8

Sorting and Searching

Exercise 8.1. What is the best case input for *list-sort-best-first*? What is its asymptotic running time on the best case input?

Solution.

Exercise 8.2. Use the *time* special form (Section 7.1) to experimentally measure the evaluation times for the *list-sort-best-first-let* procedure. Do the results match the expected running times based on the $\Theta(n^2)$ asymptotic running time?

You may find it helpful to define a procedure that constructs a list containing n random elements. To generate the random elements use the built-in procedure *random* that takes one number as input and evaluates to a pseudorandom number between 0 and one less than the value of the input number. Be careful in your time measurements that you do not include the time required to generate the input list.random

Solution.

Exercise 8.3. Define the *list-find-best* procedure using the *list-accumulate* procedure from Section 5.4.2 and evaluate its asymptotic running time.

Solution.

Exercise 8.4. [$\star$] Define and analyze a *list-sort-worst-last* procedure that sorts by finding the worst element first and putting it at the end of the list.

Solution.

Exercise 8.5. We analyzed the worst case running time of *list-sort-insert* above. Analyze the best case running time. Your analysis should identify the inputs for which *list-sort-insert* runs fastest, and describe the asymptotic running time for the best case input.

Solution.

Exercise 8.6. Both the *list-sort-best-first-sort* and *list-sort-insert* procedures have asymptotic running times in $\Theta(n^2)$. This tells us how their worst case running times grow with the size of the input, but isn't enough to know which procedure is faster for a particular input. For the questions below, use both analytical and empirical analysis to provide a convincing answer.

a. How do the actual running times of *list-sort-best-first-sort* and *list-sort-insert* on typical inputs compare?

Solution.

b. Are there any inputs for which *list-sort-best-first* is faster than *list-sort-insert*?

Solution.

c. For sorting a long list of n random elements, how long does each procedure take? (See Exercise 8.2 for how to create a list of random elements.)

Solution.

Exercise 8.7. Define a procedure *binary-tree-size* that takes as input a binary tree and outputs the number of elements in the tree. Analyze the running time of your procedure.

Solution.

Exercise 8.8. [⋆] Define a procedure *binary-tree-depth* that takes as input a binary tree and outputs the depth of the tree. The running time of your procedure should not grow faster than linearly with the number of nodes in the tree.

Solution.

Exercise 8.9. [⋆⋆] Define a procedure *binary-tree-balance* that takes as input a sorted binary tree and the comparison function, and outputs a sorted binary tree containing the same elements as the input tree but in a well-balanced tree. The depth of the output tree should be no higher than $\log_2 n + 1$ where n is the number of elements in the input tree.

Solution.

Exercise 8.10. Estimate the time it would take to sort a list of one million elements using *list-quicksort*.

Solution.

Exercise 8.11. Both the *list-quicksort* and *list-sort-tree-insert* procedures have expected running times in $\Theta(n \log n)$. Experimentally compare their actual running times.

Solution.

Exercise 8.12. What is the best case input for *list-quicksort*? Analyze the asymptotic running time for *list-quicksort* on best case inputs.

Solution.

Exercise 8.13. [⋆] Instead of using binary trees, we could use ternary trees. A node in a ternary tree has two elements, a left element and a right element, where the left element must be before the right element according to the comparison function. Each node has three subtrees: *left*, containing elements before the left element; *middle*, containing elements between the left and right elements; and *right*, containing elements after the right element. Is it possible to sort faster using ternary trees?

Solution.

Exercise 8.14. Define a procedure for finding the longest word in a document. Analyze the running time of your procedure.

Solution.

Exercise 8.15. Produce a list of the words in Shakespeare's plays sorted by their length.

Solution.

Exercise 8.16. [⋆] Analyze the running time required to build the index.

a. Analyze the running time of the *text-to-word-positions* procedure. Use n to represent the number of characters in the input string, and w to represent the number of distinct words. Be careful to clearly state all assumptions on which your analysis relies.

Solution.

b. Analyze the running time of the *insert-into-index* procedure.

Solution.

c. Analyze the running time of the *index-document* procedure.

Solution.

d. Analyze the running time of the *merge-indexes* procedure.

Solution.

e. Analyze the overall running time of the *index-pages* procedure. Your result should describe how the running time is impacted by the number of documents to index, the size of each document, and the number of distinct words.

Solution.

Exercise 8.17. [⋆] The *search-in-index* procedure does not actually have the expected running time in $\Theta(\log w)$ (where w is the number of distinct words in the index) for the Shakespeare index because of the way it is built using *merge-indexes*. The problem has to do with the running time of the binary tree on pathological inputs. Explain why the input to *list-to-sorted-tree* in the *merge-indexes* procedure leads to a binary tree where the running time for searching is in $\Theta(w)$. Modify the *merge-indexes* definition to avoid this problem and ensure that searches on the resulting index run in $\Theta(\log w)$.

Solution.

Exercise 8.18. [⋆⋆] The site http://www.speechwars.com provides an interesting way to view political speeches by looking at how the frequency of the use of different words changes over time. Use the *index-histogram* procedure to build a historical histogram program that takes as input a list of indexes ordered by time, and a target word, and output a list showing the number of occurrences of the target word in each of the indexes. You could use your program to analyze how Shakespeare's word use is different in tragedies and comedies or to compare Shakespeare's vocabulary to Jefferson's.

Solution.