目录

10 最优效率的位运算

11 综合运用各种数据结构

12 必知必会之字符串

13 啊哈，链表

14 啊哈，树

15 有图才有真相

16 DIY数据结构

第1章 算法概述

概述

第一个大分类是算法。本书先从最简单的贪心算法讲起，然后逐渐进阶到二分查找、排序算 法和搜索算法，最后是难度比较高的动态规划和分治算法。
第二个大分类是数学，包括偏向纯数学的数学问题，和偏向计算机知识的位运算问题。这类问题通常用来测试你是否聪敏，在实际工作中并不常用，笔者建议可以优先把精力放在其它大类上。
第三个大分类是数据结构，包括常见数据结构、字符串处理、链表、树和 图。其中，链表、树、和图都是用指针表示的数据结构，且前者是后者的子集。最后我们也将介绍绍一些更加复杂的数据结构，比如经典的并查集和LRU。

Code

全部代码都上传到了github地址，代码都通过了leetcode的测试用例。
代码中包含测试用例和源码

https://github.com/zsjun/leetcode.git

参考

参考A LeetCode Grinding Guide (C++ Version).pdf，因为原书是使用c++实现的，这里按照原来的目录全部采用js重新实现。

第2章贪心算法

什么是贪心算法?

1 贪心算法就是在每一步都要选择最优结果，当局部最优结果结束后，得到全局最优结果。但是其实很多看似可以使用贪心算法的，往往使用贪心算法并不能得到最优解。

445 分配饼干

题目描述

有一群孩子和一堆饼干，每个孩子有一个饥饿度，每个饼干都有一个大小。每个孩子只能吃最多一个饼干，且只有饼干的大小大于孩子的饥饿度时，这个孩子才能吃饱。求解最多有多少孩子可以吃饱。

例子

输入两个数组，分别代表孩子的饥饿度和饼干的大小。输出最多有多少孩子可以吃饱的数量。
Input: [1,2], [1,2,3]
Output: 2
在这个样例中，我们可以给两个孩子喂[1,2]、[1,3]、[2,3]这三种组合的任意一种。

思考 1

这里很明显是属于贪心算法，优先把饥饿度最小的孩子用最小的饼干喂饱，一定可以得到最多的。

实现1

```
export default (childrens, cookies) => {
```

```
  if (!childrens || childrens.length === 0) return 0;
  childrens.sort();
  cookies.sort();
  // 最多可以有多少孩子
  let res = 0;
  // 已经分配的饼干索引
  let cookiesIndex = 0;
  // 优先使用最小的饼干喂饱饥饿最下的孩子
  for (let i = 0; i < childrens.length; i++) {
    for (let j = cookiesIndex; j < cookies.length; j++) {
      if (cookies[j] >= childrens[i]) {
        res++;
        cookiesIndex = j + 1;
        break;
      }
    }
  }
  return res;
};
```

算法时间复杂度 O(childrens.length*cookies.length), 空间复杂度 O(1)

763-划分字母区间

题目描述

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

例子

输入：S = "ababcbacadefegdehijhklij"
输出：[9,7,8]
解释：
划分结果为 "ababcbaca", "defegde", "hijhklij"。
每个字母最多出现在一个片段中。
像 "ababcbacadefegde", "hijhklij" 的划分是错误的，因为划分的片段数较少。

提示

S的长度在[1, 500]之间。
S只包含小写字母 'a' 到 'z' 。

思考 1

1 思路很简单，利用递归，不断的去查找
2 利用贪心算法和双指针，首先使用一个数组存储所有字符出现的最后位置，然后利用双指针，一个指向子串开始的位置，一个指向子串结束的位置，然后不断查找，当发现一个字母已经到达了它在字符串中的最后位置的时候，就是相当于发现了一个符合条件的子串。
代码很简单，稍微看下，就明白了。

这里可以得到一个小提示，一旦涉及到字符串的时候，很自然的就要想到使用一个长度为26的数组来存储。

实现1

```
/**
 * @param {string} S
 * @return {number[]}
 */

const getBigStr = (S, begin, res) => {
  let max = begin;
  const len = S.length;
  if (begin >= S.length) {
    return;
  }
  const lastIndex = S.lastIndexOf(S[begin]);
  if (lastIndex !== -1) {
    max = Math.max(max, lastIndex);
```

```
    let s1 = S.substring(begin, lastIndex + 1);
    for (let i = 1; i < s1.length; i++) {
      const newLastIndex = S.lastIndexOf(s1[i]);
      if (newLastIndex > max) {
        max = newLastIndex;
        s1 = S.substring(begin, max + 1);
      }
    }
    res.push(S.substring(begin, max + 1));
  } else {
    res.push(S[begin]);
    max = begin++;
  }
  return max;
};
export default (S) => {
  let res = [];
  let max = -1;
  while (max < S.length) {
    max = getBigStr(S, max + 1, res);
  }
  return res.map((item) => item.length);
};
```

实现2

```
export default (S) => {
  if (S == null || S.length === 0) {
    return null;
  }
  const list = [];
  // 记录每个字符出现在字符串中的最后的位置
  const map = new Array(26).fill(0);

  for (let i = 0; i < S.length; i++) {
    map[S.charCodeAt(i) - 97] = i;
  }
  // 记录每个子串出现的开始和结束
  let last = 0;
  let start = 0;
  for (let i = 0; i < S.length; i++) {
    last = Math.max(last, map[S.charCodeAt(i) - 97]);
    if (last === i) {
      list.push(last - start + 1);
      start = last + 1;
```

```
    }
  }
  return list;
};
```

时间复杂度O（n），空间复杂度O（1）

435 区间问题

题目描述

给定多个区间，计算让这些区间互不重叠所需要移除区间的最少个数。起止相连不算重叠。

例子

输入是一个数组，数组由多个长度固定为 2 的数组组成，表示区间的开始和结尾。输出一个 整数，表示需要移除的区间数量。

Input: [[1,2], [2,4], [1,3]]
Output: 1

在这个样例中，我们可以移除区间 [1,3]，使得剩余的区间 [[1,2], [2,4]] 互不重叠

思考 1

这里的贪心策略使用是不明显的，需要做一些转换，但是如何转换呢?

这个首先得自己思考下，然后才能有收获，不然看完题解也很快就忘记了。

思考最好的方法是多写几个测试用例，看下如何解决。

比如测试用例中[[1,2], [2,4], [1,3]]，为什么会删除[1,3]，很明显是[1,3]与[1,2]的区别就是3比2大，如果[1,3] 变成[0,5]呢?
则输入的数组变成[[1,2], [2,4], [0,5]]，可以很明显看出删除[0,5]
如果输入的数组变成[[1,2], [2,4], [5,7]]呢，可以明显看出不需要删除任何数组

通过以上的例子是不是发现了什么?

所谓的贪心算法就是找到局部的最优解，然后判断局部的最优解是否是全局的最优解?

这里是不是发现如果想删除最少的数组，只需要把跨度最大的数组删除就可以了。

也就是如果两个区间重合，删除其中一个区间的结尾最小的区间，因为区间结尾最小，说明以后删除的区间也就越少，也就是这里的贪心。

实现1

```
/**
 * @param {number[][]} intervals
 * @return {number}
 */
export default (intervals) => {
  if (!intervals || intervals.length === 0 || intervals.length === 1) return 0;
```

```
  intervals.sort((a, b) => a[1] - b[1]);
  let min = 0;
  for (let i = 1; i < intervals.length; ) {
    if (
      (intervals[i][0] < intervals[i - 1][1] && intervals[i][0] >= intervals[i - 1][0]) ||
      intervals[i][0] < intervals[i - 1][0]
    ) {
      min++;
      intervals.splice(i, 1);
    } else {
      i++;
    }
  }
  return min;
};
```

时间复杂度O（nlgn）空间复杂度O（1）

实现2

```
/**
```

```
 * @param {number[][]} intervals
 * @return {number}
 */

export default (intervals) => {
  if (!intervals || intervals.length < 2) return 0;
  intervals.sort((a, b) => {
    if (a[0] === b[0]) {
      return a[1] - b[1];
    } else {
      return a[0] - b[0];
    }
  });
  let count = 0;
  for (let i = 1; i < intervals.length; ) {
    if (intervals[i][0] < intervals[i - 1][1]) {
      if (intervals[i][1] > intervals[i - 1][1]) {
        intervals.splice(i, 1);
      } else {
        intervals.splice(i - 1, 1);
      }
      count++;
    } else {
```

```
        i++;
      }
    }
    return count;
};
```

时间复杂度O（nlgn）空间复杂度O（1）

605. 种花问题

题目描述

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 n 。能否在不打破种植规则的情况下种入 n 朵花？能则返回True，不能则返回False。

例子

输入是一个数组，数组由多个长度固定为 2 的数组组成，表示区间的开始和结尾。输出一个 整数，表示需要移除的区间数量。

输入: flowerbed = [1,0,0,0,1], n = 1

输出: True

在这个样例中，可以把花种在2的位置上

输入: flowerbed = [1,0,0,0,1], n = 2
输出: False

因为这里有两颗花，不论第一棵花种在哪里，都会有相连的，从而导致有连在一起的两颗花

注意:

1 数组内已种好的花不会违反种植规则。
2 输入的数组长度范围为 [1, 20000]。
3 n 是非负整数，且不会超过输入数组的大小。

思考 1

贪心算法首先最重要的就是要寻找到最优解?
那这里的最优解是什么呢？或者换句话说我们怎么做才是最贪心的呢?

当 [1,0,0,0,1], n = 1的时候，最贪心的肯定是从第一个可以种的位置种植

当 [1,0,0,0,1], n = 2的时候，最贪心的肯定是从第一个可以种的位置种植,如果发现第二个没有位置可以种植，则返回false

实现1

```
/**
 * @param {number[]} flowerbed
 * @param {number} n
 * @return {boolean}
 */
// [1, 0, 0, 0, 1]
// [1,0,0,0,1,0,0]
export default (flowerbed, n) => {
  for (let i = 0; i < flowerbed.length; ) {
    if (i === 0 && flowerbed[i] === 0 && flowerbed[i + 1] !== 1) {
      flowerbed[i] === 1;
      n--;
      i += 2;
    } else if (i === flowerbed.length - 1 && flowerbed[i] === 0 && flowerbed[i - 1] !== 1) {
      n--;
      i++;
    } else if (flowerbed[i] === 0 && flowerbed[i - 1] !== 1 && flowerbed[i + 1] !== 1) {
      flowerbed[i] = 1;
      n--;
      i += 2;
    } else {
      i++;
```

```
    }
  }
  return n <= 0;
};
```

时间复杂度O（n）空间复杂度O（1）

452. 用最少数量的箭引爆气球

题目描述

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以纵坐标并不重要，因此只要知道开始和结束的横坐标就足够了。开始坐标总是小于结束坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

给你一个数组 points ，其中 points [i] = [xstart,xend] ，返回引爆所有气球所必须射出的最小弓箭数。

例子1

输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球

例子2

输入：points = [[1,2],[3,4],[5,6],[7,8]]
输出：4

例子3

输入：points = [[1,2],[2,3],[3,4],[4,5]]
输出：2

注意:

1 0 <= points.length <= 104
2 points[i].length == 2
3 -231 <= xstart < xend <= 231 - 1

思考 1

当 [[10,16],[2,8],[1,6],[7,12]]的时候，我们想用最少的箭去射爆更多的气球，很自然的能够想到我们射出的箭肯定得能够穿越更多的空间。

比如这个例子中[2,8],[1,6]，我们需要用2-6中间的任何一支箭，比如2，3，4，5，6，但是我们同时又希望我们这支箭可以射爆更多的区间，那应该从2，3，4，5，6中选择那只箭呢?

思考一下

很容易就想到肯定是最大的那支箭，也就是6，因为只有越大，我们才能射爆更多的其他区间。

所以这里也就是我们的最优解。

实现1

```
/**
 * @param {number[][]} points
 * @return {number}
 */
export default (points) => {
  const rowLen = points.length;
  if (rowLen === 0) return 0;
  const colLen = 2;
  points.sort((a, b) => a[0] - b[0]);
  let count = 1;
  let pre = points[0];
  for (let i = 1; i < points.length; i++) {
    if (points[i][0] < pre[1]) {
      pre[0] = Math.max(points[i][0], pre[0]);
      pre[1] = Math.min(points[i][1], pre[1]);
```

```
    } else if (points[i][0] === pre[1]) {
      pre[0] = points[i][0];
      pre[1] = points[i][0];
    } else {
      pre = points[i];
      count++;
    }
  }
  return count;
};
```

实现2

```
/**
 * @param {number[][]} points
 * @return {number}
 */
export default (points) => {
  const rowLen = points.length;
  if (rowLen === 0) return 0;
  points.sort((a, b) => a[1] - b[1]);
```

```
  let count = 1;
  // 首先使用最大的箭头，可以射到最多的
  let arrowNum = points[0][1];
  for (let i = 1; i < points.length; i++) {
    if (points[i][0] > arrowNum) {
      count++;
      arrowNum = points[i][1];
    }
  }
  return count;
};
```

时间复杂度O（nlgn） 空间复杂度O（1）

763-划分字母区间

题目描述

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一个字母只会出现在其中的一个片段。返回一个表示每个字符串片段的长度的列表。

例子1

输入: S = "ababcbacadefegdehijhklij"
输出: [9,7,8]
解释:
划分结果为 "ababcbaca", "defegde", "hijhklij"。
每个字母最多出现在一个片段中。
像 "ababcbacadefegde", "hijhklij" 的划分是错误的，因为划分的片段数较少。

注意:

1 S的长度在[1, 500]之间。
2 S只包含小写字母'a'到'z'。

思考 1

这里是主要是贪心算法，首先肯定想到了贪心

另外这里看到提示S只包含小写字母'a'到'z'，因为以前做过很多的题目，只要涉及到小写字母'a'到'z'，就联想到了使用一个大小为26的数组来存储a到z出现的次数或者位置

回到正题，看下测试用例

S = "ababcbacadefegdehijhklij"

划分结果为 "ababcbaca", "defegde", "hijhklij"。
每个字母最多出现在一个片段中。

很明显可以看出，每个划分字符串里边每个字母都是只出现在字符串里边，比如遍历字符串的时候，如果遇到a，则首先找到a最后出现的位

置，就会找到一个子串，如果子串里边所有的字母都出现在这个子串里边，则可以认定这个子串可以划分出来，如果不是则更新子串的大小，重新计算。实现1就是这个思路。

那这里和贪心有什么关系呢?

我的理解是就是每次把自己可以找到的最大子串找到。

当然这里实现1可以改进，不去不断的更新子串，利用一个数组存储每个字母在字符串中出现的最后一个位置，我们可以遍历字符串，当发现一个字母出现的位置是它在字符串中出现的位置的时候，就可以划分为一个子串，原理很简单，稍微看下代码就可以了。

实现1

```
/**
 * @param {string} S
 * @return {number[]}
 */

const getBigStr = (S, begin, res) => {
  let max = begin;
  const len = S.length;
  if (begin >= S.length) {
    return;
  }
```

```
  const lastIndex = S.lastIndexOf(S[begin]);
  if (lastIndex !== -1) {
    max = Math.max(max, lastIndex);
    let s1 = S.substring(begin, lastIndex + 1);
    for (let i = 1; i < s1.length; i++) {
      const newLastIndex = S.lastIndexOf(s1[i]);
      if (newLastIndex > max) {
        max = newLastIndex;
        s1 = S.substring(begin, max + 1);
      }
    }
    res.push(S.substring(begin, max + 1));
  } else {
    res.push(S[begin]);
    max = begin++;
  }
  return max;
};
export default (S) => {
  let res = [];
  let max = -1;
  while (max < S.length) {
    max = getBigStr(S, max + 1, res);
```

```
  }
  return res.map((item) => item.length);
};
```

实现2

```
export default (S) => {
  if (S == null || S.length === 0) {
    return null;
  }
  const list = [];
  // 记录每个字符出现在字符串中的最后的位置
  const map = new Array(26).fill(0);

  for (let i = 0; i < S.length; i++) {
    map[S.charCodeAt(i) - 97] = i;
  }
  // 记录每个子串出现的开始和结束
  let last = 0;
  let start = 0;
```

```
  for (let i = 0; i < S.length; i++) {
    last = Math.max(last, map[S.charCodeAt(i) - 97]);
    if (last === i) {
      list.push(last - start + 1);
      start = last + 1;
    }
  }
  return list;
};
```

时间复杂度O（n），空间复杂度O（1）

122. 买卖股票的最佳时机 II

题目描述

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

例子1

输入: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。

例子2
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

例子3

输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0

注意:

1 1 <= prices.length <= 3 * 10 ^ 4
2 0 <= prices[i] <= 10 ^ 4

思考 1

这里就很简单了，我们如果想获取收益最大，肯定是买在最低点，卖在最高点。

我们只需要使用贪心，找到前面连续降的最低点，然后卖在连续升的最高点就可以了。

实现1

```
/**
 * @param {number[]} prices
 * @return {number}
 */

export default (prices) => {
  let n = prices.length,
    lastBuy = -A[0],
    lastSold = 0;
  if (n === 0) return 0;

  for (let i = 1; i < n; i++) {
    let curBuy = Math.max(lastBuy, lastSold - A[i]);
    let curSold = Math.max(lastSold, lastBuy + A[i]);
    lastBuy = curBuy;
    lastSold = curSold;
```

```
    }

    return lastSold;
};
```

时间复杂度O（n），空间复杂度O（1）

406. 根据身高重建队列

题目描述

假设有打乱顺序的一群人站成一个队列。 每个人由一个整数对 (h, k) 表示，其中 h 是这个人的身高，k 是应该排在这个人前面且身高大于或等于 h 的人数。 例如：[5,2] 表示前面应该有 2 个身高大于等于 5 的人，而 [5,0] 表示前面不应该存在身高大于等于 5 的人。

编写一个算法，根据每个人的身高 h 重建这个队列，使之满足每个整数对 (h, k) 中对人数 k 的要求。

例子1

输入：[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
输出：[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

注意:

1 总人数少于 1100 人

思考 1

这道题目确实不是很好理解，但是如果看过解法之后，就可以很好的利用贪心来解释。

比如输入[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]的时候，当发现[7,0]的时候，前面要么没有区间，要么就是身高小于或者等于7且区间的结尾是0的区间。

那么是不是可以换个角度想，比如[7,0],它应该是放在结果中的0的位置，如果再发现一个比7小，或者等于7的区间，比如[5,0]，按照正常情况下，[5,0]也应该是放在结果的0的位置上的，可以在结果的0的位置上已经有[7,0]了所以[5,0]需要插入到[7,0]的前面。

所以这里可以先按照区间开头进行降序排序，当区间开头相同的时候，进行区间结尾的升序排序。然后依次插入到一个空数组中就可以了。

这里的贪心，其实不是很明显，可能贪心体现在先保证区间开头最大的位置放到数组中合理的位置。

实现1

```
/**
 * @param {number[][]} people
 * @return {number[][]}
 */
const swap = (people, i, j) => {
```

```
    const temp = people[j];
    people[j] = people[i];
    people[i] = temp;
  };
  export default (people) => {
    if (!people) return [];
    people.sort((o1, o2) => {
      return o1[0] !== o2[0] ? o2[0] - o1[0] : o1[1] - o2[1];
    });
    const res = [];
    for (let i = 0; i < people.length; i++) {
      res.splice(people[i][1], 0, people[i]);
    }
    return res;
  };
```

时间复杂度O（nlgn），空间复杂度O（n）

665. 非递减数列

题目描述

给定一个长度为 n 的整数数组，你的任务是判断在最多改变 1 个元素的情况下，该数组能否变成一个非递减数列。

我们是这样定义一个非递减数列的： 对于数组中所有的 i (1 <= i < n)，满足 array[i] <= array[i + 1]。

例子1

输入: [4,2,3]

输出: True

解释: 你可以通过把第一个4变成1来使得它成为一个非递减数列。

例子2

输入: [4,2,1]

输出: False

解释: 你不能在只改变一个元素的情况下将其变为非递减数列。

注意:

1 n 的范围为 [1, 10,000]

思考 1

这里也很简单，就是不断遍历，当发现应该需要序号i的时候，如何根据i前后的数来确定修改为什么。""

这里的贪心可能就是体现在如果根据需要修改的序号i的前后，来确定应该修改为什么。""

如果已经修改过一次了，后面如果发现还需要修改，直接返回false

实现1

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
export default (nums) => {
  let hasChangedNum = 0;
  for (let i = 1; i < nums.length && hasChangedNum <= 1; i++) {
    if (nums[i] < nums[i - 1]) {
      hasChangedNum++;
      if (nums[i - 2] <= nums[i] || i < 2) {
        nums[i - 1] = nums[i];
      } else {
        nums[i] = nums[i - 1];
      }
    }
  }
  return hasChangedNum <= 1;
```

```
};
```

时间复杂度O（n），空间复杂度O（1）

55. 跳跃游戏

题目描述

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。""

例子1

输入: [2,3,1,1,4]""

输出: True""

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。""

例子2

输入: [3,2,1,0,4]""

输出: False""

解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。""

思考 1

这里很容易理解，我们跳第一步的时候，有一个可以跳的范围，在这个范围内，我们肯定想着下一步能跳的更远，那我们应该怎么选择呢？""

肯定是选择能跳最远的，这样下一步能选择的范围更大，也更容易到达终点""

思想虽然是一样的，但是实现也不同，一种是确定第一步能跳到最远，然后反过来遍历范围，找到可以跳到最远的距离。可以看下实现1""

另外一种是把不断遍历可到达的范围，然后遍历可到达的范围，不断更新最大可到达的范围，但是这种变量在for循环中，除非逻辑特别清晰，不然很容易出错。""

如果看下这里的实现2，cover放到了for循环里边，这样for循环可能就不是固定的，而是不断的变化的，这样如果出现问题可能不是很好查找，可以使用另外一种实现方法，也就是实现3""

实现1

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
```

```
export default (nums) => {
  // 只有一个的时候，直接返回
  const len = nums.length;
  if (len === 1) {
    return true;
  }
  let maxRange = nums[0];
  let begin = 0;
  for (let i = 1; i < nums.length; i++) {
    let tempIndex = begin;
    for (let j = begin + 1; j <= maxRange; j++) {
      if (j - begin + nums[j] > maxRange) {
        maxRange = j - begin + nums[j];
        tempIndex = j;
      }
    }
    begin = tempIndex;
    if (maxRange >= len - 1) {
      return true;
    }
  }
  return false;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
export default (nums) => {
  // 只有一个元素，就是能达到
  if (nums.length === 1) return true;
  // 覆盖的范围
  let cover = 0;
  for (let i = 0; i <= cover; i++) {
    // 注意这里是小于等于cover
    cover = Math.max(i + nums[i], cover);
    if (cover >= nums.length - 1) return true; // 说明可以覆盖到终点了
  }
  return false;
};
```

实现3

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
// 68 ms    39.6 MB
export default (nums) => {
  const len = nums.length;
  let reach = 0;
  for (let i = 0; i < len; ++i) {
    // 走不到i的位置，也就是可以直接return false
    if (i > reach) {
      return false;
    }
    // 到达终点了
    if (reach >= len - 1) {
      return true;
    }
    // 更新可以到达的位置
    reach = Math.max(reach, i + nums[i]);
  }
```

```
    return false;
};
```

实现1的时间复杂度O（n^2），空间复杂度O（1）""
实现2的时间复杂度O（n），空间复杂度O（1）""
实现3的时间复杂度O（n），空间复杂度O（1）""

45. 跳跃游戏2

题目描述

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

""

例子1

输入: [2,3,1,1,4]""

输出: 2""

解释: 跳到最后一个位置的最小跳跃数是 2。
从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。""

例子2""

输入: [2,3,0,1,4]""

输出: 2""

提示：""
1 注意这里假设你一定可以达到数组的最后一个位置""

思考 1

看到题目，因为前面做过55这道题目，肯定会联想到前面的题目解决方法，改变一下。""

但是感觉思路没想清楚，这时候突然想起一个规律来，算法中很多的题目，如果正面走不通，如果逆向思维解决一下，瞬间豁然开朗。""

这时候，大家可以自己考虑一下，如果从最后一个往前查找，我们应该怎么选择呢？""

我们会怎么贪心？""

肯定是要贪到离我们最远的地方，也就是下标最小的时候。""

一步之后，继续按照这个思路来寻找下一个，直到我们走到起点，也就是数组的开始。""

当然这样时间复杂度要高一些""

实现可以看下实现1 ""

那么这里既然是跳跃游戏2，我们肯定也可以在跳跃游戏1的基础上修改下，让它符合这里的条件也可以。""

那么问题来了，应该如何修改呢？""

不管任何问题，首先都要明确目标，我们的目标是找到最小的步数到达终点，而且是肯定有一条到达终点的路""

这里只要我们稍微变化一下，就可以了，比如我们走了一步，发现这一步到达不了终点，这个时候就要加一步了，然后我们在我们可以走到的步数中找到覆盖范围最大的，看看这个覆盖范围能不能到达终点，如此反复就可以了""

思路很简单，就是当前一步覆盖的范围到达不了终点，我们就在这个覆盖范围内，找到下一步可以覆盖更大范围的，如此反复""

思路比较简单，可以看下实现2""

这里同样的思路，不过实现方式可能稍微有些区别，可以看下实现3""

这道题目也是hard难度，可以看到也是比较简单的，所以leetcode上的难度稍微参考一下就好，不要过于在意""

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// 508 ms    39.9 MB
export default (nums) => {
  if (nums.length === 1) {
```

```
    return 0;
  }
  let res = 0;
  for (let j = nums.length - 1; j >= 1; ) {
    let min = j;
    for (let i = 0; i < j; i++) {
      if (nums[i] >= j - i) {
        min = Math.min(i, min);
      }
    }
    j = min;
    res++;
    if (min === 0) {
      return res;
    }
  }
  return res;
};
```

实现2

```
```

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
export default (nums) => {
  // 只有一个元素，就是能达到
  if (nums.length === 1) return true;
  // 覆盖的范围
  let cover = 0;
  for (let i = 0; i <= cover; i++) {
    // 注意这里是小于等于cover
    cover = Math.max(i + nums[i], cover);
    if (cover >= nums.length - 1) return true; // 说明可以覆盖到终点了
  }
  return false;
};
```

实现2

```
/**
```

```
 * @param {number[]} nums
 * @return {boolean}
 */
// 88 ms    40.1 MB
export default (nums) => {
  const len = nums.length;
  // 如果只有一个元素，我们直接返回0
  if (len === 1) {
    return 0;
  }
  // 目前的步数可以达到的范围
  let reach = 0;
  let res = 0;

  for (let i = 0; i < len; ) {
    // 更新走一步可以到达的位置
    reach = Math.max(reach, i + nums[i]);
    // 如果走一步到达终点了，则返回
    if (reach >= len - 1) {
      return res + 1;
    }
    // 如果这一步，无法到达终点，我们必须走下一步了，步数加1，
    if (reach < len) {
```

```
      res++;
      // 从i+1到reach，更新可以到达的最大距离，
      let tempReach = reach;
      for (let j = i + 1; j <= tempReach; j++) {
        if (j + nums[j] > reach) {
          reach = j + nums[j];
          i = j;
          // 因为这里还是走了一步
          if (reach >= len - 1) {
            return res + 1;
          }
        }
      }
    }
  }
  return res;
};
```

实现3

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
//     92 ms    39.7 MB
export default (nums) => {
  // max 表示当前步数可以到达的最大距离
  let max = 0;
  // 表示在当前步数步数覆盖范围内下一步可以达到的最大距离
  let nextMax = 0;
  // 步数
  let jumps = 0;

  nums.some((val, index) => {
    // 如果当前步数可以达到终点了，直接返回步数
    if (max >= nums.length - 1) {
      return jumps;
    }
    //
    nextMax = Math.max(index + val, nextMax);
    // 如果当前的步数已经走到了结束，则加一步
    if (index === max) {
```

```
      max = nextMax;
      jumps++;
    }
  });

  return jumps;
};
```

实现1的时间复杂度O（n^2），空间复杂度O（1）""
实现2的时间复杂度O（n），空间复杂度O（1）""
实现3的时间复杂度O（n），空间复杂度O（1）""

贪心算法总结

贪心算法除非特备明显的可以看出需要使用贪心的，其他大多数没有必要刻意使用贪心，大多需要通过排序，修改来让数据结构变化，根据条件发现是否可以使用贪心，但大多数其实都可以无意识的使用贪心。

第3章双指针算法

什么是双指针算法?

1 双指针算法一般是指有两个指针，可以用来遍历，也可以用来查找，还可以当做滑动窗口，双指针算法的关键是寻找到什么时候更新low指

针，什么时候更新high指针，也就是根据什么条件来更新指针，和贪心算法主要找到什么是最贪心差不多。

167. 两数之和 II - 输入有序数组

题目描述

给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。
函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

> 返回的下标值（index1 和 index2）不是从零开始的。
> 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

例子

输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。

思考 1

不管做任何的算法，首先要理解好前提，也就是已知条件，比如这里已经明确表示了这是已经升序的了，所以很容易想到使用双指针，一个指针指向最小，一个指向最大，然后根据是否和targe相等不断的移动指针。

实现1

```
/**
 * @param {number[]} numbers
 * @param {number} target
 * @return {number[]}
 */
export default (numbers, target) => {
  let minPoints = 0;
  const len = numbers.length;
  let maxPoints = len - 1;
  const res = [];
  while (minPoints < maxPoints) {
    if (numbers[minPoints] + numbers[maxPoints] === target) {
      res.push(++minPoints);
      res.push(++maxPoints);
    } else if (numbers[minPoints] + numbers[maxPoints] > target) {
      maxPoints--;
    } else {
```

```
      minPoints++;
    }
  }
  return res;
};
```

算法时间复杂度 O(n), 空间复杂度 O(1)

88. 合并两个有序数组

题目描述

给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

说明:

1 初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。
2 你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

例子

输入：
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6], n = 3

输出：[1,2,2,3,5,6]

提示：

1 -10^9 <= nums1[i], nums2[i] <= 10^9
2 nums1.length == m + n
3 nums2.length == n

思考 1

1 采用两个指针，一个指向nums1的末尾，一个指向nums2的末尾，然后对比就可以了，不断的插入到num1的末尾，算法比较简单。
2 还有数组很容易就想到了排序，可以先把nums2加入到nums1，然后可以进行排序就可以了。

实现1

```
/**
 * @param {number[]} nums1
 * @param {number} m
 * @param {number[]} nums2
 * @param {number} n
 * @return {void} Do not return anything, modify nums1 in-place instead.
 */
// [0];
```

```
// (0)[1];
// 1;
export default (nums1, m, nums2, n) => {
  let i = m - 1;
  let j = n - 1;
  let k = m + n - 1;
  for (let m1 = m; m1 < m + n; m1++) {
    nums1[m1] = 0;
  }

  while (k >= 0) {
    if (nums2[j] >= nums1[i]) {
      nums1[k] = nums2[j];
      k--;
      j--;
    } else {
      if (i < 0 && j >= 0) {
        while (j >= 0) {
          nums1[k] = nums2[j];
          k--;
          j--;
        }
      } else if (j < 0 && i >= 0) {
```

```
        while (i >= 0) {
          nums1[k] = nums1[i];
          k--;
          i--;
        }
      } else if (i >= 0 && j >= 0) {
        nums1[k] = nums1[i];
        k--;
        i--;
      } else {
        k--;
      }
    }
  }
  return nums1;
};
```

算法时间复杂度 O(m+n), 空间复杂度 O(1)

实现2

```
/**
```

```
 * @param {number[]} nums1
 * @param {number} m
 * @param {number[]} nums2
 * @param {number} n
 * @return {void} Do not return anything, modify nums1 in-place instead.
 */
export default (nums1, m, nums2, n) => {
  for (let i = m; i < m + n; i++) {
    nums1[i] = nums2[i - m];
  }
  nums1.sort((a, b) => a - b);
  return nums1;
};
```

算法时间复杂度 O(m+nlg(m+n)), 空间复杂度 O(1)

142. 环形链表 II

题目描述

给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意，pos 仅仅是用于标识环的情况，并不会作为参数传递到函数中。

说明:

1 不允许修改给定的链表。

进阶：

你是否可以使用 O(1) 空间解决此题？。

例子1

输入：head = [3,2,0,-4], pos = 1

输出：返回索引为 1 的链表节点,因为链表中有一个环，其尾部连接到第二个节点。

例子2

输入：head = [1,2], pos = 0

输出：链表中有一个环，其尾部连接到第一个节点。。

例子3

输入：输入：head = [1], pos = -1

输出：返回 null，链表中没有环。

提示：

1 链表中节点的数目范围在范围 [0, 104] 内
2 -10^5 <= Node.val <= 10^5
3 pos 的值为 -1 或者链表中的一个有效索引

思考 1

1 因为以前做过一道印象特别深刻的题目，就是判断链表中是否存在环，采用快慢指针，慢指针走一步，快指针走两步，所以这里也准备使用快慢指针，可是这里并不是求链表中是否存在环，而是要找到环的开始节点，后来想到以前还做过一道龟兔赛跑，查找节点的题目，可是还是没有头绪
2 后来看了题解才想起了，这个题目是做过的，但是还没理解题目的真谛

可以发现真正的要点就是a=c，所以要找起始点就很容易了，可以分别从开始出发一个指针，从z节点出发一个指针，当两者相遇的时候就是环的起始点。

这种解法其实就是知道就知道，不知道除非看过，不然很难想出来

这种固定套路的记住就可以了

实现1

```
/**
 * Definition for singly-linked list.
```

```
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */

/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var detectCycle = function(head) {
  let temphead = head;
  let slow = head;
  let fast = head;

  do {
    slow = slow ? slow.next : null;
    fast = fast? fast.next : null;
    fast = fast ? fast.next : null;
  } while (slow !== fast && fast !== null);

  if (slow === fast && fast!==null) {
    while (temphead !== fast) {
```

```
      temphead = temphead.next;
      fast = fast.next;
    }
  }
  return fast || null;
};
```

算法时间复杂度 O(n), 空间复杂度 O(1)

76. 最小覆盖子串

题目描述

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

说明:

1 如果 s 中存在这样的子串，我们保证它是唯一的答案。

进阶：
你能设计一个在 o(n) 时间内解决此问题的算法吗?

例子1

输入：s = "ADOBECODEBANC", t = "ABC"

输出："BANC"

例子2

输入：s = "a", t = "a"

输出："a"

提示：

1 1 <= s.length, t.length <= 10^5
2 s 和 t 由英文字母组成

思考 1

1 这里首先想到的肯定是双指针，因为这个专题就是双指针， 所以首先想到的肯定是设置两个指针，一个slow，一个fast，
slow的位置肯定是小于或者等于fast指针的位置。
然后遍历字符串，找到包含所有t的子串，然后再不断更新slow指针，那么下一个问题就变成了按照什么规则更新slow指针?
然后很自然就会想到因为我们是找的最短的字符串，所以如果slow指针不断前进的时候，如果发现slow现在所指的字符不在t中，当然可以继续前进，因为slow在这种情况下前进，肯定不会影响最短字符串的是否包含t中的所有字符的。
当发现slow指针指的字符在t中的时候，这个时候因为字符串还没有遍历完，所以我们还是需要继续遍历下去的，否则不能肯定现在找到的是最短的字符串是s中能找到的包含t的最短字符串，这个时候就要更新slow指针，也就是让从我们找到的最短字符串中删除一个在t中的字符且在我们的找到的最短字符串中重复的次数不大于在t中出现的次数的字符。

这个题目有些不是很好理解的地方，可能就是在理解slow指针应该怎么前进?

比如 s="ADOBECODEBANC"，t="ABC"的时候，

我们可以很容易的发现slow = 0，fast = 5的时候，也就是"ADOBEC"的时候是包含所有t的字符的，这时候最短字符串就是"ADOBEC"

然后想办法从"ADOBEC" 删除掉一个在t中且在我们找到的最短字符串"ADOBEC"重复次数小于在t中重复的次数，当然这个例子中，t 不包含重复的字符，
可以发现A在t中，所以slow一直前进到1，然后fast前进到10，此时又发现新的包含t的字符串“DOBECODEBA”
因为D和O不在t中，所以slow可以一直前进到4，也就是变成slow=4,fast=10
这个时候重点就来了，slow下一步如何前进，如果此时slow前进到4，可以发现“B”在t中，那么是不是就删除"B",让slow=5，让fast继续寻找“B”呢？很明显不行，因为“ECODEBA”里边有"B",没有必要去寻找"B"
所以slow还得继续前进，一直前进到slow=7，也就是删除“C”,这个时候待查找字符串就变成了“ODEBA”,这样就可以让fast去寻找 “C”了。

如果还是不懂，可以看一下代码，主要看下是如何不断的更新slow的，
关键的一点是记住更新slow的目的是为了在已经查找的最短字符串中删除一个字符，然后让fast再去寻找到一个，形成新的最短字符串

通过这个题可以学习一下如何维持一个滑动窗口的状态，不断的从窗口中删除和不断的添加到滑动窗口

实现1

```
/**
 * @param {string} s
 * @param {string} t
 * @return {string}
 */
```

```
export default (s, t) => {
  const tMap = new Map();
  for (let i = 0; i < t.length; i++) {
    if (!tMap.has(t[i])) {
      tMap.set(t[i], 1);
    } else {
      let count = tMap.get(t[i]) + 1;
      tMap.set(t[i], count);
    }
  }

  let cnt = 0,
    slow = 0,
    resStart = 0,
    resLen = s.length + 1;

  for (let fast = 0; fast < s.length; fast++) {
    if (tMap.has(s[fast])) {
      let count = tMap.get(s[fast]) - 1;
      tMap.set(s[fast], count);
      if (count >= 0) {
        cnt++;
      }
```

```
      // 若目前滑动窗口已包含T中全部字符，
      // 则尝试将右移，在不影响结果的情况下获得最短子字符串
      while (cnt === t.length) {
        if (fast - slow + 1 < resLen) {
          resStart = slow;
          resLen = fast - slow + 1;
        }
        let count = tMap.get(s[slow]) + 1;
        tMap.set(s[slow], count);
        if (tMap.has(s[slow]) && count > 0) {
          cnt--;
        }
        slow++;
      }
    }
  }
  return resLen > s.length ? "" : s.substr(resStart, resLen);
};
```

这里的时间复杂度可以很容易的看出是O（s.length），空间复杂度O(t.length)

633. 平方数之和

题目描述

给定一个非负整数 c ，你要判断是否存在两个整数 a 和 b，使得 a^2 + b^2 = c 。

例子1

输入: 5
输出: True
解释: 1 * 1 + 2 * 2 = 5

例子2

输入：3
输出：false

例子3

输入：4
输出：true

例子4

输入：2
输出：true

例子5

输入：1
输出：true

提示：

1 0 <= c <= 231 - 1

思考 1

1 直接使用两个指针，遍历从1到Math.sqrt(c)就可以了,题目比较简单。

实现1

```
/**
 * @param {number} c
 * @return {boolean}
 */

export default (c) => {
  let high = Math.ceil(Math.sqrt(c));
  let low = 0;
  while (low <= high) {
```

```
    if (Math.pow(low, 2) + Math.pow(high, 2) === c) {
      return true;
    } else if (Math.pow(low, 2) + Math.pow(high, 2) > c) {
      high--;
    } else {
      low++;
    }
  }
  return false;
};
```

这里的时间复杂度可以很容易的看出是O（Math.sqrt(c)），空间复杂度O(1)

680. 验证回文字符串 II

题目描述

最多删除一个字符后，判断剩余的字符串是否为回文串。

例子1

输入: "aba"

输出: True

例子2

输入："abca"
输出：True
解释：可以删除c字符就可以了

提示：

1 字符串长度小于50000

思考 1

1 直接使用两个指针，一个指针从开头，一个从结尾，如果发现不一致就看下删除low指向的字符，看下剩下的是否是回文字符串，看下删除high指向的字符，剩下的是否是回文字符串，
最后如果发现删除了超过一个字符，则返回false，否则为true，思路比较简单。

实现1

```
/**
 * @param {string} s
 * @return {boolean}
 */
```

```
const isPalindrome = (s, low, high, count) => {
  if (count > 1) return false;
  while (low < high) {
    if (s[low] !== s[high]) {
      count++;
      return isPalindrome(s, low, high - 1, count) || isPalindrome(s, low + 1, high, count);
    } else {
      low++;
      high--;
    }
  }
  return true;
};
const validPalindrome = (s) => {
  if (!s) {
    return false;
  }
  if (s.length === 1) {
    return true;
  }
  let low = 0;
  let high = s.length - 1;
  let count = 0;
```

```
  return isPalindrome(s, 0, high, count);
};
export default isPalindrome;
```

这里的时间复杂度可以很容易的看出是O（n）因为我们不管递归多少次，最多也是遍历整个字符串进行对比，空间复杂度O(1)

524. 通过删除字母匹配到字典里最长单词

题目描述

给定一个字符串和一个字符串字典，找到字典里面最长的字符串，该字符串可以通过删除给定字符串的某些字符来得到。如果答案不止一个，返回长度最长且字典顺序最小的字符串。如果答案不存在，则返回空字符串。

例子1

输入: s = "abpcplea", d = ["ale","apple","monkey","plea"]
输出: "apple"

例子2

输入： s = "abpcplea", d = ["a","b","c"]
输出： "a"

提示：

1 所有输入的字符串只包含小写字母。
2 字典的大小不会超过 1000。
3 所有输入的字符串长度不会超过 1000。

思考 1

1 这个应该也很简单，主要是先排序数组，然后遍历数组中每个元素，假设数组中元素为s1,判断下s中是否存在s1中所有字符，且s1中出现的每个字符必须和出现在s中的相对顺序都一样，因为只能在s中删除字符变成s1,如果相对顺序不一致，就无法通过s中删除字符变成s1.

这里需要注意的是js中对于字符串的字典排序使用localeCompare

indexOf函数可以指定第二个参数

实现1

```
/**
 * @param {string} s
 * @param {string[]} d
 * @return {string}
 */
export default (s, d) => {
  // 首先按照长度降序排序，如果字符串长度相同，按照字典序升序排序
```

```
  d.sort((a, b) => {
    if (a.length !== b.length) {
      return b.length - a.length;
    }
    return a.localeCompare(b);
  });

  for (let i = 0; i < d.length; i++) {
    let index = -1;
    for (let j = 0; j < d[i].length; j++) {
      // 判断在d[i]中的字符是否也在s中，同时相对顺序是一定的
      index = s.indexOf(d[i][j], index + 1);
      // 如果不存在，就没必要进行下去了
      if (index < 0) {
        break;
      }
    }
    // 如果找到了，直接跳出
    if (index >= 0) {
      return d[i];
    }
  }
  return "";
```

```
};
```

这里的时间复杂度可以很容易的看出是Math.max(（d.length* d[i].length * s.length）,s.length* lg(s.length))
因为可以看到是三层循环和对s进行排序
空间复杂度O(1)

340. 找出至多包含k个不同字符的最长子串

题目描述

至多包含 K 个不同字符的最长子串

给定一个字符串 s ，找出至多包含 k 个不同字符的最长子串 T。

例子1

输入: s = "eceba", k = 2
输出: 3
解释:则 T 为 “ece”，所以长度为 3。

例子2

输入： s = “aa”, k = 1

输出: 2
解释:则 T 为 “aa”，所以长度为 2。

思考 1

1 因为这里是双指针的专题，很自然就想到了双指针，那么双指针的核心是什么呢?

可以思考一下

核心就是找到如何更新两个指针，这里很容易就想到一个low指针指向包含k个不同字符的字符串s1的低位，一个high指向s1高位

剩下的就是考虑如何不断的更新两个指针，什么时候更新low指针，什么时候更新high指针

比较容易想到在这种时候，如果high指针向前移动，直到找到和不能再移动，不能再移动就是指发现了和high移动第一步后不相同的字符的时候，假设此时有a个相同字符，a >= 1, 此时如何更新low呢?

此时最好的情况是low指向的字符在s1中只出现一次，此时可以直接low++（此时low指向的字符肯定不等于high指向的字符，如果此时相等，那么说明我们找到的s1是错误的）

那么剩下的就是当low指向的字符在s1中出现了不止一次，出现了多次，那么该如何更新low呢？或者low更新不了，是不是能更新high呢?

这应该是题目最难的地方，所以这里可以停下里思考一下?

这时候low可以一直删除字符，直到字符在s1中只出现一次，可以停止了，此时有发现了一个新的字符串s2，比较s2和s1的长度就可以了

2 其实这里还可以使用dp，因为很容易就可以看出dp转移方程
假设dp[i] 表示含有k个字符的最长字符串
那么dp[i+1] 很容易想到dp[i+1] = Math.max(dp[i], 以i+1为结尾的最长k个字符串)

实现1

```
/**
 * @param {string} s
 * @param {number} k
 * @return {number}
 */
export default (s, k) => {
  // 处理边界情况
  if (s.length === 0 || k === 0) return 0;
  // 不同字符的个数
  let distinct = 0;
  // 记录每个字符出现的次数
  const map = new Map();
  let low = 0;
  let res = 0;

  for (let high = 0; high < s.length; high++) {
    const highChar = s.charAt(high);
    if (!map.has(highChar)) {
      map.set(highChar, 1);
      distinct++;
```

```
    } else {
      let count = map.get(highChar) + 1;
      map.set(highChar, count);
    }
    // 如果不同的字符数小于k，继续增加high，直到找到包含大于k个字符
    if (distinct <= k) {
      res = Math.max(res, high - low + 1);
    } else {
      // 大于k个字符的时候，这时候就要更新low，不断删除字符，直到小于k个不同字符
      while (distinct > k) {
        const lowChar = s.charAt(low);
        let lowCount = map.get(lowChar);
        if (lowCount > 1) {
          map.set(lowChar, lowCount - 1);
        } else {
          map.delete(lowChar);
          distinct--;
        }
        low++;
      }
    }
  }
  return res;
```

```
};
```

这里的时间复杂度可以很容易的看出是O(n),虽然里边有while循环，但是仍然是O（n）

空间复杂度O(n)

双指针算法总结

一般是涉及到范围的都可以尝试下双指针算法，双指针算法的关键是要找到如何更新两个指针，让我们更加的接近算法的解。换句话说就是要想想如何缩小我们要查找的范围，让我们更加的接近答案。

第4章排序算法

简单的排序算法

1 排序是最常见的算法，基本上每本书开头都是先介绍常用的各种排序算法。

排序算法总体来说，只是种类繁多，但是基本上都不是很难，都是些基础，这些多写，打好基础就可以了。

215 快速选择

题目描述

在一个未排序的数组中，找到第 k 大的数字。

注意，这里你可以默认肯定有解

例子1

输入: [3,2,1,5,6,4] and k = 2
输出: 5

例子2

输入: [3,2,3,1,2,4,5,5,6] and k = 4
输出: 4

思考 1

基本上先排序，然后直接输出就可以了

这里也可以自己实现，比如使用快速排序，一般快排是查找第k大数的常用方法

实现1

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */

export default (nums, k) => {
  nums.sort((a, b) => b - a);
  return nums[k - 1];
};
```

实现2

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */

const getPosition = (nums, p, r) => {
  const mid = Math.floor(p + (r - p) / 2);
  let temp;
  let pos;
  let tempArr = [nums[p], nums[r], nums[mid]].sort((a, b) => a - b);
  temp = tempArr[1];
  pos = temp === nums[p] ? p : temp === nums[r] ? r : mid;
  for (let i = p; i <= r; i++) {
    if (nums[i] < temp && i > pos) {
      const _temp = nums[i];
      if (i > pos) {
        for (let j = i; j > pos; j--) {
          nums[j] = nums[j - 1];
        }
      }
      nums[pos] = _temp;
```

```
      pos++;
    } else if (nums[i] > temp && i < pos) {
      nums[pos] = nums[i];
      nums[i] = temp;
      pos = i;
    }
  }
  nums[pos] = temp;

  return pos;
};
// p 起始位置， r 结束位置
// 3, 2, 3, 1, 2, 4, 5, 5, 6
const quickSort = (nums, p, r) => {
  if (p < r) {
    const q = getPosition(nums, p, r);
    quickSort(nums, p, q - 1);
    quickSort(nums, q + 1, r);
  }
};
export default (nums, k) => {
  // nums.sort((a, b) => a - b);
  quickSort(nums, 0, nums.length - 1);
```

```
    return nums[nums.length - k];
};
```

实现1的算法时间复杂度 O(nlgn), 空间复杂度 O(1)

451. 根据字符出现频率排序

题目描述

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

例子1

输入: "tree"
输出: "eert"
解释: 'e'出现两次，'r'和't'都只出现一次。
因此'e'必须出现在'r'和't'之前。此外，"eetr"也是一个有效的答案

例子2

输入: "cccaaa"
输出: "cccaaa"
解释: 'c'和'a'都出现三次。此外，"aaaccc"也是有效的答案。注意"cacaca"是不正确的，因为相同的字母必须放在一起。

例子3

输入: "Aabb"
输出: "bbAa"
解释: 此外，"bbaA"也是一个有效的答案，但"Aabb"是不正确的。
注意'A'和'a'被认为是两种不同的字符。

思考 1

题目也比较简单，只要思想类似于桶排序，我们先用一个hashMap存储每个字符出现的次数，然后对hashMap进行排序，最后再取出来就可以了。

实现1

```
/**
 * @param {string} s
 * @return {string}
 */

export default (s) => {
```

```
  const map = new Map();
  for (let i = 0; i < s.length; i++) {
    if (!map.has(s[i])) {
      map.set(s[i], 1);
    } else {
      const count = map.get(s[i]) + 1;
      map.set(s[i], count);
    }
  }
  let res = [];
  for (let [key, value] of map) {
    res.push({
      key,
      value,
    });
  }
  res.sort((a, b) => b.value - a.value);
  let res1 = "";
  for (let i = 0; i < res.length; i++) {
    res1 += res[i].key.repeat(res[i].value);
  }
  return res1;
};
```

实现1的算法时间复杂度 O(nlgn), 空间复杂度 O(1)
这里空间复杂度是O（1）的原因是最多只有26个字母，所以不会一直扩张

347. 前 K 个高频元素

题目描述

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

例子1

输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]

例子2

输入: nums = [1], k = 1
输出: [1]

提示：
1 你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
2 你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。

3 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
4 你可以按任意顺序返回答案。

思考 1

桶排序顾名思义就是我们事先放几个桶，这几个桶是排好序的，然后我们可以把符合每个桶的元素放到每个桶里边，这样就天然变成有序的了

这里我们可以根据数组中每个不同的数字分别设置一个桶，每个桶里放入数字出现的次序，最后返回钱k个高频元素就可以了

实现1

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */

export default (nums, k) => {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    if (!map.has(nums[i])) {
      map.set(nums[i], 1);
```

```
    } else {
      const count = +map.get(nums[i]) + 1;
      map.set(nums[i], count);
    }
  }
  let res = [];
  for (let [key, value] of map) {
    res.push({
      key,
      value,
    });
  }
  res.sort((a, b) => b.value - a.value);
  res = res.slice(0, k);
  return res.map((item) => +item.key);
};
```

实现1的算法时间复杂度 O(nlgn), 空间复杂度 O(1)

75. 颜色分类

题目描述

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

进阶:

1 你可以不使用代码库中的排序函数来解决这道题吗？
2 你能想出一个仅使用常数空间的一趟扫描算法吗？

例子1

输入: nums = [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]

例子2

输入: nums = [2,0,1]
输出: [0,1,2]

例子3

输入: nums = [0]
输出: [0]

例子4

输入: nums = [1]
输出: [1]

提示：
1 n == nums.length
2 1 <= n <= 300
3 nums[i] 为 0、1 或 2

思考 1

这里使用桶排序，直接统计0，1，2出现的次数，直接修改原数组就可以了

当然这里还有其它方法，比如使用双指针，因为前面使用过双指针，我们可以实现下

双指针也很简单，一个low指向数组的开始，一个high指向数组的末尾，使用i进行遍历数组，如果发现nums[i]等于0，则和low交换，并且low++，当发现等于2，则和high交换并且high--

实现1

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
```

```
 */
// [2, 0, 2, 1, 1, 0][(2, 0, 2, 1, 1, 0)];
export default (nums) => {
  const res = new Array(3).fill(0);
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] === 0) {
      res[0]++;
    } else if (nums[i] === 1) {
      res[1]++;
    } else {
      res[2]++;
    }
  }
  // console.log(res);
  for (let i = 0; i < res[0]; i++) {
    nums[i] = 0;
  }
  for (let i = 0; i < res[1]; i++) {
    nums[res[0] + i] = 1;
  }
  for (let i = 0; i < res[2]; i++) {
    nums[res[0] + i + res[1]] = 2;
  }
```

```
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
 */
// [2, 0, 2, 1, 1, 0][(2, 0, 2, 1, 1, 0)];
const swap = (nums, i, j) => {
  const temp = nums[i];
  nums[i] = nums[j];
  nums[j] = temp;
};

export default (nums) => {
  let low = 0;
  while (nums[low] === 0) {
```

```
  low++;
}
let high = nums.length - 1;
while (nums[high] === 2) {
  high--;
}
for (let i = low; i <= high; ) {
  if (low > high) {
    break;
  }
  if (nums[i] === 0 || nums[i] === 2) {
    if (nums[i] === 0) {
      if (low !== i) {
        swap(nums, low, i);
      }
      low++;
      i = low;
    } else if (nums[i] === 2) {
      if (i !== high) {
        swap(nums, i, high);
      }
      high--;
    }
```

```
      continue;
    } else {
      i++;
    }
  }
  return nums;
};
```

实现1的算法时间复杂度 O(n), 空间复杂度 O(1)

实现2的算法时间复杂度 O(n), 空间复杂度 O(1)

第5章无处不在的二分查找算法

无处不在的二分查找?

1 二分查找在实际中应用的很多，但是思想确实很简单，就是类似于分治的思想，比如你想从1000甚至更多的数字中寻找特定的数，如果你挨个去查找，当然可以，但是如果可以每次查找就可以确定想要查找的数不在另外一半中，是不是要快很多。

二分查找就是这么简单，只要记住，找到方法可以把范围缩小一半，就可以使用二分查找。

69 求开方

题目描述

给定一个非负整数，求它的开方，向下取整。

例子1

输入: 8
输出: 2
解释: 8 的开方结果是 2.82842...，向下取整即是 2。

例子2

输入: 4
输出: 2
解释: 4 的开方结果是 2。

说明
0 <= x <= 2^31 - 1

思考 1

这个思路很简单，因为是求n的开方，可以查找1到n/2的数字，
这里就可以使用二分查找，如果发现等于n，则返回，如果大于n，则可以在比较小的一半中查找，如果发现小于n，则可以在比较大的一半中查找

当然这个题目还有其他的数学解法，这里我们只是了解二分查找的思想，其他的数学解法，可以自己去了解。

实现1

```javascript
/**
 * @param {number} x
 * @return {number}
 */
export default (x) => {
  const halfX = Math.ceil(x / 2);
  let low = 1;
  let high = halfX;
  while (low <= high) {
    let mid = Math.floor(low + (high - low) / 2);
    if (mid * mid === x) {
      return mid;
    } else if (mid * mid < x) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return low * low > x ? low - 1 : low + 1;
};
```

算法时间复杂度 O(lgn/2), 空间复杂度 O(1)

34 查找区间

题目描述

给定一个增序的整数数组和一个值，查找该值第一次和最后一次出现的位置。如果不存在该值，则两个返回值都设为-1

进阶
使用O（lgn）时间复杂度解决。

例子1

输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
解释: 数字 8 在第 3 位第一次出现，在第 4 位最后一次出现。

例子2

输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
解释: 6 在数组中没有出现

例子3

输入: nums = [3,3,3], target = 3

输出: [0,2]
解释: 3 在数组中出现第一次位置是0，最后一次的位置2

说明
1 0 <= nums.length <= 10^5
2 -10^9 <= nums[i] <= 10^9
3 -10^9 <= target <= 10^9

思考 1

这个思路很简单，因为数组是升序的，而且指明使用O（lgn）方法解决，很明显使用二分查找解决。

这个也比较简单，就是常用的二分查找，如果最后没有发现，返回[-1,-1]就可以了

这里需要注意的就是你要找到target在数组中第一出现的位置和target最后一次出现的位置。

实现1

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
export default (nums, target) => {
```

```
if (nums.length === 0 || (nums.length === 1 && nums[0] !== target)) {
  return [-1, -1];
}
if (nums.length === 1 && nums[0] === target) {
  return [0, 0];
}

const len = nums.length;
let low = 0;
let high = len - 1;

const res = [];
while (low <= high) {
  // 为了防止数据溢出
  let mid = Math.floor(low + (high - low) / 2);

  if (nums[mid] === target) {
    let minFlag = false;
    let maxTemp = mid;
    let minTemp = mid;
    while (minTemp >= 0 && nums[minTemp] === target) {
      minTemp--;
    }
```

```
  if (minTemp + 1 !== mid) {
    res.push(minTemp + 1);
  } else {
    res.push(mid);
  }
  while (maxTemp < len && nums[maxTemp] === target) {
    maxTemp++;
  }

  if (maxTemp === mid + 1) {
    res.push(mid);
  } else {
    res.push(maxTemp - 1);
  }
  return res;
}
if (nums[mid] < target) {
  low = mid + 1;
}
if (nums[mid] > target) {
  high = mid - 1;
}
```

```
  }
  if (res.length === 2) {
    return res.sort((a, b) => a - b);
  } else {
    return [-1, -1];
  }
};
```

算法时间复杂度 O(lgn), 空间复杂度 O(1)

81 旋转数组查找数字

题目描述

一个原本增序的数组被首尾相连后按某个位置断开(如 [1,2,2,3,4,5] → [2,3,4,5,1,2]，在第一 位和第二位断开)，我们称其为旋转数组。给定一个值，判断这个值是否存在于这个旋转数组中。如果存在返回true，如果不存在返回false

例子1

输入: nums = [2,5,6,0,0,1,2], target = 0
输出: true

例子2

输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false

思考 1

最简单的肯定直接遍历数组，不过这里显然不使用这种

可以看到数组是一部分升序，另外一部分也是升序，问题是如何找到在哪里分割开?

当然这里可以遍历数组，找到分割开的两个升序数组，但是这样那不如直接遍历，不用二分查找了。

那么是否可以换个角度，假设我们如果直接使用二分查找，会怎么样呢?

可以想一下

刚开始我是想根据mid和mid+1的关系来决定移动low和high或者根据mid和mid-1的关系来决定移动low和high，可是这样感觉逻辑很复杂

后来看了题解，才明白可以把mid和low和high的关系来移动指针，如果nums[mid] 小于nums[high],那肯定是升序，因为问的数组是升序的，如果target在这个升序数组中，那么就可以排除另一半了

当nums[mid] 大于nums[low]的时候，说明这也是一个派系数组，如果同时target在这个排序数组呢，同时也能排除另外一半数组了

这里还有另外一种情况，因为数组中存在重复的数字，如果发现nums[mid]等于num[low]，此时我们可以把low++，重新计算mid，计算target存在的范围

当nums[mid]等于num[high]，此时我们可以把high，重新计算mid，计算target存在的范围

但是运行之后，可以发现这里的二分查找和遍历数组使用时间基本差不多。

实现1

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {boolean}
 */
export default (nums, target) => {
  if (nums.length === 0 || !nums) {
    return false;
  }
  if (nums.length === 1) return nums[0] === target;

  let low = 0;
  let high = nums.length - 1;
  while (low <= high) {
    let mid = Math.floor(low + (high - low) / 2);
    if (nums[mid] === target) {
      return true;
    }
```

```
    if (nums[mid] < nums[high]) {
      // 判断target是否在这个范围内
      if (target > nums[mid] && target <= nums[high]) {
        low = mid + 1;
      } else {
        high = mid - 1;
      }
    } else if (nums[mid] > nums[low]) {
      if (target >= nums[low] && target < nums[mid]) {
        high = mid - 1;
      } else {
        low = mid + 1;
      }
    } else if (nums[low] === nums[mid]) {
      low++;
    } else {
      high--;
    }
  }
  return false;
};
```

算法时间复杂度 O(n),因为最坏的情况下，二分查找就变成了遍历数组了。 空间复杂度 O(1)

154 旋转数组查中寻找最小的元素

题目描述

假设一个排好序的数组在某处执行了一次“旋转”，找出最小的元素（数组元素可能重复），数组中包含重复数字

例子1

输入: [1,3,5]
输出: 1

例子2

输入: [2,2,2,0,1]
输出: 0

例子3

输入: [3,3,1,3]
输出: 1

例子4

输入: [10,1,10,10,10]
输出: 1

思考 1

这里和上面的81题目有点类似，应该可以采用同样的策略，只不过是把81的题目的target变成了最小的数字,思路基本类似。

另外说一下，这题在leetcode上标记为hard，但是81题标记为medium，但是两者的难度差不多，所以有时候，没有必要对题目的难度过于太在意

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// 2, 2, 2, 0, 1;
export default (nums) => {
  if (nums.length === 0 || !nums) {
    return false;
  }
  if (nums.length === 1) return nums[0];
  let low = 0;
```

```
  let high = nums.length - 1;
  let min = Number.MAX_VALUE;
  while (low <= high) {
    let mid = Math.floor(low + (high - low) / 2);
    if (nums[mid] < nums[high]) {
      high = mid - 1;
      min = Math.min(nums[mid], min);
    } else if (nums[mid] > nums[low]) {
      min = Math.min(nums[low], min);
      low = mid + 1;
    } else if (nums[low] === nums[mid]) {
      min = Math.min(nums[low], min);
      low++;
    } else {
      min = Math.min(nums[high], min);
      high--;
    }
  }
  return min;
};
```

算法时间复杂度 O(n),因为最坏的情况下，二分查找就变成了遍历数组了。 空间复杂度 O(1)

540. 有序数组中的单一元素

题目描述

给定一个只包含整数的有序数组，每个元素都会出现两次，唯有一个数只会出现一次，找出这个数。

例子1

输入: [1,1,2,3,3,4,4,8,8]
输出: 2

例子2

输入: [3,3,7,7,10,11,11]
输出: 10

注意: 您的方案应该在 O(log n)时间复杂度和 O(1)空间复杂度中运行。

思考 1

很简单，用二分查找就可以,要点就是因为数组中除了一个数字，其它的数字都是出现两次，所以可以根据mid的下标是奇数或者偶数来判断数字是否出现在那一侧

如果mid是偶数，那说明low到mid有奇数个数字，所以就可以判断nums[mid] 和nums[mid-1]是否相等，来判断只出现一次的数字是否出现在这一侧。

思路比较简单

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */

export default (nums) => {
  if (nums.length === 1) return nums[0];
  let low = 0;
  let high = nums.length - 1;
  while (low <= high) {
    let mid = Math.floor(low + (high - low) / 2);
    console.log(low, high, mid);
    if (
      (mid + 1 < nums.length && nums[mid] !== nums[mid + 1] && mid >= 1 && nums[mid] !== nums[mid - 1]) ||
      (mid === nums.length - 1 && nums[mid] !== nums[mid - 1]) ||
      (mid === 0 && nums[mid] !== nums[mid + 1])
```

```
    ) {
      return nums[mid];
    }
    if (mid % 2 !== 0) {
      if (mid >= 1 && nums[mid - 1] === nums[mid]) {
        low = mid + 1;
      } else {
        high = mid - 1;
      }
    } else {
      if (mid >= 1 && nums[mid] === nums[mid - 1]) {
        high = mid - 1;
      } else {
        low = mid + 1;
      }
    }
  }
  return nums[low];
};

};
```

算法时间复杂度 O(lgn)。 空间复杂度 O(1)

4. 寻找两个正序数组的中位数

题目描述

给定两个大小为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的中位数。

进阶：你能设计一个时间复杂度为 O(log (m+n)) 的算法解决此问题吗?

例子1

输入: nums1 = [1,3], nums2 = [2]
输出: 2.00000
解释：合并数组 = [1,2,3] ， 中位数 2

例子2

输入: nums1 = [1,2], nums2 = [3,4]
输出: 2.50000
解释：合并数组 = [1,2,3,4] ， 中位数 (2 + 3) / 2 = 2.5

例子3

输入: nums1 = [0,0], nums2 = [0,0]
输出: 0.00000

例子4

输入: nums1 = [], nums2 = [1]
输出: 1.00000

例子5

输入: nums1 = [2], nums2 = []
输出: 2.00000

提示：
1 nums1.length == m
2 nums2.length == n
3 0 <= m <= 1000
4 0 <= n <= 1000
5 1 <= m + n <= 2000
6 -10^6 <= nums1[i], nums2[i] <= 10^6

思考 1

题目如果第一次见到，很难想出如何使用二分查找的，不过也可以思考试试

首先判断nums1和nums2的数组长度，让nums1的数组长度小于等于nums2的数组长度

假设k是nums1和nums2合并之后，我们要寻找的中位数下标，这里如果nums1和nums2合并后的长度是奇数，我们可以寻找k = left的数字

```
const left = Math.floor((nums1Len + nums2Len + 1) / 2)
```

如果nums1和nums2合并后的长度是偶数，我们可以分别寻找合并后的数组中下标分别是下面这两个位置的，也就是寻找k = left和k=right两个位置的数字

```
const left = Math.floor((nums1Len + nums2Len + 1) / 2);
  const right = Math.floor((nums1Len + nums2Len + 2) / 2);
```

原理很简单，我们先从nums1中拿出k/2个数字和从nums2中拿出k/2个数字，如果nums1[k/2] 大于nums2[k/2],那么我们要寻找的k位置的数字，肯定在nums1和nums2出去0到k/2的数组中。

因为nums1[k/2]大于nums2[k/2]，所以就可以排除nums[k/2]之前的数字，但是我们不知道nums1的数字是否都大于nums2[k/2]，所以可以在剩下的数组中寻找排在k/2位置的数字。

实现1

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
```

```
 * @return {number}
 */
const getKth = (nums1, start1, end1, nums2, start2, end2, k) => {
  const len1 = end1 - start1 + 1;
  const len2 = end2 - start2 + 1;
  // 如果nums1的长度大于nums2的长度，改变两个数组的位置，让数组长度最小的在前面，防止
  // 这里的nums2[start2 + k - 1]报错
  if (len1 > len2) return getKth(nums2, start2, end2, nums1, start1, end1, k);
  if (len1 === 0) return nums2[start2 + k - 1];
  if (k === 1) return Math.min(nums1[start1], nums2[start2]);

  const i = start1 + Math.min(len1, Math.floor(k / 2)) - 1;
  const j = start2 + Math.min(len2, Math.floor(k / 2)) - 1;

  const nums1End = Math.min(end1, i + 1);
  const nums2End = Math.min(end2, j + 1);
  if (nums1[i] > nums2[j]) {
    return getKth(nums1, start1, nums1End, nums2, j + 1, end2, k - (j - start2 + 1));
  } else {
    return getKth(nums1, i + 1, end1, nums2, start2, nums2End, k - (i - start1 + 1));
  }
};
export default (nums1, nums2) => {
```

```
  if (nums1.length === 0 && nums2.length === 1) {
    return nums2[0];
  }
  if (nums1.length === 1 && nums2.length === 0) {
    return nums1[0];
  }
  const nums1Len = nums1.length;
  const nums2Len = nums2.length;
  // 分别找到奇数和偶数的中位数的左边和右边
  const left = Math.floor((nums1Len + nums2Len + 1) / 2);
  const right = Math.floor((nums1Len + nums2Len + 2) / 2);
  // 如果是奇数，只寻找一次就可以了
  if ((nums1Len + nums2Len) % 2 !== 0) {
    return getKth(nums1, 0, nums1Len - 1, nums2, 0, nums2Len - 1, left);
  }
  return (
    (getKth(nums1, 0, nums1Len - 1, nums2, 0, nums2Len - 1, left) +
      getKth(nums1, 0, nums1Len - 1, nums2, 0, nums2Len - 1, right)) *
    0.5
  );
};
```

算法时间复杂度也很容易理解
m 是nums1.length，n 是nums2.length
因为每次查找的范围都从(m+n)/2 缩小一半范围。所以时间复杂度是O(lg(m+n))
空间复杂度 O(lg(m+n))

二分查找算法总结

二分查找其实就是一个分治思想，如果你可以根据一些条件，每次缩小一半的查找范围，基本就可以确定使用二分查找。

不过有些可能不是很明显，可能需要经过转化处理，不过核心就是可以不断的缩小查找范围，让我们离答案更近一下。

第6章搜索既要广度也要深度

深度搜索 <br data-tomark-pass>

1.1 深度搜索 <br data-tomark-pass>

深度搜索其实很简单，而且也特别通俗易懂，用俗话说就是一条道走到黑，撞到南墙之后，再往回看看，重新选择一条路走到黑。不断的重复这个过程。<br data-tomark-pass>

深度搜索是首先走一个节点，然后把该节点能都走到的位置push进入一个栈，然后从栈中拿出一个节点，接着走该节点能走到的所有位置，再全部push进入栈。如此重复<br data-tomark-pass>

130. 被围绕的区域

题目描述

给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。<br data-tomark-pass>

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。<br data-tomark-pass>

例子1<br data-tomark-pass>

Input:
X X X X

X O O X
X X O X
X O X X<br data-tomark-pass>

output:
X X X X
X X X X
X X X X
X O X X<br data-tomark-pass>

解释： 被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

思考

很简单的深度搜索就可以就可以解决，从四周搜索，如果不是和四周连接在一起转为“X”就可以了

实现1

```
/**
 * @param {character[][]} board
 * @return {void} Do not return anything, modify board in-place instead.
 */
const dirs = [
```

```
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
const gfs = (board, i, j, m, n) => {
  board[i][j] = -1;
  for (let k = 0; k < 4; k++) {
    const nextI = i + dirs[k][0];
    const nextJ = j + dirs[k][1];
    if (nextI >= 0 && nextI < m && nextJ >= 0 && nextJ < n && board[nextI][nextJ] === "O") {
      gfs(board, nextI, nextJ, m, n);
    }
  }
};
// Runtime: 96 ms, faster than 84.79% of JavaScript online submissions for Surrounded Regions.
// Memory Usage: 42.2 MB, less than 87.06% of JavaScript online submissions for Surrounded Regions.
export default (board) => {
  const m = board.length;

  if (m === 0) return board;
  const n = board[0].length;
```

```
  for (let i = 0; i < n; i++) {
    if (board[0][i] === "O") {
      gfs(board, 0, i, m, n);
    }
    if (board[m - 1][i] === "O") {
      gfs(board, m - 1, i, m, n);
    }
  }
  for (let i = 1; i < m - 1; i++) {
    if (board[i][0] === "O") {
      gfs(board, i, 0, m, n);
    }
    if (board[i][n - 1] === "O") {
      gfs(board, i, n - 1, m, n);
    }
  }
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (board[i][j] === -1) {
        board[i][j] = "O";
      } else if (board[i][j] === "O") {
        board[i][j] = "X";
      }
```

```
    }
  }
  return board;
};
```

时间复杂度O（m * n），空间复杂度O（m * n）

695. 岛屿的最大面积

题目描述

给定一个包含了一些 0 和 1 的非空二维数组 grid 。

一个 岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 grid 的四个边缘都被 0（代表水）包围着。

找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为 0 。)<br data-tomark-pass>

例子1<br data-tomark-pass>

Input:

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

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

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

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

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

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

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

[0,0,0,0,0,0,0,1,1,0,0,0,0]]<br data-tomark-pass>
对于上面这个给定矩阵应返回 6。注意答案不应该是 11 ，因为岛屿只能包含水平或垂直的四个方向的 1 。

例子2<br data-tomark-pass>
Input: <br data-tomark-pass>

[[0,0,0,0,0,0,0,0]]<br data-tomark-pass>
对于上面这个给定的矩阵, 返回 0。

注意: 给定的矩阵grid 的长度和宽度都不超过 50。

思考 1

这是典型的深度搜索遍历，按照正常的搜索就可以了。

深度搜索一般可以使用栈或者递归，这两个差不多，只要抓住深度搜索的重点就可以

重点是深度搜索首先遍历当前节点可以遍历到的所有节点，然后再从里边取出一个节点，继续遍历可以遍历到的所有节点

递归写法可以看下实现2

实现1

```
/**
 * @param {number[][]} grid
 * @return {number}
 */
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];

export default (grid) => {
  const visited = [];
  const m = grid.length;
  const n = grid[0].length;
  let maxArea = 0;
  let x = 0;
  let y = 0;
  let cur_maxArea = 0;
  for (let i = 0; i < m; i++) {
```

```
    for (let j = 0; j < n; j++) {
      if (grid[i][j] === 1) {
        cur_maxArea = 1;
        grid[i][j] = 0;
        const island = [];
        island.push([i, j]);
        while (island.length > 0) {
          const [curI, curJ] = island.pop();
          for (let k = 0; k < 4; k++) {
            x = curI + dirs[k][0];
            y = curJ + dirs[k][1];
            if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] === 1) {
              grid[x][y] = 0;
              cur_maxArea++;
              island.push([x, y]);
            }
          }
        }
        maxArea = Math.max(maxArea, cur_maxArea);
      }
    }
  }
  return maxArea;
```

```
};
```

实现2

```
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
// 主函数
export default (grid) => {
  if (grid.length === 0 || grid[0].length === 0) {
    return 0;
  }
  let max_area = 0;
  for (let i = 0; i < grid.length; ++i) {
    for (let j = 0; j < grid[0].length; ++j) {
      if (grid[i][j] == 1) {
        max_area = Math.max(max_area, dfs(grid, i, j));
      }
```

```
    }
  }
  return max_area;
};
// 辅函数
const dfs = (grid, curI, curJ) => {
  grid[curI][curJ] = 0;
  let nextI;
  let nextJ;
  let area = 1;
  for (let i = 0; i < 4; ++i) {
    nextI = curI + dirs[i][0];
    nextJ = curJ + dirs[i][1];
    if (nextI >= 0 && nextI < grid.length && nextJ >= 0 && nextJ < grid[0].length && grid[nextI][nextJ] === 1) {
      area += dfs(grid, nextI, nextJ);
    }
  }
  return area;
};
```

实现1算法时间复杂度 O(n^2), 空间复杂度 O(1)
实现2算法时间复杂度 O(n^2), 空间复杂度 O(1)

547. 朋友圈

题目描述

班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。<br data-tomark-pass>

给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。
<br data-tomark-pass>

例子1<br data-tomark-pass>

Input:

[[1,1,0],

[1,1,0],

[0,0,1]]<br data-tomark-pass>
output：2 <br data-tomark-pass>
已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。
第2个学生自己在一个朋友圈。所以返回 2 。

例子2<br data-tomark-pass>
Input:

[[1,1,0],

[1,1,1],

[0,1,1]]<br data-tomark-pass>
output：1 <br data-tomark-pass>
已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，所以学生 0 和学生 2 也是朋友，所以他们三个在一个朋友圈，返回 1 。

提示： <br data-tomark-pass>

1 1 <= N <= 200<br data-tomark-pass>
2 M[i][i] == 1<br data-tomark-pass>
3 M[i][j] == M[j][i]<br data-tomark-pass>

思考 1

这里和695差不多，只不过在搜索过程中，当搜到一个1的时候，我们就相当于发现了一个朋友圈，一个人自己和自己也是一个朋友圈，当我们发现了一个朋友圈之后，我们就深度搜索这个朋友圈，直到搜索完整个朋友圈。<br data-tomark-pass>
可以实现1<br data-tomark-pass>

这个时候可以换个思路，因为只有m个人，那肯定最多只能有m个朋友圈，我们可以挨着朋友圈去深度遍历，发现一个朋友圈，深度遍历在该朋友圈以内的。<br data-tomark-pass>

可以看下实现2<br data-tomark-pass>

实现1

```
/**
 * @param {number[][]} M
 * @return {number}
 */

const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
const dfs = (M, curI, curJ) => {
  M[curI][curJ] = 0;
  for (let i = 0; i < M[0].length; i++) {
    const nextJ = i;
    const nextI = curI;
    if (nextI >= 0 && nextI < M.length && nextJ >= 0 && nextJ < M[0].length && M[nextI][nextJ] === 1) {
      dfs(M, nextI, nextJ);
    }
  }
  for (let i = 0; i < M.length; i++) {
    const nextI = i;
```

```
    const nextJ = curI;
    if (nextI >= 0 && nextI < M.length && nextJ >= 0 && nextJ < M[0].length && M[nextI][nextJ] === 1) {
      dfs(M, nextI, nextJ);
    }
  }

  for (let i = 0; i < M.length; i++) {
    const nextI = curJ;
    const nextJ = i;
    if (nextI >= 0 && nextI < M.length && nextJ >= 0 && nextJ < M[0].length && M[nextI][nextJ] === 1) {
      dfs(M, nextI, nextJ);
    }
  }
  for (let i = 0; i < M.length; i++) {
    const nextI = i;
    const nextJ = curJ;
    if (nextI >= 0 && nextI < M.length && nextJ >= 0 && nextJ < M[0].length && M[nextI][nextJ] === 1) {
      dfs(M, nextI, nextJ);
    }
  }
};
export default (M) => {
  if (M.length === 1) return 1;
```

```
  const m = M.length;
  const n = M[0].length;
  let res = 0;
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (M[i][j] === 1) {
        dfs(M, i, j);
        res++;
      }
    }
  }
  return res;
};
```

实现2

```
/**
 * @param {number[][]} M
 * @return {number}
 */
const dfs = (M, i, circles) => {
```

```
  // 已经访问过了
  circles[i] = 1;
  // 遍历和自己有关系的朋友
  for (let j = 0; j < M.length; j++) {
    if (circles[j] === 0 && M[i][j] === 1 && j !== i) {
      dfs(M, j, circles);
    }
  }
};
// Runtime: 84 ms, faster than 86.26% of JavaScript online submissions for Friend Circles.
// Memory Usage: 40.2 MB, less than 94.94% of JavaScript online submissions for Friend Circles.
export default (M) => {
  if (M.length === 1) return 1;
  const m = M.length;
  const n = M[0].length;
  let res = 0;
  const circles = new Array(m).fill(0);
  // 最多一共有m个朋友圈
  for (let i = 0; i < m; i++) {
    if (circles[i] === 0) {
      dfs(M, i, circles);
      res++;
    }
```

```
  }
  return res;
};
```

实现1算法时间复杂度 O(n^2)，因为这里最多遍历n^2个节点, 空间复杂度 O(1)
实现2算法时间复杂度 O(n^2), 空间复杂度 O(1)

417. 太平洋大西洋水流问题

题目描述

给定一个 m x n 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。<br data-tomark-pass>

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同等高度上流动。<br data-tomark-pass>

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

<br data-tomark-pass>
提示：<br data-tomark-pass>

1 输出坐标的顺序不重要<br data-tomark-pass>
2 m 和 n 都小于150<br data-tomark-pass>

<br data-tomark-pass>

例子1<br data-tomark-pass>

Input:
给定下面的 5x5 矩阵:

太平洋 ~ ~ ~ ~ ~
~ 1 2 2 3 (5) *
~ 3 2 3 (4) (4) *
~ 2 4 (5) 3 1 *
~ (6) (7) 1 4 5 *
~ (5) 1 1 2 4 *
* * * * * 大西洋

<br data-tomark-pass>
output：[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (上图中带括号的单元). <br data-tomark-pass>

思考 1

最简单的想法肯定是遍历每个节点，然后看下每个节点是否能够分别到达大西洋和太平洋，但是这里有个问题，超时了。<br data-tomark-pass>
可以实现1<br data-tomark-pass>

这里有另外一种方法，就是逆向思维，让水往里边灌，如果一个节点，太平洋的水可以到达，大西洋的水也可以到达，那么肯定可以实现。<br data-tomark-pass>

这里其实以前做过一道类似的，也是通过水往里边灌水，但是在这里的时候，仍然没有考虑到。<br data-tomark-pass>

可以看下实现2<br data-tomark-pass>

实现1

```
/**
 * @param {number[][]} matrix
 * @return {number[][]}
 */
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
// 辅函数
const dfs = (matrix, curI, curJ, visited, hadEnd) => {
  if (curI === 0 || curJ === 0) {
    hadEnd[0] = true;
  }
  if (curI === matrix.length - 1 || curJ === matrix[0].length - 1) {
    hadEnd[1] = true;
  }
  let nextI;
```

```
  let nextJ;
  visited[curI][curJ] = 1;
  for (let i = 0; i < 4; ++i) {
    nextI = curI + dirs[i][0];
    nextJ = curJ + dirs[i][1];
    if (nextI >= 0 && nextI < matrix.length && nextJ >= 0 && nextJ < matrix[0].length) {
      if (matrix[nextI][nextJ] <= matrix[curI][curJ] && visited[nextI][nextJ] === 0) {
        dfs(matrix, nextI, nextJ, visited, hadEnd);
      }
    }
  }
  visited[curI][curJ] = 0;
};
// 主函数
export default (matrix) => {
  if (matrix.length === 0 || matrix[0].length === 0) {
    return [];
  }
  const visited = [];
  const m = matrix.length;
  const n = matrix[0].length;
  for (let i = 0; i < m; i++) {
    visited[i] = new Array(n).fill(0);
```

```
  }

  let res = [];
  for (let i = 0; i < m; ++i) {
    for (let j = 0; j < n; ++j) {
      // hadEnd[0] 表示到达Pacti，hadEnd[0] 表示到达Anti
      let hadEnd = [false, false];
      dfs(matrix, i, j, visited, hadEnd);
      // 如果都到达，则加入
      if (hadEnd[0] && hadEnd[1]) {
        res.push([i, j]);
      }
    }
  }
  return res;
};
```

实现2

```
/**
```

```
 * @param {number[][]} matrix
 * @return {number[][]}
 */
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
// 辅函数
const dfs = (matrix, curI, curJ, reach) => {
  let nextI;
  let nextJ;
  if (reach[curI][curJ]) {
    return;
  }
  reach[curI][curJ] = 1;
  for (let i = 0; i < 4; i++) {
    nextI = curI + dirs[i][0];
    nextJ = curJ + dirs[i][1];
    if (nextI >= 0 && nextI < matrix.length && nextJ >= 0 && nextJ < matrix[0].length) {
      if (matrix[nextI][nextJ] >= matrix[curI][curJ] && reach[nextI][nextJ] === 0) {
        dfs(matrix, nextI, nextJ, reach);
```

```
      }
    }
  }
};

// Runtime: 112 ms, faster than 94.23% of JavaScript online submissions for Pacific Atlantic Water Flow.
// Memory Usage: 46.9 MB, less than 64.74% of JavaScript online submissions for Pacific Atlantic Water Flow.
export default (matrix) => {
  if (matrix.length === 0 || matrix[0].length === 0) {
    return [];
  }
  const m = matrix.length;
  const n = matrix[0].length;
  const canReachPacific = [];
  const canReachAtlantic = [];
  for (let i = 0; i < m; i++) {
    canReachPacific[i] = new Array(n).fill(0);
    canReachAtlantic[i] = new Array(n).fill(0);
  }
  let res = [];
  for (let i = 0; i < m; i++) {
    dfs(matrix, i, 0, canReachPacific);
  }
```

```
  for (let i = 0; i < n; i++) {
    dfs(matrix, 0, i, canReachPacific);
  }
  for (let i = 0; i < m; i++) {
    dfs(matrix, i, n - 1, canReachAtlantic);
  }
  for (let i = 0; i < n; i++) {
    dfs(matrix, m - 1, i, canReachAtlantic);
  }
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (canReachPacific[i][j] === 1 && canReachAtlantic[i][j] === 1) {
        res.push([i, j]);
      }
    }
  }
  return res;
};
```

实现1算法时间复杂度 O (m*n), 因为这里最多遍历 m*n个节点4次, 空间复杂度 O(1)

实现2算法时间复杂度 O (m*n), 因为这里最多遍历 m*n个节点4次, 空间复杂度 O(1)

回溯法

回溯法(backtracking)是优先搜索的一种特殊情况，又称为试探法，常用于需要记录节点状态的深度优先搜索。通常来说，排列、组合、选择类问题使用回溯法比较方便。

回溯其实也很简单，其实和普通的深度搜索没什么区别，就是发现不合适可以退出

46. 全排列

题目描述

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

例子1

input：[1,2,3]
output：[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]
]

例子2

input：[0,1]
output：[[0,1],[1,0]]

例子1

input：[1]
output：[[1]]

提示：

1 1 <= nums.length <= 6
2 -10 <= nums[i] <= 10
3 All the integers of nums are unique.

思考1

简单的回溯法，

怎样输出所有的排列方式呢?对于每一个当前位置 i，我们可以将其于之后的任意位置交换， 然后继续处理位置 i+1，直到处理到最后一位。

实现1

```
const swap = (nums, i, j) => {
  const temp = nums[i];
  nums[i] = nums[j];
  nums[j] = temp;
};
// count 遍历的次数
const backtracking = (nums, count, res) => {
  if (count === nums.length - 1) {
    res.push(nums);
  }

  for (let i = count; i < nums.length; i++) {
    // 修改状态
    if (i !== count) {
      swap(nums, i, count);
    }
    backtracking([...nums], count + 1, res);
    // 恢复状态
    if (i !== count) {
      swap(nums, count, i);
```

```
    }
  }
};
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
export default (nums) => {
  if (nums.length === 1) {
    return [[nums[0]]];
  }
  let res = [];
  backtracking(nums, 0, res);
  return res;
};
```

时间复杂度0(n^2(n)!), 空间复杂度是O（n^3）

47. 全排列II

题目描述

给定一个 没有含有重复数字的序列，返回其所有可能的全排列。

例子1

input：[1,2,3]
output：[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]
]

例子2

input：[1,1，2]
output：

[[1,1,2],

[1,2,1],

[2,1,1]]

例子1

input：[1]
output：[[1]]

提示：

1 1 <= nums.length <= 8
2 -10 <= nums[i] <= 10

思考1

1 可以使用46的解法，不过结果会有重复的，可以在结果中删除掉重复的。

可以看下实现1

2 另外这里主要麻烦在会有重复的元素，怎么解决掉重复的元素才是重点？

这里举个测试用例，比如[1,2,2], 可以模拟一下，为什么会出现重复的结果？

解决办法就是如何去防止重复的结果出现

通过[1,2,2]的测试用例，可以发现重复的原因就是2和2之间重复了
所以这里可以加一个used 数组，用来标记数组中那些数被使用了
每次从数组中取出一个数来，如果发现使用过或者是重复的数字，但是前面的重复的数字还没使用过，则跳出，否则使用回溯法解决就可以了。

可以看下实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
const swap = (nums, i, j) => {
  const temp = nums[j];
  nums[j] = nums[i];
  nums[i] = temp;
};
const backTracking = (nums, level, res) => {
  if (level === nums.length - 1) {
    for (let k = 0; k < res.length; k++) {
      if (res[k].join("") === nums.join("")) {
        return;
      }
    }
    res.push(nums);
    return;
  }
  for (let i = level; i < nums.length; i++) {
```

```
    if (nums[level] === nums[i] && i !== level) {
      continue;
    }
    if (level !== i) {
      swap(nums, i, level);
    }
    backTracking([...nums], level + 1, res);
    if (i !== level) {
      swap(nums, level, i);
    }
  }
};
// Runtime: 4384 ms, faster than 5.01% of JavaScript online submissions for Permutations II.
// Memory Usage: 45.7 MB, less than 21.99% of JavaScript online submissions for Permutations II.
export default (nums) => {
  if (nums.length === 1) return [nums];
  const res = [];
  nums.sort((a, b) => a - b);
  backTracking(nums, 0, res);
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
const swap = (nums, i, j) => {
  const temp = nums[j];
  nums[j] = nums[i];
  nums[i] = temp;
};
const backTracking = (nums, level, res) => {
  if (level === nums.length - 1) {
    for (let k = 0; k < res.length; k++) {
      if (res[k].join("") === nums.join("")) {
        return;
      }
    }
    res.push(nums);
    return;
  }
  for (let i = level; i < nums.length; i++) {
```

```
    if (nums[level] === nums[i] && i !== level) {
      continue;
    }
    if (level !== i) {
      swap(nums, i, level);
    }
    backTracking([...nums], level + 1, res);
    if (i !== level) {
      swap(nums, level, i);
    }
  }
};
// Runtime: 4384 ms, faster than 5.01% of JavaScript online submissions for Permutations II.
// Memory Usage: 45.7 MB, less than 21.99% of JavaScript online submissions for Permutations II.
export default (nums) => {
  if (nums.length === 1) return [nums];
  const res = [];
  nums.sort((a, b) => a - b);
  backTracking(nums, 0, res);
  return res;
};
```

77. 组合

题目描述

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

例子1

input：n = 4, k = 2
output：[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],
]

思考1

简单的回溯法，

回溯法掌握三个点就可以了

1 修改状态
2 进入递归
3 恢复状态

实现1

```
/**
 * @param {number} n
 * @param {number} k
 * @return {number[][]}
 */
const backTrack = (nums, index, k, loopRes, res) => {
  if (loopRes.length === k) {
    res.push(loopRes);
    return;
  }
  for (let i = index; i < nums.length; i++) {
    loopRes.push(nums[i]);
    backTrack(nums, i + 1, k, [...loopRes], res);
    loopRes.pop(nums[i]);
  }
```

```
};
export default (n, k) => {
  const res = [];
  const loopRes = [];
  const nums = [];
  for (let i = 1; i <= n; i++) {
    nums.push(i);
  }
  backTrack(nums, 0, k, [], res);
  return res;
};
```

40. 组合总和 II

题目描述

给定一个数组 candidates 和一个目标数 target ， 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

例子1

input：candidates = [10,1,2,7,6,1,5], target = 8,
output：[

[1, 7],

[1, 2, 5],

[2, 6],

[1, 1, 6]
]

例子2

input：candidates = [2,5,2,1,2], target = 5,
output：[

[1,2,2],

[5]
]

思考1

1 简单的回溯法，

这里主要是考虑如何处理重复元素，不导致重复的结果，当然可以采用前面的，对所有的结果进行筛选，排除重复的，或者在遍历的时候，把重复的过滤掉，也就是再遍历的时候，跳过重复的数字。
可以看下实现1，

2 另外这里应该也可以采用前面47采用的使用一个used数组标记是否遍历过，来解决重复数字的问题。

参考实现2

实现1

```
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */

// Runtime: 100 ms, faster than 38.63% of JavaScript online submissions for Combination Sum II.
// Memory Usage: 44.6 MB, less than 22.16% of JavaScript online submissions for Combination Sum II.
const dfs = (candidates, level, target, singleRes, res) => {
  let sum = 0;
  if (singleRes.length > 0) {
    sum = singleRes.reduce((a, b) => a + b);
  }
  if (sum === target) {
    res.push(singleRes);
    return;
  } else if (sum > target) {
```

```
    return;
  }
  for (let i = level; i < candidates.length; ) {
    singleRes.push(candidates[i]);
    dfs(candidates, i + 1, target, [...singleRes], res);
    singleRes.pop(candidates[i]);
    if (candidates[i + 1] === candidates[i]) {
      while (candidates[i + 1] === candidates[i]) {
        i++;
      }
      i++;
    } else {
      i++;
    }
  }
};
export default (candidates, target) => {
  candidates.sort((a, b) => a - b);
  const res = [];
  dfs(candidates, 0, target, [], res);
  return res;
};
```

实现2

```
/**
 * @param {number[]} candidates
 * @return {number[][]}
 */
const backTracking = (candidates, used, list, level, target, res) => {
  let sum = 0;
  if (list.length > 0) {
    sum = list.reduce((a, b) => a + b);
  }
  if (sum === target) {
    res.push([...list]);
    return;
  } else if (sum > target) {
    return;
  }
  for (let i = level; i < candidates.length; i++) {
    if (used[i]) {
      continue;
    }
    // 去除掉重复，防止重复
```

```
    if (i > 0 && candidates[i - 1] === candidates[i] && !used[i - 1]) {
      continue;
    }
    used[i] = 1;
    list.push(candidates[i]);
    backTracking(candidates, used, list, i + 1, target, res);
    used[i] = 0;
    list.pop();
  }
};

// Runtime: 92 ms, faster than 68.43% of JavaScript online submissions for Combination Sum II.
// Memory Usage: 40.5 MB, less than 54.71% of JavaScript online submissions for Combination Sum II.
export default (candidates, target) => {
  if (!candidates || candidates.length === 0) return [];
  const res = [];
  // 是否被使用过
  const used = new Array(candidates.length).fill(0);
  candidates.sort((a, b) => a - b);
  backTracking(candidates, used, [], 0, target, res);
  // console.log(candidates);
  return res;
};
```

79 单词搜索

题目描述

给定一个字母矩阵，所有的字母都与上下左右四个方向上的字母相连。给定一个字符串，求 字符串能不能在字母矩阵中寻找到。

例子1

input：word = "ABCCED",
board =

[

["A","B","C","E"],

["S","F","C","S"],

["A","D","E","E"]
]
output：true

例子2

input：word = "SEE",
board =

[

["A","B","C","E],

["S","F","C","S"],

["A","D","E","E"]
]
output：true

例子3

input：word = "ABCB",
board =

[

["A","B","C","E"],

["S","F","C","S"],

["A","D","E","E"]
]
output：false

思考1

题目本身比较简单，就是简单的通过回溯递归来解决

实现1

```
/**
* @param {character[][]} board
* @param {string} word
* @return {boolean}
*/
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
const dfs = (board, m, n, i, j, res, word, visted) => {
  if (res.length === word.length) {
    if (res.join("") === word) {
      return true;
    } else {
      return false;
    }
  }
  for (let k = 0; k < 4; k++) {
```

```
    const nextI = i + dirs[k][0];
    const nextJ = j + dirs[k][1];
    if (
      nextI >= 0 &&
      nextI < m &&
      nextJ >= 0 &&
      nextJ < n &&
      word.charAt(res.length) === board[nextI][nextJ] &&
      visted[nextI][nextJ] === 0
    ) {
      res.push(board[nextI][nextJ]);
      visted[nextI][nextJ] = 1;
      if (dfs(board, m, n, nextI, nextJ, [...res], word, visted)) {
        return true;
      }
      res.pop();
      visted[nextI][nextJ] = 0;
    }
  }
  return false;
};
export default (board, word) => {
  const m = board.length;
```

```
  const n = board[0].length;
  const res = [];
  const visted = [];
  for (let i = 0; i < m; i++) {
    visted[i] = new Array(n).fill(0);
  }
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (board[i][j] === word.charAt(0)) {
        res.push(board[i][j]);
        visted[i][j] = 1;
        if (dfs(board, m, n, i, j, [...res], word, visted)) {
          return true;
        }
        res.pop();
        visted[i][j] = 0;
      }
    }
  }
  return false;
};
```

257. 二叉树的所有路径

题目描述

给定一个二叉树，返回所有从根节点到叶子节点的路径。

例子1

input：
1
/
2 3

5

output：输出: ["1->2->5", "1->3"]

提示：

叶子节点是指没有子节点的节点。

思考1

直接深度遍历就可以了

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {string[]}
 */

//     1
//   /   \
//  2     3
//   \
//    5
//  Output: ["1->2->5", "1->3"]
const dfs = (root, tempRes, res) => {
```

```
  tempRes.push(root.val);
  if (!root.left && !root.right) {
    res.push(tempRes.join("->"));
    return;
  }
  if (root.left) {
    dfs(root.left, [...tempRes], res);
  }
  if (root.right) {
    dfs(root.right, [...tempRes], res);
  }
};
// Runtime: 84 ms, faster than 72.61% of JavaScript online submissions for Binary Tree Paths.
// Memory Usage: 40.5 MB, less than 30.10% of JavaScript online submissions for Binary Tree Paths.
export default (root) => {
  if (!root) return [];
  const res = [];
  dfs(root, [], res);
  return res;
};
```

时间复杂度O（n），空间复杂度O（n^2）

51 N皇后问题

题目描述

给定一个大小为 n 的正方形国际象棋棋盘，求有多少种方式可以放置 n 个皇后并使得她们互 不攻击，即每一行、列、左斜、右斜最多只有一个皇后。

例子1

input：4

output：[

[".Q..", // Solution 1
"...Q",
"Q...",
"..Q."],

["..Q.", // Solution 2
"Q...",
"...Q",
".Q.."]
]

例子2

input：1
output：[["Q"]]

提示：

1 <= n <= 9

思考1

题目虽然在leetcode上显示是困难，但是思想其实很简单，就是通过简单的回溯来解决

这里需要注意的是传递数组得注意传递复制的数组

不能重复遍历，比如在在i，j这个位置遍历到下一个位置了，肯定不能再重新遍历回到i，j这个位置了

思路比较简单，但是代码可以写的很复杂，也可以写的很简单

实现1

实现2

实现1

```
/**
* @param {number} n
* @return {string[][]}
*/
```

```
const copy = (dfsRes) => {
  const res = [];
  for (let i = 0; i < dfsRes.length; i++) {
    for (let j = 0; j < dfsRes[0].length; j++) {
      res[j] = new Array(dfsRes[0].length).fill(0);
    }
  }
  for (let i = 0; i < dfsRes.length; i++) {
    for (let j = 0; j < dfsRes[0].length; j++) {
      res[i][j] = dfsRes[i][j];
    }
  }
  return res;
};
const dfs = (dfsRes, level, res) => {
  if (level === dfsRes.length) {
    let temp = [];
    dfsRes.forEach((item) => {
      temp.push(item.join(" "));
    });
    console.log(temp);
    res.push(temp);
    return;
```

```
  }
  for (let i = 0; i < dfsRes.length; i++) {
    if (canSet(dfsRes, level, i)) {
      dfsRes[level][i] = "Q";
      const temp = copy(dfsRes);
      dfs(temp, level + 1, res);
      dfsRes[level][i] = ".";
    }
  }
};

const canSet = (dfsRes, curI, curJ) => {
  for (let i = 0; i < dfsRes.length; i++) {
    for (let j = 0; j < dfsRes[0].length; j++) {
      if (dfsRes[i][j] === "Q") {
        if (i === curI || j === curJ) {
          return false;
        }
        const n = dfsRes.length;
        // 设置i,j对角线不可放置
        let nextI = i;
        let nextJ = j;
        while (nextI >= 1 && nextJ >= 1 && nextI < n + 1 && nextJ < n + 1) {
```

```
    nextI = nextI - 1;
    nextJ = nextJ - 1;
    if (nextI === curI && nextJ === curJ) {
      return false;
    }
  }
  let nextI1 = i;
  let nextJ1 = j;
  while (nextI1 >= -1 && nextJ1 >= -1 && nextI1 < n - 1 && nextJ1 < n - 1) {
    nextI1 = nextI1 + 1;
    nextJ1 = nextJ1 + 1;
    if (nextI1 === curI && nextJ1 === curJ) {
      return false;
    }
  }

  let nextI2 = i;
  let nextJ2 = j;
  while (nextI2 >= -1 && nextJ2 >= 1 && nextI2 < n - 1 && nextJ2 < n + 1) {
    nextI2 = nextI2 + 1;
    nextJ2 = nextJ2 - 1;
    if (nextI2 === curI && nextJ2 === curJ) {
      return false;
```

```
          }
        }

        let nextI3 = i;
        let nextJ3 = j;
        while (nextI3 >= 1 && nextJ3 >= -1 && nextI3 < n + 1 && nextJ3 < n - 1) {
          nextI3 = nextI3 - 1;
          nextJ3 = nextJ3 + 1;
          if (nextI3 === curI && nextJ3 === curJ) {
            return false;
          }
        }
      }
    }
  }
  return true;
};
const creatRes = (n) => {
  const dfsRes = [];
  for (let j = 0; j < n; j++) {
    dfsRes[j] = new Array(n).fill(".");
  }
  return dfsRes;
```

```
};
export default (n) => {
  let res = [];
  const visited = [];
  for (let i = 0; i < n; i++) {
    visited[i] = new Array(n).fill(0);
  }
  for (let i = 0; i < n; i++) {
    const dfsRes = creatRes(n);
    if (canSet(dfsRes, 0, i)) {
      dfsRes[0][i] = "Q";
      dfs(dfsRes, 1, res);
      dfsRes[0][i] = ".";
    }
  }
  return res;
};
```

实现2

```
/**
 * @param {number} n
 * @return {string[][]}
 */
const valid = (chess, row, col) => {
  // 检查所有列
  for (let i = 0; i < row; i++) {
    if (chess[i][col] == "Q") {
      return false;
    }
  }
  // 检查45度角的
  for (let i = row - 1, j = col + 1; i >= 0 && j < chess.length; i--, j++) {
    if (chess[i][j] == "Q") {
      return false;
    }
  }
  // 检查135度角的
  for (let i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
    if (chess[i][j] == "Q") {
      return false;
    }
```

```
  }
  return true;
};
const solve = (res, chess, row) => {
  if (row === chess.length) {
    const temp = chess.map((item) => {
      return item.join("");
    });
    res.push(temp);
    return;
  }
  for (let col = 0; col < chess.length; col++) {
    if (valid(chess, row, col)) {
      chess[row][col] = "Q";
      solve(res, chess, row + 1);
      chess[row][col] = ".";
    }
  }
};
// Runtime: 80 ms, faster than 97.11% of JavaScript online submissions for N-Queens.
// Memory Usage: 40.8 MB, less than 76.53% of JavaScript online submissions for N-Queens.
// Next challenges:
export default (n) => {
```

```
  let res = [];
  // 定义棋盘
  const chess = [];
  for (let i = 0; i < n; i++) {
    chess[i] = new Array(n).fill(".");
  }
  solve(res, chess, 0);
  return res;
};
```

37. 解数独

题目描述

编写一个程序，通过填充空格来解决数独问题。

一个数独的解法需遵循如下规则：

数字 1-9 在每一行只能出现一次。
数字 1-9 在每一列只能出现一次。
数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

空白格用 '.' 表示。

例子1

input：

[

["5", "3", ".", ".", "7", ".", ".", ".", "."],

["6", ".", ".", "1", "9", "5", ".", ".", "."],

[".", "9", "8", ".", ".", ".", ".", "6", "."],

["8", ".", ".", ".", "6", ".", ".", ".", "3"],

["4", ".", ".", "8", ".", "3", ".", ".", "1"],

["7", ".", ".", ".", "2", ".", ".", ".", "6"],

[".", "6", ".", ".", ".", ".", "2", "8", "."],

[".", ".", ".", "4", "1", "9", ".", ".", "5"],

[".", ".", ".", ".", "8", ".", ".", "7", "9"],
]

output：

[

["5", "3", "4", "6", "7", "8", "9", "1", "2"],

["6", "7", "2", "1", "9", "5", "3", "4", "8"],

["1", "9", "8", "3", "4", "2", "5", "6", "7"],

["8", "5", "9", "7", "6", "1", "4", "2", "3"],

["4", "2", "6", "8", "5", "3", "7", "9", "1"],

["7", "1", "3", "9", "2", "4", "8", "5", "6"],

["9", "6", "1", "5", "3", "7", "2", "8", "4"],

["2", "8", "7", "4", "1", "9", "6", "3", "5"],

["3", "4", "5", "2", "8", "6", "1", "7", "9"],
]

提示：

1 给定的数独序列只包含数字 1-9 和字符 '.' 。
2 你可以假设给定的数独只有唯一解
3 给定数独永远是 9x9 形式的。

思考1

刚开始的时候想使用上下左右的深度搜索来解决此问题，可是发现逻辑越写越乱

后来看了题解，思路其实很简单

采用从上到下，一行行的遍历，如果遇到数字，则选择同一行的下一个位置继续遍历，这样就避免了上下左右的遍历出现逻辑复杂的问题

同时这里也提示了一个思路，如果是原地修改的，可以采用这种一行行遍历的手段

有了按行遍历，其他的就好解决了。

可以看下实现1

实现1

```
const isValid = (board, i, j, setChar) => {
  // 每列不能有重复的元素
  for (let m1 = 0; m1 < 9; m1++) {
    if (board[m1][j] === setChar) {
      return false;
    }
  }
  // 每行不能有重复的元素
  for (let m2 = 0; m2 < 9; m2++) {
    if (board[i][m2] === setChar) {
      return false;
    }
  }
  // 确定每个粗线格子是否存在同样的
  let row = i - (i % 3);
```

```
  let col = j - (j % 3);
  for (let x = 0; x < 3; x++) {
    for (let y = 0; y < 3; y++) {
      if (board[x + row][y + col] === setChar) {
        return false;
      }
    }
  }
  return true;
};
// 深层遍历，先行后列
const dfs = (board, i, j) => {
  // 如果全部遍历完了，返回true
  if (i === 9) {
    return true;
  }
  // 一行遍历完了，继续下一行
  if (j >= 9) {
    return dfs(board, i + 1, 0);
  }
  // 如果是数字，到下一个
  if (board[i][j] !== ".") {
    return dfs(board, i, j + 1);
```

```
  }
  for (let m1 = 1; m1 <= 9; m1++) {
    const temp = "" + m1;
    if (!isValid(board, i, j, temp)) {
      continue;
    }
    board[i][j] = temp;
    if (dfs(board, i, j + 1)) {
      return true;
    }
    board[i][j] = ".";
  }
  return false;
};
// Runtime: 116 ms, faster than 69.58% of JavaScript online submissions for Sudoku Solver.
// Memory Usage: 38.9 MB, less than 96.44% of JavaScript online submissions for Sudoku Solver.
export default (board) => {
  dfs(board, 0, 0);
  return board;
};
```

时间复杂度O(9^m),m表示需要填充的“.”长度
空间复杂度O(9^m)，因为每次需要创建一个临时变量

广度搜索

广度搜索就是一层层的搜索，使用先进先出队列，一层层的搜索，思想也很简单。

934 两个岛屿的最短距离

题目描述

给定一个二维 0-1 矩阵，其中 1 表示陆地，0 表示海洋，每个位置与上下左右相连。已知矩 阵中有且只有两个岛屿，求最少要填海造陆多少个位置才可以将两个岛屿相连。

例子1

input:

[[1,1,1,1,1],

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

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

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

[1,1,1,1,1]]

output: 1

思考

就是求两个岛屿的最短距离，难点也是如何求出两个岛屿的最短距离？

一个是深度搜索，找到最短距离

一个是广度搜索，直接找到最短距离

实现1

```
/**
 * @param {number[][]} A
 * @return {number}
 */
const dirs = [
  [-1, 0],
  [0, 1],
  [1, 0],
  [0, -1],
];
```

```
const dfs = (firstBounds, A, m, n, i, j) => {
  if (A[i][j] === 0) {
    firstBounds.push([i, j]);
    return;
  }
  A[i][j] = 2;
  for (let k = 0; k < 4; k++) {
    const nextI = i + dirs[k][0];
    const nextJ = j + dirs[k][1];
    if (nextI >= 0 && nextI < m && nextJ >= 0 && nextJ < n && A[nextI][nextJ] != 2) {
      dfs(firstBounds, A, m, n, nextI, nextJ);
    }
  }
};

export default (A) => {
  const m = A.length;
  const n = A[0].length;

  let flipped = false;
  // 第一个岛屿的边界
  const firstBounds = [];
  for (let i = 0; i < m; i++) {
```

```
    if (flipped) {
      break;
    }
    for (let j = 0; j < n; j++) {
      if (A[i][j] === 1) {
        dfs(firstBounds, A, m, n, i, j);
        flipped = true;
        break;
      }
    }
  }
  let x, y;
  let level = 0;
  while (firstBounds.length > 0) {
    level++;
    let n_firstBounds = firstBounds.length;
    while (n_firstBounds--) {
      const [r, c] = firstBounds[0];
      firstBounds.shift();
      for (let k = 0; k < 4; k++) {
        x = r + dirs[k][0];
        y = c + dirs[k][1];
        if (x >= 0 && y >= 0 && x < m && y < n) {
```

```
          if (A[x][y] === 2) {
            continue;
          }
          if (A[x][y] === 1) {
            console.log(level);
            return level;
          }
          firstBounds.push([x, y]);
          A[x][y] = 2;
        }
      }
    }
  }
  return 0;
};
```

算法时间复杂度O（m * n)
空间复杂度O（m * n）

126 单词搜索II

题目描述

给定一个起始字符串和一个终止字符串，以及一个单词表，求是否可以将起始字符串每次改 一个字符，直到改成终止字符串，且所有中间的修改过程表示的字符串都可以在单词表里找到。 若存在，输出需要修改次数最少的所有更改方式。

例子1：

input： beginWord = "hit", endWord = "cog",wordList = ["hot","dot","dog","lot","log","cog"]

output: [["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]]

例子2：

input： beginWord = "hit", endWord = "cog",
wordList = ["hot","dot","dog","lot","log"]

output: []

解释：因为cog不在wordList里边，所以不可能转换成功

思考

1 这个的思路也很简单，就是采用从beginWord和endWord开始进行广度遍历，如果两者相碰，就找到了结果

这里有两点提示

如果寻找两者最短距离，一般采用广度遍历

代码里边有个技巧，从开始beginword遍历的集合和从endword遍历的集合，交换运行

实现1

```
// Runtime: 132 ms, faster than 98.14% of JavaScript online submissions for Word Ladder II.
// Memory Usage: 42.8 MB, less than 100.00% of JavaScript online submissions for Word Ladder II.
const wordCanTransformOtherWord = (word, otherWord) => {
  let count = 0;
  for (let i = 0; i < word.length; i++) {
    if (word.charAt(i) !== otherWord.charAt(i)) {
      count++;
    }
    if (count > 1) {
      return false;
    }
  }
  return count === 1;
};
var findLadders = function (beginWord, endWord, wordList) {
  const wordSet = new Set(wordList);

  if (!wordSet.has(endWord)) {
```

```
  return [];
}
wordSet.delete(beginWord);
wordSet.delete(endWord);
// 从beiginWord开始广度搜索
let beginSet = new Set([beginWord]);
// 从endWord开始广度搜索
let endSet = new Set([endWord]);
// 从该word开始的路径
const fromWordPath = {};
// 结果
const res = [];
// 如果从beginWord开始广度遍历和从结束广度遍历都没结束
while (beginSet.size > 0 && endSet.size > 0) {
  // 轮流进行遍历，从开头遍历之后，接着从结尾开始遍历
  if (beginSet.size > endSet.size) {
    const temp = beginSet;
    beginSet = endSet;
    endSet = temp;
  }

  const newSet = new Set();
  // hit
```

```
    for (let w of beginSet) {
      for (let i = 0; i < wordList.length; i++) {
        if (wordCanTransformOtherWord(w, wordList[i])) {
          const newWord = wordList[i];
          // 找到了从前遍历和从后遍历的交点
          if (endSet.has(newWord)) {
            // console.log(fromWordPath, w, newWord);
            buildPath(w, [w, newWord], newWord);
          }
          if (wordSet.has(newWord)) {
            newSet.add(newWord);
            fromWordPath[newWord] = fromWordPath[newWord] || [];
            // fromWordPath[newWord] = [];
            fromWordPath[newWord].push(w);
          }
        }
      }
    }
    if (res.length > 0) {
      return res;
    }
    // 删除已经被选择过的单词
    newSet.forEach((w) => wordSet.delete(w));
```

```
    beginSet = newSet;
  }
  return [];

  function buildPath(s, path, d) {
    console.log(fromWordPath);
    if (!fromWordPath[s] && !fromWordPath[d]) {
      return res.push(path[0] === beginWord ? path.slice() : path.slice().reverse());
    }

    if (fromWordPath[s]) {
      fromWordPath[s].forEach((w) => buildPath(w, [w, ...path], d));
    } else {
      fromWordPath[d].forEach((w) => buildPath(s, [...path, w], w));
    }
  }
};
export default findLadders;
```

310 找出给出最小高度数的根节点

题目描述

树是一个无向图，其中任何两个顶点只通过一条路径连接。 换句话说，一个任何没有简单环路的连通图都是一棵树。

给你一棵包含 n 个节点的数，标记为 0 到 n - 1 。给定数字 n 和一个有 n - 1 条无向边的 edges 列表（每一个边都是一对标签），其中 edges[i] = [ai, bi] 表示树中节点 ai 和 bi 之间存在一条无向边。

可选择树中任何一个节点作为根。当选择节点 x 作为根节点时，设结果树的高度为 h 。在所有可能的树中，具有最小高度的树（即，min(h)）被称为 最小高度树 。

请你找到所有的 最小高度树 并按 任意顺序 返回它们的根节点标签列表。
树的 高度 是指根节点和叶子节点之间最长向下路径上边的数量。

例子1：

input：n = 4, edges = [[1,0],[1,2],[1,3]]

output: [1]
解释：如图所示，当根是标签为 1 的节点时，树的高度是 1 ，这是唯一的最小高度树。

例子2：

input： n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]

output: [3,4]

例子3：

input： n = 1, edges = []

output: [0]

例子4：

input： n = 2, edges = [[0,1]]

output: [0，1]

提示：

1 1 <= n <= 2 * 10^4
2 edges.length == n - 1
3 0 <= ai, bi < n
4 ai != bi
5 所有 (ai, bi) 互不相同
6 给定的输入 保证 是一棵树，并且 不会有重复的边

思考

1 直接进行广度遍历，遍历每个节点，然后查看每个节点作为根节点的高度，查找到高度最小的节点输出就可以。

可以查看实现1，不过这里会超时

2 还有一种方法，不过这种不容易想到。

首先可以把它想象成一张图，类似一张网，因为我们想找到座位根节点最低的树，肯定是在图最里边的，越靠近里边，肯定是越低

这里主要是如何想到使用图这种来解决此问题?

为什么在图最里边的就是最低树的根节点?

根据基本常识应该可以理解在图最里边的是最低树的根节点

那么剩下的问题就是如何发现最里边的节点了？

可以想下，如果是你，如何像剥葱一样，一层层的剥掉外面的节点找到最里边的节点

其实很简单，最外面的叶子节点的入度都是1，不断的删掉叶子节点，同时把与叶子节点相邻的节点的入度减1就可以了

这里还有一点需要注意的是因为图里是没有环的，所以最后的结果肯定是只有一个节点或者两个节点。可以反证一下，如果最后三个节点会是什么样？

可以看下实现2

实现1

```
/**
 * @param {number} n
 * @param {number[][]} edges
 * @return {number[]}
 */
const gfs = (reached, visited, edges, deep) => {
  if (reached.length === 0) {
    return deep;
```

```
  }
  const nextRoot = [];
  for (let i = 0; i < reached.length; i++) {
    visited.push(reached[i]);
    for (let j = 0; j < edges.length; j++) {
      if (edges[j].includes(reached[i])) {
        if (edges[j][0] !== reached[i] && !visited.includes(edges[j][0])) {
          nextRoot.push(edges[j][0]);
        } else if (!visited.includes(edges[j][1])) {
          nextRoot.push(edges[j][1]);
        }
      }
    }
  }
  return gfs(nextRoot, visited, edges, deep + 1);
};
export default (n, edges) => {
  let min = Number.MAX_VALUE;
  let res = [];
  for (let i = 0; i < n; i++) {
    let visited = [];
    const nextRoot = [];
    visited.push(i);
```

```
    for (let j = 0; j < edges.length; j++) {
      if (edges[j].includes(i)) {
        if (edges[j][0] !== i && !visited.includes(edges[j][0])) {
          nextRoot.push(edges[j][0]);
        } else if (!visited.includes(edges[j][1])) {
          nextRoot.push(edges[j][1]);
        }
      }
    }
    const deep = gfs(nextRoot, visited, edges, 0);
    console.log(i, deep);
    min = Math.min(min, deep);
    res.push({
      node: i,
      deep,
    });
  }
  res = res
    .filter((item) => item.deep === min)
    .map((item) => {
      return item.node;
    });
  console.log(res);
```

```
  return res;
};
```

实现2

```
/**
 * @param {number} n
 * @param {number[][]} edges
 * @return {number[]}
 */
//  Runtime: 104 ms, faster than 90.27% of JavaScript online submissions for Minimum Height Trees.
//  Memory Usage: 46.4 MB, less than 81.88% of JavaScript online submissions for Minimum Height Trees.
export default (n, edges) => {
  if (n === 1) return [0];

  const adj = new Array(n);
  for (let i = 0; i < n; ++i) {
    adj[i] = new Set();
  }
  // 建立边的链接
```

```
// 记录每个节点和它相连的节点个数
for (let edge of edges) {
  adj[edge[0]].add(edge[1]);
  adj[edge[1]].add(edge[0]);
}

let leaves = new Set();
for (let i = 0; i < n; ++i) {
  // 如果是叶子节点
  if (adj[i].size === 1) {
    leaves.add(i);
  }
}
// 因为没有环，结果要么是一个节点，要么是两个节点
while (n > 2) {
  n -= leaves.size;
  const newLeaves = new Set();
  for (let i of leaves) {
    // 查找叶子节点，
    for (let j of adj[i]) {
      // 从连接的j节点中删除叶子节点
      adj[j].delete(i);
      // 如果也是叶子节点，加入叶子节点
```

```
        if (adj[j].size === 1) {
          newLeaves.add(j);
        }
      }
    }
    leaves = newLeaves;
  }
  return Array.from(leaves);
};
```

实现2时间复杂度和空间复杂度都是O（n）

深度广度搜索总结

深度主要是一直向下走，直到走不动了，然后再从自己走过的点重新选择一个继续走

广度就是一层层的遍历，遍历完一层，继续遍历下一层

这里需要注意的是不管深度还是广度都不能重复访问已经访问过的节点,否则会容易造成死循环

如果递归感觉比较复杂的话，可以写简单的测试用例，模拟一下调用过程或者画图模拟一下，这样比单纯的想容易的多

传递数组或者对象的时候，主要不要直接传递引用，应该传递值拷贝

还有在查找shenmsh 离的时候，一般使用广度搜索

至于回溯算法也比较简单，一般三步，首先修改状态，然后进入递归，递归完成之后，恢复状态。

回溯最重要的注意保持状态，什么时候修改状态，什么时候恢复状态，必须想清楚。

第7章动态规划到底规划啥?

动态规划 <br data-tomark-pass>

1.1 动态规划 <br data-tomark-pass>
什么是动态规划，就是利用已知的条件，推出未知的条件

动态规划最重要的就是要找出状态转移方程，根据前面已经求出的状态，找到状态转移方程，从前面的状态转移到后面的状态

一维动态规划

70. 爬楼梯

题目描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 n 是一个正整数。

例子1

Input: 2

output: 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶。

例子2

Input: 3

output: 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

思考

这基本上是最经典的动态规划题目，动态规划的关键是确立动态转移方程
这里很明显可以看出dp[n] = dp[n-1]+dp[n-2],也就是爬n个台阶的方法等于爬n-1个台阶的方法和爬n-2个台阶的方法

实现1

```
/**
 * @param {number} n
 * @return {number}
 */
// Runtime: 72 ms, faster than 88.54% of JavaScript online submissions for Climbing Stairs.
// Memory Usage: 38.3 MB, less than 56.43% of JavaScript online submissions for Climbing Stairs.
export default (n) => {
  if (n === 0) return 0;
  if (n === 1) return 1;
  if (n === 2) return 2;
  let pre1 = 1;
  let pre2 = 2;
  for (let i = 3; i <= n; i++) {
    const temp = pre2;
    pre2 = pre1 + pre2;
    pre1 = temp;
```

```
  }
  return pre2;
};
```

时间复杂度O（n），空间复杂度O（1）

198. 打家劫舍

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

例子1

Input: [1,2,3,1]

output: 4

解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4 。

例子2

Input: [2,7,9,3,1]

output: 12

解释： 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
偷窃到的最高金额 = 2 + 9 + 1 = 12

提示：

1 0 <= nums.length <= 100
2 0 <= nums[i] <= 400

思考

这基本上是最经典的动态规划题目，动态规划的关键是确立动态转移方程

动态规划一般是利用已知的前面保存的状态推出下面的状态

假设dp[n]是数组长度为n的房屋可以盗窃的最大金额，那么现在可以思考下当dp[n+1]的时候，可以盗窃的最大金额是什么呢?

可以很容易的想到dp[n+1]要么不盗窃n+1的房屋的金额，那么肯定是等于dp[n],要么盗窃n+1房屋的金额，那么当盗窃n+1房屋金额的时候，总盗窃金额是dp[n+1] = dp[n-2]+(n+1)房屋的金额

实现1

```
/**
 * @param {number[]} nums
```

```
 * @return {number}
 */
// Runtime: 68 ms, faster than 97.89% of JavaScript online submissions for House Robber.
// Memory Usage: 38.2 MB, less than 91.14% of JavaScript online submissions for House Robber.
export default (nums) => {
  if (!nums) return 0;
  if (nums.length === 1) return nums[0];
  if (nums.length === 2) return Math.max(nums[0], nums[1]);

  let preN2money = nums[0];
  let preN1money = Math.max(nums[0], nums[1]);
  for (let i = 2; i < nums.length; i++) {
    const temp = preN1money;
    preN1money = Math.max(preN1money, preN2money + nums[i]);
    preN2money = temp;
  }
  return preN1money;
};
```

时间复杂度O（n），空间复杂度O（1）

198. 打家劫舍II

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，能够偷窃到的最高金额。

例子1

Input: nums = [2,3,2]

output: 3

解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2）, 因为他们是相邻的。

例子2

Input: nums = [1,2,3,1]

output: 4

解释： 你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
偷窃到的最高金额 = 1 + 3 = 4 。

例子3

Input: nums = [0]

output: 0

提示：

1 0 <= nums.length <= 100
2 0 <= nums[i] <= 1000

思考

这里基本上延续上面的打家劫舍的题目，但是和上面不一样的是这里是一个循环，从末尾又连到了开始，所以这里该如何解决呢?

如果想要解决一个问题，首先就要明白问题是什么？如果把问题定义的很清楚，也就离解决问题不远了

那这里和打家劫舍有哪些不同呢?

如果想不到，可以写个测试用例，自己看看，比如房屋的金额分别是[2,3,2],我们会得到什么结果呢?

走到这里，相信基本上已经明白了问题在哪了，我们如果按照打家劫舍的解法去做，可以发现因为我们不确定从第一家还是第二家开始打劫，所以如果我们从第一家开始打劫的话，打劫到最后一家的时候，因为最后一家和第一家相连，所以就触发了报警

所以问题也就清楚了，我们如何确定当打劫到到最后一家的时候，如何确定我们没有打劫第一家呢?

应该很容易想到我们如果从第二家开始打劫，一直到最后一家，肯定不会触发报警。

但是从第二家开始是不是就是结果呢?

我们能不能也从第一家开始打劫呢?如果从第一家开始打劫，我们是不是打劫到倒数第二家就可以了?

可以发现这两种情况的最大的，就是我们可以打劫到的最大金额。

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 68 ms, faster than 98.02% of JavaScript online submissions for House Robber II.
// Memory Usage: 38.5 MB, less than 53.03% of JavaScript online submissions for House Robber II.
export default (nums) => {
  if (!nums) return 0;
  if (nums.length === 1) return nums[0];
  if (nums.length === 2) return Math.max(nums[0], nums[1]);
  let preN2money = nums[0];
  let preN1money = Math.max(nums[0], nums[1]);
  for (let i = 2; i < nums.length - 1; i++) {
    const temp = preN1money;
    preN1money = Math.max(preN1money, preN2money + nums[i]);
    preN2money = temp;
  }
  let preN2money2 = nums[1];
  let preN1money1 = Math.max(nums[1], nums[2]);
  for (let i = 3; i < nums.length; i++) {
```

```
    const temp = preN1money1;
    preN1money1 = Math.max(preN1money1, preN2money2 + nums[i]);
    preN2money2 = temp;
  }
  return Math.max(preN1money1, preN1money);
};
```

时间复杂度O（n），空间复杂度O（1）

343. 整数拆分

题目描述

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

例子1

Input: 2

output: 1

解释：2 = 1 + 1, 1 × 1 = 1。

例子2

Input: 10

output: 36

解释：10 = 3 + 3 + 4, 3 × 3 × 4 = 36。

思考

1 题目本身的状态转移方程一眼就可以看出来
dp[i] = Math.max(dp[i], dp[i-j] * d[j])

但是这里隐藏着一个坑，如果想不到，思路是对的，但是测试用例应该还是过不了的?

因为这里dp[j]可能比j小，dp[i-j]也可能比i-j小

比如dp[4]本来最大的值应该是2 * 2 ,但是
dp[2] * dp [2] =1， dp[3] * dp [1] = 3
很明显不对

实现1

```
/**
 * @param {number} n
 * @return {number}
 */
// Input: 2
```

```
// Output: 1
// Explanation: 2 = 1 + 1, 1 × 1 = 1.

// Input: 10
// Output: 36
// Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.
// Runtime: 72 ms, faster than 98.97% of JavaScript online submissions for Integer Break.
// Memory Usage: 38.3 MB, less than 82.47% of JavaScript online submissions for Integer Break.
export default (n) => {
  if (n <= 1) return 0;
  if (n === 2) return 1;
  const dp = new Array(n + 1).fill(0);
  dp[1] = 1;
  for (let i = 2; i <= n; i++) {
    for (let j = 1; 2 * j <= i; j++) {
      const maxJ = Math.max(j, dp[j]);
      const maxDis = Math.max(i - j, dp[i - j]);
      dp[i] = Math.max(dp[i], maxJ * maxDis);
    }
  }
  // console.log(dp);
  return dp[n];
};
```

时间复杂度O（n * lgn），空间复杂度O（n）

376. 摆动序列

题目描述

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。相反, [1,4,7,2,5] 和 [1,7,4,5,5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

例子1

Input: [1,7,4,9,2,5]

output: 6

解释：整个序列均为摆动序列。

例子2

Input: [1,17,5,10,13,15,10,5,16,8]

output: 7

解释：这个序列包含几个长度为 7 摆动序列，其中一个可为[1,17,10,13,10,16,8]。

例子3

Input: [1,2,3,4,5,6,7,8,9]

output: 2

解释：比如[1,2]就是一个摆动序列

思考

1 如果是单纯的增长或者减少使用动态规划比较简单，但是这里是一个摆动序列，可能就比较麻烦了

这种涉及到几种状态的，可以拆开

这里就是拆开两种状态，分别动态规划

参考实现1

压缩空间的参考实现2

实现1

```
/**
```

```
 * @param {number[]} nums
 * @return {number}
 */

// Runtime: 76 ms, faster than 72.34% of JavaScript online submissions for Wiggle Subsequence.
// Memory Usage: 38.5 MB, less than 44.68% of JavaScript online submissions for Wiggle Subsequence.
export default (nums) => {
  if (nums.length === 0 || !nums) return 0;
  if (nums.length <= 1) return 1;
  if (nums.length === 2) {
    return nums[0] !== nums[1] ? 2 : 1;
  }
  const len = nums.length;
  const up = new Array(len).fill(0);
  const down = new Array(len).fill(0);

  up[0] = 1;
  down[0] = 1;

  for (let i = 1; i < len; i++) {
    if (nums[i] > nums[i - 1]) {
      up[i] = down[i - 1] + 1;
      down[i] = down[i - 1];
```

```
    } else if (nums[i] < nums[i - 1]) {
      down[i] = up[i - 1] + 1;
      up[i] = up[i - 1];
    } else {
      down[i] = down[i - 1];
      up[i] = up[i - 1];
    }
  }

  return Math.max(down[nums.length - 1], up[nums.length - 1]);
};
```

时间复杂度O（n），空间复杂度O（n）

实现2

```
// Runtime: 76 ms, faster than 72.34% of JavaScript online submissions for Wiggle Subsequence.
// Memory Usage: 38.5 MB, less than 36.17% of JavaScript online submissions for Wiggle Subsequence.
export default (nums) => => {
  if (nums.length < 2) return nums.length;
  let down = 1;
  let up = 1;
```

```
  for (let i = 1; i < nums.length; i++) {
    if (nums[i] > nums[i - 1]) up = down + 1;
    else if (nums[i] < nums[i - 1]) down = up + 1;
  }
  return Math.max(down, up);
};
```

时间复杂度O（n），空间复杂度O（1）

650. 只有两个键的键盘

题目描述

最初在一个记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：

1 Copy All (复制全部) : 你可以复制这个记事本中的所有字符(部分的复制是不允许的)。
2 Paste (粘贴) : 你可以粘贴你上一次复制的字符。

给定一个数字 n 。你需要使用最少的操作次数，在记事本中打印出恰好 n 个 'A'。输出能够打印出 n 个 'A' 的最少操作次数。

例子1

Input: 3

output: 3

解释：
最初, 我们只有一个字符 'A'。
第 1 步, 我们使用 Copy All 操作。
第 2 步, 我们使用 Paste 操作来获得 'AA'。
第 3 步, 我们使用 Paste 操作来获得 'AAA'。

说明：
1 n 的取值范围是 [1, 1000] 。

思考

1 题目比较简单，就是单纯的dp

我刚开始实现的时候，只是考虑了偶数的情况，没有考虑到奇数也可以被整除的

比如当n=9的时候，9也是可以整除到3的。

参考实现1

实现1

```
/**
 * @param {number} n
 * @return {number}
 */
// Runtime: 88 ms, faster than 53.27% of JavaScript online submissions for 2 Keys Keyboard.
```

```
// Memory Usage: 39.3 MB, less than 31.78% of JavaScript online submissions for 2 Keys Keyboard.
export default (n) => {
  const dp = new Array(n + 1).fill(Number.MAX_VALUE);
  dp[0] = 0;
  dp[1] = 0;
  dp[2] = 2;
  for (let i = 3; i <= n; i++) {
    if (i % 2 === 0) {
      dp[i] = Math.min(dp[i], dp[i / 2] + 2, i);
    } else {
      dp[i] = Math.min(dp[i], i);
      for (let k = 3; k < Math.floor(i / 2); k++) {
        if (i % k === 0) {
          dp[i] = Math.min(dp[i], dp[k] + i / k);
        }
      }
    }
  }
  return dp[n];
};
```

时间复杂度O（n ^ 2），空间复杂度O（1）

413. 等差数列划分

题目描述

如果一个数列至少有三个元素，并且任意两个相邻元素之差相同，则称该数列为等差数列。

例如，以下数列为等差数列:
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
数组 A 包含 N 个数，且索引从0开始。数组 A 的一个子数组划分为数组 (P, Q)，P 与 Q 是整数且满足 0<=P<Q<N 。

如果满足以下条件，则称子数组(P, Q)为等差数组：

元素 A[P], A[p + 1], ..., A[Q - 1], A[Q] 是等差的。并且 P + 1 < Q 。

函数要返回数组 A 中所有为等差数组的子数组个数。

例子1

Input: A = [1, 2, 3, 4]

output: 3

解释：A 中有三个子等差数组: [1, 2, 3], [2, 3, 4] 以及自身 [1, 2, 3, 4]。

思考

1 第一种dp[n]表示n结尾的一共有多少种排列，所以dp[n+1]呢?

这里应该很容易想到dp[n+1] 等于dp[n]和以n+1结尾的数字可以组成的等差数列的和。

不过这里需要注意的是虽然给出的测试用例是排好序的，但是实际上并没有排好序，所以不要按照排好序来处理

可以看下实现1

2 这里如果换一种看法，如果让dp[n]还是表示以下标为n结尾一共有多少种排列?

那么dp[n+1]有多少种呢?

如果想不到，可以试试测试用例[1,2,3,4,5]

分别写下dp[1],dp[2]..dp[5]的结果，看下有什么规律

所以很容易看到dp[n+1] = dp[n]+1,最后因为我们统一的是每个位置的排列数，所以最后求和就可以了

可以看下实现2，明显可以看出实现2比实现1代码少了很多，而且思路也清晰了很多

实现1

```
/**
 * @param {number[]} A
 * @return {number}
 */
// Runtime: 84 ms, faster than 18.63% of JavaScript online submissions for Arithmetic Slices.
// Memory Usage: 41.1 MB, less than 5.88% of JavaScript online submissions for Arithmetic Slices.
```

```
export default (A) => {
  if (!A || A.length < 3) return [];
  let dp = [];
  if (A[2] - A[1] === A[1] - A[0]) {
    dp.push([A[0], A[1], A[2]]);
  }
  for (let i = 3; i < A.length; i++) {
    const tempArr = [];
    tempArr.push(A[i]);
    const val = A[i] - A[i - 1];
    for (let k = i - 1; k >= 0; k--) {
      if (tempArr[0] - A[k] === val) {
        tempArr.unshift(A[k]);
        if (tempArr.length >= 3) {
          dp.push([...tempArr]);
        }
      } else {
        break;
      }
    }
  }
  return dp.length;
};
```

```
```

时间复杂度O（n^2），空间复杂度O（n^2）

实现2

```
// Runtime: 76 ms, faster than 76.47% of JavaScript online submissions for Arithmetic Slices.
// Memory Usage: 38.4 MB, less than 51.96% of JavaScript online submissions for Arithmetic Slices.
export default (A) => {
  if (!A || A.length < 3) return [];
  const len = A.length;
  const dp = new Array(len).fill(0);
  for (let i = 2; i < len; ++i) {
    if (A[i] - A[i - 1] === A[i - 1] - A[i - 2]) {
      dp[i] = dp[i - 1] + 1;
    }
  }
  // console.log(dp);
  return dp.reduce((a, b) => a + b);
};
```

时间复杂度O（n）空间复杂度O（n）

53. 最大子序和

题目描述

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

进阶：
如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

例子1

Input: [-2,1,-3,4,-1,2,1,-5,4]

output: 6

解释：连续子数组 [4,-1,2,1] 的和最大，为 6。

例子2

Input: [1]

output: 1

例子3

Input: [0]

output: 0

例子4

Input: [-1]

output: -1

例子5

Input: [-2147483647]

output: -2147483647

提示：
1 1 <= nums.length <= 2 * 10^4
2 -2^31 <= nums[i] <= 2^31 - 1

思考

1 第一种dp[n]表示n结尾的一共有多少种排列，所以dp[n+1]呢?

这里应该很容易想到dp[n+1] 等于dp[n]和以n+1结尾的数字可以组成的等差数列的和。

不过这里需要注意的是虽然给出的测试用例是排好序的，但是实际上并没有排好序，所以不要按照排好序来处理

可以看下实现1

2 这里如果换一种看法，如果让dp[n]还是表示以下标为n结尾一共有多少种排列?

那么dp[n+1]有多少种呢?

如果想不到，可以试试测试用例[1,2,3,4,5]

分别写下dp[1],dp[2]..dp[5]的结果，看下有什么规律

所以很容易看到dp[n+1] = dp[n]+1,最后因为我们统一的是每个位置的排列数，所以最后求和就可以了

可以看下实现2，明显可以看出实现2比实现1代码少了很多，而且思路也清晰了很多

实现1

```
/**
 * @param {number[]} A
 * @return {number}
 */
// Runtime: 84 ms, faster than 18.63% of JavaScript online submissions for Arithmetic Slices.
// Memory Usage: 41.1 MB, less than 5.88% of JavaScript online submissions for Arithmetic Slices.
export default (A) => {
  if (!A || A.length < 3) return [];
  let dp = [];
  if (A[2] - A[1] === A[1] - A[0]) {
    dp.push([A[0], A[1], A[2]]);
  }
  for (let i = 3; i < A.length; i++) {
```

```
    const tempArr = [];
    tempArr.push(A[i]);
    const val = A[i] - A[i - 1];
    for (let k = i - 1; k >= 0; k--) {
      if (tempArr[0] - A[k] === val) {
        tempArr.unshift(A[k]);
        if (tempArr.length >= 3) {
          dp.push([...tempArr]);
        }
      } else {
        break;
      }
    }
  }
  return dp.length;
};
```

时间复杂度O（n^2），空间复杂度O（n^2）

实现2

```
// Runtime: 76 ms, faster than 76.47% of JavaScript online submissions for Arithmetic Slices.
```

```
// Memory Usage: 38.4 MB, less than 51.96% of JavaScript online submissions for Arithmetic Slices.
export default (A) => {
  if (!A || A.length < 3) return [];
  const len = A.length;
  const dp = new Array(len).fill(0);
  for (let i = 2; i < len; ++i) {
    if (A[i] - A[i - 1] === A[i - 1] - A[i - 2]) {
      dp[i] = dp[i - 1] + 1;
    }
  }
  // console.log(dp);
  return dp.reduce((a, b) => a + b);
};
```

时间复杂度O（n）空间复杂度O（n）

279. 完全平方数

题目描述

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

例子1

Input: n = 12

output: 3

解释：12 = 4 + 4 + 4.

例子2

Input: n = 13

output: 2
解释：13 = 4 + 9.

思考

1 动态规划最重要的是什么?

一般主要是找到状态转移方程，这里的状态转移其实也很简单

这里我是怎么想出来的呢?

观察测试用例
// Input: n = 12
// output: 3

// 解释：12 = 4 + 4 + 4.
//

// 例子2
// Input: n = 13
// output: 2
// 解释：13 = 4 + 9.

可以看到dp[13]是dp[4] + 3 * 3,dp[12] 是dp[8] + 2 * 2,而dp[8] = dp[4] + 2 * 2

所以这里的状态转移方程就是dp[n] = Math.min(dp[n-i]+1),并且n = n- i* i + 1

可以参考下实现1

实现1

```
/**
 * @param {number} n
 * @return {number}
 */

// Input:  n = 12
// output: 3

// 解释：12 = 4 + 4 + 4.
//

// 例子2
```

```
// Input:  n = 13
// output: 2
// 解释：13 = 4 + 9.

// dp[13] = dp[]

// dp[1] = 1
// dp[2] = 1+1
// dp[n] = dp[n-]
export default (n) => {
  if (n === 0) return 0;
  if (n === 1) return 1;
  if (n === 2) return 2;
  const dp = [];
  dp[0] = 0;
  dp[1] = 1;

  for (let i = 2; i <= n; i++) {
    dp[i] = Number.MAX_VALUE;
    const sqrtN = Math.floor(Math.sqrt(i));
    for (let j = 1; j <= sqrtN; j++) {
      dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
    }
```

```
  }
  return dp[n];
};
```

时间复杂度O（1到n的根号的和）， 空间复杂度O（n）

646. 最长数对链

题目描述

给出 n 个数对。 在每一个数对中，第一个数字总是比第二个数字小。

现在，我们定义一种跟随关系，当且仅当 b < c 时，数对(c, d) 才可以跟在 (a, b) 后面。我们用这种形式来构造一个数对链。

给定一个数对集合，找出能够形成的最长数对链的长度。你不需要用到所有的数对，你可以以任何顺序选择其中的一些数对来构造。

例子1

Input: [[1,2], [2,3], [3,4]]

output: 2

解释：最长的数对链是 [1,2] -> [3,4]

提示：

给出数对的个数在 [1, 1000] 范围内。

思考

1 题目比较简单，先排序，然后使用动态规划缓存前面的状态就可以了。

这里唯一需要注意的可能就是这里需要使用到前面的贪心算法，在遍历的过程中需要选择尾数比较小的数字

实现1

```
/**
 * @param {number[][]} pairs
 * @return {number}
 */
//  Runtime: 92 ms, faster than 92.21% of JavaScript online submissions for Maximum Length of Pair Chain.
//  Memory Usage: 42.8 MB, less than 74.03% of JavaScript online submissions for Maximum Length of Pair Chain.
export default (pairs) => {
  if (!pairs || pairs.length === 0) return [];

  if (pairs.length === 1 || pairs.length === 2) return [pairs[0]];
  pairs.sort((a, b) => a[0] - b[0]);
  const dp = [pairs[0]];
  // console.log(pairs);
  for (let i = 1; i < pairs.length; i++) {
```

```
    if (pairs[i][0] > dp[dp.length - 1][1]) {
      dp.push(pairs[i]);
    } else if (pairs[i][1] < dp[dp.length - 1][1]) {
      dp[dp.length - 1] = pairs[i];
    }
  }
  // console.log(dp);
  return dp.length;
};
```

时间复杂度O（n），空间复杂度O（n）

91. 解码方法

题目描述

一条包含字母 A-Z 的消息通过以下方式进行了编码：

'A' -> 1
'B' -> 2
...
'Z' -> 26

给定一个只包含数字的非空字符串，请计算解码方法的总数。

题目数据保证答案肯定是一个 32 位的整数。

例子1

Input: s = "12"

output: 2

解释：它可以解码为 "AB"（1 2）或者 "L"（12）

例子2

Input: s = "226"

output: 3

解释：它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。

例子3

Input: s = "0"

output: 0

例子4

Input: s = "1"

output: 1

例子5

Input: s = "2"

output: 1

思考

1 刚开始思考，通过"226" 这个测试用例，很容易的可以得出状态转移方程是dp[n] = dp[n-1] + dp[n-2] 或者dp[n] = dp[n-1]

本题的难度不在于状态转移方程的思考，而是在于各种不同情况的处理

比如这里你可以思考下以下测试用例如何处理
“12” => 2
“226” => 3
“00” => 0
“2101” => 1
"27" => 1
"1201234" => 3
"10011" => 0
"123123" => 9
"230" => 0

正常的思考路程肯定是处理各种情况，可以参考实现1

2 通过上面的情况，可以发现各种情况需要各种处理，代码虽然不是很复杂，但是需要处理的各种情况，都需要处理，代码显得冗余，而且不优雅，显得代码难看

可以把上面的各种情况进行统一处理下

这里我们需要思考下为什么上面的情况那么复杂，需要处理的情况为什么那么多?

可以发现就是“0”的各种情况，所以需要各种特殊处理

这里有两种情况

如果当前的数字不是0，那没什么说的，最基本的是dp[i] += dp[i-1]

然后就是要处理要不要再加上dp[i-2],换句话说就是最后两个数字是不是在1到26之间，如果在1到26之间，就需要加上dp[i-2],如果不在，则不需要

可以参考实现2，可以看到代码简洁了很多

实现1

```
/**
 * @param {string} s
 * @return {number}
 */
// dp[3] = dp[1]+dp[2]
// Runtime: 104 ms, faster than 22.88% of JavaScript online submissions for Decode Ways.
// Memory Usage: 42.5 MB, less than 15.00% of JavaScript online submissions for Decode Ways.
export default (s) => {
  if (s.length === 1 && +s > 0) {
```

```
  return 1;
} else if (s.length === 1 || s[0] === "0") {
  return 0;
}
const dp = [];
dp[0] = +s[0] > 0 ? 1 : 0;
dp[1] = +s[1] > 0 ? dp[0] + 1 : dp[0];

if (+s.substring(0, 2) > 26 && +s[0] > 0 && s[1] !== "0") {
  dp[1] = 1;
} else if (+s.substring(0, 2) > 26 && +s[0] > 0 && s[1] === "0") {
  return 0;
}

for (let i = 2; i < s.length; i++) {
  if (s[i] === "0" && +s.substring(i - 1, i + 1) < 27 && +s.substring(i - 1, i + 1) > 9) {
    dp[i] = dp[i - 2];
  } else if (+s[i - 1] > 0 && +s[i] > 0 && +s.substring(i - 1, i + 1) > 10 && +s.substring(i - 1, i + 1) < 27) {
    dp[i] = dp[i - 1] + dp[i - 2];
  } else if ((+s[i - 1] === 0 && +s[i] > 0) || (+s[i - 1] > 0 && +s[i] > 0 && +s.substring(i - 1, i + 1) > 26)) {
    dp[i] = dp[i - 1];
  } else if ((+s[i] === 0 && +s.substring(i - 1, i + 1) > 26) || (+s[i - 1] === 0 && +s[i] === 0)) {
    return 0;
```

```
    }
  }
  // console.log(dp);
  return dp[dp.length - 1];
};
```

时间复杂度O（n），空间复杂度O（n）

实现2

```
/**
 * @param {string} s
 * @return {number}
 */
export default (s) => {
  if (s.length === 0) return 0;

  const len = s.length;
  const dp = new Array(len + 1).fill(0);

  dp[0] = 1;
```

```
  // 当有一个字符的时候
  dp[1] = s[0] === "0" ? 0 : 1;

  for (let i = 2; i <= len; i++) {
    // 如果不等于0，肯定是等于dp[n-1]
    if (s[i - 1] !== "0") {
      dp[i] += dp[i - 1];
    }
    // console.log(dp[i], i);
    // 如果等于0或者小于6的情况下加上dp[n-2]
    if (s[i - 2] === "1" || (s[i - 2] === "2" && s[i - 1] <= "6")) {
      dp[i] += dp[i - 2];
    }
    // console.log(dp[i], i);
  }
  // console.log(dp);
  return dp[len];
};
```

时间复杂度O(n), 空间复杂度O(n)

139. 单词拆分

题目描述

给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明:

1 拆分时可以重复使用字典中的单词。
2 你可以假设字典中没有重复的单词。

例子1

Input: s = "leetcode", wordDict = ["leet", "code"]

output: true

解释：返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

例子2

Input: s = "applepenapple", wordDict = ["apple", "pen"]

output: true

解释：返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
注意你可以重复使用字典中的单词。

例子3

Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

output: false

思考

1 这里使用动态规划比较简单，很容易就可以想到dp[n]表示长度为n的字符串是否可以用wordDict表示，那么dp[n+1]呢?

这里应该比较简单,可以想下dp[n+1]什么时候为true，什么时候为false呢?

那肯定是如果发从s[n+1]到s[i]的字符串可以用wordDict表示，同时dp[i-1]也可以表示的时候，那么dp[n+1]肯定是true了。

实现1

```
/**
 * @param {string} s
 * @param {string[]} wordDict
 * @return {boolean}
 */

// Runtime: 88 ms, faster than 57.62% of JavaScript online submissions for Word Break.
// Memory Usage: 40.5 MB, less than 62.66% of JavaScript online submissions for Word Break.
export default (s, wordDict) => {
  // dp[i] 表示s中前i个字符是否可以在wordDict中表示
  const dp = new Array(s.length).fill(0);
```

```
  for (let i = 0; i < s.length; i++) {
    for (let j = i; j >= 0; j--) {
      const subStr = s.substring(j, i + 1);
      if (wordDict.includes(subStr) && ((dp[j - 1] === 1 && j > 0) || j === 0)) {
        dp[i] = 1;
      }
    }
  }
  return dp[s.length - 1];
};
```

时间复杂度O（n * n * wordDict.length），
空间复杂度O（n）

300. 最长上升子序列

题目描述

给定一个无序的整数数组，找到其中最长上升子序列的长度。

例子1

Input:

[10,9,2,5,3,7,101,18]

output: 4

解释： 解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。

例子2

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

output: 4

例子3

Input:
nums = [7,7,7,7,7,7,7]

output: 1

提示：

1 1 <= nums.length <= 2500
2 -10^4 <= nums[i] <= 10^4<br

思考

1 题目使用动态规划，dp[i]表示以nums[i]结尾的增长子数列

然后很容易就可以发现状态转移方程dp[i] 等于 i 前面所有小于nums[i]的子数列的最大值

代码比较简单，参考实现1

2 然后题目还要求实现O（n * lgn）的时间复杂度，涉及到lgn肯定是二分查找，可以在哪里查找呢?

在原数组中查找肯定不可能，那在哪里查找呢？可以发现在上面实现1中为什么是O(n * n)? 有哪些可以改进呢?

可以发现上面就是因为需要不断去遍历前面的结果，那么我们是否可以改进下，重新定义dp[i]的含义，让dp[i]表示包含i+1个数字的增长数列。

那么dp就是一个nums中增长子数列，现在已经知道dp[i],那么如何更新dp呢?

现在有新的nums[i+1]了，那么如何更新dp呢?

可以发现在dp中使用二分查找找到nums[i+1]的位置，根据什么查找位置呢?

就是nums[i+1]在dp中的位置pos，等于nums[i+1]>dp[pos] && nums[i+1]<dp[pos+1]，或者直接替换dp[dp.length-1]

num dp
10 [10] 10加入dp
9 [9] 9加入的时候，发现9比10小，为了更长子数列，9替换10
2 [2] 2加入的时候，发现2比9小，为了更长子数列，2替换 9
5 [2，5] 5 加入，发现比大，变为 [2，5]
3 [2,3] 3比5小
7 [2,3,7] 7 比 3大
101 [2,3,7,101] 101比7大
18 [2,3,7,18] 18比101小

可以参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
//  [10,9,2,5,3,7,101,18]
//  4
// Runtime: 184 ms, faster than 18.89% of JavaScript online submissions for Longest Increasing Subsequence.
// Memory Usage: 39.6 MB, less than 25.03% of JavaScript online submissions for Longest Increasing Subsequence.
export default (nums) => {
  const len = nums.length;
  const dp = new Array(len).fill(1);
  let max = 1;
  for (let i = 1; i < nums.length; i++) {
    for (let j = i - 1; j >= 0; j--) {
      if (nums[i] > nums[j]) {
        dp[i] = Math.max(dp[j] + 1, dp[i]);
        max = Math.max(dp[i], max);
      } else if (nums[i] === nums[j]) {
```

```
        dp[i] = Math.max(dp[i], dp[j]);
        max = Math.max(dp[i], max);
      }
    }
  }
  return max;
};
```

时间复杂度O（n * n), 空间复杂度O（ n）

实现2

```
/**
 * @param {number[]} nums
 * @return {number}
 */
//  [10,9,2,5,3,7,101,18]
//  4

const binarySearchPosition = (dp, target, high) => {
  let low = 0;
```

```
  while (low <= high) {
    let mid = Math.floor(low + (high - low) / 2);
    if (target === dp[mid]) return mid;
    else if (target < dp[mid]) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }
  return low;
};
// [10, 9, 2, 5, 3, 7, 101, 18];
// Runtime: 84 ms, faster than 90.03% of JavaScript online submissions for Longest Increasing Subsequence.
// Memory Usage: 40.2 MB, less than 19.47% of JavaScript online submissions for Longest Increasing Subsequence.
export default (nums) => {
  const len = nums.length;
  if (!nums || len === 0) return 0;
  if (len === 1) return 1;
  let dp = new Array(len).fill(Number.MAX_VALUE);
  for (let i = 0; i < len; i++) {
    let pos = binarySearchPosition(dp, nums[i], i);
    dp[pos] = nums[i];
    console.log(dp);
```

```
  }

  for (let i = dp.length - 1; i >= 0; i--) {
    if (dp[i] !== Number.MAX_VALUE) return i + 1;
  }

  return 0;
};
```

时间复杂度O（n * lgn），空间复杂度O(n)

二维动态规划

1143. 最长公共子序列

题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

例子1

Input: text1 = "abcde", text2 = "ace"

output: 3

解释：最长公共子序列是 "ace"，它的长度为 3。

例子2

Input: text1 = "abc", text2 = "abc"

output: 3

解释：最长公共子序列是 "abc"，它的长度为 3。

例子3

Input: text1 = "abc", text2 = "def"

output: 0

解释：两个字符串没有公共子序列，返回 0。

提示：

1 1 <= text1.length <= 1000

2 1 <= text2.length <= 1000
3 输入的字符串只含有小写英文字符。

思考

1 这里是求公共子序列，动态规划中一般dp[i]是指i结尾的子序列

因为这里涉及到两个字符串，很容易想到使用二维动态规划
dp[i][j]

dp[i][j] 可以用来表示第一个字符串中i结尾和第二个字符串j结尾的包含的公共子序列

那么状态转移方程是什么呢?

可以看下测试用例text1 = "abcde", text2 = "ace"
可以看到dp[1][1] = 1，因为"a"等于“a”，那么接下来问题就来了?

dp[i+1][j] 是什么?
dp[i][j+1]是什么?
dp[i][j]是什么?

可以看到dp[2][1]=1,dp[1][2] =1 ,dp[2][2] = 1

那么有什么关系呢?

可以看到如果dp[i+1]等于dp[j+1],那么 dp[i+1][j+1]=dp[i][j]+1,如果dp[i+1]不等于dp[j+1]，那么dp[i+1][j+1]要么等于dp[i+1][j]要么等于dp[i][j+1]，
也就是dp[i+1][j+1]=Math.max(dp[i+1][j],dp[i][j+1])

有了状态转移方程，代码就很容易了

参考实现1

实现1

```
/**
 * @param {string} text1
 * @param {string} text2
 * @return {number}
 */

//  Runtime: 116 ms, faster than 62.14% of JavaScript online submissions for Longest Common Subsequence.
//  Memory Usage: 48.4 MB, less than 91.40% of JavaScript online submissions for Longest Common Subsequence.
// export default (text1, text2) => {
  const m = text1.length;
  const n = text2.length;
  const dp = [];
  for (let i = 0; i <= m; i++) {
    dp[i] = new Array(n + 1).fill(0);
  }
  // console.log(dp);
  for (let i = 1; i <= m; ++i) {
    for (let j = 1; j <= n; ++j) {
      if (text1[i - 1] === text2[j - 1]) {
```

```
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }
  // console.log(dp);
  return dp[m][n];
};
```

时间复杂度O（m * n), 空间复杂度O（m * n）

583. 两个字符串的删除操作

题目描述

给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

例子1

Input: "sea", "eat"

output: 2

解释：第一步将"sea"变为"ea"，第二步将"eat"变为"ea"

提示：

1 给定单词的长度不超过500。
2 给定单词中的字符只含有小写字母。

思考

1 这里是上面的1143的变种题目，可以想下两者有那些不同?
做题不是目的，举一反三才是本质

那么两者有哪些不一样呢?

我们可以从1143得到那些提示，如何修改才能解决本题呢?

从上面的1143可以得出我们同样可以定义dp[i][j],但是这里的状态转移方程是什么呢?

很容易想到，如果word1[i]等于word[j],那么dp[i][j] 就等于dp[i-1][j-1]

这里比较困难的是如何处理word1[i]不等于word[j]的时候，如何处理呢?

可以想下

这里可以提供一个思路，一般二维动态规划的状态转移方程，dp[i][j] 一般和dp[i-1][j],dp[i][j-1],
dp[i-1][j-1] 有关

然后应该可以得出
dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + 1;

参考实现1

实现1

```
/**
 * @param {string} word1
 * @param {string} word2
 * @return {number}
 */

// "sea", "eat";

// Runtime: 108 ms, faster than 93.48% of JavaScript online submissions for Delete Operation for Two Strings.
// Memory Usage: 45 MB, less than 75.00% of JavaScript online submissions for Delete Operation for Two Strings.
export default (word1, word2) => {
  if (word1 === word2) {
    return 0;
  }
  if (word1.length === 0) {
    return word2.length;
  }
  if (word2.length === 0) {
    return word1.length;
```

```
}
const m = word1.length;
const n = word2.length;
const dp = [];
for (let i = 0; i <= m; i++) {
  dp[i] = new Array(n + 1).fill(0);
}
for (let i = 1; i <= n; i++) {
  dp[0][i] = i;
}
for (let i = 1; i <= m; i++) {
  dp[i][0] = i;
}
for (let i = 1; i <= m; i++) {
  for (let j = 1; j <= n; j++) {
    if (word1[i - 1] === word2[j - 1]) {
      dp[i][j] = dp[i - 1][j - 1];
    } else {
      dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + 1;
    }
  }
}
// console.log(dp);
```

```
    return dp[m][n];
};
```

时间复杂度O（m * n), 空间复杂度O（m * n）

64. 最小路径和

题目描述

给定一个包含非负整数的 m x n 网格 grid ， 请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

例子1

Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

output: 7

解释：因为路径 1→3→1→1→1 的总和最小。

例子2

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

output: 12

提示：

1 m == grid.length
2 n == grid[i].length<br
3 1 <= m, n <= 200
4 0 <= grid[i][j] <= 100

思考

1 题目比较简单，状态转移方程很容易就可以看出来

dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];

参考实现1

实现1

```
/**
 * @param {number[][]} grid
 * @return {number}
 */
// Runtime: 84 ms, faster than 68.34% of JavaScript online submissions for Minimum Path Sum.
// Memory Usage: 41.1 MB, less than 30.49% of JavaScript online submissions for Minimum Path Sum.
export default (grid) => {
  const m = grid.length;
```

```
  const n = grid[0].length;
  if (m === 1 && n === 1) {
    return grid[0][0];
  }
  const dp = [];
  for (let i = 0; i < m; i++) {
    dp[i] = new Array(n).fill(0);
  }
  dp[0][0] = grid[0][0];
  for (let i = 1; i < n; i++) {
    dp[0][i] = dp[0][i - 1] + grid[0][i];
  }
  for (let i = 1; i < m; i++) {
    dp[i][0] = dp[i - 1][0] + grid[i][0];
  }

  // dp[0];
  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
    }
  }
  return dp[m - 1][n - 1];
```

```
};
```

时间复杂度O（m * n), 空间复杂度O（m * n）

542. 01 矩阵

题目描述

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。

两个相邻元素间的距离为 1 。

例子1

Input:

[[0,0,0],

[0,1,0],

[0,0,0]]

output:

[[0,0,0],

[0,1,0],

[0,0,0]]

例子2

Input:

[[0,0,0],

[0,1,0],

[1,1,1]]

output:

[[0,0,0],

[0,1,0],

[1,2,1]]

提示：

1 给定矩阵的元素个数不超过 10000。
2 给定矩阵中至少有一个元素是 0。<br
3 矩阵中的元素只在四个方向上相邻: 上、下、左、右。

思考

1 状态转移方程很好找，如果matrix[i][j]等于1，则dp[i][j]是四个方向中最小的值加1

但是很明显这样不对，因为就会形成循环了

那么如果避免死循环呢

后来想到了可以先从上到下遍历一遍，然后再从下到上遍历一遍，可是发现这样有个情况不是很好处理，那就是不管从上到下，还是从下到上，到matrix[0][0]等于0的时候，如何设置dp[0][0]?

可以想想应该如何设置?

后来才发现，因为是需要找到最小的，我们刚开是就直接设置dp的每个值是MAX_VALUE就可以了，如果从上到下，肯定能找到最小的

当时陷入了一个怪区，总是希望先设置完dp[0][0],才能从上到下遍历

参考实现1

实现1

```
/**
 * @param {number[][]} matrix
 * @return {number[][]}
 */
// Runtime: 148 ms, faster than 96.10% of JavaScript online submissions for 01 Matrix.
// Memory Usage: 46.8 MB, less than 79.22% of JavaScript online submissions for 01 Matrix.
export default (matrix) => {
  const m = matrix.length;
```

```
  const n = matrix[0].length;
  if (m === 0) {
    return [[]];
  }
  const dp = [];
  for (let i = 0; i < m; i++) {
    dp[i] = new Array(n).fill(Number.MAX_VALUE);
  }
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (matrix[i][j] === 0) {
        dp[i][j] = 0;
      } else {
        if (i > 0) {
          dp[i][j] = Math.min(dp[i][j], dp[i - 1][j] + 1);
        }
        if (j > 0) {
          dp[i][j] = Math.min(dp[i][j], dp[i][j - 1] + 1);
        }
      }
    }
  }
  for (let i = m - 1; i >= 0; i--) {
```

```
    for (let j = n - 1; j >= 0; j--) {
      if (matrix[i][j] != 0) {
        if (i < m - 1) {
          dp[i][j] = Math.min(dp[i][j], dp[i + 1][j] + 1);
        }
        if (j < n - 1) {
          dp[i][j] = Math.min(dp[i][j], dp[i][j + 1] + 1);
        }
      }
    }
  }
  return dp;
};
```

时间复杂度O（m * n), 空间复杂度O（m * n）

221. 最大正方形

题目描述

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

例子1

Input:
matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]

output: 4

例子2

Input:
matrix = [["0","1"],["1","0"]]

output: 1

例子3

Input:
matrix = matrix = [["0"]]

output: 0

提示：

1 m == matrix.length
2 n == matrix[i].length<br
3 1 <= m, n <= 300
4 matrix[i][j] 为 '0' 或 '1'

思考

1 题目使用动态规划，这里一看就是使用二维动态规划

dp[i][j]表示以matrix[i][j]为结尾的1的最大正方形
那么状态转移方程呢?

这里状态转移方程从逻辑上很容易发现，但是在代码中很难实现

刚开始我想求dp[i][j]的值，分别以
dp[i-1][j-1],dp[i][j-1],dp[i-1][j]分别计算出包含matrix[i][j]的最大正方形，可是发现代码写起来特别乱，及时一些测试用例可以过，但是很多测试过不了

这里的难点就是如何确立简单的状态转移方程?

不过这里确实不好想到，最多可能就是使用测试用例，一个个总结规律

假设dp[i][j]的最大正方形是k^2,那么充分条件为 dp[i-1][j-1]、dp[i][j-1] 和 dp[i-1][j] 的值必须 都不小于 (k – 1)^2，否则 (i, j) 位置不可以构成一个边长为 k 的正方形。

所以如果dp[i-1][j-1]、dp[i][j-1] 和 dp[i-1][j]三个值中的最大正方形的边长最小值为 k – 1，也就是三者的最大正方形都不小于(k – 1)^2，则 dp[i][j] 位置一定且最大可以构成一个边长为 k 的正方形，因为dp[i][j]的最大正方形是k^2

代码比较简单，参考实现1

实现1

```javascript
/**
 * @param {character[][]} matrix
 * @return {number}
 */

// Runtime: 84 ms, faster than 94.48% of JavaScript online submissions for Maximal Square.
// Memory Usage: 42.2 MB, less than 29.75% of JavaScript online submissions for Maximal Square.

export default (matrix) => {
  const m = matrix.length;
  const n = matrix[0].length;
  if (m === 0 || n === 0) {
    return 0;
  }
  const dp = [];
  for (let i = 0; i < m; i++) {
    dp[i] = new Array(n).fill(0);
  }

  let max = 0;
  for (let i = 0; i < n; i++) {
    dp[0][i] = +matrix[0][i];
```

```
    max = Math.max(max, dp[0][i]);
  }

  for (let i = 0; i < m; i++) {
    dp[i][0] = +matrix[i][0];
    max = Math.max(max, dp[i][0]);
  }

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      if (matrix[i][j] === "1") {
        dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]) + 1;
      }
      max = Math.max(max, dp[i][j]);
    }
  }
  return max * max;
};
```

时间复杂度O（m * n), 空间复杂度O（m * n）

72. 编辑距离

题目描述

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

```
插入一个字符
删除一个字符
替换一个字符
```

例子1

Input: word1 = "horse", word2 = "ros"

output: 3

解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

例子2

Input:word1 = "intention", word2 = "execution"

output: 5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')

提示：
1 0 <= word1.length, word2.length <= 500
2 word1 和 word2 由小写英文字母组成

思考

1 已经做了这么多动态规划的题目，很容易想到dp[i][j] 表示word1的i个字符变成word2的j个字符最小距离

接下来就是寻找动态转移方程了

很容易想到有两种情况

当word1[i] 等于word2[j]的时候，
dp[i][j] = dp[i-1][j-1]

当word1[i] 不等于word2[j]的时候，
因为这里有三种操作，

如果增加那么
dp[i][j] = dp[i][j-1]+1
如果删除
dp[i][j] = dp[i-1][j]+1
如果替换
dp[i][j] = dp[i-1][j-1]+1

可以参考实现1

实现1

```
/**
 * @param {string} word1
 * @param {string} word2
 * @return {number}
 */

// (word1 = "horse"), (word2 = "ros");
// Runtime: 124 ms, faster than 40.13% of JavaScript online submissions for Edit Distance.
// Memory Usage: 42.8 MB, less than 90.97% of JavaScript online submissions for Edit Distance.
export default (word1, word2) => {
  const m = word1.length;
  const n = word2.length;
```

```
  const dp = [];
  for (let i = 0; i <= m; i++) {
    dp[i] = new Array(n + 1).fill(0);
  }
  for (let i = 0; i <= n; i++) {
    dp[0][i] = i;
  }
  for (let i = 0; i <= m; i++) {
    dp[i][0] = i;
  }
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (word1[i - 1] === word2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1];
      } else {
        dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1;
      }
    }
  }
  // console.log(dp);
  return dp[m][n];
};
```

```

```

时间复杂度O（m * n ）， 空间复杂度O（ m * n）

10. 正则表达式匹配

题目描述

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

```
'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
```

所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

例子1

Input: s = "aa" p = "a"

output: false

解释：
"a" 无法匹配 "aa" 整个字符串。

例子2

Input: s = "aa" p = "a*"

output: true

解释：
因为 ' * ' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

例子3

Input: s = "ab" p = ".*"

output: true

解释：
"." *表示可匹配零个或多个（*''）任意字符（'.'）。

例子4

Input: s = "aab" p = "c*a*b"

output: true

解释：
因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。

例子5

Input: s = "mississippi" p = "mis*is*p*."

output: false

说明：
1 0 <= s.length <= 20
2 0 <= p.length <= 30
3 s 可能为空，且只包含从 a-z 的小写字母。
4 p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 * 。
5 保证每次出现字符 * 时，前面都匹配到有效的字符

思考

1 说实话这题本来还是很有难度的

如果第一次见到该题目，没法实现也很正常

这里那里比较难呢?

难在各种状态如何相互改变，各种条件是否都考虑到了，是否各种情况都考虑到了。

如果单纯看题目，本身其实也不是很难，就是要考虑各种情况，在各种情况下应该如何处理，但是很可能考虑的情况不完全或者考虑错误，很容易就测试用例过不了

这里大体上有两种定义dp[i][j]的定义方法，
一种dp[i][j]表示p的长度i匹配s的长度为j

一种dp[i][j]表示s的长度j匹配p的长度为i

两种思路差不多，但是感觉第一种比较符合逻辑，从下到上匹配比较符合常规思维

这里需要注意的几点是

1 为了考虑s为“”的情况，所以定义了一个dp[p.length + 1][s.length + 1]

这里也说明一种情况，在二维动态规划中一般都是定义这种

刚开始的时候，我本来想定义dp[p.length][s.length]在边界的情况下不是很好处理，所以可以记住以后凡是二维动态规划的时候都定义这种二维数组

2 就是各种情况下的处理，这个确实不是很好处理，但是如何看下代码你会发现很简单

参考实现1

实现2定义dp的方式刚好和实现1相反

实现1

```
/**
 * @param {string} s
 * @param {string} p
 * @return {boolean}
 */

// Runtime: 92 ms, faster than 93.90% of JavaScript online submissions for Regular Expression Matching.
// Memory Usage: 41.5 MB, less than 53.93% of JavaScript online submissions for Regular Expression Matching.
export default (s, p) => {
  const m = p.length;
  const n = s.length;
```

```
const dp = [];

for (let i = 0; i <= m; i++) {
  dp[i] = new Array(n + 1).fill(false);
}

dp[0][0] = true;

for (let i = 1; i <= m; i++) {
  if (p[i - 1] === "*") {
    if (i >= 2) {
      dp[i][0] = dp[i - 2][0];
    }
  }
}
for (let i = 1; i <= m; i++) {
  for (let j = 1; j <= n; j++) {
    if (p[i - 1] === ".") {
      dp[i][j] = dp[i - 1][j - 1];
    } else if (p[i - 1] === "*") {
      if (i >= 2) {
        dp[i][j] = dp[i - 2][j] || dp[i - 1][j];
```

```
        } else {
          dp[i][j] = dp[i - 1][j];
        }
        if (dp[i][j - 1] && (p[i - 2] === "." || p[i - 2] === s[j - 1])) {
          dp[i][j] = true;
        }
      } else {
        if (p[i - 1] === s[j - 1]) {
          dp[i][j] = dp[i - 1][j - 1];
        }
      }
    }
  }

  return dp[m][n];
};
```

时间复杂度O（n * m），空间复杂度O（n * m）

实现2

```
/**
```

```
 * @param {string} s
 * @param {string} p
 * @return {boolean}
 */

// dp[i][j];
// Runtime: 92 ms, faster than 93.90% of JavaScript online submissions for Regular Expression Matching.
// Memory Usage: 42.1 MB, less than 42.54% of JavaScript online submissions for Regular Expression Matching.
export default (s, p) => {
  const m = s.length;
  const n = p.length;
  const dp = [];
  for (let i = 0; i <= m; i++) {
    dp[i] = new Array(n + 1).fill(false);
  }

  dp[0][0] = true;

  for (let i = 1; i <= n; i++) {
    if (p[i - 1] == "*") {
      dp[0][i] = dp[0][i - 2];
    }
  }
```

```
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      //如果是.则只要dp[i - 1][j - 1] 为true则为true
      if (p[j - 1] == ".") {
        dp[i][j] = dp[i - 1][j - 1];
        // p[j-1]等于字母
      } else if (p[j - 1] != "*") {
        dp[i][j] = dp[i - 1][j - 1] && p[j - 1] == s[i - 1];
        // p[j-1] 等于“*”,
      } else if (p[j - 2] != s[i - 1] && p[j - 2] != ".") {
        dp[i][j] = dp[i][j - 2];
      } else {
        dp[i][j] = dp[i][j - 1] || dp[i - 1][j] || dp[i][j - 2];
      }
    }
  }
  return dp[m][n];
};
```

时间复杂度O（n * m）， 空间复杂度O（n * m）

0和1背包问题

背包问题是很经典的动态规划问题

有 N 个物品和容量为 W 的背包，每个物品都有
自己的体积 w 和价值 v，求拿哪些物品可以使得背包所装下物品的总价值最大。如果限定每种物 品只能选择 0 个或 1 个，则问题称为 0-1 背包问题

0和1背包问题如果使用动态规划应该很容易解决，dp[i][j]表示把i个物品放到容量为j的背包的最大的价值

然后就是定义状态转移方程。

这里很容易想到第i件物品，我们只有两种选择，一种是不放入背包中，那么dp[i][j] = dp[i-1][j]

第二种选择就是我们把i件物品放入到背包中，那么
dp[i][j] = dp[i-1][j-wi]+vi,wi是第i件物品的体积，vi是第i件物品的价值，也就是说如果我们把第i件物品放入的时候，那么此时整个背包的价值就等于dp[i-1][j-wi]加上vi（i件物品的价值）。

有了状态转移方程，那么代码就很容易实现了

```
/**
 * @param {number[]} weights
 * @param {number[]} values
 * @param {number} n
 * @param {number} w
```

```
 * @return {number}
 */
export default (weights, valuse, n, w) => {
  const dp = [];
  for (let i = 0; i <= n; i++) {
    dp[i] = new Array(w + 1).fill(0);
  }

  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= w; j++) {
      if (j >= weights[i-1]) {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weights[i-1]] + valuse[i-1]);
      } else {
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  return dp[n][w];
};
```

时间复杂度O(w * n),空间复杂度O(w * n)

当然基本上所有的动态规划都可以优化空间，可以看下当二维的时候，如下图

此时的状态转移方程是
dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weights[i]] + valuse[i]);

可以看到我们当求i=2的状态的时候，只需要记录i=1的状态就可以了，也就是可以使用一维数组表示

假设此时i=1的时候，背包容量分别是1，2，3，4，5的时候的背包价值dp=[2，3，4，5，6]，那么如果求出当i=2的时候的不同体积背包的价值呢，也就是dp的各个位置的值呢

假设此时有一个i物品的体积是2，价值是3<b
此时的还是有两种状态，要么放入i这件物品，要么不放入，

那么此时如何更新dp呢?

可以看到dp[j]要么等于不放入i物品的前一个dp[j]，要么等于放入i物品的时候，dp[i-wi]+vi。

此时状态转移方程是dp[j] =Math.max(dp[j],dp[i-wi]+vi)

但是如果正序修改dp的时候，此时可能发现dp[i-wi]可能已经放入了i物品了。所以状态转移方程就有问题了。

所以必须从后从前遍历

```
/**
 * @param {number[]} weights
 * @param {number[]} values
```

```
 * @param {number} n
 * @param {number} w
 * @return {number}
 */
export default (weights, valuse, n, w) => {
  const dp = new Array(w + 1).fill(0);

  for (let i = 1; i <= n; i++) {
    for (let j = w; j >= 0; j--) {
      if (j >= weights[i-1]) {
        dp[j] = Math.max(dp[j], dp[j - weights[i-1]] + valuse[i-1]);
      }
    }
  }
  return dp[w];
};
```

时间复杂度O(w * n),空间复杂度O(w)

416. 分割等和子集

题目描述

给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

注意：

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

例子1

Input: [1, 5, 11, 5]

output: true

解释： 数组可以分割成 [1, 5, 5] 和 [11].

例子2

Input: [1, 2, 3, 5]

output: false

解释： 数组不能分割成两个元素和相等的子集.

思考

1 这里要转换成0和1背包问题，其实还是有些难度，如何想到如何转换成0和1背包就按照0和1背包问题解决就可以了

主要是问题是一共有nums.length个数字，每个数字就涉及到选择或者不选择，这些可以当做0和1背包的物品

那么什么当做0和1背包的体积呢?

这是比较关键的，想下什么可以当做0和1背包的体积

这里一个比较重要的转换是就是所有数字的和的一半可以当做背包的体积。

我们选择任何数字，只要和等于所有数字的和的一半就可以了

这样就转换成了0和1背包的问题了

没有优化空间的，可以看下实现1

优化空间的，可以看下实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
// [1,5,11,5]
// 155 11
// 1235

// [1, 2, 3, 5];
// [1, 1, 2, 2];
```

```
// Runtime: 244 ms, faster than 45.49% of JavaScript online submissions for Partition Equal Subset Sum.
// Memory Usage: 70.8 MB, less than 31.03% of JavaScript online submissions for Partition Equal Subset Sum.
export default (nums) => {
  const len = nums.length;
  const sum = nums.reduce((a, b) => a + b);
  if (sum % 2 !== 0) return false;
  const target = sum / 2;
  const dp = new Array(len);
  for (let i = 0; i < len; i++) {
    dp[i] = new Array(target + 1).fill(false);
  }
  for (let i = 0; i < len; ++i) {
    dp[i][0] = true;
  }
  for (let i = 1; i < len; i++) {
    for (let j = 0; j <= target; ++j) {
      if (j >= nums[i - 1]) {
        dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
      } else {
        dp[i][j] = dp[i - 1][j];
      }
      if (j === target && dp[i][j]) {
        return true;
```

```
      }
    }
  }
  // console.log(dp);
  return dp[len - 1][target];
};
```

时间复杂度O（n^2），空间复杂度O（n^2）

实现2

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
// [1,5,11,5]
// 155 11
// 1235

// [1, 2, 3, 5];
// [1, 1, 2, 2];
```

```javascript
// Runtime: 108 ms, faster than 91.72% of JavaScript online submissions for Partition Equal Subset Sum.
// Memory Usage: 40.9 MB, less than 75.58% of JavaScript online submissions for Partition Equal Subset Sum.
export default (nums) => {
  const len = nums.length;
  const sum = nums.reduce((a, b) => a + b);
  if (sum % 2 !== 0) return false;
  const target = sum / 2;
  const dp = new Array(target + 1).fill(false);
  dp[0] = true;

  for (let i = 1; i < len; i++) {
    for (let j = target; j >= 0; j--) {
      if (j >= nums[i - 1]) {
        dp[j] = dp[j] || dp[j - nums[i - 1]];
      } else {
        dp[j] = dp[j];
      }
      if (j === target && dp[j]) {
        return true;
      }
    }
  }
  // console.log(dp);
```

```
    return dp[target];
};
```

时间复杂度O（n^2）， 空间复杂度O（n^2）

494. 目标和

题目描述

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

例子1

Input: nums: [1, 1, 1, 1, 1], S: 3

output: 5

解释：
-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1-1+1 = 3

+1+1+1+1-1 = 3

一共有5种方法让最终目标和为3。

提示

```
数组非空，且长度不会超过 20 。
初始的数组的和不会超过 1000 。
保证返回的最终结果能被 32 位整数存下。
```

思考

1 这里也是典型的0和1背包问题变形，也就是要么选择+nums[i]或者选择-nums[i]

状态转移方程也好解决

```
if (j + nums[i - 1] < maxN) {
    dp[i][j] += dp[i - 1][j + nums[i - 1]];
}
// 选择+nums[i - 1]
if (j - nums[i - 1] >= 0) {
    dp[i][j] += dp[i - 1][j - nums[i - 1]];
}
```

这里的难点是如何处理负数的情况?

如果没想清楚，可以看下代码，其实就是定义所有数据的和的两倍大小，这样就保证了-sum到sum都能包含进去

另外就是如何表示负数，刚开始的时候，我是想是否小于sum，来确定不同的状态转移方程，因为当是负数的情况下，状态转移方程是不一样的

比如nums: [1, 1, 1, 1, 1], S: 3
当求dp[2][5] = dp[1][4]+dp[1][6],但是实际上是不正确的，因为j<=sum的时候实际上表示的是负数

后来看了题解，原来是把当j=sum的时候表示0，j等于sum+1的时候表示正数1，当j等于sum-1的时候表示-1，这样就特别容易处理了

这里可以学习到表示从-sum到sum如何使用数组表示的方法

可以参考实现1

空间优化，参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @param {number} S
 * @return {number}
 */

// Runtime: 176 ms, faster than 76.29% of JavaScript online submissions for Target Sum.
```

```
// Memory Usage: 44.4 MB, less than 52.32% of JavaScript online submissions for Target Sum.
export default (nums, S) => {
  let sum = 0;
  for (let i of nums) {
    sum += i;
  }

  // 如果大于最大的和小于最小的
  if (S > sum || S < -sum) {
    return 0;
  }
  const dp = [];
  const len = nums.length;
  const maxN = 2 * sum + 1;
  for (let i = 0; i <= len; i++) {
    dp[i] = new Array(maxN).fill(0);
  }
  // 这里指全部选择负数的时候，只有一种选择
  dp[0][0 + sum] = 1;
  for (let i = 1; i <= nums.length; i++) {
    for (let j = 0; j < maxN; j++) {
      // 选择-nums[i - 1]
      if (j + nums[i - 1] < maxN) {
```

```
        dp[i][j] += dp[i - 1][j + nums[i - 1]];
      }
      // 选择+nums[i - 1]
      if (j - nums[i - 1] >= 0) {
        dp[i][j] += dp[i - 1][j - nums[i - 1]];
      }
    }
  }
  // console.log(dp);
  return dp[nums.length][sum + S];
};
```

时间复杂度O（n * m），空间复杂度O（n * m）

实现2

```
/**
 * @param {number[]} nums
 * @param {number} S
 * @return {number}
 */
```

```
// Runtime: 96 ms, faster than 96.39% of JavaScript online submissions for Target Sum.
// Memory Usage: 44.7 MB, less than 41.49% of JavaScript online submissions for Target Sum.
export default (nums, S) => {
  let sum = 0;
  for (let i of nums) {
    sum += i;
  }

  //
  if (S > sum || S < -sum) {
    return 0;
  }
  const len = 2 * sum + 1;
  let dp = new Array(len).fill(0);
  // 所有都选择负数
  dp[sum] = 1;

  for (let i = 0; i < nums.length; i++) {
    const next = new Array(len).fill(0);
    for (let k = 0; k < len; k++) {
      if (dp[k] != 0) {
        // 如果k有n中选择，那么当选择+ nums[i]的时候，肯定有n种，当选择- nums[i]的时候，肯定也有n种
```

```
            next[k + nums[i]] += dp[k];
            next[k - nums[i]] += dp[k];
        }
      }
      dp = next;
    }
    return dp[sum + S];
};
```

时间复杂度O（n * m），空间复杂度O（n ）

474. 一和零

题目描述

给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

例子1

Input: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3

output: 4

解释： 最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3 。

例子2

Input: strs = ["10", "0", "1"], m = 1, n = 1

output: 2

解释： 最大的子集是 {"0", "1"} ，所以答案是 2 。

提示：
1 1<= strs.length <= 600
2 1<= strs[i].length <= 100
3 strs[i] 仅由 '0' 和 '1' 组成
4 <= m, n <= 100

思考

1 这里题目感觉还是比较难的，如果是第一次接触，直接看答案就可以

这里涉及到一个三维数组，这种解法虽然说是0和1背包问题，但是实际上如果你理解为递归更容易理解

假设我们有一个函数memo是计算strs[i]开始的返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1

所以此时就有两种选择，要么加入strs[i]，要么不加入strs[i]，所以这里状态改变方程就是
Math.max(1 + memo(i), memo(i));

可以看下实现1

2 第二种方法就是定义dp[m][n] 表示已经遍历过strs从0到i的最大的子集的长度

那么状态转移方程是啥?

这里是三维降低到二维，可以类比想一下0和1背包问题从二维降低到一维的情况

所以这里dp[i][j] = Math.max(dp[i][j], 1 + dp[i - zeros][j - ones]) (zeros 表示strs[i]中0的长度， ones表示表示strs[i]中1的长度)

可以参考实现2

实现1

```
/**
 * @param {string[]} strs
 * @param {number} m
 * @param {number} n
 * @return {number}
 */

// ["10", "0001", "111001", "1", "0"];
// 5;
// 3;

// ["011111", "001", "001"], 4, 5;
```

```
const memo = (dp, start, m, n, size, strs) => {
  if (start >= size || m < 0 || n < 0) return 0;
  if (m === 0 && n === 0) return 0;
  // console.log(dp[start][m], start, m, n);
  if (dp[start][m][n] != -1) return dp[start][m][n];

  let res = 0;
  let i = start;
  let ones = 0;
  for (let k1 = 0; k1 < strs[i].length; k1++) {
    if (strs[i][k1] === "1") {
      ones++;
    }
  }
  let zeros = strs[i].length - ones;
  if (zeros <= m && ones <= n) {
    // 如果选择，则选择其中选择或者不选择中的最大的
    res = Math.max(1 + memo(dp, i + 1, m - zeros, n - ones, size, strs), memo(dp, i + 1, m, n, size, strs));
  } else {
    // 如果不符合规则，则直接下一个
    res = memo(dp, i + 1, m, n, size, strs);
  }
  dp[start][m][n] = res;
```

```
  return res;
};
// Runtime: 496 ms, faster than 34.21% of JavaScript online submissions for Ones and Zeroes.
// Memory Usage: 103 MB, less than 28.95% of JavaScript online submissions for Ones and Zeroes.
export default (strs, m, n) => {
  const len = strs.length;
  const dp = [];
  for (let i = 0; i < len; i++) {
    dp[i] = [];
    for (let k = 0; k <= m; k++) {
      dp[i][k] = new Array(n + 1).fill(-1);
    }
  }
  return memo(dp, 0, m, n, len, strs);
};
```

时间复杂度O（strs的长度 * strs[i]的长度）， 空间复杂度O（strs的长度 * m * n）

实现2

```
/**
```

```
 * @param {string[]} strs
 * @param {number} m
 * @param {number} n
 * @return {number}
 */

// Runtime: 140 ms, faster than 92.11% of JavaScript online submissions for Ones and Zeroes.
// Memory Usage: 40.9 MB, less than 78.95% of JavaScript online submissions for Ones and Zeroes.
export default (strs, m, n) => {
  const len = strs.length;
  const dp = [];
  for (let i = 0; i <= m; i++) {
    dp[i] = new Array(n + 1).fill(0);
  }

  for (let i = 0; i < len; i++) {
    let ones = 0;
    for (let k1 = 0; k1 < strs[i].length; k1++) {
      if (strs[i][k1] === "1") {
        ones++;
      }
    }
    let zeros = strs[i].length - ones;
```

```
    for (let k2 = m; k2 >= zeros; --k2) {
      for (let j = n; j >= ones; --j) {
        dp[k2][j] = Math.max(dp[k2][j], 1 + dp[k2 - zeros][j - ones]);
      }
    }
  }
  return dp[m][n];
};
```

时间复杂度O（strs.length * m * n）， 空间复杂度O（m * n）

完全背包问题

有 N 个物品和容量为 W 的背包，每个物品都有
自己的体积 w 和价值 v，求拿哪些物品可以使得背包所装下物品的总价值最大。如果限定每种物品可以选择多次，那么这里就是完全背包问题

完全背包问题和0和1背包问题差不多，解决思路也是类似，dp[i][j]表示把i个物品放到容量为j的背包的最大的价值

然后就是定义状态转移方程。

这里很容易想到第i件物品，在0和1背包的时候，我们只有两种选择，但是在完全背包的时候，我们是有多种选择的，要么选择一次，要么选择两次，要么选择三次，如果背包体积无穷大，我们甚至可以选择无数次

那么状态方程如何确定呢?

很容易想到

dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-wi*k]+vi * k) k>=1 && wi * k <=j

但是还是很复杂，不是很好解决

可以看下上图，此时
dp[2][5] = Math.max(dp[1][5], dp[1][3] + 3, dp[1][1] + 6)

那么应该如何改进上面的状态转移方程呢?

此时我们可以改下上面的状态转移方程，变成如下

dp[2][5] = Math.max(dp[1][5], Math.max(dp[1][3] , dp[1][1] + 3) + 3)

然后又以为知道
dp[2][3]= Math.max(dp[1][3] , dp[1][1] + 3)

所以此时

dp[2][5] = Math.max(dp[1][5],dp[2][3] + 3)

类似下图:

所以最后状态转移方程变成了

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

代码如下：

```
/**
 * @param {number[]} weights
 * @param {number[]} values
 * @param {number} n
 * @param {number} w
 * @return {number}
 */
export default (weights, valuse, n, w) => {
  const dp = [];
  for (let i = 0; i <= n; i++) {
    dp[i] = new Array(w + 1).fill(0);
  }

  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= w; j++) {
      // 如果选择,
      if (j >= weights[i - 1]) {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - weights[i - 1]] + valuse[i - 1]);
      } else {
```

```
        //如果不选择
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  return dp[n][w];
};
```

类似于0和1背包，这里也可以进行空间压缩，可以看下下图，这里也只需要使用一维数组就可以了,但是注意这里是使用了正向遍历

代码如下：

```
/**
 * @param {number[]} weights
 * @param {number[]} values
 * @param {number} n
 * @param {number} w
 * @return {number}
 */
export default (weights, valuse, n, w) => {
```

```
  const dp = new Array(w + 1).fill(0);
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= w; j++) {
      // 如果选择,
      if (j >= weights[i - 1]) {
        dp[j] = Math.max(dp[j], [j - weights[i]] + valuse[i]);
      }
    }
  }
  return dp[w];
};
```

322. 零钱兑换

题目描述

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

例子1

Input: coins = [1, 2, 5], amount = 11

output: 3

解释： 11 = 5 + 5 + 1。

例子2

Input:coins = [2], amount = 3

output: -1

例子3

Input:coins = [1], amount = 0

output: 0

例子4

Input:coins = [1], amount = 1

output: 1

例子5

Input:coins = [1], amount = 2

output: 2

提示：

1 1 <= coins.length <= 12
2 1 <= coins[i] <= 2^31 - 1
3 0 <= amount <= 10^4

思考

1 这是典型的完全背包问题，所以可以直接按照完全背包来解决就可以了

这里的技巧就是因为需要寻找最小的个数，所以应该如何设置最大数值呢?

可以看下代码中如何设置的

另外这里可能还要一点就是如何设置边界情况，就是二维数组dp
当i=0和当j=0的时候，应该如何设置?

这里如何边界情况设置不对的话，下面的测试用例是不过的

[186, 419, 83, 408], 6249 结果是20

其它的都比较简单，可以参考实现1

2 这里也可以优化空间，优化空间有个技巧，就是自己画图，把dp的图每步的画出来，可以看下那些不需要，那些需要，就很容易可以看出那些可以优化

这里可能状态转移
dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1) 有点不是很好理解

很多人认为得是dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1， dp[i-2* coins[j]] +2 ...) 一直到0

其实这里dp[i - coins[j]] 已经包含后面的情况

比如这里coins[j] =2 ,如果按照这个
dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1， dp[i-2* coins[j]] +2 ...)

dp[5]= Math.min(dp[5], dp[3]+1， dp[1]+2)

然后会发现
dp[5]= Math.min(dp[5], (dp[3]， dp[1]+1) + 1)
而dp[3] = Math.min(dp[3]， dp[1]+1),所以还是转换为了
dp[5]= Math.min(dp[5], dp[3]+1)

可以参考实现2

实现1

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
// Runtime: 184 ms, faster than 25.00% of JavaScript online submissions for Coin Change.
// Memory Usage: 44.9 MB, less than 19.34% of JavaScript online submissions for Coin Change.
export default (coins, amount) => {
  const len = coins.length;
```

```
const max = amount + 1;
const dp = [];
if (amount === 0) return 0;
for (let i = 0; i <= len; i++) {
  dp[i] = new Array(max).fill(max);
}
// coins.sort((a, b) => a - b);
for (let i = 0; i <= len; i++) {
  dp[i][0] = 0;
}
for (let i = 0; i <= amount; i++) {
  dp[0][amount] = max;
}
for (let i = 1; i <= len; i++) {
  for (let j = 0; j <= amount; j++) {
    if (j >= coins[i - 1]) {
      dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - coins[i - 1]] + 1);
    } else {
      dp[i][j] = dp[i - 1][j];
    }
  }
}
```

```
  return dp[len][amount] === max ? -1 : dp[len][amount];
};
```

时间复杂度O（m * n），空间复杂度O（m * n）

实现2

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
// [1, 2, 5], 11;
// Runtime: 124 ms, faster than 63.93% of JavaScript online submissions for Coin Change.
// Memory Usage: 42.7 MB, less than 88.09% of JavaScript online submissions for Coin Change.
export default (coins, amount) => {
  const Max = amount + 1;
  const dp = new Array(amount + 1).fill(Max);
  dp[0] = 0;
  for (let i = 1; i <= amount; i++) {
    for (let j = 0; j < coins.length; j++) {
```

```
      if (i >= coins[j]) {
        dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
      }
    }
  }
  return dp[amount] > amount ? -1 : dp[amount];
};
```

时间复杂度O（ m * n），空间复杂度O（n）

股票交易

动态规划来处理解决股票交易的问题，这里涉及到使用状态机来解决多种不同状态混杂的问题。

121. 买卖股票的最佳时机

题目描述

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

例子1

Input: [7,1,5,3,6,4]

output: 5

解释： 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

例子2

Input:[7,6,4,3,1]
output: 0
解释：在这种情况下, 没有交易完成, 所以最大利润为 0。

思考

1 题目比较简单,简单的一维动态规划就可以了

实现1

```
/**
 * @param {number[]} prices
 * @return {number}
 */
```

```
// Runtime: 96 ms, faster than 32.40% of JavaScript online submissions for Best Time to Buy and Sell Stock.
// Memory Usage: 39.5 MB, less than 24.29% of JavaScript online submissions for Best Time to Buy and Sell Stock.
export default (prices) => {
  const n = prices.length;
  const dp = new Array(n + 1).fill(0);

  for (let i = 2; i <= n; i++) {
    let max = 0;
    for (let j = i - 2; j >= 0; j--) {
      if (prices[j] < prices[i - 1]) {
        max = Math.max(max, prices[i - 1] - prices[j]);
      } else {
        break;
      }
    }
    dp[i] = Math.max(dp[i], max, dp[i - 1]);
  }
  return dp[n];
};
```

时间复杂度O（n^2 ）， 空间复杂度O（ n）

309. 最佳买卖股票时机含冷冻期

题目描述

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

> 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
> 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

例子1

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

output: 3

解释：对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

思考

1 如果第一次遇见这个问题，没有接触过状态机，直接看答案就可以。
本题属于状态机解决，这里如何确定有几种状态呢?

因为这里有三种动作，sell和buy和reset，所以对应三种状态

这三种状态如何转换很容易

可以看到通过buy和sell或者reset就可以转换到各种不同的状态

这里的问题就是如何把这三种状态和我们想要的题目答案结合起来?

我们想要求出第n天的时候，可以通过买和卖股票获取的最大利润，那么第n天可以获取的最大利润和什么有关系呢？换句话说和s0和s1和s2这三种状态有什么关系呢?

应该很容易想到第n天可以获取的最大利润肯定是Math.max(s0,s2),为什么呢?

因为s1的状态是通过s0在第n天购买获得的状态或者是通过第n天休息获得，但是这第n天休息的时候，其实手里还有没有卖出的股票。

剩下的就是确定初始状态了，初始状态应该很好确定，第0天s0，s1,s2都是0，0，0，那么第一天就0,-prices[0],0

这里还要需要注意的一点就是状态机只能从一个状态到另外一个状态不能跨越

参考实现1

当然这里可以压缩空间，可以参考实现2

实现1

```
/**
 * @param {number[]} prices
 * @return {number}
```

```
 */

// Runtime: 92 ms, faster than 34.36% of JavaScript online submissions for Best Time to Buy and Sell Stock with Cooldown.
// Memory Usage: 41 MB, less than 9.82% of JavaScript online submissions for Best Time to Buy and Sell Stock with Cooldown.
export default (prices) => {
  const n = prices.length;
  if (n <= 1) return 0;

  // sell之后reset，reset之后reset
  const s0 = new Array(n + 1).fill(0);
  // buy之后reset，reset之后reset
  const s1 = new Array(n + 1).fill(0);
  // sell之后
  const s2 = new Array(n + 1).fill(0);
  s0[1] = 0;
  s1[1] = -prices[0];
  s2[1] = 0;

  for (let i = 2; i <= n; i++) {
    s0[i] = Math.max(s0[i - 1], s2[i - 1]);
    s1[i] = Math.max(s0[i - 1] - prices[i - 1], s1[i - 1]);
    s2[i] = s1[i - 1] + prices[i - 1];
  }
```

```
  return Math.max(s0[n], s2[n]);
};
```

时间复杂度O（n），空间复杂度O（n）

实现2

```
/**
 * @param {number[]} prices
 * @return {number}
 */

// Runtime: 84 ms, faster than 64.42% of JavaScript online submissions for Best Time to Buy and Sell Stock with Cooldown.
// Memory Usage: 38.8 MB, less than 72.39% of JavaScript online submissions for Best Time to Buy and Sell Stock with Cooldown.
export default (prices) => {
  const n = prices.length;
  if (n <= 1) return 0;

  // sell之后reset，reset之后reset
  let s0 = 0;
  // buy之后reset，reset之后reset
  let s1 = -prices[0];
```

```
  // sell之后
  let s2 = 0;

  for (let i = 1; i < n; i++) {
    const last_s2 = s2;
    // 按照状态机顺序转换
    s2 = s1 + prices[i];
    s1 = Math.max(s0 - prices[i], s1);
    s0 = Math.max(s0, last_s2);
  }
  // console.log(s0, s2);
  return Math.max(s0, s2);
};
```

时间复杂度O（n），空间复杂度O（1）

714. 买卖股票的最佳时机含手续费

题目描述

给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格；非负整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每次交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

例子1

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

output: 8

解释：
能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

提示：
1 0 < prices.length <= 50000.
2 0 < prices[i] < 50000.
3 0 <= fee < 50000.

思考

1 题目很简单，直接和上面一样，建立三种状态，使用状态机

这三种状态如何转换很容易

可以看到通过buy和sell就可以转换到各种不同的状态

当然这里可以压缩空间，可以参考实现1

实现1

```
/**
 * @param {number[]} prices
 * @param {number} fee
 * @return {number}
 */

// Runtime: 92 ms, faster than 86.05% of JavaScript online submissions for Best Time to Buy and Sell Stock with Transaction Fee.
// Memory Usage: 47.4 MB, less than 77.91% of JavaScript online submissions for Best Time to Buy and Sell Stock with Transaction Fee.
export default (prices, fee) => {
  let s0 = 0;
  let s1 = -prices[0];
  let s2 = 0;

  for (let i = 1; i < prices.length; i++) {
    let preS2 = s2;
    s2 = s1 + prices[i] - fee;
```

```
    s0 = Math.max(s0, preS2);
    s1 = Math.max(s0 - prices[i], s1);
  }
  return Math.max(s0, s2);
};
```

时间复杂度O（n），空间复杂度O（1）

188. 买卖股票的最佳时机 IV

题目描述

给定一个整数数组 prices，它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

例子1

Input: k = 2, prices = [2,4,1]

output: 2

解释：
在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。

例子2

Input: k = 2, prices = [3,2,6,5,0,3]

output: 7

解释：
在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。

提示：
1 0 <= k <= 10^9
2 0 <= prices.length <= 1000
3 0 <= prices[i] <= 1000

思考

1 这里题目其实是比较难的

因为这里有两种状态，很明显可以想到使用二维动态规划，可是状态转移方程不是很好想出来。

刚开始做的时候，首先想到dp[i][j]表示第i次交易有j个股票的最大利润

但是一直想不出来如何得出状态转移方程

后来看了下题解，感觉也很简单，就是特别不好想出来

如果我们想要求dp[i+1][j+1]，这时候有两种选择，第一种是第i+1的时候，不卖出，那么这时候
dp[i+1][j+1] = dp[i+1][j] 也就是和第j+1天的股票没有任何关系

一种是在j+1天的时候，选择卖出股票，这里有个问题，如果想在第j+1天卖出的时候，必须在0-j的时候，必须买入一只股票，否则不能卖出，所以这个时候有j种选择，比如在第t天买入股票在j+1天卖出的时候，则dp[i+1][j+1] = dp[i][t-1]+ prices[j+1]-prices[t] (t>=0&&t<=j)

可以参考下实现1

在实现1中，可以发现,下面的代码其实每次都是重复的，其实每次，我们只需要使用tmpMax的最大值就可以了，可以参考实现2

```
for (let k1 = 2; k1 < j; k1++) {
      tmpMax = Math.max(tmpMax, dp[i - 1][k1] - prices[k1 - 1]);
    }
```

2 当然这里还有第二种解法，分别使用两个动态规划buy和sell，
buy[j]表示第j次交易购买的最大利润，sell[j]表示第j次交易卖出的最大利润

动态转移很容易
buy[j] = Math.max(buy[j],sell[j-1]-prices[i])
sell[j] =Math.max(sell[j], buy[j] + prices[j])

可以参考实现3

实现1

```javascript
/**
 * @param {number} k
 * @param {number[]} prices
 * @return {number}
 */
// 辅函数
const maxProfits = (prices) => {
  let maxProfit = 0;
  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      maxProfit += prices[i] - prices[i - 1];
    }
  }
  return maxProfit;
};

// Runtime: 92 ms, faster than 83.33% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
// Memory Usage: 40.3 MB, less than 86.23% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
export default (k, prices) => {
  const len = prices.length;
  if (len < 2) {
    return 0;
```

```
  }
  // 如果k的天数大于len
  if (k >= len) {
    return maxProfits(prices);
  }
  const dp = [];
  for (let i = 0; i <= k; i++) {
    dp[i] = new Array(len + 1).fill(0);
  }
  for (let i = 1; i <= k; i++) {
    // 默认第一天购买
    let tmpMax = -prices[0];
    for (let j = 2; j <= len; j++) {
      // 获取在前面0-j之间购买股票的最大利润
      for (let k1 = 2; k1 < j; k1++) {
        tmpMax = Math.max(tmpMax, dp[i - 1][k1] - prices[k1 - 1]);
      }
      // 有两种选择，第一种是啥也不干dp[i][j - 1]，第二种是如果在j天卖出股票的时候，必须在0到j之间购买一次
      dp[i][j] = Math.max(dp[i][j - 1], prices[j - 1] + tmpMax);
      // tmpMax = Math.max(tmpMax, dp[i - 1][j - 1] - prices[j - 1]);
    }
  }
  return dp[k][len];
```

```
};
```

时间复杂度O（n^2 * k），空间复杂度O（n * k）

实现2

```
/**
 * @param {number} k
 * @param {number[]} prices
 * @return {number}
 */
// 辅函数
const maxProfits = (prices) => {
  let maxProfit = 0;
  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      maxProfit += prices[i] - prices[i - 1];
    }
  }
  return maxProfit;
};
```

```
// Runtime: 92 ms, faster than 83.33% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
// Memory Usage: 40.3 MB, less than 86.23% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
export default (k, prices) => {
  const len = prices.length;
  if (len < 2) {
    return 0;
  }
  // 如果k的天数大于len
  if (k >= len) {
    return maxProfits(prices);
  }
  const dp = [];
  for (let i = 0; i <= k; i++) {
    dp[i] = new Array(len + 1).fill(0);
  }
  for (let i = 1; i <= k; i++) {
    // 默认第一天购买
    let tmpMax = -prices[0];
    for (let j = 2; j <= len; j++) {
      // // 获取在前面0-j之间购买股票的最大利润
      // for (let k1 = 2; k1 < j; k1++) {
      //   tmpMax = Math.max(tmpMax, dp[i - 1][k1] - prices[k1 - 1]);
      // }
```

```
      // 有两种选择，第一种是啥也不干dp[i][j - 1]，第二种是如果在j天卖出股票的时候，必须在0到j之间购买一次
      dp[i][j] = Math.max(dp[i][j - 1], prices[j - 1] + tmpMax);
      tmpMax = Math.max(tmpMax, dp[i - 1][j - 1] - prices[j - 1]);
    }
  }
  return dp[k][len];
};
```

时间复杂度O（n * k），空间复杂度O（n * k）

实现3

```
/**
 * @param {number} k
 * @param {number[]} prices
 * @return {number}
 */
// 辅函数
const maxProfits = (prices) => {
  let maxProfit = 0;
  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
```

```
      maxProfit += prices[i] - prices[i - 1];
    }
  }
  return maxProfit;
};

// Runtime: 84 ms, faster than 98.55% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
// Memory Usage: 39.8 MB, less than 92.03% of JavaScript online submissions for Best Time to Buy and Sell Stock IV.
export default (k, prices) => {
  const len = prices.length;
  if (len < 2) {
    return 0;
  }
  // 如果k的天数大于len
  if (k >= len) {
    return maxProfits(prices);
  }
  const buy = new Array(k + 1).fill(-Infinity);
  const sell = new Array(k + 1).fill(0);
  // buy[0] = -prices[0];
  // sell[0] = 0;
  for (let i = 1; i <= k; i++) {
    buy[i] = -prices[0];
```

```
    sell[i] = 0;
  }

  for (let i = 1; i < len; i++) {
    for (let j = 1; j <= k; j++) {
      let preBuy = buy[j];
      buy[j] = Math.max(buy[j], sell[j - 1] - prices[i]);
      sell[j] = Math.max(sell[j], preBuy + prices[i]);
    }
  }
  return sell[k];
};
```

时间复杂度O（n * k），空间复杂度O（n）

第8章拆箱和装箱

分治思想

这里思想很简单，但是真正涉及到题目的时候，会发现使用的时候还是很难的。

但是基本思想还是类似的，基本上就是把大问题拆成小问题，分别解决，然后再组合起来得到结果

241. 为运算表达式设计优先级

题目描述

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 * 。

例子1

Input: "2-1-1"

output: [0, 2]

解释：
((2-1)-1) = 0
(2-(1-1)) = 2

例子2

Input: "*23-45*"

output: [-34, -14, -10, -10, 10]

解释：
(2*(3-(4 * 5))) = -34
((2 * 3)-(4 * 5)) = -14
((2*(3-4)) * 5) = -10
(2*((3-4) * 5)) = -10
(((2 * 3)-4)* 5) = 10

思考

1 题目难度是中等的，但是感觉难度是hard

刚开始想对于不同运算符进行处理，可是思路一直没有理清楚，后来看了下题解，思路差不多，也是对于不同运算符进行处理，不过这里处理完了之后，采用相互乘得到结果

比如这里"*23-45*"，可以根据对不同字符进行处理，可以获得以下结果

(2(3-(45)))* = *-34*
(2((3-4)*5)) = -10

((*23*)-(*45*)) = -14

((2*(3-4))*5)* = *-10*
(((23)-4)*5) = 10

使用递归，可以参考实现1

可以发现实现1中有些字符是重复的，可以使用缓存，参考实现2

不使用递归，使用自底向上，采用动态规划解决，这里的dp[i][j] 表示在字符串中数字组成的数组中nums[i]到nums[j]的可以组成的不同结果，那么动态转移方程就是
dp[i][j] = dp[i][k] * dp[k+1][j] (k>=i&&k<j)
参考实现3

实现1

```
/**
 * @param {string} input
 * @return {number[]}
 */

// Runtime: 84 ms, faster than 61.84% of JavaScript online submissions for Different Ways to Add Parentheses.
// Memory Usage: 40.7 MB, less than 42.11% of JavaScript online submissions for Different Ways to Add Parentheses.
const diffWaysToCompute = (input) => {
  const ret = [];
  for (let i = 0; i < input.length; i++) {
    const charI = input.charAt(i);
    if (charI === "-" || charI === "*" || charI === "+") {
      const part1 = input.substring(0, i);
      const part2 = input.substring(i + 1);
```

```
    const part1Ret = diffWaysToCompute(part1);
    const part2Ret = diffWaysToCompute(part2);
    // console.log(part1, part1Ret);
    // console.log(part2, part2Ret);
    for (let p1 of part1Ret) {
      for (let p2 of part2Ret) {
        let c = 0;
        switch (charl) {
          case "+":
            c = p1 + p2;
            break;
          case "-":
            c = p1 - p2;
            break;
          case "*":
            c = p1 * p2;
            break;
        }
        ret.push(c);
      }
    }
    console.log(ret);
  }
```

```
  }
  if (ret.length === 0) {
    ret.push(+input);
  }
  return ret;
};
export default diffWaysToCompute;
```

实现2

```
/**
 * @param {string} input
 * @return {number[]}
 */
// Runtime: 80 ms, faster than 71.05% of JavaScript online submissions for Different Ways to Add Parentheses.
// Memory Usage: 41.1 MB, less than 34.21% of JavaScript online submissions for Different Ways to Add Parentheses.
const computeWithDP = (input, map) => {
  const res = [];
  const len = input.length;
  for (let i = 0; i < len; i++) {
    const charI = input.charAt(i);
```

```
if (charI == "+" || charI == "-" || charI == "*") {
  const part1Res = [];
  const part2Res = [];
  const part1 = input.substring(0, i);
  const part2 = input.substring(i + 1);
  if (map.has(part1)) {
    part1Res = map.get(part1);
  } else {
    part1Res = computeWithDP(part1, map);
    map.set(part1, part1Res);
  }
  if (map.has(part2)) {
    part2Res = map.get(part2);
  } else {
    part2Res = computeWithDP(part2, map);
    map.set(part2, part2Res);
  }

  for (let res1 of part1Res) {
    for (let res2 of part2Res) {
      switch (charI) {
        case "+":
          res.push(res1 + res2);
```

```
            break;
          case "-":
            res.push(res1 - res2);
            break;
          case "*":
            res.push(res1 * res2);
            break;
          default:
            break;
        }
      }
    }
  }
  if (res.length === 0) {
    res.push(+input);
  }
  return result;
};

export default (input) => {
  const map = new Map();
  return computeWithDP(input, map);
```

```
};
```

实现3

```
/**
 * @param {string} input
 * @return {number[]}
 */
// Runtime: 72 ms, faster than 94.74% of JavaScript online submissions for Different Ways to Add Parentheses.
// Memory Usage: 39 MB, less than 86.84% of JavaScript online submissions for Different Ways to Add Parentheses.
const diffWaysToCompute = (input, map) => {
  if (input.length === 0 || !input) return [];
  const oprs = [];
  const nums = [];
  let begin = 0;
  // 计算出有input中有多少个操作符和多少个数字
  for (let i = 0; i < input.length; i++) {
    const charI = input.charAt(i);
    if (charI == "+" || charI == "-" || charI == "*") {
      oprs.push(charI);
      nums.push(+input.substring(begin, i));
```

```
      begin = i + 1;
    }
  }
  // 把最后一个数字加入
  nums.push(+input.substring(begin));

  const numsLen = nums.length;
  const dp = [];
  // dp[i][j]表示input中数字nums[i]到数字nums[j]的之间的结果
  for (let i = 0; i < numsLen; i++) {
    dp[i] = [];
    for (let j = 0; j < numsLen; j++) {
      dp[i][j] = [];
    }
  }

  // 遍历已经发现的所有数字
  for (let i = 0; i < numsLen; i++) {
    // 计算0到i的结果
    for (let j = i; j >= 0; j--) {
      // 如果只是一个数字，直接加入
      if (i === j) {
        dp[j][i].push(nums[i]);
```

```
    } else {
      // dp[j][i] 等于dp[j][k]和dp[k+1][i]相乘
      for (let k = j; k < i; k += 1) {
        for (let left of dp[j][k]) {
          for (let right of dp[k + 1][i]) {
            let val = 0;
            switch (oprs[k]) {
              case "+":
                val = left + right;
                break;
              case "-":
                val = left - right;
                break;
              case "*":
                val = left * right;
                break;
            }
            dp[j][i].push(val);
          }
        }
      }
    }
  }
```

```
  }
  if (dp[0][numsLen - 1].length === 0) {
    dp[0][numsLen - 1].push(+input);
  }
  return dp[0][numsLen - 1];
};

export default diffWaysToCompute;
```

932. 漂亮数组

题目描述

对于某些固定的 N，如果数组 A 是整数 1, 2, ..., N 组成的排列，使得：

对于每个 i < j，都不存在 k 满足 i < k < j 使得 A[k] * 2 = A[i] + A[j]。

那么数组 A 是漂亮数组。

给定 N，返回任意漂亮数组 A（保证存在一个）。

例子1

Input: 4

output: [2,1,4,3]

例子2

Input: 5

output: [3,1,2,5,4]

提示：
1 <= N <= 1000

思考

1 题目本身感觉如果第一次接触到，估计应该不会想到如何拆分和如何合并。

这里比较巧妙的是把n给拆分成偶数和奇数，这样如果从偶数集合和奇数集合里边各自取一个数的时候，则肯定不会出现A[k] * 2 = A[i]+A[j], 因为偶数和奇数相加肯定是奇数，而A[k] * 2肯定是偶数，所以永远不会相等。

这里假设有一个漂亮数组

1.1 删除

如果删除漂亮数组中的任何一个值，到最后肯定还是漂亮数组

1.2 漂亮数组加或者减去同一个值

因为我们有 A[k] * 2 != A[i] + A[j],
(A[k] + x) * 2 = A[k] * 2 + 2x != A[i] + A[j] + 2x = (A[i] + x) + (A[j] + x)

比如: [1,3,2] + 1 = [2,4,3].

1.3 漂亮数组相乘或者相除同一个值

因为我们有 A[k] * 2 != A[i] + A[j],
对于任何x!=0
(A[k] * x) * 2 = A[k] * 2 * x != (A[i] + A[j]) * x = (A[i] * x) + (A[j] * x)

比如: [1,3,2] * 2 = [2,6,4]

通过上面可以得出，如果一个数组是漂亮数组，可以删除或者加上同一个数或者相乘同一个数都还是漂亮数组

现在假设有个漂亮数组A

A1 = A * 2 - 1
A2 = A * 2
B = A1 + A2

比如

```
A = [2, 1, 4, 5, 3]
A1 = [3, 1, 7, 9, 5]
A2 = [4, 2, 8, 10, 6]
B = A1 + A2 = [3, 1, 7, 9, 5, 4, 2, 8, 10, 6]
```

A1肯定是漂亮数组，如果从A1中任意选择一个，那么从A1中选择一个和从A2中选择一个，肯定还是不存在
A[k] * 2 === A[i] + A[j]

分治参考实现1

自顶向下参考实现2

字底向上参考实现3

实现1

```
/**
 * @param {number} N
 * @return {number[]}
 */

// Runtime: 84 ms, faster than 81.25% of JavaScript online submissions for Beautiful Array.
// Memory Usage: 40.1 MB, less than 87.50% of JavaScript online submissions for Beautiful Array.
export default (N) => {
  let res = [];
  res.push(1);
  while (res.length < N) {
    const tmp = [];
    for (let odd of res) {
      if (odd * 2 - 1 <= N) {
        tmp.push(odd * 2 - 1);
      }
```

```
    }
    for (let even of res) {
      if (even * 2 <= N) {
        tmp.push(even * 2);
      }
    }
    res = tmp;
  }
  return res;
};
```

时间复杂度O（n），空间复杂度O(n)

实现2

```
/**
 * @param {number} N
 * @return {number[]}
 */
// Runtime: 84 ms, faster than 81.25% of JavaScript online submissions for Beautiful Array.
// Memory Usage: 40.6 MB, less than 43.75% of JavaScript online submissions for Beautiful Array.
const getBeautifulArray = (N, map) => {
```

```
  if (N === 1) {
    map.set(1, [1]);
    return [1];
  }
  if (N === 2) {
    map.set(2, [1, 2]);
    return [1, 2];
  }
  const left = Math.floor(N / 2);
  let leftArr = [];
  if (map.has(left)) {
    leftArr = map.get(left);
  } else {
    leftArr = getBeautifulArray(left, map);
    map.set(left, leftArr);
  }
  let rightArr = [];
  const right = N - left;
  if (map.has(right)) {
    rightArr = map.get(right);
  } else {
    rightArr = getBeautifulArray(right, map);
    map.set(right, rightArr);
```

```
  }
  leftArr = leftArr.map((x) => x * 2);
  rightArr = rightArr.map((x) => x * 2 - 1);
  const temp = [...leftArr, ...rightArr];
  map.set(N, temp);
  return temp;
};

export default (N) => {
  if (N === 1) return [1];
  const map = new Map();
  return getBeautifulArray(N, map);
};
```

实现3

```
/**
 * @param {number} N
 * @return {number[]}
 */
// Runtime: 156 ms, faster than 6.25% of JavaScript online submissions for Beautiful Array.
```

```
// Memory Usage: 57.3 MB, less than 6.25% of JavaScript online submissions for Beautiful Array.
export default (N) => {
  if (N === 1) return [1];
  const dp = [];
  dp[1] = [1];
  dp[2] = [1, 2];

  for (let i = 3; i <= N; i++) {
    const left = Math.floor(i / 2);
    const leftArr = dp[left].map((x) => x * 2);
    const right = i - left;
    const rightArr = dp[right].map((x) => x * 2 - 1);
    dp[i] = [...leftArr, ...rightArr];
  }
  return dp[N];
};
```

312. 戳气球

题目描述

有 n 个气球，编号为0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。如果你戳破气球 i ，就可以获得 nums[left] * nums[i] * nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明：
1 你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。
2 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

例子1

Input: [3,1,5,8]

output: 167

解释： nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

例子2

Input: nums = [1,5]

output: 10

思考

1 题目本身还是有点难度的，如果第一次接触，考虑下直接看题解就可以

这里比较难的是如何把数组长度为n，拆分成两个数组。

因为这里需要我们每次戳破一个气球，然后得到相连的三个数组的硬币

如果采用拆分和组合的方式，可以按照下面考虑

1.1 首先考虑一下扎[0,nums.length-1] 这个区间可以得到的最大硬币，也就是最后结果

1.2 因为我们需要每次扎一个气球，假设我们从正面考虑，第一次先扎第i位置的气球，这个时候可以拆分成[0, i-1]，nums[i],[i+1, nums.length-1] 这三个区间，但是很明显这里不能这么划分，因为如果按照这样划分，那么nums[i]左边和右边相连的数字是不确定的。这个时候如果逆向思考一下，如果是最后扎i位置的气球呢，很明显这时候也拆分成了三个区间

[0, i-1]，nums[i],[i+1, nums.length-1]，但是这时候因为是最后扎i位置的气球，可以看到nums[i]左边和右边的位置是确定的

1.3 剩下的就是定义区间的结果，假设dp[i][j]表示从i到j扎气球的可以得到的最大硬币数目（不包括i和j），但是这里可以注意到提示
你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。

这时候我们可以重新组合下nums成为copynums，假设nums的长度是len，那么copynums的长度是len+2
copynums[0]=1,copynums[len+1]=1,copynums[1...len] = nums[0...len-1]

根据前面dp[i][j]的定义，那么我们的结果就变成了求copynums的dp[0][len+1]了

1.4 最后就剩下如何拆分再如何合并了，因为要求dp[0][len+1]的值，因为在copynums从1到len任何一个位置都可能是最后一次扎破，所以肯定得循环从1到len。假设其中一次我们选择扎破i位置的气球，那么此时的dp[0][len+1]等于什么呢?

此时dp[0][len+1] = dp[0][i]+dp[i][len+1]+copynums[0]* copynums[i]* copynums[len+1]

因为i位置的气球是最后扎破的，那么可以肯定此时的得到的最大硬币是copynums[0]* copynums[i]* copynums[len+1]

此时左边的最大硬币数是dp[0][i]，因为可以选择从1到i-1的任何一个气球最后扎破，所以i位置的气球可以作为dp[0][i]的任何一个位置的最右

边。

右边同理

因为i的范围是从1到len，所以最后的结果是要取这些所有可能的最大值

比如copynums = [1,3,1,5,8,1]的时候，选择i=3的时候，
dp[0][5] = dp[0][3]+dp[3][5]+1 * 1 * 5

可以按照dp[i][j]的定义，看看是不是已经包括了所有的可能情况

分治参考实现1

从下向上参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */

const burst = (memo, nums, left, right) => {
  if (left + 1 === right) return 0;
  if (memo[left][right] > 0) return memo[left][right];
  let ans = 0;
```

```
  for (let i = left + 1; i < right; i++) {
    ans = Math.max(ans, nums[left] * nums[i] * nums[right] + burst(memo, nums, left, i) + burst(memo, nums, i, right));
  }
  memo[left][right] = ans;
  return ans;
};

// Runtime: 448 ms, faster than 5.06% of JavaScript online submissions for Burst Balloons.
// Memory Usage: 40.1 MB, less than 89.40% of JavaScript online submissions for Burst Balloons.
export default (nums) => {
  const len = nums.length;
  const copyNums = new Array(len + 2);
  copyNums[0] = 1;
  copyNums[len + 1] = 1;
  for (let i = 0; i < len; i++) {
    copyNums[i + 1] = nums[i];
  }

  const memo = [];
  for (let i = 0; i < len + 2; i++) {
    memo[i] = new Array(len + 2).fill(0);
  }
  // console.log(memo);
```

```
  return burst(memo, copyNums, 0, len + 1);
};
```

实现2

```
const n = nums.length;
  nums = [1, ...nums, 1];
  const len = nums.length;
  const dp = [];
  for (let i = 0; i < len; i++) {
    dp[i] = new Array(len).fill(0);
  }
  // dp[i][j] 表示i到j-1的最大值
  // i2jLen 表示dp[i][j]表示的数组长度
  for (let i2jLen = 1; i2jLen <= n; i2jLen++) {
    for (let i = 0, j = i2jLen + 1; j <= len - 1; i++, j++) {
      for (let k = i + 1; k < j; k++) {
        dp[i][j] = Math.max(dp[i][j], dp[i][k] + nums[i] * nums[k] * nums[j] + dp[k][j]);
      }
    }
```

```
    }
    return dp[0][len - 1];
```

第9章数学算法

数学算法

涉及到数学的算法，基本都有固定的算法公式，如果知道就知道，不知道也不好想出来，比如求质数，进制转换等

这些题目首先你得了解这些数学知识，然后才能做，不然根本没有任何思绪。

9.1 最小公倍数

最小公倍数就是能够整除两个数的最小数

1 首先求两个数的最大公因数，然后再把两个数相乘再除以最大公因数就可以了，也就是辗转相除法。

求最大公因数代码如下：

```
const gcd = (a, b) => {
  return b === 0 ? a : gcd(b, a % b);
};
```

求最大公倍数代码如下：

```
const gcd = (a, b) => {
  return b === 0 ? a : gcd(b, a % b);
};

export default (a, b) => {
  return Math.floor((a * b) / gcd(a, b));
};
```

9.2 质数

质数又称素数，指的是指在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的自然 数。值得注意的是，每一个数都可以分解成质数的乘积。

204 计算质数的数量

题目描述

给定一个数字 n，求小于 n 的质数的个数。

例子1

Input: n = 10

output: 4

解释：
2, 3, 5, 7 是小于10的质数

例子2

Input: 0

output: 0

例子3

Input: 1

output: 0

思考

1 直接使用暴力解法就可以

2 使用一些技巧，这里涉及很多数学特性，不过也很简单，看下代码就可以了

参考实现1

参考实现2

实现1

```
/**
 * @param {number} n
```

```
 * @return {number}
 */

// Runtime: 124 ms, faster than 92.38% of JavaScript online submissions for Count Primes.
// Memory Usage: 52.1 MB, less than 45.59% of JavaScript online submissions for Count Primes.
export default (n) => {
  const notPrime = new Array(n).fill(0);
  let count = 0;
  for (let i = 2; i < n; i++) {
    if (notPrime[i] === 0) {
      count++;
      // 找到另外一个因子，如果存在另外一个因子，则不是质数
      for (let j = 2; i * j < n; j++) {
        notPrime[i * j] = 1;
      }
    }
  }

  return count;
};
```

实现2

```javascript
/**
 * @param {number} n
 * @return {number}
 */

// Runtime: 120 ms, faster than 94.49% of JavaScript online submissions for Count Primes.
// Memory Usage: 52.1 MB, less than 45.59% of JavaScript online submissions for Count Primes.
export default (n) => {
  const excludes = new Array(n).fill(false);

  if (n < 3) return 0;

  // 最大的素数,因为偶数都不是质数
  let maxCount = Math.floor(n / 2);

  // 如果奇数相乘大于n，所以肯定不存在
  for (let i = 3; i * i < n; i += 2) {
    //说明已经排除了
    if (excludes[i]) {
      continue;
    }
    // 比如3，那么9，15，21肯定不是质数
```

```
      for (let j = i * i; j < n; j += i * 2) {
        if (!excludes[j]) {
          excludes[j] = true;
          maxCount--;
        }
      }
    }
    return maxCount;
};
```

504. 七进制数

题目描述

给定一个整数，将其转化为7进制，并以字符串形式输出。

例子1

Input: 100

output: "202"

例子2

Input: -7

output: 10

思考

1 固定的转换套路，很简单

参考实现1

实现1

```
/**
 * @param {number} num
 * @return {string}
 */

// Runtime: 76 ms, faster than 90.91% of JavaScript online submissions for Base 7.
// Memory Usage: 39 MB, less than 13.29% of JavaScript online submissions for Base 7.
export default (num) => {
  if (num == 0) return "0";
  const is_negative = num < 0;
  if (is_negative) {
    num = -num;
  }
  let res = "";
  while (num > 0) {
```

```
    const a = Math.floor(num / 7);
    const b = num % 7;
    res = b + res;
    num = a;
  }
  return is_negative ? "-" + res : res;
};
```

时间复杂度O（n）空间复杂度O（1）

168. Excel表列名称

题目描述

给定一个正整数，返回它在 Excel 表中相对应的列名称。

例如:

```
    1 -> A
    2 -> B
    3 -> C
    ...
    26 -> Z
```

```
27 -> AA
28 -> AB
```

例子1

Input: 1

output: "A"

例子2

Input: 28

output: "AB"

例子3

Input: 701

output: "ZY"

思考

1 这题和上面差不多，也是类似数制转换的问题，也就是转换成26进制，但是这里会涉及到0的处理，所以得想办法处理0的问题

这里有一种是使用数组处理0，因为如果发现是0的话，必须上高位借1。

另外需要主要的是js的相除会有可能是小数，所以得向下取整

参考实现1

2 然后还有一种使用递归的方法，该方法有一个奇妙的地方就是为了避免处理0，是首先把n减去1，然后再处理

参考实现2

3 这里是把递归改成不递归

参考实现3

实现1

```
/**
 * @param {number} n
 * @return {string}
 */

const map = {
  0: "0",
  1: "A",
  2: "B",
  3: "C",
  4: "D",
  5: "E",
  6: "F",
  7: "G",
  8: "H",
```

```
  9: "I",
  10: "J",
  11: "K",
  12: "L",
  13: "M",
  14: "N",
  15: "O",
  16: "P",
  17: "Q",
  18: "R",
  19: "S",
  20: "T",
  21: "U",
  22: "V",
  23: "W",
  24: "X",
  25: "Y",
  26: "Z",
};
// Runtime: 76 ms, faster than 70.41% of JavaScript online submissions for Excel Sheet Column Title.
// Memory Usage: 38.6 MB, less than 23.60% of JavaScript online submissions for Excel Sheet Column Title.
export default (n) => {
  if (n <= 26) return map["" + n];
```

```
let res = [];
while (n > 26) {
  let a = Math.floor(n / 26);
  let b = n % 26;
  res.unshift(b);
  n = a;
}
res.unshift(n);
for (let i = 1; i < res.length; i++) {
  if (res[i] === 0) {
    let j = i - 1;
    res[i] = 26;
    res[j]--;
    while (res[j] === 0 && j >= 1 && res[j - 1] > 0) {
      res[j] = 26;
      res[j - 1]--;
      j--;
    }
  }
}
let k = 0;
while (res[k] === 0) {
  res.shift();
```

```
    k++;
  }
  console.log(res);
  const temp = res
    .map((item) => {
      return map["" + item];
    })
    .join("");

  return temp;
};
```

实现2

```
/**
 * @param {number} n
 * @return {string}
 */

const convertToTitle = (n) => {
  if (n === 0) return "";
  return convertToTitle(Math.floor(--n / 26)) + String.fromCharCode("A".charCodeAt() + (n % 26));
```

```
};
export default convertToTitle;
```

实现3

```
/**
 * @param {number} n
 * @return {string}
 */
// Runtime: 72 ms, faster than 88.06% of JavaScript online submissions for Excel Sheet Column Title.
// Memory Usage: 38.2 MB, less than 73.13% of JavaScript online submissions for Excel Sheet Column Title.
export default (n) => {
  if (n === 0) return "";
  let res = "";
  while (n > 0) {
    --n;
    const a = Math.floor(n / 26);
    const b = n % 26;
    res = String.fromCharCode("A".charCodeAt() + b) + res;
    n = a;
  }
```

```
    return res;
};
```

172. 阶乘后的零

题目描述

给定一个整数 n，返回 n! 结果尾数中零的数量。

例子1

Input: 3

output: 0

例子2

Input: 5

output: 1

解释：5！=120

思考

1 刚开始的时候想直接使用暴力解法，但是发现如果输入30之后，就会用科学计数法来表示，基本上就得不到正确的结果了

2 另外一种思路是可以发现10！= 362800，这时候我们要求出最后有几个0，很明显就是看362800可以除以10得到正整数。

此时已经发现先得到10的结果再求多少个0已经得不到正确的结果了，可以想一下10！还可以怎么表示?

10！ = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10

那么10！/ 10 就可以表示成1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 / 10

同时除以10可以看成除以2 * 5，同时10！也可以转换成
1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 2 * 5,那么公式就变成了1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 2 * 5/ 2 * 5

此时我们基本上就可以发现其实就是求10！里边含有多少对2 * 5了

那么如果求呢?

这里可以发现10！里边肯定含有的2比5多，因为任何一个偶数都含有2，但是不一定含有5

所以我们只需要求5出现了多少次就可以了

那么怎么求5出现了多少次呢?

我们可以分成几步来求
第一步求含有1个5的数量，比如在10！中那就是5 和 10
第二步求含有2个5的数量，比如在30！中那就是25
依次类推

参考实现1

实现1

```
/**
```

```
 * @param {number} n
 * @return {number}
 */
// Runtime: 88 ms, faster than 68.48% of JavaScript online submissions for Factorial Trailing Zeroes.
// Memory Usage: 39.4 MB, less than 43.21% of JavaScript online submissions for Factorial Trailing Zeroes.
const trailingZeroes = (n) => {
  if (n <= 4) return 0;
  let res = 0;
  while (n > 0) {
    n = Math.floor(n / 5);
    res += n;
  }
  return res;
};
export default trailingZeroes;
```

时间复杂度O（lgn）空间复杂度O（1）

415. 字符串相加

题目描述

给定两个字符串形式的非负整数 num1 和num2 ， 计算它们的和。

提示：

```
1 num1 和num2 的长度都小于 5100
2 num1 和num2 都只包含数字 0-9
3 num1 和num2 都不包含任何前导零
4 你不能使用任何內建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式
```

思考

1 题目比较简单，直接使用charCodeAt获取字符的值就可以

参考实现1

实现1

```
/**
 * @param {string} num1
 * @param {string} num2
 * @return {string}
 */
// Runtime: 76 ms, faster than 98.86% of JavaScript online submissions for Add Strings.
// Memory Usage: 40.9 MB, less than 47.14% of JavaScript online submissions for Add Strings.
export default (num1, num2) => {
```

```
  const len1 = num1.length;
  const len2 = num2.length;
  let len = len1;
  const maxLen = Math.max(len1, len2);
  if (len2 < len1) {
    const temp = num2;
    num2 = num1;
    num1 = temp;
    len = len2;
  }
  let flag = 0;
  const res = [];
  for (let i = len - 1; i >= 0; i--) {
    const sum = num1.charCodeAt(i) + num2.charCodeAt(maxLen - len + i) - 96 + flag;
    if (sum >= 10) {
      res.unshift(sum % 10);
      flag = 1;
    } else {
      res.unshift(sum);
      flag = 0;
    }
  }
  if (maxLen > len) {
```

```
    for (let i = maxLen - 1 - len; i >= 0; i--) {
      const sum = num2.charCodeAt(i) - 48 + flag;
      if (sum >= 10) {
        res.unshift(sum % 10);
        flag = 1;
      } else {
        res.unshift(sum);
        flag = 0;
      }
    }
  }

  if (flag === 1) {
    res.unshift(1);
  }
  return res.join("");
};
```

时间复杂度O（Math.max(len1,len2)）
空间复杂度O（len2）

67. 二进制求和

题目描述

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 非空 字符串且只包含数字 1 和 0。

提示：

```
1 每个字符串仅由字符 '0' 或 '1' 组成。
2 1 <= a.length, b.length <= 10^4
3 字符串如果不是 "0" ，就都不含前导零。
```

例子1
input：a = "11", b = "1"
output："100"

例子2
input：a = "1010", b = "1011"
output："10101"

思考

1 和415题目一样的逻辑

参考实现1

实现1

```
/**
 * @param {string} a
 * @param {string} b
 * @return {string}
 */
// Runtime: 92 ms, faster than 53.83% of JavaScript online submissions for Add Binary.
// Memory Usage: 40.4 MB, less than 66.69% of JavaScript online submissions for Add Binary.
export default (a, b) => {
  const len1 = a.length;
  const len2 = b.length;
  let len = len1;
  const maxLen = Math.max(len1, len2);
  if (len2 < len1) {
    const temp = b;
    b = a;
    a = temp;
    len = len2;
  }
  let flag = 0;
  const res = [];
  for (let i = len - 1; i >= 0; i--) {
```

```
    const sum = a.charCodeAt(i) + b.charCodeAt(maxLen - len + i) - 96 + flag;
    if (sum === 2) {
      res.unshift(0);
      flag = 1;
    } else if (sum === 3) {
      res.unshift(1);
      flag = 1;
    } else {
      res.unshift(sum);
      flag = 0;
    }
  }
  if (maxLen > len) {
    for (let i = maxLen - 1 - len; i >= 0; i--) {
      const sum = b.charCodeAt(i) - 48 + flag;
      if (sum === 2) {
        res.unshift(0);
        flag = 1;
      } else if (sum === 3) {
        res.unshift(1);
        flag = 1;
      } else {
        res.unshift(sum);
```

```
        flag = 0;
      }
    }
  }

  if (flag === 1) {
    res.unshift(1);
  }
  return res.join("");
};
```

时间复杂度O（Math.max(len1,len2)）
空间复杂度O（len2）

9.5 随机与取样

384. 打乱数组和恢复

题目描述

给定一个数组，要求实现两个指令函数。第一个函数“shuffle”可以随机打乱这个数组，第 二个函数“reset”可以恢复原来的顺序。

例子1
input：nums = [1,2,3], actions: ["shuffle","shuffle","reset"]
output：[[2,1,3],[3,2,1],[1,2,3]]

思考

1 题目很简单，就是一个简单的洗牌算法，不过面试中经常会被问到。

参考实现1

实现1

```
/**
 * @param {number[]} nums
 */

//  Runtime: 236 ms, faster than 67.68% of JavaScript online submissions for Shuffle an Array.
//  Memory Usage: 52.4 MB, less than 55.56% of JavaScript online submissions for Shuffle an Array.
var Solution = function (nums) {
  this.nums = nums || [];
};

/**
 * Resets the array to its original configuration and return it.
 * @return {number[]}
```

```
 */
Solution.prototype.reset = function () {
  return this.nums;
};

/**
 * Returns a random shuffling of the array.
 * @return {number[]}
 */
Solution.prototype.shuffle = function () {
  const tempNums = [...this.nums];
  const len = tempNums.length;
  for (let i = 0; i < tempNums.length; i++) {
    const index = Math.floor(Math.random() * (len - i) + i);
    // console.log(index)
    const temp = tempNums[index];
    tempNums[index] = tempNums[i];
    tempNums[i] = temp;
  }
  return tempNums;
};

/**
```

```
 * Your Solution object will be instantiated and called as such:
 * var obj = new Solution(nums)
 * var param_1 = obj.reset()
 * var param_2 = obj.shuffle()
 */
```

时间复杂度O（n）
空间复杂度O（n）

528. 按权重随机选择

题目描述

给定一个正整数数组 w ， 其中 w[i] 代表下标 i 的权重（下标从 0 开始），请写一个函数 pickIndex ， 它可以随机地获取下标 i，选取下标 i 的概率与 w[i] 成正比。

例如，对于 w = [1, 3]，挑选下标 0 的概率为 1 / (1 + 3) = 0.25 （即，25%），而选取下标 1 的概率为 3 / (1 + 3) = 0.75（即，75%）。

也就是说，选取下标 i 的概率为 w[i] / sum(w) 。

例子1
input：["Solution","pickIndex"]

[[[1]],[]]

output：[null,0]
解释：Solution solution = new Solution([1]);
solution.pickIndex(); // 返回 0，因为数组中只有一个元素，所以唯一的选择是返回下标 0。

例子2
input：["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]

[[[1,3]],[],[],[],[],[]]
output：[null,1,1,1,1,0]
解释：Solution solution = new Solution([1, 3]);
solution.pickIndex(); // 返回 1，返回下标 1，返回该下标概率为 3/4 。
solution.pickIndex(); // 返回 1
solution.pickIndex(); // 返回 1
solution.pickIndex(); // 返回 1
solution.pickIndex(); // 返回 0，返回下标 0，返回该下标概率为 1/4 。

由于这是一个随机问题，允许多个答案，因此下列输出都可以被认为是正确的:

[null,1,1,1,1,0]

[null,1,1,1,1,1]

[null,1,1,1,0,0]

[null,1,1,1,0,1]

[null,1,0,1,0,0]

.....
诸若此类。

思考

1 刚开始想按照次数，比如当输入[1,3]的时候，如果选择4次，一次输出1，一次输出3，但是发现这样不行，这样就不是随机的了

后来看了下题解，其实道理很简单，就是使用前缀和，搞成一个类似于区间的

然后使用随机数，看落在那个区间，刚好符合概率

参考实现1

还可以使用二分法查找，参考实现2

实现1

```
/**
 * @param {number[]} w
 */
var Solution = function (w) {
  const len = w.length;
  this.chances = new Array(len).fill(0);
  const sum = w.reduce((a, b) => a + b);
  for (let i = 0; i < w.length; i++) {
```

```
    w[i] += i === 0 ? 0 : w[i - 1];
    this.chances[i] = w[i] / sum;
  }
};

/**
 * @return {number}
 */
Solution.prototype.pickIndex = function () {
  if (this.chances.length === 1) {
    return 0;
  }
  const random = Math.random().toFixed(2);
  // console.log(this.chances);
  for (let i = 0; i < this.chances.length; i++) {
    if (random <= this.chances[i]) {
      return i;
    }
  }
  return this.chances.length - 1;
};

/**
```

```
 * Your Solution object will be instantiated and called as such:
 * var obj = new Solution(w)
 * var param_1 = obj.pickIndex()
 */
export default Solution;
```

实现2

```
/**
 * @param {number[]} w
 */
var Solution = function (w) {
  const len = w.length;
  this.chances = new Array(len).fill(0);
  const sum = w.reduce((a, b) => a + b);
  for (let i = 0; i < w.length; i++) {
    w[i] += i === 0 ? 0 : w[i - 1];
    this.chances[i] = w[i] / sum;
  }
};
```

```
/**
 * @return {number}
 */
Solution.prototype.pickIndex = function () {
  if (this.chances.length === 1) {
    return 0;
  }
  const random = Math.random().toFixed(2);
  let low = 0;
  let high = this.chances.length - 1;
  while (low <= high) {
    const mid = Math.floor(low + (high - low) / 2);
    if (this.chances[mid] >= random) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }
  return low;
};

/**
 * Your Solution object will be instantiated and called as such:
```

```
 * var obj = new Solution(w)
 * var param_1 = obj.pickIndex()
 */
export default Solution;
```

382. 链表随机节点

题目描述

给定一个单链表，随机选择链表的一个节点，并返回相应的节点值。保证每个节点被选的概率一样。

进阶:
如果链表十分大且长度未知，如何解决这个问题？你能否使用常数级空间复杂度实现？

例子1

```
// 初始化一个单链表 [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);

// getRandom()方法应随机返回1,2,3中的一个，保证每个元素被返回的概率相等。
```

思考

1 基本的可以使用数组存储整个链表，然后直接随机就可以了

2 后来看了下这里是典型的水库算法

水库算法比较简单

当遇到第一个节点的时候，我们选择第一个节点

当遇到第二个节点的时候，这个时候要么替换第一个节点，要么不替换第一个节点，如果替换第一个节点的时候，也是1/2

当遇到第三个节点的时候，如果替换那么概率是1/3 * 1 = 1/3,如果不替换，则保持原来的。

以此类推，假如我们遇到到第i个节点的时候，此时如果替换的话，概率计算可以分为两步的概率，第一步是先在前i个节点中选择1个，概率是1/i,第二步是选择替换，也就是1/1,此时概率就是1/i.

所以我们可以在遇到第i个节点的时候，如果概率小于等于1/i,则替换就可以了。

数组存储链表参考实现1
水库算法参考实现2

实现1

```
/**
 * Definition for singly-linked list.
```

```
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param head The linked list's head.
        Note that the head is guaranteed to be not null, so it contains at least one node.
 * @param {ListNode} head
 */
// Runtime: 308 ms, faster than 5.54% of JavaScript online submissions for Linked List Random Node.
// Memory Usage: 50 MB, less than 5.21% of JavaScript online submissions for Linked List Random Node.
var Solution = function (head) {
  let p = head;
  this.nums = [];
  while (p.next) {
    this.nums.push(p.next.val);
    p = p.next;
  }
};

/**
 * Returns a random node's value.
```

```
 * @return {number}
 */
Solution.prototype.getRandom = function () {
  const len = this.nums.length;
  const i = Math.floor(Math.random() * (len + 1));
  return this.nums[i];
};

/**
 * Your Solution object will be instantiated and called as such:
 * var obj = new Solution(head)
 * var param_1 = obj.getRandom()
 */
```

实现2

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
```

```
 * }
 */
/**
 * @param head The linked list's head.
        Note that the head is guaranteed to be not null, so it contains at least one node.
 * @param {ListNode} head
 */
// Runtime: 308 ms, faster than 5.54% of JavaScript online submissions for Linked List Random Node.
// Memory Usage: 50 MB, less than 5.21% of JavaScript online submissions for Linked List Random Node.
var Solution = function (head) {
  // let p = head;
  // this.nums = [];
  // while (p.next) {
  //   this.nums.push(p.next.val);
  //   p = p.next;
  // }
  this.head = head;
};

/**
 * Returns a random node's value.
 * @return {number}
 */
```

```javascript
// Runtime: 120 ms, faster than 65.35% of JavaScript online submissions for Linked List Random Node.
// Memory Usage: 46.3 MB, less than 19.25% of JavaScript online submissions for Linked List Random Node.
Solution.prototype.getRandom = function () {
  // const len = this.nums.length;
  // const i = Math.floor(Math.random() * (len + 1));
  // return this.nums[i];
  let p = this.head;
  let res = p.val;
  for (let i = 1; p != null; i++) {
    p = p.next;
    if (Math.random() <= 1 / (i + 1)) {
      res = p != null ? p.val : res;
    }
  }
  return res;
};

/**
 * Your Solution object will be instantiated and called as such:
 * var obj = new Solution(head)
 * var param_1 = obj.getRandom()
 */
```

238. 除自身以外数组的乘积

题目描述

给你一个长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output ，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

进阶:
你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

例子1
input: [1,2,3,4]
output: [24,12,8,6]

思考

1 题目比较简单，首先想到了使用一个前置数组保存i之前的结果，一个后置数组保持i之后的数组的乘积，然后相乘就可以了

2 如果想使用常数时间，可以看看实现1有哪些需要节省的，可以发现可以把前置数组用结果数组表示

参考实现1
参考实现2

实现1

```
/**
 * @param {number[]} nums
```

```
 * @return {number[]}
 */
// Runtime: 692 ms, faster than 13.05% of JavaScript online submissions for Product of Array Except Self.
// Memory Usage: 51.3 MB, less than 8.63% of JavaScript online submissions for Product of Array Except Self.
export default (nums) => {
  const preNums = [1];
  const afterNums = [1];
  for (let i = 1; i < nums.length; i++) {
    preNums[i] = preNums[i - 1] * nums[i - 1];
  }
  for (let j = nums.length - 2, k = 1; j >= 0; j--, k++) {
    const temp = afterNums[afterNums.length - k] * nums[j + 1];
    afterNums.unshift(temp);
  }
  // console.log(preNums, afterNums);
  const res = [];
  for (let i = 0; i < nums.length; i++) {
    res[i] = preNums[i] * afterNums[i];
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
// Runtime: 112 ms, faster than 92.27% of JavaScript online submissions for Product of Array Except Self.
// Memory Usage: 49.7 MB, less than 54.84% of JavaScript online submissions for Product of Array Except Self.
export default (nums) => {
  const len = nums.length;
  const res = [];
  res[0] = 1;
  for (let i = 1; i < len; i++) {
    res[i] = res[i - 1] * nums[i - 1];
  }
  let right = 1;
  for (let i = len - 1; i >= 0; i--) {
    res[i] *= right;
    right *= nums[i];
  }
  return res;
};
```

169. 多数元素

题目描述

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

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

进阶:
尝试设计时间复杂度为 O(n)、空间复杂度为 O(1) 的算法解决此问题

例子1
input: [3,2,3]
output: 3

例子2
input: [2,2,1,1,1,2,2]
output: 2

思考

1 可以先排序，然后查找超过 ⌊ n/2 ⌋ 的元素

2 进阶是是使用Boyer-Moore Majority Vote算法，这个算法原理也很简单，就是发现数组中两个不同的元素直接删除，最后剩下的就是想要的结果。因为肯定存在多数元素，而多数元素的数量肯定大于 ⌊ n/2 ⌋ 的元素

参考实现1

参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 92 ms, faster than 31.11% of JavaScript online submissions for Majority Element.
// Memory Usage: 43.1 MB, less than 7.90% of JavaScript online submissions for Majority Element.
export default (nums) => {
  if (nums.length === 1) return nums[0];
  nums.sort((a, b) => a - b);
  const max = Math.floor(nums.length / 2);
  for (let i = 0; i < nums.length; i++) {
    const tempMax = i + max;
    if (nums[tempMax] === nums[i]) {
      return nums[i];
    }
    if (tempMax > nums.length) {
      break;
    }
  }
```

```
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 68 ms, faster than 99.88% of JavaScript online submissions for Majority Element.
// Memory Usage: 40.8 MB, less than 74.90% of JavaScript online submissions for Majority Element.
export default (nums) => {
  if (nums.length === 1) return nums[0];
  let major = nums[0];
  let count = 1;
  for (let i = 1; i < nums.length; i++) {
    if (count === 0) {
      count++;
      major = nums[i];
    } else if (major === nums[i]) {
      count++;
    } else {
```

```
      count--;
    }
  }
  return major;
};
```

462. 最少移动次数使数组元素相等 II

题目描述

给定一个非空整数数组，找到使所有数组元素相等所需的最小移动数，其中每次移动可将选定的一个元素加1或减1。 您可以假设数组的长度最多为10000。

例子1
input: [1,2,3]
output: 2
解释：只有两个动作是必要的（记得每一步仅可使其中一个元素加1或减1）：

[1,2,3] => [2,2,3] => [2,2,2]

思考

1 通过测试用例，首先想到可以先排序，然后小于中间元素的增加，超过中间元素的减少就可以了

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 84 ms, faster than 82.35% of JavaScript online submissions for Minimum Moves to Equal Array Elements II.
// Memory Usage: 40.6 MB, less than 64.71% of JavaScript online submissions for Minimum Moves to Equal Array Elements II.
export default (nums) => {
  if (nums.length === 1) return 0;
  nums.sort((a, b) => a - b);
  const len = nums.length;
  const mid = Math.floor(len / 2);
  const midNum = nums[mid];
  let res = 0;
  for (let i = 0; i < len; i++) {
    res += Math.abs(nums[i] - midNum);
  }
  return res;
};
```

时间复杂度O（n * lgn），空间复杂度O（1）

470. 用 Rand7() 实现 Rand10()

题目描述

已有方法 rand7 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 rand10 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 Math.random() 方法。

例子1
input: 1
output: [7]

例子2
input: 2
output: [8,4]

例子3
input: 3
output: [8,1,10]

提示

1 rand7 已定义。
2 传入参数: n 表示 rand10 的调用次数。

进阶
1 rand7()调用次数的 期望值 是多少 ？
2 你能否尽量少调用 rand7() ？

思考

1 这里使用随机数生成随机数，比较关键的是几点

1.1 首先是每个数出现的概率必须是随机的，比如rand10这里从1到10的概率必须都是1/10.
1.2

```
已知 rand_N() 可以等概率的生成[1, N]范围的随机数
那么：
(rand_X() - 1) × Y + rand_Y() ==> 可以等概率的生成[1, X * Y]范围的随机数
即实现了 rand_XY()
```

1.3 如果已经rand49，那么如何求rand10呢?

因为rand10是要求选择从1到10的概率都是1/10,现在已经知道rand49，也就是说使用rand49选择从1到49的概率都是1/49,那么rand10是什么呢?

可以想一下，假设选择2的概率是什么?

假如通过rand49函数执行一次就得到2，那么概率就是1/49

如果需要执行rand49函数两次得到2呢，那么概率就是第一次没有取到2，那么概率就是39/49,第二次取到2，那么概率是1/49,所以这次概率是39/49 * 1 / 49

同理第三次得到2，那么概率是 39/49 * 39/49 * 1 / 49

那么执行n次，总概率就是1/49 * ((1 * (1 - (39 / 49)^n)) / 1- (39/49) = 1 / 10

同理可以得到选择1到10的概率都是 1/10

到这里基本上就可以得到实现1

1.4 从1.3可以看到从rand49到rand10，所有大于10的都被舍弃了，那是不是有其他方法可以更大效率的利用这些从11到49的数呢?

这里有个规律是
要实现rand10()，就需要先实现rand_N()，并且保证N大于10且是10的倍数。这样再通过rand_N() % 10 + 1 就可以得到[1,10]范围的随机数了。

参考实现1

使用1.4 节省时间，可以参考实现2

参考： https://leetcode-cn.com/problems/implement-rand10-using-rand7/solution/cong-zui-ji-chu-de-jiang-qi-ru-he-zuo-dao-jun-yun-/

实现1

```
/**
 * The rand7() API is already defined for you.
 * var rand7 = function() {}
 * @return {number} a random integer in the range 1 to 7
```

```
 */
// Runtime: 124 ms, faster than 29.03% of JavaScript online submissions for Implement Rand10() Using Rand7().
// Memory Usage: 49.4 MB, less than 6.45% of JavaScript online submissions for Implement Rand10() Using Rand7().
const rand10 = () => {
  let a = rand7();
  let b = rand7();
  // 得到rand49
  let num = (a - 1) * 7 + b;

  if (num <= 10) return num;

  return rand10();
};
export default rand10;
```

时间复杂度O（1）

实现2

```
/**
 * The rand7() API is already defined for you.
 * var rand7 = function() {}
```

```
 * @return {number} a random integer in the range 1 to 7
 */
//  Runtime: 112 ms, faster than 96.77% of JavaScript online submissions for Implement Rand10() Using Rand7().
//  Memory Usage: 47.4 MB, less than 48.39% of JavaScript online submissions for Implement Rand10() Using Rand7().
const rand10 = () => {
  let a = rand7();
  let b = rand7();
  // 得到rand49
  let num = (a - 1) * 7 + b;

  if (num <= 40) return (num % 10) + 1;
  a = num - 40; // rand 9
  b = rand7();
  num = (a - 1) * 7 + b; // rand 63
  if (num <= 60) return (num % 10) + 1;

  a = num - 60; // rand 3
  b = rand7();
  num = (a - 1) * 7 + b; // rand 21
  if (num <= 20) return (num % 10) + 1;
  return rand10();
};
export default rand10;
```

202. 快乐数

题目描述

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是 无限循环 但始终变不到 1。如果 可以变为 1，那么这个数就是快乐数。

如果 n 是快乐数就返回 True ；不是，则返回 False 。

例子1
input: 19
output: true
解释：

```
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
```

例子2

input: 2
output: false

思考

1 使用一个Set存储已经找到了，如果发现循环了，返回fasle

实现1

```
/**
 * @param {number} n
 * @return {boolean}
 */
// Runtime: 96 ms, faster than 35.42% of JavaScript online submissions for Happy Number.
// Memory Usage: 39.9 MB, less than 71.78% of JavaScript online submissions for Happy Number.
export default (n) => {
  const inLoop = new Set();
  let squareSum;
  let remain;
  while (!inLoop.has(n)) {
    inLoop.add(n);
    squareSum = 0;
    while (n > 0) {
      remain = n % 10;
```

```
      squareSum += remain * remain;
      n = Math.floor(n / 10);
    }
    if (squareSum === 1) {
      return true;
    } else {
      n = squareSum;
    }
  }
  return false;
};
```

第10章最优效率的位运算

位运算

基本的位运算包括与或还有异或，左移，无符号右移等操作。

循环n&n-1可以计算n中有多少个1

n & 1 是否是1，可以看出n的最低位是否是1

n ^ n是0

n >> 1 相当于除以2

n << 1 想当于乘以2

记住这些常用的技巧，在位运算中可能会比较帮助，但是大多数是需要综合使用这些技巧。

461. 汉明距离

题目描述

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。

例子1

Input: 1, 4

output: 2

解释：

```
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑

上面的箭头指出了对应二进制位不同的位置
```

思考

1 可以首先异或两个数，则就得到了含有多少个1的数，然后计算这个数里边含有多少个1就可以了

参考实现1

实现1

```
/**
```

```
 * @param {number} x
 * @param {number} y
 * @return {number}
 */

export default (x, y) => {
  let z = x ^ y;
  let res = 0;
  while (z != 0) {
    res++;
    z &= z - 1;
  }
  return res;
};
```

190. 颠倒二进制位

题目描述

颠倒给定的 32 位无符号整数的二进制位。

例子1

Input: 00000010100101000001111010011100

output: 00111001011110000010100101000000

解释： 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596，
因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

例子2

Input: 11111111111111111111111111111101

output: 10111111111111111111111111111111

解释： 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293，
因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111 。

思考

1 这个虽然是easy题目，但是感觉还是不好想到，后来看了下，可以通过设置一个结果res，让res有符号左移和n有符号右移

最后因为需要获得无符号的，所以res还得需要无符号右移0位

参考实现1

实现1

```
/**
 * @param {number} n - a positive integer
```

```
 * @return {number} - a positive integer
 */
// Runtime: 84 ms, faster than 93.00% of JavaScript online submissions for Reverse Bits.
// Memory Usage: 40.2 MB, less than 76.19% of JavaScript online submissions for Reverse Bits.
export default (n) => {
  if (n === 0) return 0;
  let result = 0;
  for (let i = 0; i < 32; i++) {
    result <<= 1;
    result |= n & 1;
    n >>= 1;
  }
  return result >>> 0;
};
```

136 只出现一次的数字

题目描述

给定一个整数数组，这个数组里只有一个数次出现了一次，其余数字出现了两次，求这个只 出现一次的数字。

例子1

Input: [4,1,2,1,2]

output: 4

例子2

Input: nums = [4,1,2,1,2]

output: 4

思考

1 很明显是使用异或，因为两个相同的数异或会是0

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 84 ms, faster than 84.68% of JavaScript online submissions for Single Number.
// Memory Usage: 40.3 MB, less than 87.12% of JavaScript online submissions for Single Number.
export default (nums) => {
  let res = nums[0];
```

```
  for (let i = 1; i < nums.length; i++) {
    res ^= nums[i];
  }
  return res;
};
```

268. 丢失的数字

题目描述

给定一个包含 [0, n] 中 n 个数的数组 nums ，找出 [0, n] 这个范围内没有出现在数组中的那个数。

进阶：
你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?

例子1

Input: [3,0,1]

output: 2
解释：n = 3，因为有 3 个数字，所以所有的数字都在范围 [0,3] 内。2 是丢失的数字，因为它没有出现在 nums 中。

例子2

Input: nums = [0,1]

output: 2

解释：n = 2，因为有 2 个数字，所以所有的数字都在范围 [0,2] 内。2 是丢失的数字，因为它没有出现在 nums 中。

例子3

Input: nums = [9,6,4,2,3,5,7,0,1]

output: 2

解释：n = 9，因为有 9 个数字，所以所有的数字都在范围 [0,9] 内。8 是丢失的数字，因为它没有出现在 nums 中。

思考

1 如果想使用位运算，因为已经知道n^n === 0,所以想办法使用异或来解决。 但是首先是想到了先排序来解决

2 后来看了下，还可以借助i来解决，因为 n^0 === n 并且n^n === 0

参考实现1

参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
```

```
// Runtime: 88 ms, faster than 65.50% of JavaScript online submissions for Missing Number.
// Memory Usage: 41.2 MB, less than 47.38% of JavaScript online submissions for Missing Number.
export default (nums) => {
  const len = nums.length;
  nums.sort((a, b) => a - b);
  let res = 0;
  for (let i = 0; i < nums.length; i++) {
    if (res === nums[i]) {
      res++;
    } else {
      if (res ^ (nums[i] != 0)) {
        return res;
      }
    }
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number}
 */

// Runtime: 88 ms, faster than 65.50% of JavaScript online submissions for Missing Number.
// Memory Usage: 41.4 MB, less than 28.25% of JavaScript online submissions for Missing Number.

export default (nums) => {
  let res = nums.length;
  for (let i = 0; i < nums.length; i++) {
    res ^= i;
    res ^= nums[i];
  }
  return res;
};
```

260. 只出现一次的数字 III

题目描述

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

例子1

Input: [1,2,1,3,2,5]

output: [3,5]

思考

1 这里使用位运算，确实很难思考到，甚至看了结果仍然不是很好理解，刚刚开始是想因为n^n === 0,所以想通过两次遍历分别求出两个不同的数字，但是发现不行，因为两个不同的数字可能是相邻在一起，后来发现测试用例过不了。

后来看了下题解，这个确实很难想到

假设a和b是数组中的两个不同的数字
那么a^b肯定是不等于0，因为如果等于0，那么a===b
如果a^b不等于0，那么在二进制表示中肯定有有一位是1
那么我们假设m = a^b^数组中其它值, 那么我们可以通过m & ~(m - 1) 获取最低位是1的值
所以数组肯定分成两部分，一部分是该位置是1，一部分该位置是0，否则不可能是异或出来该位置等于1
所以该位置都是1的最后异或出来一个结果，该位置都是0的异或出来另外一个结果

参考实现1

实现1

```
/**
```

```
 * @param {number[]} nums
 * @return {number[]}
 */

// Runtime: 80 ms, faster than 92.98% of JavaScript online submissions for Single Number III.
// Memory Usage: 39.4 MB, less than 87.72% of JavaScript online submissions for Single Number III.
export default (nums) => {
  let res = [0, 0];
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum ^= nums[i];
  }

  sum = sum & ~(sum - 1);

  for (let i = 0; i < nums.length; i++) {
    if ((nums[i] & sum) === 0) {
      res[0] ^= nums[i];
    } else {
      res[1] ^= nums[i];
    }
  }
```

```
    return res;
};
```

时间复杂度O（n），空间复杂度O（1）

342. 4的幂

题目描述

给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true ；否则，返回 false 。

整数 n 是 4 的幂次方需满足：存在整数 x 使得 n == 4x

例子1

Input: n = 16

output: true

例子2

Input: n = 16

output: false

思考

1 这里是如果是4的平方，需要符合三个条件

```
// 如果是4的平方必须符合三种情况
// 1 n > 0
// 2 n是2的平方 也就是n & n-1
// 3 n里边所有1的位置都在奇数位上
```

参考实现1

实现1

```
/**
 * @param {number} n
 * @return {boolean}
 */
// Runtime: 84 ms, faster than 97.69% of JavaScript online submissions for Power of Four.
// Memory Usage: 39.9 MB, less than 48.52% of JavaScript online submissions for Power of Four.
export default (n) => {
  // 如果是4的平方必须符合三种情况
  // 1 n > 0
  // 2 n是2的平方 也就是n & n-1
  // 3 n里边所有1的位置都在奇数位上
```

```
  return n > 0 && (n & (n - 1)) === 0 && (n & 0b10101010101010101010101010101010) > 0;
};
```

318. 最大单词长度乘积

题目描述

给定一个字符串数组 words，找到 length(word[i]) * length(word[j]) 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。
br/>

例子1

Input: ["abcw","baz","foo","bar","xtfn","abcdef"]

output: 16

解释：这两个单词为 "abcw", "xtfn"。

例子2

Input: ["a","ab","abc","d","cd","bcd","abcd"]

output: 4

解释：这两个单词为 "ab", "cd"。

例子3

Input: ["a","aa","aaa","aaaa"]

output: 0

解释：不存在这样的两个单词。

思考

1 这里思路很简单，主要难点是如果确认两个字符串中不存在相同的字符
可以使用一个长度是26的数组，如果第一位是1，则表示字符串中“a”存在，

如果两个表示字符串的数组使用与操作等于0的时候，表示两个字符串不包含相同字符

参考实现1

实现1

```
/**
 * @param {string[]} words
 * @return {number}
 */
// Runtime: 100 ms, faster than 91.23% of JavaScript online submissions for Maximum Product of Word Lengths.
// Memory Usage: 42.2 MB, less than 85.96% of JavaScript online submissions for Maximum Product of Word Lengths.
```

```
export default (words) => {
  if (words.length < 1) return 0;
  const len = words.length;
  const value = new Array(len).fill(0);
  for (let i = 0; i < len; i++) {
    for (let j = 0; j < words[i].length; j++) {
      // 也就是使用26个数组表示字符串，如果数组第一位是1，则表示存在a
      value[i] |= 1 << (words[i].charCodeAt(j) - 97);
    }
  }
  let maxProduct = 0;
  for (let i = 0; i < len; i++)
    for (let j = i + 1; j < len; j++) {
      if ((value[i] & value[j]) === 0 && words[i].length * words[j].length > maxProduct)
        maxProduct = words[i].length * words[j].length;
    }
  return maxProduct;
};
```

338. 比特位计数

题目描述

给定一个非负整数 num。对于 0 ≤ i ≤ num 范围中的每个数字 i ，计算其二进制数中的 1 的数目并将它们作为数组返回。
br/>

例子1

Input: 2

output: [0,1,1]

例子2

Input: 5

output: [0,1,1,2,1,2]

进阶:

1 给出时间复杂度为O(n * sizeof(integer))的解答非常容易。但你可以在线性时间O(n)内用一趟扫描做到吗?

2 要求算法的空间复杂度为O(n)

3 你能进一步完善解法吗？要求在C++或任何其他语言中不使用任何内置函数（如 C++ 中的 __builtin_popcount） 来执行此操作。

思考

1 O(n * sizeof(integer)) 解法很简单，直接看实现1就可以

2 可是如果想使用O（n）的算法，一直局限在实现1中如何更新，后来想到了可以使用dp来解决

dp也很容易
dp[i] 表示i拥有的个数，那么很容易想到如果i第一位是0，dp[i] = dp[i>>1] ,如果i第一位为1，则dp[i] = dp[i>>1]

参考实现1

参考实现2

实现1

```
/**
 * @param {number} num
 * @return {number[]}
 */

// Runtime: 96 ms, faster than 83.28% of JavaScript online submissions for Counting Bits.
// Memory Usage: 44.6 MB, less than 47.00% of JavaScript online submissions for Counting Bits.

export default (num) => {
  let res = [0];
  for (let i = 1; i <= num; i++) {
    let bits = 0;
    while (i !== 0) {
      bits++;
      i &= i - 1;
```

```
    }
    res.push(bits);
  }
  return res;
};
```

实现2

```
/**
 * @param {number} num
 * @return {number[]}
 */

// Runtime: 92 ms, faster than 94.32% of JavaScript online submissions for Counting Bits.
// Memory Usage: 44.8 MB, less than 34.70% of JavaScript online submissions for Counting Bits.
// dp[i] 表示i拥有的个数，那么很容易想到如果i第一位是0，dp[i] = dp[i>>1] ,如果i第一位为1，则dp[i] = dp[i>>1]
export default (num) => {
  const dp = [];
  dp[0] = 0;
  if (num === 0) {
    return dp;
```

```
  }
  dp[1] = 1;
  if (num === 1) {
    return dp;
  }
  for (let i = 2; i <= num; i++) {
    dp[i] = dp[i >> 1] + (i & 1);
  }
  return dp;
};
```

693. 交替位二进制数

题目描述

给定一个正整数，检查它的二进制表示是否总是 0、1 交替出现：换句话说，就是二进制表示中相邻两位的数字永不相同。

例子1

Input: 5

output: true
解释：5 的二进制表示是：101

例子2

Input: 7

output: false
解释：7 的二进制表示是：111.

思考

1 如果只是看题解，你会发现很简单，可是到底是如何想出来的呢?

如果想求n是否都是交替表示，如果是交替表示的话，可以先求n^n/2,这时候n^n/2 所有位置肯定都是1，然后再和n^n/2+1执行与操作，如果结果为0就是交替位，否则就不是

参考实现1

实现1

```
/**
 * @param {number} n
 * @return {boolean}
 */
// Runtime: 76 ms, faster than 80.61% of JavaScript online submissions for Binary Number with Alternating Bits.
// Memory Usage: 38.7 MB, less than 54.08% of JavaScript online submissions for Binary Number with Alternating Bits.
export default (n) => {
  n = n ^ Math.floor(n >> 1);
```

```
  return !(n & (n + 1));
};
```

476. 数字的补数

题目描述

给定一个正整数，输出它的补数。补数是对该数的二进制表示取反。

例子1

Input: 5

output: 2
解释: 5 的二进制表示为 101（没有前导零位），其补数为 010。所以你需要输出 2 。

例子2

Input: 1

output: 0
解释：1 的二进制表示为 1（没有前导零位），其补数为 0。所以你需要输出 0 。

思考

1 这里很明显想取反，比如当是5的时候，二进制是101，但是如果直接取反5，高位都是1，明显不符合结果，所以必须想办法把高位给置0

所以可以通过取得和5同样的mask，比如111，高位都是0

参考实现1

实现1

```
/**
 * @param {number} num
 * @return {number}
 */
// Runtime: 72 ms, faster than 94.29% of JavaScript online submissions for Number Complement.
// Memory Usage: 38.8 MB, less than 30.00% of JavaScript online submissions for Number Complement.
export default (n) => {
  let mask = 1;
  while (mask < n) {
    mask = (mask << 1) | 1;
  }
  return ~n & mask;
};
```

第11章综合运用各种数据结构

常用的数据结构

常用的数据结构有数组，hash表，

448. 找到所有数组中消失的数字

题目描述

给定一个范围在 1 ≤ a[i] ≤ n (n = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

例子1

Input: [4,3,2,7,8,2,3,1]

output: [5,6]

思考

1 首先想到了使用数组的特性，因为这里1 ≤ a[i] ≤ n，所可以把每个a[i]放到数组中a[i]的位置上，到最后发现数组上没有放置数的位置就是缺失的数字

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */

const swap = (nums, i, j) => {
  const temp = nums[j];
  nums[j] = nums[i];
  nums[i] = temp;
};
// Runtime: 204 ms, faster than 29.69% of JavaScript online submissions for Find All Numbers Disappeared in an Array.
// Memory Usage: 47.3 MB, less than 48.68% of JavaScript online submissions for Find All Numbers Disappeared in an Array.
export default (nums) => {
  const len = nums.length;
  const res = new Array(len).fill(0);
```

```
  for (let i = 0; i < nums.length; i++) {
    if (res[nums[i] - 1] === 0) {
      res[nums[i] - 1] = nums[i];
    }
  }

  for (let i = 0; i < res.length; i++) {
    if (res[i] === 0) {
      res[i] = i + 1;
    } else {
      res[i] = 0;
    }
  }

  return res.filter((item) => item !== 0);
};
```

时间复杂度O（n），空间复杂度O（n）

48. 旋转图像

题目描述

给定一个 n × n 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明:
你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

例子1

```
给定 matrix =
[
 [1,2,3],
 [4,5,6],
 [7,8,9]
],

原地旋转输入矩阵，使其变为:
[
 [7,4,1],
 [8,5,2],
 [9,6,3]
]
```

例子2

```
给定 matrix =
[
 [ 5, 1, 9,11],
 [ 2, 4, 8,10],
 [13, 3, 6, 7],
 [15,14,12,16]
],

原地旋转输入矩阵，使其变为:
[
 [15,13, 2, 5],
 [14, 3, 4, 1],
 [12, 6, 8, 9],
 [16, 7,10,11]
]
```

思考

1 这里首先想到了很明显可以递归，首先交换外层的，然后再递归交换里边的

参考实现1

2 第二种是比较通用的方法，是解决类似的顺时针或者逆时针旋转的方法

```
* 顺时针旋转
* 首先上下交换,这里是指第一行和最后一行交换，第二行和倒数第二行，依次类推, 然后交换对角线的数字
*1 2 3   7 8 9   7 4 1
*4 5 6 => 4 5 6 => 8 5 2
*7 8 9   1 2 3   9 6 3
```

```
* 逆时针旋转
* 首先左右交换,这里是指第一列和最后一列交换，第二列和倒数第二列，依次类推, 然后交换对角线的数字
*1 2 3   3 2 1   3 6 9
*4 5 6 => 6 5 4 => 2 5 8
*7 8 9   9 8 7   1 4 7
```

参考实现2

实现1

```
/**
 * @param {number[][]} matrix
```

```
 * @return {void} Do not return anything, modify matrix in-place instead.
 */

const rotate1 = (matrix, begin, end) => {
  const len = end - begin;
  if (begin >= end) {
    return;
  }
  let tempTop = [];
  for (let i = begin; i <= end; i++) {
    tempTop.push(matrix[begin][i]);
  }

  let tempRight = [];
  for (let i = begin; i <= end; i++) {
    tempRight.push(matrix[i][end]);
  }
  let tempBottom = [];
  for (let i = end; i >= begin; i--) {
    tempBottom.push(matrix[end][i]);
  }
  let tempLeft = [];
  for (let i = end; i >= begin; i--) {
```

```
    tempLeft.push(matrix[i][begin]);
  }
  // 替换最右边
  for (let i = begin; i <= end; i++) {
    matrix[i][end] = tempTop[i - begin];
  }
  // console.log(temp, matrix);
  for (let i = end; i >= begin; i--) {
    matrix[end][i] = tempRight[end - i];
  }
  for (let i = end; i >= begin; i--) {
    matrix[i][begin] = tempBottom[end - i];
  }
  for (let i = begin; i <= end; i++) {
    // console.log(tempLeft);
    matrix[begin][i] = tempLeft[i - begin];
  }

  rotate1(matrix, begin + 1, end - 1);
};
// Runtime: 76 ms, faster than 82.34% of JavaScript online submissions for Rotate Image.
// Memory Usage: 38.9 MB, less than 23.65% of JavaScript online submissions for Rotate Image.
const rotate = (matrix) => {
```

```
  const len = matrix.length;
  rotate1(matrix, 0, len - 1);
  // console.log(matrix);
  // return matrix;
};
export default rotate;
```

实现2

```
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-place instead.
 */

// Runtime: 72 ms, faster than 93.88% of JavaScript online submissions for Rotate Image.
// Memory Usage: 38.9 MB, less than 32.93% of JavaScript online submissions for Rotate Image.
const rotate = (matrix) => {
  const len = matrix.length;
  let low = 0;
  let high = len - 1;
```

```
  while (low < high) {
    for (let i = 0; i < len; i++) {
      const temp = matrix[low][i];
      matrix[low][i] = matrix[high][i];
      matrix[high][i] = temp;
    }
    low++;
    high--;
  }
  for (let i = 1; i < len; i++) {
    for (let j = 0; j < i; j++) {
      const temp = matrix[i][j];
      matrix[i][j] = matrix[j][i];
      matrix[j][i] = temp;
    }
  }
};
export default rotate;
```

时间复杂度O(n * n) 空间复杂度O（1）

240. 搜索二维矩阵 II

题目描述

编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：

> 每行的元素从左到右升序排列。
> 每列的元素从上到下升序排列。

例子1

input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5
output: true

思考

1 刚开始想使用二分搜索，可是后来发现不行，后来看了下题解很容易，只是特别不容易想到

参考实现1

实现1

```
```

```javascript
/**
 * @param {number[][]} matrix
 * @param {number} target
 * @return {boolean}
 */
// Runtime: 296 ms, faster than 64.11% of JavaScript online submissions for Search a 2D Matrix II.
// Memory Usage: 41.8 MB, less than 58.69% of JavaScript online submissions for Search a 2D Matrix II.
export default (matrix, target) => {
  if (matrix.length === 0) {
    return false;
  }
  let row = 0;
  let col = matrix[0].length - 1;
  while (col >= 0 && row < matrix.length) {
    if (target === matrix[row][col]) {
      return true;
    } else if (target < matrix[row][col]) {
      col--;
    } else {
      row++;
    }
  }
  return false;
```

```
};
```

时间复杂度O（max（m,n））, 空间复杂度O（1）

769. 最多能完成排序的块

题目描述

数组arr是[0, 1, ..., arr.length - 1]的一种排列，我们将这个数组分割成几个“块”，并将这些块分别进行排序。之后再连接起来，使得连接的结果和按升序排序后的原数组相同。

我们最多能将数组分成多少块?

例子1
input: arr = [4,3,2,1,0]
output: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [4, 3], [2, 1, 0] 的结果是 [3, 4, 0, 1, 2]，这不是有序的数组。

例子2
input: arr = [1,0,2,3,4]
output: 4

解释:
我们可以把它分成两块，例如 [1, 0], [2, 3, 4]。
然而，分成 [1, 0], [2], [3], [4] 可以得到最多的块数。

注意：

1 arr 的长度在 [1, 10] 之间。
2 arr[i]是 [0, 1, ..., arr.length - 1]的一种排列。

思考

1 刚开始想遇到最大值就拆开，可是发现不行，比如输入[1,2,0,5]的时候，就不能按照这种策略来执行，后来发现如果我们遇到想要拆开的时候，必须在max前面的所有数字必须都已经在前面的数组中使用过了。所以可以设置一个hasUsed数组来记录数字是否使用过

参考实现1

2 还有特别简单的方法，当然这里的简单是指解法，但是并不是指可以很简单的思考出来。

我们可以遍历数组，记录遇到的最大数字max，如果发现max等于我们遍历数组的下标，就可以拆开了。

这里主要是因为输入数组的里边的数组刚好等于数组的长度减1，所以当遇到max等于数组下标的时候，这时候就可以拆分为一个子数组，因为这时候前面的肯定都排好序了或者都在目前拆分的数组中

参考实现2

实现1

```javascript
/**
 * @param {number[]} arr
 * @return {number}
 */
// Runtime: 68 ms, faster than 98.04% of JavaScript online submissions for Max Chunks To Make Sorted.
// Memory Usage: 38.5 MB, less than 50.98% of JavaScript online submissions for Max Chunks To Make Sorted.
export default (arr) => {
  const hasUsed = new Array(arr.length).fill(0);
  let res = 1;
  let max = arr[0];
  hasUsed[max] = 1;
  for (let i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
      let flag = hasUsed[0];
      for (let j = 1; j < max; j++) {
        flag &= hasUsed[j];
      }
      if (flag) {
        res++;
      }
      max = arr[i];
    }
```

```
    hasUsed[arr[i]] = 1;
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} arr
 * @return {number}
 */
// Runtime: 64 ms, faster than 100.00% of JavaScript online submissions for Max Chunks To Make Sorted.
// Memory Usage: 38.5 MB, less than 50.98% of JavaScript online submissions for Max Chunks To Make Sorted.
export default (arr) => {
  let res = 0;
  let max = arr[0];
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] > max) {
      max = arr[i];
    }
    if (max === i) {
```

```
      res++;
    }
  }
  return res;
};
```

769. 最多能完成排序的块

题目描述

使用栈模拟队列，栈只允许两种操作，入栈和出栈

进阶：

是否可以所有的操作都可以均摊为O（1）

思考

1 使用两个栈实现，思路比较简单，就是来回搬运数据

可以参考实现1

2 如果想所有的操作都是0（1），可以使用front代表目前的队列首元素，这样push，peek和empty都很容易是O（1）操作的时间，而push的时间复杂度最坏的情况下是O（n），为了达到O（1）的时间，可以想下办法，push 的操作n次之后，我们才在pop的时候执行一次搬运操

作，可以参考下代码

可以参考实现2

实现1

```
/**
 * Initialize your data structure here.
 */
// Runtime: 72 ms, faster than 87.77% of JavaScript online submissions for Implement Queue using Stacks.
// Memory Usage: 38.7 MB, less than 10.22% of JavaScript online submissions for Implement Queue using Stacks.
var MyQueue = function () {
  this.stack1 = [];
  this.stack2 = [];
};

/**
 * Push element x to the back of queue.
 * @param {number} x
 * @return {void}
 */
MyQueue.prototype.push = function (x) {
  this.stack1.push(x);
```

```
};

/**
 * Removes the element from in front of queue and returns that element.
 * @return {number}
 */
MyQueue.prototype.pop = function () {
  if (this.stack1.length === 0) {
    return false;
  }
  while (this.stack1.length > 0) {
    const temp = this.stack1.pop();
    this.stack2.push(temp);
  }
  const res = this.stack2.pop();

  while (this.stack2.length > 0) {
    const temp = this.stack2.pop();
    this.stack1.push(temp);
  }
  return res;
};
```

```
/**
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function () {
  if (this.stack1.length === 0) {
    return false;
  }
  while (this.stack1.length > 0) {
    const temp = this.stack1.pop();
    this.stack2.push(temp);
  }
  const res = this.stack2.pop();
  this.stack2.push(res);
  while (this.stack2.length > 0) {
    const temp = this.stack2.pop();
    this.stack1.push(temp);
  }
  return res;
};

/**
 * Returns whether the queue is empty.
```

```
 * @return {boolean}
 */
MyQueue.prototype.empty = function () {
  return this.stack1.length === 0;
};

/**
 * Your MyQueue object will be instantiated and called as such:
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */
```

实现2

```
/**
 * Initialize your data structure here.
 */
// Runtime: 72 ms, faster than 87.77% of JavaScript online submissions for Implement Queue using Stacks.
```

```javascript
// Memory Usage: 38.7 MB, less than 10.22% of JavaScript online submissions for Implement Queue using Stacks.
var MyQueue = function () {
  this.stack1 = [];
  this.stack2 = [];
  this.front = "";
};

/**
 * Push element x to the back of queue.
 * @param {number} x
 * @return {void}
 */
MyQueue.prototype.push = function (x) {
  if (this.stack1.length === 0) {
    this.front = x;
  }
  this.stack1.push(x);
};

/**
 * Removes the element from in front of queue and returns that element.
 * @return {number}
 */
```

```
MyQueue.prototype.pop = function () {
  // if (this.stack2.length === 0) {
  //   return false;
  // }
  if (this.stack2.length >= 1) {
    const res = this.stack2.pop();
    if (this.stack2.length > 0) {
      this.front = this.stack2.pop();
      this.stack2.push(this.front);
    }
    return res;
  }
  if (this.stack2.length === 0) {
    while (this.stack1.length > 0) {
      const temp = this.stack1.pop();
      this.stack2.push(temp);
    }
  }

  const res = this.stack2.pop();
  if (this.stack2.length > 0) {
    this.front = this.stack2.pop();
    this.stack2.push(this.front);
```

```
  }

  return res;
};

/**
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function () {
  return this.front;
};

/**
 * Returns whether the queue is empty.
 * @return {boolean}
 */
MyQueue.prototype.empty = function () {
  return this.stack1.length === 0 && this.stack2.length === 0;
};

/**
 * Your MyQueue object will be instantiated and called as such:
```

```
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */
```

155. 最小栈

题目描述

设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。

```
push(x) —— 将元素 x 推入栈中。
pop() —— 删除栈顶的元素。
top() —— 获取栈顶元素。
getMin() —— 检索栈中的最小元素。
```

例子1

```
输入：
```

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出:
[null,null,null,null,-3,null,0,-2]

解释:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

提示:

1 pop、top 和 getMin 操作总是在 非空栈 上调用。

思考

1 思路也很简单，使用一个数组依次把最小的数字放进去，当pop的时候，如果发现了pop出去了最小数，则从最小数组中重新拿下一个最小

的

可以参考实现1

实现1

```
/**
 * initialize your data structure here.
 */
// Runtime: 120 ms, faster than 77.56% of JavaScript online submissions for Min Stack.
// Memory Usage: 45.8 MB, less than 48.96% of JavaScript online submissions for Min Stack.
var MinStack = function () {
  this.arr = [];
  this.min = Number.MAX_VALUE;
  this.minArr = [];
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function (x) {
  this.arr.push(x);
```

```
  if (this.min >= x || this.minArr.length === 0) {
    this.minArr.push(x);
    this.min = x;
  }
};
/**
 * @return {void}
 */
MinStack.prototype.pop = function () {
  const res = this.arr.pop();
  if (res === this.minArr[this.minArr.length - 1]) {
    this.minArr.pop();
    this.min = this.minArr[this.minArr.length - 1];
  }
  return res;
};

/**
 * @return {number}
 */
MinStack.prototype.top = function () {
  return this.arr[this.arr.length - 1];
};
```

```
/**
 * @return {number}
 */
MinStack.prototype.getMin = function () {
  return this.min;
};

/**
 * Your MinStack object will be instantiated and called as such:
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
 * var param_3 = obj.top()
 * var param_4 = obj.getMin()
 */
```

20. 有效的括号

题目描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

> 左括号必须用相同类型的右括号闭合。
> 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

例子1
input:"()"
output: true

思考

1 题目很简单，标准的使用栈解决问题

可以参考实现1

实现1

```
/**
 * @param {string} s
 * @return {boolean}
 */
```

```
// Runtime: 88 ms, faster than 24.85% of JavaScript online submissions for Valid Parentheses.
// Memory Usage: 39.5 MB, less than 31.06% of JavaScript online submissions for Valid Parentheses.
export default (s) => {
  const left = [];
  const leftS = ["(", "{", "["];
  const rightS = [")", "}", "]"];
  for (let i = 0; i < s.length; i++) {
    if (leftS.includes(s.charAt(i))) {
      left.push(s.charAt(i));
    } else {
      switch (s.charAt(i)) {
        case ")":
          if (left[left.length - 1] === "(") {
            left.pop();
          } else {
            return false;
          }
          break;
        case "}":
          if (left[left.length - 1] === "{") {
            left.pop();
          } else {
```

```
            return false;
          }
          break;
        case "]":
          if (left[left.length - 1] === "[") {
            left.pop();
          } else {
            return false;
          }
          break;
        default:
          break;
      }
    }
  }
  return left.length === 0 ? true : false;
};
```

739. 每日温度

题目描述

请根据每日 气温 列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示：气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。

思考

1 暴力解法很简单

可以参考实现1

2 还有一种解法是使用单调栈，不过不是很不好理解，而且这里还融和了逆向思维，但是如果立即了其实也很简单。

从后向前遍历数组，维持一个单调递减的栈，如果发现当前元素大于栈顶元素，则栈pop出当前元素，继续比较。

比如
T =[89, 62, 70, 58, 47, 47, 46, 76, 100, 70]

栈 stack = []

res = []

1 从右向左遍历数组，遇到70的时候，因为此时stack为空，所以res.push(0), 因为我们要求的是距离当前元素的个数，所以stack.push(9)

2 当遇到100的时候，因为100大于当前栈顶元素70,所以执行stack.pop(), stack变成[],因为此时stack为空，所以res.push(0)，re=[0,0] ， 最后再把当前下标push进栈，stack.push(8)

以此类推

实现1

```
/**
 * @param {number[]} T
 * @return {number[]}
 */

// Runtime: 160 ms, faster than 70.63% of JavaScript online submissions for Daily Temperatures.
// Memory Usage: 49 MB, less than 72.86% of JavaScript online submissions for Daily Temperatures.
export default (T) => {
  const len = T.length;
  const stack = [];
  const res = [];
  for (let i = len - 1; i >= 0; i--) {
    const current = T[i];
    while (stack.length > 0 && current >= T[stack[stack.length - 1]]) {
      stack.pop();
    }
    res[i] = stack.length === 0 ? 0 : stack[stack.length - 1] - i;

    stack.push(i);
    console.log(stack);
```

```
  }
  return res;
};
```

实现堆

题目描述

实现一个堆

思考

堆排序明白了思路就很容易理解，大体思路就是先遍历非叶子节点，建立一个大顶堆，然后交换第一个元素和最后一个元素，然后重新建立大顶堆，重复此过程就可以了

实现1

```
class Heap {
  constructor(initArr = []) {
    this.arr = initArr;
    const len = this.arr.length - 1;
    this.buildHeap(len);
    console.log(112, this.arr);
```

```
  }
  // 最大值
  top() {
    return this.arr[0];
  }
  // push进来一个数
  push(val) {
    this.arr.push(val);
    this.swim(this.arr.length - 1);
  }
  // push进来一个数
  pop(val) {
    const res = this.arr[0];
    const res1 = this.arr.pop();
    this.arr[0] = res1;
    const len = this.arr.length - 1;
    this.sink(0);
    return res;
  }
  swap(i, j) {
    const temp = this.arr[i];
    this.arr[i] = this.arr[j];
    this.arr[j] = temp;
```

```
}

buildHeap(pos) {
  for (let j = Math.floor(pos / 2); j >= 0; j--) {
    this.sink(j);
  }
}

// 上浮
swim(pos) {
  while (pos >= 1 && this.arr[Math.floor(pos / 2)] < this.arr[pos]) {
    this.swap(Math.floor(pos / 2), pos);
    pos = Math.floor(pos / 2);
  }
}
// 下沉
sink(pos) {
  const len = this.arr.length;
  while (2 * pos < len) {
    let i = 2 * pos;
    if (i < len && this.arr[i] < this.arr[i + 1]) {
      ++i;
    }
```

```
      if (this.arr[pos] >= this.arr[i]) break;
      this.swap(pos, i);
      pos = i;
    }
  }
}
export default Heap;
```

23. 合并有序链表

题目描述

给定 k 个增序的链表，试将它们合并成一条增序链表。

例子1
input: lists = [[1,4,5],[1,3,4],[2,6]]
output: [1,1,2,3,4,4,5,6]
解释：

[
1->4->5,
1->3->4,

2->6
]
合并后
1->1->2->3->4->4->5->6

思考

1 直接使用二分合并就可以了

可以参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
```

```
// Runtime: 364 ms, faster than 27.05% of JavaScript online submissions for Merge k Sorted Lists.
// Memory Usage: 43.7 MB, less than 74.12% of JavaScript online submissions for Merge k Sorted Lists.
const merge = (list1, list2) => {
  if (!list1 || !list2) return list1 || list2;
  let node = new ListNode(null);
  // 暂存头结点
  const root = node;
  while (list1 && list2) {
    if (list1.val <= list2.val) {
      node.next = list1;
      list1 = list1.next;
    } else {
      node.next = list2;
      list2 = list2.next;
    }
    node = node.next;
  }
  if (list1) node.next = list1;
  if (list2) node.next = list2;
  return root.next;
};

export default (lists) => {
```

```
  let root = lists[0];
  for (let i = 1; i < lists.length; i++) {
    root = merge(root, lists[i]);
  }

  return root || null;
};
```

218 天际线问题

题目描述

给定 k 个增序的链表，试将它们合并成一条增序链表。城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。现在，假设您获得了城市风光照片（图A）上显示的所有建筑物的位置和高度，请编写一个程序以输出由这些建筑物形成的天际线（图B）。

每个建筑物的几何信息用三元组[Li，Ri，Hi]表示，其中Li和Ri分别是第 i 座建筑物左右边缘的 x 坐标，Hi 是其高度。可以保证0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX和Ri - Li > 0。您可以假设所有建筑物都是在绝对平坦且高度为 0 的表面上的完美矩形。
例如，图A中所有建筑物的尺寸记录为：[[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]] 。
输出是以 [[x1,y1], [x2, y2], [x3, y3], ...] 格式的“关键点”（图B中的红点）的列表，它们唯一地定义了天际线。关键点是水平线段的左端点。请注意，最右侧建筑物的最后一个关键点仅用于标记天际线的终点，并始终为零高度。此外，任何两个相邻建筑物之间的地面都应被视为天际线

轮廓的一部分。
例如，图B中的天际线应该表示为：[[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0]]。

思考

1 首先是把所有的长方形形成一个拆成两个节点，比如[2,9,10]
拆成[2,-10],[9,10]两个节点，负数表示是长方形的起点，正数表示长方形的结束点

然后维持一个数组pq，并且假设pq的最大值是maxVal，遍历所有节点，如果发现是起始点，并且比maxVal大，则加入到结果中，如果不比maxVal大，则直接把高度push进入pq中去，如果发现是结束点，则从pq中删除该结束点的高度，如果此时maxVal变化了，此时也把该节点的坐标和此时的maxVal加入到结果中。

https://briangordon.github.io/2014/08/the-skyline-problem.html

视频 https://youtu.be/GSBLe8cKu0s

可以参考实现1

实现1

```
const remove = (arr = [], val) => {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === val) {
      for (let j = i; j < arr.length - 1; j++) {
        arr[j] = arr[j + 1];
      }
```

```
      arr.pop();
      return;
    }
  }
};
// Runtime: 384 ms, faster than 52.12% of JavaScript online submissions for The Skyline Problem.
// Memory Usage: 45.6 MB, less than 83.64% of JavaScript online submissions for The Skyline Problem.
export default (buildings) => {
  // 最后的结果
  const res = [];

  // 所有的节点
  const points = [];

  for (let i = 0; i < buildings.length; i++) {
    // 表示起始点
    points.push([buildings[i][0], -buildings[i][2]]);
    // 表示结束点
    points.push([buildings[i][1], buildings[i][2]]);
  }
  // 排序所有节点

  points.sort((a, b) => {
```

```
  if (a[0] === b[0]) {
    return a[1] - b[1];
  }
  return a[0] - b[0];
});
// 表示遍历过的高度的数组
const pq = [0];
// 表示遍历过的高度的数组pq的最大值
let pqMaxVal = 0;

for (let i = 0; i < points.length; i++) {
  if (points[i][1] < 0) {
    pq.push(-points[i][1]);
  } else {
    remove(pq, points[i][1]);
  }
  const tempMaxVal = Math.max(...pq);
  if (tempMaxVal !== pqMaxVal) {
    res.push([points[i][0], tempMaxVal]);
    pqMaxVal = tempMaxVal;
  }
}
return res;
```

```
};
```

239. 滑动窗口最大值

题目描述

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

例子1

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
```

```
 1  3  -1  -3 [5  3  6] 7      6
 1  3  -1  -3  5 [3  6  7]     7
```

思考

1 可以使用双指针，一个begin指向当前最大的值，另外一个是数组下标，如果i-begin+1 大于k，重新寻找最大值，如果小于k，则直接push进入结果就可以了

可以参考实现1

2 在实现1的时候，发现有很多需要重复的，这里可以使用单调栈来进行优化。存储从最大到次大的，比如

```
// nums = [1, 3, -1, -3, 5, 3, 6, 7], and k = 3
// Monotonic queue max
// [1] -
// [3] -
// [3, -1] 3
// [3, -1, -3] 3
// [5] 5
// [5, 3] 5
// [6] 6
// [7] 7
```

参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
// Runtime: 1996 ms, faster than 22.94% of JavaScript online submissions for Sliding Window Maximum.
// Memory Usage: 69 MB, less than 38.21% of JavaScript online submissions for Sliding Window Maximum.
export default (nums, k) => {
  const res = [];
  let max = nums[0];
  let begin = 0;
  for (let i = 1; i < k; i++) {
    if (nums[i] > max) {
      begin = i;
      max = nums[begin];
    }
  }
  res.push(max);
  for (let i = k; i < nums.length; i++) {
    if (i - begin + 1 <= k && nums[i] < nums[begin]) {
```

```
      res.push(max);
    } else {
      max = nums[begin + 1];
      begin = begin + 1;
      for (let m = begin + 1; m <= i; m++) {
        if (nums[m] > max) {
          max = nums[m];
          begin = m;
        }
      }
      res.push(max);
    }
  }
  return res;
};
```

实现2

```
// Runtime: 724 ms, faster than 46.21% of JavaScript online submissions for Sliding Window Maximum.
// Memory Usage: 69.3 MB, less than 29.89% of JavaScript online submissions for Sliding Window Maximum.
export default (nums, k) => {
```

```
  const res = [];
  const q = [];

  for (let i = 0; i < nums.length; i++) {
    // 最大单调递减
    while (q.length - 1 >= 0 && nums[i] > q[q.length - 1]) {
      q.pop();
    }
    q.push(nums[i]);

    // 从下标往前k个数
    const j = i + 1 - k;
    if (j >= 0) {
      res.push(q[0]);
      if (nums[j] === q[0]) {
        q.shift();
      }
    }
  }
  return res;
};
```

1. 两数之和

题目描述

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

你可以按任意顺序返回答案。

例子1
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
例子2
输入：nums = [3,2,4], target = 6
输出：[1,2]

提示：
1 只存在一个有效的答案

思考

1 使用hash表很好解决

可以参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
// Runtime: 80 ms, faster than 64.85% of JavaScript online submissions for Two Sum.
// Memory Usage: 38.3 MB, less than 95.29% of JavaScript online submissions for Two Sum.
export default (nums, target) => {
  const res = [];
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    if (!map.has(nums[i])) {
      map.set(nums[i], i);
    }
    if (map.get(target - nums[i]) >= 0 && i !== map.get(target - nums[i])) {
      res.push(i);
      res.push(map.get(target - nums[i]));
      return res;
    }
  }
```

```
};
```

128. 最长连续序列

题目描述

给定一个未排序的整数数组 nums ， 找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

进阶：你可以设计并实现时间复杂度为 O(n) 的解决方案吗?

例子1
输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。 。
例子2
输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9
解释：最长数字连续序列是 [0,1, 2, 3, 4,5,6,7,8]。它的长度为9.

思考

1 首先排序数组，然后遍历数组，找到每个数字的最长连续长度

可以参考实现1

2 使用hash表存储每个数字，然后遍历数组，然后找到数组中每个数字的最长长度，这里使用hash快速查找，快速跳过

可以参考实现2

3 遍历数组，对于每个数字nums[i]，找出在hash表中比该数字小1的长度len1和找出在hash表中比该数字大1的长度len2，所以此时该数字的最长连续长度就是Math.max(res,len1+len2+1),然后再不断更新hash表中nums[i]-left 的长度为Math.max(res,len1+len2+1)，nums[i]+right的长度是Math.max(res,len1+len2+1)

可以参考实现3

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 148 ms, faster than 17.10% of JavaScript online submissions for Longest Consecutive Sequence.
// Memory Usage: 40.1 MB, less than 90.23% of JavaScript online submissions for Longest Consecutive Sequence.
export default (nums) => {
  nums.sort((a, b) => a - b);
  let res = 0;
  for (let i = 0; i < nums.length; i++) {
    let count = 1;
    for (let j = i; j < nums.length - 1; j++) {
      if (nums[j + 1] === nums[j] + 1) {
```

```
        count++;
      } else if (nums[j + 1] === nums[j]) {
        continue;
      } else {
        break;
      }
    }
    res = Math.max(res, count);
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 76 ms, faster than 95.99% of JavaScript online submissions for Longest Consecutive Sequence.
// Memory Usage: 41.7 MB, less than 9.08% of JavaScript online submissions for Longest Consecutive Sequence.
export default (nums) => {
```

```
const map = new Map();

for (let i = 0; i < nums.length; i++) {
  if (!map.has(nums[i])) {
    map.set(nums[i], true);
  }
}
let res = 0;
for (let i = 0; i < nums.length; i++) {
  if (!map.has(nums[i])) {
    continue;
  }
  let right = nums[i];
  let left = nums[i] - 1;
  let max = 0;
  while (map.has(right)) {
    map.delete(right);
    max++;
    right++;
  }
  while (map.has(left)) {
    map.delete(left);
    max++;
```

```
      left--;
    }
    res = Math.max(res, max);
  }

  return res;
};
```

实现3

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 88 ms, faster than 61.43% of JavaScript online submissions for Longest Consecutive Sequence.
// Memory Usage: 40.9 MB, less than 58.81% of JavaScript online submissions for Longest Consecutive Sequence.
export default (nums) => {
  const map = new Map();
  let res = 0;
  for (let i = 0; i < nums.length; i++) {
    if (!map.has(nums[i])) {
```

```
      const left = map.get(nums[i] - 1) ? map.get(nums[i] - 1) : 0;
      const right = map.get(nums[i] + 1) ? map.get(nums[i] + 1) : 0;
      const sum = left + right + 1;
      map.set(nums[i], sum);
      res = Math.max(res, sum);
      map.set(nums[i] - left, sum);
      map.set(nums[i] + right, sum);
    } else {
      continue;
    }
  }

  return res;
};
```

149. 直线上最多的点数

题目描述

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

例子1

输入：[[1,1],[2,2],[3,3]]
输出：3
解释：

```
|
|     o
|   o
| o
+------------->
0 1 2 3 4。
```

思考

1 题目很简单，就是可以遍历所有的直线，找出那条直线上的点最多。

可以参考实现1

2 另外一种就是使用hash存储相同的斜率的

可以参考实现2

实现1

```
/**
```

```
 * @param {number[][]} points
 * @return {number}
 */

const getK = (arr1, arr2) => {
  if (Math.abs(arr1[0] - arr2[0]) !== 0) {
    return (arr1[1] - arr2[1]) / (arr1[0] - arr2[0]);
  } else {
    // 当是一条垂直线的时候
    return "cur";
  }
};
const qualArr = (arr1, arr2) => {
  for (let i = 0; i < arr1.length; i++) {
    if (arr1[i] !== arr2[i]) {
      return false;
    }
  }
  return true;
};
export default (points) => {
  let max = 0;
  if (!points || points.length === 0) return 0;
```

```
  if (points.length === 1) return 1;
  if (points.length === 2) return 2;
  for (let i = 2; i < points.length; i++) {
    let max1 = 0;
    for (let j = i - 1; j >= 0; j--) {
      const k = getK(points[i], points[j]);
      max1 = 2;
      for (let k1 = 0; k1 < i; k1++) {
        if (k1 === j) {
          continue;
        }
        if (k === getK(points[i], points[k1]) || qualArr(points[k1], points[i]) || qualArr(points[k1], points[j])) {
          max1++;
        }

        if (max1 > max) {
          max = max1;
        }
      }
    }
  }
  return max;
};
```

实现2

```
/**
 * @param {number[][]} points
 * @return {number}
 */
// Runtime: 100 ms, faster than 79.41% of JavaScript online submissions for Max Points on a Line.
// Memory Usage: 44.2 MB, less than 63.24% of JavaScript online submissions for Max Points on a Line.
export default (points) => {
  const map = new Map();
  let max_count = 0;
  // 相同x坐标的数量
  let same_points_count = 0;
  // 相同y坐标的数量
  let same_y_count = 1;

  for (let i = 0; i < points.length; i++) {
    same_points_count = 0;
    same_y_count = 1;
    for (let j = i + 1; j < points.length; j++) {
```

```
    if (points[i][1] === points[j][1]) {
      ++same_y_count;
      if (points[i][0] === points[j][0]) {
        ++same_points_count;
      }
    } else {
      const temp = ((points[i][0] - points[j][0]) * 10000) / ((points[i][1] - points[j][1]) * 10000);
      if (map.has(temp)) {
        const tempArr = map.get(temp);
        const test =
          ((points[j][0] - tempArr[tempArr.length - 1][0]) * 10000) /
          ((points[j][1] - tempArr[tempArr.length - 1][1]) * 10000);
        if (
          test === temp ||
          (points[j][0] === tempArr[tempArr.length - 1][0] && points[j][1] === tempArr[tempArr.length - 1][1])
        ) {
          tempArr.push(points[j]);
        }
        map.set(temp, tempArr);
      } else {
        map.set(temp, [points[i], points[j]]);
      }
    }
```

```
    }
    max_count = Math.max(same_y_count, max_count);
    for (let value of map.values()) {
      max_count = Math.max(max_count, same_points_count + value.length);
    }
    map.clear();
  }
  return max_count;
};
```

332. 重新安排行程

题目描述

给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从 JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 开始。

提示：

1 如果存在多种有效的行程，请你按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前
2 所有的机场都用三个大写字母表示（机场代码）。

3 假定所有机票至少存在一种合理的行程。
4 所有的机票必须都用一次 且 只能用一次。

例子1
输入：[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
输出：["JFK", "MUC", "LHR", "SFO", "SJC"]

例子2
输入：[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
输出：["JFK","ATL","JFK","SFO","ATL","SFO"]
解释：另一种有效的行程是 ["JFK","SFO","ATL","JFK","ATL","SFO"]。但是它自然排序更大更靠后。

思考

1 这里如果想要解决，就要首先明确一个问题，就是这里的输入肯定是存在一条使用完所有的机票后，可以链接起来

明白这一点后，就比较容易了，一步步寻找，如果走到死胡同，然后退出重新寻找下一条路。
比如下图如果我们从jfk出发，如果发现到了A之后，还有机票没有使用，可以退出A,重新从D开始继续寻找

可以参考实现1

实现1

```
/**
 * @param {string[][]} tickets
```

```
 * @return {string[]}
 */
const sortArray = (a, b) => {
  if (a[0] === b[0]) {
    return a[1].localeCompare(b[1]);
  }
  return a[0].localeCompare(b[0]);
};
export default (tickets) => {
  if (!tickets || tickets.length === 0) return [];
  const map = new Map();
  const result = [];

  tickets.sort(sortArray);
  for (let i = 0; i < tickets.length; i++) {
    if (map.has(tickets[i][0])) {
      map.get(tickets[i][0]).push(tickets[i][1]);
    } else {
      map.set(tickets[i][0], [tickets[i][1]]);
    }
  }

  let key = "JFK";
```

```
  const drawback = [];
  for (let i = 0; i < tickets.length; i++) {
    while (!map.has(key) || map.get(key).length === 0) {
      drawback.push(key);
      key = result.pop();
    }
    result.push(key);
    key = map.get(key).shift();
  }
  result.push(key);
  while (drawback.length > 0) {
    result.push(drawback.pop());
  }

  return result;
};
```

303. 区域和检索 - 数组不可变

题目描述

给定一个整数数组 nums，求出数组从索引 i 到 j（i ≤ j）范围内元素的总和，包含 i、j 两点。

实现 NumArray 类：

1 NumArray(int[] nums) 使用数组 nums 初始化对象
2 int sumRange(int i, int j) 返回数组 nums 从索引 i 到 j（i ≤ j）范围内元素的总和，包含 i、j 两点（也就是 sum(nums[i], nums[i + 1], ... , nums[j])）

例子1
输入：["NumArray", "sumRange", "sumRange", "sumRange"]

[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

输出：[null, 1, -1, -3]
解释：
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)
numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))
numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2 + (-1))

思考

1 刚开始想使用hash把所有的组合都存储起来，这样等到sumRange的时候，就可以直接拿出来用了，但是发现超时了

可以参考实现1

2 后来发现可以使用一个数组copyNums，把输入的数组nums所有从0
到i位置的和存储起来，这样如果求i到j的和的时候，就可以使用

copyNums[j]- copyNums[i]

实现1

```
/**
 * @param {number[]} nums
 */
var NumArray = function (nums) {
  this.nums = nums;
  this.map = new Map();
  for (let i = 0; i < nums.length; i++) {
    for (let j = i; j < nums.length; j++) {
      if (this.map.has(`${i}${j - 1}`)) {
        const count = this.map.get(`${i}${j - 1}`) + nums[j];
        this.map.set(`${i}${j}`, count);
      } else {
        this.map.set(`${i}${j}`, nums[j]);
      }
    }
  }
};

/**
```

```
 * @param {number} i
 * @param {number} j
 * @return {number}
 */
NumArray.prototype.sumRange = function (i, j) {
  return this.map.get(`${i}${j}`);
};

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)
 * var param_1 = obj.sumRange(i,j)
 */
```

实现2

```
/**
 * @param {number[]} nums
 */

// Runtime: 116 ms, faster than 87.76% of JavaScript online submissions for Range Sum Query - Immutable.
```

```javascript
// Memory Usage: 45.8 MB, less than 33.85% of JavaScript online submissions for Range Sum Query - Immutable.
var NumArray = function (nums) {
  this.nums = nums;
  this.copyNums = [0];
  for (let i = 0; i < nums.length; i++) {
    this.copyNums[i + 1] = this.copyNums[i] + nums[i];
  }
};

/**
 * @param {number} i
 * @param {number} j
 * @return {number}
 */
NumArray.prototype.sumRange = function (i, j) {
  return this.copyNums[j + 1] - this.copyNums[i];
};

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)
 * var param_1 = obj.sumRange(i,j)
 */
```

304. 二维区域和检索 - 矩阵不可变

题目描述

给定一个二维矩阵，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2)。

例子1

```
给定 matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

思考

1 这里思想和上面的题目是一样的，只不多是二维变成了一维，不过思想还是一样的

参考实现1

实现1

```
/**
 * @param {number[][]} matrix
 */
var NumMatrix = function (matrix) {
  this.sumMatrix = [];
  const m = matrix.length;
  const n = matrix[0] ? matrix[0].length : 0;
  for (let i = 0; i <= m; i++) {
    this.sumMatrix[i] = new Array(n + 1).fill(0);
  }
  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      this.sumMatrix[i + 1][j + 1] = this.sumMatrix[i][j];
```

```
      for (let k = 0; k < j; k++) {
        this.sumMatrix[i + 1][j + 1] += matrix[i][k];
      }
      for (let k = 0; k < i; k++) {
        this.sumMatrix[i + 1][j + 1] += matrix[k][j];
      }
      this.sumMatrix[i + 1][j + 1] += matrix[i][j];
    }
  }
};

/**
 * @param {number} row1
 * @param {number} col1
 * @param {number} row2
 * @param {number} col2
 * @return {number}
 */
NumMatrix.prototype.sumRegion = function (row1, col1, row2, col2) {
  return (
    this.sumMatrix[row2 + 1][col2 + 1] -
    this.sumMatrix[row2 + 1][col1] -
    this.sumMatrix[row1][col2 + 1] +
```

```
    this.sumMatrix[row1][col1]
  );
};
// Runtime: 128 ms, faster than 40.59% of JavaScript online submissions for Range Sum Query 2D - Immutable.
// Memory Usage: 44.6 MB, less than 9.90% of JavaScript online submissions for Range Sum Query 2D - Immutable.
export default NumMatrix;
/**
 * Your NumMatrix object will be instantiated and called as such:
 * var obj = new NumMatrix(matrix)
 * var param_1 = obj.sumRegion(row1,col1,row2,col2)
 */
```

560. 和为K的子数组

题目描述

给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。

例子1
input: nums = [1,1,1], k = 2
output: 2 , [1,1] 与 [1,1] 为两种不同的情况

例子2
input: nums = [1,2,3], k = 3
output: 2 , [1,2] 与 [3] 为两种不同的情况

思考

1 这里也可以使用前面的类似前缀和，但是发现超时了

参考实现1

2 可以利用空间换时间，使用hash表存储前缀和出现的次数，当遇到一个前缀和sum的时候，可以到hash表中查找是否存在hash[sum-k],如果存在，添加到结果中

参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */

export default (nums, k) => {
  const copyNums = [0];
```

```
  for (let i = 0; i < nums.length; i++) {
    copyNums[i + 1] = copyNums[i] + nums[i];
  }

  let res = 0;
  for (let i = nums.length; i >= 1; i--) {
    for (let j = i - 1; j >= 0; j--) {
      if (copyNums[i] - copyNums[j] === k) {
        res++;
      }
      // if (copyNums[i] - copyNums[j] > k && nums[j - 1] > 0) {
      //   break;
      // }
    }
  }
  return res;
};
```

实现2

```
/**
```

```
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */

// Runtime: 108 ms, faster than 73.38% of JavaScript online submissions for Subarray Sum Equals K.
// Memory Usage: 46.7 MB, less than 67.96% of JavaScript online submissions for Subarray Sum Equals K.
export default (nums, k) => {
  if (nums.length === 1 && nums[0] !== k) return 0;
  const map = new Map();
  map.set(0, 1);
  let res = 0;
  let copyNums = 0;
  for (let i = 0; i < nums.length; i++) {
    copyNums = copyNums + nums[i];
    if (map.has(copyNums - k)) {
      res += map.get(copyNums - k);
    }
    if (map.has(copyNums)) {
      const count = map.get(copyNums) + 1;
      map.set(copyNums, count);
    } else {
      map.set(copyNums, 1);
```

```
    }
  }
  return res;
};
```

566. 重塑矩阵

题目描述

在MATLAB中，有一个非常有用的函数 reshape，它可以将一个矩阵重塑为另一个大小不同的新矩阵，但保留其原始数据。

给出一个由二维数组表示的矩阵，以及两个正整数r和c，分别表示想要的重构的矩阵的行数和列数。

重构后的矩阵需要将原始矩阵的所有元素以相同的行遍历顺序填充。

如果具有给定参数的reshape操作是可行且合理的，则输出新的重塑矩阵；否则，输出原始矩阵。

例子1
input: nums =

[[1,2],

[3,4]]
r = 1, c = 4
output: [[1,2,3,4]]

例子2
input: nums =

[[1,2],

[3,4]]
r = 2, c = 4
output: [[1,2],[3,4]]

提示：
1 给定矩阵的宽和高范围在 [1, 100]。
2 给定的 r 和 c 都是正数。

思考

1 直接实现就可以了

参考实现1

实现1

```
/**
 * @param {number[][]} nums
 * @param {number} r
 * @param {number} c
 * @return {number[][]}
```

```
 */
// Runtime: 112 ms, faster than 33.95% of JavaScript online submissions for Reshape the Matrix.
// Memory Usage: 43.7 MB, less than 83.95% of JavaScript online submissions for Reshape the Matrix.
export default (nums, r, c) => {
  const m = nums.length;
  const n = nums[0] ? nums[0].length : 0;
  if (r * c !== m * n) {
    return nums;
  }
  const res = [];
  for (let i = 0; i < r; i++) {
    res[i] = new Array(c).fill(0);
  }
  let begin = 0;
  let end = 0;
  for (let i = 0; i < r; i++) {
    for (let j = 0; j < c; j++) {
      if (end < n) {
        res[i][j] = nums[begin][end];
        end++;
      } else {
        begin++;
        end = 0;
```

```
            res[i][j] = nums[begin][end];
            end++;
        }
    }
  }
  return res;
};
```

225. 队列实现栈

题目描述

使用队列实现栈

进阶：

能不能做到平均复杂度O（1）

思考

1 直接实现就可以了,如果想要平均复杂度是O（1），可以在push的时候，时间复杂度是O（n），其他是O（1）

参考实现1

实现1

```
/**
 * Initialize your data structure here.
 */
// Runtime: 76 ms, faster than 67.91% of JavaScript online submissions for Implement Stack using Queues.
// Memory Usage: 38.5 MB, less than 39.93% of JavaScript online submissions for Implement Stack using Queues.
var MyStack = function () {
  this.queue1 = [];
  this.queue2 = [];
};

/**
 * Push element x onto stack.
 * @param {number} x
 * @return {void}
 */
MyStack.prototype.push = function (x) {
  if (this.queue2.length > 0) {
    while (this.queue2.length > 0) {
      this.queue1.push(this.queue2.pop());
    }
```

```
  }
  this.queue2.push(x);
  while (this.queue1.length > 0) {
    this.queue2.push(this.queue1.pop());
  }
};

/**
 * Removes the element on top of the stack and returns that element.
 * @return {number}
 */
MyStack.prototype.pop = function () {
  return this.queue2.shift();
};

/**
 * Get the top element.
 * @return {number}
 */
MyStack.prototype.top = function () {
  return this.queue2[0];
};
```

```
/**
 * Returns whether the stack is empty.
 * @return {boolean}
 */
MyStack.prototype.empty = function () {
  return this.queue2.length === 0;
};

/**
 * Your MyStack object will be instantiated and called as such:
 * var obj = new MyStack()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.top()
 * var param_4 = obj.empty()
 */
```

503. 下一个更大元素 II

题目描述

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

例子1
输入：[1,2,1]
输出：[2,-1,2]
解释: 第一个 1 的下一个更大的数是 2；
数字 2 找不到下一个更大的数；
第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

思考

1 直接使用暴力解法

参考实现1

2 可以使用栈的特性，先入后出的特性，把数组中某个位置前面的下标都存储到栈中，如果发现当前数组中的元素比栈中的大，则从栈中弹出，并且更新结果。

这里有个技巧就是不是遍历数组的长度，而是遍历数组长度的两倍

参考实现2

实现1

```
/**
```

```
 * @param {number[]} nums
 * @return {number[]}
 */
// 1, 2, 1;
// 2, -1, 2;
// Runtime: 228 ms, faster than 17.77% of JavaScript online submissions for Next Greater Element II.
// Memory Usage: 45 MB, less than 57.87% of JavaScript online submissions for Next Greater Element II.
export default (nums) => {
  const res = [];
  for (let i = 0; i < nums.length; i++) {
    let j = i + 1;
    while (j !== i) {
      if (nums[j] <= nums[i] || j > nums.length - 1) {
        if (j > nums.length - 1) {
          j = 0;
        } else {
          j++;
        }
      } else {
        break;
      }
    }
    // console.log(j);
```

```
    if (j === i) {
      res.push(-1);
    } else {
      res.push(nums[j]);
    }
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
// Runtime: 144 ms, faster than 44.16% of JavaScript online submissions for Next Greater Element II.
// Memory Usage: 44.4 MB, less than 76.65% of JavaScript online submissions for Next Greater Element II.
export default (nums) => {
  const len = nums.length;
  const res = new Array(len).fill(-1);
  const stack = [];
```

```
  for (let i = 0; i < 2 * len; i++) {
    while (stack.length > 0 && nums[stack[stack.length - 1]] < nums[i % len]) {
      const index = stack.pop();
      res[index] = nums[i % len];
    }
    stack.push(i % len);
  }
  return res;
};
```

217. 存在重复元素

题目描述

给定一个整数数组，判断是否存在重复元素。

如果存在一值在数组中出现至少两次，函数返回 true 。如果数组中每个元素都不相同，则返回 false 。

例子1
输入：[1,2,3,1]
输出：true

思考

1 很明显使用hash表快速查找

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */

// Runtime: 96 ms, faster than 45.58% of JavaScript online submissions for Contains Duplicate.
// Memory Usage: 45 MB, less than 54.79% of JavaScript online submissions for Contains Duplicate.
export default (nums) => {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    if (map.has(nums[i])) {
      return true;
    } else {
      map.set(nums[i], 1);
    }
  }
```

```
  return false;
};
```

697. 数组的度

题目描述

给定一个非空且只包含非负数的整数数组 nums, 数组的度的定义是指数组里任一元素出现频数的最大值。

你的任务是找到与 nums 拥有相同大小的度的最短连续子数组，返回其长度。

例子1
输入：[1, 2, 2, 3, 1]
输出：2
解释: 输入数组的度是2，因为元素1和2的出现频数最大，均为2.
连续子数组里面拥有相同度的有如下所示:

[1, 2, 2, 3, 1], [1, 2, 2, 3], [2, 2, 3, 1], [1, 2, 2], [2, 2, 3], [2, 2]
最短连续子数组[2, 2]的长度为2，所以返回2.

思考

1 比较简单，直接实现就可以了

参考实现1

2 可以利用firstMap存储数字首次出现的位置,

参考实现2

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 264 ms, faster than 5.02% of JavaScript online submissions for Degree of an Array.
// Memory Usage: 42 MB, less than 91.32% of JavaScript online submissions for Degree of an Array.
export default (nums) => {
  const map = new Map();
  let max = 1;
  for (let i = 0; i < nums.length; i++) {
    if (map.has(nums[i])) {
      const count = map.get(nums[i]) + 1;
      max = Math.max(max, count);
      map.set(nums[i], count);
    } else {
      map.set(nums[i], 1);
```

```
    }
  }
  let res = Number.MAX_VALUE;
  for (let [key, val] of map) {
    if (val === max) {
      let begin = -1;
      let end = -1;
      for (let i = 0; i < nums.length; i++) {
        if (nums[i] === key && begin === -1) {
          begin = i;
          end = i;
        }
        if (nums[i] === key && begin !== -1) {
          end = i;
        }
      }
      res = Math.min(res, end - begin + 1);
    }
  }
  return res;
};
```

实现2

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 104 ms, faster than 50.23% of JavaScript online submissions for Degree of an Array.
// Memory Usage: 43.8 MB, less than 45.21% of JavaScript online submissions for Degree of an Array.
export default (nums) => {
  const map = new Map();
  const firstMap = new Map();
  let res = 0;
  let degree = 0;
  for (let i = 0; i < nums.length; ++i) {
    if (!firstMap.has(nums[i])) {
      firstMap.set(nums[i], i);
    }

    if (map.has(nums[i])) {
      const count = map.get(nums[i]) + 1;
      map.set(nums[i], count);
    } else {
```

```
      map.set(nums[i], 1);
    }
    if (map.get(nums[i]) > degree) {
      degree = map.get(nums[i]);
      res = i - firstMap.get(nums[i]) + 1;
    } else if (map.get(nums[i]) === degree) {
      res = Math.min(res, i - firstMap.get(nums[i]) + 1);
    }
  }
  return res;
};
```

594. 最长和谐子序列

题目描述

和谐数组是指一个数组里元素的最大值和最小值之间的差别正好是1。

现在，给定一个整数数组，你需要在所有可能的子序列中找到最长的和谐子序列的长度。

例子1
输入：[1,3,2,2,5,2,3,7]

输出：5
解释: 最长的和谐数组是：[3,2,2,2,3].

思考

1 很简单，直接使用hash就可以

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 112 ms, faster than 92.50% of JavaScript online submissions for Longest Harmonious Subsequence.
// Memory Usage: 48.4 MB, less than 51.88% of JavaScript online submissions for Longest Harmonious Subsequence.
export default (nums) => {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    if (map.has(nums[i])) {
      const count = map.get(nums[i]) + 1;
      map.set(nums[i], count);
    } else {
```

```
      map.set(nums[i], 1);
    }
  }
  let max = 0;
  for (let [key, val] of map) {
    const tempMax = map.get(key + 1) ? map.get(key + 1) : 0;
    const tempMin = map.get(key - 1) ? map.get(key - 1) : 0;
    if (tempMax > 0) {
      max = Math.max(max, val + tempMax);
    }
    if (tempMin > 0) {
      max = Math.max(max, val + tempMin);
    }
  }
  return max;
};
```

287. 寻找重复数

题目描述

给定一个包含 n + 1 个整数的数组 nums ， 其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。

假设 nums 只有 一个重复的整数 ， 找出 这个重复的数 。

例子1
输入： nums = [1,3,4,2,2]
输出： 2
解释: 2是重复数

思考

1 很简单，直接使用hash就可以

参考实现1

实现1

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Runtime: 68 ms, faster than 99.68% of JavaScript online submissions for Find the Duplicate Number.
// Memory Usage: 41.1 MB, less than 24.13% of JavaScript online submissions for Find the Duplicate Number.
export default (nums) => {
  const map = new Map();
```

```
  for (let i = 0; i < nums.length; i++) {
    if (map.has(nums[i])) {
      return nums[i];
    } else {
      map.set(nums[i], 1);
    }
  }
};
```

313. 超级丑数

题目描述

编写一段程序来查找第 n 个超级丑数。

超级丑数是指其所有质因数都是长度为 k 的质数列表 primes 中的正整数。

假设 nums 只有 一个重复的整数 ， 找出 这个重复的数 。

例子1
输入：n = 12, primes = [2,7,13,19]
输出：32
解释: 给定长度为 4 的质数列表 primes = [2,7,13,19]，前 12 个超级丑数序列为：[1,2,4,7,8,13,14,16,19,26,28,32] 。

说明：
1 是任何给定 primes 的超级丑数。
2 给定 primes 中的数字以升序排列。
3 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000 。
4 第 n 个超级丑数确保在 32 位有符整数范围内。

思考

1 首先需要理解这里的超级丑数是什么，就是从primes中选择所有数字和前面的超级丑数进行相乘，选择最小的并且不能重复前面的超级丑数

所以可以设置一个和primes相同长度的数组，记录每个primes 对应的超级丑数，防止得到的超级丑数重复

参考实现1

实现1

```
/**
 * @param {number} n
 * @param {number[]} primes
 * @return {number}
 */
// (12)[(2, 7, 13, 19)];
// Runtime: 108 ms, faster than 75.00% of JavaScript online submissions for Super Ugly Number.
// Memory Usage: 41.7 MB, less than 87.50% of JavaScript online submissions for Super Ugly Number.
export default (n, primes) => {
```

```
  const ugly = new Array(n).fill(0);
  const primes2uglyIndexs = new Array(primes.length).fill(0);
  ugly[0] = 1;
  for (let i = 1; i < n; i++) {
    ugly[i] = Number.MAX_VALUE;
    for (let j = 0; j < primes.length; j++) {
      if (primes[j] * ugly[primes2uglyIndexs[j]] < ugly[i]) {
        ugly[i] = primes[j] * ugly[primes2uglyIndexs[j]];
      }
    }

    for (let j = 0; j < primes.length; j++) {
      if (primes[j] * ugly[primes2uglyIndexs[j]] === ugly[i]) {
        primes2uglyIndexs[j]++;
      }
    }
  }
  return ugly[n - 1];
};
```

870. 优势洗牌

题目描述

给定两个大小相等的数组 A 和 B，A 相对于 B 的优势可以用满足 A[i] > B[i] 的索引 i 的数目来描述。

返回 A 的任意排列，使其相对于 B 的优势最大化。

假设 nums 只有 一个重复的整数 ，找出 这个重复的数 。

例子1
输入：A = [2,7,11,15], B = [1,10,4,11]
输出：[2,11,7,15]
解释:

例子2
输入：A = [12,24,8,32], B = [13,25,32,11]
输出：[2,11,7,15]
解释:

思考

1 题目的意思是尽可能的得到更多的分数，而一分就是A[i] > B[i], 所以题目就是求出A中尽可能多的比B同位置大的组合

可以使用贪心算法，如果A中的最大数比B中的最大数，可以使用，如果小于，那么说明A中其他数字也肯定小于B，所以直接使用A中最小的数来对应B中最大的数，类似田忌赛马

参考实现1

实现1

```
/**
 * @param {number[]} A
 * @param {number[]} B
 * @return {number[]}
 */

export default (A, B) => {
  // 按照索引排序数组
  const idxs = B.map((v, i) => i).sort((a, b) => B[b] - B[a]);

  A.sort((a, b) => b - a);
  const res = [];
  for (let i = 0; i < B.length; i++) {
    // 使用田忌赛马，下等马对上上等马
    res[idxs[i]] = A[0] > B[idxs[i]] ? A.shift() : A.pop();
  }
  return res;
};
```

307. 区域和检索 - 数组可修改

题目描述

给你一个数组 nums ，请你完成两类查询，其中一类查询要求更新数组下标对应的值，另一类查询要求返回数组中某个范围内元素的总和。

实现 NumArray 类：

```
NumArray(int[] nums) 用整数数组 nums 初始化对象
void update(int index, int val) 将 nums[index] 的值更新为 val
int sumRange(int left, int right) 返回子数组 nums[left, right] 的总和（即，nums[left] + nums[left + 1], ..., nums[right]）
```

假设 nums 只有 一个重复的整数 ，找出 这个重复的数 。

例子1
输入：["NumArray", "sumRange", "update", "sumRange"]

[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]

输出：[null, 9, null, 8]
解释: NumArray numArray = new NumArray([1, 3, 5]);

numArray.sumRange(0, 2); // 返回 9 ，
sum([1,3,5]) = 9

numArray.update(1, 2); // nums = [1,2,5]

numArray.sumRange(0, 2); // 返回 8 ，
sum([1,2,5]) = 9

思考

1 直接使用正常解法，时间复杂度是O（n），发现时间超时

参考实现1

2 使用线段树, 线段树也比较简单，主要是需要注意边界条件，在什么情况下退出，比如更新的时候。

参考实现2

实现1

```
/**
 * @param {number[]} nums
 */
var NumArray = function (nums) {
  this.nums = nums;
};

/**
 * @param {number} index
```

```
 * @param {number} val
 * @return {void}
 */
NumArray.prototype.update = function (index, val) {
  this.nums[index] = val;
};

/**
 * @param {number} left
 * @param {number} right
 * @return {number}
 */
NumArray.prototype.sumRange = function (left, right) {
  let sum = 0;
  for (let i = left; i <= right; i++) {
    sum += this.nums[i];
  }
  return sum;
};

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)
```

```
 * obj.update(index,val)
 * var param_2 = obj.sumRange(left,right)
 */
```

实现2

```
/**
 * @param {number[]} nums
 */
var NumArray = function (nums) {
  this.nums = nums;
  this.tree = [];
  this.build(0, nums.length - 1, 0);
};

/**
 * @param {number} index
 * @param {number} val
 * @return {void}
 */
NumArray.prototype.update = function (index, val) {
```

```
  const diff = val - this.nums[index];
  this.nums[index] = val;
  this.updateUtil(0, this.nums.length - 1, 0, index, diff);
};
NumArray.prototype.updateUtil = function (left, right, treeIdx, index, diff) {
  if (index >= left && index <= right) {
    this.tree[treeIdx] += diff;
    if (left === right) return;
    var mid = left + ((right - left) >> 1);
    this.updateUtil(left, mid, treeIdx * 2 + 1, index, diff);
    this.updateUtil(mid + 1, right, treeIdx * 2 + 2, index, diff);
  }
};

/**
 * @param {number} left
 * @param {number} right
 * @return {number}
 */
NumArray.prototype.sumRange = function (left, right) {
  return this.sumUtil(left, right, 0, this.nums.length - 1, 0);
};
NumArray.prototype.sumUtil = function (left, right, currLeft, currRight, treeIdx) {
```

```
  if (left > currRight || right < currLeft) return 0;
  if (left <= currLeft && right >= currRight) return this.tree[treeIdx];
  const mid = currLeft + ((currRight - currLeft) >> 1);
  return (
    this.sumUtil(left, right, currLeft, mid, treeIdx * 2 + 1) +
    this.sumUtil(left, right, mid + 1, currRight, treeIdx * 2 + 2)
  );
};
NumArray.prototype.build = function (left, right, idx) {
  if (left > right) return;
  const mid = left + ((right - left) >> 1);
  // 递归建立线段树
  const sum =
    left === right ? this.nums[left] : this.build(left, mid, idx * 2 + 1) + this.build(mid + 1, right, idx * 2 + 2);
  this.tree[idx] = sum;
  return sum;
};

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)
 * obj.update(index,val)
 * var param_2 = obj.sumRange(left,right)
```

```
 */
```

第12章必知必会之字符串

字符串

字符串是最常用的一种数据类型，也是面试中经常必会问题的类型之一。

242. 有效的字母异位词

题目描述

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

例子1

Input: s = "anagram", t = "nagaram"

output: true

例子2

Input: s = "rat", t = "car"

output: false

提示：
你可以假设字符串只包含小写字母。

进阶：
如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

思考

1 直接使用一个数组记录字符出现在s中的次数，然后遍历t每个字符相减就可以了。

至于如果存在unicode字符，可以先把unicode转换成字符就可以了

参考实现1

实现1

```
/**
 * @param {string} s
 * @param {string} t
 * @return {boolean}
 */
// Runtime: 88 ms, faster than 90.79% of JavaScript online submissions for Valid Anagram.
// Memory Usage: 39.8 MB, less than 93.22% of JavaScript online submissions for Valid Anagram.
export default (s, t) => {
  const arr = new Array(26).fill(0);
```

```
  for (let i = 0; i < s.length; i++) {
    arr[s.charCodeAt(i) - 97]++;
  }

  for (let i = 0; i < t.length; i++) {
    if (arr[t.charCodeAt(i) - 97] >= 1) {
      arr[t.charCodeAt(i) - 97]--;
    } else {
      return false;
    }
  }
  return arr.reduce((a, b) => a + b) === 0;
};
```

205. 同构字符串

题目描述

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以按某种映射关系替换得到 t ，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，

字符可以映射到自己本身。

例子1

Input: s = "egg", t = "add"

output: true

例子2

Input: s = "foo", t = "bar"

output: false

例子3

Input: s = "paper", t = "title"

output: true

提示：
假设s的长度和t的长度相同

思考

1 这里可以直接转换一下，依次遍历字符串，如果s和t中相同位置的字符在前面出现的位置是一样的，那字符串s和字符串t就是同构的。

参考实现1

实现1

```javascript
/**
 * @param {string} s
 * @param {string} t
 * @return {boolean}
 */
// Runtime: 88 ms, faster than 73.09% of JavaScript online submissions for Isomorphic Strings.
// Memory Usage: 39.8 MB, less than 69.51% of JavaScript online submissions for Isomorphic Strings.
export default (s, t) => {
  const s_first_index = new Map();
  const t_first_index = new Map();
  for (let i = 0; i < s.length; i++) {
    if (s_first_index.get(s.charAt(i)) !== t_first_index.get(t.charAt(i))) {
      return false;
    }
    s_first_index.set(s.charAt(i), i + 1);
    t_first_index.set(t.charAt(i), i + 1);
  }
  return true;
};
```

647. 回文子串

题目描述

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

例子1

Input: "abc"

output: 3

解释：三个回文子串: "a", "b", "c"

例子2

Input: "aaa"

output: 6
解释：6个回文子串: "a", "a", "a", "aa", "aa", "aaa"

提示：
输入的字符串长度不会超过 1000 。

思考

1 这里很明显可以使用暴力求解，分别求出不同长度的回文字符串长度，从1到s.length

参考实现1

2 发现时间复杂度太高了，想使用空间换时间，但是发现好像时间更高

参考实现2

3 后来发现回文字符串的特点，可以使用从中间从两边扩散的方法，但是刚开始使用这种方法的时候，忽略了偶数情况下向外扩散的情况，只是考虑到了奇数扩散的情况。

所以当使用这种从中间向外扩散的方法的时候，必须同时考略使用奇数和偶数两种情况向外扩散

参考实现3

4 方法3还可以写的比较简洁一些

参考实现4

实现1

```
/**
 * @param {string} s
 * @return {number}
 */
// Runtime: 312 ms, faster than 27.56% of JavaScript online submissions for Palindromic Substrings.
// Memory Usage: 37.6 MB, less than 100.00% of JavaScript online submissions for Palindromic Substrings.
```

```
export default (s) => {
  let count = s.length;
  for (let i = 2; i <= s.length; i++) {
    for (let j = 0; j < s.length - i + 1; j++) {
      let k = j + i - 1;
      let m = j;
      while (m < k) {
        if (s.charAt(m) === s.charAt(k)) {
          m++;
          k--;
        } else {
          break;
        }
      }
      if (m >= k) {
        count++;
      }
    }
  }
  return count;
};
```

实现2

```
/**
 * @param {string} s
 * @return {number}
 */
// Runtime: 600 ms, faster than 16.13% of JavaScript online submissions for Palindromic Substrings.
// Memory Usage: 79.4 MB, less than 5.04% of JavaScript online submissions for Palindromic Substrings.
export default (s) => {
  let count = s.length;
  const map = new Map();
  for (let i = 0; i < s.length; i++) {
    map.set(`${i}_${i}`, 1);
  }
  for (let i = 2; i <= s.length; i++) {
    for (let j = 0; j < s.length - i + 1; j++) {
      let k = j + i - 1;
      let m = j;
      const begin = j;
      const end = k;
      if (begin >= 1 && end >= 1 && map.get(`${begin - 1}_${end - 1}`) === 1 && s[begin] === s[end]) {
        count++;
```

```
        map.set(`${begin}_${end}`, 1);
      } else {
        while (m < k) {
          if (s.charAt(m) === s.charAt(k)) {
            m++;
            k--;
          } else {
            break;
          }
        }
        if (m >= k) {
          count++;
          map.set(`${begin}${end}`, 1);
        }
      }
    }
  }
  return count;
};
```

实现3

```javascript
/**
 * @param {string} s
 * @return {number}
 */
// Runtime: 100 ms, faster than 58.99% of JavaScript online submissions for Palindromic Substrings.
// Memory Usage: 40.3 MB, less than 48.07% of JavaScript online submissions for Palindromic Substrings.
export default (s) => {
  let count = 0;
  for (let i = 0; i < s.length; i++) {
    let begin = i;
    let end = i;
    while (begin >= 0 && end <= s.length && s.charAt(begin) === s.charAt(end)) {
      count++;
      begin--;
      end++;
    }

    begin = i;
    end = i + 1;
    while (begin >= 0 && end <= s.length && s.charAt(begin) === s.charAt(end)) {
      count++;
      begin--;
      end++;
```

```
    }
  }
  return count;
};
```

实现4

```
/**
 * @param {string} s
 * @return {number}
 */

// Runtime: 80 ms, faster than 95.97% of JavaScript online submissions for Palindromic Substrings.
// Memory Usage: 39.3 MB, less than 80.17% of JavaScript online submissions for Palindromic Substrings.
// "aaaaa"

// 0 2   00 01
// 1 6   11 02 12 03
// 2 11  22 13 04 23 14
// 3 14  33 24 34
// 4 15  44 45
```

```
const extendSubstrings = (s, begin, end) => {
  let count = 0;
  while (begin >= 0 && end < s.length && s[begin] === s[end]) {
    --begin;
    ++end;
    ++count;
  }
  return count;
};
export default (s) => {
  let count = 0;
  for (let i = 0; i < s.length; i++) {
    count += extendSubstrings(s, i, i); // 奇数长度
    count += extendSubstrings(s, i, i + 1); // 偶数长度
  }
  return count;
};
```

n
时间复杂度O（n * n），空间复杂度O（1）

696. 计数二进制子串

题目描述

给定一个字符串 s，计算具有相同数量0和1的非空(连续)子字符串的数量，并且这些子字符串中的所有0和所有1都是组合在一起的。

重复出现的子串要计算它们出现的次数。

例子1

Input: "00110011"

output: 6

解释：有6个子串具有相同数量的连续1和0：“0011”，“01”，“1100”，“10”，“0011” 和 “01”。

请注意，一些重复出现的子串要计算它们出现的次数。

另外，“00110011”不是有效的子串，因为所有的0（和1）没有组合在一起。

例子2

Input: "10101"

output: 4
解释：有4个子串：“10”，“01”，“10”，“01”，它们具有相同数量的连续1和0。

提示：
1 s.length 在1到50,000之间。

2 s 只包含“0”或“1”字符。

思考

1 这里很明显可以使用暴力求解，但是超时了

参考实现1

2 通过观察可以发现，当前面相同的字符数量大于等于后面的字符数量的时刻，那肯定存在一个子字符串符合要求，所以可以利用此思想遍历一遍就可以了

参考实现2

实现1

```
/**
 * @param {string} s
 * @return {number}
 */

export default (s) => {
  let count = 0;
  for (let i = 0; i < s.length; i++) {
    let count0 = 0;
    let j = i;
```

```
    let flag = s.charAt(j);
    while (s.charAt(j) === flag) {
      j++;
      count0++;
    }
    // console.log(i,count0)
    flag = flag === "0" ? "1" : "0";
    while (j + count0 < s.length && s.charAt(j) === flag) {
      j++;
      count0--;
      if (count0 === 0) {
        count++;
        break;
      }
    }
  }
  return count;
};
```

实现2

```javascript
/**
 * @param {string} s
 * @return {number}
 */
//  Runtime: 84 ms, faster than 92.50% of JavaScript online submissions for Count Binary Substrings.
//  Memory Usage: 42.2 MB, less than 63.75% of JavaScript online submissions for Count Binary Substrings.
export default (s) => {
  let pre = 0;
  let cur = 1;
  let count = 0;
  for (let i = 1; i < s.length; ++i) {
    if (s[i] === s[i - 1]) {
      ++cur;
    } else {
      pre = cur;
      cur = 1;
    }
    if (pre >= cur) {
      ++count;
    }
  }
  return count;
```

```
};
```

时间复杂度O（n），空间复杂度O（1）

227. 基本计算器 II

题目描述

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式仅包含非负整数，+， - ，*， / 四种运算符和空格 。 整数除法仅保留整数部分。

例子1

Input: "3+2*2"

output: 7

解释：

例子2

Input: " 3/2 "
output: 1
解释：

例子3

Input: " 3+5 / 2 "
output: 5
解释：

提示：
1 你可以假设所给定的表达式都是有效的。
2 请不要使用内置的库函数 eval。

思考

1 这里如果不是使用一些小技巧，其实还是非常复杂的

有几个问题需要考虑如何处理

1.1 如何处理输入字符串中的空格?

1.2 如何处理输入字符串中的连续的数字，比如输入"234/3+2"的时候，应该如何得到234?

1.3 如何处理输入的顺序，题解中是在输入字符串中前面添加了一个+号，这样输入字符串“23+1”就变成了“+23+1”，可以想下这样有什么好处，如果不这样搞，应该如何处理?

参考实现1

实现1

```javascript
/**
 * @param {string} si
 * @return {number}
 */
// Runtime: 100 ms, faster than 78.72% of JavaScript online submissions for Basic Calculator II.
// Memory Usage: 46.9 MB, less than 32.83% of JavaScript online submissions for Basic Calculator II.
export default (s) => {
  s = s.replace(/\s+/g, "");
  let len = s.length;
  if (!s || len === 0) return 0;
  const stack = [];
  let num = 0;
  let opr = "+";
  for (let i = 0; i < len; i++) {
    if (/^[0-9]+.?[0-9]*$/.test(s.charAt(i))) {
      num = num * 10 + s.charCodeAt(i) - 48;
    }
    if (!/^[0-9]+.?[0-9]*$/.test(s.charAt(i)) || i === len - 1) {
      if (opr === "-") {
        stack.push(-num);
      }
      if (opr === "+") {
```

```
        stack.push(num);
      }
      if (opr === "*") {
        stack.push(stack.pop() * num);
      }
      if (opr === "/") {
        const tempNum = stack.pop();
        stack.push(tempNum > 0 ? Math.floor(tempNum / num) : Math.ceil(tempNum / num));
      }
      opr = s.charAt(i);
      num = 0;
    }
  }

  // let res = 0;
  // console.log(stack);

  return stack.reduce((a, b) => a + b);
};
```

时间复杂度O（n），空间复杂度O（1）

227. 基本计算器 III

题目描述

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式仅包含非负整数，+， - ，*， / , (,) 等各种运算符和空格 。 整数除法仅保留整数部分。

例子1

Input: " 6-4 / 2 "

output: 4

解释：

例子2

Input: "2*(5+5*2)/3+(6/2+8)"
output: 21
解释：

例子3

Input: "(2+6* 3+5- (3*14/7+2)*5)+3"
output: -12
解释：

提示：
1 你可以假设所给定的表达式都是有效的。
2 请不要使用内置的库函数 eval。

思考

1 这里相比227，主要区别就是如何想办法处理掉左右括号?

1.1 如果找到办法处理左右括号，那么就可以很容易解决该问题了，可是如何处理这里的左右括号呢?

参考实现1

实现1

```
const calculate = (s) => {
  s = s.replace(/\s+/g, "");
  const sLen = s.length;
  // 当前的操作数
  let num = 0;

  const stack = [];
  // 操作符，第一个加上"+
  let opr = "+";
  for (let i = 0; i < sLen; ++i) {
    const c = s.charAt(i);
```

```
  if (/^[0-9]+.?[0-9]*$/.test(s.charAt(i))) {
    num = num * 10 + s.charCodeAt(i) - 48;
  } else if (c === "(") {
    let j = i;
    let matchCount = 0;
    for (; i < sLen; ++i) {
      if (s.charAt(i) === "(") ++matchCount;
      if (s.charAt(i) === ")") --matchCount;
      if (matchCount === 0) break;
    }
    num = calculate(s.substring(j + 1, i));
  }
  if (c === "+" || c === "-" || c === "*" || c === "/" || i === sLen - 1) {
    if (opr === "-") {
      stack.push(-num);
    }
    if (opr === "+") {
      stack.push(num);
    }
    if (opr === "*") {
      stack.push(stack.pop() * num);
    }
    if (opr === "/") {
```

```
        const tempNum = stack.pop();
        stack.push(tempNum > 0 ? Math.floor(tempNum / num) : Math.ceil(tempNum / num));
      }
      opr = c;
      num = 0;
    }
  }
  return stack.reduce((a, b) => a + b);
};
export default calculate;
```

28. 实现 strStr()

题目描述

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

例子1

Input: haystack = "hello", needle = "ll"

output: 2

解释：

例子2

Input: haystack = "aaaaa", needle = "bba"
output: -1
解释：

提示：
1 当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。
2 对于本题而言，当 needle 是空字符串时我们应当返回 0 。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

思考

1 第一种很容易想到，直接使用暴力解法

参考实现1

2 这是特别的典型使用kmp算法来解决的问题。

kmp算法其实也很简单，就是当发现不匹配的时候，如何更快的把子串移动到最大距离

kmp算法最难的就是next数组，next数组存储的就是当前字符前面的字符串中前后互相相等的长度减一。

求next数组就是使用dp来求

只要得出next数组，其他就简单了

实现1

```
/**
 * @param {string} haystack
 * @param {string} needle
 * @return {number}
 */
// Runtime: 4132 ms, faster than 5.02% of JavaScript online submissions for Implement strStr().
// Memory Usage: 40 MB, less than 27.63% of JavaScript online submissions for Implement strStr().
export default (haystack, needle) => {
  if (!needle) return 0;
  const len = needle.length;
  for (let i = 0; i < haystack.length; i++) {
    if (haystack.charAt(i) === needle.charAt(0)) {
      let begin = i;
      let needleBegin = 0;
      while (haystack.charAt(begin) === needle.charAt(needleBegin)) {
        begin++;
        needleBegin++;
        if (needleBegin === len) {
          return i;
        }
```

```
    }
   }
  }
  return -1;
};
```

时间复杂度O（m+n），空间复杂度O（m）

409. 最长回文串

题目描述

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

例子1

Input: haystack = "hello", needle = "ll"

output: 2

解释：

例子2

Input: "abccccdd"
output: 7
解释：我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

提示：
1 假设字符串的长度不会超过 1010。

思考

1 直接使用hash，根据回文字符串中只有一个字符出现一次，其他字符都是偶数次进行构造

参考实现1

实现1

```
/**
 * @param {string} s
 * @return {number}
 */
// Runtime: 76 ms, faster than 96.49% of JavaScript online submissions for Longest Palindrome.
// Memory Usage: 40 MB, less than 62.11% of JavaScript online submissions for Longest Palindrome.
export default (s) => {
  const map = new Map();
  let count = 0;
  for (let i = 0; i < s.length; i++) {
```

```
    if (map.has(s.charAt(i))) {
      const tempCount = map.get(s.charAt(i)) + 1;
      map.set(s.charAt(i), tempCount);
    } else {
      map.set(s.charAt(i), 1);
    }
  }

  // console.log(map)
  for (let [key, val] of map) {
    if (val % 2 === 0) {
      count += val;
    } else if (val % 2 !== 0) {
      count += val - 1;
    }
  }
  if (count < s.length) {
    return count + 1;
  } else {
    return count;
  }
};
```

时间复杂度O（n），空间复杂度O（n）

3. 无重复字符的最长子串

题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

例子1

Input: s = "abcabcbb"

output: 3

解释：因为无重复字符的最长子串是 "abc"，所以其长度为 3。

例子2

Input: s = "bbbbb"
output: 1
解释：因为无重复字符的最长子串是 "b"，所以其长度为 1。

例子3

Input: s = "pwwkew"
output: 3
解释：因为无重复字符的最长子串是 "wke"，所以其长度为 3。

请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

例子4

Input: s = ""
output: 0
解释：

提示：
1 0 <= s.length <= 5 * 104
2 s 由英文字母、数字、符号和空格组成

思考

1 使用类似dp的方法，dp[i]表示以s[i]结尾的最大的含有不重复字符的最大长度，那么dp[i+1]就等于从s[i+1]往前移动dp[i]个字符，发现是否有和s[i+1]相等的字符，如果没有，，则dp[i+1]=dp[i]+1,否则等于dp[i+1] = i-j(j是和s[i+1]相等的字符的位置)

参考实现1

2 可以使用hash存储每个字符的位置，然后使用类似双指针的思想来解决。

2.1 这里需要解决的是什么时候更新begin指针，如何更新?

参考实现2

实现1

```javascript
/**
 * @param {string} s
 * @return {number}
 */
// Runtime: 112 ms, faster than 76.90% of JavaScript online submissions for Longest Substring Without Repeating Characters.
// Memory Usage: 41.3 MB, less than 90.39% of JavaScript online submissions for Longest Substring Without Repeating Characters.
export default (s) => {
  if (!s) return 0;
  const dp = [];
  let max = 1;
  dp[0] = 1;
  for (let i = 1; i < s.length; i++) {
    let count = 1;
    for (let j = i - 1; j >= i - dp[i - 1]; j--) {
      if (s.charAt(j) === s.charAt(i)) {
        dp[i] = i - j;
        max = Math.max(i - j, max);
        break;
      }
      if (j === i - dp[i - 1]) {
        dp[i] = dp[i - 1] + 1;
        max = Math.max(dp[i], max);
```

```
      }
    }
  }
  return max;
};
```

实现2

```
// Runtime: 108 ms, faster than 83.81% of JavaScript online submissions for Longest Substring Without Repeating Characters.
// Memory Usage: 41.9 MB, less than 86.43% of JavaScript online submissions for Longest Substring Without Repeating Characters.
export default (s) => {
  if (!s) return 0;
  const map = new Map();
  let max = 1;
  let begin = 0;
  for (let i = 0; i < s.length; i++) {
    if (map.has(s.charAt(i))) {
      const iIndex = map.get(s.charAt(i));
      if (begin <= iIndex) {
        max = Math.max(i - map.get(s.charAt(i)), max);
        begin = map.get(s.charAt(i)) + 1;
```

```
        map.set(s.charAt(i), i);
      } else {
        max = Math.max(i - begin + 1, max);
        map.set(s.charAt(i), i);
      }
    } else {
      max = Math.max(i - begin + 1, max);
      map.set(s.charAt(i), i);
    }
  }
  // console.log(max);
  return max;
};
```

时间复杂度O（n）空间复杂度O（n）

5. 最长回文字符串

题目描述

给定一个字符串，找出最长的回文字符串

例子1

Input: s = "babad"

output: "bab"

解释：因为无重复字符的最长子串是 "abc"，所以其长度为 3。

例子2

Input: s = "cbbd"
output: “bb”
解释：

例子3

Input: s = "a"
output: “a”
解释：

例子4

Input: s = "ac"
output: “a”
解释：

提示：
1 1 <= s.length <= 1000
2 s 由英文小写字母或者英文大写字母组成

思考

1 使用dp很好解决，只不过这里的dp是从长度等于1一直到长度等于字符串长度开始

参考实现1

2 可以使用马拉车算法。

马拉车算法的关键点是什么呢?

就是我们想利用前面已经匹配到的信息，关键点就是center到左右maxRight是回文字符串，充分利用回文字符串的特性，什么特性呢，就是左右对称。

马拉车算法和kmp算法思想是类似的，都是利用前面已经匹配的，从而快速的移动后面的

参考实现2

实现1

```
/**
 * @param {string} s
 * @return {string}
 */

// "babad";
// "cbbd";
```

```
// aacabdkacaa;
export default (s) => {
  const len = s.length;
  const dp = [];

  if (s.length <= 1) {
    return s;
  }
  for (let i = 0; i < len; i++) {
    dp[i] = new Array(len).fill(false);
    dp[i][i] = true;
  }
  let begin = 0;
  let end = 1;

  for (let len1 = 2; len1 <= len; len1++) {
    for (let j = 0; j < len + 1 - len1; j++) {
      if (s.charAt(j) === s.charAt(j + len1 - 1)) {
        if (len1 === 2 || (dp[j + 1][j + len1 - 2] && j <= j + len1 - 3)) {
          dp[j][j + len1 - 1] = true;
          if (len1 > end - begin) {
            begin = j;
            end = j + len1;
```

```
        }
      }
    }
  }
 }
 return s.substring(begin, end);
};
```

实现2

```
/**
 * @param {string} s
 * @return {string}
 */

export default (s) => {
 const len = s.length;
 const dp = [];

 if (s.length <= 1) {
   return s;
```

```
  }
  for (let i = 0; i < len; i++) {
    dp[i] = new Array(len).fill(false);
    dp[i][i] = true;
  }
  let begin = 0;
  let end = 1;

  for (let len1 = 2; len1 <= len; len1++) {
    for (let j = 0; j < len + 1 - len1; j++) {
      if (s.charAt(j) === s.charAt(j + len1 - 1)) {
        if (len1 === 2 || (dp[j + 1][j + len1 - 2] && j <= j + len1 - 3)) {
          dp[j][j + len1 - 1] = true;
          if (len1 > end - begin) {
            begin = j;
            end = j + len1;
          }
        }
      }
    }
  }
  return s.substring(begin, end);
};
```

时间复杂度O（n）空间复杂度O（n）

第13章啊哈，链表

链表

链表常用技巧大多数为了好立即，最好是画图。

206. 反转链表

题目描述

反转一个单链表。

例子1

Input: 1->2->3->4->5->NULL

output: 5->4->3->2->1->NULL

进阶：
进阶:
你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

思考

1 比较简单，直接反转就可以

参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */

// Input: 1->2->3->4->5->NULL
// Output: 5->4->3->2->1->NULL

// Runtime: 84 ms, faster than 67.00% of JavaScript online submissions for Reverse Linked List.
// Memory Usage: 40.6 MB, less than 37.61% of JavaScript online submissions for Reverse Linked List.
```

```
const resver = (head, newHead) => {
  if (head === null) {
    return newHead;
  }
  const temp1 = head.next;
  head.next = newHead;
  newHead = head;
  head = temp1;
  return resver(head, newHead);
};

var reverseList = function (head) {
  let newHead = null;
  return resver(head, newHead);
};
export default (head) => {
  let newHead = null;
  //return resver(head, newHead);
  while (head) {
    const temp1 = head.next;
    head.next = newHead;
    newHead = head;
```

```
    head = temp1;
  }
  return newHead;
};
```

21. 合并两个有序链表

题目描述

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

例子1

Input: l1 = [1,2,4], l2 = [1,3,4]

output: [1,1,2,3,4,4]

例子2

Input: l1 = [], l2 = []

output: []

思考

1 比较简单

参考实现1

2 递归实现

参考实现2

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
function ListNode(val, next) {
  this.val = val === undefined ? 0 : val;
  this.next = next === undefined ? null : next;
```

```
}
// Runtime: 84 ms, faster than 92.56% of JavaScript online submissions for Merge Two Sorted Lists.
// Memory Usage: 40.8 MB, less than 10.66% of JavaScript online submissions for Merge Two Sorted Lists.
export default (l1, l2) => {
  let newHead = new ListNode();
  let head = newHead;
  while (l1 && l2) {
    let val;
    if (l1.val <= l2.val) {
      val = l1.val;
      l1 = l1.next;
    } else {
      val = l2.val;
      l2 = l2.next;
    }
    const temp = new ListNode(val);
    newHead.next = temp;
    newHead = temp;
  }
  if (l1) {
    newHead.next = l1;
  }
  if (l2) {
```

```
    newHead.next = l2;
  }
  return head.next;
};
```

实现2

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
function ListNode(val, next) {
  this.val = val === undefined ? 0 : val;
```

```
  this.next = next === undefined ? null : next;
}

const rescur = (l1, l2, newHead) => {
  if (l1 && l2) {
    let val;
    if (l1.val <= l2.val) {
      val = l1.val;
      l1 = l1.next;
    } else {
      val = l2.val;
      l2 = l2.next;
    }
    const temp = new ListNode(val);
    newHead.next = temp;
    newHead = temp;
    rescur(l1, l2, newHead);
  }
  if (l1 && !l2) {
    newHead.next = l1;
  }
  if (l2 && !l1) {
    newHead.next = l2;
```

```
  }
};

export default (l1, l2) => {
  let newHead = new ListNode();
  let head = newHead;
  rescur(l1, l2, newHead);
  return head.next;
};
```

24. 两两交换链表中的节点

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。
你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

例子1

Input: head = [1,2,3,4]

output: [2,1,4,3]

例子2

Input: l1 = [], l2 = []

output: []

思考

1 比较简单

参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */

function ListNode(val, next) {
```

```
  this.val = val === undefined ? 0 : val;
  this.next = next === undefined ? null : next;
}
const rescrue = (pre, head) => {
  if (head) {
    let next = head.next;
    if (!next) {
      return;
    }
    pre.next = next;
    head.next = next.next;
    next.next = head;
    rescrue(head, head.next);
  }
};
// Runtime: 76 ms, faster than 79.57% of JavaScript online submissions for Swap Nodes in Pairs.
// Memory Usage: 39 MB, less than 12.95% of JavaScript online submissions for Swap Nodes in Pairs.
export default (head) => {
  if (!head || !head.next) {
    return head;
  }
  let pre = new ListNode();
  const currentHead = pre;
```

```
  rescrue(pre, head);
  return pre.next;
};
```

160. 相交链表

题目描述

编写一个程序，找到两个单链表相交的起始节点。

例子1

Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

output: 8

提示：
1 不能修改链表的结构

思考

1 本来想先反转A和B链表，然后从反转之后的开头开始寻找，但是提示不能修改链表结构，所以不行。

后来想起来龟兔赛跑，好像不能用

后来看了下题解，原来是使用两个指针，分别遍历A和B两个链表，当一个指针走到一个链表的末尾之后，指针重新指向另外一个链表的开头，这样到最后两个指针走过的距离就是相同的。

参考实现1

实现1

```
// Runtime: 128 ms, faster than 18.76% of JavaScript online submissions for Intersection of Two Linked Lists.
// Memory Usage: 46 MB, less than 77.97% of JavaScript online submissions for Intersection of Two Linked Lists.
export default (headA, headB) => {
  let copyA = headA;
  let copyB = headB;

  let count = 0;
  while (headA && headB && count < 3) {
    if (headA === headB) {
      return headA;
    }
    headA = headA.next;
    headB = headB.next;
    if (!headA) {
      headA = copyB;
```

```
      ++count;
    }
    if (!headB) {
      headB = copyA;
      ++count;
    }
  }

  return null;
};
```

时间复杂度O（m+n） 空间复杂度O（1）

234. 回文链表

题目描述

请判断一个链表是否为回文链表。

例子1
Input: 1->2

output: false

例子2
Input: 1->2->2->1
output: true

思考

1 找到中间节点，然后使用栈对比就可以了
参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {boolean}
 */
```

```javascript
// Runtime: 100 ms, faster than 23.11% of JavaScript online submissions for Palindrome Linked List.
// Memory Usage: 42 MB, less than 65.81% of JavaScript online submissions for Palindrome Linked List.
export default (head) => {
  let count = 0;
  let copyHead = head;
  while (head) {
    ++count;
    head = head.next;
  }
  let half;
  if (count % 2 === 0) {
    half = count / 2 + 1;
  } else {
    half = Math.ceil(count / 2) + 1;
  }
  let curHead = copyHead;
  const stack = [];
  while (half > 1) {
    --half;
    stack.push(curHead.val);
    curHead = curHead.next;
  }
  if (count % 2 !== 0) {
```

```
    stack.pop();
  }
  while (curHead) {
    if (curHead.val !== stack.pop()) {
      return false;
    }
    curHead = curHead.next;
  }
  return true;
};
```

83. 移除链表中的重复元素

题目描述

移除链表中的重复元素。

例子1
Input: head = [1,1,2]
output: [1,2]

例子2

Input: head = [1,1,2,3,3]
output: [1,2,3]

思考

1 直接删除就可以了
参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
//  Runtime: 84 ms, faster than 94.39% of JavaScript online submissions for Remove Duplicates from Sorted List.
//  Memory Usage: 40.7 MB, less than 24.89% of JavaScript online submissions for Remove Duplicates from Sorted List.
export default (head) => {
```

```
  let copyHead = head;
  while (head) {
    const next = head.next;
    if (!next) {
      return copyHead;
    }
    if (head.val === next.val) {
      head.next = next.next;
    } else {
      head = next;
    }
  }
  return copyHead;
};
```

时间复杂度O（n），空间复杂度O（1）

328. 奇偶链表

题目描述

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的

值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

例子1
Input: 1->2->3->4->5->NULL
output: 1->3->5->2->4->NULL

例子2
Input: 2->1->3->5->6->4->7->NULL
output: 2->3->6->7->1->5->4->NULL

说明：
应当保持奇数节点和偶数节点的相对顺序。
链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推

思考

1 链接偶数和奇数就可以，不过细节很多，需要注意细节
参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
```

```
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
// Runtime: 96 ms, faster than 36.86% of JavaScript online submissions for Odd Even Linked List.
// Memory Usage: 41.3 MB, less than 19.30% of JavaScript online submissions for Odd Even Linked List.
export default (head) => {
  let copyOddHead = head;
  if (!head) {
    return head;
  }
  let copytEvenHead = head.next;
  let temp = copytEvenHead;

  while (copytEvenHead) {
    const next = copytEvenHead.next;
    if (!next) {
      head.next = temp;
      break;
```

```
    }
    head.next = next;
    head = next;

    copytEvenHead.next = head.next;
    copytEvenHead = head.next;

    if (!head.next) {
      head.next = temp;
      break;
    }
  }

  return copyOddHead;
};
```

19. 删除链表的倒数第 N 个结点

题目描述

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

进阶：你能尝试使用一趟扫描实现吗?

例子1
Input: head = [1,2,3,4,5], n = 2
output: [1,2,3,5]

例子2
Input: head = [1], n = 1
output: []

思考

1 使用快慢指针先找出倒数第n个的位置，这里有个技巧，通过增加一个头结点，你会发现逻辑清晰很多。
参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
```

```
 * @param {ListNode} head
 * @param {number} n
 * @return {ListNode}
 */
function ListNode(val, next) {
  this.val = val === undefined ? 0 : val;
  this.next = next === undefined ? null : next;
}
// Runtime: 104 ms, faster than 11.59% of JavaScript online submissions for Remove Nth Node From End of List.
// Memory Usage: 39.9 MB, less than 91.59% of JavaScript online submissions for Remove Nth Node From End of List.
export default (head, n) => {
  const tempHead = new ListNode();
  tempHead.next = head;
  head = tempHead;

  const copyHead = head;
  let p = head;
  while (n > 0) {
    --n;
    p = p.next;
  }
  while (p && p.next) {
    p = p.next;
```

```
    head = head.next;
  }

  const temp = head.next;
  head.next = temp.next;
  return copyHead;
};
```

148. 排序链表

题目描述

给你链表的头结点 head ， 请将其按 升序 排列并返回 排序后的链表 。

进阶：你可以在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序吗?

例子1
Input: head = [4,2,1,3]
output: [1,2,3,4]

例子2
Input: head = [-1,5,3,4,0]
output: [-1,0,3,4,5]

思考

1 直接使用归并排序就可以。
参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
function ListNode(val, next) {
  this.val = val === undefined ? 0 : val;
  this.next = next === undefined ? null : next;
}
const merge = (l1, l2) => {
```

```
  let head = new ListNode();
  let p = head;
  while (l1 && l2) {
    if (l1.val <= l2.val) {
      p.next = l1;
      l1 = l1.next;
    } else {
      p.next = l2;
      l2 = l2.next;
    }
    p = p.next;
  }
  if (l1) {
    p.next = l1;
  }
  if (l2) {
    p.next = l2;
  }
  return head.next;
};
// Runtime: 140 ms, faster than 86.37% of JavaScript online submissions for Sort List.
// Memory Usage: 54.6 MB, less than 37.62% of JavaScript online submissions for Sort List.
const sortList = (head) => {
```

```
  if (head === null || head.next === null) {
    return head;
  }
  let pre;
  let slow = head;
  let fast = head;

  while (fast && fast.next != null) {
    pre = slow;
    slow = slow.next;
    fast = fast.next.next;
  }
  pre.next = null;
  let l1 = sortList(head);
  let l2 = sortList(slow);
  return merge(l1, l2);
};

export default sortList;
```

第14章啊哈，树

树

树

104. 二叉树的最大深度

题目描述

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

例子1

Input: 给定二叉树 [3,9,20,null,null,15,7],

output: 3

思考

1 比较简单，直接深度递归就可以了

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
const bfs = (root, count) => {
```

```
  if (!root) {
    return count;
  }
  let leftCount = count;
  let rightCount = count;
  if (root.left) {
    leftCount = bfs(root.left, count + 1);
  }
  if (root.right) {
    rightCount = bfs(root.right, count + 1);
  }
  return Math.max(leftCount, rightCount);
};
// Runtime: 104 ms, faster than 9.88% of JavaScript online submissions for Maximum Depth of Binary Tree.
// Memory Usage: 41.6 MB, less than 44.71% of JavaScript online submissions for Maximum Depth of Binary Tree.
export default (root) => {
  if (!root) {
    return 0;
  }
  return bfs(root, 1);
};
```

110. 平衡二叉树

题目描述

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：
一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1 。

例子1

Input: root = [3,9,20,null,null,15,7]

output: true

思考

1 和108类似，也是求出子树的深度进行对比，不过这里如果发现子树不是平衡树，可以提前退出

参考实现1

2 简洁写法

参考实现2

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
const bfs = (root, count) => {
  if (!root) {
    return count;
  }
  let leftCount = count;
  let rightCount = count;
  if (root.left) {
    leftCount = bfs(root.left, count + 1);
  }
  if (root.right) {
```

```
    rightCount = bfs(root.right, count + 1);
  }
  return Math.max(leftCount, rightCount);
};
const balance = (root) => {
  if (!root) {
    return true;
  }
  let leftCount = 1;
  let rightCount = 1;
  if (root.left) {
    leftCount = bfs(root.left, 2);
  }
  if (root.right) {
    rightCount = bfs(root.right, 2);
  }
  return Math.abs(leftCount - rightCount) <= 1;
};
// Runtime: 148 ms, faster than 5.17% of JavaScript online submissions for Balanced Binary Tree.
// Memory Usage: 44.1 MB, less than 17.21% of JavaScript online submissions for Balanced Binary Tree.
export default (root) => {
  const stack = [root];
  while (stack.length > 0) {
```

```
    const root = stack.pop();
    if (!balance(root)) {
      return false;
    } else {
      if (root && root.left) {
        stack.push(root.left);
      }
      if (root && root.right) {
        stack.push(root.right);
      }
    }
  }

  return true;
};
```

实现2

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
```

```
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
const helper = (root) => {
  if (!root) {
    return 0;
  }
  let leftCount = 0;
  let rightCount = 0;
  if (root.left) {
    leftCount = helper(root.left);
  }
  if (root.right) {
    rightCount = helper(root.right);
  }
  if (leftCount === -1 || rightCount === -1 || Math.abs(leftCount - rightCount) > 1) {
    return -1;
```

```
  }
  return 1 + Math.max(leftCount, rightCount);
};

// Runtime: 148 ms, faster than 5.17% of JavaScript online submissions for Balanced Binary Tree.
// Memory Usage: 44.1 MB, less than 17.21% of JavaScript online submissions for Balanced Binary Tree.
export default (root) => {
  return helper(root) != -1;
};
```

543. 二叉树的直径

题目描述

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

例子1

Input:

```
      1
```

```
      /\
     2  3
    /\
   4  5
```

output: 返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

思考

1 树的解法大多数是使用递归

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
```

```
 * @param {TreeNode} root
 * @return {number}
 */
const getLength = (root) => {
  if (!root.left && !root.right) {
    return 1;
  }
  let left = 0;
  if (root && root.left) {
    left = 1 + getLength(root.left);
  }
  let right = 0;
  if (root && root.right) {
    right = 1 + getLength(root.right);
  }
  return Math.max(left, right, 1);
};

const diameterOfBinaryTree1 = (root) => {
  if (!root) {
    return 0;
  }
  let count = 1;
```

```
  if (root && root.left) {
    count += getLength(root.left);
  }
  if (root && root.right) {
    count += getLength(root.right);
  }
  return Math.max(count, diameterOfBinaryTree1(root.left), diameterOfBinaryTree1(root.right));
};
// Runtime: 248 ms, faster than 5.20% of JavaScript online submissions for Diameter of Binary Tree.
// Memory Usage: 41.9 MB, less than 59.36% of JavaScript online submissions for Diameter of Binary Tree.
export default (root) => {
  const res = diameterOfBinaryTree1(root);
  return res >= 1 ? res - 1 : 0;
};
```

437. 路径总和 III

题目描述

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

例子1

```
  root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     / \
    5   -3
   / \   \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有:

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11
```

思考

1 题目比较简单，但是要注意一种情况就是如果从某个节点到它的子节点如果等于sum，此时还要继续向下边寻找，因为下边有可能他们的和是0

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} sum
 * @return {number}
 */
const equalSum = (root, sum) => {
  let res = 0;
```

```
  if (root && root.val === sum) {
    res += 1;
  }

  if (root && root.left) {
    res += equalSum(root.left, sum - root.val);
  }
  if (root && root.right) {
    res += equalSum(root.right, sum - root.val);
  }
  return res;
};
var pathSum = function (root, sum) {
  const stack = [];
  if (root) {
    stack.push(root);
  }
  let res = 0;
  while (stack.length > 0) {
    const temp = stack.pop();

    res += equalSum(temp, sum);
```

```
    if (temp.right) {
      stack.push(temp.right);
    }
    if (temp.left) {
      stack.push(temp.left);
    }
  }
  return res;
};
export default pathSum;
```

101. 对称二叉树

题目描述

给定一个二叉树，检查它是否是镜像对称的。

例子1
下面的数是对称的

```
   1
  / \
```

```
  2   2
 / \ / \
3  4 4  3
```

例子2
下面的数是对称的

```
   1
  / \
 2   2
  \   \
   3   3
```

思考

1 判断一个数是否是对称，可以根据节点进行对比，如果是叶子节点肯定是对称的。

这里的关键是如何选择节点进行对比?

也就是如果在一棵树中找到相应的节点进行对比，相通这里之后，解法就很简单了

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
const isSymmetric1 = (left, right) => {
  if (!left && !right) {
    return true;
  }
  if (!left || !right) {
    return false;
```

```
  }
  if (left.val != right.val) {
    return false;
  }
  return isSymmetric1(left.left, right.right) && isSymmetric1(left.right, right.left);
};
// Runtime: 92 ms, faster than 58.10% of JavaScript online submissions for Symmetric Tree.
// Memory Usage: 40.5 MB, less than 66.28% of JavaScript online submissions for Symmetric Tree.
export default (root) => {
  return root ? isSymmetric1(root.left, root.right) : true;
};
```

1110. 删点成林

题目描述

给出二叉树的根节点 root，树上每个节点都有一个不同的值。

如果节点值在 to_delete 中出现，我们就把该节点从树上删去，最后得到一个森林（一些不相交的树构成的集合）。

返回森林中的每棵树。你可以按任意顺序组织答案。

例子1

思考

1 数的类型基本上都可以使用递归

1.1 这里的难点是如何确定当前遍历的节点是否可以加入到结果数组中?

当我们遍历到一个节点的时候，根据什么确定应该加入到结果数组中呢?

1.2 第二点需要注意的是我们在遍历树的时候，如果删除了，此时需要更新被删除节点的父元素的指向?

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
```

```
 * @param {number[]} to_delete
 * @return {TreeNode[]}
 */
const deleteVal = (root, is_root, to_delete, res) => {
  if (!root) return null;
  const deleted = to_delete.includes(root.val);
  if (is_root && !deleted) {
    res.push(root);
  }
  root.left = helper(root.left, deleted, to_delete, res);
  root.right = helper(root.right, deleted, to_delete, res);
  return deleted ? null : root;
};
// Runtime: 120 ms, faster than 32.97% of JavaScript online submissions for Delete Nodes And Return Forest.
// Memory Usage: 47.8 MB, less than 5.41% of JavaScript online submissions for Delete Nodes And Return Forest.
export default (root, to_delete) => {
  const res = [];
  helper(root, true, to_delete, res);
  return res;
};
```

105. 从前序与中序遍历序列构造二叉树

题目描述

根据一棵树的前序遍历与中序遍历构造二叉树。

你可以假设树中没有重复的元素。

例子1

思考

1 这个很好解决，一种耗时多

参考实现1
2 利用下标，防止重复复制元素

参考实现2

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
```

```
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {number[]} preorder
 * @param {number[]} inorder
 * @return {TreeNode}
 */
preorder = [3, 9, 20, 15, 7];
inorder = [9, 3, 15, 20, 7];
function TreeNode(val, left, right) {
  this.val = val === undefined ? 0 : val;
  this.left = left === undefined ? null : left;
  this.right = right === undefined ? null : right;
}
// Runtime: 4776 ms, faster than 5.07% of JavaScript online submissions for Construct Binary Tree from Preorder and Inorder Traversal.
// Memory Usage: 122.3 MB, less than 5.27% of JavaScript online submissions for Construct Binary Tree from Preorder and Inorder
Traversal.
const buildTree = (preorder, inorder) => {
  if (preorder.length > 0) {
    const rootVal = preorder.shift();
```

```
    const inIndex = inorder.indexOf(rootVal);
    const leftInorder = inorder.slice(0, inIndex);
    const rightInorder = inorder.slice(inIndex);
    const root = new TreeNode(rootVal);

    const leftPreorder = [];
    const rightPreorder = [];
    for (let i = 0; i < preorder.length; i++) {
      if (leftInorder.includes(preorder[i]) && leftPreorder.length < leftInorder.length) {
        leftPreorder.push(preorder[i]);
      } else {
        rightPreorder.push(preorder[i]);
      }
    }
    root.left = buildTree(leftPreorder, leftInorder);
    root.right = buildTree(rightPreorder, rightInorder);
    return root;
  }
  return null;
};
export default buildTree;
```

实现2

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {number[]} preorder
 * @param {number[]} inorder
 * @return {TreeNode}
 */
function TreeNode(val, left, right) {
  this.val = val === undefined ? 0 : val;
  this.left = left === undefined ? null : left;
  this.right = right === undefined ? null : right;
}
// Runtime: 96 ms, faster than 87.67% of JavaScript online submissions for Construct Binary Tree from Preorder and Inorder Traversal.
// Memory Usage: 41.7 MB, less than 93.64% of JavaScript online submissions for Construct Binary Tree from Preorder and Inorder
```

```
Traversal.
const helper = (preStart, inStart, inEnd, preorder, inorder) => {
  if (preStart > preorder.length - 1 || inStart > inEnd) {
    return null;
  }
  const rootVal = preorder[preStart];
  const root = new TreeNode(rootVal);
  const index = inorder.indexOf(rootVal);
  // index - inStart + 1;
  root.left = helper(preStart + 1, inStart, index - 1, preorder, inorder);
  root.right = helper(preStart + index - inStart + 1, index + 1, inEnd, preorder, inorder);
  return root;
};
const buildTree = (preorder, inorder) => {
  return helper(0, 0, inorder.length - 1, preorder, inorder);
};
export default buildTree;
```

144. 二叉树的前序遍历

题目描述

根据一棵树的前序遍历与中序遍历构造二叉树。

例子1
输入：root = [1,null,2,3]
输出：[1,2,3]

例子2
输入：root = []
输出：[]

思考

1 这个很好解决，一种是使用递归

参考实现1
2 如果不使用递归，可以使用栈正序遍历，也就是先遍历左子树，一直到底，然后再右子树。

2.1 这里需要考虑下什么时候退栈，什么时候入栈，什么时候该退出?

参考实现2

3 还有一种也是使用栈，但是本质上和递归有点类似，利用栈的先入后出的特性

参考实现3

实现1

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
// Runtime: 80 ms, faster than 57.41% of JavaScript online submissions for Binary Tree Preorder Traversal.
// Memory Usage: 38.9 MB, less than 11.26% of JavaScript online submissions for Binary Tree Preorder Traversal.
const travel = (root, res) => {
  if (!root) {
    return;
  }

  res.push(root.val);
  travel(root.left, res);
  travel(root.right, res);
```

```
};
export default (root) => {
  const res = [];
  travel(root, res);
  return res;
};
```

实现2

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
```

```javascript
// Runtime: 80 ms, faster than 57.41% of JavaScript online submissions for Binary Tree Preorder Traversal.
// Memory Usage: 38.8 MB, less than 41.85% of JavaScript online submissions for Binary Tree Preorder Traversal.
export default (root) => {
  let res = [];
  let stack = [];
  if (root) {
    stack.push({
      node: root,
      hasVisitedLeft: false,
      hasVisitedRight: false,
    });
  } else {
    return res;
  }

  while (stack.length > 0) {
    const temp = stack[stack.length - 1];
    if (!temp.hasVisitedLeft) {
      res.push(temp.node.val);
    }

    if (temp.node.left && !temp.hasVisitedLeft) {
      temp.hasVisitedLeft = true;
```

```
      stack.push({
        node: temp.node.left,
        hasVisitedLeft: false,
        hasVisitedRight: false,
      });
    } else if (temp.node.right && temp.hasVisitedLeft && !temp.hasVisitedRight) {
      temp.hasVisitedRight = true;
      stack.push({
        node: temp.node.right,
        hasVisitedLeft: false,
        hasVisitedRight: false,
      });
    } else if (!temp.node.left && !temp.hasVisitedLeft) {
      temp.hasVisitedLeft = true;
    } else if (!temp.node.right && !temp.hasVisitedRight) {
      temp.hasVisitedRight = true;
    } else {
      stack.pop();
    }
  }
  return res;
};
```

实现3

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
// Runtime: 80 ms, faster than 57.41% of JavaScript online submissions for Binary Tree Preorder Traversal.
// Memory Usage: 38.8 MB, less than 41.85% of JavaScript online submissions for Binary Tree Preorder Traversal.
export default (root) => {
  let res = [];
  let stack = [];
  if (root) {
    stack.push(root);
  } else {
```

```
    return res;
  }

  while (stack.length > 0) {
    const temp = stack.pop();
    res.push(temp.val);
    if (temp.right) {
      stack.push(temp.right);
    }
    if (temp.left) {
      stack.push(temp.left);
    }
  }
  return res;
};
```

实现一颗二叉查找树

题目描述

实现一颗二叉查找树

思考

1 使用递归解决

参考实现1

实现1

```
class Node {
  constructor(val, left, right) {
    this.val = val === undefined ? 0 : val;
    this.left = left === undefined ? null : left;
    this.right = right === undefined ? null : right;
  }
}
class BinarySearchTree {
  constructor(val) {
    this.root = new Node(val);
  }
  insert(val) {
    this.insertNode(this.root, new CreateNode(val));
  }
  insertNode(node, newNode) {
    // 如果节点大于新节点的值
```

```
    if (node.val > newNode.val) {
      if (node.left) {
        this.insertNode(node.left, newNode);
      } else {
        node.left = newNode;
      }
    } else if (node.val < newNode.val) {
      if (node.right) {
        this.insertNode(node.right, newNode);
      } else {
        node.right = newNode;
      }
    }
  }
  // 中序遍历
  inOrderTraverce() {
    const res = [];
    this.inOrderTraverceNodes(this.root, res);
    return res;
  }
  inOrderTraverceNodes(node, res) {
    if (node) {
      this.inOrderTraverceNodes(node.left, res);
```

```
    res.push(node.val);
    this.inOrderTraverceNodes(node.right, res);
  }
}
// 前序排列
preOrderTraverce() {
  const res = [];
  this.preOrderTraverceNodes(this.root, res);
  return res;
}
preOrderTraverceNodes(node, res) {
  if (node) {
    res.push(node.val);
    this.preOrderTraverceNodes(node.left, res);
    this.preOrderTraverceNodes(node.right, res);
  }
}

//后序遍历
afterOrderTraverce() {
  const res = [];
  this.afterOrderTraverceNodes(this.root, res);
  return res;
```

```
  }
  afterOrderTraverceNodes(node, res) {
    if (node) {
      this.afterOrderTraverceNodes(node.left, res);
      this.afterOrderTraverceNodes(node.right, res);
      res.push(node.val);
    }
  }

  //层次遍历
  levelOrderTraverce() {
    const res = [];
    const quene = [this.root];
    let node = null;
    while (quene.length > 0) {
      node = quene.shift();
      res.push(node.val);
      if (node && node.left) {
        quene.push(node.left);
      }
      if (node && node.right) {
        quene.push(node.right);
      }
```

```
    }
    return res;
  }

  //查询最小值：找到最左边的节点
  findMin() {
    return this.min(this.root);
  }
  min(node) {
    if (node) {
      while (node.left) {
        node = node.left;
      }
      return node.val;
    } else {
      return null;
    }
  }

  //查询最大值：找到最左边的节点
  findMax() {
    return this.max(this.root);
  }
```

```
  max(node) {
    if (node) {
      while (node.right) {
        node = node.right;
      }
      return node.val;
    } else {
      return null;
    }
  }

  //查找特定值
  find(val) {
    return this.findVal(this.root, val);
  }
  findVal(node, val) {
    if (!node) {
      return false;
    }
    if (node.val < val) {
      return this.findVal(node.right, val);
    } else if (node.val > val) {
      return this.findVal(node.left, val);
```

```
    } else {
      return true;
    }
  }

  remove(val) {
    this.removeNode(this.root, val);
  }
  // 找到最小值的节点
  findMinNode(node) {
    if (node) {
      while (node.left) {
        node = node.left;
      }
      return node;
    } else {
      return null;
    }
  }
  // 删除节点的方法
  removeNode(node, val) {
    if (!node) {
      return null;
```

```
    }
    if (node.val > val) {
      node.left = this.removeNode(node.left, val);
      return node;
    } else if (node.val < val) {
      node.right = this.removeNode(node.right, val);
      return node;
    } else {
      // 找到需要删除的节点后
      // 第一种，如果是叶子节点，那么可以直接删除
      if (!node.left && !node.right) {
        node = null;
        return null;
        // 如果有左节点，没有右节点，直接把node删除，然后
      } else if (node.left && !node.right) {
        node = node.left;
        return node;
        // 如果有右节点，没有左节点，
      } else if (!node.left && node.right) {
        //删除只有右节点的节点 => 将右节点替换需删除节点
        node = node.right;
        return node;
        // 同时存在左右节点
```

```
      } else {
        // 在右子树中找到最小值节点
        const minNode = this.findMinNode(node.right);
        // 最小值节点的值 赋给 被删除点的值，即完成替换
        node.val = minNode.val;
        // 替换后删除最小值节点
        node.right = this.removeNode(node.right, minNode.val);
        return node;
      }
    }
  }
}
```

99. 恢复二叉搜索树

题目描述

给你二叉搜索树的根节点 root ，该树中的两个节点被错误地交换。请在不改变其结构的情况下，恢复这棵树。

进阶：使用 O(n) 空间复杂度的解法很容易实现。你能想出一个只使用常数空间的解决方案吗？

例子1

输入：root = [1,3,null,null,2]
输出:[3,1,null,null,2]
解释：3 不能是 1 左孩子，因为 3 > 1 。交换 1 和 3 使二叉搜索树有效。

例子2
输入：root = [3,1,4,null,null,2]
输出:[2,1,4,null,null,3]
解释：2 不能在 3 的右子树中，因为 2 < 3 。交换 2 和 3 使二叉搜索树有效。

思考

1 中序遍历二叉查找树的时候，可以按照从小到打排列，但是如果想要使用O（1）的空间复杂度的时候来解决此问题，直接看答案就可以

如果想要使用O（1），可以查找下莫里斯查找算法，该算法实现了使用常量空间

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
```

```
 * }
 */
/**
 * @param {TreeNode} root
 * @return {void} Do not return anything, modify root in-place instead.
 */
// Runtime: 396 ms, faster than 5.73% of JavaScript online submissions for Recover Binary Search Tree.
// Memory Usage: 53.3 MB, less than 6.37% of JavaScript online submissions for Recover Binary Search Tree.
export default (root) => {
  if (!root) {
    return;
  }
  // 错误的节点
  const errorNodes = [];
  let current = root;
  let preNode;
  // 当前遍历到节点的前一个节点
  let preCurrentNode;
  let first;
  let second;

  while (current) {
    if (!current.left) {
```

```
      if (preCurrentNode && preCurrentNode.val > current.val) {
        if (!first) {
          first = preCurrentNode;
        }
        second = current;
      }
      preCurrentNode = current;
      current = current.right;
    } else {
      preNode = current.left;
      while (preNode.right && preNode.right.val !== current.val) {
        preNode = preNode.right;
      }
      if (!preNode.right) {
        preNode.right = current;
        current = current.left;
      } else {
        if (preCurrentNode && preCurrentNode.val > current.val) {
          if (!first) {
            first = preCurrentNode;
          }
          second = current;
        }
```

```
        preNode.right = null;
        preCurrentNode = current;
        current = current.right;
      }
    }
  }

  // 交换
  const temp = first.val;
  first.val = second.val;
  second.val = temp;
};
```

669. 修剪二叉搜索树

题目描述

给你二叉搜索树的根节点 root ，同时给定最小边界low 和最大边界 high。通过修剪二叉搜索树，使得所有节点的值在[low, high]中。修剪树不应该改变保留在树中的元素的相对结构（即，如果没有被移除，原有的父代子代关系都应当保留）。 可以证明，存在唯一的答案。

所以结果应当返回修剪好的二叉搜索树的新的根节点。注意，根节点可能会根据给定的边界发生改变。

例子1
输入：root = [1,0,2], low = 1, high = 2
输出:[1,null,2]
解释：

例子2
输入：root = [3,0,4,null,2,null,null,1], low = 1, high = 3
输出:[3,2,null,1]
解释：

思考

1 很明显使用递归解决就可以了，比较简单

参考实现1

2 递归的时候，返回节点

参考实现2

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
```

```
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
function TreeNode(val, left, right) {
  this.val = val === undefined ? 0 : val;
  this.left = left === undefined ? null : left;
  this.right = right === undefined ? null : right;
}
/**
 * @param {TreeNode} root
 * @param {number} low
 * @param {number} high
 * @return {TreeNode}
 */
const inorder = (parent, root, low, high, dir) => {
  if (root && root.val < low) {
    if (dir === "left") {
      parent.left = root.right;
      root.right = null;
      inorder(parent, parent.left, low, high, "left");
    } else {
      parent.right = root.right;
```

```
      root.right = null;
      inorder(parent, parent.right, low, high, "right");
    }
  } else if (root && root.val > high) {
    if (dir === "left") {
      parent.left = root.left;
      root.right = null;
      inorder(parent, parent.left, low, high, "left");
    } else {
      parent.right = root.left;
      root.right = null;
      inorder(parent, parent.right, low, high, "right");
    }
  } else {
    if (root && root.left) {
      inorder(root, root.left, low, high, "left");
    }
    if (root && root.right) {
      inorder(root, root.right, low, high, "right");
    }
  }
};
```

```
// Runtime: 116 ms, faster than 34.98% of JavaScript online submissions for Trim a Binary Search Tree.
// Memory Usage: 44 MB, less than 91.36% of JavaScript online submissions for Trim a Binary Search Tree.
export default (root, low, high) => {
  const parent = new TreeNode();
  parent.left = root;
  inorder(parent, root, low, high, "left");
  return parent.left;
};
```

实现2

```
// Runtime: 96 ms, faster than 69.97% of JavaScript online submissions for Trim a Binary Search Tree.
// Memory Usage: 44.5 MB, less than 44.36% of JavaScript online submissions for Trim a Binary Search Tree.
const trimBST = (root, low, high) => {
  if (!root) return null;

  if (root.val < low) return trimBST(root.right, low, high);
  if (root.val > high) return trimBST(root.left, low, high);
  root.left = trimBST(root.left, low, high);
  root.right = trimBST(root.right, low, high);
```

```
  return root;
};
export default trimBST;
```

208. 实现 Trie (前缀树)

题目描述

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

例子1

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

说明:

你可以假设所有的输入都是由小写字母 a-z 构成的。
保证所有输入均为非空字符串。

思考

1 就是一颗多叉树，很好实现

参考实现1

实现1

```
var TrirNode = function (ch) {
  this.ch = ch || -1;
  this.children = new Map();
  this.isEnd = false;
};

/**
 * Initialize your data structure here.
 */

// Runtime: 320 ms, faster than 13.70% of JavaScript online submissions for Implement Trie (Prefix Tree).
// Memory Usage: 69.2 MB, less than 5.20% of JavaScript online submissions for Implement Trie (Prefix Tree).
```

```
var Trie = function () {
  this.root = new TrirNode();
};

/**
 * Inserts a word into the trie.
 * @param {string} word
 * @return {void}
 */
Trie.prototype.insert = function (word) {
  let currentRoot = this.root;
  for (let ch of word) {
    let node = currentRoot.children.get(ch);

    if (!node) {
      node = new TrirNode(ch);
      currentRoot.children.set(ch, node);
    }
    currentRoot = node;
  }
  currentRoot.isEnd = true;
};
```

```
/**
 * Returns if the word is in the trie.
 * @param {string} word
 * @return {boolean}
 */
Trie.prototype.search = function (word) {
  let currentRoot = this.root;
  for (let ch of word) {
    if (currentRoot.children.get(ch)) {
      currentRoot = currentRoot.children.get(ch);
    } else {
      return false;
    }
  }
  return currentRoot.isEnd === true;
};

/**
 * Returns if there is any word in the trie that starts with the given prefix.
 * @param {string} prefix
 * @return {boolean}
 */
Trie.prototype.startsWith = function (prefix) {
```

```
  let currentRoot = this.root;
  for (let ch of prefix) {
    if (currentRoot.children.get(ch)) {
      currentRoot = currentRoot.children.get(ch);
    } else {
      return false;
    }
  }
  return true;
};

/**
 * Your Trie object will be instantiated and called as such:
 * var obj = new Trie()
 * obj.insert(word)
 * var param_2 = obj.search(word)
 * var param_3 = obj.startsWith(prefix)
 */
```

226. 翻转二叉树

题目描述

翻转一棵二叉树。

例子1

```
   4
  / \
 2   7
/\  /\
1 3 6  9

反转后

   4
  / \
 7   2
/\  /\
9 6 3  1
```

思考

1 刚开始想只是交换两个节点的值，但是发现在有一侧为null的情况下有问题，比如[1,2]的时候，所以直接交换节点

参考实现1

实现1

```
/**
 * Definition for a binary tree node.

 */
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */

const invert1 = (root, leftRoot, rightRoot) => {
  if (!root) {
    return;
  }
  root.right = leftRoot;
  root.left = rightRoot;
  if (rightRoot) {
    invert1(rightRoot, rightRoot.left, rightRoot.right);
```

```
  }
  if (leftRoot) {
    invert1(leftRoot, leftRoot.left, leftRoot.right);
  }
};
// Runtime: 80 ms, faster than 62.85% of JavaScript online submissions for Invert Binary Tree.
// Memory Usage: 39.2 MB, less than 8.63% of JavaScript online submissions for Invert Binary Tree.
export default (root) => {
  if (!root) return null;
  invert1(root, root.left, root.right);
  return root;
};
```

617. 合并二叉树

题目描述

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

例子1

思考

1 递归解决就可以了

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} t1
 * @param {TreeNode} t2
 * @return {TreeNode}
 */

// Runtime: 120 ms, faster than 48.51% of JavaScript online submissions for Merge Two Binary Trees.
// Memory Usage: 46.5 MB, less than 22.07% of JavaScript online submissions for Merge Two Binary Trees.
const merge = (left, right) => {
  if (!left) {
    return right;
  } else if (!right) {
```

```
    return left;
  } else {
    left.val += right.val;
    left.left = merge(left.left, right.left);
    left.right = merge(left.right, right.right);
    return left;
  }
};
export default (t1, t2) => {
  if (!t1) return t2;
  if (!t2) return t1;
  return merge(t1, t2);
};
```

572. 另一个树的子树

题目描述

给定两个非空二叉树 s 和 t，检验 s 中是否包含和 t 具有相同结构和节点值的子树。s 的一个子树包括 s 的一个节点和这个节点的所有子孙。s 也可以看做它自身的一棵子树。

例子1

```
  Tree s
    3
   / \
   4   5
  / \
 1   2

 Tree t

    4
  / \
 1   2



返回true
```

例子2

```
  Tree s
    3
```

```
   / \
  4   5
 / \
1   2
  /
 0
Tree t

   4
 / \
1   2



返回false
```

思考

1 递归解决就可以了

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} s
 * @param {TreeNode} t
 * @return {boolean}
 */
const isEqual = (s, t) => {
  if (!s && !t) {
    return true;
  }
  if (s && !t) {
    return false;
  }
  if (!s && t) {
    return false;
```

```
  }
  if (s.val != t.val) return false;
  return isEqual(s.left, t.left) && isEqual(s.right, t.right);
};
// Runtime: 108 ms, faster than 60.11% of JavaScript online submissions for Subtree of Another Tree.
// Memory Usage: 46 MB, less than 5.88% of JavaScript online submissions for Subtree of Another Tree.
const isSubtree = (s, t) => {
  if (!s) return false;
  if (isEqual(s, t)) return true;

  return isSubtree(s.left, t) || isSubtree(s.right, t);
};

export default
```

404. 左叶子之和

题目描述

计算给定二叉树的所有左叶子之和。

例子1

```
    3
   / \
  9  20
    /  \
   15   7

在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24
```

思考

1 这里的难点是如何确定是左叶子节点

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
```

```
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
const getLeftLeaves = (root, dir) => {
  if (!root) {
    return 0;
  }
  let sum = 0;
  if (root && !root.left && !root.right && dir === "left") {
    return root.val;
  } else {
    if (root && root.left) {
      sum += getLeftLeaves(root.left, "left");
    }
    if (root && root.right) {
      sum += getLeftLeaves(root.right, "right");
    }
  }
  return sum;
};
```

```
// Runtime: 76 ms, faster than 97.22% of JavaScript online submissions for Sum of Left Leaves.
// Memory Usage: 39.9 MB, less than 88.86% of JavaScript online submissions for Sum of Left Leaves.
const sumOfLeftLeaves = (root) => {
  if (!root) return 0;
  if (root && !root.left && !root.right) return 0;
  return getLeftLeaves(root, "left");
};
export default sumOfLeftLeaves;
```

513. 找树左下角的值

题目描述

给定一个二叉树，在树的最后一行找到最左边的值。

例子1

```
  输入:

   2
  / \
 1   3
```

```
输出:
1
```

例子2

```
  输入:

      1
     /\
    2  3
   /  /\
  4  5  6
    /
   7

输出:
7
```

提示：

注意: 您可以假设树（即给定的根节点）不为 NULL。

思考

1 树的解法无非就是三种递归，深度遍历，广度遍历，中序遍历等的变种

这里使用广度遍历

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
```

```
 */
export default (root) => {
  const levels = [];
  let stack = [root];
  let level = 0;
  while (stack.length) {
    levels.push([]);
    const newStack = [];
    while (stack.length) {
      const curr = stack.shift();
      levels[level].push(curr);
      if (curr.left) newStack.push(curr.left);
      if (curr.right) newStack.push(curr.right);
    }

    stack = newStack;
    level++;
  }
  return levels[levels.length - 1][0].val;
};
```

538. 把二叉搜索树转换为累加树

题目描述

给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于或等于 node.val 的值之和。

提醒一下，二叉搜索树满足下列约束条件：

节点的左子树仅包含键 小于 节点键的节点。
节点的右子树仅包含键 大于 节点键的节点。
左右子树也必须是二叉搜索树。

例子1

```
输入：[4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出：[30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

例子2

```
输入：root = [0,null,1]
输出：[1,null,1]
```

提示：

树中的节点数介于 0 和 104 之间。
每个节点的值介于 -104 和 104 之间。
树中的所有值 互不相同 。
给定的树为二叉搜索树。

思考

1 树的解法无非就是三种递归，深度遍历，广度遍历，中序遍历等的变种

这里使用先遍历右边，再中间，再左边的方法

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
```

```
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
const convert = (cur, sum) => {
  if (!cur) return sum;
  sum = convert(cur.right, sum);
  cur.val += sum;
  sum = cur.val;
  sum = convert(cur.left, sum);
  return sum;
};
// Runtime: 104 ms, faster than 95.00% of JavaScript online submissions for Convert BST to Greater Tree.
// Memory Usage: 47.4 MB, less than 85.00% of JavaScript online submissions for Convert BST to Greater Tree.
export default (root) => {
  convert(root, 0);
  return root;
};
```

235. 二叉搜索树的最近公共祖先

题目描述

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例子1

```
 输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6
```

例子2

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
```

```
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

提示：

所有节点的值都是唯一的。

p、q 为不同节点且均存在于给定的二叉搜索树中。

思考

1 直接使用二叉搜索的性质解决就可以

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
```

```javascript
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
// Runtime: 96 ms, faster than 88.11% of JavaScript online submissions for Lowest Common Ancestor of a Binary Search Tree.
// Memory Usage: 48.6 MB, less than 70.92% of JavaScript online submissions for Lowest Common Ancestor of a Binary Search Tree.
const lowestCommonAncestor = (root, p, q) => {
  if (!root) return null;
  if (p > q) {
    const temp = p;
    p = q;
    q = temp;
  }
  if (root.val === p.val || root.val === q.val) {
    return root;
  }

  if (root.val < p.val) {
    if (root && root.left) {
      return lowestCommonAncestor(root.left, p, q);
```

```
    } else {
      return null;
    }
  } else if (root.val > q.val) {
    if (root && root.right) {
      return lowestCommonAncestor(root.right, p, q);
    } else {
      return;
    }
  } else {
    return root;
  }
};
export default lowestCommonAncestor;
```

530. 二叉搜索树的最小绝对差

题目描述

给你一棵所有节点为非负值的二叉搜索树，请你计算树中任意两节点的差的绝对值的最小值。

例子1

```
输入：

   1
    \
     3
    /
   2

输出：
1

解释：
最小绝对差为 1，其中 2 和 1 的差的绝对值为 1（或者 2 和 3）。
```

提示：

树中至少有 2 个节点。

思考

1 直接使用二叉搜索的性质解决就可以

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
const inorder = (root, res) => {
  if (!root) return res;
  inorder(root.left, res);
  res.push(root.val);
  inorder(root.right, res);
};
// Runtime: 100 ms, faster than 51.72% of JavaScript online submissions for Minimum Absolute Difference in BST.
// Memory Usage: 44.2 MB, less than 77.59% of JavaScript online submissions for Minimum Absolute Difference in BST.
```

```
export default (root) => {
  const res = [];
  inorder(root, res);
  let min = -1;
  for (let i = 1; i < res.length; i++) {
    if (min < 0) {
      min = res[i] - res[i - 1];
    } else {
      min = Math.min(res[i] - res[i - 1], min);
    }
  }
  return min;
};
```

889. 根据前序和后序遍历构造二叉树

题目描述

返回与给定的前序和后序遍历匹配的任何二叉树。

pre 和 post 遍历中的值是不同的正整数。

例子1

```
输入：pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
输出：[1,2,3,4,5,6,7]
```

提示：

1 <= pre.length == post.length <= 30
pre[] 和 post[] 都是 1, 2, ..., pre.length 的排列
每个输入保证至少有一个答案。如果有多个答案，可以返回其中一个。

思考

1 直接按照前序和后序性质就可以

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
```

```
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {number[]} pre
 * @param {number[]} post
 * @return {TreeNode}
 */
// Runtime: 100 ms, faster than 58.06% of JavaScript online submissions for Construct Binary Tree from Preorder and Postorder Traversal.
// Memory Usage: 42.8 MB, less than 8.60% of JavaScript online submissions for Construct Binary Tree from Preorder and Postorder
Traversal.
const constructFromPrePost = (pre, post) => {
  if (pre.length > 0) {
    const root = new TreeNode(pre.shift());
    post.pop();
    const rootVal = post[post.length - 1];
    const index = pre.indexOf(rootVal);
    const pre1 = pre.slice(0, index);
    const pre2 = pre.slice(index);
    const index1 = post.indexOf(pre1[0]);
    const post1 = post.slice(0, index);
    const post2 = post.slice(index);
```

```
    console.log(pre1, post1);
    console.log(pre2, post2);
    root.left = constructFromPrePost(pre1, post1);
    root.right = constructFromPrePost(pre2, post2);
    return root;
  } else {
    return null;
  }
};
export default constructFromPrePost;
```

94. 二叉树的中序遍历

题目描述

给定一个二叉树的根节点 root ， 返回它的 中序 遍历。

例子1

```
输入：root = [1,null,2,3]
输出：[1,3,2]
```

例子2

```
输入：root = []
输出：[]
```

提示：

1 树中节点数目在范围 [0, 100] 内
2 -100 <= Node.val <= 100

思考

1 这里的主要难点思考下如何什么时候需要入栈，什么时候出栈

参考实现1

相似题目145

实现1

```
if (!root) return [];
  const stack = [root];
  const res = [];
```

```
let visited = false;
while (stack.length>0) {
  let pre = stack[stack.length - 1];
  while (pre && !visited) {
    pre = pre.left;
    if (pre) {
      stack.push(pre);
    }
  }
  const node = stack.pop();
  res.push(node.val);
  if (node.right) {
    stack.push(node.right);
    visited = false;
  } else {
    visited = true;
  }
}
return res;
```

236. 二叉树的最近公共祖先

题目描述

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例子1

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3 。
```

例子2

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。
```

提示：

1 树中节点数目在范围 [2, 105] 内。
2 -10^9 <= Node.val <= 10^9
3 所有 Node.val 互不相同 。
4 p != q
5 p 和 q 均存在于给定的二叉树中。

思考

1 树一般都是通过递归，这里可以思考一下，如果p或者q存在于左子树，如果p和q存在于右子树，怎么处理?

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
```

```
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
// Runtime: 92 ms, faster than 91.34% of JavaScript online submissions for Lowest Common Ancestor of a Binary Tree.
// Memory Usage: 48.2 MB, less than 27.33% of JavaScript online submissions for Lowest Common Ancestor of a Binary Tree.
export default (root, p, q) => {
  if (!root || root === p || root === q) return root;
  const left = lowestCommonAncestor(root.left, p, q);
  const right = lowestCommonAncestor(root.right, p, q);
  // 如果p或者q存在于left
  if (left) {
    // 如果p或者q存在于right
    if (right) {
      // 此时最近的肯定是root
      return root;
    } else {
      // 如果不存在于右边，那说明p或者q都存在于left
      return left;
    }
  } else {
    // 如果right存在，说明p和q都存在于right
    if (right) {
```

```
      return right;
    } else {
      // 如果不存在于左子树也不存在右子树
      return null;
    }
  }
};
```

109. 有序链表转换二叉搜索树

题目描述

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

例子1

```
给定的有序链表： [-10, -3, 0, 5, 9],

一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树：
```

```
    0
   / \
  -3   9
  /   /
 -10  5
```

思考

1 很简单，就是不断的利用快速指针找出中间的节点就可以了

参考实现1

实现1

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
```

```
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {ListNode} head
 * @return {TreeNode}
 */
function TreeNode(val, left, right) {
  this.val = val === undefined ? 0 : val;
  this.left = left === undefined ? null : left;
  this.right = right === undefined ? null : right;
}
const toBST = (head, end) => {
  let low = head;
  let high = head;
  if (head === end) return null;
  while (high !== end && high.next !== end) {
    low = low.next;
    high = high.next.next;
```

```
  }
  let root = new TreeNode(low.val);
  root.left = toBST(head, low);
  root.right = toBST(low.next, end);
  return root;
};
export default (head) => {
  if (!head) return head;
  return toBST(head, null);
};
```

897. 递增顺序查找树

题目描述

给你一个树，请你 按中序遍历 重新排列树，使树中最左边的结点现在是树的根，并且每个结点没有左子结点，只有一个右子结点。

例子1

```
输入：[5,3,6,2,4,null,8,1,null,null,null,7,9]

       5
```

输出：[1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

```
        8
         \
          9
```

思考

1 中序遍历就可以

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
```

```
 * @param {TreeNode} root
 * @return {TreeNode}
 */
//  Runtime: 84 ms, faster than 38.73% of JavaScript online submissions for Increasing Order Search Tree.
//  Memory Usage: 39 MB, less than 69.47% of JavaScript online submissions for Increasing Order Search Tree.
export default (root) => {
  if (!root) return root;
  let cur = root;
  const stack = [];
  let rootNode;
  let curNode;
  while (cur || stack.length) {
    while (cur) {
      stack.push(cur);
      cur = cur.left;
    }
    const node = stack.pop();
    if (!rootNode) {
      rootNode = new TreeNode(node.va);
      curNode = rootNode;
    } else {
      curNode.left = null;
      curNode.right = new TreeNode(node.va);
```

```
      curNode = curNode.right;
    }
    cur = node.right;
  }
  return rootNode;
};
```

50. 删除二叉搜索树中的节点

题目描述

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

首先找到需要删除的节点；
如果找到了，删除它。

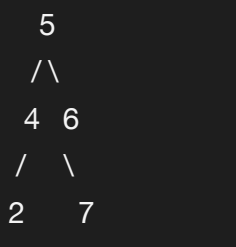
说明： 要求算法时间复杂度为 O(h)，h 为树的高度。

例子1

root = [5,3,6,2,4,null,7]
key = 3

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。



另一个正确答案是 [5,2,6,null,4,null,7]。

```
  /\
 2  6
  \  \
   4  7
```

思考

1 递归就可以，不过需要需要叶子节点和有一颗左子树和有一颗右子树的时候或者同时有两颗子树的情况

参考实现1

实现1

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
```

```
/**
 * @param {TreeNode} root
 * @param {number} key
 * @return {TreeNode}
 */
const findMin = (root) => {
  while (root.left) {
    root = root.left;
  }
  return root;
};
// Runtime: 112 ms, faster than 75.00% of JavaScript online submissions for Delete Node in a BST.
// Memory Usage: 47.2 MB, less than 65.49% of JavaScript online submissions for Delete Node in a BST.
const deleteNode = (root, key) => {
  if (!root) return root;
  if (key < root.val) {
    root.left = deleteNode(root.left, key);
  } else if (key > root.val) {
    root.right = deleteNode(root.right, key);
  } else {
    // 只有一个右节点
    if (!root.left) {
      return root.right;
```

```
      // 只有一个左节点
    } else if (!root.right) {
      return root.left;
    }
    // 同时有两个节点
    const minNode = findMin(root.right);
    root.val = minNode.val;
    root.right = deleteNode(root.right, root.val);
  }
  return root;
};

export default deleteNode;
```

第15章有图才有真相

图

图一般分为有向图和无向图两种

785. 判断二分图

题目描述

给定一个无向图graph，当这个图为二分图时返回true。

如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。

graph将会以邻接表方式给出，graph[i]表示图中与节点i相连的所有节点。每个节点都是一个在0到graph.length-1之间的整数。这图中没有自环和平行边： graph[i] 中不存在i，并且graph[i]中没有重复的值。

例子1

示例 1:
输入: [[1,3], [0,2], [1,3], [0,2]]
输出: true
解释:
无向图如下:
0----1
| |
| |
3----2
我们可以将节点分成两组: {0, 2} 和 {1, 3}。

例子2

示例 2:
输入: [[1,2,3], [0,2], [0,1,3], [0,2]]
输出: false
解释:
无向图如下:
0----1
| \ |
| \|

```
3----2
我们不能将节点分割成两个独立的子集。
```

注意：
1 graph 的长度范围为 [1, 100]。
2 graph[i] 中的元素的范围为 [0, graph.length - 1]。
3 graph[i] 不会包含 i 或者有重复的值。
4 图是无向的: 如果j 在 graph[i]里边, 那么 i 也会在 graph[j]里边。

思考

1 图一般是使用深度和广度来解决。

这里的关键是考虑如何把问题转换一下，否则感觉还是特别麻烦?

看了题解，这里转成了类似染色，把整个图染成两种颜色，那么就是可以转成两个部分，如果不能分别染成两个颜色，那么就是不能分成两个部分。

深度参考实现1

广度参考实现2

实现1

```
/**
 * @param {number[][]} graph
 * @return {boolean}
 */

// 把node染成红色或者黄色
const isValidColor = (graph, colors, color, node) => {
  // 如果已经被染色过了
  if (colors[node] !== 0) {
    // 看下是不是和需要染的颜色一样
    return colors[node] === color;
  }
  // 如果没有被染色过，则把节点染成颜色
  colors[node] = color;
  // 然后把相邻的节点都染成颜色
  for (let next of graph[node]) {
    if (!isValidColor(graph, colors, -color, next)) {
      return false;
    }
  }
  return true;
};
```

```
// Runtime: 88 ms, faster than 70.08% of JavaScript online submissions for Is Graph Bipartite?.
// Memory Usage: 41.1 MB, less than 79.22% of JavaScript online submissions for Is Graph Bipartite.
export default (graph) => {
  const len = graph.length;

  // 0没有被染色过，1 染成红色， -1 染成黄色
  const colors = new Array(len).fill(0);

  for (let i = 0; i < len; i++) {
    if (colors[i] === 0 && !isValidColor(graph, colors, 1, i)) {
      return false;
    }
  }

  return true;
};
```

实现2

```
/**
 * @param {number[][]} graph
```

```
 * @return {boolean}
 */

// Runtime: 84 ms, faster than 85.87% of JavaScript online submissions for Is Graph Bipartite?.
// Memory Usage: 41.7 MB, less than 39.34% of JavaScript online submissions for Is Graph Bipartite?.
export default (graph) => {
  const len = graph.length;
  // 0没有被染色过，1 染成红色， -1 染成黄色
  const colors = new Array(len).fill(0);

  for (let i = 0; i < len; i++) {
    if (colors[i] !== 0) continue;
    const queue = [];
    queue.push(i);
    colors[i] = 1;

    while (queue.length) {
      const cur = queue.shift();
      for (let next of graph[cur]) {
        if (colors[next] === 0) {
          colors[next] = -colors[cur];
          queue.push(next);
        } else if (colors[next] != -colors[cur]) {
```

```
          return false;
        }
      }
    }
  }

  return true;
};
```

210. 课程表 II

题目描述

现在你总共有 n 门课需要选，记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

例子1

```
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。
```

例子2

```
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
    因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

说明：

1 输入的先决条件是由边缘列表表示的图形，而不是邻接矩阵。详情请参见图的表示法。
2 你可以假定输入的先决条件中没有重复的边。

注意：
1 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2 通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3 拓扑排序也可以通过 BFS 完成。

思考

1 这是典型的拓扑排序，只不过需要复习一下入度，出度的概念，如果一个节点的入度为0，那么肯定可以放入到数组中去

参考实现1

实现1

```
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {number[]}
 */

// Runtime: 136 ms, faster than 29.13% of JavaScript online submissions for Course Schedule II.
// Memory Usage: 44.3 MB, less than 49.13% of JavaScript online submissions for Course Schedule II.
export default (numCourses, prerequisites) => {
  const inDegrees = new Array(numCourses).fill(0);
  for (const [val] of prerequisites) {
    // 获取每个节点的入度
    inDegrees[val]++;
  }
  const queue = [];
  for (let i = 0; i < inDegrees.length; i++) {
```

```
    if (inDegrees[i] === 0) {
      // 把入度为0的节点加入到队列中
      queue.push(i);
    }
  }
  const res = [];
  while (queue.length) {
    const first = queue.shift();
    // 节点的个数减去一
    numCourses--;
    res.push(first);
    for (const [val0, val1] of prerequisites) {
      if (val1 === first) {
        --inDegrees[val0];
        if (inDegrees[val0] === 0) {
          queue.push(val0);
        }
      }
    }
  }
  return numCourses === 0 ? res : [];
};
```

1059. 是否所有的路径从起点到终点

题目描述

给出一个有向边集合graph,和两个节点起始点source和结束点destination，求出是否所有的路径都可以从起始点source到结束点destination，如果符合这三种情况就返回true，否则返回false

1 至少有一条路径从起始点到结束点
2 如果从起始点到达一个出度为0的节点，那么这个节点肯定是结束点
3 从起始点到终点的路径数目是有限的。

例子1

```
输入：n = 3, edges = [[0,1],[0,2]], source = 0, destination = 2
输出：false
解释：从0到2有一条路径，但是从0到1，发现1的出度是0，但是1不等于2
```

例子2

```
输入：n = 4, edges = [[0,1],[0,3],[1,2],[2,1]], source = 0, destination = 3
输出：false
解释：因为这里存在一个循环从1到2，从2到1，所以返回false
```

思考

1 很简单，直接广度遍历，把是环的情况和到达一个节点但是不是终点的情况就可以了

参考实现1

实现1

```
export default (n, edges, source, destination) => {
  const graph = new Array(n);

  for (let i = 0; i < n; i++) {
    graph[i] = [];
  }

  const inDegrees = new Array(n).fill(0);

  for (let [key, val] of edges) {
    graph[key].push(val);
    ++inDegrees[val];
  }
  const queue = [source];
```

```
  while (queue.length) {
    const currNode = queue.shift();
    if (graph[currNode].length === 0 && currNode !== destination) {
      return fasle;
    }

    for (let node of graph[currNode]) {
      if (inDegrees[node] < 0) {
        return false;
      }
      --inDegrees[node];
      queue.push(node);
    }
  }

  return true;
};
```

1135. 最低成本联通所有城市

题目描述

有标记为从1到n的n个城市，然后给予一个connections，每个connections[i] = [city1, city2, cost]表示从city1到city2的代价是cost和从city2到city1的代价是cost

返回一个可以联通所有城市需要花费的最小代价，如果不能联通所有城市，则返回-1

例子1

```
输入：N = 3, connections = [[1,2,5],[1,3,6],[2,3,1]]
输出：6
解释：选择从1===2===3
```

例子2

```
输入：N = 4, connections = [[1,2,3],[3,4,4]]
输出：-1
解释：无法联通所有城市
```

思考

1 这是典型的求最小生成树的算法

prim 算法，这里算法很简单，切人的角度是从节点出发，首先任意选择一个节点，放入visited，然后选择所有和visited里边节点联通的代价最小的节点再加入到visited里边，一直循环。

参考实现1

2 克鲁斯卡尔(kruskal)算法

这个算法切人的角度是从边入手，首先选择所有边中代价最小的边，然后把边的两个节点加入到已经访问过的节点集合，然后继续寻找下一个边中代价最小的边，再把边的两个点再加入到已经访问过的集合里边，如果节点已经访问完了，则返回-1，如果没有，则继续寻找下一个代价最小的边，一直循环。

实现1

```
export default (n, connections) => {
  let res = 0;
  const visited = [1];
  while (visited.length !== n) {
    let min = Number.MAX_VALUE;
    let minNode;
    for (let i = 0; i < visited.length; i++) {
      for (let conn of connections) {
        const a = conn[0];
        const b = conn[1];
        const cost = conn[2];
        if (visited[i] === a && !visited.includes(b)) {
          if (cost < min) {
            min = cost;
```

```
            minNode = b;
          }
        } else if (visited[i] === b && !visited.includes(a)) {
          if (cost < min) {
            min = cost;
            minNode = a;
          }
        }
      }
    }
    if (minNode) {
      visited.push(minNode);
      res += min;
    } else {
      return -1;
    }
  }
  return res;
};
```

实现2

```
class Uf {
  constructor(n) {
    this.parent = new Array(n + 1).fill(0);
    this.size = new Array(n + 1).fill(0);
    for (let i = 0; i <= n; i++) {
      this.parent[i] = i;
      this.size[i] = 1;
    }
    this.count = n;
  }
  // 发现父元素，连成一个链表
  find(i) {
    if (i !== this.parent[i]) {
      this.parent[i] = this.find(this.parent[i]);
    }
    return this.parent[i];
  }
  // 连接起来
  union(i, j) {
    // 判断谁是谁的父元素,谁的比重大谁就是父元素
    if (this.size[i] > this.size[j]) {
      this.parent[j] = i;
      this.size[i] += this.size[j];
```

```
    } else {
      this.parent[i] = j;
      this.size[j] += this.size[i];
    }

    this.count--;
  }
}
export default (n, connections) => {
  connections.sort((a, b) => a[2] - b[2]);
  let res = 0;
  const uf = new Uf(n);
  for (let conn of connections) {
    const a = conn[0];
    const b = conn[1];
    const cost = conn[2];
    const pa = uf.find(a);
    const pb = uf.find(b);

    if (pa !== pb) {
      uf.union(pa, pb);
      res += cost;
    }
```

```
    if (uf.count === 1) return res;
  }
  return -1;
};
```

882. 细分图中的可到达结点

题目描述

给你一个无向图（原始图），图中有 n 个节点，编号从 0 到 n - 1 。你决定将图中的每条边细分为一条节点链，每条边之间的新节点数各不相同。

图用由边组成的二维数组 edges 表示，其中 edges[i] = [ui, vi, cnti] 表示原始图中节点 ui 和 vi 之间存在一条边，cnti 是将边细分后的新节点总数。注意，cnti == 0 表示边不可细分。

要细分边 [ui, vi] ，需要将其替换为 (cnti + 1) 条新边，和 cnti 个新节点。新节点为 x1, x2, ..., xcnti ，新边为 [ui, x1], [x1, x2], [x2, x3], ..., [xcnti+1, xcnti], [xcnti, vi] 。

现在得到一个新的 细分图 ，请你计算从节点 0 出发，可以到达多少个节点？节点 是否可以到达的判断条件 为：如果节点间距离是 maxMoves 或更少，则视为可以到达；否则，不可到达。

给你原始图和 maxMoves ，返回新的细分图中从节点 0 出发 可到达的节点数 。

例子1

```
输入：edges = [[0,1,10],[0,2,1],[1,2,2]], maxMoves = 6, n = 3
输出：13
解释：边的细分情况如上图所示。
可以到达的节点已经用黄色标注出来。
```

例子2

```
输入：edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], maxMoves = 10, n = 4
输出：23
```

例子3

```
输入：edges = [[1,2,4],[1,4,5],[1,3,1],[2,3,4],[3,4,5]], maxMoves = 17, n = 5
输出：1
解释：节点 0 与图的其余部分没有连通，所以只有节点 0 可以到达。
```

提示：

0 <= edges.length <= min(n * (n - 1) / 2, 104)
edges[i].length == 3
0 <= ui < vi < n
图中 不存在平行边
0 <= cnti <= 10^4
0 <= maxMoves <= 10^9
1 <= n <= 3000

思考

1 Dijkstra 无负边单源最短路算法,感兴趣的可以自己去看下。

在本题目中，首先找到所有节点到0节点的最短距离，如果最短距离小于maxMoves，则说明该节点可以到达。

然后再看下所有的边，看下每条边的两个节点，如果这条边的两个顶点可以到达，然后肯定可以访问到这条边上的节点，

参考实现1

实现1

```
/**
 * @param {number[][]} edges
 * @param {number} maxMoves
 * @param {number} n
 * @return {number}
```

```
 */
class Heap {
  constructor() {
    this.heap = [];
  }

  get length() {
    return this.heap.length;
  }

  compare(i, j) {
    if (!this.heap[j]) return false;
    return this.heap[i][1] > this.heap[j][1];
  }

  swap(i, j) {
    const temp = this.heap[i];
    this.heap[i] = this.heap[j];
    this.heap[j] = temp;
  }

  insert(num) {
    this.heap.push(num);
```

```
    let idx = this.length - 1;
    let parent = (idx - 1) >> 1;
    // 如果没有到达终点
    while (idx !== 0 && this.compare(parent, idx)) {
      this.swap(parent, idx);
      idx = parent;
      parent = (idx - 1) >> 1;
    }
  }

  remove() {
    if (this.length === 1) return this.heap.pop();
    let res = this.heap[0],
      idx = 0,
      left = 1 | (idx << 1),
      right = (1 + idx) << 1;
    this.heap[0] = this.heap.pop();
    while (this.compare(idx, left) || this.compare(idx, right)) {
      if (this.compare(left, right)) {
        this.swap(idx, right);
        idx = right;
      } else {
        this.swap(idx, left);
```

```
        idx = left;
      }
      left = 1 | (idx << 1);
      right = (1 + idx) << 1;
    }
    return res;
  }
}
export default (edges, maxMoves, n) => {
  let res = 0;
  // 最小堆
  const priorityQueue = new Heap();
  const visited = new Array(n).fill(0);
  const graph = new Array(n);
  for (let i = 0; i < n; i++) {
    graph[i] = [];
  }
  const distance = new Array(n).fill(Number.MAX_SAFE_INTEGER);
  for (let i = 0; i < edges.length; i++) {
    // 把两者的距离push进去
    graph[edges[i][0]].push([edges[i][1], edges[i][2]]);
    graph[edges[i][1]].push([edges[i][0], edges[i][2]]);
  }
```

```
  distance[0] = 0;
  priorityQueue.insert([0, distance[0]]);

  while (priorityQueue.length != 0) {
    let cur = priorityQueue.remove();
    const curNode = cur[0];
    // 如果已经访问过了，则执行下一个循环
    if (visited[curNode] === 1) continue;
    if (distance[curNode] <= maxMoves) res++;
    visited[curNode] = 1;
    for (let i of graph[curNode]) {
      // 发现从0节点到所有节点的最小距离
      if (distance[i[0]] > distance[curNode] + i[1] + 1) {
        distance[i[0]] = distance[curNode] + i[1] + 1;
        priorityQueue.insert([i[0], distance[i[0]]]);
      }
    }
  }
  // 能到到的节点前面在while循环里边已经统计完了，现在需要统计各个边上可以到达的节点
  for (let i = 0; i < edges.length; i++) {
    const a = maxMoves - distance[edges[i][0]] >= 0 ? maxMoves - distance[edges[i][0]] : 0;
    const b = maxMoves - distance[edges[i][1]] >= 0 ? maxMoves - distance[edges[i][1]] : 0;
    res += Math.min(edges[i][2], a + b);
```

```
  }
  return res;
};
```

第16章DIY数据结构

复杂数据结构

实现一个并查集

并查集的实现比较简单，不过就是使用递归，涉及到路径压缩和权重对比等操作，比较简单

```
class BingchaSet {
  constructor(n) {
    this.rank = [];
    this.father = [];
    for (let i = 0; i < n; i++) {
      this.rank[i] = 1;
      this.father[i] = i;
    }
  }
  find(i) {
    if (this.father[i] === i) {
```

```
      return i;
    }
    const parent = this.find(this.father[i]);
    this.father[i] = parent;
    return parent;
  }
  union(i, j) {
    const parentI = this.find(i);
    const parentJ = this.find(j);
    if (this.rank[parentI] <= this.rank[parentJ]) {
      this.father[parentI] = parentJ;
    } else {
      this.father[parentJ] = parentI;
    }
    if (this.rank[parentI] === this.rank[parentJ] && parentI != parentI) {
      this.rank[parentJ]++;
    }
  }
}
export default BingchaSet;
```

684. 冗余连接

题目描述

在本问题中, 树指的是一个连通且无环的无向图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以边组成的二维数组。每一个边的元素是一对[u, v] ，满足 u < v，表示连接顶点u 和v的无向图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 [u, v] 应满足相同的格式 u < v。

例子1

```
输入: [[1,2], [1,3], [2,3]]
输出: [2,3]
解释: 给定的无向图为:
  1
 / \
2 - 3
```

例子2

```
输入: [[1,2], [2,3], [3,4], [1,4], [1,5]]
输出: [1,4]
解释: 给定的无向图为:
5 - 1 - 2
  |   |
  4 - 3
```

注意：
1 输入的二维数组大小在 3 到 1000。
2 二维数组中的整数在1到N之间，其中N是输入数组的大小。

思考

1 这里很明显使用并查集来解决，因为前面我们已经介绍了什么是并查集。

题目本身并不难，就是判断是否有环，可以使用并查集的find方法，如果发现两个节点的祖先都是同一个，那就是我们需要找的。

参考实现2

实现1

```javascript
/**
 * @param {number[][]} edges
 * @return {number[]}
 */
import BingchaSet from "../bingchaSet/index";
// Runtime: 368 ms, faster than 5.94% of JavaScript online submissions for Redundant Connection.
// Memory Usage: 48.9 MB, less than 5.45% of JavaScript online submissions for Redundant Connection.
export default (edges) => {
  const len = edges.length;
  const bingset = new BingchaSet(len);
  for (let [u, v] of edges) {
    const x = bingset.find(u - 1);
    const y = bingset.find(v - 1);
    if (x !== y) {
      bingset.merge(u - 1, v - 1);
    } else {
      return [u, v];
    }
  }
};
```

146. LRU 缓存机制

题目描述

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制 。

实现 LRUCache 类：

```
LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。
```

例子1

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
```

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

注意：

1 1 <= capacity <= 3000

2 0 <= key <= 3000。

3 0 <= value <= 10^4

4 最多调用 3 * 10^4 次 get 和 put

思考

1 这里最难的就是如何找出最近最少使用的key值?

看了题解了解到，Map.keys()是按照set的顺序倒序输出的，这样就可以利用这一特性来进行解决寻找最近最少使用的key值

参考实现1

实现1

```
/**
 * @param {number} capacity
 */
var LRUCache = function (capacity) {
  this.map = new Map();
  this.max = capacity;
};

/**
 * @param {number} key
 * @return {number}
 */
LRUCache.prototype.get = function (key) {
  console.log(this.map, this.keyMap);
  if (this.map.has(key)) {
    const val = this.map.get(key);
    this.map.delete(key);
    this.map.set(key, val);
    return val;
  } else {
```

```
    return -1;
  }
};

/**
 * @param {number} key
 * @param {number} value
 * @return {void}
 */
// Runtime: 196 ms, faster than 64.75% of JavaScript online submissions for LRU Cache.
// Memory Usage: 51.3 MB, less than 59.00% of JavaScript online submissions for LRU Cache.
LRUCache.prototype.put = function (key, value) {
  console.log(this.map.size, this.max);
  if (this.map.has(key)) {
    this.map.delete(key);
  }
  this.map.set(key, value);
  if (this.map.size > this.max) {
    const firstKey = this.map.keys().next().value;
    this.cache.delete(firstKey);
  }
};
```

```
/**
 * Your LRUCache object will be instantiated and called as such:
 * var obj = new LRUCache(capacity)
 * var param_1 = obj.get(key)
 * obj.put(key,value)
 */
```

716. 最大栈

题目描述

设计一个支持 push，pop，top，peekMax 和 popMax 的 stack
1 peekMax 获取栈中的最大元素
2 popMax 获取栈中的最大元素并且删除

例子1

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
```

```
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

思考

1 使用两个栈，一个存储正常的push进来的数字，一个maxStack用来存储每次push进来的最大元素

参考实现1

实现1

```
class MaxStack {
  constructor() {
    this.normalStack = [];
    this.maxStack = [];
  }

  //O(1);
  push(val) {
```

```
    this.normalStack.push(val);
    const len = this.maxStack.length;
    if (len === 0 || (len > 0 && val >= this.maxStack[len - 1])) {
      this.maxStack.push(val);
    } else {
      this.maxStack.push(this.maxStack[len - 1]);
    }
  }
  //O(1);
  pop() {
    this.maxStack.pop();
    return this.normalStack.pop();
  }
  //O(1);
  // 获取当前最顶上的元素，但是不删除
  top() {
    if (this.normalStack.length > 0) {
      return this.normalStack[this.normalStack.length - 1];
    } else {
      return -1;
    }
  }
  //O(1);
```

```
// 获取最大的stack中max元素
peekMax() {
  if (this.maxStack.length > 0) {
    return this.maxStack[this.maxStack.length - 1];
  } else {
    return -1;
  }
}
//O(n);
// 删除最大的元素
popMax() {
  const res = this.peekMax();
  const temp = [];
  while (this.top() !== res) {
    const normalEle = this.normalStack.pop();
    temp.shift(normalEle);
    this.maxStack.pop();
  }
  this.pop();
  for (let val of temp) {
    this.push(val);
  }
  return res;
```

```
  }
}
export default MaxStack;
```

380. Insert Delete GetRandom O(1)

题目描述

设计一个插入、删除和随机取值均为 O(1) 时间复杂度的数据结构

例子1

```
输入：
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
[[], [1], [2], [2], [], [1], [2], []]
输出：
[null, true, false, true, 2, true, false, 2]
```

思考

1 使用一个map就可以了

参考实现1

实现1

```
/**
 * Initialize your data structure here.
 */
var RandomizedSet = function () {
  this.map = new Map();
};

/**
 * Inserts a value to the set. Returns true if the set did not already contain the specified element.
 * @param {number} val
 * @return {boolean}
 */
RandomizedSet.prototype.insert = function (val) {
  if (!this.map.has(val)) {
    this.map.set(val, val);
    return true;
  } else {
```

```
    return false;
  }
};

/**
 * Removes a value from the set. Returns true if the set contained the specified element.
 * @param {number} val
 * @return {boolean}
 */
RandomizedSet.prototype.remove = function (val) {
  if (this.map.has(val)) {
    this.map.delete(val);
    return true;
  } else {
    return false;
  }
};

/**
 * Get a random element from the set.
 * @return {number}
 */
//  Runtime: 228 ms, faster than 24.73% of JavaScript online submissions for Insert Delete GetRandom O(1).
```

```javascript
// Memory Usage: 49.3 MB, less than 15.96% of JavaScript online submissions for Insert Delete GetRandom O(1).
RandomizedSet.prototype.getRandom = function () {
  const random = Math.floor(this.map.size * Math.random());
  let count = 0;
  for (let [key, val] of this.map) {
    if (count === random) {
      return val;
    } else {
      ++count;
    }
  }
};

/**
 * Your RandomizedSet object will be instantiated and called as such:
 * var obj = new RandomizedSet()
 * var param_1 = obj.insert(val)
 * var param_2 = obj.remove(val)
 * var param_3 = obj.getRandom()
 */
```

432. 全 O(1) 的数据结构

题目描述

1 Inc(key) - 插入一个新的值为 1 的 key。或者使一个存在的 key 增加一，保证 key 不为空字符串。
2 Dec(key) - 如果这个 key 的值是 1，那么把他从数据结构中移除掉。否则使一个存在的 key 值减一。如果这个 key 不存在，这个函数不做任何事情。key 保证不为空字符串。
3 GetMaxKey() - 返回 key 中值最大的任意一个。如果没有元素存在，返回一个空字符串"" 。
4 GetMinKey() - 返回 key 中值最小的任意一个。如果没有元素存在，返回一个空字符串""。

思考

1 使用两个map，一个保存key对用的次数，一个保存次数对应的key值对象，然后通过一个双向链表链接起来，这样就可以达到取最大的key和最小的key都是O（1）

如果需要O(1), map和双向链表都是一种选择，

参考实现1

实现1

```
/**
 * Initialize your data structure here.
 */
```

```
class Bucket {
  constructor(val, preBucket, nextBucket) {
    this.count = val;
    this.next = nextBucket || null;
    this.pre = preBucket || null;
    this.keySet = new Set();
  }
}
var AllOne = function () {
  // key2count map
  this.key2countMap = new Map();
  // sameCount map, 每个count对应是一个双向的Bucket链表
  this.sameCountBucketMap = new Map();
  // this.head 指向最小的val
  this.head = new Bucket(Number.MIN_VALUE);
  // this.tail 指向最大的val
  this.tail = new Bucket(Number.MAX_VALUE);
  this.head.next = this.tail;
  this.tail.pre = this.head;
};

/**
 * Inserts a new key <Key> with value 1. Or increments an existing key by 1.
```

```
 * @param {string} key
 * @return {void}
 */
AllOne.prototype.inc = function (key) {
 // 如果存在，则把它加入到相同count的链表中去
 if (this.key2countMap.has(key)) {
   this.changeKey(key, 1);
 } else {
   this.key2countMap.set(key, 1);
   if (this.head.next.count !== 1) {
     this.addBucketAfter(new Bucket(1), this.head);
   }
   this.head.next.keySet.add(key);
   this.sameCountBucketMap.set(1, this.head.next);
 }
};

/**
 * Decrements an existing key by 1. If Key's value is 1, remove it from the data structure.
 * @param {string} key
 * @return {void}
 */
AllOne.prototype.dec = function (key) {
```

```
  if (this.key2countMap.has(key)) {
    const count = this.key2countMap.get(key);
    if (count === 1) {
      this.key2countMap.delete(key);
      this.removeKeyFromBucket(this.sameCountBucketMap.get(count), key);
    } else {
      this.changeKey(key, -1);
    }
  }
};

/**
 * Returns one of the keys with maximal value.
 * @return {string}
 */
AllOne.prototype.getMaxKey = function () {
  return this.tail.pre === this.head ? "" : this.tail.pre.keySet[Symbol.iterator]().next().value;
};

/**
 * Returns one of the keys with Minimal value.
 * @return {string}
 */
```

```
AllOne.prototype.getMinKey = function () {
  return this.head.next === this.tail ? "" : this.head.next.keySet[Symbol.iterator]().next().value;
};
// 把相同count的Bucket加入到sameCountBucketMap
AllOne.prototype.changeKey = function (key, offset) {
  // 首先修改原来存在key值
  const count = this.key2countMap.get(key);
  this.key2countMap.set(key, count + offset);
  const curBucket = this.sameCountBucketMap.get(count);
  let newBucket;
  if (this.sameCountBucketMap.has(count + offset)) {
    newBucket = this.sameCountBucketMap.get(count + offset);
  } else {
    // add new Bucket
    newBucket = new Bucket(count + offset);
    this.sameCountBucketMap.set(count + offset, newBucket);
    this.addBucketAfter(newBucket, offset === 1 ? curBucket : curBucket.pre);
  }
  newBucket.keySet.add(key);
  this.removeKeyFromBucket(curBucket, key);
};
AllOne.prototype.removeKeyFromBucket = function (bucket, key) {
  bucket.keySet.delete(key);
```

```
  if (bucket.keySet.size === 0) {
    this.removeBucketFromList(bucket);
    this.sameCountBucketMap.delete(bucket.count);
  }
};
AllOne.prototype.removeBucketFromList = function (bucket) {
  bucket.pre.next = bucket.next;
  bucket.next.pre = bucket.pre;
  bucket.next = null;
  bucket.pre = null;
};
AllOne.prototype.addBucketAfter = function (newBucket, preBucket) {
  newBucket.pre = preBucket;
  newBucket.next = preBucket.next;
  preBucket.next.pre = newBucket;
  preBucket.next = newBucket;
};
/**
 * Your AllOne object will be instantiated and called as such:
 * var obj = new AllOne()
 * obj.inc(key)
 * obj.dec(key)
 * var param_3 = obj.getMaxKey()
```

```
 * var param_4 = obj.getMinKey()
 */
```