

```
1 /* Features a function that prints the decimal value of a given integer value. */
2 public class IntegerToBinary {
3
4     public static void main(String[] args) {
5         integerToBinary(Integer.parseInt(args[0]));
6         System.out.println();
7     }
8
9     public static void integerToBinary(int n) {
10         if (n == 1 || n == 0) {
11             System.out.print(n);
12         } else {
13             integerToBinary(n / 2);
14             System.out.print(n % 2);
15         }
16     }
17 }
```

```
1 /** Reads a command line string and checks if it's a palindrome. */
2 public class Palindrome {
3
4     public static void main(String[] args) {
5         System.out.println(isPalindrome(args[0]));
6     }
7
8     public static boolean isPalindrome(String s) {
9         if (s.length() == 1) {
10             return true;
11         } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
12             if (isPalindrome(s.substring(1, s.length() - 1))) {
13                 return true;
14             }
15         }
16         return false;
17     }
18 }
```

```

1  /* Recieves two command line integers, n and k, and returns the respective binomial
   coefficient.
2      Uses memoization to optimize the recursive process. */
3  public class Binomial {
4
5      public static void main(String[] args) {
6          System.out.println(binomial(Integer.parseInt(args[0]), Integer.parseInt(args[1])));
7      }
8
9      // Computes and returns the Binomial coefficient
10     public static long binomial(int n, int k) {
11         long[][] memo = new long[n + 1][k + 1];
12         if (k > n) {
13             return 0;
14         }
15         if (k == 0 || n == 0) {
16             return 1;
17         }
18         return (binomial(n - 1, k, memo) + binomial(n - 1, k - 1, memo));
19     }
20
21     public static long binomial(int n, int k, long[][] memo) {
22         if (k > n) {
23             return 0;
24         }
25         if (k == 0 || n == 0) {
26             return 1;
27         }
28         if (memo[n][k] == 0) {
29             memo[n][k] = binomial(n - 1, k, memo) + binomial(n - 1, k - 1, memo);
30         }
31         return memo[n][k];
32     }
33 }

```

```

1 /** Prints the Sierpinski Triangle fractal. */
2 public class Sierpinski {
3
4     public static void main(String[] args) {
5         sierpinski(Integer.parseInt(args[0]));
6     }
7
8     // Draws a Sierpinski triangle of depth n on the standard canvass.
9     public static void sierpinski(int n) {
10         double s = Math.sqrt(3) / 2;
11         StdDraw.line(0, 0, 1, 0);
12         StdDraw.line(0.5, s, 0, 0);
13         StdDraw.line(1, 0, 0.5, s);
14         sierpinski(n, 0, 1, 0.5, 0, 0, s);
15     }
16
17     public static void sierpinski(int n, double x1, double x2, double x3,
18 , double y1, double y2, double y3) {
19         if (n == 0) {
20             return;
21         }
22         StdDraw.line((x1 + x2) / 2, (y1 + y2) / 2, (x1 + x3) / 2, (y1 + y3) / 2); // ! left
diagonal
23         StdDraw.line((x1 + x2) / 2, (y1 + y2) / 2, (x2 + x3) / 2, (y1 + y3) / 2); // !
right line
24         StdDraw.line((x1 + x3) / 2, (y1 + y3) / 2, (x2 + x3) / 2, (y1 + y3) / 2); // !
straight line
25         sierpinski(n - 1, x1, (x1 + x2) / 2, (x1 + x3) / 2, y1, (y1 + y2) / 2, (y1 + y3) /
2);
26         sierpinski(n - 1, (x1 + x2) / 2, x2, (x2 + x3) / 2, (y1 + y2) / 2, y2, (y2 + y3) /
2);
27         sierpinski(n - 1, (x1 + x3) / 2, (x2 + x3) / 2, x3, (y1 + y3) / 2, (y2 + y3) / 2,
y3);
28     }
29 }

```

```

1 /** Prints the Sierpinski Triangle fractal. */
2 public class Sierpinski {
3
4     public static void main(String[] args) {
5         sierpinski(Integer.parseInt(args[0]));
6     }
7
8     // Draws a Sierpinski triangle of depth n on the standard canvass.
9     public static void sierpinski(int n) {
10         double s = Math.sqrt(3) / 2;
11         StdDraw.line(0, 0, 1, 0);
12         StdDraw.line(0.5, s, 0, 0);
13         StdDraw.line(1, 0, 0.5, s);
14         sierpinski(n - 1, 0, 1, 0.5, 0, 0, s);
15     }
16
17     public static void sierpinski(int n, double x1, double x2, double x3,
18     , double y1, double y2, double y3) {
19         if (n == 0) {
20             return;
21         }
22         StdDraw.line((x1 + x2) / 2, (y1 + y2) / 2, (x1 + x3) / 2, (y1 + y3) / 2); // ! left
diagonal
23         StdDraw.line((x1 + x2) / 2, (y1 + y2) / 2, (x2 + x3) / 2, (y1 + y3) / 2); // ! right
line
24         StdDraw.line((x1 + x3) / 2, (y1 + y3) / 2, (x2 + x3) / 2, (y1 + y3) / 2); // ! stright
line
25         sierpinski(n - 1, x1, (x1 + x2) / 2, (x1 + x3) / 2, y1, (y1 + y2) / 2, (y1 + y3) / 2);
26         sierpinski(n - 1, (x1 + x2) / 2, x2, (x2 + x3) / 2, (y1 + y2) / 2, y2, (y2 + y3) / 2);
27         sierpinski(n - 1, (x1 + x3) / 2, (x2 + x3) / 2, x3, (y1 + y3) / 2, (y2 + y3) / 2, y3);
28     }
29 }

```

```

1 /** Draws the Koch curve and the the Koch snowflake fractal. */
2 public class Koch {
3
4     public static void main(String[] args) {
5
6         // Uncomment the first code block to test the curve function.
7         // Uncomment the second code block to test the snowflake function.
8         // Uncomment only one block in each test, and remember to compile
9         // the class whenever you change the test.
10        /*
11         * // Tests the curve function:
12         * // Gets n, x1, y1, x2, y2,
13         * // and draws a Koch curve of depth n from (x1,y1) to (x2,y2).
14         */
15        // curve(Integer.parseInt(args[0]),
16        // Double.parseDouble(args[1]), Double.parseDouble(args[2]),
17        // Double.parseDouble(args[3]), Double.parseDouble(args[4]));
18
19        /*
20         * // Tests the snowflake function:
21         * // Gets n, and draws a Koch snowflake of n edges in the standard canvass.
22         */ snowFlake(Integer.parseInt(args[0]));
23    }
24
25    /**
26     * Gets n, x1, y1, x2, y2,
27     * and draws a Koch curve of depth n from (x1,y1) to (x2,y2).
28     */
29    public static void curve(int n, double x1, double y1, double x2, double y2) {
30        if (n == 0) {
31            return;
32        }
33        StdDraw.line(x1, y1, x2, y2);
34        double p1x = x1 + (x2 - x1) / 3;
35        double p1y = y1 + (y2 - y1) / 3;
36        double p2x = x2 - (x2 - x1) / 3;
37        double p2y = y2 - (y2 - y1) / 3;
38        double p3x = ((Math.sqrt(3) / 6) * (y1 - y2)) + ((x1 + x2) / 2);
39        double p3y = ((Math.sqrt(3) / 6) * (x2 - x1)) + ((y1 + y2) / 2);
40        // * Drawing the new shape
41        StdDraw.line(p1x, p1y, p3x, p3y);
42        StdDraw.line(p3x, p3y, p2x, p2y);
43        // * Removing the original segment
44        StdDraw.setPenColor(StdDraw.WHITE);
45        StdDraw.line(p1x, p1y, p2x, p2y);
46        StdDraw.setPenColor(StdDraw.BLACK);
47        curve(n - 1, x1, y1, p1x, p1y);
48        curve(n - 1, p1x, p1y, p3x, p3y);
49        curve(n - 1, p3x, p3y, p2x, p2y);
50        curve(n - 1, p2x, p2y, x2, y2);
51    }
52
53    /** Gets n, and draws a Koch snowflake of n edges in the standard canvass. */
54    public static void snowFlake(int n) {
55        // A little tweak that makes the drawing look better
56        StdDraw.setYscale(0, 1.1);
57        StdDraw.setXscale(0, 1.1);
58        // Draws a Koch snowflake of depth n
59        curve(n, 0.0, 0.85, 1.0, 0.85);
60        curve(n, 0.5, 0.0, 0.0, 0.85);
61        curve(n, 1.0, 0.85, 0.5, 0.0);
62    }
63 }

```