热题 100 道

0001 两数之和

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，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 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] 。
```

示例 2：

```
输入：nums = [3,2,4], target = 6
输出：[1,2]
```

示例 3：

```
输入：nums = [3,3], target = 6
输出：[0,1]
```

解法1:暴力枚举

```
var twoSum = function(nums, target) {
 var results = []
 for (let i = 0; i < nums.length; i++){
     for (let j = i + 1; j < nums.length; j++) {
         if (nums[i] + nums[j] === target) {
             results.push(i)
             results.push(j)
             return results
         }
     }
 }
};
```

- 时间复杂度：O(N^2)，其中N是数组中的元素数量。最坏情况下数组中任意两个数都要被匹配一次
- 空间复杂度：O(1)

解法2:哈希表

> Hash表时一种数据结构，可以把它理解为一个key-value对，一个key-value对构成一个entry，只不过在Hash表中，需要先将原始的key经过Hash函数运算，映射到entry中的key上，然后就可以通过key值进行查询等操作，重要的是设计Hash函数，避免Hash冲突

注意到方法一的时间复杂度较高的原因是寻找 `target - x` 的时间复杂度过高。因此，我们需要一种更优秀的方法，能够快速寻找数组中是否存在目标元素。如果存在，我们需要找出它的索引。

使用哈希表，可以将寻找 `target - x` 的时间复杂度降低到从 O(N)*O*(*N*) 降低到 O(1)*O*(1)。

这样我们创建一个哈希表，对于每一个 `x`，我们首先查询哈希表中是否存在 `target - x`，然后将 `x` 插入到哈希表中，即可保证不会让 `x` 和自己匹配。

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
function HashTable() { //Hash表构造函数
  var size = 0
  var entry = new Object()

  this.push = function (key, value) {
    ++size
    entry[key] = value

  }

  this.get = function (key) {
    if (key in entry) {
      return entry[key]
    } else {
      return null
    }
  }

  this.containsKey = function (key) {
    return (key in entry)
  }
}

var twoSum = function (nums, target) {
  var results = []
  var hash = new HashTable()//实例化Hash表
  hash.push(nums[0], 0)
  for (let i = 1; i < nums.length; i++) {
    if (hash.containsKey(target - nums[i])) {
      results.push(hash.get(target - nums[i]))
      results.push(i)
      return results
    }
    hash.push(nums[i], i)
  }
}
```

还可以通过JS自带的MAP对象，可以建立映射，key-value对

```
var twoSum = function(nums, target) {
```

```
    let len = nums.length;
    // 创建 MAP
    const MAP = new Map();
    // 由于第一个元素在它之前一定没有元素与之匹配，所以先存入哈希表
    MAP.set(nums[0], 0);
    for (let i = 1; i < len; i++) {
        // 提取共用
        let other = target - nums[i];
        // 判断是否符合条件，返回对应的下标
        if (MAP.get(other) !== undefined) return [MAP.get(other), i];
        // 不符合的存入hash表
        MAP.set(nums[i], i)
    }
};
```

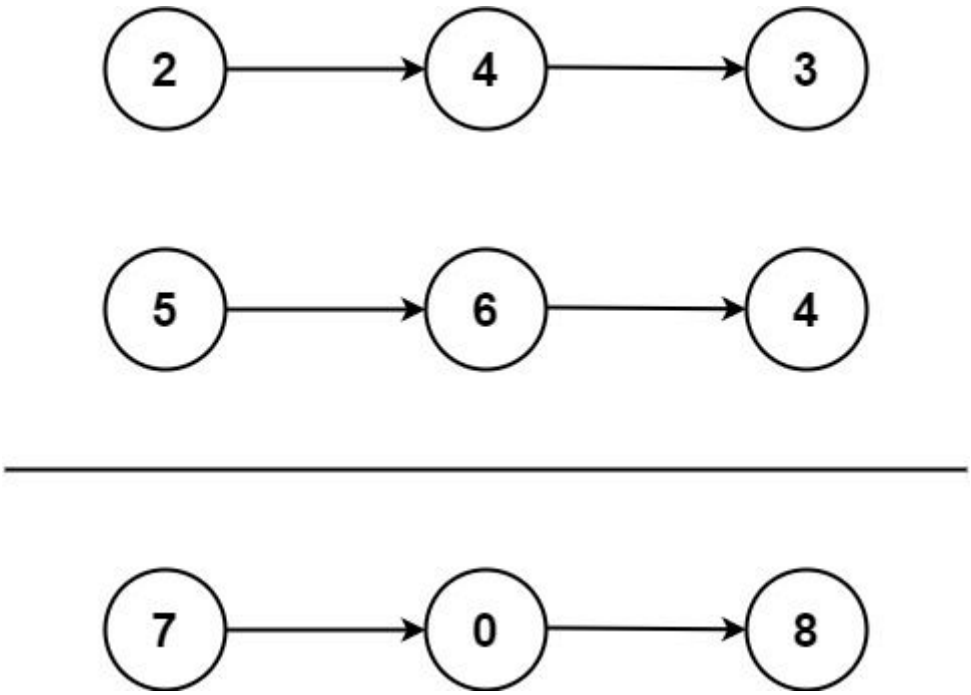
0002 两数相加

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储 **一位** 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：




```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807.
```

示例 2：

```
输入：l1 = [0], l2 = [0]
输出：[0]
```

示例 3:

```
输入：l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
输出：[8,9,9,9,0,0,0,1]
```

提示:

- 每个链表中的节点数在范围 `[1, 100]` 内
- `0 <= Node.val <= 9`
- 题目数据保证列表表示的数字不含前导零

```
/**
 * 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}
 */
var addTwoNumbers = function (l1, l2) {
    var zhengshu = 0
    var result = new ListNode(-1)//result始终指向链表头
    var cur = result//指向链表头，对链表进行操作
    while(l1 || l2 || zhengshu ) {
        var sum = (l1 ? l1.val : 0) + (l2 ? l2.val : 0) + zhengshu//判断l1、l2是否为空
        zhengshu = Math.floor(sum/10)
        cur.next = new ListNode(sum % 10)
        cur = cur.next
        l1 = l1 ? l1.next : l1;
        l2 = l2 ? l2.next : l2;
    }
    return result.next
};
```

0003 无重复字符的最长子串

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。

示例 1:

```
输入：s = "abcabcbb"
输出：3
解释：因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: s = "pwwkew"
输出: 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。
```

示例 4:

```
输入: s = ""
输出: 0
```

提示:

- `0 <= s.length <= 5 * 10^4`
- `s` 由英文字母、数字、符号和空格组成

利用**滑动窗口**，i为窗口的左指针，right为窗口右指针，右指针不断右移探寻最大无重复子串，当遇到重复时，左指针自增，还是重复，继续自增，直到无重复，用max函数对当前子串长度和记录最大长度进行比较并记录

```
/**
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function(s) {
    const n = s.length
    var number = 0
    var result = new Set()
    var right = 0
    for (let i = 0; i < n; i++) {
        let ans = 0
        if(i != 0) {
            result.delete(s.charAt(i-1))
        }
        while((right < n) && !result.has(s.charAt(right))) {
            result.add(s.charAt(right))
            right++
        }
        ans = result.size
        number = Math.max(number, ans)
    }
    return number
};
```

0004 寻找两个正序数组的中位数

给定两个大小分别为 `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** 。

示例 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
```

示例 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
```

示例 3：

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

示例 4：

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

示例 5：

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

提示：

- `nums1.length == m`
- `nums2.length == n`
- `0 <= m <= 1000`
- `0 <= n <= 1000`
- `1 <= m + n <= 2000`
- `-10^6 <= nums1[i], nums2[i] <= 10^6`

解法1：比大小后填充

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function (nums1, nums2) {
  var m = nums1.length
  var n = nums2.length
  var results = []
  var p1 = 0,p2 = 0,count = 0
  while (p1 < m && p2 < n) {
    if (nums1[p1] > nums2[p2]) {
      results.push(nums2[p2])
      p2++
    } else if (nums1[p1] < nums2[p2]) {
      results.push(nums1[p1])
      p1++
    } else if ((nums1[p1] === nums2[p2])) {
      results.push(nums1[p1])
      results.push(nums2[p2])
      p1++
      p2++
    }
  }
  while (p1 < m) {
    results.push(nums1[p1])
    p1++
  }
  while (p2 < n) {
    results.push(nums2[p2])
    p2++
  }
  console.log(results);
  count = results.length
  if (count % 2 === 0) {
    return (results[count / 2] + results[count / 2 - 1]) / 2
  } else {
    return results[Math.floor(count / 2)]
  }
};
```

解法2：二分法

```
/**
 * 二分解法
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function(nums1, nums2) {
  // make sure to do binary search for shorten array
  if (nums1.length > nums2.length) {
    [nums1, nums2] = [nums2, nums1]
  }
  const m = nums1.length
  const n = nums2.length
  let low = 0
```

```
  let high = m
  while(low <= high) {
    const i = low + Math.floor((high - low) / 2)
    const j = Math.floor((m + n + 1) / 2) - i

    const maxLeftA = i === 0 ? -Infinity : nums1[i-1]
    const minRightA = i === m ? Infinity : nums1[i]
    const maxLeftB = j === 0 ? -Infinity : nums2[j-1]
    const minRightB = j === n ? Infinity : nums2[j]

    if (maxLeftA <= minRightB && minRightA >= maxLeftB) {
      return (m + n) % 2 === 1
        ? Math.max(maxLeftA, maxLeftB)
        : (Math.max(maxLeftA, maxLeftB) + Math.min(minRightA, minRightB)) /
2
    } else if (maxLeftA > minRightB) {
      high = i - 1
    } else {
      low = low + 1
    }
  }
};
```

0005 最长回文子串

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

示例 1：

```
输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案。
```

示例 2：

```
输入：s = "cbbd"
输出："bb"
```

示例 3：

```
输入：s = "a"
输出："a"
```

示例 4：

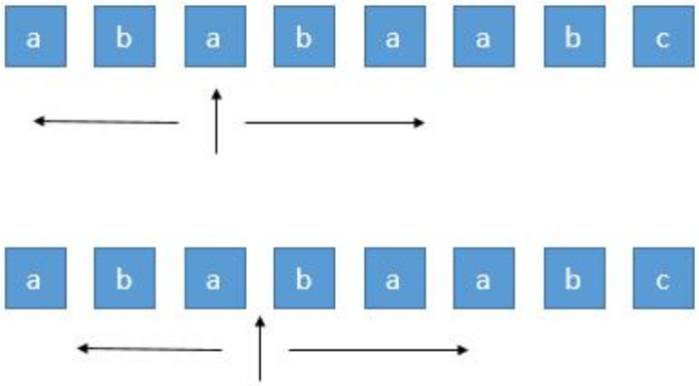
```
输入：s = "ac"
输出："a"
```

提示：

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母（大写和/或小写）组成

解法1：中心扩散法

我们知道回文串一定是对称的，所以我们可以每次循环选择一个中心，进行左右扩展，判断左右字符是否相等即可。



由于存在奇数的字符串和偶数的字符串，所以我们需要从一个字符开始扩展，或者从两个字符之间开始扩展，所以总共有 `n+n-1` 个中心。

```
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function(s) {
    let start = 0, end = 0;
    for(let i = 0; i < s.length; i ++) {
        let len1 = expandAroudcenter(s, i, i);
        let len2 = expandAroudcenter(s, i, i + 1);
        let len = Math.max(len1, len2);
        if(len >= (end - start + 1)) {
            start = i - Math.floor((len - 1) / 2);
            end = i + Math.floor(len / 2)
        }
    }
    return s.substring(start, end+1)
};

function expandAroudcenter(s, left, right) {
    let L = left, R = right
    while(L >= 0 && R < s.length && s.charAt(L) === s.charAt(R)) {
        L--;
        R++;
    }
    return R-L-1
}
```

时间复杂度：$O(n^2)$

空间复杂度：$O(1)$

解法2：动态规划

字符	b	a	b	a	b
下标	0	1	2	3	4

状态转移方程：dp[i][j] = (s[i] == s[j]) and (j - i < 3 or dp[i + 1][j - 1])

由于 dp[i][j] 参考它左下方的值：

(1) 先升序填列；

(2) 再升序填行。

子串右边界 / 子串左边界	0	1	2	3	4
0	TRUE	FALSE	TRUE	FALSE	TRUE
1		TRUE	FALSE	TRUE	FALSE
2			TRUE	FALSE	TRUE
3				TRUE	FALSE
4					TRUE

当前状态是否是回文串，取决于两个因素：左右两端是否相等 and 去掉左右两端的子串是否为回文串

0006 Z字形变换

将一个给定字符串 `s` 根据给定的行数 `numRows` ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 `"PAYPALISHIRING"` 行数为 `3` 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1：

```
输入：s = "PAYPALISHIRING", numRows = 3
输出："PAHNAPLSIIGYIR"
```

示例 2：

```
输入：s = "PAYPALISHIRING", numRows = 4
输出："PINALSIGYAHRPI"
解释：
P     I    N
A   L S  I G
Y A   H R
P     I
```

示例 3：

```
输入：s = "A", numRows = 1
输出："A"
```

提示：

- `1 <= s.length <= 1000`
- `s` 由英文字母（小写和大写）、`','` 和 `'.'` 组成
- `1 <= numRows <= 1000`

解法：按行排序

创建一个数组，存放numRows个元素，通过对down值的判断，依次从原字符串中取出字符放入

```
/**
 * @param {string} s
 * @param {number} numRows
 * @return {string}
 */
var convert = function(s, numRows) {
    if(numRows == 1)
        return s;

    const len = Math.min(s.length, numRows);
    const rows = [];
    for(let i = 0; i< len; i++) rows[i] = "";
    let loc = 0;
    let down = false;

    for(const c of s) {
        rows[loc] += c;
        if(loc == 0 || loc == numRows - 1)
            down = !down;
        loc += down ? 1 : -1;
    }

    let ans = "";
    for(const row of rows) {
        ans += row;
    }
    return ans;
};
```

0007 整数反转

给你一个 32 位的有符号整数 `x` ，返回将 `x` 中的数字部分反转后的结果。

如果反转后整数超过 32 位的有符号整数的范围 `[-2^31, 2^31 - 1]` ，就返回 0。

假设环境不允许存储 64 位整数（有符号或无符号）。

示例 1：

```
输入：x = 123
输出：321
```

示例 2：

```
输入：x = -123
输出：-321
```

示例 3：

```
输入：x = 120
输出：21
```

示例 4：

```
输入：x = 0
输出：0
```

提示：

- `-2^31 <= x <= 2^31 - 1`

解法：

本题的关键在于对范围的处理，由于题目限制只允许32位整数出现，所以要对反转后数值大小进行判断。

考虑 x > 0的情况，记

$$INT\_MAX = 2^{31} - 1 = 2147483647$$

由于

$$\begin{aligned} INT\_MAX &= \left\lfloor \frac{INT\_MAX}{10} \right\rfloor \cdot 10 + (INT\_MAX \bmod 10) \\ &= \left\lfloor \frac{INT\_MAX}{10} \right\rfloor \cdot 10 + 7 \end{aligned}$$

则不等式 rev*10 + digit <= INT_MAX 等价于

$$rev \cdot 10 + \$digit\$ \leq \left\lfloor \frac{INT\_MAX}{10} \right\rfloor \cdot 10 + 7$$

移项得

$$(rev - \left\lfloor \frac{INT\_MAX}{10} \right\rfloor) \cdot 10 \leq 7 - digit$$

讨论该不等式成立的条件：

若 $rev > \lfloor \frac{INT\_MAX}{10} \rfloor$，由于 $digit \geq 0$，不等式不成立。

若 $rev = \lfloor \frac{INT\_MAX}{10} \rfloor$，当且仅当 $digit \leq 7$ 时，不等式成立。

若 $rev < \lfloor \frac{INT\_MAX}{10} \rfloor$，由于 $digit \leq 9$，不等式成立。

注意到当 $rev = \lfloor \frac{INT\_MAX}{10} \rfloor$ 时若还能推入数字，则说明 x 的位数与 INT_MAX 的位数相同，且要推入的数字 $digit$ 为 x 的最高位。由于 x 不超过 INT_MAX，因此 $digit$ 不会超过 INT_MAX 的最高位，即 $digit \leq 2$。所以实际上当 $rev = \lfloor \frac{INT\_MAX}{10} \rfloor$ 时不等式必定成立。

因此判定条件可简化为：**当且仅当** $rev \leq \lfloor \frac{INT\_MAX}{10} \rfloor$ **时，不等式成立。**

$x < 0$ 的情况类似，留给读者自证，此处不再赘述。

综上所述，判断不等式

$$-2^{31} \leq rev \cdot 10 + digit \leq 2^{31} - 1$$

是否成立，可改为判断不等式

$$\lceil \frac{-2^{31}}{10} \rceil \leq rev \leq \lfloor \frac{2^{31} - 1}{10} \rfloor$$

是否成立，若不成立则返回 0。

```
/**
 * @param {number} x
 * @return {number}
 */
var reverse = function(x) {
    let res = 0; //存放结果
    let digit = 0; //存放整除后的余数

    while(x !== 0) {
        if(res < ~~(Math.pow(-2, 31)/10) || res > ~~((Math.pow(2, 31)-1)/10)) return 0;
        digit = x % 10;
        res = res * 10 + digit;
        x = ~~(x / 10);
    }
    return res;
};
```

注意：在JS求整操作中，尽量使用~~，Math.floor()在此处不可用，它是向下取整，而不是抹去小数，例如Math.floor(-1.5) = -2

0010 正则表达式匹配

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。

示例 1：

```
输入：s = "aa" p = "a"
输出：false
解释："a" 无法匹配 "aa" 整个字符串。
```

示例 2：

```
输入：s = "aa" p = "a*"
输出：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。
```

示例 3：

```
输入：s = "ab" p = ".*"
输出：true
解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
```

示例 4：

```
输入：s = "aab" p = "c*a*b"
输出：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"。
```

示例 5：

```
输入：s = "mississippi" p = "mis*is*p*."
输出：false
```

提示：

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

首先解题要先理解正则表达式具体匹配规则,先来做个最简单的假设:

- 假设`s=ab`,`p=a*b*`,那么毫无疑问`s.match(p)`是成立的

- 根据上面的例子,`p`变成`a*b*c`,那么这就是不成立的，因为`s`并没有`c`，如果`p`再加一个`*`变成`a*b*c*`，那么匹配又会变成立，因为`c`这个时候可以匹配为0个
- 那么当匹配到`*`号的时候具体到代码上逻辑是这样的:

```
if(p.charAt(i) === '*'){
  dp[i+1] = dp[i-1]
}
```

以上的逻辑思路就是当匹配到`a*b*c*`中时，判断最后一个`*`是否匹配成立只需要看第2个`*`,即`a*b*`是否匹配，如果匹配那么`a*b*c*`也是成立的,假如`s`换成是`ad`，那么走到`a*b`的时候就已经不匹配了,经过状态转移`a*b*c*`也不会匹配，所以使用**动态规划**就再合适不过

第一步：建立二维数组

假设`s=abbcddd`,`p=a*b*.dd*`,因此先建立一张对应二维数组的表:

0:false 1:true

j

i	s\p	''	a	*	b	*	.	d	d	*
	''									
	a									
	b									
	b									
	c									
	d									
	d									
	d									

为何要在`p`前面加一个空字符串(只是在表格中加)，原因也很简单，假设当`s=''`,`p=a*`时匹配是成立的，那么根据上面的逻辑`dp[0][j+1]=dp[0][j-1]`,那么就需要一个空值为true才能确保转移到`a*`后仍然为true

然后先建立一种简单的匹配情况，字符可以先把`s`假设为空字符串，`p`不变，那么可以s匹配p的结果如下(假设0为false,1为true):

0:false 1:true

j

i	s\p	''	a	*	b	*	.	d	d	*
	''	1	0	1	0	1	0	0	0	0

如上图，当`p[j]='.'`的时候可以当成是任意字符串但不能为空，因此匹配也为false，结合上述逻辑，一开始先匹配第一个空字符的逻辑可以总结为:

```
    dp[0][0] = true;
    for (let i = 1; i < p.length; i++) {
        if (p.charAt(i) === "*") {
            dp[0][i + 1] = dp[0][i - 1]
        }
    }
```

这个时候匹配完的 `dp[0]` 意义就是后续 `dp[i][j]` 进行状态转移的判断依据

第二步: 匹配状态转移

在 `dp[0]` 全部赋值后开始对字符串 `s` 和模式字符串 `p` 进行逐一匹配，那么匹配过程一共会碰到三种情况:

- `p[j]` 为普通字符时:

  如果 `s[i]===p[j]` ,那么通过 `dp[i+1][j+1]=dp[i][j]` 判断，例子如下图:

s\p	''	a
''	1	0
a	0	1

- `p[j]` 为 `.` 时:

  可以把 `p[j]` 看成任意但不为空的字符，也是通过 `dp[i+1][j+1]=dp[i][j]` 判断，例子如下图:

s\p	''	a	.
''	1	0	0
a	0	1	0
b	0	0	1

- `p[j]` 为 `*` 时:

1.如果`p[j-1]!==s[i]&&p[j-1]!=='.'`时，和上面的`dp[0][j+1]=dp[0][j-1]`的逻辑一样，把前一个`*`的状态值转移过来，如下图:

s\p	''	a	*	b	*
''	1	0	1	0	1
a	0	1	1	0	1

当`i=0,j=3`时,`s[0]!==p[2]`，因此`dp[i+1][j+1]=false`

2.当`p[j-1]==s[i]`时，对`*`可以匹配成功的情况可以分为三种，假设`p[j-1]=b`，出现的个数为`n`:

- `n=0`时，复用上面的逻辑，即`dp[i+1][j+1]=dp[i+1][j-1]`
- `n=1`时，`dp[i+1][j+1]`只需要等于`dp[i+1][j]`的值即可，如下图:

s\p	''	b	*
''	1	0	1
b	0	1	1

- `n>=2`时，这个时候`dp[i+1][j]`和`dp[i+1][j-1]`都有可能为`false`，如下图:

s\p	''	a	*	b	*
''	1	0	1	0	1
a	0	1	1	0	1
b	0	0	0	1	1
b	0	0	0	0	1
b	0	0	0	0	1

那么在出现多个 `b` 后 `dp[i+1][j+1]` 可以根据 `dp[i][j+1]` 判断

综上所述，当 `p.charAt(j) === '*'` 时，具体逻辑可以归纳为:

```
 if (p.charAt(j - 1) !== s.charAt(i) && p.charAt(j - 1) !== '.') {
     dp[i + 1][j + 1] = dp[i + 1][j - 1]
 } else {
     dp[i + 1][j + 1] = (dp[i + 1][j] || dp[i][j + 1] || dp[i + 1][j - 1])
 }
```

匹配完成后返回 `dp[s.length][p.length]` 就是最终结果。

```
/**
 * @param {string} s
 * @param {string} p
 * @return {boolean}
 */
var isMatch = function(s, p) {
 let dp = Array(s.length + 1);
    for (let i = 0; i < dp.length; i++) {
        dp[i] = Array(p.length + 1).fill(false)
    }
    dp[0][0] = true;
    for (let i = 1; i < p.length; i++) {
        if (p.charAt(i) === "*") {
            dp[0][i + 1] = dp[0][i - 1]
        }
    }

    for (let i = 0; i < s.length; i++) {
        for (let j = 0; j < p.length; j++) {
            if (p.charAt(j) === '.') {
                dp[i + 1][j + 1] = dp[i][j]
            }

            if (p.charAt(j) === s.charAt(i)) {
                dp[i + 1][j + 1] = dp[i][j]
```

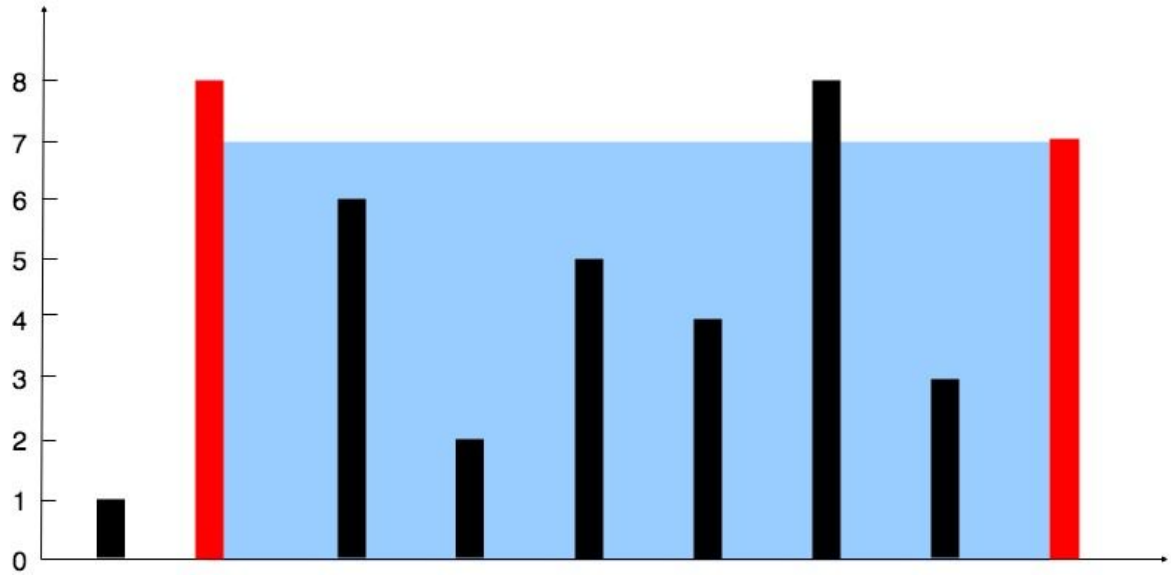
```
            }

            if (p.charAt(j) === '*') {
                if (p.charAt(j - 1) !== s.charAt(i) && p.charAt(j - 1) !==
'.') {
                    dp[i + 1][j + 1] = dp[i + 1][j - 1]
                } else {
                    dp[i + 1][j + 1] = (dp[i + 1][j] || dp[i][j + 1] || dp[i
+ 1][j - 1])
                }
            }
        }
    }
    return dp[s.length][p.length]
};
```

0011盛水最多的容器

给你 `n` 个非负整数 $a_1, a_2, ..., a_n$，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 `n` 条垂直线，垂直线 `i` 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

示例 1:



输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例 2:

输入：height = [1,1]
输出：1

示例 3：

```
输入：height = [4,3,2,1,4]
输出：16
```

示例 4：

```
输入：height = [1,2,1]
输出：2
```

提示：

- `n = height.length`
- `2 <= n <= 3 * 10^4`
- `0 <= height[i] <= 3 * 10^4`

解法：双指针（对撞指针）

对撞指针就是左右两个指针对向移动

如果我们移动数字较大的那个指针，那么前者「两个指针指向的数字中较小值」不会增加，后者「指针之间的距离」会减小，那么这个乘积会减小。因此，我们移动数字较大的那个指针是不合理的。因此，我们移动数字较小的那个指针。

但是

```javascript
/**
 * @param {number[]} height
 * @return {number}
 */
var maxArea = function (height) {
  var left = 0;
  var right = height.length - 1;
  var area = 0;
  while (right >= left) {
    let num = Math.min(height[left], height[right]) * (right - left)
    area = num >= area ? num : area;
    if (height[right] >= height[left]) {
      left++;
    } else {
      right--;
    }
  }
  return area;
};
```

0015 三数之和

给你一个包含 `n` 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a，b，c，使得 $a + b + c = 0$？请你找出所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
```

示例 2：

```
输入：nums = []
输出：[]
```

示例 3：

```
输入：nums = [0]
输出：[]
```

提示：

- `0 <= nums.length <= 3000`
- `-10^5 <= nums[i] <= 10^5`

解法：排序 + 双指针

1. 特判，对于数组长度 n，如果数组为 $null$ 或者数组长度小于 3，返回 $[]$。
2. 对数组进行排序。
3. 遍历排序后数组：
    - 若 $nums[i] > 0$：因为已经排序好，所以后面不可能有三个数加和等于 0，直接返回结果。
    - 对于重复元素：跳过，避免出现重复解
    - 令左指针 $L = i + 1$，右指针 $R = n - 1$，当 $L < R$ 时，执行循环：
        - 当 $nums[i] + nums[L] + nums[R] == 0$，执行循环，判断左界和右界是否和下一位置重复，去除重复解。并同时将 L, R 移到下一位置，寻找新的解
        - 若和大于 0，说明 $nums[R]$ 太大，R 左移
        - 若和小于 0，说明 $nums[L]$ 太小，L 右移

```javascript
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function(nums) {
```

```
    let result = [];
    if(nums.length < 3) return result;
    nums.sort((a,b) => a - b);
    for (let i = 0; i < nums.length; i ++) {
        if(nums[i] > 0) break;
        if(i > 0 && nums[i] === nums[i-1]) continue;
        let left = i + 1, right = nums.length - 1;
        while (left < right) {
            const sum = nums[i] + nums[left] + nums[right]
            if (sum === 0) {
                result.push([nums[i], nums[left], nums[right]]);
                while (nums[left] === nums[left + 1]) left++;
                left++;
                while (nums[right] === nums[right - 1]) right--;
                right--;
            }
            else if (sum > 0) right--;
            else if (sum < 0) left ++;
        }
    }
    return result;
};
```

0017 电话号码的字母组合

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例 1:

```
输入：digits = "23"
输出：["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

示例 2:

```
输入：digits = ""
输出：[]
```

示例 3:

```
输入：digits = "2"
输出：["a","b","c"]
```

提示:

- `0 <= digits.length <= 4`
- `digits[i]` 是范围 `['2', '9']` 的一个数字。

解法：回溯法

深度优先搜索

我理解的回溯法是一种递归，对于**找出全部可能**的题目，适合用回溯

- 首先通过Map建立一个Hash表，映射数字和字母的关系
- 创建backtrace方法，nowcontent是当前拼接的字符串，left是剩余的数字
- 当剩余数字长度为0时，将nowcontent作为结果push
- 否则，在当前数字映射的字母中循环，继续调用backtrace，此时传入拼接好的字符串nowcontent+当前字母char，剩余数字left左边slice一位，传入函数，进行递归

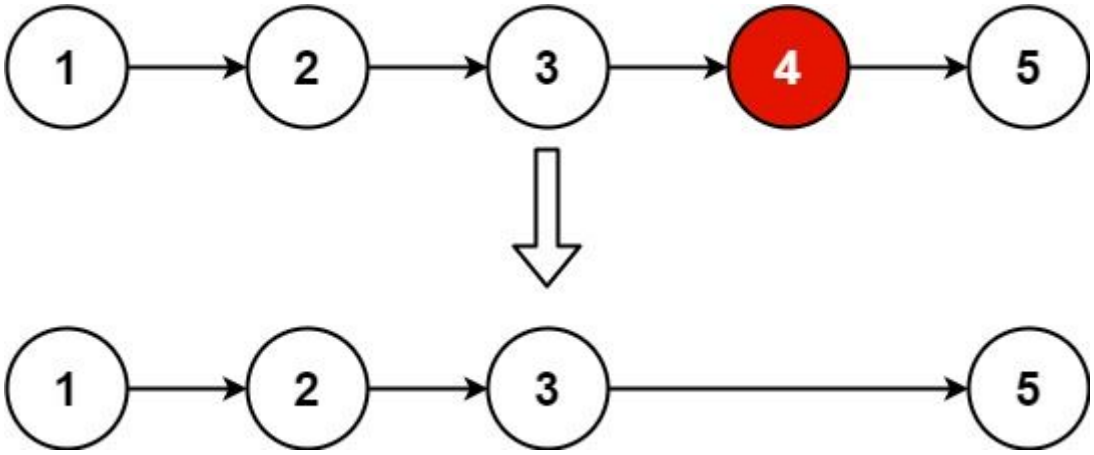
```
/**
 * @param {string} digits
 * @return {string[]}
 */
var letterCombinations = function(digits) {
    const res = [];
    if(digits.length === 0) return res;

    const number = new Map();
    number.set('2', 'abc');
    number.set('3', 'def');
    number.set('4', 'ghi');
    number.set('5', 'jkl');
    number.set('6', 'mno');
    number.set('7', 'pqrs');
    number.set('8', 'tuv');
    number.set('9', 'wxyz');

    function backtrace(nowcontent, left) {
        if(left.length === 0) {
            res.push(nowcontent)
        }else {
            for (let char of number.get(left.charAt(0))) {
                backtrace(nowcontent + char, left.slice(1));
            }
        }
    }

    backtrace('',digits);
    return res;
};
```

0019 删除链表倒数第N个节点

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？

示例 1：




```
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```

示例 2：

```
输入：head = [1], n = 1
输出：[]
```

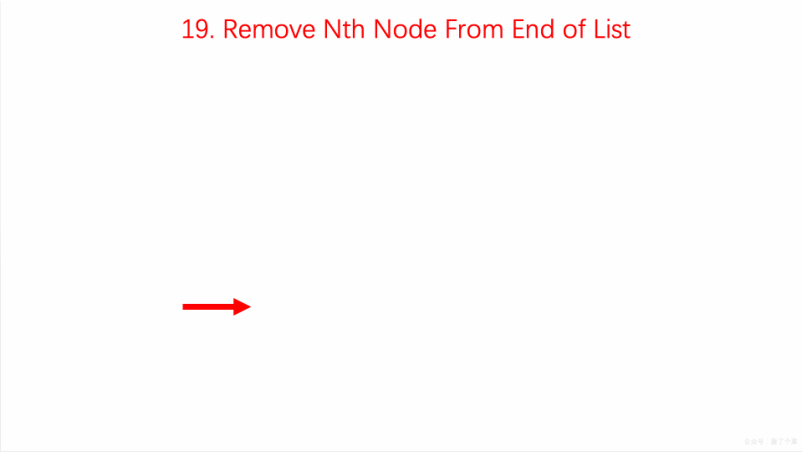
示例 3：

```
输入：head = [1,2], n = 1
输出：[1]
```

提示：

- 链表中结点的数目为 `sz`
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

解法：快慢指针




```
/**
 * 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}
 */
var removeNthFromEnd = function(head, n) {
    const markHead = head;
    let left = markHead, right = markHead;
    // 快先走 n+1 步
    while(n--) {
        right = right.next;
    }
    // fast、slow 一起前进
    while(right.next && right) {
        right = right.next
        left = left.next
    }
    left.next = left.next.next;
    return markHead;
};
```

0020 有效的括号

给定一个只包括 `'('`，`')'`，`'{'`，`'}'`，`'['`，`']'` 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

示例 1：

```
输入：s = "()"
输出：true
```

示例 2：

```
输入：s = "()[]{}"
输出：true
```

示例 3：

```
输入：s = "(]"
输出：false
```

示例 4：

```
输入：s = "([)]"
输出：false
```

示例 5：

```
输入：s = "{[]}"
输出：true
```

提示：

- `1 <= s.length <= 10^4`
- `s` 仅由括号 `'()[]{}'` 组成

解法：栈

先进先出，后进后出

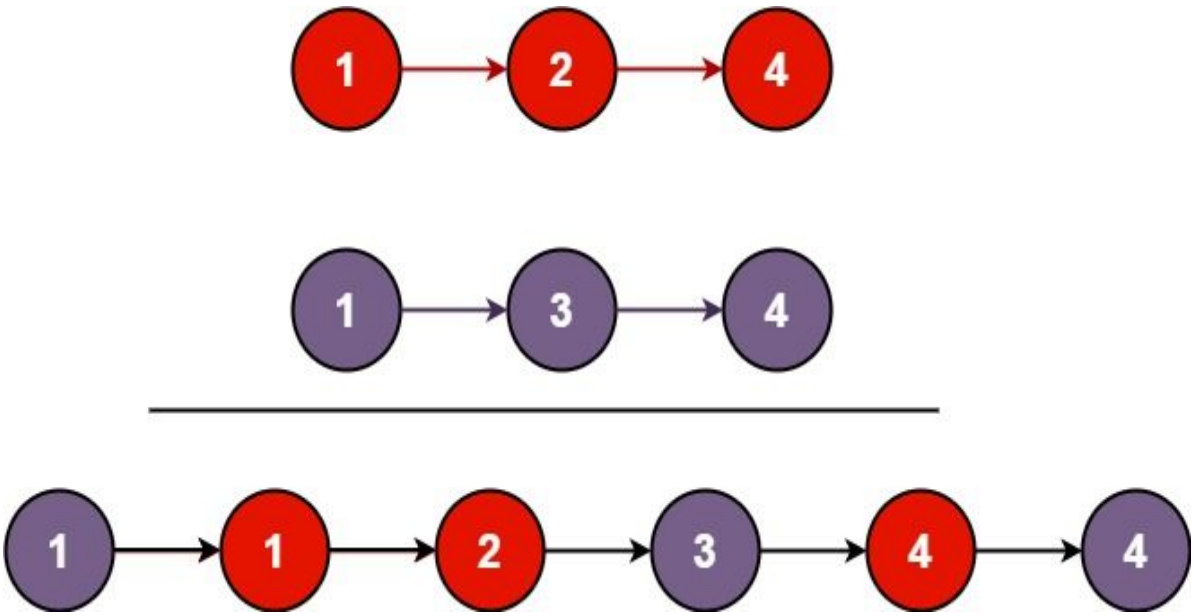
```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function(s) {
    if(s.length % 2 !==0) {
        return false;
    }
    const map = {'(':')', '{':'}', '[':']'}
    const res = [];
    res.push(s.charAt(0));
    for(let i = 1; i < s.length; i++) {
        if(s.charAt(i) === map[res[res.length - 1]]){
```

```
            res.pop();
        }else {
            res.push(s.charAt(i));
        }
    }
    return res.length === 0 ? true : false
};
```

0021 合并两个有序链表

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。

示例 1：




```
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

示例 2：

```
输入：l1 = [], l2 = []
输出：[]
```

示例 3：

```
输入：l1 = [], l2 = [0]
输出：[0]
```

提示：

- 两个链表的节点数目范围是 `[0, 50]`
- `-100 <= Node.val <= 100`
- `l1` 和 `l2` 均按 **非递减顺序** 排列

解法：迭代

比较大小，插入新链表

```
/**
 * 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}
 */
var mergeTwoLists = function(l1, l2) {
    let head = new ListNode();
    const mark = head;
    while(l1 || l2) {
        head.next = new ListNode();
        head = head.next;
        if(l1 && l2) {
            if(l1.val > l2.val) {
                head.val = l2.val;
                l2 = l2.next;
            }else {
                head.val = l1.val;
                l1 = l1.next;
              }

        }else if(l1) {
            head.val = l1.val;
            l1 = l1.next;
        }else if(l2) {
            head.val = l2.val;
            l2 = l2.next;
        }
    }
    return mark.next;
};
```

0022 括号生成

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例 1：

输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]

示例 2：

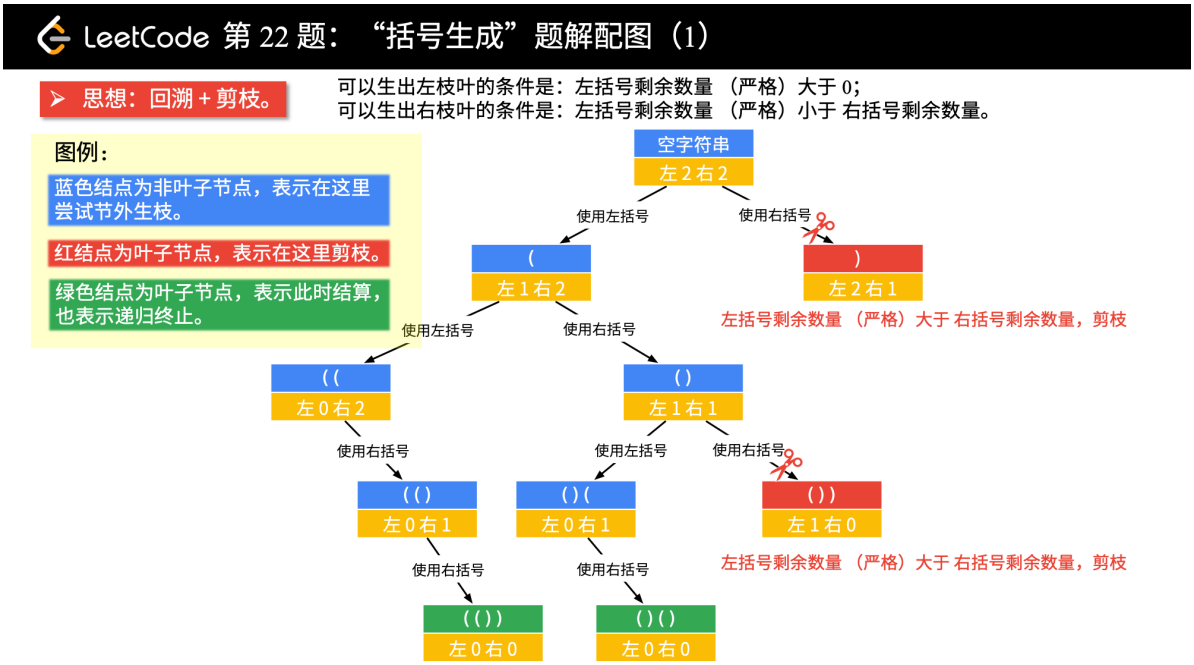
输入：n = 1
输出：["()"]

提示：

- `1 <= n <= 8`

解法：回溯法

深度优先遍历，利用系统栈




```javascript
/**
 * @param {number} n
 * @return {string[]}
 */
var generateParenthesis = function(n) {
    const res = [];
    function backtrace(str, L, R) {
        if(L === 0 && R === 0) res.push(str);
        if(L > 0) {
            backtrace(str + '(', L-1, R);
        }
        if(R > L) {
            backtrace(str + ')', L, R-1);
        }
    }
    backtrace('', n, n);
```

```
17      return res;
18  };
```

0023 合并K个升序链表

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

示例 2：

```
输入：lists = []
输出：[]
```

示例 3：

```
输入：lists = [[]]
输出：[]
```

提示：

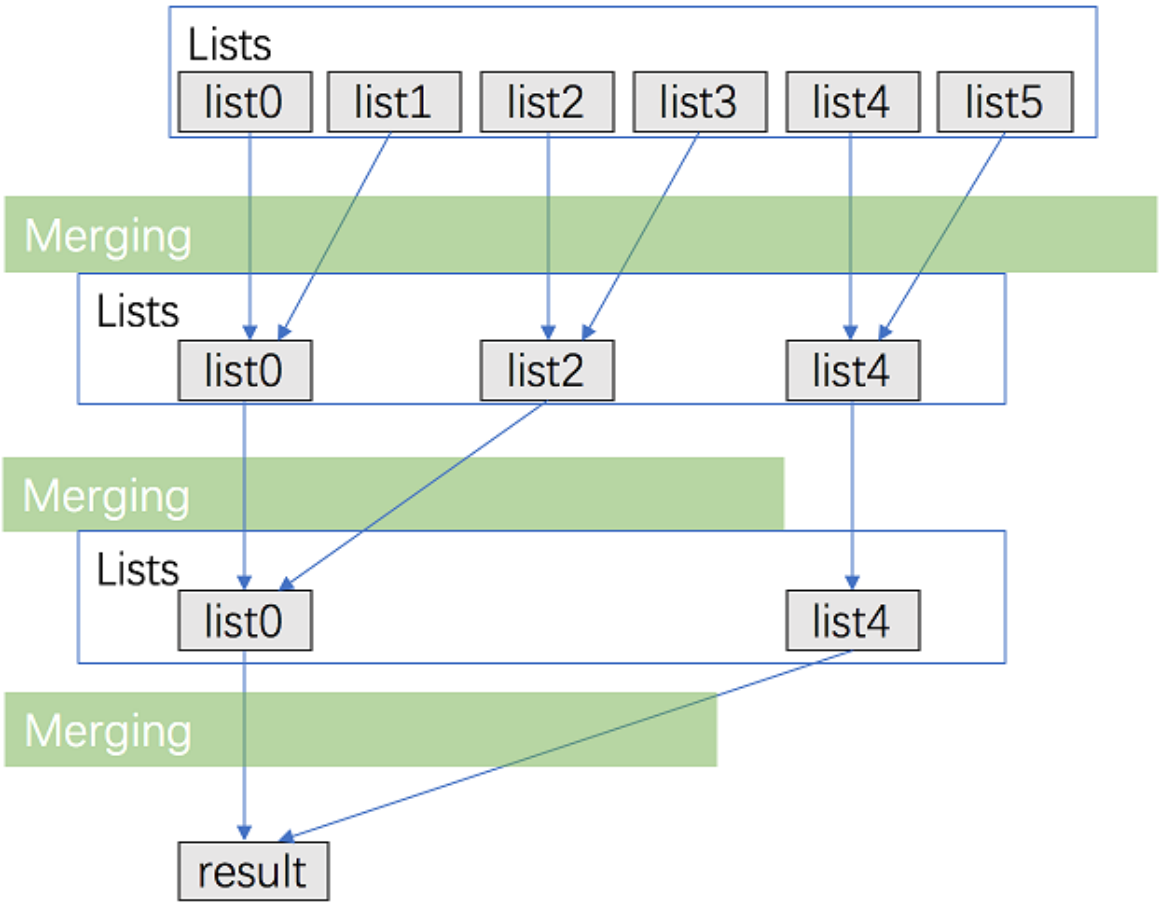
- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` 按 **升序** 排列
- `lists[i].length` 的总和不超过 `10^4`

解法1：顺序合并

通过一个for语句，对数组中的链表进行两两合并

```javascript
/**
 * 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}
 */
var mergeKLists = function(lists) {

    if(lists.length === 0) return null;
    if(lists.length === 1) return lists[0];

var mergeTwoLists = function(l1, l2) {
    const prehead = new ListNode(-1);

    let prev = prehead;
    while (l1 != null && l2 != null) {
        if (l1.val <= l2.val) {
            prev.next = l1;
            l1 = l1.next;
        } else {
            prev.next = l2;
            l2 = l2.next;
        }
        prev = prev.next;
    }

    // 合并后 l1 和 l2 最多只有一个还未被合并完，我们直接将链表末尾指向未合并完的链表即可
    prev.next = l1 === null ? l2 : l1;

    return prehead.next;
};

    for(let i = 0; i < lists.length - 1; i ++) {
        lists[i+1] = mergeTwoLists(lists[i], lists[i+1]);
    };
    return lists[lists.length - 1];
};
```

解法2：分治算法



分治算法对于递归调用的情况，可以将系统栈从n层缩减到logn层

0031 下一个排列

实现获取 **下一个排列** 的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须 **原地** 修改，只允许使用额外常数空间。

示例 1:

```
输入：nums = [1,2,3]
输出：[1,3,2]
```

示例 2:

```
输入：nums = [3,2,1]
输出：[1,2,3]
```

示例 3:

```
输入：nums = [1,1,5]
输出：[1,5,1]
```

示例 4：

```
输入：nums = [1]
输出：[1]
```

提示：

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 100`

解法：两遍扫描

那么什么是字典序法呢?

从上面的全排列也可以看出来了，从左往右依次增大，对这就是字典序法。可是如何用算法来实现字典序法全排列呢?

我们再来看一段文字描述：（用字典序法找`124653`的下一个排列）

- 如果当前排列是`124653`，找它的下一个排列的方法是，从这个序列中从右至左找第一个左邻小于右邻的数
- 如果找不到，则所有排列求解完成，如果找得到则说明排列未完成
- 本例中将找到`46`，计`4`所在的位置为`i`,找到后不能直接将`46`位置互换，而又要从右到左到第一个比`4`大的数
- 本例找到的数是`5`，其位置计为`j`，将`i`与`j`所在元素交换`125643`
- 然后将`i+1`至最后一个元素从小到大排序得到`125346`，这就是`124653`的下一个排列

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
 */
var nextPermutation = function(nums) {
    for(var i = nums.length - 1; i >0; i--) {
        if(nums[i-1] < nums[i]) {
            let j = i;
            let n = i;
            while(n < nums.length) {
                if(nums[n] > nums[i-1] && nums[n] <= nums[j]) {
                    j = n;
                }
                n++;
            }
            [nums[i-1],nums[j]] = [nums[j],nums[i-1]]//ES6解构赋值，可用于互换数据
            break;
        }else  {continue;}
    }
    let left = i, right = nums.length - 1;
    while(left < right) {
        [nums[left], nums[right]] = [nums[right],nums[left]];
        left++;
```

```
        right--;
    }
};
```

0032 最长有效括号

给你一个只包含 `'('` 和 `')'` 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

```
输入：s = "(()"
输出：2
解释：最长有效括号子串是 "()"
```

示例 2：

```
输入：s = ")()())"
输出：4
解释：最长有效括号子串是 "()()"
```

示例 3：

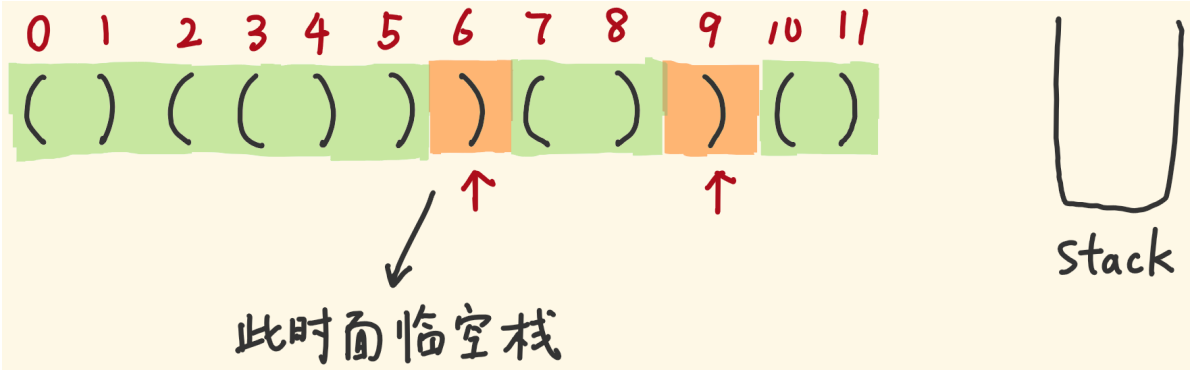
```
输入：s = ""
输出：0
```

提示：

- `0 <= s.length <= 3 * 10^4`
- `s[i]` 为 `'('` 或 `')'`

解法1：栈

- 在栈中预置 -1 作为一个“参照物”，并改变计算方式：当前索引 - 出栈后新的栈顶索引。



- 当遍历到索引 5 的右括号，此时栈顶为 2，出栈，栈顶变为 -1，有效长度为 5 - (-1) = 6。如果像之前那样，5 找不到 -1 减。
- 当遍历到索引 6 的右括号，它不是需要入栈的左括号，又匹配不到左括号，好似废物，怎么利用它呢?
- 它后面可能也出现这么一段有效长度，它要成为 -1 那样的“参照物”。它之前出现的有效长度都求过了，-1 的使命已经完成了，要被替代。
- 所以我们照常让 -1 出栈。不同的是，此时栈空了，让索引 6 入栈当 “参照物”。

总结：两种索引会入栈

1. 等待被匹配的左括号索引。
2. 充当「参照物」的右括号索引。因为：当左括号匹配光时，栈需要留一个垫底的参照物，用于计算一段连续的有效长度。

```
/**
 * @param {string} s
 * @return {number}
 */
var longestValidParentheses = function(s) {
    let maxLen = 0;
    const stack = [];
    stack.push(-1);
    for(let i = 0; i < s.length; i++) {
        const char = s[i];
        if(char === '(') {
            stack.push(i);
        }else {
            stack.pop();
            if(stack.length) {
                maxLen = Math.max(maxLen, i-stack[stack.length - 1]);
            }else {
                stack.push(i);
            }
        }
    }
    return maxLen;
};
```

0033 搜索旋转排列的数组

整数数组 `nums` 按升序排列，数组中的值 **互不相同** 。

在传递给函数之前，`nums` 在预先未知的某个下标 `k`（`0 <= k < nums.length`）上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标 **从 0 开始** 计数）。例如，`[0,1,2,4,5,6,7]` 在下标 `3` 处经旋转后可能变为 `[4,5,6,7,0,1,2]`。

给你 **旋转后** 的数组 `nums` 和一个整数 `target`，如果 `nums` 中存在这个目标值 `target`，则返回它的下标，否则返回 `-1`。

示例 1：

```
输入：nums = [4,5,6,7,0,1,2], target = 0
输出：4
```

示例 2：

```
输入：nums = [4,5,6,7,0,1,2], target = 3
输出：-1
```

示例 3：

```
输入：nums = [1], target = 0
输出：-1
```

提示：

- `1 <= nums.length <= 5000`
- `-10^4 <= nums[i] <= 10^4`
- `nums` 中的每个值都 **独一无二**
- 题目数据保证 `nums` 在预先未知的某个下标上进行了旋转
- `-10^4 <= target <= 10^4`

进阶：你可以设计一个时间复杂度为 `O(log n)` 的解决方案吗？

解法：二分法

此题不能直接用二分，因此思路是先二分，判断左右两边哪边为顺序的，将target值放进去比较，舍弃另一边，如果target不在顺序的一边，则进入非顺序一边，继续判断，直到target在顺序的一边，**注意等号的判断**

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

```
var search = function(nums, target) {
    let left = 0, right = nums.length - 1;
    let mid;
    while (left <= right) {
        mid = (left + right) >> 1;
        if(target === nums[mid]) return mid;
        if(nums[mid] >= nums[left]) {
            if(target >= nums[left] && target <= nums[mid]) {
                right = mid - 1;
            }else {
                left = mid + 1;
            }
        }else {
            if(target >= nums[mid] && target <= nums[right]) {
                left = mid + 1;
            }else {
                right = mid - 1;
            }
        }
    }
    return -1;
};
```

此题还可以先通过二分法找到最小位置，在进行二分查找

0034 在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

进阶：

- 你可以设计并实现时间复杂度为 `O(log n)` 的算法解决此问题吗？

示例 1：

```
输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]
```

示例 2：

```
输入：nums = [5,7,7,8,8,10], target = 6
输出：[-1,-1]
```

示例 3：

```
输入：nums = [], target = 0
输出：[-1,-1]
```

提示：

- `0 <= nums.length <= 10^5`
- `-10^9 <= nums[i] <= 10^9`
- `nums` 是一个非递减数组
- `-10^9 <= target <= 10^9`

解法1：二分查找 + 双指针

先通过二分法随便找到一个目标 target，之后通过while循环，让左右指针从中间开始向两边扩展，直到遇到不是target的值

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var searchRange = function(nums, target) {
    if(nums.length === 0) return [-1,-1];
    var L = 0, R = nums.length-1, mid = 0;
    while(L <= R) {
        mid = (L + R)>>1;
        if(nums[mid] > target) R = mid - 1;
        else if (nums[mid] < target) L = mid + 1;
        else {
            L=mid;
            R=mid;
            while(nums[L] === target || nums[R] === target) {
                if(nums[L] === target) L--;
                if(nums[R] === target) R++;
            }
            return [L+1,R-1];
        }
    }
    return [-1,-1];
};
```

解法2：二分查找

通过二分法寻找数组中第一个大于等于 target 的值，和第一个大于 target 的值

```
const binarySearch = (nums, target, lower) => {
    let left = 0, right = nums.length - 1, ans = nums.length;
    while (left <= right) {
        const mid = Math.floor((left + right) / 2);
        if (nums[mid] > target || (lower && nums[mid] >= target)) {
            right = mid - 1;
            ans = mid;
        } else {
            left = mid + 1;
```

```
        }
    }
    return ans;
}

var searchRange = function(nums, target) {
    let ans = [-1, -1];
    const leftIdx = binarySearch(nums, target, true);
    const rightIdx = binarySearch(nums, target, false) - 1;
    if (leftIdx <= rightIdx && rightIdx < nums.length && nums[leftIdx] === target && nums[rightIdx] === target) {
        ans = [leftIdx, rightIdx];
    }
    return ans;
};
```

0039 数组总和

给定一个**无重复元素**的数组 `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` 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 `target`）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,
所求解集为：
[
  [7],
  [2,2,3]
]
```

示例 2：

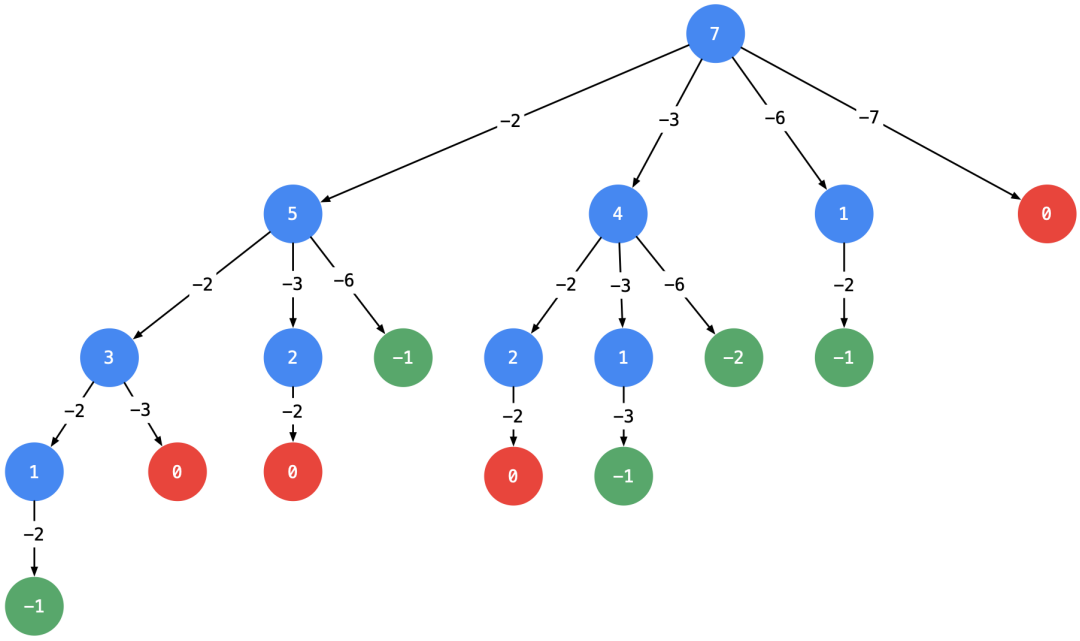
```
输入：candidates = [2,3,5], target = 8,
所求解集为：
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

提示：

- `1 <= candidates.length <= 30`
- `1 <= candidates[i] <= 200`
- `candidate` 中的每个元素都是独一无二的。
- `1 <= target <= 500`

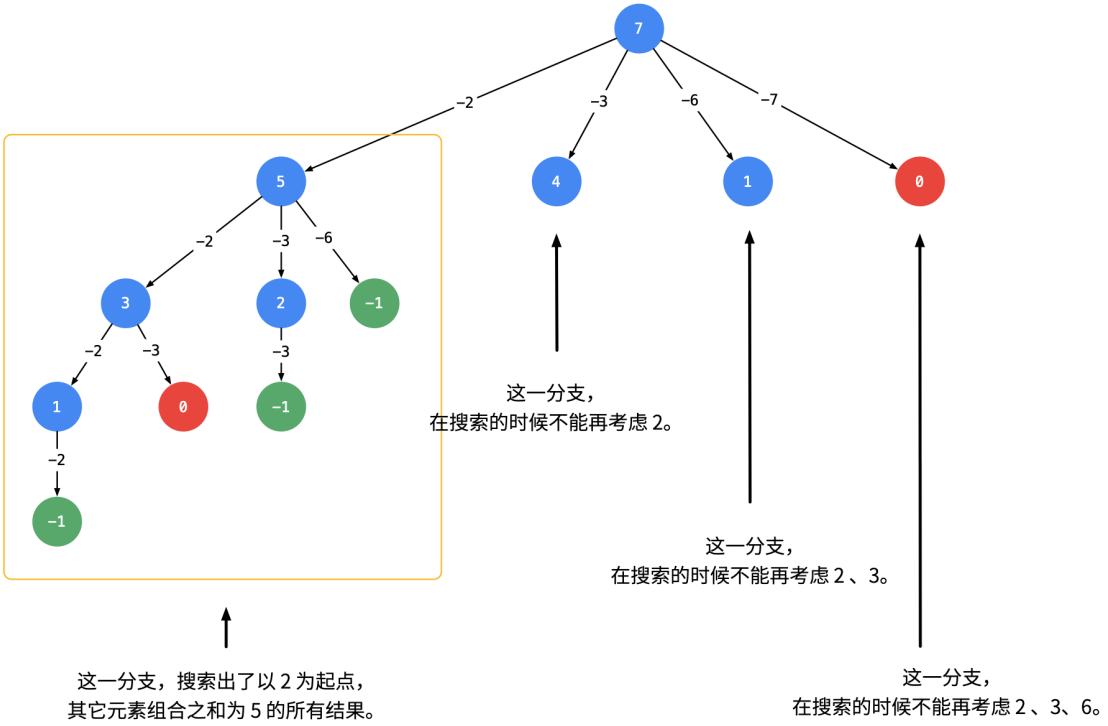
解法：回溯 + 剪枝

以输入：`candidates = [2, 3, 6, 7]`，`target = 7` 为例：



但是此种方法的结果会输出 `[2,2,3],[2,3,2],[3,2,2],[7]` 从而导致重复

而重复的原因是元素的重复使用，因此需要进行优化。遇到这一类相同元素不计算顺序的问题，我们在搜索的时候就需要 **按某种顺序搜索**。具体的做法是：每一次搜索的时候设置 **下一轮搜索的起点** `begin`



```
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function(candidates, target) {
    if (candidates.length === 0) return [];
    const res = [];
    const dfs = (target, ans, index) => {
        if(target < 0) return;
        if(target === 0) {
            res.push(ans);
            return;
        }
        for(let i = index; i < candidates.length; i++) {
            dfs(target - candidates[i], [...ans, candidates[i]], i);
        }
    }
    dfs(target, [], 0);
    return res;
};
```

此处的 `[...ans, candidates[i]]` 为扩展运算符，用于合并两个数组；此处不能把数组名传进去，因为每次都需要开辟新的独立空间，因此直接传入合并后的数组

可以不用 for 循环

```
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function(candidates, target) {
    const ans = [];
    candidates = candidates.sort((a,b)=>a-b);
    const dfs = (target, combine, idx) => {
        if (idx === candidates.length) {
            return;
        }
        if (target === 0) {
            ans.push(combine);
            return;
        }
        // 直接跳过，让其index++，之后再判断是否选择当前数
        dfs(target, combine, idx + 1);
        // 选择当前数
        if (target - candidates[idx] >= 0) {
            dfs(target - candidates[idx], [...combine, candidates[idx]],
idx);
        }
    }
    dfs(target, [], 0);
    return ans;
};
```

0042 接雨水

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：

输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]
输出：6
解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]
输出：9

提示：

- `n == height.length`
- `0 <= n <= 3 * 10^4`
- `0 <= height[i] <= 10^5`

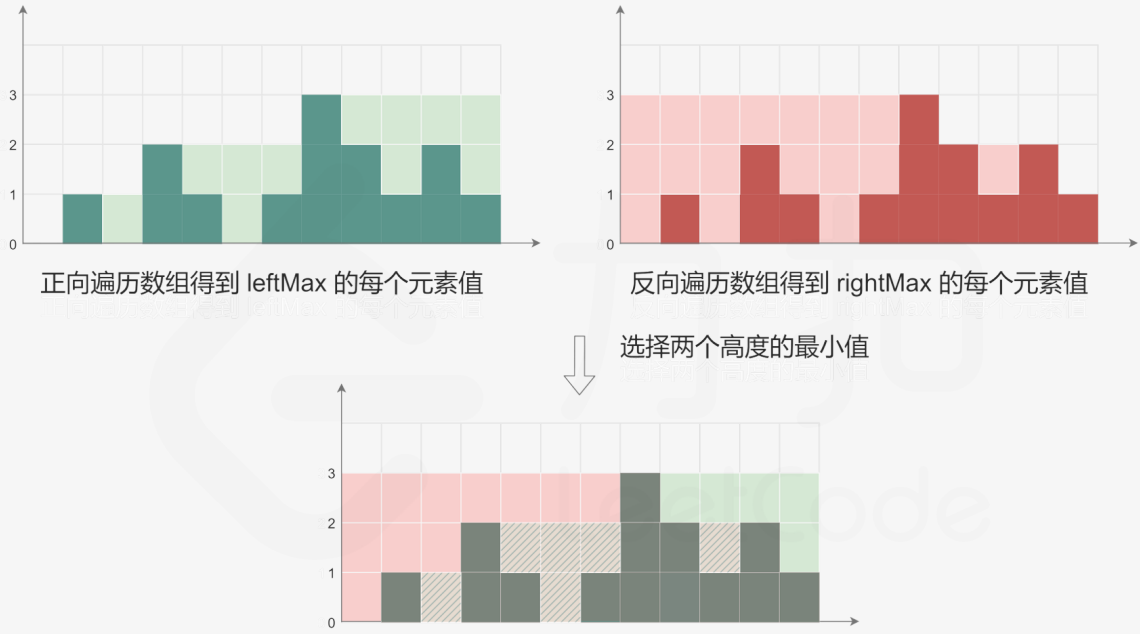
解法1：暴力法

直接按问题描述进行。对于数组中的每个元素，我们找出**下雨后水能达到的最高位置**，等于两边最大高度的较小值减去当前高度的值

```
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function(height) {
    let ans = 0;
    let size = height.length;
    for (let i = 1; i < size - 1; i++) {
        let left_max = 0, right_max = 0;
        for(let j = i; j >= 0; j--) {
            left_max = Math.max(left_max, height[j]);
        }
        for(let j = i; j < size; j++) {
```

```
            right_max = Math.max(right_max, height[j]);
        }
        ans += Math.min(left_max, right_max) - height[i];
    }
    return ans;
};
```

解法2：动态规划




```
var trap = function(height) {
    let ans = 0;
    const n = height.length;
    if(n === 0) return 0;

    const leftMax = new Array(n).fill(0);
    leftMax[0] = height[0];
    for(let i = 1; i < n ;i ++) {
        leftMax[i] = Math.max(leftMax[i-1], height[i]);
    }

    const rightMax = new Array(n).fill(0);
    rightMax[n-1] = height[n-1];
    for(let i = n - 2; i >= 0; i--) {
        rightMax[i] = Math.max(rightMax[i + 1], height[i]);
    }

    for (let i = 0; i < n; i++) {
        ans += Math.min(leftMax[i], rightMax[i]) - height[i];
    }
    return ans;
};
```

解法3：双指针

动态规划中使用两次遍历才获取左右两边最大高度，能否通过一次遍历就获取呢？思路就是左右向中间靠拢求取各自最大值的同时，取二者中较小的

把动态规划中的空间复杂度降到 O(1)

- 使用 $height[left]$ 和 $height[right]$ 的值更新 $leftMax$ 和 $rightMax$ 的值；
- 如果 $height[left] < height[right]$，则必有 $leftMax < rightMax$，下标 $left$ 处能接的雨水量等于 $leftMax - height[left]$，将下标 $left$ 处能接的雨水量加到能接的雨水总量，然后将 $left$ 加 1（即向右移动一位）；
- 如果 $height[left] \geq height[right]$，则必有 $leftMax \geq rightMax$，下标 $right$ 处能接的雨水量等于 $rightMax - height[right]$，将下标 $right$ 处能接的雨水量加到能接的雨水总量，然后将 $right$ 减 1（即向左移动一位）。

```
var trap = function(height) {
    let ans = 0;
    let left = 0, right = height.length - 1;
    let leftMax = 0, rightMax = 0;
    while (left < right) {
        leftMax = Math.max(leftMax, height[left]);
        rightMax = Math.max(rightMax, height[right]);
        if (height[left] < height[right]) {
            ans += leftMax - height[left];
            ++left;
        } else {
            ans += rightMax - height[right];
            --right;
        }
    }
    return ans;
};
```

0046 全排列

给定一个 **没有重复** 数字的序列，返回其所有可能的全排列。

示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

解法：回溯法

每一个层的节点为当前剩余长度，在每一层使用 for 循环调用 dfs 函数，splice函数删除数组中相应位置的元素，进入下一层后剩余长度-1，直到为0返回

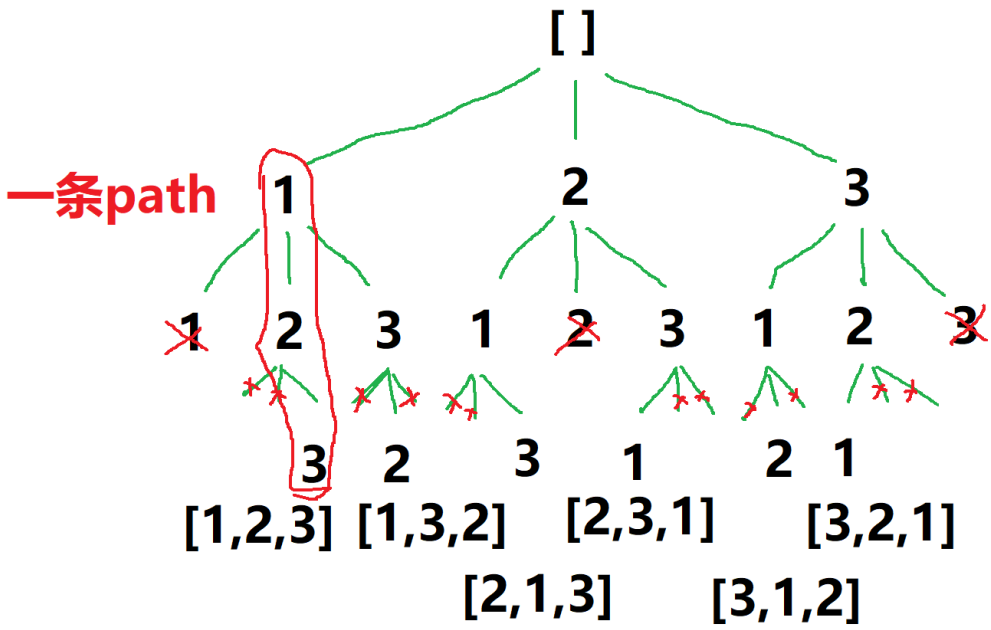
传参的时候注意开辟**新空间**

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permute = function(nums) {
    const res = [];
    let length = nums.length;
    const dfs = (left_nums, ans, left_length )=>{
        if(left_length === 0) {res.push(ans); return;}
        for (let i = 0; i < left_length; i++) {
            let left = [...left_nums];
            left.splice(i,1);
            dfs(left, [...ans, [...left_nums][i]], (left_length-1));
        }
    }
    dfs(nums, [], length);
    return res;
};
```

上述算法浪费了太多内存空间，但是没有对每种情况都进行剪枝

下面的算法只开辟了 res 和 used 额外空间，模拟每种可能的情况，并进行剪枝

used用于记录已经使用的元素，及时对 path 进行入栈和出栈操作，节省空间




```
const permute = (nums) => {
  const res = [];
  const used = {};

  function dfs(path) {
    if (path.length == nums.length) { // 个数选够了
      res.push(path.slice()); // 拷贝一份path，加入解集res
```

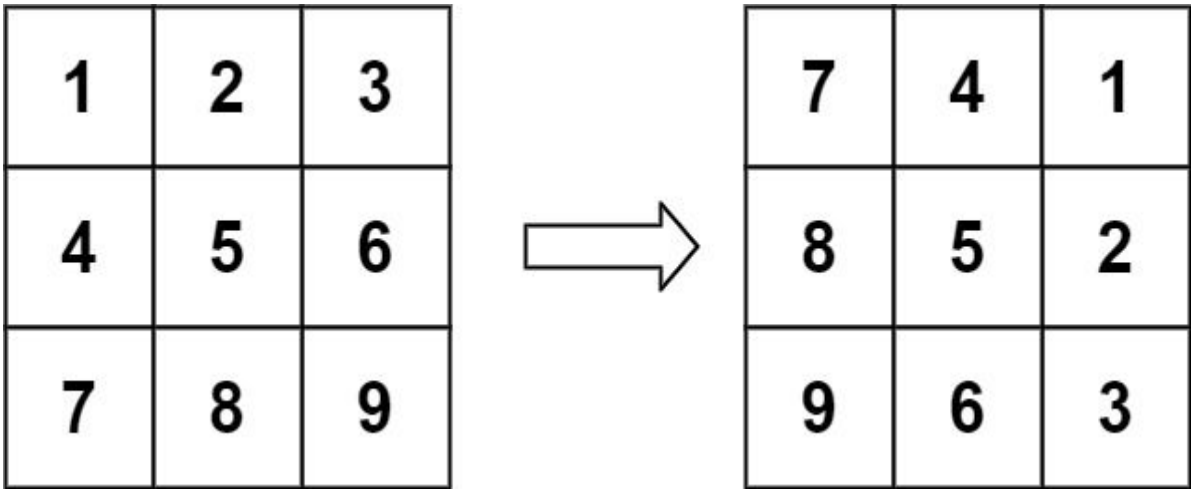
```
      return;                    // 结束当前递归分支
    }
    for (const num of nums) { // for枚举出每个可选的选项
      // if (path.includes(num)) continue; // 别这么写！查找的时间是O(n)，增加时间复杂度
      if (used[num]) continue; // 使用过的，跳过
      path.push(num);          // 选择当前的数，加入path
      used[num] = true;        // 记录一下 使用了
      dfs(path);               // 基于选了当前的数，递归
      path.pop();              // 上一句的递归结束，回溯，将最后选的数pop出来
      used[num] = false;       // 撤销这个记录
    }
  }

  dfs([]); // 递归的入口，空path传进去
  return res;
};
```

0048 旋转图像

给定一个 *n* × *n* 的二维矩阵 `matrix` 表示一个图像。请你将图像顺时针旋转 90 度。

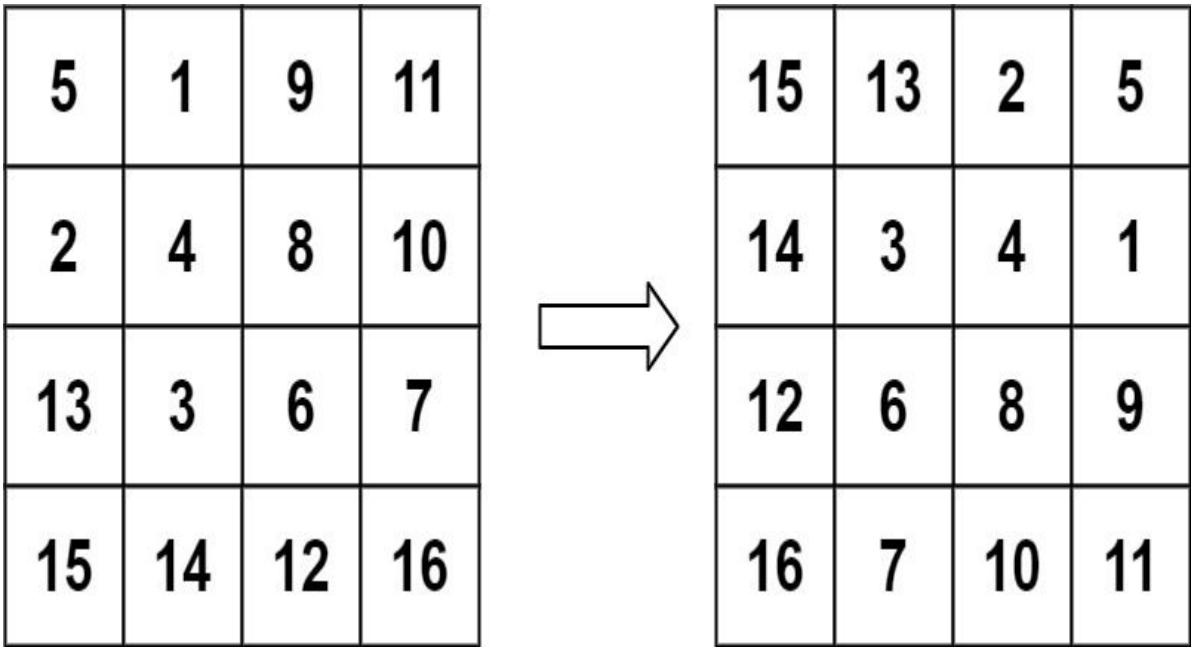
你必须在 **原地** 旋转图像，这意味着你需要直接修改输入的二维矩阵。**请不要** 使用另一个矩阵来旋转图像。

示例 1：



输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[[7,4,1],[8,5,2],[9,6,3]]

示例 2：



输入：matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
输出：[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

示例 3：

输入：matrix = [[1]]
输出：[[1]]

示例 4：

输入：matrix = [[1,2],[3,4]]
输出：[[3,1],[4,2]]

提示：

- `matrix.length == n`
- `matrix[i].length == n`
- `1 <= n <= 20`
- `-1000 <= matrix[i][j] <= 1000`

解法1：翻转数组

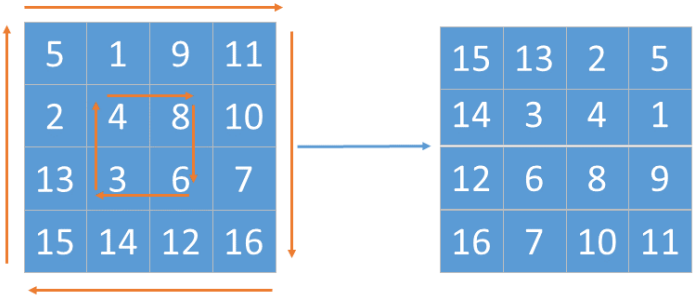
先进行水平翻转，再进行主对角线翻转

```
var rotate = function(matrix) {
    const n = matrix.length;
    for(let i = 0; i < Math.floor(n/2); i++) {
        for(let j = 0; j < n; j++) {
            [matrix[i][j], matrix[n-i-1][j]]=[matrix[n-i-1][j], matrix[i][j]];
        }
```

```
    }

    for(let i = 0; i < n; i++) {
        for(let j = 0; j < i; j++) {
            [matrix[i][j], matrix[j][i]] = [matrix[j][i], matrix[i][j]];
        }
    }
};
```

解法2：从外向内旋转

作者：数据结构和算法




```
var rotate = function(matrix) {
    const length = matrix.length;
    for(let i = 0; i < Math.floor(length/2); i++) {
        for (let j = i; j < length-i-1; j++) {
            let temp = matrix[i][j];
            let m = length - j - 1;
            let n = length - i - 1;
            matrix[i][j] = matrix[m][i];
            matrix[m][i] = matrix[n][m];
            matrix[n][m] = matrix[j][n];
            matrix[j][n] = temp;
        }
    }
};
```

0049 字母异位词分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

输入：`["eat", "tea", "tan", "ate", "nat", "bat"]`
输出：

```
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明：

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

解法：Hash表 + 排序

先对获取到的字符串转换成数组，按升序排序，再转换为字符串，存入 Map 中，之后若有修改，再进行 set 操作

```
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
var groupAnagrams = function(strs) {
    const map = new Map();
    for(let str of strs) {
        const arr = Array.from(str);
        arr.sort();
        let s = arr.toString();
        let list = map.get(s) ? map.get(s) : new Array();
        list.push(str);
        map.set(s, list);
    }
    return Array.from(map.values())
};
```

0053 最大字序和

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。

示例 1：

输入：`nums = [-2,1,-3,4,-1,2,1,-5,4]`
输出：`6`
解释：连续子数组 `[4,-1,2,1]` 的和最大，为 `6` 。

示例 2：

输入：nums = [1]
输出：1

示例 3：

输入：nums = [0]
输出：0

示例 4：

输入：nums = [-1]
输出：-1

示例 5：

输入：nums = [-100000]
输出：-100000

提示：

- `1 <= nums.length <= 3 * 10^4`
- `-10^5 <= nums[i] <= 10^5`

解法1：贪心算法

若当前指针所指元素之前的和小于0，则舍弃当前元素之前的数列

```
var maxSubArray = function(nums) {
    let pre = 0, max = nums[0];
    nums.forEach((current)=>{
        if(pre < 0 ) {
            pre = current;
        }else {
            pre = pre + current;
        }
        max = Math.max(max, pre);
    })
    return max;
};
```

解法2：动态规划

若前一个元素大于0，则将其加到当前元素上

动态规划的核心是：为了找到不同子序列之间的递推关系，**以子序列的结束点为基准的**，这点开阔了我们的思路。比如：以 `b` 为结束点的所有子序列: `[a , b]` `[b]` 以 `c` 为结束点的所有子序列: `[a, b, c]` `[b, c]` `[ c ]`。

参考题解：https://leetcode-cn.com/problems/maximum-subarray/solution/dong-tai-gui-hua-fen-zhi-fa-python-dai-ma-java-dai/ **详细解析了动态规划的过程以及关键点无后效性**

```
var maxSubArray = function(nums) {
    const n = nums.length;
    let max = nums[0];
    for(let i = 1; i < n; i++) {
        if(nums[i - 1] > 0) {
            nums[i] += nums[i - 1];
        }
        max = Math.max(max, nums[i]);
    }
    return max;
};
```

0055 跳跃游戏

给定一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。

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下标。

示例 1：

```
输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。
```

示例 2：

```
输入：nums = [3,2,1,0,4]
输出：false
解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。
```

提示：

- `1 <= nums.length <= 3 * 10^4`
- `0 <= nums[i] <= 10^5`

解法1：贪心算法

实时维护最远可以到达的位置

```
var canJump = function(nums) {
    const n = nums.length;
    let rightmost = 0;
    for(let i = 0; i < n; i++) {
        if (i <= rightmost) {
            rightmost = Math.max(rightmost, i + nums[i]);
            if (rightmost >= n - 1) {
                return true;
            }
        }
    }
    return false;
};
```

解法2：动态规划

实时更新当前位置最多能走几步

```
var canJump = function(nums) {
    const n = nums.length;
    if(n === 1) return true;
    for(let i = 0; i < n - 1; i++) {
        if(i >= 1){
            nums[i] = nums[i] >= nums[i-1] - 1 ? nums[i] : nums[i-1] - 1;
        }
        if(nums[i] >= n - 1 - i ) return true;
        if(nums[i] === 0) return false;
    }
    return false;
};
```

0056 合并区间

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [start_i, end_i]` 。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1：

```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2：

```
输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

提示：

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= start_i <= end_i <= 10^4`

解法：排序

利用 sort 函数进行排序，之后判断是否处于同一区间

```
var merge = function (intervals) {
  let res = [];
  intervals.sort((a, b) => a[0] - b[0]);

  let prev = intervals[0];

  for (let i = 1; i < intervals.length; i++) {
    let cur = intervals[i];
    if (prev[1] >= cur[0]) { // 有重合
      prev[1] = Math.max(cur[1], prev[1]);
    } else {       // 不重合，prev推入res数组
      res.push(prev);
      prev = cur;  // 更新 prev
    }
  }

  res.push(prev);
  return res;
};
```

0062 不同路径

一个机器人位于一个 `m x n` 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径？

示例 1：

输入：m = 3, n = 7
输出：28

示例 2：

输入：m = 3, n = 2
输出：3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

示例 3：

输入：m = 7, n = 3
输出：28

示例 4：

输入：m = 3, n = 3
输出：6

提示：

- `1 <= m, n <= 100`
- 题目数据保证答案小于等于 `2 * 10^9`

解法：动态规划

$$f(i,j) = f(i-1,j) + f(i,j-1)$$

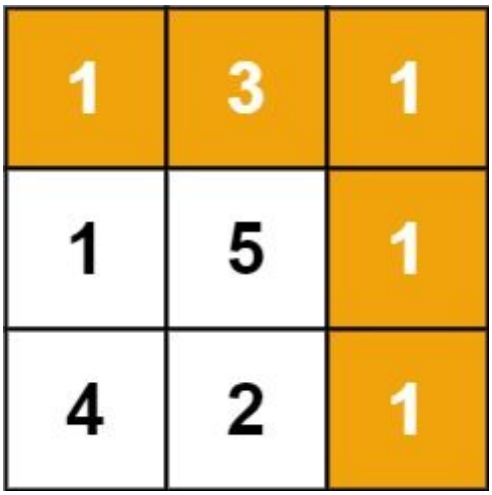
```
var uniquePaths = function(m, n) {
    const dp = new Array(m).fill(0).map(() => new Array(n).fill(0));
    for(let i = 0; i < m; i++) {
        for(let j = 0; j < n; j++) {
            if(i === 0 || j === 0) dp[i][j] = 1;
            else {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            }
        }
    }

    return dp[m-1][n-1];
};
```

0064 最小路径和

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。

示例 1：



输入：`grid = [[1,3,1],[1,5,1],[4,2,1]]`
输出：`7`
解释：因为路径 `1→3→1→1→1` 的总和最小。

示例 2：

输入：`grid = [[1,2,3],[4,5,6]]`
输出：`12`

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 200`
- `0 <= grid[i][j] <= 100`

解法：动态规划

```
var minPathSum = function(grid) {
    for(let i = 0; i < grid.length; i++) {
        for(let j = 0; j < grid[i].length; j++) {
            if(i === 0 && j === 0) continue;
            else if(i === 0) grid[i][j] += grid[i][j-1];
            else if(j === 0) grid[i][j] += grid[i-1][j];
            else grid[i][j] += Math.min(grid[i-1][j],grid[i][j-1]);
        }
    }
    return grid.pop().pop();
};
```

0070 爬楼梯

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。

示例 1:

```
输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2:

```
输入： 3
输出： 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
```

解法：动态规划

$$f(i) = f(i-1) + f(i-2)$$

```
var climbStairs = function(n) {
    const dp = new Array(n).fill(0);
    for(let i = 0; i < dp.length; i++) {
        if(i === 0) dp[i] = 1;
        else if(i === 1) dp[i] = 2;
        else {
            dp[i] = dp[i-1] + dp[i-2];
        }
    }
    return dp.pop();
};
```

0072 编辑距离

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

示例 1：

```
输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2：

```
输入：word1 = "intention", word2 = "execution"
输出：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：

- `0 <= word1.length, word2.length <= 500`
- `word1` 和 `word2` 由小写英文字母组成

解法：动态规划

	‘ ’	r	o	s
‘ ’	0	1	2	3
h	1			
o	2			
r	3			
s	4			
e	5			

```
var minDistance = function(word1, word2) {
    const m = word1.length;
    const n = word2.length;
    const dp = new Array(m+1).fill(0).map(() => new Array(n+1).fill(0));

    for(let i = 0; i <= m; i++) dp[i][0] = i;
    for(let j = 0; j <= n; j++) dp[0][j] = j;

    for(let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            if(word1.charAt(i-1) === word2.charAt(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;
        }
    }
    return dp[m][n];
};
```

动态规划的关键是找到状态转移方程

此题中，可以观察到，对于`word1`我们可以进行三种操作：插入、删除、修改

- **增** `dp[i][j] = dp[i][j - 1] + 1`
- **删** `dp[i][j] = dp[i - 1][j] + 1`
- **改** `dp[i][j] = dp[i - 1][j - 1] + 1`

则状态转移方程可以表示为`dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1`

有一种特殊情况，`if(word1.charAt(i-1) === word2.charAt(j-1)) dp[i][j] = dp[i-1][j-1]`

0075 颜色分类

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。

示例 1：

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

示例 2：

```
输入：nums = [2,0,1]
输出：[0,1,2]
```

示例 3：

```
输入：nums = [0]
输出：[0]
```

示例 4：

```
输入：nums = [1]
输出：[1]
```

提示：

- `n == nums.length`
- `1 <= n <= 300`
- `nums[i]` 为 `0`、`1` 或 `2`

进阶：

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

解法：双指针

左右指针从左右两端向中间移动，`nums[i]=2` 移到右边`nums[i]=0` 移到左边

```
var sortColors = function(nums) {
    const n = nums.length;
    let p0 = 0, p2 = n - 1;
    for(let i = 0; i <= p2; i++) {
        while(nums[i] === 2 && i <= p2) {
            [nums[i], nums[p2]] = [nums[p2], nums[i]];
            p2--;
        }
        if(nums[i] === 0) {
```

```
            [nums[i], nums[p0]] = [nums[p0], nums[i]];
            p0++;
        }
    }
};
```

0076 最小覆盖子串

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 `""` 。

注意：如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

提示：

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

解法：滑动窗口

- 利用双指针维护一个滑动窗口，不断移动右指针
- 判断右指针的字符是否与字典表中的匹配
  - 是：将字典表中的次数 - 1，直到为 0（这里有个技巧：创建一个变量（needType）记录需要匹配的字符数量，初始长度为 Map 的 size,当对应的字符次数为 0 时，就减 1）
  - 否：继续移动右指针
- 当needType的值为 0 时，就证明在当前窗口所有字符都匹配成功了
  - 移动左指针，缩小滑动窗口的大小
  - 移动过程中判断左指针指向的值是否为字典中值，如果是就证明匹配的值少了一个，这是需要更新 Map 中的次数，以及needType的数量
  - 记录每次窗口中的字符，找到最小的返回

```
var minWindow = function(s, t) {
    let l = 0, r = 0;
    let res = '';
```

```
    const map = new Map();

    for(let char of t) {
        map.set(char, map.has(char) ? map.get(char) + 1 : 1);
    }

    let needType = map.size;

    while(r < s.length) {
        const char = s.charAt(r);
        if(map.has(char)) {
            map.set(char, map.get(char) - 1);
            if(map.get(char) === 0) needType -= 1;
        }

        while(needType === 0) {
            const c = s[l];
            let current = s.slice(l, r + 1);
            if(!res || current.length < res.length) res = current;
            if(map.has(c)) {
                map.set(c, map.get(c) + 1);
                if(map.get(c) === 1) needType += 1;
            }
            l++;
        }
        r++;
    }

    return res;
};
```

0078 子集

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1：

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

示例 2：

```
输入：nums = [0]
输出：[[],[0]]
```

提示：

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有元素 **互不相同**

解法：回溯法

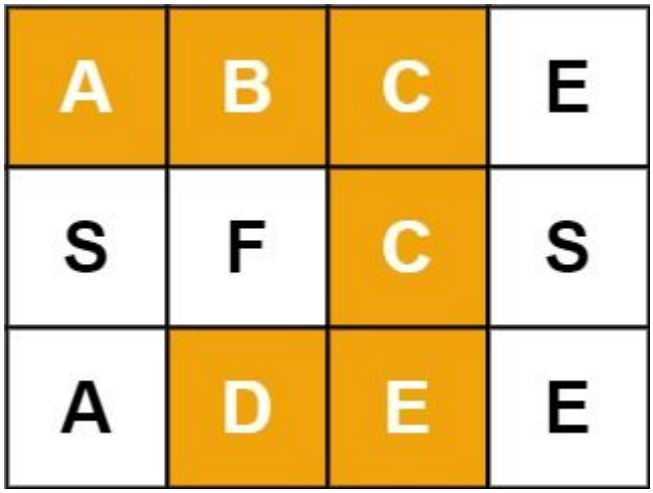
```
var subsets = function(nums) {
    const res = [[]];
    const n = nums.length
    const dfs = (idx, current) => {
        for(let i = idx; i < n; i++) {
            const c = [...current, nums[i]]
            res.push(c);
            if(i === n - 1) return;
            else dfs(i+1, c);
        }
    }
    dfs(0,[]);
    return res;
};
```

0079 单词搜索

给定一个 `m x n` 二维字符网格 `board` 和一个字符串单词 `word` 。如果 `word` 存在于网格中，返回 `true` ；否则，返回 `false` 。

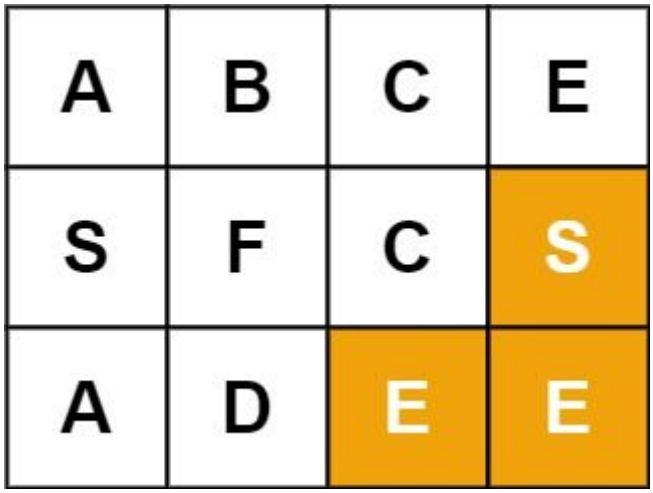
单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1：



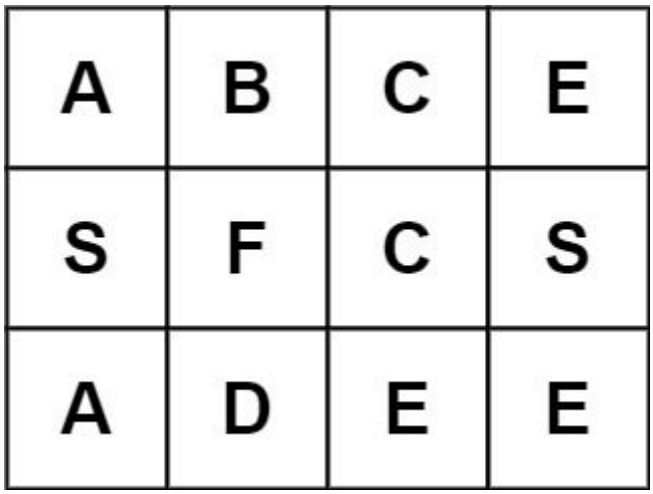
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
输出：true

示例 2：



输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
输出：true

示例 3：



输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
输出：false

提示：

- `m == board.length`
- `n = board[i].length`
- `1 <= m, n <= 6`
- `1 <= word.length <= 15`
- `board` 和 `word` 仅由大小写英文字母组成

进阶：你可以使用搜索剪枝的技术来优化解决方案，使其在 `board` 更大的情况下可以更快解决问题？

解法：回溯法

深度优先搜索

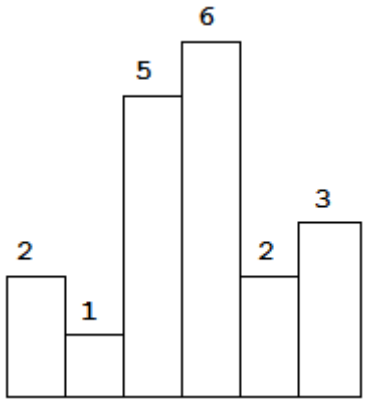
- 首先对 board 进行遍历，如果首字母相同，则开始 dfs
- 创建一个数组 status 用于记录当前位置的元素是否被遍历过，若未处理过，则本次分支置为true，**注意：在每次dfs结束后要将status置为false，因为是深度优先搜索**
- 对于每一次dfs，先判断当前元素是否被遍历过，之后判断当前元素是否相同
  - 若达到word结尾，则将flag置为true
  - 否则，想上下左右四个方向进行搜索

```
var exist = function(board, word) {
    const status = new Array(board.length).fill(false).map(() => new Array(board[0].length).fill(false));
    const directions = [[0,-1], [0, 1], [-1, 0], [1, 0]];
    let flag = false;

    const dfs = (x, y, current) => {
        if(status[x][y] === true) return;
        status[x][y] = true;
        if(word.charAt(current) === board[x][y]) {
            if(current === word.length - 1) {
                flag = true;
                return;
            }else {
                for(let direct of directions) {
                    if(x + direct[0] >= 0 && y + direct[1] >= 0 && x + direct[0] < board.length && y + direct[1] < board[0].length) {
                        dfs(x + direct[0], y + direct[1], current + 1);
                    }
                }
            }
        }
        status[x][y] = false;
        return;
    }

    for(let i = 0; i < board.length; i++) {
        for(let j = 0; j < board[0].length; j++) {
            if(flag === true) break;
            if(board[i][j] === word.charAt(0)) dfs(i, j, 0);
        }
    }

    return flag;
};
```

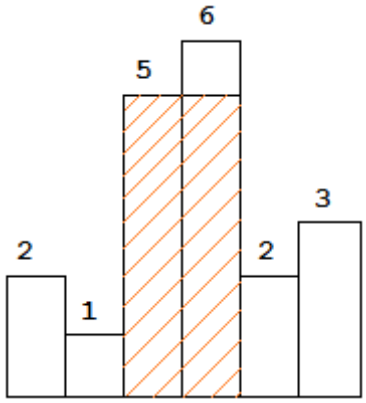
0084 柱状图中最大的矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。

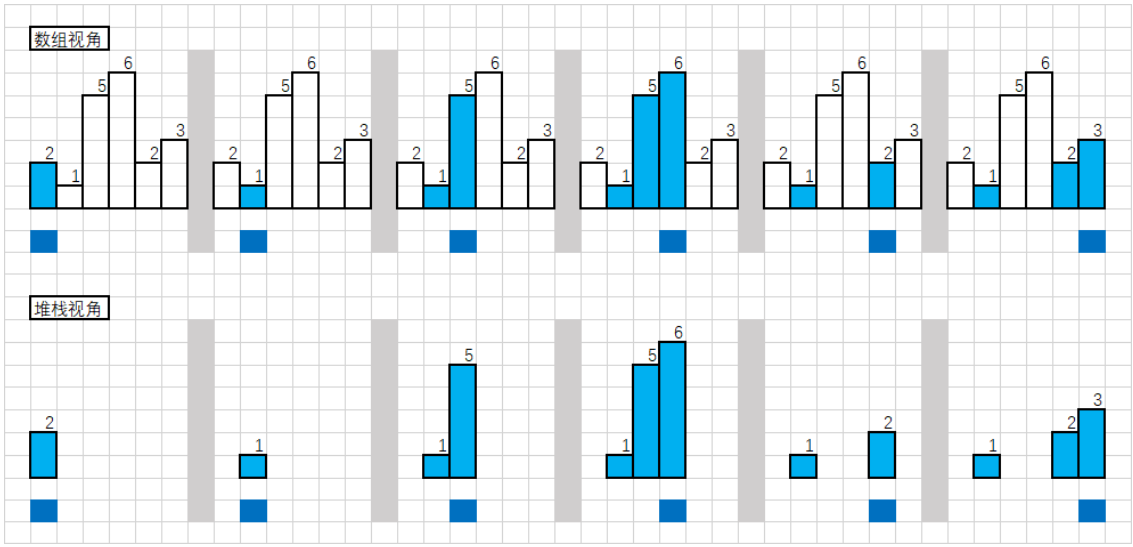


图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

示例:

输入：`[2,1,5,6,2,3]`
输出：10

解法：单调栈

1. 

2. 对于一个高度，如果能得到向左和向右的边界
3. 那么就能对每个高度求一次面积
4. 遍历所有高度，即可得出最大面积

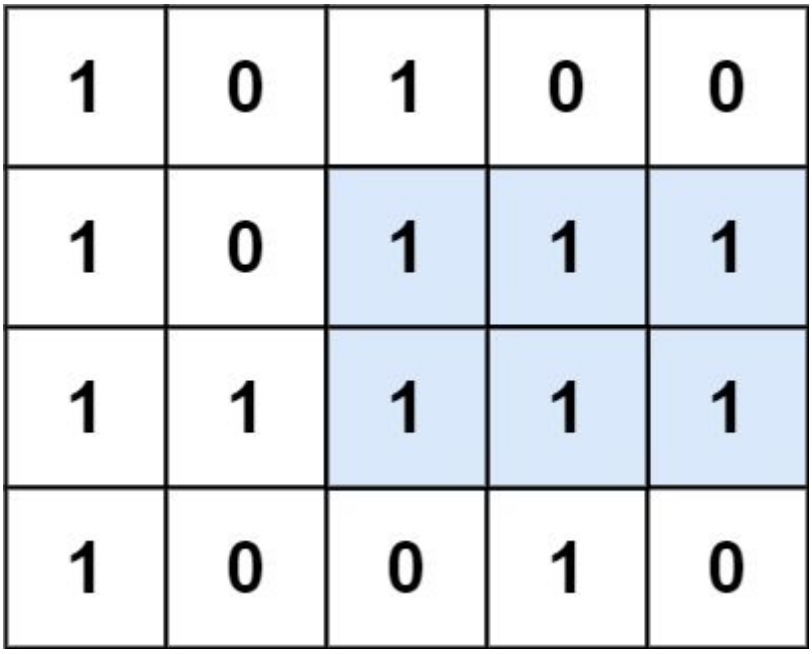
5. 使用单调栈，在出栈操作时得到前后边界并计算面积

```
var largestRectangleArea = function(heights) {
    heights.unshift(0);
    heights.push(0);
    const n = heights.length;
    const stack = [];
    let res = 0;

    for(let i = 0; i < n; i++) {
        while(stack.length > 0 && heights[i] < heights[stack[stack.length - 1]]) {
            let current = stack[stack.length - 1];
            stack.pop();
            let left = stack[stack.length - 1] + 1;
            let right = i - 1;
            res = Math.max(res, heights[current] * (right - left + 1));
        };
        stack.push(i);
    }

    return res;
};
```

0085 最大矩形

给定一个仅包含 `0` 和 `1` 、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 `1` 的最大矩形，并返回其面积。

示例 1：



输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
输出：6
解释：最大矩形如上图所示。

示例 2：

```
输入：matrix = []
输出：0
```

示例 3：

```
输入：matrix = [["0"]]
输出：0
```

示例 4：

```
输入：matrix = [["1"]]
输出：1
```

示例 5：

```
输入：matrix = [["0","0"]]
输出：0
```

提示：

- `rows == matrix.length`
- `cols == matrix[0].length`
- `0 <= row, cols <= 200`
- `matrix[i][j]` 为 `'0'` 或 `'1'`

解法1：暴力解

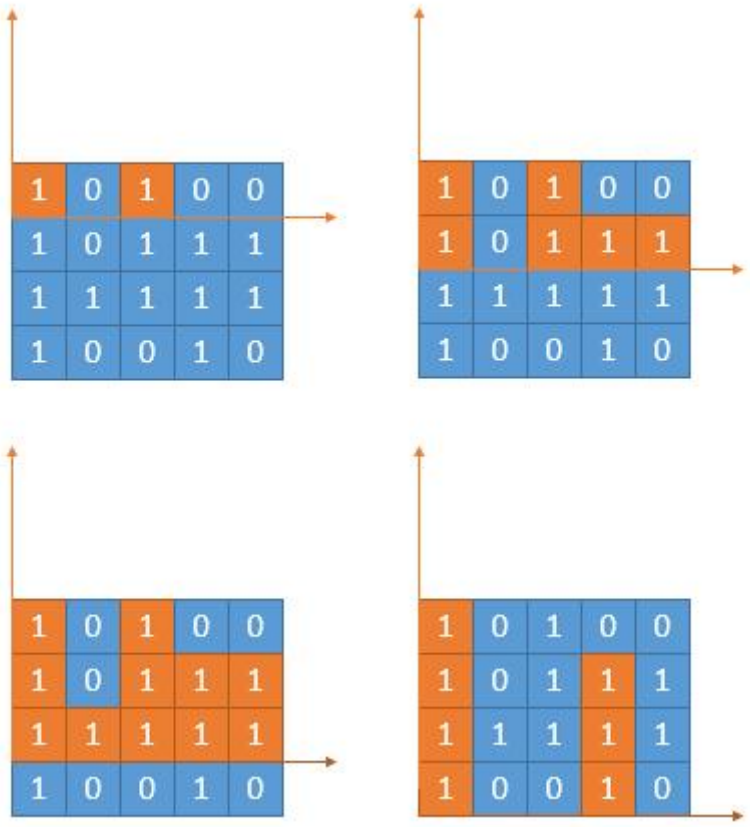
- 先求出第i行每个元素最大连通宽度
- 遍历每个元素，k = i 后 k 逐渐递减，向上寻找，求面积乘积

```
var maximalRectangle = function(matrix) {
    const m = matrix.length;
    if (m === 0) {
        return 0;
    }
    const n = matrix[0].length;
    const left = new Array(m).fill(0).map(() => new Array(n).fill(0));

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (matrix[i][j] === '1') {
                left[i][j] = (j === 0 ? 0 : left[i][j - 1]) + 1;
            }
        }
    }
```

```
    let ret = 0;
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (matrix[i][j] === '0') {
                continue;
            }
            let width = left[i][j];
            let area = width;
            for (let k = i - 1; k >= 0; k--) {
                width = Math.min(width, left[k][j]);
                area = Math.max(area, (i - k + 1) * width);
            }
            ret = Math.max(ret, area);
        }
    }
    return ret;
};
```

解法2：单调栈

本题和上一题有很大的相似之处，求出每一行的heights，传给上一题的函数即可



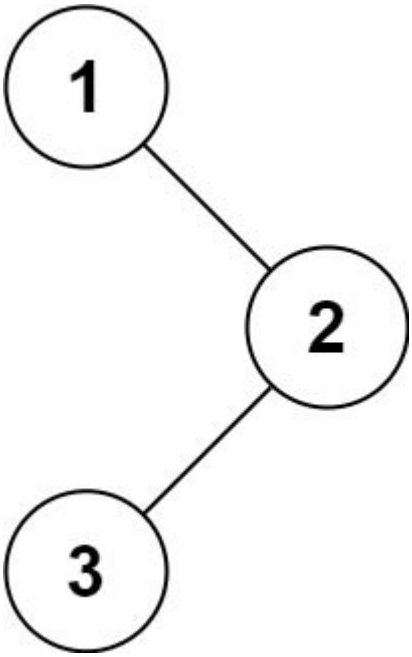

```
    var maximalRectangle = function (matrix) {
      if (matrix.length === 0) return 0;
      let res = 0;
      const height = new Array(matrix[0].length).fill(0);
      for (let i = 0; i < matrix.length; i++) {
        for (let j = 0; j < matrix[0].length; j++) {
          if (matrix[i][j] === '1') {
            height[j] += 1;
          } else {
```

```
            height[j] = 0;
          }
        }
        res = Math.max(res, largestRectangleArea(height))
      }
      return res;
    };

    var largestRectangleArea = function (heights) {
      heights.unshift(0);
      heights.push(0);
      const n = heights.length;
      const stack = [];
      let res = 0;

      for (let i = 0; i < n; i++) {
        while (stack.length > 0 && heights[i] < heights[stack[stack.length - 1]]) {
          let current = stack[stack.length - 1];
          stack.pop();
          let left = stack[stack.length - 1] + 1;
          let right = i - 1;
          res = Math.max(res, heights[current] * (right - left + 1));
        };
        stack.push(i);
      }
      heights.shift();
      heights.pop();
      return res;
    };
```

0094 二叉树的中序遍历

给定一个二叉树的根节点 `root` ，返回它的 **中序** 遍历。

示例 1：



输入：root = [1,null,2,3]
输出：[1,3,2]

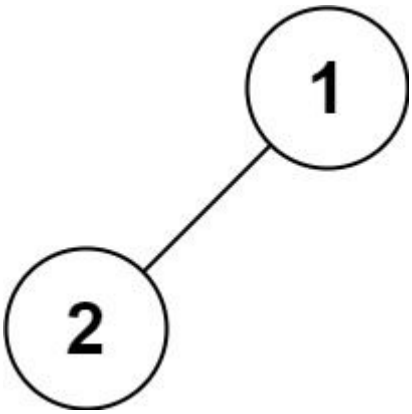
示例 2：

输入：root = []
输出：[]

示例 3：

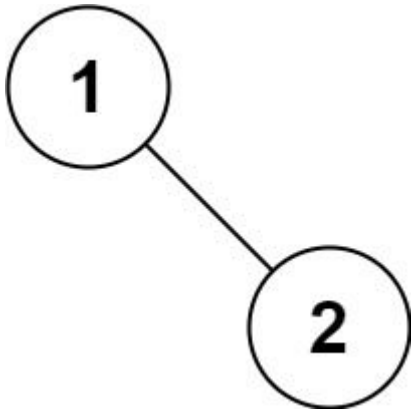
输入：root = [1]
输出：[1]

示例 4：



输入：root = [1,2]
输出：[2,1]

示例 5：



输入：root = [1,null,2]
输出：[1,2]

提示：

- 树中节点数目在范围 `[0, 100]` 内
- `-100 <= Node.val <= 100`

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

解法1：递归

中序遍历即首先遍历左子树，然后访问当前节点，最后遍历右子树

```
var inorderTraversal = function(root) {
    if(root) {
        const res = []
        function inorder(node) {
            if(node.left) {
                inorder(node.left)
            }
            res.push(node.val)
            if(node.right) {
                inorder(node.right)
            }
        }
        inorder(root)
        return res
    }else return []
};
```

解法2：迭代

```
var inorderTraversal = function(root) {
    const res = [];
    const stk = [];
    while (root || stk.length) {
```

```
        while (root) {
            stk.push(root);
            root = root.left;
        }
        root = stk.pop();
        res.push(root.val);
        root = root.right;
    }
    return res;
};
```

解法3：Morris 中序遍历

Morris 遍历算法是另一种遍历二叉树的方法，它能将非递归的中序遍历空间复杂度降为 O(1)。

Morris 遍历算法整体步骤如下（假设当前遍历到的节点为 x）：

- 如果 x 无左孩子，先将 x 的值加入答案数组，再访问 x 的右孩子，即 x=x.right
- 如果 x 有左孩子，则找到 x 左子树上最右的节点（即左子树中序遍历的最后一个节点，xx 在中序遍历中的前驱节点），我们记为 predecessor。根据 predecessor 的右孩子是否为空，进行如下操作
    - 如果 predecessor 的右孩子为空，则将其右孩子指向 x，然后访问 x 的左孩子，即 x=x.left
    - 如果 predecessor 的右孩子不为空，则此时其右孩子指向 x，说明我们已经遍历完 xx 的左子树，我们将 predecessor 的右孩子置空，将 xx 的值加入答案数组，然后访问 x 的右孩子，即 x=x.right
- 重复上述操作，直至访问完整棵树

```
var inorderTraversal = function(root) {
    const res = [];
    let predecessor = null;

    while (root) {
        if (root.left) {
            // predecessor 节点就是当前 root 节点向左走一步，然后一直向右走至无法走为止
            predecessor = root.left;
            while (predecessor.right && predecessor.right !== root) {
                predecessor = predecessor.right;
            }

            // 让 predecessor 的右指针指向 root，继续遍历左子树
            if (!predecessor.right) {
                predecessor.right = root;
                root = root.left;
            }
            // 说明左子树已经访问完了，我们需要断开链接
            else {
                res.push(root.val);
                predecessor.right = null;
                root = root.right;
            }
        }
        // 如果没有左孩子，则直接访问右孩子
        else {
            res.push(root.val);
            root = root.right;
```

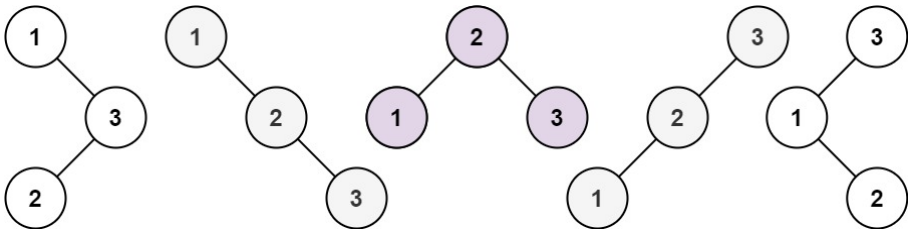
```
        }
    }

    return res;
};
```

0096 不同的二叉搜索树

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种?

示例:

```
输入: 3
输出: 5
解释:
给定 n = 3, 一共有 5 种不同结构的二叉搜索树:
```



解法：动态规划

题目要求是计算不同二叉搜索树的个数。为此，我们可以定义两个函数：

- G(n): 长度为 n 的序列能构成的不同二叉搜索树的个数。
- F(i,n): 以 i 为根、序列长度为 n 的不同二叉搜索树个数 (1 ≤ i ≤ n)。

可见，G(n) 是我们求解需要的函数

$$G(n) = \sum_{i=1}^{n} F(i, n)$$

举例而言，创建以 33 为根、长度为 77 的不同二叉搜索树，整个序列是[1,2,3,4,5,6,7]，我们需要从左子序列[1,2]构建左子树，从右子序列[4,5,6,7]构建右子树，然后将它们组合（即**笛卡尔积**）。

对于这个例子，不同二叉搜索树的个数为 F(3, 7)。我们将[1,2]构建不同左子树的数量表示为 G(2), 从[4,5,6,7]构建不同右子树的数量表示为 G(4)，注意到 G(n) 和序列的内容无关，**只和序列的长度有关**。于是，F(3,7)=G(2)·G(4)。 因此，我们可以得到以下公式：

$$G(n) = \sum_{i=1}^{n} G(i-1) \cdot G(n-i)$$

```
/**
 * @param {number} n
 * @return {number}
 */
var numTrees = function(n) {
    const G = new Array(n + 1).fill(0);
    G[0] = 1;
    G[1] = 1;
```

```
    for (let i = 2; i <= n; ++i) {
        for (let j = 1; j <= i; ++j) {
            G[i] += G[j - 1] * G[i - j];
        }
    }
    return G[n];
};
```

0098 验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。
- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

解法1：递归

如果该二叉树的左子树不为空，则左子树上所有节点的值均小于它的根节点的值； 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；它的左右子树也为二叉搜索树。

因此设计一个helper函数，传入root，上限，下限，左右子树都为二叉搜索树时，返回true

```
/**
 * 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}
 */
var isValidBST = function(root) {
    const helper = (root, lower, upper) => {
        if(root === null) {
            return true;
        }
        if(root.val <= lower || root.val >= upper) {
            return false;
        }
        return helper(root.left, lower, root.val) && helper(root.right, root.val, upper);
    }
    return helper(root, -Infinity, Infinity);
};
```

解法2：中序遍历

二叉搜索树「中序遍历」得到的值构成的序列一定是升序的，这启示我们在中序遍历的时候实时检查当前节点的值是否大于前一个中序遍历到的节点的值即可。如果均大于说明这个序列是升序的，整棵树是二叉搜索树，否则不是

```
var isValidBST = function(root) {
    let stk = [];
    let inorder = -Infinity;
    while(stk.length || root) {
        while(root) {
            stk.push(root);
            root = root.left;
        }
        root = stk.pop();
        if(root.val <= inorder) return false;
        inorder = root.val;
        root = root.right;
    }
    return true;
};
```

0101 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

进阶:

你可以运用递归和迭代两种方法解决这个问题吗?

解法1:递归

如果同时满足下面的条件,两个树互为镜像:

- 它们的两个根结点具有相同的值
- 每个树的右子树都与另一个树的左子树镜像对称

用两个指针p和q, p 右移时,q 左移,p 左移时,q右移。每次检查当前 p 和 q节点的值是否相等,如果相等再判断左右子树是否对称

在return的时候注意用与连接,达到同时判断的目的

```
var isSymmetric = function(root) {
    const check = (p,q) => {
        if(!p && !q) return true
        if(!p || !q) return false
        return p.val === q.val && check(p.left, q.right) && check(p.right,
q.left)
    }
    return check(root, root)
};
```

解法2:迭代

递归转迭代通常使用一个队列来维护

通过一个队列进行维护,首先把root的左右子树放进去

当队列长度大于0时,将头部两个取出进行比较,之后再按顺序将左子树的左子树,右子树的右子树,左子树的右子树,右子树的左子树放进队列,进行比较

```
var isSymmetric = function(root) {
    if(root === null || (root.left === null && root.right === null)) return true
    const q = []
    let left = new TreeNode(), right = new TreeNode()
    q.push(root.left)
    q.push(root.right)
    while(q.length > 0) {
        left = q.shift();
        right = q.shift();
        if(left === null && right === null) continue
        if(left === null || right === null) return false
        if(left.val !== right.val) return false
        q.push(left.left)
        q.push(right.right)
        q.push(left.right)
        q.push(right.left)
    }
    return true
};
```

0102 二叉树的层序遍历

给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。

示例：

二叉树：`[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

解法：广度优先搜索 + 循环不变式

我们可以用一种巧妙的方法修改广度优先搜索：

- 首先根元素入队
- 当队列不为空的时候
    - 求当前队列的长度length
    - 依次从队列中取length个元素进行拓展，然后进入下一次迭代

它和普通广度优先搜索的区别在于，普通广度优先搜索每次只取一个元素拓展，而这里每次取length个元素。在上述过程中的第 i 次迭代就得到了二叉树的第 i层的length个元素

```
var levelOrder = function(root) {
    const res = []
    if(!root) return res
    const q = []
    q.push(root)
    while(q.length > 0) {
        const currentLevelSize = q.length
        res.push([])
        for(let i = 1; i <= currentLevelSize; i++) {
            const node = q.shift()
            res[res.length - 1].push(node.val)
            if(node.left) q.push(node.left)
            if(node.right) q.push(node.right)
        }
    }
    return res
};
```

0104 二叉树的最大深度

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。

示例:

给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

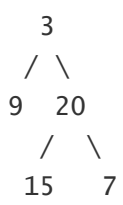
给定一个二叉树，找出其最大深度。

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。

示例:

给定二叉树 `[3,9,20,null,null,15,7]`,



返回它的最大深度 3 。

解法1：深度优先搜索

```
var maxDepth = function(root) {
    let res = 0
    if(!root) return res
    const isDeepest = (current, depth) => {
        if(current.left) isDeepest(current.left, depth + 1)
        if (current.right) isDeepest(current.right, depth + 1)
        res = Math.max(res, depth)
        return
    }
    isDeepest(root, 1)
    return res
};
```

```
var maxDepth = function(root) {
    if(!root) return 0;
    let leftD = 1 + maxDepth(root.left)
    let rightD = 1 + maxDepth(root.right)
    return Math.max(leftD, rightD)
};
```

解法2：广度优先搜索

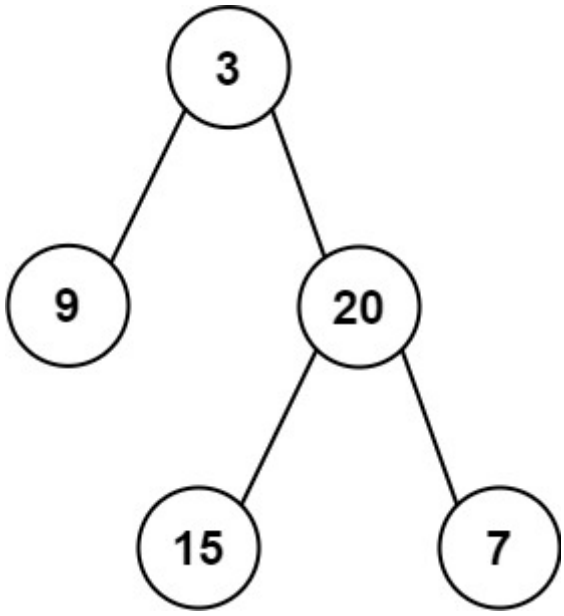
```
var maxDepth = function(root) {
    let res = 0
    if(!root) return res
    const q = []
    q.push(root)
    while(q.length > 0) {
        let length = q.length
        while(length > 0) {
            let node = q.shift()
            if(node.left) q.push(node.left)
            if(node.right) q.push(node.right)
            length--
        }
```

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

0105 从前序与中序遍历序列构造二叉树

给定一棵树的前序遍历 `preorder` 与中序遍历 `inorder`。请构造二叉树并返回其根节点。

示例 1:




```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

示例 2:

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```

提示:

- `1 <= preorder.length <= 3000`
- `inorder.length == preorder.length`
- `-3000 <= preorder[i], inorder[i] <= 3000`
- `preorder 和 inorder 均无重复元素`
- `inorder 均出现在 preorder`
- `preorder 保证为二叉树的前序遍历序列`
- `inorder 保证为二叉树的中序遍历序列`

解法：递归

对于任意一颗树而言，前序遍历的形式总是

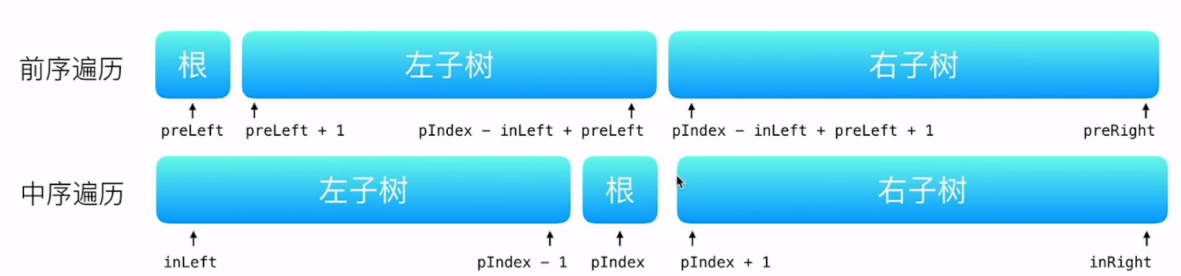
```
[ 根节点，[左子树的前序遍历结果]，[右子树的前序遍历结果] ]
```

即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是

```
[ [左子树的中序遍历结果]，根节点，[右子树的中序遍历结果] ]
```

只要我们在中序遍历中**定位**到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有**左右括号**进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地对构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。



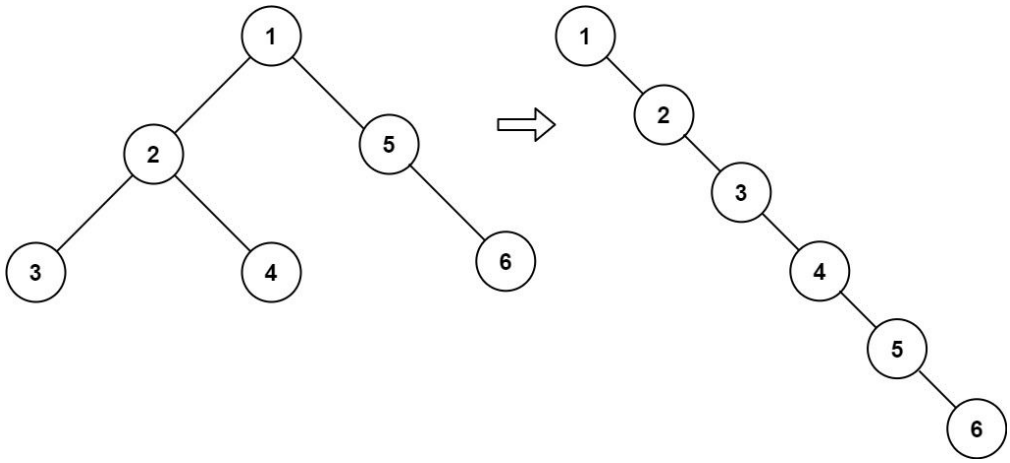

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {number[]} preorder
 * @param {number[]} inorder
 * @return {TreeNode}
 */
var buildTree = function(preorder, inorder) {
    const idxMap = new Map()
    const n = inorder.length
    for(let i = 0; i < n; i++) {
        idxMap.set(inorder[i],i)
    }

    const myBuildTree = (preLeft, preRight, inLeft, inRight) => {
        if(preLeft > preRight) return null
        const pIndex = idxMap.get(preorder[preLeft])
        const root = new TreeNode(preorder[preLeft])
        root.left = myBuildTree(preLeft+1, pIndex-inLeft+preLeft, inLeft, pIndex-1)
        root.right = myBuildTree(pIndex-inLeft+preLeft+1, preRight, pIndex+1, inRight)
        return root
    }
    return myBuildTree(0, n-1, 0, n-1)
};
```

0114 二叉树展开为链表

给你二叉树的根结点 `root` ，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode` ，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null` 。
- 展开后的单链表应该与二叉树 **先序遍历** 顺序相同。

示例 1：




```
输入：root = [1,2,5,3,4,null,6]
输出：[1,null,2,null,3,null,4,null,5,null,6]
```

示例 2：

```
输入：root = []
输出：[]
```

示例 3：

```
输入：root = [0]
输出：[0]
```

提示：

- 树中结点数在范围 `[0, 2000]` 内
- `-100 <= Node.val <= 100`

进阶：你可以使用原地算法（`O(1)` 额外空间）展开这棵树吗？

解法1：前序遍历

通过前序遍历，把每个节点按照前序遍历的顺序存到一个list中，最后再将其连接起来

```
/**
 * 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.
 */
var flatten = function(root) {
    const list = [];
    preorderTraversal(root, list);
    const size = list.length;
    for (let i = 1; i < size; i++) {
        const prev = list[i - 1], curr = list[i];
        prev.left = null;
        prev.right = curr;
    }
};

const preorderTraversal = (root, list) => {
    if (root != null) {
        list.push(root);
        preorderTraversal(root.left, list);
        preorderTraversal(root.right, list);
    }
}
```

解法2：前序遍历和展开同时进行

使用方法一的前序遍历，由于将节点展开之后会破坏二叉树的结构而丢失子节点的信息，因此前序遍历和展开为单链表分成了两步。能不能在不丢失子节点的信息的情况下，将前序遍历和展开为单链表同时进行？

之所以会在破坏二叉树的结构之后丢失子节点的信息，是因为在对左子树进行遍历时，没有存储右子节点的信息，在遍历完左子树之后才获得右子节点的信息。只要对前序遍历进行修改，在遍历左子树之前就获得左右子节点的信息，并存入栈内，子节点的信息就不会丢失，就可以将前序遍历和展开为单链表同时进行。

该做法不适用于递归实现的前序遍历，只适用于迭代实现的前序遍历。修改后的前序遍历的具体做法是，每次从栈内弹出一个节点作为当前访问的节点，获得该节点的子节点，如果子节点不为空，则依次将右子节点和左子节点压入栈内（注意入栈顺序）。

展开为单链表的做法是，维护上一个访问的节点 prev，每次访问一个节点时，令当前访问的节点为 curr，将 prev 的左子节点设为 null 以及将 prev 的右子节点设为 curr，然后将 curr 赋值给 prev，进入下一个节点的访问，直到遍历结束。需要注意的是，初始时 prev 为 null，只有在 prev 不为 null 时才能对 prev 的左右子节点进行更新。

```
var flatten = function(root) {
    if(root === null) return
    const stack = []
    stack.push(root)
    let prev = null
    while(stack.length) {
        const current = stack.pop()
        if(prev !== null) {
            prev.left = null
            prev.right = current
        }
        if(current.right !== null) stack.push(current.right)
        if(current.left !== null) stack.push(current.left)
        prev = current
    }
};
```

解法3：寻找前驱节点

注意到前序遍历访问各节点的顺序是根节点、左子树、右子树。如果一个节点的左子节点为空，则该节点不需要进行展开操作。如果一个节点的左子节点不为空，则该节点的左子树中的最后一个节点被访问之后，该节点的右子节点被访问。该节点的左子树中最后一个被访问的节点是左子树中的最右边的节点，也是该节点的前驱节点。因此，问题转化成寻找当前节点的前驱节点。

具体做法是，对于当前节点，如果其左子节点不为空，则在其左子树中找到最右边的节点，作为前驱节点，将当前节点的右子节点赋给前驱节点的右子节点，然后将当前节点的左子节点赋给当前节点的右子节点，并将当前节点的左子节点设为空。对当前节点处理结束后，继续处理链表中的下一个节点，直到所有节点都处理结束。

```
var flatten = function(root) {
    let current = root
    while(current !== null) {
        if(current.left !== null) {
            const next = current.left
            let pre = next
            while(pre.right !== null) {
                pre = pre.right
            }
            pre.right = current.right
            current.left = null
            current.right = next
        }
        current = current.right
    }
};
```

0121 买卖股票的最佳时机

给定一个数组 `prices` ，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0` 。

示例 1：

```
输入：[7,1,5,3,6,4]
输出：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。
```

示例 2：

```
输入：prices = [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。
```

提示：

- `1 <= prices.length <= 10^5`
- `0 <= prices[i] <= 10^4`

解法：找最小

```
var maxProfit = function(prices) {
    let max = 0, min = prices[0]
    for(let i = 0; i < prices.length; i++) {
        if(prices[i] < min) {
            min = prices[i]
            continue
        }
        max = (prices[i] - min) > max ? (prices[i] - min) : max
    }
    return max
};
```

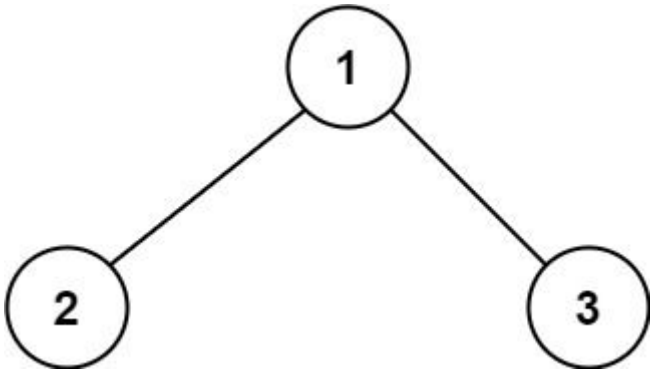
0124 二叉树中的最大路径和

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 **至多出现一次** 。该路径 **至少包含一个** 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

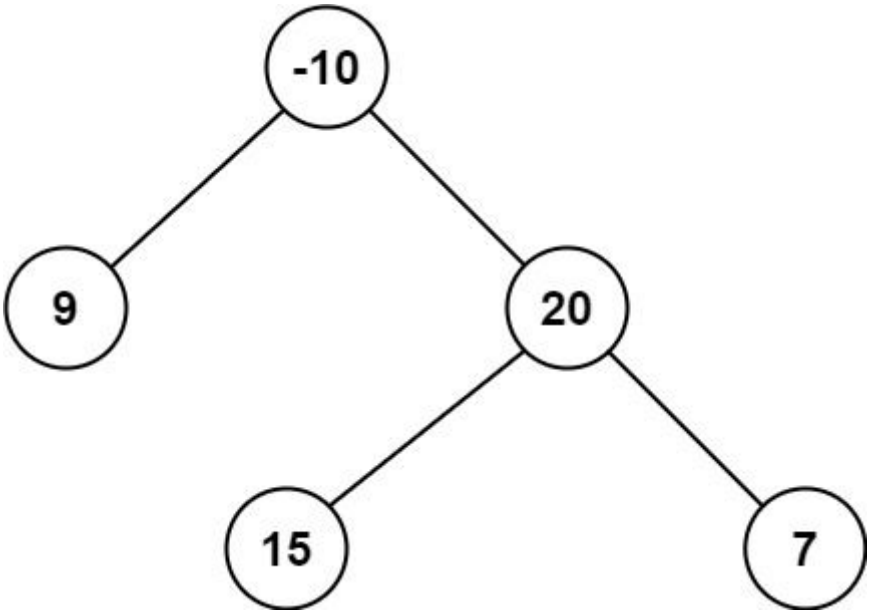
给你一个二叉树的根节点 `root` ，返回其 **最大路径和** 。

示例 1：



输入：root = [1,2,3]
输出：6
解释：最优路径是 2 -> 1 -> 3 ，路径和为 2 + 1 + 3 = 6

示例 2:



输入：root = [-10,9,20,null,null,15,7]
输出：42
解释：最优路径是 15 -> 20 -> 7 ，路径和为 15 + 20 + 7 = 42

提示:

- 树中节点数目范围是 `[1, 3 * 10^4]`
- `-1000 <= Node.val <= 1000`

解法：递归

- 通过递归，分辨判断左右子节点贡献值，如果大于0，则取用，如果小于0，则抛弃
- 注意初始值不能设为0，如果设为零，遇到全是负数的情况则无法判断，所以要设置成**负无穷**

```
var maxPathSum = function(root) {
    let res = -Infinity
    const maximum = (node) => {
        if(node === null) return 0
        let leftMax = Math.max(maximum(node.left), 0)
        let rightMax = Math.max(maximum(node.right), 0)
        let current = leftMax + rightMax + node.val
        res = Math.max(current, res)
        return node.val + Math.max(leftMax, rightMax)
    }
    maximum(root)
    return res
};
```

0128 最长连续序列

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？

示例 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。

示例 2：

输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9

提示：

- `0 <= nums.length <= 10^4`
- `-10^9 <= nums[i] <= 10^9`

解法：哈希表

- 创建一个Set，去除重复元素
- 遍历这个哈希表
  - 判断是否存在`num-1`，如果存在则跳过本次，因为从`num-1`开始一定更长
  - 不断匹配`num+1` `num+2` `num+3`等是否存在，寻找最长匹配，更新max

```
```

```
var longestConsecutive = function(nums) {
    let max = 0
    const hash = new Set()
    for(let num of nums) {
        hash.add(num)
    }
    for(let num of hash) {
        let current_max = 1
        if(hash.has(num-1)) continue
        while(hash.has(num+1)) {
            current_max += 1
            num++
        }
        max = Math.max(current_max, max)
    }
    return max
};
```

0136 只出现一次的数字

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明:

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗?

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

解法1：遍历

```
var singleNumber = function(nums) {
    nums.sort()
    if(nums.length === 1) return nums[0]
    let i = 1
    while(i <= nums.length) {
        if(nums[i] === nums[i-1]) {
            i = i+2
        }else if(nums[i] === nums[i+1]) return nums[i-1]
    }
};
```

解法2：异或运算符

异或运算有如下三种性质：

- 任何数和 0 做异或运算，结果仍然是原来的数
- 任何数和其自身做异或运算，结果是 0
- 异或运算满足交换律和结合律

根据第三条性质，数组中全部元素的异或运算结果可以写成：

$(a_1 \oplus a_1) \oplus (a_2 \oplus a_2) \oplus \cdots \oplus (a_m \theta a_m) \oplus a_{m+1}$

```
var singleNumber = function(nums) {
    return nums.reduce((a,b)=>a^b)
};
```

0139 单词拆分

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：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: 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。
```

示例 3：

```
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

解法：动态规划

状态转移方程如下：

$dp[i] = dp[j] \&\& check(s[j \ldots i-1]$

注意：

- dp的长度设为n+1是为了预设0位置为分割点
- dp[0]设为true

- **状态转移方程也可以带有类似check的判断条件，不要总局限于dp**

```
var wordBreak = function(s, wordDict) {
    const n = s.length
    const dp = new Array(n+1).fill(false)
    dp[0] = true
    for(let i = 1; i <= n; i++) {
        for(let j = 0; j < i; j++) {
            if(dp[j] && wordDict.includes(s.substr(j, i - j))) {
                dp[i] = true
                break
            }
        }
    }
    return dp[n]
};
```

0141 环形链表

给定一个链表，判断链表中是否有环。

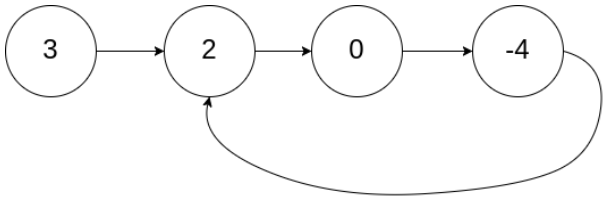
如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。 为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。**注意：** `pos` **不作为参数进行传递**，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true` 。 否则，返回 `false` 。

进阶：

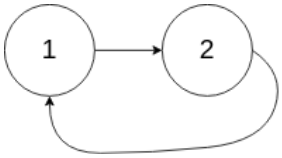
你能用 *O(1)*（即，常量）内存解决此问题吗？

示例 1：




```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```

示例 2：



输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1
输出：false
解释：链表中没有环。

提示：

- 链表中节点的数目范围是 `[0, 10^4]`
- `-10^5 <= Node.val <= 10^5`
- `pos` 为 `-1` 或者链表中的一个 **有效索引** 。

解法：快慢指针

快指针每次走两步，慢指针每次走一步，如果有环，那么二者必然会相遇

```
var hasCycle = function(head) {
    let slow = fast = head
    while(slow && fast && fast.next) {
        slow = slow.next
        fast = fast.next.next
        if(slow === fast) return true
    }
    return false
};
```

0142 环形链表2

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`。

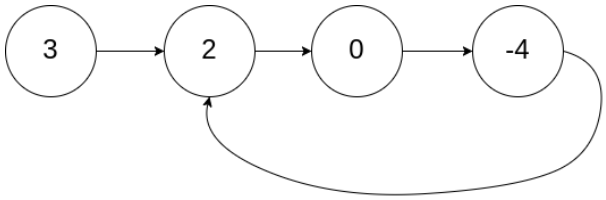
为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。**注意，`pos` 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

说明：不允许修改给定的链表。

进阶：

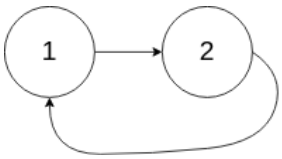
- 你是否可以使用 `O(1)` 空间解决此题？

示例 1：



输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。

示例 2：



输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1
输出：返回 null
解释：链表中没有环。

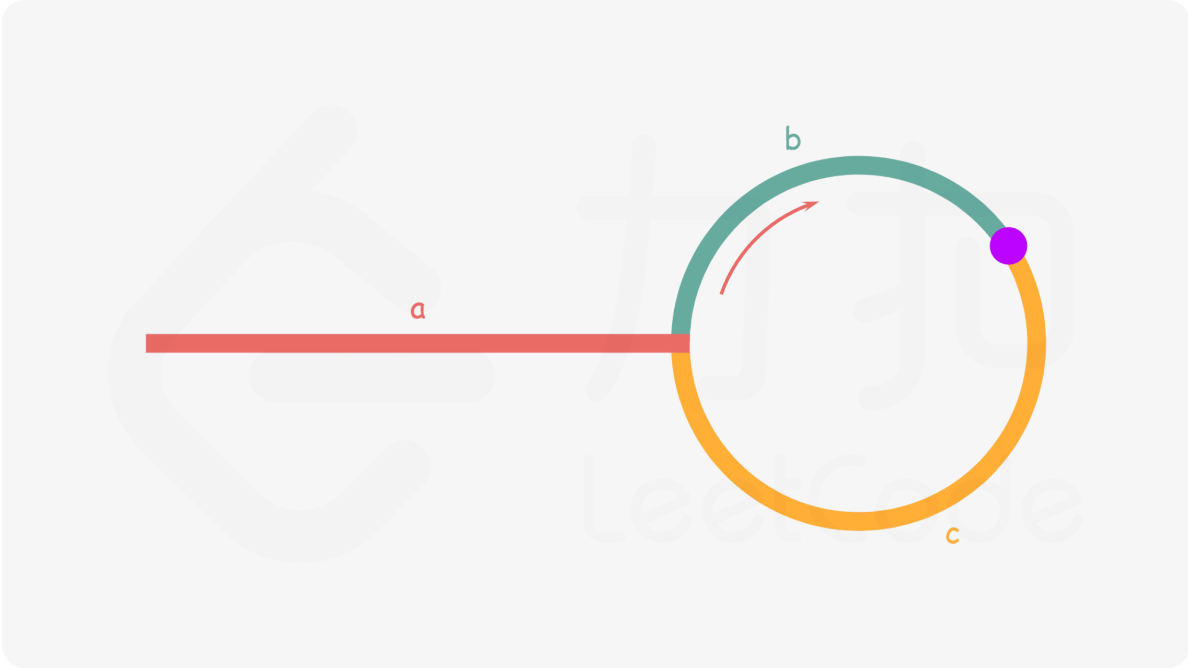
提示：

- 链表中节点的数目范围在范围 `[0, 10^4]` 内
- `-10^5 <= Node.val <= 10^5`
- `pos` 的值为 `-1` 或者链表中的一个有效索引

解法1：Hash表

```
var detectCycle = function(head) {
    const hash = new Map()
    while(head) {
        if(hash.get(head)) return head
        hash.set(head, 1)
        head = head.next
    }
    return null
};
```

解法2：快慢指针



如下图所示，设链表中环外部分的长度为 a。slow 指针进入环后，又走了 b 的距离与 fast 相遇。此时，fast 指针已经走完了环的 n 圈，因此它走过的总距离为 a+n(b+c)+b=a+(n+1)b+nc

根据题意，任意时刻，fast 指针走过的距离都为 slow 指针的 2 倍。因此，我们有$a=c+(n-1)(b+c)$

因此，当发现 slow 与fast 相遇时，我们再额外使用一个指针 ptr。起始，它指向链表头部；随后，它和 slow 每次向后移动一个位置。最终，它们会在入环点相遇。

```
var detectCycle = function(head) {
    if (head === null) {
        return null;
    }
    let slow = head, fast = head;
    while (fast !== null) {
        slow = slow.next;
        if (fast.next !== null) {
            fast = fast.next.next;
        } else {
            return null;
        }
        if (fast === slow) {
            let ptr = head;
            while (ptr !== slow) {
                ptr = ptr.next;
                slow = slow.next;
            }
            return ptr;
        }
    }
    return null;
};
```