数据结构与算法第一次月考cheating sheet

二分查找：最小化最大值模板

```
# 核心思想：答案具有单调性，将“求解”问题转化为“判定”问题。

# 适用于求解 “最大值最小化” 或 “最小值最大化” 问题。

# 例如：袋子里的球、月度开销、Aggressive cows

def solve_binary_search(nums, max_operations): # 根据题目修改函数签名
    # 1. 确定答案的范围 [left, right]
    left = 1  # 答案的最小值 (例如：每个袋子至少1个球)
    right = max(nums) # 答案的最大值 (例如：不操作时的最大开销)
    ans = right # 初始化一个可能的结果
# 2. check(k) 函数：判定函数，核心逻辑
# 功能：假设最终答案是 k，我们是否能用不超过 M 次操作达成目标？
# 返回值：True (可行) 或 False (不可行)
def check(k):
    operations_needed = 0
    # 遍历输入，计算要达成 k 这个目标需要多少次操作
    for num in nums:
        # 这里的计算逻辑根据题目变化
        # 例1: 袋子分球，将 num 拆到每份<=k，需要 (num - 1) // k 次操作
        operations_needed += (num - 1) // k
        # 例2: 月度开销，开销上限k，需要多少个月？
        # current_sum += expense; if current_sum > k: months += 1 ...
    return operations_needed <= max_operations # 根据题意判断

# 3. 二分查找的标准循环
while left <= right:
    mid = (left + right) // 2 # 猜测的答案 k

    if check(mid):
        # 如果 mid 是一个可行的解，说明真正的最小解可能更小
        ans = mid
        right = mid - 1 # 尝试在左半部分寻找更优解
    else:
        # 如果 mid 不可行，说明目标太小了，必须放宽要求
        left = mid + 1 # 在右半部分寻找可行解

return ans
```

OOP基础与运算符重载 （以Fraction为例）

```
# OOP核心：封装数据(属性)和行为(方法)到 class 中。

import math

class Fraction:
    # 构造函数：创建对象时自动调用，用于初始化属性
    def __init__(self, top, bottom):
```

```
        # self 指向实例对象本身
        if bottom == 0:
            raise ValueError("Denominator cannot be zero.")
    # 化简逻辑：在创建时就保证数据规范
    common = math.gcd(top, bottom)
    self.num = top // common  # 实例属性：分子
    self.den = bottom // common # 实例属性：分母

# 字符串表示：当 print(对象) 或 str(对象) 时调用
def __str__(self):
    return f"{self.num}/{self.den}"

# 加法运算符重载：让 '+' 对 Fraction 对象起作用
def __add__(self, other_fraction):
    # self 是 '+' 左边的对象，other_fraction 是右边的对象
    new_num = self.num * other_fraction.den + self.den * other_fraction.num
    new_den = self.den * other_fraction.den

    # 返回一个新的 Fraction 对象，__init__ 会自动处理化简
    return Fraction(new_num, new_den)

# 等于运算符重载：让 '==' 对 Fraction 对象起作用
def __eq__(self, other):
    # 交叉相乘判断是否相等
    return self.num * other.den == self.den * other.num
# --- 使用示例 ---

# f1 = Fraction(1, 2)

# f2 = Fraction(1, 4)

# f3 = f1 + f2  # 调用 __add__

# print(f3)     # 调用 __str__
```

OOP进阶：矩阵与 运算符@

```
# 要让自定义类支持 @ 运算符，必须实现 __matmul__ 方法

class Matrix:
    def __init__(self, data):
        self.data = data
        self.rows = len(data)
        self.cols = len(data[0]) if self.rows > 0 else 0
# 让 '@' 运算符生效 (Matrix Multiplication)
def __matmul__(self, other):
    if self.cols != other.rows:
        raise ValueError("Dimension mismatch for multiplication")

    # 创建结果矩阵
    res_data = [[0 for _ in range(other.cols)] for _ in range(self.rows)]

    # 三重循环计算矩阵乘法
    for i in range(self.rows):
        for j in range(other.cols):
```

```
            for k in range(self.cols):
                res_data[i][j] += self.data[i][k] * other.data[k][j]

    return Matrix(res_data) # 返回新的 Matrix 对象

# 让 '+' 运算符生效
def __add__(self, other):
    if self.rows != other.rows or self.cols != other.cols:
        raise ValueError("Dimension mismatch for addition")

    res_data = [[0 for _ in range(self.cols)] for _ in range(self.rows)]
    for i in range(self.rows):
        for j in range(self.cols):
            res_data[i][j] = self.data[i][j] + other.data[i][j]

    return Matrix(res_data)

# 方便打印
def __str__(self):
    return "\n".join(" ".join(map(str, row)) for row in self.data)
# --- 使用示例 ---

# A = Matrix([[1, 2], [3, 4]])

# B = Matrix([[5, 6], [7, 8]])

# C = A @ B  # 调用 __matmul__

# D = A + C  # 调用 __add__

# print(D)   # 调用 __str__
```

回溯/DFS通用模板

```
# 回溯 = 深度优先搜索(DFS) + 状态重置(撤销选择)

# 适用于：排列、组合、子集、棋盘路径等问题

# --- 模板框架 ---

def backtrack(参数):
    # 1. 递归终止条件
    if (满足结束条件):
        # 收集结果
        # return
# 2. 遍历当前层的所有可能选择
for 选择 in 当前层的选择列表:
    # 2a. 剪枝 (可选，用于去重或排除无效分支)
    if (当前选择不合法):
        continue

    # 3. 做出选择 (修改路径、标记已使用)
    path.append(选择)
    used[选择] = True
```

```
    # 4. 进入下一层递归
    backtrack(下一层的参数)

    # 5. 撤销选择 (回溯，关键步骤！)
    path.pop()
    used[选择] = False
```

回溯应用：全排列 & 子集

```
# --- 全排列 (无重复数字) ---
def permute(nums):
    result, path, used = [], [], [False] * len(nums)
    def backtrack():
        if len(path) == len(nums):
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]: continue
            path.append(nums[i]); used[i] = True
            backtrack()
            path.pop(); used[i] = False
    backtrack()
    return result

# --- 全排列 (有重复数字) ---
def permuteUnique(nums):
    nums.sort() # 关键1：排序
    result, path, used = [], [], [False] * len(nums)
    def backtrack():
        if len(path) == len(nums):
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]: continue
            # 关键2：剪枝，如果当前数字和前一个相同，且前一个没用，则跳过
            if i > 0 and nums[i] == nums[i-1] and not used[i-1]:
                continue
            path.append(nums[i]); used[i] = True
            backtrack()
            path.pop(); used[i] = False
    backtrack()
    return result

# --- 子集 (幂集) ---
def subsets(nums):
    result, path = [], []
    def backtrack(start_index):
        result.append(path[:]) # 每个节点都是一个解
        for i in range(start_index, len(nums)):
            path.append(nums[i])
            backtrack(i + 1) # 传入 i+1 避免重复选择
            path.pop()
    backtrack(0)
    return result
```

链表核心操作

```
# 链表节点定义
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# --- 1. 反转链表 (迭代法，O(1) 空间) ---
def reverseList(head: ListNode) -> ListNode:
    prev, curr = None, head
    while curr:
        next_temp = curr.next # 1. 保存下一个
        curr.next = prev      # 2. 反转指针
        prev = curr           # 3. prev 前进
        curr = next_temp      # 4. curr 前进
    return prev # 返回新的头节点

# --- 2. 寻找相交链表节点 (双指针法，O(1) 空间) ---
# 思路：pA和pB分别走过 A+B 和 B+A 的路程，长度相等，必在交点或None相遇
def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    if not headA or not headB:
        return None
    pA, pB = headA, headB
    while pA is not pB:
        # 如果 pA 到达末尾，就跳到 B 的头；否则前进一步
        pA = headB if pA is None else pA.next
        # 如果 pB 到达末尾，就跳到 A 的头；否则前进一步
        pB = headA if pB is None else pB.next
    return pA # 返回相遇点
```

栈 (Stack) 的应用

```
# Python 的 list 可以完美模拟栈：用 append() 入栈，用 pop() 出栈。

# --- 应用1: 有效的括号 ---
def isValid(s: str) -> bool:
    stack = []
    mapping = {")": "(", "}": "{", "]": "["}
    for char in s:
        if char in mapping: # 如果是右括号
            # 栈为空或栈顶不匹配，则无效
            top_element = stack.pop() if stack else '#'
            if mapping[char] != top_element:
                return False
        else: # 如果是左括号
            stack.append(char)
    return not stack # 最终栈必须为空

# --- 应用2: 十进制转任意进制 (以八进制为例) ---
def dec_to_base8(decimal_num):
    if decimal_num == 0: return "0"
    stack = []
    while decimal_num > 0:
```

```
        remainder = decimal_num % 8
        stack.append(str(remainder))
        decimal_num //= 8
    return "".join(stack[::-1]) # 出栈并拼接
```

队列 （Queue） 应用:

入队 （右侧）
出队 （左侧）collections.deque 是高效队列。
从集合 进口 dequeq = 德克（）q.附加（x）q.popleft（）

字典 （Dict） 与集合 （Set） 技巧

```
from collections import defaultdict, Counter

# --- 1. 倒排索引 (分组) ---
# defaultdict(set) 可以在访问新key时自动创建空集合
inverted_index = defaultdict(set)
docs = {1: "hello world", 2: "great world"}
for doc_id, content in docs.items():
    words = content.split()
    for word in words:
        inverted_index[word].add(doc_id)
# inverted_index -> {'hello': {1}, 'world': {1, 2}, 'great': {2}}

# --- 2. 直播计票 (计数) ---
# Counter 可以一步到位完成计数
votes = [1, 10, 2, 3, 3, 10]
vote_counts = Counter(votes)
# vote_counts -> Counter({10: 2, 3: 2, 1: 1, 2: 1})
max_votes = max(vote_counts.values())
winners = [option for option, count in vote_counts.items() if count == max_votes]
winners.sort()
# winners -> [3, 10]

# --- 3. 模型整理 (复杂排序) ---
# 存储 (可排序的值, 原始字符串), 使用 lambda 排序
models = defaultdict(list)
# ... 解析 'GPT-1.3B' ...
# value = 1.3 * 1000  (统一单位)
# models['GPT'].append((value, '1.3B'))
# 排序
# sorted_params = sorted(models['GPT'], key=lambda item: item[0])
# 提取原始字符串
# original_strings = [item[1] for item in sorted_params]
```

正则表达式 （Regex） 快速参考

```
import re

# --- 常用元字符 ---
# .       : 匹配除换行外的任意单字符
# *       : 前一字符重复 0 次或多次
# +       : 前一字符重复 1 次或多次
```

```
# ?       : 前一字符重复 0 次或 1 次
# ^       : 匹配字符串开头
# $       : 匹配字符串结尾
# \d      : 匹配一个数字 [0-9]
# \w      : 匹配字母、数字、下划线 [a-zA-Z0-9_]
# [abc]   : 匹配 a, b, 或 c
# [^abc]  : 匹配除 a, b, c 之外的任意字符
# ( ... ) : 分组

# --- 常用函数 ---
# re.match(pattern, string): 从字符串开头匹配，成功返回Match对象，否则None
# re.search(pattern, string): 扫描整个字符串，找到第一个匹配
# re.split(pattern, string): 分割字符串
# re.findall(pattern, string): 找到所有匹配，返回列表

# --- 示例：邮箱验证 (根据讲义题目简化) ---
# 规则1: 仅一个'@'
# 规则2: '@'和'.'不出现在首尾
# 规则3: '@'后至少一个'.', 且不与'@'相邻
def is_valid_email_simple(s: str) -> bool:
    # 使用正则表达式 (讲义中的复杂版本)
    # reg = r'^[^\.@][^\@]*@[^\.@][^\@\.]*\.[^\.@]+$' # 一个简化版
    reg = r'^[^\.@]+(\.[^\.@]+)*@[^\.@]+(\.[^\.@]+)+$' # 讲义中的精确版
    if re.match(reg, s):
        return True
    return False

# --- 示例：通配符匹配转正则 ---
# '?' -> '.' (或 '.{1}')
# '*' -> '.*'
# p_regex = p.replace('?', '.').replace('*', '.*') + '$'
# if re.match(p_regex, s): ...
```

时间复杂度 & 算法选择

- **O（1） - 常数时间**： 访问数组/哈希表元素， 。栈/队列的， 。arr[i]dict[键]推流行
- **O（log n） - 对数时间**： 。**二分查找**
- **O（n） - 线性时间**: 遍历数组/链表。
- **O（n log n） - 线性对数时间**： 高效的。**排序算法 (快排、归并)**
- **O（n²） - 平方时间**： （如冒泡排序、卷积）。**双重嵌套循环**
- **O（2^n） - 指数时间**: 未剪枝的递归，如求子集。
- **O（n！） - 阶乘时间**: 未剪枝的求全排列。

算法选择速查:

- **查找**:
  - **有序数组** -> **二分查找 O（log n）**
  - 无序数组 ->遍历 O（n）
  - 需要快速查找、插入、删除 ->**哈希表/字典 O（1）**
- **排序**:
  - 一般情况 -> Python内置 或 （Timsort， O（n排序（）排序（）

    - 归并排序：，需要。**稳定****O（n） 额外空间**
    - 快速排序：，平均 O（n log n**不稳定**
- **求所有可能**(排列/组合/子集/路径)：->**回溯/DFS**
- **括号/表达式/进制转换**: ->**栈**
- **链表**：->**双指针/快慢指针**