or letters[b] < 0:

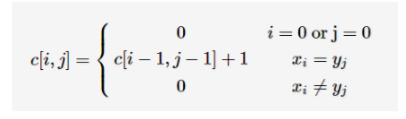
return False

return True

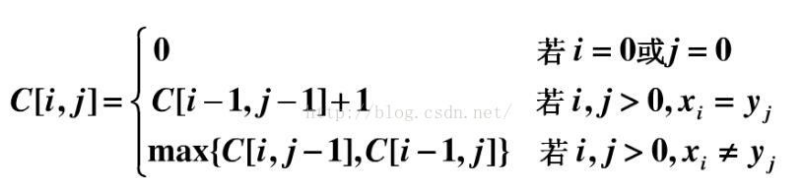
 首先需要科普一下，最长公共子序列（longest common sequence）和最长公共子串（longest common substring）不是一回事儿。什么是子序列呢？即一个给定的序列的子序列，就是将给定序列中零个或多个元素去掉之后得到的结果。什么是子串呢？给定串中任意个连续的字符组成的子序列称为该串的子串。给一个图再解释一下：

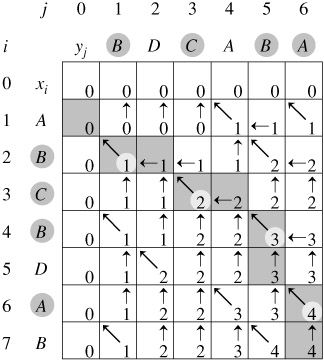


最长公共子串：



最长公共子序列：





int longestSubSeq(string x, string y) {

int \*\*dp = new int\*[x.size() + 1];

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

dp[i] = new int[y.size() + 1];

}

for (int i = 0; i <= x.size(); i++) dp[i][0] = 0;

for (int i = 0; i <= y.size(); i++) dp[0][i] = 0;

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

for (int j = 1; j <= y.size(); j++) {

if (x[i] == y[j]) {

dp[i][j] = dp[i - 1][j - 1] + 1;

}

else {

dp[i][j] = dp[i - 1][j] > dp[i][j - 1] ? dp[i - 1][j] : dp[i][j - 1];

}

}

}

return dp[x.size()][y.size()];

}

**Rotate String 旋转字符串**

给定一个字符串和一个偏移量，根据偏移量旋转字符串(从左向右旋转)

样例   
对于字符串 “abcdefg”.

offset=0 => “abcdefg”   
offset=1 => “gabcdef”   
offset=2 => “fgabcde”   
offset=3 => “efgabcd”

常见的翻转法应用题，仔细观察规律可知翻转的分割点在从数组末尾数起的offset位置。先翻转前半部分，随后翻转后半部分，最后整体翻转。

def rotateString(self, A, offset):

if A is None or len(A) == 0:

return A

offset %= len(A)

before = A[:len(A) - offset]

after = A[len(A) - offset:]

# [::-1] means reverse in Python

A = before[::-1] + after[::-1]

A = A[::-1]

return A

**Valid Palindrome**

给定字符串，判断该字符串是否是“回文”。只考虑字母和数字，并忽略字母大小写。   
注：空字符串也是回文

解法1：逐个比较首尾字符是否相等最为便利，即常见的『两根指针』技法。此题忽略大小写：  
def isPalindrome(self, s):

if not s:

return True

l, r = 0, len(s) - 1

while l < r:

# find left alphanumeric character

if not s[l].isalnum():

l += 1

continue

# find right alphanumeric character

if not s[r].isalnum():

r -= 1

continue

# case insensitive compare

if s[l].lower() == s[r].lower():

l += 1

r -= 1

else:

return False

#

return True

解法2：

def isPalindrome(self, s):

cleanlist = [c for c in s.lower() if c.isalnum()]

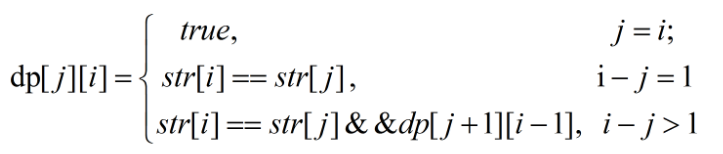
return cleanlist == cleanlist[::-1]

最长回文子串；

https://blog.csdn.net/shineboyxxb/article/details/52079360

https://www.cnblogs.com/leavescy/p/5878336.html

解法二： 动态规划



1. **class** Solution {
2. **public**:
3. string longestPalindrome(string s) {
4. **const** **int** n=s.size();
5. **bool** dp[n][n];
6. fill\_n(&dp[0][0],n\*n,**false**);
7. **int** max\_len=1; //保存最长回文子串长度
8. **int** start=0;//保存最长回文子串起点
9. **for**(**int** i=0;i<s.size();++i)
10. {
11. **for**(**int** j=0;j<=i;++j)
12. {
13. **if**(i-j<2)
14. dp[j][i]=(s[i]==s[j]);
15. **else**
16. dp[j][i]=(s[i]==s[j] && dp[j+1][i-1]);
17. **if**(dp[j][i] && max\_len<(i-j+1))
18. {
19. max\_len=i-j+1;
20. start=j;
21. }
22. }
23. }
24. **return** s.substr(start,max\_len);
25. }
26. };

解法三：中心扩展法

中心扩展就是把给定的字符串的每一个字母当做中心，向两边扩展，这样来找最长的子回文串。算法复杂度为O(N^2)。

但是要考虑两种情况：

1、像aba，这样长度为奇数。

2、想abba，这样长度为偶数。

public static String findLongestPalindrome2(String s){

int len = s.length();

int maxlength = 0;

int start = 0;

// 类似于aba这种情况，以i为中心向两边扩展

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

int j = i - 1;

int k = i + 1;

while(j >= 0 && k < len && s.charAt(j) == s.charAt(k)){

if(k - j + 1 > maxlength){

maxlength = k - j + 1;

start = j;

}

j --;

k ++;

}

}

// 类似于abba这种情况，以i，i+1为中心向两边扩展

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

int j = i;

int k = i + 1;

while(j >= 0 && k <len && s.charAt(j) == s.charAt(k)){

if(k - j + 1 > maxlength){

maxlength = k - j + 1;

start = j;

}

j --;

k ++;

}

}

if(maxlength > 0)

return s.substring(start, start + maxlength);

return null;

}

解法4 马拉车 有点复杂不看了

[Count and Say](https://leetcode.com/problems/count-and-say/)

The count-and-say sequence is the sequence of integers beginning as follows:  
1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.  
11 is read off as "two 1s" or 21.  
21 is read off as "one 2, then one 1" or 1211.

def countAndSay(self, n):

seq = '1'

for \_ in range(n - 1):

seq = re.sub(r'(.)\1\*', lambda m: str(len(m.group(0))) + m.group(1), seq)

return seq

删除指定元素

Given input array nums = [3,2,2,3], val = 3 删除所有的3 剩下[2,2]

## 题解1 - 两根指针从前往后遍历

使用两根指针从前往后依次遍历，一根指针遍历数组，另一根指针则指向数组当前不含给定值的索引。遍历时索引处的值不等于给定值则递增1，否则继续遍历，直至遍历结束，返回的索引值恰好是原地替换后的数组（不含给定值）的长度。

### Python

class Solution(object):

def removeElement(self, nums, val):

left = 0

for num in nums:

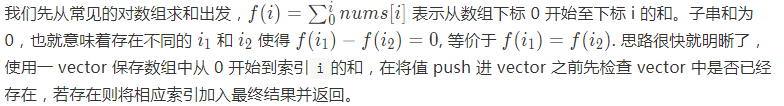
if num != val:

nums[left] = num

left += 1

给定一个整数数组，找到**和为零的子数组**。你的代码应该返回满足要求的子数组的起始位置和结束位置。

样例   
给出[-3, 1, 2, -3, 4]，返回[0, 2] 或者 [1, 3].



使用curr\_sum保存到索引i处的累加和，sum\_i保存不同索引处的和。执行sum\_i.push\_back之前先检查curr\_sum是否为0，再检查curr\_sum是否已经存在于sum\_i中。是不是觉得这种方法会比题解1好？错！时间复杂度是一样一样的！根本原因在于find操作的时间复杂度为线性。与这种方法类似的有哈希表实现，哈希表的查找在理想情况下可认为是 O(1)

也就是说sum\_i 不用数组，用hashmap

**Subarray Sum Closest**

这道题求的是和最接近于0的子序列

由于要求的子串和不一定，故哈希表的方法不再适用，

具体步骤如下：

1. 首先遍历一次数组求得子串和。
2. 对子串和排序。
3. 逐个比较相邻两项差值的绝对值，返回差值绝对值最小的两项。

比如原来数组[-3, 1, 1, -3, 5],

第一次遍历得到子串和

[-3，-2，-1，-4, 1]

排序：

A： [-4， -3，-2，-1，1]

A[i] - A[i-1]相差

[1， 1，1，2] 最小的就是1

**Product of Array Exclude Itself**

For A=[1, 2, 3], return [6, 3, 2].

除本身之外的其他元素的乘积

**解法：先遍历一次，对于每一个元素A[i]，先用result[i]表示其左边所有元素的乘积。再倒着遍历一次，用result[i] 再乘以其右边所有元素的乘积**

class Solution {

public:

vector<long long> productExcludeItself(vector<int> &nums) {

const int nums\_size = nums.size();

vector<long long> result(nums\_size, 1);

// solve the left part first

for (int i = 1; i < nums\_size; ++i) {

result[i] = result[i - 1] \* nums[i - 1];

}

// solve the right part

long long temp = 1;

for (int i = nums\_size - 1; i >= 0; --i) {

result[i] \*= temp;

temp \*= nums[i];

}

return result;

}

};

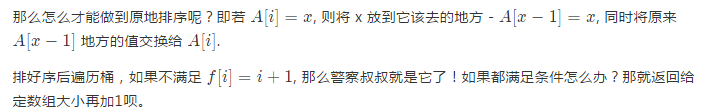
时间复杂度同上，空间复杂度为 *O*(1).

**First Missing Positive**

找到缺失的第一个正数For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

利用哈希表来求解，使用桶排序的思想排序，第一个桶放1，第二个桶放2，，但是都用到了空间，所有也不满足题意。

上面提到了哈希表，那样的话时间复杂度为线性的。这里稍微变通也下，我们也能得到类似的哈希表。



public class Solution {

public int firstMissingPositive(int[] nums) {

if (nums == null) return -1;

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

while (nums[i] > 0

&& nums[i] <= nums.length

&& nums[i] != i + 1

&& (nums[i] != nums[nums[i] - 1])) {

int temp = nums[nums[i] - 1];

nums[nums[i] - 1] = nums[i];

nums[i] = temp;

//其实就是 Swap(nums[nums[i] - 1], nums[i])

}

}

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

if (nums[i] != i + 1) return i + 1;

}

return 1 + nums.length;

}

}

**Two Sum**



Given nums = [2, 7, 11, 15], 给定目标和 = 9, 找到两个元素的和为9的pair对

Because nums[\*\*0\*\*] + nums[\*\*1\*\*] = 2 + 7 = 9,

return [\*\*0\*\*, \*\*1\*\*].

def twoSum(self, numbers, target):

hashdict = {}

for i, item in enumerate(numbers):

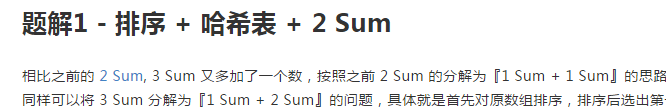
if (target - item) in hashdict:

return (hashdict[target - item] + 1, i + 1)

hashdict[item] = i

return (-1, -1)

**3 Sum**

i个元素，随后在剩下的元素中使用 2 Sum 的解法。

def threeSum(self, numbers):

triplets = []

length = len(numbers)

if length < 3:

return triplets

numbers.sort()

for i in xrange(length):

target = 0 - numbers[i]

# 2 Sum

hashmap = {}

twoSum(numbers[i+1:], target):

return triplets

**Remove Duplicates from Sorted Array**

使用两根指针(下标)，一个指针(下标)遍历数组，另一个指针(下标)只取不重复的数置于原数组中。

int newIndex = 0;

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

if (nums[i] != nums[newIndex]) {

newIndex++;

nums[newIndex] = nums[i];

}

}

**Remove Duplicates from Sorted Array II**

What if duplicates are allowed at most twice?

For example,

Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

在上题基础上加了限制条件元素最多可重复出现两次。 核心思想仍然是两根指针，只不过此时新索引自增的条件是当前遍历的数组值和『新索引』或者『新索引-1』两者之一不同。

for (int i = 2; i < nums.length; i++) {

if (nums[i] != nums[newIndex] || nums[i] != nums[newIndex - 1]) {

newIndex++;

nums[newIndex] = nums[i];

}

}

**Kth Largest Element in an Array**

For example,  
Given [3,2,1,5,6,4] and k = 2, return 5.

**Note:**  
You may assume k is always valid, 1 ≤ k ≤ array's length.

找第 K 大数，基于比较的排序的方法时间复杂度为 *O*(*n*), 数组元素无区间限定，故无法使用线性排序。由于只是需要找第 K 大数，这种类型的题通常需要使用快排的思想解决。 这里比较基准值最后的位置的索引值和 K 的大小关系即可递归求解。

**求中位数类似：**

故递归的终止条件即为当基准元素的位置(索引)满足中位数的条件时(左半部分长度为原数组长度一半

## 位操作

共有2\*n + 1个数，且有且仅有一个数落单，要找出相应的「单数」。

int result = 0;

for (int num : nums) {

result ^= num;

}

return result;

}

### 2的幂

For n=4, return true;

For n=5, return false;

2的整数幂若用二进制来表示，则其中必只有一个1，其余全是0，

n & (n - 1) 最低位为1的那位变为0。

x & (x - 1) 的含义为去掉二进制数中1的最后一位，无论 x 是正数还是负数都成立。

def checkPowerOf2(self, n):

if n < 1:

return False

else:

return (n & (n - 1)) == 0

比较两个数不同的比特位个数，显然容易想到可以使用异或处理两个整数，相同的位上为0，不同的位上为1，故接下来只需将异或后1的个数求出即可（用n &= (n - 1)循环）。

Given n = 31 (11111), m = 14 (01110), return 2.

**求阶乘数末尾0的个数**

11! = 39916800, so the out should be 2

比较准确的思路应该是计算质因数5和2的个数，取小的即可。质因数2的个数显然要大于5的个数，故只需要计算给定阶乘数中质因数中5的个数即可

count = 0

while n > 0:

n /= 5

count += n

return count

11/5 = 2 也就是说会出现5 10 两次 5的倍数

**Count 1 in Binary**

public int countOnes(int num) {

int count = 0;

while (num != 0) {

num = num & (num - 1);

count++;

}

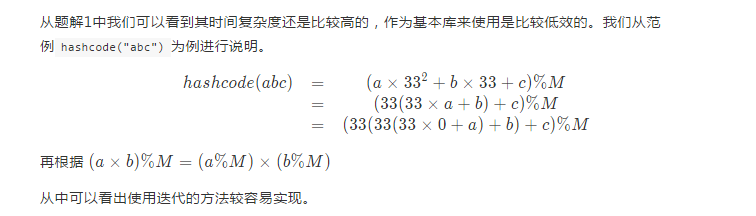
return count;

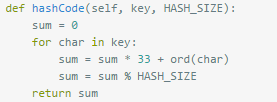
}

实现hash函数

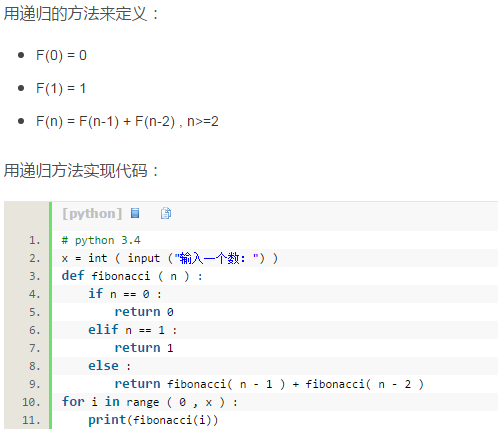
哈希函数是用来将一个字符串（或任何其他类型）转化为小于哈希表大小且大于等于零的整数。一个好的哈希函数可以尽可能少地产生冲突。一种广泛使用的哈希函数算法是使用数值33，假设任何字符串都是基于33的一个大整数。给出一个字符串作为key和一个哈希表的大小，返回这个字符串的哈希值。

基本实现题，大多数人看到题目的直觉是按照定义来递推不就得了嘛，但其实这里面大有玄机，因为在字符串较长时使用 long 型来计算33的幂会溢出！所以这道题的关键在于如何处理**大整数溢出**。对于整数求模，(a \* b) % m = a % m \* b % m 这个基本公式务必牢记。**（也就是说，不要先求和再%，每一步都取%）**

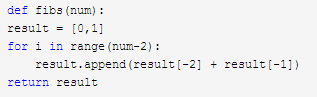




在数学上，**斐波纳契数**列以如下被以[递归](https://baike.baidu.com/item/%E9%80%92%E5%BD%92)的方法定义：F(0)=1，F(1)=1, F(n)=F(n-1)+F(n-2)



非递归



## Majority Element

分析：

抵消的思想，维护一个result和count,与result相同则count++，否则count--，到0时更新result的值，由于主元素个数较多，所有最后一定能留下。

代码：

[复制代码](javascript:void(0);)

1 class Solution {

2 public:

3 int majorityElement(vector<int>& nums) {

4 int result = 0;

5 int count = 0;

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

7 if (nums[i] == result && count != 0) {

8 count++;

9 }

10 else if (count == 0) {

11 result = nums[i];

12 count = 1;

13 }

14 else {

15 count--;

16 }

17 }

18 return result;

19 }

20 };

## Majority NumberII

Given an array of integers, the majority number is the number that occurs more than 1/3 of the size of the array. （Medium）

Find it.

Notice : There is only one majority number in the array.

分析：

同majority number1的思路相似，维护两个result和count，相同则对相应count操作，不同则均减一；

注意最后剩下两个元素，并不一定count值大的就一定是出现次数多的（可能另一个参与抵消过多），所以需要重新遍历一遍，对这个两个数比较出现次数大小。

**Majority Number III**

给定一个整型数组，找到主元素，它在数组中的出现次数严格大于数组元素个数的1/k。

从前一题的1/3变为1/k，道理还是一样，不过这次需要用一个hashmap来维护出现的次数，注意unordered\_map插入删除相关操作的写法即可。

尤其hashmap元素个数等于k需要删除的时候，需要维护一个vector存key，如果用iterator边走边删除可能出现位置变化。

思路和Majority NumberII 一样，维护k-1个candidate 在map里面，key为数字值，value为出现次数。先找到这k-1个candidate后，扫描所有元素，如果该元素存在在map里面，update map；如果不存在，1： 如果map里面有值为count= 0，那么删除掉这个元素，加入新元素；2：map里面没有0出现，那么就每个元素的count--

剩下的map里面的值都有可能是majority，所以重新扫描数组，记录下每一个元素出现次数，次数最大的就是majority

这道题让我们检测一个数是否为丑陋数，所谓丑陋数就是其质数因子只能是2,3,5。那么最直接的办法就是不停的除以这些质数，如果剩余的数字是1的话就是丑陋数了，

bool isUgly(int num) {

if (num <= 0) return false;

while (num % 2 == 0) num /= 2;

while (num % 3 == 0) num /= 3;

while (num % 5 == 0) num /= 5;

return num == 1;

}

是否回文数

本题很简单，算出x的倒置数a，比较a是否和x相等就行了。

class Solution {

public:

bool isPalindrome(int x) {

//算出x的倒置数a，比较a是否和x相等就行了

int a = 0, b = x;

while(b > 0){

a = a \* 10 + b % 10;

b /= 10;

}

if(a == x)

return true;

else

return false;

}

};

[[LeetCode] Wildcard Matching 外卡匹配](http://www.cnblogs.com/grandyang/p/4401196.html)

Implement wildcard pattern matching with support for '?' and '\*'.

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "\*") → true

isMatch("aa", "a\*") → true

isMatch("ab", "?\*") → true

isMatch("aab", "c\*a\*b") → false

**public** boolean isMatch(**String** s, **String** p) {

int i = 0, j = 0;

int stari = -1, starj = -1;

**while**(i < s.length()) {

// 1. match

**if**(j < p.length() && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '?')) {

i++; j++;

}

// 2. star

**else** **if**(j < p.length() && p.charAt(j) == '\*') {

// first match 0

stari = i;

starj = ++j;

}

// different number that '\*' matches

**else** **if**(stari != -1) {

// match number +1

i = ++stari;

j = starj;

}

// 3. not match and no star

**else** **return** **false**;

}

// remove last '\*' in p

**while**(j < p.length() && p.charAt(j) == '\*') j++;

**return** j == p.length();

}

[Partition Array](http://www.lintcode.com/en/problem/partition-array/)

这道题目给了我们一个数组有2n integers， 需要我们把这个数组分成n对，然后从每一对里面拿小的那个数字，把所有的加起来，返回这个sum。并且要使这个sum 尽量最大。如何让sum 最大化呢，我们想一下，如果是两个数字，一个很小，一个很大，这样的话，取一个小的数字，就浪费了那个大的数字。所以我们要使每一对的两个数字尽可能接近。我们先把nums sort 一下，让它从小到大排列，接着每次把index： 0， 2， 4...偶数位的数字加起来就可以了。

[First Missing Positive 首个缺失的正数](http://www.cnblogs.com/grandyang/p/4395963.html)

Given an unsorted integer array, find the first missing positive integer.

For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

Your algorithm should run in *O*(*n*) time and uses constant space.

这道题让我们找缺失的首个正数，由于限定了O(n)的时间，所以一般的排序方法都不能用，最开始我没有看到还限制了空间复杂度，所以想到了用哈希表来解，这个思路很简单，第一遍遍历数组把所有的数都存入哈希表中，并且找出数组的最大值，下次循环从1开始递增找数字，哪个数字找不到就返回哪个数字，如果一直找到了最大的数字，则返回最大值+1，代码如下：

[复制代码](javascript:void(0);)

// NOT constant space

class Solution {

public:

int firstMissingPositive(int A[], int n) {

if (n <= 0) return 1;

unordered\_map<int, int> m;

int mx = A[0];

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

if (A[i] > 0) {

m[A[i]] = 1;

mx = max(mx, A[i]);

}

}

for (int i = 1; i <= mx; ++i) {

if (m.find(i) == m.end()) return i;

}

return mx + 1;

}

};

[复制代码](javascript:void(0);)

但是上面的解法不是O(1)的时间复杂度，所以我们需要另想一种解法，既然不能建立新的数组，那么我们只能覆盖原有数组，我们的思路是把1放在数组第一个位置A[0]，2放在第二个位置A[1]，即需要把A[i]放在A[A[i] - 1]上，那么我们遍历整个数组，如果A[i] != i + 1, 而A[i]为整数且不大于n，另外A[i]不等于A[A[i] - 1]的话，我们将两者位置调换，如果不满足上述条件直接跳过，最后我们再遍历一遍数组，如果对应位置上的数不正确则返回正确的数，代码如下：

[复制代码](javascript:void(0);)

class Solution {

public:

int firstMissingPositive(int A[], int n) {

int i = 0;

while (i < n) {

if (A[i] != i + 1 && A[i] > 0 && A[i] <= n && A[i] != A[A[i] - 1]) {

swap(A[i], A[A[i] - 1]);

} else {

++ i;

}

}

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

if (A[i] != i + 1) return i + 1;

}

return n + 1;

}

};

[复制代码](javascript:void(0);)

* [(365) Count 1 in Binary](http://www.lintcode.com/en/problem/count-1-in-binary/) 二进制中1 的个数

Count how many 1 in binary representation of a 32-bit integer.

Example

Given 32, return 1

Given 5, return 2

Given 1023, return 9

x & (x - 1) 的含义为 把二进制数中最右边的1变为0

public int countOnes(int num) {

int count = 0;

while (num != 0) {

num = num & (num - 1);

count++;

}

return count;

}

**Remove Duplicates from Sorted List**

Given 1->1->2, return 1->2.  
Given 1->1->2->3->3, return 1->2->3.

def deleteDuplicates(self, head):

curt = head

while curt:

while curt.next and curt.next.val == curt.val:

curt.next = curt.next.next

curt = curt.next #更换当前节点为下一个节点继续遍历

return head

Given 1->2->3->3->4->4->5, return 1->2->5.  
Given 1->1->1->2->3, return 2->3.

上题为保留重复值节点的一个，这题删除全部重复节点，看似区别不大，但是考虑到链表头不确定(可能被删除，也可能保留)，因此若用传统方式需要较多的if条件语句。这里介绍一个**处理链表头节点不确定的方法——引入dummy node. （**start**）**

需要定义一个新的节点，然后链上原链表，然后定义一个前驱指针和一个现指针，每当前驱指针指向新建的节点，现指针从下一个位置开始往下遍历，遇到相同的则继续往下，直到遇到不同项时，把前驱指针的next指向下面那个不同的元素。如果现指针遍历的第一个元素就不相同，则把前驱指针向下移一位。

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (!head || !head->next) return head;

ListNode \*start = new ListNode(0);

start->next = head;

ListNode \*pre = start;

while (pre->next) {

ListNode \*cur = pre->next;

while (cur->next && cur->next->val == cur->val)

cur = cur->next;

if (cur != pre->next) pre->next = cur->next;

else pre = pre->next;

}

return start->next;

}

};

无序的去重

12->11->12->21->41->43->21,

then removeDuplicates() should convert the list to 12->11->21->41->43.

对于每一个节点，如果在哈希表中存在，则删掉，若不存在，则加入哈希表

用hash

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

ListNode \*pre = NULL, \*cur = head;

int hash[256] = {0};

while (cur) {

if (hash [cur->val] > 0) {

pre->next = cur->next;

} else {

++ hash [cur->val];

pre = cur; //保存当前节点为pre，用于下一轮比较

}

cur = cur->next; // 前节点后移

}

return head;

}

};