Efficiency

CS135 Lecture 09

Big-*O* in practice

Name	Big-O Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
“n log n”	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

In this lecture, we will look at specific examples of linear, quadratic and exponential functions.

L09.1 Linear

Linear time example - `countdown`

In CS135, efficiency is usually expressed in terms of the length of a consumed list or in terms of the value of a consumed natural number *n*.

```
(define (countdown n)
  (cond [(zero? n) empty]
        [else (cons n (countdown (sub1 n)))]))
```

`(countdown 0)` ⇒ 4 steps
`(countdown 1)` ⇒ 10 steps
`(countdown 10)` ⇒ 64 steps
`(countdown 100)` ⇒ 604 steps

Number of steps = $f(n) = 4 + 6n = O(n)$

Linear time example - `last`

Generally, any function written with the Rules for Recursion (second version) will be linear in the length of the list it consumes, or the value of *n* it consumes, as long as it only uses built-in functions that are constant time.

```
(define (last lst)
  (cond [(empty? (rest lst)) (first lst)]
        [else (last (rest lst))]))
```

`(last (list 1))` ⇒ 6 steps
`(last (list 1 2))` ⇒ 12 steps
`(last (list 1 2 3))` ⇒ 18 steps
`(last (list 1 2 3 4))` ⇒ 24 steps

$f(n) = 6n = O(n)$

Linear time example - **`attach`**

```
;; Add an element to the end of a list
;; attach: Any (listof Any) -> (listof Any)
(define (attach element lst)
  (cond [(empty? lst) (cons element empty)]
        [else (cons (first lst) (attach element (rest lst)))]))
```

`(attach -1 empty)` ⇒ 5 steps
`(attach -1 (list 1))` ⇒ 12 steps
`(attach -1 (list 1 2))` ⇒ 19 steps
`(attach -1 (list 1 2 3))` ⇒ 26 steps
`(attach -1 (list 1 2 3 4))` ⇒ 33 steps

$f(n) = 5 + 7n = O(n)$

Appending a list to a list (**`appnd`**) - examples

```
(appnd (list 1 2 3) (list 'a 'b 'c)) ⇒ (list 1 2 3 'a 'b 'c)

(appnd (list 1 2 3) empty) ⇒ (list 1 2 3)

(appnd empty (list 'a 'b 'c)) ⇒ (list 'a 'b 'c)

(appnd empty empty) ⇒ empty
```

Purpose, contract and header

```
;; append the second list to the first list
;; appnd: (listof Any) (listof Any) -> (listof Any)
(define (appnd lst0 lst1) ...)
```

This function is consistent with the 2nd version of the rule of recursion as long as we only recurse on one of the two lists.

Which one do we recurse on?

Thinking about a list function

1. **What should the function produce in the base case?**

The first list is empty. Produce the second list.

2. **What should the function do to the first element in a non-empty list?**

It becomes the first element in the appended list.

3. **What should applying the function to the rest of the list produce?**

The second list appended to the rest of the first list

4. **How should the function combine #2 and #3 to produce the answer for the entire list?**

Put the first element on the front of the appended list.

Appending a list to a list

```
;; append the second list to the first list
;; appnd: (listof Any) (listof Any) -> (listof Any)
(define (appnd lst0 lst1)
  (cond [(empty? lst0) lst1]
        [else (cons (first lst0) (appnd (rest lst0) lst1))]))
```

This function is $O(n)$, where n is the length of the first list:

`(appnd (list 1 2 3) (list 'a 'b 'c))` ⇒ 25 steps

`(appnd (list 1 2 3) empty)` ⇒ 25 steps

DrRacket has a built in function **`append`** that you can now use. Like **`length`**, it requires a single step in the stepper, but you should assume it's secretly $O(n)$.

L09.2 Quadratic

Sorting a list in ascending order

```
(check-expect (sort empty) empty)

(check-expect (sort (list 100)) (list 100))

(check-expect
  (sort (list 5 -10 1 14 3 1 8 9 6 -12))
  (list -12 -10 1 1 3 5 6 8 9 14))
```

Any other important test cases to consider?

Insertion Sort - an algorithm for sorting

```
(sort (list 5 -10 1 14 3 1 8 9 6 -12))
```

An insertion sort has a natural recursive definition

1. Sort the rest of the list

```
(sort (list -10 1 14 3 1 8 9 6 -12))
    ⇒ (list -12 -10 1 1 3 6 8 9 14)
```

2. Insert the first into its proper place

```
(insert 5 (list -12 -10 1 1 3 6 8 9 14))
    ⇒ (list -12 -10 1 1 3 5 6 8 9 14)
```

What about the empty list?

We already know how to insert into an ordered list

```
;; Insert a number into an ordered list
;; insert: Num (listof Num) -> (listof Num)
;; Requires: values must be ascending order
(define (insert n lst)
  (cond [(empty? lst) (cons n empty)]
        [(< n (first lst)) (cons n lst)]
        [else (cons (first lst) (insert n (rest lst)))]))
```

It’s okay (and even encouraged) for you to reuse code we give you in class, tutorials, and assignments, as well as your own answers to previous assignment questions.

Insertion sort

```
;; Sort a list of numbers in ascending order
;; sort: (listof Num) -> (listof Num)
(define (sort lst)
  (cond [(empty? lst) empty]
        [else (insert (first lst) (sort (rest lst)))]))
```

Does this follow the Rules for Recursion?

Counting substitution steps

What is the “worst case” for this algorithm, i.e., in what situation will it take the most steps?

Let’s think about boundary cases, e.g., already sorted vs. ordered in reverse.

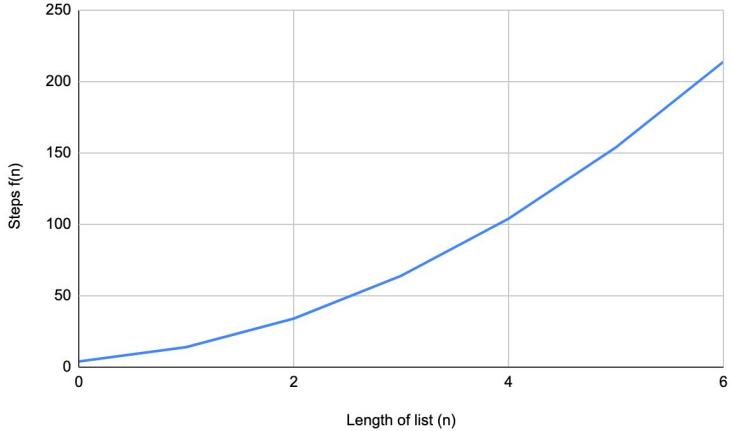
`(sort (list 1 2 3 4 5))` ⇒ 66 steps
`(sort (list 5 4 3 2 1))` ⇒ 154 steps

When we think about execution time, we typically think about the “worst case”.

Here the worst case appears to be the case where the list is ordered in reverse.

Counting substitution steps

`(sort empty)` ⇒ 4 steps
`(sort (list 1))` ⇒ 14 steps
`(sort (list 2 1))` ⇒ 34 steps
`(sort (list 3 2 1))` ⇒ 64 steps
`(sort (list 4 3 2 1))` ⇒ 104 steps
`(sort (list 5 4 3 2 1))` ⇒ 154 steps
`(sort (list 6 5 4 3 2 1))` ⇒ 214 steps



$f(n) = 5n^2+5n+4 = O(n^2)$

Execution time is quadratic.

Recursion in helper functions

We have now seen many examples of “helper functions”, which is a function that helps another function solve a problem. In the case of insertion sort, the function `insert` acts as a helper function, even though it is itself recursive.

When a problem seems complicated to solve, we can often break the problem down into smaller problems, solve them, and then combine the solutions.

Since we want to encourage you to write helper functions, we don’t penalize you on assignments if you don’t provide provide a purpose, or contract for helper functions (unless it’s a helper function we specifically ask you to write). We still require full test coverage for all code you submit.

Quadratic time functions

If you have a linear time function being used as a helper for function that follows the Rules for Recursion, it can generally be written so that the overall function is quadratic in efficiency.

This is especially true if the helper is being used to filter or transform the result of a recursive call on the rest of the list (like we are doing in `sort`).

Reversing a list

```
(define (attach element lst)
  (cond [(empty? lst) (cons element empty)]
        [else (cons (first lst) (attach element (rest lst)))]))

(define (rev lst)
  (cond [(empty? lst) empty]
        [else (attach (first lst) (rev (rest lst)))]))
```

Steps to reverse a list of length $f(n) = 3.5n^2 + 6.5n + 4 = O(n^2)$

There's a built-in **`reverse`**, but you can't use it yet. Later we will look at a linear method for reversing a list. You can use the built-in **`reverse`** after that.

L09.3 Exponential

How do we find the largest element in a non-empty list?

1. **What should the function produce in the base case?**

If the rest of the list is empty, produce the only element in the list is the max

2. **What should the function do to the first element in a non-empty list?**

If the rest of the list is not empty, find the largest of the rest of the list and compare them

3. **What should applying the function to the rest of the list produce?**

The largest in the rest of the list.

4. **How should the function combine #2 and #3 to produce the answer for the entire list?**

Compare them and produce the largest.

Finding the largest number in a non-empty list.

```
(define (largest lst)
  (cond ((empty? (rest lst)) (first lst))
        ((> (first lst) (largest (rest lst))) (first lst))
        (else (largest (rest lst)))))
```

`(largest (list 1))` ⇒ 6 steps
`(largest (list 2 1))` ⇒ 15 steps
`(largest (list 3 2 1))` ⇒ 24 steps
`(largest (list 4 3 2 1))` ⇒ 33 steps
`(largest (list 5 4 3 2 1))` ⇒ 42 steps
`(largest (list 6 5 4 3 2 1))` ⇒ 51 steps

$f(n) = 9n - 3 = O(n)$ - Linear (but is this the worst case?)

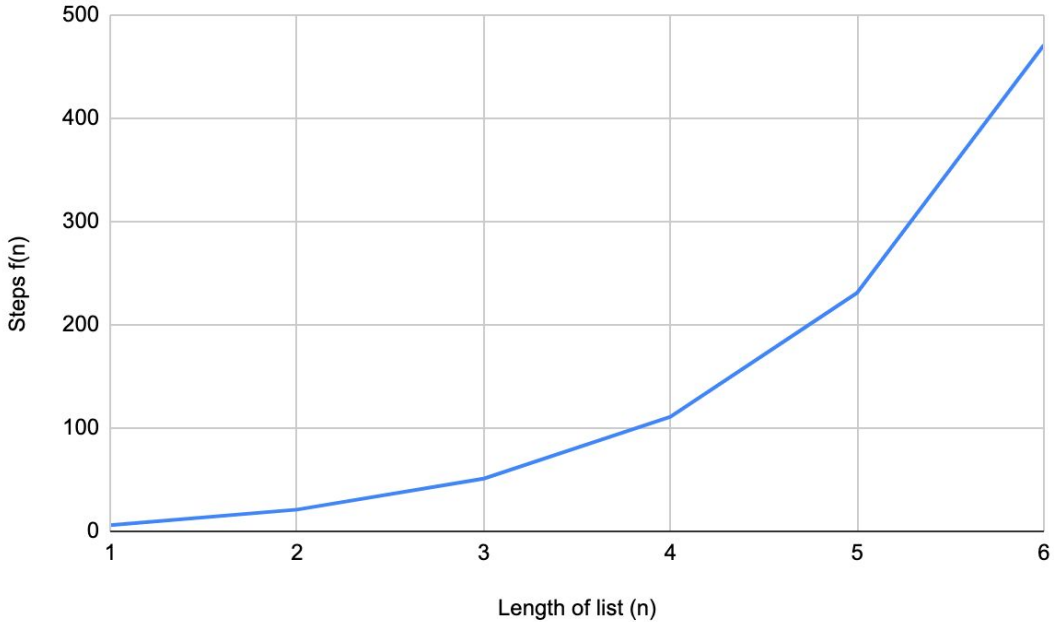
Finding the largest number in a non-empty list.

```
(define (largest lst)
  (cond ((empty? (rest lst)) (first lst))
        ((> (first lst) (largest (rest lst))) (first lst))
        (else (largest (rest lst)))))
```

`(largest (list 1))` ⇒ 6 steps
`(largest (list 1 2))` ⇒ 21 steps
`(largest (list 1 2 3))` ⇒ 51 steps
`(largest (list 1 2 3 4))` ⇒ 111 steps
`(largest (list 1 2 3 4 5))` ⇒ 231 steps
`(largest (list 1 2 3 4 5 6))` ⇒ 471 steps

The number of steps roughly doubles with the length of the list (*n*)

Exponential “blow up”

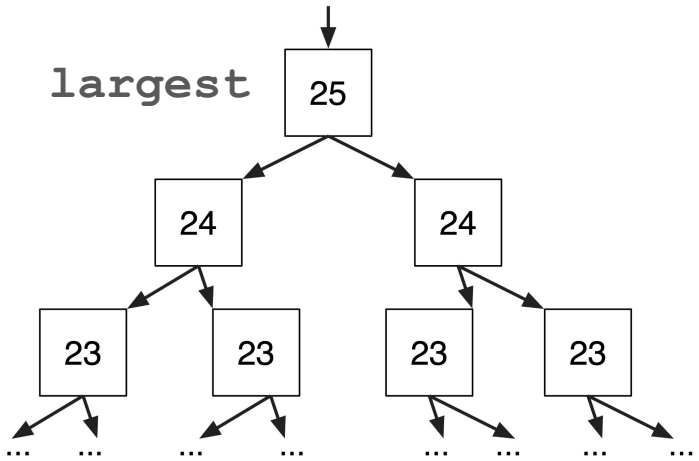


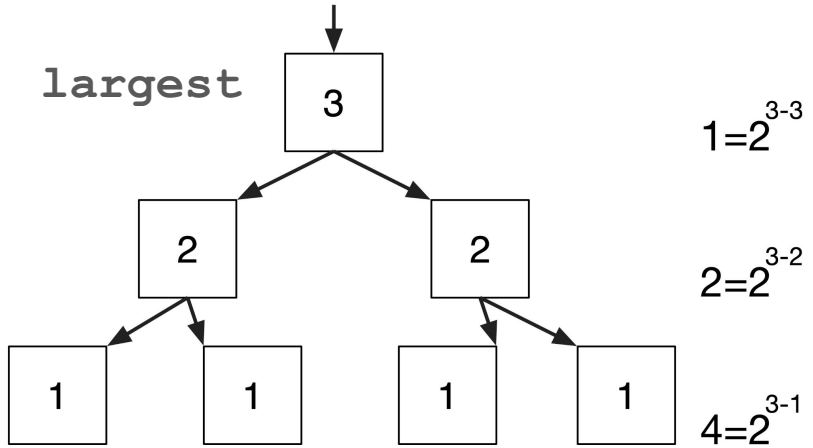
The number of substitution steps roughly doubles with the length of the list.

Don’t even try this `largest` on a list longer than 25.

If initial application is on a list of length 25, there are two recursive applications on the rest of this list, which is of length 24. Each of those makes two recursive applications, and so on.

Exponential “blow up”





`largest` can make $O(2^n)$ recursive applications, i.e., $O(2^n)$ steps.

In general, avoid multiple recursive calls on the rest of a list

Linear-time `largest`

```
;; produce the larger of a pair of numbers
;; larger: Num Num -> Num
(define (larger a b)
  (cond [(> a b) a]
        [else b]))

;; produce the largest of a non-empty list of numbers
;; largest: (listof Num) -> Num
(define (largest lst)
  (cond [(empty? (rest lst)) (first lst)]
        [else (larger (first lst) (largest (rest lst)))]))
```

(Of course, the built-in function ***`max`*** *could be used instead of* ***`larger`****.)*

Lecture 9 Summary

All you need to know about efficiency (for now anyway)

1. Most built-in functions and functions that don't use recursion are $O(1)$, i.e., "constant time".
2. Efficiency (or "time") is measured by counting substitution steps. The number of steps will be expressed in terms of a natural number, such as the length of a list, n.
3. If the number of steps is of the form $a + bn$, efficiency is $O(n)$ or "linear".
4. While they only take one step in the stepper, we want you to think of some built-in functions as linear: **`length`**, **`append`**, and **`reverse`**
5. If the number of steps is of the form $a + bn + cn^2$, efficiency is $O(n^2)$ or "quadratic".
6. The number of steps can be "exponential" in n, e.g., $O(2^n)$. Large n will cause "exponential blowup". The function will be unusably slow for larger n.

L09: You should know

- How to append one list to another.
- How to sort elements with insertion sort.
- The difference between linear, quadratic, and exponential time functions.
- How to recognize and remedy exponential time “blow up”.
- **… and not to worry too much about efficiency.**

Some students worry that answers to assignment and exam questions are not “efficient enough”. If you are worried about efficiency, try your code on lists of a few hundred elements. If it runs in less than ten seconds, it’s efficient enough.

In class, we might talk about efficiency in more detail, but for your own code, you don’t have to worry about it, beyond avoiding exponential blow up.

L09: Allowed constructs

Newly allowed constructs:
`append`

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and asin atan boolean? check-expect
check-within cond cons cons? cos define e else empty empty?
even? exp expt false first inexact? integer? length list
list? log max min not number? odd? or pi quotient rational?
remainder rest second sin sqr sqrt sub1 symbol? symbol=? tan
third true zero?
listof Any anyof Bool Int Nat Num Rat Sym
```

Recursion **must** follow the Rules for Recursion (second version)