

# intern手撕

---

[华为OD面试手撕真题 - 二叉树的层序遍历 II \\_od手撕考二叉树多不多-CSDN博客](#)

[华为OD 面试手撕真题 - 连接棒材的最低费用-CSDN博客](#)

[华为OD面试手撕真题 - 统计各个数字都不同的数字个数 \(C++ & Python & JAVA & JS & GO\)-CSDN博客](#)

[2025华为OD面试手撕真题 - 验证回文串 II -CSDN博客](#)

[华为OD面试手撕真题 - 拆分字符串使唯一子字符串的数目最大-CSDN博客](#)

[华为OD面试手撕真题 - 最大数-CSDN博客](#)

[华为OD面试手撕真题 - 字符串解码 \(C++ & Python & JAVA & JS & GO\)-CSDN博客](#)

[华为OD面试手撕真题 - 最长递增子序列-CSDN博客](#)

[华为OD面试手撕真题 - 课程表 II -CSDN博客](#)

[华为OD面试手撕真题 - 子集-CSDN博客](#)

[求1-n的最小公倍数-CSDN博客](#)

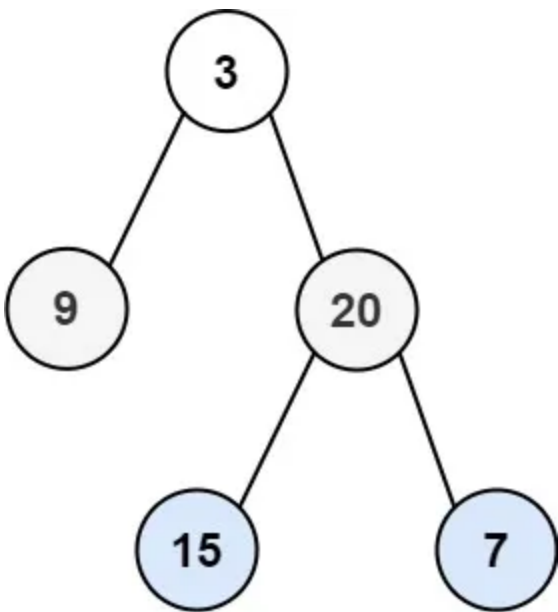
# 华为OD面试手撕真题 - 二叉树的层序遍历 II\_od

## 手撕考二叉树多不多-CSDN博客

### 题目描述

给你二叉树的根节点 `root` ，返回其节点值 自底向上的层序遍历 。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

### 示例1



▼ Plain Text |

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

### 示例2

▼ Plain Text |

```
1  输入: root = [1]
2  输出: [[1]]
```

### 示例3

```
1  输入: root = []  
2  输出: []
```

## 提示

提示：

- 树中节点数目在范围 `[0, 2000]` 内
- `-1000 <= Node.val <= 1000`

## 题解

[力扣原题链接](#)

思路： 队列数据结构运用

1. 层序遍历采用 队列 进行模拟。
2. 从下到上层序遍历 和 从上到下遍历 的结果差异就是 结果顺序相反。可以使用我们熟悉的从上到下  
进行层序遍历，最后将结果反转即可。

**C++**

```
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), right(right) {}
10  * };
11  */
12  class Solution {
13  public:
14      vector<vector<int>> levelOrderBottom(TreeNode* root) {
15          vector<vector<int>> res;
16          if (root == nullptr) {
17              return res;
18          }
19          // 使用队列模拟层序遍历
20          queue<TreeNode*> q;
21          q.push(root);
22          while (!q.empty()) {
23              queue<TreeNode*> tmpQueue;
24              // 存储每一层遍历的值
25              vector<int> valueArr;
26              while (!q.empty()) {
27                  TreeNode* current = q.front();
28                  q.pop();
29                  valueArr.push_back(current->val);
30                  // 加入左节点
31                  if (current->left != nullptr) {
32                      tmpQueue.push(current->left);
33                  }
34                  // 加入右节点
35                  if (current->right != nullptr) {
36                      tmpQueue.push(current->right);
37                  }
38              }
39
40              res.push_back(valueArr);
41              q = tmpQueue;
42          }
43          // 从上到下遍历 反转就是 从下到上
44          reverse(res.begin(), res.end());
```

```
45         return res;
46     }
47 };
```

## JAVA

```
▼ Plain Text |
1  class Solution {
2      public List<List<Integer>> levelOrderBottom(TreeNode root) {
3          List<List<Integer>> res = new ArrayList<>();
4          if (root == null) {
5              return res;
6          }
7
8          // 使用队列模拟层序遍历
9          Queue<TreeNode> q = new LinkedList<>();
10         q.offer(root);
11
12         while (!q.isEmpty()) {
13             Queue<TreeNode> tmpQueue = new LinkedList<>();
14             List<Integer> valueArr = new ArrayList<>();
15
16             while (!q.isEmpty()) {
17                 TreeNode current = q.poll();
18                 valueArr.add(current.val);
19
20                 // 加入左节点
21                 if (current.left != null) {
22                     tmpQueue.offer(current.left);
23                 }
24                 // 加入右节点
25                 if (current.right != null) {
26                     tmpQueue.offer(current.right);
27                 }
28             }
29
30             res.add(valueArr);
31             q = tmpQueue;
32         }
33
34         // 从上到下遍历, 反转就是从下到上
35         Collections.reverse(res);
36         return res;
37     }
38 }
```

## Python



Plain Text

```
1  from collections import deque
2
3  class Solution:
4      def levelOrderBottom(self, root):
5          res = []
6          if not root:
7              return res
8
9          # 使用队列模拟层序遍历
10         q = deque([root])
11
12         while q:
13             tmp_queue = deque()
14             value_arr = []
15
16             while q:
17                 current = q.popleft()
18                 value_arr.append(current.val)
19
20                 # 加入左节点
21                 if current.left:
22                     tmp_queue.append(current.left)
23                 # 加入右节点
24                 if current.right:
25                     tmp_queue.append(current.right)
26
27             res.append(value_arr)
28             q = tmp_queue
29
30         # 从上到下遍历，反转就是从下到上
31         return res[::-1]
```

## JavaScript

```
1  /**
2   * Definition for a binary tree node.
3   * function TreeNode(val, left, right) {
4   *     this.val = (val===undefined ? 0 : val)
5   *     this.left = (left===undefined ? null : left)
6   *     this.right = (right===undefined ? null : right)
7   * }
8   */
9  /**
10   * @param {TreeNode} root
11   * @return {number[][]}
12   */
13  var levelOrderBottom = function(root) {
14      const res = [];
15      if (!root) return res;
16
17      // 使用队列模拟层序遍历
18      let q = [root];
19
20      while (q.length > 0) {
21          const tmpQueue = [];
22          const valueArr = [];
23
24          for (const node of q) {
25              valueArr.push(node.val);
26
27              // 加入左节点
28              if (node.left) tmpQueue.push(node.left);
29              // 加入右节点
30              if (node.right) tmpQueue.push(node.right);
31          }
32
33          res.push(valueArr);
34          q = tmpQueue;
35      }
36
37      // 从上到下遍历，反转就是从下到上
38      return res.reverse();
39  };
```

Go

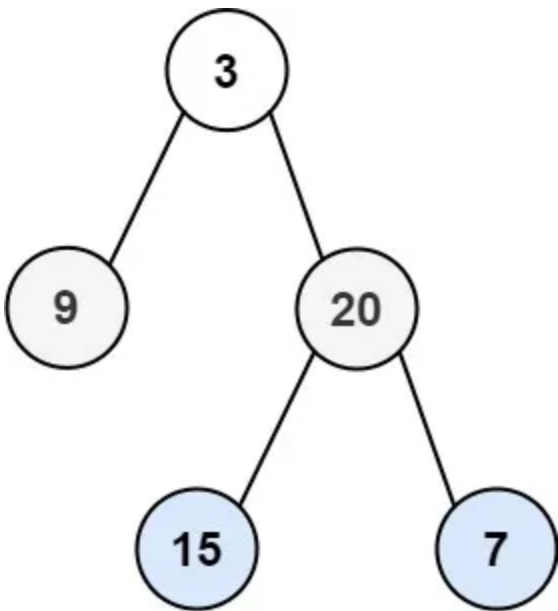
```
1  /**
2   * Definition for a binary tree node.
3   * type TreeNode struct {
4   *     Val int
5   *     Left *TreeNode
6   *     Right *TreeNode
7   * }
8   */
9  func levelOrderBottom(root *TreeNode) [][]int {
10     var res [][]int
11     if root == nil {
12         return res
13     }
14
15     // 使用队列模拟层序遍历
16     q := []*TreeNode{root}
17
18     for len(q) > 0 {
19         var tmpQueue []*TreeNode
20         var valueArr []int
21
22         for _, node := range q {
23             valueArr = append(valueArr, node.Val)
24
25             // 加入左节点
26             if node.Left != nil {
27                 tmpQueue = append(tmpQueue, node.Left)
28             }
29             // 加入右节点
30             if node.Right != nil {
31                 tmpQueue = append(tmpQueue, node.Right)
32             }
33         }
34
35         res = append(res, valueArr)
36         q = tmpQueue
37     }
38
39     // 从上到下遍历，反转就是从下到上
40     for i, j := 0, len(res)-1; i < j; i, j = i+1, j-1 {
41         res[i], res[j] = res[j], res[i]
42     }
43
44     return res
45 }
```



# 题目描述

给你 [二叉树](#) 的根节点 `root`，返回其节点值 **自底向上的层序遍历**。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

## 示例1



▼ Plain Text

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

## 示例2

▼ Plain Text

```
1  输入: root = [1]
2  输出: [[1]]
```

## 示例3

▼ Plain Text

```
1  输入: root = []
2  输出: []
```

## 提示

提示：

- 树中节点数目在范围 `[0, 2000]` 内
- `-1000 <= Node.val <= 1000`

## 题解

[力扣原题链接](#)

思路： 队列数据结构运用

1. 层序遍历采用 队列 进行模拟。
2. 从下到上层序遍历 和 从上到下遍历 的结果差异就是 结果顺序相反。可以使用我们熟悉的从上到下 进行层序遍历，最后将结果反转即可。

**C++**

```
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), right(right) {}
10  * };
11  */
12  class Solution {
13  public:
14      vector<vector<int>> levelOrderBottom(TreeNode* root) {
15          vector<vector<int>> res;
16          if (root == nullptr) {
17              return res;
18          }
19          // 使用队列模拟层序遍历
20          queue<TreeNode*> q;
21          q.push(root);
22          while (!q.empty()) {
23              queue<TreeNode*> tmpQueue;
24              // 存储每一层遍历的值
25              vector<int> valueArr;
26              while (!q.empty()) {
27                  TreeNode* current = q.front();
28                  q.pop();
29                  valueArr.push_back(current->val);
30                  // 加入左节点
31                  if (current->left != nullptr) {
32                      tmpQueue.push(current->left);
33                  }
34                  // 加入右节点
35                  if (current->right != nullptr) {
36                      tmpQueue.push(current->right);
37                  }
38              }
39
40              res.push_back(valueArr);
41              q = tmpQueue;
42          }
43          // 从上到下遍历 反转就是 从下到上
44          reverse(res.begin(), res.end());
```

```
45         return res;
46     }
47 };
```

## JAVA

▼ Plain Text

```
1  class Solution {
2      public List<List<Integer>> levelOrderBottom(TreeNode root) {
3          List<List<Integer>> res = new ArrayList<>();
4          if (root == null) {
5              return res;
6          }
7
8          // 使用队列模拟层序遍历
9          Queue<TreeNode> q = new LinkedList<>();
10         q.offer(root);
11
12         while (!q.isEmpty()) {
13             Queue<TreeNode> tmpQueue = new LinkedList<>();
14             List<Integer> valueArr = new ArrayList<>();
15
16             while (!q.isEmpty()) {
17                 TreeNode current = q.poll();
18                 valueArr.add(current.val);
19
20                 // 加入左节点
21                 if (current.left != null) {
22                     tmpQueue.offer(current.left);
23                 }
24                 // 加入右节点
25                 if (current.right != null) {
26                     tmpQueue.offer(current.right);
27                 }
28             }
29
30             res.add(valueArr);
31             q = tmpQueue;
32         }
33
34         // 从上到下遍历, 反转就是从下到上
35         Collections.reverse(res);
36         return res;
37     }
38 }
```

## Python

```
▼ Plain Text |
1  from collections import deque
2
3  class Solution:
4      def levelOrderBottom(self, root):
5          res = []
6          if not root:
7              return res
8
9          # 使用队列模拟层序遍历
10         q = deque([root])
11
12         while q:
13             tmp_queue = deque()
14             value_arr = []
15
16             while q:
17                 current = q.popleft()
18                 value_arr.append(current.val)
19
20                 # 加入左节点
21                 if current.left:
22                     tmp_queue.append(current.left)
23                 # 加入右节点
24                 if current.right:
25                     tmp_queue.append(current.right)
26
27             res.append(value_arr)
28             q = tmp_queue
29
30         # 从上到下遍历, 反转就是从下到上
31         return res[::-1]
```

## JavaScript

```
1  /**
2   * Definition for a binary tree node.
3   * function TreeNode(val, left, right) {
4   *     this.val = (val===undefined ? 0 : val)
5   *     this.left = (left===undefined ? null : left)
6   *     this.right = (right===undefined ? null : right)
7   * }
8   */
9  /**
10   * @param {TreeNode} root
11   * @return {number[][]}
12   */
13  var levelOrderBottom = function(root) {
14      const res = [];
15      if (!root) return res;
16
17      // 使用队列模拟层序遍历
18      let q = [root];
19
20      while (q.length > 0) {
21          const tmpQueue = [];
22          const valueArr = [];
23
24          for (const node of q) {
25              valueArr.push(node.val);
26
27              // 加入左节点
28              if (node.left) tmpQueue.push(node.left);
29              // 加入右节点
30              if (node.right) tmpQueue.push(node.right);
31          }
32
33          res.push(valueArr);
34          q = tmpQueue;
35      }
36
37      // 从上到下遍历，反转就是从下到上
38      return res.reverse();
39  };
```

Go

```
1  /**
2   * Definition for a binary tree node.
3   * type TreeNode struct {
4   *     Val int
5   *     Left *TreeNode
6   *     Right *TreeNode
7   * }
8   */
9  func levelOrderBottom(root *TreeNode) [][]int {
10     var res [][]int
11     if root == nil {
12         return res
13     }
14
15     // 使用队列模拟层序遍历
16     q := []*TreeNode{root}
17
18     for len(q) > 0 {
19         var tmpQueue []*TreeNode
20         var valueArr []int
21
22         for _, node := range q {
23             valueArr = append(valueArr, node.Val)
24
25             // 加入左节点
26             if node.Left != nil {
27                 tmpQueue = append(tmpQueue, node.Left)
28             }
29             // 加入右节点
30             if node.Right != nil {
31                 tmpQueue = append(tmpQueue, node.Right)
32             }
33         }
34
35         res = append(res, valueArr)
36         q = tmpQueue
37     }
38
39     // 从上到下遍历, 反转就是从下到上
40     for i, j := 0, len(res)-1; i < j; i, j = i+1, j-1 {
41         res[i], res[j] = res[j], res[i]
42     }
43
44     return res
45 }
```

---

来自: [华为OD面试手撕真题 - 二叉树的层序遍历 II \\_od手撕考二叉树多不多-CSDN博客](#)

来自: [华为OD面试手撕真题 - 二叉树的层序遍历 II \\_od手撕考二叉树多不多-CSDN博客](#)



# 华为OD 面试手撕真题 - 连接棒材的最低费用-CSDN博客

## 题目描述

你有一些长度为正整数的棍子。这些长度以数组 `sticks` 的形式给出，`sticks[i]` 是第*i*个木棍的长度。你可以通过 `x + y` 的成本将任意两个长度为 `x` 和 `y` 的棍子连接成一个棍子，你需要连接所有的棍子，直到剩下一个棍子。返回以这种方式将所有给定的棍子连接成一个棍子的 最小成本。

## 示例1

1 输入: `sticks = [2,4,3]`  
2 输出: 14  
3 解释: 从 `sticks = [2,4,3]` 开始。  
4 1、连接2和3，费用为2+3=5。现在 `sticks = [5,4]`  
5 2、连接5和4，费用为5+4=9。现在 `sticks = [9]`

## 示例2

1 输入: `sticks = [1,8,3,5]`  
2 输出: 30  
3 解释: 从 `sticks = [1,8,3,5]` 开始。1.连接1和3，费用为1+3=4。现在 `sticks = [4,8,5]` 2.连接4和5，费用为4+5=9。现在 `sticks = [9,8]` 3.连接9和8，费用为9+8=17。现在 `sticks = [17]`  
4 [17]所有木棍已经连成一根，总费用4+9+17=30

## 提示

- $1 \leq \text{sticks.length} \leq 10^4$
- $1 \leq \text{sticks}[i] \leq 10^4$

## 题解

思路: 贪心，分析

1. 贪心逻辑:  $n$ 根棍子要合并成1根棍子，需要进行的合并次数是  $n-1$ ，要想总花费更少，优先让短的两个相连。总次数不会变，所以让短的更多参与连接，总花费会变小。
2. 理解1的贪心逻辑之后，接下来我们可以使用 优先队列(小顶堆，值小的位于顶部) 来模拟连接过程，

每次从队列中取出顶部两个元素(top1 + top2)相连, 成本+= top1 + top2 , 拼接之后的木棍重新放入优先队列中。

3. 重复2的操作, 直到优先队列中元素长度为1, 此时得到的总成本就是最少的。

## C++

```
1  class Solution {
2  public:
3      int countSmaller(vector<int>& nums) {
4          if (nums.empty()) {
5              return 0;
6          }
7          // 定义小顶堆 小的值在栈顶
8          priority_queue<int, vector<int>, greater<>> pq;
9          for (int i = 0; i < nums.size(); i++) {
10             pq.push(nums[i]);
11         }
12         int res = 0;
13
14         // 每次取出两个最短木棍进行连接, 贪心
15         while (pq.size() != 1) {
16             int first = pq.top();
17             pq.pop();
18             int second = pq.top();
19             pq.pop();
20             // 成本
21             res += (first + second);
22             // 新生成的木棍重新加入木棍
23             pq.push(first + second);
24         }
25         return res;
26     }
27 };
```

## JAVA

```
1  import java.util.*;
2
3  public class Solution {
4      public int countSmaller(int[] nums) {
5          if (nums.length == 0) return 0;
6
7          // 定义小顶堆, 小的值在堆顶
8          PriorityQueue<Integer> pq = new PriorityQueue<>();
9          for (int num : nums) {
10             pq.offer(num);
11         }
12
13         int res = 0;
14
15         // 每次取出两个最短木棍进行连接, 贪心
16         while (pq.size() > 1) {
17             int first = pq.poll();
18             int second = pq.poll();
19             // 成本
20             res += first + second;
21             // 新生成的木棍重新加入
22             pq.offer(first + second);
23         }
24
25         return res;
26     }
27 }
```

## Python

```
1  import heapq
2
3  class Solution:
4      def countSmaller(self, nums):
5          if not nums:
6              return 0
7
8          # 定义小顶堆, 小的值在堆顶
9          heapq.heapify(nums)
10
11         res = 0
12
13         # 每次取出两个最短木棍进行连接, 贪心
14         while len(nums) > 1:
15             first = heapq.heappop(nums)
16             second = heapq.heappop(nums)
17             # 成本
18             res += first + second
19             # 新生成的木棍重新加入
20             heapq.heappush(nums, first + second)
21
22         return res
```

## JavaScript

```
1  class MinHeap {
2      constructor() {
3          this.data = [];
4      }
5
6      push(val) {
7          this.data.push(val);
8          this._bubbleUp();
9      }
10
11     pop() {
12         if (this.data.length === 1) return this.data.pop();
13         const min = this.data[0];
14         this.data[0] = this.data.pop();
15         this._bubbleDown();
16         return min;
17     }
18
19     _bubbleUp() {
20         let i = this.data.length - 1;
21         while (i > 0) {
22             const p = Math.floor((i - 1) / 2);
23             if (this.data[p] <= this.data[i]) break;
24             [this.data[p], this.data[i]] = [this.data[i], this.data[p]];
25             i = p;
26         }
27     }
28
29     _bubbleDown() {
30         let i = 0;
31         const n = this.data.length;
32         while (true) {
33             let l = 2 * i + 1, r = 2 * i + 2, smallest = i;
34             if (l < n && this.data[l] < this.data[smallest]) smallest = l;
35             if (r < n && this.data[r] < this.data[smallest]) smallest = r;
36             if (smallest === i) break;
37             [this.data[i], this.data[smallest]] = [this.data[smallest], th
is.data[i]];
38             i = smallest;
39         }
40     }
41
42     size() {
43         return this.data.length;
44     }
45 }
```

```
45 }
46
47 function countSmaller(nums) {
48     if (nums.length === 0) return 0;
49
50     // 定义小顶堆, 小的值在堆顶
51     const pq = new MinHeap();
52     for (const num of nums) {
53         pq.push(num);
54     }
55
56     let res = 0;
57
58     // 每次取出两个最短木棍进行连接, 贪心
59     while (pq.size() > 1) {
60         const first = pq.pop();
61         const second = pq.pop();
62         // 成本
63         res += first + second;
64         // 新生成的木棍重新加入
65         pq.push(first + second);
66     }
67
68     return res;
69 }
```

Go

```

1  package main
2
3  import (
4      "container/heap"
5      "fmt"
6  )
7
8  // 定义小顶堆结构
9  type MinHeap []int
10
11 func (h MinHeap) Len() int           { return len(h) }
12 func (h MinHeap) Less(i, j int) bool { return h[i] < h[j] } // 小顶堆
13 func (h MinHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }
14 func (h *MinHeap) Push(x any)        { *h = append(*h, x.(int)) }
15 func (h *MinHeap) Pop() any {
16     old := *h
17     n := len(old)
18     val := old[n-1]
19     *h = old[:n-1]
20     return val
21 }
22
23 func countSmaller(nums []int) int {
24     if len(nums) == 0 {
25         return 0
26     }
27
28     // 定义小顶堆, 小的值在堆顶
29     h := &MinHeap{}
30     for _, num := range nums {
31         heap.Push(h, num)
32     }
33     heap.Init(h)
34
35     res := 0
36
37     // 每次取出两个最短木棍进行连接, 贪心
38     for h.Len() > 1 {
39         first := heap.Pop(h).(int)
40         second := heap.Pop(h).(int)
41         // 成本
42         res += first + second
43         // 新生成的木棍重新加入
44         heap.Push(h, first+second)
45     }

```

```
46     return res
47 }
48
49 func main() {
50     nums := []int{1, 3, 5, 2}
51     fmt.Println(countSmaller(nums)) // 输出总成本
52 }
```

来自: [华为OD 面试手撕真题 - 连接棒材的最低费用-CSDN博客](#)



# 华为OD面试手撕真题 - 统计各个数字都不同的数字个数 (C++ & Python & JAVA & JS & GO)- CSDN博客

## 题目描述

给你一个整数  $n$ ，统计并返回各位数字都不同的数字  $x$  的个数，其中  $0 \leq x < 10^n$ 。

## 示例1

	Plain Text
1	输入: $n = 2$
2	输出: 91
3	解释: 答案应为除去 11、22、33、44、55、66、77、88、99 外，在 $0 \leq x < 100$ 范围内的所有数字。

## 示例2

	Plain Text
1	输入: $n = 0$
2	输出: 1

## 提示

- $0 \leq n \leq 8$

## 题解

[力扣原题链接](#)

思路：数学规律

组合问题，对于 $n$ 位的数，存在以下规律，第一位可以去  $1-9$  的数，9种可能，第二位可以取  $0-9$  减去第一位选择的数，存在9种可能，第二位可以取  $0-9$  减去第一、第二位选择数字，存在8种可能，规律类似。

根据上面这个规律，可以计算  $(1 - n - 1)$  位数的数量就是结果。

C++

```
1  class Solution {
2  public:
3      int countNumbersWithUniqueDigits(int n) {
4          if (n == 0) {
5              return 1;
6          }
7          if (n == 1) {
8              return 10;
9          }
10         int ans = 10;
11
12         int number = 9;
13         for (int i = 0; i < n - 1; i++) {
14             // 最后位可选数量等于 上一个最后数可选数 * -1
15             number *= 9 - i;
16             ans += number;
17         }
18         return ans;
19     }
20 };
```

## JAVA

```
1  class Solution {
2      public int countNumbersWithUniqueDigits(int n) {
3          if (n == 0) {
4              return 1;
5          }
6          if (n == 1) {
7              return 10;
8          }
9          int ans = 10;
10
11         int number = 9;
12         for (int i = 0; i < n - 1; i++) {
13             // 最后位可选数量等于上一个最后数可选数 * (9 - i)
14             number *= 9 - i;
15             ans += number;
16         }
17         return ans;
18     }
19 }
```

## Python

```
1  class Solution:
2      def countNumbersWithUniqueDigits(self, n: int) -> int:
3          if n == 0:
4              return 1
5          if n == 1:
6              return 10
7          ans = 10
8          number = 9
9          for i in range(n - 1):
10             # 最后位可选数量等于上一个最后数可选数 * (9 - i)
11             number *= 9 - i
12             ans += number
13         return ans
```

## JavaScript

```
1 function countNumbersWithUniqueDigits(n) {
2     if (n === 0) {
3         return 1;
4     }
5     if (n === 1) {
6         return 10;
7     }
8     let ans = 10;
9     let number = 9;
10    for (let i = 0; i < n - 1; i++) {
11        // 最后位可选数量等于上一个最后数可选数 * (9 - i)
12        number *= (9 - i);
13        ans += number;
14    }
15    return ans;
16 }
```

## Go

```
1 func countNumbersWithUniqueDigits(n int) int {
2     if n == 0 {
3         return 1
4     }
5     if n == 1 {
6         return 10
7     }
8     ans := 10
9     number := 9
10    for i := 0; i < n-1; i++ {
11        // 最后位可选数量等于上一个最后数可选数 * (9 - i)
12        number *= 9 - i
13        ans += number
14    }
15    return ans
16 }
```

# 2025华为OD面试手撕真题 - 验证回文串 II - CSDN博客

## 题目描述

给你一个字符串 `s`，最多 可以从中删除一个字符。

请你判断 `s` 是否能成为回文字符串：如果能，返回 `true`；否则，返回 `false`。

## 示例1

	Plain Text
1	输入: <code>s = "aba"</code>
2	输出: <code>true</code>

## 示例2

	Plain Text
1	输入: <code>s = "abca"</code>
2	输出: <code>true</code>
3	解释: 你可以删除字符 <code>'c'</code> 。

## 示例3

	Plain Text
1	输入: <code>s = "abc"</code>
2	输出: <code>false</code>

## 提示

- `1 <= s.length <= 105`
- `s` 由小写英文字母组成

## 题解

[力扣原题链接](#)

思路: 双指针

1. 回文串简单来说就是 呈中心对称 的字符串。
2. 如果要判断一个字符串是否是回文串，使用 双指针算法 是一个比较常见手段。这个题也可以借助一个思路，使用双指针从两端判断是否呈中心对称，如果指针移动过程中发现 `s[left] != s[right]` 时，题目描述可以删除一个字符，这时候可以采取两种处理方式：
  - 移除左指针对应字符，`left++`。判断剩余是否成中心对称
  - 移除右指针对应字符，`right--`。判断剩余是否成中心对称
3. 2就是处理这个问题的基本思路。这种解法的时间复杂度为 `O(n)`，空间复杂度 `O(1)`

## C++

```
1  class Solution {
2  public:
3      // 判断是否满足回文串
4      bool valid(string &s, int left, int right) {
5          while (left < right) {
6              if (s[left] != s[right]) {
7                  return false;
8              }
9              left++;
10             right--;
11         }
12         return true;
13     }
14
15     bool validPalindrome(string s) {
16         int n = s.size();
17         // 本身就是回文串
18         if (n == 1) {
19             return true;
20         }
21         int left = 0, right = n - 1;
22         while (left < right) {
23             if (s[left] != s[right]) {
24                 // 尝试移除左边字符 和 右边字符判断是否满足回文串
25                 return valid(s, left + 1, right) || valid(s, left, right -
26             1);
27             }
28             left++;
29             right--;
30         }
31         return true;
32     };
33 }
```

## JAVA

Plain Text

```
1  class Solution {
2      // 判断是否满足回文串
3      private boolean valid(String s, int left, int right) {
4          while (left < right) {
5              if (s.charAt(left) != s.charAt(right)) {
6                  return false;
7              }
8              left++;
9              right--;
10         }
11         return true;
12     }
13
14     public boolean validPalindrome(String s) {
15         int n = s.length();
16         // 本身就是回文串
17         if (n == 1) {
18             return true;
19         }
20         int left = 0, right = n - 1;
21         while (left < right) {
22             if (s.charAt(left) != s.charAt(right)) {
23                 // 尝试移除左边字符 和 右边字符判断是否满足回文串
24                 return valid(s, left + 1, right) || valid(s, left, right
25 - 1);
26             }
27             left++;
28             right--;
29         }
30         return true;
31     }
32 }
```

## Python

```
1 class Solution:
2     # 判断是否满足回文串
3     def valid(self, s: str, left: int, right: int) -> bool:
4         while left < right:
5             if s[left] != s[right]:
6                 return False
7             left += 1
8             right -= 1
9         return True
10
11     def validPalindrome(self, s: str) -> bool:
12         n = len(s)
13         # 本身就是回文串
14         if n == 1:
15             return True
16         left, right = 0, n - 1
17         while left < right:
18             if s[left] != s[right]:
19                 # 尝试移除左边字符 和 右边字符判断是否满足回文串
20                 return self.valid(s, left + 1, right) or self.valid(s, left, right - 1)
21             left += 1
22             right -= 1
23         return True
```

## JavaScript



```
1  function valid(s, left, right) {
2      // 判断是否满足回文串
3      while (left < right) {
4          if (s[left] !== s[right]) {
5              return false;
6          }
7          left++;
8          right--;
9      }
10     return true;
11 }
12
13 function validPalindrome(s) {
14     const n = s.length;
15     // 本身就是回文串
16     if (n === 1) return true;
17     let left = 0, right = n - 1;
18     while (left < right) {
19         if (s[left] !== s[right]) {
20             // 尝试移除左边字符 和 右边字符判断是否满足回文串
21             return valid(s, left + 1, right) || valid(s, left, right - 1);
22         }
23         left++;
24         right--;
25     }
26     return true;
27 }
```

Go

```
1 // 判断是否满足回文串
2 func valid(s string, left, right int) bool {
3     for left < right {
4         if s[left] != s[right] {
5             return false
6         }
7         left++
8         right--
9     }
10    return true
11 }
12
13 func validPalindrome(s string) bool {
14     n := len(s)
15     // 本身就是回文串
16     if n == 1 {
17         return true
18     }
19     left, right := 0, n-1
20     for left < right {
21         if s[left] != s[right] {
22             // 尝试移除左边字符 和 右边字符判断是否满足回文串
23             return valid(s, left+1, right) || valid(s, left, right-1)
24         }
25         left++
26         right--
27     }
28     return true
29 }
```

## 题目描述

给你一个字符串 `s`，最多 可以从中删除一个字符。

请你判断 `s` 是否能成为回文字符串：如果能，返回 `true`；否则，返回 `false`。

## 示例1

```
1 输入: s = "aba"
2 输出: true
```

## 示例2

```
1  输入: s = "abca"
2  输出: true
3  解释: 你可以删除字符 'c' 。
```

## 示例3

```
1  输入: s = "abc"
2  输出: false
```

## 提示

- `1 <= s.length <= 105`
- `s` 由小写英文字母组成

## 题解

[力扣原题链接](#)

思路: 双指针

1. 回文串简单来说就是 呈中心对称 的字符串。
2. 如果要判断一个字符串是否是回文串, 使用 双指针算法 是一个比较常见手段。这个题也可以借助一个思路, 使用双指针从两端判断是否呈中心对称, 如果指针移动过程中发现 `s[left] != s[right]` 时, 题目描述可以删除一个字符, 这时候可以采取两种处理方式:
  - 移除左指针对应字符, `left++`。判断剩余是否成中心对称
  - 移除右指针对应字符, `right--`。判断剩余是否成中心对称
3. 2就是处理这个问题的基本思路。这种解法的时间复杂度为 `O(n)`, 空间复杂度 `O(1)`

C++

```
1  class Solution {
2  public:
3      // 判断是否满足回文串
4      bool valid(string &s, int left, int right) {
5          while (left < right) {
6              if (s[left] != s[right]) {
7                  return false;
8              }
9              left++;
10             right--;
11         }
12         return true;
13     }
14
15     bool validPalindrome(string s) {
16         int n = s.size();
17         // 本身就是回文串
18         if (n == 1) {
19             return true;
20         }
21         int left = 0, right = n - 1;
22         while (left < right) {
23             if (s[left] != s[right]) {
24                 // 尝试移除左边字符 和 右边字符判断是否满足回文串
25                 return valid(s, left + 1, right) || valid(s, left, right -
26                     1);
27             }
28             left++;
29             right--;
30         }
31         return true;
32     };
33 }
```

## JAVA

```
1  class Solution {
2      // 判断是否满足回文串
3      private boolean valid(String s, int left, int right) {
4          while (left < right) {
5              if (s.charAt(left) != s.charAt(right)) {
6                  return false;
7              }
8              left++;
9              right--;
10         }
11         return true;
12     }
13
14     public boolean validPalindrome(String s) {
15         int n = s.length();
16         // 本身就是回文串
17         if (n == 1) {
18             return true;
19         }
20         int left = 0, right = n - 1;
21         while (left < right) {
22             if (s.charAt(left) != s.charAt(right)) {
23                 // 尝试移除左边字符 和 右边字符判断是否满足回文串
24                 return valid(s, left + 1, right) || valid(s, left, right
25                     - 1);
26             }
27             left++;
28             right--;
29         }
30         return true;
31     }
```

## Python

```
1 class Solution:
2     # 判断是否满足回文串
3     def valid(self, s: str, left: int, right: int) -> bool:
4         while left < right:
5             if s[left] != s[right]:
6                 return False
7             left += 1
8             right -= 1
9         return True
10
11     def validPalindrome(self, s: str) -> bool:
12         n = len(s)
13         # 本身就是回文串
14         if n == 1:
15             return True
16         left, right = 0, n - 1
17         while left < right:
18             if s[left] != s[right]:
19                 # 尝试移除左边字符 和 右边字符判断是否满足回文串
20                 return self.valid(s, left + 1, right) or self.valid(s, left, right - 1)
21             left += 1
22             right -= 1
23         return True
```

## JavaScript

```
1  function valid(s, left, right) {
2      // 判断是否满足回文串
3      while (left < right) {
4          if (s[left] !== s[right]) {
5              return false;
6          }
7          left++;
8          right--;
9      }
10     return true;
11 }
12
13 function validPalindrome(s) {
14     const n = s.length;
15     // 本身就是回文串
16     if (n === 1) return true;
17     let left = 0, right = n - 1;
18     while (left < right) {
19         if (s[left] !== s[right]) {
20             // 尝试移除左边字符 和 右边字符判断是否满足回文串
21             return valid(s, left + 1, right) || valid(s, left, right - 1);
22         }
23         left++;
24         right--;
25     }
26     return true;
27 }
```

Go

```
1 // 判断是否满足回文串
2 func valid(s string, left, right int) bool {
3     for left < right {
4         if s[left] != s[right] {
5             return false
6         }
7         left++
8         right--
9     }
10    return true
11 }
12
13 func validPalindrome(s string) bool {
14     n := len(s)
15     // 本身就是回文串
16     if n == 1 {
17         return true
18     }
19     left, right := 0, n-1
20     for left < right {
21         if s[left] != s[right] {
22             // 尝试移除左边字符 和 右边字符判断是否满足回文串
23             return valid(s, left+1, right) || valid(s, left, right-1)
24         }
25         left++
26         right--
27     }
28     return true
29 }
```

来自: [2025华为OD面试手撕真题 - 验证回文串 II -CSDN博客](#)

来自: [2025华为OD面试手撕真题 - 验证回文串 II -CSDN博客](#)



# 华为OD面试手撕真题 - 拆分字符串使唯一子字符串的数目最大-CSDN博客

## 题目描述

给你一个字符串 `s`，请你拆分该字符串，并返回拆分后唯一子字符串的最大数目。

字符串 `s` 拆分后可以得到若干 非空子字符串，这些子字符串连接后应当能够还原为原字符串。但是拆分出来的每个子字符串都必须是 唯一的。

注意：子字符串 是字符串中的一个连续字符序列。

## 示例1

	Plain Text
1	输入：s = "ababccc"
2	输出：5
3	解释：一种最大拆分方法为 ['a', 'b', 'ab', 'c', 'cc']。像 ['a', 'b', 'a', 'b', 'c', 'cc'] 这样拆分不满足题目要求，因为其中的 'a' 和 'b' 都出现了不止一次。

## 示例2

	Plain Text
1	输入：s = "aba"
2	输出：2
3	解释：一种最大拆分方法为 ['a', 'ba']。

## 示例3

	Plain Text
1	输入：s = "aa"
2	输出：1
3	解释：无法进一步拆分字符串。

## 提示

- `1 <= s.length <= 16`
- `s` 仅包含小写英文字母

# 题解

[力扣原题链接](#)

思路： 递归回溯

1. 简单的 递归回溯 算法应用，整体思路就是枚举所有拆分方案，记录其中合法拆分方案能够拆分最多子串数量。
2. 对于不重复字符串可以引入 集合 数据结构进行判断,合法拆分最终能够拆分子串数量其实就是集合的长度。一个小经验，对于递归算法，一般你都需要考虑是否能够进行 剪枝操作 。这能提高你的代码执行效率，执行非必要代码执行。
3. 下面这个递归回溯的时间复杂度为  $O(2^n * n)$ ，空间复杂度为  $O(n)$

C++

```
1  class Solution {
2  public:
3      int res = 0;
4      // 当前开始下标
5      void dfs(int index, string& s, unordered_set<string>& uniquSub) {
6          int n = s.size();
7          // 剪枝 不可能存在更多的数量
8          if (n - index + uniquSub.size() < res) {
9              return;
10         }
11         // 形成一个合法拆分方案
12         if (index >= n) {
13             res = max(res, (int)uniquSub.size());
14             return;
15         }
16         for (int i = index; i < n; i++) {
17             string subString = s.substr(index, i - index + 1);
18             // 说明重复,直接跳过
19             if (uniquSub.find(subString) != uniquSub.end()) {
20                 continue;
21             }
22             // 递归回溯
23             uniquSub.insert(subString);
24             dfs(i+1, s, uniquSub);
25             uniquSub.erase(subString);
26         }
27     }
28
29     int maxUniqueSplit(string s) {
30         unordered_set<string> uniquSub;
31         dfs(0, s, uniquSub);
32         return res;
33     }
34 };
```

## JAVA

```
1  class Solution {
2      int res = 0;
3
4      public int maxUniqueSplit(String s) {
5          Set<String> uniqueSub = new HashSet<>();
6          dfs(0, s, uniqueSub);
7          return res;
8      }
9
10     // 回溯函数, index 表示当前起始位置
11     void dfs(int index, String s, Set<String> uniqueSub) {
12         int n = s.length();
13         // 剪枝: 如果剩余字符加上当前集合大小仍不超过 res, 就不需要继续递归
14         if (n - index + uniqueSub.size() < res) return;
15
16         // 到达末尾, 形成一个合法拆分方案
17         if (index == n) {
18             res = Math.max(res, uniqueSub.size());
19             return;
20         }
21
22         for (int i = index; i < n; i++) {
23             String sub = s.substring(index, i + 1);
24             if (uniqueSub.contains(sub)) continue;
25
26             // 尝试添加并回溯
27             uniqueSub.add(sub);
28             dfs(i + 1, s, uniqueSub);
29             uniqueSub.remove(sub); // 回溯删除
30         }
31     }
32 }
```

## Python

```
1 class Solution:
2     def maxUniqueSplit(self, s: str) -> int:
3         self.res = 0
4
5         def dfs(index: int, unique_sub: set[str]):
6             n = len(s)
7             # 剪枝: 如果剩余长度加当前集合大小小于 res, 无需继续
8             if n - index + len(unique_sub) < self.res:
9                 return
10
11             # 到达末尾, 更新结果
12             if index == n:
13                 self.res = max(self.res, len(unique_sub))
14                 return
15
16             for i in range(index, n):
17                 sub = s[index:i + 1]
18                 if sub in unique_sub:
19                     continue
20                 unique_sub.add(sub)
21                 dfs(i + 1, unique_sub)
22                 unique_sub.remove(sub) # 回溯
23
24         dfs(0, set())
25         return self.res
```

## JavaScript

```
1  /**
2   * @param {string} s
3   * @return {number}
4   */
5  var maxUniqueSplit = function(s) {
6      let res = 0;
7
8      function dfs(index, uniqueSub) {
9          const n = s.length;
10         // 剪枝优化
11         if (n - index + uniqueSub.size < res) return;
12
13         // 到达末尾
14         if (index === n) {
15             res = Math.max(res, uniqueSub.size);
16             return;
17         }
18
19         for (let i = index; i < n; i++) {
20             const sub = s.substring(index, i + 1);
21             if (uniqueSub.has(sub)) continue;
22
23             // 添加并回溯
24             uniqueSub.add(sub);
25             dfs(i + 1, uniqueSub);
26             uniqueSub.delete(sub); // 回溯
27         }
28     }
29
30     dfs(0, new Set());
31     return res;
32 };
```

Go

```
1  var maxUniqueSplit = func(s string) int {
2      res := 0
3      n := len(s)
4
5      // 定义 DFS 递归函数
6      var dfs func(index int, uniqueSet map[string]bool)
7      dfs = func(index int, uniqueSet map[string]bool) {
8          // 剪枝: 剩余部分 + 当前已选的数量 < res, 不可能更优
9          if len(s)-index+len(uniqueSet) < res {
10             return
11         }
12
13         // 成功拆分整个字符串
14         if index >= n {
15             if len(uniqueSet) > res {
16                 res = len(uniqueSet)
17             }
18             return
19         }
20
21         // 枚举所有可能子串
22         for i := index + 1; i <= n; i++ {
23             subStr := s[index:i]
24             if uniqueSet[subStr] {
25                 continue // 重复子串跳过
26             }
27             uniqueSet[subStr] = true
28             dfs(i, uniqueSet)
29             delete(uniqueSet, subStr) // 回溯
30         }
31     }
32
33     dfs(0, make(map[string]bool))
34     return res
35 }
```

## 题目描述

给你一个字符串 `s`，请你拆分该字符串，并返回拆分后唯一子字符串的最大数目。

字符串 `s` 拆分后可以得到若干 **非空子字符串**，这些子字符串连接后应当能够还原为原字符串。但是拆分出来的每个子字符串都必须是 **唯一的**。

注意：子字符串 是字符串中的一个连续字符序列。

## 示例1

	Plain Text
1	输入: <code>s = "ababccc"</code>
2	输出: 5
3	解释: 一种最大拆分方法为 <code>['a', 'b', 'ab', 'c', 'cc']</code> 。像 <code>['a', 'b', 'a', 'b', 'c', 'cc']</code> 这样拆分不满足题目要求, 因为其中的 <code>'a'</code> 和 <code>'b'</code> 都出现了不止一次。

## 示例2

	Plain Text
1	输入: <code>s = "aba"</code>
2	输出: 2
3	解释: 一种最大拆分方法为 <code>['a', 'ba']</code> 。

## 示例3

	Plain Text
1	输入: <code>s = "aa"</code>
2	输出: 1
3	解释: 无法进一步拆分字符串。

## 提示

- `1 <= s.length <= 16`
- `s` 仅包含小写英文字母

## 题解

[力扣原题链接](#)

思路: 递归回溯

1. 简单的 递归回溯 算法应用, 整体思路就是枚举所有拆分方案, 记录其中合法拆分方案能够拆分最多子串数量。
2. 对于不重复字符串可以引入 集合 数据结构进行判断, 合法拆分最终能够拆分子串数量其实就是集合的长度。一个小经验, 对于递归算法, 一般你都需要考虑是否能够进行 剪枝操作。这能提高你的代码执行效率, 执行非必要代码执行。
3. 下面这个递归回溯的时间复杂度为  $O(2^n * n)$ , 空间复杂度为  $O(n)$

C++



```
1  class Solution {
2  public:
3      int res = 0;
4      // 当前开始下标
5      void dfs(int index, string& s, unordered_set<string>& uniquSub) {
6          int n = s.size();
7          // 剪枝 不可能存在更多的数量
8          if (n - index + uniquSub.size() < res) {
9              return;
10         }
11         // 形成一个合法拆分方案
12         if (index >= n) {
13             res = max(res, (int)uniquSub.size());
14             return;
15         }
16         for (int i = index; i < n; i++) {
17             string subString = s.substr(index, i - index + 1);
18             // 说明重复,直接跳过
19             if (uniquSub.find(subString) != uniquSub.end()) {
20                 continue;
21             }
22             // 递归回溯
23             uniquSub.insert(subString);
24             dfs(i+1, s, uniquSub);
25             uniquSub.erase(subString);
26         }
27     }
28
29     int maxUniqueSplit(string s) {
30         unordered_set<string> uniquSub;
31         dfs(0, s, uniquSub);
32         return res;
33     }
34 };
```

## JAVA

```
1  class Solution {
2      int res = 0;
3
4      public int maxUniqueSplit(String s) {
5          Set<String> uniqueSub = new HashSet<>();
6          dfs(0, s, uniqueSub);
7          return res;
8      }
9
10     // 回溯函数, index 表示当前起始位置
11     void dfs(int index, String s, Set<String> uniqueSub) {
12         int n = s.length();
13         // 剪枝: 如果剩余字符加上当前集合大小仍不超过 res, 就不需要继续递归
14         if (n - index + uniqueSub.size() < res) return;
15
16         // 到达末尾, 形成一个合法拆分方案
17         if (index == n) {
18             res = Math.max(res, uniqueSub.size());
19             return;
20         }
21
22         for (int i = index; i < n; i++) {
23             String sub = s.substring(index, i + 1);
24             if (uniqueSub.contains(sub)) continue;
25
26             // 尝试添加并回溯
27             uniqueSub.add(sub);
28             dfs(i + 1, s, uniqueSub);
29             uniqueSub.remove(sub); // 回溯删除
30         }
31     }
32 }
```

## Python

```
1 class Solution:
2     def maxUniqueSplit(self, s: str) -> int:
3         self.res = 0
4
5         def dfs(index: int, unique_sub: set[str]):
6             n = len(s)
7             # 剪枝: 如果剩余长度加当前集合大小小于 res, 无需继续
8             if n - index + len(unique_sub) < self.res:
9                 return
10
11             # 到达末尾, 更新结果
12             if index == n:
13                 self.res = max(self.res, len(unique_sub))
14                 return
15
16             for i in range(index, n):
17                 sub = s[index:i + 1]
18                 if sub in unique_sub:
19                     continue
20                 unique_sub.add(sub)
21                 dfs(i + 1, unique_sub)
22                 unique_sub.remove(sub) # 回溯
23
24         dfs(0, set())
25         return self.res
```

## JavaScript

```
1  /**
2   * @param {string} s
3   * @return {number}
4   */
5  var maxUniqueSplit = function(s) {
6      let res = 0;
7
8      function dfs(index, uniqueSub) {
9          const n = s.length;
10         // 剪枝优化
11         if (n - index + uniqueSub.size < res) return;
12
13         // 到达末尾
14         if (index === n) {
15             res = Math.max(res, uniqueSub.size);
16             return;
17         }
18
19         for (let i = index; i < n; i++) {
20             const sub = s.substring(index, i + 1);
21             if (uniqueSub.has(sub)) continue;
22
23             // 添加并回溯
24             uniqueSub.add(sub);
25             dfs(i + 1, uniqueSub);
26             uniqueSub.delete(sub); // 回溯
27         }
28     }
29
30     dfs(0, new Set());
31     return res;
32 };
```

Go

```
1  var maxUniqueSplit = func(s string) int {
2      res := 0
3      n := len(s)
4
5      // 定义 DFS 递归函数
6      var dfs func(index int, uniqueSet map[string]bool)
7      dfs = func(index int, uniqueSet map[string]bool) {
8          // 剪枝: 剩余部分 + 当前已选的数量 < res, 不可能更优
9          if len(s)-index+len(uniqueSet) < res {
10             return
11         }
12
13         // 成功拆分整个字符串
14         if index >= n {
15             if len(uniqueSet) > res {
16                 res = len(uniqueSet)
17             }
18             return
19         }
20
21         // 枚举所有可能子串
22         for i := index + 1; i <= n; i++ {
23             subStr := s[index:i]
24             if uniqueSet[subStr] {
25                 continue // 重复子串跳过
26             }
27             uniqueSet[subStr] = true
28             dfs(i, uniqueSet)
29             delete(uniqueSet, subStr) // 回溯
30         }
31     }
32
33     dfs(0, make(map[string]bool))
34     return res
35 }
```

来自: [华为OD面试手撕真题 - 拆分字符串使唯一子字符串的数目最大-CSDN博客](#)

来自: [华为OD面试手撕真题 - 拆分字符串使唯一子字符串的数目最大-CSDN博客](#)

# 华为OD面试手撕真题 - 最大数-CSDN博客

## 题目描述

给定一组非负整数 `nums`，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

\*\*注意：\*\*输出结果可能非常大，所以你需要返回一个字符串而不是整数。

## 示例1

	Plain Text
1	输入: <code>nums = [10,2]</code>
2	输出: <code>"210"</code>

## 示例2

	Plain Text
1	输入: <code>nums = [3,30,34,5,9]</code>
2	输出: <code>"9534330"</code>

## 提示

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 109`

## 题解

[力扣原题链接](#)

思路：贪心

- n个数字按字符串方式进行拼接，任何顺序拼接得到字符串长度是相同。相同长度全是数字字符的字符串的字典序和数字比大小是相同的。
- 明白1的规律之后，我们将输入的数字数组转换为字符串数组。然后使用贪心算法的规律进行排序 `a` `b` 两个字符串，如果 `a + b` 的字典序大于 `b + a` 则应该 `a` 排在 `b` 的前面。
- 排序之后，按照顺序进行拼接返回即可。额外注意处理一下全是0的情况

C++

```
1  class Solution {
2  public:
3      string largestNumber(vector<int>& nums) {
4          int n = nums.size();
5          vector<string> numsStr(n);
6          for (int i = 0; i < n; i++) {
7              numsStr[i] = to_string(nums[i]);
8          }
9          // 自定义排序
10         sort(numsStr.begin(), numsStr.end(), [](const string &a, const string &b) {
11             return a + b > b + a;
12         });
13
14         if (numsStr[0] == "0") {
15             return "0";
16         }
17         string res = "";
18         for (int i = 0; i < n; i++) {
19             res += numsStr[i];
20         }
21         // 移除
22         return res;
23     }
24 };
```

## JAVA

```
1  import java.util.*;
2
3  class Solution {
4      public String largestNumber(int[] nums) {
5          int n = nums.length;
6          String[] numsStr = new String[n];
7          for (int i = 0; i < n; i++) {
8              numsStr[i] = String.valueOf(nums[i]);
9          }
10
11         // 自定义排序
12         Arrays.sort(numsStr, (a, b) -> (b + a).compareTo(a + b));
13
14         // 特殊情况：全是0
15         if (numsStr[0].equals("0")) {
16             return "0";
17         }
18
19         // 拼接结果
20         StringBuilder res = new StringBuilder();
21         for (String num : numsStr) {
22             res.append(num);
23         }
24
25         return res.toString();
26     }
27 }
```

## Python



```
1 class Solution:
2     def largestNumber(self, nums):
3         # 自定义排序函数
4         def compare(a, b):
5             if a + b > b + a:
6                 return -1
7             elif a + b < b + a:
8                 return 1
9             else:
10                return 0
11
12        numsStr = list(map(str, nums))
13        numsStr.sort(key=cmp_to_key(compare))
14
15        # 特殊情况: 全是0
16        if numsStr[0] == "0":
17            return "0"
18
19        # 拼接结果
20        return ''.join(numsStr)
```

## JavaScript

```
1  /**
2   * @param {number[]} nums
3   * @return {string}
4   */
5  var largestNumber = function(nums) {
6      // 将数字转为字符串
7      let numsStr = nums.map(String);
8
9      // 自定义排序
10     numsStr.sort((a, b) => (b + a).localeCompare(a + b));
11
12     // 特殊情况：全是0
13     if (numsStr[0] === "0") {
14         return "0";
15     }
16
17     // 拼接结果
18     return numsStr.join('');
19 };
```

Go

```
1 func largestNumber(nums []int) string {
2     n := len(nums)
3     numsStr := make([]string, n)
4     for i := 0; i < n; i++ {
5         numsStr[i] = strconv.Itoa(nums[i])
6     }
7
8     // 自定义排序
9     sort.Slice(numsStr, func(i, j int) bool {
10         return numsStr[i]+numsStr[j] > numsStr[j]+numsStr[i]
11     })
12
13     // 特殊情况：全是0
14     if numsStr[0] == "0" {
15         return "0"
16     }
17
18     // 拼接结果
19     return strings.Join(numsStr, "")
20 }
```

## 题目描述

给定一组非负整数 `nums`，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

**\*\*注意：**\*\*输出结果可能非常大，所以你需要返回一个[字符串](#)而不是整数。

### 示例1

```
1 输入: nums = [10,2]
2 输出: "210"
```

### 示例2

```
1 输入: nums = [3,30,34,5,9]
2 输出: "9534330"
```

## 提示

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 109`

## 题解

[力扣原题链接](#)

思路：贪心

1. n个数字按字符串方式进行拼接，任何顺序拼接得到字符串长度是相同。相同长度全是数字字符的字符串的字典序和数字比大小是相同的。
2. 明白1的规律之后，我们将输入的数字数组转换为字符串数组。然后使用贪心算法的规律进行排序 `a + b` 两个字符串，如果 `a + b` 的字典序大于 `b + a` 则应该 `a` 排在 `b` 的前面。
3. 排序之后，按照顺序进行拼接返回即可。额外注意处理一下全是0的情况

## C++

```
1  class Solution {
2  public:
3      string largestNumber(vector<int>& nums) {
4          int n = nums.size();
5          vector<string> numsStr(n);
6          for (int i = 0; i < n; i++) {
7              numsStr[i] = to_string(nums[i]);
8          }
9          // 自定义排序
10         sort(numsStr.begin(), numsStr.end(), [](const string &a, const string &b) {
11             return a + b > b + a;
12         });
13
14         if (numsStr[0] == "0") {
15             return "0";
16         }
17         string res = "";
18         for (int i = 0; i < n; i++) {
19             res += numsStr[i];
20         }
21         // 移除
22         return res;
23     }
24 };
```

## JAVA

```
1  import java.util.*;
2
3  class Solution {
4      public String largestNumber(int[] nums) {
5          int n = nums.length;
6          String[] numsStr = new String[n];
7          for (int i = 0; i < n; i++) {
8              numsStr[i] = String.valueOf(nums[i]);
9          }
10
11         // 自定义排序
12         Arrays.sort(numsStr, (a, b) -> (b + a).compareTo(a + b));
13
14         // 特殊情况：全是0
15         if (numsStr[0].equals("0")) {
16             return "0";
17         }
18
19         // 拼接结果
20         StringBuilder res = new StringBuilder();
21         for (String num : numsStr) {
22             res.append(num);
23         }
24
25         return res.toString();
26     }
27 }
```

## Python

```
1 class Solution:
2     def largestNumber(self, nums):
3         # 自定义排序函数
4         def compare(a, b):
5             if a + b > b + a:
6                 return -1
7             elif a + b < b + a:
8                 return 1
9             else:
10                return 0
11
12        numsStr = list(map(str, nums))
13        numsStr.sort(key=cmp_to_key(compare))
14
15        # 特殊情况: 全是0
16        if numsStr[0] == "0":
17            return "0"
18
19        # 拼接结果
20        return ''.join(numsStr)
```

## JavaScript

```
1  /**
2   * @param {number[]} nums
3   * @return {string}
4   */
5  var largestNumber = function(nums) {
6      // 将数字转为字符串
7      let numsStr = nums.map(String);
8
9      // 自定义排序
10     numsStr.sort((a, b) => (b + a).localeCompare(a + b));
11
12     // 特殊情况：全是0
13     if (numsStr[0] === "0") {
14         return "0";
15     }
16
17     // 拼接结果
18     return numsStr.join('');
19 };
```

Go

```
1 func largestNumber(nums []int) string {
2     n := len(nums)
3     numsStr := make([]string, n)
4     for i := 0; i < n; i++ {
5         numsStr[i] = strconv.Itoa(nums[i])
6     }
7
8     // 自定义排序
9     sort.Slice(numsStr, func(i, j int) bool {
10         return numsStr[i]+numsStr[j] > numsStr[j]+numsStr[i]
11     })
12
13     // 特殊情况: 全是0
14     if numsStr[0] == "0" {
15         return "0"
16     }
17
18     // 拼接结果
19     return strings.Join(numsStr, "")
20 }
```

来自: [华为OD面试手撕真题 - 最大数-CSDN博客](#)



# 华为OD面试手撕真题 - 字符串解码 (C++ & Python & JAVA & JS & GO)-CSDN博客

## 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

▼ Plain Text |

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

## 示例2

▼ Plain Text |

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

## 示例3

▼ Plain Text |

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

## 示例4

```
1  输入: s = "abc3[cd]xyz"
2  输出: "abccdcddxyz"
```

## 提示

- `1 <= s.length <= 30`
- `s` 由小写英文字母、数字和方括号 `'[]'` 组成
- `s` 保证是一个 **有效** 的输入。
- `s` 中所有整数的取值范围为 `[1, 300]`

## 题解

[力扣原题链接](#)

思路：一道经典 栈模拟

1. 可以使用两个栈 `numberStack` 和 `textStack` 分别存储数字和字符串。使用 `currentText` 和 `currentNumber` 存储遍历过程中出现的普通字符串和重复数量数字。对于不同字符串处理情况如下：
  - 对于出现 `]` 说明新一层嵌套出现，往 `textStack` 中压入一个空字符串。
  - 当 `]` 说明需要弹出一层嵌套借助两个 栈 进行解码即可。依次弹出 `numberStack` 和 `textStack` 栈顶元素，重复 `textStack` 栈顶元素 `numberStack` 栈顶元素对应次数。得到的结果拼接到 `textStack` 新栈顶元素后面。这里可能会递归弹出情况出现
  - 遇到 数字字符 代表重复倍数数字出现，此时应该将 `currentText` 拼接到 `textStack` 栈顶元素后面。数字字符其实代表接下来会出现新的嵌套层，这里需要将两层字符串进行切分。
  - 遇到 字母字符 ，拼接到 `currentText` 后面即可。
2. 按照2的逻辑进行逻辑遍历完输入字符串之后，注意收尾操作，如果 `currentText` 不为空，则压入 `textStack` 中。
3. 最后就是生成最终结果。按照从前往后拼接栈中元素即可(为了实现这个需求所以我代码才会使用数组模拟栈)。

C++

```
1  class Solution {
2      public:
3          string decodeString(string s) {
4              // 使用数组模拟栈
5              vector<int> numberStack;
6              // 使用数组模拟栈
7              vector<string> textStack;
8              // 守卫字符串防止越界
9              textStack.push_back("");
10             string currentText = "";
11             string currentNnumber = "";
12             int n = s.size();
13
14             for (int i = 0; i < n; i++) {
15                 char c = s[i];
16                 // 数字字符
17                 if ('0' <= c && '9' >= c) {
18                     if (!currentText.empty()) {
19                         textStack[textStack.size() - 1] += currentText;
20                         currentText.clear();
21                     }
22                     currentNnumber.push_back(c);
23                     // 左括号 代表新的嵌套出现
24                 } else if (c == '[') {
25                     // 新嵌套出现
26                     textStack.push_back("");
27                     // 存储数字
28                     numberStack.push_back(stoi(currentNnumber));
29                     currentNnumber.clear();
30                     // 执行重复操作
31                 } else if (c == ']') {
32                     // 发生在连续第一个]执行
33                     if (!currentText.empty()) {
34                         textStack[textStack.size() - 1] += currentText;
35                         currentText.clear();
36                     }
37
38                     string repeateStrSub = textStack.back();
39                     textStack.pop_back();
40                     // 重复字符串
41                     string repeateStr;
42                     for (int j = 1; j <= numberStack.back(); j++) {
43                         repeateStr += repeateStrSub;
44                     }
45                     numberStack.pop_back();
```

```

46         textStack[textStack.size() - 1] += repeateStr;
47
48     } else {
49         currentText.push_back(c);
50     }
51 }
52 // 收尾操作
53 if (!currentText.empty()) {
54     textStack.push_back(currentText);
55 }
56
57 // 拼接结果
58 string res = "";
59 for (int i = 0; i < textStack.size(); i++) {
60     res += textStack[i];
61 }
62 return res;
63 }
64 };

```

## JAVA

```
1  class Solution {
2      public String decodeString(String s) {
3          // 使用栈模拟嵌套
4          Stack<Integer> numberStack = new Stack<>();
5          Stack<StringBuilder> textStack = new Stack<>();
6          textStack.push(new StringBuilder());
7
8          StringBuilder currentText = new StringBuilder();
9          StringBuilder currentNumber = new StringBuilder();
10
11         for (char c : s.toCharArray()) {
12             if (Character.isDigit(c)) {
13                 // 当前是数字
14                 if (currentText.length() > 0) {
15                     textStack.peek().append(currentText);
16                     currentText.setLength(0);
17                 }
18                 currentNumber.append(c);
19             } else if (c == '[') {
20                 // 开启新嵌套
21                 numberStack.push(Integer.parseInt(currentNumber.toString
22             ());
23                 currentNumber.setLength(0);
24                 textStack.push(new StringBuilder());
25             } else if (c == ']') {
26                 // 嵌套结束, 进行重复
27                 if (currentText.length() > 0) {
28                     textStack.peek().append(currentText);
29                     currentText.setLength(0);
30                 }
31                 String repeatStr = textStack.pop().toString();
32                 int count = numberStack.pop();
33                 StringBuilder repeated = new StringBuilder();
34                 for (int i = 0; i < count; i++) {
35                     repeated.append(repeatStr);
36                 }
37                 textStack.peek().append(repeated);
38             } else {
39                 // 普通字符
40                 currentText.append(c);
41             }
42         }
43         // 收尾操作
44         if (currentText.length() > 0) {
45             textStack.push(currentText);
46         }
47     }
48 }
```

```
45         }
46         // 构建结果
47         StringBuilder result = new StringBuilder();
48         for (StringBuilder sb : textStack) {
49             result.append(sb);
50         }
51         return result.toString();
52     }
53 }
```

## Python

```
1 class Solution:
2     def decodeString(self, s: str) -> str:
3         number_stack = []
4         text_stack = [""]
5         current_text = ""
6         current_number = ""
7
8         for c in s:
9             if c.isdigit():
10                # 数字字符
11                if current_text:
12                    text_stack[-1] += current_text
13                    current_text = ""
14                    current_number += c
15            elif c == '[':
16                # 开启新嵌套
17                text_stack.append("")
18                number_stack.append(int(current_number))
19                current_number = ""
20            elif c == ']':
21                # 嵌套结束
22                if current_text:
23                    text_stack[-1] += current_text
24                    current_text = ""
25                repeat_str = text_stack.pop()
26                count = number_stack.pop()
27                text_stack[-1] += repeat_str * count
28            else:
29                # 普通字符
30                current_text += c
31
32        if current_text:
33            text_stack.append(current_text)
34
35        return ''.join(text_stack)
```

## JavaScript

```
1  /**
2   * @param {string} s
3   * @return {string}
4   */
5  function decodeString(s) {
6      const numberStack = [];
7      const textStack = [];
8      let currentText = "";
9      let currentNumber = "";
10
11     for (let c of s) {
12         if (/\\d/.test(c)) {
13             // 数字字符
14             if (currentText !== "") {
15                 textStack[textStack.length - 1] += currentText;
16                 currentText = "";
17             }
18             currentNumber += c;
19         } else if (c === '[') {
20             // 开启新嵌套
21             textStack.push("");
22             numberStack.push(parseInt(currentNumber));
23             currentNumber = "";
24         } else if (c === ']') {
25             // 嵌套结束
26             if (currentText !== "") {
27                 textStack[textStack.length - 1] += currentText;
28                 currentText = "";
29             }
30             const str = textStack.pop();
31             const count = numberStack.pop();
32             textStack[textStack.length - 1] += str.repeat(count);
33         } else {
34             // 普通字符
35             currentText += c;
36         }
37     }
38
39     if (currentText !== "") {
40         textStack.push(currentText);
41     }
42
43     return textStack.join("");
44 }
```



Go

```
1 func decodeString(s string) string {
2     numberStack := []int{}
3     textStack := []string{}
4     currentText := ""
5     currentNumber := ""
6
7     for _, c := range s {
8         ch := string(c)
9         if ch >= "0" && ch <= "9" {
10             // 数字字符
11             if currentText != "" {
12                 textStack[len(textStack)-1] += currentText
13                 currentText = ""
14             }
15             currentNumber += ch
16         } else if ch == "[" {
17             // 开启新嵌套
18             textStack = append(textStack, "")
19             num, _ := strconv.Atoi(currentNumber)
20             numberStack = append(numberStack, num)
21             currentNumber = ""
22         } else if ch == "]" {
23             // 嵌套结束
24             if currentText != "" {
25                 textStack[len(textStack)-1] += currentText
26                 currentText = ""
27             }
28             repeatStr := textStack[len(textStack)-1]
29             textStack = textStack[:len(textStack)-1]
30             count := numberStack[len(numberStack)-1]
31             numberStack = numberStack[:len(numberStack)-1]
32             repeated := strings.Repeat(repeatStr, count)
33             textStack[len(textStack)-1] += repeated
34         } else {
35             // 普通字符
36             currentText += ch
37         }
38     }
39
40     if currentText != "" {
41         textStack = append(textStack, currentText)
42     }
43
44     return strings.Join(textStack, "")
45 }
```

---

来自: [华为OD面试手撕真题 - 字符串解码 \(C++ & Python & JAVA & JS & GO\)-CSDN博客](#)

# 华为OD面试手撕真题 - 最长递增子序列-CSDN 博客

## 最长递增子序列

华为面试手撕高频真题目录点击查看: [华为面试手撕高频真题目录](#)

### 题目描述

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。  
子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3, 6, 2, 7]` 是数组 `[0, 3, 1, 6, 2, 2, 7]` 的子序列。

### 示例1

▼ Plain Text |

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

### 示例2

▼ Plain Text |

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

### 示例3

▼ Plain Text |

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

### 提示

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

进阶：

- 你能将算法的时间复杂度降低到  $O(n \log(n))$  吗？

## 题解

[力扣原题链接](#)

思路：[动态规划](#)

1. 题目要求算法复杂度为  $O(n \log(n))$  ,可以使用 [动态规划](#) + [二分](#) 来实现。
2. 定义 `dp` 数组, `dp[i]` 的含义为 组成 `i+1` 严格递增长度子串 末尾的最小值
3. 状态转移方程为
  - a. `nums[i] > dp[dp.size() - 1]` 时, 说明为此时长度则加一, 将 `nums[i]` 压入数组尾部。
  - b. `nums[i] <= dp[dp.size() - 1]` 时, 使用 [二分算法](#) 找到第一个 大于等于的 `nums[i]` 的位置元素, 并更换为 `nums[i]`
4. 经过2的逻辑进行逻辑, 最终能够得到最长递增子序列长度就为 `dp` 的长度。

**C++**

```
1  class Solution {
2  public:
3      int lengthOfLIS(vector<int>& nums) {
4          // dp数组 其中dp[i] 的含义 组成 i+1 严格递增长度子串 末尾的最小值
5          vector<int> dp;
6          int n = nums.size();
7          for (int i = 0; i < n; i++) {
8              // 说明长度可以加一
9              if (dp.empty() || dp.back() < nums[i]) {
10                 dp.push_back(nums[i]);
11             } else {
12                 // 替换指定对应长度 末尾最小值
13                 auto it = lower_bound(dp.begin(), dp.end(), nums[i]);
14                 *it = nums[i];
15             }
16         }
17         return dp.size();
18     }
19 };
```

**JAVA**

```
1  import java.util.*;
2
3  class Solution {
4      public int lengthOfLIS(int[] nums) {
5          // dp 数组, 其中 dp[i] 表示长度为 i+1 的严格递增子序列的最小末尾值
6          List<Integer> dp = new ArrayList<>();
7          for (int num : nums) {
8              // 说明长度可以加一
9              if (dp.isEmpty() || dp.get(dp.size() - 1) < num) {
10                 dp.add(num);
11             } else {
12                 // 替换对应长度的末尾最小值
13                 int left = 0, right = dp.size() - 1;
14                 while (left < right) {
15                     int mid = (left + right) / 2;
16                     if (dp.get(mid) < num) {
17                         left = mid + 1;
18                     } else {
19                         right = mid;
20                     }
21                 }
22                 dp.set(left, num);
23             }
24         }
25         return dp.size();
26     }
27 }
```

## Python

```
1  import bisect
2
3  class Solution:
4      def lengthOfLIS(self, nums: list[int]) -> int:
5          # dp 数组, 其中 dp[i] 表示长度为 i+1 的严格递增子序列的最小末尾值
6          dp = []
7          for num in nums:
8              # 说明长度可以加一
9              if not dp or dp[-1] < num:
10                 dp.append(num)
11             else:
12                 # 替换对应长度的末尾最小值
13                 index = bisect.bisect_left(dp, num)
14                 dp[index] = num
15         return len(dp)
```

## JavaScript

```
1  /**
2   * @param {number[]} nums
3   * @return {number}
4   */
5  function lengthOfLIS(nums) {
6      // dp 数组, 其中 dp[i] 表示长度为 i+1 的严格递增子序列的最小末尾值
7      const dp = [];
8      for (let num of nums) {
9          // 说明长度可以加一
10         if (dp.length === 0 || dp[dp.length - 1] < num) {
11             dp.push(num);
12         } else {
13             // 替换对应长度的末尾最小值
14             let left = 0, right = dp.length - 1;
15             while (left < right) {
16                 let mid = Math.floor((left + right) / 2);
17                 if (dp[mid] < num) {
18                     left = mid + 1;
19                 } else {
20                     right = mid;
21                 }
22             }
23             dp[left] = num;
24         }
25     }
26     return dp.length;
27 }
```

Go



```
1 func lengthOfLIS(nums []int) int {
2     // dp 数组, 其中 dp[i] 表示长度为 i+1 的严格递增子序列的最小末尾值
3     dp := []int{}
4     for _, num := range nums {
5         // 说明长度可以加一
6         if len(dp) == 0 || dp[len(dp)-1] < num {
7             dp = append(dp, num)
8         } else {
9             // 替换对应长度的末尾最小值
10            idx := sort.Search(len(dp), func(i int) bool { return dp[i] >= num
11        })
12        dp[idx] = num
13    }
14    return len(dp)
15 }
```

来自: [华为OD面试手撕真题 - 最长递增子序列-CSDN博客](#)

# 华为OD面试手撕真题 - 课程表 II -CSDN博客

## 题目描述

现在你总共有 `numCourses` 门课程需要选，记为 `0` 到 `numCourses - 1`。给你一个数组 `prerequisites`，其中 `prerequisites[i] = [ai, bi]`，表示在选修课程 `ai` 前 必须 先选修 `bi`。

- 例如，想要学习课程 `0`，你需要先完成课程 `1`，我们用一个匹配来表示：`[0,1]`。

返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序，你只要返回 任意一种 就可以了。如果不可能完成所有课程，返回 一个空数组。

## 示例1

▼ Plain Text

```
1  输入: numCourses = 2, prerequisites = [[1,0]]
2  输出: [0,1]
3  解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1]。
```

## 示例2

▼ Plain Text

```
1  输入: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
2  输出: [0,2,1,3]
3  解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
4  因此，一个正确的课程顺序是 [0,1,2,3]。另一个正确的排序是 [0,2,1,3]。
```

## 示例3

▼ Plain Text

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

## 提示

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= numCourses * (numCourses - 1)`
- `prerequisites[i].length == 2`
- `0 <= ai, bi < numCourses`
- `ai != bi`
- 所有 `[ai, bi]` 互不相同

## 题解

[力扣原题链接](#)

思路：拓扑排序 练习题

题目要求一个课程能够学得将他所有依赖的课程先学完，这个关系可以转换为 拓扑排序 中的入度，只有入度为0的课程才能学习。基于此可以使用下面这种逻辑进行处理这道题：

1. 根据输入数据统计各个课程的入度，并利用 哈希表 来记录 课程a是那些课程的先修课 关系，方便后续更新依赖课程的入度。
2. 接下来使用 bfs 模拟拓扑排序就行，下面代码采用 队列模拟BFS，初始将所有入度为0的课程加入队列，接下来进行处理过程：
  - 每次循环取出队列中队首课程，更新将该课程当作先修课的入度值，如果某个依赖该课程的课程入度变更为  $\leq 0$  则加入队列。
3. 常规情况下，根据2的规律进行迭代即可求出结果。但是有个特殊情况需要处理，就是拥有 循环依赖 的情况，其实这就是题目中说 不可能完成所有课程 情况。存在循环依赖的情况怎么判断？这个有个定理，拓扑排序如果存在循环依赖，在进行BFS模拟时，入队次数会大于课程数量。所以在进行BFS遍历过程中，定义count统计入队数量，然后进行判断就行。

C++

```
1  class Solution {
2  public:
3      vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
4          // 记录每个课程的入度
5          map<int,int> inOrderCount;
6          // 初始化入度为0
7          for (int i = 0; i < numCourses; i++) {
8              inOrderCount[i] = 0;
9          }
10
11         // 记录依赖关系
12         map<int, set<int>> depend;
13         int n = prerequisites.size();
14         for (int i = 0; i < n; i++) {
15             int second = prerequisites[i][0];
16             int first = prerequisites[i][1];
17             inOrderCount[second]++;
18             depend[first].insert(second);
19         }
20         // 处理拓扑排序
21         queue<int> q;
22         // 初始添加入度为0的课程
23         for (auto &p : inOrderCount) {
24             if (p.second == 0) {
25                 q.push(p.first);
26             }
27         }
28         // 存储结果
29         vector<int> res(numCourses);
30
31         // 已访问课程数量
32         int count = 0;
33         while (!q.empty()) {
34             int top = q.front();
35             q.pop();
36             count++;
37             if (count > numCourses) {
38                 break;
39             }
40             res[count - 1] = top;
41
42             // 更新依赖它的课程入度
43             for (auto id : depend[top]) {
44                 inOrderCount[id]--;
```

```
45         if (inOrderCount[id] <= 0) {
46             q.push(id);
47         }
48     }
49 }
50 if (count != numCourses) {
51     return {};
52 }
53 return res;
54 }
55 };
```

## JAVA

```
1  class Solution {
2      public int[] findOrder(int numCourses, int[][] prerequisites) {
3          // 记录每个课程的入度
4          int[] inOrderCount = new int[numCourses];
5          // 记录依赖关系
6          Map<Integer, List<Integer>> depend = new HashMap<>();
7
8          for (int[] pre : prerequisites) {
9              int second = pre[0];
10             int first = pre[1];
11             inOrderCount[second]++;
12             depend.computeIfAbsent(first, k -> new ArrayList<>()).add(second);
13         }
14
15         // 处理拓扑排序
16         Queue<Integer> q = new LinkedList<>();
17         // 初始添加入度为0的课程
18         for (int i = 0; i < numCourses; i++) {
19             if (inOrderCount[i] == 0) {
20                 q.offer(i);
21             }
22         }
23
24         int[] res = new int[numCourses];
25         int count = 0;
26
27         while (!q.isEmpty()) {
28             int top = q.poll();
29             res[count++] = top;
30
31             if (depend.containsKey(top)) {
32                 for (int id : depend.get(top)) {
33                     inOrderCount[id]--;
34                     if (inOrderCount[id] == 0) {
35                         q.offer(id);
36                     }
37                 }
38             }
39         }
40
41         if (count != numCourses) return new int[0];
42         return res;
43     }
44 }
```

## Python

```
▼ Plain Text |
1  from collections import deque, defaultdict
2
3  class Solution:
4      def findOrder(self, numCourses: int, prerequisites: list[list[int]]) -
> list[int]:
5      # 记录每个课程的入度
6      in_order_count = [0] * numCourses
7      # 记录依赖关系
8      depend = defaultdict(list)
9
10     for second, first in prerequisites:
11         in_order_count[second] += 1
12         depend[first].append(second)
13
14     # 初始添加入度为0的课程
15     q = deque()
16     for i in range(numCourses):
17         if in_order_count[i] == 0:
18             q.append(i)
19
20     res = []
21     count = 0
22
23     while q:
24         top = q.popleft()
25         res.append(top)
26         count += 1
27         for neighbor in depend[top]:
28             in_order_count[neighbor] -= 1
29             if in_order_count[neighbor] == 0:
30                 q.append(neighbor)
31
32     return res if count == numCourses else []
```

## JavaScript

```
1  /**
2   * @param {number} numCourses
3   * @param {number[][]} prerequisites
4   * @return {number[]}
5   */
6  var findOrder = function(numCourses, prerequisites) {
7      // 记录每个课程的入度
8      const inOrderCount = Array(numCourses).fill(0);
9      // 记录依赖关系
10     const depend = new Map();
11
12     for (const [second, first] of prerequisites) {
13         inOrderCount[second]++;
14         if (!depend.has(first)) depend.set(first, []);
15         depend.get(first).push(second);
16     }
17
18     // 初始添加入度为0的课程
19     const queue = [];
20     for (let i = 0; i < numCourses; i++) {
21         if (inOrderCount[i] === 0) queue.push(i);
22     }
23
24     const res = [];
25     let count = 0;
26
27     while (queue.length > 0) {
28         const top = queue.shift();
29         res.push(top);
30         count++;
31         if (depend.has(top)) {
32             for (const neighbor of depend.get(top)) {
33                 inOrderCount[neighbor]--;
34                 if (inOrderCount[neighbor] === 0) {
35                     queue.push(neighbor);
36                 }
37             }
38         }
39     }
40
41     return count === numCourses ? res : [];
42 };
```



```
1 func findOrder(numCourses int, prerequisites [][]int) []int {
2     // 记录每个课程的入度
3     inOrderCount := make([]int, numCourses)
4     // 记录依赖关系
5     depend := make(map[int][]int)
6
7     for _, pre := range prerequisites {
8         second := pre[0]
9         first := pre[1]
10        inOrderCount[second]++
11        depend[first] = append(depend[first], second)
12    }
13
14    // 初始添加入度为0的课程
15    queue := []int{}
16    for i := 0; i < numCourses; i++ {
17        if inOrderCount[i] == 0 {
18            queue = append(queue, i)
19        }
20    }
21
22    res := []int{}
23    count := 0
24
25    for len(queue) > 0 {
26        top := queue[0]
27        queue = queue[1:]
28        res = append(res, top)
29        count++
30        for _, id := range depend[top] {
31            inOrderCount[id]--
32            if inOrderCount[id] == 0 {
33                queue = append(queue, id)
34            }
35        }
36    }
37
38    if count != numCourses {
39        return []int{}
40    }
41    return res
42 }
```

来自: [华为OD面试手撕真题 - 课程表 II - CSDN博客](#)

# 华为OD面试手撕真题 - 子集-CSDN博客

## 题目描述

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。  
解集 **不能** 包含 **重复** 的子集。你可以按 **任意顺序** 返回解集。

## 用例1

	Plain Text
1	输入: <code>nums = [1,2,3]</code>
2	输出: <code>[[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]</code>

## 用例2

	Plain Text
1	输入: <code>nums = [0]</code>
2	输出: <code>[[], [0]]</code>

## 备注

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有元素 **互不相同**

## 题解

[力扣原题链接](#)

思路: 动态规划思路

1. 假设这道题是求数量，创建 `dp` 数组，`dp[i]` 含义为前 `i` 个字符拥有的子集数量，初始化 `dp[0] = 1`，因为 `[]` 也属于一个子集。那么这个动态规划的转移方程就是: `dp[i] = 2 * dp[i-1]` 因为 `nums[i]` 有两种处理情况，**选择和不选择**。
2. 再来看这道题，需要求出具体的分配方案，初始设置 `res = [[]]`，接下来直接使用两层循环，每次将上一个位置得到的子集保留，然后再往上一个所有子集位置添加当前值，这样就能构成当前位置的子集。

C++

```
1  class Solution {
2  public:
3      vector<vector<int>> subsets(vector<int>& nums) {
4          // 定义结果数组
5          vector<vector<int>> res;
6          res.push_back({});
7          int n = nums.size();
8          for (int i = 0; i < n; i++) {
9              int currentLen = res.size();
10             // 全部添加当前值
11             for (int j = 0; j < currentLen; j++) {
12                 vector<int> tmp = res[j];
13                 tmp.push_back(nums[i]);
14                 res.push_back(tmp);
15             }
16         }
17         return res;
18     }
19 };
```

## JAVA

```
1  class Solution {
2      public List<List<Integer>> subsets(int[] nums) {
3          // 定义结果数组
4          List<List<Integer>> res = new ArrayList<>();
5          res.add(new ArrayList<>());
6          int n = nums.length;
7          for (int i = 0; i < n; i++) {
8              int currentLen = res.size();
9              // 全部添加当前值
10             for (int j = 0; j < currentLen; j++) {
11                 List<Integer> tmp = new ArrayList<>(res.get(j));
12                 tmp.add(nums[i]);
13                 res.add(tmp);
14             }
15         }
16         return res;
17     }
18 }
```

## Python



Plain Text |

```
1 class Solution:
2     def subsets(self, nums):
3         # 定义结果数组
4         res = [[]]
5         n = len(nums)
6         for i in range(n):
7             current_len = len(res)
8             # 全部添加当前值
9             for j in range(current_len):
10                 tmp = res[j][:] # 拷贝当前子集
11                 tmp.append(nums[i])
12                 res.append(tmp)
13         return res
```

## JavaScript



Plain Text |

```
1 var subsets = function(nums) {
2     // 定义结果数组
3     let res = [[]];
4     let n = nums.length;
5     for (let i = 0; i < n; i++) {
6         let currentLen = res.length;
7         // 全部添加当前值
8         for (let j = 0; j < currentLen; j++) {
9             let tmp = res[j].slice(); // 拷贝当前子集
10            tmp.push(nums[i]);
11            res.push(tmp);
12        }
13    }
14    return res;
15 };
```

## Go

```
1 func subsets(nums []int) [][]int {
2     // 定义结果数组
3     res := [][]int{{}}
4     n := len(nums)
5     for i := 0; i < n; i++ {
6         currentLen := len(res)
7         // 全部添加当前值
8         for j := 0; j < currentLen; j++ {
9             tmp := make([]int, len(res[j]))
10            copy(tmp, res[j])
11            tmp = append(tmp, nums[i])
12            res = append(res, tmp)
13        }
14    }
15    return res
16 }
```

来自: [华为OD面试手撕真题 - 子集-CSDN博客](#)

# 求1-n的最小公倍数-CSDN博客

## 题目描述

给你一个整数n，求出[1,n]所有数的[最小公倍数](#)，并对1000000007取模（ $1 \leq n \leq 10000000$ ）

## 输入描述

输入一个整数

## 输出描述

输出一个整数

## 用例1

### 输入

▼		Plain Text
1	10	

### 输出

▼		Plain Text
1	2520	

## 用例2

### 输入

▼		Plain Text
1	11	

### 输出

▼		Plain Text
1	27720	

## 题解

思路：数学问题，使用 素数筛 + 最大幂乘积 解决

- 数字分为：质数 + 合数(合数一般是一个或多个质数相乘)。
- 通过 素数筛 快速求出  $1-n$  中范围的质数。素数筛的算法逻辑：
  - for循环从小到达遍历，如果遍历到的这个数没有被标记为合数，那么它就是质数。
  - 然后将质数的倍数全部标记为合数。
- 然后通过 最大幂乘积 快速求出  $1-n$  的最小公倍数。简单来说就是将  $1-n$  中所有质数 $pi$ 转换为  $pi^{k_i}$  其中需要满足  $pi^{k_i} \leq n$ 。简单举个例子说明 最大幂乘积 原理

▼ Plain Text

```
1  假设当前有这些数：
2  4 = 2^2
3  6 = 2 * 3
4  9 = 3 ^ 2
5  为了能整除4 你需要至少有 2^2
6  为了能整除6， 你至少需要有 2^1 和 3 ^1
7  为了能整除9， 你至少有3^2
8  因此LCM(4,6,9) = 2^2 * 3^2 = 36
```

- 所以 $1-n$ 我们只需要考虑每个质数的 $pi$ 的最大幂既可。

C++



```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  // 质数筛
6  vector<bool> sieve(int n ) {
7      vector<bool> isPrime(n + 1, true);
8      isPrime[0] = isPrime[1] = false;
9      for (int i = 2; i * i <= n; i++) {
10         if (isPrime[i]) {
11             // 把质数的所有倍数全部标记非质数
12             for (int j = i * i; j <= n; j+= i) {
13                 isPrime[j] = false;
14             }
15         }
16     }
17     return isPrime;
18 }
19
20
21 int main() {
22     int n ;
23     cin >> n;
24     vector<bool> isPrime = sieve(n);
25     vector<int> primeNum;
26     for (int i = 1; i <= n; i++) {
27         if (isPrime[i]) {
28             primeNum.push_back(i);
29         }
30     }
31     // 防止溢出
32     unsigned long long res = 1;
33
34     // 最大幂定理
35     for (int i = 0; i < primeNum.size(); i++) {
36         int tmp = primeNum[i];
37         while (tmp * primeNum[i] <= n) {
38             tmp *= primeNum[i];
39         }
40         res = (res *tmp) % 1000000007;
41     }
42
43     cout << res;
44     return 0;
45 }
```

---

## JAVA

```
1  import java.util.*;
2
3  public class Main {
4      static final int MOD = 1000000007;
5
6      // 质数筛
7      static boolean[] sieve(int n) {
8          boolean[] isPrime = new boolean[n + 1];
9          Arrays.fill(isPrime, true);
10         isPrime[0] = isPrime[1] = false;
11         for (int i = 2; i * i <= n; i++) {
12             if (isPrime[i]) {
13                 for (int j = i * i; j <= n; j += i) {
14                     isPrime[j] = false;
15                 }
16             }
17         }
18         return isPrime;
19     }
20
21     public static void main(String[] args) {
22         Scanner sc = new Scanner(System.in);
23         int n = sc.nextInt();
24
25         boolean[] isPrime = sieve(n);
26         List<Integer> primeNum = new ArrayList<>();
27         for (int i = 2; i <= n; i++) {
28             if (isPrime[i]) {
29                 primeNum.add(i);
30             }
31         }
32
33         long res = 1;
34         for (int p : primeNum) {
35             long tmp = p;
36             while (tmp * p <= n) {
37                 tmp *= p;
38             }
39             res = (res * tmp) % MOD;
40         }
41
42         System.out.println(res);
43     }
44 }
```

## Python

Plain Text

```
1  MOD = 1000000007
2
3  # 质数筛
4  def sieve(n):
5      is_prime = [True] * (n + 1)
6      is_prime[0:2] = [False, False]
7      for i in range(2, int(n ** 0.5) + 1):
8          if is_prime[i]:
9              for j in range(i * i, n + 1, i):
10                 is_prime[j] = False
11     return is_prime
12
13     n = int(input())
14     is_prime = sieve(n)
15     prime_list = [i for i, val in enumerate(is_prime) if val]
16
17     res = 1
18     for p in prime_list:
19         tmp = p
20         while tmp * p <= n:
21             tmp *= p
22         res = (res * tmp) % MOD
23
24     print(res)
```

## JavaScript

```
1  const readline = require('readline');
2  const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
3
4  const MOD = 1000000007n;
5
6  rl.on('line', (line) => {
7      let n = parseInt(line);
8      let isPrime = Array(n + 1).fill(true);
9      isPrime[0] = isPrime[1] = false;
10
11     // 质数筛
12     for (let i = 2; i * i <= n; i++) {
13         if (isPrime[i]) {
14             for (let j = i * i; j <= n; j += i) {
15                 isPrime[j] = false;
16             }
17         }
18     }
19
20     let res = 1n;
21     for (let i = 2; i <= n; i++) {
22         if (isPrime[i]) {
23             let tmp = BigInt(i);
24             while (tmp * BigInt(i) <= BigInt(n)) {
25                 tmp *= BigInt(i);
26             }
27             res = (res * tmp) % MOD;
28         }
29     }
30
31     console.log(res.toString());
32     rl.close();
33 });
```

Go

```
1  package main
2
3  import (
4      "fmt"
5      "math/big"
6  )
7
8  const MOD = 1000000007
9
10 // 质数筛
11 func sieve(n int) []bool {
12     isPrime := make([]bool, n+1)
13     for i := range isPrime {
14         isPrime[i] = true
15     }
16     isPrime[0], isPrime[1] = false, false
17     for i := 2; i*i <= n; i++ {
18         if isPrime[i] {
19             for j := i * i; j <= n; j += i {
20                 isPrime[j] = false
21             }
22         }
23     }
24     return isPrime
25 }
26
27 func main() {
28     var n int
29     fmt.Scan(&n)
30     isPrime := sieve(n)
31
32     res := big.NewInt(1)
33     mod := big.NewInt(MOD)
34
35     for i := 2; i <= n; i++ {
36         if isPrime[i] {
37             tmp := big.NewInt(int64(i))
38             for {
39                 mul := new(big.Int).Mul(tmp, big.NewInt(int64(i)))
40                 if mul.Cmp(big.NewInt(int64(n))) > 0 {
41                     break
42                 }
43                 tmp = mul
44             }
45             res.Mul(res, tmp)
```

```
46         res.Mod(res, mod)
47     }
48 }
49
50     fmt.Println(res.String())
51 }
```

来自: [求1-n的最小公倍数-CSDN博客](#)