CS61B Lab #3

Please finish this lab today, preferably during the lab period.  It is due
at midnight tonight.

0. Summary
----------

    1. An graded quiz (ungraded---it's for your use) covering
       pointer-related concepts.

    2. Exercises with some classes from the Java library and their
       performance.  We start with an implementation of a program to
       print duplicated entries in a sequence, showing how, thanks to the
       Java libraries API (Application Programmer's Interface), two entirely
       different representations of the same data (as a linked list and as
       an array) can be handled by exactly the same method.

       In 2a, we show that the two representations differ considerably
       in speed of execution, and how, with a little more use of
       abstraction, we can (in some cases) overcome the performance
       difference while still using the same code for both representations.
       The new abstraction is called a "list iterator", a form of "iterator",
       a kind of "finger into a data structure".  This part asks you to
       examine the API for ListIterator and find a slightly different
       encoding that makes fuller use of their features.

       In 2b, we ask you to change the behavior of Dups2.java in a
       "useful" fashion by poking around in the library documentation
       for a suitable method, rather than implementing the change completely
       from scratch.

       2c asks for further exploration of the data structures supported
       by the Java library, this time to find a faster way to look up things.
       Again, you are to modify Dups2 appropriately.

    3. Questions from bug-submit and Piazzza indicate that some of you could
       stand a bit more practice with the debugger, so we finally include an
       exercise in using gjdb to step through and debug a procedure related
       to a HW2 problem.

1. Quiz
-------

Please fill out the quiz for today on the lab page.


2. Collection Classes in the Java Standard Library
--------------------------------------------------

So far, we've seen fairly primitive types for representing lists of
things.

  * Arrays:
      + Advantages: fast random access to items.
      + Disadvantages: hard to insert items or delete items.
  * Hand-made linked lists (e.g., IntList):
      + Advantages: easy to expand, insert, delete.
      + Disadvantages: random access (as opposed to in-sequence access)
        is slow, requires more space (for pointers) than arrays, and
         pointer manipulation can be tricky.

Both represent the same things (sequences of things), but their syntax
is very different, so it's hard to switch from one to the other if you
change your mind.

The Java library has types to represent collections of objects,
including

  * Lists (sequences) of objects: ArrayList, LinkedList.
  * Sets of objects: TreeSet, HashSet.
  * Maps (dictionaries): TreeMap, HashMap.

all of which are in the package java.util.  A *package* is a set of
(supposedly related) classes and subpackages. As elsewhere in Java,
the notation 'java.util.ArrayList' means "The class named "ArrayList"
in the (sub)package named "util" in the package named "java".

The names of the classes in the three categories above reflect
implementations, but they "publicize" very similar APIs ("Application
Programmer Interface") to the outside.  Thus, it is easy to change
from using an ArrayList to using a LinkedList.

There are types List, Set, and Map in java.util as well.  These are
Java *interfaces* that various of the classes above *implement*.  As
we'll see, A Java interface is an abstraction of an API (without any
implementation) that may be shared by many classes. By specifying such
an interface as the type of a variable, you get a program that is able
to accommodate any of the implementing classes without having to
change most of the program.

For example, consider the program in Dups1.java, which comes with
this lab.  Compile this program and execute it with

    java Dups1 linked < activities.txt

and see what happens.  Now try

    java Dups1 arrays < activities.txt

You should get identical results.

2a. ListIterators
-----------------

There is a problem, however.  Assuming that LinkedList actually uses
linked lists (like our IntList examples), what can you say about the
cost (the number of operations performed either by your program or
the library) in executing 'java Dups1 linked' ?  Try the Unix time commands

    time java Dups1 linked < some-words.txt
    time java Dups1 array < some-words.txt

and compare the results. (We've also set up 'gmake time' to run these and
a couple of other timings.  You might want to look at Makefile to how that
is done).

The program Dups2.java uses the ListIterator abstraction to get around
this problem and put the two implementations on a more nearly equal
footing.  The interface types Iterator and ListIterator describe
"moving fingers" that effectively point at elements in the middle of
one of the collection types from the Java library and allow a program to
move through the collection sequentially, regardless of how the
collection is actually implemented.  Iterators move only forward, and
ListIterators move in either direction (and make sense only for Lists,
which have an order).  Again time the two programs:

    time java Dups2 linked < some-words.txt
    time java Dups2 array < some-words.txt

(In trying to make sense of the result, it's useful to know that
LinkedLists are a bit more sophisticated than IntLists: each item has
two pointers, one to the next and one to the previous element.  Also,
the list itself has a pointer to both the first and last elements.)

Read and understand both programs (the Javadoc documentation we

assigned for this lab should be useful here).  The Dups2.duplicates
method uses two int variables, m and n, to keep track of positions in
the list L (and stop once we have scanned back to the position of
element we are checking for duplication).  By making better use of the
ListIterator interface, we can remove both of these variables.
Rewrite Dups2.java so that duplicates does not use any variables other
than result, p1, and p2.

2b. Utilities
-------------

Currently, the list of duplicate words gets printed in whatever order
the words occur in the text. By adding one short line to the program,
you can arrange for the list to be printed in sorted order.  See if
you can find how to do this (there is a hint about where to look
towards the top of the file) and change Dups2.java accordingly.  With
a bit more searching (this time in the documentation for
java.lang.String), you may be able to find out how to get the sort to
ignore the case of letters, instead of putting all capital letters
before all lower-case letters.

2c. Sets
--------

Dups1 and Dups2 both use the .contains method of ArrayList to make
sure they don't add duplicates to results.  The problem is that on
Lists, these methods scan the list sequentially, looking for
duplicates.  While this is not much of a problem for the small inputs
we've been using, it can cause difficulties with larger ones.  The
Java library provides collections that are faster for the purpose of
collecting sets of objects without duplicates.  Create a new version
of Dups2 that uses a java.util.TreeSet for the variable results in the
.duplicate method.  However, make sure that the method still returns a
List<String> (which a TreeSet<String> is not).  (Hint: A TreeSet is a
kind of Collection<String>.  Take a look through the documentation of
ArrayList.)

3. A Little More Debugging
--------------------------

The program natural.java gives another implementation of the
naturalRuns method you did for homework, but this time with Java
library Lists.  Unfortunately, the program does not work, as you can
see by compiling it (don't forget the -g option) and executing

    java Natural 1 3 7 5 4 6 9 10

You can probably figure this out by inspection, but try it in the
debugger.  Step through the code of naturalRuns, and make sure you can
see what is happening (that is, that you can execute each statement in
turn and see the values of the variables and observe how they get
updated).  Now is your chance to ask the TA if the debugger doesn't
seem to work for you.

While you're at it, stop execution of your program at the start of the
naturalRuns function, and print the value of L (the parameter to the
function).  The command

    p L

will result in an uninformative output that basically tells you that L
is some kind of list.  But try

    p L.toString()

This is actually a call that the debugger will execute, producing a
handy String representation of L.  You can produce the same string (in the
debugger and in real Java) by typing

    p "" + L

because the "+" operator on Strings, if given an operand that is a
non-String object, will convert it into a String by calling toString()
on it.

4. Turn in the your lab work
----------------------------

Use svn commit to commit all your lab3 files (Dups2.java and
Natural.java at least).  Check that your submission attempt was
successful (that is, if you don't know how to do this, now is a great
time to find out!).