高等理工学院《算法设计与分析》
(2020 年秋季学期)

第二次作业参考答案

1 字符串编码问题 (20 分)

假设将小写英文字符使用如下策略编码：

1. a 编码成 1;
2. b 编码成 2;
3. $\cdots$
4. z 编码成 26。

$$dp[i] = \begin{cases} a[i-2]+1 & a[i-1] \neq 0 \text{ and } a[i-1]\times 10 + a[i] \le 26 \\ a[i-1] & \text{otherwise} \end{cases}$$

给定一个长度为 n 的数字串 α，试设计算法求解有多少种小写英文字符串可以编码成该串，并分析该算法的时间复杂度。

例如长度为 $n = 3$ 的数字串 $\alpha = 126$ 可以由"abf","az","lf" 三种小写英文字符串编码而来。

解:

1. 状态设计

设 $T(i)$ 表示有多少种小写英文字符串可以表示 $\alpha[1 \sim i]$ 这个前缀字符串。

2. 状态转移

有两种边界情况:

1. $T(0)$ 此时对应还没有使用任何一个数字，故方案为 1
2. $T(1)$ 此时仅使用一个数字，$\alpha[1] \in 1, \cdots, 9$ 时存在 1 个方案。

对于 $i \geq 2$ 而言，需要分别考虑如下两种情况是否存在贡献:

1. $\alpha[i] \in 1, \cdots, 9$ 此时可以考虑第 i 个数字单独转换成小写英文字母，共有 $T(i-1)$ 种情况
2. $\alpha[i-1 \sim i] \in 10, \cdots, 26$ 此时可以考虑第 $i-1$ 个数字和第 i 个数字组成的两位数对应一个小写英文字母。共有 $T(i-2)$ 种情况

则转移如下

$$T(i) = \begin{cases} 1 & i = 0 \\ 1 & i = 1 \text{ and } \alpha[i] \in \{1, \cdots, 9\} \\ T(i-1) & i \geq 2 \text{ and } \alpha[i] \in \{1, \cdots, 9\} \text{ and } \alpha[i-1 \sim i] \notin \{10, \cdots, 26\} \\ T(i-2) & i \geq 2 \text{ and } \alpha[i] \notin 1, \cdots, 9 \text{ and } \alpha[i-1 \sim i] \in \{10, \cdots, 26\} \\ T(i-1) + T(i-2) & i \geq 2 \text{ and } \alpha[i] \in 1, \cdots, 9 \text{ and } \alpha[i-1 \sim i] \in \{10, \cdots, 26\} \\ 0 & \text{otherwise} \end{cases}$$

3. 时间复杂度分析

注意到问题的状态数为 $O(n)$ 级别的，而每次转移的复杂度是 $O(1)$ 的。故时间复杂度为 $T(n) = O(n)$。伪代码如下 (Algorithm 1):

Algorithm 1 $count(\alpha[1..n])$

Input:

一个长度为 n 的字符串, $\alpha[1..n]$;

Output:

有多少种小写字符串可以编码成该串。

$T[0] \leftarrow 0$
for $i : 1 \rightarrow n$ **do**
  $T(i) \leftarrow 0$
  if $\alpha[i] \in \{1, \cdots, 9\}$ **then**
    $T(i) \leftarrow T(i) + T(i-1)$
  end if
  if $i \geq 2 \cap \alpha[i-1 \sim i] \in \{10, \cdots, 26\}$ **then**
    $T(i) \leftarrow T(i) + T(i-2)$
  end if
end for
return $T(n)$

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。

$T(i) = \begin{cases} 1 & i = 1 \\ & i > 1 \end{cases}$

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。

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 $x_r k, x_{r[r[k]]}$，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 $O(n^2)$。

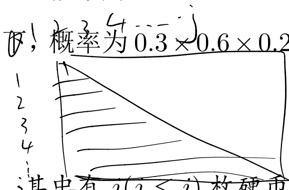
3 硬币问题 (20 分)

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。

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 $j (j \le i)$ 枚硬币朝上的概率。

$\boxed{i \geq j}$

i=0, 1,
i=1,

2. 状态转移

则有如下转移方程: $dp[i][j] = dp[i-1][j] \cdot (1-p[i])$

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

→ i=0
→ 直负
→ 直正
→ 能正能负

这可以理解为两种情形

1. 第 i 枚硬币正面朝上，这时对 $dp[i][j] (j \ge 1)$ 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

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。

4. 时间复杂度分析

故原问题的答案为 $\sum_{j=\frac{n+1}{2}}^{n} dp[n][j]$，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 (2) 所示。

4 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。

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。

Algorithm 2 $coin(n, p[1..n])$

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。

$dp[0][0] \leftarrow 1$
for $i : 1 \to n$ **do**
  for $j : 0 \to i$ **do**
    $dp[i][j] \leftarrow 0$
    if $j > 0$ **then**
      $dp[i][j] \leftarrow dp[i][j] + dp[i-1][j-1] * p[i]$
    end if
    if $j < i$ **then**
      $dp[i][j] \leftarrow dp[i][j] + dp[i-1][j] * (1 - p[i])$
    end if
  end for
end for
return $\sum_{j=\frac{n+1}{2}}^{n} dp[n][j]$

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

初始化仅需将所有的 $f[i][j]$ 置为 0。

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 3 所示。

Algorithm 3 $Feasible(a[1..n], k, p)$

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

5 箱子问题 (20 分)

给定 n 种箱子 $a_1, \cdots, a_n$，第 i 种箱子 a_i 可表示为 $h_i \times w_i \times d_i$ 的长方体。请用这些箱子搭建一个尽可能高的塔：如果一个箱子 A 要水平的放在另一个箱子 B 上，那么要求箱子 A 底面的长和宽都严格小于箱子 B。可以任意旋转箱子，每种箱子可以用任意次。

设计一个算法求出一个建塔方案使得该塔的高度最高，并分析该算法的时间复杂度。

例如给定 $n = 1$ 种箱子，其可表示为 $3 \times 4 \times 5$ 的长方体，建塔方案如下：

1. 最底层，放置一个以 4×5 为底面的箱子，该箱子高度为 3;
2. 第二层，放置一个以 3×4 为底面的箱子，该箱子高度为 5。

此时该塔高度最高，为 $3 + 5 = 8$。

如下的建塔方案不合法：

1. 最底层，放置一个以 4×5 为底面的箱子，该箱子高度为 3;
2. 第二层，放置一个以 3×5 为底面的箱子，此时底面的长为 5，不满足条件。

解:

1. 确定决策顺序

此题可以看做是最长上升子序列问题的一个变形，先要注意到如下两个事实:

1. 每种箱子最多使用 3 次，分别是 $h \times w \times d, w \times h \times d, d \times h \times w$ 即以 $w \times d, h \times d, h \times w$ 作为底面各尝试一次。
2. 只有底面积比当前箱子大的箱子，才可能允许在其之上放置当前箱子。

2. 状态设计

故我们，可以先将箱子个数按照三种底面的情形扩充至 $3n$ 的情形，再按照底面积大小从小到大排序为 $b_1, \cdots b_{3n}$(b_i 箱子对应的长、宽、高分别记做 d'_i, w'_i, h'_i)。$dp[i]$ 表示最后一个选择的箱子为 b_i 的最高塔高。

3. 状态转移

采用如下转移:

$$dp[i] = \max \begin{cases} h'_i \\ dp[j] + h'_i \quad (d'_j > d'_i \cap w'_j > w'_i) \end{cases}$$

这是因为在考虑阶段 i 时，可以枚举之前所有考虑过的阶段 $j(j < i)$，只要其满足摆放条件就可以转移给状态 i。

而状态 i 的边界条件就是仅放了 b_i 这一个箱子的情形。

4. 时间复杂度分析

故状态数为 $O(n)$ 级别，每个状态的转移要考虑其之前的所有状态，故时间复杂度为 $T(n) = \sum_{i=1}^{3n} O(i) = O(n^2)$。算法伪代码如 Algorithm 4 所示。

Algorithm 4 $boxing(n, h[1..n], w[1..n], d[1..n])$

Input:
n 种箱子，第 i 种的规格为 $h_i \times w_i \times d_i$

Output:
使得塔最高的建塔方案

将 $w_i \times d_i, h_i \times d_i, h_i \times w_i$ 分别作为底面扩充成数组 $b[1..3n]$（并约束每个箱子的长大于等于宽）
按照底面积从大到小对 $b[1..3n]$ 数组排序
for $i : 1 \to 3 \times n$ **do**
    $dp[i] \leftarrow h'_i$
    $rule[i] = 0$
    for $j : 1 \to i - 1$ **do**
        if $d'_j > d'_i \cap w'_j > w'_i \cap dp[j] + h'_i > dp[i]$ **then**
            $dp[i] \leftarrow dp[j] + h'_i$
            $rule[i] = j$
        end if
    end for
end for
$mi \leftarrow \arg\max_i dp[i]$
$plan \leftarrow \varphi$
$maxheight \leftarrow dp[mi]$
for $mi \neq 0$ **do**
    add mi into $plan$
end for
return $plan, maxheight$