1. 合并两个有序链表

题目描述

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

前置知识

- 递归
- 链表

思路

本题可以使用递归来解，将两个链表头部较小的一个与剩下的元素合并，并返回排好序的链表头，当两条链表中的一条为空时终止递归。

关键点

- 掌握链表数据结构
- 考虑边界情况

代码

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
```

```
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
const mergeTwoLists = function (l1, l2) {
  if (l1 === null) {
    return l2;
  }
  if (l2 === null) {
    return l1;
  }
  if (l1.val < l2.val) {
    l1.next = mergeTwoLists(l1.next, l2);
    return l1;
  } else {
    l2.next = mergeTwoLists(l1, l2.next);
    return l2;
  }
};
```

复杂度分析

M、N 是两条链表 l1、l2 的长度

- 时间复杂度：$O(M+N)$
- 空间复杂度：$O(M+N)$

扩展

- 你可以使用迭代的方式求解么?

迭代的 CPP 代码如下：

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode head, *tail = &head;
        while (a && b) {
            if (a->val <= b->val) {
                tail->next = a;
                a = a->next;
            } else {
                tail->next = b;
```

```
                b = b->next;
            }
            tail = tail->next;
        }
        tail->next = a ? a : b;
        return head.next;
    }
};
```

迭代的 JS 代码如下：

```
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;
  }
  prev.next = l1 === null ? l2 : l1;

  return prehead.next;
};
```

2. 括号生成

题目描述

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：

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

```
    "(()())",
    "(())()"
  ]
```


前置知识

- DFS
- 回溯法

思路

本题是 20. 有效括号 的升级版。

由于我们需要求解所有的可能， 因此回溯就不难想到。回溯的思路和写法相对比较固定，并且回溯的优化手段大多是剪枝。

不难想到， 如果左括号的数目小于右括号，我们可以提前退出，这就是这道题的剪枝。 比如 `())....`，后面就不用看了，直接退出即可。回溯的退出条件也不难想到，那就是：

- 左括号数目等于右括号数目
- 左括号数目 + 右括号数目 = 2 * n

由于我们需要剪枝， 因此必须从左开始遍历。（WHY？）

因此这道题我们可以使用深度优先搜索(回溯思想)，从空字符串开始构造，做加法， 即 dfs(左括号数, 右括号数目, 路径)， 我们从 dfs(0, 0, '') 开始。

伪代码：

```
res = []
def dfs(l, r, s):
   if l > n or r > n: return
   if (l == r == n): res.append(s)
   # 剪枝，提高算法效率
   if l < r: return
   # 加一个左括号
   dfs(l + 1, r, s + '(')
   # 加一个右括号
   dfs(l, r + 1, s + ')')
dfs(0, 0, '')
return res
```

由于字符串的不可变性， 因此我们无需`撤销 s 的选择`。但是当你使用 C++ 等语言的时候， 就需要注意撤销 s 的选择了。类似：


```
s.push_back(')');
dfs(l, r + 1, s);
s.pop_back();
```

关键点

- 当 l < r 时记得剪枝

代码

JS Code:

```
/**
 * @param {number} n
 * @return {string[]}
 * @param l 左括号已经用了几个
 * @param r 右括号已经用了几个
 * @param str 当前递归得到的拼接字符串结果
 * @param res 结果集
 */
const generateParenthesis = function (n) {
  const res = [];

  function dfs(l, r, str) {
    if (l == n && r == n) {
      return res.push(str);
    }
    // l 小于 r 时不满足条件 剪枝
    if (l < r) {
      return;
    }
    // l 小于 n 时可以插入左括号，最多可以插入 n 个
    if (l < n) {
      dfs(l + 1, r, str + "(");
    }
    // r < l 时 可以插入右括号
    if (r < l) {
      dfs(l, r + 1, str + ")");
    }
  }
```

```
  dfs(0, 0, "");
  return res;
};
```


复杂度分析

- 时间复杂度：O(2^N)
- 空间复杂度：O(2^N)

3. 合并 K 个排序链表

题目描述

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例：

```
输入：
[
  1->4->5,
  1->3->4,
  2->6
]
输出：1->1->2->3->4->4->5->6
```

前置知识

- 链表
- 归并排序

思路

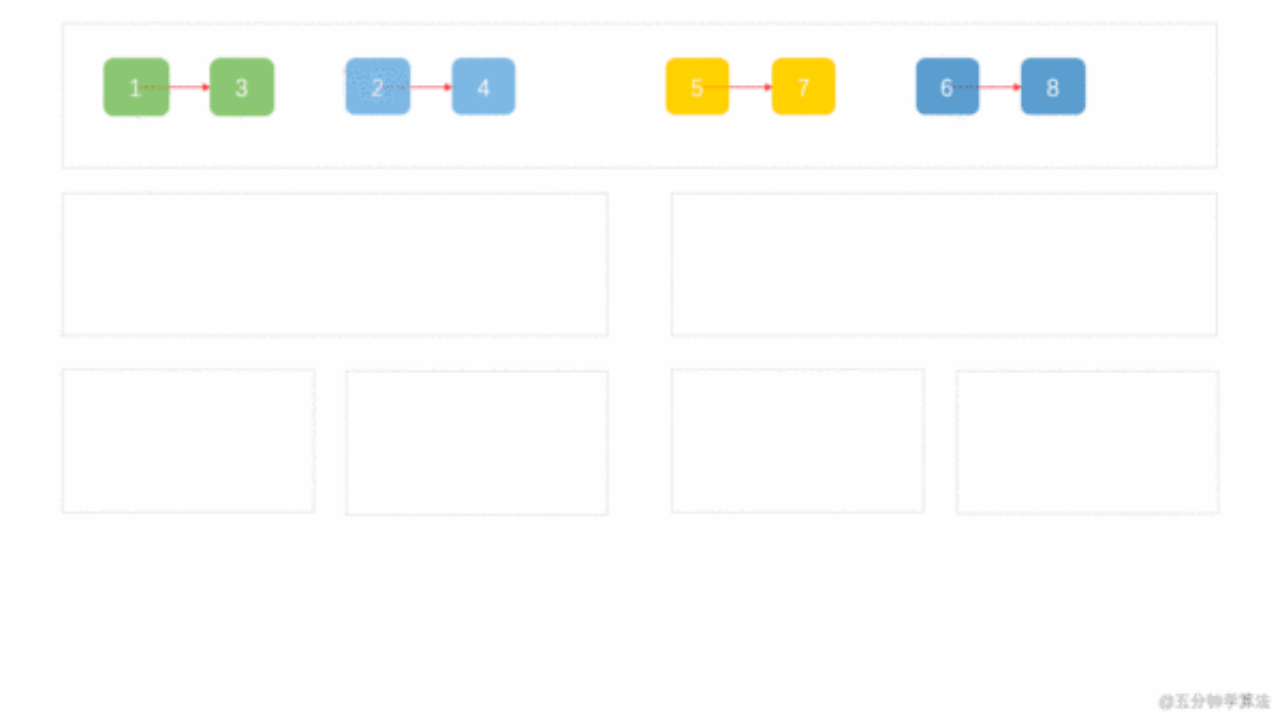
这道题目是合并 k 个已排序的链表，号称 leetcode 目前`最难`的链表题。 和之前我们解决的88.merge-sorted-array很像。
他们有两点区别：

1. 这道题的数据结构是链表，那道是数组。这个其实不复杂，毕竟都是线性的数据结构。

2. 这道题需要合并 k 个元素，那道则只需要合并两个。这个是两题的关键差别，也是这道题难度为 `hard` 的原因。

因此我们可以看出，这道题目是 `88.merge-sorted-array` 的进阶版本。其实思路也有点像，我们来具体分析下第二条。
如果你熟悉合并排序的话，你会发现它就是 `合并排序的一部分` 。

具体我们可以来看一个动画



（动画来自 https://zhuanlan.zhihu.com/p/61796021 ）

关键点解析

- 分治
- 归并排序(merge sort)

代码

JavaScript Code：

```
/*
 * @lc app=leetcode id=23 lang=javascript
 *
 * [23] Merge k Sorted Lists
 *
 * https://leetcode.com/problems/merge-k-sorted-lists/description/
```

```
 *
 */
function mergeTwoLists(l1, l2) {
  const dummyHead = {};
  let current = dummyHead;
  // l1: 1 -> 3 -> 5
  // l2: 2 -> 4 -> 6
  while (l1 !== null && l2 !== null) {
    if (l1.val < l2.val) {
      current.next = l1; // 把小的添加到结果链表
      current = current.next; // 移动结果链表的指针
      l1 = l1.next; // 移动小的那个链表的指针
    } else {
      current.next = l2;
      current = current.next;
      l2 = l2.next;
    }
  }

  if (l1 === null) {
    current.next = l2;
  } else {
    current.next = l1;
  }
  return dummyHead.next;
}
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function (lists) {
  // 图参考：  https://zhuanlan.zhihu.com/p/61796021
  if (lists.length === 0) return null;
  if (lists.length === 1) return lists[0];
  if (lists.length === 2) {
    return mergeTwoLists(lists[0], lists[1]);
  }

  const mid = lists.length >> 1;
  const l1 = [];
  for (let i = 0; i < mid; i++) {
```

```
    l1[i] = lists[i];
  }

  const l2 = [];
  for (let i = mid, j = 0; i < lists.length; i++, j++) {
    l2[j] = lists[i];
  }

  return mergeTwoLists(mergeKLists(l1), mergeKLists(l2));
};
```


复杂度分析

- 时间复杂度：$O(kn * logk)$
- 空间复杂度：$O(logk)$

相关题目

- 88.merge-sorted-array

扩展

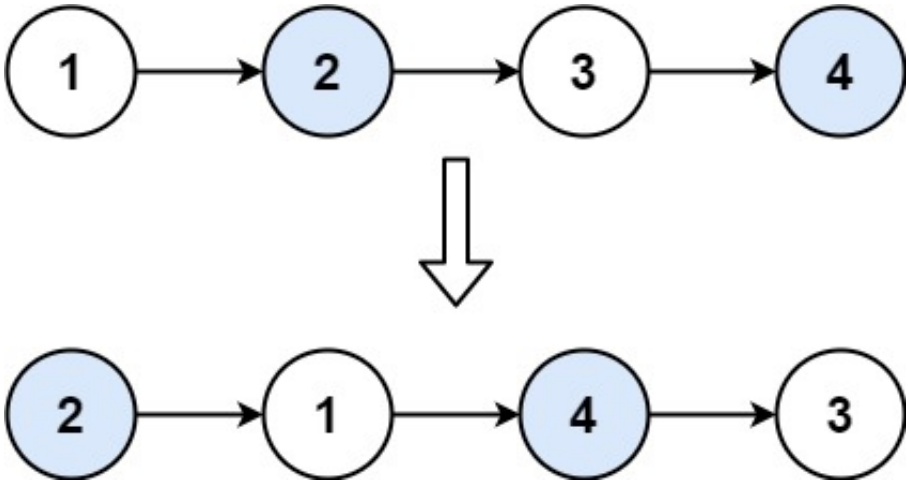
这道题其实可以用堆来做，感兴趣的同学尝试一下吧。

4. 两两交换链表中的节点

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。




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

示例 2:
输入: head = []
输出: []

示例 3:
输入: head = [1]
输出: [1]

提示:
链表中节点的数目在范围 [0, 100] 内
0 <= Node.val <= 100
```

前置知识

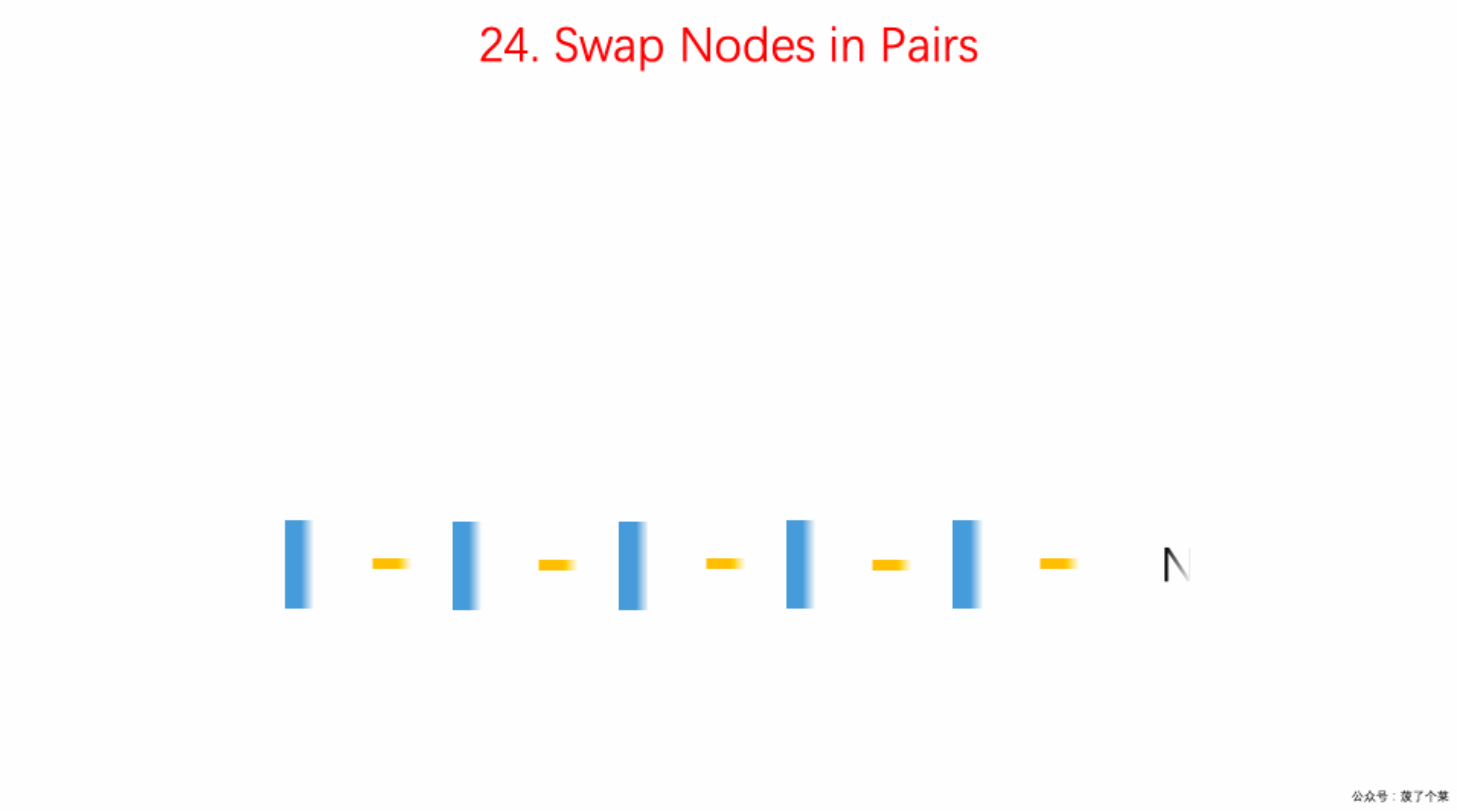
- 链表

思路

设置一个 dummy 节点简化操作，dummy next 指向 head。

1. 初始化 first 为第一个节点
2. 初始化 second 为第二个节点
3. 初始化 current 为 dummy
4. first.next = second.next
5. second.next = first
6. current.next = second

7. current 移动两格
8. 重复



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

1. 链表这种数据结构的特点和使用
2. dummyHead 简化操作

代码

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
```

```
 * @return {ListNode}
 */
var swapPairs = function (head) {
  const dummy = new ListNode(0);
  dummy.next = head;
  let current = dummy;
  while (current.next != null && current.next.next != null) {
    // 初始化双指针
    const first = current.next;
    const second = current.next.next;

    // 更新双指针和 current 指针
    first.next = second.next;
    second.next = first;
    current.next = second;

    // 更新指针
    current = current.next.next;
  }
  return dummy.next;
};
```

5. K 个一组翻转链表

题目描述

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给你这个链表：1->2->3->4->5

当 k = 2 时，应当返回：2->1->4->3->5

当 k = 3 时，应当返回：3->2->1->4->5

说明：

你的算法只能使用常数的额外空间。
你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

前置知识

- 链表

思路

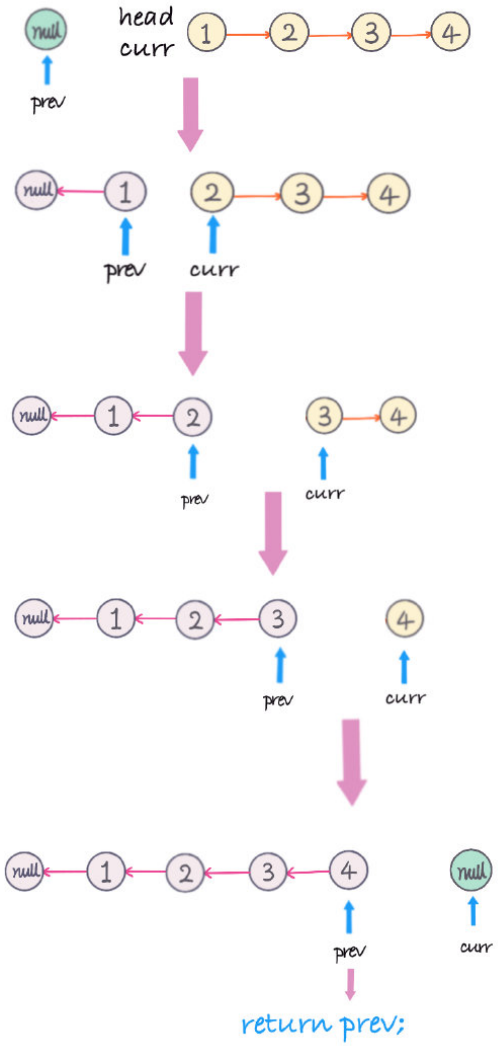
题意是以 `k` 个 nodes 为一组进行翻转，返回翻转后的`linked list`.

从左往右扫描一遍`linked list`，扫描过程中，以 k 为单位把数组分成若干段，对每一段进行翻转。给定首尾 nodes，如何对链表进行翻转。

链表的翻转过程，初始化一个为`null`的`previous node (prev)`，然后遍历链表的同时，当前`node (curr)`的下一个（next）指向前一个`node (prev)`，
在改变当前 node 的指向之前，用一个临时变量记录当前 node 的下一个`node (curr.next)`. 即

```
ListNode temp = curr.next;
curr.next = prev;
prev = curr;
curr = temp;
```

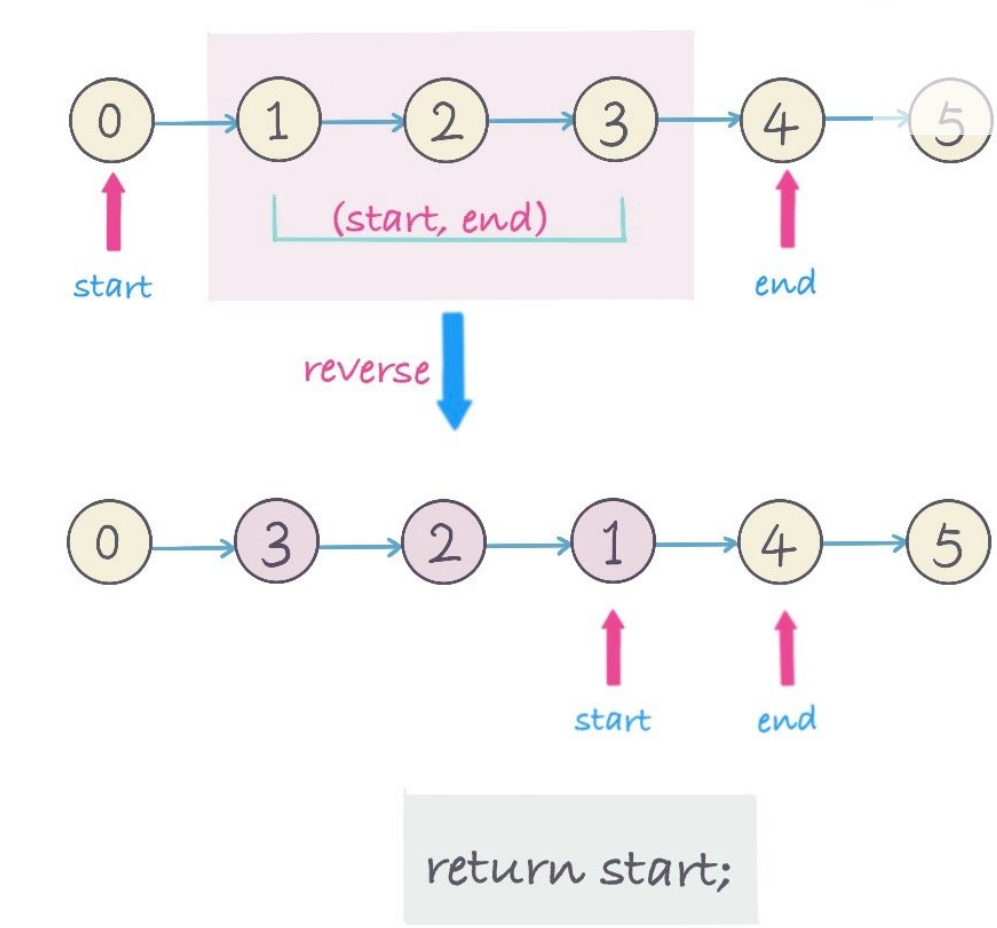
举例如图：翻转整个链表 `1->2->3->4->null` -> `4->3->2->1->null`



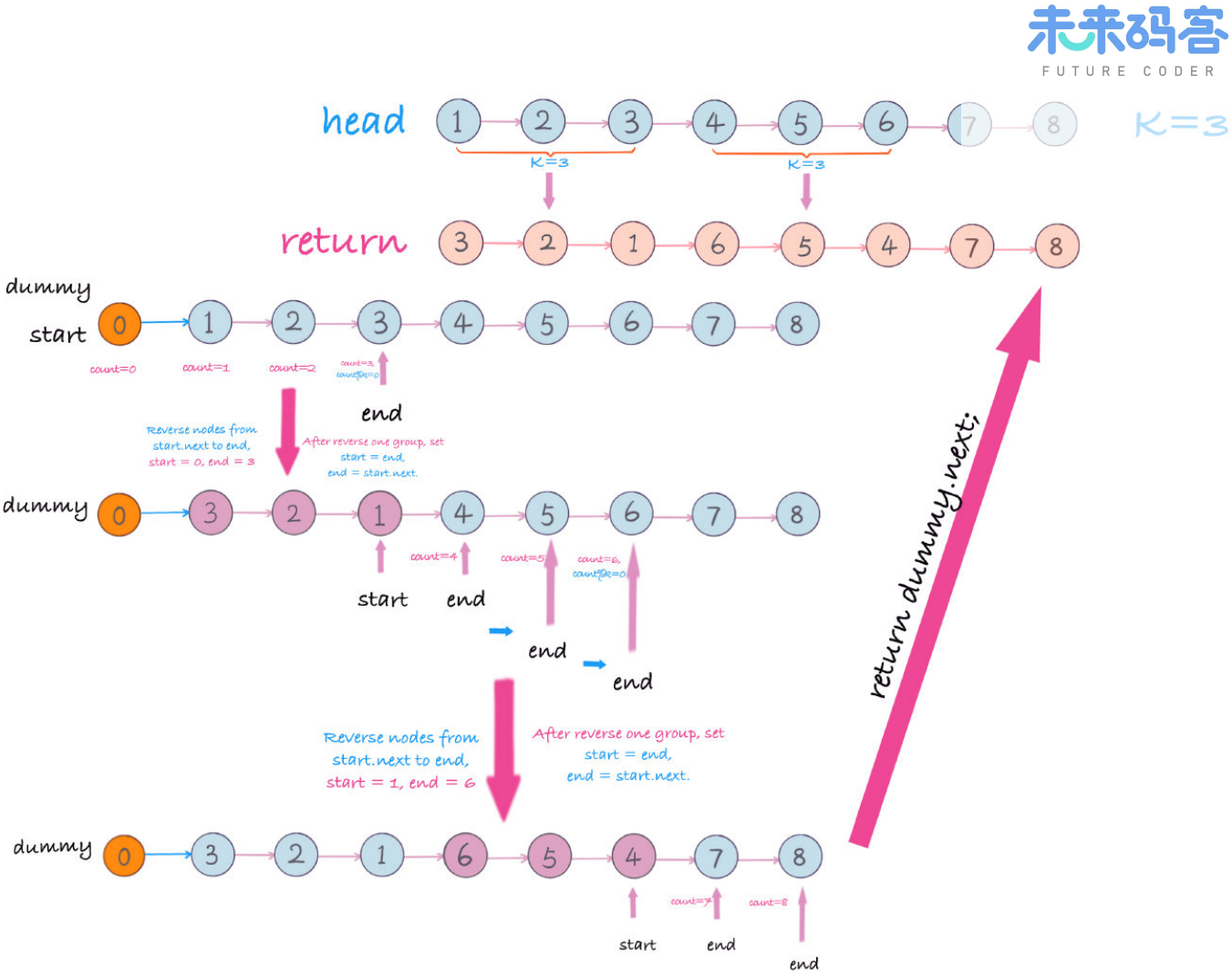
这里是对每一组（`k个nodes`）进行翻转，

1. 先分组，用一个`count`变量记录当前节点的个数
2. 用一个`start`变量记录当前分组的起始节点位置的前一个节点
3. 用一个`end`变量记录要翻转的最后一个节点位置
4. 翻转一组（`k个nodes`）即`(start, end) - start and end exclusively`。
5. 翻转后，`start`指向翻转后链表, 区间`(start, end)`中的最后一个节点, 返回`start`节点。
6. 如果不需要翻转，`end`就往后移动一个（`end=end.next`），每一次移动，都要`count+1`.

如图所示 步骤 4 和 5： 翻转区间链表区间`(start, end)`



举例如图，`head=[1,2,3,4,5,6,7,8], k = 3`



> **NOTE**: 一般情况下对链表的操作，都有可能会引入一个新的`dummy node`，因为`head`有可能会改变。这里`head 从1->3`，
> `dummy (List(0))`保持不变。

复杂度分析

- *时间复杂度:* `O(n) - n is number of Linked List`
- *空间复杂度:* `O(1)`

关键点分析

1. 创建一个 dummy node
2. 对链表以 k 为单位进行分组，记录每一组的起始和最后节点位置
3. 对每一组进行翻转，更换起始和最后的位置
4. 返回`dummy.next`.

代码

javascript code

```
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function (head, k) {
  // 标兵
  let dummy = new ListNode();
  dummy.next = head;
  let [start, end] = [dummy, dummy.next];
  let count = 0;
  while (end) {
    count++;
    if (count % k === 0) {
      start = reverseList(start, end.next);
      end = start.next;
    } else {
      end = end.next;
    }
  }
  return dummy.next;

  // 翻转stat -> end的链表
  function reverseList(start, end) {
    let [pre, cur] = [start, start.next];
    const first = cur;
    while (cur !== end) {
      let next = cur.next;
      cur.next = pre;
      pre = cur;
      cur = next;
    }
    start.next = pre;
    first.next = cur;
    return first;
  }
};
```

参考（References）

- Leetcode Discussion (yellowstone)

扩展 1

- 要求从后往前以 `k` 个为一组进行翻转。**(字节跳动（ByteDance）面试题)**

  例子，`1->2->3->4->5->6->7->8, k = 3`，

  从后往前以 `k=3` 为一组，

  - `6->7->8` 为一组翻转为 `8->7->6`，
  - `3->4->5` 为一组翻转为 `5->4->3`.
  - `1->2` 只有 2 个 nodes 少于 `k=3` 个，不翻转。

  最后返回：`1->2->5->4->3->8->7->6`

这里的思路跟从前往后以 `k` 个为一组进行翻转类似，可以进行预处理：

1. 翻转链表
2. 对翻转后的链表进行从前往后以 k 为一组翻转。
3. 翻转步骤 2 中得到的链表。

例子：`1->2->3->4->5->6->7->8, k = 3`

1. 翻转链表得到：`8->7->6->5->4->3->2->1`
2. 以 k 为一组翻转：`6->7->8->3->4->5->2->1`
3. 翻转步骤#2 链表：`1->2->5->4->3->8->7->6`

扩展 2

如果这道题你按照 92.reverse-linked-list-ii 提到的 `p1, p2, p3, p4`（四点法） 的思路来思考的话会很清晰。

代码如下（Python）：

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        if head is None or k < 2:
            return head
        dummy = ListNode(0)
        dummy.next = head
```

```
        pre = dummy
        cur = head
        count = 0
        while cur:
            count += 1
            if count % k == 0:
                pre = self.reverse(pre, cur.next)
                # end 调到下一个位置
                cur = pre.next
            else:
                cur = cur.next
        return dummy.next
    # (p1, p4)  左右都开放

    def reverse(self, p1, p4):
        prev, curr = p1, p1.next
        p2 = curr
        # 反转
        while curr != p4:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        # 将反转后的链表添加到原链表中
        # prev 相当于 p3
        p1.next = prev
        p2.next = p4
        # 返回反转前的头,  也就是反转后的尾部
        return p2

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

6. 删除排序数组中的重复项

题目描述

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

前置知识

- 数组
- 双指针

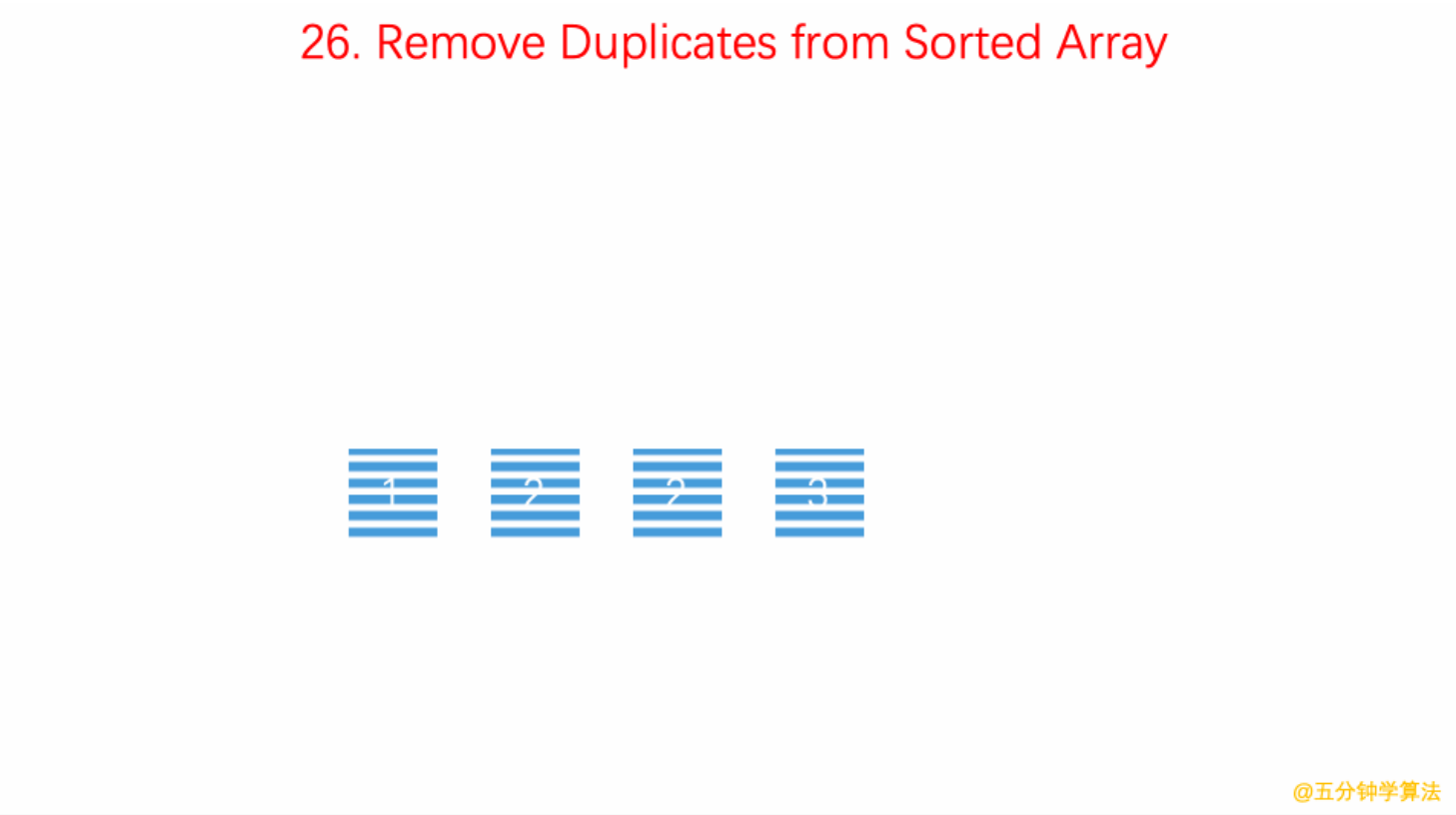
公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook
- microsoft

思路

使用快慢指针来记录遍历的坐标。

- 开始时这两个指针都指向第一个数字
- 如果两个指针指的数字相同，则快指针向前走一步
- 如果不同，则两个指针都向前走一步
- 当快指针走完整个数组后，慢指针当前的坐标加 1 就是数组中不同数字的个数



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

实际上这就是双指针中的快慢指针。在这里快指针是读指针， 慢指针是写指针。**从读写指针考虑， 我觉得更符合本质**。

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

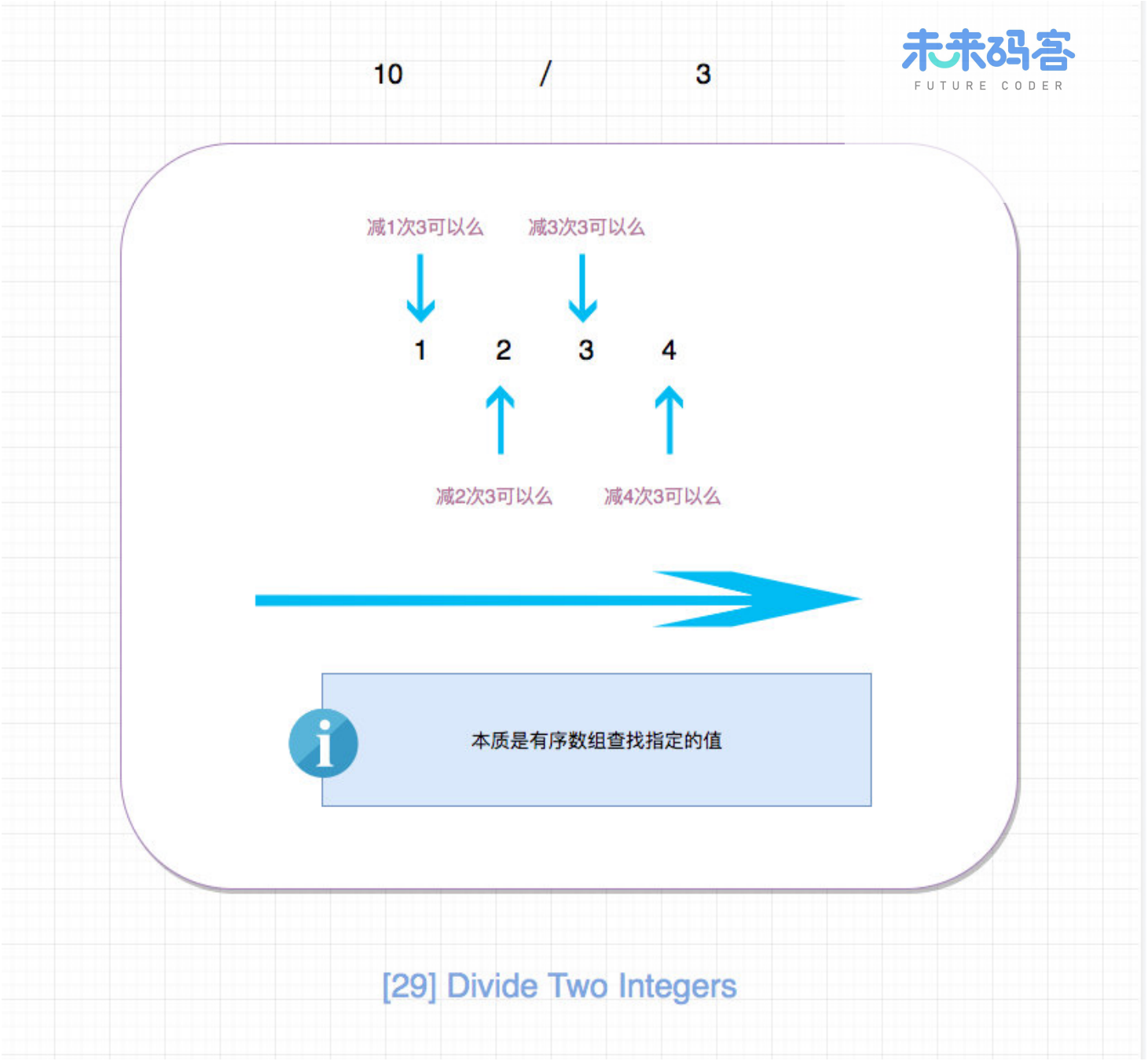
思路

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

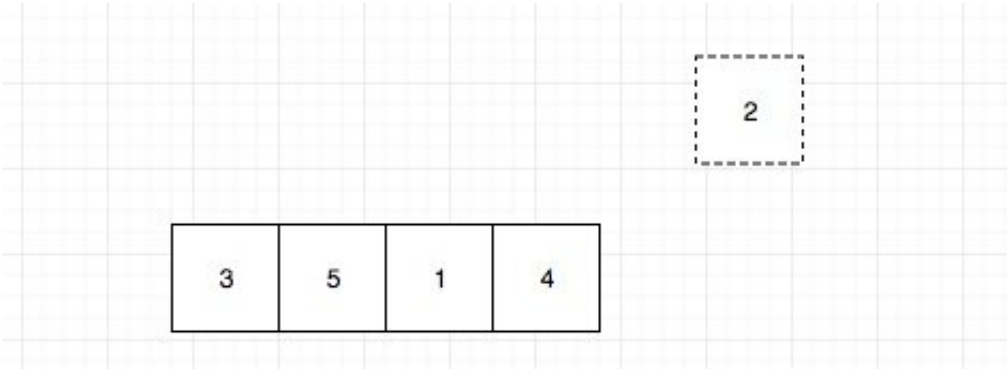
- 阿里
- 腾讯
- 百度
- 字节

思路

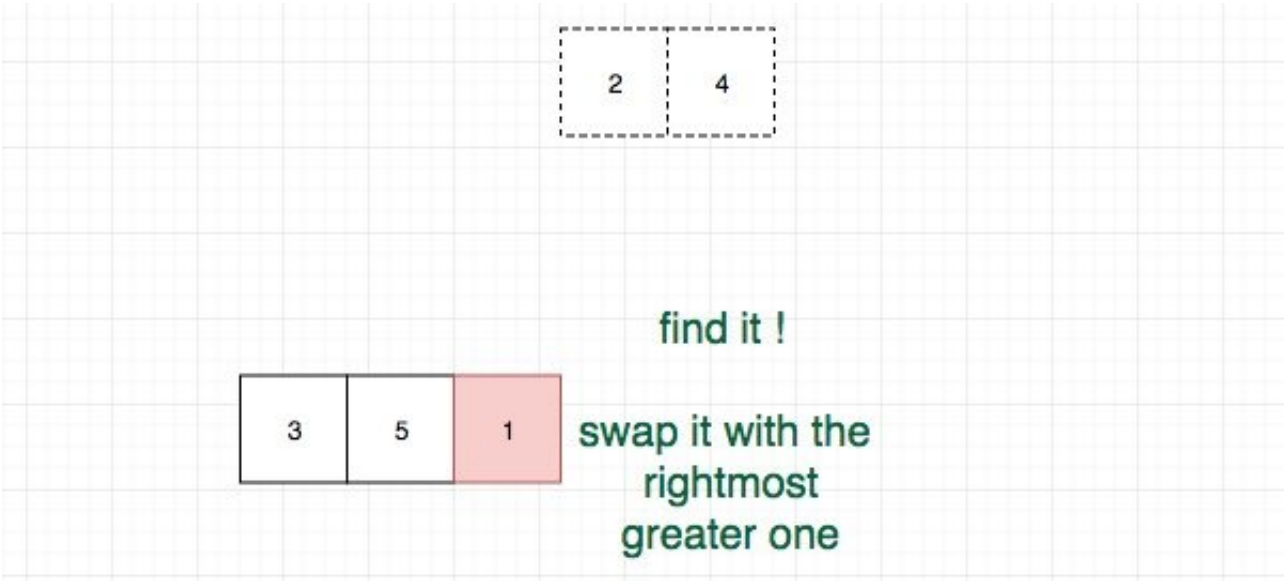
符合直觉的方法是按顺序求出所有的排列，如果当前排列等于 nums，那么我直接取下一个但是这种做法不符合 constant space 要求（题目要求直接修改原数组）,时间复杂度也太高,为 O(n!),肯定不是合适的解。

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：

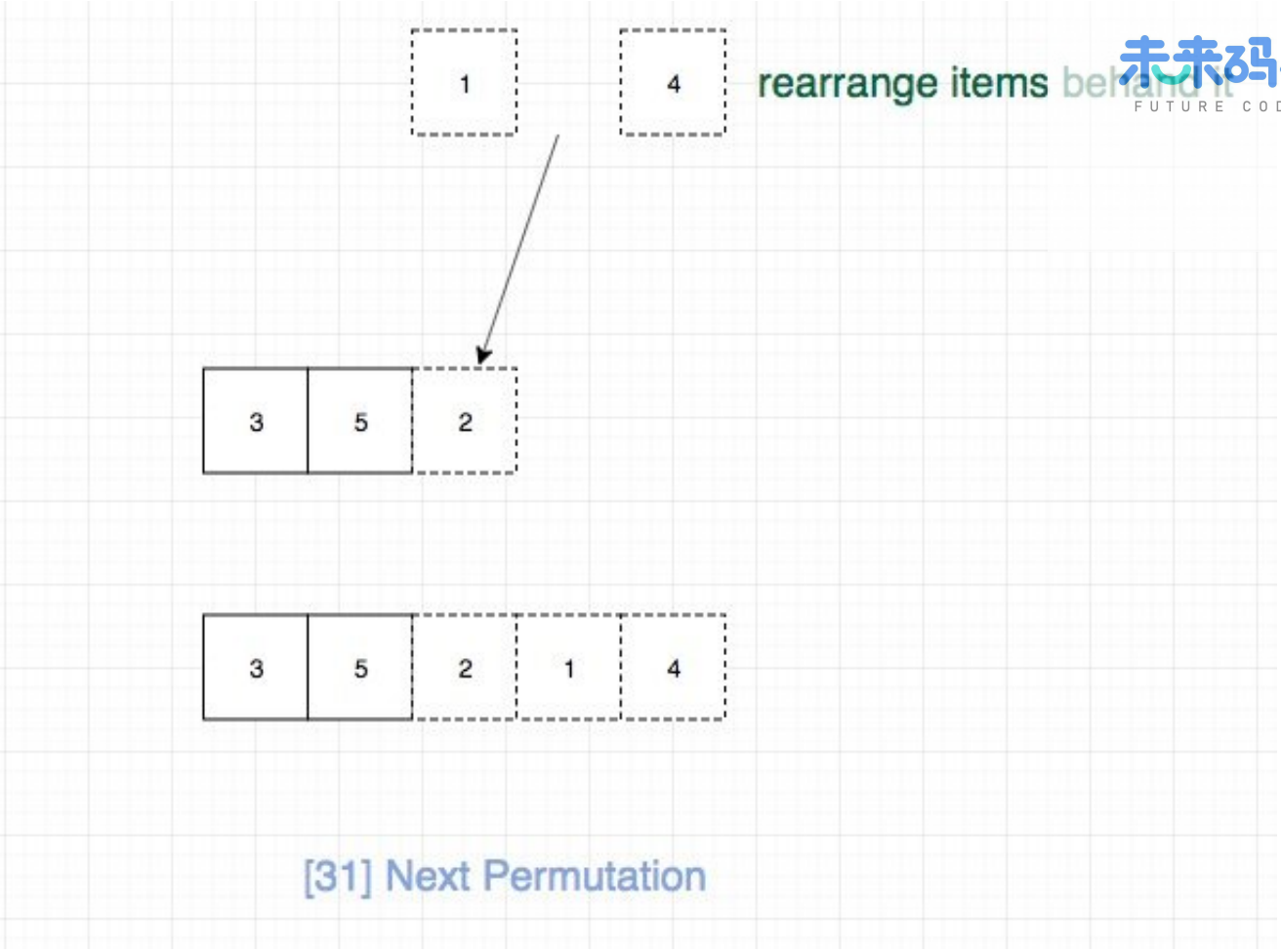


我们发现我们可以交换 4 和 2 就会变小，因此我们不能进行交换。

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:


```javascript
/*
 * @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

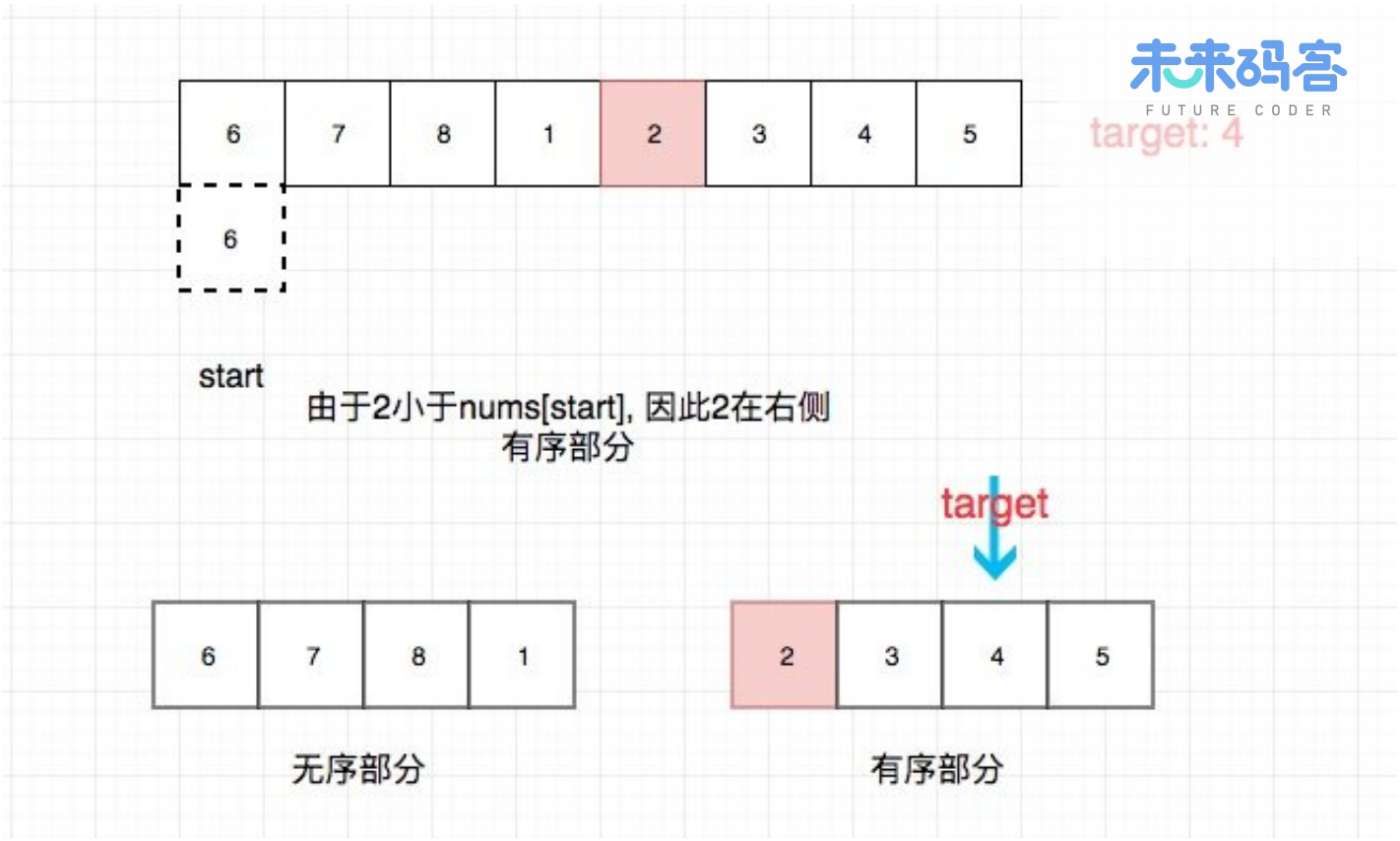
假如 mid 小于 start，则 mid 一定在右边有序部分。
假如 mid 大于等于 start， 则 mid 一定在左边有序部分。

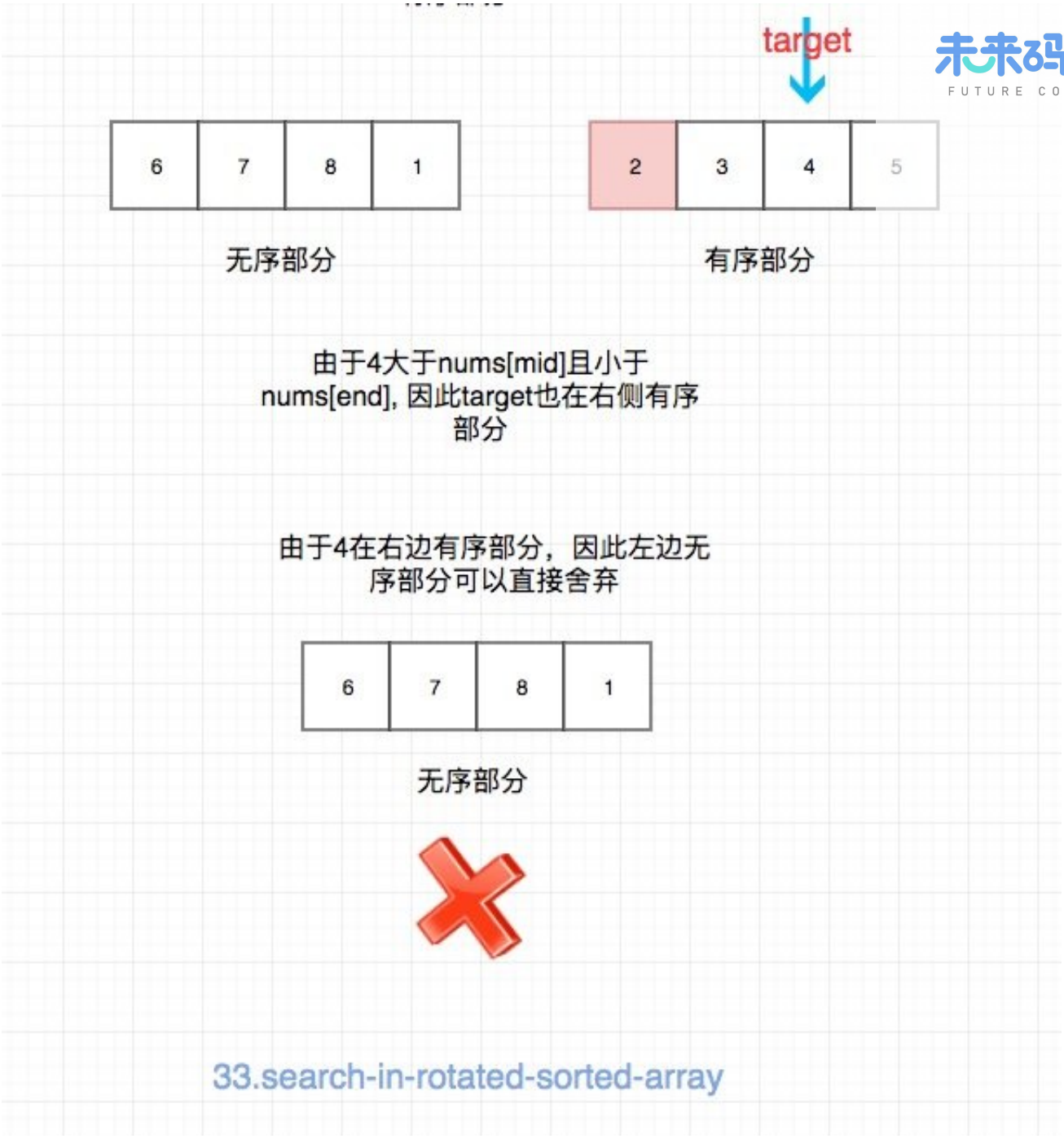
> 注意等号的考虑

- 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：


未来码客
FUTURE CODER
target: 4
6
7
8
1
2
3
4
5
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

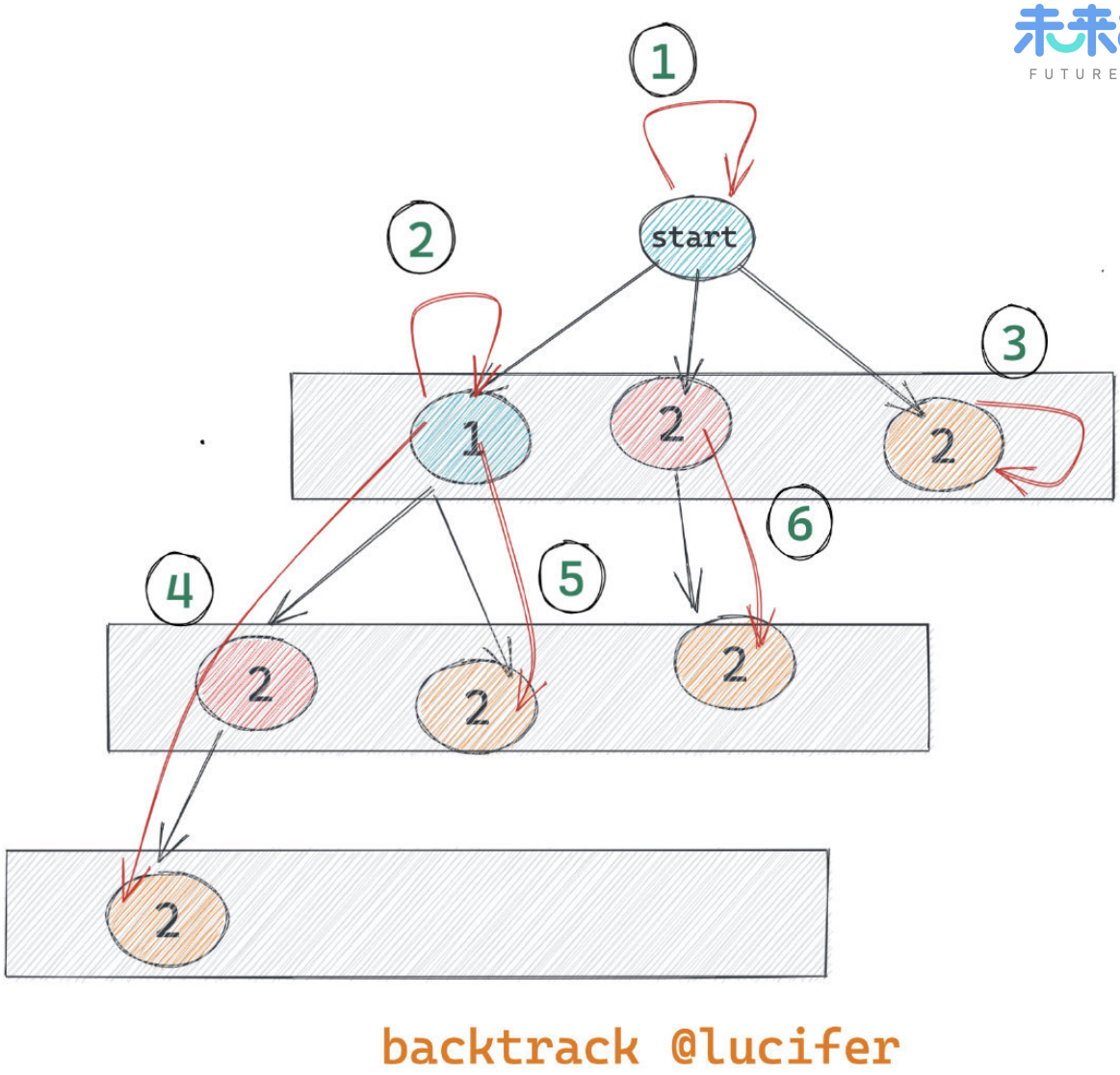
- 阿里
- 腾讯
- 百度
- 字节

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

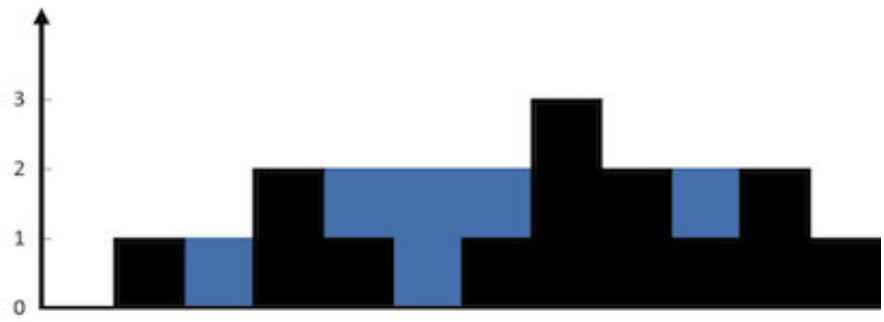
JS Code:

```
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
/**

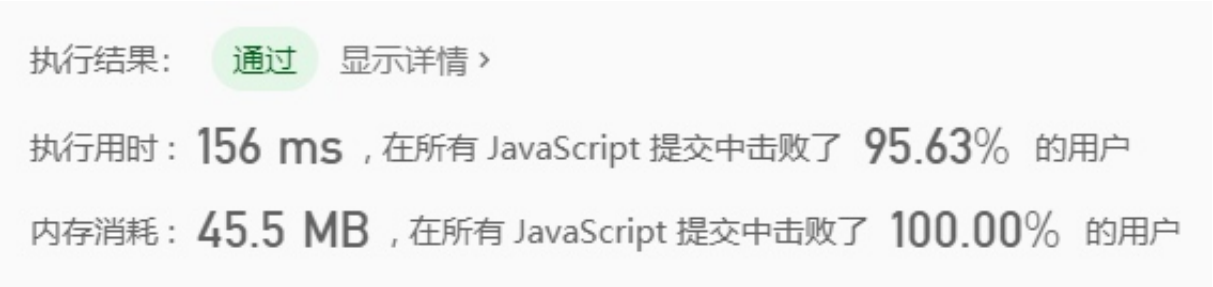
- @param {number[]} nums
- @return {number[][]}
  */
  var threeSum = function(nums) {
  let res = []
  let sum = 0
  // 将数组元素排序

```
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

更多精彩内容，持续汇总中.....