Cheat Sheet

常见的代码错误

输入输出

注意输入的字符串与数值不同，即1！='1',一定要检查！

函数的书写问题

```
max() #注意是小括号，和数学中的表示不同
for key in adict: #直接用dic表示key的所有值
for value in adict.values():
for key,value in adict.items(): #不能直接使用，需要指定.
```

函数的返回值与调用问题

```
a.reverse() #直接修改列表元素
a.sort(key = lambda x: (x[3:], int(x[:3])), reverse=(False, True)) #注意多重排序的答案的书写方式
b = sorted(a) #直接返回列表
b = list(reversed(a)) #需要指定变量类型
c = a/b #只要出现除法就不会返回整数，需要int()
alist.remove(a) # 删除第一个为2的值
# enumerate 函数使用
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
print(f"Index: {index}, Fruit: {fruit}")
```

输出形式

```
print(f'a+{b}+c') #b为变量名
print('\n'.join(alist)) #''之间填充连接符号
print(*alist, sep = '')#如果不指定对应的sep的值，那么会自动按照空格连接
## 小数输出
print("{:.2f}".format(num))
#或
print("%.2f" % num)
print(f"{value:.nf}") #其中n是希望保留的小数位数
#百分数
print(f"Percentage: {percentage:.2%}") # 输出： 85.00%
#科学计数法
print(f"Scientific notation: {large_number:.2e}") # 输出： 1.23e+06
```

优先级问题

```
# 不确定优先级的问题一定记得加括号！
if a or b == c: #这样的写法是错误的，只要 a != 0 就会执行
```

矩阵

必须掌握的是数组越界的问题
思路一：加保护圈（判断邻近值）
思路二：注意max(),min()函数的使用，这种报错不一定是RE,也可以是WA(减小方向越界)，一定要注意检查的方向

螺旋矩阵

```
n = int(input())
s = [[401]*(n+2)]
mx = s + [[401] + [0]*n + [401] for _ in range(n)] + s

dirL = [[0,1], [1,0], [0,-1], [-1,0]]

row = 1
col = 1
N = 0
drow, dcol = dirL[0]

for j in range(1, n*n+1):
    mx[row][col] = j
    if mx[row+drow][col+dcol]:
        N += 1
        drow, dcol = dirL[N%4]

    row += drow
    col += dcol

for i in range(1, n+1):
    print(' '.join(map(str, mx[i][1:-1])))
```

正则表达式

```
import re
if re.match(r'R\d+C\d+', s)
```

ASCII码

```
ord('A') = 65
ord('Z') = 90
ord('a') = 97
ord('z') = 122
import string
print(string.ascii_uppercase) # ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(list(string.ascii_lowercase)) # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z']
```

数学

```
import math
math.ceil() #向上取整
math.floor() #向下取整
math.pow(x, y) #指数 x^y
math.sqrt(x) #开根号
math.log(x, y) #log_y^x 注意顺序!
```

位运算符的应用

```
# 解释位运算符的含义：将两个数转成二进制，然后比较对应的位，如果相等且均为1，则输出1；否则输出0。最后将二进制数转化成十进制数输出。而且，用这种计算方式可以相对加快程序的运行速度。
a&(a-1) # 判断a是否是2的整数次幂
a&1 # 判断a是否是奇数
```

常见的代码块

统一输入输出，节约时间

```
import sys
input = sys.stdin.read
data = input().split()  # 读入所有数据并分割为列表
```

优化递归程序

```
import sys
from functools import lru_cache
sys.setrecursionlimit(1<<30) ### 设置递归深度为2**30
@lru_cache(maxsize=None)### 缓存最近计算的数
```

素数筛---打表!

```
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes
```

最大整数排序问题

```
from functools import cmp_to_key ### 用冒泡排序的方法优化实现
def cmp_max(a, b):
    if a+b > b+a:
        return -1
    elif a+b < b+a:
        return 1
    else:
        return 0
n = int(input())
s = list(input().split())
s.sort(key = cmp_to_key(cmp_max))
```

计数

```
import defaultdict
adict = default(int)
for item in alist:
    adict[item] += 1 #可以很好的规避key不存在导致的报错问题
```

或:

```
from collections import Counter
counter = Counter(alist) #输出Counter({key:value})
```

或:

```
alist.count(item)
```

日期判断

```
from datetime import datetime, timedelta
s = input()
given_date = datetime(int(s[0:4]), int(s[5:7]), int(s[8:]))
days_to_add = int(input())
future_date = given_date+timedelta(days = days_to_add)
print(future_date.strftime('%Y-%m-%d'))
```

素数筛

```
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes
```

矩阵加保护圈

```
dx = [-1, -1, -1, 0, 1, 1,  1,  0]
dy = [-1,  0,  1, 1, 1, 0, -1, -1]

def check(board, y, x):
    c = 0
    for i in range(8):
        nx = x + dx[i]
        ny = y + dy[i]
        c += board[ny][nx]
```

二分查找

```
from bisect import bisect_left, bisect_right
idx = bisect_right(a, x, lo=0, hi=None, *, key=None) #相同元素，x会排列在升序列表的最右端，返回的是x插入的位置
idx = bisect_left(a, x, lo=0, hi=None, *, key=None)  。#相同元素，x会排列在升序列表的最左端，返回的是x插入的位置
# 下面是一个bisect_left样例，说明如何手动实现这一操作

def bisect_left(a, x, lo=0, hi=None, *, key=None):
 if lo < 0:
     raise ValueError('lo must be non-negative')
 if hi is None:
     hi = len(a)
 # Note, the comparison uses "<" to match the
 # __lt__() logic in list.sort() and in heapq.
 if key is None:
     while lo < hi:
         mid = (lo + hi) // 2
         if a[mid] < x:
             lo = mid + 1
         else:
             hi = mid
 else:
     while lo < hi:
```

```
        mid = (lo + hi) // 2
        if key(a[mid]) < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

二分查找变形

> 在一维问题中提到“FJ必有二分”，这里的"FJ"通常指的是“单调性”或“Feasibility Jump（可行性跳跃）”。这种说法源自于某些优化问题中的特性，即如果一个问题具有某种单调性（例如单调递增或递减），那么可以利用二分查找（Binary Search）来高效地解决问题。

```
#判断是否能达到
def can_reach(distance):
    count = 1
    cur = stall[0]
    for i in range(1, n):
        if stall[i] - cur >= distance:
            count += 1
            cur = stall[i]
    return count >= c
#二分查找最大的能达到的距离


def binary_search():
    low = 0
    high = (stall[-1] - stall[0])//(c-1)
    while low <= high:
        mid = (low + high) // 2
        if can_reach(mid):
            low = mid + 1
        else:
            high = mid - 1
    return high


n, c = map(int, input().split())
stall = sorted([int(input()) for _ in range(n)])
print(binary_search())
```

dp 基本模型——最长不下降子序列

```
k = int(input())
s = list(map(int, input().split()))
dp = [1]*k
for i in range(k):
    for j in range(i):
        if s[j] >= s[i]:
            dp[i] = max(dp[i], dp[j]+1)
print(max(dp)) ### 讨论以xxx数字结尾的字符最长有多长
```

或使用二分查找，利用Diworth定理搞定

```
import bisect

def length_of_lis_binary(nums):
    if not nums:
        return 0
    tails = []
    for num in nums:
        pos = bisect.bisect_left(tails, num)
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num
    return len(tails)
```

dp 基本模型——最长回文子串

```
s = input()
dp = [[0]*len(s) for _ in range(len(s))]
for i in range(len(s)):
    dp[i][i] = 1
max_length = 1
for i in range(len(s)-1):
    if s[i] == s[i+1]:
        dp[i][i+1] = 1
        max_length = 2
for length in range(3,len(s)+1):
    for i in range(len(s)-length+1):
        j = i +length -1
        if s[i] == s[j] and dp[i+1][j-1]:
            dp[i][j] = 1
            max_length = length
print(max_length)  ## 可以讨论基本长度为1和长度为2的回文字符串，然后进行推广=
```

dp 基本模型——背包问题

完全背包问题，物品数量无限

```
def min_coins_for_change(amount, coins):
    # 初始化 dp 数组，dp[i] 表示组成金额 i 所需的最少硬币数
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 组成金额 0 所需的硬币数为 0

    # 遍历每个金额 i
    for i in range(1, amount + 1): ### 正向遍历
        # 遍历每个硬币面额 coin
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    # 返回结果
    return dp[amount] if dp[amount] != float('inf') else -1
```

01背包问题，物品只能选或者不选

```
dp=[0]*(B+1)
for i in range(N):
    for j in range(B, w[i] - 1, -1): ###反向遍历
        dp[j] = max(dp[j], dp[j-w[i]]+p[i])

print(dp[-1])
```

相同物品的01背包问题，二进制优化

```
n = int(input())
num = list(map(int, input().split()))
prices = [1,2,5,10,20,50,100] ### 注意数据有没有特殊之处，将数据减小有助于优化程序
n = n//50 ## 特判数据点看能否提前退出
dp = [float('inf')]*(n+1)
dp[0] = 0
for i in range(7):
    price = prices[i]
    max_count = num[i]

    k = 1
    while k <= max_count:
        for j in range(n, price*k-1, -1):
            dp[j] = min(dp[j], dp[j-price*k]+k)
        max_count -= k
        k *= 2 ###二进制可以保证所有值被取到
    if max_count > 0:
        for j in range(n, price*max_count-1, -1):
            dp[j] = min(dp[j], dp[j-price*max_count]+max_count)

if dp[-1] == float('inf'):
    print('Fail')
else:
    print(dp[-1])
```

多重背包问题

二维费用01背包

> 注意数据范围，不同的角度讨论数据效果相同，但是数据范围不同，时间复杂度有量级的区别。

- `dp` 是一个二维列表，`dp[i][j]` 表示在收服了 `i` 个小精灵且皮卡丘剩余 `j` 体力的情况下，小智还剩下的精灵球数量。
- `dp[0][M] = N` 表示初始状态，没有收服任何小精灵，皮卡丘的初始体力为 `M`，小智有 `N` 个精灵球。

```
N, M, K = map(int, input().split())
dp = [[-1] * (M + 1) for i in range(K + 1)]
dp[0][M] = N
for i in range(K):
    cost, dmg = map(int, input().split())
    for p in range(M):
        for q in range(i + 1, 0, -1):
```

```
            if p + dmg <= M and dp[q - 1][p + dmg] != -1:
                dp[q][p] = max(dp[q][p], dp[q - 1][p + dmg] - cost)


def find():
    for i in range(K, -1, -1):
        for j in range(M, -1, -1):
            if dp[i][j] != -1:
                return i, j


captured, remaining_life = find()
print(captured, remaining_life)
```

分类dp--分苹果

```
def apple_distribution(t, cases):
    # 最大苹果数和盘子数
    max_m = max(c[0] for c in cases)
    max_n = max(c[1] for c in cases)

    # 初始化DP数组
    dp = [[0] * (max_n + 1) for _ in range(max_m + 1)]

    # 基础情况
    for m in range(max_m + 1):
        dp[m][1] = 1  # 只有一个盘子
    for n in range(max_n + 1):
        dp[0][n] = 1  # 没有苹果

    # 填表
    for m in range(1, max_m + 1):
        for n in range(2, max_n + 1):
            if n > m:
                dp[m][n] = dp[m][m]  # 盘子多于苹果
            else:
                dp[m][n] = dp[m][n-1] + dp[m-n][n]

    # 处理每个测试用例
    results = []
    for m, n in cases:
        results.append(dp[m][n])

    return results

# 主函数
def main():
    t = int(input())  # 测试数据数目
    cases = []
    for _ in range(t):
        m, n = map(int, input().split())
        cases.append((m, n))
    results = apple_distribution(t, cases)
    for res in results:
        print(res)
```

```
# 样例测试
if __name__ == "__main__":
    main()
```

区间问题

> 按照右端点排序对后面的区间影响最小。

区间合并问题

给出一堆区间，要求**合并**所有**有交集的区间**（端点处相交也算有交集）。最后问合并之后的**区间**。

【步骤一】：按照区间**左端点**从小到大排序。

【步骤二】：维护前面区间中最右边的端点为ed。从前往后枚举每一个区间，判断是否应该将当前区间视为新区间。

假设当前遍历到的区间为第i个区间 $[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间**有交集**。因此**不需要**增加区间个数，但需要设置 $ed = \max(ed, r_i)$。
- $li > ed$: 说明当前区间与前面**没有交集**。因此**需要**增加区间个数，并设置 $ed = \max(ed, ri)$。

选择不相交区间问题

给出一堆区间，要求选择**尽量多**的区间，使得这些区间**互不相交**，求可选取的区间的**最大数量**。这里端点相同也算有重复。

【步骤一】：按照区间**右端点**从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

假设当前遍历到的区间为第i个区间 $[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间有交集。因此直接跳过。
- $l_i > ed$: 说明当前区间与前面没有交集。因此选中当前区间，并设置 $ed = r_i$。

区间选点问题

给出一堆区间，取**尽量少**的点，使得每个区间内**至少有一个点**（不同区间内含的点可以是同一个，位于区间端点上的点也算作区间内）。

【步骤一】：按照区间**右端点**从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

假设当前遍历到的区间为第i个区间$[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- $l_i > ed$: 说明当前区间与前面没有交集。因此选中当前区间，并设置 $ed = r_i$。

区间覆盖问题

给出一堆区间和一个目标区间，问最少选择多少区间可以**覆盖**掉题中给出的这段目标区间。

【步骤一】：按照区间**左端点**从小到大排序。

【步骤二】：**从前往后**依次枚举每个区间，在所有能覆盖当前目标区间起始位置start的区间之中，选择**右端点**最大的区间。

假设右端点最大的区间是第i个区间，右端点为r_i。

最后将目标区间的start更新成r_i

区间分组问题

给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。

【步骤一】：按照区间**左端点**从小到大排序。

【步骤二】：从**前往后**依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

（即判断是否存在某个组的右端点在当前区间之中。如果可以，则不能放到这一组）

假设现在已经分了m 组了，第 k 组最右边的一个点是 r_k，当前区间的范围是$[L_i, R_i]$。则：

如果$L_i \leq r_k$ 则表示第i 个区间无法放到第 k 组里面。反之，如果 $L_i > r_k$，则表示可以放到第k 组。

- 如果所有 m 个组里面没有组可以接收当前区间，则当前区间新开一个组，并把自己放进去。
- 如果存在可以接收当前区间的组 k，则将当前区间放进去，并更新当前组的 $r_k = R_i$。

dfs模板与数据一次性读取

```
import sys
# input = sys.stdin.read 设置一次性读入所有数据
sys.setrecursionlimit(20000)
def dfs(x, y):
    # 标记当前位置为已访问
    field[x][y] = '.'
    # 遍历8个方向
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        # 检查新位置是否在地图范围内且未被访问
        if 0 <= nx < n and 0 <= ny < m and field[nx][ny] == 'W':
            dfs(nx, ny)
# 一次性读取所有输入
# data = input().split()
# n, m = map(int, data[:2])
# field = [list(row) for row in data[2:2+n]]
n, m = map(int, input().split())
field = [list(input()) for _ in range(n)]
# 初始化8个方向
directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1,
1)]
# 计数器
cnt = 0
# 遍历地图
for i in range(n):
    for j in range(m):
```

```
        if field[i][j] == 'W':
            dfs(i, j)
            cnt += 1
print(cnt)
```

dfs 提前退出+sticks模板

```
@lru_cache(maxsize=8192)  # 使用 LRU 缓存优化，尝试maxsize看是否能过
def f(idx, test):
    if idx == len(s):
        return all(x == per_tot for x in test)  # 检查所有部分是否都等于目标值

    for j in range(len(test)):
        if j > 0 and test[j] == test[j - 1]:  # 避免重复尝试
            continue
        if test[j] + s[idx] <= per_tot:  # 当前部分可以放入当前元素
            next_test = list(test)
            next_test[j] += s[idx]
            next_test = tuple(sorted(next_test))  # 转换为元组以支持缓存
            if f(idx + 1, next_test):  # 递归调用
                return True
    return False
```

全排列组合

```
import itertools
s = list(map(int, input().split()))
r = int(input())
result = list(itertools.permutations(s, r)) ###生成的是列表加元组的形式，元素出现的顺序是按照s中元素的顺序出现的
result = list(itertools.combinations(s, r))###组合
```

bfs模板

```
from collections import deque
def bfs(start, end):
    q = deque([(0, start)])  # (step, start)
    in_queue = {start}
    while q:
        step, front = q.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q，并加入集合in_queue设置为已入队
```

global变量compile error

```
#pylint:skip-file 跳过OJ静态检查
```

Manacher 算法

借助回文串的对称性进行优化，尽可能应用对称性进行递归。

```
class Solution:
    def expand(self, s, left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return (right - left - 2) // 2
    def longestPalindrome(self, s: str) -> str:
        end, start = -1, 0
        s = '#' + '#'.join(list(s)) + '#'
        arm_len = []
        right = -1
        j = -1
        for i in range(len(s)):
            if right >= i:
                i_sym = 2 * j - i
                min_arm_len = min(arm_len[i_sym], right - i)
                cur_arm_len = self.expand(s, i - min_arm_len, i + min_arm_len)
            else:
                cur_arm_len = self.expand(s, i, i)
            arm_len.append(cur_arm_len)
            if i + cur_arm_len > right:
                j = i
                right = i + cur_arm_len
            if 2 * cur_arm_len + 1 > end - start:
                start = i - cur_arm_len
                end = i + cur_arm_len
        return s[start+1:end+1:2]
```

Kadane's algorithm --最长子序列

```
def max_subarray_sum(arr):
    if not arr:
        return 0
    max_current = max_global = arr[0]
    for num in arr[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current

    return max_global
```

NP算法--下一个序列

```
def NP(nums):
    for i in range(len(nums)-2,-1,-1):
        if nums[i]<nums[i+1]:
            for j in range(len(nums)-1,i,-1):
                if nums[j]>nums[i]:
                    nums[j],nums[i] = nums[i],nums[j]
                    tmp=nums[len(nums)-1:i:-1]
                    nums[i+1:]=tmp
                    return nums
    else:
        nums.reverse()
        return nums
print(NP([4,2,6,3]))
```

懒删除、懒更新

每次pop元素时检查以下这个元素是否在我们研究的范围内，同时结合集合搜索加快速度。

```
import heapq
from collections import defaultdict
out = defaultdict(int)
pigs_heap = []
pigs_stack = []
while True:
    try:
        s = input()
    except EOFError:
        break

    if s == "pop":
        if pigs_stack:
            out[pigs_stack.pop()] += 1
    elif s == "min":
        if pigs_stack:
            while True:
                x = heapq.heappop(pigs_heap)
                if not out[x]:
                    heapq.heappush(pigs_heap, x)
                    print(x)
                    break
                out[x] -= 1
    else:
        y = int(s.split()[1])
        pigs_stack.append(y)
        heapq.heappush(pigs_heap, y)
```

单调栈

维护一个单调下降的序列。

```
class Solution:
    def trap(self, height: List[int]) -> int:
        stack = []
        water = 0
        for i in range(len(height)):
            while stack and height[i] > height[stack[-1]]:
                top = stack.pop()
                if not stack:
                    break
                distance = i - stack[-1] - 1
                bounded_height = min(height[i], height[stack[-1]]) - height[top]
                water += distance * bounded_height
            stack.append(i)
        return water
```

单调栈实例：寻找最大矩形面积

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        left, right = [0] * n, [n] * n

        mono_stack = list()
        for i in range(n):
            while mono_stack and heights[mono_stack[-1]] >= heights[i]:
                right[mono_stack[-1]] = i
                mono_stack.pop()
            left[i] = mono_stack[-1] if mono_stack else -1
            mono_stack.append(i)

        ans = max((right[i] - left[i] - 1) * heights[i] for i in range(n)) if n >
0 else 0
        return ans
```

Dijkstra算法

找最短路径：按照短路径进行操作。

```
import heapq

def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
```

```
    distances[s] = 0

    while pq:
        dist, node = heapq.heappop(pq)
        if node == t:
            return dist
        if node in visited:
            continue
        visited.add(node)
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                new_dist = dist + weight
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))
    return -1

# Read input
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]

# Solve the problem and print the result
result = dijkstra(n, edges, s, t)
print(result)
```

Huffman 剪绳子 heapq 就是好用

```
import bisect
N=int(input())
ribbons=sorted(list(map(lambda x:-int(x),input().split())))
mini=0
for i in [0]*(N-1):
    A=ribbons.pop()
    B=ribbons.pop()
    mini-=A+B
    bisect.insort(ribbons,A+B)
print(mini)
```

01矩阵

```
from typing import List
from collections import deque

class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        dp = [[float('inf')] * n for _ in range(m)]
        queue = deque()

        # 将所有0的元素加入队列并初始化dp数组
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    dp[i][j] = 0
```

```
                    queue.append((i, j))

        # 定义四个方向的移动
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        # BFS开始
        while queue:
            x, y = queue.popleft()
            # 对当前点的四个方向进行处理
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n:
                    # 如果新位置的dp值可以更新（即发现更短的路径）
                    if dp[nx][ny] > dp[x][y] + 1:
                        dp[nx][ny] = dp[x][y] + 1
                        queue.append((nx, ny))

        return dp
```

滑动窗口

双指针，一边记录最长的长度。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # 初始化变量
        start = -1  # 当前无重复子串的起始位置的前一个位置
        max_length = 0  # 最长无重复子串的长度
        char_index = {}  # 字典，记录每个字符最近一次出现的位置

        # 遍历字符串
        for i, char in enumerate(s):
            # 如果字符在字典中且上次出现的位置大于当前无重复子串的起始位置
            if char in char_index and char_index[char] > start:
                # 更新起始位置为该字符上次出现的位置
                start = char_index[char]

            # 更新字典中字符的位置
            char_index[char] = i

            # 计算当前无重复子串的长度，并更新最大长度
            current_length = i - start
            max_length = max(max_length, current_length)

        return max_length
```

并查集

递归寻找根节点

```
def find(x):
    if parent[x] != x: # 如果不是根结点，继续循环
        parent[x] = find(parent[x])
    return parent[x]
```

```
def union(x, y):
    parent[find(x)] = find(y)

n, m = map(int, input().split())
parent = list(range(n + 1)) # parent[i] == i，则说明元素i是该集合的根结点

for _ in range(m):
    a, b = map(int, input().split())
    union(a, b)

classes = set(find(x) for x in range(1, n + 1))
print(len(classes))
```

game 策略

```
s = input()
if s[0] == 'R':
    dp1 = [0]
    dp2 = [1]
else:
    dp1 = [1]
    dp2 = [0]
for i in range(1, len(s)):
    if s[i] == 'R':
        dp1.append(min(dp1[-1], dp2[-1]+1))
        dp2.append(min(dp1[-1]+1, dp2[-1]+1))
    else:
        dp1.append(min(dp1[-1]+1, dp2[-1]+1))
        dp2.append(min(dp2[-1], dp1[-1]+1))
print(dp1[-1])
```

通过交替输出True和False实现交替判断谁赢谁输。

小技巧

前缀和，优化求和的时间复杂度

用字典查找

双指针作用小结，找灵感用

1. **寻找两数之和**：例如，在一个有序数组中找到两个数使它们的和等于给定的目标值。
2. **合并两个有序数组**：例如，合并两个已排序的数组。
3. **删除重复元素**：例如，在一个数组中移除重复的元素。
4. **反转字符串**：例如，反转一个字符串中的字符。
5. **滑动窗口**：例如，在一个数组中找到满足特定条件的最长子数组。

判断是否是偶数：a&1 = 1 if a is an odd number else 0

greedy

炸鸡排

思路：最理想情况是每块鸡排都炸完,用时为所有鸡排用时的平均值 如果耗时最长的鸡排比其他鸡排的平均用时更长,说明无论其他鸡排怎么摆都会使那块鸡排炸不完, 那索性那块鸡排就一直放那炸,考虑剩下的鸡排 否则就可以达到最理想情况:平均用时

CPU 调度

可以考虑有一个进程的读写数据时间特别长，那自然是它越早开始越好。而且它读写的时候，其他进程还可以并行计算。

为什么只需要按照写文件时间排序？ 计算时间 (compute[i]) 是占用 CPU 的独占时间，调度顺序只会影响当前的 CPU 执行时间。 写文件时间 (write[i]) 可以与其他任务的计算时间重叠，因此 尽早开始写文件时间更长的任务，能最大程度地减少所有进程的完成时间。 因此，贪心策略的核心就是：按照写文件时间从大到小排序。