减少耗时的小 tips

1. 使用生成器
2. 内置函数 (map, filter, sorted)
3. 数据结构

集合：快速查找 添加 删除 → 适用：查找、去重、交并

字典：快速对应 查找 增删 → 关联数据

列表：插入 删除 O(n)！

4. 优化循环：减少循环内部的运算，将不变计算移至外部
5. 避免全局变量：在高频调用情况下，将其值存在局部变量中
6. 减少函数的调用：对于复杂的计算函数，在循环中多次调用 → 逻辑写在循环中
7. str.join() 代替 + 连接字符串
8. 列表推导式的使用

9. 重用对象 避免不必要变量创建
10. 切片 快于手动循环
11. 分批次处理大数据集
12. 限制异常处理
13. import sys

```
input = sys.stdin.read
data = input().split()  # 读取所有并分割为列表
```

针对标准多量手输入数据，以此法一次性读取

编程思路

1. 要求多次输出不同的 i, j 下，i 至 j 之间满足条件的个数

⇒先将和从头至第 m 个的满足个数存入列表，需要时调用相减

2. 前缀和的算法（加快时间）　　适于范围和及多次需要求和有用

对于一个数字列表 arr

```
n=len(arr)
prefix_sum=[0]*(n+1)
for i in range(n):
    prefix_sum[i+1]=prefix_sum[i]+arr[i]
```

#得到 prefix_sum(第 i 项表 arr 前 i 项和)

循环中比较前一个值与后一个值大小关系判断情况

例，罗马数字转整数

```
roman_map={'I':1, ... , 'M':1000}
total, prevalue=0,0
for char in s
    value = roman_map[char]
    if value > prev_value    比较判断  ☆处理减法情况
        total += value - 2*prev_value
    else:
        total += value
        prev_value = value    更新前一个数
```

4. 使处理过的元素不会再被处理

```
used = [0]*n
for i in range(n):
    if used[i]:
        continue
    #处理代码…
    used[i]=1
```

（0布尔值为False 被处理过 | 则为True，continue中当前循环剩余部分，直接开始下一次循环）

5. 排序思路　任何A更高的数据 B高于X
任何B更低的数据 A更低

找（Ax, Bx）构成列表排序
（Ax为主排列指标）

则从Ax为标准，Ax一定是在后遍历中，找到前面所有中Bx的最值（不断更新）与下一项比较

例：酒店预约 手作梦

```
candiates=1
max_cost=hotels[0][0]
for i in range(n):
    if hotels[i][1] < max_cost:
        condiates+=1
        max_cost = hotels[i][1]
```

6. 最多的不重叠区间问题

按照区间右端点排序，遍历排序后列表
选择最邻近的左端点，
（若当前区间左端 >= 已选区间右端，则选择该区间，
并为后面腾出更多空间）

#表a会排在b后

7. 自定义比较函数排序

```
from functools import cmp_to_key
def compare(a,b):
    if …
        return -1
    # 表示a会排在b前
    elif …
        return 1
```

```
else:
    return 0  # 表顺序无所谓
sorted_numbers = sorted(numbers, key=cmp_to_key(compare))
```

7. 复杂的图形问题 递归

用全局列表，分别定义放置函数与主函数

```
def f(n, x, y, mp):
        维数 起始标 矩阵
def main()
if __name__=="__main__":
    main()
```

8. 记忆化 防止递归出现重复结果 记忆{处理过的数据 / 得到的结果

9. 用字典作为dp的二维数组

s[键]=setdefault(键,默认值)+某个数字

0

10. 保护圈输入

```
m=[[0 for _ in range(m+2)] for _ in range(n+2)]
for i in range(1,n+1):
    m[i][1:-1]=input()
```

11. and操作符在Python中具有短路行为，A and B

如果A为假，则B不会被评估 ⇒ 防止潜在错误：越

12. 在记录数字次数递归时 可用全局变量

```
global ans
ans+=1
return
```

不像list有类型码和空隙

☆ array中，每个元素都是固定大小、连续存储，
在访问、更改的值、节约用时，内存有优势
（相对list），
但对 pop()，
remove()，
append()
耗时更多

11. array的使用

```
import array
arr = array.array('i', [1,2,3,4,5])
```
#一般都选'i'，较大范围的整数

12. orderedDict 记录插入顺序

```
from collections import OrderedDict
a = OrderedDict()
x,y = a.popitem()
```

13. 对于数字可能较大的题目，若题目允许输出取模后的结果，则在每一个计算中先取模再以节约内存

14. 窗口问题 动态调整窗口，双指针 扩展+收缩 → 连续子数组/子字符串

典型模式：右指针先扩大，直到满足特定条件/或框定右边界

左指针再收缩，缩小窗口范围至

满足条件或优化

通常右指针是循环变量，左指针动态调整

作减法的动态规划

求满足某个限制的安排/路径总数

例 将一个正整数n分成若干部分，每部分不能大于k，并至少包含一个大于等于d的部分

和为目标值的最小连续
长度，最长无重复子字符
替换字符后
长重复
最小覆盖子

```
A = [1] + [0]*n    B = [1] + [0]*n
for i in range(1, n+1):
    for j in range(1, min(i,k)+1): 全不大于k的划分
        A[i] += A[i-j]
    for j in range(1, min(d, i+1)): 全不大于d的划分
        B[i] += B[i-j]
```

例2. 换电站 Dp = [0]*60 不能连续放m个材料求总方案数

```
for i in range(1, n+1):
    if i < m:
        dp[i] = dp[i-1]*2
    elif i == m:
        dp[i] = dp[i-1]*2 - 1
    else:
        dp[i] = dp[i-1]*2 - dp[i-m-1]
```

☆总方案减去坏的方案

16. 用栈处理符号匹配 括号匹配对 后进先出

```
stack = []
mapping = {')':'(', '}':'{', ']':'['}
for char in s:
    if char in mapping:
        top = stack.pop() if stack else '#'
        if mapping[char] != top:
            return False
    else:
        stack.append(char)
```

17. 贪心+后悔 ⇒ 路径优化中 部分路径代价可能动态变化
保留变化时代替方案

最小覆盖子串 在字符串s中找到最小的含t所有字符的子串

```
from collections import Counter
need=Counter(t) #统计t中每个需要的次数
window={} #统计目前已有窗口中各次数
left,right=0,0
valid=0 #窗口中满足need的字符数
start,length=0,float('inf') #最优起点与长度
while right<len(s):
    char=s[right] #当前右指针字符
    right+=1 #移动右指针扩展窗口
    if char in need:
        window[char]=window.get(char,0)+1
        if window[char]==need[char]:
            valid+=1
    while valid==len(need):
        if right-left<length: #更新最优窗口
            start=left
            length=right-left
        char=s[left]
        left+=1
        if char in need:
            if window[char]==need[char]:
                valid-=1
            window[char]-=1
return [start:st
```

初始化

若当前字符为t中字符

更新答案

移动左指针并更新至优化

和为k的连续子数组个数

```
count=0
current_sum=0
prefix_sum={0:1}
for num in nums:
    current_sum+=num
    if current_sum-k in prefix_sum:
        count+=prefix_sum[current_sum-k]
    prefix_sum[current_sum]=prefix_sum.get(current_sum,0)+1
return count
```

18. 哈夫曼编码算法（压缩合并文件代价最小化

多个文件合并，每次只能合并两个，代价是大小之和

⇒ 用 heapq 每次合并最小的两个

→ 最优通法：使用 defaultdict 记录前缀和及索引

还可解决平均值问题等

给定

☆连续子数组和为给定 求最长

模板

```
from collections import defaultdict
def find_target_sum(n,sums,target):
    prefix_sum, max_len = 0,0
    sum_indices= defaultdict(int)
    sum_indices[0]=-1        # 允许前缀和为0，位置-1
    for i,num in enumerate(nums):
        prefix_sum+=sum
```

```
        if prefix_sum - target in sum_indices:
            max_length = max(max_len, i - sum_indices[prefix_sum - target])
        if prefix_sum not in sum_indices:
            sum_indices[prefix_sum] = i
    return max_len
```

推了 ① 连续子数组和 = 目标倍数 ⇒ 取模

② 连续子数组很长为目标 ⇒ 前缀和

③ 连续子数组和在范围内 ⇒ 遍历两次 ⇒ 取差集

小于等于 high　大于等于 low-1

19. 将字典化为键值对元组

```
items_list = list(my_dict.items())
```

20. 对输入中不好用想换成其他形式(欺骗)，可在输入时写成迭代器

```
*records = map(int, input.split())
arr = [(records[i], records[i+1]) for in range(0, 2*n, 2)]
```

21. 自定义比较函数：可处理整数拼接等

```
from functools import cmp_to_key
def compare(a,b):
    return int(b+a) - int(a+b)   # 规定：为负则a在b前，为0相等
nums.sort(key=cmp_to_key(compare))
```

22. 书写搜索题时，一定要把自身位置(针对原地)，终点位置加到检查条件

20、二分搜索+贪心算法

① 将其转化为决策性问题：给定目标值，判断能否满足约束条件

② 二分搜索优化目标值，贪心算法验证可行性

☆ 适用于 问题的可行性具有单调性（由大到小的目标值，对应满足约束的难易程度由易到难）

可行性：可贪心验证

举例：① 分配/分组问题　n个任务分给k个工人，每个任务时间不同，求最大组和的最小值　要求最小化分配给单个工人最大工时

核心算法：设定目标值为上下界进行二分查找，保持下界可行而上界不断缩，更新值至上下界相等。　☆与排序无关

```
def check(nums, max_sum, k):
    current_sum=0
    group_count=1 #至少一个
    for num in nums:
        if current_sum + sum > max_sum:
            group_count += 1
            current_sum = num
            if group_count > k:
                return False
        else:
            current_sum += num
    return True
left, right = max(nums), sum(nums)
while left < right:
    mid=(left+right)//2 #当前尝试的最大组和
```

n个组 不行 最小 下界

1个组 可以 最大 上界

最值的最值

No.

Date / /

if check(nums, mid, k):
    right = mid # 缩小上界
else:
    left = mid + 1 # 增大下界
left 为最小的最大组和

跳石头 删除、

主循环 lo, hi = 0, L+1
ans = -1
while lo < hi:
    mid = (lo + hi) // 2
    if check(mid):
        hi = mid
    else:
        ans = mid
        lo = mid + 1
Print(ans)

二分查找

```
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
```

递归

① 函数调用自己

② 基线条件

```
def countdown(i):
```

☆ CF中的 Tags

1. dp 动态规划 2. greedy 贪心 3. math
4. ds (Data Structures) 用到数据结构
5. graphs 图论相关
6. sortings 排序 (冒泡~, 选择~, 插入~, 快速~)
7. constructive algorithms 构造性找出解 8. strings 字符串
9. combinatorics：组合数学 (计数，排列·组合
10. number theory 数论
11. two pointers 双指针法

☆ 浅拷贝

修改原可变对象时，浅拷贝对应对象也会受影响

例. [[0]*n] * m 创建的二维数组

修改一个影响所有行

改变 matrix = [[0]*m for _ in range(n)]

☆ 无穷大

浮点数的无穷大 float("inf") +∞

float("-inf") -∞

KOKUYO

☆ 使用 set 和 dict 进行成员测试

O(1)　　列表 O(n)

print (666 in set(numbers))

☆ 访问全局变量慢于访问局部变量慢

☆ n^=1　　^= 按位异或赋值

result = a^b 比较二进制下a,b每一个位，相同为0，不同为1，得到一个新的二进制数，转整数赋给 result

n^=1 对n与1进行按位异或运算并赋给n

☆运用：开关转换　n=0　if…　n^=1

适合多次进行的两种状态互换

二分查找

```
def binary_search(arr, target):
    left, right = 0, (len(arr)-1)
    while left <= right:
```

递归

① 函数调用自己
② 基线条件

```
def countdown(i):
```

寻找素数的欧拉筛

```
def oula(r)
    prime = [0 for i in range(r+1)]#初始化为0便于筛选
    common = [ ]  #存放素数
    for i in range(2, r+1):
        if prime[i] == 0
            common.append(i)
        for j in common:
            if i*j > r:
                break
            prime[i*j] = 1  #将重复筛选剔除
            if i%j == 0:
                break
    return common
prime = oula(20000)
```

在仅需要判断是否为质数时，只需用朴素筛选

```
def isprime(n):
    for i in range(2, int(n**5)+1):
        if n%i == 0:
            return False
    else:
        return True
```

二分查找

```
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 left
```

快速排序

① 分而治之 D&C

原理：① 找出简单的基线条件（为空/只有一个元素）

② 确定如何缩小问题规模

② 快速排序

```
def quicksort(array):
    if len(array)<2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

递归

① 函数调用自己

② 基线条件

```
def countdown(i):
    print i
    if i <= 0:        # 基线条件
        return
    else:             # 递归条件
        countdown(i-1)
```

※ 连接字符串（想输出 抱着…我的鱼…的

```
def word(n):
    if n==0
        return "我的鱼"
    else:
        return '抱着' + word(n-1) + '
```

有时也用 print 递归

```
def story(n)
    if n=0
        print('0')
    else:
        print('n')
        print(n-1)
        story(n-1)
```

广度优先搜索

①列表实现图

```
graph={}
graph['you']=['A','B','C']
```

②例：找芒果专卖家

```
from collections import deque
search_queue=deque()  创建了一个队列
search_queue+=graph["you"]
searched=[]
while search_queue:
    person=search_queue.popleft()  从队列左端取出一个人
    if person_is_seller(person):
        return person+' is seller'
    else:
        search_queue+=graph[person]
return False
```

防止互为朋友被反复加入无限循环 ← searched=[]

if person not in searched

狄克斯特拉算法（加权图）

不含负权

```
graph={}
graph['start']={}
graph['start']['a']=6
graph['start']['b']=2
graph['a']={}
...
graph['fin']={}
```

实现加权图

父节点统计 parents={}

[a] [b] [fin]

用过除 processed=[]

```
node=find_lowest_cost_node(c
while node is not None:
    cost=costs[node]
    neighbors=graph[node]
    for n in neighbors.key
```

所有都被处理后结束

遍历邻居

实现距离统计

```
infinity=float("inf")
costs={}
costs['a']=...['b']=['fin']
```

递归

1. 判断行型递归

判断i是否为回文

```
def check(s,i,j)
    if i >=j
        return True
    reture s[i]==s[j] and check(s,i-1,j+1)
```

2. 汉诺塔

```
def move(n, from_rod, to_rod, mid_rod)
    if n==0:
        return
    move(n-1, from_rod, mid_rod, to_rod)
    print(f"{from_rod} -> {ro_rod}")
    move(n-1, mid_rod, to_rod, from_rod)
move(n,'A','B','C')
```

3. 八皇后(n)

```
def solve_n_queens(n):
    result=[]
    queens=[-1]*n
    def is_safe(row,col)
        for i in range(row):
        if queens[i]==col or abs(queens[i]-col)=row-i:
            return False
        return True
```

```
def place_queen(row):
    if row == n:
        result.append(queens[:])
        return
    for col in range(n):
        if is_safe(row,col):
            queens[row] = col
            place_queen(row+1)
            queens[row] = -1
place_queen(0)
return result
```

☆有规律的绘画问题 确定起始点、迭代/行进方向、找层次递进

☆连通分量计数算法

当发现一个色块，递归搜索它的连接色块

```
dire = [[-1,0],[1,0],[0,-1],[0,1]]
def dfs(x,y,c):
    m[x][y] = "#"
    for i in range(len(dire)):
        tx = x+dire[i][0]
        ty = y+dire[i][1]
        if m[tx][ty] == c:
            dfs(tx,ty,c)
```

```
for i in range(1,n+1):
    for j in range(1,n+1):
        if m[i][j] == 'r':
            dfs(i,j,'r')
            rt+=1
        if m[i][j] == 'b':
            dfs(i,j,'b')
            bt+=1
```

☆递归程序优化 递归深度上限增加

缓存返回值

```
import sys
sys.setrecursionlimit(1<
from functools import lru_cac
@lru_cache(maxsize=None)
```

☆使用栈模拟递归

```
def simulate_recursion(基本情况):
    stack = [初始状态,] # 一般含一个结果参数
    while stack
        current_state = stack.pop()
        if 基础情况:
            result = ...
            return result
        #处理当前状态
        #计算下一个状态并压入栈
        next_states = push_next_states(current_state)
        for state in reversed(next_states): #反转保护顺序
            stack.append(state)
```

回文字符串常见算法

① 中心扩展法 $O(n^2)$ 时 $O(1)$空

```
def expand(left, right):
    while left>=0 and right<len(s) and s[left]=s[right]:
        left-=1
        right+=1
    return left+1, right-1
start, end=0,0
for i in range(len(s)):
    l1, r1=expand(i,i)
    l2, r2=expand(i, i+1)
    if r1-l1>end-start:
        start, end=l1, r1
    if r2-l2>end-start:
        start, end=l2, r2
print(s[start:end+1])
```

② dp $O(n^2)$ $O(n^2)$ 需要额外信息(计数/所有回文串)可用

```
n=len(s)
if n==0
    return ""
dp=[[False]*n for _ in range(n)]
for i in range(n):
    dp[i][i]=True
for i in range(n-1):
    if s[i]==s[i+1]:
        dp[i][i+1]=True
        start, max_length=i,2
for length in range(3,n+1):
    for i in range(n-length+1):
        j=i+length-1
        if s[i]=s[j] and
            dp[i+1][j-1]:
            dp[i][j]=True
            start, max_length=i
return s[start:start+max
```

马拉车算法 Manacher

统一处理奇回文/偶回文，利用已有回文减少计算

```
def manacher(s):
    T = "#" + "#".join(s) + "#"
    n = len(T)
    P = [0] * n          # P[i]表：从T[i]为中心的回文半径
    C, R = 0, 0          # 当前中心、右边界
    max_len = 0
    center = 0
    for i in range(n):
        mirror = 2*C - i     # 获取对称位置
        if i < R:
            P[i] = min(R - i, P[mirror])   # 对称性生成减少计算
        while i + P[i] + 1 < n and i - P[i] - 1 >= 0 and
              T[i + P[i] + 1] == T[i - P[i] - 1]:
            P[i] += 1        # 尝试扩展
        if i + P[i] > R:
            C, R = i, i + P[i]
        if P[i] > max_len:
            max_len = P[i]
            center = i
    start = (center - max_len) // 2
```

转换字符串

初始化回文半径数组

结束