Lecture 15: Insertion Sort, Selection Sort, and More

10:00 AM, Oct 10, 2018

Contents

1 insertion sort
   1.1 The code ....................................................... 1
   1.2 Upper bound on running time .................................. 1
   1.3 Lower bound on running time .................................. 2

2 Selection sort ........................................ 2

3 omit-from ...................................................... 5

1 insertion sort

1.1 The code

insertion-sort

(define insertion-sort
  (lambda (L)
    (if (empty? L)
        empty
        (insert-in-order (car L) (insertion-sort (cdr L)))))

insert-in-order

• input: number \( x \), list \( lst \) of numbers in ascending order

• output: list consisting of \( x \) and elements of \( lst \), all in ascending order.

(define insert-in-order
  (lambda (x lst)
    (if (empty? lst)
        (list x)
        (if (<= x (car lst))
            (cons x lst)
            (cons (car lst) (insert-in-order x (cdr lst)))))))
1.2 Upper bound on running time

Analysis of insert-in-order shows that it is an $O(n)$ procedure. That is, there are constants $c$ and $n_0$ such that the worst-case running time on an input of size $n$ is at most $cn$ for $n \geq n_0$.

Therefore the recurrence for insertion-sort is as follows. Let $f(n)$ be the worst-case running time of the procedure on inputs of size $n$.

$$
g(n) \leq \begin{cases} 
a & \text{for } n \leq n_0 \\
b + cn + f(n-1) & \text{for } n > n_0
\end{cases}
$$

Therefore

$$
g(n) \leq (b + cn) + (b + c \cdot (n - 1)) + \cdots + (b + c \cdot (n_0 + 1)) + f(n_0) \\
\leq (n - n_0)(b + cn) + a \\
\leq a + bn + cn^2
$$

We conclude that $f(n)$ is $O(n^2)$.

1.3 Lower bound on running time

We can also show that the running time for insert-in-order is $\Omega(n)$. Recall this means that there are constants $c$ and $n_0$ such that the worst-case time is at least $cn$ for $n \geq n_0$. (I’m using the same names $n_0$ and $c$ for constants as I did for $O(n)$, but they don’t have to match.)

We will use the notation $\lceil x \rceil$ to denote the smallest integer greater than or equal to $x$. For example, $\lceil 7.5 \rceil$ is 8 and $\lceil 7 \rceil$ is 7.

Now let’s write a recurrence relation for $f(n)$:

$$
f(n) \geq cn + f(n - 1) \text{ for } n \geq n_0
$$

Therefore

$$
f(n) \geq \begin{cases} 
cn & \text{for } n \leq n_0 \\
cn + c \cdot (n - 1) + (n - 2) + \cdots + c \cdot \lceil n/2 \rceil + (n/2 + 1) + \cdots + c \cdot (n_0 + 1) & \text{for } n > n_0
\end{cases}
$$

$$
\geq \begin{cases} 
cn & \text{for } n \leq n_0 \\
cn + c \cdot (n - 1) + (n - 2) + \cdots + c \cdot \lceil n/2 \rceil \\
cn + c \cdot \lceil n/2 \rceil + \cdots + c \cdot \lceil n/2 \rceil \\
cn + c \cdot \lceil n/2 \rceil \cdot (n - \lceil n/2 \rceil + 1) \\
cn + c \cdot (n/2) \cdot (n - (n/2 + 1) + 1) \\
cn + c \cdot (n/2)(n/2) = \frac{cn}{4}n^2
\end{cases}
$$

2 Selection sort

I’ll illustrate another algorithm for sorting. Say we want to sort $(3 \ 1 \ 4 \ 1 \ 5 \ 9)$.

The smallest item in this list is 1, so we know that 1 goes at the beginning of the output list. We will plan to cons 1 onto the result of recursively sorting the list obtaining by removing that occurrence
of 1, namely the list \((3 \ 4 \ 1 \ 5 \ 9)\). Sorting that list gives us \((1 \ 3 \ 4 \ 5 \ 9)\). We cons 1 onto the list, obtaining \((1 \ 1 \ 3 \ 4 \ 5 \ 9)\).

We often go from the original input list to the recursive input list by using cdr. That’s the easiest way of getting a list that is smaller by one, but not always the “right” way.

Recursion diagram:

original input: \((3 \ 1 \ 4 \ 1 \ 5 \ 9)\)

recursive input: \((3 \ 4 \ 1 \ 5 \ 9)\)
recursive output: \((1 \ 3 \ 4 \ 5 \ 9)\)
original output: \((1 \ 1 \ 3 \ 4 \ 5 \ 9)\)

In the past, we have talked about the “ideation space” between the recursive output and the original output because often the hard part is deriving the original output from the recursive output. This time the hard part is deriving the recursive input from the original input.

It should be clear, however, what we need to do:

- calculate the minimum element of the list, and
- compute the list obtained by omitting that element from the list.

Let’s first write the procedure \(\text{list-min}\) to compute the minimum. To make this procedure easier to write, we will require that its input list be nonempty.

- input: a nonempty list of numbers
- output: the smallest number in the list

original input: \((3 \ 1 \ 4 \ 1 \ 5 \ 9)\)

recursive input: \((1 \ 4 \ 1 \ 5 \ 9)\)
recursive output: 1
original output: 1

In this example, the original output is just the recursive output. This is so easy, it should cue us that we need another case. This is the case in which the car of the list is smaller than any other element in the list.

original input: \((1 \ 5 \ 9)\)

recursive input: \((5 \ 9)\)
recursive output: 5
original output: 1

This should make it more clear: the original output is whichever is smaller, the car of the original input or the recursive output. We can use the procedure \(\text{my-min}\) that, given two numbers, outputs whichever is smaller:
(define my-min
  (lambda (x y)
    (if (> x y)
        y
        x))))

We called it my-min because min is already built in. Now we can write list-min using either my-min or min:

(define list-min
  (lambda (L)
    (if (empty? (cdr L))
        (car L)
        (min (car L) (list-min (cdr L))))))

There is a nicer way to write list-min, using the fact that Racket considers infinity a number.

original input: 1, (3 1 4 1 5 9)
recursive input: 1, (1 4 1 5 9)
recursive output: (4 1 5 9)
original output: (3 4 1 5 9)

The original output was derived from the recursive output by consing the car of the original input, so we do need another case: the case in which the car is the element to omit:

original input: 1, (1 4 1 5 9)
original output: (4 1 5 9)

In this case, what good would a recursive call do? We already know what the output should be, just by virtue of x matching the car: the output should be the cdr.

This is a base case. Often identifying the base case depends on the structure of the input (e.g. whether an input list is empty or singleton) but this time it depends on the specifics of the data.

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

Analysis: Let’s assume 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_1n for n > n_1.

Let f(n) be the worst-case running time of omit-from on inputs of size n.
\[ f(1) \leq a \]
\[ f(n) \leq b + f(n-1) \]

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.

\[
\begin{align*}
g(n) &\leq d &\text{for } n \leq n_2 \\
g(n) &\leq c_2n + g(n-1) &\text{for } n > n_2
\end{align*}
\]

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

### 3 omit-from

**omit-from**

- input: number \( x \), list \( L \) of numbers such that \( x \) occurs in \( L \)
- output: the list consisting of all elements of \( L \) except for the first occurrence of \( x \)

**Quiz:** Write \texttt{omit-from}

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)