算法题私人笔记

Tyrael.wang

资料来源：编程之美，leetcode，lintcode，数据结构严蔚敏，算法导论，wiki，剑指Offer，网络

# 做题流程

1. 审题！审题！审题！

测试用例：把容易出错的测试用例先写出来

入参：

空参数：字符串为空，数组为空等

输入边界，分支处理

影响主逻辑：

0？负数？

数值范围（转化时容易出问题）？越界？是否需要用long？

输入对象可变？

是否保证必然有结果

1. 实现

确认算法、复杂度

列出算法步骤

列出需要注意的细节

再实现

容易出错的代码：

数值字面量是整型，注意计算不能越界，comparator注意越界

边界条件：大于还是大于等于

空指针；数组越界

**循环：不要忘记变量步进（尽量用for循环）**

1. 代码走查、单元测试

**小难度重沟通例题：**

**53. 翻转字符串**

输入字符串是否包括前导或者尾随空格？可以包括，但是反转后的字符不能包括

如何处理两个单词间的多个空格？在反转字符串中间空格减少到只含一个

# JAVA常用API

## 基本类型

字符

Character.isDigit(char ch)；isLetter；isLowerCase；isUpperCase

整数

public static int bitCount(int i)

*Returns the number of one-bits in the two's complement binary representation*

\*实现分析：https://blog.csdn.net/zhouzipeng000/article/details/56676885

## 集合

**1. 栈**Stack类继承Vector类。主要方法：push、pop、peek、empty**2. 队列**LinkedList实现了Queue接口。offer/add插入队尾；remove、poll;element、peek**4. 集合**

Collections.sort 排序List；Arrays.sortArrays.binSearch的返回值

*\** ***@return*** *index of the search key, if it is contained in the array;  
\* otherwise, <tt>(****-(<i>insertion point</i>) - 1****)</tt>. The  
\* <i>insertion point</i> is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or <tt>a.length</tt> if all elements in the array are less than the specified key.*

*Note that this guarantees that the return value will be &gt;= 0 if and only if the key is found.*

**private static int** binarySearch0(**int**[] a, **int** fromIndex, **int** toIndex, **int** key) {  
 **int** low = fromIndex;  
 **int** high = toIndex - 1;  
 **while** (low <= high) {  
 **int** mid = (low + high) >>> 1;  
 **int** midVal = a[mid];  
 **if** (midVal < key) low = mid + 1;  
 **else if** (midVal > key) high = mid - 1;  
 **else return** mid; *// key found* }  
 **return** -(low + 1); *// key not found.*}

6. 算法相关：**PriorityQueue** 基于堆实现的无界队列，非线程安全的

**Treemap**基于红黑树实现

floorKey/floorEntry:返回稍小的元素；ceilingKey/ceilingEntry：返回稍大的元素

Guava有range版本。

**BitSet**

\* This class implements a vector of bits that grows as needed. Each

\* component of the bit set has a boolean value. The

\* bits of a BitSet} are indexed by nonnegative integers.

\* Individual indexed bits can be examined, set, or cleared. One

\* BitSet} may be used to modify the contents of another

\* {@code BitSet} through logical AND, logical inclusive OR, and logical exclusive OR operations.

\* <p>By default, all bits in the set initially have the value false}.

\* <p>Every bit set has a current size, which is the number of bits

\* of space currently in use. Note that the size is

\* related to the implementation of a bit set, so it may change with implementation.

The **length** of a bit set relates to logical length of a bit set and is defined independently of implementation.

## 拾遗

**3. 字符串相关**StringBuffer 线程安全; StringBuilder 非线程安全字符串逆转 stringBuilder.reverse

1. **运算符**

&：按位与 |：按位或。

~：按位非。 ^：按位异或。

<<：左位移运算符。 >>：右位移运算符。

>>>：无符号右移运算符。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 优先级 | 运算符 | 名称或含义 | 使用形式 | 结合方向 | 说明 |
| 1 | [] | 数组下标 | 数组名[常量表达式] | 左到右 |  |
| () | 圆括号 | （表达式）/函数名(形参表) |  |
| . | 成员选择（对象） | 对象.成员名 |  |
| -> | 成员选择（指针） | 对象指针->成员名 |  |
| 2 | - | 负号运算符 | -常量 | 右到左 | 单目运算符 |
| (类型) | 强制类型转换 | (数据类型)表达式 |  |
| ++ | 自增运算符 | ++变量名 | 单目运算符 |
| -- | 自减运算符 | --变量名 | 单目运算符 |
| \* | 取值运算符 | \*指针变量 | 单目运算符 |
| & | 取地址运算符 | &变量名 | 单目运算符 |
| ! | 逻辑非运算符 | !表达式 | 单目运算符 |
| ~ | 按位取反运算符 | ~表达式 | 单目运算符 |
| sizeof | 长度运算符 | sizeof(表达式) |  |
| 3 | / | 除 | 表达式/表达式 | 左到右 | 双目运算符 |
| \* | 乘 | 表达式\*表达式 | 双目运算符 |
| % | 余数（取模） | 整型表达式/整型表达式 | 双目运算符 |
| 4 | + | 加 | 表达式+表达式 | 左到右 | 双目运算符 |
| - | 减 | 表达式-表达式 | 双目运算符 |
| **5** | **<<** | **左移** | **变量<<表达式** | **左到右** | **双目运算符** |
| **>>** | **右移** | **变量>>表达式** | **双目运算符** |
| 6 | > | 大于 | 表达式>表达式 | 左到右 | 双目运算符 |
| >= | 大于等于 | 表达式>=表达式 | 双目运算符 |
| < | 小于 | 表达式<表达式 | 双目运算符 |
| <= | 小于等于 | 表达式<=表达式 | 双目运算符 |
| 7 | == | 等于 | 表达式==表达式 | 左到右 | 双目运算符 |
| != | 不等于 | 表达式!= 表达式 | 双目运算符 |
| **8** | **&** | **按位与** | **表达式&表达式** | **左到右** | **双目运算符** |
| **9** | **^** | **按位异或** | **表达式^表达式** | **左到右** | **双目运算符** |
| **10** | **|** | **按位或** | **表达式|表达式** | **左到右** | **双目运算符** |
| 11 | && | 逻辑与 | 表达式&&表达式 | 左到右 | 双目运算符 |
| 12 | || | 逻辑或 | 表达式||表达式 | 左到右 | 双目运算符 |
| 13 | ?: | 条件运算符 | 表达式1? 表达式2: 表达式3 | 右到左 | 三目运算符 |
| 14 | = | 赋值运算符 | 变量=表达式 | 右到左 |  |
| /= | 除后赋值 | 变量/=表达式 |  |
| \*= | 乘后赋值 | 变量\*=表达式 |  |
| %= | 取模后赋值 | 变量%=表达式 |  |
| += | 加后赋值 | 变量+=表达式 |  |
| -= | 减后赋值 | 变量-=表达式 |  |
| <<= | 左移后赋值 | 变量<<=表达式 |  |
| >>= | 右移后赋值 | 变量>>=表达式 |  |
| &= | 按位与后赋值 | 变量&=表达式 |  |
| ^= | 按位异或后赋值 | 变量^=表达式 |  |
| |= | 按位或后赋值 | 变量|=表达式 |  |
| 15 | , | 逗号运算符 | 表达式,表达式,… | 左到右 | 从左向右顺序运算 |

同一优先级的运算符，运算次序由结合方向所决定。

简单记就是：！ > 算术运算符 > 关系运算符 > && > || > 赋值运算符

# 经典数据结构/算法

## 数学

### gcd

编程之美实现

**public int** gcd(**int** a, **int** b) {  
 **if** (a < 0 || b < 0) **throw new** RuntimeException(**"不支持负数"**);  
 **if** (a < b) **return** gcd(b, a);//确保前面的数较大  
 **if** (b == 0) **return** a;  
 **if** (a % 2 == 0) {  
 **if** (b % 2 == 0) **return** gcd(a >> 2, b >> 2) << 2;  
 **else return** gcd(a >> 2, b);  
 } **else** {  
 **if** (b % 2 == 0) **return** gcd(a, b >> 2);  
 **else return** gcd(a - b, b);  
 }  
}

gauva实现

*\** ***@throws*** *IllegalArgumentException if {****@code*** *a < 0} or {****@code*** *b < 0}***public static int** gcd(**int** a, **int** b) {  
 */\*  
 \* The reason we require both arguments to be >= 0 is because otherwise, what do you return on  
 \* gcd(0, Integer.MIN\_VALUE)? BigInteger.gcd would return positive 2^31, but positive 2^31 isn't an int.  
 \*/  
 checkNonNegative*(**"a"**, a);  
 *checkNonNegative*(**"b"**, b);  
 **if** (a == 0) **return** b;  
 *// 0 % b == 0, so b divides a, but the converse doesn't hold.* **if** (b == 0) **return** a;  */\*  
 \* Uses the binary GCD algorithm; see http://en.wikipedia.org/wiki/Binary\_GCD\_algorithm. This is >40% faster than the Euclidean algorithm in benchmarks.  
 \*/* **int** aTwos = Integer.*numberOfTrailingZeros*(a);  
 a >>= aTwos; *// divide out all 2s* **int** bTwos = Integer.*numberOfTrailingZeros*(b);  
 b >>= bTwos; *// divide out all 2s* **while** (a != b) { *// both a, b are odd  
 // The key to the binary GCD algorithm is as follows:  
 // Both a and b are odd. Assume a > b; then gcd(a - b, b) = gcd(a, b).  
 // But in gcd(a - b, b), a - b is even and b is odd, so we can divide out powers of two.  
 // We bend over backwards to avoid branching, adapting a technique from  
 // http://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax* **int** delta = a - b; *// can't overflow, since a and b are nonnegative* **int** minDeltaOrZero = delta & (delta >> (Integer.***SIZE*** - 1));  
 *// equivalent to Math.min(delta, 0)* a = delta - minDeltaOrZero - minDeltaOrZero; *// sets a to Math.abs(a - b)  
 // a is now nonnegative and even* b += minDeltaOrZero; *// sets b to min(old a, b)* a >>= Integer.*numberOfTrailingZeros*(a); *// divide out all 2s, since 2 doesn't divide b* }  
 **return** a << *min*(aTwos, bTwos);  
}

## 排序

#### 比较

其他重要排序：**计数排序：桶排序**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 | 特点 |
| 最好 | 平均 | 最坏 | 辅助存储 |  |  |
| 插入  排序 | 直接插入 | O(N) | O(N2) | O(N2) | O(1) | 稳定 |  |
| 希尔排序 | O(N) | N1.3 | O(N2) | O(1) | 不稳定 |  |
| 选择  排序 | 直接选择 | O(N2) | O(N2) | O(N2) | O(1) | 不稳定 |  |
| **堆排序** | **NlogN** | **NlogN** | **NlogN** | **O(1)** | **不稳定** |  |
| 交换  排序 | 冒泡排序 | O(N) | O(N2) | O(N2) | O(1) | 稳定 | 1、n小时好 2、最坏情况把逆序的数列变成顺序，最差时间复杂度O(N^2)只是表示其操作次数的数量级 3、最好的情况是数据本来就有序 |
| **快速排序** | **NlogN** | **NlogN** | **O(N2)** | **logn~n** | **不稳定** | 1、n大时好，快速排序比较占用内存。 2、最坏：划分后一边是1个，一边是n-1个 3、最好：每次都能均匀的划分 |
| **归并排序** | | **NlogN** | **NlogN** | **NlogN** | **O(n)** | **稳定** | **1、n大时好，归并比较占用内存，内存随n的增大而增大，但效率高且稳定。** |
| **基数排序** | | **d(r+n)** | **d(r+n)** | **d(r+n)** | **O(rd+n)** | **稳定** |  |

1 直接插入排序：比较次数 最少n-1次；最多(n-1)(n+2)/2

移动次数 最少0； 最多(n-1)(n+4)/2

2 折半插入排序：比较次数 最少与最多，都是n\*log2n，

移动次数 最少0，最多时间复杂度为O(n2);(n的平方，以下也如此表示)；

使用一个辅助存储空间，是稳定的排序；

3 冒泡排序： 比较最少为：n-1次，最多时间复杂度表示为o(n2);

移动次数最少为0，最多时间复杂度表示为O(n2);

4 简单选择排序： 比较次数没有多少之分，均是n(n-1)/2;

移动次数最少为0，最多为3(n-1);

5 快速排序：比较和移动次数最少时间复杂度表示为O(n\*log2n);

比较和移动次数最多的时间复杂度表示为O(n2);

6 堆排序： 比较和移动次数没有好坏之分，都是O(n\*log2n);

7 2-路归并排序：比较和移动次数没有好坏之分，都是O(n\*log2n);

需要n个辅助存储空间，是稳定的排序；

### 快排代码

**public class** QuickSort {  
 **final int**[] **a**;  
  
 **public** QuickSort(**int**[] a) {  
 **this**.**a** = a;  
 }  
  
 **public void** sort(){  
 **if** (**a** == **null** || **a**.**length** < 2) **return**;  
 sort(0, **a**.**length**-1);  
 }  
  
 **private void** sort(**int** start, **int** end){  
 **if** (start >= end) **return**;  
 **int** pivotIndex = partition(start, end);  
 sort(start, pivotIndex-1);  
 sort(pivotIndex+1, end);  
 }  
   
 **private int** partition(**int** start, **int** end){  
 **int** pivot = **a**[start];  
 **int** left = start, right = end;  
 **while** (left < right){  
 **while** (left < right && **a**[right] >= pivot) right--;  
 *//a[right] < pivot* **if** (left < right) **a**[left] = **a**[right];  
 **while**(left < right && **a**[left] <= pivot) left++;  
 **if** (left < right) **a**[right] = **a**[left];  
 }  
 **a**[left] = pivot;  
 **return** left;  
 }  
}

**324. Wiggle Sort II**：大小元素交叉摆

快排的应用：划分出大小分组。大小元素的交叉摆放。

### 归并排序代码

**public class** MergeSort {  
 **final int**[] **a**;  
  
 **public** MergeSort(**int**[] a) {  
 **this**.**a** = a;  
 }  
  
 **public void** sort(){  
 **if** (**a** == **null** || **a**.**length** < 2) **return**;  
 sort(0, **a**.**length**-1);  
 }  
  
 **private void** sort(**int** begin, **int** end){  
 **if** (begin >= end) **return**;  
 **int** mid = begin + (end -begin)>>1;  
 sort(begin, mid);  
 sort(mid+1, end);  
 *//归并* **int**[] sub1 = Arrays.*copyOfRange*(**a**, begin, mid +1);  
 **int**[] sub2 = Arrays.*copyOfRange*(**a**, mid+1, end+1);  
 **int** i = 0, iEnd = mid-begin, j = 0, jEnd = end - mid -1, aIndex= 0;  
 **while** (i <= iEnd && j<=jEnd){  
 **if** (sub1[i] < sub2[j]) **a**[aIndex++] = sub1[i++];  
 **else a**[aIndex++] = sub2[j++];  
 }  
 **while** (i <= iEnd) **a**[aIndex++] = sub1[i++];  
 **while** (j <= jEnd) **a**[aIndex++] = sub2[j++];  
 }  
}

**lintcode 64. 合并排序数组: A = [1, 2, 3, empty, empty], B = [4, 5]**

**解：从空的位置开始处理，避免大量移动**

### 堆排序代码

**public class** HeapSort {  
 **final int**[] **a**;  
  
 **public** HeapSort(**int**[] a) {  
 **this**.**a** = a;  
 }  
  
 */\*\*  
 \* 大根堆，从根开始往下沉  
 \** ***@param start*** *\** ***@param end*** *\*/* **public void** adjust(**int** start, **int** end) {  
 *//左孩子* **int** top = **a**[start];  
 **int** i;  
 **for** (i = 2 \* start + 1; i <= end; i = 2 \* (i + 1)) {  
 *//i 较大的孩子* **int** maxIndex = i;  
 **if** (i+1<= end && **a**[i] < **a**[i + 1]) {  
 maxIndex = i + 1;  
 }  
 **if** (**a**[maxIndex] > top) {  
 *//较大的值上浮* **a**[(i-1)/2] = **a**[maxIndex];  
 } **else** {  
 **a**[(i-1)/2] = top;  
 **return**;  
 }  
 }  
 **a**[(i-1)/2] = top;  
 }  
  
 **public void** init(**int**[] a){  
 **for** (**int** i = a.**length** /2 -1; i >= 0; i--) {  
 adjust(i, a.**length**-1);  
 }  
 }  
  
 */\*\*  
 \* 从大到小排队  
 \*/* **public void** sort(){  
 init(**a**);  
 **for** (**int** i = 0; i < **a**.**length**; i++) {  
 **int** temp = **a**[**a**.**length**-1-i];  
 **a**[**a**.**length** -1 -i] = **a**[0];  
 **a**[0] = temp;  
 adjust(0, **a**.**length**-1-i-1);  
 }  
 }  
}

### 计数排序

**791. Custom Sort String**

S and T are strings composed of lowercase letters. In S, no letter occurs more than once.

S was sorted in some custom order previously. We want to permute the characters of T so that they match the order that S was sorted. More specifically, if x occurs before y in S, then x should occur before y in the returned string.

Return any permutation of T (as a string) that satisfies this property.

**Example :**

**Input:**

S = "cba"

T = "abcd"

**Output:** "cbad"

**Explanation:**

"a", "b", "c" appear in S, so the order of "a", "b", "c" should be "c", "b", and "a".

Since "d" does not appear in S, it can be at any position in T. "dcba", "cdba", "cbda" are also valid outputs.

解：计数排序

### 外部排序：归并排序的拓展

**总体流程：**

1. 按内存大小，将总文件分成若干归并段，逐个输入内存排序，再重新写入外存。
2. 逐趟归并，知道整个文件有序。

**2-路平衡归并**

**外部排序的两两归并，不仅有merge过程，还要进行外存的读写。因为不能同时将所有输入和结果保存在内存中。**

**对外存的读写以物理块为单位**

**外部排序时间 = 内部排序时间（m\*tis）+ 外存读写时间（d\*tio）+ 内部归并时间（s+utmg）**

提高效率着眼于减少外存读写次数。增加归并路数、减少初始归并段数，可以减少读写次数。

**多路归并：**

单纯增加归并路数，会增加内部归并时间。

使用败者树可以解决内部归并时间问题。

败者树：每个非终端节点记录败者，胜者往上升。

**置换选择排序**

减少初始归并段数

基于败者树选择排序。选择最值的排序过程和输入输出并行进行。

输入队列fi，输出队列fo，内存大小w

初始，读入w个到内存。选择最小值记录为min，输出。

从输入文件补一个，选择比min大的元素中，最小的一个输出。直到无法输出。

解决比min大元素问题：增加一段号标记，数据进入内存和当前min比较，设置标记。

所得初始归并段的长度大约为内存的大小的两倍

**最佳归并树**

初始归并段长度不等的影响

确定几路归并后，构造赫夫曼树来减少外存读写次数。

越短的段离树根越远。

## 二叉树

### 性质

第i层有2^(i-1)个节点，i>=1.

一棵树至多有2^k-1个节点，k>=1。（引出满二叉树，完全二叉树（最后一层不满，靠左））

完全二叉树的高度是log2n/1 +1（/1下取整）

n0 = n2+1

n个节点不相似二叉树1/(n+1)C(2n)(n)

|  |  |  |
| --- | --- | --- |
|  | 0-based | 1-based |
| 数组表示法 | parent(i) = （i-1）/2  lchild(i) = 2i+1  rchild(i) = 2i+2 | parent(i) = i/2  lchild(i) = 2i  rchild(i) = 2i+1 |

### 遍历

**public class** IterativeInOrder {  
 Stack<TreeNode> **stack** = **new** Stack<>();  
  
 **public void** run(TreeNode root){  
 *//初始化* pushAll(root);  
 **while**(!**stack**.empty()){  
 TreeNode visit = **stack**.pop();  
 doWork(visit);//实际工作  
 pushAll(visit.**right**);  
 }  
 }  
  
 */\*\*  
 \* 沿左树，一直压到叶子  
 \** ***@param root*** *\*/* **private void** pushAll(TreeNode root){  
 TreeNode thisNode = root;  
 **while**(thisNode!= **null**){  
 **stack**.push(thisNode);  
 thisNode = thisNode.**left**;  
 }  
 }  
}

### 二叉搜索树Binary Search Tree

插入：沿根下降，找到空节点。

删除：

**450. Delete Node in a BST**

1. 叶子节点直接删除

2. 单孩节点：直接替代

3. 双孩节点：

方案一：前驱(后继)值替代，再删除前驱

方案二：左树替代，右树下降到最低

**线索二叉搜索树**

n个节点的二叉树，存在n+1个空指针域。

利用的时候，需要增加一个标记。

中序线索

查找后续：叶节点：右链；非叶节点：右子树左下。

查找前驱：叶节点：左链；非叶节点：做子树右下。

建立线索过程：

在遍历过程中改变空域指向。

习题：

https://leetcode.com/problems/unique-binary-search-trees/description/

C(2n,n)/(n+1)

1. 完全二叉樹、二叉搜索树

Given n, how many structurally unique BST's (binary search trees) that store values 1 ... n?

**unique-binary-search-trees-ii**：如何生成具体的树：递归。左子树从0到n。

98. Validate Binary Search Tree：中根序遍历，递增

230. Kth Smallest Element in a BST：中根序遍历，计数

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

增加一个计数域，修改时，更新后面的节点

### 红黑树：近似平衡二叉搜索树

搜索、前驱、后继、最小、最大、插入、删除：最快时间复杂度lgn

确保没有任何一条路径比其他路径长2倍

构造：

根节点，和底部节点指向NIL节点。

节点属性：color，key，left，right，parent

定义：

1 每个节点是红色或黑色

2 根节点是黑色的

3 nil节点是黑色的

4 红色节点的子节点是黑色的

5 任意节点到后代叶节点的路径上，黑节点数目相同

引理：有n个内部节点的红黑树的高度<=2lg(n+1)

**旋转：以链为支点，中间的子树更换父亲**

**插入：**

**1 普通插入，着色为红色**

**2 调整**

1. **父叔都为红色：父叔变黑色，爷变红=》递归，处理爷节点**
2. **父叔一红一黑：先左旋，再右旋。升高中间的节点。**

**删除：**

**\*贴图说明**

## 字典树

**public class** Node {  
 Node[] **children**;  
 String **leaf**;  
  
 */\*\*  
 \** ***@param n*** *孩子容量  
 \*/* **public** Node(**int** n) {  
 **children** = **new** Node[n];  
 }  
}

**public class** Trie {  
 Node **root**;  
  
 **public** Node create(){  
 **root** = **new** Node(26);  
 **return root**;  
 }  
  
 **public void** insert(String s){  
 *//逐个查找，找不到则建点* Node node = **root**;  
 **for** (**int** i = 0; i < s.length(); i++) {  
 **int** index = order(s.charAt(0));  
 Node child = node.**children**[index];  
 **if**(child == **null**){  
 child = **new** Node(26);  
 node.**children**[index] = child;  
 node = child;  
 }**else**{  
 node = child;  
 }  
 }  
 node.**leaf** = s;  
 }  
  
 **public boolean** search(String s){  
 Node node = **root**;  
 **for** (**int** i = 0; i < s.length(); i++) {  
 **int** index = order(s.charAt(0));  
 Node child = node.**children**[index];  
 **if**(child == **null**) **return false**;  
 **else** node = child;

}  
 *//如果该词是其他词的前缀，也不存在。* **return** node.**leaf** != **null**;  
 }  
  
 **public int** order(**char** a){  
 **return** a - **'a'**;  
 }  
}

**211. Add and Search Word - Data structure design**

变种：search(word) can search a literal word or a regular expression string containing only letters a-z or .. . means it can represent any one letter.

**648. Replace Words**

**677. Map Sum Pairs**

## 素数计算

650. 2 Keys Keyboard

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

1. Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).
2. Paste: You can paste the characters which are copied **last time**.

Given a number n. You have to get **exactly** n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

解：n = (CP…)(CPP…)(CPP…)=>分解n的因子

考虑p+q <= pq=》求质因子和

## 二分查找

含重复元素、等号、上下界

*/\*\*  
 \* 有序数组，返回值相等的，最大的索引  
 \*/***public class** MaxEqual **implements** IBinarySearch {  
 **public int** binarySearch(**int**[] arr, **int** needle){  
 **if** (arr== **null** || arr.**length** == 0) **return** -1;  
 **int** low =0, high = arr.**length**-1, mid;  
*//最终high = low-1* **while** (low <= high){  
 mid = low + ((high-low)>>1);  
 **if** (arr[mid] <= needle) low = mid +1;  
 **else** high = mid-1;  
 }  
 **if** (high <0 || arr[high] != needle) **return** -1;  
 **return** high;  
 }  
}

34. Search for a Range

## 流算法

### 摩尔多数投票算法

查找1/2水王

记录一个可能元素和对应的计数值。一次处理一个新元素。

如果计数值为0，则记录新来的元素为多数，计数值为1.

如果计数值不为0：新来的元素和旧元素相等，计数值+1；否则，计数值-1.

拓展：事先不知道是否存在超半数元素。

如果队列存在超半数值，则该值就是最终的可能值。如果不存在，则可以再遍历一趟，计数确定可能值是否超半数。

it is not possible for a sublinear-space algorithm to determine whether there exists a majority element in a single pass through the input

另一种思路：配对删除。最终状态：留下一个（就是水王）；两个（必然两个水王）

1/3水王

we need two candidates with top 2 frequency. If meeting different number from the candidate, then decrease 1 from its count, or increase 1 on the opposite condition. Once count equals 0, then switch the candidate to the current number. The trick is that we need to count again for the two candidates after the first loop. Finally, output the numbers appearing more than n/3 times.

另一种思路：配对删除。最终状态：如果最终留下两个元素，就无法判断了，所以还需要再遍历一边。

### 水库采样

从包含n个项目的集合S中选取k个样本，n为一很大或未知的数量，尤其适用于不能把所有n个项目都存放到主内存的情况。

Google面试题： I have a linked list of numbers of length N. N is very large and I don’t know in advance the exact value of N. How can I most efficiently write a function that will return k completely random numbers from the list（中文简化的意思就是：**在不知道文件总行数的情况下，如何从文件中随机的抽取一行？**）。

**解**：定义取出的行号为choice，第一次直接以第一行作为取出行 choice ，第二次以二分之一概率决定是否用第二行替换 choice ，第三次以三分之一的概率决定是否以第三行替换 choice ……，以此类推。由上面的分析我们可以得出结论，**在取第n个数据的时候，我们生成一个0到1的随机数p，如果p小于1/n，保留第n个数。大于1/n，继续保留前面的数。直到数据流结束，返回此数，算法结束。**

**证明：**

**问题一**

首先考虑k为1的情况。

设当前读取的是第n个元素，采用归纳法分析如下：

1. n = 1 时，只有一个元素，直接返回即可，概率为1。
2. n = 2 时，需要等概率返回前两个元素，显然概率为1/2。可以生成一个0～1之间的随机数p，p < 0.5 时返回第一个，否则返回第二个。
3. n = 3 时，要求每个元素返回的概率为1/3。此时前两个元素留下来的概率均为1/2。做法是：生成一个0～1之间的随机数，若<1/3，则返回第三个，否则返回上一步留下的那个。元素1和2留下的概率均为：1/2 \* (1 - 1/3) = 1/3，即上一步留下的概率乘以这一步留下（即元素3不留下）的概率。
4. 假设 n = m 时，前n个元素留下的概率均为：1/n = 1/m；
5. 那么 n = m+1 时，生成0～1之间的随机数并判断是否<1/(m+1)，若是则留下元素m+1，否则留下上一步留下的元素。这样一来，元素m+1留下的概率为1/(m+1)，前m个元素留下来的概率均为：1/m \* (1 - 1/(m+1)) = 1/(m+1)，也就是1/n。

**问题二**

最终返回的元素有k个。要求是：取到第n个元素时，前n个元素被留下的几率相等，即k/n。

在取第n个数据的时候，我们生成一个0到1的随机数p，如果p小于k/n，替换池中任意一个为第n个数。大于k/n，继续保留前面的数。直到数据流结束，返回此k个数。但是**为了保证计算机计算分数额准确性，一般是生成一个0到n的随机数，跟k相比，道理是一样的**。

1. 初始情况 n <= k：此时每个元素留下的概率均为1。
2. 当 n = k+1 时，第k+1个元素留下的概率为k/(k+1)，前k个元素留下的概率均为：k/k \* (1 - k/(k+1) \* 1/k) = k/(k+1)，即上一步留下的概率乘以这一步留下的概率。
3. 假设 n = m 时，每个元素留下的概率均为 k/n = k/m。
4. 那么，当 n = m+1 时，第m+1个元素留下的概率为1/(m+1)，前m个元素留下的概率均为：k/m \* (1 - k/(m+1) \* 1/k) = k/(m+1)，其中：k/m为上一步留下来的概率，k/(m+1) \* 1/k 为这一步不能留下来的概率（第m+1个留下来，同时池中一个元素被踢出的概率）。

伪代码如下：

//stream代表数据流

//reservoir代表返回长度为k的池塘；从stream中取前k个放入reservoir；

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

reservoir[i] = stream[i];

for (i = k; stream != null; i++) {

p = random(0, i);

if (p < k) reservoir[p] = stream[i];

return reservoir;

## 下一个排列

**public** **void** **nextPermutation(int[]** nums**)** **{**

**int** i **=** nums**.**length **-** 2**;**

**//找到第一个不满足逆序的数**

**while** **(**i **>=** 0 **&&** nums**[**i **+** 1**]** **<=** nums**[**i**])** i**--;**

**//i之后的数是单调递减的。**

**if** **(**i **>=** 0**)** **{**

**int** j **=** nums**.**length **-** 1**;**

**//找到比i位置大一点的数**

**while** **(**j **>=** 0 **&&** nums**[**j**]** **<=** nums**[**i**])** j**--;**

swap**(**nums**,** i**,** j**); //i位置确定好**

**}**

reverse**(**nums**,** i **+** 1**); //余下的按正序排列即可**

**}**

## 树状数组/binary indexed tree/ Fenwick tree

1-based

**问题：求一个数组中连续n项的和。多次计算不同区间。**

**（经过预处理）有O（nlogn）的解法**

如果需要大量的求和操作，比如第一次求下标（1，1234）的和第二次求下标（2，1024）的和，很容易发现在第一次计算的过程中（2，1024）的和是计算过的。

树状数组是一个查询和修改复杂度都为log(n)的数据结构。用于查询任意两位之间的所有元素之和，但是每次只能修改一个元素的值。

**核心思想:**

* 树状数组中的每个元素是原数组中一个或者多个连续元素的和。
* 在进行连续求和操作a[1]+…+a[n]时，只需要将树状数组中某几个元素的和即可。



a[]: 保存原始数据的数组   
e[]: 树状数组，其中的任意一个元素e[i]可能是一个或者多个a数组中元素的和。如e[2]=a[1]+a[2]; e[3]=a[3]，e[4]=a[1]+a[2]+a[3]+a[4]。   
e[i]中的元素：如果数字 i 的二进制表示中末尾有**k个0，则e[i]是a数组中2^k个元素的和**，则e[i]=a[**i-2^k+1**]+a[i-2^k+2]+…+a[i-1]+a[i]。也就是说，**e[i]中每一个元素管理着a[]中若干个元素的和，并且各个元素管理的区间没有重叠。**

　　　　如：4=100(2)　　e[4]=a[1]+a[2]+a[3]+a[4];   
　　　　　　6=110(2)　　e[6]=a[5]+a[6]   
　　　　　　7=111(2)　　e[7]=a[7]

计算2^k的两个方法

* 2^k = (i & (-i)); (利用机器补码特性)
* 2^k = (i & (i^(i-1));

**父节点**

是离它最近的，且编号末位连续0比它多的就是它的父亲,如e[2]是e[1]的父亲；e[4]是e[2]的父亲。   
e[4] = e[2]+e[3]+a[4] = a[1]+a[2]+a[3]+a[4] ，e[2]、e[3]的后继就是e[4]。

**计算方法**

lowbit(i) = ( (i-1) ^ i) & i ; //或(i & (-i))   
**节点e[i]的父节点为 e[ i + lowbit(i) ]**

**子节点**

最近的，编号即为比自己小的，最末连续0比自己少的节点。如e[7]的子节点是e[6],e[6]的子节点是e[4]

**计算方法**

lowbit(i) = ( (i-1) ^ i) & i ; //或者(i & (-i))   
**节点e[i]的子节点为 e[ i - lowbit(i) ]**

**实现代码**

**public class** FenwickTree {  
 *//1-based* **public final int**[] **tree**;  
 *//0-based* **final int**[] **src**;  
  
 **public** FenwickTree(**int**[] src) {  
 **this**.**tree** = **new int**[src.**length**+1];  
 **this**.**src** = src;  
 **for** (**int** i = 1; i < **tree**.**length**; i++) {  
 *//十进制，i的势力范围* **int** lowBit = i & -i;  
 **for** (**int** j = i - lowBit +1; j <= i; j++) **tree**[i] += **src**[j-1];  
 }  
 }  
  
 **public void** update(**int** index, **int** value){  
 **int** diff = value - **src**[index];  
 **src**[index] = value;  
 **for** (**int** i = index+1; i < **tree**.**length**; i += i& -i) **tree**[i] += diff;  
 }  
  
 */\*\*  
 \** ***@param end*** *inclusive  
 \** ***@return*** *\*/* **public int** sum(**int** end){  
 **int** sum = 0;  
 **for** (**int** i = end+1; i > 0; i -= i&-i) sum+= **tree**[i];  
 **return** sum;  
 }  
  
 */\*\*  
 \*  
 \** ***@param start*** *inclusive  
 \** ***@param end*** *inclusive  
 \** ***@return*** *\*/* **public int** sum(**int** start, **int** end){  
 **return** sum(end) - sum(start-1);  
 }  
}

## 树

树的表示法

1双亲表示法：数组

2 孩子表示法

3 孩子兄弟表示法

**310. Minimum Height Trees**

a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

//确定树根

We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

## 线段树

**307. Range Sum Query – Mutable**

<https://leetcode.com/problems/range-sum-query-mutable/solution/>

https://en.wikipedia.org/wiki/Segment\_tree

Segment tree is used to solve numerous range query problems like finding minimum, maximum, sum, greatest common divisor, least common denominator in array in logarithmic time.



The segment tree for array *a*[0,1,…,*n*−1] is a binary tree in which each node contains **aggregate** information (min, max, sum, etc.) for a subrange [*i*…*j*] of the array, as its left and right child hold information for range  [*i*… (*i*+*j)/2*] and [(*i*+*j)/2*+1,*j*].

Segment tree could be implemented using either an array or a tree.

In the example above, every leaf node contains the initial array elements {2,4,5,7,8,9}. The root (35) being the sum of its children (6) and (29), holds the total sum of the entire array.

使用步骤:

**1. Build segment tree：bottom-up approach**

to find the sum of node p, we need to calculate the sum of its right and left child in advance.

We begin from the leaves, initialize them with input array elements *a*[0,1,…,*n*−1]. Then we move upward to the higher level to calculate the parents' sum till we get to the root of the segment tree.

//注意从1开始

**int[]** tree**;**

**int** n**;**

**public** **NumArray(int[]** nums**)** **{**

n **=** nums**.**length**;**

tree **=** **new** **int[**n **\*** 2**];**

**for** **(int** i **=** n**,** j **=** 0**;** i **<** 2 **\*** n**;** i**++,** j**++)**

tree**[**i**]** **=** nums**[**j**];//后面的一半是叶子节点**

**for** **(int** i **=** n **-** 1**;** **i > 0;** **--**i**)**

**tree[i] = tree[i \* 2] + tree[i \* 2 + 1];//主干节点**

**}**

**Complexity Analysis**

* Time complexity :*O*(*n*)

we calculate the sum of one node during each iteration of the for loop. There are approximately 2*n* nodes.

This could be proved in the following way: Segmented tree for array with nelements has n leaves (the array elements itself). The number of nodes in each level is half the number in the level below.

So if we sum the number by level we will get:

n + n/2 + n/4 + n/8 + \ldots + 1 \approx 2n*n*+*n*/2+*n*/4+*n*/8+…+1≈2*n*

* Space complexity : O(n).

**2. Update segment tree：**bottom-up approach

there are tree nodes which contain the sum of the modified element. We update the leaf node that stores a[i]. From there we will follow the path up to the root updating the value of each parent as a sum of its children values.

**void** **update(int** pos**,** **int** val**)** **{**

**pos += n;**

tree**[**pos**]** **=** val**;**

**while** **(**pos **>** 0**)** **{**

**int** left, right**;**

**if** **(**pos **%** 2 **==** 0**)** **{**

left **=** pos**;**

right **=** pos **+** 1**;**

**}** **else** **{**

left **=** pos **-** 1**;**

right **=** pos**;**

**}**

*// parent is updated after child is updated*

tree**[**pos **>>** 1**]** **=** tree**[**left**]** **+** tree**[**right**];**

pos **>>** 1**;**

**}**

**}**

**Complexity Analysis**

* Time complexity : *O*(log*n*).

because there are a few tree nodes with range that include I th array element, one on each level. There are log(*n*) levels.

* Space complexity : O(1).

**3. Range Sum Query**

We can find range sum query [L, R]:

Algorithm hold loop invariant:

l, *r* and sum of [*L*…*l*] and [*r*…*R*] has been calculated, where land rare the left and right boundary of calculated sum. Initially we set l with left leaf *L* and *r* with right leaf *R*. Range [l, r] hrinks on each iteration **till range borders meets**

* Loop till l ≤*r*
  + Check if *l* is right child of its parent *P*
    - l*l* is right child of P*P*. Then P*P* contains sum of range of l*l* and another child which is outside the range [l, r][*l*,*r*] and we don't need parent P*P* sum. Add l*l* to *sum* without its parent P*P* and set l*l* to point to the right of P*P* on the upper level.
    - l*l* is not right child of P*P*. Then parent P*P* contains sum of range which lies in [l, r][*l*,*r*]. Add P*P* to sum*sum*and set l*l* to point to the parent of P*P*
  + Check if r*r* is left child of its parent P*P*

**public** **int** **sumRange(int** l**,** **int** r**)** **{**

*// get leaf with value 'l'*

l **+=** n**;**

r **+=** n**;**

**int** sum **=** 0**;**

**while** **(**l **<=** r**)** **{**

**if** **(**l **%** 2 **==** 1**)** **{//l是右孩子，不包含缩减后的上层中。**

sum **+=** tree**[**l**];**

l**++;**

**}**

**if** **(**r **%** 2 **==** 0**)** **{//r是左孩子**

sum **+=** tree**[**r**];**

r**--;**

**}**

**l >>= 1;**

**r >>= 1;**

**}**

**return** sum**;**

**}**

**Complexity Analysis**

* Time complexity :  *O*(log*n*)

because on each iteration of the algorithm we move one level up, either to the parent of the current node or to the next sibling of parent to the left or right direction till the two boundaries meet. In the worst-case scenario this happens at the root after nlog*n* iterations.

* Space complexity :  *O*(1).

**308．Range Sum Query 2D – Mutable**

2维树状数组/线段树

## 分桶法和平方分割

<https://leetcode.com/problems/range-sum-query-mutable/solution/>

思路就是避免重复计算

分割成sqrt（n）个桶

## 图

### 图的表示

1 2D数组

2 邻接表：顶点数组+边链表

3 十字链表：邻接表+逆邻接表

4 邻接多重表

133. Clone Graph

节点映射，记录对应关系？

### 二分图Bipartite graph/Bigraph

[](https://en.wikipedia.org/wiki/File:Simple-bipartite-graph.svg)

Example of a bipartite graph without cycles

[](https://en.wikipedia.org/wiki/File:Biclique_K_3_5.svg)

A [complete bipartite graph](https://en.wikipedia.org/wiki/Complete_bipartite_graph) with m = 5 and n = 3

A **bigraph** is a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) whose [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) can be divided into two [disjoint](https://en.wikipedia.org/wiki/Disjoint_sets) and [independent sets](https://en.wikipedia.org/wiki/Independent_set_(graph_theory)) U and V such that every [edge](https://en.wikipedia.org/wiki/Edge_(graph_theory)) connects a vertex in U to one in V. Vertex sets U  and V are usually called the *parts* of the graph. Equivalently, a bipartite graph is a graph that does not contain any odd-length [cycles](https://en.wikipedia.org/wiki/Cycle_(graph_theory)).[[1]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-diestel2005graph-1)[[2]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-2)

The two sets U and V may be thought of as a [coloring](https://en.wikipedia.org/wiki/Graph_coloring) of the graph with two colors: if one colors all nodes in U blue, and all nodes in V green, each edge has endpoints of differing colors, as is required in the graph coloring problem.[[3]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-adh98-7-3)[[4]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-s12-4) In contrast, such a coloring is impossible in the case of a non-bipartite graph, such as a [triangle](https://en.wikipedia.org/wiki/Gallery_of_named_graphs): after one node is colored blue and another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color.

One often writes G=(U,V,E) to denote a bipartite graph with E denoting the edges of the graph. If a bi graph is not [connected](https://en.wikipedia.org/wiki/Connected_graph), it may have more than one bipartition;[[5]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-5) in this case, the (U,V,E) notation is helpful in specifying one particular bipartition. If |U|=|V| , that is, if the two subsets have equal [cardinality](https://en.wikipedia.org/wiki/Cardinality), then G is called a *balanced* bipartite graph.[[3]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-adh98-7-3) If all vertices on the same side of the bipartition have the same [degree](https://en.wikipedia.org/wiki/Degree_(graph_theory)), then G is called [biregular](https://en.wikipedia.org/wiki/Biregular_graph).

**Contents**

* [1Examples](https://en.wikipedia.org/wiki/Bipartite_graph#Examples)
* [2Properties](https://en.wikipedia.org/wiki/Bipartite_graph#Properties)
  + [2.1Characterization](https://en.wikipedia.org/wiki/Bipartite_graph#Characterization)
  + [2.2König's theorem and perfect graphs](https://en.wikipedia.org/wiki/Bipartite_graph#K%C3%B6nig's_theorem_and_perfect_graphs)
  + [2.3Degree](https://en.wikipedia.org/wiki/Bipartite_graph#Degree)
  + [2.4Relation to hypergraphs and directed graphs](https://en.wikipedia.org/wiki/Bipartite_graph#Relation_to_hypergraphs_and_directed_graphs)
* [3Algorithms](https://en.wikipedia.org/wiki/Bipartite_graph#Algorithms)
  + [3.1Testing bipartiteness](https://en.wikipedia.org/wiki/Bipartite_graph#Testing_bipartiteness)
  + [3.2Odd cycle transversal](https://en.wikipedia.org/wiki/Bipartite_graph#Odd_cycle_transversal)
  + [3.3Matching](https://en.wikipedia.org/wiki/Bipartite_graph#Matching)
* [4Additional applications](https://en.wikipedia.org/wiki/Bipartite_graph#Additional_applications)

Examples[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=1)]

When modelling relations between two different classes of objects, bipartite graphs very often arise naturally. For instance, a graph of football players and clubs, with an edge between a player and a club if the player has played for that club, is a natural example of an *affiliation network*, a type of bipartite graph used in [social network analysis](https://en.wikipedia.org/wiki/Social_network_analysis).[[6]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-6)

Another example is in the ([NP-complete](https://en.wikipedia.org/wiki/NP-complete)) railway optimization problem, in which the input is a schedule of trains and their stops, and the goal is to find a set of train stations as small as possible such that every train visits at least one of the chosen stations. This problem can be modeled as a [dominating set](https://en.wikipedia.org/wiki/Dominating_set) problem in a bipartite graph that has a vertex for each train and each station and an edge for each pair of a station and a train that stops at that station.[[7]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-niedermeier2006invitiation-7)

A third example is in the academic field of numismatics. Ancient coins are made using two positive impressions of the design (the obverse and reverse). The charts numismatists produce to represent the production of coins are bipartite graphs. [[8]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-bracey2012-8)

More abstract examples include the following:

* Every [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) is bipartite.[[4]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-s12-4)
* [Cycle graphs](https://en.wikipedia.org/wiki/Cycle_graph) with an even number of vertices are bipartite.[[4]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-s12-4)
* Every [planar graph](https://en.wikipedia.org/wiki/Planar_graph) whose [faces](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#Genus) all have even length is bipartite.[[9]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-9) Special cases of this are [grid graphs](https://en.wikipedia.org/wiki/Grid_graph) and [squaregraphs](https://en.wikipedia.org/wiki/Squaregraph), in which every inner face consists of 4 edges and every inner vertex has four or more neighbors.[[10]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-10)
* The [complete bipartite graph](https://en.wikipedia.org/wiki/Complete_bipartite_graph) on *m* and *n* vertices, denoted by *Kn,m* is the bipartite graph G=(U,V,E), where *U* and *V* are disjoint sets of size *m* and *n*, respectively, and *E* connects every vertex in *U* with all vertices in *V*. It follows that *Km,n* has *mn* edges.[[11]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-11) Closely related to the complete bipartite graphs are the [crown graphs](https://en.wikipedia.org/wiki/Crown_graph), formed from complete bipartite graphs by removing the edges of a [perfect matching](https://en.wikipedia.org/wiki/Perfect_matching).[[12]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-12)
* [Hypercube graphs](https://en.wikipedia.org/wiki/Hypercube_graph), [partial cubes](https://en.wikipedia.org/wiki/Partial_cube), and [median graphs](https://en.wikipedia.org/wiki/Median_graph) are bipartite. In these graphs, the vertices may be labeled by [bitvectors](https://en.wikipedia.org/wiki/Bitvector), in such a way that two vertices are adjacent if and only if the corresponding bitvectors differ in a single position. A bipartition may be formed by separating the vertices whose bitvectors have an even number of ones from the vertices with an odd number of ones. Trees and squaregraphs form examples of median graphs, and every median graph is a partial cube.[[13]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-13)

Properties[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=2)]

**Characterization**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=3)]

Bipartite graphs may be characterized in several different ways:

* A graph is bipartite ⬄ it does not contain an [odd cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)).[[14]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-14)
* A graph is bipartite ⬄ it is 2-colorable, (i.e. its [chromatic number](https://en.wikipedia.org/wiki/Chromatic_number) is less than or equal to 2).[[3]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-adh98-7-3)
* The [spectrum](https://en.wikipedia.org/wiki/Spectral_graph_theory) of a graph is symmetric ⬄ it's a bipartite graph.[[15]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-15)

**König's theorem and perfect graphs**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=4)]

In bipartite graphs, the size of [minimum vertex cover](https://en.wikipedia.org/wiki/Minimum_vertex_cover) is equal to the size of the [maximum matching](https://en.wikipedia.org/wiki/Maximum_matching);

An equivalent form is that the size of the [maximum independent set](https://en.wikipedia.org/wiki/Maximum_independent_set) plus the size of the maximum matching is equal to the number of vertices. In any graph without [isolated vertices](https://en.wikipedia.org/wiki/Isolated_vertex) the size of the [minimum edge cover](https://en.wikipedia.org/wiki/Minimum_edge_cover) plus the size of a maximum matching equals the number of vertices.[[18]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-18) Combining this equality with König's theorem leads to the facts that, in bipartite graphs, the size of the minimum edge cover is equal to the size of the maximum independent set, and the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.

Another class of related results concerns [perfect graphs](https://en.wikipedia.org/wiki/Perfect_graph): every bipartite graph, the [complement](https://en.wikipedia.org/wiki/Complement_(graph_theory)) of every bipartite graph, the [line graph](https://en.wikipedia.org/wiki/Line_graph) of every bipartite graph, and the complement of the line graph of every bipartite graph, are all perfect. Perfection of bipartite graphs is easy to see (their [chromatic number](https://en.wikipedia.org/wiki/Chromatic_number) is two and their [maximum clique](https://en.wikipedia.org/wiki/Maximum_clique) size is also two) but perfection of the [complements](https://en.wikipedia.org/wiki/Complement_(graph_theory)) of bipartite graphs is less trivial, and is another restatement of König's theorem. This was one of the results that motivated the initial definition of perfect graphs.[[19]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-19) Perfection of the complements of line graphs of perfect graphs is yet another restatement of König's theorem, and perfection of the line graphs themselves is a restatement of an earlier theorem of König, that every bipartite graph has an [edge coloring](https://en.wikipedia.org/wiki/Edge_coloring) using a number of colors equal to its maximum degree.

According to the [strong perfect graph theorem](https://en.wikipedia.org/wiki/Strong_perfect_graph_theorem), the perfect graphs have a [forbidden graph characterization](https://en.wikipedia.org/wiki/Forbidden_graph_characterization) resembling that of bipartite graphs: a graph is bipartite if and only if it has no odd cycle as a subgraph, and a graph is perfect if and only if it has no odd cycle or its [complement](https://en.wikipedia.org/wiki/Complement_(graph_theory)) as an [induced subgraph](https://en.wikipedia.org/wiki/Induced_subgraph). The bipartite graphs, line graphs of bipartite graphs, and their complements form four out of the five basic classes of perfect graphs used in the proof of the strong perfect graph theorem.[[20]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-20)

**Degree**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=5)]

For a vertex, the number of adjacent vertices is called the [degree](https://en.wikipedia.org/wiki/Degree_(graph_theory)) of the vertex and is denoted deg(v). The *degree sum formula* for a bipartite graph states that

C:\Users\ADMINI~1\AppData\Local\Temp\1532140039(1).png

The degree sequence of a bipartite graph is the pair of lists each containing the degrees of the two parts U and V. For example, the complete bipartite graph *K*3,5 has degree sequence (5,5,5),(3,3,3,3,3) .

Isomorphic bipartite graphs have the same degree sequence. However, the degree sequence does not, in general, uniquely identify a bipartite graph; in some cases, non-isomorphic bipartite graphs may have the same degree sequence.

The [bipartite realization problem](https://en.wikipedia.org/wiki/Bipartite_realization_problem) is the problem of finding a simple bipartite graph with the degree sequence being two given lists of natural numbers. (Trailing zeros may be ignored since they are trivially realized by adding an appropriate number of isolated vertices to the digraph.)

**Relation to hypergraphs and directed graphs**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=6)]

The [biadjacency matrix](https://en.wikipedia.org/wiki/Adjacency_matrix_of_a_bipartite_graph) of a bipartite graph (U,V,E) is a [(0,1) matrix](https://en.wikipedia.org/wiki/(0,1)_matrix) of size |U|\*|V| that has a one for each pair of adjacent vertices and a zero for nonadjacent vertices.[[21]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-21) Biadjacency matrices may be used to describe equivalences between bipartite graphs, hypergraphs, and directed graphs.

A [hypergraph](https://en.wikipedia.org/wiki/Hypergraph) is a combinatorial structure that, like an undirected graph, has vertices and edges, but in which the edges may be arbitrary sets of vertices rather than having to have exactly two endpoints. A bipartite graph {\displaystyle (U,V,E)} may be used to model a hypergraph in which *U* is the set of vertices of the hypergraph, *V* is the set of hyperedges, and *E* contains an edge from a hypergraph vertex *v* to a hypergraph edge *e* exactly when *v* is one of the endpoints of *e*. Under this correspondence, the biadjacency matrices of bipartite graphs are exactly the [incidence matrices](https://en.wikipedia.org/wiki/Incidence_matrix) of the corresponding hypergraphs. As a special case of this correspondence between bipartite graphs and hypergraphs, any [multigraph](https://en.wikipedia.org/wiki/Multigraph) (a graph in which there may be two or more edges between the same two vertices) may be interpreted as a hypergraph in which some hyperedges have equal sets of endpoints, and represented by a bipartite graph that does not have multiple adjacencies and in which the vertices on one side of the bipartition all have [degree](https://en.wikipedia.org/wiki/Degree_(graph_theory)) two.[[22]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-22)

A similar reinterpretation of adjacency matrices may be used to show a one-to-one correspondence between [directed graphs](https://en.wikipedia.org/wiki/Directed_graph) (on a given number of labeled vertices, allowing self-loops) and balanced bipartite graphs, with the same number of vertices on both sides of the bipartition. For, the adjacency matrix of a directed graph with *n* vertices can be any [(0,1) matrix](https://en.wikipedia.org/wiki/(0,1)_matrix) of size {\displaystyle n\times n}, which can then be reinterpreted as the adjacency matrix of a bipartite graph with *n* vertices on each side of its bipartition.[[23]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-23) In this construction, the bipartite graph is the [bipartite double cover](https://en.wikipedia.org/wiki/Bipartite_double_cover) of the directed graph.

Algorithms[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=7)]

**Testing bipartiteness[**[**edit**](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=8)**]**

It is possible to test whether a graph is bipartite, and to return either a two-coloring (if it is bipartite) or an odd cycle (if it is not) in [linear time](https://en.wikipedia.org/wiki/Linear_time), using [**depth-first search**](https://en.wikipedia.org/wiki/Depth-first_search).

**#1. The idea is to assign to each vertex the color that differs from the color of its parent in the depth-first search forest,** assigning colors in a [preorder traversal](https://en.wikipedia.org/wiki/Preorder_traversal) of the depth-first-search forest. This will necessarily provide a two-coloring of the [spanning forest](https://en.wikipedia.org/wiki/Spanning_forest) consisting of the edges connecting vertices to their parents, but it may not properly color some of the non-forest edges. In a depth-first search forest, one of the two endpoints of every non-forest edge is an ancestor of the other endpoint, and when the depth first search discovers an edge of this type it should check that these two vertices have different colors. If they do not, then the path in the forest from ancestor to descendant, together with the miscolored edge, form an odd cycle, which is returned from the algorithm together with the result that the graph is not bipartite. However, if the algorithm terminates without detecting an odd cycle of this type, then every edge must be properly colored, and the algorithm returns the coloring together with the result that the graph is bipartite.[[24]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-24)

2.  [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search) . Again, each node is given the opposite color to its parent in the search forest. If, when a vertex is colored, there exists an edge connecting it to a previously-colored vertex with the same color, then this edge together with the paths in the breadth-first search forest connecting its two endpoints to their [lowest common ancestor](https://en.wikipedia.org/wiki/Lowest_common_ancestor) forms an odd cycle. If the algorithm terminates without finding an odd cycle, then it must have found a proper coloring, and can safely conclude that the graph is bipartite.[[25]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-25)

For the [intersection graphs](https://en.wikipedia.org/wiki/Intersection_graph) of n [line segments](https://en.wikipedia.org/wiki/Line_segment) or other simple shapes in the [Euclidean plane](https://en.wikipedia.org/wiki/Euclidean_plane), it is possible to test whether the graph is bipartite and return either a two-coloring or an odd cycle in time O(nlogn), even though the graph itself may have upto O(n^{2} edges.[[26]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-26)

**Odd cycle transversal**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=9)]

[](https://en.wikipedia.org/wiki/File:Odd_Cycle_Transversal_of_size_2.png)

A graph with an odd cycle transversal of size 2: removing the two blue bottom vertices leaves a bipartite graph.

[Odd cycle transversal](https://en.wikipedia.org/wiki/Odd_cycle_transversal) is an [NP-complete](https://en.wikipedia.org/wiki/NP-complete) [algorithmic](https://en.wikipedia.org/wiki/Algorithm) problem that asks, given a graph *G* = (*V*,*E*) and a number *k*, whether there exists a set of *k* vertices whose removal from *G* would cause the resulting graph to be bipartite.[[27]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-yannakakis1978node-27) The problem is [fixed-parameter tractable](https://en.wikipedia.org/wiki/Parameterized_complexity), meaning that there is an algorithm whose running time can be bounded by a polynomial function of the size of the graph multiplied by a larger function of *k*.[[28]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-reed2004finding-28) Hence, to delete vertices from a graph in order to obtain a bipartite graph, one needs to "hit all odd cycle", or find a so-called odd cycle [transversal](https://en.wikipedia.org/wiki/Transversal_(combinatorics)) set. In the illustration, every odd cycle in the graph contains the blue (the bottommost) vertices, so removing those vertices kills all odd cycles and leaves a bipartite graph.

The *edge bipartization* problem is the problem of deleting as few edges as possible to make a graph bipartite. This problem is also [fixed-parameter tractable](https://en.wikipedia.org/wiki/Fixed-parameter_tractable), and can be solved in time *O*(2*k* *m*2),[[29]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-guo2006compression-29) where *k* is the number of edges to delete and *m* is the number of edges in the input graph.

**Matching**[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=10)]

A [matching](https://en.wikipedia.org/wiki/Matching_(graph_theory)) in a graph is a subset of its edges, no two of which share an endpoint. [Polynomial time](https://en.wikipedia.org/wiki/Polynomial_time) algorithms are known for many algorithmic problems on matchings, including [maximum matching](https://en.wikipedia.org/wiki/Maximum_matching) (finding a matching that uses as many edges as possible), [maximum weight matching](https://en.wikipedia.org/wiki/Maximum_weight_matching), and [stable marriage](https://en.wikipedia.org/wiki/Stable_marriage).[[30]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-30) In many cases, matching problems are simpler to solve on bipartite graphs than on non-bipartite graphs,[[31]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-31) and many matching algorithms such as the [Hopcroft–Karp algorithm](https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm) for maximum cardinality matching[[32]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-32) work correctly only on bipartite inputs.

As a simple example, suppose that a set {\displaystyle P} of people are all seeking jobs from among a set of {\displaystyle J} jobs, with not all people suitable for all jobs. This situation can be modeled as a bipartite graph {\displaystyle (P,J,E)}where an edge connects each job-seeker with each suitable job.[[33]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-33) A [perfect matching](https://en.wikipedia.org/wiki/Perfect_matching) describes a way of simultaneously satisfying all job-seekers and filling all jobs; [Hall's marriage theorem](https://en.wikipedia.org/wiki/Hall%27s_marriage_theorem) provides a characterization of the bipartite graphs which allow perfect matchings. The [National Resident Matching Program](https://en.wikipedia.org/wiki/National_Resident_Matching_Program) applies graph matching methods to solve this problem for [U.S. medical student](https://en.wikipedia.org/wiki/Medical_education_in_the_United_States) job-seekers and [hospital residency](https://en.wikipedia.org/wiki/Residency_(medicine)) jobs.[[34]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-34)

The [Dulmage–Mendelsohn decomposition](https://en.wikipedia.org/wiki/Dulmage%E2%80%93Mendelsohn_decomposition) is a structural decomposition of bipartite graphs that is useful in finding maximum matchings.[[35]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-35)

Additional applications[[edit](https://en.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&section=11)]

Bipartite graphs are extensively used in modern [coding theory](https://en.wikipedia.org/wiki/Coding_theory), especially to decode [codewords](https://en.wikipedia.org/wiki/Codeword) received from the channel. [Factor graphs](https://en.wikipedia.org/wiki/Factor_graph) and [Tanner graphs](https://en.wikipedia.org/wiki/Tanner_graph) are examples of this. A Tanner graph is a bipartite graph in which the vertices on one side of the bipartition represent digits of a codeword, and the vertices on the other side represent combinations of digits that are expected to sum to zero in a codeword without errors.[[36]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-36) A factor graph is a closely related [belief network](https://en.wikipedia.org/wiki/Belief_network) used for probabilistic decoding of [LDPC](https://en.wikipedia.org/wiki/LDPC) and [turbo codes](https://en.wikipedia.org/wiki/Turbo_codes).[[37]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-37)

In computer science, a [Petri net](https://en.wikipedia.org/wiki/Petri_net) is a mathematical modeling tool used in analysis and simulations of concurrent systems. A system is modeled as a bipartite directed graph with two sets of nodes: A set of "place" nodes that contain resources, and a set of "event" nodes which generate and/or consume resources. There are additional constraints on the nodes and edges that constrain the behavior of the system. Petri nets utilize the properties of bipartite directed graphs and other properties to allow mathematical proofs of the behavior of systems while also allowing easy implementation of simulations of the system.[[38]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-38)

In [projective geometry](https://en.wikipedia.org/wiki/Projective_geometry), [Levi graphs](https://en.wikipedia.org/wiki/Levi_graph) are a form of bipartite graph used to model the incidences between points and lines in a [configuration](https://en.wikipedia.org/wiki/Configuration_(geometry)). Corresponding to the geometric property of points and lines that every two lines meet in at most one point and every two points be connected with a single line, Levi graphs necessarily do not contain any cycles of length four, so their [girth](https://en.wikipedia.org/wiki/Girth_(graph_theory)) must be six or more

### 拓扑排序

偏序=》全序

类似选择排序，每次选度为0的。

引申：计算入度、出度

### 单源路径

**final int**[][] **times**; //两两距离  
 **final int K**; //起点  
 **int N**; //点数  
//arrived中的点是已经确定的，他们的距离不可能再小了。  
 HashSet<Integer> **arrived** = **new** HashSet<>();  
 **public int**[] **distance**; //起点到各点的距离  
 *//经过的索引路径* **public** List<List<Integer>> **path** = **new** ArrayList<>();  
  
 **public** Dijkstra(**int**[][] times, **int** k) {  
 **this**.**times** = times;  
 **K** = k;  
 **N** = times[0].**length**;  
 **distance** = **new int**[**N**];  
 **for** (**int** i = 0; i < **N**; i++) {  
 **distance**[i] = times[**K**][i];  
 **path**.add(**new** ArrayList<>());  
 }  
 **arrived**.add(**K**);  
 }  
  
 **public void** run() {  
 *//每次加入一个点* **for** (**int** i = 0; i < **N**-1; i++) {  
 **int** nearIndex = getNear();  
 **if** (nearIndex == -1) **return**;  
 **arrived**.add(nearIndex);  
 updateDistance(nearIndex);  
 }  
 }  
  
 **private void** updateDistance(**int** bridgeIndex){  
 **for** (**int** i = 0; i < **N**; i++) {  
 **if**(**arrived**.contains(i)) **continue**;  
 **if** (**distance**[i] - **distance**[bridgeIndex]> **times**[bridgeIndex][i]){  
 *//加号越界  
// if (distance[i] > distance[bridgeIndex] + times[bridgeIndex][i]){* **distance**[i] = **distance**[bridgeIndex] + **times**[bridgeIndex][i];  
 List<Integer> thisPath = **path**.get(i);  
 thisPath.add(i);  
 }  
 }  
 }  
//还没确定的点，离K最近的点。在外部循环执行了n次  
 **private int** getNear(){  
 **int** minDistance = Integer.***MAX\_VALUE***;  
 **int** minIndex = -1;  
 **for** (**int** j = 0; j < **N**; j++) {  
 **if** (**arrived**.contains(j)){  
 **continue**;  
 }  
 **if** (minDistance > **distance**[j]){  
 minDistance = **distance**[j];  
 minIndex = j;  
 }  
 }  
 **return** minIndex;  
 }

<https://leetcode.com/problems/cheapest-flights-within-k-stops/description/>

单源路径

743. Network Delay Time

带权单源路径

### 多源路径

1. **多次运行**Dijkstra
2. **Floyd**

**public class** Floyd {  
 **final int**[][] **times**;  
 **final int K**;  
 **int N**;  
  
 **int**[][] **distance**;  
  
 **public** Floyd(**int**[][] times, **int** k) {  
 **this**.**times** = times;  
 **K** = k;  
 **N** = times[0].**length**;  
  
 **distance** = **new int**[**N**][**N**];  
 **for** (**int** i = 0; i < **N**; i++) {  
 **distance**[i] = Arrays.*copyOf*(times[i], **N**);  
 }  
 }  
  
 **public void** run(){  
 **for** (**int** i = 0; i < **N**; i++) {  
 *//借助点i，从j到k的距离可以更新了* **for** (**int** j = 0; j < **N**; j++) {  
 **for** (**int** k = 0; k < **N**; k++) {  
*// if (distance[j][k] > distance[j][i] + distance[i][k]){* **if** (**distance**[j][k] -**distance**[j][i] > **distance**[i][k]){  
 **distance[j][k] = distance[j][i] + distance[i][k];** }  
 }  
 }  
 }  
 }  
}

399. Evaluate Division：除式链接/路径查找

## 并查集

**int**[] **parent**;//双亲表示法  
**int**[] **rank**;//秩，有点层次的意思  
*//每个集合元素个数***int**[] **count**;//确保parent的值是准的  
*//集合总数***int countSet**;  
  
**public** DisjointCountSet(**int** n) {  
 **parent** = **new int**[n];  
 **rank** = **new int**[n];  
 **count** = **new int**[n];  
}  
  
*/\*\*  
 \* 要求本来没有。  
 \** ***@param x*** *\*/***public void** makeSet(**int** x) {  
 **parent**[x] = x;  
 **rank**[x] = 0;  
 **count**[x] = 1;  
 **countSet**++;  
}  
  
**public void** union(**int** x, **int** y) {  
 **if** (findSet(x) == findSet(y)){  
 **return**;  
 }  
 **countSet**--;  
 **int** cx = getCount(x),  
 cy = getCount(y);  
 setCount(x, cx+cy);  
 setCount(y, cx +cy);  
 link(findSet(x), findSet(y));  
}  
  
**private void** link(**int** set, **int** set1) {//较大的秩的点为父节点  
 **if** (**rank**[set] > **rank**[set1]) {  
 **parent**[set1] = set;  
 } **else if** (**rank**[set] < **rank**[set1]) {  
 **parent**[set] = set1;  
 } **else** {  
 **rank**[set1] += 1;  
 **parent**[set] = set1;  
 }  
}  
  
**public int** findSet(**int** x) {  
 **if** (**parent**[x] != x) {//这里是扁平化操作  
 **int** parentIndex = findSet(**parent**[x]);  
 **parent[x] = parentIndex;** }  
 **return parent**[x];  
}  
  
**public int** getCountSet(){  
 **return countSet**;  
}  
//parent的数值才是准的  
**public int** getCount(**int** i){  
 **int** parent = findSet(i);  
 **return count**[parent];  
}  
//只处理parent，孩子太多了，不好处理  
**public void** setCount(**int** i, **int** c){  
 **int** parent = findSet(i);  
 **count**[parent] = c;  
}

**200. Number of Islands**

## 字符串匹配

**输入参数，注意两个字符串为空的情况。**

朴素

Rabin-Karp/RollingHash

自动机

KMP

对每个模式串 P 而言，都有一个相应的一个匹配自动机。如图给出了一个模式 P = ababaca 的自动机构造过程：



为了构造一个自动机，我们应当首先定义一个辅助函数σ，称为相应模式串 P[1..m] 的后缀函数。函数 σ 是一个从 ∑\* 到 {0, 1, 2, ..., m} 上定义的映射。σ(x)表示文本串 x 的后缀的长度，且该后缀是模式串 P 的最长前缀。  
即 σ(x) = max{k : Pk ⊐ x}。例如，对于模式串 P = ab，有 σ(ɛ) = 0, σ(ccaca) = 1, σ(ccab) = 2。对于一个长度为 m 的模式串 P 而言，当且仅当 P ⊐ x 时，σ(x) = m。根据后缀函数的定义有：x ⊐ y，则 σ(x) ≤ σ(y)。

所以，对于给定模式串 P[1..m]，对应字符串匹配自动机定义如下：  
　　状态集 Q = {0, 1, ..., m}，初始状态 q0 = 0，接受状态 A = {m}；  
　　对任意状态 q 和字符 a，变迁函数 δ 定义为：δ(q, a) = σ(Pqa)。

我们之所以有 δ(q, a) = σ(Pqa)，是为了追踪当前已匹配最长的模式串 P 的前缀。考虑当前读取的最后一个字符 T[i]，为了寻找文本串 T 的一个子字符串可以匹配模式串 P 的前缀 Pj，Pj一定是 T[i] 的后缀。  
假设状态 q 是读取 T[i] 后的状态，即 q = φ(T[i])。因为我们有变迁函数δ，所以当其状态 q 能够告诉我们匹配 T[i] 后缀的P的最长前缀的长度，即在状态 q 下，Pq ⊐ Ti 且 q = σ(Ti)。  
所以当 q = m 时，便可以知道匹配成功。因为 φ(Ti) 和 σ(Ti) 都等于 q，所以自动机运行时能够保持如下不变式：φ(Ti) = σ(Ti)。



这个数组的base和代码对不上？仅作示例，帮助理解。下面基于算法导论的实现。

**public class** KMP {  
 **final** String **text**, **pattern**;  
 **int pi**[];  
  
 **public** KMP(String text, String pattern) {  
 **this**.**text** = text;  
 **this**.**pattern** = pattern;  
 **pi** = **new int**[pattern.length()];  
 }  
  
 */\*\*  
 \* 返回匹配的起始位置  
 \** ***@return*** *失败-1  
 \*/* **public int** match(){  
 computePrefix();  
 *//模式和前缀数组的索引，***int k = -1, i = 0;** **for** (; i < **text**.length(); i++) {  
 **while** (k > -1 && **pattern**.charAt(k+1) != **text**.charAt(i)) k = **pi**[k];  
 *//准备试探下一个位置* **if** (**pattern**.charAt(k+1) == **text**.charAt(i)) k++;  
 *//else已经没有退路了，下一轮，主序列指针右移，从0开始匹配。  
 //如果前面升上来了* **if** (k == **pattern**.length()-1) **return** i-k;  
 }  
 **return** -1;  
 }  
  
 **void** computePrefix(){  
 **pi**[0] = -1;  
 *//子串/滑动的串的指针* **int** k = -1;  
 *//文本/父串的指针* **for** (**int** i = 1; i < **pattern**.length(); i++) {  
 *//从开头，k = 0  
 //pi[i] 可以在哪个基础上推进  
 //注意这里试探的是k+1;试验失败，回撤;中间的必然不能匹配，直接回撤到上次记录的地方  
 //一直回撤到0* **while** (k > -1 && **pattern**.charAt(k+1) != **pattern**.charAt(i)) k = **pi**[k];  
 *//在k的基础上推进* **if** (**pattern**.charAt(k+1) == **pattern**.charAt(i)) k ++;  
 *//else否则，只能以0为基础* **pi**[i] = k;  
 }  
 }  
}

## 计算几何

### 1 线段

叉积：p1×p2 = x1y2-x2y1

若叉积>0，p1在p2顺时针方向；若为负，则逆时针方向；若为0，则共线，向量方向相同或相反。

若（p2-p0）×（p1-p0）>0,则p0p2在p0p1顺时针方向，在p1处右转；

若<0,则p0p2在p0p1逆时针方向，在p1处左转；

若=0，则共线

线段p1p2与p3p4相交 ⬄ 互相跨越 或者 端点在另一条线段上

### 2 多条线段中，确定任意一对线段是否相交

每个线段端点都是事件点。

当扫除线遇到线段左端点时，加入比较集合。当遇到右端点时，删除。

每当两条线段首次在比较中变为连续时，就检查是否相交。

ANY-SEGMENTS-INTERSECT(S)

1 T=空集;

2 sort the endpoints of the segments in S from left to right,

breaking ties by putting left endpoints before right endpoints

and breaking further ties by putting points with lower y-coordinates ﬁrst

3 for each point p in the sorted list of endpoints

4 if p is the left endpoint of a segment s

5 INSERT.T; s/

6 if (ABOVE.T; s/ exists and intersects s) or (BELOW.T; s/ exists and intersects s)

7 return TRUE

8 if p is the right endpoint of a segment s

9 if both ABOVE.T; s/ and BELOW.T; s/ exist and ABOVE.T; s/ intersects BELOW.T; s/

10 return TRUE

11 DELETE.T; s/

12 return FALSE

### 3 凸包

1. Graham扫描法nlogn

从最左下点开始，按角度排序。

维护一个栈，每次取两个点，和新来的点作计算转向。

如果新来的点没有向左转，则丢弃栈顶的点。

1. Jarvis步进法nh

最左下点TODO？？

### 4 寻找最近点对

分治：

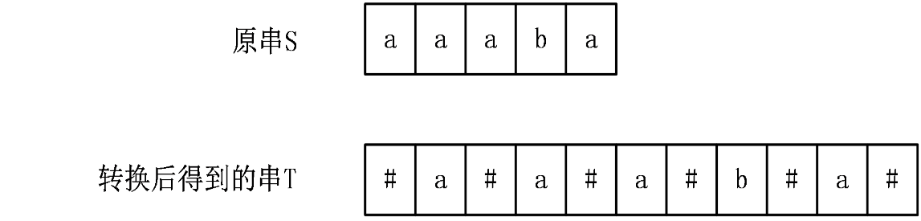
1. 以x分为左右两部分，得到mindistL，minDistR，两者较小的mindist
2. 考虑由两部分接近的点，考虑x-mindist和x+minDist之间的点。
3. 带状点按y坐标排序，只需要考虑一个点和后面7个点的距离即可。（两个正方形内最多有8个点，这里考虑后面7点，戳戳有余，实际可能已经超了。）

## 扩展

### 马拉车算法/计算所有回文子串长度

**1.原理与实现**

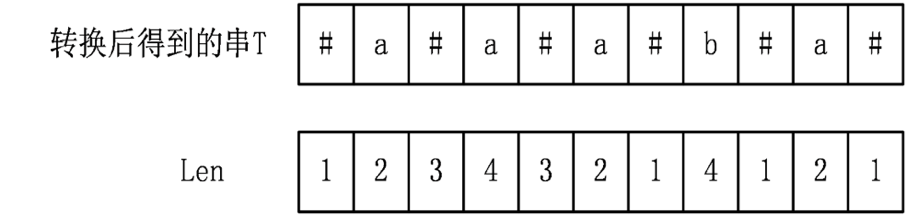
将长度为奇数的回文串和长度为偶数的回文串一起考虑，在原字符串的每个相邻两个字符中间插入一个分隔符，同时在首尾也要添加一个分隔符，分隔符的要求是不在原串中出现，可以用#号：



**（1）Len数组简介与性质**

辅助数组**Len[i]表示以字符T[i]为中心的最长回文字串的最右字符到T[i]的长度**，比如以T[i]为中心的最长回文字串是T[l,r],那么Len[i]=r-i+1。

对于上面的例子，可以得出Len[i]数组为:



Len[i]-1就是该回文子串在原字符串S中的长度，

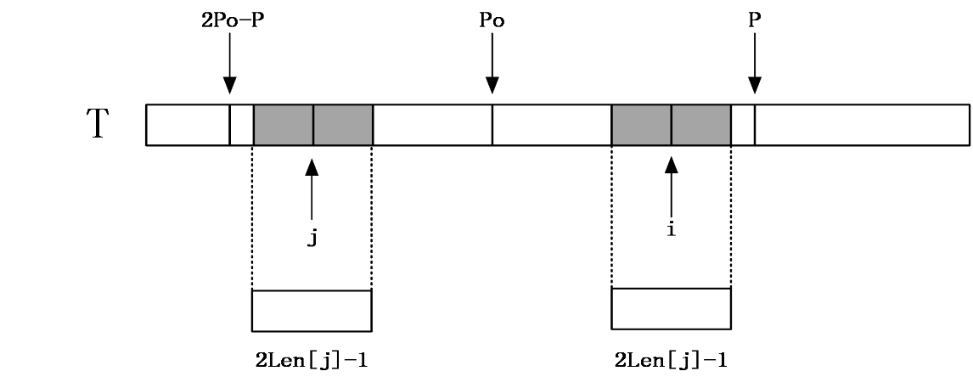
证明，首先在转换得到的字符串T中，所有的回文字串的长度都为奇数，那么对于以T[i]为中心的最长回文字串，其长度就为2\*Len[i]-1,经过观察可知，T中所有的回文子串，其中分隔符的数量一定比其他字符的数量多1，也就是有Len[i]个分隔符，剩下Len[i]-1个字符来自原字符串，所以该回文串在原字符串中的长度就为Len[i]-1。

**（2）Len数组的计算o(n)**

首先从左往右依次计算Len[i]。设P为之前计算中最长回文子串的右端点的最大值，并且设取得这个最大值的位置为po，分两种情况：

第一种情况：i<=P

那么找到i相对于po的对称位置，设为j，那么如果Len[j]<P-i，如下图：



那么说明以j为中心的回文串一定在以po为中心的回文串的内部，且j和i关于位置po对称，由回文串的定义可知，一个回文串反过来还是一个回文串，所以以i为中心的回文串的长度至少和以j为中心的回文串一样，即Len[i]>=Len[j]。因为Len[j]<P-i,所以说i+Len[j]<P。由对称性可知Len[i]=Len[j]。

如果Len[j]>=P-i,由对称性，说明以i为中心的回文串可能会延伸到P之外，而大于P的部分我们还没有进行匹配，所以要从P+1位置开始一个一个进行匹配，直到发生失配，从而更新P和对应的po以及Len[i]。



第二种情况: i>P

如果i比P还要大，说明对于中点为i的回文串还一点都没有匹配，只能一个一个匹配了，匹配完成后要更新P的位置和对应的po以及Len[i]。



**2.时间复杂度分析**

算法只有遇到还没有匹配的位置时才进行匹配，已经匹配过的位置不再匹配，所以对于T字符串中的每一个位置，只进行一次匹配，所以Manacher算法的总体时间复杂度为O(n)，其中n为T字符串的长度，由于T的长度事实上是S的两倍，所以时间复杂度依然是线性的。

**final** String **src**;  
  
**char**[] **extend**;  
**int**[] **halfWidth**;  
//可能存在多个回文子串  
**public** List<String> **longest** = **new** ArrayList<>();  
  
**public** Manacher(String src) {  
 **this**.**src** = src;  
 **extend** = **new char**[2\* src.length()+1];  
 **halfWidth** = **new int**[2\*src.length()+1];  
}  
  
**public void** run(){  
 insertBoundary();  
 computeEveryLength();  
 computeLongest();  
}  
  
**void** computeLongest(){  
 **int** max = 0;  
 List<Integer> maxIndexes = **new** ArrayList<>();  
 **for** (**int** i = 0; i < **src**.length()\*2+1; i++) {   
 **if** (**halfWidth**[i] < max) **continue**;  
 **if** (**halfWidth**[i] == max){  
 maxIndexes.add(i);  
 **continue**;  
 }  
 maxIndexes.clear();  
 maxIndexes.add(i);  
 max = **halfWidth**[i];  
 }  
 maxIndexes.stream().forEach(index->{  
 StringBuilder stringBuilder = **new** StringBuilder();  
 **for** (**int** i = index - **halfWidth**[index] + 1; i < index + **halfWidth**[index]; i++) {  
 **if (extend[i] != '#')** stringBuilder.append(**extend**[i]);  
 }  
 **longest**.add(stringBuilder.toString());  
 });  
}  
  
**void** computeEveryLength(){  
 **halfWidth**[0] = 1;  
 **int** rightEdge = 0, center = 0;  
 **for** (**int** i = 1; i < **src**.length()\*2+1; i++) {//计算以i为中心的字串  
 **int** thisHalfWidth = 0;  
 **if** (i <= rightEdge){  
 *//对称位置* thisHalfWidth = **halfWidth**[center-(i-center)];  
 **if** (i + thisHalfWidth-1 < rightEdge){  
 *//在已探测范围内，必然无法拓展* **halfWidth**[i] = thisHalfWidth;  
 **continue**;  
 }  
 thisHalfWidth = rightEdge - i + 1;  
 }**else**{  
 thisHalfWidth = 1;  
 }  
 *//以i为中心扩展* **while**(i + thisHalfWidth< **src**.length()\*2+1  
 && i - thisHalfWidth >= 0  
 && **extend**[i + thisHalfWidth] == **extend**[i - thisHalfWidth]){  
 thisHalfWidth++;  
 }  
 **halfWidth**[i] = thisHalfWidth;  
 **if** (i + thisHalfWidth-1 > rightEdge){  
 *//探测范围已拓展* rightEdge = i+ thisHalfWidth-1;  
 center = i;  
 }  
 }  
}  
  
**void** insertBoundary(){  
 **for** (**int** i = 0; i < **src**.length(); i++) {  
 **extend**[2\*i] = **'#'**;  
 **extend**[2\*i+1] = **src**.charAt(i);  
 }  
 **extend**[**src**.length() \*2] = **'#'**;  
}

### 格雷码/循环码/最小差错码

生成N位格雷码组

for (int i = 0; i < 1<<n; i++)

result.add(i ^ i>>1);

如何生成：

1 递归

!2进制公式

3 卡诺图

3 异或乘除

## 简单

### 逆波兰表达式

150. Evaluate Reverse Polish Notation

# 从形式到思路

## 几何图形

**149. Max Points on a Line**

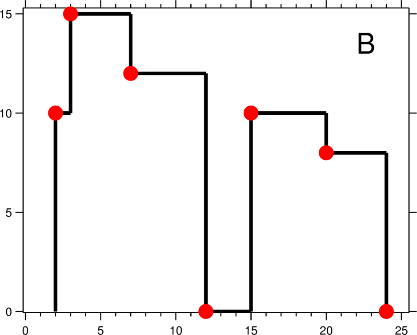
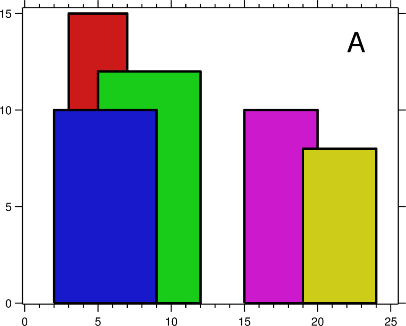
解：循环每个点，其他点与该点的斜率是否相等，如果相等，则在同一直线上

**注意点：**

1. 输入：点数组长度为0，长度为1

2. 实现：斜率正、负无穷；同一个点；斜率无损表示。（另，考虑叉积代替斜率计算）

**218. The Skyline Problem**



解：每个关键点作为事件处理：开始事件，结束事件。根据当前高度和新的关键点计算下一高度。

边界条件：同一个点多个开始事件，多个结束事件等。

**public class** Solution {  
 **class** Event {  
 **static final int *TYPE\_START*** = 0;  
 **static final int *TYPE\_END*** = 1;  
  
 **int start**;  
 **int end**;  
 **int type**;  
 **int height**;  
  
 **public** Event(**int** start, **int** end, **int** type, **int** height) {  
 **this**.**start** = start;  
 **this**.**end** = end;  
 **this**.**type** = type;  
 **this**.**height** = height;  
 }  
  
 */\*\*  
 \* 事件发生的坐标  
 \*  
 \** ***@return*** *\*/* **public int** getHappenPoint() {  
 **if** (**type** == Event.***TYPE\_START***) **return start**;  
 **else**  **return end**;  
 }  
 }  
//每个高度的结束点，livingHeight，存活的高度。动态变化的集合。  
 TreeMap<Integer, Integer> **heightToEnd** = **new** TreeMap<>(**new** Comparator<Integer>() {  
 @Override  
 **public int** compare(Integer o1, Integer o2) {

return o2-o1;  
 }  
 });

//事件队列排在前面的先处理  
 PriorityQueue<Event> **queue** = **new** PriorityQueue<>(**new** Comparator<Event>() {  
 @Override  
 **public int** compare(Event o1, Event o2) {  
 **int** dif = o1.getHappenPoint() - o2.getHappenPoint();  
 **if** (dif == 0) {  
 **if** (o1.**type** == o2.**type**) {  
 *//高度按从高到低排列*

return o2.**height** – o1.**height**;  
 } **else** {  
 *//按常规习惯，end排前面* **if** (o1.**type** == Event.***TYPE\_END***) **return** -1;  
 **if** (o2.**type** == Event.***TYPE\_END***) **return** 1;  
 **throw new** RuntimeException();  
 }  
 } **else** {  
 **return** diff;  
 }  
 }  
 });  
  
 **public** List<**int**[]> getSkyline(**int**[][] buildings) {  
 List<**int**[]> result = **new** ArrayList<>();  
 *//heightToEnd.put(0, Integer.MAX\_VALUE);* **for** (**int** i = 0; i < buildings.**length**; i++) {  
 **int**[] current = buildings[i];  
 **queue**.add(**new** Event(current[0], current[1], Event.***TYPE\_START***, current[2]));  
 **queue**.add(**new** Event(current[0],current[1], Event.***TYPE\_END***, current[2]));  
 }  
 **int** heightCurrent = 0;  
 Event lastDownEvent = **null**;  
 **while** (!**queue**.isEmpty()) {  
 Event event = **queue**.poll();  
 *//System.out.println(event.coordinate);  
 //把同一个事件点的,同类型，事件都取出来。  
 //前面是按高度排序的，第一个是最高的* List<Event> happenHereOther = **new** ArrayList<>();  
 **while**(!**queue**.isEmpty()  
 && **queue**.peek().**type** == event.**type** && **queue**.peek().getHappenPoint() == event.getHappenPoint()){  
 happenHereOther.add(**queue**.poll());  
 }  
  
 **switch** (event.**type**) {  
 **case** Event.***TYPE\_START***://开始点只需要要考虑高度上升  
 *//维持高度表，所有的都加进去，因为不知道哪个点会存续到最后* **int** end = **heightToEnd**.getOrDefault(event.**height**, -1);  
 **if** (end < event.**end**) {  
 *//以前没有，或者以前比较短  
 //续命* **heightToEnd**.put(event.**height**, event.**end**);  
 }  
 **for** (Event eventStart :  
 happenHereOther) {  
 end = **heightToEnd**.getOrDefault(eventStart.**height**, -1);  
 **if** (end < eventStart.**end**) {  
 *//以前没有，或者以前比较短  
 //续命* **heightToEnd**.put(eventStart.**height**, eventStart.**end**);  
 }  
 }  
 *//高度表不影响本次判断  
 //考察是否发生了上升，只需要考虑最高的点。* **if** (event.**height** > heightCurrent) {  
 *//如果上次下降的点，恰好和这次重合* **if** (lastDownEvent!= **null** && lastDownEvent.getHappenPoint() == event.getHappenPoint()){  
 *//移除* result.remove(result.size()-1);  
 heightCurrent = event.**height**;  
 **if**(lastDownEvent.**height** != event.**height**){  
 *//添加本次上升的点* result.add(**new int**[]{event.getHappenPoint(), heightCurrent});  
 }  
 }**else**{  
 *//添加本次上升的点* heightCurrent = event.**height**;  
 result.add(**new int**[]{event.getHappenPoint(), heightCurrent});  
 }  
 }  
 **break**;  
 **case** Event.***TYPE\_END***://结束点只需要考虑高度下降。注意紧贴着的开始点。  
 *//为了获取下一个高度，把当前死亡的高度都移除掉* **for** (Event eventEnd :  
 happenHereOther) {  
 *//该点的高度，可能被续命了，不能移除  
 //不可能为空* **int** endX = **heightToEnd**.getOrDefault(eventEnd.**height**, -1);  
 **if**(endX == eventEnd.getHappenPoint()){  
 *//真的终结了* **heightToEnd**.remove(eventEnd.**height**);  
 }  
 }  
 *//剩下的最高的，就是下一个高度  
  
 //再来考虑天际线有没有下降,只需要处理最高的点，其他点都没有用。* **int** end2 = **heightToEnd**.getOrDefault(event.**height**, -1);  
*//event可能在上面循环处理过了，也可能没处理。-1表示在上面循环移除了（多个同一高度事件同时结束）*

**if** (end2==-1 || end2 == event.getHappenPoint()) {  
 *//真的终结了* **heightToEnd**.remove(event.**height**);  
 **if** (event.**height** == heightCurrent) {  
 *//天际线发生下降  
 //剩余的最高点* **if** (**heightToEnd**.isEmpty()) {  
 heightCurrent = 0;  
 } **else** {  
 heightCurrent = **heightToEnd**.firstKey();  
 }  
 result.add(**new int**[]{event.getHappenPoint(), heightCurrent});  
 lastDownEvent = event;  
 }  
 } **else** {  
 *//还没终结  
 //heightToEnd不用处理  
 //result不用处理* }  
 **break**;  
 }  
  
 }  
  
 **return** result;  
 }  
}

## 计数/概率

形式：在上一次的基础上累加，进行计数。每增加一个数，考虑所有和的情况

**一维**

**494. Target Sum**：n个数，1个和，每个数前面放+号，或者-号，一共有多少种方式。

下面两题本质一模一样

剑 题43 骰子的点数

**377. Combination Sum IV**：Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

***nums*** = [1, 2, 3]

***target*** = 4

The possible combination ways are:

(1, 1, 1, 1)、(1, 1, 2)。。。

**二维**

leetcode 马走日的概率问题

leetcode 将的概率问题

576. Out of Boundary Paths

There is an **m** by **n** grid with a ball. Given the start coordinate **(i,j)** of the ball, you can move the ball to **adjacent** cell or cross the grid boundary in four directions (up, down, left, right). However, you can **at most** move **N** times. Find out the number of paths to move the ball out of grid boundary.

出界

Dp：上一步推下一步，用矩阵记录上一步的值

## 广义表

341. Flatten Nested List Iterator: [[1,1],2,[1,1]]🡪 [1,1,2,1,1]

385. Mini Parser

394. Decode String

借助栈实现

## 数字

正数的补码就是其本身

负数，符号位不变，取反后加一。

### 构建数字

**321. Create Maximum Number**

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits.

高位尽量大=》贪心选取=》分析选取范围

**402. Remove K Digits**：remove *k* digits from the number so that the new number is the smallest possible.

解法一：

删除数字-》保留数字-》高位重要

解法二：

one can simply scan from left to right, and remove the first "peak" digit; the peak digit is larger than its right neighbor. One can repeat this procedure k times. One can simulate the above procedure by using a stack, and obtain a O(n) algorithm. Note, when the result stack (i.e. res) pop a digit, it is equivalent as remove that "peak" digit.

**738. Monotone Increasing Digits**: Given a non-negative integer N, find the largest number that is less than or equal to N with monotone increasing digits.

when the number is 123454321, we could have a candidate of 123449999. It seems like a decent strategy is to take a monotone increasing prefix of N, then decrease the number before the "cliff" (the index where adjacent digits decrease for the first time) if it exists, and replace the rest of the characters with 9s.

例外123444321-》123399999

### 位操作

**剑 题47 加减法交换两个数、位操作交换两个数**

a = a+ b; b =a –b; a = a-b; (b’ = (a+b)-b; a’ = (a+b)-b’)

a = a ^ b; b = a ^ b; a= a^ b; (b’ = (a ^b) ^b; a’ = (a ^ b) ^ b’)

**剑 题10 二进制中1的个数**

**397. Integer Replacement**

If n is even, replace n with n/2.

If n is odd, you can replace n with either n + 1 or n - 1.

What is the minimum number of replacements needed for n to become 1?

解:

If n is even, halve it.

If n=3 or n-1 has less 1's than n+1, decrement n. Otherwise, increment n.

doing bitCount on every iteration is not the best way. It is enough to examine the last two digits to figure out whether incrementing or decrementing will give more 1's.

if a number ends with 01, then certainly decrementing is the way to go.

if it ends with 11, then certainly incrementing is at least as good as decrementing (\*011 -> \*010 / \*100) or even better (if there are three or more 1's).

海明距离

**477. Total Hamming Distance**：数字集合的总海明距离

按位划分成两组，当前位的贡献k\*(n-k).

**421. Maximum XOR of Two Numbers in an Array**

从结果最大值考虑从，整体考虑，进行筛选：

1 最大结果，二进制从高位到低位

2 a^b = c <=> c^b=a

3 对于每一位，对于结果xx1yy, 逆转一下，查找元素。xx1 ^ b = a（b是已知元素，a是待查找元素）, 判断a存在

4 hash

**338. Counting Bits：n个数**

dp 后面的数依赖前面+1

**779. K-th Symbol in Grammar**

On the first row, we write a 0. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10.

Given row N and index K, return the K-th indexed symbol in row N. (The values of K are 1-indexed.) (1 indexed).

画图找关系

**return** Integer.bitCount(K-1) & 1;

***Updates***: (all index discussed below are 0-based)  
***Observation 1***: **N** does not matter as long as "K will be an integer in the range [1, 2^(N-1)]". We can ignore **N**.

***Observation 2***: let f(k) be the value of kth position (0-based), then:  
f(2 \* k) = 0 {if f(k) = 0} or, 1 {if f(k) = 1} => f(2 \* k) = f(k) xor 0  
f(2 \* k + 1) = 0 {if f(k) = 1} or 1 {if f(k) = 0} => f(2 \* k + 1) = f(k) xor 1

***Obervation 3***: if binary string of **k** is used, let **k = 1001010**, then we have:  
f(1001010) = f(100101) ^ 0 = f(10010) ^ 1 ^ 0 = f(1001) ^ 0 ^ 1 ^ 0 =...= f(0) ^ 1 ^ 0 ^ 0 ^1 ^ 0 ^ 1 ^ 0 = 1 ^ 0 ^ 0 ^1 ^ 0 ^ 1 ^ 0  
So, the result is the **xor** operation on all bits of **k**. Since 0 does not change **xor** result, we can ignore all 0s.  
f(1001010) = 1 ^ 1 ^ 1 = (1^1) ^ 1 = 0 ^ 1 = 1  
f(11110011) = 1 ^ 1^ 1 ^ 1 ^ 1 ^1 = (1 ^ 1) ^ (1 ^ 1) ^ (1 ^1) = 0

f(k) = 0 if **k** has **even** number of 1s in binary representation, and f(k) = 1 when **k** has **odd** number of 1s

## 数组/矩阵区间和/积

**303. Range Sum Query - Immutable**

累积和

**304. Range Sum Query 2D – Immutable**

dp/累积和，

## 四则运算

**241. Different Ways to Add Parentheses**

Given a string of numbers and operators, return all possible results

((2-1)-1) = 0

(2-(1-1)) = 2

穷举，分治，备忘录

**399. Evaluate Division**：连环计算

Given a / b = 2.0, b / c = 3.0. queries are: a / c = ?

构造图搜索

**lintcode 849. Basic Calculator III:+-\*/()计算器**

解：一个操作数栈，一个操作符栈。后面的操作符决定了是否计算前面的值，如果前面优先级高则计算；遇到括号则计算。最终，操作符栈内的优先级从低到高。

## 字符/数字集合

字符有时候也可以看做数字

通常用数组来存储，但是没有前后关系，跟索引也没有关系。

### 相同字母异序词Anagram

字母数相等，与位置无关。通常用map来计数。

**微软 lintcode 1169. Permutation in String**：Given two strings s1 and s2, write a function to return true if s2 contains the permutation of s1.

Input:s1 = "ab" s2 = "eidbaooo"

Output:True

Explanation: s2 contains one permutation of s1 ("ba").

### 排列/组合

**357. Count Numbers with Unique Digits：**

Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.

**Example:**  
Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])

**60. Permutation Sequence**：Given *n* and *k*, return the *k*th permutation sequence.

按规律推导，从高位到低位，逐个确定

**46. Permutations**：Given a collection of **distinct** integers, return all possible permutations.

**47. Permutations II**：Given a collection of numbers that might contain duplicates, return all possible unique permutations.

重复的下一个排列

**556. Next Greater Element III**：Given a positive **32-bit** integer **n**, you need to find the smallest **32-bit** integer which has exactly the same digits existing in the integer **n** and is greater in value than n. If no such positive **32-bit** integer exists, you need to return -1.

**77. Combinations**

递归构造。

**752. Open the Lock**

两端bfs

### 背包问题/选或不选

NP问题，只能穷举

集合划分问题

<https://en.wikipedia.org/wiki/Partition_problem>

优化：

1 Dfs 大值在前

2 双向bfs，中间map配对

**416. Partition Equal Subset Sum**

**698. Partition to K Equal Sum Subsets**

不同角度，穷尽搜索

Approach #1: Search by Constructing Subset Sums

A approach is to simulate the k groups (disjoint subsets of nums). For each number in nums, we'll check whether putting it in the i-th group solves the problem. We can check those possibilities by recursively searching.

**Complexity Analysis**

* Time Complexity: O(k^{N-k} k!)*O*(*k*​*N*−*k*​​*k*!), where N*N* is the length of nums, and k*k* is as given. As we skip additional zeroes in groups, naively we will make O(k!)*O*(*k*!) calls to search, then an additional O(k^{N-k})*O*(*k*​*N*−*k*​​)calls after every element of groups is nonzero.
* Space Complexity: O(N)*O*(*N*), the space used by recursive calls to search in our call stack.

Approach #2: Dynamic Programming on Subsets of Input

we investigate methods of exhaustive search, and find target = sum(nums) / k in the same way.

Let used have the i-th bit set if and only if nums[i] has already been used. Our goal is to use nums in some order so that placing them into groups in that order will be valid. search(used, ...) will answer the question: can we partition unused elements of nums[i] appropriately?

This will depend on todo, the sum of the remaining unused elements, not crossing multiples of target within one number. If for example our target is 10, and our elements to be placed in order are [6, 5, 5, 4], this would not work as 6 + 5 "crosses" 10 prematurely.

If we could choose the order, then after placing 5, our unused elements are [4, 5, 6]. Using 6 would make todo go from 15 to 9, which crosses 10 - something unwanted. However, we could use 5 since todo goes from 15 to 10; then later we could use 4 and 6 as those placements do not cross.

It turns out the maximum value that can be chosen so as to not cross a multiple of target, is targ = (todo - 1) % target + 1. This is essentially todo % target, plus target if that would be zero.

Now for each unused number that doesn't cross, we'll search on that state, and we'll return true if any of those searches are true.

Notice that the state todo depends only on the state used, so when memoizing our search, we only need to make lookups by used.

**Complexity Analysis**

* Time Complexity: O(N \* 2^N)*O*(*N*∗2​*N*​​), where N*N* is the length of nums. There are 2^N2​*N*​​ states of used (or state in our bottom-up variant), and each state performs O(N) work searching through nums.
* Space Complexity: O(2^N)*O*(2​*N*​​), the space used by memo (or dp, total in our bottom-up variant).

**473. Matchsticks to Square**

Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

**474. Ones and Zeroes**

For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

Note:

1. The given numbers of 0s and 1s will both not exceed 100
2. The size of given string array won't exceed 600.

Example 1:

Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3

Output: 4

Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

解：dp[i][j] = Math.max(1 + dp[i-**count**[0]][j-**count**[1]], dp[i][j]);i表示1个数，j表示0的个数

**39. Combination Sum TODO**

选用多次

**40. Combination Sum II**

集合中有重复元素，选用一次

**216. Combination Sum III**

元素不可重复

**805. Split Array With Same Average**：分成两组，平均值相等

双向搜索，中间用map找到对应点

Time Complexity: O(2^{N/2}).

Space Complexity: O(2^{N/2})

本题技巧：将平均数转化为0，方便处理

## 字符串

字符间有位置关系

### 字符串匹配

**187. Repeated DNA Sequences**

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

Rollinghash把字符串映射成数字，再比较 TODO 怎么样映射比较好？

### 两个字符串的关系

总结：两个字符串关系，可以用一维DP，也可以用二维DP

**801. Minimum Swaps To Make Sequences Increasing**

We have two integer sequences A and B of the same non-zero length.

We are allowed to swap elements A[i] and B[i]. Note that both elements are in the same index position in their respective sequences.

At the end of some number of swaps, A and B are both strictly increasing. (A sequence is strictly increasing if and only if A[0] < A[1] < A[2] < ... < A[A.length - 1].)

Given A and B, return the minimum number of swaps to make both sequences strictly increasing.

**Intuition**

The cost of making both sequences increasing up to the first i columns can be expressed in terms of the cost of making both sequences increasing up to the first i-1 columns. This is because the only thing that matters to the ith column is whether the previous column was swapped or not.

**Let's remember n1 (natural1), the cost of making the first i-1 columns increasing and not swapping the i-1th column; and s1 (swapped1), the cost of making the first i-1 columns increasing and swapping the i-1th column.**

Now we want candidates n2 (and s2), the costs of making the first i columns increasing if we do not swap (or swap, respectively) the ith column.

Algorithm

For convenience, say a1 = A[i-1], b1 = B[i-1] and a2 = A[i], b2 = B[i].

Now, if a1 < a2 and b1 < b2, then it is allowed to have both of these columns natural (unswapped), or both of these columns swapped. This possibility leads to n2 = min(n2, n1) and s2 = min(s2, s1 + 1).

Another, (not exclusive) possibility is that a1 < b2 and b1 < a2. This means that it is allowed to have exactly one of these columns swapped. This possibility leads to n2 = min(n2, s1) or s2 = min(s2, n1 + 1).

Note that it is important to use two if statements separately, because both of the above possibilities might be possible.

At the end, the optimal solution must leave the last column either natural or swapped, so we take the minimum number of swaps between the two possibilities.

**467. Unique Substrings in Wraparound String**

Consider the string s to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so s will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string p. Your job is to find out how many **unique non-empty substrings of p** are present in s. In particular, your input is the string p and you need to output the number of different non-empty substrings of p in the string s.

解： if we know the max number of unique substrings in p ends with 'a', 'b', ..., 'z', then the sum of them is the answer.

The max number of unique substring ends with a letter equals to the length of max contiguous substring ends with that letter. Example "abcd", the max number of unique substring ends with 'd' is 4, apparently they are "abcd", "bcd", "cd" and "d".

If there are overlapping, we only need to consider the longest one because it covers all the possible substrings. Example: "abcdbcd", the max number of unique substring ends with 'd' is 4 and all substrings formed by the 2nd "bcd" part are covered in the 4 substrings already.

No matter how long is a contiguous substring in p, it is in s since s has infinite length.

**583. Delete Operation for Two Strings**: Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.

解1：最长公共子序列：比较第一个字母；递归+备忘=》dp

**712. Minimum ASCII Delete Sum for Two Strings:** find the lowest ASCII sum of deleted characters to make two strings equal.

Let dp[i][j] be the answer to the problem for the strings s1[i:], s2[j:].

When one of the input strings is empty, the answer is the ASCII-sum of the other string. We can calculate this cumulatively using code like dp[i][s2.length()] = dp[i+1][s2.length()] + s1.codePointAt(i).

When s1[i] == s2[j], we have dp[i][j] = dp[i+1][j+1] as we can ignore these two characters.

When s1[i] != s2[j], we will have to delete at least one of them. We'll have dp[i][j] as the minimum of the answers after both deletion options.

Complexity Analysis

Time Complexity: O(M∗N), where M,N are the lengths of the given strings. We use nested for loops: each loop is O(M) and O(N) respectively.

Space Complexity: O(M∗N), the space used by dp.

**44. Wildcard Matching**

The most confusing part is how to deal with '\*'. At first I couldn't figure out why the condition would be (dp[i-1][j] == true || dp[i][j-1] == true).

dp[i][j]: true if the first i char in String s matches the first j chars in String p

Base case:

origin: dp[0][0]: they do match, so dp[0][0] = true

first row: dp[0][j]: except for String p starts with \*, otherwise all false

first col: dp[i][0]: can't match when p is empty. All false.

Recursion:

Iterate through every dp[i][j]

dp[i][j] = true:

if (s[ith] == p[jth] || p[jth] == '?') && dp[i-1][j-1] == true

elif p[jth] == '\*' && (dp[i-1][j] == true || dp[i][j-1] == true)

-for dp[i-1][j], means that \* acts like an empty sequence.

eg: ab, ab\*

-for dp[i][j-1], means that \* acts like any sequences

eg: abcd, ab\*

Start from 0 to len

Output put should be dp[s.len][p.len], referring to the whole s matches the whole p

Be careful about the difference of index i,j in String (0 to len-1) and the index i, j in dp (0 to len)!

Thanks for writing the explanation. I have same question about how to handle the case of \*. You are right in (dp[i-1][j] || dp[i][j-1]) for the \* case, however the explanation provided is slightly incorrect:

dp[i-1][j] means that the \* matches a 1 character (most recent) in S and we can continue to use \* to match more previous characters

dp[i][j-1] means that the \* matches no character in S

Take Example:

S: xxx

P: xx\*

For i=2, j=2:

dp[1][2] means we use the \* to match s[2], can we match the remaining sequence in S if we reuse \* in P?

dp[2][1] means if we don't use \* to match any characters at all, our s[2] would have to match p[1].

IE: the characters before the \* in P would have to match characters in S so far. Meaning p[i][j-1].

### 拾遗

**722. Remove Comments**

line comments, and block comments.

注意嵌套情况

“”字符串嵌套，转义字符

//可以在一行的任意位置

/\*/不完整(*/\*/\*/是合法的*)

**3. Longest Substring Without Repeating Characters**：Given a string, find the length of the **longest substring** without repeating characters.

解：**We use HashSet to store the characters in current window** [i, j] (*j*=*i* initially). Then we slide the index *j* to the right. If it is not in the HashSet, we slide *j* further. Doing so until s[j] is already in the HashSet. At this point, we found the maximum size of substrings without duplicate characters start with index *i*.

**If we do this for all *i*, we get our answer.**

**剑 题35 第一个只出现一次的字符**

扫描两遍，第一遍用map计数

## 回文

回文利用位置对称关系

1. 中心拓展
2. 马拉车
3. 前缀后缀关系

**子串**

**647. Palindromic Substrings**：一个字符串有多少个回文子串

马拉车算法

**131. Palindrome Partitioning**

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

找到第一个回文，然后递归余下的

1**32. Palindrome Partitioning II**：Return the minimum cuts needed for a palindrome partitioning of *s*.

递归+备忘=dp TODO

**子序列（中漏）**

**516. Longest Palindromic Subsequence**

dp, 是否要左右端点的字母

dp[i][j]: the longest palindromic subsequence's length of substring(i, j)

State transition:

if s.charAt(i) == s.charAt(j), dp[i][j] = dp[i+1][j-1] + 2

otherwise, dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1])

Initialization: dp[i][i] = 1

**214. Shortest Palindrome**：Given a string s, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome.

Input: "abcd"

Output: "dcbabcd"

解：转化为前缀后缀关系：we reserved the original string *s* as rev. We iterate over *i* from 0 to *n*−1 and check for s[0:n-i] == rev[i:]. Pondering over this statement, had the rev been concatenated to *s*, this statement is just finding the longest prefix that is equal to the suffix.

正向的前缀 = 反向的后缀，剩余的就是需要添加的字母。

* We use the KMP lookup table generation
* Create new\_s as s+"#"+reverse(s) and use the string in the lookup-generation algorithm
  + The "#" is required, since without the #, the 2 strings could mix with each other For example, take the string "aaaa". Had we not inserted "#" in the middle, the new string would be "aaaaaaaa" and the largest prefix size would be 7 corresponding to "aaaaaaa" which would be obviously wrong.
* Return reversed string after the largest palindrome from beginning length(given by n−f[n\_new-1] + original string s*s*

*//TODO*

**336. Palindrome Pairs**

Given a list of **unique** words, find all pairs of ***distinct*** indices (i, j), so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome.

**Example 2:**  
Given words = ["abcd", "dcba", "lls", "s", "sssll"]  
Return [[0, 1], [1, 0], [3, 2], [2, 4]]  
The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]

O(nk^2):字典树/map

## 单词列表

**809. Expressive Words**

Sometimes people repeat letters to represent extra feeling, such as "hello" -> "heeellooo", "hi" -> "hiiii".  Here, we have groups, of adjacent letters that are all the same character, and adjacent characters to the group are different.  A group is extended if that group is length 3 or more, so "e" and "o" would be extended in the first example, and "i" would be extended in the second example.  As another example, the groups of "abbcccaaaa" would be "a", "bb", "ccc", and "aaaa"; and "ccc" and "aaaa" are the extended groups of that string.

For some given string S, a query word is *stretchy* if it can be made to be equal to S by extending some groups. Formally, we are allowed to repeatedly choose a group (as defined above) of characters c, and add some number of the same character c to it so that the length of the group is 3 or more.  Note that we cannot extend a group of size one like "h" to a group of size two like "hh" - all extensions must leave the group extended - ie., at least 3 characters long.

Given a list of query words, return the number of words that are stretchy.

**Example:**

**Input:**

S = "heeellooo"

words = ["hello", "hi", "helo"]

**Output:** 1

**Explanation:**

We can extend "e" and "o" in the word "hello" to get "heeellooo".

We can't extend "helo" to get "heeellooo" because the group "ll" is not extended.

解：For some word, write the head character of every group, and the count of that group. For example, for "abbcccddddaaaaa", we'll write the "key" of "abcda", and the "count" [1,2,3,4,5].

Let's see if a word is stretchy. Evidently, it needs to have the same key as S.

Now, let's say we have individual counts c1 = S.count[i] and c2 = word.count[i].

* If c1 < c2, then we can't make the ith group of word equal to the ith word of S by adding characters.
* If c1 >= 3, then we can add letters to the ith group of word to match the ith group of S, as the latter is *extended*.
* Else, if c1 < 3, then we must have c2 == c1 for the ith groups of word and S to match.

**524. Longest Word in Dictionary through Deleting**：Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string.

解：简单：逐个比较是否是子序列

**720. Longest Word in Dictionary**

Given a list of strings words, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order.

**Example 1:**

**Input:**

words = ["w","wo","wor","worl", "world"]

**Output:** "world"

**Explanation:**

The word "world" can be built one character at a time by "w", "wo", "wor", and "worl".

解：字典树

**318. Maximum Product of Word Lengths**：two not share common letters

一个整数标记一个单词的字母含量。位操作快一点。

**126. Word Ladder II**

单词变种最短路径

双向bfs

注意，没有答案的，树的高度只要一半就可以了。

新生的节点，不要重复已有的节点。但是同一层的可以重复。

变种后集合的hash查找，比逐个单词比较是否变种，要快很多。

**676. Implement Magic Dictionary**

For the method search, you'll be given a word, and judge whether if you modify exactly one character into another character in this word, the modified word is in the dictionary you just built.

字典树

**521. Longest Uncommon Subsequence I**

解：较长的串；如果长度相等，则任意一个；若完全相等，则没有

**522. Longest Uncommon Subsequence II**

O(n^2)找最长的几个分析。

Sort the strings in the reverse order. If there is not duplicates in the array, then the longest string is the answer.

if there are duplicates, and if the longest string is not the answer, then we need to check other strings.

But the smaller strings can be subsequence of the bigger strings. we need to check if the string is a subsequence of all the strings bigger than itself. If it's not, that is the answer.

## 数组

### 递推关系

**368. Largest Divisible Subset**：Given a set of **distinct** positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.

O(n^2):排序后，dp，每增加一个数，在前面的基础上，找到最长的。

### 前K

**373. Find K Pairs with Smallest Sums**

You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k.

Define a pair (u,v) which consists of one element from the first array and one element from the second array.

Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the **smallest sums**.

简单，记录数组第一个元素，匹配每一个。然后推进。

**215. Kth Largest Element in an Array**

快排

**692. Top K Frequent Words**

简单，堆排序

### 波形

**496. Next Greater Element I**

You are given two arrays **(without duplicates)** nums1 and nums2 where nums1’s elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.

**Example 1:**

**Input:** **nums1** = [4,1,2], **nums2** = [1,3,4,2].

**Output:** [-1,3,-1]

**Explanation:**

For number 4 in the first array, you cannot find the next greater number for it in the second array, so output -1.

For number 1 in the first array, the next greater number for it in the second array is 3.

For number 2 in the first array, there is no next greater number for it in the second array, so output -1.

**503. Next Greater Element II**:循环数组

O(n):从后往前推算大一点元素，右往左下坡入栈，上坡出栈。最后一个往前，跑1-2遍。

**376. Wiggle Subsequence**：A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence.

解一：贪心：波形简化

解二：记录最后一个点作为波峰、波谷的最有值，再根据前面所有推导下一个点。<https://leetcode.com/problems/wiggle-subsequence/solution/> TODO

**股票系列**

**309. Best Time to Buy and Sell Stock with Cooldown**

Input: [1,2,3,0,2]

Output: 3

Explanation: transactions = [buy, sell, cooldown, buy, sell]

状态机 + dp

**714. Best Time to Buy and Sell Stock with Transaction Fee**

Input: prices = [1, 3, 2, 8, 4, 9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

Buying at prices[0] = 1

Selling at prices[3] = 8

Buying at prices[4] = 4

Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

计算时，考虑fee即可

**456. 132 Pattern**：checks whether there is a 132 pattern in the list：不一定连续

区间中的值

Input: [3, 1, 4, 2]

Output: True

Explanation: There is a 132 pattern in the sequence: [1, 4, 2].

解一：O(n^2):对于每一个中间值，向左找到最小的值，向右找区间内的值

解二：O(n^2):波形

### 划分

**813. Largest Sum of Averages**：We partition a row of numbers A into at most K adjacent (non-empty) groups, then our score is the sum of the average of each group. What is the largest score we can achieve?

1. 回溯 + memo

2. 二维dp

dp(n,k) = max (dp(x, k-1) + x到n的平均数), x从1到n-1

### 子序列长度

一个字符串有n!个子序列

s是不是t的子序列

public boolean isSubsequence(String s, String t) {

if (s.length() == 0) return true;

int indexS = 0, indexT = 0;

while (indexT < t.length()) {

if (t.charAt(indexT) == s.charAt(indexS)) {

indexS++;

if (indexS == s.length()) return true;

}

indexT++;

}

return false;

}

**491. Increasing Subsequences**：find all the different possible increasing subsequences of the given array

Input: [4, 6, 7, 7]

Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]

Dfs？？？

**673. Number of Longest Increasing Subsequence**

**最长递增子序列LIS**

解1：n^2: 记录每个数结尾的长度，每次增加一个数，判断是否可以在前面的基础上+1；

解2：nlog2: 记录每个长度结尾的数的值，每次增加一个数，判断长度是否可以增加，判断同长度的结尾值是否可以变小。搜索的时候采用二分搜索。

长度1=》结尾值3

长度2=》结尾值4

。。。

### 丑数

质因子只含235的数

263. Ugly Number: check whether a given number is an ugly number.

解：不断除235

264. Ugly Number II

三个倍数队列包含了所有的丑数，然后归并排序。

313. Super Ugly Number

同上

### 排序数组查找

二分查找变种

### 逆序

775. Global and Local Inversions

O(n): find in range 0 to i-2, see if there is a element larger than A[i]

**public** boolean isIdealPermutation(int[] A) {

int cmax = 0;

**for** (int i = 0; i < A.size() - 2; ++i) {

cmax = Math.max(cmax, A[i]);

**if** (cmax > A[i + 2]) **return** **false**;

}

**return** **true**;

}

**剑 题36 逆序个数**

归并排序

### 两个数的关系

2/3/4 Sum （smaller、closet）

暂定一个值，排序查找另一个值

注意平均值关系，运用两个指针单向滑动

### 小区间

思路：计数排序、二分查找

例：

825. Friends Of Appropriate Ages

### 区间和/积

累积和：累积和特点：对于单一元素，单调递增。

209. Minimum Size Subarray Sum:find the minimal length of a contiguous subarray of which the sum ≥ s

O(nlogn) 累计和，递增=》二分搜索，分割元素查找=》扣除固定值=》在和上扣除

O(n)窗口滑动

**713. Subarray Product Less Than K**

log(∏​*i*​​*x*​*i*​​)=∑​*i*​​log*x*​*i*​​，转化同上

明显：列举所有：O(n3)

累积和：O(n2)

累积和：注意脑中有一个递增序列：二分查找(nlogn)

两个指针：某一个index，有条件提前结束查找：（n）

**238. Product of Array Except Self**

空间利用，利用输出数组

Dp空间的优化

**523. Continuous Subarray Sum:** Given a list of non-negative numbers and a target integer k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to **the multiple of k**, that is, sums up to n\*k where n is also an integer.

We iterate through the input array exactly once, keeping track of the running sum mod k of the elements in the process. If we find that a running sum value at index j has been previously seen before in some earlier index i in the array, then we know that the sub-array (i,j] contains a desired sum.

### 利用索引

数组实际上是一个map 可以在原数组操作

**380. Insert Delete GetRandom O(1)**

位置关系

**769. Max Chunks To Make Sorted**：Given an array arr that is a permutation of [0, 1, ..., arr.length - 1],分组排序后，全数组有序。如何分最多的组？

O(n)：利用数值和索引的关系

**768. Max Chunks To Make Sorted II**：Given an array arr of integers (**not necessarily distinct**), we split the array into some number of "chunks" (partitions), and individually sort each chunk.  After concatenating them, the result equals the sorted array.What is the most number of chunks we could have made?

解：O(nlogn):排序后，建立值和索引关系，转化成上一个问题。

O(n): 索引无关：Use two arrays to store the left max and right min: Iterate through the array, each time all elements to the left are smaller (or equal) to all elements to the right, there is a new chunck.

**565. Array Nesting**：A zero-indexed array A of length N contains all integers from 0 to N-1. Find and return the longest length of set S, where S[i] = {A[i], A[A[i]], A[A[A[i]]], ... } subjected to the rule below.

O(n)：构成链表，查找最长环.记录已访问。利用原数组，可以减少空间。

442. Find All Duplicates in an Array: Given an array of integers, 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.

O(n)：用Map记录已访问数据-》使用原数组减少空间-》映射空间大的话，最终可以恢复数组

**274. H-Index**：Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

定义：A scientist has index h if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each

Input: citations = [3,0,6,1,5]

Output: 3

Explanation: [3,0,6,1,5] means the researcher has 5 papers in total and each of them had

received 3, 0, 6, 1, 5 citations respectively.

Since the researcher has 3 papers with at least 3 citations each and the remaining

two with no more than 3 citations each, her h-index is 3.

O(n)：总文章数可能比较小，计数排序，利用索引，从后往前计算累计和。

**275. H-Index II**：Given an array of citations in ascending order

Input: citations = [0,1,3,5,6]

Output: 3

Explanation: [0,1,3,5,6] means the researcher has 5 papers in total and each of them had

received 0, 1, 3, 5, 6 citations respectively.

Since the researcher has 3 papers with at least 3 citations each and the remaining

two with no more than 3 citations each, her h-index is 3.

O(nlogn)二分搜索

**526. Beautiful Arrangement**

Suppose you have **N** integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these **N** numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

1. nums[i]%i == 0
2. i%nums[i] == 0.

Now given N, how many beautiful arrangements can you construct?

O(n!):暴力列举所有验证

O(k): k refers to the number of valid permutations:在构建过程中验证，提前剪支。

We try to create all the permutations of numbers from 1 to N. We try to place each of the numbers at each position.

We can fix one number at a particular position and check for the divisibility criteria of that number at the particular position. But, we need to keep a track of the numbers which have already been considered earlier so that they aren't reconsidered while generating the permutations.

If the current number doesn't satisfy the divisibility criteria, we can leave all the permutations that can be generated with that number at the particular position. This helps to prune the search space of the permutations to a great extent.

We make use of a visited array of size N. visited[i] refers to the ith number being already placed/not placed in the array being formed till now(True indicates that the number has already been placed).

We make use of a calculate function, which puts all the numbers pending numbers from 1 to N(i.e. not placed till now in the array), indicated by a False at the corresponding visited[i] position, and tries to create all the permutations with those numbers starting from the pos index onwards in the current array. While putting the pos th number, we check whether the I th number satisfies the divisibility criteria on the go i.e. we continue forward with creating the permutations with the number i at the pos th position only if the number i and pos satisfy the given criteria. Otherwise, we continue with putting the next numbers at the same position and keep on generating the permutations.

**667. Beautiful Arrangement II**

Given two integers n and k, you need to construct a list which contains n different positive integers ranging from 1 to n and obeys the following requirement:   
Suppose this list is [a1, a2, a3, ... , an], then the list [|a1 - a2|, |a2 - a3|, |a3 - a4|, ... , |an-1 - an|] has exactly k distinct integers.

O (n):贪心构建：

When \text{k = n-1}k = n-1, a valid construction is \text{[1, n, 2, n-1, 3, n-2, ....]}[1, n, 2, n-1, 3, n-2, ....]. One way to see this is, we need to have a difference of \text{n-1}n-1, which means we need \text{1}1 and \text{n}n adjacent; then, we need a difference of \text{n-2}n-2, etc.

Also, when \text{k = 1}k = 1, a valid construction is \text{[1, 2, 3, ..., n]}[1, 2, 3, ..., n]. So we have a construction when \text{n-k}n-k is tiny, and when it is large. This leads to the idea that we can stitch together these two constructions: we can put \text{[1, 2, ..., n-k-1]}[1, 2, ..., n-k-1]first so that \text{n}n is effectively \text{k+1}k+1, and then finish the construction with the first \text{"k = n-1"}"k = n-1" method.

For example, when n = 6 and k = 3 , we will construct the array as [1, 2, 3, 6, 4, 5]. This consists of two parts: a construction of [1, 2] and a construction of [1, 4, 2, 3] where every element had 2 added to it (i.e. [3, 6, 4, 5]).

### 环/并查集

并查集问题也可以用dfs/bfs来解决。

565. Array Nesting

A zero-indexed array A of length N contains all integers from 0 to N-1. Find and return the longest length of set S, where S[i] = {A[i], A[A[i]], A[A[A[i]]], ... } subjected to the rule below.

Suppose the first element in S starts with the selection of element A[i] of index = i, the next element in S should be A[A[i]], and then A[A[A[i]]]… By that analogy, we stop adding right before a duplicate element occurs in S.

Example 1:

Input: A = [5,4,0,3,1,6,2]

Output: 4

Explanation:

A[0] = 5, A[1] = 4, A[2] = 0, A[3] = 3, A[4] = 1, A[5] = 6, A[6] = 2.

One of the longest S[K]:

S[0] = {A[0], A[5], A[6], A[2]} = {5, 6, 2, 0}

**547. Friend Circles**

统计不同集个数

### 拾遗

**剑 题45 约瑟夫环：**n个人，不断去除第m个

问题描述：n个人（编号0~(n-1))，从0开始报数，报到(m-1)的退出，剩下的人继续从0开始报数。求胜利者的编号。

第一个人(编号m%n-1) 出列之后，剩下的n-1个人组成了一个新的约瑟夫环（以编号为k=m%n的人开始）:

k k+1 k+2 ... n-2, n-1, 0, 1, 2, ... k-2并且从k开始报0。

做一下转换：

k --> 0

k+1 --> 1

k+2 --> 2

...

...

k-2 --> n-2

k-1 --> n-1

变换后就完完全全成为了(n-1)个人报数的子问题，假如知道这个子问题的解：例如x是最终的胜利者，那么根据上面这个表把这个x变回去不刚好就是n个人情况的解！变回去的公式：x'=(x+k)%n



**lintcode 843. 数字翻转**

给你一个01构成的数组。请你找出最小翻转(1变0，0变1)步数，使得数组满足以下规则：1的后面可以是1或者0，但是0的后面必须是0。

样例

给出 array = [1,0,0,1,1,1] , 返回2。

解释：

把两个0翻转成1。

给出 array = [1,0,1,0,1,0] , 返回2。

解释：

把第二个1和第三个1都翻转成0。

**解**：对于长度为n的数组，只有n+1种反转后的状态，对于样例就是：

0，0，0，0，0，0

1，0，0，0，0，0

1，1，0，0，0，0

…..

1，1，1，1，1，1

因此可以先假设全翻转为0，记录1的个数t（需要1->0翻转的次数），然后从头开始遍历，使第i位之前都是1，第i位之后都是0。

具体操作就是遇到array[i]=1时t-1（不需要翻转，但上面已经翻转过了，再减回去），遇到array[i]=0时t+1（多一次翻转），每次操作后更新t的最小值。

220. Contains Duplicate III

find out whether there are two distinct indices i and j in the array such that the absolute difference between nums[i] and nums[j] is at most t and the absolute difference between i and j is at most k.

窗口，TreeSet(红黑树)，桶

注意：比较时相减越界。

桶：https://leetcode.com/problems/contains-duplicate-iii/discuss/61645/AC-O(N)-solution-in-Java-using-buckets-with-explanation

777. Swap Adjacent in LR String

去掉X，比较LR

396. Rotate Function

Given an array of integers A and let *n* to be its length.

Assume Bk to be an array obtained by rotating the array A *k* positions clock-wise, we define a "rotation function" F on A as follow:

F(k) = 0 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1].

Calculate the maximum value of F(0), F(1), ..., F(n-1).

dp:数学推导：f(k) = f(k-1) + sum – nBk[0]：通项公式=》递推公式

## 矩阵

解题思路：回溯、dfs、bfs、dp

路径搜索，一般只依赖邻居，通常可以用dp解决

最简单的空间占用是n2

有些可以优化到n，如2向路径。跟依赖多少方向有关系。

有些可以优化到1

417. Pacific Atlantic Water Flow：中间高两边低，水往两边流

解：

1. Two Queue and add all the Pacific border to one queue; Atlantic border to another queue.
2. Keep a visited matrix for each queue. In the end, add the cell visited by two queue to the result.  
   BFS: Water flood from ocean to the cell. Since water can only flow from high/equal cell to low cell, add the neighboor cell with height larger or equal to current cell to the queue and mark as visited.

### 矩阵2向路径

**62 Unique Paths**：左上到右下有多少种路径排列组合问题哦：DRRRDRRR（m\*n）!/(m!\*n!)

**63. Unique Paths II**：Now consider if some obstacles are added to the grids. How many unique paths would there be?

DP

**174. Dungeon Game:** 向右下移动，血量不能低于1.

|  |  |  |
| --- | --- | --- |
| -2(K) | -3 | 3 |
| -5 | -10 | 1 |
| 10 | 30 | -5(P) |

解：从右下开始dp，dp[row-1][column-1] = min(1, -dungeon[row-1][column-1]+1)。dp[i][j] 由dp[i+1][j]、dp[i][j+1]其中一个决定；该点的最小值要保证该点的血量至少为1，下一点的血量至少为要求的dp值。

### 矩阵4向路径

矩阵邻居搜索

79. Word Search：Given a 2D board and a word, find if the word exists in the grid.

回溯

212. Word Search II：Given a 2D board and a list of words from the dictionary, find all words in the board.

回溯+字典树

542. 01 Matrix：Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.

简单dp,bfs

529. Minesweeper

解：搜索：Search rules:

1. If click on a mine ('M'), mark it as 'X', stop further search.
2. If click on an empty cell ('E'), depends on how many surrounding mine:  
   2.1 Has surrounding mine(s), mark it with number of surrounding mine(s), stop further search.  
   2.2 No surrounding mine, mark it as 'B', continue search its 8 neighbors.

### 矩阵搜索

74. Search a 2D Matrix

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

二分搜索，mid的计算比较复杂

240. Search a 2D Matrix II

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

判断右上角，每次排除一行/列

### 三角形

120. Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

[

[**2**],

[**3**,4],

[6,**5**,7],

[4,**1**,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., **2** + **3** + **5** + **1** = 11).

坐标变换、dp

756. Pyramid Transition Matrix

We are stacking blocks to form a pyramid. Each block has a color which is a one letter string, like `'Z'`.

For every block of color `C` we place not in the bottom row, we are placing it on top of a left block of color `A` and right block of color `B`. We are allowed to place the block there only if `(A, B, C)` is an allowed triple.

We start with a bottom row of bottom, represented as a single string. We also start with a list of allowed triples allowed.

Return true if we can build the pyramid all the way to the top, otherwise false.

**Example 1:**

**Input:** bottom = "XYZ", allowed = ["XYD", "YZE", "DEA", "FFF"]

**Output:** true

**Explanation:**

We can stack the pyramid like this:

A

/ \

D E

/ \ / \

X Y Z

This works because ('X', 'Y', 'D'), ('Y', 'Z', 'E'), and ('D', 'E', 'A') are allowed triples.

一行一行处理，逐渐往上堆

字符可以看作数字

### 区域面积

拆分维度，分别dp，再综合计算

221. Maximal Square：Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Dp

764. Largest Plus Sign：最大十字架

If we knew the longest possible arm length L\_u, L\_l, L\_d, L\_rin each direction from a center, we could know the order min(L\_u, L\_l, L\_d, L\_r) of a plus sign at that center. We could find these lengths separately using dynamic programming.

### 拾遗

789. Escape The Ghosts

a simplified Pacman game. You start at the point (0, 0), and your destination is (target[0], target[1]). There are several ghosts on the map, the i-th ghost starts at (ghosts[i][0], ghosts[i][1]).

Each turn, you and all ghosts simultaneously \*may\* move in one of 4 cardinal directions: north, east, west, or south, going from the previous point to a new point 1 unit of distance away.

You escape if and only if you can reach the target before any ghost reaches you (for any given moves the ghosts may take.)  If you reach any square (including the target) at the same time as a ghost, it doesn't count as an escape.

Return True if and only if it is possible to escape.

**Example 1:**

**Input:**

ghosts = [[1, 0], [0, 3]]

target = [0, 1]

**Output:** true

**Explanation:**

You can directly reach the destination (0, 1) at time 1, while the ghosts located at (1, 0) or (0, 3) have no way to catch up with you.

**Example 2:**

**Input:**

ghosts = [[1, 0]]

target = [2, 0]

**Output:** false

**Explanation:**

You need to reach the destination (2, 0), but the ghost at (1, 0) lies between you and the destination.

解法：如果能比鬼先到终点，就能赢。

## 链表

**19. Remove Nth Node From End**两个指针O(n)

**138. Copy List with Random Pointer**利用原链表指针相对位置关系=》指针关系

**142. Linked List Cycle II**

解: using two pointers, one of them one step at a time. another pointer each take two steps. Suppose the first meet at step k,the length of the Cycle is r. so..2k-k=nr,k=nr

Now, the distance between the start node of list and the start node of cycle is s. the distance between the start of list and the first meeting node is k(the pointer which wake one step at a time waked k steps).the distance between the start node of cycle and the first meeting node is m, so...s=k-m,

s=nr-m=(n-1)r+(r-m),here we takes n = 1.

**using one pointer start from the start node of list, another pointer start from the first meeting node, all of them wake one step at a time, the first time they meeting each other is the start of the cycle.**

**剑 题37 两个链表的第一个公共节点**

两个指针算出长度差值，再走一遍。

## 区间维护

**区间合并**

56 Merge Intervals排序以后按顺序合并

228. Summary Ranges：Given a sorted integer array without duplicates, return the summary of its ranges.

**非重叠区域计数**

646. Maximum Length of Pair Chain

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

452. Minimum Number of Arrows to Burst Balloons

435. Non-overlapping Intervals：Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

课程时间安排

<https://en.wikipedia.org/wiki/Interval_scheduling#Interval_Scheduling_Maximization>

非重叠区域计数 end

## 二叉树

236. Lowest Common Ancestor of a Binary Tree

根后序遍历

117. Populating Next Right Pointers in Each Node II

层次遍历

**114. Flatten Binary Tree to Linked List**

**652. Find Duplicate Subtrees**

把树转化为标志id的思想。

Suppose we have a unique identifier for subtrees: two subtrees are the same if and only if they have the same id.

Then, for a node with left child id of x and right child id of y, (node.val, x, y) uniquely determines the tree.

int t;

Map<String, Integer> trees;

Map<Integer, Integer> count;

List<TreeNode> ans;

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

t = 1;

trees = new HashMap();

count = new HashMap();

ans = new ArrayList();

lookup(root);

return ans;

}

public int lookup(TreeNode node) {

if (node == null) return 0;

String serial = node.val + "," + lookup(node.left) + "," + lookup(node.right);

int uid = trees.computeIfAbsent(serial, x-> t++);

count.put(uid, count.getOrDefault(uid, 0) + 1);

if (count.get(uid) == 2)

ans.add(node);

return uid;

}

623. Add One Row to Tree

### 路径

路径dfs

113. Path Sum II：Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

437. Path Sum III：单条路径任意点

优化算法：每个路径构建前缀和

124. Binary Tree Maximum Path Sum：任意点到点

### Level

129. Sum Root to Leaf Numbers：Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.An example is the root-to-leaf path 1->2->3 which represents the number 123.Find the total sum of all root-to-leaf numbers.

换一个维度，从下往上，计算层次和。

337. House Robber III：二叉树必须隔层抢

There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

3

/ \

2 3

\ \

3 1

Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

Example 2:

3

/ \

4 5

/ \ \

1 3 1

Maximum amount of money the thief can rob = 4 + 5 = 9.

Dp

遍历

103. Binary Tree Zigzag Level Order Traversal：from left to right, then right to left for the next level

使用栈

## 图

**207 Course Schedule**依赖问题：拓扑排序（需要手写）：偏序变全序时间问题：贪心算法

**332. Reconstruct Itinerary**

欧拉回路

<https://www.cnblogs.com/acxblog/p/7390301.html>

<https://blog.csdn.net/u011466175/article/details/18861415>

解1：Fleury's algorith：沿着路径上的点，每次选择一个不会使图中点断开的边作为路径。如果没有，则任选一个。

解2：Hierholzer's algorithm

Hierholzer's is more efficient than Fleury's algorithm:

Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v.

It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

环检测

**802. Find Eventual Safe States：**In a directed graph, we start at some node and every turn, walk along a directed edge of the graph.  If we reach a node that is terminal (it has no outgoing directed edges), we stop.

say our starting node is *eventually safe* ⬄ we must eventually walk to a terminal node. More specifically, there exists a natural number K so that for any choice of where to walk, we must have stopped at a terminal node in less than K steps.

Which nodes are eventually safe?  Return them as an array in sorted order.

### 二分图

Our goal is trying to use two colors to color the graph and see if there are any adjacent nodes having the same color.

Initialize a color[] array for each node. Here are three states for colors[] array:

-1: Haven't been colored.

0: Blue.

1: Red.

For each node,

If it hasn't been colored, use a color to color it. Then use the other color to color all its adjacent nodes (DFS).

If it has been colored, check if the current color is the same as the color that is going to be used to color it.

## 极大极小

375. Guess Number Higher or Lower II：when you guess a particular number x, and you guess wrong, you pay $x. You win the game when you guess the number I picked.

解：For each number x in range[i~j]  
we do: result\_when\_pick\_x = x + **max**{DP([i~x-1]), DP([x+1, j])}  
--> *// the max means whenever you choose a number, the feedback is always bad and therefore leads you to a worse branch.*  
then we get DP([i~j]) = **min**{xi, ... ,xj}  
--> *// this min makes sure that you are minimizing your cost.*

**486. Predict the Winner**

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

**Approach #1 Using Recursion [Accepted]：太绕了，自己增添复杂性，可以不管。**

The idea behind the recursive approach is simple. The two players Player 1 and Player 2 will be taking turns alternately. For the Player 1 to be the winner, we need scorePlayer\_1≥scorePlayer\_2. Or in other terms, scorePlayer\_1−scorePlayer\_2≥0.

Thus, for the turn of Player 1, we can add its score obtained to the total score and for Player 2's turn, we can substract its score from the total score. At the end, we can check if the total score is greater than or equal to zero(equal score of both players), to predict that Player 1 will be the winner.

Thus, by making use of a recursive function winner(nums,s,e,turn) which predicts the winner for the numsnums array as the score array with the elements in the range of indices [s,e][s,e] currently being considered, given a particular player's turn, indicated by turn=1turn=1 being Player 1's turn and turn=-1turn=−1 being the Player 2's turn, we can predict the winner of the given problem by making the function call winner(nums,0,n-1,1). Here, nn refers to the length of numsnums array.

In every turn, we can either pick up the first(nums[s]nums[s]) or the last(nums[e]nums[e]) element of the current subarray. Since both the players are assumed to be playing smartly and making the best move at every step, both will tend to maximize their scores. Thus, we can make use of the same function winner to determine the maximum score possible for any of the players.

Now, at every step of the recursive process, we determine the maximum score possible for the current player. It will be the maximum one possible out of the scores obtained by picking the first or the last element of the current subarray.

To obtain the score possible from the remaining subarray, we can again make use of the same winner function and add the score corresponding to the point picked in the current function call. But, we need to take care of whether to add or subtract this score to the total score available. If it is Player 1's turn, we add the current number's score to the total score, otherwise, we need to subtract the same.

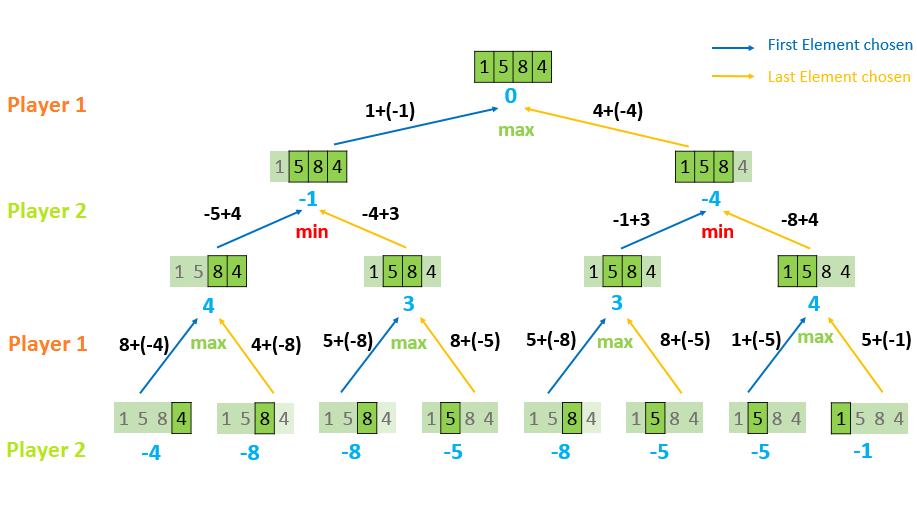
Thus, at every step, we need update the search space appropriately based on the element chosen and also invert the turnturn's value to indicate the turn change among the players and either add or subtract the current player's score from the total score available to determine the end result.

Further, note that the value returned at every step is given by turn \*\text{max}(turn \* a, turn \* b)turn∗max(turn∗a,turn∗b). This is equivalent to the statement max(a,b)max(a,b) for Player 1's turn and min(a,b)min(a,b) for Player 2's turn.

This is done because, looking from Player 1's perspective, for any move made by Player 1, it tends to leave the remaining subarray in a situation which minimizes the best score possible for Player 2, even if it plays in the best possible manner. But, when the turn passes to Player 1 again, for Player 1 to win, the remaining subarray should be left in a state such that the score obtained from this subarrray is maximum(for Player 1).

This is a general criteria for any arbitrary two player game and is commonly known as the Min-Max algorithm.

The following image shows how the scores are passed to determine the end result for a simple example.



public class Solution {

public boolean PredictTheWinner(int[] nums) {

return winner(nums, 0, nums.length - 1, 1) >= 0;

}

public int winner(int[] nums, int s, int e, int turn) {

if (s == e) return turn \* nums[s];

int a = turn \* nums[s] + winner(nums, s + 1, e, -turn);

int b = turn \* nums[e] + winner(nums, s, e - 1, -turn);

return turn \* Math.max(turn \* a, turn \* b);

}

}

Complexity Analysis

Time complexity : O(2^n)O(2

​n

​​ ). Size of recursion tree will be 2^n2

​n

​​ . Here, nn refers to the length of numsnums array.

Space complexity : O(n)O(n). The depth of the recursion tree can go upto nn.

**Approach #2 Similar Approach [Accepted]**

If the current turn belongs to, say Player 1, we pick up an element, say xx, from either end, and give the turn to Player 2. Thus, if we obtain the score for the remaining elements(leaving xx), this score, now belongs to Player 2. Thus, since Player 2 is competing against Player 1, this score should be subtracted from Player 1's current(local) score(xx) to obtain the effective score of Player 1 at the current instant.

While returning the result from winner for the current function call, we return the larger of the effective scores possible by choosing either the first or the last element from the currently available subarray.

in order to remove the duplicate function calls, we can make use of a 2-D memoization array, memomemo, such that we can store the result obtained for the function call winner for a subarray with starting and ending indices being s and e at memo[s][e].

public class Solution {

public boolean PredictTheWinner(int[] nums) {

Integer[][] memo = new Integer[nums.length][nums.length];

return winner(nums, 0, nums.length - 1, memo) >= 0;

}

public int winner(int[] nums, int s, int e, Integer[][] memo) {

if (s == e) return nums[s];

if (memo[s][e] != null) return memo[s][e];

int a = nums[s] - winner(nums, s + 1, e, memo);

int b = nums[e] - winner(nums, s, e - 1, memo);

memo[s][e] = Math.max(a, b);

return memo[s][e];

}

}

Time complexity : O(n^2). The entire memomemo array of size nxn is filled only once. Here, n refers to the size of nums array.

Space complexity : O(n^2). memomemo array of size nnxnn is used for memoization.

**Approach #3 Dynamic Programming [Accepted]:**

the effective score for the current player for any given subarray nums[x:y] only depends on the elements within the range [x,y]. It mainly depends on whether the element nums[x] or nums[y] is chosen in the current turn and also on the maximum score possible for the other player from the remaining subarray left after choosing the current element. Thus, it is certain that the current effective score isn't dependent on the elements outside the range [x,y].

if know the maximum effective score possible for the subarray nums[x+1,y] and nums[x,y−1], we can easily determine the maximum effective score possible for the subarray nums[x,y].

public class Solution {

public boolean PredictTheWinner(int[] nums) {

int[][] dp = new int[nums.length + 1][nums.length];

for (int s = nums.length; s >= 0; s--) {

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

int a = nums[s] - dp[s + 1][e];

int b = nums[e] - dp[s][e - 1];

dp[s][e] = Math.max(a, b);

}

}

return dp[0][nums.length - 1] >= 0;

}

}

Time complexity : O(n^2)

Space complexity : O(n^2)

**Approach #4 1-D Dynamic Programming [Accepted]:**

for filling in any entry in dpdp array, only the entries in the next row(same column) and the previous column(same row) are needed.

Instead of making use of a new row in dpdp array for the current dpdp row's updations, we can overwrite the values in the previous row itself and consider the values as belonging to the new row's entries, since the older values won't be needed ever in the future again. Thus, instead of making use of a 2-D dpdp array, we can make use of a 1-D dpdp array and make the updations appropriately.

Time complexity : O(n^2). The elements of dpdp array are updated 1+2+3+...+n1+2+3+...+n times. Here, nn refers to the length of numsnums array.

Space complexity : O(n)). 1-D dpdp array of size nn is used.

**649. Dota2 Senate**

In the world of Dota2, there are two parties: the Radiant and the Dire.

The Dota2 senate consists of senators coming from two parties. Now the senate wants to make a decision about a change in the Dota2 game. The voting for this change is a round-based procedure. In each round, each senator can exercise one of the two rights:

1. Ban one senator's right:   
   A senator can make another senator lose **all his rights** in this and all the following rounds.
2. Announce the victory:   
   If this senator found the senators who still have rights to vote are all from **the same party**, he can announce the victory and make the decision about the change in the game.

Given a string representing each senator's party belonging. The character 'R' and 'D' represent the Radiant party and the Dire party respectively. Then if there are n senators, the size of the given string will be n.

The round-based procedure starts from the first senator to the last senator in the given order. This procedure will last until the end of voting. All the senators who have lost their rights will be skipped during the procedure.

Suppose every senator is smart enough and will play the best strategy for his own party, you need to predict which party will finally announce the victory and make the change in the Dota2 game. The output should be Radiant or Dire.

贪心，分析

https://leetcode.com/problems/can-i-win/discuss/95277/Java-solution-using-HashMap-with-detailed-explanation

## 拾遗

**lintcode 842 折纸**

第 1 条折痕总为 0

之后每次在新产生的折痕两侧产生一个凹折痕和一个凸折痕

\*于是这个序列可以看成是以下二叉树中序遍历的结果：

根结点的值为 0

每个节点的左子树的根结点值恒为 0，右子树的的根结点的值为 1。

**621. Task Scheduler**

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.

**767. Reorganize String**: Given a string S, check if the letters can be rearranged so that two characters that are adjacent to each other are not the same.

O(nlogA):同上，先对每个字母计数，再排列

**406. Queue Reconstruction by Height**

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.

Note:  
The number of people is less than 1,100.

Example

Input:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

一个人的位置，由比他高的人决定。从高到低确定位置。

**260. Single Number III：**捞针

全部两次+两个一次

按位划分成两个数组，转化成低级问题。

//more

# 常用解题思路

## 二分查找

剑 题8：旋转数组最小数字

1. 正常情况：二分查找：（max/min = mid，与习惯不同）
2. 没有旋转：直接判断出，返回。
3. 非严格递增序列：顺序查找。

## 答案空间穷举回溯搜索

大多数问题都可以用该方法解决。有些问题存在效率更高的方式；有些不存在，例如背包问题。

需要有能力看出哪些问题不能优化。

参考资料，算法导论NP问题。

心中一颗答案空间搜索树

两种搜索：dfs、bfs

Dfs用递归、栈实现

Bfs队列实现

细节优化：可以采用双端bfs、减支方法优化。虽然不能提高复杂度级别，可以快一点。

## DP

能够应用DP的问题，可能的形式：

1. 可以构建无后效性递归式
2. 可以应用回溯算法，并且有大量重复。

表面上是TopDown算法，一般可以转化为bottomUp算法。

1. 队列推进，可以直接看出dp

DP的空间，

思考时，可以用较大的空间。

最终的优化，看递归式，到底依赖了多少上一步的结果。

### 推进

**213. House Robber II**

You are a robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

解：多状态推荐类型

**740. Delete and Earn**

Given an array nums of integers, you can perform operations on the array.

In each operation, you pick any nums[i] and delete it to earn nums[i] points. After, you must delete **every** element equal to nums[i] - 1 or nums[i] + 1.

You start with 0 points. Return the maximum number of points you can earn by applying such operations.

解：Because all numbers are positive, if we "take" a number (use it to score points), we might as well take all copies of it, since we've already erased all its neighbors. We could keep a count of each number so we know how many points taking a number is worth total.

Now let's investigate what happens when we add a new number X (plus copies) that is larger than all previous numbers. Naively, our answer would be the previous answer, plus the value of X . However, this fails if our previous answer had a number taken that was adjacent to X.

Luckily, we can remedy this. Let's say we knew using, the value of our previous answer, and avoid, the value of our previous answer that doesn't use the previously largest value prev. Then we could compute new values of using and avoid appropriately.

* Time Complexity (Java): We performed a radix sort instead, so our complexity is O(N+W)*O*(*N*+*W*) where W*W* is the range of allowable values for nums[i].
* Space Complexity (Java): O(W)*O*(*W*), the size of our count.

### 拾遗

**650. 2 Keys Keyboard**

**I -- Naive DP**

let's define T(k) as the minimum number of steps to get exactly k **'A'**. Then the original problem will be T(n) and we have the following termination conditions: T(1) = 0.

Apparently T(k) will depend on the sequence of operations performed to get k **'A'**.

1. Can this sequence of operations end with Copy All? The answer is **NO**, because if this is the case, the last operation can always be removed without changing the number of **'A'** on the notepad yet yielding a smaller number of steps.
2. Can this sequence of operations contain only ? The answer is again **NO**, because at the beginning there is no character being copied so pasting along won't change the number of **'A'** on the notepad (a special case is when n = 1, but then we don't even need any operations).

So we conclude the sequence of operations must end with Paste with at least one Copy All in the middle.

from the point of the last Paste, it only cares about characters which are copied last time, that is, the first Copy All operation from the end.

Assume the number of **'A'** is i when this Copy All operation is performed. Then how many more steps do we need to get k **'A'** on the notepad by pasting only? The answer is (k-i)/i.

So the total number of steps from getting i **'A'** to k **'A'** is k / i, that is, one Copy All plus (k-i)/i Paste.

Since we aim to minimize number of steps to get k **'A'**, we surely want to minimize the number of steps to i **'A'**, which by definition is T(i), therefore the total number of steps getting k **'A'** for this case is given by T(i) + k/i.

Since we don't really know when to perform the last Copy All operation, we can try each options. we have a total of k - 1 such choices (i running from 1 to k - 1). we require that the number of steps be integers, therefore i must be a divisor of k. In summary, we have:

**T(k) = min(T(i) + k/i) where 1 <= i < k && k % i == 0.**

with time complexity O(n^2) and space complexity O(n).

**public** **static** **int** **minSteps**(**int** n) {

**int**[] dp = **new** **int**[n + 1];

**for** (**int** k = 2; k <= n; k++) {

dp[k] = Integer.MAX\_VALUE;

**for** (**int** i = 1; i < k; i++) {

**if** (k % i != 0) **continue**;

dp[k] = Math.min(dp[k], dp[i] + k / i);

}

}

**return** dp[n];

}

**II -- Optimized DP**

It turns out that the inner loop in the naive DP solution can terminate early based on the following observation: "In **ascending** order, let k\_1, k\_2, ..., k\_m be the **proper divisors** of k, then we have T(k) = T(k\_m) + k / k\_m <= T(k\_j) + k / k\_j for all 1 <= j < m". （选较小的除数）

证明：

First the statement is true for the simple case when n = 2. Next we assume it is valid for all cases with n < k, and will show it holds for n = k.

To smoothen the proof, it is useful to introduce the [prime factorization](https://en.wikipedia.org/wiki/Integer_factorization) of integers, which says any positive integer can be uniquely decomposed into product of prime numbers. For an integer k, let [p\_1, p\_2, ..., p\_t] be the prime numbers in ascending order in its factorization and [e\_1, e\_2, ..., e\_t] be the corresponding exponent array, then k = p\_1^e\_1 \* p\_2^e\_2 \* ... \* p\_t^e\_t. Here we assume all the exponents are positive (i.e., prime factors of zero exponents are ignored).

Now if another integer k' is a divisor of k, then the prime factorization of k' must satisfy the following two conditions:

1. The exponents of prime factors other than [p\_1, p\_2, ..., p\_t] must be zero.
2. Let [e'\_1, e'\_2, ..., e'\_t] be the corresponding exponent array for k' in reference to [p\_1, p\_2, ..., p\_t], then e'\_j <= e\_j for all 1 <= j <= t(Note that now e'\_j is not necessarily positive but can be zero).

With the notations given above, we know k, k\_m and k\_i can be represented by three different exponent arrays, in reference to the same prime factors [p\_1, p\_2, ..., p\_t], since the latter two are proper divisors of the former. Assume again the exponent array for k is [e\_1, e\_2, ..., e\_t], then the exponent array for k\_mwill be [e\_1 - 1, e\_2, ..., e\_t], due to the fact that k\_m is the largest proper divisor of k. Let [e'\_1, e'\_2, ..., e'\_t] be the exponent array for k\_i, we need to consider two cases: k\_m % k\_i != 0 or k\_m % k\_i == 0.

For the former case, k\_i is not a divisor of k\_m. From our conclusion above, we must have e'\_1 > e\_1 - 1 >= 0, i.e., e'\_1 is positive. This is because e'\_j <= e\_j holds for all 2 <= j <= t as k\_i is a factor of k. If e'\_1 <= e'\_1 - 1, then k\_i will also be a factor of k\_m, contradicting with the condition that k\_m % k\_i != 0. Now let d\_i be the largest proper factor of k\_i, then the exponent array of d\_i will be [e'\_1 - 1, e'\_2, ..., e'\_t]. It is easy to show that d\_i will also be a factor of k\_m, since e'\_1 - 1 <= e\_1 - 1. Also we have the following equation k \* d\_i = k\_m \* k\_i, which manifests itself in the notation of exponent arrays: k \* d\_i = [e\_1 + e'\_1 - 1, e\_2 + e'\_2, ..., e\_t + e'\_t] and k\_m \* k\_i = [e\_1 - 1 + e'\_1, e\_2 + e'\_2, ..., e\_t + e'\_t]. Here comes the real proof for this case: T(k\_m) + k/k\_m <= T(d\_i) + k\_m/d\_i + k/k\_m = T(d\_i) + k/k\_i + k\_i/d\_i = T(k\_i) + k/k\_i. The first inequality comes from the induction assumption: k\_m < k and k\_m % d\_i == 0 implies T(k\_m) <= T(d\_i) + k\_m/d\_i. The following equality takes advantage of the equation k \* d\_i = k\_m \* k\_i and finally the last equality uses the induction assumption again: T(k\_i) = T(d\_i) + k\_i/d\_i.

For the latter case, k\_i is a proper divisor of k\_m. Then by our induction assumption, T(k\_m) + k/k\_m <= T(k\_i) + k\_m/k\_i + k/k\_m. We only need to show that k\_m/k\_i + k/k\_m <= k/k\_i. Note that k = k\_m \* p\_1, then k/k\_i = p\_1 \* k\_m/k\_i = p\_1 + p\_1 \* (k\_m/k\_i - 1) >= p\_1 + k\_m/k\_i = k/k\_m + k\_m/k\_i, where we have used the facts that p\_1 >= 2 and k\_m/k\_i >= 2 to derive the inequality in the middle.

Thus we conclude the induction assumption is also true for the case n = k, hence validate our observation above.

Here is the optimized DP solution:

**public** **int** **minSteps**(**int** n) {

**int**[] dp = **new** **int**[n + 1];

**for** (**int** k = 2, i = 0; k <= n; k++) {

**for** (i = k >> 1; i >= 1 && k % i != 0; i--);//i是最小的除数

dp[k] = dp[i] + k / i;

}

**return** dp[n];

}

**III -- Non-DP solution**

Our DP solution is based on the assumption that there is overlapping among subproblems. However, from the observation in part II, to solve T(k), we only need T(k\_m), which in turn only requires knowledge of T(k\_m\_m), ..., where k\_m is the largest proper divisor of k\_m, and k\_m\_m is the largest proper divisor of k\_m, and so on. Since the largest proper divisors are decreasing, there won't be any overlapping among the subproblems and the DP solution is an overkill here.

Here is the non-DP solution, which reduces the space complexity to O(1):

**public** **int** **minSteps**(**int** n) {

**int** res = 0;

**for** (**int** k = n, i = 0; k > 1; k = i) {

**for** (i = k >> 1; k % i != 0; i--);

res += k / i;

}

**return** res;

}

If we look at the solution above more carefully, it is actually equivalent to adding up the prime factors of n, since the transformation sequence of the largest proper divisors will be (in the form of exponent array): [e\_1, e\_2, ..., e\_t] ==> [e\_1 - 1, e\_2, ..., e\_t] ==> ... ==> [0, e\_2, ..., e\_t] ==> [0, e\_2 - 1, ..., e\_t] ==> ... ==> [0, 0, ..., 0]. And for each such divisor, we add the corresponding prime factor to the resulting number of steps. Therefore the final answer will be p\_1 \* e\_1 + p\_2 \* e\_2 + ... + p\_t \* e\_t. Here is the reformulated non-DP solution (more of a pure math solution):

**public** **static** **int** **minSteps**(**int** n) {

**int** res = 0;

**for** (**int** k = 2; k <= n; k++) {

**for** (; n % k == 0; res += k, n /= k);

}

**return** res;

}

IV -- Summary

Although we have developed the non-DP solutions above, the ideas of the naive DP in part I carry merit as they can be adapted to solve the 4 Keys Keyboardproblem. The middle two operations Ctrl-A and Ctrl-C can be combined into one that resembles the Copy All operation here. Then again we can just try each position to perform the last Ctrl-A and Ctrl-C combo operation and choose the one that yields the most number of **'A'** on the screen. A key difference between the two is that we need to press the keys twice to complete the combo operation. And interestingly enough, we also have observations similar in part II, which can reduce the time complexity down to O(n). Anyway, hope this helps for solving the 2 Keys and 4 Keys Keyboard problems.

**微软 Lintcode 867. 四键键盘**

假设你有一个特殊的键盘，键盘上有如下键:

键1:(A):在屏幕上打印一个'A'。

键2:(Ctrl-A):选择整个屏幕。

键3:(Ctrl-C):复制选择到缓冲区。

键4:(Ctrl-V):在屏幕上已有的内容后面追加打印缓冲区的内容。

你只能按键盘上N次(使用以上四个键)，找出你可以在屏幕上打印的“A”的最大数量

**解：**N步最少都能打印N个A出来。

由于全选和复制要占用两步，所以能增加A的个数的操作其实只有N-2步，最简单的方法就是遍历所有的情况然后取最大值，打印A的次数在[1, N-3]之间，粘贴的次数为N-2-i，加上打印出的部分，就是N-1-i了，参见代码如下：

**解法一：递归遍历：**

class Solution {

public:

int maxA(int N) {

int res = N;

for (int i = 1; i < N - 2; ++i) {

res = max(res, maxA(i) \* (N - 1 - i));

}

return res;

}

};

这道题也可以用DP来做，用一个一维数组dp，其中dp[i]表示步骤总数为i时，能打印出的最多A的个数，初始化为N+1个，然后我们来想递推公式怎么求。对于dp[i]来说，求法其实跟上面的方法一样，还是要遍历所有打印A的个数，然后乘以粘贴的次数加1，用来更新dp[i]，参见代码如下：

解法二：

class Solution {

public:

int maxA(int N) {

vector<int> dp(N + 1, 0);

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

dp[i] = i;

for (int j = 1; j < i - 2; ++j) {

dp[i] = max(dp[i], dp[j] \* (i - j - 1));

}

}

return dp[N];

}

};

这道题还有个O(1)时间复杂度的解法，好像利用了数学知识，参见[这个帖子](https://discuss.leetcode.com/topic/97764/o-1-time-o-1-space-c-solution-possibly-shortest-and-fastest)

## 贪心算法专题

关键是需要证明为什么贪心是对的。

狭义贪心算法是动态规划的特例。贪心算法代码看起来比动态规划简单，实际上思维上要复杂一点。算法导论有介绍**321 Create Maximum Number** Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + nfrom digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits.

**Example 1:**

**Input:**

nums1 = [3, 4, 6, 5]

nums2 = [9, 1, 2, 5, 8, 3]

k = 5

**Output:**

[9, 8, 6, 5, 3]

广义贪心算法

## 逆向思维，从答案空间反推

这类问题通常答案空间比较小。可以采用二分搜索、计数排序等。

## 窗口法

通常指窗口不断沿序列右移。

## 两个指针

与窗口不同，指针变化方向没有要求。

Lint 57 三数之和：给出一个有n个整数的数组S，在S中找到三个整数a, b, c，找到所有使得a + b + c = 0的三元组。

**public** List<List<Integer>> threeSum(**int**[] nums) {  
 List<List<Integer>> result = **new** ArrayList<>();  
 **if**(nums == **null** || nums.**length** < 3) **return** result;  
  
 Set<List<Integer>> midResult = **new** HashSet<>();  
 Arrays.*sort*(nums);  
 **for** (**int** ai = 0; ai < nums.**length**; ai++) {  
 **if** (ai > 0 && nums[ai] == nums[ai-1]) **continue**;  
 **int** bi = ai+1, ci = nums.**length**-1;  
 **while**(bi < ci){  
 **if** (bi == ai){  
 bi++;  
 **continue**;  
 }  
 **if**(ci == ai) {  
 ci--;  
 **continue**;  
 }  
 **if** (nums[bi] + nums[ci] == -nums[ai]){  
 List<Integer> oneResult = **new** ArrayList<>();  
 oneResult.add(nums[ai]);  
 oneResult.add(nums[bi]);  
 oneResult.add(nums[ci]);  
 oneResult.sort((o1, o2) -> {**return** o1-o2;});  
 midResult.add(oneResult);  
 *//跳过重复的数* **do**{bi++;}**while**(bi < ci && nums[bi] == nums[bi-1]);  
 **do**{ci--;}**while**(bi< ci && nums[ci] == nums[ci+1]);  
 **continue**;  
 }  
 **if** (nums[bi] + nums[ci] < -nums[ai] ) bi ++;  
 **if** (nums[bi] + nums[ci] > -nums[ai] ) ci --;  
 }  
 }  
  
 result.addAll(midResult);  
 **return** result;  
}

## 拾遗

其他常用解决方案：分治、哈希、双指针、排序双向bfs

# 繁琐处理问题

## 低通过率的题目

**微软 Leetcode 794. Valid Tic-Tac-Toe State**

1. 最多存在一个3连.如果存在交叉，可以存在两个3连

2. 如果x赢，则x比o多1；如果O赢，则相等。

**420 StrongPassword**: 将字符串转成强密码需要多少步

三个条件，正向和反向分析，有些条件可以合并处理。

窗口的思路是错误的

连续字母的解决注意3k+2的特点。

The basic principle is straightforward: if we want to make MINIMUM changes to turn s into a strong password, each change made should fix as many problems as possible.

So to start, let's first identify all the problems in the input string s and list what changes are suitable for righting each of them. To clarify, each change should be characterized by at least two parts: the **type of operation** it takes and the **position in the string** where the operation is applied (Note: Ideally we should also include the characters involved in the operation and the "power" of each operation for eliminating problems but they turn out to be partially relevant so I will mention them only when appropriate).

1. Length problem: if the total length is **less than 6**, the change that should be made is (**insert, any position**), which reads as "the operation is insertion and it can be applied to anywhere in the string". If the total length is **greater than 20**, then the change should be (**delete, any position**).
2. Missing letter or digit: if any of the lowercase/uppercase letters or digits is missing, we can do either (**insert, any position**) or (**replace, any position**) to correct it. (Note here the characters for insertion or replacement can only be those missing.)
3. Repeating characters: for repeating characters, all three operations are allowed but the positions where they can be applied are limited within the repeating characters. For example, to fix "aaaaa", we can do one replacement (replace the middle 'a') or two insertions (one after the second 'a' and one after the fourth 'a') or three deletions (delete any of the three 'a's). So the possible changes are (**replace, repeating characters**), (**insert, repeating characters**), (**delete, repeating characters**). (Note however the "power" of each operation for fixing the problem are different -- replacement is the strongest while deletion is the weakest.)

All right, what's next? If we want a change to eliminate as many problems as it can, it must be shared among the possible solutions to each problem it can fix. So our task is to find out possible overlapping among the changes for fixing each problem.

Since there are most (three) changes allowed for the third problem, we may start from combinations first problem & third problem and second problem & third problem. It's not too hard to conclude that any change that can fix the first or second problems is also able to fix the third one (since the type of operation here is irrelevant, we are free to choose the position of the operation to match those of the repeating characters). For combination first problem & second problem, depending on the length of the string, there will be overlapping if length is less than 6 or no overlapping if length is greater than 20.

From the analyses above, it seems worthwhile to distinguish between the two cases: when the input string is too short or too long.

For the former case, it can be shown that the total changes needed to fix the first and second problems always outnumber those for the third one. Since whatever change used fixing the first two problems can also correct the third one, we may concern ourselves with only the first two. Also as there is overlapping between the changes for fixing the first two problems, we will prefer those overlapping ones, i.e. (**insert, any position**). Another point is that the characters involved in the operation matters now. To fix the first problem, only those missing characters can be inserted while for the second condition, it can be any character. Therefore correcting the first problem takes precedence over the second one.

For the latter case, there is overlapping between the **first & third** and **second & third** problems, so those overlapping changes will be taken, i.e., first problem => (**delete, any position**), second problem => (**replace, any position**). The reason not to use (**insert, any position**) for the second problem is that it contradicts the changes made to the first problem (therefore has the tendency to cancel its effects). After fixing the first two problems, what operation(s) should we choose for the third one?

Now the "power" of each operation for eliminating problems comes into play. For the third problem, the "power" of each operation will be measured by the maximum number of repeating characters it is able to get rid of. For example, one replacement can eliminate at most 5 repeating characters while insertion and deletion can do at most 4 and 3, respectively. In this case, we say replacement has more "power" than insertion or deletion. Intuitively the more "powerful" the operation is, the less number of changes is needed for correcting the problem. Therefore (**replace, repeating characters**) triumphs in terms of fixing the third problem.

Furthermore, another very interesting point shows up when the "power" of operation is taken into consideration (And thank [**yicui**](https://discuss.leetcode.com/user/yicui) for pointing it out). As I mentioned that there is overlapping between changes made for fixing the first two problems and for the third one, which means the operations chosen above for the first two problems will also be applied to the third one. For the second problem with change chosen as (**replace, any position**), we have no problem adapting it so that it coincides with the optimal change (**replace, repeating characters**) made for the third problem. However, there is no way to do the same for the first problem with change (**delete, any position**). We have a conflict now!

How do we reconcile it? The trick is that for a sequence of repeating characters of length k (k >= 3), instead of turning it all the way into a sequence of length 2 (so as to fix the repeating character problem) by the change (**delete, any position**), we will first reduce its length to (3m + 2), where (3m + 2) is the largest integer of the form yet no more than k. That is to say, if k is a multiple of 3, we apply once such change so its length will become (k - 1); else if k is a multiple of 3 plus 1, we apply twice such change to cut its length down to (k - 2), provided we have more such changes to spare (be careful here as we need at least two changes but the remaining available changes may be less than that, so we should stick to the smaller one: 2 or the remaining available changes). The reason is that the optimal change (**replace, repeating characters**) for the third problem will be most "powerful" when the total length of the repeating characters is of this form. Of course, if we still have more changes (**delete, any position**) to do after that, then we are free to turn the repeating sequence all the way into a sequence of length 2.

Here is the java program based on the above analyses. Both time and space complexity is O(n). Not sure if we can reduce the space down to O(1) by computing the arr array on the fly. A quick explanation is given at the end.

public int strongPasswordChecker(String s) {

int res = 0, a = 1, A = 1, d = 1;

char[] carr = s.toCharArray();

int[] arr = new int[carr.length];

**for** (int i = 0; i < arr.length;) {

**if** (Character.isLowerCase(carr[i])) a = 0;

**if** (Character.isUpperCase(carr[i])) A = 0;

**if** (Character.isDigit(carr[i])) d = 0;

int j = i;

**while** (i < carr.length && carr[i] == carr[j]) i++;

arr[j] = i - j;

}

int total\_missing = (a + A + d);

**if** (arr.length < 6) {

res += total\_missing + Math.max(0, 6 - (arr.length + total\_missing));

} **else** {

int over\_len = Math.max(arr.length - 20, 0), left\_over = 0;

res += over\_len;

**for** (int k = 1; k < 3; k++) {

**for** (int i = 0; i < arr.length && over\_len > 0; i++) {

**if** (arr[i] < 3 || arr[i] % 3 != (k - 1)) **continue**;

arr[i] -= Math.min(over\_len, k);

over\_len -= k;

}

}

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

**if** (arr[i] >= 3 && over\_len > 0) {

int need = arr[i] - 2;

arr[i] -= over\_len;

over\_len -= need;

}

**if** (arr[i] >= 3) left\_over += arr[i] / 3;

}

res += Math.max(total\_missing, left\_over);

}

**return** res;

}

A quick explanation of the program:

1. res denotes the minimum changes; a, A and d are the number of missing lowercase letters, uppercase letters and digits, respectively; arr is an integer array whose element will be the number of repeating characters starting at the corresponding position in the string.
2. In the following loop we fill in the values for a, A, d and arr to identify the problems for each condition. The total number of missing characters total\_missing will be the summation of a, A, d and fixing this problem takes at least total\_missing changes.
3. We then distinguish the two cases when the string is too short or too long. If it is too short, we pad its length to at least 6 (note in this case we've already inserted total\_missing characters so the new length is the summation of the original length and total\_missing).
4. Otherwise, to fix the first condition, we need to delete over\_len (number of surplus characters) characters. Since fixing the first problem also corrects the third one, we need to get rid of those parts from the arr array. And as I mentioned, we need to first turn all numbers in the arr array greater than 2 into the form of (3m + 2) and then reduce them all the way to 2 if over\_len is still greater than 0. After that, we need to replace total\_missing characters to fix the second problem, which also fixes part (or all) of the third problem. Therefore we only need to take the larger number of changes needed for fixing the second problem (which is total\_missing) and for the third one (which is left\_over, since it is the number of changes still needed after fixing the first problem).

**151. Reverse Words in a String**

**public** String reverseWords(String s) {  
 *//判空* **if** (s == **null**) **return null**;  
 **if** (s.length() == 0) **return ""**;  
 *//整体反转* **char**[] letters = s.toCharArray();  
 reverse(0, letters.**length**-1, letters);  
 *//逐个单词反转* **int** start = -1, end =-1;  
 **while**(end < letters.**length**-1){  
 start = end+1;  
 **while** (start < letters.**length** && letters[start] == **' '**) start ++;  
 end = start;  
 **while** (end+1 < letters.**length** && letters[end+1] != **' '**)end ++;  
 **if** (end < letters.**length**) reverse(start, end, letters);  
 }  
 *//去除多余的0* **boolean** flagHasSpace = **true**;  
 *//当前可以空余的位置* **int** endIndex = 0;  
 **for** (**int** i = 0; i < letters.**length**; i++) {  
 **if** (letters[i] == **' '**){  
 **if** (flagHasSpace){  
 **continue**;  
 }**else**{  
 letters[endIndex] = letters[i];  
 endIndex++;  
 flagHasSpace = **true**;  
 }  
 }**else** {  
 flagHasSpace = **false**;  
 letters[endIndex] = letters[i];  
 endIndex++;  
 }  
 }  
 *//去除最后一个空格* **if** (endIndex >0 && endIndex-1 < letters.**length** && letters[endIndex-1] == **' '**){  
 endIndex--;  
 }  
 **return new** String(letters, 0, endIndex);  
}

## 数字验证/解析

<https://leetcode.com/problems/valid-number/discuss/23977/A-clean-design-solution-By-using-design-pattern>

测试用例总结

整数、小数、科学计数法1. 前后空格2. 前0，多个03. 负04. 小数，后多个05. 小数，小数点前后没有06. 指数，指数为07. 字母、空格，中间夹杂8. 空值test(2, " 123 ", true);test(3, "0", true);test(4, "0123", true); //Cannot agreetest(5, "00", true); //Cannot agreetest(6, "-10", true);test(7, "-0", true);test(8, "123.5", true);test(9, "123.000000", true);test(10, "-500.777", true);test(11, "0.0000001", true);test(12, "0.00000", true);test(13, "0.", true); //Cannot be more disagree!!!test(14, "00.5", true); /ly cannot agreetest(15, "123e1", true);test(16, "1.23e10", true);test(17, "0.5e-10", true);test(18, "1.0e4.5", false);test(19, "0.5e04", true);test(20, "12 3", false);test(21, "1a3", false);test(22, "", false);test(23, " ", false);test(24, null, false);test(25, ".1", true); //Ok, if you say sotest(26, ".", false);test(27, "2e0", true); //Really?!test(28, "+.8", true); (29, " 005047e+6", true); //Damn = =|||Here is the final Regex I got based on their definitionPattern.matches("(\\+|-)?(\\d+(\\.\\d\*)?|\\.\\d+)(e(\\+|-)?\\d+)?", s);But I thought my original one should be more rigorous!Pattern.matches("-?(([1-9]{1}+\\d\*|0)(\\.\\d+)?|\\.\\d+)(e-?[1-9]{1}+\\d\*)?", s);

数字解析

https://leetcode.com/problems/string-to-integer-atoi/discuss/4654/My-simple-solution

I think we only need to handle four cases:

discards all leading whitespaces

sign of the number

overflow

invalid input

过程中计算负数，因为负数范围大。

注意越界的处理

//容错比较低，需要提前验证

public static int parseInt(String s, int radix) throws NumberFormatException

{

if (s == null) throw new NumberFormatException("null");

if (radix < Character.MIN\_RADIX) {

throw new NumberFormatException("radix " + radix +

" less than Character.MIN\_RADIX");

}

if (radix > Character.MAX\_RADIX) {

throw new NumberFormatException("radix " + radix +

" greater than Character.MAX\_RADIX");

}

int result = 0;

boolean negative = false;

int i = 0, len = s.length();

int limit = -Integer.MAX\_VALUE;

int multmin;

int digit;

if (len > 0) {

char firstChar = s.charAt(0);

if (firstChar < '0') { // Possible leading "+" or "-"

if (firstChar == '-') {

negative = true;

limit = Integer.MIN\_VALUE;

} else if (firstChar != '+')

throw NumberFormatException.forInputString(s);

if (len == 1) // Cannot have lone "+" or "-"

throw NumberFormatException.forInputString(s);

i++;

}

multmin = limit / radix;

while (i < len) {

// Accumulating negatively avoids surprises near MAX\_VALUE

digit = Character.digit(s.charAt(i++),radix);

if (digit < 0) {

throw NumberFormatException.forInputString(s);

}

//是否可以乘，升位

if (result < multmin) {

throw NumberFormatException.forInputString(s);

}

result \*= radix;

//是否可以增加本位的值

if (result < limit + digit) {

throw NumberFormatException.forInputString(s);

}

result -= digit;

}

} else {

throw NumberFormatException.forInputString(s);

}

return negative ? result : -result;

}

## 其他验证

**393. UTF-8 Validation**

A character in UTF8 can be from 1 to 4 bytes long:

For 1-byte character, the first bit is a 0, followed by its unicode code.

For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

Char. number range | UTF-8 octet sequence

(hexadecimal) | (binary)

--------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

**71. Simplify Path**

* Did you consider the case where **path** = "/../"?  
  In this case, you should return "/".
* Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/".  
  In this case, you should ignore redundant slashes and return "/home/foo".

单元测试：null, “”, “/”

public String simplifyPath(String path) {

Deque<String> stack = new LinkedList<>();

Set<String> skip = new HashSet<>(Arrays.asList("..",".",""));

for (String dir : path.split("/")) {

if (dir.equals("..") && !stack.isEmpty()) stack.pop();

else if (!skip.contains(dir)) stack.push(dir);

}

String res = "";

for (String dir : stack) res = "/" + dir + res;

**return res.isEmpty() ? "/" : res;**

}

**591. Tag Validator**

Input: "<DIV>This is the first line <![CDATA[<div>]]></DIV>"

**Approach #1 Using Stack [Accepted]**

checking the following properties.

* 1. The code should be wrapped in valid closed tag.
  2. The TAG\_NAME should be valid.
  3. The TAG\_CONTENT should be valid.
  4. The cdata should be valid.

All the tags should be closed. i.e. each start-tag should have a corresponding end-tag and the order of the tags should be correct as well.

firstly, we need to identify which parts of the given string act as which part.

辨识tag和cdata

We iterate over the given string. Whenever a < is encountered(unless we are currently inside <![CDATA[...]]>), it indicates the beginning of either a TAG\_NAME(start tag or end tag) or the cdata.

If the character following this < is an !, the characters following this < can't be a part of a valid TAG\_NAME, since only upper-case letters(in case of a start tag) or / followed by upper-case letters(in the case of an end tag). Thus, the choice now narrows down to only cdata.

处理cdata

we need to check if the current bunch of characters following <!(including it) constitute a valid cdata. firstly we find out the first matching ]]> following the current <! to mark the ending of cdata. If no such matching ]]> exists, the string is considered as invalid.

the <! should also be followed by CDATA[ for the cdata to be valid.

The characters lying inside the <![CDATA[ and ]]> do not have any constraints on them.

处理tag

If the character immediately following the < encountered isn't an !, this < can only mark the beginnning of TAG\_NAME. Now, since a valid start tag can't contain anything except upper-case letters, if a / is found after <, the </ pair indicates the beginning of an end tag.

when a < refers to the beginning of a TAG\_NAME(either start-tag or end-tag), we find out the first closing > following the < to find out the substring(say s), that constitutes the TAG\_NAME.

验证tag name

for every such s, we check if it contains all upper-case letters and also check its length(It should be between 1 to 9).

处理成对问题

the tags always exist in pairs. Further, we can note that in case of multiple TAG\_NAME's, the TAG\_NAME whose start-tag comes later than the other ones, should have its end-tag appearing before the end-tags of those other TAG\_NAME's. i.e. the tag which starts later should end first.

we can make use of a stackstack. whenever we find out a valid start-tag, we push its TAG\_NAME string onto a stackstack. whenever an end-tag is found, we compare its TAG\_NAME with the TAG\_NAME at the top the stackstack and remove this element from the stackstack.

after the complete string has been traversed, the stack should be empty if all the start-tags have their corresponding end-tags as well.

在中间过程中，栈不该为空

Further, we also need to ensure that the given code is completely enclosed within closed tags. For this, we need to ensure that the first cdata found is also inside the closed tags. Thus, when we find a possibility of the presence of cdata, we proceed further only if we've already found a start tag, indicated by a non-empty stack. Further, to ensure that no data lies after the last end-tag, we need to ensure that the stack doesn't become empty before we reach the end of the given codecode string.

# 数学

## 数字

### 数字表示

**微软 13. Roman to Integer**

Roman numerals are represented by seven different symbols:

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

two is written as II. Twelve is written as, XII. The number twenty seven is written as XXVII, which is XX + V + II.

Roman numerals are written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four.

There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer. Input is guaranteed to be within the range from 1 to 3999.

**Example 1:**

**Input:** "III"

**Output:** 3

**Example 2:**

**Input:** "IV"

**Output:** 4

**Example 3:**

**Input:** "IX"

**Output:** 9

**Example 4:**

**Input:** "LVIII"

**Output:** 58

**Explanation:** C = 100, L = 50, XXX = 30 and III = 3.

**Example 5:**

**Input:** "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**public** **static** **int** romanToInt(**String** s) {

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

HashMap<Character, Integer> map = **new** HashMap<Character, Integer>();

map.put('I', 1);

map.put('V', 5);

map.put('X', 10);

map.put('L', 50);

map.put('C', 100);

map.put('D', 500);

map.put('M', 1000);

**int** len = s.length(), result = map.get(s.charAt(len - 1));

for (**int** i = len - 2; i >= 0; i--) {

//从后往前，遇到大的数就累加，遇到小的数则减。

if (map.get(s.charAt(i)) >= map.get(s.charAt(i + 1))) result += map.get(s.charAt(i));

else result -= map.get(s.charAt(i));

}

return result;

}

**12. Integer to Roman**

public **static** String intToRoman(int num) {

String M[] = {"", "M", "MM", "MMM"};

String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

**return** M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];

}

**273. Integer to English Words**

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 231 - 1.

private final String[] LESS\_THAN\_20 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

private final String[] TENS = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

private final String[] THOUSANDS = {"", "Thousand", "Million", "Billion"};

public String numberToWords(int num) {

if (num == 0) return "Zero";

int i = 0;

String words = "";

while (num > 0) {

if (num % 1000 != 0) words = helper(num % 1000) +THOUSANDS[i] + " " + words;

num /= 1000;

i++;

}

return words.trim();

}

private String helper(int num) {

if (num == 0) return "";

if (num < 20) return LESS\_THAN\_20[num] + " ";

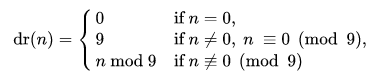
if (num < 100) return TENS[num / 10] + " " + helper(num % 10);

return LESS\_THAN\_20[num / 100] + " Hundred " + helper(num % 100);

}

### 数字的根

\*<https://en.wikipedia.org/wiki/Digital_root>



即dr(n)=1+((n-1) mod 9) （减1是为了处理0和9的情况？）

The digital root is the value modulo 9 because {\displaystyle 10\equiv 1{\pmod {9}},} 10\equiv 1{\pmod {9}}, and thus {\displaystyle 10^{k}\equiv 1^{k}\equiv 1{\pmod {9}},} 10^{k}\equiv 1^{k}\equiv 1{\pmod {9}}, so regardless of position, the value mod 9 is the same – {\displaystyle a\cdot 100\equiv a\cdot 10\equiv a{\pmod {9}}} a\cdot 100\equiv a\cdot 10\equiv a{\pmod {9}} – which is why digits can be meaningfully added. Concretely, for a three-digit number,

{\mbox{dr}}(abc)\equiv a\cdot 10^{2}+b\cdot 10+c\cdot 1\equiv a\cdot 1+b\cdot 1+c\cdot 1\equiv a+b+c{\pmod {9}}

To obtain the modular value with respect to other numbers n, one can take weighted sums, where the weight on the kth digit corresponds to the value of {\displaystyle 10^{k}} 10^{k} modulo n, or analogously for {\displaystyle b^{k}} b^{k} for different bases. This is simplest for 2, 5, and 10, where higher digits vanish (since 2 and 5 divide 10), which corresponds to the familiar fact that the divisibility of a decimal number with respect to 2, 5, and 10 can be checked by the last digit (even numbers end in 0, 2, 4, 6, or 8).

Also of note is the modulus 11: since {\displaystyle 10\equiv -1{\pmod {11}},} 10\equiv -1{\pmod {11}}, and thus {\displaystyle 10^{2}\equiv (-1)^{2}\equiv 1{\pmod {11}},} 10^{2}\equiv (-1)^{2}\equiv 1{\pmod {11}}, taking the alternating sum of digits yields the value modulo 11.

Some properties of digital roots

The digital root of a number is zero if and only if the number is itself zero.

{\displaystyle {\mathit {dr}}(n)=0\Leftrightarrow n=0.} {\mathit {dr}}(n)=0\Leftrightarrow n=0.

The digital root of a number is a positive integer if and only if the number is itself a positive integer.

{\displaystyle {\mathit {dr}}(n)>0\Leftrightarrow n>0.} {\mathit {dr}}(n)>0\Leftrightarrow n>0.

The digital root of {\displaystyle n} n is {\displaystyle n} n itself if and only if the number has exactly one digit.

{\displaystyle {\mathit {dr}}(n)=n\Leftrightarrow n\in \{0,1,2,3,4,5,6,7,8,9\}.} {\mathit {dr}}(n)=n\Leftrightarrow n\in \{0,1,2,3,4,5,6,7,8,9\}.

The digital root of {\displaystyle n} n is less than {\displaystyle n} n if and only if the number is greater than or equal to 10.

{\displaystyle {\mathit {dr}}(n)<n\Leftrightarrow n\geq 10.} {\mathit {dr}}(n)<n\Leftrightarrow n\geq 10.

The digital root of {\displaystyle a} a + {\displaystyle b} b is digital root of the sum of the digital root of {\displaystyle a} a and the digital root of {\displaystyle b} b.

{\displaystyle {\mathit {dr}}(a+b)={\mathit {dr}}({\mathit {dr}}(a)+{\mathit {dr}}(b)).} {\displaystyle {\mathit {dr}}(a+b)={\mathit {dr}}({\mathit {dr}}(a)+{\mathit {dr}}(b)).}

The digital root of {\displaystyle a} a - {\displaystyle b} b is congruent with the difference of the digital root of {\displaystyle a} a and the digital root of {\displaystyle b} b modulo 9.

{\displaystyle {\mathit {dr}}(a-b)\equiv {\mathit {dr}}(a)-{\mathit {dr}}(b){\pmod {9}}.} {\mathit {dr}}(a-b)\equiv {\mathit {dr}}(a)-{\mathit {dr}}(b){\pmod {9}}.

Especially, we can define the digital root of minus {\displaystyle n} n as follows:

{\displaystyle {\mathit {dr}}(-n)\equiv -{\mathit {dr}}(n){\pmod {9}}.} {\displaystyle {\mathit {dr}}(-n)\equiv -{\mathit {dr}}(n){\pmod {9}}.}

The digital root of {\displaystyle a} a × {\displaystyle b} b is digital root of the product of the digital root of {\displaystyle a} a and the digital root of {\displaystyle b} b.

{\displaystyle {\mathit {dr}}(a\times b)={\mathit {dr}}({\mathit {dr}}(a)\times {\mathit {dr}}(b)).} {\displaystyle {\mathit {dr}}(a\times b)={\mathit {dr}}({\mathit {dr}}(a)\times {\mathit {dr}}(b)).}

The digital root of a nonzero number is 9 if and only if the number is itself a multiple of 9.

{\displaystyle {\mathit {dr}}(n)=9\Leftrightarrow n=9m\ \ \ {\text{for}}\ m=1,2,3,\cdots .} {\mathit {dr}}(n)=9\Leftrightarrow n=9m\ \ \ {\text{for}}\ m=1,2,3,\cdots .

The digital root of a nonzero number is a multiple of 3 if and only if the number is itself a multiple of 3.

{\displaystyle {\begin{aligned}{\mathit {dr}}(n)&=3\Leftrightarrow n=9m+3&\ {\text{for}}\ m=0,1,2,\cdots ,\\{\mathit {dr}}(n)&=6\Leftrightarrow n=9m+6&\ {\text{for}}\ m=0,1,2,\cdots ,\\{\mathit {dr}}(n)&=9\Leftrightarrow n=9m&\ {\text{for}}\ m=1,2,3,\cdots .\end{aligned}}} {\begin{aligned}{\mathit {dr}}(n)&=3\Leftrightarrow n=9m+3&\ {\text{for}}\ m=0,1,2,\cdots ,\\{\mathit {dr}}(n)&=6\Leftrightarrow n=9m+6&\ {\text{for}}\ m=0,1,2,\cdots ,\\{\mathit {dr}}(n)&=9\Leftrightarrow n=9m&\ {\text{for}}\ m=1,2,3,\cdots .\end{aligned}}

The digital root of a factorial ≥ 6! is 9.

{\displaystyle {\mathit {dr}}(n!)=9\Leftrightarrow n\geq 6.} {\mathit {dr}}(n!)=9\Leftrightarrow n\geq 6.

The digital root of a square is 1, 4, 7, or 9. Digital roots of square numbers progress in the sequence 1, 4, 9, 7, 7, 9, 4, 1, 9.

The digital root of a perfect cube is 1, 8 or 9, and digital roots of perfect cubes progress in that exact sequence.

The digital root of a prime number (except 3) is 1, 2, 4, 5, 7, or 8.

The digital root of a power of 2 is 1, 2, 4, 5, 7, or 8. Digital roots of the powers of 2 progress in the sequence 1, 2, 4, 8, 7, 5. This even applies to negative powers of 2; for example, 2 to the power of 0 is 1; 2 to the power of -1 (minus one) is .5, with a digital root of 5; 2 to the power of -2 is .25, with a digital root of 7; and so on, ad infinitum in both directions. This is because negative powers of 2 share the same digits (after removing leading zeroes) as corresponding positive powers of 5, whose digital roots progress in the sequence 1, 5, 7, 8, 4, 2.

The digital root of a power of 5 is 1, 2, 4, 5, 7 or 8. Digital roots of the powers of 5 progress in the sequence 1, 5, 7, 8, 4, 2. This even applies to negative powers of 5; for example, 5 to the power of 0 is 1; 5 to the power of -1 (minus one) is .2, with a digital root of 2; 5 to the power of -2 is .04, with a digital root of 4; and so on, ad infinitum in both directions. This is because the negative powers of 5 share the same digits (after removing leading zeroes) as corresponding positive powers of 2, whose digital roots progress in sequence 1, 2, 4, 8, 7, 5.

The digital roots of powered numbers progress in sequence (only certain for positive powers, although in for some exceptions it also may occur for negative powers), and this is because of one of the previously shown properties. As the digital root of a b is congruent with the multiple of the digital root of a and the digital root of b modulo 9, the digital root of a a will also do it. So, for example, as shown above, powers of 2 will follows the sequence 1, 2, 4, 8, 7, 5; Powers of 47 (whose digital root is 2) will also follow this sequence. The very sequence follows this rule, and is appliable to any other number.

{\displaystyle {\mathit {dr}}(a^{n})\equiv {\mathit {dr}}^{n}(a){\pmod {9}}.} {\mathit {dr}}(a^{n})\equiv {\mathit {dr}}^{n}(a){\pmod {9}}.

The digital root of an even perfect number (except 6) is 1.

The digital root of a centered hexagram, or star number is 1 or 4. Digital roots of star numbers progress in the sequence 1, 4, 1.

The digital root of a centered hexagonal number is 1 or 7, their digital roots progressing in the sequence 1, 7, 1.

The digital root of a triangular number is 1, 3, 6 or 9. Digital roots of triangular numbers progress in the sequence 1, 3, 6, 1, 6, 3, 1, 9, 9, which is palindromic after the first eight terms.

The digital root of Fibonacci numbers is a repeating pattern of 1, 1, 2, 3, 5, 8, 4, 3, 7, 1, 8, 9, 8, 8, 7, 6, 4, 1, 5, 6, 2, 8, 1, 9.

The digital root of Lucas numbers is a repeating pattern of 2, 1, 3, 4, 7, 2, 9, 2, 2, 4, 6, 1, 7, 8, 6, 5, 2, 7, 9, 7, 7, 5, 3, 8.

The digital root of the product of twin primes, other than 3 and 5, is 8. The digital root of the product of 3 and 5 (twin primes) is 6.

**357. Count Numbers with Unique Digits**

Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.

**Example:**  
Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])

解：超过10位的必定重复。

**343. Integer Break**：Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers.

I use a function to express this product: f=x(N-x). When x=N/2, we get the maximum of this function.

However, factors should be integers. Thus the maximum is (N/2)\*(N/2) when N is even or (N-1)/2 \*(N+1)/2 when N is odd.

When the maximum of f is larger than N, we should do the break.

(N/2)\*(N/2)>=N, then N>=4

(N-1)/2 \*(N+1)/2>=N, then N>=5

These two expressions mean that factors should be less than 4, otherwise we can do the break and get a better product. The factors in last result should be 1, 2 or 3. Obviously, 1 should be abandoned. Thus, the factors of the perfect product should be 2 or 3.

The reason why we should use 3 as many as possible is

For 6, 3 \* 3>2 \* 2 \* 2. Thus, the optimal product should contain no more than three 2.

O(N) solution.

**public** **class** Solution {

**public** int integerBreak(int n) {

**if**(n==2) **return** 1;

**if**(n==3) **return** 2;

int product = 1;

**while**(n>4){

product\*=3;

n-=3;

}

product\*=n;

**return** product;

}

}

### 四平方定理/三平方定理

常规方法：

for(i:1-n) for(j:1-i) diff = i-j\*j;？？

定理：

a natural number can be represented as the sum of three squares of integers n=x^{2}+y^{2}+z^{2}

《==》 n is not of the form n = 4^a(8b + 7) for integers a and b.

如果是这个形式，就是四平方和

注意x、y、z可以是0.

int numSquares(int n) {

//去除公式中的4//n%4 == 0

while ((n & 3) == 0) n >>= 2;

if ((n & 7) == 7) return 4; //n % 8 == 7

if(is\_square(n)) return 1;

int sqrt\_n = (int) sqrt(n);

for(int i = 1; i<= sqrt\_n; i++){

if (is\_square(n-i\*i)) return 2;

}

return 3;

}

**int** **is\_square**(**int** n){

**int** temp = (**int**) sqrt(n);

**return** temp \* temp == n;

}

## 运算符

### 乘

43. Multiply Strings: Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

### 除

29. Divide Two Integers：divide two integers without using multiplication, division and mod operator.

位操作，长除法->变大除数

### 取模运算

定义: 给定一个正整数p，任意一个整数n，一定存在等式 ：n = kp + r ；

其中 k、r 是整数，且 0 ≤ r < p，则称 k 为 n 除以 p 的商，r 为 n 除以 p 的余数。

对于正整数 p 和整数 a,b，定义如下运算：

取模运算：a % p（或a mod p），表示a除以p的余数。

模p加法： ，其结果是a+b算术和除以p的余数。

模p减法： ，其结果是a-b算术差除以p的余数。

模p乘法： ，其结果是 a \* b算术乘法除以p的余数。

说明：

1. 同余式：正整数a，b对p取模，余数相同，记做 或者a ≡ b (mod p)。

2. n % p 得到结果的正负由被除数n决定,与p无关。例如：7%4 = 3， -7%4 = -3， 7%-4 = 3， -7%-4 = -3。

基本性质

1. 若(a-b)%p = 0，则a≡b (% p)。例如 11 ≡ 4 (% 7)， 18 ≡ 4(% 7)
2. 对称性：a≡b (% p)等价于b≡a (% p)
3. 传递性：若a≡b (% p)且b≡c (% p) ，则a≡c (% p)

运算规则

模运算与基本四则运算有些相似，但是除法例外。其规则如下：

1. (a + b) % p = (a % p + b % p) % p （1）
2. (a - b) % p = (a % p - b % p) % p （2）
3. (a \* b) % p = (a % p \* b % p) % p （3）
4. a ^ b % p = ((a % p)^b) % p （4）

* 结合律：

((a+b) % p + c) % p = (a + (b+c) % p) % p （5）

((a\*b) % p \* c)% p = (a \* (b\*c) % p) % p （6）

* 分配律：

((a +b)% p \* c) % p = ((a \* c) % p + (b \* c) % p) % p （10）

重要定理

* 若a≡b (% p)，则对于任意的c，都有(a + c) ≡ (b + c) (%p)；（11）
* 若a≡b (% p)，则对于任意的c，都有(a \* c) ≡ (b \* c) (%p)；（12）
* 若a≡b (% p)，c≡d (% p)，则 (a + c) ≡ (b + d) (%p)，(a - c) ≡ (b - d) (%p)，

(a \* c) ≡ (b \* d) (%p)，(a / c) ≡ (b / d) (%p)； （13）

应用

**判别奇偶数**

已知一个整数n对2取模，如果[余数](https://baike.baidu.com/item/%E4%BD%99%E6%95%B0)为0，则表示n为偶数，否则n为奇数。

**判别素数**

一个数，如果只有1和它本身两个因数，这样的数叫做[质数](https://baike.baidu.com/item/%E8%B4%A8%E6%95%B0)（或[素数](https://baike.baidu.com/item/%E7%B4%A0%E6%95%B0)）。例如 2，3，5，7 是质数，而 4，6，8，9 则不是，后者称为合成数或[合数](https://baike.baidu.com/item/%E5%90%88%E6%95%B0)。

判断某个自然数是否是素数最常用的方法就是试除法——用比该自然数的平方根小的正整数去除这个自然数，若该自然数能被整除，则说明其非素数。

**求最大公约数**

求最大公约数最常见的方法是[欧几里德算法](https://baike.baidu.com/item/%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%B7%E7%AE%97%E6%B3%95)（又称辗转相除法），其计算原理依赖于定理：gcd(a,b) = gcd(b,a mod b)

证明：

a可以表示成a = kb + r，则r = a mod b

假设d是a,b的一个公约数，则有d|a, d|b，而r = a - kb，因此d|r

因此d是(b,a mod b)的公约数

假设d 是(b,a mod b)的公约数，则d | b , d |r ，但是a = kb +r

因此d也是(a,b)的公约数

因此(a,b)和(b,a mod b)的公约数是一样的，其最大公约数也必然相等，得证。

**水仙花数**

水仙花数是指一个 n 位正整数，每个位上的数字的 n 次幂之和等于它本身。（例如：1^3 + 5^3+ 3^3 = 153）。

水仙花数只是[自幂数](https://baike.baidu.com/item/%E8%87%AA%E5%B9%82%E6%95%B0)的一种，严格来说三位数的3次幂数才成为水仙花数。

附：其他位数的自幂数名字

一位自幂数：独身数

两位自幂数：没有

三位自幂数：水仙花数

。。。

十位自幂数：十全十美数

**模幂运算**

想知道3333^5555的末位是什么。不可能直接把3333^5555的结果计算出来，太大了。但我们想要确定的是3333^5555（%10）。

根据运算规则a^b % p = ((a % p)^b) % p ， 3333^5555（%10）= 3^5555（%10）。

根据运算规则 (a \* b) % p = (a % p \* b % p) % p ，由于5555 = 4 \* 1388 + 3，得到

3^5555（%10）=（3^(4\*1388) \* 3^3）（%10）

=（（3^(4\*1388)（%10）\* 3^3（%10））（%10）

=（（3^(4\*1388)（%10）\* 7）（%10）。

根据欧拉定理可以得到 3 ^ (4 \* k) % 10 = 1, 所以（（3^(4\*1388)（%10）\* 7）（%10）= (1 \* 7) (% 10) = 7

利用这些规则可以有效地计算X^N(% P)。简单的算法是将result初始化为1，然后重复将result乘以X，每次[乘法](https://baike.baidu.com/item/%E4%B9%98%E6%B3%95)之后应用%运算符（这样使得result的值变小，以免溢出），执行N次相乘后，result就是我们要找的答案。

这样对于较小的N值来说，实现是合理的，但是当N的值很大时，需要计算很长时间，是不切实际的。

下面的结论可以得到一种更好的算法。

如果N是偶数，那么X^N =（X\*X）^[N/2]；

如果N是奇数，那么X^N = X\*X^(N-1) = X \*（X\*X）^[N/2]；

其中[N]是指小于或等于N的最大整数。

C++实现功能函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | /\*函数功能：利用模运算规则，采用递归方式，计算X^N(%P)函数名：PowerMod输入值：unsigned int x，底数x unsigned int n，指数nunsigned int p，模p返回值：unsigned int，X^N(%P)的结果\*/  unsignedintPowerMod(unsignedintx,unsignedintn,unsignedintp)  {  if(n==0)  {  return1;  }  unsignedinttemp=PowerMod((x\*x)%p,n/2,p);//递归计算（X\*X）^[N/2]  if((n&1)!=0)//判断n的奇偶性  {  temp=(temp\*x)%p;  }  returntemp;  } |

《孙子问题(中国剩余定理)》

《[孙子算经](https://baike.baidu.com/item/%E5%AD%99%E5%AD%90%E7%AE%97%E7%BB%8F)》中有这样一个问题：

“一个数除以3余2，除以5余3，除以7余2.求适合这个条件的最小数。”

我国古代学者早就研究过这个问题。例如我国明朝数学家[程大位](https://baike.baidu.com/item/%E7%A8%8B%E5%A4%A7%E4%BD%8D)在他著的《算法统宗》（1593年）中就用四句很通俗的口诀暗示了此题的解法：

三人同行七十稀，五树梅花廿一枝，七子团圆正半月，除百零五便得知。

"正半月"暗指15。"除百零五"的原意是，当所得的数比105大时，就105、105地往下减，使之小于105；这相当于用105去除，求出余数。

这四句口诀暗示的意思是：当除数分别是3、5、7时，用70乘以用3除的余数，用21乘以用5除的余数，用15乘以用7除的余数，然后把这三个乘积相加。加得的结果如果比105大，就除以105，所得的余数就是满足题目要求的最小正整数解。

根据剩余定理，我把此种解法推广到有n(n为自然数）个除数对应n个余数，求最小被除数的情况。输入n个除数（除数不能互相整除）和对应的余数，计算机将输出最小被除数。

### 幂

326. Power of Three: Given an integer, write a function to determine if it is a power of three.

解：3^n%3^x=0,x<n-

// 1162261467 is 3^19, 3^20 is bigger than intreturn ( n>0 && 1162261467%n==0);

372. Super Pow：Your task is to calculate *ab* mod 1337 where *a* is a positive integer and *b* is an extremely large positive integer given in the form of an array.

## 公约数

微软 365. Water and Jug Problem

You are given two jugs with capacities *x* and *y* litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly *z* litres using these two jugs.

If *z* liters of water is measurable, you must have *z* liters of water contained within **one or both buckets** by the end.

Operations allowed:

* Fill any of the jugs completely with water.
* Empty any of the jugs.
* Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

**The basic idea is to use the property of Bézout's identity and check if z is a multiple of GCD(x, y)**

Quote from wiki:

Bézout's identity (also called Bézout's lemma) is a theorem in the elementary theory of numbers:

let a and b be nonzero integers and let d be their greatest common divisor. Then there exist integers x and y such that ax+by=d

In addition, the greatest common divisor d is the smallest positive integer that can be written as ax + by

every integer of the form ax + by is a multiple of the greatest common divisor d.

If a or b is negative this means we are emptying a jug of x or y gallons respectively.

Similarly if a or b is positive this means we are filling a jug of x or y gallons respectively.

x = 4, y = 6, z = 8.

GCD(4, 6) = 2

8 is multiple of 2

so this input is valid and we have:

-1 \* 4 + 6 \* 2 = 8

In this case, there is a solution obtained by filling the 6 gallon jug twice and emptying the 4 gallon jug once. (Solution. Fill the 6 gallon jug and empty 4 gallons to the 4 gallon jug. Empty the 4 gallon jug. Now empty the remaining two gallons from the 6 gallon jug to the 4 gallon jug. Next refill the 6 gallon jug. This gives 8 gallons in the end)

See wiki:

[Bézout's identity](https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity)

**public** **boolean** **canMeasureWater**(**int** x, **int** y, **int** z) {

//limit brought by the statement that water is finallly in one or both buckets

**if**(x + y < z) **return** **false**;

//case x or y is zero

**if**( x == z || y == z || x + y == z ) **return** **true**;

//get GCD, then we can use the property of Bézout's identity

**return** z%GCD(x, y) == 0;

}

## 排列组合

### Catalan

* *Cn* is the number of [Dyck words](https://en.wikipedia.org/wiki/Dyck_word)[[2]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-2) of length 2*n*. A Dyck word is a [string](https://en.wikipedia.org/wiki/String_(computer_science)) consisting of *n* X's and *n* Y's such that no initial segment of the string has more Y's than X's. For example, the following are the Dyck words of length 6:

XXXYYY     XYXXYY     XYXYXY     XXYYXY     XXYXYY.

* Re-interpreting the symbol X as an open [parenthesis](https://en.wikipedia.org/wiki/Bracket#Parentheses) and Y as a close parenthesis, *Cn* counts the number of expressions containing *n* pairs of parentheses:

((()))     ()(())     ()()()     (())()     (()())

* *Cn* is the number of different ways *n* + 1 factors can be completely [parenthesized](https://en.wikipedia.org/wiki/Bracket) (or the number of ways of [associating](https://en.wikipedia.org/wiki/Associativity) *n* applications of a [binary operator](https://en.wikipedia.org/wiki/Binary_operator)). For *n* = 3, for example, we have the following five different parenthesizations of four factors:

((ab)c)d     (a(bc))d     (ab)(cd)     a((bc)d)     a(b(cd))

[](https://en.wikipedia.org/wiki/File:Tamari_lattice,_trees.svg)

The [associahedron](https://en.wikipedia.org/wiki/Associahedron) of order 4 with the C4=14 full binary trees with 5 leaves

* Successive applications of a binary operator can be represented in terms of a full [binary tree](https://en.wikipedia.org/wiki/Binary_tree). (A rooted binary tree is *full* if every vertex has either two children or no children.) It follows that *Cn* is the number of full binary [trees](https://en.wikipedia.org/wiki/Tree_(graph_theory)) with *n* + 1 leaves:

[](https://en.wikipedia.org/wiki/File:Catalan_number_binary_tree_example.png)

* *Cn* is the number of non-isomorphic ordered trees with *n* vertices. (An ordered tree is a rooted tree in which the children of each vertex are given a fixed left-to-right order.)[[3]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-3)
* *Cn* is the number of monotonic [lattice paths](https://en.wikipedia.org/wiki/Lattice_path) along the edges of a grid with *n* × *n* square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for "move right" and Y stands for "move up".

[](https://en.wikipedia.org/wiki/File:Catalan_number_4x4_grid_example.svg)

This can be succinctly represented by listing the Catalan elements by column height:[[4]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-4)

[0,0,0,0][0,0,0,1][0,0,0,2][0,0,1,1]

[0,1,1,1] [0,0,1,2] [0,0,0,3] [0,1,1,2][0,0,2,2][0,0,1,3]

[0,0,2,3][0,1,1,3] [0,1,2,2][0,1,2,3]

[](https://en.wikipedia.org/wiki/File:Tamari_lattice,_hexagons.svg)

The triangles correspond to internal nodes of the binary trees.

* A [convex polygon](https://en.wikipedia.org/wiki/Convex_polygon) with *n* + 2 sides can be cut into [triangles](https://en.wikipedia.org/wiki/Triangle) by connecting vertices with non-crossing [line segments](https://en.wikipedia.org/wiki/Line_segment) (a form of [polygon triangulation](https://en.wikipedia.org/wiki/Polygon_triangulation)). The number of triangles formed is *n* and the number of different ways that this can be achieved is *Cn*. The following hexagons illustrate the case *n* = 4:

[](https://en.wikipedia.org/wiki/File:Catalan-Hexagons-example.svg)

* *Cn* is the number of [stack](https://en.wikipedia.org/wiki/Stack_(data_structure))-sortable [permutations](https://en.wikipedia.org/wiki/Permutation) of {1, ..., *n*}. A permutation *w* is called [stack-sortable](https://en.wikipedia.org/wiki/Stack-sortable_permutation) if *S*(*w*) = (1, ..., *n*), where *S*(*w*) is defined recursively as follows: write *w* = *unv* where *n* is the largest element in *w* and *u* and *v* are shorter sequences, and set *S*(*w*) = *S*(*u*)*S*(*v*)*n*, with *S* being the identity for one-element sequences.
* *Cn* is the number of permutations of {1, ..., *n*} that avoid the [permutation pattern](https://en.wikipedia.org/wiki/Permutation_pattern) 123 (or, alternatively, any of the other patterns of length 3); that is, the number of permutations with no three-term increasing subsequence. For *n* = 3, these permutations are 132, 213, 231, 312 and 321. For *n* = 4, they are 1432, 2143, 2413, 2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.
* *Cn* is the number of [noncrossing partitions](https://en.wikipedia.org/wiki/Noncrossing_partition) of the set {1, ..., *n*}. [*A fortiori*](https://en.wikipedia.org/wiki/A_fortiori_argument), *Cn* never exceeds the *n*th [Bell number](https://en.wikipedia.org/wiki/Bell_number). *Cn* is also the number of noncrossing partitions of the set {1, ..., 2*n*} in which every block is of size 2. The conjunction of these two facts may be used in a proof by [mathematical induction](https://en.wikipedia.org/wiki/Mathematical_induction) that all of the *free* [cumulants](https://en.wikipedia.org/wiki/Cumulant)of degree more than 2 of the [Wigner semicircle law](https://en.wikipedia.org/wiki/Wigner_semicircle_law) are zero. This law is important in [free probability](https://en.wikipedia.org/wiki/Free_probability) theory and the theory of [random matrices](https://en.wikipedia.org/wiki/Random_matrices).
* *Cn* is the number of ways to tile a stairstep shape of height *n* with *n* rectangles. The following figure illustrates the case *n* = 4:

[](https://en.wikipedia.org/wiki/File:Catalan_stairsteps_4.svg)

* *Cn* is the number of rooted [binary trees](https://en.wikipedia.org/wiki/Binary_tree) with *n* internal nodes (*n* + 1 [leaves](https://en.wikipedia.org/wiki/Tree_(graph_theory)#Definitions) or external nodes). Illustrated in following Figure are the trees corresponding to *n* = 0,1,2 and 3. There are 1, 1, 2, and 5 respectively. Here, we consider as binary trees those in which each node has zero or two children, and the internal nodes are those that have children.

[](https://en.wikipedia.org/wiki/File:Binary_Tree.png)

* *Cn* is the number of ways to form a "mountain range" with *n* upstrokes and *n* downstrokes that all stay above a horizontal line. The mountain range interpretation is that the mountains will never go below the horizon.

[](https://en.wikipedia.org/wiki/File:Mountain_Ranges.png)

* *Cn* is the number of [standard Young tableaux](https://en.wikipedia.org/wiki/Young_tableau#Tableaux) whose diagram is a 2-by-*n* rectangle. In other words, it is the number of ways the numbers 1, 2, ..., 2*n* can be arranged in a 2-by-*n* rectangle so that each row and each column is increasing. As such, the formula can be derived as a special case of the [hook-length formula](https://en.wikipedia.org/wiki/Young_tableau#Dimension_of_a_representation).
* *Cn* is the number of ways that the vertices of a convex 2*n*-gon can be paired so that the line segments joining paired vertices do not intersect. This is precisely the condition that guarantees that the paired edges can be identified (sewn together) to form a closed surface of genus zero (a topological 2-sphere).
* *Cn* is the number of [semiorders](https://en.wikipedia.org/wiki/Semiorder) on *n* unlabeled items.[[5]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-5)

## 中位数

**462. Minimum Moves to Equal Array Elements II**

Given a **non-empty** integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

算法导论：快排找中位数

**4. Median of Two Sorted Arrays**

解：如果两个数组中位数相等，则直接就是该数。注意考虑数目奇偶。

否则，可以两个数组分别剔除相同数量的数，可以缩小规模。

## 概率

808. Soup Servings

There are two types of soup: type A and type B. Initially we have N ml of each type of soup. There are four kinds of operations:

1. Serve 100 ml of soup A and 0 ml of soup B
2. Serve 75 ml of soup A and 25 ml of soup B
3. Serve 50 ml of soup A and 50 ml of soup B
4. Serve 25 ml of soup A and 75 ml of soup B

When we serve some soup, we give it to someone and we no longer have it.  Each turn, we will choose from the four operations with equal probability 0.25. If the remaining volume of soup is not enough to complete the operation, we will serve as much as we can.  We stop once we no longer have some quantity of both types of soup.

Note that we do not have the operation where all 100 ml's of soup B are used first.

Return the probability that soup A will be empty first, plus half the probability that A and B become empty at the same time.

**Example:**

**Input:** N = 50

**Output:** 0.625

**Explanation:**

If we choose the first two operations, A will become empty first. For the third operation, A and B will become empty at the same time. For the fourth operation, B will become empty first. So the total probability of A becoming empty first plus half the probability that A and B become empty at the same time, is 0.25 \* (1 + 1 + 0.5 + 0) = 0.625.

**Notes:**

* 0 <= N <= 10^9.
* Answers within 10^-6 of the true value will be accepted as correct.

比较小的范围，用dp

无限大时，答案固定某个值

编4.1 金刚坐飞机

如果金刚坐在第n个位置，那么第i个乘客坐在自己位置的概率为f(n)

注意有个隐含条件金刚的机票座位号是1

## 拾遗

**453. Minimum Moves to Equal Array Elements**

Given a **non-empty** integer array of size *n*, find the minimum number of moves required to make all array elements equal, where a move is incrementing *n* - 1 elements by 1.

解：let's define sum as the sum of all the numbers, before any moves; minNum as the min number int the list; n is the length of the list;

After, say m moves, we get all the numbers as x , and we will get the following equation

sum + m \* (n - 1) = x \* n

and actually,

**x** = minNum + m

it comes from two observations:

1. the minum number will always be minum until it reachs the final number, because every move, other numbers (besides the max) will be increamented too;
2. from above, the minum number will be incremented in every move. So, if the final number is x, it would be minNum + moves;

and finally, we will get

sum - minNum \* n = m

319. Bulb Switcher

倍数、平方数；反向思维，一个数在哪些轮会被按到

672. Bulb Switcher II

There is a room with n lights which are turned on initially and 4 buttons on the wall. After performing exactly m unknown operations towards buttons, you need to return how many different kinds of status of the n lights could be.

Suppose n lights are labeled as number [1, 2, 3 ..., n], function of these 4 buttons are given below:

1. Flip all the lights.
2. Flip lights with even numbers.
3. Flip lights with odd numbers.
4. Flip lights with (3k + 1) numbers, k = 0, 1, 2, ...

解：后面灯和前面是等价的。答案空间很小，无限循环。

754. Reach a Number

You are standing at position 0 on an infinite number line. There is a goal at position target.

On each move, you can either go left or right. During the *n*-th move (starting from 1), you take *n* steps.

Return the minimum number of steps required to reach the destination.

**Intuition**

The crux is to put + and - signs on the numbers 1, 2, 3, ..., k so that the sum is target.

When target < 0 and we made a sum of target, we could switch the signs of all the numbers so that it equals Math.abs(target). Thus, the answer for target is the same as Math.abs(target), and so, we can consider only target > 0.

Now let's say k is the smallest number with S = 1 + 2 + ... + k >= target. If S == target, the answer is k.

If S > target, we need to change some number signs. If delta = S - target is even, then we can always find a subset of {1, 2, ..., k} equal to delta / 2 and switch the signs, so the answer is k. (This depends on T = delta / 2 being at most S.) [The proof is simple: either T <= k and we choose it, or we choose k in our subset and try to solve the same instance of the problem for T -= k and the set {1, 2, ..., k-1}.]

Otherwise, if delta is odd, we can't do it, as every sign change from positive to negative changes the sum by an even number. So let's consider a candidate answer of k+1, which changes delta by k+1. If this is odd, then delta will be even and we can have an answer of k+1. Otherwise, delta will be odd, and we will have an answer of k+2.

For concrete examples of the above four cases, consider the following:

* If target = 3, then k = 2, delta = 0 and the answer is k = 2.
* If target = 4, then k = 3, delta = 2, delta is even and the answer is k = 3.
* If target = 7, then k = 4, delta = 3, delta is odd and adding k+1 makes delta even. The answer is k+1 = 5.
* If target = 5, then k = 3, delta = 1, delta is odd and adding k+1 keeps delta odd. The answer is k+2 = 5.

**Algorithm**

Subtract ++k from target until it goes non-positive. Then k will be as described, and target will be delta as described. We can output the four cases above: if delta is even then the answer is k, if delta is odd then the answer is k+1 or k+2 depending on the parity of k.

**Complexity Analysis**

* Time Complexity:  *O*(√​target​​​). Our while loop needs this many steps, as 1+2+⋯+k=k(k+1)21+2+⋯+k=k(k+1)2.
* Space Complexity: *O*(1).

# 系统设计

**编1.10 双线程高效下载**

信号量

**284. Peeking Iterator**

封装一层，组合模式，提前获取next并缓存

**355. Design Twitter**

Design a simplified Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.
2. **getNewsFeed(userId)**: Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
3. **follow(followerId, followeeId)**: Follower follows a followee.
4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

参考

**private** **static** **int** timeStamp=0;

// easy to find if user exist

**private** Map<Integer, User> userMap;

// Tweet link to next Tweet so that we can save a lot of time

// when we execute getNewsFeed(userId)

**private** **class** **Tweet**{

**public** **int** id;

**public** **int** time;

**public** Tweet next;

**public** **Tweet**(**int** id){

**this**.id = id;

time = timeStamp++;

next=null;

}

}

// OO design so User can follow, unfollow and post itself

**public** **class** **User**{

**public** **int** id;

**public** Set<Integer> followed;

**public** Tweet tweet\_head;

**public** **User**(**int** id){

**this**.id=id;

followed = **new** HashSet<>();

follow(id); // first follow itself

tweet\_head = null;

}

**public** **void** **follow**(**int** id){

followed.**add**(id);

}

**public** **void** **unfollow**(**int** id){

followed.**remove**(id);

}

// everytime user post a new tweet, add it to the head of tweet list.

**public** **void** **post**(**int** id){

Tweet t = **new** Tweet(id);

t.next=tweet\_head;

tweet\_head=t;

}

}

/\*\* Initialize your data structure here. \*/

**public** **Twitter**() {

userMap = **new** HashMap<Integer, User>();

}

/\*\* Compose a new tweet. \*/

**public** **void** **postTweet**(**int** userId, **int** tweetId) {

**if**(!userMap.containsKey(userId)){

User u = **new** User(userId);

userMap.put(userId, u);

}

userMap.**get**(userId).post(tweetId);

}

// Best part of this.

// first get all tweets lists from one user including itself and all people it followed.

// Second add all heads into a max heap. Every time we poll a tweet with

// largest time stamp from the heap, then we add its next tweet into the heap.

// So after adding all heads we only need to add 9 tweets at most into this

// heap before we get the 10 most recent tweet.

**public** List<Integer> **getNewsFeed**(**int** userId) {

List<Integer> res = **new** LinkedList<>();

**if**(!userMap.containsKey(userId)) **return** res;

Set<Integer> users = userMap.**get**(userId).followed;

PriorityQueue<Tweet> q = **new** PriorityQueue<Tweet>(users.size(), (a,b)->(b.time-a.time));

**for**(**int** user: users){

Tweet t = userMap.**get**(user).tweet\_head;

// very imporant! If we add null to the head we are screwed.

**if**(t!=null){

q.**add**(t);

}

}

**int** n=0;

**while**(!q.isEmpty() && n<10){

Tweet t = q.poll();

res.**add**(t.id);

n++;

**if**(t.next!=null)

q.**add**(t.next);

}

**return** res;

}

/\*\* Follower follows a followee. If the operation is invalid, it should be a no-op. \*/

**public** **void** **follow**(**int** followerId, **int** followeeId) {

**if**(!userMap.containsKey(followerId)){

User u = **new** User(followerId);

userMap.put(followerId, u);

}

**if**(!userMap.containsKey(followeeId)){

User u = **new** User(followeeId);

userMap.put(followeeId, u);

}

userMap.**get**(followerId).follow(followeeId);

}

/\*\* Follower unfollows a followee. If the operation is invalid, it should be a no-op. \*/

**public** **void** **unfollow**(**int** followerId, **int** followeeId) {

**if**(!userMap.containsKey(followerId) || followerId==followeeId)

**return**;

userMap.**get**(followerId).unfollow(followeeId);

}

# 纯数学基础

一元二次方程

x=[-b±√(b²-4ac)]/2a

递推等式求通项

已知A1和A2，形如aA(n+2)+bA(n+1)+cA(n)=0的数列，[特征方程](https://www.baidu.com/s?wd=%E7%89%B9%E5%BE%81%E6%96%B9%E7%A8%8B&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)为ax^2+bx+c=0,求出两根为x1,x2。那么  
数列[通项公式](https://www.baidu.com/s?wd=%E9%80%9A%E9%A1%B9%E5%85%AC%E5%BC%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)为A(n)=M x1^n+N x2^n,M N为待定系数，由已知的A1 A2代入[通项公式](https://www.baidu.com/s?wd=%E9%80%9A%E9%A1%B9%E5%85%AC%E5%BC%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)求出。

海伦公式

p是半周长

s = p(p-a)(p-b)(p-c)

# 其他基础

float

float的尾数：23位，其范围为：0~223，而223=8388608=106.92，所以float的精度为6~7位，能保证6位为绝对精确，7位一般也是正确的，8位就不一定了（但不是说8位就绝对不对了），

注意这里的6~7位是有效小数位（大的数你先需要转换成小数的指数形式，例如：8317637.5，其有效小数位：8.3176375E6，七位），而有效位（从第一个不为0的开始数）是7~8位，是包括整数位的，像8317637.5，你不转换，则要从有效位的角度来看，有8位有效位。 

System.out.println((float)Math.pow(10,6.92));//注意加float强制转换

//打印结果8317637.5，float只保证7~8位有效位，其余位数舍入

　　可以再这样想：23位，二进制0101……0101，尾数表示小数位，最小为0000……0001（22个0，最后一个1），即2−23=1.1920929E-7 ，这是float的最小单元（大概是0.0000001192大小，你想表示比这更小的，比如0.00000001，不可能啊），这是一个7位小数位小数，最小就是这么小，比这个更小的，计算机就无能为力了，比这个更大的，每次通过加这么一个最小单元，直到相等或接近（两个相差一个最小单元的数，它们之间的数也是不能表示的，所以有的7位也是不能精确的，因为最小不是0.0000001，而是比这个稍大）。

double

　　double的尾数：52位，2−52=2.220446049250313E-16，最小是16位，但最小不是1.0E-16，所以精度是15~16，能保证15，一般16位。

# ms拾遗

**LeetCode 774. Minimize Max Distance to Gas Station**

**给定一组加油站的位置stations，在这些加油站中新增K个加油站，使得相邻加油站之间的最大距离最小化。求最小化的最大距离。**

解1：每次在最大区间插入一个点，使之变小。注意每次插入不是均分。

解2：搜索目标距离，计算每个区间达到最小距离需要多少个点。

**75 寻找峰值**

二分，根据坡度判断舍弃一半。

**Leetcode 101 Symetric Tree**

递归：转化为子树对称问题；迭代：根中序遍历，左边左孩子优先，右边右孩子优先。

**lintcode 1197. Find Bottom Left Tree Value**

解：层次遍历，记录每层第一个。最后一层第一个就是要求的。

**lintcode 986. Battleships in a Board：**给一个举证，长条形军舰横着或者竖着放，统计数目。

解: 扫描每一行，如果军舰在这一行之上，则计数；否则下一行再看。

易错 Lintcode 167. 链表求和

1. 注意循环的推进 2. 注意最后的进位，有可能要增加新的节点

**编程之美的题目**

# 未整理

# End