目录

第一部分

二分搜索

34. 在排序数组中查找元素的第一个和最后一个位置

```
# 34. 在排序数组中查找元素的第一个和最后一个位置
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        return self.findZone(nums, target, 0, len(nums)-1)

    def findZone(self, nums, target, start, end):
        if start > end:
            return [-1, -1]
        else:
            mid = int((start + end)/2)
            if nums[mid] == target:
                left = mid
                while left >= 0 and nums[left] == target :
                    left -= 1
                right = mid
                while right < len(nums) and  nums[right] == target:
                    right += 1
                return [left+1, right-1]
            elif nums[mid] > target:
                return self.findZone(nums, target, start, mid-1)
            elif nums[mid] < target:
                return self.findZone(nums, target, mid+1, end)

# 非递归的方式
class Solution:

    def searchRange(self, n_list, target):
        return [self.leftBound(n_list, target), self.rightBound(n_list, target)]
```

```
    def leftBound(self, n_list, target):
        low, hight = 0, len(n_list)-1
        while low <= hight:
            mid = (low + hight)//2
            if n_list[mid] > target:
                hight = mid -1
            elif n_list[mid] < target:
                low = mid + 1
            elif n_list[mid] == target:
                hight = mid -1
        if low >= len(n_list):
            return -1
        elif low < len(n_list):
            return  low if n_list[low] == target else -1

    def rightBound(self, n_list, target):
        low, hight = 0, len(n_list) - 1
        while low <= hight:
            mid = (low + hight)//2
            if n_list[mid] > target:
                hight = mid-1
            elif n_list[mid] < target:
                low = mid + 1
            elif n_list[mid] == target:
                low = mid + 1
        if hight < 0:
            return -1
        else:
            return hight if n_list[hight] == target else -1
```

704. 二分查找

```
# 704. 二分查找
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        return self.findN(nums, target, 0, len(nums)-1)

    def findN(self, nums, target, s, e):
        if s > e:
            return -1
        mid = int((s+e)/2)
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            return self.findN(nums, target, s, mid-1)
        elif nums[mid] < target:
            return self.findN(nums, target, mid+1, e)
```

35. 搜索插入位置

```python
# 35. 搜索插入位置
from typing import List

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        return self.findN(nums, target, 0, len(nums)-1)

    def findN(self, nums, target, s, e):
        if s > e:
            return s
        mid = int((s + e)/2)
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            return self.findN( nums,target, s, mid-1)
        elif nums[mid] < target:
            return self.findN(nums, target, mid+1, e)
```

354. 俄罗斯套娃信封问题

```python
# 354. 俄罗斯套娃信封问题
from functools import cmp_to_key
from typing import List
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        def sort_item(item1, item2):
            if item1[0] == item2[0]:
                return item2[1] - item1[1]
            else:
                return item1[0] - item2[0]
        envelopes_list = sorted(envelopes, key=cmp_to_key(sort_item))
        height_list = [item[1] for item in envelopes_list]
        # O(N*2)的方式记录dp会超时，但是这个方式确实太fancy了
        return self.lengthOfLIS(height_list)

    def lengthOfLIS(self, nums):
        piles, n = 0, len(nums)
        top = [0 for _ in range(n)]
        for i in range(n):
            poker = nums[i]
            left, right = 0, piles
            while left < right:
                mid = int((left + right)/2)
                if top[mid] >= poker:
                    right = mid
                else:
                    left = mid + 1
```

```
            if left == piles:
                piles += 1
            top[left] = poker
        return piles
```

300. 最长递增子序列

```
# 300. 最长递增子序列
from typing import List

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        dp = [1 for _ in nums]
        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j]+1)
        return max(dp)

class Solution1:
    def lengthOfLIS(self, nums):
        maxL = 0
        # 存放当前的递增序列的潜在数据
        dp = [0 for _ in nums]
        for num in nums:
            lo, hi = 0, maxL
            # 二分查找，并替换，这一步维护dp的本质是维护一个潜在的递增序列。非常trick
            while lo < hi:
                mid = lo + (hi-lo)/2
                if dp[mid] < num:
                    lo = mid + 1
                else:
                    hi = mid
            dp[lo] = num
            # 若是接在最后面，则连续递增序列张长了
            if lo == maxL:
                maxL += 1
        return maxL
```

392. 判断子序列

```
# 392. 判断子序列
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i,j = 0, 0
        while i < len(s) and j < len(t):
            if s[i] == t[j]:
                i += 1
                j += 1
```

```
            else:
                j += 1
        if i == len(s):
            return True
        else:
            return False
```

793. 阶乘函数后 K 个零

```
# 793. 阶乘函数后 K 个零
import sys
class Solution:
    def preimageSizeFZF(self, k: int) -> int:
        return self.rightBound(k) - self.leftBound(k) + 1

    def trailingZeros(self, n):
        res = 0
        d = n
        while d //5 > 0:
            res += d//5
            d = d//5
        return res

    def leftBound(self, target):
        low, hight = 0 , sys.maxsize
        while low < hight:
            mid = int((low + hight)/2)
            mid_zero = self.trailingZeros(mid)
            if mid_zero < target:
                low = mid+1
            elif mid_zero > target:
                hight = mid
            else:
                hight = mid
        return low

    def rightBound(self, target):
        low, hight = 0, sys.maxsize
        while low < hight:
            mid = int((low+hight)//2)
            mid_zero = self.trailingZeros(mid)
            if mid_zero < target:
                low = mid+1
            elif mid_zero > target:
                hight = mid
            else:
                low = mid+1
        return low-1
```

172. 阶乘后的零

```python
# 172. 阶乘后的零
class Solution:
    def trailingZeroes(self, n: int) -> int:
        res = 0
        divisor = 5
        while divisor <= n:
            res += n // divisor
            divisor *= 5
        return res
```

875. 爱吃香蕉的珂珂

```python
# 875. 爱吃香蕉的珂珂
from typing import List

class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        max_val = max(piles)
        if max_val <= 0:
            return 0
        return self.minHelp(piles, 1, max_val, h)

    def minHelp(self, piles: List[int], s: int, e: int, h: int):
        if s == e:
            return s
        mid = (s+e)//2
        if self.eatCount(piles, mid) > h:
            return self.minHelp(piles, mid+1, e, h)
        else:
            return self.minHelp(piles, s, mid, h)

    def eatCount(self, piles: List[int], k: int):
        counter = 0
        for n in piles:
            if n % k == 0:
                counter += n // k
            else:
                counter += n // k + 1
        return counter
```

1011. 在 D 天内送达包裹的能力

```python
from typing import List

# 1011. 在 D 天内送达包裹的能力
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        max_val = sum(weights)
```

```
        min_val = max(weights)
        return self.carrayWeight(weights, min_val, max_val, days)

    def carrayWeight(self, weights, s, e, days):
        if s == e:
            return s
        mid = (s + e) // 2
        if self.carrayDays(weights, mid) > days:
            return self.carrayWeight(weights, mid + 1, e, days)
        else:
            return self.carrayWeight(weights, s, mid, days)

    def carrayDays(self, weights, limitWeight):
        days = 0
        cumWeight = 0
        for w in weights:
            if cumWeight + w > limitWeight:
                days += 1
                cumWeight = w
            elif cumWeight + w == limitWeight:
                days += 1
                cumWeight = 0
            else:
                cumWeight += w
        if cumWeight != 0:
            days += 1
        return days

if __name__ == "__main__":
    s = Solution()
    weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    limitWeight = 11
    print(s.carrayDays(weights, limitWeight))
```

滑动窗口

3. 无重复字符的最长子串

```
# 3. 无重复字符的最长子串
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        idx_dict = dict()
        cur_idx = 0
        max_len = 0
        for j, alpha in enumerate(s):
            if idx_dict.get(s[j]) is None:
                idx_dict[s[j]] = [j]
                max_len = max(max_len, j-cur_idx+1)
            else:
                pre_idx = idx_dict.get(s[j])[-1]
                if cur_idx > pre_idx:
```

```
                    max_len = max(max_len, j-cur_idx+1)
                    idx_dict[s[j]].append(j)
                else:
                    max_len = max(max_len, j-pre_idx)
                    cur_idx = pre_idx + 1
                    idx_dict[s[j]].append(j)
        return max_len
```

76. 最小覆盖子串

```
from collections import defaultdict
class Solution(object):
    def minWindow(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        mem = defaultdict(int)
        for char in t:
            mem[char]+=1
        t_len = len(t)

        minLeft, minRight = 0,len(s)
        left = 0

        for right,char in enumerate(s):
            if mem[char]>0:
                t_len-=1
            mem[char]-=1

            if t_len==0:
                while mem[s[left]]<0:
                    mem[s[left]]+=1
                    left+=1

                if right-left<minRight-minLeft:
                    minLeft,minRight = left,right

                mem[s[left]]+=1
                t_len+=1
                left+=1
        return '' if minRight==len(s) else s[minLeft:minRight+1]
```

438. 找到字符串中所有字母异位词

```
# 438. 找到字符串中所有字母异位词
from collections import defaultdict
```

```python
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        if len(s) < len(p):
            return []
        p_dict, s_dict = dict(), dict()
        left, right = 0, -1
        res_list = []
        for a in p:
            if p_dict.get(a) is None:
                p_dict[a] = 0
            p_dict[a] += 1
            right += 1
            if s_dict.get(s[right]) is None:
                s_dict[s[right]] = 0
            s_dict[s[right]] += 1
        if self.cmp_dict(p_dict, s_dict):
            res_list.append(left)
        right += 1
        while right < len(s):
            if s_dict.get(s[right]) is None:
                s_dict[s[right]] = 0
            s_dict[s[right]] += 1
            if s_dict.get(s[left]) is None:
                s_dict[s[left]] = 0
            s_dict[s[left]] -= 1
            left += 1
            if self.cmp_dict(p_dict, s_dict):
                res_list.append(left)
            right += 1
        return res_list

    def cmp_dict(self, p1_dict, p2_dict):
        for k, v in p1_dict.items():
            if v == 0:
                continue
            if p2_dict.get(k) != v:
                return False
        for k, v in p2_dict.items():
            if v == 0:
                continue
            if p1_dict.get(k) != v:
                return False
        return True

s = "cbaebabacd"
p = "abc"
slo = Solution()
res = slo.findAnagrams(s, p)
print(res)
print(s[6:9])
```

567. 字符串的排列

```python
# 567. 字符串的排列
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s2) < len(s1):
            return False

        s2_dict, s1_dict = dict(), dict()
        left, right = 0, 0
        for s in s1:
            if s1_dict.get(s) is None:
                s1_dict[s] = 0
            s1_dict[s] += 1
            if s2_dict.get(s2[right]) is None:
                s2_dict[s2[right]] = 0
            s2_dict[s2[right]] += 1
            right += 1
        if self.cmp_dict(s1_dict, s2_dict):
            return True
        print(s2_dict)
        print(right)
        while right < len(s2):
            if s2_dict.get(s2[right]) is None:
                s2_dict[s2[right]] = 0
            s2_dict[s2[right]] += 1
            if s2_dict.get(s2[left]) is None:
                s2_dict[s2[left]] = 0
            s2_dict[s2[left]] -= 1
            left += 1
            if self.cmp_dict(s2_dict, s1_dict):
                return True
            right += 1
            print(s2_dict)
        return False

    def cmp_dict(self, t1_dict, t2_dict):
        for k, v in t1_dict.items():
            if v == 0:
                continue
            if t2_dict.get(k) != v:
                return False
        for k, v in t2_dict.items():
            if v == 0:
                continue
            if t1_dict.get(k) != v:
                return False
        return True

s1 = "adc"
s2 = "dcda"

sol = Solution()
```

```
res = sol.checkInclusion(s1, s2)
print(res)
```

239.滑动窗口最大值

```
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        win, ret = [], []
        for i, v in enumerate(nums):
            if i >= k and win[0] <= i - k:
                win.pop(0)
            while win and nums[win[-1]] <= v:
                win.pop()
            win.append(i)
            if i >= k - 1:
                ret.append(nums[win[0]])
        return ret
```

其他问题

26. 删除有序数组中的重复项

```
# 26. 删除有序数组中的重复项
from typing import List

class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if len(nums) <= 0:
            return 0
        s_p = 1
        pre_val = nums[0]
        for f_p in range(1, len(nums)):
            if pre_val != nums[f_p]:
                nums[s_p]= nums[f_p]
                s_p += 1
            pre_val = nums[f_p]
        return s_p

if __name__ == "__main__":
    nums = [1,1,2,2,2,3,3]
    sol = Solution()
    ret = sol.removeDuplicates(nums)
    print(ret)
```

83. 删除排序链表中的重复元素

```python
# 83. 删除排序链表中的重复元素
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if head is None:
            return head
        s_p = head
        pre_val = head.val
        f_p = head
        while f_p is not None:
            if pre_val != f_p.val:
                s_p.next = f_p
                s_p = s_p.next
            pre_val = f_p.val
            f_p = f_p.next
        s_p.next = None
        return head
```

27.移除元素

```python
# 27.移除元素
from typing import List

class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        s_p = 0
        for f_p, n in enumerate(nums):
            if nums[f_p] != val:
                nums[s_p] = nums[f_p]
                s_p += 1
        return s_p
```

283. 移动零

```python
# 283. 移动零
from typing import List

class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        s_p = 0
```

```
        for i, n in enumerate(nums):
            if n != 0:
                nums[s_p] = nums[i]
                s_p += 1
        while s_p < len(nums):
            nums[s_p] = 0
            s_p += 1
        return nums
```

11. 盛最多水的容器

```
# 11. 盛最多水的容器
from typing import List

class Solution:
    def maxArea(self, height: List[int]) -> int:
        i, j = 0, len(height)-1
        area = 0
        while i < j:
            cur_area = min(height[i], height[j]) * (j - i)
            area = max(cur_area, area)
            if height[i] < height[j]:
                i += 1
            else:
                j -= 1
        return area

height = [1,8,6,2,5,4,8,3,7]
sol = Solution()
assert 49 == sol.maxArea(height)
```

15.三数之和

```
from typing import List

#15.三数之和
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums = sorted(nums)
        pair_list = []
        for idx, n in enumerate(nums):
            tmp_nums = nums[idx+1:]
            if nums[idx] > 0:
                break
            if idx >= 1 and nums[idx] == nums[idx-1]:
                continue
            pair_list.extend(self.twoSum(-n, tmp_nums))
        return pair_list
```

```
    def twoSum(self, target, nums):
        i, j = 0, len(nums)-1
        pair_list = []
        while i < j:
            if nums[i] + nums[j] > target:
                j -= 1
            elif nums[i] + nums[j] < target:
                i += 1
            elif nums[i] + nums[j] == target:
                pair_list.append([-target, nums[i], nums[j]])
                while i < j and nums[i] == nums[i+1]:
                    i += 1
                while i < j and nums[j] == nums[j-1]:
                    j -= 1
                i += 1
                j -= 1
        return pair_list
```

18.四数之和

```
from typing import List

# 18.四数之和
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums = sorted(nums)
        return self.nSums(target, 4, nums)

    def nSums(self, target, n, nums):
        if n == 2:
            return self.twoSum(target, nums)
        else:
            res_list = []
            for i, val in enumerate(nums):
                tmp_nums = nums[i+1:]
                if i >= 1 and nums[i] == nums[i-1]:
                    continue
                sub_list = self.nSums(target-val, n-1, tmp_nums)
                for sub in sub_list:
                    sub.append(val)
                    res_list.append(sub)
            return res_list

    def twoSum(self, target, nums):
        i, j = 0, len(nums)-1
        res_list = []
        while i < j:
            if nums[i] + nums[j] == target:
                res_list.append([nums[i], nums[j]])
                while i< j and nums[i] == nums[i+1]:
                    i += 1
```

```
                while i< j and nums[j] == nums[j-1]:
                    j -= 1
                i += 1
                j -= 1
            elif nums[i] + nums[j] < target:
                i += 1
            elif nums[i] + nums[j] > target:
                j -= 1
        return res_list

if __name__ == "__main__":
    nums = [1, 0, -1, 0, -2, 2]
    target = 0
    sol = Solution()
    ret = sol.fourSum(nums, target)
    print(ret)
```

870. 优势洗牌

```
# 870. 优势洗牌
from typing import List


class Solution:
    def advantageCount(self, nums1: List[int], nums2: List[int]) -> List[int]:
        if len(nums1) != len(nums2):
            return []
        # 排序
        nums1_s = sorted(nums1)
        nums2_s = sorted(nums2)
        n2_dict = dict()
        # nums2的词典记录索引位置
        for i, n in enumerate(nums2):
            if n2_dict.get(n) is None:
                n2_dict[n] = []
            n2_dict[n].append(i)
        i, j = 0, 0
        nums1_tmp = [None for _ in nums1]
        # 排序后双指针对比
        while i < len(nums1) and j < len(nums2):
            if nums1_s[i] <= nums2_s[j]:
                i += 1
            else:
                nums1_tmp[j] = nums1_s[i]
                nums1_s[i] = None
                i += 1
                j += 1

        nums1_s = list(filter(lambda x: x != None, nums1_s))
        # 补数
        while j < len(nums2):
```

```
            nums1_tmp[j] = nums1_s.pop()
            j += 1
        # 还原
        for i, n in enumerate(nums1_tmp):
            n2_idx = n2_dict.get(nums2_s[i]).pop()
            nums1[n2_idx] = n
        return nums1
```

42.接雨水

```
#42.接雨水
from typing import List

class Solution:
    def trap(self, height: List[int]) -> int:
        if len(height) <= 2:
            return 0
        right_idx_list = sorted(range(len(height)), key=lambda i: height[i],
reverse=True)
        left_idx_list = []
        water = 0
        right_idx_list.remove(0)
        for i in range(1, len(height)-1):
            left_idx_list = self.update_idx_list(left_idx_list, i-1, height)
            right_idx_list.remove(i)
            left_max, right_max = height[left_idx_list[0]],
height[right_idx_list[0]]
            h = min(left_max, right_max) - height[i]
            print("left:", left_idx_list, "right:", right_idx_list, "cur_idx:", i)
            if h > 0:

                water += h
        return water

    def update_idx_list(self, idx_list, idx, height):
        if len(idx_list) == 0:
            idx_list.append(idx)
            return idx_list
        for i, cur_idx in enumerate(idx_list):
            if height[cur_idx] < height[idx]:
                idx_list.insert(i, idx)
                break
        return idx_list

class Solution1:
    def trap(self, height: List[int]) -> int:
        if len(height) <= 2:
            return 0
        water = 0
        for i in range(1, len(height)-1):
            h = min(max(height[:i]), max(height[i+1:])) - height[i]
```

```
            if h > 0:
                water += h
        return water

class Solution2:
    def trap(self, height: List[int]) -> int:
        ans = 0
        h1 = 0
        h2 = 0
        for i in range(len(height)):
            h1 = max(h1,height[i])
            h2 = max(h2,height[-i-1])
            ans = ans + h1 + h2 -height[i]
        return  ans - len(height)*h1

if __name__ == "__main__":
    height = [0,1,0,2,1,0,1,3,2,1,2,1]
    sol = Solution()
    ret = sol.trap(height)
    ret1 = sol.trap(height)
    print(ret)
    print(ret1)
```

区间问题

986.区间列表的交集

```
# 986.区间列表的交集
from typing import List

class Solution:
    def intervalIntersection(self, firstList: List[List[int]], secondList:
List[List[int]]) -> List[List[int]]:
        i,j = 0, 0
        result_list = []
        while i < len(firstList) and j < len(secondList):
            if firstList[i][0] > secondList[j][0]:
                if firstList[i][0] > secondList[j][1]:
                    j += 1
                else:
                    if firstList[i][1] < secondList[j][1]:
                        result_list.append([firstList[i][0], firstList[i][1]])
                        i += 1
                    else:
                        result_list.append([firstList[i][0], secondList[j][1]])
                        j += 1
            else:
                if secondList[j][0] > firstList[i][1]:
                    i += 1
                else:
                    if secondList[j][1] > firstList[i][1]:
```

```
                        result_list.append([secondList[j][0], firstList[i][1]])
                        i += 1
                    else:
                        result_list.append([secondList[j][0], secondList[j][1]])
                        j += 1
        return result_list
```

1288.删除被覆盖区间

```
# 1288.删除被覆盖区间
from functools import cmp_to_key
from typing import List

class Solution:
    def removeCoveredIntervals(self, intervals: List[List[int]]) -> int:
        def cmp(t1, t2):
            if t1[0] != t2[0] :
                return t1[0] - t2[0]
            else:
                return t2[1] - t1[1]
        intervals = sorted(intervals, key=cmp_to_key(cmp))
        print(intervals)
        j = 0
        while j < len(intervals) -1:
            if intervals[j][0] == intervals[j+1][0]:
                del intervals[j+1]
            else:
                if intervals[j][1] >= intervals[j+1][1]:
                    del intervals[j+1]
                else:
                    j += 1
        print(intervals)
        return len(intervals)


if __name__ == "__main__":
    # intervals = [[1,4],[3,6],[2,8]]
    intervals = [[34335,39239],[15875,91969],[29673,66453],[53548,69161],
[40618,93111]]
    sol = Solution()
    ret = sol.removeCoveredIntervals(intervals)
    print(ret)
```

56.合并区间

```
# 56.合并区间
from functools import cmp_to_key
from typing import List
```

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        def cmp(t1, t2):
            if t1[0] != t2[0]:
                return t1[0] - t2[0]
            else:
                return t2[1] - t1[1]
        intervals = sorted(intervals, key=cmp_to_key(cmp))
        i = 0
        while i < len(intervals) -1:
            if intervals[i][0] == intervals[i+1][0]:
                del intervals[i+1]
            else:
                if intervals[i][1] >= intervals[i+1][1]:
                    del intervals[i+1]
                else:
                    if intervals[i][1] < intervals[i+1][0]:
                        i += 1
                    elif intervals[i][1] < intervals[i+1][1]:
                        intervals[i] = [intervals[i][0], intervals[i+1][1]]
                        del intervals[i+1]
        return intervals
```

435.无重叠区间

```
# 435.无重叠区间
from functools import cmp_to_key
from typing import List

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals = sorted(intervals, key=lambda x: x[0])
        i = 0
        counter = 0
        while i <= len(intervals) - 2:
            if intervals[i][0] == intervals[i+1][0]:
                if intervals[i][1] > intervals[i+1][1]:
                    i += 1
                else:
                    intervals[i], intervals[i+1] = intervals[i+1], intervals[i]
                    i += 1
                counter += 1
            else:
                if intervals[i][1] >= intervals[i+1][1]:
                    i += 1
                    counter += 1
                else:
                    if intervals[i][1] <= intervals[i+1][0]:
                        i += 1
                    elif intervals[i][1] < intervals[i+1][1]:
                        intervals[i], intervals[i+1] = intervals[i+1],
```

```
intervals[i]
                        i += 1
                        counter += 1
        return counter
```

452.用最少数量的箭引爆气球

```
from typing import List

# 452.用最少数量的箭引爆气球
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if len(points) <= 0:
            return 0
        points = sorted(points, key=lambda x: x[1])
        counter = 0
        i =  0
        piv = points[i]
        while i < len(points):
            counter += 1
            while i < len(points) and piv[1] >= points[i][0]:
                i += 1
            if i < len(points):
                piv = points[i]
            else:
                piv = None
        return counter if piv is None else counter + 1

if __name__ == "__main__":
    points = [[1,2]]
    sol = Solution()
    ret = sol.findMinArrowShots(points)
    print(ret)
```

1024.视频拼接

```
from typing import List

# 1024.视频拼接
class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        clips = sorted(clips, key=lambda x: x[0])
        i = 0
        while i < len(clips) -1 and clips[i][0] == clips[i+1][0]:
            if clips[i][1] > clips[i+1][1]:
                clips[i], clips[i+1] = clips[i+1], clips[i]
            i += 1
        piv = clips[i]
        counter = 1
```

```
        if piv[0] == 0 and piv[1] >= time:
            return counter

        while i < len(clips):
            if piv[1] >= clips[i][0]:
                # print("cur_id:", i)
                if i < len(clips) -1:
                    while i < len(clips) -1 and clips[i+1][0] <= piv[1]:
                        if clips[i][1] > clips[i+1][1]:
                            clips[i], clips[i+1] = clips[i+1], clips[i]
                        i += 1
                    piv = [piv[0], clips[i][1]]
                    counter += 1
                    if piv[0] == 0 and piv[1] >= time:
                        return counter
                    i += 1
                else:
                    piv = [piv[0], clips[i][1]]
                    counter += 1
                    i += 1
            else:
                return -1
        print(piv)
        if piv[0] == 0 and piv[1] >= time:
            return counter
        else:
            return -1

if __name__ == "__main__":
    # clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],
[2,6],[3,4],[4,5],[5,7],[6,9]]
    clips = [[5,7],[1,8],[0,0],[2,3],[4,5],[0,6],[5,10],[7,10]]
    print(len(clips))
    time = 5
    sol = Solution()
    ret = sol.videoStitching(clips, time)
    print(ret)
```

链表双指针

2.两数相加

```
#2.两数相加

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
```

```
        l1_len = self.getLength(l1)
        l2_len = self.getLength(l2)
        if l2_len > l1_len:
            l1, l2 = l2, l1
        p = 0
        pre_node = None
        l1_head = l1
        while l1 is not None:
            l2_val = 0 if l2 is None else l2.val
            cur_val = l1.val + l2_val + p
            if cur_val // 10 == 1:
                p = 1
                cur_val = cur_val % 10
            else:
                p = 0
            l1.val = cur_val
            pre_node = l1
            l1 = l1.next
            l2 = None if l2 is None else l2.next
        if p != 0:
            pre_node.next = ListNode(val=p)
        return l1_head

    def getLength(self, list_node):
        counter = 0
        while list_node is not None:
            counter += 1
            list_node = list_node.next
        return counter
```

19.删除链表的倒数第N个节点

```
# Definition for singly-linked list.
# 19.删除链表的倒数第N个节点
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        p_head = ListNode(next=head)
        p_head_bk = p_head
        length = self.getLength(head)
        if length == n and n == 1:
            return None
        i = 0
        while i < length -n:
            p_head = p_head.next
            i += 1
        p_head_next = p_head.next
        if p_head_next is not None:
```

```
            p_head.next = p_head_next.next
        else:
            p_head.next = None
        return p_head_bk.next

    def getLength(self, head):
        counter = 0
        while head is not None:
            counter += 1
            head = head.next
        return counter
```

21.合并两个有序链表

```
# Definition for singly-linked list.
# 21.合并两个有序链表
from typing import Optional


class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode])
-> Optional[ListNode]:
        head = ListNode()
        p_head = head
        while list1 is not None and list2 is not None:
            if list1.val < list2.val:
                head.next = list1
                list1 = list1.next
            else:
                head.next = list2
                list2 = list2.next
            head = head.next
        if list1 is not None:
            head.next = list1
        if list2 is not None:
            head.next = list2
        return p_head.next
```

23.合并K个升序链表

```
# 23.合并K个升序链表
# Definition for singly-linked list.
from typing import List
import heapq
```

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 当成n-1个两个升序列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if len(lists) == 0:
            return None
        p1 = lists.pop()
        for p2 in lists:
            p1 = self.mergeListNode(p1, p2)
        return p1

    def mergeListNode(self, p1, p2):
        p_head = ListNode()
        p_head_bk = p_head
        while p1 is not None and p2 is not None:
            if p1.val < p2.val:
                p_head.next = p1
                p1 = p1.next
            else:
                p_head.next = p2
                p2 = p2.next
            p_head = p_head.next
        if p1 is not None:
            p_head.next = p1
        if p2 is not None:
            p_head.next = p2
        return p_head_bk.next

class Solution1:
    # 排序利用最小堆替代就是时间复杂度OK的K个升序的列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        p_head = ListNode()
        p_head_bk = p_head
        lists = list(filter(lambda node : node is not None, lists))
        if len(lists) == 0:
            return None

        heap_list = sorted(lists, key=lambda node:node.val)
        while len(heap_list) > 0:
            node = heap_list.pop(0)
            p_head.next = node
            p_head = p_head.next

            node = node.next
            if node is not None:
                heap_list.append(node)
                heap_list = sorted(heap_list, key=lambda node: node.val)
        return p_head_bk.next
```

141.环形链表

```
# Definition for singly-linked list.
from typing import Optional

# 141.环形链表
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```

142.环形链表2

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
# 142.环形链表2
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        slow, fast = head, head
        meet = None
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                meet = fast
                break
        if meet is None:
            return None
        slow = head
        while slow != fast:
            slow = slow.next
            fast = fast.next
        return slow
```

160.链表相交

```python
# Definition for singly-linked list.
# 160.链表相交
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        p1, p2 = headA, headB
        while p1 != p2:
            if p1 is None:
                p1 = headB
            else:
                p1 = p1.next
            if p2 is None:
                p2 = headA
            else:
                p2 = p2.next
        return p1
```

876.链表的中间结点

```python
# Definition for singly-linked list.
# 876. 链表的中间结点
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow, fast = head, head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

25. K 个一组翻转链表

```python
# Definition for singly-linked list.
from typing import Optional

# 25. K 个一组翻转链表
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
```

```
    def reverseKGroup(self, head: Optional[ListNode], k: int) ->
Optional[ListNode]:
        a, b = head, head
        for i in range(k):
            if b == None:
                return head
            b = b.next
        new_head = self.reverse(a, b)
        a.next = self.reverseKGroup(b, k)
        return new_head

    def reverse(self, a, b):
        pre, cur, nxt = None, a, a
        while cur != b:
            nxt = cur.next
            cur.next = pre
            pre = cur
            cur = nxt
        return pre
```

92.反转链表2

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
# 92.反转链表2
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        p_head = head
        counter = 1
        pre_node = None
        while counter < left :
            counter += 1
            pre_node = p_head
            p_head = p_head.next
        next_node = self.reverse(p_head, right-left+1)
        if pre_node is None:
            return next_node
        else:
            pre_node.next = next_node
            return head

    def reverse(self, head, right):
        if right == 1:
            return head
        p_head, pre_node = head, None
        counter = 1
        while counter <= right:
```

```
            counter += 1
            next_node = p_head.next
            p_head.next = pre_node
            pre_node = p_head
            p_head = next_node
        head.next = p_head
        return pre_node
```

234.回文链表

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
# 234.回文链表
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        n_list = []
        while head is not None:
            n_list.append(head.val)
            head = head.next
        print(n_list)
        i, j = 0, len(n_list)-1
        status = True
        while i < j:
            if n_list[i] != n_list[j]:
                status = False
                break
            else:
                i +=1
                j -= 1
        return status

class Solution1:
    def isPalindrome(self, head: ListNode) -> bool:
        p_head = head
        pre_node = ListNode(p_head.val)
        p_head = p_head.next
        while p_head is not None:
            node = ListNode(p_head.val)
            node.next = pre_node
            pre_node = node
            p_head = p_head.next

        status = True
        while pre_node is not None and head is not None:
            if pre_node.val == head.val:
                pre_node = pre_node.next
                head = head.next
            else:
```

```
                status = False
                break
        return status
```

前缀和问题

303. 区域和检索 - 数组不可变

```
# 303. 区域和检索 - 数组不可变
class NumArray:

    def __init__(self, nums: List[int]):
        self.acc_list = []
        acc = 0
        for n in nums:
            acc += n
            self.acc_list.append(acc)

    def sumRange(self, left: int, right: int) -> int:
        if left > 0:
            return self.acc_list[right] - self.acc_list[left-1]
        else:
            return self.acc_list[right]
```

304.二位区域和检索-矩阵不可变

```
# 304.二位区域和检索-矩阵不可变
from typing import List

class NumMatrix:

    def __init__(self, matrix: List[List[int]]):
        self.acc_matrix = [self.getAccListt(matrix[0])]
        for i in range(1, len(matrix)):
            acc_line = 0
            acc_list = []
            for j, n in enumerate(matrix[i]):
                acc_line += n
                acc_list.append(acc_line + self.acc_matrix[i-1][j])
            self.acc_matrix.append(acc_list)

    def getAccListt(self, n_list):
        acc = 0
        tmp = []
        for n in n_list:
            acc += n
            tmp.append(acc)
        return tmp
```

```
    def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:
        if col1 > 0 and row1 > 0:
            return self.acc_matrix[row2][col2] - self.acc_matrix[row2][col1-1] -
self.acc_matrix[row1-1][col2] + self.acc_matrix[row1-1][col1-1]
        elif col1 == 0  and row1 > 0:
            return self.acc_matrix[row2][col2] - self.acc_matrix[row1-1][col2]
        elif col1 > 0 and row1 == 0:
            return self.acc_matrix[row2][col2] - self.acc_matrix[row2][col1-1]
        elif col1 == 0 and row1 == 0:
            return self.acc_matrix[row2][col2]
```

560. 和为 K 的子数组

```
# 560. 和为 K 的子数组
from typing import List

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        acc_list, acc_dict = self.getAccList(nums)
        base = acc_dict[k] if acc_dict.get(k) is not None else 0
        count = 0
        for n in acc_list[::-1]:
            acc_dict[n] -= 1
            if acc_dict.get(n - k) is not None and acc_dict.get(n-k) > 0:
                count += acc_dict.get(n-k)
        return count + base

    def getAccList(self, nums):
        acc_dict = {}
        acc_list = []
        for n in nums:
            val = acc_list[-1] + n if len(acc_list) > 0 else  n
            acc_list.append(val)
            if acc_dict.get(val) is None:
                acc_dict[val] = 1
            else:
                acc_dict[val] += 1
        return acc_list, acc_dict

class Solution1:
    def subarraySum(self, nums: List[int], k: int) -> int:
        preSumDict = dict()
        preSumDict[0] = 1
        res, sum0_i = 0, 0
        for i in range(len(nums)):
            sum0_i += nums[i]
            sum0_j = sum0_i - k
            if preSumDict.get(sum0_j) is not None:
                res += preSumDict.get(sum0_j)
            if preSumDict.get(sum0_i) is None:
                preSumDict[sum0_i] = 1
```

```
            else:
                preSumDict[sum0_i] += 1
        return res
```

差分数组

1094. 拼车

```
# 1094. 拼车
from typing import List

class Solution:
    def carPooling(self, trips: List[List[int]], capacity: int) -> bool:
        max_stop = max([trip[2] for trip in trips])
        delta_cap_list = [0 for _ in range(max_stop+1)]
        for trip in trips:
            cap, start, stop = trip[0], trip[1], trip[2]
            delta_cap_list[start] += cap
            delta_cap_list[stop] -= cap
        cap_list = []
        status = True
        for delta in delta_cap_list:
            cap = cap_list[-1] + delta if len(cap_list) > 0 else delta
            if cap > capacity:
                status = False
                break
            else:
                cap_list.append(cap)
        return status
```

1109. 航班预订统计

```
# 1109. 航班预订统计
from typing import List

# 差分数组
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]], n: int) -> List[int]:
        delta_seat_list = [0 for _ in range(n+1)]
        for booking in bookings:
            first, last, seats = booking[0], booking[1], booking[2]
            delta_seat_list[first] += seats
            if last+1 < len(delta_seat_list):
                delta_seat_list[last+1] -= seats
        all_seat_list = []
        for delta in delta_seat_list[1:]:
            alls = all_seat_list[-1] + delta  if len(all_seat_list) > 0 else delta
```

```
            all_seat_list.append(alls)
        return all_seat_list
```

370.区间加法

```
# 370.区间加法
from typing import List
from winreg import REG_RESOURCE_LIST

class Solution:
    def zoneAdd(self, updates: List[List[int]], length: int) -> int:
        # max_val = max([item[1] for item in updates])
        delta_list = [0 for _ in range(length)]
        for item in updates:
            start, end , inc = item[0], item[1], item[2]
            delta_list[start] += inc
            if end + 1 < len(delta_list):
                delta_list[end+1] -= inc
        res_list = []
        for delta in delta_list:
            val = res_list[-1] + delta if len(res_list) > 0 else delta
            res_list.append(val)
        return res_list

if __name__ == "__main__":
    updates = [[1,3,2],[2,4,3],[0,2,-2]]
    length = 5
    s = Solution()
    ret = s.zoneAdd(updates, length)
    print(ret)
```

队列和栈

20.有效的括号

```
# 20.有效的括号
class Solution:
    def isValid(self, s: str) -> bool:
        n_list = []
        a_dict = {"(": ")", "[":"]", "{": "}"}
        status = True
        for a in s:
            if a not in [")", "}", "]"]:
                n_list.append(a)
            else:
                if len(n_list) > 0:
                    left = n_list.pop()
                    if a_dict.get(left) != a:
```

```
                        status = False
                        break
                else:
                    status = False
                    break
        if len(n_list) > 0:
            status = False
        return status
```

921. 使括号有效的最少添加

```
#921. 使括号有效的最少添加

# 骚操作
class Solution:
    def minAddToMakeValid(self, s: str) -> int:
        while  "()" in s :
            s = s.replace("()", "")
        return len(s)

# 正常版本
class Solution1:
    def minAddToMakeValid(self, s: str) -> int:
        left = []
        for a in s:
            if a == "(":
                left.append(a)
            elif a == ")":
                if len(left) > 0:
                    if left[-1] == "(":
                        left.pop()
                    else:
                        left.append(a)
                else:
                    left.append(a)
        return len(left)
```

1541.平衡括号字符串的最少插入次数

```
# 1541.平衡括号字符串的最少插入次数
class Solution:
    def minInsertions(self, s: str) -> int:
        left = []
        counter = 0
        i = 0
        while i < len(s):
            if s[i] == "(":
                left.append(s[i])
                i += 1
```

```
            elif s[i] == ")":
                if len(left) > 0:
                    left.pop()
                    if i+1 < len(s):
                        if s[i+1] == "(":
                            counter += 1
                            i += 1
                        elif s[i+1] == ")":
                            i += 2
                    else:
                        counter += 1
                        i += 1
                else:
                    counter += 1
                    if i+1 < len(s):
                        if s[i+1] == ")":
                            i += 2
                        elif s[i+1] == "(":
                            counter += 1
                            i += 1
                    else:
                        counter += 1
                        i += 1

        return counter + len(left) * 2

def is_exists(s):
    while "())" in s:
        s = s.replace("())", "")
        print(s)

if __name__ == "__main__":
    s = "((()))(()))()(()))"
    sol = Solution1()
    ret = sol.minInsertions(s)
    print(ret)
    print(is_exists(s))
```

32. 最长有效括号

```
# 32. 最长有效括号
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        left_idx_list = []
        dp = [0 for i in range(len(s) +1)]
        for i, a in enumerate(s):
            if a == "(":
                left_idx_list.append(i)
                dp[i+1] = 0
            elif a == ")":
```

```
                if len(left_idx_list) > 0:
                    left_idx = left_idx_list.pop()
                    length = i - left_idx + 1 + dp[left_idx]
                    dp[i+1] = length
                    i += 1
                else:
                    dp[i+1] = 0
        return max(dp)
```

225.用队列实现栈

```
import queue

# 225.用队列实现栈
def print_queue(que):
    n_list = []
    while not que.empty():
        n_list.append(que.get())
    print(n_list)

class MyStack:

    def __init__(self):
        self.que = queue.Queue()
        self.top_num = None

    def push(self, x: int) -> None:
        self.que.put(x)
        self.top_num = x

    def pop(self) -> int:
        rev_que = queue.Queue()
        q_size = self.que.qsize()
        if q_size > 2:
            while q_size > 2:
                cur = self.que.get()
                rev_que.put(cur)
                q_size -= 1
            self.top_num = self.que.get()
            pop_val = self.que.get()

            rev_que.put(self.top_num)
            self.que = rev_que
        elif q_size == 2:
            self.top_num = self.que.get()
            pop_val = self.que.get()
            self.que.put(self.top_num)
        elif q_size == 1:
            self.top_num = None
            pop_val = self.que.get()
        elif q_size == 0:
```

```
            pop_val = None
        return pop_val

    def top(self) -> int:
        return self.top_num

    def empty(self) -> bool:
        return self.que.empty()

# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

232.用栈实现队列

```
from collections import deque
# 232.用栈实现队列
class MyQueue:

    def __init__(self):
        self.stack = deque()
        self.top = None

    def push(self, x: int) -> None:
        if len(self.stack) == 0:
            self.top = x
        self.stack.append(x)

    def pop(self) -> int:
        if len(self.stack) == 0:
            return None
        stack_bak = deque()
        while len(self.stack) > 0:
            stack_bak.append(self.stack.pop())
        pop_val = stack_bak.pop()

        if len(stack_bak) > 0:
            self.top = stack_bak.pop()
            self.stack.append(self.top)
            while len(stack_bak) > 0:
                self.stack.append(stack_bak.pop())
        else:
            self.top = None
        return pop_val

    def peek(self) -> int:
        return self.top
```

```
    def empty(self) -> bool:
        return len(self.stack) == 0

# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

239.滑动窗口最大值

```
from typing import List

# 239.滑动窗口最大值
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        win, ret = [], []
        for i, v in enumerate(nums):
            if i >= k and win[0] <= i - k:
                win.pop(0)
            while win and nums[win[-1]] <= v:
                win.pop()
            win.append(i)
            if i >= k - 1:
                ret.append(nums[win[0]])
        return ret
```

二叉堆

23.合并K个升序链表

```
# 23.合并K个升序链表
# Definition for singly-linked list.
from typing import List
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 当成n-1个两个升序列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if len(lists) == 0:
            return None
        p1 = lists.pop()
```

```
        for p2 in lists:
            p1 = self.mergeListNode(p1, p2)
        return p1

    def mergeListNode(self, p1, p2):
        p_head = ListNode()
        p_head_bk = p_head
        while p1 is not None and p2 is not None:
            if p1.val < p2.val:
                p_head.next = p1
                p1 = p1.next
            else:
                p_head.next = p2
                p2 = p2.next
            p_head = p_head.next
        if p1 is not None:
            p_head.next = p1
        if p2 is not None:
            p_head.next = p2
        return p_head_bk.next


class Solution1:
    # 排序利用最小堆替代就是时间复杂度OK的K个升序的列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        p_head = ListNode()
        p_head_bk = p_head
        lists = list(filter(lambda node : node is not None, lists))
        if len(lists) == 0:
            return None

        heap_list = sorted(lists, key=lambda node:node.val)
        while len(heap_list) > 0:
            node = heap_list.pop(0)
            p_head.next = node
            p_head = p_head.next

            node = node.next
            if node is not None:
                heap_list.append(node)
                heap_list = sorted(heap_list, key=lambda node: node.val)
        return p_head_bk.next
```

215.数组中的第K大元素

```
from typing import List
import heapq
# 215.数组中的第K大元素
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
```

```
        class SmallHeap():
            def __init__(self, k: int):
                self.k = k
                self.small_heap = []

            def add(self, n: int):
                if len(self.small_heap) == k:
                    val = heapq.heappop(self.small_heap)
                    heapq.heappush(self.small_heap, max(val, n))
                else:
                    heapq.heappush(self.small_heap, n)

            def getTop(self):
                return self.small_heap[0]

        s_heap = SmallHeap(k)
        for n in nums:
            s_heap.add(n)
        print(s_heap.small_heap)
        return s_heap.getTop()

if __name__ == "__main__":
    nums = [3,2,1,5,6,4]
    k = 2
    sol = Solution()
    ret = sol.findKthLargest(nums, k)
    print(ret)
```

295.数据流的中位数

```
import heapq
# 295.数据流的中位数
class MedianFinder:
    def __init__(self):
        class SmallHeap:
            def __init__(self):
                self.small_heap = []

            def add(self, n):
                heapq.heappush(self.small_heap, n)

            def getTop(self):
                if self.num() > 0:
                    return self.small_heap[0]
                else:
                    return None

            def pop(self):
                if self.num() > 0:
                    return heapq.heappop(self.small_heap)
                else:
```

```
                    return None

            def num(self):
                return len(self.small_heap)

        class BigHeap:
            def __init__(self):
                self.big_heap = []

            def add(self, n):
                heapq.heappush(self.big_heap, n * (-1))

            def getTop(self):
                if self.num() > 0:
                    return self.big_heap[0] * -1
                else:
                    return None

            def pop(self):
                if self.num() > 0:
                    return heapq.heappop(self.big_heap) * (-1)
                else:
                    return None

            def num(self):
                return len(self.big_heap)

        self.s_heap = SmallHeap()
        self.b_heap = BigHeap()

    def addNum(self, num: int) -> None:
        small_top, big_top = self.s_heap.pop(), self.b_heap.pop()
        small_count, big_count = self.s_heap.num(), self.b_heap.num()
        sorted_list = []
        if small_top is not None:
            sorted_list.append(small_top)
        if big_top is not None:
            sorted_list.append(big_top)
        sorted_list.append(num)
        sorted_list = sorted(sorted_list)
        if small_count > big_count:
            if len(sorted_list) == 3:
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])
                self.b_heap.add(sorted_list[1])
            else:
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
        elif small_count < big_count:
            if len(sorted_list) == 3:
                self.s_heap.add(sorted_list[-2])
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])
            else:
```

```
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
        elif small_count == big_count:
            if len(sorted_list) == 1:
                self.s_heap.add(sorted_list[0])
            elif len(sorted_list) == 2:
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
            else:
                self.s_heap.add(sorted_list[-2])
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])

    def findMedian(self) -> float:
        small_count, big_count = self.s_heap.num(), self.b_heap.num()
        small_top, big_top = self.s_heap.getTop(), self.b_heap.getTop()
        if big_count > small_count:
            return big_top
        elif big_count < small_count:
            return small_top
        elif big_count == small_count:
            if small_count == 0:
                return None
            else:
                return (small_top + big_top)/2.0


# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()
```

703. 数据流中的第 K 大元素

```
# 703. 数据流中的第 K 大元素
from typing import List
import heapq

class KthLargest:
    def __init__(self, k: int, nums: List[int]):
        class SmallHeap:
            def __init__(self, k):
                self.k = k
                self.small_heap = []

            def add(self, val):
                if len(self.small_heap) == k:
                    top_val = heapq.heappop(self.small_heap)
                    heapq.heappush(self.small_heap, max(val, top_val))
                else:
```

```
                    heapq.heappush(self.small_heap, val)
                return self.small_heap[0]

        self.s_heap = SmallHeap(k)
        for n in nums:
            self.s_heap.add(n)

    def add(self, val: int) -> int:
        return self.s_heap.add(val)

# Your KthLargest object will be instantiated and called as such:
# obj = KthLargest(k, nums)
# param_1 = obj.add(val)
```

数据结构设计

146. LRU 缓存

```
# 146. LRU 缓存
from collections import OrderedDict

class LRUCache:

    def __init__(self, capacity: int):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if self.cache.get(key) is None:
            return -1
        self.makeRecentKey(key)
        return self.cache.get(key)

    def put(self, key: int, value: int) -> None:
        if self.cache.get(key) is not None:
            self.cache[key] = value
            self.makeRecentKey(key)
            return
        if len(self.cache) >= self.capacity:
            early_key = list(self.cache.keys())[0]
            del self.cache[early_key]
        self.cache[key] = value

    def makeRecentKey(self, key: int) -> None:
        val = self.cache.get(key)
        del self.cache[key]
        self.cache[key] = val



# Your LRUCache object will be instantiated and called as such:
```

```
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)

def testCase():
    capacity = 2
    lru = LRUCache(capacity)
    # ret = [lru.put(1, 1) ,lru.put(2, 2), lru.get(1), lru.put(3, 3), lru.get(2),
lru.put(4, 4), lru.get(1), lru.get(3), lru.get(4)]
    ret1 = [lru.put(2, 1), lru.put(2, 2), lru.get(2) ,lru.put(1, 1), lru.put(4,
1), lru.get(2)]
    print(ret1)


if __name__ == "__main__":
    testCase()
```

341. 扁平化嵌套列表迭代器

```
# """
# This is the interface that allows for creating nested lists.
# You should not implement it, or speculate about its implementation
# """

# 341. 扁平化嵌套列表迭代器
class NestedInteger:

   def isInteger(self) -> bool:
       """
       @return True if this NestedInteger holds a single integer, rather than a
nested list.
       """

   def getInteger(self) -> int:
       """
       @return the single integer that this NestedInteger holds, if it holds a
single integer
       Return None if this NestedInteger holds a nested list
       """

   def getList(self) -> [NestedInteger]:
       """
       @return the nested list that this NestedInteger holds, if it holds a nested
list
       Return None if this NestedInteger holds a single integer
       """

class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        self.nested_list = nestedList
```

```
    def next(self) -> int:
        return self.nested_list.pop(0).getInteger()

    def hasNext(self) -> bool:
        while len(self.nested_list) > 0 and not self.nested_list[0].isInteger():
            first_list = self.nested_list.pop(0).getList()
            for n in first_list[::-1]:
                self.nested_list.insert(0, n)
        return len(self.nested_list) > 0

# Your NestedIterator object will be instantiated and called as such:
# i, v = NestedIterator(nestedList), []
# while i.hasNext(): v.append(i.next())
```

380. O(1) 时间插入、删除和获取随机元素

```
# 380. O(1) 时间插入、删除和获取随机元素
import random
class RandomizedSet:

    def __init__(self):
        self.nums = []
        self.valToIdx = dict()

    def insert(self, val: int) -> bool:
        if self.valToIdx.get(val) is not None:
            return False
        self.valToIdx[val] = len(self.nums)
        self.nums.append(val)
        return True

    def remove(self, val: int) -> bool:
        if self.valToIdx.get(val) is None:
            return False

        idx = self.valToIdx.get(val)
        last_val = self.nums[-1]
        self.valToIdx[last_val] = idx
        self.nums[idx] = last_val

        self.nums.pop()
        del self.valToIdx[val]
        return True

    def getRandom(self) -> int:
        return self.nums[random.randint(0, len(self.nums)-1)]


# Your RandomizedSet object will be instantiated and called as such:
# obj = RandomizedSet()
```

```
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()

def testCase():
    rs_sol = RandomizedSet()
    ret = [rs_sol.insert(1), rs_sol.remove(2), rs_sol.insert(2),
rs_sol.getRandom(), rs_sol.remove(1), rs_sol.insert(2), rs_sol.getRandom()]
    ret = [rs_sol.insert(0), rs_sol.insert(1), rs_sol.remove(0), rs_sol.insert(2),
rs_sol.remove(1), rs_sol.getRandom()]

    print(ret)



if __name__ == "__main__":
    testCase()
```

460. LFU 缓存

```
from collections import OrderedDict
# 460. LFU 缓存
class LFUCache:
    def __init__(self, capacity: int):
        self.key_to_value = dict()
        self.key_to_freq = dict()
        self.freq_to_order_key = dict()
        self.capacity = capacity
        self.min_freq = 0

        self.cum_time = 0

    def get(self, key: int) -> int:
        if key not in self.key_to_value:
            return -1
        self.updateFreq(key)
        return self.key_to_value.get(key)

    def put(self, key: int, value: int) -> None:
        if self.capacity <= 0:
            return 0

        if self.key_to_value.get(key) is None:
            self.balanceFreq(key, value)
        else:
            self.key_to_value[key] = value
            self.updateFreq(key)


    def updateFreq(self, key):
```

```
        if self.key_to_freq.get(key) is None:
            self.key_to_freq[key] = 1
            if self.freq_to_order_key.get(1) is None:
                self.freq_to_order_key[1] = OrderedDict([(key, 1)])
            else:
                self.freq_to_order_key[1][key] = 1
            # 维护min_freq
            self.min_freq = 1
        else:
            # 更新freq, 删除老的freq
            cur_freq = self.key_to_freq[key]
            if self.freq_to_order_key.get(cur_freq) is not None:
                if key in self.freq_to_order_key.get(cur_freq):
                    del self.freq_to_order_key[cur_freq][key]
                    if len(self.freq_to_order_key.get(cur_freq)) == 0:
                        del self.freq_to_order_key[cur_freq]
                        # 维护min_freq
                        if cur_freq == self.min_freq:
                            self.min_freq = cur_freq + 1

            # 添加新的freq
            cur_freq += 1
            self.key_to_freq[key] = cur_freq
            if self.freq_to_order_key.get(cur_freq) is None:
                self.freq_to_order_key[cur_freq] = OrderedDict([(key, 1)])
            else:
                self.freq_to_order_key[cur_freq][key] = 1


    def balanceFreq(self, key, value):
        if len(self.key_to_value) == self.capacity:
            olden_key =
self.getOrderDictTopKey(self.freq_to_order_key.get(self.min_freq))
            del self.freq_to_order_key.get(self.min_freq)[olden_key]
            if len(self.freq_to_order_key.get(self.min_freq)) == 0:
                del self.freq_to_order_key[self.min_freq]
            del self.key_to_value[olden_key]
            del self.key_to_freq[olden_key]

        self.key_to_value[key] = value
        self.updateFreq(key)

    def getOrderDictTopKey(self, orderDict):
        for key in orderDict.keys():
            return key



# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

```
def testCase():
    cache = LFUCache(2)
    ret = [cache.put(1, 1), cache.put(2, 2), cache.get(1),
    cache.put(3, 3), cache.get(2), cache.get(3), cache.put(4,4),
    cache.get(1), cache.get(3), cache.get(4)]

    print(ret)

def testCase1():
    import time
    cache = LFUCache(10000)
    t1 = time.time()
    for i in range(100000):
        key, value = i, i * 5
        cache.put(key, value)
    for i in range(1000000):
        cache.get(key)
    t2 = time.time()
    delta = (t2 - t1) * 1000
    print(delta)
    print('cum_time:', cache.cum_time)
# [null,null,null,1,null,-1,3,null,-1,3,4]

    #285998.8663196564

    #25608.806371688843

if __name__ == "__main__":
    testCase1()
```

895. 最大频率栈

```
#895. 最大频率栈
class FreqStack:

    def __init__(self):
        self.valToFreq = dict()
        self.freqToValDict = dict()
        self.max_freq = 0


    def push(self, val: int) -> None:
        freq = self.valToFreq.get(val, 0) + 1
        self.valToFreq[val] = freq
        if self.freqToValDict.get(freq) is None:
            self.freqToValDict[freq] = list()
        self.freqToValDict[freq].append(val)
        self.max_freq = max(self.max_freq, freq)


    def pop(self) -> int:
```

```
        val = self.freqToValDict.get(self.max_freq).pop()
        freq = self.valToFreq.get(val) -1
        self.valToFreq[val] = freq
        if len(self.freqToValDict.get(self.max_freq)) == 0:
            del self.freqToValDict[self.max_freq]
            self.max_freq -= 1
        return val


# Your FreqStack object will be instantiated and called as such:
# obj = FreqStack()
# obj.push(val)
# param_2 = obj.pop()
```

第二部分

二叉树

94. 二叉树的中序遍历

```
# Definition for a binary tree node.
# 94. 二叉树的中序遍历
from typing import List, Optional


class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []
        self.recur_mid(root, result)
        return result

    def recur_mid(self, root, result):
        if root is None:
            return
        self.recur_mid(root.left, result)
        result.append(root.val)
        self.recur_mid(root.right, result)


# 非递归版本
class Solution1:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
```

```
        result = []
        tmp = [root]
        while len(tmp) > 0:
            node = tmp.pop(0)
            new_tmp = []
            if node.left is not None:
                new_tmp.append(node.left)
                node.left = None
                new_tmp.append(node)
                tmp = new_tmp + tmp
                continue
            else:
                result.append(node.val)
            if node.right is not None:
                new_tmp.append(node.right)
                tmp = new_tmp + tmp
                continue
        return result


def testCase():
    tn1 = TreeNode(val=1)
    tn2 = TreeNode(val=2)
    tn3 = TreeNode(val=3)
    tn1.right = tn2
    tn2.left = tn3

    sol = Solution1()
    ret = sol.inorderTraversal(tn1)
    print(ret)

if __name__ == "__main__":
    testCase()
```

100. 相同的树

```
# Definition for a binary tree node.
# 100. 相同的树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if p is None and q is None:
            return True
        elif (p is None and q is not None) or (p is not None and q is None):
            return False
        else:
```

```
            if p.val == q.val:
                return self.isSameTree(p.left, q.left) and
self.isSameTree(p.right, q.right)
            else:
                return False
```

102. 二叉树的层序遍历

```
# Definition for a binary tree node.
# 102. 二叉树的层序遍历
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_line(node_list, result)
        return result

    def recur_line(self, node_list, result):
        if len(node_list) == 0:
            return
        tmp_list = []
        lines = []
        for node in node_list:
            if node.left is not None:
                tmp_list.append(node.left)
            if node.right is not None:
                tmp_list.append(node.right)
            lines.append(node.val)
        result.append(lines)
        self.recur_line(tmp_list, result)
```

103. 二叉树的锯齿形层序遍历

```
# Definition for a binary tree node.
# 103. 二叉树的锯齿形层序遍历
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        node_list = [root]
        result = []
        self.recur_node(node_list, result, 1)
        return result

    def recur_node(self, node_list, result, depth):
        if len(node_list) == 0:
            return
        sub_nodes = []
        lines = []
        for node in node_list:
            if node.left is not None:
                sub_nodes.append(node.left)
            if node.right is not None:
                sub_nodes.append(node.right)
            lines.append(node.val)
        if depth % 2 == 0:
            lines = lines[::-1]
        result.append(lines)
        self.recur_node(sub_nodes, result, depth+1)
```

104.二叉树最大深度

```
# Definition for a binary tree node.
# 104.二叉树最大深度
from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        return self.getDepth(root)

    def getDepth(self, root):
        if root is None:
            return 0
        return max(self.getDepth(root.left), self.getDepth(root.right)) + 1

def testCase():
    tn1 = TreeNode(val=1)
```

```
    tn2 = TreeNode(val=9)
    tn3 = TreeNode(val=20)
    tn4 = TreeNode(val=15)
    tn5 = TreeNode(val=7)
    tn1.left = tn2
    tn1.right = tn3
    tn3.left = tn4
    tn3.right = tn5
    sol = Solution()
    ret = sol.maxDepth(tn1)
    print(ret)

if __name__ == "__main__":
    testCase()
```

105.从前序遍历和中序遍历构造二叉树

```
# Definition for a binary tree node.
# 105.从前序遍历和中序遍历构造二叉树
from typing import List
import sys
sys.setrecursionlimit(100000)

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if len(preorder) == 0:
            return None
        val = preorder.pop(0)
        node = TreeNode(val)
        in_idx = inorder.index(val)
        left_inorder = inorder[:in_idx]
        right_inorder = inorder[in_idx+1:]
        left_preorder, right_preorder = self.getLeftRight(preorder, left_inorder)
        node.left = self.buildTree(left_preorder, left_inorder)
        node.right = self.buildTree(right_preorder, right_inorder)
        return node

    def getLeftRight(self, preorder, left_inorder):
        left_preorder = []
        right_preorder = []
        left_inorder_dict = {n:1 for n in left_inorder}
        for o in preorder:
            if o in left_inorder_dict:
                left_preorder.append(o)
            else:
                right_preorder.append(o)
```

```
        return left_preorder, right_preorder


def testCase():
    import json
    import time
    line_list = []
    with open("./test_case/105.txt") as f:
        for line in f.readlines():
            n_list = json.loads(line.strip())
            line_list.append([int(n) for n in n_list])
    preorder = line_list[0]
    inorder = line_list[1]

    # print(len(preorder))
    # print(len(inorder))
    t1 = time.time()
    sol = Solution()
    sol.buildTree(preorder, inorder)
    t2 = time.time()
    print((t2-t1)*1000)

if __name__ == "__main__":
    testCase()
```

106. 从中序与后序遍历序列构造二叉树

```
# Definition for a binary tree node.
from typing import List

# 106. 从中序与后序遍历序列构造二叉树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        if len(postorder) == 0:
            return None
        val = postorder.pop()
        node = TreeNode(val)
        inorder_idx = inorder.index(val)
        left_inorder = inorder[:inorder_idx]
        right_inorder = inorder[inorder_idx+1:]
        left_postorder, right_postorder = self.getPostOrderLeftRight(postorder,
left_inorder)
        node.left = self.buildTree(left_inorder, left_postorder)
        node.right = self.buildTree(right_inorder, right_postorder)
        return node
```

```
    def getPostOrderLeftRight(self, postorder, left_inorder):
        left_inorder_dict = {i:1 for i in left_inorder}
        left_postorder, right_postorder = [] , []
        for o in postorder:
            if o in left_inorder_dict:
                left_postorder.append(o)
            else:
                right_postorder.append(o)
        return left_postorder, right_postorder
```

654. 最大二叉树

```
# Definition for a binary tree node.

# 654. 最大二叉树
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> TreeNode:
        if len(nums) == 0:
            return None
        max_val, max_idx = self.getMaxIdx(nums)
        print(max_val)
        left_nums = nums[:max_idx]
        right_nums = nums[max_idx+1:]
        node = TreeNode(max_val)
        node.left = self.constructMaximumBinaryTree(left_nums)
        node.right = self.constructMaximumBinaryTree(right_nums)
        return node


    def getMaxIdx(self, nums):
        max_val, max_idx = nums[0], 0
        for i in range(1, len(nums)):
            if max_val < nums[i]:
                max_val = nums[i]
                max_idx = i
        return max_val, max_idx

def testCase():
    nums = [3,2,1,6,0,5]
    sol = Solution()
    node = sol.constructMaximumBinaryTree(nums)
    print_node([node])
```

```
if __name__ == "__main__":
    testCase()
```

107. 二叉树的层序遍历 II

```
# Definition for a binary tree node.
from typing import List

# 107. 二叉树的层序遍历 II
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_lines(node_list, result)
        return result

    def recur_lines(self, node_list, res):
        if len(node_list) == 0:
            return
        new_nodes = []
        tmp = []
        for node in node_list:
            if node.left is not None:
                new_nodes.append(node.left)
            if node.right is not None:
                new_nodes.append(node.right)
            tmp.append(node.val)
        self.recur_lines(new_nodes, res)
        res.append(tmp)
        return
```

111. 二叉树的最小深度

```
# Definition for a binary tree node.

# 111. 二叉树的最小深度
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
    def minDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        return self.getDepth(root)

    def getDepth(self, root):
        if root.left is None and root.right is None:
            return 1
        elif root.left is None and root.right is not None:
            return self.getDepth(root.right) + 1
        elif root.left is not None and root.right is None:
            return self.getDepth(root.left) + 1
        elif root.left is not None and root.right is not None:
            return min(self.getDepth(root.left), self.getDepth(root.right)) + 1
```

114. 二叉树展开为链表

```
# Definition for a binary tree node.

# 114. 二叉树展开为链表
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        if root is None:
            return []

        node_list = [root]
        while len(node_list) > 0:
            node = node_list.pop(0)
            tmp_nodes = []
            node_left, node_right = node.left , node.right
            node.left = None
            if node_left is not None:
                node.right = node_left
                if node_right is not None:
                    tmp_nodes = [node_left, node_right]
                else:
                    tmp_nodes = [node_left]
                tmp_nodes.extend(node_list)
                node_list = tmp_nodes
                continue
            elif node_right  is not None:
```

```
                tmp_nodes = [node_right]
                tmp_nodes.extend(node_list)
                node_list = tmp_nodes
                continue
            elif node_right is None:
                if len(node_list) > 0:
                    node.right = node_list[0]
                else:
                    break
        return root
```

116. 填充每个节点的下一个右侧节点指针

```
"""
# Definition for a Node.

"""
# 116. 填充每个节点的下一个右侧节点指针
from typing import Optional

class Node:
    def __init__(self, val: int = 0, left: 'Node' = None, right: 'Node' = None,
next: 'Node' = None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next

class Solution:
    def connect(self, root: Optional[Node]) -> Optional[Node]:
        if root is None:
            return None
        node_list = [root]
        while len(node_list) > 0:
            nodes = []
            for i in range(len(node_list)):
                if i + 1 < len(node_list):
                    next_node = node_list[i+1]
                else:
                    next_node = None
                node_list[i].next = next_node
                if node_list[i].left is not None:
                    nodes.append(node_list[i].left)
                if node_list[i].right is not None:
                    nodes.append(node_list[i].right)
            node_list = nodes
        return root
```

226. 翻转二叉树

```python
# Definition for a binary tree node.
# 226. 翻转二叉树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if root is None:
            return root
        node_list = [root]
        self.invertTreeHelp(node_list)
        return root

    def invertTreeHelp(self, node_list):
        if len(node_list) == 0:
            return
        tmp = []
        print(node_list)
        for node in node_list:
            left_node, right_node = node.left, node.right
            node.left, node.right = right_node, left_node
            if left_node is not None:
                tmp.append(left_node)
            if right_node is not None:
                tmp.append(right_node)
        node_list = tmp
        self.invertTreeHelp(node_list)
```

144.二叉树的前序遍历

```python
# Definition for a binary tree node.
from typing import List, Optional

# 144.二叉树的前序遍历
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# 递归版本
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []
        self.preorder(root, result)
        return result
```

```
    def preorder(self, root, result):
        if root is None:
            return
        result.append(root.val)
        self.preorder(root.left, result)
        self.preorder(root.right, result)

# 非递归版本
class Solution1:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        node_list = [root]
        result = []
        while len(node_list) > 0:
            node = node_list.pop(0)
            tmp = []
            result.append(node.val)
            if node.left is not None:
                tmp.append(node.left)
            if node.right is not None:
                tmp.append(node.right)
            node_list = tmp + node_list
        return result
```

145.二叉树的后序遍历

```
# Definition for a binary tree node.
from typing import List, Optional

# 145.二叉树的后序遍历
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# 递归版本
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        result = []
        self.postOrder(root, result)
        return result

    def postOrder(self, root, result):
        if root is None:
            return []
        self.postOrder(root.left, result)
        self.postOrder(root.right, result)
        result.append(root.val)
```

```
# 非递归版本
class Solution1:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        result = []
        node_list = [root]
        while len(node_list) > 0:
            print(node_list)
            node = node_list.pop(0)
            if node.left is None and node.right is None:
                result.append(node.val)
            else:
                tmp = []
                if node.left is not None:
                    tmp.append(node.left)
                if node.right is not None:
                    tmp.append(node.right)
                node.left, node.right = None, None
                tmp.append(node)
                node_list = tmp + node_list
        return  result
```

222. 完全二叉树的节点个数

```
# Definition for a binary tree node.

# 222. 完全二叉树的节点个数
# 等比数列公式 Sn = a1 * (1-q**n)/(1-q), q为比值
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def countNodes(self, root: TreeNode) -> int:
        left, right = root, root
        hight_left, hight_right = 0, 0
        while left is not None:
            left = left.left
            hight_left += 1
        while right is not None:
            right = right.right
            hight_right += 1
        if hight_left == hight_right:
            return 2 ** hight_left - 1
        return 1 + self.countNodes(root.left) + self.countNodes(root.right)

class Solution1:
```

```
    def countNodes(self, root: TreeNode) -> int:
        if root is None:
            return 0
        if root.left is None and root.right is None:
            return 1
        elif root.left is None and root.right is not None:
            return 1 + self.countNodes(root.right)
        elif root.left is not None and root.right is None:
            return 1 + self.countNodes(root.left)
        elif root.left is not None and root.right is not None:
            return 1 + self.countNodes(root.left) + self.countNodes(root.right)
```

236. 二叉树的最近公共祖先

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

# 236. 二叉树的最近公共祖先
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        if root is None:
            return None
        # 这里处理p=5 q=4的情况
        if root.val == p.val or root.val == q.val:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if left is not None and right is not None:
            return root
        if left is None and right is None:
            return None
        if left is None:
            return right
        else:
            return left

# 会超时 因为递归在反复计算
class Solution1:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        if root is None:
            return root
        if (root.val == p.val and self.is_exists(root, q)) or (root.val == q.val
and self.is_exists(root, p)):
            return root
        left_p, right_p = self.is_exists(root.left, p), self.is_exists(root.righ,
```

```
p)
        left_q, right_q = self.is_exists(root.left, q), self.is_exists(root.right,
q)

        if (left_p and right_q) or (left_q and right_p):
            return root
        elif left_p and left_q:
            return self.lowestCommonAncestor(root.left, p, q)
        elif right_p and right_q:
            return self.lowestCommonAncestor(root.right, p, q)
        else:
            return None

    def is_exists(self, root, node):
        if root is None:
            return False
        if root.val == node.val:
            return True
        return self.is_exists(root.left, node) or self.is_exists(root.right, node)
```

297. 二叉树的序列化与反序列化

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
# 297. 二叉树的序列化与反序列化
class Codec:
    def __init__(self):
        self.sep=','
        self.null="#"

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        result_list = []
        self.serHelp(root, result_list)
        return self.sep.join(result_list)

    def serHelp(self, root, result_list):
        if root is None:
            result_list.append(self.null)
            return
        result_list.append(str(root.val))
        self.serHelp(root.left, result_list)
        self.serHelp(root.right, result_list)
```

```
    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        data_list = data.split(self.sep)
        return self.desHelp(data_list)

    def desHelp(self, data_list):
        if len(data_list) == 0:
            return None
        first = data_list.pop(0)
        if first == self.null:
            return None
        node = TreeNode(int(first))
        node.left = self.desHelp(data_list)
        node.right = self.desHelp(data_list)
        return node

# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))
```

501. 二叉搜索树中的众数

```
# Definition for a binary tree node.
# 501. 二叉搜索树中的众数
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def __init__(self):
        self.pre_node = None
        self.cur_count = 0
        self.max_count = 0
        self.mode_list = []

    def findMode(self, root: TreeNode) -> List[int]:
        self.traverse(root)
        return self.mode_list

    def traverse(self, root):
        if root is None:
```

```
            return
        self.traverse(root.left)
        # 开头
        if self.pre_node is None:
            self.cur_count = 1
            self.max_count = 1
            self.mode_list.append(root.val)
        else:
            if root.val == self.pre_node.val:
                self.cur_count += 1
                if self.cur_count == self.max_count:
                    self.mode_list.append(root.val)
                elif self.cur_count > self.max_count:
                    self.mode_list.clear()
                    self.max_count = self.cur_count
                    self.mode_list.append(root.val)
            elif root.val != self.pre_node.val:
                self.cur_count = 1
                if self.cur_count == self.max_count:
                    self.mode_list.append(root.val)
        self.pre_node = root
        self.traverse(root.right)
```

543. 二叉树的直径

```
# Definition for a binary tree node.
# 543. 二叉树的直径
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.max_len = 0
        self.getDepth(root)
        return self.max_len

    def getDepth(self, root):
        if root is None:
            return 0
        left_depth, right_depth = self.getDepth(root.left),
self.getDepth(root.right)
        self.max_len = max(left_depth + right_depth, self.max_len)
        return max(left_depth, right_depth) + 1
```

559. N 叉树的最大深度

```
"""
# Definition for a Node.
"""
# 559. N 叉树的最大深度
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class Solution:
    def maxDepth(self, root: 'Node') -> int:
        if root is None:
            return 0
        depth = 0
        for node in root.children:
            depth =  max(self.maxDepth(node), depth)
        return depth + 1
```

589. N 叉树的前序遍历

```
from typing import List

# 589. N 叉树的前序遍历
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if root is None:
            return []
        self.result = []
        self.preoderHelp(root)
        return self.result

    def preoderHelp(self, root):
        if root is None:
            return
        self.result.append(root.val)
        if root.children is not None:
            for node in root.children:
                self.preoderHelp(node)

# 非递归遍历
class Solution1:
    def preorder(self, root: 'Node') -> List[int]:
        if root is None:
            return []
        node_list = [root]
```

```
        result = []
        while len(node_list) > 0:
            node = node_list.pop(0)
            result.append(node.val)
            if node.children is not None:
                node_list = node.children + node_list
        return result
```

652. 寻找重复的子树

```
# Definition for a binary tree node.
# 652. 寻找重复的子树
from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def findDuplicateSubtrees(self, root: Optional[TreeNode]) ->
List[Optional[TreeNode]]:
        if root is None:
            return []
        self.subtree_dict = dict()
        self.result = []
        self.traverse(root)
        return self.result

    def traverse(self, root):
        if root is None:
            return "#"
        left, right = self.traverse(root.left), self.traverse(root.right)
        subtree = left + "," + right + "," + str(root.val)
        if self.subtree_dict.get(subtree) is None:
            self.subtree_dict[subtree] = 1
        else:
            self.subtree_dict[subtree] += 1
        if self.subtree_dict.get(subtree) == 2:
            self.result.append(root)
        return subtree
```

654. 最大二叉树

```
# Definition for a binary tree node.
# 654. 最大二叉树
from typing import List
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> TreeNode:
        if len(nums) == 0:
            return None
        max_val, max_idx = self.getMaxIdx(nums)
        print(max_val)
        left_nums = nums[:max_idx]
        right_nums = nums[max_idx+1:]
        node = TreeNode(max_val)
        node.left = self.constructMaximumBinaryTree(left_nums)
        node.right = self.constructMaximumBinaryTree(right_nums)
        return node

    def getMaxIdx(self, nums):
        max_val, max_idx = nums[0], 0
        for i in range(1, len(nums)):
            if max_val < nums[i]:
                max_val = nums[i]
                max_idx = i
        return max_val, max_idx
```

965. 单值二叉树

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# 965. 单值二叉树
class Solution:
    def isUnivalTree(self, root: TreeNode) -> bool:
        if root is None:
            return False
        val = root.val
        return self.univalTree(root, val)

    def univalTree(self, root, val):
        if root is None:
            return True
        if root.val == val:
            return self.univalTree(root.left, val) and self.univalTree(root.right,
val)
```

```
        else:
            return False
```

二叉搜索树

95. 不同的二叉搜索树 II

```
# Definition for a binary tree node.
# 95. 不同的二叉搜索树 II
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if n == 0:
            return []
        return self.build(1, n)

    def build(self, left, right):
        res = []
        if left > right:
            res.append(None)
            return res
        for i in range(left, right+1):
            # 自顶向下
            left_tree_list = self.build(left, i-1)
            right_tree_list = self.build(i+1, right)
            for left_tree in left_tree_list:
                for right_tree in right_tree_list:
                    node = TreeNode(i)
                    node.left = left_tree
                    node.right = right_tree
                    res.append(node)
        # 自底向上
        return res
```

96. 不同的二叉搜索树

```
# 96. 不同的二叉搜索树
class Solution:
    def numTrees(self, n: int) -> int:
        if n == 0:
            return 0
        self.val_list = [[0 for _ in range(n+2)] for _ in range(n+2)]
        return self.countTree(1, n+1)
```

```
    def countTree(self, left, right):
        if left >= right:
            return 1
        res = 0
        if self.val_list[left][right] != 0:
            return self.val_list[left][right]
        for i in range(left, right):
            left_count = self.countTree(left, i)
            right_count = self.countTree(i+1, right)
            res += left_count * right_count
        self.val_list[left][right] = res
        return res
```

98. 验证二叉搜索树

```
# Definition for a binary tree node.
# 98. 验证二叉搜索树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        if root is None:
            return True
        elif root.left is None and root.right is not None:
            if root.val < self.getLeftNode(root.right).val:
                return self.isValidBST(root.right)
            else:
                return False
        elif root.left is not None and root.right is None:
            if self.getRigthNode(root.left).val < root.val:
                return self.isValidBST(root.left)
            else:
                return False
        elif root.left is not None and root.right is not None:
            if root.val > self.getRigthNode(root.left).val and root.val <
self.getLeftNode(root.right).val:
                return self.isValidBST(root.left) and self.isValidBST(root.right)
            else:
                return False
        else:
            return True

    def getLeftNode(self, node):
        if node.left is None:
            return node
        return self.getLeftNode(node.left)
```

```
    def getRigthNode(self, node):
        if node.right is None:
            return node
        return self.getRigthNode(node.right)
```

450. 删除二叉搜索树中的节点

```
# Definition for a binary tree node.
from typing import Optional

# 450. 删除二叉搜索树中的节点
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) ->
Optional[TreeNode]:
        if  root is None:
            return None
        if root.val > key:
            root.left = self.deleteNode(root.left, key)
        elif root.val < key:
            root.right = self.deleteNode(root.right, key)
        else:
            if root.left is None or root.right is None:
                root = root.left if root.left is not None else root.right
            else:
                cur = root.right
                while cur.left is not None:
                    cur = cur.left
                root.val = cur.val
                root.right = self.deleteNode(root.right, cur.val)
        return root
```

700. 二叉搜索树中的搜索

```
#700. 二叉搜索树中的搜索

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```python
class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return None
        if root.val > val:
            return self.searchBST(root.left, val)
        elif root.val < val:
            return self.searchBST(root.right, val)
        else:
            return root

# 非递归版本
class Solution1:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return None
        val_node = None
        while root is not None:
            if root.val > val:
                root = root.left
            elif root.val < val:
                root = root.right
            else:
                val_node = root
                break
        return val_node
```

701. 二叉搜索树中的插入操作

```python

# Definition for a binary tree node.
# 701. 二叉搜索树中的插入操作
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def insertIntoBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return TreeNode(val)
        if root.val > val:
            root.left = self.insertIntoBST(root.left, val)
        elif root.val < val:
            root.right = self.insertIntoBST(root.right, val)
        return root
```

230.二叉搜索树中第K小的元素

```python
# Definition for a binary tree node.
from typing import Optional

# 230.二叉搜索树中第K小的元素
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        k, target = self.getK(root, k, None)
        return target

    def getK(self, root, k, target):
        if root is None:
            return k, target
        k, target = self.getK(root.left, k, target)
        k -= 1
        if k == 0:
            target = root.val
            return k, target
        k, target = self.getK(root.right, k, target)
        return k, target
```

538. 把二叉搜索树转换为累加树

```python
# Definition for a binary tree node.
from typing import Optional

# 538. 把二叉搜索树转换为累加树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if root is None:
            return None
        self.cum_val = 0
        result = 0
        self.CumSumTree(root, result)
        return root

    def CumSumTree(self, root, result):
        if root is None:
            return result
```

```
        result = self.CumSumTree(root.right, result)
        result += root.val
        root.val = result
        result = self.CumSumTree(root.left, result)
        return result
```

530. 二叉搜索树的最小绝对差

```
# Definition for a binary tree node.
#530. 二叉搜索树的最小绝对差
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def getMinimumDifference(self, root: TreeNode) -> int:
        pre_node = None
        min_val = 0
        if root is None:
            return min_val
        else:
            min_val = (root.val - root.left.val) if root.left is not None else
(root.right.val - root.val)
        min_val, _ = self.getMinDelta(root, min_val, pre_node)
        return min_val

    def getMinDelta(self, root, min_val, pre_node):
        if root is None:
            return min_val, pre_node
        min_val, pre_node = self.getMinDelta(root.left, min_val, pre_node)
        if pre_node is not None:
            min_val = min(root.val - pre_node.val, min_val)
        pre_node = root
        min_val , pre_node = self.getMinDelta(root.right, min_val, pre_node)
        return min_val, pre_node
```

783. 二叉搜索树节点最小距离

```
# Definition for a binary tree node.
# 783. 二叉搜索树节点最小距离
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
import sys
class Solution:
```

```
    def minDiffInBST(self, root: TreeNode) -> int:
        if root is None:
            return 0
        pre_node = None
        min_val = sys.maxsize
        min_val, pre_node = self.getMin(root, pre_node, min_val)
        return min_val


    def getMin(self, root, pre_node, min_val):
        if root is None:
            return min_val, pre_node
        min_val, pre_node = self.getMin(root.left, pre_node, min_val)
        if pre_node is not None:
            min_val = min(root.val - pre_node.val, min_val)
        pre_node = root
        min_val, pre_node = self.getMin(root.right, pre_node, min_val)
        return min_val, pre_node
```

1373. 二叉搜索子树的最大键值和

```
# Definition for a binary tree node.
# 1373. 二叉搜索子树的最大键值和

from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def maxSumBST(self, root: Optional[TreeNode]) -> int:
        if root.val is None:
            return 0
        self.max_val = 0
        status, val = self.searchBST(root, True, 0)
        return self.max_val

    def searchBST(self, root, status, val):
        if root is None:
            return True and status, 0
        left_status, left_val = self.searchBST(root.left, status, val)
        right_status, right_val = self.searchBST(root.right, status, val)
        status = left_status and right_status
        if root.left is not None:
            left_node = self.getRightNode(root.left)
            if left_node.val < root.val:
                status = status and True
            else:
                status = status and False
```

```
        if root.right is not None:
            right_node = self.getLeftNode(root.right)
            if right_node.val > root.val:
                status = status and True
            else:
                status = status and False
        if status:
            cum_val = root.val + left_val + right_val
            self.max_val = max(cum_val, self.max_val)
        else:
            cum_val = 0
        return status, cum_val

    def getLeftNode(self, root):
        if root is None:
            return None
        while root.left is not None:
            root = root.left
        return root

    def getRightNode(self, root):
        if root is None:
            return None
        while root.right is not None:
            root = root.right
        return root
```

图论算法

797. 所有可能的路径

```
# 797. 所有可能的路径
from typing import List

class Solution:
    def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:
        pre_list = [0]
        path_list = []
        self.getPath(graph[0], graph, path_list, pre_list)
        return path_list

    def getPath(self, n_list, graph, path_list, pre_list):
        for i in n_list:
            if i == len(graph)-1:
                tmp = pre_list[::]
                tmp.append(i)
                path_list.append(tmp)
            else:
                self.getPath(graph[i], graph, path_list, pre_list + [i])
```

785.判断二分图(二分图)

```
from typing import List

# 785.判断二分图
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n = len(graph)
        self.ok = True
        self.color = [False for _ in range(n)]
        self.visited = [False for _ in range(n)]
        for v in range(n):
            if self.visited[v] is False:
                self.traverse(graph, v)
        return self.ok

    def traverse(self, graph, v):
        if self.ok is False:
            return
        self.visited[v] = True
        for w in graph[v]:
            if self.visited[w] is False:
                self.color[w]  = not self.color[v]
                self.traverse(graph, w)
            else:
                if self.color[w] == self.color[v]:
                    self.ok = False
```

886. 可能的二分法(二分图)

```
# 886. 可能的二分法

class Solution:
    def possibleBipartition(self, n: int, dislikes: List[List[int]]) -> bool:
        graph = self.buildGraph(n, dislikes)

        self.visited = [False for _ in range(len(graph))]
        self.color = [False for _ in range(len(graph))]
        self.ok = True
        for v in range(1, n+1):
            if self.visited[v] is False:
                self.trans(graph, v)
        return self.ok

    def buildGraph(self, n, dislikes):
        graph = [[] for _ in range(n+1)]
        for edge in dislikes:
            v, w = edge[1], edge[0]
            if graph[v] is None:
                graph[v] = []
```

```
            graph[v].append(w)
            if graph[w] is None:
                graph[w] = []
            graph[w].append(v)
        return graph

    def trans(self, graph, v):
        if self.ok is False:
            return
        self.visited[v] = True
        for w in graph[v]:
            if self.visited[w] is False:
                self.color[w] = not self.color[v]
                self.trans(graph, w)
            else:
                if self.color[w] == self.color[v]:
                    self.ok = False
```

207. 课程表 (拓扑排序)

```
# 207. 课程表
from typing import List

class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        # 递归的点，防止递归的loop
        self.on_path = [False for _ in range(numCourses+1)]
        # 遍历的点，防止遍历的loop
        self.visited = [False for _ in range(numCourses+1)]
        self.has_clcle = False
        graph = self.buildGraph(numCourses, prerequisites)
        for i in range(numCourses):
            self.traverse(graph, i)
        return not self.has_clcle

    def buildGraph(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]
        for edge in prerequisites:
            froms = edge[1]
            tos = edge[0]
            graph[froms].append(tos)
        return graph

    def traverse(self, graph, i):
        if self.on_path[i] is True:
            self.has_clcle = True
        if self.visited[i] or self.has_clcle:
            return
        self.visited[i] = True
        self.on_path[i] = True
        for t in graph[i]:
```

```
            self.traverse(graph, t)
        self.on_path[i] = False
```

210. 课程表 II (拓扑排序)

```
# 210. 课程表 II
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        self.postorder = []
        self.has_clcle = False
        self.visited, self.on_path = [False for _ in range(numCourses+1)], [False
for _ in range(numCourses+1)]
        graph = self.buildGraph(numCourses, prerequisites)
        # 遍历图
        for i in range(numCourses):
            self.traverse(graph, i)
        # 有环就直接返回
        if self.has_clcle:
            return []
        self.postorder = self.postorder[::-1]
        return self.postorder

    def traverse(self, graph, s):
        if self.on_path[s]:
            self.has_clcle = True
        if self.visited[s] or self.has_clcle:
            return
        # 前序遍历
        self.on_path[s] = True
        self.visited[s] = True
        for t in graph[s]:
            self.traverse(graph, t)
        self.on_path[s] = False
        # 后序遍历 这里整个图遍历的时候不会有问题吗？
        self.postorder.append(s)
        return


    def buildGraph(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses+1)]
        for edge in prerequisites:
            # 单向
            froms, tos = edge[1], edge[0]
            graph[froms].append(tos)
        return graph
```

130. 被围绕的区域

```python
# 130. 被围绕的区域
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        if board is None or len(board) == 0:
            return
        row, col = len(board), len(board[0])
        for i in range(row):
            self.dfs(board, i, 0)
            self.dfs(board, i, col - 1)
        for j in range(col):
            self.dfs(board, 0, j)
            self.dfs(board, row-1, j)
        for i in range(row):
            for j in range(col):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                if board[i][j] == '-':
                    board[i][j] = 'O'

    def dfs(self, board, i, j):
        if (i< 0 or j < 0 or i >= len(board) or j >= len(board[0]) or board[i][j]
!= "O"):
            return
        board[i][j] = '-'
        self.dfs(board, i-1, j)
        self.dfs(board, i+1, j)
        self.dfs(board, i, j-1)
        self.dfs(board, i, j+1)
        return
```

990. 等式方程的可满足性

```python
# 990. 等式方程的可满足性
from typing import List

class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
```

```
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

class Solution:
    def equationsPossible(self, equations: List[str]) -> bool:
        uf = UF(26)
        a_id = ord('a')
        for eq in equations:
            if eq[1] == '=':
                x = eq[0]
                y = eq[3]
                uf.union(ord(x) - a_id, ord(y) - a_id)
        for eq in equations:
            if eq[1] == '!':
                x = eq[0]
                y = eq[3]
                if uf.connected(ord(x) - a_id, ord(y) - a_id):
                    return False
        return True
```

261.以图判树(最小生成树)

```
# 261.以图判树
from msilib.schema import PublishComponent
from typing import List


class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]
```

```
    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

class Solution:
    def validTree(n: int, edges:List[List[int]]):
        uf = UF(n)
        for edge in edges:
            u, v = edge[0], edge[1]
            if uf.connected(u, v):
                return False
            # 这条边不会产生环，可以是树的一部分
            uf.union(u, v)
        # 保证最后只有一个连通分量
        return uf.count() == 1
```

1135.最低成本连通所有城市(最小生成树)

```
# 1135.最低成本连通所有城市
from typing import List

class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]
```

```
    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

# 1135.最低成本连通所有城市
class Solution:
    def minimumCost(n:int, connections: List[List[int]]):
        uf = UF(n+1)
        # 最小生成树，按照weight升序排序
        connections = sorted(connections, key=lambda x: x[2])
        mst = 0.0
        for edge in connections:
            u, v, weight = edge[0], edge[1], edge[2]
            # 成环就跳过
            if uf.connected(u, v):
                continue
            mst += weight
            uf.union(u, v)
        # 保证所有节点连通，因为0没有被使用，额外占有一个，因此总共是2
        return mst if uf.count() == 2 else -1
```

1584.连接所有点的最小费用(最小生成树)

```
# 1584.连接所有点的最小费用
from typing import List
```

```python
class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

# 1584.连接所有点的最小费用
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n = len(points)
        edges = []
        for i in  range(n):
            for j in range(i+1, n):
                xi, yi = points[i][0], points[i][1]
                xj, yj = points[j][0], points[j][1]
                edges.append([i, j, abs(xi - xj) + abs(yi-yj)])
        edges = sorted(edges, key=lambda x: x[2])
        mst = 0.0
        uf = UF(n)
        for edge in edges:
            u, v, weight = edge[0], edge[1], edge[2]
            if uf.connected(u, v):
                continue
            mst += weight
            uf.union(u, v)
```

```
        return mst
```

743. 网络延迟时间(最短路径)

```
# 743. 网络延迟时间
from typing import List
import sys

class State:
    def __init__(self, id, distFromStart):
        self.id = id
        self.dist_from_start = distFromStart

class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        # 节点从1开始，所以需要一个n+1的领接表
        graph = [[] for _ in range(n+1)]
        # 构造图
        for edge in times:
            froms, tos, weight = edge[0], edge[1], edge[2]
            graph[froms].append([tos, weight])
        print(graph)
        distTo = self.dijkstra(k, graph)
        print(distTo)
        res = 0
        for dist in distTo[1:]:
            if dist == sys.maxsize:
                return -1
            res = max(res, dist)
        return res

    def dijkstra(self, start:int, graph:List[List[List[int]]]):
        # 用于记录节点的最小距离
        distTo = [sys.maxsize for _ in range(len(graph))]
        distTo[start] = 0

        pq = []
        # 从start节点开始进行BFS
        pq.append(State(start, 0))

        while len(pq) > 0:
            cur_state = pq.pop(0)
            cur_node_id = cur_state.id
            cur_dist_from_start = cur_state.dist_from_start

            if cur_dist_from_start > distTo[cur_node_id]:
                continue
            # 将cur_node 的相邻节点装入队列
            for neighbor in graph[cur_node_id]:
                next_node_id = neighbor[0]
```

```
                dist_to_next_node = distTo[cur_node_id] + neighbor[1]
                if distTo[next_node_id] > dist_to_next_node:
                    distTo[next_node_id] = dist_to_next_node
                    pq.append(State(next_node_id, dist_to_next_node))
        return distTo
```

1514. 概率最大的路径(最短路径)

```
# 1514. 概率最大的路径

from typing import List
import sys

class State:
    def __init__(self,cur_id, cur_prob):
        self.cur_id = cur_id
        self.cur_prob = cur_prob

class Solution:
    def maxProbability(self, n: int, edges: List[List[int]], succProb:
List[float], start: int, end: int) -> float:
        graph = self.buildGraph(n, edges, succProb)
        probTo = [0 for _ in range(n)]
        pq = []
        pq.append(State(start, 1))
        probTo[start] = 1
        # 广度优先遍历
        while len(pq) > 0:
            cur_state = pq.pop(0)
            cur_id = cur_state.cur_id
            cur_prob = cur_state.cur_prob

            if probTo[cur_id] > cur_prob:
                continue
            for neighbor in graph[cur_id]:
                next_id, prob = neighbor[0], neighbor[1]
                next_prob = cur_prob * prob
                if probTo[next_id] < next_prob:
                    probTo[next_id] = next_prob
                    pq.append(State(next_id, next_prob))
        return probTo[end]

    def buildGraph(self, n: int, edges:List[List[int]], succProb: List[float]) ->
List[List[int]]:
        graph = [[] for _ in range(n)]
        for edge, prob in zip(edges, succProb):
            u, v = edge[0], edge[1]
            graph[u].append([v, prob])
            graph[v].append([u, prob])
        return graph
```

1631. 最小体力消耗路径(最短路径)

```
import sys
from typing import List
import sys

# 1631. 最小体力消耗路径
class State:
    def __init__(self, x, y, delta):
        self.x = x
        self.y = y
        self.delta = delta

class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        if len(heights) == 1 and len(heights[0]) == 1:
            return 0
        row, col = len(heights), len(heights[0])
        graph = self.buildGraph(heights)
        pq = []
        delta_list = [[sys.maxsize for _ in range(col)] for _ in range(row)]

        delta_list[0][0] = 0
        pq.append(State(0, 0, 0))
        while len(pq) > 0:
            cur_state = pq.pop(0)
            if delta_list[cur_state.x][cur_state.y] < cur_state.delta:
                continue
            for neighbor in graph[cur_state.x][cur_state.y]:
                next_x, next_y, height = neighbor[0], neighbor[1], neighbor[2]
                val = max(height, delta_list[cur_state.x][cur_state.y])
                if delta_list[next_x][next_y] > val:
                    delta_list[next_x][next_y] = val
                    state = State(next_x, next_y, val)
                    pq.append(state)
        return delta_list[-1][-1]

    def buildGraph(self, heights):
        graph = [[[] for _ in range(len(heights[0]))] for _ in
range(len(heights))]
        for i in range(len(heights)):
            for j in range(len(heights[0])):
                val = heights[i][j]
                left, right, top, bot = None, None, None, None
                if j - 1 >= 0:
                    left = heights[i][j-1]
                    delta = abs(left - val)
                    graph[i][j].append([i, j-1, delta])
                if j + 1 < len(heights[0]):
                    right = heights[i][j+1]
                    delta = abs(right - val)
                    graph[i][j].append([i, j+1, delta])
```

```
                if i -1 >= 0:
                    top = heights[i-1][j]
                    delta = abs(top - val)
                    graph[i][j].append([i-1, j, delta])
                if i + 1 < len(heights):
                    bot = heights[i+1][j]
                    delta = abs(bot - val)
                    graph[i][j].append([i+1, j, delta])
        return graph
```

第三部分

回溯算法

17. 电话号码的字母组合

```
# 17. 电话号码的字母组合
from typing import List

class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        alpha_dict = {"2":"abc", "3":"def", "4":"ghi", "5":"jkl",
"6":"mno","7":"pqrs","8":"tuv","9":"wxyz"}
        result_list = []
        if len(digits) == 0:
            return []
        for alpha in alpha_dict.get(digits[0]):
            self.letterHelp(alpha_dict, alpha, result_list, digits[1:])
        return result_list

    def letterHelp(self, alpha_dict: dict, cur_alpha: str, result_list, digits:
str):
        if len(digits) == 0:
            result_list.append(cur_alpha)
            return None
        for alpha in alpha_dict.get(digits[0]):
            self.letterHelp(alpha_dict, cur_alpha + alpha, result_list,
digits[1:])
```

22. 括号生成

```
# 22. 括号生成
from typing import List

class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        cur_list = [['(', 1, 0]]
```

```
        cur_list = self.tran(n, 1, cur_list)
        result = [n[0] for n in cur_list]
        return result

    def tran(self, n, i , cur_list):
        if i == 2 * n:
            return cur_list
        tmp = []
        for ele in cur_list:
            out, left, right = ele[0], ele[1], ele[2]
            if left == n:
                out += ')'
                right += 1
                tmp.append([out, left, right])
            elif left == right:
                out += "("
                left += 1
                tmp.append([out, left, right])
            elif left > right:
                tmp.append([out + "(", left + 1, right])
                tmp.append([out + ")", left, right+1])
        cur_list = tmp
        return self.tran(n , i+1, cur_list)
```

37. 解数独

```
from typing import List

# 37. 解数独
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        self.recur_solve(board)
        return board

    def recur_solve(self, board):
        status = False
        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':
                    row_idx_list = self.get_row_list(i, j)
                    col_idx_list = self.get_col_list(i, j)
                    row_col_list = self.get_row_col_list(i, j)
                    remain_n_list = self.get_remain_n(row_idx_list + col_idx_list
+ row_col_list, board)
                    # 有符合的数字
                    status = False
                    for n in remain_n_list:
                        board[i][j] = n
```

```
                        next_status = self.recur_solve(board)
                        if next_status is False:
                            continue
                        else:
                            status = True
                            break
                    # 没有符合的数字
                    if status is False:
                        board[i][j] = "."
                        return status
        return True

    def get_remain_n(self, idx_list, board):
        n_list = [str(i) for i in range(1, 10)]
        for row_col in idx_list:
            i, j = row_col[0], row_col[1]
            if board[i][j]  != "." and board[i][j] in n_list:
                n_list.remove(board[i][j])
        return n_list

    def get_row_list(self, i, j):
        tmp = []
        for n in range(9):
            if n != j:
                tmp.append([i, n])
        return tmp

    def get_col_list(self, i, j):
        tmp = []
        for n in range(9):
            if n != i:
                tmp.append([n, j])
        return tmp

    def get_row_col_list(self, i, j):
        row_list, col_list = None, None
        if i < 3:
            row_list = [0, 3]
        elif i >= 3 and i < 6:
            row_list = [3, 6]
        elif i >= 6:
            row_list = [6, 9]

        if j < 3:
            col_list = [0, 3]
        elif j >= 3 and j < 6:
            col_list = [3, 6]
        elif j >= 6:
            col_list = [6, 9]

        row_col_list = []
        for row in range(row_list[0], row_list[1]):
            for col in range(col_list[0], col_list[1]):
                if i == row and j == col:
```

```
                    continue
                row_col_list.append([row, col])
        return row_col_list
```

39. 组合总和

```
# 39. 组合总和
from typing import List

class Solution:
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
        result_list = []
        pre_list = []
        self.getSum(candidates, result_list, target, pre_list)
        return result_list


    def getSum(self, candidates, result_list, target, pre_list):
        if target == 0:
            result_list.append(pre_list):
            return

        for n in candidates:
            if target - n >= 0:
                tmp_list = pre_list[::]
                tmp_list.append(n)
                self.getSum(cand_dict, result_list, target-n, pre_list)
        return
```

46. 全排列

```
# 46. 全排列
from typing import List

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        result, pre_list = [], []
        self.permute_help(nums, result, pre_list)
        return result

    def permute_help(self, nums, result, pre_list):
        if len(nums) == 0:
            result.append(pre_list)
            return
        for n in nums:
            tmp = pre_list[::]
            tmp.append(n)
            tmp_nums = nums[::]
```

```
            tmp_nums.remove(n)
            self.permute_help(tmp_nums, result, tmp)
        return
```

77. 组合

```
#   77. 组合
from typing import List

class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        result, pre_list = [], []
        self.combine_help(1, n+1, k, result, pre_list)
        return result

    def combine_help(self, i,j, k, result, pre_list):
        if k == 0:
            result.append(pre_list)
            return
        for n in range(i, j):
            tmp_list = pre_list[::]
            tmp_list.append(n)
            self.combine_help(n+1, j, k-1 ,  result, tmp_list)
        return
```

78. 子集

```
# 78. 子集
from typing import List

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        result = []
        self.sub_list(nums, result, [])
        return result

    def sub_list(self, nums, result, pre_list):
        if len(nums) == 0:
            result.append(pre_list[::])
            return
        result.append(pre_list[::])
        for i, n in enumerate(nums):
            tmp_list = pre_list[::]
            tmp_list.append(n)
            tmp_nums = nums[i+1:]
            self.sub_list(tmp_nums, result, tmp_list)
```

51. N 皇后

```python
#  51. N 皇后
from typing import List

class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        self.res = []
        board = [['.' for _ in range(n)] for _ in range(n)]
        self.backtrack(board, 0)
        return self.res

    def backtrack(self, board, row):
        if row == len(board):
            tmp_list = ["".join(ns) for ns in board]
            self.res.append(tmp_list)
            return
        n = len(board[row])
        for col in range(n):
            if not self.isValid(board, row, col):
                continue
            board[row][col] = 'Q'
            self.backtrack(board, row + 1)
            board[row][col] = '.'

    def isValid(self, board, row, col):
        n = len(board)
        # 列检查
        for i in range(n):
            if board[i][col] == 'Q':
                return False
        # 右上方
        i, j = row -1 , col + 1
        while i >= 0 and j < n:
            if board[i][j] == 'Q':
                return False
            i -= 1
            j += 1
        # 左上方
        i , j = row-1, col-1
        while i >= 0 and j >= 0:
            if board[i][j] == 'Q':
                return False
            i -= 1
            j -= 1
        return True
```

104.二叉树最大深度

```python
# Definition for a binary tree node.
from typing import Optional
```

```python
# 104.二叉树最大深度
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        return self.getDepth(root)

    def getDepth(self, root):
        if root is None:
            return 0
        return max(self.getDepth(root.left), self.getDepth(root.right)) + 1
```

494. 目标和

```python
# 494. 目标和
from typing import List

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        result = 0
        result = self.trac(nums, 0, result, 0, len(nums) , target)
        return result

    def trac(self, nums, base, result, i, n, target):
        if i == n:
            if base == target:
                result += 1
            return result
        result = self.trac(nums, base + nums[i], result, i+1, n, target)
        result = self.trac(nums, base - nums[i], result, i+1, n, target)
        return result

    def calc(self, n_list):
        base = 0
        for i in range(int(len(n_list)/2)):
            symbol, val = n_list[i*2], n_list[i*2 + 1]
            if symbol == '+':
                base += int(val)
            else:
                base -= int(val)
        return base

class Solution1:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        if len(nums) == 0:
            return 0
        self.memo = dict()
        return self.dp(nums, 0, target)
```

```
    def dp(self, nums, i, rest):
        if len(nums) == i:
            if rest == 0:
                return 1
            return 0
        key = str(i) + "," + str(rest)
        # 自底向上的过程中这个剪枝很关键
        if self.memo.get(key) is not None:
            return self.memo.get(key)
        result = self.dp(nums, i + 1, rest - nums[i]) + self.dp(nums, i+1, rest +
nums[i])
        self.memo[key] = result
        return result
```

698. 划分为k个相等的子集

```
#698. 划分为k个相等的子集
# leetcode 暴力搜索会超时
from typing import List

class Solution:
    def canPartitionKSubsets(self, nums: List[int], k: int) -> bool:
        if k > len(nums):
            return False
        sum_val = sum(nums)
        if sum_val % k != 0:
            return False
        used_list = [False for _ in nums]
        target = sum_val/k
        return self.backtrack(k, 0, nums, 0, used_list, target)

    def backtrack(self, k, bucket, nums, start, used_list, target):
        status = False
        if k == 0:
            status = True
            return status
        if bucket == target:
            return self.backtrack(k -1, 0, nums, 0, used_list, target)
        for i in range(start, len(nums)):
            if used_list[i]:
                continue
            if nums[i] + bucket > target:
                continue

            used_list[i] = True
            bucket += nums[i]
            if self.backtrack(k, bucket, nums, i+1, used_list, target):
                status = True
                return status
            used_list[i] = False
```

```
                bucket -= nums[i]
            return False
```

DFS算法

130. 被围绕的区域

```
# 130. 被围绕的区域
from typing import List

class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        if board is None or len(board) == 0:
            return
        row, col = len(board), len(board[0])
        for i in range(row):
            self.dfs(board, i, 0)
            self.dfs(board, i, col - 1)
        for j in range(col):
            self.dfs(board, 0, j)
            self.dfs(board, row-1, j)
        for i in range(row):
            for j in range(col):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                if board[i][j] == '-':
                    board[i][j] = 'O'

    def dfs(self, board, i, j):
        if (i< 0 or j < 0 or i >= len(board) or j >= len(board[0]) or board[i][j]
!= "O"):
            return
        board[i][j] = '-'
        self.dfs(board, i-1, j)
        self.dfs(board, i+1, j)
        self.dfs(board, i, j-1)
        self.dfs(board, i, j+1)
        return
```

200. 岛屿数量

```
# 200. 岛屿数量
from typing import List

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
```

```
        row, col = len(grid), len(grid[0])
        counter = 0
        for i in range(row):
            for j in range(col):
                if grid[i][j] == '1':
                    counter += 1
                    self.dfs(grid, i, j, row, col)
                    print(i, j, grid)
        return counter

    def dfs(self, grid, i, j, row, col):
        if grid[i][j] == '1':
            grid[i][j] = '0'
        else:
            return
        if i + 1 < row:
            self.dfs(grid, i+1, j, row, col)
        if i -1 >= 0:
            self.dfs(grid, i-1, j, row, col)
        if j + 1 < col:
            self.dfs(grid, i, j+1, row, col)
        if j - 1 >= 0:
            self.dfs(grid, i, j-1, row, col)
        return
```

694.不同的岛屿数量

```
# 694.不同的岛屿数量
class Solution:
    def numDistinctIslands(self, grid):
        islands_dict = dict()
        row, col = len(grid), len(grid[0])
        for i in range(row):
            for j in range(col):
                if grid[i][j] == "1":
                    res = []
                    self.dfs(grid, i, j, res, 666, row, col)
                    islands_dict[",".join(res)] = 1
        return len(islands_dict)

    def dfs(self, grid, i, j, res, dir, row, col):
        if grid[i][j] == '1':
            grid[i][j] = '0'
            res.append(str(dir))
        else:
            return
        if i + 1 < row:
            self.dfs(grid, i+1, j, res, 1, row, col)
        if i - 1 >= 0:
            self.dfs(grid, i-1, j, res, 2, row, col)
        if j + 1 < col:
```

```
            self.dfs(grid, i, j+1, res, 3, row, col)
        if j - 1 >= 0:
            self.dfs(grid, i, j-1, res, 4, row, col)
        return
```

695. 岛屿的最大面积

```
# 695. 岛屿的最大面积
from typing import List


class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        row, col = len(grid), len(grid[0])
        area_list = []
        for i in range(row):
            for j in range(col):
                if grid[i][j] == 1:
                    area = self.dfs(grid, i, j, row, col, 0)
                    area_list.append(area)
        return max(area_list)

    def dfs(self, grid, i, j, row, col, area):
        if grid[i][j] == 1:
            grid[i][j] = 0
            area += 1
        else:
            return area
        if i + 1 < row:
            area = self.dfs(grid, i+1, j, row, col, area)
        if i - 1 >= 0:
            area = self.dfs(grid, i-1, j, row, col, area)
        if j + 1 < col:
            area = self.dfs(grid, i, j+1, row, col, area)
        if j - 1 >= 0:
            area = self.dfs(grid, i, j-1, row, col, area)
        return area
```

1020.飞地的数量

```
# 1020.飞地的数量
from typing import List


class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        row, col = len(grid), len(grid[0])
        for j in range(col):
            if grid[0][j] == 1:
```

```
                self.dfs(grid, 0, j, row, col)
            if grid[row-1][j] == 1:
                self.dfs(grid, row-1, j, row, col)

        for i in range(row):
            if grid[i][0] == 1:
                self.dfs(grid, i, 0, row, col)
            if grid[i][col-1] == 1:
                self.dfs(grid, i, col-1, row, col)

        counter = 0
        for i in range(row):
            for j in range(col):
                if grid[i][j] == 1:
                    counter += 1
        return counter

    def dfs(self, grid, i, j, row, col):
        if grid[i][j] == 1:
            grid[i][j] = 0
        else:
            return
        if i + 1 < row:
            self.dfs(grid, i + 1, j, row, col)
        if i - 1 >= 0:
            self.dfs(grid, i-1, j, row, col)
        if j + 1 < col:
            self.dfs(grid, i, j+1, row, col)
        if j - 1 >= 0:
            self.dfs(grid, i, j-1, row, col)
        return
```

1254. 统计封闭岛屿的数目

```
# 1254. 统计封闭岛屿的数目
from typing import List

class Solution:
    def closedIsland(self, grid: List[List[int]]) -> int:
        row, col = len(grid), len(grid[0])
        for j in range(col):
            if grid[0][j] == 0:
                self.dfs(grid, 0, j, row, col)
            if grid[row-1][j] == 0:
                self.dfs(grid, row-1, j, row, col)

        for i in range(row):
            if grid[i][0] ==  0:
                self.dfs(grid, i, 0, row, col)
            if grid[i][col-1] == 0:
                self.dfs(grid, i, col-1, row, col)
```

```
        counter = 0
        for i in range(1, row-1):
            for j in range(1, col-1):
                if grid[i][j] == 0:
                    self.dfs(grid, i, j, row, col)
                     counter += 1
        return counter

    def dfs(self, grid, i, j, row, col):
        if grid[i][j] == 0:
            grid[i][j] = 1
        else:
            return
        if i + 1 < row:
            self.dfs(grid, i+1, j, row, col)
        if i - 1 >= 0:
            self.dfs(grid, i-1, j, row, col)
        if j + 1 < col:
            self.dfs(grid, i, j+1, row, col)
        if j - 1 >= 0:
            self.dfs(grid, i, j-1, row, col)
        return
```

1905. 统计子岛屿

```
# 1905. 统计子岛屿
from typing import List

class Solution:
    def countSubIslands(self, grid1: List[List[int]], grid2: List[List[int]]) ->
int:
        row, col = len(grid1), len(grid1[0])
        counter = 0
        for i in range(row):
            for j in range(col):
                if grid2[i][j] == 1:
                    status = self.findIsland(grid1, grid2,i, j, row, col, True)
                    print("mark1")
                    if status:
                        counter += 1
        return counter

    def findIsland(self, grid1, grid2, i, j, row, col, status):
        if grid2[i][j] == 1:
            grid2[i][j] = 0
            if grid1[i][j] == 0:
                # print("grid1:", i, j)
                status = status and False
        else:
            return status
```

```
        if i + 1 < row:
            status = self.findIsland(grid1, grid2, i+1, j, row, col, status)
        if i -1 >= 0:
            status = self.findIsland(grid1, grid2, i-1, j, row, col, status)
        if j + 1 < col:
            status = self.findIsland(grid1, grid2, i, j+1, row, col, status)
        if j - 1 >= 0:
            status = self.findIsland(grid1, grid2, i, j-1, row, col, status)
        return status
```

BFS算法

102. 二叉树的层序遍历

```
# Definition for a binary tree node.
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# 102. 二叉树的层序遍历
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_line(node_list, result)
        return result

    def recur_line(self, node_list, result):
        if len(node_list) == 0:
            return
        tmp_list = []
        lines = []
        for node in node_list:
            if node.left is not None:
                tmp_list.append(node.left)
            if node.right is not None:
                tmp_list.append(node.right)
            lines.append(node.val)
        result.append(lines)
        self.recur_line(tmp_list, result)
```

103. 二叉树的锯齿形层序遍历

```python
# Definition for a binary tree node.
# 103. 二叉树的锯齿形层序遍历
from typing import List


class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        node_list = [root]
        result = []
        self.recur_node(node_list, result, 1)
        return result

    def recur_node(self, node_list, result, depth):
        if len(node_list) == 0:
            return
        sub_nodes = []
        lines = []
        for node in node_list:
            if node.left is not None:
                sub_nodes.append(node.left)
            if node.right is not None:
                sub_nodes.append(node.right)
            lines.append(node.val)
        if depth % 2 == 0:
            lines = lines[::-1]
        result.append(lines)
        self.recur_node(sub_nodes, result, depth+1)
```

107. 二叉树的层序遍历 II

```python

# Definition for a binary tree node.
from typing import List

# 107. 二叉树的层序遍历 II
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
```

```
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_lines(node_list, result)
        return result

    def recur_lines(self, node_list, res):
        if len(node_list) == 0:
            return
        new_nodes = []
        tmp = []
        for node in node_list:
            if node.left is not None:
                new_nodes.append(node.left)
            if node.right is not None:
                new_nodes.append(node.right)
            tmp.append(node.val)
        self.recur_lines(new_nodes, res)
        res.append(tmp)
        return
```

111. 二叉树的最小深度

```
# Definition for a binary tree node.
# 111. 二叉树的最小深度
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def minDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        return self.getDepth(root)

    def getDepth(self, root):
        if root.left is None and root.right is None:
            return 1
        elif root.left is None and root.right is not None:
            return self.getDepth(root.right) + 1
        elif root.left is not None and root.right is None:
            return self.getDepth(root.left) + 1
        elif root.left is not None and root.right is not None:
            return min(self.getDepth(root.left), self.getDepth(root.right)) + 1
```

752. 打开转盘锁

```python
# 752. 打开转盘锁
from typing import List

class Solution:
    def openLock(self, deadends: List[str], target: str) -> int:
        dead_dict = {d:1 for d in deadends}
        visited_dict = dict()
        q = []
        step = 0
        q.append(['0', '0', '0', '0'])
        visited_dict["0000"] = 1
        while len(q) > 0:
            sz = len(q)
            for i in range(sz):
                cur = q.pop(0)
                cur_str = ''.join(cur)
                if dead_dict.get(cur_str) is not None:
                    continue
                if cur_str == target:
                    return step

                for j in range(4):
                    up = self.plusOne(cur[::], j)
                    up_str = ''.join(up)
                    if visited_dict.get(up_str) is None:
                        q.append(up)
                        visited_dict[up_str] = 1

                    down = self.minusOne(cur[::], j)
                    down_str = ''.join(down)
                    if visited_dict.get(down_str) is None:
                        q.append(down)
                        visited_dict[down_str] = 1
            step += 1
        return -1

    def plusOne(self, s, j):
        if s[j] == '9':
            s[j] = '0'
        else:
            s[j] = str(int(s[j]) + 1)
        return s

    def minusOne(self, s, j):
        if s[j] == '0':
            s[j] = '9'
        else:
            s[j] = str(int(s[j]) -1)
        return s
```

773. 滑动谜题

```python
# 773. 滑动谜题
from typing import List

class Solution:
    def slidingPuzzle(self, board: List[List[int]]) -> int:
        row, col = len(board), len(board[0])
        start_row, start_col = None, None
        for i in range(row):
            for j in range(col):
                if board[i][j] == 0:
                    start_row, start_col = i, j

        q_list = [[[start_row, start_col, board]]]
        board_dict = dict()

        counter = 0
        while len(q_list) > 0:
            cur_list =  q_list.pop(0)
            tmp_list = []
            for cur in cur_list:
                cur_x, cur_y, cur_board = cur[0], cur[1], cur[2]
                if self.checkBoard(cur_board):
                    return counter
                if cur_x + 1 < row:
                    tmp_board = [b[::] for b in cur_board]
                    tmp_board[cur_x][cur_y], tmp_board[cur_x+1][cur_y] =
tmp_board[cur_x+1][cur_y], tmp_board[cur_x][cur_y]
                    tmp_board_str = self.castStr(tmp_board)
                    if board_dict.get(tmp_board_str) is None:
                        tmp_list.append([cur_x+1, cur_y, tmp_board])
                        board_dict[tmp_board_str] = 1
                if cur_x - 1 >= 0:
                    tmp_board = [b[::] for b in cur_board]
                    tmp_board[cur_x-1][cur_y], tmp_board[cur_x][cur_y] =
tmp_board[cur_x][cur_y], tmp_board[cur_x-1][cur_y]
                    tmp_board_str = self.castStr(tmp_board)
                    if board_dict.get(tmp_board_str) is None:
                        tmp_list.append([cur_x-1, cur_y, tmp_board])
                        board_dict[tmp_board_str] = 1
                if cur_y + 1 < col:
                    tmp_board = [b[::] for b in cur_board]
                    tmp_board[cur_x][cur_y], tmp_board[cur_x][cur_y+1] =
tmp_board[cur_x][cur_y+1], tmp_board[cur_x][cur_y]
                    tmp_board_str = self.castStr(tmp_board)
                    if board_dict.get(tmp_board_str) is None:
                        tmp_list.append([cur_x, cur_y+1, tmp_board])
                        board_dict[tmp_board_str] = 1
                if cur_y - 1 >= 0:
                    tmp_board = [b[::] for b in cur_board]
                    tmp_board[cur_x][cur_y], tmp_board[cur_x][cur_y-1] =
tmp_board[cur_x][cur_y-1], tmp_board[cur_x][cur_y]
                    tmp_board_str = self.castStr(tmp_board)
                    if board_dict.get(tmp_board_str) is None:
```

```
                        tmp_list.append([cur_x, cur_y-1, tmp_board])
                        board_dict[tmp_board_str] = 1
            if len(tmp_list) > 0:
                q_list.append(tmp_list)
                counter += 1
        return -1

    def checkBoard(self, board):
        status = True
        for n1, n2 in zip(board[0], [1, 2, 3]):
            if n1 != n2:
                status = False
        for n1, n2 in zip(board[1],[4, 5, 0]):
            if n1 != n2:
                status = False
        return status

    def castStr(self, board):
        n_list = []
        for n in board:
            n_list.append("".join([str(i) for i in n]))
        return "".join(n_list)
```

一维DP

45. 跳跃游戏 II

```
# 45. 跳跃游戏 II
from typing import List

class Solution:
    def jump(self, nums: List[int]) -> int:
        n = len(nums)
        end, farthest = 0, 0
        jump_count = 0
        for i in range(n-1):
            farthest = max(nums[i]+i, farthest)
            if end == i:
                jump_count += 1
                end = farthest
        return jump_count
```

55. 跳跃游戏

```
# 55. 跳跃游戏
from typing import List

class Solution:
    def canJump(self, nums: List[int]) -> bool:
```

```
        farthest = 0
        end = 0
        for i in range(len(nums)-1):
            farthest = max(nums[i] + i, farthest)
            # 碰上0值就直接返回为False
            if farthest <= i:
                return False
            if end == i:
                end = farthest
        return farthest >= len(nums) - 1
```

53. 最大子数组和

```
# 53. 最大子数组和
import sys
from typing import List
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0
        dp = [0 for _ in range(n)]
        dp[0] = nums[0]
        for i in range(1, n):
            dp[i] = max(nums[i], nums[i] + dp[i-1])
        return max(dp)
```

70. 爬楼梯

```
# 70. 爬楼梯
class Solution:
    def climbStairs(self, n: int) -> int:
        if n <= 2:
            return n
        dp = [0 for _ in range(n+1)]
        dp[1] = 1
        dp[2] = 2
        for i in range(3, n+1):
            dp[i] = dp[i-1] + dp[i-2]
        return dp[n]

# 会超时
class Solution1:
    def climbStairs(self, n: int) -> int:
        return self.climbStairs(n - 1) + self.climbStairs(n-2) if n > 2 else n
```

198. 打家劫舍

```python
# 198. 打家劫舍
from typing import List

class Solution:
    def rob(self, nums: List[int]) -> int:
        dp = [ 0 for _ in range(len(nums))]
        if len(nums) == 0:
            return 0
        if len(nums) == 1:
            return nums[0]
        dp[0] = nums[0]
        dp[1] = max(dp[0], nums[1])
        if len(nums) == 1:
            return dp[0]
        if len(nums) == 2:
            return dp[1]
        for i in range(2, len(nums)):
            dp[i] = max(nums[i] + dp[i-2], dp[i-1])
        return max(dp)
```

213. 打家劫舍 II

```python
# 213. 打家劫舍 II
from typing import List

class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return nums[0]
        val_1 = self.dp(nums, 1, len(nums)-1)
        val_2 = self.dp(nums, 0, len(nums)-2)
        return max(val_1, val_2)

    def dp(self, nums, start, end):
        if end-start == 0:
            return nums[start]
        dp = [0 for _ in range(len(nums))]
        dp[start] = nums[start]
        dp[start+1] = max(nums[start+1], dp[start])
        for i in range(start+2, end+1):
            dp[i] = max(nums[i] + dp[i-2], dp[i-1])
        return max(dp)

class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 1:
            return nums[0]
        memo1 = [-1 for _ in range(n)]
        memo2 = [-1 for _ in range(n)]
```

```
        return max(self.dp(nums, 0, n-2, memo1), self.dp(nums, 1, n-1, memo2))

    def dp(self, nums, start, end, memo):
        if start > end:
            return 0
        if memo[start] != -1:
            return memo[start]
        res = max(self.dp(nums, start+2, end, memo) + nums[start], self.dp(nums,
start+1, end, memo))
        memo[start] = res
        return res
```

337. 打家劫舍 III

```
# Definition for a binary tree node.
# 337. 打家劫舍 III
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def rob(self, root: TreeNode) -> int:
        self.memo = dict()
        res = self.robHelp(root)
        return res

    def robHelp(self, root):
        if root is None:
            return 0
        if self.memo.get(root) is not None:
            return self.memo.get(root)
        do_it = root.val
        if root.left is not None:
            do_it += self.robHelp(root.left.left) + self.robHelp(root.left.right)
        if root.right is not None:
            do_it += self.robHelp(root.right.left) +
self.robHelp(root.right.right)
        not_do = self.robHelp(root.left) + self.robHelp(root.right)
        res = max(do_it, not_do)
        self.memo[root] = res
        return res
```

300. 最长递增子序列

```
# 300. 最长递增子序列
from typing import List
```

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        dp = [1 for _ in nums]
        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j]+1)
        return max(dp)

class Solution1:
    def lengthOfLIS(self, nums):
        maxL = 0
        # 存放当前的递增序列的潜在数据
        dp = [0 for _ in nums]
        for num in nums:
            lo, hi = 0, maxL
            # 二分查找，并替换，这一步维护dp的本质是维护一个潜在的递增序列。非常trick
            while lo < hi:
                mid = lo + (hi-lo)/2
                if dp[mid] < num:
                    lo = mid + 1
                else:
                    hi = mid
            dp[lo] = num
            # 若是接在最后面，则连续递增序列张长了
            if lo == maxL:
                maxL += 1
        return maxL
```

322. 零钱兑换

```python
# 322. 零钱兑换
import sys
from typing import List
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        if amount == 0:
            return 0
        dp = [sys.maxsize for _ in range(amount+1)]
        for i in range(1, len(dp)):
            for coin in coins:
                if i == coin:
                    dp[i] = 1
                else:
                    if i > coin:
                        dp[i] = min(dp[i-coin] + 1, dp[i])
        return -1 if dp[-1] == sys.maxsize else dp[-1]
```

354. 俄罗斯套娃信封问题

```python
# 354. 俄罗斯套娃信封问题
from functools import cmp_to_key
from typing import List
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        def sort_item(item1, item2):
            if item1[0] == item2[0]:
                return item2[1] - item1[1]
            else:
                return item1[0] - item2[0]
        envelopes_list = sorted(envelopes, key=cmp_to_key(sort_item))
        height_list = [item[1] for item in envelopes_list]
        # O(N*2)的方式记录dp会超时，但是这个方式确实太fancy了
        return self.lengthOfLIS(height_list)

    def lengthOfLIS(self, nums):
        piles, n = 0, len(nums)
        top = [0 for _ in range(n)]
        for i in range(n):
            poker = nums[i]
            left, right = 0, piles
            while left < right:
                mid = int((left + right)/2)
                if top[mid] >= poker:
                    right = mid
                else:
                    left = mid + 1
            if left == piles:
                piles += 1
            top[left] = poker
        return piles
```

二维DP

10. 正则表达式匹配

```python
# 10. 正则表达式匹配
class Solution:
    def isMatch(self, s, p):
        self.memo = dict()
        return self.dp(s, p, 0, 0)

    def dp(self, s, p, i, j):
        m , n = len(s), len(p)
        if j == n:
            return i == m
        if i == m:
            if (n -j) % 2 == 1:
                return False
            while j+1 < n:
```

```
                if p[j+1] != "*":
                    return False
                j += 2
            return True

        key = str(i) + "," + str(j)
        if self.memo.get(key) is not None:
            return self.memo.get(key)
        res = False
        if s[i] == p[j] or p[j] == '.':
            if j < n - 1 and p[j+1] == '*':
                res = self.dp(s, p, i, j+2) or self.dp(s, p, i+1, j)
            else:
                res = self.dp(s, p, i+1, j+1)
        else:
            if j < n - 1 and p[j+1] == '*':
                res = self.dp(s, p, i, j+2)
            else:
                res = False
        self.memo[key] = res
        return res
```

62. 不同路径

```
# 62. 不同路径
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[0 for _ in range(n)] for _ in range(m)]
        for j in range(n):
            dp[0][j] = 1
        for i in range(m):
            dp[i][0] = 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[m-1][n-1]
```

64. 最小路径和

```
# 64. 最小路径和
from typing import List

class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        row, col = len(grid), len(grid[0])
        dp = [[0 for _ in range(col)] for _ in range(row)]
        for j in range(col):
            if j - 1 >= 0:
                dp[0][j] =  grid[0][j] + dp[0][j-1]
```

```
            else:
                dp[0][j] =  grid[0][j]

        for i in range(row):
            if i - 1 >= 0:
                dp[i][0] = grid[i][0] + dp[i-1][0]
            else:
                dp[i][0] = grid[i][0]

        for i in range(1, row):
            for j in range(1, col):
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
        return dp[row-1][col-1]
```

72.编辑距离

```
# 72.编辑距离
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m , n = len(word1), len(word2)
        # m * n 的数组保存的是，对应的word1[:i] 字符串和 对应的word2[:j]字符串的编辑距
离
        dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
        for i in range(1, m+1):
            dp[i][0] = i
        for j in range(1, n+1):
            dp[0][j] = j
        for i in range(1, m+1):
            for j in range(1, n+1):
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    # i-1, j 删除; i,j-1 插入； i-1,j-1 替换
                    # i-1, j 删除，是指：word1[:i]删除 对应的i的字符，所以操作+1 并取
上一步的编辑距离
                    # i,j-1 插入是指，插入当前word1[i]的位置字符为对应Word2[j] 所以操
作+1,插入之前的word1的字符串保持不变，
                    # 相当于插入到之前字符串的i+1的位置，因此取上一步的编辑距离，i,j-1
                    # i-1, j-1 替换操作，则+1，然后直接取上一步的编辑距离。
                    dp[i][j] = min(dp[i-1][j]+1, min(dp[i][j-1]+1, dp[i-1][j-1] +
1))
        return dp[m][n]
```

121. 买卖股票的最佳时机

```
# 121. 买卖股票的最佳时机
from typing import List

class Solution:
```

```
    def maxProfit(self, prices: List[int]) -> int:
        delta_list = []
        for p1, p2 in zip(prices[:-1], prices[1:]):
            delta_list.append(p2-p1)
        cum_val = 0
        max_val = 0
        for delta in delta_list:
            if cum_val + delta > 0:
                cum_val += delta
                max_val = max(cum_val, max_val)
            else:
                cum_val = 0
        return max_val

# DP的方式
class Solution1:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0, 0] for _ in prices]
        # 卖出
        dp[0][0] = 0
        # 买入
        dp[0][1] = -prices[0]
        for i in range(1, len(prices)):
            # 卖出   =  max(保持 ,   前一天买入 + 当天卖出  )
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
            # 买入 =    max( 保持,  当天买入 )
            dp[i][1] = max(dp[i-1][1], -prices[i])
        return dp[-1][0]
```

122. 买卖股票的最佳时机 II

```
#122. 买卖股票的最佳时机 II
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        delta_list = []
        for p1, p2 in zip(prices[:-1], prices[1:]):
            delta_list.append(p2-p1)
        val_list = list(filter(lambda x: x> 0 , delta_list))
        return sum(val_list)

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0, 0] for _ in 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]
```

123. 买卖股票的最佳时机 III

```
#123. 买卖股票的最佳时机 III
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        max_k = 2
        dp = [ [[0, 0] for _ in range(max_k+1)] for _ in prices]
        for k in range(max_k, 0, -1):
            dp[0][k][0] = 0
            dp[0][k][1] = -prices[0]

        for i in range(1, len(prices)):
            for k in range(max_k, 0, -1):
                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][max_k][0]
```

188. 买卖股票的最佳时机 IV

```
# 188. 买卖股票的最佳时机 IV
from typing import List

class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        n = len(prices)
        if n <= 0:
            return 0

        if k > n/2:
            return self.profitNoLimit(prices)

        dp = [[[0, 0] for _ in range(k+1)] for _ in prices]
        for i in range(k, 0, -1):
            dp[0][i][0] = 0
            dp[0][i][1] = -prices[0]
        for i in range(1, len(prices)):
            for j in range(k, 0, -1):
                dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i])
                dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices[i])
        return dp[-1][k][0]

    def profitNoLimit(self, prices):
        dp = [[0, 0] for _ in 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]
```

309. 最佳买卖股票时机含冷冻期

```
# 309. 最佳买卖股票时机含冷冻期
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0, 0, 0] for _ in prices]
        dp[0][0] = 0
        dp[0][1] = -prices[0]
        dp[0][2] = 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][2] - prices[i])
            dp[i][2] = dp[i-1][0]
        return max(dp[-1][0], dp[-1][2])
```

714. 买卖股票的最佳时机含手续费

```
# 714. 买卖股票的最佳时机含手续费
from typing import List

class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        dp = [[0, 0] for _ in prices]
        dp[0][0] = 0
        dp[0][1] = -prices[0] - fee
        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] - fee)
        return dp[-1][0]
```

174. 地下城游戏

```
# 174. 地下城游戏
from typing import List

import sys
class Solution1:
    def calculateMinimumHP(self, dungeon: List[List[int]]) -> int:
        m , n = len(dungeon), len(dungeon[0])
```

```
        self.memo = [[-1 for _ in range(n)] for _ in range(m)]
        return self.dp(dungeon, 0, 0)

    def dp(self, dungeon, i, j):
        m, n = len(dungeon), len(dungeon[0])
        if i == m-1 and j == n-1:
            return 1 if dungeon[i][j] >= 0 else 1 - dungeon[i][j]

        if i == m or j == n:
            return sys.maxsize

        if self.memo[i][j] != -1:
            return self.memo[i][j]

        res = min(self.dp(dungeon, i, j+1), self.dp(dungeon, i+1, j)) - dungeon[i]
[j]

        self.memo[i][j] = 1 if res <= 0 else res

        return self.memo[i][j]
```

312. 戳气球

```
# 312. 戳气球
from typing import List

class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        points = [0 for _ in range(n+2)]
        points[0] = 1
        points[n+1] = 1
        for i in range(1, n+1):
            points[i] = nums[i-1]
        dp = [[0 for _ in range(n+2)] for _ in range(n+2)]
        for i in range(n, -1, -1):
            for j in range(i+1, n+2):
                for k in range(i+1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + points[i] *
points[j] * points[k])
        return dp[0][n+1]
```

416.分割等和子集

```
from typing import List

# 416.分割等和子集
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
```

```
        sum_val = sum(nums)
        if sum_val % 2 != 0:
            return False
        n = len(nums)
        sum_val = int(sum_val/2)
        dp = [[False for _ in range(sum_val + 1)] for _ in range(n+1)]
        for i in range(n+1):
            dp[i][0] = True

        for i in range(1, n+1):
            for j in range(1, sum_val+1):
                if j - nums[i-1] < 0:
                    dp[i][j] = dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]
        return dp[n][sum_val]
```

494. 目标和

```
# 494. 目标和
from typing import List

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        result = 0
        result = self.trac(nums, 0, result, 0, len(nums) , target)
        return result

    def trac(self, nums, base, result, i, n, target):
        if i == n:
            if base == target:
                result += 1
            return result
        result = self.trac(nums, base + nums[i], result, i+1, n, target)
        result = self.trac(nums, base - nums[i], result, i+1, n, target)
        return result

    def calc(self, n_list):
        base = 0
        for i in range(int(len(n_list)/2)):
            symbol, val = n_list[i*2], n_list[i*2 + 1]
            if symbol == '+':
                base += int(val)
            else:
                base -= int(val)
        return base

class Solution1:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        if len(nums) == 0:
            return 0
```

```
        self.memo = dict()
        return self.dp(nums, 0, target)

    def dp(self, nums, i, rest):
        if len(nums) == i:
            if rest == 0:
                return 1
            return 0
        key = str(i) + "," + str(rest)
        # 自底向上的过程中这个剪枝很关键
        if self.memo.get(key) is not None:
            return self.memo.get(key)
        result = self.dp(nums, i + 1, rest - nums[i]) + self.dp(nums, i+1, rest +
nums[i])
        self.memo[key] = result
        return result
```

514.自由之路

```
# 514.自由之路
import sys

class Solution:
    def findRotateSteps(self, ring: str, key: str) -> int:
        m, n = len(ring), len(key)
        self.charToIndex = dict()
        self.memo = [[0 for _ in range(n)] for _ in range(m)]
        # 索引初始化
        for i in range(m):
            if self.charToIndex.get(ring[i]) is None:
                self.charToIndex[ring[i]] = [i]
            else:
                self.charToIndex[ring[i]].append(i)
        return self.dp(ring, 0, key, 0)

    def dp(self, ring, i, key, j):
        if j == len(key):
            return 0
        if self.memo[i][j] != 0:
            return self.memo[i][j]
        n = len(ring)
        res = sys.maxsize
        # 找到最小的字母
        for k in self.charToIndex.get(key[j]):
            delta = abs(k - i)
            # 正向和逆向都可以，找到最小的
            delta = min(delta, n - delta)
            subProblem = self.dp(ring, k, key, j+1)
            res = min(res, 1 + delta +subProblem)
        self.memo[i][j] = res
        return res
```

```
if __name__ == "__main__":
    rings = "godding"
    key = "godding"
    sol = Solution()
    ret = sol.findRotateSteps(rings, key)
    print(ret)
```

518. 零钱兑换 II

```
# 518. 零钱兑换 II
from typing import List

class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [[0 for _ in range(amount+1)] for _ in range(len(coins)+ 1)]
        for i in range(len(coins) + 1):
            dp[i][0] = 1

        for i in range(1, len(coins)+1):
            for j in range(1, amount + 1):
                coin = coins[i-1]
                if j - coin < 0:
                    dp[i][j] = dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j] + dp[i][j-coin]
        return dp[-1][-1]
```

583. 两个字符串的删除操作

```
# 583. 两个字符串的删除操作
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        self.memo = [[-1 for _ in range(len(word2))] for _ in range(len(word1))]
        res = self.dp(word1, 0, word2, 0)
        return res

    def dp(self, word1, i, word2, j):
        if i == len(word1) and j == len(word2):
            print("mark1")
            return 0
        elif i == len(word1) and j < len(word2):
            return len(word2) - j
        elif i < len(word1) and j == len(word2):
            return len(word1) - i

        if self.memo[i][j] != -1:
            return self.memo[i][j]
```

```
        if word1[i] == word2[j]:
            self.memo[i][j] = self.dp(word1, i+1, word2, j+1)
        else:
            self.memo[i][j] = min(self.dp(word1, i+1, word2, j), self.dp(word1, i,
word2, j+1)) + 1

        return self.memo[i][j]

if __name__ == "__main__":
    word1 = "sea"
    word2 = "eat"
    sol = Solution()
    ret = sol.minDistance(word1, word2)
    print(ret)
```

712. 两个字符串的最小ASCII删除和

```
# 712. 两个字符串的最小ASCII删除和
class Solution:
    def minimumDeleteSum(self, s1: str, s2: str) -> int:
        self.memo = [[0 for _ in s2] for _ in s1]
        return self.dp(s1, 0, s2, 0)

    def dp(self, s1, i, s2, j):
        if i == len(s1) and j == len(s2):
            return 0
        elif i == len(s1) and j < len(s2):
            val = 0
            while j < len(s2):
                val += ord(s2[j])
                j += 1
            return val
        elif i < len(s1) and j == len(s2):
            val = 0
            while i < len(s1):
                val += ord(s1[i])
                i += 1
            return val

        if self.memo[i][j] != 0:
            return self.memo[i][j]

        if s1[i] == s2[j]:
            self.memo[i][j] = self.dp(s1, i+1, s2, j+1)
        else:
            left = ord(s1[i]) + self.dp(s1, i+1, s2, j)
            right = ord(s2[j]) + self.dp(s1, i, s2, j+1)
            self.memo[i][j] = min(left, right)
        return self.memo[i][j]
```

1143. 最长公共子序列

```
# 1143. 最长公共子序列
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        self.memo =[[0 for _ in text2] for _ in text1]
        return self.dp(text1, 0, text2, 0)

    def dp(self, text1, i, text2, j):
        if i == len(text1) or j == len(text2):
            return 0
        if self.memo[i][j] != 0:
            return self.memo[i][j]
        if text1[i] == text2[j]:
            self.memo[i][j] = 1 + self.dp(text1, i+1, text2, j+1)
        else:
            self.memo[i][j] = max(self.dp(text1, i+1, text2, j), self.dp(text1, i,
text2, j+1))
        return self.memo[i][j]
```

787.K站中专内最便宜的航班

```
# 787.K站中专内最便宜的航班
from typing import List
import sys

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst:
int, k: int) -> int:
        edges = self.getEdges(flights, n)
        self.memo = [[-2 for _ in range(n)] for i in range(k+2)]
        return self.bfs(edges, src, dst, k+1)

    def bfs(self, edges, src, dst, k):
        if src == dst:
            return 0
        if k == 0:
            return -1

        if self.memo[k][src] != -2:
            return self.memo[k][src]
        res = sys.maxsize
        for edge in edges[src]:
            sub_fee = self.bfs(edges, edge[0], dst, k-1)
            if sub_fee != -1:
                res = min(sub_fee + edge[1], res)
        res = -1 if res == sys.maxsize else res
        self.memo[k][src] = res
        return res
```

```
    def getEdges(self, flights, n):
        edges = [[] for _ in range(n)]
        for f in flights:
            start, end,  fee = f[0], f[1], f[2]
            edges[start].append([end, fee])
        return edges
```

887. 鸡蛋掉落/810

```
import sys
sys.setrecursionlimit(100000) #例如这里设置为十万
# 887. 鸡蛋掉落/810

class Solution:
    def superEggDrop(self, k: int, n: int) -> int:
        dp = [[0 for _ in range(n+1)] for _ in range(k+1)]
        m = 0
        while dp[k][m] < n:
            m += 1
            for i in range(1, k+1):
                dp[i][m] = dp[i][m-1] + dp[i-1][m-1] + 1
        return m
```

931. 下降路径最小和

```
# 931. 下降路径最小和
import sys
from typing import List
class Solution:
    def minFallingPathSum(self, matrix: List[List[int]]) -> int:
        row ,col = len(matrix), len(matrix[0])
        self.memo = [[float("inf") for _ in range(col)] for _ in range(row)]
        res_list = []
        for j in range(col):
            res = self.minPath(matrix, 0, j, row, col)
            res_list.append(res)
        return min(res_list)

    def minPath(self, matrix, i, j, row, col):
        if i == row - 1:
            self.memo[i][j] = matrix[i][j]
            return matrix[i][j]

        if self.memo[i][j] != float("inf"):
            return self.memo[i][j]
        res = float("INF")
        res = min(res, self.minPath(matrix, i+1, j, row, col))
        if j - 1 >= 0:
            res = min(res, self.minPath(matrix, i+1, j-1, row, col))
```

```
        if j + 1 < col:
            print(j)
            res = min(res, self.minPath(matrix, i+1, j+1, row, col))
        res += matrix[i][j]
        self.memo[i][j] = res
        return res
```

背包问题

416.分割等和子集

```
from typing import List

# 416.分割等和子集
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        sum_val = sum(nums)
        if sum_val % 2 != 0:
            return False
        n = len(nums)
        sum_val = int(sum_val/2)
        dp = [[False for _ in range(sum_val + 1)] for _ in range(n+1)]
        for i in range(n+1):
            dp[i][0] = True

        for i in range(1, n+1):
            for j in range(1, sum_val+1):
                if j - nums[i-1] < 0:
                    dp[i][j] = dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]
        return dp[n][sum_val]
```

494. 目标和

```
# 494. 目标和
from typing import List

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        result = 0
        result = self.trac(nums, 0, result, 0, len(nums) , target)
        return result

    def trac(self, nums, base, result, i, n, target):
        if i == n:
            if base == target:
                result += 1
            return result
```

```
        result = self.trac(nums, base + nums[i], result, i+1, n, target)
        result = self.trac(nums, base - nums[i], result, i+1, n, target)
        return result

    def calc(self, n_list):
        base = 0
        for i in range(int(len(n_list)/2)):
            symbol, val = n_list[i*2], n_list[i*2 + 1]
            if symbol == '+':
                base += int(val)
            else:
                base -= int(val)
        return base

class Solution1:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        if len(nums) == 0:
            return 0
        self.memo = dict()
        return self.dp(nums, 0, target)

    def dp(self, nums, i, rest):
        if len(nums) == i:
            if rest == 0:
                return 1
            return 0
        key = str(i) + "," + str(rest)
        # 自底向上的过程中这个剪枝很关键
        if self.memo.get(key) is not None:
            return self.memo.get(key)
        result = self.dp(nums, i + 1, rest - nums[i]) + self.dp(nums, i+1, rest +
nums[i])
        self.memo[key] = result
        return result
```

518. 零钱兑换 II

```
# 518. 零钱兑换 II
from typing import List

class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [[0 for _ in range(amount+1)] for _ in range(len(coins)+ 1)]
        for i in range(len(coins) + 1):
            dp[i][0] = 1

        for i in range(1, len(coins)+1):
            for j in range(1, amount + 1):
                coin = coins[i-1]
                if j - coin < 0:
                    dp[i][j] = dp[i-1][j]
```

```
                else:
                    dp[i][j] = dp[i-1][j] + dp[i][j-coin]
        return dp[-1][-1]
```