计算机学院《算法设计与分析》
(2019 年秋季学期)

第二次作业参考答案

1 二进制串变换问题 (20 分)

给定两个长度均为 n 的仅由 0 和 1 组成的字符串 a 和 b，你可以对串 a 进行如下操作：

1. 对任意 $i, j (1 \le i, j \le n)$，交换 a_i 和 a_j，操作代价为 $|i - j|$；
2. 对任意 $i (1 \le i \le n)$，取反 a_i，操作代价为 1。

请你设计算法计算将串 a 变为串 b 所需的最小代价（只能对串 a 进行操作），并分析该算法的时间复杂度。

解：定义状态 $C[i]$ 表示将串 a 的前 i 位变成 b 所需的最小代价，显然将串 A 变为串 B 的最小代价为 $C[n]$。 ……（5 分）

很容易发现，仅在存在连续两位需要取反且这两位不相等时才会选择交换操作，其他情况下使用取反操作即可。据此可得出如下递归式：

$$C[i] = \max \begin{cases} C[i-1] & \text{若} a[i], b[i] \text{相等} \\ C[i-1]+1 & \text{若} a[i], b[i] \text{不相等} \\ C[i-2]+1 & \text{若} a[i] \text{和} b[i-1] \text{相等且} b[i] \text{和} a[i-1] \text{相等} \end{cases}$$

……（10 分）

按照递增的顺序依次计算每个 $C[i]$，初始条件为 $i = 0$ 时，此时串 a 和串 b 均为空，无需任何操作，因此 $C[0] = 0$。

……（2 分）

算法的伪代码如 Algorithm 1 所示。

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

Input: 两个长度为 n 的二进制串 a, b
Output: 将串 a 变为串 b 的最小代价

```
C[0] ← 0
for i ← 1 to n do
    if a[i] = b[i] then
        C[i] ← C[i − 1]
    else
        C[i] ← C[i − 1] + 1
    end if
    if i > 1 and a[i] = b[i − 1] and a[i − 1] = b[i] then
        C[i] ← max{C[i], C[i − 2] + 1}
    end if
end for
return C[n]
```

时间复杂度分析：该算法共有 $O(n)$ 种状态，每种状态仅需 $O(1)$ 的时间进行转移，总的时间复杂度为 $O(n)$。 ……（3 分）

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：

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。

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 $\max\limits_{1\le i\le n} c[i]$。 ……（4 分）

考虑 $c[i]$ 的更新过程，若存在某个 $x_j < x_i (1 \le j < i)$，那么以 x_j 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 $c[j]$ 最大的 j 来更新 $c[i]$。据此可以写出如下递归式：

$$c[i] = \max_{1\le j<i, x_j<x_i} c[j] + 1$$

注意到以 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，因此初始化 $c[i] = 1 (1 \le i \le n)$。

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

Input: 包含 n 个元素的序列 $x[1..n]$
Output: 序列 x 的最长递增子序列

```
1: //动态规划
2: for i ← 1 to n do
3:     c[i] ← 1
4:     r[i] ← −1
5:     for j ← 1 to i − 1 do
6:         if x[j] < x[i] and c[j] + 1 > c[i] then
7:             c[i] ← c[j] + 1
8:             r[i] ← j
9:         end if
10:    end for
11: end for
12: k ← 1
13: for i ← 2 to n do
14:    if c[i] > c[k] then
15:        k ← i
16:    end if
17: end for
18: //方案追踪
19: s ← 空串
20: while k ≠ −1 do
21:    将 x[k] 放置在串 s 的首位
22:    k ← r[k]
23: end while
24: return s
```

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 $c[k] = \max\limits_{1\le i\le n} c[i]$ 即是序列 X 的最长递增子序列的长度。 ……（8 分）

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 $r[i] = \arg\max\limits_{1\le j<i, x_r<x_i} c[j]$。令 $c[k] = \max\limits_{1\le i\le n} c[i]$，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。 ……（5 分）

该算法的伪代码如 Algorithm 2 所示。

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 $O(\sum_{i=1}^{n} i) = O(n^2)$。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 分）

时间复杂度为 $O(n \log n)$ 的正确算法也可得满分。

3 括号匹配问题 (20 分)

定义**合法的括号串**如下：

1. 空串是合法的括号串；
2. 若串 s 是合法的，则 （s）和 [s] 也是合法的；
3. 若串 a, b 均是合法的，则 ab 也是合法的。

现在给定由‘[’，‘]’和‘(’，‘)’构成的字符串，请你设计算法计算该串中合法的子序列的最大长度，并分析该算法时间复杂度。例如字符串“([(])])”，最长的合法子序列“([()])”长度为 6。

解：

令 $S =< s_1, s_2, \cdots, s_n >$ 为给定的字符串，我们希望计算该串中最长的合法子序列的长度。

定义状态 $D[i, j]$ 表示子串 $s[i..j]$ 的最长合法子序列的长度，则我们的目标是计算 $D[1, n]$。 ……（5 分）

考虑合法子序列的构成方式，一种方式是整个合法子序列被一对括号包裹起来，形如“(..)”或“[..]”；另一种方式是整个合法子序列由多个由括号包裹的合法子序列连接起来；形如“(..)[..]”。对于第一种方式，除去两个括号之外剩余的部分构成了一个子问题；对于第二种方式，我们可以枚举第一个合法子序列的位置来找到最长的合法子序列。递归式如下：

$$D[i,j] = \max \begin{cases} D[i+1, j-1] + 2 & \text{若S[i]=‘(’,S[j]=‘)’或S[i]=‘[’,S[j]=‘]’} \\ D[i,k] + D[k+1, j] & \text{对所有} k = \{i, i+1, \cdots, j-1\} \end{cases}$$

……（10 分）

由于长度大于 0 的合法子序列至少包含了两个字符，可以初始化所有长度为 1 的区间其最长子序列的长度为 0($D[i, i] = 0 (1 \le i \le n)$)。和矩阵链乘问题类似，我们按照区间长度递增的顺序来进行转移。 ……（2 分）

算法的伪代码如 Algorithm 3 所示。

Algorithm 3 $LVS(s[1..n])$

```
for i ← 1 to n do
    D[i, i] ← 0
end for
for l ← 2 to n do
    for i ← 1 to n − l + 1 do
        j ← i + l − 1
        if s[i] = ‘(’ and s[j] = ‘)’ or s[i] = ‘[’ and s[j] = ‘]’ then
            D[i, j] ← d[i + 1, j − 1] + 2
        end if
        for k ← i to j − 1 do
            D[i, j] ← max{D[i, j], D[i, k] + D[k + 1, j]}
        end for
    end for
end for
return D[1, n]
```

时间复杂度分析：该算法共有 (n^2) 种状态，每种状态有 $O(n)$ 种转移的方式，总的时间复杂度为 $O(n^3)$。 ……（3 分）

4 分组可行性判定问题 (20 分)

给定按非降序排列的 n 个数 $a_1, a_2, \ldots, a_n$。现需将这 n 个数分组，满足：

1. 每个数 a_i 仅属于一个组；
2. 每个组中包含至少 k 个数；
3. 对属于同一组的任意两个元素 a_i, a_j，需满足 $|a_i - a_j| \leq d$。

请你设计算法判断是否可以将给定的 n 个数按照上述要求分组，并分析该算法的时间复杂度。

解：

由于题目中对每一组内的元素做了限制：$|a_i - a_j| \leq d$。故应该按照元素从小到大的顺序依次进行分组。（若存在其他分组方式，也一定可以将其转化为这种方式） ……（2 分）

令 $dp[i]$ 表示是否可以将前 i 个元素在满足题目要求的前提下进行分组。$dp[i] = 1$ 表示可以将前 i 个元素正确分组，$dp[i] = 0$ 表示不可以。我们的目标是求出 $dp[n]$。 ……（5 分）

考虑 $dp[i]$ 的转移过程，可以枚举上一个能够正确分组的位置 j，则最后一组是由 $a[j+1..i]$ 构成的，且必须满足：

1. $a[1..j]$ 可以正确分组 ($dp[j] = 1$)；
2. $a[j+1..i]$ 这一组的元素个数大于等于 k ($i - j \geq k$)；
3. $a[j+1..i]$ 这一组中的最大值与最小值之差小于等于 d ($a[i] - a[j+1] \leq d$)。

据此，可写出如下递归式：

$$dp[i] = \begin{cases} 1 & \text{存在} j \in [0..i-k] \text{满足} dp[j] = 1, a_i - a_{j+1} \leq d \\ 0 & \text{否则} \end{cases}$$

……（8 分）

按照 i 从小到大的顺序依次计算每个 $dp[i]$，初始化 $dp[0] = 1$。 ……（2 分）

直接使用上述递归式求解，其共有 $O(n)$ 种状态，每种状态需要花费 $O(n)$ 的时间进行转移，总时间复杂度为 $O(n^2)$。 ……（2 分）

该方法可进一步优化，由于给定的数组 $a[1..n]$ 是有序的，因此满足 $a_i - a_{j+1} \leq d$ 的元素一定构成了一段连续区间。我们令 $L[i]$ 表示上述区间的左端点，即满足 $a_i - a_{j+1} \leq d$ 的最小的 j。则可通过 Algorithm 4 预处理出数组 L。

Algorithm 4 $getL(a[1..n], d)$

```
j ← 0;
for i ← 1 to n do
    while j < i and a[i] − a[j + 1] > d do
        j ← j + 1;
    end while
    L[i] = j;
end for
return L;
```

在有了 L 数组后，原递归式可改写为：

$$dp[i] = \begin{cases} 1 & \text{存在} j \in [L[i]..i-k] \text{满足} dp[j] = 1 \\ 0 & \text{否则} \end{cases}$$

根据上述递归式，对状态 $dp[i]$ 需判断子数组 $dp[L[i]..i-k]$ 中是否出现了 1。为了快速判断，我们可以求出 $\sum_{t=L[i]}^{i-k} dp[t]$，若其值大于 0，说明子数组 $dp[L[i]..i-k]$ 中出现了 1。求和的方法可采用前缀求和法。即令 $sum[i] = \sum_{j=0}^{i} dp[i]$。则：

$$\sum_{t=L[i]}^{i-k} dp[t] = \begin{cases} sum[i-k] - sum[L[i]-1] & L[i] > 0 \\ sum[i-k] & L[i] = 0 \end{cases}$$

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

```
L ← getL(a[1..n], d)
dp[0] ← 1
for i ← 0 to k − 1 do
    sum[i] ← 1
end for
for i ← k to n do
    if L[i] ≤ i − k then
        tag ← 0
        if L[i] = 0 then
            tag ← sum[i − k]
        else
            tag ← sum[i − k] − sum[L[i] − 1]
        end if
        if tag > 0 then
            dp[i] ← 1
        else
            dp[i] ← 0
        end if
        sum[i] ← sum[i − 1] + dp[i]
    end if
end for
if dp[n] = 1 then
    return true
else
    return false
end if
```

算法伪代码如 Algorithm 5 所示。

给出时间复杂度为 $O(n^2)$ 或 $O(n)$ 的算法均可得满分。

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。

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：

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。我们的目标是求出 $f[n, k]$。 ……（5 分）

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：

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

……（10 分）

初始化仅需将所有的 $f[i, j]$ 置为 0，按照 i, j 从小到大的顺序进行更新即可。 ……（2 分）

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 $O(n^2k)$。

……（3 分）

该问题存在另外一种解法，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，

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

算法伪代码如 Algorithm 6 所示。

Algorithm 6 $Feasible(a[1..n],k,p)$

```
1:  sum[0] ← 0;
2:  for i ← 1 to n do
3:      sum[i] ← sum[i − 1] + a[i]
4:  end for
5:  for i ← 1 to n do
6:      for j ← 1 to k do
7:          for t ← 0 to p − 1 do
8:              f[i, j] = max{f[i, j], g[t, j − 1] + (sum[i] − t + p)%p}
9:              g[sum[i], j] = max{g[sum[i], j], f[i, j]}
10:         end for
11:     end for
12: end for
13: return f[n, k]
```

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 $O(nkp)$。

给出时间复杂度为 $O(n^2k)$ 或 $O(nkp)$ 的算法均可得满分。