计概B-CS

事先声明：内容很基础，针对初次接触的来复习

第 1 章

矩阵：二维状态空间与图建模

矩阵不是二维数组，而是一个“规则图（graph）”或一个“二维 DP 状态空间”

因此，矩阵实际就是BFS、DFS、DP、单调栈、Kadane等等的根源

1.1 矩阵问题的三种“等价视角”

给你一个 `n × m` 的矩阵：

```
grid[i][j]
```

你可以用两种方式看它：

视角 1：二维 DP 表/状态空间

```
dp[i][j] = 到达 (i, j) 的最优解
```

视角 2：规则图/物理空间

- 每个格子是一个节点
- 上下左右是边

1.2 矩阵 DFS：连通性与区域问题

之所以在这里提dfs，是因为矩阵地图是最能体现出dfs一条路走到头的思想的

例题 1：岛屿数量（经典）

问题
给定 `0/1` 矩阵，`1` 表示陆地，求岛屿个数。

思路拆解

- 每遇到一个没访问过的 `1`
- 用 DFS 把整个连通块“染掉”/visited
- 每启动一次 DFS/覆盖一个连通块，岛屿数 +1

代码

```
def numIslands(grid):
    n, m = len(grid), len(grid[0])
    visited = [[False]*m for _ in range(n)]

    def dfs(x, y):
        if x < 0 or x >= n or y < 0 or y >= m:
            return
        if grid[x][y] == '0' or visited[x][y]:
            return

        visited[x][y] = True
        for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(x+dx, y+dy)

    ans = 0
    for i in range(n):
        for j in range(m):
            if grid[i][j] == '1' and not visited[i][j]:
                dfs(i, j)
                ans += 1

    return ans
```

关键理解点

- visited 是 **状态表**
- dfs 不是“函数”，而是 **在图上走一遍**
- 上下左右 = 图的邻接关系

1.4 矩阵 BFS：最短路径与扩散

同样，在这里提到bfs是最能体现bfs如同液体般一层一层向外铺开的感觉的。但bfs的变种很多，所以在文末还会对bfs做一次总结

例题 2：矩阵最短路径（无权）

从左上角到右下角，`0` 可走，`1` 障碍。

思路拆解

- 每一层 BFS = 多走一步
- dist 数组记录最短距离
- queue 中存的是状态 `(x, y)`

代码

```
from collections import deque

def shortestPath(grid):
    n, m = len(grid), len(grid[0])
    dist = [[-1]*m for _ in range(n)]
    q = deque()

    q.append((0, 0))
    dist[0][0] = 0

    while q:
        x, y = q.popleft()
        for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
            nx, ny = x+dx, y+dy
            if 0 <= nx < n and 0 <= ny < m:
                if grid[nx][ny] == 0 and dist[nx][ny] == -1:
                    dist[nx][ny] = dist[x][y] + 1
                    q.append((nx, ny))

    return dist[n-1][m-1]
```

BFS 与 DFS 的本质区别

	DFS	BFS
目标	连通性	最短路径
数据结构	栈 / 递归	队列
状态顺序	深	层

1.5 多源 BFS：同时扩散的思想

核心思想

> 多个起点一起进队列

例题 3：01 矩阵（最近的 0）

- `0` 作为源点
- `1` 需要找最近的 `0`

思路拆解

- 所有 `0` 同时入队
- BFS 扩散
- 第一次到达的距离就是最短

代码

```
def updateMatrix(mat):
    n, m = len(mat), len(mat[0])
    from collections import deque
    q = deque()
    dist = [[-1]*m for _ in range(n)]

    for i in range(n):
        for j in range(m):
            if mat[i][j] == 0:
                dist[i][j] = 0
                q.append((i, j))

    while q:
        x, y = q.popleft()
        for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
            nx, ny = x+dx, y+dy
            if 0 <= nx < n and 0 <= ny < m and dist[nx][ny] == -1:
                dist[nx][ny] = dist[x][y] + 1
                q.append((nx, ny))

    return dist
```

1.6 矩阵前缀和：二维区间的 O(1) 查询

核心思想

> 预处理一次，查询无数次

定义

```
pre[i][j] = 左上 (0,0) 到 (i,j) 的和
```

查询任意子矩阵

```
sum = pre[x2][y2] - pre[x1-1][y2]-pre[x2][y1-1]+pre[x1-1][y1-1]
```

1.8 Kadane 算法（一维最大子数组）

核心思想

> 当前和为负，就重新开始

代码

```python
def maxSubArray(nums):
    cur = ans = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)
        ans = max(ans, cur)
    return ans
```

1.9 二维 Kadane：最大子矩阵和（v2.0 重点）

核心思想

> 把二维问题“压成一维”

思路拆解

1. 固定上边界 top
2. 向下枚举 bottom
3. 每一列累加 → 一维数组
4. 对该数组跑 Kadane

代码（理解级，不要求手写）

```
def maxSumSubmatrix(matrix):
    n, m = len(matrix), len(matrix[0])
    ans = float('-inf')

    for top in range(n):
        colSum = [0]*m
        for bottom in range(top, n):
            for j in range(m):
                colSum[j] += matrix[bottom][j]

            cur = colSum[0]
            best = colSum[0]
            for x in colSum[1:]:
                cur = max(x, cur+x)
                best = max(best, cur)

            ans = max(ans, best)

    return ans
```

第 2 章

排序：贪心成立的结构前提

2.1 本章的核心思想

> **贪心思想的成立是建立“排序后，局部最优 = 全局最优”的结构之上的**

也就是说，选定合适的排序方法是贪心的基础

接下来会以以**区间问题**为例来展开

2.2 排序在算法中的真实含义

排序是为了“建立决策顺序”

2.2.1 sort + lambda 的含义

```
intervals.sort(key=lambda x: x[1])
```

这句话等价于：

> **“右端点最小，就一定是当前最优决策/就可以对应当前最优解”**

2.3 区间问题的统一抽象

区间问题本质都在回答三类问题之一：

1. **选多少个区间**
2. **能否覆盖某个范围**
3. **区间之间如何冲突**

2.4 区间调度问题（贪心的起点）

例题 1：最多不重叠区间数

给定若干区间，选出最多个互不重叠的区间。

思路拆解

选**结束最早**的区间——给后面留下最大自由度 ——也就是按右端点排序

算法步骤

1. 按右端点排序
2. 每次选能接上的第一个区间

代码

```
def intervalSchedule(intervals):
    intervals.sort(key=lambda x: x[1])
    end = float('-inf')
    count = 0

    for l, r in intervals:
        if l >= end:
            count += 1
            end = r

    return count
```

2.5 区间覆盖问题

例题 2：最少区间覆盖 [L, R]

思路拆解

与前一部分要求尽可能多的区间——即给后来的区间留尽可能多的空间——不同，此处要求：单个区间的覆盖面尽可能广——按左端点排序，在当前能覆盖的位置中选右端点最远的

代码

```
def minCover(intervals, L, R):
    intervals.sort()
    i = 0
    n = len(intervals)
    cur = L
    ans = 0

    while cur < R:
        far = cur
        while i < n and intervals[i][0] <= cur:
            far = max(far, intervals[i][1])
            i += 1

        if far == cur:
            return -1  # 无法覆盖

        ans += 1
        cur = far

    return ans
```

2.6 区间合并（非贪心，但依赖排序）

例题 3：合并重叠区间

思路

- 按左端点排序
- 能合就合，不能合就开新区间

代码

```python
def merge(intervals):
    intervals.sort()
    res = []

    for l, r in intervals:
        if not res or res[-1][1] < l:
            res.append([l, r])
        else:
            res[-1][1] = max(res[-1][1], r)

    return res
```

2.7 扫描线（Sweep Line）思想

核心思想：把“区间”转成“事件”

例题 4：最大区间重叠数

思路拆解——和为0来构造判断某一个区间（事件）是否结束

1. 每个区间 `[l, r)`
   → `l`：+1
   → `r`：-1
2. 按位置排序
3. 扫描累加

代码

```python
def maxOverlap(intervals):
    events = []
    for l, r in intervals:
        events.append((l, 1))
        events.append((r, -1))

    events.sort()
    cur = ans = 0

    for _, delta in events:
        cur += delta
        ans = max(ans, cur)

    return ans
```

扫描线的本质

> **区间问题 → 前缀和问题**

2.8 资源调度模型

例题 5：最少主持人问题

问题描述

每个区间是一场活动，
需要多少个主持人才能不冲突？

思路拆解

1. 按开始时间排序
2. 用最小堆维护当前结束时间
3. 能复用就复用，否则新增

代码

```
import heapq

def minHosts(intervals):
    intervals.sort()
    heap = []

    for l, r in intervals:
        if heap and heap[0] <= l:
            heapq.heappop(heap)
        heapq.heappush(heap, r)

    return len(heap)
```

非常重要的理解

> **堆不是为了排序，而是为了“快速找到最早结束的资源”**

2.9 贪心为什么有时会失败?

贪心不是万能的，很多时候用贪心无法解决

- 局部最优 ≠ 全局最优
- 决策之间有“后效性”

判断标准（非常重要）

如果一个问题：

- 当前选择会**永久影响未来可行性**
- 且影响无法被“排序顺序”完全刻画

👉 **贪心很可能不成立，需要 DP**（也有一个比较邪修的办法——多写几个不同角度的贪心，取结果最优解）

2.10 排序 + 贪心 + DP 的边界

特征	方法
一次决定，不能回头	贪心
决定之间相互影响	DP
需要全局最短 / 最优	DP / 搜索

第 3 章

栈与单调结构

3.1 stack的核心

暂时还不能确定答案的元素集合

一旦条件满足，它们的答案就“被结算”。

3.2 单调结构的核心思想

首先，我们需要知道stack通常对应下标，那么单调栈就是为了准备一个随侍侯选的有序队列，方便调度

3.3 最基础模型：下一个更大元素

例题 1：下一个更大元素（Next Greater Element）

给定数组 `nums`，
对每个元素，找右侧第一个比它大的数。

思路拆解

1. 从左到右扫描
2. 栈中存 **还没找到答案的元素下标**
3. 一旦当前元素更大
   → 栈顶元素的答案确定

单调性

栈中元素值单调递减

代码

```
def nextGreater(nums):
    n = len(nums)
    ans = [-1] * n
    stack = []  # 存下标

    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            idx = stack.pop()
            ans[idx] = x
        stack.append(i)

    return ans
```

注：

- 元素**进栈一次、出栈一次**
- 时间复杂度 (O(n))，不是 (O(n^2))

3.4 左右最近更大 / 更小元素

这是单调栈的“总模板”

目标

对每个元素 `i`，找：

- 左侧最近 **更小**
- 右侧最近 **更小**

代码模板（右侧最近更小）

```
def rightSmaller(nums):
    n = len(nums)
    right = [n] * n
    stack = []

    for i in range(n):
        while stack and nums[stack[-1]] > nums[i]:
            right[stack.pop()] = i
        stack.append(i)

    return right
```

左侧同理

之于单调性的严格与否，视题目而定

3.5 区间贡献思想

核心思想

每个元素作为“最小值 / 最大值”，能覆盖多大的区间?

区间贡献公式

如果元素位次 `i`：左边界：`L`；右边界：`R`

那么它作为最小值的贡献区间数（有多少个区间的最小子数是它）是：

```
(i - L) * (R - i)
```

3.6 例题 2：子数组最小值之和（经典难题）

问题描述

给定数组 `arr`，
求 **所有子数组的最小值之和。**

正确思路

不枚举子数组，**枚举“最小值是谁”。**

思路拆解

1. 用单调栈找：
    - 左侧最近严格小于arr[i]
    - 右侧最近小或等于arr[i]（防止在同一个贡献区间内最小值出现两次导致重复计算，故直接将这个区间分给两个最小值）
2. 计算每个元素的贡献

代码（理解级）

```
def sumSubarrayMins(arr):
    n = len(arr)
    left = [-1]*n
    right = [n]*n
    stack = []

    # 左边界
    for i in range(n):
        while stack and arr[stack[-1]] > arr[i]:
            stack.pop()
        left[i] = stack[-1] if stack else -1
        stack.append(i)

    stack.clear()
    # 右边界
    for i in range(n-1, -1, -1):
        while stack and arr[stack[-1]] >= arr[i]:
            stack.pop()
        right[i] = stack[-1] if stack else n
        stack.append(i)

    ans = 0
    for i in range(n):
        ans += arr[i] * (i - left[i]) * (right[i] - i)
```

```
    return ans
```

非常重要的一点

> **左严格、右非严格（或反过来）**
> **是为了避免重复计算**

3.7 接雨水问题（单调栈版）

例题 3：接雨水

思路直觉

- 凹槽由 **左右更高的柱子** 决定
- 中间低的柱子延迟结算

思路拆解

1. 维护一个单调递减栈
2. 一旦遇到更高的柱子则形成一个“凹槽”

代码（核心版）

```
def trap(height):
    stack = []
    ans = 0
#逐层计算
    for i, h in enumerate(height):
        while stack and height[stack[-1]] < h:
            bottom = stack.pop() #找凹槽
            if not stack: #栈是空的（不再有凹槽）
                break
            left = stack[-1] #右侧高度对应的左侧高度
            w = i - left - 1 #水宽
            bounded = min(height[left], h) - height[bottom] #水深
            ans += w * bounded
        stack.append(i)

    return ans
```

3.8 柱状图最大矩形

例题 4：Largest Rectangle in Histogram

思路本质

首先要明白，最终的矩形高度必定是heights中的元素

那么我们就可以来反推——当矩形的高一定的时候它的宽是多少？

这其实就是把找到每个元素作为最小值时分别对应的最长连续子数组

算法步骤

1. 对每根柱子：
    - 找左边第一个更矮
    - 找右边第一个更矮
2. 面积 = 高度 × 宽度

代码（模板级）

```
def largestRectangleArea(heights):
    heights.append(0)
    stack = []
    ans = 0

    for i, h in enumerate(heights):
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            left = stack[-1] if stack else -1
            width = i - left - 1
            ans = max(ans, height * width)
        stack.append(i)

    return ans
```

3.9 单调队列：滑动窗口最大值

与单调栈的区别

栈	队列
区间扩展	滑动窗口
单向结算	动态维护

例题 5：滑动窗口最大值

思路

- 队列中时刻保持 **从大到小**
- 窗口左端移出时同步维护

代码

```
from collections import deque

def maxSlidingWindow(nums, k):
    q = deque()
    res = []

    for i, x in enumerate(nums):
        while q and nums[q[-1]] <= x: #引入新数据，同时保持单调性
            q.pop()
        q.append(i)

        if q[0] <= i - k: #检查原有的最大值是不是窗口左端点，还在不在
            q.popleft()

        if i >= k - 1: #检查这个窗口是否合法
            res.append(nums[q[0]])

    return res
```

其实滑动窗口的思想都很一致，核心就是取定k，然后确保右进一左出一

3.10 单调结构的实质

由以上内容我们不难看出，相较于单纯的栈——一个存储留待处理的元素的结构——单调栈/单调结构更多的作用是保障/寻找一个对应区间内的最大/小值

第 4 章

递归、DFS 与回溯：状态空间树（v2.0）

4.1 核心

递归 / DFS / 回溯，本质上都是：在一棵「状态空间树」上做搜索

下文中的：全排列、子集、N 皇后、数独、组合枚举**全部都是这棵树的不同形态。**

因此，将**DFS（Depth-First Search，深度优先搜索）**与**递归、回溯**放在一起来理解

4.2 什么是“状态空间树”

把问题抽象成一棵树

- 每一个节点：**一个状态**
- 从父节点到子节点：**一次决策**
- 根节点：初始状态
- 叶节点：完整解 / 失败解

举例：子集问题

数组 `[1,2,3]`

```
            []
          /    \
        [1]     []
      /    \   /   \
  [1,2]  [1] [2]   []
```

DFS 的过程，就是把这棵树“走一遍”

4.3 递归的真正含义

递归 = 把“大问题”拆成“结构完全相同的小问题”

递归三要素

1. **状态参数**：描述当前处境
2. **递归出口**：什么时候停止
3. **递归转移**：如何走向下一个状态

4.4 DFS 与回溯的关系

单纯的dfs有种不管别的路径死活的美感——也就是只要我这条路挖得够深，可以到达目标就好

而回溯可以避免这一问题

DFS 只是“走”，回溯还要“回来”。

当你的状态包含：

- 可变对象（list / set）
- 全局路径（path）

就必须回溯。

4.5 递归+回溯的模板

```
def backtrack(状态):
    if 满足结束条件:
        记录当前一种答案
        return #跳出当前结果

    for 每一种选择:
        做选择
        backtrack(新状态)
        撤销选择
```

所有回溯题，都在填这个模板

4.6 例题 1：全排列（Permutation）

问题描述

给定 arr，输出所有排列。

状态设计

- path：当前排列
- used：哪些数已经用过

代码

```
def permute(nums):
    res = []
    used = [False] * len(nums)
    path = []

    def dfs():
        if len(path) == len(nums):
            res.append(path[:]) #针对可变对象
            return

        for i in range(len(nums)):
            if used[i]:
                continue
            used[i] = True
            path.append(nums[i])
            dfs()
```

```
            path.pop()
            used[i] = False

    dfs()
    return res
```

必须理解的点

- `path[:]` 是 **拷贝**
- `pop()` 是 **回溯的灵魂**
- `dfs()` 的套用是**递归的体现**
- used 是 **状态的标定**

4.7 例题 2：子集

问题描述

给定 `[1,2,3]`，输出所有子集。

与排列的本质区别

	排列	子集
顺序重要	是	否
状态参数	used	start 下标

代码

```
def subsets(nums):
    res = []
    path = []

    def dfs(start):
        res.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            dfs(i + 1)
            path.pop()

    dfs(0)
    return res
```

注：

start 参数 = 剪枝 + 去重

4.8 例题 3：组合总和（回溯 + 剪枝）

问题描述

给定候选数组，可重复使用，求和为 target。

思路——选后撤回（回溯的核心）

- 排序
- 剪枝（当前和 > target）

代码

```
def combinationSum(candidates, target):
    res = []
    path = []
    candidates.sort()

    def dfs(start, remain):
        if remain == 0: #找到合法组合
            res.append(path[:])
            return
        for i in range(start, len(candidates)): #枚举选哪一个
            if candidates[i] > remain:
                break
            path.append(candidates[i])
            dfs(i, remain - candidates[i]) #选定
            path.pop() #撤回

    dfs(0, target)
    return res
```

此处其实有类似于完全背包问题的做法，将选后撤回改成选不选start+1的问题，即：

```
def combinationSum(candidates, target):
    res = []
    path = []
    candidates.sort()

    def dfs(start, remain):
        if remain == 0: #找到合法组合
            res.append(path[:])
            return
```

```
        if start == len(candidates) or remain<candidates[start]: #判断这条路有没有走到头
            return
        dfs(start+1,remain) #不选start
        path.append(candidates[start]) #选start
        dfs(start, remain - candidates[start])
        path.pop() #撤回——无论是那种做法，都必须有pop来保证每一步的dfs可以回去

    dfs(0, target)
    return res
```

4.9 N 皇后问题

基本量

- row：当前行
- cols：列是否被占
- diag1：主对角线
- diag2：副对角线

核心思想

从r推出r+1的情况，不断扩充棋盘直至n

实质

有要求的列数字的全排列

代码（简化理解版）

```
def solveNQueens(n):
    ans = []
    cols = set()
    diag1 = set()
    diag2 = set()
    path = []

    def dfs(r):
        if r == n: #达到n维要求
            ans.append(path[:])
            return

        for c in range(n):
            if c in cols or r-c in diag1 or r+c in diag2: #排除同列、同对角线的情况
                continue
            cols.add(c)
```

```
            diag1.add(r-c)
            diag2.add(r+c)
            path.append(c)
            dfs(r + 1)
            #撤回
            path.pop()
            cols.remove(c)
            diag1.remove(r-c)
            diag2.remove(r+c)

    dfs(0)
    return ans
```

4.10 剪枝策略总结

在面对dfs+backtrack+recursion的情况中，显然会因为深度深以及状况过多的情况导致爆炸，这种时候就需要减去不必情况

常见剪枝类型

1. **顺序剪枝**——start 下标
2. **可行性剪枝**——超出 target
3. **对称剪枝**——重复排列
4. **约束剪枝**——皇后冲突

4.11 DFS vs 记忆化 DFS

一个关键判断标准

如果dfs中存在某个状态：会被反复访问且返回值只依赖状态本身

👉 **可以用记忆化 DFS**

示例：斐波那契

```
from functools import lru_cache

@lru_cache(None) #None是无限制的缓存，实际情况不一定需要None，适当填写maxsize即可
                 #注：lru_cache一定要写在所需的记忆化函数的上一行
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
print(fib(2048)) #在这个过程中，每一个fib(k)仅需计算一次，此后可直接调用
```

第 5 章

动态规划（Dynamic Programming）：系统性计算状态

5.1 本质结论

其实在这里可以简单的将dp理解为需要记忆的递归

也就是说，如果不考虑各种成本问题的话，其实dp和递归在很大程度上是存在相似性与相互转化的能力的。两种算法的本质要求其实都是找到清晰明了的不同状态之间的转移关系——状态转移方程

5.2 什么时候“应该想到 DP”

一个问题，如果同时满足：**有重叠子问题**、**一个状态的答案只依赖于更“简单”的状态**、**可以人为规定计算顺序**

那么—— **DP 是第一选择**

5.3 DP 的建模方法

在递归问题中，处理的关键是搞明白不同情况间的关系，但我们所做的递归的题目中状态常常是明确的。而在dp题目中，可能会出现状态不明确的问题，所以会有如下的步骤：

第一步：状态定义

> **dp[i] 表示什么?**

一个好的状态定义要满足：

- 不多——eg：有些dp既可以用二维数组表达也可以用一维数组，此时显然一维更优
- 不少——能够充分表达出不同状态的信息
- 只依赖“过去”

第二步：状态转移

常常需要类似于数列递推式的表达式（也可能会添加复杂的限制与判断条件）

第三步：初始化

规定好各种0状态/基本量以及限制不合法的状态

第四步：遍历顺序

保证数列递推是一步步进行的

5.4 一维 DP

例题 1：爬楼梯

问题

一次爬 1 或 2 步，爬到第 n 阶有几种方法?

状态定义

```
dp[i] = 爬到第 i 阶的方法数
```

转移方程

```
dp[i] = dp[i-1] + dp[i-2]
```

初始化

```
dp[0] = 1
dp[1] = 1
```

代码

```python
def climbStairs(n):
    dp=[0]*(n+1)
    dp[0], dp[1] = 1, 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

显然，这里的dp其实并不是随时都需要大量的已计算过的数据，所以并需要完全存储已算过的状态——这事就可以引入滚动数组的方法：复用存储空间

```python
def climbStairs(n):
    a, b = 1, 1
    for _ in range(2, n+1):
        a, b = b,a + b
    return b
```

这种方法可以有效降低空间复杂度，接下来的01背包实际上就是把一个二维数组简化之后的

5.5 背包 DP（DP 的第一座大山）

5.5.1 01 背包（最核心模型）

问题描述

每个物品只能选一次，容量为 W，最大价值?

状态定义

```
dp[w] = 容量为 w 时的最大价值
```

状态转移

```
dp[w] = max(dp[w], dp[w-weight[i]] + value[i])
```

遍历顺序（非常重要）

> **w 必须从大到小**

代码

```python
def knapsack(weights, values, W):
    dp = [0]*(W+1)
    for wgt, val in zip(weights, values):
        for w in range(W, wgt-1, -1):
            dp[w] = max(dp[w], dp[w-wgt] + val)
    return dp[W]
```

一句话本质

> **倒序遍历 = 防止一个物品被用多次**

5.5.2 完全背包

唯一变化

> w 从小到大遍历

代码对比

```
for w in range(wgt, W+1):
    dp[w] = max(dp[w], dp[w-wgt] + val)
```

本质差异总结

背包类型	w 遍历	效果
01 背包	从大到小	防止重复使用同一个物品
完全背包	从小到大	实现同一个物品的重复利用

5.5.3 多重背包

问题

每个物品有数量限制。

核心思想——实际上就是多个01背包

一：可以简单地把每一个物品当成一个独立的个体，然后就又变成了01背包

二：可以用二进制的拆分方法，将每个物品的数量拆分成1+2+4+8+...+余项

```
def manybag():
    #s为每个物品的个数
    dp=[0]*(W+1)
    for i in range(1,n+1):
        k=1
        while s[i]>0:
            cnt=min(k,s[i])
            for j in range(V,cnt*cost[i]-1,-1):
                dp[j]=max(dp[j],cnt*price[i]+dp[j-cnt*cost[i]])
            s[i]-=cnt
            k*=2
    return dp[-1]
```

意义

> 多重背包 → 若干个 01 背包

5.5.4 二维费用背包

问题

与01背包相比，实际上就是在weight以外再多了一项限制条件（eg：cost） 所以其实再套一层逆向循环就好了

状态定义

```
dp[w][c] = 在重量 w、费用 c 下的最大价值
```

代码

```python
def knapsack(weights, costs, values, W,C):
    dp = [[0]*(W+1) for _ in range(C+1)]
    for wgt, cos, val in zip(weights, costs, values):
        for w in range(W, wgt-1, -1):
            for c in range(C, cos-1,-1):
                dp[w][c] = max(dp[w][c], dp[w-wgt][c-cos] + val)
    return dp[W][C]
```

本质

> 状态多了一维约束

5.5.5 恰好装满 vs 不必装满（极易考）

区别在初始化

- 不必装满：`dp[0] = 0`——用不到就是0
- 恰好装满：
    - `dp[0] = 0`
    - 其他初始化为 `-inf`——必须走可以用完的路

5.6 整数划分问题——每个数字视为一个物品

例题 2：整数 n 划分为若干正整数（不考虑顺序）

建模方式

> 完全背包

状态定义

```
dp[i] = 和为 i 的划分方案数
```

转移

```
dp[i] += dp[i - k] #没加入一个分割基数k
```

代码

```
def divide1(n):
    dp=[1]+[0]*n #0划分只有一种
    for i in range(1,n+1):
        for j in range(i,n+1): #正向遍历每个容量（每个n）
            dp[j]+=dp[j-i]
    return dp[n]
```

本质理解

> “不考虑顺序”=固定某一顺序=物品由小到大放入

若考虑顺序：

代码

```
def divide2(n):
    dp=[1]+[0]*n
    for i in range(1,n+1):
        for j in range(1,i+1): #比i小的数字的排放顺序在这一步进行改变
            dp[i]+=dp[i-j]
    return dp[n]
```

若分成的数字要不同，但不考虑顺序

建模方式

> 01背包

代码

```
def divide3(n):
    dp=[1]+[0]*n
    for i in range(1,n+1):
        for j in range(n,i-1,-1):
            dp[j]+=dp[j-i]
    return dp[n]
```

若分成k个正整数，不考虑顺序

代码

```
def divide4(n,k):
    dp=[[0]*(k+1) for _ in range(n+1)]
    for i in range(n+1):
        dp[i][1]=1
    for i in range(1,n+1):
        for j in range(2,k+1):
            if i>=j:
                dp[i][j]=dp[i-1][j-1]+dp[i-j][j]
    return dp[n][k]
```

5.7 序列 DP

例题 3：最长递增子序列（LIS）

5.7.1 O(n²) DP 版

状态定义

```
dp[i] = 以 i 结尾的 LIS 长度
```

转移

```
dp[i] = max(dp[i], dp[j] + 1), j < i 且 nums[j] < nums[i]
```

代码

```python
def LIS(nums):
    n = len(nums)
    dp = [1]*n
    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

5.7.2 O(n log n) + bisect

核心思想

dp[k] = 长度为 k 的 LIS 的最小结尾值(其实应该放到stack那里的)

代码

```python
import bisect

def LIS(nums):
    dp = []
    for x in nums:
        pos = bisect.bisect_left(dp, x)
        if pos == len(dp):
            dp.append(x)
        else:
            dp[pos] = x
    return len(dp)
```

---

dp 数组本身不是答案序列

5.8 区间 DP

例题 4：戳气球

状态定义

```
dp[l][r] = 戳破 (l, r) 区间的最大得分
```

转移

```
dp[l][r] = max(dp[l][k] + dp[k][r] + nums[l]*nums[k]*nums[r])
```

遍历顺序

> 区间长度从小到大

5.9 状态压缩 DP

例题 5：旅行商问题（TSP）

状态定义

```
dp[visited][i] = 访问 mask，最后在 i 的最短路
```

代码

```
n=int(input())
M=[list(map(int,input().split())) for _ in range(n)]
max_=float('inf')
dp=[[max_]*n for _ in range(1<<n)];dp[1][0]=0
for visited in range(1,1<<n):
    for i in range(n):
        if dp[visited][i]==max_:
            continue
        for j in range(n):
            if visited & (1<<j):
                continue
            newvisited=visited|1<<j
            newcost=dp[visited][i]+M[i][j]
            if newcost<dp[newvisited][j]:
                dp[newvisited][j]=newcost
ans=max_
for i in range(n):
```

```
    if dp[(1<<n)-1][i]!=max_:
        ans=min(ans,dp[(1<<n)-1][i]+M[i][0])
print(ans)
```

第 6 章

搜索算法：BFS / Dijkstra / 状态最短路（v2.0）

6.1 总纲

搜索算法，本质上是在“状态空间图”上求最优值

6.2 状态空间图的统一建模

```
状态 = (位置, 附加状态)
边 = 一次合法操作
代价 = 操作的花费
目标 = 最小总代价
```

6.3 BFS：无权图的最短路

BFS 的状态含义

```
queue 中的元素 = 当前这一“层”的状态
```

BFS 标准模板

```
from collections import deque

def bfs(start):
    q = deque([start])
    dist = {start: 0}

    while q:
        cur = q.popleft()
        for nxt in neighbors(cur):
            if nxt not in dist:
                dist[nxt] = dist[cur] + 1
                q.append(nxt)
```

visited vs dist（极重要）

情况	用什么
只关心是否访问	visited
关心最短距离	dist

> **dist 的存在，本身就隐含了 visited**

6.4 多源 BFS（再次强化）

核心思想

> **多个起点 = 同时入队**

典型特征

- 初始队列不止一个状态
- 距离从 0 同时向外扩散

6.5 双向 BFS

- 起点明确
- 终点明确
- 状态空间巨大

核心思想

> **从起点和终点同时扩展，**
> **在中间相遇**

时间复杂度直觉

- 单向 BFS：(O(b^d))
- 双向 BFS：(O(b^{d/2}))

模板

```
front = {start}
back = {end}
visited = {start, end}
step = 0

while front and back:
    if len(front) > len(back):
        front, back = back, front

    next_front = set()
    for cur in front:
        if cur in back:
            return step
        for nxt in neighbors(cur):
            if nxt not in visited:
                visited.add(nxt)
                next_front.add(nxt)

    front = next_front
    step += 1
```

6.6 Dijkstra：正权最短路

使用前提（一定要检查）

> 所有边权 ≥ 0

核心思想

每次确定“当前距离最小”的状态，且这个距离不可能再被更新

Dijkstra 模板（极其重要）

```
import heapq

def dijkstra(start):
    heap = [(0, start)]
    dist = {start: 0}

    while heap:
        d, cur = heapq.heappop(heap)
        if d > dist[cur]:
            continue  # 懒删除

        for nxt, w in graph[cur]:
```

```
            nd = d + w
            if nxt not in dist or nd < dist[nxt]:
                dist[nxt] = nd
                heapq.heappush(heap, (nd, nxt))
```

懒删除

> **堆中允许"过期状态"，**
> **出堆时再判断**

6.7 BFS vs Dijkstra 的本质对比

特征	BFS	Dijkstra
边权	相等	非负
数据结构	deque	heap
扩展顺序	按层	按最小距离

6.8 状态最短路（v2.0 重点）

什么是"状态最短路"?

> **节点要求外，还有附加状态要求**

例子

- 有钥匙 / 没钥匙
- 剩余能量
- 已用次数

状态定义示例

```
(x, y, key_mask)
```

算法选择

情况	方法
状态 + 无权	BFS

情况	方法
状态 + 有权	Dijkstra

---

visited 的关键变化

```
visited[x][y][mask]
```

第 7 章

二分答案

7.1 二分答案（Binary Search on Answer）

不是在数组里找数而是在“答案空间”里找最优值

典型题型特征

- 最大值中的最小；最小值中的最大；是否存在一个 X，使得条件成立

核心前提

> **可行性函数是单调的**

```
X 可行 → X-1 也可行
X 不可行 → X+1 也不可行
```

经典模型：Aggressive Cows

问题简述

- 给定若干位置
- 放 k 头牛
- 最大化牛之间的最小距离

思路拆解

1. 答案是“最小距离”
2. 对距离 d：
    - 能否放下 k 头牛？
3. 可行性是单调的 → 二分

代码

```
n,c=map(int,input().split())
positions=sorted([int(input()) for _ in range(n)])
def accesible(cows,step,positions,n):
    count=1
    last=positions[0]
    for i in range(1,n):
        if count>=cows:
            break
        if positions[i]-last>=step:
            count+=1
            last=positions[i]
    return int(count>=cows)
left=1;right=positions[-1]-positions[0]
while left<=right:
    mid=left+(right-left)//2
    if accesible(c,mid,positions,n)==1:
        result=mid
        left=mid+1
    else:
        right=mid-1
print(result)
```

##

PS：目前只有这些了 希望能尽可能覆盖到了