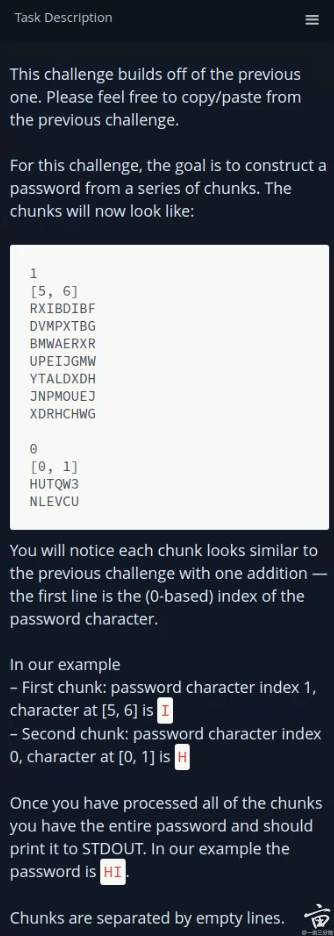
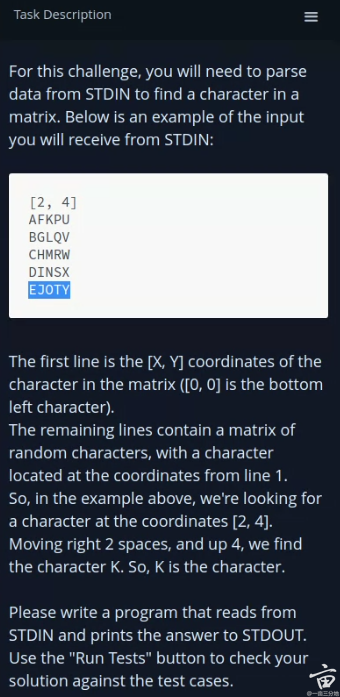
**Decode password**



**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。

**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）。

**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 ""。

**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。