Wellesley College ◊ CS111 Computer Programming and Problem Solving ◊ Spring 2002

FINAL EXAM REVIEW PROBLEMS

The CS111 final exam is a self-scheduled exam held during the normal final exam period. It is an open book exam: you may refer to any books, notes, and assignments. You may not talk to other people about the exam before or after taking it, nor may you use a computer during the exam.

Here is a list of topics covered by the course that may be tested on the final exam:

- **problem solving patterns:** divide/conquer/glue, recursion, iteration (tail recursion, loops);
- **abstraction:** method abstraction, data abstraction, abstraction barriers/contracts/APIs.
- **modularity:** constructing programs out of mix and match parts (e.g. generators, mappers, filters, accumulators) that use standard interfaces (lists, arrays, vectors).
- **control structures:** sequencing, method invocation, conditionals (`if`/`else`), loops (`while`, `for`), `return`.
- **data structures:** objects, lists, arrays, vectors.
- **models:** execution diagrams, invocation trees, object diagrams, box-and-pointer-diagrams, inheritance hierarchies.
- **Java methods:** declaration vs. invocation; parameter declaration and use; formal vs. actual parameters; scope of parameter names; `void` vs. non-`void` return types; using `return` to return result; invocation model (create a frame in Java execution model).
- **Java class declarations:** instance variables, class (static) variables; constructor methods, instance methods, class (static) methods; inheritance; abstract classes and methods.
- **Java statements:** local variable declarations, method invocations, assignments, `if`/`else` conditionals, `while` loops, `for` loops, `return`; security keywords (`public`, `protected`, `private`)
- **Java expressions:** literals (numbers, booleans, characters, strings), variable references (instance variables [e.g., `foo.x`], array subscripts [e.g., `foo[i]`], `this`, `super`), constructor method invocations (`new`), non-`void` method invocations, primitive operator applications (arithmetic, relational, logical).
- **microworlds:** BuggleWorld, TurtleWorld, PictureWorld, ListWorld, Java Graphics, AnimationWorld.

Below are some problems intended to help you review material for the final exam. The problems do **not** cover all of the topics listed above, so you should also review your notes and assignments.

The problems range in difficulty. Some problems are from previous final exams. Others (with some rewriting) could be turned into reasonable final exam problems. Others are too long or complex for a final exam problem, but review material that is covered on the exam.

The problems are not in any particular order, so you should not feel compelled to do them in order. Rather, you should first work on those problems that cover material in which you think you need the most practice. To help you decide which problems to work on, each problem lists the concepts that the problem covers.

Solutions to some problems will be presented during the review session. Solutions to most problems will be posted on-line by Monday, May 13.

Problem 1: Sales Statistics *(Data Abstraction, Arrays, Lists, Objects, Object Diagrams)*

The management of the Decelerate Clothing Store (specializing in "clothes that slow you down") wants to track certain statistics about customer purchases. In particular, they want to track the amount of each purchase and whether it was made with cash or credit card. Later, they want to be able to calculate statistics based on this information, such as the largest cash purchase amount, the average amount of a credit card purchase, and the percentage of credit card purchases.

The management has hired Abby Stracksen of Simplistic Statistics to implement a Java program for tracking the purchase information and calculating the desired statistics. Abby begins by designing a contract for a `History` class that maintains a history of customer purchase:

Contract for the History Class

Constructor method for the History class

```
public History (int maxEntries);
```
Returns a new `History` object that can store up to `maxEntries` purchase entries.

Instance methods for the History class

```
public void add (int amount, boolean cash);
```
Adds a new purchase entry to this history: amount is the amount of the purchase; `cash` value `true` indicates a cash purchase, `cash` value `false` indicates a credit card purchase. An attempt to add an entry is ignored if `maxEntries` entries have been stored.

```
public int size ();
```
Returns the number of entries in this history.

```
public int min (boolean cash);
```
Returns the minimum amount of a purchase made with the given `cash` value.
I.e. if `cash` is true, return the minimum purchase made with cash, otherwise return the minimum purchase made with credit card. If there are no purchases with the given `cash` value, returns the largest integer.

```
public int max (boolean cash);
```
Returns the maximum amount of a purchase made with the given `cash` value. If there are no purchases with the given `cash` value, returns the smallest integer.

```
public int average (boolean cash);
```
Returns the average amount of a purchase made with the given `cash` value. If there are no purchases with the given `cash` value, returns 0.

```
public int number (boolean cash);
```
Returns the number of purchases made with the given `cash` value.

```
public double percentage (boolean cash);
```
Returns the percentage (by number of purchases) of all purchases made with the given `cash` value. Returns 0 if there have been no purchases.

Abby has also defined the contract for a `Purchase` class that models an individual purchase:

Contract for the Purchase class:

Constructor method for the Purchase class

`public` `Purchase (`**`int`** `amount,` **`boolean`** `cash);`
Returns a new `Purchase` object that with amount `amount` and cash/credit mode `cash`. A `cash` value true indicates a cash purchase, `cash` value false indicates a credit card purchase.

Instance methods for the Purchase class

`public int` `getAmount ();`
Returns the amount of this purchase.

`public void` `setAmount (int newAmount);`
Sets the amount of this purchase to be `newAmount`.

`public boolean` `getCash ();`
Returns the cash/credit mode of this purchase.

`public void` `setCash (`**`boolean`** `newCash);`
Set the cash/credit mode of this purchase to be `newCash`.

Abby's co-worker Emil P. Mentor has begun to implement Abby's contract. Here is his implementation of the Purchase class:

```
public class Purchase {

  // Instance Variables
  private int amount;
  private boolean cash;

  // Constructor Method
  public Purchase (int i, boolean b) {
    amount = i;
    cash = b;
  }

  // Instance Methods
  public int getAmount () {
    return amount;
  }

  public void setAmount (int newAmount) {
    amount = newAmount;
  }

  public boolean getCash () {
    return cash;
  }

  public void setCash (boolean newCash) {
    cash = newCash;
  }
}
```

Emil also started to implement the `History` class, but was called away on a business trip. Here's how far he got:

```
public class History {

  // Instance Variables:
  private Purchase [ ] purchases;
  private int size;

  // Constructor Method:
  public History (int maxEntries) {
    purchases = new Purchase[maxEntries];
    size = 0;
  }

  // Instance Methods:
  public void add (int amount, boolean cash) {
    if (size < purchases.length) {
      purchases[size] = new Purchase(amount, cash);
      size = size + 1;
    }
  }

  // I still need to finish the other methods! - Emil -
}
```

Part a. Based on Emil's implementation, draw an object diagram that shows the result of executing the following statements. Your diagram should include the local variable `h` and all objects that are accessible from `h` via some sequence of pointers.

```
History h = new History(5);
h.add(82, false);   // credit purchase
h.add(53, true);    // cash purchase
h.add(178, false);  // credit purchase
```

Part b. Finish Emil's implementation by fleshing out the missing instance methods from his `History` class.

Part c. Your colleague Bud Lojack believes that Emil could also have written the constructor method for `Purchase` in either of the two ways below:

```
  public Purchase (int amount, boolean cash) {
    amount = amount;
    cash = cash;
  }

  public Purchase (int a, boolean c) {
    int amount = a;
    int cash = c;
  }
```

Is Bud right? Explain.

Part d. On his desk, Emil left the following notes about alternative implementations of the Purchase class:

> Many ways to implement `Purchase` instance. E.g.
> 1) As an array of two integers. Slot 0 = amount; Slot 1 = cash (use 0 for `false`, 1 for `true`).
> 2) As an integer list with two elements. First element = amount; second = cash (use 0 for `false`, 1 for `true`)
> 3) As a single positive/negative integer. Amount is the absolute value. Positive indicates cash; negative indicates credit.
> 4) As a single positive integer `n`. Amount = `n / 2`; cash = `n % 2`, where 0 is `false`, 1 is `true`.

Based on Emil's notes, provide four alternative implementations of the `Purchase` class that all satisfy the `Purchase` contract.

Part e. Bud Lojack thinks that Emil should have made the `amount` and `cash` instance variables of the `Purchase` class **`public`** rather than **`private`**. Explain to Bud why this is a bad idea.

Part f. Emil returns from his trip, and says that he had an epiphany about an alternative representation of `History` instances that does not involve `Purchase` objects. Instead, Emil thinks a history instance can be implemented as an object with three instance variables:

1. The `maxEntries` integer.
2. An integer list `cashes` holding the amounts of the cash purchases.
3. An integer list `credits` holding the amounts of the credit purchases.

i. Repeat part (a) using this representation of the `History` class.

ii. Write an alternative implementation of the `History` class based on this representation.

Part g. The implementations of the `History` class considered above are only two of many possible implementations. What are some other implementations? For each implementation you can think of, draw an object diagram of the example from part (a).

Problem 2: List Partitioning *(Iteration, Recursion, Lists)*

The following `partition()` method takes an integer named `pivot` and a list of integers named `L` and partitions the list into two lists:

1. All the elements in `L` less than or equal to `pivot`.
2. All the elements in `L` greater than `pivot`.

The two resulting lists are returned as an array containing two integer lists. All `IntList` operations are prefixed with "`IL.`"

```
public static IntList [] partition (int pivot, IntList L) {
  return partitionTail(pivot, L, IL.empty(), IL.empty());
}

public static IntList [] partitionTail (int pivot, IntList list,
                                        IntList lesses, IntList greaters) {
  if (IL.isEmpty(list)) {
    return twoLists(lesses, greaters);
  } else if (IL.head(list) <= pivot)
    return partitionTail(pivot,
                         IL.tail(list),
                         IL.prepend(IL.head(list), lesses),
                         greaters);
  } else {
    return partitionTail(pivot,
                         IL.tail(list),
                         lesses,
                         IL.prepend(IL.head(list), greaters));
  }
}

// Auxiliary method used by partitionTail() that glues two lists into an array.
public static IntList [] twoLists (IntList L1, IntList L2) {
  IntList [] result = new IntList [2];
  result[0] = L1;
  result[1] = L2;
  return result;
}
```

Part a. The `partitionTail()` method is a tail recursive method that specifies an iteration in four state variables named `pivot`, `list`, `lesses`, and `greaters`. Any iteration can be characterized by how the values of the state variables change over time. Below is a table with four columns, one for each state variable of the iteration described by `partitionTail()`. Each row represents the values of the parameters to a particular invocation of `partitionTail()`.

pivot	list	lesses	greaters

Suppose that the list `A` has the printed representation `[7,2,3,5,8,6]`. Fill in the above table to show the parameters passed to successive calls to `partitionTail()` in the computation that begins with the invocation `partition(5, A)`. You have been provided with more rows than you need, so some rows should remain empty when you are done.

Part b. It is possible to express any iteration as a **`while`** loop. Give a complete definition of a class method `partitionWhile()` that behaves just like the above `partition()` method except that it uses a while loop rather than tail recursion to express the iteration of `partitionTail()`.

Part c. In the above `partition()` method, elements in the two returned lists are in a relative order opposite to their relative order in the original lists. For instance, partitioning the list `[1,4,8,3,6,7,5,2]` about the pivot `5` yields the lists `[2,5,3,4,1]` and `[7,6,8]`.

Suppose that we want the elements of the resulting lists to have the same relative order as in the original list. If we are provided with a list reversal method `reverse()`, we can easily accomplish this by reversing the two lists before putting them into the result array. That is, we can change the line

```
return twoLists(lesses, greaters);
```

within `partitionTail()` to be

```
return twoLists(reverse(lesses), reverse(greaters));
```

An alternative to using `reverse()` to achieve this behavior is to write `partition()` as a non-tail recursive method. Flesh out the following skeleton of `partitionNotTail()` which partitions the elements of a list about the pivot but maintains the relative order of the elements in the resulting lists:

```
public static IntList [] partitionNonTail (int pivot, IntList L) {
  if (isEmpty(L)) {
    return twoLists(IL.empty(), IL.empty());
  } else {
    IntList [] subresult = partitionNonTail(pivot, IL.tail(L));
    // flesh out the missing code here ...
  }
}
```

Part d. An important use of the `partition()` method is a sorting algorithm known as `quicksort`. Here is the idea behind quicksort:

> To sort the elements of a list, partition the elements of the tail of the list around its head into result lists that we'll call *lesses* and *greaters*. Then result of sorting the whole list can be obtained by appending the result of sorting *lesses* to the result of prepending the head of the list to the result of sorting *greaters*.

For example, if the initial list is `[5,2,8,3,6,7,1,4]`, then partitioning the tail of the list around the head (5) yields:

```
lesses = [4,1,3,2]
greaters = [7,6,8]
```

By wishful thinking, sorting `lesses` will yield `[1,2,3,4]` and sorting greaters will yield `[6,7,8]`. The result of sorting the original list is the result of appending `[1,2,3,4]` to the result of prepending `5` to `[6,7,8]`.

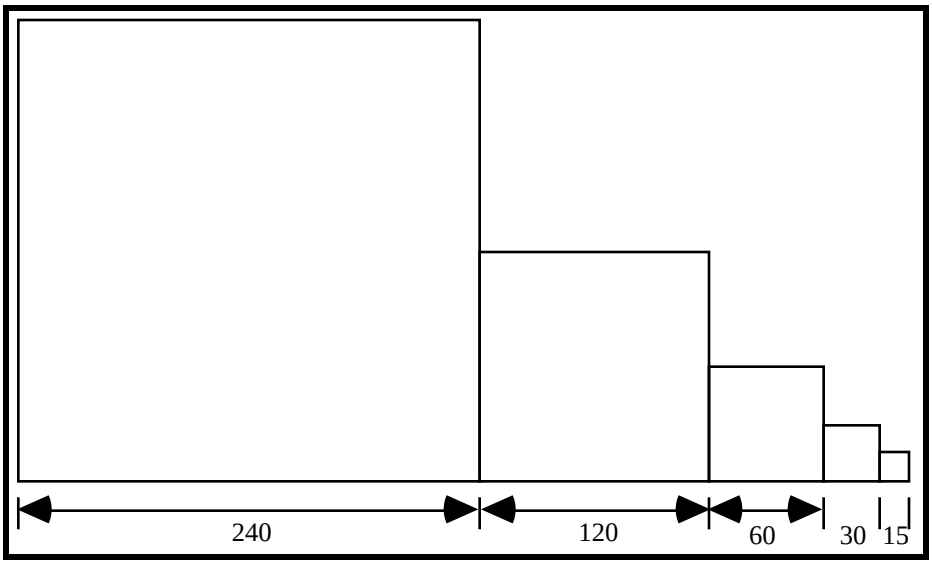
Flesh out a method **`public static`** `IntList quicksort (IntList L)` that uses this idea to sort the elements of a list. You may use `IL.append()` to append two lists.

Problem 3: Squares *(Recursion, Iteration, TurtleWorld, BuggleWorld, PictureWorld, Java Graphics)*

Below are four parts that implement a similar problem in four different microwolds that we have studied. In all parts, you should write any auxiliary methods that simplify the definition of the requested method.

Part a. Write the following instance method for a `SquareTurtle` subclass of `Turtle`:

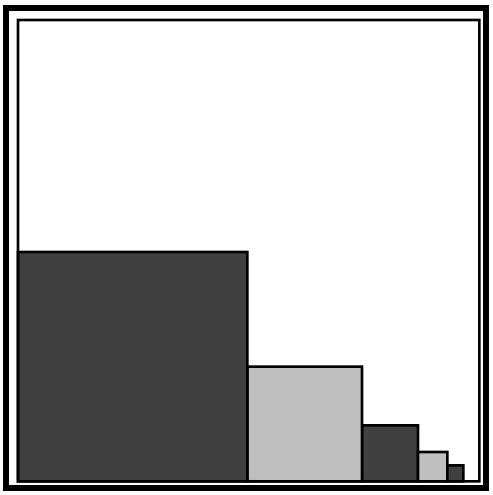
```
public void squares (int n, int len)
```
Draws `n` adjacent squares, the first of which has side length `len`, and the rest of which have a side length that is one half the side length of the previous square. After drawing the squares, the position and heading of the turtle should be the same as it was before drawing the squares. For instance, if `sara` is a `SquareTurtle` facing `EAST`, then `sara.squares(5, 240)` should draw the following picture and return `sara` to the same position and heading.



Part b. Write an instance method for a `SquareBuggle` subclass as `Buggle` that has the same interface as the `squares()` method from Part a in which each square of side length `len` is drawn as a `len` by `len` square of grid cells filled with bagels.

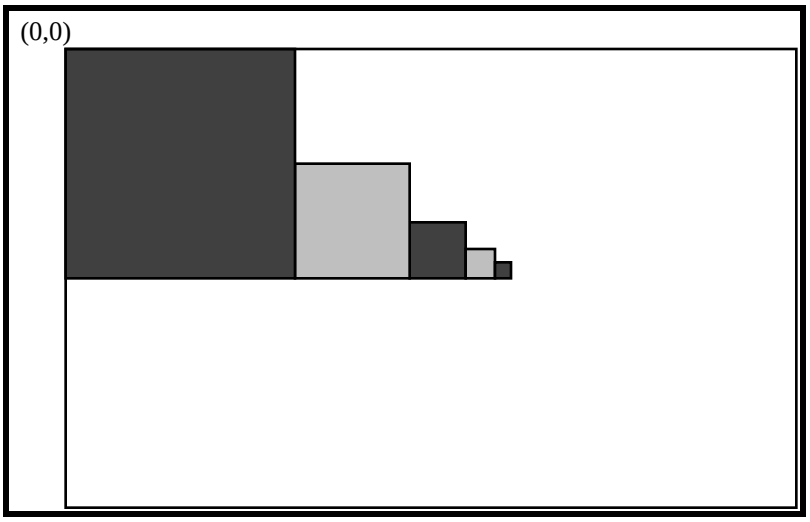
Part c. Write the following `PictureWorld` method:

`public Picture squares (int n, Color c1, Color c2)`
Returns a picture with `n` adjacent squares sitting at the bottom of the frame. The leftmost square should fill the lower left quadrant of the frame. Each subsequent square should be one-half the size of the square to its left. The colors of the squares should alternate between `c1` and `c2` from left to right. Assume that `public Picture patch (Color c)` returns a rectangular picture with color `c` that fills whole frame.



Part d. Write the following instance method squares() for a `SquareCanvas` subclass of `Canvas`.

`public void squares (Graphics g, int n, int len, Color c1, Color c2)`
Draws in this canvas a picture with `n` adjacent colored squares as shown below. The leftmost square has side length `len` and an upper left corner at (0,0). Each successive square has a side length that is half the side length of the square to its left. The colors of the squares should alternate between `c1` and `c2` from left to right.



Problem 4: Greatest Common Divisor *(Iteration)*

Wyla Lupe has been experimenting with ways to calculate the **greatest common divisor** (GCD) of two integers. The GCD of two integers A and B is the largest integer that evenly divides into both A and B. For example, the GCD of 30 and 18 is 6, the GCD of 28 and 16 is 4, and the GCD of 17 and 11 is 1.

A clever algorithm for computing GCDs was developed by Euclid in 300 B.C. (In fact, it is considered by many to be the oldest non-trivial algorithm!) Wyla has expressed Euclid's algorithm in Java as the following tail recursive `GCDTail` method. (You do **not** have to understand **why** the algorithm works!)

```
public static int GCDTail(int A, int B) {
  if (B == 0) {
    return A;
  } else {
    return GCDTail(B, A % B);
  }
}
```

Recall that `A % B` (pronounced "`A` mod `B`") calculates the remainder of `A` divided by `B`. For example, `10%3` is `1`, `10%4` is `2`, `10%5` is `0`, and `10%6` is `4`.

Part a In the following table, show the sequence of values that the parameters `A` and `B` take on in the iterative calculation of `GCDTail(95,60)`. **Important:** You have been provided with more rows than you need, so some rows should remain empty when you are done.

A	B

Part b . In the following `GCDWhile` code skeleton, implement an alternative version of Euclid's GCD algorithm that uses a **`while`** loop to express the same iteration that is expressed by Wyla's `GCDTail` method. You may wish to introduce one or more local variables.

```
public static int GCDWhile (int A, int B) {
   // Flesh out this skeleton
}
```

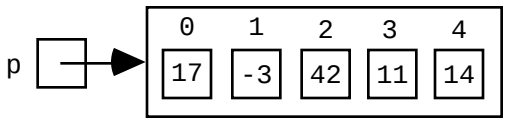
Part c. Would it be easy to re-express Wyla's `GCDTail` program as a **`for`** loop? **Briefly** explain your answer.

Problem 5: Array Reversal *(Arrays, Iteration)*

Part a. Implement the following `copyReverse()` method on integer arrays:

> **`public static int`** `[] copyReverse (`**`int`**` [] a);`
> Returns a new array that has the same length as `a` and all the elements of `a` in reverse order.

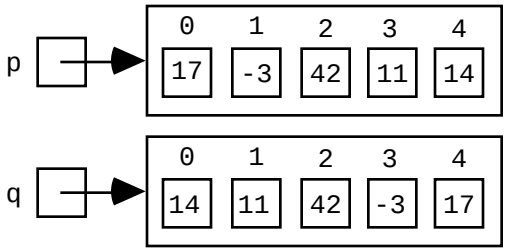
For example, suppose that `p` is the following array:



Then executing the statement

```
int [] q = copyReverse (p)
```

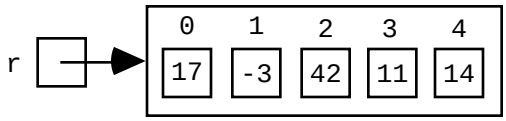
gives rise to the following diagram:



Part b. Implement the following `reverse()` method on integer arrays:

> **`public static void`** `reverse (`**`int`**` [] a);`
> Modifies `a` so that its elements are in the reverse of their original order. You should not create any intermediate arrays.

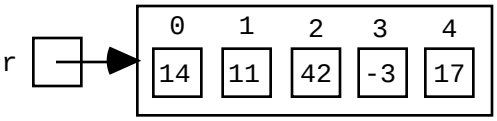
For example, suppose that `p` is the following array:



Then executing the statement

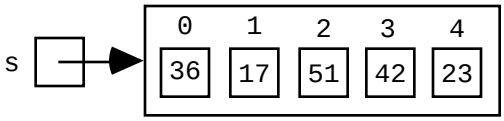
```
reverse (r)
```

changes the diagram to be:



Problem 6: Inversions *(Arrays, Iteration, Lists)*

In an integer array A, an inversion is defined to be a pair of indices (i, j) such that i < j and A[i] > A[j]. For instance, the following array `s` has five inversions: (0, 1), (0, 4), (2, 3), (2, 4), and (3, 4).



Part a. Implement the following method:

> **`public static int`** `countInversions (`**`int`**` [] a);`
> Returns the number of inversions in a.

For instance, `countInversions(s)` should return 5.

Part b. Implement the following method:

> **`public static`** `ObjectList listInversions (`**`int`**` [] a);`
> Returns a list of all the inversions in a. Each inversion (i, j) should be represented as a Point instance whose x field is i and y field is j. The order of inversions in the resulting list is immaterial.

For instance, `System.out.println(listInversions(s))` might (among many possible orderings) display:

```
[java.awt.Point[x=3,y=4], java.awt.Point[x=2,y=4], java.awt.Point[x=2,y=3],
java.awt.Point[x=0,y=4],java.awt.Point[x=0,y=1]]
```

Problem 7: Inheritance *(Inheritance)*

Consider the following five simple classes

```
class A {
  public int m1() {return 1;}
  public int m2() {return m3();}
  public int m3() {return 2;}
}
```

```
class B extends A {
  public int m1() {return m2();}
  public int m3() {return 3;}
}
```

```
class C extends B {
  public int m2() {return 4;}
}
```

```
class D extends A {
  public int m1() {return super.m2();}
  public int m3() {return 5;}
}
```

```
class E extends D {
   public int m2() {return 6;}
}
```

What is printed in the `stdout` window when the following statements are executed?

```
System.out.println((new A()).m1());
System.out.println((new B()).m1());
System.out.println((new C()).m1());
System.out.println((new D()).m1());
System.out.println((new E()).m1());
```

Problem 8: Converting Betweeen Different Forms of Iteration *(Lists, Arrays, Iteration)*

We saw in class that iterations could be expressed as tail recursions, **`while`** loops, and **`for`** loops. Each of the following parts contains a method that uses one of these forms of iteration. For each part, write two equivalent methods that use the other two forms of iteration.

Part a.

```
public static int weightedSum (IntList L) {
  return weightedSumTail (L, 1, 0);
}

public static int weightedSumTail (IntList L, int index, int total) {
  if (isEmpty(L)) {
    return total;
  } else {
    return weightedSumTail(tail(L), index + 1, (index*head(L)) + total);
  }
}
```

Part b.

```
public static boolean isMember (int n, int [] a) {
  int i = a.length - 1;
  while ((i >= 0) && (a[i] != n)) {
    i = i - 1;
  }
  return (i >= 0); // Will only be true if n is in a.
}
```

Part c.

```
public static void partialSum (int [] a) {
  int sum = 0;
  for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
    a[i] = sum;
  }
}
```

Part d.

```
public static void squiggle (Graphics g, int x1, int y1, int x2, int y2) {
  if ((x1 > 0) || (y1 > 0) || (x2 > 0) || (y2 > 0)) {
    g.drawLine(x1, y1, x2, y2);
    squiggle(g, x2, y2, y1/4, x1*2);
  }
}
```

Problem 9: Converting Betweeen Arrays and Lists *(Lists, Arrays, Iteration)*

Implement the following two methods for converting between lists and arrays of integers:

`public static int` `[] listToArray (IntList L);`
Returns an array of integers whose length is the same as the length of L and whose elements, from low to high index, are in the same order as the elements of L.

`public static` `IntList arrayToList (`**`int`**` [] a);`
Returns a list of integers whose length is the same as the length of a and whose elements are in the same order as the elements of a (from low to high index).

Problem 10: Iterative List Reversal *(Invocation Trees, Recursion, Iteration, Lists,)*

In class we studied the following recursive method for reversing a list:

```
public static IntList reverse (IntList L) {
  if (isEmpty(L)) {
    return empty();
  } else {
    return postpend(reverse(tail(L)), head(L));
  }
}

public static IntList postpend (IntList L, int n) {
  if (isEmpty(L)) {
    return prepend(n, empty());
  } else {
    return prepend(head(L), postpend(tail(L), n));
  }
}
```

This is not an efficient way to reverse a list. Each call to postpend() creates a new list whose length is one more than the length of its first argument. Furthermore, postpend() is called once for each element in the list being reversed. As a consequence, lots of intermediate list nodes are created that do not appear in the final result.

Part a. Assume that `A` is the list whose printed representation is `[1,2,3,4]`. Assuming that `reverse()` is implemented as shown above, draw an invocation treeand object diagram (i.e. box-and-pointer list representations) for the invocation `reverse(A)`. Your tree should have one node for each call to `reverse()` and one node for each call to `postpend()`.

Follow the conventions used in Problem 1 of Exam 2 for drawing invocation trees and object diagrams. That is, your nodes should have the form

`reverse(`*`ListArgument`*`) : `*`ListResult`*

`postpend(`*`ListArgument IntegerArgument`*`): `*`ListResult`*

where *`IntegerArgument`* is an integer, and *`ListArgument`* and *`ListResult`* are references to list nodes that appear in your object diagram.

Part b. An alternative technique for reversing a list is to follow the strategy one would use in reversing a pile of cards: form a new pile by iteratively removing the top card of the original pile and putting it on the new pile. When there are no more cards in the original pile, the new pile contains the cards in reverse order from the original pile.

Based on this idea, here is a table corresponding to an iterative reversal of the list `[1,2,3,4]`:

list	**result**
[1, 2, 3, 4]	[]
[2, 3, 4]	[1]
[3, 4]	[2, 1]
[4]	[3, 2, 1]
[]	[4, 3, 2, 1]

Implement this iterative list reversal strategy in a `reverse()` method in three ways: (1) using an auxiliary tail recursive `reverseTail()` method to implement the iteration; (2) using a **`while`** loop to implement the iteration; and (3) using a **`for`** loop to implement the iteration.

Problem 11: Leftist Turtles *(Tests Instance Variables, Class Declarations, Inheritance)*

Supposewe want a `LeftistTurtle` subclass of `Turtle` that remembers the number of times that it has been invoked with the `lt()` method. A `LeftistTurtle` has the same contract as `Turtle` except that it also understands the following additional instance method:

> **`public int`** `numberOfLefts ();`
> Returns the number of times that the `lt()` method has been invoked on this turtle.

Part a. Write a complete class declaration for LeftistTurtle implementing the LeftistTurtle contract.

Part b. Consider the following test of the LeftistTurtle class:

```
LefistTurtle leo = new LeftistTurtle();
leo.lt(45);
leo.rt(30);
System.out.println(leo.numberOfLefts());
```

You might expect that executing the above statements should print 1, but, depending on the implementation o the `rt()` method in `Turtle`, it might in fact print 2. Explain how this could happen. (Hint: study the implementation of the `TextTurtle` discussed in class.)

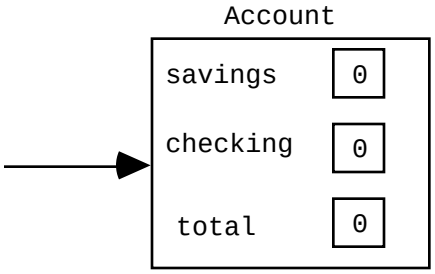
Part c. Suppose that Turtle is implemented in such a way that the test code in Part b returns 2. Modify your implementation of `LeftistTurtle` so that numberOfLefts() returns only the number of lt() method calls and does not count the number of rt() method calls. You should only change `LeftistTurtle`; you should **not** change `Turtle`!

Problem 12: Bank Accounts *(Data Abstraction, Lists)*

Suppose that there is a Java class `Account` that represents bank accounts. A straightforward way to represent a simple bank account is to keep track of three integer instance variables representing, respectively, the savings account balance, the checking account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). Assume further that the constructor method for the class `Account` has zero parameters. With this straightforward approach, the constructor method is

```
public Account ( ) {
   this.savings = 0;
   this.checking = 0;
   this.total = 0;
}
```

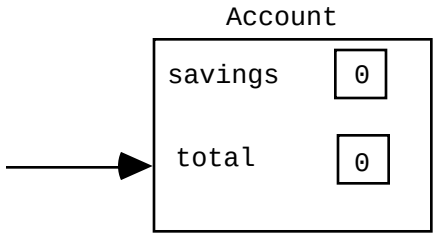
resulting in the following object diagram representation:



Below is a Java class that implements this straightforward representation.

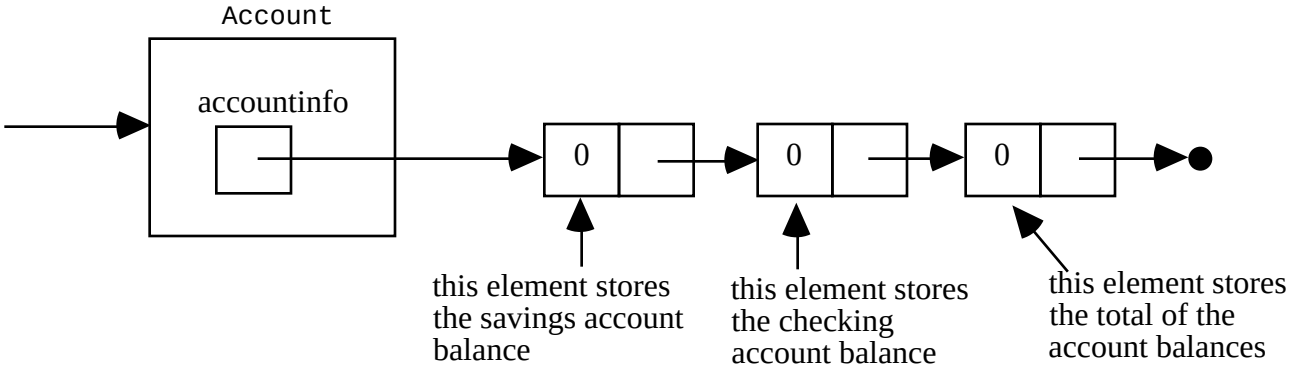
```
class Account {

  private int savings;
  private int checking;
  private int total;

  public account () {
    this.savings = 0;
    this.checking = 0;
    this.total = 0;
  }

  public int getSavings() {return this.savings;}
  public int getChecking() {return this.checking;}
  public int getTotal() {return this.total;}

  public void depositToSavings(int amountToAdd) {
    this.savings = this.savings + amountToAdd;
    this.total = this.total + amountToAdd;
  }

  public void transferFromSavingsToChecking(int transferAmount) {
    this.savings = this.savings - transferAmount;
    this.checking = this.checking + transferAmount;
  }

  public void withdrawFromChecking(int withdrawalAmount) {
    this.checking = this.checking - withdrawalAmount;
    this.total = this.total - withdrawalAmount;
  }
}
```

Part a. An alternative representation for bank accounts is to keep track of only two integer instance variables representing, respectively, the savings account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). This approach results in the following object diagram representation:



Write an implementation of the `Account` class that uses this alternative representation. It is important to note that it is only the *internal representation* of the bank account that has changed. The *external interface* to the bank account itself remains the same. Additionally, every instance method should behave the same as previously.

Part b. Yet another representation for bank accounts is to use only one instance variable: an integer linked list that has three elements. The elements store, respectively, the savings account balance, the checking account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). The resulting object diagram representation is:



Write an implementation of the `Account` class that uses this representation. As in Part a of this problem, it is important to note that it is only the *internal representation* of the bank account that has changed. The *external interface* to the bank account remains the same. Additionally, every instance method should behave the same as previously.

Your solution will use the `IntList` class discussed in class to represent an integer linked list. When writing your code, you may assume, that `IntList.head()` may be abbreviated by IL.`head()` and similarly for the other `IntList` methods.