动态规划

适用于 LeetCode 中等题的

Koshiba / 2021.03

适用于 LeetCode 中等题的动态规划

- LeetCode 中等题 2 例
- 总结 2 道中等题的解题规律
- 引出“动态规划”

983. 最低票价

- 在一个火车旅行很受欢迎的国度，你提前一年计划了一些火车旅行。在接下来的一年里，你要旅行的日子将以一个名为 days 的数组给出。每一项是一个从 1 到 365 的整数。
- 火车票有 3 种不同的销售方式：
  - 一张为期 1 天的通行证售价为 costs[0] 美元；
  - 一张为期 7 天的通行证售价为 costs[1] 美元；
  - 一张为期 30 的通行证售价为 costs[2] 美元。
- 通行证允许数天无限制的旅行。 例如，如果我们在第 2 天获得一张为期 7 天的通行证，那么我们可以连着旅行 7 天：第 2 天、第 3 天、第 4 天、第 5 天、第 6 天、第 7 天和第 8 天。
- 返回你想要完成在给定的列表 days 中列出的每一天的旅行所需要的最低消费。

983. 最低票价

- 输入： days = [1,4,6,7,8,20], costs = [2,7,15]
- 输出： 11
- 解释：
  - 在第 1 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 1 天生效。
  - 在第 3 天，你花了 costs[1] = $7 买了一张为期 7 天的通行证，它将在第 3, 4, ..., 9 天生效。
  - 在第 20 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 20 天生效。
  - 你总共花了 $11，并完成了你计划的每一天旅行。

983. 最低票价

- 输入：days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
- 输出：17
- 解释：
  - 在第 1 天，你花了 costs[2] = $15 买了一张为期 30 天的通行证，它将在第 1, 2, ..., 30 天生效。
  - 在第 31 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 31 天生效。
  - 你总共花了 $17，并完成了你计划的每一天旅行。

983. 最低票价 思路

- 出门旅行的日期是随机的，票价也是随机的
  - 总体来说只能**暴力枚举**，并找出总票价最低的组合
  - 但旅行的日期（可排列为）严格增序，可使用剪枝
    - 例如第 1 日买了 30 日券后，随后的 29 日如有旅行应**不买票**，而不需要再加算购买 1日券、7 日券、30 日券。
    - 因为如果多买票，则总票价一定不是最低的

983. 最低票价 思路

- 从前往后遍历要旅行的日期，假设今天是第 i 天，今天
  - 买了 1 日券，花费为 costs[0]
  - 买了 7 日券，花费为 costs[1]，且随后 6 天不用买票
  - 买了 30 日券，花费为 costs[2]，且随后 29 天不用买票
- 对应的总开销为
  - C1 = costs[0] + solve(i + 1)
  - C2 = costs[1] + solve(j)，其中 j 为第一个大于 i + 6 的旅行日期
    - 例如旅行日期为 [1, **3**, 5, 7, 9, 11, 13]，今天是第 **3** 天，则下一个应买票的日子是 11
  - C3 = costs[2] + solve(k)，其中 k 为第一个大于 i + 29 的旅行日期
- 那么很简单，最低开销就是 min([C1, C2, C3])

983. 最低票价 思路

- 从前往后遍历要旅行的日期，假设今天是第 i 天，今天
  - 买了 1 日券，花费为 costs[0]
  - 买了 7 日券，花费为 costs[1]，且随后 6 天不用买票
  - 买了 30 日券，花费为 costs[2]，且随后 29 天不用买票
- 对应的总开销为

运行一下，然后速度就爆炸了

  - C1 = costs[0] + solve(i + 1)
  - C2 = costs[1] + solve(j)，其中 j 为第一个大于 i + 6 的旅行日期
    - 例如旅行日期为 [1, **3**, 5, 7, 9, 11, 13]，今天是第 **3** 天，则下一个应买票的日子是 11
  - C3 = costs[2] + solve(k)，其中 k 为第一个大于 i + 29 的旅行日期
- 那么很简单，最低开销就是 min([C1, C2, C3])

983. 最低票价 思路

- ~~那么很简单，最低开销就是 min([C1, C2, C3])~~
- 速度怎么就爆炸了？设 i 为旅行的日期，solve(i) 会被调用多少次呢?
- 总是买 1 日券时
  - 对于 solve (1)：它会调 solve (2)，solve(2) 会调 solve(3), solve(3) 会调 solve(4)
  - 对于 solve(2)：它会调 solve (3)，solve(3) 会调 solve(4)
  - 对于 solve (3)：它会调 solve(4)
  - solve(4) 已经被调了 4 次，可以想象，solve(30)被调了 30 次。
- 总是买 7 日券时：
  - ……
- 混着买时：
  - ……

983. 最低票价 思路

- ~~那么很简单，最低开销就是 min([C1, C2, C3])~~
- 速度怎么就爆炸了？设 i 为旅行的日期，solve(i) 会被调用多少次呢?
- 问题找到了，解决方法也就找到了
  - **计算过的（solve(i)）就别重复计算了呗**

- 今天是第 i 天呢
  - 我懒，我先查查 solve(i) 有没有计算过，计算过的话直接读取结果
  - 啊，没有。我得亲自去计算 solve(i)。顺便算完了我把结果存起来以后用

- 然后速度就正常了

983. 最低票价 思路

```
var price1 = costs[0];
if (memoization[pos] != 0) return memoization[pos];
var remainingCost1 = solve(pos + 1, days, costs);
var totalCost1 = price1 + remainingCost1;

if (memoization[pos] != 0) return memoization[pos];
var price7 = costs[1];
var posCopy7 = pos;
while (posCopy7 < daysLength && days[posCopy7] < today + 7) { posCopy7++; }
var remainingCost7 = solve(posCopy7, days, costs);
var totalCost7 = price7 + remainingCost7;
……

var cheapest = Math.min(Math.min(totalCost1, totalCost7), totalCost30);
memoization[pos] = cheapest;
return cheapest;
```

days 与 costs 从不改变，可不当参数传递

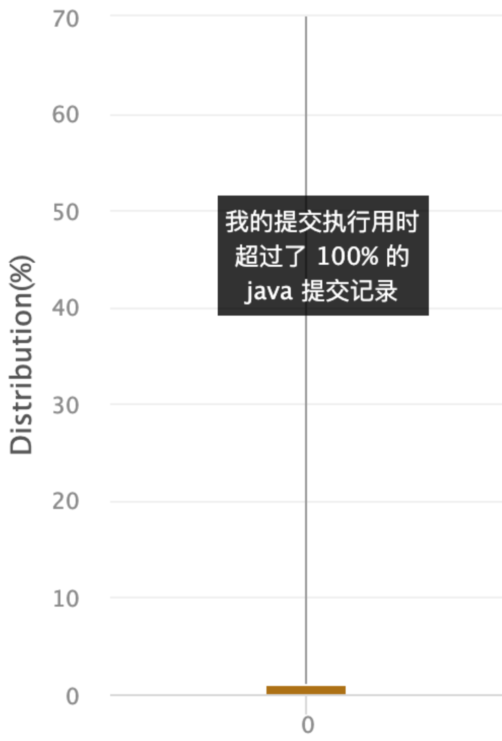
983. 最低票价 成绩还行

提交记录

66 / 66 个通过测试用例
执行用时: **0 ms**
内存消耗: **36.1 MB**

提交时间	提交结果	运行时间	内存消耗	语言
23 天前	通过	0 ms	36.1 MB	Java
23 天前	通过	0 ms	36.2 MB	Java
23 天前	超出时间限制	N/A	N/A	Java

执行用时分布图表



309. 最佳买卖股票时机含冷冻期

- 给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。
- 设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：
  - 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
  - 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

- 示例：
  - 输入: [1,2,3,0,2]；输出: 3
  - 解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

309. 最佳买卖股票时机含冷冻期 思路

- 套路一下：今天是第 i 天，我今天可以做的决策是1 买，2 卖，3 观望
- 这题的“状态”稍微复杂一些，今天是第 i 天
  - 如果我之前没买股票、持有现金，我今天只可以买或者观望，我没办法卖
    - 二择一，选出利润较高者
  - 如果我之前买入股票，我今天只能卖出或者观望
    - 二择一，算出利润较高者
  - 如果我昨天卖出股票，我今天只能观望，没法买卖
    - 没得选
  - 因此需要一个枚举类型的函数参数来指示今天的状态（也可用 field 表示）

309. 最佳买卖股票时机含冷冻期 思路

- 剩下的不用我说大家也知道了
  - 速度爆炸了
- 解决思路是一样的
  - 在调用 solve(状态, 日期) 前先查查有没有求解过，有就直接读取
  - 如果没有求解过，则求解并存储该结果

- ~~然后速度虽然没爆炸，但还是没别人的快~~
  - ~~感觉是没有做一些剪枝~~

小小总结

- 用递归的思路，给出一个解
- 记录下已经求解过的结果，不重复求解

- 注意：用递归的思路时，要求解的诸多子问题与原来的大问题是相似的
- 我 LeetCode 刷的不多，暂时在中等题里没遇到子问题与原问题不相似的情况

- 我不怎么用循环，就不讲如何用（嵌套）循环自底向上求解了。
  - 反正科目一时间复杂度对了就能过，比别人快也不加分，圈复杂度大了还扣分
  - ~~对了我还没过~~

教科书式的动态规划

- 动态规划是运筹学的一个分支，是求解决策过程（decision process）最优化的数学方法。
- 20 世纪 50 年代初 R. E. Bellman 等人在研究多阶段决策过程（multistep decision process）的优化问题时，提出了著名的最优性原理（principle of optimality），把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程 优化问题的新方法–动态规划。1957 年出版了他的名著《Dynamic Programming》，这 是该领域的第一本著作。
-

教科书式的动态规划——适用条件（来自维基百科）

- **最优子结构性质。**如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。
- **无后效性。**即子问题的解一旦确定，就不再改变，不受在这之后、包含它的更大的问题的求解决策影响。
- **子问题重叠性质。**子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次。~~然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率，降低了时间复杂度。~~
- 举例：买车票的例子。

教科书式的动态规划——适用条件（来自维基百科）

- 理论部分要不以后再讲吧，本人还没掌握
- 大家可以去考试了（大概？），中等难度题的动态规划=送分