Computational Efficiency

1 December 2020
Assignment 5 is out!
Recall: Generative recursion
**Structural recursion:**

The *structure* of the data determines the structure of the recursive function.

**Generative recursion.**

The recursive cases are *generated* based on the problem to be solved.

Generative recursion can be harder because neither the base nor recursive cases follow from the data definitions.
If we make a list smaller by recursively using **first** and **rest** until it’s eventually empty, would that be structural or generative recursion?
If we make a list smaller by recursively splitting it in half until eventually it’s empty, would that be structural or generative recursion?
There’s one more example of generative recursion I want to show you, but before we get there, we need to step back a bit, returning to our discussion of computational efficiency.
A familiar problem
Max of a list

Implement the function `max-item` that returns the biggest number in a list of numbers.
Max of a list

Implement the function $\text{max-item}$ that returns the biggest number in a list of numbers.

**Data:** $[\text{List-of Number}]$, obviously

**Signature:**

$$;; [\text{List-of Number}] \rightarrow \text{Number}$$
Max of a list

Implement the function `max-item` that returns the biggest number in a list of numbers.

Data: `[List-of Number]`, obviously

Signature:

```ml
;; [List-of Number] -> Number
```

Examples:

```
(max-item '(2 7 5)) → 7
```
Max of a list

Implement the function `max-item` that returns the biggest number in a list of numbers.

**Data:** `[List-of Number]`, obviously

**Signature:**

```
;; [List-of Number] -> Number
```

**Examples:**

```
(max-item '(2 7 5)) → 7
(max-item '()) → ?
```
Max of a list

Implement the function \texttt{max-item} that returns the biggest number in a \texttt{non-empty} list of numbers.

\textbf{Data:} \texttt{NonEmptyListOfNums}

\begin{verbatim}
;; A \texttt{NonEmptyListOfNums} is either:
;; - (cons Number '())
;; - (cons Number \texttt{NonEmptyListOfNums})
\end{verbatim}

\textbf{Signature:}

\begin{verbatim}
;; \texttt{NonEmptyListOfNums} \rightarrow \texttt{Number}
\end{verbatim}
Max of a list

Implement the function `max-item` that returns the biggest number in a non-empty list of numbers.

Data: `NonEmptyListOfNums`

```plaintext
;; A NonEmptyListOfNums is either:
;; - (cons Number '())
;; - (cons Number NonEmptyListOfNums)
```

Signature:

```plaintext
;; NonEmptyListOfNums -> Number
```

Examples:

```plaintext
(max-item '(2 7 5)) → 7
(max-item '(2)) → 2
```
Implementation

No existing functions for non-empty lists, so start with the template:

```scheme
;; A NonEmptyListOfNums is either:
;; - (cons Number '())
;; - (cons Number NonEmptyListOfNums)

(define (max-item nel)
  (cond [(empty? (rest nel))
         (... (first nel) ...)]
        [else
         (... (first nel) ...
              (max-item (rest nel))))])
```

```scheme
```
```scheme
```
Implementation complete

(define (max-item nel)
  (cond [(empty? (rest nel))
               (first nel)]
         [else
          (cond [>(first nel)
                     (max-item (rest nel))]
                [else
                 (max-item (rest nel))])))
Test

(check-expect (max-item '(2)) 2)

works fine 😊

(check-expect (max-item '(1 2 3 4 5 6 7 8 9 10)) 10)

works fine 😊

(check-expect (max-item '( 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)) 30)

answer never appears 😒
The speed of max-item

Somewhere around 20 items, the **max-item** function starts to take way too long.

Even if you buy a computer that’s 10 times faster, the problem shows up with around 23 items…

So, how do we think about how long a program takes to run?
Steps of computation
How long does

(+ 1 (* 6 7))

take to execute?
How long does

\((+ 1 (* 6 7))\)

take to execute?

Computer speeds differ in “real time”, but we can count steps:

\[
(+ 1 (* 6 7)) \\
\rightarrow (+ 1 42) \\
\rightarrow 43
\]

So, evaluation takes two steps
Steps for max-item with 1 element

How long does this expression take?

(max-item '(2))
Steps for `max-item` with 1 element

How long does this expression take?

```
(max-item '(2))
```

```
(max-item '(2))
→ (cond [(empty? (rest '(2))) (first '(2))] ...) 
→ (cond [(empty? '()) (first '(2))] ...) 
→ (cond [#true (first '(2))] ...) 
→ (first '(2)) 
→ 2 
```

5 steps – and any list with one item will take five steps.

[+ 1 if you use Intermediate Student with Lambda…]
Steps for max-item with 2 elements

How long does this expression take?

(max-item '(2 1))
Steps for \texttt{max-item} with 2 elements

How long does this expression take?

\[
\texttt{(max-item '(2 1))}
\]

\[
\texttt{(max-item '(2 1))}
\]

\[
\rightarrow (\texttt{cond} [(\texttt{empty?} (\texttt{rest} '(2 1))) (\texttt{first} '(2 1))] [\texttt{else} ...])
\]

\[
\rightarrow (\texttt{cond} [(\texttt{empty?} '(1)) (\texttt{first} '(2 1))] [\texttt{else} ...])
\]

\[
\rightarrow (\texttt{cond} [\#\texttt{false} (\texttt{first} '(2 1))] [\texttt{else} ...])
\]

\[
\rightarrow (\texttt{cond} [\texttt{else} (\texttt{cond} [(> (\texttt{first} '(2 1)) ...) ...] [\texttt{else} ...]))]
\]

\[
\rightarrow (\texttt{cond} [(> (\texttt{first} '(2 1))
\begin{verbatim}
(\texttt{max-item (\texttt{rest} '(2 1)))) ...
\end{verbatim}
[\texttt{else} ...])
\]

\[
\rightarrow (\texttt{cond} [(> 2 (\texttt{max-item (\texttt{rest} '(2 1)))) ...) [\texttt{else} ...])
\]

\[
\rightarrow (\texttt{cond} [(> 2 (\texttt{max-item (\texttt{rest} '(2 1)))) ...) [\texttt{else} ...])
\]

\[
\rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow ...
\]

\[
\rightarrow (\texttt{cond} [(> 2 1) (\texttt{first} '(2 1))] [\texttt{else} ...])
\]

\[
\rightarrow (\texttt{first} '(2 1))
\]

\[
\rightarrow 2
\]
Steps for `max-item` with 2 elements

How long does this expression take?

```
(max-item '(2 1))
```

```
(max-item '(2 1))
→ (cond [(empty? (rest '(2 1))) (first '(2 1))] [else ...])
→ (cond [(empty? '(1)) (first '(2 1))] [else ...])
→ (cond [#false (first '(2 1))] [else ...])
→ (cond [else (cond [(> (first '(2 1)) ...) ...] [else ...]))]
→ (cond [(> (first '(2 1)) (max-item (rest '(2 1)))) ...) [else ...])
→ (cond [(> 2 (max-item (rest '(2 1)))) ...) [else ...])
→ (cond [(> 2 (max-item '(1))) ...) [else ...])
→ ... → ... → ... → ...
→ (cond [(> 2 1) (first '(2 1))] [else ...])
→ (first '(2 1))
→ 2
```

14 steps – where 5 come from the recursive call.
Steps for max-item with 2 elements

How long does this expression take?

(max-item '(2 1))

→ (cond [(empty? (rest '(2 1))) (first '(2 1))] [else ...])
→ (cond [(empty? '(1)) (first '(2 1))] [else ...])
→ (cond [#false (first '(2 1))] [else ...])
→ (cond [else (cond [(> (first '(2 1)) ...) ...] [else ...])])
→ (cond [(> (first '(2 1))

  (max-item (rest '(2 1)))) ...] [else ...])
→ (cond [(> 2 (max-item (rest '(2 1)))) ...] [else ...])
→ (cond [(> 2 (max-item '(1))) ...] [else ...])
→ ... → ... → ... → ...
→ (cond [(> 2 1) (first '(2 1))] [else ...])
→ (first '(2 1))
→ 2

14 steps – where 5 come from the recursive call.

Are all 2-element lists the same?
Steps for max-item with 2 elements

(max-item '(1 2))
→ (cond [(empty? (rest '(1 2))) (first '(1 2))] [else ...])
→ (cond [(empty? '(2)) (first '(1 2))] [else ...])
→ (cond #[false (first '(1 2))] [else ...])
→ (cond [else (cond [(> (first '(1 2)) ...) ...] [else ...])])
→ (cond [(> (first '(1 2))
  (max-item (rest '(1 2))))) ...]
  [else ...])
→ (cond [(> 1 (max-item (rest '(1 2)))) ...] [else ...])
→ (cond [(> 1 (max-item '(2))) ...] [else ...])
→ ... → ... → ... → ...
→ (cond [(> 1 2) ...] [else ...])
→ (cond [else (max-item (rest '(1 2)))]
→ (max-item (rest '(1 2)))
→ (max-item '(2))
→ ... → ... → ... → ...
→ 2

20 steps – where 10 come from two recursive calls.
Steps for max-item and $n$ elements

In the worst case, the step count $T$ for an $n$-element list passed to max-item is

$$T(n) = 10 + 2T(n-1)$$
Steps for \texttt{max-item} and \( n \) elements

In the worst case, the step count \( T \) for an \( n \)-element list passed to \texttt{max-item} is

\[ T(n) = 10 + 2T(n-1) \]

\[
\begin{align*}
T(1) &= 5 \\
T(2) &= 10 + 2T(1) = 20 \\
T(3) &= 10 + 2T(2) = 50 \\
T(4) &= 10 + 2T(3) = 110 \\
T(5) &= 10 + 2T(4) = 230 \\
&\ldots
\end{align*}
\]
Steps for max-item and n elements

In the worst case, the step count $T$ for an $n$-element list passed to max-item is

$$T(n) = 10 + 2T(n-1)$$

- $T(1) = 5$
- $T(2) = 10 + 2T(1) = 20$
- $T(3) = 10 + 2T(2) = 50$
- $T(4) = 10 + 2T(3) = 110$
- $T(5) = 10 + 2T(4) = 230$

... 

In general, $T(n) > 2^n$

Note that $2^{30}$ is 1,073,741,824 – which is why our last test never produced a result.
Removing redundant computation
Repairing **max-item**

In the case of **max-item**, the problem is easily fixed with **local**:

```scheme
(define (max-item nel)
  (cond [(empty? (rest nel))
         (first nel)]
       [else
        (local [(define r (max-item (rest nel)))]
          (cond [(> (first nel) r)
                 (first nel)]
                [else r]))]))
```

With this definition, there’s only one recursive call. 

`(max-item '(1 2))` takes 17 steps.
Steps for new max-item and n elements

In the worst case, now, the step count $T$ for an $n$-element list passed to max-item is

$$T(n) = 12 + T(n - 1)$$

$T(1) = 5$

$T(2) = 12 + T(1) = 17$

$T(3) = 12 + T(2) = 29$

$T(4) = 12 + T(3) = 41$

$T(5) = 12 + T(4) = 53$

$\ldots$
Steps for new \texttt{max-item} and $n$ elements

In the worst case, now, the step count $T$ for an $n$-element list passed to \texttt{max-item} is

$$T(n) = 12 + T(n - 1)$$

So our last test takes only 343 steps
Sorting: Insertion sort revisited
Back on October 6, we implemented *insertion sort* for a list of images, using standard structural recursion:

We can sort a list by inserting the first element into the right spot in result of sorting the rest of the list.

The empty list is already sorted!

To insert an element into the right spot in a sorted list, we stop as soon as we find a larger element, and we insert it before that.
If we make a list smaller by recursively using \texttt{first} and \texttt{rest} until it’s eventually empty, would that be structural or generative recursion?

\textbf{That’s what insertion sort does; it’s structural recursion!}
Insertion sort

;; [List-of Number] -> [List-of Number]
(define (insertion-sort l)
  (cond [(empty? l) '()]
        [(cons? l)
         (insert (first l)
                 (insertion-sort (rest l))))))
### Insertion sort

```scheme
;; [List-of Number] -> [List-of Number]
(define (insertion-sort l)
  (cond [(empty? l) '()] [(cons? l)
    (insert (first l)
       (insertion-sort (rest l)))]))
```

How long does it take to sort a list of $n$ numbers?
Insertion sort

;;; [List-of Number] -> [List-of Number]
(define (insertion-sort l)
  (cond [(empty? l) '()]  
        [(cons? l) 
           (insert (first l)  
                    (insertion-sort (rest l)))]))

How long does it take to sort a list of \( n \) numbers?

We only have one recursive call to \texttt{insertion-sort}, so it doesn’t have the same problem as before.
Insertion sort

...but what about \textbf{insert}?

$$\text{;; [List-of Number]} \rightarrow \text{[List-of Number]}$$
\begin{verbatim}
(define (insertion-sort l)
  (cond [(empty? l) '()] [(cons? l)
     (insert (first l)
      (insertion-sort (rest l)))]))
\end{verbatim}

$$\text{;; Number [List-of Number]} \rightarrow \text{[List-of Number]}$$
\begin{verbatim}
(define (insert n l)
  (cond [(empty? l) (list n)] [(cons? l)
   (if (< n (first l))
    (cons n l)
    (cons (first l) (insert n (rest l))))]))
\end{verbatim}
Insertion sort

...but what about insert?

;;; [List-of Number] -> [List-of Number]
(define (insertion-sort l)
  (cond [(empty? l) '()]
    [(cons? l)
      (insert (first l)
        (insertion-sort (rest l))))])

;;; Number [List-of Number] -> [List-of Number]
(define (insert n l)
  (cond [(empty? l) (list n)]
    [(cons? l)
      (if (< n (first l))
        (cons n l)
        (cons (first l) (insert n (rest l))))]))
**insert** time

**insert** is like the repaired **max-item**:

```lisp
;; Number [List-of Number] -> [List-of Number]
(define (insert n l)
  (cond [(empty? l) (list n)]
        [(cons? l)
         (if (< n (first l))
             (cons n l)
             (cons (first l) (insert n (rest l))))]))
```

In the worst case, **inserting** into a list of size \( n \) takes \( k_1 + k_2n \), where \( k_1 \) and \( k_2 \) stand for some constants.
insertion-sort time

Given that the time for insert is \( k_1 + k_2n \)...

\[
\text{;; [List-of Number] -> [List-of Number]}
\]
\[
\text{(define (insertion-sort l)}
\]
\[
\text{(cond [(empty? l) '()]
\text{[(cons? l)
\text{ (insert (first l)
\text{ (insertion-sort (rest l)))])])})
\]

The time for insertion-sort is defined by

\[
T(0) = k_3
\]
\[
T(n) = k_4 + T(n-1) + k_1 + k_2n
\]
**insertion-sort time**

\[ T(0) = k_3 \]
\[ T(n) = k_4 + T(n-1) + k_1 + k_2n \]

Even if each \( k \) were only 1:

\[ T(0) = 1 \]
\[ T(1) = 4 \]
\[ T(2) = 8 \]
\[ T(3) = 13 \]
\[ T(4) = 19 \]

... 

In the long run, \( T(n) \) is a lot like \( n^2 \).

This is a lot better than \( 2^n \) – but sorting a list of 10,000 items takes more than 100,000,000 steps.
The [List-of X] template leads to the insertion sort algorithm – it’s structural recursion.

But it’s not practical for large lists.

Generative recursion to the rescue!

Other important sorting algorithms include merge sort and quicksort
Sorting: Quicksort
A more efficient approach, devised in 1959 by Tony Hoare, is called **quicksort**.

This is a kind of “divide and conquer” algorithm:

- Divide the problem into smaller subproblems,
- Recursively solve each one, and
- Combine the solutions to solve the original problem.

In particular, quicksort sorts a list by first choosing a “pivot” element from the list, then sorting the subproblems of all elements less than the pivot and all elements greater than the pivot.
If we make a list smaller by recursively splitting it in half until eventually it’s empty, would that be structural or generative recursion?

That’s (almost) what quicksort does; it’s generative recursion!
Given a list, we use the pivot to create two new lists: \( l_1 \) comprises every value less than the pivot, and \( l_2 \) comprises every value greater than the pivot.

Suppose you have \((\text{list } 9 \ 4 \ 15 \ 2 \ 12 \ 20)\), and your pivot is 9.

What are your output lists, \( l_1 \) and \( l_2 \)?
If the input is

(list 9 4 15 2 12 20)

and the pivot is 9, then the subproblems are:

(list 4 2)
(list 15 12 20).

Recursively sorting the two subproblem lists gives:

(list 2 4)
(list 12 15 20)

and it’s now simple to combine them with the pivot:

(append (list 2 4)
         (list 9)
         (list 12 15 20))
If we assume the input list `lon` is in random order, the easiest pivot to select is `(first lon)`.

A function that tests whether another item is less than the pivot is

\[
\text{lambda } (x) \ (\lt \ x \ (\text{first lon}))
\]

So we can use the built-in higher-order function `filter` to get the first subproblem:

\[
(\text{filter } (\text{lambda } (x) \ (\lt \ x \ (\text{first lon}))) \ \text{lon})
\]

And a similar expression will find the second subproblem, consisting of items greater than the pivot.
;; [List-of Number] -> [List-of Number]
;;  Sort lon in non-decreasing order

(check-expect (quick-sort '()) '())
(check-expect (quick-sort '(1 2 3)) '(1 2 3))
(check-expect (quick-sort '(3 2 1 4 0)) '(0 1 2 3 4))

(define (quick-sort lon)
  (if (empty? lon)
    '()
    (local [((define pivot (first lon))
             (define less
              (filter (lambda (x) (< x pivot))
              (rest lon)))
             (define more
              (filter (lambda (x) (>= x pivot))
              (rest lon)))]
      (append (quick-sort less)
              (list pivot)
              (quick-sort more))))))
How do we know that quick-sort always terminates?

**Base case:** (empty? lon)

**Reduction step:**

(filter (lambda (x) (< x pivot))
  (rest lon))

(filter (lambda (x) (>= x pivot))
  (rest lon))

**Argument that repeated application of reduction step will eventually reach the base case:**

Every time we are removing one element from lon, since neither of the new subproblems includes the (first instance of the) pivot number.

This wouldn’t have been the case if we’d mistakenly written

(filter (lambda (x) (< x pivot)) lon)
When we count the steps for evaluating quick-sort, in the long run (i.e., as the size of the list increases), it’s a lot like $n \log_2 n$.

That is, sorting a list of 10,000 items takes something like 100,000 steps (which is 1,000 times faster than insertion-sort!)
We can contrast the behavior of an algorithm in the average case vs the worst case.

In the *average* case, quicksort is much better than insertion sort, but in the *worst* case it’s similar:

Intuitively, quicksort works best when the two recursive function applications are on arguments about the same size.

When one recursive function application is always on an empty list – as is the case when the input is already sorted – the pattern of recursion is similar to the worst case of insertion sort, and the number of steps is roughly proportional to the square of the length of the list.
In the teaching languages, the built-in function `quicksort` (no hyphen) consumes two arguments: a list and a comparison function.

```lisp
> (quicksort '(1 5 2 4 3) <)
'(1 2 3 4 5)

> (quicksort '(1 5 2 4 3) >)
'(5 4 3 2 1)
```
The cost of computation

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

Practical points:

- Use `local` to avoid redundant computations
  - Something you can always do to tame evaluation.
- Algorithms like quicksort are in textbooks
  - You mostly learn them rather than invent them.

Other courses will teach you more about the second category.
Acknowledgments

This lecture incorporates material from:

- Matthias Felleisen
- Robert Bruce Findler
- Matthew Flatt
- Gregor Kiczales
- Shriram Krishnamurthi
- Marc Smith
- U. Waterloo CS135