计算机学院《算法设计与分析》
(2021 年秋季学期)
第二次作业参考答案

1 数组填充问题 (20 分)

李华有一个长度为 n 的整数数组 a，这个数组有以下性质：

1. 这个数组的所有元素之和为 3 的倍数。
2. 这个数组的每个元素 a_i 都满足 $a_i \in [l, r]$

李华忘记了这个数组的元素，请你设计一个高效的算法，帮助他找出有多少个满足条件的数组，并分析该算法的时间复杂度。

例如，长度为 $n = 2$，满足区间 $l = 1, r = 3$ 的数组，包括 $[1, 2], [2, 1], [3, 3]$，则答案为 3。

解：

1. 状态设计

通过在长度为 k 的数组后增加一个元素，可以构造出长度为 $k+1$ 的数组。任何数模 3 的结果只可能为 $0, 1$ 和 2。因此可以将数组长度、数组元素之和模 3 的结果作为动态规划状态。

令 $A_i =< a_1, \cdots, a_i >$ 表示序列 a 的前 i 个元素，定义状态 $dp[i][j]$ 考虑前 i 个元素之和模 3 为 j 的数组个数。

2. 状态转移

考虑区间 $[l, r]$ 中的整数，其中模 3 为 k 的数的个数为 m_k。对状态 $dp[i][j]$ 而言，若第 i 位填充的数模 3 为 k，则可从 $dp[i-1, (j-k)\%3]$ 转移过来。因此，枚举所有可能的 k 即可得到转移方程如下：

$$dp[i][j] = \sum_{k=0}^{2} dp[i-1][(j-k)\%3] * m_k$$

3. 边界条件与目标状态

起始状态为长度为 0 的数组，元素之和模 3 为 0，即 $dp[0][0] = 1$，$dp[0][1] = dp[0][2] = 0$。

最终答案为长度为 n，数组元素和模 3 为 0 的数组，即 $dp[n][0]$。

4. 时间复杂度分析

总计状态数为 $3n$，单次转移复杂度为 $O(1)$，因此总复杂度为 $O(n)$。

伪代码见 Algorithm1。

2 最长递增子序列问题 (20 分)

递增子序列是指：从原序列中按顺序挑选出某些元素组成一个新序列，并且该新序列中的任意一个元素均大于该元素之前的所有元素。例如，对于序列 $< 5, 24, 8, 17, 12, 45 >$，该序列的两个递增子序列为 $< 5, 8, 12, 45 >$ 和 $< 5, 8, 17, 45 >$，并且可以验证它们也是原序列最长的递增子序列。请设计算法来求出一个包含 n 个元素的序列 $A =< a_1, a_2, \cdots, a_n >$ 中的最长递增子序列，并分析该算法的时间复杂度。

解：

1. 求解思路

令 $X =< x_1, \ldots, x_n >$ 为给定的包含 n 个元素的序列，我们需要找到序列 X 的最长递增子序列。

Algorithm 1 $fill\_array(n, l, r)$

Input:

数组长度 n，数组元素范围 l, r。

Output:

满足条件的数组的数量。

初始化数组 $dp[0..n][0..2], m[0..2]$
$dp[0][0] \leftarrow 1$
$dp[0][1], dp[0][2] \leftarrow 0$
$m[0] \leftarrow r/3 - (l-1)/3$
$m[1] \leftarrow (r-1)/3 - (l-2)/3$
$m[2] \leftarrow (r-2)/3 - (l-3)/3$
for $i : 1 \rightarrow n$ **do**
  for $j : 0 \rightarrow 2$ **do**
    $dp[i][j] \leftarrow dp[i][j] + dp[i-1][0] * m[j\%3]$
    $dp[i][j] \leftarrow dp[i][j] + dp[i-1][1] * m[(j+2)\%3]$
    $dp[i][j] \leftarrow dp[i][j] + dp[i-1][2] * m[(j+1)\%3]$
  end for
end for
return $dp[n][0]$

我们首先给出求解最长递增子序列长度的算法，之后再介绍如何找到该递增的上升子序列。

2. 状态设计

令 $X_i =< x_1, \ldots, x_i >$ 表示序列 X 的前 i 个元素，定义状态 $c[i]$ 表示以 x_i 为结尾的最长递增子序列的长度，显然整个序列的最长递增子序列的长度为 $\max\limits_{1\leq i\leq n} c[i]$。

3. 状态转移

考虑 $c[i]$ 的更新过程，若存在某个 $x_r < x_i (1 \leq r < i)$，那么以 x_r 为结尾的最长递增子序列加上 x_i 就构成了一个新的最长递增子序列，因此我们要选择满足上述条件同时 $c[r]$ 最大的 r 来更新 $c[i]$。据此可以写出如下递归式：

$$c[i] = \begin{cases} 1 & i = 1 \\ 1 & x_r \geq x_i \forall 1 \leq r < i \\ \max\limits_{1\leq r<i, x_r<x_i} c[r] + 1 & i > 1 \end{cases}$$

4. 边界条件

递归式的终止条件基于如下事实：以 x_i 结尾的仅包含一个数字的最长递增子序列就是它本身。

5. 求解原问题与记录方案

按照递增的顺序对每个 i 依次计算 $c[i]$ 的值，在计算完 c 数组后，其中的最大元素即是序列 X 的最长递增子序列的长度。

为了输出所求出的最长递增子序列，我们在计算 $c[i]$ 时，需要同时记录 $r[i] = \arg\max\limits_{1\leq r<i, x_r<x_i} c[r]$。令 $c[k] = \max\limits_{1\leq i\leq n} c[i]$，那么 x_k 就是所求最长递增子序列的最后一个元素，之后我们依次找出 $x_rk, x_{r[r[k]]}$，将这些元素逆序输出即为原序列 X 的最长递增子序列。

6. 时间复杂度分析

时间复杂度分析：在计算 $c[i]$ 时，需要花费 $O(i)$ 的时间，因此，总的运行时间为 $O(\sum i) = O(n^2)$。之后需要 $O(n)$ 的时间来确定最长递增子序列的每个元素，因此，总的时间复杂度为 $O(n^2)$。

伪代码见 Algorithm2。

3 硬币问题 (20 分)

给定 n 枚硬币 (n 为奇数)，编号为 $1, 2, \cdots, n$。投掷第 i 枚硬币时有 p_i 的概率正面朝上，有 $1 - p_i$ 的概率反面朝上。

设计算法求解投掷这 n 枚硬币，其中正面朝上的硬币数量多于反面朝上的概率，并分析该算法的时间复杂度。

Algorithm 2 $LIS(A[1..n])$

Input:

数组 $X[1..n]$。

Output:

最长递增子序列。

初始化数组 $c[1..n], r[1..n]$
$len \leftarrow 0$
$pos \leftarrow -1$
for $i : 1 \rightarrow n$ **do**
  $c[i] \leftarrow 1$
  $r[i] \leftarrow -1$
  for $j : 1 \rightarrow i-1$ **do**
    if $x[j] < x[i]$ and $c[j]+1 > c[i]$ **then**
      $r[i] \leftarrow j$
      $c[i] \leftarrow c[j]+1$
    end if
  end for
  if $len < c[i]$ **then**
    $len \leftarrow c[i]$
    $pos \leftarrow i$
  end if
end for
初始化数组 $ans[1..len]$
$p \leftarrow 1$
while $pos \neq -1$ **do**
  $ans[p] \leftarrow pos$
  $p \leftarrow p+1$
  $pos \leftarrow r[pos]$
end while
$ans \leftarrow reverse(ans)$//答案反向
return ans

例如给定 $n=3$ 枚硬币，其正面朝上的概率分别为 $p_1=0.3, p_2=0.6, p_3=0.8$。有下述四种情况正面朝上的硬币数量多于反面朝上:

1. 三枚硬币同时朝上，概率为 $0.3 \times 0.6 \times 0.8 = 0.144$。
2. 第一枚硬币朝下，第二枚硬币朝上，第三枚硬币朝上，概率为 $0.7 \times 0.6 \times 0.8 = 0.336$。
3. 第一枚硬币朝上，第二枚硬币朝下，第三枚硬币朝上，概率为 $0.3 \times 0.4 \times 0.8 = 0.096$。
4. 第一枚硬币朝上，第二枚硬币朝上，第三枚硬币朝下，概率为 $0.3 \times 0.6 \times 0.2 = 0.036$。

故总概率为 $0.144+0.336+0.096+0.036=0.612$。

1. 状态设计

用二维的状态 $dp[i][j]$ 表示，当前已经考虑了前 i 枚硬币，其中有 $j(j \leq i)$ 枚硬币朝上的概率。

2. 状态转移

则有如下转移方程:

$$dp[i][j] = \begin{cases} 1 & i=0 \\ dp[i-1][j]*(1-p[i]) & i>0 \text{ and } j=0 \\ dp[i-1][j-1]*p[i] & i>0 \text{ and } j=i \\ dp[i-1][j]*(1-p[i])+dp[i-1][j]*p[i] & i>0 \text{ and } 0<j<i \end{cases}$$

这可以理解为两种情形

1. 第 i 枚硬币正面朝上，这时对 $dp[i][j](j \geq 1)$ 的贡献为 $dp[i-1][j-1] \times p[i]$
2. 第 i 枚硬币反面朝上，这时对 $dp[i][j](j < i)$ 的贡献为 $dp[i-1][j] \times (1-p[i])$

3. 边界条件与目标状态

边界情况如状态转移方程所述，在 $i=0$ 时，没有硬币朝上的概率为 1。

问题的答案为 $\sum_{j=\frac{n+1}{2}}^{n} dp[n][j]$。

4. 时间复杂度分析

该动态规划的状态数为 $O(n^2)$ 级别，每个状态需要 $O(1)$ 的时间转移，故总时间复杂度为 $T(n)=O(n^2)$。伪代码如 Algorithm (3) 所示。

Algorithm 3 $coin(n, p[1..n])$

Input:

n 枚硬币投掷后正面朝上的概率数组 $p[1..n]$

Output:

投掷 n 枚硬币，正面朝上的硬币数多于反面朝上的概率。

```
dp[0][0] ← 1
for i : 1 → n do
    for j : 0 → i do
        dp[i][j] ← 0
        if j > 0 then
            dp[i][j] ← dp[i][j] + dp[i−1][j−1] ∗ p[i]
        end if
        if j < i then
            dp[i][j] ← dp[i][j] + dp[i−1][j] ∗ (1 − p[i])
        end if
    end for
end for
return Σ_{j=(n+1)/2}^{n} dp[n][j]
```

---

4 鲜花组合问题 (20 分)

花店共有 n 种不同颜色的花，其中第 i 种库存有 a_i 枝，现要从中选出 m 枝花组成一束鲜花。请设计算法计算有多少种组合一束花的方案，并分析该算法时间复杂度。(每种花数量均相同的方案算一种方案)

解:

1. 状态设计

该问题类似背包问题，考虑每种花为一个物品，最多可以购买 a_i 件，问题转化为总共购买 m 件的前提下，有多少种购买方案。

状态 $dp[i][j]$ 表示考虑前 i 种花，已经购买了 j 枝花的方案数。

2. 状态转移

在上一状态，即只考虑了前 $i-1$ 种花的情况下，枚举购买多少枝当前种类的花，进行转移。转移方程为:

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

3. 边界条件与目标状态

起始条件为 $dp[0][0]=1$。

最终答案为 $dp[n][m]$，考虑全部 n 种花，恰好购买了 m 支的方案数量。

4. 时间复杂度分析

每次转移需要枚举当前种类的花购买的枝数，复杂度为 $O(m)$，状态数为 $n \times m$，因此复杂度为 $O(n \times m^2)$。

5. DP 转移优化

观察转移方程可以发现，其形式与前缀和相似，因此可以使用前缀和优化转移方程。

在转移之前计算

$$sum[j] = dp[i-1][j] + sum[j-1]$$

转移方程可以改写为

$$dp[i][j] = sum[j] - sum[j - \min(a_i, j) - 1]$$

每次转移复杂度为 $O(1)$，因此总复杂度降低为 $O(n \times m)$。

伪代码见 Algorithm4。

Algorithm 4 $flower(A[1..n])$

Input:

鲜花库存数量 $A[1..n]$，选择数量 m。

Output:

鲜花组合方案数量。

$dp[1..n][1..m]$
$sum[1..m]$
$dp[0][0] \leftarrow 1$
for $i : 1 \to n$ **do**
  $sum[0] \leftarrow dp[i-1][0]$
  for $j : 1 \to m$ **do**
    $sum[j] \leftarrow sum[j-1] + dp[i-1][j]$
  end for
  for $j : 0 \to m$ **do**
    $dp[i][j] \leftarrow sum[j] - sum[j - min(a[i], j) - 1]$
  end for
end for
return $dp[n][m]$

5 最大分值问题 (20 分)

给定一个包含 n 个整数的序列 $a_1, a_2, \ldots, a_n$，对其中任意一段连续区间 $a_i..a_j$，其分值为

$$(\sum_{t=i}^{j} a_t)\%p$$

符号 % 表示取余运算符，可以认为 p 远小于 n。

现请你设计算法计算将其分为 k 段 (每段至少包含 1 个元素) 后分值和的最大值，并分析该算法的时间复杂度。

例如，将 $3, 4, 7, 2$ 分为 3 段，模数为 $p = 10$，则可将其分为 $(3, 4), (7), (2)$ 这三段，其分值和为 $(3+4)\%10 + 7\%10 + 2\%10 = 16$。

解:

1. 状态设计

记 $val(i, j) = (\sum_{t=i}^{j} a_t)\%p$，令 $f[i][j]$ 表示将前 i 个数分为 j 段可获得的最大分值。

2. 状态转移

枚举第 j 段的起始位置 $t+1$，若最后一段是由 $a[t+1..i]$ 构成，则这一段对答案的贡献为 $val(t+1, i)$，而 $a[1..t]$ 应被分为 $j-1$ 段，其最大分值为 $f[t][j-1]$。

故递归式如下:

$$f[i][j] = \max_{t=0}^{i-1}\{f[t][j-1] + val(t+1, i)\}$$

3. 边界条件与目标状态

初始化仅需将所有的 $f[i][j]$ 置为 0。

目标状态为 $f[n][k]$。

4. 时间复杂度分析

其状态数为 $O(nk)$，每次转移的复杂度为 $O(n)$，故总的时间复杂度为 $O(n^2k)$。

5. 状态转移的改进

考虑如何进行优化，通过观察可发现，上述公式中，$val(t+1,j)$ 可能的取值只有 p 种。这意味着我们可以将 $f[t][j-1]$ 按照其对应的 $val(t+1,i)$ 的不同取值进行分组，对每一组预统计出其最大值，之后仅需花费 $O(p)$ 的时间进行转移。

具体来说，记 $sum[i] = \sum_{t=1}^{i} a_t$。则 $val(t+1,i)$ 可改写为

$$val(t+1,i) = (sum[i] - sum[t])\%p$$

由此可看出，在计算状态 $f[i][j]$ 时，i 已经确定，那么 $val(t+1,i)$ 的取值仅和 $sum[t]\%p$ 相关。因此，可将所有 $f[t][j-1]$ 根据 $sum[t]\%p$ 分组，并统计每组的最大值 (记 $g[x][j-1]$ 表示所有满足 $sum[t]\%p = x$ 的状态 $f[t][j-1]$ 的最大值)。之后需将每一组的最大值 $g[x][j-1]$ 加上这一组对应的 val 值。根据公式

$$val(t+1,i) = (sum[i] - sum[t])\%p$$

可知，对所有满足 $sum[t]\%p = x$ 的 t，其对应的 $val(t+1,i)$ 均为 $(sum[i]-x)\%p$。

根据上述分析，递归式可写为：

$$f[i][j] = \max_{x=0}^{p-1}\{g[x][j-1] + (sum[i] - x + p)\%p\}$$

其中，

$$g[x][j-1] = \max_{t<i, sum[t]=x} f[t][j-1]$$

6. 改进后的时间复杂度分析

其状态数为 $O(nk)$，每次转移的复杂度为 $O(p)$，故总的时间复杂度为 $O(npk)$。

算法伪代码如 Algorithm 5 所示。

Algorithm 5 $Feasible(a[1..n], k, p)$

$sum[0] \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do**
  $sum[i] \leftarrow sum[i-1] + a[i]$;
end for
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** k **do**
    for $t \leftarrow 0$ **to** $p-1$ **do**
      $f[i][j] \leftarrow \max\{f[i][j], g[t][j-1] + (sum[i] - t + p)\%p\}$;
      $g[sum[i]\%p][j] \leftarrow \max\{g[sum[i]\%p][j], f[i][j]\}$;
    end for
  end for
end for
return $f[n][k]$;