

An Introduction to Sorting from a Human Perspective

Lon Levy
Oregon High School
Oregon, WI

Abstract

Computer Science teachers often debate when to teach sorting and which sorting techniques to teach. Most papers and texts approach sorting from perspectives of algorithm efficiency, programming language standards, or legacy textbook approaches. This paper approaches sorting from normal human activity and suggests a pedagogy that follows naturally from normal human activity.

1. Rationale

1.1 Sorting As a Normal Activity

Sorting is a normal human activity. Every child already knows some technique for sorting. Children sort their toys (perhaps by frequency of use), their books (perhaps by size), and their clothing (perhaps with a different value for clothing on the chair than that under the bed). As such, it is a great way to explore a set of algorithms, much of which students already know but have never explored in detail.

1.2 When to Teach Sorting

This is an ongoing debate when to teach which introductory computer science topics. Some of the code phrases are “objects-first,” “algorithms-first,” “functions-first,” or other paradigms or specific topics. Without delving too far into those debates, part of a first course in computer science should be an introduction to algorithms. If less time is to be spent on algorithms, then selecting algorithms that reflect human thought processes and expanding from there is even more important. Sorting algorithms are at that balance point. The following lessons can be expanded to include a great deal more.

2. First Lesson on Sorting

2.1 The Setup

Give each student a deck of cards that is already shuffled. I like to use double-size novelty playing cards. There are sites on the Internet that carry them, but be careful as not all are suitable for school use. However, any standard playing cards can be used. It is important that these are complete decks of cards (52 cards). I tell my students that after a recent gathering of my friends, we forgot to sort the cards before putting them away. “Oops.” Now I am asking for their help to sort the cards.

2.2 The Sequence

I wait for the question. “What order should we put them in?” or “Are aces high or low?” or some similar question. This is worthy of an “Atta-girl!” or an “Atta-boy!”

It is not enough to discuss sorting without getting into questions of ascending or descending order. More important is students analyzing a problem and thinking about it. I take a few minutes to discuss different possible ways to sort, including the question of ascending versus descending. For example, should the cards be sorted by number or by suit? What order should the suits be in? Are aces high or low? After a discussion, I request that cards be sorted by number within their suits, aces low, and the suits be in standard bridge ordering.

Why bridge ordering of the suits? It happens to be in ascending alphabetic order: clubs, diamonds, hearts, and then spades. We discuss that as an arbitrary, but convenient to remember, way to order the cards. Now students can resume (or begin) sorting.

2.3 Raw Technique

After about a minute and a half, once I am sure that all students have started sorting the cards and are no longer just thinking about the process, I stop the students. Each student is asked to take out a piece of paper and write down their technique. Individual cards should not be written down, but a general description of how they are going about what they are doing.

2.4 Analysis

So far, even with writing down what they have done, many students do not really understand their own procedures. Further, many complicate matters by mixing techniques (an excellent idea for practical human applications, but not great for understanding them). So, at this point I ask each student to read their procedure aloud, when called upon. Most of them will fall neatly into the next four categories (2.4.1–2.4.4), with most modifying the procedure as described in 2.4.5. I do not have a problem if Alexandra says that she did the same thing John said two students earlier. Good enough. Move on. There are likely to be only a few variations, albeit quite a few ways to describe the same things.

Usually students do the sorting in two stages: first they sort into suits, and then they sort the cards within each suit. Often the suits are separated into stacks on their desks at the halfway point. Some students will do unexpected things (see 2.4.6).

2.4.1 Insertion Sort

Some students will draw the cards sequentially, without regard to the value of the cards, from the unsorted pile and

put them into the correct places in the sorted pile. This is all there is to insertion sort. When coding there are some small issues of whether to start looking through the sorted pile at the front or the back, but for humans it is natural to just stick the card in its correct place.

This is the same concept whether students sort the whole deck in one pass or take two passes. That is, the first pass usually is to insert cards into the right suit (set the cards on piles of cards of the same suit) and the second pass to insert cards into the correct ordering within their respective suits.

2.4.2 Selection Sort

Some students will look through the unsorted pile to find the smallest card and place it as the first card in the sorted pile (or they find the largest card and make it the last card in the sorted pile). Then they find the next smallest card in the unsorted pile and place it next in the sorted pile; and so on. That is all there is to selection sort. When coding there are some small issues of whether to start looking through the unsorted pile at the front or the back, but it is human nature to view the entire collection to find the smallest.

This is the same concept whether students sort the whole deck in one pass or take two passes. The first pass usually is to select all of the cards of each suit in turn (find all the clubs and make a pile of just clubs and so on for the remaining suits) and the second pass to select the smallest card within that suit, ordering each suit one at a time.

2.4.3 Insertion Then Selection

Some students will draw cards sequentially from the unsorted pile, without regard to the value of the cards, and insert each card into a pile of the same suit. The second pass is selection sort within the suits.

2.4.4 Selection Then Insertion

Some students will select all the cards of each suit in turn from the unsorted pile to create a pile for each suit. The second pass is insertion sort within the suits.

2.4.5 Chunking

I prefer that students bring this topic up, but it is important enough that the teacher should raise it after going through the four most common variations (2.4.1–2.4.4) if students have not already done so. Most students will automatically grab two cards at once when they happen to already be together in the unsorted pile. They will even grab several cards at once if it is convenient. This is sometimes known as “chunking.”

While it is possible to write computer sorting algorithms that take advantage of chunking, it raises the processing overhead on small sorts and is considerably more complicated to write. For beginning students, it is easiest to approach this issue as one of the many things that humans are much better at than computers.

The advantage that computers have over humans is that they are not bored when doing simple repetitive tasks. We humans can do things faster than a computer (neurons travel at the speed of light, without as much resistance as electricity encounters in the wiring of computers so we really are faster), but we tend to get bored with repetition. Try adding a small number to each of a sequence of numbers and most humans can do so mentally faster than a computer can. However, we would rather do something more interesting, almost anything, than continuing to add two to each of a series of numbers.

2.4.6 Solitaire

Three times in the last 11 years, I have had a student play solitaire as a technique to sort the cards. Each time I have told a little fib: “That’s brilliant. I have never thought of doing that and never heard of anyone else doing that. Wow!!!” If I have any leftover small “prizes,” like pens from a software or hardware vendor, I dig one out to give it to this exceptionally gifted student.

Lest it sound like I am making too much of this, I do believe that such students are worthy of praise. Why? Because when given a straight-forward task, like sorting cards, they are thinking about how they can make it a fun chore instead of a tedious one. Such students should be recognized as role models. Most things one is trying to learn can be tedious or fun. It is very much the mindset of the learner that makes the difference. These students are seeking out fun without explicit direction to do so. That is notable and laudable.

Also, our purpose is to teach human beings. Playing a game of solitaire is extremely human. Teaching people who “think like computers” would be a terrible job.

3. Coding the Sorts

In the following sorts, I offer a pseudo-code rather than the *language du jour*. Rather than basing pseudo-code on a specific language, I have based it on algebraic, functional notation. That is, the name of a function is followed by any parameters within parenthesis. Helper functions follow the main function. I do not know of a computer language that looks quite like this, but anyone who has succeeded in Algebra II should not have too much trouble following the notation.

I have assumed a constructor function exists which concatenates a card and a deck to create a new deck, with the order of the pieces as given.

The first method or function is called with the unsorted deck and a sorted deck, the latter of which starts as an empty deck of cards. (Merge Sort, in Code Sample 3, follows the same conventions, but does not need to begin with an empty sorted deck).

Insertion Sort in Code Sample 1 and Selection Sort in Code Sample 2 are generally not very efficient, $O(n^2)$. Even so, the algorithms as I am presenting them are worse than one would usually code. The goal of this presentation is clarity, not efficiency.

```

insertion (unsortedDeck, sortedDeck)
    if unsortedDeck is empty, return sortedDeck
    otherwise
        return insertion(allButFirst(unsortedDeck),
            (stickIn (first (unsortedDeck), sortedDeck)))

allButFirst (deck)
    return deck, except for the first card

first (deck)
    return first card of deck

stickIn (card, deck)
    if card < first (deck),
        return new Deck (card, deck)
    otherwise return new Deck (first (deck),
        stickIn (card, allButFirst (deck)))

```

Code Sample 1 – Pseudo-Code for Insertion Sort

While the Insertion Sort is inefficient, there are times when it is considerably more efficient than the Selection Sort. If you wish to explore these efficiencies with your students, try using a small collection of cards and follow the algorithms when the cards are reverse sorted, randomly mixed, and nearly perfectly sorted. The results will surprise your students.

```

selection (unsortedDeck, sortedDeck)
    if unsortedDeck is empty, return sortedDeck
    otherwise return
        selection(allButSmallest (unsortedDeck),
            new Deck (sortedDeck, smallest (unsortedDeck)))

allButSmallest (deck)
    if first (deck) = smallest (deck),
        return allButFirst (deck)
    otherwise return new Deck (first (deck),
        allButSmallest (allButFirst (deck)))

smallest (deck)
    if deck has only one card, return first (deck)
    otherwise if first (deck) < second (deck),
        return smallest (allButSecond (deck))
    otherwise return smallest (allButFirst (deck))

```

```

allButFirst (deck)
    return deck, except for the first card

first (deck)
    return first card of deck

allButSecond (deck)
    return deck, except for the second card

second (deck)
    return second card of deck

```

Code Sample 2 – Pseudo-Code for Selection Sort

3. Second Lesson on Sorting, Rationale for Merge Sort

After spending a day discussing inefficient, but human, sorting techniques, I assign these to students as in-class projects. I like these to be paired projects for a number of reasons, including speed and the need for each student to be part of a pair that completes the assignment. Larger groups and individuals are problematic (based on Extreme Programming and Agile Programming models). Then it is time to discuss how a computer might be more efficient than a human being. While I favor constructivist approaches for many things, expecting students to derive an algorithm that is not based on natural human thinking is not realistic. I give students the algorithm as we act it out.

So, why Merge Sort? It is one of many sorting techniques that are not intuitive but is relatively easy to grasp once one sees and participates in the process.

If one does not want to use Merge Sort, another practical algorithm for this exercise is Radix Sort. Partition sorts, like Quick Sort are more difficult for high school students (beginning programmers) to understand.

While Merge Sort is a relatively easy non-intuitive sort, it is best to guide students through acting out merge sort and then it makes more sense. The only problem is having enough students. A full deck of cards is too much. In general, I like to begin with about as many cards as students.

4. Acting Out Merge Sort

I start the sort by asking for two volunteers and splitting the deck evenly between them. I ask them to return the decks (i.e., each half deck) to me once they are sorted. The first time we do this, I let the students sort them any way that they want (so long as the cards end up in ascending order from top of the facedown deck) and then I model the merge procedure.

Each half deck (face down) has its smallest card on top and is sorted. Select the top card from each deck. The smaller of

the two cards is set face up in a third pile. Set the other back on top of the pile it came from. (Returning it to the pile will not be necessary after a few repetitions). Repeat until one of the piles is empty. At that point the not-empty pile can be flipped over onto the third pile and the merge is complete.

Now shuffle the deck again and ask for the volunteers to each take half again, but to get helpers to sort for them. This means that (in our example with 20 students in class) I give each of two students a deck of about 10 cards each. Each of them gives half (about five cards) to more students. Each of them gives two or three card half decks to more students. Each of them gives one or two card decks to more students.

This requires 38 students in addition to the teacher who starts things off. So, we will have to carefully step through the process of dividing the cards, with almost every student taking two positions. Then go ahead and step through the process. Have students keep an eye on one another to make sure that the process is running correctly. (There are lots of places here where small mistakes can result in big problems). When done correctly, we can step through all parts of a merge sort and students understand what happened, including that the students have acted out the same part more than once just as the function or method must.

```
mergesort (deck)
  if deck has only one card, return deck
  otherwise merge (mergesort (firstHalf (deck)),
                  mergesort (secondHalf (deck)))

merge (deckOne, deckTwo)
  if deckOne is empty, return deckTwo
  otherwise if deckTwo is empty, return deckOne
  otherwise if first (deckOne) < first (deckTwo),
    return new Deck (first (deckOne),
                    merge (allButFirst (deckOne), deckTwo))
  otherwise return new Deck (first (deckTwo),
                             merge (deckOne, allButFirst (deckTwo)))

allButFirst (deck)
  return deck, except for the first card

first (deck)
  return first card of deck
```

Code Sample 3 – Pseudo-Code for Merge Sort

5. Back to the Computers

Of course, students are assigned writing the code for these; after all, programming is a significant part of computer science. I start my students with DrScheme, but this could

be done in any language. Code Sample 4 is Insertion Sort in DrScheme, including appropriate unit tests for the Testing TeachPack. Code Sample 4 also includes a contract and purpose statement, and a structural template as called for in the Design Recipe of “How to Design Programs” (see Resources). Code Sample 5 is Insertion Sort in Java, but the unit tests for all the Java code can be found in Code Sample 10. The unit test for Java assumes that all three sorts for Java are in a class called *Sorts*.

```
;; stick-it: num list-of-numbers ->
                                                list-of-numbers
;; stick the num into the correct position
;; in the list of numbers
; ... (first alist)...
; ... (rest alist)...

(define (stick-it n alist)
  (cond [(empty? alist) (cons n empty)]
        [(< n (first alist)) (cons n alist)]
        [else (cons (first alist)
                    (stick-it n (rest alist)))]))

(check-expect (stick-it 3 empty) (list 3))
(check-expect (stick-it 3 (list 1)) (list 1 3))
(check-expect (stick-it 3 (list 5)) (list 3 5))
(check-expect (stick-it 3 (list 1 2))
              (list 1 2 3))
(check-expect (stick-it 3 (list 4 5))
              (list 3 4 5))
(check-expect (stick-it 3 (list 2 4))
              (list 2 3 4))
(check-expect (stick-it 1 (list 2 3 4 5 6 7))
              (list 1 2 3 4 5 6 7))
(check-expect (stick-it 2 (list 1 3 4 5 6 7))
              (list 1 2 3 4 5 6 7))
(check-expect (stick-it 6 (list 1 2 3 4 5 7))
              (list 1 2 3 4 5 6 7))
(check-expect (stick-it 7 (list 1 2 3 4 5 6))
              (list 1 2 3 4 5 6 7))

;; i-help: list-of-numbers list-of-numbers ->
                                                list-of-numbers
;; place the numbers of the second list into
;; the first list in order
; ... (first sorted)...
; ... (rest sorted)...
; ... (first old)...
; ... (rest old)...

(define (i-help sorted old)
```

```

(cond [(empty? old) sorted]
      [else (i-help
              (stick-it (first old) sorted)
              (rest old))]])

(check-expect (i-help empty empty) empty)
(check-expect (i-help empty (list 5)) (list 5))
(check-expect (i-help empty (list 1 2 3 4 5))
              (list 1 2 3 4 5))
(check-expect (i-help empty (list 5 4 3 2 1))
              (list 1 2 3 4 5))
(check-expect (i-help empty (list 5 1 3 4 2))
              (list 1 2 3 4 5))
(check-expect (i-help (list 5) (list 1 2 3 4))
              (list 1 2 3 4 5))
(check-expect (i-help (list 3) (list 5 4 2 1))
              (list 1 2 3 4 5))
(check-expect (i-help (list 2 3 4) (list 1 5))
              (list 1 2 3 4 5))
(check-expect (i-help (list 1 3 5) (list 4 2))
              (list 1 2 3 4 5))
(check-expect (i-help (list 1 2 3 4 5) empty)
              (list 1 2 3 4 5))

;; insert-sort : list-of-numbers ->
;;                                     list-of-numbers
;; use the insertion sort to sort a list of
;; numbers into ascending order.
; ... (first alist) ...
; ... (rest alist) ...

(define (insert-sort alist)
  (i-help empty alist))

(check-expect (insert-sort empty) empty)
(check-expect (insert-sort (list 5)) (list 5))
(check-expect (insert-sort (list 1 2 3 4 5))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 5 4 3 2 1))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 5 1 3 4 2))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 1 2 3 5 4))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 1 2 5 3 4))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 1 5 2 3 4))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 5 1 2 3 4))
              (list 1 2 3 4 5))

```

```

(list 1 2 3 4 5))
(check-expect (insert-sort (list 4 1 3 5 2))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 2 1 3 4 5))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 2 3 1 4 5))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 2 3 4 1 5))
              (list 1 2 3 4 5))
(check-expect (insert-sort (list 2 3 4 5 1))
              (list 1 2 3 4 5))

```

Code Sample 4 – Insertion Sort in DrScheme

```

public static LinkedList<Integer>
    insertion(LinkedList<Integer> alist)
{
    LinkedList<Integer> empty =
        new LinkedList<Integer>();
    return iHelper(empty, alist);
}

private static LinkedList<Integer>
    iHelper(LinkedList<Integer> sorted,
            LinkedList<Integer> unsorted)
{
    if (unsorted.size() == 0)
    {
        return sorted;
    }
    Integer first = unsorted.remove(0);
    return iHelper(stickIt(first, sorted),
                  unsorted);
}

private static LinkedList<Integer>
    stickIt(Integer num, LinkedList<Integer> alist)
{
    ListIterator<Integer> it =
        alist.listIterator();
    while (it.hasNext())
    {
        if (it.next().intValue() > num.intValue())
        {
            it.previous();
            it.add(num);
            return alist;
        }
    }
    it.add(num);
    return alist;
}

```

```

}

Code Sample 5 – Insertion Sort in Java

;; smallest : list-of-numbers -> num
;; find the smallest of a non-empty list of
;; numbers
; ... (first alist) ...
; ... (rest alist) ...

(define (smallest alist)
  (cond [(empty? (rest alist)) (first alist)]
        [(< (first alist) (first (rest alist)))
         (smallest (cons (first alist)
                         (rest (rest alist))))]
        [else (smallest (rest alist))]))

(check-expect (smallest (list 1)) 1)
(check-expect (smallest (list 1 2)) 1)
(check-expect (smallest (list 2 1)) 1)
(check-expect (smallest (list 3 1 2)) 1)
(check-expect (smallest (list 5 7 1 9 2)) 1)

;; all-but-smallest : list-of-numbers ->
;;                    list-of-numbers
;; returns a list with all but the smallest of a
;; non-empty list
; ... (first alist) ...
; ... (rest alist) ...

(define (all-but-smallest alist)
  (cond [(= (smallest alist) (first alist))
         (rest alist)]
        [else (cons (first alist)
                    (all-but-smallest (rest alist)))]))

(check-expect (all-but-smallest (list 1)) empty)
(check-expect (all-but-smallest (list 1 2 3 4 5))
              (list 2 3 4 5))
(check-expect (all-but-smallest (list 5 4 3 2 1))
              (list 5 4 3 2))
(check-expect (all-but-smallest (list 5 2 4 1 3))
              (list 5 2 4 3))
(check-expect (all-but-smallest (list 2 1 3 4 5))
              (list 2 3 4 5))

;; s-sort : list-of-numbers list-of-numbers ->
;;                    list-of-numbers
;; does the actual selection sort, with two
;; helpers: smallest and all-but-smallest

; ... (first unsorted) ...
; ... (rest unsorted) ...
; ... (first sorted) ...
; ... (rest sorted) ...

(define (s-sort unsorted sorted)
  (cond [(empty? unsorted) sorted]
        [else (s-sort (all-but-smallest unsorted)
                      (cons (smallest unsorted)
                          sorted))]))

(check-expect (s-sort empty empty) empty)
(check-expect (s-sort (list 17) empty) (list 17))
(check-expect (s-sort (list 1 2) empty)
              (list 2 1))
(check-expect (s-sort (list 2) (list 1))
              (list 2 1))
(check-expect (s-sort (list 2 1) empty)
              (list 2 1))
(check-expect (s-sort (list 2) (list 1))
              (list 2 1))
(check-expect (s-sort (list 1 2 3 4 5) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 2 1 3 4 5) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 3 1 2 4 5) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 4 1 2 3 5) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 5 1 2 3 4) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 5 4 3 2 1) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 4 2 1 3 5) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 5 3 1 2 4) empty)
              (list 5 4 3 2 1))
(check-expect (s-sort (list 5 3 4) (list 2 1))
              (list 5 4 3 2 1))
(check-expect (s-sort (list 4 5) (list 3 2 1))
              (list 5 4 3 2 1))

;; turn-around : list list -> list
;; reverse the first list into the second list,
;; which should begin as empty
; ... (first alist) ...
; ... (rest alist) ...
; ... (first fin) ...
; ... (rest fin) ...

```

```

(define (turn-around alist fin)
  (cond [(empty? alist) fin]
        [else (turn-around (rest alist)
                            (cons (first alist) fin))]))

(check-expect (turn-around empty empty) empty)
(check-expect (turn-around (list 1) empty)
              (list 1))
(check-expect (turn-around (list 3 5) empty)
              (list 5 3))
(check-expect (turn-around (list 1 3 5 7 2) empty)
              (list 2 7 5 3 1))

;; select-sort : list-of-numbers ->
;;                                     list-of-numbers
;; sort numbers into ascending order,
;; using selection sort
; ... (first alist) ...
; ... (rest alist) ...

(define (select-sort alist)
  (turn-around (s-sort alist empty) empty))

(check-expect (select-sort empty) empty)
(check-expect (select-sort (list 17)) (list 17))
(check-expect (select-sort (list 1 2)) (list 1 2))
(check-expect (select-sort (list 2 1)) (list 1 2))
(check-expect (select-sort (list 1 2 3 4 5))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 2 1 3 4 5))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 3 1 2 4 5))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 4 1 2 3 5))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 5 1 2 3 4))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 5 4 3 2 1))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 4 2 1 3 5))
              (list 1 2 3 4 5))
(check-expect (select-sort (list 5 3 1 2 4))
              (list 1 2 3 4 5))

```

Code Sample 6 – Selection Sort in DrScheme

```

public static LinkedList<Integer>
    selection(LinkedList<Integer> alist)
{
    LinkedList<Integer> empty =

```

```

        new LinkedList<Integer>();
    return sHelper(empty, alist);
}

private static LinkedList<Integer>
    sHelper(LinkedList<Integer> sorted,
           LinkedList<Integer> unsorted)
{
    if (unsorted.size() == 0)
    {
        return sorted;
    }
    Integer smallest = smallest(unsorted);
    sorted.add(smallest);
    unsorted = allBut(smallest, unsorted);
    return sHelper(sorted, unsorted);
}

private static Integer
    smallest(LinkedList<Integer> alist)
{
    Integer smallest =
        new Integer(alist.get(0).intValue());
    for (Integer num : alist)
    {
        If (num.intValue() < smallest.intValue())
        {
            smallest =
                new Integer(num.intValue());
        }
    }
    return smallest;
}

private static LinkedList<Integer>
    allBut(Integer num,
           LinkedList<Integer> alist)
{
    ListIterator<Integer> it =
        alist.listIterator();
    while (it.hasNext())
    {
        if (num.equals(it.next()))
        {
            it.remove();
            return alist;
        }
    }
    // this point should never be reached

```

```

    return alist;
}

```

Code Sample 7 – Selection Sort in Java

```

;; merge : list-of-numbers list-of-numbers ->
           list-of-numbers
;; merges two sorted lists of numbers together
;; into one sorted list
; ... (first listA) ...
; ... (rest listA) ...
; ... (first listB) ...
; ... (rest listB) ...

(define (merge listA listB)
  (cond [(empty? listA) listB]
        [(empty? listB) listA]
        [(< (first listA) (first listB))
         (cons (first listA)
                (merge (rest listA) listB))]
        [else (cons (first listB)
                     (merge listA (rest listB)))]))

(check-expect (merge empty empty) empty)
(check-expect (merge empty (list 1 2 3))
              (list 1 2 3))
(check-expect (merge (list 1 2 3) empty)
              (list 1 2 3))
(check-expect (merge (list 1 2 3) (list 4 5 6))
              (list 1 2 3 4 5 6))
(check-expect (merge (list 4 5 6) (list 1 2 3))
              (list 1 2 3 4 5 6))
(check-expect (merge (list 1 5 6) (list 2 3 4))
              (list 1 2 3 4 5 6))
(check-expect (merge (list 2 3 4) (list 1 5 6))
              (list 1 2 3 4 5 6))

;; firsthalf : list-of-numbers n ->
              list-of-numbers
;; returns the first half of the given list ...
;; more precisely, a list of the first n elements
;; of the given list
; ... (first alist) ...
; ... (rest alist) ...

(define (firsthalf alist n)
  (cond [(< n 1) empty]
        [else (cons (first alist)
                    (firsthalf (rest alist) (- n 1)))]))

(check-expect (firsthalf (list 1 2) 1) (list 1))

```

```

(check-expect (firsthalf (list 1 2 3 4 5 6 7) 3)
              (list 1 2 3))
(check-expect (firsthalf (list 1 2 3 4 5 6 7 8) 4)
              (list 1 2 3 4))

;; secondhalf : list-of-numbers n ->
              list-of-numbers
;; returns the second half of the given list ...
;; more precisely, all but the first n elements
;; of the given list
; ... (first alist) ...
; ... (rest alist) ...

(define (secondhalf alist n)
  (cond [(< n 1) alist]
        [else (secondhalf (rest alist) (- n 1))]))

(check-expect (secondhalf (list 1 2) 1) (list 2))
(check-expect (secondhalf (list 1 2 3 4 5 6 7) 3)
              (list 4 5 6 7))
(check-expect (secondhalf (list 1 2 3 4 5 6 7 8)
                      4)
              (list 5 6 7 8))

;; mergesort : list-of-numbers -> list-of-numbers
;; use mergesort to sort the numbers in the
;; given list
; ... (first alist) ...
; ... (rest alist) ...

(define (mergesort alist)
  (cond [(< (length alist) 2) alist]
        [else (merge
                (mergesort
                 (firsthalf alist
                    (floor (/ (length alist) 2))))
                (mergesort
                 (secondhalf alist
                    (floor (/ (length alist) 2)))))]))

(check-expect (mergesort empty) empty)
(check-expect (mergesort (list 17)) (list 17))
(check-expect (mergesort (list 1 2)) (list 1 2))
(check-expect (mergesort (list 2 1)) (list 1 2))
(check-expect (mergesort (list 1 2 3 4 5))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 2 1 3 4 5))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 3 1 2 4 5))
              (list 1 2 3 4 5))

```

```

        (list 1 2 3 4 5))
(check-expect (mergesort (list 4 1 2 3 5))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 5 1 2 3 4))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 5 4 3 2 1))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 4 2 1 3 5))
              (list 1 2 3 4 5))
(check-expect (mergesort (list 5 3 1 2 4))
              (list 1 2 3 4 5))

```

Code Sample 8 – Merge Sort in DrScheme

```

public static LinkedList<Integer>
    mergeSort(LinkedList<Integer> alist)
{
    if (alist.size() < 2)
    {
        return alist;
    }
    return merge(mergeSort(firstHalf(alist)),
                mergeSort(secondHalf(alist)));
}

private static LinkedList<Integer>
    merge(LinkedList<Integer> a,
          LinkedList<Integer> b)
{
    if (a.size() == 0)
    {
        return b;
    }
    if (b.size() == 0)
    {
        return a;
    }
    LinkedList<Integer> result =
        new LinkedList<Integer>();
    while (a.size() > 0 && b.size() > 0)
    {
        if (a.get(0).intValue() <
            b.get(0).intValue())
        {
            result.add(a.remove(0));
        }
        else
        {
            result.add(b.remove(0));
        }
    }
}

```

```

    if (a.size() == 0)
    {
        result.addAll(b);
    }
    else
    {
        result.addAll(a);
    }
    return result;
}

private static LinkedList<Integer>
    firstHalf(LinkedList<Integer> alist)
{
    LinkedList<Integer> result =
        new LinkedList<Integer>();
    for (int i = 0; i < alist.size() / 2; i++)
    {
        result.add(alist.get(i));
    }
    return result;
}

private static LinkedList<Integer>
    secondHalf(LinkedList<Integer> alist)
{
    LinkedList<Integer> result =
        new LinkedList<Integer>();
    for (int i = alist.size() / 2;
         i < alist.size(); i++)
    {
        result.add(alist.get(i));
    }
    return result;
}

```

Code Sample 9 – Merge Sort in Java

```

public class TestSorts extends
    junit.framework.TestCase
{
    private LinkedList<Integer> empty, a, b0, b1,
        c0, c1, c2, c3, c4, c5, c6, c7,
        s0, s1, s2, s5;

    public TestSorts()
    {
    }

    protected void setUp()

```

```

{
    this.empty = new LinkedList<Integer>();
    this.a = new LinkedList<Integer>();
    this.b0 = new LinkedList<Integer>();
    this.b1 = new LinkedList<Integer>();
    this.c0 = new LinkedList<Integer>();
    this.c1 = new LinkedList<Integer>();
    this.c2 = new LinkedList<Integer>();
    this.c3 = new LinkedList<Integer>();
    this.c4 = new LinkedList<Integer>();
    this.c5 = new LinkedList<Integer>();
    this.c6 = new LinkedList<Integer>();
    this.c7 = new LinkedList<Integer>();
    this.s0 = new LinkedList<Integer>();
    this.s1 = new LinkedList<Integer>();
    this.s2 = new LinkedList<Integer>();
    this.s5 = new LinkedList<Integer>();
    this.a.add(new Integer(17));
    this.b0.add(new Integer(1));
    this.b0.add(new Integer(2));
    this.b1.add(new Integer(2));
    this.b1.add(new Integer(1));
    this.c0.add(new Integer(1));
    this.c0.add(new Integer(2));
    this.c0.add(new Integer(3));
    this.c0.add(new Integer(4));
    this.c0.add(new Integer(5));
    this.c1.add(new Integer(2));
    this.c1.add(new Integer(1));
    this.c1.add(new Integer(3));
    this.c1.add(new Integer(4));
    this.c1.add(new Integer(5));
    this.c2.add(new Integer(3));
    this.c2.add(new Integer(1));
    this.c2.add(new Integer(2));
    this.c2.add(new Integer(4));
    this.c2.add(new Integer(5));
    this.c3.add(new Integer(4));
    this.c3.add(new Integer(1));
    this.c3.add(new Integer(2));
    this.c3.add(new Integer(3));
    this.c3.add(new Integer(5));
    this.c4.add(new Integer(5));
    this.c4.add(new Integer(1));
    this.c4.add(new Integer(2));
    this.c4.add(new Integer(3));
    this.c4.add(new Integer(4));
    this.c5.add(new Integer(5));
    this.c5.add(new Integer(4));

    this.c5.add(new Integer(3));
    this.c5.add(new Integer(2));
    this.c5.add(new Integer(1));
    this.c6.add(new Integer(4));
    this.c6.add(new Integer(2));
    this.c6.add(new Integer(1));
    this.c6.add(new Integer(3));
    this.c6.add(new Integer(5));
    this.c7.add(new Integer(5));
    this.c7.add(new Integer(3));
    this.c7.add(new Integer(1));
    this.c7.add(new Integer(2));
    this.c7.add(new Integer(4));
    this.s1.add(new Integer(17));
    this.s2.add(new Integer(1));
    this.s2.add(new Integer(2));
    this.s5.add(new Integer(1));
    this.s5.add(new Integer(2));
    this.s5.add(new Integer(3));
    this.s5.add(new Integer(4));
    this.s5.add(new Integer(5));
}

private boolean
    listEquals(LinkedList<Integer> a,
               LinkedList<Integer> b)
{
    if (a.size() != b.size())
    {
        return false;
    }
    ListIterator ita = a.listIterator();
    ListIterator itb = b.listIterator();
    while (ita.hasNext())
    {
        if (ita.next() != itb.next())
        {
            return false;
        }
    }
    return true;
}

public void testInsertion()
{
    LinkedList<Integer> results;
    results = Sorts.insertion(this.empty);
    assertEquals(this.s0, results);
    results = Sorts.insertion(this.a);
}

```

```

assertEquals(this.s1, results);
results = Sorts.insertion(this.b0);
assertEquals(this.s2, results);
results = Sorts.insertion(this.b1);
assertEquals(this.s2, results);
results = Sorts.insertion(this.c0);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c1);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c2);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c3);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c4);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c5);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c6);
assertEquals(this.s5, results);
results = Sorts.insertion(this.c7);
assertEquals(this.s5, results);
}

public void testSelection()
{
    LinkedList<Integer> results;
    results = Sorts.selection(this.empty);
    assertEquals(this.s0, results);
    results = Sorts.selection(this.a);
    assertEquals(this.s1, results);
    results = Sorts.selection(this.b0);
    assertEquals(this.s2, results);
    results = Sorts.selection(this.b1);
    assertEquals(this.s2, results);
    results = Sorts.selection(this.c0);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c1);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c2);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c3);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c4);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c5);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c6);
    assertEquals(this.s5, results);
    results = Sorts.selection(this.c7);
    assertEquals(this.s5, results);
}

    assertEquals(this.s5, results);
}

public void testMerge ()
{
    LinkedList<Integer> results;
    results = Sorts.mergeSort(this.empty);
    assertEquals(this.s0, results);
    results = Sorts.mergeSort(this.a);
    assertEquals(this.s1, results);
    results = Sorts.mergeSort(this.b0);
    assertEquals(this.s2, results);
    results = Sorts.mergeSort(this.b1);
    assertEquals(this.s2, results);
    results = Sorts.mergeSort(this.c0);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c1);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c2);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c3);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c4);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c5);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c6);
    assertEquals(this.s5, results);
    results = Sorts.mergeSort(this.c7);
    assertEquals(this.s5, results);
}

protected void tearDown()
{
}
}

```

Code Sample 10 – JUnit Tests for Java Sorts

6. Summary

For many computer geeks, sorting is already fun. Delving into Big-Oh efficiencies and compiler optimization is like candy. Beginning students are not ready for all of that. Instead, they need an approach that is based on what they already know and how they already act when faced with a similar task.

The approach outlined above does exactly that. It is using students' normal human understanding as the base from which to explore the details of what they already know and expand to new ways of thinking for them. As a small bonus, since we are playing with cards through the initial activities,

students expect to be having fun; making it much more likely that they will.

Acknowledgements

The idea for using playing cards to step through some sorting algorithms I first saw at an AP Computer Science Workshop at the Maritime Institute of Technology and Graduate Studies in spring 2003.

DrScheme code is based on How to Design Programs (<http://www.htdp.org/>) Lots more information about functional programming, problem solving and more via that site.

Many more ideas here from too many people and too many workshops where I did not take careful enough notes; and I still have more to learn!

Resources

<http://www.nist.gov/dads/> is the National Institute of Standards and Technology Web page, "Dictionary of Algorithms and Data Structures." It is the gold standard of such resources.

<http://www.extremeprogramming.org/rules/pair.html> is a starting point for a great deal of information about Pair Programming.

<http://www.youtube.com/watch?v=A6kdFdJp4jY> is a 10-minute video that will help some students begin with pair every 10 minutes. Some less mature students need the programming. I do modify things a little. At the start of

required times to swap early in the semester. Later in the semester I allow students to choose their own swap times, encouraging them to take some time in each role.

<http://www.drscheme.org/> includes a complete IDE and set of learning languages based on Scheme. A free companion text is available at <http://www.htdp.org/>. There is also included a set of learning languages for Java called ProfessorJ.

<http://java.sun.com/> is the home Web site for Sun Microsystems and the free Java programming language.

<http://www.bluej.org/> is a free IDE for beginning students in Java.

Author Information

Lon Levy, M.S. – C.S.Ed.

Computer Science Teacher

456 North Perry Parkway

Oregon High School

Oregon, WI 53575

608-835-1316

LXL@oregon.k12.wi.us

cs@levytree.net

Lon teaches an Introduction to Computer Science and AP Computer Science "AB": Data Structures at OHS. He has been playing and working with computers since the mid-1970s and teaching in Wisconsin since 1997. He is a regular attendee of SIGCSE since 2004 and has served as an AP Computer Science Reader since 2006. Lon is eager to continue learning how to better teach problem solving.