61A LECTURE 17 – ORDERS OF GROWTH, EXCEPTIONS

Steven Tang and Eric Tzeng
July 23, 2013
Announcements

• Regrades for project 1 composition scores, due by next **Monday**
  • See Piazza post for more details

• **Midterm 2** is next Thursday, August 1, at 7pm.
  • If you have a conflict at that time, fill out the conflict form on Piazza ASAP

• Potluck on Friday in the Woz at 6PM. See you there!
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)

\[ R(n) = \Theta(f(n)) \]

means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]

for sufficiently large values of \( n \).
A graphical explanation

\[ R(n) = \Theta(f(n)) \]

means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]

for sufficiently large values of \( n \).
Warm up!

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

def sunshine(n):
    if n == 0:
        return 0
    happiness = 1
    while happiness < 10000000:
        happiness += 1
    return happiness + sunshine(n - 1)

def eternity(n):
    i = 0
    while i < n:
        factorial(n)
        i += 1
```

Time

- $\Theta(n)$
- $\Theta(n)$
- $\Theta(n^2)$

A constant amount of work – doesn’t contribute to the order of growth!
Comparing Orders of Growth (n is problem size)

- **Θ(b^n)**: Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

- **Θ(n^6)**: Incrementing the problem scales $R(n)$ by a factor.

- **Θ(n^2)**: Quadratic growth. E.g., operations on all pairs. Incrementing $n$ increases $R(n)$ by the problem size $n$.

- **Θ(n)**: Linear growth. Resources scale with the problem.

- **Θ(\sqrt{n})**: Linear growth. Resources scale with the problem.

- **Θ(log n)**: Logarithmic growth. These processes scale well. Doubling the problem only increments $R(n)$.

- **Θ(1)**: Constant. The problem size doesn't matter.
Implementing Sets

What we should be able to do with a set:

- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in \( set1 \) or \( set2 \)
- Intersection: Return a set with any elements in \( set1 \) and \( set2 \)
- Adjunction: Return a set with all elements in \( s \) and a value \( v \)
Implementation considerations

• Many ways to accomplish this
• Not all solutions are made equal!
• Some implementations might be better than other implementations when performing certain operations
Sets as Unordered Sequences

Proposal 1: A set is represented by a recursive list that contains no duplicate items.

```python
def empty(s):
    return s is Rlist.empty

def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)

Θ(n)  
The size of the set
```
Sets as Unordered Sequences

```python
def adjoin_set(s, v):
    if setContains(s, v):
        return s
    return Rlist(v, s)

def intersect_set(set1, set2):
    f = lambda v: setContains(set2, v)
    return filter_rlist(set1, f)

def union_set(set1, set2):
    f = lambda v: not setContains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)
```

Time order of growth

- \( \Theta(n) \)
The size of the set

- \( \Theta(n^2) \)
Assume sets are the same size

- \( \Theta(n^2) \)
Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```python
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    return set_contains2(s.rest, v)
```

Order of growth? $\Theta(n)$
def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)

def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)

set_contains2 is slightly more optimized than set_contains, but they are still both linear time operations. Both functions have an order of growth $\Theta(n)$.
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
    elif e2 < e1:
        return intersect_set2(set1, set2.rest)
```

Order of growth? $\Theta(n)$

Compare to the first version of intersect_set.
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
Tree Sets

**Proposal 3:** A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch
Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```python
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```

If 9 is in the set, it is in this branch

Order of growth?
Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

None  None  None  None
What Did I Leave Out?

Sets as ordered sequences:
• Adjoining an element to a set
• Union of two sets

Sets as binary trees:
• Intersection of two sets
• Union of two sets

That's homework 9!
Break
Handling Errors

Sometimes, computers don't do exactly what we expect

• A function receives unexpected argument types
• Some resource (such as a file) is not available
• A network connection is lost

September 9 1947: Moth found in a Mark II Computer
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

These def statements create function objects as always, but their names are bound as attributes of the class.
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

Mastering exceptions:

Exceptions are objects! They have classes with constructors

They enable non-local continuations of control:

If $f$ calls $g$ and $g$ calls $h$, exceptions can shift control from $h$ to $f$ without waiting for $g$ to return

However, exception handling tends to be slow
Assert Statements

Assert statements raise an exception of type `AssertionError`

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in production systems

```
python3 -O
```

"O" stands for optimized. Among other things, it disables assertions

Whether assertions are enabled is governed by the built-in bool `__debug__`
Raise Statements

Exceptions are raised with a `raise statement`

```
raise <expression>
```

`<expression>` must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from `BaseException`

- **TypeError** -- A function was passed the wrong number/type of argument
- **NameError** -- A name wasn't found
- **KeyError** -- A key wasn't found in a dictionary
- **RuntimeError** -- Catch-all for troubles during interpretation
Try Statements

*Try statements* handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...  
```

Execution rule:

• The `<try suite>` is executed first;
• If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and
• If the class of the exception inherits from `<exception class>`, then
• The `<except suite>` is executed, with `<name>` bound to the exception
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

**Multiple try statements**: Control jumps to the except suite of the most recent try statement that handles that type of exception.
WWPD: What Would Python Do?

How will the Python interpreter respond?

```python
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)

>>> invert_safe(1/0)
Never printed if x is 0

>>> try:
    invert_safe(0)
except BaseException:
    print('Handled!')

>>> inverrrrt_safe(1/0)
```

WWPD?
Quick Break!

- We will start talking about Scheme today – Eric will dive more deeply into Scheme tomorrow!
Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of OOP

“The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.”
-Richard Stallman, founder of the Free Software movement

“Probably my favorite programming language.”
-Eric Tzeng, CS61A Instructor
Scheme Fundamentals

Scheme programs consist of expressions, which can be:

• Primitive expressions: 2, 3.3, true, +, quotient, ...
• Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3 (+ (* 2 4) (+ 3 5)))
(+ (- 10 7) 6))

“quotient” names Scheme’s built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn’t matter)
Special Forms

A combination that is not a call expression is a special form:

• If expression: \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\)

• And and or: \((\text{and} \ <e_1> \ ... \ <e_n>) \ (\text{or} \ <e_1> \ ... \ <e_n>)\)

• Binding names: \((\text{define} \ <\text{name}> \ <\text{expression}>)\)

• New procedures: \((\text{define} \ (<\text{name}> \ <\text{formal parameters}>) \ <\text{body}>)\)

> (define pi 3.14)
> (* pi 2)
6.28

> (define (abs x)
   (if (< x 0)
       (- x)
       x))
> (abs -3)
3

The name “pi” is bound to 3.14 in the global frame
A procedure is created and bound to the name “abs”
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
(\text{lambda} \ (<\text{formal-parameters}>) \ <\text{body}>)
\]

Two equivalent expressions:

\[
(\text{define} \ (\text{plus4} \ x) \ (+ \ x \ 4))
\]

\[
(\text{define} \ \text{plus4} \ (\text{lambda} \ (x) \ (+ \ x \ 4)))
\]

An operator can be a combination too:

\[
((\text{lambda} \ (x \ y \ z) \ (+ \ x \ y \ (\text{square} \ z))) \ 1 \ 2 \ 3)
\]

Evaluates to the \textit{add-x-amp-y-amp-z^2} procedure
Pairs

We can implement pairs functionally:

```scheme
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also has built-in pairs that use weird names:

- `cons`: Two-argument procedure that creates a pair
- `car`: Procedure that returns the first element of a pair
- `cdr`: Procedure that returns the second element of a pair

A pair is represented by a dot between the elements, all in parens

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```
Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list

Scheme lists are recursive lists:

• nil is the empty list
• A non-empty Scheme list is a pair in which the second element is nil or a Scheme list

Scheme lists are written as space-separated combinations

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

Not a well-formed list!