



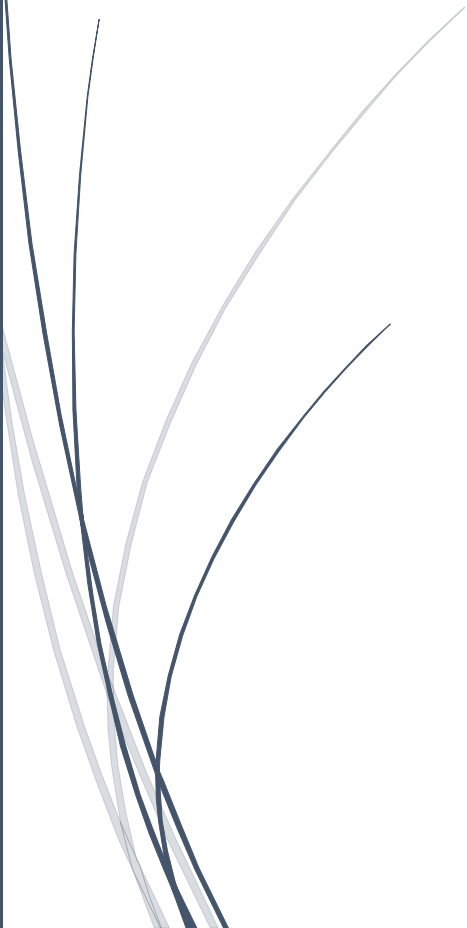
2020-3-30

# 数据结构与算法

LeetCode 刻意练习 V1.0

[yanpengma@163.com](mailto:yanpengma@163.com)

华北电力大学（保定）马燕鹏



# 目录

序言 .....	3
1. 数组 .....	6
1.1 两数之和 .....	6
1.2 删除排序数组中的重复项 .....	10
1.3 移除元素 .....	13
1.4 三数之和 .....	16
1.5 最接近的三数之和 .....	19
1.6 买卖股票的最佳时机 III .....	22
2. 链表 .....	26
2.1 合并两个有序链表 .....	26
2.2 删除排序链表中的重复元素 .....	29
2.3 环形链表 .....	31
2.4 两数相加 .....	36
2.5 删除链表的倒数第 N 个节点 .....	39
2.6 合并 K 个排序链表 .....	43
3. 字符串 .....	48
3.1 罗马数字转整数 .....	48
3.2 最长公共前缀 .....	55
3.3 有效的括号 .....	58
3.4 无重复字符的最长子串 .....	61
3.5 最长回文子串 .....	64

3.6 正则表达式匹配 .....	70
4. 树 .....	75
4.1 相同的树 .....	75
4.2 对称二叉树 .....	78
4.3 二叉树的最大深度 .....	81
4.4 二叉树的中序遍历 .....	85
4.5 不同的二叉搜索树 II .....	88
4.6 恢复二叉搜索树 .....	92
5. 贪心算法 .....	98
5.1 买卖股票的最佳时机 II .....	98
5.2 判断子序列 .....	101
5.3 分发饼干 .....	105
5.4 跳跃游戏 .....	108
5.5 加油站 .....	111
5.6 通配符匹配 .....	115

# 序言

---

[第二期基础算法（Leetcode）刻意练习训练营](#) 已经结束了，本次刻意练习采用分类别练习的模式，即选择了五个知识点（**数组、链表、字符串、树、贪心算法**），每个知识点选择了 **三个简单、两个中等、一个困难** 等级的题目，共计三十道题，利用三十天的时间完成这组刻意练习。

这本小册子就是对这次刻意练习题目的一个总结。该总结即可以作为学习数据结构与算法课程的参考资料，也可以作为备考计算机类研究生的备考资料。希望能够帮助到有这样需求的同学们，也希望大家能够参与到我们的刻意练习训练营中，大家一起学习一起成长。

- [Task01. 两数之和](#)
- [Task02. 删除排序数组中的重复项](#)
- [Task03. 移除元素](#)
- [Task04. 三数之和](#)
- [Task05. 最接近的三数之和](#)
- [Task06. 买卖股票的最佳时机 III](#)
- [Task07. 合并两个有序链表](#)
- [Task08. 删除排序链表中的重复元素](#)
- [Task09. 环形链表](#)
- [Task10. 两数相加](#)
- [Task11. 删除链表的倒数第 N 个节点](#)

- [Task12. 合并 K 个排序链表](#)
- [Task13. 罗马数字转整数](#)
- [Task14. 最长公共前缀](#)
- [Task15. 有效的括号](#)
- [Task16. 无重复字符的最长子串](#)
- [Task17. 最长回文子串](#)
- [Task18. 正则表达式匹配](#)
- [Task19. 相同的树](#)
- [Task20. 对称二叉树](#)
- [Task21. 二叉树的最大深度](#)
- [Task22. 二叉树的中序遍历](#)
- [Task23. 不同的二叉搜索树 II](#)
- [Task24. 恢复二叉搜索树](#)
- [Task25. 买卖股票的最佳时机 II](#)
- [Task26. 判断子序列](#)
- [Task27. 分发饼干](#)
- [Task28. 跳跃游戏](#)
- [Task29. 加油站](#)
- [Task30. 通配符匹配](#)

**我是** 终身学习者 **“老马”**，一个长期践行 **“结伴式学习”** 理念的 **中年大叔**。

我崇尚分享，渴望成长，于 2010 年创立了 **“LSGO 软件技术团队”**，并加入了国内著名的开源组织 **“Datawhale”**，也是 **“Dre@mtech”**、**“智能机器人研究中心”** 和 **“大数据与哲学社会科学实验室”** 的一员。

愿我们一起学习，一起进步，相互陪伴，共同成长。

欢迎关注，请扫描二维码：



# 1. 数组

---

**数组** 是在程序设计中，为了处理方便，把具有相同类型的若干元素按有序的形式组织起来的一种形式。抽象地讲，数组即是有限个类型相同的元素的有序序列。若将此序列命名，那么这个名称即为数组名。组成数组的各个变量称为数组的分量，也称为数组的元素。而用于区分数组的各个元素的数字编号则被称为下标，若为此定义一个变量，即为下标变量。

---

## 1.1 两数之和

---

- 题号：1
- 难度：简单
- <https://leetcode-cn.com/problems/two-sum/>

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

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

**示例 1:**

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

```
因为 nums[0] + nums[1] = 2 + 7 = 9, 所以返回 [0, 1]
```

**示例 2:**

```
给定 nums = [230, 863, 916, 585, 981, 404, 316, 785, 88, 12, 70, 435, 384, 778, 887, 755, 740, 337, 86, 92, 325, 422, 815, 650, 920, 125, 277, 336, 221, 847, 168, 23, 677, 61, 400, 136, 874, 363, 394, 199, 863, 997, 794, 587, 124, 321, 212, 957, 764, 173, 314, 422, 927, 783, 930, 282, 306, 506, 44, 926, 691, 568, 68, 730, 933, 737, 531, 180, 414, 751, 28, 546, 60, 371, 493, 370, 527, 387, 43, 541, 13, 457, 328, 227, 652, 365, 430, 803, 59, 858, 538, 427, 583, 368, 375, 173, 809, 896, 370, 789], target = 542
```

因为  $\text{nums}[28] + \text{nums}[45] = 221 + 321 = 542$ , 所以返回  $[28, 45]$

---

### 第一种：暴力匹配算法

- 执行结果：通过
- 执行用时：432 ms, 在所有 C# 提交中击败了 65.82% 的用户
- 内存消耗：30.8 MB, 在所有 C# 提交中击败了 8.67% 的用户

```
public class Solution
{
    public int[] TwoSum(int[] nums, int target)
    {
        int[] result = new int[2];
        for (int i = 0; i < nums.Length; i++)
        {
            int find = target - nums[i];
            for (int j = i + 1; j < nums.Length; j++)
            {
                if (find == nums[j])
                {
                    result[0] = i;
                    result[1] = j;
                    return result;
                }
            }
        }
        return result;
    }
}
```



## 第二种：通过 Hash 的方式

- 执行结果：通过
- 执行用时：280 ms, 在所有 C# 提交中击败了 96.53% 的用户
- 内存消耗：31.1 MB, 在所有 C# 提交中击败了 6.89% 的用户

```
public class Solution
{
    public int[] TwoSum(int[] nums, int target)
    {
        int[] result = new int[2];
        Dictionary<int, int> dic = new Dictionary<int, int>();
        for (int i = 0; i < nums.Length; i++)
        {
            int find = target - nums[i];
            if (dic.ContainsKey(find))
            {
                result[0] = dic[find];
                result[1] = i;
                break;
            }
            if (dic.ContainsKey(nums[i]) == false)
                dic.Add(nums[i], i);
        }
        return result;
    }
}
```

## Python 语言 (while 循环)

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 94.96% 的用户
- 内存消耗：15.1 MB, 在所有 Python3 提交中击败了 7.35% 的用户

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        i = 0
```

```

result = list()
dic = dict()
while i < len(nums):
    find = target - nums[i]
    j = dic.get(find)
    if j is not None:
        result = [j, i]
    else:
        dic[nums[i]] = i
    i += 1

return result

```

## Python 语言 (for 循环)

- 执行结果: 通过
- 执行用时: 44 ms, 在所有 Python3 提交中击败了 94.96% 的用户
- 内存消耗: 14.8 MB, 在所有 Python3 提交中击败了 24.05% 的用户

```

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        result = list()
        dic = dict()
        for i in range(len(nums)):
            find = target - nums[i]
            if find in dic is not None:
                result = [dic[find], i]
            else:
                dic[nums[i]] = i

        return result

```

## Python 语言

- 执行结果: 通过
- 执行用时: 52 ms, 在所有 Python3 提交中击败了 86.77% 的用户
- 内存消耗: 15.1 MB, 在所有 Python3 提交中击败了 7.35% 的用户

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        result = list()
        dic = dict()
        for index, val in enumerate(nums):
            find = target - val
            if find in dic is not None:
                result = [dic[find], index]
            else:
                dic[val] = index

        return result
```

---

## 1.2 删除排序数组中的重复项

---

- 题号：26
- 难度：简单
- <https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/>

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

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

### 示例 1:

给定数组 `nums = [1,1,2]`,

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

你不需要考虑数组中超出新长度后面的元素。

## 示例 2:

给定 `nums = [0,0,1,1,1,2,2,3,3,4]`,

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

你不需要考虑数组中超出新长度后面的元素。

## 说明:

为什么返回数值是整数, 但输出的答案是数组呢?

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

你可以想象内部操作如下:

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

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

---

## 第一种: 双指针法

**思路** 就是一个快指针一个慢指针,  $j$  快  $i$  慢, 当 `nums[j] == nums[i]` 时,  $j++$  就可以跳过重复项, 不相等时, 让  $i++$  并让 `nums[i] = nums[j]`, 把值复制过来继续执行到末尾即可, 时间复杂度为  $O(n)$ 。

- 执行结果: 通过

- 执行用时: 300 ms, 在所有 C# 提交中击败了 64.43% 的用户
- 内存消耗: 33.5 MB, 在所有 C# 提交中击败了 5.48% 的用户

```
public class Solution
{
    public int RemoveDuplicates(int[] nums)
    {
        if (nums.Length < 2)
            return nums.Length;

        int i = 0;
        for (int j = 1; j < nums.Length; j++)
        {
            if (nums[j] != nums[i])
            {
                i++;
                nums[i] = nums[j];
            }
        }
        return i + 1;
    }
}
```

## Python 语言

- 执行结果: 通过
- 执行用时: 56 ms, 在所有 Python3 提交中击败了 95.28% 的用户
- 内存消耗: 14.4 MB, 在所有 Python3 提交中击败了 57.13% 的用户

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if len(nums) < 2:
            return 1
        i, j = 0, 1
        while j < len(nums):
            if nums[i] != nums[j]:
                i += 1
                nums[i] = nums[j]
            else:
```

```
j += 1
return i + 1
```

---

## 1.3 移除元素

---

- 题号：27
- 难度：简单
- <https://leetcode-cn.com/problems/remove-element/>

给定一个数组 `nums` 和一个值 `val`，你需要**原地**移除所有数值等于 `val` 的元素，返回移除后数组的新长度。

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

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

### 示例 1:

给定 `nums = [3,2,2,3]`, `val = 3`,

函数应该返回新的长度 2, 并且 `nums` 中的前两个元素均为 2。

你不需要考虑数组中超出新长度后面的元素。

### 示例 2:

给定 `nums = [0,1,2,2,3,0,4,2]`, `val = 2`,

函数应该返回新的长度 5, 并且 `nums` 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。

### 示例 3:

输入: [] value = 0

输出: 0

### 示例 4:

输入: [1] value = 1

输出: 0

### 示例 5:

输入: [4,5] value = 5

输出: 1

### 说明:

为什么返回数值是整数，但输出的答案是数组呢？

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

你可以想象内部操作如下：

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

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

## 第一种：双指针法

**思路** 利用双指针  $i$  和  $j$ ,  $i$  为慢指针拖后,  $j$  为快指针向前冲。如果  $nums[j] \neq val$ , 将  $num[j]$  的值赋给  $num[i]$ , 循环结束后,  $i$  指针前面的元素, 即为需要保留的元素, 从而达到了移除元素的目的, 时间复杂度为  $O(n)$ 。

- 执行结果：通过
- 执行用时：272 ms, 在所有 C# 提交中击败了 94.41% 的用户
- 内存消耗：29.9 MB, 在所有 C# 提交中击败了 5.21% 的用户

```
public class Solution
{
    public int RemoveElement(int[] nums, int val)
    {
        int i = 0;
        for (int j = 0; j < nums.Length; j++)
        {
            if (nums[j] != val)
            {
                nums[i] = nums[j];
                i++;
            }
        }
        return i;
    }
}
```

## Python 语言

**思路** 利用 Python 语言中列表本身的特性。

- 执行结果：通过
- 执行用时：28 ms, 在所有 Python3 提交中击败了 96.72% 的用户
- 内存消耗：13.4 MB, 在所有 Python3 提交中击败了 27.52% 的用户



```
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        for i in range(len(nums) - 1, -1, -1):
            if nums[i] == val:
                nums.pop(i)
        return len(nums)
```

---

## 1.4 三数之和

---

- 题号: 15
- 难度: 中等
- <https://leetcode-cn.com/problems/3sum/>

给定一个包含  $n$  个整数的数组  $nums$ ，判断  $nums$  中是否存在三个元素  $a, b, c$ ，使得  $a + b + c = 0$ ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

给定数组  $nums = [-1, 0, 1, 2, -1, -4]$ ，

满足要求的三元组集合为：

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

---

### 第一种：三指针法

**思路：**为了避免三次循环，提升执行效率。首先，对 `nums` 进行排序。然后，固定 3 个指针 `i, l(left), r(right)`，`i` 进行最外层循环，`l` 指向 `nums[i]` 之后数组的最小值，`r` 指向 `nums[i]` 之后数组的最大值。模仿快速排序的思路，如果 `nums[i] > 0` 就不需要继续计算了，否则计算 `nums[i] + nums[l] + nums[r]` 是否等于零并进行相应的处理。如果大于零，向 `l` 方向移动 `r` 指针，如果小于零，向 `r` 方向移动 `l` 指针，如果等于零，则加入到存储最后结果的 `result` 链表中。当然，题目中要求这个三元组不可重复，所以在进行的过程中加入去重就好。

- 执行结果：通过
- 执行用时：348 ms, 在所有 C# 提交中击败了 99.54% 的用户
- 内存消耗：35.8 MB, 在所有 C# 提交中击败了 6.63% 的用户

```
public class Solution
{
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();

        nums = nums.OrderBy(a => a).ToArray();
        int len = nums.Length;

        for (int i = 0; i < len - 2; i++)
        {
            if (nums[i] > 0)
                break; // 如果最小的数字大于0，后面的操作已经没有意义

            if (i > 0 && nums[i - 1] == nums[i])
                continue; // 跳过三元组中第一个元素的重复数据

            int l = i + 1;
            int r = len - 1;

            while (l < r)
            {
```

```

        int sum = nums[i] + nums[l] + nums[r];
        if (sum < 0)
        {
            l++;
        }
        else if (sum > 0)
        {
            r--;
        }
        else
        {
            result.Add(new List<int>() {nums[i], nums[l], nums[r]});
            // 跳过三元组中第二个元素的重复数据
            while (l < r && nums[l] == nums[l + 1])
            {
                l++;
            }
            // 跳过三元组中第三个元素的重复数据
            while (l < r && nums[r - 1] == nums[r])
            {
                r--;
            }
            l++;
            r--;
        }
    }
}
return result;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：660 ms, 在所有 Python3 提交中击败了 95.64% 的用户
- 内存消耗：16.1 MB, 在所有 Python3 提交中击败了 75.29% 的用户

```

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums = sorted(nums)

```

```

result = []

for i in range(0, len(nums) - 2):
    # 如果最小的数字大于0, 后面的操作已经没有意义
    if nums[i] > 0:
        break
    # 跳过三元组中第一个元素的重复数据
    if i > 0 and nums[i-1] == nums[i]:
        continue

    # 限制 nums[i] 是三元组中最小的元素
    l = i + 1
    r = len(nums) - 1
    while l < r:
        sum = nums[i] + nums[l] + nums[r]
        if sum < 0:
            l += 1
        elif sum > 0:
            r -= 1
        else:
            result.append([nums[i], nums[l], nums[r]])
            # 跳过三元组中第二个元素的重复数据
            while l < r and nums[l] == nums[l+1]:
                l += 1
            # 跳过三元组中第三个元素的重复数据
            while l < r and nums[r] == nums[r-1]:
                r -= 1
            l += 1
            r -= 1

return result

```

---

## 1.5 最接近的三数之和

---

- 题号: 16
- 难度: 中等
- <https://leetcode-cn.com/problems/3sum-closest/>

给定一个包括  $n$  个整数的数组 `nums` 和一个目标值 `target`。找出 `nums` 中的三个整数，使得它们的和与 `target` 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

**示例：**

例如，给定数组 `nums = [-1, 2, 1, -4]`，和 `target = 1`。

与 `target` 最接近的三个数的和为 `2`。 ( $-1 + 2 + 1 = 2$ )。

---

### 第一种：利用暴力算法

- 状态：通过
- 125 / 125 个通过测试用例
- 执行用时: 680 ms, 在所有 C# 提交中击败了 12.07% 的用户
- 内存消耗: 23.7 MB, 在所有 C# 提交中击败了 7.41% 的用户

```
public class Solution
{
    public int ThreeSumClosest(int[] nums, int target)
    {
        double error = int.MaxValue;
        int sum = 0;
        for (int i = 0; i < nums.Length - 2; i++)
            for (int j = i + 1; j < nums.Length - 1; j++)
                for (int k = j + 1; k < nums.Length; k++)
                {
                    if (Math.Abs(nums[i] + nums[j] + nums[k] - target) < error)
                    {
                        sum = nums[i] + nums[j] + nums[k];
                        error = Math.Abs(sum - target);
                    }
                }
        return sum;
    }
}
```

## 第二种：利用双指针算法

- 状态：通过
- 125 / 125 个通过测试用例
- 执行用时: 132 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 24 MB, 在所有 C# 提交中击败了 5.55% 的用户

```
public class Solution
{
    public int ThreeSumClosest(int[] nums, int target)
    {
        nums = nums.OrderBy(a => a).ToArray();
        int result = nums[0] + nums[1] + nums[2];
        for (int i = 0; i < nums.Length - 2; i++)
        {
            int start = i + 1, end = nums.Length - 1;
            while (start < end)
            {
                int sum = nums[start] + nums[end] + nums[i];
                if (Math.Abs(target - sum) < Math.Abs(target - result))
                    result = sum;
                if (sum > target)
                    end--;
                else if (sum < target)
                    start++;
                else
                    return result;
            }
        }
        return result;
    }
}
```

## Python 语言

- 执行结果：通过
- 执行用时: 124 ms, 在所有 Python3 提交中击败了 72.19% 的用户

- 内存消耗: 13.2 MB, 在所有 Python3 提交中击败了 22.06% 的用户

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums = sorted(nums)
        result = nums[0] + nums[1] + nums[2]
        for i in range(0, len(nums) - 2):
            start = i + 1
            end = len(nums) - 1
            while start < end:
                sum = nums[start] + nums[end] + nums[i]
                if abs(target - sum) < abs(target - result):
                    result = sum
                if sum > target:
                    end -= 1
                elif sum < target:
                    start += 1
                else:
                    return result
        return result
```

---

## 1.6 买卖股票的最佳时机 III

- 题号: 123
- 难度: 困难
- <https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iii/>

给定一个数组，它的第  $i$  个元素是一支给定的股票在第  $i$  天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **两笔** 交易。

**注意:** 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

### 示例 1:

输入: [3,3,5,0,0,3,1,4]

输出: 6

解释:

在第 4 天 (股票价格 = 0) 的时候买入, 在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 =  $3 - 0 = 3$ 。

随后, 在第 7 天 (股票价格 = 1) 的时候买入, 在第 8 天 (股票价格 = 4) 的时候卖出, 这笔交易所能获得利润 =  $4 - 1 = 3$ 。

### 示例 2:

输入: [1,2,3,4,5]

输出: 4

解释:

在第 1 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 =  $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票, 之后再将它们卖出。

因为这样属于同时参与了多笔交易, 你必须在再次购买前出售掉之前的股票。

### 示例 3:

输入: [7,6,4,3,1]

输出: 0

解释: 在这个情况下, 没有交易完成, 所以最大利润为 0。

---

### 思路:

- $dp1[i] = \max(dp[i-1], prices[i] - \text{minval})$  从前往后遍历, 表示第 1 天到第  $i$  天之间的最大利润 (通过是否在第  $i$  天卖出确认);
- $dp2[i] = \max(dp[i+1], \text{maxval} - prices[i])$  从后往前遍历, 表示第  $i$  天到最后一天之间的最大利润 (通过是否在第  $i$  天买进确认);
- $res = \max(dp1 + dp2)$ ,  $(dp1 + dp2)[i]$  正好表示从第 1 天到最后一天经过两次交易的最大利润, 我们的目标是找到令总利润最大的  $i$ 。



### 例子:

输入: [1,2,4,2,5,7,2,4,9,0]

预期结果: 13

- prices: 1, 2, 4, 2, 5, 7, 2, 4, 9, 0
- leftMaxProfit: 0, 1, 3, 3, 4, 6, 6, 6, 8, 8
- rightMaxProfit: 8, 7, 7, 7, 7, 7, 7, 5, 0, 0

当  $i=5$  时, 前面的最大利润为 6, 后面的最大利润为 7, 总的最大利润为 13。

- 执行结果: 通过
- 执行用时: 108 ms, 在所有 C# 提交中击败了 89.74% 的用户
- 内存消耗: 25.5 MB, 在所有 C# 提交中击败了 20.00% 的用户

```
public class Solution
{
    public int MaxProfit(int[] prices)
    {
        //可以 leftMaxProfit[i] +rightMaxProfit[i] 之和最大值
        if (prices.Length == 0)
        {
            return 0;
        }
        int[] leftMaxProfit = new int[prices.Length]; //前 i 天中最大的
        int minPrice = prices[0]; //买入最低价格
        for (int i = 1; i < prices.Length; i++)
        {
            if (minPrice > prices[i])
            {
                minPrice = prices[i];
            }
            leftMaxProfit[i] = Math.Max(prices[i] - minPrice, leftMaxProfit[i - 1]);
        }
    }
}
```

```

int[] rightMaxProfit = new int[prices.Length]; //后i 天中最大的
int maxPrice = prices[prices.Length - 1]; //卖出最高价格
for (int i = prices.Length - 2; i >= 0; i--)
{
    if (maxPrice < prices[i])
    {
        maxPrice = prices[i];
    }
    rightMaxProfit[i] = Math.Max(maxPrice - prices[i], rightMaxProfit[i +
1]);
}

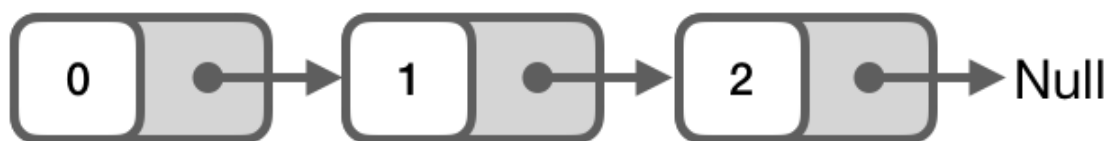
//前i 天最大值和i 天后最大值之和最大值
int maxProfit = 0;
for (int i = 0; i < prices.Length; i++)
{
    maxProfit = Math.Max(maxProfit, leftMaxProfit[i] +
rightMaxProfit[i]);
}
return maxProfit;
}
}

```

## 2. 链表

---

**链表 (Linked List)** 是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里除了存放本身数据 (data fields) 之外还存放其后继节点的指针 (Pointer) 。



使用链表结构可以克服数组需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表有很多种不同的类型：单向链表，双向链表以及循环链表。

---

### 2.1 合并两个有序链表

---

- 题号：21
- 难度：简单
- <https://leetcode-cn.com/problems/merge-two-sorted-lists/>

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

## 示例:

输入: 1->2->4, 1->3->4

输出: 1->1->2->3->4->4

---

## C# 语言

- 执行结果: 通过
- 执行用时: 108 ms, 在所有 C# 提交中击败了 83.80% 的用户
- 内存消耗: 25.9 MB, 在所有 C# 提交中击败了 5.85% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(int.MaxValue);
        ListNode temp = pHead;

        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                temp.next = l1;
                l1 = l1.next;
            }
            else
            {
                temp.next = l2;
                l2 = l2.next;
            }
        }
    }
}
```

```

    }
    temp = temp.next;
}

if (l1 != null)
    temp.next = l1;

if (l2 != null)
    temp.next = l2;

return pHead.next;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 68.12% 的用户
- 内存消耗：13.4 MB, 在所有 Python3 提交中击败了 17.11% 的用户

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        pHead = ListNode(None)
        temp = pHead
        while(l1 and l2):
            if (l1.val <= l2.val):
                temp.next = l1
                l1 = l1.next
            else :
                temp.next = l2
                l2 = l2.next
            temp = temp.next

        if l1 is not None:
            temp.next = l1

```

```
else :  
    temp.next = 12  
  
return pHead.next
```

---

## 2.2 删除排序链表中的重复元素

---

- 题号：83
- 难度：简单
- <https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/>

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

### 示例 1:

输入：1->1->2  
输出：1->2

### 示例 2:

输入：1->1->2->3->3  
输出：1->2->3

---

### C# 语言

- 执行结果：通过
- 执行用时：160 ms, 在所有 C# 提交中击败了 5.23% 的用户
- 内存消耗：25.9 MB, 在所有 C# 提交中击败了 5.72% 的用户

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode DeleteDuplicates(ListNode head)
    {
        if (head == null)
            return head;

        ListNode first = head.next;
        ListNode second = head;
        while (first != null)
        {
            if (first.val == second.val)
                second.next = first.next;
            else
                second = second.next;
            first = first.next;
        }
        return head;
    }
}

```

## Python 语言

- 执行结果：通过
- 执行用时：52 ms, 在所有 Python3 提交中击败了 33.88% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 12.75% 的用户

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

```

```
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if head is None:
            return head

        first = head.next
        second = head
        while first is not None:
            if first.val == second.val:
                second.next = first.next
            else:
                second = second.next
            first = first.next
        return head
```

---

## 2.3 环形链表

---

- 题号：141
- 难度：简单
- <https://leetcode-cn.com/problems/linked-list-cycle/>

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。

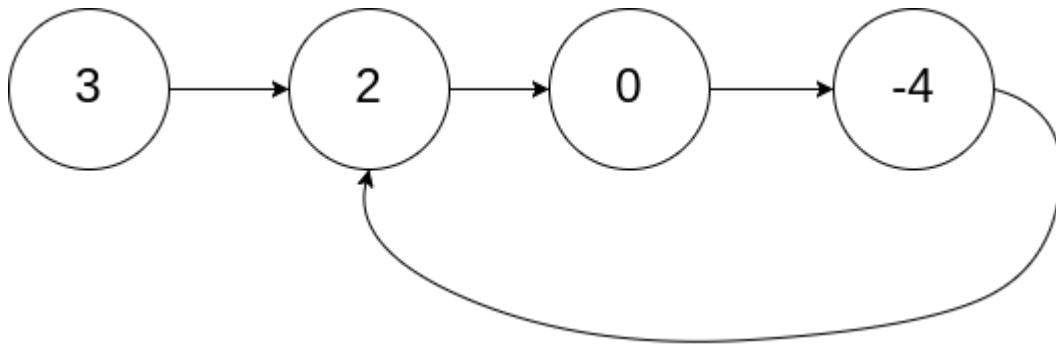
**示例 1：**

输入：head = [3,2,0,-4], pos = 1

输出：true

解释：链表有一个环，其尾部连接到第二个节点。



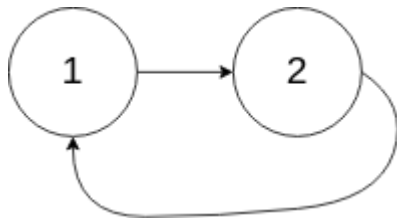


### 示例 2:

输入: head = [1,2], pos = 0

输出: true

解释: 链表中有环, 其尾部连接到第一个节点。

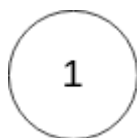


### 示例 3:

输入: head = [1], pos = -1

输出: false

解释: 链表中没有环。



### 进阶:

你能用  $O(1)$  (即, 常量) 内存解决此问题吗?

---

### 第一种: 利用 Hash 的方式

通过检查一个结点此前是否被访问过来判断链表是否为环形链表。

- 状态：通过
- 执行用时：172 ms, 在所有 C# 提交中击败了 8.84% 的用户
- 内存消耗：26.9 MB, 在所有 C# 提交中击败了 5.17% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */

public class Solution
{
    public bool HasCycle(ListNode head)
    {
        HashSet<ListNode> hashSet = new HashSet<ListNode>();
        hashSet.Add(head);

        while (head != null)
        {
            head = head.next;
            if (head == null)
                return false;
            if (hashSet.Contains(head))
                return true;
            hashSet.Add(head);
        }
        return false;
    }
}
```

**Python 语言**

- 执行结果: 通过
- 执行用时: 88 ms, 在所有 Python3 提交中击败了 16.80% 的用户
- 内存消耗: 16.9 MB, 在所有 Python3 提交中击败了 7.04% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        hashset = set()
        hashset.add(head)
        while head is not None:
            head = head.next
            if head is None:
                return False
            if head in hashset:
                return True
            hashset.add(head)
        return False
```

## 第二种: 利用双指针的方式

- 状态: 通过
- 执行用时: 112 ms, 在所有 C# 提交中击败了 98.43% 的用户
- 内存消耗: 24.9 MB, 在所有 C# 提交中击败了 5.13% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
```

```

* }
*/
public class Solution {
    public bool HasCycle(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;

        while (fast != null && fast.next != null)
        {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow)
                return true;
        }
        return false;
    }
}

```

## Python 语言

- 执行结果：通过
- 执行用时：56 ms, 在所有 Python3 提交中击败了 60.97% 的用户
- 内存消耗：16.6 MB, 在所有 Python3 提交中击败了 11.81% 的用户

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        fast = head
        slow = head
        while fast is not None and fast.next is not None:
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                return True
        return False

```

---

## 2.4 两数相加

---

- 题号：2
- 难度：中等
- <https://leetcode-cn.com/problems/add-two-numbers/>

给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

### 示例 1：

输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)

输出：7 -> 0 -> 8

原因：342 + 465 = 807

### 示例 2：

输入：(3 -> 7) + (9 -> 2)

输出：2 -> 0 -> 1

原因：73 + 29 = 102

---

### C# 语言

**思路：**模仿我们小学时代学的加法运算。个位相加超过十进一，十位相加有进位则加上进位，依次类推。

- 状态：通过
- 执行用时: 144 ms, 在所有 C# 提交中击败了 97.98% 的用户
- 内存消耗: 26.7 MB, 在所有 C# 提交中击败了 5.07% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode AddTwoNumbers(ListNode l1, ListNode l2)
    {
        ListNode result = new ListNode(-1);
        ListNode l3 = result;
        int flag = 0;
        while (l1 != null && l2 != null)
        {
            int a = l1.val;
            int b = l2.val;
            int c = a + b + flag;
            l3.next = new ListNode(c%10);

            flag = c >= 10 ? 1 : 0;
            l1 = l1.next;
            l2 = l2.next;
            l3 = l3.next;
        }

        while (l1 != null)
        {
            int a = l1.val + flag;
```

```

        l3.next = new ListNode(a%10);

        flag = a >= 10 ? 1 : 0;
        l1 = l1.next;
        l3 = l3.next;
    }

    while (l2 != null)
    {
        int b = l2.val + flag;

        l3.next = new ListNode(b%10);
        flag = b >= 10 ? 1 : 0;
        l2 = l2.next;
        l3 = l3.next;
    }

    if (flag == 1)
    {
        l3.next = new ListNode(flag);
    }
    return result.next;
}
}

```

## Python 实现

- 执行结果: 通过
- 执行用时 :108 ms, 在所有 Python 提交中击败了 16.54% 的用户
- 内存消耗 :13.6 MB, 在所有 Python 提交中击败了 8.73 %的用户

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

```

**class Solution:**

```

    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        result = ListNode(-1)
        l3 = result

```

```
flag = 0
while l1 is not None and l2 is not None:
    a = l1.val
    b = l2.val
    c = a + b + flag
    l3.next = ListNode(c % 10)
    flag = 1 if c >= 10 else 0
    l1 = l1.next
    l2 = l2.next
    l3 = l3.next

while l1 is not None:
    a = l1.val + flag
    l3.next = ListNode(a % 10)
    flag = 1 if a >= 10 else 0
    l1 = l1.next
    l3 = l3.next

while l2 is not None:
    b = l2.val + flag
    l3.next = ListNode(b % 10)
    flag = 1 if b >= 10 else 0
    l2 = l2.next
    l3 = l3.next

if flag == 1:
    l3.next = ListNode(flag)

return result.next
```

---

## 2.5 删除链表的倒数第 N 个节点

---

- 题号: 19
- 难度: 中等
- <https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/>



给定一个链表，删除链表的倒数第  $n$  个节点，并且返回链表的头结点。

**示例：**

给定一个链表：1->2->3->4->5，和  $n = 2$ 。

当删除了倒数第二个节点后，链表变为 1->2->3->5。

**说明：**

给定的  $n$  保证是有效的。

**进阶：**

你能尝试使用一趟扫描实现吗？

---

### 第一种：先求链表长度的方法

**思路：**先求出链表的长度  $len$ ，再求出要删除结点的位置  $index = len - n$ 。这样就可以从头结点开始遍历到该位置，删除该结点即可。

- 执行结果：通过
- 执行用时：148 ms, 在所有 C# 提交中击败了 6.77% 的用户
- 内存消耗：24.6 MB, 在所有 C# 提交中击败了 5.43% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
```

```

public class Solution
{
    public ListNode RemoveNthFormEnd(ListNode head, int n)
    {
        int len = GetLength(head);
        int index = len - n;

        if (index == 0)
        {
            head = head.next;
            return head;
        }
        ListNode temp = head;
        for (int i = 0; i < index - 1; i++)
        {
            temp = temp.next;
        }
        temp.next = temp.next.next;
        return head;
    }

    public int GetLength(ListNode head)
    {
        ListNode temp = head;
        int i = 0;
        while (temp != null)
        {
            i++;
            temp = temp.next;
        }
        return i;
    }
}

```

## 第二种：双指针法

**思路：**使用两个指针，前面的指针 p2 先走 n 步，接着让后面的指针 p1 与 p2 同步走，p2 走到终点，p1 即走到要移除的结点位置。

- 执行结果：通过
- 执行用时：108 ms, 在所有 C# 提交中击败了 74.84% 的用户

- 内存消耗: 24.8 MB, 在所有 C# 提交中击败了 5.43% 的用户

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode RemoveNthFromEnd(ListNode head, int n) {
        ListNode p1 = head;
        ListNode p2 = head;

        while (n > 0)
        {
            p2 = p2.next;
            n--;
        }

        if (p2 == null) // 移除头结点
        {
            return head.next;
        }

        while (p2.next != null)
        {
            p2 = p2.next;
            p1 = p1.next;
        }

        p1.next = p1.next.next;
        return head;
    }
}

```

## Python 语言

- 执行结果: 通过
- 执行用时: 48 ms, 在所有 Python3 提交中击败了 23.58% 的用户

- 内存消耗: 13.5 MB, 在所有 Python3 提交中击败了 7.83% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        p1 = head
        p2 = head
        while(n>0):
            p2 = p2.next
            n-=1

        if(p2 is None): #移除头结点
            return head.next

        while(p2.next):
            p1 = p1.next
            p2 = p2.next

        p1.next = p1.next.next
        return head
```

---

## 2.6 合并 K 个排序链表

---

- 题号: 23
- 难度: 困难
- <https://leetcode-cn.com/problems/merge-k-sorted-lists/>

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

## 示例:

输入:

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

输出: 1->1->2->3->4->4->5->6

---

## 第一种: 两两合并的方式

构造合并两个有序链表得到一个新的有序链表的方法: `ListNode MergeTwoLists(ListNode l1, ListNode l2)`。可以使用该方法合并前两个有序链表得到一个新的有序链表, 之后把这个新链表与第三个有序链表合并, 依次类推, 最后得到合并  $k$  个有序列表的新列表。

- 执行结果: 通过
- 执行用时: 256 ms, 在所有 C# 提交中击败了 36.69% 的用户
- 内存消耗: 29.3 MB, 在所有 C# 提交中击败了 18.37% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(int.MaxValue);
        ListNode temp = pHead;
```

```

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

if (l1 != null)
    temp.next = l1;

if (l2 != null)
    temp.next = l2;

return pHead.next;
}

public ListNode MergeKLists(ListNode[] lists) {
    if (lists.Length == 0)
        return null;

    ListNode result = lists[0];
    for (int i = 1; i < lists.Length; i++)
    {
        result = MergeTwoLists(result, lists[i]);
    }
    return result;
}
}

```

## 第二种：选择值最小结点的方式

- 执行结果：通过
- 执行用时：776 ms, 在所有 C# 提交中击败了 7.10% 的用户

- 内存消耗: 29.2 MB, 在所有 C# 提交中击败了 22.45% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode MergeKLists(ListNode[] lists)
    {
        int len = lists.Length;
        if (len == 0)
            return null;

        ListNode pHead = new ListNode(-1);
        ListNode temp = pHead;
        while (true)
        {
            int min = int.MaxValue;
            int minIndex = -1;
            for (int i = 0; i < len; i++)
            {
                if (lists[i] != null)
                {
                    if (lists[i].val < min)
                    {
                        minIndex = i;
                        min = lists[i].val;
                    }
                }
            }
            if (minIndex == -1)
            {
                break;
            }
            temp.next = lists[minIndex];
            temp = temp.next;
            lists[minIndex] = lists[minIndex].next;
        }
    }
}
```

```

        return pHead.next;
    }
}

```

## Python 语言

- 执行结果：通过
- 执行用时：7772 ms, 在所有 Python3 提交中击败了 5.02% 的用户
- 内存消耗：16.5 MB, 在所有 Python3 提交中击败了 44.31% 的用户

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        import sys
        count = len(lists)
        if count == 0:
            return None
        pHead = ListNode(-1)
        temp = pHead
        while True:
            minValue = sys.maxsize
            minIndex = -1
            for i in range(count):
                if lists[i] is not None:
                    if lists[i].val < minValue :
                        minIndex = i
                        minValue = lists[i].val
            if minIndex == -1:
                break
            temp.next = lists[minIndex]
            temp = temp.next
            lists[minIndex] = lists[minIndex].next
        return pHead.next

```



## 3. 字符串

**字符串或串 (string)** 是由数字、字母、下划线组成的一串字符。一般记为  $s =$

“ $a_1a_2\dots a_n$ ” ( $n \geq 0$ )。它是编程语言中表示文本的数据类型。

通常以串的整体作为操作对象，如：在串中查找某个子串在该串中首次出现的位置、在串的某个位置上插入一个子串以及删除一个子串等。两个字符串相等的充要条件是：长度相等，并且各个对应位置上的字符都相等。串通常以顺序的方式进行存储与实现。

---

### 3.1 罗马数字转整数

- 题号：13
- 难度：简单
- <https://leetcode-cn.com/problems/roman-to-integer/>

罗马数字包含以下七种字符: I, V, X, L, C, D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 II，即为两个并列的 1。12 写做 XII，即为 X + II。 27 写做 XXVII，即为 XX + V + II。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。  
X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。  
C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

### 示例 1:

输入: "III"  
输出: 3

### 示例 2:

输入: "IV"  
输出: 4

### 示例 3:

输入: "IX"  
输出: 9

### 示例 4:

输入: "LVIII"  
输出: 58  
解释: L = 50, V = 5, III = 3.

### 示例 5:

输入: "MCMXCIV"  
输出: 1994  
解释: M = 1000, CM = 900, XC = 90, IV = 4.

---

### 第一种：直接法

根据“罗马数字中小的数字在大的数字的右边”以及六种特殊情况的规则直接去写代码。

- 执行结果：通过
- 执行用时：96 ms, 在所有 C# 提交中击败了 95.88% 的用户
- 内存消耗：25.7 MB, 在所有 C# 提交中击败了 5.27% 的用户

```
public class Solution
{
    public int RomanToInt(string s)
    {
        int result = 0;
        int count = s.Length;
        int i = 0;
        while (i < count)
        {
            char c = s[i];
            int move = 0;
            switch (c)
            {
                case 'I':
                    result += PreI(i, s, ref move);
                    break;
                case 'V':
                    result += 5;
                    move = 1;
                    break;
                case 'X':
                    result += PreX(i, s, ref move);
                    break;
                case 'L':
                    result += 50;
                    move = 1;
                    break;
                case 'C':
                    result += PreC(i, s, ref move);
                    break;
                case 'D':
                    result += 500;
                    move = 1;
                    break;
            }
            i += move;
        }
        return result;
    }
}
```

```

        case 'M':
            result += 1000;
            move = 1;
            break;
    }
    i += move;
}
return result;
}

private int PreI(int index, string s, ref int move)
{
    //I 1
    //IV 4
    //IX 9
    //II 2
    int result = 0;
    int count = s.Length;
    if (index + 1 < count)
    {
        char c = s[index + 1];
        switch (c)
        {
            case 'V':
                result = 4;
                move = 2;
                break;
            case 'X':
                result = 9;
                move = 2;
                break;
            case 'I':
                result = 2;
                move = 2;
                break;
        }
    }
    else
    {
        result = 1;
        move = 1;
    }
    return result;
}

```

```

private int PreX(int index, string s, ref int move)
{
    //X 10
    //XL 40
    //XC 90
    int result = 0;
    int count = s.Length;
    if (index + 1 < count)
    {
        char c = s[index + 1];
        switch (c)
        {
            case 'L':
                result = 40;
                move = 2;
                break;
            case 'C':
                result = 90;
                move = 2;
                break;
            default:
                result = 10;
                move = 1;
                break;
        }
    }
    else
    {
        result = 10;
        move = 1;
    }
    return result;
}

private int PreC(int index, string s, ref int move)
{
    //C 100
    //CD 400
    //CM 900
    int result = 0;
    int count = s.Length;
    if (index + 1 < count)
    {
        char c = s[index + 1];

```

```

        switch (c)
        {
            case 'D':
                result = 400;
                move = 2;
                break;
            case 'M':
                result = 900;
                move = 2;
                break;
            default:
                result = 100;
                move = 1;
                break;
        }
    }
    else
    {
        result = 100;
        move = 1;
    }
    return result;
}
}

```

## 第二种：利用字典的方法

把罗马字符的所有组合作为 key 和 value 存到字典当中，每次取一个字符，判断这个字符之后是否还有字符。如果有，则判断这两个字符是否在字典中，如果存在则取值。否则，按照一个字符去取值即可。

- 执行结果：通过
- 执行用时：120 ms, 在所有 C# 提交中击败了 42.16% 的用户
- 内存消耗：25.8 MB, 在所有 C# 提交中击败了 5.27% 的用户

```

public class Solution
{
    public int RomanToInt(string s)
    {

```

```

{
    Dictionary<string, int> dic = new Dictionary<string, int>();
    dic.Add("I", 1);
    dic.Add("II", 2);
    dic.Add("IV", 4);
    dic.Add("IX", 9);
    dic.Add("X", 10);
    dic.Add("XL", 40);
    dic.Add("XC", 90);
    dic.Add("C", 100);
    dic.Add("CD", 400);
    dic.Add("CM", 900);
    dic.Add("V", 5);
    dic.Add("L", 50);
    dic.Add("D", 500);
    dic.Add("M", 1000);

    int result = 0;
    int count = s.Length;
    int i = 0;
    while (i < count)
    {
        char c = s[i];
        if (i + 1 < count && dic.ContainsKey(s.Substring(i, 2)))
        {
            result += dic[s.Substring(i, 2)];
            i += 2;
        }
        else
        {
            result += dic[c.ToString()];
            i += 1;
        }
    }
    return result;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：72 ms, 在所有 Python3 提交中击败了 24.93% 的用户

- 内存消耗: 13.5 MB, 在所有 Python3 提交中击败了 5.05% 的用户

```
class Solution:
    def romanToInt(self, s: str) -> int:
        dic = {"I": 1, "II": 2, "IV": 4, "IX": 9, "X": 10, "XL": 40, "XC": 90,
               "C": 100, "CD": 400, "CM": 900, "V": 5,
               "L": 50, "D": 500, "M": 1000}
        result = 0
        count = len(s)
        i = 0
        while i < count:
            c = s[i]
            if i + 1 < count and s[i:i + 2] in dic:
                result += dic[s[i:i + 2]]
                i += 2
            else:
                result += dic[c]
                i += 1
        return result
```

---

## 3.2 最长公共前缀

---

- 题号: 14
- 难度: 简单
- <https://leetcode-cn.com/problems/longest-common-prefix/>

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: ["flower", "flow", "flight"]



输出: "f1"

## 示例 2:

输入: ["dog", "racecar", "car"]

输出: ""

解释: 输入不存在公共前缀。

## 说明:

所有输入只包含小写字母 a-z。

---

## C# 语言

- 状态: 通过
- 118 / 118 个通过测试用例
- 执行用时: 144 ms, 在所有 C# 提交中击败了 94.92% 的用户
- 内存消耗: 23.4 MB, 在所有 C# 提交中击败了 11.69% 的用户

```
public class Solution {
    public string LongestCommonPrefix(string[] strs)
    {
        if (strs.Length == 0)
            return string.Empty;

        string result = strs[0];
        for (int i = 1; i < strs.Length; i++)
        {
            result = Prefix(result, strs[i]);
            if (string.IsNullOrEmpty(result))
                break;
        }
        return result;
    }

    public string Prefix(string str1, string str2)
```

```

{
    int len1 = str1.Length;
    int len2 = str2.Length;
    int len = Math.Min(len1, len2);
    int i = 0;
    for (; i < len; i++)
    {
        if (str1[i] != str2[i])
            break;
    }
    return i == 0 ? string.Empty : str1.Substring(0, i);
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 35.93% 的用户
- 内存消耗：13.6 MB, 在所有 Python3 提交中击败了 5.14% 的用户

```

class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if len(strs) == 0:
            return ""
        result = strs[0]
        for i in range(len(strs)):
            result = self.Prefix(result, strs[i])
            if result == "":
                break
        return result

    def Prefix(self, str1: str, str2: str) -> str:
        len1, len2 = len(str1), len(str2)
        i = 0
        while i < min(len1, len2):
            if str1[i] != str2[i]:
                break
            i += 1
        return "" if i == 0 else str1[0:i]

```

## 3.3 有效的括号

---

- 题号：20
- 难度：简单
- <https://leetcode-cn.com/problems/valid-parentheses/>

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

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

### 示例 1:

输入: "()"

输出: true

### 示例 2:

输入: "()[]{}"

输出: true

### 示例 3:

输入: "(]"

输出: false

### 示例 4:

输入: "([)]"

输出: false

### 示例 5:

输入: "{[]}"

输出: true

### 示例 6:

输入: ""

输出: true

### 示例 7:

输入: "("

输出: false

---

## 方式一：利用栈

**思路：**首先判断该字符串长度的奇偶性，如果是奇数，则返回 false。否则，利用栈先进后出的特点，遇到 "{", "[", "(" 进行入栈操作，遇到 "}", "]", ")" 就与栈顶元素进行比较，如果是对应括号则出栈，否则返回 false。

- 执行结果：通过
- 执行用时：88 ms, 在所有 C# 提交中击败了 70.31% 的用户
- 内存消耗：22 MB, 在所有 C# 提交中击败了 17.91% 的用户

```
public class Solution {  
    public bool IsValid(string s)  
    {  
        if (string.IsNullOrEmpty(s))  
            return true;  
  
        int count = s.Length;  
        if(count%2 == 1)  
            return false;  
    }  
}
```

```

Stack<char> stack = new Stack<char>();
for (int i = 0; i < count; i++)
{
    char c = s[i];
    if (stack.Count == 0)
    {
        stack.Push(c);
    }
    else if(c == '[' || c == '{' || c == '(')
    {
        stack.Push(c);
    }
    else if (stack.Peek() == GetPair(c))
    {
        stack.Pop();
    }
    else
    {
        return false;
    }
}
return stack.Count == 0;
}

public static char GetPair(char c)
{
    if (c == ')')
        return '(';
    if (c == '}')
        return '{';
    if (c == ']')
        return '[';
    return '\0';
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 47.30% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 5.16% 的用户

```
class Solution:
    def isValid(self, s: str) -> bool:
        dic = {"(": ")", "{": "}", "[": "]": "["}
        if s is None:
            return True

        count = len(s)
        if count % 2 == 1:
            return False

        lst = list()
        for i in range(count):
            c = s[i]
            if len(lst) == 0:
                lst.append(c)
            elif c == "[" or c == "{" or c == "(":
                lst.append(c)
            elif lst[-1] == dic[c]:
                lst.pop()
            else:
                return False
        return len(lst) == 0
```

---

## 3.4 无重复字符的最长子串

---

- 题号：3
- 难度：中等
- <https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

**示例 1:**

输入: "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: "bbbbbb"

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: "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。

---

## C# 语言

**思路:** 借助动态规划的思路, 从前到后求出以每个位置为终止位置, 所构成无重复子串的长度, 之后求这些长度的最大值即可。

对于任意位置 `index` 其最长无重复子串的长度为 `result[index] = min{k1,k2}`, `k1 = result[index-1] + 1`, `k2` 为从 `index` 位置往前推直到出现 `index` 位置的字符或 `index=0` 为止的子串长度。

- 执行结果: 通过
- 执行用时: 96 ms, 在所有 C# 提交中击败了 82.81% 的用户
- 内存消耗: 25.2 MB, 在所有 C# 提交中击败了 25.52% 的用户

```
public class Solution
{
    public int LengthOfLongestSubstring(string s)
    {
```

```

    if (string.IsNullOrEmpty(s))
        return 0;
    int[] result = new int[s.Length];
    result[0] = 1;

    for (int i = 1; i < s.Length; i++)
    {
        int count = GetLength(i, s);
        result[i] = result[i-1] < count ? result[i-1]+1 : count;
    }
    return result.Max();
}
private int GetLength(int index, string s)
{
    char c = s[index];
    int result = 1;
    for (int i = index-1; i >= 0; i--)
    {
        if (s[i] != c)
            result += 1;
        else
            break;
    }
    return result;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：724 ms, 在所有 Python3 提交中击败了 9.22% 的用户
- 内存消耗：13.7 MB, 在所有 Python3 提交中击败了 5.01% 的用户

```

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        if len(s) == 0:
            return 0
        result = list()
        result.append(1)
        for i in range(1, len(s)):
            count = self.GetLength(i, s)

```



```
        result.append(result[i - 1] + 1 if result[i - 1] < count else count)
    return max(result)
```

```
def GetLength(self, index: int, s: str):
    c = s[index]
    result = 1
    for i in range(index - 1, -1, -1):
        if s[i] != c:
            result += 1
        else:
            break
    return result
```

---

## 3.5 最长回文子串

---

- 题号：5
- 难度：中等
- <https://leetcode-cn.com/problems/longest-palindromic-substring/>

给定一个字符串  $s$ ，找到  $s$  中最长的回文子串。你可以假设  $s$  的最大长度为 1000。

### 示例 1：

输入："babad"

输出："bab"

注意："aba" 也是一个有效答案。

### 示例 2：

输入："cbbd"

输出："bb"

### 示例 3：

输入："a"

输出: "a"

---

回文是一个正读和反读都相同的字符串，例如，“aba”是回文，而“abc”不是。

**第一种：暴力法，列举所有的子串，判断该子串是否为回文。**

- 执行结果：超出时间限制

```
public class Solution
{
    public string LongestPalindrome(string s)
    {
        if (string.IsNullOrEmpty(s))
            return string.Empty;
        if (s.Length == 1)
            return s;

        int start = 0;
        int end = 0;
        int len = int.MinValue;
        for (int i = 0; i < s.Length; i++)
        {
            for (int j = i + 1; j < s.Length; j++)
            {
                string str = s.Substring(i, j - i + 1);
                if (isPalindrome(str) && str.Length > len)
                {
                    len = str.Length;
                    start = i;
                    end = j;
                }
            }
        }
        return s.Substring(start, end - start + 1);
    }

    public bool isPalindrome(string s)
    {
        for (int i = 0, len = s.Length / 2; i < len; i++)
```

```
{
    if (s[i] != s[s.Length - 1 - i])
        return false;
}
return true;
}
```

## 第二种：动态规划

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。

与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。

我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。

具体的动态规划算法多种多样，但它们具有相同的填表格式。

使用记号  $s[l, r]$  表示原始字符串的一个子串， $l$ 、 $r$  分别是区间的左右边界的索引值，使用左闭、右闭区间表示左右边界可以取到。

$dp[l, r]$  表示子串  $s[l, r]$ （包括区间左右端点）是否构成回文串，是一个二维布尔型数组。

- 当子串只包含 1 个字符，它一定是回文子串；

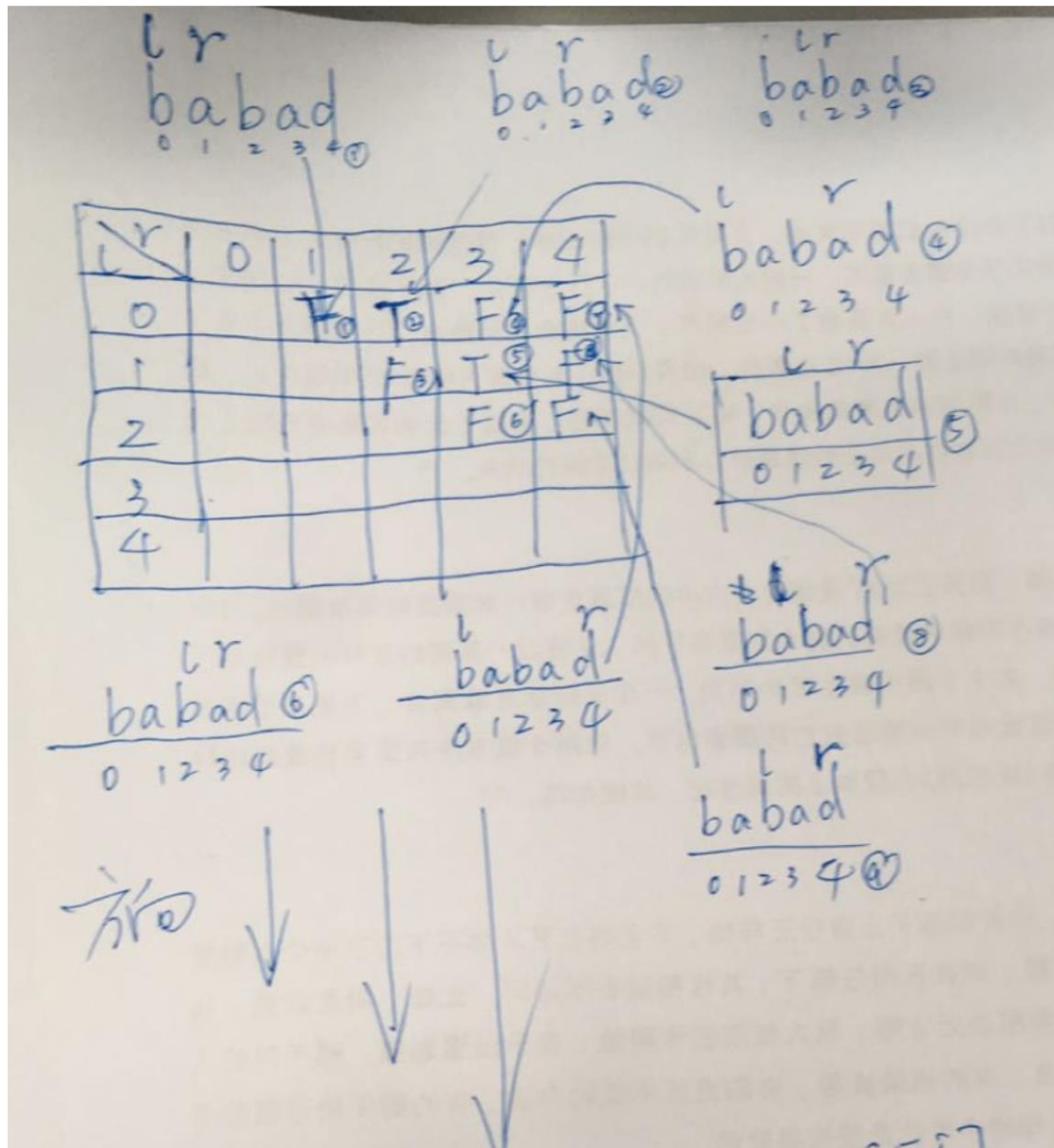
- 当子串包含 2 个以上字符的时候:  $s[l, r]$  是一个回文串, 那么这个回文串两边各往里面收缩一个字符 (如果可以的话) 的子串  $s[l + 1, r - 1]$  也一定是回文串。

故, 当  $s[l] == s[r]$  成立的时候,  $dp[l, r]$  的值由  $dp[l + 1, r - 1]$  决定, 这里还需要再多考虑一点点: “原字符串去掉左右边界” 的子串的边界情况。

- 当原字符串的元素个数为 3 的时候, 如果左右边界相等, 那么去掉它们以后, 只剩下 1 个字符, 它一定是回文串, 故原字符串也一定是回文串;
- 当原字符串的元素个数为 2 的时候, 如果左右边界相等, 那么去掉它们以后, 只剩下 0 个字符, 显然原字符串也一定是回文串。

综上, 如果一个字符串的左右边界相等, 判断为回文只需以下二者之一成立即可:

- 去掉左右边界以后的字符串不构成区间, 即  $s[l + 1, r - 1]$  包含元素少于 2 个, 即:  $r - l <= 2$ 。
- 去掉左右边界以后的字符串是回文串, 即  $dp[l + 1, r - 1] == \text{true}$ 。



- 状态：通过
- 103 / 103 个通过测试用例
- 执行用时: 232 ms, 在所有 C# 提交中击败了 46.79% 的用户
- 内存消耗: 40.9 MB, 在所有 C# 提交中击败了 5.43% 的用户

```
public class Solution {
    public string LongestPalindrome(string s)
    {
        if (string.IsNullOrEmpty(s))
```

```

        return string.Empty;
    int len = s.Length;
    if (len == 1)
        return s;
    int longestPalindromelen = 1;
    string longestPalindromeStr = s.Substring(0, 1);
    bool[,] dp = new bool[len, len];

    for (int r = 1; r < len; r++)
    {
        for (int l = 0; l < r; l++)
        {
            if (s[r] == s[l] && (r - l <= 2 || dp[l + 1, r - 1] == true))
            {
                dp[l, r] = true;
                if (longestPalindromelen < r - l + 1)
                {
                    longestPalindromelen = r - l + 1;
                    longestPalindromeStr = s.Substring(l, r - l + 1);
                }
            }
        }
    }
    return longestPalindromeStr;
}
}

```

## Python 语言

- 执行结果: 通过
- 执行用时: 3392 ms, 在所有 Python3 提交中击败了 43.87% 的用户
- 内存消耗: 21.2 MB, 在所有 Python3 提交中击败了 18.81% 的用户

```

class Solution:
    def longestPalindrome(self, s: str) -> str:
        count = len(s)
        if count == 0 or count == 1:
            return s
        longestPalindromelen = 1
        longestPalindromeStr = s[0:1]
        dp = [[False] * count for i in range(count)]

```

```

for r in range(1, count):
    for l in range(0, r):
        if s[r] == s[l] and (r - l <= 2 or dp[l + 1][r - 1] == True):
            dp[l][r] = True
            if longestPalindromelen < r - l + 1:
                longestPalindromelen = r - l + 1
                longestPalindromeStr = s[l:l + longestPalindromelen]
return longestPalindromeStr

```

## 3.6 正则表达式匹配

- 题号: 10
- 难度: 困难
- <https://leetcode-cn.com/problems/regular-expression-matching/>

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

'.' 匹配任意单个字符

'\*' 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 **整个** 字符串  $s$  的，而不是部分字符串。

**说明:**

- $s$  可能为空，且只包含从  $a$ - $z$  的小写字母。
- $p$  可能为空，且只包含从  $a$ - $z$  的小写字母，以及字符 '.' 和 '\*'。

**示例 1:**

输入:

$s = "aa"$

$p = "a"$

输出: `false`

解释: "a" 无法匹配 "aa" 整个字符串。

## 示例 2:

输入:

```
s = "aa"
```

```
p = "a*"
```

输出: `true`

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

## 示例 3:

输入:

```
s = "ab"
```

```
p = ".*"
```

输出: `true`

解释: ".\*" 表示可匹配零个或多个 ('\*') 任意字符 ('.').

## 示例 4:

输入:

```
s = "aab"
```

```
p = "c*a*b"
```

输出: `true`

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

## 示例 5:

输入:

```
s = "mississippi"
```

```
p = "mis*is*p*."
```

输出: `false`

---

## 第一种: 回溯法

思路: 这种匹配思路其实就是不断地减掉 s 和 p 的可以匹配首部, 直至一个或两个字符串被减为空的时候, 根据最终情况来得出结论。

如果只是两个普通字符串进行匹配, 按序遍历比较即可:



```
if (s[i] == p[i])
```

如果正则表达式字符串  $p$  只有一种"."一种特殊标记，依然是按序遍历比较即可：

```
if (s[i] == p[i] || p[i] == '.')
```

上述两种情况实现时还需要判断字符串长度和字符串判空的操作。

但是，"\*"这个特殊字符需要特殊处理，当  $p$  的第  $i$  个元素的下一个元素是星号时会有两种情况：

- $i$  元素需要出现 0 次，我们就保持  $s$  不变，将  $p$  的减掉两个元素，调用 `IsMatch`。

例如  $s: bc$ 、 $p: a*bc$ ，我们就保持  $s$  不变，减掉  $p$  的" $a*$ "，调用

`IsMatch(s:bc,p:bc)`。

- $i$  元素需要出现一次或更多次，先比较  $i$  元素和  $s$  首元素，相等则保持  $p$  不变， $s$

减掉首元素，调用 `IsMatch`。例如  $s: aabb$ 、 $p: a*bb$ ，就保持  $p$  不变，减掉  $s$  的首元

素，调用 `IsMatch(s:abb,p:a*bb)`。

此时存在一些需要思考的情况，例如  $s: abb$ 、 $p: a*abb$ ，会用两种方式处理：

- 按照上述第二种情况比较  $i$  元素和  $s$  首元素，发现相等就会减掉  $s$  的首字符，调用

`IsMatch(s:bb,p:a*abb)`。在按照上述第一种情况减去  $p$  的两个元素，调用

`IsMatch(s:bb,p:abb)`，最终导致 `false`。

- 直接按照上述第一种情况减去  $p$  的两个元素，调用 `IsMatch(s:abb,p:abb)`，最终导致 `true`。

所以说这算是一种暴力方法，会将所有的情况走一边，看看是否存在可以匹配的情况。

- 执行结果：通过

- 执行用时：768 ms, 在所有 C# 提交中击败了 10.69% 的用户
- 内存消耗：25.6 MB, 在所有 C# 提交中击败了 5.00% 的用户

```
public class Solution
{
    public bool IsMatch(string s, string p)
    {
        //若正则串p为空串, 则s为空串匹配成功, s不为空串匹配失败。
        if (string.IsNullOrEmpty(p))
            return string.IsNullOrEmpty(s) ? true : false;

        //判断s和p的首字符是否匹配, 注意要先判断s不为空
        bool headMatched = string.IsNullOrEmpty(s) == false
            && (s[0] == p[0] || p[0] == '.');

        if (p.Length >= 2 && p[1] == '*')
        {
            //如果p的第一个元素的下一个元素是*, 则分别对两种情况进行判断
            return IsMatch(s, p.Substring(2)) ||
                (headMatched && IsMatch(s.Substring(1), p));
        }
        else if (headMatched)
        {
            //否则, 如果s和p的首字符相等
            return IsMatch(s.Substring(1), p.Substring(1));
        }
        else
        {
            return false;
        }
    }
}
```

## Python 语言

- 执行结果：通过
- 执行用时：1940 ms, 在所有 Python3 提交中击败了 6.96% 的用户
- 内存消耗：13.6 MB, 在所有 Python3 提交中击败了 5.06% 的用户

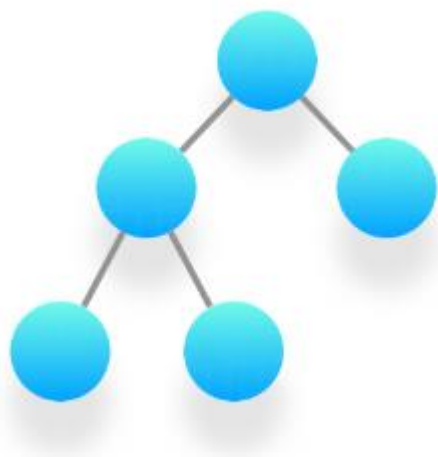
```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        if len(p) == 0:
            return True if len(s) == 0 else False
        headMatched = len(s) != 0 and (s[0] == p[0] or p[0] == '.')

        if len(p) >= 2 and p[1] == '*':
            return self.isMatch(s, p[2:]) or (headMatched and self.isMatch(s[1:], p))
        elif headMatched == True:
            return self.isMatch(s[1:], p[1:])
        else:
            return False
```

## 4. 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  $n(n > 0)$  个有限节点组成的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：

- 每个节点都只有有限个子节点或无子节点；
- 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。

---

### 4.1 相同的树

- 题号: 100
- 难度: 简单
- <https://leetcode-cn.com/problems/same-tree/>

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

### 示例 1:

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

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

输出: `true`

### 示例 2:

```
输入:      1      1
          /      \
         2        2

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

输出: `false`

### 示例 3:

```
输入:      1      1
          / \    / \
         2  1   1  2

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

输出: `false`

## C# 语言

- 执行结果：通过
- 执行用时：160 ms, 在所有 C# 提交中击败了 5.85% 的用户
- 内存消耗：24 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public bool IsSameTree(TreeNode p, TreeNode q)
    {
        //递归终止条件
        if (p == null && q == null)
            return true;

        if (p != null && q != null && p.val == q.val)
        {
            return IsSameTree(p.left, q.left)
                && IsSameTree(p.right, q.right);
        }
        return false;
    }
}
```

## Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 39.08% 的用户

- 内存消耗: 13.4 MB, 在所有 Python3 提交中击败了 5.27% 的用户

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if p is None and q is None:
            return True
        if p is not None and q is not None and p.val == q.val:
            return self.isSameTree(p.left, q.left) and self.isSameTree(p.right,
q.right)
        else:
            return False
```

---

## 4.2 对称二叉树

---

- 题号: 101
- 难度: 简单
- <https://leetcode-cn.com/problems/symmetric-tree/>

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

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

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

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

---

### 第一种：采用递归的方法

- 执行结果：通过
- 执行用时：132 ms, 在所有 C# 提交中击败了 16.67% 的用户
- 内存消耗：25.1 MB, 在所有 C# 提交中击败了 5.17% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */

// 镜像对称的递归函数
public class Solution
{
    public bool IsSymmetric(TreeNode root)
    {
        return IsMirror(root, root);
    }
    private bool IsMirror(TreeNode t1, TreeNode t2)
    {
        if (t1 == null && t2 == null) return true;
        if (t1 == null || t2 == null) return false;
        return (t1.val == t2.val)
            && IsMirror(t1.left, t2.right)
            && IsMirror(t1.right, t2.left);
    }
}
```



```
}  
}
```

## Python 语言

- 执行结果：通过
- 执行用时：48 ms, 在所有 Python3 提交中击败了 30.95% 的用户
- 内存消耗：13.7 MB, 在所有 Python3 提交中击败了 5.17% 的用户

```
# Definition for a binary tree node.  
# class TreeNode:  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution:  
    def isSymmetric(self, root: TreeNode) -> bool:  
        return self.isMirror(root, root)  
  
    def isMirror(self, t1: TreeNode, t2: TreeNode) -> bool:  
        if t1 is None and t2 is None:  
            return True  
        if t1 is None or t2 is None:  
            return False  
        return t1.val == t2.val and self.isMirror(t1.left, t2.right) and  
self.isMirror(t1.right, t2.left)
```

## 第二种：采用队列的方法

**思路：**利用二叉树的层次遍历的方式来实现。

- 执行结果：通过
- 执行用时：112 ms, 在所有 C# 提交中击败了 70.93% 的用户
- 内存消耗：24.9 MB, 在所有 C# 提交中击败了 5.17% 的用户

```
public class Solution
```

```

{
    public bool IsSymmetric(TreeNode root)
    {
        if (root == null)
            return true;

        Queue<TreeNode> nodes = new Queue<TreeNode>();
        nodes.Enqueue(root.left);
        nodes.Enqueue(root.right);
        while (nodes.Count != 0)
        {
            TreeNode node1 = nodes.Dequeue();
            TreeNode node2 = nodes.Dequeue();

            if (node1 == null && node2 == null)
                continue;
            if (node1 == null || node2 == null)
                return false;
            if (node1.val != node2.val)
                return false;
            nodes.Enqueue(node1.left);
            nodes.Enqueue(node2.right);
            nodes.Enqueue(node1.right);
            nodes.Enqueue(node2.left);
        }
        return true;
    }
}

```

---

## 4.3 二叉树的最大深度

---

- 题号：104
- 难度：简单
- <https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/>

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

**示例:**

给定二叉树 [3,9,20,null,null,15,7]

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

返回它的最大深度 3 。

---

### 第一种：利用队列实现层次遍历的思路

- 执行结果：通过
- 执行用时：108 ms, 在所有 C# 提交中击败了 88.13% 的用户
- 内存消耗：25.5 MB, 在所有 C# 提交中击败了 5.97% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public int MaxDepth(TreeNode root)
    {
```

```

        if (root == null)
            return 0;

        Queue<TreeNode> q = new Queue<TreeNode>();
        int deep = 0;
        q.Enqueue(root);

        while (q.Count != 0)
        {
            deep++;
            int count = 0;
            int size = q.Count;

            while (count < size)
            {
                TreeNode node = q.Dequeue();
                count++;

                if (node.left != null)
                    q.Enqueue(node.left);
                if (node.right != null)
                    q.Enqueue(node.right);
            }
        }
        return deep;
    }
}

```

## 第二种：利用递归

**思路：**递归分别求左右子树的最大深度，并加到原有层数上，最后返回两者中的最大值。

- 执行结果：通过
- 执行用时：132 ms, 在所有 C# 提交中击败了 16.62% 的用户
- 内存消耗：25.5 MB, 在所有 C# 提交中击败了 6.06% 的用户

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;

```

```

*   public TreeNode left;
*   public TreeNode right;
*   public TreeNode(int x) { val = x; }
* }
*/
public class Solution
{
    public int MaxDepth(TreeNode root)
    {
        if (root == null)
            return 0;
        int llen = 1;
        int rlen = 1;
        if (root.left != null)
        {
            llen += MaxDepth(root.left);
        }
        if (root.right != null)
        {
            rlen += MaxDepth(root.right);
        }
        return llen > rlen ? llen : rlen;
    }
}

```

## Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 93.87% 的用户
- 内存消耗：14.9 MB, 在所有 Python3 提交中击败了 10.18% 的用户

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if root is None:

```

```
        return 0
    llen, rlen = 1, 1
    if root.left is not None:
        llen += self.maxDepth(root.left)
    if root.right is not None:
        rlen += self.maxDepth(root.right)
    return max(llen, rlen)
```

---

## 4.4 二叉树的中序遍历

---

- 题号: 94
- 难度: 中等
- <https://leetcode-cn.com/problems/binary-tree-inorder-traversal/>

给定一个二叉树，返回它的中序 遍历。

**示例:**

输入: [1,null,2,3]

```
  1
   \
    2
   /
  3
```

输出: [1,3,2]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

---

**第一种: 利用栈**

- 执行结果：通过
- 执行用时：284 ms, 在所有 C# 提交中击败了 53.59% 的用户
- 内存消耗：30 MB, 在所有 C# 提交中击败了 6.67% 的用户

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public IList<int> InorderTraversal(TreeNode root)
    {
        IList<int> lst = new List<int>();
        Stack<TreeNode> stack = new Stack<TreeNode>();
        while (stack.Count != 0 || root != null)
        {
            if (root != null)
            {
                stack.Push(root);
                root = root.left;
            }
            else
            {
                TreeNode node = stack.Pop();
                lst.Add(node.val);
                root = node.right;
            }
        }
        return lst;
    }
}

```

## 第二种：使用递归

- 执行结果：通过

- 执行用时: 264 ms, 在所有 C# 提交中击败了 99.16% 的用户
- 内存消耗: 29.8 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public IList<int> InorderTraversal(TreeNode root)
    {
        IList<int> lst = new List<int>();
        MidOrder(root, lst);
        return lst;
    }
    private void MidOrder(TreeNode node, IList<int> lst)
    {
        if (node == null)
            return;
        MidOrder(node.left, lst);
        lst.Add(node.val);
        MidOrder(node.right, lst);
    }
}
```

## python 语言

- 执行结果: 通过
- 执行用时: 40 ms, 在所有 Python3 提交中击败了 41.39% 的用户
- 内存消耗: 13.6 MB, 在所有 Python3 提交中击败了 5.28% 的用户

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
```



```

#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        lst = list()
        self.midOrder(root, lst)
        return lst

    def midOrder(self, node: TreeNode, lst: list):
        if node is None:
            return
        self.midOrder(node.left, lst)
        lst.append(node.val)
        self.midOrder(node.right, lst)

```

## 4.5 不同的二叉搜索树 II

- 题号：95
- 难度：中等
- <https://leetcode-cn.com/problems/unique-binary-search-trees-ii/>

给定一个整数  $n$ ，生成所有由  $1 \dots n$  为节点所组成的二叉搜索树。

**示例:**

```

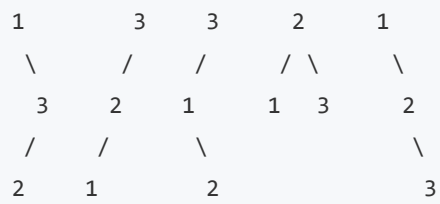
输入：3
输出：
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]

```

]

解释：

以上的输出对应以下 5 种不同结构的二叉搜索树：



## 第一种：递归

如果将  $i$  作为根节点，那么  $[1, i-1]$  为  $i$  的左子树节点， $[i+1, n]$  为右子树节点。

问题就被拆分为两个子问题了：

- 求左区间构成的所有二叉搜索树作为  $i$  的左子树
- 求右区间构成的所有二叉搜索树作为  $i$  的右子树

递归终止条件：

- 区间为空，返回 `null`。
- 区间的左右端点相同，即只包含一个数，返回该数构成的根结点。

以上就是利用递归求解该问题的思路。

- 执行结果：通过
- 执行用时：256 ms, 在所有 C# 提交中击败了 43.40% 的用户
- 内存消耗：28.6 MB, 在所有 C# 提交中击败了 9.09% 的用户

/\*\*

```

* Definition for a binary tree node.
* public class TreeNode {
*     public int val;
*     public TreeNode left;
*     public TreeNode right;
*     public TreeNode(int x) { val = x; }
* }
*/
public class Solution
{
    public IList<TreeNode> GenerateTrees(int n)
    {
        if (n == 0)
        {
            return new List<TreeNode>();
        }
        return GenerateTrees(1, n);
    }
    public List<TreeNode> GenerateTrees(int start, int end)
    {
        List<TreeNode> lst = new List<TreeNode>();
        //此时没有数字, 将 null 加入结果中
        if (start > end)
        {
            lst.Add(null);
            return lst;
        }
        //只有一个数字, 当前数字作为一棵树加入结果中
        if (start == end)
        {
            TreeNode tree = new TreeNode(start);
            lst.Add(tree);
            return lst;
        }
        //尝试每个数字作为根节点
        for (int i = start; i <= end; i++)
        {
            //得到所有可能的左子树
            List<TreeNode> leftTrees = GenerateTrees(start, i - 1);
            //得到所有可能的右子树
            List<TreeNode> rightTrees = GenerateTrees(i + 1, end);
            //左子树右子树两两组合
            foreach (TreeNode leftTree in leftTrees)
            {

```

```

        foreach (TreeNode rightTree in rightTrees)
        {
            TreeNode root = new TreeNode(i);
            root.left = leftTree;
            root.right = rightTree;
            //加入到最终结果中
            lst.Add(root);
        }
    }
}
return lst;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：52 ms, 在所有 Python3 提交中击败了 87.67% 的用户
- 内存消耗：15.2 MB, 在所有 Python3 提交中击败了 6.19% 的用户

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

```

**class Solution:**

```

    def generateTrees(self, n: int) -> List[TreeNode]:
        if n == 0:
            return list()
        return self.generate(1, n)

    def generate(self, start: int, end: int) -> List[TreeNode]:
        lst = list()
        if start > end:
            lst.append(None)
            return lst

        if start == end:
            tree = TreeNode(start)

```

```

        lst.append(tree)
    return lst

    for i in range(start, end + 1):
        leftTrees = self.generate(start, i - 1)
        rightTrees = self.generate(i + 1, end)
        for leftTree in leftTrees:
            for rightTree in rightTrees:
                tree = TreeNode(i)
                tree.left = leftTree
                tree.right = rightTree
                lst.append(tree)
    return lst

```

## 4.6 恢复二叉搜索树

- 题号：99
- 难度：困难
- <https://leetcode-cn.com/problems/recover-binary-search-tree/>

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树。

### 示例 1:

输入: [1,3,null,null,2]

```

  1
 /
3
 \
 2

```

输出: [3,1,null,null,2]

```
  3
 /
1
 \
 2
```

## 示例 2:

输入: [3,1,4,null,null,2]

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

输出: [2,1,4,null,null,3]

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

## 进阶:

- 使用  $O(n)$  空间复杂度的解法很容易实现。
- 你能想出一个只使用常数空间的解决方案吗?

---

## 第一种: 利用中序遍历

二叉搜索树是左孩子小于根节点, 右孩子大于根节点的树, 所以做一次中序遍历, 产生的序列就是从小到大排列的有序序列。

回到这道题，题目交换了两个数字，其实就是在有序序列中交换了两个数字。而我们只需要把它还原。

交换位置有两种情况：

- 相邻的两个数字交换

[ 1 2 3 4 5 ] 中 2 和 3 进行交换, [ 1 3 2 4 5 ], 这样的话只产生一组逆序的数字（正常情况是从小到大排序，交换后产生了从大到小），3 2。

我们只需要遍历数组，找到后，把这一组的两个数字进行交换即可。

- 不相邻的两个数字交换

[ 1 2 3 4 5 ] 中 2 和 5 进行交换, [ 1 5 3 4 2 ], 这样的话其实就是产生了两组逆序的数字对。5 3 和 4 2。

所以我们只需要遍历数组，然后找到这两组逆序对，然后把第一组前一个数字和第二组后一个数字进行交换即完成了还原。

综上，在中序遍历中，只需要利用一个 pre 节点和当前节点比较，如果 pre 节点的值大于当前节点的值，那么就是我们要找的逆序的数字。分别用两个指针 first 和 second 保存即可。如果找到第二组逆序的数字，我们就把 second 更新为当前节点。最后把 first 和 second 两个的数字交换即可。

- 执行结果：通过
- 执行用时：132 ms, 在所有 C# 提交中击败了 72.41% 的用户

- 内存消耗: 27.6 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public void RecoverTree(TreeNode root)
    {
        inorderTraversal(root);
        if(first == null || second == null)
            return;
        int temp = first.val;
        first.val = second.val;
        second.val = temp;
    }

    private TreeNode pre = null;
    private TreeNode first = null;
    private TreeNode second = null;
    private void inorderTraversal(TreeNode root)
    {
        if (root == null)
        {
            return;
        }
        inorderTraversal(root.left);

        if (pre != null && pre.val > root.val)
        {
            //第一次遇到逆序对
            if (first == null)
            {
                first = pre;
                second = root;
            }
            else
            {
                first = root;
            }
        }
        pre = root;
    }
}
```



```

        {
            //第二次遇到逆序对
            second = root;
        }
    }
    else
    {
        pre = root;
    }

    inorderTraversal(root.right);
}
}

```

## Python 语言

- 执行结果: 通过
- 执行用时: 64 ms, 在所有 Python3 提交中击败了 93.67% 的用户
- 内存消耗: 14 MB, 在所有 Python3 提交中击败了 5.32% 的用户

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

```

**class Solution:**

```

    def recoverTree(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        self.inorderTraversal(root)
        if self.first is None or self.second is None:
            return None
        self.first.val, self.second.val = self.second.val, self.first.val

    def __init__(self):
        self.pre = None
        self.first = None
        self.second = None

```

```
def inorderTraversal(self, root: TreeNode) -> None:
    if root is None:
        return None
    self.inorderTraversal(root.left)
    if self.pre is not None and self.pre.val > root.val:
        if self.first is None:
            self.first = self.pre
            self.second = root
        else:
            self.second = root
    else:
        self.pre = root
    self.inorderTraversal(root.right)
```

## 5. 贪心算法

---

**贪心算法** (greedy algorithm) , 又称贪婪算法, 是一种在每一步选择中都采取在当前状态下最好或最优 (即最有利) 的选择, 从而希望导致结果是最好或最优的算法。

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是局部最优解能决定全局最优解。简单地说, 问题能够分解成子问题来解决, 子问题的最优解能递推到最终问题的最优解。

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择, 不能回退。动态规划则会保存以前的运算结果, 并根据以前的结果对当前进行选择, 有回退功能。

---

### 5.1 买卖股票的最佳时机 II

---

- 题号: 122
- 难度: 简单
- <https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/>

给定一个数组, 它的第  $i$  个元素是一支给定股票第  $i$  天的价格。

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

**注意:** 你不能同时参与多笔交易 (你必须在再次购买前出售掉之前的股票) 。

### 示例 1:

输入: [7,1,5,3,6,4]

输出: 7

解释:

在第 2 天 (股票价格 = 1) 的时候买入, 在第 3 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 =  $5 - 1 = 4$  。

随后, 在第 4 天 (股票价格 = 3) 的时候买入, 在第 5 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 =  $6 - 3 = 3$  。

### 示例 2:

输入: [1,2,3,4,5]

输出: 4

解释:

在第 1 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 =  $5 - 1 = 4$  。

注意你不能在第 1 天和第 2 天接连购买股票, 之后再将它们卖出。

因为这样属于同时参与了多笔交易, 你必须在再次购买前出售掉之前的股票。

### 示例 3:

输入: [7,6,4,3,1]

输出: 0

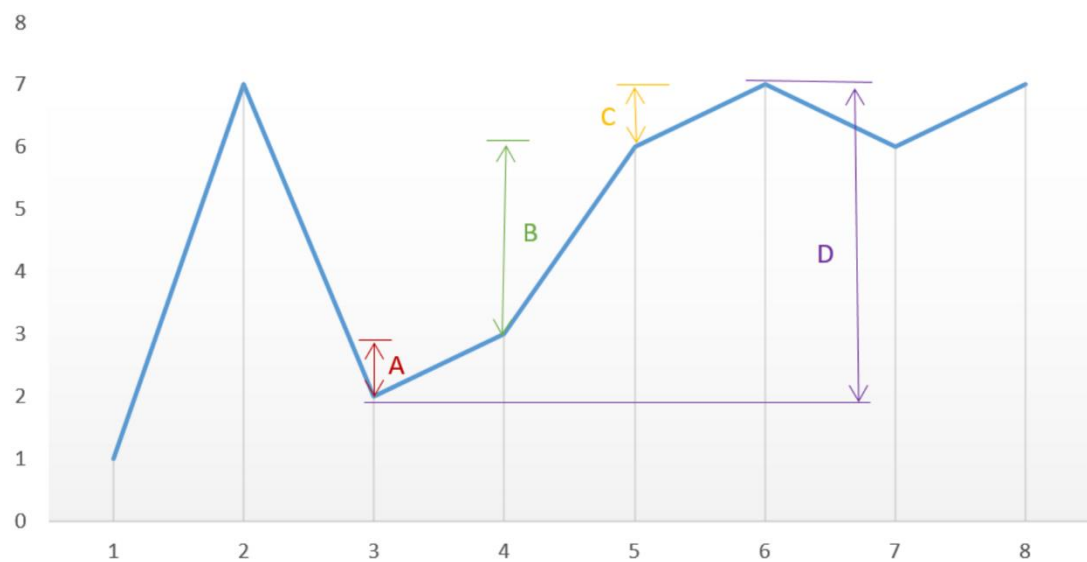
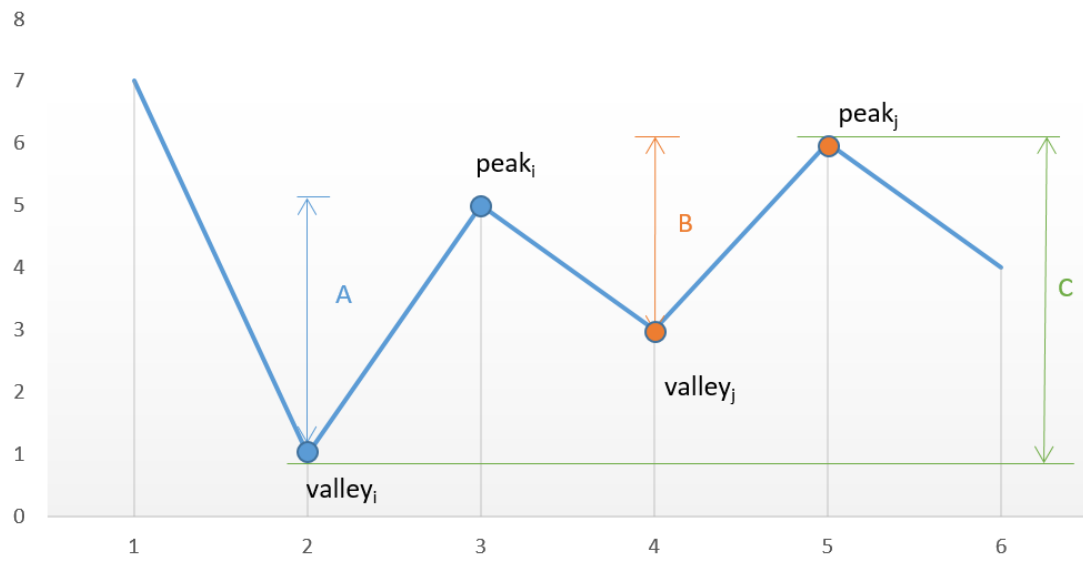
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

---

### 第一种: 贪心算法

贪心策略: 只要后一天价格比前一天高, 就在前一天买进后一天卖出。

## Maximum Profit



- 状态: 通过
- 201 / 201 个通过测试用例
- 执行用时: 140 ms, 在所有 C# 提交中击败了 72.02% 的用户
- 内存消耗: 24.2 MB, 在所有 C# 提交中击败了 5.36% 的用户

```
public class Solution
{
    public int MaxProfit(int[] prices)
```

```

{
    int earn = 0;
    for (int i = 0; i < prices.Length-1; i++)
    {
        earn += Math.Max(prices[i + 1] - prices[i], 0);
    }
    return earn;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 92.58% 的用户
- 内存消耗：14.7 MB, 在所有 Python3 提交中击败了 9.98% 的用户

```

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        earn = 0
        for i in range(0, len(prices) - 1):
            earn += max(prices[i + 1] - prices[i], 0)
        return earn

```

---

## 5.2 判断子序列

- 题号：392
- 难度：简单
- <https://leetcode-cn.com/problems/is-subsequence/>

给定字符串  $s$  和  $t$ ，判断  $s$  是否为  $t$  的子序列。

你可以认为  $s$  和  $t$  中仅包含英文小写字母。字符串  $t$  可能会很长（长度  $\sim 500,000$ ），而  $s$  是个短字符串（长度  $\leq 100$ ）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

### 示例 1:

```
s = "abc", t = "ahbgdc"
```

返回 `true`。

### 示例 2:

```
s = "axc", t = "ahbgdc"
```

返回 `false`。

### 示例 3:

```
s = "", t = "ahbgdc"
```

返回 `true`。

### 后续挑战:

如果有大量输入的  $s$ ，称作  $s_1, s_2, \dots, s_k$  其中  $k \geq 10$  亿，你需要依次检查它们是否为  $t$  的子序列。在这种情况下，你会怎样改变代码？

---

### 第一种：贪心算法

贪心算法必须具备后无效性，也就是不必考虑前面的影响，只需考虑当前的状态。

若  $s = \text{"abc"}$ ,  $t = \text{"ahbgdc"}$  要判断字符串  $s$  和  $t$  的排序是否一致, 我们这样构造贪心策略:

- 取  $s$  中的第一个字符  $a$ , 判断  $t$  中是否存在字符  $a$
- 若存在, 则判断  $t$  中字符  $a$  之后的剩余字符串是否存在第二个字符  $b$
- 依次类推

用一句通俗的话就是剩余字符串中是否存在下一个字符, 利用贪心算法的概念就是局部是否存在最优解。

- 执行结果: 通过
- 执行用时: 116 ms, 在所有 C# 提交中击败了 42.17% 的用户
- 内存消耗: 43 MB, 在所有 C# 提交中击败了 7.41% 的用户

```
public class Solution
{
    public bool IsSubsequence(string s, string t)
    {
        for (int i = 0; i < s.Length; i++)
        {
            int j = t.IndexOf(s[i]);
            if (j == -1)
                return false;
            t = t.Substring(j + 1);
        }
        return true;
    }
}
```

## 第二种: 双索引法

利用两个变量  $i, j$  来记录搜索的位置,  $i$  记录搜索到  $t$  的位置,  $j$  记录搜索到  $s$  的位置, 一旦  $j == s.Length$  表示  $s$  的字符全部搜索完成, 即  $s$  是  $t$  的子序列, 返回 `true` 即可。



- 执行结果：通过
- 执行用时：112 ms, 在所有 C# 提交中击败了 56.63% 的用户
- 内存消耗：38 MB, 在所有 C# 提交中击败了 25.93% 的用户

```
public class Solution
{
    public bool IsSubsequence(string s, string t)
    {
        if (string.IsNullOrEmpty(s))
            return true;
        int j = 0;
        for (int i = 0; i < t.Length; i++)
        {
            if (t[i] == s[j])
            {
                j++;
            }
            if (j == s.Length)
                return true;
        }
        return false;
    }
}
```

## Python 语言

- 执行结果：通过
- 执行用时：268 ms, 在所有 Python3 提交中击败了 35.49% 的用户
- 内存消耗：18 MB, 在所有 Python3 提交中击败了 8.35% 的用户

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        if len(s) == 0:
            return True
        j = 0
        for i in range(0, len(t)):
            if t[i] == s[j]:
                j += 1
```

```
        if j == len(s):
            return True
    return False
```

---

## 5.3 分发饼干

---

- 题号：455
- 难度：简单
- <https://leetcode-cn.com/problems/assign-cookies/>

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子  $i$ ，都有一个胃口值  $g_i$ ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干  $j$ ，都有一个尺寸  $s_j$ 。如果  $s_j \geq g_i$ ，我们可以将这个饼干  $j$  分配给孩子  $i$ ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

**注意：**

你可以假设胃口值为正。

一个小朋友最多只能拥有一块饼干。

**示例 1：**

输入：[1,2,3], [1,1]

输出：1

解释：

你有三个孩子和两块小饼干，3 个孩子的胃口值分别是：1,2,3。  
虽然你有两块小饼干，由于他们的尺寸都是 1，你只能让胃口值是 1 的孩子满足。  
所以你应该输出 1。

## 示例 2:

输入: [1,2], [1,2,3]

输出: 2

解释:

你有两个孩子和三块小饼干，2 个孩子的胃口值分别是 1,2。  
你拥有的饼干数量和尺寸都足以让所有孩子满足。  
所以你应该输出 2。

## 示例 3:

输入: [10,9,8,7], [5,6,7,8]

输出: 2

---

## 第一种：贪心算法

贪心策略：优先满足胃口小的小朋友的需求。

- 对  $g$  和  $s$  升序排序
- 初始化两个索引  $i$  和  $j$  分别指向  $g$  和  $s$  初始位置

若  $g[i] \leq s[j]$ ：饼干满足胃口，把能满足的孩子数量加 1，并移动指针  $i++$ ,  $j++$ 。

若  $g[i] > s[j]$ ：无法满足胃口， $j++$ ，继续查看下一块饼干是否可以满足胃口。

- 执行结果：通过
- 执行用时：160 ms, 在所有 C# 提交中击败了 58.21% 的用户
- 内存消耗：30.6 MB, 在所有 C# 提交中击败了 11.11% 的用户

```

public class Solution
{
    public int FindContentChildren(int[] g, int[] s)
    {
        Array.Sort(g);
        Array.Sort(s);

        int i = 0;
        int j = 0;
        int result = 0;

        while (i < g.Length && j < s.Length)
        {
            if (s[j] < g[i])
            {
                j++;
            }
            else
            {
                i++;
                j++;
                result++;
            }
        }
        return result;
    }
}

```

对以上代码进行简化：

- 执行结果：通过
- 执行用时：160 ms, 在所有 C# 提交中击败了 58.21% 的用户
- 内存消耗：30.4 MB, 在所有 C# 提交中击败了 11.11% 的用户

```

public class Solution
{
    public int FindContentChildren(int[] g, int[] s)
    {
        Array.Sort(g);
        Array.Sort(s);
    }
}

```

```

int i = 0;
int j = 0;
int result = 0;

while (i < g.Length && j < s.Length)
{
    if (s[j] >= g[i])
    {
        i++;
        result++;
    }
    j++;
}
return result;
}
}

```

## Python 语言

- 执行结果：通过
- 执行用时：336 ms, 在所有 Python3 提交中击败了 15.90% 的用户
- 内存消耗：15.3 MB, 在所有 Python3 提交中击败了 5.49% 的用户

```

class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        g = sorted(g)
        s = sorted(s)
        i, j, result = 0, 0, 0
        while i < len(g) and j < len(s):
            if s[j] >= g[i]:
                i += 1
                result += 1
            j += 1
        return result

```

---

## 5.4 跳跃游戏

- 题号: 55
- 难度: 中等
- <https://leetcode-cn.com/problems/jump-game/>

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

### 示例 1:

输入: [2,3,1,1,4]

输出: true

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

### 示例 2:

输入: [3,2,1,0,4]

输出: false

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

### 示例 3:

输入: [0]

输出: true

---

### 第一种: 贪心算法

贪心策略: 每次记录能跳到点的最大值，如果当前点超出最大值，返回 false，如果最大值

达到最后一个位置，返回 true。

- 执行结果：通过
- 执行用时：120 ms, 在所有 C# 提交中击败了 57.32% 的用户
- 内存消耗：26.2 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
public class Solution
{
    public bool CanJump(int[] nums)
    {
        int maxlength = 0;    //记录所能到达的最远点
        for (int i = 0; maxlength < nums.Length-1; i++)
        {
            if (i > maxlength)
            {
                return false; //若此点已不能到达, 返回false
            }
            maxlength = Math.Max(i + nums[i], maxlength);
        }
        return true;
    }
}
```

## Python 语言

- 执行结果：通过
- 执行用时：48 ms, 在所有 Python3 提交中击败了 88.65% 的用户
- 内存消耗：14.8 MB, 在所有 Python3 提交中击败了 66.85% 的用户

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        maxlength = 0
        for i, num in enumerate(nums):
            if maxlength >= len(nums) - 1:
                break
            if i > maxlength:
                return False
            maxlength = max(i + num, maxlength)
        return True
```

---

## 5.5 加油站

---

- 题号：134
- 难度：中等
- <https://leetcode-cn.com/problems/gas-station/>

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。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回  $-1$ 。

### 说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

### 示例 1:

输入:

$gas = [1,2,3,4,5]$

$cost = [3,4,5,1,2]$

输出: 3

解释:

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

解释:

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。

---

## 第一种: 模拟仿真

首先: 只有第  $i$  站的  $gas[i] \geq cost[i]$  (加入汽油数  $\geq$  消耗汽油数), 才能作为起始加油站。此时剩余油数为  $Remaining = gas[i] - cost[i]$ 。

其次: 第  $i+1$  站的剩余油数为  $Remaining += gas[i+1] - cost[i+1]$  只有  $Remaining > 0$  才能前往下一站, 否则不能前往下一站。

以此类推, 若能跑完一周, 则找到起始的加油站。

- 执行结果: 通过
- 执行用时: 236 ms, 在所有 C# 提交中击败了 20.75% 的用户
- 内存消耗: 24.7 MB, 在所有 C# 提交中击败了 14.29% 的用户

```

public class Solution
{
    public int CanCompleteCircuit(int[] gas, int[] cost)
    {
        for (int i = 0; i < gas.Length; i++)
        {
            if (gas[i] >= cost[i])
            {
                if (IsCan(gas, cost, i))
                {
                    return i;
                }
            }
        }
        return -1;
    }

    private bool IsCan(int[] gas, int[] cost, int index)
    {
        int Remaining = gas[index] - cost[index];
        for (int i = 1; i < gas.Length; i++)
        {
            int j = (index + i) % gas.Length;
            Remaining += gas[j] - cost[j];
            if (Remaining < 0)
            {
                return false;
            }
        }
        return true;
    }
}

```

## 第二种：连续求和法

易知当总加入汽油数大于等于总消耗油汽油数时，无论如何都有解，关键就在于找出起点。

又因为有解时答案唯一，故可将问题转化为求连续和法，若连续和小于等于 0，那么此段中肯定不含可以作起点的油站，将下一个油站记为起点继续计算。

- 执行结果：通过
- 执行用时：116 ms, 在所有 C# 提交中击败了 60.38% 的用户
- 内存消耗：24.7 MB, 在所有 C# 提交中击败了 14.29% 的用户

```
public class Solution
{
    public int CanCompleteCircuit(int[] gas, int[] cost)
    {
        int total = 0;
        int index = 0;
        int keeping = 0;
        for (int i = 0; i < gas.Length; i++)
        {
            int Remaining = gas[i] - cost[i];
            total += Remaining;
            if (keeping > 0)
            {
                keeping += Remaining;
            }
            else
            {
                keeping = Remaining;
                index = i;
            }
        }
        return total < 0 ? -1 : index;
    }
}
```

## python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 90.29% 的用户
- 内存消耗：14.3 MB, 在所有 Python3 提交中击败了 5.52% 的用户

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        total, index, keeping = 0, 0, 0
```

```
for i in range(len(gas)):
    remaining = gas[i] - cost[i]
    total += remaining
    if keeping > 0:
        keeping += remaining
    else:
        keeping = remaining
        index = i
return -1 if total < 0 else index
```

---

## 5.6 通配符匹配

---

- 题号：44
- 难度：困难
- <https://leetcode-cn.com/problems/wildcard-matching/>

给定一个字符串(s) 和一个字符模式(p) ，实现一个支持 '?' 和 '\*' 的通配符匹配。

'?' 可以匹配任何单个字符。

'\*' 可以匹配任意字符串（包括空字符串）。

两个字符串完全匹配才算匹配成功。

**说明:**

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 \*。

**示例 1:**

输入:

s = "aa"

p = "a"

输出: false

解释: "a" 无法匹配 "aa" 整个字符串。

### 示例 2:

输入:

```
s = "aa"
```

```
p = "*"
```

输出: `true`

解释: '\*' 可以匹配任意字符串。

### 示例 3:

输入:

```
s = "cb"
```

```
p = "?a"
```

输出: `false`

解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。

### 示例 4:

输入:

```
s = "adceb"
```

```
p = "*a*b"
```

输出: `true`

解释: 第一个 '\*' 可以匹配空字符串, 第二个 '\*' 可以匹配字符串 "dce".

### 示例 5:

输入:

```
s = "acdc b"
```

```
p = "a*c?b"
```

输出: `false`

### 示例 6:

输入:

```
"abefcdgiescdfimde"
```

```
"ab*cd?i*de"
```

输出: `true`

### 示例 7:

输入:

```
"aaaa"
```

```
"***a"
```

输出: `true`

---

## 第一种：双索引法

我们用  $i$  和  $j$  分别标记  $s$  和  $p$  的第一个字符下标，即都初始化为 0。用  $istart$  和  $jstart$  分别标记  $s$  和  $p$  中 '\*' 匹配过的位置，即初始化为 -1。

和普通字符串匹配的思路差不多，已经匹配成功的部分就不再考虑了，所以要用  $i$  和  $j$  标记当前正在比较的字符；但是最近匹配过的 '\*' 可能会被重复使用去匹配更多的字符，所以我们要用  $istart$  和  $jstart$  分别标记  $s$  和  $p$  中最近匹配过 '\*' 的位置。

1. 如果  $i$  和  $j$  标记的字符正好相等或者  $j$  字符是 '?' 匹配成功，则"移除"  $i$  和  $j$  元素，即自增  $i$ 、 $j$ 。
2. 否则如果  $j$  字符是 '\*' 依然可以匹配成功，则用  $istart$  和  $jstart$  分别标记  $i$  元素和  $j$  元素，自增  $j$ 。
3. 再否则如果  $istart > -1$  说明之前匹配过 '\*'，因为 '\*' 可以匹配多个字符，所以这里要再次利用这个最近匹配过的 '\*' 匹配更多的字符，移动  $i$  标记  $istart$  的下一个字符，再让  $istart$  重新标记  $i$  元素，同时移动  $j$  标记  $jstart$  的下一个字符。
4. 上述三种情况都不满足，则匹配失败，返回 false。

最后当  $s$  中的字符都判断完毕，则认为  $s$  为空，此时需要  $p$  为空或者  $p$  中只剩下星号的时候，才能成功匹配。

- 执行结果：通过
- 执行用时：92 ms, 在所有 C# 提交中击败了 95.00% 的用户
- 内存消耗：25.7 MB, 在所有 C# 提交中击败了 66.67% 的用户

```

public class Solution
{
    public bool IsMatch(string s, string p)
    {
        //若正则串 p 为空串, 则 s 为空串匹配成功, s 不为空串匹配失败。
        if (string.IsNullOrEmpty(p))
            return string.IsNullOrEmpty(s) ? true : false;

        int i = 0, j = 0, istart = -1, jstart = -1, plen = p.Length;

        //判断 s 的所有字符是否匹配
        while (i < s.Length)
        {
            //三种匹配成功情况以及匹配失败返回 false
            if (j < plen && (s[i] == p[j] || p[j] == '?'))
            {
                i++;
                j++;
            }
            else if (j < plen && p[j] == '*')
            {
                istart = i;
                jstart = j;
                j++;
            }
            else if (istart > -1)
            {
                i = istart + 1;
                istart = i;
                j = jstart + 1;
            }
            else
            {
                return false;
            }
        }
        //s 中的字符都判断完毕, 则认为 s 为空
        //此时需要 p 为空或者 p 中只剩下星号的时候, 才能成功匹配。
        //如果 p 中剩余的都是*, 则可以移除剩余的*
        while (j < plen && p[j] == '*')
        {
            j++;
        }
        return j == plen;
    }
}

```

```
}  
}
```

## 第二种：动态规划

dp 数组的含义：dp[i,j]意思是 s 的前 i 个元素能否被 p 的前 j 个元素成功匹配。

知道了 dp 数组的含义之后，我们就知道了初始化细节：

1. bool 类型的 dp 数组，大小是[s.length+1,p.length+1]，因为存在 s 前 0 个字符和 p 前 0 个字符的情况，即 s 为空串或 p 为空串。
2. dp[0,0]一定是 true，因为 s 空串和 p 空串是可以匹配成功的；dp[1,0] ~ dp[s.length,0]一定都是 false，因为 s 不为空串而 p 为空串是不能匹配成功的。
3. dp[0,1] ~ dp[0,p.length]当 s 为空串的时候，而 p 不是空串的时候，当且仅当 p 的 j 字符以及前面都为 '\*' 才为 true。
4. dp[s.length,p.length]就得到了 s 和 p 最终的匹配情况。

有了上述理解之后，就可以初始化 dp 数组了。

然后填写 dp 数组剩余部分即可，状态转移方程：

1. 当 s[i] == p[j]或者 p[j] == '?', 则 dp[i,j] = dp[i-1,j-1]。可以理解为当前字符成功匹配后，只需要考虑之前的字符串是否匹配即可。
2. 当 p[j] == '\*', 则 dp[i,j] = dp[i-1,j] || dp[i,j-1]。可以理解为当字符为 '\*' 的时候会出现两种情况，第一种是 '\*' 需要作为一个字母与 s[i] 进行匹配；第二种是 '\*' 需要作为空字符(即不需要 '\*' 可以直接"移除")，所以 dp[i,j-1]；用逻辑或将两种情况连接，是因为只要有一种情况可以匹配成功则当前匹配成功，有点暴力算法的感觉。



3. 最后当  $s[i] \neq p[j]$  且  $p[j] \neq '*'$ ,  $dp[i,j] = false$ 。这步可以省略, 因为  $dp$

数组元素的默认值就是 `false`, 所以不必要进行显式的赋值为 `false`。

有了上面的理解, 我们就可以写代码了。

- 执行结果: 通过
- 执行用时: 112 ms, 在所有 C# 提交中击败了 62.50% 的用户
- 内存消耗: 28.6 MB, 在所有 C# 提交中击败了 22.22% 的用户

```
class Solution
{
    public bool IsMatch(string s, string p)
    {
        if (string.IsNullOrEmpty(p))
            return string.IsNullOrEmpty(s) ? true : false;

        int slen = s.Length, plen = p.Length;
        bool[,] dp = new bool[slen + 1, plen + 1];

        //初始化 dp 数组
        //dp[1][0]~dp[s.Length][0]默认值false 不需要显式初始化为false
        dp[0, 0] = true;

        //dp[0][1]~dp[0][p.Length]只有p的j字符以及前面所有字符都为'*'才为true
        for (int j = 1; j <= plen; j++)
        {
            dp[0, j] = p[j - 1] == '*' && dp[0, j - 1];
        }

        //填写 dp 数组剩余部分
        for (int i = 1; i <= slen; i++)
        {
            for (int j = 1; j <= plen; j++)
            {
                char si = s[i - 1], pj = p[j - 1];
                if (si == pj || pj == '?')
                {
                    dp[i, j] = dp[i - 1, j - 1];
                }
            }
        }
    }
}
```

```
    }  
    else if (pj == '*')  
    {  
        dp[i, j] = dp[i - 1, j] || dp[i, j - 1];  
    }  
}  
}  
return dp[slen, plen];  
}  
}
```