Computer Project

(2017-2019)

Satvik Saha

Class: XI B

Roll number: 24

"Writing code a computer can understand is science. Writing code other programmers can understand is an art."

— **Jason Gorman**

"When in doubt, use brute force."

— **Ken Thompson**

Problem 1 An n digit integer $(a_1 a_2 \dots a_n)$, where each digit $a_i \in \{0, 1, \dots, 9\}$, is said to have *unique digits* if no digits are repeated, i.e., there is no i, j such that $a_i = a_j$ $(i \neq j)$.

Verify whether an inputted number has *unique digits*.

Solution

Source Code

```
1  public class Unique {
2         public static void main (String[] args) {
3                 try {
4                         long number = Long.parseLong(args[0]);
5                         if (isUnique(number)) {
6                                 System.out.println("Unique Number!");
7                         } else {
8                                 System.out.println("Not a Unique Number!");
9                         }
10                } catch (NumberFormatException e) {
11                        System.out.println("Enter an integer as the first argument!");
12                } catch (IndexOutOfBoundsException e) {
13                        System.out.println("Enter 1 argument (integer)!");
14                }
15        }
16
17        public static boolean isUnique (long number) {
18                int[] count = new int[10];
19                for (long n = Math.abs(number); n > 0; n /= 10) {
20                        int digit = (int) n % 10;
21                        count[digit]++;
22                        if (count[digit] > 1){
23                                return false;
24                        }
25                }
26                return true;
27        }
28 }
```

"Elegance is not a dispensable luxury but a factor that decides between success and failure."

— **Edsger W. Dijkstra**

Problem 2 A *partition* of a positive integer n is defined as a collection of other positive integers such that their sum is equal to n. Thus, if $(a_1, a_2, \ldots, a_k)$ is a partition of n,

$$n = a_1 + a_2 + \cdots + a_k \qquad (a_i \in \mathbb{Z}^+)$$

Display every *unique partition* of an inputted number.

Solution

Source Code

```
1
2  public class Partition {
3          public static void main (String[] args) {
4                  try {
5                          int target = Integer.parseInt(args[0]);
6                          if (target < 1) {
7                                  throw new NumberFormatException();
8                          }
9                          partition(target);
10                 } catch (NumberFormatException e) {
11                         System.out.println("Enter a natural number as the first
                               argument!");
12                 } catch (IndexOutOfBoundsException e) {
13                         System.out.println("Enter 1 argument (natural number)!");
14                 }
15         }
16         public static void partition (int target) {
17                 partition(target, target, "");
18         }
19         public static void partition (int target, int previousTerm, String suffix) {
20                 if (target == 0)
21                         System.out.println(suffix);
22                 for (int i = 1; i <= target && i <= previousTerm; i++)
23                         partition(target - i, i, suffix + " " + i);
24         }
25 }
```

"Simplicity is the ultimate sophistication."

— **Leonardo da Vinci**

Problem 3 A *Caesar cipher* is a type of monoalphabetic substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. The positions are circular, i.e., after reaching Z, the position wraps around to A. For example, following is some encrypted text, using a right shift of 5.

```
Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:  FGHIJKLMNOPQRSTUVWXYZABCDE
```

Thus, after mapping the alphabet according to the scheme $A \mapsto 0, B \mapsto 1, \ldots, Z \mapsto 23$, we can define an encryption function E_n, in which a letter x is shifted rightwards by n as follows.

$$E_n(x) \;=\; (x + n) \mod 26$$

The corresponding decryption function D_n is simply

$$D_n(x) \;=\; (x - n) \mod 26$$

Implement a simple version of a *Caesar cipher*, encrypting capitalized plaintext by shifting it by a given value. Interpret positive shifts as rightwards, negative as leftwards.

Solution

Source Code

```
1  public class CaesarShift {
2         public static void main (String[] args) {
3                 try {
4                         int shift = Integer.parseInt(args[0]) % 26;
5                         String plaintext = args[1].toUpperCase();
6                         String ciphertext = "";
7                         for (int i = 0; i < plaintext.length(); i++) {
8                                 char plain = plaintext.charAt(i);
9                                 char crypt = ' ';
10                                if ('A' <= plain && plain <= 'Z') {
11                                        crypt = numToChar(charToNum(plain) + shift);
12                                } else {
13                                        crypt = plain;
14                                }
15                                ciphertext += crypt;
```

```
                }
                System.out.println(ciphertext);
            } catch (NumberFormatException e) {
                System.out.println("Enter an integer as the first argument!");
            } catch (IndexOutOfBoundsException e) {
                System.out.println("Enter 2 arguments (shift, plaintext)!");
            }
        }

        public static int charToNum (char letter) {
            return Character.toUpperCase(letter) - 'A';
        }

        public static char numToChar (int number) {
            return (char) ('A' + Math.floorMod(number, 26));
        }
}
```

"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."

— **Leon Bambrick**

Problem 4 A *palindrome* is a sequence of characters which reads the same backwards as well as forwards. For example, `madam`, `racecar` and `kayak` are words which are palindromes. Similarly, the sentence "`A man, a plan, a canal -- Panama!`" is also a plaindrome.

Analyze a sentence of input and display all *words* which are palindromes. If the entire *sentence* is also a palindrome, display it as well.

(A word is an unbroken sequence of characters, separated from other words by whitespace. Ignore single letter words such as I *and* a*. Ignore punctuation, numeric digits, whitespace and case while analyzing the entire sentence.)*

Solution

Source Code

```
1  import java.util.Scanner;
2
3  public class Palindrome {
4         public static void main (String[] args) {
5                System.out.print("Enter your sentence : ");
6                String sentence = (new Scanner(System.in)).nextLine().trim();
7                boolean foundPalindrome = false;
8                System.out.println("Palindromes : ");
9                foundPalindrome |= checkWords(sentence);
10               foundPalindrome |= checkSentence(sentence);
11               if (!foundPalindrome) {
12                      System.out.println("(No palindromes found!)");
13               }
14        }
15
16        public static boolean checkWords (String sentence) {
17               boolean foundPalindrome = true;
18               int start = -1;
19               int end = 0;
20               while (end < sentence.length()) {
21                      while (Character.isWhitespace(sentence.charAt(++start)));
22                      end = start;
```

```
23                      while (end < sentence.length() &&
                            !Character.isWhitespace(sentence.charAt(end++)));
24                      String word = sentence.substring(start, end).trim();
25                      if (isPalindrome(word)) {
26                              foundPalindrome = true;
27                              System.out.println(getAlphabets(word));
28                      }
29                      start = end - 1;
30              }
31              return foundPalindrome;
32      }
33
34      public static boolean checkSentence (String sentence) {
35              if (isPalindrome(sentence)) {
36                      System.out.println(sentence);
37                      return true;
38              }
39              return false;
40      }
41
42      public static boolean isPalindrome (String text) {
43              String rawText = getAlphabets(text).toUpperCase();
44              for (int i = 0, j = rawText.length() - 1; i < j; i++, j--) {
45                      if (rawText.charAt(i) != rawText.charAt(j)) {
46                              return false;
47                      }
48              }
49              return (rawText.length() > 1);
50      }
51
52      public static String getAlphabets (String text) {
53              String rawText = "";
54              for (int i = 0; i < text.length(); i++) {
55                      if (Character.isAlphabetic(text.charAt(i))) {
56                              rawText += text.charAt(i);
57                      }
58              }
59              return rawText;
60      }
61  }
```

"Any fool can use a computer. Many do."

— **Ted Nelson**

Problem 5 Design a simple interface for an examiner which can format and display marks scored by a group of students in a particular examination. Calculate the percentage scored by each candidate and display the list of students and percentages in an ASCII bar chart, arranged alphabetically.

Solution

Source Code

```
1  public class Marksheet {
2          public static final int SCREEN_WIDTH = 100;
3          private final double maxMarks;
4          private final int numberOfStudents;
5          private int lastStudent;
6          private String[] names;
7          private double[] marks;
8
9          public Marksheet (double maxMarks, int numberOfStudents) {
10                 this.maxMarks = maxMarks;
11                 this.numberOfStudents = numberOfStudents;
12                 this.names = new String[numberOfStudents];
13                 this.marks = new double[numberOfStudents];
14                 this.lastStudent = -1;
15         }
16
17         public boolean addMarks (String name, double score) {
18                 try {
19                         names[++lastStudent] = name;
20                         marks[lastStudent] = score;
21                         return true;
22                 } catch (IndexOutOfBoundsException e) {
23                         return false;
24                 }
25         }
26
27         public void displayChart () {
28                 System.out.println(Marksheet.multiplyString("-",
                       Marksheet.SCREEN_WIDTH));
29                 for (int i = 0; i <= lastStudent; i++) {
30                         double fraction = marks[i] / maxMarks;
```

```
31                  String name = (names[i].length() < 16)
32                                ? names[i]
33                                : (names[i].substring(0,13) + "...");
34                  int points = (int) (fraction * (SCREEN_WIDTH - 34));
35                  String bar = multiplyString("*", points)
36                         + multiplyString(" ", SCREEN_WIDTH - 34 - points);
37                  System.out.printf("| %16s | %s | %6.2f %% |%n"
38                                                , name
39                                                , bar
40                                                , fraction * 100);
41          }
42          System.out.println(Marksheet.multiplyString("-",
                Marksheet.SCREEN_WIDTH));
43      }
44
45      public void displayMaxScorers () {
46          String maxScorers = "";
47          double maxScore = getMaxScore();
48          for (int i = 0; i <= lastStudent; i++) {
49                  if (marks[i] == maxScore) {
50                          maxScorers += ", " + names[i];
51                  }
52          }
53          System.out.println(maxScorers.substring(1)
54                                  + " scored the highest ("
55                                  + maxScore + "/"
56                                  + maxMarks + ")");
57      }
58
59      public void sortByName () {
60          for (int right = lastStudent; right > 0; right--)
61                  for (int i = 1; i <= right; i++)
62                          if (names[i-1].compareToIgnoreCase(names[i]) > 0)
63                                  swapRecords(i, i - 1);
64
65      }
66
67      public double getMaxScore () {
68          double max = Integer.MIN_VALUE;
69          for (int i = 0; i <= lastStudent; i++) {
70                  max = Math.max(max, marks[i]);
71          }
72          return max;
73      }
74
75      private void swapRecords (int x, int y) {
```

```
76                String tempName = names[x];
77                double tempMark = marks[x];
78                names[x] = names[y];
79                marks[x] = marks[y];
80                names[y] = tempName;
81                marks[y] = tempMark;
82        }
83
84        public static String multiplyString (String s, int n) {
85                String out = "";
86                while (n --> 0)
87                        out += s;
88                return out;
89        }
90  }
```

```
1   import java.util.Scanner;
2   import java.util.InputMismatchException;
3
4   public class ScoreRecorder {
5         public static void main (String[] args) {
6                 Scanner inp = new Scanner(System.in);
7                 double maxMarks = 0.0;
8                 int numberOfStudents = 0;
9                 try {
10                        System.out.print("Enter the maximum marks alotted for each
                              student : ");
11                        maxMarks = inp.nextDouble();
12                        System.out.print("Enter the total number of students : ");
13                        numberOfStudents = inp.nextInt();
14                        if (maxMarks <= 0) {
15                              System.out.println("Maximum marks must be positive!");
16                              System.exit(0);
17                        }
18                        if (numberOfStudents <= 0) {
19                              System.out.println("Number of students must be
                                    positive!");
20                              System.exit(0);
21                        }
22                        Marksheet sheet = new Marksheet(maxMarks, numberOfStudents);
23                        System.out.println("Enter " + numberOfStudents + " students'
                              names and marks : ");
24                        for (int i = 0; i < numberOfStudents; i++) {
25                              String name = "";
26                              while (!inp.hasNextDouble()) {
```

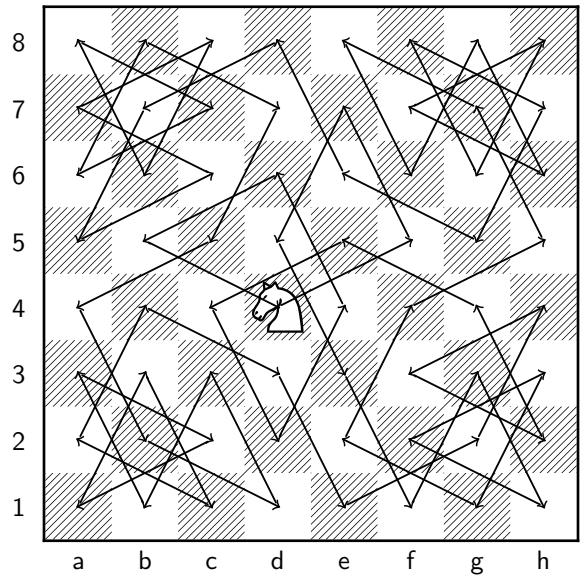
```
27                                      name += inp.next() + " ";
28                              }
29                              double marks = inp.nextDouble();
30                              if (marks <= 0 || marks > maxMarks) {
31                                      System.out.println("Marks must be within 0.0 and
                                            " + maxMarks + "!");
32                                      System.exit(0);
33                              }
34                              sheet.addMarks(name.trim(), marks);
35                      }
36                      sheet.sortByName();
37                      sheet.displayChart();
38                      sheet.displayMaxScorers();
39              } catch (InputMismatchException e) {
40                      System.out.println("Invalid Input!");
41                      System.exit(0);
42              }
43      }
44  }
```

"My project is 90% done. I hope the second half goes as well."

— **Scott W. Ambler**

Problem 6 A *Knight's Tour* is a sequence of moves of a knight on a chessboard such that the *knight* visits every square only once. If the knight ends on a square that is one knight's move from the beginning square, the tour is *closed* forming a closed loop, otherwise it is *open*.

There are many ways of constructing such paths on an empty board. On an 8×8 board, there are no less than $26,534,728,821,064$ *directed*[1] *closed* tours. Below is one of them.

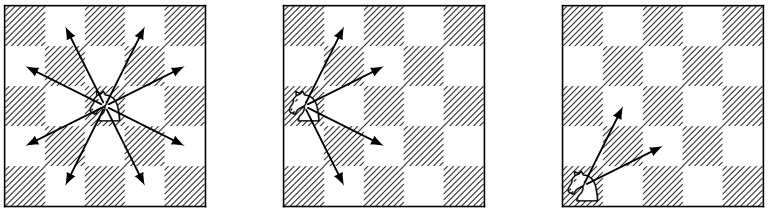


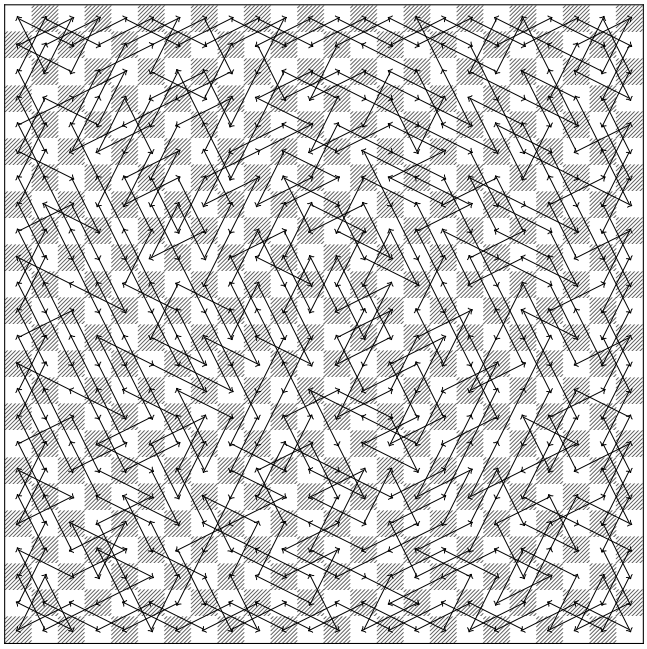
Construct a *Knight's Tour* (*open* or *closed*) on an $n \times n$ board, starting from a given square.

(Mark each square with the move number on which the knight landed on it. Mark the starting square 1*.)*

[1]Two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections.

Solution A knight on a chessboard can move to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally.





Source Code

```
public class TourSolver {
        private final int size;
        private Position[] path;
        private int numberOfMoves;
        private int[][] board;
        private int[][] degreesOfFreedom;
        private Position initPosition;
        private double tieBreakRandomness;
```

```
9
10      private static final int[][] KNIGHT_MOVES = {
11              {-1, -2}, {-1, 2}, {1, -2}, {1, 2},
12              {-2, -1}, {-2, 1}, {2, -1}, {2, 1}
13      };
14
15      public TourSolver (int size, Position initPosition, double randomness) {
16              this.size = size;
17              this.initPosition = initPosition;
18              this.tieBreakRandomness = randomness / 2.0;
19              this.path = new Position[size * size];
20              this.numberOfMoves = 0;
21              this.initBoard();
22              this.initDegreesOfFreedom();
23      }
24
25      public void resetSolution () {
26              this.path = new Position[size * size];
27              this.numberOfMoves = 0;
28              this.initBoard();
29      }
30
31      private void initBoard () {
32              board = new int[size][size];
33              for (int i = 0; i < size; i++)
34                      for (int j = 0; j < size; j++)
35                              board[i][j] = 0;
36      }
37
38      private void initDegreesOfFreedom () {
39              degreesOfFreedom = new int[size][size];
40              for (int i = 0; i < size; i++)
41                      for (int j = 0; j < size; j++)
42                              degreesOfFreedom[i][j] = getPossibleMovesCount(new
                                    Position(i, j));
43      }
44
45      public boolean addMove (Position p) {
46              if (numberOfMoves < (size * size)) {
47                      path[numberOfMoves++] = p;
48                      board[p.getX()][p.getY()] = numberOfMoves;
49                      return true;
50              }
51              return false;
52      }
53
```

```
54          public boolean removeMove () {
55                  if (numberOfMoves > 0) {
56                          Position p = path[numberOfMoves - 1];
57                          board[p.getX()][p.getY()] = 0;
58                          path[--numberOfMoves] = null;
59                          return true;
60                  }
61                  return false;
62          }
63
64          public int[][] getBoard () {
65                  return board;
66          }
67
68          public Position[] getSolution () {
69                  if (size < 5)
70                          return null;
71                  addMove(initPosition);
72                  if(solve(initPosition))
73                          return path;
74                  return null;
75          }
76
77          public boolean solve (Position p) {
78                  if (numberOfMoves == (size * size))
79                          return true;
80                  Position[] possibleMoves = getPossibleMoves(p);
81                  if (possibleMoves[0] == null)
82                          return false;
83                  sortMoves(possibleMoves);
84                  for (Position move : possibleMoves) {
85                          if (move != null) {
86                                  addMove(move);
87                                  if (solve(move))
88                                          return true;
89                                  removeMove();
90                          }
91                  }
92                  return false;
93          }
94
95          public void sortMoves (Position[] moves) {
96                  int count = 0;
97                  for (Position p : moves)
98                          if (p != null)
99                                  count++;
```

```
100             for (int right = count; right > 0; right--)
101                     for (int i = 1; i < right; i++)
102                             if (compareMoves(moves[i-1], moves[i]) > 0)
103                                     swapMoves(i-1, i, moves);
104     }
105
106     public int compareMoves (Position a, Position b) {
107             int aCount = getPossibleMovesCount(a);
108             int bCount = getPossibleMovesCount(b);
109             if (aCount != bCount)
110                     return aCount - bCount;
111             int aFree = degreesOfFreedom[a.getX()][a.getY()];
112             int bFree = degreesOfFreedom[b.getX()][b.getY()];
113             if (aFree != bFree)
114                     return aFree - bFree;
115             return (Math.random() < tieBreakRandomness)? 1 : -1;
116     }
117
118     private static void swapMoves (int x, int y, Position[] moves) {
119             Position t = moves[x];
120             moves[x] = moves[y];
121             moves[y] = t;
122     }
123
124     public Position[] getPossibleMoves (Position start) {
125             Position[] possibleMoves = new Position[KNIGHT_MOVES.length];
126             int i = 0;
127             for (int[] move : KNIGHT_MOVES) {
128                     int x = start.getX() + move[0];
129                     int y = start.getY() + move[1];
130                     if (isWithinBoard(x, y) && board[x][y] == 0) {
131                             possibleMoves[i++] = new Position(x, y);
132                     }
133             }
134             return possibleMoves;
135     }
136
137     public int getPossibleMovesCount (Position start) {
138             int i = 0;
139             for (Position p : getPossibleMoves(start))
140                     if (p != null)
141                             i++;
142             return i;
143     }
144
145     public boolean isWithinBoard (int x, int y) {
```

```
146                 return (x >= 0 && x < size && y >= 0 && y < size);
147         }
148 }
```

```
1  public class Position {
2         private final int x;
3         private final int y;
4
5         public Position (int x, int y) {
6                 this.x = x;
7                 this.y = y;
8         }
9
10        public Position (String s) {
11                int x = 0;
12                int i = 0;
13                while (i < s.length() && Character.isAlphabetic(s.charAt(i))) {
14                        x = (x * 26) + Character.toLowerCase(s.charAt(i)) - 'a' + 1;
15                        i++;
16                }
17                int y = Integer.parseInt(s.substring(i));
18                this.x = x - 1;
19                this.y = y - 1;
20        }
21
22        public int getX () {
23                return x;
24        }
25
26        public int getY () {
27                return y;
28        }
29
30        public boolean equals (Position p) {
31                return (p != null)
32                        && (this.getX() == p.getX()) && (this.getY() == p.getY());
33        }
34
35        @Override
36        public String toString () {
37                return xToString(this.x) + (this.y + 1);
38        }
39
40        public static String xToString (int n) {
41                int x = n + 1;
```

```
                String letters = "";
                while (x > 0) {
                        letters = (char) ('a' + (--x % 26)) + letters;
                        x /= 26;
                }
                return letters;
        }
}
```

```
public class KnightTour {
        public static void main (String[] args) {
                try {
                        int boardSize = Integer.parseInt(args[0]);
                        if (boardSize <= 0)
                                throw new NumberFormatException();
                        String initSquare = (args.length > 1)? args[1] : "a1";
                        double randomness = (args.length > 2)?
                            Double.parseDouble(args[2])
                                                        : Math.pow(0.8, boardSize) * 2;
                        TourSolver t = new TourSolver(boardSize, new
                            Position(initSquare), randomness);
                        Position[] solution = t.getSolution();
                        if (solution != null) {
                                showBoard(t.getBoard());
                                showMoves(solution);
                                if (isClosed(solution))
                                        System.out.println("\nThe tour is Closed!");
                        } else {
                                System.out.println("No Knight's Tours found!");
                        }
                } catch (Exception e) {
                        System.out.print("Enter an integer (> 1) as the first
                            argument, ");
                        System.out.println("and a well formed chessboard coordinate as
                            the second!");
                        System.out.println("                              (size,
                            startSquare * , randomness * )");
                        System.out.println();
                        System.out.println("(size     -> Solve a Tour on a (size x
                            size) board)");
                        System.out.println("(startSquare * -> A square in algebraic
                            chess notation of the form 'fr',");
                        System.out.println("            where f = the letter
                            representing the file(column)");
                        System.out.println("            and  r = the number
```

```
                    representing the rank(row).)");
29                  System.out.println("(startSquare is set to 'a1' by default)");
30                  System.out.println("(randomness * -> A number between 0(no
                        randomness) and 1(even chances),");
31                  System.out.println("            determining the randomness in
                        ranking positions of");
32                  System.out.println("            the same weightage while
                        searching. A randomness of 0 will");
33                  System.out.println("            produce the same tour every
                        time, for a specific size and");
34                  System.out.println("            startSquare. Keep extremely
                        small values of randomness for");
35                  System.out.println("            very large boards.)");
36                  System.out.println("(randomness is set to 2 * (0.8)^boardSize
                        by default)");
37                  System.out.println();
38                  System.out.println("                                          <
                        * = optional arguments >");
39          }
40      }
41
42      public static void showBoard (int[][] board) {
43          String hLine = " " + multiplyString("+-----", board.length) + "+";
44          System.out.println(hLine);
45          for (int column = board.length - 1; column >= 0; column--) {
46              System.out.printf(" %2d ", column + 1);
47              for (int row = 0; row < board.length; row++) {
48                  System.out.printf("| %3d ", board[row][column]);
49              }
50              System.out.printf("|%n%s%n", hLine);
51          }
52          System.out.print(" ");
53          for (int i = 0; i < board.length; i++) {
54              System.out.printf(" %2s ", Position.xToString(i));
55          }
56          System.out.println();
57      }
58
59      public static void showMoves (Position[] moves) {
60          System.out.print("\nMoves : ");
61          String movesOut = "";
62          for (int i = 1; i < moves.length; i++) {
63              movesOut += (moves[i-1] + "-" + moves[i] + ", ");
64          }
65          System.out.println(movesOut.substring(0, movesOut.length() - 2));
66      }
```

```
67
68          public static String multiplyString (String s, int n) {
69                  String result = "";
70                  while (n --> 0)
71                          result += s;
72                  return result;
73          }
74
75          public static boolean isClosed (Position[] path) {
76                  int l = path.length - 1;
77                  int dX = Math.abs(path[0].getX() - path[l].getX());
78                  int dY = Math.abs(path[0].getY() - path[l].getY());
79                  return (dX == 1 && dY == 2) || (dX == 2 && dY == 1);
80          }
81  }
```

"To iterate is human, to recurse divine"

— L. Peter Deutsch

Problem 7 The *determinant* of a square matrix $A_{n,n}$ is defined recursively as follows.

$$det(A_{n,n}) \;=\; \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} \;=\; \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \cdot det(M_{i,j})$$

where $M_{i,j}$ is defined as the minor of $A_{n,n}$, an $(n-1) \times (n-1)$ matrix formed by removing the ith row and jth column from $A_{n,n}$.

The determinant of a (2×2) matrix is simply given by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \;=\; ad \;-\; bc$$

For example, the determinant of a (3×3) matrix is given by the following expression.

$$\begin{aligned} \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \;&=\; a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &=\; aei + bfg + cdh - ceg - bdi - afh \end{aligned}$$

Calculate the *determinant* of an inputted $(n \times n)$ square matrix.

Solution