```java
import java.util.*;

/**
 * @author DOCTORY
 */
public class Main {

    static int Row_Workpiece, Column_Machine;
    static int[][] timeTab;

    static final int GENERATION = 200;

    static final double POSSIBILITY_OF_REMAINING = 0.03, POSSIBILITY_OF_EXCHANGE = 0.91, POSSIBILITY_OF_VARIATION = 0.03,
POSSIBILITY_OF_SELFING = 0.03;

    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {

        System.out.println("请输入工件数,机器数.");
        Row_Workpiece = scanner.nextInt();
        Column_Machine = scanner.nextInt();
        timeTab = new int[Row_Workpiece][Column_Machine];
        /* 初始化 */
        System.out.println("请输入各工件在各机器上消耗的时间.");
        for (int i = 0; i < Row_Workpiece; i++) {
            for (int j = 0; j < Column_Machine; j++) {
                scanner.nextInt();
                timeTab[i][j] = scanner.nextInt();
            }
        }
        Chromosome.setGeneLength(Row_Workpiece);
        Chromosome.setGeneDepth(Column_Machine);
```

```
34
35          //创建初始随机染色体
36          for (int i = 0; i < Population.NUMBER_OF_SPECIES; i++) {
37              Chromosome individual = new Chromosome();
38              individual.initChromosome();
39              Population.chromosomes.add(individual);
40          }
41
42          Population.generateTurntable(Population.chromosomes);
43          Chromosome.show(Population.chromosomes);
44
45          System.out.println(Population.calibratedFitnessSum);
46
47          Population.findMaxFitnessValue(Population.chromosomes);
48          Population.setEpsilonMaxFitness();
49          /* 开始演化 */
50          for (int g = 0; g < GENERATION; g++) {
51              /* 重置染色体数组 */
52              Population.chromosomesSaved = Population.chromosomes;
53              Population.chromosomes = new ArrayList<>();
54
55              Population.generateTurntable(Population.chromosomesSaved);
56              for (int i = 0; i < Population.NUMBER_OF_SPECIES / 2; i++) {
57                  Population.selectParents();
58                  Chromosome.Crossover.crossover(
59                          Population.chromosomesSaved.get(Population.parentIndex1),
60                          Population.chromosomesSaved.get(Population.parentIndex1));
61              }
62
63              Chromosome.show(Population.chromosomes);
64              Population.iterateEpsilon();
65          }
66
```

```
67          Population.generateTurntable(Population.chromosomes);
68  //          Chromosome.show(Population.chromosomes);
69  //          System.out.println(Population.calibratedFitnessSum);
70  //          System.out.println(Population.epsilon);
71  //          System.out.println(Population.min);
72
73      }
74
75  }
```

```java
import java.util.*;

/**
 * @author DOCTORY
 */
public class Chromosome {

    /**
     * 生成一条新的染色体
     */
    public Chromosome() {
        this.geneData = new int[geneLength];
    }

    /**
     * 随机生成染色体片段，计算初始适应值
     */
    public void initChromosome() {
        initGeneData();
        calculateRawFitness();
    }

    private static int geneLength;
    private static int geneDepth;

    private int rawFitness;

    private int calibratedFitness;

    public int[] geneData;

    static Random random = new Random();

```

```java
    /**
     * 计算初始适应值
     */
    public void calculateRawFitness() {
        int[][] dp     = new int[geneLength][geneDepth];
        int[][] values = Main.timeTab;

        dp[0][0] = values[geneData[0]][0];

        for (int i = 1; i < geneLength; i++) {
            dp[i][0] = dp[i - 1][0] + values[geneData[i]][0];
        }
        for (int j = 1; j < geneDepth; j++) {
            dp[0][j] = dp[0][j - 1] + values[geneData[0]][j];
        }
        for (int i = 1; i < geneLength; i++) {
            for (int j = 1; j < geneDepth; j++) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) + values[geneData[i]][j];
            }
        }
        rawFitness = dp[geneLength - 1][geneDepth - 1];
    }


    /**
     * 此内部类提供交叉交换的方法
     */
    static class Crossover {

        public static void crossover(Chromosome parent1, Chromosome parent2) {
            cycleCrossover(parent1, parent2);
        }

```

```java
    /**
     * 循环交叉法生成子代,子代的基因的位置与父代相同
     */
    public static void cycleCrossover(Chromosome parent1, Chromosome parent2) {
        /* KEY: value, VALUE: index. */
        HashMap<Integer, Integer> map1 = new HashMap<>(geneLength);
        for (int i = 0; i < geneLength; i++) {
            map1.put(parent1.geneData[i], i);
        }
        Chromosome child1   = new Chromosome();
        Chromosome child2   = new Chromosome();
        boolean[]  visited  = new boolean[geneLength];
        boolean    onesTurn = true;
        /* 父代染色体交叉交换 */
        for (int i = 0; i < geneLength; i++) {
            if (visited[i]) {
                continue;
            }
            int j = i;
            do {
                visited[j] = true;
                if (onesTurn) {
                    child1.geneData[j] = parent1.geneData[j];
                    child2.geneData[j] = parent2.geneData[j];
                } else {
                    child1.geneData[j] = parent2.geneData[j];
                    child2.geneData[j] = parent1.geneData[j];
                }
                j = map1.get(parent2.geneData[j]);
            } while (j != i);
            onesTurn = !onesTurn;
        }
        /* 子代染色体以一定几率发生变异 */
```

```
            Random random = new Random();
            if (random.nextDouble() < 0.1) {
                child1.swap(random.nextInt(geneLength), random.nextInt(geneLength));
                child2.swap(random.nextInt(geneLength), random.nextInt(geneLength));
            }

            child1.calculateRawFitness();

            child2.calculateRawFitness();

            Population.chromosomes.add(child1);
            Population.chromosomes.add(child2);

        }

        public static void orderCrossover(Chromosome parent1, Chromosome parent2) {
        /*
         染色体交换片段
         Order Crossover (OX)
         若不控制随机数的值,有 p= 2/(geneLength)^2 的概率不发生交换, 即完全保留亲本.
        */
            int    rand1, rand2;
            boolean isNotValidExchange;
            do {
                rand1 = random.nextInt(geneLength);
                rand2 = random.nextInt(geneLength - rand1) + rand1;
                isNotValidExchange = (rand1 == 0 && rand2 == geneLength) || (rand1 == rand2 && parent1.geneData[rand1] ==
parent2.geneData[rand2]);
            } while (isNotValidExchange);

            Chromosome child1 = new Chromosome();
            Chromosome child2 = new Chromosome();

```

```
        boolean[] visited1 = new boolean[geneLength];
        boolean[] visited2 = new boolean[geneLength];

        for (int i = rand1; i <= rand2; i++) {
            child1.geneData[i] = parent1.geneData[i];
            visited1[parent1.geneData[i]] = true;
            child2.geneData[i] = parent2.geneData[i];
            visited2[parent2.geneData[i]] = true;
        }

        int target1 = 0, target2 = 0;

        for (int i = 0; i < geneLength; i++) {
            while (target1 >= rand1 && target1 <= rand2) {
                target1++;
            }
            while (target2 >= rand1 && target2 <= rand2) {
                target2++;
            }
            if (!visited1[parent2.geneData[i]]) {
                child1.geneData[target1] = parent2.geneData[i];
                target1++;
            }
            if (!visited2[parent1.geneData[i]]) {
                child2.geneData[target2] = parent1.geneData[i];
                target2++;
            }
        }
        /* 考虑变异 */

        child1.calculateRawFitness();
        child2.calculateRawFitness();

```

```
166             Population.chromosomes.add(child1);
167             Population.chromosomes.add(child2);
168 
169         }
170 
171         public static void microbialCrossover(Chromosome parent1, Chromosome parent2) {/* p1 适应值高, p2 适应值低 */
172             HashMap<Integer, Integer> map1 = new HashMap<>(geneLength);
173             for (int i = 0; i < geneLength; i++) {
174                 map1.put(parent1.geneData[i], i);
175             }
176             boolean[] visited = new boolean[geneLength];
177 
178             int index = 0;
179             do {
180                 visited[parent1.geneData[index]] = true;
181                 parent2.geneData[index] = parent1.geneData[index];
182 
183                 index = map1.get(parent2.geneData[index]);
184             } while (index != 0);
185 
186             int target=0;
187             for (int i = 0; i < geneLength; i++) {
188                 if (!visited[parent2.geneData[i]]) {
189                     parent2.geneData[target] = parent2.geneData[i];
190                     target++;
191                 }
192             }
193 
194             /* 考虑变异 */
195 
196             parent1.calculateRawFitness();
197             parent2.calculateRawFitness();
198             Population.chromosomes.add(parent1);
```

```
199             Population.chromosomes.add(parent2);
200 
201         }
202     }
203 
204     /**
205      * 此内部类提供变异的方法
206      */
207     class Mutate {
208 
209         public void mutate() {
210 
211         }
212 
213         static Random rand = new Random();
214 
215         /**
216          * 染色体随机交换片段
217          */
218         public void ectopicMutate() {
219             final double ECTOPIC_POSSIBILITY = 0.5;
220             do {
221                 int random1 = rand.nextInt(geneLength);
222                 int random2 = rand.nextInt(geneLength);
223                 swap(random1, random2);
224             } while (rand.nextDouble() < ECTOPIC_POSSIBILITY);
225         }
226 
227         /**
228          * 染色体倒转
229          */
230         public void reversedMutate() {
231             for (int i = 0; i < geneLength / 2; i++) {
```

```
            swap(i, geneLength - 1 - i);
        }
    }

    /**
     * 剪切染色体后重新拼接
     */
    public void shearMutate() {
        int cutPoint;
        do {
            cutPoint = rand.nextInt(geneLength);
        } while (cutPoint == 0);
        int[] geneDataSaved = geneData;
        geneData = new int[geneLength];
        int index = 0;
        for (int i = cutPoint; i < geneLength; i++) {
            geneData[index++] = geneDataSaved[i];
        }
        for (int i = 0; i < cutPoint; i++) {
            geneData[index++] = geneDataSaved[i];
        }

    }

    /**
     * 随机重置染色体上的所有基因
     */
    public void resetMutate() {
        for (int i = geneLength; i > 0; i--) {
            swap(random.nextInt(i), i - 1);
        }
    }

```

```
265
266     }
267
268
269     /**
270      * 打印染色体信息
271      */
272     public static void show(List<Chromosome> chromosomeList) {
273         for (Chromosome ch : chromosomeList) {
274             int    m = Main.Row_Workpiece, n = Main.Column_Machine;
275             int[][] t = new int[m][n], a = Main.timeTab;
276             for (int i = 0; i < m; i++) {
277                 for (int j = 0; j < n; j++) {
278                     t[i][j] = a[ch.geneData[i]][j];
279                 }
280             }
281             int[][] dp = new int[m][n];
282             dp[0][0] = t[0][0];
283             /* 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. */
284             for (int i = 1; i < m; i++) {
285                 dp[i][0] = dp[i - 1][0] + t[i][0];
286             }
287             for (int i = 1; i < n; i++) {
288                 dp[0][i] = dp[0][i - 1] + t[0][i];
289             }
290             for (int i = 1; i < m; i++) {
291                 for (int j = 1; j < n; j++) {
292                     dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) + t[i][j];
293                 }
294             }
295             /* 时间表右下角的值即为当前加工顺序下的最小工作时间 */
296             int v = dp[m - 1][n - 1];
297             if (v < Population.min) {
```

```
298                 Population.min = v;
299
300             }
301 //              System.out.println(" fitness: " + ch.calibratedFitness + " geneData: " + Arrays.toString(ch.geneData) + " value: " +
302 v);
303         }
304         System.out.println(Population.min);
305     }
306
307     public void swap(int i, int j) {
308         /* 交换 */
309         int temp = geneData[i];
310         geneData[i] = geneData[j];
311         geneData[j] = temp;
312     }
313
314     /**
315      * 随机初始化染色体基因序列
316      */
317     public void initGeneData() {
318         for (int i = 0; i < geneLength; i++) {
319             geneData[i] = i;
320         }
321         for (int i = geneLength; i > 0; i--) {
322             swap(random.nextInt(i), i - 1);
323         }
324     }
325
326     public static void setGeneLength(int geneLength) {
327         Chromosome.geneLength = geneLength;
328     }
329
330     public static void setGeneDepth(int geneDepth) {
```

```
        Chromosome.geneDepth = geneDepth;
    }

    public int getCalibratedFitness() {
        return calibratedFitness;
    }

    public int getRawFitness() {
        return rawFitness;
    }

    public void setCalibratedFitness(int calibratedFitness) {
        this.calibratedFitness = calibratedFitness;
    }

}
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

/**
 * @author DOCTORY
 */
public class Population {

    public static final int    NUMBER_OF_SPECIES = 40;
    public static final double ITERATION        = 0.99;
    public static final double EPSILON          = 0;
    public static      double epsilon          = EPSILON;

    public static int min=9999999;


    public static int maxFitnessValue;
    public static int calibratedFitnessSum;

    public static int[] turntable = new int[NUMBER_OF_SPECIES];

    static List<Chromosome> chromosomes     = new ArrayList<>();
    static List<Chromosome> chromosomesSaved = new ArrayList<>();

    public static int parentIndex1, parentIndex2;

    static Random random = new Random();

    /**
     * 找到最大的初始值
     */
    public static void findMaxFitnessValue(List<Chromosome> chromosomeList) {
```

```
        for (Chromosome ch : chromosomeList) {
            if (ch.getRawFitness() > maxFitnessValue) {
                maxFitnessValue = ch.getRawFitness();
            }
        }
    }

    /**
     * 标定所有染色体的适应值
     */
    public static void calibrateFitness(List<Chromosome> chromosomeList) {
        for (Chromosome ch : chromosomeList) {
            ch.setCalibratedFitness(maxFitnessValue - ch.getRawFitness() + (int) epsilon);
        }
    }

    /**
     * 计算标定适应值之和
     */
    public static void calculateCalibratedFitnessValueSum(List<Chromosome> chromosomeList) {
        calibratedFitnessSum = 0;
        for (Chromosome ch : chromosomeList) {
            calibratedFitnessSum += ch.getCalibratedFitness();
        }
    }

    /**
     * 根据标定适应值生成转盘, 完成了1.寻找最大初始适应值;2.标定适应值;3.生成转盘;4.计算标定适应值之和.
     */
    public static void generateTurntable(List<Chromosome> chromosomeList) {
        findMaxFitnessValue(chromosomeList);
        calibrateFitness(chromosomeList);
        int value = 0;
```

```
67          for (int i = 0; i < chromosomeList.size(); i++) {
68              value += chromosomeList.get(i).getCalibratedFitness();
69              turntable[i] = value;
70          }
71          calibratedFitnessSum = value;
72      }
73
74      /**
75       * 使用转盘正比例选择双亲
76       */
77      public static void selectParents() {
78          int rand1 = random.nextInt(calibratedFitnessSum);
79          int rand2 = random.nextInt(calibratedFitnessSum);
80          parentIndex1 = 0;
81          parentIndex2 = 0;
82
83          /* 查找双亲 */
84          boolean parent1Found = false;
85          boolean parent2Found = false;
86          while (!(parent1Found && parent2Found)) {
87              if (!parent1Found && turntable[parentIndex1] < rand1) {
88                  parentIndex1++;
89              } else {
90                  parent1Found = true;
91              }
92              if (!parent2Found && turntable[parentIndex2] < rand2) {
93                  parentIndex2++;
94              } else {
95                  parent2Found = true;
96              }
97          }
98
99      }
```

```
100
101     public static void iterateEpsilon() {
102         epsilon *= ITERATION;
103     }
104
105     public static void setEpsilonMaxFitness() {
106         epsilon = 1000;
107     }
108 }
109
```