动态规划的优化

一、动态规划优化的分类

1. 状态转移过程的优化，不改变状态定义，使用一些特殊的数据结构或者算法专门优化转移过程
2. 程序实现的优化，例如：01背包问题。状态定义没有变、转移过程也没变。
3. 状态定义的优化，大量训练，才能培养出来的能力，从源头进行优化
4. 状态定义->源头，转移过程->过程，程序实现->结果

程序优化：01背包，钱币问题，滚动数组

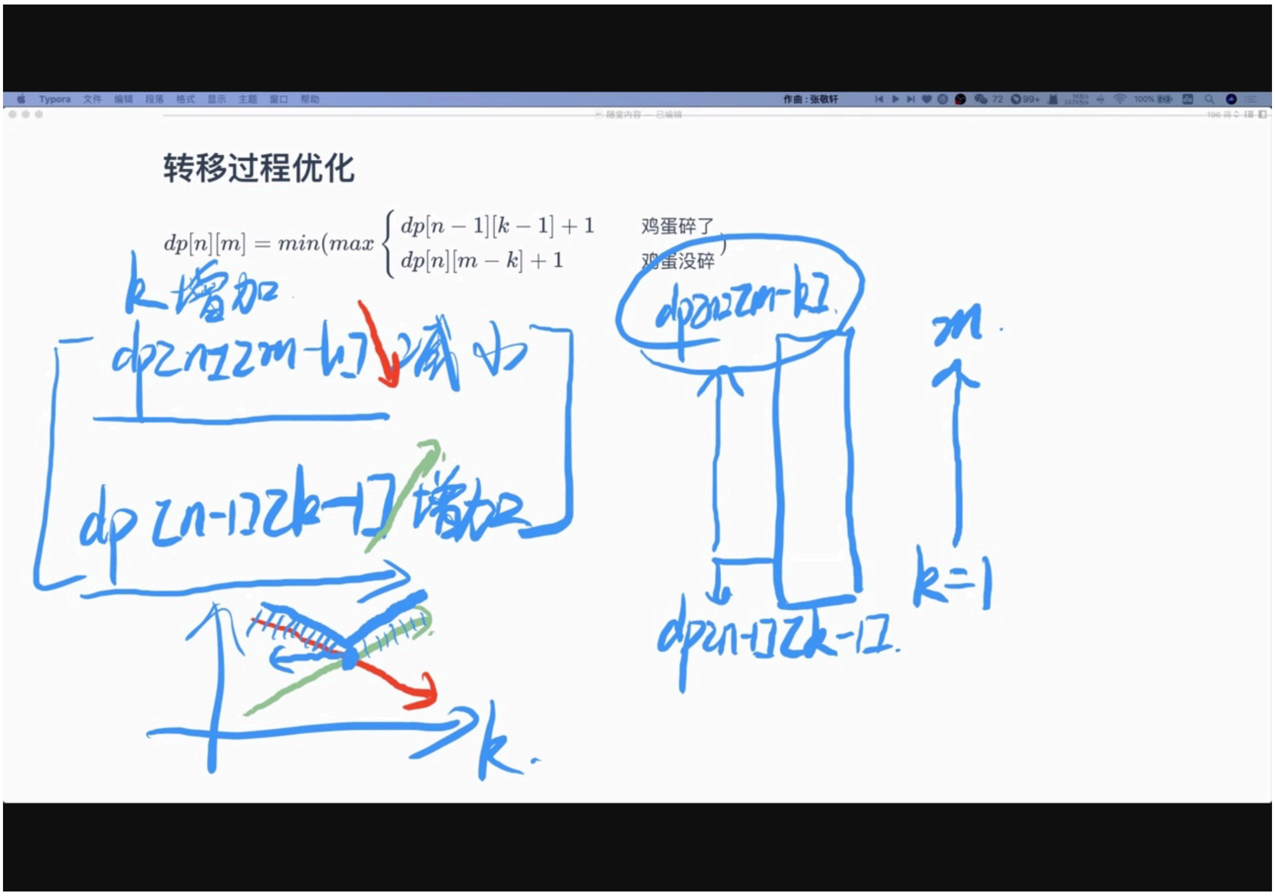
二、扔鸡蛋问题的优化

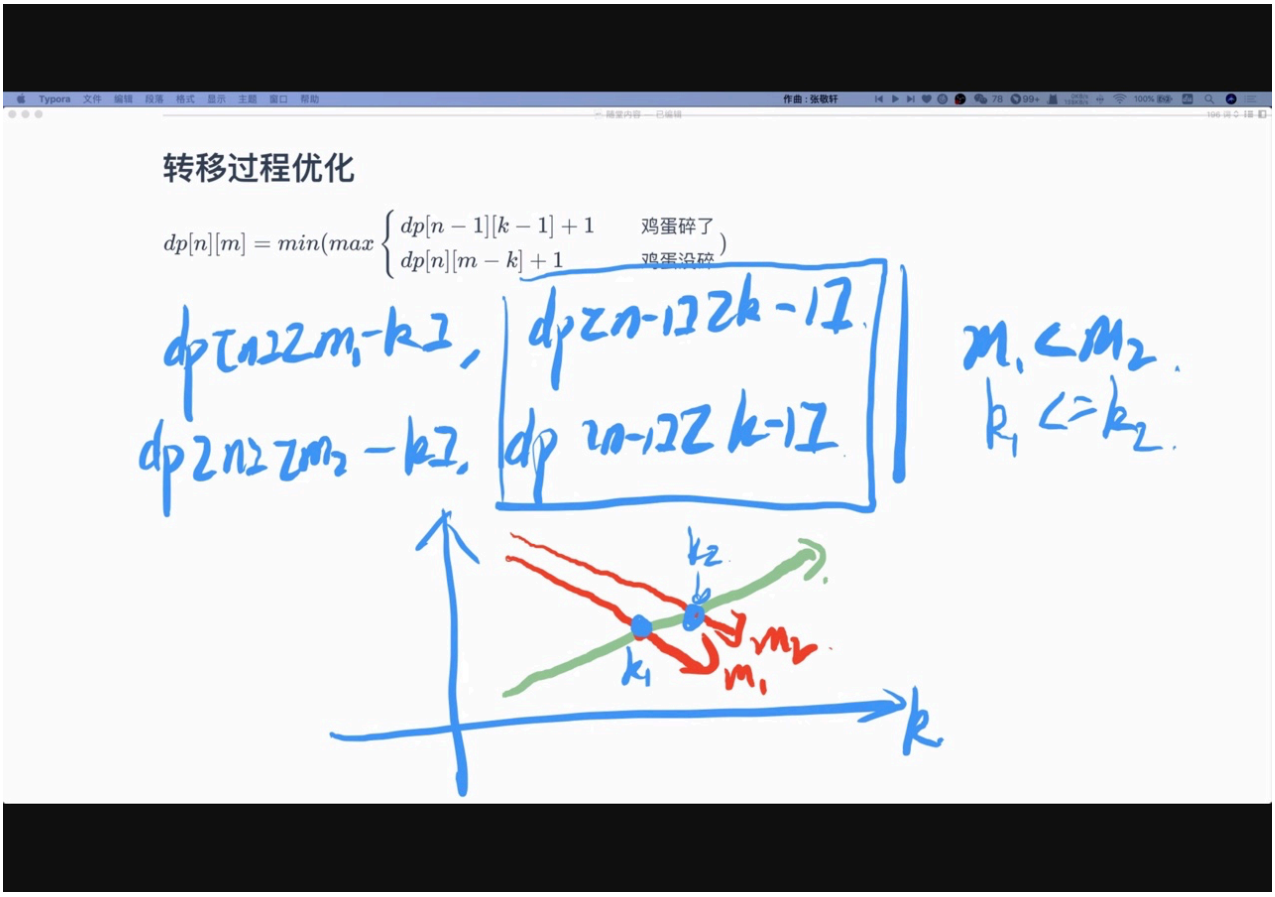
转移过程优化

$$dp[n][m] = min(max\begin{cases} dp[n-1][k-1]+1 & \text{鸡蛋碎了} \\ dp[n][m-k]+1 & \text{鸡蛋没碎} \end{cases})$$

通过观察 k 与 $dp[n-1][k-1]$与 $dp[n][m-k]$之间的关系，最优的转移 k 值，一定发生在两个函数的交点处

优化掉 min 以后，总体时间复杂度变成了 $O(n \times m)$


转移过程优化
$dp[n][m] = min(max\begin{cases} dp[n-1][k-1]+1 & 鸡蛋碎了 \\ dp[n][m-k]+1 & 鸡蛋没碎 \end{cases})$
k增加
dp[n][m-k] 减小
dp[n-1][k-1] 增加
dp[n][m-k]
m
k=1
dp[n-1][k-1]
k

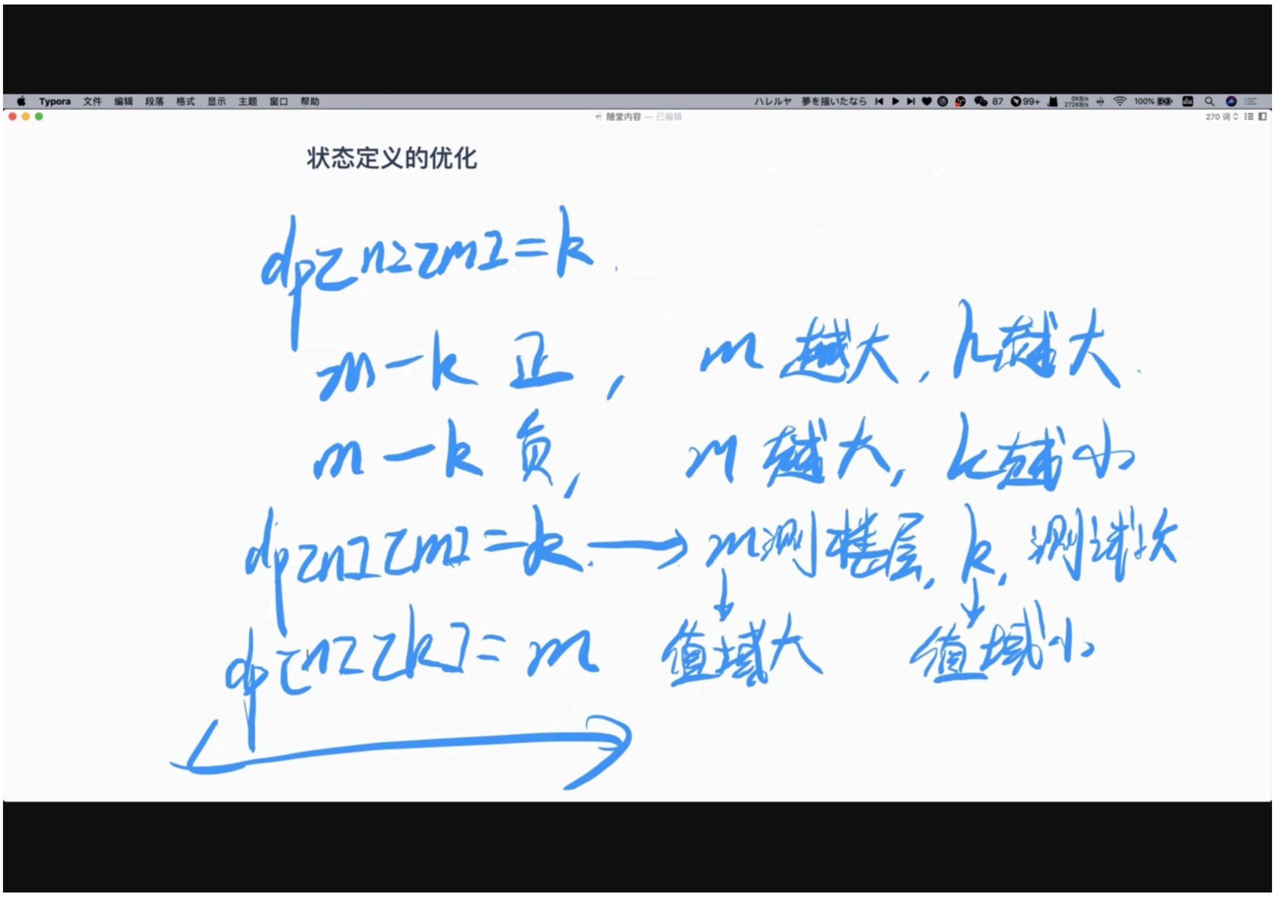

转移过程优化
$dp[n][m] = min(max\begin{cases} dp[n-1][k-1]+1 & 鸡蛋碎了 \\ dp[n][m-k]+1 & 鸡蛋没碎 \end{cases})$
$dp[n][m_1-k]$, $dp[n-1][k-1]$
$dp[n][m_2-k]$, $dp[n-1][k-1]$
$m_1 < m_2$
$k_1 <= k_2$
k_2
k_1
m_2
m_1
k

状态定义的优化

1. 原状态定义所需存储空间与 m 相关，m 值域大，所以存不下
2. 当发现某个自变量与因变量之间存在相关性的时候，两者即可对调
3. $dp[n][m]=k$ 重定义为 $dp[n][k]=m$，代表 n 个鸡蛋扔 k 次，最多测多少层楼
4. k 的值域小，当 n=2 时，$k \le \sqrt{2m}$

状态转移方程： $dp[n][k]=dp[n-1][k-1]+dp[n][k-1]+1$

本质上已经不是一个动态规划题目了，实际上变成了一个递推问题



三、多重背包的优化

二进制拆分法

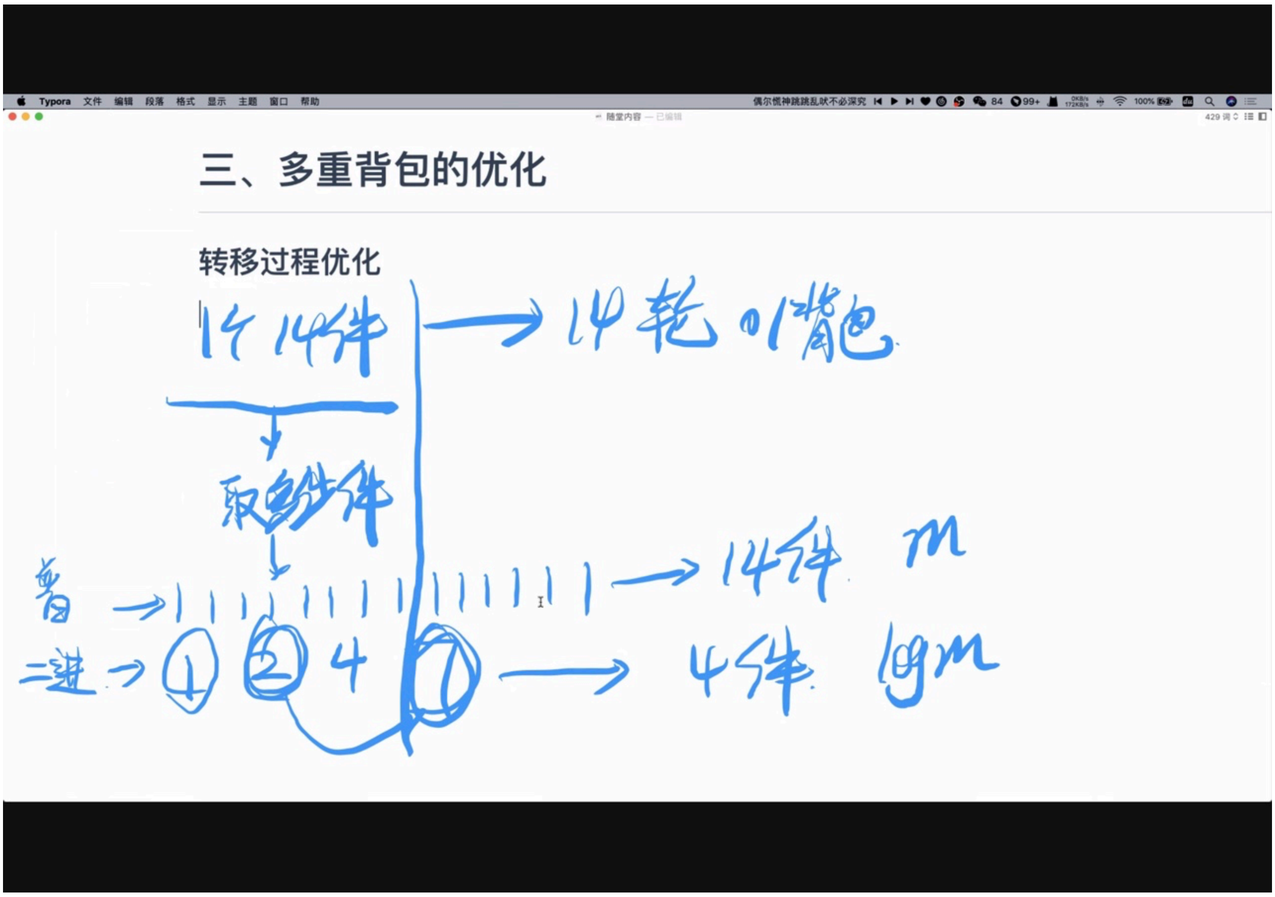
1. 本质上，对于某一类物品，我们具体要选择多少件，才是最优答案
2. 普通的单一拆分法，实际上只是想枚举某个物品选择 1--s 件的所有情况
3. 二进制拆分法可以达到相同的效果，拆分出来的物品数量会更少
4. 拿14举例，普通拆分法 14 份，二进制拆分法 4 份物品

时间复杂度： $O(nm\sum_{i=1}^{i=n} logs_i)$

最优时间复杂度：$O(nm)$，借助单调队列，后续再讲

01背包时间复杂度：$O(nm)$

完全背包时间复杂度：$O(nm)$



四、最长上升子序列

状态定义

$dp[i]$，代表以 i 位做为结尾的最长上升子序列的长度

状态转移

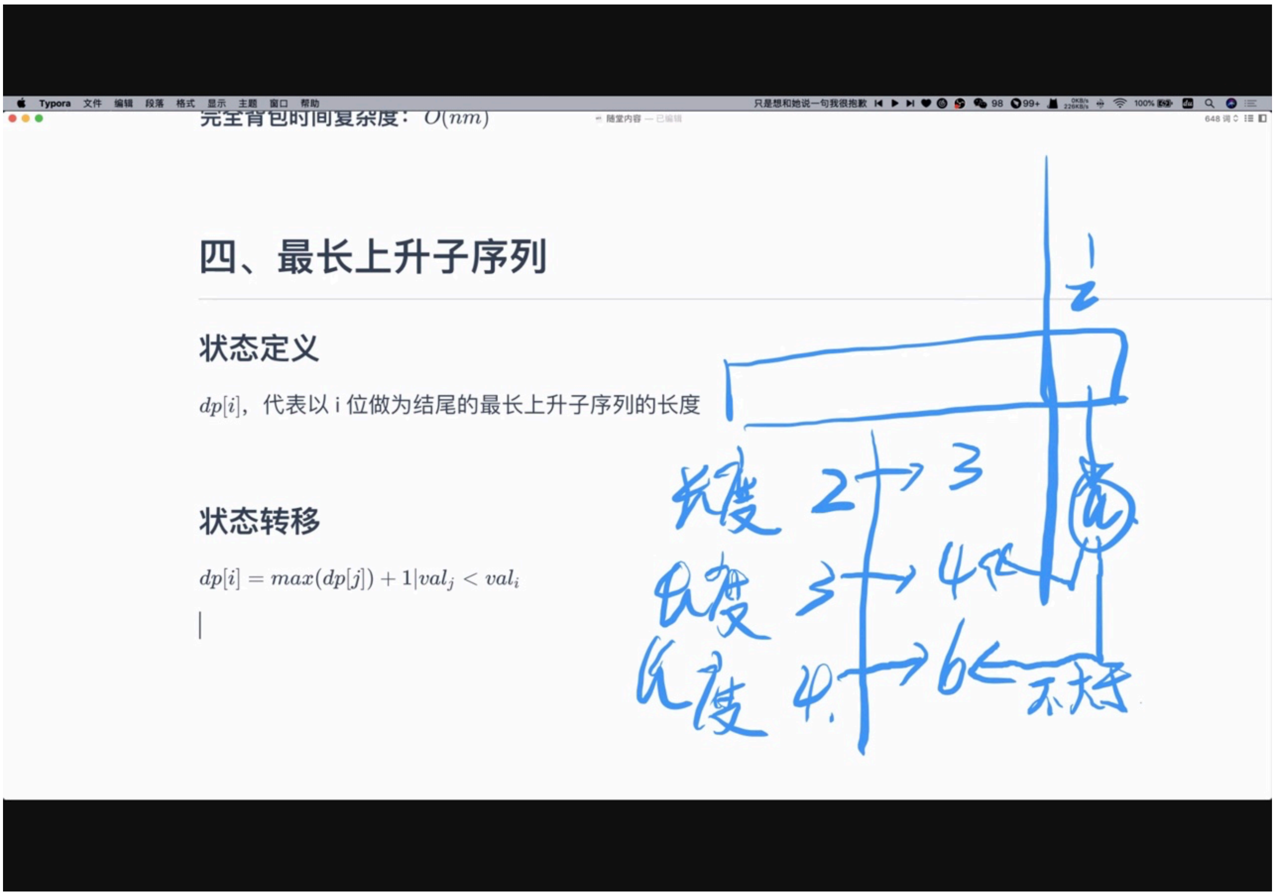
$dp[i] = max(dp[j]) + 1 | val_j < val_i$

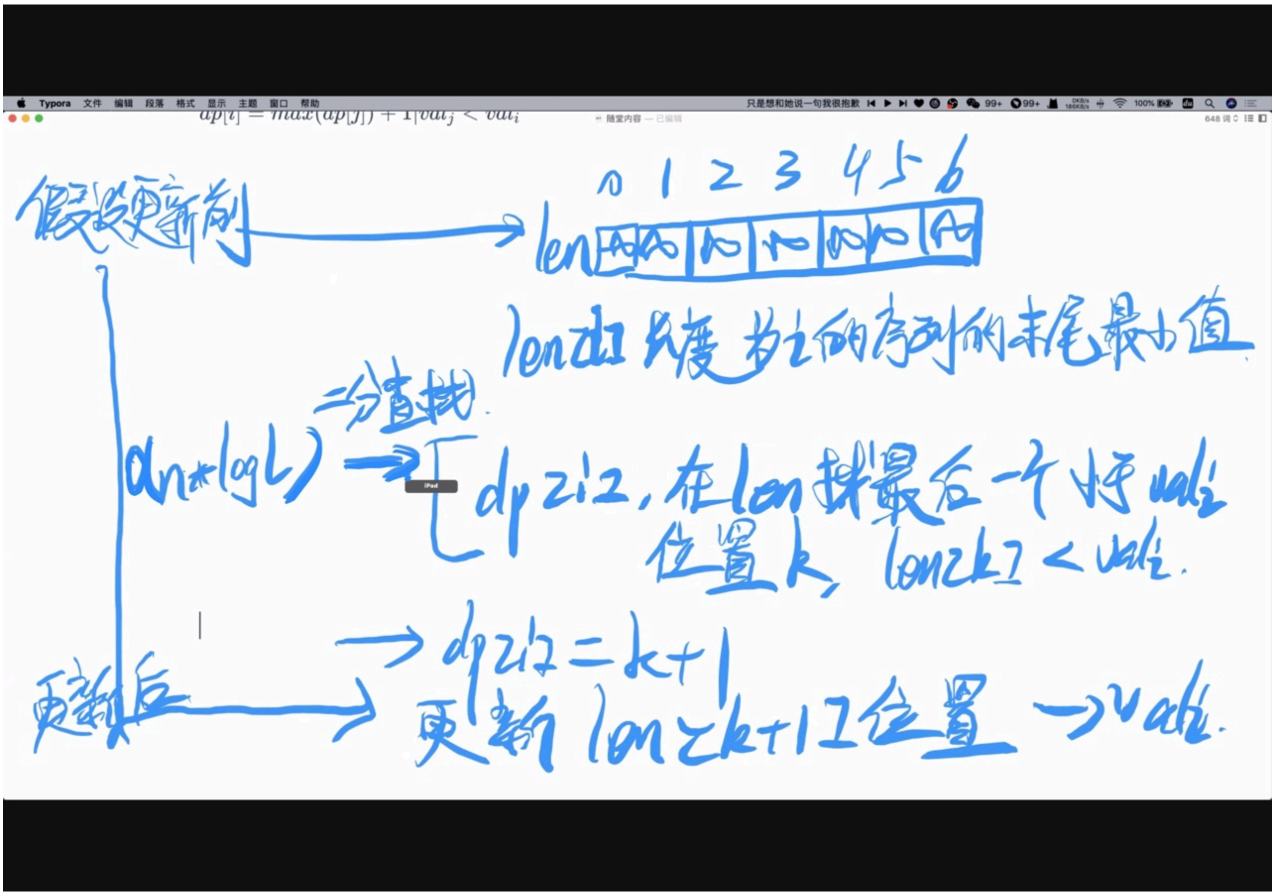
优化方式

1. 维护一个单调数组 len，len[i] 代表长度为 i 的序列，结尾最小值
2. $dp[i]$ 在转移的时候，在 len 数组中查找第一个 $len[k] >= val_i$ 的位置，$dp[i] = k$
3. 更新 $len[k] = val_i$
4. 需要明确，len 数组为什么是单调的

5. 证明过程：假设，更新前是单调的，更新以后，一定是单调的
6. 在 len 数组中查找位置 k，实际上就是二分算法搞定

时间复杂度：$O(nlogl)$





五、切割回文

提前处理得到 mark 数组，$mark[i]$ 存储的是所有以 i 位置做为结尾的回文串的起始坐标，在转移过程中，利用 mark 数组，就可以避免掉大量的无用循环遍历过程。

时间复杂度：$O(n+m)$，m 是字符串中回文串的数量