Recursion Exercises

Exercise 1

Write a recursive Java method with the signature

```
public static long factorial(int n)
```

that implements the factorial function using the following definition

$$n! = \begin{cases} 1 \text{ if } n = 0 \\ n \times (n-1)! \text{ if } n > 0 \end{cases}$$

Exercise 2

Write a recursive Java method with the signature

```
public static long fibonacci(int n)
```

that returns the n-th Fibonacci number. The definition for the Fibonacci sequence is

$$f(n) = \begin{cases} 1 \text{ if } n = 1 \\ 1 \text{ if } n = 2 \\ f(n-1) + f(n-2) \text{ if } n > 2 \end{cases}$$

Exercise 3

A ball is kicked vertically from ground level and reaches a maximum height of 8 meters. Every time the ball bounces, it reaches a maximum height that is 70% of the previous maximum height. Write a recursive Java method (and any necessary helping methods) with the signature

```
public static double ballDistance(int n)
```

that returns the total distance travelled by the ball from the moment it's kicked to the moment it touces the ground for the time `nth`. At start, `n=0` Here are some sample runs:

```
n     returns
0     0
1     16
2     27.2
6     47.05872
13    52.816592
```

Exercise 4

The following is a portion of Pascal's Triangle

```
                1                 row 0
              1   1               row 1
            1   2   1             row 2
          1   3   3   1           row 3
        1   4   6   4   1         row 4
```

If the first term of any row is in column zero, then any interior term of the triangle can be calculated by

$$t_{row,col} = t_{row-1,col-1} + t_{row-1,col}$$

For example, if *row* = *4* and *col* = *2* then $t_{4,2} = t_{3,1} + t_{3,2} = 3 + 3 = 6$

Write a recursive expression that calculates any term of the triangle from a given row and column

Exercise 5

Write a recursive Java method with the signature

```
public static int pascalTerm(int row, int col)
```

that returns the term of the Pascal Triangle in `(row, col)`

Exercise 6

Write a Java method with the signature

```
public static void pascalTriangle(int row)
```

that uses the method of the previous exercise and prints all the rows of Pascal Triangle from `0` to `row`. For example, the call

```
pascalTriangle(4)
```

prints

```
    1
  1  1
    1  2  1
  1  3  3  1
1  4  6  4  1
```

Challenge: if `row > 4` the format of the triangle becomes more complicated as the numbers to print have more than one digit

Exercise 7

Euclid's algorithm finds the greatest common divisor of two positive integers *m* and *n*. We implemented the algorithm *iteratively* in the past. Now we implement it recursively. The definition is,

$$\gcd(m,n) = \begin{cases} m, \text{ if } m = n \\ \gcd(n, m-n), \text{ if } m > n \\ \gcd(n,m), \text{ if } m < n \end{cases}$$

Write a Java method with the signature

```
public static int gcd(int m, int n)
```

that implements a recursive algorithm for finding the greatest common divisor of two positive integers. Test your method: `gcd(21, 18)` is 3; `gcd(37, 36)` is 1; `gcd(120, 20)` is 20. When writing the Java code pay close attention to the order of the actual and the formal parameters as given in the definition.

Exercise 8

Generating permutations from a given pool of characters can be done with recursion. For example, all possible permutations of length 3 with the characters A, B, C (allowing character repetition) results in

```
AAA
AAB
AAC
ABA
ABB
ABC
ACA
ACB
ACC
BAA
BAB
BAC
BBA
BBB
BBC
BCA
BCB
BCC
CAA
CAB
CAC
CBA
CBB
CBC
CCA
CCB
CCC
```

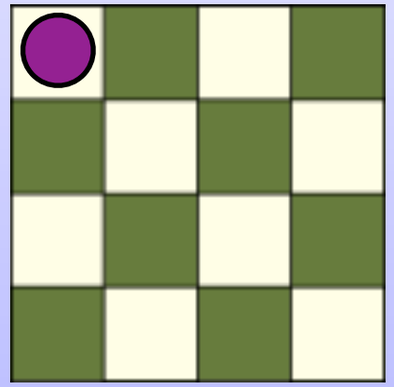
Here is code that prints the permutations

```
public class Exercise8 {
    public static int permutationLength = 3;

    public static void main(String[] args) {
        permutations("");
    }

    public static void permutations(String s) {
        if (s.length() == 3) {
            System.out.println(s);
        }
        else {
            for (char letter = 'A'; letter <= 'C'; letter++) {
                permutations(s + letter);
            }
        }
    }
}
```

Trace the code that prints at least the first 4 permutations.

Exercise 9 (1/3)

Recursion is used in a type of algorithm known as *backtracking*. These are particularly useful for 2D board games because we can look ahead board configurations. In the following simple example, a piece can move only one position to the right or one position down. We can write a recursive method that generates all the possible pathways (of six moves) from the top left to the bottom right of the board



The idea is that if we can move the piece to the right, we go ahead and move it. Then we will have the same problem as originally but the board just got a whole column smaller. So we try to find all the pathways from the new position to the bottom right. We attempt to move to the right again and find all the pathways from the new position. We keep track of each move that brings us closer to the solution.

If we get to a position where cannot move to the right, we then move down and from this new position we find all the pathways. As soon as we have a path that reaches the final goal we write it down, then we *backtrack* to a position in the board where the process can continue generating pathways. If we implement the process correctly, *recursion will automatically take care of the backtracking for us.*

This type of algorithm that checks all the possible permutations in order to arrive at solutions is know as *brute force.* Games like chess, sudoku, and most board games use a bruce force approach. Some *heuristics* can be included in games to reduce the computation time and make the game more intelligent.

Exercise 9 (2/3)

To implement our solution we need the following:

A 4x4 array that represents the board. The contents of the array do not matter in this case. So we can make it an array of `int` and fill it with zeroes. We will declare indices to keep track of the location of the piece.

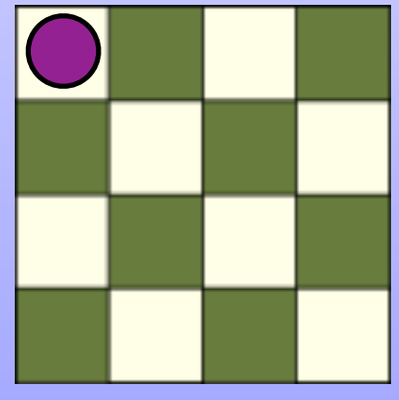
To make the programming easier, we can make the array global so that it is accessible from anywhere in the program and this way we do not need to pass it as a parameter to the methods. A global declaration is placed **outside** the methods. A good place is right after the class name. Global declarations need to be static,

```
static int[][] board = new int[4][4];
```

In the `main()` method we initialize the array to zeroes (again, the contents do not matter, but we initialize just so that we know what's in the array)

In the `main()` method we place a call to the recursive method:

```
findPath(0, 0, "");
```

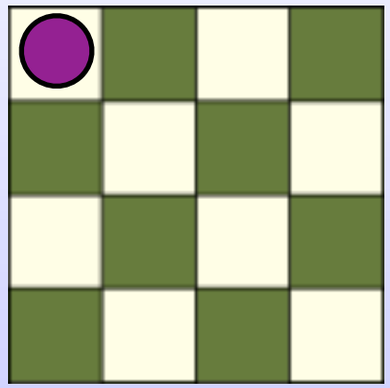


The first two parameters are the row and column where the piece is located
The third parameter will contain the pathway that has been generated so far. At the start no pathway has been generated, therefore the pathway is a null string.

The algorithm for the `findPath()` method is on the next slide

Exercise 9 (3/3)

recursive algorithm



```
findPath(int row, int column, String pathway) {
    if goal position has been reached
        print pathway
    else {
        if the piece can be moved to the right {
            move the piece to the right one position
            pathway = pathway + "RIGHT:"
            findPath(row, column, pathway)
            move the piece to the left one position        % this is the backtracking
            remove "RIGHT:" from pathway
        }
        if the piece can be moved down {
            move the piece down one position
            pathway = pathway + "DOWN:"
            findPath(row, column, pathway)
            move the piece up one position                 % this is the backtracking
            remove "DOWN:" from pathway
        }
    }
}
```

Task. Write a Java program with the algorithm given above that finds all the pathways from the top left corner of the board to the bottom right corner

Exercise 10

The truth-table for the sentence

```
P AND Q OR S AND T
```

has $2^4 = 16$ entries since there are four different variables, each of which can assume the value `TRUE` or `FALSE`.

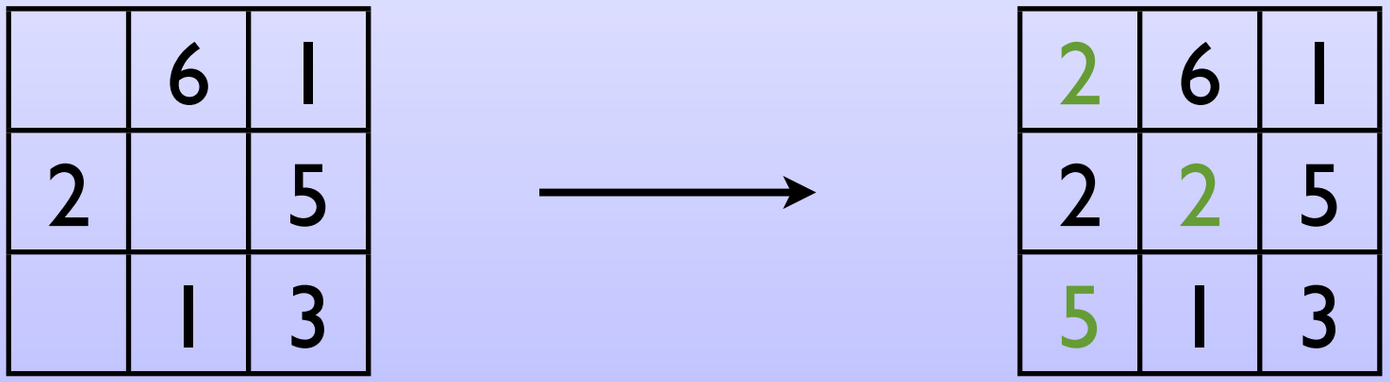
Modify the program of Exercise 8 so that it generates all the possible permutations of four variables. Do not worry about evaluating the expression above. If we use `1` for `TRUE` and `0` for `FALSE`, the printout of the program is,

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Incidentally, these are the first 16 binary numbers.

Exercise 11

The following is a sample configuration of magic numbers. The task is to fill the empty slots of a given board with any numbers from 0 to 9 so that adding the numbers of a row results in 9 and adding the numbers of a column results also in 9. In this example the board on the left would become the board on the right:



Write a recursive algorithm in pseudo-code that uses a 2D array to represent a board such that,

a) An empty slot is represented with -1
b) All other slots contain the given numbers

Use a brute force approach to determine the numbers that will satisfy the game

In the example above there are 1000 possible permutations since 3 slots are available. In general, there are 10^n possible permutations for any game with *n* empty slots.

Selected Solutions

Solution to Exercise 2

```
public class Recursion2 {
    public static long fibonacci(int n) {
        if (n == 1 || n == 2) {
            return 1;
        }
        else {
            return(fibonacci(n - 1) + fibonacci(n - 2));
        }
    }
}
```

Solution to Exercise 3

```
public class Recursion3 {
    public static double height = 8;
    public static double distance(int bounce) {
        if (bounce == 1) {
            return 2 * height;
        }
        else {
            return 2 * height * Math.pow(0.7, bounce - 1) + distance(bounce - 1);
        }
    }
}
```

Solution to Exercises 4-6

```
import hsa.*;
public class PascalTriangle {
    public static int pascalTerm(int row, int col) {
        if (col == 0) {
            return 1;
        }
        else if (row == col) {
            return 1;
        }
        else {
            return pascalTerm(row - 1, col - 1) + pascalTerm(row - 1, col);
        }
    }

    public static void pascalTriangle(int row) {
        for (int r = 0; r <= row; r++) {
            Stdout.print("", -20 + r);
            for (int col = 0; col <= r; col++) {
                System.out.print(pascalTerm(r, col) + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        pascalTriangle(4);
    }
}
```

Solution to Exercise 7

```
public class Recursion7 {
    public static int gcd(int m, int n) {
        if (m == n) {
            return m;
        }
        else if (m > n) {
            return gcd(n, m - n);
        }
        else {
            return gcd(n, m);
        }
    }

    public static void main(String[] args) {
        System.out.println(gcd(21, 18));
        System.out.println(gcd(37, 36));
        System.out.println(gcd(120, 20));
    }
}
```

Solution to Exercise 8

```
public class Recursion8 {
    public static int permutationLength = 3;

    public static void permutations(String s) {
        if (s.length() == permutationLength) {
            System.out.println(s);
        }
        else {
            for (char letter = 'A'; letter <= 'C'; letter++) {
                permutations(s + letter);
            }
        }
    }

    public static void main(String[] args) {
        permutations("");
    }

}
```

Solution to Exercise 9

```
public class Recursion9 {
    static int[][] board = new int[4][4];

    private static boolean endReached(int row, int col) {
        return row == board.length - 1 && col == board[0].length - 1;
    }

    private static boolean rightObstacleReached(int col) {
        return col == board[0].length - 1;
    }

    private static boolean bottomObstacleReached(int row) {
        return row == board.length - 1;
    }

    public static void findPath(int row, int col, String path) {
        if (endReached(row, col))
            System.out.println(path);
        else {
            if (!(rightObstacleReached(col))) {
                col++;
                path = path + "R:";
                findPath(row, col, path);
                col--;
                path = path.substring(0, path.length() - 2);
            }

            if (!(bottomObstacleReached(row))) {
                row++;
                path = path + "D:";
                findPath(row, col, path);
                row--;
                path = path.substring(0, path.length() - 2);
            }
        }
    }

    public static void main(String[] args) {
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board[0].length; col++) {
                board[row][col] = 0;
            }
        }
        findPath(0, 0, "");
    }
}
```

Solution to Exercise 10

```
public class Recursion10 {
    public static int permutationLength = 4;

    public static void permutations(String s) {
        if (s.length() == permutationLength) {
            System.out.println(s);
        }
        else {
            for (int state = 0; state < 2; state++) {
                permutations(s + state);
            }
        }
    }

    public static void main(String[] args) {
        permutations("");
    }
}
```

Solution to Exercise 11

```
import hsa.*;
public class Recursion12 {
    public static int[][] board = new int[3][3];

    public static boolean boardDone() {
        boolean ok = true;
        for (int i = 0; i < 3; i++) {
            ok = ok && board[i][0] + board[i][1] + board[i][2] == 9 &&
                       board[0][i] + board[1][i] + board[2][i] == 9;
        }
        return ok;
    }

    public static boolean boardReadyForCheck() {
        boolean ready = true;
        for (int row = 0; row < 3 && ready; row++) {
            for (int col = 0; col < 3 && ready; col++) {
                ready = board[row][col] != -1;
            }
        }
        return ready;
    }

    public static void printBoard() {
        System.out.println("     0  1  2");
        System.out.println("  ----------");
        for (int row = 0; row < 3; row++) {
            System.out.print(row + " |");
            for (int col = 0; col < 3; col++) {
                Stdout.print(board[row][col], 3);
            }
            System.out.println();
        }
    }
```

```
    public static void generateBoard() {
        if (boardReadyForCheck()) {
            if (boardDone()) {
                printBoard();
                int r = Stdin.readInt(); // suspend artificially
                // the exiting process needs to be implemented
            }
        }
        else {
            boolean endProcess = false;
            for (int row = 0; row < 3 && !endProcess; row++) {
                for (int col = 0; col < 3 && !endProcess; col++) {
                    if (board[row][col] == -1) {
                        for (int value = 0; value < 10; value++) {
                            board[row][col] = value;
                            generateBoard();
                        }
                        board[row][col] = -1;
                        endProcess = true;
                    }
                }
            }
        }

    public static void main(String[] args) {
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                board[row][col] = -1;
            }
        }
        board[0][1] = 6;
        board[0][2] = 1;
        board[1][0] = 2;
        board[1][2] = 5;
        board[2][1] = 1;
        board[2][2] = 3;

        generateBoard();
    }
}
```

Note: Once the correct board is found and printed, this program suspends but does not end. It must be ended manually. The mechanism to recover from the deepest recursion level is not implemented in the recursive method.