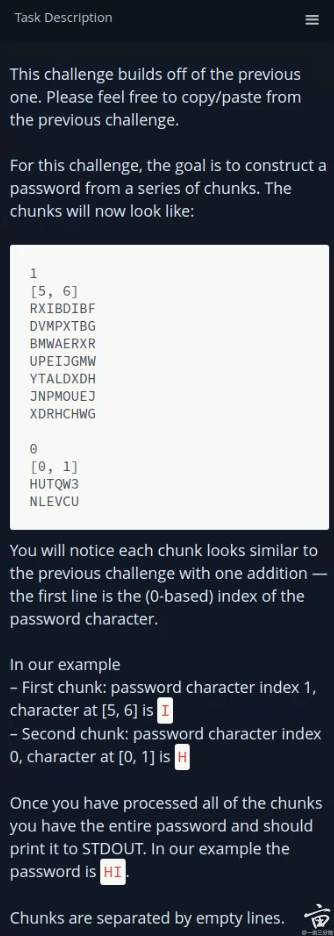
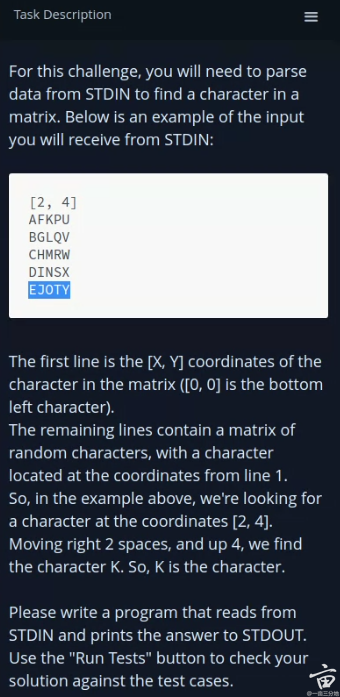
**Decode password**



**题目本质**

* 给定一个文件路径，文件中有：
  + **第一问**：坐标 [x, y] + 一个矩阵 → 返回对应字符。
  + **第二问**：多个 block，每个 block 包含一个 **index**、一个坐标 [x, y] 和一个矩阵，中间用空行隔开 → 按照 index 拼接字符组成密码。
  + **第三问**：多个密码在同一个文件里，识别“第一个完整密码”，一旦遇到重复的 index，就认为是新密码开始 → 返回第一个完整密码。

**常见 follow-up & 考点**

1. **输入处理**：面试官会明确要求从文件读取（而不是 stdin），重点是要熟悉文件读取 API（Java: Scanner(File)，Python: open(path)，JS: fs.readFileSync(path)）。
2. **优化方向**（加问）：
   * 空间优化：不需要整行整矩阵都存，只存你要取的那一列字符，用 rolling update/fixed size queue 来节省内存。
   * 生产代码：建议把“读取 block”的逻辑封装成函数，代码更干净。
3. **字符串解析**：用正则/简单 split 来解析坐标。
4. **结束条件**：第三问的重点是“什么时候算密码完整”。常见做法：用一个 Map 记录 index → char，当发现 index 重复，就说明进入下一个密码，立即返回第一个。

**第一问 decodeFromFile(path)**

* ✅ 用 fs.readFileSync(path) 读文件，和面经要求一致（Java Scanner 的 JS 等价物）。
* ✅ 解析坐标 [x,y] 用了 split(/,\s\*/)，能兼容 [2,3] 和 [2, 3]，符合面经里提到的 regex parse。
* ✅ 没有用 reverse()，而是直接计算 row = board.length - 1 - y，逻辑完全正确。
* ✔️ 面经里提到“空间优化”你还没加，但题目默认可以 load 全部文件，这样写也完全 OK。

**第二问 decodeBlocksFromFile(path)**

* ✅ 逻辑和面经一致：遇到空行分隔 block，读 index + 坐标 + board，存进 Map，最后按 0..maxIndex 拼接。
* ✅ 面经强调的“parse block”你是用 while 循环实现的，干净明了。
* ✔️ 面经里有人提到可以“提前检测 index 连续性”，但题目没要求必须，你的写法完全能过。

**第三问 decodeFirstCompletePassword(path)**

* ✅ 和面经要求的“遇到重复 index 就认为新密码开始，返回第一个”完全 match。
* ✅ 每次读完 block 就检查是否完整，一旦完整就返回，逻辑与面经描述完全对齐。
* ✅ 使用 Map 存储，能避免重复 index 冲突。

**总规则（适用于三问）**

* **输入来源**：从 **STDIN** 读取全部文本；输出打印到 **STDOUT**。
* **坐标系**：原点在**左下角** (0,0)；x 向右增，y 向上增。
* **矩阵输入顺序**：矩阵按\*\*从上到下（top → bottom）\*\*给出。
* **取字符规则**：若矩阵共有 m 行、每行 n 列（行号自上而下是 0..m-1），坐标 [x, y] 对应的字符位于：
  + 目标行索引：row = (m - 1 - y)（因为 y=0 在最底行）
  + 目标列索引：col = x
  + 取值：matrix[row][col]
* **坐标格式**：第一行（或每个块的第二行）是形如 "[x, y]" 的坐标，空格可有可无（[x,y] / [x, y] 等都应被接受）。
* **行分隔**：输入可能是 \n 或 \r\n，需要兼容。
* **边界**：题目保证给定坐标一定在矩阵范围内（无需越界防御即可通过）。

**问题一（Q1）：在一个矩阵中取坐标字符**

**输入（STDIN）：**

* 第 1 行：坐标 [x, y]
* 第 2 行起：矩阵文本，每行相同长度（英文字母或任意字符），顺序为 **top → bottom**

**输出（STDOUT）：**

* 打印位于坐标 [x, y] 的**单个字符**

**示例：**

输入：

[2, 4]

AFLOW

BGLOW

COSMW

DENSX

输出：

S

解释：[2,4] 表示从左往右 x=2，从下往上 y=4，落在字符 S 处。

**问题二（Q2）：由多个“块（chunk）”拼接成一个密码**

**输入（STDIN）：** 由若干 **chunk** 串联构成，每个 **chunk** 结构如下：

1. 第 1 行：**索引**（0-based 整数），表示该字符在最终密码中的位置；
2. 第 2 行：坐标 [x, y]；
3. 第 3 行起：**矩阵**（top → bottom），紧接着下一个 chunk 的索引行或输入结束（EOF）。

注意：**chunk 与 chunk 之间没有强制空行**。一个 chunk 何时结束？当你读到\*\*下一行是一个新的索引（纯数字行）\*\*时，说明上一个 chunk 的矩阵到此结束；或遇到 **EOF**。

**处理：**

* 依次读取每个 chunk；
* 对于每个 chunk，根据其坐标从矩阵提取一个字符；
* 将该字符放到密码的指定 **index** 上（0-based）；
* **所有 chunk 处理完毕后**，按索引从 0..maxIndex 拼接成密码并输出。

**输出（STDOUT）：**

* 打印完整密码字符串（按索引顺序拼好的字符）

**示例：**

输入：

1

[3, 4]

SOMETHING

UMFJZOWE

VALDKDMS

ZANDOWSD

WONDOROW

2

[0, 3]

MLFOWSD

输出：

IM

解释：

* chunk1：index=1，字符位于 [3,4]，取到 'I'
* chunk2：index=2，字符位于 [0,3]，取到 'M'
* 最终密码（从 index=1 开始到 index=2）为 "IM"

**问题三（Q3）：输入包含多个密码，输出第一个完整密码**

**输入（STDIN）：** 仍是由多个 **chunk** 串成，chunk 的结构与 Q2 完全相同（index 行 + 坐标行 + 矩阵若干行）。**输入里包含多个密码**，这些密码是一个接着一个直接串起来的。

**判定与输出规则：**

* 从头开始读 chunk，并将字符按照其 **index** 填入当前构建中的密码；
* **当发现 index 重复**（即一个已经出现过的 index 又出现了），这表示**下一个密码开始了**：
  + 此时，**前一个密码**被视为已经结束；
  + 将前一个密码按 0..maxIndex 的索引顺序拼接并**立即输出**（忽略后续所有输入）；
* 或者，你也可以选择在每次填入字符后检查是否已经填满 0..maxIndex 的连续索引：一旦“完整”，立刻输出第一个完整密码并结束；
* 两种等价的终止方式任选其一，常用做法是“遇到**重复 index**立即输出前一个密码”。

**输出（STDOUT）：**

* 打印**第一个**完整密码的字符串

**示例（展示“重复 index ⇒ 下一密码开始”的场景）：**

输入：

1

[2, 3]

ABCDEFG

HIGKLMN

OPQRSTU

VWXYZAB

0

[0, 0]

ABCDEFG

HIGKLMN

OPQRSTU

VWXYZAB

0

[1,1]

ABC

DEF

GHI

1

[2,2]

ABC

DEF

GHI

输出：

VC

解释：

* 前两个 chunk 组成第一个密码，其中 index=1 → 'C'，index=0 → 'V'，第一串密码是 "VC"；
* 第三个 chunk 出现 index=0（**重复**），标志第二个密码开始，因此在此处就输出第一个密码 "VC"，后续不再处理。

**实施细节与建议（助你 AC）**

* **解析 index 行**：判断一行是否为“纯数字”（可能需要 ^\d+$）以识别新 chunk 的开始。
* **解析坐标**："[x, y]" 或 "[x,y]" 都可能出现，写解析时允许逗号后可有可无空格（正则 ,\s\*）。
* **矩阵结束**：当下一行是“纯数字行”（新 index）或到达 EOF，视为当前矩阵结束。
* **字符拼接**：
  + Q2：处理完全部 chunk 之后，再从 0..maxIndex 组装密码；
  + Q3：可在遇到重复 index 或确认 0..maxIndex 全部到齐时立刻输出第一个密码；
* **时间/空间**：输入规模较小（常见约束如 text.length ≤ 1000），一次读入内存即可；若被问到优化，可提“只保留需要的那一列字符、滚动缓存行”等作为进阶答法。

**📌 题目总结（包含三问）**

**背景**

* 输入文件来自 STDIN，包含坐标和一个由随机字符组成的矩阵。
* 坐标 [x, y] 的原点 (0,0) 在矩阵的左下角。
* 需要根据给定坐标在矩阵中找到对应的字符，或拼接多个字符组成密码。

**第一问：单字符查找**

**输入格式：**

[2, 4]

AFLOW

BGLOW

COSMW

DENSX

* 第一行是坐标 [x, y]。
* 后续几行是矩阵，从 **上到下** 给出。
* 目标字符位置：矩阵高度 - 1 - y = 行号，列号 = x。

**示例：**

* 输入坐标 [2, 4]，矩阵高度为 4（0~3 行），所以目标行 = 4 - 1 - 4 = -1（翻转后算正确的行）。
* 找到的字符就是 S。

👉 **任务**：编写程序，读取输入并输出该字符。

**第二问：单个密码（由多个 chunk 组成）**

**输入格式：**

1

[3, 4]

SOMETHING

UMFJZOWE

VALDKDMS

ZANDOWSD

WONDOROW

2

[0, 3]

MLFOWSD

* 输入包含多个 **chunk**，每个 chunk 的格式为：
  1. 第一行是密码字符的 **index**（0-based 索引）。
  2. 第二行是坐标 [x, y]。
  3. 后续几行是矩阵。
* 块与块之间可能用空行分隔。

**示例：**

* 第 1 个 chunk：index = 1，坐标 [3,4] → 矩阵字符 = 'I'。
* 第 2 个 chunk：index = 2，坐标 [0,3] → 矩阵字符 = 'M'。

👉 **任务**：按索引顺序拼接字符，得到完整密码 "IM"。

**第三问：多个密码**

**输入格式：**

0

[2, 1]

ABC

DEF

1

[0, 0]

XYZ

<空行>

0

[1, 2]

PQRS

TUVW

XYZA

* 输入文件中包含 **多个密码**，每个密码由若干 chunk 组成。
* **不同密码之间用空行分隔**。
* 每个密码仍然要求索引从 0 到 max 连续，否则该密码无效/返回空串。

**示例：**

* 第一组 chunk → 密码1 = "??"。
* 空行分隔后，第二组 chunk → 密码2 = "??"。

👉 **任务**：处理整个文件，输出所有密码（通常是一行一个）。

**✅ 总结要点**

1. **第一问**：给一个坐标和矩阵 → 输出单个字符。
2. **第二问**：给一组 chunk（index + 坐标 + 矩阵） → 拼接成一个密码。
3. **第三问**：输入包含多组密码（用空行分隔的 chunk 集合） → 输出所有密码。

**Follow-up 问题简版总结**

**Q1. How can you optimize the space usage?**

* **复杂度**：
  + 时间：O(N)
  + 空间：O(1)
* **面试回答**：
  + EN: *“Instead of storing the whole matrix, I can scan line by line and only keep the character at column x. That reduces space from O(N*M) to O(1). Time is still O(N).”\*
  + ZH: “我可以逐行扫描，只保留第 x 列的字符，空间从 O(N\*M) 降到 O(1)，时间还是 O(N)。”

**Q2. What if the file/matrix is very large?**

* **复杂度**：
  + 时间：O(N)
  + 空间：O(1)
* **面试回答**：
  + EN: *“For very large files, I’d use a streaming approach: read line by line and only process the needed character, discarding the rest immediately. This keeps memory almost constant regardless of file size.”*
  + ZH: “大文件场景我会用流式处理，逐行读取并只处理需要的部分，其余立刻丢掉。这样内存占用几乎不随文件大小增加。”

**Q3. How to do further optimization using rolling update ?**

**复杂度**：

* + 时间：O(N)
  + 空间：O(K\*W)（K = 保留的行数，通常远小于 N）
* **面试回答**：
  + EN: *“If we need multiple recent rows, we can use a rolling buffer with fixed size, e.g. 2–3 rows. New rows overwrite the oldest. That reduces space from O(N*M) to O(K*W).”*
  + ZH: “如果需要多行，可以用固定大小的滚动缓冲，只存最近几行，新行进来就丢掉最旧的。这样空间从 O(N*M) 优化到 O(K*W)。”

**Q4. Is there any trade-off in these optimizations?**

* **复杂度**：
  + 时间：O(N)
  + 空间：优化后更小（O(1) 或 O(K\*W)）
* **面试回答**：
  + EN: *“The trade-off is that memory drops significantly, but time remains O(N). Rolling update also adds some implementation complexity, but it’s worthwhile for large inputs.”*
  + ZH: “权衡点是节省了内存，但时间复杂度仍然是 O(N)，rolling update 也增加了实现复杂度。不过在大规模输入下这是值得的。”

**Find Minimal Shoppers**

这道题是 **LeetCode 1701. Average Waiting Time** 的进阶版。

**背景**

一家物流公司有多个 **shoppers**（相当于服务员/工人），需要处理客户订单。每个订单包含：

* duration：完成该订单所需的时间
* arrivalTime：订单到达的时间

**规则**

1. 每个 shopper 一次只能处理一个订单。
2. 订单必须按到达时间顺序处理。
3. 如果某个订单到达时有 shopper 空闲，则立即开始处理；否则需要等待最早空闲的 shopper。
4. **等待时间** = 订单完成时间 - 到达时间。
5. **平均等待时间** = 所有订单的等待时间之和 ÷ 总订单数。

**要求**

给定一个阈值 k，求 **最少需要多少个 shoppers**，才能保证所有订单的平均等待时间不超过 k。

* 如果无法满足条件，返回 -1。

**示例**

* orders = [[4,1],[5,2],[2,3]], k=5 → 需要 2 个 shoppers
* orders = [[4,1],[4,2],[4,3],[4,4]], k=4.3 → 需要 3 个 shoppers
* orders = [[10,1],[10,2],[10,3],[1,4]], k=2 → 无解，返回 -1

**面试考点**

* **第一问**：单个 shopper 的平均等待时间（类似 1701，重点是排序 + 模拟）。
* **第二问**：最少 shoppers 数量
  + 解法：**二分搜索 + 小根堆（priority queue）**
  + 时间复杂度需要分析，并解释为什么要用二分和堆。
* 面试官可能还会追问复杂度、是否能优化、如果是 production code 如何实现。

**题目背景**

这是一道变体题，和 LeetCode 1701. **Average Waiting Time** 有关。  
区别在于：LeetCode 1701 只有一个服务员（server），而本题扩展到了多个 **shoppers（服务员/处理人）**，要找出最少需要多少个 shoppers，才能让平均等待时间不超过给定阈值 k。

**输入**

* orders：无序二维数组，每个元素为 [duration, arrivalTime]
  + duration = 完成该订单所需时间
  + arrivalTime = 订单到达时间
* k：浮点数，要求的最大平均等待时间。

**定义**

* **等待时间**：一个订单的完成时间 - 该订单的到达时间。
* **平均等待时间**：所有订单的等待时间之和 ÷ 总订单数。

**规则**

1. 有多个 shoppers（工人），每人同时只能处理一个订单。
2. 订单必须按到达时间顺序处理（FIFO），不能跳过。
3. 如果某个订单到达时有 shopper 空闲 → 立刻开始处理。
4. 如果所有 shoppers 忙碌 → 订单排队，等最早完成的 shopper 空出来才开始处理。
5. 目标：找到最小的 shoppers 数量，使得平均等待时间 ≤ k。若无解 → 返回 -1。

**示例**

**Example 1**

orders = [[4,1],[5,2],[2,3]], k = 5.0

* 1 个 shopper → 平均等待 7.0 (> 5.0)
* 2 个 shoppers → 平均等待 4.33 (≤ 5.0)  
  → 输出 2

**Example 2**

orders = [[4,1],[4,2],[4,3],[4,4]], k = 4.3

* 最少需要 3 个 shoppers。

**Example 3**

orders = [[10,1],[10,2],[10,3],[1,4]], k = 2.0

* 无论多少 shoppers，平均等待时间都无法 ≤ 2.0  
  → 输出 -1

**面试考点**

1. **第一问（基础版）**
   * 类似 LeetCode 1701，先将订单按照 arrivalTime 排序，计算平均等待时间（1 个 shopper 情况）。
   * 考查基本模拟/排序。
2. **第二问（进阶版）**
   * 要找最小 shoppers 数量，使得平均等待时间 ≤ k。
   * 思路：
     + 用 **binary search** 在 [1, orders.length] 范围内找最小 shoppers 数量。
     + 检查时用 **priority queue（小根堆）** 模拟 shoppers 的最早空闲时间。
3. **附加讨论**
   * 时间复杂度：排序 O(n log n)，验证过程 O(n log m)（m = shoppers 数量），二分整体 O(n log n log m)。
   * 空间复杂度：O(m)
   * 面试官可能会追问为什么用 heap、二分是否最优、生产环境如何优化（如分布式调度、批处理策略）。

**Find Minimal Shoppers 面经两问题意总结**

**第一问（对应 LeetCode 1701 变体）**

* **场景**：有一个 shopper（或厨师）处理订单。
* **输入**：
  + orders[i] = [duration, arrivalTime]（可能未排序）。
  + duration = 订单处理时间
  + arrivalTime = 订单到达时间
* **规则**：
  + 订单必须按照到达顺序处理。
  + 如果 shopper 空闲 → 立刻处理；否则等到他完成上一个订单。
* **要求**：计算所有订单的 **平均等待时间**。
* **本质**：就是 **LeetCode 1701**，只不过输入未排序，所以要先 sort by arrivalTime。

**第二问（进阶版，本题重点）**

* **场景**：有多个 shoppers（工人）可以并行处理订单。
* **输入**：
  + 同样的 orders[i] = [duration, arrivalTime]（未排序）。
  + 给定一个浮点数 k。
* **规则**：
  + 当一个订单到达时：
    - 如果有空闲 shopper → 立即分配。
    - 如果都忙 → 等最早完成的 shopper 空出来再处理。
  + 所有订单仍需保持到达顺序。
  + **等待时间** = 完成时间 - 到达时间。
* **要求**：找出 **最少需要多少 shoppers**，才能让平均等待时间 ≤ k。
  + 如果无论多少 shoppers 都不行，返回 -1。
* **关键点**：
  + 用 **priority queue** 模拟 shoppers 的最早空闲时间。
  + 用 **binary search** 在 [1, n] 范围内搜索最小人数。

**总结一句话**

* **第一问**：一个 shopper，算平均等待时间（1701 + 排序）。
* **第二问**：多个 shoppers，要求平均等待 ≤ k，返回所需最少人数（heap + binary search）。

**Follow-up 问题简版总结**

**1) 这一步/这个参数的意义是什么？**

**CN：**

* **排序（按 arrivalTime）**：把未排序输入变为时间线，保证按到达顺序处理。
* **最小堆（shoppers 的下一空闲时刻）**：随时拿到**最早空闲**的工人，模拟调度。
* **m（工人数）**：容量参数，越大平均等待越小，存在**单调性**。
* **k（平均等待阈值）**：业务 SLA；我们的目标是找到最小 m 使平均等待 ≤ k。
* **can(m)**：可行性判定；给定 m 计算平均等待是否达标。
* **二分**：利用“m 增大 → 等待不增（单调）”来找**最小**可行解。

**EN:**

* **Sort by arrivalTime:** Build a timeline from unsorted input; enforce FIFO by arrival.
* **Min-heap (next-free time per shopper):** Always pick the **earliest-available** worker to simulate scheduling.
* **m (number of shoppers):** Capacity knob; larger m ⇒ smaller (or equal) average wait; **monotonic**.
* **k (avg-wait SLA):** Business target; we seek the minimal m s.t. average wait ≤ k.
* **can(m):** Feasibility check; compute average wait under m.
* **Binary search:** Exploits monotonicity to find the **minimal** feasible m.

**2) 这一步的效率是多少？**

**CN：**

* 排序：O(n log n)
* can(m)（堆模拟）：O(n log m)
* 二分迭代次数：O(log n)

**EN:**

* Sorting: O(n log n)
* can(m) (heap simulation): O(n log m)
* Binary-search iterations: O(log n)

**3) 总体效率是多少？**

**CN：** 总体 O(n log n + n log m log n)；最坏 m≈n，可记为 **O(n log² n)**。空间为堆 O(m)，最坏 O(n)。  
**EN:** Overall O(n log n + n log m log n), worst-case **O(n log² n)** when m≈n. Space O(m) (heap), up to O(n).

**4) 能更快吗？为什么选二分？**

**CN：**

* 选择二分是因为**单调性**：m 越大，平均等待不增（可行集合是前缀）。
* 比起线性扫描 m=1..n（O(n \* n log n)），二分将外层从 n 降到 log n。
* 下界上，必须至少排序一次（或等价线性稳定分桶）来建立到达顺序；一次 can(m) 至少要 n 次堆操作。因此在**通用**比较模型下，再快很难。
* 若利用**约束上界**（如 arrival ≤ 1e4），可用**计数排序/桶**取代比较排序，把排序降到 O(n + U)（U 为时间上界），整体可到 O((n+U) + n log m log n)。

**EN:**

* We use binary search due to **monotonicity**: average wait is non-increasing with m, so the feasible set is a prefix.
* Versus linear scan over m, binary search cuts the outer factor from n to log n.
* Lower bounds: you must at least order by arrival (or bucket by bounded domain), and each feasibility check touches all n orders with heap ops. In the comparison model, asymptotically faster is hard.
* With bounded arrival (e.g., ≤1e4), replace comparison sort with **counting/bucket sort** to get O(n+U) sorting, improving constants.

**5) 存储效率是多少？**

**CN：** 堆存 m 个下一空闲时刻：O(m)；若只做第一问（单 worker），为 O(1) 额外空间（排序栈 O(log n)）。  
**EN:** O(m) for the heap (next-free times). For Q1 (single worker), O(1) extra (plus O(log n) sorting stack).

**6) 复杂度与优化（可口述版本）**

**CN（一句话）：**

* 时间：O(n log² n)（排序 O(n log n) + 二分 O(log n) × 可行性 O(n log m)）。
* 空间：O(m)。
* 优化：剪枝 sum(duration) > k\*n ⇒ -1；达标时早停；用整数比较 total\_wait ≤ k\*n 降低浮点误差；若时间上界小，用桶排序降常数。

**EN (one-liner):**

* Time: O(n log² n) overall; space: O(m).
* Optimizations: prune with sum(duration) > k\*n ⇒ -1, early-stop when exceeding k\*n, compare integers (total\_wait vs k\*n) to avoid FP drift, and use bucket/counting sort when time domain is bounded.

**7) 如果是生产代码，如何改进？**

**CN：**

* **正确性与健壮性**：
  + 精度安全：用整数累计等待时间，与 k\*n 比较；边界（k=0、同一时刻大量到达、极大 n）。
  + 输入校验与防御式编程；单/集成测试（含极端与随机用例）。
* **性能与可维护**：
  + 采用桶排序（若时间上限小）、避免重复分配、尽量一次遍历。
  + 将 can(m) 纯函数化，便于基准测试与并行 A/B 对比。
* **系统化（服务化场景）**：
  + **可观测性**：打点平均等待、P95/P99、二分迭代次数、堆大小、超时率。
  + **参数化**：k、最大 m、二分边界、早停阈值可配置；特性开关。
  + **弹性与容量**：根据实时负载（到达率/处理时长分布）动态估算所需 m 并与自动扩缩容系统对接。
  + **鲁棒性**：超时和回退策略、重试/幂等、速率限制、背压（backpressure）。
  + **数据治理**：输入去噪与异常检测（异常 duration/arrival）。

**EN:**

* **Correctness & robustness:**
  + Numeric safety: keep totals as integers and compare to k\*n; handle edges (k=0, bursty arrivals, large n).
  + Input validation; strong unit/integration tests (edge and randomized cases).
* **Performance & maintainability:**
  + Bucket sort when time domain is small; avoid reallocations; single-pass patterns.
  + Make can(m) pure for benchmarking and easier parallel A/B checks.
* **Operational (service context):**
  + **Observability:** metrics for avg wait, P95/P99, binary-search iterations, heap size, timeouts.
  + **Configurability:** expose k, max m, search bounds, early-stop thresholds; feature flags.
  + **Elasticity & capacity:** integrate with autoscaling using live arrival/service distributions.
  + **Resilience:** timeouts & fallbacks, retries/idempotency, rate limiting, backpressure.
  + **Data hygiene:** sanitize/validate inputs; detect anomalous duration/arrival.

**Follow-up: 为什么使用 heap？**

**CN：**

* 如果不用 heap，只用数组存储每个 shopper 的空闲时间，每次新订单都要 **线性扫描**所有 shoppers 找最早空闲的那个，效率是 O(m)。
* 而 heap 能保证在 O(log m) 时间内拿到并更新**最早空闲**的 shopper，极大降低了复杂度。
* 因为我们要重复 n 次（每个订单一次），使用 heap 把整体从 O(n·m) 优化到 O(n log m)。
* 另外，heap 在数据量较大时也更节省内存局部性，相比排序每次重排更加高效。

**EN:**

* Without a heap, we would need to **linearly scan** all shoppers’ free times to find the earliest available one for each order, which costs O(m) per order.
* A min-heap guarantees we can always extract and update the **earliest-free** shopper in O(log m) time.
* Since we repeat this for all n orders, heap reduces the total from O(n·m) to O(n log m).
* Heap also maintains efficiency without fully re-sorting the list of shoppers each time.

👉 面试时简答一句话就够：  
**CN：**“因为要频繁取最早空闲的 shopper，heap 能在 O(log m) 完成，比线性扫描 O(m) 高效得多。”  
**EN:** “Because we frequently need the earliest-free shopper; a heap gives that in O(log m) instead of O(m) with linear scan.”

**Expression**

**题目还原（按面经输入格式）**

**一、基础版**

**输入**

* target: string
* expressions: string[]，每条形如 "T1 = 5" 或 "T2 = T1"（右侧仅整数或单变量）

**要求**

* 解析依赖链，从 target 出发 DFS 追溯到整数，返回 **字符串形式** 的值。
* 每个变量**恰好被定义一次**；**无环**、**无未定义变量**。

**示例（你的面经风格）**

* target="T2", expressions=["T1 = 1","T2 = T3","T3 = T1"] → 返回 "1"

**二、Follow-up 1（支持加/减表达式）**

**输入**

* target: string
* expressions: string[]，每条形如：
  + T = <term> ( (+|-) <term> )\*，<term> 可为 **整数或变量**，运算符与操作数**以空格分隔**

**要求**

* 右侧可以包含**一个或多个**加/减运算与多个变量/整数（单层、左结合）。
* 每个变量**恰好被定义一次**；**无环**、**无未定义变量**。
* 自顶向下递归求值 + 记忆化，返回 target 的字符串值。

**示例**

* target="T2",  
  ["T1 = 1","T2 = T3 - 5","T3 = T1 + 7"] → "3"
* target="T2",  
  ["T1 = 1","T2 = 2 + T4","T3 = T1 - 4","T4 = T1 + T3"] → "0"

**三、Follow-up 2（允许无效/歧义；需判空）**

**输入**

* target: string
* expressions: string[]，**变量可能被多次定义**；表达式仍只含整数/变量/+/-/空格

**返回**

* 仅当能**唯一且一致**地确定 target 的值时，返回其**字符串**；
* 否则返回空串 ""。出现以下任一情况即返回 ""：
  1. **依赖成环**（如 T1=T2, T2=T1 且目标涉及该环）；
  2. **目标变量未定义**；
  3. **同一变量多定义且导致不同结果**（冲突）
     + 若同一变量多条定义都可达且**结果一致**，仍可接受；
     + 只要存在可达且**结果不一致**的定义，或可达性导致值不唯一，即判 ""。

**注**：你先前中文笔记里有“返回 IMPOSSIBLE”的表述——以**面经题**为准，这里规范为**返回空串 ""**。

**统一解法要点（覆盖三问，按面经输入）**

* 建表：defs: Map<string, string[]>，记录**变量 -> 所有右侧表达式**（三问统一；一、二问每变量仅一条）。
* eval(var)（DFS 自顶向下）：
  + visiting 检测环；memo 记忆化**唯一已确值**。
  + 解析右侧表达式：按空格 split 得 token，左结合处理 +/-。变量递归求值，整数直接解析（注意负数）。
  + **第三问**：若同一变量有多条定义，分别求值并收集：
    - 去掉“不可确定”的结果后，若得到的不同数值 ≥ 2 → 冲突；
    - 若全不可确定 → 不可确定；
    - 否则唯一值成立并写入 memo。
* 仅从 target 出发，不必全量解所有变量。
* 返回：
  + 一/二问：必定能解，返回值字符串；
  + 三问：若不可唯一确定（环/未定义/冲突），返回 ""。

**Follow Up常见追问 & 速答（中英双语）**

**1) 这个步骤/参数是做什么用的？**

**中**：

* defs：把每个变量映射到它的表达式列表，便于按变量快速找到 RHS。
* memo：缓存已确定的变量值，避免重复计算。
* visiting：DFS 过程中检测环，防止自依赖导致无限递归。
* evalExpr(expr)：按空格切分、左结合计算 +/- 表达式。
* evalVar(v)：从目标变量出发，递归解析依赖链。

**EN**:

* defs: maps each variable to its RHS expressions for quick lookup.
* memo: caches resolved values to avoid recomputation.
* visiting: detects cycles during DFS to prevent infinite recursion.
* evalExpr(expr): tokenizes by spaces and evaluates +/- left-to-right.
* evalVar(v): recursively resolves the dependency chain from the target.

**2) 这个步骤的效率是多少？(e.g., evalVar / evalExpr)**

**中**：

* evalVar：每个变量最多被确定一次，均摊 **O(1)** 次访问，整体 **O(M)**（M 为所有表达式 token 总数）。
* evalExpr：线性扫描 token，单次 **O(k)**，k 是该表达式 token 数。

**EN**:

* evalVar: each variable is resolved at most once; amortized constant lookups → overall **O(M)** where M is total tokens.
* evalExpr: linear in tokens of that expression → **O(k)** per call.

**3) 总体时间复杂度是多少？还能更快吗？**

**中**：

* **时间复杂度**：
  + Round 1：**O(N)**（表达式总数）。
  + Round 2/3：**O(M)**（总 token 数），依赖只走一遍。
* **是否更快**：理论下界也是线性（必须至少看一遍输入），因此已是最优数量级。

**EN**:

* **Time**:
  + Round 1: **O(N)** by number of assignments.
  + Round 2/3: **O(M)** by total tokens.
* **Faster?** Linear is optimal since we must read the entire input at least once.

**4) 存储效率是多少？**

**中**：

* **空间复杂度**：defs + memo + visiting + 递归栈。
  + Round 1：**O(N)**。
  + Round 2/3：**O(N + M)**（保存表达式文本/词元的规模）。

**EN**:

* **Space Complexity**: defs, memo, visiting, and recursion stack.
  + Round 1: **O(N)**.
  + Round 2/3: **O(N + M)** due to expression storage/tokens.

**5) 为什么选择 DFS + 记忆化？（如果问到“为什么不用拓扑排序/二分搜索”等）**

**中**：

* 用 DFS + memo 从 target 出发，只解析**必要子图**，实现简单且访问量最小。
* 拓扑排序也可行，但会**全量**处理 DAG，没必要。
* 二分搜索需要单调性，这里不适用。

**EN**:

* DFS + memo resolves **only the reachable subgraph** from the target, simple and efficient.
* Topological sort works but processes the **entire** DAG unnecessarily.
* Binary search requires monotonicity; not applicable here.

**6) 如果要上线为 production code，你会怎么改进？**

**中**：

* **解析优化**：预处理/预编译表达式为 AST 或预 token 化，避免多次 split。
* **健壮性**：更严格的输入校验、报错信息（未定义、冲突、环）、边界数值校验。
* **性能**：大输入改为**迭代**或**Kahn 拓扑**避免深递归；限制最大深度；可选启发式缓存/持久化缓存。
* **可观测性**：指标、日志、tracing，方便定位循环/异常定义。
* **测试**：单测/性质测试（随机生成可验证的 DAG 和含环用例）；fuzzing。
* **并发/安全**：如果来自多用户输入，隔离解析上下文，防止互相污染。

**EN**:

* **Parsing**: pre-tokenize or compile to an AST to avoid repeated splits.
* **Robustness**: strict validation, explicit error types (undefined, conflict, cycle), numeric bounds checks.
* **Performance**: iterative evaluation or Kahn’s topological order to avoid deep recursion; depth limits; optional caching.
* **Observability**: metrics, logs, tracing for cycles and bad inputs.
* **Testing**: unit/property tests, fuzzing with random DAGs and cyclic cases.
* **Concurrency/Safety**: isolate evaluation contexts per user/input.

**7) 你如何分析每个步骤的复杂度并做优化？**

**中**：

* **瓶颈**通常在：字符串解析（split）与重复解析。
* **优化**：
  1. 构建时就**预 token 化**；
  2. 只从 target 可达子图解析；
  3. 记忆化只缓存**已确定**的值；
  4. 大规模数据时：换成**拓扑排序**的迭代解法，减少栈风险。

**EN**:

* **Hotspots**: string tokenization and repeated parsing.
* **Optimizations**:
  1. pre-tokenize at build time;
  2. traverse only the reachable subgraph from the target;
  3. memoize only **determined** values;
  4. switch to **topological** iterative eval for large graphs to avoid deep stacks.

**8) 如果让你进一步提速/降内存，你会怎么做？**

**中**：

* 解析期把变量名**映射成整数 ID**；token 用小型结构（如 {kind, id/int}）。
* 共享常见小整数字面量（interning）。
* 对无歧义场景，把单定义变量**内联折叠**（constant folding）。
* 表达式很长时，按需流式解析（lazy tokenize）。

**EN**:

* Map variable names to integer IDs; use compact token structs.
* Intern common small integer literals.
* Inline-fold single-definition variables (constant folding).
* Lazy tokenization for long expressions.

**9) 如果有人问“这里能用二分吗？为什么？”**

**中**：  
不能。二分需要**单调性**或“有序空间+判定函数”，而本题是图依赖求值，不具备单调性。

**EN**:  
No. Binary search needs a **monotonic predicate** or an ordered search space; this problem is dependency evaluation on a graph, not monotonic.

**10)（必备）三轮复杂度一句话版**

**中**：

* 第一问：时间 **O(N)**，空间 **O(N)**。
* 第二问：时间 **O(M)**，空间 **O(N + M)**。
* 第三问：时间 **O(M)**，空间 **O(N + M)**。

**EN**:

* Round 1: Time **O(N)**, Space **O(N)**.
* Round 2: Time **O(M)**, Space **O(N + M)**.
* Round 3: Time **O(M)**, Space **O(N + M)**.

**Data pivot table**

**背景**

你有一个二维表格数据（List<List<String>>），

* 第一行是表头（header），
* 后续每一行是一个订单记录。  
  表头中至少包含：order\_id, cost, sell\_price, product, date，可能还会有额外的列（如 state, region 等）。  
  日期格式固定为 "YYYY-MM-DD"。

关键点：

* **列的位置不固定**，只能通过列名来定位，不可以 hardcode 列索引。
* 每条数据都是字符串，需要自己转换类型（数字、日期等）。

**三个问题**

**第一问：总销售额**

* 要求计算整张表的总销售额。
* 本质：遍历所有行，找到 sell\_price 列，累加得到总和。
* 难点：列索引要根据表头动态确定。

**第二问：按某个列分组统计**

* 输入参数：pivotColumn（比如 state、product 等）。
* 要求：对这个列的每个不同值，分别统计它对应的总销售额。
* 类似于 SQL 的：
* SELECT pivotColumn, SUM(sell\_price)
* FROM orders
* GROUP BY pivotColumn;
* 输出：一个 “pivot table”，即每个分组的聚合结果。
* 实现方式：用一个 HashMap<String, Integer> 来累加。

**第三问：按日期 + 利润最大**

* 输入参数：pivotColumn, startDate。
* 要求：
  1. 先过滤掉所有日期小于 startDate 的订单。
  2. 在剩余订单中，计算每条的 **profit = sell\_price - cost**。
  3. 按 pivotColumn 分组，把 profit 累加。
  4. 找出 **profit 最大**的那个分组。
     + 如果有平局，取字典序最小的值。
  5. 按指定格式输出：
  6. f"The most {pivotColumn} is {value}"

例如："The most state is CA"。

* 如果没有匹配记录，返回空字符串 ""。

**补充要求**

* **时间复杂度**：三问基本都是 O(N)（一次遍历 N 行）。
* **空间复杂度**：O(K)，K 是 pivotColumn 的不同取值数目。
* **面试 follow-up**：
  + 如果数据量很大（production 环境），能否做优化？
    - 比如用数据库/分布式系统处理（Hive/Presto），或者用 streaming + partial aggregation。
  + 是否能提前终止？（如只需要 top-1 时可以用 min-heap 而不是全量排序）。

✅ **总结一句话**：  
这是一个模拟 **SQL 聚合查询 / Pivot Table** 的编程题，三问分别是：

1. 求总销售额；
2. 按列分组聚合；
3. 在给定日期过滤后，按列分组计算利润并找最大值。

**1️⃣ 现在的效率是多少？**

**Interviewer**: What’s the efficiency of your current solution?

**你可以说**：

* **中文**：

我的实现是一趟线性扫描，所以时间复杂度是 O(N)。  
第一问只需要一个累加器，是 O(1) 空间；第二问和第三问要按照分组统计，所以空间是 O(K)，K 是分组的数量。  
我把表头预先解析成了列名到索引的映射，这样查列是 O(1)。整体上已经是最优的一遍扫描方案。

* **English**:

My solution does a single linear scan, so the time complexity is O(N).  
For space, the first part only needs a single accumulator, that’s O(1). For grouping in part two and three, it’s O(K), where K is the number of unique groups.  
I also preprocessed the header into a name-to-index map, so column lookups are O(1). Overall, this is already optimal for a one-pass solution.

**2️⃣ 有没有更高的效率？**

**Interviewer**: Can this be more efficient?

**你可以说**：

* **中文**：

如果只是单次查询，我已经是一边扫描一边统计并更新最优值，没什么再优化的空间。  
但在生产环境里，可能要支持很多查询，就可以优化：  
比如数据按日期排序，查询时二分到起点，只扫后半部分；  
或者提前做预聚合，把每天或每周的结果存成物化视图；  
也可以缓存一些常见参数的结果；  
数据量特别大时，可以用 Spark 或 Presto 做分布式的局部聚合再全局合并。

* **English**:

For a single query, it’s already optimal since I scan once and update the best result on the fly.  
But in production with repeated queries, we can optimize:  
Sort or partition data by date, so with binary search we only scan the suffix;  
Pre-aggregate by day or week into materialized views;  
Cache results for common parameters;  
And if the dataset is huge, we can use distributed systems like Spark or Presto, doing local aggregation first and then global merging.

**3️⃣ 每个步骤的代价是多少？**

**Interviewer**: What’s the cost of each step in your solution?

**你可以说**：

* **中文**：

大致分三步：  
第一步解析表头，O(C)，只做一次；  
第二步逐行处理，取值、计算、更新哈希表，是 O(N)；  
第三步找最大值，我在扫描时就维护了最优，所以这一部分基本是 O(1)。  
所以主要开销就是遍历数据行。

* **English**:

It breaks down into three steps:  
Step one, parsing the header is O(C), and that’s just once.  
Step two, processing each row—reading values, doing arithmetic, updating the hash map—that’s O(N).  
Step three, finding the max, I maintain the best candidate on the fly, so that’s basically O(1).  
So the main cost is scanning all the rows.

**4️⃣ 能不能提前终止？**

**Interviewer**: Can we stop early, like once we know the top group?

**你可以说**：

* **中文**：

在无序的数据里，没办法提前结束，因为最后面的数据可能改变最优结果。  
但如果数据按日期排好序，就可以优化。比如升序存储时，我可以二分找到 startDate，只看后面的；如果是降序，当遇到小于 startDate 的时候就能停。  
还有一种情况是，如果有每个分组的上界信息，比如预聚合过，那当当前最优已经超过所有上界时，也可以提前结束。  
但是仅仅用一个堆，并不能减少扫描，堆只是避免了对所有分组做全量排序。

* **English**:

With unsorted data, we can’t really stop early, because later rows could still change the best result.  
But if the data is sorted by date, we can optimize. With ascending order, I can binary search for the startDate and only scan the suffix; with descending order, once I hit a date below startDate, I can stop.  
Another case is if we have an upper bound per group from pre-aggregation. If the current best is already higher than all remaining bounds, we can cut off.  
But just using a heap doesn’t reduce scanning—it only avoids a full sort over all groups.

**5️⃣ 如果要找 Top-K 呢？**

**Interviewer**: What if we want the top-K groups instead of just one?

**你可以说**：

* **中文**：

那我会用一个大小为 K 的最小堆来维护。  
遍历时把分组的利润放进去，如果比堆顶大就替换掉。  
最终得到的就是前 K 个分组。  
时间复杂度是 O(N + K log K)。

* **English**:

In that case, I’d maintain a min-heap of size K.  
As I scan, I push group profits into the heap, and if it’s larger than the smallest, I replace it.  
At the end, the heap holds the top K groups.  
The complexity is O(N + K log K).

**6️⃣ 如果分组数量特别大呢？**

**Interviewer**: What if the number of groups, K, is very large?

**你可以说**：

* **中文**：

如果分组基数特别大，比如按 product\_id，有上百万个，那哈希表可能撑不住。  
这种情况下可以做分片聚合，最后再合并结果。  
如果内存还是不够，可以落盘分桶再归并。  
如果只关心热门的分组，可以用近似算法，比如 Count-Min Sketch 或 Heavy Hitters，快速找到前几名。

* **English**:

If the pivot column has very high cardinality, like millions of product IDs, the hash table could blow up.  
In that case, we can shard the aggregation and merge later.  
If memory still isn’t enough, spill to disk and bucket-merge.  
And if we only care about the top hot groups, we can use approximate methods like Count-Min Sketch or Heavy Hitters to quickly estimate the top ones.

**7️⃣ 生产环境怎么优化？**

**Interviewer**: How would you optimize this in production?

**你可以说**：

* **中文**：

在生产里，我会分三层考虑：  
存储层：用列式存储，比如 Parquet，只读需要的列；数据按日期分区，这样查询时只扫部分分区。  
计算层：做预聚合，比如维护 (pivot, date\_bucket) 的累积表；大数据场景下用 Spark/Presto，先局部聚合再全局合并。  
缓存层：缓存一些热点参数的结果，或者缓存前缀和后缀和，减少重复计算。  
如果允许近似，还可以用概率算法快速找出热门分组。

* **English**:

In production, I’d think in three layers:  
**Storage**: use columnar formats like Parquet, only read needed columns; partition data by date so queries only scan relevant partitions.  
**Computation**: do pre-aggregation, like maintaining (pivot, date\_bucket) summary tables; for big data, use Spark or Presto to aggregate locally then merge globally.  
**Caching**: cache results for hot queries, or cache prefix/suffix sums to reduce recomputation.  
And if approximation is acceptable, we can use probabilistic algorithms to quickly identify hot groups.

**Wildcard Matching**

**📌 题目总览（四问） | Problem Overview (Four Questions)**

**Q1. 实现 indexOf（不能用内置函数）**

**中文**

* **任务**：给定主串 str 和目标串 target，返回 target 在 str 中第一次出现的下标；若不存在返回 -1。
* **思路**：
  + 朴素匹配：双指针逐位比较，容易实现。
  + KMP：预处理目标串的 lps/next 表，整体 O(n+m)。
* **复杂度**：
  + 朴素：时间 O(n·m)，空间 O(1)。
  + KMP：时间 O(n+m)，空间 O(m)。

**English**

* **Task**: Given a string str and a target substring target, return the first index where target occurs in str; return -1 if not found.
* **Approach**:
  + Naive: two pointers comparing one by one.
  + KMP: preprocess the target with LPS table, achieving O(n+m).
* **Complexity**:
  + Naive: Time O(n·m), Space O(1).
  + KMP: Time O(n+m), Space O(m).

**Q2. indexOf + 通配符 \* | indexOf with Wildcard \***

**中文**

* **通配规则**：\* 可以匹配任意长度（包括 0）的任意字符；其他字母必须按顺序匹配。
* **与 LC44 区别**：LC44 要求完整匹配；本题只需子串匹配并返回首个起点。
* **思路**：
  1. 合并连续 \*。
  2. 用 \* 分割模式串为字母 token。
  3. 若模式以 \* 开头：只需 tokens 按序在主串中出现 → 返回 0。
  4. 若模式不以 \* 开头：第一个 token 的出现位置就是候选起点；从该位置开始依次查找剩余 token。
  5. Token 查找可用 KMP 提速。
* **复杂度**：
  1. 设 n=|str|，L 为所有 token 总长度（≤ |pattern|）。整体近似 O(n+L)，最坏情况下可能退化，但仍可接受。

**English**

* **Wildcard Rule**: \* matches any sequence of characters (including empty). Other letters must appear in order.
* **Difference from LC44**: LC44 checks full-string match; here we want a substring match and return the first start index.
* **Approach**:
  1. Collapse consecutive \*.
  2. Split the pattern into letter tokens.
  3. If pattern starts with \*: check if tokens appear in order → return 0.
  4. If not: the first token must align with the start index; check subsequent tokens sequentially.
  5. Use KMP to speed up token search.
* **Complexity**:
  1. Let n=|str|, L=total length of tokens. Overall ≈ O(n+L). Worst case may degrade but still manageable.

**Examples | 示例**

* "\*\*A" vs "CDFGAGB" → 0
* "A\*" vs "CDFGAGB" → 4
* "B\*\*Y" vs "PROGRAMMING" → -1

**Q3. 是否 at most 1 edit distance | At Most 1 Edit Distance**

**中文**

* **允许操作**：替换一个字符 / 插入一个字符 / 删除一个字符。
* **思路**：
  1. 若 |len1 - len2| > 1 → False。
  2. 长度相等：统计不同字符个数，≤1 即 True。
  3. 长度差 1：双指针扫描，允许长串多走一步，若不一致超过一次 → False，否则 True。
* **复杂度**：时间 O(min(n,m))，空间 O(1)。

**English**

* **Allowed edits**: Replace one char / Insert one char / Delete one char.
* **Approach**:
  1. If |len1 - len2| > 1 → False.
  2. If equal length: count mismatches, return True if ≤1.
  3. If length differs by 1: use two pointers, allow skipping one char in the longer string; if more than one mismatch → False.
* **Complexity**: Time O(min(n,m)), Space O(1).

**Q4. 是否 at most K edit distances | At Most K Edit Distances**

**中文**

* **标准解法**：编辑距离动态规划 (Levenshtein Distance)。
  + dp[i][j] = 使前 i、j 个字符相等所需最小操作数。
  + 状态转移：插入 / 删除 / 替换。
* **优化**：
  + 剪枝：若 |len1 - len2| > K，直接 False。
  + 带宽 DP（Banded DP）：只在主对角线 ±K 范围计算，复杂度 O(K·min(n,m))。
  + 提前结束：若某行/列最小代价 > K，可提前退出。
* **复杂度**：
  + 普通 DP：时间 O(n·m)，空间 O(min(n,m))。
  + Banded DP：时间 O(K·min(n,m))，适合 K 较小的情况。

**English**

* **Standard solution**: Edit distance DP (Levenshtein).
  + dp[i][j] = min edits to make first i and j chars equal.
  + Transition: insert / delete / replace.
* **Optimizations**:
  + Prune: if |len1 - len2| > K, return False.
  + Banded DP: only compute within ±K diagonal band → O(K·min(n,m)).
  + Early stop: if current row’s min cost > K, terminate.
* **Complexity**:
  + Standard DP: Time O(n·m), Space O(min(n,m)).
  + Banded DP: Time O(K·min(n,m)), efficient when K is small.

⚡ **面试话术小贴士 | Interview Tips**

* **Q1**：先写朴素版保证正确，再提到 KMP 优化。
* **Q2**：把 \* 当“占位符”，拆成 token 顺序查找；是否以 \* 开头决定起点是否能为 0。
* **Q3**：分类讨论（长度差 >1 / 等长 / 差 1），双指针一次搞定。
* **Q4**：先说 DP，再补充剪枝 & 带宽优化，表现对复杂度的思考。

**🟢 第一问 & 第二问 —— 紧密相关**

* **第一问**：就是实现 indexOf，纯粹的字符串匹配。
* **第二问**：在第一问的基础上加了 \* 通配符，变成 **“支持通配符的 indexOf”**。  
  👉 所以第二问其实是第一问的**进阶版**，属于同一类考点：字符串搜索。  
  面试官一般是想看你先能不能写出基础的 indexOf，然后再扩展到更复杂的匹配场景。

**🔸 第三问 & 第四问 —— 突兀的转场**

* **第三问**：突然切换到 **编辑距离 (edit distance)**，问两个字符串是否在 ≤1 步编辑之内。
* **第四问**：推广到 ≤K 步编辑。  
  👉 这两个问题跟前两个的 **题型不直接相关**，更多是**题库里的高频题**，面试官拿出来继续考察你的算法功底。

**✅ 总结关系**

1. **Q1 & Q2**：一脉相承，Q2 是 Q1 的升级版。
2. **Q3 & Q4**：新话题（编辑距离），和前两问没啥联系。
3. 整场面试看下来，前半部分测的是**字符串搜索**，后半部分测的是**动态规划/双指针思路**。