Lecture #10: Sequences

Public Service Announcement

"Align is a new student organization on campus that hopes to unite undergraduate students across the disciplines of biology, computer science, applied math, and statistics who have an interest in computational biology. We are interested in creating accessible workshops and professional development opportunities, as well as social events, to increase awareness of this growing field and encourage student involvement.

If you are interested in being involved in our organization or want to learn more about us, you can send us an e-mail at align.uscb@gmail.com."

Public Service Announcement

"The International Society for Pharmaceutical Engineering (ISPE) connects students from a variety of backgrounds. This year we aim to promote innovation and teamwork among Cal students in the field of bioengineering, health, and medicine through Hack for Humanity. This event will focus on a bettering the world via human health, challenging participants to design and create a medical device to solve a health related problem.

Hack for Humanity will take place 2/26–2/28 with a kickoff event on 2/25, for more information, check out: `https://www.facebook.com/events/120509`

To learn more about Hack for Humanity, check out our infosession on 2/11 at 7PM in 310 Soda: `https://www.facebook.com/events/562681597241046/`"

Announcements

- Homework party this Thursday (2/11) from 6–9pm in B6 Evans.
- DO NOT PUBLICLY POST YOUR CODE ON PIAZZA!!!!

Sequences

- The term *sequence* refers generally to a data structure consisting of an *indexed collection of values.*
- That is, there is a first, second, third value (which CS types call #0, #1, #2, etc.
- A sequence may be *finite* (with a length) or *infinite.*
- As an object, it may be *mutable* (elements can change) or *immutable*.
- There are numerous alternative interfaces (i.e., sets of operations) for manipulating it.
- And, of course, numerous alternative implementations.
- Today: immutable, finite sequences, recursively defined.

A Recursive Definition

- A possible definition: A sequence consists of
  - An empty sequence, or
  - A first element and a sequence consisting of the elements of the sequence other than the first—the rest of the sequence or *tail.*
- The definition is clearly recursive ("a sequence consists of ...a sequence ..."), so let's call it an `rlist` for now.
- Suggests the following ADT interface:

```
empty_rlist = ...
def make_rlist(first, rest = empty_rlist):
    """A recursive list, r, such that first(r) is FIRST and
    rest(r) is REST,  which must be an rlist."""
def first(r):
    """The first item in R."""
def rest(r):
    """The tail of R."""
def isempty(r):
    """True iff R is the empty sequence"""
```

Implementation With Pairs

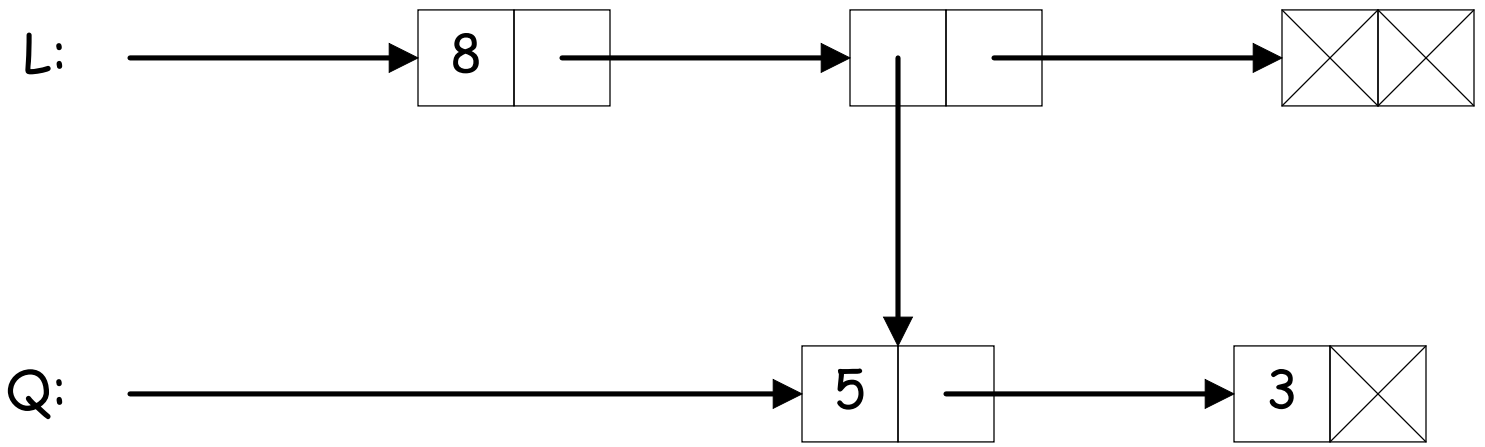
- An obvious implementation uses two-element tuples (pairs). The result is called a *linked list.*

```
empty_rlist = None
def make_rlist(first, rest = empty_rlist):
    return first, rest
def first(r):
    return r[0]
def rest(r):
    return r[1]
```

Box-and-Pointer Diagrams for Linked Lists

- Diagrammatically, one gets structures like this:

```
# The sequence containing:  8; the sequence containing 5 and 3;
#    and the empty sequence
Q = make_rlist(5, make_rlist(3, empty_rlist))
L = make_rlist(8,
               make_rlist(Q, make_rlist(empty_rlist, empty_rlist)))
# or
# Q = make_rlist(5, make_rlist(3))
# L = make_rlist(8, make_rlist(Q, make_rlist(empty_rlist)))
```



From Recursive Structure to Recursive Algorithm

- The cases in the recursive definition of list often suggest a recursive approach to implementing functions on them.
- Example: length of an rlist:

```
def len_rlist(s):                    # A sequence is:
    """The length of rlist 's'."""
    if isempty(s):                 # Empty or...
        return 0
    else:
        return 1 + len_rlist(rest(s))
                                     # A first element and
                                     # the rest of the list
```

- **Q:** Why do we know the comment is accurate?
- ***A*:** Because we assume the comment is accurate!
  (For "smaller" arguments, that is).
- An example of reasoning by *structural induction...*
- ...or *recursive thinking* about data structures.

Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'.
    >>> L = make_rlist(2, make_rlist(3, make_rlist (4)))
    >>> getitem_rlist(L, 1)
    3"""

    if _______:
        return _______
    else:
        return _______
```

getitem_rlist (II)

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, i):
    "Return the element at index ‘i’ of recursive list ‘s’."

    if i == 0:
        return first(s)
    else:
        return getitem_rlist(rest(s), i-1)
```

Iterative getitem_rlist

```
def getitem_rlist(s, i):
    "Return the element at index ‘i’ of recursive list ‘s’."
    while i != 0:
        s, i = rest(s), i-1
    return first(s)
```

On to Higher Orders!

```
def map_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist
       S in order."""
    if _______:
        return _______
    else:
        return _______
```

Map implemented

```
def map_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist
       S in order."""
    if isempty(s):
        return empty_rlist
    else:
        return make_rlist(f(first(s)), map_rlist(f, rest(s)))
```

- So `map_rlist(lambda x:x**2, L)` produces a list of squares.
- [Python 3 produces a different kind of result from its `map` function; we'll get to it.]
- Iterative version not so easy here!

Filtering

- Map unconditionally applies its function argument to elements of a list. It is essentially a loop.
- The analog of applying an **if** statement to items in a list is called *filtering*:

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of
    rlist 'seq' for which the 1-argument function 'cond'
    returns a true value."""

    if _______: return _______
    elif _______: return _______
    else:       return _______
```

Filtering Implemented

```
def filter_rlist(cond, seq):
    """The rlist consisting of the subsequence of
    rlist 'seq' for which the 1-argument function 'cond'
    returns a true value."""

    if isempty(seq):
        return empty_rlist
    elif cond(first(seq)):
        return make_rlist(first(seq),
                          filter_rlist(cond, rest(seq)))
    else:
        return filter_rlist(cond, rest(seq))
```

- Oops! Not tail-recursive. Iteration is problematic (again).
- In fact, until we get to talking about mutable recursive lists, we won't be able to do it iteratively without creating an extra list along the way.

Python's Sequences

- Rlists are sequences with a particular choice of interface that emphasizes their recursive structure.
- Python has a much different approach to sequences built into its standard data structures, one that emphasizes their *iterative* characteristics.
- There are several different kinds of sequence embodied in the standard types: *tuples, lists, ranges, iterators,* and *generators*. We'll start with the first two, which are run-of-the mill data structures.

Sequence Features

- For this part of the course, where we emphasize computation by *construction* rather than *modification*, the interesting characteristics include:
  - **Explicit Construction:**

```
t = (2, 0, 9, 10, 11)   # Tuple
L = [2, 0, 9, 10, 11]   # List
R = range(2, 13)        # Integers 2-12.
R0 = range(13)          # Integers 0-12.
E = range(2, 13, 2)     # Even integers 2-12.
S = "Hello, world!"     # Strings
```

  - **Indexing:**

```
t[2] == L[2] == 9,  R[2] == 4,   E[2] == 6
t[-1] == t[len(t)-1] == 11
S[1] == "e"
```

  - **Slicing:**

```
t[1:4] == (t[1], t[2], t[3]) == (0, 9, 10),
t[2:] == t[2:len(t)] == (9, 10, 11)
t[::2] == t[0:len(t):2] == (2, 9, 11),  t[::-1] == (11, 10, 9, 0, 2)
S[0:5] == "Hello",  S[0:5:2] == "Hlo", S[4::-1] == "olleH"
```

Sequence Iteration: For Loops

- We can write more compact and clear versions of **while** loops:

```
>>> t = (2, 0, 9, 10, 11)
>>> s = 0
>>> for x in t:
>>>     s += x
>>> print(s)
32
```

- Iteration over numbers is really the same, conceptually:

```
>>> s = 0
>>> for i in range(1, 10):
>>>     s += i
>>> print(s)
45
```

Higher-Order Manipulation of Sequences

- Python 3 defines `map` (just as on rlists), as well as `accumulate` (called `reduce`), and `filter` on sequences just as we did on rlists.
- So to compute the sum of the even Fibonacci numbers among the first 12 numbers of that sequence, we could proceed like this:

```
First 20 integers:
    0   1   2   3   4   5   6   7   8   9   10   11
Map fib:
    0   1   1   2   3   5   8  13  21  34   55   89
Filter to get even numbers:
    0           2           8          34
Reduce to get sum:
    44
```

- ...or:

```
reduce(add, filter(iseven, map(fib, range(12))))
```

- Why is this important? Sequences are amenable to *parallelization*.

List Comprehensions

- In fact, one doesn't often need `map` and `filter` because Python has a succinct syntax for expressing their application: the list comprehension.
- Full form:

```
[ <expression> for <var> in <sequence expression>
                if <boolean expression> ]
```

- Example: Squares of the prime numbers up to 100.

```
[ x*x for x in range(101) if isprime(x) ]
```

An aside: Sequences in Unix

g

- Many Unix utilities operate on *streams of characters,* which are sequences.
- With the help of pipes, one can do amazing things. One of my favorites:

```
tr -c -s '[:alpha:]' '[\n*]' < FILE | \
sort | \
uniq -c | \
sort -n -r -k 1,1 | \
sed 20q
```

  which prints the 20 most frequently occuring words in *FILE*, with their frequencies, most frequent first.