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

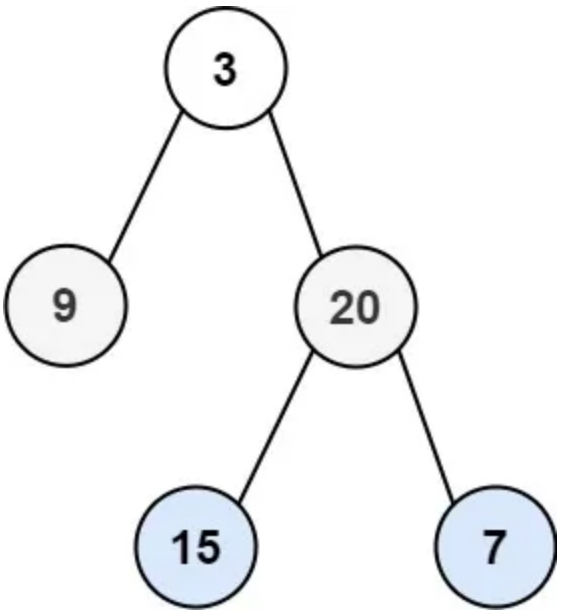
求1-n的最小公倍数-CSDN博客

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




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

示例2

```
输入：root = [1]
输出：[[1]]
```

示例3

```
输入：root = []
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++

Plain Text

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(lef
t), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> res;
        if (root == nullptr) {
            return res;
        }
        // 使用队列模拟层序遍历
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            queue<TreeNode*> tmpQueue;
            // 存储每一层遍历的值
            vector<int> valueArr;
            while (!q.empty()) {
                TreeNode* current = q.front();
                q.pop();
                valueArr.push_back(current->val);
                // 加入左节点
                if (current->left != nullptr) {
                    tmpQueue.push(current->left);
                }
                // 加入右节点
                if (current->right != nullptr) {
                    tmpQueue.push(current->right);
                }
            }

            res.push_back(valueArr);
            q = tmpQueue;
        }
        // 从上到下遍历 反转就是 从下到上
        reverse(res.begin(), res.end());
```

```
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) {
            return res;
        }

        // 使用队列模拟层序遍历
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        while (!q.isEmpty()) {
            Queue<TreeNode> tmpQueue = new LinkedList<>();
            List<Integer> valueArr = new ArrayList<>();

            while (!q.isEmpty()) {
                TreeNode current = q.poll();
                valueArr.add(current.val);

                // 加入左节点
                if (current.left != null) {
                    tmpQueue.offer(current.left);
                }
                // 加入右节点
                if (current.right != null) {
                    tmpQueue.offer(current.right);
                }
            }

            res.add(valueArr);
            q = tmpQueue;
        }

        // 从上到下遍历，反转就是从下到上
        Collections.reverse(res);
        return res;
    }
}
```

Python

```
Plain Text
```

```
from collections import deque

class Solution:
    def levelOrderBottom(self, root):
        res = []
        if not root:
            return res

        # 使用队列模拟层序遍历
        q = deque([root])

        while q:
            tmp_queue = deque()
            value_arr = []

            while q:
                current = q.popleft()
                value_arr.append(current.val)

                # 加入左节点
                if current.left:
                    tmp_queue.append(current.left)
                # 加入右节点
                if current.right:
                    tmp_queue.append(current.right)

            res.append(value_arr)
            q = tmp_queue

        # 从上到下遍历，反转就是从下到上
        return res[::-1]
```

JavaScript

Plain Text

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrderBottom = function(root) {
    const res = [];
    if (!root) return res;

    // 使用队列模拟层序遍历
    let q = [root];

    while (q.length > 0) {
        const tmpQueue = [];
        const valueArr = [];

        for (const node of q) {
            valueArr.push(node.val);

            // 加入左节点
            if (node.left) tmpQueue.push(node.left);
            // 加入右节点
            if (node.right) tmpQueue.push(node.right);
        }

        res.push(valueArr);
        q = tmpQueue;
    }

    // 从上到下遍历，反转就是从下到上
    return res.reverse();
};
```

Go

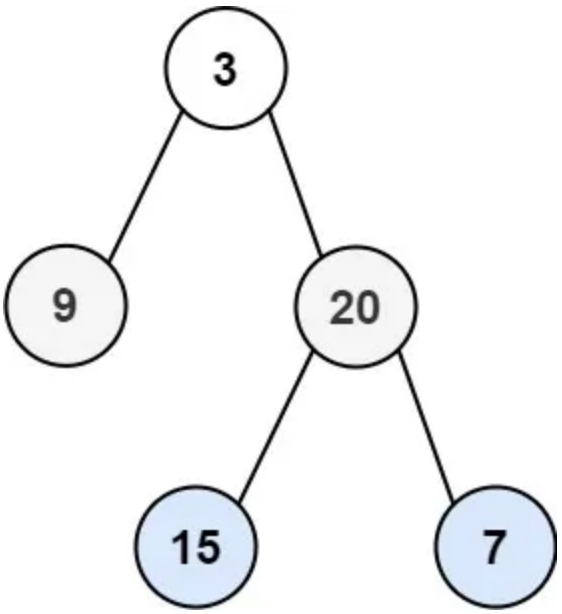
Plain Text

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func levelOrderBottom(root *TreeNode) [][]int {
    var res [][]int
    if root == nil {
        return res
    }

    // 使用队列模拟层序遍历
    q := []*TreeNode{root}

    for len(q) > 0 {
        var tmpQueue []*TreeNode
        var valueArr []int

        for _, node := range q {
            valueArr = append(valueArr, node.Val)

            // 加入左节点
            if node.Left != nil {
                tmpQueue = append(tmpQueue, node.Left)
            }
            // 加入右节点
            if node.Right != nil {
                tmpQueue = append(tmpQueue, node.Right)
            }
        }

        res = append(res, valueArr)
        q = tmpQueue
    }

    // 从上到下遍历，反转就是从下到上
    for i, j := 0, len(res)-1; i < j; i, j = i+1, j-1 {
        res[i], res[j] = res[j], res[i]
    }

    return res
}
```

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




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

示例2

```
输入：root = [1]
输出：[[1]]
```

示例3

```
输入：root = []
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++

Plain Text

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(lef
t), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> res;
        if (root == nullptr) {
            return res;
        }
        // 使用队列模拟层序遍历
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            queue<TreeNode*> tmpQueue;
            // 存储每一层遍历的值
            vector<int> valueArr;
            while (!q.empty()) {
                TreeNode* current = q.front();
                q.pop();
                valueArr.push_back(current->val);
                // 加入左节点
                if (current->left != nullptr) {
                    tmpQueue.push(current->left);
                }
                // 加入右节点
                if (current->right != nullptr) {
                    tmpQueue.push(current->right);
                }
            }

            res.push_back(valueArr);
            q = tmpQueue;
        }
        // 从上到下遍历 反转就是 从下到上
        reverse(res.begin(), res.end());
```

```
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) {
            return res;
        }

        // 使用队列模拟层序遍历
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        while (!q.isEmpty()) {
            Queue<TreeNode> tmpQueue = new LinkedList<>();
            List<Integer> valueArr = new ArrayList<>();

            while (!q.isEmpty()) {
                TreeNode current = q.poll();
                valueArr.add(current.val);

                // 加入左节点
                if (current.left != null) {
                    tmpQueue.offer(current.left);
                }
                // 加入右节点
                if (current.right != null) {
                    tmpQueue.offer(current.right);
                }
            }

            res.add(valueArr);
            q = tmpQueue;
        }

        // 从上到下遍历，反转就是从下到上
        Collections.reverse(res);
        return res;
    }
}
```

Python

```
Plain Text
```

```
from collections import deque

class Solution:
    def levelOrderBottom(self, root):
        res = []
        if not root:
            return res

        # 使用队列模拟层序遍历
        q = deque([root])

        while q:
            tmp_queue = deque()
            value_arr = []

            while q:
                current = q.popleft()
                value_arr.append(current.val)

                # 加入左节点
                if current.left:
                    tmp_queue.append(current.left)
                # 加入右节点
                if current.right:
                    tmp_queue.append(current.right)

            res.append(value_arr)
            q = tmp_queue

        # 从上到下遍历，反转就是从下到上
        return res[::-1]
```

JavaScript

Plain Text

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrderBottom = function(root) {
    const res = [];
    if (!root) return res;

    // 使用队列模拟层序遍历
    let q = [root];

    while (q.length > 0) {
        const tmpQueue = [];
        const valueArr = [];

        for (const node of q) {
            valueArr.push(node.val);

            // 加入左节点
            if (node.left) tmpQueue.push(node.left);
            // 加入右节点
            if (node.right) tmpQueue.push(node.right);
        }

        res.push(valueArr);
        q = tmpQueue;
    }

    // 从上到下遍历，反转就是从下到上
    return res.reverse();
};
```

Go

Plain Text

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func levelOrderBottom(root *TreeNode) [][]int {
    var res [][]int
    if root == nil {
        return res
    }

    // 使用队列模拟层序遍历
    q := []*TreeNode{root}

    for len(q) > 0 {
        var tmpQueue []*TreeNode
        var valueArr []int

        for _, node := range q {
            valueArr = append(valueArr, node.Val)

            // 加入左节点
            if node.Left != nil {
                tmpQueue = append(tmpQueue, node.Left)
            }
            // 加入右节点
            if node.Right != nil {
                tmpQueue = append(tmpQueue, node.Right)
            }
        }

        res = append(res, valueArr)
        q = tmpQueue
    }

    // 从上到下遍历，反转就是从下到上
    for i, j := 0, len(res)-1; i < j; i, j = i+1, j-1 {
        res[i], res[j] = res[j], res[i]
    }

    return res
}
```

> 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

Plain Text

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

示例2

Plain Text

```
输入：sticks = [1,8,3,5]
输出：30
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]所有木棍已经连成一根，总费用4+9+17=30
```

提示

- 1 <= sticks.length <= 10^4
- 1 <= sticks[i] <= 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++

Plain Text

```
class Solution {
public:
    int countSmaller(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        // 定义小顶堆 小的值在栈顶
        priority_queue<int, vector<int>, greater<>> pq;
        for (int i = 0; i < nums.size(); i++) {
            pq.push(nums[i]);
        }
        int res = 0;

        // 每次取出两个最短木棍进行连接, 贪心
        while (pq.size() != 1) {
            int first = pq.top();
            pq.pop();
            int second = pq.top();
            pq.pop();
            // 成本
            res += (first + second);
            // 新生成的木棍重新加入木棍
            pq.push(first + second);
        }
        return res;
    }
};
```

JAVA

Plain Text

```
import java.util.*;

public class Solution {
    public int countSmaller(int[] nums) {
        if (nums.length == 0) return 0;

        // 定义小顶堆，小的值在堆顶
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int num : nums) {
            pq.offer(num);
        }

        int res = 0;

        // 每次取出两个最短木棍进行连接，贪心
        while (pq.size() > 1) {
            int first = pq.poll();
            int second = pq.poll();
            // 成本
            res += first + second;
            // 新生成的木棍重新加入
            pq.offer(first + second);
        }

        return res;
    }
}
```

Python

```
import heapq

class Solution:
    def countSmaller(self, nums):
        if not nums:
            return 0

        # 定义小顶堆，小的值在堆顶
        heapq.heapify(nums)

        res = 0

        # 每次取出两个最短木棍进行连接，贪心
        while len(nums) > 1:
            first = heapq.heappop(nums)
            second = heapq.heappop(nums)
            # 成本
            res += first + second
            # 新生成的木棍重新加入
            heapq.heappush(nums, first + second)

        return res
```

JavaScript

Plain Text

```
class MinHeap {
    constructor() {
        this.data = [];
    }

    push(val) {
        this.data.push(val);
        this._bubbleUp();
    }

    pop() {
        if (this.data.length === 1) return this.data.pop();
        const min = this.data[0];
        this.data[0] = this.data.pop();
        this._bubbleDown();
        return min;
    }

    _bubbleUp() {
        let i = this.data.length - 1;
        while (i > 0) {
            const p = Math.floor((i - 1) / 2);
            if (this.data[p] <= this.data[i]) break;
            [this.data[p], this.data[i]] = [this.data[i], this.data[p]];
            i = p;
        }
    }

    _bubbleDown() {
        let i = 0;
        const n = this.data.length;
        while (true) {
            let l = 2 * i + 1, r = 2 * i + 2, smallest = i;
            if (l < n && this.data[l] < this.data[smallest]) smallest = l;
            if (r < n && this.data[r] < this.data[smallest]) smallest = r;
            if (smallest === i) break;
            [this.data[i], this.data[smallest]] = [this.data[smallest], th
is.data[i]];
            i = smallest;
        }
    }

    size() {
        return this.data.length;
    }
```

```
}

function countSmaller(nums) {
    if (nums.length === 0) return 0;

    // 定义小顶堆，小的值在堆顶
    const pq = new MinHeap();
    for (const num of nums) {
        pq.push(num);
    }

    let res = 0;

    // 每次取出两个最短木棍进行连接，贪心
    while (pq.size() > 1) {
        const first = pq.pop();
        const second = pq.pop();
        // 成本
        res += first + second;
        // 新生成的木棍重新加入
        pq.push(first + second);
    }

    return res;
}
```

Go

Plain Text

```
package main

import (
  "container/heap"
  "fmt"
)

// 定义小顶堆结构
type MinHeap []int

func (h MinHeap) Len() int            { return len(h) }
func (h MinHeap) Less(i, j int) bool  { return h[i] < h[j] } // 小顶堆
func (h MinHeap) Swap(i, j int)       { h[i], h[j] = h[j], h[i] }
func (h *MinHeap) Push(x any)         { *h = append(*h, x.(int)) }
func (h *MinHeap) Pop() any {
  old := *h
  n := len(old)
  val := old[n-1]
  *h = old[:n-1]
  return val
}

func countSmaller(nums []int) int {
  if len(nums) == 0 {
    return 0
  }

  // 定义小顶堆，小的值在堆顶
  h := &MinHeap{}
  for _, num := range nums {
    heap.Push(h, num)
  }
  heap.Init(h)

  res := 0

  // 每次取出两个最短木棍进行连接，贪心
  for h.Len() > 1 {
    first := heap.Pop(h).(int)
    second := heap.Pop(h).(int)
    // 成本
    res += first + second
    // 新生成的木棍重新加入
    heap.Push(h, first+second)
  }
```

```
    return res
}

func main() {
    nums := []int{1, 3, 5, 2}
    fmt.Println(countSmaller(nums)) // 输出总成本
}
```

> 来自: 华为OD 面试手撕真题 – 连接棒材的的最低费用–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 <= x < 10^n` 。

示例1

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

示例2

```
输入：n = 0
输出：1
```

提示

- `0 <= n <= 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++

Plain Text

```
class Solution {
public:
    int countNumbersWithUniqueDigits(int n) {
        if (n == 0) {
            return 1;
        }
        if (n == 1) {
            return 10;
        }
        int ans = 10;

        int number = 9;
        for (int i = 0; i < n - 1; i++) {
            // 最后位可选数量等于 上一个最后数可选数 * -1
            number *= 9 - i;
            ans += number;
        }
        return ans;
    }
};
```

JAVA

```
class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        if (n == 0) {
            return 1;
        }
        if (n == 1) {
            return 10;
        }
        int ans = 10;

        int number = 9;
        for (int i = 0; i < n - 1; i++) {
            // 最后位可选数量等于上一个最后数可选数 * (9 - i)
            number *= 9 - i;
            ans += number;
        }
        return ans;
    }
}
```

Python

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

JavaScript

```
function countNumbersWithUniqueDigits(n) {
    if (n === 0) {
        return 1;
    }
    if (n === 1) {
        return 10;
    }
    let ans = 10;
    let number = 9;
    for (let i = 0; i < n - 1; i++) {
        // 最后位可选数量等于上一个最后数可选数 * (9 - i)
        number *= (9 - i);
        ans += number;
    }
    return ans;
}
```

Go

```
func countNumbersWithUniqueDigits(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 10
    }
    ans := 10
    number := 9
    for i := 0; i < n-1; i++ {
        // 最后位可选数量等于上一个最后数可选数 * (9 - i)
        number *= 9 - i
        ans += number
    }
    return ans
}
```

> 来自: 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

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

```
输入：s = "aba"
输出：true
```

示例2

Plain Text

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

示例3

Plain Text

```
输入：s = "abc"
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++

Plain Text

```
class Solution {
public:
    // 判断是否满足回文串
    bool valid(string &s, int left, int right) {
        while (left < right) {
            if (s[left] != s[right]) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    bool validPalindrome(string s) {
        int n = s.size();
        // 本身就是回文串
        if (n == 1) {
            return true;
        }
        int left = 0, right = n -1;
        while (left < right) {
            if (s[left] != s[right]) {
               // 尝试移除左边字符 和 右边字符判断是否满足回文串
               return valid(s, left + 1, right) || valid(s, left, right - 1);
            }
            left++;
            right--;
        }
        return true;
    }
};
```

JAVA

Plain Text

```
class Solution {
    // 判断是否满足回文串
    private boolean valid(String s, int left, int right) {
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    public boolean validPalindrome(String s) {
        int n = s.length();
        // 本身就是回文串
        if (n == 1) {
            return true;
        }
        int left = 0, right = n - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                // 尝试移除左边字符 和 右边字符判断是否满足回文串
                return valid(s, left + 1, right) || valid(s, left, right - 1);
            }
            left++;
            right--;
        }
        return true;
    }
}
```

Python

```
class Solution:
    # 判断是否满足回文串
    def valid(self, s: str, left: int, right: int) -> bool:
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True

    def validPalindrome(self, s: str) -> bool:
        n = len(s)
        # 本身就是回文串
        if n == 1:
            return True
        left, right = 0, n - 1
        while left < right:
            if s[left] != s[right]:
                # 尝试移除左边字符 和 右边字符判断是否满足回文串
                return self.valid(s, left + 1, right) or self.valid(s, left, right - 1)
            left += 1
            right -= 1
        return True
```

JavaScript

Plain Text

```
function valid(s, left, right) {
    // 判断是否满足回文串
    while (left < right) {
        if (s[left] !== s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

function validPalindrome(s) {
    const n = s.length;
    // 本身就是回文串
    if (n === 1) return true;
    let left = 0, right = n - 1;
    while (left < right) {
        if (s[left] !== s[right]) {
            // 尝试移除左边字符 和 右边字符判断是否满足回文串
            return valid(s, left + 1, right) || valid(s, left, right - 1);
        }
        left++;
        right--;
    }
    return true;
}
```

Go

```
// 判断是否满足回文串
func valid(s string, left, right int) bool {
  for left < right {
    if s[left] != s[right] {
      return false
    }
    left++
    right--
  }
  return true
}

func validPalindrome(s string) bool {
  n := len(s)
  // 本身就是回文串
  if n == 1 {
    return true
  }
  left, right := 0, n-1
  for left < right {
    if s[left] != s[right] {
      // 尝试移除左边字符 和 右边字符判断是否满足回文串
      return valid(s, left+1, right) || valid(s, left, right-1)
    }
    left++
    right--
  }
  return true
}
```

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

```
输入：s = "aba"
输出：true
```

示例2

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

示例3

```
输入：s = "abc"
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++

Plain Text

```
class Solution {
public:
    // 判断是否满足回文串
    bool valid(string &s, int left, int right) {
        while (left < right) {
            if (s[left] != s[right]) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    bool validPalindrome(string s) {
        int n = s.size();
        // 本身就是回文串
        if (n == 1) {
            return true;
        }
        int left = 0, right = n -1;
        while (left < right) {
            if (s[left] != s[right]) {
               // 尝试移除左边字符 和 右边字符判断是否满足回文串
               return valid(s, left + 1, right) || valid(s, left, right - 1);
            }
            left++;
            right--;
        }
        return true;
    }
};
```

JAVA

Plain Text

```
class Solution {
    // 判断是否满足回文串
    private boolean valid(String s, int left, int right) {
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    public boolean validPalindrome(String s) {
        int n = s.length();
        // 本身就是回文串
        if (n == 1) {
            return true;
        }
        int left = 0, right = n - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                // 尝试移除左边字符 和 右边字符判断是否满足回文串
                return valid(s, left + 1, right) || valid(s, left, right - 1);
            }
            left++;
            right--;
        }
        return true;
    }
}
```

Python

Plain Text

```
class Solution:
    # 判断是否满足回文串
    def valid(self, s: str, left: int, right: int) -> bool:
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True

    def validPalindrome(self, s: str) -> bool:
        n = len(s)
        # 本身就是回文串
        if n == 1:
            return True
        left, right = 0, n - 1
        while left < right:
            if s[left] != s[right]:
                # 尝试移除左边字符 和 右边字符判断是否满足回文串
                return self.valid(s, left + 1, right) or self.valid(s, left, right - 1)
            left += 1
            right -= 1
        return True
```

JavaScript

Plain Text

```
function valid(s, left, right) {
    // 判断是否满足回文串
    while (left < right) {
        if (s[left] !== s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

function validPalindrome(s) {
    const n = s.length;
    // 本身就是回文串
    if (n === 1) return true;
    let left = 0, right = n - 1;
    while (left < right) {
        if (s[left] !== s[right]) {
            // 尝试移除左边字符 和 右边字符判断是否满足回文串
            return valid(s, left + 1, right) || valid(s, left, right - 1);
        }
        left++;
        right--;
    }
    return true;
}
```

Go

Plain Text

```
// 判断是否满足回文串
func valid(s string, left, right int) bool {
  for left < right {
    if s[left] != s[right] {
      return false
    }
    left++
    right--
  }
  return true
}

func validPalindrome(s string) bool {
  n := len(s)
  // 本身就是回文串
  if n == 1 {
    return true
  }
  left, right := 0, n-1
  for left < right {
    if s[left] != s[right] {
      // 尝试移除左边字符 和 右边字符判断是否满足回文串
      return valid(s, left+1, right) || valid(s, left, right-1)
    }
    left++
    right--
  }
  return true
}
```

> 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

```
输入：s = "ababccc"
输出：5
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

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

示例3

```
输入：s = "aa"
输出：1
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++

```
Plain Text
```

```
class Solution {
public:
    int res = 0;
    // 当前开始下标
    void dfs(int index, string& s, unordered_set<string>& uniquSub) {
        int n = s.size();
        // 剪枝 不可能存在更多的数量
        if (n - index + uniquSub.size() < res) {
            return;
        }
        // 形成一个合法拆分方案
        if (index >= n) {
            res = max(res, (int)uniquSub.size());
            return;
        }
        for (int i = index; i < n; i++) {
            string subString = s.substr(index, i - index + 1);
            // 说明重复,直接跳过
            if (uniquSub.find(subString) != uniquSub.end()) {
                continue;
            }
            // 递归回溯
            uniquSub.insert(subString);
            dfs(i+1, s, uniquSub);
            uniquSub.erase(subString);
        }
    }

    int maxUniqueSplit(string s) {
        unordered_set<string> uniquSub;
        dfs(0, s, uniquSub);
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    int res = 0;

    public int maxUniqueSplit(String s) {
        Set<String> uniqueSub = new HashSet<>();
        dfs(0, s, uniqueSub);
        return res;
    }

    // 回溯函数，index 表示当前起始位置
    void dfs(int index, String s, Set<String> uniqueSub) {
        int n = s.length();
        // 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
        if (n - index + uniqueSub.size() < res) return;

        // 到达末尾，形成一个合法拆分方案
        if (index == n) {
            res = Math.max(res, uniqueSub.size());
            return;
        }

        for (int i = index; i < n; i++) {
            String sub = s.substring(index, i + 1);
            if (uniqueSub.contains(sub)) continue;

            // 尝试添加并回溯
            uniqueSub.add(sub);
            dfs(i + 1, s, uniqueSub);
            uniqueSub.remove(sub); // 回溯删除
        }
    }
}
```

Python

```
class Solution:
    def maxUniqueSplit(self, s: str) -> int:
        self.res = 0

        def dfs(index: int, unique_sub: set[str]):
            n = len(s)
            # 剪枝：如果剩余长度加当前集合大小小于 res，无需继续
            if n - index + len(unique_sub) < self.res:
                return

            # 到达末尾，更新结果
            if index == n:
                self.res = max(self.res, len(unique_sub))
                return

            for i in range(index, n):
                sub = s[index:i + 1]
                if sub in unique_sub:
                    continue
                unique_sub.add(sub)
                dfs(i + 1, unique_sub)
                unique_sub.remove(sub)  # 回溯

        dfs(0, set())
        return self.res
```

JavaScript

```
/**
 * @param {string} s
 * @return {number}
 */
var maxUniqueSplit = function(s) {
    let res = 0;

    function dfs(index, uniqueSub) {
        const n = s.length;
        // 剪枝优化
        if (n - index + uniqueSub.size < res) return;

        // 到达末尾
        if (index === n) {
            res = Math.max(res, uniqueSub.size);
            return;
        }

        for (let i = index; i < n; i++) {
            const sub = s.substring(index, i + 1);
            if (uniqueSub.has(sub)) continue;

            // 添加并回溯
            uniqueSub.add(sub);
            dfs(i + 1, uniqueSub);
            uniqueSub.delete(sub); // 回溯
        }
    }

    dfs(0, new Set());
    return res;
};
```

Go

```
var maxUniqueSplit = func(s string) int {
  res := 0
  n := len(s)

  // 定义 DFS 递归函数
  var dfs func(index int, uniqueSet map[string]bool)
  dfs = func(index int, uniqueSet map[string]bool) {
    // 剪枝：剩余部分 + 当前已选的数量 < res，不可能更优
    if len(s)-index+len(uniqueSet) < res {
      return
    }

    // 成功拆分整个字符串
    if index >= n {
      if len(uniqueSet) > res {
        res = len(uniqueSet)
      }
      return
    }

    // 枚举所有可能子串
    for i := index + 1; i <= n; i++ {
      subStr := s[index:i]
      if uniqueSet[subStr] {
        continue // 重复子串跳过
      }
      uniqueSet[subStr] = true
      dfs(i, uniqueSet)
      delete(uniqueSet, subStr) // 回溯
    }
  }

  dfs(0, make(map[string]bool))
  return res
}
```

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

```
输入：s = "ababccc"
输出：5
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

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

示例3

```
输入：s = "aa"
输出：1
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++

```
class Solution {
public:
    int res = 0;
    // 当前开始下标
    void dfs(int index, string& s, unordered_set<string>& uniquSub) {
        int n = s.size();
        // 剪枝 不可能存在更多的数量
        if (n - index + uniquSub.size() < res) {
            return;
        }
        // 形成一个合法拆分方案
        if (index >= n) {
            res = max(res, (int)uniquSub.size());
            return;
        }
        for (int i = index; i < n; i++) {
            string subString = s.substr(index, i - index + 1);
            // 说明重复,直接跳过
            if (uniquSub.find(subString) != uniquSub.end()) {
                continue;
            }
            // 递归回溯
            uniquSub.insert(subString);
            dfs(i+1, s, uniquSub);
            uniquSub.erase(subString);
        }
    }

    int maxUniqueSplit(string s) {
        unordered_set<string> uniquSub;
        dfs(0, s, uniquSub);
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    int res = 0;

    public int maxUniqueSplit(String s) {
        Set<String> uniqueSub = new HashSet<>();
        dfs(0, s, uniqueSub);
        return res;
    }

    // 回溯函数，index 表示当前起始位置
    void dfs(int index, String s, Set<String> uniqueSub) {
        int n = s.length();
        // 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
        if (n - index + uniqueSub.size() < res) return;

        // 到达末尾，形成一个合法拆分方案
        if (index == n) {
            res = Math.max(res, uniqueSub.size());
            return;
        }

        for (int i = index; i < n; i++) {
            String sub = s.substring(index, i + 1);
            if (uniqueSub.contains(sub)) continue;

            // 尝试添加并回溯
            uniqueSub.add(sub);
            dfs(i + 1, s, uniqueSub);
            uniqueSub.remove(sub); // 回溯删除
        }
    }
}
```

Python

Plain Text

```
class Solution:
    def maxUniqueSplit(self, s: str) -> int:
        self.res = 0

        def dfs(index: int, unique_sub: set[str]):
            n = len(s)
            # 剪枝：如果剩余长度加当前集合大小小于 res，无需继续
            if n - index + len(unique_sub) < self.res:
                return

            # 到达末尾，更新结果
            if index == n:
                self.res = max(self.res, len(unique_sub))
                return

            for i in range(index, n):
                sub = s[index:i + 1]
                if sub in unique_sub:
                    continue
                unique_sub.add(sub)
                dfs(i + 1, unique_sub)
                unique_sub.remove(sub)  # 回溯

        dfs(0, set())
        return self.res
```

JavaScript

```
/**
 * @param {string} s
 * @return {number}
 */
var maxUniqueSplit = function(s) {
    let res = 0;

    function dfs(index, uniqueSub) {
        const n = s.length;
        // 剪枝优化
        if (n - index + uniqueSub.size < res) return;

        // 到达末尾
        if (index === n) {
            res = Math.max(res, uniqueSub.size);
            return;
        }

        for (let i = index; i < n; i++) {
            const sub = s.substring(index, i + 1);
            if (uniqueSub.has(sub)) continue;

            // 添加并回溯
            uniqueSub.add(sub);
            dfs(i + 1, uniqueSub);
            uniqueSub.delete(sub); // 回溯
        }
    }

    dfs(0, new Set());
    return res;
};
```

Go

Plain Text

```
var maxUniqueSplit = func(s string) int {
  res := 0
  n := len(s)

  // 定义 DFS 递归函数
  var dfs func(index int, uniqueSet map[string]bool)
  dfs = func(index int, uniqueSet map[string]bool) {
    // 剪枝：剩余部分 + 当前已选的数量 < res，不可能更优
    if len(s)-index+len(uniqueSet) < res {
      return
    }

    // 成功拆分整个字符串
    if index >= n {
      if len(uniqueSet) > res {
        res = len(uniqueSet)
      }
      return
    }

    // 枚举所有可能子串
    for i := index + 1; i <= n; i++ {
      subStr := s[index:i]
      if uniqueSet[subStr] {
        continue // 重复子串跳过
      }
      uniqueSet[subStr] = true
      dfs(i, uniqueSet)
      delete(uniqueSet, subStr) // 回溯
    }
  }

  dfs(0, make(map[string]bool))
  return res
}
```

> 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

```
输入: nums = [10,2]
输出: "210"
```

示例2

```
输入: nums = [3,30,34,5,9]
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++

```
class Solution {
public:
    string largestNumber(vector<int>& nums) {
        int n = nums.size();
        vector<string> numsStr(n);
        for (int i = 0; i < n; i++) {
            numsStr[i] = to_string(nums[i]);
        }
        // 自定义排序
        sort(numsStr.begin(), numsStr.end(),[](const string &a, const string &b) {
            return a + b > b + a;
        });

        if (numsStr[0] == "0") {
            return "0";
        }
        string res = "";
        for (int i = 0; i < n; i++) {
            res += numsStr[i];
        }
        // 移除
        return res;
    }
};
```

JAVA

```
import java.util.*;

class Solution {
    public String largestNumber(int[] nums) {
        int n = nums.length;
        String[] numsStr = new String[n];
        for (int i = 0; i < n; i++) {
            numsStr[i] = String.valueOf(nums[i]);
        }

        // 自定义排序
        Arrays.sort(numsStr, (a, b) -> (b + a).compareTo(a + b));

        // 特殊情况：全是0
        if (numsStr[0].equals("0")) {
            return "0";
        }

        // 拼接结果
        StringBuilder res = new StringBuilder();
        for (String num : numsStr) {
            res.append(num);
        }

        return res.toString();
    }
}
```

Python

```
class Solution:
    def largestNumber(self, nums):
        # 自定义排序函数
        def compare(a, b):
            if a + b > b + a:
                return -1
            elif a + b < b + a:
                return 1
            else:
                return 0

        numsStr = list(map(str, nums))
        numsStr.sort(key=cmp_to_key(compare))

        # 特殊情况：全是0
        if numsStr[0] == "0":
            return "0"

        # 拼接结果
        return ''.join(numsStr)
```

JavaScript

Plain Text

```
/**
 * @param {number[]} nums
 * @return {string}
 */
var largestNumber = function(nums) {
    // 将数字转为字符串
    let numsStr = nums.map(String);

    // 自定义排序
    numsStr.sort((a, b) => (b + a).localeCompare(a + b));

    // 特殊情况：全是0
    if (numsStr[0] === "0") {
        return "0";
    }

    // 拼接结果
    return numsStr.join('');
};
```

Go

```
Plain Text
```

```
func largestNumber(nums []int) string {
  n := len(nums)
  numsStr := make([]string, n)
  for i := 0; i < n; i++ {
    numsStr[i] = strconv.Itoa(nums[i])
  }

  // 自定义排序
  sort.Slice(numsStr, func(i, j int) bool {
    return numsStr[i]+numsStr[j] > numsStr[j]+numsStr[i]
  })

  // 特殊情况：全是0
  if numsStr[0] == "0" {
    return "0"
  }

  // 拼接结果
  return strings.Join(numsStr, "")
}
```

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
```

```
输入：nums = [10,2]
输出："210"
```

示例2

```
Plain Text
```

```
输入：nums = [3,30,34,5,9]
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++

Plain Text

```
class Solution {
public:
    string largestNumber(vector<int>& nums) {
        int n = nums.size();
        vector<string> numsStr(n);
        for (int i = 0; i < n; i++) {
            numsStr[i] = to_string(nums[i]);
        }
        // 自定义排序
        sort(numsStr.begin(), numsStr.end(),[](const string &a, const string &b) {
            return a + b > b + a;
        });

        if (numsStr[0] == "0") {
            return "0";
        }
        string res = "";
        for (int i = 0; i < n; i++) {
            res += numsStr[i];
        }
        // 移除
        return res;
    }
};
```

JAVA

```
import java.util.*;

class Solution {
    public String largestNumber(int[] nums) {
        int n = nums.length;
        String[] numsStr = new String[n];
        for (int i = 0; i < n; i++) {
            numsStr[i] = String.valueOf(nums[i]);
        }

        // 自定义排序
        Arrays.sort(numsStr, (a, b) -> (b + a).compareTo(a + b));

        // 特殊情况：全是0
        if (numsStr[0].equals("0")) {
            return "0";
        }

        // 拼接结果
        StringBuilder res = new StringBuilder();
        for (String num : numsStr) {
            res.append(num);
        }

        return res.toString();
    }
}
```

Python

```
class Solution:
    def largestNumber(self, nums):
        # 自定义排序函数
        def compare(a, b):
            if a + b > b + a:
                return -1
            elif a + b < b + a:
                return 1
            else:
                return 0

        numsStr = list(map(str, nums))
        numsStr.sort(key=cmp_to_key(compare))

        # 特殊情况：全是0
        if numsStr[0] == "0":
            return "0"

        # 拼接结果
        return ''.join(numsStr)
```

JavaScript

Plain Text

```
/**
 * @param {number[]} nums
 * @return {string}
 */
var largestNumber = function(nums) {
    // 将数字转为字符串
    let numsStr = nums.map(String);

    // 自定义排序
    numsStr.sort((a, b) => (b + a).localeCompare(a + b));

    // 特殊情况：全是0
    if (numsStr[0] === "0") {
        return "0";
    }

    // 拼接结果
    return numsStr.join('');
};
```

Go

```
func largestNumber(nums []int) string {
  n := len(nums)
  numsStr := make([]string, n)
  for i := 0; i < n; i++ {
    numsStr[i] = strconv.Itoa(nums[i])
  }

  // 自定义排序
  sort.Slice(numsStr, func(i, j int) bool {
    return numsStr[i]+numsStr[j] > numsStr[j]+numsStr[i]
  })

  // 特殊情况：全是0
  if numsStr[0] == "0" {
    return "0"
  }

  // 拼接结果
  return strings.Join(numsStr, "")
}
```

> 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

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

示例2

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

示例3

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

示例4

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

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++

Plain Text

```
class Solution {
  public:
    string decodeString(string s) {
        // 使用数组模拟栈
        vector<int> numberStack;
        // 使用数组模拟栈
        vector<string> textStack;
        // 守卫字符串防止越界
        textStack.push_back("");
        string currentText = "";
        string currentNnumber = "";
        int n = s.size();

        for (int i = 0; i < n; i++) {
            char c = s[i];
            // 数字字符
            if ('0' <= c && '9' >= c) {
                if (!currentText.empty()) {
                    textStack[textStack.size() - 1] += currentText;
                    currentText.clear();
                }
                currentNnumber.push_back(c);
            // 左括号 代表新的嵌套出现
            } else if (c == '[') {
                // 新嵌套出现
                textStack.push_back("");
                // 存储数字
                numberStack.push_back(stoi(currentNnumber));
                currentNnumber.clear();
            // 执行重复操作
            } else if (c == ']') {
                // 发生在连续第一个]执行
                if (!currentText.empty()) {
                    textStack[textStack.size() - 1] += currentText;
                    currentText.clear();
                }

                string repeateStrSub = textStack.back();
                textStack.pop_back();
                // 重复字符串
                string repeateStr;
                for (int j = 1; j <= numberStack.back(); j++) {
                    repeateStr += repeateStrSub;
                }
                numberStack.pop_back();
```

```
                textStack[textStack.size() - 1] += repeateStr;

            } else {
                currentText.push_back(c);
            }
        }
        // 收尾操作
        if (!currentText.empty()) {
            textStack.push_back(currentText);
        }

        //  拼接结果
        string res = "";
        for (int i = 0; i < textStack.size(); i++) {
            res += textStack[i];
        }
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    public String decodeString(String s) {
        // 使用栈模拟嵌套
        Stack<Integer> numberStack = new Stack<>();
        Stack<StringBuilder> textStack = new Stack<>();
        textStack.push(new StringBuilder());

        StringBuilder currentText = new StringBuilder();
        StringBuilder currentNumber = new StringBuilder();

        for (char c : s.toCharArray()) {
            if (Character.isDigit(c)) {
                // 当前是数字
                if (currentText.length() > 0) {
                    textStack.peek().append(currentText);
                    currentText.setLength(0);
                }
                currentNumber.append(c);
            } else if (c == '[') {
                // 开启新嵌套
                numberStack.push(Integer.parseInt(currentNumber.toString
()));
                currentNumber.setLength(0);
                textStack.push(new StringBuilder());
            } else if (c == ']') {
                // 嵌套结束，进行重复
                if (currentText.length() > 0) {
                    textStack.peek().append(currentText);
                    currentText.setLength(0);
                }
                String repeatStr = textStack.pop().toString();
                int count = numberStack.pop();
                StringBuilder repeated = new StringBuilder();
                for (int i = 0; i < count; i++) {
                    repeated.append(repeatStr);
                }
                textStack.peek().append(repeated);
            } else {
                // 普通字符
                currentText.append(c);
            }
        }
        // 收尾操作
        if (currentText.length() > 0) {
            textStack.push(currentText);
```

```
            }
            // 构建结果
            StringBuilder result = new StringBuilder();
            for (StringBuilder sb : textStack) {
                result.append(sb);
            }
            return result.toString();
        }
    }
```

Python

Plain Text

```
class Solution:
    def decodeString(self, s: str) -> str:
        number_stack = []
        text_stack = [""]
        current_text = ""
        current_number = ""

        for c in s:
            if c.isdigit():
                # 数字字符
                if current_text:
                    text_stack[-1] += current_text
                    current_text = ""
                current_number += c
            elif c == '[':
                # 开启新嵌套
                text_stack.append("")
                number_stack.append(int(current_number))
                current_number = ""
            elif c == ']':
                # 嵌套结束
                if current_text:
                    text_stack[-1] += current_text
                    current_text = ""
                repeat_str = text_stack.pop()
                count = number_stack.pop()
                text_stack[-1] += repeat_str * count
            else:
                # 普通字符
                current_text += c

        if current_text:
            text_stack.append(current_text)

        return ''.join(text_stack)
```

JavaScript

Plain Text

```
/**
 * @param {string} s
 * @return {string}
 */
function decodeString(s) {
    const numberStack = [];
    const textStack = [""];
    let currentText = "";
    let currentNumber = "";

    for (let c of s) {
        if (/\d/.test(c)) {
            // 数字字符
            if (currentText !== "") {
                textStack[textStack.length - 1] += currentText;
                currentText = "";
            }
            currentNumber += c;
        } else if (c === '[') {
            // 开启新嵌套
            textStack.push("");
            numberStack.push(parseInt(currentNumber));
            currentNumber = "";
        } else if (c === ']') {
            // 嵌套结束
            if (currentText !== "") {
                textStack[textStack.length - 1] += currentText;
                currentText = "";
            }
            const str = textStack.pop();
            const count = numberStack.pop();
            textStack[textStack.length - 1] += str.repeat(count);
        } else {
            // 普通字符
            currentText += c;
        }
    }

    if (currentText !== "") {
        textStack.push(currentText);
    }

    return textStack.join("");
}
```

Go

Plain Text

```
func decodeString(s string) string {
    numberStack := []int{}
    textStack := []string{""}
    currentText := ""
    currentNumber := ""

    for _, c := range s {
        ch := string(c)
        if ch >= "0" && ch <= "9" {
            // 数字字符
            if currentText != "" {
                textStack[len(textStack)-1] += currentText
                currentText = ""
            }
            currentNumber += ch
        } else if ch == "[" {
            // 开启新嵌套
            textStack = append(textStack, "")
            num, _ := strconv.Atoi(currentNumber)
            numberStack = append(numberStack, num)
            currentNumber = ""
        } else if ch == "]" {
            // 嵌套结束
            if currentText != "" {
                textStack[len(textStack)-1] += currentText
                currentText = ""
            }
            repeatStr := textStack[len(textStack)-1]
            textStack = textStack[:len(textStack)-1]
            count := numberStack[len(numberStack)-1]
            numberStack = numberStack[:len(numberStack)-1]
            repeated := strings.Repeat(repeatStr, count)
            textStack[len(textStack)-1] += repeated
        } else {
            // 普通字符
            currentText += ch
        }
    }

    if currentText != "" {
        textStack = append(textStack, currentText)
    }

    return strings.Join(textStack, "")
}
```

> 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

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

示例2

Plain Text

```
输入：nums = [0,1,0,3,2,3]
输出：4
```

示例3

Plain Text

```
输入：nums = [7,7,7,7,7,7,7]
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++

Plain Text

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        // 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
        vector<int> dp;
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            // 说明长度可以加一
            if (dp.empty() || dp.back() < nums[i]) {
                dp.push_back(nums[i]);
            } else {
                // 替换指定对应长度 末尾最小值
                auto it = lower_bound(dp.begin(), dp.end(), nums[i]);
                *it = nums[i];
            }
        }
        return dp.size();
    }
};
```

JAVA

Plain Text

```
import java.util.*;

class Solution {
    public int lengthOfLIS(int[] nums) {
        // 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
        List<Integer> dp = new ArrayList<>();
        for (int num : nums) {
            // 说明长度可以加一
            if (dp.isEmpty() || dp.get(dp.size() - 1) < num) {
                dp.add(num);
            } else {
                // 替换对应长度的末尾最小值
                int left = 0, right = dp.size() - 1;
                while (left < right) {
                    int mid = (left + right) / 2;
                    if (dp.get(mid) < num) {
                        left = mid + 1;
                    } else {
                        right = mid;
                    }
                }
                dp.set(left, num);
            }
        }
        return dp.size();
    }
}
```

Python

```
import bisect

class Solution:
    def lengthOfLIS(self, nums: list[int]) -> int:
        # 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
        dp = []
        for num in nums:
            # 说明长度可以加一
            if not dp or dp[-1] < num:
                dp.append(num)
            else:
                # 替换对应长度的末尾最小值
                index = bisect.bisect_left(dp, num)
                dp[index] = num
        return len(dp)
```

JavaScript

Plain Text

```
/**
 * @param {number[]} nums
 * @return {number}
 */
function lengthOfLIS(nums) {
    // 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
    const dp = [];
    for (let num of nums) {
        // 说明长度可以加一
        if (dp.length === 0 || dp[dp.length - 1] < num) {
            dp.push(num);
        } else {
            // 替换对应长度的末尾最小值
            let left = 0, right = dp.length - 1;
            while (left < right) {
                let mid = Math.floor((left + right) / 2);
                if (dp[mid] < num) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            }
            dp[left] = num;
        }
    }
    return dp.length;
}
```

Go

Plain Text

```
func lengthOfLIS(nums []int) int {
  // 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
  dp := []int{}
  for _, num := range nums {
    // 说明长度可以加一
    if len(dp) == 0 || dp[len(dp)-1] < num {
      dp = append(dp, num)
    } else {
      // 替换对应长度的末尾最小值
      idx := sort.Search(len(dp), func(i int) bool { return dp[i] >= num
})
      dp[idx] = num
    }
  }
  return len(dp)
}
```

> 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

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

```
输入：numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
输出：[0,2,1,3]
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。
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

```
输入：numCourses = 1, prerequisites = []
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 < = 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++

Plain Text

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        // 记录每个课程的入度
        map<int,int> inOrderCount;
        // 初始化入度为0
        for (int i = 0; i < numCourses; i++) {
            inOrderCount[i] = 0;
        }

        // 记录依赖关系
        map<int, set<int>> depend;
        int n = prerequisites.size();
        for (int i = 0; i < n; i++) {
            int second = prerequisites[i][0];
            int first = prerequisites[i][1];
            inOrderCount[second]++;
            depend[first].insert(second);
        }
        // 处理拓扑排序
        queue<int> q;
        // 初始添加入度为0的课程
        for (auto &p : inOrderCount) {
            if (p.second == 0) {
                q.push(p.first);
            }
        }
        // 存储结果
        vector<int> res(numCourses);

        // 已访问课程数量
        int count = 0;
        while (!q.empty()) {
            int top = q.front();
            q.pop();
            count++;
            if (count > numCourses) {
                break;
            }
            res[count - 1] = top;

            // 更新依赖它的课程入度
            for (auto id : depend[top]) {
                inOrderCount[id]--;
```

```
                if (inOrderCount[id] <= 0) {
                    q.push(id);
                }
            }
        }
        if (count != numCourses) {
            return {};
        }
        return res;
    }
};
```

JAVA

Plain Text

```
class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        // 记录每个课程的入度
        int[] inOrderCount = new int[numCourses];
        // 记录依赖关系
        Map<Integer, List<Integer>> depend = new HashMap<>();

        for (int[] pre : prerequisites) {
            int second = pre[0];
            int first = pre[1];
            inOrderCount[second]++;
            depend.computeIfAbsent(first, k -> new ArrayList<>()).add(seco
nd);
        }

        // 处理拓扑排序
        Queue<Integer> q = new LinkedList<>();
        // 初始添加入度为0的课程
        for (int i = 0; i < numCourses; i++) {
            if (inOrderCount[i] == 0) {
                q.offer(i);
            }
        }

        int[] res = new int[numCourses];
        int count = 0;

        while (!q.isEmpty()) {
            int top = q.poll();
            res[count++] = top;

            if (depend.containsKey(top)) {
                for (int id : depend.get(top)) {
                    inOrderCount[id]--;
                    if (inOrderCount[id] == 0) {
                        q.offer(id);
                    }
                }
            }
        }

        if (count != numCourses) return new int[0];
        return res;
    }
}
```

Python

```
Plain Text
```

```
from collections import deque, defaultdict

class Solution:
    def findOrder(self, numCourses: int, prerequisites: list[list[int]]) -> list[int]:
        # 记录每个课程的入度
        in_order_count = [0] * numCourses
        # 记录依赖关系
        depend = defaultdict(list)

        for second, first in prerequisites:
            in_order_count[second] += 1
            depend[first].append(second)

        # 初始添加入度为0的课程
        q = deque()
        for i in range(numCourses):
            if in_order_count[i] == 0:
                q.append(i)

        res = []
        count = 0

        while q:
            top = q.popleft()
            res.append(top)
            count += 1
            for neighbor in depend[top]:
                in_order_count[neighbor] -= 1
                if in_order_count[neighbor] == 0:
                    q.append(neighbor)

        return res if count == numCourses else []
```

JavaScript

Plain Text

```
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {number[]}
 */
var findOrder = function(numCourses, prerequisites) {
    // 记录每个课程的入度
    const inOrderCount = Array(numCourses).fill(0);
    // 记录依赖关系
    const depend = new Map();

    for (const [second, first] of prerequisites) {
        inOrderCount[second]++;
        if (!depend.has(first)) depend.set(first, []);
        depend.get(first).push(second);
    }

    // 初始添加入度为0的课程
    const queue = [];
    for (let i = 0; i < numCourses; i++) {
        if (inOrderCount[i] === 0) queue.push(i);
    }

    const res = [];
    let count = 0;

    while (queue.length > 0) {
        const top = queue.shift();
        res.push(top);
        count++;
        if (depend.has(top)) {
            for (const neighbor of depend.get(top)) {
                inOrderCount[neighbor]--;
                if (inOrderCount[neighbor] === 0) {
                    queue.push(neighbor);
                }
            }
        }
    }

    return count === numCourses ? res : [];
};
```

Go

Plain Text

```
func findOrder(numCourses int, prerequisites [][]int) []int {
    // 记录每个课程的入度
    inOrderCount := make([]int, numCourses)
    // 记录依赖关系
    depend := make(map[int][]int)

    for _, pre := range prerequisites {
        second := pre[0]
        first := pre[1]
        inOrderCount[second]++
        depend[first] = append(depend[first], second)
    }

    // 初始添加入度为0的课程
    queue := []int{}
    for i := 0; i < numCourses; i++ {
        if inOrderCount[i] == 0 {
            queue = append(queue, i)
        }
    }

    res := []int{}
    count := 0

    for len(queue) > 0 {
        top := queue[0]
        queue = queue[1:]
        res = append(res, top)
        count++
        for _, id := range depend[top] {
            inOrderCount[id]--
            if inOrderCount[id] == 0 {
                queue = append(queue, id)
            }
        }
    }

    if count != numCourses {
        return []int{}
    }
    return res
}
```

> 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

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

用例2

Plain Text

```
输入：nums = [0]
输出：[[],[0]]
```

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++

```
Plain Text
```

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        // 定义结果数组
        vector<vector<int>> res;
        res.push_back({});
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            int currentLen = res.size();
            // 全部添加当前值
            for (int j = 0; j < currentLen; j++) {
                vector<int> tmp = res[j];
                tmp.push_back(nums[i]);
                res.push_back(tmp);
            }
        }
        return res;
    }
};
```

JAVA

```
Plain Text
```

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        // 定义结果数组
        List<List<Integer>> res = new ArrayList<>();
        res.add(new ArrayList<>());
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            int currentLen = res.size();
            // 全部添加当前值
            for (int j = 0; j < currentLen; j++) {
                List<Integer> tmp = new ArrayList<>(res.get(j));
                tmp.add(nums[i]);
                res.add(tmp);
            }
        }
        return res;
    }
}
```

Python

```
Plain Text
```

```python
class Solution:
    def subsets(self, nums):
        # 定义结果数组
        res = [[]]
        n = len(nums)
        for i in range(n):
            current_len = len(res)
            # 全部添加当前值
            for j in range(current_len):
                tmp = res[j][:]  # 拷贝当前子集
                tmp.append(nums[i])
                res.append(tmp)
        return res
```

JavaScript

```
Plain Text
```

```javascript
var subsets = function(nums) {
    // 定义结果数组
    let res = [[]];
    let n = nums.length;
    for (let i = 0; i < n; i++) {
        let currentLen = res.length;
        // 全部添加当前值
        for (let j = 0; j < currentLen; j++) {
            let tmp = res[j].slice();  // 拷贝当前子集
            tmp.push(nums[i]);
            res.push(tmp);
        }
    }
    return res;
};
```

Go

Plain Text

```
func subsets(nums []int) [][]int {
    // 定义结果数组
    res := [][]int{{}}
    n := len(nums)
    for i := 0; i < n; i++ {
        currentLen := len(res)
        // 全部添加当前值
        for j := 0; j < currentLen; j++ {
            tmp := make([]int, len(res[j]))
            copy(tmp, res[j])
            tmp = append(tmp, nums[i])
            res = append(res, tmp)
        }
    }
    return res
}
```

> 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<=n<=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` 其中需要满足 `pi^k <= n` 。简单举个例子说明 `最大幂乘积` 原理

Plain Text

```
假设当前有这些数：
4 = 2^2
6 = 2 * 3
9 = 3 ^ 2
为了能整除4 你需要至少有 2^2
为了能整除6， 你至少需要有 2^1 和 3 ^1
为了能整除9， 你至少有3^2
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++

Plain Text

```
#include<iostream>
#include<vector>
using namespace std;

// 质数筛
vector<bool> sieve(int n ) {
    vector<bool> isPrime(n + 1, true);
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            // 把质数的所有倍数全部标记非质数
            for (int j = i * i; j <= n; j+= i) {
                isPrime[j] = false;
            }
        }
    }
    return isPrime;
}


int main() {
    int n ;
    cin >> n;
    vector<bool> isPrime = sieve(n);
    vector<int> primeNum;
    for (int i = 1; i <= n; i++) {
        if (isPrime[i]) {
            primeNum.push_back(i);
        }
    }
    // 防止溢出
    unsigned long long res = 1;

    // 最大幂定理
    for (int i = 0; i < primeNum.size(); i++) {
        int tmp = primeNum[i];
        while (tmp * primeNum[i] <= n) {
            tmp *= primeNum[i];
        }
        res =  (res *tmp) % 1000000007;
    }

    cout << res;
    return 0;
}
```

JAVA

Plain Text

```
import java.util.*;

public class Main {
    static final int MOD = 1000000007;

    // 质数筛
    static boolean[] sieve(int n) {
        boolean[] isPrime = new boolean[n + 1];
        Arrays.fill(isPrime, true);
        isPrime[0] = isPrime[1] = false;
        for (int i = 2; i * i <= n; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }
        return isPrime;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        boolean[] isPrime = sieve(n);
        List<Integer> primeNum = new ArrayList<>();
        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                primeNum.add(i);
            }
        }

        long res = 1;
        for (int p : primeNum) {
            long tmp = p;
            while (tmp * p <= n) {
                tmp *= p;
            }
            res = (res * tmp) % MOD;
        }

        System.out.println(res);
    }
}
```

Python

```
MOD = 1000000007

# 质数筛
def sieve(n):
    is_prime = [True] * (n + 1)
    is_prime[0:2] = [False, False]
    for i in range(2, int(n ** 0.5) + 1):
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    return is_prime

n = int(input())
is_prime = sieve(n)
prime_list = [i for i, val in enumerate(is_prime) if val]

res = 1
for p in prime_list:
    tmp = p
    while tmp * p <= n:
        tmp *= p
    res = (res * tmp) % MOD

print(res)
```

JavaScript

Plain Text

```
const readline = require('readline');
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });

const MOD = 1000000007n;

rl.on('line', (line) => {
    let n = parseInt(line);
    let isPrime = Array(n + 1).fill(true);
    isPrime[0] = isPrime[1] = false;

    // 质数筛
    for (let i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            for (let j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    let res = 1n;
    for (let i = 2; i <= n; i++) {
        if (isPrime[i]) {
            let tmp = BigInt(i);
            while (tmp * BigInt(i) <= BigInt(n)) {
                tmp *= BigInt(i);
            }
            res = (res * tmp) % MOD;
        }
    }

    console.log(res.toString());
    rl.close();
});
```

Go

Plain Text

```
package main

import (
    "fmt"
    "math/big"
)

const MOD = 1000000007

// 质数筛
func sieve(n int) []bool {
    isPrime := make([]bool, n+1)
    for i := range isPrime {
        isPrime[i] = true
    }
    isPrime[0], isPrime[1] = false, false
    for i := 2; i*i <= n; i++ {
        if isPrime[i] {
            for j := i * i; j <= n; j += i {
                isPrime[j] = false
            }
        }
    }
    return isPrime
}

func main() {
    var n int
    fmt.Scan(&n)
    isPrime := sieve(n)

    res := big.NewInt(1)
    mod := big.NewInt(MOD)

    for i := 2; i <= n; i++ {
        if isPrime[i] {
            tmp := big.NewInt(int64(i))
            for {
                mul := new(big.Int).Mul(tmp, big.NewInt(int64(i)))
                if mul.Cmp(big.NewInt(int64(n))) > 0 {
                    break
                }
                tmp = mul
            }
            res.Mul(res, tmp)
```

```
                res.Mod(res, mod)
            }
        }

        fmt.Println(res.String())
    }
```

> 来自: 求1–n的最小公倍数–CSDN博客