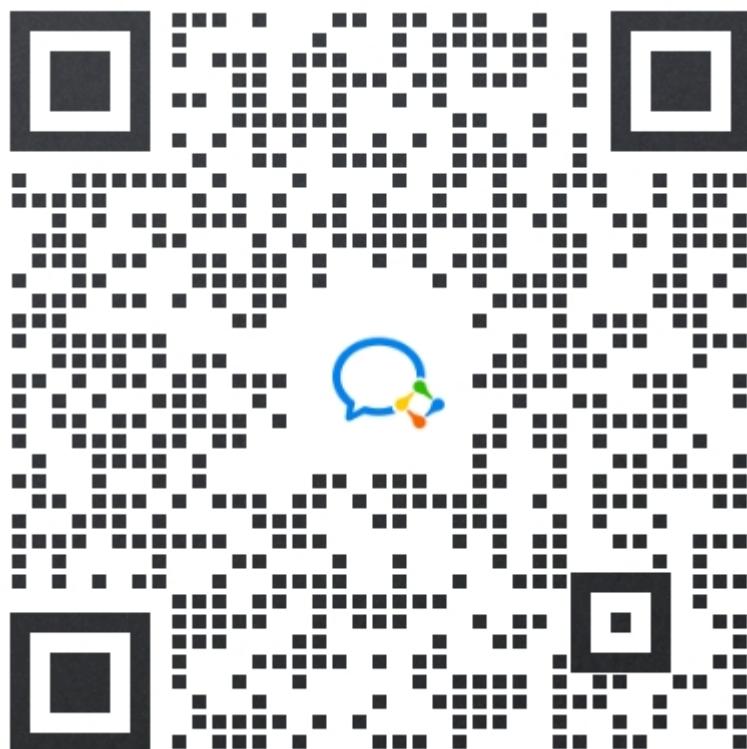


优选算法精品课(No.053~No.081)

版权说明

本“**比特就业课**”优选算法精品课(No.053~No.081)（以下简称“本精品课”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本精品课的开发者或授权方拥有版权。我们鼓励个人学习者使用本精品课进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本精品课的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，**未经我们明确授权，个人学习者不得将本精品课的内容用于任何商业目的**，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本精品课内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本精品课的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们取得联系，获取书面授权。感谢您对“**比特就业课**”优选算法精品课(No.053~No.081)的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方。

对比特算法感兴趣，可以联系这个微信。



板书链接

链表

51. 两数相加 (medium)

1. 题目链接: [2. 两数相加](#)

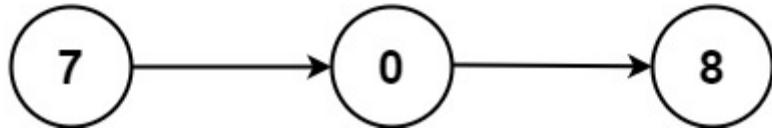
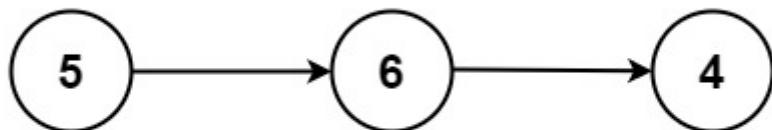
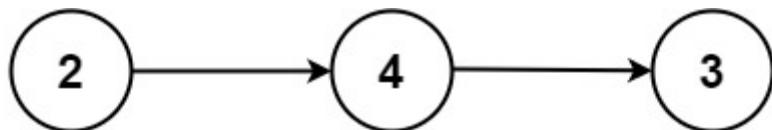
2. 题目描述:

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。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1:



输入: $l1 = [2,4,3], l2 = [5,6,4]$

输出: $[7,0,8]$

解释: $342 + 465 = 807.$

示例 2:

输入: $l1 = [0], l2 = [0]$

输出: $[0]$

示例 3：

输入： l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

输出： [8,9,9,9,0,0,0,1]

提示：

每个链表中的节点数在范围 [1, 100] 内

0 <= Node.val <= 9

题目数据保证列表表示的数字不含前导零

3. 解法（模拟）：

算法思路：

两个链表都是逆序存储数字的，即两个链表的个位数、十位数等都已经对应，可以直接相加。

在相加过程中，我们要注意是否产生进位，产生进位时需要将进位和链表数字一同相加。如果产生进位的位置在链表尾部，即答案位数比原链表位数长一位，还需要再 `new` 一个结点储存最高位。

C++ 算法代码：

```
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 */
11 class Solution
12 {
13 public:
14     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
15     {
16         ListNode* cur1 = l1, *cur2 = l2;
17         ListNode* newhead = new ListNode(0); // 创建一个虚拟头结点，记录最终结果
18         ListNode* prev = newhead; // 尾指针
19         int t = 0; // 记录进位
20     }
```

```

21     while(cur1 || cur2 || t)
22     {
23         // 先加上第一个链表
24         if(cur1)
25         {
26             t += cur1->val;
27             cur1 = cur1->next;
28         }
29         // 加上第二个链表
30         if(cur2)
31         {
32             t += cur2->val;
33             cur2 = cur2->next;
34         }
35         prev->next = new ListNode(t % 10);
36         prev = prev->next;
37         t /= 10;
38     }
39
40     prev = newhead->next;
41     delete newhead;
42
43     return prev;
44 }
45 };

```

C++ 运行结果：

C++



Java 算法代码：

```

1 /**
2 * Definition for singly-linked list.
3 * public class ListNode {
4 *     int val;
5 *     ListNode next;
6 *     ListNode() {}
7 *     ListNode(int val) { this.val = val; }
8 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

```

```

9  *
10 */
11 class Solution
12 {
13     public ListNode addTwoNumbers(ListNode l1, ListNode l2)
14     {
15         ListNode cur1 = l1, cur2 = l2;
16         ListNode newHead = new ListNode(0); // 创建一个虚拟头结点，方便记录结果
17         ListNode prev = newHead; // 尾插操作的尾指针
18         int t = 0; // 记录进位
19
20         while(cur1 != null || cur2 != null || t != 0)
21     {
22             // 先加上第一个链表
23             if(cur1 != null)
24             {
25                 t += cur1.val;
26                 cur1 = cur1.next;
27             }
28             // 加上第二个链表
29             if(cur2 != null)
30             {
31                 t += cur2.val;
32                 cur2 = cur2.next;
33             }
34             prev.next = new ListNode(t % 10);
35             prev = prev.next;
36             t /= 10;
37         }
38
39         return newHead.next;
40     }
41 }

```

Java 运行结果：



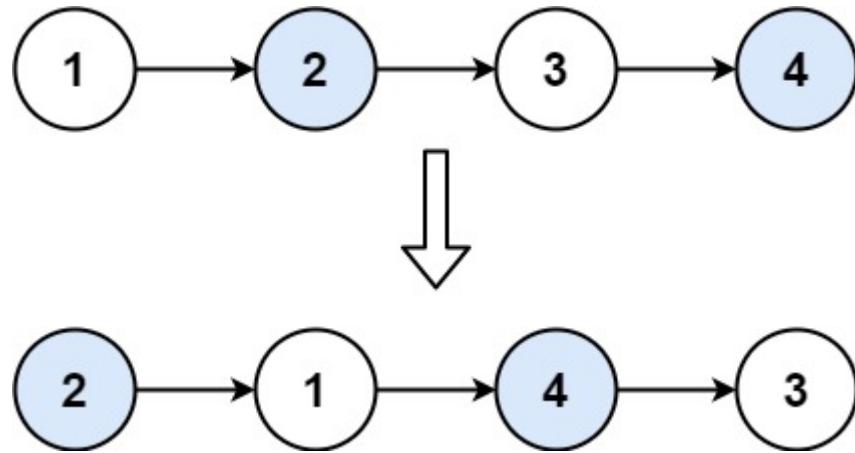
52. 两两交换链表中的节点 (medium)

1. 题目链接：[24. 两两交换链表中的节点](#)

2. 题目描述：

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 1：



输入： head = [1,2,3,4]

输出： [2,1,4,3]

示例 2：

输入： head = []

输出： []

示例 3：

输入： head = [1]

输出： [1]

提示：

链表中节点的数目在范围 [0, 100] 内

$0 \leq \text{Node.val} \leq 100$

3. 解法（模拟）：

算法思路：

画图画图画图，重要的事情说三遍~

C++ 算法代码：

```
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 */
11 class Solution
12 {
13 public:
14     ListNode* swapPairs(ListNode* head)
15     {
16         if(head == nullptr || head->next == nullptr) return head;
17
18         ListNode* newHead = new ListNode(0);
19         newHead->next = head;
20
21         ListNode* prev = newHead, *cur = prev->next, *next = cur->next, *nnext
22         = next->next;
23         while(cur && next)
24         {
25             // 交换结点
26             prev->next = next;
27             next->next = cur;
28             cur->next = nnnext;
29
30             // 修改指针
31             prev = cur; // 注意顺序
32             cur = nnnext;
33             if(cur) next = cur->next;
34             if(next) nnnext = next->next;
35         }
36         cur = newHead->next;
37         delete newHead;
38         return cur;
39     }
}
```

C++ 代码结果：

时间 4 ms

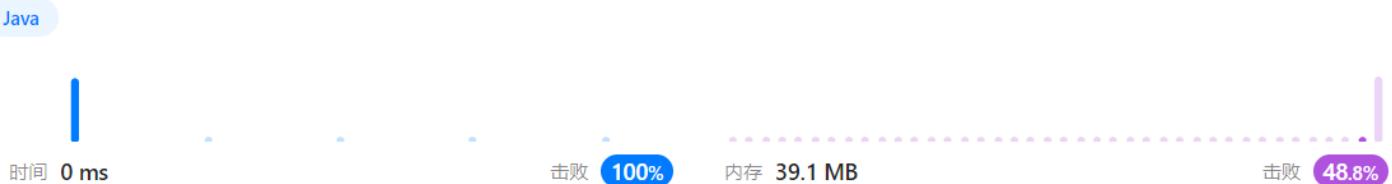
击败 55.87%

内存 7.4 MB

击败 50.47%

Java 算法代码：

```
1  /**
2  * Definition for singly-linked list.
3  */
4  public class ListNode {
5      int val;
6      ListNode next;
7      ListNode() {}
8      ListNode(int val) { this.val = val; }
9      ListNode(int val, ListNode next) { this.val = val; this.next = next; }
10 }
11
12 class Solution {
13     public ListNode swapPairs(ListNode head)
14     {
15         if(head == null || head.next == null) return head;
16
17         ListNode newHead = new ListNode(0);
18         newHead.next = head;
19
20         ListNode prev = newHead, cur = prev.next, next = cur.next, nnnext =
21             next.next;
22         while(cur != null && next != null)
23         {
24             // 交换节点
25             prev.next = next;
26             next.next = cur;
27             cur.next = nnnext;
28
29             // 修改指针
30             prev = cur; // 注意顺序
31             cur = nnnext;
32             if(cur != null) next = cur.next;
33             if(next != null) nnnext = next.next;
34         }
35
36         return newHead.next;
37     }
38 }
```

Java 运行结果：**53. 重排链表 (medium)**

1. 题目链接：[143. 重排链表](#)

2. 题目描述：

给定一个单链表 L 的头节点 $head$ ，单链表 L 表示为：

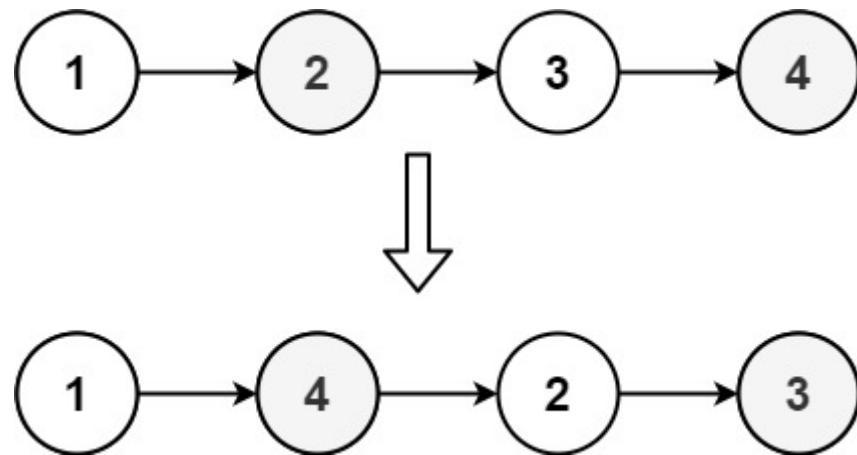
$L(0) \rightarrow L(1) \rightarrow \dots \rightarrow L(n - 1) \rightarrow L(n)$

请将其重新排列后变为：

$L(0) \rightarrow L(n) \rightarrow L(1) \rightarrow L(n - 1) \rightarrow L(2) \rightarrow L(n - 2) \rightarrow \dots$

不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

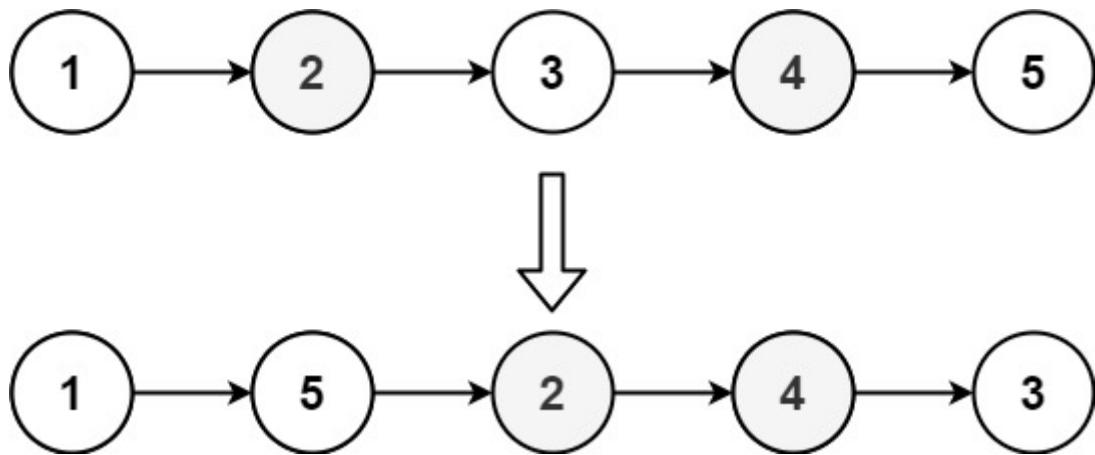
示例 1：



输入： $head = [1,2,3,4]$

输出： $[1,4,2,3]$

示例 2：



输入： head = [1,2,3,4,5]

输出： [1,5,2,4,3]

提示：

- 链表的长度范围为 `[1, 5 * 10(4)]`
- `1 <= node.val <= 1000`

3. 解法：

算法思路：

画图画图画图，重要的事情说三遍~

1. 找中间节点；
2. 中间部分往后的逆序；
3. 合并两个链表

C++ 算法代码：

```

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 */
11 class Solution
12 {
13 public:
14     void reorderList(ListNode* head)

```

```
15     {
16         // 处理边界情况
17         if(head == nullptr || head->next == nullptr || head->next->next == nullptr) return;
18
19         // 1. 找到链表的中间节点 - 快慢双指针 (一定要画图考虑 slow 的落点在哪里)
20         ListNode* slow = head, *fast = head;
21         while(fast && fast->next)
22         {
23             slow = slow->next;
24             fast = fast->next->next;
25         }
26
27         // 2. 把 slow 后面的部分给逆序 - 头插法
28         ListNode* head2 = new ListNode(0);
29         ListNode* cur = slow->next;
30         slow->next = nullptr; // 注意把两个链表给断开
31         while(cur)
32         {
33             ListNode* next = cur->next;
34             cur->next = head2->next;
35             head2->next = cur;
36             cur = next;
37         }
38
39         // 3. 合并两个链表 - 双指针
40         ListNode* ret = new ListNode(0);
41         ListNode* prev = ret;
42         ListNode* cur1 = head, *cur2 = head2->next;
43         while(cur1)
44         {
45             // 先放第一个链表
46             prev->next = cur1;
47             cur1 = cur1->next;
48             prev = prev->next;
49
50             // 再放第二个链表
51             if(cur2)
52             {
53                 prev->next = cur2;
54                 prev = prev->next;
55                 cur2 = cur2->next;
56             }
57         }
58         delete head2;
59         delete ret;
60     }
```

```
61 };
```

C++ 代码结果：

C++



Java 算法代码：

```
1 /**
2  * Definition for singly-linked list.
3  * public class ListNode {
4  *     int val;
5  *     ListNode next;
6  *     ListNode() {}
7  *     ListNode(int val) { this.val = val; }
8  *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
9  * }
10 */
11 class Solution
12 {
13     public void reorderList(ListNode head)
14     {
15         // 处理边界情况
16         if(head == null || head.next == null || head.next.next == null) return;
17
18         // 1. 找链表的中间节点 - 快慢双指针 (一定要画图分析 slow 的落点)
19         ListNode slow = head, fast = head;
20         while(fast != null && fast.next != null)
21         {
22             slow = slow.next;
23             fast = fast.next.next;
24         }
25
26         // 2. 把 slow 后面的部分给逆序 - 头插法
27         ListNode head2 = new ListNode(0);
28         ListNode cur = slow.next;
29         slow.next = null; // 把两个链表分离
30         while(cur != null)
31         {
32             ListNode next = cur.next;
```

```
33         cur.next = head2.next;
34         head2.next = cur;
35         cur = next;
36     }
37
38     // 3. 合并两个链表 - 双指针
39     ListNode cur1 = head, cur2 = head2.next;
40     ListNode ret = new ListNode(0);
41     ListNode prev = ret;
42     while(cur1 != null)
43     {
44         // 先放第一个链表
45         prev.next = cur1;
46         prev = cur1;
47         cur1 = cur1.next;
48
49         // 在合并第二个链表
50         if(cur2 != null)
51         {
52             prev.next = cur2;
53             prev = cur2;
54             cur2 = cur2.next;
55         }
56     }
57 }
58 }
```

Java 运行结果：



54. 合并 K 个升序链表 (hard)

1. 题目链接：[23. 合并 K 个升序链表](#)

2. 题目描述：

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 1:

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

示例 2:

输入: lists = []

输出: []

示例 3:

输入: lists = [[]]

输出: []

提示:

$k == \text{lists.length}$

$0 \leq k \leq 10^4$

$0 \leq \text{lists}[i].length \leq 500$

$-10^4 \leq \text{lists}[i][j] \leq 10^4$

$\text{lists}[i]$ 按升序排列

$\text{lists}[i].length$ 的总和不超过 10^4

3. 解法一（利用堆）：

算法思路:

合并两个有序链表是比较简单且做过的，就是用双指针依次比较链表 1、链表 2 未排序的最小元素，选择更小的那个加入有序的答案链表中。

合并 K 个升序链表时，我们依旧可以选择 K 个链表中，头结点值最小的那个。那么如何快速的得到头结点最小的是哪一个呢？用堆这个数据结构就好啦~

我们可以把所有的头结点放进一个小根堆中，这样就能快速的找到每次 K 个链表中，最小的元素是哪个。

C++ 算法代码：

```
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 */
11 class Solution
12 {
13     struct cmp
14     {
15         bool operator()(const ListNode* l1, const ListNode* l2)
16         {
17             return l1->val > l2->val;
18         }
19     };
20
21 public:
22     ListNode* mergeKLists(vector<ListNode*>& lists)
23     {
24         // 创建一个小根堆
25         priority_queue<ListNode*, vector<ListNode*>, cmp> heap;
26
27         // 让所有的头结点进入小根堆
28         for(auto l : lists)
29             if(l) heap.push(l);
30
31         // 合并 k 个有序链表
32         ListNode* ret = new ListNode(0);
33         ListNode* prev = ret;
34         while(!heap.empty())
```

```

35     {
36         ListNode* t = heap.top();
37         heap.pop();
38         prev->next = t;
39         prev = t;
40         if(t->next) heap.push(t->next);
41     }
42
43     prev = ret->next;
44     delete ret;
45     return prev;
46 }
47 };

```

C++ 运行结果：



Java 算法代码：

```

1 /**
2 * Definition for singly-linked list.
3 * public class ListNode {
4 *     int val;
5 *     ListNode next;
6 *     ListNode() {}
7 *     ListNode(int val) { this.val = val; }
8 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
9 * }
10 */
11 class Solution
12 {
13     public ListNode mergeKLists(ListNode[] lists)
14     {
15         PriorityQueue<ListNode> heap = new PriorityQueue<>((v1, v2) -> v1.val
16 - v2.val);
17
18         // 将所有头结点加入到小根堆中
19         for(ListNode l : lists)
20             if(l != null)
21                 heap.push(l);
22
23         ListNode dummy = new ListNode(0);
24         ListNode cur = dummy;
25
26         while(!heap.isEmpty())
27         {
28             cur.next = heap.poll();
29             cur = cur.next;
30             if(cur.next != null)
31                 heap.push(cur.next);
32         }
33
34         return dummy.next;
35     }
36 }

```

```

20             heap.offer(l);
21
22     // 合并
23     ListNode ret = new ListNode(0);
24     ListNode prev = ret;
25     while(!heap.isEmpty())
26     {
27         ListNode t = heap.poll();
28         prev.next = t;
29         prev = t;
30         if(t.next != null) heap.offer(t.next);
31     }
32
33     return ret.next;
34 }
35 }
```

Java 运行结果：



4. 解法二（递归/分治）：

算法思路：

逐一比较时，答案链表越来越长，每个跟它合并的小链表的元素都需要比较很多次才可以成功排序。

比如，我们有 8 个链表，每个链表长为 100。

逐一合并时，我们合并链表的长度分别为(0, 100), (100, 100), (200, 100), (300, 100), (400, 100), (500, 100), (600, 100), (700, 100)。所有链表的总长度共计 3600。

如果尽可能让长度相同的链表进行两两合并呢？这时合并链表的长度分别是(100, 100) x 4, (200, 200) x 2, (400, 400)，共计 2400。比上一种的计算量整整少了 1/3。

迭代的做法代码细节会稍多一些，这里给出递归的实现，代码相对简洁，不易写错。

算法流程：

1. 特判，如果题目给出空链表，无需合并，直接返回；
2. 返回递归结果。

递归函数设计：

1. 递归出口：如果当前要合并的链表编号范围左右值相等，无需合并，直接返回当前链表；
2. 应用二分思想，等额划分左右两段需要合并的链表，使这两段合并后的长度尽可能相等；
3. 对左右两段分别递归，合并[l, r]范围内的链表；
4. 再调用 mergeTwoLists 函数进行合并（就是合并两个有序链表）

C++ 算法代码：

```
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 */
11 class Solution
12 {
13 public:
14     ListNode* mergeKLists(vector<ListNode*>& lists)
15     {
16         return merge(lists, 0, lists.size() - 1);
17     }
18
19     ListNode* merge(vector<ListNode*>& lists, int left, int right)
20     {
21         if(left > right) return nullptr;
22         if(left == right) return lists[left];
23
24         // 1. 平分数组
25         int mid = left + right >> 1;
26         // [left, mid] [mid + 1, right]
27
28         // 2. 递归处理左右区间
29         ListNode* l1 = merge(lists, left, mid);
30         ListNode* l2 = merge(lists, mid + 1, right);
31
32         // 3. 合并两个有序链表
33         return mergeTowList(l1, l2);
34     }
35
36     ListNode* mergeTowList(ListNode* l1, ListNode* l2)
```

```

37     {
38         if(l1 == nullptr) return l2;
39         if(l2 == nullptr) return l1;
40
41         // 合并两个有序链表
42         ListNode head;
43         ListNode* cur1 = l1, *cur2 = l2, *prev = &head;
44         head.next = nullptr;
45
46         while(cur1 && cur2)
47         {
48             if(cur1->val <= cur2->val)
49             {
50                 prev = prev->next = cur1;
51                 cur1 = cur1->next;
52             }
53             else
54             {
55                 prev = prev->next = cur2;
56                 cur2 = cur2->next;
57             }
58         }
59
60         if(cur1) prev->next = cur1;
61         if(cur2) prev->next = cur2;
62
63         return head.next;
64     }
65 };

```

C++ 运行结果：

C++



Java 算法代码：

```

1 /**
2  * Definition for singly-linked list.
3  * public class ListNode {
4  *     int val;

```

```
5 *     ListNode next;
6 *     ListNode() {}
7 *     ListNode(int val) { this.val = val; }
8 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
9 *
10 */
11 class Solution
12 {
13     public ListNode mergeKLists(ListNode[] lists)
14     {
15         return merge(lists, 0, lists.length - 1);
16     }
17
18     public ListNode merge(ListNode[] lists, int left, int right)
19     {
20         if(left > right) return null;
21         if(left == right) return lists[left];
22
23         // 1. 平分数组
24         int mid = (left + right) / 2;
25         // [left, mid] [mid + 1, right]
26
27         // 2. 递归处理左右两部分
28         ListNode l1 = merge(lists, left, mid);
29         ListNode l2 = merge(lists, mid + 1, right);
30
31         // 3. 合并两个有序链表
32         return mergeTwoList(l1, l2);
33     }
34
35     public ListNode mergeTwoList(ListNode l1, ListNode l2)
36     {
37         if(l1 == null) return l2;
38         if(l2 == null) return l1;
39
40         // 合并两个有序链表
41         ListNode head = new ListNode(0);
42         ListNode cur1 = l1, cur2 = l2, prev = head;
43
44         while(cur1 != null && cur2 != null)
45         {
46             if(cur1.val <= cur2.val)
47             {
48                 prev.next = cur1;
49                 prev = cur1;
50                 cur1 = cur1.next;
51             }
52         }
53     }
54 }
```

```
52         else
53     {
54         prev.next = cur2;
55         prev = cur2;
56         cur2 = cur2.next;
57     }
58 }
59
60 if(cur1 != null) prev.next = cur1;
61 if(cur2 != null) prev.next = cur2;
62
63 return head.next;
64 }
65 }
```

Java 运行结果：

Java



55. K个一组翻转链表 (hard)

1. 题目链接：[25. K 个一组翻转链表](#)

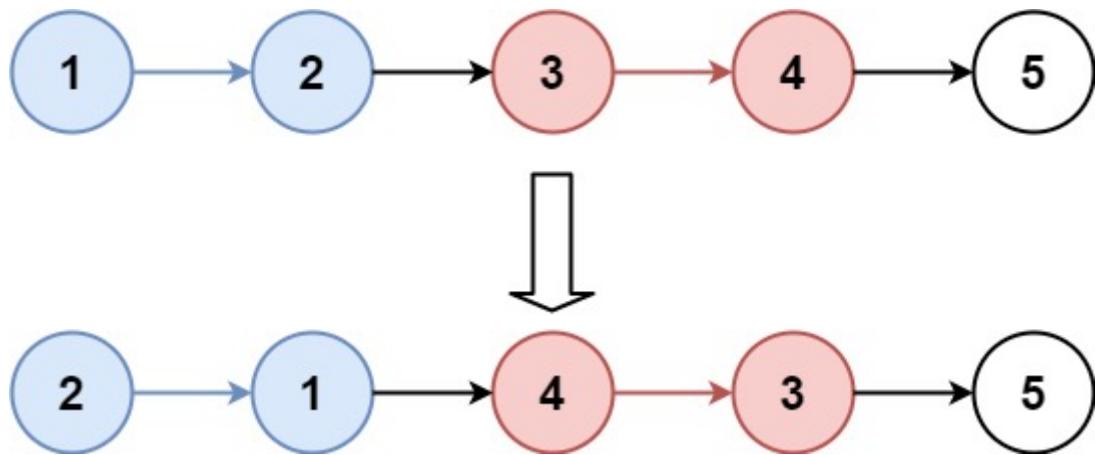
2. 题目描述：

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。

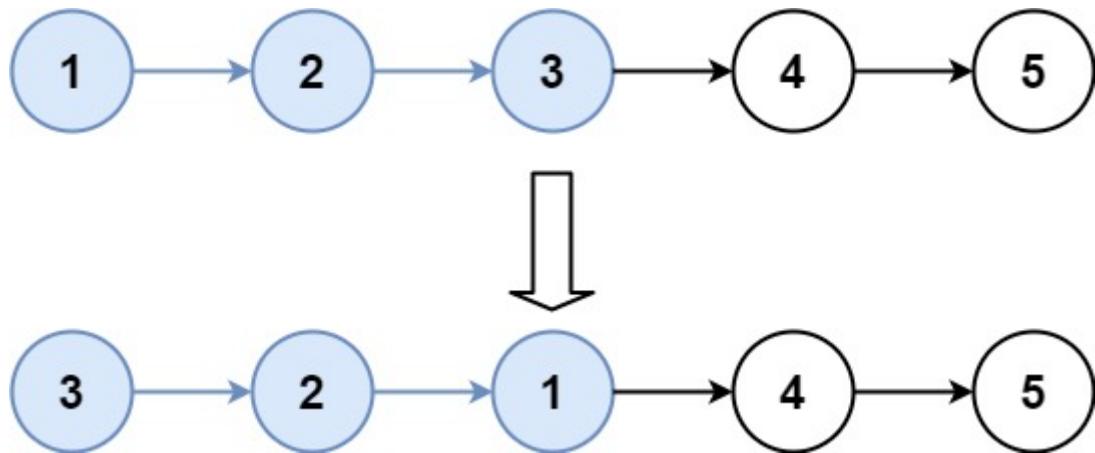
示例 1：



输入： head = [1,2,3,4,5], k = 2

输出： [2,1,4,3,5]

示例 2：



输入： head = [1,2,3,4,5], k = 3

输出： [3,2,1,4,5]

提示：

链表中的节点数目为 n

$1 \leq k \leq n \leq 5000$

$0 \leq \text{Node.val} \leq 1000$

进阶：你可以设计一个只用 $O(1)$ 额外内存空间的算法解决此问题吗？

3. 解法（模拟）：

算法思路：

本题的目标非常清晰易懂，不涉及复杂的算法，只是实现过程中需要考虑的细节比较多。

我们可以把链表按 k 个为一组进行分组，组内进行反转，并且记录反转后的头尾结点，使其可以和前、后连接起来。思路比较简单，但是实现起来是比较复杂的。

我们可以先求出一共需要逆序多少组（假设逆序 n 组），然后重复 n 次长度为 k 的链表的逆序即可。

C++ 算法代码：

```
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 */
11 class Solution
12 {
13 public:
14     ListNode* reverseKGroup(ListNode* head, int k)
15     {
16         // 1. 先求出需要逆序多少组
17         int n = 0;
18         ListNode* cur = head;
19         while(cur)
20         {
21             cur = cur->next;
22             n++;
23         }
24         n /= k;
25
26         // 2. 重复 n 次：长度为 k 的链表的逆序即可
27         ListNode* newHead = new ListNode(0);
28         ListNode* prev = newHead;
29         cur = head;
30
31         for(int i = 0; i < n; i++)
32         {
33             ListNode* tmp = cur;
34             for(int j = 0; j < k; j++)
35             {
36                 ListNode* next = cur->next;
```

```

37         cur->next = prev->next;
38         prev->next = cur;
39         cur = next;
40     }
41     prev = tmp;
42 }
43 // 把不需要翻转的接上
44 prev->next = cur;
45 cur = newHead->next;
46 delete newHead;
47 return cur;
48 }
49 };

```

C++ 运行结果：



Java 算法代码：

```

1 /**
2  * Definition for singly-linked list.
3  * public class ListNode {
4  *     int val;
5  *     ListNode next;
6  *     ListNode() {}
7  *     ListNode(int val) { this.val = val; }
8  *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
9  * }
10 */
11 class Solution
12 {
13     public ListNode reverseKGroup(ListNode head, int k)
14     {
15         // 1. 先求出需要逆序多少组
16         int n = 0;
17         ListNode cur = head;
18         while(cur != null)
19         {
20             cur = cur.next;

```

```

21             n++;
22         }
23         n /= k;
24
25         // 2. 重复 n 次: 长度为 k 的链表的逆序
26         ListNode newHead = new ListNode(0);
27         ListNode prev = newHead;
28         cur = head;
29
30         for(int i = 0; i < n; i++)
31     {
32             ListNode tmp = cur;
33             for(int j = 0; j < k; j++)
34     {
35                 // 头插的逻辑
36                 ListNode next = cur.next;
37                 cur.next = prev.next;
38                 prev.next = cur;
39                 cur = next;
40             }
41             prev = tmp;
42         }
43
44         // 把后面不需要逆序的部分连接上
45         prev.next = cur;
46         return newHead.next;
47     }
48 }

```

Java 运行结果:



哈希表

56. 两数之和 (easy)

1. 题目链接: [1. 两数之和](#)

有人相爱，有人夜里开车看海，有人 leetcode 第一题都做不出来。

2. 题目描述：

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出和为目标值 `target` 的那两个整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 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 3：

输入： `nums = [3,3]`, `target = 6`

输出： `[0,1]`

3. 解法（哈希表）：

算法思路：

- 如果我们可以事先将「数组内的元素」和「下标」绑定在一起存入「哈希表」中，然后直接在哈希表中查找每一个元素的 `target - nums[i]`，就能快速的找到「目标和的下标」。
- 这里有一个小技巧，我们可以不用将元素全部放入到哈希表之后，再来二次遍历（因为要处理元素相同的情况）。而是在将元素放入到哈希表中的「同时」，直接来检查表中是否已经存在当前元素所对应的目标元素（即 `target - nums[i]`）。如果它存在，那我们已经找到了对应解，并立即将其返回。无需将元素全部放入哈希表中，提高效率。
- 因为哈希表中查找元素的时间复杂度是 $O(1)$ ，遍历一遍数组的时间复杂度为 $O(N)$ ，因此可以将时间复杂度降到 $O(N)$ 。

这是一个典型的「用空间交换时间」的方式。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<int> twoSum(vector<int>& nums, int target)
5     {
6         unordered_map<int, int> hash; // <nums[i], i>
```

```
7     for(int i = 0; i < nums.size(); i++)
8     {
9         int x = target - nums[i];
10        if(hash.count(x)) return {hash[x], i};
11        hash[nums[i]] = i;
12    }
13    // 照顾编译器
14    return {-1, -1};
15}
16};
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public int[] twoSum(int[] nums, int target)
4     {
5         Map<Integer, Integer> hash = new HashMap<>(); // <nums[i], i>
6         for(int i = 0; i < nums.length; i++)
7         {
8             int x = target - nums[i];
9             if(hash.containsKey(x))
10             {
11                 return new int[]{i, hash.get(x)};
12             }
13             hash.put(nums[i], i);
14         }
15         // 照顾编译器
16         return new int[]{-1, -1};
17     }
18 }
```

Java 运行结果：



57. 判断是否互为字符重排 (easy)

1. 题目链接：面试题 01.02. 判定是否互为字符重排

2. 题目描述：

给定两个字符串 `s1` 和 `s2`，请编写一个程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。

示例 1：

输入: `s1 = "abc", s2 = "bca"`

输出: `true`

示例 2：

输入: `s1 = "abc", s2 = "bad"`

输出: `false`

3. 解法（哈希表）：

算法思路：

- 当两个字符串的长度不相等的时候，是不可能构成互相重排的，直接返回 `false`；
- 如果两个字符串能够构成互相重排，那么每个字符串中「各个字符」出现的「次数」一定是相同的。因此，我们可以分别统计出这两个字符串中各个字符出现的次数，然后逐个比较是否相等即可。这样的话，我们就可以选择「哈希表」来统计字符串中字符出现的次数。

C++ 算法代码：

```

1 class Solution
2 {
3 public:
4     bool CheckPermutation(string s1, string s2)
5     {
6         if(s1.size() != s2.size()) return false;
7
8         int hash[26] = { 0 };

```

```

9     // 先统计第一个字符串的信息
10    for(auto ch : s1)
11        hash[ch - 'a']++;
12
13    // 扫描第二个字符串，看看是否能重排
14    for(auto ch : s2)
15    {
16        hash[ch - 'a']--;
17        if(hash[ch - 'a'] < 0) return false;
18    }
19    return true;
20 }
21 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public boolean CheckPermutation(String s1, String s2)
4     {
5         if(s1.length() != s2.length()) return false;
6
7         int[] hash = new int[26];
8         // 先把第一个字符串的信息统计到哈希表中
9         for(int i = 0; i < s1.length(); i++)
10        {
11            hash[s1.charAt(i) - 'a']++;
12        }
13
14        // 遍历第二个字符串，判断是否可以重排
15        for(int i = 0; i < s2.length(); i++)
16        {
17            hash[s2.charAt(i) - 'a']--;
18            if(hash[s2.charAt(i) - 'a'] < 0) return false;
19        }
20        return true;

```

```
21      }
22 }
```

Java 运行结果：



58. 存在重复元素 I (easy)

1. 题目链接：[217. 存在重复元素](#)

2. 题目描述：

给你一个整数数组 `nums`。如果任一值在数组中出现 至少两次，返回 `true`；如果数组中每个元素互不相同，返回 `false`。

示例 1：

输入： `nums = [1,2,3,1]`

输出： `true`

示例 2：

输入： `nums = [1,2,3,4]`

输出： `false`

3. 解法（哈希表）：

算法思路：

分析一下题目，出现「至少两次」的意思就是数组中存在着重复的元素，因此我们可以无需统计元素出现的数目。仅需在遍历数组的过程中，检查当前元素「是否在之前已经出现过」即可。

因此我们可以利用哈希表，仅需存储数「组内的元素」。在遍历数组的时候，一边检查哈希表中是否已经出现过当前元素，一边将元素加入到哈希表中。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
```

```
4     bool containsDuplicate(vector<int>& nums)
5     {
6         unordered_set<int> hash;
7         for(auto x : nums)
8             if(hash.count(x)) return true;
9             else hash.insert(x);
10        return false;
11    }
12 }
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public boolean containsDuplicate(int[] nums)
4     {
5         Set<Integer> hash = new HashSet<>();
6         for(int x : nums)
7         {
8             if(hash.contains(x)) return true;
9             hash.add(x);
10        }
11        return false;
12    }
13 }
```

Java 运行结果：



59. 存在重复元素 II (easy)

1. 题目链接：[219. 存在重复元素 II](#)

2. 题目描述：

给你一个整数数组 `nums` 和一个整数 `k`，判断数组中是否存在两个不同的索引 `i` 和 `j`，满足 `nums[i] == nums[j]` 且 `abs(i - j) <= k`。如果存在，返回 `true`；否则，返回 `false`。

示例 1：

输入： `nums = [1,2,3,1], k = 3`

输出： `true`

示例 2：

输入： `nums = [1,0,1,1], k = 1`

输出： `true`

3. 解法（哈希表）：

算法思路：

解决该问题需要我们快速定位到两个信息：

- 两个相同的元素；
- 这两个相同元素的下标。

因此，我们可以使用「哈希表」，令数组内的元素做 `key` 值，该元素所对应的下标做 `val` 值，将「数组元素」和「下标」绑定在一起，存入到「哈希表」中。

思考题：

如果数组内存在大量的「重复元素」，而我们判断下标所对应的元素是否符合条件的时候，需要将不同下标的元素作比较，怎么处理这个情况呢？

答：这里运用了一个「小贪心」。

我们按照下标「从小到大」的顺序遍历数组，当遇到两个元素相同，并且比较它们的下标时，这两个下标一定是距离最近的，因为：

- 如果当前判断符合条件直接返回 `true`，无需继续往后查找。
- 如果不符合条件，那么前一个下标一定不可能与后续相同元素的下标匹配（因为下标在逐渐变大），那么我们可以大胆舍去前一个存储的下标，转而将其换成新的下标，继续匹配。

C++ 算法代码：

```

1 class Solution
2 {
3     public:
4         bool containsNearbyDuplicate(vector<int>& nums, int k)
5     {
6         unordered_map<int, int> hash;
7         for(int i = 0; i < nums.size(); i++)
8         {
9             if(hash.count(nums[i]))
10             {
11                 if(i - hash[nums[i]] <= k) return true;
12             }
13             hash[nums[i]] = i;
14         }
15         return false;
16     }
17 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public boolean containsNearbyDuplicate(int[] nums, int k)
4     {
5         Map<Integer, Integer> hash = new HashMap<>();
6         for(int i = 0; i < nums.length; i++)
7         {
8             if(hash.containsKey(nums[i]))
9             {
10                 if(i - hash.get(nums[i]) <= k) return true;
11             }
12             hash.put(nums[i], i);
13         }
14         return false;
15     }
16 }

```

Java 运行结果：



60. 字母异位词分组 (medium)

从这道题我们可以拓展一下视野，不要将容器局限于基本类型，它也可以是一个容器嵌套另一个容器的复杂类型。

1. 题目链接：[49. 字母异位词分组](#)

2. 题目描述：

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字母得到的一个新单词，所有源单词中的字母通常恰好只用一次。

示例 1:

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

输出: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

3. 解法（哈希表 + 排序）：

算法思路：

互为字母异位词的单词有一个特点：将它们「排序」之后，两个单词应该是「完全相同」的。

所以，我们可以利用这个特性，将单词按照字典序排序，如果排序后的单词相同的话，就划分到同一组中。

这时我们就要处理两个问题：

- 排序后的单词与原单词需要能互相映射；
- 将排序后相同的单词，「划分到同一组」；

利用语言提供的「容器」的强大的功能就能实现这两点：

- 将排序后的字符串（`string`）当做哈希表的 `key` 值；
- 将字母异位词数组（`string[]`）当成 `val` 值。

定义一个「哈希表」即可解决问题。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<vector<string>> groupAnagrams(vector<string>& strs)
5     {
6         unordered_map<string, vector<string>> hash;
7
8         // 1. 把所有的字母异位词分组
9         for(auto& s : strs)
10        {
11            string tmp = s;
12            sort(tmp.begin(), tmp.end());
13            hash[tmp].push_back(s);
14        }
15
16         // 2. 结果提取出来
17         vector<vector<string>> ret;
18         for(auto& [x, y] : hash)
19        {
20             ret.push_back(y);
21         }
22         return ret;
23     }
24 }
```

C++ 运行结果：

C++



Java 算法代码：

```
1 class Solution
2 {
3     public List<List<String>> groupAnagrams(String[] strs)
4     {
5         Map<String, List<String>> hash = new HashMap<>();
```

```
6
7     // 1. 先把所有的字母异位词分组
8     for(String s : strs)
9     {
10         char[] tmp = s.toCharArray();
11         Arrays.sort(tmp);
12         String key = new String(tmp);
13
14         if(!hash.containsKey(key))
15         {
16             hash.put(key, new ArrayList());
17         }
18         hash.get(key).add(s);
19     }
20
21     // 2. 提取结果
22     return new ArrayList(hash.values());
23 }
24 }
```

Java 运行结果：

Java



字符串

61. 最长公共前缀 (easy)

1. 题目链接：[14. 最长公共前缀](#)

2. 题目描述：

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例 1：

输入：strs = ["flower", "flow", "flight"]

输出: "fl"

示例 2:

输入: strs = ["dog", "racecar", "car"]

输出: ""

解释: 输入不存在公共前缀。

提示:

$1 \leq \text{strs.length} \leq 200$

$0 \leq \text{strs[i].length} \leq 200$

strs[i] 仅由小写英文字母组成

3. 解法:

算法思路:

解法一（两两比较）：

我们可以先找出前两个的最长公共前缀，然后拿这个最长公共前缀依次与后面的字符串比较，这样就可以找出所有字符串的最长公共前缀。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     string longestCommonPrefix(vector<string>& strs)
5     {
6         // 解法一：两两比较
7         string ret = strs[0];
8         for(int i = 1; i < strs.size(); i++)
9             ret = findCommon(ret, strs[i]);
10        return ret;
11    }
12
13    string findCommon(string& s1, string& s2)
14    {
15        int i = 0;
16        while(i < min(s1.size(), s2.size()) && s1[i] == s2[i]) i++;
17        return s1.substr(0, i);
```

```
18     }
19 };
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public String longestCommonPrefix(String[] strs)
4     {
5         // 解法一：两两比较
6         String ret = strs[0];
7         for(int i = 1; i < strs.length; i++)
8         {
9             ret = findCommon(strs[i], ret);
10        }
11        return ret;
12    }
13
14    public String findCommon(String s1, String s2)
15    {
16        int i = 0;
17        while(i < Math.min(s1.length(), s2.length()) && s1.charAt(i) ==
18            s2.charAt(i))
19        {
20            i++;
21        }
22    }
23 }
```

Java 运行结果：



解法二（统一比较）：

题目要求多个字符串的公共前缀，我们可以逐位比较这些字符串，哪一位出现了不同，就在哪一位截止。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string longestCommonPrefix(vector<string>& strs)
5     {
6         // 解法二：统一比较
7         for(int i = 0; i < strs[0].size(); i++)
8         {
9             char tmp = strs[0][i];
10            for(int j = 1; j < strs.size(); j++)
11                if(i == strs[j].size() || tmp != strs[j][i])
12                    return strs[0].substr(0, i);
13        }
14        return strs[0];
15    }
16 }
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public String longestCommonPrefix(String[] strs)
4     {
5         // 解法二：统一比较
6         for(int i = 0; i < strs[0].length(); i++)
7         {
```

```
8         char tmp = strs[0].charAt(i);
9         for(int j = 1; j < strs.length; j++)
10        {
11            if(i == strs[j].length() || strs[j].charAt(i) != tmp)
12                return strs[0].substring(0, i);
13        }
14    }
15    return strs[0];
16 }
17 }
```

Java 运行结果：



62. 最长回文子串 (medium)

1. 题目链接：[5. 最长回文子串](#)

2. 题目描述：

给你一个字符串 s，找到 s 中最长的回文子串。

如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。

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"

提示：

1 <= s.length <= 1000

s 仅由数字和英文字母组成

3. 解法（中心扩散）：

算法思路：

枚举每一个可能的子串非常费时，有没有比较简单一点的方法呢？

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。如此这样去除，一直除到长度小于等于 2 时呢？长度为 1 的，自身与自身就构成回文；而长度为 2 的，就要判断这两个字符是否相等了。

从这个性质可以反推出来，从回文串的中心开始，往左读和往右读也是一样的。那么，是否可以枚举回文串的中心呢？

从中心向两边扩展，如果两边的字母相同，我们就可以继续扩展；如果不同，我们就停止扩展。这样只需要一层 for 循环，我们就可以完成先前两层 for 循环的工作量。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string longestPalindrome(string s)
5     {
6         // 中心扩展算法
7         int begin = 0, len = 0, n = s.size();
8         for(int i = 0; i < n; i++) // 依次枚举所有的中点
9         {
10             // 先做一次奇数长度的扩展
11             int left = i, right = i;
12             while(left >= 0 && right < n && s[left] == s[right])
13             {
14                 left--;
15                 right++;
16             }
17             if(right - left - 1 > len)
18             {
19                 begin = left + 1;
20                 len = right - left - 1;
21             }
22             // 偶数长度的扩展
23             left = i, right = i + 1;
24             while(left >= 0 && right < n && s[left] == s[right])
```

```

25         {
26             left--;
27             right++;
28         }
29         if(right - left - 1 > len)
30         {
31             begin = left + 1;
32             len = right - left - 1;
33         }
34     }
35     return s.substr(begin, len);
36 }
37 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public String longestPalindrome(String s)
4     {
5         int begin = 0, len = 0, n = s.length();
6         for(int i = 0; i < n; i++) // 固定所有的中间点
7         {
8             // 先扩展奇数长度的子串
9             int left = i, right = i;
10            while(left >= 0 && right < n && s.charAt(left) == s.charAt(right))
11            {
12                left--;
13                right++;
14            }
15            if(right - left - 1 > len)
16            {
17                begin = left + 1;
18                len = right - left - 1;
19            }
20        } // 扩展偶数长度

```

```

21         left = i; right = i + 1;
22         while(left >= 0 && right < n && s.charAt(left) == s.charAt(right))
23     {
24         left--;
25         right++;
26     }
27     if(right - left - 1 > len)
28     {
29         begin = left + 1;
30         len = right - left - 1;
31     }
32 }
33 return s.substring(begin, begin + len);
34 }
35 }
```

Java 运行结果：

Java



63. 二进制求和 (easy)

1. 题目链接：67. 二进制求和

2. 题目描述：

给你两个二进制字符串 a 和 b，以二进制字符串的形式返回它们的和。

示例 1：

输入: a = "11", b = "1"

输出: "100"

示例 2：

输入: a = "1010", b = "1011"

输出: "10101"

3. 解法（模拟十进制的大数相加的过程）：

算法思路：

模拟十进制中我们列竖式计算两个数之和的过程。但是这里是二进制的求和，我们不是逢十进一，而是逢二进一。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string addBinary(string a, string b)
5     {
6         string ret;
7
8         int cur1 = a.size() - 1, cur2 = b.size() - 1, t = 0;
9         while(cur1 >= 0 || cur2 >= 0 || t)
10        {
11            if(cur1 >= 0) t += a[cur1--] - '0';
12            if(cur2 >= 0) t += b[cur2--] - '0';
13            ret += t % 2 + '0';
14            t /= 2;
15        }
16        reverse(ret.begin(), ret.end());
17
18        return ret;
19    }
20}
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public String addBinary(String a, String b)
4     {
5         StringBuffer ret = new StringBuffer();
6         int cur1 = a.length() - 1, cur2 = b.length() - 1, t = 0;
```

```

7         while(cur1 >= 0 || cur2 >= 0 || t != 0)
8     {
9         if(cur1 >= 0) t += a.charAt(cur1--) - '0';
10        if(cur2 >= 0) t += b.charAt(cur2--) - '0';
11        ret.append((char)('0' + (char)(t % 2)));
12        t /= 2;
13    }
14    ret.reverse();
15    return ret.toString();
16}
17

```

Java 运行结果：



64. 字符串相乘 (medium)

1. 题目链接：[43. 字符串相乘](#)

2. 题目描述：

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

注意：不能使用任何内置的 BigInteger 库或直接将输入转换为整数。

示例 1:

输入: num1 = "2", num2 = "3"

输出: "6"

示例 2:

输入: num1 = "123", num2 = "456"

输出: "56088"

3. 解法（无进位相乘然后相加，最后处理进位）：

算法思路：

整体思路就是模拟我们小学列竖式计算两个数相乘的过程。但是为了我们书写代码的方便性，我们选择一种优化版本的，就是在计算两数相乘的时候，先不考虑进位，等到所有结果计算完毕之后，再去考虑进位。如下图：

The diagram illustrates the manual calculation of the multiplication problem:

$$\begin{array}{r} 43210 \\ \times 456 \\ \hline 16 & 20 & 24 \\ 12 & 15 & 18 \\ \hline 12 & 31 & 38 & 24 \\ 3 & 4 & 2 \\ \hline 1 & 5 & 0 & 4 \end{array}$$

Annotations in red explain the process:

- 数组下标 (→ array index) pointing to the top row.
- num1 数组 (→ num1 array) pointing to the second row.
- num2 数组 (→ num2 array) pointing to the third row.
- 无进位相乘后相加 (→ no carry multiplication followed by addition) pointing to the intermediate step.
- 相乘相加后的结果 (→ multiplication and addition result) pointing to the intermediate result.
- 处理进位 (→ handle carry) pointing to the final result.

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string multiply(string n1, string n2)
5     {
6         // 解法：无进位相乘后相加，然后处理进位
7         int m = n1.size(), n = n2.size();
8         reverse(n1.begin(), n1.end());
9         reverse(n2.begin(), n2.end());
10        vector<int> tmp(m + n - 1);
11
12        // 1. 无进位相乘后相加
13        for(int i = 0; i < m; i++)
14            for(int j = 0; j < n; j++)
15                tmp[i + j] += (n1[i] - '0') * (n2[j] - '0');
16
17        // 2. 处理进位
```

```

18     int cur = 0, t = 0;
19     string ret;
20     while(cur < m + n - 1 || t)
21     {
22         if(cur < m + n - 1) t += tmp[cur++];
23         ret += t % 10 + '0';
24         t /= 10;
25     }
26
27     // 3. 处理前导零
28     while(ret.size() > 1 && ret.back() == '0') ret.pop_back();
29
30     reverse(ret.begin(), ret.end());
31     return ret;
32 }
33 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public String multiply(String num1, String num2)
4     {
5         int m = num1.length(), n = num2.length();
6         char[] n1 = new StringBuffer(num1).reverse().toString().toCharArray();
7         char[] n2 = new StringBuffer(num2).reverse().toString().toCharArray();
8
9         int[] tmp = new int[m + n - 1];
10
11        // 1. 无进位相乘后相加
12        for(int i = 0; i < m; i++)
13            for(int j = 0; j < n; j++)
14                tmp[i + j] += (n1[i] - '0') * (n2[j] - '0');
15
16        // 2. 处理进位
17        int cur = 0, t = 0;

```

```

18     StringBuffer ret = new StringBuffer();
19     while(cur < m + n - 1 || t != 0)
20     {
21         if(cur < m + n - 1) t += tmp[cur++];
22         ret.append((char)(t % 10 + '0'));
23         t /= 10;
24     }
25
26     // 3. 处理进位
27     while(ret.length() > 1 && ret.charAt(ret.length() - 1) == '0')
28         ret.deleteCharAt((ret.length() - 1));
29
30     return ret.reverse().toString();
31 }
32 }
```

Java 运行结果：



栈

65. 删除字符串中的所有相邻重复项 (easy)

1. 题目链接：[1047. 删除字符串中的所有相邻重复项](#)

2. 题目描述：

给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

输入： "abbaca"

输出： "ca"

解释：

例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

3. 解法（栈）：

算法思路：

本题极像我们玩过的「开心消消乐」游戏，仔细观察消除过程，可以发现本题与我们之前做过的「括号匹配」问题是类似的。当前元素是否被消除，需要知道上一个元素的信息，因此可以用「栈」来保存信息。

但是，如果使用 `stack` 来保存的话，最后还需要把结果从栈中取出来。不如直接用「数组模拟一个栈」结构：在数组的尾部「尾插尾删」，实现栈的「进栈」和「出栈」。那么最后数组存留的内容，就是最后的结果。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string removeDuplicates(string s)
5     {
6         string ret; // 搞一个数组，模拟栈结构即可
7         for(auto ch : s)
8         {
9             if(ret.size() && ch == ret.back()) ret.pop_back(); // 出栈
10            else ret += ch; // 入栈
11        }
12        return ret;
13    }
14 }
```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public String removeDuplicates(String _s)
4     {
5         StringBuffer ret = new StringBuffer(); // 用数组来模拟栈结构
6         char[] s = _s.toCharArray();
7         for(char ch : s)
8         {
9             if(ret.length() > 0 && ch == ret.charAt(ret.length() - 1))
10            {
11                // 出栈
12                ret.deleteCharAt(ret.length() - 1);
13            }
14            else
15            {
16                // 进栈
17                ret.append(ch);
18            }
19        }
20
21        return ret.toString();
22    }
23 }

```

Java 运行结果：



66. 比较含退格的字符串 (easy)

1. 题目链接：844. 比较含退格的字符串

2. 题目描述：

给定 s 和 t 两个字符串，当它们分别被输入到空白的文本编辑器后，如果两者相等，返回 true。# 代表退格字符。

注意：如果对空文本输入退格字符，文本继续为空。

示例 1：

输入： $s = "ab#c"$, $t = "ad#c"$

输出: true

解释:

s 和 t 都会变成 "ac"。

示例 2:

输入: s = "ab##", t = "c#d#"

输出: true

解释:

s 和 t 都会变成 ""。

3. 解法（用数组模拟栈）：

算法思路:

由于退格的时候需要知道「前面元素」的信息，而且退格也符合「后进先出」的特性。因此我们可以使用「栈」结构来模拟退格的过程。

- 当遇到非 # 字符的时候，直接进栈；
- 当遇到 # 的时候，栈顶元素出栈。

为了方便统计结果，我们使用「数组」来模拟实现栈结构。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     bool backspaceCompare(string s, string t)
5     {
6         return changeStr(s) == changeStr(t);
7     }
8
9     string changeStr(string& s)
10    {
11        string ret; // 用数组模拟栈结构
12        for(char ch : s)
13        {
14            if(ch != '#') ret += ch;
15            else
16            {
17                if(ret.size()) ret.pop_back();
18            }
19        }
20        return ret;
21    }
22 }
```

```
19     }
20     return ret;
21 }
22 };
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public boolean backspaceCompare(String s, String t)
4     {
5         return changeStr(s).equals(changeStr(t));
6     }
7
8     public String changeStr(String s)
9     {
10        StringBuffer ret = new StringBuffer(); // 用数组模拟栈结构
11        for(int i = 0; i < s.length(); i++)
12        {
13            char ch = s.charAt(i);
14            if(ch != '#')
15            {
16                ret.append(ch); // 入栈
17            }
18            else
19            {
20                if(ret.length() > 0) ret.deleteCharAt(ret.length() - 1); // 出栈
21            }
22        }
23        return ret.toString();
24    }
25 }
```

Java 运行结果：



67. 基本计算器 II (medium)

1. 题目链接：[227. 基本计算器 II](#)

2. 题目描述：

给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

你可以假设给定的表达式总是有效的。所有中间结果将在 $[-2^{31}, 2^{31} - 1]$ 的范围内。

注意：不允许使用任何将字符串作为数学表达式计算的内置函数，比如 `eval()`。

示例 1：

输入： $s = "3+2*2"$

输出：7

示例 2：

输入： $s = "3/2"$

输出：1

示例 3：

输入： $s = "3+5 / 2"$

输出：5

提示：

- $1 \leq s.length \leq 3 * 10^5$
- s 由整数和算符 ('+', '-', '*', '/') 组成，中间由一些空格隔开
- s 表示一个有效表达式
- 表达式中的所有整数都是非负整数，且在范围 $[0, 2^{31} - 1]$ 内
- 题目数据保证答案是一个 32-bit 整数

题目解析：

一定要认真看题目的提示，从提示中我们可以看到这道题：

- 只有「加减乘除」四个运算；

- 没有括号；
- 并且每一个数都是大于等于 0 的；

这样可以大大的「减少」我们需要处理的情况。

3. 解法（栈）：

算法思路：

由于表达式里面没有括号，因此我们只用处理「加减乘除」混合运算即可。根据四则运算的顺序，我们可以先计算乘除法，然后再计算加减法。由此，我们可以得出下面的结论：

- 当一个数前面是 '+' 号的时候，这一个数是否会被立即计算是「不确定」的，因此我们可以先压入栈中；
- 当一个数前面是 '-' 号的时候，这一个数是否被立即计算也是「不确定」的，但是这个数已经和前面的 - 号绑定了，因此我们可以将这个数的相反数压入栈中；
- 当一个数前面是 '*' 号的时候，这一个数可以立即与前面的一个数相乘，此时我们让将栈顶的元素乘上这个数；
- 当一个数前面是 '/' 号的时候，这一个数也是可以立即被计算的，因此我们让栈顶元素除以这个数。

当遍历完全部的表达式的时候，栈中剩余的「元素之和」就是最终结果。

C++ 算法代码：

```

1 class Solution
2 {
3 public:
4     int calculate(string s)
5     {
6         vector<int> st; // 用数组来模拟栈结构
7         int i = 0, n = s.size();
8         char op = '+';
9         while(i < n)
10        {
11            if(s[i] == ' ') i++;
12            else if(s[i] >= '0' && s[i] <= '9')
13            {
14                // 先把这个数字给提取出来
15                int tmp = 0;
16                while(i < n && s[i] >= '0' && s[i] <= '9')
17                    tmp = tmp * 10 + (s[i++] - '0');
18                if(op == '+') st.push_back(tmp);

```

```

19             else if(op == '-') st.push_back(-tmp);
20             else if(op == '*') st.back() *= tmp;
21             else st.back() /= tmp;
22         }
23     else
24     {
25         op = s[i];
26         i++;
27     }
28 }
29 int ret = 0;
30 for(auto x : st) ret += x;
31 return ret;
32 }
33 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public int calculate(String _s)
4     {
5         Deque<Integer> st = new ArrayDeque<>();
6         char op = '+';
7         int i = 0, n = _s.length();
8         char[] s = _s.toCharArray();
9
10        while(i < n)
11        {
12            if(s[i] == ' ') i++;
13            else if(s[i] >= '0' && s[i] <= '9')
14            {
15                int tmp = 0;
16                while(i < n && s[i] >= '0' && s[i] <= '9')
17                {
18                    tmp = tmp * 10 + (s[i] - '0');
19
20                }
21                if(op == '+') st.push(tmp);
22                else if(op == '-')
23                {
24                    st.push(-tmp);
25                }
26                else if(op == '*')
27                {
28                    st.push(st.pop() * tmp);
29                }
30                else if(op == '/')
31                {
32                    st.push(st.pop() / tmp);
33                }
34            }
35        }
36        int ret = 0;
37        for(int x : st) ret += x;
38        return ret;
39    }
40 }

```

```

19             i++;
20         }
21         if(op == '+') st.push(tmp);
22         else if(op == '-') st.push(-tmp);
23         else if(op == '*') st.push(st.pop() * tmp);
24         else st.push(st.pop() / tmp);
25     }
26     else
27     {
28         op = s[i];
29         i++;
30     }
31 }
32 // 统计结果
33 int ret = 0;
34 while(!st.isEmpty())
35 {
36     ret += st.pop();
37 }
38 return ret;
39 }
40 }

```

Java 运行结果：

Java



68. 字符串解码 (medium)

1. 题目链接：394. 字符串解码

2. 题目描述：

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。

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"

3. 解法（两个栈）：

算法思路：

对于 `3[ab2[cd]]`，我们需要先解码内部的，再解码外部（为了方便区分，使用了空格）：

- `3[ab2[cd]] -> 3[abcd cd] -> abcdcd abcdcd abcdcd`

在解码 `cd` 的时候，我们需要保存 `3 ab 2` 这些元素的信息，并且这些信息使用的顺序是从后往前，正好符合栈的结构，因此我们可以定义两个栈结构，一个用来保存解码前的重复次数 `k`（左括号前的数字），一个用来保存解码之前字符串的信息（左括号前的字符串信息）。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string decodeString(string s)
5     {
6         stack<int> nums;
7         stack<string> st;
8         st.push("");
9         int i = 0, n = s.size();
10
11        while(i < n)
12        {
13            if(s[i] >= '0' && s[i] <= '9')
14            {
15                int tmp = 0;
16                while(s[i] >= '0' && s[i] <= '9')
17                {
```

```
18             tmp = tmp * 10 + (s[i] - '0');
19             i++;
20         }
21         nums.push(tmp);
22     }
23     else if(s[i] == '[')
24     {
25         i++; // 把括号后面的字符串提取出来
26         string tmp = "";
27         while(s[i] >= 'a' && s[i] <= 'z')
28         {
29             tmp += s[i];
30             i++;
31         }
32         st.push(tmp);
33     }
34     else if(s[i] == ']')
35     {
36         string tmp = st.top();
37         st.pop();
38         int k = nums.top();
39         nums.pop();
40
41         while(k--)
42         {
43             st.top() += tmp;
44         }
45         i++; // 跳过这个右括号
46     }
47     else
48     {
49         string tmp;
50         while(i < n && s[i] >= 'a' && s[i] <= 'z')
51         {
52             tmp += s[i];
53             i++;
54         }
55         st.top() += tmp;
56     }
57 }
58 return st.top();
59 }
60 };
```

C++ 运行结果：

时间 4 ms

击败 32.33%

内存 6.5 MB

击败 26.63%

Java 算法代码：

```
1 class Solution
2 {
3     public String decodeString(String _s)
4     {
5         Stack<StringBuffer> st = new Stack<>();
6         st.push(new StringBuffer()); // 先放一个空串进去
7         Stack<Integer> nums = new Stack<>();
8
9         int i = 0, n = _s.length();
10        char[] s = _s.toCharArray();
11
12        while(i < n)
13        {
14            if(s[i] >= '0' && s[i] <= '9')
15            {
16                int tmp = 0;
17                while(i < n && s[i] >= '0' && s[i] <= '9')
18                {
19                    tmp = tmp * 10 + (s[i] - '0');
20                    i++;
21                }
22                nums.push(tmp);
23            }
24            else if(s[i] == '[')
25            {
26                i++; // 把后面的字符串提取出来
27                StringBuffer tmp = new StringBuffer();
28                while(i < n && s[i] >= 'a' && s[i] <= 'z')
29                {
30                    tmp.append(s[i]);
31                    i++;
32                }
33                st.push(tmp);
34            }
35            else if(s[i] == ']')
36            {
37                // 解析
```

```

38         StringBuffer tmp = st.pop();
39         int k = nums.pop();
40
41         while(k-- != 0)
42         {
43             st.peek().append(tmp);
44         }
45         i++;
46     }
47     else
48     {
49         StringBuffer tmp = new StringBuffer();
50         while(i < n && s[i] >= 'a' && s[i] <= 'z')
51         {
52             tmp.append(s[i]);
53             i++;
54         }
55         st.peek().append(tmp);
56     }
57 }
58 return st.peek().toString();
59 }
60 }
```

Java 运行结果：



69. 验证栈序列 (medium)

1. 题目链接：[946. 验证栈序列](#)

2. 题目描述：

给定 `pushed` 和 `popped` 两个序列，每个序列中的 值都不重复，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 `true`；否则，返回 `false`。

示例 1：

输入： `pushed` = [1,2,3,4,5], `popped` = [4,5,3,2,1]

输出： `true`

解释：我们可以按以下顺序执行：

`push(1), push(2), push(3), push(4), pop() -> 4,`
`push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1`

示例 2：

输入： `pushed = [1,2,3,4,5], popped = [4,3,5,1,2]`

输出： `false`

解释： 1 不能在 2 之前弹出。

提示：

- `1 <= pushed.length <= 1000`
- `0 <= pushed[i] <= 1000`
- `pushed` 的所有元素 **互不相同**
- `popped.length == pushed.length`
- `popped` 是 `pushed` 的一个排列

3. 解法（栈）：

算法思路：

用栈来模拟进出栈的流程。

一直让元素进栈，进栈的同时判断是否需要出栈。当所有元素模拟完毕之后，如果栈中还有元素，那么就是一个非法的序列。否则，就是一个合法的序列。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     bool validateStackSequences(vector<int>& pushed, vector<int>& popped)
5     {
6         stack<int> st;
7         int i = 0, n = popped.size();
8         for(auto x : pushed)
9         {
10             st.push(x);
11             while(st.size() && st.top() == popped[i])
12             {
13                 st.pop();
14                 i++;
15             }
16         }
17     }
18 }
```

```
16         }
17     return i == n;
18 }
19 };
```

C++ 运行结果：



Java 算法代码：

```
1 class Solution
2 {
3     public boolean validateStackSequences(int[] pushed, int[] popped)
4     {
5         Stack<Integer> st = new Stack<>();
6         int i = 0, n = popped.length;
7         for(int x : pushed)
8         {
9             st.push(x);
10            while(!st.isEmpty() && st.peek() == popped[i])
11            {
12                st.pop();
13                i++;
14            }
15        }
16        return i == n;
17    }
18 }
```

Java 运行结果：



队列 + 宽搜 (BFS)

70. N 叉树的层序遍历 (medium)

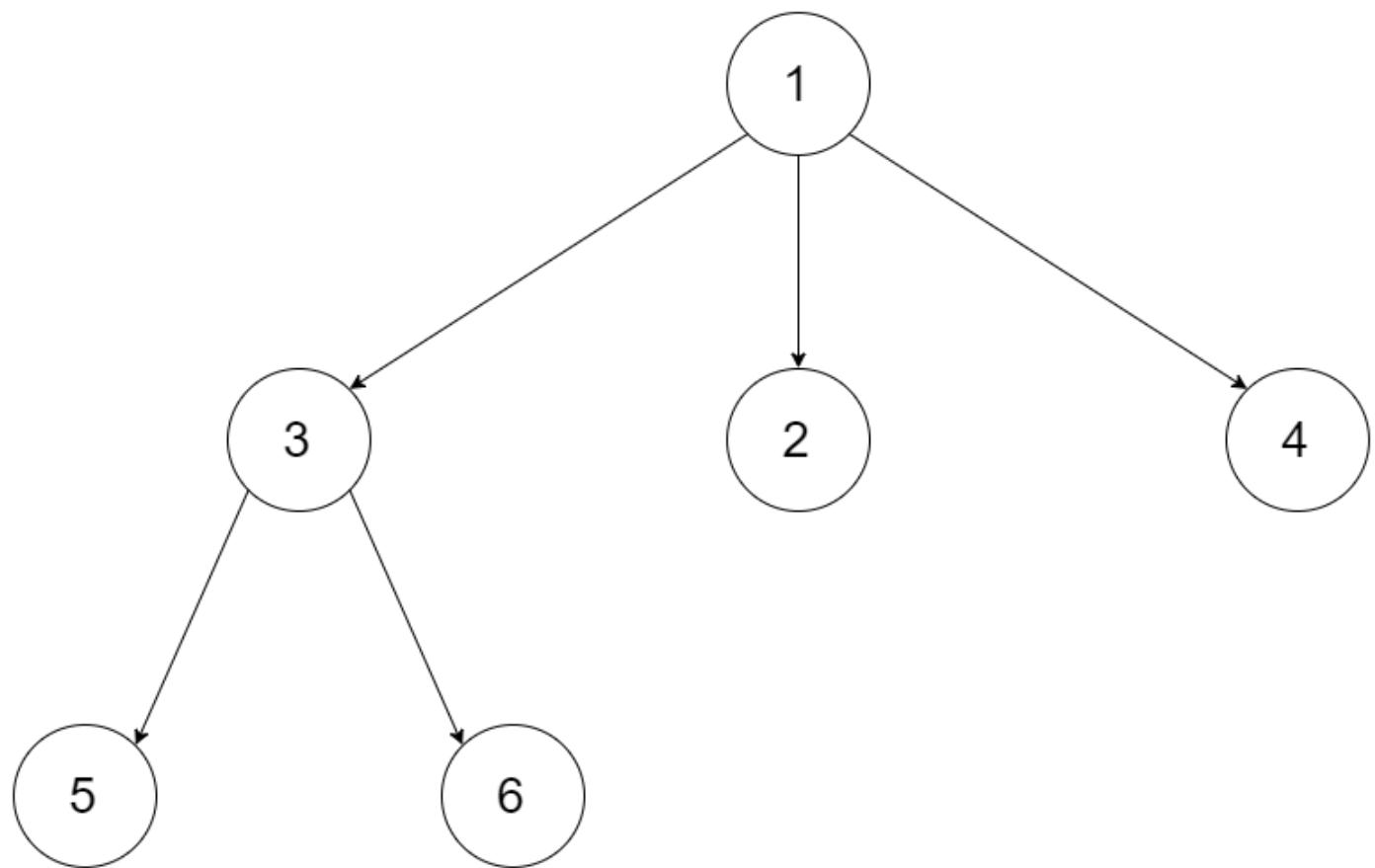
1. 题目链接: [429. N 叉树的层序遍历](#)

2. 题目描述:

给定一个 N 叉树，返回其节点值的层序遍历。 (即从左到右，逐层遍历)。

树的序列化输入是用层序遍历，每组子节点都由 null 值分隔 (参见示例)。

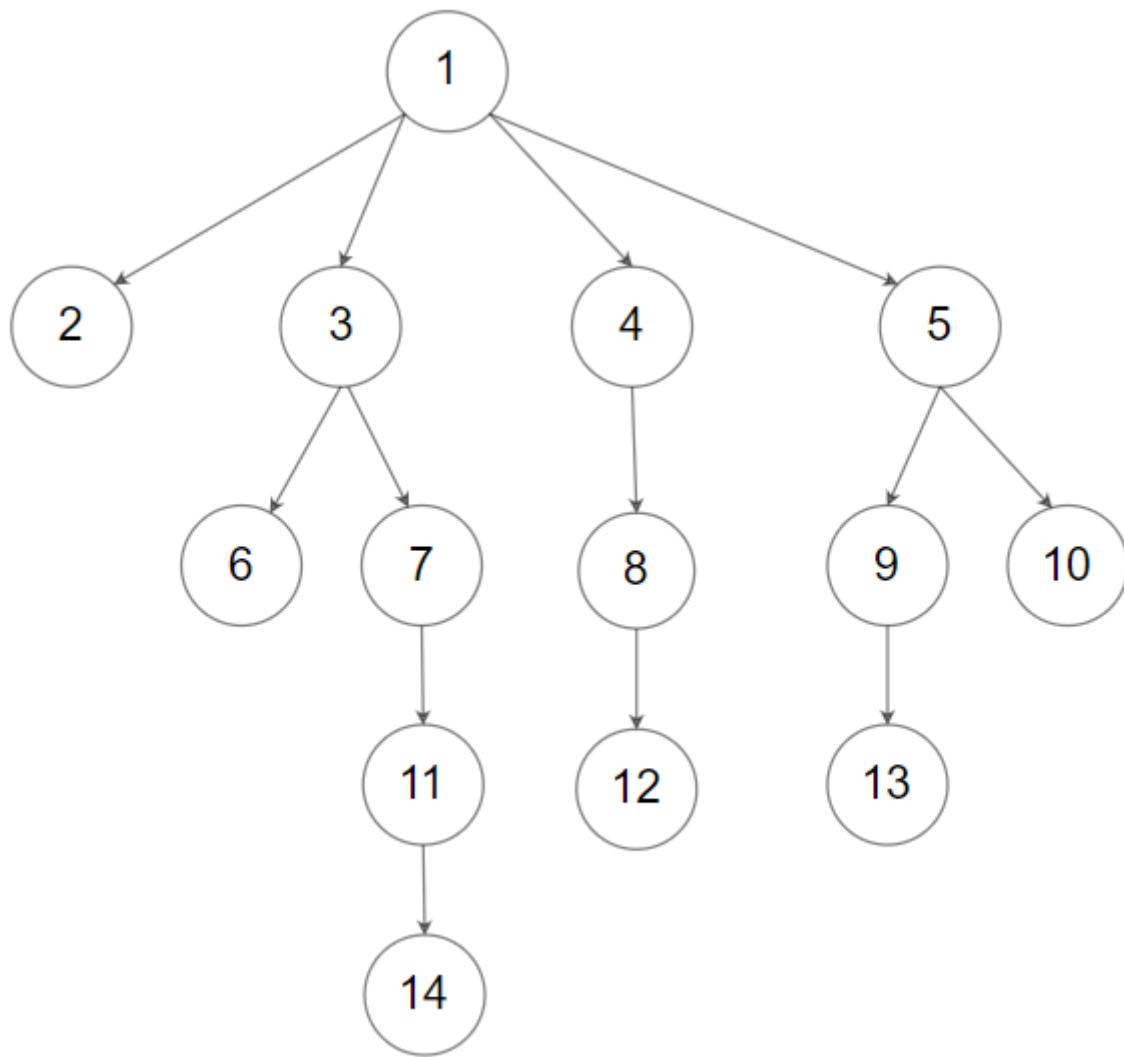
示例 1:



输入: `root = [1,null,3,2,4,null,5,6]`

输出: `[[1],[3,2,4],[5,6]]`

示例 2:



输入：root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

输出：[[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

提示：

- 树的高度不会超过 1000
- 树的节点总数在 [0, 10^4] 之间

3. 解法：

算法思路：

层序遍历即可~

仅需多加一个变量，用来记录每一层结点的个数就好了。

C++ 算法代码：

```

1 /*
2 // Definition for a Node.

```

```
3 class Node {
4 public:
5     int val;
6     vector<Node*> children;
7
8     Node() {}
9
10    Node(int _val) {
11        val = _val;
12    }
13
14    Node(int _val, vector<Node*> _children) {
15        val = _val;
16        children = _children;
17    }
18 };
19 */
20
21 class Solution
22 {
23 public:
24     vector<vector<int>> levelOrder(Node* root)
25     {
26         vector<vector<int>> ret; // 记录最终结果
27         queue<Node*> q; // 层序遍历需要的队列
28         if(root == nullptr) return ret;
29
30         q.push(root);
31         while(q.size())
32         {
33             int sz = q.size(); // 先求出本层元素的个数
34             vector<int> tmp; // 统计本层的节点
35             for(int i = 0; i < sz; i++)
36             {
37                 Node* t = q.front();
38                 q.pop();
39                 tmp.push_back(t->val);
40                 for(Node* child : t->children) // 让下一层结点入队
41                 {
42                     if(child != nullptr)
43                         q.push(child);
44                 }
45             }
46             ret.push_back(tmp);
47         }
48
49         return ret;
50 }
```

```
50     }
51 };
```

C++ 代码结果：



Java 算法代码：

```
1 /*
2 // Definition for a Node.
3 class Node {
4     public int val;
5     public List<Node> children;
6
7     public Node() {}
8
9     public Node(int _val) {
10         val = _val;
11     }
12
13     public Node(int _val, List<Node> _children) {
14         val = _val;
15         children = _children;
16     }
17 }
18 */
19
20 class Solution
21 {
22     public List<List<Integer>> levelOrder(Node root)
23     {
24         List<List<Integer>> ret = new ArrayList<>();
25         if(root == null) return ret;
26         Queue<Node> q = new LinkedList<>();
27         q.add(root);
28
29         while(!q.isEmpty())
30         {
31             int sz = q.size();
```

```
32         List<Integer> tmp = new ArrayList<>(); // 统计本层的结点信息
33         for(int i = 0; i < sz; i++)
34         {
35             Node t = q.poll();
36             tmp.add(t.val);
37             for(Node child : t.children) // 让孩子入队
38             {
39                 if(child != null)
40                     q.add(child);
41             }
42         }
43         ret.add(tmp);
44     }
45
46     return ret;
47 }
48 }
```

Java 运行结果：



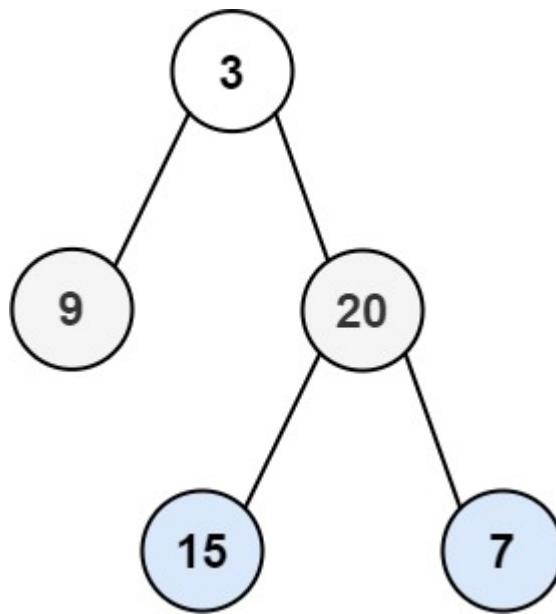
71. 二叉树的锯齿形层序遍历 (medium)

1. 题目链接：[103. 二叉树的锯齿形层序遍历](#)

2. 题目描述：

给你二叉树的根节点 root，返回其节点值的 锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

示例 1：



输入：root = [3,9,20,null,null,15,7]

输出：[[3],[20,9],[15,7]]

示例 2：

输入：root = [1]

输出：[[1]]

示例 3：

输入：root = []

输出：[]

3. 解法（层序遍历）：

算法思路：

在正常的层序遍历过程中，我们是可以把一层的结点放在一个数组中去的。

既然我们有这个数组，在合适的层数逆序就可以得到锯齿形层序遍历的结果。

C++ 算法代码：

```

1 /**
2 * Definition for a binary tree node.
3 * struct TreeNode {
4 *     int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
  
```

```

9     *      TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
10    *      right(right) {}
11    */
12 class Solution
13 {
14 public:
15     vector<vector<int>> zigzagLevelOrder(TreeNode* root)
16     {
17         vector<vector<int>> ret;
18         if(root == nullptr) return ret;
19         queue<TreeNode*> q;
20         q.push(root);
21         int level = 1;
22
23         while(q.size())
24         {
25             int sz = q.size();
26             vector<int> tmp;
27             for(int i = 0; i < sz; i++)
28             {
29                 auto t = q.front();
30                 q.pop();
31                 tmp.push_back(t->val);
32                 if(t->left) q.push(t->left);
33                 if(t->right) q.push(t->right);
34             }
35             // 判断是否逆序
36             if(level % 2 == 0) reverse(tmp.begin(), tmp.end());
37             ret.push_back(tmp);
38             level++;
39         }
40
41         return ret;
42     }
43 };

```

C++ 运行结果：

C++



Java 算法代码：

```
1  /**
2  * Definition for a binary tree node.
3  * public class TreeNode {
4  *     int val;
5  *     TreeNode left;
6  *     TreeNode right;
7  *     TreeNode() {}
8  *     TreeNode(int val) { this.val = val; }
9  *     TreeNode(int val, TreeNode left, TreeNode right) {
10 *         this.val = val;
11 *         this.left = left;
12 *         this.right = right;
13 *     }
14 * }
15 */
16 class Solution
17 {
18     public List<List<Integer>> zigzagLevelOrder(TreeNode root)
19     {
20         List<List<Integer>> ret = new ArrayList<>();
21         if(root == null) return ret;
22         Queue<TreeNode> q = new LinkedList<>();
23         q.add(root);
24         int level = 1;
25
26         while(!q.isEmpty())
27         {
28             int sz = q.size();
29             List<Integer> tmp = new ArrayList<>();
30             for(int i = 0; i < sz; i++)
31             {
32                 TreeNode t = q.poll();
33                 tmp.add(t.val);
34                 if(t.left != null) q.add(t.left);
35                 if(t.right != null) q.add(t.right);
36             }
37             // 判断是否逆序
38             if(level % 2 == 0) Collections.reverse(tmp);
39             ret.add(tmp);
40             level++;
41         }
42
43         return ret;
44     }
45 }
```

Java 运行结果：



72. 二叉树的最大宽度 (medium)

1. 题目链接：[662. 二叉树最大宽度](#)

2. 题目描述：

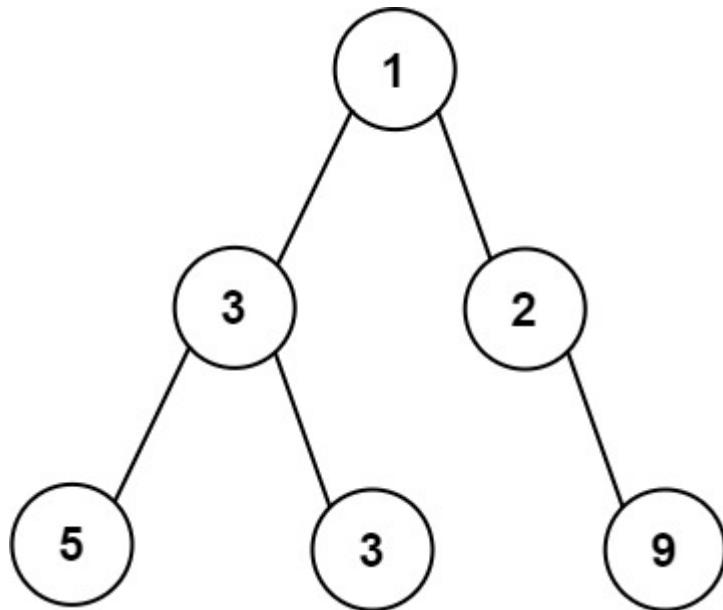
给你一棵二叉树的根节点 root，返回树的最大宽度。

树的最大宽度是所有层中最大的宽度。

每一层的宽度被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 null 节点，这些 null 节点也计入长度。

题目数据保证答案将会在 32 位带符号整数范围内。

示例 1：



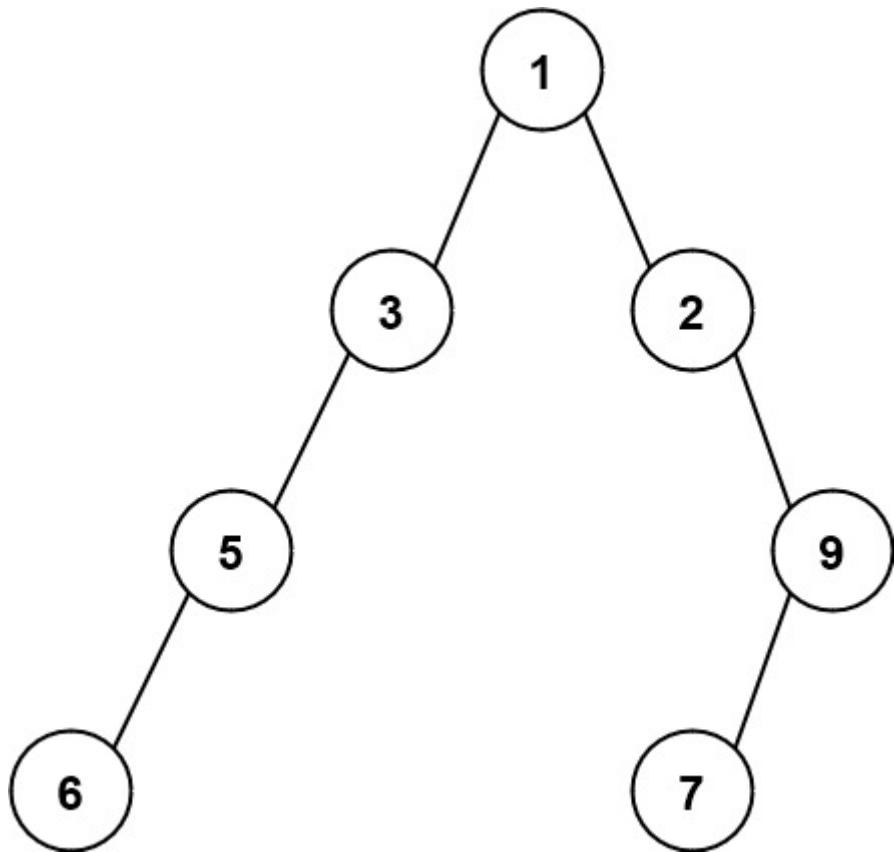
输入：root = [1,3,2,5,3,null,9]

输出：4

解释：

最大宽度出现在树的第 3 层，宽度为 4 (5,3,null,9)。

示例 2：



输入：root = [1,3,2,5,null,null,9,6,null,7]

输出：7

解释：

最大宽度出现在树的第 4 层，宽度为 7 (6,null,null,null,null,null,7)。

3. 解法（层序遍历）：

算法思路：

1. 第一种思路（会超过内存限制）：

既然统计每一层的最大宽度，我们优先想到的就是利用层序遍历，把当前层的结点全部存在队列里面，利用队列的长度来计算每一层的宽度，统计出最大的宽度。

但是，由于空节点也是需要计算在内的。因此，我们可以选择将**空节点也存在队列里面**。

这个思路是我们正常会想到的思路，但是极端境况下，最左边一条长链，最右边一条长链，我们需要存几亿个空节点，会超过最大内存限制。

2. 第二种思路（利用二叉树的顺序存储 - 通过根节点的下标，计算左右孩子的下标）：

依旧是利用层序遍历，但是这一次队列里面不单单存结点信息，并且还存储当前结点如果在数组中存储所对应的下标（在我们学习数据结构 - 堆的时候，计算左右孩子的方式）。

这样我们计算每一层宽度的时候，无需考虑空节点，只需将当层结点的左右结点的下标相减再加 1 即可。

但是，这里有个细节问题：如果二叉树的层数非常恐怖的话，我们任何一种数据类型都不能存下下标的值。但是没有问题，因为

- 我们数据的存储是一个环形的结构；
- 并且题目说明，数据的范围在 `int` 这个类型的最大值的范围之内，因此不会超出一圈；
- 因此，如果是求差值的话，我们无需考虑溢出的情况。

C++ 算法代码：

```
1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8  *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9  *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
10    right(right) {}
11 };
12 class Solution
13 {
14 public:
15     int widthOfBinaryTree(TreeNode* root)
16     {
17         vector<pair<TreeNode*, unsigned int>> q; // 用数组模拟队列
18         q.push_back({root, 1});
19         unsigned int ret = 0;
20
21         while(q.size())
22         {
23             // 先更新这一层的宽度
24             auto& [x1, y1] = q[0];
25             auto& [x2, y2] = q.back();
26             ret = max(ret, y2 - y1 + 1);
27
28             // 让下一层进队
29             vector<pair<TreeNode*, unsigned int>> tmp; // 让下一层进入这个队列
30             for(auto& [x, y] : q)
```

```

31         {
32             if(x->left) tmp.push_back({x->left, y * 2});
33             if(x->right) tmp.push_back({x->right, y * 2 + 1});
34         }
35         q = tmp;
36     }
37
38     return ret;
39 }
40 };

```

C++ 运行结果：



Java 算法代码：

```

1 /**
2  * Definition for a binary tree node.
3  * public class TreeNode {
4  *     int val;
5  *     TreeNode left;
6  *     TreeNode right;
7  *     TreeNode() {}
8  *     TreeNode(int val) { this.val = val; }
9  *     TreeNode(int val, TreeNode left, TreeNode right) {
10    *         this.val = val;
11    *         this.left = left;
12    *         this.right = right;
13    *     }
14  * }
15 */
16 class Solution
17 {
18     public int widthOfBinaryTree(TreeNode root)
19     {
20         List<Pair<TreeNode, Integer>> q = new ArrayList<>(); // 用数组模拟队列
21         q.add(new Pair<TreeNode, Integer>(root, 1));
22         int ret = 0; // 记录最终结果
23     }

```

```

24     while(!q.isEmpty())
25     {
26         // 先更新一下这一层的宽度
27         Pair<TreeNode, Integer> t1 = q.get(0);
28         Pair<TreeNode, Integer> t2 = q.get(q.size() - 1);
29         ret = Math.max(ret, t2.getValue() - t1.getValue() + 1);
30
31         // 让下一层进队
32         List<Pair<TreeNode, Integer>> tmp = new ArrayList<>();
33         for(Pair<TreeNode, Integer> t : q)
34         {
35             TreeNode node = t.getKey();
36             int index = t.getValue();
37             if(node.left != null)
38             {
39                 tmp.add(new Pair<TreeNode, Integer>(node.left, index * 2));
40             }
41             if(node.right != null)
42             {
43                 tmp.add(new Pair<TreeNode, Integer>(node.right, index * 2
44 + 1));
45             }
46         }
47         q = tmp;
48     }
49 }
50 }
```

Java 运行结果：



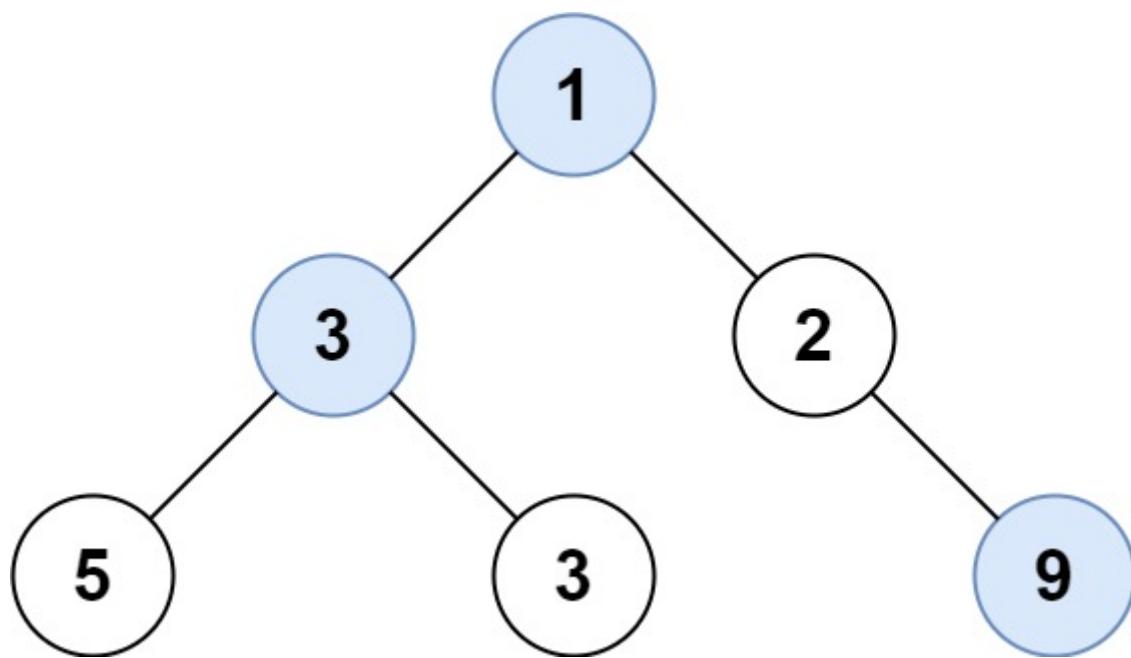
73. 在每个树行中找最大值 (medium)

1. 题目链接：515. 在每个树行中找最大值

2. 题目描述：

给定一棵二叉树的根节点 root，请找出该二叉树中每一层的最大值。

示例 1：



输入: root = [1,3,2,5,3,null,9]

输出: [1,3,9]

示例 2:

输入: root = [1,2,3]

输出: [1,3]

3. 解法 (bfs) :

算法思路:

层序遍历过程中，在执行让下一层节点入队的时候，我们是可以在循环中统计出当前层结点的最大值的。

因此，可以在 **bfs** 的过程中，统计出每一层结点的最大值。

C++ 算法代码:

```

1  /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8  *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9  *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
10    right(right) {}
11 };

```

```

11  */
12 class Solution
13 {
14 public:
15     vector<int> largestValues(TreeNode* root)
16     {
17         vector<int> ret;
18         if(root == nullptr) return ret;
19
20         queue<TreeNode*> q;
21         q.push(root);
22
23         while(q.size())
24         {
25             int sz = q.size();
26             int tmp = INT_MIN;
27             for(int i = 0; i < sz; i++)
28             {
29                 auto t = q.front();
30                 q.pop();
31                 tmp = max(tmp, t->val);
32                 if(t->left) q.push(t->left);
33                 if(t->right) q.push(t->right);
34             }
35
36             ret.push_back(tmp);
37         }
38         return ret;
39     }
40 };

```

C++ 运行结果:



Java 算法代码:

```

1 /**
2 * Definition for a binary tree node.
3 * public class TreeNode {
4 *     int val;

```

```

5  *      TreeNode left;
6  *      TreeNode right;
7  *      TreeNode() {}
8  *      TreeNode(int val) { this.val = val; }
9  *      TreeNode(int val, TreeNode left, TreeNode right) {
10 *          this.val = val;
11 *          this.left = left;
12 *          this.right = right;
13 *      }
14 * }
15 */
16 class Solution
17 {
18     public List<Integer> largestValues(TreeNode root)
19     {
20         List<Integer> ret = new ArrayList<>();
21         if(root == null) return ret;
22
23         Queue<TreeNode> q = new LinkedList<>();
24         q.add(root);
25
26         while(!q.isEmpty())
27         {
28             int sz = q.size();
29             int tmp = Integer.MIN_VALUE;
30             for(int i = 0; i < sz; i++)
31             {
32                 TreeNode t = q.poll();
33                 tmp = Math.max(tmp, t.val);
34                 if(t.left != null) q.add(t.left);
35                 if(t.right != null) q.add(t.right);
36             }
37             ret.add(tmp);
38         }
39
40         return ret;
41     }
42 }

```

Java 运行结果：

Java



优先级队列（堆）

74. 最后一块石头的重量 (easy)

1. 题目链接：[1046. 最后一块石头的重量](#)

2. 题目描述：

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出两块最重的石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y ，且 $x \leq y$ 。那么粉碎的可能结果如下：

- 如果 $x == y$ ，那么两块石头都会被完全粉碎；
- 如果 $x != y$ ，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 $y - x$ 。

最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 0。

示例：

输入： [2,7,4,1,8,1]

输出： 1

解释：

先选出 7 和 8，得到 1，所以数组转换为 [2,4,1,1,1]，

再选出 2 和 4，得到 2，所以数组转换为 [2,1,1,1]，

接着是 2 和 1，得到 1，所以数组转换为 [1,1,1]，

最后选出 1 和 1，得到 0，最终数组转换为 [1]，这就是最后剩下那块石头的重量。

提示：

$1 \leq \text{stones.length} \leq 30$

$1 \leq \text{stones}[i] \leq 1000$

3. 解法（利用堆）：

算法思路：

其实就是一个模拟的过程：

- 每次从石堆中拿出最大的元素以及次大的元素，然后将它们粉碎；
- 如果还有剩余，就将剩余的石头继续放在原始的石堆里面

重复上面的操作，直到石堆里面只剩下一个元素，或者没有元素（因为所有的石头可能全部抵消了）

那么主要的问题就是解决：

- 如何顺利的拿出最大的石头以及次大的石头；
- 并且将粉碎后的石头放入石堆中之后，也能快速找到下一轮粉碎的最大石头和次大石头；

这不正好可以利用堆的特性来实现嘛？

- 我们可以创建一个大根堆；
- 然后将所有的石头放入大根堆中；
- 每次拿出前两个堆顶元素粉碎一下，如果还有剩余，就将剩余的石头继续放入堆中；

这样就能快速的模拟出这个过程。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int lastStoneWeight(vector<int>& stones)
5     {
6         // 1. 创建一个大根堆
7         priority_queue<int> heap;
8         // 2. 将所有元素丢进这个堆里面
9         for(auto x : stones) heap.push(x);
10        // 3. 模拟这个过程
11        while(heap.size() > 1)
12        {
13            int a = heap.top(); heap.pop();
14            int b = heap.top(); heap.pop();
15            if(a > b) heap.push(a - b);
16        }
17
18        return heap.size() ? heap.top() : 0;
19    }
20};
```

C++ 运行结果：

C++



Java 算法代码：

```
1 class Solution
2 {
3     public int lastStoneWeight(int[] stones)
4     {
5         // 1. 创建一个大根堆
6         PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> b - a);
7         // 2. 把所有的石头放进堆里面
8         for(int x : stones)
9         {
10             heap.offer(x);
11         }
12         // 3. 模拟
13         while(heap.size() > 1)
14         {
15             int a = heap.poll();
16             int b = heap.poll();
17             if(a > b)
18             {
19                 heap.offer(a - b);
20             }
21         }
22
23         return heap.isEmpty() ? 0 : heap.peek();
24     }
25 }
```

Java 运行结果：

Java



75. 数据流中的第 K 大元素 (easy)

1. 题目链接: [703. 数据流中的第 K 大元素](#)

2. 题目描述:

设计一个找到数据流中第 k 大元素的类 (class)。注意是排序后的第 k 大元素，不是第 k 个不同的元素。

请实现 `KthLargest` 类：

`KthLargest(int k, int[] nums)` 使用整数 k 和整数流 `nums` 初始化对象。

`int add(int val)` 将 `val` 插入数据流 `nums` 后，返回当前数据流中第 k 大的元素。

示例：

输入：

```
["KthLargest", "add", "add", "add", "add", "add", "add"]
```

```
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
```

输出：

```
[null, 4, 5, 5, 8, 8]
```

解释：

```
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
```

```
kthLargest.add(3); // return 4
```

```
kthLargest.add(5); // return 5
```

```
kthLargest.add(10); // return 5
```

```
kthLargest.add(9); // return 8
```

```
kthLargest.add(4); // return 8
```

提示：

$1 \leq k \leq 10^4$

$0 \leq \text{nums.length} \leq 10^4$

$-10^4 \leq \text{nums}[i] \leq 10^4$

$-10^4 \leq \text{val} \leq 10^4$

最多调用 `add` 方法 10^4 次

题目数据保证，在查找第 k 大元素时，数组中至少有 k 个元素

3. 解法（优先级队列）：

算法思路：

我相信，看到 TopK 问题的时候，兄弟们应该能立马想到「堆」，这应该是刻在骨子里的记忆。

C++ 算法代码：

```
1 class KthLargest
2 {
3     // 创建一个大小为 k 的小跟堆
4     priority_queue<int, vector<int>, greater<int>> heap;
5     int _k;
6
7 public:
8     KthLargest(int k, vector<int>& nums)
9     {
10         _k = k;
11         for(auto x : nums)
12         {
13             heap.push(x);
14             if(heap.size() > _k) heap.pop();
15         }
16     }
17
18     int add(int val)
19     {
20         heap.push(val);
21         if(heap.size() > _k) heap.pop();
22         return heap.top();
23     }
24 };
25
26 /**
27 * Your KthLargest object will be instantiated and called as such:
28 * KthLargest* obj = new KthLargest(k, nums);
29 * int param_1 = obj->add(val);
30 */
```

C++ 运行结果：

C++



Java 算法代码：

```
1 class KthLargest
2 {
3     // 创建一个大小为 k 的小根堆
4     PriorityQueue<Integer> heap;
5     int _k;
6
7     public KthLargest(int k, int[] nums)
8     {
9         _k = k;
10        heap = new PriorityQueue<>();
11        for(int x : nums)
12        {
13            heap.offer(x);
14            if(heap.size() > _k)
15            {
16                heap.poll();
17            }
18        }
19    }
20
21    public int add(int val)
22    {
23        heap.offer(val);
24        if(heap.size() > _k)
25        {
26            heap.poll();
27        }
28        return heap.peek();
29    }
30 }
31
32 /**
33 * Your KthLargest object will be instantiated and called as such:
34 * KthLargest obj = new KthLargest(k, nums);
35 * int param_1 = obj.add(val);
36 */
```

Java 运行结果：

Java



76. 前 K 个高频单词 (medium)

1. 题目链接：[692. 前 K 个高频单词](#)

2. 题目描述：

给定一个单词列表 words 和一个整数 k，返回前 k 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字典顺序排序。

示例 1：

输入：

```
words = ["i", "love", "leetcode", "i", "love", "coding"], k = 2
```

输出：

```
["i", "love"]
```

解析：

"i" 和 "love" 为出现次数最多的两个单词，均为 2 次。

注意，按字母顺序 "i" 在 "love" 之前。

示例 2：

输入：

```
["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4
```

输出：

```
["the", "is", "sunny", "day"]
```

解析：

"the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，

出现次数依次为 4, 3, 2 和 1 次。

注意：

$1 \leq \text{words.length} \leq 500$

$1 \leq \text{words[i]} \leq 10$

words[i] 由小写英文字母组成。

k 的取值范围是 $[1, \text{不同 words[i] 的数量}]$

进阶：尝试以 $O(n \log k)$ 时间复杂度和 $O(n)$ 空间复杂度解决。

3. 解法（堆）：

算法思路：

- **稍微处理一下原数组：**

- 我们需要知道每一个单词出现的频次，因此可以先使用哈希表，统计出每一个单词出现的频次；
- 然后在哈希表中，选出前 k 大的单词（为什么不在原数组中选呢？因为原数组中存在重复的单词，哈希表里面没有重复单词，并且还有每一个单词出现的频次）

- **如何使用堆，拿出前 k 大元素：**

- 先定义一个自定义排序，我们需要的是前 k 大，因此需要一个小根堆。但是当两个字符串的频次相同的时候，我们需要的是字典序较小的，此时是一个大根堆的属性，在定义比较器的时候需要注意！

- 当两个字符串出现的频次不同的时候：需要的是基于频次比较的小根堆
- 当两个字符串出现的频次相同的时候：需要的是基于字典序比较的大根堆

- 定义好比较器之后，依次将哈希表中的字符串插入到堆中，维持堆中的元素不超过 k 个；
- 遍历完整个哈希表后，堆中的剩余元素就是前 k 大的元素

C++ 算法代码：

```
1 class Solution
2 {
3     typedef pair<string, int> PSI;
4
5     struct cmp
6     {
7         bool operator()(const PSI& a, const PSI& b)
8         {
```

```

9             if(a.second == b.second) // 频次相同，字典序按照大根堆的方式排列
10            {
11                return a.first < b.first;
12            }
13            return a.second > b.second;
14        }
15    };
16
17 public:
18     vector<string> topKFrequent(vector<string>& words, int k)
19     {
20         // 1. 统计一下每一个单词的频次
21         unordered_map<string, int> hash;
22         for(auto& s : words) hash[s]++;
23
24         // 2. 创建一个大小为 k 的堆
25         priority_queue<PSI, vector<PSI>, cmp> heap;
26
27         // 3. TopK 的主逻辑
28         for(auto& psi : hash)
29         {
30             heap.push(psi);
31             if(heap.size() > k) heap.pop();
32         }
33
34         // 4. 提取结果
35         vector<string> ret(k);
36         for(int i = k - 1; i >= 0; i--)
37         {
38             ret[i] = heap.top().first;
39             heap.pop();
40         }
41         return ret;
42     }
43 };

```

C++ 运行结果：

C++



Java 算法代码：

```
1 class Solution
2 {
3     public List<String> topKFrequent(String[] words, int k)
4     {
5         // 1. 统计一下每一个单词出现的频次
6         Map<String, Integer> hash = new HashMap<>();
7         for(String s : words)
8         {
9             hash.put(s, hash.getOrDefault(s, 0) + 1);
10        }
11
12        // 2. 创建一个大小为 k 的堆
13        PriorityQueue<Pair<String, Integer>> heap = new PriorityQueue<>
14        (
15            (a, b) ->
16            {
17                if(a.getValue().equals(b.getValue())) // 频次相同的时候，字典序按照
18                    大根堆的方式排列
19                {
20                    return b.getKey().compareTo(a.getKey());
21                }
22                return a.getValue() - b.getValue();
23            }
24        );
25
26        // 3. TopK 的主逻辑
27        for(Map.Entry<String, Integer> e : hash.entrySet())
28        {
29            heap.offer(new Pair<>(e.getKey(), e.getValue()));
30            if(heap.size() > k)
31            {
32                heap.poll();
33            }
34        }
35
36        // 4. 提取结果
37        List<String> ret = new ArrayList<>();
38        while(!heap.isEmpty())
39        {
40            ret.add(heap.poll().getKey());
41        }
42        // 逆序数组
43        Collections.reverse(ret);
44        return ret;
45    }
}
```

Java 运行结果：



77. 数据流的中位数 (hard)

1. 题目链接：295. 数据流的中位数

2. 题目描述：

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 $\text{arr} = [2,3,4]$ 的中位数是 3。
- 例如 $\text{arr} = [2,3]$ 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。

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
```

提示:

$-10^5 \leq num \leq 10^5$

在调用 `findMedian` 之前，数据结构中至少有一个元素

最多 $5 * 10^4$ 次调用 `addNum` 和 `findMedian`

3. 解法（利用两个堆）：

算法思路：

这是一道关于「堆」这种数据结构的一个「经典应用」。

我们可以将整个数组「按照大小」平分成两部分（如果不能平分，那就让较小部分的元素多一个），较小的部分称为左侧部分，较大的部分称为右侧部分：

- 将左侧部分放入「大根堆」中，然后将右侧元素放入「小根堆」中；
- 这样就能在 $O(1)$ 的时间内拿到中间的一个数或者两个数，进而求的平均数。

如下图所示：



于是问题就变成了「如何将一个一个从数据流中过来的数据，动态调整到大根堆或者小根堆中，并且保证两个堆的元素一致，或者左侧堆的元素比右侧堆的元素多一个」

为了方便叙述，将左侧的「大根堆」记为 `left`，右侧的「小根堆」记为 `right`，数据流中来的「数据」记为 `x`。

其实，就是一个「分类讨论」的过程：

1. 如果左右堆的「数量相同」，`left.size() == right.size()`：

- a. 如果两个堆都是空的，直接将数据 `x` 放入到 `left` 中；

b. 如果两个堆非空：

i. 如果元素要放入左侧，也就是 `x <= left.top()`：那就直接放，因为不会影响我们制定的规则；

ii. 如果要放入右侧

- 可以先将 `x` 放入 `right` 中，
- 然后把 `right` 的堆顶元素放入 `left` 中；

2. 如果左右堆的数量「不相同」，那就是 `left.size() > right.size()`：

a. 这个时候我们关心的是 `x` 是否会放入 `left` 中，导致 `left` 变得过多：

i. 如果 `x` 放入 `right` 中，也就是 `x >= right.top()`，直接放；

ii. 反之，就是需要放入 `left` 中：

- 可以先将 `x` 放入 `left` 中，
- 然后把 `left` 的堆顶元素放入 `right` 中；

只要每一个新来的元素按照「上述规则」执行，就能保证 `left` 中放着整个数组排序后的「左半部分」，`right` 中放着整个数组排序后的「右半部分」，就能在 $O(1)$ 的时间内求出平均数。

C++ 算法代码：

```
1 class MedianFinder
2 {
3     priority_queue<int> left; // 大根堆
4     priority_queue<int, vector<int>, greater<int>> right; // 小根堆
5
6 public:
7     MedianFinder()
8     {}
9
10    void addNum(int num)
11    {
12        // 分类讨论即可
13        if(left.size() == right.size()) // 左右两个堆的元素个数相同
14        {
15            if(left.empty() || num <= left.top()) // 放 left 里面
16            {
17                left.push(num);
18            }
19        } else
20        {
```

```

21             right.push(num);
22             left.push(right.top());
23             right.pop();
24         }
25     }
26     else
27     {
28         if(num <= left.top())
29         {
30             left.push(num);
31             right.push(left.top());
32             left.pop();
33         }
34         else
35         {
36             right.push(num);
37         }
38     }
39 }
40
41     double findMedian()
42     {
43         if(left.size() == right.size()) return (left.top() + right.top()) /
44             2.0;
45         else return left.top();
46     }
47
48 /**
49 * Your MedianFinder object will be instantiated and called as such:
50 * MedianFinder* obj = new MedianFinder();
51 * obj->addNum(num);
52 * double param_2 = obj->findMedian();
53 */

```

C++ 运行结果：

C++



Java 算法代码：

```
1 class MedianFinder
2 {
3     PriorityQueue<Integer> left;
4     PriorityQueue<Integer> right;
5
6     public MedianFinder()
7     {
8         left = new PriorityQueue<Integer>((a, b) -> b - a); // 大根堆
9         right = new PriorityQueue<Integer>((a, b) -> a - b); // 小根堆
10    }
11
12    public void addNum(int num)
13    {
14        // 分情况讨论
15        if(left.size() == right.size())
16        {
17            if(left.isEmpty() || num <= left.peek())
18            {
19                left.offer(num);
20            }
21            else
22            {
23                right.offer(num);
24                left.offer(right.poll());
25            }
26        }
27        else
28        {
29            if(num <= left.peek())
30            {
31                left.offer(num);
32                right.offer(left.poll());
33            }
34            else
35            {
36                right.offer(num);
37            }
38        }
39    }
40
41    public double findMedian()
42    {
43        if(left.size() == right.size()) return (left.peek() + right.peek()) /
44        2.0;
45        else return left.peek();
46    }
}
```

```
47
48 /**
49 * Your MedianFinder object will be instantiated and called as such:
50 * MedianFinder obj = new MedianFinder();
51 * obj.addNum(num);
52 * double param_2 = obj.findMedian();
53 */
```

Java 运行结果：

