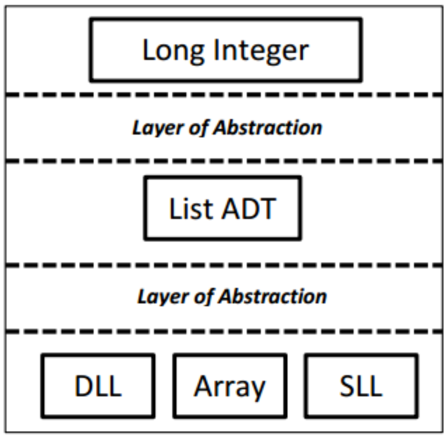
CS 610 – 108	Programming Project A Long Integer Data Type	Spring 2016

Many applications, such as cryptography, require very large integers (much larger than would fit into 32- bit or 64-bit integer types). The standard library of many languages includes a "Big Integer" data type. In this project you will implement such a **Long Integer Abstract Data Type**, where there is no predefined limit on the maximum value that can be stored. Such a Long Integer can have an arbitrary number of digits (i.e., tens of thousands of digits, or millions of digits, or more…). The following abstract operations are defined for the Long Integer ADT: **ADD, SUBSTRACT, MULTIPLY, POWER**, and various comparison operations (e.g., **LESS THAN**). The Long Integer ADT also has two properties associated with it: **sign** and **number of digits**. There are many possible implementations of a Long Integer ADT.

For this project you will implement the Long Integer ADT using two data structures: **singly linked list** and **array**. You will experimentally test the performance of the Long Integer for both data structures.



Note that changing the implementation of the Long Integer from one data structure to another would usually require you to rewrite every operation associated with the Long Integer. This would take an unnecessary amount of time for this project. Thus, there needs to be a layer of abstraction between the Long Integer and the data structure(s) used to implement it.

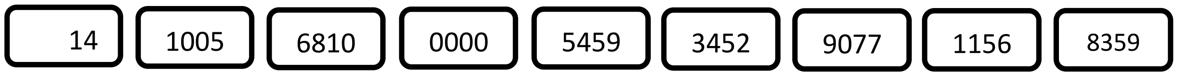
Specifically, for this project the Long Integer ADT will be implemented using another ADT: a simplified version of the List ADT called the SimpleList ADT. The Long Integer will be implemented using the SimpleList ADT and its abstract operations (defined on Page 3). With this design the Long Integer will never directly interact with the data structure it is implemented with. To change the data structure implementation of the Long Integer you will only need to re-implement SimpleList's abstract operations, a significantly simpler process.

When representing a Long Integer, each position of the SimpleList ADT will store at most **4 digits** of the overall number. Every node will store a positive value, as the sign will be represented as a property of the Long Integer. When adding two such 4-digit integers the result can be a 5 digit integer. This 5 digit integer must be separated into a 4 digit integer containing the 4 rightmost (least significant) digits and one overflow digit (the most significant digit) that will be added to the next node to the left. To perform addition of two Long Integers of c digits each you will need to perform c/4 additions of 4 digit integers and potentially handle c/4 overflow digits. Similarly, multiplication of two 4 digit numbers may result in an 8 digit number, which is not allowed to be stored in a SimpleList position.

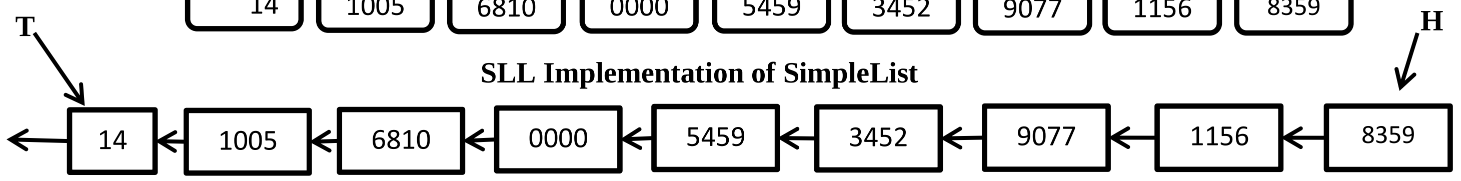
For this project you will be provided with a skeleton program that declares (but does not define) the necessary classes and methods that are needed for this project. You must use this skeleton program. The required operations are listed on the following page but the required implementation of each operation will be described in class.

A = 14100568100000545934529077115 68359

A’s SimpleList



SLL Implementation of SimpleList

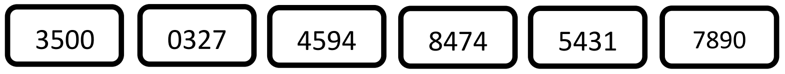


Array Implementation of SimpleList

…	**4**	**3**	**2**	**1**	**0**
…	Value: 5459 Index: 4	Value: 3452 Index: 3	Value: 9077 Index: 2	Value: 1156 Index: 1	Value: 8359 Index: 0

B = -35000327459484745431 7890

B’s SimpleList



SimpleList Abstraction Operations

Operation	Description
insertFirst(int v)	Adds a new node with value v at front of the list
insertLast(int v)	Adds a new node with value v at the end of the list
first()	Returns first Position of the list
last()	Returns last Position of the list
isFirst(Position p)	Determines if Position p is the first position of the list
isLast(Position p)	Determines if Position p is the last position of the list
before(Position p)	Returns the Position before Position p (error if p is first position)
after(Position p)	Returns the Position after Position p (error if p is the last position)
isEmpty()	Returns Boolean value indicating if the list has no nodes
size()	Returns the number of positions in the list as an integer

Long Integer Abstract Operations

Operation	Description
LongInteger(String s)	Initializes a new Long Integer which represents the number in String s. String s is in the format: [-?][1-9][0-9]*
output()	Prints the Long Integer to standard output in format: [-?][1-9][0-9]*. This method does not return anything.
getSign() / setSign()	Returns the sign of the long integer (represented by a boolean isNegative), gets the sign as a Boolean (true for negative, false for positive)
getDigitCount()	Returns the number of decimal digits in the Long Integer as an integer. This value has to be computed dynamically.

equalTo(LongInteger i)
lessThan(LongInteger i)
greaterThan(LongInteger i)
	Return Boolean values indicating if the Long Integer is less than, equal to, or greater than Long Integer i, respectively. Each must be implemented as separate functions.

add(LongInteger i)	Adds the Long Integer to Long Integer i and returns the result as a new Long Integer. Must be implemented separately from subtract, but add and subtract can call each other when necessary.
subtract(LongInteger i)	Subtracts the Long Integer i from the Long Integer and returns the result as a new Long Integer. Must be implemented separately from add, but add and subtract can call each other when necessary.
multiply(LongInteger i)	Multiplies the Long Integer by Long Integer i and returns the result as a new Long Integer
power(int p)	Raises the Long Integer to the power p (a regular integer) and returns the result as a new Long Integer. Note that no more than 2lg(p) calls to multiply are allowed.

Note: All of the above Long Integer operations, except LongInteger(String s) and setSign, are immutable.

Utility Operations

Operation	Description
overflow (int t)	Returns the overflow digits (5th – 8th digits in an integer with more than 4 digits)
underflow (int t)	Returns the underflow digits (1st – 4th digits in an integer with more than 4 digits)
digits(int t)	Returns the number of decimal digits in a regular integer as an integer

The above utility operations are used by the Long Integer operations to **(1)** enforce the “4-digits per SimpleList node rule” and **(2)** provide functionality needed for the various Long Integer operations as needed.

This project is separated into three major steps. Each step is broken down into several parts. You cannot move on to the next step of the project until every part of the previous step is implemented correctly or you are cleared to move onto the next step (you will be explicitly told that you cannot move on when you receive a grade for a step). Correcting a part of a step after a grade was given will result in no additional credit. You must execute all of the specified test cases or you will receive a grade of zero.

Step 1 (40 pts)	**SLL SimpleList, Utility Operations, Basic Long Integer Operations**	**Due Date:** 3/24

Using the C++ or Java skeleton program provided on Moodle, implement a Singly linked list SimpleList, the initial parts of the Long Integer ADT, and the specified utility operations. **NOTE:** The given projects are incomplete and only include the bare minimum needed to get the project to compile. While you are free to add onto what is given (e.g., add attributes, methods) you cannot remove or modify what is given.

- Implement singly-linked-list-based SimpleList (SLLSimpleList) (10 pts)
- Implement all utility operations (2 pts each)
- Implement Long Integer initialization (5 pts)
- Implement Long Integer output (5 pts)
- Implement getSign and setSign (1 pt each)
- Implement getDigitCount (2 pts)
- Implement equalTo, lessThan, and greaterThan (as separate operations) (5 pts total)
- Implement and automatically run all Step 1 test cases (5 pts)

Step 2 (35 pts)	**Long Integer Arithmetic Operations**	**Due Date:** 4/14

In this step you will implement the core functionality of the Long Integer, the arithmetic operations, using the abstract operations defined for the SimpleList.

- Implement add (5 pts)
- Implement subtract (5 pts)
- Implement multiply (15 pts)
- Implement power (5 pts)
- Implement and automatically run all Step 1 and Step 2 test cases (5 pts)

Step 3 (25 pts)	**Array SimpleList, Experimental Analysis**	**Due Date:** 4/28

- Implement an array-based SimpleList called ArraySimpleList (15 pts)
- Run the Step 1 and Step 2 test cases using both the Singly Linked List and Array implementations of the SimpleList. You cannot modify the LongInteger methods to get the different implementations to work.
- For each test case compute the time (in C++ use ticks, in Java use nanoseconds) needed to complete **each individual test case** using the SLL and Array implementations of SimpleList. Include the running times of each test case for both implementations in a comment in the source file that contains your main function/method. (10 pts)

Due Dates and Late Submissions

Each step is due by 11:59:59PM on the given due date. You may submit a step up to one week late at a penalty of two points per day. For example, if you submit Step 1 three days late the maximum possible grade for Step 1 would be 34/40. If you cannot complete a step within one week of the due date you will need to meet with me privately during my office hours to discuss your progress on the project.

Implementation Guidelines

You will receive no credit for your submission if you do not follow these guidelines. **READ THESE CAREFULLY**. If you are unsure if you are following these guidelines do not hesitate to contact me.

1. You can use either C++ or Java. You **must** use the skeleton project code provided on Moodle.
2. Your SimpleList ADT must be implemented using a singly linked list data structure for Steps 1 and 2. This has already been started for you but the code provided is mostly incomplete. Your Long Integer must be implemented using the SimpleList ADT, as given.
3. Your Long Integer must never directly access the singly linked list or array data structure. All LongInteger operations must be implementing using SimpleList's abstract operations.
4. You must implement all of your own data types and data structures. You cannot use any built in data types (aside from Strings for initialization) or include any data type libraries.
5. The correct implementation of the required functions is dependent on the correct implementation of the specified data types and data structures. If you have the required functions working with incorrect data types of data structures you will receive no credit.
6. Use of the long primitive type (or double, or long long, or anything similar) is not permitted at any time. Only 32 bit ints can be used.
7. A Long Integer can NEVER be converted into a string. No exceptions. Additionally, Strings can only be used in main and LongInteger(String s).
8. A Long Integer can never be converted into another data structure or data type.
9. The Long Integer must use the minimum number of nodes needed to represent a number.
10. The Long Integer operations must be implemented using the algorithms described in class. No other implementations will receive credit. Additionally:
    - **a.** The add and subtract operations must be implemented separately. They can call each other. Don't handle subtraction in add and don't handle addition in subtract.
    - **b.** The multiply operation cannot be implemented as repeated addition.
    - **c.** The operation power(p) must call multiply and you can use at most 2lg(p) calls to multiply (i.e., power cannot be implemented as repeated multiplication).
11. All of the specified Long Integer operations are immutable operations. The Long Integer, and the operation argument(s), must remain unchanged. You may add operations which are mutable in support of the required operations, but any operation that is mutable should be a private method of the LongInteger.
12. Every step **must** run the required test cases.
13. You can implement additional functions as needed, as long as they do not violate any of the above guidelines. You can also name any function or variable as you wish, as long as it's a meaningful name.

Submission Guidelines

- Email your submission to the TA me76@njit.edu with the subject "CS 610 - 108 Project" followed by the step you are submitting, e.g. "CS 610 - 108 Project Step 1."
- Submit only what is required for the current step.
- Submit **ONLY** the source code for your submission. Only submit .cpp/.cc/.h/.hh/.c/.java files. Combined multiple such files into a single Zip file.
- Your code **MUST** compile and execute on NJIT's AFS computers (afsconnect1.njit.edu or afsconnect2.njit.edu) using either g++ or javac.
    - Test that your program compiles on AFS early and often. Even with Java.
    - This WILL be a problem if you are using Microsoft Visual Studio. I guarantee it.
    - Do not use non-standard libraries (e.g., conio.h).
    - If your Step 1 submission does not compile on AFS you will be given 24 hours to correct the problem with no penalty. After 24 hours it will be considered late starting from the day the TA determined your project did not compile.
    - In all other situations, if your submission does not compile on AFS it will receive a grade of zero.
    - Mention any specific instructions for compiling in your submission email.
- You can modify a previous step's code for a future step, if necessary. However, no credit will be given.

Academic Honesty Policy
Copying code from the internet, another student, or from any other source (including books, YouTube videos, 8-tracks, etc.), even if it is just a single line of code, is strictly prohibited. Any cases of copying will result in everyone involved being reported to the Dean of Students and everyone involved will receive a zero for the entire project. The course's academic honesty policy, specified in the syllabus, will be enforced.

- Your code is yours and yours alone. **You are responsible for it.**
- **This is not a group project.** Work by yourself at all times.
- Do not share your implementation or code with another student.
- Do not look at another student's code.
- Do not share your code as a "guide" for other students.
- Do not put your code in a publicly accessible place (e.g., Github)

This academic honesty policy will be **strictly** enforced.

Test Cases
To receive any credit your project must automatically run, without any user input, the specified test cases for each step. If your submission does not automatically run the required test cases you will receive a grade of zero for the step. Without these test cases the TA will not be able to determine if your project works. To reiterate: **this program requires no user input.**

A = 2222
B = 99999999
C = -246813575732
D = 180270361023456789
E = 1423550000000010056810000054593452907711568359
F = -350003274594847454317890
G = 29302390234702973402973420937420973420937420937234872349872934872893472893749287423847
H = -98534342983742987342987339234098230498203894209928374662342342342356723423423
I =
84364131684386183513516846948354348943643518468434351684843516846843153846843138468131538431
35138413513843813513813138438435153454154515151513141592654543515316848613242587561516511233
246174561276521672162416274123076527612

Step 1 Test Cases
1. Initialize Long Integers A-I.
2. For each Long Integer traverse it's SimpleList and print the value stored at each position (put a space between each node). This test is to make sure each node stores at most 4 digits.
3. Print each Long Integer to standard output using output().
4. Print the sign and number of digits of each Long Integer.
5. Store the value of A and B in regular ints and apply all of the utility methods.
6. For each Long Integer compare it to A-I using equalTo, lessThan, greaterThan (i.e., apply all three methods pair-wise, including on itself).

Step 2 Test Cases
1. For each Long Integer add it to every other Long Integer (one at a time) and print the result
2. For each Long Integer subtract it from every other Long Integer (one at a time) and print the result
3. For each Long Integer multiply it by every other Long Integer (one at a time) and print the result
4. Raise each Long Integer to the 5th, 10th, 20th, and 30th power.
5. Compute the following (in alphabetical order): J = B + C, K = C + D, L = I + I, M = A + I, N = B + K, O = A – D, P = C – D, Q = D – C, R = L – L, S = P – O, T = N – Q, U = A * D, V = B * C, W = D * D, X = O * I, Y = J * P, Z = M * N