Leetcode 算法题解精选

从入门到入职 Rust 语言篇

晁岳攀 @ 鸟窝　著

Version 0.21.0
2024 年 3 月 17 日

题目题解源自力扣（*https://leetcode.cn/*）

前言

春节假期我制定了美好的计划：

- ~~过一边 python: 阅读《Python 编程~~
- ~~再深入过一遍 Rust: 《阅读 Rust 程序第二版》~~
- ~~完成几本书的大纲：把章节列出来~~
- ~~ebpf: 练习 cilium 的 ebpf 示例~~
- 做一些算法题：联系联系脑子

理想很丰满，现实很骨感，不幸的是我先选择了最后一项，本来想快速的在 leetcode 每个方向上找几道算法题，整理出来学习，估计也就一两天的功夫吧。

没想到整理这些题都会这么耗费时间，结果一个春节全部在整理 LeetCode 题目上了。不过开了一个头，我就不想放弃了，自己的泪水还得自己演，硬着头皮坚持下去，终于在假期的最后一天整理完了，我好想给自己放个假啊，一会去超时买点好吃的。

其实我也没有做太多的工作，主要是：

- 确定好章节，将主要的算法题目按照题目的类型进行了分类
- 每个分类精心挑选了几道题目。我最初的想法是最多选三道题目，不过在有个章节超过了三道，因为好题目太多了。
- 精选了官方的题解或者更好的网友提供的题解
- 对于没有 Go 语言的答案，我自己写了一些 Go 语言的答案 (其实是利用 github coplilot 进行了翻译，我只是做了校对和验证)

我希望这个文档能够帮助到大家，尤其对于那些想快速刷题面试的同学们。对于我来说，已经过了刷题的那个年纪，但是我也特别喜欢这些题，时不时的拿出来练习练习，也是一种锻炼。更何况有时候招一些同学入职，也需要一些题目的资源。

本书中所有的题目都来自力扣。力扣的版权声明如下：

> 作者：力扣官方题解著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

目录

1

链表

链表是一种常用的数据结构，特点是数据元素以节点的形式顺序存储，每个节点由两个部分组成：数据和指针。链表中的节点使用指针把节点串联起来，每个节点的指针都指向下一个节点，最后一个节点的指针指向 null。

链表的常见操作包括：

- **插入**：在链表的特定位置插入一个节点。需要更新前一个节点的指针域，使其指向新插入的节点，同时新节点的指针域指向原下一个节点。
- **查找**：在链表中查找特定数据的节点。需要从头节点开始遍历，依次检查每个节点的数据域，直到找到满足条件的节点。
- **遍历**：从头节点开始，依次访问每个节点，直到最后一个节点。

链表的优点是插入和删除操作效率很高，不需要移动其他元素。缺点是访问任意位置的元素需要从头开始遍历，时间复杂度为 O(n)。另外，链表中的节点是分配在堆上的，需要注意及时释放不再使用的节点空间。

常见的链表变种包括双向链表、循环链表等，用于处理更复杂的情况。

1.1 反转链表

No. 206 难度 Easy

给你单链表的头节点 *head* ， 请你反转链表， 并返回反转后的链表。

示例

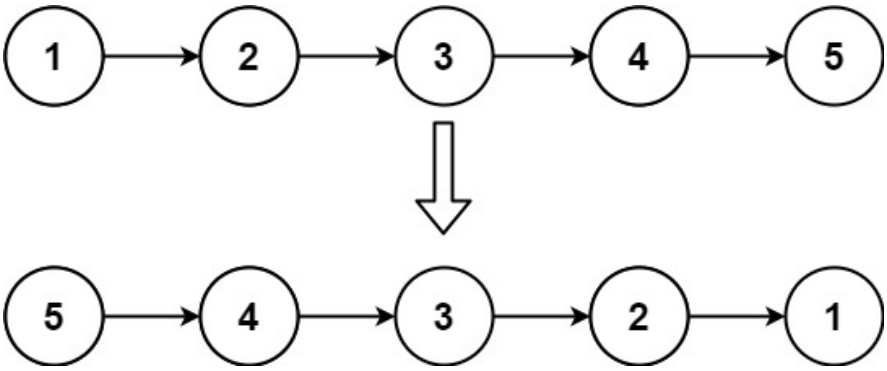


图 1.1. 反转链表

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

解法一: 迭代

思路:

- 从头节点开始, 依次遍历每个节点。
- 每次遍历, 将当前节点的 *curr.Next* 指针指向前一个节点。
- 由于节点没有引用其前一个节点，因此必须事先存储其前一个节点 (*prev*)。
- 为了不丢失后续节点, 需要在修改 *curr.Next* 指针之前, 先保存下一个节点 *next*。
- 最后返回新的头节点。

假设链表为 $1 \rightarrow 2 \rightarrow 3 \rightarrow \varnothing$，我们想要把它改成 $\varnothing \leftarrow 1 \leftarrow 2 \leftarrow 3$。

代码 1.1: 迭代反转链表

```rust
    pub fn reverse_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let mut prev = None;
        let mut curr = head;

        while let Some(mut node) = curr {
            let next = node.next.take();
            node.next = prev;
            prev = Some(node);
            curr = next;
        }

        prev
    }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的长度。需要遍历链表一次。

- 空间复杂度：$O(1)$。

解法二：递归

思路: 递归版本稍微复杂一些，其关键在于反向工作。假设链表的其余部分已经被反转，现在应该如何反转它前面的部分？

假设链表为：$n_1 \to \cdots \to n_{k-1} \to n_k \to n_{k+1} \to \cdots \to n_m \to \varnothing$，其中 n_k 是链表的第 k 个节点。假设从节点 n_{k+1} 到 n_m 已经被反转，我们正处于 n_k。$n_1 \to \cdots \to n_{k-1} \to n_k \to n_{k+1} \leftarrow \cdots \leftarrow n_m$。

我们希望 n_{k+1} 的下一个节点指向 n_k。所以，$n_k.Next.Next = n_k$。

需要注意的是 n_1 的下一个必须指向 $\varnothing$。如果忽略了这一点，链表中可能会产生环。

代码 1.2: 递归反转链表

```rust
    pub fn reverse_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        // Recursive termination condition:  return head if head is None or
       head.Next is None
        if head.is_none() || head.as_ref().unwrap().next.is_none() {
            return head;
        }

        // Recursive reverse of head.Next node
        let new_head = reverse_list(head.as_ref().unwrap().next.clone());

        // Reverse the Next pointer of head.Next node to point to head
        head.as_mut().unwrap().next.as_mut().unwrap().next = head;

        // Set head.Next to None
        head.as_mut().unwrap().next = None;

        new_head
    }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的长度。需要对链表的每个节点进行反转操作。
- 空间复杂度：$O(n)$，其中 n 是链表的长度。空间复杂度主要取决于递归调用的栈空间，最多为 n 层。

解法三：创建新节点

思路:

不考虑内存使用的问题，我们可以遍历原链表，每次遍历都新生成每一个节点，把原节点的数据复制过来，把指针指向新链表的前一个节点即可。

代码 1.3: 递归反转链表

```rust
    pub fn reverse_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
```

```
        let mut cur = None;
        let mut x = head;

        while let Some(node) = x {
            let mut new_node = Box::new(ListNode::new(node.val));
            new_node.next = cur;
            cur = Some(new_node);
            x = node.next;
        }

        cur
    }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的长度。需要对链表的每个节点进行反转操作。
- 空间复杂度：$O(n)$，其中 n 是链表的长度。每一个节点都复制了一份。

1.2 链表内指定区间反转

No. 92 难度 Medium

给你单链表的头指针 *head* 和两个整数 *left* 和 *right* ，其中 *left* <= *right* 。请你反转从位置 *left* 到位置 *right* 的链表节点，返回反转后的链表。

示例

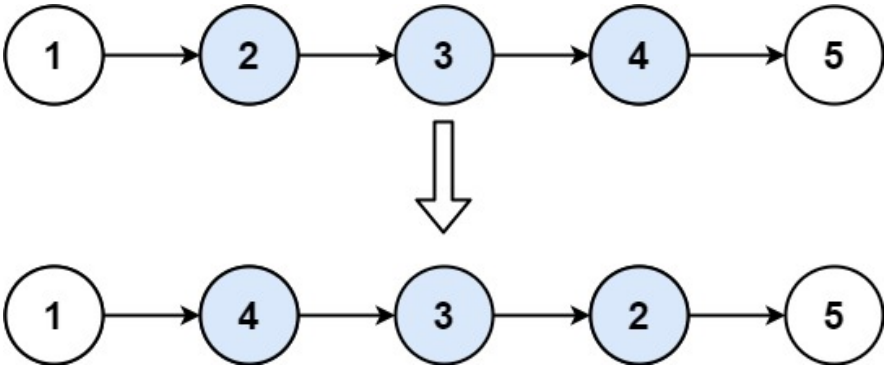


图 1.2. 指定区间反转链表

输入: head = [1,2,3,4,5], left = 2, right = 4
输出: [1,4,3,2,5]

解法一：处理子链表法

思路:

使用上一节的反转链表的算法，反转 *left* 到 *right* 部分以后，再拼接起来。我们还需要记录 *left* 的前一个节点，和 *right* 的后一个节点。

代码 1.4: 子链表法反转链表部分区段

```rust
// 反转一个链表，上一节的算法
fn reverse_linked_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
    let mut pre = None;
    let mut cur = head;

    while let Some(mut node) = cur {
        let next = node.next.take();
        node.next = pre;
        pre = Some(node);
        cur = next;
    }

    pre
}

// 反转链表的一个区间
fn reverse_between(head: Option<Box<ListNode>>, left: i32, right: i32) -> Option<Box<ListNode>> {
    // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
    let mut dummy_node = Some(Box::new(ListNode::new(-1)));
    dummy_node.as_mut().unwrap().next = head;

    // 前一个节点
    let mut pre = &mut dummy_node;
    // 找到 left 的前一个节点
    // 第 0 个是虚拟节点，第 left-1 个是 left 的前一个节点
    for _ in 0..left-1 {
        pre = &mut pre.as_mut().unwrap().next;
    }

    // 找到 right 节点，再走 right - left + 1 步，来到 right 节点
    let mut right_node = pre;
    for _ in 0..right-left+1 {
        right_node = &mut right_node.as_mut().unwrap().next;
    }

    // 左节点
    let mut left_node = pre.as_mut().unwrap().next.take();
    // 右节点的下一个节点
    let right_node_next = right_node.as_mut().unwrap().next.take();

    // 切出子链表，也就是 left 的前一个节点的 next 和 right 的 next 都设置为 None
    pre.as_mut().unwrap().next = None;
    right_node.as_mut().unwrap().next = None;

    // 反转子链表
    let new_head = reverse_linked_list(left_node);

```

```
48          // 把子链表重新接回原链表
49          pre.as_mut().unwrap().next = new_head;
50          left_node.as_mut().unwrap().next = right_node_next;
51
52          dummy_node.unwrap().next
53      }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表总节点数。最坏情况下，需要遍历整个链表。
- 空间复杂度：$O(1)$，只用到了常数个变量。

漂亮哈，借助我们已知的算法，去解决更复杂的问题！

解法 2: 头插法

思路: 如果 *left* 和 *right* 的区域很大，比如极端情况它们分别是是链表的头节点和尾节点时，找到 *left* 和 *right* 需要遍历一次，反转它们之间的链表还需要遍历一次，虽然总的时间复杂度为 $O(N)$，但遍历了链表 2 次，可不可以只遍历一次呢？答案是可以的。

整体思想是：在需要反转的子区间里，每遍历到一个节点，让这个新节点来到反转部分的起始位置，也就是头插法。

我们使用三个指针变量 *pre*、*cur*、*next* 来记录反转的过程中需要的变量，它们的意义如下：

- **cur**: 开始指向待反转区域的第一个节点 *left*，循环过程中不断变化指向后一个节点；
- **next**: 永远指向 *cur* 的下一个节点，循环过程中，*cur* 变化以后 *next* 会变化；
- **pre**: 永远指向待反转区域的第一个节点 *left* 的前一个节点，在循环过程中不变。

注意待反转区域不是 子区间，而是子区间中待处理的部分。

代码 1.5: 头插法反转链表部分区段

```
1       fn reverse_between(head: Option<Box<ListNode>>, left: i32, right: i32) ->
          Option<Box<ListNode>> {
2           // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
3           let mut dummy_node = Some(Box::new(ListNode::new(-1)));
4           dummy_node.as_mut().unwrap().next = head;
5
6           // 前一个节点
7           let mut pre = &mut dummy_node;
8           // 找到 left 的前一个节点
9           for _ in 0..left-1 {
10              pre = &mut pre.as_mut().unwrap().next;
11          }
12
13          // 待反转区的第一个节点
14          let mut cur = pre.as_mut().unwrap().next.take();
```

```
15          // 遍历子区间
16          for _ in 0..right-left {
17              let mut next = cur.as_mut().unwrap().next.take(); // 保存待处理区间的
        第二个节点
18              cur.as_mut().unwrap().next = next.as_mut().unwrap().next.take(); //
        cur 的下一个节点指向 next 的下一个节点，在子区间中向后移动得到新的待反转区域
19              next.as_mut().unwrap().next = pre.as_mut().unwrap().next.take(); //
        接下来的这两句就是插入到子区间的头部
20              pre.as_mut().unwrap().next = next;
21          }
22
23          dummy_node.unwrap().next
24      }
```

每次遍历子区间，我们就从待反转区域摘下一个节点，放在子区间的头部。注意这里：

- 我们从子区间的第二个节点开始摘；
- *cur* 指向刚摘下的节点的 *next*。
- 摘下的节点的 *next* 指向子区间的头部节点；
- *pre* 的 *next* 指向这个节点。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表总节点数。最多只遍历了链表一次，就完成了反转。
- 空间复杂度：$O(1)$，只用到了常数个变量。

更简单的是，删除链表的一个节点，或者一个子区间，或者满足条件的节点，基本上我们一次遍历就可以解决。稍微复杂一点的就是从排好序或者排好序的链表中删除重复的元素。

更复杂的是K 个一组反转链表，属于 Hard 级别的题目。我们可以采用类似的方法，将链表分为已反转区，待反转区和未反转区，然后对待反转区进行反转。感兴趣的同学可以自行尝试。

1.3 排序链表

No. 148 繁 Easy

给定链表的头结点 *head* ，请将其按升序排列并返回排序后的链表。

示例

输入: head = [-1,5,3,4,0]
输出: [1,4,3,2,5]

这道题考虑时间复杂度优于 $O(n^2)$ 的排序算法。题目的进阶问题要求达到 $O(n \log n)$ 的时间复杂度和 $O(1)$ 的空间复杂度，时间复杂度是 $O(n \log n)$ 的排序算法包括归并排序、

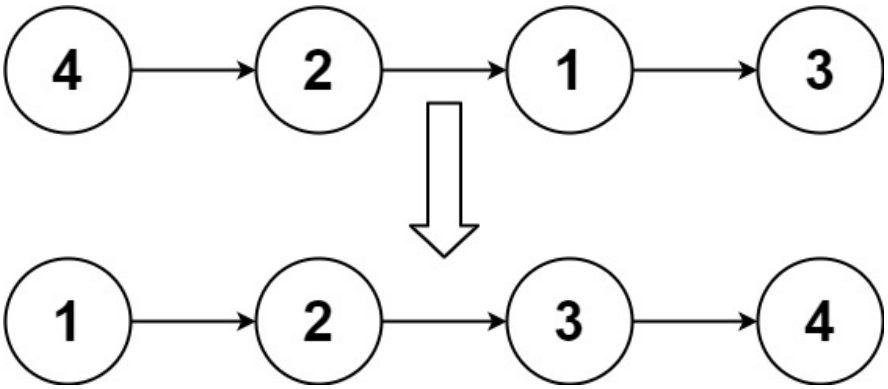


图 1.3. 排序链表

堆排序和快速排序（快速排序的最差时间复杂度是 $O(n^2)$），其中最适合链表的排序算法是归并排序。

归并排序基于分治算法。最容易想到的实现方式是自顶向下的递归实现，但是考虑到递归调用的栈空间，自顶向下归并排序的空间复杂度是 $O(\log n)$。如果要达到 $O(1)$ 的空间复杂度，则需要使用自底向上的实现方式。

接下来我们使用自底向上的方法实现归并排序，以便达到 $O(1)$ 的空间复杂度。

首先求得链表的长度 *length*，然后将链表拆分成子链表进行合并。

具体做法如下。

1. 用 $subLength$ 表示每次需要排序的子链表的长度，初始时 $subLength = 1$。
2. 每次将链表拆分成若干个长度为 $subLength$ 的子链表（最后一个子链表的长度可以小于 $subLength$），按照每两个子链表一组进行合并，合并后即可得到若干个长度为 $subLength \times 2$ 的有序子链表（最后一个子链表的长度可以小于 $subLength \times 2$）。
3. 将 $subLength$ 的值加倍，重复第 2 步，对更长的有序子链表进行归并操作，直到有序子链表的长度大于或等于 $length$，整个链表排序完毕。

代码 1.6: 归并排序链表

```rust
    // 将两个已排好序的链表合并
    fn merge(head1: Option<Box<ListNode>>, head2: Option<Box<ListNode>>) ->
      Option<Box<ListNode>> {
        // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
        let mut dummy_head = Some(Box::new(ListNode::new(0)));
        let mut temp = &mut dummy_head;
        let (mut temp1, mut temp2) = (head1, head2);

        while let (Some(node1), Some(node2)) = (temp1.as_ref(), temp2.as_ref()) {
            if node1.val <= node2.val {
                temp.as_mut().unwrap().next = temp1.take();
                temp1 = temp1.unwrap().next;
            } else {
                temp.as_mut().unwrap().next = temp2.take();
```

```
                temp2 = temp2.unwrap().next;
            }
            temp = &mut temp.as_mut().unwrap().next;
        }

        // 两个链表中有一个为空，将另一个链表的剩余部分接到新链表的尾部
        if temp1.is_some() {
            temp.as_mut().unwrap().next = temp1;
        } else if temp2.is_some() {
            temp.as_mut().unwrap().next = temp2;
        }

        dummy_head.unwrap().next
    }

    // 按升序排链表，采用自底而上的归并算法
    fn sort_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let mut length = 0;
        let mut node = &head;
        while let Some(n) = node {
            length += 1;
            node = &n.next;
        }

        // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
        let mut dummy_head = Some(Box::new(ListNode::new(0)));
        dummy_head.as_mut().unwrap().next = head;

        // 每次将链表拆分成若干个子链表，长度逐次翻倍
        let mut sub_length = 1;
        while sub_length < length {
            let mut prev = &mut dummy_head;
            let mut cur = prev.as_mut().unwrap().next.take();

            while cur.is_some() {
                // 第一个子链表
                let mut head1 = cur.take();
                let mut temp = &mut head1;
                for _ in 1..sub_length {
                    temp = &mut temp.as_mut().unwrap().next;
                }

                // 第二个子链表
                let mut head2 = temp.as_mut().unwrap().next.take();
                temp.as_mut().unwrap().next = None;
                cur = head2.take();
                temp = &mut cur;
                for _ in 1..sub_length {
                    if let Some(node) = temp {
                        temp = &mut node.next;
                    }
                }

```

```
67                  // 剩余的子链表
68                  let mut next = None;
69                  if let Some(node) = temp {
70                      next = node.next.take();
71                  }
72
73                  // 合并两个子链表
74                  prev.as_mut().unwrap().next = merge(head1, head2);
75
76                  // 找到合并后的链表的尾部
77                  let mut tail = prev.as_mut().unwrap().next.as_mut();
78                  while let Some(node) = tail {
79                      tail = node.next.as_mut();
80                  }
81                  prev = tail;
82
83                  // 当前待合并的链表
84                  cur = next;
85              }
86
87              sub_length <<= 1;
88          }
89
90          dummy_head.unwrap().next
91      }
```

为啥是自底而上的归并算法呢？就是先一一合并，然后再二二合并，再四四合并...，一直合并下去，直到所有的节点都合并好了。

复杂度分析

- 时间复杂度：$O(n \log n)$，其中 n 是链表的长度。
- 空间复杂度：$O(1)$，只用到了常数个变量。

一个更复杂的问题是 重排链表，这个问题是中度难度级别的题目，就是把一个链表首尾首尾 ...进行重新排序，这个是有时间复杂度为 $O(n \log n)$，空间负责度为 $O(1)$ 的解法的：找到链表的中点 (快慢指针)，反转右边的子链表，然后合并左右两边的子链表就好了，注意这里的合并是交叉合并，不是排序。当然时间复杂度为 $O(n)$，空间负责度为 $O(n)$ 的解法更简单些，把节点放在一个数组中，依次从数组的首尾各取一个节点组成新的节点就可以了。

本节开始我们说归并排序、堆排序和快速排序时间复杂度都是 $O(n \log n)$，为什么说归并排序更适合链表呢？请思考。

1.4 合并链表

No 23 难度 Hard

给定一个链表数组，每个链表都已经按升序排列。

请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

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。

$$1->1->2->3->4->4->5->6$$

解法一：顺序合并法

思路: 在解决合并 K 个排序链表这个问题之前，我们先来看一个更简单的问题：如何合并两个有序链表？假设链表 *a* 和 *b* 的长度都是 n，如何在 $O(n)$ 的时间代价以及 $O(1)$ 的空间代价完成合并？这个问题在面试中常常出现，为了达到空间代价是 $O(1)$，我们的宗旨是原地调整链表元素的 *next* 指针完成合并，这个属于初级的问题了，我们就不多说了，它的时间复杂度：$O(n)$, 空间复杂度：$O(1)$。

我们可以想到一种最朴素的方法：用一个变量 *ans* 来维护合并后的链表，第 i 次循环把第 i 个链表和 *ans* 合并，答案保存到 *ans* 中。这使用的就是两个已排序链表的归并算法。

代码 1.7: 归并排序多个已排序链表

```
1     // 合并两个链表
2     fn merge_two_lists(a: Option<Box<ListNode>>, b: Option<Box<ListNode>>) ->
        Option<Box<ListNode>> {
3         match (a, b) {
4             (None, None) => None,
5             (Some(a), None) => Some(a),
6             (None, Some(b)) => Some(b),
7             (Some(mut a), Some(mut b)) => {
8                 if a.val < b.val {
9                     a.next = merge_two_lists(a.next, Some(b));
10                    Some(a)
11                } else {
12                    b.next = merge_two_lists(Some(a), b.next);
13                    Some(b)
14                }
15            }
16        }
17    }
18
19    // 合并 K 个排序链表
20    fn merge_k_lists(lists: Vec<Option<Box<ListNode>>>) -> Option<Box<ListNode>>
        {
21        let mut ans = None;
22        for list in lists {
23            ans = merge_two_lists(ans, list);
24        }
```

```
25          ans
26      }
```

复杂度分析

- 时间复杂度：故渐进时间复杂度为 $O(k^2n)$，其中 n 是链表的长度，k 是链表的数量。
- 空间复杂度：$O(1)$，只用到了常数个变量。

这种顺序合并的方法，有一个明显可以提升的方法就是两两合并链表，**分而治之**。怎么实现呢，大家可以思考。这种算法的时间复杂度是 $O(kn \times \log k)$，空间复杂度是 $O(k)$。

解法二：使用优先队列合并

思路:

这个方法和前两种方法的思路有所不同，我们需要维护当前每个链表没有被合并的元素的最前面的一个节点，k 个链表就最多有 k 个满足这样条件的元素，每次在这些元素里面选取 *val* 属性最小的元素合并到答案中。在选取最小元素的时候，我们可以用优先队列来优化这个过程。

代码 1.8: 使用优先队列合并已排序链表

```rust
// 优先级队列
use std::cmp::Ordering;
use std::collections::BinaryHeap;

// 合并 K 个排序链表
fn merge_k_lists(lists: Vec<Option<Box<ListNode>>>) -> Option<Box<ListNode>> {
    let mut pq = BinaryHeap::new();

    // 把各队列节点加入优先级队列
    for node in lists {
        if let Some(n) = node {
            pq.push(n);
        }
    }

    // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
    let mut dummy = Box::new(ListNode { val: 0, next: None });
    // 尾节点
    let mut tail = &mut dummy;

    // 从优先级队列中取出最小的节点，加入新链表
    while let Some(node) = pq.pop() {
        tail.next = Some(node);
        tail = tail.next.as_mut().unwrap();
        if let Some(next) = tail.next.take() {
            pq.push(next);
        }
```

```
28          }
29
30          dummy.next
31      }
```

这里使用 Go 语言标准库中的优先级队列不算作弊欧。

复杂度分析

- 时间复杂度：故渐进时间复杂度为 $O(kn \times \log k)$，其中 n 是链表的长度，k 是链表的数量。
- 空间复杂度：$O(k)$。优先队列中的元素不超过 k 个

1.5 旋转链表

No. 61 难度 Medium

给你一个链表的头节点 *head* ，旋转链表，将链表每个节点向右移动 k 个位置。

示例

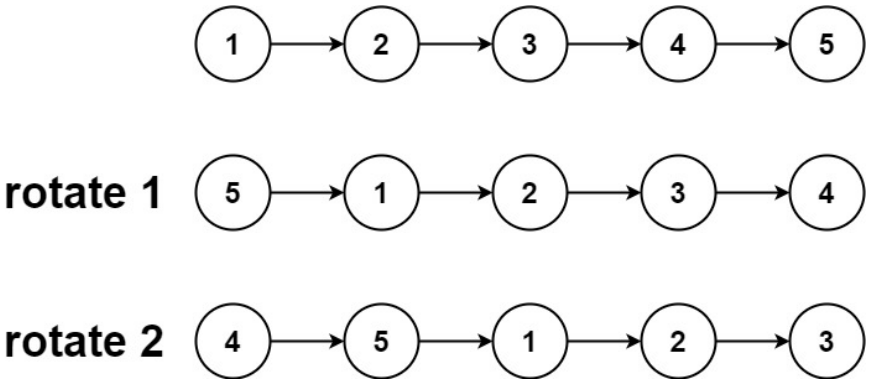


图 1.4. 右移链表

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

解法一：闭合为环

思路:

记给定链表的长度为 n，注意到当向右移动的次数 $k \geq n$ 时，我们仅需要向右移动 $k \bmod n$ 次即可。因为每 n 次移动都会让链表变为原状。这样我们可以知道，新链表的最后一个节点为原链表的第 $(n-1)-(k \bmod n)$ 个节点（从 0 开始计数）。

这样，我们可以先将给定的链表连接成环，然后将指定位置断开。

具体代码中，我们首先计算出链表的长度 n，并找到该链表的末尾节点，将其与头节点相连。这样就得到了闭合为环的链表。然后我们找到新链表的最后一个节点（即原链表的第 $(n-1)-(k \bmod n)$ 个节点），将当前闭合为环的链表断开，即可得到我们所需要的结果。

特别地，当链表长度不大于 1，或者 k 为 n 的倍数时，新链表将与原链表相同，我们无需进行任何处理。

代码 1.9: 子链表法反转链表部分区段

```rust
    fn rotate_right(head: Option<Box<ListNode>>, k: i32) -> Option<Box<ListNode>>
        {
        // 处理特殊情况
        if k == 0 || head.is_none() || head.as_ref().unwrap().next.is_none() {
            return head;
        }

        // 得到链表长度
        let mut n = 1;
        let mut iter = head.as_ref().unwrap();
        while let Some(next) = &iter.next {
            iter = next;
            n += 1;
        }

        // 计算实际移动的次数
        let add = n - k % n;
        if add == n {
            return head;
        }

        // 移动到新的头节点
        let mut iter = head.as_ref().unwrap();
        for _ in 0..add {
            iter = iter.next.as_ref().unwrap();
        }

        // 新的头节点
        let ret = iter.next.take();
        // 新的队尾
        iter.next = None;

        ret
    }
```

复杂度分析

- 时间复杂度：$O(n)$，最坏情况下，我们需要遍历该链表两次。
- 空间复杂度：$O(1)$，我们只需要常数的空间存储若干变量。

1.6 分隔链表

No. 86 难度 Medium

给你一个链表的头节点 *head* 和一个特定值 x，请你对链表进行分隔，使得所有小于 x 的节点都出现在大于或等于 x 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

示例

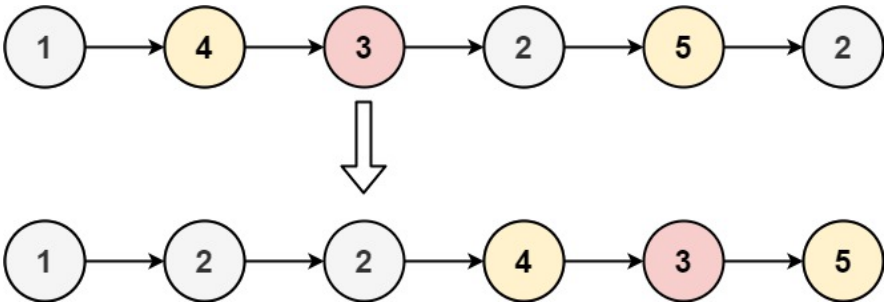


图 1.5. 分隔链表

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

注意我们只要求比 x 小的值挪到左边，并没有要求大于等于 x 的值挪到右边。

解法一：模拟

思路: 直观来说我们只需维护两个链表 *small* 和 *large* 即可，*small* 链表按顺序存储所有小于 x 的节点，*large* 链表按顺序存储所有大于等于 x 的节点。遍历完原链表后，我们只要将 *small* 链表尾节点指向 *large* 链表的头节点即能完成对链表的分隔。

为了实现上述思路，我们设 *smallHead* 和 *largeHead* 分别为两个链表的虚拟节点，即它们的 *next* 指针指向链表的头节点，这样做的目的是为了更方便地处理头节点为空的边界条件。同时设 *small* 和 *large* 节点指向当前链表的末尾节点。开始时 *smallHead=small*,*largeHead=large*。随后，从前往后遍历链表，判断当前链表的节点值是否小于 x，如果小于就将 *small* 的 *next* 指针指向该节点，否则将 *large* 的 *next* 指针指向该节点。

遍历结束后，我们将 *large* 的 *next* 指针置空，这是因为当前节点复用的是原链表的节点，而其 *next* 指针可能指向一个小于 x 的节点，我们需要切断这个引用。同时将 *small* 的 *next* 指针指向 *largeHead* 的 *next* 指针指向的节点，即真正意义上的 *large* 链表的头节点。最后返回 *smallHead* 的 *next* 指针即为我们要求的答案。

代码 1.10: 模拟法分隔链表

```
1    // 分隔链表
2    fn partition(head: Option<Box<ListNode>>, x: i32) -> Option<Box<ListNode>> {
```

```
        // 一个技巧，使用虚拟头节点可以避免复杂的分类讨论
        let mut small = ListNode::new(0);
        let mut small_head = &mut small;
        let mut large = ListNode::new(0);
        let mut large_head = &mut large;

        // 遍历链表
        let mut curr = head;
        while let Some(mut node) = curr {
            curr = node.next.take();
            if node.val < x {
                small_head.next = Some(node);
                small_head = small_head.next.as_mut().unwrap();
            } else {
                large_head.next = Some(node);
                large_head = large_head.next.as_mut().unwrap();
            }
        }
        // 必须将 large 的尾部置空，此时 large.Next 可能指向一个小于 x 的节点，也可能为 nil，不管怎么样，我们都需要切断这个引用
        large_head.next = None;
        // 将 small 的尾部指向 large 的头部
        small_head.next = large.next;

        small.next
    }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是原链表的长度。我们对该链表进行了一次遍历。
- 空间复杂度：$O(1)$。

类似的，奇偶链表把链表按照奇偶进行分隔也可以使用这个算法。

这里还有一个中级难度的分隔链表的题目，就是将一个链表均匀的分隔成 k 个部分。注意不节点逐个的放在子链表中，而且从链表中割下一块放在一个子链表中，再割下一块放在第二个子链表中。这个题目的解法是先计算出链表的长度，再计算出子链表的长度，然后进行逐步的分隔就好。

1.7 交换链表

No. 24 难度 Medium

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

这个题目是中级难度的题目，我们可以使用递归或者迭代的方法来解决，无论使用哪种算法，代码都很简练，很能考察我们的编程能力。

示例

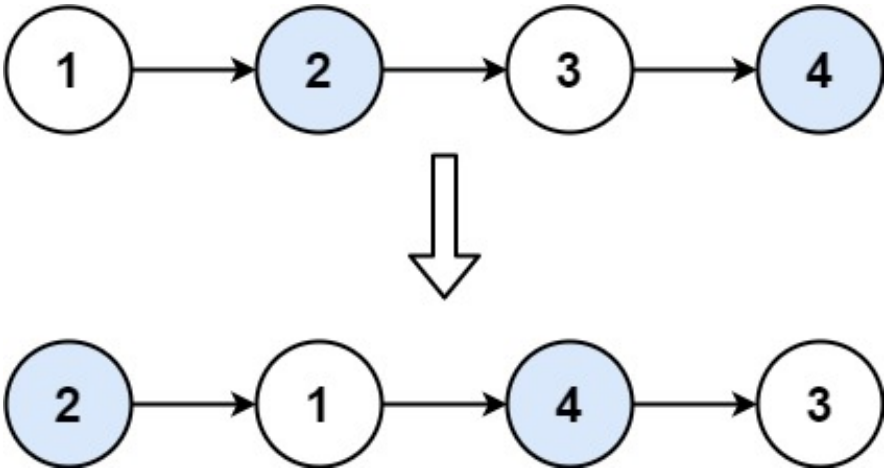


图 1.6. 两两交换链表的节点

输入: head = [1,2,3,4]
输出: [2,1,4,3]

解法一: 递归

思路: 递归的终止条件是链表中没有节点，或者链表中只有一个节点，此时无法进行交换。

如果链表中至少有两个节点，则在两两交换链表中的节点之后，原始链表的头节点变成新的链表的第二个节点，原始链表的第二个节点变成新的链表的头节点。链表中的其余节点的两两交换可以递归地实现。在对链表中的其余节点递归地两两交换之后，更新节点之间的指针关系，即可完成整个链表的两两交换。

用 *head* 表示原始链表的头节点，新的链表的第二个节点，用 *newHead* 表示新的链表的头节点，原始链表的第二个节点，则原始链表中的其余节点的头节点是 *newHead.next*。令 *head.next = swapPairs(newHead.next)*，表示将其余节点进行两两交换，交换后的新的头节点为 *head* 的下一个节点。然后令 *newHead.next = head*，即完成了所有节点的交换。最后返回新的链表的头节点 *newHead*

代码 1.11: 递归交换节点

```
    fn swap_pairs(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        // 递归的终止条件是链表中没有节点，或者链表中只有一个节点，此时无法进行交换
        if head.is_none() || head.as_ref().unwrap().next.is_none() {
            return head;
        }
        // 交换节点，新的头结点是原来的第二个节点
        let mut new_head = head.as_mut().unwrap().next.take();
        // 原来的头结点变成新的第二个节点，并且连接上后面交换后的链表
        head.as_mut().unwrap().next = swap_pairs(new_head.as_mut().unwrap().next.
    take());
        // 新的头结点连接上原来的头结点
        new_head.as_mut().unwrap().next = head;
```

```
12
13          new_head
14      }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的节点数量。需要对每个节点进行更新指针的操作。
- 空间复杂度：$O(n)$，其中 n 是链表的节点数量。空间复杂度主要取决于递归调用的栈空间。

解法二：迭代

递归就是那样，可以用简洁的代码实现一个算法，但是不是那么容易理解，因为栈的原因，空间复杂度相对较高，事实上我们也可以通过迭代的方式实现两两交换链表中的节点。

思路: 创建虚拟结点 *dummyHead*，令 *dummyHead.next = head*。令 *temp* 表示当前到达的节点，初始时 *temp = dummyHead*。每次需要交换 *temp* 后面的两个节点。

如果 *temp* 的后面没有节点或者只有一个节点，则没有更多的节点需要交换，因此结束交换。否则，获得 *temp* 后面的两个节点 *node1* 和 *node2*

具体而言，交换之前的节点关系是 *temp -> node1 -> node2*，交换之后的节点关系要变成 *temp -> node2 -> node1*，因此需要进行如下操作。

```
temp.next = node2
node1.next = node2.next
node2.next = node1
```

完成上述操作之后，节点关系即变成 *temp -> node2 -> node1*。再令 *temp = node1*，对链表中的其余节点进行两两交换，直到全部节点都被两两交换。两两交换链表中的节点之后，新的链表的头节点是 *dummyHead.next*，返回新的链表的头节点即可。

代码 1.12: 迭代交换节点

```rust
fn swap_pairs(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
    // Create a dummy head node to avoid complex classification discussions
    let mut dummy_head = Some(Box::new(ListNode { val: 0, next: head }));
    let mut temp = &mut dummy_head;

    // Iterate and swap two nodes at a time
    while temp.as_ref().unwrap().next.is_some() && temp.as_ref().unwrap().
    next.as_ref().unwrap().next.is_some() {
        // Get the first node
        let mut node1 = temp.as_mut().unwrap().next.take().unwrap();
        // Get the second node
        let mut node2 = node1.next.take().unwrap();
        // Swap the nodes
        temp.as_mut().unwrap().next = Some(node2);
```

```
14              node1.next = node2.next.take();
15              node2.next = Some(node1);
16
17              // Move to the next pair of nodes
18              temp = &mut temp.as_mut().unwrap().next.as_mut().unwrap().next;
19          }
20
21          dummy_head.unwrap().next
22      }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的节点数量。需要对每个节点进行更新指针的操作。
- 空间复杂度：$O(1)$。

1.8 链表随机节点

No. 382 难度 Medium

给你一个单链表，随机选择链表的一个节点，并返回相应的节点值。每个节点被选中的概率一样。

实现 *Solution* 类：

- *Solution(ListNode head)* 使用整数数组初始化对象。
- *int getRandom()* 从链表中随机选择一个节点并返回该节点的值。链表中所有节点被选中的概率相等。

示例

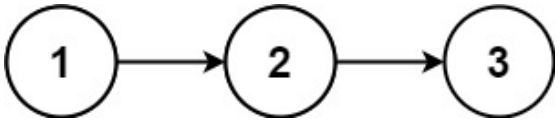


图 1.7. 链表

输入:

```
["Solution", "getRandom", "getRandom", "getRandom", "getRandom", "getRandom"]
    [[[1, 2, 3]], [], [], [], [], []]
```

输出:

```
[null, 1, 3, 2, 2, 3]
```

解释:

```
Solution solution = new Solution([1, 2, 3]);
solution.getRandom(); // 返回 1
solution.getRandom(); // 返回 3
solution.getRandom(); // 返回 2
solution.getRandom(); // 返回 2
solution.getRandom(); // 返回 3
// getRandom() 方法应随机返回 1、2、3中的一个，每个元素被返回的概率相等。
```

解法一：记录所有的链表节点

思路：我们可以在初始化时，用一个数组记录链表中的所有元素，这样随机选择链表的一个节点，就变成在数组中随机选择一个元素。

代码 1.13: 记录所有的链表节点

```rust
use rand::Rng;

struct Solution {
    values: Vec<i32>,
}

impl Solution {
    fn new(head: Option<Box<ListNode>>) -> Solution {
        let mut values = Vec::new();
        let mut node = head;
        while let Some(n) = node {
            values.push(n.val);
            node = n.next;
        }
        Solution { values }
    }

    fn get_random(&self) -> i32 {
        let mut rng = rand::thread_rng();
        let index = rng.gen_range(0..self.values.len());
        self.values[index]
    }
}
```

复杂度分析

- 时间复杂度：初始化为 $O(n)$，随机选择为 $O(1)$，其中 n 是链表的元素个数。
- 空间复杂度：$O(n)$。我们需要 $O(n)$ 的空间存储链表中的所有元素。

解法二：水塘抽样法

思路：我们可以设计如下算法：从链表头开始，遍历整个链表，对遍历到的第 i 个节点，随机选择区间 $[0, i)$ 内的一个整数，如果其等于 0，则将答案置为该节点值，否则答案不变。

该算法会保证每个节点的值成为最后被返回的值的概率均为 $\frac{1}{n}$，证明如下：

P(第 i 个节点的值成为最后被返回的值)
=P(第 i 次随机选择的值 =0) $\times$ P(第 i+1 次随机选择的值 $\neq$ 0) $\times \cdots \times$ P(第 n 次随机选择的值 $\neq$ 0)
= $\frac{1}{i} \times$ (1- $\frac{1}{i+1}$) $\times \cdots \times$ (1- $\frac{1}{n}$)
= $\frac{1}{i} \times \frac{i}{i+1} \times \cdots \times \frac{n-1}{n}$
= $\frac{1}{n}$

代码 1.14: 水塘取样法

```rust
use rand::Rng;

struct Solution {
    head: Option<Box<ListNode>>,
}

impl Solution {
    fn new(head: Option<Box<ListNode>>) -> Solution {
        Solution { head }
    }

    fn get_random(&self) -> i32 {
        let mut ans = 0;
        let mut i = 1;
        let mut node = &self.head;
        let mut rng = rand::thread_rng();

        while let Some(n) = node {
            if rng.gen_range(0..i) == 0 {
                ans = n.val;
            }
            i += 1;
            node = &n.next;
        }

        ans
    }
}
```

复杂度分析

- 时间复杂度：初始化为 $O(1)$，随机选择为 $O(n)$，其中 n 是链表的元素个数。
- 空间复杂度：$O(1)$。我们只需要常数的空间保存若干变量。

这道题最重要的收我们要记住一个算法：水塘抽样算法，也叫蓄水池抽样算法，它是一种随机抽样算法，用于从包含 n 个项目的集合 S 中随机选取 k 个样本，其中 n 是一个非常大或未知的数量。这个算法的时间复杂度是 $O(n)$，空间复杂度是 $O(k)$，这个算法的核心思想是：当我们遍历到第 i 个元素时，我们以 $\frac{1}{i}$ 的概率选择这个元素，以 $1-\frac{1}{i}$ 的

概率保持原来的选择。

面试的时候，一旦出这道题，说明面试官相当的刁钻，因为这个算法不是那么常见，但是这个算法的思想是非常重要的，我们可以用这个算法来解决很多问题，比如说：从一个很大的文件中随机抽取一行或者，从一个很大的数据库中随机抽取一条记录或者几条等等。你知道了这个算法就很容易解答，你不知道的这个算法基本上就答不出来了。

1.9 环形链表

No 141 类型 Easy

给你一个链表的头节点 *head* ，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，评测系统内部使用整数 *pos* 来表示链表尾连接到链表中的位置（索引从 0 开始）。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` 。

示例

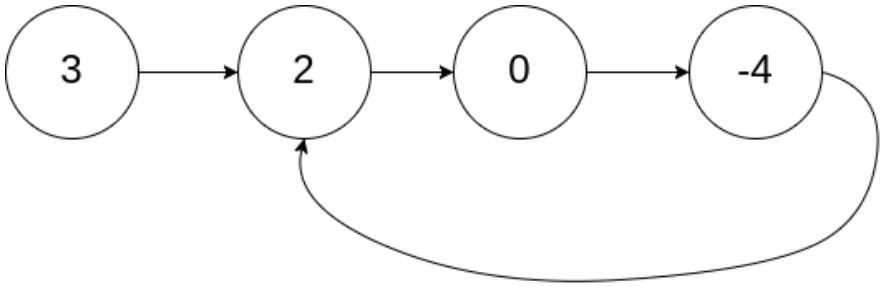


图 1.8. 链表中包含环

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。

解法一：哈希表

思路: 最朴素的算法，我们使用一个哈希表记录访问过的节点，如果在检查某个节点的时候，发现它已经被访问过了，那么说明链表中存在环。

代码 1.15: 使用哈希表检测环

```
1      fn has_cycle(head: Option<Box<ListNode>>) -> bool {
2          let mut seen = std::collections::HashSet::new();
3          let mut node = head;
4
5          while let Some(n) = node {
6              if seen.contains(&n) {
```

```
7                 return true;
8             }
9             seen.insert(n);
10            node = n.next;
11        }
12
13        false
14    }
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是链表中的节点数。最坏情况下我们需要遍历每个节点一次。
- 空间复杂度：$O(N)$，其中 N 是链表中的节点数。主要为哈希表的开销，最坏情况下我们需要将每个节点插入到哈希表中一次。

解法二：快慢指针

思路：本方法需要读者对Floyd 判圈算法（又称龟兔赛跑算法）有所了解。

假想**乌龟**和**兔子**在链表上移动，**兔子**跑得快，**乌龟**跑得慢。当**乌龟**和**兔子**从链表上的同一个节点开始移动时，如果该链表中没有环，那么**兔子**将一直处于**乌龟**的前方；如果该链表中有环，那么**兔子**会先于**乌龟**进入环，并且一直在环内移动。等到**乌龟**进入环时，由于**兔子**的速度快，它一定会在某个时刻与乌龟相遇，即套了**乌龟**若干圈。

我们可以根据上述思路来解决本题。具体地，我们定义两个指针，一快一慢。慢指针每次只移动一步，而快指针每次移动两步。初始时，慢指针在位置 *head*，而快指针在位置 *head.next*。这样一来，如果在移动的过程中，快指针反过来追上慢指针，就说明该链表为环形链表。否则快指针将到达链表尾部，该链表不为环形链表。

为什么我们要规定初始时慢指针在位置 *head*，快指针在位置 *head.next*，而不是两个指针都在位置 *head*（即与**乌龟**和**兔子**中的叙述相同）？
我们使用的是 循环，循环条件先于循环体。由于循环条件一定是判断快慢指针是否重合，如果我们将两个指针初始都置于 *head*，那么 *for* 循环就不会执行。所以我们让快指针先走一步，如果链表中真的有环，两个指针还是会撞上的。

代码 1.16: 使用快慢指针检测环

```
1  pub fn has_cycle(head: Option<Box<ListNode>>) -> bool {
2      if head.is_none() || head.as_ref().unwrap().next.is_none() {
3          return false;
4      }
5      let mut slow = head.clone();
6      let mut fast = head.unwrap().next;
7
8      while fast != slow {
```

```
9           if fast.is_none() || fast.as_ref().unwrap().next.is_none() {
10              return false;
11          }
12          slow = slow.unwrap().next;
13          fast = fast.unwrap().next.unwrap().next;
14      }
15
16      true
17  }
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是链表中的节点数。
- 空间复杂度：$O(1)$。我们只使用了两个指针的额外空间。

这一般是面试官经常想考的一个解答，必须掌握。当然你一旦知道了 Floyd 判圈算法，就一切迎刃而解了。

进一步哈，有一个升级的题目，让你找到进入环的第一个节点。比如下图的链表，环的第一个节点在链表的索引为 1 的位置，返回 1.

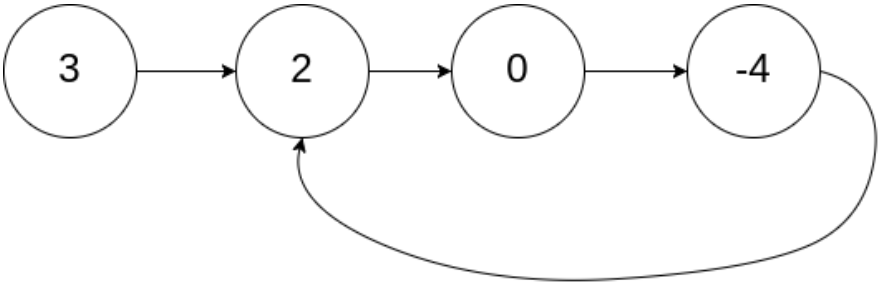


图 1.9. 链表中包含环

这个通过快慢指针也可以巧妙解决。

我们使用两个指针，*fast* 与 *slow*。它们起始都位于链表的头部。随后，*slow* 指针每次向后移动一个位置，而 *fast* 指针向后移动两个位置。如果链表中存在环，则 *fast* 指针最终将再次与 *slow* 指针在环中相遇。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。这可以通过数学推导进行证明，我们在此不再赘述。当然你需要记住这个算法，不知道就很难使用这个算法解答了。

代码 1.17: 使用快慢指针找入环点

```
1       pub fn detect_cycle(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
2           let mut slow = head.clone();
3           let mut fast = head.clone();
4
5           while let Some(f) = fast {
6               slow = slow.unwrap().next;
```

```
            fast = f.next.and_then(|x| x.next);

            if slow == fast {
                let mut p = head.clone();
                while p != slow {
                    p = p.unwrap().next;
                    slow = slow.unwrap().next;
                }
                return p;
            }
        }

        None
    }
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是链表中的节点数。
- 空间复杂度：$O(1)$。我们只使用了三个指针的额外空间。

1.10 相交链表

No. 160 难度 Easy

给定两个单链表的头节点 *headA* 和 *headB* ，请找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 *null* 。

图示两个链表在节点 *c1* 开始相交：

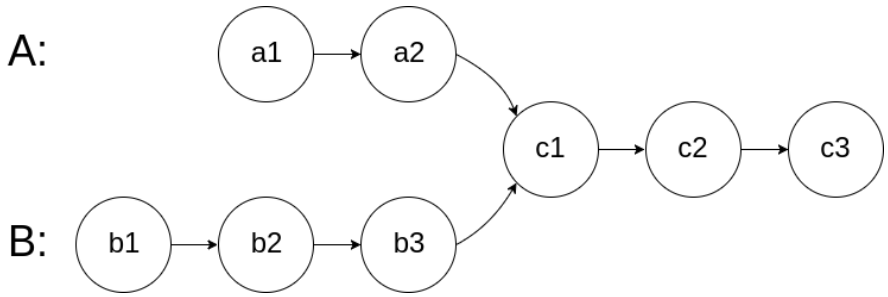


图 1.10. 链表相交

示例

输入:

```
intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
```

输出:

```
Intersected at '8'
```

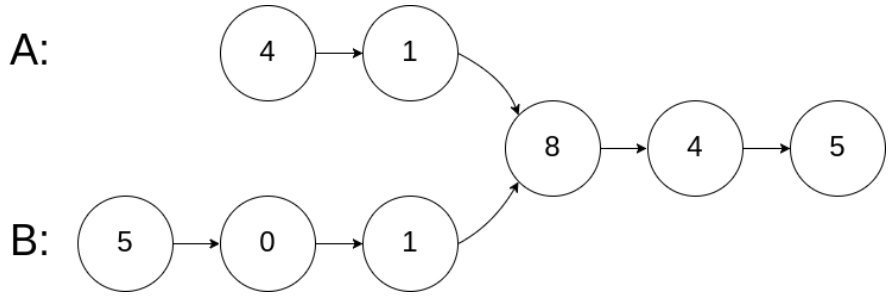


图 1.11. 链表相交示例

解释:

相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

这道题使用哈希表可以容易解决，但是空间复杂度是 $O(m)$, 先遍历 *headA* 链表，把它的节点存储在哈希表中，再遍历 *headB*, 如果哈希表中存在此节点，那就找到了。

我们可以使用双指针法来解决这个问题，空间复杂度是 $O(1)$。

解法一：双指针法

思路: 使用双指针的方法，可以将空间复杂度降至 $O(1)$。

只有当链表 *headA* 和 *headB* 都不为空时，两个链表才可能相交。因此首先判断链表 *headA* 和 *headB* 是否为空，如果其中至少有一个链表为空，则两个链表一定不相交，返回 null。

当链表 *headA* 和 *headB* 都不为空时，创建两个指针 *pA* 和 *pB*，初始时分别指向两个链表的头节点 *headA* 和 *headB*，然后将两个指针依次遍历两个链表的每个节点。具体做法如下：

- 每步操作需要同时更新指针 *pA* 和 *pB*。
- 如果指针 *pA* 不为空，则将指针 *pA* 移到下一个节点；如果指针 *pB* 不为空，则将指针 *pB* 移到下一个节点。
- 如果指针 *pA* 为空，则将指针 *pA* 移到链表 *headB* 的头节点；如果指针 *pB* 为空，则将指针 *pB* 移到链表 *headA* 的头节点。
- 当指针 *pA* 和 *pB* 指向同一个节点或者都为空时，返回它们指向的节点或者 null

通过数学可以简单证明，这种方式下，每个指针都会遍历到所有的节点，所以如果两个链表相交，那么它们一定会在相交的节点相遇。

代码 1.18: 使用哈希表检测环

```
1       fn get_intersection_node(head_a: Option<Box<ListNode>>, head_b: Option<Box<
          ListNode>>) -> Option<Box<ListNode>> {
```

```
        // 两个指针都为 None，不相交
        if head_a.is_none() || head_b.is_none() {
            return None;
        }
        let (mut pa, mut pb) = (head_a, head_b);
        while pa != pb {
            if pa.is_none() { // pa 为空，指向 headb
                pa = head_b.clone();
            } else {
                pa = pa.unwrap().next;
            }
            if pb.is_none() { // pb 为空，指向 heada
                pb = head_a.clone();
            } else {
                pb = pb.unwrap().next;
            }
        }
        pa
    }
```

复杂度分析

- 时间复杂度：$O(m+n)$。
- 空间复杂度：$O(1)$。

1.11 回文链表

No. 234 难度 Easy

给定一个链表的头节点 *head* ，请判断其是否为回文链表。

如果一个链表是回文，那么链表节点序列从前往后看和从后往前看是相同的。

示例

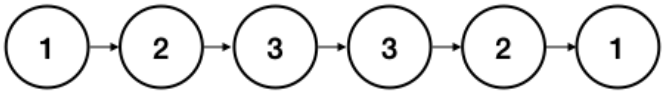


图 1.12. 回文链表

输入: head = $1,2,3,3,2,1$

输出: true

解法一：双指针法

将值复制到数组中后用双指针法。

思路: 确定数组列表是否回文很简单，我们可以使用双指针法来比较两端的元素，并向中间移动。一个指针从起点向中间移动，另一个指针从终点向中间移动。这需要 $O(n)$ 的时间，因为访问每个元素的时间是 $O(1)$，而有 n 个元素要访问。然而同样的方法在链表上操作并不简单，因为不论是正向访问还是反向访问都不是 $O(1)$。而将链表的值复制到数组列表中是 $O(n)$，因此最简单的方法就是将链表的值复制到数组列表中，再使用双指针法判断。

代码 1.19: 使用双指针检测回文

```
fn is_palindrome(head: Option<Box<ListNode>>) -> bool {
    let mut vals = Vec::new();
    let mut cur = head;
    while let Some(node) = cur {
        vals.push(node.val);
        cur = node.next;
    }
    let n = vals.len();
    for (i, &v) in vals.iter().take(n / 2).enumerate() {
        if v != vals[n - 1 - i] {
            return false;
        }
    }
    true
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

这种情况下把指针压如一个栈，然后栈和链表进行比较也可以。使用递归的话也和栈类似，更抽象巧妙，但是空间复杂度依然是 $O(n)$。

解法二: 快慢指针

避免使用 $O(n)$ 额外空间的方法就是改变输入。

思路: 我们可以将链表的后半部分反转（修改链表结构），然后将前半部分和后半部分进行比较。比较完成后我们应该将链表恢复原样。虽然不需要恢复也能通过测试用例，但是使用该函数的人通常不希望链表结构被更改。

该方法虽然可以将空间复杂度降到 $O(1)$，但是在并发环境下，该方法也有缺点。在并发环境下，函数运行时需要锁定其他线程或进程对链表的访问，因为在函数执行过程中链表会被修改。

代码 1.20: 使用快慢指针检测回文

```
// 反转链表
fn reverse_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
    let mut prev = None;
    let mut cur = head;
```

```
        while let Some(mut node) = cur {
            let next_tmp = node.next.take();
            node.next = prev;
            prev = Some(node);
            cur = next_tmp;
        }
        prev
    }

    // 寻找中间节点
    fn end_of_first_half(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let mut fast = head.clone();
        let mut slow = head.clone();
        while let Some(fast_node) = fast {
            if let Some(next_node) = fast_node.next {
                fast = next_node.next;
                slow = slow.unwrap().next;
            } else {
                break;
            }
        }
        slow
    }

    fn is_palindrome(head: Option<Box<ListNode>>) -> bool {
        if head.is_none() {
            return true;
        }

        // 找到前半部分链表的尾节点并反转后半部分链表
        let first_half_end = end_of_first_half(head.clone());
        let second_half_start = reverse_list(first_half_end.clone().unwrap().next
    );

        // 判断是否回文
        let mut p1 = head;
        let mut p2 = second_half_start;
        let mut result = true;
        while result && p2.is_some() {
            if p1.as_ref().unwrap().val != p2.as_ref().unwrap().val {
                result = false;
            }
            p1 = p1.unwrap().next;
            p2 = p2.unwrap().next;
        }

        // 还原链表并返回结果
        first_half_end.unwrap().next = reverse_list(second_half_start);

        result
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

链表是最常见的数据结构之一，面试中经常会出现链表的题目，所以我们需要熟练掌握链表的基本操作，比如反转链表等，再结合快慢指针、双指针等方法技巧性去解答。如果全无思路，不妨使用“最笨”的办法，把链表复制到数组中，使用解数组的方法去解决。

2

双指针

双指针是一种常见的算法技巧，使用两个指针在数组或链表上进行遍历或比较，可以解决一些特定问题。

双指针的常见模式：

- 快慢指针：一个指针走的快，一个走的慢，一般用于链表中查找中间节点或检测环。
- 左右指针：左右两个指针从数组两端向中间移动，一般用于二分查找或判断回文。
- 对撞指针：两个指针相向而行，当两个指针指向同一个元素时，所指元素满足某项性质，一般用于查找两个有序数组的公共元素。

双指针的典型应用：

- 链表中点：快指针一次走两步，慢指针一次走一步，当快指针到链表尾部时，慢指针指向中间。
- 链表中的环：快慢指针起点相同，快指针一次两步，慢指针一次一步，如果相遇了则存在环。
- 二分查找：左右指针指向数组首尾，每次根据中点调整左右指针，直到找到目标。
- 反转链表：前指针指向头节点，后指针从头开始遍历，每次把后指针节点接到前指针之后。
- 合并两个有序数组：左右指针分别指向两个数组头部，比较指向的两个元素大小，依次填入新数组。

在上一章链表中我们已经介绍了一些双指针的应用，这一章我们详细介绍它，拿下它。

我们的题目也由易到难，方便我们逐步掌握它。

2.1 移动零

283 Easy

给定一个数组 *nums*，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意，必须在不复制数组的情况下原地对数组进行操作。

示例

输入: nums = [0,1,0,3,12]
输出: [1,3,12,0,0]

解法一：迭代

思路: 使用双指针，左指针指向当前已经处理好的序列的尾部，右指针指向待处理序列的头部。右指针不断向右移动，每次右指针指向非零数，则将左右指针对应的数交换，同时左指针右移。

注意到以下性质：

1. 左指针左边均为非零数；
2. 右指针左边直到左指针处均为零。

因此每次交换，都是将左指针的零与右指针的非零数交换，且非零数的相对顺序并未改变。*left* 和 *right* 之间的元素都是 0. *left* 和 *right* 之间的隔阂越来越大，不断有 0 横亘在他们中间，*right* 还要负重前行，直到走到终点。

代码 2.1: 移动零

```rust
fn move_zeroes(nums: &mut [i32]) {
    let mut left = 0;
    let mut right = 0;
    let n = nums.len();

    while right < n {
        if nums[right] != 0 {
            nums.swap(left, right);
            left += 1;
        }
        right += 1;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为序列长度。每个位置至多被遍历两次。
- 空间复杂度：$O(1)$。只需要常数的空间存放若干变量。

2.2 部分排序

No. 16.16 难度 Medium

给定一个整数数组，编写一个函数，找出索引 m 和 n，只要将索引区间 $[m,n]$ 的元素排好序，整个数组就是有序的。注意：$n-m$ 尽量最小，也就是说，找出符合条件的最短序列。函数返回值为 $[m,n]$，若不存在这样的 m 和 n（例如整个数组是有序的），请返回 $[-1,-1]$。

示例

输入: [1,2,4,7,10,11,7,12,6,7,16,18,19]
输出: [3,9]

解法一：双指针

思路: 题目没说最终的数组是递增还是递减，我们默认数列是递增的。

那么对于元素 $a[i]$ 来说，如果它左边存在大于 $a[i]$ 的元素，那么 $a[i]$ 必须在排序区间，一定要参与到排序里去的。或者说如果它右边存在小于 $a[i]$ 的元素，那么 $a[i]$ 也是要参与到排序里去的。

所以我们只需要寻找最靠右的那个数（满足左边存在大于它的数），和最靠左的那个数（满足右边存在小于它的数），那么这两个数之间就是要排序的区间了。

为什么最靠右的那个（满足左边存在大于它的数）数一定能保证右边没有更小的数了呢？因为如果右边还有更小的数，那么那个更小的数才是更靠右的啊，这就矛盾了。

所以我们只需要从左到右扫描一遍，用一个变量维护一下最大值就行了，然后反向再遍历一遍，维护一个最小值。

代码 2.2: 部分排序-双指针

```rust
    fn sub_sort(array: &[i32]) -> [i32; 2] {
        let n = array.len();

        // Maximum value
        let mut max_value = i32::MIN;
        // Minimum value
        let mut min_value = i32::MAX;

        let mut left = -1; // Left boundary
        let mut right = -1; // Right boundary

        for i in 0..n {
            if array[i] < max_value { // Find the right boundary
                right = i as i32;
            } else {
                max_value = array[i]; // Update the maximum value
            }
        }
        for i in (0..n).rev() {
```

```
            if array[i] > min_value { // Find the left boundary
                left = i as i32;
            } else {
                min_value = array[i]; // Update the minimum value
            }
        }

        [left, right]
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

这道题是和最短无序连续子数组一样的。

2.3 数对和

No. 16.24 类型 Medium

设计一个算法，找出数组中两数之和为指定值的所有整数对。一个数只能属于一个数对。

示例 1

输入: nums = [5,6,5], target = 11
输出: [[5,6]]

示例 1

输入: nums = [5,6,5,6], target = 11
输出: [[5,6],[5,6]]

解法一：双指针

思路: 先对数组进行排序处理。

代码 2.3: 部分排序-双指针

```
use std::collections::HashSet;

fn pair_sums(nums: Vec<i32>, target: i32) -> Vec<Vec<i32>> {
    let mut ans = Vec::new();
    let mut set = HashSet::new();

    for num in nums {
        let complement = target - num;
        if set.contains(&complement) {
```

```
10                  ans.push(vec![complement, num]);
11                  set.remove(&complement);
12              } else {
13                  set.insert(num);
14              }
15          }
16
17          ans
18      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

类似的题目还有最小差等。

2.4 盛最多水的容器

No. 11 难度 Medium

给定一个长度为 n 的整数数组 *height* 。有 n 条垂线，第 i 条线的两个端点是 $(i, 0)$ 和 $(i, height[i])$ 。

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。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

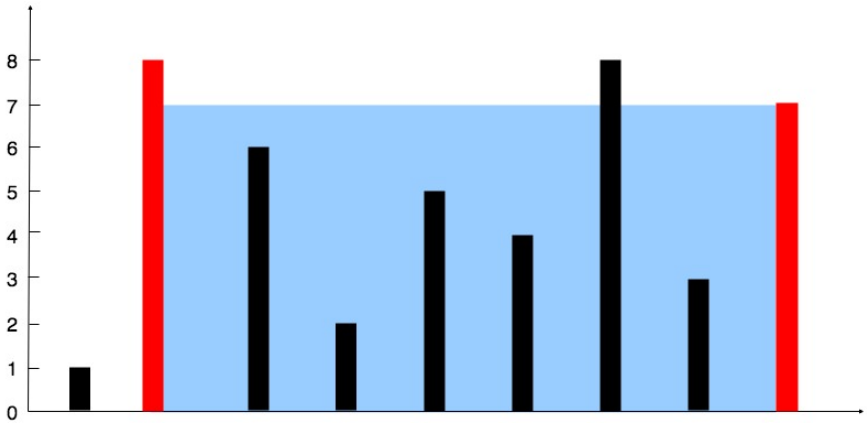


图 2.1. 盛水的容器

示例

输入: nums = [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。

解法一：双指针

思路: 设两指针 i , j ，指向的水槽板高度分别为 $h[i]$, $h[j]$ ，此状态下水槽面积为 $S(i,j)$。由于可容纳水的高度由两板中的 短板决定，因此可得如下面积公式：

$$S(i,j) = min(h[i],h[j]) \times (j-i)$$

在每个状态下，无论长板或短板向中间收窄一格，都会导致水槽底边宽度 -1 变短：

- 若向内移动短板，水槽的短板 $min(h[i],h[j])$ 可能变大，因此下个水槽的面积可能增大。
- 若向内移动长板，水槽的短板 $min(h[i],h[j])$ 不变或变小，因此下个水槽的面积一定变小。

因此，初始化双指针分列水槽左右两端，循环每轮将短板向内移动一格，并更新面积最大值，直到两指针相遇时跳出；即可获得最大面积。

算法流程：

1. 初始化：双指针 i , j 分列水槽左右两端；
2. 循环收窄：直至双指针相遇时跳出；
3. 更新面积最大值 res；
4. 选定两板高度中的短板，向中间收窄一格；
5. 返回值：返回面积最大值 res 即可；

代码 2.4: 盛水最多的容器

```rust
fn max_area(height: Vec<i32>) -> i32 {
    let mut i = 0;
    let mut j = height.len() - 1;
    let mut res = 0;

    while i < j {
        if height[i] < height[j] {
            res = res.max((j - i) * height[i]);
            i += 1;
        } else {
            res = res.max((j - i) * height[j]);
            j -= 1;
        }
    }

    res
}
```

复杂度分析

- 时间复杂度：$O(N)$，双指针总计最多遍历整个数组一次。
- 空间复杂度：$O(1)$，只需要额外的常数级别的空间。

2.5 接雨水

No 42 难度 Hard

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

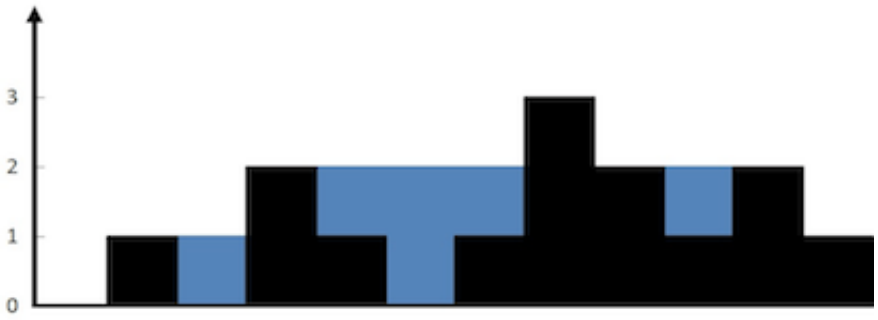


图 2.2. 盛水的容器

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）。

解法一：双指针

思路: 注意到下标 i 处能接的雨水量由 *leftMax*[i] 和 *rightMax*[i] 中的最小值决定。由于数组 *leftMax* 是从左往右计算，数组 *rightMax* 是从右往左计算，因此可以使用双指针和两个变量代替两个数组。

维护两个指针 *left* 和 *right*，以及两个变量 *leftMax* 和 *rightMax*，初始时 *left*=0,*right*=n-1,*leftMax*=0,*rightMax*=0。指针 *left* 只会向右移动，指针 *right* 只会向左移动，在移动指针的过程中维护两个变量 *leftMax* 和 *rightMax* 的值。

当两个指针没有相遇时，进行如下操作：

- 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 $leftMax - height[eft]$，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）。

当两个指针相遇时，即可得到能接的雨水总量。

代码 2.5: 接雨水

```rust
fn trap(height: Vec<i32>) -> i32 {
    let mut left = 0;
    let mut right = height.len() - 1;
    let mut left_max = 0;
    let mut right_max = 0;
    let mut ans = 0;

    while left < right {
        left_max = left_max.max(height[left]);
        right_max = right_max.max(height[right]);

        if height[left] < height[right] {
            ans += left_max - height[left];
            left += 1;
        } else {
            ans += right_max - height[right];
            right -= 1;
        }
    }

    ans
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

2.6 颜色分类

No. 75 难度 Medium

给定一个包含红色、白色和蓝色、共 n 个元素的数组 $nums$，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。

我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

必须在不使用库内置的 $sort$ 函数的情况下解决这个问题。

示例

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

解法一：双指针

我们可以考虑使用单指针，对数组进行两次遍历。在第一次遍历中，我们将数组中所有的 0 交换到数组的头部。在第二次遍历中，我们将数组中所有的 1 交换到头部的 0 之后。此时，所有的 2 都出现在数组的尾部，这样我们就完成了排序。

我们是否可以仅使用一次遍历呢？我们可以额外使用一个指针，即使用两个指针分别用来交换 0 和 1。可以的，我们使用双指针法。

思路：我们用指针 p_0 来交换 0，p_1 来交换 1，初始值都为 0。当我们从左向右遍历整个数组时：

- 如果找到了 1，那么将其与 $nums[p_1]$ 进行交换，并将 p_1 向后移动一个位置；
- 如果找到了 0，那么将其与 $nums[p_0]$ 进行交换，并将 p_0 向后移动一个位置。这样做是正确的吗？我们可以注意到，因为连续的 0 之后是连续的 1，因此如果我们将 0 与 $nums[p_0]$ 进行交换，那么我们可能会把一个 1 交换出去。当 $p_0 < p_1$ 时，我们已经将一些 1 连续地放在头部，此时一定会把一个 1 交换出去，导致答案错误。因此，如果 $p_0 < p_1$，那么我们需要再将 $nums[i]$ 与 $nums[p_1]$ 进行交换，其中 i 是当前遍历到的位置，在进行了第一次交换后，$nums[i]$ 的值为 1，我们需要将这个 1 放到「头部」的末端。在最后，无论是否有 $p_0 < p_1$，我们需要将 p_0 和 p_1 均向后移动一个位置，而不是仅将 p_0 向后移动一个位置。

代码 2.6: 颜色分类

```rust
fn sort_colors(nums: &mut [i32]) {
    let mut p0 = 0;
    let mut p1 = 0;
    for i in 0..nums.len() {
        if nums[i] == 0 {
            nums.swap(i, p0);
            if p0 < p1 {
                nums.swap(i, p1);
            }
            p0 += 1;
            p1 += 1;
        } else if nums[i] == 1 {
            nums.swap(i, p1);
            p1 += 1;
        }
    }
}
```

当然你也可以使用 p_0 和 p_2 来交换 0 和 2，这样也可以得到一个完整的排序。

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

3

滑动窗口

滑动窗口算法是一种解决数组或字符串处理问题的有效技巧。它通过使用两个指针定义一个窗口，并沿着数组或字符串移动窗口来解决问题。

滑动窗口算法基于以下两个指针：

- 左指针 `left`：定义窗口的左边界
- 右指针 `right`：定义窗口的右边界

窗口内的数据表示当前关注的部分。窗口的大小可以固定，也可以根据问题需求进行调整。算法逻辑如下：

1. 初始化 `left` 和 `right` 指针
2. 将 `right` 指针向右移动，扩大窗口
3. 根据问题需求调整窗口内数据
4. 将 `left` 指针向右移动，缩小窗口
5. 重复步骤 2 到 4，直到 `right` 到达数组/字符串末尾

通过上述流程，窗口在数组/字符串上滑动，解决问题的关键信息就包含在这个滑动窗口中。

滑动窗口算法可以解决许多字符串和数组处理问题，例如：

- 找到字符串中最长的无重复字符子串
- 找到字符串中所有字母异位词
- 找到子数组的最大和

3.1 找到字符串中最长的无重复字符子串

No 3 难度 Medium

给定一个字符串 s ，请你找出其中不含有重复字符的 **最长子串** 的长度。

子串 (Substring) 是指字符串中连续的字符组成的片段。例如, 对于字符串"abcxyz", 它的子串有"bcx"、"xyz" 等。子串必须是连续的。

而**子序列** (Subsequence) 并不要求字符是连续的, 只要是字符串中保持相对顺序的字符组成的序列即可。例如, 对于同样的字符串"abcxyz", 它的子序列有"ayz"、"xc" 等。子序列允许字符间有间隔。

总结一下:

- 子串必须是连续的, 子序列可以是不连续的。
- 子串是字符串的一部分, 子序列不必是字符串的一部分。
- 任何子串都可以看成是一个子序列, 但是子序列不一定是子串。

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。

解法一: 滑动窗口

思路: 我们先用一个例子考虑如何在较优的时间复杂度内通过本题。

我们不妨以示例一中的字符串 `abcabcbbabcabcbbabcabcbb` 为例，找出从每一个字符开始的，不包含重复字符的最长子串，那么其中最长的那个字符串即为答案。对于示例一中的字符串，我们列举出这些结果，其中括号中表示选中的字符以及最长的字符串:

- 以 `(a)bcabcbb` 开始的最长字符串为 `(abc)abcbb`;
- 以 `a(b)cabcbb` 开始的最长字符串为 `a(bca)bcbb`;
- 以 `ab(c)abcbb` 开始的最长字符串为 `ab(cab)cbb`;
- 以 `abc(a)bcbb` 开始的最长字符串为 `abc(abc)bb`;

- 以 `abca(b)cbb` 开始的最长字符串为 `abca(bc)bb`；
- 以 `abcab(c)bb` 开始的最长字符串为 `abcab(cb)b`；
- 以 `abcabc(b)b` 开始的最长字符串为 `abcabc(b)b`；
- 以 `abcabcb(b)` 开始的最长字符串为 `abcabcb(b)`。

发现了什么？如果我们依次递增地枚举子串的起始位置，那么子串的结束位置也是递增的！这里的原因在于，假设我们选择字符串中的第 k 个字符作为起始位置，并且得到了不包含重复字符的最长子串的结束位置为 r_k。那么当我们选择第 $k+1$ 个字符作为起始位置时，首先从 $k+1$ 到 r_k 的字符显然是不重复的，并且由于少了原本的第 k 个字符，我们可以尝试继续增大 r_k，直到右侧出现了重复字符为止。

这样一来，我们就可以使用**滑动窗口**来解决这个问题了：

- 我们使用两个指针表示字符串中的某个子串（或窗口）的左右边界，其中左指针代表着上文中「枚举子串的起始位置」，而右指针即为上文中的 r_k；
- 在每一步的操作中，我们会将左指针向右移动一格，表示**我们开始枚举下一个字符作为起始位置**，然后我们可以不断地向右移动右指针，但需要保证这两个指针对应的子串中没有重复的字符。在移动结束后，这个子串就对应着**以左指针开始的，不包含重复字符的最长子串**。我们记录下这个子串的长度；
- 在枚举结束后，我们找到的最长的子串的长度即为答案。

在上面的流程中，我们还需要使用一种数据结构来判断是否有重复的字符，常用的数据结构为哈希集合（map 类型）。在左指针向右移动的时候，我们从哈希集合中移除一个字符，在右指针向右移动的时候，我们往哈希集合中添加一个字符。

下面是具体的实现。

代码 3.1: 滑动窗口法

```rust
fn length_of_longest_substring(s: String) -> i32 {
    let mut m: std::collections::HashMap<char, i32> = std::collections::HashMap::new();
    let n = s.len();
    let mut rk = -1;
    let mut ans = 0;

    for i in 0..n {
        if i != 0 {
            m.remove(&s.chars().nth(i - 1).unwrap());
        }
        while rk + 1 < n && !m.contains_key(&s.chars().nth(rk + 1).unwrap()) {
            m.insert(s.chars().nth(rk + 1).unwrap(), 1);
            rk += 1;
        }
        ans = ans.max(rk - i + 1);
    }

```

```
18          ans
19      }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是字符串的长度。左指针和右指针分别会遍历整个字符串一次。
- 空间复杂度：$O(|\Sigma|)$，其中 Σ 表示字符集（即字符串中可以出现的字符）

3.2 最短超串

No. 17.18 难度 Medium

假设你有两个数组，一个长一个短，短的元素均不相同。找到长数组中包含短数组所有的元素的最短子数组，其出现顺序无关紧要。返回最短子数组的左端点和右端点，如有多个满足条件的子数组，返回左端点最小的一个。若不存在，返回空数组。

示例 1
输入:
big = [7,5,9,0,2,1,3,5,7,9,1,1,5,8,8,9,7]
small = [1,5,9] 输出: [7,10]

示例 2
输入:
big = [1,2,3]
small = [4] 输出: []

解法一：滑动窗口

思路:

- 将 small 数组中的数存在 map 中，其 value 初始化为-1
- 遍历 big 数组，map 存储 small 数组中每一个值在 big 数组中的位置，并更新
- 当在 big 数组中找齐了所有在 small 数组中的数字后，就用当前下标 i 减去 map 中 value 的最小值（即位置的最小值），得到的差即为“包含短数组所有的元素的子数组长度”

代码 3.2: 滑动窗口法

```
1       fn shortest_seq(big: Vec<i32>, small: Vec<i32>) -> Vec<i32> {
2           // Exception check
3           if small.len() > big.len() {
4               return vec![];
5           }
6
7           // Save the occurrence of elements in the small array, value is the index
```

```rust
        let mut small_map: std::collections::HashMap<i32, i32> = std::collections
    ::HashMap::new();
        let mut count = small.len();
        let mut ans: Vec<i32> = vec![];

        // Initialize small_map, elements have not appeared yet
        for i in small {
            small_map.insert(i, -1);
        }

        for i in 0..big.len() {
            // Check condition
            if small_map.contains_key(&big[i]) {
                if small_map[&big[i]] == -1 {
                    count -= 1; // An element has appeared
                }
                small_map.insert(big[i], i as i32); // Always use the latest
    index
            }

            // If count <= 0, it means that the big array already contains all
    the elements in the small array
            if count <= 0 {
                let min_num = get_min(&small_map);
                if ans.is_empty() || i as i32 - min_num + 1 < ans[1] - ans[0] + 1
     {
                    ans = vec![min_num, i as i32];
                }
            }
        }

        if count > 0 {
            return vec![];
        }

        ans
    }

    // Get the minimum value in small_map
    fn get_min(small_map: &std::collections::HashMap<i32, i32>) -> i32 {
        let mut min_num = std::i32::MAX;
        for &v in small_map.values() {
            if v < min_num {
                min_num = v;
            }
        }
        min_num
    }
```

好吧，你可以说这个不算滑动窗口，没看到窗口的存在。但是，这个解法的思路和滑动窗口法是一样的，只是没有显式地使用窗口。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是字符串的长度。左指针和右指针分别会遍历整个字符串一次。
- 空间复杂度：$O(small)$，其中 $small$ 表示短数组的长度。

那么我们来一个明显使用滑动窗口的例子：

代码 3.3: 纯滑动窗口解法

```rust
fn shortest_seq(big: Vec<i32>, small: Vec<i32>) -> Vec<i32> {
    let n = big.len();
    // 结果
    let mut ans: Vec<i32> = Vec::with_capacity(2);

    // 记录滑动窗口内需要覆盖的数字，及其对应的个数
    let mut need: std::collections::HashMap<i32, i32> = std::collections::HashMap::new();
    // 记录滑动窗口一共需要覆盖的数字个数
    let mut diff = small.len();
    let mut min_len = n;

    // 初始化 need，统计需要覆盖的数字有多少个
    for e in small {
        *need.entry(e).or_insert(0) += 1;
    }

    // l 指向窗口左边界，r 指向窗口右边界
    let (mut l, mut r) = (0, 0);

    // 滑动窗口
    while r < n {
        // 如果此数字在 small 数组中
        if let Some(count) = need.get_mut(&big[r]) {
            *count -= 1; // 滑动窗口内的此数字的需求数减一，有可能是负数，代表此数字供应充足，很富裕
            if *count >= 0 {
                diff -= 1;
            }
        }

        // 滑动窗口内已经包含了 small 数组中的所有元素
        while diff == 0 {
            if r - l < min_len {
                min_len = r - l;
                ans = vec![l as i32, r as i32];
            }
            // 如果 big[l] 在 small 数组中
            if let Some(count) = need.get_mut(&big[l]) {
                *count += 1; // 滑动窗口内的此数字的需求数加一
                if *count > 0 {
                    diff += 1;
                }
            }
            l += 1;
        }
```

```
45
46              r += 1;
47          }
48
49          ans
50      }
```

3.3 最小覆盖子串

No. 76 难度 Hard

给定两个字符串 s 和 t。返回 s 中包含 t 的所有字符的最短子字符串。如果 s 中不存在符合条件的子字符串，则返回空字符串"" 。如果 s 中存在多个符合条件的子字符串，返回任意一个。

注意：对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。

示例

输入: s = "ADOBECODEBANC", t = "ABC"
输出: "BANC"
解释: "BANC" 包含了字符串 t 的所有字符'A'、'B'、'C'

解法一：滑动窗口

思路: 本问题要求我们返回字符串 s 中包含字符串 t 的全部字符的最小窗口。我们称包含 t 的全部字母的窗口为「可行」窗口。

我们可以用滑动窗口的思想解决这个问题。在滑动窗口类型的问题中都会有两个指针，一个用于「延伸」现有窗口的 r 指针，和一个用于「收缩」窗口的 l 指针。在任意时刻，只有一个指针运动，而另一个保持静止。我们在 s 上滑动窗口，通过移动 r 指针不断扩张窗口。当窗口包含 t 全部所需的字符后，如果能收缩，我们就收缩窗口直到得到最小窗口。

如何判断当前的窗口包含所有 t 所需的字符呢？我们可以用一个哈希表表示 t 中所有的字符以及它们的个数，用一个哈希表动态维护窗口中所有的字符以及它们的个数，如果这个动态表中包含 t 的哈希表中的所有字符，并且对应的个数都不小于 t 的哈希表中各个字符的个数，那么当前的窗口是「可行」的。

注意：这里 t 中可能出现重复的字符，所以我们要记录字符的个数。

代码 3.4: 最小覆盖子串

```rust
fn min_window(s: String, t: String) -> String {
    use std::collections::HashMap;
    let mut ori: HashMap<u8, i32> = HashMap::new();
    let mut cnt: HashMap<u8, i32> = HashMap::new();

```

```rust
        for b in t.bytes() {
            *ori.entry(b).or_insert(0) += 1;
        }

        let s_bytes = s.as_bytes();
        let mut len = std::i32::MAX;
        let mut ans_l = -1;
        let mut ans_r = -1;

        let check = || -> bool {
            for (k, v) in &ori {
                if cnt.get(k).unwrap_or(&0) < v {
                    return false;
                }
            }
            true
        };

        let mut l = 0;
        let mut r = 0;
        while r < s_bytes.len() {
            if let Some(v) = ori.get(&s_bytes[r]) {
                *cnt.entry(s_bytes[r]).or_insert(0) += 1;
            }

            while check() && l <= r {
                if r - l + 1 < len {
                    len = r - l + 1;
                    ans_l = l as i32;
                    ans_r = (l + len) as i32;
                }

                if let Some(v) = ori.get(&s_bytes[l]) {
                    *cnt.entry(s_bytes[l]).or_insert(0) -= 1;
                }

                l += 1;
            }

            r += 1;
        }

        if ans_l == -1 {
            return String::new();
        }

        s[ans_l as usize..ans_r as usize].to_string()
    }
```

复杂度分析

- 时间复杂度：$O(C\ s\ +\ t\ )$，其中 C 字符集大小。

- 空间复杂度：$O(C)$。

4

数组

数组是算法中最基础和常用的数据结构之一。数组可以存储同一类型的数据，并且可以通过索引来快速访问每个数据元素。数组由于内存布局连续，可以利用索引随机访问任意元素，算法中常会利用这一特性，进行遍历、搜索、访问元素等操作。但数组大小固定，插入删除元素需要数据搬移，所以经常需要权衡使用数组还是链表。

4.1 合并两个有序数组

No. 88 难度 Easy

给你两个按 **非递减顺序** 排列的整数数组 `nums1` 和 `nums2`，另有两个整数 m 和 n，分别表示 `nums1` 和 `nums2` 中的元素数目。

请你合并 `nums2` 到 `nums1` 中，使合并后的数组同样按非递减顺序排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 `nums1` 中。为了应对这种情况，`nums1` 的初始长度为 $m + n$，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。`nums2` 的长度为 n。

示例

输入: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
输出: [1,2,2,3,5,6]
解释: 需要合并 [1,2,3] 和 [2,5,6] 。
合并结果是 [1,2,2,3,5,6] ，其中斜体加粗标注的为 nums1 中的元素。

解法一：逆向双指针

注意题目中 `nums1` 的初始长度为 $m+n$，已经分配好了空间。

思路：既然 `nums1` 的后半部分是空的，可以直接覆盖而不会影响结果。因此可以指针设置为从后向前遍历，每次取两者之中的较大者放进 `nums1` 的最后面。

代码 4.1: 合并两个有序数组

```rust
fn merge(nums1: &mut Vec<i32>, m: i32, nums2: &mut Vec<i32>, n: i32) {
    let mut p1 = m - 1;
    let mut p2 = n - 1;
    let mut tail = m + n - 1;

    while p1 >= 0 || p2 >= 0 {
        let cur;
        if p1 == -1 {
            cur = nums2[p2 as usize];
            p2 -= 1;
        } else if p2 == -1 {
            cur = nums1[p1 as usize];
            p1 -= 1;
        } else if nums1[p1 as usize] > nums2[p2 as usize] {
            cur = nums1[p1 as usize];
            p1 -= 1;
        } else {
            cur = nums2[p2 as usize];
            p2 -= 1;
        }
        nums1[tail as usize] = cur;
        tail -= 1;
    }
}
```

复杂度分析

- 时间复杂度：$O(m+n)$。
- 空间复杂度：$O(1)$。

4.2 删除有序数组的重复项

No. 26 难度 Easy

给你一个有序数组 `nums` ，请你**原地**删除重复出现的元素，使得出现次数超过两次的元素只出现两次，返回删除后数组的新长度。

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

输入: nums = [1,1,1,2,2,3]
输出: 5, nums = [1,1,2,2,3] 输出: 函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1, 2, 2, 3。不需要考虑数组中超出新长度后面的元素。

解法一: 双指针

思路: 因为给定数组是有序的，所以相同元素必然连续。我们可以使用双指针解决本题，遍历数组检查每一个元素是否应该被保留，如果应该被保留，就将其移动到指定位置。具体地，我们定义两个指针 *slow* 和 *fast* 分别为慢指针和快指针，其中慢指针表示处理后结果的数组的长度，快指针表示已经检查过的数组的长度，即 *nums*[*fast*] 表示待检查的第一个元素，*nums*[*slow* - 1] 为上一个应该被保留的元素所移动到的指定位置。

因为本题要求相同元素最多出现两次而非一次，所以我们需要检查上上个应该被保留的元素 $nums[slow-2]$ 是否和当前待检查元素 $nums[fast]$ 相同。当且仅当 $nums[slow-2]=nums[fast]$ 时，当前待检查元素 $nums[fast]$ 不应该被保留（因为此时必然有 $nums[slow-2]=nums[slow-1]=nums[fast]$。最后，$slow$ 即为处理好的数组的长度。

特别地，数组的前两个数必然可以被保留，因此对于长度不超过 2 的数组，我们无需进行任何处理，对于长度超过 2 的数组，我们直接将双指针的初始值设为 2 即可。

代码 4.2: 删除有序数组的重复项

```rust
fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {
    let n = nums.len();
    if n <= 2 {
        return n as i32;
    }
    let mut slow = 2;
    let mut fast = 2;
    while fast < n {
        if nums[slow - 2] != nums[fast] {
            nums[slow] = nums[fast];
            slow += 1;
        }
        fast += 1;
    }
    return slow as i32;
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

4.3 轮转数组

No. 189 难度 Medium

给定一个整数数组 `nums`，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。

示例

输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]

解法一：数组反转

思路: 该方法基于如下的事实：当我们将数组的元素向右移动 k 次后，尾部 $k \bmod n$ 个元素会移动至数组头部，其余元素向后移动 $k \bmod n$ 个位置。

该方法为数组的翻转：我们可以先将所有元素翻转，这样尾部的 $k \bmod n$ 个元素就被移至数组头部，然后我们再翻转 $[0, k \bmod n - 1]$ 区间的元素和 $[k \bmod n, n - 1]$ 区间的元素即能得到最后的答案。

我们以 $n = 7$ $k = 3$ 为例进行如下展示：| 操作 | 结果 | |–|–| | 原始数组 |1,2,3,4,5,6,7| | 翻转所有元素 |7,6,5,4,3,2,1| | 翻转 $[0, k \bmod n - 1]$ 区间的元素 |5,6,7,4,3,2,1| | 翻转 $[k \bmod n, n - 1]$ 区间的元素 |5,6,7,1,2,3,4|

代码 4.3: 轮转数组

```rust
// BEGIN: ed8c6549bwf9
fn reverse(a: &mut [i32]) {
    let n = a.len();
    for i in 0..n/2 {
        a.swap(i, n-1-i);
    }
}

fn rotate(nums: &mut [i32], k: usize) {
    let k = k % nums.len();
    reverse(nums);
    reverse(&mut nums[..k]);
    reverse(&mut nums[k..]);
}
// END: ed8c6549bwf9
```

复杂度分析

- 时间复杂度：$O(2n)$。
- 空间复杂度：$O(1)$。

4.4 加油站

No. 134 难度 Medium

在一条环路上有 n 个加油站，其中第 i 个加油站有汽油 $gas[i]$ 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 $i+1$ 个加油站需要消耗汽油 $cost[i]$ 升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 gas 和 cos ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1 。如果存在解，则 保证它是 唯一的。

示例 1
输入: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
输出:3
输出: 从 3 号加油站 (索引为 3 处) 出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。因此，3 可为起始索引。

示例 2
输入: gas = [2,3,4], cost = [3,4,3]
输出:-1
输出: 你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。我们从 2 号加油站出发，可以获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。因此，无论怎样，你都不可能绕环路行驶一周。

解法一：遍历

思路: 最容易想到的解法是：从头到尾遍历每个加油站，并检查以该加油站为起点，最终能否行驶一周。我们可以通过减小被检查的加油站数目，来降低总的时间复杂度。

假设我们此前发现，从加油站 x 出发，每经过一个加油站就加一次油（包括起始加油站），最后一个可以到达的加油站是 y（不妨设 $x<y$）。这就说明：站 x 出发，每经过一个加油站就加一次油（包括起始加油站），最后一个可以到达的加油站是 y（不妨设 $x<y$）。这就说明：

$$\sum_{i=x}^{y} gas[i] < \sum_{i=x}^{y} cost[i]$$

$$\sum_{i=x}^{j} gas[i] \geqslant \sum_{i=x}^{j} cost[i](Forall j \in [x,y))$$

第一个式子表明无法到达加油站 y 的下一个加油站，第二个式子表明可以到达 y 以及 y 之前的所有加油站。

现在，考虑任意一个位于 x,y 之间的加油站 z（包括 x 和 y），我们现在考察从该加油站出发，能否到达加油站 y 的下一个加油站，也就是要判断 $\sum_{i=z}^{y} gas[i]$ 与 $\sum_{i=z}^{y} cost[i]$ 之间的大小关系。

推导一下：$\sum_{i=z}^{y}$ gas[i]= $\sum_{i=x}^{y}$ gas[i]- $\sum_{i=x}^{z-1}$ gas[i] < $\sum_{i=x}^{y}$ cos t[i]- $\sum_{i=x}^{z-1}$ gas[i] < $\sum_{i=x}^{y}$ cos t[i]- $\sum_{i=x}^{z-1}$ cos t[i] cos t[i]

从上面的推导中，能够得出结论：从 x, y 之间的任何一个加油站出发，都无法到达加油站 y 的下一个加油站。

代码 4.4: 加油站

```rust
fn can_complete_circuit(gas: Vec<i32>, cost: Vec<i32>) -> i32 {
    let n = gas.len();
    // Iterate through each gas station
    for i in 0..n {
        // Initialize variables for gas and cost, as well as the count of gas stations
        let (mut sum_of_gas, mut sum_of_cost, mut cnt) = (0, 0, 0);
        // Start circling the route from gas station i
        while cnt < n {
            // Calculate the current station index
            let j = (i + cnt) % n;
            // Calculate the gas and cost at the current gas station
            sum_of_gas += gas[j];
            sum_of_cost += cost[j];
            // If the total cost is greater than the total gas, break the loop
            if sum_of_cost > sum_of_gas {
                break;
            }
            // Continue to the next gas station
            cnt += 1;
        }
        // If we have visited all gas stations, return the starting index
        if cnt == n {
            return i as i32;
        } else {
            // If not, try checking from the first unreachable gas station
            // Note that we optimize here because neither i nor i+cnt can complete a full circle
            i += cnt;
        }
    }
    // Not found
    -1
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

这道题如果没有上面的推导，很难想到这种解法，可能就是暴力遍历了。

5

字符串

字符串是计算机科学中常见的数据结构，在很多算法问题中都会涉及对字符串的操作。下面几个方面比较重要：

1. 字符串匹配算法
用于在文本中查找是否存在指定的模式字符串，典型的有 KMP 算法、Rabin-Karp 算法等。应用场景例如文字处理、信息检索等。

2. 最长公共子串/子序列
求两个字符串的最长公共子串或子序列，可以使用动态规划设计算法。应用场景有基因序列分析等。

3. 字符串编辑距离
计算将一个字符串转换成另一个字符串需要的最少编辑操作次数，典型算法是 Levenshtein 距离，应用在拼写检查等场景。

4. 字符串排序
对字符串数组进行排序，可以使用归并排序、快速排序等修改版算法。

5. 字符串数据压缩
利用字符串模式的重复和统计特点，可以设计压缩算法对字符串进行编码，例如游程编码、Huffman 编码等。

6. 字符串搜索算法
在大量文本数据中搜索字符串的算法，例如后缀树、后缀数组等。应用在全文索引和搜索中。

5.1 最长公共前缀

No 14 难度 Easy

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串""。

示例 1
输入: strs = ["flower","flow","flight"]
输出: "fl"

示例 2
输入: strs = ["dog","racecar","car"]
输出: ""

解法一: 横向扫描

思路: 用 $LCP(S_1 \dots S_n)$ 表示字符串 $S_1 \dots S_n$ 的最长公共前缀。

可以得到以下结论:

$$LCP(S1 \dots Sn) = LCP(LCP(LCP(S1, S2), S3), \dots Sn)$$

基于该结论，可以得到一种查找字符串数组中的最长公共前缀的简单方法。

依次遍历字符串数组中的每个字符串，对于每个遍历到的字符串，更新最长公共前缀，当遍历完所有的字符串以后，即可得到字符串数组中的最长公共前缀。

如果在尚未遍历完所有的字符串时，最长公共前缀已经是空串，则最长公共前缀一定是空串，因此不需要继续遍历剩下的字符串，直接返回空串即可。

代码 5.1: 最长公共前缀

```rust
fn longest_common_prefix(strs: Vec<String>) -> String {
    if strs.is_empty() {
        return String::new();
    }
    let mut prefix = strs[0].clone();
    let count = strs.len();
    for i in 1..count {
        prefix = lcp(&prefix, &strs[i]);
        if prefix.is_empty() {
            break;
        }
    }
    prefix
}

fn lcp(str1: &str, str2: &str) -> String {
    let length = str1.len().min(str2.len());
```

```
18          let mut index = 0;
19          while index < length && str1.chars().nth(index) == str2.chars().nth(index
            ) {
20              index += 1;
21          }
22          str1[..index].to_string()
23      }
```

复杂度分析

- 时间复杂度：$O(mn)$。
- 空间复杂度：$O(1)$。

另外一种方法就是纵向扫描，从前往后遍历所有字符串的每一列，比较相同列上的字符是否相同，如果相同则继续对下一列进行比较，如果不相同则当前列不再属于公共前缀，当前列之前的部分为最长公共前缀。方法类似，不赘述了。

5.2 旋转字符串

No. 796 难度 Easy

给定两个字符串，`s` 和 `goal`。如果在若干次旋转操作之后，`s` 能变成 `goal`，那么返回 `true`。

`s` 的 旋转操作就是将 `s` 最左边的字符移动到最右边。

例如，若 s = 'abcde'，在旋转一次之后结果就是'bcdea'。

示例 1
输入: s = "abcde", goal = "cdeab"
输出: true

示例 2
输入: s = "abcde", goal = "abced"
输出: false

解法一：搜索子字符串

思路：首先，如果 s 和 $goal$ 的长度不一样，那么无论怎么旋转，s 都不能得到 $goal$，返回 $false$。字符串 $s+s$ 包含了所有 s 可以通过旋转操作得到的字符串，只需要检查 $goal$ 是否为 $s+s$ 的子字符串即可。本题解中采用直接调用库函数的方法。

代码 5.2: 旋转字符串

```
1       fn rotate_string(s: &str, goal: &str) -> bool {
2           s.len() == goal.len() && (s.to_owned() + s).contains(goal)
3       }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(2n)$。

5.3 找到字符串中所有的异位词

No 438 难度 Medium

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词**的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是"cba", 它是"abc" 的异位词。
起始索引等于 6 的子串是"bac", 它是"abc" 的异位词。

示例 2
输入: s = "abab", p = "ab"
输出: [0,1,2]
解释:
起始索引等于 0 的子串是"ab", 它是"ab" 的异位词。
起始索引等于 1 的子串是"ba", 它是"ab" 的异位词。
起始索引等于 2 的子串是"ab", 它是"ab" 的异位词。

解法一：滑动窗口

思路: 根据题目要求，我们需要在字符串 s 寻找字符串 p 的异位词。因为字符串 p 的异位词的长度一定与字符串 p 的长度相同，所以我们可以在字符串 s 中构造一个长度为与字符串 p 的长度相同的滑动窗口，并在滑动中维护窗口中每种字母的数量；当窗口中每种字母的数量与字符串 p 中每种字母的数量相同时，则说明当前窗口为字符串 p 的异位词。

在算法的实现中，我们可以使用数组来存储字符串 p 和滑动窗口中每种字母的数量。

代码 5.3: 找到异位词

```rust
    fn find_anagrams(s: &str, p: &str) -> Vec<usize> {
        let s_len = s.len();
        let p_len = p.len();
        if s_len < p_len {
            return vec![];
        }

```

```rust
        let mut s_count = [0; 26];
        let mut p_count = [0; 26];
        for (i, ch) in p.chars().enumerate() {
            s_count[s.as_bytes()[i] as usize - b'a' as usize] += 1;
            p_count[ch as usize - b'a' as usize] += 1;
        }

        let mut ans = vec![];
        if s_count == p_count {
            ans.push(0);
        }

        for (i, ch) in s.chars().enumerate().take(s_len - p_len) {
            s_count[s.as_bytes()[i] as usize - b'a' as usize] -= 1;
            s_count[s.as_bytes()[i + p_len] as usize - b'a' as usize] += 1;
            if s_count == p_count {
                ans.push(i + 1);
            }
        }

        ans
    }
```

复杂度分析

- 时间复杂度：$O(m + (n - m) \times \Sigma)$，其中 m 为字符串 p 的长度，n 为字符串 s 的长度，Σ 为字符集大小。
- 空间复杂度：$O(\Sigma)O$。用于存储字符串 ppp 和滑动窗口中每种字母的数量。

6

队列

队列是一种先进先出 (FIFO) 的线性表数据结构。队列只允许在表的一端进行插入，而在另一端进行删除。队列的操作有两种：入队和出队。入队是在队列的末尾插入一个元素；出队是删除队列的第一个元素。

6.1 滑动窗口最大值

No. 239 难度 Hard

给你一个整数数组 `nums`，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

滑动窗口每滑动一次，就将最大值放入到结果数组中。

示例

输入: nums = [1,3,-1,-3,5,3,6,7], k = 3
输出: [3,3,5,5,6,7]
解释:
滑动窗口的位置　　最大值

[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

解法一：优先队列

思路: 对于「最大值」，我们可以想到一种非常合适的数据结构，那就是优先队列（堆），其中的大根堆可以帮助我们实时维护一系列元素中的最大值。

对于本题而言，初始时，我们将数组 $nums$ 的前 k 个元素放入优先队列中。每当我们向右移动窗口时，我们就可以把一个新的元素放入优先队列中，此时堆顶的元素就是堆中所有元素的最大值。然而这个最大值可能并不在滑动窗口中，在这种情况下，这个值在数组 $nums$ 中的位置出现在滑动窗口左边界的左侧。因此，当我们后续继续向右移动窗口时，这个值就永远不可能出现在滑动窗口中了，我们可以将其永久地从优先队列中移除。

我们不断地移除堆顶的元素，直到其确实出现在滑动窗口中。此时，堆顶元素就是滑动窗口中的最大值。为了方便判断堆顶元素与滑动窗口的位置关系，我们可以在优先队列中存储二元组 $(num, index)$，表示元素 num 在数组中的下标为 $index$。

代码 6.1: 滑动窗口最大值

```rust
use std::collections::BinaryHeap;

pub fn max_sliding_window(nums: Vec<i32>, k: i32) -> Vec<i32> {
    let n = nums.len();
    let k = k as usize;
    let mut heap = BinaryHeap::new();
    let mut ans = Vec::with_capacity(n - k + 1);

    for i in 0..k {
        heap.push((nums[i], i));
    }
    ans.push(heap.peek().unwrap().0);

    for i in k..n {
        heap.push((nums[i], i));
        while let Some((num, idx)) = heap.peek() {
            if *idx <= i - k {
                heap.pop();
            } else {
                ans.push(*num);
                break;
            }
        }
    }

    ans
}
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为序列长度。每个位置至多被遍历两次。
- 空间复杂度：$O(1)$。只需要常数的空间存放若干变量。

6.2 买票所需的时间

No. 2073 难度 Easy

有 n 个人前来排队买票，其中第 0 人站在队伍最前方，第 $(n-1)$ 人站在队伍最后方。

给你一个下标从 0 开始的整数数组 $tickets$ ，数组长度为 n ，其中第 i 人想要购买的票数为 $tickets[i]$ 。

每个人买票都需要用掉恰好 1 秒。一个人一次只能买一张票，如果需要购买更多票，他必须走到队尾重新排队（瞬间发生，不计时间）。如果一个人没有剩下需要买的票，那他将会离开队伍。

返回位于位置 k（下标从 0 开始）的人完成买票需要的时间（以秒为单位）。

示例

输入: tickets = [2,3,2], k = 2
输出: 6
- 第一轮，队伍中的每个人都买到一张票，队伍变为 [1, 2, 1] 。
- 第二轮，队伍中的每个都又都买到一张票，队伍变为 [0, 1, 0] 。
位置 2 的人成功买到 2 张票，用掉 3 + 3 = 6 秒。

解法一：计算满足条件的所有人所需时间之和

思路: 为了计算第 k 个人买完票所需的时间，我们可以首先计算在这个过程中每个人买票所需要的时间，再对这些时间求和得到答案。

我们可以对每个人的下标 i 分类讨论：

- 如果这个人初始在第 k 个人的前方，或者这个人恰好为第 k 个人，即 $i \leq k$，此时在第 k 个人买完票之前他最多可以购买 $tickets[k]$ 张。考虑到他想要购买的票数，那么他买票所需时间即为 $min\ (tickets[k], tickets[i])$；
- 如果这个人初始在第 k 个人的后方，即 $i > k$，此时在第 k 个人买完票之前他最多可以购买 $tickets[k]-1$ 张。考虑到他想要购买的票数，那么他买票所需时间即为 $min\ (tickets[k]-1, tickets[i])$。

我们遍历每个人的下标，按照上述方式计算并维护每个人买票所需时间之和 (在第 k 人买到票之前)，即可得到第 k 个人买完票所需的时间，我们返回该值作为答案。

代码 6.2: 买票所需的时间

```rust
fn time_required_to_buy(tickets: Vec<i32>, k: usize) -> i32 {
    let n = tickets.len();
    let mut res = 0;
    for i in 0..n {
        // Calculate the time required for each person
        if i <= k {
            res += std::cmp::min(tickets[i], tickets[k]);
        } else {
```

```
                res += std::cmp::min(tickets[i], tickets[k] - 1);
            }
        }
        res
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

这道题就是如果你能梳理清楚这个过程，那么代码就很容易写出来。

6.3 化栈为队列

No 03.04 难度 Easy

实现一个 MyQueue 类，该类用两个栈来实现一个队列。

示例：

```
MyQueue queue = new MyQueue();

queue.push(1);
queue.push(2);
queue.peek();  // 返回 1
queue.pop();   // 返回 1
queue.empty(); // 返回 false
```

说明：

- 你只能使用标准的栈操作 – 也就是只有 push to top, peek/pop from top, size 和 is empty 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
- 假设所有操作都是有效的（例如，一个空的队列不会调用 pop 或者 peek 操作）。

解法一：双栈

思路: 将一个栈当作输入栈，用于压入 push 传入的数据；另一个栈当作输出栈，用于 pop 和 peek 操作。

每次 $textttpop$ 或 peek 时，若输出栈为空则将输入栈的全部数据依次弹出并压入输出栈，这样输出栈从栈顶往栈底的顺序就是队列从队首往队尾的顺序。

代码 6.3: 化栈为队列

```rust
// 双栈实现的队列
struct MyQueue {
    in_stack: Vec<i32>,
    out_stack: Vec<i32>,
}

impl MyQueue {
    fn new() -> Self {
        MyQueue {
            in_stack: Vec::new(),
            out_stack: Vec::new(),
        }
    }

    // 入队，放入输入栈
    fn push(&mut self, x: i32) {
        self.in_stack.push(x);
    }

    // 将输入栈的元素转移到输出栈
    fn in2out(&mut self) {
        while let Some(x) = self.in_stack.pop() {
            self.out_stack.push(x);
        }
    }

    // 出队，从输出栈弹出
    fn pop(&mut self) -> i32 {
        // 如果输出栈为空，将输入栈的元素转移到输出栈
        if self.out_stack.is_empty() {
            self.in2out();
        }
        // 弹出输出栈的栈顶元素
        self.out_stack.pop().unwrap()
    }

    // 返回队首元素
    fn peek(&mut self) -> i32 {
        if self.out_stack.is_empty() {
            self.in2out();
        }
        *self.out_stack.last().unwrap()
    }

    // 判断队列是否为空
    fn empty(&self) -> bool {
        self.in_stack.is_empty() && self.out_stack.is_empty()
    }
}
```

复杂度分析

- 时间复杂度：$O(1)$。

- 空间复杂度：$O(n)$。

你可以思考下，如何使用队列实现栈？注意不能使用 $[]int$ 来实现队列，可以使用 channel 来模拟队列。

7

栈

栈是一种重要的数据结构, 它具有后进先出 (LIFO,Last-In-First-Out) 的特性。栈可以用来暂时存储数据, 当需要时再将数据取出来使用。

栈的主要操作有: - 入栈 (push): 将一个新元素压入栈顶。- 出栈 (pop): 将栈顶元素移除并返回该元素的值。- 取栈顶元素 (top/peek): 返回栈顶元素的值, 但不移除它。- 判断栈空 (isEmpty): 判断栈是否为空。

栈常用于以下场景: - 函数调用堆栈: 在计算机中, 每当发生函数调用时, 就会将参数、局部变量和返回地址等压入内存区的堆栈中, 待函数执行完成后, 再从堆栈中恢复数据, 并跳转到返回地址继续执行。- 括号匹配: 需要匹配包含括号的表达式时, 可以使用栈来跟踪左括号, 遇到右括号时就从栈顶取出一个左括号进行匹配。- 网页浏览历史记录: 通常浏览器会使用栈来存储网页访问历史, 当点击前进或后退按钮时, 可以很方便地重新加载所需网页。- 撤销功能: 在文本编辑器或绘图软件中, 撤销功能通常使用栈来保存每一次的操作记录, 撤销时将栈顶元素取出并执行相应的逆操作。- 深度优先搜索: 在图论算法中, 深度优先搜索会使用栈存储待访问的顶点, 以便按深度优先顺序访问。

7.1 下一个更大元素

No. 496 难度 Easy

$nums1$ 中数字 x 的下一个更大元素是指 x 在 $nums2$ 中对应位置右侧的第一个比 x 大的元素。

给你两个没有重复元素的数组 $nums1$ 和 $nums2$ ， 下标从 0 开始计数， 其中 $nums1$ 是 $nums2$ 的子集。

对于每个 $0 <= i < nums1.length$ ， 找出满足 $nums1[i] == nums2[j]$ 的下标 j ， 并且

在 $nums2$ 确定 $nums2[j]$ 的下一个更大元素。如果不存在下一个更大元素，那么本次查询的答案是 -1 。

返回一个长度为 $nums1.length$ 的数组 ans 作为答案，满足 $ans[i]$ 是如上所述的下一个更大元素。

示例

输入: nums1 = [4,1,2], nums2 = [1,3,4,2]
输出: [-1,3,-1]
解释:
nums1 中每个值的下一个更大元素如下所述:

- 4 ，用加粗斜体标识，nums2 = [1,3,**4**,2]。不存在下一个更大元素，所以答案是 -1 。
- 1 ，用加粗斜体标识，nums2 = [**1**,3,4,2]。下一个更大元素是 3 。
- 2 ，用加粗斜体标识，nums2 = [1,3,4,**2**]。不存在下一个更大元素，所以答案是 -1 。

解法一: 单调栈 + 哈希表

思路: 我们可以先预处理 $nums2$，使查询 $nums1$ 中的每个元素在 $nums2$ 中对应位置的右边的第一个更大的元素值时不需要再遍历 $nums2$。于是，我们将题目分解为两个子问题:

- 第 1 个子问题: 如何更高效地计算 $nums2$ 中每个元素右边的第一个更大的值;
- 第 2 个子问题: 如何存储第 1 个子问题的结果。

我们可以使用单调栈来解决第 1 个子问题。倒序遍历 $nums2$，并用单调栈中维护当前位置右边的更大的元素列表，从栈底到栈顶的元素是单调递减的。

具体地，每次我们移动到数组中一个新的位置 i，就将当前单调栈中所有小于 $nums2[i]$ 的元素弹出单调栈，当前位置右边的第一个更大的元素即为栈顶元素，如果栈为空则说明当前位置右边没有更大的元素。随后我们将位置 i 的元素入栈。

可以结合以下例子来理解。

代码 7.1: 下一个更大元素

```rust
fn next_greater_element(nums1: Vec<i32>, nums2: Vec<i32>) -> Vec<i32> {
    let mut mp = std::collections::HashMap::new();
    let mut stack = Vec::new();

    for &num in nums2.iter().rev() {
        while !stack.is_empty() && num >= *stack.last().unwrap() {
            stack.pop();
        }
        if let Some(&top) = stack.last() {
            mp.insert(num, top);
```

```
11              } else {
12                  mp.insert(num, -1);
13              }
14              stack.push(num);
15          }
16
17          let mut res = Vec::new();
18          for num in nums1 {
19              res.push(*mp.get(&num).unwrap());
20          }
21          res
22      }
```

复杂度分析

- 时间复杂度：$O(m+n)$，其中 n 为序列长度。每个位置至多被遍历两次。
- 空间复杂度：$O(n)$。

7.2 有效的括号

No. 20 难度 Easy

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：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。
- 每个右括号都有一个对应的相同类型的左括号。

示例 1

输入: s = "()"
输出: true

示例 2

输入: s = s = "(]"
输出: false

示例 3
输入: s = "()"
输出: true

解法一：迭代

思路: 判断括号的有效性可以使用「栈」这一数据结构来解决。

我们遍历给定的字符串 s。当我们遇到一个左括号时，我们会期望在后续的遍历中，有一个相同类型的右括号将其闭合。由于后遇到的左括号要先闭合，因此我们可以将这个左括号放入栈顶。

当我们遇到一个右括号时，我们需要将一个相同类型的左括号闭合。此时，我们可以取出栈顶的左括号并判断它们是否是相同类型的括号。如果不是相同的类型，或者栈中并没有左括号，那么字符串 s 无效，返回 $false$。为了快速判断括号的类型，我们可以使用哈希表存储每一种括号。哈希表的键为右括号，值为相同类型的左括号。

在遍历结束后，如果栈中没有左括号，说明我们将字符串 sss 中的所有左括号闭合，返回 $true$，否则返回 $false$。

注意到有效字符串的长度一定为偶数，因此如果字符串的长度为奇数，我们可以直接返回 $false$，省去后续的遍历判断过程。

代码 7.2: 有效的括号

```rust
fn is_valid(s: String) -> bool {
    let n = s.len();
    if n % 2 == 1 {
        return false;
    }
    let pairs: std::collections::HashMap<char, char> = [
        (')', '('),
        (']', '['),
        ('}', '{'),
    ].iter().cloned().collect();
    let mut stack: Vec<char> = Vec::new();

    for c in s.chars() {
        if let Some(&top) = stack.last() {
            if let Some(&pair) = pairs.get(&c) {
                if top == pair {
                    stack.pop();
                    continue;
                }
            }
        }
        stack.push(c);
    }
    stack.is_empty()
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n + |\Sigma|)$。

7.3 买卖股票的最佳时机

No. 121 难度 Easy

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。

解法一: 单调栈

思路: 我们可以使用单调栈来解决本题。我们维护一个单调递减的栈，栈中存放的是股票的价格。当我们遍历到一个股票价格时，我们将其与栈顶的股票价格比较，如果当前股票价格大于栈顶的股票价格，我们就找到了一次交易机会，我们计算其利润。当前股票价格小于等于栈顶的股票价格。我们将当前股票价格入栈。

代码 7.3: 单调栈解法

```rust
fn max_profit(prices: Vec<i32>) -> i32 {
    let mut res = 0;
    let mut stack: Vec<i32> = Vec::new();

    // Traverse the stock prices
    for price in prices {
        // Monotonic decreasing stack, push the current stock price if it's lower
        if stack.is_empty() || *stack.last().unwrap() >= price {
            stack.push(price);
        } else { // Current stock price is higher, calculate the profit
            // If the current profit is higher, update the result
            if price - *stack.last().unwrap() > res {
                res = price - *stack.last().unwrap();
            }
        }
    }
```

```
17
18          res
19      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

不过从上面分析，我们还可以将空间复杂度降到 $O(1)$，因为我们不需要栈，只需要记住当前股票最小的价格就可以：

代码 7.4: 买卖股票的最佳时机

```rust
fn max_profit(prices: Vec<i32>) -> i32 {
    let mut min_price = i32::MAX;
    let mut max_profit = 0;

    for price in prices {
        if price < min_price {
            min_price = price;
        } else if price - min_price > max_profit {
            max_profit = price - min_price;
        }
    }

    max_profit
}
```

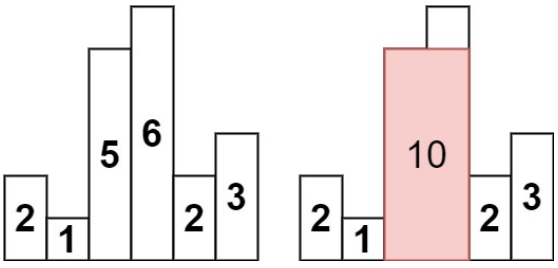
7.4 柱状图中最大的矩形

No. 84 难度 Hard

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 。

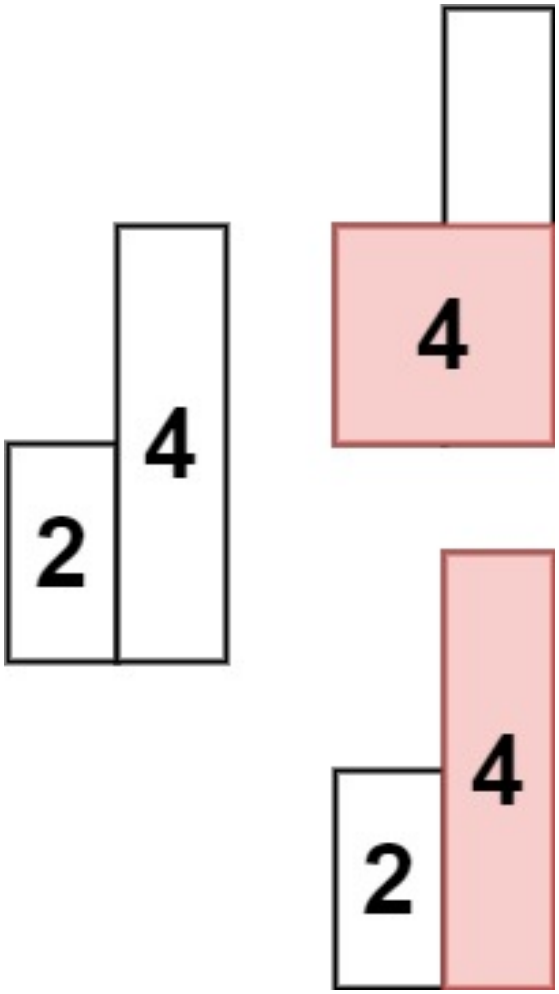
求在该柱状图中，能够勾勒出来的矩形的最大面积。

示例 1



输入: heights = [2,1,5,6,2,3]
输出: 10
解释: 最大的矩形为图中红色区域，面积为 10。

示例 2



输入: heights = [2,4]
输出: 4

解法一: 单调栈

思路: 暴力解法会超时，我们使用单调递增栈，记录每个柱子的左右边界。当当前访问的柱子比栈顶元素的柱子小的时候，就说明我们找到了右边界（就是当前柱子下标），就可以求解栈顶柱子的矩形面积了，左边界呢？因为是递增的，使用左边界就是栈顶元素的后一个元素（也就是比他先一步入栈的那个元素），高就是栈顶柱子的高度，

代码 7.5: 单调栈

```
1       fn largest_rectangle_area(heights: Vec<i32>) -> i32 {
```

```rust
        let mut stack: Vec<usize> = Vec::new();
        stack.push(0); // stack 的哨兵，方便确定左边界
        let mut heights = heights;
        heights.push(0); // 添加一个哨兵，减少代码量

        let n = heights.len();
        let mut res = 0; // 结果

        for i in 0..n {
            // 因为我们无法访问 heights[-1]，所以限制 len(stack) > 1
            while stack.len() > 1 && heights[*stack.last().unwrap()] > heights[i] {
                // 栈顶元素
                let top = stack.pop().unwrap();
                // 左边界（栈顶元素的后一个元素）
                let l = *stack.last().unwrap();

                // 矩形面积：(右边界-左边界-1) * 高度
                // 右边界就是 i
                // 高度就是以栈顶元素为下标的柱子的高度
                // 左边界就是栈顶元素的下一个元素（因为我们添加了哨兵 0，所以这公式依旧成立）
                res = res.max((i - l - 1) as i32 * heights[top]);
            }
            stack.push(i);
        }

        res
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

8

堆

堆 (最小堆、最大堆、优先级队列)。

8.1 数组中的第 K 个最大元素

No. 215 难度 Medium

给定整数数组 $nums$ 和整数 k，请返回数组中第 k 个最大的元素。

请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

你必须设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

示例

输入: [3,2,1,5,6,4], k = 2
输出: 5

提示:

- $1 <= k <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

解法一: 最大堆

思路: 可以先构造一个空间大小为 k 的最小堆，再从 $nums[k+1]$ 开始与堆顶元素比较，如果 $nums[i] > heap$, 则将当前堆顶元素出堆, 并将 $nums[i]$ 入堆, 这样遍历完数组后堆顶元素就是 ans。

代码 8.1: 数组中的第 K 个最大元素

```rust
use std::collections::BinaryHeap;

fn find_kth_largest(nums: Vec<i32>, k: usize) -> i32 {
    let mut min_heap = BinaryHeap::new();
    for i in 0..k {
        min_heap.push(nums[i]);
    }
    for i in k..nums.len() {
        if nums[i] > *min_heap.peek().unwrap() {
            min_heap.pop();
            min_heap.push(nums[i]);
        }
    }
    *min_heap.peek().unwrap()
}
```

复杂度分析

- 时间复杂度：$O(n \log k)$，n 轮入堆和出堆，堆的最大长度 k。
- 空间复杂度：$O(k)$。

实际上这个解答不满足题目要求，题目要求时间复杂度是 $O(n)$ 的，这个解答的时间复杂度是 $O(n \log k)$ 的。

解法二：快速排序

思路：正常的快速排序时间复杂度是 $O(n \log n)$，但是因为我们的题目并没有要求对数组进行完全排序，所以我们可以使用快速排序的思想，对数组进行划分，如果划分的位置是 k，那么我们就找到了答案。如果划分的位置大于 k，那么我们就在左边继续划分，如果划分的位置小于 k，那么我们就在右边继续划分。

代码 8.2: 数组中的第 K 个最大元素

```rust
fn find_kth_largest(nums: &mut [i32], k: usize) -> i32 {
    let n = nums.len();
    quickselect(nums, 0, n - 1, n - k)
}

fn quickselect(nums: &mut [i32], l: usize, r: usize, k: usize) -> i32 {
    if l == r {
        return nums[k];
    }
    let partition = nums[l];
    let mut i = l as i32 - 1;
    let mut j = r as i32 + 1;
    while i < j {
        i += 1;
        while nums[i as usize] < partition {
            i += 1;
        }
```

```
18              j -= 1;
19              while nums[j as usize] > partition {
20                  j -= 1;
21              }
22              if i < j {
23                  nums.swap(i as usize, j as usize);
24              }
25          }
26          if k <= j as usize {
27              quickselect(nums, l, j as usize, k)
28          } else {
29              quickselect(nums, j as usize + 1, r, k)
30          }
31      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(\log n)$。

解法三：桶排序

思路: 因为题目中给出了数组元素的范围是 $-10^4 <= nums[i] <= 10^4$, 所以我们可以使用桶排序的思想, 先统计每个元素出现的次数, 然后从大到小遍历桶, 直到找到第 k 个元素。

代码 8.3: 数组中的第 K 个最大元素

```
1       fn find_kth_largest(nums: Vec<i32>, k: i32) -> i32 {
2           // 桶
3           let mut buckets = vec![0; 20001];
4           for num in nums {
5               buckets[(num + 10000) as usize] += 1; // 10000 是为了将负数转为正数, 并
          计算每个数字出现的次数
6           }
7
8           // 从大到小遍历桶
9           let mut k = k;
10          for i in (0..=20000).rev() {
11              // 剔除 buckets[i] 个元素
12              k -= buckets[i];
13              // 如果 k 小于等于零, 说明数字在刚在剔除的桶中
14              if k <= 0 {
15                  return i as i32 - 10000; // 减去 10000 是为了将数字恢复
16              }
17          }
18          0
19      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

清晰而标准的答案。在充分考虑到限定条件后的聪明的解答。

类似题目 No 347 No 692 No 1985

8.2 数据流的中位数

No 295 难度 Hard

中位数是有序整数列表中的中间值。如果列表的大小是偶数，则没有中间值，中位数是两个中间值的平均值。

例如 $arr = [2,3,4]$ 的中位数是 3 。例如 $arr = [2,3]$ 的中位数是 $(2 + 3)/2 = 2.5$ 。

示例

输入:

```
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
```

输出: [null, null, null, 1.5, null, 2.0]

解法一：优先队列

思路：我们用两个优先队列 $queMax$ 和 $queMin$ 分别记录大于中位数的数和小于等于中位数的数。当累计添加的数的数量为奇数时，$queMin$ 中的数的数量比 $queMax$ 多一个，此时中位数为 $queMin$ 的队头。当累计添加的数的数量为偶数时，两个优先队列中的数的数量相同，此时中位数为它们的队头的平均值。

为了方便讨论，我们把 $queMin$ 称之为左边的优先队列，$queMax$ 称之为右边的优先队列。

当我们尝试添加一个数 num 到数据结构中，我们需要分情况讨论：

1. $num\ maxqueMin$ 此时 num 小于等于中位数，我们需要将该数添加到 $queMin$ 中。新的中位数将小于等于原来的中位数，因此我们可能需要将 $queMin$ 中最大的数移动到 $queMax$ 中。
2. $num > maxqueMin$ 此时 num 大于中位数，我们需要将该数添加到 $queMax$ 中。新的中位数将大于等于原来的中位数，因此我们可能需要将 $queMax$ 中最小的数移动到 $queMin$ 中。

特别地，当累计添加的数的数量为 0 时，我们将 num 添加到 $queMin$ 中。

代码 8.4: 数据流的中位数

```
1    // hp is a min heap, to simulate a max heap in queMin, we store the negative
       values in it
```

```rust
struct MedianFinder {
    que_min: BinaryHeap<i32>,
    que_max: BinaryHeap<Reverse<i32>>,
}

impl MedianFinder {
    fn new() -> Self {
        MedianFinder {
            que_min: BinaryHeap::new(),
            que_max: BinaryHeap::new(),
        }
    }

    fn add_num(&mut self, num: i32) {
        let min_q = &mut self.que_min;
        let max_q = &mut self.que_max;

        // First time or smaller (or equal) than the maximum value in the left priority queue
        if min_q.is_empty() || num <= -min_q.peek().unwrap() {
            min_q.push(-num); // Store the negative value in the left priority queue
            if max_q.len() + 1 < min_q.len() {
                // If the number of elements in the left priority queue is 2 more than the right priority queue, adjust
                max_q.push(Reverse(-min_q.pop().unwrap()));
            }
        } else {
            max_q.push(Reverse(num)); // Store the value in the right priority queue
            if max_q.len() > min_q.len() {
                // If the number of elements in the right priority queue is 1 more than the left priority queue, adjust
                min_q.push(-max_q.pop().unwrap().0);
            }
        }
    }

    fn find_median(&self) -> f64 {
        let min_q = &self.que_min;
        let max_q = &self.que_max;

        if min_q.len() > max_q.len() {
            // Odd number of elements, return the maximum value in the left priority queue
            return -(*min_q.peek().unwrap() as f64);
        }

        (max_q.peek().unwrap().0 - *min_q.peek().unwrap() as i32) as f64 / 2.0
    }
}
```

复杂度分析

- 时间复杂度：
  - addNum: $O(\log n)$，其中 n 为累计添加的数的数量。
  - findMedian: $O(1)$。
- 空间复杂度：$O(n)$，主要为优先队列的开销。

9

树

树是一种抽象数据类型（ADT）或是实现这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由 $n(n>0)$ 个有限节点组成一个具有层次关系的集合。把它叫做「树」是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。

它具有以下的特点：

- 每个节点有零个或多个子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；
- 树里面没有环路。

9.1 二叉树的中序遍历

No. 94 难度 Easy

给定一个二叉树的根节点 $root$，返回它的 中序遍历。

示例

输入：root = [1,null,2,3]
输出：[1,3,2]

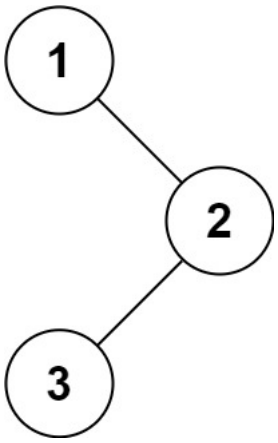


图 9.1. 二叉树

解法一：递归

思路: 首先我们需要了解什么是二叉树的中序遍历：
按照访问左子树——根节点——右子树的方式遍历这棵树，而在访问左子树或者右子树的时候我们按照同样的方式遍历，直到遍历完整棵树。因此整个遍历过程天然具有递归的性质，我们可以直接用递归函数来模拟这一过程。

定义 $inorder(root)$ 表示当前遍历到 $root$ 节点的答案，那么按照定义，我们只要递归调用 $inorder(root.left)$ 来遍历 $root$ 节点的左子树，然后将 $root$ 节点的值加入答案，再递归调用 $inorder(root.right)$ 来遍历 $root$ 节点的右子树即可，递归终止的条件为碰到空节点。

代码 9.1: 二叉树中序遍历

```rust
fn inorder_traversal(root: Option<Rc<RefCell<TreeNode>>>) -> Vec<i32> {
    let mut res = Vec::new();

    // Define the inorder traversal function
    fn inorder(node: Option<Rc<RefCell<TreeNode>>>, res: &mut Vec<i32>) {
        if let Some(node) = node {
            // Traverse left subtree
            inorder(node.borrow().left.clone(), res);
            // Visit current node
            res.push(node.borrow().val);
            // Traverse right subtree
            inorder(node.borrow().right.clone(), res);
        }
    }

    // Call the inorder traversal function
    inorder(root, &mut res);

    res
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

9.2 不同的二叉搜索树

No. 96 难度 Medium

给你一个整数 n ，求恰由 n 个节点组成且节点值从 1 到 n 互不相同的二叉搜索树有多少种？返回满足题意的二叉搜索树的种数。

示例

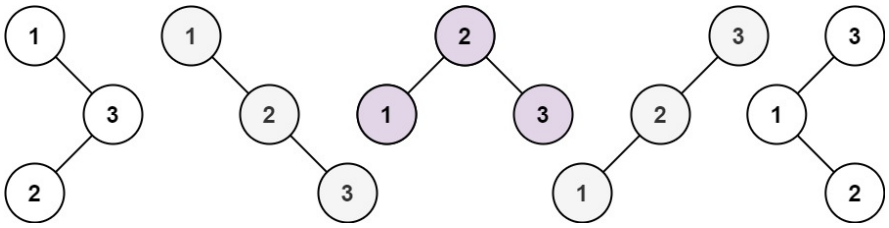


图 9.2. 不同的二叉搜索树

输入: n = 3
输出: 5

解法一：卡塔兰数

这道题使用动态规划可以求出答案，比较复杂。

这里我们介绍一种更简单的方法，使用卡塔兰数。这是一种" 作弊" 的方法，直接算出答案。

卡塔兰数在组合数学中有着广泛的应用，常见的组合意义包括：

- 括号表达式的正确配对数
- 二叉树的不同形态数
- 无重复访问迷宫路径数
- 平面树的形态数

思路: 卡塔兰数更便于计算的定义如下：

$$C_0 = 1, \qquad C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

我们直接求答案即可：

代码 9.2: 不同的二叉搜索树

```
1       fn num_trees(n: i32) -> i32 {
2           let mut c = 1;
3           for i in 0..n {
4               c = c * 2 * (2 * i + 1) / (i + 2);
5           }
6           c
7       }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

9.3 层序遍历

No. 102 难度 Medium

给你二叉树的根节点 $root$，返回其节点值的 层序遍历。(即逐层地，从左到右访问所有节点)。

示例

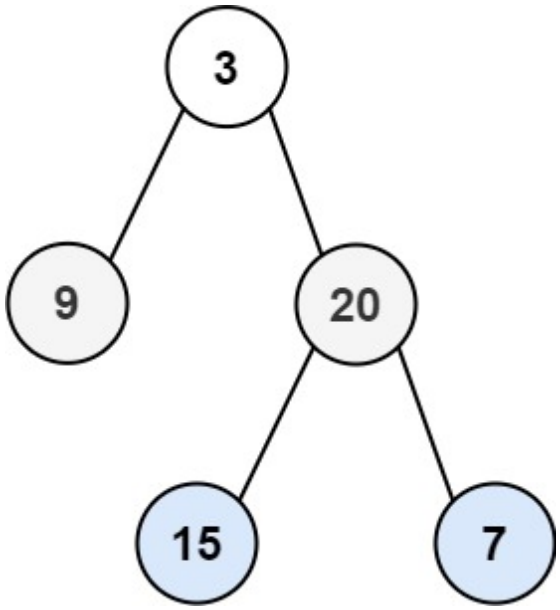


图 9.3. 层序遍历二叉搜索树

输入: root = [3,9,20,null,null,15,7]
输出: [[3],[9,20],[15,7]]

解法一：广度优先搜索

思路：我们可以用广度优先搜索解决这个问题。

我们可以想到最朴素的方法是用一个二元组 $(node, level)$ 来表示状态，它表示某个节点和它所在的层数，每个新进队列的节点的 $level$ 值都是父亲节点的 $level$ 值加一。最后根据每个点的 $level$ 对点进行分类，分类的时候我们可以利用哈希表，维护一个以 $level$ 为键，对应节点值组成的数组为值，广度优先搜索结束以后按键 $level$ 从小到大取出所有值，组成答案返回即可。

考虑如何优化空间开销：如何不用哈希映射，并且只用一个变量 $node$ 表示状态，实现这个功能呢？

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 s
  - 依次从队列中取 s 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 s_i 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 s_i 个元素。

代码 9.3: 层序遍历

```rust
use std::collections::VecDeque;

fn level_order(root: Option<Rc<RefCell<TreeNode>>>) -> Vec<Vec<i32>> {
    let mut ret: Vec<Vec<i32>> = Vec::new();
    if let Some(root) = root {
        let mut queue: VecDeque<Rc<RefCell<TreeNode>>> = VecDeque::new();
        queue.push_back(root);

        while !queue.is_empty() {
            let mut level: Vec<i32> = Vec::new();
            let mut next_level: VecDeque<Rc<RefCell<TreeNode>>> = VecDeque::
    new();

            while let Some(node) = queue.pop_front() {
                let node = node.borrow();
                level.push(node.val);

                if let Some(left) = &node.left {
                    next_level.push_back(Rc::clone(left));
                }
                if let Some(right) = &node.right {
                    next_level.push_back(Rc::clone(right));
                }
            }

            ret.push(level);
```

```
26                  queue = next_level;
27              }
28          }
29          ret
30      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

9.4 将数组转换成二叉搜索树

No. 108 难度 Easy

给你一个整数数组 $nums$，其中元素已经按 升序排列，请你将其转换为一棵 高度平衡二叉搜索树。

高度平衡二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

示例

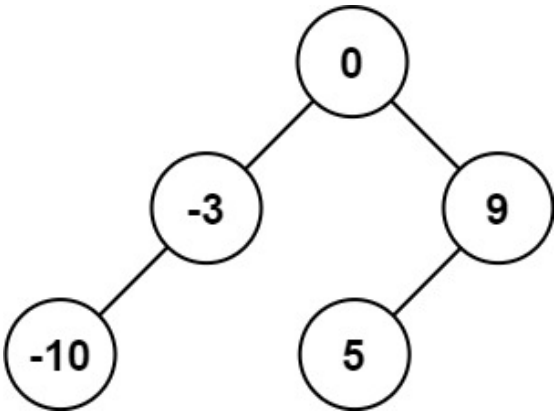


图 9.4. 二叉搜索树

输入: nums = [-10,-3,0,5,9]
输出: [0,-3,9,-10,null,5] **解释**: [0,-10,5,null,-3,null,9] 也将被视为正确答案：

解法一：中序遍历

思路: 选取数组中间的节点做平衡二叉搜索树的根节点之后，其余的数字分别位于平衡二叉搜索树的左子树和右子树中，左子树和右子树分别也是平衡二叉搜索树，因此可以通过递归的方式创建平衡二叉搜索树。

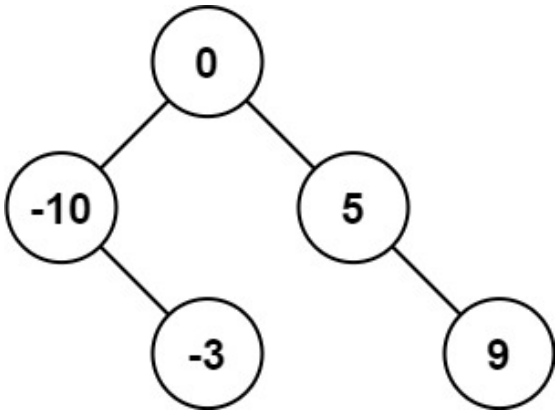


图 9.5. 二叉搜索树

选择中间位置左边的数字作为根节点，则根节点的下标为 $mid = (left + right)/2$，此处的除法为整数除法。

代码 9.4: 将数组转换成二叉搜索树

```rust
fn sorted_array_to_bst(nums: Vec<i32>) -> Option<Rc<RefCell<TreeNode>>> {
    helper(&nums, 0, nums.len() - 1)
}

fn helper(nums: &Vec<i32>, left: usize, right: usize) -> Option<Rc<RefCell<
   TreeNode>>> {
    if left > right {
        return None;
    }
    // Middle node, in-order
    let mid = (left + right) / 2;
    let root = Rc::new(RefCell::new(TreeNode::new(nums[mid])));

    // Recursively construct left and right subtrees
    root.borrow_mut().left = helper(nums, left, mid - 1);
    root.borrow_mut().right = helper(nums, mid + 1, right);

    Some(root)
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(\log n)$。

9.5 二叉树的最小深度

No. 111 简单 Easy

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

示例

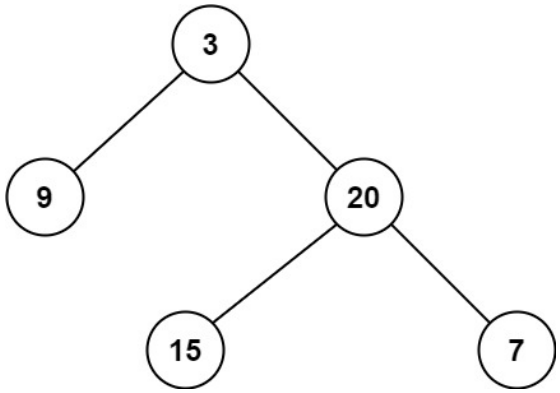


图 9.6. 二叉树

输入: root = [3,9,20,null,null,15,7]
输出: 2

解法一：深度优先搜索

思路: 首先可以想到使用深度优先搜索的方法，遍历整棵树，记录最小深度。

对于每一个非叶子节点，我们只需要分别计算其左右子树的最小叶子节点深度。这样就将一个大问题转化为了小问题，可以递归地解决该问题。

代码 9.5: 二叉树的最小深度

```
1     fn min_depth(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
2         match root {
3             None => 0,
4             Some(node) => {
5                 let node = node.borrow();
6                 if node.left.is_none() && node.right.is_none() {
7                     return 1;
8                 }
9                 let mut min_d = std::i32::MAX;
10                if let Some(left) = &node.left {
11                    min_d = min(min_depth(Some(Rc::clone(left))), min_d);
12                }
13                if let Some(right) = &node.right {
14                    min_d = min(min_depth(Some(Rc::clone(right))), min_d);
15                }
16                min_d + 1
17            }
18        }
```

```
19      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(h)$。其中 h 是树的高度。

9.6 二叉树的最近公共祖先

No. 236 难度 Medium

给定一个**二叉搜索树**, 找到该树中两个指定节点的最近公共祖先。

最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]

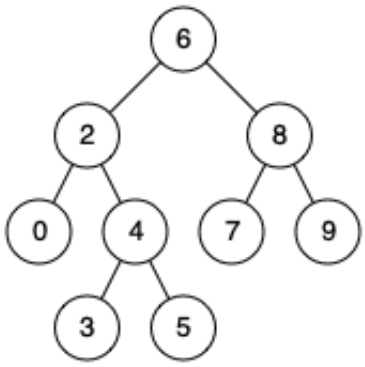


图 9.7. 二叉搜索树

示例

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6 **解释**: 节点 2 和节点 8 的最近公共祖先是 6。

解法一：深度优先搜索

思路: 我们对从根节点开始，通过遍历找出到达节点 p 和 q 的路径，找到分叉点，一共需要两次遍历。我们也可以考虑将这两个节点放在一起遍历。

代码 9.6: 二叉树的最近公共祖先

```
1       fn lowest_common_ancestor(root: Option<Rc<RefCell<TreeNode>>>, p: i32, q: i32
          ) -> Option<Rc<RefCell<TreeNode>>> {
2           let mut ancestor = root;
3           while let Some(node) = ancestor {
```

```
4               let node = node.borrow();
5               if p < node.val && q < node.val {
6                   ancestor = node.left.clone();
7               } else if p > node.val && q > node.val {
8                   ancestor = node.right.clone();
9               } else {
10                  return Some(Rc::clone(&node));
11              }
12          }
13          None
14      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(h)$。其中 h 是树的高度。

10

图

图是我们现实生活中连接关系的抽象，例如朋友圈、微博的关注关系。

图论中的很多问题都可以使用「深度优先搜索」或者「广度优先搜索」完成。图论中有很多专门的问题，都使用计算机科学家的名字命名，例如 Dijkstra 算法、Bellman-Ford 算法、Floyd 算法、Prim 算法、Kruskal 算法，学习这些算法需要我们深刻理解算法应用的场景，并且通过练习逐步掌握它们。

关于图的算法图一般都是中等难度或者困难难度的题目，需要我们有一定的基础才能解决。

10.1 找出星型图的中心节点

No. 1791 难度 Easy

我们先来一个 Easy 难度的题目开开胃。

有一个无向的 星型 图，由 n 个编号从 1 到 n 的节点组成。星型图有一个中心节点，并且恰有 $n-1$ 条边将中心节点与其他每个节点连接起来。

给你一个二维整数数组 $edges$ ，其中 $edges[i]=[u_i,v_i]$ 表示在节点 u_i 和 v_i 之间存在一条边。请你找出并返回 $edges$ 所表示星型图的中心节点。

示例

输入: edges = [[1,2],[2,3],[4,2]]
输出: 2 **解释**: 如上图所示， 节点 2 与其他每个节点都相连，所以节点 2 是中心节点。

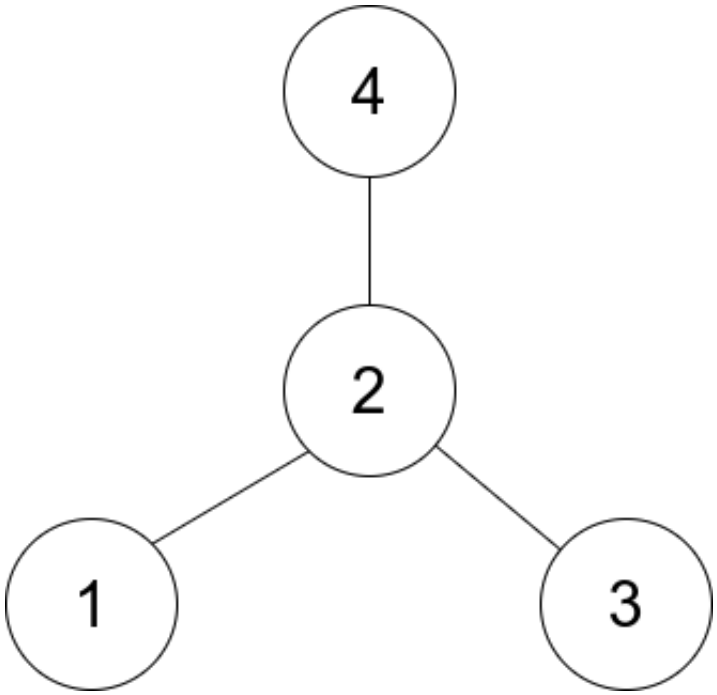


图 10.1. 星型图

解法一：计算度

思路: 由 n 个节点组成的星型图中，有一个中心节点，有 $n-1$ 条边分别连接中心节点和其余的每个节点。因此，中心节点的度是 $n-1$，其余每个节点的度都是 1。一个节点的度的含义是与该节点相连的边数。

遍历 $edges$ 中的每条边并计算每个节点的度，度为 $n-1$ 的节点即为中心节点。

代码 10.1: 找出星型图的中心节点

```rust
fn find_center(edges: Vec<Vec<i32>>) -> i32 {
    // Number of nodes is the number of edges + 1
    let n = edges.len() + 1;
    // Degrees, since node numbering starts from 1, the length is n+1
    let mut degrees = vec![0; n+1];

    // Traverse the edges
    for e in edges {
        degrees[e[0] as usize] += 1;
        degrees[e[1] as usize] += 1;
    }

    // Find the node with degree n-1
    for (i, d) in degrees.iter().enumerate() {
        if *d == n-1 {
            return i as i32;
        }
    }
```

```
19          -1
20      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

当然，这道题有一个更简便的算法，因为中心节点必然存在于每一条边上，那么只要用两条边，找到它们的交集即可。这个算法的时间复杂度是 $O(1)$。

代码 10.2: 找出星型图的中心节点

```rust
    fn find_center(edges: Vec<Vec<i32>>) -> i32 {
        if edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1] {
            edges[0][0]
        } else {
            edges[0][1]
        }
    }
```

10.2 矩阵中的最长递增路径

No. 329 难度 Hard

给定一个 mxn 整数矩阵 $matrix$ ，找出其中 **最长递增路径**的长度。

对于每个单元格，你可以往上，下，左，右四个方向移动。你不能在对角线方向上移动或移动到边界外（即不允许环绕）。

示例

输入: matrix = [[9,9,4],[6,6,8],[2,1,1]]
输出: 4 **解释**: 最长递增路径为 [1, 2, 6, 9]。

解法一: 记忆化深度优先搜索

思路: 将矩阵看成一个有向图，每个单元格对应图中的一个节点，如果相邻的两个单元格的值不相等，则在相邻的两个单元格之间存在一条从较小值指向较大值的有向边。问题转化成在有向图中寻找最长路径。

深度优先搜索是非常直观的方法。从一个单元格开始进行深度优先搜索，即可找到从该单元格开始的最长递增路径。对每个单元格分别进行深度优先搜索之后，即可得到矩阵中的最长递增路径的长度。由于同一个单元格对应的最长递增路径的长度是固定不变的，因此可以使用记忆化的方法进行优化。用矩阵 $memo$ 作为缓存矩阵，已经计算过的单元格的结果存储到缓存矩阵中。

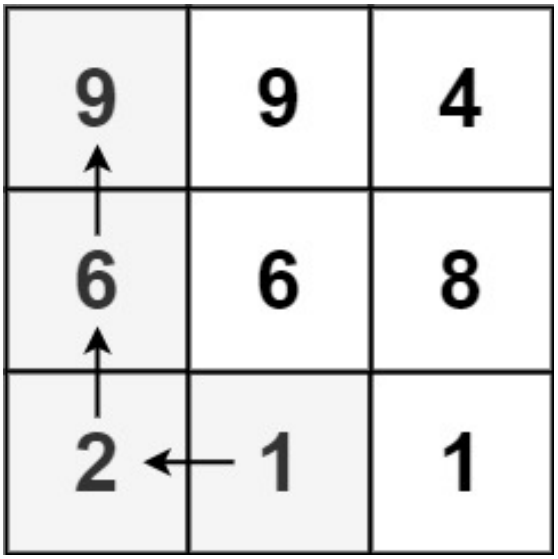


图 10.2. 矩阵

使用记忆化深度优先搜索，当访问到一个单元格 (i, j) 时，如果 $memo[i][j] \neq 0$，说明该单元格的结果已经计算过，则直接从缓存中读取结果，如果 $memo[i][j] = 0$，说明该单元格的结果尚未被计算过，则进行搜索，并将计算得到的结果存入缓存中。

代码 10.3: 矩阵中的最长递增路径

```rust
    // 四至
    const DIRS: [[i32; 2]; 4] = [[-1, 0], [1, 0], [0, -1], [0, 1]];
    static mut ROWS: usize = 0;
    static mut COLUMNS: usize = 0;

    // 记忆化深度优先搜索
    fn longest_increasing_path(matrix: Vec<Vec<i32>>) -> i32 {
        if matrix.is_empty() || matrix[0].is_empty() {
            return 0;
        }

        // 初始化
        unsafe {
            ROWS = matrix.len();
            COLUMNS = matrix[0].len();
        }
        let mut memo = vec![vec![0; COLUMNS]; ROWS];

        let mut ans = 0;
        // 遍历每个单元格
        for i in 0..ROWS {
            for j in 0..COLUMNS {
                ans = ans.max(dfs(&matrix, i, j, &mut memo));
            }
        }
        ans
```

```
27      }
28
29      // 深度优先搜索
30      fn dfs(matrix: &Vec<Vec<i32>>, row: usize, column: usize, memo: &mut Vec<Vec<
          i32>>) -> i32 {
31          unsafe {
32              if memo[row][column] != 0 {
33                  return memo[row][column];
34              }
35              memo[row][column] += 1;
36          }
37
38          // 遍历四个方向
39          for dir in &DIRS {
40              let new_row = (row as i32 + dir[0]) as usize;
41              let new_column = (column as i32 + dir[1]) as usize;
42              unsafe {
43                  if new_row < ROWS && new_column < COLUMNS && matrix[new_row][
          new_column] > matrix[row][column] {
44                      memo[row][column] = memo[row][column].max(dfs(matrix, new_row
          , new_column, memo) + 1);
45                  }
46              }
47          }
48          unsafe {
49              memo[row][column]
50          }
51      }
```

复杂度分析

- 时间复杂度：$O(mn)$。
- 空间复杂度：$O(mn)$。

10.3 获取你好友已观看的视频

No. 1311 难度 Medium

有 n 个人，每个人都有一个 0 到 $n-1$ 的唯一 id 。

给你数组 $watchedVideos$ 和 $friends$ ，其中 $watchedVideos[i]$ 和 $friends[i]$ 分别表示 $id=i$ 的人观看过的视频列表和他的好友列表。

$Level1$ 的视频包含所有你好友观看过的视频，$level2$ 的视频包含所有你好友的好友观看过的视频，以此类推。一般的，$Level$ 为 k 的视频包含所有从你出发，最短距离为 k 的好友观看过的视频。

给定你的 id 和一个 $level$ 值，请你找出所有指定 $level$ 的视频，并将它们按观看频率升序返回。如果有频率相同的视频，请将它们按字母顺序从小到大排列。

示例

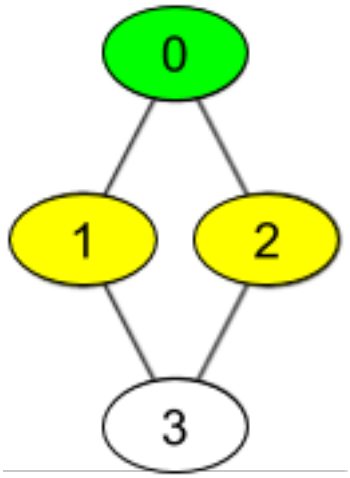


图 10.3. 好友

输入: watchedVideos = [["A","B"],["C"],["B","C"],["D"]], friends = [[1,2],[0,3],[0,3],[1,2]], id = 0, level = 1
输出: ["B","C"]
解释:
你的 id 为 0（绿色），你的朋友包括（黄色）:
id 为 1 -> watchedVideos = ["C"]
id 为 2 -> watchedVideos = ["B","C"]
你朋友观看过视频的频率为:
B -> 1
C -> 2

解法一: 广度优先搜索

思路: 本题将几个面试中常考的知识点（广度优先搜索、Set/Map 的应用、排序）进行了结合。题目本身的难度不大，但需要仔细考虑清楚每一个模块之间的关系，并且能尽量做到一次性通过，否则调试起来会较为复杂。

- 步骤一：找出所有 Level k 的好友
- 步骤二：统计好友观看过的视频
- 步骤三：将视频按照要求排序

代码 10.4: 获取你好友已观看的视频

```
use std::collections::HashMap;

fn watched_videos_by_friends(watched_videos: Vec<Vec<String>>, friends: Vec<
    Vec<i32>>, id: usize, level: usize) -> Vec<String> {
    // 步骤一：找出所有 Level k 的好友
    let n = friends.len();
```

```rust
        let mut queue = vec![id];
        let mut visited = vec![false; n];
        visited[id] = true;
        // 广度优先搜索，找到所以 level 的好友
        let mut level = level;
        while level > 0 {
            let size = queue.len();
            for i in 0..size {
                let f = queue[i];
                for &friend in &friends[f] {
                    if !visited[friend as usize] {
                        queue.push(friend as usize);
                        visited[friend as usize] = true;
                    }
                }
            }
            // 下一层级的好友
            queue = queue[size..].to_vec();
            // 下一层级
            level -= 1;
        }

        // 步骤二：统计好友观看过的视频
        let mut videos = HashMap::new();
        for &f in &queue {
            for video in &watched_videos[f] {
                *videos.entry(video.to_string()).or_insert(0) += 1;
            }
        }

        // 步骤三：将视频按照要求排序
        let mut res: Vec<String> = videos.keys().cloned().collect();
        res.sort_by(|a, b| {
            let freq_a = videos.get(a).unwrap();
            let freq_b = videos.get(b).unwrap();
            if freq_a == freq_b {
                a.cmp(b)
            } else {
                freq_a.cmp(freq_b)
            }
        });

        res
    }
```

复杂度分析

- 时间复杂度：$)O(N+M+V\log V)$，其中 N 是人数，M 是好友关系的总数，V 是电影的总数。
- 空间复杂度：$O()$。只需要常数的空间存放若干变量。

11

区间

区间表示为一个左右端点定义的数字范围，通常用 $[left, right]$ 表示。

区间相关问题包括：区间交集、区间合并、区间覆盖、区间切割等。

常见的区间处理算法：

- 排序 + 遍历：先对区间端点排序，然后线性遍历处理。
- 分治：递归分割问题为子问题求解。
- 线段树：将区间划分在树结构上，支持高效查询。
- 扫描线：利用“扫描线”逐步处理区间间关系。

11.1 合并区间

56 Medium

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

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]。

解法一：排序

思路：如果我们按照区间的左端点排序，那么在排完序的列表中，可以合并的区间一定是连续的。可合并的区间在排完序的列表中是连续的。

我们用数组 $merged$ 存储最终的答案。

首先，我们将列表中的区间按照左端点升序排序。然后我们将第一个区间加入 $merged$ 数组中，并按顺序依次考虑之后的每个区间：

如果当前区间的左端点在数组 $merged$ 中最后一个区间的右端点之后，那么它们不会重合，我们可以直接将这个区间加入数组 $merged$ 的末尾；

否则，它们重合，我们需要用当前区间的右端点更新数组 $merged$ 中最后一个区间的右端点，将其置为二者的较大值。

代码 11.1: 合并区间

```rust
fn merge(intervals: Vec<Vec<i32>>) -> Vec<Vec<i32>> {
    // 排序
    let mut intervals = intervals;
    intervals.sort_by(|a, b| a[0].cmp(&b[0]));

    // 结果
    let mut merged: Vec<Vec<i32>> = Vec::new();
    for interval in intervals {
        // 如果 merged 为空，或者当前区间的左端点大于 merged 中最后区间的右端点
        // 则不重合，直接加入 merged
        if merged.is_empty() || merged.last().unwrap()[1] < interval[0] {
            merged.push(interval);
        } else {
            // 否则，重合，更新 merged 中最后区间的右端点
            merged.last_mut().unwrap()[1] = merged.last().unwrap()[1].max(
    interval[1]);
        }
    }
    merged
}
```

复杂度分析

- 时间复杂度：$O(n \log n)$。
- 空间复杂度：$O(\log n)$。

11.2 区间列表的交集

No 986 难度 Medium

给定两个由一些闭区间组成的列表，$firstList$ 和 $secondList$ ，其中 $firstList[i] = [start_i, end_i]$ 而 $secondList[j] = [start_j, end_j]$ 。每个区间列表都是成对不相交的，并且已经排序。

返回这两个区间列表的交集。

形式上，闭区间 $[a, b]$（其中 $a <= b$）表示实数 x 的集合，而 $a <= x <= b$ 。

两个闭区间的交集是一组实数，要么为空集，要么为闭区间。例如，$[1, 3]$ 和 $[2, 4]$ 的交集为 $[2, 3]$ 。

示例

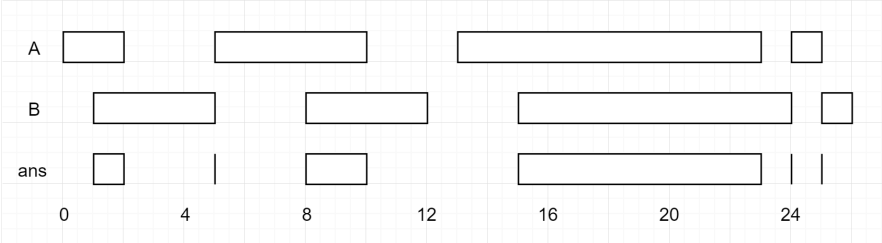


图 11.1. 两个区间的交集

输入: firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]
输出: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]

解法一: 归并区间

我们称 b 为区间 $[a, b]$ 的末端点。

在两个数组给定的所有区间中，假设拥有最小末端点的区间是 $A[0]$。(为了不失一般性，该区间出现在数组 A 中)

然后，在数组 B 的区间中，$A[0]$ 只可能与数组 B 中的至多一个区间相交。(如果 B 中存在两个区间均与 $A[0]$ 相交，那么它们将共同包含 $A[0]$ 的末端点，但是 B 中的区间应该是不相交的，所以存在矛盾)

思路: 如果 $A[0]$ 拥有最小的末端点，那么它只可能与 $B[0]$ 相交。然后我们就可以删除区间 $A[0]$，因为它不能与其他任何区间再相交了。

相似的，如果 $B[0]$ 拥有最小的末端点，那么它只可能与区间 $A[0]$ 相交，然后我们就可以将 $B[0]$ 删除，因为它无法再与其他区间相交了。我们用两个指针 i 与 j 来模拟完成删除 $A[0]$ 或 $B[0]$ 的操作。

代码 11.2: 归并区间

```
1     fn interval_intersection(a: Vec<Vec<i32>>, b: Vec<Vec<i32>>) -> Vec<Vec<i32>>
          {
2         let mut ans: Vec<Vec<i32>> = Vec::new();
3         let (mut i, mut j) = (0, 0);
4
5         // 归并区间
6         while i < a.len() && j < b.len() {
7             // 计算左节点的最大值
8             let lo = a[i][0].max(b[j][0]);
9             // 计算右节点的最小值
```

```
            let hi = a[i][1].min(b[j][1]);
            // 如果左节点小于右节点，说明有交集
            if lo <= hi {
                ans.push(vec![lo, hi]);
            }

            // 删除右节点小的区间
            if a[i][1] < b[j][1] {
                i += 1;
            } else {
                j += 1;
            }
        }

        ans
    }
```

复杂度分析

- 时间复杂度：$O(m+n)$。
- 空间复杂度：$O(m+n)$。

11.3 最小时间差

No. 539 难度 Medium

给定一个 24 小时制（小时: 分钟"**HH:MM**"）的时间列表，找出列表中任意两个时间的最小时间差并以分钟数表示。

示例 1
输入: timePoints = ["23:59","00:00"]
输出: 1

示例 2
输入: timePoints = ["00:00","23:59","00:00"]
输出:

解法一：排序

思路: 将 $timePoints$ 排序后，最小时间差必然出现在 $timePoints$ 的两个相邻时间，或者 $timePoints$ 的两个首尾时间中。因此排序后遍历一遍 $timePoints$ 即可得到最小时间差。

代码 11.3: 最小时间差

```
    // 获取分钟数
    fn get_minutes(t: &str) -> i32 {
        let hours = t[0..2].parse::<i32>().unwrap();
        let minutes = t[3..5].parse::<i32>().unwrap();
```

```rust
        hours * 60 + minutes
    }

    // 获取最小值
    fn find_min_difference(time_points: Vec<String>) -> i32 {
        // 鸽巢原理，如果超过 24*60 分钟，那么必然有重复，返回 0
        if time_points.len() > 1440 {
            return 0;
        }

        // 排序
        let mut sorted_time_points = time_points.clone();
        sorted_time_points.sort();

        let mut ans = std::i32::MAX;
        let t0_minutes = get_minutes(&sorted_time_points[0]);
        let mut pre_minutes = t0_minutes;

        for t in sorted_time_points.iter().skip(1) {
            let minutes = get_minutes(t);
            ans = ans.min(minutes - pre_minutes); // 相邻时间的时间差，和答案取最小值
            if ans == 0 {
                return 0;
            }
            pre_minutes = minutes;
        }

        ans = ans.min(t0_minutes + 1440 - pre_minutes); // 首尾时间的时间差
        ans
    }
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为序列长度。每个位置至多被遍历两次。
- 空间复杂度：$O(1)$。只需要常数的空间存放若干变量。

12

并查集

并查集（Union-find Data Structure）是一种树型的数据结构。它的特点是由子结点找到父亲结点，用于处理一些不交集（Disjoint Sets）的合并及查询问题。

- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

基本上都是中级难度和困难难度的题目。

12.1 最长连续序列

No. 128 Medium

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。

请你设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

示例

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]。它的长度是 4。

解法一：哈希表

时间复杂度 $O(n)$，就不要考虑排序然后再寻找连续序列了。

思路: 我们考虑枚举数组中的每个数 x，考虑以其为起点，不断尝试匹配 $x+1, x+2, \cdots$ 是否存在，假设最长匹配到了 $x+y$，那么以 x 为起点的最长连续序列即为 $x+1, x+2, \cdots, x+y$，其长度为 $y+1$，我们不断枚举并更新答案即可。

对于匹配的过程，暴力的方法是 $O(n)$ 遍历数组去看是否存在这个数，但其实更高效的方法是用一个哈希表存储数组中的数，这样查看一个数是否存在即能优化至 $O(1)$ 的时间复杂度。

仅仅是这样我们的算法时间复杂度最坏情况下还是会达到 $O(n^2$ 即外层需要枚举 $O(n)$ 个数，内层需要暴力匹配 $O(n)$ 次)，无法满足题目的要求。但仔细分析这个过程，我们会发现其中执行了很多不必要的枚举，如果已知有一个 $x, x+1, x+2, \cdots, x+y$ 的连续序列，而我们却重新从 $x+1$，$x+2$ 或者是 $x+y$ 处开始尝试匹配，那么得到的结果肯定不会优于枚举 x 为起点的答案，因此我们在外层循环的时候碰到这种情况跳过即可。

那么怎么判断是否跳过呢？由于我们要枚举的数 x 一定是在数组中不存在前驱数 $x-1$ 的，不然按照上面的分析我们会从 $x-1$ 开始尝试匹配，因此我们每次在哈希表中检查是否存在 $x-1$ 即能判断是否需要跳过了。

代码 12.1: 最长连续序列

```rust
// 最长连续序列
fn longest_consecutive(nums: Vec<i32>) -> i32 {
    // 哈希表
    let mut num_set: std::collections::HashSet<i32> = nums.into_iter().collect();

    let mut longest_streak = 0;
    // 遍历哈希表
    for &num in &num_set {
        // 如果前一个数字不存在，那么它就是一个连续序列的起点
        if !num_set.contains(&(num - 1)) {
            let mut current_num = num;
            let mut current_streak = 1;
            // 寻找连续序列
            while num_set.contains(&(current_num + 1)) {
                current_num += 1;
                current_streak += 1;
            }

            // 更新最长的序列
            longest_streak = longest_streak.max(current_streak);
        }
    }
    longest_streak
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

12.2 冗余连接

No. 684 难度 Medium

树可以看成是一个连通且无环的 无向图。

给定往一棵 n 个节点 (节点值 1 n) 的树中添加一条边后的图。添加的边的两个顶点包含在 1 到 n 中间，且这条附加的边不属于树中已存在的边。图的信息记录于长度为 n 的二维数组 $edges$ ，$edges[i] = [a_i, b_i]$ 表示图中在 a_i 和 b_i 之间存在一条边。

请找出一条可以删去的边，删除后可使得剩余部分是一个有着 n 个节点的树。如果有多个答案，则返回数组 $edges$ 中最后出现的那个。

示例

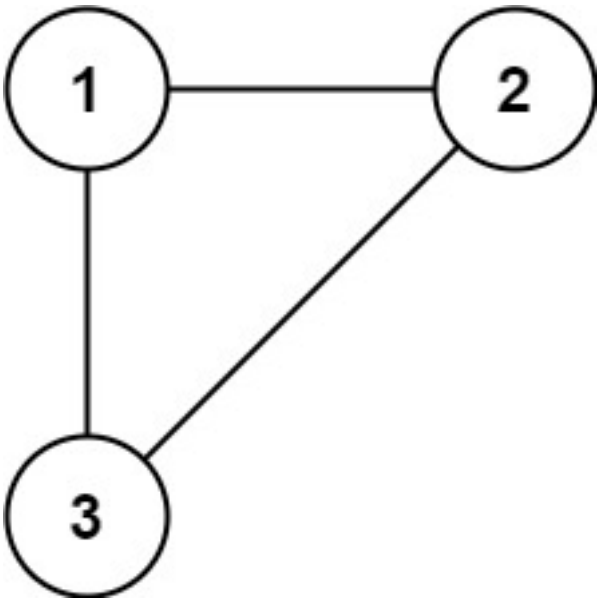


图 12.1. 冗余连接图

输入: edges = [[1,2], [1,3], [2,3]]
输出: [2,3]

解法一: 并查集

思路: 在一棵树中，边的数量比节点的数量少 1。如果一棵树有 n 个节点，则这棵树有 $n-1$ 条边。这道题中的图在树的基础上多了一条附加的边，因此边的数量也是 n。

树是一个连通且无环的无向图，在树中多了一条附加的边之后就会出现环，因此附加的边即为导致环出现的边。

可以通过并查集寻找附加的边。初始时，每个节点都属于不同的连通分量。遍历每一条边，判断这条边连接的两个顶点是否属于相同的连通分量。

- 如果两个顶点属于不同的连通分量，则说明在遍历到当前的边之前，这两个顶点之间不连通，因此当前的边不会导致环出现，合并这两个顶点的连通分量。
- 如果两个顶点属于相同的连通分量，则说明在遍历到当前的边之前，这两个顶点之间已经连通，因此当前的边导致环出现，为附加的边，将当前的边作为答案返回。

代码 12.2: 冗余连接

```rust
fn find_redundant_connection(edges: Vec<Vec<i32>>) -> Vec<i32> {
    let n = edges.len();
    // 初始化并查集
    let mut parent = vec![0; n + 1];
    for i in 1..=n {
        parent[i] = i;
    }

    // 遍历边
    for edge in edges {
        let a = edge[0] as usize;
        let b = edge[1] as usize;

        // 查找根节点
        let root_a = find(&parent, a);
        let root_b = find(&parent, b);

        // 如果根节点相同，说明有环
        if root_a == root_b {
            return edge;
        }
        // 否则，合并连通分量
        parent[root_a] = root_b;
    }
    vec![]
}

// 查找根节点
fn find(parent: &[i32], x: usize) -> usize {
    let mut x = x;
    while parent[x] != x as i32 {
        x = parent[x] as usize;
    }
    x
}
```

复杂度分析

- 时间复杂度：$O(n \log n)$，其中 n 为序列长度。每个位置至多被遍历两次。
- 空间复杂度：$O(n)$。只需要常数的空间存放若干变量。

13

搜索

二叉搜索数、广度优先搜索、深度优先搜索、回溯算法、剪枝算法等搜索算法是算法中常见的技巧。搜索算法可以解决很多问题，例如图的遍历、路径搜索、状态搜索等。

13.1 二叉搜索树的最小绝对差

No 530 简单 Easy

给你一个二叉搜索树的根节点 $root$ ，返回树中任意两不同节点值之间的最小差值。

差值是一个正数，其数值等于两值之差的绝对值。

示例 输入: root = [4,2,6,1,3]
输出: 1

解法一: 中序遍历

思路: 考虑对升序数组 a 求任意两个元素之差的绝对值的最小值，答案一定为相邻两个元素之差的最小值，即

$$ans = \min_{i=0}^{n-2} \{a[i+1] - a[i]\}$$

其中 n 为数组 a 的长度。其他任意间隔距离大于等于 2 的下标对 (i, j) 的元素之差一定大于下标对 $(i, i+1)$ 的元素之差，故不需要再被考虑。

回到本题，本题要求二叉搜索树任意两节点差的绝对值的最小值，而我们知道二叉搜索树有个性质为 二叉搜索树中序遍历得到的值序列是递增有序的 ，因此我们只要得到中序遍历后的值序列即能用上文提及的方法来解决。

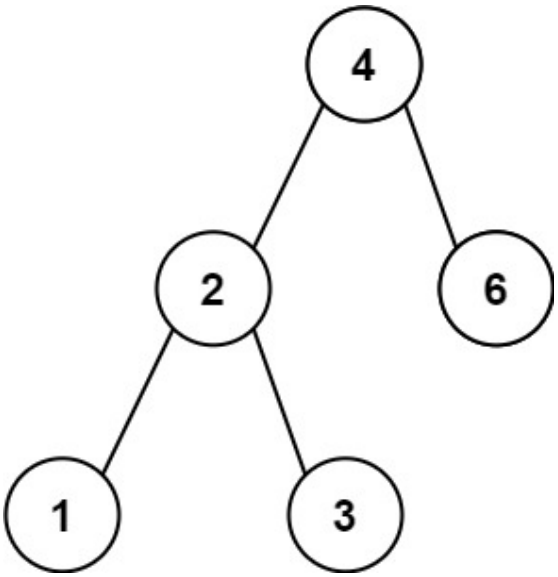


图 13.1. 二叉搜索树

朴素的方法是经过一次中序遍历将值保存在一个数组中再进行遍历求解，我们也可以在中序遍历的过程中用 pre 变量保存前驱节点的值，这样即能边遍历边更新答案，不再需要显式创建数组来保存，需要注意的是 pre 的初始值需要设置成任意负数标记开头，下文代码中设置为 -1。

二叉树的中序遍历有多种方式，包括递归、栈、Morris 遍历等，读者可选择自己最擅长的来实现。

代码 13.1: 二叉搜索树的最小绝对差

```rust
use std::cell::RefCell;
use std::rc::Rc;

fn get_minimum_difference(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
    let mut ans = std::i32::MAX;
    let mut pre = -1;

    fn dfs(node: Option<Rc<RefCell<TreeNode>>>, ans: &mut i32, pre: &mut i32)
    {
        if let Some(node) = node {
            let node = node.borrow();
            // Left node
            dfs(node.left.clone(), ans, pre);
            // In-order
            if *pre != -1 && node.val - *pre < *ans {
                *ans = node.val - *pre;
            }
            *pre = node.val;
            // Right node
            dfs(node.right.clone(), ans, pre);
        }
```

```
21          }
22
23          dfs(root, &mut ans, &mut pre);
24          ans
25      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

13.2 修剪二叉搜索树

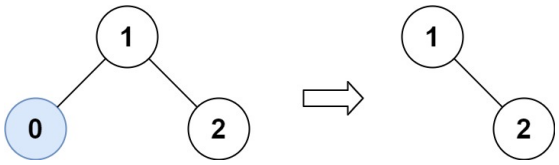
669 Medium

给你二叉搜索树的根节点 $root$ ，同时给定最小边界 low 和最大边界 $high$。通过修剪二叉搜索树，使得所有节点的值在 $[low, high]$ 中。修剪树不应该改变保留在树中的元素的相对结构 (即，如果没有被移除，原有的父代子代关系都应当保留)。可以证明，存在唯一的答案。

所以结果应当返回修剪好的二叉搜索树的新的根节点。注意，根节点可能会根据给定的边界发生改变。

示例 1

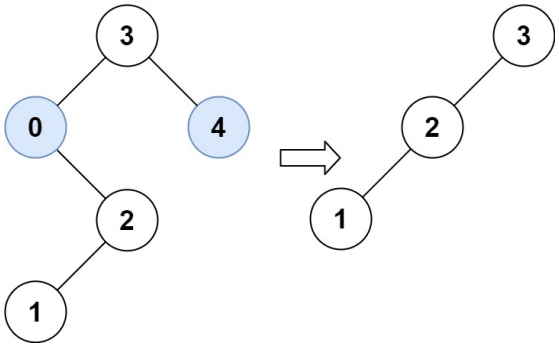
输入: root = [1,0,2], low = 1, high = 2



输出: [1,null,2]

示例 2

输入: root = [3,0,4,null,2,null,null,1], low = 1, high = 3



输出: [3,2,null,1]

解法一：递归

思路：对根结点 $root$ 进行深度优先遍历。对于当前访问的结点，如果结点为空结点，直接返回空结点；如果结点的值小于 low，那么说明该结点及它的左子树都不符合要求，我们返回对它的右结点进行修剪后的结果；如果结点的值大于 $high$，那么说明该结点及它的右子树都不符合要求，我们返回对它的左子树进行修剪后的结果；如果结点的值位于区间 $[low, high]$，我们将结点的左结点设为对它的左子树修剪后的结果，右结点设为对它的右子树进行修剪后的结果。

代码 13.2: 修剪二叉搜索树

```rust
fn trim_bst(root: Option<Rc<RefCell<TreeNode>>>, low: i32, high: i32) ->
  Option<Rc<RefCell<TreeNode>>> {
    if let Some(node) = root {
        let mut node = node.borrow_mut();
        // Trim left subtree
        if node.val < low {
            return trim_bst(node.right.take(), low, high);
        }
        // Trim right subtree
        if node.val > high {
            return trim_bst(node.left.take(), low, high);
        }
        // Trim both left and right subtrees
        node.left = trim_bst(node.left.take(), low, high);
        node.right = trim_bst(node.right.take(), low, high);
    }
    root
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

通过迭代的方法也可以解决这个问题，空间负责度可以降低到 $O(1)$，但是没有递归的方法简洁。

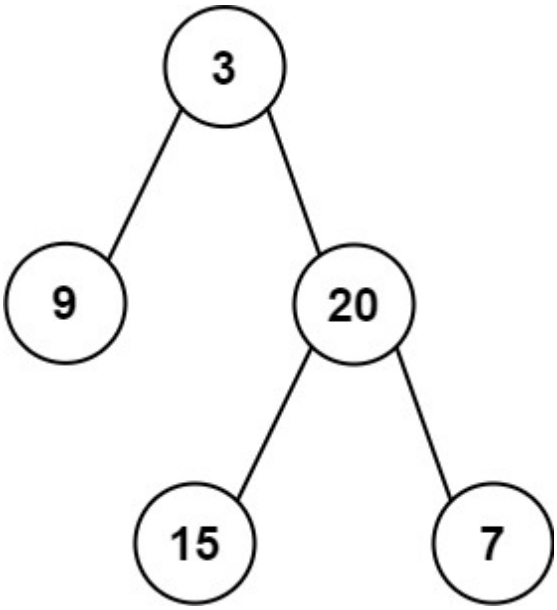
13.3 二叉树的层平均值

637 Easy

给定一个非空二叉树的根节点 $root$, 以数组的形式返回每一层节点的平均值。与实际答案相差 10^{-5} 以内的答案可以被接受。

示例 1

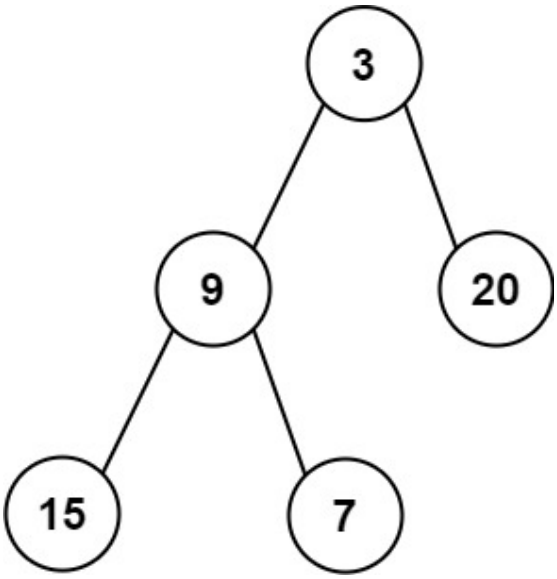
输入: root = [3,9,20,null,null,15,7]
输出: [3.00000,14.50000,11.00000]
解释:



第 0 层的平均值为 3, 第 1 层的平均值为 14.5, 第 2 层的平均值为 11 。因此返回 [3, 14.5, 11] 。

示例 2

输入: root = [3,9,20,15,7]



输出: [3.00000,14.50000,11.00000]

解法一: 广度优先搜索

思路: 前面的章节我们已经介绍了一个类似的题目，就是层序遍历，这一道题就当是加深印象吧。

从根节点开始搜索，每一轮遍历同一层的全部节点，计算该层的节点数以及该层的节点值之和，然后计算该层的平均值。

如何确保每一轮遍历的是同一层的全部节点呢？我们可以借鉴层次遍历的做法，广度优先搜索使用队列存储待访问节点，只要确保在每一轮遍历时，队列中的节点是同一层的全部节点即可。具体做法如下：

- 初始时，将根节点加入队列；
- 每一轮遍历时，将队列中的节点全部取出，计算这些节点的数量以及它们的节点值之和，并计算这些节点的平均值，然后将这些节点的全部非空子节点加入队列，重复上述操作直到队列为空，遍历结束。

由于初始时队列中只有根节点，满足队列中的节点是同一层的全部节点，每一轮遍历时都会将队列中的当前层节点全部取出，并将下一层的全部节点加入队列，因此可以确保每一轮遍历的是同一层的全部节点。

具体实现方面，可以在每一轮遍历之前获得队列中的节点数量 $size$，遍历时只遍历 $size$ 个节点，即可满足每一轮遍历的是同一层的全部节点。

代码 13.3: 二叉搜索树的最小绝对差

```rust
use std::collections::VecDeque;

fn average_of_levels(root: Option<Rc<RefCell<TreeNode>>>) -> Vec<f64> {
    let mut averages = Vec::new();
    if let Some(root) = root {
        let mut queue = VecDeque::new();
        queue.push_back(root);

        while !queue.is_empty() {
            let level_size = queue.len();
            let mut level_sum = 0;

            for _ in 0..level_size {
                if let Some(node) = queue.pop_front() {
                    let node = node.borrow();
                    level_sum += node.val;

                    if let Some(left) = &node.left {
                        queue.push_back(left.clone());
                    }

                    if let Some(right) = &node.right {
                        queue.push_back(right.clone());
                    }
                }
            }

            averages.push(level_sum as f64 / level_size as f64);
        }
    }

```

```
32          averages
33      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

层序遍历、前序遍历、中序遍历、后序遍历是必须掌握的技巧，这些技巧在解决树的问题时非常有用。

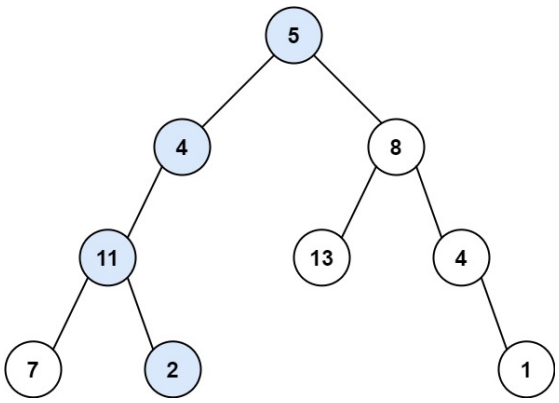
13.4 路径之和

No. 112 难度 Easy

给你二叉树的根节点 $root$ 和一个表示目标和的整数 $targetSum$ 。判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和 $targetSum$ 。如果存在，返回 $true$ ；否则，返回 $false$ 。

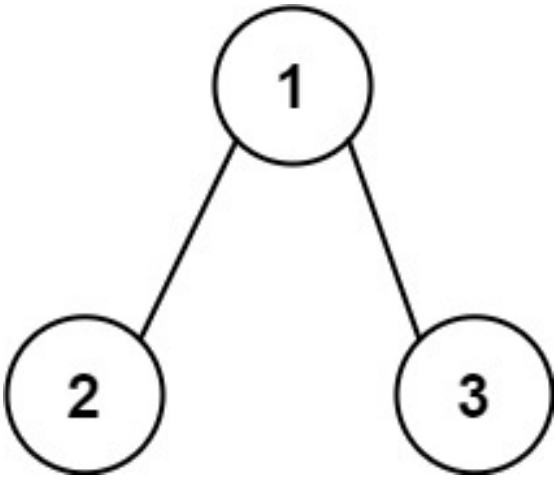
叶子节点是指没有子节点的节点。

示例 1
输入: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22



输出: true
解释: 等于目标和的根节点到叶节点路径如上图所示。

示例 2
输入: root = [], targetSum = 0
输出: false
解释: 由于树是空的，所以不存在根节点到叶子节点的路径。



解法一：递归

思路：观察要求我们完成的函数，我们可以归纳出它的功能：询问是否存在从当前节点 $root$ 到叶子节点的路径，满足其路径和为 sum。

假定从根节点到当前节点的值之和为 val，我们可以将这个大问题转化为一个小问题：是否存在从当前节点的子节点到叶子的路径，满足其路径和为 $sum - val$。

不难发现这满足递归的性质，若当前节点就是叶子节点，那么我们直接判断 sum 是否等于 val 即可（因为路径和已经确定，就是当前节点的值，我们只需要判断该路径和是否满足条件）。若当前节点不是叶子节点，我们只需要递归地询问它的子节点是否能满足条件即可。

代码 13.4: 路径之和

```
    fn has_path_sum(root: Option<Rc<RefCell<TreeNode>>>, sum: i32) -> bool {
        if let Some(node) = root {
            let node = node.borrow();
            // Leaf node
            if node.left.is_none() && node.right.is_none() {
                return sum == node.val;
            }
            // Non-leaf node
            return has_path_sum(node.left.clone(), sum - node.val)
                || has_path_sum(node.right.clone(), sum - node.val);
        }
        false
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

本题也可以使用广度优先搜索来解决，但是递归的方法更加简洁。

13.5 水域大小

No. 16.19 难度 Medium

你有一个用于表示一片土地的整数矩阵 $land$，该矩阵中每个点的值代表对应地点的海拔高度。若值为 0 则表示水域。由垂直、水平或对角连接的水域为池塘。池塘的大小是指相连接的水域的个数。编写一个方法来计算矩阵中所有池塘的大小，返回值需要从小到大排序。

示例 1
输入:

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

输出: [1,2,4]

解法一: 深度优先搜索

思路: 遍历矩阵 $land$，如果当前遍历的点 (i, j) 满足 $land[i][j] = 0$，那么对 (i, j) 进行深度优先搜索，深度优先搜索的过程如下:

如果 (i, j) 越界或 $land[i][j] \neq 0$，直接返回 0。

令 $land[i][j] = -1$，表示该点已经被搜索过，然后对该点的八个相邻点执行深度优先搜索。

返回值为执行的深度优先搜索的结果和加 1。

将所有结果放入一个数组内，然后从小到大进行排序，返回结果。

代码 13.5: 路径之和

```rust
fn pond_sizes(land: Vec<Vec<i32>>) -> Vec<i32> {
    let m = land.len();
    let n = land[0].len();
    let mut res = Vec::new();

    fn dfs(land: &mut Vec<Vec<i32>>, x: usize, y: usize) -> i32 {
        if x >= land.len() || y >= land[0].len() || land[x][y] != 0 {
            return 0;
        }
        land[x][y] = -1;

        let mut count = 1;

        let directions = vec![-1, 0, 1];
        for dx in &directions {
```

```
                for dy in &directions {
                    if *dx == 0 && *dy == 0 {
                        continue;
                    }
                    count += dfs(land, (x as i32 + dx) as usize, (y as i32 + dy)
        as usize);
                }
            }
            count
        }

        for i in 0..m {
            for j in 0..n {
                if land[i][j] == 0 {
                    res.push(dfs(&mut land.clone(), i, j));
                }
            }
        }

        res.sort();
        res
    }
```

复杂度分析

- 时间复杂度：$O(mn \times \log mn)$。
- 空间复杂度：$O(m \times n)$。

14

回溯

回溯算法是对树形或者图形结构执行一次深度优先遍历，实际上类似枚举的搜索尝试过程，在遍历的过程中寻找问题的解。

深度优先遍历有个特点：当发现已不满足求解条件时，就返回，尝试别的路径。此时对象类型变量就需要重置成为和之前一样，称为「状态重置」。

许多复杂的，规模较大的问题都可以使用回溯法，有「通用解题方法」的美称。实际上，回溯算法就是暴力搜索算法，它是早期的人工智能里使用的算法，借助计算机强大的计算能力帮助我们找到问题的解。

14.1 组合总和

No. 39 难度 Medium

给你一个 **无重复元素** 的整数数组 $candidates$ 和一个目标整数 $target$，找出 $candidates$ 中可以使数字和为目标数 $target$ 的所有不同组合，并以列表形式返回。你可以按任意顺序返回这些组合。

$candidates$ 中的同一个数字可以无限制重复被选取。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 $target$ 的不同组合数少于 150 个。

示例 1
输入: candidates = [2,3,6,7], target = 7
输出: [[2,2,3],[7]]
解释:
2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。

7 也是一个候选，7 = 7 。
仅有这两种组合。

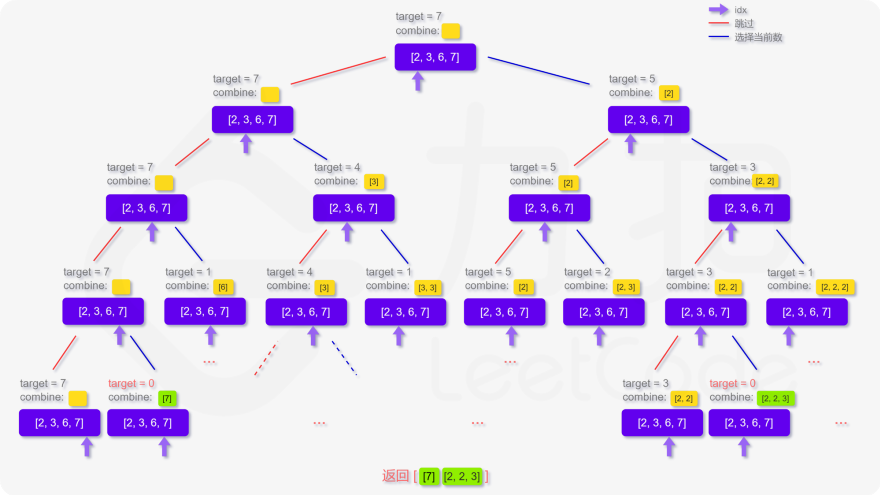
示例 2
输入:candidates = [2,3,5], target = 8
输出: [[2,2,2,2],[2,3,3],[3,5]]

解法一：搜索回溯

思路: 对于这类寻找所有可行解的题，我们都可以尝试用「搜索回溯」的方法来解决。

回到本题，我们定义递归函数 $dfs(target, combine, idx)$ 表示当前在 $candidates$ 数组的第 idx 位，还剩 $target$ 要组合，已经组合的列表为 $combine$。递归的终止条件为 $target \leq 0$ 或者 $candidates$ 数组被全部用完。那么在当前的函数中，每次我们可以选择跳过不用第 idx 个数，即执行 $dfs(target, combine, idx+1)$。也可以选择使用第 idx 个数，即执行 $dfs(target-candidates[idx], combine, idx)$，注意到每个数字可以被无限制重复选取，因此搜索的下标仍为 idx。

更形象化地说，如果我们将整个搜索过程用一个树来表达，即如下图呈现，每次的搜索都会延伸出两个分叉，直到递归的终止条件，这样我们就能不重复且不遗漏地找到所有可行解。左边的分叉代表选下一个数，右边的分叉代表选当前的数。



代码 14.1: 组合总和

```
1     fn combination_sum(candidates: Vec<i32>, target: i32) -> Vec<Vec<i32>> {
2         let mut ans: Vec<Vec<i32>> = Vec::new();
3         let mut comb: Vec<i32> = Vec::new();
4
5         // 搜索回溯
6         fn dfs(candidates: &Vec<i32>, target: i32, idx: usize, ans: &mut Vec<Vec<
        i32>>, comb: &mut Vec<i32>) {
7             if idx == candidates.len() {
8                 return;
9             }
```

```
10
11              // 找到一个解
12              if target == 0 {
13                  ans.push(comb.clone());
14                  return;
15              }
16
17              // 遍历左结点，下一个树
18              dfs(candidates, target, idx + 1, ans, comb);
19
20              // 遍历有节点，还是选当前数，但是 target 要减去当前数
21              if target - candidates[idx] >= 0 {
22                  comb.push(candidates[idx]);
23                  dfs(candidates, target - candidates[idx], idx, ans, comb);
24                  comb.pop();
25              }
26          }
27
28          dfs(&candidates, target, 0, &mut ans, &mut comb);
29          ans
30      }
```

复杂度分析

- 时间复杂度：$O(S)$，其中 S 为所有可行解的长度之和。。
- 空间复杂度：$O(target)$。

14.2 复原 IP 地址

No. 93 难度 Easy

有效 IP 地址正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用'.' 分隔。

例如："0.1.2.201" 和"192.168.1.1" 是有效 IP 地址，但是"0.011.255.245"、"192.168.1.312" 和"192.168@1.1" 是无效 IP 地址。给定一个只包含数字的字符串 s，用以表示一个 IP 地址，返回所有可能的有效 IP 地址，这些地址可以通过在 s 中插入'.' 来形成。你不能重新排序或删除 s 中的任何数字。你可以按任何顺序返回答案。

示例 1
输入: s = "25525511135"
输出: ["255.255.11.135","255.255.111.35"]

示例 2
输入: s = "0000"
输出: ["0.0.0.0"]

示例 2
输入: s = "101023"

输出: ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]

解法一: 回溯

思路: 我使用最朴素的回溯算法。

首先选选择第一个字符，然后再判断

代码 14.2: 恢复 IP 地址

```rust
fn restore_ip_addresses(s: String) -> Vec<String> {
    let mut res: Vec<String> = Vec::new();
    // Check for exceptional cases
    if s.len() < 4 || s.len() > 12 {
        return res;
    }

    backtracking(&s, 0, 0, String::new(), &mut res);

    res
}

// Backtracking function
fn backtracking(s: &str, start: usize, part: usize, cur_ip: String, res: &mut Vec<String>) {
    // Found all 4 parts and traversed the entire string
    if part == 4 && start == s.len() {
        res.push(cur_ip[1..].to_string()); // Remove the leading '.'
        return;
    }
    // Exceptional case
    if part > 4 {
        return;
    }

    // Traverse 3 characters starting from 'start'
    for i in start..start + 3.min(s.len() - start) {
        // If the first character of the current part is '0', we won't process the second and third characters
        if s.chars().nth(start).unwrap() == '0' && i != start {
            continue;
        }

        // If it's a valid character, i.e., 0 <= part <= 255, move on to the next part
        if let Ok(num) = s[start..=i].parse::<u8>() {
            backtracking(s, i + 1, part + 1, format!("{}.{}", cur_ip, num), res);
        }
    }
}
```

这个国外的同学的解法要比官方的题解更简洁。更有甚者，国外还有一种更粗暴的解法，使用四重循环来解:

代码 14.3: 四重循环恢复 IP 地址

```rust
fn restore_ip_addresses(s: String) -> Vec<String> {
    let mut res: Vec<String> = Vec::new();
    let mut ans: String = String::new();

    if s.len() < 4 || s.len() > 12 {
        return res;
    }

    for a in 1..=3 {
        for b in 1..=3 {
            for c in 1..=3 {
                let rest = s.len() - a - b - c;
                if rest > 3 {
                    continue;
                }

                for d in 1..=rest {
                    let A: u32 = s[..a].parse().unwrap();
                    let B: u32 = s[a..a+b].parse().unwrap();
                    let C: u32 = s[a+b..a+b+c].parse().unwrap();
                    let D: u32 = s[a+b+c..a+b+c+d].parse().unwrap();

                    if A <= 255 && B <= 255 && C <= 255 && D <= 255 {
                        ans = format!("{}.{}.{}.{}", A, B, C, D);

                        if ans.len() == s.len() + 3 {
                            res.push(ans.clone());
                        }
                    }
                }
            }
        }
    }

    res
}
```

这两种解法都会击败 100% 的 Go 用户，为啥四重循环性能还这么高，不是时间复杂度为 $O(n^4)$ 么？我们得灵活的看，这个四重循环每个循环也就执行了 4 次，所以时间复杂度也就是 $O(4^4)$，这个是一个常数级别的时间复杂度，所以性能还是很高的。空间复杂度也是 $O(1)$，因为我们最多就处理 12 个字符。

14.3 优美的排列

No. 526 难度 Medium

假设有从 1 到 n 的 n 个整数。用这些整数构造一个数组 $perm$（下标从 1 开始），只要满足下述条件之一，该数组就是一个优美的排列：

- $perm[i]$ 能够被 i 整除
- i 能够被 $perm[i]$ 整除

给你一个整数 n ，返回可以构造的优美排列的数量。

示例

输入: n = 2
输出: 2
解释:

```
第 1 个优美的排列是 [1,2]:
    - perm[1] = 1 能被 i = 1 整除
    - perm[2] = 2 能被 i = 2 整除
第 2 个优美的排列是 [2,1]:
    - perm[1] = 2 能被 i = 1 整除
    - i = 2 能被 perm[2] = 1 整除
```

解法一: 回溯

思路: 我们可以使用回溯法解决本题，从左向右依次向目标排列中放入数即可。

具体地，我们定义函数 $backtrack(index, n)$，表示尝试向位置 $index$ 放入数。其中 n 表示排列的长度。在当前函数中，我们首先找到一个符合条件的未被使用过的数，然后递归地执行 $backtrack(index + 1, n)$，当该函数执行完毕，回溯到当前层，我们再尝试下一个符合条件的未被使用过的数即可。

回溯过程中，我们可以用 vis 数组标记哪些数被使用过，每次我们选中一个数 x，我们就将 vis 标记为 $true$，回溯完成后，我们再将其置为 $false$。

特别地，为了优化回溯效率，我们可以预处理每个位置的符合条件的数有哪些，用二维数组 $match$ 保存。当我们尝试向位置 $index$ 放入数时，我们只需要遍历 $match$ 即可。

代码 14.4: 优美的排列

```rust
fn count_arrangement(n: i32) -> i32 {
    let mut ans = 0;
    let mut vis = vec![false; (n + 1) as usize];
    let mut match_arr = vec![vec![]; (n + 1) as usize];

    // Preprocessing
    for i in 1..=n {
        for j in 1..=n {
            if i % j == 0 || j % i == 0 {
                match_arr[i as usize].push(j);
            }
        }
    }

    // Backtracking function
```

```
16          fn backtrack(index: i32, vis: &mut Vec<bool>, match_arr: &Vec<Vec<i32>>,
        ans: &mut i32) {
17              if index > n {
18                  *ans += 1;
19                  return;
20              }
21
22              for &x in &match_arr[index as usize] {
23                  if !vis[x as usize] {
24                      vis[x as usize] = true;
25                      backtrack(index + 1, vis, match_arr, ans);
26                      vis[x as usize] = false;
27                  }
28              }
29          }
30
31          backtrack(1, &mut vis, &match_arr, &mut ans);
32
33          ans
34      }
```

复杂度分析

- 时间复杂度：$O(n!)$。
- 空间复杂度：$O(n^2)$。

15

动态规划

动态规划（英语：Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

动态规划常常适用于有重叠子问题和最优子结构性质的问题，并且记录所有子问题的结果，因此动态规划方法所耗时间往往远少于朴素解法。

动态规划有自底向上和自顶向下两种解决问题的方式。自顶向下即记忆化递归，自底向上就是递推。

15.1 爬楼梯

No. 70 难度 Easy

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？

示例

输入: n = 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

解法一：动态规划

思路：我们用 $f(x)$ 表示爬到第 x 级台阶的方案数，考虑最后一步可能跨了一级台阶，也可能跨了两级台阶，所以我们可以列出如下式子：

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

它意味着爬到第 x 级台阶的方案数是爬到第 $x-1$ 级台阶的方案数和爬到第 $x-2$ 级台阶的方案数的和。很好理解，因为每次只能爬 1 级或 2 级，所以 $f(x)$ 只能从 $f(x-1)$ 和 $f(x-2)$ 转移过来，而这里要统计方案总数，我们就需要对这两项的贡献求和。

以上是动态规划的转移方程，下面我们来讨论边界条件。我们是从第 0 级开始爬的，所以从第 0 级爬到第 0 级我们可以看作只有一种方案，即 $f(0) = 1$；从第 0 级到第 1 级也只有一种方案，即爬一级，$f(1) = 1$。这两个作为边界条件就可以继续向后推导出第 n 级的正确结果。我们不妨写几项来验证一下，根据转移方程得到 $f(2) = 2$，$f(3) = 3$，$f(4) = 5$，……，我们把这些情况都枚举出来，发现计算的结果是正确的。

我们不难通过转移方程和边界条件给出一个时间复杂度和空间复杂度都是 $O(n)$ 的实现，但是由于这里的 $f(x)$ 只和 $f(x-1)$ 与 $f(x-2)$ 有关，所以我们可以用「滚动数组思想」把空间复杂度优化成 $O(1)$。下面的代码中给出的就是这种实现。

代码 15.1: 爬楼梯

```rust
fn climb_stairs(n: i32) -> i32 {
    let mut p = 0;
    let mut q = 0;
    let mut r = 1;
    for i in 1..=n {
        p = q; // f(x-2)
        q = r; // f(x-1)
        r = p + q;
    }
    r
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

15.2 打家劫舍

No. 198 难度 Medium

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

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

示例 1

输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4 。

示例 2

输入: [2,7,9,3,1]
输出: 12
解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
偷窃到的最高金额 = 2 + 9 + 1 = 12 。

解法一: 动态规划

思路: 首先考虑最简单的情况。如果只有一间房屋，则偷窃该房屋，可以偷窃到最高总金额。如果只有两间房屋，则由于两间房屋相邻，不能同时偷窃，只能偷窃其中的一间房屋，因此选择其中金额较高的房屋进行偷窃，可以偷窃到最高总金额。

如果房屋数量大于两间，应该如何计算能够偷窃到的最高总金额呢？对于第 k $(k > 2)$ 间房屋，有两个选项：

- 偷窃第 k 间房屋，那么就不能偷窃第 $k-1$ 间房屋，偷窃总金额为前 $k-2$ 间房屋的最高总金额与第 k 间房屋的金额之和。
- 不偷窃第 k 间房屋，偷窃总金额为前 $k-1$ 间房屋的最高总金额。

在两个选项中选择偷窃总金额较大的选项，该选项对应的偷窃总金额即为前 k 间房屋能偷窃到的最高总金额。

用 $dp[i]$ 表示前 i 间房屋能偷窃到的最高总金额，那么就有如下的状态转移方程：

$$dp[i] = \max(dp[i-2] + nums[i], dp[i-1])$$

边界条件为：

$$\begin{cases} dp[0] = nums[0] & , \\ dp[1] = \max(nums[0], nums[1]) & , \end{cases}$$

最终的答案即为 $dp[n-1]$，其中 n 是数组的长度。

代码 15.2: 单调栈计算买卖股票的最佳时机

```
1       fn rob(nums: Vec<i32>) -> i32 {
2           if nums.is_empty() {
```

```
3               return 0;
4           }
5           if nums.len() == 1 {
6               return nums[0];
7           }
8
9           let mut dp = vec![0; nums.len()];
10          dp[0] = nums[0];
11          dp[1] = nums[0].max(nums[1]);
12          for i in 2..nums.len() {
13              dp[i] = (dp[i - 2] + nums[i]).max(dp[i - 1]);
14          }
15          dp[nums.len() - 1]
16      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

如果使用滚动数组来代替 dp，可以将空间复杂度降到 $O(1)$。

15.3 最小的必要团队

No. 1125 难度 Hard

作为项目经理，你规划了一份需求的技能清单 req_skills，并打算从备选人员名单 $people$ 中选出些人组成一个「必要团队」（编号为 i 的备选人员 $people[i]$ 含有一份该备选人员掌握的技能列表）。

所谓「必要团队」，就是在这个团队中，对于所需求的技能列表 req_skills 中列出的每项技能，团队中至少有一名成员已经掌握。可以用每个人的编号来表示团队中的成员：

例如，团队 $team = [0, 1, 3]$ 表示掌握技能分别为 $people[0]$，$people[1]$，和 $people[3]$ 的备选人员。请你返回任一规模最小的必要团队，团队成员用人员编号表示。你可以按任意顺序返回答案，题目数据保证答案存在。

示例

输入:

```
req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","
```

输出: [0,2]

解法一：动态规划

思路: 题目输入数组 req_skills 的长度最大为 16，req_skills 中的每一项，被选择或者不被选择，总共的组合情况为 2^{16} 种。因此可以通过「状态压缩」来表示一个技能集合。

我们将每一个技能 $req_skills[i]$ 映射到一个二进制数的第 i 位。例如：

req_skills$[0]$ 用 $2^0 = (1 << 0) = 1$ 来表示。*req_skills*$[1]$ 用 $2^1 = (1 << 1) = 2$ 来表示。以此类推，如此一来我们就可以用一个数字来表示一个技能集合。同时两个集合的并集计算，就可以转化为两个整数的或运算。

我们采用自下而上的「动态规划」的思路来解题，用 $dp[i]$ 来表示状态，状态含义是满足技能集合为 i 的最小人数的数组。初始化状态是 $dp[0]$，为空数组，因为如果不需要任何技能，不用任何人就可以完成。

我们首先依次遍历 $peoples$，求出当前这个人所有的技能集合 cur_skill。然后遍历 dp 表中的结果 $dp[prev]$，其中原来的技能集合用 $prev$ 来表示。设加入当前这个人后新的技能集合是 $comb$，由原来的技能集合和当前技能集合求并集后，可以得到：$comb = prev \mid cur\_skill$。状态转移的规则是，如果 $dp[comb]$ 不存在，或 $dp[prev]$ 的长度加上 1 小于 $dp[comb].size()$，那么我们就需要更新 $dp[comb]$ 为 $dp[prev]$，再将当前人加入到 $dp[comb]$。这里我们更新 $dp[comb]$ 时候，可以采用直接覆盖的方式，因为更新后的结果因为已经包含了当前员工的技能，所以不会再次满足转移规则，而发生重复转移。

最后，所有技能的集合用 $(1 << n) - 1$ 来表示，其中 n 是 req_skills 的长度，我们只需要返回最终答案 $dp[(1 << n)-1]$。

代码 15.3: 最小的必要团队

```rust
fn smallest_sufficient_team(req_skills: Vec<String>, people: Vec<Vec<String
    >>) -> Vec<i32> {
    let n = req_skills.len();
    let m = people.len();
    let mut skill_index = std::collections::HashMap::new();
    for (i, skill) in req_skills.iter().enumerate() {
        skill_index.insert(skill.clone(), i);
    }
    let mut dp = vec![vec![]; 1 << n];
    dp[0] = vec![];
    for i in 0..m {
        let mut cur_skill = 0;
        for s in &people[i] {
            cur_skill |= 1 << skill_index[s];
        }
        for prev in 0..dp.len() {
            if dp[prev].is_empty() {
                continue;
            }
            let comb = prev | cur_skill;
            if dp[comb].is_empty() || dp[prev].len() + 1 < dp[comb].len() {
                dp[comb] = dp[prev].clone();
                dp[comb].push(i as i32);
            }
        }
    }
    dp[(1 << n) - 1].clone()
}
```

复杂度分析

- 时间复杂度：$O(m^2 \times 2^n)$，其中 n 是 req_skills 的长度，m 是 $peoples$ 的长度。
- 空间复杂度：$O(m \times 2^n)$。

有的动态规划题很容易，有的却很难，鸡蛋掉落是谷歌的一道面试题，相当的 Hard，你就知道名企面试有多难了吧。

16

贪心

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择，就能得到问题的答案。贪心算法需要充分挖掘题目中条件，没有固定的模式，解决有贪心算法需要一定的直觉和经验。

贪心算法不是对所有问题都能得到整体最优解。能使用贪心算法解决的问题具有「贪心选择性质」。「贪心选择性质」严格意义上需要数学证明。能使用贪心算法解决的问题必须具备「无后效性」，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

16.1 最长回文串

409 Easy

给定一个包含大写字母和小写字母的字符串 s，返回通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如"Aa" 不能当做一个回文字符串。

示例

输入: s = "abccccdd"
输出: 7
解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

解法一: 贪心

回文串是一个正着读和反着读都一样的字符串。以回文中心为分界线，对于回文串中左侧的字符 ch，在右侧对称的位置也会出现同样的字符。例如在字符串"abba" 中，回

文中心是"ab|ba" 中竖线的位置，而在字符串"abcba" 中，回文中心是"ab(c)ba" 中的字符"c" 本身。我们可以发现，在一个回文串中，只有最多一个字符出现了奇数次，其余的字符都出现偶数次。

那么我们如何通过给定的字符构造一个回文串呢？我们可以将每个字符使用偶数次，使得它们根据回文中心对称。在这之后，如果有剩余的字符，我们可以再取出一个，作为回文中心。

思路:

对于每个字符 ch，假设它出现了 v 次，我们可以使用该字符 $v/2*2$ 次，在回文串的左侧和右侧分别放置 $v/2$ 个字符 ch，其中 $/$ 为整数除法。例如若"a" 出现了 5 次，那么我们可以使用"a" 的次数为 4，回文串的左右两侧分别放置 2 个"a"。

如果有任何一个字符 ch 的出现次数 v 为奇数（即 $v\%2==1$），那么可以将这个字符作为回文中心，注意只能最多有一个字符作为回文中心。在代码中，我们用 ans 存储回文串的长度，由于在遍历字符时，ans 每次会增加 $v/2*2$，因此 ans 一直为偶数。但在发现了第一个出现次数为奇数的字符后，我们将 ans 增加 1，这样 ans 变为奇数，在后面发现其它出现奇数次的字符时，我们就不改变 ans 的值了。

代码 16.1: 最长回文串

```rust
fn longest_palindrome(s: String) -> i32 {
    let mut count = [0; 128];
    let length = s.len();

    for c in s.chars() {
        count[c as usize] += 1;
    }

    let mut ans = 0;
    for v in count.iter() {
        ans += v / 2 * 2;
        if v % 2 == 1 && ans % 2 == 0 {
            ans += 1;
        }
    }

    ans
}
```

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(S)$。

16.2 任务调度器

No. 621 难度 Medium

给你一个用字符数组 $tasks$ 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间，CPU 可以完成一个任务，或者处于待命状态。

然而，两个相同种类的任务之间必须有长度为整数 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的最短时间。

示例

输入: tasks = ["A","A","A","B","B","B"], n = 2
输出: 8
解释: A -> B -> (待命) -> A -> B -> (待命) -> A -> B
在本示例中，两个相同类型任务之间必须间隔长度为 n = 2 的冷却时间，而执行一个任务只需要一个单位时间，所以中间出现了（待命）状态。

解法一：贪心

思路: 官方题解的过于复杂了，网友“人类二分法精华”给出了一个简洁的贪心解法

代码 16.2: 任务调度器

```rust
    fn least_interval(tasks: Vec<char>, n: i32) -> i32 {
        let mut hash = vec![0; 26];
        for i in 0..tasks.len() {
            hash[(tasks[i] as u8 - b'A') as usize] += 1;
        }
        hash.sort();

        let min_len = (n + 1) * (hash[25] - 1) + 1;

        let mut i = 24;
        while i >= 0 && hash[i] == hash[25] {
            i -= 1;
        }

        let remaining_tasks = tasks.len() as i32 - min_len;
        let additional_slots = remaining_tasks - i * (hash[25] - 1);

        let result = min_len + additional_slots.max(0);
        result.max(tasks.len() as i32)
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

巧妙。

16.3 跳跃游戏

No 55 难度 Medium

给你一个非负整数数组 $nums$，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，如果可以，返回 $true$；否则，返回 $false$ 。

示例

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。

解法一: 贪心

思路:

1. 如果某一个作为起跳点的格子 i 可以跳跃的距离是 3，那么表示后面 3 个格子 $i+1$、$i+2$、$i+3$ 都可以作为起跳点
2. 可以对每一个能作为起跳点的格子都尝试跳一次，把能跳到最远的距离不断更新
3. 如果可以一直跳到最后，就成功了

代码 16.3: 跳跃游戏

```rust
    fn can_jump(nums: Vec<i32>) -> bool {
        let mut max_pos = 0;
        for (i, &num) in nums.iter().enumerate() {
            // If we can't reach the current position, return false
            if i > max_pos {
                return false;
            }

            // Update the maximum position we can reach
            max_pos = max_pos.max(i + num as usize);

            // If the maximum position is beyond or equal to the last index, we can reach the end
            if max_pos >= nums.len() - 1 {
                return true;
            }
        }
        false
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

这道题也可以用动态规划解，状态转移方程是 $dp[i] = max(dp[i-1], i + nums[i])$, 因为不需要先前的 dp, 所以只使用 $maxPos$ 一个变量即可。

17

位运算

位操作（Bit Manipulation）是程序设计中对位模式或二进制数的一元和二元操作。在许多古老的微处理器上，位运算比加减运算略快，通常位运算比乘除法运算要快很多。在现代编程语言中，情况并非如此，很多编程语言的解释器都会基本的运算进行了优化，因此我们在实际开发中可以不必做一些编译器已经帮我们做好的优化，而就写出代码本身所要表现的意思。

位运算的问题，很多都很有技巧性，大家需要掌握一定的位运算的应用，达到融会贯通的目的。

正好,春节看到一个介绍二进制的资源,是圣克鲁斯加利福尼亚大学的老师 Eric Lengyel 的总结，我觉得很好，就分享给大家。你也可以直接下载二进制基础。

Binary Fundamentals

Powers of Two

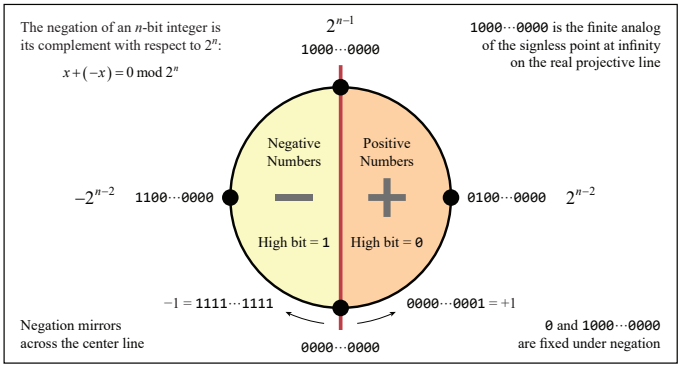
Power	Decimal	Hexadecimal
2^{1}	2	0x00000002
2^{2}	4	0x00000004
2^{3}	8	0x00000008
2^{4}	16	0x00000010
2^{5}	32	0x00000020
2^{6}	64	0x00000040
2^{7}	128	0x00000080
2^{8}	256	0x00000100
2^{9}	512	0x00000200
2^{10}	1024	0x00000400
2^{11}	2048	0x00000800
2^{12}	4096	0x00001000

Power	Decimal	Hexadecimal
2^{13}	8192	0x00002000
2^{14}	16,384	0x00004000
2^{15}	32,768	0x00008000
2^{16}	65,536	0x00010000
2^{17}	131,072	0x00020000
2^{18}	262,144	0x00040000
2^{19}	524,288	0x00080000
2^{20}	1,048,576	0x00100000
2^{21}	2,097,152	0x00200000
2^{22}	4,194,304	0x00400000
2^{23}	8,388,608	0x00800000
2^{24}	16,777,216	0x01000000

Two's Complement

The negation of an n-bit integer is its complement with respect to 2^n:

$$x + (-x) = 0 \bmod 2^n$$



Logical Complement

NOT

Bitwise NOT

`~x`

x	~x
0	1
1	0

Logical Identities

Unary	Binary
`~x = -x - 1`	`~(x & y) = ~x | ~y`
`-x = ~x + 1`	`~(x | y) = ~x & ~y`
`--~x = x + 1`	`~(x ^ y) = ~x ^ y` or `x ^ ~y`
`-~x = x - 1`	

Binary Logical Operations

AND

Bitwise AND

`x & y`

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

OR

Bitwise OR

`x | y`

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

NAND

Not AND

`~(x & y)`

x	y	~(x & y)
0	0	1
0	1	1
1	0	1
1	1	0

NOR

Not OR

`~(x | y)`

x	y	~(x | y)
0	0	1
0	1	0
1	0	0
1	1	0

ANDC

AND with complement

`x & ~y`

x	y	x & ~y
0	0	0
0	1	0
1	0	1
1	1	0

ORC

OR with complement

`x | ~y`

x	y	x | ~y
0	0	1
0	1	0
1	0	1
1	1	1

XOR

Exclusive OR

`x ^ y`

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

Exclusive NOR

`~(x ^ y)`

x	y	~(x ^ y)
0	0	1
0	1	0
1	0	0
1	1	1

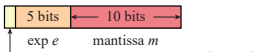
Bit Manipulation

Formula	Operation / Effect	Illustration
`x & (x - 1)`	Clear lowest 1 bit. If result is zero, then `x` is zero or 2^k. `000···000` is unchanged.	1 0 1 1 **1** 0 0 0 → 1 0 1 1 **0** 0 0 0
`x | (x + 1)`	Set lowest 0 bit. `111···111` is unchanged.	0 1 1 0 **0** 1 1 1 → 0 1 1 0 **1** 1 1 1
`x | (x - 1)`	Set all bits to right of lowest 1 bit. `000···000` becomes `111···111`.	1 0 1 1 **1** 0 0 0 → 1 0 1 1 **1** 1 1 1
`x & (x + 1)`	Clear all bits to right of lowest 0 bit. If result is zero, then `x` is zero or $2^k - 1$. `111···111` becomes `000···000`.	0 1 1 0 **0** 1 1 1 → 0 1 1 0 **0** 0 0 0
`x & -x`	Extract lowest 1 bit. `000···000` is unchanged.	1 0 1 1 **1** 0 0 0 → 0 0 0 0 **1** 0 0 0
`~x & (x + 1)`	Extract lowest 0 bit (as a 1 bit). `111···111` becomes `000···000`.	0 1 1 0 **0** 1 1 1 → 0 0 0 0 **1** 0 0 0

Floating-Point

Half precision

16-bit floating-point

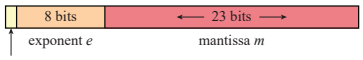


$$\text{value} = (-1)^s \, 2^{e-15} \left(1 + \frac{m}{2^{10}}\right)$$

Single precision

32-bit floating-point

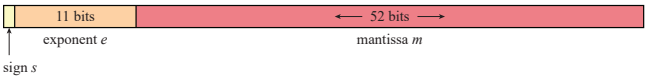
$$\text{value} = (-1)^s \, 2^{e-127} \left(1 + \frac{m}{2^{23}}\right)$$



Double precision

64-bit floating-point

$$\text{value} = (-1)^s \, 2^{e-1023} \left(1 + \frac{m}{2^{52}}\right)$$



Special Floating-Point Value	Half	Float	Double
+0.0	0x0000	0x00000000	0x00000000_00000000
+1.0	0x3C00	0x3F800000	0x3FF00000_00000000
Positive infinity	0x7C00	0x7F800000	0x7FF00000_00000000
Smallest positive normalized value	0x0400	0x00800000	0x00100000_00000000
Upper limit of non-integer values	0x6400	0x4B000000	0x43300000_00000000
Largest representable positive value	0x7BFF	0x7F7FFFFF	0x7FEFFFFF_FFFFFFFF

Mask Creation

Formula	Operation / Effect	Illustration
`~x | (x - 1)`	Create mask for all bits other than lowest 1 bit. `000···000` becomes `111···111`.	1 0 1 1 **1** 0 0 0 → 1 1 1 1 **0** 1 1 1
`x | ~(x + 1)`	Create mask for all bits other than lowest 0 bit. `111···111` is unchanged.	0 1 1 0 **0** 1 1 1 → 1 1 1 1 **0** 1 1 1
`x | -x`	Create mask for bits left of lowest 1 bit, inclusive. `000···000` is unchanged.	1 0 1 1 **1** 0 0 0 → 1 1 1 1 **1** 0 0 0
`x ^ -x`	Create mask for bits left of lowest 1 bit, exclusive. `000···000` is unchanged.	1 0 1 1 **1** 0 0 0 → 1 1 1 1 **0** 0 0 0
`~x | (x + 1)`	Create mask for bits left of lowest 0 bit, inclusive. `111···111` becomes `000···000`.	0 1 1 0 **0** 1 1 1 → 1 1 1 1 **1** 0 0 0
`~x ^ (x + 1)`	Create mask for bits left of lowest 0 bit, exclusive. `111···111` becomes `000···000`.	0 1 1 0 **0** 1 1 1 → 1 1 1 1 **0** 0 0 0
`x ^ (x - 1)`	Create mask for bits right of lowest 1 bit, inclusive. `000···000` becomes `111···111`.	1 0 1 1 **1** 0 0 0 → 0 0 0 0 **1** 1 1 1
`~x & (x - 1)`	Create mask for bits right of lowest 1 bit, exclusive. `000···000` becomes `111···111`.	1 0 1 1 **1** 0 0 0 → 0 0 0 0 **0** 1 1 1
`x ^ (x + 1)`	Create mask for bits right of lowest 0 bit, inclusive. `111···111` is unchanged.	0 1 1 0 **0** 1 1 1 → 0 0 0 0 **1** 1 1 1
`x & (~x - 1)`	Create mask for bits right of lowest 0 bit, exclusive. `111···111` is unchanged.	0 1 1 0 **0** 1 1 1 → 0 0 0 0 **0** 1 1 1

17.1 只出现一次的数字

No. 136 难度 Easy

给你一个非空整数数组 $nums$，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。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

示例

输入: nums = [2,2,1]
输出: 1

解法一: 位运算

思路: 对于这道题，可使用异或运算 $\oplus$。异或运算有以下四个性质。

- 归零律：任何数和 0 做异或运算，结果仍然是原来的数，即 $a \oplus 0 = a$。
- 端元律: 任何数和其自身做异或运算，结果是 0，即 $a \oplus a = 0$。
- 交换律: 对于任意两个二进制值 a 和 b，即 $a \oplus b = b \oplus a$。
- 结合律: 对于任意三个二进制值 a 和 b，即 $(a \oplus b) \oplus c = a \oplus (b \oplus c)$。

异或运算满足交换律和结合律，即 $a \oplus b \oplus a = b \oplus a \oplus a = b \oplus (a \oplus a) = b \oplus 0 = b$。

数组中的全部元素的异或运算结果即为数组中只出现一次的数字。

你知道了这个结论解这个题易如反掌，不知道的就会抓耳挠腮了。

代码 17.1: 只出现一次的数字

```rust
fn single_number(nums: Vec<i32>) -> i32 {
    let mut ans = 0;
    for v in nums {
        ans ^= v;
    }
    ans
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

17.2 位 1 的个数

No. 191 难度 Easy

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为'1' 的个数（也被称为汉明重量）。

示例

输入: n = 00000000000000000000000000001011
输出: 3

解法一：位运算

思路: 观察这个运算：$n \ \& \ (n-1)$，其运算结果恰为把 n 的二进制位中的最低位的 1 变为 0 之后的结果。

这样我们可以利用这个位运算的性质加速我们的检查过程，在实际代码中，我们不断让当前的 n 与 $n-1$ 做与运算，直到 n 变为 0 即可。因为每次运算会使得 n 的最低位的 1 被翻转，因此运算次数就等于 n 的二进制位中 1 的个数。

代码 17.2: 位 1 的个数

```rust
fn hamming_weight(num: u32) -> i32 {
    let mut res = 0;
    let mut n = num;
    while n != 0 {
        n &= n - 1;
        res += 1;
    }
    res
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

更快的算法是查表，把 uint32、uint64 可以分成每 8 位分别查表，这样可以大大提高速度。

"Hacker's Delight" 一书中的第五章提供了一个 Counting Bits 算法，这•也是 Go 语言中 $funcOnesCount64(xuint64)int$ 的实现。这个算法的时间复杂度是 $O(1)$，空间复杂度是 $O(1)$。

代码 17.3: 位 1 的个数

```rust
const M0: u64 = 0x5555555555555555; // 01010101 ...
const M1: u64 = 0x3333333333333333; // 00110011 ...
const M2: u64 = 0x0f0f0f0f0f0f0f0f; // 00001111 ...
const M3: u64 = 0x00ff00ff00ff00ff; // etc.
const M4: u64 = 0x0000ffff0000ffff;

fn ones_count_64(x: u64) -> i32 {
    let m: u64 = (1 << 64) - 1;
    let mut x = (x >> 1) & (M0 & m) + (x & (M0 & m));
```

```
10          x = (x >> 2) & (M1 & m) + (x & (M1 & m));
11          x = (x >> 4 + x) & (M2 & m);
12          x += x >> 8;
13          x += x >> 16;
14          x += x >> 32;
15          (x as i32) & ((1 << 7) - 1)
16      }
```

17.3 消失的数字

No. 17.04 难度 Easy

数组 $nums$ 包含从 0 到 n 的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在 $O(n)$ 时间内完成吗？

示例

输入: [3,0,1], n = 3
输出: 2

解法一：异或

思路: 利用异或的特性，$res = res \oplus x \oplus x$。对同一个值异或两次，那么结果等于它本身，所以我们对 res 从 $0 - nums.length$ 进行异或，同时对 $nums$ 数组中的值进行异或，出现重复的会消失，所以最后 res 的值是只出现一次的数字，也就是 $nums$ 数组中缺失的那个数字。

通过这个处理，就和第一节的题目一样，我们可以得到缺失的数字。

代码 17.4: 消失的数字

```
1       fn missing_number(nums: Vec<i32>) -> i32 {
2           let n = nums.len();
3           let mut res = n as i32;
4
5           for i in 0..n {
6               res ^= i as i32 ^ nums[i];
7           }
8
9           res
10      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

17.4 消失的两个数字

No 17.19 难度 Hard

给定一个数组，包含从 1 到 N 所有的整数，但其中缺了两个数字。你能在 $O(N)$ 时间内只用 $O(1)$ 的空间找到它们吗？

以任意顺序返回这两个数字均可。

示例

输入: [2,3], n = 4
输出: [1,4]

解法一：异或

思路：我们还是利用异或的特性，$res = res \oplus x \oplus x$。

我们将 num 中所有数异或到 xor，再将 $[1, 2..n]$ 的所有数也异或到 xor。得到的 xor 是两个缺失的正整数的异或和，并且肯定不为零。

然后我们运用 $lowbit$ 获取最低一位的 1，那么这两个缺失的正整数在这一位上必然一个为 1，一个为 0。我们据此进行分组异或。最终得到两个缺失的正整数 a 和 b。

代码 17.5: 消失的数字

```rust
fn missing_two(nums: Vec<i32>) -> Vec<i32> {
    let n = nums.len() + 2;
    // 补上消失的两个数
    let mut xor = n ^ (n - 1);
    for (i, v) in nums.iter().enumerate() {
        xor ^= v ^ (i as i32 + 1);
    }

    // low bit
    let diff = xor & -xor;

    // 求单个丢失数的算法
    let mut a = 0;
    for v in nums.iter() {
        if (v & diff) != 0 {
            a ^= v;
        }
    }
    for i in 1..=n {
        if (i & diff) != 0 {
            a ^= i;
        }
    }

    let b = xor ^ a;

    vec![a, b]
```

```
28      }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

18

前缀和

前缀和是一个非常有用的算法技巧，主要应用在数组的区间查询问题上。

所谓前缀和，就是将数组中从索引 1 到当前索引 i 的元素之和，存储在另一个数组 $preSum$ 中。

例如原数组 $a = [1, 2, 3, 4, 5]$，前缀和数组 $preSum = [1, 3, 6, 10, 15]$，其中 $preSum[i] = a[1] + a[2] + ... + a[i]$。

这样一来，如果要计算 $a[i]$ 到 $a[j]$ 之间的区间和，只需要用 $preSum[j] - preSum[i-1]$ 就可以在 $O(1)$ 时间内计算出来。

前缀和主要有以下几个关键特点：

1. 可以高效计算区间和，时间复杂度 O(1)
2. 需要 O(n) 空间存储前缀和数组
3. 可以应用在多维数组上，扩展为前缀和矩阵等
4. 可以配合二分查找、差分等算法使用

18.1 找到最高海拔

No. 1732 难度 Easy

有一个自行车手打算进行一场公路骑行，这条路线总共由 $n + 1$ 个不同海拔的点组成。自行车手从海拔为 0 的点 0 开始骑行。

给你一个长度为 n 的整数数组 $gain$ ，其中 $gain[i]$ 是点 i 和点 $i + 1$ 的净海拔高度差 $(0 <= i < n)$。请你返回最高点的海拔。

示例

输入: gain = [-5,1,5,0,-7]
输出: 1
解释: 海拔高度依次为 [0,-5,-4,1,1,-6] 。最高海拔为 1 。

解法一：前缀和

思路: 根据题目描述，点 0 的海拔高度为 0，点 i $(i > 0)$ 的海拔高度为：

$$\sum_{k=0}^{i-1} gain[i]$$

因此，我们只需要对数组 $gain$ 进行一次遍历，在遍历到第 i 个元素时，使用前缀和的思想维护前 i 个元素的和，并用和更新答案即可。

代码 18.1: 找到最高海拔

```rust
fn largest_altitude(gain: Vec<i32>) -> i32 {
    let mut total = 0;
    let mut ans = 0;
    for x in gain {
        total += x;
        ans = ans.max(total);
    }
    ans
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

18.2 边界上的蚂蚁

No. 3028 难度 Easy

边界上有一只蚂蚁，它有时向左走，有时向右走。

给你一个非零整数数组 $nums$ 。蚂蚁会按顺序读取 $nums$ 中的元素，从第一个元素开始直到结束。每一步，蚂蚁会根据当前元素的值移动：

- 如果 nums[i] < 0 ， 向左移动 -nums[i] 单位。
- 如果 nums[i] > 0 ， 向右移动 nums[i] 单位。

返回蚂蚁返回到边界上的次数。

注意：

- 边界两侧有无限的空间。
- 只有在蚂蚁移动了 |nums[i]| 单位后才检查它是否位于边界上。换句话说，如果蚂蚁只是在移动过程中穿过了边界，则不会计算在内。

示例

输入: nums = [2,3,-5]
输出: 1
解释: 第 1 步后，蚂蚁距边界右侧 2 单位远。
第 2 步后，蚂蚁距边界右侧 5 单位远。
第 3 步后，蚂蚁位于边界上。
所以答案是 1 。

解法一: 前缀和

思路: 这不就是前缀和么。当前缀和等于 0 时，就会到达边界。

一边遍历数组，一边累加元素，如果发现累加值等于 0，说明返回到边界，把答案加一。

代码 18.2: 边界上的蚂蚁

```rust
    fn return_to_boundary_count(nums: Vec<i32>) -> i32 {
        let mut ans = 0;
        let mut total = 0;
        for x in nums {
            total += x;
            if total == 0 {
                ans += 1;
            }
        }
        ans
    }
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(1)$。

19

哈希

哈希表（Hash Table，也叫散列表），是根据键（Key）而直接访问在内存存储位置的数据结构。哈希表通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做哈希函数，存放记录的数组称做哈希表。

19.1 两数之和

No. 1 难度 Easy

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 $target$ 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

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]

解法一: 哈希表

思路: 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。

代码 19.1: 两数之和

```rust
fn two_sum(nums: Vec<i32>, target: i32) -> Vec<i32> {
    let mut ht = std::collections::HashMap::new();
    for (i, x) in nums.iter().enumerate() {
        if let Some(&p) = ht.get(&(target - x)) {
            return vec![p as i32, i as i32];
        }
        ht.insert(x, i);
    }
    vec![]
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

19.2 四数相加 II

No. 454 难度 Medium

给你四个整数数组 $nums1$、$nums2$、$nums3$ 和 $nums4$，数组长度都是 n，请你计算有多少个元组 (i, j, k, l) 能满足：

- $0 <= i, j, k, l < n$
- $nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0$

示例

输入: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
输出: 2
解释:
两个元组如下：
1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0

解法一：哈希表

多数运算结果等于指定值问题都可转为两数相加。超过两数就通过枚举固定多出数。本题转为 （A + B） + （C + D） = 0

思路: 对于 A 和 B，我们使用二重循环对它们进行遍历，得到所有 $A[i] + B[j]$ 的值并存入哈希映射中。对于哈希映射中的每个键值对，每个键表示一种 $A[i] + B[j]$，对应的值为 $A[i] + B[j]$ 出现的次数。

对于 C 和 D，我们同样使用二重循环对它们进行遍历。当遍历到 $C[k]+D[l]$ 时，如果 $-(C[k]+D[l])$ 出现在哈希映射中，那么将 $-(C[k]+D[l])$ 对应的值累加进答案中。

最终即可得到满足 $A[i]+B[j]+C[k]+D[l]=0$ 的四元组数目。

代码 19.2: 四数相加

```rust
fn four_sum_count(a: Vec<i32>, b: Vec<i32>, c: Vec<i32>, d: Vec<i32>) -> i32 {
    let mut count_ab = std::collections::HashMap::new();
    for &v in &a {
        for &w in &b {
            *count_ab.entry(v + w).or_insert(0) += 1;
        }
    }
    let mut ans = 0;
    for &v in &c {
        for &w in &d {
            ans += count_ab.get(&(-v - w)).unwrap_or(&0);
        }
    }
    ans
}
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$。

19.3 两个数组的交集 II

No. 350 简单 Easy

给你两个整数数组 $nums1$ 和 $nums2$，请你以数组形式返回两数组的交集。返回结果中每个元素出现的次数，应与元素在两个数组中都出现的次数一致（如果出现次数不一致，则考虑取较小值）。可以不考虑输出结果的顺序。

示例 1

输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2,2]

示例 2

输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [4,9]

解法一：哈希表

思路：由于同一个数字在两个数组中都可能出现多次，因此需要用哈希表存储每个数字出现的次数。对于一个数字，其在交集中出现的次数等于该数字在两个数组中出现次数的最小值。

首先遍历第一个数组，并在哈希表中记录第一个数组中的每个数字以及对应出现的次数，然后遍历第二个数组，对于第二个数组中的每个数字，如果在哈希表中存在这个数字，则将该数字添加到答案，并减少哈希表中该数字出现的次数。

为了降低空间复杂度，首先遍历较短的数组并在哈希表中记录每个数字以及对应出现的次数，然后遍历较长的数组得到交集。

代码 19.3: 两个数组的交集

```rust
fn intersect(nums1: Vec<i32>, nums2: Vec<i32>) -> Vec<i32> {
    if nums1.len() > nums2.len() {
        return intersect(nums2, nums1);
    }
    // HashMap
    let mut m = std::collections::HashMap::new();
    for num in nums1 {
        *m.entry(num).or_insert(0) += 1;
    }
    // Intersection
    let mut intersection = Vec::new();
    for num in nums2 {
        if let Some(count) = m.get_mut(&num) {
            if *count > 0 {
                intersection.push(num);
                *count -= 1;
            }
        }
    }
    intersection
}
```

复杂度分析

- 时间复杂度：$O(m+n)$。
- 空间复杂度：$O(min(m,n))$。

19.4 重复的 DNA 序列

No 187 题型 Medium DNA 序列由一系列核苷酸组成，缩写为'A', 'C', 'G' 和'T'.。

例如，"ACGAATTCCG" 是一个 DNA 序列。在研究 DNA 时，识别 DNA 中的重复序列非常有用。

给定一个表示 DNA 序列的字符串 s，返回所有在 DNA 分子中出现不止一次的长度为 10 的序列 (子字符串)。你可以按任意顺序返回答案。

示例 1

输入: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
输出: ["AAAAACCCCC","CCCCCAAAAA"]

示例 2

输入: s = "AAAAAAAAAAAAA"
输出: ["AAAAAAAAAA"]

解法一: 哈希表

思路: 我们可以用一个哈希表统计 s 所有长度为 10 的子串的出现次数，返回所有出现次数超过 10 的子串。

代码实现时，可以一边遍历子串一边记录答案，为了不重复记录答案，我们只统计当前出现次数为 2 的子串。

代码 19.4: 重复的 DNA 序列

```rust
const L: usize = 10;

fn find_repeated_dna_sequences(s: &str) -> Vec<String> {
    let mut cnt = std::collections::HashMap::new();
    let mut ans = Vec::new();

    for i in 0..=s.len()-L {
        let sub = &s[i..i+L];
        *cnt.entry(sub.to_string()).or_insert(0) += 1;
        if cnt[&sub.to_string()] == 2 {
            ans.push(sub.to_string());
        }
    }

    ans
}
```

复杂度分析

- 时间复杂度: $O(NL)$。
- 空间复杂度: $O(min(NL))$。

这道题如果使用哈希表 + 滑动窗口 + 位运算，还可以实现 $O(N)$ 的时间复杂度。

20

数学

利用一些数学性质进行巧算。

20.1 x 的平方根

No. 69 难度 Easy 给你一个非负整数 x ，计算并返回 x 的算术平方根。

由于返回类型是整数，结果只保留整数部分，小数部分将被舍去。

注意：不允许使用任何内置指数函数和算符，例如 $pow(x, 0.5)$ 或者 $x * *0.5$ 。

示例 1

输入: x = 4
输出: 2

示例 2

输入: x = 8
输出: 2
解释: 8 的平方根是 2.82842...，由于返回类型是整数，小数部分将被舍去。

解法一: 牛顿迭代

思路: 牛顿迭代法是一种可以用来快速求解函数零点的方法。

为了叙述方便，我们用 C 表示待求出平方根的那个整数。显然，C 的平方根就是函数 $y = f(x) = x^2 - C$ 的零点。

牛顿迭代法的本质是借助泰勒级数，从初始值开始快速向零点逼近。

代码 20.1: x 的平方根

```rust
fn my_sqrt(x: i32) -> i32 {
    if x == 0 {
        return 0;
    }
    let c = x as f64;
    let mut x0 = x as f64;
    loop {
        let xi = 0.5 * (x0 + c / x0);
        if (x0 - xi).abs() < 1e-7 {
            break;
        }
        x0 = xi;
    }
    x0 as i32
}
```

复杂度分析

- 时间复杂度：$O(\log x)$。
- 空间复杂度：$O(1)$。

20.2 2 的幂

No 231 简单 Easy 给你一个整数 n，请你判断该整数是否是 2 的幂次方。如果是，返回 true ；否则，返回 false 。如果存在一个整数 x 使得 $n == 2^x$ ，则认为 n 是 2 的幂次方。

示例 1

输入: n = 1
输出: true

示例 2

输入: n = 16
输出: true

示例 3

输入: n = 5
输出: false

解法一: 位运算

思路: 如果 n 是正整数并且 $n \ \& \ (n \ - \ 1) \ = \ 0$，那么 n 就是 2 的幂。

代码 20.2: 2 的幂

```rust
fn is_power_of_two(n: i32) -> bool {
    n > 0 && n & (n - 1) == 0
```

```
3       }
```

复杂度分析

- 时间复杂度：$O(1)$。
- 空间复杂度：$O(1)$。

类似的题目：3 的幂 No.326、4 的幂 No.342。

很多题目，恕不一一列举了，比较发散，比如取石子游戏、10 瓶水中一瓶毒药小白鼠问题、开关灯问题等等。

20.3 用 Rand7() 实现 Rand10()

No. 470 难度 Medium

给定方法 $rand7$ 可生成 $[1,7]$ 范围内的均匀随机整数，试写一个方 Medium 法 $rand10$ 生成 $[1,10]$ 范围内的均匀随机整数。

你只能调用 $rand7()$ 且不能调用其他方法。请不要使用系统的 $Math.random()$ 方法。

每个测试用例将有一个内部参数 n，即你实现的函数 $rand10()$ 在测试时将被调用的次数。请注意，这不是传递给 $rand10()$ 的参数。

示例 1

输入: n = 1
输出: [2]

示例 2

输入: n = 2
输出: [2, 8]

示例 3

输入: n = 3
输出: [3, 8, 10]

解法一：拒绝采样

思路: 我们可以用拒绝采样的方法实现 $Rand10()$。在拒绝采样中，如果生成的随机数满足要求，那么就返回该随机数，否则会不断生成，直到生成一个满足要求的随机数为止。

- 我们只需要能够满足等概率的生成 10 个不同的数即可，具体的生成方法可以有很多种，比如我们可以利用两个 $Rand7()$ 相乘，我们只取其中等概率的 0 个不同的数的组合即可，当然还有许多其他不同的解法，可以利用各种运算和函数的组合等方式来实现。

- 题目中要求尽可能的减少 $Rand7()$ 的调用次数，则我们应该尽量保证生成的每个不同的数的生成概率尽可能的大，即调用 $Rand7()$ 期望次数尽可能的小。
- 我们可以调用两次 $Rand7()$，那么可以生成 $[1,49]$ 之间的随机整数，我们只用到其中的前 40 个用来实现 $Rand10()$，而拒绝剩下的 9 个数。
- 我们可以看到表中的 $[1,49]$ 每个数生成的概率为 $\frac{1}{49}$。我们实际上只取 $[1,40]$ 这前 40 个数，转化为 $[1,10]$ 时，这 10 个数中每个数的生成概率则为 $\frac{4}{49}$

代码 20.3: rand10

```rust
    fn rand10() -> i32 {
        loop {
            let row = rand7();
            let col = rand7();
            let idx = (row - 1) * 7 + col;
            if idx <= 40 {
                return 1 + (idx - 1) % 10;
            }
        }
    }
    // END: ed8c6549bwf9
```

复杂度分析

- 时间复杂度：$O(1)$。
- 空间复杂度：$O(1)$。

这个可以进一步优化，在第二轮利用丢弃的那 9 个。不过算法就复杂了。理论上平均不到三次 (2.45) 就可以得到一个随机数。