MEMOIZATION, RECURSIVE DATA, AND SETS 4b

COMPUTER SCIENCE 61A

July 18, 2013

1 Memoization

Later in this class, you'll learn about orders of growth and how to analyze exactly how efficient (or inefficient) a function is. However, for now we'll just tell you that many of the naive implementations for recursive functions that we have used (i.e. `recursive_fib`) are actually very expensive in terms of time. Memoization provides a way for us to reduce the expense of recursive functins. It works by storing the return value of our function as they are computed. This way, if at some time we have to compute the return value with an argument we have already seen, we can just return that value instead of spending the time to compute it again. Here is an example of a function `memo` that takes a function and returns a memoized version of it.

```
def memo(f):
    """Return a memoized version of single-argument function f."""
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Is there a trade off? Memoization requires space to store values that have already been computed. For large `n`, this means a lot of stored values, which equates to a lot of used space. So, we can describe the relationship between space and time as an inverse relationship. If we want our functions to use less time, we'll have to use more space, while if we want our functions to use less space, we'll have to wait longer.

1.1 Questions

1. List three recursive functions that would benefit from being memoized. (Hint: Think back to the recursive functions you've implemented in homeworks, labs, and discussion)

2. Our current memoization function only works with functions that take one argument. How could we make a simple to change to allow it to work with functions that have an arbitrary number of arguments?

3. Give an example of when using a memoized `fib_recursive` would be faster than using `fib_iter` to compute the 1000th fibonacci number.

2 Recursive Lists

We've already seen Rlists implemented as recursive pairs, and we've drawn box-and-pointer representing their structure. What we'll go through today is an object implementation.

Here is the the code to implement an OOP verision of an Rlist.

```
class Rlist(object):
        """A recursive list consisting of
           a first element and the rest."""

        empty = False

        def __init__(self, first, rest=None):
            if first == None:
                self.empty = True
            elif rest == None:
                rest = Rlist(None)
            self.first = first
            self.rest = rest
        def __repr__(self):
            args = repr(self.first)
            if not self.rest.empty:
                args += ', {0}'.format(repr(self.rest))
            return `Rlist({0})'.format(args)
        def __len__(self):
            if empty:
                return 0
            return 1 + len(self.rest)
        def __getitem__(self, i):
            if i == 0:
                return self.first
            return self.rest[i-1]
```

We can construct an Rlist like so:

```
s = Rlist(1, Rlist(2, Rlist(3)))
```

For a given Rlist `s`, remember that it has two main attributes:

- **s.first**: the actual item stored in the current index of the Rlist
- **s.rest**: the rest of the Rlist sequence, represented recursively as another Rlist

In our implementation, we can construct `empty_rlists` by passing the Rlist constructor None. Each Rlist has an instance attribute `empty` that stores whether or not the Rlist is empty. This attribute defaults to false for all Rlists, unless the Rlist is constructed using None as a paramater.

```
>>> empty_rlist = Rlist(None)
>>> empty_rlist.empty
True
```

2.1 Questions

1. Write a function `pop_rlist` that takes an Rlist and index, and pops off the value at that index. Note: if pop is called with no index, it should default to the front.

   ```
   >>> s = Rlist(4, Rlist(2, Rlist(3)))
   >>> pop_rlist(s, 1)
   2
   >>> s
   Rlist(4, Rlist(3))
   >>> s.pop(s)
   4
   >>> s
   Rlist(3)

   def pop_rlist(s, index=0):
   ```

2. Write a function `push_rlist` that takes an Rlist and index, and pushes a value onto the front of it.

   ```
   >>> s = Rlist(2, Rlist(4, Rlist(1)))
   >>> push_rlist(s, 9)
   >>> s
   Rlist(9, Rlist(2, Rlist(4, Rlist(1))))

   def push_rlist(s, value)
   ```

3. Selection sort is a sorting algorithm that works by finding the largest element of a list, placing it with the element in the first index, then recursively sorting the rest of the list. Write a function `selection_sort_rlist` that will perform an selection sort on an rlist. Hint: you may want to define a function get largest element index. You also may find the functions `pop_rlist` and `insert_rlist` that have already been defined. earlier useful.

```
def insert_rlist(rlist, value, index):
    if index == 0:
        rlist.rest = Rlist(rlist.first, rlist.rest)
        rlist.first = value
    elif rlist.rest.empty:
        print('Index out of bounds')
    else:
        insert(rlist.rest, value, index - 1)

def selection_sort_rlist(s):
"""
>>> s = Rlist(3, Rlist(4, Rlist(5, Rlist(2))))
>>> selection_sort_rlist(s)
>>> s
Rlist(5, Rlist(4, Rlist(3, Rlist(2))))
```

4. Define a function `rlist_fixer` that takes in poorly constructed Rlist and fixes them, preserving the order of the elements.

```
def rlist_fixer(s):
    """
    >>> s = Rlist(3, Rlist(Rlist(4, Rlist(5)), Rlist(4)))
    >>> s
    Rlist(3, Rlist(Rlist(4, Rlist(5)), Rlist(4)))
    >>> rlist_fixer(s)
    >>> s
    Rlist(3, Rlist(4, Rlist(5, Rlist(4))))
```

3 Sets

Now we're gonna add to the list of built-in Python containers that you already know. As a refresher, you have used list, tuples, and dictionaries and containers for storing various things. A **set** is a python container, which looks visually in Python like the offspring of a dictionary and list. We use the same notation that is used in math to denote a set, which are curly braces. In Python, sets are unordered collections, so the printed ordering may differ from the element ordering in the set literal.

```
>>> my_set = {3, 5, 4, 7, 4, 9, 5, 3}
>>> my_set
{3, 4, 5, 6, 9}
```

Like the other containers we've already worked with, Python sets support various operations.
We can find the length of set.

```
>>> len(my_set)
5
```

We can test membership.

```
>>> 9 in s
True
```

We can even do fancy things like union and intersect.

```
>>> my_set.union({1, 2, 7, 8})
{1, 2, 3, 4, 5, 6, 7, 8}
>>> my_set.intersection({1, 2, 3})
{3}
```

3.1 Questions

1. Define a function `split_set` that takes in a set and a pivot, and splits the set into two subsets. One subset is for elements smaller than the pivot, and the second is for the elements greater than or equal to the pivot. Return the two subsets in a list where the first element is set of values smaller than the pivot and the second set is the set of values greater than or equal to the pivot. Hint: sets support the operators `pop` which will remove and return an arbitrary element from a set and `add` which will add an element to a set.

```
def split_set(s, pivot):
"""
>>> s = {1, 2, 3, 4}
>>> split_set(s, 3)
[{1, 2}, {3, 4}]
```

2. Now that you have seen Python's built in set, let's implement our own version! Fill in the definition for a Set class with the supported operations using whatever implementation you like. However, you are not allowed to use Python's built in sets!

```
class Set(Object):

    def __init__(self, elements=[]):
    # For our constructor, we can give it a list of
    # initial elements, or just initialize an empty set.
```

```
def __len__(self):
# returns the length of the set
```

```
def add(self, elem):
# adds an element to the set
```

```
def remove(self, elem):
# adds an element to the set
```

```
def pop(self):
# pops a random element off the set
```

```
def contains(self, value):
# returns whether or not the set contains a value
```

```
def union(self, s):
# unions this set object with another set s
# after running this function s can be empty or remain the same
```

```
def intersect(self, s):
# intersects this set object with another set s
# again, s can be empty or unchanged
```