CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 13

The sorting problem

Suppose we have a list of numbers and want to produce the same list in ascending (i.e., increasing non-decreasing) order.

We need a function that consumes a list of numbers and produces a list of numbers, such that the output list of numbers is sorted in ascending order.

Sorting may not sound like a major problem, but vast computational power is used to sort enormous data sets.

Therefore people have devised many sorting *algorithms* – general methods – which vary in running time, i.e., the number of steps taken by the algorithm.

Running time is expressed in terms of *n*, the size of the input, and the lower, the better.

Approach 1: *Insertion sort*

Consider an unordered list of numbers:

```
> (define unordered '(5 3 6 2))
```

The first element is 5.

The rest is `(3 6 2)`.

We can sort the rest: `(2 3 6)`.

And then insert 5 into the sorted list: `(2 3 5 6)`.

This is a simple recursive method of sorting named *insertion sort*.

To define the insertion sort algorithm, we need a function that, given a number and a sorted list, returns a new list with the number placed in the appropriate location.

```
;; INSERT
;; ------
;; INPUTS: NUM, a number
;;         SORTED, a list of numbers already sorted
;;           into non-decreasing order.
;; OUTPUT: The list obtained by inserting NUM into SORTED
;;   while preserving the non-decreasing ordering.
(define insert
  (lambda (num sorted)
    ...?...)

> (insert 4 '(1 2 3 5 6 7))
(1 2 3 4 5 6 7)

> (insert 4 '())
(4)
```

```
;; INSERT
;; ------
;; INPUTS: NUM, a number
;;         SORTED, a list of numbers already sorted
;;           into non-decreasing order.
;; OUTPUT: The list obtained by inserting NUM into SORTED
;;   while preserving the non-decreasing ordering.
(define insert
  (lambda (num sorted)
    (cond
      ;; Base case 1: SORTED is empty
      ((null? sorted)
       (list num))

      ;; Base case 2: We found where NUM goes
      ((<= num (first sorted))
       (cons num sorted))

      ;; Recursive case: We haven't found where NUM goes
      (else
       (cons (first sorted)
             (insert num (rest sorted)))))))
```

Now that we have a way to insert numbers into an already sorted list, we still need a function that takes a totally unordered list of numbers and returns the same numbers sorted in non-decreasing order.

Insertion sort

```
;; INSERTION-SORT
;; --------------
;; INPUTS: LST, a list of numbers
;; OUTPUT: A list containing the same elements as LST, but
;;   sorted into non-decreasing order.
(define insertion-sort
  (lambda (lst)
    ...?...))

> (insertion-sort '(2 3 1))
(1 2 3)
```

```
;; INSERTION-SORT
;; --------------
;; INPUTS: LST, a list of numbers
;; OUTPUT: A list containing the same elements as LST, but
;;   sorted into non-decreasing order.
(define insertion-sort
  (lambda (lst)
    (if (null? lst)
        ;; Base case: The list is already sorted.
        '()
        ;; Recursive case: Sort the rest of the list and
        ;;   then put the first item in the right place.
        (insert (first lst)
                (insertion-sort (rest lst))))))
```

Insertion sort: accumulator version

```
;; INSERTION-SORT
;; --------------
;; INPUTS: LST, a list of numbers
;; OUTPUT: A list containing the same elements as LST, but
;;   sorted into non-decreasing order.
(define insertion-sort
  (lambda (lst)
    (letrec ((helper
              (lambda (lst acc)
                (if (null? lst)
                    acc
                    (helper (rest lst)
                            (insert (first lst) acc))))))

      (helper lst '()))))
```

How many items in a list of n items will insertion sort need to compare?

To insert the first element, it might need to go through the other $n - 1$ items.

To insert the second element, it might need to go through $n - 2$ items.

So, $(n - 1) + (n - 2) + \cdots + 1 = n^2/2$

Since 1/2 is a constant factor, we simplify this and say that insertion sort has a worst case performance of n^2 operations, written $O(n^2)$

Variant: *Sorting strings*

Comparing strings

```
> (string<=? "aardvark" "zebra")
#t
> (string<=? "zebra" "aardvark")
#f
```

String comparison predicates:

```
string<?
string<=?
string>?
string>=?
```

Modify **`insert`** for strings

```
(define insert-string
  (lambda (item lst)
    (cond ((null? lst)
           (list item))

          ((string<=? item (first lst))
           (cons item lst))

          (else
           (cons (first lst)
                 (insert-string item (rest lst)))))))
```

Modify **insertion-sort** for strings

```
(define insertion-sort-strings
  (lambda (lst)
    (if (null? lst)
        '()
        (insert-string
          (first lst)
          (insertion-sort-strings
            (rest lst))))))
```

Generating data to sort

Pseudorandom numbers and nondeterminism

Every function we've seen so far is ***deterministic***, i.e., given an input, it will always return the same result.

An example of a nondeterministic function is random, which takes as its input an upper bound and generates a random number *i*, such that 0 < *i* < *bound*:

```
> (random 2) ;; Flip a coin
0
```

```
> (random 6) ;; Roll a die
4
```

"Pseudorandom?"

How random does something need to be to be random?

How can anything a computer does really be random?

Approaches:

- User inputs (keyboard, mouse)
- Atmospheric noise
- Radio static

Why does this matter?

Generating a list of random numbers in the range 0 to (bound − 1)

```
;; RANDOM-NUMS
;; -----------
;; INPUTS: N, a positive integer
;;         BOUND, a positive integer
;; OUTPUT: A list containing N numbers, each randomly
;;   generated from the set {0, 1, ... BOUND - 1}
(define random-nums
  (lambda (n bound)
    (if (<= n 0)
        ;; Base case: No (more) numbers to generate
        '()
        ;; Recursive case:
        ;;   Generate at least one more number
        (cons (random bound)
              (random-nums (- n 1) bound)))))
```

```
> (define data (random-nums 10000 100000))

> (time (insertion-sort data) #t)
cpu time: 4816 real time: 4871 gc time: 446
#t
```

Approach 2: *Merge sort*

The ***merge sort*** algorithm uses a “divide and conquer” approach.

It breaks the list in half, sorts the halves, and then merges the sorted halves.

It can do this recursively until a list has fewer than two elements, in which case it’s already sorted!

We can start by figuring out how to split a list into equal halves.

```
;; SPLIT-ACC
;; ---------
;; INPUTS: LST, any list
;;         LEFT, RIGHT, two list accumulators
;; OUTPUT: A list containing two sub-lists,
;;   each having (roughly) half the elements of LST
(define split-acc
  (lambda (lst left right)
    ...?...))

(define split
  (lambda (lst)
    (split-acc lst '() '())))

> (split '(1 2 3 4 5 6 7))
((6 4 2) (7 6 4 2))
```

```
(define split-acc
  (lambda (lst left right)
    (cond
     ;; Base case 1: LST is empty
     ;;   Create a two-element list whose elements are
     ;;   LEFT and RIGHT.
     ((null? lst)
      (list left right))

     ;; Base case 2: LST has one element
     ;;   Arbitrarily let RIGHT accumulate the one element
     ((null? (rest lst))
      (list left
            (cons (first lst) right)))

     ;; Recursive Case: LST has at least two elements
     ;;   Put one on each accumulator.
     (else
      (split-acc (rest (rest lst))
                 (cons (first lst) left)
                 (cons (second lst) right))))))
```

Now we can consider how we merge two lists that are already sorted.

```
;; MERGE
;; -----
;; INPUTS: SORTED1, SORTED2, two already sorted lists of
;;   numbers
;; OUTPUT: A list containing all of the elements of both
;;   input lists, sorted.
(define merge
  (lambda (sorted1 sorted2)
    ...?...))

> (merge '(1 2 3) '())
(1 2 3)

> (merge '(1 3 5) '(2 4 6))
(1 2 3 4 5 6)
```

```
(define merge
  (lambda (sorted1 sorted2)
    (cond
     ;; Base Case 1
     ((null? sorted1)
      sorted2)

     ;; Base Case 2
     ((null? sorted2)
      sorted1)

     ;; Recursive Case 1: Both lists are non-empty
     ((<= (first sorted1) (first sorted2))
      (cons (first sorted1)
            (merge (rest sorted1) sorted2)))

     ;; Recursive Case 2
     (else
      (cons (first sorted2)
            (merge sorted1 (rest sorted2)))))))
```

Given split and merge, writing the main merge-sort function isn't too difficult!

We split the input when it has more than one element, recursively sorting the halves and then merging them.

```
;; MERGE-SORT
;; ----------
;; INPUT: LST, a list of numbers
;; OUTPUT: A list containing the elements of LST in
;;   non-decreasing order.
(define merge-sort
  (lambda (lst)
    (if (or (null? lst)
            (null? (rest lst)))
        ;; Base case: LST has 0 or 1 elements, so it's
        ;; already sorted
        lst
        ;; Recursive case: LST has at least TWO elements
        (let* ((pair-of-lists (split lst))
               (left (first pair-of-lists))
               (right (second pair-of-lists))
               (sorted-left (merge-sort left))
               (sorted-right (merge-sort right)))
          (merge sorted-left sorted-right)))))
```

```
> (define data (random-nums 10000 100000))

> (time (insertion-sort data) #t)
cpu time: 4816 real time: 4871 gc time: 446
#t

> (time (merge-sort data) #t)
cpu time: 58 real time: 59 gc time: 11
#t
```

That's a lot quicker!

The time to sort any list will increase as the list gets longer, but it grows quicker for insertion sort than for merge sort.

While insertion sort was $O(n^2)$, merge sort is only $O(n \log n)$

The study of execution time is called *algorithm analysis*, and the theoretical bound for a given problem is the subject of *complexity theory*.

Acknowledgments

This lecture incorporates material from:

- Luke Hunsberger
- Jennifer Walter