

## Laboratory: Quicksort

**Summary:** In this laboratory, you will explore the Quicksort algorithm. Quicksort takes advantage of two key algorithm design techniques: divide-and-conquer and the power of randomness (a.k.a. “luck”).

### Contents:

- Exercises
  - Exercise 0: Preparation
  - Exercise 1: Partitioning
  - Exercise 2: Random Partitioning
  - Exercise 3: Quicksort Basics
  - Exercise 4: Observing Quicksort
  - Exercise 5: Detour: Counting Steps
  - Exercise 6: Analyzing Insertion Sort
  - Exercise 7: Analyzing Merge Sort
  - Exercise 8: Analyzing Quicksort
  - Exercise 9: Choosing a Sort

## Exercises

### Exercise 0: Preparation

- a. Make a copy of `sorts.scm`.
- b. Scan through the code and make sure you understand the purpose of each procedure. (You need not understand how the procedure does what it does, but you must understand what it does.)

### Exercise 1: Partitioning

- a. Generate a list, `grades`, of 50 random numbers, all between 0 and 100 (inclusive).
- b. How many of those numbers in `grades` do you expect to be less than 50?
- c. Using `partition`, segment `grades` into three parts: The values less than 50, the values equal to 50, and the values greater than 50.
- d. Determine the size of each of those three lists using  

```
(map length (partition codes 50 <))
```
- e. Determine the size of each segment in a few random lists of length 1000, using

```
(map length (partition (random-list 100 1000) 50 <))
```

- f. What do you expect partition to do if you use `<=` rather than `<` for the `precedes?` parameter?
- g. Check your answer experimentally.

## Exercise 2: Random Partitioning

- a. Generate a list, `codes`, of 1000 random numbers, all between 0 and 100,000 (inclusive).
- b. Suppose we segment `codes` into three parts, using a random pivot. How many numbers appeared in each part? You can answer this question with

```
(map length (partition codes (random-element codes) <))
```

- d. Repeat the first two steps nine more times, recording the size of each part each time.
- e. Given the results from the previous step, how well does it seem that a randomly chosen number partitions the list?

## Exercise 3: Quicksort Basics

- a. Generate a list, `codes`, of 50 random numbers, all between 0 and 1000 (inclusive).
- b. What do you expect to happen if you sort those 50 numbers using `Quicksort`, with `<` as the `precedes?` parameter? For example,

```
> (Quicksort codes <)
```

- c. Check your result experimentally.
- d. What do you expect to happen if you sort those 50 numbers using `Quicksort`, with `>` as the `precedes?` parameter? For example,

```
> (Quicksort codes >)
```

- e. Check your result experimentally.
- f. What do you expect to happen if you sort those 50 numbers using `Quicksort`, with `<=` as the `precedes?` parameter? For example,

```
> (Quicksort codes <=)
```

- g. Check your result experimentally.

## Exercise 4: Observing Quicksort

- a. Uncomment the lines in `quicksort` that report on what it is doing. (Hint: They are calls to `display` and `newline`.)
- b. Generate a list, `codes`, of 50 random numbers, all between 0 and 1000 (inclusive).
- c. Observe what happens when you sort the list using Quicksort with `<` as the `precedes?` parameter.
- d. Observe what happens when you sort the list using Quicksort with `<=` as the `precedes?` parameter.
- e. Comment-out or delete the lines you uncommented in step a.

## Exercise 5: Detour: Counting Steps

In studying algorithms, we often count the number of times we call particular procedures. For example, we might consider how many times we call `cons` or `precedes?`. The `analyst.scm` library (already included in `sorts.scm`) supports some simple analyses. There are a few simple steps to analysis:

- For a procedure of interest, replace `define` with `define$`.
- To report on the number of counted procedure calls to a single procedure in the evaluation of a particular expression, we write

```
(analyze expression proc)
```

- To report on all the counted procedure calls made during the evaluation of a particular expression, we write

```
(analyze expression)
```

a. Replace the `define` for `partition` with `define$`. This change will slow down `partition`, but will also allow us to analyze it.

a. Build a list, `nums`, of 50 random numbers, all between 0 and 1000 (inclusive).

b. How many times do you expect that `partition` will call `cons` in partitioning that list, using a pivot of 500?

c. Check your answer, using

```
(analyze (partition nums 500 <) cons)
```

d. How many times do you expect that `partition` will call `precedes?` in partitioning that list, using a pivot of 500?

e. Check your answer, using

```
(analyze (partition nums 500 <) precedes?)
```

e. What other procedures do you expect to be called, and how many times do you expect each to be called?

f. Check your answer, using

```
(analyze (partition nums 500 <))
```

g. In doing analysis, we often care more about the steps than the result (once we've determined that the procedure works correctly). Hence, instead of printing the result, we might want to store it in a variable. For example, we see just the calls for the previous part of the exercise with

```
(define parts (analyze (partition nums 500 <)))
```

Verify that this works (and that parts is defined appropriately).

## Exercise 6: Analyzing Insertion Sort

a. Update `insertion-sort` and `insert` to use `define$` instead of `define`.

b. Use the following two lines to determine how many calls to `precedes?` the `insertion-sort` procedure makes when sorting a list of fifty random numbers. (Note that the following `define` commands let us capture the results without looking at the results.)

```
(define sample (random-list 1000 50))  
(define result (analyze (insertion-sort sample <) may-precede?))
```

c. Try the previous step (that is, generating a random list, insertion sorting it, and counting the number of calls to `may-precede?`) three more times. Did you get the same number every time? If not, why do you think you got different numbers?

d. Rerun the analysis using lists of size 100, 200, and 400 (four lists of each size; simply change the 50 in the lines above and re-run the code). Compute the average number of steps for each size of list.

e. When the list length doubles, what happens to the average number of calls to `may-precede?`

## Exercise 7: Analyzing Merge Sort

a. Update `merge-sort`, `split`, and `merge` to use `define$` instead of `define`.

b. Repeat the remaining steps from exercise 6, using `merge-sort` instead of `insertion-sort`.

## Exercise 8: Analyzing Quicksort

a. Update `Quicksort` and `partition` to use `define$` instead of `define`.

b. Repeat the remaining steps from exercise 6, using `Quicksort` instead of `insertion-sort` (and `precedes?` instead of `may-precede?`).

## Exercise 9: Choosing a Sort

- a. What other procedures might you use to compare the sorting routines (other than may-precede)?
  - b. Perform tests to compare the routines based on the other procedures.
  - c. Which sorting algorithm seems the most efficient?
- 

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.