Loops and Sorting

In the first part of the lab you will learn how to turn *recursive* loop-functions into *imperative* (mutating) loop using Java's `while` and `for` statements.

In the second part we will look at using direct access of list elements to implement a different kind of sorting algorithm.

9.1 Setup

For this lab download the files in *Lab9.zip*. The folder contains the following files:

- `Balloon.java`: a class representing Baloons
- `ISelect.java`: an interface for generic predicates
- `RedBallon.java` and `SmallBalloon.java`: implement the `ISelect` interface for the `Balloon`s
- `IList.java`, `MtList.java`, and `ConsList.java`: define a generic list that implements the `Traversal` interface
- `ALTrav.java`: implements a `Traversal` wrapper for the `ArrayList`s
- `Algorithms.java`: shows an implementation of several algorithms that consume `Traversal`s
- `Examples.java` file that defines examples of all data and defines all tests
- The `ForEach.java`: Implements a wrapper for three forms of Java loops (recursion, **`while`**, and **`for`**).

Create a new **Project** *Lab9* and import the files from the zip. Import the `tester.jar`. Lookover the class definitions so you understand how they work. We are particularly interested in the implementations of the `Traversal` interface (i.e., `IList` classes and `ALTrav`).

9.2 Converting Recursive Loops into `while` Loops

The goal of this part of the lab is to implement traversals over data within an `ArrayList` using Java **`while`** and **`for`** loops. After completion make sure you understand the role of each part of the loop definition: *BASE VALUE*, *CONTINUATION-PREDICATE*, *CURRENT*, *ADVANCE*, *UPDATE*, and know how to construct both the `while` loop and the `for` loop.

We will be working with the *Loop Handout* from the main Lab page, but all the templates/discussions are also in the `Algorithms.java` file (below the original implementations of `contains` and `countSuch`.

- First read the code for the `contains` and `countSuch` methods (from the `Algorithms` class). These methods have been designed in a recursive style similar to functions as we implement them in *Racket*.
- We will look at examples of `orMap` (another name for `contains`, the way we've implemented it) in the `Algorithms` class.

  Read the *template analysis* for our recursive loop version that uses the `Traversal` *iterator*. As we have done in lecture, we start by converting the recursive method into accumulator style: the accumulator remembers information about what we have seen, and is updated for recursive invocations.

  Read the *Template Analysis* portion carefully and make sure you understand the meaning of all parts: `BASE-VALUE`, `TERMINATION/CONTINUATION-PREDICATE`, `CURRENT-ELEMENT`, `UPDATE`, and `ADVANCE`.

```
// TEMPLATE ANALYSIS:
public <T> ReturnType methodName(Traversal<T> tr){
   //                        +--------------------+
   // Invoke the methodAcc: | acc <-- BASE-VALUE |
   //                        +--------------------+
   return methodNameAcc(tr, BASE-VALUE);
}

public <T> ReturnType methodNameAcc(Traversal<T> tr, ReturnType acc){

  ... tr.isEmpty()) ...              -- boolean      :: PREDICATE
  if true:
    ... acc ...                      -- ReturnType   :: BASE-VALUE
  else:
       +---------------+
    ...| tr.getFirst() | ...         -- T            :: CURRENT-ELEMENT
       +---------------+

       +------------------------+
   ... | update(T, ReturnType) |    -- ReturnType   :: UPDATE
       +------------------------+
   e.g.: update(tr.getFirst(), acc)

       +--------------+
   ... | tr.getRest() | ...         -- Traversal<T> :: ADVANCE
       +--------------+
   e.g.: methodNameAcc(tr.getRest(), update(tr.getFirst(), acc))
}
```

Based on this analysis, we can design a complete-template for solutions to many different (but similar) problems with the solution divided into three methods as follows:

```
// COMPLETED LOOP METHOD(S) TEMPLATE:
//*******************************************************
public <T> ReturnType methodName(Traversal<T> tr){
  //                        +--------------------+
  // Invoke the Acc method: | acc <-- BASE-VALUE |
  //                        +--------------------+
  return methodNameAcc(tr, BASE-VALUE);
}

public <T> ReturnType methodNameAcc(Traversal<T> tr, ReturnType acc){
     +---predicate--+
  if(| tr.isEmpty() |)
     +--------------+
    return acc;
  else
                         +----advance---+ +---update-using-current---+
    return methodNameAcc(| tr.getRest() |,|update(tr.getFirst(), acc)|);
                         +--------------+ +--------------------------+
}

<T> ReturnType update(T t, ReturnType acc){
   ... Update Accumulator ...
}
```

9.2.1 Understanding Different Loops

- Look at the first two variants of the `orMap` method: recursive definition and the variant that uses a **`while`** loop. Identify the four parts in each definition (`BASE-VALUE`, `TERMINATION/CONTINUATION-PREDICATE`, `UPDATE`, and `ADVANCE`).

  Look at the tests in the `Examples` class to see how the methods are used. After you understand how the `while` loop works compared to the recursive version, move on...

- Design two versions of a `filter` method, which produces a new `ArrayList` that contains all elements of the original list that satisfy the given `ISelect` predicate.

  Test your methods by producing all *red balloons* or all *small balloons*.

- Design and test two versions of the method `andMap` method, which determines whether *all* elements of a given `Traversal` satisfy the given `ISelect` predicate.

  Test the methods by checking whether a list contains all *red balloons* or all *small balloons*.

9.2.2 Converting **`while`** loops into **`for`** loops

Repeat the previous tasks with the two **`for`** loop variants: one that uses the **`for`** loop with a `Traversal`, and one that uses a *counted* **`for`** loop with an `ArrayList`.

A `ForEach` Abstraction

Take a look at the *ultimate* abstraction of these various accumulator-based traversals in the `ForEach` class.

Read the tests for for each variant of the `compute` method from the `ForEach` in `Examples.java`. Make sure you understand how the methods work, and how we've designed the `OrSelectUpdater`.

9.3 Selection Sort

Selection Sort is one of the more familiar sorting algorithms. It is well suited for the situations when we try to minimize the movment (i.e., copying) of data.

9.3.1 The Algorithm

Lets try to sort an `ArrayList` of size S. Assume that the first K elements in the list are sorted and are *smaller* than the rest of the elements (by a given ordering). If the first K elements are sorted (and smaller), then we have a *partitioning* of the list into two regions (sorted $[0..K-1]$, and unsorted [K..`size()`]).

We want to sort the *unsorted* (obviously) region of the `ArrayList`, and we know how to swap two items in the list. So, all we need to do is find the location of the "smallest" (for our defined comparison) and swap it with the first unsorted item. The sorted region grows by one, the unsorted region shrinks by one, and all our assumptions still hold.

If we repeat this until the last item is swapped into its correct position, we will have finished sorting the remainder of the `ArrayList`.

9.3.2 The Idea

If it helps, think about how you might sort your favorite *Basketball* (or *Cricket*, if you prefer) team by height. You grab the *shortest* player, and move him (or her) all the way to the left of the group. Now (s)he is sorted right?

Then you select the *shortest* of the remaining players and move him (or her) to the front of the remaining group... right after the previously sorted player. Now they're both sorted right?

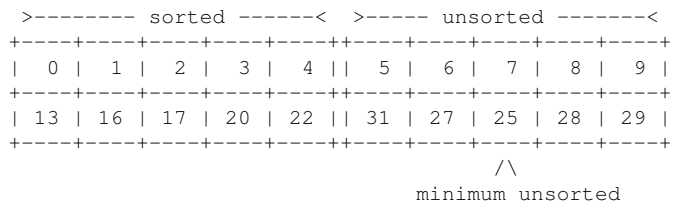
Repeat the *selection* process until we run out of "unsorted" players.

9.3.3 A Small Example

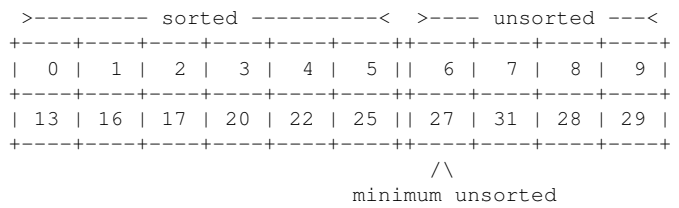
We start in the middle:

```
 >---- sorted -----<  >------- unsorted ----------<
+----+----+----+----++----+----+----+----+----+----+
|  0 |  1 |  2 |  3 ||  4 |  5 |  6 |  7 |  8 |  9 |
+----+----+----+----++----+----+----+----+----+----+
| 13 | 16 | 17 | 20 || 27 | 31 | 22 | 25 | 28 | 29 |
+----+----+----+----++----+----+----+----+----+----+
                              /\
                        minimum unsorted
```

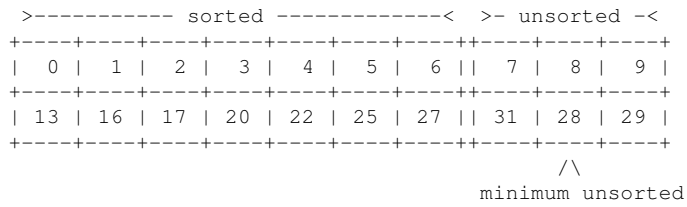
Swap elements at locations `4` and `6`:



Swap elements at locations `5` and `7`:



Swap elements at locations `6` and `6`... or optimize by skipping:



What about the starting case when none of the `ArrayList` is sorted? Well, then the sorted part has size `0`, and the unsorted part starts at the index `0`, no big deal.

1. In the `Algorithms` class design the helper method `findMinLoc` that finds the *location* of the smallest item in the unsorted part of the given `ArrayList` using a given `Comparator`.

   See `java.util.Comparator` documentation, and don't forget to **`import`** it:

   `http://download.oracle.com/javase/6/docs/api/java/util/Comparator.html`

   Note: Think carefully through the first step of the *design recipe*, to make sure you know what the method consumes and what it produces.

2. In the `Algorithms` class design the method `selectionSort` that implements the *selection sort* algorithm.

3. Design two `Comparator`s for the `Balloon`s, the `BalloonsBySize` that compares the balloons by their radius, and `BalloonsByHeight` that compares them by their distance from the top of the scene (the `y` value).

4. Test your sorting method and the helper method on lists of balloons using each of the two `Comparator`s.