算法题准备材料

Tyrael.wang

# JAVA常用API

整数

public static int bitCount(int i)

https://blog.csdn.net/zhouzipeng000/article/details/56676885

1. 栈Stack类继承Vector类。主要方法：pushpoppeekempty（Java编程思想：Java本身的栈设计欠佳。暴露了很多Vector的方法。）

2. 队列Java 有Queue接口。LinkedList实现了Queue接口。offer插入队尾remove、pollelement、peek3. 字符串相关StringBuffer 线程安全StringBuilder 非线程安全字符串逆转stringBuilder.reverse4. 集合Arrays.binSearch的返回值5. Math算法相关：PriorityQueue 基于堆实现的无界队列，非线程安全的An unbounded priority {@linkplain Queue queue} based on a priority heap.

Treemap

Floor

ceiling

# 经典数据结构/算法

## 排序

## 二叉树

### 性质

### 构建

106. Construct Binary Tree from Inorder and Postorder Traversal

### 遍历

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

### 二叉搜索树Binary Search Tree

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

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

<https://en.wikipedia.org/wiki/Catalan_number#Applications_in_combinatorics>

1. 完全二叉樹、二叉搜索树

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

unique-binary-search-trees-ii：如何生成具体的树：递归。左子树从0到n。（注意类似的情况都可以用catalan数计数。）

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

450. Delete Node in a BST

1. 叶子节点直接删除

2. 单孩节点：直接替代

3. 双孩节点：

方案一：前驱值替代，再删除前驱

方案二：左树替代，右树下降低到最低

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

### 红黑树

理解要点

## 字典树

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

**public class** Trie {  
 Node **root**;  
  
 **public** Node create(){  
 **root** = **new** Node(26);  
 **return root**;  
 }  
  
 **public void** insert(String s){  
 *//逐个查找，找不到则建点* Node node = **root**;  
 **for** (**int** i = 0; i < s.length(); i++) {  
 **int** index = order(s.charAt(0));  
 Node child = node.**children**[index];  
 **if**(child == **null**){  
 child = **new** Node(26);  
 node.**children**[index] = child;  
 node = child;  
 }**else**{  
 node = child;  
 }  
 }  
 node.**leaf** = s;  
 }  
  
 **public boolean** search(String s){  
 Node node = **root**;  
 **for** (**int** i = 0; i < s.length(); i++) {  
 **int** index = order(s.charAt(0));  
 Node child = node.**children**[index];  
 **if**(child == **null**){  
 **return false**;  
 }**else**{  
 node = child;  
 }  
 }  
 *//如果该词是其他词的前缀，也不存在。* **return** node.**leaf** != **null**;  
 }  
  
 **public int** order(**char** a){  
 **return** a - **'a'**;  
 }  
}

211. Add and Search Word - Data structure design

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

648. Replace Words

676. Implement Magic Dictionary

677. Map Sum Pairs

红黑树

前缀和

线段树

图

拓扑排序

并查集扩展元素数量集合数量

排序

计算几何

算法导论33章

字符串匹配

kmp算法ac自动机

## 素数计算

https://leetcode.com/problems/2-keys-keyboard/solution/

## 二分查找

含重复元素、等号、上下界

34. Search for a Range

## 流算法

### 摩尔多数投票算法

查找1/2水王

The algorithm maintains in its [local variables](https://en.wikipedia.org/wiki/Local_variable) a sequence element and a counter, with the counter initially zero. It then processes the elements of the sequence, one at a time. When processing an element *x*, if the counter is zero, the algorithm stores *x* as its remembered sequence element and sets the counter to one. Otherwise, it compares *x* to the stored element and either increments the counter (if they are equal) or decrements the counter (otherwise). At the end of this process, if the sequence has a majority, it will be the element stored by the algorithm. This can be expressed in [pseudocode](https://en.wikipedia.org/wiki/Pseudocode) as the following steps:

* Initialize an element *m* and a counter *i* with *i* = 0
* For each element *x* of the input sequence:
  + If *i* = 0, then assign *m* = *x* and *i* = 1
  + else if *m* = *x*, then assign *i* = *i* + 1
  + else assign *i* = *i* − 1
* Return *m*

Even when the input sequence has no majority, the algorithm will report one of the sequence elements as its result. However, it is possible to perform a second pass over the same input sequence in order to count the number of times the reported element occurs and determine whether it is actually a majority. This second pass is needed, as it is not possible for a sublinear-space algorithm to determine whether there exists a majority element in a single pass through the input

另一种思路：配对删除。最终状态：留下一个（就是水王）；两个（必然两个水王）

1/3水王

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

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

### 水库采样

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

在高德纳的计算机程序设计艺术中，有如下问题：**可否在一未知大小的集合中，随机取出一元素？**。或者是Google面试题： I have a linked list of numbers of length N. N is very large and I don’t know in advance the exact value of N. How can I most efficiently write a function that will return k completely random numbers from the list（中文简化的意思就是：**在不知道文件总行数的情况下，如何从文件中随机的抽取一行？**）。两题的核心意思都是在总数不知道的情况下如何等概率地从中抽取一行？即是说如果最后发现文字档共有N行，则每一行被抽取的概率均为1/N？

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

**问题一**

首先考虑k为1的情况，即：给定一个长度很大或者长度未知数据流，限定对每个元素只能访问一次，写出一个随机选择算法，使得所有元素被选中的概率相等。

设当前读取的是第n个元素，采用归纳法分析如下：

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

**问题二**

将问题一中的条件变为，k为任意整数的情况，即要求最终返回的元素有k个，这就是水塘抽样（Reservoir Sampling）问题。要求是：取到第n个元素时，前n个元素被留下的几率相等，即k/n。

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

同样采用归纳法来分析：

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

伪代码如下：

//stream代表数据流

//reservoir代表返回长度为k的池塘

//从stream中取前k个放入reservoir；

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

reservoir[i] = stream[i];

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

p = random(0, i);

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

return reservoir;

## 下一个排列

O(n);O(1)

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

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

**while** **(**i **>=** 0 **&&** nums**[**i **+** 1**]** **<=** nums**[**i**])** **{ //找到第一个不满足逆序的数**

i**--;**

**}**

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

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

**while** **(**j **>=** 0 **&&** nums**[**j**]** **<=** nums**[**i**])** **{ //找到比i位置大一点的数**

j**--;**

**}**

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

**}**

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

**}**

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

https://blog.csdn.net/l664675249/article/details/50157669

**问题：求一个数组中连续n项的和。**

首先想到的肯定是做一个循环，把这个连续的n项加起来，时间复杂度为O（n）。

**会不会有O（logn）的解法？**

如果只操作一次还是可以接受的，但是如果需要大量的求和操作，比如第一次求下标（1，1234）的和第二次求下标（2，1024）的和，很容易发现在第一次计算的过程中（2，1024）的和是计算过的，只是没有保存下来，导致第二次求和的时候还要再算一遍。如果事先把一部分的和先计算并保存起来，这样会不会更快一些呢？

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

**核心思想:**

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

下面是一个示意图



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

　　　　如：4=100(2)　　e[4]=a[1]+a[2]+a[3]+a[4];   
　　　　　　6=110(2)　　e[6]=a[5]+a[6]   
　　　　　　7=111(2)　　e[7]=a[7]   
　　　　　　   
计算2^k的两个方法

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

**父节点**

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

**计算方法**

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

**子节点**

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

**计算方法**

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

**实现代码**

public class NumArray {

private int[] tree; //Binary Indexed Tree

private int[] nums; //原始数组

public NumArray(int[] nums) {

this.nums = nums;

int sum = 0;

int lowbit;

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

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

sum = 0;

lowbit = i & ((i - 1) ^ i);

for (int j = i; j > i - lowbit; j--) {

sum = sum + nums[j - 1];

}

tree[i] = sum;

}

}

//更新

void update(int i, int val) {

int tem = val - nums[i];

nums[i] = val;

i++;

for (; i < tree.length; i = i + (i & ((i - 1) ^ i))) {

tree[i] += tem;

}

}

public int sumRange(int i, int j) {

return getSum(j) - getSum(i - 1);

}

//求和

public int getSum(int i) {

int sum = 0;

i++;

while (i > 0) {

sum = sum + tree[i];

i = i - (i & ((i - 1) ^ i));

}

return sum;

}

}

## 树

310. Minimum Height Trees

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

//确定树根

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

## 线段树

307. Range Sum Query – Mutable

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

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

Segment tree is a very flexible data structure, because it is used to solve numerous range query problems like finding minimum, maximum, sum, greatest common divisor, least common denominator in array in logarithmic time.

这图是不是有问题？

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

Segment tree could be implemented using either an array or a tree. For an array implementation, if the element at index iis not a leaf, its left and right child are stored at index 2iand 2i + 1respectively.

In the example above (Figure 2), every leaf node contains the initial array elements {2,4,5,7,8,9}. The internal nodes contain the sum of the corresponding elements in range - (11) for the elements from index 0 to index 2. The root (35) being the sum of its children (6);(29), holds the total sum of the entire array.

Segment Tree can be broken down to the three following steps:

1. Pre-processing step which builds the segment tree from a given array.
2. Update the segment tree when an element is modified.
3. Calculate the Range Sum Query using the segment tree.

1. Build segment tree：bottom-up approach

We already know from the above that if some node p*p* holds the sum of [*i*…*j*] range, its left and right children hold the sum for range [i \ldots \frac{i + j}{2}][*i*…​2​​*i*+*j*​​] and [\frac{i + j}{2} + 1, j][​2​​*i*+*j*​​+1,*j*] respectively.

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

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

//注意从1开始

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

**int** n**;**

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

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

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

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

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

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

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

**}**

**Complexity Analysis**

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

because we calculate the sum of one node during each iteration of the for loop. There are approximately2*n* nodes.

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

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

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

* Space complexity : O(n).

2. Update segment tree

When we update the array at some index i*i* we need to rebuild the segment tree, because there are tree nodes which contain the sum of the modified element. Again we will use a bottom-up approach. We update the leaf node that stores a[i]. From there we will follow the path up to the root updating the value of each parent as a sum of its children values.

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

pos **+=** n**;**

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

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

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

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

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

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

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

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

**}**

*// parent is updated after child is updated*

tree**[**pos **/** 2**]** **=** tree**[**left**]** **+** tree**[**right**];**

pos **/=** 2**;**

**}**

**}**

**Complexity Analysis**

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

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

* Space complexity : O(1)*O*(1).

3. Range Sum Query

We can find range sum query [L, R]:

Algorithm hold loop invariant:

l \le r*l*≤*r* and sum of [*L*…*l*] and [*r*…*R*] has been calculated, where land rare the left and right boundary of calculated sum. Initially we set lwith left leaf L*L* and r*r* with right leaf R*R*. Range [l, r] hrinks on each iteration till range borders meets after approximately \log nlog*n* iterations of the algorithm

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

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

*// get leaf with value 'l'*

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

*// get leaf with value 'r'*

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

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

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

**if** **((**l **%** 2**)** **==** 1**)** **{**

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

l**++;**

**}**

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

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

r**--;**

**}**

**l /= 2;**

**r /= 2;**

**}**

**return** sum**;**

**}**

**Complexity Analysis**

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

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

* Space complexity : O(1)*O*(1).

## 分桶法和平方分割

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

思路就是避免重复计算

分割成sqrt（n）个桶

## 图

### 图的表示

133. Clone Graph

### 二分图

Bipartite graph

From Wikipedia, the free encyclopedia

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

Example of a bipartite graph without cycles

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

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

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

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

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

**Contents**

  [hide]

* [1Examples](https://en.wikipedia.org/wiki/Bipartite_graph#Examples)
* [2Properties](https://en.wikipedia.org/wiki/Bipartite_graph#Properties)
  + [2.1Characterization](https://en.wikipedia.org/wiki/Bipartite_graph#Characterization)
  + [2.2König's theorem and perfect graphs](https://en.wikipedia.org/wiki/Bipartite_graph#K%C3%B6nig's_theorem_and_perfect_graphs)
  + [2.3Degree](https://en.wikipedia.org/wiki/Bipartite_graph#Degree)
  + [2.4Relation to hypergraphs and directed graphs](https://en.wikipedia.org/wiki/Bipartite_graph#Relation_to_hypergraphs_and_directed_graphs)
* [3Algorithms](https://en.wikipedia.org/wiki/Bipartite_graph#Algorithms)
  + [3.1Testing bipartiteness](https://en.wikipedia.org/wiki/Bipartite_graph#Testing_bipartiteness)
  + [3.2Odd cycle transversal](https://en.wikipedia.org/wiki/Bipartite_graph#Odd_cycle_transversal)
  + [3.3Matching](https://en.wikipedia.org/wiki/Bipartite_graph#Matching)
* [4Additional applications](https://en.wikipedia.org/wiki/Bipartite_graph#Additional_applications)
* [5See also](https://en.wikipedia.org/wiki/Bipartite_graph#See_also)
* [6References](https://en.wikipedia.org/wiki/Bipartite_graph#References)
* [7External links](https://en.wikipedia.org/wiki/Bipartite_graph#External_links)

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

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

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

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

More abstract examples include the following:

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

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

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

Bipartite graphs may be characterized in several different ways:

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

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

In bipartite graphs, the size of [minimum vertex cover](https://en.wikipedia.org/wiki/Minimum_vertex_cover) is equal to the size of the [maximum matching](https://en.wikipedia.org/wiki/Maximum_matching); this is [König's theorem](https://en.wikipedia.org/wiki/K%C3%B6nig%27s_theorem_(graph_theory)).[[16]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-16)[[17]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-17) An alternative and equivalent form of this theorem is that the size of the [maximum independent set](https://en.wikipedia.org/wiki/Maximum_independent_set) plus the size of the maximum matching is equal to the number of vertices. In any graph without [isolated vertices](https://en.wikipedia.org/wiki/Isolated_vertex) the size of the [minimum edge cover](https://en.wikipedia.org/wiki/Minimum_edge_cover) plus the size of a maximum matching equals the number of vertices.[[18]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-18) Combining this equality with König's theorem leads to the facts that, in bipartite graphs, the size of the minimum edge cover is equal to the size of the maximum independent set, and the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.

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

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

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

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

{\displaystyle \sum \_{v\in V}\deg(v)=\sum \_{u\in U}\deg(u)=|E|\,.}

The degree sequence of a bipartite graph is the pair of lists each containing the degrees of the two parts {\displaystyle U} and {\displaystyle V}. For example, the complete bipartite graph *K*3,5 has degree sequence {\displaystyle (5,5,5),(3,3,3,3,3)}. Isomorphic bipartite graphs have the same degree sequence. However, the degree sequence does not, in general, uniquely identify a bipartite graph; in some cases, non-isomorphic bipartite graphs may have the same degree sequence.

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

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

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

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

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

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

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

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

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

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

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

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

*Main article:*[*Odd cycle transversal*](https://en.wikipedia.org/wiki/Odd_cycle_transversal)

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

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

[Odd cycle transversal](https://en.wikipedia.org/wiki/Odd_cycle_transversal) is an [NP-complete](https://en.wikipedia.org/wiki/NP-complete) [algorithmic](https://en.wikipedia.org/wiki/Algorithm) problem that asks, given a graph *G* = (*V*,*E*) and a number *k*, whether there exists a set of *k* vertices whose removal from *G* would cause the resulting graph to be bipartite.[[27]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-yannakakis1978node-27) The problem is [fixed-parameter tractable](https://en.wikipedia.org/wiki/Parameterized_complexity), meaning that there is an algorithm whose running time can be bounded by a polynomial function of the size of the graph multiplied by a larger function of *k*.[[28]](https://en.wikipedia.org/wiki/Bipartite_graph#cite_note-reed2004finding-28) The name *odd cycle transversal* comes from the fact that a graph is bipartite if and only if it has no odd [cycles](https://en.wikipedia.org/wiki/Cycle_(graph_theory)). Hence, to delete vertices from a graph in order to obtain a bipartite graph, one needs to "hit all odd cycle", or find a so-called odd cycle [transversal](https://en.wikipedia.org/wiki/Transversal_(combinatorics)) set. In the illustration, every odd cycle in the graph contains the blue (the bottommost) vertices, so removing those vertices kills all odd cycles and leaves a bipartite graph.

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

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

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

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

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

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

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

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

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

### 拓扑排序

### 单源路径

**final int**[][] **times**;  
 **final int K**;  
 **int N**;  
  
 HashSet<Integer> **arrived** = **new** HashSet<>();  
 **public int**[] **distance**;  
 *//经过的索引路径* **public** List<List<Integer>> **path** = **new** ArrayList<>();  
  
 **public** Dijkstra(**int**[][] times, **int** k) {  
 **this**.**times** = times;  
 **K** = k;  
 **N** = times[0].**length**;  
 **distance** = **new int**[**N**];  
 **for** (**int** i = 0; i < **N**; i++) {  
 **distance**[i] = times[**K**][i];  
 **path**.add(**new** ArrayList<>());  
 }  
 **arrived**.add(**K**);  
 }  
  
 **public void** run() {  
 *//每次加入一个点* **for** (**int** i = 0; i < **N**-1; i++) {  
 **int** nearIndex = getNear();  
 **if** (nearIndex == -1){  
 **return**;  
 }  
 **arrived**.add(nearIndex);  
 updateDistance(nearIndex);  
 }  
 }  
  
 **private void** updateDistance(**int** bridgeIndex){  
 **for** (**int** i = 0; i < **N**; i++) {  
 **if**(**arrived**.contains(i)){  
 **continue**;  
 }  
 **if** (**distance**[i] - **distance**[bridgeIndex]> **times**[bridgeIndex][i]){  
 *//加号越界  
// if (distance[i] > distance[bridgeIndex] + times[bridgeIndex][i]){* **distance**[i] = **distance**[bridgeIndex] + **times**[bridgeIndex][i];  
 List<Integer> thisPath = **path**.get(i);  
 thisPath.add(i);  
 }  
 }  
 }  
  
 **private int** getNear(){  
 **int** minDistance = Integer.***MAX\_VALUE***;  
 **int** minIndex = -1;  
 **for** (**int** j = 0; j < **N**; j++) {  
 **if** (**arrived**.contains(j)){  
 **continue**;  
 }  
 **if** (minDistance > **distance**[j]){  
 minDistance = **distance**[j];  
 minIndex = j;  
 }  
 }  
 **return** minIndex;  
 }

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

单源路径

743. Network Delay Time

带权单源路径

### 多源路径

带权多源路径

399. Evaluate Division

## 01背包问题

只能穷举

NP问题

字符串匹配

318. Maximum Product of Word Lengths

## 并查集

**int**[] **parent**;  
**int**[] **rank**;  
*//每个集合元素个数***int**[] **count**;  
*//集合总数***int countSet**;  
  
**public** DisjointCountSet(**int** n) {  
 **parent** = **new int**[n];  
 **rank** = **new int**[n];  
 **count** = **new int**[n];  
}  
  
*/\*\*  
 \* 要求本来没有。  
 \** ***@param x*** *\*/***public void** makeSet(**int** x) {  
 **parent**[x] = x;  
 **rank**[x] = 0;  
 **count**[x] = 1;  
 **countSet**++;  
}  
  
**public void** union(**int** x, **int** y) {  
 **if** (findSet(x) == findSet(y)){  
 **return**;  
 }  
 **countSet**--;  
 **int** cx = getCount(x),  
 cy = getCount(y);  
 setCount(x, cx+cy);  
 setCount(y, cx +cy);  
 link(findSet(x), findSet(y));  
}  
  
**private void** link(**int** set, **int** set1) {  
 **if** (**rank**[set] > **rank**[set1]) {  
 **parent**[set1] = set;  
  
 } **else if** (**rank**[set] < **rank**[set1]) {  
 **parent**[set] = set1;  
 } **else** {  
 **rank**[set1] += 1;  
 **parent**[set] = set1;  
 }  
  
}  
  
**public int** findSet(**int** x) {  
 **if** (**parent**[x] != x) {  
 **int** parentIndex = findSet(**parent**[x]);  
 **parent**[x] = parentIndex;  
 }  
 **return parent**[x];  
}  
  
**public int** getCountSet(){  
 **return countSet**;  
}  
  
**public int** getCount(**int** i){  
 **int** parent = findSet(i);  
 **return count**[parent];  
}  
  
**public void** setCount(**int** i, **int** c){  
 **int** parent = findSet(i);  
 **count**[parent] = c;  
}  
  
**public int** getMaxCount(){  
 **int** max = 0;  
 **for** (**int** i = 0; i < **count**.**length**; i++) {  
 **if** (max < **count**[i]){  
 max = **count**[i];  
 }  
 }  
 **return** max;  
}

200. Number of Islands

## 字符串匹配

朴素、Rabin-Karp

几个定理

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



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

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

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

## 计算几何

### 寻找凸包

1. Graham扫描法
2. Jarvis步进法

### 寻找最近点对

## 扩展

### 马拉车算法

**1.Manacher算法原理与实现**

下面介绍Manacher算法的原理与步骤。

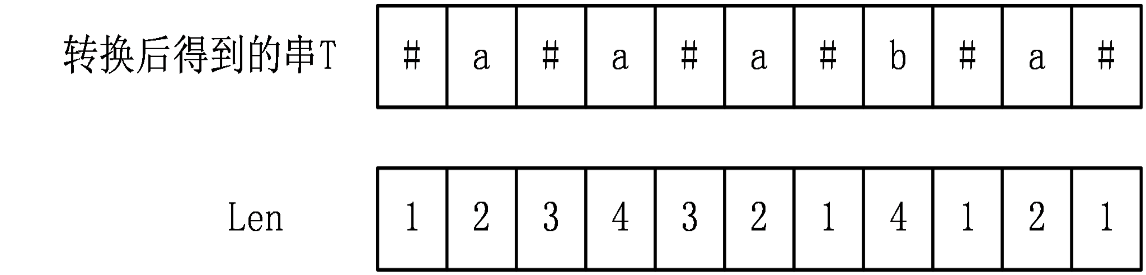
首先，Manacher算法提供了一种巧妙地办法，将长度为奇数的回文串和长度为偶数的回文串一起考虑，具体做法是，在原字符串的每个相邻两个字符中间插入一个分隔符，同时在首尾也要添加一个分隔符，分隔符的要求是不在原串中出现，一般情况下可以用#号。下面举一个例子：



**（1）Len数组简介与性质**

Manacher算法用一个辅助数组Len[i]表示以字符T[i]为中心的最长回文字串的最右字符到T[i]的长度，比如以T[i]为中心的最长回文字串是T[l,r],那么Len[i]=r-i+1。

对于上面的例子，可以得出Len[i]数组为:



Len数组有一个性质，那就是Len[i]-1就是该回文子串在原字符串S中的长度，至于证明，首先在转换得到的字符串T中，所有的回文字串的长度都为奇数，那么对于以T[i]为中心的最长回文字串，其长度就为2\*Len[i]-1,经过观察可知，T中所有的回文子串，其中分隔符的数量一定比其他字符的数量多1，也就是有Len[i]个分隔符，剩下Len[i]-1个字符来自原字符串，所以该回文串在原字符串中的长度就为Len[i]-1。

有了这个性质，那么原问题就转化为求所有的Len[i]。下面介绍如何在线性时间复杂度内求出所有的Len。

**（2）Len数组的计算**

首先从左往右依次计算Len[i]，当计算Len[i]时，Len[j](0<=j<i)已经计算完毕。设P为之前计算中最长回文子串的右端点的最大值，并且设取得这个最大值的位置为po，分两种情况：

第一种情况：i<=P

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



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

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



第二种情况: i>P

如果i比P还要大，说明对于中点为i的回文串还一点都没有匹配，这个时候，就只能老老实实地一个一个匹配了，匹配完成后要更新P的位置和对应的po以及Len[i]。



**2.时间复杂度分析**

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

5. Longest Palindromic Substring

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

### 格雷码

N位格雷码

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

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

### 四平方定理/三平方定理

a natural number can be represented as the sum of three squares of integers

{\displaystyle n=x^{2}+y^{2}+z^{2}} n=x^{2}+y^{2}+z^{2}

if and only if n is not of the form {\displaystyle n=4^{a}(8b+7)} n = 4^a(8b + 7) for integers a and b.

注意x、y、z可以是0.

int numSquares(int n) {

//去除公式中的4

while ((n & 3) == 0) //n%4 == 0

n >>= 2;

if ((n & 7) == 7) return 4; //n % 8 == 7

if(is\_square(n)) return 1;

int sqrt\_n = (int) sqrt(n);

for(int i = 1; i<= sqrt\_n; i++){

if (is\_square(n-i\*i)) return 2;

}

return 3;

}

**int** **is\_square**(**int** n){

**int** temp = (**int**) sqrt(n);

**return** temp \* temp == n;

}

## 简单

### 逆波兰表达式

150. Evaluate Reverse Polish Notation

# 从形式到思路

## 广义表

341. Flatten Nested List Iterator

385. Mini Parser

394. Decode String

借助栈实现

## 数字

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

解法一：

删除数字-》保留数字-》高位重要

解法二：

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

使用栈

738. Monotone Increasing Digits

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

例外123444321

### 位操作

397. Integer Replacement

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

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

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

S:

If n is even, halve it.

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

Otherwise, increment n.

doing bitCount on every iteration is not the best way. It is enough to examine the last two digits to figure out whether incrementing or decrementing will give more 1's. Indeed, if a number ends with 01, then certainly decrementing is the way to go. Otherwise, if it ends with 11, then certainly incrementing is at least as good as decrementing (\*011 -> \*010 / \*100) or even better (if there are three or more 1's).

海明距离

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

按未划分成两组，当前位的贡献k\*(n-k).

421. Maximum XOR of Two Numbers in an Array

338. Counting Bits

779. K-th Symbol in Grammar

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

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

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

***Obervation 3***: if binary string of **k** is used, let **k = 1001010**, then we have:  
f(1001010) = f(100101) ^ 0 = f(10010) ^ 1 ^ 0 = f(1001) ^ 0 ^ 1 ^ 0 = ... = f(0) ^ 1 ^ 0 ^ 0 ^1 ^ 0 ^ 1 ^ 0 = 1 ^ 0 ^ 0 ^1 ^ 0 ^ 1 ^ 0  
So, the result is the **xor** operation on all bits of **k**. Since 0 does not change **xor** result, we can ignore all 0s.  
f(1001010) = 1 ^ 1 ^ 1 = (1^1) ^ 1 = 0 ^ 1 = 1  
f(11110011) = 1 ^ 1^ 1 ^ 1 ^ 1 ^1 = (1 ^ 1) ^ (1 ^ 1) ^ (1 ^1) = 0  
Now, it's easy to tell f(k) = 0 if **k** has **even** number of 1s in binary representation, and f(k) = 1 when **k** has **odd** number of 1s

## 从回溯/递归/推进到dp

494. Target Sum

## 数组/矩阵区间和/积

1. 303. Range Sum Query - Immutable

累积和

304. Range Sum Query 2D – Immutable

dp/累积和



307. Range Sum Query – Mutable

308．Range Sum Query 2D – Mutable

2维树状数组/线段树

## 四则运算

241. Different Ways to Add Parentheses

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

((2-1)-1) = 0

(2-(1-1)) = 2

穷举，分治，备忘录

399. Evaluate Division：连环计算

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

构造图搜索

## 字符/数字集合

通常用数组来存储，但是没有前后关系，跟索引也没有关系。

### 排列/组合

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

按规律推导

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

47. Permutations II

重复的下一个排列

77. Combinations

递归构造。

752. Open the Lock

两端bfs

### 元素和

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

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

***target*** = 4

The possible combination ways are:

(1, 1, 1, 1)、(1, 1, 2)。。。

回溯-》dp

### 背包问题/选或不选

集合划分问题

https://en.wikipedia.org/wiki/Partition\_problem

416. Partition Equal Subset Sum

698. Partition to K Equal Sum Subsets

不同角度，穷尽搜索

<https://leetcode.com/problems/partition-to-k-equal-sum-subsets/solution/>

Approach #1: Search by Constructing Subset Sums

Approach #2: Dynamic Programming on Subsets of Input

473. Matchsticks to Square

Dfs 大值在前

背包问题

https://leetcode.com/problems/ones-and-zeroes/description/

39. Combination Sum

选用多次

40. Combination Sum II

集合中有重复元素，选用一次

216. Combination Sum III

元素不可重复

377. Combination Sum IV

### 推进

<https://leetcode.com/problems/house-robber-ii/description/>

DP

### 两个字符串的关系

缩小规模-》DP

https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/description/

722. Remove Comments

line comments, and block comments.

注意嵌套情况

“”字符串嵌套，转义字符

//可以在一行的任意位置

/\*/不完整

https://leetcode.com/problems/unique-substrings-in-wraparound-string/description/

https://leetcode.com/problems/unique-substrings-in-wraparound-string/discuss/95439/Concise-Java-solution-using-DP

712. Minimum ASCII Delete Sum for Two Strings

## 字符串

字符间有位置关系

### 回文

回文利用位置对称关系

1. 中心拓展
2. 马拉车

子串

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

如上所述

131. Palindrome Partitioning

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

Return all possible palindrome partitioning of *s*.

找到第一个回文，然后递归余下的

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

递归+备忘=dp

子序列（中漏）

https://leetcode.com/problems/longest-palindromic-subsequence/description/

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

State transition:

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

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

Initialization: dp[i][i] = 1

## 单词列表

查找：字典树

212. Word Search II

318. Maximum Product of Word Lengths

126. Word Ladder II

单词变种最短路径

双向bfs

注意，没有答案的，树的高度只要一半就可以了。

新生的节点，不要重复已有的节点。但是同一层的可以重复。

变种后集合的hash查找，比逐个单词比较是否变种，要快很多。

676. Implement Magic Dictionary

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

## 数组

220. Contains Duplicate III

380. Insert Delete GetRandom O(1)

777. Swap Adjacent in LR String

### 排序

324. Wiggle Sort II

快排的应用。

大小元素的摆放

### 前K

373. Find K Pairs with Smallest Sums

215. Kth Largest Element in an Array、

快排

692. Top K Frequent Words

### 波形

股票

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/description/

状态思想

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/discuss/75928/Share-my-DP-solution-(By-State-Machine-Thinking)

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/description/

456. 132 Pattern

区间中的值

//more

### 划分

813. Largest Sum of Averages

1. 回溯 + memo

2. 二维dp

### 子序列长度

n!个子序列

https://leetcode.com/problems/wiggle-subsequence/description/

https://leetcode.com/problems/is-subsequence/description/

public boolean isSubsequence(String s, String t) {

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

int indexS = 0, indexT = 0;

while (indexT < t.length()) {

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

indexS++;

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

}

indexT++;

}

return false;

}

491. Increasing Subsequences

dfs

### 丑数

质因子只含235的数

https://leetcode.com/problems/ugly-number-ii/description/

三个队列

313. Super Ugly Number

### 递增子串

https://leetcode.com/problems/number-of-longest-increasing-subsequence/description/

### 排序数组查找

二分查找变种

### 逆序

775. Global and Local Inversions

To check if #global inversion=#local inversion, we just need to ensure that there are no such inversion:i>j+1, a[i]<a[j]

### 两个数的关系

2/3/4 Sum （smaller、closet）

暂定一个值，排序查找另一个值

注意平均值关系，运用两个指针单向滑动

### 小区间

思路：计数排序、二分查找

例：

825. Friends Of Appropriate Ages

### 区间和/积

log(∏​*i*​​*x*​*i*​​)=∑​*i*​​log*x*​*i*​​

前缀和

累积和：累积和特点：对于单一元素，单调递增。

209. Minimum Size Subarray Sum

713. Subarray Product Less Than K

明显：列举所有：O(n3)

累积和：O(n2)

累积和：注意脑中有一个递增序列：二分查找(nlogn)

两个指针：某一个index，有条件提前结束查找：（n）

238. Product of Array Except Self

空间利用，

利用输出数组

Dp空间的优化

### 利用索引

769. Max Chunks To Make Sorted

768. Max Chunks To Make Sorted II

565. Array Nesting

442. Find All Duplicates in an Array

映射空间大的话，最终可以恢复数组

274. H-Index

275. H-Index II

利用索引+计数排序+DP

二分搜索

526. Beautiful Arrangement

### 环/并查集

并查集问题也可以用dfs/bfs来解决。

565. Array Nesting

547. Friend Circles

统计不同集个数

### 其他

数组实际上是一个map可以在原数组操作

https://leetcode.com/problems/delete-and-earn/description/

## 矩阵

解题思路：回溯、dfs、bfs、dp

路径搜索，一般只依赖邻居，通常可以用dp解决

最简单的空间占用是n2

有些可以优化到n，如2向路径。跟依赖多少方向有关系。

有些可以优化到1

417. Pacific Atlantic Water Flow

### 矩阵2向路径

62 Unique Paths排列组合问题哦：DRRRDRRR（m\*n）!/(m!\*n!)

63. Unique Paths II

DP

### 矩阵4向路径

矩阵邻居搜索

79. Word Search

212. Word Search II

出界

https://leetcode.com/problems/out-of-boundary-paths/description/

542. 01 Matrix

//more

529. Minesweeper

Search rules:

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

### 矩阵搜索

74. Search a 2D Matrix

二分搜索，mid的计算比较复杂

### 三角形

120. Triangle

坐标变换、dp

756. Pyramid Transition Matrix

一行一行处理，逐渐往上堆

字符可以看作数字

### 区域面积

拆分维度，分别dp，再综合计算

https://leetcode.com/problems/maximal-square/description/

764. Largest Plus Sign

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

## 数学

3.2.4. 数组积

29 Divides1：位操作，变大除数s2：长除法

326 PowerOfThree// 1162261467 is 3^19, 3^20 is bigger than int return ( n>0 && 1162261467%n==0);

概率

https://leetcode.com/problems/soup-servings/description/

丑数

https://leetcode.com/problems/integer-break/description/

all factors should be 2 or 3 (N > 4)

3 \* 3 > 2 \* 2 \* 2.

## 链表

19. Remove Nth Node From End两个指针O(n)

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

## 区间维护

56 Merge Intervals排序以后按顺序合并

228. Summary Ranges

646. Maximum Length of Pair Chain

452. Minimum Number of Arrows to Burst Balloons

435. Non-overlapping Intervals

课程时间安排

https://en.wikipedia.org/wiki/Interval\_scheduling#Interval\_Scheduling\_Maximization

## 二叉树

236. Lowest Common Ancestor of a Binary Tree

层次遍历

117. Populating Next Right Pointers in Each Node II

114. Flatten Binary Tree to Linked List

652. Find Duplicate Subtrees

标志id的思想。

623. Add One Row to Tree

### 路径

路径dfs

113. Path Sum II

根到叶

437. Path Sum III

任意点

优化算法：每个路径构建前缀和

124. Binary Tree Maximum Path Sum

129. Sum Root to Leaf Numbers

### Level

129. Sum Root to Leaf Numbers

## 树

<https://leetcode.com/problems/house-robber-iii/description/>

遍历

https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/description/

## 图

207 Course Schedule依赖问题：拓扑排序（需要手写）时间问题：贪心算法

缔结斯科拉

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

332. Reconstruct Itinerary

欧拉回路

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

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

环检测

802. Find Eventual Safe States

### 二分图

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

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

-1: Haven't been colored.

0: Blue.

1: Red.

For each node,

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

If it has been colored, check if the current color is the same as the color that is going to be used to color it. (Please forgive my english... Hope you can understand it.)

## 猜大小

https://leetcode.com/problems/guess-number-higher-or-lower-ii/description/

## 极大极小

486. Predict the Winner

https://leetcode.com/problems/predict-the-winner/solution/

649. Dota2 Senate

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

## 拾遗

284. Peeking Iterator

473. Matchsticks to Square

621. Task Scheduler

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

Output: 8

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

406. Queue Reconstruction by Height

一个人的位置，由比他高的人决定。

从高到低确定位置。

393. UTF-8 Validation

A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

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

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

This is how the UTF-8 encoding would work:

Char. number range | UTF-8 octet sequence

(hexadecimal) | (binary)

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

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

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

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

捞针

260. Single Number III

全部两次+两个一次

//more

# 常用解题思路

## 答案空间穷举回溯搜索

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

需要有能力看出哪些问题不能优化。

参考资料，算法导论NP问题。

细节优化：可以采用双端bfs、减支方法优化。虽然不能提高O，可以快一点。

## DP

能够应用DP的问题，通常有两种可能

1. 可以构建无后效性递归式
2. 可以应用回溯算法，并且有大量重复。

表面上是TopDown算法，一般可以转化为bottomUp算法。

DP的空间，

思考时，可以用较大的空间。

最终的优化，看递归式，到底依赖了多少上一步的结果。

https://leetcode.com/problems/2-keys-keyboard/discuss/105932/Java-solutions-from-naive-DP-to-optimized-DP-to-non-DP

常用解决方案：分治、二分、DP哈希、双指针、排序

双向bfs

贪心算法专题

关键是需要证明为什么贪心是对的。

狭义贪心算法是动态规划的特例。贪心算法代码看起来比动态规划简单，实际上思维上要复杂一点。算法导论有介绍321 Create Maximum Number广义贪心算法

回溯

心中一颗答案空间搜索树

两种搜索：dfs、bfs

Dfs用递归、栈实现

Bfs队列实现

# 常用正则表达式

# 测试细节

解题流程

确认算法、复杂度

确认输入边界，说明分支处理。

大于还是大于等于？

编写常规流程

代码走查

完善测试用例，确认测试通过。

Comparator注意越界

## 数字验证/解析

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

测试用例总结parsedouble的代码没有开源

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

数字解析

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

I think we only need to handle four cases:

discards all leading whitespaces

sign of the number

overflow

invalid input

过程中计算负数，因为负数范围大。

注意越界的处理

//容错比较低，需要提前验证

public static int parseInt(String s, int radix)

throws NumberFormatException

{

/\*

\* WARNING: This method may be invoked early during VM initialization

\* before IntegerCache is initialized. Care must be taken to not use

\* the valueOf method.

\*/

if (s == null) {

throw new NumberFormatException("null");

}

if (radix < Character.MIN\_RADIX) {

throw new NumberFormatException("radix " + radix +

" less than Character.MIN\_RADIX");

}

if (radix > Character.MAX\_RADIX) {

throw new NumberFormatException("radix " + radix +

" greater than Character.MAX\_RADIX");

}

int result = 0;

boolean negative = false;

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

int limit = -Integer.MAX\_VALUE;

int multmin;

int digit;

if (len > 0) {

char firstChar = s.charAt(0);

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

if (firstChar == '-') {

negative = true;

limit = Integer.MIN\_VALUE;

} else if (firstChar != '+')

throw NumberFormatException.forInputString(s);

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

throw NumberFormatException.forInputString(s);

i++;

}

multmin = limit / radix;

while (i < len) {

// Accumulating negatively avoids surprises near MAX\_VALUE

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

if (digit < 0) {

throw NumberFormatException.forInputString(s);

}

//是否可以乘，升位

if (result < multmin) {

throw NumberFormatException.forInputString(s);

}

result \*= radix;

//是否可以增加本位的值

if (result < limit + digit) {

throw NumberFormatException.forInputString(s);

}

result -= digit;

}

} else {

throw NumberFormatException.forInputString(s);

}

return negative ? result : -result;

}

3. Longest Substring Without Repeating Characters

滑动窗口

If a substring s\_{ij}s

​ij

​​ from index ii to j - 1j−1 is already checked to have no duplicate characters. We only need to check if s[j]s[j] is already in the substring s\_{ij}s

​ij

​​ .

355. Design Twitter

字符串匹配

Rolling hash

187. Repeated DNA Sequences

//TODO

凡是单调的列表，就要考虑下二分搜索。

或者存在有个有限空间。

Anagram

性质

字母数相等

与位置无关

注意结果集，或者任何集比较小的情况，

可以在此做文章。

二分、以结果为key作map

676. Implement Magic Dictionary

Trie

排序

快排

链表快排

插入法

两个指针法

链表归并排序

快慢指针划分

非递归归并排序

写循环注意参照for循环，不要忘了推进

链表环检测

环入口

142. Linked List Cycle II

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

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

s=nr-m=(n-1)r+(r-m),here we takes n = 1..so, using one pointer start from the start node of list, another pointer start from the first meeting node, all of them wake one step at a time, the first time they meeting each other is the start of the cycle.

高度平衡二叉树构建

Power（x，n）

数学

https://leetcode.com/problems/reach-a-number/solution/

公约数

裴蜀定理

https://leetcode.com/problems/water-and-jug-problem/discuss/83715/Math-solution-Java-solution

四则运算的原始算法

https://leetcode.com/problems/multiply-strings/description/

排列

60. Permutation Sequence

https://leetcode.com/problems/permutation-sequence/discuss/22507/%22Explain-like-I'm-five%22-Java-Solution-in-O(n)

https://leetcode.com/problems/next-greater-element-iii/description/

https://leetcode.com/problems/integer-replacement/description/

https://leetcode.com/problems/ugly-number/description/

<https://leetcode.com/problems/ugly-number-ii/description/>

<https://leetcode.com/problems/ugly-number-ii/discuss/69362/O(n)-Java-solution>

https://leetcode.com/problems/super-ugly-number/description/

https://leetcode.com/problems/rotate-function/discuss/87853/Java-O(n)-solution-with-explanation

F(k) = F(k-1) + sum - nBk[0]

旋转函数，注意前后的数之间的关系

368. Largest Divisible Subset

取模运算

https://baike.baidu.com/item/%E5%8F%96%E6%A8%A1%E8%BF%90%E7%AE%97/10739384?fr=aladdin

算法道路数论章节

<https://leetcode.com/problems/super-pow/description/>

四平方定理

<https://leetcode.com/problems/perfect-squares/description/>

平方数、奇数偶数

<https://leetcode.com/problems/bulb-switcher/description/>

<https://leetcode.com/problems/bulb-switcher-ii/solution/>

答案空间很小，无限循环

数字单词

The **even** digits all have a unique letter while the **odd** digits all don't:

zero: Only digit with z  
two: Only digit with w  
four: Only digit with u  
six: Only digit with x  
eight: Only digit with g

The odd ones for easy looking, each one's letters all also appear in other digit words:  
one, three, five, seven, nine

<https://leetcode.com/problems/count-numbers-with-unique-digits/discuss/>

<https://leetcode.com/problems/integer-break/description/>

https://leetcode.com/problems/escape-the-ghosts/discuss/116678/Why-interception-in-the-middle-is-not-a-good-idea-for-ghosts.

https://leetcode.com/problems/minimum-moves-to-equal-array-elements-ii/description/

子串

<https://leetcode.com/problems/longest-word-in-dictionary-through-deleting/description/>

回文

https://leetcode.com/problems/longest-palindromic-substring/description/

https://leetcode.com/problems/longest-palindromic-substring/solution/

https://leetcode.com/problems/shortest-palindrome/description/

https://leetcode.com/problems/palindrome-pairs/description/

<https://leetcode.com/problems/longest-palindromic-subsequence/description/>

<https://leetcode.com/problems/palindromic-substrings/description/>

测试

https://leetcode.com/problems/simplify-path/description/

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

子串

https://leetcode.com/problems/longest-uncommon-subsequence-ii/description/

https://leetcode.com/problems/delete-operation-for-two-strings/solution/

<https://leetcode.com/problems/expressive-words/solution/>

https://leetcode.com/problems/reorganize-string/description/

排序

<https://leetcode.com/problems/custom-sort-string/description/>

bst

https://leetcode.com/problems/kth-smallest-element-in-a-bst/discuss/

https://leetcode.com/problems/kth-smallest-element-in-a-bst/discuss/63659/What-if-you-could-modify-the-BST-node's-structure

# 数学

## 排列组合

### Catalan

* *Cn* is the number of [Dyck words](https://en.wikipedia.org/wiki/Dyck_word)[[2]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-2) of length 2*n*. A Dyck word is a [string](https://en.wikipedia.org/wiki/String_(computer_science)) consisting of *n* X's and *n* Y's such that no initial segment of the string has more Y's than X's. For example, the following are the Dyck words of length 6:

XXXYYY     XYXXYY     XYXYXY     XXYYXY     XXYXYY.

* Re-interpreting the symbol X as an open [parenthesis](https://en.wikipedia.org/wiki/Bracket#Parentheses) and Y as a close parenthesis, *Cn* counts the number of expressions containing *n* pairs of parentheses which are correctly matched:

((()))     ()(())     ()()()     (())()     (()())

* *Cn* is the number of different ways *n* + 1 factors can be completely [parenthesized](https://en.wikipedia.org/wiki/Bracket) (or the number of ways of [associating](https://en.wikipedia.org/wiki/Associativity) *n*applications of a [binary operator](https://en.wikipedia.org/wiki/Binary_operator)). For *n* = 3, for example, we have the following five different parenthesizations of four factors:

((ab)c)d     (a(bc))d     (ab)(cd)     a((bc)d)     a(b(cd))

[](https://en.wikipedia.org/wiki/File:Tamari_lattice,_trees.svg)

The [associahedron](https://en.wikipedia.org/wiki/Associahedron) of order 4 with the C4=14 full binary trees with 5 leaves

* Successive applications of a binary operator can be represented in terms of a full [binary tree](https://en.wikipedia.org/wiki/Binary_tree). (A rooted binary tree is *full* if every vertex has either two children or no children.) It follows that *Cn* is the number of full binary [trees](https://en.wikipedia.org/wiki/Tree_(graph_theory)) with *n* + 1 leaves:

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

* *Cn* is the number of non-isomorphic ordered trees with *n* vertices. (An ordered tree is a rooted tree in which the children of each vertex are given a fixed left-to-right order.)[[3]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-3)
* *Cn* is the number of monotonic [lattice paths](https://en.wikipedia.org/wiki/Lattice_path) along the edges of a grid with *n* × *n* square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for "move right" and Y stands for "move up".

The following diagrams show the case *n* = 4:

[](https://en.wikipedia.org/wiki/File:Catalan_number_4x4_grid_example.svg)

This can be succinctly represented by listing the Catalan elements by column height:[[4]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-4)

[0,0,0,0][0,0,0,1][0,0,0,2][0,0,1,1]

[0,1,1,1] [0,0,1,2] [0,0,0,3] [0,1,1,2][0,0,2,2][0,0,1,3]

[0,0,2,3][0,1,1,3] [0,1,2,2][0,1,2,3]

[](https://en.wikipedia.org/wiki/File:Tamari_lattice,_hexagons.svg)

The triangles correspond to internal nodes of the binary trees.

* A [convex polygon](https://en.wikipedia.org/wiki/Convex_polygon) with *n* + 2 sides can be cut into [triangles](https://en.wikipedia.org/wiki/Triangle) by connecting vertices with non-crossing [line segments](https://en.wikipedia.org/wiki/Line_segment) (a form of [polygon triangulation](https://en.wikipedia.org/wiki/Polygon_triangulation)). The number of triangles formed is *n* and the number of different ways that this can be achieved is *Cn*. The following hexagons illustrate the case *n* = 4:

[](https://en.wikipedia.org/wiki/File:Catalan-Hexagons-example.svg)

* *Cn* is the number of [stack](https://en.wikipedia.org/wiki/Stack_(data_structure))-sortable [permutations](https://en.wikipedia.org/wiki/Permutation) of {1, ..., *n*}. A permutation *w* is called [stack-sortable](https://en.wikipedia.org/wiki/Stack-sortable_permutation) if *S*(*w*) = (1, ..., *n*), where *S*(*w*) is defined recursively as follows: write *w* = *unv* where *n* is the largest element in *w* and *u* and *v* are shorter sequences, and set *S*(*w*) = *S*(*u*)*S*(*v*)*n*, with *S* being the identity for one-element sequences.
* *Cn* is the number of permutations of {1, ..., *n*} that avoid the [permutation pattern](https://en.wikipedia.org/wiki/Permutation_pattern) 123 (or, alternatively, any of the other patterns of length 3); that is, the number of permutations with no three-term increasing subsequence. For *n* = 3, these permutations are 132, 213, 231, 312 and 321. For *n* = 4, they are 1432, 2143, 2413, 2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.
* *Cn* is the number of [noncrossing partitions](https://en.wikipedia.org/wiki/Noncrossing_partition) of the set {1, ..., *n*}. [*A fortiori*](https://en.wikipedia.org/wiki/A_fortiori_argument), *Cn* never exceeds the *n*th [Bell number](https://en.wikipedia.org/wiki/Bell_number). *Cn* is also the number of noncrossing partitions of the set {1, ..., 2*n*} in which every block is of size 2. The conjunction of these two facts may be used in a proof by [mathematical induction](https://en.wikipedia.org/wiki/Mathematical_induction) that all of the *free* [cumulants](https://en.wikipedia.org/wiki/Cumulant)of degree more than 2 of the [Wigner semicircle law](https://en.wikipedia.org/wiki/Wigner_semicircle_law) are zero. This law is important in [free probability](https://en.wikipedia.org/wiki/Free_probability) theory and the theory of [random matrices](https://en.wikipedia.org/wiki/Random_matrices).
* *Cn* is the number of ways to tile a stairstep shape of height *n* with *n* rectangles. The following figure illustrates the case *n* = 4:

[](https://en.wikipedia.org/wiki/File:Catalan_stairsteps_4.svg)

* *Cn* is the number of rooted [binary trees](https://en.wikipedia.org/wiki/Binary_tree) with *n* internal nodes (*n* + 1 [leaves](https://en.wikipedia.org/wiki/Tree_(graph_theory)#Definitions) or external nodes). Illustrated in following Figure are the trees corresponding to *n* = 0,1,2 and 3. There are 1, 1, 2, and 5 respectively. Here, we consider as binary trees those in which each node has zero or two children, and the internal nodes are those that have children.

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

* *Cn* is the number of ways to form a "mountain range" with *n* upstrokes and *n* downstrokes that all stay above a horizontal line. The mountain range interpretation is that the mountains will never go below the horizon.

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

* *Cn* is the number of [standard Young tableaux](https://en.wikipedia.org/wiki/Young_tableau#Tableaux) whose diagram is a 2-by-*n* rectangle. In other words, it is the number of ways the numbers 1, 2, ..., 2*n* can be arranged in a 2-by-*n* rectangle so that each row and each column is increasing. As such, the formula can be derived as a special case of the [hook-length formula](https://en.wikipedia.org/wiki/Young_tableau#Dimension_of_a_representation).
* *Cn* is the number of ways that the vertices of a convex 2*n*-gon can be paired so that the line segments joining paired vertices do not intersect. This is precisely the condition that guarantees that the paired edges can be identified (sewn together) to form a closed surface of genus zero (a topological 2-sphere).
* *Cn* is the number of [semiorders](https://en.wikipedia.org/wiki/Semiorder) on *n* unlabeled items.[[5]](https://en.wikipedia.org/wiki/Catalan_number#cite_note-5)

## 概率

编4.1 金刚坐飞机

如果金刚坐在第n个位置，那么第i个乘客坐在自己位置的概率为f(n)

注意有个隐含条件金刚的机票座位号是1

# End