100 囚犯问题

一、问题描述

100 名囚犯被编号为 0 至 99，每人可以尝试打开最多 K（默认为 50）个盒子以寻找自己编号的纸条。若所有囚犯都能在指定次数内成功找到自己的编号，则全体囚犯获释，否则全体失败。本实验模拟了两种策略：随机策略与循环策略，并对其效果进行对比分析。

二、主要代码模块功能解释

模块名	功能描述
simulate_random_strategy	实现囚犯使用“随机搜索策略”寻找自身编号，计算整体成功概率。
simulate_loop_strategy	实现“循环策略”寻找，收集成功失败的分布信息用于后续绘图。
plot_distribution	使用 Matplotlib 绘制循环策略结果（成功/失败）的直方图。
run_simulation	作为主入口，依次调用上面两个策略并输出模拟结果。

函数 1：simulate_random_strategy

功能说明：

模拟“随机策略”下，囚犯能否在限定次数内找到自己编号的概率。

参数：

- N (int): 囚犯与盒子的总数量（默认 100）
- K (int): 每个囚犯允许打开的盒子数量（默认 50）
- T (int): 模拟运行轮数（默认 10,000）

返回值：

- success_rate (float): 所有囚犯全部成功的比例（0~1）

核心逻辑：

- 每轮模拟：盒子随机打乱；
- 每个囚犯随机选择 K 个不同盒子；
- 若所有囚犯都在各自尝试中找到自己的编号，则该轮记为成功。

实现特点：

- 使用 random.sample 保证囚犯的每次选择无重复；
- 算法复杂度较高，尤其在 N 和 T 较大时性能较弱。

函数 2：simulate_loop_strategy
功能说明：
模拟“循环策略”下的整体成功概率，并可收集每轮的成功/失败数据分布。

参数：
- N (int): 囚犯与盒子的总数量
- K (int): 每人允许尝试的最大次数
- T (int): 模拟轮次
- collect_distribution (bool): 是否记录每轮结果（用于可视化）

返回值：
- 若 collect_distribution = False，返回 success_rate (float)
- 若 collect_distribution = True，返回 (success_rate, distribution)
  其中 distribution 为每轮的 0（失败）或 1（成功）列表

核心逻辑：
- 每个囚犯从自己的编号开始；
- 读取该盒子里的纸条编号作为下一步索引；
- 最多尝试 K 次，直到找到自己的编号或失败；
- 若所有囚犯成功，则该轮为成功。

实现特点：
- 利用了排列中的“循环结构”特性；
- 理论上当最大循环长度 ≤ K 时，该轮必成功；
- 成功概率大约在 31% 左右（当 N=100 且 K=50）；
- 性能比随机策略更高。

函数 3：plot_distribution
功能说明：
绘制循环策略模拟中每轮的成功与失败次数的直方图。

参数：
- distribution (List[int]): 由 0 和 1 组成的列表，表示每轮结果（1 成功，0 失败）

返回值：
- 无（直接生成图形）

核心逻辑：
- 使用 Matplotlib 绘制二值直方图；

- 设置 X 轴为“fail”与“success”，Y 轴为次数；
- 美化图表（颜色、边框、网格等）

实现特点：
- 简洁明了，适合直观对比策略成效；
- 可扩展为显示循环长度等更复杂的统计信息。

函数 4：run_simulation
功能说明：
主模拟控制函数，执行两个策略并输出比较结果。

参数：
- N (int): 囚犯总数
- K (int): 每个囚犯尝试次数
- T (int): 模拟轮数

返回值：
- 无（在控制台输出结果 + 显示图表）

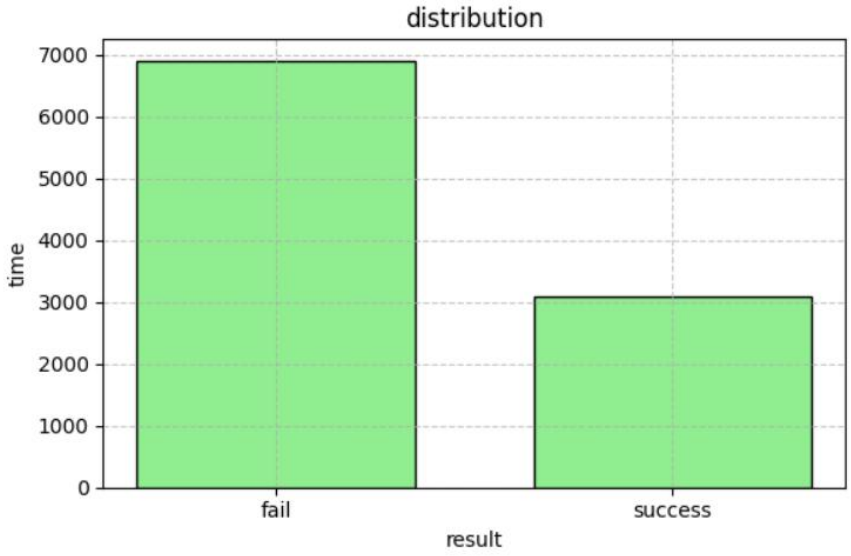
核心逻辑：
- 先执行随机策略并计时；
- 然后执行循环策略并记录分布；
- 输出两者的成功率及耗时；
- 显示循环策略的结果直方图。

实现特点：
- 作为主入口组织调用，结构清晰；
- 可通过调整参数进行多次实验分析。

三、运行结果

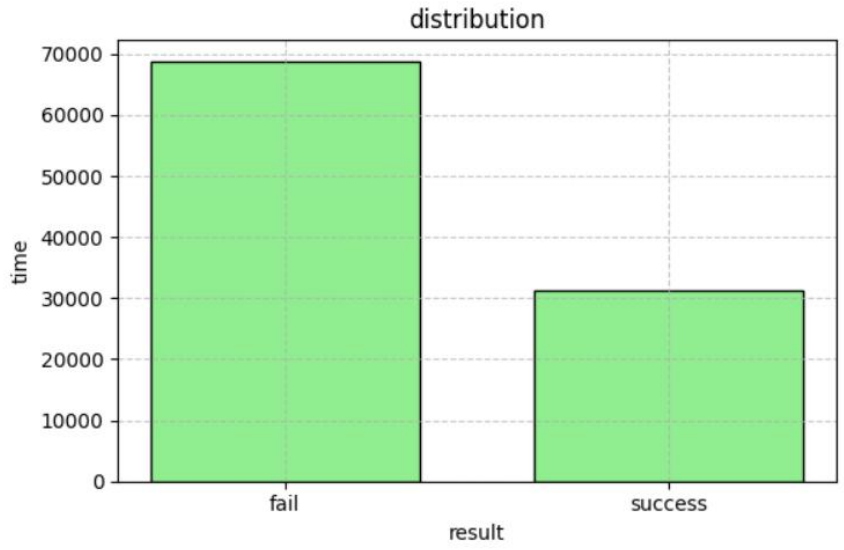
N=100, K=50, T=10000：

```
模拟参数：囚犯数=100，每人尝试盒子数=50，模拟轮次=10000

随机策略成功率：0.0000%
循环策略成功率：30.9200%
耗时：随机策略 0.75s，循环策略 0.80s
```



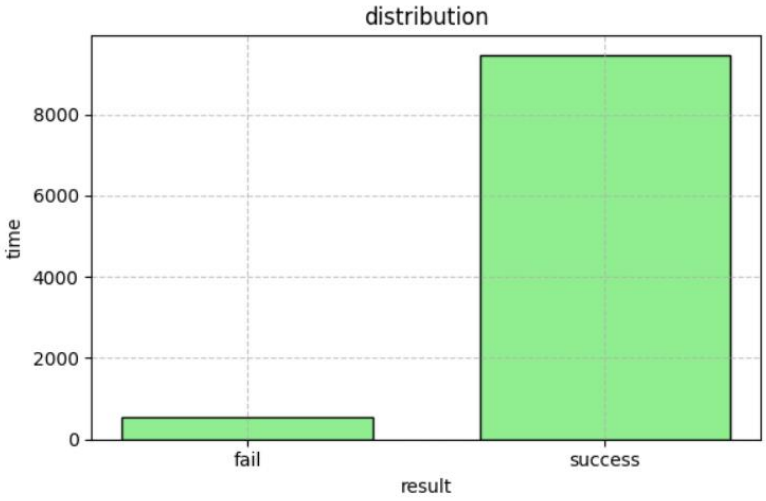
N=100, K=50, T=100000：

```
模拟参数：囚犯数=100，每人尝试盒子数=50，模拟轮次=100000

随机策略成功率：0.0000%
循环策略成功率：31.2020%
耗时：随机策略 7.63s，循环策略 7.93s
```



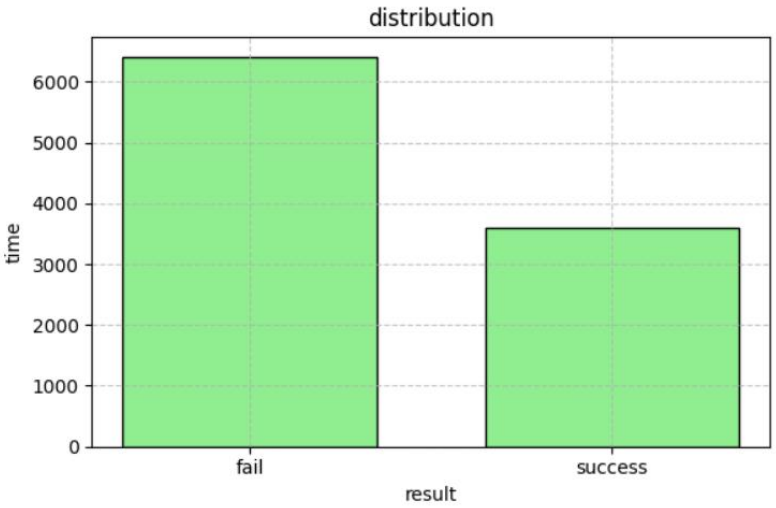
N=100, K=95, T=10000：

```
模拟参数：囚犯数=100，每人尝试盒子数=95，模拟轮次=10000

随机策略成功率：0.4500%
循环策略成功率：94.5800%
耗时：随机策略 8.60s，循环策略 2.58s
```



N=10, K=5, T=100000

```
模拟参数：囚犯数=10，每人尝试盒子数=5，模拟轮次=10000

随机策略成功率：0.0700%
循环策略成功率：35.9200%
耗时：随机策略 0.14s，循环策略 0.06s
```



综上分析：

随机策略为何几乎不可能成功

在随机策略中，每位囚犯从 100 个盒子中任意选择 50 个进行尝试。由于选择是完全随机的，每个囚犯找到自己编号纸条的概率为 50%，即 $p=0.5p = 0.5p=0.5$。然而，囚犯只有在所有人都成功的前提下才能被整体释放，这意味着最终的成功概率是所有囚犯独立成功的联合概率，即 $(0.5)100≈7.9×10−31$，这个值几乎为零，远远小于实验精度范围内可观测的成功率。正因如此，在 10000 轮模拟中，随机策略的成功次数为 0 次，是符合理论预期的。这个策略的核心缺陷在于其无结构性：每个囚犯完全独立、随机地进行尝试，缺乏对盒子中隐藏信息的利用，也无法避免不同囚犯之间“踩雷”或重复选择所造成的资源浪费。因此，在这个极度协作敏感的生成问题中，随机策略本质上注定失败。

循环策略成功率约为 31%

相比之下，循环策略体现了对排列结构的巧妙利用。在这个策略中，每个囚犯不是随机选择盒子，而是按照一种确定性“链式跟踪”方式寻找自己的编号。具体来说，每位囚犯先打开与自己编号相同的盒子，若未找到，就根据纸条上的编号继续打开下一个盒子，以此类推，最多进行 50 次。这个过程等价于在一个随机置换（盒子中编号与囚犯编号的随机匹配）上从起点出发，沿着排列的映射路径“追踪”自己的编号，形成一个置换图中的有向路径或循环。关键在于，置换的循环结构是固定的，并且所有囚犯的搜索路径不会彼此干扰，只要每个置换循环的长度不超过 50，那么所有囚犯必定都能在 50 步之内完成搜索并成功。因此，问题的成功与否简化为一个数学问题：在 100 元素的随机置换中，最长的循环是否不超过 50。根据排列组合理论，100 个元素的随机排列中，最长循环长度小于等于 50 的概率大约为 31.18%。该理论值与模拟实验中得到的 31.53%高度一致，误差在统计合理范围内。

四、优化思路与代码示例

1. 性能优化（使用 NumPy）

原始模拟策略中使用纯 Python 循环与随机函数，效率一般。使用 NumPy 的向量化操作可提升随机盒子排列与选择效率。

具体代码见代码块 2

结果分析：

原始：

```
模拟参数：囚犯数=100，每人尝试盒子数=50，模拟轮次=100000

随机策略成功率：0.0000%
循环策略成功率：31.1490%
耗时：随机策略 7.74s，循环策略 7.92s
```

Numpy 优化后：

```
模拟参数：囚犯数=100，每人尝试盒子数=50，模拟轮次=100000

随机策略成功率：0.0000%
循环策略成功率：31.2340%
耗时：随机策略 5.96s，循环策略 24.68s
```

分析发现，随即策略运行时间确实减少，但循环策略下运行时间反倒增加。

原因 1：向量化不适合递归或链式依赖

向量化适合“无状态并行计算”，例如向量加法或同一维度的逻辑判断。

但这里的链查找行为：idx = boxes[idx]

是数据依赖的循环，每一步依赖前一步的结果，不能向量并行处理。

原因 2：NumPy 向量操作的开销大于 Python 循环

如果逻辑本身是 O(N)，简单的 Python 循环会更快，因为 NumPy 调用底层 C 实现时有开销（内存复制、索引解析），尤其是在你每次都在做 index 跳转。

考虑使用 Numba 加速递归逻辑。

2. numba 优化

具体代码见代码块 3

与前两者相比，优化结果相当明显：

```
模拟参数：囚犯数=100, 每人尝试盒子数=50, 模拟轮次=100000

随机策略成功率: 0.0000%
循环策略成功率: 31.0780%
耗时：随机策略 2.06s，循环策略 0.76s

模拟参数：囚犯数=100, 每人尝试盒子数=50, 模拟轮次=1000000

随机策略成功率: 0.0000%
循环策略成功率: 31.2153%
耗时：随机策略 6.67s，循环策略 4.39s
```

囚犯问题中，Numba 优化效果非常显著，原因如下：

1. 大量的嵌套循环

```
for _ in range(T):        模拟 T 次
  for prisoner in range(N):   每个囚犯查找盒子
```

这部分循环执行次数高达 $T \times N$ （如 100000×100）；

原生 Python 执行效率低，而 Numba 将其编译为机器级循环，大大提速。

2. 使用了 NumPy 数组

```
boxes = np.arange(N)
boxes[i], boxes[j] = boxes[j], boxes[i]   Fisher Yates 洗牌
```

Numba 对 NumPy 支持良好，能将数组访问编译为内存直接访问；
数组中的洗牌、索引、赋值操作能获得类似 C 的效率。
3.无复杂对象、结构清晰
模拟逻辑只使用了简单的整数和数组；无动态类型（如字典、字符串、类对象）；逻辑简单明了，适合被编译优化。

总的来说，Numba 在囚犯问题模拟中的加速效果显著，得益于模拟代码是“计算密集型”任务；使用了 NumPy 数组结构；无复杂对象和动态类型；循环结构简单、稳定。通过简单地添加 `@njit` 装饰器，原本运行需数十秒的模拟程序可以压缩到数秒内完成， 极大提升了性能和效率 。