Cheatsheet

读取输入

```
while True:
    try:
        line = input()
        lines.append(line)
    except EOFError:  # 按Ctrl+D (Unix) 或 Ctrl+Z+Enter (Windows)
        break

import sys
lines = sys.stdin.read().splitlines()  # 读取所有行
lines = sys.stdin.readlines()          # 包含换行符
lines = [line.strip() for line in sys.stdin]  # 推荐
```

列表的语法

```
# 切片操作 [start:end:step]
sublist1 = lst[1:3]       # ['b', 'c']（包含1，不包含3）
sublist2 = lst[:3]         # ['a', 'b', 'c']（从头到3）
sublist3 = lst[2:]         # ['c', 'd', 'e']（从2到尾）
sublist4 = lst[::2]        # ['a', 'c', 'e']（步长为2）
sublist5 = lst[::-1]       # ['e', 'd', 'c', 'b', 'a']（反转列表）

# 获取索引
index = lst.index('c')     # 2（返回第一个匹配的索引）
# 注意：元素不存在会抛出 ValueError
index = lst.index('c', 0, 3)  # 在子范围[0,3)内查找
lst = [1, 2, 3, 4, 5]

#反差
lst = [3, 1, 4, 1, 5, 9]
lst.reverse()              # 反转: [9, 5, 1, 4, 1, 3]
lst.sort()                 # 升序排序: [1, 1, 3, 4, 5, 9]
lst.sort(reverse=True)     # 降序排序: [9, 5, 4, 3, 1, 1]
new_sorted = sorted(lst)   # 返回新排序列表，不改变原列表
new_reversed = list(reversed(lst))  # 返回新反转列表
lst = [1, 2, 2, 3, 4, 2]

# 长度
length = len(lst)          # 6

# 计数
count = lst.count(2)        # 3（值为2的元素个数）

# 查找索引
index = lst.index(3)        # 3（元素3的索引）
```

```python
# 复制列表
copy1 = lst.copy()          # 浅拷贝
copy2 = lst[:]              # 切片拷贝
copy3 = list(lst)           # 构造函数拷贝

# 合并列表
lst1 = [1, 2, 3]
lst2 = [4, 5, 6]
combined = lst1 + lst2      # [1, 2, 3, 4, 5, 6]
lst1.extend(lst2)           # lst1变为[1,2,3,4,5,6]

# 列表乘法
repeated = [0] * 5          # [0, 0, 0, 0, 0]
repeated = lst1 * 2         # 重复列表: [1,2,3,1,2,3]
# 基本形式
squares = [x**2 for x in range(5)]        # [0, 1, 4, 9, 16]

# 带条件的推导式
evens = [x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]

# 多个for循环
pairs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

# 嵌套列表推导式
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]  # [1,2,3,4,5,6,7,8,9]

# 复杂表达式
result = [(x, x**2) for x in range(6)]  # [(0,0), (1,1), (2,4), ...]
```

一些零碎

```python
# 常用ASCII范围
# 0-31: 控制字符
# 32-47: 标点符号 !"#$%&'()*+,-./
# 48-57: 数字 0-9
# 58-64: :;<=>?@
# 65-90: 大写字母 A-Z
# 91-96: [\]^_`
# 97-122: 小写字母 a-z
# 123-126: {|}~

# 字符与ASCII码互转
char = 'A'
ascii_code = ord('A')        # 65
char_from_code = chr(65)     # 'A'
# 字符类型判断
print('H'.isalpha())     # True 是否是字母
print('5'.isdigit())     # True 是否是数字(0-9)
print('5'.isnumeric())   # True 是否是数字字符(包括中文数字)
print('Ⅳ'.isnumeric())   # True 罗马数字4
print('a'.islower())     # True 是否小写
```

```python
print('A'.isupper())     # True 是否大写
print('Hello'.istitle()) # True 是否每个单词首字母大写
print(' \t\n'.isspace()) # True 是否空白字符
print('abc123'.isalnum())# True 是否字母或数字
print('hello'.isascii()) # True 是否ASCII字符

# 实用函数
def is_printable_ascii(c):
    """判断是否可打印ASCII字符"""
    return 32 <= ord(c) <= 126
text = "Hello World 123"

# 基本转换
print(text.upper())       # HELLO WORLD 123
print(text.lower())       # hello world 123
print(text.title())       # Hello World 123
print(text.capitalize())  # Hello world 123
print(text.swapcase())    # hELLO wORLD 123


# 不同进制的表示
binary = 0b1010      # 二进制: 10
octal = 0o12         # 八进制: 10
hexadecimal = 0xA    # 十六进制: 10
decimal = 10         # 十进制: 10

# 进制转换函数
num = 42

# 转换为不同进制的字符串
bin_str = bin(num)       # '0b101010' 二进制
oct_str = oct(num)       # '0o52'     八进制
hex_str = hex(num)       # '0x2a'     十六进制


import math
# 幂和对数
print(math.pow(2, 3))    # 8.0
print(math.sqrt(16))      # 4.0
print(math.exp(1))        # e ≈ 2.718
print(math.log(10))       # 自然对数 ≈ 2.302
print(math.log10(100))    # 2.0
print(math.log2(8))       # 3.0
# 绝对值
print(math.fabs(-5.5))     # 5.5（返回浮点数）
print(abs(-5.5))           # 5.5（内置函数，保持类型）

# 最大公约数和最小公倍数
print(math.gcd(12, 18))    # 6
print(math.lcm(12, 18))    # 36（Python 3.9+）

# 阶乘
print(math.factorial(5))   # 120
```

```
# 组合与排列
print(math.comb(5, 2))      # 10（C(5,2)）
print(math.perm(5, 2))      # 20（P(5,2)）

# 距离和角度
print(math.hypot(3, 4))     # 5.0（欧几里得距离）
```

多种输出方法整理

```
# 基本输出
print("Hello, World!")

# 输出多个值（自动用空格分隔）
name = "张三"
age = 20
print("姓名:", name, "年龄:", age)  # 输出: 姓名: 张三 年龄: 20

# 修改分隔符
print("2024", "12", "25", sep="-")  # 输出: 2024-12-25

# 修改结束符（默认是换行）
print("Hello", end=" ")  # 不换行
print("World")  # 输出: Hello World

# 2. str.format() 方法
print("学生: {}, 分数: {:.2f}".format(name, score))
print("学生: {0}, 分数: {1:.2f}".format(name, score))  # 带索引
print("学生: {name}, 分数: {score:.2f}".format(name=name, score=score))

name = "王五"
age = 25
score = 88.8888

# 基本用法
print(f"姓名: {name}, 年龄: {age}")  # 输出: 姓名: 王五, 年龄: 25

# 数字格式化
print(f"分数: {score:.2f}")    # 保留2位小数: 分数: 88.89
print(f"分数: {score:.0f}")    # 无小数: 分数: 89
print(f"百分比: {0.4567:.1%}")  # 百分比: 45.7%

# 对齐和填充
print(f"姓名: {name:<10} 年龄: {age}")  # 左对齐，宽度10
print(f"分数: {score:>10.2f}")          # 右对齐，宽度10
print(f"分数: {score:^10.2f}")          # 居中对齐
print(f"分数: {score:*>10.2f}")         # 右侧*填充: *****88.89

# 表达式计算
a, b = 10, 3
print(f"{a} + {b} = {a + b}")  # 输出: 10 + 3 = 13
print(f"{a} / {b} = {a / b:.2f}")  # 输出: 10 / 3 = 3.33
```

```python
my_list = [1, 2, 3, 4, 5]

# 1. 直接打印列表
print(my_list)  # 输出: [1, 2, 3, 4, 5]

# 2. 遍历输出每个元素
for item in my_list:
    print(item)  # 每个元素一行

# 3. 不带中括号的输出
print(*my_list)  # 输出: 1 2 3 4 5
print(*my_list, sep=", ")  # 输出: 1, 2, 3, 4, 5

# 4. 用join（仅适用于字符串元素）
str_list = ["apple", "banana", "cherry"]
print(", ".join(str_list))  # 输出: apple, banana, cherry

# 5. 数字列表转字符串列表输出
num_list = [1.1, 2.2, 3.3]
# 方法1: map + join
print(", ".join(map(str, num_list)))  # 输出: 1.1, 2.2, 3.3
# 方法2: 列表推导式
print(", ".join([f"{x:.2f}" for x in num_list]))  # 输出: 1.10, 2.20, 3.30

# 6. 带索引的输出
for i, item in enumerate(my_list, 1):  # 从1开始计数
    print(f"{i}. {item}")
# 输出:
# 1. 1
# 2. 2
# 3. 3
# ...

# 7. 格式化输出列表中的数字
prices = [12.5, 9.99, 23.456, 5.0]
for price in prices:
    print(f"¥{price:>8.2f}")  # 右对齐，宽度8，保留2位小数
# 输出:
# ¥   12.50
# ¥    9.99
# ¥   23.46
# ¥    5.00

num = 123.456789

# 保留2位小数
print(f"{num:.2f}")    # 123.46 (四舍五入)
print(round(num, 2))   # 123.46 (返回浮点数)
print("{:.2f}".format(num))  # 123.46

# 保留整数
print(f"{num:.0f}")    # 123
```

```
print(round(num))      # 123

# 科学计数法
print(f"{num:.2e}")    # 1.23e+02

# 百分比
percent = 0.4567
print(f"{percent:.1%}")  # 45.7%
print(f"{percent:.2%}")  # 45.67%

# 补零对齐
num2 = 7.5
print(f"{num2:08.2f}")  # 00007.50 (总宽度8，不足补0)
# 使用示例
scores = [95.5, 88.3, 92.7, 76.4]
print_list(scores, fmt="{:.1f}")  # 输出: 95.5, 88.3, 92.7, 76.4
```

队列相关语法

```
from collections import deque  # 双端队列，常用作队列

# 创建队列
queue = deque()  # 空队列
queue = deque([1, 2, 3])  # 从列表初始化
queue = deque(maxlen=5)  # 最大长度限制

# 基本操作
queue.append(4)       # 入队: 从右侧添加
queue.appendleft(0)   # 从左侧添加
item = queue.popleft()  # 出队: 从左侧弹出（队列标准操作）
item = queue.pop()    # 从右侧弹出

# 查看
front = queue[0]      # 查看队首（不移除）
rear = queue[-1]      # 查看队尾
length = len(queue)   # 队列长度
is_empty = len(queue) == 0  # 判断是否为空

# 其他操作
queue.clear()         # 清空队列
queue.remove(2)       # 删除指定元素
queue.rotate(2)       # 向右旋转2步
queue.rotate(-1)      # 向左旋转1步
queue.extend([5, 6])  # 从右侧批量添加
queue.extendleft([-1, 0])  # 从左侧批量添加
```

字典

```
# 按键排序
sorted_by_key = dict(sorted(my_dict.items()))  # 默认按键排序
sorted_by_key = dict(sorted(my_dict.items(), key=lambda x: x[0]))
```

```
# 按值排序
sorted_by_value = dict(sorted(my_dict.items(), key=lambda x: x[1]))

# 按值降序排序
sorted_desc = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))

# 复杂排序示例
data = {'Alice': 25, 'Bob': 30, 'Charlie': 20}
# 按年龄排序
sorted_by_age = dict(sorted(data.items(), key=lambda x: x[1]))
# 创建嵌套字典
students = {
    'Alice': {'age': 20, 'score': 95},
    'Bob': {'age': 22, 'score': 88},
    'Charlie': {'age': 21, 'score': 92}
}
```

4.3 递归三部曲：全排列

示例sy132: 全排列I 中等

https://sunnywhy.com/sfbj/4/3/132

```
# 初始化访问标记数组，长度为11（因为n最大为10，索引从1开始使用更方便）
table = [False] * 11
# 读取输入的n，表示要生成1到n的全排列
n = int(input())
# 递归函数，生成全排列
def solve(n, prefix=[]):
    # 如果当前前缀长度已经等于n，说明已经生成了一个完整的排列
    if len(prefix) == n:
        return [prefix]  # 返回当前排列，包装在列表中以便后续拼接
    result = []  # 用于存储所有子排列的结果
    # 遍历1到n的所有数字
    for i in range(1, 1 + n):
        if table[i]:  # 如果数字i已经被使用过，则跳过
            continue
        table[i] = True  # 标记数字i为已使用
        # 递归生成剩余数字的排列，并将当前数字i添加到前缀中
        result += solve(n, prefix + [i])
        table[i] = False  # 回溯：取消标记，以便其他排列可以使用这个数字
    return result  # 返回所有可能的排列
# 调用函数生成全排列
result = solve(n)
# 遍历并输出所有排列
for r in result:
    # 将排列中的数字转换为字符串并用空格连接，然后打印
    print(' '.join(map(str, r)))
```

练习02386: Lake Counting

dfs similar, http://cs101.openjudge.cn/practice/02386

```
#1.dfs
import sys
sys.setrecursionlimit(20000)
def dfs(x,y):
    #标记，避免再次访问
    field[x][y]='.'
    for k in range(8):
        nx,ny=x+dx[k],y+dy[k]
        #范围内且未访问的lake
        if 0<=nx<n and 0<=ny<m\
                and field[nx][ny]=='W':
            #继续搜索
            dfs(nx,ny)
n,m=map(int,input().split())
field=[list(input()) for _ in range(n)]
cnt=0
dx=[-1,-1,-1,0,0,1,1,1]
dy=[-1,0,1,-1,1,-1,0,1]
for i in range(n):
    for j in range(m):
        if field[i][j]=='W':
            dfs(i,j)
            cnt+=1
print(cnt)
```

练习M78.子集

backtracking, https://leetcode.cn/problems/subsets/

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

```
class Solution:
    def solve(self, nums):
        n = len(nums)
        ans, sol = [], []
        def backtrack(i):
            if i == n:
                ans.append(tuple(sol))
                return
            backtrack(i + 1)
            sol.append(nums[i])
            backtrack(i + 1)
            sol.pop()
        backtrack(0)
        return ans
```

M131.分割回文串

dp, backtracking, https://leetcode.cn/problems/palindrome-partitioning/

```
from functools import lru_cache
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n = len(s)
        ans = []
        @lru_cache(None)
        def is_pal(i, j):
            return i >= j or (s[i] == s[j] and is_pal(i + 1, j - 1))

        def dfs(start, path):
            if start == n:
                ans.append(path[:])
                return
            for end in range(start, n):
                if is_pal(start, end):
                    path.append(s[start:end + 1])
                    dfs(end + 1, path)
                    path.pop()
        dfs(0, [])
        return ans
```

晴问9.6.1 学校的班级个数

https://sunnywhy.com/sfbj/9/6/360

```
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))
```

晴问9.6.4 迷宫连通性

https://sunnywhy.com/sfbj/9/6/363

```
def find(x):
    if parent[x] != x:
        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))

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

sets = set(find(x) for x in range(1, n + 1))
if len(sets) == 1:
    print('Yes')
else:
    print('No')
```

week10~11

2 选择不相交区间

选择不相交区间问题大概题意就是：

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。

```
from typing import List
import sys

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # 按照右端点从小到大排序
        # 这是贪心算法的关键：总是选择结束时间最早的区间
        intervals.sort(key=lambda x: x[1])

        res = 0  # 记录最多能选出的不重叠区间数量
        ed = -sys.maxsize  # 初始化结束时间为负无穷

        for v in intervals:
            # 如果当前区间的开始时间 >= 上一个选中区间的结束时间
            if ed <= v[0]:
                res += 1  # 可以选择这个区间
                ed = v[1]  # 更新结束时间
```

```
        # 需要移除的区间 = 总区间数 - 最多可保留的不重叠区间数
        return len(intervals) - res
```

3 区间选点问题

区间选点问题大概题意就是：

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）。和上一题一样。

4 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。

解题步骤：按照区间左端点从小到大排序、**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

```
from typing import List
class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        # 对 clips 按起点升序排序
        clips.sort()
        st, ed = 0, time
        res = 0
        i = 0
        while i < len(clips) and st < ed:
            maxR = 0
            while i < len(clips) and clips[i][0] <= st:
                maxR = max(maxR, clips[i][1])
                i += 1
            if maxR <= st:
                return -1
            # 更新 st 为 maxR，并增加结果计数
            st = maxR
            res += 1
            if maxR >= ed:
                # 已经覆盖到终点
                return res
        # 如果没有成功覆盖到终点
        return -1写这个程序的解释，顺便删掉多余的空行
```

5 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。

主持人调度

另一种解法：

```
from typing import List
import heapq
```

```python
class Solution:
    def minmumNumberOfHost(self, n: int, startEnd: List[List[int]]) -> int:
        # 按左端点从小到大排序
        startEnd.sort(key=lambda x: x[0])

        # 创建小顶堆
        q = []

        for i in range(n):
            if not q or q[0] > startEnd[i][0]:
                heapq.heappush(q, startEnd[i][1])
            else:
                heapq.heappop(q)
                heapq.heappush(q, startEnd[i][1])

        return len(q)
```

6 覆盖连续区间

在给定若干种硬币面值（每种无限供应）的前提下，选出最少数量的硬币，使得它们的组合能够表示出从 1 到 X** 的所有整数值。

```python
import bisect

def solve():
    X, N = map(int, input().split())
    coins = list(map(int, input().split()))
    coins.sort()

    # 如果最小面值 > 1，无法凑出 1
    if coins[0] > 1:
        print(-1)
        return

    reach = 0
    ans = 0

    # 可以重复使用同一个面值多次，所以不移除
    while reach < X:
        target = reach + 1
        # 找 coins 中最大且 <= target
        idx = bisect.bisect_right(coins, target) - 1
        if idx < 0:
            print(-1)
            return
        c = coins[idx]
        reach += c
        ans += 1

    print(ans)
```

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

单调栈

```
def next_greater_element(nums):
    stack = []
    result = [0] * len(nums)

    for i in range(len(nums)):
        # 当栈不为空且当前考察的元素大于栈顶元素时
        while stack and nums[i] > nums[stack[-1]]:
            index = stack.pop()
            result[index] = nums[i]
        # 将当前元素的索引压入栈中
        stack.append(i)

    # 对于栈中剩余的元素，它们没有更大的元素
    while stack:
        index = stack.pop()
        result[index] = -1

    return result

nums = [4, 5, 2, 25]
print(next_greater_element(nums))  # 输出: [5, 25, 25, -1]
```

五、「滑动窗口最大值」三者合一

```
from collections import deque

def maxSlidingWindow(nums, k):
    dq = deque()  # 存下标，保证对应值递减
    res = []
    for right, x in enumerate(nums):
        # step 1: 窗口右扩，保持单调递减
        while dq and nums[dq[-1]] <= x:
            dq.pop()
        dq.append(right)

        # step 2: 移除滑出窗口的左端元素
        if dq[0] <= right - k:
            dq.popleft()

        # step 3: 当窗口形成（长度 >= k）时，记录最大值
        if right >= k - 1:
            res.append(nums[dq[0]])
    return res
```

六、前缀和优化区域统计(垃圾炸弹)

```
def main():
    d = int(input().strip())
    n = int(input().strip())

    # 使用字典存储垃圾点，节省空间（稀疏数据时特别有效）
    moskow = {}
    max_coord = 1024
    min_coord = 0

    for _ in range(n):
        x, y, weight = map(int, input().strip().split())
        if (x, y) not in moskow:
            moskow[(x, y)] = 0
        moskow[(x, y)] += weight

    # 构建二维前缀和数组（只构建 [0..1024] 范围）
    size = max_coord + 1
    prefix = [[0] * (size + 1) for _ in range(size + 1)]  # prefix[0..1024+1][0..1024+1]

    for i in range(1, size + 1):
        for j in range(1, size + 1):
            # 注意：prefix[i][j] 对应坐标 (i-1, j-1)
            val = moskow.get((i - 1, j - 1), 0)
            prefix[i][j] = val + prefix[i - 1][j] + prefix[i][j - 1] - prefix[i - 1][j - 1]

    max_total = 0
    count = 0

    # 遍历所有可能的爆炸中心 (i, j)，范围 [0, 1024]
    for i in range(min_coord, max_coord + 1):
        for j in range(min_coord, max_coord + 1):
            # 计算爆炸影响区域 [x1, x2] × [y1, y2]
            x1 = max(min_coord, i - d)
            y1 = max(min_coord, j - d)
            x2 = min(max_coord, i + d)
            y2 = min(max_coord, j + d)

            # 查询前缀和：注意 prefix 是 1-indexed
            total = prefix[x2 + 1][y2 + 1] \
                    - prefix[x1][y2 + 1] \
                    - prefix[x2 + 1][y1] \
                    + prefix[x1][y1]

            if total > max_total:
                max_total = total
                count = 1
            elif total == max_total:
                count += 1

    print(count, max_total)
```

```
if __name__ == '__main__':
    main()
```

week11-12 dp

完全背包问题

完全背包问题是一种背包问题，其中每种物品可以无限次选择。在最少零钱组合问题中，每种面额的硬币可以无限次使用，因此它可以被视为一个完全背包问题。

代码实现

```
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):
        # 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

# 示例
amount = 11
coins = [1, 2, 5]
print(min_coins_for_change(amount, coins))  # 输出 3 (5 + 5 + 1)
```

2 动态规划的递归写法和递推写法

寻找最大子数组

```
n = int(input())
*a, = map(int, input().split())

dp = [0]*n
dp[0] = a[0]

for i in range(1, n):
    dp[i] = max(dp[i-1]+a[i], a[i])

print(max(dp))
```

最长递增子序列

```
n = int(input())
*b, = map(int, input().split())
dp = [1] * n  # dp[i]表示以b[i]结尾的最长递增子序列长度，初始每个元素自己长度为1

for i in range(1, n):  # 遍历每个元素
    for j in range(i):  # 遍历i之前的所有元素
        if b[j] < b[i]:  # 如果之前的元素比当前元素小，可以构成递增序列
            # 状态转移：以b[i]结尾的LIS长度 = max(当前值, 以b[j]结尾的长度+1)
            dp[i] = max(dp[i], dp[j] + 1)

print(max(dp))  # 取所有dp[i]中的最大值
```

bisect用法

```
import bisect
n = int(input())
*lis, = map(int, input().split())

# dp数组：用于维护当前找到的最小末尾元素
# 初始化为一个大数（1e9表示无穷大）
dp = [1e9] * n

for i in lis:  # 遍历每个元素
    # bisect.bisect_left(dp, i)：
    # 在dp中找到第一个 >= i 的位置
    pos = bisect.bisect_left(dp, i)
    # 用当前元素i替换dp[pos]处的值
    dp[pos] = i

# 找到dp中第一个小于1e8的位置，就是LIS长度
# 注意：这里应该是1e9，但代码写成了1e8，可能是笔误
print(bisect.bisect_left(dp, 1e8))
```

5 背包DP

5.1 0-1背包（每个物品选或者不选）

```
# 动态规划之背包问题（算法图解书中例子实现）

#第一步建立网格(横坐标表示[0,c]整数背包承重):(n+1)*(c+1)
def knapsack(n, c, w, p):
    cell = [[0 for j in range(c+1)]for i in range(n+1)]
    for j in range(c+1):
        #第0行全部赋值为0，物品编号从1开始.为了下面赋值方便
        cell[0][j] = 0
    for i in range(1, n+1):
        for j in range(1, c+1):
            #生成了n*c有效矩阵，以下公式w[i-1],p[i-1]代表从第一个元素w[0],p[0]开始取。
            if j >= w[i-1]:
                cell[i][j] = max(cell[i-1][j], p[i-1] + cell[i-1][j - w[i-1]])
            else:
```

```
                cell[i][j] = cell[i-1][j]
    return cell


goodsnum, bagsize = map(int, input().split())
#goodsnum, bagsize = 3, 4
*value, = map(int, input().split())
*weight, = map(int, input().split())
#value, weight = [1500, 3000, 2000], [1, 4, 3]  # guitar, stereo, laptop

cell = knapsack(goodsnum, bagsize, weight, value)
print(cell[goodsnum][bagsize])
```

优化23421:《算法图解》小偷背包问题

从 **大到小更新**，总是基于"之前未包含当前物品的最优解"来更新新的状态，因此能保证每个物品在每次主循环中只会被计算一次。

```
# 压缩矩阵/滚动数组 方法
N,B = map(int, input().split())
*p, = map(int, input().split())
*w, = map(int, input().split())

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

print(dp[-1])
```

最长回文串问题

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        if n <= 1:
            return s

        # ---------- 第一部分：预处理所有回文子串（DP） ----------
        is_palindrome = [[False] * n for _ in range(n)]

        for right in range(n):
            for left in range(right + 1):
                if s[left] == s[right] and (right - left <= 1 or is_palindrome[left + 1]
[right - 1]):
                    is_palindrome[left][right] = True

        # ---------- 第二部分：扫描所有 (left, right) 求最长 ----------
        max_len = 1
        start = 0
```

```
        for left in range(n):
            for right in range(left, n):
                if is_palindrome[left][right] and (right - left + 1) > max_len:
                    max_len = right - left + 1
                    start = left

        return s[start:start + max_len]
```

5.2 完全背包（每种物品可以选0个-无限个）

```
n, a, b, c = map(int, input().split())
dp = [0]+[float('-inf')]*n

for i in range(1, n+1):
    for j in (a, b, c):
        if i >= j:
            dp[i] = max(dp[i-j] + 1, dp[i])

print(dp[n])
```

5.3 多重背包（每个物品数量有上限）

最简单的思路是将多个同样的物品看成多个不同的物品，从而化为0-1背包。稍作优化：可以改善拆分方式，譬如将m个1拆成x_1,x_2,……,x_t个1，只需要这些x_i中取若干个的和能组合出1至m即可。最高效的拆分方式是尽可能拆成2的幂，也就是所谓“二进制优化”

```
def max_muscle_gain(T, n, trainings):
    # 定义一个很大的负数作为无效值
    INF = -10 ** 9
    # INF表示"不可行"状态
    # 为什么用负数？因为我们要取max，负数会被过滤掉

    # 初始化 dp 数组
    # dp[i][j] 表示：考虑前i个训练组，花费恰好j分钟时的最大增肌量
    # 二维数组：(n+1)行 × (T+1)列
    dp = [[INF] * (T + 1) for _ in range(n + 1)]

    # 设置初始条件
    for i in range(n + 1):
        dp[i][0] = 0  # 时间为0时，增肌量为0
        # 注意：这是"恰好用0分钟"，当然是可行的，增肌量为0

    # 动态规划转移
    for i in range(1, n + 1):  # 遍历每个训练组
        ti, wi = trainings[i - 1]  # 第i个训练组的时间和增肌量

        for j in range(T + 1):  # 遍历所有可能的时间
            # 情况1：不选择第i个训练组
            dp[i][j] = dp[i - 1][j]
```

```
            # 情况2：如果时间足够，选择第i个训练组
            if j >= ti:
                # dp[i-1][j-ti] + wi 表示：
                # 在前i-1个训练组中花费j-ti分钟的最大增肌量 + 当前训练组的增肌量
                # 必须用dp[i-1]而不是dp[i]，确保每个训练组只用一次
                dp[i][j] = max(dp[i][j], dp[i - 1][j - ti] + wi)

    # 输出结果
    if dp[n][T] < 0:
        return -1  # 如果结果为负数，说明不可行
    else:
        return dp[n][T]  # 考虑所有n个训练组，恰好用T分钟的最大增肌量
```

6 最长公共子串Longest common substring

```
n = int(input())
li = list(map(int, input().split()))

# 初始化：单个元素，长度为1，上升和下降都算1
dpup = 1   # 当前结尾为“上升”的最长摆动序列长度
dpdo = 1   # 当前结尾为“下降”的最长摆动序列长度

for i in range(1, n):
    if li[i] > li[i-1]:
        dpup = max(dpdo + 1, dpup)    # 上升：接在下降后面
        # dpdo 不变
    elif li[i] < li[i-1]:
        dpdo = max(dpup + 1, dpdo)    # 下降：接在上升后面
        # dpup 不变
    # 如果相等，两者都不变

print(max(dpup, dpdo))
```

往年题目

```
n = int(input())
num = 0
stus = []
for _ in range(n):
    num += 1
    a, b, c = map(int, input().split())
    d = a + b + c
    stus.append((d, a, num))
stus.sort(key=lambda x: (-x[0], -x[1], x[2]))
for i in range(5):
    print(stus[i][2], stus[i][0])
```

```
n = int(input())
nums = list(map(int, input().split()))
up = 1
down = 1
for i in range(1, len(nums)):
    if nums[i] > nums[i-1]:
        up = down + 1
    elif nums[i] < nums[i-1]:
        down = up + 1
print(max(up, down))
```