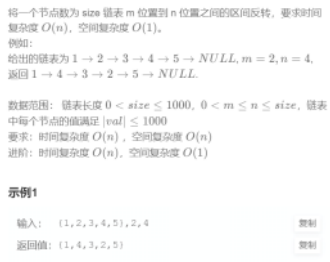
面试必刷Top101

2022年5月9日 21:27

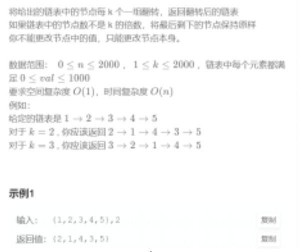
- BM2 链表内指定区间反转
  - 找区间+反转
  - 

    将一个节点数为 size 链表 m 位置到 n 位置之间的区间反转，要求时间复杂度 $O(n)$，空间复杂度 $O(1)$。
    例如：
    给出的链表为 $1 \to 2 \to 3 \to 4 \to 5 \to NULL, m = 2, n = 4$,
    返回 $1 \to 4 \to 3 \to 2 \to 5 \to NULL$.

    数据范围：链表长度 $0 < size \leq 1000$，$0 < m \leq n \leq size$，链表中每个节点的值满足 $|val| \leq 1000$
    要求：时间复杂度 $O(n)$，空间复杂度 $O(n)$
    进阶：时间复杂度 $O(n)$，空间复杂度 $O(1)$

    示例1

    输入： {1,2,3,4,5},2,4
    返回值： {1,4,3,2,5}

  -
    ```
    class Solution:
        def reverseBetween(self , head: ListNode, m: int, n: int) -> ListNode:
            # write code here
            if not head or not head.next:
                return head
            dummy = ListNode(0)
            dummy.next = head
            pre = dummy
            start = head
            end = head
            for _ in range(1, m):
                pre = pre.next
                start = start.next
            for _ in range(1, n):
                end = end.next
            temp = end.next
            start, end = self.reverses(start, end)
            pre.next = start
            end.next = temp
            return dummy.next
        def reverses(self, start, end):
            prev = None
            cur = start
            while prev != end:
                temp = cur.next
                cur.next = prev
                prev = cur
                cur = temp
            return end, start
    ```
- BM3 链表中的节点每k个一组翻转
  - 找区间+整个反转
  - 

    将给出的链表中的节点每 k 个一组翻转，返回翻转后的链表
    如果链表中的节点数不是 k 的倍数，将最后剩下的节点保持原样
    你不能更改节点中的值，只能更改节点本身。

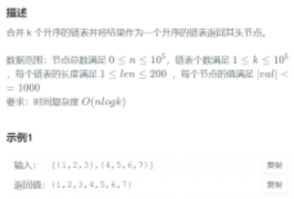
    数据范围：$0 \leq n \leq 2000$，$1 \leq k \leq 2000$，链表中每个元素都满足 $0 \leq val \leq 1000$
    要求空间复杂度 $O(1)$，时间复杂度 $O(n)$
    例如：
    给定的链表是 $1 \to 2 \to 3 \to 4 \to 5$
    对于 $k = 2$，你应该返回 $2 \to 1 \to 4 \to 3 \to 5$
    对于 $k = 3$，你应该返回 $3 \to 2 \to 1 \to 4 \to 5$

    示例1

    输入： {1,2,3,4,5},2
    返回值： {2,1,4,3,5}

  -
    ```
    class Solution:
        def reverseKGroup(self , head: ListNode, k: int) -> ListNode:
            # write code here
            if not head or not head.next:
                return head
            dummy = ListNode(0)
            dummy.next = head
            prev = end = dummy
            start = head
            while start:
                for _ in range(k):
                    end = end.next
                    if not end:
                        return dummy.next
                temp = end.next
                start, end = self.reverses(start, end)
                prev.next = start
                end.next = temp
                start = temp
                prev = end
            return dummy.next
        def reverses(self, start, end):
    ```

```
        prev = None
        cur = start
        while prev != end:
            temp = cur.next
            cur.next = prev
            prev = cur
            cur = temp
        return end, start
```

- BM5 合并k个已排序的链表
  - 堆里每次保留k个元素
  - 
    描述

    合并 k 个升序的链表并将结果作为一个升序的链表返回其头节点。

    数据范围：节点总数满足 $0 \leq n \leq 10^5$，链表个数满足 $1 \leq k \leq 10^5$，每个链表的长度满足 $1 \leq len \leq 200$ ，每个节点的值满足 $|val| <= 1000$

    要求：时间复杂度 $O(nlogk)$

    示例1

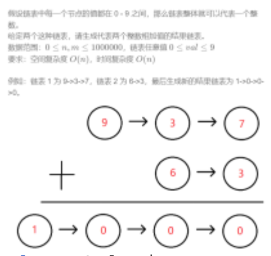
    输入：[{1,2,3},{4,5,6,7}] 复制

    返回值：{1,2,3,4,5,6,7} 复制

  -
    ```
    class Solution:
        def mergeKLists(self , lists: List[ListNode]) -> ListNode:
            # write code here
            import heapq
            dummy = ListNode(0)
            p = dummy
            head = []
            for i in range(len(lists)):
                if lists[i]:
                    heapq.heappush(head, (lists[i].val, i))
                    lists[i] = lists[i].next
            while head:
                temp, idx = heapq.heappop(head)
                p.next = ListNode(temp)
                p = p.next
                if lists[idx]:
                    heapq.heappush(head, (lists[idx].val, idx))
                    lists[idx] = lists[idx].next
            return dummy.next
    ```
- BM7 链表中环的入口结点
  - 快慢指针

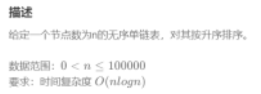
  -
    ```
    class Solution:
        def EntryNodeOfLoop(self, pHead):
            # write code here
            if not pHead or not pHead.next:
                return
            slow = fast = pHead
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
                if slow == fast:
                    break
            if not fast or not fast.next:
                return
            slow = pHead
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow
    ```
- BM9 删除链表的倒数第n个节点
  - 快慢指针

  -
    ```
    class Solution:
        def removeNthFromEnd(self, head, n):
            # write code here
            if not head:
                return
            dummy = ListNode(0)
            dummy.next = head
            slow = fast = dummy
            for _ in range(n):
                fast = fast.next
            while fast and fast.next:
                slow = slow.next
                fast = fast.next
            slow.next = slow.next.next
            return dummy.next
    ```
- BM11 链表相加（二）
  - 链表反转+逐位相加+原地相加

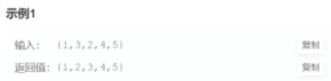
- 
  假设链表中每一个节点的值都在 0 - 9 之间，那么链表整体就可以代表一个整数。
  给定两个这种链表，请生成代表两个整数相加值的结果链表。
  数据范围：$0 \le n, m \le 1000000$，链表任意值 $0 \le val \le 9$
  要求：空间复杂度 $O(n)$，时间复杂度 $O(n)$

  例如：链表 1 为 9->3->7，链表 2 为 6->3，最后生成新的结果链表为 1->0->0->0。

-
  ```python
class Solution:
    def addInList(self , head1 , head2 ):
        # write code here
        if not head1:
            return head2
        if not head2:
            return head1
        head1 = self.reverses(head1)
        head2 = self.reverses(head2)
        dummy = ListNode(0)
        dummy.next = head1
        h1 = head1
        h2 = head2
        c = 0
        while h1 and h2:
            temp = h1.val + h2.val + c
            c = temp // 10
            h1.val = temp % 10
            tail = h1
            h1 = h1.next
            h2 = h2.next
        if h1:
            while c > 0 and h1:
                temp = h1.val + c
                h1.val = temp % 10
                c = temp // 10
                tail = h1
                h1 = h1.next
        if h2:
            tail.next = h2
            while c > 0 and h2:
                temp = h2.val + c
                h2.val = temp % 10
                c = temp // 10
                tail = h2
                h2 = h2.next
        if c > 0:
            tail.next = ListNode(c)
        return self.reverses(dummy.next)
    def reverses(self, head):
        if not head or not head.next:
            return head
        pre = None
        while head:
            temp = head.next
            head.next = pre
            pre = head
            head = temp
        return pre
```

* BM12 单链表排序
  - 归并排序
  - 
    描述
    给定一个节点数为n的无序单链表，对其按升序排序。

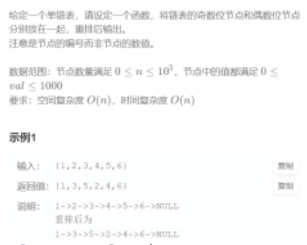
    数据范围：$0 < n \le 100000$
    要求：时间复杂度 $O(nlogn)$

    
    示例1
    输入： {1,3,2,4,5}　　复制
    返回值：{1,2,3,4,5}　　复制

  -
    ```python
class Solution:
    def sortInList(self , head ):
        # write code here
        if not head or not head.next:
            return head
        slow = fast = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        mid = slow.next
        slow.next = None
        left_list = self.sortInList(head)
        right_list = self.sortInList(mid)
        res = self.Merge(left_list, right_list)
```

```python
        return res
    def Merge(self, l1, l2):
        dummy = p = ListNode(0)
        while l1 and l2:
            if l1.val <= l2.val:
                p.next = l1
                l1 = l1.next
            else:
                p.next = l2
                l2 = l2.next
            p = p.next
        p.next = l1 if l1 else l2
        return dummy.next
```

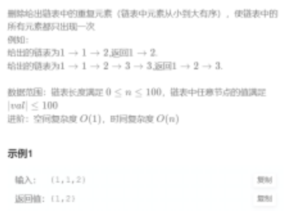
- BM13 判断一个链表是否是回文结构
  - 转换为列表+双指针

  -
    ```python
    class Solution:
        def isPail(self , head: ListNode) -> bool:
            # write code here
            if not head or not head.next:
                return True
            temp = []
            while head:
                temp.append(head.val)
                head = head.next
            left = 0
            right = len(temp) - 1
            while left <= right:
                if temp[left] == temp[right]:
                    left += 1
                    right -= 1
                else:
                    return False
            return True
    ```
- BM14 链表奇偶重排
  - 奇链表+偶链表+合并
  - 
    给定一个单链表，请设定一个函数，将链表的奇数位节点和偶数位节点分别放在一起，重排后输出。
    注意是节点的编号而非节点的数值。

    数据范围：节点数量满足 $0 \le n \le 10^5$，节点中的值都满足 $0 \le val \le 1000$
    要求：空间复杂度 $O(n)$，时间复杂度 $O(n)$

    示例1

    输入： {1,2,3,4,5,6} 复制
    返回值： {1,3,5,2,4,6} 复制
    说明： 1->2->3->4->5->6->NULL
    重排后为
    1->3->5->2->4->6->NULL

  -
    ```python
    class Solution:
        def oddEvenList(self , head: ListNode) -> ListNode:
            # write code here
            if not head or not head.next:
                return head
            evenHead = head.next
            odd = head
            even = evenHead
            while even and even.next:
                odd.next = even.next
                odd = odd.next
                even.next = odd.next
                even = even.next
            odd.next = evenHead
            return head
    ```
- BM15 删除链表重复元素-I
  - 保留一个：双指针移动+两个指针之间的比较
  - 
    删除给出链表中的重复元素（链表中元素从小到大有序），使链表中的所有元素都只出现一次
    例如：
    给出的链表为 $1 \to 1 \to 2$,返回 $1 \to 2$.
    给出的链表为 $1 \to 1 \to 2 \to 3 \to 3$,返回 $1 \to 2 \to 3$.

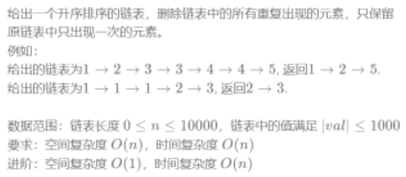
    数据范围：链表长度满足 $0 \le n \le 100$，链表中任意节点的值满足 $|val| \le 100$
    进阶：空间复杂度 $O(1)$，时间复杂度 $O(n)$

    示例1

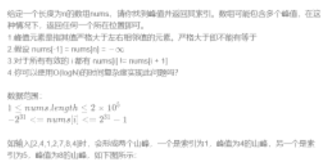
    输入： {1,1,2} 复制
    返回值： {1,2} 复制

  -
    ```python
    class Solution:
        def deleteDuplicates(self , head: ListNode) -> ListNode:
            # write code here
            if not head or not head.next:
                return head
            slow = head
            fast = head.next
            while fast:
                if slow.val == fast.val:
                    slow.next = fast.next
                    fast = fast.next
    ```

```
            else:
                slow = fast
                fast = fast.next
        return head
```

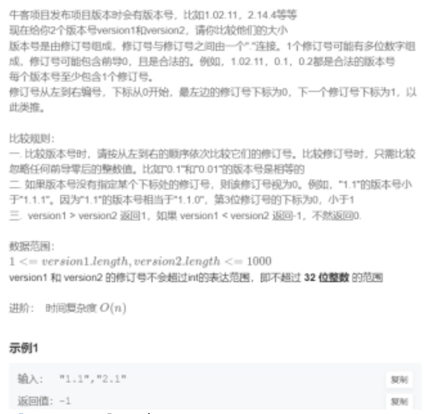
- BM16 删除链表重复元素-II
  - 重复全部删除：双指针+fast与fast.next比较
  - 

    给出一个升序排序的链表，删除链表中的所有重复出现的元素，只保留原链表中只出现一次的元素。
    例如：
    给出的链表为$1 \to 2 \to 3 \to 3 \to 4 \to 4 \to 5$, 返回$1 \to 2 \to 5$.
    给出的链表为$1 \to 1 \to 1 \to 2 \to 3$, 返回$2 \to 3$.

    数据范围：链表长度 $0 \le n \le 10000$，链表中的值满足 $|val| \le 1000$
    要求：空间复杂度 $O(n)$，时间复杂度 $O(n)$
    进阶：空间复杂度 $O(1)$，时间复杂度 $O(n)$
  -
```
class Solution:
    def deleteDuplicates(self , head: ListNode) -> ListNode:
        # write code here
        dummy = ListNode(0)
        dummy.next = head
        pre = dummy
        cur = head
        while cur and cur.next:
            if cur.val != cur.next.val:
                pre = cur
                cur = cur.next
            else:
                while cur.val == cur.next.val:
                    cur = cur.next
                    if not cur.next:
                        break
                pre.next = cur.next
                cur = cur.next
        return dummy.next
```
- BM19 寻找峰值
  - 二分查找
  - 

    给定一个长度为n的数组nums，请你找到峰值并返回其索引。数组可能包含多个峰值，在这种情况下，返回任何一个所在位置即可。
    1.峰值元素是指其值严格大于左右相邻值的元素。严格大于即不能有等于
    2.假设 nums[-1] = nums[n] = $-\infty$
    3.对于所有有效的 i 都有 nums[i] != nums[i + 1]
    4.你可以使用O(logN)的时间复杂度实现此问题吗?

    数据范围：
    $1 \le nums.length \le 2 \times 10^5$
    $-2^{31} <= nums[i] <= 2^{31} - 1$

    如输入[2,4,1,2,7,8,4]时，会形成两个山峰，一个是索引为1，峰值为4的山峰，另一个是索引为5，峰值为8的山峰，如下图所示：
  -
```
class Solution:
    def findPeakElement(self , nums: List[int]) -> int:
        # write code here
        left = 0
        right = len(nums) - 1
        while left < right:
            if nums[left] <= nums[right]:
                left += 1
            elif nums[right] < nums[left]:
                right -= 1
        return left
```
- BM22 比较版本号
  - 两个指针分别指向两个字符串同时逐位右移，碰到'.'停止表示一个数字，进行比较
  - 

    牛客项目发布项目版本时会有版本号，比如1.02.11，2.14.4等等
    现在给你2个版本号version1和version2，请你比较他们的大小
    版本号是由修订号组成，修订号与修订号之间由一个"."连接。1个修订号可能有多位数字组成，修订号可能包含前导0，且是合法的。例如，1.02.11，0.1，0.2都是合法的版本号
    每个版本号至少包含1个修订号。
    修订号从左到右编号，下标从0开始，最左边的修订号下标为0，下一个修订号下标为1，以此类推。

    比较规则：
    一. 比较版本号时，请按从左到右的顺序依次比较它们的修订号。比较修订号时，只需比较忽略任何前导零后的整数值。比如"0.1"和"0.01"的版本号是相等的
    二. 如果版本号没有指定某个下标处的修订号，则该修订号视为0。例如，"1.1"的版本号小于"1.1.1"。因为"1.1"的版本号相当于"1.1.0"，第3位修订号的下标为0，小于1
    三. version1 > version2 返回1，如果 version1 < version2 返回-1，不然返回0.

    数据范围：
    $1 <= version1.length, version2.length <= 1000$
    version1 和 version2 的修订号不会超过int的表达范围，即不超过 **32 位整数** 的范围

    进阶： 时间复杂度 $O(n)$

    示例1

    输入： "1.1","2.1" 复制
    返回值：-1 复制
  -
```
class Solution:
    def compare(self , version1: str, version2: str) -> int:
        # write code here
        p1 = 0
        p2 = 0
        while p1 < len(version1) or p2 < len(version2):
            num1 = 0
            while p1 < len(version1) and version1[p1] != '.':
                num1 = num1 * 10 + int(version1[p1])
                p1 += 1
            num2 = 0
            while p2 < len(version2) and version2[p2] != '.':
                num2 = num2 * 10 + int(version2[p2])
                p2 += 1
            if num1 > num2:
                return 1
            elif num1 < num2:
                return -1
            else:
                p1 += 1
                p2 += 1
        return 0
```

- BM23 二叉树前序遍历
  - 颜色标记法+stack

  -
    ```
    class Solution:
        def preorderTraversal(self , root: TreeNode) -> List[int]:
            # write code here
            if not root:
                return []
            res = []
            stack = [root]
            while stack:
                cur = stack.pop()
                if isinstance(cur, TreeNode):
                    stack.append(cur.right)
                    stack.append(cur.left)
                    stack.append(cur.val)
                if isinstance(cur, int):
                    res.append(cur)
            return res
    ```
- BM29 和为某一值的路径（一）
  - 递归

  -
    ```
    class Solution:
        def hasPathSum(self , root: TreeNode, sum: int) -> bool:
            # write code here
            if not root:
                return False
            if root.val == sum and not root.left and not root.right:
                return True
            return self.hasPathSum(root.left, sum-root.val) or self.hasPathSum(root.right, sum-root.val)
    ```
- BM32 合并二叉树

  -
    ```
    class Solution:
        def mergeTrees(self , t1: TreeNode, t2: TreeNode) -> TreeNode:
            # write code here
            if not t1:
                return t2
            if not t2:
                return t1
            queue = [(t1, t2)]
            while queue:
                cur1, cur2 = queue.pop(0)
                cur1.val = cur1.val + cur2.val
                if cur1.left and cur2.left:
                    queue.append((cur1.left, cur2.left))
                elif not cur1.left:
                    cur1.left = cur2.left
                if cur1.right and cur2.right:
                    queue.append((cur1.right, cur2.right))
                elif not cur1.right:
                    cur1.right = cur2.right
            return t1
    ```
- BM34 判断是否为二叉搜索树
  - 递归

  -
    ```
    class Solution:
        def isValidBST(self , root: TreeNode) -> bool:
            # write code here
            if not root:
                return True
            min_val = float('-inf')
            max_val = float('inf')
            def recur(cur, max_, min_):
                if not cur:
                    return True
                if not min_ < cur.val < max_:
                    return False
                else:
                    return recur(cur.left, cur.val, min_) and recur(cur.right, max_, cur.val)
            return recur(root.left, root.val, min_val) and recur(root.right, max_val, root.val)
    ```
- BM35 判断是否为完全二叉树

- BFS+判断None

-
  ```
  class Solution:
      def isCompleteTree(self , root: TreeNode) -> bool:
          # write code here
          if not root:
              return True
          queue = [root]
          temp = []
          while queue:
              cur = queue.pop(0)
              if cur:
                  temp.append(cur.val)
                  queue.append(cur.left)
                  queue.append(cur.right)
              else:
                  temp.append('#')
          while temp[-1] == '#':
              temp.pop()
          return not '#' in temp
  ```

- BM41 输出二叉树右视图
  - 重建二叉树+BFS

  -
    ```
    class TreeNode(object):
        def __init__(self, x):
            self.val = x
            self.left = None
            self.right = None

    class Solution:
        def solve(self , xianxu , zhongxu):
            # write code here
            root = self.restruction(xianxu, zhongxu)
            if not root:
                return []
            queue = [root]
            res = []
            while queue:
                len_q = len(queue)
                path = []
                for _ in range(len_q):
                    cur = queue.pop(0)
                    path.append(cur.val)
                    if cur.left:
                        queue.append(cur.left)
                    if cur.right:
                        queue.append(cur.right)
                if path:
                    res.append(path[-1])
            return res
        def restruction(self, pre, mid):
            if not pre or not mid or len(pre) != len(mid):
                return
            root = TreeNode(pre[0])
            idx = mid.index(root.val)

            left_pre = pre[1:idx+1]
            right_pre = pre[idx+1:]
            left_mid = mid[:idx]
            right_mid = mid[idx+1:]
            root.left = self.restruction(left_pre, left_mid)
            root.right = self.restruction(right_pre, right_mid)
            return root
    ```
  - 递归

  -
    ```
    class Solution:
        def solve(self , xianxu , zhongxu ):
            # write code here
            res = []
            def recur(pre, mid, level):
                if not pre:
                    return
                if level >= len(res):
                    res.append(pre[0])
                else:
                    res[level] = pre[0]
                temp = mid.index(pre[0])
                recur(pre[1:temp+1], mid[:temp], level+1)
                recur(pre[temp+1:], mid[temp+1:], level+1)
            recur(xianxu, zhongxu, 0)
            return res
    ```
- BM44 有效括号序列
  - 栈保存左括号

  -
    ```
    class Solution:
    ```

```python
    def isValid(self , s ):
        # write code here
        if not s:
            return True
        stack = []
        for item in s:
            if item == '(' or item == '[' or item == '{':
                stack.append(item)
            elif item == ')':
                if not stack or stack.pop() != '(':
                    return False
            elif item == ']' :
                if not stack or stack.pop() != '[':
                    return False
            elif item == '}':
                if not stack or stack.pop() != '{':
                    return False
        return len(stack) == 0
```

- BM49 表达式求值
    - 栈+递归

    -
      ```python
      class Solution:
          def solve(self , s ):
              # write code here
              s = s.strip()
              stack = []
              res = 0
              sign = '+'
              number = 0
              index = 0
              while index < len(s):
                  if s[index] == ' ':
                      index += 1
                      continue
                  if s[index] == '(':
                      end = index + 1
                      lens = 1
                      while lens > 0:
                          if s[end] == '(':
                              lens += 1
                          if s[end] == ')':
                              lens -= 1
                          end += 1
                      number = self.solve(s[index+1:end-1])
                      index = end-1
                      continue
                  if '0' <= s[index] <= '9':
                      number = number * 10 + int(s[index])
                  if not '0' <= s[index] <= '9' or index == len(s) - 1:
                      if sign == '+':
                          stack.append(number)
                      elif sign == '-':
                          stack.append(-1 * number)
                      elif sign == '*':
                          stack.append(stack.pop() * number)
                      elif sign == '/':
                          stack.append(stack.pop() / number)
                      number = 0
                      sign = s[index]
                  index += 1
              while stack:
                  res += stack.pop()
              return res
      ```
- BM53 缺失的第一个正整数
    - 对应索引赋为负值

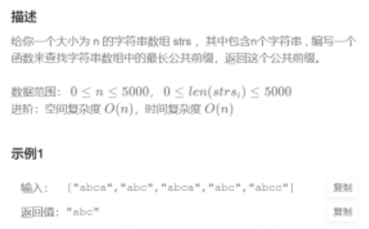
    -
      ```python
      class Solution:
          def minNumberDisappeared(self , nums: List[int]) -> int:
              # write code here
              N = len(nums) + 2
              for i in range(len(nums)):
                  if nums[i] <= 0:
                      nums[i] = N
              for i in range(len(nums)):
                  if 1 <= abs(nums[i]) <= len(nums):
                      nums[abs(nums[i])-1] = -abs(nums[abs(nums[i])-1])
              for i in range(len(nums)):
                  if nums[i] > 0:
                      return i + 1
              return len(nums) + 1
      ```
    - Set()函数去重

    -
      ```python
      class Solution:
          def minNumberDisappeared(self , nums: List[int]) -> int:
              # write code here
      ```

```python
        nums = set(nums)
        for i in range(1, len(nums)+1):
            if not i in nums:
                return i
        return len(nums)+1
```

- BM54 三数之和
    - 排序+固定+两数之和+去重

    -
      ```python
      class Solution:
          def threeSum(self , num ):
              # write code here
              num.sort()
              res = []
              for i in range(len(num)):
                  if i > 0 and num[i] == num[i-1]:
                      continue
                  temp = self.twoSum(num[i+1:], -num[i])
                  if temp:
                      for item in temp:
                          item.append(num[i])
                          res.append(item)
              return res
          def twoSum(self, num, target):
              res = []
              left = 0
              right = len(num) - 1
              while left < right:
                  cur_left = num[left]
                  cur_right = num[right]
                  if cur_left + cur_right == target:
                      res.append([cur_left, cur_right])
                      while left < right and num[right] == cur_right:
                          right -= 1
                      while left < right and num[left] == cur_left:
                          left += 1
                  elif cur_left + cur_right < target:
                      while left < right and num[left] == cur_left:
                          left += 1
                  elif cur_left + cur_right > target:
                      while left < right and num[right] == cur_right:
                          right -= 1
              return res
      ```
- BM83 字符串变形

    -
      ```python
      class Solution:
          def trans(self , s: str, n: int) -> str:
              # write code here
              l = s.split(' ')
              for i in range(len(l)):
                  l[i] = l[i].swapcase()
              return ' '.join(l[::-1])
      ```
- BM84 最长公共前缀
    - 
      描述

      给你一个大小为 n 的字符串数组 strs，其中包含n个字符串，编写一个函数来查找字符串数组中的最长公共前缀，返回这个公共前缀。

      数据范围：$0 \le n \le 5000$，$0 \le len(strs_i) \le 5000$
      进阶：空间复杂度 $O(n)$，时间复杂度 $O(n)$

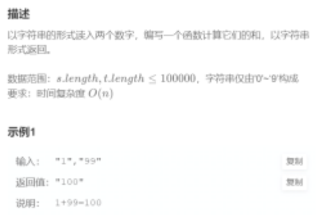
      示例1

      输入： ["abca","abc","abca","abc","abcc"]
      返回值： "abc"
    - 找最小+从左到右遍历

    -
      ```python
      class Solution:
          def longestCommonPrefix(self , strs: List[str]) -> str:
              # write code here
              if not strs:
                  return ''
              if len(strs) == 1:
                  return strs[0]
              res = ''
              min_str = strs[0]
              for item in strs:
                  if len(item) < len(min_str):
                      min_str = item
              for i in range(1, len(min_str)+1):
                  cur = min_str[:i]
                  for item in strs:
                      if cur not in item:
                          return res
                  res = cur
              return res
      ```
- BM85 验证IP地址

- 根据符号分类

-
  ```
  class Solution:
      def solve(self , IP: str) -> str:
          # write code here
          if not IP:
              return 'Neither'
          if '.' in IP:
              list_IP = IP.split('.')
              for item in list_IP:
                  if not '0' <= item <= '255' or len(item) > 1 and item[0] == '0':
                      return 'Neither'
              return 'IPv4'
          elif ':' in IP:
              list_IP = IP.split(':')
              for item in list_IP:
                  if not item or len(item) > 1 and len(item) == item.count('0'):
                      return 'Neither'
                  for cur in item:
                      if not cur.isdigit() and not cur in 'abcdef' and not cur in 'ABCDEF':
                          return 'Neither'
              return 'IPv6'
          else:
              return 'Neither'
  ```

- BM86 大数加法
  - 

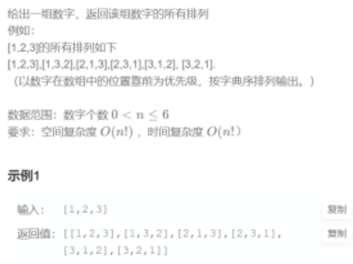
    描述

    以字符串的形式读入两个数字，编写一个函数计算它们的和，以字符串形式返回。

    数据范围：$s.length, t.length \leq 100000$，字符串仅由'0'~'9'构成
    要求：时间复杂度 $O(n)$

    示例1

    输入： "1","99" 复制
    返回值： "100" 复制
    说明： 1+99=100
  - 补零右对齐+逆序加+进位

  -
    ```
    class Solution:
        def solve(self , s , t ):
            # write code here
            len_s = len(s)
            len_t = len(t)
            max_len = max(len_s, len_t)
            s = s.zfill(max_len)
            t = t.zfill(max_len)
            res = ''
            carry = 0
            for i in range(max_len-1, -1, -1):
                temp = int(s[i]) + int(t[i]) + carry
                if temp >= 10:
                    carry = 1
                    temp = temp - 10
                else:
                    carry = 0
                res = str(temp) + res
            if carry == 1:
                res = str(carry) + res
            return res
    ```
- BM55 没有重复项数字的全排列
  - 

    给出一组数字，返回该组数字的所有排列
    例如：
    [1,2,3]的所有排列如下
    [1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2], [3,2,1].
    （以数字在数组中的位置靠前为优先级，按字典序排列输出。）

    数据范围：数字个数 $0 < n \leq 6$
    要求：空间复杂度 $O(n!)$，时间复杂度 $O(n!)$

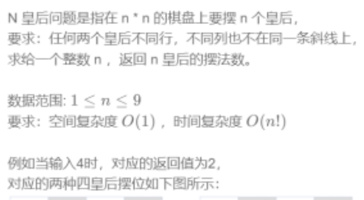
    示例1

    输入： [1,2,3] 复制
    返回值： [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]] 复制
  - 回溯

  -
    ```
    class Solution:
        def permute(self , num: List[int]) -> List[List[int]]:
            # write code here
            res = []
            cur = []
            vis = [0] * len(num)
            def recur(x, l):
                if x == l:
                    res.append(cur[:])
                    return
                for i in range(len(num)):
                    if vis[i] == 1:
                        continue
                    cur.append(num[i])
    ```

```
                    vis[i] = 1
                    recur(x+1, l)
                    vis[i] = 0
                    cur.pop()
            recur(0, len(num))
            return res
```

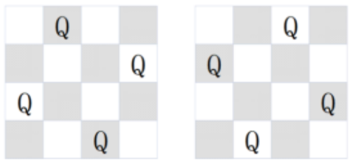
- BM56 有重复项数字的全排列
  - 回溯法+考虑重复元素

  -
    ```
    class Solution:
        def permuteUnique(self , num: List[int]) -> List[List[int]]:
            # write code here
            num.sort()
            res = []
            cur = []
            vis = [0] * len(num)
            def recur(x, l):
                if x == l:
                    res.append(cur[:])
                    return
                for i in range(len(num)):
                    if vis[i] == 1:
                        continue
                    if i > 0 and vis[i] == 0 and num[i] == num[i-1] and vis[i-1] == 0:
                        continue
                    cur.append(num[i])
                    vis[i] = 1
                    recur(x+1, l)
                    vis[i] = 0
                    cur.pop()
            recur(0, len(num))
            return res
    ```
- N皇后问题
  - 

    N 皇后问题是指在 n * n 的棋盘上要摆 n 个皇后，
    要求：任何两个皇后不同行，不同列也不在同一条斜线上，
    求给一个整数 n ，返回 n 皇后的摆法数。

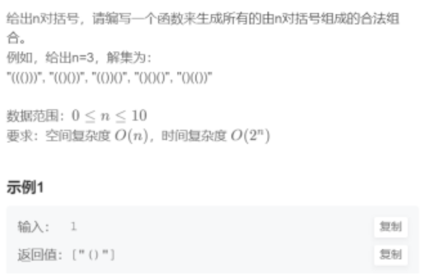
    数据范围: $1 \le n \le 9$
    要求：空间复杂度 $O(1)$ ，时间复杂度 $O(n!)$

    例如当输入4时，对应的返回值为2，
    对应的两种四皇后摆位如下图所示：

  - 回溯法+控制行

  -
    ```
    class Solution:
        def Nqueen(self , n ):
            # write code here
            self.res = 0
            col = []
            zheng = []
            fan = []
            def recur(row):
                if row == n:
                    self.res += 1
                    return
                for c in range(n):
                    if c in col or row-c in zheng or c+row in fan:
                        continue
                    col.append(c)
                    zheng.append(row-c)
                    fan.append(row+c)
                    recur(row+1)
                    col.pop()
                    zheng.pop()
                    fan.pop()
            recur(0)
            return self.res
    ```
- BM60 括号生成



- 给出n对括号，请编写一个函数来生成所有的由n对括号组成的合法组合。
  例如，给出n=3，解集为：
  "((()))", "(()())", "(())()", "()()()", "()(())"

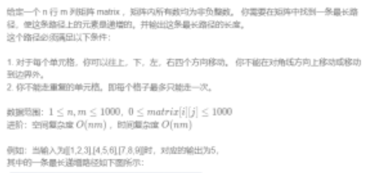
  数据范围：$0 \le n \le 10$
  要求：空间复杂度 $O(n)$，时间复杂度 $O(2^n)$

  示例1

  输入： 1 复制
  返回值： ["()"] 复制
- 回溯法+确定当前位

-
  ```
  class Solution:
      def generateParenthesis(self , n: int) -> List[str]:
          # write code here
          res = []
          cur = []
          def recur(left, right):
              if left == n and right == n:
                  res.append(''.join(cur))
                  return
              if left < n:
                  cur.append('(')
                  left += 1
                  recur(left, right)
                  left -= 1
                  cur.pop()
              if right < left:
                  cur.append(')')
                  right += 1
                  recur(left, right)
                  right -= 1
                  cur.pop()
          recur(0, 0)
          return res
  ```

- BM61 矩阵最长递增路径

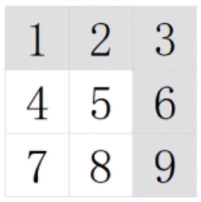


  给定一个 n 行 m 列矩阵 matrix，矩阵内所有数均为非负整数。你需要在矩阵中找到一条最长路径，使这条路径上的元素是递增的。并输出这条最长路径的长度。
  这个路径必须满足以下条件：

  1. 对于每个单元格，你可以往上，下，左，右四个方向移动。你不能在对角线方向上移动或移动到边界外。
  2. 你不能走重复的单元格。即每个格子最多只能走一次。

  数据范围：$1 \le n, m \le 1000$，$0 \le matrix[i][j] \le 1000$
  进阶：空间复杂度 $O(nm)$，时间复杂度 $O(nm)$

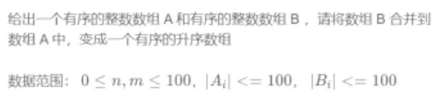
  例如：当输入为[[1,2,3],[4,5,6],[7,8,9]]时，对应的输出为5，
  其中的一条最长递增路径如下图所示：



- 回溯法+以当前元素为起点的递增数组的长度

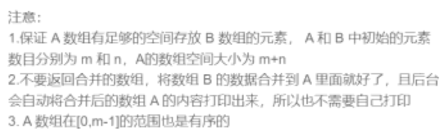
-
  ```
  class Solution:
      def solve(self , matrix ):
          # write code here
          if not matrix:
              return 0
          # 以当前点为起点的递增数组长度
          def recur(i, j, pre):
              if i < 0 or j < 0 or i >= len(matrix) or j >= len(matrix[0]):
                  return 0
              if matrix[i][j] <= pre:
                  return 0
              if memo[i][j] != -1:
                  return memo[i][j]
              cur = 0
              cur = max(cur, recur(i+1, j, matrix[i][j]))
              cur = max(cur, recur(i, j+1, matrix[i][j]))
              cur = max(cur, recur(i, j-1, matrix[i][j]))
              cur = max(cur, recur(i-1, j, matrix[i][j]))
              memo[i][j] = cur + 1
              return cur + 1
          res = 0
          memo = [[-1 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]
          for m in range(len(matrix)):
              for n in range(len(matrix[0])):
                  cur = recur(m, n, -1)
                  res = max(res, cur)
          return res
  ```

- BM87 合并两个有序数组



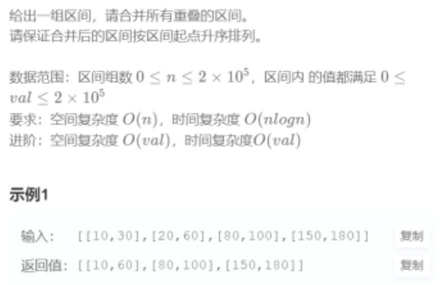
给出一个有序的整数数组 A 和有序的整数数组 B ，请将数组 B 合并到数组 A 中，变成一个有序的升序数组

数据范围： $0 \le n, m \le 100$，$|A_i| <= 100$，$|B_i| <= 100$

- 

  注意：
  1.保证 A 数组有足够的空间存放 B 数组的元素， A 和 B 中初始的元素数目分别为 m 和 n，A的数组空间大小为 m+n
  2.不要返回合并的数组，将数组 B 的数据合并到 A 里面就好了，且后台会自动将合并后的数组 A 的内容打印出来，所以也不需要自己打印
  3. A 数组在[0,m-1]的范围也是有序的
- 双指针+逆序

-
  ```
  class Solution:
      def merge(self , A, m, B, n):
          # write code here
          p = m-1
          q = n-1
          while q >= 0 and p >= 0:
              if A[p] > B[q]:
                  A[p+q+1] = A[p]
                  p -= 1
              else:
                  A[p+q+1] = B[q]
                  q -= 1
          while q >= 0:
              A[p+q+1] = B[q]
              q -= 1
          while p >= 0:
              A[p+q+1] = A[p]
              p -= 1
          return A
  ```

* BM89 合并区间
  - 

    给出一组区间，请合并所有重叠的区间。
    请保证合并后的区间按区间起点升序排列。

    数据范围：区间组数 $0 \le n \le 2 \times 10^5$，区间内 的值都满足 $0 \le val \le 2 \times 10^5$
    要求：空间复杂度 $O(n)$，时间复杂度 $O(nlogn)$
    进阶：空间复杂度 $O(val)$，时间复杂度$O(val)$

    示例1

    输入： [[10,30],[20,60],[80,100],[150,180]] 复制
    返回值： [[10,60],[80,100],[150,180]] 复制
  - 双指针+前一个区间的尾部+后一个区间的头部

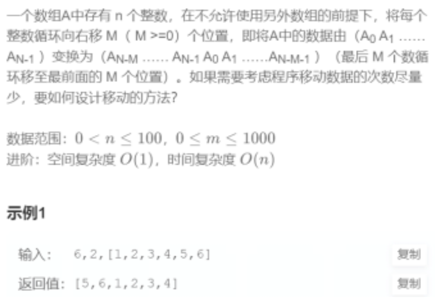
  -
    ```
    class Solution:
        def merge(self , intervals: List[Interval]) -> List[Interval]:
            # write code here
            if len(intervals) <= 1:
                return intervals
            res = []
            intervals.sort(key=lambda x:x.start, reverse=False)
            res.append(intervals[0])
            for i in range(1, len(intervals)):
                if intervals[i].start <= res[-1].end:
                    res[-1].end = max(intervals[i].end, res[-1].end)
                else:
                    res.append(intervals[i])
            return res
    ```
* BM93 盛水最多的容器
  - 双指针+左右指针比较，值较小的移动

  -
    ```
    class Solution:
        def maxArea(self , height ):
            # write code here
            if len(height)<=1:
                return 0
            left = 0
            right = len(height) - 1
            res = 0
            while left < right:
                res = max(res, min(height[right],height[left])*(right-left))
                if height[left] < height[right]:
                    left += 1
                else:
                    right -= 1
            return res
    ```
* BM94 接雨水问题
  - 双指针+先找中间最大值+左区间+右区间

  -
    ```
    class Solution:
        def maxWater(self , arr: List[int]) -> int:
            # write code here
            if len(arr) <= 2:
    ```

```
                return 0
            max_val = max(arr)
            idx = arr.index(max_val)
            left = 0
            right = 0
            water = 0
            while right <= idx:
                while arr[left] > arr[right]:
                    water += arr[left] - arr[right]
                    right += 1
                left = right
                right += 1
            right = len(arr) - 1
            left = len(arr) - 1
            while left >= idx:
                while arr[left] < arr[right]:
                    water += arr[right] - arr[left]
                    left -= 1
                right = left
                left -= 1
            return water
```

- BM90 最小覆盖子串
  - 滑动窗口+哈希表+need+windows

  -
    ```
    class Solution:
        def minWindow(self , S: str, T: str) -> str:
            # write code here
            need = {}
            windows = {}
            valid = 0
            start = 0
            end = len(S)+8
            for item in T:
                if item in need:
                    need[item] += 1
                else:
                    need[item] = 1
            left = 0
            right = 0
            while right < len(S):
                cur = S[right]
                right += 1
                if cur in need:
                    if cur in windows:
                        windows[cur] += 1
                    else:
                        windows[cur] = 1
                    if windows[cur] == need[cur]:
                        valid += 1
                while left < right and valid == len(need):
                    if len(S[left:right]) < end - start:
                        res = S[left:right]
                        end = right
                        start = left
                    cur_left = S[left]
                    left += 1
                    if cur_left in need:
                        windows[cur_left] -= 1
                        if windows[cur_left] < need[cur_left]:
                            valid -= 1
            if end == len(S)+8:
                return ''
            else:
                return res
    ```
- BM97 旋转数组
  - 
    一个数组A中存有 n 个整数，在不允许使用另外数组的前提下，将每个整数循环向右移 M（ M >=0）个位置，即将A中的数据由（A_0 A_1 ……A_{N-1} ）变换为（A_{N-M} ……A_{N-1} A_0 A_1 ……A_{N-M-1} ）（最后 M 个数循环移至最前面的 M 个位置）。如果需要考虑程序移动数据的次数尽量少，要如何设计移动的方法?

    数据范围：$0 < n \le 100$，$0 \le m \le 1000$
    进阶：空间复杂度 $O(1)$，时间复杂度 $O(n)$

    示例1

    输入：　6,2,[1,2,3,4,5,6]
    返回值：[5,6,1,2,3,4]

  - 数组反转+分区间反转

  -
    ```
    class Solution:
        def solve(self , n , m , a ):
            # write code here
            temp = m % n
            self.reverses(a, 0, len(a)-1)
    ```

```
            self.reverses(a, 0, temp-1)
            self.reverses(a, temp, len(a)-1)
            return a
        def reverses(self, arr, left, right):
            while left < right:
                arr[left], arr[right] = arr[right], arr[left]
                left += 1
                right -= 1
            return
```

- BM99 顺时针旋转矩阵
    - 原地交换+矩阵转置+每一行反转

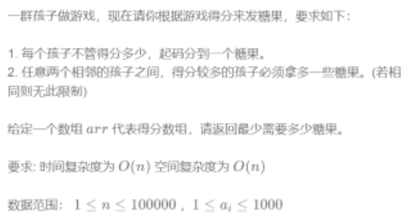
    -
      ```
      class Solution:
          def rotateMatrix(self , mat: List[List[int]], n: int) -> List[List[int]]:
              # write code here
              # 沿左对角线翻转（第一行变第一列）
              for i in range(len(mat)):
                  for j in range(i):
                      mat[i][j], mat[j][i] = mat[j][i], mat[i][j]
              # 每一行翻转
              for i in range(len(mat)):
                  mat[i].reverse()
              return mat
      ```
    - 额外申请空间

    -
      ```
      class Solution:
          def rotateMatrix(self , mat: List[List[int]], n: int) -> List[List[int]]:
              # write code here
              res = []
              for col in range(len(mat[0])):
                  cur = []
                  for row in range(len(mat)-1, -1, -1):
                      cur.append(mat[row][col])
                  res.append(cur)
              return res
      ```
- BM100 LRU缓存
    - 有序字典+move_to_end+popitem

    -
      ```
      class Solution:
          def __init__(self, capacity: int):
              # write code here
              import collections
              self.capacity = capacity
              self.LRU_cache= collections.OrderedDict()
          def get(self, key: int) -> int:
              # write code here
              if key in self.LRU_cache:
                  self.LRU_cache.move_to_end(key, last=True)
                  return self.LRU_cache[key]
              else:
                  return -1
          def set(self, key: int, value: int) -> None:
              # write code here
              if len(self.LRU_cache) < self.capacity:
                  self.LRU_cache[key] = value
              else:
                  self.LRU_cache.popitem(last=False)
                  self.LRU_cache[key] = value
              self.LRU_cache.move_to_end(key, last=True)
      ```
- BM101 LFU缓存
    - 无序字典(有序字典)+key_to_freq+freq_to_key

    -
      ```
      class Solution:
          def LFU(self , operators: List[List[int]], k: int) -> List[int]:
              # write code here
              from collections import OrderedDict, defaultdict
              self.key_to_freq = {}
              self.freq_to_key = defaultdict(OrderedDict)
              self.min_freq = 0
              self.capacity = k
              self.out = []
              for item in operators:
                  if item[0] == 1:
                      self.set(item[1], item[2])
                  else:
                      self.out.append(self.get(item[1]))
              return self.out
          def set(self, key, value):
              if self.capacity == 0:
                  return
              if key in self.key_to_freq:
                  freq = self.key_to_freq[key]
                  self.freq_to_key[freq+1][key] = value
      ```

```
            self.key_to_freq[key] = freq + 1
            self.freq_to_key[freq].pop(key)
            if not self.freq_to_key[freq] and freq == self.min_freq:
                self.min_freq += 1
        else:
            if len(self.key_to_freq) < self.capacity:
                self.key_to_freq[key] = 1
                self.freq_to_key[1][key] = value
                self.min_freq = 1
            else:
                pop_item = self.freq_to_key[self.min_freq].popitem(last=False)
                del self.key_to_freq[pop_item[0]]
                self.key_to_freq[key] = 1
                self.freq_to_key[1][key] = value
                self.min_freq = 1
    def get(self, key):
        if not key in self.key_to_freq:
            return -1
        else:
            freq = self.key_to_freq[key]
            val = self.freq_to_key[freq][key]
            self.freq_to_key[freq+1][key] = val
            self.key_to_freq[key] = freq + 1
            del self.freq_to_key[freq][key]
            if not self.freq_to_key[freq] and freq == self.min_freq:
                self.min_freq += 1
            return val
```

- BM95 分糖果问题
  - 

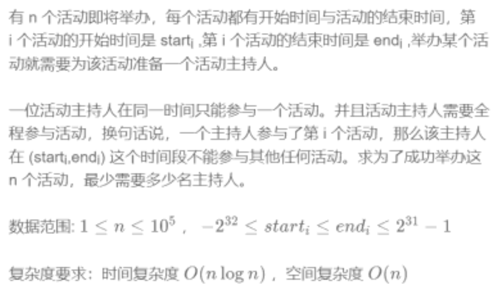
    一群孩子做游戏，现在请你根据游戏得分来发糖果，要求如下：

    1. 每个孩子不管得分多少，起码分到一个糖果。
    2. 任意两个相邻的孩子之间，得分较多的孩子必须拿多一些糖果。(若相同则无此限制)

    给定一个数组 arr 代表得分数组，请返回最少需要多少糖果。

    要求: 时间复杂度为 $O(n)$ 空间复杂度为 $O(n)$

    数据范围： $1 \le n \le 100000$, $1 \le a_i \le 1000$
  - 贪心算法+从左到右遍历+从右到左遍历

  -
    ```
    class Solution:
        def candy(self , arr ):
            # write code here
            temp = [1] * len(arr)
            for i in range(1, len(arr)):
                if arr[i] > arr[i-1]:
                    temp[i] = temp[i-1] + 1
            for i in range(len(arr)-2, -1, -1):
                if arr[i] > arr[i+1] and temp[i] <= temp[i+1]:
                    temp[i] = temp[i+1] + 1
            return sum(temp)
    ```
- BM96 主持人调度（二）
  - 

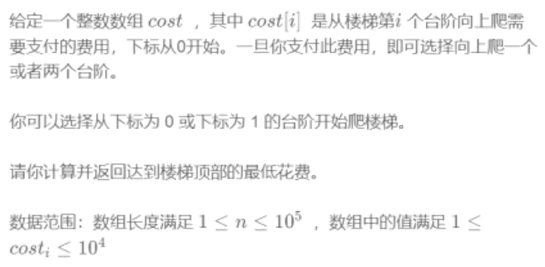
    有 n 个活动即将举办，每个活动都有开始时间与活动的结束时间，第 i 个活动的开始时间是 $start_i$,第 i 个活动的结束时间是 end_i ,举办某个活动就需要为该活动准备一个活动主持人。

    一位活动主持人在同一时间只能参与一个活动。并且活动主持人需要全程参与活动，换句话说，一个主持人参与了第 i 个活动，那么该主持人在 $(start_i,end_i)$ 这个时间段不能参与其他任何活动。求为了成功举办这 n 个活动，最少需要多少名主持人。

    数据范围: $1 \le n \le 10^5$ ， $-2^{32} \le start_i \le end_i \le 2^{31} - 1$

    复杂度要求：时间复杂度 $O(n \log n)$ ，空间复杂度 $O(n)$
  - 贪心算法+开始数组排序+结束数组排序+比较

  -
    ```
    class Solution:
        def minmumNumberOfHost(self , n , startEnd ):
            # write code here
            start = []
            end = []
            for i in range(len(startEnd)):
                start.append(startEnd[i][0])
                end.append(startEnd[i][1])
            start.sort()
            end.sort()
            res = 0
            idx = 0
            for i in range(len(start)):
                if start[i] < end[idx]:
                    res += 1
                else:
                    idx += 1
            return res
    ```

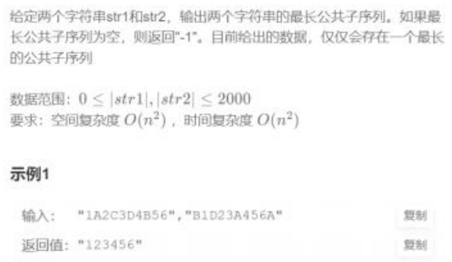
- BM64 最小花费跳楼梯
    - 

      给定一个整数数组 $cost$ ，其中 $cost[i]$ 是从楼梯第i 个台阶向上爬需要支付的费用，下标从0开始。一旦你支付此费用，即可选择向上爬一个或者两个台阶。

      你可以选择从下标为 0 或下标为 1 的台阶开始爬楼梯。

      请你计算并返回达到楼梯顶部的最低花费。

      数据范围：数组长度满足 $1 \leq n \leq 10^5$ ，数组中的值满足 $1 \leq cost_i \leq 10^4$
    - 动态规划+dp[i]表示调到第i个台阶的最小代价

    -
      ```
      class Solution:
          def minCostClimbingStairs(self , cost ):
              # write code here
              if cost <= 2:
                  return 0
              # dp[i]表示跳i个台阶的最小代价
              dp = [0, 0]
              for i in range(2, len(cost)+1):
                  dp.append(min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2]))
              return dp[-1]
      ```
- BM65 最长公共子序列（二）
    - 

      给定两个字符串str1和str2，输出两个字符串的最长公共子序列。如果最长公共子序列为空，则返回"-1"。目前给出的数据，仅仅会存在一个最长的公共子序列

      数据范围：$0 \leq |str1|, |str2| \leq 2000$
      要求：空间复杂度 $O(n^2)$ ，时间复杂度 $O(n^2)$

      示例1

      输入： "1A2C3D4B56","B1D23A456A" 复制
      返回值："123456" 复制
    - 动归+反推

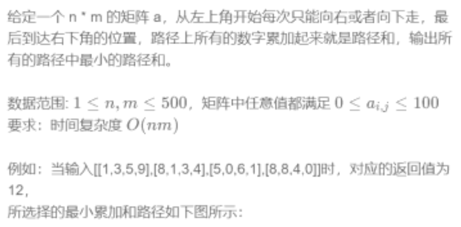
    -
      ```
      class Solution:
          def LCS(self , s1: str, s2: str) -> str:
              # write code here
              len1 = len(s1)
              len2 = len(s2)
              dp = [[0 for i in range(len2+1)] for j in range(len1+1)]
              for i in range(1, len1+1):
                  for j in range(1, len2+1):
                      if s1[i-1] == s2[j-1]:
                          dp[i][j] = dp[i-1][j-1] + 1
                      else:
                          dp[i][j] = max(dp[i-1][j], dp[i][j-1])
              res = ''
              m = len1
              n = len2
              while m > 0 and n > 0:
                  if s1[m-1] == s2[n-1]:
                      res += s1[m-1]
                      m -= 1
                      n -= 1
                  elif dp[m][n] == dp[m-1][n]:
                      m -= 1
                  elif dp[m][n] == dp[m][n-1]:
                      n -= 1
              if dp[-1][-1] == 0:
                  return '-1'
              return res[::-1]
      ```
- BM66 最长公共子串
    - 滑动窗口法

    -
      ```
      class Solution:
          def LCS(self , str1 , str2 ):
              # write code here
              res = ''
              left = 0
              for i in range(1, len(str1)+1):
                  if str1[left:i] in str2:
                      if len(str1[left:i]) > len(res):
                          res = str1[left:i]
                  else:
                      left += 1
              return res
      ```
    - 动态规划法

    -
      ```
      class Solution:
      ```

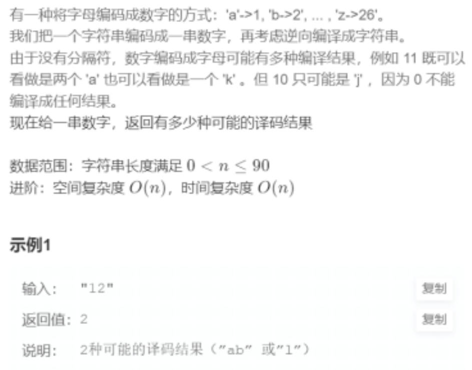
```
    def LCS(self , str1: str, str2: str) -> str:
        # write code here
        # write code here
        dp = [[0 for _ in range(len(str2)+1)] for _ in range(len(str1)+1)]
        res = ''
        cur = 0
        for i in range(1, len(str1)+1):
            for j in range(1, len(str2)+1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                    if dp[i][j] > cur:
                        cur = dp[i][j]
                        res = str1[i-dp[i][j]:i]
                else:
                    dp[i][j] = 0
        return res
```

- BM67 不同路径的数目
  - 动态规划+左边的格子的路径数+上边格子的路径数

  -
    ```
    class Solution:
        def uniquePaths(self , m: int, n: int) -> int:
            # write code here
            dp = [[0 for i in range(n)] for j in range(m)]
            for i in range(m):
                dp[i][0] = 1
            for i in range(n):
                dp[0][i] = 1
            for i in range(1, m):
                for j in range(1, n):
                    dp[i][j] = dp[i-1][j] + dp[i][j-1]
            return dp[-1][-1]
    ```
- BM68 矩阵的最小路径和
  - 
    给定一个 n * m 的矩阵 a，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，输出所有的路径中最小的路径和。

    数据范围: $1 \le n, m \le 500$，矩阵中任意值都满足 $0 \le a_{i,j} \le 100$
    要求：时间复杂度 $O(nm)$

    例如：当输入[[1,3,5,9],[8,1,3,4],[5,0,6,1],[8,8,4,0]]时，对应的返回值为12，
    所选择的最小累加和路径如下图所示：
  - 动态规划+dp表示当前点的最小路径和

  -
    ```
    class Solution:
        def minPathSum(self , matrix: List[List[int]]) -> int:
            # write code here
            if not matrix:
                return 0
            dp = [[0 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]
            dp[0][0] = matrix[0][0]
            for i in range(1, len(matrix)):
                dp[i][0] = dp[i-1][0] + matrix[i][0]
            for i in range(1, len(matrix[0])):
                dp[0][i] = dp[0][i-1]+matrix[0][i]
            for i in range(1, len(matrix)):
                for j in range(1, len(matrix[0])):
                    dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + matrix[i][j]
            return dp[-1][-1]
    ```
- BM69 把数字翻译成字符串
  - 
    有一种将字母编码成数字的方式：'a'->1, 'b->2', ... , 'z->26'。
    我们把一个字符串编码成一串数字，再考虑逆向编译成字符串。
    由于没有分隔符，数字编码成字母可能有多种编译结果，例如 11 既可以看做是两个 'a' 也可以看做是一个 'k'。但 10 只可能是 'j'，因为 0 不能编译成任何结果。
    现在给一串数字，返回有多少种可能的译码结果

    数据范围：字符串长度满足 $0 < n \le 90$
    进阶：空间复杂度 $O(n)$，时间复杂度 $O(n)$

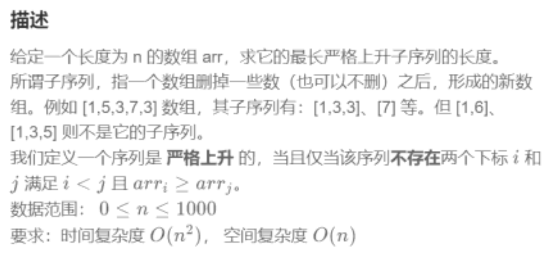
    示例1

    输入： "12"
    返回值： 2
    说明： 2种可能的译码结果（"ab" 或"l"）
  - 动态规划+dp表示以当前字母为尾元素的子字符串的翻译总数

  -
    ```
    class Solution:
        def solve(self , nums: str) -> int:
            # write code here
            if not nums:
                return 0
            dp = [0] * (len(nums)+1)
            dp[0] = 1
    ```

```
        dp[1] = 1
        for i in range(2, len(nums)+1):
            if '10' <= nums[i-2:i] <= '26' and nums[i-1] != '0':
                dp[i] = dp[i-1] + dp[i-2]
            elif nums[i-2:i] == '10' or nums[i-2:i] == '20':
                dp[i] = dp[i-2]
            elif nums[i-1] != '0':
                dp[i] = dp[i-1]
            else:
                dp[i] = 0
        return dp[-1]
```

- BM70 兑换零钱（一）
  - 动态规划+dp表示金额为i时需要的最小面币数+初始化为最大值

  -
    ```
    class Solution:
        def minMoney(self , arr: List[int], aim: int) -> int:
            # write code here
            dp = [aim+1 for _ in range(aim+1)]
            dp[0] = 0
            for i in range(1, aim+1):
                for item in arr:
                    if i-item >= 0:
                        dp[i] = min(dp[i], dp[i-item] + 1)
            if dp[-1] > aim:
                return -1
            else:
                return dp[-1]
    ```
- BM71 最长上升子序列（一）
  - 

    描述

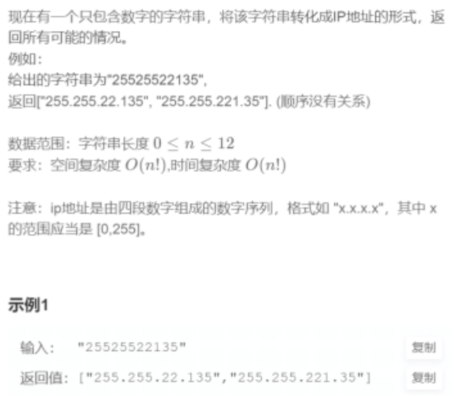
    给定一个长度为 n 的数组 arr，求它的最长严格上升子序列的长度。
    所谓子序列，指一个数组删掉一些数（也可以不删）之后，形成的新数组。例如 [1,5,3,7,3] 数组，其子序列有：[1,3,3]、[7] 等。但 [1,6]、[1,3,5] 则不是它的子序列。
    我们定义一个序列是 **严格上升** 的，当且仅当该序列**不存在**两个下标 i 和 j 满足 $i < j$ 且 $arr_i \geq arr_j$。
    数据范围： $0 \leq n \leq 1000$
    要求：时间复杂度 $O(n^2)$， 空间复杂度 $O(n)$
  - 贪心算法+temp数组记录长度为i的子序列的尾元素

  -
    ```
    class Solution:
        def LIS(self , arr: List[int]) -> int:
            # write code here
            if not arr:
                return 0
            dp = [0] * len(arr)
            temp = []
            for i in range(len(arr)):
                import bisect
                idx = bisect.bisect_left(temp, arr[i])
                if idx >= len(temp):
                    temp.append(arr[i])
                else:
                    temp[idx] = arr[i]
                dp[i] = idx + 1
            return max(dp)
    ```

  -
    ```
    class Solution:
        def LIS(self , arr: List[int]) -> int:
            # write code here
            if not arr:
                return 0
            dp = [0] * len(arr)
            temp = []
            for i in range(len(arr)):
                for j in range(len(temp)):
                    if temp[j] < arr[i]:
                        continue
                    else:
                        temp[j] = arr[i]
                        break
                if not temp or temp[-1] < arr[i]:
                    temp.append(arr[i])
                dp[i] = temp.index(arr[i]) + 1
            return max(dp)
    ```
- BM73 最长回文子串
  - 双指针+以每个元素和它右边的元素为中心+从中间向两边扩展

  -
    ```
    class Solution:
        def getLongestPalindrome(self , A: str) -> int:
            # write code here
            if not A:
                return 0
    ```

```
        res = ''
        for i in range(len(A)):
            s1 = self.find_rome(A, i, i)
            s2 = self.find_rome(A, i, i+1)
            if len(res) < len(s1):
                res = s1
            if len(res) < len(s2):
                res = s2
        return len(res)
    def find_rome(self, s, i, j):
        while i >= 0 and j < len(s) and s[i] == s[j]:
            i -= 1
            j += 1
        return s[i+1:j]
```

- BM74 数字字符串转换为IP地址
    - 
      现在有一个只包含数字的字符串，将该字符串转化成IP地址的形式，返回所有可能的情况。
      例如：
      给出的字符串为"25525522135",
      返回["255.255.22.135", "255.255.221.35"]. (顺序没有关系)

      数据范围：字符串长度 $0 \le n \le 12$
      要求：空间复杂度 $O(n!)$,时间复杂度 $O(n!)$

      注意：ip地址是由四段数字组成的数字序列，格式如 "x.x.x.x"，其中 x 的范围应当是 [0,255]。

      示例1

      输入： "25525522135"
      返回值： ["255.255.22.135","255.255.221.35"]
    - 回溯法+结束时考虑分割点的个数+当前阶段的开始点
    - class Solution:

```
    def restoreIpAddresses(self , s: str) -> List[str]:
        # write code here
        if not s:
            return []
        res = []
        self.recur(s, 0, 0, '', res)
        return res
    def recur(self, s, startIdx, dotNum, path, res):
        if dotNum == 3 and self.IsValid(s[startIdx:]):
            res.append(path+s[startIdx:])
            return
        elif dotNum == 3:
            return
        for i in range(startIdx+1, startIdx+4):
            cur_str = s[startIdx:i]
            if self.IsValid(cur_str):
                new_path = path + cur_str + '.'
                self.recur(s, i, dotNum+1, new_path, res)
            elif not self.IsValid(s) or i > len(s):
                continue
    def IsValid(self, s):
        if not s:
            return False
        flag = True
        # 值大于255
        if int(s) > 255:
            flag = False
        # 以0开头
        if len(s) > 1 and s[0] == '0':
            flag = False
        return flag
```

- BM75 编辑距离（一）
    - 动态规划+左上角（替换）+左边（插入）+右边（删除）
    - class Solution:

```
    def editDistance(self , str1: str, str2: str) -> int:
        # write code here
        dp = [[0 for _ in range(len(str2)+1)] for _ in range(len(str1)+1)]
        for i in range(1, len(str1)+1):
            dp[i][0] = i
        for j in range(1, len(str2)+1):
            dp[0][j] = j
        for i in range(1, len(str1)+1):
            for j in range(1, len(str2)+1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i-1][j-1], dp[i][j-1], dp[i-1][j]) + 1
        return dp[-1][-1]
```

- BM76 正则表达式匹配
  - 动态规划+p[j-1]是'*'和p[j-1]不是'*'

  -
    ```
    class Solution:
        def match(self , str: str, pattern: str) -> bool:
            # write code here
            dp = [[False for _ in range(len(pattern)+1)] for _ in range(len(str)+1)]
            for i in range(len(str)+1):
                for j in range(len(pattern)+1):
                    if j == 0:
                        dp[i][j] = i == 0
                        continue
                    if pattern[j-1] != '*':
                        if i > 0 and (pattern[j-1] == str[i-1] or pattern[j-1] == '.'):
                            dp[i][j] = dp[i-1][j-1]
                    elif pattern[j-1] == '*':
                        if j > 1:
                            dp[i][j] = dp[i][j-2]
                        if j > 1 and i > 0 and (str[i-1] == pattern[j-2] or pattern[j-2] == '.'):
                            dp[i][j] |= dp[i-1][j]
            return dp[-1][-1]
    ```
- BM77 最长的括号子串
  - 动态规划+dp表示以当前字符结束时的最长长度+当是左括号+当是右括号

  -
    ```
    class Solution(object):
        def longestValidParentheses(self, s):
            """
            :type s: str
            :rtype: int
            """
            stack = []  # 记录左括号的索引
            dp = [0] * (len(s)+1)  # dp[i]:以s[i-1]结尾的最长括号子串的长度
            for i in range(len(s)):
                if s[i] == '(':
                    stack.append(i)
                    dp[i+1] = 0   # 以s[i]左括号结尾的括号子串一定是无效的
                else:
                    if stack:
                        left_idx = stack.pop()   # 弹出与当前右括号匹配的左括号索引
                        # 以s[i]右括号结尾的最长括号子串的长度=当前右括号与其匹配的左括号之间的长度
                        # + 匹配的左括号之前的括号子串的长度dp[left_idx](以s[left_idx-1]结尾)
                        len_l = i - left_idx + 1 + dp[left_idx]
                        dp[i+1] = len_l
                    else:
                        dp[i+1] = 0
            return max(dp)
    ```
- BM78 打家劫舍（一）
  - 动态规划+dp[i][0]+dp[i][1]+第i间房子偷与不偷的最大金额

  -
    ```
    class Solution:
        def rob(self , nums: List[int]) -> int:
            # write code here
            dp = [[0, 0] for _ in range(len(nums))]
            dp[0][0] = 0
            dp[0][1] = nums[0]
            for i in range(1, len(nums)):
                dp[i][0] = max(dp[i-1][1], dp[i-1][0])
                dp[i][1] = dp[i-1][0] + nums[i]
            return max(dp[-1][0], dp[-1][1])
    ```
  - dp[i]表示从第i间房子开始偷可以获得的最大金额

  -
    ```
    class Solution:
        def rob(self , nums: List[int]) -> int:
            # write code here
            dp = [0] * (len(nums)+2)
            dp[-1] = 0
            dp[-2] = 0
            for i in range(len(nums)-1, -1, -1):
                dp[i] = max(dp[i+1], dp[i+2] + nums[i])
            return dp[0]
    ```
- BM79 打家劫舍（二）
  - 动态规划+dp[i]表示从第i间房子开始偷可以获得最高金额+从后往前推

  -
    ```
    class Solution:
        def rob(self , nums: List[int]) -> int:
            # write code here
            if not nums:
                return 0
            if len(nums) == 1:
                return nums[0]
            return max(self.money(nums, 0, len(nums)-2), self.money(nums, 1, len(nums)-1))
    ```

```
        def money(self, nums, left, right):
            dp = [0] * (len(nums)+2)
            dp[-1] = 0
            dp[-2] = 0
            if right < len(nums) - 1:
                dp[-3] = 0
            for i in range(right, left-1, -1):
                dp[i] = max(dp[i+1], dp[i+2]+nums[i])
            return dp[left]
```

- BM81 买卖股票的最好时机（二）
  - 动态规划+可多次购买

  -
    ```
    class Solution:
        def maxProfit(self , prices: List[int]) -> int:
            # write code here
            dp = [[0, 0] for _ in range(len(prices))]
            dp[0][0] = 0
            dp[0][1] = -prices[0]
            for i in range(1, len(prices)):
                dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i])
                dp[i][1] = max(dp[i-1][1], dp[i-1][0]-prices[i])
            return dp[-1][0]
    ```
- BM82 股票买卖的最好时机（三）
  - 动态规划+K=2

  -
    ```
    class Solution:
        def maxProfit(self , prices: List[int]) -> int:
            # write code here
            K = 2
            dp = [[[0, 0] for _ in range(K+1)] for _ in range(len(prices))]
            for i in range(len(prices)):
                # K是允许的最大交易次数
                for k in range(1, K+1):
                    if i == 0:
                        dp[i][k][0] = 0
                        dp[i][k][1] = -prices[i]
                        continue
                    dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1]+prices[i])
                    dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0]-prices[i])
            return dp[-1][-1][0]
    ```

  -
    ```
    class Solution:
        def maxProfit(self , prices ):
            # write code here
            dp_i_10 = 0
            dp_i_11 = -prices[0]
            dp_i_20 = 0
            dp_i_21 = -prices[0]
            for i in range(1, len(prices)):
                dp_i_10 = max(dp_i_10, dp_i_11+prices[i])
                dp_i_11 = max(dp_i_11, -prices[i])
                dp_i_20 = max(dp_i_20, dp_i_21+prices[i])
                dp_i_21 = max(dp_i_21, dp_i_10-prices[i])
            return dp_i_20
    ```