Department of EECS
York University

Term: Fall 2017
Instructor: Andy Mirzaian

EECS2011: Fundamentals of Data Structures
Section A
Assignment 2
Due: 10 pm, Friday, October 20, 2017

Instructions:

- This is an individual assignment; you may not share code with other students.
- Read the course FAQ on the generic marking scheme and how to submit assignments electronically. The submission deadline is strictly enforced.
- This assignment consists of 4 problems and is designed to let you practice on (linked) lists, iteration versus recursion, and analysis.
- Here you can access the directory of this assignment, **A2** (which is also the package name). It includes the file **a2.pdf** that you are now reading, and some source codes that you may use to complete your assignment solutions.
- Print your name, eecs account, and student ID number on top of **every** file you submit. We are not responsible for submitted files that have no student identifications.
- Use informative Javadoc and non-Javadoc comments to further explain your code.
- In addition to the required .java files, you should also submit an **a2sol.pdf** file that includes all other required answers, justifications, explanations, analysis, etc.

Problem 1. [25%] Pancake Sorting on Doubly Linked Lists

Pancake Sort is a well known procedure to sort any sequence of comparable elements by applying a number of prefix reversals (aka pancake flips). A simple strategy that resembles Selection Sort requires at most 2n-3 prefix reversals to sort any given sequence of n elements. In this strategy, with one flip we can bring the max element to the front of the sequence, and with one more flip we can send that max element to its final position at the end of the sequence. Then repeat the process on the remaining pancakes. For more details see https://en.wikipedia.org/wiki/Pancake_sorting.

Here we want to simulate this strategy using a doubly linked list that represents the sequence to be sorted. The [GTG] source code *DoublyLinkedList.java* is provided in the A2 directory. You may use it without modification.
Design the *PancakeSort.java* program with the following specifications.

- The [GTG] doubly linked list uses the sentinel nodes *header* and *trailer*. Any non-empty prefix of such a list can be specified by a reference *prefixLast* that points to any non-sentinel node of the list. The prefix nodes are the list nodes from the front up to and including the *prefixLast* node.
- *getPrefixMaxNode(prefixLast)* is an instance access method that returns a pointer to the node with maximum element within the specified prefix of the instance list. (Ties for the max are broken arbitrarily).
- *reversePrefix(prefixLast)* is an update instance method that reverses the sequence of nodes of the specified prefix in the instance list.
- *buildList()* is a static method that reads a sequence of elements of a generic comparable type from the input (or as parameter argument) and returns a doubly linked list consisting of those elements in the given order.
- *pancakeSort()* is an instance update method that calls the above methods to sort the given instance list.
- The *main()* method provides mechanism to test-run pancake sort. It calls *buildList(), pancakeSort(),* and *DoublyLinkedList.toString()* to print the input sequence and the corresponding sorted output sequence.
- Implement these methods and analyze time complexities of *getPrefixMaxNode, reversePrefix, buildList,* and *pancakeSort.*

Problem 2. [22%] A Recursive Function

Consider the following recursive function:

```
public static int f ( int n )  {
   if ( n > 1000 ) return n – 4;
   return f ( f (n + 5) );
}
```

a) What is f(1001) ? What is f(1000) ? What is f(0) ?

b) Either exhibit an input instance n on which a call to f(n) gets into infinite-recursion, or argue that on all n, f(n) terminates after a finite number of steps.
[You may answer this in conjunction with part (c) below.]

c) For an arbitrary given n, what does f(n) return (if it terminates) ?
Express your answer in its simplest form as a function of n.

d) [Optional]
What is the exact number of calls to f() made by a call to f(n) (finite?) ?
Express your answer in its simplest form as a function of n.

[Hint: use "backwards" mathematical induction. What are the base cases?]

Problem 3. [23%] Decimal to Binary Conversion

Write a program *DecimalToBinary.java* that provides two static utility functions *Rec10To2(int n)* and *Iter10To2(int n),* the first recursive and the second iterative, that given a positive integer *n* interpreted as a decimal number, each prints the binary representation of *n* from most significant to least significant bit.
For example, if the input is **25**, the output must exactly be **11001**.

Restrictions: Rec10To2() should not use any local variables nor any loops. The output bits must be computed using ordinary arithmetic operations on integers: + , − , * , / , %.

Analyze the time complexities of both methods.
Provide a main() method to test-run the above methods on any user input *n*.

Problem 4. [30%] Budgeted Itinerary

Suppose the input is a list of n items given by their *unit prices* from least to most expensive, and a *budget* value; all positive integers. An *itinerary* is a list specifying the quantities of each of these n items in the given order, followed by their total cost. The latter is item unit prices times their quantities summed over all items in the list. We say such an itinerary is *within budget* if its total price does not exceed the given budget. We say the itinerary is *saturated* if it is within budget, but it will fail to be within budget if we add any more item to it.

Design a program *SaturatedItineraries.java* that includes an efficient method *reportSI(unitPriceList, budget)* that is *purely recursive* (i.e., it is recursive but contains no loops). This (possibly overloaded) method prints out the input unit price list followed by the given budget, followed by a new line. Then it prints all possible saturated itineraries as a list of item quantities, one list per line, each followed by its total price on the same line. (Make sure the output shows each saturated itinerary exactly once, not less not more.) Then on the last line it prints the number of saturated itineraries.

See the example below with n = 3. You should also try some larger examples.

You may implement the item price list and the quantities list as arrays. Pay attention not only to time efficiency but also space efficiency by reusing the quantities array for all itineraries.

Example output (partially shown):

```
          Unit Price List = [ 3 , 5 , 7 ].   Budget = 88.
          The saturated itineraries are:
                    …
                    Quantities = [ 8 , 4 , 6 ].    Total Price = 86.
                    …
                    Quantities = [ 1 , 3 , 10 ].    Total Price = 88.
                    …
                    Quantities = [ 1 , 0 , 12 ].    Total Price = 87.
                    …
          The number of saturated itineraries =  …
```