1.数组

前言：

- 不影响结果的情况下，无序数组先转换为有序数组
- 先不考虑空间复杂度
- 数组可以看作特殊的链表：index代表结点，index--->nums[in]

1.1 二分查找

1.1.1适用场合

- 题目要求

$$\mathcal{O}(\log_2 n) \tag{2}$$

- 有序数组查找元素（前缀和）
- 平方根

1.1.2例题

在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。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]`。

查找左边界`[l, mid], [mid+1, r]`

```
int findLeftBound(int l, int r) {
    while(l<r) {
        int mid = (l+r)/2;
        if(nums[mid]>=target) {
            r = mid;
        }
        else {
            l = mid+1;
        }
    }
    return l;//可能target在数组中不存在，需判断nums[l] == target
}
```

查找右边界`[l, mid-1], [mid, r]`

```
int findLeftBound(int l, int r) {
    while(l<r) {
        int mid = (l+r+1)/2;
        if(nums[mid]<=target) {
            l = mid;
        }
        else {
            r = mid-1;
        }
    }
    return l;//可能target在数组中不存在，需判断nums[l] == target
}
```

有效的完全平方数

难度简单239

给定一个 **正整数** `num` ，编写一个函数，如果 `num` 是一个完全平方数，则返回 `true` ，否则返回 `false` 。

进阶：不要 使用任何内置的库函数，如 `sqrt` 。

```
class Solution {
public:
    bool isPerfectSquare(int num) {
        int res = mySqrt(num);
        return res*res == num;
    }

    int mySqrt(int num) {
        long l = 0, r = num;
        while(l<r) {
            long mid = (l + r + 1) /2;
            //注意这里是平方根的定义
            if(mid*mid <= num) {
                l = mid;
            }
            else {
                r = mid - 1;
            }
        }

        return l;
    }
};
```

错误写法：下面的写法也能得出正确答案，不推荐

```
int mySqrt(double l, double r, double target) {
    while((r-l)>1e-7) {
        double mid = (l+r)/2;
        if(mid*mid < target) {
            l = mid;
        }
        else {
            r = mid;
        }
    }

    return r;//注意这里必须返回r
}
```

1.2双指针

1.2.1左右指针

有序数组的平方

给你一个按 **非递减顺序** 排序的整数数组 `nums`，返回 **每个数字的平方** 组成的新数组，要求也按 **非递减顺序** 排序。

- 分析问题
  - 数组问题：二分 或 双指针、自哈希
  - 选择双指针：左右指针
  - 定义 `int left = 0, right = nums.size()-1;`
  - `while(left <= right)` 中 `left` 和 `right` 指向的元素都是未处理的元素
  - 若写成 `while(left < right)`，则当有奇数个时，中间有一个元素未处理

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        vector<int> res;

        for(int i=0; i<nums.size(); i++) {
            nums[i] *= nums[i];
        }

        int left = 0;
        int right = nums.size()-1;

        while(left <= right) {
            if(nums[left] > nums[right]) {
                res.emplace_back(nums[left]);
                left++;
```

```
            }
            else {
                res.emplace_back(nums[right]);
                right--;
            }
        }

        return res;

    }
};
```

1.2.2快慢指针

移除元素

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 `O(1)` 额外空间并 **原地** **修改输入数组**。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

- 分析问题：
  - 题目没说 数组是否有序，但此题不用排序
  - 数组考虑使用 二分 和 双指针 、 自哈希
  - 第一个元素有可能被移除，故 `slow` 从 `0` 开始
  - `slow` 表示 `<slow` 都符合条件， `return slow`
  - `for` 循环中用 `fast` 指针找到符合条件可以留下的元素，从而移动 `slow`
  - `nums[slow] = nums[fast]` //直接覆盖不符合条件元素
  - `swap(nums[slow], nums[fast])` //保留不符合条件元素到数组后面

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        //没有必要排序

        //第一个元素有可能被移除，故slow从0开始
        int slow = 0;
        for(int fast = 0; fast<nums.size(); fast++) {
            if(nums[fast] != val) {
                nums[slow] = nums[fast];
                slow++;
            }
        }

        return slow;
    }
```

```
};
```

移动零（练习思考过程）

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。

- 分析问题
  - 数组是否有序
  - 数组考虑使用二分和双指针、自哈希
  - 移除 `0`,第 `0` 个位置元素可能被移除，故 `slow` 从 `0` 开始。
  - 所有 `0` 移动到数组的末尾，使用 `swap(nums[slow], nums[fast])`

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

        int slow = 0;
        for(int fast = 0; fast<nums.size(); fast++) {
            if(nums[fast] != 0) {
                swap(nums[slow], nums[fast]);
                slow++;
            }
        }
    }
};
```

寻找重复数-环形链表

给定一个包含 `n + 1` 个整数的数组 `nums`，其数字都在 `1` 到 `n` 之间（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**，找出 **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。

- 分析问题
  - 不能修改原数组，常数级空间复杂度
  - `n+1` 个数
  - 数组下标 `0, 1, 2, ....., n`
  - 数字范围 `1, 2, ....., n`

```cpp
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int slow  = 0;
```

```
        int fast = 0;
        while(true) {
            slow = nums[slow];
            fast = nums[nums[fast]];
            if(slow == fast) {
                break;
            }
        }

        slow = 0;
        while(slow != fast) {
            slow = nums[slow];
            fast = nums[fast];
        }

        return slow;
    }
};
```

1.2.3滑动窗口

长度最小的子数组

给定一个含有 `n` 个正整数的数组和一个正整数 `target` 。

找出该数组中满足其和 `≥ target` 的长度最小的 **连续子数组** `[numsl, numsl+1, ..., numsr-1, numsr]` ，并返回其长度。如果不存在符合条件的子数组，返回 `0` 。

- 分析问题
  - 寻找连续区间中一段满足特定条件的连续子区间：滑动窗口
  - 滑动窗口本质还是快慢指针
  - `fast` 指针负责扩大区间直到满足特定条件。代表 `windows` 窗口右边界。
  - `slow` 指针在 `windows` 达到条件时，收缩 `windows` 窗口大小。代表 `windows` 窗口左边界。
  - 收缩 `windows` 窗口时，用全局变量记录下满足特定条件的最小窗口值。
  - 窗口 `windows` 定义为 `[slow, fast]` ,故 `slow` 初始化为 `0` 。
  - 问题不一定有返回值。

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        //
        int slow = 0;
        int sum = 0;
        int minWindowSize = INT_MAX;
        for(int fast = 0; fast<nums.size(); fast++) {
            //扩大窗口导致代表窗口内状态的某个变量变化
            sum += nums[fast];
            while(sum >= target) {
```

```
                minWindowSize = min(minWindowSize, fast-slow+1);
                //收缩窗口导致代表窗口内状态的某个变量变化
                sum -= nums[slow];
                slow++;
            }
        }

        return minWindowSize == INT_MAX ? 0 : minWindowSize;
    }
};
```

最小覆盖子串

难度困难1336收藏分享切换为英文接收动态反馈

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"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

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。
```

提示：

- `1 <= s.length, t.length <= 105`
- `s` 和 `t` 由英文字母组成

- 分析问题
  - 问题寻找连续区间中一段满足特定条件的连续子区间：滑动窗口
  - 难点：收缩窗口的条件难写，不是和一个数target比较，而是和一个 `string t` 比较
  - 我们需要把窗口 `windows` 的状态和 `string t` 都转换为另一个量进行比较。
  - 增加变量表示窗口的状态。

```cpp
class Solution {
public:
    string minWindow(string s, string t) {
        unordered_map<char, int> need;
        unordered_map<char, int> window;
        int valid = 0;
        //和string t的比较转为测试need
        for(auto& ch: t) {
            need[ch]++;
        }

        int start = 0, end = -1;
        int result = INT_MAX;
        int sublength = 0;

        int slow=0;
        for(int fast=0; fast<s.size(); fast++) {

            if(need.count(s[fast])) {
                window[s[fast]]++;
                if(window[s[fast]]==need[s[fast]]) {
                    valid++;
                }
            }

            while(valid == need.size()) {
                sublength = fast-slow+1;
                if(sublength < result) {
                    result = sublength;
                    start = slow;
                    end = fast;
                }

                if(window.count(s[slow])) {
                    window[s[slow]]--;
                    if(window[s[slow]]<need[s[slow]]) {
                        valid--;
                    }
                }
```

```
                slow++;
            }
        }
        cout << start << " " << end << endl;
        return s.substr(start, end-start+1);
    }
};
```

水果成篮

在一排树中，第 `i` 棵树产生 `tree[i]` 型的水果。
你可以**从你选择的任何树开始**，然后重复执行以下步骤：

1. 把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
2. 移动到当前树右侧的下一棵树。如果右边没有树，就停下来。

请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤 2，然后返回步骤 1，然后执行步骤 2，依此类推，直至停止。

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一种类型的水果。

用这个程序你能收集的水果树的最大总量是多少？

示例 1：

```
输入：[1,2,1]
输出：3
解释：我们可以收集 [1,2,1]。
```

示例 2：

```
输入：[0,1,2,2]
输出：3
解释：我们可以收集 [1,2,2]
如果我们从第一棵树开始，我们将只能收集到 [0, 1]。
```

示例 3：

```
输入：[1,2,3,2,2]
输出：4
解释：我们可以收集 [2,3,2,2]
如果我们从第一棵树开始，我们将只能收集到 [1, 2]。
```

示例 4：

```
输入：[3,3,3,1,2,1,1,2,3,3,4]
输出：5
解释：我们可以收集 [1,2,1,1,2]
如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 棵水果树。
```

提示：

- `1 <= tree.length <= 40000`
- `0 <= tree[i] < tree.length`

- 分析问题
  - 数组问题：二分和双指针、自哈希
  - 返回从任意树开始的一段只包含两个树的最大子区间
  - 只包含两颗树的最大子区间-->包含`>2`颗树的最小子区间

```cpp
class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        int target = 2;//

        unordered_map<int, int> hash;

        int res = INT_MIN;


        int slow = 0;
        for(int fast = 0; fast<fruits.size(); fast++) {
            hash[fruits[fast]]++;
            while(hash.size()>target) {
                hash[fruits[slow]]--;
                if(hash[fruits[slow]] == 0) {
                    hash.erase(fruits[slow]);
                }
                slow++;
            }

            res = max(res, fast-slow+1);

        }

        return res == INT_MIN ? 0 : res;
    }
};
```

1.3自哈希

降低空间复杂度

缺失的第一个正数

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。

- 分析问题
  - 不考虑空间复杂度，可以直接遍历一遍数组，记录存在的数字。然后从1开始测试缺失的第一个正数。
  - 时间复杂度为 `O(n)` 并且只使用常数级别额外空间
    - 时间复杂度为 `O(n)` 的备选方案：二分、自哈希
    - 时间复杂度为 `O(n)` 并且只使用常数级别额外空间的备选方案：自哈希
    - 前置知识：未出现的第一个正数要么是 `1~n` ，要么是 `n+1` 。
    - `nums[i]` 当作数组下标，把 `nums[nums[i]]` 做标记，表示 `nums[i]` 存在。
    - `nums[0...n]` 中未被标记的就是缺失的非负数
    - 细节：
      - 与 `nums[i]` 相等的值存在多个， `nums[nums[i]]` 会被多次标记
      - 当数组中出现 `1~n` 时，数组下标 `0` 处未被标记，无法得出是n没出现（越界，无法记录），还是 `n+1` 没出现
      - 根据前置知识：我们需要把 `1~n` 和 `0~n-1` 相互映射
      - 出现负数，我们没法判断它到底是被标记的，还是原本就是一个负数
      - 标记过程中，可能把未访问到的数标记为负
    - 对于一个长度为 N的数组，其中没有出现的最小正整数只能在` [1, N+1][1,N+1] 中。这是因为如果 [1, N][1,N] 都出现了，那么答案是 N+1，否则答案是 [1, N][1,N] 中没有出现的最小正整数。这样一来，我们将所有在 [1, N][1,N] 范围内的数放入哈希表，也可以得到最终的答案。而给定的数组恰好长度为 N，这让我们有了一种将数组设计成哈希表的思路：

      我们对数组进行遍历，对于遍历到的数 x，如果它在 [1, N][1,N] 的范围内，那么就将数组中的第x-1 个位置（注意：数组下标从0开始）打上「标记」。在遍历结束之后，如果所有的位置都被打上了标记，那么答案是 N+1，否则答案是最小的没有打上标记的位置加 1。

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

```
10
11          for(int i=0; i<nums.size(); i++) {
12              //此处容易出错，取出来的数可能为负，因为之前可能改过
13              int number = abs(nums[i]);
14              if(number>=1 && number <=n) {
15                  int index = number - 1;
16                  nums[index] = -abs(nums[index]);
17              }
18          }
19
20
21          for(int i=0; i<nums.size(); i++) {
22              if(nums[i]>0) {
23                  return i+1;
24              }
25          }
26          return n+1;
27      }
28  };
```

```
1   class Solution {
2   public:
3       int firstMissingPositive(vector<int>& nums) {
4           int n = nums.size();
5           for(int i=0; i<nums.size(); i++) {
6               if(nums[i]<=0) {
7                   错误写法:
8                   nums[i] = 0;
9                   //[1,2,0]
10                  //[-1, -2, 0]//输出4
11                  //[-1, -2, 4]//输出3
12              }
13          }
14          ......
15      }
16  };
```

丢失的数字

给定一个包含 `[0, n]` 中 `n` 个数的数组 `nums` ，找出 `[0, n]` 这个范围内没有出现在数组中的那个数。

```
1   class Solution {
2   public:
3       int missingNumber(vector<int>& nums) {
4
5           int n = nums.size();
```

```
6
7           //数组大小为n
8           //数字范围在[0, n]
9           [2,0]---未出现1,但输出2
10          //转换成[1, n+1]//表示第几个;
11          //寻找[1, n+1]缺失的数
12          for(int i=0; i<n; i++) {
13              nums[i]++;
14          }
15
16          for(int i=0; i<n; i++) {
17              int number = abs(nums[i]);
18              if(number>=1 && number <=n) {
19                  int index = number - 1;
20                  nums[index] = -abs(nums[index]);
21              }
22          }
23
24          for(int i=0; i<n; i++) {
25              if(nums[i]>0) {
26                  return (i+1)-1;
27              }
28          }
29          return (n+1)-1;
30
31      }
32  };
33
```

1.4置换

缺失的第一个正数

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。

方法二：置换

除了打标记以外，我们还可以使用置换的方法，将给定的数组「恢复」成下面的形式：

> 如果数组中包含 $x \in [1, N]$，那么恢复后，数组的第 $x - 1$ 个元素为 x。

在恢复后，数组应当有 `[1, 2, ..., N]` 的形式，但其中有若干个位置上的数是错误的，每一个错误的位置就代表了一个缺失的正数。以题目中的示例二 `[3, 4, -1, 1]` 为例，恢复后的数组应当为 `[1, -1, 3, 4]`，我们就可以知道缺失的数为 2。

那么我们如何将数组进行恢复呢？我们可以对数组进行一次遍历，对于遍历到的数 $x = nums[i]$，如果 $x \in [1, N]$，我们就知道 x 应当出现在数组中的 $x - 1$ 的位置，因此交换 $nums[i]$ 和 $nums[x - 1]$，这样 x 就出现在了正确的位置。在完成交换后，新的 $nums[i]$ 可能还在 $[1, N]$ 的范围内，我们需要继续进行交换操作，直到 $x \notin [1, N]$。

注意到上面的方法可能会陷入死循环。如果 $nums[i]$ 恰好与 $nums[x - 1]$ 相等，那么就会无限交换下去。此时我们有 $nums[i] = x = nums[x - 1]$，说明 x 已经出现在了正确的位置。因此我们可以跳出循环，开始遍历下一个数。

由于每次的交换操作都会使得某一个数交换到正确的位置，因此交换的次数最多为 N，整个方法的时间复杂度为 $O(N)$。

```cpp
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int n = nums.size();
        for(int i=0; i<nums.size(); i++) {
            //此处必须是while,if会出错
            while(nums[i]>=1 && nums[i] <=n && nums[i]!=nums[nums[i]-1]) {
                swap(nums[i], nums[nums[i]-1]);
            }
        }

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

        return n+1;
    }
};
```

2.字符串

3.链表

前言：

- 先不考虑空间复杂度，链表可以转换为数组求解

1.1遍历

1.2递归

本质是分治

1.2.1思路

- 定义好**递归函数的含义（参数，返回值）**，并牢记
- 题目给的**递归函数可能参数不够**，需要自己定义另一个递归函数
- 思考**Base条件**，**尝试缩小问题规模**。

1.2.2例题

反转链表

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。

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(head == nullptr || head->next == nullptr) {
            return head;
        }

        ListNode* last = reverseList(head->next);
        head->next->next = head;
        head->next = nullptr;

        return last;
    }
};
```

1.3双指针

1.3.1前后指针

移除链表元素

给你一个链表的头节点 `head` 和一个整数 `val` ，请你删除链表中所有满足 `Node.val == val` 的节点，并返回 **新的头节点**

- 分析问题
  - 头结点是否可能改变
  - 本题中头节点可能被删除，需要必须设置虚拟头节点。其他可设可设。
  - 设置ptr指针，本题是删除，故需要指向待删除结点的前一个结点

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {

        ListNode* dummy = new ListNode(val);
        dummy->next = head;

        ListNode* ptr = dummy;
        ListNode* cur = head;//当前遍历结点

        while(cur != nullptr) {
            if(cur->val == val) {
                ptr->next = cur->next;
                cur = ptr->next;
            }
            else {
                ptr = cur;
                cur = cur->next;
            }
        }

        return dummy->next;

    }
};
```

反转链表

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。

- 此处为迭代写法

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = nullptr;
        ListNode* cur = head;
```

```
6
7           while(cur != nullptr) {
8               ListNode* tmp = cur->next;
9               cur->next = pre;
10              pre =cur;
11              cur = tmp;
12          }
13
14          return pre;
15      }
16  };
```

1.3.2快慢指针

链表中倒数第k个节点

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

例如，一个链表有 `6` 个节点，从头节点开始，它们的值依次是 `1、2、3、4、5、6` 。这个链表的倒数第 `3` 个节点是值为 `4` 的节点。

- 分析问题
  - 快慢指针容易出错，考虑缩小问题规模

```
1   class Solution {
2   public:
3       ListNode* getKthFromEnd(ListNode* head, int k) {
4
5           ListNode* dummy = new ListNode(0);
6           dummy->next = head;
7
8           ListNode* slow = dummy;
9           ListNode* fast = head;
10
11          while(--k>0) {
12              fast = fast->next;
13          }
14
15          while(fast != nullptr) {
16              slow = slow->next;
17              fast = fast->next;
18          }
19
20          return slow;
21      }
22  };
```

链表的中间结点

给定一个头结点为 `head` 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;

        while(fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }

        return slow;
    }
};
```

环形链表 II

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

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。**注意，`pos` 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

说明：不允许修改给定的链表。

进阶：

- 你是否可以使用 `O(1)` 空间解决此题？

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;

        bool hasCycle = false;

        while(fast!=nullptr && fast->next!=nullptr) {
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) {
                hasCycle = true;
                break;
            }
```

```
        }

        if(!hasCycle) return nullptr;;

        slow = head;
        while(slow!=fast) {
            slow = slow->next;
            fast = fast->next;
        }

        return slow;
    }
};
```

1.3.3双指针（多个链表）

分治：直至问题可以求解

合并两个有序链表

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。

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* ptr = dummy;

        while(l1 != nullptr && l2 != nullptr) {
            if(l1->val < l2->val) {
                ptr->next = l1;
                ptr = ptr->next;
                l1 = l1->next;
            }
            else {
                ptr->next = l2;
                ptr = ptr->next;
                l2 = l2->next;
            }
        }

        if(l1 != nullptr) {
            ptr->next = l1;
        }

        if(l2 != nullptr) {
            ptr->next = l2;
        }

```

```
        return dummy->next;
    }
};
```

合并K个升序链表

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。

```
ListNode* mergeKLists(vector<ListNode*>& lists) {
    if(lists.size()==0) return nullptr;

    ListNode* l1 = lists[0];
    for(int i=1; i<lists.size(); i++) {
        l1 = mergeTwoLists(l1, lists[i]);
    }

    return l1;
}
```

4.树

4.1遍历

4.1.1层序遍历

```
/**
 * 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<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res={};
        if(root == nullptr) return res;

        queue<TreeNode*> q;
        q.push(root);

        while(!q.empty()) {
            int len = q.size();
            vector<int> tmp;
            for(int i=0; i<len; i++) {
                TreeNode* cur = q.front();
                q.pop();
                tmp.push_back(cur->val);
                if(cur->left != nullptr) q.push(cur->left);
                if(cur->right != nullptr) q.push(cur->right);
            }

            res.push_back(tmp);
        }

        return res;

    }
};
```

4.1.2中序遍历

```
void traverseInOrder(TreeNode* root) {
    if(root != nullptr) {
        traverseInOrder(root->left);
        此处可以
            1.访问当前结点
            2.构造结点
        traverseInOrder(root->right);
    }
}
```

4.1.3中序遍历的应用

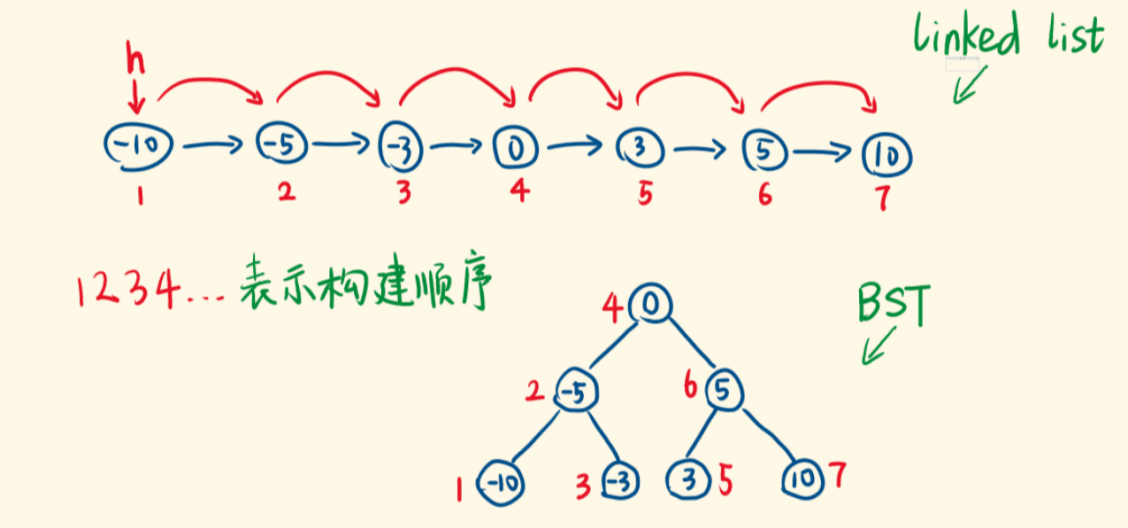
有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

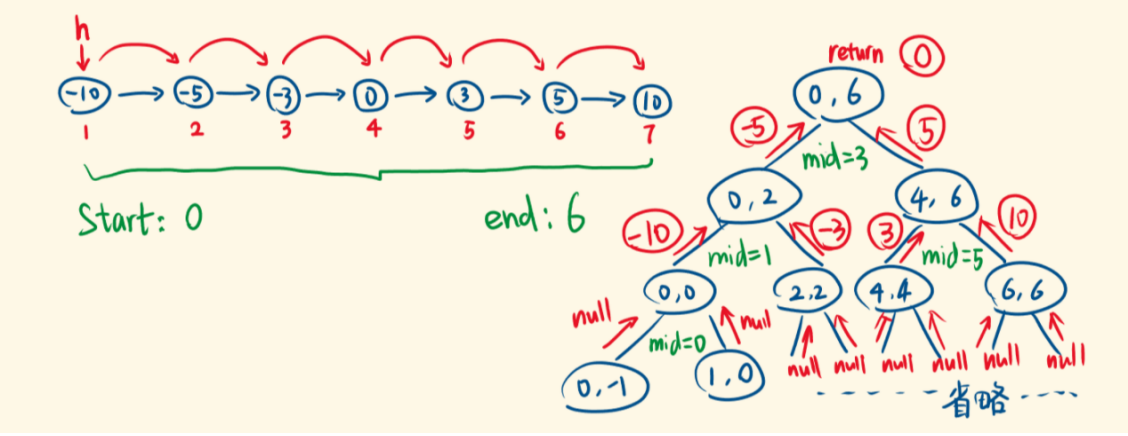
本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

方法3： 中序遍历策略带来的优化

- 方法1每次获取数组中点：O(1)O(1)，方法2每次获取链表中点：O(N)O(N)，所以更慢。
- 其实直接获取链表头结点：O(1)O(1)，不如直接构建它吧！它对应 BST 最左子树的根节点。
- 于是我们先构建左子树，再构建根节点，再构建右子树。——遵循中序遍历。
- 其实，BST 的中序遍历，打印的节点值正是这个有序链表的节点值顺序。如下图，维护指针 h，从头结点开始，用 h.val 构建节点，构建一个，指针后移一位。



- 求出链表结点总个数，用于每次二分求出链表的中点。
- 为什么要这么做，因为我们构建的节点值是：从小到大，我们希望在递归中处理节点的顺序和链表结点顺序一一对应
- 看看下图的递归树，感受一下二分法怎么做到的。用二分后的左链递归构建左子树，然后用 h.val 创建节点，接上创建好的左子树，再用右链构建右子树，再接上。
- 递归中会不断进行二分，直到无法划分就返回 null，即来到递归树的底部h.val 创建完结点后，h 指针就后移，锁定出下一个要构建的节点值……




```
class Solution {
    ListNode* cur;

```

```
public:
    TreeNode* sortedListToBST(ListNode* head) {
        cur = head;

        int len = 0;

        while(head != nullptr) {
            len++;
            head = head->next;
        }


        return buildTree(1, len);

    }

    TreeNode* buildTree(int start, int end) {
        if(start > end) return nullptr;

        int mid = start + (end-start)/2;

        TreeNode* leftTree = buildTree(start, mid-1);//构造左子树
        TreeNode* root = new TreeNode(cur->val);
        cur = cur->next;
        TreeNode* rightTree = buildTree(mid+1, end);//构造右子树

        root->left = leftTree;
        root->right = rightTree;

        return root;
    }
};
```

4.2递归

本质是分治

4.2.1不同的二叉搜索树 II

给你一个整数 `n` ，请你生成并返回所有由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的不同 **二叉搜索树** 。可以按 **任意顺序** 返回答案。

思路与算法

二叉搜索树关键的性质是根节点的值大于左子树所有节点的值，小于右子树所有节点的值，且左子树和右子树也同样为二叉搜索树。因此在生成所有可行的二叉搜索树的时候，假设当前序列长度为 n，如果我们枚举根节点的值为 i，那么根据二叉搜索树的性质我们可以知道左子树的节点值的集合为 [1…i−1]，右子树的节点值的集合为 [i+1…n]。而左子树和右子树的生成相较于原问题是一个序列长度缩小的子问题，因此我们可以想到用递

归的思想来解决这道题目。

```cpp
class Solution {
public:
    vector<TreeNode*> generateTrees(int start, int end) {
        if (start > end) {
            return { nullptr };
        }
        vector<TreeNode*> allTrees;
        // 枚举可行根节点
        for (int i = start; i <= end; i++) {
            // 获得所有可行的左子树集合
            vector<TreeNode*> leftTrees = generateTrees(start, i - 1);

            // 获得所有可行的右子树集合
            vector<TreeNode*> rightTrees = generateTrees(i + 1, end);

            // 从左子树集合中选出一棵左子树，从右子树集合中选出一棵右子树，拼接到根节点上
            for (auto& left : leftTrees) {
                for (auto& right : rightTrees) {
                    TreeNode* currTree = new TreeNode(i);
                    currTree->left = left;
                    currTree->right = right;
                    allTrees.emplace_back(currTree);
                }
            }
        }
        return allTrees;
    }

    vector<TreeNode*> generateTrees(int n) {
        if (!n) {
            return {};
        }
        return generateTrees(1, n);
    }
};
```

4.2.2不同的二叉搜索树

> 题目要求是计算不同二叉搜索树的个数。为此，我们可以定义两个函数：
>
> G(n): 长度为 nn 的序列能构成的不同二叉搜索树的个数。
>
> F(i, n): 以 i 为根、序列长度为 nn 的不同二叉搜索树个数 (1≤i≤n)。

4.2.3验证二叉搜索树

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。
- [x] 所有左子树和右子树自身必须也是二叉搜索树。

4.2.4从前序与中序遍历序列构造二叉树

给定一棵树的前序遍历 `preorder` 与中序遍历 `inorder` 。请构造二叉树并返回其根节点。

```
/**
 * 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 preL = 0, preR = preorder.size() - 1;
        int inL = 0, inR = inorder.size() - 1;
        return buildTree(preorder, preL, preR, inorder, inL, inR);
    }

    TreeNode* buildTree(vector<int>& preorder, int preL, int preR, vector<int>& inorder, int inL, int inR) {
        if(preL > preR || inL > inR) return nullptr;
```

```
22
23          TreeNode* root = new TreeNode(preorder[preL]);
24
25          int index = -1;
26          for(int i = inL; i <= inR; i++) {
27              if(inorder[i] == root->val) {
28                  index = i;
29                  break;
30              }
31          }
32          int left_num = index - inL;
33          int right_num = inR - index;
34          root->left = buildTree(preorder, preL + 1, (preL + 1) + left_num - 1,
    inorder, inL, index - 1);
35          root->right = buildTree(preorder, preR - right_num + 1, preR, inorder,
    inddex + 1, inR);
36
37          return root;
38      }
39  };
```

4.2.5有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

前言

将给定的有序链表转换为二叉搜索树的第一步是确定根节点。由于我们需要构造出平衡的二叉树，因此比较直观的想法是让根节点左子树中的节点个数与右子树中的节点个数尽可能接近。这样一来，左右子树的高度也会非常接近，可以达到高度差绝对值不超过 1的题目要求。

如何找出这样的一个根节点呢？我们可以找出链表元素的中位数作为根节点的值。

这里对于中位数的定义为：如果链表中的元素个数为奇数，那么唯一的中间值为中位数；如果元素个数为偶数，那么唯二的中间值都可以作为中位数，而不是常规定义中二者的平均值。

解法一：分治

```
1   /**
2    * Definition for singly-linked list.
3    * struct ListNode {
4    *     int val;
5    *     ListNode *next;
6    *     ListNode() : val(0), next(nullptr) {}
7    *     ListNode(int x) : val(x), next(nullptr) {}
8    *     ListNode(int x, ListNode *next) : val(x), next(next) {}
9    * };
10   */
```

```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:
    TreeNode* sortedListToBST(ListNode* head) {

        return buildTree(head);
    }

    TreeNode* buildTree(ListNode* head) {
        if(head == nullptr) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* preSlow = nullptr;

        while(fast!=nullptr && fast->next!=nullptr) {
            preSlow = slow;
            slow = slow->next;
            fast = fast->next->next;
        }

        TreeNode* root = new TreeNode(slow->val);

        if(preslow != nullptr) {
            preSlow->next = nullptr;
            root->left = build(head);
        }

        root->right = build(slow->next);

        return root;
    }
};
```

4.2.6二叉树的最小深度

注：二叉树的最大深度（高度）很容易求解

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if(root == nullptr) return 0;
        else if(root->left == nullptr && root->right == nullptr) return 1;
        else if(root->left == nullptr) return minDepth(root->right) + 1;
        else if(root->right == nullptr) return minDepth(root->left) + 1;
        else return min(minDepth(root->left), minDepth(root->right)) + 1;

    }
};
```

4.2.7路径总和

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

叶子节点 是指没有子节点的节点

```
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(root == nullptr) return false;
        if(root->left==nullptr && root->right==nullptr && root->val == targetSum)
{
            return true;
        }
        else {
            return hasPathSum(root->left, targetSum - root->val) || \
                hasPathSum(root->right, targetSum - root->val);
        }
    }
};
```

5.哈希表

环形链表 II

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

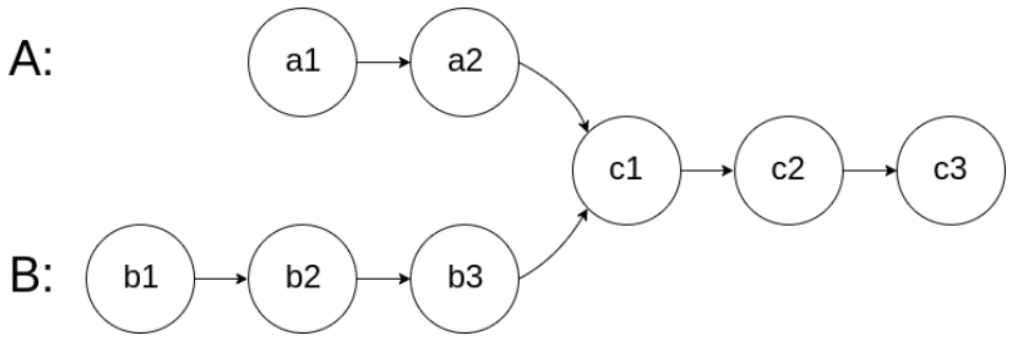
为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。**注意，** `pos` **仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        unordered_set<ListNode *> visited;
        while (head != nullptr) {
            if (visited.count(head)) {
                return head;
            }
            visited.insert(head);
            head = head->next;
        }
        return nullptr;
    }
};
```

相交链表

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。如果两个链表没有交点，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** 。

两个列表的最小索引总和

难度简单119收藏分享切换为英文接收动态反馈

假设Andy和Doris想在晚餐时选择一家餐厅，并且他们都有一个表示最喜爱餐厅的列表，每个餐厅的名字用字符串表示。

你需要帮助他们用**最少的索引和**找出他们**共同喜爱的餐厅**。 如果答案不止一个，则输出所有答案并且不考虑顺序。 你可以假设总是存在一个答案。

```
class Solution {
public:
    vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {

        unordered_map<string, int> record;
        unordered_map<string, int> hash;
        for(int i=0; i<list1.size(); i++) {
            hash[list1[i]] += i;
        }

        for(int i=0; i<list2.size(); i++) {
            if(!hash.count(list2[i])) continue;
            record[list2[i]] = hash[list2[i]] + i;
        }
        int minVal = INT_MAX;
        for(auto& tmp: record) {
            if(tmp.second<minVal) {
                minVal = tmp.second;
            }
        }

        vector<string> res;

        for(auto& tmp: record) {
            if(tmp.second == minVal) {
                res.push_back(tmp.first);
            }
        }

        return res;
    }
};
```

6.位运算

只出现一次的数字

给定一个**非空**整数数组，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。

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for(int i = 0; i<nums.size(); i++) {
            res ^= nums[i];
        }

        return res;
    }
};
```

只出现一次的数字 II

给你一个整数数组 `nums` ，除某个元素仅出现 **一次** 外，其余每个元素都恰出现 **三次** 。请你找出并返回那个只出现了一次的元素。

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        i |= (1<<j);设置
        i &= ~(1<<j);清除
        i & (1<<j);检测;

        int cnt[32] = {0};

        for(int i=0; i<nums.size(); i++) {
            for(int j=0; j<32; j++) {
                if(nums[i] & (1<<j)) {
                    cnt[j]++;
                }
            }
        }

        int res = 0;

        for(int i=0; i<32; i++) {
            cnt[i]%=3;
            if(cnt[i] == 1) {
                res |= (1<<i);
            }
            else {
                res &= ~(1<<i);//此语句可以删掉
            }
```

```
28
29          }
30
31          return res;
32
33
34      }
35  };
```

7.优先队列

8.模拟

螺旋矩阵

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。

- 分析问题
  - 环形向内打印
  - 特殊情况：只有一列和只有一行
  - `[(top, left), (top,  right)]`
  - `[(top+1, right), (bottom, right)]`
  - `[(bottom, right-1), (bottom, left))`
  - `[(bottom, left), (top, left))`

```
1   class Solution {
2   public:
3       vector<int> spiralOrder(vector<vector<int>>& matrix) {
4
5           int rows = matrix.size();
6           int cols = matrix[0].size();
7
8           int top = 0, bottom = rows-1;
9           int left = 0, right = cols-1;
10
11          vector<int> res;
12
13          while(top<=bottom && left <=right) {//只有一列或一行也打印
14              for(int j = left; j<=right; j++) {
15                  res.emplace_back(matrix[top][j]);
```

```
            }

            for(int i=top+1; i<=bottom; i++) {
                res.emplace_back(matrix[i][right]);
            }

            if(top<bottom && left<right) {//大于一列或一行才打印
                for(int j=right-1; j>left; j--) {
                    res.emplace_back(matrix[bottom][j]);
                }

                for(int i=bottom; i>top; i--) {
                    res.emplace_back(matrix[i][left]);
                }
            }

            left++;
            right--;
            top++;
            bottom--;

        }
        return res;
    }
};
```

9.DFS+BFS

10.回溯+剪枝

11.记忆化搜索

12.分治

13.动态规划

14.贪心

15.C++补充

```cpp
vector<int> split(string s, string delimiter) {

    vector<int> res;

    size_t pos = 0;
    string token;
    int tmp;
    while ((pos = s.find(delimiter)) != string::npos) {
        token = s.substr(0, pos);
        tmp = stoi(token);
        res.push_back(tmp);
        s.erase(0, pos + delimiter.length());
    }
    tmp = stoi(s);
    res.push_back(tmp);

    return res;

}
```