CSCI 2110 Data Structures and Algorithms
Laboratory No. 5
Week of 14 October

~~**Due: Sunday 20 October**~~
Due: Sunday 27 October
23h55 (five minutes to midnight)

Recursion

The objective of this lab is to help you get familiar with recursion. Remember that in order to design a recursive program, you need to establish a *base case* (this is the trivial case where a problem can be solved without recursion) and *recursive case* (this is the case where a problem will be decomposed into smaller parts, and additional calls to the program made). Once you are able to define a problem in this way, you can often write a very simple program to solve it.

Recursion has not yet been covered in your lectures, so a supplementary, learning lab will be presented for students not yet ready to tackle that topic. All lab work will be due 27 October. The objective of the supplementary lab is to provide some additional practice with working with ordered lists in Java. Download the example code/files provided along with this lab document. You will need the following files to complete your work:

Node.java (Generic Node Class)
LinkedList.java (Generic Linked List Class)
List.java (Generic Unordered List Class)
OrderedList.java (Generic Ordered List Class)
Employees.txt (Sample text file for input in Exercise0A)
Names.txt (Sample text file for input in Exercise0B)

Marking Scheme
Each exercise carries 10 points.
Working code, Outputs included, Efficient, Good basic comments included: 10/10
No comments or poor commenting: subtract one point
Unnecessarily inefficient: subtract one point
No outputs and/or not all test cases covered: subtract up to two points
Code not working: subtract up to six points depending upon how many methods are incorrect. TAs can and will test cases beyond those shown in sample inputs, and may test methods independently.

Your final score will be scaled down to a value out of 10. For example, if there are three exercises and you score 9/10, 10/10 and 8/10 on the three exercises, your total is 27/30 or 9/10.

Error checking: Unless otherwise specified, you may assume that a user enters the correct data types and the correct number of input entries, that is, you need not check for errors on input.

Submission: All submissions are through Brightspace. Log on to dal.ca/brightspace using your Dal NetId. Deadline for submission: Sunday 27 October 23h55 (five minutes before midnight).

What to submit:
Submit one ZIP file containing all source code (files with .java suffixes) and text documents containing sample outputs. For each exercise you will minimally have to submit a demo class with a main method and the required static method(s), and sample outputs. If you wish, you may combine your test outputs into a single file called Outputs.txt via cut-and-paste or a similar method.
Your final submission should include the following files: Employee.java, EmployeeList.java, EmployeeListDemo.java, UnorderedSearchDemo.java, OrderedSearchDemo.java, Exercise1.java, Exercise2.java, Exercise3.java, Exercise4.java, Exercise5.java, Exercise6.java, Exercise7.java, Node.java, LinkedList.java, List.java, OrderedList.java, Employees.txt, Names.txt, Outputs.txt *or* labelled test outputs for each exercise.

You MUST SUBMIT .java files that are readable by your TAs. If you submit files that are unreadable such as .class, you will lose points. Please additionally comment out package specifiers.

Exercise0A (Ordered List)
For this exercise you will develop a program that can store and manage information about a company's employees. You will need the OrderedList.java file to complete your solution.
First create an Employee class. Call it Employee.java. It will be similar to other classes you have written in past labs (e.g. Expense.java), but will have to implement the Comparable interface. Your Employee Objects are going to be stored in an OrderedList structure that implements a binary search. Employee Objects should have the following fields: *EmployeeID*, *FirstName*, *LastName*, *Email*, *Department*. **The EmployeeID field will be the key used to compare different Employee Objects.** Add appropriate methods and build out upon this framework as required.
Next create an EmployeeList class. Call it EmployeeList.java. EmployeeList Objects should have a field that is an OrderedList of type Employee, and the following methods:

1. `public void addEmployee(Employee e)`: Add an Employee to the list.

2. `public void deleteEmployee(int ID)`: Delete an Employee with the specified EmployeeID number.

3. `public Employee searchID(int ID)`: Search for an Employee given an EmployeeID number.

You may add other methods as necessary.

Finally, create a demo program. Call it EmployeeListDemo.java. Your program should **accept input from a user** (specifying the name of an input file) and read a file formatted like the one below (see also: Employees.txt). Your program should create an ordered list of Employee Objects, and demonstrate all the methods that you have developed (you may hard code these tests). Each line of the input file will consist of a Employee ID number, first name, last name, email address, and department separated by whitespace.

```
30001 John Smith jsmith@email.com Management
22344 Allison Page apage@email.com Accounting
87987 Craig Cambell ccambell@email.com Research & Development
78976 Steve Edwards sedwards@email.com Management
23455 Mike Williams mwilliams@email.com Sales
22349 Jane Reid jreid@email.com Management
23456 Kate West kwest@email.com Accounting
23457 Julie McLain jmclain@email.com Sales
30002 Tom Erlich terlich@email.com Sales
```

```
21117 Mark Smith msmith@email.com Accounting
22345 Jen Foster jfoster@email.com Sales
23458 Matt Knight mknight@email.com Management
22346 Karen Weaver jkweaver@email.com Management
```

Note: One sample output which shows the results for all the methods that you have developed is sufficient.

Exercise0B (Complexity of Search)

This exercise requires you to demonstrate experimentally the difference in complexity between searching unordered and ordered lists (that is the difference between searching a structure via linear traversal and binary search). You will need the Node.java, LinkedList.java, List.java, and OrderedList.java files to complete your solution.

Write a program called UnorderedSearchDemo.java that can read a list of names from a file and construct an unordered list. The input file will be a very long list of names, one per line, in ascending order. Your program should **accept input from a user** (specifying a name String) and call a static method called *search* to determine if a name is present in the your unordered list. Your method header should be:

```
public static Boolean search(String name)
```

Test your program with a 10 randomly selected names, and record the name String and execution time for each search operation in an output file via cut-and-paste or a similar method. You can use the following code to capture the execution time:

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//code segment

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
```

Now write a program called OrderedSearchDemo.java that can read a list of names from a file and construct an ordered list. Your program should **accept input from a user** (specifying a name String) and call a static method called *search* (with the same header as above) to determine if a name is present in the your ordered list. Repeat the above experiment, and record the name String and execution time for each search operation in an output file via cut-and-paste or a similar method.

The output files from your experimental runs of the two programs created should help illustrate the difference in complexity between searching unordered and ordered lists (that is the difference between searching a structure via linear traversal and binary search).

Exercise1 (Recursive Review)

This exercise presents a brief review of recursion. You will write a series of short methods to implement the recursive programs discussed in recent lectures. Each program is presented in pseudo code. Implement all 3 methods as static methods and test them in the main method in a file called Exercise1.java.

a) Factorial of an integer n:

```
if n == 0, then factorial(n) = 1 //base case
if n > 0, then factorial(n) = n * factorial(n-1)
```

Write a recursive method that calculates the factorial of a non-negative integer *n*. Call this method in your main method, using a loop to print the factorials of the integers 1 through 10. Your method header should be:

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

b) Fibonacci series:

```
if n==0, then fib(n) = 0 //base case
if n==1, then fib(n) = 1 //base case
if n>1, then fib(n) = fib(n-1) + fib(n-2)
```

Write a recursive method that finds the *n*th number in the Fibonacci series. Call this method in your main method, using a loop to print the first 20 numbers in the Fibonacci series. Your method header should be:

```
public static int fib(int n)
```

c) Exponentiation (*x* to the power *n*):

```
if n == 0, then power(x,n) = 1 //base case
if n>0, then power(x,n) = power(x,n-1)*x
```

Write a recursive method that calculates *x* to the power *n*, where *x* and *n* are both positive integers. Call this method in your main method, **prompting a user to enter *x* and *n***, and print *x* to the power of *n*. Your method header should be:

```
public static int power(int x, int n)
```

Test your program to ensure its proper operation. Save data from one test run in a text file.

Exercise2 (Countdown Part 1)

Write a recursive method called *countDown* that takes a single positive integer *n* as parameter and prints the numbers *n* through 1 followed by 'BlastOff!'. Your method header should be:

```
public static void countDown(int n)
```

`countDown(10)` would print:

```
10    9    8    7    6    5    4    3    2    1    BlastOff!
```

Write a program called Exercise2.java. Your program should **accept input from a user** (specifying an integer *n*) and count down to 1 from *n* before printing 'BlastOff!'. Test your program with a variety of inputs to ensure its proper operation. Do not hard code inputs. Save data from one test run in a text file.

Exercise3 (Countdown Part 2)

Modify the *countDown* method you wrote in the previous exercise so that it prints only even numbers when *n* is even, and only odd numbers when *n* is odd. Your main method and the header for your *countDown* method can remain unchanged.

`countDown(10)` would print:

```
10      8       6       4       2       BlastOff!
```

`countDown(9)` would print:

```
9       7       5       3       1       BlastOff!
```

Write a program called Exercise3.java. Your program should **accept input from a user** (specifying an integer *n*) and count down from *n* before printing 'BlastOff!'. Test your program with a variety of inputs to ensure its proper operation. Do not hard code inputs. Save data from one test run in a text file.

Exercise4 (Multiples)

Write a recursive method called *multiples* to print the first *m* multiples of a positive integer *n*. Your method header should be:

```
public static void multiples(int n, int m)
```

`multiples(2, 5)` would print:

```
2, 4, 6, 8, 10
```

`multiples(3, 6)` would print:

```
3, 6, 9, 12, 15, 18
```

Write a program called Exercise4.java. Your program should **accept input from a user** (specifying integers *n* and *m*) and print multiples. Test your program with a variety of inputs to ensure its proper operation. Do not hard code inputs. Save data from one test run in a text file.

Note: You may print multiples in either ascending or descending order.

Exercise5 (Write Vertical)

Write a recursive method called *writeVertical* that takes that takes a single positive integer *n* as a parameter and prints the digits of that integer vertically, one per line. Your method header should be:

```
public static void writeVertical(int n)
```

`writeVertical(1234)` would print:

```
1
2
3
4
```

Write a program called Exercise5.java. Your program should **accept input from a user** (specifying an integer *n*) and print its digits vertically. Test your program with a variety of inputs to ensure its proper operation. Do not hard code inputs. Save data from one test run in a text file.

Exercise6 (Sum of Squares)

Write a recursive method called *squares* that takes that takes a single positive integer *n* as a parameter and calculates the sum of the squares of all digits 1 through *n*. Your method header should be:

```
public static int squares(int n)
```

`squares(4)` would return:

```
30
```

Write a program called Exercise6.java. Your program should **accept input from a user** (specifying an integer *n*) and return the sum of the squares of all digits 1 through *n*. Test your program with a variety of inputs to ensure its proper operation. Do not hard code inputs. Save data from one test run in a text file.

Exercise7 (Towers of Hanoi)

Rewrite the recursive solution to the Towers of Hanoi problem discussed in recent lectures, and modify it to return the number of moves required to solve the problem for a given number of discs. Your recursive method, called *solve*, should take a single positive integer *n* as a parameter and return the number of moves required for an optimal solution. Your method header will likely take two forms:

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

and

```
public static long solve(int n, int start, int end, int tmp)
```

Write a program called Exercise7.java. Your program should **accept input from a user** (specifying an integer *n*) and return the number of moves required to optimally solve the Towers of Hanoi problem with *n* discs. Determine experimentally how the execution time for solving the Towers of Hanoi problem grows as the value of *n* increases. You can use the following code to capture the execution time:

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//code segment

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
```

Test your program with inputs of 8, 12, 16, 20, 24, 28, and 32. Save data from your test runs in a text file showing *n*, the number of moves, and execution time in three columns.

Note: Calculating the number of moves required to solve the Towers of Hanoi problem for a given number of discs is straightforward. The formula is (2^n)-1. Do not use this formula in your solution. Your *solve* method must return this value upon exit.