Class 17

Final Analysis
Review of a few things about types
The truth about lambda
Set equality and equality testing
Revisit big O

• Main ideas
• Two more useful theorems
• Example of a complete analysis of a procedure or two
Big-O, take two!

- Concentrate on functions $\mathbb{N} \rightarrow \mathbb{R}^+$
  - typical of operation-counting functions for recursive procedures!
  - The restriction to positive reals isn't standard in math, but makes things simpler for us,
- We say "$f$ is eventually less than $g$, up to constants" if there are numbers $M, c > 0$ with the property that for $n > M$, we have $f(n) \leq c \cdot g(n)$
- The notation $O(g)$ denotes the set of all functions that are eventually less than $g$, up to constants.
First big-O theorem

• **Big-O theorem 1:**
  If $A, B > 0$, and the function $H: \mathbb{N} \rightarrow \mathbb{R}^+$ satisfies a recurrence
  
  \[ H(0) = A \\
  H(n) \leq B + H(n - 1) \text{ for } n > 0 \]

  then $H \in O(n \mapsto n)$.

  NB: Also works if $H(n) < B + H(n - 1)$ for $n > 0$
Second big-O theorem

• Big-O theorem 2:
  If $A, B, C > 0$, and the function $H: \mathbb{N} \rightarrow \mathbb{R}^+$ satisfies
  
  $\begin{align*}
  & H(0) = A \\
  & H(n) \leq B + Cn + H(n-1) \text{ for } n > 0
  \end{align*}$

  then $H \in O(n \mapsto n^2)$.

Proof: very similar

NB: also works for $<$, as in theorem 1.
Big-O

Big-O categories are what computer scientists use to speak about algorithms

- "merge sort is in $O(n \leftrightarrow n \log n)$" is shorthand for "the operation-counting function for the merge-sort algorithm is an element of the set $O(n \leftrightarrow n \log n)$"
Big-O classes

• If \( f < g \), then \( O(f) \subset O(g) \)
• So \( O(n \mapsto 1) \subset O(n \mapsto n) \subset O(n \mapsto n^2) \subset \)
• So if you have a linear-time algorithm, you could say
  • "I've got a quadratic time algorithm!"
  • But people will be more impressed by "I've got a linear time algorithm!"
  • We generally aim to use the \textit{smallest} big-O category possible
  • In rare cases, everyone "knows" an algorithm is linear-time, but the only thing we can prove is, say, exponential time
    • Great opportunity to impress the algorithms community by finding a new proof
Big-O thinking

• From now on, I'm constantly going to be asking "Is what we've written here linear-time? Constant time? Quadratic time?"
• You'll get better at answering with experience.
Two more small but very useful theorems

• Big-O theorem 3:
  If $H: \mathbb{N} \rightarrow \mathbb{R}^+$, and $H \in O(n \mapsto n)$, then there are numbers $A, B > 0$ with the property that for all natural numbers $n$, we have
  $$H(n) < An + B$$

• Why isn't this obvious?
  • We know that for some $M, c > 0$ having $n > M$ tells us $H(n) \leq cn$, so can't we just choose $A = c, B = 0$?
    • Sure...but that would only make the inequality true for $n > M$, not for all $n$.
  • Didn't we show that if $H(n) < An + B$, then $H \in O(n \mapsto n)$?
    • We did, but now we're trying to prove the opposite thing!
An example

- \( H(n) = \frac{10}{n+1} + n \)
An example

- \( H(n) = \frac{10}{n+1} + n \)
- Slightly larger than \( K(n) = n \), in green
An example

- $H(n) = \frac{10}{n+1} + n$
- Slightly larger than $K(n) = n$, in green
- Evidently eventually less than $L(n) = 2n$, in blue
- Thus it's in $O(n \mapsto n)$
- We want to show it's less than $An + B$, for some $A, B$.
- Just shift the blue graph up. How much?
Big-O Theorem 3

If $H: \mathbb{N} \to \mathbb{R}^+$, and $H \in O(n \mapsto n)$, then there are numbers $A, B > 0$ with the property that for all natural numbers $n$, we have

$$H(n) < An + B$$

Proof:

1. The definition of big-O tells us that there are numbers $M, c > 0$ with the property that for $n > M > 0$, we have $H(n) \leq cn$, hence $H(n) < (c + 1)n$. Let $B = c + 1$.

2. Look at the numbers $H(0), H(1), \ldots, H(k)$, where $k$ is the first integer greater than $M$. Let $A$ be one more than the largest of these numbers. Then for $n = 0, 1, \ldots, k$, we have

$$H(n) < A \leq A + Bn$$

3. By step 1, for $n > k > M$, we have $H(n) < Bn < A + Bn$

So for all natural numbers $n$, we have $H(n) < A + Bn$, as claimed!
Big-O Theorem 4

If $H : \mathbb{N} \to \mathbb{R}^+$, and $H \in O(n \mapsto n^2)$, then there are numbers $A, B > 0$ with the property that for all natural numbers $n$, we have

$$H(n) < An^2 + B$$

Proof:

Essentially the same.
Two complete analyses
"contains17? runs in $O(n \leftrightarrow n)$ time"

1. (define (contains17? aloi)
2.   (cond
3.     [(empty? aloi) false]
4.     [(cons? aloi) (or (= (first aloi) 17) (contains17? (rest aloi)))]))

- Let $C(n)$ be the largest number of operations involved in evaluating contains17? on any list of length $n$. Then $C: \mathbb{N} \to \mathbb{R}^+$, because the number of operations is always positive.
- From lines 1, 2, 3 of code, we see that $C(0)$ is some small constant $A > 0$ as cond, empty?, testing a boolean, etc., are all unit-time operations.
- From lines 1-4, we can see that for list of nonzero length $n$, the number of operations aside from those in the recursive call in line 4, is some other small constant, for similar reasons; the time taken in the recursive call is no more than $C(n - 1)$, because the recursive argument has length $n - 1$, and $C(n - 1)$ represents the greatest possible number of operations on such a list.
"contains17? runs in $O(n \mapsto n)$ time"

1. (define (contains17? aloi)
2.   (cond
3.     [(empty? aloi) false]
4.     [(cons? aloi) (or (= (first aloi) 17) (contains17? (rest aloi)))]))

- (short form for sake of space; you should write out stuff from prior slide!)
- Let $C(n)$ be the largest number of operations involved in evaluating contains17? on any list of length $n$. Then $C: \mathbb{N} \rightarrow \mathbb{R}^+$, because the number of operations is always positive.
- $C(0) = A > 0$
- For $n > 0$, $C(n) < B + C(n - 1)$, and $B > 0$.
- Big-O theorem 1 tells us that $C \in O(n \mapsto n)$. QED.
member?

• Essentially identical argument lets us conclude...

Let \( M(n) \) be the largest number of operations involved in evaluating (member? x alod) on any list alod of length \( n \)," then

\[ M \in O(n \mapsto n) \]
Analysis of set?

1. (define (set? aloi)
2.   (cond
3.     [(empty? aloi) true]
4.     [(cons? aloi) (and (not (member? (first aloi) (rest aloi)))
5.       (set? (rest aloi)))]))

- Let $S(n)$ be the largest number of operations involved in evaluating set? on any list of length $n$. Then $S: \mathbb{N} \to \mathbb{R}^+$, because the number of operations is always positive.
- From lines 1, 2, 3 of code, we see that $S(0)$ is some small constant $A > 0$ as cond, empty?, testing a boolean, etc., are all unit-time operations.
- From lines 1-5, we can see that for list of nonzero length $n$, the number of operations aside from those in the recursive call in line 4, is some other small constant, for similar reasons, plus the time taken by member? on a list of length $n-1$. The time taken in the recursive call is no more than $S(n - 1)$, because the recursive argument has length $n - 1$, and $S(n - 1)$ represents the greatest possible number of operations on such a list.
Analysis of set?

1. (define (set? aloi)
2.   (cond
3.     [(empty? aloi) true]
4.     [(cons? aloi) (and (not (member? (first aloi) (rest aloi)))
5.       (set? (rest aloi)))]))

1. Let $S(n)$ be the largest number of operations involved in evaluating set? on any list of length $n$. Then $S: \mathbb{N} \to \mathbb{R}^+$, because the number of operations is always positive.

2. $S(0) = A > 0$

3. $S(n) < B + M(n - 1) + S(n - 1)$, where $B > 0$, and $M$ is the operation-counting function for member?, and we know that $M \in O(n \mapsto n)$.

4. Big-O theorem 3 tells us there are numbers $C, D > 0$ such that $M(n) < Cn + D$ for all $n$, so $M(n - 1) < C(n - 1) + D$

5. $S(n) < B + C(n - 1) + D + S(n - 1) = (B + D - C) + Cn + S(n - 1) < (B + D) + Cn + S(n - 1)$

6. Note that $(B+D)$ and $C$ are both positive, so big-O theorem 2 applies!

7. Applying big O theorem 2, we see that $S \in O(n \mapsto n^2)$ \text{QED}
Whew!

• That's how analysis, in most of CS17, looks.
• Define some op-counting function
• Write a recurrence it satisfies
• Use the big-O theorems to show it's in some big-O class

• Behind-the-scenes work: Prove those big-O theorems as needed.
set equality

• Represent sets as lists of items, with no repetitions.
• Order doesn't matter
• Ordinary list-equality isn't a test of set-equality!
  • (list 1 2) (list 2 1): different lists, same set!
• Sets B and C are equal if
  1. every member of B is a member of C ("B is a subset of C")
  2. every member of C is a member of B ("C is a subset of B")

(define (set-equal? b c) (and (subset? b c) (subset? c b)))
Subset?

• is b a subset of c?
  • Is each element of b an element of c?

(define (subset? b c)
    (foldr (lambda (item result) (and (member? item c) result))
      true b))
Subtlety

• We can only test equality of sets containing things for which "equality" is defined
  • num, bool, string
  • lists of those things
  • lists of lists of those things
  • not procedures!
    • *Deep theorem says it's impossible to test "equality" of procedures, where "equality" means "produces same result given same input(s)"*
Limits on lambda

• We cannot seem to write a recursive procedure using lambda

(let
  ((len (lambda (alod)
            (cond
              [(empty? alod) 0]
              [(cons? alod) (+ 1 (len (rest alod)))])))
   (len (list 1 2 3)))

• Inner "len" cannot refer to outer one (see the rules of evaluation)
A solution

(letrec
  ((len (lambda (alod)
      (cond
       [(empty? alod) 0]
       [(cons? alod) (+ 1 (len (rest alod)))]))))
  (len (list 1 2 3))) => 3
Another lambda secret

• We said the value of a lambda expression was a "closure" containing
  1. *The argument list*
  2. *the body*

• The truth is that there's a third part of the closure

• It also contains a "local environment", informally consisting of all the "bindings" in the current environment except those in the top-level-environment

• The rule for evaluating a lambda expression changes too:
  1. First extend the current environment by the closure's local environment
  2. bind formal arguments to actual arguments
  3. evaluate the body in the extended environment
  4. remove the extensions from step 1.
Example

\[
\text{(define (incrementer b) (lambda (x) (+ x b))}
\text{(define (add2 x) (incrementer 2))}
\text{(add2 5)}
\]

Without the "local environment", we'd get "b undefined" error!
This will be, for many of you, the single hardest part of the next project.