计算机学院《算法设计与分析》
(2020 年秋季学期)

第二次作业参考答案

1 假日愉悦值问题 (20 分)

Ange 开始计划她的 N 天长假，第 i 天她可以从以下三种活动中选择一种进行。

1. 去海边游泳。可以获得 a_i 点愉悦值;
2. 去野外爬山。可以获得 b_i 点愉悦值;
3. 在家里学习。可以获得 c_i 点愉悦值。

由于她希望自己的假日丰富多彩，她并不希望连续两天（或者两天以上）进行相同类型的活动。

试设计算法制定一个假日安排，使得在满足 Ange 要求的情形下所获得的愉悦值的和最大，并分析该算法的时间复杂度。

解：

1. 状态设计

设 $f[i][j]$ 表示在第 i 天进行的活动为第 j 种的条件下前 i 天进行活动后获得的最大愉悦值。

2. 状态转移

考虑到连续两天不能进行相同种类的活动，

1. 第 i 天选择去海边游泳，则前一天可以选择 2、3 两种活动，可能获得的最大预约值为 $a_i + \max\{f[i-1][2], f[i-1][3]\}$
2. 第 i 天选择去野外爬山，则前一天可以选择 1、3 两种活动，可能获得的最大预约值为 $b_i + \max\{f[i-1][1], f[i-1][3]\}$
3. 第 i 天选择在家里学习，则前一天可以选择 1、2 两种活动，可能获得的最大预约值为 $a_i + \max\{f[i-1][1], f[i-1][2]\}$

故有如下递推式

$$f[i][j] = \max_{j' \in \{1,2,3\} \cap j' \neq j} \{f[i-1][j']\} + \begin{cases} a_i & \text{if } j = 1 \\ b_i & \text{if } j = 2 \\ c_i & \text{if } j = 3 \end{cases}$$

3. 记录决策方案

为了记录决策方案，可以记 $pa[i][j]$ 表示 $f[i][j]$ 状态是由 $f[i-1][pa[i][j]]$ 状态转移而来的。那么可以利用 pa 逐次得到在第 n 天，第 $n-1$ 天，$\cdots$，第 1 天选择的最优活动。

4. 边界条件

第一天选择任何活动没有前置约束，故对应的愉悦值 $f[1][1] = a_1, f[1][2] = b_1, f[1][3] = c_1$，且没有对应 $pa[1][1], pa[1][2], pa[1][3]$ 无意义。

5. 目标状态

则情形下最大愉悦值对应的状态为 $f[n][k]$, 对应的最优安排为 $pa[n][k]$ 及其前序安排，其中 $k = \arg\max_{i=1,2,3} f[n][i]$

6. 时间复杂度分析

故总状态是 $O(n)$ 级别的 (注意到活动仅有 3 种)，每个状态的转移是 $O(1)$ 的，故总的时间复杂度为 $T(n) = O(n)$。伪代码如 Algorithm 1。

Algorithm 1 $fun(a[1..n], b[1..n], c[1..n])$

Input:

三个数组 $a[1..n], b[1..n], c[1..n]$ 表示每天完成每种活动的愉悦值;

Output:

n 天可获得的最大愉悦值，及其假日安排。

$f[1][1] \leftarrow a_1, f[1][2] \leftarrow b_1, f[1][3] \leftarrow c_i$
for $i : 2 \rightarrow n$ **do**
  $pa[i][1] \leftarrow \arg\max_{k=2,3} f[i-1][k]$
  $f[i][1] \leftarrow \max_{k=2,3} f[i-1][k]$
  $pa[i][2] \leftarrow \arg\max_{k=1,3} f[i-1][k]$
  $f[i][2] \leftarrow \max_{k=1,3} f[i-1][k]$
  $pa[i][3] \leftarrow \arg\max_{k=1,2} f[i-1][k]$
  $f[i][3] \leftarrow \max_{k=1,2} f[i-1][k]$
end for
$plan \leftarrow \emptyset$
$now \leftarrow \arg\max_{k=1,2,3} f[n][k]$
for $i : n \rightarrow 1$ **do**
  add the i-th day do the kind-now activity in front of $plan$.
  $now \leftarrow pa[i][now]$
end for

2 小跳蛙问题 (20 分)

给定 n 块石头，依次编号为 1 到 n。第 i 块石头的高度是 h_i。
现有一只小跳蛙在第 1 块石头上，它重复以下操作，直到它到达第 n 块石头：
若它当前在第 i 块石头上，则可跳到第 $j(i+1 \leq j \leq i+k)$ 块石头上，耗费的体力为 $|h_i - h_j|$。
试设计算法求它最少耗费多少体力可以到达第 n 块石头，并分析该算法的时间复杂度。

解:

1. 状态设计

记 $f[i]$ 表示小跳蛙跳到第 i 号石头上时的最小代价。

2. 状态转移

因为只可以从第 $i-1$ 到 $i-k$ 块石头跳到第 i 块石头，而跳到第 i 块石头的代价为上一块和第 i 块的高度差

故转移描述如下:

$$f[i] = \min_{j=i-k}^{i-1} \{f[j] + |h_i - h_j|\}$$

3. 边界条件

临界状态即 $f[1] = 0$。因为小跳蛙初始在第 1 块石头上。

4. 目标状态

由状态含义可知，到达第 n 块石头的最小代价为 $f[n]$

5. 时间复杂度分析

故总状态是 $O(n)$ 级别的，而每个状态的转移是 $O(k)$ 时间的。故总的时间复杂度为 $O(nk)$，参考伪代码如 Algorithm 2。

3 取星星问题 (20 分)

Ange 在玩一个取星星游戏：现有 n 个格子排成一排，每个格子里面存在一颗星星。第 i 个格子和第 $i+1(i<n)$ 个格子相邻。两颗星星满足如下两个条件之一则称为相连：

1. 两颗星星所在的格子相邻;
2. 两颗星星都与另外某颗星星相连。

Algorithm 2 $jump(\{h_n\})$

Input:

1 到 n 每块石头的高度 $\{h_n\}$

Output:

最少需要耗费多少体力

```
f[1] ← 0
for i : 2 → n do
    f[i] ← ∞
    for j : max(i − k, 1) → i − 1 do
        f[i] ← min(f[i], f[j] + |h_i + h_j|)
    end for
end for
return f[n]
```

目前 Ange 想从这些格子中取出给定的 m 颗星星，其对应得格子编号依次为 $b_1..b_m$ (已经正序)。取出星星 b_i 的花费为当前与 b_i 相连的星星个数。试设计算法求出一个取星星的顺序使得总花费最小，并分析该算法的时间复杂度。

例如 $n = 20, m = 3$,$[b_1, b_2, b_3] = [3, 6, 14]$。此时总花费最小的取星星顺序如下:

1. 取下第 14 个格子里的星星，格子 1 到 20 里的星星 (除自己外) 都与之相连，花费 19;
2. 取下第 6 个格子里的星星，格子 1 到 13 里的星星 (除自己外) 都与之相连，花费 12;
3. 取下第 3 个格子里的星星，格子 1 到 5 里的星星 (除自己外) 都与之相连，花费 4。

总花费为 $19 + 12 + 4 = 35$。

解:

1. 状态设计

可以考虑设计二维状态 $dp[i][j]$ 表示将 $b(i, j)$ 开区间内的所有星星都取出的最小花费。

2. 状态转移

枚举 $b(i, j)$ 开区间第一个被取出的星星 k 则总的花费变成:

$b(i, k)$ 和 $b(k, j)$ 两个区间的代价 (即 $dp[i][k], dp[k][j]$)，以及取出星星 k 的代价 $b_j - b_i - 2$(即区间的中剩下的星星个数)。

则转移如下:

$$dp[i][j] = \min_{k \in (i,j)} \{dp[i][k] + b_j - b_i - 2 + dp[k][j]\}$$

3. 记录决策方案

注意到

$$dp[i][j] = \min_{k \in (i,j)} \{dp[i][k] + b_j - b_i - 2 + dp[k][j]\}$$

的转移只有 $dp[i][k] + dp[k][j]$ 与 k 相关，故我们可以记录 $plan[i][j]$ 表示取出的星星的顺序，并考虑该区间第一个选出的位置:

$$pivot = \arg\max_{k \in (i,j)} \{dp[i][k] + dp[k][j]\}$$

则有

$$plan[i][j] = pivot + plan[i][pivot] + plan[pivot][j]$$

这里 $+$ 表示顺次拼接，即先选择 $pivot$ 的星星取出，再考虑把 $(i, pivot)$ 区间的星星取出，最后考虑 $(pivot, j)$ 区间的星星取出。

4. 边界条件

考虑到所有的开区间 $(i, i+1)$ 实际上是空区间，故其对应的开销都是零，且无需方案。

$$dp[i][i+1] = 0$$
$$plan[i][i+1] = \emptyset$$

5. 目标状态

注意到我们可以添加 $b_0 = 0$ 和 $b_{m+1} = n+1$ 这样 $dp[0][m+1]$ 恰好对应了将 $(0, n+1)$ 开区间内所有星星取出的代价，即原题意所求。而 $plan[0][m+1]$ 恰好对应了此时的一种取星星方案。

6. 时间复杂度分析

故总状态空间的大小为 $O(m^2)$，对于状态 $dp[i][j]$, 记 $|(i,j)| = k$ 每次需要用 $O(k)$ 的时间转移。故总时间复杂度为 $T(m) = \sum_{k=1}^{m} O(k) \times (m-k+1) = O(m^3)$。示例伪代码如 Algorithm 3。

Algorithm 3 $star(n, m, \{b_m\})$

Input:

星星总数 n，需要取出的 m 颗星星 $b_1 \sim b_m$。

Output:

取这 m 颗星星顺序，以及最小花费

do $dp[i][i+1] \leftarrow 0$ for $i \in [0, m]$
$b_0 \leftarrow 0, b_{m+1} \leftarrow n+1$
for $len : 2 \to m+1$ **do**
  for $i : 0 \to m+1-len$ **do**
    $j \leftarrow i + len$
    $pivot \leftarrow \arg\max_{k \in (i,j)} \{dp[i][k] + dp[k][j]\}$
    $plan[i][j] \leftarrow \left[pivot, plan[i][pivot], plan[pivot][j] \right]$
    $dp[i][j] \leftarrow b_j - b_i - 2 + \max_{k \in (i,j)} \{dp[i][k] + dp[k][j]\}$
  end for
end for
return $plan[0][m+1], dp[0][m+1]$

4 叠塔问题 (20 分)

给定 n 块积木，编号为 1 到 n。第 i 块积木的重量为 w_i(w_i 为整数)，硬度为 s_i，价值为 v_i。
现要从中选择部分积木垂直摞成一座塔，要求每块积木满足如下条件:
若第 i 块积木在积木塔中，那么在其之上的积木的重量和不能超过其硬度。
试设计算法求出满足上述条件的价值和最大的积木塔，并分析该算法的时间复杂度。

解:

1. 确定决策顺序

在确定 dp 状态前，我们先确定积木选择的顺序，思路如下:

假定目前已经放了重量为 W 的方块，考虑在底部放 i, j 两个积木，假定最优策略是 i 在 j 的上面，则有:

$$s_i < W + w_j$$
$$s_j \geq W + w_i$$

故有 $s_i + w_i < s_j + w_j$。如此可知在决策时按照 $s_i + w_i$ 从小到大的顺序决策正好能决策出一个自顶至底的最优策略。

2. 状态设计

故我们想将原积木按照 $s_i + w_i$ 排序，设 $dp[i][W]$ 表示，当前已经考虑了前 i 个积木，搭建了重量为 W 的积木塔的最大价值。

3. 状态转移

则转移类似背包问题:

考虑 $dp[i][W]$ 从哪些状态转移而来:

1. $W - w_i \leq s_i$，此时可以将 (s_i, w_i, v_i) 这块积木放置到塔的最底下，故可以选择状态 $dp[i-1][W-w_i]$ 对应的塔放在 (s_i, w_i, v_i) 这块积木之上。
2. $W - w_i > s_i$，此时不满足放置条件，只能考虑舍弃这块积木无法放置，即选择状态 $dp[i-1][W]$。

故转移方程为:

$$dp[i][W] = \begin{cases} dp[i-1][W] & W - w_i > s_i \\ \max\{dp[i-1][W], dp[i-1][W-w_i] + v_i\} & W - w_i \leq s_i \end{cases}$$

故答案为 $\max_w dp[n][w]$。

4. 记录决策方案

为了求出满足条件的最大价值积木塔，还需要记录哪些积木块被选择了。

注意到仅有在 $W - w_i \leq s_i$ 的条件下，且选择了决策 $dp[i-1][W-w_i] + v_i$ 才会选择积木块 (s_i, w_i, v_i)。

故记录 $elect[i][w]$ 表示 $dp[i][w]$ 状态时有哪些积木块被使用了。

则在转移选了决策 $dp[i-1][W-w_i]+v_i$ 时有 $elect[i][W] = elect[i-1][W-w_i] \cup \{(s_i, w_i, v_i)\}$，否则 $elect[i][W]$ 与 $elect[i-1][W]$ 相同。

5. 边界条件

对于没有考虑任何积木时，即 $dp[0][0 \sim \sum_i w_i]$，获得积木塔的价值都是 0，且 $elect[0][0 \sim \sum_i w_i]$ 均为 $\emptyset$

6. 目标状态

类似背包问题，最优方案为 $elect[n][mw]$，对应的最大高度为 $dp[n][mw]$，其中 $mw = \arg\max_w dp[n][w]$

7. 时间复杂度分析

考虑至少有 $O(n \times \sum_i w_i)$ 的状态，每个状态需要 $O(1)$ 的时间转移，故总的复杂度为 $O(n \times \sum_i w_i)$，伪代码如 Algorithm 4。

Algorithm 4 $tower(n, \{w_n\}, \{s_n\}, \{v_n\})$

Input:

n 块积木，第 i 块积木的重量为 w_i，硬度为 s_i，价值为 v_i。

Output:

价值和最大的积木塔选择了哪些积木块

```
sort (w_i, s_i, v_i) tuples by s_i + w_i in ascending order
sum ← 0
for i : 1 → n do
    for w : 0 → sum do
        if w − w_i ≤ s_i ∩ dp[i−1][w−w_i] + v_i > dp[i−1][w] then
            dp[i][w] ← dp[i−1][w−w_i] + v_i
            elect[i][w] ← {(s_i, w_i, v_i)} ∪ elect[i−1][w−w_i]
        else
            dp[i][w] ← dp[i−1][w]
            elect[i][w] ← elect[i−1][w]
        end if
    end for
    sum ← sum + w_i
end for
mw ← arg max_w dp[n][w]
return elect[n][w]
```

5 数组排序问题 (20 分)

给定整数数组 $A = [a_1, a_2, \cdots, a_n]$。你的任务是对该数组进行一系列操作使其变为非降序数组：每次操作你需要选定一个数字 x，然后将数组中所有等于 x 的元素统一移至数组的开始或结尾。例如数组 $A = [2, 1, 3, 1, 1, 3, 2]$ 可以通过如下两次操作变为非降序数组：

1. 将所有等于 1 的元素移至数组开头得到 $[1,1,1,2,3,3,2]$;

2. 将所有等于 3 的元素移至数组结尾得到 $[1,1,1,2,2,3,3]$。

请你设计算法来计算将给定数组变为非降序数组所需要的最少操作次数，并分析该算法的时间复杂度。

解：

1. 确定决策顺序

通过分析最优方案的执行过程，可以将原序列中所有数字可以分为两种：

1. 在将原序列变为有序序列的最优方案中，该数字被选中并执行过操作；

2. 在将原序列变为有序序列的最优方案中，该数字从未被选中。

据此，可以将原数列中包含的所有不同的数字分为两个序列 $m_1 < m_2 < \cdots < m_k$ 和 $d_1 < d_2 < \cdots < d_l$。序列 m 表示所有在最优方案中被操作过的数字。序列 d 表示所有在最优方案中未被操作过的数字。

例如，如果 $a = [2,1,5,1,3,5,4]$，则最优方案是将所有 1 移到开头，然后将所有 5 移到结尾，因此 $m = [1,5], d = [2,3,4]$。

Algorithm 5 $MinSort(a[1..n])$

```
for i ← 1 to n do
    max Ind(a[i]) ← max{i, max Ind(a[i])}
    min Ind(a[i]) ← min{i, min Ind(a[i])}
end for
s[1..t] ← all distinct numbers in a[1..n]
sort s[1..t] in ascending order
len[1] ← 1;
for i ← 2 to t do
    if max Ind(d_{i-1}) < min Ind(d_i) then
        len[i] ← len[i − 1] + 1
    else
        len[i] ← 1
    end if
end for
ans ← ∞
for i ← 1 to t do
    ans ← min{ans, t − len[i]}
end for
return ans
```

通过分析可发现如下两条重要性质：

1. 对于所有 $i \in \{2,3,\cdots,n\}$，$\text{max Ind}(d_{i-1}) < \text{min Ind}(d_i)$。其中，$\text{min Ind}(x)$ 是指满足 $a[i] = x$ 的最小的下标 i，$\text{max Ind}(x)$ 是指满足 $a[i] = x$ 的最大的下标 i；

2. 对于所有 $i \in \{2,3,\cdots,n\}$，在序列 m 中均不存在数 x 满足 $d_{i-1} < x < d_i$。

而最优方案的操作次数为 $|m| = k$，我们的目标是最小化操作次数，即为最大化序列 d 的长度。

2. 状态设计

因此,我们先取出原序列中所有不同的元素并将其以升序排列,得到新序列 $s_1, s_2, \cdots, s_t (s_{i-1} < s_i)$。定义 $len[i]$ 为考虑序列 s 中前 i 个元素时序列 d 的最长长度，可写出递归式：

3. 状态转移

$$len[i] = \begin{cases} len[i-1]+1 & \text{max Ind}(d_{i-1}) < \text{min Ind}(d_i) \\ 1 & otherwise \end{cases}$$

4. 边界条件

仅考虑第一个元素时，显然有 $len[1]=1$。

5. 目标状态

最优方案的操作次数即为 $t-\max_{i=1}^{t} len[i]$，其中 t 为序列 A 中所有不同的元素的数目。伪代码如 Algorithm 5所示。

6. 时间复杂度分析

该算法预处理由于涉及到排序，需要 $O(n\log n)$ 的时间，动态规划的过程共有 $O(n)$ 种状态，每种状态仅需要 $O(1)$ 的时间进行转移，因此时间复杂度为 $O(n)$，总的时间复杂度为 $O(n\log n)$。