Lists in Racket; Complexity and Big-O

January 23–24, 2012
CS 60: Principles of Computer Science

Suppose there's a new meal program for faculty. At the start of the year, I choose between:

- ✓ **Option A.** One fixed fee for the whole year.
- ✓ **Option B.** A registration fee, plus a fixed per-meal charge.
- ✓ **Option C.** A per-plate charge, plus a per-meal surcharge that doubles (!) every meal.

At the end of the year, I might get some of my money back (e.g., if Dining Services made a big profit).

Which should I pick?

Recall: Racket

```
;; (fac n) returns n!
(define (fac n)
  (if (< n 2)
    1
    (* n (fac (- n 1)))))

;; (fib n) returns the n-th Fibonacci number
(define (fib n)
  (cond
    ( (equal? n 0)  0                   )
    ( (equal? n 1)  1                   )
    ( else          (+ (fib (- n 1))
                       (fib (- n 2))) )
  )
)
```

Seeing Double

What is this?

```
(define answer (* 6 7))
```

✓ **It's code!** *It creates a variable named* `answer` *whose value is 42.*

✓ **It's data!** It's a list containing three items:

1. The symbol `define`
2. The symbol `answer`
3. A list that contains a symbol and two integers.

How does Racket decide?

The Code Assumption

Racket assumes everything you write is ***code***, and evaluates it.

```
(* 3 27)              ;; ===> 81

(odd? (* 3 27))       ;; ===> #t

(1 2 3)               ;; oops

(reverse (1 2 3))     ;; oops
```

What if we want to refer to data?

You Can **quote** Me On This!

The **quote** operation simply returns its argument **unevaluated**.

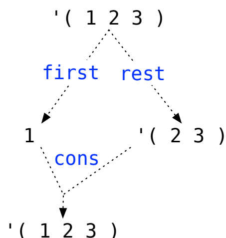
```
(quote (* 3 27))             ;; ===> (* 3 27)

(quote (odd? (* 3 27)))      ;; ===> (odd? (* 3 27))

(quote (1 2 3))              ;; ===> (1 2 3)

(quote (reverse (1 2 3)))    ;; ===> (reverse (1 2 3))

(reverse (quote (1 2 3)))    ;; ===> (3 2 1)
```

"Syntactic Sugar" Makes Racket Sweeter

In practice, we'll always use a handy abbreviation

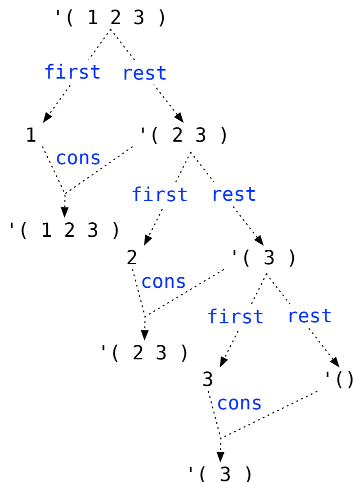
```
'(* 3 27)                    ;; ===> (* 3 27)

'(odd? (* 3 27))             ;; ===> (odd? (* 3 27))

'(1 2 3)                     ;; ===> (1 2 3)

'(reverse (1 2 3))           ;; ===> (reverse (1 2 3))

(reverse '(1 2 3))           ;; ===> (3 2 1)
```

Nonempty Lists Have a `first` and a `rest`




```
(first '(1 2 3))   ====>  1

(rest '(1 2 3))    ====>  '(2 3)

(cons 1 '(2 3))    ====>  '(1 2 3)
```

rest's All The Way Down...



Other Useful Operations, by Example

```
(null? '(1 2 3))          ;; ===>  #f
(null? '()     )          ;; ===>  #t

(second '(1 2 3))         ;; ===>  2
(third  '(1 2 3))         ;; ===>  3

(length '(1 2 3))         ;; ===>  3

(append '(1 2) '(3))      ;; ===> '(1 2 3)

(list 1 2 (+ 1 2))        ;; ===> '(1 2 3)

'(1 2 (+ 1 2))            ;; ===> '(1 2 (+ 1 2))
```

Exercise: What Do These Evaluate To?

```
(cons 'a '(b c))              (cons '(a) '(b c))

(list 'a '(b c))              (list '(a) '(b c))

(append 'a '(b c))            (append '(a) '(b c))

(reverse '((a b) c d))        (first ''a)
```

"Quiz" 1 NAME:

Fill in the 6 blanks.

```
(define L '(h a r))

(define M '(e v))

(list L L)                ;; ===>

(cons L M)                ;; ===>

(list M 'u 'd 'd)         ;; ===>

(append L (reverse L))    ;; ===>

                          ;; ===> '(r a v e)

                          ;; ===> '(h a r v e y)   [use 'y]
```

Running Time

Given a fixed "algorithm," the actual running time can depend on

- ✓ the way the program is translated into machine code
- ✓ the details of the hardware (CPU, memory, cache, hard drive)
- ✓ the details of the operating system
- ✓ the particular input being tested
- ✓ and *many* other factors.

Asymptotic analysis ("Big O") lets us ignore all these factors, by

- ✓ Finding an upper bound on number of "steps" required
- ✓ Ignoring constant factors

Big-O Examples

We ignore constant factors, and non-dominant terms.

- ✓ 60 is $O(1)$.
- ✓ $n + 60$ is $O(n)$
- ✓ $2n^2 + 5n$ is $O(n^2)$
- ✓ $\ln(n^9)$ is $O(\log n)$.
- ✓ $12n^2 + 4n^3 + 235255$ is $O(n^3)$.

Example: Computing the Running Time of `halve-count`

Algorithm:

```
(define (halve-count N)
   (if (equals? 1 N)
       0
       (+ 1 (halve-count (quotient N 2)))
```

1. How do we measure the "size" n of the input?
   The value of the input N
2. What constant-time "step" occurs most frequently?
   (***or***, within a constant factor of being most frequent!)
   `equals?`, *or* **`+`**, *or* **`quotient`**, *or calling the* **`halve-count`** *function, or ...*
3. Count (asymptotically) occurrences of this step.
   $O(\log n)$, *where* n *is the value of the input*

Example: Finding the Minimum in a List of Numbers

Algorithm:

Check each number in turn; keep track of the minimum so far; return that value when we reach the end.

1. How do we measure the "size" n of the input?
   The length of the given list
2. What constant-time "step" occurs most frequently?
   (***or***, within a constant factor of being most frequent!)
   comparisons, or . . .
3. Count (asymptotically) occurrences of this step.
   $O(n)$, where n is the length of the list

Example: Sorting a List of Numbers

Algorithm ("Selection Sort")

> *Find the smallest number that has not yet been marked, mark it, and copy it into the output. Repeat until all elements have been marked.*

1. How do we measure the "size" n of the input?
   The length of the given list
2. What constant-time "step" occurs most frequently?
   (***or***, within a constant factor of being most frequent!)
3. Count (asymptotically) occurrences of this step.
   $O(n^2)$, where n is the length of the list

"Quiz" 2 NAME:

Estimate the worst-case complexity for these problems:

1. Finding the minimum of a list, where n is the length of the list. Count the number of comparisons. $O(n)$
2. Sorting a list of length n via repeated minimum-finding. Count the number of comparisons. $O(n^2)$
3. Determine a hidden integer between 1 and n. You're told whether each guess is $>$ or $<$ (or $=$). Count the number of guesses required.
4. Opening a combination bike lock, where the combination has n decimal digits. Count the number of guesses required.
5. Finding the dot product of two vectors, both of length n. Count the multiplications required.
6. Multiply two square n by n matrices. Count the multiplications required.
7. Find the median of a list of length n. Count the number of comparisons.
8. Determine if a program with n characters will go into an infinite loop when run. Count all operations.

Evaluating Big-O

Suppose I want to choose among three algorithms that process a list of integers. In terms of n, the number of integers, their running time is:

- ✓ **Algorithm A.** $O(1)$.
- ✓ **Algorithm B.** $O(n)$.
- ✓ **Algorithm C.** $O(2^n)$.

Which should I pick?

A Story

Suppose there's a new meal program for faculty. At the start of the year, I choose between:

- ✓ **Option A.** One fixed fee for the whole year.
- ✓ **Option B.** A registration fee, plus a fixed per-meal charge.
- ✓ **Option C.** A per-plate charge, plus a per-meal surcharge that doubles (!) every meal.

At the end of the year, I might *get some* of my money back (e.g., if Dining Services made a big profit).

Which should I pick?

Evaluating Big-O

Suppose I want to choose among three algorithms that process a list of integers. In terms of n, the number of integers, their running time is:

- ✓ **Algorithm A.** $O(1)$.
- ✓ **Algorithm B.** $O(n)$.
- ✓ **Algorithm C.** $O(2^n)$.

Which should I pick?