算法分析题：

3-1 最长单调递增子序列。

设计一个 $O(n^2)$时间的算法，找出由 n 个数组成的序列的最长单调递增子序列。

答：**思路：**设数组为 nums，定义 dp[i]表示以 nums[i]结尾的最长单调递增子序列的长度。初始时，每个元素自身构成长度为 1 的子序列，因此 dp[i]=1（i=0,1, ...,n-1）。

对于每个 i，遍历 j = 0 到 i-1：若 nums[i] > nums[j]，说明 nums[i] 可接在 nums[j] 之后形成更长的递增子序列，此时 dp[i] = max(dp[i], dp[j] + 1)。

流程：①初始化 dp 数组，所有元素为 1。

②外层循环遍历数组元素（i 从 1 到 n-1）。

③内层循环遍历 j 从 0 到 i-1，若 nums[i] > nums[j]，更新 dp[i]。

④遍历结束后，dp 数组中的最大值即为最长单调递增子序列的长度。

时间复杂度：算法为二重循环，故时间复杂度为 O（n^2）。

3-4 二维 0-1 背包问题。

给定 n 种物品和一背包。物品 i 的重量是 w_i，体积是 b_i，其价值为 v_i，背包的容量为 c，容积为 d。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

答：**思路：**定义动态规划数组 dp[j][k]，表示背包重量不超过 j、体积不超过 k 时的最大价值，初始化为全 0。遍历每个物品 i，逆序遍历重量 j（从背包容量 c 到物品重量 wi）和体积 k（从背包容积 d 到物品体积 bi），通过状态转移方程 dp[j][k]=max(dp[j][k], dp[j-wi][k-bi]+vi) 更新价值。最终 dp[c][d]即为最大总价值。

时间复杂度：算法需遍历 n 种物品，对每个物品遍历重量 c 次、体积 d 次，整体时间复杂度为 **O(n×c×d)**。

算法设计题：

3-3 石子合并问题。

问题描述：在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。试设计一个算法，计算出将 n 堆石子合并成一堆的最小得分和最大得分。

算法设计：对于给定 n 堆石子，计算合并成一堆的最小得分和最大得分。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n（$1 \leq n \leq 100$），表示有 n 堆石子。第 2 行有 n 个数，分别表示每堆石子的个数。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行的数是最小得分，第 2 行中的数是最大得分。

输入文件示例	输出文件示例
input.txt	output.txt
4	43
4 4 5 9	54

答：**思路：** 由于石子呈环形摆放，将原始长度为 n 的石子堆数组复制一份，形成长度为 2n 的线性数组，将环形问题转化为线性区间合并问题，便于动态规划处理。

动态规划状态定义：

定义 dp_min[i][j]表示合并区间[i, j]石子堆的最小得分。

定义 dp_max[i][j]表示合并区间[i, j]石子堆的最大得分。

预处理 sum[i][j]表示区间[i, j]石子堆的总数和（合并后的得分基数）。

状态转移方程：枚举区间分割点 k（$i \leq k < j$），对每个区间 [i, j]：

最小得分：dp_min[i][j] = min(dp_min[i][k] + dp_min[k+1][j] + sum[i][j])

最大得分：dp_max[i][j] = max(dp_max[i][k] + dp_max[k+1][j] + sum[i][j])

遍历顺序：按区间长度从小到大遍历（长度从 2 到 n），确保计算 [i, j] 时，其子区间 [i, k] 和 [k+1, j] 已计算完成。最终在长度为 n 的区间结果中取最小值和最大值（需遍历所有起点对应的环形合并情况）

时间复杂度：$O(n^3)$。需遍历区间起点 i、终点 j，以及分割点 k，总操作次数为 $O(n^3)$。

空间复杂度：$O(n^2)$。需存储 dp_min、dp_max、sum 等二维数组，空间规模为 $O(n^2)$。

3-13　最大 k 乘积问题。

问题描述：设 I 是一个 n 位十进制整数。如果将 I 划分为 k 段，则可得到 k 个整数。这 k 个整数的乘积称为 I 的一个 k 乘积。试设计一个算法，对于给定的 I 和 k，求出 I 的最大 k 乘积。

算法设计：对于给定的 I 和 k，计算 I 的最大 k 乘积。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 k。正整数 n 是序列的长度，正整数 k 是分割的段数。接下来的一行中是一个 n 位十进制整数（$n \leqslant 10$）。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行中的数是计算出的最大 k 乘积。

输入文件示例	输出文件示例
input.txt	output.txt
2 1	15
15	

答：**思路：**定义 dp[i][j] 表示将前 i 位数字划分为 j 段时的最大乘积。其中，i 对应数字位数，j 对应分割段数。

计算 pre[i][j]，表示从第 i 位到第 j 位组成的整数（如原数字字符串为 s，则 pre[i][j] = s[i..j] 转换的整数）。

然后遍历分割段数 j（从 1 到 k），对每一位 i（从 j 到 n），枚举最后一段的起始位置 m（j-1 ≤ m < i）。dp[i][j] = max(dp[i][j], dp[m][j-1] * pre[m+1][i])，即通过前 m 位分 j-1 段的结果，结合最后一段数字的乘积，取最大值。

初始化 dp[i][1] = pre[1][i]（分 1 段时就是整个数字）。最终 dp[n][k] 即为将 n 位数字划分为 k 段的最大乘积。

3-14　最少费用购物问题。

问题描述：商店中每种商品都有标价。例如，一朵花的价格是 2 元，一个花瓶的价格是 5 元。为了吸引顾客，商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组，并降价销售。例如，3 朵花的价格不是 6 元而是 5 元，2 个花瓶加 1 朵花的优惠价是 10 元。试设计一个算法，计算出某顾客所购商品应付的最少费用。

算法设计：对于给定欲购商品的价格和数量，以及优惠商品价，计算所购商品应付的最少费用。

数据输入：由文件 input.txt 提供欲购商品数据。文件的第 1 行中有 1 个整数 B（$0 \leqslant B \leqslant 5$），表示所购商品种类数。在接下来的 B 行中，每行有 3 个数 C、K 和 P。C 表示商品的编码（每种商品有唯一编码），$1 \leqslant C \leqslant 999$；$K$ 表示购买该种商品总数，$1 \leqslant K \leqslant 5$；$P$ 是该种商品的正常单价（每件商品的价格），$1 \leqslant P \leqslant 999$。注意，一次最多可购买 $5 \times 5 = 25$ 件商品。

由文件 offer.txt 提供优惠商品价数据。文件的第 1 行中有 1 个整数 S（$0 \leqslant S \leqslant 99$），表示共有 S 种优惠商品组合。接下来的 S 行，每行的第 1 个数描述优惠商品组合中商品的种类数 j。接着是 j 个数字对（C、K），其中 C 是商品编码，$1 \leqslant C \leqslant 999$；$K$ 表示该种商品在此组合中的数量，$1 \leqslant K \leqslant 5$。每行最后一个数字 P（$1 \leqslant P \leqslant 9999$）表示此商品组合的优惠价。

结果输出：将计算出的所购商品应付的最少费用输出到文件 output.txt。

输入文件示例		输出文件示例
input.txt	offer.txt	output.txt
2	2	14
7 3 2	1 7 3 5	
8 2 5	2 7 1 8 2 10	

答：**思路：**定义 cost(a, b, c, d, e) 表示剩余需购买商品数量为 (a, b, c, d, e)（对应不同商品编码）时的最少费用。因商品种类最多 5 种，每种最多购 5 件，状态空间有限，可覆盖所有可能的剩余购买组合。

应用优惠方案：对每种优惠方案 m，若当前剩余数量满足优惠组合要求（即每种商品剩余数 ≥ 优惠组合中对应商品的数量），则转移状态为减去优惠组合数量后的新剩余量，费用更新为 cost(新剩余量) + offer(m)。遍历所有优惠方案，取最小值。

不应用优惠：直接按原价购买剩余商品，计算对应费用，与应用优惠的情况比较，取更小值。

遍历：通过递归或迭代方式，从初始状态出发，穷举所有可能的优惠应用组合，逐步更新每个状态的最小费用，最终得到初始状态对应的 cost 值即为最少费用。

3-17　字符串比较问题。

问题描述：对于长度相同的两个字符串 A 和 B，其距离定义为相应位置字符距离之和。两个非空格字符的距离是它们的 ASCII 编码之差的绝对值。空格与空格的距离为 0，空格与其他字符的距离为一定值 k。

在一般情况下，字符串 A 和 B 的长度不一定相同。字符串 A 的扩展是在 A 中插入若干空格字符所产生的字符串。在字符串 A 和 B 的所有长度相同的扩展中，有一对距离最小的扩展，该距离称为字符串 A 和 B 的扩展距离。

对于给定的字符串 A 和 B，试设计一个算法，计算其扩展距离。

算法设计：对于给定的字符串 A 和 B，计算其扩展距离。

数据输入：由文件 input.txt 给出输入数据。第 1 行是字符串 A，第 2 行是字符串 B，第 3 行是空格与其他字符的距离定值 k。

结果输出：将计算出的字符串 A 和 B 的扩展距离输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
cmc	10
snmn	
2	

答：**思路：**设 dp[i][j] 表示字符串 A 的前 i 个字符与字符串 B 的前 j 个字符扩展后的最小距离。

初始化：若 A 为空（i=0），需对 B 的前 j 个字符全插空格，距离为 j×k，即 dp[0][j] = j×k。

若 B 为空（j=0），需对 A 的前 i 个字符全插空格，距离为 i×k，即 dp[i][0] = i×k。

根据题意可以分为三种情况：

不插空格：若 A 的第 i 个字符与 B 的第 j 个字符均不插空格，距离为两字符实际距离（非空格按 ASCII 差，有空格按 k 计算），加上 dp[i-1][j-1]。

A 插空格：B 的第 j 个字符不插空格，A 对应位置插空格，距离为 k + dp[i][j-1]。

B 插空格：A 的第 i 个字符不插空格，B 对应位置插空格，距离为 k + dp[i-1][j]。
取三种情况的最小值：$dp[i][j]=\min$（字符距离$+dp[i-1][j-1],k+dp[i][j-1],k+dp[i-1][j]$
）

最终 dp[len(A)][len(B)] 即为字符串 A 和 B 的扩展距离。