算法设计与分析 (2019 年秋季学期)

第三次作业参考答案

1 二进制串变换问题 (25 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 $C[n]$。

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：

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

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 $C[0] = 0$。

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] ← min{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 $O(n)$。

2 括号匹配问题 (25 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 $D[1, n]$。

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：

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

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。算法的伪代码如 Algorithm 2 所示。

Algorithm 2 $LVS(s[1..n])$

```
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 数据修改问题 (25 分)

给定一个长度为 n 的数组 $a[1..n]$。最初，a 的所有元素都等于 0。

现要对 a 中的数据进行 Q 次修改，每次修改给定三个数 l_i, r_i 和 $x_i (1 \le i \le Q)$，其中 $1 \le l_i \le r_i \le n$ 且 $x_i > 0$，表示将数组 $a[l_i..r_i]$ 中的所有元素均加上 x。

现允许你从这 Q 次修改中删掉一次，使得删掉这次修改后，执行剩余的修改得到的数组 a 中的最大元素尽可能小。请求出该最小值。

例如，对于一长度为 4 的数组 $a = [0, 0, 0, 0]$，有三次修改分别为 $\begin{cases} l_1 = 2, r_1 = 4, x_1 = 2 \\ l_2 = 1, r_2 = 2, x_2 = 2 \\ l_3 = 3, r_3 = 4, x_3 = 1 \end{cases}$。经过第一次修改后，该数组变为 $[0, 2, 2, 2]$；经过第二次修改后，该数组变为 $[2, 4, 2, 2]$; 经过第三次修改后，该数组变为 $[2, 4, 3, 3]$。

如果我们删掉第 1 次修改，则经过 2、3 次修改得到的数组为 $[2,2,1,1]$，该数组中的最大值为 2。可以验证这是在所有可行方案中数组的最大值最小的方案。（删掉第二次修改得到的数组为 $[0,2,3,3]$，最大值为 3; 删掉第三次修改得到的数组为 $[2,4,2,2]$，最大值为 4。）

解：

在所有的修改完成之后，我们可以先找出当前数组中的最大值，记为 $a[m] = \max_{1 \le i \le n} a[i]$。其中，$m$ 为该最大值在数组中的下标。

若某次修改 (l_i, r_i, x_i) 所操作的区间 $a[l_i..r_i]$ 没有包含元素 $a[m]$，那么删掉这次修改，原数组的最大值不会发生变化。因此，我们仅需考虑区间包含了 $a[m]$ 的修改操作。

Algorithm 3 $DataUpdate(M[1..n, 1..m])$

```
// 初始化
for i ← 1 to Q do
    a[l_i] ← a[l_i] + x_i
    a[r_i + 1] ← a[r_i + 1] − x_i
end for
a[0] ← 0
for i ← 1 to n do
    a[i] ← a[i − 1] + a[i]
end for
for i ← 1 to n do
    mx_l[i] ← max{a[i], mx_l[i − 1]}
end for
a[n + 1] ← 0
for i ← n downto 1 do
    mx_r[i] ← max{a[i], mx_r[i + 1]}
end for
mx ← mx_r[1]
ans ← mx
// 枚举区间，计算答案
for i ← 1 to Q do
    val ← max{mx − x[i], mx_l[l_i − 1], mx_r[r_i + 1]}
    ans ← min{ans, val}
end for
return ans
```

假定第 j 次修改 (l_j, r_j, x_j) 所操作的区间 $a[l_j..r_j]$ 包含了元素 $a[m]$，那么删去这次修改一定会导致原数组的最大值变小。我们枚举所有这样的区间即可找到答案。那么，如何计算将第 j 次修改删掉后，数组 $a[1..n]$ 的最大值呢？

我们可以将数组 $a[1..n]$ 分为三部分：

1. $a[1..l_j - 1]$，删去第 j 次操作对这部分的元素不会有任何修改，因此这部分的最大值就是原数组中这一段的最大值；
2. $a[l_j..r_j]$，这部分的最大值为 $a[m]$，而删去第 j 次操作会使这部分中每个元素都减少 x_j，因此在删去第 j 次操作后这部分的最大值为 $a[m] - x_j$；
3. $a[r_j + 1..n]$，和第一部分相同，删去第 j 次操作对这部分的元素不会有任何修改，因此这部分的最大值就是原数组中这一段的最大值。

对于第一部分和第三部分，我们可以通过维护数组的前缀最大值和后缀最大值从而用 $O(1)$ 的时间得到这两段的最大值。

该算法的伪代码如 Algorithm 3所示。可以看到，伪代码中直接枚举了所有区间进行计算，这是因为枚举不包括 $a[m]$ 的区间一定不会影响答案。

时间复杂度分析：该算法预处理需要 $O(n + Q)$ 的时间，后续枚举每个区间并计算答案需要 $O(Q)$ 的时间，总的时间复杂度为 $O(n + Q)$。

4 最大矩阵问题 (25 分)

给定一个包含 n 行 m 列的二进制矩阵 M(对于 $1 \le i \le n, 1 \le j \le m, M_{i,j} = 0$ 或 1)。可以对其执行如下两种的操作：

1. 选择矩阵的第 i 行，并将该行的每个元素取反（如果该元素为 0 则将其改为 1，如果该元素为 1 则将其改为 0）；
2. 选择矩阵的第 j 列，并将该列的每个元素取反。

例如，矩阵 $\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ 将第一行取反可以得到 $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$；在此基础上再将第三列取反可以得到 $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$。每种操作都可以执行无限次。

最终，将得到的矩阵的每一行解释为一个数字的二进制表示，例如，矩阵 $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$ 的第一行为二进制数 $(101)_2$，其对应的十进制数为 $4+0+1=5$，第二行为二进制数 $(111)_2$，其对应的十进制数为 7，第三行为二进制数 $(110)_2$，其对应的十进制数为 6，这三个数之和为 $5+7+6=18$。

现请你计算在通过执行上述操作可以得到的矩阵中，n 个二进制数之和的最大值。

解：

很容易发现，该矩阵的任意一行或任意一列最多仅会被操作一次。因为第二次操作会取消第一次操作做过的更改。因此，对每一行（列），我们仅需判断这一行（列）是不是需要操作即可。

由于该矩阵的每一行为一个二进制表示，对于二进制数而言，若它的最高位为 0，则将它的最高位修改为 1 一定会使该二进制数变大。因此，我们可以先对每一行进行判断，若该行中第一列的数字为 0，则对这一行做一次操作。

经过上述操作之后，该矩阵每一行的第一个元素均为 1。在此情况，每一行再做任何操作只会使答案变小。因此接下来我们考察每一列是否还可通过反转使答案变大。

对每一列，若将该列反转会使 1 的个数增加，那么最终该矩阵构成的二进制数之和一定会增加。因此，对每一列，我们检查该列中 0 的个数和 1 的个数，若 0 的个数大于 1 的个数，则对该列进行一次操作。该算法的伪代码如 Algorithm 4所示。

时间复杂度分析：该算法仅需对整个矩阵遍历一次，时间复杂度为 $O(nm)$。

Algorithm 4 $MaxMatrix(M[1..n, 1..m])$

```
// 检查每一行是否需要交换
for i ← 1 to n do
    if M[i, 0] = 0 then
        for j ← 1 to m do
            M[i, j] ← 1 − M[i, j]
        end for
    end if
end for
// 检查每一列是否需要交换
for j ← 1 to m do
    count ← 0
    for i ← 1 to n do
        count ← count + matrix[i, j]
    end for
    if count ≤ n/2 then
        for i ← 1 to n do
            M[i, j] ← 1 − M[i, j]
        end for
    end if
end for
// 计算答案
res ← 0
for j ← 1 to m do
    count ← 0
    for i ← 1 to n do
        count ← count + M[i, j]
    end for
    res ← res + count × 2^(m−j)
end for
return res
```