# 解题思路

## 1. 题型篇

### 1.1 Array类型

#### 1.1.2 数组值与数组索引

根据数组元素值，找到该值所在的索引下标对应的值，然后进行的相关操作，可以做到array类型题目中的one-pass解决问题。例题： 189、41、287、

也可以使用双指针的方法（同向快慢指针，同向窗口指针，相向指针），实现one-pass；

#### 1.1.3 数组排序

很多数组类型的题目，可以通过先排序数组，在寻找解决方案 。O(nlogn) + O(n)

当数组中元素取值范围一定且不不是很大时，可以通过线段树来优化，排序时间：O(k) + O(n)，查询时间： O(logn)

样题： LintCode Count of Smaller Number

#### 1.1.4 前缀和数组

前缀和数组（Prefix Sum Array）方便解决区间内的求和等操作，适合读请求，对写请求不是很友好，需要重新创建前缀和数组；

前缀和数组特点：由于是累加操作，你会发现每个索引点上的值可以看作为原来元素值的按比例分配，例如： 1， 3， 4 【原数组】=》 1，4， 8 【前缀和数组】

原素组中元素按比例分配权重分别是1/8、3/8、4/8，可以生成随机数取模总和8，得出的点按照大小分布，各归于原数组索引： 0归属于索引0, 1/2/3归属于索引1, 4/5/6/7，归属于索引2；

随机数取模之后得到的值a，按二分搜索在前缀和数组中搜索a+1，搜索找到的点的索引值，就对应着原数组中的索引；

样题： 528. Random Pick with Weight

前缀和数组的改进：

* + 树状数组 Binary Indexed Tree
    - 前缀和数组对写更新操作支持的不好，需要重新创建整个前缀和数组；当有写更新操作时，则使用树状数组；缺点是query操作时间复杂度从O(1)降为O(logn)
  + 区间树 Segment Tree
    - 区间数可以解决所有区间问题，比前缀和/树状数组更强大，但是好用比较大的空间，叶子节点如果为n，则空间复杂度为O(4n)

#### 1.1.5 循环数组

1. 分裂
2. 重复
3. 逆向

#### 1.1.6 数组题型注意点

* 元素：正负0的取值
  + 边缘case
* 元素：取值范围
  + 取值范围不大可以考虑线段树
* 元素：是否重复出现
  + 边缘case
* 数组：是否有序
  + 可以排序/利用有序来解题
* 数组：数组长度范围
* 数组：是否可以移动数组元素
  + 利用数组索引来解题

### 1.2 List类型

解题思路1：

设置 dummyRoot 作为头结点，可以简化代码复杂度

解题思路2：

One Pass 解题思路，使用双指针：快慢指针。

### 1.3 String类型

1. 对于字符串中的字符，可以将其看做为ascii表中对应的数值来看待，减去对应的首字符可以转换成相应的int值，可以做为索引以及其他有意义的值
2. 可以使用ascii值作为hash值创建hash表（数组实现方式）；
3. 也可以使用Tire Tree来解决一些字符遍历，求前缀相关的问题；Trie Tree也是对字符串哈希表的一种空间上的优化；

### 1.4 Bytemask类型

#### 1.4.1 比特移位

* arithmetic shift = signed shift: >> 如果是负数，则补1
* logical shift = unsigned shift: >>> 如果是负数，则补0

**防止在移位的时候被坑，尽量使用向左移位。**

#### 1.4.2 比特移位技巧

* A|B = A&B + A^B
* n & -n => 得到最后一个1的位置
* n & n-1 => 消除最后一个1
* n ^ n 归零

样题：判断一个数是不是2的幂；判断一个数中1的个数；寻找出现奇数次的数字；两个数交换，但不引入第三变量；

#### 1.4.3 数字电路连通图

AB => A & B

A' => ~A

AB' + A'B = A⊕B => A ^ B

A⊕B + AB => A ^ B + A & B => A | B

样题： 137 Single number II

#### 1.4.4 位运算做加减法

* 两个数字求和

异或：11得0，00得0，10和01得1；相异或的结果和相加是一致的；

相与 ：11得1，00得0，10和01得0； 得到的结果和进位是一致的；

所以 a + b = a ^ b + （（a & b）<< 1）

* 两个数字相减

a – b = a + (-b)，

一个正数b，取反操作 ~b，

得到结果为 ~b = -b – 1，

推导得： -b = ~b + 1

所以 a – b = a + ~b + 1

#### 1.4.5 数字的二分分解

一个大小为N的数字，可以被分解为logN个数字，这些数字的全部组合就是对n的遍历；适用于情景：某物体可以被取最多n次情况下，求解一些问题；我们可以把物体分解成logN个物体；将问题演变成：对这些logN物体取或者不取的情况下，求解一些问题；

## 2. 数据结构和算法篇

### 2.1 线性数据结构

#### 2.1.1 Stack

主要用于暂存有用信息。

单调栈应用有特殊性，当寻找每一个当前元素左边/右边第一个比它自身小/大的元素时，就是用单调栈来暂存有效信息。

样题：456 132Pattern [Monotonous Stack]

84 Largest Rectangle in Histogram [Monotonous Stack]

85 Max Rectangle [Monotonous Stack]

654 Maximum Binary Tree [Monotonous Stack]

LintCode 12 Min Stack

155 Min Stack

716 Max Stack

232 Implement Queue using Stacks

LintCode 40 Implement Queue by Tow Stacks

394 Decode String

341 Flatten Nested List Iterator

#### 2.2.2 Deque

单调Deque特殊性，类似单调stack，寻找当前元素左边/右边比自身小/大的元素，当时同时该临时暂存的数据还有可能变动，例如滑动窗口情况下，有些数据会被踢出去，如果该数据在当前暂存的集合中，单调栈是无法快速有效剔除该元素的，正好适合单调队列。

样题： 239 Sliding Window Maximum

滑动窗口 + Heap， 则为O(nlogK)复杂度，继续优化？只能O(n)，线性遍历所有数据one pass情况，单调栈不适合，没法pop出栈底元素，故适用单调Deque。

### 2.3 二分法 Binary Search

#### 2.3.1 一维二分

出现数组进行排序，或者出现已经排序的数组。

样题： 162 Find Peak Element I

#### 2.3.2 二维二分

保留答案的一半，当可以肯定我们要的答案肯定在某一半，或者不在某一半的时候，我们就可以进行二维的二分。

样题： LintCode Find Peak Element II

#### 2.3.3 值域二分

往往没有给你一个数组让你二分，但同样是找到满足某个条件的最大或者最小值。通过猜值判断是否满足题意不对去搜索可能解

1. 找到可行解范围(可行解范围指的是所求结果的范围)

有的很容易看出这个范围，例如Sqrt(x)。有时候不容易看出来，可能需要枚举所有结果，从而发现二分性，例如wood cut

1. 猜答案并且遍历所有答案，找二分性

需要定位head、tail，尤其是tail，可以提前检测并排除，如果不方便则放置于循环体中进行统一检测

1. 检验题目中的条件是否满足
2. 不满足调整搜索范围再次检验条件是否满足

样题： 69 Sqrt(x)

变种：sqrt求上界，如何操作？

首先，正常解法中为了防止mid \* mid越界，有两种方法:

* + 使用判断mid < x/mid
  + 使用long定义mid，x等所有值

一律采用第二种方法进行规避；当求上界时，完全改变一下取值逻辑即可；

LintCode Sqrt(x) II

278 First Bad Version

LintCode Wood Cut

644 Maximum Average Subarray II

LintCode Copy Books I/II

#### 2.3.4 注意事项

1. 涉及到double的相关二分代码，循环体需要写成：

double esp = 1e-5; //可以设置成比容错率低一级

while (start + esp < end) {…}， 同时start/end都直接等于middle即可；

1. **二分代码中的循环边界确定和取值方式的组合：**

**基本思路就是，明确求上下界，再明确是否可能溢出，以及溢出的方向性问题**

**1. 使用左闭合区间；**

**2. Floor：nums[middle] <= target, 存middle，移动left；否则移动right；循环退出后，保存的middle就是目标索引；**

**3. Ceiling：nums[middle] < target, 移动left；否则移动right；left/right重合导致循环退出后， left/right就是目标索引；**

样题： LeetCode 34 Find First and Last Position of Element in Sorted Array

**思路 1**: 求解target-1上界和target+1下界

* target - 1，找上界，小于等于target-1的最大索引，会左溢出；
* target + 1，找下界，大于等于target+1的最小索引，会右溢出；

**思路 2**：求解target的上界和下界

* 找上界的同时需要防止右溢出；
* 找下界的同事需要防止左溢出；

**代码组织**：求Floor，基本上对于小于等于target的middle，保存当前middle，移动left；大于target的middle，移动right；循环退出后之前保存的middle就是目标索引；

求Ceiling，基本上对于小于target的middle，移动left；大于等于target的middle，移动right，重合循环退出后，left/right即为目标索引；

1. 如果不好提前处理head/tail，因为该尾部值可能是满足情况之一，还需要区分同样满足情况下的最优解，那么可以将尾部扩大一个单元（注意扩大一个单元后的有效性），从而实现左闭合区间的head->tail。（例如Copy Books I）

### 2.4 树 Tree

满二叉树（Full Tree）， 完全二叉树（Complete Tree）

二叉查找树（BST），字典树（Trie Tree），树状数组（Binary Indexed Tree, BIT）, 线段树（Segment Tree），堆（Heap）

二叉树的遍历，递归、栈、Morris

#### 2.4.1 堆 Heap

当遇到求解结合中的最大值，最小值，就应该想到适用堆（Heap）

主要实现有2种，PriorityQueue和TreeSet

##### Priority Queue：

* offer: O(logN)
* poll: O(logN)
* peek: O(1)
* remove: O(n)

remove操作时间复杂度高主要因为是需要遍历二叉树找到相应的数据

如果想解决remove操作耗时过大的问题，自己实现一个heap，同时辅助以HashMap来优化remove操作，可以使remove操作达到logN的复杂度；

##### Tree Set

* add : O(logN)
* pollFirst/pollLast: O(logN)
* first/last: O(logN)
* remove: O(logN)

**注意点**

* compare函数如果是第一个参数减去/compareTo第二个参数，那么就是一个升序排列，第一个元素是最小值，即为最小堆
* TreeSet是一个平衡二叉树（B+树），不能放入重复元素，如果遇到有重复元素，可以定义新的类，在类中引入id和val的概念，当val相同时候返回id的相差。Id的相差可以解读为题目中不同节点的先后顺序排列（即操作优先级），例如不同飞机的降落和起飞优先级；
* 注意val做数值比较时，如果值有正负值的时候，需要防止溢出
* 注意使用TreeSet时，引入的新类声明需要 class newClass implements Comparable<newClass>{…}
* 注意使用TreeSet时， 引入的新类中需要实现

public int compareTo (newClass other) {…} ，才能将该类放到TreeSet中使用。

样题：480 Sliding Window Median

295 Find Median from Data Stream

1. Smallest Number in Multiplication Table
2. Smallest Element in a BST
3. Kth Largest Element in an Array

378 Kth Smallest Element in a Sorted Matrix

LintCode 543 Kth largest in N Arrays

LintCode 465 Kth Smallest Sum in Two Sorted Arrays

小技巧： two sorted arrays => one sorted matrix

407 Trapping Water II

小技巧：矩阵有外围向内的遍历方式

#### 2.4.2 线段树 Segment Tree

##### 2.4.2.1 适用题型

使用数组的索引作为叶子节点，创建线段树，非叶子节点可以实现统计信息，例如counter，sum、乘积、最大值、最小值；

使用数组中元素值可能出现的范围作为叶子节点，在创建线段树的过程中，可以统计/比较数字的大小关系

要求5分钟写完模板

##### 应用技巧

数组索引作为区间窗口用来创建线段树，实现对不同区间内（可变窗口条件下）元素进行统计，寻求解题结果。

数值根据大小也可以用来创建线段树，从而实现了排序，但需要数据取值有边界。可以实现O(n)的排序，在排序情况下解题。适用条件：数组元素个数非常大，但是数据的取值上下限还可以接受。例如：数据取值范围为K（<1,000,000），则线段树的叶子节点个数为K，线段树总体空间为4K，线段树上查询、更新复杂度为logK。

数组排序+查找小于某值的元素个数，原来需要O(nlogn) + O(n)，可以采用：

* 创建一个最大范围的线段树，数据范围有边界 => SegmentTree.build()
* 将当前数组出现的值一一填入线段树中 => SegmentTree.modify()
* 用求和操作查询该线段树找到满足条件

即小于某值的元素个数 =》 SegmentTree.query()

复杂度优化为： O(K) + O(nlogK) + O(logK)，K为数组元素可能出现的值范围

（样题：Count of Smaller Number）

样题： LintCode

Segment Tree Query

Segment Tree Build II

Segment Tree Modify

Interval Minimum Number

Interval Sum

Interval Sum II

Count of Smaller Number

Count of Smaller Number before itself

315 Count of Smaller Number After Self

思考： 1. 如何将问题转换为区间问题？

2. 怎样把区间问题联想到线段树？

3. 怎么样分解问题为插入和查询两步。

#### 2.4.3 二叉索引树/树状数组 Binary Indexed Tree(Fenwick Tree)

##### 1. 树状数组 vs 前缀和数组

树状数组是前缀和数组 (Prefix Sum Array) 的增强版实现，在前缀和数组的基础上为了实现query和update之间的trade off，才引入树状数组

1. Prefix Sum Array Space: O(n) = Binary Indexed Tree Space: O(n)
2. Prefix Sum Array Update: O(n) > Binary Indexed Tree Update: O(logn)
3. Prefix Sum Array Query: O (1) > Binary Indexed Tree Update: O(logn)

感想：树状数组和前缀和数组，基本上功能是一致的，主要用于做数组内部区间的求和问题。但是基于这个基本功能，可以解决很多相关问题，例如数据统计功能，数据比较功能

##### 2. 树状数组 vs 线段树

树状数组十分容易实现，代码量小，时间复杂度低，并且经过数学处理后也可以实现成段更新。线段树 (Segment Tree) 也可以做到和树状数组一样的效果，但是代码要复杂得多。

1. Segment Tree Space: O(4n) > Binary Indexed Tree Space O(n)
2. Segment Tree Time O(logn) >= Binary Indexed Tree Time O(logn) ，实际操作复杂度树状数组小。
3. 一般情况下树状数组能解决的问题线段树都能解决，反之有些线段树能解决的问题树状数组却不行，例如区间求最大/最小值等。

##### 3. 应用技巧

* 线段树：枚举数据取值范围作为叶子节点，创建出线段树，非叶子节点存储额外信息（例如统计信息），遍历原数组更新相关节点，对每个数组元素都能求得统计信息；（要求5-10分钟写完模板）
* 树状数组：当数组元素取值没有范围，或者范围过大，使用树状数组替代线段树求解；（要求5-10分钟写完模板）

1. 将原数组制作一份拷贝，并对拷贝进行排序Arrays.sort();
2. 遍历原来数组元素，根据元素值通过Arrays.binarySearch()求得在拷贝中的索引值
3. 创建树状数组；将上一步返回的索引值，加上1作为树状数组的叶子节点；（等同于于线段树的叶子节点是枚举数据取值范围）
4. 对于每一个数组元素，都能求得统计信息；

样题： 315. Count of Smaller Numbers After Self

方法1：排序数组，再二分查找返回索引值，存储到新数组。BIT空间复杂度为O(n)，线段树空间复杂度为O(4n)

方法2：遍历数组，求的最小值；再遍历数组，将数组元素更新为nums[i]-min+1；再遍历数组求得当前数组的max，这样数据就散列成0-max，BIT空间复杂度为O(max+1) ，线段树空间复杂度为O( 4 \* max)

##### 4. 总结

需要注意和明确传入的外部索引值得的取值范围是从0开始还是从1开始，推荐外部函数传入的都是以外部函数所以为主；树状数组内部实现再转换成内部索引。

同时理解理解树状数组update函数对传入值的意义，是差值，而不是绝对值；如果是绝对值需要转换成差值；

#### 2.4.3 字典树 Trie Tree

##### 2.4.3.1 实现

样题：208 Implement Trie (Prefix Tree)

要求10分钟内写完 模板

用来解决前缀搜索等问题；本质上是对HashTable解题思路的优化，当且仅当HashTable使用String作为key的情况下；

1.与Hash的解题比较，两者的时间复杂度都为O(1)或者说O(n)，1是针对字符串级别的个数1，而n是针对字符串中的字符个数n。

2.在空间复杂度上，Trie树比Hash更优，因为字符串的前缀充分被利用，导致空间开销更少。

3.Hash实现能实现操作检测字符串是否存在；而Trie树除了这个操作之外还能额外实现检测前缀是否存在。

##### 2.4.3.2 适用题型

1. 一个一个字符的遍历
2. 节约Hash表空间复杂度
3. 查找字符串、查找前缀、 检测字符串种是否匹配某种正则表示式。
4. 矩阵类字符串，一个一个字符深度遍历问题（DFS + Trie）

目标字符串集一般情况下可以用Hash方式存储，当我们DFS查找到相关结果后去Hash表中进行验证，这样就失去了利字符串前缀相同的优势。

所以可以将目标字符串以Trie树方式进行存储进行。根据Trie树的构建，首先能实现DFS pruning，跳过某些不需要遍历的路径。

样题：211. Add and Search Word - Data structure design

212. Word Search II

79. Word Search

思路：Trie树结构体下，实现DFS with pruning，优化DFS下矩阵的遍历。79简单矩阵DFS，但是212增加了search的次数，所以简单DFS已经不再合适，需要DFS with pruning

样题：425 Word Square

**思想：**DFS递归函数，带入临时结果集和最终结果集，整个函数递归过程中不检查临时结果集是否满足，只在递归函数的尾部才检查临时结果集是否满足。是，则将临时结果集放入最终结果集。这个递归过程，只是简单的做DFS遍历所有的可能节点。

而Tire Tree只是帮助DFS便利过程中进行剪枝，减少递归的次数。

**注意：** Tire Tree在创建时，加入add的字符串是空串，可能导致问题。需要在add之前进行规避。同时需要注意的是：当递归到尾部，将临时结果集放入最终结果集后，递归返回，临时结果集中的数据可能会被移除，如果最终结果集，和临时结果集都是标准collection容器的话，例如List，那么临时结果集中数据变化会影响到最终结果集。所以**需要深度拷贝**，放入最终数据集的是根据临时数据集而新生成的对象。

#### 2.4.4 思考

树作为数据存储的方式优化了数据搜索/查询的效率，同时进一步优化方案就是：具体在每个数的节点上存储什么？可以直接存储我们要求的解；例如线段树/Trie树的每一个节点，可以存储当前节点所表示的区间范围内的最大值，最小值，平均值，总和，topN热点词等；这样就能大大优化解决问题的效率

适用场景：TrieTree存储搜索的高频词的统计，可以用于推荐系统等**；**

### 2.5 并查集 Union Find

将题目转化为下面两个个操作：

* 1. 找到出现的点
  2. 判断当前点和目标点是否在一个集合中
  3. 集合合并

操作： find/union/query 都是O(1)复杂度

几个注意点：路径压缩，带权重合并，统计连通块数量（要求5分钟写完模板）

样题： 200 Number of Islands

305 Number of Islands II

261 Graph Valid Tree

小技巧：n\*m的二维数组转成一维数组:(x, y) => x \* m + y

并查集的大集合代表了所有可能的元素， 200/305中大集合有1的若干集合，也有0的若干集合，所以在并查集外使用计数器统计1集合的个数，题目求解只需要关注1的结合。261大集合即为求解的结合，所以用并查集内部的计数器统计。

### 2.6 宽度优先搜索 BFS

1. 图的遍历 Traversal in Graph/Tree
   * + 层级遍历 Level Order Traversal
     + 由点及面 连通图 Connected Component
     + 拓扑排序 Topological Sorting
2. 最短路径 Shortest Path in Simple Graph
   * + 仅限简单图求最短路径
     + 即，图中每条边长度为1，且没有方向

**思路**：

1. 遍历当前图/矩阵：构造邻接表，或者做一些计数等准备工作；
2. 确定BFS开始的起始点并放入queue，**可能一个，也可能多个**；

例如：拓扑排序就是把indegree为0的所有点作为起始点放入queue；

1. 使用queue + set组合来遍历所有的点，queue用来遍历节点，set访问表用来防止节点被多次访问。**这是一对好基友，任何时候有点一旦入queue，那么它必须也要入hash访问表。Hash访问表的使用与否取决于当前处理的数据集元素之间是否可能存在回路。同时需要注意set访问表的有效范围，是全局有效还是对每一种可能性有效（图的最短路径长度vs图的最短路径下的所有路径）。**树的遍历以及拓扑排序（有向无环图）中不需要set，数据之间没有回路的可能性；图的节点遍历，或者是字符串的转换遍历，那么数据集元素之间是存在回路的；
2. 如果是分层遍历，每次遍历都需要提取当前queue的size来分层；

例如：简单图最短路径/求最少时间/求最少步数都是分层遍历+计数；

|  |
| --- |
| int size = queue.size();  for (int i = 0; i < **size**; ++i) {  //access nodes and put new nodes into queue  } //for循环体重的size如果直接用queue.size()替换会如何 |

#### 2.6.1 层级遍历

总体思路： 使用Queue临时存储层级节点，进行遍历

样题: 102 Binary Tree Level Order Traversal

107 Binary Tree Level Order Traversal II

103 Binary Tree Zigzag Level Order Traversal

#### 2.6.2 由点及面的图遍历(连通块 Connected Components)

##### 2.6.2.1 图的存储

邻接表(Adjacency List)：

|  |
| --- |
| Map<Integer, Set<Integer>> graph |

或者：

|  |
| --- |
| ArrayList<Node> lists;  Class Node {  int label;  List<Node> neighbors;  } |

##### 2.6.2.2 图的遍历

总体思路：queue + 访问表/Hash表，进行遍历

E个点，V条边的图，BFS时每个点最多被访问一次，但是每条边最多被访问2次，所以时间复杂度为： O(E + V), E为点的数量，V为边的数量。

样题：261 Graph Valid Tree

思路：图 => 树，表示该图不存在回路，即满足以下两个条件：

1. 边的数量 = 点的数量 – 1；
2. 所有的点都被连接了 =》BFS遍历计数；

样题：133 Clone Graph

思路：将提供的表用邻接表来表示出来 => BFS 遍历

拷贝所有的点；

拷贝所有的边；

样题：323 Number of Connected Components in an Undirected Graph

127 Word Ladder

126 Word Ladder II

思路：126/127，题意求解字符串变迁后是否得到目标字符串，有，则返回最短路径长度，以及具体路径。

* 1. 根据字符串变迁规则创建无向图，注意无向图的创建是否正确，是否包含首尾字符串等；
  2. 使用BFS求出当前无向图上首尾字符串之间的最短路径，当然可能不存在该路径。与此同时创建从首字符串开始到图上每个节点的分层信息表，即每个点的最短路径值。
  3. 根据分层信息表做DFS，从尾字符串开始遍历，只搜索层级比当前节点低的节点，实现广度和深度上的pruning，得到结果。

拓展：BFS可以在求得目标字符串后，即停止，代表这是一条最短路径已经确定；然后DFS从尾开始DFS，只遍历在层级信息表中出现的点并且点的层级比当前节点低，这样就能求得一条最短路径。

如果要求得所有的最短路径，那么BFS层级遍历在遍历到目标字符串后，不要停止，需要把当前层级遍历完成，为当前层的节点标注上层级信息后，意味着在当前最短路径下所有可能的节点都已经被遍历过且已经标注了层级信息；同样DFS只需要遍历层级信息表中出现的点并且点层级比自己低即可。

#### 2.6.3 拓扑排序(Topological Sorting)

定义：对一个有**向无环图**(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u,v)∈E(G)，则u在线性序列中出现在v之前。 通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。

思路：从indegree为0的点开始遍历，依次将indegree为0的节点放入queue中。由于有了indegree为0的判断，所以hash访问表不需要了，因为一旦存在环路，那么存在环路的点是不可能出现入度为0的情况。

一般情况下为DAG， 如果出现环路，根据indegree的条件入queue，导致有的点可能无被遍历到，所以需要多判断几点：1. 初始开始遍历的时候是否存在indegree为0的节点；2. 遍历完成之后，需要统计遍历节点信息，才能得知是否真的已经遍历了所有的节点。

时间复杂度和图上的BFS一样： O(E + V), E为点的数量，V为边的数量

样题：LintCode 127 topological Sorting

207 Course Schedule

210 Course Schedule II

444 Sequence Reconstruction

思路：444分析，分析题意可以得出以下2个步骤，

* + - 创建最小共同序列，这个最小共同序列就是所有序列重合在一起所组成的有向图。每个序列为图中的若干条边，也就满足子集的关系；

注意生成图时可能遇到： 重复的边，空边等问题。

* + - 验证这个最小序列（图）是唯一的，即BFS遍历图，满足：

1. 每次indegree为1的点只能有1个，即代表没有多个选项的存在

2. 必须遍历到所有的点，即无环

* + - 比较这个序列和目标序列一致；

思路2：点和边的检查法：org为1-n的点组成，即具有n个点，所以需要有n-1次的边来连接直接相邻的两个点，所以遍历所有的边，当有相邻点连接的边出现，则匹配中一次。最后匹配结果为0代表存在这样的一个结果（即遍历到了所有的点）。同时匹配过程中为了验证是唯一的，需要判断： 1. 如果边中出现数字超过当前最大值n，则为错（即入度为1的点不只1个）。2.如果边中的前后两个元素的顺序和org这两个元素的前后顺序不同，则为错（即代表无环，且图序列和目标序列一致）。3. 遇到重复的边，跳过不能重复计算n-1的计数（即seq匹配生成的图有重复边）。

#### 2.6.4 矩阵宽度优先搜索

N行M列矩阵，BFS时每个点被访问一次，每个点有4条边，每条边又被两个节点共享，所以边的总数为： （N \* M \* 4）/2， 所以时间复杂度为： O(N \* M + 2 \* N \* M) => O(N \* M)

总体思路：寻找合适的点作为初始点放入队列中,在这个过程中可能需要先遍历一次矩阵，以得到某些初始信息，统计数量等。同时需要维护一个二维访问矩阵，防止原矩阵内部节点被多次访问，也可以考虑在原有矩阵上修改值。

Tips：矩阵横纵坐标一起入queue的方式除了定义一个新的节点类外，最方便的方法就是： int num = x \* col + y; 即一个int值代表了横纵坐标。

样题：200 Number of Islands

LintCode 611 Knight Shortest Path

LintCode 573 Build Post Office II

### 2.7 深度优先搜索 DFS

适用于求解满足条件的所有可能方案，求解的答案：

* + - 排列（顺序相关），组合（顺序无关）。

1. 数据刷新发生在DFS递归嵌套中的每一次，Floyd-Warshall所以单次函数命中/匹配/找到排列/找到组合,临时数据集放入最终数据集，本次嵌套继续；
2. 搜索的时间复杂度：O(答案总数 \* 构造每个答案的时间)。举例：子集问题，求所有的子集。子集个数一共 2^n，每个集合的平均长度是 O(n) 的，所以时间复杂度为 O(n \* 2^n)，同理组合问题的时间复杂度为：O(n \* n!)
   * + 字符串DFS匹配和图的DFS求解最短路径下所有可能路径。
3. 数据刷新发生在DFS递归嵌套的的最后那一次，函数一旦命中，临时数据集放入最终数据集，本次嵌套结束返回；

#### 2.7.1 定义与实现

深度优先搜索是回溯法的一种特殊形式，主要和搜索树形的数据结构相关。

DFS实现： 递归或栈；

DFS实现复杂度： O(答案个数 \* 构造答案的时间)，N个元素的组合总共有答案2^n个，N个元素的组合总共有答案n！个。所以组合复杂度： O(n \* 2^n)，排列复杂度O(n \* n!),但是如果有剪枝，则很难估算复杂度。

#### 2.7.1 剪枝

DFS剪枝：深度、广度、计算量

字符串匹配问题中使用Trie Tree确定DFS具体有那些个遍历方向，从而实现广度上剪枝；例如：图遍历求最短路径下的所有路径方案问题中使用BFS先确定DFS具体有哪些个遍历方向，实现广度上的剪枝，同时又可以使用与之前BFS完全相反的方向进行DFS，根据BFS得到的最短路径，又实现了深度上的剪枝。DFS搜索实现计算量剪枝；

样题：139 Word Break

思路：DFS求解可行性，在常规DFS的思路上额外使用了：

* + - 访问列表

表示是否已经访问过该节点，已经访问过的节点作为缓存，不再访问；

* + - 优先队列

表示优先找到最靠前的节点，如果有的话就能节省时间；

* + - 哈希表

正向思路用目标串去匹配字典中的各个字符串，复杂度取决于字典中字符串数量；换个思路将字典作为set存储，并统计字典中最长数据的长度，然后枚举当前字符串每一个字符直到相对于当前位置的最长数据长度，截取作为一个新的字符串，匹配set中是否包含；

#### 2.7. DFS递归三要素

1. DFS函数的定义：注意参数定义；

最后两个参数一般为临时数据集和最终数据集；

2. DFS函数的内容：

* + - 出口：判断当前临时数据集是否满足条件，满足则**深拷贝**至最终数据集；
    - 拆解：遍历节点，判断当前节点是否满足条件，进一步递归运行，同时动态维护临时数据集；

样题：331 Verify Preorder Serialization of a Binary Tree

297 Serialize and Deserialize Binary Tree

78 Subset [递归][非递归]

90 SubSet II

注意：重复数据是如何在子集中去重的？当遍历循环中，非头结点时，如果和前一个节点值相同，则跳过不处理；

39 Combination Sum

40 Combination Sum II

131 Palindrome Partitioning

思路：长度为N的字符串，每两个字符之间可以切分，那么存在有N-1个切分位置，将这些切分位置看成为数组中的元素，就转化为了求解数组元素的所有组合，然后在所有组合下是否满足palindrome

Tips: 如何判断一个字符串是Palindrome？

如何得到一个字符串中的各个子串是否是Palindrome？

46 Permutations [递归]

47 Permutations II

31 Next Permutation [非递归]

484 Find Permutation

51 N-Queens

**特例：332. Reconstruct Itinerary**

**注意点：** 1. 有向有环图；2. 可能存在重复的边，且重复边有效；

**思路：**由于存在环路，所以不能使用BFS，使用DFS求解所有最大解，最后在左右的最大解里求得字典序最小的一个解；

**坑1：**创建邻接表，DFS遍历，遍历过程中，邻接节点列表做动态维护，那么用什么存储邻接节点呢？list？在当前容器中for遍历，用到了容器的迭代器，如果同时动态删改容器元素，将会出错。思考如何解决？遍历容器中的数据但是又不能使用迭代器，不断从链表头部取数据，然后加到容器尾部，类似于BFS中的层级遍历时那样取数据；可选方案：List：获取当前list大小，从头部开始取数据，dfs结束后再加入尾部；

**tips2：**优化方案一：求解最优解中按字典序排列第一个，考虑将邻接表中的目的地按照字母序列排列，这样当找到相同长度的解时，无需在放入临时数据集；

**tops3：**使用Hierholzer 算法求解欧拉路径；1. min heap 可以自动保证先访问 lexicographical order 较小的；2. 同时 poll 出来的 node 自动删除，免去了用 List 的话要先 collections.sort 再 remove 的麻烦。3. 这种以 “edge” 为重心的算法多靠 heap，比如 dijkstra.

### 2.8 动态规划 DP

#### 2.8.1 经典题推演

样题：120 Triangle

Tips：递归DFS + Traverse方式实现，维护一个全局的最终结果集，在最后满足条件的时候，和最终结果集比较并保存或者直接保存；

分而治之（Divide Conquer），同样是递归函数实现，和上面区别在于不需要维护全局变量作为最优/最终结果集，递归函数完成后返回的解，即为最终解；

记忆化搜索（Divide Conquer + Memorization），分而治之整个处理过程中带有中间状态，该中间状态也会被后续多次使用；所有中间状态可以提前保存、记忆，为后续的使用直接提供，减少重复计算;

记忆化搜索就是动态规划的思想；当出现重复节点，重复计算，冗余计算的时候，思考使用动态规划思想； 思路：当计算（x，y）的时候，需要依赖（x+1，y）和（x+1，y+1），所以我们计算到后两者的时候，首先计算，然后保存；以后如果使用到需要先check是否已经计算过了，已经计算过了就直接使用。

上述思路有什么问题？递归时个缺陷！如何解决？有没有想过，我们先不计算（x，y），而是使用相反的方向，**自下而上**，一上来首先就先去求得所有的（x+1，y）和（x+1，y+1）？

这样就过渡到我们正式的动态规划实现：多重循环 vs 记忆化搜索；

本样题即可以实现自下而上，也可以自上而下的多重循环动态规划；最终的目标就是一个，介绍重复计算，每个节点都只计算一次！！！

**动态规划的实质**：**使用记忆化搜索可以解决重复计算的搜索；在记忆化搜索的基础上，使用逆向思维优先实现记忆化的步骤，提前求出子/小问题的解，利用子/小问题的解来求大问题。**

**拓展**：当需要求解具体方案的时候，可以使用DFS+记忆化搜索实现优化；

#### 2.8.2 适用题型

以下题型极有可能适用动态规划求解：

1. 最大值最小值；

拓展思考：求最小值题型时需要和BFS求解题目进行判断和区分；同时也可以求解最小方案/最大方案：DP方法先求解最大/最小，同时标注来自于那个之前的状态，通过回溯状态可以得到具体方案解；

1. 判断可行性；
2. 统计方案个数；

以下题型极不可能适用动态规划求解：

1. 如果要求寻找所有具体的方案，那么肯定不适用动态规划；（有时候要求找一个最大、最小的方案时，也能使用DP来求解）；
2. 当输入数据为一个集合，而不是一个序列，即输入的一组数据可以重排（背包问题是例外）；
3. 暴力算法的复杂度已经是多项式级别；

动态规划擅长优化指数级别复杂度（2^n，n!）到多项式复杂度(n^2，n^3)，所以不擅长优化n^3到n^2；（所以分析问题时先考虑暴力方法，当暴力方法达到多项式级别，就不可能用动态规划）

#### 动态规划四要素

1. 状态 State

存储小规模问题的结果；

1. 方程 Function

状态之间的联系，如果通过小状态，来计算大状态；

1. 初始化 Initialization：

最极限的小状态是什么，起点

1. 答案 Answer

最大的那个状态是什么，终点

#### 2.8.4 坐标型DP

特点就是拥有坐标，有明确的起点/终点：

* State: f[x] 表示从起点走到坐标x

f[x][y]表示从起点走到坐标x,y

* Function： 研究走到x，y这个点之前从哪里来
* Initialize: 起点
* Answer： 终点

一般来说题目求解的答案是什么，那什么就是状态，例如求最短路径，那么f[x][y]就表示走到x，y的最短路径是多少？求总共有多少条路线，那么f[x][y]表示走到x，y总共有多少条路线；

样题: 64 Minimum Path Sum

思路: State：f[x][y]从起点走到当前x，y的最短路径

f[x][y] = Min(f[x-1][y], f[x][y-1]);

Initialize：初始化第0行和第0列，因为他们的初始化值无法由Function函数得到；

**初始化一个二维的动态规划时，通常初始化第0行，第0列；**

**有感：状态转换方程中无法推导出来的项一般都是初始化项**

样题：62 Unique Paths

63 Unique Paths II

思路：State：f[x][y]从起点走到x，y总共有多少条路径

f[x][y] = f[x-1][y] + f[x][y-1]

Initialize：初始化第0行第0列

样题：70 Climbing Stairs

扩展思路：二维矩阵上假如4个方向都能走，还能使用动态规划吗？使用动态规划的要求就是不能出现死循环，即在矩阵中绕圈圈，这样的话就没法求解，只要题目中有限制条件导致我们有一个确定的纬度方向去走，就可以用动态规划；

例如：矩阵滑雪问题，从任意点出发，由于滑雪高度限制，导致必定是有一定方向可以走的，不会死循环不断绕圈；

例如：带权值的矩阵任一点x，y走K步，取得sum最大；其中的K就是限制因素，导致我们不会死循环不断绕圈；引入K，扩展至三维的动态规划： f[x][y][k]，表示走到x，y点第K步的sum值；

**有感**：动态规划的题型，一般能给人一种一定的方向性，求解总是沿着一定的地理方向性在发展和延伸；

#### 2.8.5 序列型DP

与坐标型有一定的类似性，存在着某种意义上的坐标，更确切地说存在一个在序列中的位置。

序列型 vs 坐标型：

* 坐标型的dp[i]代表了当前i的状态；

即前面的坐标点的状态相加或者相减，或者取大/小即可得到i的状态；

* 序列型的dp[i]代表了i前面的所有序列点的状态的集合；

即需要考虑前面所有序列点的状态，选取一个最优解，再由这个最优解算出当前i状态的最优解；

有些dp很容易看出来状态转换方程，有的则不是那么容易。**有感：**大的问题依赖于小的问题，转换方程可以枚举当数据从小到大的各种情况，使用最简单的数据样例来寻找规律，尤其是可以使用题目中的样例来做为枚举的例子找规律，从一个小的状态得到大的状态的值，这就是需要实现的状态转换方程；

样题：128 Longest Consecutive Subsequence

300 Longest Increasing Subsequence

思路：State： f[i]从前面的数走到i，当前递增序列长度；

f[i] = f[j] + 1; Array[j] < Array[i]

Initialize：所有点都初始化为1，因为所有点都可能做为起点形成最长的龙；function遍历所有节点再寻找是否每个节点之前是否有比自己小的，导致自己当前的龙能变的更大；

样题: 279 Perfect Squares

可以理解为枚举/罗列所有比i小的j状态的组合，这些j状态的组合都可能成为当前i状态的解的candidate，罗列这些Candidate，选择一个，就是我们人肉寻找的标准，也是代码的解题思路；

**感觉**：这也可以作为一种dp类型，就是dp点的值不是O(1)，而是O(n),n为当前子问题的规模，所以这一类dp很容易形成平方或者立方的复杂度；

887 Super Egg Drop

DP1:Brute force解法下，就是dp的每一种情况，都遍历之前所有可能情况；

DP2:该题最优解：逆向思维，类似于二分思想下的值域二分，在已知解的情况下，反推是否满足情况，求得满足情况下的最优解；**按值反向递推DP**

样题： 368 Largest Divisible Subset

思路：之前有说过，动态规划极有可能不能解哪些问题？其中一个就是输入数据为set而不是sequence，但是这个题目其实是可以的，可以先排序就实现了输入数据为sequence，然后寻找最大长度序列，递增且满足后者是前者倍数关系；得到了最大长度，但是无法得到Subset，这是动态规划的短处，如何处理？单独维护一个prev数组，数组下标i的值得默认是自己，动规的时候f[i]遍历前面[1,i-1]时找到的的那个状态f[j]+1，那个j下标就是这个prev数组当前i的值得，即指明了我前一个满足倍数关系的值是什么，最后的时候找到最大长度后，遍历一次prev就能得到所有subset

思路2：记忆搜索：遍历所有节点，对当前节点做类似DFS，遍历其他节点，满足是当前节点倍数的点都记录下来并维护最长路径；全局访问表记录节点是否被访问过了；最后也能得到一组最长的长度，同时也能得到Subset；

样题：354 Russian Doll Envelopes

#### 2.8.6 滚动数组优化

对某些特殊的坐标型、序列型动态规划进行空间上的优化，滚动数组可以实现从O(n)优化到O(1);

创建状态表时，不再根据原有的数据集来创建一个同等大小的状态表集合，而是创建一个具有2个元素一维数组状态表，第一个元素索引0作为一个dummyHead，f(0) = 0，第二个元素索引1对应原有数据集中索引为0的数据的状态f(1) = a[1]，以此为初始状态；后续的状态更新使用一下公式：

|  |
| --- |
| int num = A.length;  long[] result = new long[2];  result[0] = 0; // 加一个dummyHead  result[1] = A[0];  for(int i = 2; i <= num; ++i) {  result[i%2] = Math.max(result[(i-1)%2],  result[(i-2)%2] + A[i-1]);  }  return result[num%2]; |

这类题目的特点：

f[i] = max(f[i-1], f[i-2] + A[i]); 由 f[i-1],f[i-2] 来决定状态

可以转化为：

f[i%2] = max(f[(i-1)%2]和 f[(i-2)%2]) 由f[(i-1)%2]和 f[(i-2)%2] 来决定状态

**观察我们需要保留的状态来确定模数：滚动数组（rolling array）**

样题：198 House Robber

213 House Robber II

70 Climbing Stairs

221 Maximal Square

思路：一个正方形可以通过1个坐标点，1个边来确定；我们选在右下角作为坐标点，从该坐标点出发往3个方向（左上角、向上、向左）延伸寻找做多能有多少个1，即最多能把边撑到多大，3个方向的取值取min，就能确定出从当前点延伸，最大能画出多大的正方形；

三个方向确定是否能延伸多少个1，即能延伸多长的边，每一个方向都是一个简单dp，得出3个dp状态图，在这3个状态图的基础上，最后做第4次dp，得出最终的解。算法推演的步骤，就是从大到小的推演过程。 f[i]完全可以考虑成最后一个节点。

* + 网格类题目，正方形用右下角作为定位角，长方形可以用左上角和右下角作为定位角度。（用右下角的原因：遍历数组时从左上往右下遍历，所以以右下为定位角，可以直接使用之前已经计算得出的解）；
  + 网格坐标递归，所以初始化第0行第0列；
  + 滚动数组优化空间，滚动数组为一个二维数组；写法上可以先写非滚动数组版本，分配O(N)空间，完成代码后再改成使用滚动数组：

1.修改状态数组大小；

2.所有用到状态数组的地方横坐标做摸2操作；

f[i][j]行由f[i-1][j]行来决定，那么状态转变为：

f[i%2][j]行由f[(i-1)%2][j]行来决定

样题：LintCode 631 Maximal Square II

62 Unique Path

**64 Minimum Path Sum**

**63 Unique Path II**

**注意： 64/63题目如果需要使用滚动数组优化，则在初始化状态时不可以一下子同时初始化所有初始状态，因为这些初始状态每个点都不同（62题中的所有初始状态值都是相同的），滚动数组需要在滚动的同时计算出当前的初始状态；所以要使用滚动数组优化64、63，那么代码上稍微看起来复杂一点；**

#### 2.8.7 记忆化搜索

特点：1. 动态规划状态转换方程特别麻烦，不是顺序性；(不同于以前那种直观的

顺序性：从左往右递推，或从左上往右下递推)

2. 起点不好选；

3. 记忆化搜索可以解决博弈类DP和区间类DP；

记忆化搜索本质上就是“深度遍历 + Pruning with缓存数据”；

记忆搜索vs动态规划

* 都是从小往大去递推/递归;
* 动态规划从小往大递推，把小问题的解求出，递推得到大问题的解；
* 记忆搜索从小往大递归搜索，在解决子问题的同时，保存子问题的结果;
* 记忆搜索在区间DP里的使用，需要逆向思维，从小往大递归搜索，无法找到可以重复利用的子问题；故改为从大往小递归搜搜，假设结果已经求出来了，然后逆推出可能出现的上一次是什么情况；

样题：LintCode 398 Longest Increasing Continuous Subsequence II

403 Frog Jump

问题：状态转换方程不容易表示出来；

思路：普通DP无法解的存在几个卡点：1. 不存在DP中的递推顺序性；有最多4个方向可以递推，不像之前那样有直观的顺序性：一维数组那样只能从左往右，二维数组从左上到右下；2. 起点不好选； 故使用记忆化搜索，搜索可以解决一切问题，加上记忆化功能实现剪枝优化；

#### 2.8.8 博弈类DP

可以使用顺序性DP求解，因为他是从头到位的顺序遍历；也可以记忆化搜索解决

（**记忆化搜索可以解决一切问题）**

样题：LintCode 394 Coins in a Line 【标准DP/记忆化搜索】

思路：搜索/枚举当 5个点的时候，画树状图，列举所有可能性；；（枚举的个数尽量保证在各种取的方案结束之后，3层即结束：先手层->后手层->先手层）；将先手层定义为状态，并设计状态转换方程；定义状态dp[i]为当前剩i个点时，先手是输是赢；

Tips：n%3 == 0,后手必赢；n%3 ！= 0，先手必赢

样题：LintCode 395 Coins in a Line II 【标准DP/记忆化搜索】

问题：转换方程能够表示出来，代码实现有一点小小复杂难度；但是初始化状态无法确定，根本没法写出来；

思路：枚举4个元素，画树状图，列举所有可能性；（枚举的个数尽量保证在各种取的方案结束之后，3层即结束；同样先手层定义为状态，设计转换方程；定义状态dp[i]为当前还剩下i个节点时，先手取得的所有币值总和；当我们发现先手币值总和大于所有币值总和的一半时，意味着先手能赢；

#### 2.8.9 区间型DP

特点：1.求最大一段区间内的解max/min/count

2.转移方程通过区间更新 【区别于Segment Tree】

3.逆向思维，实现从大(终状态)到小（始状态）的更新 + 记忆化搜索

题目的共性表现为：

* 区间最后求【0，n - 1】这样一个区间；
* 逆向思维分析，从大（最终状态）到小（初始状态）分析问题；

感想：感觉区间DP的题目都很类似于那种给定一组数据，然后在这组数据中做随机操作，没有一致的顺序性，该操作会导致相应的数据做合并或者发生相应的变化，求怎么操作可以求得一个最优解;

样题：LintCode 396 Coins in a Line III

问题：状态转换方程代码不是很方便写；

思路：和之前两题完全不一样，已经没有了顺序性的特征，故只能当做区间性DP求解，使用记忆化搜索；dp[i][j]现在还剩下第i到第j个硬币，先手能拿到的最多的硬币；采记忆搜索方式：初始化一般为所有[i, i]的值，即左对角线上的值，求解的结果一般为[0,n-1]位置的值，即右上角的值；状态装换方程容易表示，但是代码不方便写；

样题：LintCode 476 Stone Game

877 Stone Game

思路：普通搜索：从原始状态开始枚举各种两两合并，会发现中间状态没法出现被重复利用的可能性，说明从小到大搜索不可行；逆向思维，如果从大到小进行搜索，枚举从已经合并为一堆的情况开始，推演上一次有哪些状态，可以发现有重复计算的中间状态可以被利用；区间DP的特点，逆向思维，从大到小的更新；

dp[i][j]为当前区间内获得做少的石头数，分解为三部分，前两个部分为有可能分解成的两个堆，最后一部分是当前区间的sum；

将不需要合并的单独项初始化为0，所以状态转换函数中对单独项dp得值为0，需要合并的项[i,j]可以继续递归，同时还要加上当前的[i,j]的sum

问题：状态转换方程可以表示出来，但是代码不好写，故应使用记忆化搜索

样题：312 Burst Balloons

思路：普通搜索：从原始状态枚举不同情况，发现没有重复的中间状态可以利用；逆向思维，从大到小搜索，

问题：起点不好选；状态转换方程可以表示，但代码不好写，故使用记忆化搜索；

样题：87 Scramble String

错误：1.String.substring的参数用错；2.记忆化搜索应用在字符串上时，和数组一样，数据源不变，移动相对的索引；但是错误的使用相对应的子串代入到DFS函数中，而索引确实全局的；3.当修改了错误2后，简单的做了删除相应的code，以为解决了，犯了经常犯的错误：找到一个错误，轻易的修改以为能解决，但是带来了新的错误；应该重新审视整个代码逻辑，查看是否正确合理，而不是简单粗暴的修改了相应的地方就完事；

思路：dp[i][j][k]字符串1从i开始的k个字符内，字符串2从j开始的k个字符内，是否满足Scramble；

这题在状态转换成引入了第三个纬度，实现了2个字符串上的区间DP；需要体会和匹配类型DP之间的相似和不同

#### 2.8.10 匹配型DP

题型一般表现为，给定两个字符串或者数组，求解各种match后的结果；

State：dp[i][j]，**第一个seq前i个数字/字符，第二个seq前j个数字/字符，能否实现要求的结果**;

Function：dp[i][j]研究第i和第j个匹配关系

Initialize：dp[i][0]和dp[j][0]（矩阵表示方式中意味着第一行第一列）

Answer：dp[n][m] min/max/数量/是否存在关系

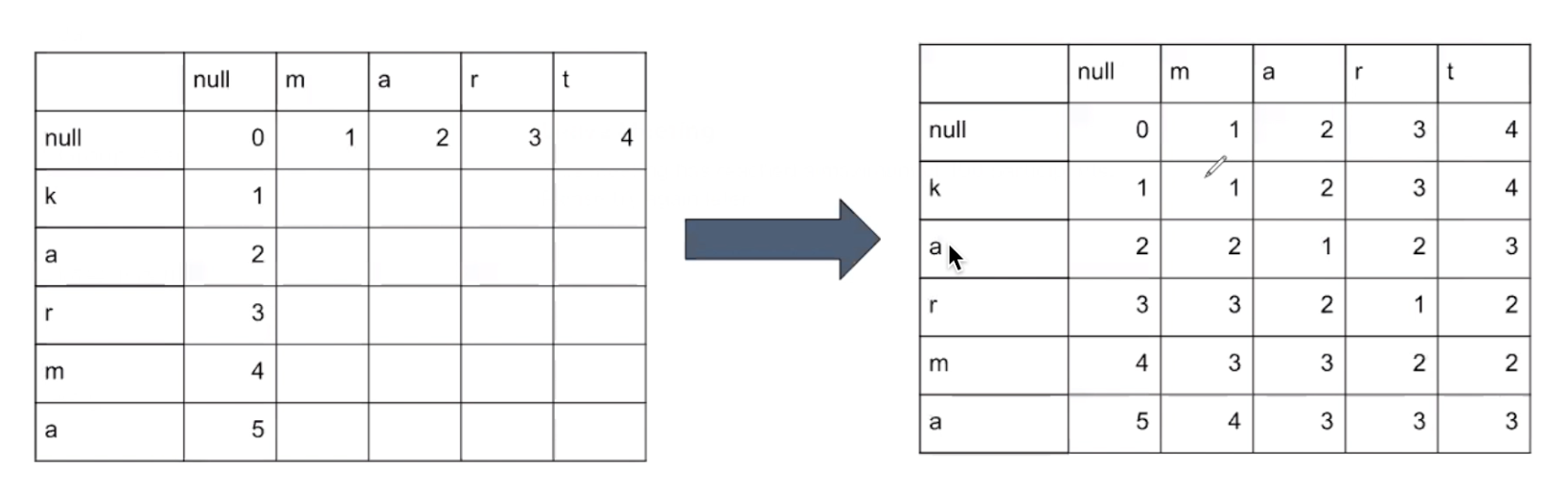
n = s1.length;

m = s2.length;

样题：72 Edit distance

思路：状态dp[i][j]，代表s1字符串的前i个，需要多少edit distance才能变成s2字符串的前j个；

状态转换方程：1.画矩阵；2.填写矩阵；



样题：115 Distinct Subsequences

LintCode 623 K Edit Distance (可以用其他更优解而不用DP)

97 Interleaving String

LintCode 77 Longest Common Subsequence

LintCode 762 Longest Common Subsequence II

#### 2.8.11 背包类DP

* 按值作为DP纬度（类似于二分法中的按值进行二分,当出现一个上限值，我们可以使用背包来解决，枚举该上限值）
* DP过程就是填写矩阵
* 可以滚动数组优化

样题：LintCode 125 BackPack II （0/1背包，有价值背包）

LintCode 440 BackPack III（完全背包，有价值背包）

LintCode 92 BackPack （无价值背包，一次取，求可能装最大数）

LintCode 562 BackPack IV （无价值背包，填满，多次取，几种可能）

LintCode 563 BackPack V （无价值背包，填满，一次取，几种可能）

* Backpack II（0/1 Knapsack）和Backpack III（Unbounded Knapsack）

有价值的背包，求在一定的成本/体积下，放入物品价值最优问题；

**状态值**：dp[i][j]前i个物品当中选一些物品组成容量为j的最大价值

**状态转换函数**：可以参考背包九讲；定位节点[i,j]，有2种选择，选和不选，得到如下：

0/1背包：dp[i][j] = **max**(dp[i-1][j],dp[i-1][j-A[i-1]] + V[i-1])

多重背包：dp[i][j] = **max**(dp[i-1][j],dp[i][j-A[i-1]] + V[i-1])

可见，无论是多重背包还是01背包，都依赖于本行以及上一行，所以可以使用滚动数组，甚至单行数组来进行优化；

01背包从右往左遍历，遍历到A[i-1]的点，该节点右边的节点值由转换函数得出，左边的节点值继承自上一个状态即可；

Unbounded背包可以从左往右遍历，起始位置为A[i-1]点，左边点的值继承自上一个状态值即可，右边点的值由状态转换公式得到；

**初始化**：第0行第0列都初始化为0

**Follow up:** 如果修改“背包能装满情况下”， 改为“背包正好装满”，需要修改初始化值，第一列初始化为0，第一行除去第一个元素外，都设置为无效值-1；

* BackPack IV： 无价值，一定的成本/体积下填满，最多几种可能，多次取物；
* BackPack V: 无价值，一定的成本/体积下填满，最多几种可能，一次取物；

**状态值**：dp[i][j]前i个物品组成和为j的组合最多个数；

**状态转换函数**：定位节点[i,j]，有两种选择，选和不选，得到如下：

IV无价值的多重背包：dp[i][j] = dp[i-1][j] + dp[i][j-A[i-1]]

V无价值的01背包：dp[i][j] = dp[i-1][j] + dp[i-1][j–A[i-1]]

可见，状态值仅仅依赖于上一行状态，可以使用滚动数组，甚至以为数组就能求得结果；同理01背包从右往左遍历到A[i-1]点，Unbounded背包从左边往右遍历，起始于A[i-1]点；

**初始化**：第0列初始化为1（都存在一种可能0物品填满0大小背包），第0行除第一个元素都初始化为0（注意和下面**不要求**装满背包情况下的初始化之间的区别）

* BackPack：无价值背包，求一定的成本/体积下，可能放入物品个数最多；

**状态值**：dp[i][j]前i个物品能否取出一些组成体积为j

**状态转换函数**：定位节点[i,j]，有2种选择，选和不选，得到如下：

dp[i][j] = dp[i - 1][j] **|** dp[I - 1][j – A[j]]

**初始化**：第0列初始化为True（空包永远都可能被填满），第0行除第一个元素初始化为False（注意和上面**要求**装满背包情况下的初始化之间的区别）

**最后得到所有体积中可能被最终填满的，在其中挑选最大的一个；**

样题：LintCode 89 K Sum

LintCode Minimum Adjustment Cost

**感想：如何识别出背包问题的马甲题，难点在于如何发现找到那个背包的值，即一堆的数字在凑，最终凑出来一个带有上限的值；**

#### 2.8.12 双DP空间问题

需要构建2张DP状态表来进行求解；区别于之前遇到的动态规划题型：

* 1. 立足于最后一个点（中间的某点），他的状态值由前一个或者若干个状态值，即经过O（1）求得；
  2. 立足于最后一个点（中间的某点），他的状态值由之前所有的状态值比较组合，即经过O（n）而求得；

双DP空间的题目有点类似于第一种情况，但是不同之处在于，前一个状态或者前n个状态，不能再O(1)复杂度内求得，即没法直接使用该DP空间中之前算出的值，也没法表示成经过若干装填计较而得；从而不得不使用第二张DP空间表来求，当然第二张DP空间表可能也会依赖于第一张DP表中的一些旧值；

样题：123 Best Time to Buy and Sell Stock III

188.Best Time to Buy and Sell Stock IV

思路：涉及到交易次数，需要将次数考虑到dp定义中；

714 Best Time to Buy and Sell Stock with Transaction Fee

309 Best Time to Buy and Sell Stock with Cooldown

思路：内有设计到交易次数，则直接定义买卖两种操作的dp；

LintCode 42 Maximum Subarray II

LintCode 43 Maximum Subarray III

## 3. 应用篇

### 3.1 窗口应用/子数组应用

#### 3.1.1 Sliding window with fixed size

窗口大小固定，在集合中滑动。滑动的过程的解题思路：就是减去一个元素，加上一个元素的过程。

1. 均值

遍历集合，窗口内元素的维护实现增量加减，遍历求均值。

样题： 643 Maximum Average Subarray I

1. 最大值、最小值、中值

遍历集合，窗口内元素的维护实现增量加减，并存入Heap（PriorityQueue/TreeSet），Monotonous Deque

样题： 480 Sliding Window Median

239 Sliding Window Maximum

#### 3.1.2 Sliding window with unfixed size

窗口大小不固定，在集合中滑动。在满足窗口大小条件下，在子集中寻找结果。由于窗口大小不确定，所以之前的方法不适合，频繁的窗口变动增加时间复杂度，同时也不适合对窗口进行不同大小的遍历，窗口大小的遍历又增加额外一层的复杂度。

1. 可变窗口大小子集的均值最大值

可变大小窗口求结果，但是窗口大小不固定，要求最优解。可以使用二分法枚举可能的结果，在反向推导是否满足检测窗口是否满足条件。

样题： 644 Maximum Average Subarray II

1. 可变窗口大小子集的最大值/最小值/求和/乘积/计数

可变大小的窗口求结果，但是窗口大小会固定，求解。使用线段树/树状数组对各个区间进行相关的信息存储，从而实现查找某段区间内的信息。

样题：LintCode: Interval Minimum Number

LintCode: Interval Sum

LintCode: Interval Sum II

LintCode: Count of Smaller Number

### 3.2 双指针应用

解题思路: 适用于array、string、list等类型的题目。

双指针 |----》 前向 |----》 窗口指针（固定/可变窗口大小）

| |----》 快慢指针

|----》 相向

#### 3.2.1 前向窗口指针

##### 3.2.1.1 可变窗口大小

由两层for循环优化而来，内部指针不需要回退。适用在线性结构中寻找一个子结构，子结构需要满足一定的条件。

|  |
| --- |
| int j = 0;  for (int i = 0; i < n; ++i) {  while(j < n && s[i - j]满足条件) {  j++;  更新j的状态，保存当前结果  例如，如果条件为最大子串且不能有重复元素，  则需要保存当前j元素已经被占用，并暂存当前长度  (代表j暂时不再动了)  }  更新i的状态（代表i即将开始移动，需要将当前i元素释放，表示不被占用）  } |

**感想：**

* **仔细体会s[i-j]满足条件，并不是对处理完之后的条件进行判断，而是提前对将要得到的处理结果做判断！ => L3/L59**
* **当内层循环体达到边界并退出，判断是否需要同时退出外层循环**
* **当内层循环体达到边界并退出，判断得出同时需要退出外层循环，还需判断是否需处理最后一次结果j => L209 vs L3**

“非”=>“是”（一般用于求满足条件），不需要处理；“是”=>“非”（一般用于求最优），需要处理；小规律：当最后一个j参与到计数中时，意味着“非”=》“是”，最后一个j就是“是”的那一刻，所以一般加入计数；否则，就是“是”=》“非”，最后一个j就是“非”的那一刻，所以一般不加入计数；

样题：209 Minimum Size Subarray Sum

3 Longest Substring Without Repeating Characters

159 Longest Substring with At Most 2 Distinct Characters

340 Longest Substring with At Most K Distinct Characters

76 Minimum Window SubString

##### 3.2.1.2 固定窗口大小

遍历目标数据集，做简单的更新操作，一旦长度满足

|  |
| --- |
| for (int i = 0; i < n; ++i) {  add(i);  if (i > windowSize - 1) {  check(是否满足条件) {  update结果集;  }  remove (i – windowsSize + 1);  }  } |

#### 3.2.2 前向快慢指针

样题：283 Move Zeros

26 Remove Duplicates from Sorted Array

55 Jump Game

45 Jump Game II

#### 3.2.3 相向指针

样题：125 Valid Palindrome

#### 3.2.4 Partition相关问题

属于相向指针

样题：75 Sort Color

LintCode 143 Sort Color II (Rainbow Sort)

#### 3.2.5 Two Sum相关问题

对于求两个变量组合的问题，可以遍历一个变量，研究另一个变量的变化；

样题：1 Two Sum

LintCode 608 Two Sum – Input array is sorted

15 3Sum

167 Two Sum II – Input array is sorted

LintCode 609 Two Sum – Less than or equal to target

16 3Sum Closest

18 4Sum

2Sum difference equals to target

前向窗口指针

### 3.3 二分性应用

枚举所有可能出现的结果，反向逆推验证是否满足题目中的条件；

样题： Lint Code Wood Cut，枚举结果集的start和end

二分查找代码书写的时候，如果存在可能性寻找的目标值超出当前数据集的范围，需要注意返回值：

1. 找到，返回值（例如大于等于目标值的第一个元素）；
2. 找不到，且所有元素小于目标值；
3. 找不到，且所有元素大于目标值；

样题：LintCode 404 Subarray Sum II

### 3.4 数组的应用

当原题中出现数组，或者对原题进行解构后遇到数组，可以尝试一下几个思路解题：

1. 数组进行排序后思考解决问题；
2. 数组元素的值作为数组的下标索引再次查找元素；
3. 使用前缀和数组 （Prefix Sum Array）思考解决问题；
4. 使用树状数组 （Binary Indexed Tree） 改进前缀和数组来解决问题；
5. 使用线段树 （Segment Tree）来解决不同区间内的解题；
6. 循环数组：重复、分裂、取反

### 3.5 哈希表应用

哈希表可以使用HashMap实现，同时也可以使用数组来实现，尤其是当需要根据字符进行哈希时；例如Trie Tree中存储字符的哈希表就是数组实现；

样题：L76 Minimum Window Substring

L49 Group Anagram

L249 Group Shifted Strings

### 3.6 一维/二维空间

二维 n \* m 的矩阵行列二分，可以得到 O(n + m)

样题： Find Peak Element => LintCode Find Peak Element II

二维矩阵转化为一维矩阵： x \* col + y，应用：

将二维矩阵节点对应到一维矩阵中；

用一个int表示二位矩阵的坐标；

### 3.7 排列组合数据去重

#### 3.7.1 组合结果集去重

1. 原始数据重复出现，但是数据只能用1次：
2. 处理原始数据，排序（Arrays.sort）；
3. DFS递归调用时，每次索引+1，即当前索引的下一个；
4. DFS递归函数中，循环体中，非头部节点的当前节点，如果和之前处理的一个节点相同时，跳过当前这个重复节点的处理；

|  |
| --- |
| if(i != start && nums[i] = nums[i - 1]) {  continue;  } |

当重复元素被选中时，且选中元素不是DFS递归函数的第一个元素，则跳过不处理。

1. 原始数据重复出现，但是数据可以重复使用多次：
2. 处理原始数据，排序 + 去重（双指针去重）；
3. DFS递归调用时，每次索引继续使用当前索引；

样题：78 Subsets

90 Subsets II

40 Combination Sum II

39 Combination Sum

#### 3.7.2 排列结果集去重

1. 原始数据集没有重复数据，所有数据只能用1次
   1. dfs函数遍历节点都从0开始，未被选取的数据在新的DFS递归中都有可能被选中；
   2. DFS递归函数动态维护visited访问数组，表示已被当前排序选中的数据不再被选取；
2. 原始数据集有重复数据，所有数据只能用1次

和没有重复数据的版本相比，增加以下两行语句（重复数据被选中且前一个重复数据没有被选中时，需要跳过）；

|  |
| --- |
| 1. Arrays.sort(nums) // 排序这样所有重复的数  2. if (i > 0 && nums[i] == nums[i - 1]  && !visited[i - 1]){  continue;  } // 跳过会造成重复的情况 |

样题：46 Permutations

1. Permutations II

感想：组合和排列代码上的区别和注意点：组合dfs函数需要传入startIndex，因为后续dfs将会从这里开始；而排序则不需要该参数，因为排序的dfs永远从0开始遍历，但是由此也需要注意不要重复访问当前元素，即需要引入visited数组标记当前已经被访问过的临时数据集中的数据；

### 3.8 Partition题型

快速排序的partition实现，从数组中取第一个值作为pivot，设置头尾各一个指针，将小于目标元素的放左边，大于目标元素的放右边，直到最后头尾指针相遇（left < right），将pivot放入即可；

Quick Select：O(n)；

变种1：如果涉及到1pivot3way的，那么在等于的那半边需要单独设定一个索引指向等于pivot的起始位置。

变种2：如果pivot不是现有数组中的，而是额外提供的，需要注意a)额外提供的pivot可能小于当前所有的元素，也可能大于当前所有的元素；b)注意头尾指针相遇的时候，这最后一个元素是什么情况，大于小于等于pivot，可能涉及到最后的返回值情况，所以最后一个值需要单独最后一次判断一下；

样题：215 Kth Largest Element

思路：分别思考O(nlogn)/O(n)来求解

LintCode 143 Sort Color II (Rainbow Sort)

280 Wiggle Sort

324 Wiggle Sort II

### 3.9 Iterator题型

Iterator类型的题都不要提前将数据全都遍历出来；这类题思想不难，主要是写法比较复杂：

* 使用Stack来解决问题；
* 处理逻辑放入hasNext中；
* next只做pop数据；

样题：LintCode 5 Flatten List

173 Binary Search Tree Iterator

173比较适合将主逻辑放入next中

### 3.10 Follow up

1. 一维转二维（可以套用相同的思路试一试）

* Find Peak Element I/II
* Trapping Water I/II
* Subarray Sum/Submatrix Sum

1. 数组变循环数组

* Continuous Subarray Sum

1. 题目条件加强（可能题目的解题方法会变化）

* Wiggle Sort I/II
* 215 Kth Largest Element in an Array VS 703 Kth Largest Element in a Stream

1. 换马甲（本质不变，变一个描述）

* Number of Airplane on the Sky/Meeting Room
* Backpack Problem
* 前向型指针问题
* Quick Sort/Bolts Nuts Problem

### 3.11 设计类的题型

一定要思考并且沟通该数据结构所适用的情形，例如有多少数据将会被输入/缓存？read heavy vs write heavy？不同的情况所设计的数据结构将会有所不同，牵扯到的函数操作的时间复杂度也会有所不同；

样题：LintCode 607 Two Sum III – Data structure design

### 3.12 二叉树相关题

Traverse vs Divide Conquer

* + They are both Recursion Algorithm
  + Result in parameter vs Result in return value
  + Top down vs Bottom up

DC解决大部分问题；

有时候不得不使用Traverse。例如求具体方案时；

有的时候需要使用Traverse + DC结合的方法，既有全局数据的记录，同时也有分治返回的结果进行处理。例如，求任意两点之间的最优问题时，就需要维护一个全局变量记录最优值，同时分治处理左右节点；

代码书写三要素：函数定义 -> 函数拆解 -> 函数出口

使用自定义的ReturnType返回自定义的返回值；

样题：LintCode 66 Binary Tree Preorder Traversal

LintCode 67 Binary Tree Inorder Traversal

LintCode 68 Binary Tree Postorder Traversal

LintCode 97 Maximum Depth of Binary Tree

LintCode 480 Binary Tree Paths

LintCode 93 Balanced Binary Tree

LintCode 453 Flatten Binary Tree to LinkedList

LintCode 88 Lowest Common Ancestor of a Binary Tree

LintCode 474 Lowest Common Ancestor of a Binary Tree II

Followup：有父亲节点

LintCode 578 Lowest Common Ancestor III

Followup：节点可能不存在树中

LintCode 595 Binary Tree Longest Consecutive Sequence

LintCode 614 Binary Tree Longest Consecutive Sequence II

Followup：二叉树上的路径变化：根->叶子,任意两个节点之间

LintCode 475 Binary Tree Maximum Path Sum II

根->任意节点: Divide Conquer

LintCode 94 Binary Tree Maximum Path Sum

任意两个父子节点之间：Divide Conquer + 全局变量

LeetCode 124 Binary Tree Maximum Path Sum

任意两个节点之间：Divide Conquer + 全局变量

小技巧：当存在负数，代码逻辑写起来会比较复杂，因为当负数出现时求解最优的时候可以放弃；根左右3点之间的正负关系判断会导致代码很复杂；所以当求解左右节点值的时候，自然的与0比较取值最大值，这样当我们选取当前根下的最大取值的时候，直接根+左+右与全局的最大作比较；同时向上传递最大路径的时候，也能直接传递根+Max(左，右)；实现了当负数出现直接丢弃；

LintCode 376 Binary Tree Path Sum I/II/III

根->叶子和满足K：Traverse，访问到叶子节点时检查是否满足和为K，满足则找到一个有效方案；

任意两个层级节点之间（不一定包含根和叶子）满足K：Traverse，访问到任何一个节点时检查从该点开始向根回溯，是否满足和为K，满足则找到一个有效方案；

任意两个节点之间满足K：Traverse，当访问到任何一个节点，做findSum操作（DFS）：findSum也是是一个DFS，将当前节点作为一个新的根节点，从这个新的根节点开始Traverse其他所有节点（Parent，left, right），此刻搜索的工作简化为：查找从当前根到其他节点之间满足sum为K的；

需要注意的是：节点上必须有指向父节点的指针，这样能帮助我们排除重复遍历的可能性，引入参数TreeNode comeFrom, 当且仅当comeFrom不等于parent/left/right的时候，才会进入继续做findSum；

如果没有指向父亲节点，如何解决？开始以当前节点为转折节点，则左右两边都是直线路径，左右各求得所有直线路径；在当前节点包含的情况下，求左右两个集合中的数据两两匹配，求得集合；

LintCode 95 Validate Binary Search Tree

LintCode 378 Convert Binary Search Tree to Doubly Linked List

LintCode 11 Search Range in Binary Search Tree

LintCode 85 Insert Node in a Binary Search Tree

LintCode 87 Remove Node in a Binary Search Tree

LintCode 245 Subtree

LintCode 469 Same Tree

LintCode 1360/L101 Symmetric Tree

LintCode 73 Construct Binary Tree from Preorder and Inorder Traversal

二叉树

* 给出一棵Binary Tree的字符串表示，比如[1,[2,3]]，还原这棵二叉树（高频）
* 给出一棵Binary Tree的先序和中序遍历序列，还原这棵二叉树
* 给出一棵Binary Tree，按照深度（同样深度从左往右）遍历并输出结果
* 给出一棵Binary Tree，输出每一条从根节点到叶子节点的路径
* 给出一棵Binary Tree，输出与之镜面对称的二叉树
* 两棵Binary Tree，判断这两棵二叉树是否完全一样（包括形状和每个点value）
* 给出两棵Binary Tree，A和B，判断B是否为A的子树

分治法

* 求一棵二叉树的最大深度（分治思想的简单应用）
* 给出一棵Binary Tree，求出这棵二叉树上和最大的路径
* 给出一棵Binary Search Tree，问是否是Balanced Binary Search Tree
* 合并k个排好序的List（高频）LintCode 104 Merge K Sorted Lists
* 求一个Array中的中位数（高频，partition方法） LintCode 80 Median
* 给出两个排好序的List，输出这两个序列中的中位数，如果存在两个中位数则输出这两个数的平均数 LintCode 65 Median of Two Sorted Arrays
* 给出一个Array，求出Array中的每个元素的右边比其小的元素个数（归并排序应用）LintCode 532 Reverse Pairs
* 给出一个平面上的若干个点，求最近的两个点的距离（要求时间复杂度小于n^2）
* 给出一个n\*n的棋盘，n是2的幂，开始时这个棋盘上只有一个格子是黑色的，其他均是白色的。现在需要用一个黑色的L型去填满这个棋盘，求一种填满的方案（要求时间复杂度尽可能低）

参考：<https://www.geeksforgeeks.org/category/algorithm/divide-and-conquer/>

<http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/BST-delete.html>

### 3.13 Priority Queue的使用

优先队列可以应用在很多场合做为辅助算法，来进行优化。

LeetCode 332 Reconstruct Itinerary

题目本质上求欧拉路径，但是需要求得字典序下最小的一条欧拉路径，考虑将路径用优先队列存储，那么找到第一条必然是字典序下最小

LeetCode 139 Word Break

本题可以使用动态规划的方法的到O(n\*n)和O(n\*k)的解题方案，基本上区别在于是遍历字符串的同时，选择枚举字典，还是选择枚举字符串的所有可能性。

但是如果排开这些算法，使用BFS，同样可以求得O(n\*k)的算法，尤其当BFS中的队列使用优先队列（降序排列），那么离字符串尾部最近的索引值将会被优先返回，在某些特征的数据下，能大大优化解题速度；

可见优先队列不仅仅是一些题的主要解题思路，同时在某些算法题中可以作为一种辅助的数据结构，帮助提高性能；应用场合在于：

当需要在众多方案中选择一个最小的/最近的/最大的/最远的等等方案时，可以考虑在主要算法思路中加入优先队列；（当然动态规划可以作为上述问题的一种常规思路，我们仅仅探讨当使用其他方案进行求解）；

# 题型归类

## Sum相关问题

### 1.1 Sum = K

**类型：**

* 一维数组子数组求和满足K
* LintCode 138 Subarry Sum
* LintCode 838 Subarray Sum Equals K
* LintCode 404 Subarray Sum II
* 循环数组，以及其他变种求和满足K
  + - LintCode 911 Maximum Size Subarray Sum Equals k
    - LeetCode 523 Continuous Subarray Sum
    - LeetCode 209 Minimum Size Subarray Sum
    - LeetCode 643 Maximum Average Subarray I
    - 其他变种一维数组子数组求和满足K，搜索LintCode
* 二维矩子矩阵求和满足K
  + - LintCode 405 Submatrix Sum
* 二叉树两点（根到叶子之间，层级节点之间，任意两点之间）求和满足K
  + - LintCode 376 Binary Tree Path Sum I/II/III

**思路：**

* 一维数组子数组求和，可以转化为前缀和数组下的2Sum + Hash/双指针，
* 一维数组元素都是正值时，可以考虑直接双指针（固定/非固定同向双指针，相向指针）；
* 二维矩阵子矩阵求和，可以转化为矩阵的横纵前缀和数组的综合使用；
* 二叉树两点之间求和，使用分而治之的方法；

### 1.2 Max Sum

* + 一维数组子数组求最大Sum
    - LeetCode 53 Maximum Subarray
    - LintCode 41 Maximum Subarray
    - LintCode 402 Continuous Subarray Sum
  + 循环数组求最大Sum
    - LintCode 403 Continuous Subarray Sum II
  + 二维矩阵子矩阵求最大Sum
    - LintCode 944 Maximum Submatrix
  + 二叉树两点（根到叶子之间，层级节点之间，任意两点之间）求最大Sum
    - LintCode 94 Binary Tree Maximum Path Sum
    - LintCode 475 Binary Tree Maximum Path Sum II
  + 多个子数组最大Sum，子数组没有重合且有效元素是连续的
    - 689 Maximum Sum of N (N = 3) Non-Overlapping Subarrays
    - Lintcode 43 maximum Subarray III
    - Lintcode 42. Maximum Subarray II
  + 多个子数组最大Sum，子数组没有重合，但有效数字可以不连续

（赔钱的交易可以忽略）

* + - 123. Best Time to Buy and Sell Stock III
    - 188. Best Time to Buy and Sell Stock IV

**思路：**

* 一维数组求子数组最大和问题：
* Kadane算法；
* 动态规划；
* 二分；
* 前缀和数组+TwoSum的应用
* 二维矩阵求子矩阵最大和问题，转化为矩阵的横纵前缀和数组的综合使用；
* 二叉树求任意两点之间最大和问题，使用分而治之方法；
* 多个子数组最大Sum，需要维护两个DP表；

考虑到当前节点分取中和不取中两种情况，将节点序列作为横轴；

* + 689 普通DP，当前节点在“被取中”时的值可以O(1)获得；
  + 123/188和689的DP不同，无法O(1)获得“被取中”时的值，同时也不同于另一种DP，无法在一段区间内枚举所有值来选在最优从而获得“被取中”时的值，因为所谓的“在一段区间内”无法确定；解法：维护2个DP空间！！！；
  + 42/43 和上述123/188中的DP类似，唯一区别就是，上述DP空间中第一行第一列可以初始化为0，股票交易默认不做生意即不亏钱是默认值；但是这两题中可能出现负值，所以如果再初始化为0就无法处理负值出现时的情况;处理方法：保证pick次数和起始数字的匹配

### 1.3 Two Sum相关变种

2Sum本身解法有很多：

* + 类似气泡排序： Time: O(n^2) Space: O(1)
  + 排序+双指针：Time: O(nlogn) Space: O(1)
  + 哈希表方法等: Time: O(n) Space: O(n)

2Sum的衍生题型：

3Sum/3Sum closest（排序+遍历+双指针） O(n^2)

4Sum等基本上也是基于2Sum原有的解法来解题；

还有很多题目最终能转换成2Sum，例如求和问题，可以求得前缀和数组，这样子数组求和问题，转换成前缀和数组下的2Sum， pre[j + 1] – pre[i] = K

样题：644 Maximum Average Subarray II

K Sum 使用动态规划求解，n个输入取m个数字，和为K的解，时间复杂度是O(n\*m\*k)

## 2. 求最大/最小/最优解

BFS求解简单图的最短路径

动态规划求解最大/小/优值

双指针遍历求解

滑动窗口求解

## 3. 求Kth最大/最小

一维数组： O(n) Quick Select

一维数组： O(nlogK) Heap

O(nlogn) Sort + 遍历

二维矩阵： O（nlogK） Heap

O(nlog(max-min)) 二分思想

二叉树： O（n） Quick Select思想

多次查询优化：缓存非叶节点的儿子个数；

样题：215 Kth Largest Element in an Array

230 Kth Smallest Element in a BST

378 Kth smallest Element in a Sorted Matrix

## 4. 求具体方案题型

基本上使用DFS求解，但是DFS存在暴栈的可能，需要pruning：

* BFS得到层级关系，如遍历图的时候，可以使用BFS得到最短路径，再使用层级关系进行剪枝；
* 如果设计字符遍历相关的题型，可以使用Trie Tree进行剪枝；
* 使用动态规划中的记忆化功能，实现计算量上的剪枝效果；

## 5. 字符串序列相关应用

1.对字符串中连续/不连续子串，求是否匹配或者求满足某种条件的问题，使用动态规划来解决，匹配型DP或者序列型DP；

2.对字符串中连续子串，求满足某种条件的问题，也可以使用双指针来解决问题；

## 6. 扫描线问题 Sweep Line

事件往往是以区间的形式存在，区间两端代表时间的开始和结束，将所有开始点和结束点进行排序，遍历这些点，辅以heap等作为解题思路。

样题： 218 The Skyline Problem

## 7. 子集/组合/排列问题

## 8. 题意解析

### 8.1寻找任意元素左边/右边比自己大/小的元素

#### 8.1.1 Monotone Deque

样题： 239 Sliding Window Maximum

#### 8.1.2 Monotone Stack

NGE问题（Next Greater Element）

样题：496 Next Greater Element I

求输入数据流的NGE，基准数据无重复数据；

使用单调栈处理基准数据并缓存到哈希表，在处理输入数据流；

503 Next Greater Element II

求基准数据中每个元素的NGE，可能有重复数据；

单调栈处理基准数据，数据直接存储到输出数组；

556 Next Greater Element III

31 Next Permutation

并没有求多个数据的NGE，理论上只是求当前数字的NGE，即求1个。

使用单调栈思想，但不需要开辟单调栈（因为不是求多个，只求1个）；

思路：从后向前按照递增序列遍历；

发现小序列，交换改数据，目标为之前数据集中第一个比自己小的；

交换完毕后，那个第一个比自己小的数据的位置上需要做倒序排列；

注意数据有效性，即新数据是否为Intger的有效范围内；

### 8.2 数组元素比大小统计/求和/求积/最大/最小

数组问题可以分解成子数组来求解，相当于分解成子数组解决问题。使用如下两个转换方程来解决：

* 顺序性转换方程：T(i, j) = T(i, j - 1) + C[j]
* 分区性转换方程：T(i, j) = T(i, mid) + T(mid + 1, j) + C[mid]

当数组问题被分解成子数组问题之后，我们所做的就是找到一个成本最小的方法来实现C；

L493 Reverse Pairs

L315 Count of Smaller Number After Self

L327 Count of Range Sum

#### 8.2.1 顺序性递归转换方程

顺序性转换方程：T(i, j) = T(i, j - 1) + C[j]

实现方式有：

* BST-based：比大小求统计
* BIT-based：求和/比大小求统计
* Segment Tree：求和/积/比大小统计/求最大/求最小
* DP(especiall for overlapping subproblems)：求最大/最小

#### 8.2.2 分区性递归转换方程

分区性转换方程：T(i, j) = T(i, mid) + T(mid + 1, j) + C[mid]

实现方式：

* Merge-sort：

### 8.3 数组滑动窗口问题

固定窗口问题，可以看做为在一个给定长度的子数组求解问题，窗口滑动的过程就是挪出一个元素，加入一个元素的实现；一般使用Heap（TreeSet/PriorityQueue）可以解决这类题目；也可能是双指针类型的题目，固定长度的双指针；

非固定窗口问题，一般使用Segment Tree/Binary Indexed Tree可以解决这类的问题，同时也可能是双指针类型的题目，非固定长度的双指针；

## 9. 题型vs算法

二叉树问题（Binary Tree problem）

* Traverse
* Divide and Conquer

Connected Graph:

* Union Find
* BFS / DFS

是否可行/最优解（做少步数、少路径）/可行解总和/求所有可行解

* 是否可行：bfs和dp都可以，暴力dfs可以但不好
* 最优解（最少步数、路径）：bfs dp都可以，暴力dfs可以但不好
* 可行解总和（最多解）：dp，暴力dfs可以但不好
* 求所有可行解：dfs with pruning（bfs，memorization， Trie）

# 出错点

## 1. int溢出

两个int相加可能出现溢出为题，当正负值同时可能出现时，2个int相减也有溢出问题。所以代码上需要做特殊处理，防止溢出。溢出可能出现情况：

* return (a1 + a2)/2 => return a1 + (a2-a1)/2
* return a1 – a2 => check +/- first, then return a1-a2;
* if (middle \* middle < x) => if (middle < x/middle)
* 2 \* A[i] < A[j] => 2 \* (long)A[i] < (long)A[j]

为了防止溢出，同时又不想时时留意在代码上做特殊处理，可以将提供的int放在自己维护的数据结构中以long的形势存储，注意如果使用long来解决这个问题，必须将int的值直接转成long，在做运算，而不能两个int做完运算后再转为long；例如：

错误写法： long tmp = 2 \* A[i] (A[i]取值为2147483647)

该错误写法原因在于后面的运算部分还是在按int做运算；

正确写法： long tmp = 2L \* A[i]

凡是涉及的int值操作边界值的，代码中务必先确定int的安全边界！

## 2. 遍历

* 切记索引的正确性！代码中务必提前设定好索引的安全边界值！例如：二分遍历时的边界性可以提前判断；
* while大循环里套while小循环，注意小循环的条件判断中需要带上大循环条件判断中的边界判断，防止越界；
* 遍历结束之后，出了循环体，可能还有最后一次的数据处理没有做！！！

1. 一种是循环中判断的条件一直不满足，遍历直到满足情况出现，由“非”转向“是”，当循环到达边界后不需要处理最后一次；
2. 一种是循环中判断的条件一直满足，但是求满足的最优值，遍历直到不满足情况出现，由“是”转向“非”，当循环体到达边界后需要处理最后一次，因为这也是一次有效的情况；

代码可以判断2种情况的或关系：即是否由是转为非，或者是否到达边界；

## 3. DFS函数实现

函数定义 + 函数分解 + 函数出口

#### 3.1 DFS函数定义

一般为：当前坐标/位置/状态，加上当前要做的下一步操作；这样函数内部实现为首先判断当前坐标是否已经到达最后一步（求解方案：判断是否到达地步；求解最优：判断是否已经到了最底或者是否已经之前访问过），然后在考虑是否要执行下一步操作，执行下一步操作一般意味着函数的分解；

#### 3.2 DFS函数求解具体方案的问题

一般都**动态维护一个临时结果集**。DFS函数首先判断是否已经到达搜索的底部（越界），是则直接刷新临时结果集到最终结果集，**需要深度拷贝！！！**退出；如果不是，则继续遍历求接（函数分解）。

在调用DFS函数之前，将下一个要遍历的数据装入临时结果集（判断数据有效+将数据放入临时结果集在调用DFS函数之外运行），DFS函数运行下一把数据，待DFS函数退出之后，再将这个已经运行完的数据导出临时结果集；

例如：46 Permutation

也可以使用另一种方法，在DFS函数内部实现：判断数据有效，再将该数据放入临时结果集，然后做数据处理，处理过程中有任何退出的地方，都需要将当前数据挪出临时结果集，所以尽量实现只有一个函数出口，最好是在最后；中间单反有退出函数的地方，一定要记得将当前数据挪出临时结果集；

#### 3.3 DFS函数求解一个可行性/最优解/方案总数的问题

一般都不需要维护临时结果集；

1. DFS函数退出与分解1：

* 【退出】判断是否越界，越界即到达底部，则直接返回相应结果值；
* 【退出】如果有记忆化功能，且已经计算过，则直接返回记忆值；
* 【分解】遍历元素，不判断下一个递归是否越界；
* 【分解】对递归返回值进行处理；

样题：LeetCode 115/403 + LintCode 394

1. DFS函数退出与分解2：

* 【退出】如果有记忆化功能，且已经计算过，则直接返回记忆值；
* 【分解】遍历元素，判断是否越界，不越界则进行递归；
* 【分解】对递归返回值进行处理；

样题：LintCode 398

样题：140 Word Break II

思路：DFS数据过大，导致时间超时，需要用记忆化搜索缓存数据；又为了避免使用过多的数据空间，每次缓存的都是下一条缓存入口，而不是缓存可能的所有数据解；所以当发现某一个索引曾经被缓存过时，需要对该缓存过的节点进行第二维的DFS；

错误点：dfs函数设置返回值，返回值表示当前index是否有匹配命中：

* + - 达到目标串尾部，返回true；
    - 曾经匹配过，匹配结果有返回true，无返回false；
    - 递归分解处，如果上级返回ture，首先需要缓存下一跳的入口，其次需要标记当前index有匹配结果，即设置返回值；

## 4. 动态规划

### 4.1 二维矩阵空间对应原始数据索引

动态规划解题时，额外开辟的dp二维数组经常是在原有输入数据的长度基础上+1，无论横纵左边都是如此；在函数中使用的坐标基本上是dp空间的坐标，对应到原有输入数据中对应的字符或元素坐标，都需要-1操作，需要注意这个问题，遇到这个问题，可以在解题时写下tips；

### 4.2 滚动数组优化动态规划问题

动态规划依赖于前n个状态，那么动态规划就可以使用滚动数组来进行优化；书写的方式为：先开辟正常的额外内存使用空间，使用正常的动态规划解决问题，等完成之后，再对额外开辟的空间进行优化，用n取模对所有dp空间进行优化；但是要想这种优化正常工作，还是要注意一下几个问题：

1. 动态规划的下一个状态值依赖于前n个状态值;
2. 二维动态规划的初始化值需要在遍历的过程中动态的生成，而不是静态地去完成初始化；即遍历数组的同时初始化第0行第0列；
3. 二维动态规划在遍历过程中进行初始化值（第一行/第一列），每一个值都要强行赋值，即显性的赋值；例如赋值1或者0，而不可以在满足条件时赋值1，不满足时（因为开辟空间是默认为0），却不去赋值为0；这个做法是错的，必须强行赋值，不满足条件时也要强行赋0；如果没有什么条件判断，默认赋值1或者0，就不涉及这个问题了；
4. 注意匹配性DP中的**矩阵索引对应到数组索**引需要-1操作；（很多类似的问题同样出现在BIT、其他DP与原数组序列的对应关系）
5. 上述3的法则同样适用于动态规划过程中状态值的赋值；

样题： 97 Interleaving String

### 4.3 背包问题空间优化

在滚动数组优化的基础上，进一步实现一维数组优化。

需要注意每一个值需要显性强行赋值；以防止旧数据被重复使用。

样题：LintCode 125 Backpack II

一维数组优化的前提是当前状态依赖于前一个状态（上一个状态+当前状态的某个值），在某些特殊动态规划中，例如找多个子数组最大和的时候，动态规划如果想使用一维数组优化，则必须前一个状态指定为上一个状态，如果还依赖当前状态的某个值，则无法使用一维数组优化，只能使用滚动数组优化；

### 4.4 DP感想

DP的初始化状态和递推方程是最主要的部分，当立足于给定状态dp[i][j]，假设i/j都是最后状态，那么递推方程可能有如下：

* + 最后节点的角度下，DP[i][j]依赖于之前所有状态集的比较并选择出最优；

当发现站在最后节点角度下无法列出方程，因为该状态还依赖后续状态，那么以为这可能是下一种可能性；

* + 任意中间节点（包括最后节点）角度下，中间节点的前后两部分决定了i/j的状态，改状态点属于[0,j]，类似这样。即DP[i][j]依赖于之前状态集的所有状态中选择最优；

参考887 Super Egg Drop， 考虑当前i/j

初始状态：p[K][N]为最后状态，即K eggs N floors下所需要的最少move；

递推方程：最后状态dp[N][K]对应的递推方程是第二种，可能存在某一楼，那个楼层扔鸡蛋后得到我当前的结果，对每一个楼进行计算比较，实现如下公式：

1 + Min(Max（dp[i-1][j-1], dp[i][N-j]）) (1<= j <= N)

**由于N很大，中间状态空间复杂度会很高，同时时间复杂度也很高，考虑是否可以绕过N，所以逆向思考，值域dp，适合求最小xxx这类题目；**

初始状态：p[i][j]为i move j eggs下能达到的最高楼层；

递推方程：中间状态dp[i][j]对应的递推方程变成了第一种，求当前能达到的最高楼层，实现如下公式：

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

## 5. Collection通用操作

PriorityQueue: offer/poll/peek

TreeSet: add/pollFirst/pollLast/first/last

Stack (LinkedList): push/pop

offer: header/first, poll/peak: tail/last

LinkedList: offer/poll/peek (offerFirst/offerLast/pollFirst/pollLast/peekFirst/peekLast)

Deque: offer/poll/peek (offerFirst/offerLast/pollFirst/pollLast/peekFirst/peekLast)

HashMap: put/get/containsKey/keySet

HashSet: add/contains

Arrays.sort(T[] a,); //Can add one more parameter: Comparator

Arrays.sort(T[] a, int fromIndex, int toIndex)

Arrays.binarySearch(int[] array, int fromIndex, int toIndex, int key)

区间[fromIndex，toIndex）是一个左闭合区间；

如果返回值为正数，搜索到这个元素在数组中，元素索引即为返回值；如果有重复元素，则永远返回一个值，但是不确定是重复元素中的某个；

如果为负数，则转换为：-(result) - 1，表示该元素并没有在数组中找到，转换后的值为搜索值应该放入到数组的索引值，存在3中情况：

1. 索引值为0，即搜索值比当前数组中所欲元素都小；

2. 索引值为当前数组最后一个元素之后的索引值，即超出当前搜索范围后的第一个值得，例如搜索[i,j]，搜索值大于元素当前i/j范围内所有值，那么返回值经过转换后得到索引值为j+1；

3.索引值为搜索范围内的某值，表示当前可插入的位置，即大于搜索值的第一个数字的位置；

感想：binarySearch可以做散列

Comparable vs Comparator

String: substring(int beginIndex)

substring(int beginIndex, int endIndex),

indexOf

String(char[] value)

String(char[] value, int offset, int count)

## 6. 图相关问题

需要注意在生成图的邻接表（adjancent list）的时候，对于点的创建，有时候一个顶点只作为目标点，那么该点是否需要在邻接表中生成一个entry。如果生成的话，那么遍历点的过程中只需要判断邻接点列表长度是否为0；如果对于这样的点不生成一个独立的entry，那么就需要对连接表判空操作，而不是判断邻接表长度；

注意有向图深度遍历中可能出现，既需要遍历某个顶点的临界点列表，同时又需要动态删改临界点列表元素。不要使用for来进行遍历，而是采用取首入尾的操作，例如LinkedList就可以进行首尾操作；如果需要考虑排序优先的情况，可选择heap(PriorityQueue)实现进行首尾操作(例如求字典序下欧拉路径)。

**无向图BFS**:

1. 邻接表使用set存储
2. queue + visited遍历
3. 是否有孤岛，visited访问量和图节点总量做比较

**无向图DFS**:

1. 邻接表使用queue存储（DFS的过程需要修改邻接表）
2. 使用visited防止重复访问；
3. 是否有孤岛，visited访问量和图节点总量做比较

**有向图BFS（拓扑排序）**:

1. 邻接表使用set存储（重复边会被排除）
2. 基于入度哈希表来遍历，不会重复访问
3. 如果需要判断是否有闭环，计算节点访问量并与图节点总量做比较

**有向图DFS**：

1. 邻接表使用queue存储（重复边都会存在）
2. 有可能重复访问同一个点；

## 7. 树状数组

初始化传入n，内部记录长度为n + 1, 开辟空间为n + 1;

注意1：传入参数均为外部数组索引，函数内部计算需要转为内部bit数组索引；

update/query传入的参数都是外部数组索引，需要转换成内部BIT数组的索引

注意2：update的时候，需要明确传入的是差值；

外部函数传入参数为更新值的话，则需要求得原来的当前值是多少，有两种方法：

1. bit数组相邻点相减求得，耗时logN；
2. bit数组外部引入copy数组，保留原来数组，这样可以得知原数组上之前的值了，但是需要注意update操作需要同时更新改copy数组；

推荐：基于以上两个注意点,外部函数的update/query函数和bit的update/quer函数互相独立实现；

样题：L307 Range Sum Query Mutable

# Appendix 相关知识点

## 1.原码、反码、补码知识详细讲解

本篇文章讲解了计算机的原码, 反码和补码. 并且进行了深入探求了为何要使用反码和补码, 以及更进一步的论证了为何可以用反码, 补码的加法计算原码的减法. 论证部分如有不对的地方请各位牛人帮忙指正! 希望本文对大家学习计算机基础有所帮助!

### 一. 机器数和真值

在学习原码, 反码和补码之前, 需要先了解机器数和真值的概念.

#### 1、机器数

一个数在计算机中的二进制表示形式,  叫做这个数的机器数。机器数是带符号的，在计算机用一个数的最高位存放符号, 正数为0, 负数为1.

比如，十进制中的数 +3 ，计算机字长为8位，转换成二进制就是00000011。如果是 -3 ，就是 10000011 。那么，这里的 00000011 和 10000011 就是机器数。

#### 2、真值

因为第一位是符号位，所以机器数的形式值就不等于真正的数值。例如上面的有符号数 10000011，其最高位1代表负，其真正数值是 -3 而不是形式值131（10000011转换成十进制等于131）。所以，为区别起见，将带符号位的机器数对应的真正数值称为机器数的真值。例：0000 0001的真值 = +000 0001 = +1，1000 0001的真值 = –000 0001 = –1

### 二. 原码, 反码, 补码的基础概念和计算方法.

在探求为何机器要使用补码之前, 让我们先了解原码, 反码和补码的概念.对于一个数, 计算机要使用一定的编码方式进行存储. 原码, 反码, 补码是机器存储一个具体数字的编码方式.

#### 1. 原码

原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

[+1]原 = 0000 0001

[-1]原 = 1000 0001

第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

[1111 1111 , 0111 1111]

即

[-127 , 127]

原码是人脑最容易理解和计算的表示方式.

#### 2. 反码

反码的表示方法是:正数的反码是其本身;负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算.

#### 3. 补码

补码的表示方法是:正数的补码就是其本身;负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

### 三. 为何要使用原码, 反码和补码

在开始深入学习前, 我的学习建议是先"死记硬背"上面的原码, 反码和补码的表示方式以及计算方法.现在我们知道了计算机可以有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

[+1] = [00000001]原 = [00000001]反 = [00000001]补

所以不需要过多解释. 但是对于负数:

[-1] = [10000001]原 = [11111110]反 = [11111111]补

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. (真值的概念在本文最开头). 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了.

于是人们开始探索 将符号位参与运算, 并且只保留加法的方法. 首先来看原码:计算十进制的表达式: 1-1=0

1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原 = [10000010]原 = -2

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:计算十进制的表达式: 1-1=0

1 - 1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原= [0000 0001]反 + [1111 1110]反 = [1111 1111]反 = [1000 0000]原 = -0

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

于是补码的出现, 解决了0的符号以及两个编码的问题:

1-1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原 = [0000 0001]补 + [1111 1111]补 = [0000 0000]补=[0000 0000]原

这样0用[0000 0000]表示, 而以前出现问题的-0则不存在了.而且可以用[1000 0000]表示-128:

(-1) + (-127) = [1000 0001]原 + [1111 1111]原 = [1111 1111]补 + [1000 0001]补 = [1000 0000]补

-1-127的结果应该是-128, 在用补码运算的结果中, [1000 0000]补 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示[1000 0000]补算出来的原码是[0000 0000]原, 这是不正确的)

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值.

### 四 原码, 反码, 补码 再深入

计算机巧妙地把符号位参与运算, 并且将减法变成了加法, 背后蕴含了怎样的数学原理呢?

将钟表想象成是一个1位的12进制数. 如果当前时间是6点, 我希望将时间设置成4点, 需要怎么做呢?我们可以:

1. 往回拨2个小时: 6 - 2 = 4

2. 往前拨10个小时: (6 + 10) mod 12 = 4

3. 往前拨10+12=22个小时: (6+22) mod 12 =4

2,3方法中的mod是指取模操作, 16 mod 12 =4 即用16除以12后的余数是4. 所以钟表往回拨(减法)的结果可以用往前拨(加法)替代! 现在的焦点就落在了如何用一个正数, 来替代一个负数. 上面的例子我们能感觉出来一些端倪, 发现一些规律. 但是数学是严谨的. 不能靠感觉.

首先介绍一个数学中相关的概念: 同余

#### 同余的概念

两个整数a，b，若它们除以整数m所得的余数相等，则称a，b对于模m同余

记作 a ≡ b (mod m)

读作 a 与 b 关于模 m 同余。

举例说明:

4 mod 12 = 4

16 mod 12 = 4

28 mod 12 = 4

所以4, 16, 28关于模 12 同余.

#### 负数取模

正数进行mod运算是很简单的. 但是负数呢?

下面是关于mod运算的数学定义:

[clip_image001](http://images.cnblogs.com/cnblogs_com/zhangziqiu/201103/201103302155507894.jpg)

上面是截图, "取下界"符号找不到如何输入(word中粘贴过来后乱码). 下面是使用"L"和"J"替换上图的"取下界"符号:

x mod y = x - y L x / y J

上面公式的意思是:

x mod y等于 x 减去 y 乘上 x与y的商的下界.

以 -3 mod 2 举例:

-3 mod 2

= -3 - 2xL -3/2 J

= -3 - 2xL-1.5J

= -3 - 2x(-2)

= -3 + 4 = 1

所以:

(-2) mod 12 = 12-2=10

(-4) mod 12 = 12-4 = 8

(-5) mod 12 = 12 - 5 = 7

#### 开始证明

再回到时钟的问题上:

回拨2小时 = 前拨10小时

回拨4小时 = 前拨8小时

回拨5小时= 前拨7小时

注意, 这里发现的规律!

结合上面学到的同余的概念.实际上:

(-2) mod 12 = 10

10 mod 12 = 10

-2与10是同余的.

(-4) mod 12 = 8

8 mod 12 = 8

-4与8是同余的.

距离成功越来越近了. 要实现用正数替代负数, 只需要运用同余数的两个定理:

反身性:

a ≡ a (mod m)

这个定理是很显而易见的.

线性运算定理:

如果a ≡ b (mod m)，c ≡ d (mod m) 那么:

(1)a ± c ≡ b ± d (mod m)

(2)a \* c ≡ b \* d (mod m)

如果想看这个定理的证明, 请看:<http://baike.baidu.com/view/79282.htm>

所以:

7 ≡ 7 (mod 12)

(-2) ≡ 10 (mod 12)

7 -2 ≡ 7 + 10 (mod 12)

现在我们为一个负数, 找到了它的正数同余数. 但是并不是7-2 = 7+10, 而是 7 -2 ≡ 7 + 10 (mod 12) , 即计算结果的余数相等.

接下来回到二进制的问题上, 看一下: 2-1=1的问题.

2-1=2+(-1) = [0000 0010]原 + [1000 0001]原= [0000 0010]反 + [1111 1110]反

先到这一步, -1的反码表示是1111 1110. 如果这里将[1111 1110]认为是原码, 则[1111 1110]原 = -126, 这里将符号位除去, 即认为是126.

发现有如下规律:

(-1) mod 127 = 126

126 mod 127 = 126

即:

(-1) ≡ 126 (mod 127)

2-1 ≡ 2+126 (mod 127)

2-1 与 2+126的余数结果是相同的! 而这个余数, 正式我们的期望的计算结果: 2-1=1

所以说一个数的反码, 实际上是这个数对于一个膜的同余数. 而这个膜并不是我们的二进制, 而是所能表示的最大值! 这就和钟表一样, 转了一圈后总能找到在可表示范围内的一个正确的数值!

而2+126很显然相当于钟表转过了一轮, 而因为符号位是参与计算的, 正好和溢出的最高位形成正确的运算结果.

既然反码可以将减法变成加法, 那么现在计算机使用的补码呢? 为什么在反码的基础上加1, 还能得到正确的结果?

2-1=2+(-1) = [0000 0010]原 + [1000 0001]原 = [0000 0010]补 + [1111 1111]补

如果把[1111 1111]当成原码, 去除符号位, 则:

[0111 1111]原 = 127

其实, 在反码的基础上+1, 只是相当于增加了膜的值:

(-1) mod 128 = 127

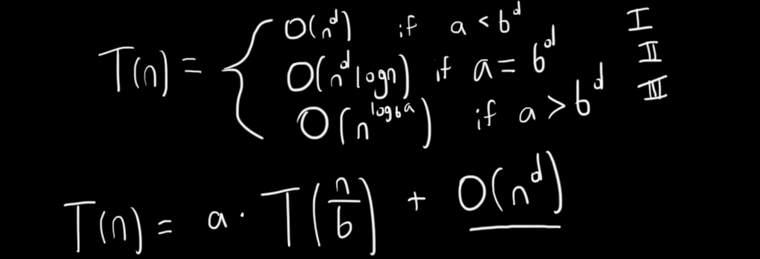
127 mod 128 = 127

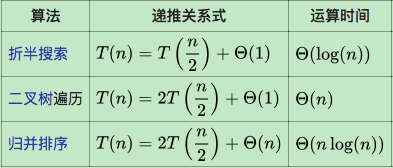
2-1 ≡ 2+127 (mod 128)

此时, 表盘相当于每128个刻度转一轮. 所以用补码表示的运算结果最小值和最大值应该是[-128, 128].

但是由于0的特殊情况, 没有办法表示128, 所以补码的取值范围是[-128, 127]

## 2.Master Theorem





Tips: big-O表示的是上界，big-Theta表示的是确界，big-Omega表示的是下界

## 3.线段树 Segment Tree

### 3.1 定义

线段树是一个满二叉树，每一个节点代表一个区间，中间节点区间被平分，叶子节点不可分。

### 3.2 线段树性质

1. 这是一个满二叉树
2. 每个节点代表一段区间的max/min/sum/production
3. 非叶子节点：左右儿子平分区间，值等于左右儿子值的更新

### 3.3 线段树的操作

#### 3.3.1 Query操作

节点区间和被查找的区间的关系如下

1. 节点区间包含被查找区间 -> 查找区间递归向下
2. 节点区间不相交于查找区间 -> 停止搜索
3. 节点区间相交于但不包含于查找区间 -> 查找区间分裂成两段区间，分别被左右两个区间包含。
4. 节点区间相等于查找区间 -> 返回查找结果

#### 3.3.2 Build操作

1. 自上而下递归分裂
2. 自下而上回溯更新

#### 3.3.3 Modify操作

1. 自上而下搜索
2. 自下而上更新

### 区间更新的优化

原来之前的操作都是单点更新，操作方法基本上是区间查找自上而下分裂，单点更新完毕之后，再从下而上进行回溯更新；

区间更新：使用延迟标记进行优化区间更新的操作。当更新/查询发生分裂，下沉延迟标记,并更新子节点。不需要再回溯更新；

#### 3.3.4.1 区间Update/Modify

参数为一个区间，不再是一个索引值，

1. 参数区间相等，更新当前节点值。函数返回
2. 参数区间等于节点区间

* 标记延迟标记值，addFlag += addVal
* 更新当前节点值val += (end - left + 1) \* addVal

1. 参数区间真包含于/跨越节点区间

* 下沉当前节点的延迟标记

（下沉函数需要检测当前节点是否有延迟标记）。

* 分裂区间，更新左右节点
* 用左右节点的新节点值求出当前节点的节点值

#### 3.3.4.2 查询函数

1. 参数区间等于节点区间，返回当前节点值。
2. 下沉当前节点的延迟标记（下沉函数需要检测当前节点是否有延迟标记）。
3. 分裂区间，查询左右节点
4. 用左右节点的新值，作为函数返回值。

#### 3.3.4.3 下沉函数

1. 检测当前节点是否具有延迟标记，无则返回；有则：
2. 为左右子节点增加延迟标记

|  |
| --- |
| left.addFlag += addFlag;  right.addFlag += addFlag; |

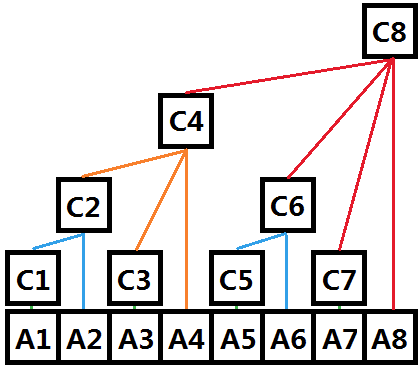
1. 为左右子节点更新节点值

|  |
| --- |
| left.val = (left.end – left.start + 1) \* addFlag;  right.val = (right.end – right.start + 1) \* addFlag |

## 4.二叉索引树/树状数组 Binary Indexed Tree

### 4.1 定义

名曰树状数组，那么究竟它是树还是数组呢？数组在物理空间上是连续的，而树是通过父子关系关联起来的，而树状数组正是这两种关系的结合，首先在存储空间上它是以数组的形式存储的，即下标连续；其次，对于两个数组下标x，y(x < y)，如果x + 2^k = y (k等于x的二进制表示中末尾0的个数)，那么定义(y, x)为一组树上的父子关系，其中y为父结点，x为子结点。



如图，其中A为普通数组，C为树状数组（C在物理空间上和A一样都是连续存储的）。树状数组的第4个元素C4的父结点为C8 (4的二进制表示为"100"，所以k=2，那么4 + 2^2 = 8)，C6和C7同理。C2和C3的父结点为C4，同样也是可以用上面的关系得出的，那么从定义出发，奇数下标一定是叶子结点。

### 4.2 节点的含义

然后我们来看树状数组上的结点Ci具体表示什么，这时候就需要利用树的递归性质了。我们定义Ci的值为它的所有子结点的值 和 Ai 的总和，之前提到当i为奇数时Ci一定为叶子结点，所以有Ci = Ai ( i为奇数 )。从图中可以得出：

C1 = A1

C2 = C1 + A2 = A1 + A2

C3 = A3

C4 = C2 + C3 + A4 = A1 + A2 + A3 + A4

C5 = A5

C6 = C5 + A6 = A5 + A6

C7 = A7

C8 = C4 + C6 + C7 + A8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8

我们从中可以发现，其实Ci还有一种更加普适的定义，它表示的其实是一段原数组A的连续区间和。根据定义，右区间是很明显的，一定是i，即Ci表示的区间的最后一个元素一定是Ai，那么接下来就是要求Ci表示的第一个元素是什么。从图上可以很容易的清楚，其实就是顺着Ci的最左儿子一直找直到找到叶子结点，那个叶子结点就是Ci表示区间的第一个元素。更加具体的，如果i的二进制表示为 ABCDE1000，那么它最左边的儿子就是 ABCDE0100，这一步是通过结点父子关系的定义进行逆推得到，并且这条路径可以表示如下：

ABCDE1000 => ABCDE0100 => ABCDE0010 => ABCDE0001

这时候，ABCDE0001已经是叶子结点了，所以它就是Ci能够表示的第一个元素的下标，那么我们发现，如果用k来表示i的二进制末尾0的个数，Ci能够表示的A数组的区间的元素个数为2^k，又因为区间和的最后一个数一定是Ai，所以有如下公式：

Ci = sum{ A[j] | i - 2^k + 1 <= j <= i } （帮助理解：将j的两个端点相减+1 等于2^k）

### 4.3 求和操作

sum(i) = sum (i – lowbit(i)) + C[i]， 迭代代码如下：

|  |
| --- |
| int sum (int x) {  int s =0;  for (int i = x; i; i -= lowbit(i)) {  s += c[i];  }  return s;  } |

### 4.4 更新操作

更新操作就是之前提到的add(i, 1) 和 add(i, -1)，更加具体得，可以推广到add(i, v)，表示的其实就是 A[i] = A[i] + v。但是我们不能在原数组A上操作，而是要像求和操作一样，在树状数组C上进行操作。

那么其实就是求在Ai改变的时候会影响哪些Ci，看图二-1-1的树形结构就一目了然了，Ai的改变只会影响Ci及其祖先结点，即A5的改变影响的是C5、C6、C8；而A1的改变影响的是C1、C2、C4、C8。

也就是每次add(i, v)，我们只要更新Ci以及它的祖先结点，之前已经讨论过两个结点父子关系是如何建立的，所以给定一个x，一定能够在最多log(n) (这里的n是之前提到的值域) 次内更新完所有x的祖先结点

|  |
| --- |
| void add (int x, int v) {  for (int i = x; i <= n; i += lowbit(i)) {  C[i] += v;  }  } |

### 4.5 lowbit函数

O(1) 实现

|  |
| --- |
| int lowbit (int x) {  return x & -x;  } |

### 4.6 小结

BIT可以应用于区间和、逆序数等等，而他的核心解决的就是求前缀和，求前缀和最naive的就是把所有的前缀都求出来，但是这样做的缺点就是在修改原数组时的代价是 ，原因就是我们存储的前缀和有n个。所以，要想降低update的代价就要存储部分前缀和，在query和update之间做一个tradeoff，至于存储那一部分的前缀和，我们只需要降低到 级别即可，所以我们选择在\*1000（\*表示任意的01串）的位置存储所有形如\*0xxx的和，这样可以保证update只有log级别。

### 4.7 应用场景

#### 4.7.1 单点修改区间查询

add x v : 给第x个元素的值加上v => add(x, v)

sub x v : 给第x个元素的值减去v => add(x, -v)

sum x y： 询问第x到第y个元素的和 => sum(y) – sum(x - 1)

#### 4.7.2 区间更新单点查询

这里介绍树状数组+差分思想，算是对下面大神的补充吧。何为差分？现在我们有一个从小到大的数列a[] a 1 3 6 8 9, 然后还有一个差分数组b[]b 1 2 3 2 1, 相信某些小伙伴已经看出端倪了..这里b[i]=a[i]-a[i-1]，我令a[0]=0，故b[1]=a[1]。

拥有了b数组，我们就可以很简单的求出bit[]中任意一个数，只需bit[i]=sigma(k=1 to i) b[k]（这个很好推吧..）上面的a[]和b[]，现在我们使区间[2,4]的所有数均+2，则a[]/b[]变为

a 1 5 8 10 9

b 1 4 3 2 -1

事实上，这里只有b[2]和b[5]发生了变化，因为区间内元素均增加了同一个值，所以b[3]，b[4]是不会变化的。这里我们就有了第二个式子：对于区间[x,y]的修改（增加值为d）在b数组内引起变化的只有 b[x]+=d，b[y+1]-=d。（这个也很好推的..）这样，我们就把树状数组的软肋用差分解决了

#### 4.7.3 区间更新区间查询

<TBD>

#### 4.7.4 逆序模型

<TBD>

## 5. 背包九讲

### 5.1 01背包（0/1 Knapsack）

#### 5.1.1 题目

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。

#### 5.1.2 基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为f[i-1][v]；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

#### 5.1.3 优化空间复杂度

以上方法的时间和空间复杂度均为O(VN)，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到O(N)。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环i=1..N，每次算出来二维数组f[i][0..V]的所有值。那么，如果只用一个数组f[0..V]，能不能保证第i次循环结束后f[v]中表示的就是我们定义的状态f[i][v]呢？f[i][v]是由f[i-1][v]和f[i-1][v-c[i]]两个子问题递推而来，能否保证在推f[i][v]时（也即在第i次主循环中推f[v]时）能够得到f[i-1][v]和f[i-1][v-c[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的顺序推f[v]，这样才能保证推f[v]时f[v-c[i]]保存的是状态f[i-1][v-c[i]]的值。伪代码如下：

|  |
| --- |
| for i=1..N  for v=V..0  f[v]=max{f[v],f[v-c[i]]+w[i]}; |

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，

因为现在的f[v-c[i]]就相当于原来的f[i-1][v-c[i]]。

事实上，使用一维数组解01背包的程序在后面会被多次用到，所以这里抽象出一个处理一件01背包中的物品过程，以后的代码中直接调用不加说明。

过程ZeroOnePack，表示处理一件01背包中的物品，两个参数cost、weight分别表明这件物品的费用和价值。

|  |
| --- |
| procedure ZeroOnePack(cost, weight)  for v=V..cost  f[v]=max{f[v],f[v-cost]+weight} |

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成v=V..0是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为cost的物品不会影响状态f[0..cost-1]，这是显然的。

有了这个过程以后，01背包问题的伪代码就可以这样写：

|  |
| --- |
| for i=1..N  ZeroOnePack(c[i],w[i]); |

#### 5.1.4 初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了f[0]为0其它f[1..V]均设为-∞，这样就可以保证最终得到的f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将f[0..V]全部设为0。

为什么呢？可以这样理解：初始化的f数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为0的背包可能被价值为0的nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是-∞了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

### 5.2 完全背包（Unbounded Knapsack）

#### 5.2.1 题目

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

#### 5.2.2 基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……等很多种。如果仍然按照解01背包时的思路，令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：f[i][v]=max{f[i-1][v-k\*c[i]]+k\*w[i]|0<=k\*c[i]<=v}

这跟01背包问题一样有O(VN)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度可以认为是O(V\*Σ(V/c[i]))，是比较大的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

#### 5.2.3 简单优化

1. C[i] <= C[j] && W[i] >= W[j]，丢弃物品j；
2. C[j] > V，丢弃物品j;

#### 5.2.4 转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c[i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第i种物品拆成费用为c[i]\*2^k、价值为w[i]\*2^k的若干件物品，其中k满足c[i]\*2^k<=V。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件物品的和。这样把每种物品拆成O(log V/c[i])件物品，是一个很大的改进。

但我们有更优的O(VN)的算法。

#### 5.2.5 O(VN)的算法

这个算法使用一维数组，先看伪代码：

|  |
| --- |
| for i=1..N  for v=0..V  f[v]=max{f[v],f[v-cost]+weight} |

你会发现，这个伪代码与01背包的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么01背包中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v=0..V的顺序循环。这就是这个简单的程序为何成立的道理。

值得一提的是，上面的伪代码中两层for循环的次序可以颠倒。这个结论有可能会带来算法时间常数上的优化。

这个算法也可以以另外的思路得出。例如，将基本思路中求解f[i][v-c[i]]的状态转移方程显式地写出来，代入原方程中，会发现该方程可以等价地变形成这种形式：

f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}

最后抽象出处理一件完全背包类物品的过程伪代码：

|  |
| --- |
| procedure CompletePack(cost, weight)  for v=cost..V  f[v]=max{f[v],f[v-cost]+w[i]} |

注意此处坐标的起始不再是从0开始，而是从cost[i]开始，因为小于当前物品成本/体积的，只能采取：不取当前物品，由上一次状态来定；在当前一维数组优化的情况下，这部分数据可以不必再做赋值，原值即为上一次状态值；

#### 5.2.6 总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法“的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

### 5.3 多重背包 (Bounded Knapsack)

#### 5.3.1 题目

有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

#### 5.3.2 基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第i种物品有n[i]+1种策略：取0件，取1件……取n[i]件。令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值，则有状态转移方程：

|  |
| --- |
| f[i][v] =max{f[i-1] [v-k\*c[i]] +k\*w[i]|0<=k<=n[i]} |

复杂度是O(V\*Σn[i])。

#### 5.3.3 转化为01背包问题

另一种好想好写的基本方法是转化为01背包求解：把第i种物品换成n[i]件01背包中的物品，则得到了物品数为Σn[i]的01背包问题，直接求解，复杂度仍然是O(V\*Σn[i])。

但是我们期望将它转化为01背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

方法是：将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。比如：

7的二进制 7 = 111 它可以分解成 001 010 100 这三个数可以组合成任意小于等于7 的数，而且每种组合都会得到不同的数

15 = 1111 可分解成 0001 0010 0100 1000 四个数字

13 = 1101 则分解为 0001 0010 0100 0110 前三个数字可以组合成7以内任意一个数，加上 0110 = 6 可以组合成任意一个大于6 小于13的数。

**小技巧：按幂次来分割数据，分到最后一个不够幂次的，直接作为最后一个数字；**

虽然有重复但总是能把 13 以内所有的数都考虑到了，基于这种思想去把多件物品转换为，多种一件物品，就可用01 背包求解了。

|  |
| --- |
| //对该种类的c[i]件物品进行二进制分解  for(j=1;j<=c[i];j<<=1)  {  Value[count]=j\*v[i];  size[count++]=j\*w[i];  c[i] -= j;  }  if(c[i] > 0)  {  Value[count]=c[i]\*v[i];  size[count++]=c[i]\*w[i];  } |

分成的这几件物品的系数和为n[i]，表明不可能取多于n[i]件的第i种物品。另外这种方法也能保证对于0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0..2^k-1和2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第i种物品分成了O(log n[i])种物品，将原问题转化为了复杂度为O(V\*Σlog n[i])的01背包问题，是很大的改进。

下面给出O(log amount)时间处理一件多重背包中物品的过程（同时使用0/1背包和完全背包，但是其实可以只用0/1背包来解），其中amount表示物品的数量：

|  |
| --- |
| procedure MultiplePack(cost, weight, amount)  if cost\*amount>=V  CompletePack(cost, weight)  return  integer k=1  while k<amount  ZeroOnePack(k\*cost, k\*weight)  amount=amount-k  k=k\*2  ZeroOnePack(amount\*cost, amount\* weight) |

希望你仔细体会这个伪代码，如果不太理解的话，不妨翻译成程序代码以后，单步执行几次，或者头脑加纸笔模拟一下，也许就会慢慢理解了。

#### 5.3.4 O(VN)的算法

多重背包问题同样有O(VN)的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊O(1)的时间求解。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

#### 5.3.5 小结

这里我们看到了将一个算法的复杂度由O(VΣn[i])改进到O(VΣlog n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)算法。希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并将完整的程序代码写出来。

## 6. 其他排序算法和洗牌算法

### 6.1 烙饼排序

Flip(array, size) + findMax(array, size)

### 6.2 Fisher Yates Shuffle

算法的思想是每次从未选中的数字中随机挑选一个加入排列，时间复杂度为O(n)

|  |
| --- |
| void Shuffle\_FisherYates(char \*arr, const int len)  {  for(int i = len - 1; i > 0; i--)  {  int a = rand()%(i + 1);  int temp = arr[i];  arr[i] = arr[a];  arr[a] = temp;  }  } |

### 6.3 Inside-out Shuffle

如果我们想保留原始的排列，洗牌后的排列放到一个额外的数组，那么改用怎么样的洗牌算法呢?

inside-out算法，算法的思想就是遍历原数组，把原数组中位置 i 的数据随机放到新数组的前i个位置（包括第i个）中的某一个（假设放到第k个），然后把新数组的第k个位置的数放到新数组的第 i 个位置，代码如下：

|  |
| --- |
| void Shuffle\_InsideOut(char \*arrSrc, const int len, char \*arrDest)  {  arrDest[0] = arrSrc[0];  for(int i = 1; i < len; i++)  {  int k = rand()%(i + 1);  arrDest[i] = arrDest[k];  arrDest[k] = arrSrc[i];  }  } |

## 7. 数组相关解题思路参考

LeetCode 493 Reverse Pairs

It looks like a host of solutions are out there (BST-based, BIT-based, Merge-sort-based). Here I'd like to focus on the general principles behind these solutions and its possible application to a number of similar problems.

The fundamental idea is very simple: break down the array and solve for the subproblems.

A breakdown of an array naturally reminds us of subarrays. To smoothen our following discussion, let's assume the input array is nums, with a total of n elements. Let nums[i, j] denote the subarray starting from index i to index j (both inclusive), T(i, j) as the same problem applied to this subarray (for example, for Reverse Pairs, T(i, j) will represent the total number of important reverse pairs for subarray nums[i, j]).

With the definition above, it's straightforward to identify our original problem as T(0, n - 1). Now the key point is how to construct solutions to the original problem from its subproblems. This is essentially equivalent to building recurrence relations for T(i, j). Since if we can find solutions to T(i, j) from its subproblems, we surely can build solutions to larger subarrays until eventually the whole array is spanned.

While there may be many ways for establishing recurrence relations for T(i, j), here I will only introduce the following two common ones:

* + **T(i, j) = T(i, j - 1) + C,** i.e., elements will be processed sequentially and C denotes the subproblem for processing the last element of subarray nums[i, j]. We will call this sequential recurrence relation.
  + **T(i, j) = T(i, m) + T(m + 1, j) + C** where m = (i+j)/2, i.e., subarray nums[i, j] will be further partitioned into two parts and C denotes the subproblem for combining the two parts. We will call this partition recurrence relation.

For either case, the nature of the subproblem C will depend on the problem under consideration, and it will determine the overall time complexity of the original problem. So usually it's crucial to find efficient algorithm for solving this subproblem in order to have better time performance. Also pay attention to possibilities of overlapping subproblems, in which case a dynamic programming (DP) approach would be preferred.

Next, I will apply these two recurrence relations to this problem "Reverse Pairs" and list some solutions for your reference.

### I -- Sequential recurrence relation

Again, we assume the input array is nums with n elements and T(i, j) denotes the total number of important reverse pairs for subarray nums[i, j]. For sequential recurrence relation, we can set i = 0, i.e., the subarray always starts from the beginning. Therefore, we end up with:

T(0, j) = T(0, j - 1) + C

where the subproblem C now becomes "find the number of important reverse pairs with the first element of the pair coming from subarray nums[0, j - 1] while the second element of the pair being nums[j]".

Note that for a pair (p, q) to be an important reverse pair, it has to satisfy the following two conditions:

1. p < q: the first element must come before the second element;
2. nums[p] > 2 \* nums[q]: the first element has to be greater than twice of the second element.

For subproblem C, the first condition is met automatically; so, we only need to consider the second condition, which is equivalent to searching for all elements within subarray nums[0, j - 1] that are greater than twice of nums[j].

The straightforward way of searching would be a linear scan of the subarray, which runs at the order of O(j). From the sequential recurrence relation, this leads to the naive O(n^2) solution.

To improve the searching efficiency, a key observation is that the order of elements in the subarray does not matter, since we are only interested in the total number of important reverse pairs. This suggests we may sort those elements and do a binary search instead of a plain linear scan.

If the searching space (formed by elements over which the search will be done) is "static" (it does not vary from run to run), placing the elements into an array would be perfect for us to do the binary search. However, this is not the case here. After the j-th element is processed, we need to add it to the searching space so that it becomes searchable for later elements, which renders the searching space expanding as more and more elements are processed.

Therefore, we'd like to strike a balance between searching and insertion operations. This is where data structures like binary search tree (BST) or binary indexed tree (BIT) prevail, which offers relatively fast performance for both operations.

#### 1. BST-based solution

We will define the tree node as follows, where val is the node value and cnt is the total number of elements in the subtree rooted at current node that are greater than or equal to val:

|  |
| --- |
| class Node {  int val, cnt;  Node left, right;  Node(int val) {  this.val = val;  this.cnt = 1;  }  } |

The searching and insertion operations can be done as follows:

|  |
| --- |
| private int search(Node root, long val) {  if (root == null) {  return 0;  } else if (val == root.val) {  return root.cnt;  } else if (val < root.val) {  return root.cnt + search(root.left, val);  } else {  return search(root.right, val);  }  }  private Node insert(Node root, int val) {  if (root == null) {  root = new Node(val);  } else if (val == root.val) {  root.cnt++;  } else if (val < root.val) {  root.left = insert(root.left, val);  } else {  root.cnt++;  root.right = insert(root.right, val);  }    return root;  } |

And finally, the main program, in which we will search for all elements no less than twice of current element plus 1 (converted to long type to avoid overflow) while insert the element itself into the BST.

Note: this homemade BST is not self-balanced and the time complexity can go as bad as O(n^2) (in fact you will get TLE if you copy and paste the solution here). To guarantee O(nlogn) performance, use one of the self-balanced BST's (e.g. Red-black tree, AVL tree, etc.).

|  |
| --- |
| public int reversePairs(int[] nums) {  int res = 0;  Node root = null;  for (int ele : nums) {  res += search(root, 2L \* ele + 1);  root = insert(root, ele);  }  return res;  } |

#### 2. BIT-based solution

For BIT, the searching and insertion operations are:

|  |
| --- |
| private int search(int[] bit, int i) {  int sum = 0;  while (i < bit.length) {  sum += bit[i];  i += i & -i;  }  return sum;  }  private void insert(int[] bit, int i) {  while (i > 0) {  bit[i] += 1;  i -= i & -i;  }  } |

And the main program, where again we will search for all elements greater than twice of current element while insert the element itself into the BIT. For each element, the "index" function will return its index in the BIT. Unlike the BST-based solution, this is guaranteed to run at O(nlogn).

|  |
| --- |
| public int reversePairs(int[] nums) {  int res = 0;  int[] copy = Arrays.copyOf(nums, nums.length);  int[] bit = new int[copy.length + 1];    Arrays.sort(copy);  for (int ele : nums) {  res += search(bit, index(copy, 2L \* ele + 1));  insert(bit, index(copy, ele));  }  return res;  }  private int index(int[] arr, long val) {  int l = 0, r = arr.length - 1, m = 0;  while (l <= r) {  m = l + ((r - l) >> 1);  if (arr[m] >= val) {  r = m - 1;  } else {  l = m + 1;  }  }  return l + 1;  } |

More explanation for the BIT-based solution:

We want the elements to be sorted so there is a sorted version of the input array which is copy.

The bit is built upon this sorted array. Its length is one greater than that of the copy array to account for the root.

Initially the bit is empty and we start doing a sequential scan of the input array. For each element being scanned, we first search the bit to find all elements greater than twice of it and add the result to res. We then insert the element itself into the bit for future search.

Note that conventionally searching of the bit involves traversing towards the root from some index of the bit, which will yield a predefined running total of the copy array up to the corresponding index. For insertion, the traversing direction will be opposite and go from some index towards the end of the bit array.

For each scanned element of the input array, its searching index will be given by the index of the first element in the copy array that is greater than twice of it (shifted up by 1 to account for the root), while its insertion index will be the index of the first element in the copy array that is no less than itself (again shifted up by 1). This is what the index function is for.

For our case, the running total is simply the number of elements encountered during the traversal process. If we stick to the convention above, the running total will be the number of elements smaller than the one at the given index, since the copy array is sorted in ascending order. However, we'd actually like to find the number of elements greater than some value (i.e., twice of the element being scanned), therefore we need to flip the convention. This is what you see inside the search and insert functions: the former traversing towards the end of the bit while the latter towards the root.

### II -- Partition recurrence relation

For partition recurrence relation, setting i = 0, j = n - 1, m = (n-1)/2, we have:

|  |
| --- |
| T(0, n - 1) = T(0, m) + T(m + 1, n - 1) + C |

where the subproblem C now reads "find the number of important reverse pairs with the first element of the pair coming from the left subarray nums[0, m] while the second element of the pair coming from the right subarray nums[m + 1, n - 1]".

Again, for this subproblem, the first of the two aforementioned conditions are met automatically. As for the second condition, we have as usual this plain linear scan algorithm, applied for each element in the left (or right) subarray. This, to no surprise, leads to the O(n^2) naive solution.

Fortunately, the observation holds true here that the order of elements in the left or right subarray does not matter, which prompts sorting of elements in both subarrays. With both subarrays sorted, the number of important reverse pairs can be found in linear time by employing the so-called two-pointer technique: one pointing to elements in the left subarray while the other to those in the right subarray and both pointers will go only in one direction due to the ordering of the elements.

The last question is which algorithm is best here to sort the subarrays. Since we need to partition the array into halves anyway, it is most natural to adapt it into a Merge-sort. Another point in favor of Merge-sort is that the searching process above can be embedded seamlessly into its merging stage.

So here is the Merge-sort-based solution, where the function "reversePairsSub" will return the total number of important reverse pairs within subarray nums[l, r]. The two-pointer searching process is represented by the nested while loop involving variable p, while the rest is the standard merging algorithm.

|  |
| --- |
| public int reversePairs(int[] nums) {  return reversePairsSub(nums, 0, nums.length - 1);  }  private int reversePairsSub(int[] nums, int l, int r) {  if (l >= r) return 0;    int m = l + ((r - l) >> 1);  int res = reversePairsSub(nums, l, m)  + reversePairsSub(nums, m + 1, r);  int i = l, j = m + 1, k = 0, p = m + 1;  int[] merge = new int[r - l + 1];    while (i <= m) {  while (p <= r && nums[i] > 2 L \* nums[p]) p++;  res += p - (m + 1);    while (j <= r && nums[i] >= nums[j]) {  merge[k++] = nums[j++];  }  merge[k++] = nums[i++];  }  while (j <= r) {  merge[k++] = nums[j++];  }  System.arraycopy(merge, 0, nums, l, merge.length);  return res;  } |

### III -- Summary

Many problems involving arrays can be solved by breaking down the problem into subproblems applied on subarrays and then link the solution to the original problem with those of the subproblems, to which we have sequential recurrence relation and partition recurrence relation. For either case, it's crucial to identify the subproblem C and find efficient algorithm for approaching it.

If the subproblem C involves searching on "dynamic searching space", try to consider data structures that support relatively fast operations on both searching and updating (such as self-balanced BST, BIT, Segment tree, ...).

If the subproblem C of partition recurrence relation involves sorting, Merge-sort would be a nice sorting algorithm to use. Also, the code could be made more elegant if the solution to the subproblem can be embedded into the merging process.

If there are overlapping among the subproblems T(i, j), it's preferable to cache the intermediate results for future lookup.

Lastly let me name a few leetcode problems that fall into the patterns described above and thus can be solved with similar ideas.

315. Count of Smaller Numbers After Self

327. Count of Range Sum

For leetcode 315, applying the sequential recurrence relation (with j fixed), the subproblem C reads: find the number of elements out of visited ones that are smaller than current element, which involves searching on "dynamic searching space"; applying the partition recurrence relation, we have a subproblem C: for each element in the left half, find the number of elements in the right half that are smaller than it, which can be embedded into the merging process by noting that these elements are exactly those swapped to its left during the merging process.

For leetcode 327, applying the sequential recurrence relation (with j fixed) on the pre-sum array, the subproblem C reads: find the number of elements out of visited ones that are within the given range, which again involves searching on "dynamic searching space"; applying the partition recurrence relation, we have a subproblem C: for each element in the left half, find the number of elements in the right half that are within the given range, which can be embedded into the merging process using the two-pointer technique.

## 8. 欧拉回路（Eulerian Path）

今天讨论的主题是一类问题，就是欧拉路问题。有两种欧拉路。第一种叫做 Eulerian path(trail)，沿着这条路径走能够走遍图中每一条边；第二种叫做 Eularian cycle，沿着这条路径走，不仅能走遍图中每一条边，而且起点和终点都是同一个顶点。注意：欧拉路要求每条边只能走一次，但是对顶点经过的次数没有限制。

满足什么性质的图才能有欧拉路？根据 wikipedia 对欧拉路的介绍：

* 在无向图中，所有顶点的度数均为偶，则存在 Eularian cycle；若有且仅有两个顶点的度数为奇，其余的都为偶，则存在 Eularian path；
* 在有向图中，所有顶点的入度数等于出度数，则存在 Eularian cycle；若有且仅有两个顶点：其中一个入度数比出度数大 1，另一个入度数比出度数小 1，其余的顶点入度数等于出度数，则存在 Eularian path.

另外我们还需要知道，对于那些 Eularian path，起点和终点分别在那两个度数为奇的顶点上（对于无向图）或是入度数不等于出度数的顶点上（对于有向图）。

然而知道这些并没有给我们带来多少实惠。因为我们除了判定一个图有没有欧拉路之外，更想找到其中的一条欧拉路径。**于是这就是我们今天的重点：寻找欧拉路径的算法**。

### 8.1 Fleury 算法

Fleury 算法的思想就是：在过河拆桥之前，先想想有没有退路。为什么这么说？Fleury 算法每个回合进行到一个顶点上的时候，都会删除已经走过的边。在选择下一条边的时候，不应该出现这样的状况：在删除下一条边之后，连通图被分割成两个不连通的图。除非没有别的边可选择。该算法从一个奇度数顶点开始（若所有顶点度数均为奇，则任选一个顶点）。当所有的边都走完的时候，该算法结束，欧拉路径为删除路径的顺序。用算法伪代码描述就是：

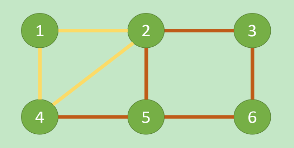
|  |
| --- |
| v\_0 <- a vertex with odd degree or, if no such vertex, any arbitrary vertex.  Repeat:  select an vertex v\_i+1 adjacent of v\_i, which should not separate the graph or, the only adjacent vertex of v\_i  remove edge <v\_i, v\_i+1> and jump to v\_i+1  Until all edges have been visited.  Return the sequence of visited edges. |

但是该算法的问题就是，怎么判断一条边是否是一个桥呢？如果使用 Tarjan 算法判断，则算法运行时间就是 O(E2)。在实际写代码的时候，我可没考虑那么多。我只考虑，如果在某一点处深搜的结果导致图被分离，那么在某一个边必然走过了一个桥，那么就返回走另一条边。这样的思想形成的算法如下：

|  |
| --- |
| #include <cstdio>  #include <stack>  #include <iostream>  #include <string>  using namespace std;  int G[1001][1001];  int N,M;  stack<int> S;  bool dfs(int u){ //返回的状态说明是否走过了一个桥  S.push(u); //进入某一节点时推入节点，如果误入歧途还要负责弹出。  if(G[u][0]==0){ //G[u][0]代表该节点所邻接的边的数目。此段代码判断是否走完了所有边，或者没有走完。  bool flag=true;  for(int i=1; i<=N; i++){  if (i==u) continue;  flag=((G[i][0]==0) && flag);  }  if(flag==false){  S.pop();  }  return flag;  }  for(int v=1; v<=N; v++)  if(G[u][v]){  //删除边  G[u][v]-=1;  G[v][u]-=1;  G[v][0]-=1;  G[u][0]-=1;  if(dfs(v)) return true;  else{//撤销删除边  G[u][v]+=1;  G[v][u]+=1;  G[v][0]+=1;  G[u][0]+=1;  }  }  S.pop();  return false;  }  int main(){  freopen("testcase", "r", stdin);  cin>>N>>M;  int u,v;  for(int i=0; i!=M; i++){  cin>>u>>v;  G[u][v]+=1;  G[v][u]+=1;  G[u][0]+=1;  G[v][0]+=1;  }  //寻找起点  for(u=1; u<=N; u++){  if(G[u][0]&1) break;  }  if(u==N+1) dfs(1);  else dfs(u);  while(!S.empty()){  cout<<S.top()<<" ";  S.pop();  }  cout<<endl;  return 0;  } |

### 8.2 Hierholzer 算法

这种算法是基于这样的观察：



在手动寻找欧拉路的时候，我们从点 4 开始，一笔划到达了点 5，形成路径 4-5-2-3-6-5。此时我们把这条路径去掉，则剩下三条边，2-4-1-2 可以一笔画出。

这两条路径在点 2 有交接处（其实点 4 也是一样的）。那么我们可以在一笔画出红色轨迹到达点 2 的时候，一笔画出黄色轨迹，再回到点 2，把剩下的红色轨迹画完。

由于明显的出栈入栈过程，这个算法可以用 DFS 来描述。

|  |
| --- |
| DFS(u):  While (u存在未被删除的边e(u,v))  删除边e(u,v)  DFS(v)  End  PathSize ← PathSize + 1  Path[ PathSize ] ← u |

如果想看得更仔细一点，下面是从点 4 开始到点 5 结束的 DFS 过程，其中 + 代表入栈，- 代表出栈。

4+ 5+ 2+ 3+ 6+ 5+ 5- 6- 3- 1+ 4+ 2+ 2- 4- 1- 2- 5- 4-

我们把所有出栈的记录连接起来，得到

5-6-3-2-4-1-2-5-4

诸位看官可以自己再选一条路径尝试一下。不过需要注意的是，起始点的选择和 Fleury 要求的一样。

需要注意的是这个算法时间复杂度是O(E)。其在 DFS 的过程中不用恢复边，靠出栈记录轨迹。这个算法明显要比 Fleury 高效，它不用判断每条边是否是一个桥。

|  |
| --- |
| #include <cstdio>  #include <stack>  #include <iostream>  #include <string>  #include <vector>  using namespace std;  int G[1001][1001];  int N,M;  stack<int> S;  void dfs(int u){  for(int v=1; v<=N; v++)  if(G[u][v]){  G[u][v]-=1;  G[v][u]-=1;  dfs(v);  //不用恢复边！  }  S.push(u);//出栈时记录  }  int main(){  freopen("testcase", "r", stdin);  cin>>N>>M;  int u,v;  vector<int> cnt(N+1,0);  for(int i=0; i!=M; i++){  cin>>u>>v;  G[u][v]+=1;  G[v][u]+=1;  cnt[u]^=1;  cnt[v]^=1;  }  for(u=1; u<=N; u++){  if(cnt[u]) break;  }  if(u==N+1) dfs(1);  else dfs(u);  while(!S.empty()){  cout<<S.top()<<" ";  S.pop();  }  cout<<endl;  return 0;  } |

### 8.3 Leetcode样题

332 Reconstruct Itinerary

题目描述: 某次旅行后小九留下了若干张飞机票，每张飞机票上写有出发和到达的机场名。他出发的机场是“JFK”，他希望知道自己旅行时的准确路线。

数据保证有解。当有多组解时输出字典序最小的一组。

|  |
| --- |
| Example 1:tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]  Return ["JFK", "MUC", "LHR", "SFO", "SJC"].  Example 2:  tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]  Return ["JFK","ATL","JFK","SFO","ATL","SFO"]. |

分析解答

根据题目要求，一种贪心加深度优先搜索的方法是：从JFK机场出发，每次都选择一个可以到达的且字典序最小的机场，进行下一步的搜索。搜索的终止条件为所有的飞机票都使用完毕。这种暴力搜索算法理论上是可行的，但时间复杂度是指数级别的。

此题有一种更好的做法，把机场想象成点，飞机票想象成有向边，问题就转化成了在一个有向图中求一条经过所有边的路径。求欧拉路径有一种O(M)(M为边数)的Fleury算法：[；当一个点无法再经过未遍历的边到达其他点时把该点加入栈中；最后把栈中序列输出得到欧拉路径。

简单证明如下：首先定义割边，去掉该边原图不连通则该边为割边；在搜索欧拉路径的过程中一旦遇到割边就意味着当前的搜索路径需要改进，即提前经过其他某一联通子图内的边。回溯添加节点的方法可以保证割边一定最先加入栈，也就是出现在欧拉路径的最后，因此可以通过一次深度优先搜索得到完整的欧拉路径。

## 9. 回文算法之马拉车算法

这个马拉车算法Manacher‘s Algorithm是用来查找一个字符串的最长回文子串的线性方法，由一个叫Manacher的人在1975年发明的，这个方法的最大贡献是在于将时间复杂度提升到了线性，这是非常了不起的。对于回文串想必大家都不陌生，就是正读反读都一样的字符串，比如 "bob", "level", "noon" 等等，那么如何在一个字符串中找出最长回文子串呢，可以以每一个字符为中心，向两边寻找回文子串，在遍历完整个数组后，就可以找到最长的回文子串。但是这个方法的时间复杂度为O(n\*n)，并不是很高效，下面我们来看时间复杂度为O(n)的马拉车算法。

由于回文串的长度可奇可偶，比如"bob"是奇数形式的回文，"noon"就是偶数形式的回文，马拉车算法的第一步是预处理，做法是在每一个字符的左右都加上一个特殊字符，比如加上'#'，那么

|  |
| --- |
| bob --> #b#o#b#  noon --> #n#o#o#n# |

这样做的好处是不论原字符串是奇数还是偶数个，处理之后得到的字符串的个数都是奇数个，这样就不用分情况讨论了，而可以一起搞定。接下来我们还需要和处理后的字符串t等长的数组p，其中p[i]表示以t[i]字符为中心的回文子串的半径，若p[i] = 1，则该回文子串就是t[i]本身，那么我们来看一个简单的例子：

|  |
| --- |
| # 1 # 2 # 2 # 1 # 2 # 2 #  1 2 1 2 5 2 1 6 1 2 3 2 1 |

为啥我们关心回文子串的半径呢？看上面那个例子，以中间的 '1' 为中心的回文子串 "#2#2#1#2#2#" 的半径是6，而为添加井号的回文子串为 "22122"，长度是5，为半径减1。这是个普遍的规律么？我们再看看之前的那个 "#b#o#b#"，我们很容易看出来以中间的 'o' 为中心的回文串的半径是4，而 "bob"的长度是3，符合规律。再来看偶数个的情况"noon"，添加井号后的回文串为 "#n#o#o#n#"，以最中间的 '#' 为中心的回文串的半径是5，而 "noon" 的长度是4，完美符合规律。所以我们只要找到了最大的半径，就知道最长的回文子串的字符个数了。只知道长度无法确定子串，我们还需要知道子串的起始位置。

我们还是先来看中间的 '1' 在字符串 "#1#2#2#1#2#2#" 中的位置是7，而半径是6，貌似7-6=1，刚好就是回文子串 "22122" 在原串 "122122" 中的起始位置1。那么我们再来验证下 "bob"，"o" 在 "#b#o#b#" 中的位置是3，但是半径是4，这一减成负的了，肯定不对。所以我们应该至少把中心位置向后移动一位，才能为0啊，那么我们就需要在前面增加一个字符，这个字符不能是井号，也不能是s中可能出现的字符，所以我们暂且就用美元号吧，毕竟是博主最爱的东西嘛。这样都不相同的话就不会改变p值了，那么末尾要不要对应的也添加呢，其实不用的，不用加的原因是字符串的结尾标识为'\0'，等于默认加过了。那此时 "o" 在 "$#b#o#b#" 中的位置是4，半径是4，一减就是0了，貌似没啥问题。我们再来验证一下那个数字串，中间的 '1' 在字符串 "$#1#2#2#1#2#2#" 中的位置是8，而半径是6，这一减就是2了，而我们需要的1，所以我们要除以2。之前的 "bob" 因为相减已经是0了，除以2还是0，没有问题。再来验证一下 "noon"，中间的 '#' 在字符串 "$#n#o#o#n#" 中的位置是5，半径也是5，相减并除以2还是0，完美。可以任意试试其他的例子，都是符合这个规律的，**最长子串的长度是半径减1，起始位置是中间位置减去半径再除以2**。

那么下面我们就来看如何求p数组，需要新增两个辅助变量mx和id，其中id为能延伸到最右端的位置的那个回文子串的中心点位置，mx是回文串能延伸到的最右端的位置，这个算法的最核心的一行如下：

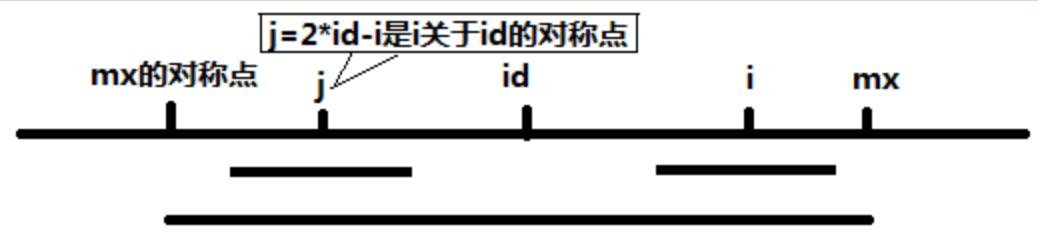
|  |
| --- |
| p[i] = mx > i ? min(p[2 \* id - i], mx - i) : 1; |

可以这么说，这行要是理解了，那么马拉车算法基本上就没啥问题了，那么这一行代码拆开来看就是

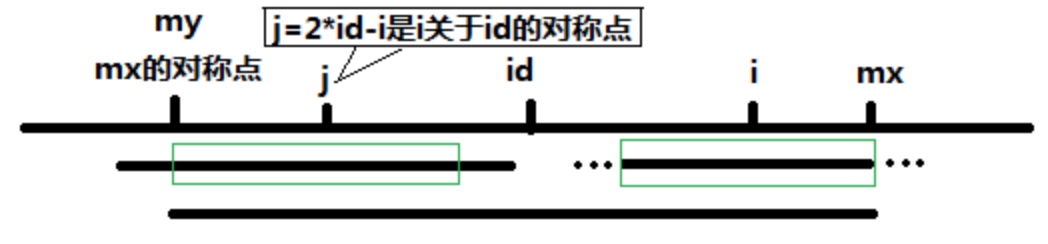
如果 mx > i, 则 p[i] = min( p[2 \* id - i] , mx - i )

否则，p[i] = 1

当 mx - i > P[j] 的时候，以S[j]为中心的回文子串包含在以S[id]为中心的回文子串中，由于 i 和 j 对称，以S[i]为中心的回文子串必然包含在以S[id]为中心的回文子串中，所以必有 P[i] = P[j]，见下图。



当 P[j] >= mx - i 的时候，以S[j]为中心的回文子串不一定完全包含于以S[id]为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以S[i]为中心的回文子串，其向右至少会扩张到mx的位置，也就是说 P[i] >= mx - i。至于mx之后的部分是否对称，就只能老老实实去匹配了。



对于 mx <= i 的情况，无法对 P[i]做更多的假设，只能P[i] = 1，然后再去匹配了。

## 10. The Two Eggs Problem

### 1. Puzzle Definition

You are given two eggs, and access to a 100-storey building. Both eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and does not break, it is undamaged and can be dropped again. However, once an egg is broken, that’s it for that egg.

If an egg breaks when dropped from floor n, then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that.

The question is: What strategy should you adopt to minimize the number egg drops it takes to find the solution? (And what is the worst case for the number of drops it will take?)

There are no tricks, gotchas or other devious ruses. Don’t rat-hole with issues related to terminal velocity, potential energy or wind resistance. This is a math puzzle plain and simple.

### 2. One Egg

Whilst it’s not strictly part of the puzzle, let’s first imagine what we’d do if we had only one egg.

Once this egg is broken, that’s it, no more egg. So, we really have no other choice but to start at floor 1. If it survives, great, we go up to floor 2 and try again, then floor 3 … all the way up the building; one floor at a time. Eventually the egg will break\* and we’ll have a solution. For example, if it breaks on floor 57, we know that the highest floor that an egg can withstand a drop from is floor 56.

There’s no other one egg solution. Sure, if we’d been feeling lucky we could have gone up the floors in two’s but imagine if the egg broke on floor 16; we have no way of knowing if it would have also broken on floor 15!

### 3. Many Eggs

At the other extreme, what if we had an infinite number of eggs? (Or at least as many eggs as we need). What would our strategy be here? In this case we’d use one of a programmer’s favorite tools, the binary tree.

First, we’d go to floor 50 and drop an egg. It either breaks, or it does not. The outcome of this drop instantly cuts our problem in half. If it breaks, we know the solution lives in the bottom half of the building (floor 1 – floor 49). If it survives, we know the solution is in the top half of the building (floor 51 – 100). On each drop, we keep dividing the problem in half and half again until we get to our solution.

The mathematicians in the audience will quickly see that the number of drops required for this solution is log2 n, where n is the number of floors of the building. (This is like asking how many powers of two there are).

Because this building does not have a number of floors equal to a round number power of two, we need to round up to number of drops to get seven ( log2 100 = 6.644 )

(Using seven drops, we could solve this problem any building up to 128 floors. Eight drops would allow us to solve the puzzle for a building with twice the height at 256 floors …)

Depending on the final answer, the actual number of eggs broken using a binary search will vary. At worst it will be seven eggs (if the floor is actually floor 1, since every drop will break the egg). At best it will be no eggs (if the actually answer is floor 100, since the egg will survive every drop made).

### 4. Back to two eggs

OK, let’s get back to the original two egg problem. As we’ve seen from above, the worst case using a binary search would break seven eggs; not acceptable when we only have two eggs.

It does not take much imagination to see why a binary search solution will not work (optimally) for two eggs. Let’s imagine we did try a binary search and dropped our first egg from floor 50. If it broke, we’d be instantly reduced to a one egg problem, so then we would have to start with our last egg from floor 1 and keep going up one floor at a time until that breaks. As a worst case, it will take us 50 drops. We can do better …

What happens if we started off with our first egg going up by floors ten at a time? We can start dropping our egg from floor 10; if it survives we try floor 20, then floor 30 … we carry on until the first egg breaks. Once we’ve broken our first egg we know that the solution must lay somewhere in the nine floors just below, so we back off nine floors and step through these floors one at a time until we find a solution.

This is certainly an improvement, but what is our worst case with this strategy? Well, imagine we dropped eggs on every 10th floor all the way up, and our first egg broke on floor 100. This has taken us ten drops so far. Now we know the solution must lay somewhere between floor 91 and floor 99 and we have to go through these in ones, starting at floor 91. The worst case would be if the solution was on floor 99, and this would take us nine more drops to determine. The worst case therefore, if we go by tens, is 19 drops.

This is a huge improvement, but can we do better?

Thinking about the 10-floor strategy again we can see that, whilst our worst case is 19 drops, some other possible solutions will take less than this (for instance, if the first egg broke on floor 10 then, at worst, from here we only have to make nine more drops to find the solution). Knowing that, what if we dropped our first egg from floor 11 instead? If the egg breaks on this floor, it will still only take ten more drops to find the solution (and if it doesn’t break, great, we’ve eliminated more floors than before! win-win?) Let's follow this idea, and see where it leads …

Well, how about if we dropped our first egg from floor 12 then? A similar argument to above; if it breaks, we only have to try eleven floors with the second egg. If it doesn’t break, we can step up another dozen floors, and so on … But hold on a minute! … If first we try floor 12, then 24, then 36, then 48, 60, 72, 84, 96 then it finally breaks, we’ve wasted eight drops already, and we still have to check (up to) eleven more floors with our second egg, so we’re back at a worst case of 19 drops.

Problems where the solution lays lower down the building are taking less drops than when the solution lays higher up. We need to come up with a strategy that makes things more uniform.

### 5. Minimization of Maximum Regret

What we need is a solution that minimizes our maximum regret. The examples above hint towards what we need is a strategy that tries to make solutions to all possible answers the same depth (same number of drops). The way to reduce the worst case is to attempt to make all cases take the same number of drops.

As I hope you can see by now, if the solution lays somewhere in a floor low down, then we have extra-headroom when we need to step by singles, but, as we get higher up the building, we’ve already used drop chances to get there, so there we have less drops left when we have to switch to going floor-by-floor.

Let’s break out some algebra.Imagine we drop our first egg from floor n, if it breaks, we can step through the previous (n-1) floors one-by-one.

If it doesn’t break, rather than jumping up another n floors, instead we should step up just (n-1) floors (because we have one less drop available if we have to switch to one-by-one floors), so the next floor we should try is floor n + (n-1)

Similarly, if this drop does not break, we next need to jump up to floor n + (n-1) + (n-2), then floor n + (n-1) + (n-2) + (n-3) …

We keep reducing the step by one each time we jump up, until that step-up is just one floor, and get the following equation for a 100 floor building:

n + (n-1) + (n-2) + (n-3) + (n-4) + … + 1 >= 100

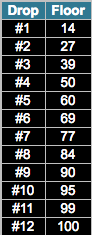
This summation, as many will recognize, is the formula for triangular numbers (which kind of makes sense, since we’re reducing the step by one each drop we make) and can be simplified to:

n (n+1) / 2 >= 100

This is a quadratic equation, with the positive root of 13.651 (Which we have to round up to 14. This is not a John Malkovich movie!).

### 6. Two egg solution

We now have all the information to compute the optimal two egg solution.



Our first drop should be from floor 14, if egg survives we step up 13 floors to floor 27, then up 12 floors to 39 …

The optimal strategy is to work our way through the table until the first egg breaks, then back up to one floor higher than the line above and then proceed floor-by-floor until we find the exact solution.

The maximum of 14 drops is a combination first egg drops (made in steps), and second egg drops (made in ones). For every drop we take hopping up the tower, we reduce the worst-case number of single drops we’d have to take so that no solution is an outlier.

### 7. Look see I can do three

Why stop at two eggs? What is the optimal strategy if we had three eggs? (Dare I say four? five? …)

Things get more complex with three eggs, but taking a deep breath we can apply the same principle of minimizing maximum regret. We need to select our first egg such that, if it breaks, or does not break, it results in a problem, which recursively is now a two-egg problem, that we already know how to solve to minimize maximum regret!

Let's define our building to have a maximum of k floors. Let's say that the number of eggs we have is represented by e, and the floor we are currently attempting to drop from is represented by n. With me so far? OK, what we need to find our optimal solution is to calculate what would happen if we dropped an egg from every floor (1 through to k) and (recursively) calculate the minimum number of droppings needed in the worst case. We're looking for the floor that gives the minimum solution to the worst case.

If we drop an egg from floor n, one of two events happen:

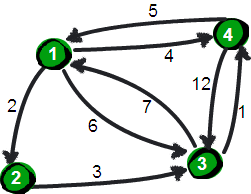
1. The egg breaks, so now our problem is reduced to a tower of height n-1, and we now have e-1 eggs.
2. The egg doesn't break and now we need to check k-n floors and we still have e eggs.

Remember we need to mimimize the number of drops in the worst case, so we take the higher (max) of these two situations, and select the floor which yields the minimum number of drops.

The code to solve this is fairly easy to write recursively, but suffers from a common problem that occurs with recursive solutions in which the same sub-problems are evaluated again, and again, and again, dragging performance to a grind for anything other than trivial solutions. To get around this, we need to keep track of values already computed so that we don't have to repeat the calculation.

## 11. Floyd-Warshall多源最短路径算法

暑假，小哼准备去一些城市旅游。有些城市之间有公路，有些城市之间则没有，如下图。为了节省经费以及方便计划旅程，小哼希望在出发之前知道任意两个城市之前的最短路程。



上图中有 4 个城市 8 条公路，公路上的数字表示这条公路的长短。请注意这些公路是单向的。我们现在需要求任意两个城市之间的最短路程，也就是求任意两个点之间的最短路径。这个问题这也被称为“多源最短路径”问题。

现在需要一个数据结构来存储图的信息，我们仍然可以用一个 4\*4 的矩阵（二维数组 e）来存储。比如 1 号城市到 2 号城市的路程为 2，则设 e[1][2]的值为 2。2 号城市无法到达 4 号城市，则设置 e[2][4]的值为 ∞。另外此处约定一个城市自己是到自己的也是 0，例如 e[1][1]为 0，具体如下。



现在回到问题：如何求任意两点之间最短路径呢？通过之前的学习我们知道通过深度或广度优先搜索可以求出两点之间的最短路径。所以进行 n2 遍深度或广度优先搜索，即对每两个点都进行一次深度或广度优先搜索，便可以求得任意两点之间的最短路径。可是还有没有别的方法呢？

我们来想一想，根据我们以往的经验，如果要让任意两点（例如从顶点 a 点到顶点 b）之间的路程变短，只能引入第三个点（顶点 k），并通过这个顶点 k 中转即 a->k->b，才可能缩短原来从顶点 a 点到顶点 b 的路程。那么这个中转的顶点 k 是 1~n 中的哪个点呢？甚至有时候不只通过一个点，而是经过两个点或者更多点中转会更短，即 a->k1->k2b->或者 a->k1->k2…->k->i…->b。比如上图中从 4 号城市到 3 号城市（4->3）的路程 e[4][3]原本是 12。如果只通过 1 号城市中转（4->1->3），路程将缩短为 11（e[4][1]+e[1][3]=5+6=11）。其实 1 号城市到 3 号城市也可以通过 2 号城市中转，使得 1 号到 3 号城市的路程缩短为 5（e[1][2]+e[2][3]=2+3=5）。所以如果同时经过 1 号和 2 号两个城市中转的话，从 4 号城市到 3 号城市的路程会进一步缩短为 10。通过这个的例子，我们发现每个顶点都有可能使得另外两个顶点之间的路程变短。好，下面我们将这个问题一般化。

当任意两点之间不允许经过第三个点时，这些城市之间最短路程就是初始路程，如下。



如现在只允许经过 1 号顶点，求任意两点之间的最短路程，应该如何求呢？只需判断 e[i][1]+e[1][j]是否比 e[i][j]要小即可。e[i][j]表示的是从 i 号顶点到 j 号顶点之间的路程。e[i][1]+e[1][j]表示的是从 i 号顶点先到 1 号顶点，再从 1 号顶点到 j 号顶点的路程之和。其中 i 是 1~n 循环，j 也是 1~n 循环，代码实现如下。

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

{

for(j=1;j<=n;j++)

{

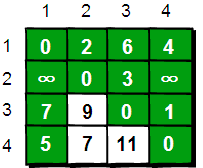
if ( e[i][j] > e[i][1]+e[1][j] )

e[i][j] = e[i][1]+e[1][j];

}

}

在只允许经过 1 号顶点的情况下，任意两点之间的最短路程更新为：



通过上图我们发现：在只通过 1 号顶点中转的情况下，3 号顶点到 2 号顶点（e[3][2]）、4 号顶点到 2 号顶点（e[4][2]）以及 4 号顶点到 3 号顶点（e[4][3]）的路程都变短了。

接下来继续求在只允许经过 1 和 2 号两个顶点的情况下任意两点之间的最短路程。如何做呢？我们需要在只允许经过 1 号顶点时任意两点的最短路程的结果下，再判断如果经过 2 号顶点是否可以使得 i 号顶点到 j 号顶点之间的路程变得更短。即判断 e[i][2]+e[2][j]是否比 e[i][j]要小，代码实现为如下。

//经过1号顶点

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

for(j=1;j<=n;j++)

if (e[i][j] > e[i][1]+e[1][j]) e[i][j]=e[i][1]+e[1][j];

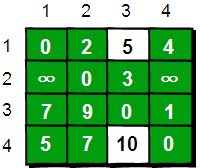
//经过2号顶点

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

for(j=1;j<=n;j++)

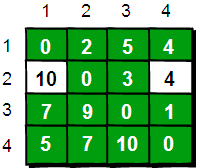
if (e[i][j] > e[i][2]+e[2][j]) e[i][j]=e[i][2]+e[2][j];

在只允许经过 1 和 2 号顶点的情况下，任意两点之间的最短路程更新为：

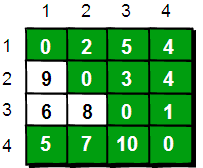


通过上图得知，在相比只允许通过 1 号顶点进行中转的情况下，这里允许通过 1 和 2 号顶点进行中转，使得 e[1][3]和 e[4][3]的路程变得更短了。

同理，继续在只允许经过 1、2 和 3 号顶点进行中转的情况下，求任意两点之间的最短路程。任意两点之间的最短路程更新为：



最后允许通过所有顶点作为中转，任意两点之间最终的最短路程为：



整个算法过程虽然说起来很麻烦，但是代码实现却非常简单，核心代码只有五行：

for(k=1;k<=n;k++)

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

for(j=1;j<=n;j++)

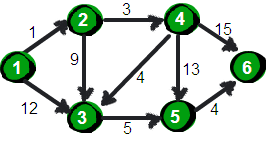
if(e[i][j]>e[i][k]+e[k][j])

e[i][j]=e[i][k]+e[k][j];

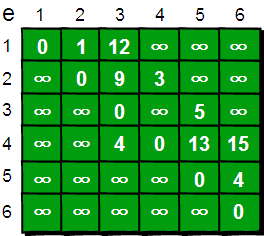
这段代码的基本思想就是：最开始只允许经过 1 号顶点进行中转，接下来只允许经过 1 和 2 号顶点进行中转……允许经过 1~n 号所有顶点进行中转，求任意两点之间的最短路程。用一句话概括就是：从 i 号顶点到 j 号顶点只经过前k号点的最短路程。

## 12. Dijkstra 单源最短路算法

之前我们介绍了神奇的只有五行的 Floyd 最短路算法，它可以方便的求得任意两点的最短路径，这称为“多源最短路”。本周来来介绍指定一个点（源点）到其余各个顶点的最短路径，也叫做“单源最短路径”。例如求下图中的 1 号顶点到 2、3、4、5、6 号顶点的最短路径。



与 Floyd-Warshall 算法一样这里仍然使用二维数组 e 来存储顶点之间边的关系，初始值如下。



我们还需要用一个一维数组 dis 来存储 1 号顶点到其余各个顶点的初始路程，如下。

picture7.3

我们将此时 dis 数组中的值称为最短路的“估计值”。

既然是求 1 号顶点到其余各个顶点的最短路程，那就先找一个离 1 号顶点最近的顶点。通过数组 dis 可知当前离 1 号顶点最近是 2 号顶点。当选择了 2 号顶点后，dis[2]的值就已经从“估计值”变为了“确定值”，即 1 号顶点到 2 号顶点的最短路程就是当前 dis[2]值。为什么呢？你想啊，目前离 1 号顶点最近的是 2 号顶点，并且这个图所有的边都是正数，那么肯定不可能通过第三个顶点中转，使得 1 号顶点到 2 号顶点的路程进一步缩短了。因为 1 号顶点到其它顶点的路程肯定没有 1 号到 2 号顶点短，对吧 O(∩\_∩)O~

既然选了 2 号顶点，接下来再来看 2 号顶点有哪些出边呢。有 2->3 和 2->4 这两条边。先讨论通过 2->3 这条边能否让 1 号顶点到 3 号顶点的路程变短。也就是说现在来比较 dis[3]和 dis[2]+e[2][3]的大小。其中 dis[3]表示 1 号顶点到 3 号顶点的路程。dis[2]+e[2][3]中 dis[2]表示 1 号顶点到 2 号顶点的路程，e[2][3]表示 2->3 这条边。所以 dis[2]+e[2][3]就表示从 1 号顶点先到 2 号顶点，再通过 2->3 这条边，到达 3 号顶点的路程。

我们发现 dis[3]=12，dis[2]+e[2][3]=1+9=10，dis[3]>dis[2]+e[2][3]，因此 dis[3]要更新为 10。这个过程有个专业术语叫做“松弛”。即 1 号顶点到 3 号顶点的路程即 dis[3]，通过 2->3 这条边松弛成功。这便是 Dijkstra 算法的主要思想：通过“边”来松弛 1 号顶点到其余各个顶点的路程。

同理通过 2->4（e[2][4]），可以将 dis[4]的值从 ∞ 松弛为 4（dis[4]初始为 ∞，dis[2]+e[2][4]=1+3=4，dis[4]>dis[2]+e[2][4]，因此 dis[4]要更新为 4）。

刚才我们对 2 号顶点所有的出边进行了松弛。松弛完毕之后 dis 数组为：

picture7.4

接下来，继续在剩下的 3、4、5 和 6 号顶点中，选出离 1 号顶点最近的顶点。通过上面更新过 dis 数组，当前离 1 号顶点最近是 4 号顶点。此时，dis[4]的值已经从“估计值”变为了“确定值”。下面继续对 4 号顶点的所有出边（4->3，4->5 和 4->6）用刚才的方法进行松弛。松弛完毕之后 dis 数组为：

picture7.5

继续在剩下的 3、5 和 6 号顶点中，选出离 1 号顶点最近的顶点，这次选择 3 号顶点。此时，dis[3]的值已经从“估计值”变为了“确定值”。对 3 号顶点的所有出边（3->5）进行松弛。松弛完毕之后 dis 数组为：

picture7.6

继续在剩下的 5 和 6 号顶点中，选出离 1 号顶点最近的顶点，这次选择 5 号顶点。此时，dis[5]的值已经从“估计值”变为了“确定值”。对5号顶点的所有出边（5->4）进行松弛。松弛完毕之后 dis 数组为：

picture7.7

最后对 6 号顶点所有点出边进行松弛。因为这个例子中 6 号顶点没有出边，因此不用处理。到此，dis 数组中所有的值都已经从“估计值”变为了“确定值”。

最终 dis 数组如下，这便是 1 号顶点到其余各个顶点的最短路径。

picture7.8

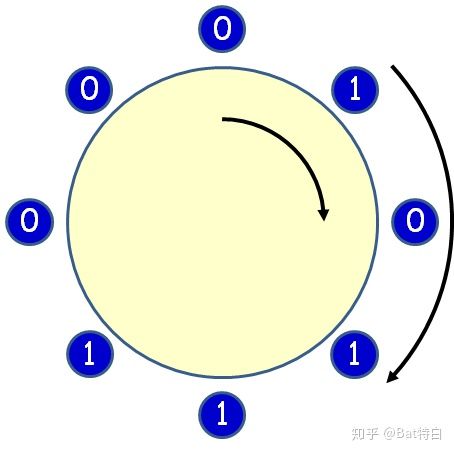
OK，现在来总结一下刚才的算法。算法的基本思想是：每次找到离源点（上面例子的源点就是 1 号顶点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。基本步骤如下：

* 将所有的顶点分为两部分：已知最短路程的顶点集合 P 和未知最短路径的顶点集合 Q。最开始，已知最短路径的顶点集合 P 中只有源点一个顶点。我们这里用一个 book[ i ]数组来记录哪些点在集合 P 中。例如对于某个顶点 i，如果 book[ i ]为 1 则表示这个顶点在集合 P 中，如果 book[ i ]为 0 则表示这个顶点在集合 Q 中。
* 设置源点 s 到自己的最短路径为 0 即 dis=0。若存在源点有能直接到达的顶点 i，则把 dis[ i ]设为 e[s][ i ]。同时把所有其它（源点不能直接到达的）顶点的最短路径为设为 ∞。
* 在集合 Q 的所有顶点中选择一个离源点 s 最近的顶点 u（即 dis[u]最小）加入到集合 P。并考察所有以点 u 为起点的边，对每一条边进行松弛操作。例如存在一条从 u 到 v 的边，那么可以通过将边 u->v 添加到尾部来拓展一条从 s 到 v 的路径，这条路径的长度是 dis[u]+e[u][v]。如果这个值比目前已知的 dis[v]的值要小，我们可以用新值来替代当前 dis[v]中的值。
* 重复第 3 步，如果集合 Q 为空，算法结束。最终 dis 数组中的值就是源点到所有顶点的最短路径

## 13. 德布鲁因序 (De Bruijn Sequence)

### 13.1 存储器轮

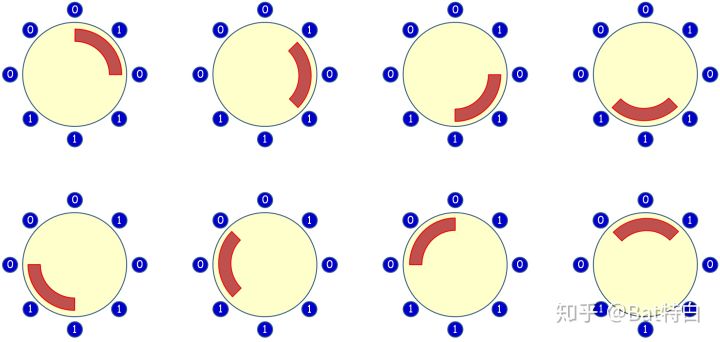
考虑如下图所示，由4个0和4个1组成的一个圆形排列：



每次顺时针连续读三个数字的话，一共有8种方法（8个开始位置）。

上图所示的情况延内侧弧线读出的是010，而延外侧弧线读出的是101。

而这8种读法就恰好包含了所有长度为3的0-1序列：（依次是）010、101、011、111、110、111、100、000（下一次又回到了010）。



于是这个圆周可以描述所有8个3位0-1串，而仅需要8位0-1数字。

我们称其为（3位）**存储器轮**。

下面我们将这个问题一般化：能否将**若干个**二进制数字0或1排列成圆，使得所有的相邻的n元组（即指顺时针连续读n个数字）包括所有的2^个n位二进制序列**一次且仅一次**?

我们首先分析一些基本结果：

**问题1：**上面提到的“**若干个**”到底是多少个？

**回答1: 2^n**个。

考虑2^n个n位二进制序列的第一位，一定是在彼此不同的位置，而且恰好填满圆周。

**问题2：**这2^n个0或1中，有多少个0？多少个1？

**回答2：**各有2^(n-1)个。

考虑2^(n-1)个n位二进制序列的第一位，一定是一半0一半1，于是圆周上也一定是一半0一半1。

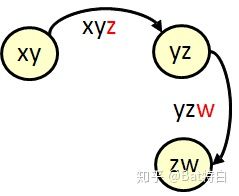
**问题3：**能否将2^(n-1)个0和2^(n-1)个1排列成圆，使得所有的相邻的n元组（即指顺时针连续读n个数字）包括所有的  个  位二进制序列**一次且仅一次**?

**分析3 （方法2）**：1946年，古德（I.J.Good）在一篇关于数论的论文中，使用不同的数学模型解决了这个问题。

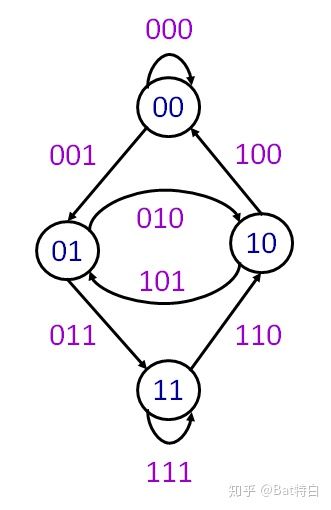
以**n = 3**为例，将三元组*xyz*作为顶点 “*xy*” 到顶点 “*yz*” 的有向边。



考虑“相邻”的两个三元组，必然是“*xyz*”和“*yzw*”的形式，将用顶点“*yz*”连接。

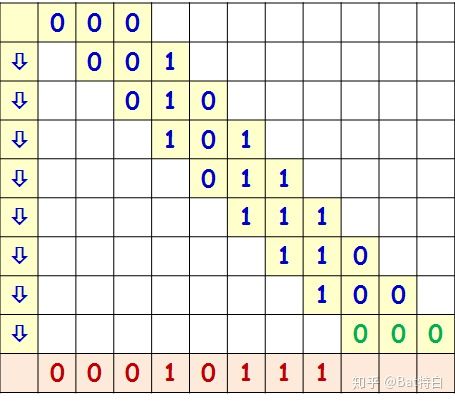


于是，这时就形成一个有向图：



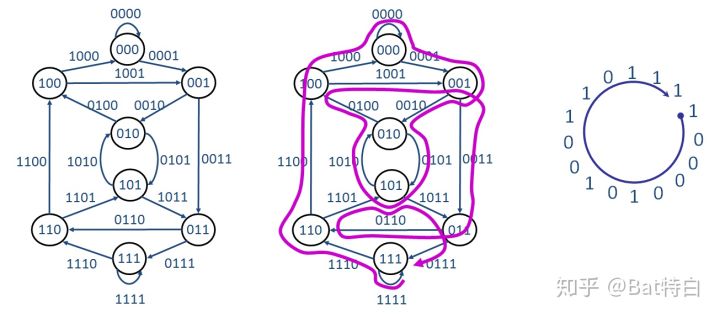
因为它所有顶点的入度都等于出度，所以它是一个**有向欧拉图**。

其中一条欧拉回路是：



得到同样的存储器轮。

**N = 4**时，下左图中存在欧拉回路（下中图所示），由之可以得到下右图所示的四位存储器轮。



结论：当n >= 3时（n = 1, 2的情况手绘都可以得到存储器轮），问题就转换为在一个**有向图中寻找欧拉回路**。

而有向图存在欧拉回路的充要条件是：

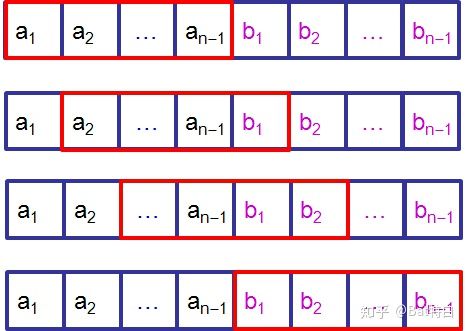
* 有向图连通；
* 每个顶点的入度等于出度。

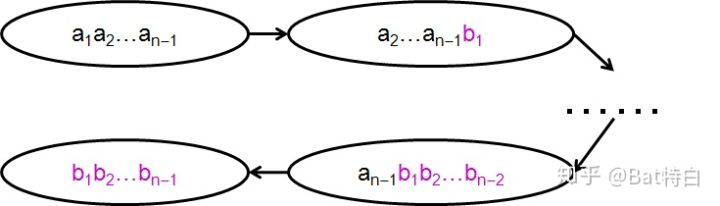
(1) 在任意两个顶点a1a2…an和b1b2...bn之间，是否存在道路？

答案是**肯定**的。

https://pic2.zhimg.com/80/v2-742885cd83b376deef8de20c8a684905_hd.jpg

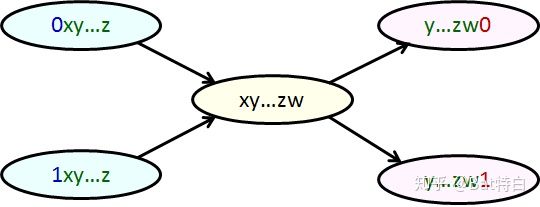
再使用一个大小为n - 1的“窗口”从左一直走到右，就得到了顶点a1a2…an和b1b2…bn之间的有向道路





(2) 每个顶点的**入度和出度都是2**。

考察某个顶点“xy…zw” ，由图的构造方法很容易得到它有2条入边和2条出边。

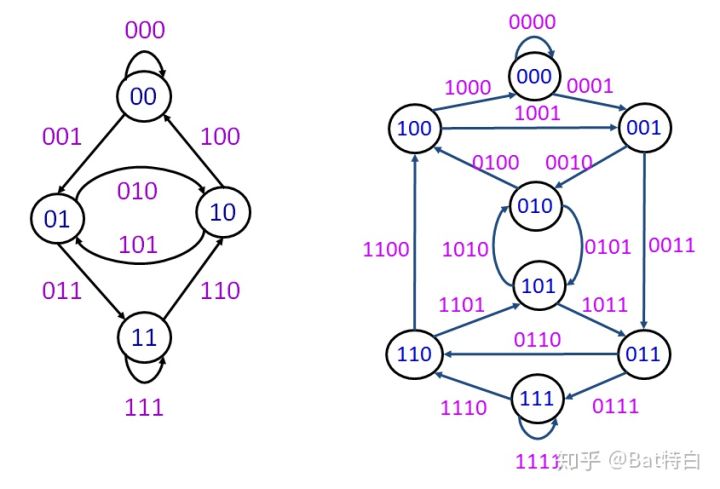


因此所有顶点的入度和出度都等于2。

由以上二者可以得到：**该图是欧拉图，每一条欧拉回路对应一个存储器轮**

### 13.2 德布鲁因有向图和德布鲁因序列

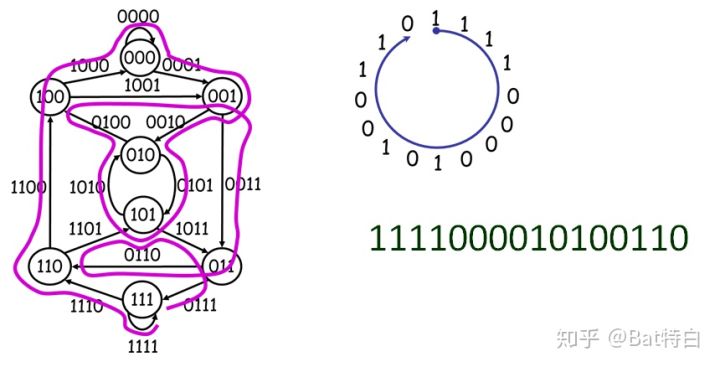
在上一章“[存储器轮](https://zhuanlan.zhihu.com/p/43960965)”中，曾经引入了两个图：



事实上，这样的图的顶点集是

从任意顶点x1x2…xn到2个其他顶点x2x3…xn0和x2x3...xn1分别各有一条有向边，它们称作**德•布鲁因（de Bruijn）有向图**B(2, n)。

**构造方法1**：由B（2，n）中欧拉回路得到的0-1串称作长度2^n的**德·布鲁因（de Bruijn）序列**。例如由下图左得到长度16的德·布鲁因序列1111011001010000。



**构造方法2**：马丁（M.A.Martin）1934证明了以下贪婪算法对所有的n>=2都可以构造出一个长度2^n的德·布鲁因序列:

* 写出n个0构成序列开始的n项;
* 如果在序列尾部添加一个1后，和前n-1项相连构成已经出现过的长为n的0-1子串，则在序列尾部添加一个0；否则在序列尾部添加一个1。
* 序列若还不够2^n项，则返回步骤2；否则序列就是一个长度2^n的德·布鲁因序列。

N = 3时算法过程的情况如下表所示：



n = 4时算法过程的情况如下表所示：



### 13.3 德布鲁因序列的应用

德布鲁因序列相当于是全排列，枚举了所有的情况。所以它的两两子序列之间肯定不相同，这就是完美的哈希。

样题：753 Cracking the Safe

## 14. 水塘抽样（Reservoir Sampling）

在大数据流中的随机抽样问题，即：当内存无法加载全部数据时，如何从包含未知大小的数据流中随机选取k个数据，并且要保证每个数据被抽取到的概率相等;

### Problem Statement

Reservoir Sampling is an algorithm for sampling elements from a stream of data. Imagine you are given a really large stream of data elements, for example:

* Queries on google searches in May
* Products bought at Walmart during the Christmas season
* Names in a phone book

Your goal is to efficiently return a random sample of 1,000 elements evenly distributed from the original stream. How would you do it?

The right answer is generating random integers between **0** and **N - 1**, then retrieving the elements at those indices and you have your answer. If you need to generate unique elements, then just throw away indices you've already generated.

So, let me make the problem harder. You don't know **N** (the size of the stream) in advance and you can't index directly into it. You can count it, but that requires making 2 passes of the data. You can do better. There are some heuristics you might try: for example, to guess the length and hope to undershoot. It will either not work in one pass or will not be evenly distributed.

### Simple Solution

A relatively easy and correct solution is to assign a random number to every element as you see it in the stream, and then always keep the top 1,000 numbered elements at all times. This is similar to how a SQL Query with ORDER BY RAND() works. This strategy works well, and only requires additionally storing the randomly generated number for each element.

### Reservoir Sampling

Another, more complex option is **reservoir sampling**. First, you want to make a reservoir (array) of 1,000 elements and fill it with the first 1,000 elements in your stream. That way if you have exactly 1,000 elements, the algorithm works. This is the base case.

Next, you want to process the **i'th** element (starting with **i = 1001**) such that at the end of processing that step, the 1,000 elements in your reservoir are randomly sampled amongst the **i** elements you've seen so far.

How can you do this? Start with **i = 1001**. With what probability after the **1001'th** step whould element 1,001 (or any element for that matter) be in the set of 1,000 elements? The answer is easy 1000/1001. (C(999,1000) / C(1000,1001)) Generate a random number between 0 and 1, and if it is less than 1000/1001 you should take element **1,001**. In other words, choose to add element **1,001** to your reservoir with probability 1000/1001. If you choose to add it, then replace any element in the reservoir chosen randomly.

We've seen that this produces a 1000/1001 chance of selecting the **1,001'th** element, but what about the **2nd** element in the list? Since it's already in the reservoir, so the probability of removing it is the probability of **1,001** element selected multiplied by the probability of **2nd** getting randomly chosen as the replacement candidate. The probability is thus :

1000/1001 \* 1/1000 = 1/1001

So, the probability that the **2nd** element survives this round is :

1 - 1/1001 = 1000/1001

That's exactly what we want.

|  |
| --- |
| // A function to randomly select k items from stream[0..n-1].  public static void selectKItems(int stream[], int n, int k) {  // index for elements in stream[]  int i;  // first initialize the reservoir  int reservoir[] = new int[k];  for (i = 0; i < k; ++i)  reservoir[i] = stream[i];  Random r = new Random();  for (; i < n; ++i) {  // pick a random index from 0 to i [0, i]  int j = r.nextInt(i + 1);  // If the randomly picked index is smaller than k,  // then replace the element present at the index  // with new element from stream  if (j < k)  reservoir[j] = stream[i];  }  } |

This can be extended for the i'th round: keep the i'th element with probability 1000/i and if it is kept, replace a random element from the reservoir.

It is pretty easy to prove that this works for all values of i using induction: It obviously works for the i'th element based on the way the algorithm selects the i'th element with the correct probability outright. The probability any element before this step being in the reservoir is 1000/i-1. The probability that they are removed is the probability that the 'th element is selected multiplied by the probability that the prior element was selected to drop from the reservoir:

1000/i \* 1/1000 = 1/i

This is the probability that a given element will be removed in that round given that it has made it to the reservoir so far. The probabilty that it isn't removed is the inverse. The final probability that it survives to the i'th round is the probability that it survived to the previous round multiplied by the probability that it isn't removed:

1000/i-1 \* (1 - 1/i) = 1000/i-1 \* i-1/i = 1000/i

This is the probability we want.

对于1000以后的点，被选中，后备预留的概率： 1000/i

对于1000以前的点，被选中，将被替换走的概率： 1/1000

一个1000以后的点被选中并保留到水塘里的概率： 1000/i \* 1/1000 = 1/i

同理，一个1000以前的点存活下来的概率是： 1- 1/i = i-1/i

对于1000以前的点，如果在i轮之后还能幸存在水塘里的概率计算如下：

i-1轮活下的概率 \* i轮没有被选中的概率

1000/i-1 \* i-1/I = 1000/i

可见这个理论是可行的；

### 应用题目

L384 Shuffle an Array

L382 Linked List Random Node

L398 Random Pick Index

## 15. 内排序

### 1. 非基于比较的排序

在计算机科学中，排序是一门基础的算法技术，许多算法都要以此作为基础，不同的排序算法有着不同的时间开销和空间开销。排序算法有非常多种，如我们最常用的快速排序和堆排序等算法，这些算法需要对序列中的数据进行比较，因为被称为基于比较的排序。

**基于比较的排序算法是不能突破O(NlogN)的**。简单证明如下：N个数有N!个可能的排列情况，也就是说基于比较的排序算法的判定树有N!个叶子结点，比较次数至少为log(N!)=O(NlogN)(斯特林公式)。

而**非基于比较的排序**，如计数排序，桶排序，和在此基础上的基数排序，则可以突破O(NlogN)时间下限。但要注意的是，非基于比较的排序算法的使用都是有条件限制的，例如元素的大小限制，相反，基于比较的排序则没有这种限制(在一定范围内)。但并非因为有条件限制就会使非基于比较的排序算法变得无用，对于特定场合有着特殊的性质数据，非基于比较的排序算法则能够非常巧妙地解决。

### 2. 计数排序

首先从**计数排序(Counting Sort)**开始介绍起，假设我们有一个待排序的整数序列A，其中元素的最小值不小于0，最大值不超过K。建立一个长度为K的线性表C，用来记录不大于每个值的元素的个数。

算法思路如下：

* 扫描序列A，以A中的每个元素的值为索引，把出现的个数填入C中。此时C[i]可以表示A中值为i的元素的个数。
* 对于C从头开始累加，使C[i]<-C[i]+C[i-1]。这样，C[i]就表示A中值**不大于i的元素的个数**。
* 按照统计出的值，输出结果。

由线性表C我们可以很方便地求出排序后的数据，定义B为目标的序列，Order[i]为排名第i的元素在A中的位置，则可以用以下方法统计。

|  |
| --- |
| void CountingSort(int \*A,int \*B,int \*Order,int N,int K)  {  int \*C = new int[K+1];  int i;  memset(C,0,sizeof(int)\*(K+1));  for (i=1;i<=N;i++) //把A中的每个元素分配  C[A[i]]++;  for (i=2;i<=K;i++) //统计不大于i的元素的个数  C[i]+=C[i-1];  for (i=N;i>=1;i--)  {  B[C[A[i]]]=A[i]; //按照统计的位置，将值输出到B中，将顺序输出到Order中  Order[C[A[i]]]=i;  C[A[i]]--;  }  }  int main()  {  int \*A,\*B,\*Order,N=15,K=10,i;  A=new int[N+1], B=new int[N+1], Order=new int[N+1];  for (i=1;i<=N;i++)  A[i]=rand()%K+1; //生成1..K的随机数  printf("Before CS:\n");  for (i=1;i<=N;i++)  printf("%d ",A[i]);  CountingSort(A,B,Order,N,K);  printf("\nAfter CS:\n");  for (i=1;i<=N;i++)  printf("%d ",B[i]);  printf("\nOrder:\n");  for (i=1;i<=N;i++)  printf("%d ",Order[i]);  return 0;  }  程序运行效果如下：  Before CS: 2 8 5 1 10 5 9 9 3 5 6 6 2 8 2  After CS: 1 2 2 2 3 5 5 5 6 6 8 8 9 9 10  Order: 4 1 13 15 9 3 6 10 11 12 2 14 7 8 5 |

显然地，计数排序的时间复杂度为O(N+K)，空间复杂度为O(N+K)。当K不是很大时，这是一个很有效的线性排序算法。更重要的是，它是一种**稳定排序算法**，即排序后的相同值的元素原有的相对位置不会发生改变(表现在Order上)，这是计数排序很重要的一个性质，就是根据这个性质，我们才能把它应用到基数排序。

### 3. 桶排序

可能你会发现，计数排序似乎饶了点弯子，比如当我们刚刚统计出C，C[i]可以表示A中值为i的元素的个数，此时我们直接顺序地扫描C，就可以求出排序后的结果。的确是这样，不过这种方法不再是计数排序，而是**桶排序(Bucket Sort)**，确切地说，是桶排序的一种特殊情况；

用这种方法，可以很容易写出程序，比计数排序还简单，只是不能求出稳定的Order。

|  |
| --- |
| void BucketSort(int \*A,int \*B,int N,int K)  {  int \*C=new int[K+1];  int i,j,k;  memset(C,0,sizeof(int)\*(K+1));  for (i=1;i<=N;i++) //把A中的每个元素按照值放入桶中  C[A[i]]++;  for (i=j=1;i<=K;i++,j=k) //统计每个桶元素的个数，并输出到B  for (k=j;k<j+C[i];k++)  B[k]=i;  }  int main()  {  int \*A,\*B,N=1000,K=1000,i;  A=new int[N+1];  B=new int[N+1];  for (i=1;i<=N;i++)  A[i]=rand()%K+1; //生成1..K的随机数  BucketSort(A,B,N,K);  for (i=1;i<=N;i++)  printf("%d ",B[i]);  return 0;  } |

种特殊实现的方式时间复杂度为O(N+K)，空间复杂度也为O(N+K)，同样要求每个元素都要在K的范围内。更一般的，如果我们的K很大，无法直接开出O(K)的空间该如何呢？

首先定义桶，桶为一个数据容器，每个桶存储一个区间内的数。依然有一个待排序的整数序列A，元素的最小值不小于0，最大值不超过K。假设我们有M个桶，第i个桶Bucket[i]存储iK/M至(i+1)K/M之间的数，有如下桶排序的一般方法：

* 扫描序列A，根据每个元素的值所属的区间，放入指定的桶中(顺序放置)。
* 对每个桶中的元素进行排序，什么排序算法都可以，例如快速排序（但是如果做快速排序，则不再具有稳定性），如果强调稳定性，则每个桶内使用链表结构，桶内插入元素时，遍历链表找到合适的位置插入；

注意：此处桶内链表搜索查找相应插入位置：如果没有找到，则找的是应该插入的位置；如果已经存在相同值，则应该找的是相同值中的最后一个位置，实现稳定性

* 依次收集每个桶中的元素，顺序放置到输出序列中。

对该算法简单分析，如果数据是期望平均分布的，则每个桶中的元素平均个数为N/M。如果对每个桶中的元素排序使用的算法是快速排序(或者链表结构里做二分搜索查找应该出入的下一个位置)，每次排序的时间复杂度为O(N/Mlog(N/M))。则总的时间复杂度为O(N)+O(M)O(N/Mlog(N/M)) = O(N+ Nlog(N/M)) = O(N + NlogN - NlogM)。当M接近于N是，桶排序的时间复杂度就可以近似认为是O(N)的。就是桶越多，时间效率就越高，而桶越多，空间却就越大，由此可见时间和空间是一个矛盾的两个方面。

**桶中元素的顺序放入和顺序取出是有必要的，因为这样可以确定桶排序是一种稳定排序算法，桶内结构可以使用链表结构，入桶数据需要遍历链表寻找和是的位置并插入，这个实现方式配合基数排序是很好用的。**

|  |
| --- |
| struct linklist  {  linklist \*next;  int value;  linklist(int v,linklist \*n):value(v),next(n){}  ~linklist() {if (next) delete next;}  };  inline int cmp(const void \*a,const void \*b)  {  return \*(int \*)a-\*(int \*)b;  }  /\*  为了方便，我把A中元素加入桶中时是倒序放入的，而收集取出时也是倒序放入序列的，所以不违背稳定排序。  \*/  void BucketSort(int \*A,int \*B,int N,int K)  {  linklist \*Bucket[101],\*p;//建立桶  int i,j,k,M;  M=K/100;  memset(Bucket,0,sizeof(Bucket));  for (i=1;i<=N;i++)  {  k=A[i]/M; //把A中的每个元素按照的范围值放入对应桶中  Bucket[k]=new linklist(A[i],Bucket[k]);  }  for (k=j=0;k<=100;k++)  {  i=j;  for (p=Bucket[k];p;p=p->next)  B[++j]=p->value; //把桶中每个元素取出，排序并加入B  delete Bucket[k];  qsort(B+i+1,j-i,sizeof(B[0]),cmp);  }  }  int main()  {  int \*A,\*B,N=100,K=10000,i;  A=new int[N+1];  B=new int[N+1];  for (i=1;i<=N;i++)  A[i]=rand()%K+1; //生成1..K的随机数  BucketSort(A,B,N,K);  for (i=1;i<=N;i++)  printf("%d ",B[i]);  return 0;  } |

### 4. 基数排序

下面说到我们的重头戏，**基数排序(Radix Sort)**。上述的基数排序和桶排序都只是在研究一个关键字的排序，现在我们来讨论有**多个关键字的排序**问题。

假设我们有一些二元组(a,b)，要对它们进行以a为首要关键字，b的次要关键字的排序。我们可以先把它们先按照首要关键字排序，分成首要关键字相同的若干堆。然后，在按照次要关键值分别对每一堆进行单独排序。最后再把这些堆串连到一起，使首要关键字较小的一堆排在上面。按这种方式的基数排序称为**MSD(Most Significant Dight)排序**。

第二种方式是从最低有效关键字开始排序，称为**LSD(Least Significant Dight)排序**。首先对所有的数据按照次要关键字排序，然后对所有的数据按照首要关键字排序。要注意的是，使用的排序算法必须是稳定的，否则就会取消前一次排序的结果。由于不需要分堆对每堆单独排序，LSD方法往往比MSD简单而开销小。下文介绍的方法全部是基于LSD的。

**通常，基数排序要用到计数排序或者桶排序。使用计数排序时，需要的是Order数组。使用桶排序时，可以用链表的方法直接求出排序后的顺序，实现了稳定性**。下面是一段用桶排序对二元组基数排序的程序：

|  |
| --- |
| struct data  {  int key[2];  };  struct linklist  {  linklist \*next;  data value;  linklist(data v,linklist \*n):value(v),next(n){}  ~linklist() {if (next) delete next;}  };  void BucketSort(data \*A,int N,int K,int y)  {  linklist \*Bucket[101],\*p;//建立桶  int i,j,k,M;  M=K/100+1;  memset(Bucket,0,sizeof(Bucket));  for (i=1;i<=N;i++)  {  k=A[i].key[y]/M; //把A中的每个元素按照的范围值放入对应桶中  Bucket[k]=new linklist(A[i],Bucket[k]);  }  for (k=j=0;k<=100;k++)  {  for (p=Bucket[k];p;p=p->next) j++;  for (p=Bucket[k],i=1;p;p=p->next,i++)  A[j-i+1]=p->value; //把桶中每个元素取出  delete Bucket[k];  }  }  void RadixSort(data \*A,int N,int K)  {  for (int j=1;j>=0;j--) //从低优先到高优先 LSD  BucketSort(A,N,K,j);  }  int main()  {  int N=100,K=1000,i;  data \*A=new data[N+1];  for (i=1;i<=N;i++)  {  A[i].key[0]=rand()%K+1;  A[i].key[1]=rand()%K+1;  }  RadixSort(A,N,K);  for (i=1;i<=N;i++)  printf("(%d,%d) ",A[i].key[0],A[i].key[1]);  printf("\n");  return 0;  } |

基数排序是一种用在老式穿卡机上的算法。一张卡片有80列，每列可在12个位置中的任一处穿孔。排序器可被机械地"程序化"以检查每一迭卡片中的某一列，再根据穿孔的位置将它们分放12个盒子里。这样，操作员就可逐个地把它们收集起来。其中第一个位置穿孔的放在最上面，第二个位置穿孔的其次，等等。

对于一个位数有限的十进制数，我们可以把它看作一个多元组，从高位到低位关键字重要程度依次递减。**可以使用基数排序对一些位数有限的十进制数排序。**

### 5. 线性排序比较

从整体上来说，计数排序，桶排序都是非基于比较的排序算法，而其时间复杂度依赖于数据的范围，桶排序还依赖于空间的开销和数据的分布。而基数排序是一种对多元组排序的有效方法，具体实现要用到计数排序或桶排序。

相对于快速排序、堆排序等基于比较的排序算法，计数排序、桶排序和基数排序限制较多，不如快速排序、堆排序等算法灵活性好。但反过来讲，这三种线性排序算法之所以能够达到线性时间，是因为充分利用了待排序数据的特性，如果生硬得使用快速排序、堆排序等算法，就相当于浪费了这些特性，因而达不到更高的效率。

在实际应用中，基数排序可以用于后缀数组的倍增算法，使时间复杂度从O(NlogNlogN)降到O(N\*logN)。线性排序算法使用最重要的是，**充分利用数据特殊的性质，以达到最佳效果。**

### 6. 编码注意事项

Number Range : 0 - N

Bucket number: k

Space for each bucket: (N + 1)/k

Please be sure to create "k+1" buckets!!!

桶排序

* 确定桶的索引： num/space => bucket index
* 桶内寻找索引算法需要找到大于目标值的第一个索引，即二分查找上界；

二分查找时比较数值

基数排序

* 确定桶的索引：

space = (key range + 1)/k

key value / space => bucket index

* 桶内寻找索引算法需要找到大于目标值的第一个索引，即二分查找上界；

二分查找时比较 key的值

# Appendix 网络资源

== 入门 ==

对于0基础的同学们 下面的资料可以按顺序开始看

1. http://www.hiredintech.com/app#system-design

这是一个专门准备面试的网站 你只用关心system design部分 有很多的link后面会重

复提到 建议看完至少一遍

2. https://www.youtube.com/watch?v=-W9F\_\_D3oY4

非常非常好的入门资料 建议看3遍以上！

这是1里面提到的资料 是Harvard web app课的最后一节 讲scalability 里面会讲到很

多基础概念比如Vertical scaling, Horizontal scaling, Caching, Load balancing,

Database replication, Database partitioning 还会提到很多基本思想比如avoid

single point of failure

再强调一遍 非常好的资料！

3. http://www.lecloud.net/post/7295452622/scalability-for-dummies-part-1-clones

1里面提到的 Scalability for Dummies 还算不错 可以看一遍 知道基本思想

结束语：当你结束这一部分的学习的时候 你已经比50%的candidate知道的多了(因为很

多人都不准备 或者不知道怎么准备system design) 恭喜:)

== 进阶 ==

这一部分的资料更加零散 每个看的可能不一样 但是你每多看一篇文章或者一个视频

你就比别人强一点

这部分你会遇到很多新名词 我的建议是每当你遇到一个不懂的概念时 多google一下

看看这个概念或者技术是什么意思 优点和缺点各是什么 什么时候用 这些你都知道以

后 你就可以把他运用到面试中 让面试官刮目相看了

4. http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html

Database Sharding是一个很重要的概念 建议看一看

5. http://highscalability.com/all-time-favorites/

这个里面会讲到很多非常流行的网站架构是如何实现的 比如Twitter, Youtube,

Pinterest, Google等等 我的建议是看5-6个 然后你应该已经建立起了一些基本的意识

还有知道了某些技术和产品的作用和mapping 比如说到cache你会想到memcached和

Redis 说到

load balancer你会想到 Amazon ELB, F5一类的

6. http://www.infoq.com/

5里面很多的文章都会有链接 其中有很多会指向这个网站 这里面有很多的tech talk

很不错 可以看看

7. https://www.facebook.com/Engineering/notes

Facebook非常好的技术日志 会讲很多facebook的feature怎么实现的 比如facebook

message:https://www.facebook.com/notes/facebook-engineering/the-underlying-

technology-of-messages/454991608919 建议看看 尤其是准备面facebook的同学

这有一个facebook talk讲storage的<https://www.youtube.com/watch?v=5RfFhMwRAic>

8. 一些国内网站上的资料

http://blog.csdn.net/sigh1988/article/details/9790337

http://blog.csdn.net/v\_july\_v/article/details/6279498

9. 最后一些概念很有用 都是我再看这些资料的时候发现的 如果你没有遇到或者查过

建议查查

Distributed Hash Table

Eventual Consistency vs Strong Consistency

Read Heavy vs Write Heavy

Consistent Hashing

Sticky Sessions

Structured Data (uses DynamoDB) vs Unstructured Data(uses S3)http://smartdatacollective.com/michelenemschoff/206391/quick-guide-structured-and-unstructured-data http://stackoverflow.com/questions/18678315/amazon-s3-or-dynamodb

10 给有兴趣深入研究的人看的

Mining Massive Datasets --讲很多big data和data mining的东西

Big Data: Principles and best practices of scalable realtime data systems(http://www.amazon.com/gp/product/1617290343) --

twitter的前员工讲述如何处理实时数据 目前市面上讲解big data最好的一本书

10 凌乱的资料 随便看看吧

http://highscalability.com/blog/2013/10/28/design-decisions-for-scaling-your-high-traffic-feeds.html

== 小结＝＝

看多了以后 你的最终目标应该是心里有了一个大框架 一个基本的distributed system

是怎么搭起来的 然后心里有很多if condition 如果要是满足这个条件 我应该用什么

技术 比如如果read heavy那么用cache会提升performance之类的 同时知道应该避免什

么东西 比如避免single point of failure 再比如时间和空间的tradeoff在read

heavy的时候应该倾向于时间 Write heavy的时候倾向于空间等等

你总结出来的和我总结出来的大框架和if conditions肯定不完全一样 但因为system

design本来就是一个open ended question 所以不用害怕 能够自圆其说 就不会有问题

最后 本文纯属抛砖引玉 如果有大牛发现有错误或者有补充 欢迎留言 大家一起讨论

== FAQ ==

1. New Grad需要看System Design么?

答案是it depends. 有的公司会考system design 有的公司只考到OO design 有的公司

压根不考 当然 考到的公司对new grad的期望值会稍微低一点 但是 你有这么一个机会

能让你gain leverage over other candidates why not? 为什么要让自己在面试前害怕

面试官出system design的题目呢?

# Appendix comment

## 1. 审题

### 1.1 理解数据特点，构造测试用例，理解题意 【沟通】

* 观察已知测试用例；
* 观察数据特点（数据的个数，取值范围，是否重复，是否有序）并构造测试用例；
* 观察返回结果类型以及条件（不同的范围条件和结果意味着不同的逻辑）；

**需要沟通对题目的理解！！！**

### 1.2 数据得到的思考

数据（正负0）之间的算数运算是否会导致溢出（正正，正负）；同时正负0的出现可能也意味着edge-case

是否可能通过枚举所有可能出现的值来解决问题，例如:

* 根据取值大小来开辟缓存空间；
* 根据取值大小，使用Segment Tree就是枚举所有可能出现的值；
* 动态规划中数据取值过大， 则该值就不适合作为中间状态的纬度；

是否重复，可能涉及到边界情况或者可能需要去重；是否有序可能涉及到排序后能简化求解；

**需要沟通对数据的疑惑！！！**

## 2. 思考

解题思路的思考：

* + 正向思维：暴力/正常的解决方案思考：
  + 逆向思维：正向思维的解决过于复杂或者不易解决问题，可以考虑逆向思维下，逆向推到是否更容易解，例如值域二分，按值DP；

**需要沟通解题思路：不同解题思路中使用的数据结构和算法，需要沟通比较不同解法之间，适应场景（读、写）不同，时间复杂度和空间复杂度不同，trade off不同的优劣；**

## 3. 代码

写代码的时候注意随时写下注意点，以及每一步的操作步骤；

注意包装函数，甚至可以忽略简写包装函数；

切忌：当发现一个重要的bug，例如某种情况没有考虑到，千万不要快速的动手改代码，因为在一个明显的错误情况下，快速改代码也许解决这个问题，但是更有可能带来更多的问题： 1. Break以前的逻辑。 2. 该情况可能解决了，但是还存在类似变种。

## 4. 复查

1. 检查代码语法和算法实现

输入完整性检测

函数确保声明和使用是一致，包括返回值，形参和实参是否一致；

变量定义是否类型准确；

返回值是否和要求一致；

2. 抱着**理解代码运行**的思想快速阅读代码（以之前的数据特点作为注意事项）；

3. 使用之前的测试用例进行测试，用数据严格走代码流程 [必做]

**代码结束后，还需要沟通一些对于当前代码的看法，是否有冗余，是否可以更进一步优化等等；**