

COMPUTER SCIENCE 51

Spring 2009
cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

Today

- Complexity of algorithms
 - space & time
 - asymptotic growth, big-O notation
 - recurrences
 - tail-calls & tail-recursion

High-Level Goal

- Compare algorithms & data structures:
 - why is merge-sort “better” than insertion-sort?
 - why is a rb-tree “better” than an association list?
 - why is it “better” to use foldl than foldr to sum a list?
- One possible comparison:
 - measure the running time & space used
 - problems?

Problems with Measuring

- If you measured the running time as a function of the input list length n , you'd see that:
 - insertion-sort runs in worst-case time $k_1 * n^2 + k_2$ for some constants k_1 and k_2 .
 - merge-sort runs in worst-case time $c_1 * n * \lg_2 n + c_2$ for constants c_1 and c_2 .
- So times are different for different inputs.
- The constants depend upon many factors:
 - The language & compiler used
 - The machine used
 - What other processes are running
 - The temperature, your altitude, ...

Plotting it out

$$T_{ms}(n) = 2 * n * \lg_2 n + 30 \quad T_{is}(n) = 1 * n^2 + 0$$

Shifting the Curves

$$T_{ms}(n) = 4 * n * \lg_2 n + 30 \quad T_{is}(n) = 1 * n^2 + 0$$

Asymptotic Complexity

- Given the two functions:
 - $T_{is}(n) = k_1 * n^2 + k_2$ and $T_{ms}(n) = c_1 * n * g_2 n + c_2$
- as n grows towards infinity T_{is} grows "faster" than T_{ms} *regardless* of the constants.
 - in contrast, $T(n) = q_1 * n^2 + q_2 * n + q_3$ does *not* grow faster than T_{is} independent of the constants.
- If we can formalize "faster", then we have a *robust* way to compare algorithms.
 - Won't depend upon language, compiler, machine, OS, etc. as long as these only affect the constants.

Our Goals

- Examine a piece of code and determine it's worst case running time (or space) as a function of its inputs.
 - without having to measure it.
 - obviously, in terms of symbolic constants.
 - e.g., $T_{ms}(n) = c_1 * n * g_2 n + c_2$
- Be able to compare the symbolic running time (space) of two functions and determine which one is asymptotically better.
 - without resorting to plots.
 - For this, we will utilize big-O notation.

Formal Big-O notation

- Given $f : \text{number} \rightarrow \text{number}$:
- $O(f)$ is the set of all functions g such that:
 - for all $n > 0$,
 - there exists a number c such that
 - $g(n) \leq c * f(n)$
- If $f \in O(g)$ but g is not in $O(f)$, then g grows faster than f .
 - we'll write $g \gg f$ to reflect this.
- Example:
 - I claim that $\lambda n. 10 * n^2 + 3 \in O(\lambda n. n^2)$
 - But not in $O(\lambda n. n * g_2 n)$

Arguing big-O

- Claim that $\lambda n. 10 * n^2 + 3 \in O(\lambda n. n^2)$
 - must find a constant c such that for all $n > 0$, $10 * n^2 + 3 \leq c * n^2$
- When $n=1$: $(\lambda n. 10 * n^2 + 3) 1 = 10 * 1 + 3 = 13$, so need $13 \leq c$
- When $n=2$: $(\lambda n. 10 * n^2 + 3) 2 = 10 * 4 + 3 = 43$, so need $43 \leq 4 * c$, or $10.75 \leq c$
- When $n=3$: $(\lambda n. 10 * n^2 + 3) 3 = 10 * 9 + 3 = 93$, so need $93 \leq 9 * c$, or $10.34 \leq c$
- When $n=4$: $(\lambda n. 10 * n^2 + 3) 4 = 10 * 16 + 3 = 163$, so need $163 \leq 16 * c$, or $10.19 \leq c$
- Perhaps 13 is a good choice for c ?

Arguing big-O Continued

Lemma: for all $n > 0$, $10n^2 + 3 \leq 13n^2$.

Proof: assume $n=m+1$ for some $m \geq 0$.

Need to show $10n^2 + 3 \leq 13n^2$

$$= 10(m+1)^2 + 3 \leq 13(m+1)^2$$

$$= 10(m^2 + 2m + 1) + 3 \leq 13(m^2 + 2m + 1)$$

$$= 10m^2 + 20m + 13 \leq 13m^2 + 26m + 13$$

$$= 0 \leq 3m^2 + 6m$$

Follows since $m \geq 0$.

Therefore, $\lambda n. 10 * n^2 + 3 \in O(\lambda n. n^2)$

Informal big-O notation

- Mathematicians are too lazy to write out the lambdas.
 - $O(n)$ is really short hand for $O(\lambda n. n)$
 - $O(m^2)$ is really short hand for $O(\lambda m. m^2)$
 - This gets confusing when there is a variable clash.
 - It also makes it look like $O(-)$ is a function on numbers -- it's not! It's a function that maps functions to sets.
 - A higher-order function!
 - Alas, so common, we might as well stick to the convention...

Some Useful O-facts

- $\lambda n. f(n) + k \in O(f)$
adding a constant doesn't matter.
- $\lambda n. k * f(n) \in O(f)$
multiplying by a constant doesn't either.
- $\lambda n. n^k + n^c \in O(n^k)$ when $k \geq c$.
highest-degree term wins.

So for instance,

$\lambda n. 4 * n^3 + 12 * n^2 + 782 * n + 42 \in O(?)$

How to prove?

More Useful Facts

- $\lambda n. n^k \gg \lambda n. n^c$ when $k > c$.
bigger exponents grow faster.
- $\lambda n. n \gg \lambda n. \lg_2 n$
logarithms grow slower than linear.
- $\lambda n. \lg_k n \in O(\lambda n. \lg_2 n)$
so dividing by k is not faster than by 2.
- $\lambda n. 2^n \gg \lambda n. n^k$.
exponentials grow faster than polynomials.
- $\lambda n. 3^n \gg \lambda n. 2^n$.
the base for an exponential matters.

Big-O Summary

- Big-O gives takes a function and returns a set of functions.
 - intuitively: those functions that have the same growth, modulo constants.
 - formally: the difference in growth is bounded by a constant.
- Big-O gives us a way to compare algorithms.
 - Captures the informal notion that “constants shouldn't matter” since they will change with technology.
- But we're only half-way there:
 - Need to be able to calculate the worst-case running time (or space) of a program, as a function of the size of its input.

Basic Rules of Thumb

Most Scheme primitives take $O(1)$ time (constant time) when given values:

- `cons`, `car`, `cdr`, `cons?`, `list?`, etc.
 - but not `length`, `append`, `map`, `foldr`, etc.
- `make-foo`, `foo-x`, `foo?` for a struct `foo`
- numeric primitives: `+`, `*`, `-`, `<=`, etc.
 - this is not actually true for Scheme's big-numbers.
 - (in reality the time to do an operation like `+` on numbers of magnitude n is $O(\lg n)$.)
 - Warning: `equals?` is *not* constant time on data structures like lists!

Execution Time of Functions

- The worst-case time to evaluate $(f\ e_1\ e_2\ \dots\ e_n)$ is given by:
 - time to evaluate f to a $(\lambda (x_1 \dots x_n) e)$
 - plus the time to evaluate e_1 to a value v_1
 - plus the time to evaluate e_2 to a value v_2
 - ...
 - plus the time to evaluate e_n to a value v_n
 - plus the time to evaluate e after substituting the v 's for the x 's.
 - you can assume the substitution is $O(1)$.

Time To Evaluate If

- The worst-case time to evaluate $(if\ e_1\ e_2\ e_3)$ is
 - $T(e_1) + T(e_2)$ if we know e_1 evaluates to true
 - $T(e_1) + T(e_3)$ if we know e_1 evaluates to false
 - $T(e_1) + \max(T(e_2), T(e_3))$ otherwise.
 - i.e., we always evaluate e_1 so we have to account for that cost.
 - we either evaluate e_2 or e_3 , but in general, we won't know whether e_1 will evaluate to true or false.
 - adding the “larger” of the two times is conservative and gives us an upper bound.

Putting Ideas to Work

```
(define (append x y)
  (if (empty? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Want to define T_{append} as a function of the size of the inputs (i.e., lengths of x and y).

Setting up Recurrences

```
(define (append x y)
  (if (empty? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

$$T_{\text{append}}(0,m) = T(\text{empty?}) = c_1$$

$$T_{\text{append}}(n+1,m) = T(\text{empty?}) + T(\text{car}) + T(\text{cons}) + T(\text{cdr}) + T_{\text{append}}(n,m) = c_1 + c_2 + c_3 + c_4 + T_{\text{append}}(n,m) = k + T_{\text{append}}(n,m)$$

Need to Solve Recurrence

$$T_{\text{append}}(0,m) = c_1$$

$$T_{\text{append}}(n+1,m) = k + T_{\text{append}}(n,m)$$

$$T_{\text{append}}(n,m) = \underbrace{k + k + k + k + k + \dots + k}_{n} + c_1$$

$$T_{\text{append}}(n,m) = k*n + c_1$$

- Note that is only dependent on the length of x and not the length of y .
- The function $\lambda n.(k*n + c_1) \in O(n)$.
- So we say T_{append} is linear in the size of x .

Another Example

```
(define (reverse x)
  (if (empty? x)
      empty
      (append (reverse (cdr x))
              (cons x empty))))
```

$$T_{\text{reverse}}(0) = T(\text{empty?}) + T(\text{empty}) = q$$

$$T_{\text{reverse}}(n+1) = T(\text{empty?}) + T(\text{cons}) + T_{\text{reverse}}(n) + T_{\text{append}}(n,1) = r + T_{\text{reverse}}(n) + T_{\text{append}}(n,1)$$

Solving the Recurrence

$$T_{\text{reverse}}(0) = q$$

$$T_{\text{reverse}}(n+1) = r + T_{\text{reverse}}(n) + T_{\text{append}}(n,1)$$

$$= r + T_{\text{reverse}}(n) + (k*n + c)$$

$$= s + k*n + T_{\text{reverse}}(n)$$

$$T_{\text{reverse}}(n) = s + k*n + T_{\text{reverse}}(n-1)$$

$$= s + k*n + s + k*(n-1) + T_{\text{reverse}}(n-2)$$

$$= s + k*n + s + k*(n-1) + s + k*(n-2) + T_{\text{reverse}}(n-3)$$

$$= s*n + k*\sum_{i=1}^n i = s*n + k*(n*(n-1)) / 2$$

$$\in O(n^2)$$

Can we do better?

```
(define (rev-app x y)
  (if (empty? x)
      y
      (rev-app (cdr x)
              (cons (car x) y))))

(define (reverse x) (rev-app x empty))
```

$$T_{\text{reverse}}(n) = q + T_{\text{rev-app}}(n,0)$$

$$T_{\text{rev-app}}(0,m) = c$$

$$T_{\text{rev-app}}(n+1,m) = k + T_{\text{rev-app}}(n,m)$$

$$T_{\text{reverse}}(n) \in O(n)$$

Another Example

```
(define (insert x xs)
  (cond [(empty? xs) x]
        [(> x (car xs))
         (cons (car xs) (insert x (cdr xs)))]
        [else (cons x xs)]))

(define (insert-sort xs) (foldr insert empty xs))
```

$T_{\text{insert}}(1,0) = c$
 $T_{\text{insert}}(1,n+1) = \max(k_1 + T_{\text{insert}}(1,n), k_2) = k_1 + T_{\text{insert}}(1,n)$
 $T_{\text{insert}}(1,n) \in O(n)$
 $T_{\text{insert-sort}}(n) = ???$

Foldr

```
(define (foldr f u xs)
  (if (empty? xs) u
      (f (car xs) (foldr f u (cdr xs)))))

(define (insert-sort xs) (foldr insert empty xs))
```

$T_{\text{foldr-insert-empty}}(0) = c$
 $T_{\text{foldr-insert-empty}}(n+1) = T_{\text{insert}}(1,n) + T_{\text{foldr-insert-empty}}(n)$

$T_{\text{foldr-insert-empty}} \in O(n^2)$
 $T_{\text{insert-sort}} \in O(n^2)$

Merge

```
(define (merge xs ys)
  (cond [(empty? xs) ys]
        [(empty? ys) xs]
        [(<= (car xs) (car ys))
         (cons (car xs) (merge (cdr xs) ys))]
        [else (cons (car ys) (merge xs (cdr ys)))]))
```

$T_{\text{merge}}(0,m) = c1$
 $T_{\text{merge}}(n,0) = c2$
 $T_{\text{merge}}(n+1,m+1) = k + \max(T(n,m+1), T(n+1,m))$

Worst case?
 $T_{\text{merge}} \in O(n)$

Evens and Odds

```
(define (evens xs)
  (cond [(empty? xs) empty]
        [(empty? (cdr xs)) xs]
        [else (cons (car xs) (evens xs))]))

(define (odds xs)
  (if (empty? xs) empty (evens (cdr xs))))
```

$T_{\text{evens}} \in O(n)$
 $T_{\text{odds}} \in O(n)$

Mergesort

```
(define (mergesort xs)
  (cond [(empty? xs) xs]
        [(empty? (cdr xs)) xs]
        [else (merge (mergesort (odds xs))
                      (mergesort (evens xs)))]))
```

$T_{\text{mergesort}}(0) = c1$
 $T_{\text{mergesort}}(1) = c2$
 $T_{\text{mergesort}}(n) = k + 2 * T_{\text{mergesort}}(n/2) + T_{\text{merge}}(n/2, n/2)$
 $= k + q * n + 2 * T_{\text{mergesort}}(n/2)$
 $= k + q * n + 2 * (k + q * (n/2) + T_{\text{mergesort}}(n/4))$
 $= k + q * n + 2 * (k + q * (n/2) + 2 * (k + q * (n/4) + T_{\text{mergesort}}(n/8)))$

This continues for $\lg n$ iterations.
 And we're doing work proportional to n each iteration.
 So $T_{\text{mergesort}} \in O(n)$

Faster?

```
(define-struct (pair first second))
(define empty-pair (make-pair empty empty))
(define (add-and-flip x p)
  (make-pair (cons x (second p)) (first p)))

(define (evens-and-odds xs p)
  (if (empty? xs) p
      (evens-and-odds (cdr xs)
                      (add-and-flip (car x) p))))

(define (mergesort xs)
  (cond [(empty? xs) xs]
        [(empty? (cdr xs)) xs]
        [else (let ([p (evens-and-odds xs empty-pair)])
                  (merge (mergesort (first p))
                          (mergesort (second p))))]))
```

Another Comparison

```
(define (union-1 xs ys)
  (if (< (length xs) (length ys))
      (foldr insert ys xs)
      (foldr insert xs ys)))

(define (union-2 xs ys)
  (merge (mergesort xs) (mergesort ys)))
```

$T_{\text{union-1}}(n,m) \in O(???)$

$T_{\text{union-2}}(n,m) \in O(???)$

Lookup

```
(define (list-member? x xs)
  (cond ([empty? xs] false)
        [(= x (car xs)) true]
        [else (list-member? x (cdr xs))]))

(define (tree-member? x t)
  (cond ([empty? t] false)
        [(= x (node-elt t)) true]
        [< x (node-elt t)]
          (tree-member? x (node-left t))
        [else (tree-member? x (node-right t))]))
```

Balanced Trees

- If a tree is unbalanced, then it could degenerate to a list, so we are only reducing the size of the input by 1.
 - hence $O(n)$
- If a tree is balanced, then on each recursive call, we are cutting the tree in half.
 - hence $O(\lg n)$

One More Example

```
(define (power n m)
  (if (= m 0) 1 (* n (power n (- m 1)))))

(define (power-2 n m)
  (cond ([= m 0] 1)
        [(even? m) (power-2 (* n n) (/ n 2))]
        [else (* n (power-2 n (- m 1)))]))
```

Assuming `even?`, `*`, `/`, `-` are $O(1)$.

Subtlety: the running time is a function of the *magnitude* m (but the size of m is assumed to be a constant.)

Basic Recurrence Patterns

- $T(n) = c + T(n-1) \in O(n)$
- $T(n) = k*n + T(n-1) \in O(n^2)$
- $T(n) = c + T(n/2) \in O(\lg n)$
- $T(n) = k*n + T(n/2) \in O(n)$

Next time: how do you prove these?