Cheet Sheet

一、杂

`map(function, *iterables)`:将函数应用于传入的每个可迭代对象的各个元素.e.g.

```
squared = list(map(lambda x: x**2, [1, 2, 3, 4]))  # [1, 4, 9, 16]
```

debug

RTE:数组越界/除0;TLE/MLE:程序错误(例如递归未设置边界)/复杂度过高(list→dict;使用高效算法)

输入操作

1. 对要求以空行为判定输入结束的依据的,加`input()`即可
2. **sys** 不需要`EOFError`判断是否读完

```
import sys
input = sys.stdin.read  # 一次性读入
output = sys.stdout.write  # 一次性输出
def solve():
  data = input().split()  # 分割处理
    n = int(data[0])
    results = []
    for i in range(1, n + 1):
    results.append(str(int(data[2*i - 1]) + int(data[2*i])))
    output('\n'.join(results) + '\n')
solve()
```

输出操作

1. **解包输出** 例如对矩阵`list[[]]`,输出则对每一行`line`可用`print(*line)`
   (或使用`list`结果分行输出:`print("\n".join(list))`)
2. **F-string** `f"{sum1:.2f}"`

```
print(f"The value of x is {x=}")  # 输出: The value of x is x=value
```

debug

1. **随机数**

```
import random
x=random.randint(a,b)  # >=a,<=b的随机整数
x=random.random()  # 0~1随机浮点数
x=random.uniform(a,b)  # a,b之间随机浮点数
x=random.choice(list)  # 在列表list中随机选择
```

string操作 `string.replace(old, new)`;`string.split()`;`string.strip()` 移除首位指定字符,默认为空格/换行;`string.find(item)` 会输出第一个位置,或给出`-1`.

list/tuple操作 `list('abc')==['a','b','c']`;`list(range(4))==[0,1,2,3]`
`len(list)`;`list.append(item)`;`list.extend(list_extend)`;`list.insert(index, item)`;`list.clear(self)`;`list.reverse(self)`
`list.pop(index)`,返回`index`处元素的值
`list.index(item)` 会输出第一个位置,或抛出`ValueError`.可以通过`try...except`结构捕捉.
`list.sort(key = None, reverse = False)` 原地更改,返回`None`.只可用于list
`list_sorted = sorted(list)` 返回排序后的列表,也可用于tuple/string
`"str_join".join(list_str)`(将序列中元素以指定字符串连接)
`zip()` 压缩可迭代对象

```
a = [1, 2]
b = ['A', 'B', 'C']
c = [True, False, True]
zipped = list(zip(a, b, c))  # [(1, 'A', True), (2, 'B', False)]
e, f, g = zip(*zipped)  # (1, 2), ('A', 'B'), (True, False)
```

`filter(function, iterable)` 过滤可迭代对象

```
numbers = [1, 2, 3, 4, 5, 6]
result = list(filter(lambda n: n % 2 == 0, numbers))  # [2, 4, 6]
```

`list[::1]`得到反转后的`list`
`list[a:b+1]` 返回从索引值为`a`到`b-1`的列表型结果,无法应用原地算法,正确做法e.g.:

```
list[a:b+1] = reversed(list[a:b+1])
```

1. **浅拷贝/深拷贝** 声明二维数组时,`.copy()`导致所有行引用同一个列表,从而修改一个元素时牵连影响.建议列表推导式或深拷贝.

```
matrix[0][0] = 1
print(matrix)  # 输出: [[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
m = 3; n = 4  # 列表推导式
matrix = [[0 for _ in range(n)] for _ in range(m)]
print(matrix)  # 输出: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

```
from copy import deepcopy
lcopy=deepcopy(l)
```

2. **自定义排序** `cmp_to_key`/直接调用函数为`key`

```
from functools import cmp_to_key
def compare_items(x, y):
    if x > y:return 1   # 表示 x 应该排在 y 后面
    elif x < y:return -1  # 表示 x 应该排在 y 前面
    else:return 0   # x 和 y 相等，顺序不变
data.sort(key=cmp_to_key(compare_items))  # 对list类型的data调用
```

```
def compare(x):
    return x.ljust(22, x[0])  # 左对齐,用字符串第一个字符x[0]填充字符串长度到22
x = sorted(list(map(str, input().split())), reverse=True, key=compare)
```

3. **deque** `from collections import deque`,`.popleft()`,`.pop`,`.appendleft()`,`.append()`

4. **namedtuple** 定义字段名称,不止可以用索引访问

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x)  # 10
print(p.y)  # 20
print(p)  # Point(x=10, y=20)
```

set操作

`s=set()` 创建无序不重复空集合/集合化list/tuple/dict. `s.add(s)`;`x (not) in s`;`s.remove(x)` 删除,可能抛出`KeyError`;`s.discard(x)` 删除,如不存在不抛出异常; `s.clear()`;`s.pop()` 移除随机元素,会返回元素值,对空集合抛出`keyError`

1. 集合运算: `a|b` 并; `a&b` 交; `a-b` 差($x \in a, x \notin b$); `a^b` 对称差(`==a|b-a&b`); `<(=)`,`>(=)` 表包含关系

dict操作

key只能0维的,不可是复合数据类型.

`del d[key]`;`d.pop(key)` 返回`value`,可能抛出`KeyError`;`d.popitem()` 随机弹出键值对元组; `d.clear()` 清空; `d.get(key,dft=None)` 可能返回`dft`但不报错; `d.keys()`;`d.values()` `>>dict_keys([items])`.

1. defultdict:会为缺失的键提供一个默认值(`int->0,list->[],set->set(),str->""`),而不是抛出`KeyError`

```
from collections import defaultdict
dictionary = defaultdict(list)  # 自定义默认值defaultdict(lambda: XXX)
```

如果访问不存在的键,则输出异常.会带有类型标记,输出时需要转换格式.

(index, item)生成器enumerate `enumerate(list,tuple,string)`,输出一些tuple.欲对`item`排列,需转化为新列表:

```
indexed_list1 = list(enumerate(list1))
indexed_list1.sort(key=lambda x: x[1], reverse = True)  # False升序,True降序;也可直接-x[1]
```

二分插入内置函数bisect `import bisect`,重要的两个方法如下:

`bisect_left(a, x, lo=0, hi=len(a))` 返回`x`在升序列表`a`中的插入位置(左起寻找第一个满足条件的位置,right相反)

```
print(bisect.bisect_left([1, 3, 4, 7, 9], 5))  # >>>3
```

`insort_left(a, x, lo=0, hi=len(a))`将`x`从左侧插入`a`,并保证`a`的有序性

日期与时间库calendar, datetime

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)  # 更改周初为星期日(默认为星期一)
calendar.isleap(2024)  # 返回True
calendar.leapdays(2000, 2025)  # 返回6(含首不含尾)
calendar.month(2024, 1)  # 返回str类型2024-1日历
calendar.monthrange(2024, 1)  # 返回(0, 31)
calendar.monthcalendar(2024, 1)  # 返回二维数组weeks,缺日(不在此月)为0
```

```
from datetime import datetime
datetime.now()  # 返回"2024-01-01 12:34:56.789123";.today()/.time()同
dt = datetime(2024, 1, 1, 12, 30, 0)  # 2024-01-01 12:30:00
dt.strftime('%Y-%m-%d %H:%M:%S')  # 2024-01-01 12:30:00
dt.strftime('%A, %B %d, %Y')  # Monday, January 01, 2024

date_str = "2024-01-01 12:30:00"
parsed_date = datetime.strptime(date_str, '%Y-%m-%d %H:%M:%S')
print(parsed_date)  # 2024-01-01 12:30:00

now = datetime.now()
future_date = now + timedelta(days=7)
past_date = now - timedelta(hours=1)

d1 = datetime(2024, 1, 1)
d2 = datetime(2023, 12, 31)
d1 > d2  # True
d1 - d2  # 1 day, 0:00:00
```

迭代器库itertools

```
import itertools
for item in itertools.product('AB', repeat=2):  # 生成笛卡尔积
    print(item)  # ('A', 'A')\n('A', 'B')\n('B', 'A')\n('B', 'B')
for item in itertools.product('AB', '12'):
    print(item)  # ('A', '1')\n('A', '2')\n('B', '1')\n('B', '2')
result = list(permutations([1,2,3]))  # [(1, 2, 3), (1, 3, 2), ……];生成全排列
```

二、dp

0-1背包:只有一件,选择只有拿/不拿(重点:初始化,转移方程,提取结果)

```
def knapsack_2d(weights, values, W):
    n = len(weights)
    dp = [[0] * (W + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(W + 1):
            if j >= weights[i - 1]:  # 第i个物品能装进,判断不选/选这个物品
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] +
values[i - 1])
            else: dp[i][j] = dp[i - 1][j]
    return dp[n][W]
```

```
def knapsack_1d(weights, values, W):  # 一维视为二维的滚动数组实现
    n = len(weights)
    dp = [0] * (W + 1)  # 初始化 dp 数组，容量从 0 到 W
    for i in range(n):  # 遍历每件物品
        for j in range(W, weights[i] - 1, -1):  # 倒序遍历背包容量(保证每件物品只能选一
次)
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[W]
```

1. **状态转移关系的映射**

二维DP状态	一维DP状态
`dp[i][j]`	`dp[j]`
`dp[i-1][j]`	上一轮的 `dp[j]`
`dp[i-1][j-k]`	当前数组中未覆盖部分

2. **倒序遍历的原因**:在一维DP中,为了保证当前状态 `dp[j]` 只使用上一轮的状态值,我们需要从容量 `W` 倒序遍历.如果正序遍历,`dp[j]` 会被更新后的 `dp[j - weight[i]]` 影响,从而导致错误的结果.

区间dp: `dp[i][j]` 表示区间 `[i, j]` 的最优解
状态转移:拆分区间`dp[i][j]=min/max{dp[i][k]+dp[k+1][j]+cost(i,j,k)}(i<=k<j)`,从小到大递推.
预处理:利用前缀和等快速计算区间`cost`;结合特殊性质优化(否则复杂度为$O(n^3)$)

```
n = int(input())  # 石子的堆数
stones = list(map(int, input().split()))
sum_ = [0] * (n + 1)  # 前缀和，用于快速计算区间和
for i in range(1, n + 1):
    sum_[i] = sum_[i - 1] + stones[i - 1]
dp = [[float('inf')] * n for _ in range(n)]  # dp 数组，初始化为正无穷
for i in range(n):
    dp[i][i] = 0  # 单堆的代价为 0
for L in range(2, n + 1):  # 枚举区间长度 L，从 2 到 n
    for i in range(n - L + 1):  # 起始位置从0到n-L
```

```
        j = i + L - 1  # 长度为L后的终点
        for k in range(i, j):  # 枚举分割点 k
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + sum_[j + 1] -
sum_[i])
print(dp[0][n - 1])
```

完全背包问题 允许在不超容量的前提下无限次选取

```
def knapsack_complete(weights, values, capacity):
    dp = [0] * (capacity + 1)  # dp[j]为当背包容量为j时,背包所能容纳的最大价值
    dp[0] = 0
    for i in range(len(weights)):  # 遍历所有物品
        for j in range(weights[i], capacity + 1):  # 从当前物品的重量开始，计算每个容量
的最大价值
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[capacity]
```

必须装满的类型:

```
def knapsack_complete_fill(weights, values, capacity):
    dp = [-float('inf')] * (capacity + 1)  # 初始值为负无穷，表示不能达到该容量
    dp[0] = 0  # 容量为 0 时，价值为 0
    for i in range(len(weights)):  # 遍历所有物品
        for w in range(weights[i], capacity + 1):  # 遍历所有容量，从 weights[i] 开
始
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    # 如果 dp[capacity] 仍为 -inf，说明无法填满背包
    return dp[capacity] if dp[capacity] != -float('inf') else 0
```

多重背包 每个物品有上限

```
def binary_optimized_multi_knapsack(weights, values, quantities, capacity):
    # 使用二进制优化解决多重背包问题
    n = len(weights)
    items = []
    # 将每种物品根据数量拆分成若干子物品（使用二进制优化）
    for i in range(n):
        w, v, q = weights[i], values[i], quantities[i]
        k = 1
        while k < q:
            items.append((k * w, k * v))  # 添加子物品(weight, value)
            q -= k
            k << 1  # 位运算,相当于k *= 2,按二进制拆分,物品时间复杂度由q变为log(q)
        if q > 0:
            items.append((q * w, q * v))  # 添加剩余部分,如果有的话
    # 动态规划求解0-1背包问题
    dp = [0] * (capacity + 1)
    for w, v in items:  # 遍历所有子物品
        for j in range(capacity, w - 1, -1):  # 01背包的倒序遍历
            dp[j] = max(dp[j], dp[j - w] + v)
    return dp[capacity]
```

dp中“正难则反”的常见类型

1. **区间问题**:从目标区间反向分解为子区间
2. **路径问题**:从终点反向推导到起点，利用已有结果
3. **子序列问题**:从末尾回溯，逐步构建解

双重dp 解决有不同方案的情况,e.g.土豪购物

```
value = list(map(int, input().split(",")))
dp_keep = value[0]       # 不放回
dp_remove = value[0]     # 放回一件商品
ans = value[0]  # 答案记录
for i in range(1, len(value)):  # 对结尾为第i件商品的选法
    previous_dp_keep = dp_keep  # 保存结尾尾i-1时的选法
    dp_keep = max(dp_keep + value[i], value[i])  # 维护dp_keep
    dp_remove = max(previous_dp_keep, dp_remove + value[i])  # 判断是否放回第i个商品更划算
    ans = max(ans, dp_keep, dp_remove)
print(ans)
```

三、math

数学表示 fractions与float混合时结果为fractions;demical不建议从float转换,有精度问题;demical可以避免float的精度问题

```
import fractions, decimal
f1=fractions.Fraction(1, 3)  # 1/3
f1.numerator  # 1
f1.denominator  # 3
f2=Fraction('1.5')  # 3/2
f3=Fractino(0.5)  # 1/2
dec=decimal.Decimal("0.1") # 0.1
```

数学运算 绝对值`abs()`;幂函数`pow(x, y)`:x^y.二进制`bin`(binary),从编号`2:`(第三个字符开始)才是结果
除后取整:向下`num // n`,向上`(num + n - 1) // n`
`math.ceil(number)`向上取整,`math.floor(number)`向下取整,`math.trunc(number)`截断取整(去掉小数位数)

Euler筛 得到`1<=i<=n`的素数(以列表查找的形式为例,截至`10 ** 6`)

```
is_prime = [True] * 1000001
for i in range(2, int(1000000 ** 0.5) + 1):  # 只需查找到开根得到的数
    if is_prime[i]: # 如果小的数是素数
        for j in range(i * i, 1000001, i):  #所有i*j(j=i, i+1, ……)都不是素数
            is_prime[j] = False
```

求最大公约数(gcd):辗转相除法(辗转相除直至余数为`0`)(省事可以`from math import gcd`,`ans = gcd(a, b)`)

```
def gcd(a, b):
  while b:
    a, b = b, a%b
  return a
```

四、recursion

设置递归深度

```
import sys
sys.setrecursionlimit(1<<30)  # 设置递归深度为2^30
```

五、two pointers

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = left = pre_max = suf_max = 0  # 初始化结果、左指针和两个最大高度为0
        right = len(height) - 1  # 初始化右指针为数组末尾
        while left < right:  # 当左指针小于右指针时循环
            pre_max = max(pre_max, height[left])  # 更新左指针位置的最大高度
            suf_max = max(suf_max, height[right])  # 更新右指针位置的最大高度
            if pre_max < suf_max:  # 如果左指针位置的最大高度小于右指针位置的最大高度
                ans += pre_max - height[left]  # 计算并累加左指针位置能够接住的雨水量
                left += 1  # 移动左指针
            else:  # 否则
                ans += suf_max - height[right]  # 计算并累加右指针位置能够接住的雨水量
                right -= 1  # 移动右指针
        return ans  # 返回最终结果
```

六、matrix

滑动窗口求极值:创建大窗口--根据输入对对应点周围"检查窗口"可作用的范围赋值(注意大窗口边界)---初始化极值及其数量--遍历,同则加数量,大则赋值给极值,重置数量

> 易错点:未将输入值`-=1`转化为以0开始记数的表格里的位置;数组越界;直径/半径

七、searching

图(包括矩阵,迷宫,树)搜索包括:
DFS:走到边界后回溯;适用于图遍历,确认路径存在性;使用递归或栈.(为了防止回头注意临时填路)

```
d=[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
def dfs(x,y):
    for dx, dy in d:
        nx, ny=x+dx, y+dy
        if 0<=nx<len(mat) and 0<=ny<len(mat[0]):
            if mat[nx][ny]==1:
                mat[x][y]=0
                dfs(nx,ny)
                mat[x][y]=1
```

高级操作:永久填路;记录参数(步数/路径/权值);记录面积(`global`);判断终点并返回,等可能开头需要`sys.setrecursionlimit()`来防止超出默认递归深度.

> 注意加保护圈,或调用判断函数;声明全局变量注意要在之前定义过,否则`Compile Error`

对某些图,为防止重复计算,可以用`functools`中的`lru_cache`在`def dfs()`之前(Least Recently Used算法存入cache)

```
from functools import lru_cache
@lru_cache(maxsize=None)  # 不设置缓存上限
def expensive_function(param):
    return result
```

BFS:从起点逐层扩展;适用于寻找最短路径/图遍历

```
from collections import deque
def bfs(start, end):
    queue = deque([(0, start)])  # 定义队列queue,起点元组(step, start)入队,目前步长
step = 0
    in_queue = {start}   # 记录已入队节点(而非已被访问的节点)
    while queue:  # 如果queue非空
        step, front = queue.popleft() # 取出队首元素
        if front == end:
            return step # 返回需要的结果，如：步长、路径等信息
        # 将 front 的下一层结点中未曾入队的结点全部入队queue，并加入集合in_queue设置为已入
队
```

1. 定义队列`queue`,起点`(0, start)`入队,目前步长为`0`
2. `while`循环,条件是队列`queue`非空
3. `while`循环中,取出队首元素`front`
4. 将`front`的下一层结点中未曾入队的结点入队,标记他们的层号为`step`的层号加`1`并加入`in_queue`
5. 返回第二步继续循环

```
from collections import deque
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:  # 烂橘子问题
        m, n = len(grid), len(grid[0])
        queue = deque()
        fresh_oranges = 0
        for i in range(m):
```

```
            for j in range(n):
                if grid[i][j] == 2:
                    queue.append((i, j, 0))  # 腐烂橘子的位置和初始时间
                elif grid[i][j] == 1:
                    fresh_oranges += 1
        if fresh_oranges == 0:
            return 0
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        max_time = 0
        while queue:
            x, y, time = queue.popleft()
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] == 1:
                    grid[nx][ny] = 2
                    fresh_oranges -= 1
                    queue.append((nx, ny, time + 1))
                    max_time = max(max_time, time + 1)
        return -1 if fresh_oranges > 0 else max_time
```

树状图结构,构建邻接表

```
from collections import deque
n = int(input())  # 节点数目
can_go = [[] for _ in range(n)]
for _ in range(n-1):
    s, e = map(int, input().strip().split())
    can_go[s].append(e)  # 无向树结构,需要加两次
    can_go[e].append(s)
strict = set(map(int, input().split()))  # 受限节点
def bfs(can_go, strict):  # 其实用bfs没有很合适
    queue = deque([0])
    in_queue = {0}
    while queue:
        front = queue.popleft()
        if not can_go[front]: break
        for ne in can_go[front]:
            if (ne not in strict) and (ne not in in_queue):
                in_queue.add(ne)
                queue.append(ne)
    return len(in_queue)
print(bfs(can_go, strict))
```

> `in_queue`用来判断节点是否已经入队,而不是节点是否已经被访问.
> 若设置成是否被访问,可能由于其他节点可以到达正在队列中的某节点(未被访问)而将其反复入队,大大增加算量.

Dijkstra:用最小堆实现;适用于寻找加权图单源最短路径(起始点到其他节点的最短路径,需要权值非负)

```
import heapq
directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
def dijkstra(xs, ys, xe, ye):  # 走山路
    if region[xs][ys] == "#" or region[xe][ye] == "#":
```

```
        return "NO"
    if (xs, ys) == (xe, ye):
        return 0
    pq = []  # 初始化堆
    heapq.heappush(pq, (0, xs, ys))  # (体力消耗, x坐标, y坐标)
    visited = set()  # 已访问坐标集
    efforts = [[float('inf')] * n for _ in range(m)]  # 到达图中位置需要的体力,初始化为inf,即不可达到
    efforts[xs][ys] = 0  # 初始化体力记录表
    while pq:
        current_effort, x, y = heapq.heappop(pq)  # 取出堆顶元素
        if (x, y) in visited: continue  # 若访问过,跳过这个节点
        visited.add((x, y))  # 未访问过,加入已访问
        if (x, y) == (xe, ye):  return current_effort  # 达到终点则返回
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if 0 <= nx < m and 0 <= ny < n and (nx, ny) not in visited:
                if region[nx][ny] == "#":  continue  # 无法达到,跳过
                effort = current_effort + abs(int(region[x][y]) - int(region[nx][ny]))
                if effort < efforts[nx][ny]:
                    efforts[nx][ny] = effort
                    heapq.heappush(pq, (effort, nx, ny))  # 放入堆中
    return "NO"
```

八、排序/查找

Binary Search 有序数集查找.注意`while`条件不容易写对

```
def binary_search(arr, target):
  left, right = 0, len(arr) - 1
  while left <= right:
    mid = (left + right) // 2
    if arr[mid] == target:
      return mid
    elif arr[mid] <target:
      left = mid + 1
    else:
      right = mid - 1
    return -1
```

Merge Sort 归并排序

```
def merge_sort(arr):
    if len(arr) <= 1: return arr  # 如果数组长度小于等于1,则无需排序
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    sorted_left = merge_sort(left)  # 递归地对两半进行分割排序
    sorted_right = merge_sort(right)
    return merge(sorted_left, sorted_right)  # 合并排序后的两半
def merge(left, right):
    result = []  # 用于存放合并后的结果
```

```
    i = j = 0    # 两个指针，分别指向左半部分和右半部分的开头
    while i < len(left) and j < len(right):  # 比较左右两部分的元素，将较小的放入结果数组
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])  # 将剩余的元素加入结果数组
    result.extend(right[j:])
    return result
```

Quick Sort 不稳定,如**5**,5->5,**5**

```
def quicksort(arr):
    if len(arr) <= 1: return arr  # 如果数组长度为0或1,直接返回(已排序)
    pivot = arr[0]  # 选择一个基准值(pivot),这里选择第一个元素
    less = [x for x in arr[1:] if x <= pivot]  # 小于等于基准值的元素
    greater = [x for x in arr[1:] if x > pivot]  # 大于基准值的元素
    return quicksort(less) + [pivot] + quicksort(greater)  # 递归地对两部分排序并合并
```

In-place Quick Sort

```
def quick_sort(arr):
    def partition(low, high):
        pivot = arr[high]  # 选择最后一个元素作为基准
        i = low - 1  # i 是小于 pivot 的最后一个元素索引
        for j in range(low, high):
            if arr[j] < pivot:  # 保证所有小于pivot的元素被按顺序排到左侧
                i += 1
                arr[i], arr[j] = arr[j], arr[i]  # 交换较小元素到左侧
        arr[i + 1], arr[high] = arr[high], arr[i + 1]  # 把 pivot 放到正确位置
        return i + 1  # 返回pivot的最终位置
    def quick_sort_recursive(low, high):
        if low < high:
            pi = partition(low, high)  # 分区，获取基准值位置
            quick_sort_recursive(low, pi - 1)
            quick_sort_recursive(pi + 1, high)  # 递归排序基准值左右两部分
    quick_sort_recursive(0, len(arr) - 1)
    return arr
print(quick_sort([9, 3, 7, 1, 4, 2, 5]))  # >>>[1, 2, 3, 4, 5, 7, 9]
```

九、Heapq

列表堆化方法(无返回值)`heap.heapify(pre_heap)`.大顶堆只能小顶堆元素加负号实现

模版题potions:`.heappush(heap, item)`压入元素并排序,`.heappop(heap)`弹出堆顶(返回堆顶元素)

`heapq.heappushpop(heap, item)`先推再弹,`heapq.heapreplace(heap, item)`先弹再推

```
import heapq
def max_potions(n, potions):
    heap = []
    health = 0
```

```
    num = 0
    for effect in potions:
        if health + effect >= 0:  # 喝下不会寄掉
            heapq.heappush(heap, effect)
            health += effect
            num += 1
        elif heap and effect > heap[0]:  # 当替换最差药水可以得到更好的健康值
            health -= (heapq.heappop(heap) - effect)  # 弹出最差的药水
            heapq.heappush(heap, effect)  # 加入新的药水
    return num  # 输入输出结构省略
```

`heap.heapify(list)`时间复杂度$O(n)$.

十、区间问题

(1)区间合并后区间:

```
intervals.sort(key=lambda x: x[0])  # 左端点从小到大排序
ans = []
for interval in intervals:
    if ans and interval[0] <= ans[-1][1]:  # 左端点可以连上最后的区间,维护最后区间末端为两者最大值
        ans[-1][1] = max(ans[-1][1], interval[1])
    else:  # 空ans,加入第一个区间/未能连上,加入新的合并区间
        ans.append(interval)
```

(2)选取数量最多的不相交区间集(认为点接触不算重叠);与"区间任意选点使所有区间含点且点数最小"思路一致

```
intervals.sort(key = lambda x :x[1])  # 右端从小到大排序
count = 1; end = intervals[0][1]  # 选第一个区间,初始化
for i in range(1, len(intervals)):
  if intervals[i][0] >= end:  # 若与之前没有交集
    end = intervals[i][1]; count += 1
print(count)
```

(3)覆盖目标区间的最少区间选取数目

```
clips.sort(key = lambda x:x[0])  # 左端点从小到大排序
start, end = 0, time
ans = 0; i = 0  # 初始化答案和计数器
while i < len(clips):
    maxR = -1  # 维护:满足能覆盖初始值的区间的右侧最大时间
    while i < len(clips) and clips[i][0] <= start:
        maxR = max(maxR, clips[i][1])
        i += 1
    if maxR < start:  # 未找到符合要求的区间
        return -1
    ans += 1  # 找到了,ans加1
    if maxR >= end:      # 达若到结尾,返回ans
        return ans
    start = maxR  # 未达到结尾,将目标区间开头设定为maxR,继续第一层while
return -1  # 若i超出列表长度,找不到
```

(4)区间分组,每组内不相交,求使组数最小的分组方式(greedy)
思路:排序后对当前遍历到的区间$[L_i, r_i]$,若对目前分组的第k组右端点r_k满足$L_i \leq r_k$,则不可放入;反之可以.
若未被接收,新开一个组放入.为了能快速查找接收组,可以使用右端点优先队列.

```
import heapq
def minmumNumberOfHost(self, n, startEnd):
    startEnd.sort()
    min_heap = []  # 创建最小堆,维护所有组的末端并快速找到最小末端
    for interval in startEnd:
        start, end = interval
        if not min_heap or min_heap[0] > start:  # 若空堆或最小的右端大于当前左端,加入新的一组
            heappush(min_heap, end)
        else:  # 弹出堆顶,压入当前结束时间
            heappop(min_heap)
            heappush(min_heap, end)
    return len(min_heap)    # 最终堆中的元素个数就是组数
```

(5)是否存在不重叠区间:

```
import heapq
from collections import defaultdict
q = int(input())
left_set = defaultdict(int); right_set = defaultdict(int); min_r = []; max_l = []
for _ in range(q):
    operate = input().split()
    l, r = int(operate[1]), int(operate[2])
    if operate[0] == "+":
        left_set[l] += 1; right_set[r] += 1
        heapq.heappush(max_l, -l); heapq.heappush(min_r, r)
    else:
        left_set[l] -= 1; right_set[r] -= 1
    # 清除堆中无效边界
    while max_l and left_set[-max_l[0]] <= 0:heapq.heappop(max_l)
    while min_r and right_set[min_r[0]] <= 0:heapq.heappop(min_r)
    # 贪心策略:若最小右边界小于最大左边界,存在不重叠的一组区间
    if max_l and min_r and min_r[0] < -max_l[0]:print("YES")
    else:print("NO")
```

十一、单调栈

```
class Solution:
    def trap(self, height: List[int]) -> int:
        stack = []  # 初始化一个空栈，用于存储柱子的索引
        water = 0  # 初始化水的总量为0
        for i in range(len(height)):  # 遍历每个柱子
            while stack and height[i] > height[stack[-1]]:  # 当栈不为空且当前柱子高于栈顶柱子时
                top = stack.pop()  # 弹出栈顶柱子的索引
                if not stack:  # 如果栈为空，说明没有左边的边界，无法形成水坑
                    break
                distance = i - stack[-1] - 1  # 计算左右边界之间的距离
```

```
                bounded_height = min(height[i], height[stack[-1]]) - height[top]  # 计算水坑的高度
                water += distance * bounded_height  # 计算水坑的容量，并累加到总水量中
            stack.append(i)  # 将当前柱子的索引压入栈中
        return water  # 返回总的水容量
```

十二、回溯

```
class Solution:  # 划分为k个和相等的子集
    def canPartitionKSubsets(self, nums: list[int], k: int) -> bool:
        total_sum = sum(nums)
        if total_sum % k != 0: return False # 无法整除
        target = total_sum // k
        nums.sort(reverse=True)
        if nums[0] > target: return False  # 若有元素超出目标
        aim = [0] * k  # 实际上是一个buckets结构(桶)
        def backtrack(index):  # 对index位置的num回溯
            if index == len(nums): return all(tot == target for tot in aim)
            for i in range(k):  # 对aim中每个元素尝试分配当前nums[index]
                if aim[i] + nums[index] > target: continue  # 若和超出目标,尝试下一个元素
                aim[i] += nums[index]  # 若未超过,求和
                if backtrack(index + 1): return True  # 递归调用尝试下一数字,能达成目标返回
                aim[i] -= nums[index]  # 不能达成目标,回溯,撤销分配
                if aim[i] == 0: break
                # 如果放入第一个空元素的尝试失败,不再尝试剩下的元素(都是空的)
            return False
        return backtrack(0)
```

十三、Hashmap

挑选礼物:$\frac{S[j]-S[i-1]}{j-(i-1)} = 520 \Rightarrow S[j] - 520 \times j = S[i-1] - 520 \times (i-1)$

```
def max_total_value(n, values):
    prefix_sum = 0
    max_k = 0
    hashmap = {}
    hashmap[0] = 0
    for j in range(1, n+1):
        prefix_sum += values[j-1]  # 维护前缀和
        key = prefix_sum - 520 * j  # 得到S[j] - 520*j
        if key in hashmap:  # 在hashmap中查找是否为已有的键
            i = hashmap[key]  # 调用此键对应的前一个位置
            k = j - i  # 得到一个可能的数组长度
            max_k = max(k, max_k)  # 维护最大长度
        else:hashmap[key] = j  # 如果是不存在的键,存下其对应的值
    return 520 * max_k
```

第一个缺失正数:索引为隐式hashmap

```python
def firstMissingPositive(self, nums: List[int]) -> int:
  n = len(nums)
    for i in range(n): # 为使用常量级别空间,直接原地更改,将正整数i放到索引值i-1对应的位置
  while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
    nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]
  for i in range(n):
     if nums[i] != i + 1:
        return i + 1
  return n + 1
```

十四、排列

1. 求下一个排列:
   从右往左找到第一个降序相邻数对`nums[i],nums[i+1]`,未找到说明已经逆序,直接返回反转值.
   找到后,在右边找第一个`nums[j]`比`nums[i]`大,调换.
   反转`nums[i+1]`

   ```python
   import bisect
   class Solution:
       def nextPermutation(self, nums: List[int]) -> None:
           def find_break(nums):
               point = len(nums) - 1
               while point >= 1:
                   if nums[point - 1] < nums[point]:
                       return point
                   point -= 1
               return False
           if not find_break(nums):
               nums.sort()  # 需要减少内存占用时可用两个指针对应的逆序段中元素替换
               return
           index = find_break(nums) - 1
           for i in range(len(nums) - 1, index, -1):
               if nums[i] > nums[index]:
                   nums[i], nums[index] = nums[index], nums[i]
                   break
           nums[index+1:] = sorted(nums[index+1:])
           return
   ```

2. 得到所有排列

   ```python
   def permute(nums):
       if len(nums) == 1: return [nums]  # 递归终止条件：只有一个元素时返回自身
       permutations = []  # 存储答案
       for i in range(len(nums)):
           current = nums[i]  # 当前元素
           remaining = nums[:i] + nums[i+1:]  # 剩余元素
           for p in permute(remaining):  # 递归生成剩余元素的排列，并加上当前元素
               permutations.append([current] + p)
       return permutations
   ```

3. Cantor展开:排列⟼编号;编号计算方式:$\sum_{i=0}^{n-1} \text{贡献}_i \times (n-1-i)!$;贡献:$i$之后比它更小的个数.
   逆Cantor展开:编号⟼排列

   1. 给定一个索引值k和长度n,初始化一个包含1到n的数字数组`elements = [1, 2, ..., n]`
   2. 从高位到低位,确定排列的每一位:用$k \div (n-1)!$$k \div (n-1)!$确定当前位,用$k \mod (n-1)$更新剩余索引.

   不适用于重复元素、组合或不规则排列

```python
import math
def Cantor(nums):
    result = 0  # result 是最终的编号
    n = len(nums)
    for i in range(n):#对于nums中的每个数
        count = 0  # 用于统计当前数字后面比它小的数的个数
        for j in range(i+1, n):
            if nums[j]<nums[i]:
                count += 1
        result + =count * math.factorial(n-1-i)
    return result
def retro_cantor(index, length):
    result = []  # 存储还原的排列
    available_numbers = list(range(1, length + 1))  # 可选数字列表
    for i in range(length-1,-1,-1):
        f = math.factorial(i)  # 计算当前阶乘
        position = index // factorial  # 确定当前位的数字在可选数字中的位置
        result.append(available_numbers.pop(position))  # 从可选数字中取出对应数字
        index %= f  # 更新编号以处理后续位
    return result
```

十五、LIS问题

1. dp解法:`dp[i]`截至`i`的最长上升序列

```python
def lengthOfLIS(arr):
    n = len(arr)
    dp = [1] * n  # 记录最长序列
    prev = [-1] * n  # 记录前驱索引
    for i in range(1, n):
        for j in range(i):  # 对小于i的状态转移
            if arr[i] > arr[j] and dp[j] + 1 > dp[i]:
              dp[i] = dp[j] + 1
              prev[i] = j  # 截至i的最长序列对应的上一个索引
    max_index = max(range(n), key=lambda i: dp[i])  # 最大的dp[i]对应的索引
    lis = []
    while max_index != -1:  # 回溯 LIS
        lis.append(arr[max_index])
        max_index = prev[max_index]
    return lis[::-1]  # 反转得到正序的 LIS
```

2. 二分解法: `dp[i]` 长度为 `i` 的上升子序列的最小末尾索引(必然递增,核心思想); `dp` 有多少项填了数,LIS长度就是多少

```
import bisect
def lis(a):
    dp=[float('inf')]*(len(a)+2)
    for i in range(len(a)):  # 对于第i项,找到可插入位置,替换(不能直接插入)
        dp[bisect.bisect_left(dp,a[i])]=a[i]
    return bisect.bisect_left(dp,float('inf'))  # 第一个"inf"的位置即为所求
```

十六、Kadane算法求最大和子序列

```
def maxSubArray(arr):
    current_sum = 0
    max_sum = float('-inf')  # 初始化为负无穷，处理全负数组的情况
    for num in arr:
        current_sum = max(num, current_sum + num)  # 更新当前子数组和
        max_sum = max(max_sum, current_sum)  # 更新最大子数组和
    return max_sum
```

这看起来很dp.若要存储路径(即最大子数组):

```
def maxSubArray(arr):
    current_sum = 0
    max_sum = float('-inf')
    start = 0  # 当前子数组的起始索引
    end = 0    # 最大子数组的结束索引
    temp_start = 0  # 临时存储起始索引
    for i, num in enumerate(arr):
        if num > current_sum + num:  # if……else状态转移
            current_sum = num
            temp_start = i  # 重新开始新的子数组
        else:
            current_sum += num
        if current_sum > max_sum:
            max_sum = current_sum
            start = temp_start  # 更新最大子数组的起始索引
            end = i  # 更新最大子数组的结束索引
    return max_sum, arr[start:end+1]
```

二维数组扩展

```
def maxSubMatrix(matrix):
    if not matrix or not matrix[0]:
        return 0
    rows, cols = len(matrix), len(matrix[0])
    max_sum = float('-inf')
    for top in range(rows):  # 遍历子矩阵上边界
        row_sum = [0] * cols  # 对每一次上边界,初始化列和
        for bottom in range(top, rows):  # 从top到rows行之间的列和
            for col in range(cols):
                row_sum[col] += matrix[bottom][col]
```

```
            current_sum = 0  # 使用 Kadane's 算法求当前列和的最大值
            # 求列和的最大子数组和,并更新全局最大和
            local_max = float('-inf')
            for x in row_sum:
                current_sum = max(x, current_sum + x)
                local_max = max(local_max, current_sum)
            max_sum = max(max_sum, local_max)
    return max_sum
```

一维周期数组(环形数组)扩展

```
def maxCircularSubArray(arr):
    max_kadane = kadane(arr)  # 普通 Kadane 最大子数组和
    total_sum = sum(arr)
    min_kadane = kadane([-x for x in arr])  # (负的)最小子数组和
    max_circular = total_sum + min_kadane  # 环形子数组和
    # 如果全是负数，max_circular 会变成 0，所以只返回 max_kadane
    return max(max_kadane, max_circular)  # 两种可能情况取max
def kadane(arr):
    current_sum = 0
    max_sum = float('-inf')
    for num in arr:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

十七、Manacher算法求最长回文子串

```
def manacher(s):
    # 1.预处理字符串
    t = '^#' + '#'.join(s) + '#$'  # 字符间插入#,从而对于偶数子串也可以中心扩展
    n = len(t)  # 得到新字符串长度
    P = [0] * n  # P[i]表示以t[i]为中心的回文半径
    C, R = 0, 0  # C为当前回文中心,R为当前回文的右边界
    # 2.计算回文半径
    for i in range(1, n - 1):  # i位置为中心
        # 如果 i 在 R 范围内，用对称位置的回文半径初始化 P[i]
        P[i] = min(R - i, P[2 * C - i]) if i < R else 0
        # 中心扩展，尝试扩展回文半径
        while t[i + P[i] + 1] == t[i - P[i] - 1]:
            P[i] += 1
        # 更新回文的中心和右边界
        if i + P[i] > R:
            C, R = i, i + P[i]
    # 3.找到最长回文
    max_len = max(P)  # 最长回文半径
    center_index = P.index(max_len)  # 最长回文对应的中心索引
    # 原始字符串中的起始索引
    start = (center_index - max_len) // 2
    return s[start:start + max_len]
```