2nd: Dynamic Programming

手写题

Step Problem

1. Modeling

- 如果青蛙跳上第n级台阶，它可以从第n-1级台阶跳1级，也可以从第n-2级台阶跳2级。因此，跳上n级台阶的方法数是跳上n-1级和跳上n-2级台阶的方法数之和。
- 该问题可以抽象为斐波那契数列问题，其递推公式为：`f(n) = f(n-1) + f(n-2)`。

1. Algorithm description
2. 定义一个数组`dp`，其中`dp[i]`表示青蛙跳到第i级台阶的方法数。
3. 初始条件为`dp[1] = 1`，`dp[2] = 2`。
4. 对于`n >= 3`，`dp[i] = dp[i-1] + dp[i-2]`。
5. 最终`dp[n]`即为青蛙跳到第n级台阶的所有可能方法数。
6. Time complexity

时间复杂度为 **O(n)**，因为我们只需遍历一次从第3级到第n级台阶。

1. Space complexity

空间复杂度为 **O(n)**，因为我们使用了一个大小为`n+1`的数组来存储中间结果。

Buy!

1. Modeling

- 经典的01背包问题。
- 01背包问题的决策：
  - 不选，去考虑下一个；
  - 选，背包容量减掉相应重量，总值加上相应价值。
- 01背包问题的状态转移方程：（j表示花费预算，i表示当前物品）

- $ f[j]=max(f[j],f[j−v[i]]+p[i]) $这里 v[i] 是物品的价格，p[i]是物品的重要程度。

- 目标状态：f[n]

1. Algorithm description
2. 初始化：
   - 记预算钱数为n，物品数为m，每个物品的价格为 *v*，重要程度为 *p*。

     i

     i
   - 读入输入时更新item_w[M]、item_c[M]；
   - 初始化f[N]为0。
3. 计算过程：两重循环，第一重循环遍历m个物品，第二重循环从预算为n开始递减。
4. 边界条件：注意选择物品时需要判断是否还有足够的预算。
5. Time complexity

$O(MN)$

分析：状态转移过程中两重循环

1. Space complexity

$O(N)$

分析：预算n远大于物品数m，f[N]占用最大空间

Counting

(1) Modeling

- 该题只求a的数量，则可用计数类动态规划解决。
- 令$f[i]$表示数组长度为i的方案数，则状态转移时其由两部分构成：

  （设新加入的区间长度为$S[j]$）

  1. $[1, i-S[j]]$，这部分方案数为$f[i-S[j]]$；
  2. $[i-S[j]+1, i]$，这部分长度为$S[j]$，设其方案数为v_j（表示S可组成的所有长度为$S[j]$、最小值也为$S[j]$的数组的数量）。

两部分相互独立，因此总方案数为二者方案数之积。

- 求解v_j：由定义知第二部分区间内最小值为$S[j]$，故数组由集合$T_j = \{x \in S | x \geq S[j]\}$中的元素组成。

  可以发现，$v_j =$方案数(从T_j中取出必须包含$S[j]$的$S[j]$个元素)。

  即$v_j =$方案数(从T_j中取出$S[j]$个元素)$-$方案数(从T_j中取出不包含$S[j]$的$S[j]$个元素)。

  易得：前者方案数为$|T_j|^{S[j]}$，后者方案数为$|T_j \setminus \{S[j]\}|^{S[j]}$。

  由S的定义可得$|T_j| = m - j + 1$，$|T_j \setminus \{S[j]\}| = m - j$。

  则最终$v_j = (m - j + 1)^{S[j]} - (m - j)^{S[j]}$

(2) Algorithm description

1. 初始化$f[...] = 0$
2. 提前从1到m枚举j，用快速幂快速计算v_j
3. 边界条件：$f[0] = 1$
4. 状态转移：

   for i = 1 to n

   　　for j = 1 to m，且$S[j] \leq i$

   　　　　$f[i]$ += $f[i - S[j]]$ * v_j
5. 最终结果：$f[n]$

(3) Time complexity

快速幂的时间复杂度为$O(logS[j])$，由S的性质，即步骤2的时间复杂度为$O(mlogS[m])$。

第4步的时间复杂度为$O(nm)$。

则总体时间复杂度为$O(nm + mlogS[m])$

(4) Space complexity

共涉及变量$v[1...m], f[1...n]$，因此总体空间复杂度为$O(n + m)$。

Ex. Buy! Buy! Buy!

1. Modeling
    - 本质是带有依赖的01背包问题。
    - 01背包问题的决策：
        - 不选，去考虑下一个；
        - 选，背包容量减掉相应重量，总值加上相应价值。
    - 01背包问题的状态转移方程：（j表示花费预算，i表示当前物品）
      f[j] = max(f[j],f[j-w[i]]+c[i])
    - 本题的决策：
        - 不选，去考虑下一个；
        - 选且只选这个主件；
        - 选这个主件，并且选附件1（需要判断是否可以进行）；
        - 选这个主件，并且选附件2（需要判断是否可以进行）；
        - 选这个主件，并且选附件1和附件2（需要判断是否可以进行）。
    - 令main_item_w数组表示某个主件的费用，main_item_c数组表示某个主件的价值。
    - 令二维数组acc_item_w表示某个附件的费用，acc_item_c表示某个附件的价值，数组中[i][0]表示这个主件i的附件数量，只能等于0、1或2，[i][1]或[i][2]的值代表以i为主件的附件1或者附件2的相关信息。
    - 本题的状态转移方程：
        - 不选附件：
          f[j] = max(f[j],f[j - main_item_w[i]] + main_item_c[i])
        - 选附件1：
          f[j] = max(f[j],f[j - main_item_w[i] - acc_item_w[i][1]] + main_item_c[i] + acc_item_c[i][1])
        - 选附件2：
          f[j] = max(f[j],f[j - main_item_w[i] - acc_item_w[i][2]] + main_item_c[i] + acc_item_c[i][2])

- 选附件1和附件2：
  f[j] = max(f[j],f[j - main_item_w[i] - acc_item_w[i][1] - acc_item_w[i][2]] + main_item_c[i] + acc_item_c[i][1] + acc_item_c[i][2])

- 目标状态：f[n]

2. Algorithm description
   - 初始化：
     - 记预算钱数为n，物品数为m，每个物品的价格为 v_i，重要程度为 p_i，对应的主件为 q_i，如果 $q_i = 0$，则表示这个物品就是主件。
     - 读入输入时更新main_item_w[M]、main_item_c[M]、acc_item_w[M][3]和acc_item_c[M][3]；
     - 初始化f[N]为0。
   - 计算过程：两重循环，第一重循环遍历m个物品，第二重循环从预算为n开始递减。
   - 边界条件：注意选择附件时需要判断是否还有足够的预算。
     - 选附件1：
       if (j >= main_item_w[i] + annex_item_w[i][1])
     - 选附件2：
       if (j >= main_item_w[i] + annex_item_w[i][2])
     - 选附件1和附件2：
       if (j >= main_item_w[i] + annex_item_w[i][1] + annex_item_w[i][2])
3. Time complexity
   - $O(MN)$
   - 分析：状态转移过程中两重循环
4. Space complexity
   - $O(N)$
   - 分析：预算n远大于物品数m，f[N]占用最大空间

OJ题

Optimal Display Rental

1. Modeling

问题是找到三个广告牌，它们的字体大小严格递增 $s_i < s_j < s_k$，并且租金成本最小。可以使用动态规划解决此问题

定义一个动态规划数组 `dp[i][l]`，表示以广告牌 i 结尾的长度为 l 的严格递增子序列的最小总成本。这里的 l=1,2,3对应的是长度为 1（单个广告牌）、长度为 2 和长度为 3 的严格递增子序列。

1. Algorithm description
2. 初始化 $dp[i][1] = c[i]$,表示每个广告牌的初始租金。
3. 状态转移方程：对于每个广告牌*i*,遍历所有广告牌*j* (当*j* < *i*且*s*[*j*] < *s*[*i*]时):
   ·当
   l = 2或*l* = 3时：

   $$dp[i][l] \;=\; min\,(dp[i][l], dp[j][l-1] + c[i])$$
4. 最后找到所有$dp[i][3]$中的最小值。如果没有找到有效的子序列，则输出 -1。
5. Time complexity

该算法主要时间复杂度来源于嵌套的双重循环：

- 外层循环 遍历广告牌*i*,总共执行了$n \;-\; 1$次
- 内层循环 对于每个广告牌*i*,我们从0遍历到$i \;-\;$ 1,并且在每次循环中判断$s[j] \;<\; s[i]$,如果条件满足，则进行状态转移

综上，算法的时间复杂度为：$O(n2)$

1. Space complexity

该算法使用了一个二维动态规划数组 $dp[i][l]$，大小为 n×3，存储每个广告牌对应长度为 1、2、3 的最小租金子序列。

由于每个广告牌有 3 个状态需要存储，所以 `dp` 的空间复杂度是 *O*(*n*)

Sheep Transport Problem

1. Modeling

- 问题需要找到分批次运输绵羊过河的最优策略，使得总的渡河时间最短；
- 可设$dp[j]$为运输j头绵羊渡河所需要的最短时间，$sum[i]$表示一次运输i头绵羊渡河的时间，可得到状态转移方程$dp[j] = min(dp[j], dp[j-i] + sum[i])$。

2. Algorithm description
   a. 要计算运输N头绵羊渡河的最短时间，初始化$dp[i]$为一个大数INF；
   b. 设$sum[i]$为一次运输i头绵羊过河所需要的时间，因为题目给出了每增加一头羊所需要加上的渡河时间，故可通过计算前缀和的方式计算出$sum[i]$；
   c. 如果要多次渡河，每次运送绵羊后都需要返回，故在$sum[i]$上加上Bob和木筏来回所需的时间$2m$（最后一次渡河后不需要回来，故最后需减去m）；
   d. 得到状态转移方程：$dp[j] = min(dp[j], dp[j-i] + sum[i])$。
3. Time complexity

- $sum[i]$计算的复杂度为$O(n)$，二重循环计算状态转移方程的时间复杂度为$O(n^2)$，故总的时间复杂度为$O(n^2)$。

4. Space complexity

- 存储动态规划数组dp数组，运输时间sum数组，$weight$数组所需要的空间都为$O(n)$，因此总的空间复杂度为$O(n)$。

Ant Foraging

（1）Modeling

- 问题可简化为找到两条从起点到终点的不相交路径，使得所经过格子上的食物总量最大
- 定义$dp[k][i][j]$表示蚂蚁从起点分别在两条路径上经过k步后，分别处于位置（i，$k+2-i$）和(j，$k+2-j$)时能收集到的最大食物量，动态规划的解法将模拟两条路径，并确保两条路径不重叠。

（2）Algorithm description

1、初始状态为$dp[0][1][1] = 0$

2、状态转移方程：

对于每一步k，计算 $dp[k][i][j]$：

$$dp[k][i][j] = max(dp[k-1][i-1][j-1], dp[k-1][i][j-1], dp[k-1][i-1][j], dp[k-1][i][j]) + a[i][k+2-i] + a[j][k+2-j]$$

（3）Time complexity

该算法的主要循环是基于三维动态规划表 $dp[k][i][j]$的更新，其中：

- 外层循环k遍历了从 1 到 $m+n-2$，总共遍历$m+n-2$次。
- 中层循环
  j遍历从 1 到 m，但每次最多遍历到 $k+1$，因此遍历次数与k 的大小有关，最多为 m。
- 内层循环
  i遍历从$j+1$到 m，与中层循环j类似，最多也是m次。

对于每一个 k，最多执行$O(m^2)$次的状态转移。
综上，算法的时间复杂度为：
$O((m+n)*m^2)$

（4）Space complexity

该算法使用了一个三维动态规划数组 $dp[k][i][j]$，其维度为$dp[k][i][j]$：

- k的取值范围为 0 到 $m+n-2$，因此有$m+n-2$个状态。
- i 和j 的最大值分别为 m。

因此，动态规划表的空间复杂度为：$O((m+n)*m^2)$

Extra Problem

1. Modeling

定义一个动态规划数组 dp[i][j]，表示用前 i 种花来摆放恰好 j 盆花的方案数。

- 边界条件：dp[0][0] = 1，表示没有花时摆放0盆的方案数为1。
- 递推关系：对于第 i 种花，我们可以摆放 0 到 a[i-1] 盆花。因此，dp[i][j] 可以从 dp[i-1][j-k] 转移而来，其中 0 ≤ k ≤ a[i-1] 且 j-k ≥ 0。

$dp[i][i] = \sum_{k=0}^{min(j,a[i-1])}dp[i - 1][j-k]$

为了优化上面的求和，可以使用前缀和优化：

$dp[i][j] = dp[i][j-1] + dp[i-1][j] - dp[i-1][j-a[i-1]-1]$ (if $j - a[i-1] - 1 \geq 0$)

1. Algorithm description
2. 初始化 dp[0][0] = 1，其他 dp[0][j] = 0。
3. 对于每种花 i 从 1 到 n：
   对于每个可能的总盆数 j 从 0 到 m： 计算 dp[i][j] 使用前缀和优化。
4. 结果为 dp[n][m]，并对答案取模 1e6+7。
5. Time complexity

$O(n \times m)$, 因为我们需要填充一个 $n \times m$ 的表格。

1. Space complexity

$O(n \times m)$, 用于存储dp数组。