

A Two Day Lab-Based Introduction to CS

Contents

- Introduction [p 2]
- Syllabus [p 4]
- Day One [p 6]
 - Exercise 1: A Sandwich Algorithm [p 7]
 - Exercise 2: A Revised Sandwich Algorithm [p 8]
 - Notes: The Parts of an Algorithm [p 9]
 - Exercise 3: Sorting Books [p 11]
 - Exercise 4: Sorting Numbers [p 12]
 - Exercise 5: An Introduction to Scheme [p 13]
- Overnight [p 19]
- Day Two [p 23]
 - Exercise 6: Comparing Algorithms [p 24]
 - Exercise 7: Studying Selection Sort [p 25]
 - Exercise 8: Formalizing Requirements [p 28]
 - Exercise 9: Comparing Sorting Algorithms [p 29]
 - Exercise 10: Finding the Largest Value [p 30]

Introduction

- About Computer Science [p 2]
- About This Experience [p 2]

About Computer Science

The name “Computer Science” is a somewhat awkward one for the discipline, since it’s arguable that we don’t focus on the scientific method (as some say, “If you have to put science in the name, it’s not science”) and we don’t necessarily study computers. We’re also too often associated with the more practical skill of computer programming, which has a similar relationship to the discipline that reading has to the English major: it’s a necessary skill, but it’s not all there is. However, we’re stuck with the name and the misconception, so let’s just consider what computer scientists do.

Although computer scientists differ somewhat in the ways in which they define the discipline, most would agree that

Computer science is the study of algorithms and data structures.

By “algorithms”, we mean sets of instructions that can be used to solve problems. Some problems are mathematical. For example, you might write an algorithm to find the square root of a real number. Other problems deal with textual information. For example, you might write an algorithm that tells how to find a name in the phone book. You can write algorithms for a wide variety of problems.

By “data structures”, we mean mechanisms for organizing information. For example, we organize some information in lists and other information in tables.

By “study”, we mean specify, design, describe, evaluate mathematically, evaluate experimentally, implement in software, implement in hardware, prove properties, consider applications and implications, and much, much more.

In our studies, we rely on the tools and techniques from a number of other disciplines. From *mathematics*, we take proof techniques, formal language for describing problems and solutions, and even core ideas. From *science*, we take experimental techniques. From *engineering*, we take techniques for designing and constructing things. From *psychology and the social sciences*, we take techniques for understanding the relationship of our work to human endeavors.

About This Experience

The primary goal of this two-day laboratory experience is to give you a sense of what it means to “do” computer science. We’ll carefully specify some problems, design some algorithms to solve them, implement them in the programming language Scheme, and evaluate them experimentally.

Today we’ll focus on some background ideas. We’ll start by writing an algorithm for an everyday problem. We’ll continue by looking at a traditional and well-studied problem in computer science, that of putting a list of things in order. We’ll spend a little time working with the basics of Scheme. We’ll conclude by reflecting on what we’ve done today.

Tonight you'll have the opportunity to read my somewhat more formal descriptions of some common sorting algorithms in preparation for tomorrow's lab. You'll also reflect on what you've done today.

Tomorrow, we'll consider these sorting algorithms and some related algorithms in more depth. We'll be doing a good deal of experimental analysis and also look at some related design and theory principles.

You can find more information on my plans for the two days in the syllabus.

Approximate Syllabus

First Day

Introduction [10 minutes]

An introduction to computer science and to the labs, given in Sam's traditional (lecture + discussion + recitation) format.

Exercises 1: PBJ Algorithm and 2: PBJ Algorithm, Revised [30 minutes]

Students write an algorithm (instructions) for making a peanut butter and jelly sandwich. Sam tries to follow the instructions as a computer might. Students try again. We reflect on common algorithm components.

Exercises 3: Designing A Sorting Algorithm and 4: Designing a Second Sorting Algorithm [30 minutes]

Students write algorithms to sort a collection of books/CDS/etc.

Exercise 5: Starting Scheme [40 minutes]

Students play with some Scheme basics.

Reflection [10 minutes]

Students reflect on what they've learned so far.

Overnight

Review: Today's material

Scan through the handouts that correspond to today's discussion and your notes from the discussion.

Read: Some Standard Sorting Algorithms

Second Day

Review [10 minutes]

Students ask Sam questions. Sam asks students questions.

Exercise 6: Comparing Algorithms [10 minutes]

We'll approach this exercise as a discussion question.

Exercise 7: Selection Sort [30 minutes]

Students run and test an implementation of selection sort. Students also experimentally determine a formula for the running time of selection sort.

Exercise 8: Formalizing Requirements [20 minutes]

Using discussion format, students work to come up with a formal definition for what it means to remove a particular element from a list, as in selection sort.

Exercise 9: Comparing Sorting Algorithms [40 minutes]

Students experimentally compare three sorting algorithms.

Reflection [10 minutes]

Students reflect on what they've learned over the two days.

If There is Extra Time

Exercise 10: Finding the Largest Value

Working from a Scheme procedure to find the smallest value in a list, students build a Scheme procedure to find the largest value in a list. They then generalize to find the best value in a list. We'll do this exercise as a group.

Day One

- Exercise 1: A Sandwich Algorithm [p 7]
- Exercise 2: A Revised Sandwich Algorithm [p 8]
- Notes: The Parts of an Algorithm [p 9]
- Exercise 3: Sorting Books [p 11]
- Exercise 4: Sorting Numbers [p 12]
- Exercise 5: An Introduction to Scheme [p 13]

Exercise: A Sandwich Algorithm

Write a set of instructions for making a peanut butter and jelly sandwich. Assume that the person you're writing instructions for is fairly clueless (e.g., an "ivory tower" professor, a computer programmer, or whatever group to whom you assign little common sense).

Exercise: A Revised Sandwich Algorithm

Given your experience of dealing with a simulated clueless person, write a set of instructions for making N peanut butter and jelly sandwiches, where N is a non-negative integer. That is, the person should be able to follow these instructions to make 1, 10, 5, or even 0 sandwiches. You can assume that the person has enough supplies to make N sandwiches and that the person can count.

The Parts of an Algorithm

As you may have noted, there are some common aspects to algorithms. That is, there are techniques that we use in many of the algorithms we write. It is worthwhile to think about these algorithm parts because we can rely on them when we write new algorithms. Common parts of algorithms include

- Variables: Named Values [p 9]
- Parameters: Named Inputs [p 9]
- Conditionals: Handling Different Conditions [p 9]
- Repetition [p 9]
- Subroutines: Named Helper Algorithms [p 10]
- Recursion: Helping Yourself [p 10]

Variables: Named Values

As we write algorithms, we like to name things. Sometimes we use long names, such as “the piece of bread in your left hand”. Sometimes, we use shorter names, such as “bread-left”. As we start to write more formal algorithms, we may explicitly say that we’re using names.

Parameters: Named Inputs

Many algorithms work on data that are presented to the algorithm. For example, our PBJ algorithm should work no matter what kind of bread or peanut butter we give to the algorithm. That algorithm also requires that we give it bread and peanut butter. Similarly, an algorithm to find a name in the phone book might take the name and the phone book as input values. We call such input values “*parameters*”.

Conditionals: Handling Different Conditions

At times, our algorithms have to account for different conditions, doing different things depending on those conditions. In our PBJ algorithm, we might check whether the jar of peanut butter is open or what kind of lid is on the jelly jar. We call such operations conditionals. They typically take either the form

if some condition holds then do something

of the more complex form

if some condition holds then do something otherwise do something else

At times, we need to decide between more than two possibilities.

Repetition

At times, our algorithms require us to do something again and again. In our PBJ algorithm, we may have had to turn the twisty-tie again and again until it was untwisted. In our many-sandwiches algorithm, we made one sandwich again and again. We call this technique “*repetition*”. Repetition takes many forms. We might do work until we’ve reached a desired state.

repeat *some action* until *some condition holds*

We might continue work as long as we're in some state.

repeat *some action* while *some condition holds*

We might repeat an action a fixed number of times.

repeat *some action* *N* times

You can probably think of other forms of repetition.

Subroutines: Named Helper Algorithms

Many algorithms require common actions for their operation. For example, to make *N* sandwiches, you benefit from knowing how to make one sandwich. To make a peanut butter and jelly sandwich, it helps to know how to spread something on bread. We can write additional algorithms for these common actions and use them as part of our broader algorithm. We can also use them in other algorithms. We call these helper algorithms “*subroutines*”.

Recursion: Helping Yourself

As strange as it may seem, we sometimes find it useful to define an algorithm in terms of itself. In particular, we can have an algorithm deal with a large or complex input by using itself as a helper with a smaller or less complex input.

Consider the following example

```
To make N peanut butter and jelly sandwiches
  If N is 0 then           conditional
    Celebrate, you're done!
  Otherwise
    Make one PBJ sandwich  subroutine
    Make N-1 PBJ sandwiches recursion
```

We call this technique *recursion*. Mastering recursion is one of the key steps on your path to becoming a computer scientist.

A helpful way to think about recursion: Every time you use a subroutine you hand the input to that algorithm and the instructions for the algorithm to a friend. That friend executes the algorithm and hands the result back to you. It shouldn't matter whether the algorithm you give your friend is a different one than you're using or the same one.

Exercise: Designing a Sorting Algorithm

Write an algorithm that tells someone how to put a collection of books in alphabetical order by author. You may find it useful to design additional algorithms to solve parts of the problem.

Exercise: Designing Another Sorting Algorithm

Write an algorithm that tells someone how to put a list of numbers in order from smallest to largest. Your algorithm should use a different strategy than your “sort books” algorithm. Once again, you may find it useful to design additional algorithms to solve parts of the problem.

Exercise: An Introduction to Scheme

- DrScheme [p 13]
 - DrScheme's Interactions Pane [p 13]
 - DrScheme's Definitions Pane [p 14]
- Lists in Scheme [p 14]
 - Constructing Lists [p 14]
 - Extracting Values [p 16]
 - Common list procedures [p 16]
- Defining Procedures [p 17]
- Conditionals [p 17]

Please don't be intimidated! Although this lab contains many details which may seem overwhelming at first, these mechanics will become familiar rather quickly.

DrScheme

Short Version:

- We use a programming language called Scheme and a programming environment called DrScheme.

Many of the fundamental ideas of computer science are best learned by reading, writing, and executing small computer programs that illustrate them. One of our most important tools, therefore, is a computer program designed specifically to make it easier to read, write, and execute other computer programs. In our introductory courses, we will often use a *programming environment* named DrScheme.

DrScheme's Interactions Pane

Short Version:

- The lower text area is called the *interactions pane*.
- You type scheme commands there and DrScheme responds.
- Try typing `(sqrt 137641)`

Now you're ready to look at the parts of DrScheme.

In the *interactions pane* -- the lower of the two large text areas -- DrScheme displays a one-line greeting, a reminder of which dialect of Scheme it expects to see, and a *prompt* (in this case, a greater-than sign), indicating that DrScheme is ready to deal with any command that you type in.

To enter a Scheme program, move the mouse pointer to the right of the prompt, click the left mouse button, and type in the program. (If you prefer, you can *select* the program from another window by moving the mouse pointer to the beginning of the program, pressing and holding the left mouse button, dragging the mouse pointer to the end of the program, and releasing the left mouse button. The background color against which the text that you have selected changes during this process, so that you can see the boundaries of the selection clearly. You can then *paste* the selected text into the interactions pane by moving the mouse pointer to the right of the prompt and clicking the *middle* mouse button.)

To get DrScheme to execute your program, press the **Enter** key after the right parenthesis. At this point, DrScheme examines your program, translates it into a sequence of instructions to the computer's *central processor* (the electronic circuit that directs the movement and transformation of data inside the computer), executes it, and prints out the result of its computation. Because this particular program is extremely simple, the result is printed immediately.

You may notice that DrScheme prints out another prompt after executing your program. This is because DrScheme cannot be sure that it has seen all the steps in the program. A program written in Scheme has a particularly simple structure: it is a sequence of definitions and commands -- any number of them, in any order. DrScheme reacts to each definition that you type into the interactions pane by memorizing it and to each command by carrying out the command. (The expression `(sqrt 137641)` is a command -- "Compute the square root of 137641!") Because a program might contain several commands rather than just one, DrScheme has to be prepared to receive another after carrying out the first.

DrScheme's Definitions Pane

Short Version:

- The interactions pane is temporary. The top pane, called the *definitions pane* is more permanent.
- When you click **Execute**, the interactions pane is erased and the instructions in the definitions pane are executed.

The upper text area in the DrScheme window, which is called the *definitions pane*, is used when you want to prepare a program "off-line," that is, without immediately executing each step. Instead of processing what you type line by line, DrScheme waits for you to click on the button labelled **Execute** (the second button from the right, in the row just below the menu bar) before starting to execute the program in the definitions pane. If you never click on that button, fine -- your program is never executed.

As its name implies, the definitions pane usually contains definitions rather than commands, although either kind of expression can be written in either pane. The difference is simply that we generally want an immediate response to a command, whereas definitions are usually processed in bulk.

*Warning: When you click on the **Execute** button, the contents of the interactions pane are erased.* The idea is that executing the program in the definitions pane may invalidate the results of previous interactions. Erasing the results that may now be inconsistent with the new definitions ensures that all visible interactions use the same vocabulary. This is actually a helpful feature of DrScheme, but it can take you by surprise the first time you see it happen. Just make sure that you have everything you need from the interactions pane before clicking on **Execute**.

Lists in Scheme

Constructing Lists

In addition to "unstructured" data types such as symbols and numbers, Scheme supports *lists*, which are structures that contain other values as elements. There is one list, the *empty list*, that contains no elements at all. Any other list is constructed by attaching some value, called the *car* of the new list, to a previously constructed list, which is called the *cdr* of the new list.

Scheme's name for the empty list is a pair of parentheses with nothing between them: `()`. When we refer to the empty list in a Scheme program, we have to put an apostrophe before the left parenthesis, so that Scheme won't mistake the parentheses for a procedure call:

```
> ()  
( )
```

Since this conventional name for the empty list is not very readable, our implementation of Scheme also provides a built-in name, `null`, for the empty list. We follow this usage and recommend it.

```
> null  
( )
```

The simplest way to build a list is with the `list` procedure. You tell Scheme to build a list by writing an open parenthesis, the procedure name `list`, the values you want in the list, and a close parenthesis.

```
> (list 1 2 3)  
(1 2 3)  
> (list 3 1 2 5 6)  
(3 1 2 5 6)
```

You can name a list you've created with the `define` procedure.

```
> (define lst (list 1 2 3 4 5))  
> lst  
(1 2 3 4 5)
```

You can also add an element to the front of a list with `cons`. It takes two arguments and returns a list that is just like the second argument, except that the first argument has been added at the beginning, as a new first element. Once again, this change does not affect the original list.

```
> (define lst (list 1 2 3 4 5))  
> lst  
(1 2 3 4 5)  
> (cons 6 lst)  
(6 1 2 3 4 5)  
> (define newlst (cons 0 lst))  
> newlst  
(0 1 2 3 4 5)  
> lst  
(1 2 3 4 5)
```

Task: Try this example!

You can even use `cons` to build up a list from nothing (or at least from `null`).

```

> (define singleton (cons 5 null))
> singleton
(5)
> (define doubleton (cons 8 singleton))
> doubleton
(8 5)
> (define tripleton (cons 0 doubleton))
> tripleton
(0 8 5)
> (cons 1 (cons 2 (cons 3 (cons 4 null))))
(1 2 3 4)

```

Task: Try this example!

Extracting Values

Once you've built a list, you can extract the first element with `car` and the rest of the list (all but the first element) with `cdr`. Neither operation affects the original list; they simply create a "new" value or list.

```

> (define lst (list 1 2 3 4 5))
> lst
(1 2 3 4 5)
> (car lst)
1
> (cdr lst)
(2 3 4 5)
> lst
(1 2 3 4 5)

```

Task: Try this example!

Task: Figure out how to get the second value in a list.

Common list procedures

Scheme provides several built-in procedures that operate on lists. Here are a few that are very frequently used:

The `length` procedure takes one argument, which must be a list, and computes the number of elements in the list.

Task: Play with `length` to make sure you understand it.

The `reverse` procedure takes a list and returns a new list containing the same elements, but in the opposite order.

```

> (reverse (list 1 2 3))
(3 2 1)

```

Defining Procedures

Scheme even lets you define your own procedures. Here's a simple procedure that gives you the second value in a list. You might want to enter it in the top half of the DrScheme window (the *definitions* pane).

```
(define (second lst)
  (car (cdr lst)))
```

Task:

- Enter that procedure in the definitions pane
- Try to find the second value in the list (4 5 6). In most cases, you'll fail.
- Click the *Execute* button at the top of the DrScheme window.
- Try to find the second value in the list (4 5 6).

Task: Write a procedure, (`swap-first-two lst`) that swaps the first two elements of a list. For example,

```
> (swap-first-two (list 1 2 3 4 5))
(2 1 3 4 5)
> (swap-first-two (list 10 100))
(100 10)
```

Conditionals

Scheme lets you write conditionals with the `if` procedure. It takes the form

```
(if test
    expression-to-use-if-test-holds
    expression-to-use-if-test-fails)
```

Scheme tests usually have the form

```
(test-operation val1 ... valn)
```

You can use the `null?` operation to determine if a list is empty.

```
> (null? (list 1 2 3))
#f
> (null? null)
#t
> (null? (cdr (list 1)))
#t
```

Task: Write a procedure, `only-one?` that determines if a list has only one element. (Try not to use `length`.) For example,

```
> (only-one? (list 1))
#t
> (only-one? null)
#f
> (only-one? (list 1 2 3 4 5))
#f
```

Scheme also provides five numeric tests, `<`, `<=`, `>`, `>=`, and `=`. For example, to check if x is less than y , I might use

```
> (< 4 6)
#t
> (< 6 -2)
#f
> (define x 10)
> (< x 4)
#f
```

Here's a simple use of `if` to find the smaller of two numbers.

```
(define (smaller num1 num2)
  (if (< num1 num2)
      num1
      num2))
```

Task: Test this procedure.

Task: Update `swap-first-two` so that it does not crash when you try to swap the first two elements of a list with fewer than two elements.

Overnight

1. Review the work you've done so far and make a list of any questions or comments you may have.
2. Read the following document that discusses common sorting routines.

Some Standard Sorting Algorithms

The problem of putting a collection of values in order appears so frequently that it's one of the most commonly studied problems in computer science. As with many problems, it has many different potential solutions. In this document, we consider a few of the standard algorithms that computer scientists have developed over the years.

As you read through these algorithms, you might want to try running them by hand to see how (and whether) they work.

Selection Sort

Selection sort relies on your ability to easily identify the smallest value in the collection. To sort using selection sort, you repeatedly (1) identify the smallest value and (2) put it at the front of the values not yet identified. We can naturally phrase this algorithm recursively.

```
To sort a list of values using selection sort
  If the list is empty then
    You're done
  Otherwise
    Identify small, the smallest value in the list
    Remove small from the list
    Sort the modified list
    Shove small back on the front of the sorted list
```

Of course, for this algorithm to work, we need a way to find the smallest value in a list and to remove an element from a list. We'll consider finding the smallest value and leave removal as a thought problem.

It's fairly easy to find the smallest value in a list. Here are two techniques, one using repetition and one using recursion.

```
To find the smallest value in a list (version 1: repetition)
  Let guess be the first value in the list
  For each value, val, in the list
    If  $val < guess$  then
      Let guess be val
  guess is now the smallest value in the list
```

```
To find the smallest value in a list (version 2: recursion)
  If the list contains only one element then
    That element is the smallest value in the list
  Otherwise
    Let first be the first element in the list
    Let guess be the smallest value in the remainder of the list
    The smaller of guess and first is the smallest
    value in the list
```

Insertion Sort

Insertion sort works by building the sorted list from the bottom up. We repeatedly take the first element from the list to be sorted and put it in the correct place in the sorted list.

```
To sort a list, lst, using insertion sort
  Create an empty list sorted
  Insert each value in lst into the correct place in sorted
```

```
To insert each value in lst into the correct place in sorted
  For each value, val in lst
    Insert val into the correct place in sorted
```

Note that we might also want to phrase a key step in insertion sort recursively.

```
To insert each value in lst into the correct place in sorted
  If lst is empty then
    You're done
  Otherwise
    Insert the first element in lst into sorted
    Insert the remaining elements in lst into the new sorted
```

In either case, we rely on a helper procedure. This time, the goal of the procedure is to insert a value into a sorted list.

```
To insert one value, val, at the proper position
in a sorted list, slist
  Possibility One: slist is empty
    Make a list from val
  Possibility Two: val is less than or equal to
the first value in slist
    Add val to the front of slist
  Possibility Three: val is greater than the first value
in slist
    Keep the first value in slist and insert val into
the rest of slist
```

Merge Sort

Both selection sort and insertion sort build the sorted list one element at a time. As you get more experience in computer science, you'll quickly learn that there are sometimes better ways to break up your work. The *divide-and-conquer* technique suggests breaking your work in half (or other large parts). We can use that technique along with recursion to come up with a sorting algorithm called "*Merge Sort*".

```
To sort a list using merge sort
  Split the list into two equal halves (or as equal as possible)
  Sort each half
  Merge the two halves together
```

Once again, we rely on some helper algorithms. We need a way to split the list in half and a way to merge two sorted lists into one sorted list. We'll leave splitting as an exercise to the reader. Merging is somewhat more interesting.

To **merge** two sorted lists, *slst1* and *slst2*

Possibility 1: *slst1* is empty

Just use *slst2*

Possibility 2: *slst2* is empty

Just use *slst1*

Possibility 3: The first element of *slst1* is less than element of *slst2*

Use the first element of *slst1* as the first element of the merged list.

Build the rest of the merged list by merging the rest of *slst1* with *slst2*.

Possibility 4: The first element of *slst1* is not less than the first element of *slst2*

Use the first element of *slst2* as the first element of the merged list.

Build the rest of the merged list by merging the *slst1* with the rest of *slst2*.

Day Two

- Exercise 6: Comparing Algorithms [p 24]
- Exercise 7: Studying Selection Sort [p 25]
- Exercise 8: Formalizing Requirements [p 28]
- Exercise 9: Comparing Sorting Algorithms [p 29]
- Exercise 10: Finding the Largest Value [p 30]

Exercise: Comparing Algorithms

We've seen that it's possible to write different algorithms to solve the same problem. Given more than one algorithm that solves the same problem, what criteria might you use to select one algorithm rather than another?

Exercise: Studying Selection Sort

Here is the selection sort algorithm as implemented in Scheme. (Note that I've eliminated many of the comments that would typically accompany the program code.)

```
(define (selection-sort lst)
  (if (null? lst)
      null
      (selection-sort-helper (smallest lst) lst)))

; The helper procedure handles the deletion of the smallest
; element and the repetition
(define (selection-sort-helper smallest-value lst)
  (cons smallest-value
        (selection-sort (list-remove lst smallest-value))))

(define (smallest lst)
  (if (null? (cdr lst))
      (car lst)
      (smaller (car lst) (smallest (cdr lst)))))

(define (smaller val1 val2)
  (if (< val1 val2) val1 val2))

(define (list-remove lst val)
  (if (eq? val (car lst))
      (cdr lst)
      (cons (car lst) (list-remove (cdr lst) val))))
```

You can use these procedures by adding the following line to your definitions pane and then clicking execute.

```
(load "/home/rebelsky/Web/Scheme/simple-selection-sort.ss")
```

1. Test the three key helper procedures, `smallest`, `smaller`, and `list-remove`. Here are some examples to get you started.

```
> (smaller 1 2)
> (smaller 2 1)
> (smaller -5 2)
> (smallest (list 1 2 3 4 5))
> (list-remove 4 (list 1 2 3 4 5))
```

2. Test `selection-sort` on some simple lists you create yourself.

3. For more interesting testing, you need a way to create large lists. Here's a procedure I like to use.

```
(define (random-list n)
  (if (= n 0)
      null
      (cons (random 1000) (random-list (- n 1)))))
```

You can add this definition in your definitions window. I've also put a copy of this procedure in a file, which you can load with

```
(load "/home/rebelsky/Web/Scheme/random-list.ss")
```

Try creating “random” lists of 5, 10, and 100 elements.

4. You can now combine pieces to build and sort larger lists as a way of testing selection sort. Here are some examples:

```
> (define sample (random-list 500))
> (length sample)
500
> sample
(243
 590
 ...
 866)
> (selection-sort sample)
(0
 4
 5
 ...
 999
 999)
> (selection-sort (reverse (selection-sort sample)))
(0
 4
 5
 ...
 999
 999)
```

a. Try some of your own examples.

b. Why do you think I included that last test?

5. As we discussed, it is often useful to be able to predict the running time of an algorithm. We'll use an experimental approach. In your CS courses, you will also see a more formal approach. We need to begin by gathering information on the running time of selection sort on various sizes of inputs. Here's a technique I've found useful. Enter the following in your definitions window

```
(load "/home/rebelsky/Web/Scheme/simple-selection-sort.ss")
(load "/home/rebelsky/Web/Scheme/random-list.ss")
(define sample (random-list 1000))
(define result null)
(time (set! result (selection-sort sample)))
```

When you click the **Execute** button, DrScheme will tell you how much time it spent sorting the list (use cpu time, it's the most accurate). We use the odd syntax because the default behavior of time is to print the result, and we really don't want the result printed.

Gather between ten and twenty data points for running time using different size lists.

6. Most of us can't see patterns in numeric data and instead rely on visualization techniques. Although we normally use computer programs to help us understand data, we'll work on the board to visualize the data.

a. There should be a scale on the board. Plot your points.

b. Once everyone has plotted his or her points, we'll try to figure out the approximate shape of the curve.

c. How do we test our hypothesis about the shape of the curve?

Exercise: A Formal Problem Specification

The selection sort algorithm requires us to repeatedly remove a value from a list. Without worrying about how to remove an element from a list, carefully describe what it means to remove a value from a list. Think about it in the sense of writing a contract for the programmer: What guarantees do you require for the result of the algorithm?

Exercise: Comparing Sorting Algorithms

Selection sort is not the only well-known sorting algorithm. Other common ones include insertion sort and merge sort. Which is best? Let's try to figure out experimentally (a) how each does on a variety of different kinds of inputs and (b) what the approximate curve is for the running time of each.

You can load an implementation of selection sort with

```
(load "/home/rebelsky/Web/Scheme/simple-selection-sort.ss")
```

That Scheme file provides the procedures

- `(selection-sort lst)`
- `(smallest lst)`
- `(remove val lst)`

You can load an implementation of insertion sort with

```
(load "/home/rebelsky/Web/Scheme/simple-insertion-sort.ss")
```

That Scheme file provides the procedures

- `(insertion-sort list)`
- `(insert val lst)`

You can load an implementation of merge sort with

```
(load "/home/rebelsky/Web/Scheme/simple-merge-sort.ss")
```

That Scheme file provides the procedures

- `(merge-sort lst)`
- `(merge slst1 slst2)`
- `(split lst)`

Using the techniques you've developed for evaluating the running time of selection sort, compare the running times of selection sort, insertion sort, and merge sort. Which is fastest on large problems? On small? How much faster? Does it matter if we start with a sorted list? With a reverse-sorted list?

Note that you may have difficulty finding a quadratic function that matches the running time of merge sort. You might want to try other functions to see if you can get a better fit.

Exercise: Finding the Largest Value

Here's a Scheme procedure that finds the smallest value in a list of numbers.

```
(define (smallest lst)
  ; If the list has only one element
  (if (null? (cdr lst))
      ; That one element is the smallest
      (car lst)
      ; Otherwise, use the smaller of the first element and
      ; the smallest remaining element
      (smaller (car lst) (smallest (cdr lst)))))

(define (smaller val1 val2)
  (if (< val1 val2) val1 val2))
```

1. Test those procedures to make sure that they work as you'd expect.
2. Write and test a Scheme procedure that finds the largest value in a list of procedures.

As you may have observed, `smallest` and `largest` are surprisingly similar, differing only in their use of a procedure to select between two values and the procedure used on the rest of the list. We can perhaps phrase the procedure more generally as

```
(define (best lst better)
  ; If the list has only one element
  (if (null (cdr? lst))
      ; Then that value is the best
      (car lst)
      ; Otherwise, use the better of the first element and the
      ; best remaining element
      (better (car lst) (best (cdr lst) better))))
```

3. Test `best` using a list of numbers and `smaller` as parameters.
4. Test `best` using a list of numbers and `larger` as parameters.
5. Write a `closer-to-100` procedure that, given two values, returns the one that is closer to 100.
6. Using `best` and `closer-to-100`, find the value in a list of numbers that is closest to 100.

About This Document
