关键点解析

- 双指针

这道题如果不要求，O(n) 的时间复杂度， O(1) 的空间复杂度的话，会很简单。
但是这道题是要求的，这种题的思路一般都是采用双指针

- 如果是数据是无序的，就不可以用这种方式了，从这里也可以看出排序在算法中的基础性和重要性。
- 注意 nums 为空时的边界条件。

代码

Javascript Code:

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function (nums) {
  const size = nums.length;
  if (size == 0) return 0;
  let slowP = 0;
  for (let fastP = 0; fastP < size; fastP++) {
    if (nums[fastP] !== nums[slowP]) {
      slowP++;
      nums[slowP] = nums[fastP];
    }
  }
  return slowP + 1;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

7. 两数相除

题目描述

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

整数除法的结果应当截去（truncate）其小数部分，例如：truncate(8.345) = 8 以及 truncate(-2.7335) = -2

示例 1:

输入: dividend = 10, divisor = 3
输出: 3
解释: 10/3 = truncate(3.33333..) = truncate(3) = 3
示例 2:

输入: dividend = 7, divisor = -3
输出: -2
解释: 7/-3 = truncate(-2.33333..) = -2

提示：

被除数和除数均为 32 位有符号整数。
除数不为 0。
假设我们的环境只能存储 32 位有符号整数，其数值范围是 [−231, 231 − 1]。本题中，如果除法结果溢出，则返回 231 − 1。

前置知识

- 二分法

公司

- Facebook
- Microsoft
- Oracle

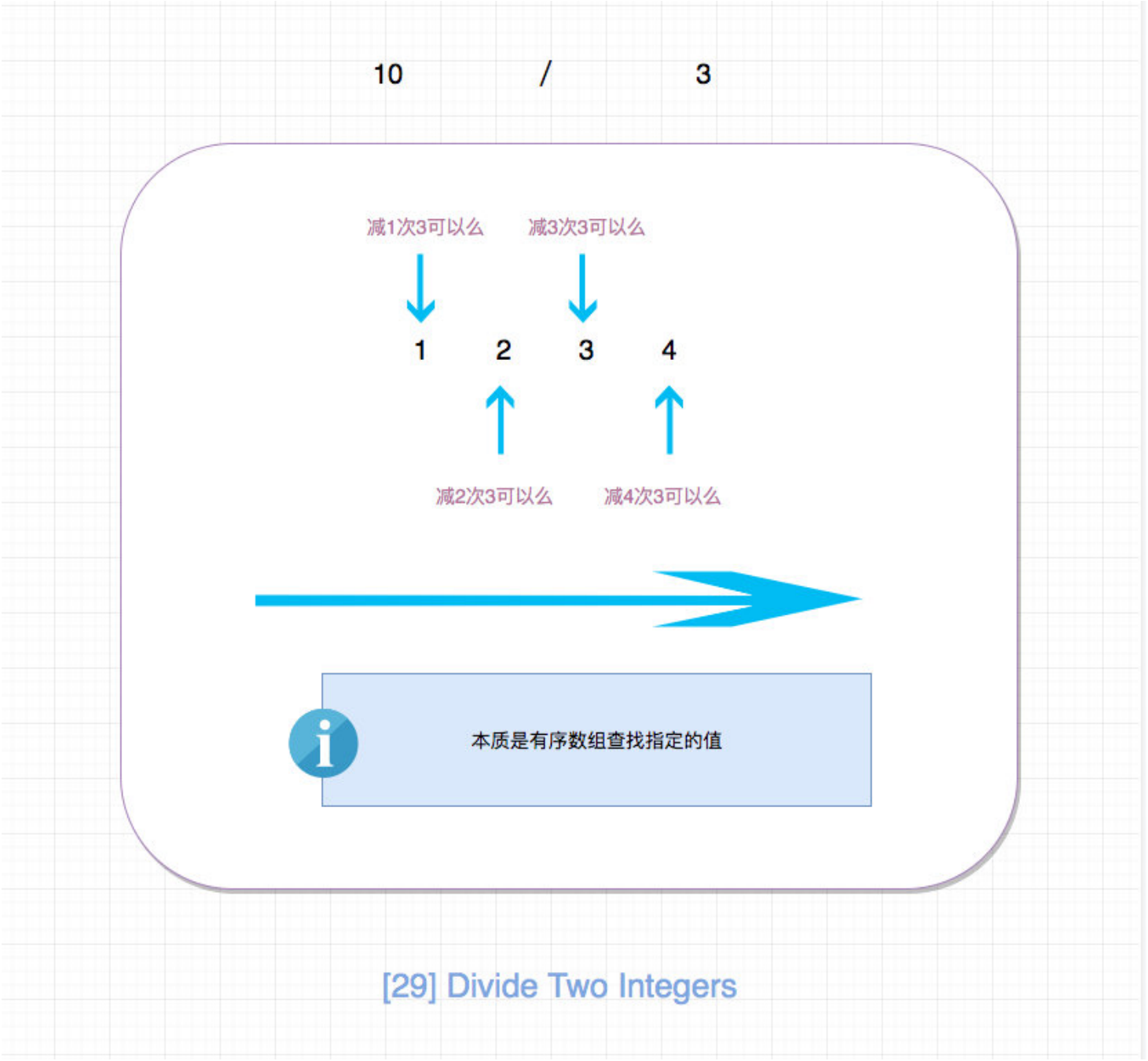
思路

符合直觉的做法是，减数一次一次减去被减数，不断更新差，直到差小于 0，我们减了多少次，结果就是多少。

核心代码：

```
let acc = divisor;
let count = 0;

while (dividend - acc >= 0) {
  acc += divisor;
  count++;
}

return count;
```

这种做法简单直观，但是性能却比较差. 下面来介绍一种性能更好的方法。



通过上面这样的分析，我们直到可以使用二分法来解决，性能有很大的提升。

关键点解析

- 二分查找
- 正负数的判断中，这样判断更简单。

```
const isNegative = dividend > 0 !== divisor > 0;
```

或者利用异或：

```
const isNegative = dividend ^ (divisor < 0);
```

代码

```javascript
/*
 * @lc app=leetcode id=29 lang=javascript
 *
 * [29] Divide Two Integers
 */
/**
 * @param {number} dividend
 * @param {number} divisor
 * @return {number}
 */
var divide = function (dividend, divisor) {
  if (divisor === 1) return dividend;

  // 这种方法很巧妙，即符号相同则为正，不同则为负
  const isNegative = dividend > 0 !== divisor > 0;

  const MAX_INTERGER = Math.pow(2, 31);

  const res = helper(Math.abs(dividend), Math.abs(divisor));

  // overflow
  if (res > MAX_INTERGER - 1 || res < -1 * MAX_INTERGER) {
    return MAX_INTERGER - 1;
  }

  return isNegative ? -1 * res : res;
};

function helper(dividend, divisor) {
  // 二分法
  if (dividend <= 0) return 0;
  if (dividend < divisor) return 0;
  if (divisor === 1) return dividend;

  let acc = 2 * divisor;
  let count = 1;

  while (dividend - acc > 0) {
    acc += acc;
    count += count;
  }
  // 直接使用位移运算，比如acc >> 1会有问题
  const last = dividend - Math.floor(acc / 2);
```

```
  return count + helper(last, divisor);
}
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

8. 下一个排列

题目描述

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。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1

前置知识

- 回溯法

公司

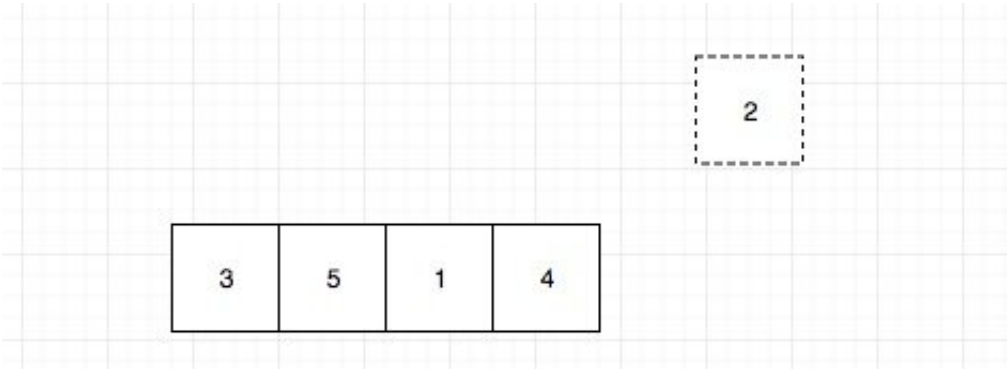
- 阿里
- 腾讯
- 百度
- 字节

思路

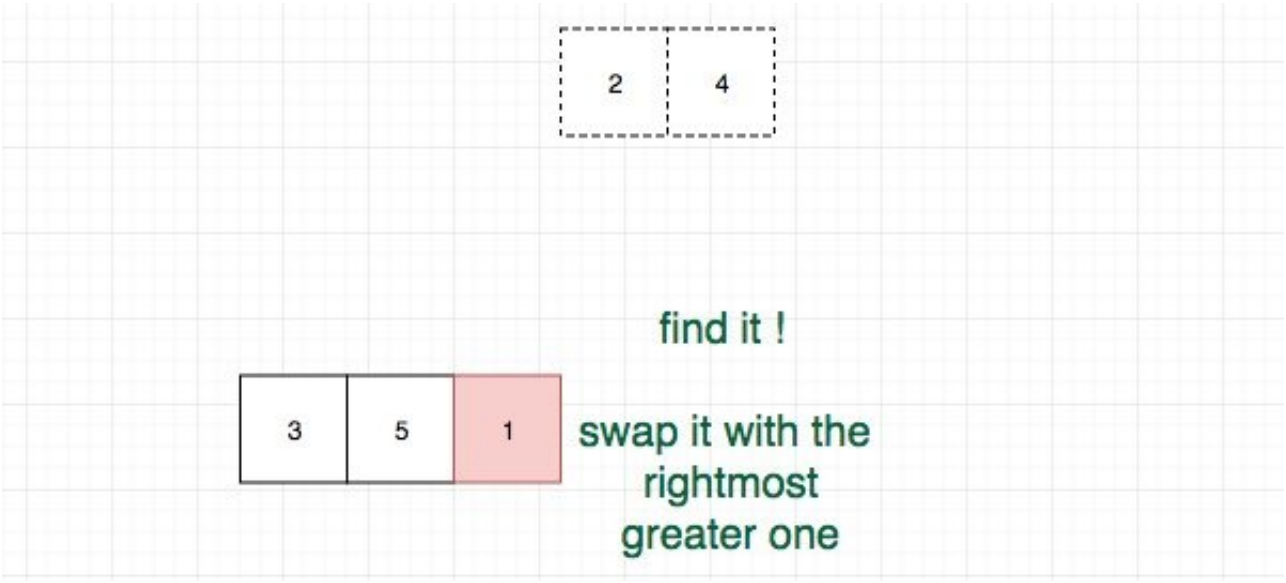
符合直觉的方法是按顺序求出所有的排列，如果当前排列等于 nums，那么我直接取下一个但是这种做法不符合 constant space 要求（题目要求直接修改原数组）,时间复杂度也太高，为 O(n!),肯定不是合适的解。

我们也可以以回溯的角度来思考这个问题，即从后往前思考。

让我们先回溯一次，即思考最后一个数字是如何被添加的。



由于这个时候可以选择的元素只有 2，我们无法组成更大的排列，我们继续回溯，直到如图：

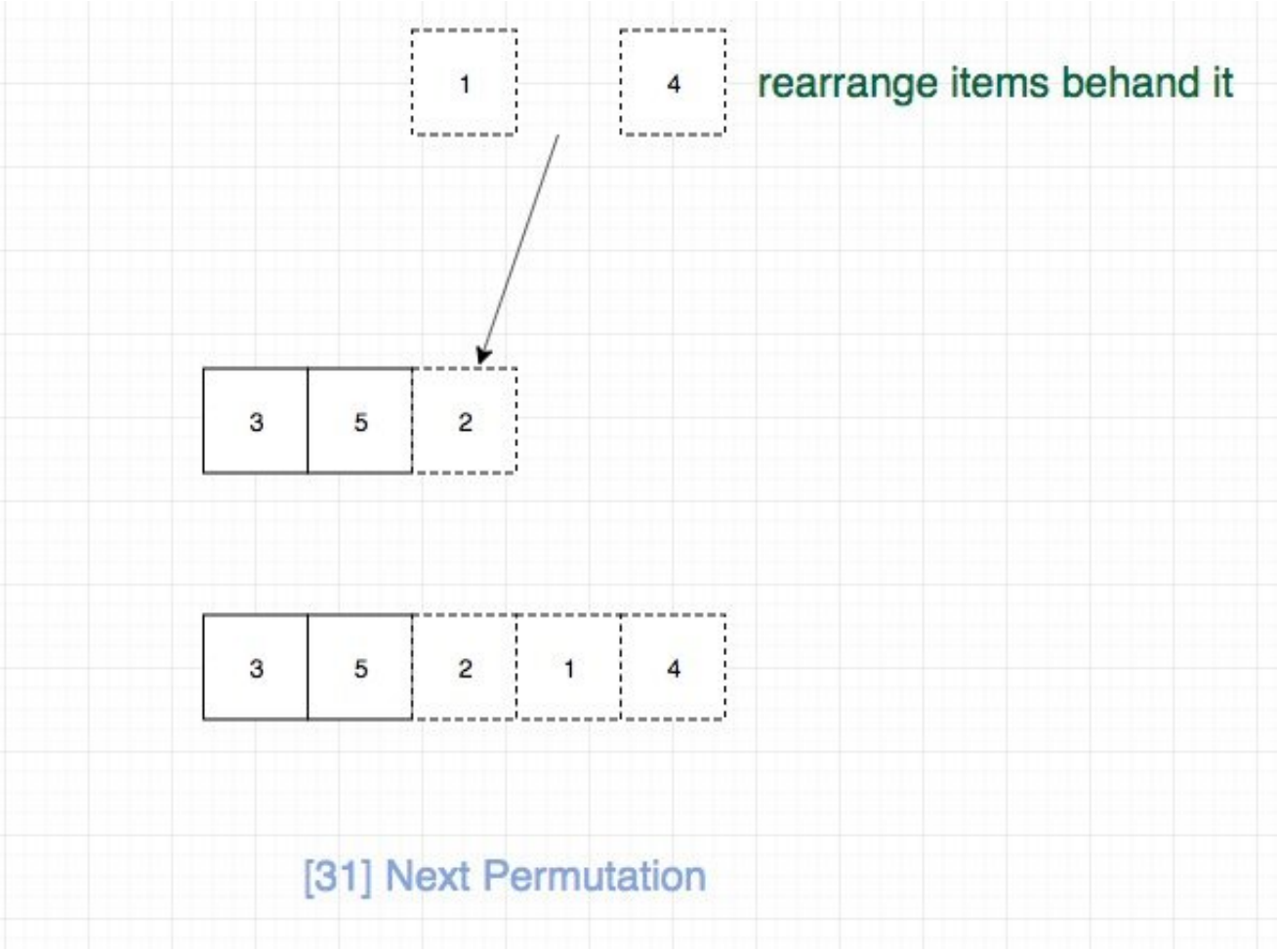


我们发现我们可以交换 4 和 2 就会变小，因此我们不能进行交换。

接下来碰到了 1。 我们有两个选择：

- 1 和 2 进行交换
- 1 和 4 进行交换

两种交换都能使得结果更大，但是 和 2 交换能够使得增值最小，也就是题目中的下一个更大的效果。因此我们 1 和 2 进行交换。



还需要继续往高位看么？不需要，因为交换高位得到的增幅一定比交换低位大，这是一个贪心的思想。

那么如何保证增幅最小呢? 其实只需要将 1 后面的数字按照从小到大进行排列即可。

注意到 1 后面的数已经是从大到小排列了（非严格递减），我们其实只需要用双指针交换即可，而不需要真正地排序。

> 1 后面的数一定是从大到小排好序了吗？当然，否则，我们找到第一个可以交换的回溯点就不是 1 了，和 1 是第一个可以交换的回溯点矛盾。因为第一个可以交换的回溯点其实就是从后往前第一个递减的值。

关键点解析

- 写几个例子通常会帮助理解问题的规律
- 在有序数组中首尾指针不断交换位置即可实现 reverse
- 找到从右边起`第一个大于nums[i]的`，并将其和 nums[i]进行交换

代码

JavaScript Code:

```
/*
 * @lc app=leetcode id=31 lang=javascript
 *
 * [31] Next Permutation
 */

function reverseRange(A, i, j) {
  while (i < j) {
    const temp = A[i];
    A[i] = A[j];
    A[j] = temp;
    i++;
    j--;
  }
}
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
 */
var nextPermutation = function (nums) {
  // 时间复杂度O(n) 空间复杂度O(1)
  if (nums == null || nums.length <= 1) return;

  let i = nums.length - 2;
  // 从后往前找到第一个降序的,相当于找到了我们的回溯点
  while (i > -1 && nums[i + 1] <= nums[i]) i--;

  // 如果找了就swap
  if (i > -1) {
    let j = nums.length - 1;
    // 找到从右边起第一个大于nums[i]的，并将其和nums[i]进行交换
    // 因为如果交换的数字比nums[i]还要小肯定不符合题意
    while (nums[j] <= nums[i]) j--;
    const temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }

  // 最后我们只需要将剩下的元素从左到右，依次填入当前最小的元素就可以保证是大于当前排列的最小值了
  // [i + 1, A.length -1]的元素进行反转

  reverseRange(nums, i + 1, nums.length - 1);
};
```

9. 搜索旋转排序数组

题目描述

给你一个升序排列的整数数组 nums ，和一个整数 target 。

假设按照升序排序的数组在预先未知的某个点上进行了旋转。（例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] ）。

请你在数组中搜索 target ，如果数组中存在这个目标值，则返回它的索引，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
nums 肯定会在某个点上旋转
-10^4 <= target <= 10^4
```

前置知识

- 数组
- 二分法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。

题目要求时间复杂度为 logn，因此基本就是二分法了。 这道题目不是直接的有序数组，不然就是 easy 了。

首先要知道，我们随便选择一个点，将数组分为前后两部分，其中一部分一定是有序的。

具体步骤：

- 我们可以先找出 mid，然后根据 mid 来判断，mid 是在有序的部分还是无序的部分

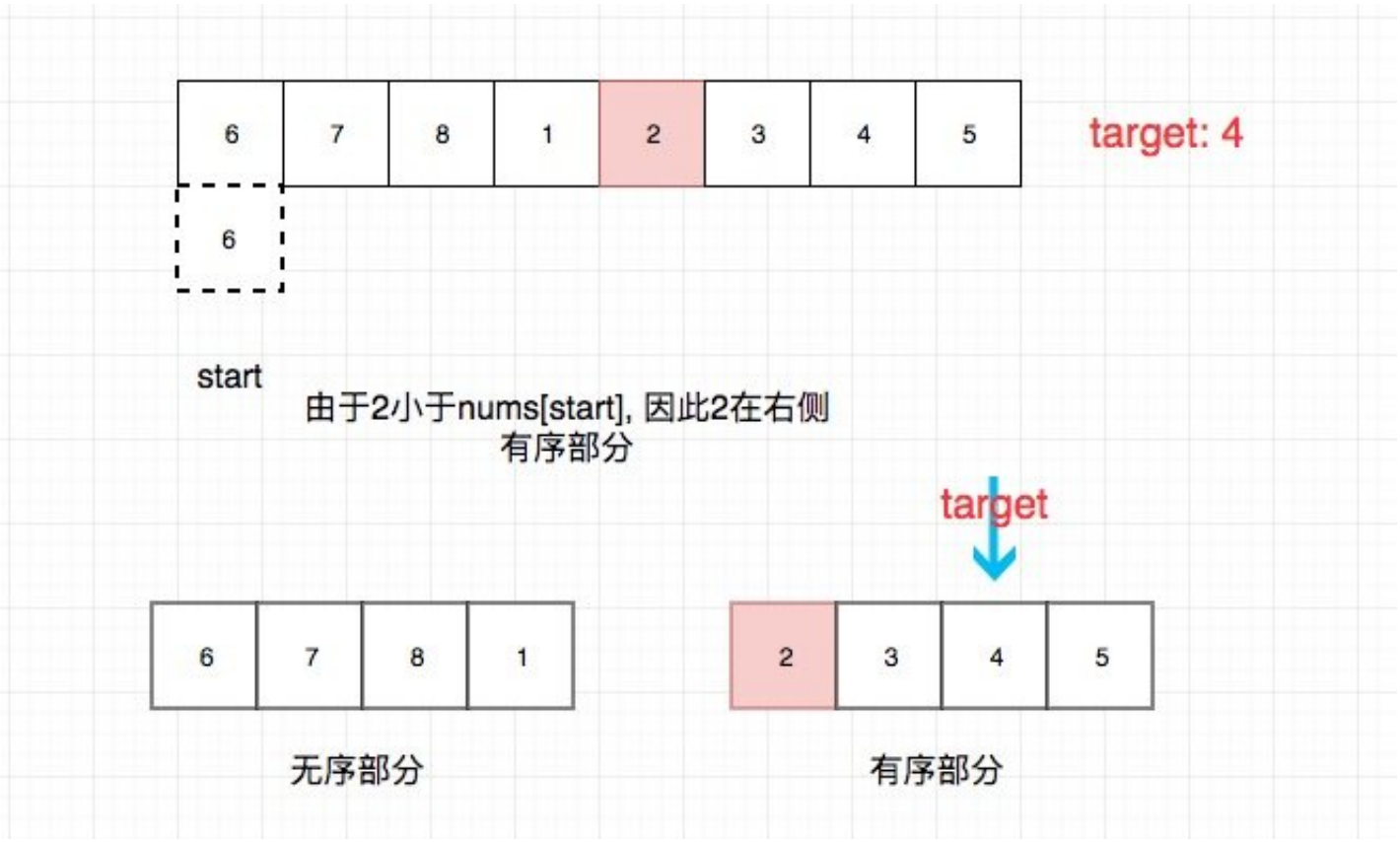
假如 mid 小于 start，则 mid 一定在右边有序部分。
假如 mid 大于等于 start， 则 mid 一定在左边有序部分。

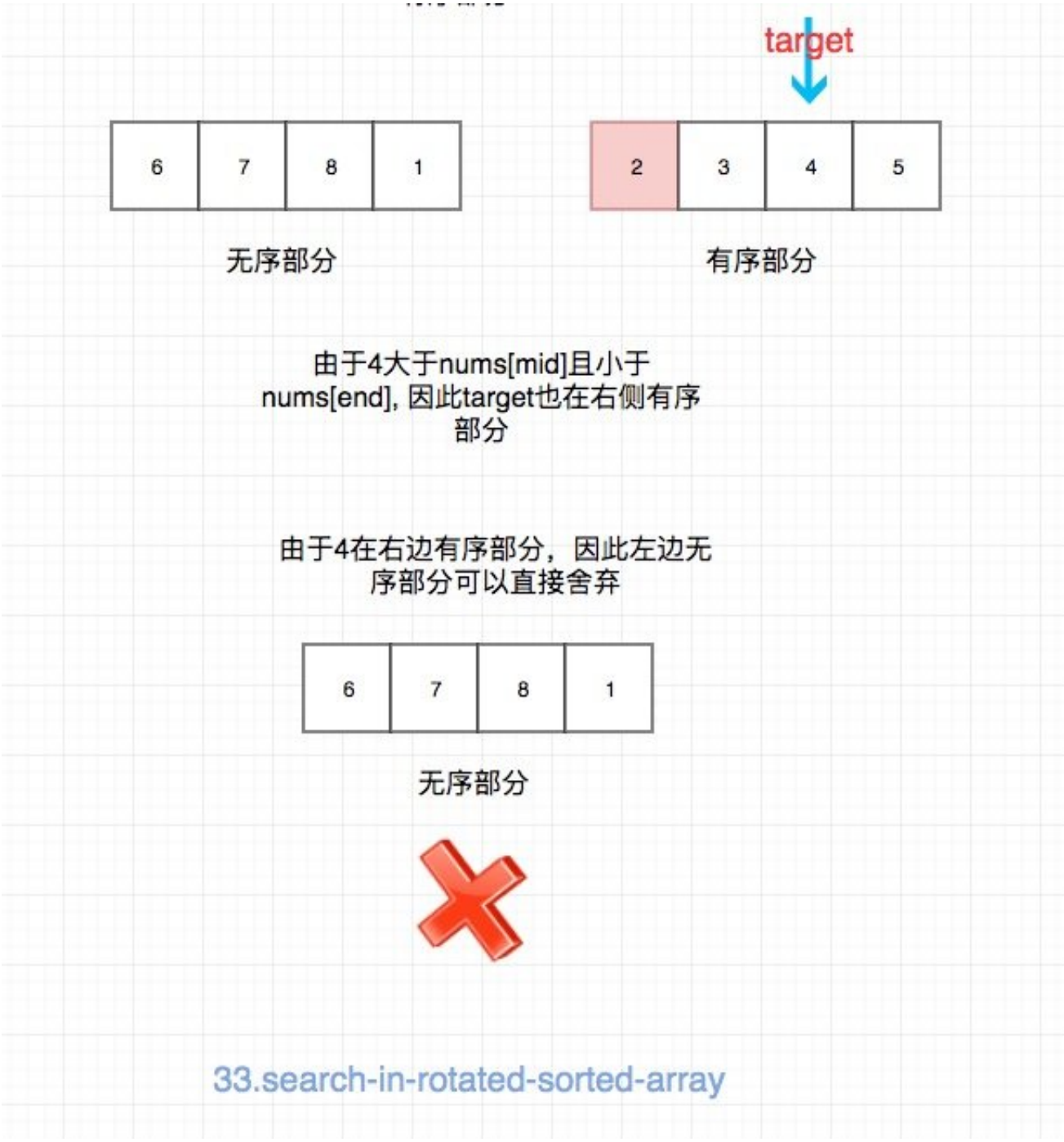
> 注意等号的考虑

- 然后我们继续判断 target 在哪一部分， 我们就可以舍弃另一部分了

我们只需要比较 target 和有序部分的边界关系就行了。 比如 mid 在右侧有序部分，即[mid, end]
那么我们只需要判断 target >= mid && target <= end 就能知道 target 在右侧有序部分，我们就可以舍弃左边部分了(start = mid + 1)， 反之亦然。

我们以([6,7,8,1,2,3,4,5], 4)为例讲解一下：


6
7
8
1
2
3
4
5
target: 4
6
start
由于2小于nums[start], 因此2在右侧
有序部分
target
6
7
8
1
2
3
4
5
无序部分
有序部分



关键点解析

- 二分法
- 找出有序区间，然后根据 target 是否在有序区间舍弃一半元素

代码

```
/*
 * @lc app=leetcode id=33 lang=javascript
 *
 * [33] Search in Rotated Sorted Array
```

```
 */
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var search = function (nums, target) {
  // 时间复杂度：O(logn)
  // 空间复杂度：O(1)
  // [6,7,8,1,2,3,4,5]
  let start = 0;
  let end = nums.length - 1;

  while (start <= end) {
    const mid = start + ((end - start) >> 1);
    if (nums[mid] === target) return mid;

    // [start, mid]有序

    //   ⚠️注意这里的等号
    if (nums[mid] >= nums[start]) {
      //target 在 [start, mid] 之间

      // 其实target不可能等于nums[mid]，  但是为了对称，我还是加上了等号
      if (target >= nums[start] && target <= nums[mid]) {
        end = mid - 1;
      } else {
        //target 不在 [start, mid] 之间
        start = mid + 1;
      }
    } else {
      // [mid, end]有序

      // target 在 [mid, end] 之间
      if (target >= nums[mid] && target <= nums[end]) {
        start = mid + 1;
      } else {
        // target 不在 [mid, end] 之间
        end = mid - 1;
      }
    }
  }

  return -1;
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

10. 组合总和

题目描述

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

前置知识

- 回溯法

公司

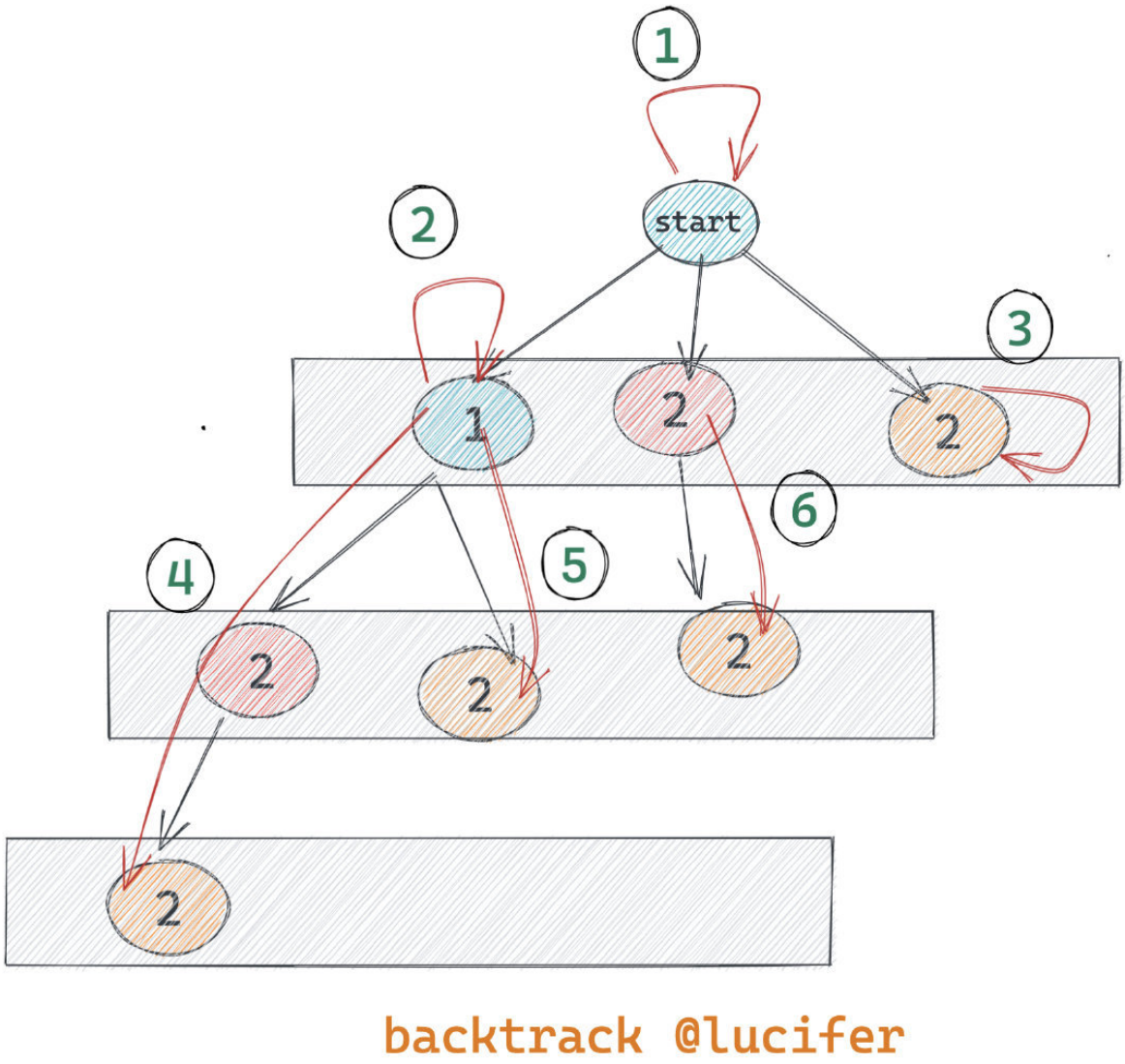
- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是`求极值`，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。网上也有大神给出了这种回溯法解题的通用写法，这里的所有的解法使用通用方法解答。
除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：



> 每一层灰色的部分，表示当前有哪些节点是可以选择的， 红色部分则是选择路径。1，2，3，4，5，6 则分别表示我们的 6 个子集。

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

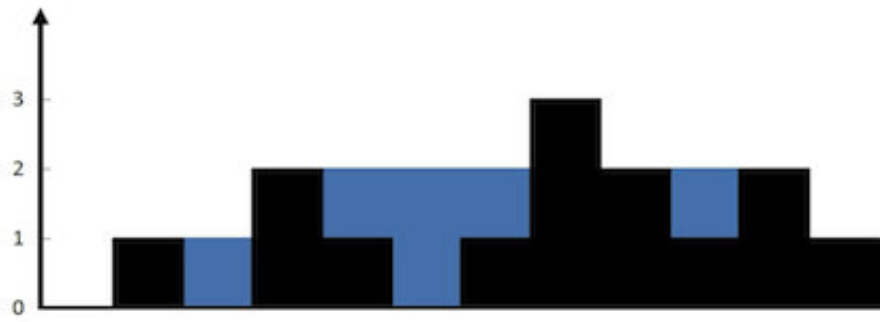
JS Code:

```
function backtrack(list, tempList, nums, remain, start) {
  if (remain < 0) return;
  else if (remain === 0) return list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i); // 数字可以重复使用,
i + 1代表不可以重复利用
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function (candidates, target) {
  const list = [];
  backtrack(
    list,
    [],
    candidates.sort((a, b) => a - b),
    target,
    0
  );
  return list;
};
```

11. 接雨水

题目描述

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。



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）。 感谢 Marcos 贡献此图。

示例:

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

前置知识

- 空间换时间
- 双指针
- 单调栈

公司

- 阿里
- 腾讯
- 百度
- 字节

双数组

思路

这是一道雨水收集的问题， 难度为 `hard`. 如图所示，让我们求下过雨之后最多可以积攒多少的水。

如果采用暴力求解的话，思路应该是 height 数组依次求和，然后相加。

- 伪代码

```
for (let i = 0; i < height.length; i++) {
  area += (h[i] - height[i]) * 1; // h为下雨之后的水位
}
```

问题转化为求 h，那么 h[i]又等于 `左右两侧柱子的最大值中的较小值`，即 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)`

如上图那么 h 为 [0, 1, 1, 2, 2, 2 ,2, 3, 2, 2, 2, 1]

问题的关键在于求解 `左边柱子最大值` 和 `右边柱子最大值`，
我们其实可以用两个数组来表示 `leftMax`，`rightMax`，
以 leftMax 为例，leftMax[i]代表 i 的左侧柱子的最大值，因此我们维护两个数组即可。

关键点解析

- 建模 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)` (h 为下雨之后的水位)

代码

JS Code:

```
/*
 * @lc app=leetcode id=42 lang=javascript
 *
 * [42] Trapping Rain Water
 *
 */
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function (height) {
  let max = 0;
  let volume = 0;
  const leftMax = [];
  const rightMax = [];

  for (let i = 0; i < height.length; i++) {
    leftMax[i] = max = Math.max(height[i], max);
  }

  max = 0;

  for (let i = height.length - 1; i >= 0; i--) {
    rightMax[i] = max = Math.max(height[i], max);
  }

  for (let i = 0; i < height.length; i++) {
    volume = volume + Math.min(leftMax[i], rightMax[i]) - height[i];
  }

  return volume;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

12. 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。

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]
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯的基本思路清参考上方的回溯专题。

以 [1,2,3] 为例，我们的逻辑是：

- 先从 [1,2,3] 选取一个数。
- 然后继续从 [1,2,3] 选取一个数，并且这个数不能是已经选取过的数。

> 如何确保这个数不能是已经选取过的数？我们可以直接在已经选取的数字中线性查找，也可以将已经选取的数字中放到 hashset 中，这样就可以在 $O(1)$ 的时间来判断是否已经被选取了，只不过需要额外的空间。

- 重复这个过程直到选取的数字个数达到了 3。

关键点解析

- 回溯法
- backtrack 解题公式

代码

Javascript Code:

```
function backtrack(list, tempList, nums) {
  if (tempList.length === nums.length) return list.push([...tempList]);
  for (let i = 0; i < nums.length; i++) {
    if (tempList.includes(nums[i])) continue;
    tempList.push(nums[i]);
    backtrack(list, tempList, nums);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permute = function (nums) {
  const list = [];
  backtrack(list, [], nums);
  return list;
};
```

复杂度分析
令 N 为数组长度。

- 时间复杂度：$O(N!)$
- 空间复杂度：$O(N)$

13.两数之和

题目描述

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

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。

示例:
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
##解题思路
对于这道题，我们很容易想到使用两层循环来解决这个问题，但是两层循环的复杂度为O（n2），我们可以考虑能否换一种思路，减小复杂度。

这里使用一个map对象来储存遍历过的数字以及对应的索引值。我们在这里使用减法进行计算
● 计算target和第一个数字的差，并记录进map对象中，其中两数差值作为key，其索引值作为value。
● 再计算第二个数字与target的差，并与map对象中的数值进行对比，若相同，直接返回，如果没有相同值，就将这个差值也存入map对象中。
● 重复第二步，直到找到目标值。

代码实现

暴力循环：
/**

- @param {number[]} nums
- @param {number} target
- @return {number[]}
  /
  var twoSum = function(nums, target) {

var len=nums.length;
for(var i=0;i<len;i++){
for(var j=0;j<len;j++){
if(nums[i]+nums[j] == target&&i!=j){
return [i,j];
}
}
}
};
使用map对象存储方法：
*/**

- @param {number[]} nums
- @param {number} target
- @return {number[]}

/
var twoSum = function(nums, target) {
const maps = {}
const len = nums.length

for(let i=0;i<len;i++) {
if(maps[target-nums[i]]!==undefined) {
return [maps[target - nums[i]], i]
}
maps[nums[i]]=i
}
};

提交结果

第二种方法的提交结果：

14.三数之和

题目描述

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 ？ 请你找出所有满足条件且不重复的三元组。
注意：答案中不可以包含重复的三元组。
示例：

解题思路

这个题和之前的两数之和完全不一样，不过这里依旧可以使用双指针来实现。
我们在使用双指针时，往往数组都是有序的，这样才能不断缩小范围，所以我们要对已知数组进行排序。
（1）首先我们设置一个固定的数，然后在设置两个指针，左指针指向固定的数的后面那个值，右指针指向最后一个值，两个指针相向而行。
（2）每移动一次指针位置，就计算一下这两个指针与固定值的和是否为0，如果是，那么我们就得到一组符合条件的数组，如果不是0，就有一下两种情况：
相加之和大于0，说明右侧值大了，右指针左移
相加之和小于0，说明左侧值小了，左指针右移
（3）按照上边的步骤，将前len-2个值依次作为固定值，最终得到想要的结果。
因为我们需要三个值的和，所以我们无需最后两个值作为固定值，他们后面已经没有三个值可以进行计算了。

代码实现

JavaScript
复制代码
/**

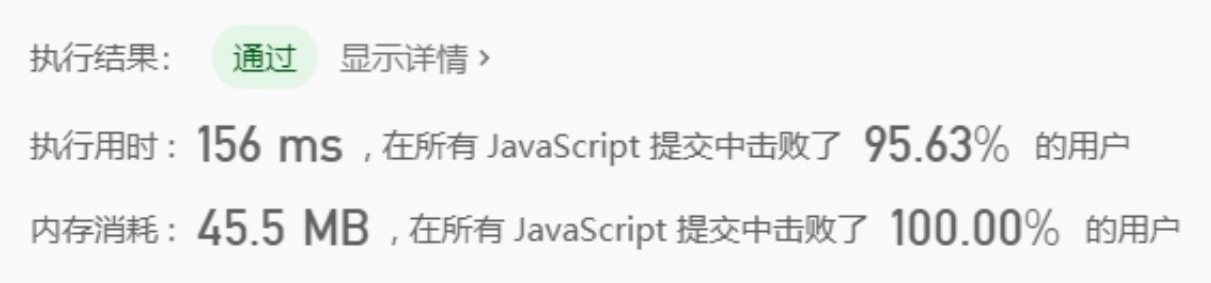
- @param {number[]} nums
- @return {number[][]}
  */
  var threeSum = function(nums) {
  let res = []
  let sum = 0
  // 将数组元素排序

```
nums.sort((a,b) => {
return a-b
})

const len =nums.length

for(let i =0; i<len-2; i++){
let j = i+1
let k = len-1
// 如果有重复数字就跳过
if(i>0&& nums[i]===nums[i-1]){
continue
}
while(j<k){
// 三数之和小于0，左指针右移
if(nums[i]+nums[j]+nums[k]<0){
j++
// 处理左指针元素重复的情况
while(j<k&&nums[j]===nums[j-1]){
j++
}
// 三数之和大于0，右指针左移
}else if(nums[i]+nums[j]+nums[k]>0){
k--
// 处理右指针元素重复的情况
while(j<k&&nums[k]===nums[k+1]){
k--
}
}else{
// 储存符合条件的结果
res.push([nums[i],nums[j],nums[k]])
j++
k--
```

```
        while(j<k&&nums[j]===nums[j-1]){
            j++
        }
        while(j<k&&nums[k]===nums[k+1]){
            k--
        }
    }
}
```

```
}
return res
};
```

提交结果



15.四数之和

题目描述

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。
注意：答案中不可以包含重复的四元组。
示例：

给定数组 nums = [1, 0, -1, 0, -2, 2]，和 target = 0。

满足要求的四元组集合为：
[
[-1, 0, 0, 1],
[-2, -1, 1, 2],
[-2, 0, 0, 2]
]

解题思路

这个题实际上和三数之和类似，我们也使用双指针来解决。
在三数之和中，使用两个指针分别指向两个元素，左指针指向固定数后面的数，右指针指向最后一个数。在固定一个数，进行遍历。左指针不断向右移动，右指针不断向左移动，直至遍历完所有的数字。

在四数之和中，我们可以固定两个数字，然后再初始化两个指针，左指针指向固定数之后的数字，右指针指向最后一个数字。两层循环进行遍历，直至遍历完所有的结果。
需要注意的是，当使双指针的时候，往往需要对数组元素进行排序。

代码实现

/**

- @param {number[]} nums
- @param {number} target
- @return {number[][]}
  */
  var fourSum = function(nums, target) {
  const res = []
  if(nums.length < 4){
  return []
  }
  nums.sort((a, b) => a - b)
  for(let i = 0; i < nums.length - 3; i++){
  if(i > 0 && nums[i] === nums[i - 1]){
  continue
  }
  if(nums[i] + nums[i +1] + nums[i + 2] + nums[i + 3] > target){
  break
  }

```
for(let j = i + 1; j < nums.length -2; j++){
    // 若与已遍历过的数字相同，就跳过，避免结果中出现重复的数组
    if(j > i + 1 && nums[j] === nums[j - 1]){
        continue
    }
    let left = j + 1, right = nums.length - 1

    while(left < right){
        const sum = nums[i] + nums[j] +nums[left] + nums[right]
        if(sum === target){
            res.push([nums[i], nums[j], nums[left], nums[right]])
        }
        if(sum <= target){
            left ++
            while(nums[left]  === nums[left - 1]) {
```

```
                left ++
            }
        }else{
            right --
            while(nums[right] === nums[right + 1]){
                right --
            }
        }
    }
}
```

}
return res

};

提交结果

执行结果： 通过 显示详情 >

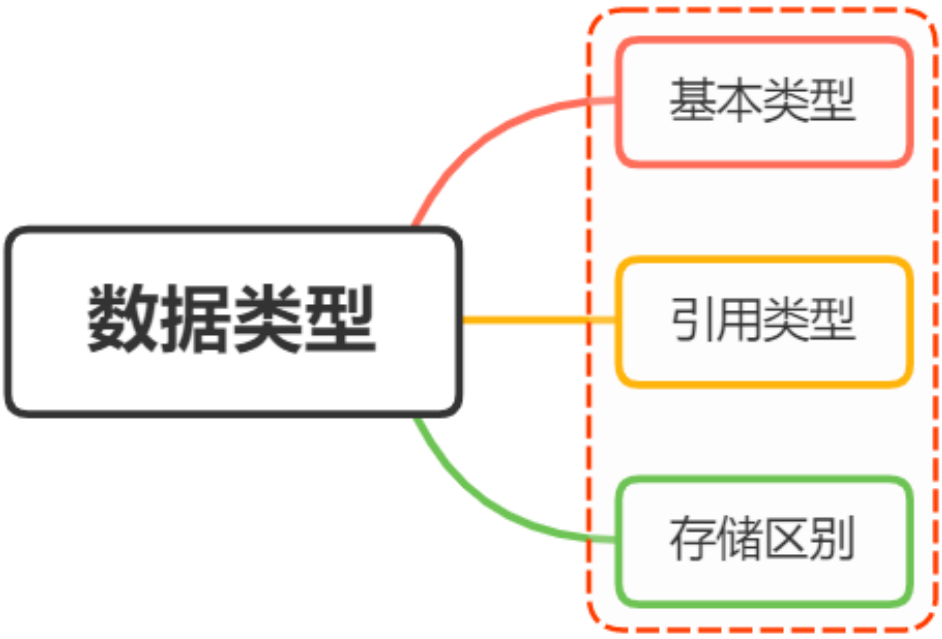
执行用时： 108 ms ，在所有 JavaScript 提交中击败了 72.14% 的用户

内存消耗： 38.9 MB ，在所有 JavaScript 提交中击败了 51.05% 的用户

更多精彩内容，持续汇总中.....

JavaScript面试真题（35题）

1. 说说JavaScript中的数据类型？存储上的差别？



1.1. 前言

在 `JavaScript` 中，我们可以分成两种类型：

- 基本类型
- 复杂类型

两种类型的区别是：存储位置不同

1.2. 基本类型

基本类型主要为以下6种：

- Number
- String
- Boolean
- Undefined
- null

- symbol

1.2.1. Number

数值最常见的整数类型格式则为十进制，还可以设置八进制（零开头）、十六进制（0x开头）

```JavaScript
let intNum = 55 // 10进制的55
let num1 = 070 // 8进制的56
let hexNum1 = 0xA //16进制的10
```

浮点类型则在数值汇总必须包含小数点，还可通过科学计数法表示

```JavaScript
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum3 = .1; // 有效，但不推荐
let floatNum = 3.125e7; // 等于 31250000
```

在数值类型中，存在一个特殊数值 `NaN` ，意为“不是数值”，用于表示本来要返回数值的操作失败了（而不是抛出错误）

```JavaScript
console.log(0/0); // NaN
console.log(-0/+0); // NaN
```

1.2.2. Undefined

`Undefined` 类型只有一个值，就是特殊值 `undefined` 。当使用 `var` 或 `let` 声明了变量但没有初始化时，就相当于给变量赋予了 `undefined` 值

```JavaScript
let message;
console.log(message == undefined); // true
```

包含 `undefined` 值的变量跟未定义变量是有区别的

```javascript
let message; // 这个变量被声明了，只是值为 undefined

console.log(message); // "undefined"
console.log(age); // 没有声明过这个变量，报错
```

1.2.3. String

字符串可以使用双引号（"）、单引号（'）或反引号（`）标示

```javascript
let firstName = "John";
let lastName = 'Jacob';
let lastName = `Jingleheimerschmidt`
```

字符串是不可变的，意思是一旦创建，它们的值就不能变了

```javascript
let lang = "Java";
lang = lang + "Script";  // 先销毁再创建
```

1.2.4. Null

`Null` 类型同样只有一个值，即特殊值 `null`

逻辑上讲， null 值表示一个空对象指针，这也是给 `typeof` 传一个 `null` 会返回 `"object"` 的原因

```javascript
let car = null;
console.log(typeof car); // "object"
```

`undefined` 值是由 `null` 值派生而来

```javascript
console.log(null == undefined); // true
```

只要变量要保存对象，而当时又没有那个对象可保存，就可用 `null` 来填充该变量

1.2.5. Boolean

`Boolean` （布尔值）类型有两个字面值： `true` 和 `false`

通过 `Boolean` 可以将其他类型的数据转化成布尔值

规则如下：

JavaScript 复制代码

数据类型	转换为 true 的值	转换为 false 的值
String	非空字符串	""
Number	非零数值（包括无穷值）	0 、 NaN
Object	任意对象	null
Undefined	N/A （不存在）	undefined

1.2.6. Symbol

Symbol （符号）是原始值，且符号实例是唯一、不可变的。符号的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险

JavaScript 复制代码

```
let genericSymbol = Symbol();
let otherGenericSymbol = Symbol();
console.log(genericSymbol == otherGenericSymbol); // false

let fooSymbol = Symbol('foo');
let otherFooSymbol = Symbol('foo');
console.log(fooSymbol == otherFooSymbol); // false
```

1.3. 引用类型

复杂类型统称为 `Object` ，我们这里主要讲述下面三种：

- Object
- Array
- Function

1.3.1. Object

创建 `object` 常用方式为对象字面量表示法，属性名可以是字符串或数值

```JavaScript
let person = {
    name: "Nicholas",
    "age": 29,
    5: true
};
```

1.3.2. Array

`JavaScript` 数组是一组有序的数据，但跟其他语言不同的是，数组中每个槽位可以存储任意类型的数据。并且，数组也是动态大小的，会随着数据添加而自动增长

```JavaScript
let colors = ["red", 2, {age: 20 }]
colors.push(2)
```

1.3.3. Function

函数实际上是对象，每个函数都是 `Function` 类型的实例，而 `Function` 也有属性和方法，跟其他引用类型一样

函数存在三种常见的表达方式：

- 函数声明

```JavaScript
// 函数声明
function sum (num1, num2) {
    return num1 + num2;
}
```

- 函数表达式

```JavaScript
let sum = function(num1, num2) {
    return num1 + num2;
};
```

- 箭头函数

函数声明和函数表达式两种方式

```JavaScript
let sum = (num1, num2) => {
    return num1 + num2;
};
```

1.3.4. 其他引用类型

除了上述说的三种之外，还包括 `Date` 、 `RegExp` 、 `Map` 、 `Set` 等……

1.4. 存储区别

基本数据类型和引用数据类型存储在内存中的位置不同：

- 基本数据类型存储在栈中
- 引用类型的对象存储于堆中

当我们把变量赋值给一个变量时，解析器首先要确认的就是这个值是基本类型值还是引用类型值

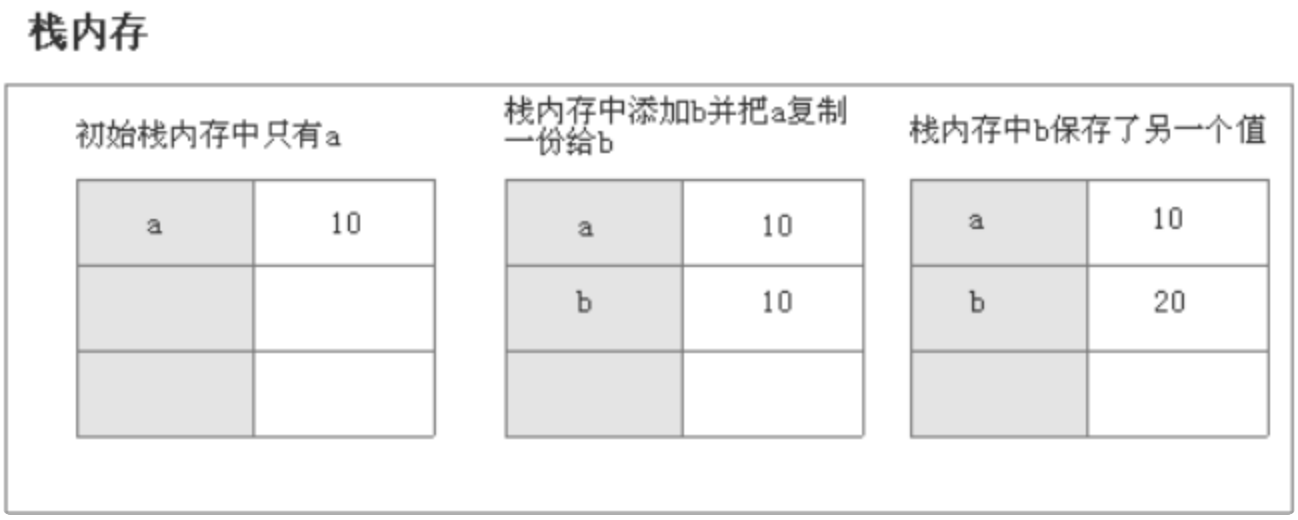
下面来举个例子

1.4.1. 基本类型

```JavaScript
let a = 10;
let b = a; // 赋值操作
b = 20;
console.log(a); // 10值
```

`a` 的值为一个基本类型，是存储在栈中，将 `a` 的值赋给 `b` ，虽然两个变量的值相等，但是两个变量保存了两个不同的内存地址

下图演示了基本类型赋值的过程：



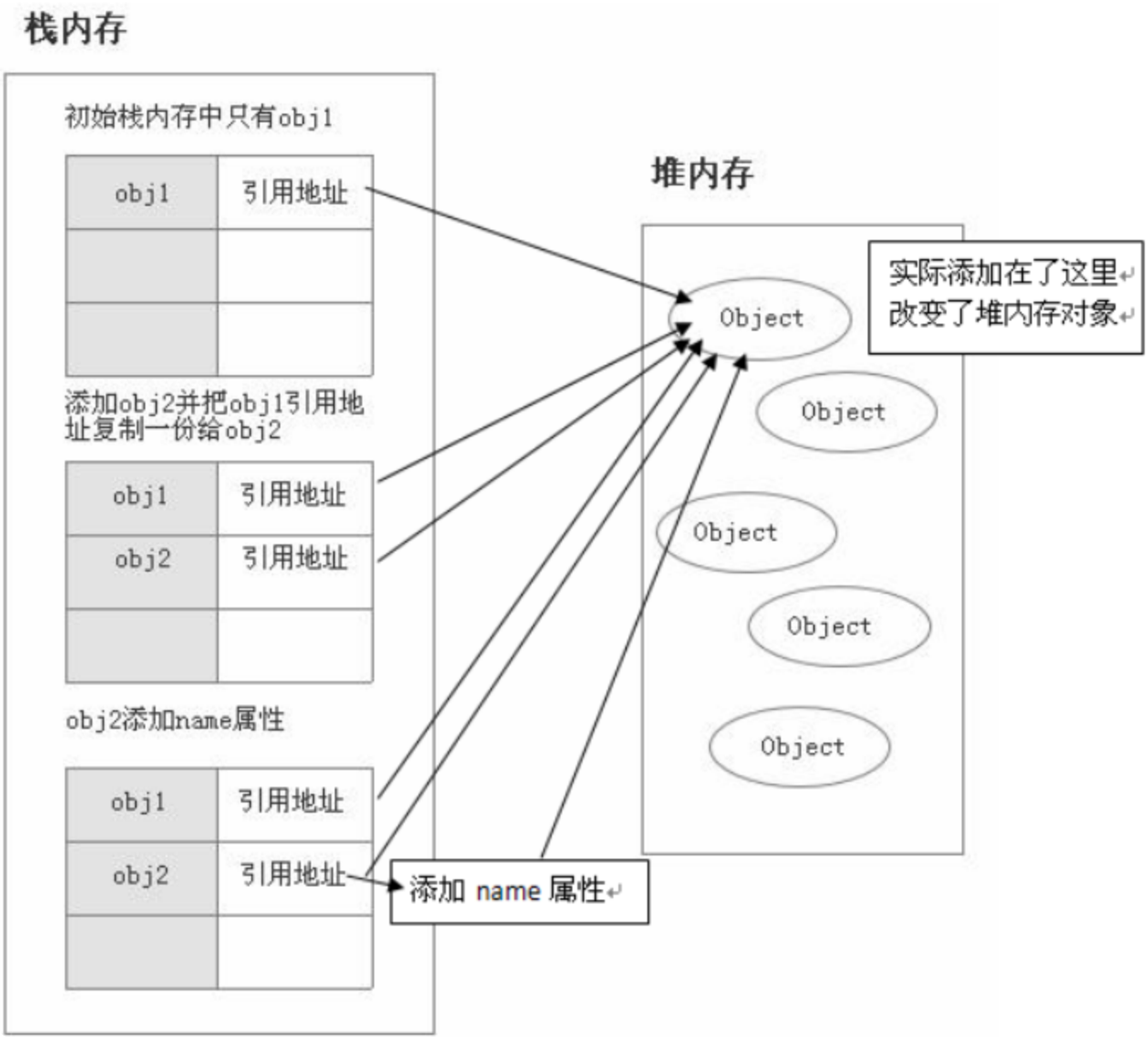
1.4.2. 引用类型

```
1  var obj1 = {}
2  var obj2 = obj1;
3  obj2.name = "Xxx";
4  console.log(obj1.name); // xxx
```

引用类型数据存放在堆中，每个堆内存对象都有对应的引用地址指向它，引用地址存放在栈中。

`obj1` 是一个引用类型，在赋值操作过程汇总，实际是将堆内存对象在栈内存的引用地址复制了一份给了 `obj2` ，实际上他们共同指向了同一个堆内存对象，所以更改 `obj2` 会对 `obj1` 产生影响

下图演示这个引用类型赋值过程



1.5. 小结

- 声明变量时不同的内存地址分配：
  - 简单类型的值存放在栈中，在栈中存放的是对应的值
  - 引用类型对应的值存储在堆中，在栈中存放的是指向堆内存的地址
- 不同的类型数据导致赋值变量时的不同：
  - 简单类型赋值，是生成相同的值，两个对象对应不同的地址
  - 复杂类型赋值，是将保存对象的内存地址赋值给另一个变量。也就是两个变量指向堆内存中同一个对象

2. 说说你了解的js数据结构?

2.1. 什么是数据结构?

数据结构是计算机存储、组织数据的方式。
数据结构意味着接口或封装：一个数据结构可被视为两个函数之间的接口，或者是由数据类型联合组成的存储内容的访问方法封装。

我们每天的编码中都会用到数据结构
数组是最简单的内存数据结构
下面是常见的数据结构：

1. 数组（Array）
2. 栈（Stack）
3. 队列（Queue）
4. 链表（Linked List）
5. 字典
6. 散列表（Hash table）
7. 树（Tree）
8. 图（Graph）
9. 堆（Heap）

2.2. 数组（Array）

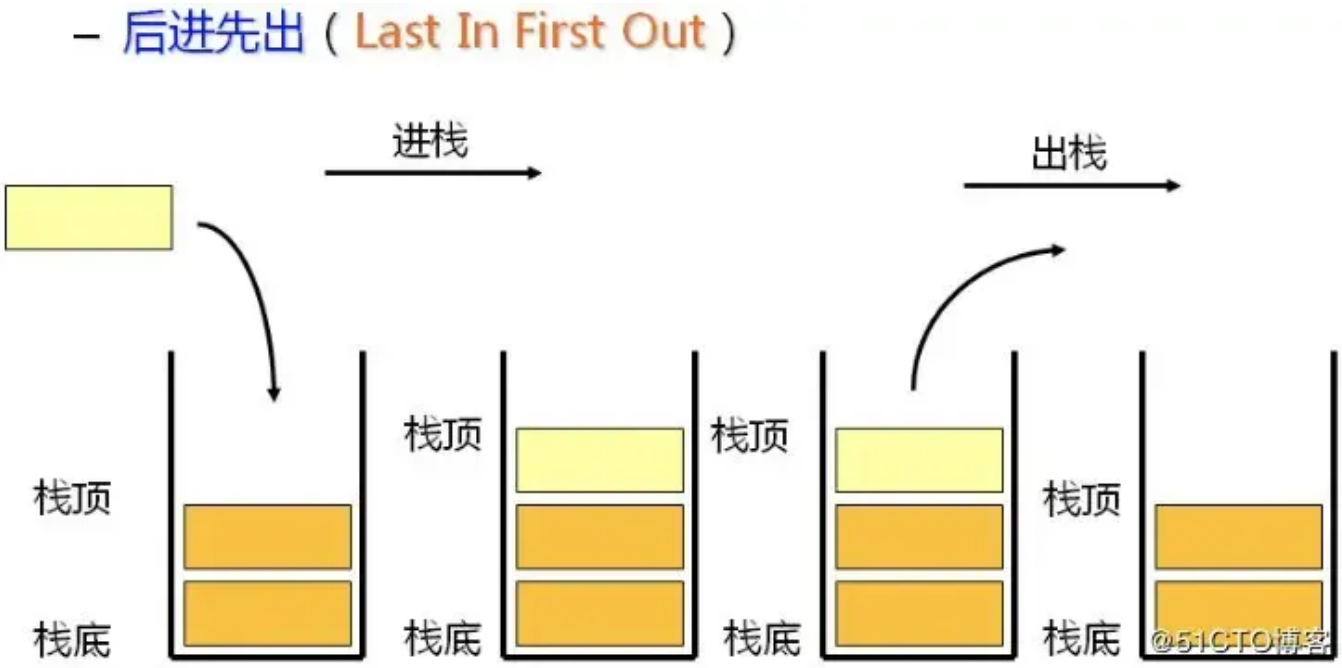
数组是最最基本的数据结构，很多语言都内置支持数组。
数组是使用一块连续的内存空间保存数据，保存的数据的个数在分配内存的时候就是确定的。

在日常生活中，人们经常使用列表：待办事项列表、购物清单等。

而计算机程序也在使用列表，在下面的条件下，选择列表作为数据结构就显得尤为有用：
数据结构较为简单
不需要在一个长序列中查找元素，或者对其进行排序
反之，如果数据结构非常复杂，列表的作用就没有那么大了。

2.3. 栈（Stack）

栈是一种遵循后进先出（LIFO）原则的有序集合
在栈里，新元素都接近栈顶，旧元素都接近栈底。
每次加入新的元素和拿走元素都在顶部操作



2.4. 队列（Queue）

队列是遵循先进先出（FIFO，也称为先来先服务）原则的一组有序的项
队列在尾部添加新元素，并从顶部移除元素