北京航空航天大学《算法设计与分析》第二次作业

1 假日愉悦值问题 (20 分)

F 同学开始计划他的 N 天长假，第 i 天他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。

由于他希望自己的假日丰富多彩，他并不希望连续两天（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 F 同学要求的情形下所获得的愉悦值的和最大，输出这个假日安排以及最大愉悦和，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

1.1 思路分析

将该问题转化为子问题，即前 i 天的最大愉悦值等于前 $(i-1)$ 天的愉悦值和加上第 i 天的愉悦值，因为计算第 i 天愉悦值时需要考虑该天选择的活动不能和前一天重复，因此计算前 i 天最大愉悦值时需要对当天选择的各种活动类型分别计算并记录。因此做如下状态定义：

- **状态设定**：将第 i 天愉悦值的状态记为 $dp[i][j]$，其中 i 表示天数，j 表示当天选择的活动，j 的取值为 $0,1,2$，分别表示第 i 天选择的活动为海边游泳、野外爬山、在家学习。
- **状态转移方程**：

$$dp[i][j] = max(dp[i-1][k] + point[i][j]) \quad (i >= 1; k, j = 0, 1, 2; j \neq k) \tag{1}$$

  其中，$point[i][j]$ 表示第 i 天选择第 j 种活动的愉悦值。
- **状态记录**：用 $step[i][j]$ 记录第 i 天选择活动 j 时前一天的活动类型。最后计算出最大的 $dp[N-1][j]$ 即为最大的愉悦值和，通过 $step$ 数组回溯即可得到最大愉悦值时的假日安排。

1.2 伪代码实现

Algorithm 1: 假日愉悦值问题

Input: 天数 N，第 i 天三种活动的愉悦值 $point[i][j]$

Output: 最大愉悦值和 max，第 i 天的活动类型 $activity[i]$

```
1  function Holiday(N, point):
2      Let dp[N][3] and step[N][3] be new 2-dimension arrays;
3      Let activity[N] be a new array;
4      for i = 0 to N − 1 do
5          for j = 0 to 2 do
6              if i == 0 then
7                  dp[i][j] = point[i][j];
8                  step[i][j] = −1;
9              end
10             else
11                 maxac = 0;
12                 for k = 0 to 2 do
13                     if k! = j and dp[i − 1][k] + point[i][j] > maxac then
14                         maxac = dp[i − 1][k] + point[i][j];
15                         dp[i][j] = maxac;
16                         step[i][j] = k;
17                     end
18                 end
19             end
20         end
21     end
22     max = 0;
23     for j = 0 to 2 do
24         if dp[N − 1][j] > max then
25             max = dp[N − 1][j];
26             activity[N − 1] = j;
27         end
28     end
29     for i = N − 2 to 0 do
30         activity[i] = step[i + i][activity[i + 1]];
31     end
32     return max, activity;
33 end
```

1.3 时间复杂度分析

- 填愉悦值表的时有三层循环，最外层复杂度为 $O(N)$，内部两层复杂度均为 $O(3)$，因此时间复杂度为 $O(9N) = O(N)$。
- 找最大愉悦值的时间复杂度为 $O(3)$。
- 回溯填活动类型序列的时间复杂度为 $O(N)$。
- 因此总的时间复杂度为 $O(N)$。

2 倍序数组问题 (20 分)

一个长度为 n 的正整数数组 a，被称为“倍序数组”当且仅当对于任意 $1 \leq i \leq n-1$，都有 $a_i | a_{i+1}$（其中"|" 表示整除，即存在一个正整数 s 使得 $a_{i+1} = s \times a_i$）。请设计算法求出长度为 n 的倍序数组 a 的个数，且数组中每个元素满足 $1 \leq ai \leq k$，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。例如，当 k = 3, n = 2 时，满足条件的数组有 5 个，分别是 {1, 1}，{1, 2}，{1, 3}，{2, 2} 和 {3, 3}。

2.1 思路分析

考虑该问题的子问题，长度为 i 的倍序数组的数量等于长度为 $(i-1)$ 的倍序数组中结尾数字的相应倍数仍在给定范围内的数量。因此做如下状态定义：

- **状态设定**：将长度为 i 且以 j 结尾的倍序数组的数量记为 $dp[i][j]$
- **状态转移方程**：

$$dp[i][j] = \sum_{k \in F}^{j} dp[i-1][j/k] \quad (i \geq 2, F = \{k \in N_+ | j\%k == 0\}) \tag{2}$$

在实现过程中比较 trick 的一点时，因为最内层循环需要从 1 开始遍历出 j 的所有因数，时间复杂度较高，为了降低时间复杂度，我们可以选择正向填表的方式，即在填上一层时就将值加到下一层的相应位置上，从而避免了寻找因子时的遍历操作。

2.2 伪代码实现

Algorithm 2: 倍序数组问题

Input: 数组长度 n，数组元素最大值 k

Output: 倍序数组数量 sum

```
function MultipleSequence(n, k):
    Let dp[n][k] be a new 2-dimension array;
    for i = 1 to n − 1 do
        for j = 1 to k do
            if i == 1 then
                dp[i][j] = 1;
            end
            for m = 1 to k do
                if m ∗ j ≤ k then
                    dp[i + 1][m ∗ j]+ = dp[i][j];
                end
                else
                    break;
                end
            end
        end
    end
    sum = 0;
    for j = 1 to k do
        sum+ = dp[n][j];
    end
    return sum;
end
```

2.3 时间复杂度分析

- 初始化 dp 的时间复杂度为 $O(k)$。
- 填表时最内层循环循环的时间复杂度为 $O(\sum_{i=1}^{k} \frac{k}{i}) = O(\log k)$，因此填表的时间复杂度为 $O(nk \log k)$。
- 最后求和的时间复杂度为 $O(k)$.
- 因此总的时间复杂度为 $O(nk \log k)$。

3 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，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。（两种方案不同当且仅当存在一个花的种类 i，两种方案中第 i 种花的数量不同）

3.1 思路分析

考虑子问题，从前 i 种花中选出 j 枝花的方案数等于从前 $(i-1)$ 种花中选出 $k(max(1, j-a_i) \leq k \leq j)$ 枝花的方案数之和。因此做如下状态定义：

- **状态设定**：将从前 i 种花中选出 j 枝花的方案数记为 $dp[i][j]$
- **边界条件**：$dp[1][j] = \begin{cases} 1 & j \le a_1 \\ 0 & j > a_1 \end{cases}$
- **状态转移方程**：

$$dp[i][j] = \sum_{k=max(1,j-a_i)}^{j} dp[i-1][k] \quad (i \ge 2) \tag{3}$$

可以看到上述转移方程有大量重复求和运算，并且同一行相邻项的求和范围高度重叠，因此可以考虑通过待求项左边的项来简化计算。简化后的状态转移方程为：

$$dp[i][j] = \begin{cases} dp[i-1][j] + dp[i][j-1] & (i \ge 2, j \le a_i) \\ dp[i-1][j] + dp[i][j-1] - dp[i-1][j-a_i-1] & (i \ge 2, j > a_i) \end{cases} \tag{4}$$

3.2 伪代码实现

Algorithm 3: 鲜花组合问题

Input: 花种类数 n，每种花库存 $a[1 \dots n]$，花束花数 m

Output: 花束组合方案数 sum

```
function FlowerCombination(n, a, m):
    Let dp[n][m] be a new 2-dimension array;
    for i = 1 to n do
        dp[i][0] = 1;
    end
    for j = 1 to m do
        if j ≤ a[1] then
            dp[1][j] = 1;
        end
        else
            dp[1][j] = 0;
        end
    end
    for i = 2 to n do
        for j = 1 to m do
            if j ≤ a[i] then
                dp[i][j] = dp[i − 1][j] + dp[i][j − 1];
            end
            else
                dp[i][j] = dp[i − 1][j] + dp[i][j − 1] − dp[i − 1][j − a[i] − 1];
            end
        end
    end
    sum = dp[n][m];
    return sum;
end
```

3.3 时间复杂度分析

- 初始化 dp 的时间复杂度为 $O(n+m)$。
- 填表的时间复杂度为 $O(nm)$。
- 总的时间复杂度为 $O(nm)$。

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，那么在其之上摆放的所有积木的重量之和不能超过第 i 块积木的硬度。

试设计算法求出满足上述条件的价值和最大的积木塔，输出摆放方案和最大价值和。请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

4.1 思路分析

因为要考虑硬度和重量的关系，我们首先采用贪心算法的思想，对积木按照硬度与重量的和从小到大进行排序（证明：设 s1 在 s2 上方，且 s1 上方积木总重量为 c，并且假设 $s1.s+s1.w > s2.s+s2.w$，因为 s1 在 s2 上方，所以有 $s2.s \geq s1.w+c$，即 $s2.s-s1.w-c \geq 0$，所以有 $s1.s-s2.w-c > s2.s-s1.w-c \geq 0$，即如果 s1 的硬度与质量和大于 s2，那么它一定可以放在 s2 下方，也就是说所有合法序列都可以转换成下方的积木硬度质量和大于上方的积木的新序列），和相同的则按价值从大到小排序。

然后考虑子问题，做如下定义：

- **状态设定**: 设 $dp[i][j]$ 为从前 i 块中选重量之和为 j 的积木的最大价值
- **边界条件**：$dp[1][w[i]] = v[1]$

  $dp[i][0] = 0(0 \leq i \leq n)$
- **状态转移方程**：

$$dp[i][j] = \begin{cases} max(dp[i-1][j], dp[i-1][j-w_i]+v_i), & dp[i-1][j] \neq 0, \quad j-w_i \leq s_i, \quad j \geq w_i \\ dp[i-1][j], & else \end{cases} \tag{5}$$

- **状态记录**：用 $keep[i][j]$ 表示 $dp[i][j]$ 的状态是由哪个状态转移而来，如果是由 $dp[i-1][j]$ 转移而来，则 $keep[i][j] = 0$，否则 $keep[i][j] = 1$。最终通过回溯获得摆放方案。
- **复杂度优化**：由于在更新 dp 的过程中，我们的目的是找到最大的价值和，不关心具体计算到了前多少块，因此我们可以将 dp 压缩为一维数组，$dp[i]$ 直接存储总重量为 i 时的最大价值和。

4.2 伪代码实现

Algorithm 4: 叠塔问题

Input: 积木块数 n，积木序号 $id[1\dots n]$, 积木重量 $w[1\dots n]$，积木硬度 $s[1\dots n]$，积木价值 $v[1\dots n]$

Output: 最大价值和 $maxvalue$，自顶向下的摆放方案 $plan$

```
function Tower(n, id, w, s, v):
    Let dp[∑_{i=1}^{n} w[i]] be a new 1-dimension array;
    Let keep[n][∑_{i=1}^{n} w[i]] be a new 2-dimension array;
    Sort id, w, s, v by (s + w) ascending, v descending;
    maxvalue = 0;
    maxweight = 0;
    maxnum = 0;
    dp[1][w[1]] = v[1];
    for i = 2 to n do
        for j = min(∑_{p=1}^{n} w[p], w[i] + s[i]) to w[i] do
            if dp[j − w[i]] ≠ 0 and dp[j] < dp[j − w[i]] + v[i] then
                dp[j] = dp[j − w[i]] + v[i];
                if dp[j] > maxvalue then
                    maxvalue = dp[j];
                    maxweight = j;
                    maxnum = i;
                    flag = 1;
                end
                keep[i][j] = 1;
                continue;
            end
            keep[i][j] = 0;
        end
    end
    t = 0;
    for i = maxnum to 1 do
        if keep[i][maxweight] = 1 then
            plan[t++] = id[i];
            maxweight− = w[i];
        end
    end
    return maxvalue, plan;
end
```

4.3 时间复杂度分析

4.3.1 优化前

- 排序的时间复杂度为 $O(n\log n)$
- 初始化的时间复杂度为 $O(1)$

- 填表的时间复杂度为 $O(n\sum_{i=1}^{n} w[i])$（因为 $w[n]+s[n]$ 决定了所有合法序列的重量和，所以也可写成 $O(n(w[n]+s[n]))$）
- 回溯的时间复杂度为 $O(n)$
- 由于 w_i 是整数，所以 $\sum_{i=1}^{n} w[i] \geq n > \log n$，因此总的时间复杂度为 $O(n\sum_{i=1}^{n} w[i])$（或 $O(n(w[n]+s[n]))$）。

4.3.2 优化后

- 排序的时间复杂度为 $O(n\log n)$
- 初始化的时间复杂度为 $O(1)$
- 填表的时间复杂度为 $O(\sum_{i=1}^{n} min(s[i], \sum_{k=i+1}^{n} w[i]))$
- 回溯的时间复杂度为 $O(n)$
- 总的时间复杂度为 $O(\sum_{i=1}^{n} min(s[i], \sum_{k=i+1}^{n} w[k]) + n\log n)$。因为减少了填表时内层循环的次数，因此时间复杂度一定小于优化前的复杂度。

5 最大分值问题 (20 分)

给定一个包含 n 个整数的序列 a1, a2, . . . , an，对其中任意一段连续区间 ai..aj，其分值为

$$\left(\sum_{t=i}^{j} a_t\right) \%p \tag{6}$$

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，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

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。

5.1 思路分析

- **状态设定**：设 $dp[i][j]$ 为将前 i 个数分为 j 段的最大分值和，$prefix[i]$ 表示前 i 个数之和
- **状态转移方程**：

$$dp[i][j] = \begin{cases} max(dp[v][j-1] + ((\sum_{t=v+1}^{i} a_t)\%p)), & j \leq i, \quad j-1 \leq v \leq i-1 \\ -\infty, & j > i \end{cases} \tag{7}$$

- **优化处理**：为了减少每次更新 dp 时遍历的次数，因为 dp 的所有值 mod$p$ 都小于 p，所以可以维护一个 p 行 k 列的数组，将目前已知最大的满足 $prefix[i]\%p == i_0$ 的 $dp[i][j_0]$ 记录在 $P[i_0][j_0]$ 中，在后面用到第 j_0 列的数据时，直接遍历 P 的第 j_0 列即可。

5.2 伪代码实现

Algorithm 5: 最大分值问题

Input: 序列长度 n，序列 $a[1 \dots n]$，分段数 k，模数 p

Output: 最大分值和 $maxvalue$

```
function MaxValue(n, a, k, p):
    Let dp[n][k] be a new 2-dimension array;
    Let P[p][k] be a new 2-dimension array;
    Let prefix[n] be a new array;
    prefix[0] = 0;
    for i = 1 to n do
        prefix[i] = prefix[i − 1] + a[i];
        dp[i][1] = prefix[i]%p;
        for j = 2 to k do
            dp[i][j] = −∞;
        end
    end
    for i = 2 to n do
        for j = 2 to k do
            if j < i then
                max = 0;
                for g = 0 to p − 1 do
                    temp = P[g][j − 1] + (prefix[i] − g)%p;
                    if temp > max then
                        max = temp;
                    end
                end
                dp[i][j] = max;
                if P[max%p][j] < max then
                    P[max%p][j] = max;
                end
            end
            else if j == i then
                dp[i][j] = dp[i − 1][j − 1] + a[i]%p;
            end
        end
    end
    maxvalue = dp[n][k];
    return maxvalue;
end
```

5.3 时间复杂度分析

- 初始化的时间复杂度为 $O(nk)$
- 填表的时间复杂度为 $O(npk)$
- 因此总的时间复杂度为 $O(npk)$