Module 7: Generative Recursion

Topics:

- Generative Recursion
- Sorting Algorithms
- Analyzing generative recursive code
- Designing generative recursive code

Readings: HtDP 25, 26, Intermezzo 5,
ThinkP 5.8-5.10, 6.5-6.7

Types of recursion so far

- Structural recursion
  - Based on recursive definition of input data values
  - Standard templates
- Accumulative recursion
  - Builds up intermediate results on recursive calls
  - Specialized template
- Some algorithms do not fall into either of these categories
- Applies to recursion in Scheme and Python

Example: `gcd`

- The greatest common divisor (*gcd*) of two natural numbers is the largest natural number that divides evenly into both.
  - *gcd (10, 25) = 5*
  - *gcd (20, 22) = 2*
  - *gcd (47, 21) = 1*
- Exercise: Write `gcd` function using the standard count down template.

Euclid's Algorithm for **gcd**

- gcd(m,0) = m
- gcd(m,n) = gcd(n, m mod n)

```
def gcd(m,n):
    if m==0:  return n
    elif n==0:  return m
    else:  return gcd(n, m % n)
```


Tracing **gcd**

```
gcd(25, 10)
⇒ gcd(10, 25 % 10)
⇒ gcd(10, 5)
⇒ gcd(5, 10 % 5)
⇒ gcd(5, 0)
⇒ 5
```


Comments on **gcd**

- Not structural (not counting up or down by 1)
- Not accumulative

⇒Generative recursion

- Still has a base case
- Still has a recursive case – but problem is broken down in a new way

Why generative recursion?

- Allow more creativity in solutions
- Remove restrictions on solutions
- May allow for improved efficiency
- May be more intuitive for some problems
  – Breaking into more “natural” subproblems
- We need to "generate" (figure out) the subproblems

Steps for Generative Recursion

1. Divide the problem into subproblem(s)
2. Determine base case(s)
3. Figure out how to combine subproblem solutions to solve original problem
4. Use local variables and helper functions to make division more understandable
5. TEST! TEST! TEST!

Example: removing duplicates

```
# singles: (listof X) -> (listof X)
# Produces a list like lst, but
# containing only the first
# occurrences of each element in lst
# Examples: singles([]) => []
# singles([1,2,1,3,4,2]) => [1,2,3,4]
```

```
def singles(lst):
  if lst==[]: return []
  else:
    first = lst[0]
    rest = lst[1:]
    f_rem = filter(lambda x:
                     x!=first, rest)
    return [first] + singles(f_rem)
```

Question: What is the running time of **`singles`?**

Example: reversing a number

Write a function **`backwards`** that consumes a natural number and produces a new number with the digits in reverse order.

For example,

- **`backwards (6) => 6`**
- **`backwards (89) => 98`**
- **`backwards (10011) => 11001`**
- **`backward (5800) => 85`**

A Possible Approach

Consider the number n = 5678

- Divide the number into:
  - Last digit: 8
  - Everything else: 567
- Next, reverse 567
  - Take last digit (7) and "add to" 8 => 87
  - What's left? 56
- Repeat the process until all digits processed.

Coding the Approach

- Use accumulative recursion
- The helper function will keep track of:
  - The digits that have been reversed so far
  - The digits that still need to be reversed
  - The helper will use generative recursion
    - Counting up or down by 1 doesn't help!


```
# bw_acc: nat nat -> nat
# produces the number resulting from
# adding the reversed digits of
# res to the end of so-far
def bw_acc(so_far, res):
    if res == 0:
        return so_far
    else:
        next_so_far = so_far*10 + res % 10
        next_res = res / 10
        return bw_acc(next_so_far,
                      next_res)

def backwards(n):
    return bw_acc(0, n)
```


The Sorting Problem

```
# sort-list:
# (listof float) -> (listof float)
# Produces a list with all of the same
# elements as lst, but in sorted order
# from smallest to largest.
# The original list is not changed.
# Assumption: No duplicate values.
# Example:
# sort-list([1.0,4.0,3.0,2.0])
# => [1.0,2.0,3.0,4.0]
```

Insertion Sort (from CS115)

- An empty list is already sorted
- If the rest of the list was already sorted
  - Just find the correct spot to insert the first value in the list

Insertion Sort from CS115 (increasing order)

```
def insert_sort(L):
  if L == []: return []
  else:
    val = L[0]
    sorted = insert_sort(L[1:])
    pos = insert_pos(val, sorted)
    sorted.insert(pos, val)
    return sorted
```


The **insert_pos** helper function

```
# insert_pos: float (listof float)
#              -> int[>=0]
# Produces the position in which to insert v
# so that it is in sorted order within
# already sorted L
def insert_pos(v, L):
    if L == []:
        return 0
    elif v < L[0]:
        return 0
    else:
        return 1 + insert_pos(v, L[1:])
```

Running time of `insert_sort`

Sorting `[`x_1, x_2, . . . , x_n`]`:

- n calls to **`insert_sort`**
- Each call involves:
  - Calculating **`L[1:]`**
  - Calling **`insert_pos`**
  - Calling **`insert`**
- Each of those steps are each *O(n)* worst-case *(What is the best-case?)*

⇒**`insert_sort`** has worst case quadratic running time

Selection sort: another sorting algorithm

Consider this approach to sorting:

– Find the smallest value

– Put it at beginning of list

– Sort what's left by repeating this process


```
def selection_sort(L):
  if L==[]: return []
  sm = min(L)
  not_sm = filter(lambda x:
                  x!=sm, L)
  return [sm] + \
       selection_sort(not_sm)
```

Running Time of **selection-sort** (assume list has length **n**)

List Size		sm	not_sm	+	Total steps
n		n	n	n	3n
n-1		n-1	n-1	n-1	3n-3
n-2		n-2	n-2	n-2	3n-6
…					
1		1	1	1	3

Mergesort– another sorting algorithm

Consider the following approach

– Divide the list into two halves
– Sort the first half
– Sort the second half
– Combine the sorted lists together

=> Done!

Mergesort questions

- How to split the list?
  – Find the middle – before and after
- How to sort smaller lists?
  – Use same idea again (mergesort recursively)
- When to stop recursion?
  – When the list is empty
- How to combine the parts?
  – merge

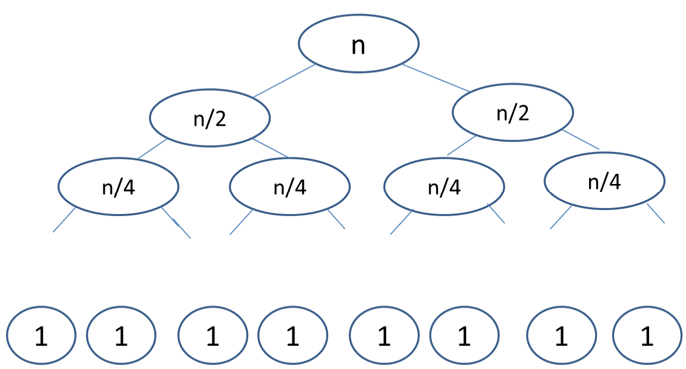
Merge helper function

```
# L1, L2 are in increasing order
def merge(L1,L2):
    if L1==[]: return L2
    if L2==[]: return L1
    if L1[0] < L2[0]:
        return [L1[0]] + \
           merge(L1[1:], L2)
    return [L2[0]] + \
           merge(L1,L2[1:])
```



```
def mergesort(L):
    if len(L) < 2: return L
    mid = len(L)/2
    L1 = L[:mid]
    L2 = L[mid:]
    sortedL1 = mergesort(L1)
    sortedL2 = mergesort(L2)
    return merge(sortedL1,
                 sortedL2)
```


Calls to **mergesort**



Running time of **`mergesort`**

- Consider the time across each level of the tree of the tree
  - How long does it take to divide the lists in half?
  - How long does it take to merge the lists together?
- How many levels of the tree are there?

- Total running time is O(n log n)

Built-in **`sorted`** and **`sort`**

- Python: **`sorted`**
  - Built-in function
  - Consumes a list and produces a sorted copy
- Python: **`sort`**
  - A list method
  - Consumes a list and *modifies* into sorted order
- Additional arguments can be provided to change the sort (e.g. into decreasing order)
- Scheme: **`quicksort`**
  - Built-in function
  - Consumes a list and produces a sorted copy

Comments on Generative Recursion

- More choices increases chances for errors
- Design recipe:
  - No general template for generative recursion
  - Contract, purpose, examples are still important
  - Testing more important than ever!
- Structural and accumulative recursion remain best choice for many problems
  - Templates are still important!
- An algorithm can use combinations of different types of recursion

Goals of Module 7

- Understand how generative recursion is more general than structural or accumulative recursion
- Understand how insertion sort, selection sort and mergesort work
- Be able to compare running times of sorting algorithms
- Be aware that generatively recursive solutions may be harder to debug