**Assignment 1 makeup:**

This is a two part programming assignment. The first part provides the student with practice in constructing a simple doubly linked list. The second part introduces singly linked lists and is a bit more challenging. The second part also serves as an introduction to NP-Complete problems.

Part 1.  Doubly Linked Lists 30%

Attached are abbreviated Javadoc specifications for two classes, DoubleNode.java and DoublyLinkedList.java.

You have seen how singly linked lists work in class. A singly linked list consists of a series of nodes linked together via pointers. Each node has a next pointer which points to the next node in the list. The next pointer of the last node in the list is set to null.

A doubly linked list is similar to a singly linked list, however in a doubly linked list each node also contains a previous pointer, which points to the previous node in the list. The previous pointer of the node at the head of the list is set to null. The next pointer of the last node is set to null.

Following the Javadoc specifications given, you are to write implementations of DoubleNode and DoublyLinkedList. You need to use the exact same names as those provided so that your classes may be included in our test driver. You should begin by writing only the method signatures and Javadoc comments for each class. You should then generate Javadoc specifications for your classes similar to those attached. You are required to include pre- and post-conditions in your Javadoc for each method. After that, you should write your own implementation for each of the methods.

A simple test driver that you are required to use is provided below.

Submit the following as a single zipped project to the Assignment Section of Blackboard:

1. DoubleNode.java and DoublyLinkedList.java, containing your Java source for these classes.
2. Javadoc for each class and method. This Javadoc must contain each method’s best and worst case run time complexity. This must be expressed using big theta notation. If you are not able to determine the best and worst case run time complexity, explain why.  Again, the Javadoc will contain pre- and post-conditions for each method.
3. A screen scrape showing the output of the main program shown below.

Part 1 Grading:

      Working code passing the test driver: 70%

      Javadoc describing worst and best case big theta values

and pre- and post-conditions : 20%

      Overall presentation: 10%

Part 2 Merkle-Hellman Knapsack Cryptosystem   70%

In this assignment you will implement key generation, encryption and decryption using the Merkle-Hellman Knapsack Cryptosystem. A very clear and well-written description of this algorithm can be found at the following link. This is a required reading for the course and should be understood prior to writing code:

[http://en.wikipedia.org/wiki/Merkle–Hellman\_knapsack\_cryptosystem](http://en.wikipedia.org/wiki/Merkle%E2%80%93Hellman_knapsack_cryptosystem)

You will write two doubly linked lists of objects to hold BigIntegers. One list, w, will be used to hold the superincreasing sequence of integers that make up part of the private key and used for decryption. The second list, b, will be used to hold the public key material used for encryption. Your list class should encapsulate all of the work associated with lists and should not know anything about Merkle-Hellman. You will include pre- and post-conditions with each of your methods and will describe each method’s run-time complexity using Big Theta.

Using a doubly linked list for this problem is appropriate but not ideal. It is very appropriate for a first course in data structures. Hence, use it for this homework but you should be aware that there are alternatives.

Use the built in methods of the BigInteger class provided by Java. These methods make it fairly easy to implement some of the tedious but essential parts of Merkle-Hellman.

To do a practice example, you might like to use Wolfram Alpha at [http://www.wolframalpha.com](http://www.wolframalpha.com/). It accepts operations such as

(3 \* 5) mod 2, gcd(32,5) and PowerMod(15,-1,64) to find the multiplicative inverse of 15 modulo 64.

Your program will be interactive and will behave in a similar manner as mine. An example run of my program appears below. You will need to submit several screenshots showing example executions of your code. Note that you may assume that the user will enter a string of less than 80 characters in length. If the user enters a longer string, inform them that the string entered is too long and ask the user to try again.

Example execution:

Enter a string and I will encrypt it as single large integer.

Welcome to Data Structures and Algorithms

Clear text:

Welcome to Data Structures and Algorithms

Number of clear text bytes = 41

Welcome to Data Structures and Algorithms is encrypted as 317817076350145785260656997739621373931469119807217110529280649332942724728174120224095587842484635884305367159163265896397856056390477056525611299805189287161133817602808317806202994211855425964737022224210974561644533817599450919039345942975178915818105632933959978787221138943336909734004773052722627400695

Result of decryption: Welcome to Data Structures and Algorithms

Part 2 Grading:

Working code performing key generation, encryption and decryption and using doubly linked lists to hold BigIntegers: 70%

Javadoc describing worst and best case big theta values

and pre-and post-conditions in your linked list class. : 20%

Overall presentation including several screenshots: 10%