HW01 算法概述作业

2253666-刘禹锡-HW01

一、 相关知识

本章为算法概述，重点在理解算法的概念，掌握算法在最坏，最好情况和平均情况下的计算复杂性概念。掌握算法复杂性的渐进性态的数学表述。以及了解 NP 类问题的基本概念。

算法满足 4 条性质：①零个或多个输入②一个或多个输出③确定性，每条指令清晰，无歧义④有限性，执行次数和执行时间有限。程序不一定满足性质④。时间复杂度

算法的时间复杂度又分为最好时间复杂度，最坏时间复杂度、平均时间复杂度。其中默认为最坏时间复杂度。渐进记号：渐近上界记号 O，渐近下界记号 Ω，渐进精确界符号 Θ。

二、 算法分析题

1-6

(1) $\log n^2 = 2\log n$

$O(f(n)) = O(2\log n) = O(\log n)$ $\quad \Omega(f(n)) = \Omega(\log n) = \Omega(g(n))$

$O(g(n)) = O(\log n + 5) = O(\log n)$

$\therefore f(n) = \Theta(g(n))$

$\therefore \log n^2 = \Theta(\log n + 5)$

(2) $f(n) = 2\log n$ $\quad g(n) = n^{\frac{1}{2}}$ $\quad 2\log n < 2n^{\frac{1}{2}}$. $c=2$. $n>16$. $\quad \therefore \log n^2 = O(\sqrt{n})$

(3) $n>16$ 时. $\quad n > \log^2 n$ $\quad f(n) = \Omega(g(n))$ $\quad \therefore n = \Omega(\log^2 n)$

(4) 当$n>1$时. $n\log n + n > \log n + n > \log n$ $\quad \therefore n\log n + n = \Omega(\log n)$

(5) $10 = \Theta(\log 10)$ 都为常数,同阶

(6) 当$n>1$时. $\log^2 n > \log n$ $\quad \log^2 n = \Omega(\log n)$ $\qquad n\lg^2 = 2 + 2\lg^n$

(7) 当 $n>100$ 时 $\quad 2^n > 100n^2$ $\quad 2^n = \Omega(100n^2)$

(8) 当$n>0$时 $\quad 2^n < 3^n$ $\quad 2^n = O(3^n)$

1-7 证明：$n! = o(n^n)$

要证 $n! = o(n^n)$ 即证：对于 $\forall \varepsilon > 0$，$\exists N_0 \in N^+$，使得当$N \geq N_0$时 $f(N)/g(N) < \varepsilon$

$$\frac{f(N)}{g(N)} = \frac{n!}{n^n} = \frac{n}{n}\cdot\frac{n-1}{n}\cdot\frac{n-2}{n}\cdot\cdots\cdot\frac{1}{n}$$
$$= \left(\frac{n-1}{n}\cdot\frac{n-2}{n}\cdots\frac{2}{n}\right)\cdot\frac{1}{n}$$

$\because \frac{n-1}{n}\cdots\frac{2}{n} < 1$ $\quad \therefore \frac{f(N)}{g(N)} < \frac{1}{n}$ $\quad \lim_{N\to\infty}\frac{f(N)}{g(N)}$ $\quad \lim_{n\to\infty}\frac{1}{n} = 0$

对$\forall \varepsilon > 0$, $\quad N \geq N_0$. $\frac{f(N)}{g(N)} < \frac{1}{N} \leq \frac{1}{N_0}$ $\quad$ 取 $N_0 > \frac{1}{\varepsilon}$ 即可

$\therefore f(N)/g(N) < \varepsilon$

法二：Stirling approximation: $\quad n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$

$$\frac{n!}{n^n} = \frac{\sqrt{2\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{e^n} \qquad \lim_{n\to\infty}\frac{n!}{n^n} = \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{e^n} = 0$$

$\therefore f(N)/g(N) < \varepsilon$ $\quad \therefore n! = o(n^n)$

三、 算法实现题

（一） 最多约数问题

1） 问题描述

正整数 x 的约数是能整除 x 的正整数。正整数 x 的约数个数记为 div(x)。例如，1，2，5，10 都是正整数 10 的约数，且 div(10)=4。设 a 和 b 是 2 个正整数，a≤b，找出 a 和 b 之间约数个数最多的数。

2） 算法设计

使用循环遍历，找出 a 和 b 之间的约数，不断更新最多的约数个数。但是不晓得这题具体是问数还是个数，看样例应该是个数，如果还需要这个数，则使用一个变量随同个数同步更新即可。

```
/// @brief 求约数个数
/// @param x  当前数x
/// @return sum 约数个数
int div(int x)
{
    int sum = 0;
    for (int i = 1; i <= x; ++i) {
        if(x%i==0)
            sum++;
    }
    return sum;
}
```

时间复杂度分析：O(n).

3） 源代码

```
#include <iostream>
using namespace std;
/// @brief 求约数个数
/// @param x
/// @return
int div(int x)
{
    int sum = 0;
    for (int i = 1; i <= x; ++i) {
        if(x%i==0)
            sum++;
    }
    return sum;
}

int main()
{
    int a, b, maxNum = 0;
    int tmp = 0;
    cin >> a >> b;
    for (int i = a; i <= b; ++i) {
        tmp=div(i);
        if(tmp>maxNum)
            maxNum=tmp; //更新
    }
    cout << maxNum << endl;
    return 0;
}
```

（二） 金币阵列问题

1） 问题描述

有 m×n(m≤100,n≤100)m×n(m≤100,n≤100)个金币在桌面上排成一个 mm 行 nn 列的金币阵列。每一枚金币或正面朝上或背面朝上。用数字表示金币状态，00 表示金币正面朝上，11 表示背面朝上。金币阵列游戏的规则是：

（1）每次可将任一行金币翻过来放在原来的位置上；

（2）每次可任选 22 列，交换这 22 列金币的位置。

给定金币阵列的初始状态和目标状态，计算按金币游戏规则，将金币阵列从初始状态变换到目标状态所需的最少变换次数。

2） 算法设计

现在一组数据中有两个矩阵，要实现两个矩阵的转换必须要对原阵列进行相应操作，这必须要有一个临时阵列存放最终状态数组 b。实际可以由 b 反向求到 a，如果可以实现，说明存在一个方法，记录步数，反之不存在输出-1。至于先行还是先列差别不大，这里选择使用先变化列。

依次将 b 中第 k 列与第一列交换，同时初始化步数 step，随后看第一列的各行是否相同，不相同再将该行执行①操作，即将该行硬币全翻转，随后处理剩下的列去寻找是否存在对应的列与最初状态 a 的某一列相同，如果存在，则进行交换，否则不存在进行下一个循环，将 k+1 列作为第一列处理，最后如果所有的都找到并经过列交换归位，更新结果 ans。

```
/// @brief 翻转指定行的金币状态
/// @param num 要翻转的行的索引
/// @details 将指定行的每个金币的状态进行翻转（0 变1，1 变0）
void changeRow(int num) {
    for (int i = 1; i <= Col; ++i) {
        b[num][i] ^= 1; // 通过异或操作实现状态翻转
    }
}
```

时间复杂度：O(n).

```
/// @brief 交换两列金币的位置，如果两列不同，则进行交换，并增加步数计数
/// @param num1 第一列的索引
/// @param num2 第二列的索引
/// @param step 操作步数的引用，用于记录总的操作步数
void changeCol(int num1, int num2, int&amp; step) {
    if (num1 == num2) return; // 如果列索引相同，则不需要操作
    step++;
    for (int i = 1; i <= Row; ++i) {
        int temp = b[i][num1];
        b[i][num1] = b[i][num2];
        b[i][num2] = temp;
    }
}
```

时间复杂度：O(n).

```
/// @brief 处理每个测试案例，计算达到目标状态的最小操作步数
/// @return 最小操作步数，如果无法达到目标状态则返回-1
int findMinStep() {
    int ans = inf;
    memcpy(tmp, b, sizeof(tmp));
    // 尝试每一列作为初始列进行列交换
    for (int k = 1; k <= Col; ++k) {
        int step = 0; // 当前尝试的步数
        memcpy(b, tmp, sizeof(tmp)); // 恢复 b 状态
        changeCol(1, k, step); // 尝试将第一列与第 k 列交换
        // 遍历每行，检查并翻转不匹配的行
        for (int i = 1; i <= Row; ++i) {
            if (a[i][1] != b[i][1]) {
                changeRow(i);
                step++;
            }
        }
        // 检查并尝试交换剩余的列
```

(n).

时间复杂度：$O(n^2)$.

```
int found = 1;
        for (int i = 1; i <= Col && found; ++i) {
            found = 0;
            for (int j = i; j <= Col; ++j) {
                if (check(i, j)) {
                    changeCol(i, j, step);
                    found = 1;
                    break;
                }
            }
        }
        // 如果找到一个可行的方案，更新最小步数
        if (found &amp;&amp; step < ans) ans = step;
    }
    return ans < inf ? ans : -1; // 如果找到则返回最小步数，否则返回-1
}
```

时间复杂度：$O(n^2)$.

3) 遇到问题

首先是先行还是先列，分析发现，行是反转，列是交换，反转的话，该行应该有一个参照行，但是如果后续还需要交换列，前面的翻转可能就无效了，容易混乱；所以采取先交换的原则，在需要时再反转。同时为了保证考虑的情况全面，将 b 中每一列均作为第一列尝试，在此基础上再交换列，这样条理清晰，如果不可交换列了说明此路不通，换 k+1 列，都不通则说明不可转换，输出-1。

其次是否把所有情况都遍历了：实际并不需要将所有情况遍历，只把可能无序的变换（且不会有循环的变换），转化为了有序的变换。

4) 源代码

```
#include<iostream>
#include<string.h>
using namespace std;

#define MAXN 105
const int inf = 0x3f3f3f3f; // 表示无解的情况
int a[MAXN][MAXN], b[MAXN][MAXN], tmp[MAXN][MAXN]; // 分别存储初始状态、目标状
```

```
态和临时状态
int t, Col, Row; // 测试案例数、列数、行数

/// @brief 检查两列金币的状态是否相同
/// @param num1 第一列的索引
/// @param num2 第二列的索引
/// @return 如果两列状态相同，则返回1；否则返回0
int check(int num1, int num2) {
    for (int i = 1; i <= Row; ++i) {
        if (a[i][num1] != b[i][num2])
            return 0;
    }
    return 1;
}

/// @brief 翻转指定行的金币状态
/// @param num 要翻转的行的索引
/// @details 将指定行的每个金币的状态进行翻转（0 变1，1 变0）
void changeRow(int num) {
    for (int i = 1; i <= Col; ++i) {
        b[num][i] ^= 1; // 通过异或操作实现状态翻转
    }
}

/// @brief 交换两列金币的位置，如果两列不同，则进行交换，并增加步数计数
/// @param num1 第一列的索引
/// @param num2 第二列的索引
/// @param step 操作步数的引用，用于记录总的操作步数
void changeCol(int num1, int num2, int&amp; step) {
    if (num1 == num2) return; // 如果列索引相同，则不需要操作
    step++;
    for (int i = 1; i <= Row; ++i) {
        int temp = b[i][num1];
        b[i][num1] = b[i][num2];
        b[i][num2] = temp;
    }
}

/// @brief 处理每个测试案例，计算达到目标状态的最小操作步数
/// @return 最小操作步数，如果无法达到目标状态则返回-1
int findMinStep() {
    int ans = inf;
    memcpy(tmp, b, sizeof(tmp));
    // 尝试每一列作为初始列进行列交换
    for (int k = 1; k <= Col; ++k) {
        int step = 0; // 当前尝试的步数
        memcpy(b, tmp, sizeof(tmp)); // 恢复b 状态
        changeCol(1, k, step); // 尝试将第一列与第k 列交换
        // 遍历每行，检查并翻转不匹配的行
        for (int i = 1; i <= Row; ++i) {
            if (a[i][1] != b[i][1]) {
                changeRow(i);
                step++;
            }
        }
        // 检查并尝试交换剩余的列
```

```
        int found = 1;
        for (int i = 1; i <= Col &amp;&amp; found; ++i) {
            found = 0;
            for (int j = i; j <= Col; ++j) {
                if (check(i, j)) {
                    changeCol(i, j, step);
                    found = 1;
                    break;
                }
            }
        }
        // 如果找到一个可行的方案，更新最小步数
        if (found &amp;&amp; step < ans) ans = step;
    }
    return ans < inf ? ans : -1; // 如果找到则返回最小步数，否则返回-1
}

int main() {
    cin >> t;
    while (t--) {
        cin >> Row >> Col;
        for (int i = 1; i <= Row; ++i)
            for (int j = 1; j <= Col; ++j)
                cin >> a[i][j];
        for (int i = 1; i <= Row; ++i)
            for (int j = 1; j <= Col; ++j)
                cin >> b[i][j];
        cout << findMinStep() << endl;
    }
    return 0;
}
```

四、 总结

第一章学习算法概述，主要是介绍算法的特点性质，及与程序的区别，程序是算法的具体实现，算法可以通过多种方式实现。同时还学习了描述算法效率的时间复杂性，渐进时间复杂性等，定量地分析算法的效率。