```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.*;

public class LatestVersion {

    /*** 获取 CPU 可用线程数*/
    static final int AVAILABLE_PROCESSORS = Runtime.getRuntime().availableProcessors(), REPETITION_TIMES = 8;
    /*** 模拟退火参数设置: 初始温度, 临界温度, 降温系数, 同一温度下重复次数, 同一用例处理次数.*/
    static final double INITIAL_TEMPERATURE = 1500.0, CRITICAL_TEMPERATURE = 1e-4, SA_RATIO = 0.99;
    static final int REPETITIONS_SAME_TEMPERATURE = 206, REPETITIONS_SAME_SAMPLE = REPETITION_TIMES * AVAILABLE_PROCESSORS;
    /*** 登山算法参数设置: 尝试攀登次数.*/
    static final int CLIMBING_TIMES = 300000;
    /*** 工件(行)数, 机器(列)数.*/
    static      int Row_Workpiece, Column_Machine;
    /*** 用例所在文件的路径, 保存答案的文件的路径*/
    static final File       FILE_INPUT  = new File("instances/SA9.txt");
    static final File       FILE_OUTPUT = new File("answer/ANS9.txt");
    static      Random     rand       = new Random();
    static      PrintWriter printWriter;

    /* 实例化 PrintWrite 类 */
    static {
        try {
            printWriter = new PrintWriter(FILE_OUTPUT);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    /*** 备份用例中各工件在各机器上需要的时间*/
    static      int[][]      data;
    /*** THREADS 列表保存正在运行的线程，用于线程管理；RESULT 列表保存每次计算得到的结果*/
    static final List<Thread>  THREADS = new ArrayList<>();
    static final List<Integer> RESULTS = new ArrayList<>();

    public static void main(String[] args) {

        long start = System.currentTimeMillis();

        try (Scanner scanner = new Scanner(FILE_INPUT)) {
            /* 数据初始化：输入工件数,机器数,时间表. */
            Row_Workpiece = scanner.nextInt();
            Column_Machine = scanner.nextInt();
            data = new int[Row_Workpiece][Column_Machine];

            for (int i = 0; i < Row_Workpiece; i++) {
                for (int j = 0; j < Column_Machine; j++) {
                    scanner.nextInt();
                    data[i][j] = scanner.nextInt();
                }
            }
            /* 检查用例是否读取完整 */
            if (scanner.hasNext()) {
                System.out.println("ERROR" + scanner.nextInt());
            } else {
                System.out.println("SUCCEED");
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        /* 开始多线程计算,以AMD Ryzen 7 4800H 为例，每次运行16 个线程，重复REPETITION_TIMES 次,共计得到REPETITIONS_SAME_SAMPLE 个结果 */
        for (int k = 0; k < REPETITIONS_SAME_SAMPLE; k++) {
            /* 创建一个线程并启动 */
            Thread thread = new Thread(new IndividualTask());
            thread.start();
            /* 将线程加入到用于现场管理的列表中 */
            THREADS.add(thread);
            /* 如果正在同时运行的线程数达到了 CPU 可用的最大线程数,等待当前正在处理的多个线程完成任务再添加新线程 */
            if (THREADS.size() % AVAILABLE_PROCESSORS == 0) {
                waitForThreads();
            }
        }

        long end = System.currentTimeMillis();

        printResult(start, end);

    }

    private static void waitForThreads() {
        for (Thread thread : LatestVersion.THREADS) {

            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        LatestVersion.THREADS.clear();
    }

    private static void printResult(long start, long end) {
        long totalMillisecond   = end - start;
        long currentMillisecond = totalMillisecond % 1000;

        long totalSecond   = totalMillisecond / 1000;
        long currentSecond = totalSecond % 60;

        long totalMinutes  = totalSecond / 60;
        long currentMinute = totalMinutes % 60;
        /* 输出程序总运行时间,格式:<分>:<秒>:<毫秒>.并将时间输出到控制台和输出文件中 */
        System.out.println("Time:" + currentMinute + ":" + currentSecond + ":" + currentMillisecond);
        printWriter.println("Time:" + currentMinute + ":" + currentSecond + ":" + currentMillisecond);
        /* 将输出结果按自然顺序排序(即升序),并输出到文件 */
        RESULTS.sort(Comparator.naturalOrder());
        RESULTS.forEach(printWriter::println);

        printWriter.close();
    }

    /*** 实现Runnable 接口， 以实现多线程*/
    static class IndividualTask implements Runnable {
        /* 当前工件顺序加工所需时间,随机交换两个工件顺序后加工所需时间 */
        int currentTime, nextTime;
        /* 随机交换的两个工件的索引 */
        int exchangeIndex1, exchangeIndex2;
        double t = INITIAL_TEMPERATURE;
        /* 实例化当前加工时间表和交换后的加工时间表 */
        private final int[][] current = new int[Row_Workpiece][Column_Machine];
        private final int[][] next    = new int[Row_Workpiece][Column_Machine];

        /* 实例化一个线程时初始化时间表 */
        public IndividualTask() {
            for (int i = 0; i < Row_Workpiece; i++) {
                for (int j = 0; j < Column_Machine; j++) {
                    current[i][j] = next[i][j] = data[i][j];
                }
            }
        }

        @Override /* 重写Runnable 接口中的run 方法,里面实现了多线程运行的代码 */
        public void run() {
            simulatedAnnealing(current, next);
    //        若要使用登山算法，注释上一行并取消注释下一行
    //          hillClimbing(current, next);
            /*将结果保存到RESULT 列表中,并在控制台输出当前线程的运行结果*/
            RESULTS.add(currentTime);
            System.out.println(Thread.currentThread().getName() + " " + currentTime);
        }

        /*** 随机加工顺序*/
        private void randomClimbing(int[][] current) {
            for (int i = Row_Workpiece; i > 0; i--) {
                exchangeIndex1 = rand.nextInt(i);
                exchangeIndex2 = i - 1;
                exchange(current);
            }
            currentTime = getMinTimeCost(current);
        }

        /* 登山算法 */
        private void hillClimbing(int[][] current, int[][] next) {
            currentTime = getMinTimeCost(current);

            for (int i = 0; i < CLIMBING_TIMES; i++) {
                randomExchange(next);
                nextTime = getMinTimeCost(next);
                /* 如果交换后工作时间更短,保存这种更改,否则还原这种更改. */
                if (nextTime - currentTime < 0) {
                    currentTime = nextTime;
                    exchange(current);
                } else {
                    exchange(next);
                }
            }
        }

        /*** 模拟退火算法*/
        private void simulatedAnnealing(int[][] current, int[][] next) {
            /* 改变后的工作时间与当前工作时间的差值,如果工件的加工顺序改变后工作时间变短,该值为负数 */
            int delta;
            currentTime = getMinTimeCost(current);

            while (t > CRITICAL_TEMPERATURE) {
                /* 每个温度下重复操作多次 */
                for (int num = 0; num < REPETITIONS_SAME_TEMPERATURE; num++) {
                    randomExchange(next);
                    nextTime = getMinTimeCost(next);

                    delta = nextTime - currentTime;

                    if (delta < 0) {
                        /* 如果交换工件加工顺序后加工时间更短了,保存这一有利改变 */
                        currentTime = nextTime;
                        exchange(current);
                    } else {
                        /* 否则生成一个[0,1)的随机数,以决定是否尝试这一"更坏"的加工顺序 */
                        double r = rand.nextDouble();
                        if (r < Math.exp(-delta / t)) {
                            /* 有 p=exp(-delta / t)的概率接受这一"更坏"的加工顺序 */
                            currentTime = nextTime;
                            exchange(current);
                        } else {
                            /* 否则恢复原先加工顺序 */
                            exchange(next);
                        }
                    }
                }
                t *= SA_RATIO;
            }
        }

        /*** 当前加工顺序下的最小工作时间为从时间表的左上角按"向下"和"向右"的方法走到时间表的右下角的最大值。典型的动态规划问题,采用动态规划求解，相关说
    明见报告*/
        private static int getMinTimeCost(final int[][] value) {
            int[][] dp = new int[Row_Workpiece][Column_Machine];
            dp[0][0] = value[0][0];
            /* 因为第一行的格子只能从第一行中此前的格子以"向右"的方法到达,第一列同理.避免循环中额外的判断,将第一行和第一列单独计算. */
            for (int i = 1; i < Row_Workpiece; i++) {
                dp[i][0] = dp[i - 1][0] + value[i][0];
            }
            for (int i = 1; i < Column_Machine; i++) {
                dp[0][i] = dp[0][i - 1] + value[0][i];
            }
            for (int i = 1; i < Row_Workpiece; i++) {
                for (int j = 1; j < Column_Machine; j++) {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) + value[i][j];
                }
            }
            /* 时间表右下角的值即为当前加工顺序下的最小工作时间 */
            return dp[Row_Workpiece - 1][Column_Machine - 1];
        }

        /*** 随机交换两个工件的加工顺序*/
        private void randomExchange(int[][] next) {
            do {
                exchangeIndex1 = rand.nextInt(Row_Workpiece);
                exchangeIndex2 = rand.nextInt(Row_Workpiece);
            } while (exchangeIndex1 == exchangeIndex2);
```

```java
            exchange(next);
        }

        /*** 交换两个工件的加工顺序,既用于next时间表的随机尝试,也用于current时间表保存next时间表的尝试,还用于next时间表恢复原状.*/
        private void exchange(int[][] next) {
            int temp;
            for (int i = 0; i < Column_Machine; i++) {
                temp = next[exchangeIndex1][i];
                next[exchangeIndex1][i] = next[exchangeIndex2][i];
                next[exchangeIndex2][i] = temp;
            }
        }
    }
}
```