{% note danger no-icon flat %}
{% btn 'https://leetcode.cn/studyplan/top-100-liked/',LeetCode,fa fa-share ,larger outline%}
Start: 2024.10.23
End: 2025.03.02
Plan: AM 9:00-10:30, PM 22:30-24:00
Check:

1	2	3	4	5	6	7	8	9	10
10.23	10.23	10.23	10.24	10.24	10.24	10.25	10.25	10.25	10.27
10.27	10.28	10.28	10.28	10.29	10.29	10.29	11.1	11.1	11.1
11.4	11.5	11.5	11.5	11.6	11.6	11.6	11.7	11.7	11.9
11.9	11.9	11.10	11.10	11.11	11.11	11.11	11.12	11.12	11.12
11.13	11.13	11.13	11.14	11.14	11.14	11.16	01.06	01.07	01.08
02.09	02.09	02.09	**02.17**	02.17	02.17	02.18	02.18	02.18	02.18
02.18	02.19	02.19	02.20	02.20	02.21	02.21	02.21	02.21	02.22
02.22	02.24	02.24	02.24	02.24	02.25	02.25	02.25	02.25	02.26
02.26	02.26	02.26	02.27	02.27	02.27	02.27	02.28	02.28	02.28
02.28	03.01	03.01	03.01	03.01	03.02	03.02	03.02	03.02	03.02

- 2025.03.02：完结撒花 🌸

{% endnote %}

一、哈希

1. 两数之和

- **题面：**

  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，并且你不能使用两次相同的元素。

  你可以按任意顺序返回答案。

  - **只会存在一个有效答案**

- **示例 1：**

```
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
```

示例 2：

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

示例 3：

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

{% tabs 解法, -1 %}

暴力枚举肯定是可以的，但二重循环，时间复杂度高。

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                if (nums[i] + nums[j] == target) {
                    return {i, j};
                }
            }
        }
        return {};
    }
};
```

时间复杂度：$O(N^2)$.

空间复杂度：$O(1)$.

哈希表

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        // 哈希表
        unordered_map<int, int> hashtable;
        for(int i=0; i<nums.size(); ++i){
            auto it = hashtable.find(target-nums[i]);
            if(it != hashtable.end())
                return {it->second, i};
            // 没找到的插入，以数值为键，索引为值
            hashtable[nums[i]] = i;
        }
        return {};
    }
};
```

时间复杂度：$O(N)$.

空间复杂度：$O(N)$.

{% endtabs %}

2. 字母异位词分组

- **题面：**

  给你一个字符串数组，请你将 **字母异位词** 组合在一起。可以按任意顺序返回结果列表。

  字母异位词 是由重新排列源单词的所有字母得到的一个新单词。

  - `strs[i]` 仅包含小写字母

- **示例 1:**

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

  示例 2:

  ```
  输入: strs = [""]
  输出: [[""]]
  ```

  示例 3:

```
输入：strs = ["a"]
输出：[["a"]]
```

{% tabs 解法, -1 %}

字符串自身排序作为键

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        // 把字符串自身排序，作为键，原字符串数组作为值
        unordered_map<string, vector<string>> hashtable;
        for(string &s : strs){
            string key = s;
            sort(key.begin(), key.end());
            hashtable[key].emplace_back(s);
        }
        // 插入完毕就完成了
        vector<vector<string>> ans;
        for(auto it=hashtable.begin(); it!=hashtable.end(); ++it){
            ans.emplace_back(it->second);
        }
        return ans;
    }
};
```

时间复杂度：$O(nk\log k)$，其中，n字符串数量，k最长字符串长度.

空间复杂度：$O(nk)$.

还可以计数 26 个字母出现的次数，成为一个数组，这个数组作为键。

写的太复杂，这里不给代码了。

{% endtabs %}

3. 最长连续序列

- **题面：**

  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 1：**

```
输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

示例 2：

```
输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9
```

{% tabs 解法, -1 %}

注意点：

1. 使用 `unordered_set`
2. 不要重复查找，有前序数字的数字肯定不会是最长序列的开端。

   例如，有4,5,6,7,那么，5 肯定不会是最长序列的开端，同理，6,7也不会是。

   怎么判断：看是不是存在前序数字即可，比如 5 看是不是存在 4。

```cpp
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        // 存入 hash 集合
        unordered_set<int> hashset;
        for(int &x : nums)
            hashset.insert(x);
        // 遍历
        int maxlen = 0;
        for(int x : hashset){
            // 若有前序，则本次数字不会是最长序列的开端
            if(hashset.find(x-1) != hashset.end())
                continue;
            // 若可为开端，则依次查到最长序列为止
            int len = 0;
            while(hashset.find(x+len) != hashset.end())
                len++;
            maxlen = max(maxlen, len);
        }
        return maxlen;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

二、双指针

4. 移动零

- **题面：**

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 1:**

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

示例 2:

```
输入: nums = [0]
输出: [0]
```

{% tabs 解法, -1 %}

双指针法

```cpp
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        // 不用管 0，看到非 0 的搬到前面即可
        // 从头扫一遍，p 记录下一个非 0 可以放的位置
        int n = nums.size(), p = 0;
        for(int i=0; i<n; i++)
            if(nums[i] != 0)
                nums[p++] = nums[i];
        // 剩余是末尾全 0
        while(p < n) nums[p++] = 0;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

5. 盛最多水的容器

- **题面:**

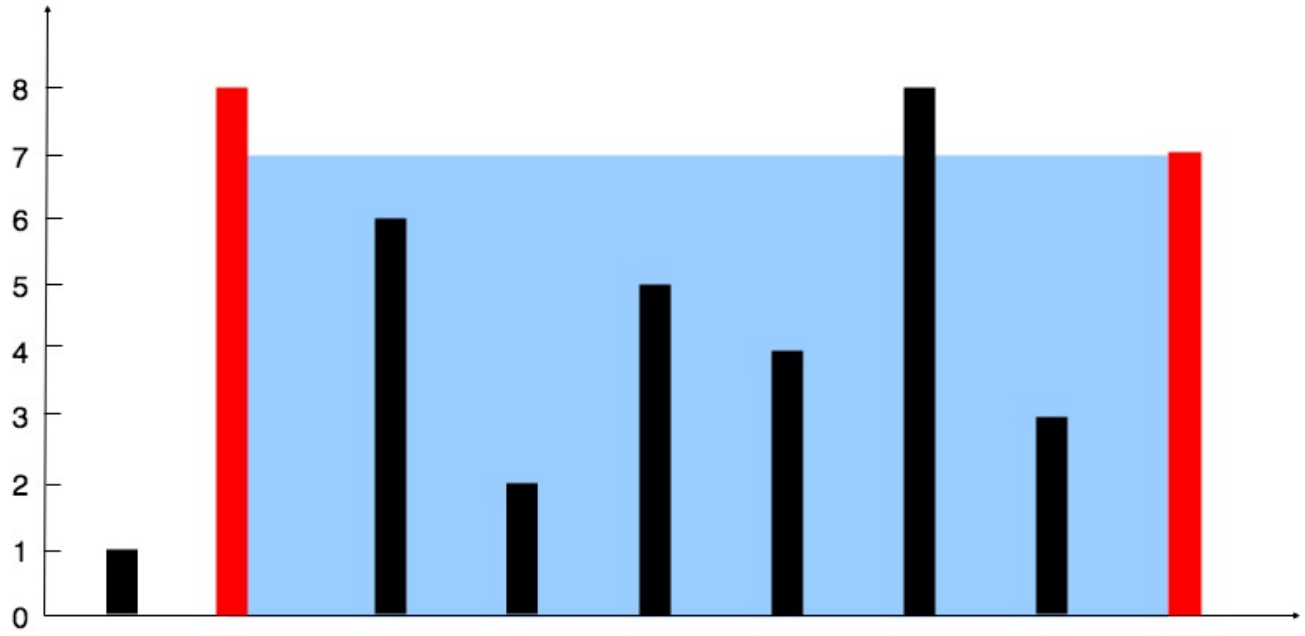
给定一个长度为 `n` 的整数数组 `height` 。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])` 。

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。

- **示例 1:**




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

示例 2：

```
输入：height = [1,1]
输出：1
```

{% tabs 解法, -1 %}

关键在于思考怎么移动指针。

```cpp
class Solution {
public:
    int maxArea(vector<int>& height) {
        int n = height.size();
        int L = 0, R = n-1, maxS = 0;
        while(L < R){
            int S = (R - L)* min(height[L], height[R]);
            if(S > maxS)
                maxS = S;
            // 总是移动 L,R 中目前高度更小的那个，保留高的
            if(height[L] < height[R]) L++;
            else R--;
        }
        return maxS;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

6. 三数之和

- **题面：**

  给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请你返回所有和为 0 且不重复的三元组。

  注意：答案中不可以包含重复的三元组。

- **示例 1：**

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
解释：
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。
注意，输出的顺序和三元组的顺序并不重要。
```

示例 2：

```
输入：nums = [0,1,1]
输出：[]
解释：唯一可能的三元组和不为 0 。
```

示例 3：

```
输入：nums = [0,0,0]
输出：[[0,0,0]]
解释：唯一可能的三元组和为 0 。
```

{% tabs 解法, -1 %}

三重循环查找不可避免。

1. 查找时，同一个位置不能出现相同的数字，因此先排序是必要的。
2. 可以发现，要 `a+b+c = 0` 那么，确定`a`的情况下，`b`和`c`是对立的，此消彼长，因此不需要完完全全的三重循环，二重循环即可。
3. 注意 `c` 下标应该大于 `b` 下标。

```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
```

```
        int n = nums.size();
        vector<vector<int>> ans;
        // 为了不重复，先排序
        sort(nums.begin(), nums.end());
        // 开始查找
        for(int i=0; i<n; i++){
            // 注意，不能重复，因此下一次的数字应与前一个数字不一样
            if(i!=0 && nums[i-1]==nums[i])
                continue;
            for(int j=i+1, k=n-1; j<n && j<k; j++){
                if(j!=i+1 && nums[j-1]==nums[j])
                    continue;
                while(nums[i]+nums[j]+nums[k] > 0 && j<k) k--;
                if(nums[i]+nums[j]+nums[k] == 0 && j<k)
                    ans.push_back({nums[i], nums[j], nums[k]});
            }
        }
        return ans;
        // 注意理解，若原数组是[-4,-1,-1,0,1,2],那么[-1,-1,0]是可以的
        // 不能重复是三元组本身不能重复，不是说三元组内的数字不能重复
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(\log n)$或$O(n)$. 主要是排序的消耗。

{% endtabs %}

7. 接雨水

- **题面：**

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

- **示例 1：**


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

示例 2：

```
输入：height = [4,2,0,3,2,5]
输出：9
```

{% tabs 解法, -1 %}

动态规划

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        // 动态规划
        // 下标i处能接的最多水 = 下标i左右两侧最高度的较小值 - 下标i的高度
        // 1. 动态规划得到每个i的左右两侧最高度
        int n = height.size();
        vector<int> lmax(n), rmax(n);
        // 顺着求 lmax
        lmax[0] = height[0];
        for(int i=1; i<n; i++)
            lmax[i] = max(lmax[i-1], height[i]);
        // 逆着求 rmax
        rmax[n-1] = height[n-1];
        for(int i=n-2; i>=0; i--)
            rmax[i] = max(rmax[i+1], height[i]);

        // 2. 计算即可
        int Sum = 0;
        for(int i=0; i<n; i++)
            Sum += min(lmax[i], rmax[i]) - height[i];

        return Sum;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

双指针，可以降低空间复杂度

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        // 双指针
        // 双方轮流占据最高点，低的一方移动
        int n = height.size();
        int lp = 0, rp = n-1;
        int lmax = 0, rmax = 0;
        int Sum = 0;
```

```
        while(lp < rp){
            lmax = max(lmax, height[lp]), rmax = max(rmax, height[rp]);
            if(lmax < rmax){
                Sum += lmax - height[lp];
                lp++;
            }
            else{
                Sum += rmax - height[rp];
                rp--;
            }
        }
        return Sum;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

三、滑动窗口

8. 无重复字符的最长子串

- **题面：**

  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：是字符串中**连续**的 **非空** 字符序列。

- **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。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        // 滑动窗口 + 哈希集合（用来检测是否存在）
        int n = s.size();
        unordered_set<char> lookup;
        int maxlen = 0, left = 0;
        for(int i=0; i<n; i++){
            // 当前的 i 字符无论如何，必须进去
            // 那么窗口左侧一直滑动，滑到可以为止
            while(lookup.find(s[i]) != lookup.end()){
                lookup.erase(s[left]);
                left++;
            }
            maxlen = max(maxlen, i-left+1);
            lookup.insert(s[i]);
        }

        return maxlen;
    }
};
```

{% endtabs %}

9. 找到字符串中所有字母异位词

- **题面：**

  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。

- **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。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        // 滑动窗口
        // 窗口固定大小，把p也看做窗口，直接检测整个窗口是否一样即可
        // 全是小写字母，转成 26 字母计数
        vector<int> SW(26);
        vector<int> PW(26);
        vector<int> ans;
        int n = s.size(), m = p.size();
        // swin pwin 都有固定大小m，后续移动，一定是直接整体移动
        if(n < m)
            return {};
        for(int i=0; i<m; i++)
            SW[s[i]-'a']++, PW[p[i]-'a']++;
        // 开始滑动
        if(SW == PW) ans.emplace_back(0);
        for(int i=0; i<n-m; i++){
            SW[s[i+m]-'a']++, SW[s[i]-'a']--;
            if(SW == PW) ans.emplace_back(i+1); // 已移动，左侧是 i+1
        }
        return ans;
    }
};
```

{% endtabs %}

四、子串

10. 和为 K 的子数组

- **题面:**

  给你一个整数数组 `nums` 和一个整数 `k` ，请你统计并返回 该数组中和为 `k` 的子数组的个数 。

  子数组是数组中元素的连续非空序列。

- **示例 1:**

```
输入：nums = [1,1,1], k = 2
输出：2
```

示例 2：

```
输入：nums = [1,2,3], k = 3
输出：2
```

{% tabs 解法, -1 %}

首先，朴素做法：

不佳，会报出：超出内存限制。

```cpp
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        // 找出 [i,j] 和为 k
        int n = nums.size();
        int ans = 0;
        vector<vector<int>> Sum(n, vector<int>(n, 0));
        // 初始化
        for(int i=0; i<n; i++){
            Sum[i][i] = nums[i];
            if(Sum[i][i] == k) ans++;
        }
        // 更新
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++){
                Sum[i][j] = Sum[i][j-1] + nums[j];  // 动态规划
                if(Sum[i][j] == k) ans++;
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(n^2)$.

改进变量使用：

```cpp
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        // 找出 [i,j] 和为 k
        int n = nums.size();
        int ans = 0;
        // 原本动态规划，实际上发现，每次只会用到前一次的计算，那么一个变量就够
```

```
        for(int i=0; i<n; i++){
            int Sum = 0;
            for(int j=i; j<n; j++){
                Sum += nums[j];
                if(Sum == k) ans++;
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

只求前缀和，并且用哈希表存储：

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        // 关键在于希望不要每次[i, j] 每次到j，要匹配前面的i出来
        // 能不能直接得到满足条件的i数量?
        // [i, j] = [0, j] - [0, i]  (其中，i <= j)
        // 那么    [i, j] == k
        // 可以转为 [0, i] == [0, j] - k 每次要找到这个的数量即可
        int n = nums.size(), pre = 0, ans = 0;
        unordered_map<int, int> mp;  // 哈希表，键是前缀和，值是次数
        mp[pre]++;
        // 很巧妙，计算的同时插入mp，顺利保证了不会出现序列下标的错误
        for(int &num : nums){
            pre += num;
            if(mp.find(pre - k) != mp.end())  ans += mp[pre - k];
            mp[pre]++;  // 应该放后面，否则可能出现给自己加次数
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

11.滑动窗口最大值

- **题面：**

  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 。

- **示例 1：**

```
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
```

示例 2：

```
输入：nums = [1], k = 1
输出：[1]
```

{% tabs 解法, -1 %}

朴素做法，超出时间限制：

```cpp
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> ans;
        deque<int> Q;
        // 初始窗口
        for(int i=0; i<k; i++)  Q.push_back(nums[i]);
        int m = Q[0];
        for(int i=1; i<k; i++)  m = max(m, Q[i]);
        ans.emplace_back(m);
        // 移动窗口
        for(int i=k; i<n; i++){
            Q.pop_front(), Q.push_back(nums[i]);
            int m = Q[0];
            for(int i=1; i<k; i++)  m = max(m, Q[i]);
            ans.emplace_back(m);
        }
        return ans;
    }
};
```

时间复杂度：$O(nk)$.

空间复杂度：$O(n+k)$.

优先队列是最应该想到的正确方法：

```cpp
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        // 优先队列，大根堆（默认）
        int n = nums.size();
        vector<int> ans;
        priority_queue<pair<int, int>> Q;   // 用 (nums[i], i) 作为一个 pair 来记录
        // 初始
        for(int i=0; i<k; i++)  Q.push({nums[i], i});
        ans.emplace_back(Q.top().first);
        for(int i=k; i<n; i++){
            // 如何判断当前最大值应该pop？看下标是否已经拉出了
            Q.push({nums[i], i});
            while(Q.top().second + k <= i)  Q.pop();   // 这里应该是while而不是if ！！！
            ans.emplace_back(Q.top().first);
        }
        return ans;
    }
};
```

时间复杂度：$O(n \log n)$其中插入一个元素进入队列需要 $\log n$的时间.

空间复杂度：$O(n+n)$.

优化算法，可以只用单调队列（要用双端队列，队头队尾都能插入删除，但人为保持单调特性）

```cpp
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        // 单调队列
        int n = nums.size();
        vector<int> ans;
        deque<int> Q;
        // 保持需要加入的才入队， i>a 且 nums[i] >= nums[a] 时，a就没必要了，pop除去a
        // 初始
        for(int i=0; i<k; i++){
            // 保持性质 1：在窗口内 （只需要看front，因为是按顺序进来的，肯定是前面的index更小）
            while(!Q.empty() && Q.front()+k<=i)  Q.pop_front();
            // 保持性质 2：末尾新加入的数字若比末尾数字大，则末尾数字退出（最终导致 整个队列递减）
            while(!Q.empty() && nums[i] >= nums[Q.back()])  Q.pop_back();
            // 现在可以插入了（插入的是比末尾数字更小的）
            Q.push_back(i);
        }
        // 取出队头就是当前最大的
        ans.emplace_back(nums[Q.front()]);
        // 更新
        for(int i=k; i<n; i++){
            while(!Q.empty() && Q.front()+k<=i)  Q.pop_front();
            while(!Q.empty() && nums[i] >= nums[Q.back()])  Q.pop_back();
            Q.push_back(i);
```

```
            ans.emplace_back(nums[Q.front()]);  // 每次都有一个ans
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n+k)$.

此题还可以使用“分块+预处理”，由于前面算法足够好，这里不写了。

参考：参考链接

时间复杂度：$O(n)$.

空间复杂度：$O(n+n)$.

{% endtabs %}

12. 最小覆盖子串

- **题面：**

  给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

  注意：

  - 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。
  - 如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

- **示例 1：**

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
解释：整个字符串 s 是最小覆盖子串。
```

示例 3:

```
输入：s = "a", t = "aa"
输出：""
解释：t 中两个字符 'a' 均应包含在 s 的子串中，
因此没有符合条件的子字符串，返回空字符串。
```

{% tabs 解法, -1 %}

1. 哈希表的数量检测不可避免
2. 滑动窗口，右指针 `r` 扩大，直到 `check true`。再左指针`l` 缩小，直到 `check false`。

```cpp
class Solution {
public:
    // 两个哈希表
    unordered_map<char, int> tmap, wmap;
    // 检测
    bool check(){
        for(auto &x : tmap)
            if(wmap[x.first] < x.second)
                return false;
        return true;
    }
    // 题解函数
    string minWindow(string s, string t) {
        int n = s.size();
        // 初始化 tmap
        for(auto &c : t)  tmap[c]++;
        // 左右两个指针，l缩小，r 扩大
        int l = 0, r = 0;
        int ansL = 0, len = n+1;
        // r 扩大，直到check true
        while(r < n){
            // 先本次记录
            wmap[s[r]]++;
            // l 缩小，直到check false
            while(check()){
                // 先记录当前 true
                if(r-l+1 < len)  len = r-l+1, ansL = l;
                // 再 l 缩小
                wmap[s[l]]--;
                l++;
            }
            // 再 r 扩大
            r++;
        }
        return len == n+1 ? "" : s.substr(ansL, len);
    }
};
```

时间复杂度：$O(C \cdot |s| + |t|)$，其中C是字符集大小，26或52.

空间复杂度：$O(C)$.

{% endtabs %}

五、普通数组

13. 最大子数组和

- **题面：**

  给你一个整数数组 nums ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

  子数组 是数组中的一个连续部分。

  进阶：如果你已经实现复杂度为 `O(n)` 的解法，尝试使用更为精妙的 **分治法** 求解。

- **示例 1：**

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

  示例 2：

```
输入：nums = [1]
输出：1
```

  示例 3：

```
输入：nums = [5,4,-1,7,8]
输出：23
```

{% tabs 解法, -1 %}

滑动窗口，自己想的

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // 滑动窗口
        int n = nums.size();
        int l = 0, r = 0;
        int maxS = nums[0], S = 0;
        while(r < n){
            S += nums[r];
            maxS = max(maxS, S);
            while(S < 0){
                S -= nums[l];
                l++;
            }
            if(l < r)  maxS = max(maxS, S);
            r++;
        }
        return maxS;
```

```
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

不用记录那么多，动态规划即可

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // 改进，动态规划求前序的最佳即可
        int pre = 0, ans = nums[0];
        for(int &num : nums){
            pre = max(pre + num, num);
            ans = max(ans, pre);
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

还可以有更加精妙的分治法，精妙在可以求任意区间。

参考：链接

时间复杂度：$O(n)$.

空间复杂度：$O(\log n)$.

{% endtabs %}

14. 合并区间

- **题面：**

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

- **示例 1：**

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

  示例 2：

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

{% tabs 解法, -1 %}

时间安排问题，都是先按照开始时间排序再合并。

```cpp
class Solution {
public:
    // 注意需要加上 static
    static bool compare(const vector<int> &a, const vector<int> &b){
        return a[0] < b[0];
    }
    // 题解函数
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        int n = intervals.size();
        // 先按 start 升序
        sort(intervals.begin(), intervals.end(), compare);
        // 那么依次从左到右融合即可
        vector<vector<int>> ans;
        ans.push_back(intervals[0]);
        for(int i=1; i<n; i++){
            // 可衔接时，扩并 (因为已经按照 start 升序，肯定可以直接选 ans 中最后一个来合并即可)
            if(intervals[i][0] <= ans.back()[1]){
                ans.back()[1] = max(ans.back()[1], intervals[i][1]);    // 注意选 max 的
            }
            // 不可衔接时，加入
            else{
                ans.push_back(intervals[i]);
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n \log n)$，主要是排序时间.

空间复杂度：$O(\log n)$，此处只统计额外空间，不含ans.

{% endtabs %}

15. 轮转数组

- **题面：**

  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。

  进阶：

尽可能想出更多的解决方案，至少有 三种 不同的方法可以解决这个问题。
你可以使用空间复杂度为 $O(1)$ 的 原地 算法解决这个问题吗?

- **示例 1:**

```
输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右轮转 1 步: [7,1,2,3,4,5,6]
向右轮转 2 步: [6,7,1,2,3,4,5]
向右轮转 3 步: [5,6,7,1,2,3,4]
```

示例 2:

```
输入：nums = [-1,-100,3,99], k = 2
输出：[3,99,-1,-100]
解释:
向右轮转 1 步: [99,-1,-100,3]
向右轮转 2 步: [3,99,-1,-100]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        // 先来个蠢办法，牺牲一下空间
        int n = nums.size();
        vector<int> rotate(n);
        // 使用模数
        for(int i=0; i<n; i++)
            rotate[(i+k) % n] = nums[i];
        // 移回数组
        nums.assign(rotate.begin(), rotate.end());
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        // 直接秒了
        k = k % nums.size();
        reverse(nums.begin(), nums.end());
        reverse(nums.begin(), nums.begin()+k);
        reverse(nums.begin()+k, nums.end());
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

16. 除自身以外数组的乘积

- **题面：**

  给你一个整数数组 `nums`，返回 数组 `answer` ，其中 `answer[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积 。

  题目数据 **保证** 数组 `nums`之中任意元素的全部前缀元素和后缀的乘积都在 **32 位** 整数范围内。

  请 **不要使用除法，**且在 `O(n)` 时间复杂度内完成此题。

进阶：你可以在 O(1) 的额外空间复杂度内完成这个题目吗？ （ 出于对空间复杂度分析的目的，输出数组 不被视为 额外空间。）

- **示例 1:**

```
输入: nums = [1,2,3,4]
输出: [24,12,8,6]
```

  示例 2:

```
输入: nums = [-1,1,0,-3,3]
输出: [0,0,9,0,0]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        // 先来个蠢办法
        vector<int> ans;
        int All = 1, zero = 0, n = nums.size(), pos = -1;
        for(int i=0; i<n; i++)
            if(nums[i] != 0) All *= nums[i];    // 注意 0
            else zero++, pos = i;
        // 有 0 存在
        if(zero >= 2)
            ans.assign(n, 0);
        // 只有一个 0 时，那么只有这个位置结果非 0
        else if(zero == 1){
            ans.assign(n, 0);
            ans[pos] = All;
        }
        // 无 0 时
```

```
        else{
            for(int &num : nums)
                ans.push_back(All / num);
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

但是违规了，题目要求不能除法。

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        // 不允许除法，那么我们记录前缀乘积，后缀乘积
        int n = nums.size();
        vector<int> L(n), R(n), ans;
        L[0] = 1, R[n-1] = 1;
        for(int i=1; i<n; i++)
            L[i] = L[i-1] * nums[i-1];
        for(int i=n-2; i>=0; i--)
            R[i] = R[i+1] * nums[i+1];
        // 结果 = 前缀 * 后缀
        for(int i=0; i<n; i++)
            ans.push_back(L[i] * R[i]);
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        // 更进一步，省空间
        int n = nums.size();
        // 1. 把ans当做 L 使用
        vector<int> ans(n);
        ans[0] = 1;
        for(int i=1; i<n; i++)
            ans[i] = ans[i-1] * nums[i-1];
        // 2. 后缀乘积用一个变量即可
        int R = 1;
        for(int i=n-1; i>=0; i--){
            ans[i] *= R;
            R *= nums[i];
        }
```

```
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

17. 缺失的第一个正数

- **题面：**

  给你一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。

  请你实现时间复杂度为 `O(n)` 并且只使用常数级别额外空间的解决方案。

- **示例 1：**

```
输入：nums = [1,2,0]
输出：3
解释：范围 [1,2] 中的数字都在数组中。
```

示例 2：

```
输入：nums = [3,4,-1,1]
输出：2
解释：1 在数组中，但 2 没有。
```

示例 3：

```
输入：nums = [7,8,9,11,12]
输出：1
解释：最小的正数 1 没有出现。
```

{% tabs 解法, -1 %}

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        // 当然，先违规做一下，排序就行，默认升序
        sort(nums.begin(), nums.end());
        int ans = 1, i = 0, n = nums.size();
        // 先到正数
        for(i=0; i<n; i++)
            if(nums[i] > 0)
                break;
        // 检验正数存在性
        for(; i<n; i++){
```

```
            if(nums[i] == ans)
                ans++;
            else if(nums[i] < ans)
                continue;
            else
                break;
        }
        return ans;
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        // 哈希表
        unordered_set<int> table;
        for(int &num : nums)
            table.insert(num);
        // 正数依次查找
        int ans = 1;
        while(table.find(ans) != table.end())
            ans++;
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        // 构造某种意义上的哈希表
        int n = nums.size();
        // 1. 把所有负数改为 N+1（答案只会是 [1, N+1] 的数）
        for(int &num : nums)  if(num <= 0)  num = n+1;
        // 2. 现在所有数字都是正数了，所以符号已经不用管了
        // 那我们把符号用来标记这个索引下标i+1表示的数字是否存在
        for(int &num : nums)
            if(abs(num) < n+1)  // 注意，num在动态变，因此需要加 abs 来判断
                nums[abs(num)-1] = - abs(nums[abs(num)-1]);    // 仔细体会，是把 num 看做下标
        // 3. 检索下标，第一个正数所在的下标，就是答案（下标+1 是答案，不是数字是答案）
        int idx = 0;
        for(; idx<n; idx++)
            if(nums[idx] > 0)  break;
```

```
        return idx+1;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        // 条件换位
        int n = nums.size();
        // 1. 把[1,N] 的数字放到对应下标 [0, N-1] 的位置上
        for(int i=0; i<n; i++){
            while(nums[i] >=1 && nums[i] <= n){
                int x = nums[i];
                if(nums[x-1] == x)
                    break;
                else
                    swap(nums[x-1], nums[i]);
            }
        }
        //  2. 那么下标与数字不对应的第一个，就是答案
        int idx = 0;
        for(; idx<n; idx++)
            if(nums[idx] != idx+1) break;
        return idx+1;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

18. 矩阵置零

- **题面：**

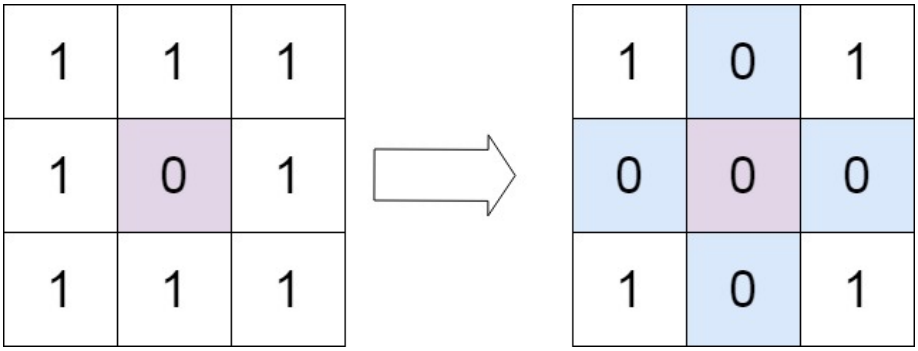
  给定一个 m x n 的矩阵，如果一个元素为 0 ，则将其所在行和列的所有元素都设为 0 。

  请使用 **原地** 算法。

  进阶：

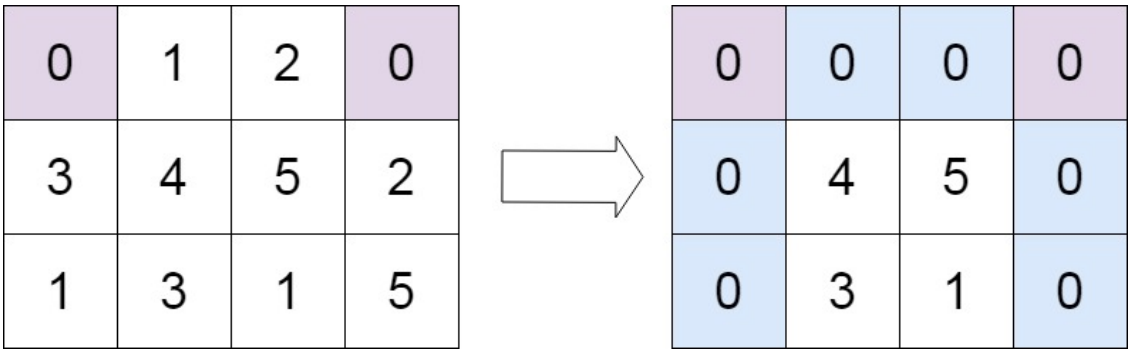
  - 一个直观的解决方案是使用 `O(m*n)` 的额外空间，但这并不是一个好的解决方案。
  - 一个简单的改进方案是使用 `O(m + n)` 的额外空间，但这仍然不是最好的解决方案。
  - 你能想出一个仅使用常量空间的解决方案吗?

- **示例 1：**




```
输入：matrix = [[1,1,1],[1,0,1],[1,1,1]]
输出：[[1,0,1],[0,0,0],[1,0,1]]
```

示例 2：




```
输入：matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
输出：[[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        // 开两个集合，用来存储需要改 0 的那些行、列
        int m = matrix.size(), n = matrix[0].size();
        set<int> row, column;
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
                if(matrix[i][j] == 0)  row.insert(i), column.insert(j);
        for(int r : row)  for(int j=0; j<n; j++)  matrix[r][j] = 0;
        for(int c : column)  for(int i=0; i<m; i++)  matrix[i][c] = 0;
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(m+n)$.

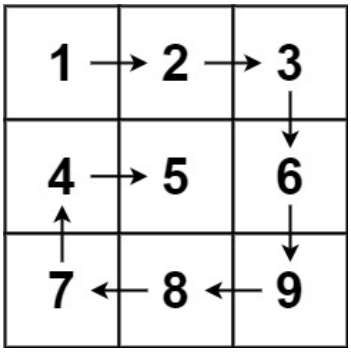
{% endtabs %}

19. 螺旋矩阵

- **题面：**

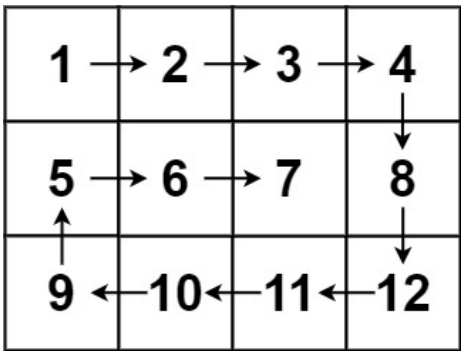
  给你一个 m 行 n 列的矩阵 matrix ， 请按照 顺时针螺旋顺序 ， 返回矩阵中的所有元素。

- **示例 1：**




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

示例 2：




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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size();
        int count = 0;
        // flag 指示当前的运动方向
        int i = 0, j = 0, flag = 1;
        // 分别记录行列的边界 ist ied 行开始 结束   jst jed 列开始 结束
        int ist = 0, ied = m, jst = 0, jed = n;
        vector<int> ans;
        while(count < m * n){
            ans.push_back(matrix[i][j]);
            switch(flag){
                // 从左到右
                case 1:
                    j++;
                    if(j >= jed)  j = jed-1, i++, flag = 2, ist++;
```

```
                    break;
                // 从上到下
                case 2:
                    i++;
                    if(i >= ied)  i = ied-1, j--, flag = 3, jed--;
                    break;
                // 从右到左
                case 3:
                    j--;
                    if(j < jst)  j = jst, i--, flag = 4, ied--;
                    break;
                // 从下到上
                case 4:
                    i--;
                    if(i < ist)  i = ist, j++, flag = 1, jst++;
                    break;
            }
            // 根据已输出的数量判断是否结束
            count++;
        }
        return ans;
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(mn+1)$.

{% endtabs %}

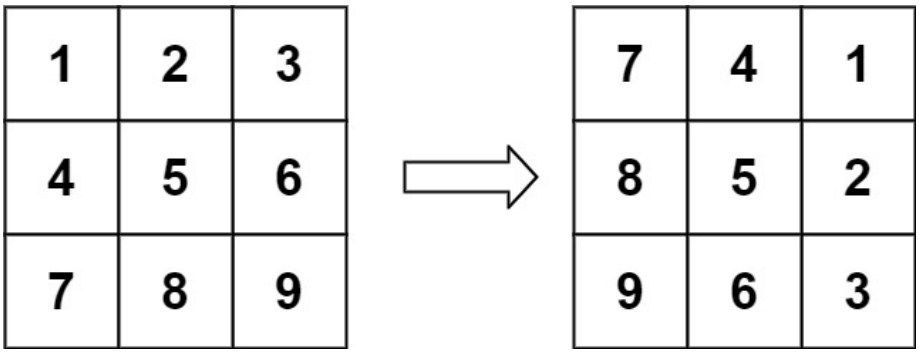
20. 旋转图像

- **题面：**

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

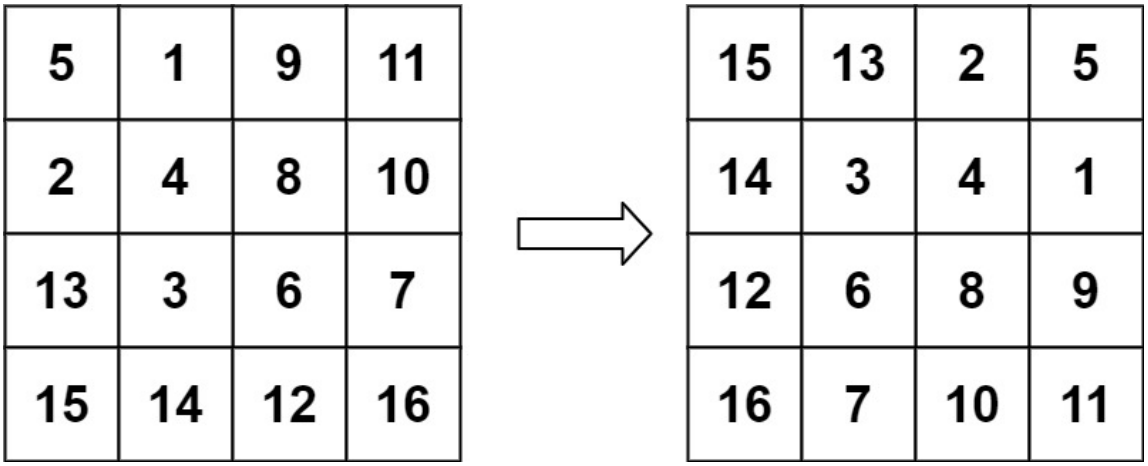
  你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

- **示例 1：**




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

示例 2：




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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        // 先来一个违规方法
        int n = matrix.size();
        vector<vector<int>> mr(n, vector<int>(n));
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                mr[j][n-i-1] = matrix[i][j];
        // 赋值回去
        matrix.assign(mr.begin(), mr.end());
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(n^2)$.

```cpp
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        // 每转一圈，是四个数字，只需要转整个矩阵 1/4 的左上角即可
        int n = matrix.size();
        for(int i=0; i<n/2; i++){
            for(int j=0; j<(n+1)/2; j++){
                // 本次四个位置轮转
                int x = matrix[i][j];
                matrix[i][j] = matrix[n-1-j][i];
                matrix[n-1-j][i] = matrix[n-1-i][n-1-j];
                matrix[n-1-i][n-1-j] = matrix[j][n-1-i];
                matrix[j][n-1-i] = x;
```

```
            }
        }
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

{% endtabs %}

21. 搜索二维矩阵 II

- **题面：**

  编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：

  - 每行的元素从左到右升序排列。
  - 每列的元素从上到下升序排列。

- 示例 1：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 5
输出：true
```

示例 2：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 20
输出：false
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        // 首先肯定暴力
        // 从左到右的列，从上到下的行
        int m = matrix.size(), n = matrix[0].size();
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
                if(matrix[i][j] == target)  return true;    // 找到
        // 退出循环则是没找到
        return false;
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 二分查找
    bool query(vector<int>& M, int target){
        int n = M.size();
        int L = -1, R = n;
        while(L+1 != R){
            int mid = L + R >> 1;   // 别写错了，除以 2 是右移 1 位
            if(M[mid] == target)  return true;
            if(M[mid] > target) R = mid;
```

```
            else L = mid;
        }
        return false;
    }

    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        // 再来二分法
        int m = matrix.size(), n = matrix[0].size();
        // 先找到正确的右边界
        int R = m-1;
        while(matrix[R][0] > target && R > 0)  R--;
        // 然后对每个数组采用二分查找
        for(int i=0; i<=R; i++)
            if(query(matrix[i], target))  return true;
        // 一直未找到
        return false;
    }
};
```

时间复杂度：$O(m \log n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        // 充分利用特性，z 字形查找
        // 当前状态下明确知道应该往哪边走，则走一步，有点像梯度下降
        int m = matrix.size(), n = matrix[0].size();
        int x = 0, y = n-1;
        while(x <= m-1 && y >= 0){
            // 找到
            if(matrix[x][y] == target)  return true;
            // 当前数字更大，明确知道同一行的偏左数字更小，因此向左走
            else if(matrix[x][y] > target)  y--;
            // 当前数字更小，明确知道同一列的偏下数字更大，因此向下走
            else x++;
        }
        // 没找到
        return false;
    }
};
```

时间复杂度：$O(m + n)$.

空间复杂度：$O(1)$.

{% endtabs %}

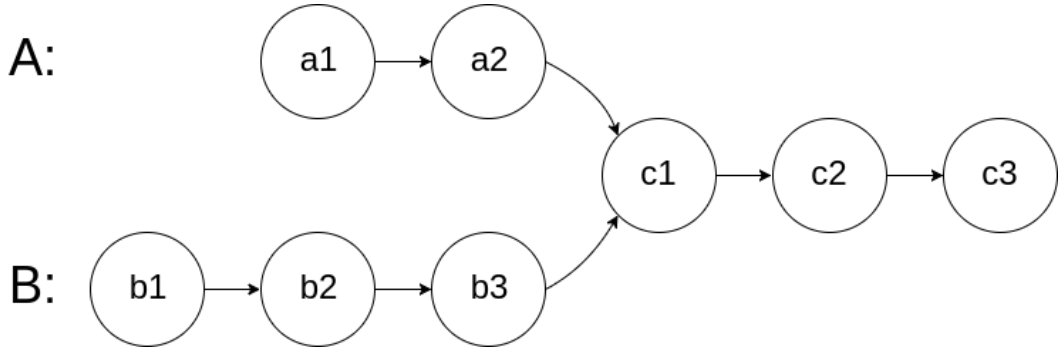
六、链表

22. 相交链表

- **题面：**

  给你两个单链表的头节点 headA 和 headB ， 请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null 。

  图示两个链表在节点 c1 开始相交：



  题目数据 保证 整个链式结构中不存在环。

  注意，函数返回结果后，链表必须 保持其原始结构 。

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        // 哈希表
        unordered_set<ListNode *> table;
        // 把 A 全存入
        ListNode *pa = headA;
        while(pa != NULL){
            table.insert(pa);
            pa = pa->next;
        }
        // 依次查找 B, 第一个找到的 pb 就是答案
        ListNode *pb = headB;
        while(pb != NULL){
            if(table.find(pb) != table.end())  return pb;
            else  pb = pb->next;
        }
        return NULL;
    }
};
```

时间复杂度：$O(m+n)$.

空间复杂度：$O(m)$.

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        // 双指针，走完自己的则走对方的
        ListNode *pa = headA, *pb = headB;
        // 若是有相交，则 pa 走了 a+c+b，pb 走了 b+c+a ，会相遇，相遇点就是答案
        // 若是无相交，则 pa，pb 会都把两个链表走完，然后到了 null
        // 无论如何，结束时，都是 pa == pb
        while(pa != pb){
            // pa 向后走，走完则走 B
            if(pa == NULL)  pa = headB;
            else  pa = pa->next;
            // pb 向后走，走完则走 A
            if(pb == NULL)  pb = headA;
            else  pb = pb->next;
        }
        // 无论什么情况，出来的时候，都就是答案
        return pa;
    }
};
```

时间复杂度：$O(m+n)$.

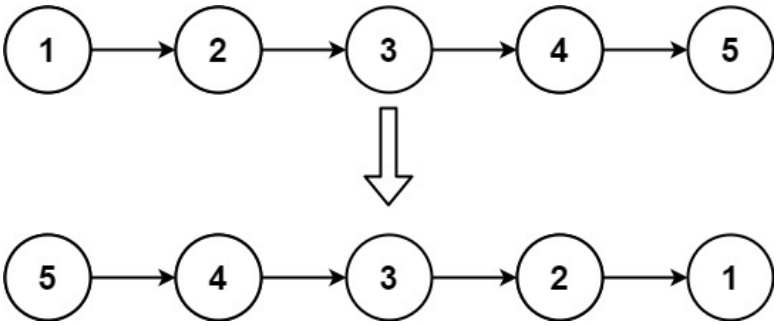
空间复杂度：$O(1)$.

{% endtabs %}

23. 反转链表

- **题面：**
  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 1：**



{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        // 改即可
        ListNode *pre = NULL, *cur = head;
        while(cur != NULL){
            // 当前节点的 next 改为前节点，要先把 正常走的下一个节点存储下来
            ListNode *t = cur->next;
            cur->next = pre;
            pre = cur;
            cur = t;
        }
        // 头节点是最后一个pre
        return pre;
    }
};
```

时间复杂度：$O(n)$.

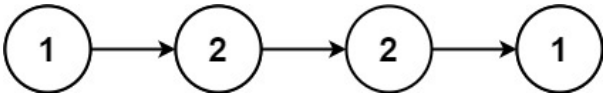
空间复杂度：$O(1)$.

{% endtabs %}

24. 回文链表

- **题面：**
  给你一个单链表的头节点 head ，请你判断该链表是否为回文链表。如果是，返回 true ；否则，返回 false 。

- 示例 1：



{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        // 1. 先复制到数组中
        vector<int> arr;
        ListNode *p = head;
        while(p != NULL)
            arr.emplace_back(p->val), p = p->next;
        // 2. 用两个指针，一头一尾的对比是不是一样的
        int l = 0, r = arr.size()-1;
        while(l < r)
            if(arr[l] == arr[r])  l++, r--;
            else  return false;
        return true;
    }
};
```

时间复杂度：$O(n)$.

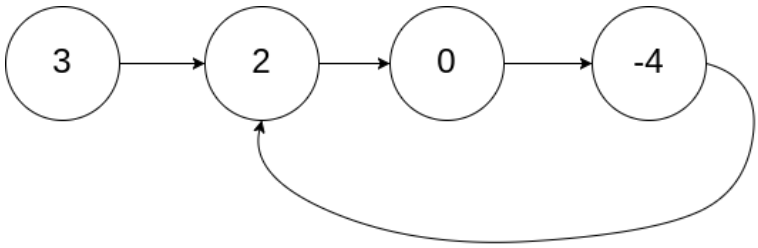
空间复杂度：$O(n)$.

{% endtabs %}

25. 环形链表

- **题面：**
  给你一个链表的头节点 head ，判断链表中是否有环。

  如果链表中存在环 ，则返回 true 。 否则，返回 false 。



{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        // 哈希表鉴别
        unordered_set<ListNode*> table;
        ListNode *p = head;
        while(p != NULL){
            // 若有重复的指针，说明有环
            if(table.find(p) != table.end())
                return true;
            table.insert(p);
            p = p->next;
        }
        // 出来则无环
        return false;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        // 龟兔指针，兔子会先进入环，乌龟也会慢慢进入环
```

```
        // 进入环之后，肯定兔子会碰到乌龟(进入环之后，龟兔的相对距离一直是以 1 递减的，肯定会相遇)
        ListNode *p1 = head, *p2 = head;
        // 特例，如果只有0或1个元素
        if(head == NULL)  return false;
        if(head->next == NULL)  return false;
        while(p1 != NULL && p2 != NULL){
            // p1 走一步，  p2 走两步
            p1 = p1->next;
            p2 = p2->next;
            if(p2)  p2 = p2->next;
            else  return false;
            if(p1 == p2)
                return true;
        }
        // 如若有环，是不会走出循环的；走出循环，则无环
        return false;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

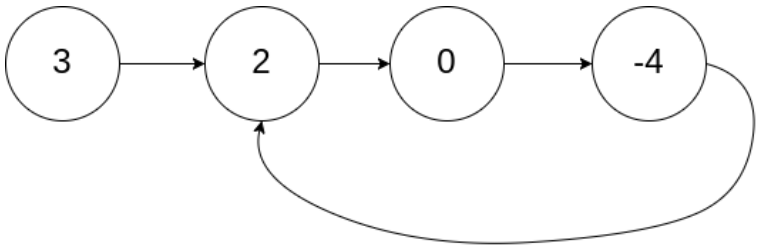
{% endtabs %}

26. 环形链表 II

- **题面：**

  给定一个链表的头节点 head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

  不允许修改 链表。



{% tabs 解法, -1 %}

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
```

```
public:
    ListNode *detectCycle(ListNode *head) {
        // 哈希表，第一个检测出来的就是环开始点
        unordered_set<ListNode*> table;
        ListNode *p = head;
        while(p != NULL){
            if(table.find(p) != table.end())
                return p;
            table.insert(p);
            p = p->next;
        }
        return NULL;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        // 有数量关系 兔子 = 2 乌龟
        // a + n(b+c) + b = 2 [ a + b ]
        // 则 a = (n-1)(b+c) + c
        // 因此，相遇后，再用 ptr 去从开头往后记录，那么乌龟会和ptr在入环点相遇
        ListNode *p1 = head, *p2 = head, *ptr = head;
        if(head == NULL || head->next == NULL)  return NULL;
        while(p1!=NULL && p2!=NULL){
            // p1走一步，p2走两步
            p1 = p1->next;
            p2 = p2->next;
            if(p2)  p2 = p2->next;
            else  return NULL;
            // 相遇了，ptr 开始走，和 p1 相遇就是入环点
            if(p1 == p2){
                while(ptr != p1)
                    ptr = ptr->next, p1 = p1->next;
                return ptr;
            }
        }
        return NULL;
    }
};
```

时间复杂度：$O(n)$.

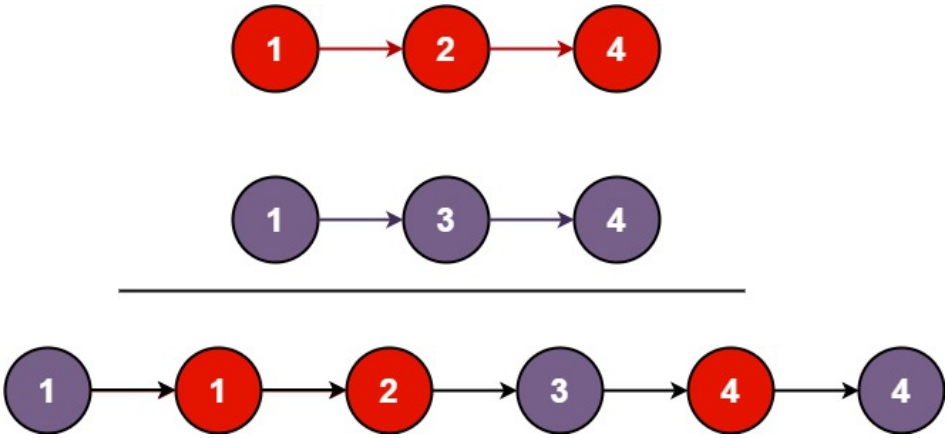
空间复杂度：$O(1)$.

{% endtabs %}

27. 合并两个有序链表

- **题面：**

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



{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // 新建 head 为开头，返回结果是 head.next
        ListNode *p1 = list1, *p2 = list2;
        ListNode head = ListNode(0), *pre = &head;
        while(p1 != NULL || p2 != NULL){
            // 若有空 NULL，则只需要接上，就结束了
            if(p1 == NULL){
                pre->next = p2;
                break;
            }
            else if(p2 == NULL){
                pre->next = p1;
                break;
```

```
            }
            // p2 更小，把 p2 拼接到 pre
            else if(p1->val >= p2->val){
                pre->next = p2;
                pre = p2;
                p2 = p2->next;
            }
            // p1 更小
            else{
                pre->next = p1;
                pre = p1;
                p1 = p1->next;
            }
        }
        // 结果就是
        return head.next;
    }
};
```

时间复杂度：$O(m+n)$.

空间复杂度：$O(1)$.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        // 递归法
        if(l1 == NULL)
            return l2;
        else if(l2 == NULL)
            return l1;
        else if(l1->val < l2->val){
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else{
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

时间复杂度：$O(m+n)$.

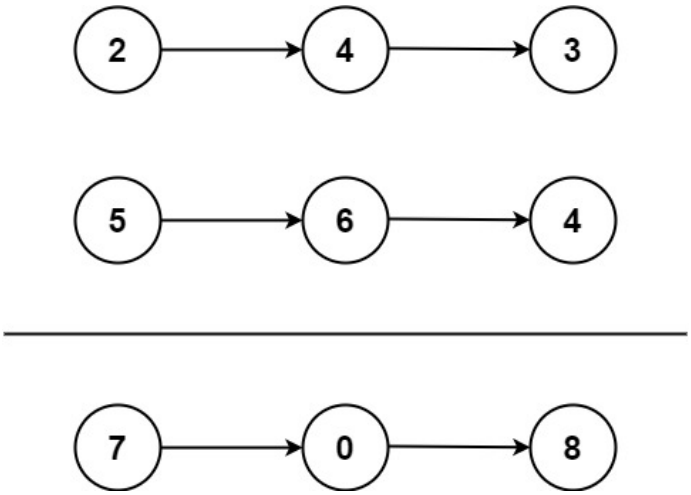
空间复杂度：$O(m+n)$.

{% endtabs %}

28. 两数相加

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

  请你将两个数相加，并以相同形式返回一个表示和的链表。
- **示例：**




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

{% tabs 解法, -1 %}

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 开一个 head 做开头
        ListNode *pre = new ListNode(0);
        ListNode *p1 = l1, *p2 = l2, *head = pre;
```

```
        int carry = 0;
        while(p1 != NULL || p2 != NULL){
            // 计算
            int v1 = p1 ? p1->val : 0;
            int v2 = p2 ? p2->val : 0;
            int sum = carry + v1 + v2;
            carry = sum / 10;
            sum = sum % 10;
            // 创建数据结构存储
            ListNode *data = new ListNode(sum);
            pre->next = data;
            pre = pre->next;
            // 下一位
            if(p1)  p1 = p1->next;
            if(p2)  p2 = p2->next;
        }
        // 还有进位，则需要再创建一位
        if(carry != 0){
            ListNode *tail = new ListNode(carry);
            pre->next = tail;
        }
        return head->next;
    }
};
```

时间复杂度：$O(m+n)$.

空间复杂度：$O(1)$.

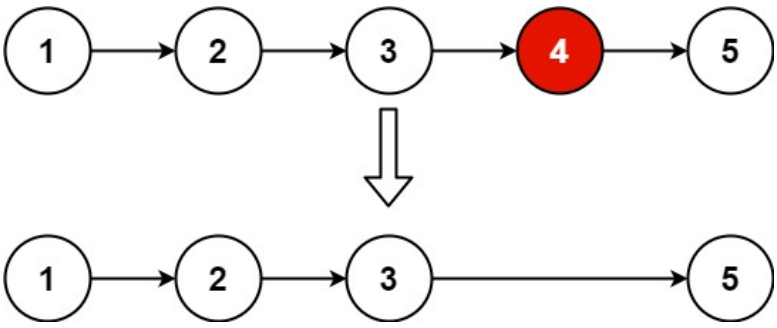
{% endtabs %}

29. 删除链表的倒数第 N 个结点

- **题面：**

  给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

- **示例：**




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

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // 先来蠢办法，先计算长度，就知道需删除的正序位置
        // 1. 计算长度
        ListNode *myhead = new ListNode(0);  // 创建一个不存储值的头指针，处理特殊情况会方便很多
        myhead->next = head;
        ListNode *p = myhead;
        int len = -1;
        while(p)  len++, p = p->next;
        // 2. 抵达位置
        int pos = len - n;
        ListNode *pre = myhead;
        p = head;
        while(pos) pos--, pre = p, p = p->next;
        // 3. 删除
        pre->next = p->next;
        delete p;
        // 返回答案
        return myhead->next;
    }
};
```

时间复杂度：$O(L)$.

空间复杂度：$O(1)$.

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // 使用栈，弹出时刚好是反序
```

```
        ListNode *dummy = new ListNode(0, head);
        stack<ListNode*> stk;
        ListNode *p = dummy;
        // 1. 入栈
        while(p)
            stk.push(p), p = p->next;
        // 2. 出栈
        for(int i=0; i<n; i++)
            stk.pop();
        // 3. 删除
        ListNode *pre = stk.top();
        pre->next = pre->next->next;
        return dummy->next;
    }
};
```

时间复杂度：$O(L)$.

空间复杂度：$O(L)$.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // 不预处理计算链表长度：
        // 两个指针，既然要找到倒数第 n 个
        //  1. 初始时，p2 就比 p1 早 n 步，那么两个都一步一步走
        ListNode *dummy = new ListNode(0, head);
        ListNode *p1 = dummy, *p2= head;    // 注：p1 要多留一步，从dummy开始
        for(int i=0; i<n; i++)
            p2 = p2->next;
        // 2. p2到末尾，p1就是删除的位置
        while(p2)
            p1 = p1->next, p2 = p2->next;
        // 3. 删除
        p1->next = p1->next->next;
        // 返回
        return dummy->next;
    }
};
```

时间复杂度：$O(L)$.

空间复杂度：$O(1)$.

{% endtabs %}

30. 两两交换链表中的节点

- **题面：**

  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）。

- **示例：**

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        // 从头到尾，依次数出两个，交换
        ListNode *dummy = new ListNode(0, head);    // 老规矩，先来一个空头
        ListNode *p1 = dummy, *p2 = head, *pre;
        // 先进入第一轮
        pre = p1;
        if(p1)  p1 = p1->next;
        if(p2)  p2 = p2->next;
        // 依次处理每一轮
        while(p1 != NULL && p2 != NULL){
            pre->next = p2;
            ListNode *t = p2->next;
            p2->next = p1;
            p1->next = t;
            pre = p1;
            // 再下一轮两个数字(注意，现在已经交换过位置了)
            if(p1)  p1 = p1->next;
            if(p1)  p2 = p1->next;
        }
        // 有 dummy 的好处就在于不需要单独处理开头 head 的特殊处理
        return dummy->next;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        // 递归，我笨脑子是想不出来的
        // 1. 只有 0 或 1 个则直接返回当前
        if(head == NULL || head->next == NULL)
            return head;
        // 2. 交换
        ListNode *newhead = head->next;
        head->next = swapPairs(newhead->next);
        newhead->next = head;
        // 3. 返回
        return newhead;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

31. K 个一组翻转链表

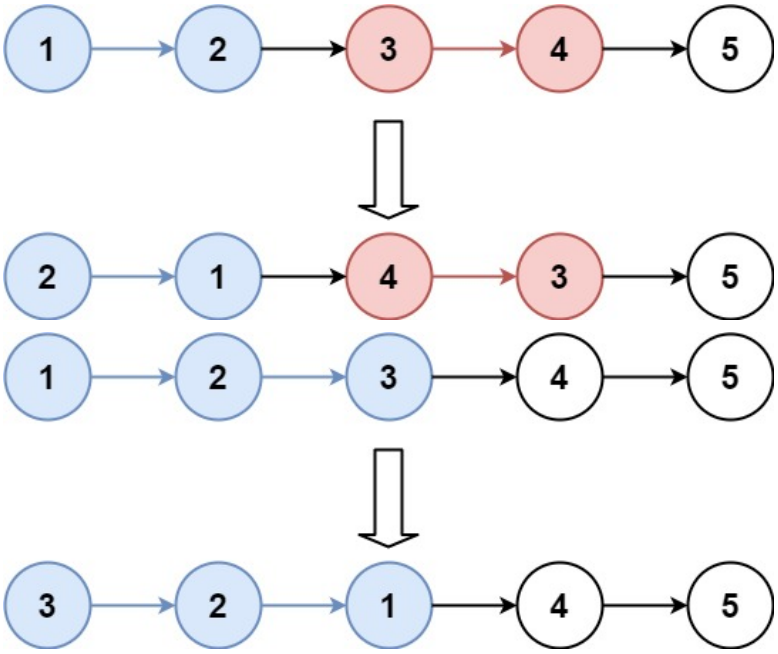
- **题面：**

  给你链表的头节点 head ，每 k 个节点一组进行翻转，请你返回修改后的链表。

  k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

  你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

- 示例：



{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        // 一眼用栈
        stack<ListNode*> stk;
        ListNode *dummy = new ListNode(0, head);
        ListNode *pre = dummy, *p = head;
        // 多轮交换，每轮交换 k 个
        while(p != NULL){
            int i = 0;
            // 压入 k 个
            for(i=0; i<k && p; i++){
                stk.push(p);
                p = p->next;
            }
            // 弹出 k 个
            if(i == k){
                for(i=0; i<k; i++){
                    ListNode *t = stk.top();    // 注意，这里必须是重新定义一个 t，不可沿用 p
```

```
                    pre->next = t;              // 因为前面的 p 刚好作为 while 的条件，不可动
                    pre = t;
                    stk.pop();
                }
                // 本轮结束，刚好可以用 pre 接上下一轮开头的 p
                pre->next = p;
            }
        }
        // 返回最终
        return dummy->next;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

32. 随机链表的复制

- **题面：**

  给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

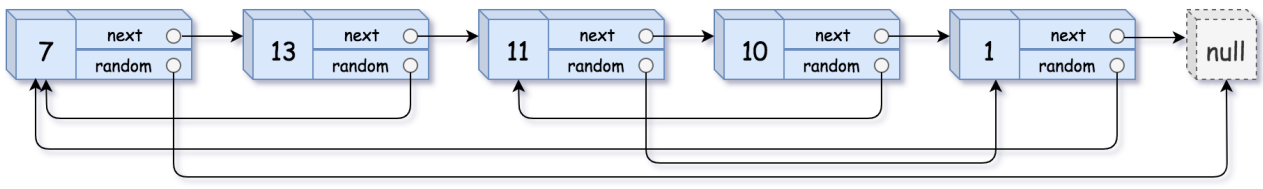
  构造这个链表的 **深拷贝**。 深拷贝应该正好由 n 个 全新 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。复制链表中的指针都不应指向原链表中的节点 。

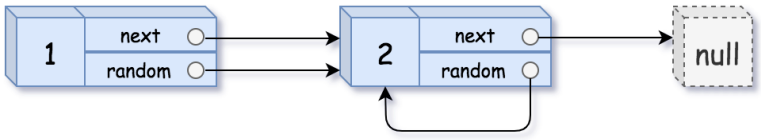
  例如，如果原链表中有 X 和 Y 两个节点，其中 X.random --> Y 。那么在复制链表中对应的两个节点 x 和 y ，同样有 x.random --> y 。如果不指向任何节点，则为 null 。

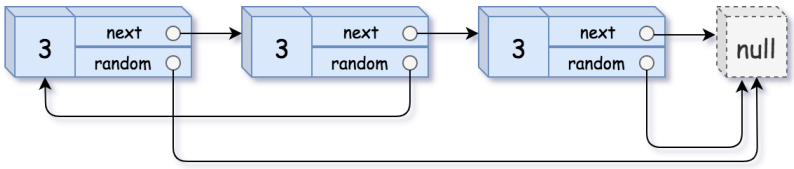
  返回复制链表的头节点。

  你的代码 只 接受原链表的头节点 head 作为传入参数。

- 示例：







{% tabs 解法, -1 %}

```cpp
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/

class Solution {
public:
    Node* copyRandomList(Node* head) {
        // 哈希表
        unordered_map<Node*, Node*> table;
        // 1. 创建整个新链表，并把每个节点对应的新节点存入哈希表
        Node *dummy = new Node(0);
        Node *p = head;
        Node *pre = dummy, *newp;
        while(p != NULL){
            // 创建
            newp = new Node(p->val);
            pre->next = newp;
            pre = newp;
            // 存入
            if(p)  table[p] = newp;  // 注意 null 不存
            // 下一个
            p = p->next;
        }
        // 2. 依次编写正确的 random
        p = head, newp = dummy->next;
        while(newp != NULL){
            // 注意寻找的是 p->random
            if(table.find(p->random) != table.end())
                newp->random = table[p->random];
            // 下一个
```

```
            p = p->next;
            newp = newp->next;
        }
        // 3. 返回
        return dummy->next;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

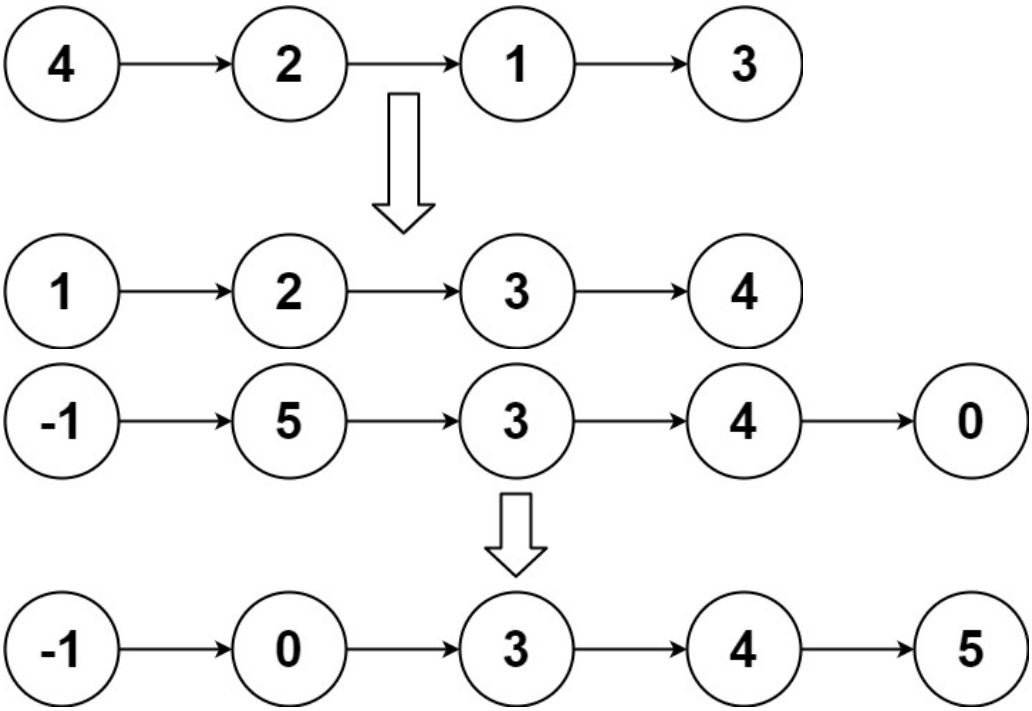
{% endtabs %}

33. 排序链表

- **题面：**

  给你链表的头结点 head ， 请将其按 升序 排列并返回 排序后的链表 。

- 示例：



{% tabs 解法, -1 %}

插入排序，超出时间限制。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
```

```
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        // 插入排序
        if(head == NULL || head->next == NULL)  return head;    // 只有 0/1 个元素
        ListNode *dummy = new ListNode(0, head);
        // 默认第一个元素是已排好序，从第二个开始，去已排好序的部分里面找到自己的位置，插入
        ListNode *cur = head->next;
        head->next = NULL;  // NULL 用来隔开已排序的部分
        // 开始插入排序
        while(cur){
            // 从头开始寻找已排序部分里面自己的位置，最多到已排序部分的 NULL 就会停下
            ListNode *pre = dummy;
            while(pre->next && pre->next->val < cur->val)
                pre = pre->next;
            // 插入
            ListNode *tmp = pre->next;  // 记录交换值
            ListNode *next = cur->next; // 记录下一轮待排序的数
            pre->next = cur;
            cur->next = tmp;
            // 下一个
            cur = next;
        }
        // 返回答案
        return dummy->next;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

归并排序，可以。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
```

```
        // 归并排序
        return sortSection(head, NULL);
    }
    // 给定区间 [ head, tail ) 排序，返回排序后的 head
    ListNode* sortSection(ListNode* head, ListNode* tail){
        // 1. 无元素、一个元素（递归结束条件）
        if(head == NULL)  return NULL;
        if(head->next == tail){ // 注意，并不排序 tail
            head->next = NULL;
            return head;
        }
        // 2. 找到中间点（快慢指针）
        ListNode *slow = head, *fast = head;
        while(fast != tail){    // 注意，由于我们有区间，不能用 NULL
            slow = slow->next;
            fast = fast->next;
            if(fast != tail)  fast = fast->next;
        }
        // 3. 分开排序
        ListNode *head1 = sortSection(head, slow);
        ListNode *head2 = sortSection(slow, tail);
        // 4. 归并
        ListNode *newhead = mergeSection(head1, head2);
        // 5. 返回
        return newhead;
    }
    // 归并两个已经排序的链表，返回新的链表头
    ListNode* mergeSection(ListNode *head1, ListNode *head2){
        ListNode *dummy = new ListNode(0);
        ListNode *p1 = head1, *p2 = head2, *pre = dummy;
        // 1. 这里可以使用 null 判断，因为排序后各个区间已经是使用 null 打散的
        while(p1 != NULL && p2 != NULL){
            if(p1->val < p2->val)
                pre->next = p1, pre = p1, p1 = p1->next;
            else
                pre->next = p2, pre = p2, p2 = p2->next;
        }
        // 2. 出循环后，没有走完的链表，就直接接上即可
        if(p1 != NULL)  pre->next = p1;
        if(p2 != NULL)  pre->next = p2;
        // 返回
        return dummy->next;
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(\log n)$.

注：若改为自底向上归并排序，还可以空间复杂度降为 $O(1)$，但是没必要。
参考：链接

耍赖也未尝不可！

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        // 当然，还可以偷懒
        // 1. 转成 vector
        vector<int> arr;
        ListNode *p = head;
        while(p){
            arr.push_back(p->val);
            p = p->next;
        }
        // 2. 排序
        sort(arr.begin(), arr.end());
        // 3. 填回去
        p = head;
        for(int num : arr){
            p->val = num;
            p = p->next;
        }
        return head;
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(n)$.

{% endtabs %}

34. 合并 K 个升序链表

- **题面：**

  给你一个链表数组，每个链表都已经按升序排列。

  请你将所有链表合并到一个升序链表中，返回合并后的链表。

- **示例：**

```
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
```

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        // K 次归并即可
        int k = lists.size();
        if(k == 0)  return NULL;
        ListNode *newhead = lists[0];
        for(int i=1; i<k; i++)
            newhead = mergeSection(newhead, lists[i]);
        // 返回
        return newhead;
    }
    // 归并两个已排序的链表，返回排序完成的新链表头
    ListNode* mergeSection(ListNode *head1, ListNode *head2){
        ListNode *dummy = new ListNode(0);
        ListNode *p1 = head1, *p2 = head2, *pre = dummy;
        // 选择小的连接
        while(p1 != NULL && p2 != NULL){
            if(p1->val < p2->val)
                pre->next = p1, pre= p1, p1 = p1->next;
            else
                pre->next = p2, pre= p2, p2 = p2->next;
        }
        // 未完成的直接连上
        if(p1 != NULL)  pre->next = p1;
        if(p2 != NULL)  pre->next = p2;
        // 返回
        return dummy->next;
```

```
    }
};
```

时间复杂度：$O(k^2n)$.

空间复杂度：$O(1)$.

耍赖未尝不可。

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        // 流氓还是有的
        if(lists.size() == 0)  return NULL;
        // 1. 转为 arr
        vector<int> arr;
        ListNode *pre = NULL, *first = NULL;
        for(ListNode* head : lists){
            // 并且直接把全部链表串起来
            if(head == NULL)
                continue;
            if(first)  pre->next = head;
            else  pre = first = head;
            // 本次读取
            ListNode *t = head;
            while(t != NULL){
                arr.push_back(t->val);
                pre = t;
                t = t->next;
            }
        }
        // 2. 排序
        sort(arr.begin(), arr.end());
        // 3. 填入
        ListNode *p = first;
        for(int num : arr){
            p->val = num;
            p = p->next;
        }
        // 返回
        return first;
```

```
    }
};
```

时间复杂度：$O(kn \log kn)$.

空间复杂度：$O(kn)$.

{% endtabs %}

35. LRU 缓存

- **题面：**

  请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。
  实现 `LRUCache` 类：
  `LRUCache(int capacity)` 以 正整数 作为容量 `capacity` 初始化 LRU 缓存
  `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1` 。
  `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value` ；如果不存在，则向缓存中插入该组 `key-value` 。如果插入操作导致关键字数量超过 `capacity` ，则应该 逐出 最久未使用的关键字。
  函数 `get` 和 `put` 必须以 `O(1)` 的平均时间复杂度运行。

- **示例：**

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

{% tabs 解法, -1 %}

```
// 本题需要设计数据结构
// 双向链表节点
struct DNode{
    // 数据
    int key, value;
    DNode *prev;
```

```
    DNode *next;
    // 构造
    DNode(): key(0), value(0), prev(NULL), next(NULL){}
    DNode(int _key, int _value): key(_key), value(_value), prev(NULL), next(NULL){}
};

// 解题
class LRUCache {
private:
    // 数据
    unordered_map<int, DNode*> cache_map;
    int capacity;
    int size;
    DNode *head;
    DNode *tail;

public:
    // 函数
    LRUCache(int _capacity): capacity(_capacity), size(0) {
        // 初始化两个空头尾，方便很多
        head = new DNode();
        tail = new DNode();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        // 1. 键不存在
        if(!cache_map.count(key))   // count 只返回 0 不存在，1 存在
            return -1;
        // 2. 键存在
        DNode *p = cache_map[key];
        // 添加到最近头
        removeNode(p);
        addToHead(p);
        // 返回值
        return p->value;
    }

    void put(int key, int value) {
        // 1. 键不存在
        if(!cache_map.count(key)){
            // 创建、写入、提到头部
            DNode *p = new DNode(key, value);
            cache_map[key] = p;
            addToHead(p);
            // 处理容量问题
            size++;
            if(size > capacity){
                // 删除尾部的
                DNode *t = tail->prev;  // 这里要暂存一下，因为后面操作了之后，会变动关系
                removeNode(t);
```

```
                cache_map.erase(t->key);
                delete t;    // 真实删除
                size--;
            }
        }
        // 2. 键存在
        else{
            // 直接改值即可，并提到头部
            DNode *p = cache_map[key];
            p->value = value;
            removeNode(p);
            addToHead(p);
        }
    }

    // 从链表中移除 p（注：并没有删除，只是从链表中移除）
    void removeNode(DNode *p){
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }
    // 添加 p 到最头部
    void addToHead(DNode *p){
        p->prev = head;
        p->next = head->next;
        head->next->prev = p;
        head->next = p;
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

时间复杂度：`put`和`get`均为$O(1)$.

空间复杂度：$O(capacity)$.

{% endtabs %}

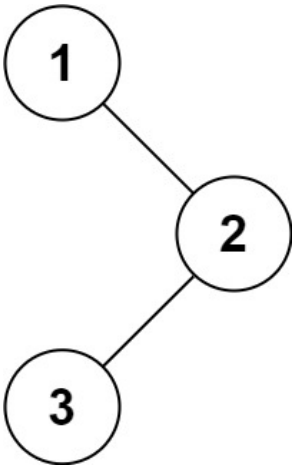
七、二叉树

36. 二叉树的中序遍历

- **题面：**

  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]
```

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 数据结构讲过，递归写一下即可
    vector<int> arr;
    vector<int> inorderTraversal(TreeNode* root) {
        if(root == NULL)
            return arr;
        inorderTraversal(root->left);
        arr.push_back(root->val);
        inorderTraversal(root->right);
        return arr;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

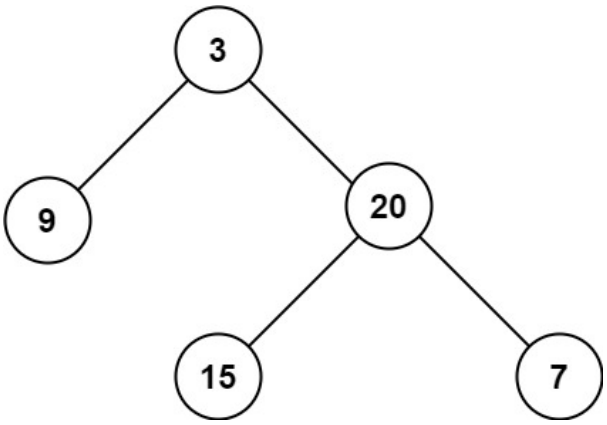
37. 二叉树的最大深度

- **题面：**

  给定一个二叉树 `root` ，返回其最大深度。

  二叉树的 最大深度 是指从根节点到最远叶子节点的最长路径上的节点数。

- **示例：**




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

{% tabs 解法, -1 %}

DFS

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 递归很简单(本质上是 深度优先遍历 DFS)
    int maxDepth(TreeNode* root) {
        // 结束条件
        if(root == NULL)
            return 0;
        // 分别找到左右子树的最大深度，+1 即可
        int lmax = maxDepth(root->left);
        int rmax = maxDepth(root->right);
        return max(lmax, rmax) + 1;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

BFS

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    // 广度优先遍历 BFS 也能做
    int maxDepth(TreeNode* root) {
        if(root == NULL)  return 0;
        queue<TreeNode*> Q;
        int ans = 0;
        Q.push(root);
        // 思想是做到严格按层计算
        while(!Q.empty()){
            int num = Q.size(); // 只记录本层数量
            while(num > 0){
                TreeNode *p = Q.front();
                Q.pop();
                if(p->left)  Q.push(p->left);
                if(p->right)  Q.push(p->right);
                num--;
            }
            // 本层完成，进入下一层，因此深度+1
            ans++;
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

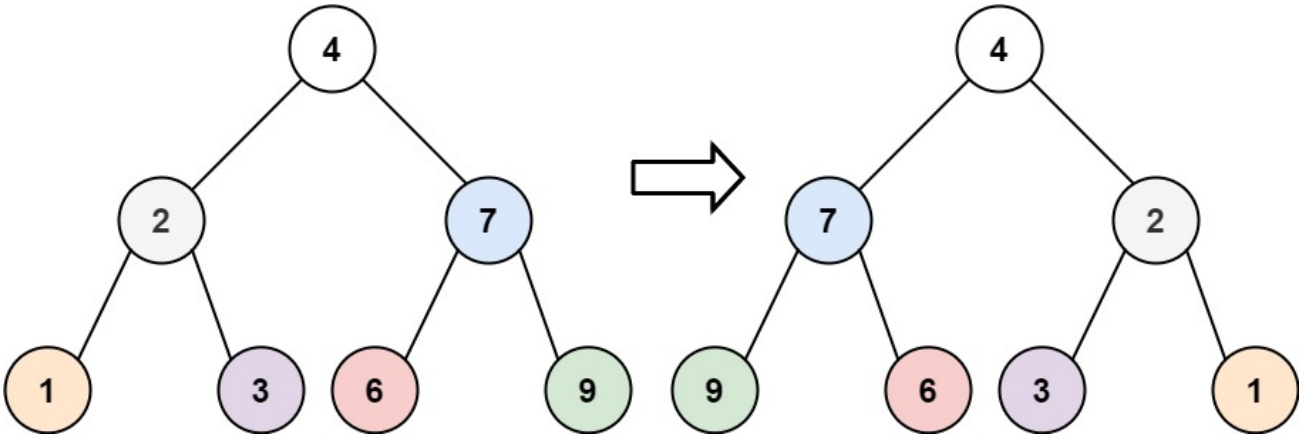
{% endtabs %}

38. 翻转二叉树

- **题面：**

  给你一棵二叉树的根节点 `root` ，翻转这棵二叉树，并返回其根节点。

- **示例：**



{% tabs 解法, -1 %}

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        // 直接递归改左右指针即可
        if(root == NULL)  return NULL;
        TreeNode *t = root->left;
        root->left = root->right;
        root->right = t;
        invertTree(root->left);
        invertTree(root->right);
        return root;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

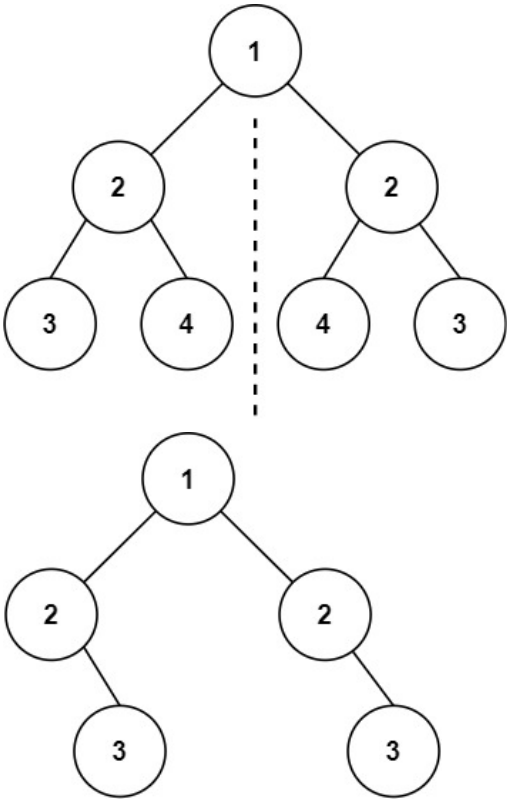
{% endtabs %}

39. 对称二叉树

- **题面：**

  给你一个二叉树的根节点 `root` ， 检查它是否轴对称。

- **示例：**



{% tabs 解法, -1 %}

自己想的使用栈的方法。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(root == NULL)  return true;
        // 左右两个指针同时向下
        queue<TreeNode*> LQ, RQ;
        if(root->left)  LQ.push(root->left);
```

```
        if(root->right)  RQ.push(root->right);
        // 按层收割
        while(LQ.size() == RQ.size() && !LQ.empty()){
            // 依次取出，比较相同
            TreeNode *l = LQ.front();  LQ.pop();
            TreeNode *r = RQ.front();  RQ.pop();
            if(l && r && l->val != r->val)
                return false;   // 注意，若是 l/r 均 null 也是 true
            // 压入子树(注意压入顺序，且会压入null)
            if(l)  LQ.push(l->left), LQ.push(l->right);
            if(r)  RQ.push(r->right), RQ.push(r->left);
        }
        // 退出循环时，看大小就知道是不是正常对称结束的
        if(LQ.size() != RQ.size())
            return false;
        return true;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

递归方法，复杂度是一样的，但是逻辑更巧妙。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return check(root->left, root->right);
    }
    // 递归两个指针向下找
    bool check(TreeNode *l, TreeNode *r){
        // 1. 均空
        if(!l && !r)
            return true;
        // 2. 一空一非空
        if(!l || !r)    // 这个逻辑式本质并不是表示 2，但是有 1 在前面保证 2
            return false;
        // 3. 均非空
```

```
        if(l->val == r->val)     // 注意下面的比较
            return check(l->left, r->right) && check(l->right, r->left);
        else
            return false;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

40. 二叉树的直径

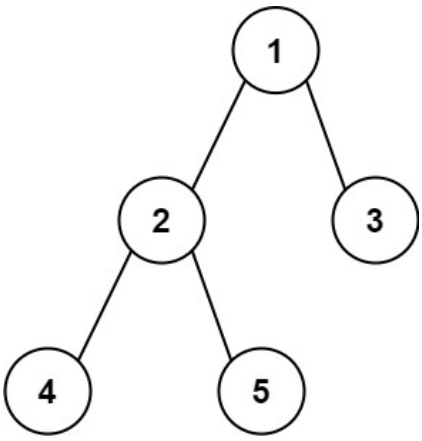
- **题面：**

  给你一棵二叉树的根节点，返回该树的 `直径` 。

  二叉树的 直径 是指树中任意两个节点之间最长路径的 `长度` 。这条路径可能经过也可能不经过根节点 `root` 。

  两节点之间路径的 `长度` 由它们之间边数表示。

- **示例：**




```
输入：root = [1,2,3,4,5]
输出：3
解释：3 , 取路径 [4,2,1,3] 或 [5,2,1,3] 的长度。
```

{% tabs 解法, -1 %}

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int ans = 0;
    int diameterOfBinaryTree(TreeNode* root) {
        // 在求深度的过程中来伴随更新最大直径
        depth(root);
        return ans;
    }
    // 求深度
    int depth(TreeNode *root){
        if(root == NULL)  return 0;
        int ldp = depth(root->left);
        int rdp = depth(root->right);
        // 每次求完，都算一下当前的最大直径
        ans = max(ans, ldp+rdp);
        return max(ldp, rdp) + 1;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(height)$.

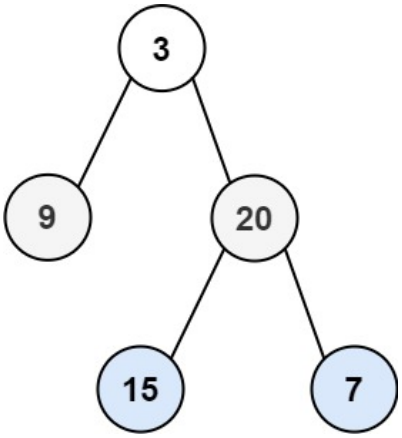
{% endtabs %}

41. 二叉树的层序遍历

- **题面：**

  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：




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

{% tabs 解法, -1 %}

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 层序遍历：用队列
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        queue<TreeNode*> Q;
        Q.push(root);
        int count = Q.size();    // 记录本层节点数量
        vector<int> tmp;
        while(!Q.empty()){
            // 节点操作
            TreeNode *node = Q.front();  Q.pop();
            if(node){
                tmp.push_back(node->val);
                Q.push(node->left);
                Q.push(node->right);
            }
            count--;
            // 分割层次
            if(count == 0 && tmp.size()){
                ans.push_back(tmp);
                tmp.clear();
                count = Q.size();
            }
        }
        // 返回
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
```

```
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 还是层序遍历，换个写法
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        queue<TreeNode*> Q;
        Q.push(root);
        while(!Q.empty()){
            // 本层遍历
            int count = Q.size();
            vector<int> tmp;
            for(int i=0; i<count; i++){
                TreeNode *node = Q.front();  Q.pop();
                if(node){
                    tmp.push_back(node->val);
                    Q.push(node->left);
                    Q.push(node->right);
                }
            }
            // 本层结束
            if(tmp.size())  ans.push_back(tmp);
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

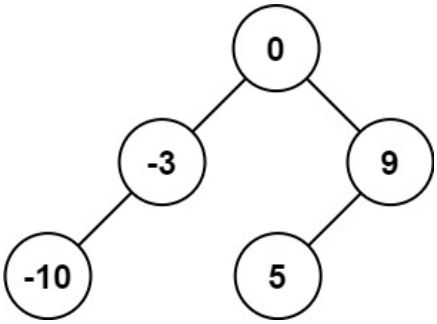
42. 将有序数组转换为二叉搜索树

- **题面：**

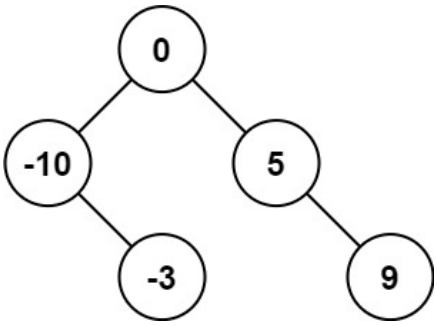
  给你一个整数数组 `nums` ，其中元素已经按 `升序` 排列，请你将其转换为一棵 **平衡** 二叉搜索树。

  `平衡二叉树` 是指该树所有节点的左右子树的深度相差不超过 1。

- 示例：



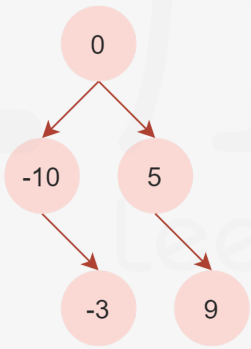

```
输入：nums = [-10,-3,0,5,9]
输出：[0,-3,9,-10,null,5]
解释：[0,-10,5,null,-3,null,9] 也将被视为正确答案：
```



{% tabs 解法, -1 %}

题解完整

总是选择中间位置左边的数字作为根节点 [-10, -3, 0, 5, 9]




```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
```

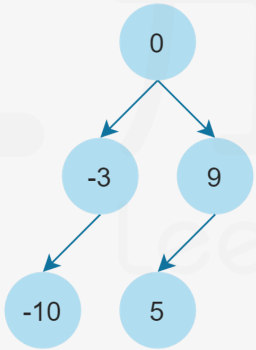
```
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    // 中序遍历，总是选择中间位置左边的数字作为根节点
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(0, nums.size()-1, nums);
    }
    // 根据区间来选数字 [l, r]
    TreeNode* helper(int l, int r, const vector<int>& nums){
        if(l > r)  return NULL;
        int mid = (l+r)/2;
        TreeNode *root = new TreeNode(nums[mid]);
        root->left = helper(l, mid-1, nums);
        root->right = helper(mid+1, r, nums);
        return root;
    }

};
```

时间复杂度：$O(n)$.

空间复杂度：$O(\log n)$, 主要是递归栈的深度.

总是选择中间位置右边的数字作为根节点 [-10, -3, 0, 5, 9]




```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 中序遍历，总是选择中间位置 右 边的数字作为根节点
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(0, nums.size()-1, nums);
    }
    // 根据区间来选数字 [l, r]
    TreeNode* helper(int l, int r, const vector<int>& nums){
        if(l > r)  return NULL;
        int mid = (l+r+1)/2;    // 就改这里一处即可
        TreeNode *root = new TreeNode(nums[mid]);
        root->left = helper(l, mid-1, nums);
        root->right = helper(mid+1, r, nums);
        return root;
    }

};
```

时间复杂度：$O(n)$.

空间复杂度：$O(\log n)$, 主要是递归栈的深度.

{% endtabs %}

43. 验证二叉搜索树

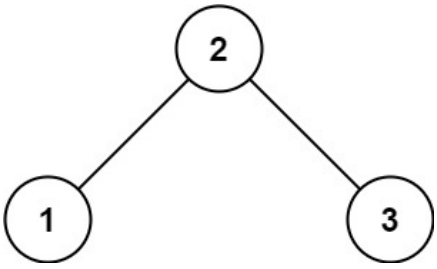
- **题面：**

  给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。

  有效`二叉搜索树`定义如下：

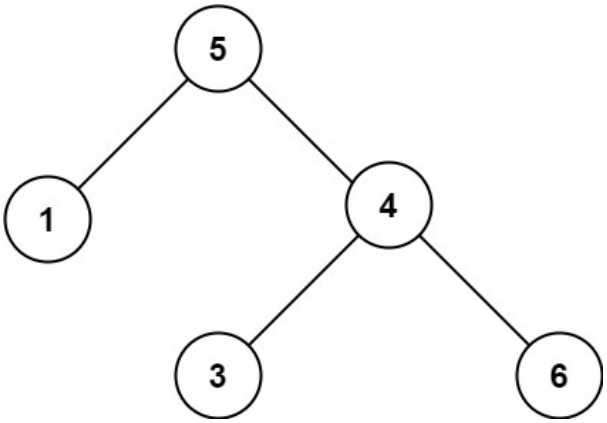
  - 节点的**左子树**只包含 **小于** 当前节点的数。
  - 节点的**右子树**只包含 **大于** 当前节点的数。
  - 所有左子树和右子树自身必须也是二叉搜索树。

- **示例1：**




```
输入：root = [2,1,3]
输出：true
```

示例2：




```
输入：root = [5,1,4,null,null,3,6]
输出：false
解释：根节点的值是 5 ，但是右子节点的值是 4 。
```

{% tabs 解法, -1 %}

自己想的，感觉通俗易懂。

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<int> arr;
    // 只需中序遍历一遍看是否严格递增即可
    bool isValidBST(TreeNode* root) {
        midSearch(root);
        return check();
    }
    // 中序遍历
    void midSearch(TreeNode* root){
        if(root == NULL)  return;
        midSearch(root->left);
        arr.push_back(root->val);
        midSearch(root->right);
    }
    // 检查是否严格递增
    bool check(){
```

```
        int n = arr.size();
        if(n == 0)  return true;
        int Max = arr[0];
        for(int i=1; i<n; i++){
            if(arr[i] == Max)  return false;    // 必须严格递增，不能相等
            Max = max(Max, arr[i]);
            if(arr[i] != Max)  return false;
        }
        return true;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

递归

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 递归
    bool isValidBST(TreeNode* root) {
        // 从最值开始,由于出现了边界值，那我们用更大的 LONG
        return helper(root, LONG_MIN, LONG_MAX);
    }
    // 判断数值 val 是否在 (low, high) 之间
    bool helper(TreeNode *root, long low, long high){
        if(root == NULL)  return true;
        long value = root->val;
        if(value >= high || value <= low)   // 注意也不能等于
            return false;
        // 左子树要比自己还小
        bool l = helper(root->left, low, value);
        bool r = helper(root->right, value, high);
        return l && r;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

用栈实现的中序遍历，不如我想的方法，复杂度一样。

但是**栈实现中序遍历**的思想要学会。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 还是中序遍历，换个写法，栈
    bool isValidBST(TreeNode* root) {
        stack<TreeNode*> stk;
        long premax = LONG_MIN;
        TreeNode *node = root;
        while(!stk.empty() || node != NULL){    // 思考为什么！因为每次当前节点是还未压入栈的。
            // 不断压入左子树
            while(node != NULL){
                stk.push(node);
                node = node->left;
            }
            // 取出节点
            node = stk.top();  stk.pop();
            long val = node->val;
            if(premax >= val)
                return false;
            premax = val;
            // 走向右子树，压入会在下一轮循环操作
            node = node->right;
        }
        return true;
    }

};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

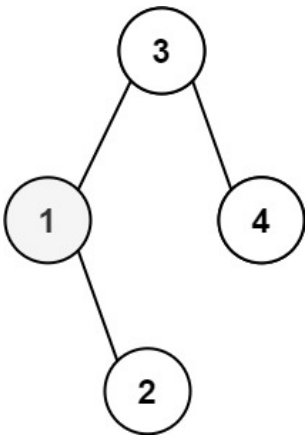
{% endtabs %}

44. 二叉搜索树中第 K 小的元素

- **题面：**

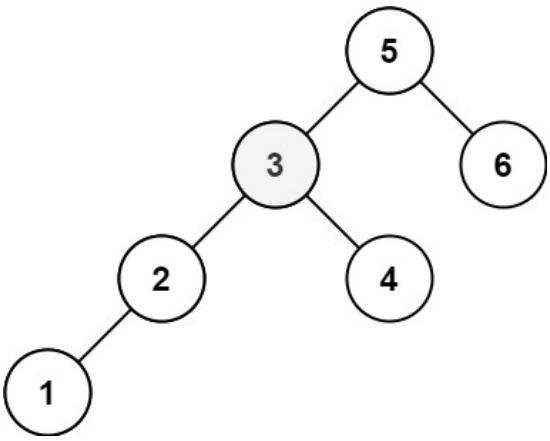
  给定一个二叉搜索树的根节点 `root` ，和一个整数 `k` ，请你设计一个算法查找其中第 `k` 小的元素（从 1 开始计数）。

- **示例 1：**




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

示例 2：




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

{% tabs 解法, -1 %}

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
```

```
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    vector<int> arr;
    int kthSmallest(TreeNode* root, int k) {
        // 1. 中序遍历
        inorderSearch(root);
        // 2. 取出第 k 小的（从 1 开始）
        return arr[k-1];
    }
    // 中序遍历出来
    void inorderSearch(TreeNode *root){
        if(root == NULL)  return;
        inorderSearch(root->left);
        arr.push_back(root->val);
        inorderSearch(root->right);
    }

};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        // 中序遍历：栈实现
        stack<TreeNode*> stk;
        TreeNode *node = root;
        int count = k;
        while(!stk.empty() || node != NULL){
            // 向左走
            while(node != NULL){
                stk.push(node);
```

```
                node = node->left;
            }
            // 弹出当前栈顶节点
            node = stk.top();  stk.pop();
            k--;
            if(k == 0)  break;
            // 向右走
            node = node->right;
        }
        return node->val;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

平衡二叉搜索树（AVL树）

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

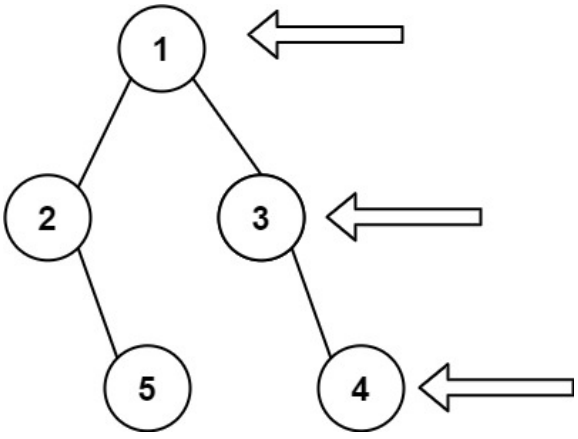
{% endtabs %}

45. 二叉树的右视图

- **题面：**

  给定一个二叉树的 根节点 `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

- **示例:**




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

{% tabs 解法, -1 %}

```
/**
```

```
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 也就是返回最右侧的那些节点（即，每层的最后一个节点）
    vector<int> rightSideView(TreeNode* root) {
        vector<int> ans;
        queue<TreeNode*> Q;
        if(root)  Q.push(root);
        while(!Q.empty()){
            int levelNum = Q.size();    // 本层节点数量
            TreeNode *node;
            for(int i=0; i<levelNum; i++){
                node = Q.front();  Q.pop();
                if(node->left)  Q.push(node->left);
                if(node->right)  Q.push(node->right);
            }
            ans.push_back(node->val);   // 只填写最后一个节点
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

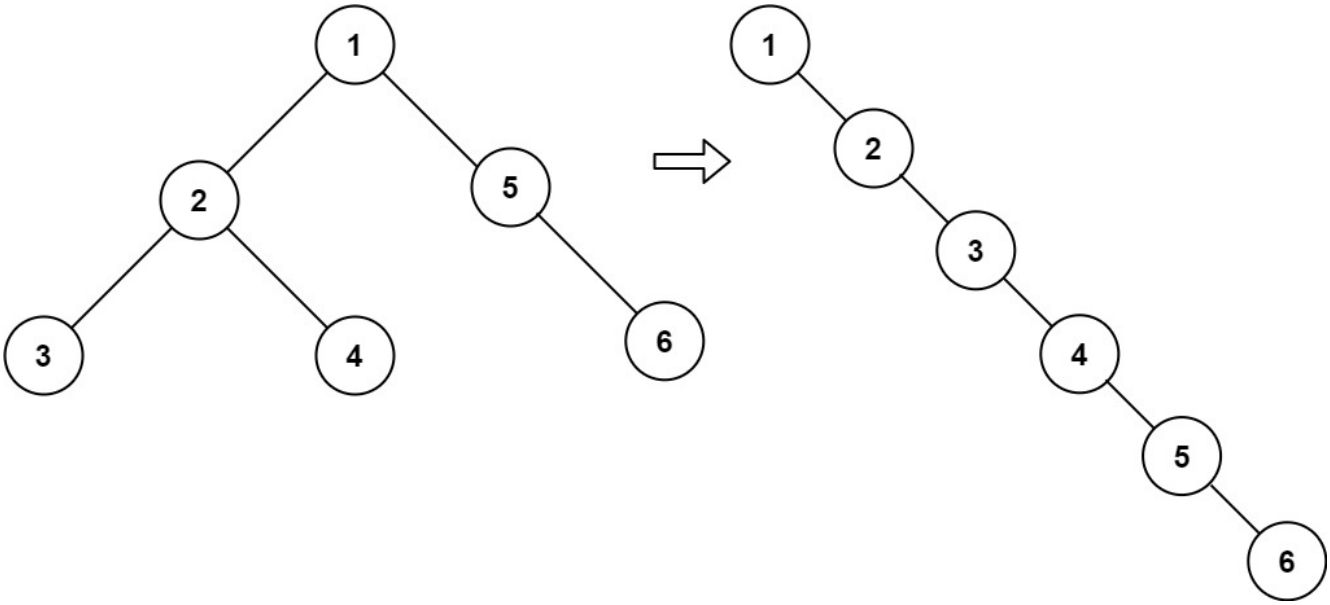
46. 二叉树展开为链表

- **题面：**

  给你二叉树的根结点 `root` ，请你将它展开为一个`单链表`：

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

- **示例：**




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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    void flatten(TreeNode* root) {
        helper(root, NULL);
    }
    // 递归：把以 root 为根的树拉平后，右侧再接 next
    TreeNode* helper(TreeNode* root, TreeNode* next){
        if(root == NULL)  return next;  // 已经没有了，那么返回给上层需要接的就是 next

        // 1. 拉平右子树，并连 next。再重新连接为 root 右子树
        root->right = helper(root->right, next);

        // 2. 拉平左子树，并连 right。再转为 root 右子树
        root->right = helper(root->left, root->right);
        root->left = NULL;

        // 3. 返回当前 root 用于上层的连接
        return root;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 依次改正一个一个
    void flatten(TreeNode* root) {
```

```
        if(root == NULL)  return;

        // 1. 有左子树，那么先改正这一个左子树到 right，并且保证改完后整个树还是连接的。
        if(root->left){
            // 左子树如果改到 right，需要找到最右节点来连接原来的 right
            TreeNode *p = root->left;
            while(p->right)  p = p->right;
            p->right = root->right;
            // 现在可以安心地把左子树改到 right 去了
            root->right = root->left;
            root->left = NULL;
        }

        // 2. 无论前面改没改，下一次，都直接进入下一个节点，即 root 的 right（可能是刚刚改过来的左子树）
        flatten(root->right);
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 直接记录前序遍历，然后依次改left right即可
    vector<TreeNode*> arr;
    void flatten(TreeNode* root) {
        if(root == NULL)  return;
        // 1. 先序遍历 记录下来
        preTraversal(root);

        // 2.依次更改指针
        int n = arr.size();
        for(int i=0; i<n-1; i++){
            arr[i]->left = NULL;
            arr[i]->right = arr[i+1];
        }
        arr[n-1]->left = NULL;
        arr[n-1]->right = NULL;
    }
    // 先序遍历
    void preTraversal(TreeNode *root){
        if(root == NULL)  return;
        arr.push_back(root);
        preTraversal(root->left);
        preTraversal(root->right);
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 先序遍历借用栈，就可以不使用记录了
    void flatten(TreeNode* root) {
        stack<TreeNode*> stk;
        if(root)  stk.push(root);
        TreeNode *pre = NULL;
        // 开始连接
        while(!stk.empty()){
            // 连接到当前的
            TreeNode *cur = stk.top();  stk.pop();
            if(pre != NULL){
                pre->left = NULL;
                pre->right = cur;
            }
            // 注意：先压right，因为后进先出
            if(cur->right)  stk.push(cur->right);
            if(cur->left)  stk.push(cur->left);
            // 更新 pre
            pre = cur;
        }
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

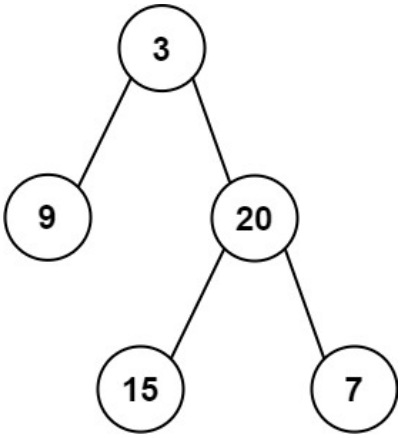
47. 从前序与中序遍历序列构造二叉树

- **题面：**

  给定两个整数数组 `preorder` 和 `inorder` ，其中 `preorder` 是二叉树的先序遍历， `inorder` 是同一棵树的中序遍历，请构造二叉树并返回其根节点。

  - `preorder` 和 `inorder` 均 无重复 元素
  - `inorder` 均出现在 `preorder`
  - `preorder` 保证 为二叉树的前序遍历序列
  - `inorder` 保证 为二叉树的中序遍历序列

- 示例：




```
输入: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
输出: [3,9,20,null,null,15,7]
```

{% tabs 解法, -1 %}

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = preorder.size();
        if(n == 0)  return NULL;

        // 栈 + 指针
        stack<TreeNode*> S;
        int I = 0;  // inorder 的索引指针
        TreeNode *root = new TreeNode(preorder[0]);
        S.push(root);

        for(int i=1; i<n; i++){
            int num = preorder[i];
            TreeNode *node = new TreeNode(num);

            // 当前节点是左子树
            if(S.top()->val != inorder[I])
                S.top()->left = node;

            // 当前节点是右子树
            else{
```

```
                TreeNode *parent = NULL;
                while(!S.empty() && S.top()->val == inorder[I]){
                    parent = S.top();
                    S.pop();
                    I++;
                }
                if(parent)  parent->right = node;
            }

            // 压入在最后，不能在前面，因为前面需要判断前一次是不是 inorder[I]
            S.push(node);
        }

        return root;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    // 哈希表：节点值->中序的索引
    unordered_map<int, int> index;

    // 递归 先序的下标[pl, pr]  中序的下标[il, ir]
    TreeNode* buildSubTree(vector<int>& preorder, vector<int>& inorder,
    int pl, int pr, int il, int ir){
        // 由于是采用下标，当前准备构建的就是 pl 节点
        if(pl > pr)  return NULL;

        // 根节点
        int InRoot = index[preorder[pl]];    // 中序所在的位置
        TreeNode *root = new TreeNode(preorder[pl]);

        // 左子树
        int LSize = InRoot - il;
        root->left = buildSubTree(preorder, inorder, pl+1, pl+LSize, il, InRoot-1);

```

```
        // 右子树
        root->right = buildSubTree(preorder, inorder, pl+LSize+1, pr, InRoot+1, ir);

        // 返回本次构建的节点
        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        // 构造哈希表
        int n = preorder.size();
        for(int i=0; i<n; i++)
            index[inorder[i]] = i;

        // 构建树
        return buildSubTree(preorder, inorder, 0, n-1, 0, n-1);
    }
};
```

时间复杂度：$O(n)$.

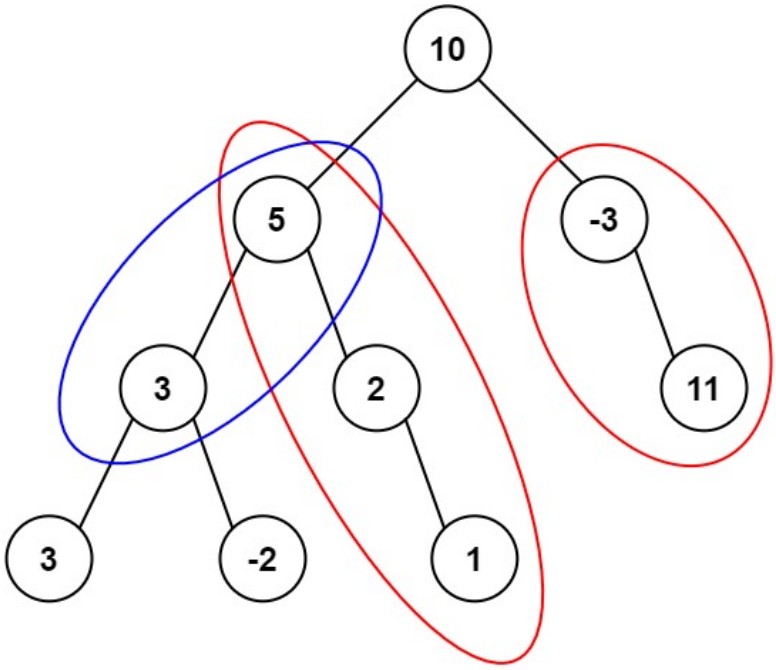
空间复杂度：$O(n)$.

{% endtabs %}

48. 路径总和 III

- **题面：**
  给定一个二叉树的根节点 `root` ，和一个整数 `targetSum` ，求该二叉树里节点值之和等于 `targetSum` 的 `路径` 的数目。

  `路径` 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。
- 示例：




```
输入：root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
输出：3
解释：和等于 8 的路径有 3 条，如图所示。
```

{% tabs 解法, -1 %}

单纯把所以可能都遍历一遍。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // DFS 计算一个当前节点为根向下可能有多少种
    int rootSum(TreeNode* p, long targetSum){   // 这里用 long
        if(!p)  return 0;
        int ret = 0;
        // 当前节点已经达到
        if(p->val == targetSum)  ret++;
        // 当前节点向左达到
        ret += rootSum(p->left, targetSum - p->val);
        // 当前节点向右达到
        ret += rootSum(p->right, targetSum - p->val);
```

```
        return ret;
    }
    // DFS 把所有节点都作为根，求出所有可能，加和
    int pathSum(TreeNode* root, int targetSum) {
        if(!root)  return 0;
        // 当前节点为根求出可能
        int ans = rootSum(root, targetSum);
        // 左右子树的所有可能
        ans += pathSum(root->left, targetSum);
        ans += pathSum(root->right, targetSum);
        return ans;
    }
};
```

时间复杂度：$O(N^2)$.

空间复杂度：$O(N)$.

借用哈希表，存储前面已经计算的结果。
每次借用前缀和出现的次数，来寻找本次能满足条件的数量。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 哈希表存储目前的前缀和数值的数量
    unordered_map<long long, int> prefix;

    // DFS 若有 前缀和 + targetSum = 当前前缀和，说明中间有路径到当前可以满足=targetSum
    int dfs(TreeNode* root, long long curr, int targetSum){
        if(!root)  return 0;
        int ret = 0;
        curr += root->val;
        // 看前缀和有多少个满足条件
        if(prefix.count(curr - targetSum))
            ret = prefix[curr - targetSum];
        // 进入更深
        prefix[curr]++;
        ret += dfs(root->left, curr, targetSum);
        ret += dfs(root->right, curr, targetSum);
        // 要回退了，因此把当前的 curr 值删去
```

```
        prefix[curr]--;
        // 返回
        return ret;
    }
    // 答案就是从根开始，一直向下dfs即可
    int pathSum(TreeNode* root, int targetSum) {
        int curr = 0;
        prefix[curr]++;  // 注意，最开始只有前缀和为 0
        return dfs(root, curr, targetSum);
    }
};
```

时间复杂度：$O(N)$.

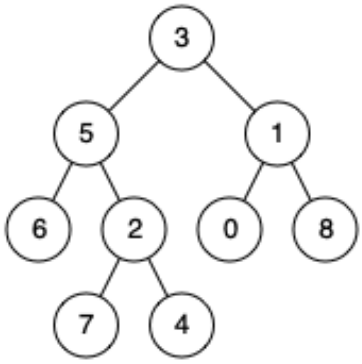
空间复杂度：$O(N)$.

{% endtabs %}

49. 二叉树的最近公共祖先

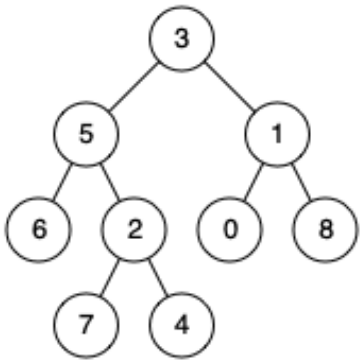
- **题面：**
  给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

  百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 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）。”
- **示例 1：**




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3 。
```

示例 2：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。
```

{% tabs 解法, -1 %}

找到判断条件。

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode *ans = NULL;
    // DFS 深度优先遍历
    bool dfs(TreeNode *root, TreeNode *p, TreeNode *q){
        if(root == NULL)  return false;
        bool lson = dfs(root->left, p, q);
        bool rson = dfs(root->right, p, q);
        bool self = root->val == p->val || root->val == q->val;
        // 答案在过程中:两种可能，1.在左右子树 2.自身是其中一个
        if( (lson && rson) || (self && (lson || rson)) )
            ans = root;
        // 返回当前树是否包含 p 或 q
        return lson || rson || self;
    }
    // 题目
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        dfs(root, p, q);
        return ans;
    }
};
```

时间复杂度：$O(N)$.

空间复杂度：$O(N)$.

哈希表存储父节点。

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    // 哈希表存储法，把父节点按照值存储下来，每个值可以直接查自己的父节点
    unordered_map<int, TreeNode*> father;
    // p依次把父节点全部访问一遍，q再访问，第一次遇到相同的，就是 LCA
    unordered_map<int, bool> visit;
    // 先 DFS 遍历把所有父节点记录下来
    void dfs(TreeNode *root){
        // 下面都会保证进入子树的dfs前自身都不是null,
        // 因此最初调用dfs要保证root不null即可
        if(root->left){
            father[root->left->val] = root;
            dfs(root->left);
        }
        if(root->right){
            father[root->right->val] = root;
            dfs(root->right);
        }
    }
    // 题目函数
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == NULL)  return NULL;
        // 记得存最初的根无父亲
        father[root->val] = NULL;
        dfs(root);  // 注意这里 root 最初不能是 null
        // 开始遍历
        TreeNode *t = p;
        while(t){
            visit[t->val] = true;
            t = father[t->val];
        }
        // 当 q 再向上遍历时，第一个公共的就是 LCA
        t = q;
        while(t){
            if(visit[t->val])  return t;
            t = father[t->val];
```

```
        }
        // 未找到
        return NULL;
    }
};
```

时间复杂度：$O(N)$.

空间复杂度：$O(N)$.

{% endtabs %}

50. 二叉树中的最大路径和

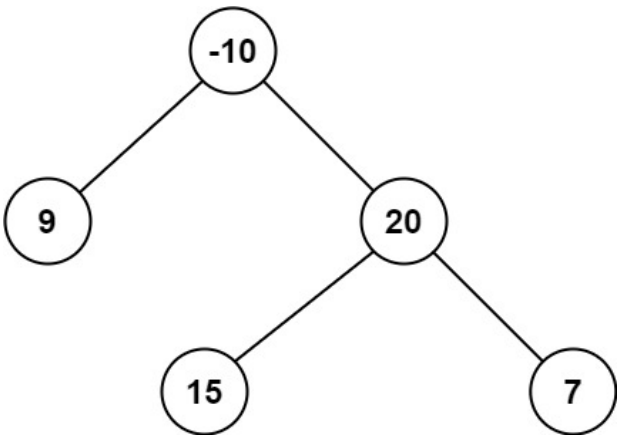
- **题面：**

  二叉树中的 `路径` 被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中 至多出现一次 。该路径 至少包含一个 节点，且不一定经过根节点。

  `路径和` 是路径中各节点值的总和。

  给你一个二叉树的根节点 `root` ，返回其 `最大路径和` 。

- **示例：**




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

{% tabs 解法, -1 %}
递归方法

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
```

```
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    // 答案
    int ans = INT_MIN;
    // 递归获取每个节点的最大单程贡献值（即从这个节点向下的最大贡献）
    int maxGain(TreeNode* root){
        if(root == NULL)  return 0;
        // 左右向下的最大可能值（若负，则不选，即 0）
        int lmax = max(maxGain(root->left), 0);
        int rmax = max(maxGain(root->right), 0);

        // 当前完全路径的最大可能值
        int curr = root->val + lmax + rmax;
        ans = max(ans, curr);

        // 返回当前的单程最大贡献
        return root->val + max(lmax, rmax);
    }
    // 题目
    int maxPathSum(TreeNode* root) {
        maxGain(root);
        return ans;
    }
};
```

时间复杂度：$O(N)$.

空间复杂度：$O(N)$.

{% endtabs %}

八、图论

51. 岛屿数量

- **题面：**
  给你一个由 `'1'`（陆地）和 `'0'`（水）组成的的二维网格，请你计算网格中岛屿的数量。

  岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

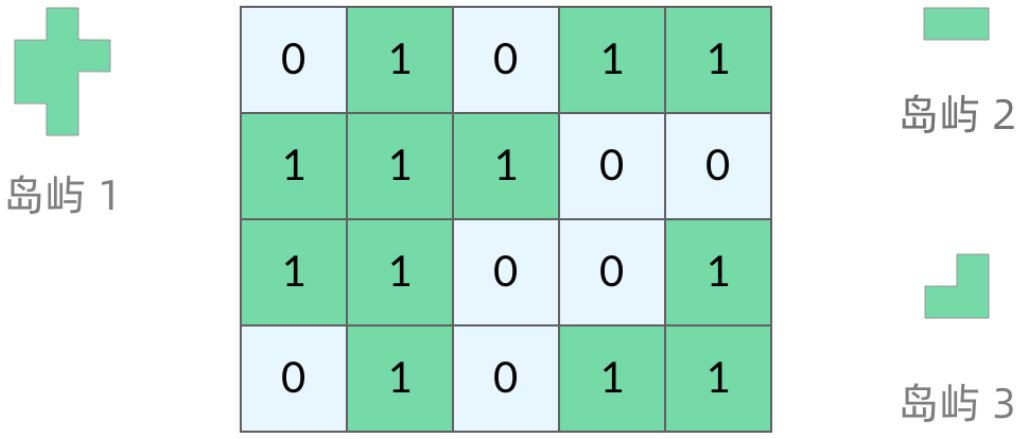
  此外，你可以假设该网格的四条边均被水包围。

- **示例 1：**

```
输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

示例 2：

```
输入：grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
输出：3
```



{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 网格问题通用做法（参考二叉树思想）
    void traverse(vector<vector<char>>& grid, int r, int c){
        // 判断 base case
        if(!inArea(grid, r, c))
            return ;

        // 是岛屿才继续访问四周
        if(grid[r][c]!='1')
            return;
        // 如何避免重复访问（用标记2）
        grid[r][c] = '2';

        // 依次遍历四周
        traverse(grid, r-1, c);
```

```
        traverse(grid, r+1, c);
        traverse(grid, r, c-1);
        traverse(grid, r, c+1);
    }
    // 是否在区域内
    bool inArea(vector<vector<char>>& grid, int r, int c){
        int R = grid.size(), C = grid[0].size();
        return 0<=r && r<R && 0<=c && c<C;
    }

    int numIslands(vector<vector<char>>& grid) {
        // 下面就把所有的情况都访问即可
        int ans = 0;
        int R = grid.size(), C = grid[0].size();
        for(int i=0; i<R; i++){
            for(int j=0; j<C; j++){
                if(grid[i][j]=='1'){
                    traverse(grid, i, j);
                    ans++;      // 完全遍历完这个空间，那就是一个岛屿
                }
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

52. 腐烂的橘子

- **题面：**
  在给定的 m x n 网格 grid 中，每个单元格可以有以下三个值之一：

  值 0 代表空单元格；

  值 1 代表新鲜橘子；

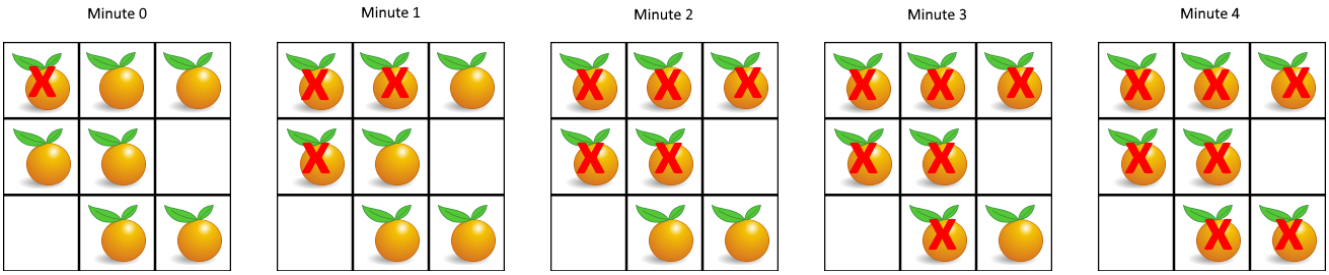
  值 2 代表腐烂的橘子。

  每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。

  返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1 。

- **示例 1：**

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



示例 2：

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

解释：左下角的橘子（第 2 行， 第 0 列）永远不会腐烂，因为腐烂只会发生在 4 个方向上。

示例 3：

```
输入：grid = [[0,2]]
输出：0
```

解释：因为 0 分钟时已经没有新鲜橘子了，所以答案就是 0 。

{% tabs 解法, -1 %}
我自己的想法，希望套用前面的框架。

```cpp
class Solution {
public:
    int curRot = 2; // 当前轮次的可感染标志（最初是标号 2 可以感染，第二轮是 3...）
    // 感染四周
    int Rot4(vector<vector<int>>& grid, int r, int c){
        // 判断 base case
        if(!inArea(grid, r, c))
            return 0;
        // 是本轮感染标号的，则感染四周
        if(grid[r][c]==curRot){
            int rotNum = 0;
            if(inArea(grid, r-1, c) && grid[r-1][c]==1)
                grid[r-1][c] = curRot+1, rotNum++;
            if(inArea(grid, r+1, c) && grid[r+1][c]==1)
                grid[r+1][c] = curRot+1, rotNum++;
            if(inArea(grid, r, c-1) && grid[r][c-1]==1)
                grid[r][c-1] = curRot+1, rotNum++;
            if(inArea(grid, r, c+1) && grid[r][c+1]==1)
                grid[r][c+1] = curRot+1, rotNum++;
            return rotNum;
        }
        return 0;
```

```
    }
    bool inArea(vector<vector<int>>& grid, int r, int c){
        int R = grid.size(), C = grid[0].size();
        return 0<=r && r<R && 0<=c && c<C;
    }
    // 感染一轮
    int RotEpoch(vector<vector<int>>& grid){
        int sum = 0;
        int R = grid.size(), C = grid[0].size();
        for(int i=0; i<R; i++){
            for(int j=0; j<C; j++){
                if(grid[i][j]==curRot)
                    sum += Rot4(grid, i, j);
            }
        }
        return sum;
    }
    // 检测是否全部腐烂
    bool allRotted(vector<vector<int>>& grid){
        int R = grid.size(), C = grid[0].size();
        for(int i=0; i<R; i++)
            for(int j=0; j<C; j++)
                if(grid[i][j]==1)  return false;
        return true;
    }

    int orangesRotting(vector<vector<int>>& grid) {
        int sum = -1;
        // 一轮一轮腐烂
        while(sum!=0){
            sum = RotEpoch(grid);
            curRot++;
        }
        // 检测最终
        if(allRotted(grid))
            return curRot-3;
        else
            return -1;
    }
};
```

时间复杂度：$O(k*mn)$.

空间复杂度：$O(1)$.

官方方法，使用了记录，要确保橘子数量不超过10*10大小

```
class Solution {
private:
    // 记录全部橘子各自的腐烂时间
    int badTime[10][10];
```

```
    int cnt = 0;    // 新鲜橘子数量
    // 方向向量
    int dirX[4] = {0, 0, 1, -1};
    int dirY[4] = {1, -1, 0, 0};

public:
    // 初始化 badTime 和 cnt， 生成一个初始的腐烂队列 Q
    void init(vector<vector<int>>& grid, queue<pair<int, int>>& Q){
        int R = grid.size(), C = grid[0].size();
        memset(badTime, -1, sizeof(badTime));   // 初始-1
        //（注意实际上是按字节填写的，只不过刚好无论是单字节/四字节，都刚好是-1）
        for(int i=0; i<R; i++){
            for(int j=0; j<C; j++){
                if(grid[i][j]==1)       // 新鲜橘子
                    cnt++;
                else if(grid[i][j]==2){ // 腐烂橘子
                    badTime[i][j] = 0;
                    Q.push({i, j});
                }
            }
        }
    }
    // 该位置是否是一个有效的新鲜橘子
    bool validGood(vector<vector<int>>& grid, int x, int y){
        int R = grid.size(), C = grid[0].size();
        return 0<=x && x<R && 0<=y && y<C && grid[x][y]==1 && badTime[x][y]==-1;
    }

    int orangesRotting(vector<vector<int>>& grid) {
        queue<pair<int, int>> Q;
        // 初始化变量
        init(grid, Q);
        // 依次取队列
        int ans = 0;
        while(!Q.empty()){
            // 取出一个腐烂橘子
            pair<int, int> pos = Q.front();
            Q.pop();
            int x = pos.first, y = pos.second;
            // 感染四周
            for(int t=0; t<4; t++){
                int tx = x + dirX[t], ty = y + dirY[t];
                if(validGood(grid, tx, ty)){
                    // 感染
                    badTime[tx][ty] = badTime[x][y] + 1;
                    cnt--;
                    Q.push({tx, ty});
                    ans = badTime[tx][ty];
                }
            }
        }
        return cnt==0? ans : -1;
```

```
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(k)$.

{% endtabs %}

53. 课程表

- **题面：**
  你这个学期必须选修 numCourses 门课程，记为 0 到 numCourses - 1 。

  在选修某些课程之前需要一些先修课程。 先修课程按数组 prerequisites 给出，其中 prerequisites[i] = [ai, bi] ，表示如果要学习课程 ai 则 必须 先学习课程 bi 。

  例如，先修课程对 [0, 1] 表示：想要学习课程 0 ，你需要先完成课程 1 。

  请你判断是否可能完成所有课程的学习？如果可以，返回 true ；否则，返回 false 。

- **示例 1：**

  ```
  输入：numCourses = 2, prerequisites = [[1,0]]
  输出：true
  ```

  解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0 。这是可能的。

  示例 2：

  ```
  输入：numCourses = 2, prerequisites = [[1,0],[0,1]]
  输出：false
  ```

  解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0 ；并且学习课程 0 之前，你还应先完成课程 1 。这是不可能的。

{% tabs 解法, -1 %}

```
class Solution {
private:
    vector<vector<int>> preCourse;  // 先修课程
    vector<int> state;              // 修读状态（0 未修读，1 正在准备修读，2 修读完成）
    bool flag = true;               // 是否能完成

public:
    // 深度优先去修读这个课程（把先修课程修完，然后修读本课程）
    void study(int course){
        // 进入正在修读状态
```

```
        state[course] = 1;
        // 把所有先修课程修读完毕
        for(int i=0; i<preCourse[course].size(); i++){
            int pre = preCourse[course][i];
            if(state[pre]==0)       // 未修读，现在就去先修读
                study(pre);
            else if(state[pre]==1)  // 先修课程也正在修读，说明冲突了
                flag = false;
            // 每次都看是否已经失败了，失败提前退
            if(flag == false)
                return;
        }
        // 先修课程都修完了，现在修读本课程，修读完毕
        state[course] = 2;
    }


    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        // 初始化
        preCourse.resize(numCourses);
        state.resize(numCourses);
        for(int i=0; i<prerequisites.size(); i++){
            int u = prerequisites[i][0], v = prerequisites[i][1];
            preCourse[v].push_back(u);  // 修读 v 需要先修 u
        }
        // 开始依次学习
        for(int i=0; i<numCourses; i++){
            if(!state[i]){
                study(i);
            }
        }
        // 返回结果
        return flag;
    }
};
```

时间复杂度：$O(n+m)$, n课程数, m先修课程对.

空间复杂度：$O(n+m)$.

{% endtabs %}

54. 实现 Trie (前缀树)

- **题面：**
  Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补全和拼写检查。

  请你实现 Trie 类：

  - `Trie()` 初始化前缀树对象。

- `void insert(String word)` 向前缀树中插入字符串 `word` 。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true` （即，在检索之前已经插入）；否则，返回 `false` 。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix` ，返回 `true` ；否则，返回 `false` 。

- **示例：**

```
输入
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
输出
[null, null, true, false, true, null, true]
解释
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // 返回 True
trie.search("app");     // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app");     // 返回 True
```

{% tabs 解法, -1 %}

```cpp
class Trie {
private:
    // 这里是自动补全、拼写检查的那种功能，思考一下
    // 设计字典树：每一个节点是一个有 26 个空指针的数组，其中哪个不空，就意为那个字符(a-z)
    // (但是并不存储字符，而是存储下一个字母的指针，本身字符是什么，由哪个不空来表示)
    vector<Trie*> next;
    bool isEnd;          // 表示到当前字符，是否结束
    //(即便结束，也还有可以指向后续的，表示多种字符串，不矛盾)

    // 寻找匹配前缀的节点
    Trie* searchPrefix(string word){
        Trie* node = this;  // 从最开始
        for(char ch : word){
            ch -= 'a';
            if(node->next[ch] == NULL)
                return NULL;
            else
                node = node->next[ch];
        }
        // 返回完全匹配前缀的最后一个字符
        return node;
    }

public:
    // 初始化，每个节点是一个 26位空指针 的数组
    Trie() : next(26), isEnd(false) {
```

```
27
28      }
29
30      void insert(string word) {
31          // 从最初开始插入
32          Trie* node = this;
33          for(char ch : word){
34              ch -= 'a';
35              if(node->next[ch] == NULL){
36                  node->next[ch] = new Trie();
37              }
38              node = node->next[ch];
39          }
40          // 到最后一个字符，可以结束
41          node->isEnd = true;
42      }
43
44      bool search(string word) {
45          // 找到完全匹配的前缀，且返回的最后一个字符是结束的
46          Trie* node = this->searchPrefix(word);
47          return node!=NULL && node->isEnd;
48      }
49
50      bool startsWith(string prefix) {
51          // 找到完全匹配的前缀存在即可
52          Trie* node = this->searchPrefix(prefix);
53          return node!=NULL;
54      }
55  };
56
57  /**
58   * Your Trie object will be instantiated and called as such:
59   * Trie* obj = new Trie();
60   * obj->insert(word);
61   * bool param_2 = obj->search(word);
62   * bool param_3 = obj->startsWith(prefix);
63   */
```

时间复杂度：$O(1), O(|S|)$.

空间复杂度：$O(|T| \cdot \sum)$, 其中$|T|$是所有字符串的总长度.

{% endtabs %}

九、回溯

55. 全排列

- **题面：**
  给定一个不含重复数字的数组 nums ， 返回其 所有可能的全排列 。你可以 按任意顺序 返回答案。

- **示例 1：**

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

示例 2：

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

示例 3：

```
输入：nums = [1]
输出：[[1]]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
// 回溯法：按照一定顺序尽可能探索出所有的解
private:
    vector<vector<int>> ans;
    vector<bool> visit;
    vector<int> output;

public:
    // 回溯：把所有的可能情况按顺序依次填入0-n 个空格中，每次回溯时，记得维护访问
    void backtrace(vector<int>& nums, int s, int e){
        // 若已经完成一个完整的组合
        if(s == e){
            ans.emplace_back(output);
            return;
        }
        // 还未完成，那么把可选的都放入当前位置做一遍
        for(int i=0; i<e; i++){
            if(!visit[i]){
                output[s] = nums[i];
                visit[i] = true;
                backtrace(nums, s+1, e);
                visit[i] = false;
            }
        }
        return;
    }

    vector<vector<int>> permute(vector<int>& nums) {
        visit.resize(nums.size());
        output.resize(nums.size());
        backtrace(nums, 0, nums.size());
        return ans;
    }
```

```
};
```

时间复杂度：$O(n*n!)$.

空间复杂度：$O(n)$.

{% endtabs %}

56. 子集

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

  解集 `不能` 包含重复的子集。你可以按 任意顺序 返回解集。

- **示例 1：**

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

示例 2：

```
输入：nums = [0]
输出：[[],[0]]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> t;

public:
    // 采用 二进制数字 mask 刚好对应 0/1 序列 选/不选（数字逻辑的内容）
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int MaxMask = 1 << n;    // 即 2^n
        for(int mask=0; mask<MaxMask; mask++){
            // 每个 mask 对应一个完整的序列，按位来查看是否选用
            t.clear();
            for(int i=0; i<n; i++)
                if(mask & (1<<i))  t.emplace_back(nums[i]);
            // 完成一次
            ans.emplace_back(t);
        }
        // 遍历结束即可
        return ans;
    }
};
```

时间复杂度：$O(n*2^n)$,一共2^n个答案，每个答案需要n次判断选/不选.

空间复杂度：$O(n)$.

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> t;

public:
    // 关于选/不选，也可以采用深度优先遍历的过程中来完成
    void dfs(int cur, vector<int>& nums){
        // 已经完成一次完整的序列
        if(cur == nums.size()){
            ans.emplace_back(t);
            return;
        }
        // 按照是否选用当前i所在字符
        // 选：先选入，再进入下一个的判断
        t.push_back(nums[cur]);
        dfs(cur+1, nums);
        t.pop_back();   // 恢复
        // 不选：不选，直接进入下一个的判断
        dfs(cur+1, nums);
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        dfs(0, nums);
        return ans;
    }
};
```

时间复杂度：$O(n * 2^n)$.

空间复杂度：$O(n)$.

{% endtabs %}

57. 电话号码的字母组合

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

  给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

- **示例 1：**

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

示例 2：

```
输入：digits = ""
输出：[]
```

示例 3：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    // 哈希表存储
    unordered_map<char, string> map{
        {'2', "abc"},
        {'3', "def"},
        {'4', "ghi"},
        {'5', "jkl"},
        {'6', "mno"},
        {'7', "pqrs"},
        {'8', "tuv"},
        {'9', "wxyz"}
    };
    // 存储答案
    string output;
    vector<string> ans;

public:
    // 深度优先遍历
    void dfs(const string& digits, int k){
        // 集满则放入
        if(digits.size() == k){
            ans.emplace_back(output);
            return;
        }
        // 每次选取一个
        char num = digits[k];
        for(char ch: map[num]){
            output.push_back(ch);
            dfs(digits, k+1);
            output.pop_back();
        }
    }
```

```
    // 题目
    vector<string> letterCombinations(string digits) {
        // 题目特殊情况空时，不能输出空串，要空数组
        if(digits.size() == 0)  return ans;

        // 从 k=0 开始dfs
        dfs(digits, 0);
        return ans;
    }
};
```

时间复杂度：$O(3^m * 4^n)$, 这里m为6，n为2.

空间复杂度：$O(m+n)$.

{% endtabs %}

58. 组合总和

- **题面：**
  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]]
  ```

  示例 3：

  ```
  输入: candidates = [2], target = 1
  输出: []
  ```

{% tabs 解法, -1 %}

```
class Solution {
```

```
private:
    vector<vector<int>> ans;
    vector<int> output;

public:
    // dfs: 接下来选 k, 当前还剩 cur 大小
    void dfs(int k, int cur, vector<int>& candidates){
        if(k >= candidates.size())  return;
        // 已经满足
        if(cur == 0){
            ans.emplace_back(output);
            return;
        }
        // 选用 k (会重复选, 也可能不选)
        if(candidates[k] <= cur){
            output.push_back(candidates[k]);
            dfs(k, cur-candidates[k], candidates);
            output.pop_back();
        }
        // 选用 k+1
        dfs(k+1, cur, candidates);
    }

    // 题目
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        dfs(0, target, candidates);
        return ans;
    }
};
```

时间复杂度：$O(S)$或$O(n*2^n)$，其中S为所有可行解的长度，n为位置长度.

空间复杂度：$O(target)$，最差情况的栈深度.

```
class Solution {
public:
    // 动态规划：空间需求高
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        // dp[i] 表示所有和为 i 的组合
        vector<vector<vector<int>>> dp(target+1);
        dp[0] = {{}};

        // 每次选用数字都更新一次所有可以更新的组合
        for(int candidate: candidates){
            // 如何更新：加上当前candidate 可以到达的新组合，都更新一遍（但是不要超过target）
            for(int i=0; candidate+i<=target; i++){
                for(vector<int>& group: dp[i]){   // 和为 i 的所有组合都取出来看看
                    vector<int> newgroup = group;
                    newgroup.push_back(candidate);  // 原来是 i 加上 candidate
                    dp[candidate+i].push_back(newgroup);
                }
            }
```

```
        }
        // 和为 target 的所有组合就是答案
        return dp[target];
    }
};
```

时间复杂度：$O(n * target * 2^n)$.

空间复杂度：$O(target * 2^n)$, 主要是dp耗费空间.

{% endtabs %}

59. 括号生成

- **题面：**
  数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。
- **示例 1：**

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

示例 2：

```
输入：n = 1
输出：["()"]
```

{% tabs 解法, -1 %}

```
class Solution {
private:
    vector<string> ans;
public:
    // 暴力法：一共 2^{2n} 个解，检测所有解是否正确即可
    // 检测方法：用 flag 表示 '('数量 - ')'数量，这个数字只能 >=0 且最终 =0
    bool check(int mask, int n, string& output){
        int flag = 0;
        for(int i=0; i<2*n; i++){
            // 左括号
            if(mask & (1<<i)){
                output.push_back('(');
                flag++;
            }
            // 右括号
            else{
                output.push_back(')');
                flag--;
            }
            // 过程中都不能 <0
            if(flag < 0)  break;
```

```
        }
        return flag==0;
    }
    // 题目
    vector<string> generateParenthesis(int n) {
        int MaxMask = 1<<(2*n);
        for(int mask=0; mask<=MaxMask; mask++){
            string output;
            if(check(mask, n, output))
                ans.push_back(output);
        }
        return ans;
    }
};
```

时间复杂度：$O(2^{2n} * n)$.

空间复杂度：$O(n)$栈深度最差$2n$.

```
class Solution {
private:
    vector<string> ans;

public:
    // 深度优先遍历：当前是output，还需要 lc个( 和 rc个)
    void dfs(string output, int lc, int rc){
        // 满足条件
        if(lc==0 && rc==0){
            ans.emplace_back(output);
            return;
        }
        // 加 (
        if(lc > 0)  dfs(output+'(', lc-1, rc);
        // 加 ) ：注意应该是还需要的 rc 更多
        if(rc > 0 && rc-lc > 0) dfs(output+')', lc, rc-1);
    }
    // 题目
    vector<string> generateParenthesis(int n) {
        dfs("", n, n);
        return ans;
    }
};
```

时间复杂度：低于$O(2^{2n} * n)$，因为相当于有剪枝.

空间复杂度：$O(n)$.

{% endtabs %}

60. 单词搜索

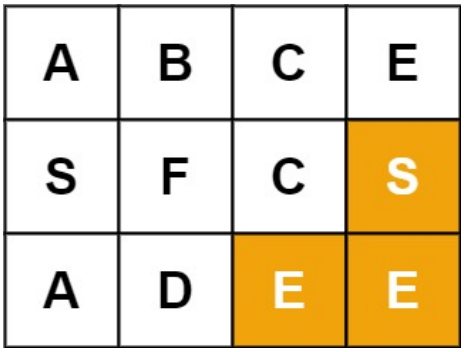
- **题面：**

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

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

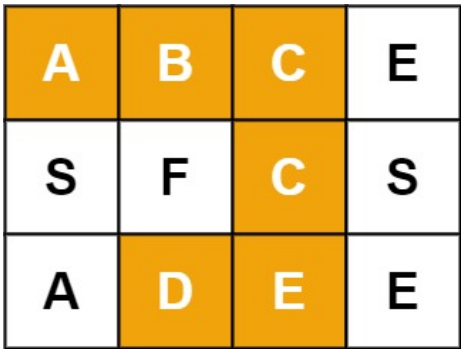
  `board` 和 `word` 仅由大小写英文字母组成

- **示例 1：**



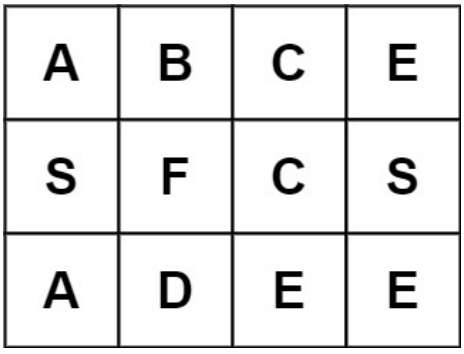

```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
输出：true
```

示例 2：




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

示例 3：



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

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<vector<int>> visited;    // 标记是否访问过
    int DIRS[4][2] = { {0,-1}, {0,1}, {-1,0}, {1,0} };  // 左右上下

public:
    // 是否为可行位置：没有越界，且还未访问过
    bool feasible(int i, int j, const vector<vector<char>>& board){
        return i>=0 && i<board.size() && j>=0 && j<board[0].size() && !visited[i][j];
    }

    // 检测从 i,j 位置出发，能否搜索到字符串 word[k:]
    bool check(int i, int j, int k, const vector<vector<char>>& board, const string& word){
        // 当前位置不符，直接退出
        if(board[i][j] != word[k])  return false;
        // 当前位置符合
        // 1. 已经完成
        if(k == word.size()-1)  return true;
        // 2. 还能四方向探索
        visited[i][j] = true;
        for(auto& [dx, dy]: DIRS){
            int newi = i+dx, newj = j+dy;
            if(feasible(newi, newj, board)){            // 是可行区域，那么探索
                if(check(newi, newj, k+1, board, word)) // 探索则从 k+1 开始的字符串即可
                    return true;    // 只要有一个成功，就提前退出
            }
        }
        visited[i][j] = false;
        return false;   // 到此，肯定是失败了
    }

    // 题目
    bool exist(vector<vector<char>>& board, string word) {
        // 初始化 visited 为 m * n 大小
        int m = board.size(), n = board[0].size();
        visited.resize(m);
        for(int i=0; i<m; i++)  visited[i].resize(n, 0);
        // 把每个位置作为起始都探索一遍即可
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
                if(check(i, j, 0, board, word))
                    return true;   // 有一个位置成功，就算成功
        return false;   // 全都失败
    }
```

```
45 };
```

时间复杂度：宽松上界$O(m*n*3^L)$, mn为$board$大小，L为$word$长度.

空间复杂度：$O(mn)$.

{% endtabs %}

61. 分割回文串

- **题面：**
  给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是 `回文串`。返回 `s` 所有可能的分割方案。
  `s` 仅由小写英文字母组成
- **示例 1：**

```
输入：s = "aab"
输出：[["a","a","b"],["aa","b"]]
```

示例 2：

```
输入：s = "a"
输出：[["a"]]
```

{% tabs 解法, -1 %}

```
class Solution {
private:
    // dp[i][j] 表示 子串s[i:j] 是否为回文串
    vector<vector<int>> dp;
    // 存储答案
    vector<vector<string>> ans;
    vector<string> path;    // 一次的答案

public:
    // 1. 先动态规划预处理把所有子字符串s[i:j] 是否为回文串的判断做完
    void DP(const string& s){
        int n = s.size();
        // 初始化全为true 主要是把 i>=j 的都赋值为 true
        dp.assign(n, vector<int>(n, true));
        // 动态规划改值
        for(int i=n-1; i>=0; i--)   // 如何判断 i,j 怎么变化? 看状态转移方程的需求
            for(int j=i+1; j<n; j++)
                dp[i][j] = dp[i+1][j-1] && (s[i]==s[j]);
    }

    // 2. 再使用深度优先遍历回溯枚举，假定 k 前面已经完成分割，从 k 为起始分割后面的
    void dfs(int k, const string& s){
        // 已经完成
```

```
        if(k == s.size()){
            ans.emplace_back(path);
            return;
        }
        // 分割出回文串 s[k:j] 那么需要枚举 j
        for(int j=k; j<s.size(); j++){
            if(dp[k][j]){
                path.push_back(s.substr(k, j-k+1)); // s.substr(start, len)
                dfs(j+1, s);    // 然后继续深度优先把 j+1 起始往后的补全
                path.pop_back();    // 回溯，吐出来
            }
        }
    }

    // 题目
    vector<vector<string>> partition(string s) {
        DP(s);      // 动态规划预处理
        dfs(0, s);  // 深度优先回溯，从0起始开始分割
        return ans;
    }
};
```

时间复杂度：$O(n*2^n)$最差情况下，字符串n长度但全是相同字符，因此可以2^{n-1}种划分.

空间复杂度：$O(n^2)$.

{% endtabs %}

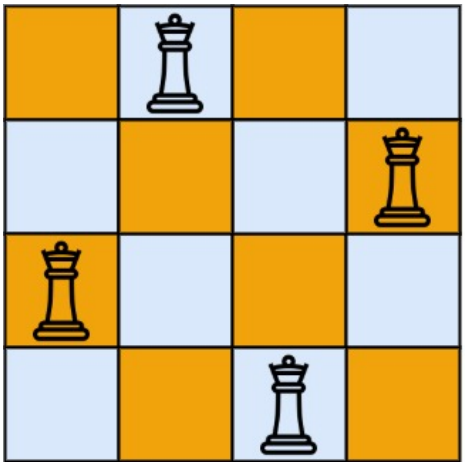
62. N 皇后

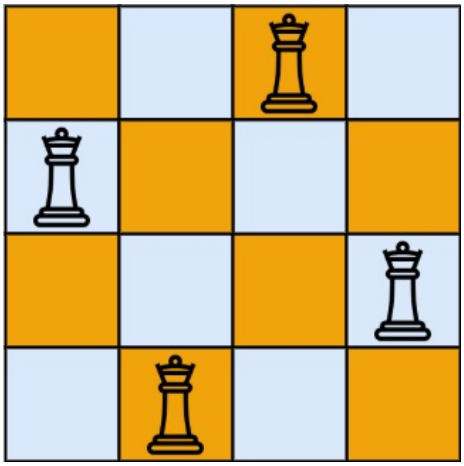
- **题面：**
  按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

  n 皇后问题 研究的是如何将 n 个皇后放置在 `n×n` 的棋盘上，并且使皇后彼此之间不能相互攻击。

  给你一个整数 `n` ，返回所有不同的 n 皇后问题 的解决方案。

  每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。
- **示例 1：**





```
输入：n = 4
输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
解释：如上图所示，4 皇后问题存在两个不同的解法。
```

示例 2：

```
输入：n = 1
输出：[["Q"]]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<vector<string>> ans;
    vector<string> path;
    vector<pair<int, int>> cur;

public:
    // 判断是否可摆放
    bool check(int i, int j){
        // 同行、同列、斜线
        for(auto& [x, y]: cur)
            if(x==i || y==j || abs(i-x)==abs(j-y))
                return false;
        return true;
    }

    // 回溯法：k行之前已经完成可行，接下来在第k行开始摆放后续的皇后
    void backtrace(int k, int n){
        // 完成
        if(k == n){
            ans.emplace_back(path);
            return;
        }
        // 在第 k 行依次摆放，记得撤回
        for(int i=0; i<n; i++){
            if(check(k, i)){    // 检查这个位置是否可以摆放
                string s(n, '.');  s[i] = 'Q';
                path.push_back(s);  cur.push_back({k, i});
                backtrace(k+1, n);  // 下面再填写 k+1 行即可
                path.pop_back();  cur.pop_back();
            }
        }
    }
    // 题目
    vector<vector<string>> solveNQueens(int n) {
        backtrace(0, n);
        return ans;
    }
};
```

时间复杂度：$O(n!)$.

空间复杂度：$O(n)$.

{% endtabs %}

十、二分查找

63. 搜索插入位置

- **题面：**
  给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

  请必须使用时间复杂度为 `O(log n)` 的算法。

  `nums` 为 `无重复元素` 的 `升序` 排列数组。

- **示例 1:**

```
输入: nums = [1,3,5,6], target = 5
输出: 2
```

示例 2:

```
输入: nums = [1,3,5,6], target = 2
输出: 1
```

示例 3:

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        // 直接写循环二分
        int lc = 0, rc = nums.size()-1;
        while(lc <= rc){
            int mid = (lc+rc)/2;
            if(target <= nums[mid])
                rc = mid-1;
            else
                lc = mid+1;
        }
        return lc;
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

{% endtabs %}

64. 搜索二维矩阵

- **题面：**
  给你一个满足下述两条属性的 `m x n` 整数矩阵：

  每行中的整数从左到右按非严格递增顺序排列。

  每行的第一个整数大于前一行的最后一个整数。

  给你一个整数 `target` ，如果 `target` 在矩阵中，返回 `true` ；否则，返回 `false` 。
- **示例 1：**

1	3	5	7
10	11	16	20
23	30	34	60

```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出：true
```

示例 2：

1	3	5	7
10	11	16	20
23	30	34	60

```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
输出：false
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
```

```
        // 两次二分查找
        // 第一次：找到可能包含的那一行
        int li = 0, ri = matrix.size()-1;
        while(li <= ri){
            int mi = (li+ri)/2;
            if(target == matrix[mi][0])
                return true;
            else if(target < matrix[mi][0])
                ri = mi-1;
            else
                li = mi+1;
        }
        int row = max(0, min(li, ri));
        // 第二次：找到这一行里面有没有
        int lj = 0, rj = matrix[0].size()-1;
        while(lj <= rj){
            int mj = (lj+rj)/2;
            if(target == matrix[row][mj])
                return true;
            else if(target < matrix[row][mj])
                rj = mj-1;
            else
                lj = mj+1;
        }
        int col = max(0, min(lj, rj));
        // 最终判断
        return matrix[row][col] == target;
    }
};
```

时间复杂度：$O(\log m + \log n = \log mn)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 完美使用 STL
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        // 编写二维数组大小判断的 lambda 函数
        // 函数指针 = [捕获外界变量表](函数参数表) -> 返回类型 { 函数体 };
        auto f = [](int a, vector<int>& b) -> bool { return a < b[0]; };
        // 找到第一个大于 target 的所在行
        auto row = upper_bound(matrix.begin(), matrix.end(), target, f);
        // 判断:注意 row 是指针
        if(row == matrix.begin())
            return false;   // 第一行就更大，说明不存在
        // 否则，应该回退一行，才是所在行
        row--;
        // 下面直接在所在行二分查找即可
        return binary_search(row->begin(), row->end(), target);
    }
};
```

时间复杂度：$O(\log mn)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 还有办法，直接一次二分查找（关键看如何处理行列：取整、取余）
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int lc = 0, rc = m*n-1;
        while(lc <= rc){    // 注意有等于
            int mid = (lc+rc)/2;
            int x = matrix[mid/n][mid%n];   // 关键点！！！
            if(target == x)
                return true;
            else if(target < x)
                rc = mid-1;
            else
                lc = mid+1;
        }
        // 最终失败
        return false;
    }
};
```

时间复杂度：$O(\log mn)$.

空间复杂度：$O(1)$.

{% endtabs %}

65. 在排序数组中查找元素的第一个和最后一个位置

- **题面：**
  给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。

  如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

  你必须设计并实现时间复杂度为 `O(log n)` 的算法解决此问题。

- **示例 1：**

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

示例 2：

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

示例 3：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        // 空单独处理
        int n = nums.size();
        if(n==0)
            return vector<int>{-1, -1};
        // 正常处理
        int lc = 0, rc = n-1;
        int ansl = -1, ansr = -1;
        // 仍然套用二分查找：目标是找到第一个 target 所在位置
        while(lc <= rc){
            int mid = (lc+rc)/2;
            if(target <= nums[mid])
                rc = mid-1;
            else
                lc = mid+1;
        }
        // 若有找到，那么继续增大到第一个不是 target 的即可
        if(lc<=n-1 && nums[lc] == target){          // 注意超界
            ansl = lc;
            ansr = ansl;
            while(ansr<=n-1 && nums[ansr]==target)  // 注意超界
                ansr++;
            ansr--;
        }
        // 返回
        return vector<int>{ansl, ansr};
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        // 直接无脑两次二分查找也可以
        if(nums.size()==0)  return vector<int>{-1, -1};
        // 第一次找左边界
        int l = 0, r = nums.size()-1;
        while(l <= r){
            int mid = (l+r)/2;
```

```
            if(target <= nums[mid])
                r = mid-1;
            else
                l = mid+1;
        }
        int ansl = l;
        if(ansl > nums.size()-1 || nums[ansl] != target)    // 注意超界
            return vector<int>{-1, -1};
        // 第二次找右边界
        l = 0, r = nums.size()-1;
        while(l <= r){
            int mid = (l+r)/2;
            if(target >= nums[mid])
                l = mid+1;
            else
                r = mid-1;
        }
        int ansr = r;
        return vector<int>{ansl, ansr};
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

{% endtabs %}

66. 搜索旋转排序数组

- **题面：**
  整数数组 `nums` 按**升序**排列，数组中的值 **互不相同** 。

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

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

  你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。

- **示例 1：**

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

示例 2：

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

示例 3：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 仍然可以二分法，有一半可以拿来作为条件支撑
    int search(vector<int>& nums, int target) {
        int l = 0, r = nums.size()-1;
        while(l <= r){
            int mid = (l+r)/2;
            if(target == nums[mid])
                return mid;
            // 看哪边有序，就拿来判断(前提：不能有相同元素)
            else{
                // 1. 左一半有序
                if(nums[l] <= nums[mid]){    // 这里必须是 <=
                    if(nums[l] <= target && target < nums[mid])
                        r = mid-1;
                    else
                        l = mid+1;
                }
                // 2. 右一半有序
                else{
                    if(nums[mid] < target && target <= nums[r])
                        l = mid+1;
                    else
                        r = mid-1;
                }
            }
        }
        // 出循环是失败
        return -1;
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

{% endtabs %}

67. 寻找旋转排序数组中的最小值

- **题面：**
  已知一个长度为 `n` 的数组，预先按照**升序**排列，经由 1 到 n 次 旋转 后，得到输入数组。例如，原数组 `nums = [0,1,2,4,5,6,7]` 在变化后可能得到：

若旋转 `4` 次，则可以得到 `[4,5,6,7,0,1,2]`

若旋转 `7` 次，则可以得到 `[0,1,2,4,5,6,7]`

注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` 旋转一次 的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]` 。

给你一个元素值 **互不相同** 的数组 `nums` ，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素** 。

你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。

- **示例 1：**

```
输入：nums = [3,4,5,1,2]
输出：1
解释：原数组为 [1,2,3,4,5] ，旋转 3 次得到输入数组。
```

示例 2：

```
输入：nums = [4,5,6,7,0,1,2]
输出：0
解释：原数组为 [0,1,2,4,5,6,7] ，旋转 4 次得到输入数组。
```

示例 3：

```
输入：nums = [11,13,15,17]
输出：11
解释：原数组为 [11,13,15,17] ，旋转 4 次得到输入数组。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 违规方法
    int findMin(vector<int>& nums) {
        auto p = min_element(nums.begin(), nums.end());
        return *p;  // 若空，则 p == nums.end()，这里不用考虑
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 仍然二分法
    int findMin(vector<int>& nums) {
        int l = 0, r = nums.size()-1;
        int ans = nums[0];
        while(l <= r){
```

```
            int mid = (l+r)/2;
            if(nums[mid] < ans)
                ans = nums[mid];
            // 哪边边界最小，往那边走
            int left = min(nums[l], nums[max(mid-1, l)]);   // 注意防止超界
            int right = min(nums[min(mid+1, r)], nums[r]);
            if(left < right)
                r = mid-1;
            else
                l = mid+1;
        }
        return ans;
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 二分法：右边界判别，若 nums[r] 比 nums[mid] 更大了，那么右边肯定是有序递增。
    // 原因：由于旋转，nums[r] 一定是 小于 nums[l] 的。再思考一下。
    int findMin(vector<int>& nums) {
        int l = 0, r = nums.size()-1;
        while(l < r){   // 等于时退出
            int mid = (l+r)/2;
            if(nums[mid] < nums[r])      // 右边界大，右边肯定是递增，因此抛弃右边
                r = mid;
            else                         // 否则，是最小值卡在右边，因此抛弃左边
                l = mid+1;
        }
        // 最终答案
        return nums[l];
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

{% endtabs %}

68. 寻找两个正序数组的中位数

- **题面：**
  给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数** 。

  算法的时间复杂度应该为 `O(log (m+n))` 。
- **示例 1：**

```
输入：nums1 = [1,3], nums2 = [2]
输出：2.00000
解释：合并数组 = [1,2,3] ，中位数 2
```

示例 2：

```
输入：nums1 = [1,2], nums2 = [3,4]
输出：2.50000
解释：合并数组 = [1,2,3,4] ，中位数 (2 + 3) / 2 = 2.5
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 违规方法：先合并排序，再找到中位数即可
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        // 合并
        vector<int> nums;
        nums.insert(nums.end(), nums1.begin(), nums1.end());
        nums.insert(nums.end(), nums2.begin(), nums2.end());
        // 排序
        sort(nums.begin(), nums.end());
        int n = nums.size();
        if(n%2 == 0)
            return (nums[n/2-1] + nums[n/2]) / 2.0;      // 注意是 2.0
        else
            return nums[n/2];
    }
};
```

时间复杂度：$O(m+n)$.

空间复杂度：$O(m+n)$.

```cpp
class Solution {
public:
    // 获得第 k 小的数字
    int getKth(vector<int>& nums1, vector<int>& nums2, int k){
        int m = nums1.size(), n = nums2.size();
        int s1 = 0, s2 = 0;
        // 每次排除前 k-2 个数字
        while(true){
            // 边界情况
            if(s1 == m)
                return nums2[s2 + k-1];
            if(s2 == n)
                return nums1[s1 + k-1];
            // 结束情况
            if(k == 1)
                return min(nums1[s1], nums2[s2]);
```

```
            // 一般情况
            int p1 = min(s1 + k/2-1, m-1);
            int p2 = min(s2 + k/2-1, n-1);
            if(nums1[p1] <= nums2[p2])
                k -= p1-s1+1, s1 = p1+1;    // 已经排除 s1--p1之间的数字了
            else
                k -= p2-s2+1 , s2 = p2+1;
        }
    }
    // 题目
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int N = nums1.size() + nums2.size();
        if(N%2 == 1)
            return getKth(nums1, nums2, N/2+1);
        else{
            int a = getKth(nums1, nums2, N/2);
            int b = getKth(nums1, nums2, N/2+1);
            return (a+b)/2.0;
        }
    }
};
```

时间复杂度：$O(\log(m+n))$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int len =nums1.size()+nums2.size();
        if(len % 2 == 0)
            return (GetK(nums1, nums2, len/2, 0, 0) + GetK(nums1, nums2, len/2+1, 0, 0)) / 2;
        else
            return GetK(nums1, nums2, len/2+1, 0, 0);
    }

    // 每次寻找第k小的数，每次先去除前k/2小的数
    double GetK(vector<int>& nums1, vector<int>& nums2, int k, int start1, int start2){
        int len1 = nums1.size()-start1, len2 = nums2.size()-start2;
        if(len1==0)
            return nums2[start2+k-1];
        if(len2==0)
            return nums1[start1+k-1];

        if(k==1)
            return min(nums1[start1], nums2[start2]);

        int k2 = min(k/2, len1);
        k2 = min(k2, len2);
        if(nums1[start1+k2-1] <= nums2[start2+k2-1])
```

```
            return GetK(nums1, nums2, k-k2, start1+k2, start2);
        else
            return GetK(nums1, nums2, k-k2, start1, start2+k2);
    }
};
```

时间复杂度：$O(\log(m+n))$.

空间复杂度：$O(m+n)$.

{% endtabs %}

十一、栈

69. 有效的括号

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

  有效字符串需满足：

  1. 左括号必须用相同类型的右括号闭合。
  2. 左括号必须以正确的顺序闭合。
  3. 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 = "()[]{}"
  输出：true
  ```

  示例 3：

  ```
  输入：s = "(]"
  输出：false
  ```

  示例 4：

  ```
  输入：s = "([])"
  输出：true
  ```

{% tabs 解法, -1 %}

```
class Solution {
```

```
private:
    vector<char> stack;

public:
    bool check(char ch){
        // 左括号直接加入
        if(ch == '(' || ch == '[' || ch == '{'){
            stack.push_back(ch);
            return true;
        }
        // 右括号时
        // 若空，则直接失败
        if(stack.empty())
            return false;
        // 不空，则需要匹配
        char top = stack[stack.size()-1];
        switch(ch){
            case ')':
                if(top == '('){
                    stack.pop_back();
                    return true;
                }
                break;
            case ']':
                if(top == '['){
                    stack.pop_back();
                    return true;
                }
                break;
            case '}':
                if(top == '{'){
                    stack.pop_back();
                    return true;
                }
                break;
        }

        // 未成功匹配
        return false;
    }

    // 模拟栈空间
    bool isValid(string s) {
        int n = s.size();
        for(int i=0; i<n; i++)
            if(!check(s[i]))
                return false;
        // 最后还需要是空的
        return stack.empty();
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 直接用栈 STL
    bool isValid(string s) {
        // 优化：若 s 奇数，必然失败
        int n = s.size();
        if(n%2)  return false;

        // 一般情况
        stack<char> stk;
        unordered_map<char, char> map = { {')', '('}, {']', '['}, {'}', '{'} };
        // 开始匹配
        for(int i=0; i<n; i++){
            char ch = s[i];
            // 右括号
            if(map.count(ch))
                // 空或无法匹配，都失败
                if(stk.empty() || stk.top() != map[ch])
                    return false;
                else
                    stk.pop();
            // 左括号
            else
                stk.push(ch);
        }
        // 返回
        return stk.empty();
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

70. 最小栈

- **题面：**
  设计一个支持 `push` ， `pop` ， `top` 操作，并能在**常数时间内**检索到**最小元素**的栈。

  实现 `MinStack` 类:

  - `MinStack()` 初始化堆栈对象。
  - `void push(int val)` 将元素`val`推入堆栈。
  - `void pop()` 删除堆栈顶部的元素。

- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

`pop`、`top` 和 `getMin` 操作总是在 **非空栈** 上调用

- **示例 1:**

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]
输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

{% tabs 解法, -1 %}

```cpp
class MinStack {
private:
    vector<int> stk;         // 依次存储数据
    vector<int> min_stk;     // 依次存储对应数据在时的最小数

public:
    MinStack() {
        min_stk.push_back(INT_MAX);
    }

    void push(int val) {
        stk.push_back(val);
        int newmin = min(val, min_stk.back());  // 每次对应数据进入时所对应的最小数字
        min_stk.push_back(newmin);
    }

    void pop() {
        stk.pop_back();
        min_stk.pop_back();
    }

    int top() {
        return stk.back();
    }

    int getMin() {
```

```
        return min_stk.back();
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(val);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

时间复杂度：$O(1)$.

空间复杂度：$O(n)$.

{% endtabs %}

71. 字符串解码

- **题面：**
  给定一个经过编码的字符串，返回它解码后的字符串。

  编码规则为: `k[encoded_string]`，表示其中方括号内部的 `encoded_string` 正好重复 `k` 次。注意 `k` 保证为正整数。

  你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

  此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 `k` ，例如不会出现像 `3a` 或 `2[4]` 的输入。

  - `s` 由`小写英文字母`、`数字`和`方括号 '[]'` 组成
  - `s` 保证是一个 **有效** 的输入。
  - `s` 中所有整数的取值范围为 `[1, 300]`

- **示例 1：**

  ```
  输入：s = "3[a]2[bc]"
  输出："aaabcbc"
  ```

  示例 2：

  ```
  输入：s = "3[a2[c]]"
  输出："accaccacc"
  ```

  示例 3：

```
输入：s = "2[abc]3[cd]ef"
输出："abcabccdcdcdef"
```

示例 4：

```
输入：s = "abc3[cd]xyz"
输出："abccdcdcdxyz"
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 巧解：每次记录数字和复制开始位置
    // 好处是因为复制操作直接复制从 start 到末尾的，不需要管中途这个末尾是否变长了
    string decodeString(string s) {
        string ans = "";
        int num = 0;
        stack<pair<int, int>> stk;
        for(int i=0; i<s.size(); i++){
            char ch = s[i];
            // 数字
            if(isdigit(ch))  num = num*10 + ch-'0';
            // 字母
            else if(isalpha(ch))  ans += ch;      // 直接加
            // 左括号：在当前位置开始，后续的字符串，要复制 num 个
            else if(ch == '['){
                stk.push({num, ans.size()});
                num = 0;
            }
            // 右括号：开始复制
            else if(ch == ']'){
                auto [n, si] = stk.top();
                string one = ans.substr(si, ans.size()-si);
                for(int i=1; i<=n-1; i++)  ans += one;    // 注意最初就有一个
                stk.pop();
            }
        }
        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 直观做法，两个栈
    string decodeString(string s) {
```

```
        stack<string> preStack; // 前面已完成的字符串
        stack<int> numStack;    // 下一个字符串需要重复的次数
        string curString = "";
        int curNum = 0;
        for(int i=0; i<s.size(); i++){
            char ch = s[i];
            // 数字
            if(isdigit(ch))  curNum = curNum*10 + ch-'0';
            // 字母
            else if(isalpha(ch))  curString += ch;
            // 左括号
            else if(ch == '['){
                preStack.push(curString);   // 前面的字符串，存起来
                numStack.push(curNum);      // 后续 curS 字符串需要复制的次数
                curNum = 0;
                curString = "";
            }
            // 右括号
            else if(ch == ']'){ // 每遇一次]，只会解体拼接一次，刚好
                // 退栈，当前字符串重复 num 次，接在前序字符串末尾
                string pre = preStack.top();  preStack.pop();
                int num = numStack.top();  numStack.pop();
                for(int i=0; i<num; i++)  pre += curString;
                curString = pre;
            }
        }
        // 最终的当前字符串，就是答案
        return curString;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

72. 每日温度

- **题面：**
  给定一个整数数组 `temperatures` ，表示每天的温度，返回一个数组 `answer` ，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

  提示：

  - 1 <= `temperatures.length` <= 105
  - 30 <= `temperatures[i]` <= 100
- **示例 1:**

```
输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]
```

示例 2:

```
输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]
```

示例 3:

```
输入: temperatures = [30,60,90]
输出: [1,1,0]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<int> ans;
public:
    // 暴力1（不行）：自身暴力遍历（超出时间限制，不行！）
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        int n = temperatures.size();
        for(int i=0; i<n; i++){
            int j=i+1, count = 0;
            for(; j<n; j++){
                count++;
                if(temperatures[j] > temperatures[i])  break;
            }
            if(j==n)  count = 0;
            ans.emplace_back(count);
        }
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
private:
    vector<int> ans;
    vector<int> least;
    // least[i] 表示气温 i 所在的最小天编号（会动态变化，保证取值时符合当前要求）

public:
    // 暴力 2（可以）：记录温度（只有 30-100，数量级小），反向倒看
    vector<int> dailyTemperatures(vector<int>& temperatures) {
```

```
        // 初始化
        int n = temperatures.size();
        ans.resize(n);
        least.assign(101, INT_MAX);
        // 开始查找
        for(int i=n-1; i>=0; i--){
            int t = temperatures[i];
            // 保存气温 t 的最近的天气编号是当前的i
            least[t] = i;
            // 遍历更大的气温，查看是否有天气编号在，选取最近的更大天气
            int near = INT_MAX;
            for(t=t+1; t<=100; t++)
                near = min(near, least[t]);
            // 判断查找结果
            if(near == INT_MAX)  ans[i] = 0;
            else  ans[i] = near - i;
        }
        // 返回
        return ans;
    }
};
```

时间复杂度：$O(70n) = O(n)$.

空间复杂度：$O(m)$.

```
class Solution {
private:
    vector<int> ans;
    stack<int> stk;

public:
    // 单调栈：保持一个逐渐递减温度的单调栈（栈中存储天编号）
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        int n = temperatures.size();
        ans.resize(n);
        // 从前往后
        for(int i=0; i<n; i++){
            // 栈不空，当前 push 要先把那些不满足单调的退栈（刚好退栈时记录答案）
            if(!stk.empty()){
                while(!stk.empty() && temperatures[stk.top()] < temperatures[i]){
                    ans[stk.top()] = i-stk.top();
                    stk.pop();
                }
            }
            // 当前气温现在是最小的了，可以入栈了
            stk.push(i);
        }
        // 最后处理（可能有多个并列相等温度的）
        while(!stk.empty()){
            ans[stk.top()] = 0;
            stk.pop();
```

```
        }
        // 答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

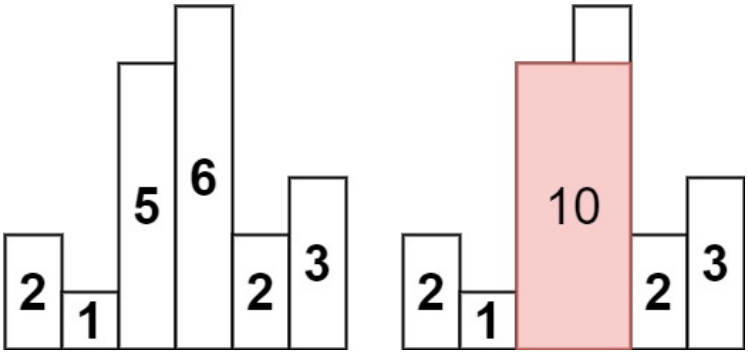
{% endtabs %}

73. 柱状图中最大的矩形

- **题面：**
  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` 。

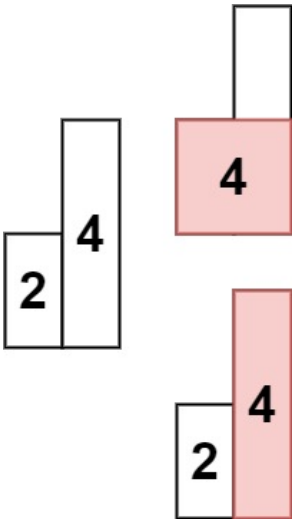
  求在该柱状图中，能够勾勒出来的矩形的最大面积。

- 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
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 单纯暴力：超出时间限制
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        int ans = 0;
        // 遍历左右边界
        for(int l=0; l<n; l++){
            int h = heights[l];
            for(int r=l; r<n; r++){
                h = min(h, heights[r]);      // h 取整个路过过程中最小的
                ans = max(ans, (r-l+1)*h);  // 计算面积，取答案
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 暴力 2：前面是相当于在枚举宽，这里，我们枚举高。（也是 n^2，超出时间限制）
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        int ans = 0;
        for(int i=0; i<n; i++){
            // 要保证本次就是这个高度，那么向两边延伸，直到不能是这个高度
            int hi = heights[i];
            int l = i, r = i;
            while(l-1 >= 0 && heights[l-1] >= hi)  l--;
            while(r+1 < n && heights[r+1] >= hi)  r++;
            // 更新答案
            ans = max(ans, (r-l+1)*hi);
        }
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
private:
    vector<int> L;  // L[i] 表示保证高度 h[i] 时，左边界下标
    vector<int> R;  // R[i] ...

public:
    // 接着枚举高，但是快速找到左右边界：单调栈法
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        L.resize(n);
        R.resize(n);
        // 下面开始，把 L 和 R 都找到。利用单调栈：保证栈中高度在递增
        stack<int> stk;
        for(int i=0; i<n; i++){
            while(!stk.empty() && heights[stk.top()] >= heights[i]) // 注意：有等于。下面解释
                stk.pop();
            // 当前 i 的满足 h 高度的最远左边界，就是栈顶那个了
            L[i] = stk.empty()? 0 : stk.top()+1;   // 若空，说明直接把开始作为左边界即可
            // 当前h[i]高度更高了，i进栈
            stk.push(i);
        }
        while(!stk.empty())  stk.pop();
        // 同样计算右边界
        for(int i=n-1; i>=0; i--){
            while(!stk.empty() && heights[stk.top()] >= heights[i])
                stk.pop();
            R[i] = stk.empty()? n-1 : stk.top()-1;    // 若空，什么直接把末尾作为右边界即可
            stk.push(i);
        }

        // L 和 R 准备好了，计算即可
        int ans = 0;
        for(int i=0; i<n; i++)
            ans = max(ans, (R[i]-L[i]+1) * heights[i]);
        return ans;
    }
};

/*
这里的 heights[stk.top()] >= heights[i] 有等于，
因为这里的 stk 是为了找到第一个不满足条件的左边界。
*/
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

十二、堆

74. 数组中的第K个最大元素

- **题面：**

  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 1:**

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

  示例 2:

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 违规方法
    int findKthLargest(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        reverse(nums.begin(), nums.end());
        return nums[k-1];
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 违规写法
    int findKthLargest(vector<int>& nums, int k) {
        // 默认大根堆
        priority_queue<int> heap;
        // 元素全部放入
        for(int num: nums)  heap.push(num);
        // 弹出 k-1 个
        for(int i=1; i<=k-1; i++)  heap.pop();
        // 当前堆顶即是
        return heap.top();
    }
```

```
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(n)$.

```cpp
// 正确做法：需要自己实现大根堆(递归实现，利用完全二叉树的性质)
class Solution {
public:
    // 调整一个三角形区域的大根堆
    void heapify(vector<int>& nums, int i, int n){
        int largest = i;
        int left = 2*i+1, right = 2*i+2;    // 完全二叉树的节点的左右节点
        // 找到三个里面最大的
        if(left < n && nums[left] > nums[largest])  largest = left;
        if(right < n && nums[right] > nums[largest])  largest = right;
        // 最大值有变化，则改动，且保证改动是整个大根堆都是正确的
        if(largest != i){
            swap(nums[i], nums[largest]);
            heapify(nums, largest, n);     // 调整子树
        }
    }

    // 题目
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();

        // 建立好大根堆，从最后一个非叶子节点开始全面调整
        for(int i=n/2-1; i>=0; i--)  heapify(nums, i, n);

        // 弹出 k-1 个出来（放在最末尾，不使用）
        for(int i=1; i<=k-1; i++){
            swap(nums[0], nums[n-i]);        // 弹出一个最大的num[0]，放到末尾，不需要了
            heapify(nums, 0, n-i);           // 调整剩余的堆
        }

        // 现在的堆顶就是第 K 大的
        return nums[0];
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(\log n)$.

{% endtabs %}

75. 前 K 个高频元素

- **题面：**
  给你一个整数数组 `nums` 和一个整数 `k` ，请你返回其中出现频率前 `k` 高的元素。你可以按 `任意顺序` 返回答案。

  提示：

  - 1 <= `nums.length` <= 105
  - `k` 的取值范围是 `[1, 数组中不相同的元素的个数]`
  - 题目数据保证答案唯一，换句话说，数组中前 `k` 个高频元素的集合是唯一的

  进阶：你所设计算法的时间复杂度 必须 优于 `O(n log n)` ，其中 `n` 是数组大小。
- **示例 1:**

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

  示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<int> ans;

public:
    // 注意这里比较器的写法
    struct Compare{
        bool operator()(const pair<int, int>& A, const pair<int, int>& B){
            return A.second < B.second;
        }
    };

    vector<int> topKFrequent(vector<int>& nums, int k) {
        // 哈希表
        unordered_map<int, int> dict;
        for(int num: nums)
            if(dict.count(num))  dict[num]++;
            else dict[num] = 1;

        // 存入大根堆
        priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> heap;
        for(auto& kv : dict)  heap.push({kv.first, kv.second});

        // 弹出前 k 个即可
        for(int i=0; i<k; i++){
```

```
            ans.push_back(heap.top().first);
            heap.pop();
        }

        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n \log n)$.

空间复杂度：$O(n)$.

```
class Solution {
private:
    vector<int> ans;

public:
    // 注意这里比较器的写法
    struct Compare{
        bool operator()(const pair<int, int>& A, const pair<int, int>& B){
            return A.second > B.second;
        }
    };

    // 基于此，还可以优化，改成大小为 k 的最小堆，每次把更小的替换走，最终堆中的 k 个就是答案。
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // 哈希表
        unordered_map<int, int> dict;
        for(int num: nums)
            if(dict.count(num))  dict[num]++;
            else dict[num] = 1;

        // 存入小根堆
        priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> heap;
        for(auto& kv : dict)
            if(heap.size()<k)  heap.push({kv.first, kv.second});
            else{
                // 堆顶是最小的，若这个最小的的确比待进入的更小，那么最小的离开
                if(heap.top().second < kv.second){
                    heap.pop();
                    heap.push({kv.first, kv.second});
                }
            }

        // 弹出前 k 个即可
        for(int i=0; i<k; i++){
            ans.push_back(heap.top().first);
            heap.pop();
        }

        // 返回答案
```

```
        return ans;
    }
};
```

时间复杂度：$O(n \log k)$.

空间复杂度：$O(k)$.

```
class Solution {
private:
    vector<int> ans;
    vector<vector<int>> bucket; // bucket[i] 表示出现 i 次的字符（可能多个）

public:
    // 桶排序：把频率作为下标，存储起来
    vector<int> topKFrequent(vector<int>& nums, int k) {
        int n = nums.size();
        // 哈希表
        unordered_map<int, int> dict;
        for(int num: nums)
            if(dict.count(num))  dict[num]++;
            else dict[num] = 1;

        // 按频率存入桶中
        bucket.resize(n+1);
        for(auto& [k, v] : dict)  bucket[v].push_back(k);

        // 把 k 个最大拿出来
        int count = 0;
        for(int i=n; i>=0 && count<k; i--){
            if(bucket[i].size()){
                ans.insert(ans.end(), bucket[i].begin(), bucket[i].end());
                count += bucket[i].size();
            }
        }

        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

76. 数据流的中位数

- **题面：**
  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` 。

  实现 `MedianFinder` 类:

  `MedianFinder()` 初始化 `MedianFinder` 对象。

  `void addNum(int num)` 将数据流中的整数 `num` 添加到数据结构中。

  `double findMedian()` 返回到目前为止所有元素的中位数。与实际答案相差 `10^{-5}` 以内的答案将被接受。

  提示:

  - `-105 <= num <= 105`
  - 在调用 `findMedian` 之前，数据结构中至少有一个元素
  - 最多 `5 * 104` 次调用 `addNum` 和 `findMedian`
- **示例 1：**

```
输入
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
输出
[null, null, null, 1.5, null, 2.0]

解释
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // 返回 1.5 ((1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

{% tabs 解法, -1 %}

```
// 超出时间限制
class MedianFinder {
private:
    vector<int> increase;      // 递增单调栈

public:
    MedianFinder() {

    }
    // 完全保证 stk 是递增趋势的
    void addNum(int num) {
        stack<int> leave;
        // 弹出不满足的
        while(!increase.empty() && num < increase.back()){
```

```
            leave.push(increase.back());
            increase.pop_back();
        }
        // 找到合适位置，放入
        increase.push_back(num);
        // 放回去
        while(!leave.empty()){
            increase.push_back(leave.top());
            leave.pop();
        }
    }

    double findMedian() {
        // 每次取中间的即可
        int n = increase.size();
        if(n%2)  return increase[n/2];
        else  return (increase[n/2-1] + increase[n/2]) / 2.0;
    }
};

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class MedianFinder {
// 本质上，需要一直记录中间的数字，通过两个往中间逼近的优先队列来完成
private:
    // frontQ 存储中位数左边的（按照大根堆存储）中位数本身也存储左边
    priority_queue<int, vector<int>, less<int>> frontQ;
    // backQ 存储中位数右边的（按照小根堆存储）
    priority_queue<int, vector<int>, greater<int>> backQ;

public:
    MedianFinder() {

    }

    void addNum(int num) {
        // 最初，是直接存储前面
        if(frontQ.size()==0){
            frontQ.push(num);
            return;
        }
        // 根据 num 与中位数大小，来判断去哪
        double mid = findMedian();
```

```
        if(num <= mid)  frontQ.push(num);
        else  backQ.push(num);
        // 调整左右两边的队列大小：只可能是 frontQ == backQ + 0/1
        int ln = frontQ.size(), rn = backQ.size();
        // 左边过长
        if(ln == rn+2){
            backQ.push(frontQ.top());
            frontQ.pop();
        }
        // 右边过长
        if(rn == ln+1){
            frontQ.push(backQ.top());
            backQ.pop();
        }
    }

    double findMedian() {
        if(frontQ.size() == backQ.size())
            return (frontQ.top() + backQ.top()) / 2.0;
        else
            return frontQ.top() * 1.0;
    }
};

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(n)$.

{% endtabs %}

十三、贪心算法

77. 买卖股票的最佳时机

- **题面：**
  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。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // 找到前序最低点，后序最高点
        int ans = 0;
        int low = prices[0], high = prices[0];
        for(int price : prices){
            ans = max(ans, price - low);
            low = min(low, price);
            high = max(high, price);
        }
        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

78. 跳跃游戏

- **题面：**
  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 1：**

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

示例 2：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态更新最长可达位置即可
    bool canJump(vector<int>& nums) {
        int n = nums.size();
        int maxlen = 0;
        for(int i=0; i<n && i<=maxlen; i++)
            maxlen = max(maxlen, i+nums[i]);
        // 最长是否超过末尾了，就是答案
        return maxlen >= n-1;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

79. 跳跃游戏 II

- **题面：**
  给定一个长度为 `n` 的 `0` 索引整数数组 `nums`。初始位置为 `nums[0]`。

  每个元素 `nums[i]` 表示从索引 `i` 向后跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处:

  - `0 <= j <= nums[i]`
  - `i + j < n`

  返回到达 `nums[n - 1]` 的最小跳跃次数。

  生成的测试用例可以到达 `nums[n - 1]`。

  提示:

  - `1 <= nums.length <= 104`

- `0 <= nums[i] <= 1000`
- `题目保证可以到达 nums[n-1]`

- **示例 1:**

```
输入: nums = [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

示例 2:

```
输入: nums = [2,3,0,1,4]
输出: 2
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 倒推位置
    int jump(vector<int>& nums) {
        int pos = nums.size()-1;
        int count = 0;
        while(pos > 0){
            // 看哪个位置跳过来更好
            for(int i=0; i<pos; i++)
                if(i+nums[i] >= pos){
                    pos = i;
                    count++;
                    break;
                }
        }
        // 返回答案
        return count;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 动态更新最长可达，在确定一次必跳位置后
    // 要思考是否本次就直接跳到最远，有时候可能跳中间会为后续创造更长的范围
    int jump(vector<int>& nums) {
        int n = nums.size();
        int count = 0, cur = 0;
        // 到达 n-1 位置即可退出
        while(cur < n-1){
```

```
            int R = cur + nums[cur];     // 当前的右边界
            count++;                     // 跳，具体跳到哪里，下面分析
            // 完成
            if(R >= n-1)  break;
            // 未完成，则需要想最佳位置，而不是到本次的最远
            int best = R;
            for(int i=cur+1; i<=R && i<n; i++)
                if(i+nums[i] > best+nums[best])
                    best = i;
            // 那就选择跳到这个 best 位置
            cur = best;
        }
        // 返回答案
        return count;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 也是可以一次遍历的
    int jump(vector<int>& nums) {
        int n = nums.size();

        // 标记一下，若已经超出右边界 R，则说明本次跳了，最远可达就是当前的 maxlen
        //（其实会发现，之前判断失败的那些位置，后续也不会用）
        int count = 0, maxlen = 0;
        int R = 0;
        for(int i=0; i<n-1; i++){
            maxlen = max(maxlen, i+nums[i]);
            if(i == R){
                R = maxlen;
                count++;
            }
        }

        // 返回答案
        return count;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

80. 划分字母区间

- **题面：**
  给你一个字符串 `s` 。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。例如，字符串 `"ababcc"` 能够被分为 `["abab", "cc"]`，但类似 `["aba", "bcc"]` 或 `["ab", "ab", "cc"]` 的划分是非法的。

  注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 `s` 。

  返回一个表示每个字符串片段的长度的列表。

  提示：

  - `1 <= s.length <= 500`
  - `s` 仅由小写英文字母组成

- **示例 1：**

```
输入：s = "ababcbacadefegdehijhklij"
输出：[9,7,8]
解释：
划分结果为 "ababcbaca"、"defegde"、"hijhklij" 。
每个字母最多出现在一个片段中。
像 "ababcbacadefegde", "hijhklij" 这样的划分是错误的，因为划分的片段数较少。
```

  示例 2：

```
输入：s = "eccbbbbdec"
输出：[10]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<int> ans;

public:
    // 暴力
    vector<int> partitionLabels(string s) {
        int n = s.size();
        int L = -1, R = -1;  // 片段的左右边界
        for(int i=0; i<=n-1; i++){
            // 找到本次字符对应的右边界
            int j;
            for(j=n-1; j>R; j--)  if(s[i] == s[j])  break;

            // 有进展
            if(j > R){
                // 新切片
                if(i > R){
                    R = j;
```

```
                    ans.push_back(R-L);
                }
                // 旧切片
                else{
                    R = j;
                    ans.pop_back();
                    ans.push_back(R-L);
                }
            }
            // 是否更新左边界
            if(i==R && j<=R)  L = i;
        }
        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

```
class Solution {
private:
    int last[26];   // 每个字符的最排后的下标
    vector<int> ans;

public:
    // 一次遍历：准备好每个字符的最末尾位置即可
    vector<int> partitionLabels(string s) {
        int n = s.size();
        // 准备好 last
        for(int i=0; i<n; i++)  last[s[i]-'a'] = max(i, last[s[i]-'a']);

        // 开始看每个字符的所需长度
        int L = 0, R = 0;  // 标记
        for(int i=0; i<n; i++){
            R = max(R, last[s[i]-'a']);
            // 全部都是抵达边界再存入答案、更新 L（确实！）
            if(i==R){
                ans.push_back(R-L+1);
                L = i+1;
            }
        }

        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

{% endtabs %}

十四、动态规划

81. 爬楼梯

- **题面：**
  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?

  提示：

  - `1 <= n <= 45`
- **示例 1：**

```
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
```

示例 2：

```
输入：n = 3
输出：3
解释：有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    int dp[46];     // dp[i] 表示爬 n 个阶的方法数量

public:
    // 动态规划：关键在于找到递推公式即可
    int climbStairs(int n) {
        // 初始化
        dp[1] = 1, dp[2] = 2;

        // 爬到i  = 爬到i-1再爬一个 或者 爬到i-2再爬两个
        for(int i=3; i<=n; i++)
            dp[i] = dp[i-1] + dp[i-2];

        // 返回答案
```

```
        return dp[n];
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    // 动态规划：关键在于找到递推公式即可
    // 可以看到，实际上很简单的递推公式，就直接两个变量即可
    int climbStairs(int n) {
        // 初始化
        int pre1 = 0, pre2 = 1, cur;
        for(int i=1; i<=n; i++){
            cur = pre1 + pre2;
            pre1 = pre2;
            pre2 = cur;
        }

        // 返回答案
        return cur;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

这里找到递推公式的矩阵乘法运算，使用快速矩阵幂运算，进行一次运算即可。

时间复杂度降到快速幂运算的时间复杂度 $O(\log n)$。

不强求会代码，但是思路很好，可以参考链接：方法二

```
class Solution {
public:
    vector<vector<long long>> multiply(vector<vector<long long>> &a,
vector<vector<long long>> &b) {
        vector<vector<long long>> c(2, vector<long long>(2));
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
            }
        }
        return c;
    }

    vector<vector<long long>> matrixPow(vector<vector<long long>> a, int n) {
        vector<vector<long long>> ret = {{1, 0}, {0, 1}};
```

```
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }

    int climbStairs(int n) {
        vector<vector<long long>> ret = {{1, 1}, {1, 0}};
        vector<vector<long long>> res = matrixPow(ret, n);
        return res[0][0];
    }
};
```

时间复杂度：$O(\log n)$.

空间复杂度：$O(1)$.

{% endtabs %}

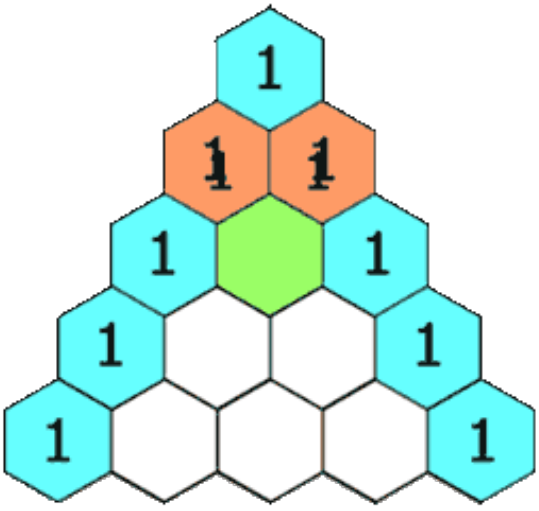
82. 杨辉三角

- **题面：**
  给定一个非负整数 `numRows`，生成 `「杨辉三角」` 的前 `numRows` 行。

  在 `「杨辉三角」` 中，每个数是它左上方和右上方的数的和。

  提示:

  - `1 <= numRows <= 30`



- **示例 1:**

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

示例 2:

```
输入：numRows = 1
输出：[[1]]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
private:
    vector<vector<int>> C;

public:
    // 第 n 行第 m 个数字（m、n 下标均从 0 开始）
    // 就是组合数 C(n,m)
    // 递推公式： C(n,m) = C(n-1, m) + C(n-1, m-1);
    vector<vector<int>> generate(int numRows) {
        C.resize(numRows);

        // n 行(下标从 0 开始)
        for(int n=0; n<numRows; n++){
            // 本行的两侧直接初始化
            C[n].resize(n+1);
            C[n][0] = C[n][n] = 1;
            // 本行的中间用递推公式
            for(int m=1; m<n; m++)
                C[n][m] = C[n-1][m] + C[n-1][m-1];
        }

        // 返回答案
        return C;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$不考虑答案返回值占用空间.

{% endtabs %}

83. 打家劫舍

- **题面：**
  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。

提示：

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

- **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 。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int rob(vector<int>& nums) {
        int n = nums.size();
        // 特例：由于需要初始化dp[1] 表示必须长度 2 及以上
        if(n == 1)  return nums[0];

        vector<int> dp = vector<int>(n, 0); // dp[k] 表示前 k 家的最高金额
        // 初始化
        dp[0] = nums[0];                    // 前一家，就偷这一家
        dp[1] = max(nums[0], nums[1]);      // 前两家，选一家更大的

        // 递推（有多家）偷取前 k 家的最高金额
        // 1. 要么不偷第k家 =  只偷前k-1家的
        // 2. 要么偷取第k家 =  只能偷前k-2家的 + 偷取第k家
        for(int k=2; k<n; k++)
            dp[k] = max(dp[k-1], dp[k-2] + nums[k]);

        // 返回答案
        return dp[n-1];
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$不考虑返回值答案的占用.

{% endtabs %}

84. 完全平方数

- **题面：**
  给你一个整数 `n` ，返回 **和为** `n` 的完全平方数的最少数量 。

  完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，`1、4、9 和 16` 都是完全平方数，而 `3` 和 `11` 不是。

  提示：

  - `1 <= n <= 104`
- 示例 1：

```
输入：n = 12
输出：3
解释：12 = 4 + 4 + 4
```

  示例 2：

```
输入：n = 13
输出：2
解释：13 = 4 + 9
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int numSquares(int n) {
        // dp[i] 是 和为i的完全平方数的最少数量
        int dp[n+1];

        // 初始化
        dp[0] = 0, dp[1] = 1;

        // 递推
        for(int i=2; i<=n; i++){
            // 最坏情况，全用 1
            dp[i] = i;
            // 依次看 i - j*j 的数量：一个j*j 再加上 i-j*j 的所需量
            for(int j=1; i-j*j>=0; j++)
                dp[i] = min(dp[i], 1 + dp[i-j*j]);
        }

        // 返回答案
        return dp[n];
    }
```

```
};
```

时间复杂度：$O(n\sqrt{n})$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 检测完全平方数
    bool isSquare(int a){
        int b = sqrt(a);
        return b*b == a;
    }

    // 检测是否满足 4^k * (8m+7)
    bool isFour(int a){
        while(a%4 == 0)  a /= 4;
        return a%8 == 7;
    }

    // 检测是否为两个完全平方数的和
    bool isTwo(int a){
        for(int b=1; b*b<a; b++)
            if(isSquare(a-b*b)) // 检测第三个数是不是完全平方数即可
                return true;
        // 否则不是
        return false;
    }

    // 数学是唯一真神：四平方和定理（答案只可能是 1/2/3/4）
    int numSquares(int n) {
        // 答案为 1：自身是完全平方数  复杂度 O(1)
        if(isSquare(n))  return 1;

        // 答案为 4：n = 4^k * (8m+7)  复杂度 O(log n)
        if(isFour(n))  return 4;

        // 答案为 2：直接从 0-根号n 查找  复杂度 O(sqrt n)
        if(isTwo(n))  return 2;

        // 答案为 3：最复杂，不用算，排除法已经出来了
        return 3;
    }
};
```

时间复杂度：$O(\sqrt{n})$.

空间复杂度：$O(1)$.

{% endtabs %}

85. 零钱兑换

- **题面：**
  给你一个整数数组 `coins` ，表示不同面额的硬币；以及一个整数 `amount` ，表示总金额。

  计算并返回可以凑成总金额所需的 **最少的硬币个数** 。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。

  你可以认为每种硬币的数量是无限的。

- 示例 1：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

  示例 2：

```
输入：coins = [2], amount = 3
输出：-1
```

  示例 3：

```
输入：coins = [1], amount = 0
输出：0
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int coinChange(vector<int>& coins, int amount) {
        int n = coins.size();
        // dp[i] 表示 合成总金额i所需的最少硬币数（用vector 赋值 0 更好，否则可能初始不是0！！！）
        vector<int> dp = vector<int>(amount+1, 0);

        // 初始化
        dp[0] = 0;
        for(int coin: coins)
            if(coin <= amount)  dp[coin] = 1;

        // 递推
        for(int i=1; i<=amount; i++){
            // 初始化没有
            if(!dp[i])  dp[i] = -1;
            // 逐个硬币分析
            for(int coin: coins){
                // 硬币可用
                if(coin <= amount && i-coin >= 0 && dp[i-coin] != -1)
                    // 第一次赋值
                    if(dp[i] == -1)  dp[i] = 1 + dp[i-coin];
```

```
                    // 更新
                    else  dp[i] = min(dp[i], 1 + dp[i-coin]);
                }
            }

            // 返回答案
            return dp[amount];
        }
};
```

时间复杂度：$O(Sn)$其中S是总金额，n是不同硬币数量.

空间复杂度：$O(S)$.

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int Max = amount + 1;
        vector<int> dp(amount + 1, Max);
        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for (int j = 0; j < (int)coins.size(); ++j) {
                if (coins[j] <= i) {
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

时间复杂度：$O(Sn)$.

空间复杂度：$O(S)$.

{% endtabs %}

86. 单词拆分

- **题面：**
  给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。

  注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

  提示：

  - `1 <= s.length <= 300`
  - `1 <= wordDict.length <= 1000`

- `1 <= wordDict[i].length <= 20`
- `s` 和 `wordDict[i]` 仅由小写英文字母组成
- `wordDict` 中的所有字符串 互不相同

- **示例 1：**

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以由 "apple" "pen" "apple" 拼接成。
     注意，你可以重复使用字典中的单词。
```

示例 3：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        int n = s.size();
        // dp[i] 表示s的前i个字符能否合法
        vector<bool> dp = vector<bool>(n+1, false);     // 初始为不合法

        // 初始化
        dp[0] = true;   // 空字符串合法

        // 用哈希表的字典，平均复杂度 O(1)
        unordered_set<string> uset;
        for(string& wd: wordDict)  uset.insert(wd);

        // 递推
        for(int i=1; i<=n; i++){
            // 前i个字符是否合法 = 前j个字符合法 + 后续字符合法
            for(int j=0; j<i; j++){
                if(dp[j] && uset.count(s.substr(j, i-j))){
                    dp[i] = true;
                    break;
                }
            }
        }

```

```
        // 返回答案
        return dp[n];
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(n)$.

{% endtabs %}

87. 最长递增子序列

- **题面：**
  给你一个整数数组 `nums` ，找到其中最长严格递增子序列的长度。

  子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

  提示：

  - `1 <= nums.length <= 2500`
  - `-104 <= nums[i] <= 104`

进阶：

- 你能将算法的时间复杂度降低到 $O(nlog(n))$ 吗?
- **示例 1：**

```
输入：nums = [10,9,2,5,3,7,101,18]
输出：4
解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
```

示例 2：

```
输入：nums = [0,1,0,3,2,3]
输出：4
```

示例 3：

```
输入：nums = [7,7,7,7,7,7,7]
输出：1
```

{% tabs 解法, -1 %}

```
// 不能直接使用单调栈，单调栈是保证最后一个元素一定入栈
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
```

```
        int n = nums.size();
        // dp[i] 表示在前i个元素中选，能满足条件的最大数量
        vector<int> dp = vector<int>(n, 0);

        // 初始化
        dp[0] = 1;

        // 递推
        for(int i=1; i<n; i++){
            // 实在不行就本身一个，肯定可以满足递增
            dp[i] = 1;
            // 看前 j 个能达到的最长，再连上本次的 nums[i]，就是新长度（连接前提是能单调递增）
            for(int j=0; j<i; j++)
                if(nums[j] < nums[i])
                    dp[i] = max(dp[i], dp[j]+1);
        }

        // 返回答案
        int dpmax = *max_element(dp.begin(), dp.end());
        return dpmax;
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(n)$.

若看不懂，请参考：深度解析

```
class Solution {
public:
    // 贪心+二分查找
    // 尽可能的使得上升的更慢，序列就会更长
    int lengthOfLIS(vector<int>& nums) {
        // p[i] 表示长度为i的序列保证上升得最慢时的那个末尾元素
        vector<int> p;          // 注：p本身并没有保存答案的完整序列！不是完整序列！
        p.push_back(nums[0]);

        // 递推
        for(int num: nums){
            // 更大时，直接接上
            if(num > p.back())
                p.push_back(num);
            // 更小时，把里面可以换得更小的位置换掉
            else{
                auto it = lower_bound(p.begin(), p.end(), num); // 返回第一个 >= num 的迭代器
                // 因为末尾比num大，所以一定会有，只需看是什么情况
                if(*it == num)  continue;   // 相等不用管
                else  p[it-p.begin()] = num;  // 确实num更小则替换
            }
        }
```

```
23
24          // 返回答案
25          return p.size();
26      }
27  };
```

时间复杂度：$O(n \log n)$二分查找是$O(\log n)$.

空间复杂度：$O(n)$.

{% endtabs %}

88. 乘积最大子数组

- **题面：**
  给你一个整数数组 `nums` ，请你找出数组中乘积最大的非空连续 **子数组**（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

  测试用例的答案是一个 `32-位` 整数。

  提示:

  - `1 <= nums.length <= 2 * 104`
  - `-10 <= nums[i] <= 10`
  - `nums` 的任何子数组的乘积都 **保证** 是一个 `32-位` 整数

- **示例 1:**

```
输入: nums = [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

示例 2:

```
输入: nums = [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
```

{% tabs 解法, -1 %}

```
1  class Solution {
2  public:
3      // 动态规划，两个dp
4      // 关键在于：产生答案的不一定是前序最大的，而有可能是前序最小的!
5      int maxProduct(vector<int>& nums) {
6          int n = nums.size();
7          // dpMax[i] 表示以数字i结尾的序列，尽可能大
8          // dpMin[i] 表示已数字i结尾的序列，尽可能小
```

```
        vector<int> dpMax = vector<int>(n);
        vector<int> dpMin = vector<int>(n);

        // 初始化
        dpMax[0] = nums[0], dpMin[0] = nums[0];

        // 递推
        for(int i=1; i<n; i++){
            int num = nums[i];              // 以i单独开辟序列
            int a = dpMax[i-1] * nums[i];   // 把i接上前面最大
            int b = dpMin[i-1] * nums[i];   // 把i接上前面最小
            // 得到以i结尾的数字序列的最大乘积和最小乘积
            dpMax[i] = max(num, max(a, b));
            dpMin[i] = min(num, min(a, b));
        }

        // 返回答案：所有可能i结尾的数字序列中，最大的乘积
        int ans = *max_element(dpMax.begin(), dpMax.end());
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    // 动态规划，简化，每次只会使用前序i-1的
    int maxProduct(vector<int>& nums) {
        int preMax = nums[0], preMin = nums[0];
        int ans = nums[0];

        for(int i=1; i<nums.size(); i++){
            int num = nums[i];
            int a = preMax * num;
            int b = preMin * num;
            preMax = max(num, max(a, b));
            preMin = min(num, min(a, b));
            // 需要伴随更新答案
            ans = max(ans, preMax);
        }

        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

89. 分割等和子集

- **题面：**
  给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

  提示：

  - `1 <= nums.length <= 200`
  - `1 <= nums[i] <= 100`

- **示例 1：**

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2：

```
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 问题转化：总和是sum，也就是挑出数字和为 sum/2 即可
    // 0-1 背包问题
    bool canPartition(vector<int>& nums) {
        int n = nums.size();
        int sum = accumulate(nums.begin(), nums.end(), 0);      // 来自<numric>
        // 预判 1：奇数和，则不可能实现
        if(sum & 1)  return false;
        int target = sum / 2;
        // 预判 2：有元素非常大，则不可能实现
        int maxNum = *max_element(nums.begin(), nums.end());
        if(maxNum > target)  return false;

        // dp[i][j] 表示前i个数字中，能否挑选出和为j
        vector<vector<bool>> dp(n, vector<bool>(target+1, false));

        // 初始化
        for(int i=0; i<n; i++)  dp[i][0] = true, dp[i][nums[i]] = true;

        // 递推:前i数字中挑选出和为j
```

```
        for(int i=1; i<n; i++){
            for(int j=0; j<=target; j++){
                // 1. 选用数字i达到和j
                // 2. 不用数字i达到和j
                if(j-nums[i] >= 0)
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i]];
                else
                    dp[i][j] = dp[i-1][j];
            }
        }

        // 返回答案
        return dp[n-1][target];
    }
};
```

时间复杂度：$O(n * target)$.

空间复杂度：$O(n * target)$.

```
class Solution {
public:
    // 问题转化：总和是sum，也就是挑出数字和为 sum/2 即可
    // 0-1 背包问题
    bool canPartition(vector<int>& nums) {
        int n = nums.size();
        int sum = accumulate(nums.begin(), nums.end(), 0);      // 来自<numric>
        // 预判 1：奇数和，则不可能实现
        if(sum & 1)  return false;
        int target = sum / 2;
        // 预判 2：有元素非常大，则不可能实现
        int maxNum = *max_element(nums.begin(), nums.end());
        if(maxNum > target)  return false;

        // 简化：dp[k] 表示目前能否挑出数字使得和为k
        vector<bool> dp = vector<bool>(target+1, false);
        dp[0] = true;
        for(int i=0; i<n; i++)
            for(int j=target; j-nums[i]>=0; j--)    // 注意：需要倒着来，否则会使得重复使用本次数字
                dp[j] = dp[j-nums[i]] || dp[j];

        // 返回答案
        return dp[target];
    }
};
```

时间复杂度：$O(n * target)$.

空间复杂度：$O(target)$.

{% endtabs %}

90. 最长有效括号

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

  提示：

  - `0 <= s.length <= 3 * 104`
  - `s[i]` 为 `'('` 或 `')'`
- **示例 1：**

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

示例 2：

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

示例 3：

```
输入：s = ""
输出：0
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int longestValidParentheses(string s) {
        int n = s.size();
        if(n == 0)  return 0;
        // dp[i] 表示以i结尾时，合法的最长长度
        vector<int> dp(n, 0);

        // 递推：用栈来存储后续可能会配对的左括号（记录位置）
        stack<int> iStk;
        for(int i=0; i<n; i++){
            // 左括号进栈，不会合法
            if(s[i] == '('){
                iStk.push(i);
                dp[i] = 0;
            }
            // 右括号，找栈中寻找最近的一个匹配
```

```
            else{
                // 匹配
                if(!iStk.empty()){
                    int left = iStk.top();
                    iStk.pop();
                    // 匹配上这个括号，同时加上这个括号前面的合法长度
                    if(left-1 >= 0)  dp[i] = i - left + 1 + dp[left-1];
                    else  dp[i] = i - left + 1;
                }
                // 无法匹配
                else
                    dp[i] = 0;
            }
        }

        // 返回答案
        int ans = *max_element(dp.begin(), dp.end());
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    // 巧解
    int longestValidParentheses(string s) {
        int n = s.size();
        int ans = 0;

        int left = 0, right = 0;    // 分别计算左右括号有多少个了
        // 向右遍历
        for(int i=0; i<n; i++){
            // 计数
            if(s[i] == '(')  left++;
            else  right++;
            // 若左右括号一样了，那就是合法长度
            if(left == right)  ans = max(ans, left + right);
            // 右括号不能更多，不可能合法了，重新归零
            else if(left < right)  left = 0, right = 0;
        }

        left = 0, right = 0;        // 记得重新归零
        // 向左遍历
        for(int i=n-1; i>=0; i--){
            // 计数
            if(s[i] == '(')  left++;
            else  right++;
            // 匹配
            if(left == right)  ans = max(ans, left + right);
```

```
            else if(left > right)  left = 0, right = 0;
        }

        // 返回答案
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

十五、多维动态规划

91. 不同路径

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

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

  问总共有多少条不同的路径?

  提示：

  - `1 <= m, n <= 100`
  - 题目数据保证答案小于等于 `2 * 109`
- **示例 1：**


```
输入：m = 3, n = 7
输出：28
```

示例 2：

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

示例 3：

```
输入：m = 7, n = 3
输出：28
```

示例 4：

```
输入：m = 3, n = 3
输出：6
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 多维动态规划
    int uniquePaths(int m, int n) {
        // dp[i][j] 表示到达 i,j 位置的路径数量
        vector<vector<int>> dp(m, vector<int>(n, 0));

        // 初始化
        dp[0][0] = 1;   // 在原点处

        // 递推
        // 位于 i,j 位置时，可以走向下/右
        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                if(i+1 < m)  dp[i+1][j] += dp[i][j];
                if(j+1 < n)  dp[i][j+1] += dp[i][j];
            }
        }

        // 返回答案
        return dp[m-1][n-1];
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(mn)$.

```cpp
class Solution {
public:
    // 数学真神登场：组合数学
    // 实际上就是 m-1 次向右走和 n-1 次向下走
    // 那就是 m+n-2 里面挑 m-1 为向右即可。即，C(m+n-2, m-1)
    // = { (m+n-2)(m+n-1)...n } / { (m-1)(m)...1 }
    int uniquePaths(int m, int n) {
        long long ans = 1;                          // 过程中可能会超出大小，改 long long
        for(int a=n, b=1; b<=m-1; a++, b++)         // 必须是从小到大乘，这样可以保证整除没有失误
            ans = ans * a / b;                      // 不能写ans *= a / b，这样会计算错误
        return ans;
    }
};
```

时间复杂度：$O(m)$.

空间复杂度：$O(1)$.

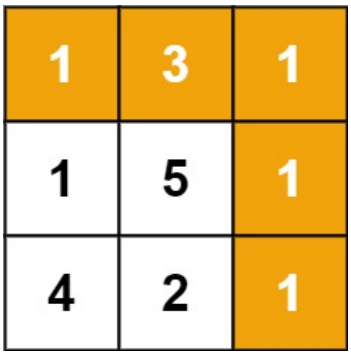
{% endtabs %}

92. 最小路径和

- **题面：**
  给定一个包含非负整数的 `m x n` 网格 `grid` ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

  说明：每次只能向下或者向右移动一步。

- **示例 1：**




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

示例 2：

```
输入：grid = [[1,2,3],[4,5,6]]
输出：12
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        // dp[i][j] 表示到达 i,j 位置所需的最小代价
        vector<vector<int>> dp(m, vector<int>(n, INT_MAX));

        // 初始化
        dp[0][0] = grid[0][0];

        // 递推：从当前位置走向右/向下，都可以更新
        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                if(i+1<m)
                    dp[i+1][j] = min(dp[i+1][j], dp[i][j] + grid[i+1][j]);
                if(j+1<n)
                    dp[i][j+1] = min(dp[i][j+1], dp[i][j] + grid[i][j+1]);
            }
        }

        // 返回答案
        return dp[m-1][n-1];
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(mn)$.

{% endtabs %}

93. 最长回文子串

- **题面：**
  给你一个字符串 `s`，找到 `s` 中最长的 **回文** 子串。

  提示：

  - `1 <= s.length <= 1000`
  - `s` 仅由数字和英文字母组成
- **示例 1：**

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

示例 2：

```
输入：s = "cbbd"
输出："bb"
```

{% tabs 解法, -1 %}

```
class Solution {
public:
    // 动态规划
    string longestPalindrome(string s) {
        int n = s.size();
        // dp[i][j] 表示 i..j 是否为回文串
        vector<vector<bool>> dp(n, vector<bool>(n, false));

        int l = 0, r =0;
        // 初始化：一个字符，是回文；两个字符，相同是回文
        for(int i=0; i<n; i++)  dp[i][i] = true;
        for(int i=0; i<n-1; i++)  if(s[i] == s[i+1])  dp[i][i+1] = true, l = i, r = i+1;

        // 递推：多个字符，两边两个字符相同，且中间是回文，则新组合是回文
        for(int len=3; len<=n; len++){
            for(int i=0; i+len-1<n; i++){   // 边界i+len-1在这里就不进循环，不要后续做无意义的判断！
                dp[i][i+len-1] = dp[i+1][i+len-2] && (s[i] == s[i+len-1]);  // 计算
                if(dp[i][i+len-1] && len > r-l+1)  l = i, r = i+len-1;       // 更新答案
            }
        }

        // 返回答案
        return s.substr(l, r-l+1);
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(n^2)$.

中心扩展法，时间复杂度不变，但是空间占用小。

参考链接

```
class Solution {
public:
    pair<int, int> expandAroundCenter(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
```

```
        }
        return {left + 1, right - 1};
    }

    string longestPalindrome(string s) {
        int start = 0, end = 0;
        for (int i = 0; i < s.size(); ++i) {
            auto [left1, right1] = expandAroundCenter(s, i, i);
            auto [left2, right2] = expandAroundCenter(s, i, i + 1);
            if (right1 - left1 > end - start) {
                start = left1;
                end = right1;
            }
            if (right2 - left2 > end - start) {
                start = left2;
                end = right2;
            }
        }
        return s.substr(start, end - start + 1);
    }
};
```

时间复杂度：$O(n^2)$.

空间复杂度：$O(1)$.

{% endtabs %}

94. 最长公共子序列

- **题面：**

  给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列** ，返回 `0` 。

  一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

  例如，`"ace"` 是 `"abcde"` 的子序列，但 `"aec"` 不是 `"abcde"` 的子序列。

  两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

  提示：

  - `1 <= text1.length, text2.length <= 1000`
  - `text1` 和 `text2` 仅由小写英文字符组成。

- **示例 1：**

```
输入：text1 = "abcde", text2 = "ace"
输出：3
解释：最长公共子序列是 "ace" ，它的长度为 3 。
```

示例 2：

```
输入：text1 = "abc", text2 = "abc"
输出：3
解释：最长公共子序列是 "abc" ，它的长度为 3 。
```

示例 3：

```
输入：text1 = "abc", text2 = "def"
输出：0
解释：两个字符串没有公共子序列，返回 0 。
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int longestCommonSubsequence(string text1, string text2) {
        int n1 = text1.size(), n2 = text2.size();
        // dp[i][j] 表示 text1前i个 和 text2前j个 的最长公共序列的长度
        vector<vector<int>> dp(n1+1, vector<int>(n2+1, 0));

        // 初始化：用 0 表示空（不能直接从单字符作为初始 0，会出问题）
        // dp[i][0] = dp [0][j] = 0 ，不用做，最初设置就是 0

        // 递推：前面公共长度 加上 当前字符的公共长度
        for(int i=1; i<=n1; i++){
            for(int j=1; j<=n2; j++){
                if(text1[i-1] == text2[j-1])    // 第i个字符是i-1下标
                    dp[i][j] = dp[i-1][j-1] + 1;
                else
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }

        // 返回答案
        return dp[n1][n2];
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(mn)$.

{% endtabs %}

95. 编辑距离

- **题面：**
  给你两个单词 `word1` 和 `word2`， 请返回将 `word1` 转换成 `word2` 所使用的最少操作数 。

  你可以对一个单词进行如下三种操作：

  - 插入一个字符
  - 删除一个字符
  - 替换一个字符

  提示：

  - `1 <= text1.length, text2.length <= 1000`
  - `text1` 和 `text2` 仅由小写英文字符组成。
- **示例 1：**

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

  示例 2：

```
输入：word1 = "intention", word2 = "execution"
输出：5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 动态规划
    int longestCommonSubsequence(string text1, string text2) {
        int n1 = text1.size(), n2 = text2.size();
        // dp[i][j] 表示 text1前i个 和 text2前j个 的最长公共序列的长度
        vector<vector<int>> dp(n1+1, vector<int>(n2+1, 0));

        // 初始化：用 0 表示空（不能直接从单字符作为初始 0，会出问题）
        // dp[i][0] = dp [0][j] = 0 ，不用做，最初设置就是 0

        // 递推：前面公共长度 加上 当前字符的公共长度
        for(int i=1; i<=n1; i++){
            for(int j=1; j<=n2; j++){
                if(text1[i-1] == text2[j-1])    // 第i个字符是i-1下标
```

```
                    dp[i][j] = dp[i-1][j-1] + 1;
                else
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }

        // 返回答案
        return dp[n1][n2];
    }
};
```

时间复杂度：$O(mn)$.

空间复杂度：$O(mn)$.

{% endtabs %}

十六、技巧

96. 只出现一次的数字

- **题面：**
  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 1：**

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

  示例 2：

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

  示例 3：

  ```
  输入：nums = [1]
  输出：1
  ```

{% tabs 解法, -1 %}

```
class Solution {
public:
    // 很自然的想法是哈希表（会用额外空间）
    int singleNumber(vector<int>& nums) {
        unordered_set<int> table;
```

```
        // 第一次进表，第二次删除
        for(int num: nums)
            if(table.find(num) != table.end())  table.erase(num);
            else  table.insert(num);
        // 最后剩余的就是答案
        int ans = nums[0];
        for(int num: table)  ans = num; // 就只会有一个
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```
class Solution {
public:
    // 异或操作：a^a = 0, a^0 = a，所以全部数字一起异或，最后结果刚好是答案
    int singleNumber(vector<int>& nums) {
        int ans = 0;
        for(int& num: nums)  ans ^= num;
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

97. 多数元素

- **题面：**
  给定一个大小为 `n` 的数组 `nums` ，返回其中的多数元素。多数元素是指在数组中出现次数 **大于** `⌊ n/2 ⌋` 的元素。

  你可以假设数组是非空的，并且给定的数组总是存在多数元素。

  进阶：尝试设计时间复杂度为 `O(n)`、空间复杂度为 `O(1)` 的算法解决此问题。

- **示例 1：**

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

  示例 2：

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

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 很自然的想法又是哈希表
    int majorityElement(vector<int>& nums) {
        unordered_map<int, int> table;
        int maxNum = nums[0];
        for(int& num: nums){
            // 计数
            if(table.find(num) != table.end())  table[num]++;
            else  table[num] = 1;
            // 更新答案
            if(table[num] > table[maxNum])  maxNum = num;
        }
        // 返回答案
        return maxNum;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 本题的众数是数量过半的，众数看做正粒子，其他看做负粒子
    // 正负粒子混合，正粒子多，最终是显正的
    int majorityElement(vector<int>& nums) {
        int ans, count = 0;
        for(int& num: nums){
            // 中性时，来一个数字，就看做正电
            if(count == 0)  ans = num, count++;
            // 根据是否为当前数字ans来判断电性增减
            else
                if(num == ans)  count++;
                else  count--;
        }
        // 最终存在的 ans 就是众数
        return ans;
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

98. 颜色分类

- **题面：**
  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。

  进阶：你能想出一个仅使用常数空间的一趟扫描算法吗?

- **示例 1：**

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

  示例 2：

```
输入：nums = [2,0,1]
输出：[0,1,2]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 直接作弊
    void sortColors(vector<int>& nums) {
        sort(nums.begin(), nums.end());
    }
};
```

时间复杂度：$O(n\log n)$不过似乎底层会自动优化为$O(n)$.

空间复杂度：$O(1)$.

```cpp
class Solution {
public:
    // 违规做法：计数重写
    void sortColors(vector<int>& nums) {
        // 计数
        int count[3] = {0, 0, 0};
        for(int& num: nums)  count[num]++;
        // 重写
        for(int i=0; i<nums.size(); i++)
            if(i < count[0])  nums[i] = 0;
            else if(i < count[0]+count[1])  nums[i] = 1;
```

```
            else  nums[i] = 2;
        }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 单指针交换头部
    void sortColors(vector<int>& nums) {
        // 先把所有 0 交换到头部
        int idx = 0;
        for(int i=0; i<nums.size(); i++)
            if(nums[i] == 0)
                swap(nums[i], nums[idx++]);
        // 再把所有 1 交换到 0 之后的头部
        for(int i=idx; i<nums.size(); i++)
            if(nums[i] == 1)
                swap(nums[i], nums[idx++]);
        // 所有 2 自然在末尾了
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

```
class Solution {
public:
    // 延续前面的思路，一边把 0 放到头部，一边把 2 放到尾部
    void sortColors(vector<int>& nums) {
        int n = nums.size();
        int head = 0, tail = n-1;
        // 注：数字 0 换到前面，是直接换，不会有什么影响（最多是把 1 换出来的，但是仍然正确）
        //     但是数字 2 换到末尾，是需要换完之后，又继续判断当前数字的，直到不再是 2 为止
        for(int i=0; i<n && i <= tail; i++){
            // 先循环判断 2：需要循环是因为可能后面交换过来的是 2
            while(i <= tail && nums[i] == 2)  swap(nums[i], nums[tail--]);
            // 再判断 0：只需一次是因为不可能会交换0 过来的，换过来的是 1
            if(nums[i] == 0)  swap(nums[i], nums[head++]);
        }
    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

99. 下一个排列

- **题面：**
  整数数组的一个 **排列** 就是将其所有成员以序列或线性顺序排列。

  例如，`arr = [1,2,3]`，以下这些都可以视作 `arr` 的排列：`[1,2,3]`、`[1,3,2]`、`[3,1,2]`、`[2,3,1]`。

  整数数组的 **下一个排列** 是指其整数的**下一个字典序更大**的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 **下一个排列** 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

  例如，`arr = [1,2,3]` 的下一个排列是 `[1,3,2]`。

  类似地，`arr = [2,3,1]` 的下一个排列是 `[3,1,2]`。

  而 `arr = [3,2,1]` 的下一个排列是 `[1,2,3]`，因为 `[3,2,1]` 不存在一个字典序更大的排列。

  给你一个整数数组 `nums`，找出 `nums` 的下一个排列。

  必须 [原地](#) 修改，只允许使用额外**常数空间**。

- **示例 1：**

```
输入：nums = [1,2,3]
输出：[1,3,2]
```

  示例 2：

```
输入：nums = [3,2,1]
输出：[1,2,3]
```

  示例 3：

```
输入：nums = [1,1,5]
输出：[1,5,1]
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 希望尽可能靠右的两个数字交换(因此从后往前找)
    // 这两个数字必须是小数a在前，大数b在后，且大数b尽可能小，只需大于小数a即可
    // 交换b到前面后，b之后的数字应该直接重新排为从小到大
    void nextPermutation(vector<int>& nums) {
        int n = nums.size();
        // 1. 找到第一个相邻升序，就是小数a
        int ai = -1;
        for(int i=n-2; i>=0; i--){
            if(nums[i] < nums[i+1]){
                ai = i;
```

```
                break;
            }
        }

        // 1.* 若全是降序，则直接全部排序为从小到大，结束
        if(ai == -1){
            reverse(nums.begin(), nums.end());  // 直接逆转就行
            return ;
        }

        // 2. 找到尽可能小的大数b
        int bi = -1;
        for(int i=n-1; i>=ai+1; i--)
            if(nums[i] > nums[ai]){
                bi = i;
                break;  // 第一个找到的肯定是最小的（因为前面找相邻升序，保证了ai..end是降序的）
            }

        // 3. 交换 a 和 b
        if(ai != -1 && bi != -1)  swap(nums[ai], nums[bi]);

        // 4. 把从 ai+1 开始之后的重新排序为从小到大，直接逆转就行
        reverse(nums.begin()+ai+1, nums.end());

    }
};
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}

100. 寻找重复数

- **题面：**
  给定一个包含 `n + 1` 个整数的数组 `nums` ，其数字都在 `[1, n]` 范围内（包括 `1` 和 `n`），可知至少存在一个重复的整数。

  假设 `nums` 只有 **一个重复的整数** ，返回 这个重复的数 。

  你设计的解决方案必须 **不修改** 数组 `nums` 且只用**常量级** `O(1)` **的额外空间**。

  注：`nums` 中 只有一个整数 出现 **两次或多次** ，其余整数均只出现 **一次**。

  进阶：

  - 如何证明 `nums` 中至少存在一个重复的数字?
  - 你可以设计一个线性级时间复杂度 `O(n)` 的解决方案吗?
- **示例 1：**

```
输入：nums = [1,3,4,2,2]
输出：2
```

示例 2：

```
输入：nums = [3,1,3,4,2]
输出：3
```

示例 3：

```
输入：nums = [3,3,3,3,3]
输出：3
```

{% tabs 解法, -1 %}

```cpp
class Solution {
public:
    // 违规：哈希表
    int findDuplicate(vector<int>& nums) {
        unordered_set<int> table;
        int ans;
        for(int& num: nums)
            if(table.find(num) == table.end())
                table.insert(num);
            else{
                ans = num;
                break;
            }

        return ans;
    }
};
```

时间复杂度：从$O(n)$到$O(n^2)$。由于哈希表的$find$函数平均是$O(1)$，但最差是$O(n)$.

空间复杂度：$O(n)$.

```cpp
class Solution {
public:
    // 快慢指针：把数组全看做是一个链表指针，意味着有两个位置执向同一个next，即存在环
    int findDuplicate(vector<int>& nums) {
        int slow = 0, fast = 0;

        // 第一阶段：相遇（相遇点并不一定是入环点）
        do{
            // slow 走一步，fast 走两步
            slow = nums[slow];
            fast = nums[nums[fast]];
        }while(slow != fast);
```

```
13
14          // 第二阶段：找到入环点
15          fast = 0;
16          while(slow != fast){
17              // 全走一步
18              slow = nums[slow];
19              fast = nums[fast];
20          }
21
22          // 返回答案
23          return slow;
24      }
25  };
26  /*
27  第一阶段的相遇，是一定会相遇的，因为进入环了，一快一慢，必然会追逐相遇
28  第二节点的入环点相遇，是推理证明出来的，两个指针均一步走。
29  */
```

时间复杂度：$O(n)$.

空间复杂度：$O(1)$.

{% endtabs %}