Lecture 16: Sorting, continued
10:00 AM, Oct 12, 2018

Contents

1 Selection sort
  1.1 Code ......................................................... 1
  1.2 Analysis ..................................................... 1

2 Merging two sorted lists ........................................ 1

3 Merge sort ....................................................... 3

4 Analysis ........................................................ 5

5 Summary ........................................................ 5

1 Selection sort

1.1 Code

(define selection-sort
  (lambda (L)
    (if (empty? L)
        empty
      (cons (list-min L) (selection-sort (omit-from (list-min L) L)))))

1.2 Analysis

Let’s trust that you can show that list-min is an \( O(n) \) procedure. Therefore for constants \( c_1 \) and \( n_1 \), its running time on inputs of size \( n \) is bounded by \( c_1 n \) for \( n > n_1 \).

Let \( f(n) \) be the worst-case running time of omit-from on inputs of size \( n \).

\[
\begin{align*}
  f(1) & \leq a \\
  f(n) & \leq b + f(n-1)
\end{align*}
\]

Thus \( f(n) \leq a + bn \).

Let \( g(n) \) be the worst-case running time of selection-sort on inputs of size \( n \). When \( n > 0 \), the procedure must make two calls to list-min on an input of size \( n \) and one call to omit-from
on an input of size $n - 1$ and a recursive call on input of size $n$. Aside from the recursive call, the number of operations is at most $c_2n$ when $n > n_2$ where $c_2$ and $n_2$ are constants.

$$g(n) \leq d \text{ for } n \leq n_2$$
$$g(n) \leq c_2n + g(n - 1) \text{ for } n > n_2$$

Therefore $g(n)$ is $O(n^2)$.

2 Merging two sorted lists

Let’s think again about insertion sort. The strategy was:

- Separate the input list into (a) the first item and (b) the rest.
- Sort the rest.
- Combine the first item into the result of sorting the rest.

Here’s a slightly modified version:

- Separate the input list into (a) the first and second items and (b) the rest (items 3 through $n$).
- Sort the rest.
- Combine the first and second items into the result of sorting the rest.

This won’t change the order of growth of the running time because the recurrence looks like $f(n) \leq b + cn + f(n - 2)$, which still gives us that $f(n)$ is $O(n^2)$.

But maybe it’s slightly better? The good news is that the recursive call works on a smaller list. The bad news is that merging the two items into the result of sorting that smaller list is more complicated.

We could try an algorithm that pulls out even a few more items, sorts the rest, and merges the missing items back into the sorted list. As the algorithm is changed to pull out more items, merging them back into the sorted list gets more complicated.

Here’s one idea to make that process easier: the algorithm take the items it pulled out, and it sorts them as well. This makes it easier to merge those items with the rest. We use a merge procedure:

- input: two sorted lists $L_1$ and $L_2$ of numbers
- output: a single sorted list consisting of all occurrences of numbers in $L_1$ and $L_2$.

example:

input: (3 5 7) and (2 4 6)
output: (2 3 4 5 6 7)
The strategy for merge is this:

- Compare the cars of the two lists.
- Whichever one is smaller, remove it and recursively merge the remaining lists
- Cons the smaller element onto the recursive output.

Recursion diagrams:

original input: (3 5 7) and (2 4 6)
  recursive input: (3 5 7) and (4 6)
  recursive output: (3 4 5 6 7)
original output: (2 3 4 5 6 7)

original input: (1 5 7) and (2 4 6)
  recursive input: (5 7) and (2 4 6)
  recursive output: (2 4 5 6 7)
original output: (1 3 4 5 6 7)

(define merge
  (lambda (L1 L2)
    (cond...
      ((< (car L1) (car L2)) (cons (car L1) (merge (cdr L1) L2)))
      (#true (cons (car L2) (merge L1 (cdr L2)))))))

What base cases do we need? The code takes the car of each list, so we had better separately handle the case where each list is empty. Fortunately, merging an empty list with another list is easy.

(define merge
  (lambda (L1 L2)
    (cond
      ((empty? L1) L2)
      ((empty? L2) L1)
      ((< (car L1) (car L2)) (cons (car L1) (merge (cdr L1) L2)))
      (#true (cons (car L2) (merge L1 (cdr L2)))))))

3 Merge sort

Now we can build a sorting algorithm. It all depends on how many items we pull out. We can use the take and drop procedures you wrote earlier:

Take

- input: nonnegative integer \( n \), list \( L \) with at least \( n \) items
• output: list consisting of first \( n \) items of \( L \)

\textbf{drop}

• input: nonnegative integer \( n \), list \( L \) with at least \( n \) items
• output: list consisting of all items \( L \) except the first \( n \) items

\begin{verbatim}
(define merge-sort
 (lambda (L)
   ...
   (merge (merge-sort (take 10 L)) (merge-sort (drop 10 L))))))
\end{verbatim}

What should base case be? If \( L \) has length less than 10, we cannot take 10 times from it, so we had better handle the case where \( L \) has fewer than 10 items.

What do we do in that case? The input size is at most a constant, so whatever we do the running time will be at most a constant. We already have \texttt{insertion-sort} working, so let’s use that.

\begin{verbatim}
(define merge-sort
 (lambda (L)
   (if (< (length L) 10)
     (insertion-sort L)
     (merge (merge-sort (take 10 L)) (merge-sort (drop 10 L))))))
\end{verbatim}

Great, now let’s try it out!

\textit{See lecture capture for the demonstration of what happens}

Uh oh, something went wrong. Maybe this isn’t such a fast algorithm after all! It’s certainly much slower than selection sort for small inputs.

Let’s draw a diagram of the invocations.

\textit{See slide 1 in the lecture slides for our diagram}

We need to correct this. Splitting a ten-item list into a ten-item list and a zero-item list does not make progress, since the first recursive call is not on a smaller input. So we need to change this logic. It turns out, there’s a fix to this issue which is shown here:

\begin{verbatim}
(define merge-sort
 (lambda (L)
   (if (<= (length L) 10)
     (insertion-sort L)
     (merge (merge-sort (take 10 L)) (merge-sort (drop 10 L))))))
\end{verbatim}

While this does work, it’s not exactly \texttt{merge-sort}. The question to ask here is why did we choose 10 as the arbitrary length to check for? It turns out, we don’t really have a reason. Because of this, a better length to check for is half the length of the list, and to split the list in half with each recurrence. This is shown here:
(define merge-sort
  (lambda (L)
    ...
    (merge (merge-sort (take (quotient (length L) 2) L))
           (merge-sort (drop (quotient (length L) 2) L))))))

Fill in the base case:

(define merge-sort
  (lambda (L)
    (if (<= (length L) 1)
        (insertion-sort L)
        (merge (merge-sort (take (quotient (length L) 2) L))
               (merge-sort (drop (quotient (length L) 2) L))))))

Let’s try comparing running times between merge-sort and insertion-sort, say on a list of length 5000. Try a list of length 10,000. By how much does the running time increase?

*See lecture capture for the demonstration of these comparisons.*

It turns out that for small lists, insertion-sort is actually quicker than merge-sort. How can we take advantage of that?

The answer is to change merge-sort to use insertion-sort when the list gets small enough.

## 4 Analysis

Let’s look at the tree of invocations of pure merge-sort. If we think about the operations which are taking place at each level of our merge-sort tree, there are \( n \) elements being processed. This is shown by adding up all of the \( n \) values at each level of the tree. Because of this, we can say that for each level of our tree, the runtime can be described as linear, otherwise known as \( O(n) \), otherwise known as \( C \times n \).

However, now we have to account for the recursive calls which are taking place. This can also be described as the number of levels to our tree. In this tree, we have 6 levels, so for this example of a call to merge-sort, our runtime is \( 6 \times (C \times n) \). However, where did this 6 come from? It came from taking the \( \log_2 \) of \( n \). Therefore, more generally, the runtime of our merge-sort algorithm can be described as \( O(n \log_2 n) \) time.

## 5 Summary

- Runtime complexity is important to consider when creating different algorithms
- While runtime is calculated by looking at very large inputs, looking at which algorithms are efficient for shorter inputs is also useful.
- The merge-sort algorithm runs in \( O(n \log_2 n) \) time.