Recursion: Introduction
Announcements

- Deadlines
- Exam on Thursday: Units 1 – 5 (inclusive)
- PA 4 due tonight
- OLI Recursion over the weekend
Today

- Review of Big-O

- Recursion:
  - Introduction to recursion
  - What it is
  - Recursion and the stack
  - Recursion and iteration
  - Examples of simple recursive functions
  - Geometric recursion: fractals
Big-O Review
Asymptotic Analysis

- Beyond number of operations
- Goal: understanding behavior of program over the long run, with increasingly large inputs
- We are not concerned with constants factors:
  - How many iterations?
  - Not operations in each iteration
- Gives a useful approximation, suppresses details
- Worst-case
Order of Complexity

- We express this as the (time) order of complexity.
- Normally expressed using Big-O notation.
- Big-0 is ignores constants, focuses on highest power of \( n \).

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( 3n+3 )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( 2n+8 )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>
Linear Search: Worst Case

# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None

Total: \(3n+3\)
# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        n iterations
        if list[index] == key:
            return index
        index = index + 1
    return None
$O(n)$ ("Linear")

Number of Operations

$n$ (amount of data)

2n + 8

3n + 3

n
For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).
For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.

4 = O(1)

1 = O(1)
Insertion Sort: worst case

# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):
        move_left(list,i)   # n-1 iterations
        i = i + 1
    return list

Total cost: cost of move_left as i goes from 1 to n-1

Simplest form including of all the move_lefts:

\[(5n^2 - 5n) / 2 = (5/2)n^2 - (5/2)n\]
<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$(5/2)n^2 - (1/2)n$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$2n^2 + 7$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Usually doesn’t matter what the constants are… we are only concerned about the highest power of $n$.

$f(n)$ is $O(g(n))$ means $f(n) < g(n) \cdot k$ for some positive $k$
$O(n^2)$ ("Quadratic")

Number of Operations

- $2n^2 + 7$
- $n^2$
- $n^2/2 + 3n/2 - 1$

$n$ (amount of data)
For a quadratic algorithm, if you double the amount of data, the amount of work you do quadruples (approximately).
Two Examples

- Linear Sort \( O(n) \) linear
- Insertion Sort \( O(n^2) \) quadratic
Big O

- $O(1)$ constant
- $O(\log n)$ logarithmic
- $O(n)$ linear
- $O(n \log n)$ log linear
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $O(2^n)$ exponential
How did we calculate Insertion Sort?
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):
        move_left(list, i)
        i = i + 1
    return list
# move_left

def move_left(a, i):
    x = a.pop(i)
    j = i - 1
    while j >= 0 and a[j] > x:
        j = j - 1
    a.insert(j + 1, x)

but how long do pop and insert take?
def move_left(a, i):
    x = a.pop(i)  # n iterations
    j = i - 1
    while j >= 0 and a[j] > x:  # i iterations
        j = j - 1
    a.insert(j + 1, x)  # n iterations
Insertion Sort: what is the cost of move_left?

```python
# let n = the length of list.
def move_left(a, i):
    x = a.pop(i)  # n iterations
    j = i - 1
    while j >= 0 and a[j] > x:  # i iterations
        j = j - 1
    a.insert(j + 1, x)  # n iterations
```

Total cost (at most): $n + i + n$

But what is $i$? To find out, look at isort, which calls move_left, supplying a value for $i$
Insertion Sort: what is the cost of the whole thing?

# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):  # n-1 iterations
        move_left(list, i)  # i goes from 1 to n-1
        i = i + 1
    return list

Total cost: cost of move_left as i goes from 1 to n-1

Cost of all the move_lefts: n + 1 + n
+ n + 2 + n
+ n + 3 + n
... 
+ n + n-1 + n
Figuring out the sum

- \( n + 1 + n \)  
- \( + n + 2 + n \)  
- \( + n + 3 + n \)  
- ...  
- \( + n + n-1 + n \)  

\[ (n-1) \times 2n \]

+ 1  
+ 2  
+ 3  
...  
+ n-1
Adding 1 through n-1

(6 * 7) / 2
blue circles
Adding 1 through n-1

- We saw \(1 + 2 + ... + 6 = (6 * 7) / 2\)

- Generalizing, \(1 + 2 + ... + n-1 = (n-1)(n) / 2\)

- So our whole cost is:

\[
(n-1)*2n + 1 + 2 + 3 ... + n-1
= (n-1)*2n + (n-1)(n) / 2
= 2n^2 - 2n + (n^2 - n) / 2
= (5n^2 - 5n) / 2 = (5/2)n^2 - (5/2)n
\]

- Observe that the highest-order term is \(n^2\)
Insertion Sort

- Worst Case: $O(n^2)$
Survey

https://goo.gl/forms/4Pr52SM0JcRhUKP02

Will post to Piazza
Recursion

The Loopless Loop
A recursive function is one that calls itself.

```python
def i_am_recursive(x):
    maybe do some work
    if there is more work to do:
        i_am_recursive(next(x))
    return the desired result
```

Infinite loop? Not necessarily, not if `next(x)` needs less work than `x`. 
Recursive Definitions

- Every recursive function definition includes two parts:
  - **Base case(s) (non-recursive)**
    One or more simple cases that can be done directly or immediately
  - **Recursive case(s)**
    One or more cases that require solving “simpler” version(s) of the original problem.
    - By “simpler”, we mean “smaller” or “shorter” or “closer to the base case”.
Example: Factorial

- \( n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \)

\[
egin{align*}
2! &= 2 \times 1 & 9! &= 362,880 \\
3! &= 3 \times 2 \times 1 & 10! &= 3,628,800 \\
4! &= 4 \times 3 \times 2 \times 1 & 10! &= 10 \times 9!
\end{align*}
\]

- alternatively:

\[\text{(Recursive case)}\]

\[\text{(Base case)}\]

\[
egin{align*}
0! &= 1 \\
n! &= n \times (n-1)! \\
\text{So } 4! &= 4 \times 3! & 3! &= 3 \times 2! & 2! &= 2 \times 1! \\
1! &= 1 \times 0! & 0! &= 1
\end{align*}
\]
Recursion conceptually

\[ 4! = 4(3!) \]
\[ 3! = 3(2!) \]
\[ 2! = 2(1!) \]
\[ 1! = 1 (0!) \]

make smaller instances of the same problem

Base case
Recursion conceptually

4! = 4(3!)
  3! = 3(2!)
  2! = 2(1!)
  1! = 1 (0!) = 1(1) = 1

Compute the base case

make smaller instances of the same problem
Recursion conceptually

4! = 4(3!)
3! = 3(2!)
2! = 2(1!) = 2
1! = 1 (0!) = 1(1) = 1

Compute the base case

make smaller instances of the same problem

build up the result
Recursion conceptually

Compute the base case

\[
4! = 4(3!)
\]
\[
3! = 3(2!)
\]
\[
2! = 2(1!)
\]
\[
1! = 1 (0!) = 1(1) = 1
\]

make smaller instances of the same problem

build up the result
Recursion conceptually

\[ 4! = 4(3!) \]
\[ 3! = 3(2!) \]
\[ 2! = 2(1!) \]
\[ 1! = 1 \]
\[ 0! = 1(1) = 1 \]

Compute the base case

make smaller instances of the same problem

build up the result
Recipe for Writing Recursive Functions
(by Dave Feinberg)

1. Write if. (Why?)
   There must be at least 2 cases: base and recursive

2. Handle simplest case(s).
   No recursive call needed (base case).

3. Write recursive calls(s).
   Input is slightly simpler to get closer to base case.

4. Assume the recursive call works!
   Ask yourself: What does it do?
   Ask yourself: How does it help?
Recursive Factorial in Python

```python
# Assumes n >= 0
def factorial(n):
    if n == 0:  # base case
        return 1
    else:       # recursive case
        result = factorial(n-1)
        return n * result
```

0! = 1  (Base case)
n! = n × (n-1)!  (Recursive case)
factorial(4)?
n=4 factorial(4)?
n=4    \( \text{factorial}(4)\) = 4 \(*\) \( \text{factorial}(3)\)
n=4 \quad \text{factorial}(4)? = 4 \times \text{factorial}(3)

n=3 \quad \text{factorial}(3)?
n=4  \[ \text{factorial}(4) = 4 \times \text{factorial}(3) \]

n=3  \[ \text{factorial}(3) = 3 \times \text{factorial}(2) \]
n=4  \text{factorial}(4) = 4 \times \text{factorial}(3)

n=3  \text{factorial}(3) = 3 \times \text{factorial}(2)

n=2  \text{factorial}(2)
n=4 \quad \text{factorial}(4)? = 4 \times \text{factorial}(3)

n=3 \quad \text{factorial}(3)? = 3 \times \text{factorial}(2)

n=2 \quad \text{factorial}(2)? = 2 \times \text{factorial}(1)
\[
\begin{align*}
\text{n=4} & \quad \text{factorial}(4)? = 4 \times \text{factorial}(3) \\
\text{n=3} & \quad \text{factorial}(3)? = 3 \times \text{factorial}(2) \\
\text{n=2} & \quad \text{factorial}(2)? = 2 \times \text{factorial}(1) \\
\text{n=1} & \quad \text{factorial}(1)?
\end{align*}
\]
n=4  \( \text{factorial}(4) = 4 \times \text{factorial}(3) \)

n=3  \( \text{factorial}(3) = 3 \times \text{factorial}(2) \)

n=2  \( \text{factorial}(2) = 2 \times \text{factorial}(1) \)

n=1  \( \text{factorial}(1) = 1 \times \text{factorial}(0) \)
factorial(4)? = 4 * factorial(3)

factorial(3)? = 3 * factorial(2)

factorial(2)? = 2 * factorial(1)

factorial(1)? = 1 * factorial(0)

factorial(0) = 1
n=4 \quad \text{factorial}(4) = 4 \times \text{factorial}(3)

n=3 \quad \text{factorial}(3) = 3 \times \text{factorial}(2)

n=2 \quad \text{factorial}(2) = 2 \times \text{factorial}(1)

n=1 \quad \text{factorial}(1) = 1 \times 1 = 1
n=4 \quad \text{factorial}(4)? = 4 \times \text{factorial}(3)

n=3 \quad \text{factorial}(3)? = 3 \times \text{factorial}(2)

n=2 \quad \text{factorial}(2) = 2 \times 1 = 2
factorial(4)? = 4 * factorial(3)

n=3 \hspace{1cm} \text{factorial(3) = 3 * 2 = 6}
n=4 \quad \text{factorial}(4) = 4 \times 6 = 24
Recursive vs. Iterative Solutions

- For every recursive function, there is an equivalent iterative solution.
- For every iterative function, there is an equivalent recursive solution.

- But some problems are easier to solve one way than the other way.
- And be aware that most recursive programs need space for the stack, behind the scenes.
Factorial Function (Iterative)

def factorial(n):
    result = 1  # initialize accumulator var
    for i in range(1, n+1):
        result = result * i
    return result

Versus (Recursive):

def factorial(n):
    if n == 0:  # base case
        return 1
    else:  # recursive case
        return n * factorial(n-1)
A Strategy for Recursive Problem Solving (hat tip to Dave Evans)

- Think of the smallest size of the problem and write down the solution (base case)

- **Be optimistic.** Assume you magically have a working function to solve any size. How could you use it on a smaller size and use the answer to solve a bigger size? (recursive case)

- Combine the base case and the recursive case
Mathematicians have proved \( \pi^2/6 = 1 + 1/4 + 1/9 + 1/16 + \ldots \)

We can use this formula to approximate \( \pi \):

Compute the sum, multiply by 6, take the square root

```python
def pi_series_iter(n):
    result = 0
    for i in range(1, n+1):
        result = result + 1/(i**2)
    return result

def pi_approx_iter(n):
    x = pi_series_iter(n)
    return (6*x)**(.5)
```

Let's convert this to a recursive function (see file pi_approx.py for a sample solution.)
Recursion on Lists

Do we know how to use iteration to sum the elements in a list?
Recursion on Lists

First we need a way of getting a smaller input from a larger one:

- Forming a sub-list of a list:

  ```python
  >>> a = [1, 11, 111, 1111, 11111, 111111]
  >>> a[1:]    # the "tail" of list a
  [11, 111, 1111, 11111, 111111]
  >>> a[2:]   
  [111, 1111, 11111, 111111]
  >>> a[3:]   
  [1111, 11111, 111111]
  >>> a[3:5]  
  [1111, 11111]
  ```
Recursive sum of a list

def sumlist(items):

    if :

        What is the smallest size list?
Recursive sum of a list

```python
def sumlist(items):
    if items == []:
        What is the sum of an empty list?
        The smallest size list is the empty list.
```

What is the sum of an empty list?
def sumlist(items):
    if items == []:
        return 0

Base case:
The sum of an empty list is 0.
def sumlist(items):
    if items == []:
        return 0
    else:
        Recursive case: the list is not empty
Recursive sum of a list

def sumlist(items):
    if items == []:
        return 0
    else:
        ...
        sumlist(   ) ...

What is a simpler/smaller case?
Recursive sum of a list

def sumlist(items):
    if items == []:
        return 0
    else:
        ... sumlist(items[1:]) ...
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])

What if we already know the sum of the list's tail?
We can just add in the list's first element!
Tracing `sumlist`

```python
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

```python
>>> sumlist([2,5,7])
sumlist([2,5,7]) = 2 + sumlist([5,7])
    5 + sumlist([7])
        7 + sumlist([])
            0
```

After reaching the base case, the final result is built up by the computer by adding 0+7+5+2.
List Recursion: exercise

- Let's create a recursive function \( \text{rev}(\text{items}) \)

- **Input**: a list of items

- **Output**: another list, with all the same items, but in reverse order

- **Remember**: it's usually sensible to break the list down into its *head* (first element) and its *tail* (all the rest). The tail is a smaller list, and so "closer" to the base case.

- Soooo... (picture on next slide)
Reversing a list: recursive case

see file rev_list.py
Multiple Recursive Calls

- So far we've used just one recursive call to build up our answer.

- The real **conceptual** power of recursion happens when we need more than one!

- Example: Fibonacci numbers
Fibonacci Numbers

A sequence of numbers:

0
1
1
2
3
5
8
13
...

...
Fibonacci Numbers in Nature

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.

- Number of branches on a tree, petals on a flower, spirals on a pineapple.

- Vi Hart's video on Fibonacci numbers (http://www.youtube.com/watch?v=ahXI MUkSXX0)
Let $\text{fib}(n)$ = the $n$th Fibonacci number, $n \geq 0$

- $\text{fib}(0) = 0$ (base case)
- $\text{fib}(1) = 1$ (base case)
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), n > 1$

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Two recursive calls!
Recursive Call Tree

\[
\begin{align*}
\text{fib}(5) & \\
\text{fib}(4) & \\
\text{fib}(3) & \\
\text{fib}(2) & \\
\text{fib}(1) & \\
\text{fib}(0) & = 0 \\
fib(1) & = 1 \\
fib(n) & = fib(n-1) + fib(n-2), \ n > 1
\end{align*}
\]
Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1
def fib(n):
    x = 0
    next_x = 1
    for i in range(1, n+1):
        old_x = x
        x = next_x
        next_x = old_x + x
    return x

sequence:
0
+ 1
+ 1
+ 2
+ 3
+ 5
+ 8
+ 13
...
Simultaneous Assignment

Assign values to multiple variables in a single statement:

\[
\begin{align*}
\text{sum, diff} &= x + y, x - y \\
x, y &= y, x
\end{align*}
\]
Iterative Fibonacci

```python
def fib(n):
    x = 0
    next_x = 1
    for i in range(1, n+1):
        x, next_x = next_x, x + next_x
    return x
```

Faster than the recursive version. Why?
Geometric Recursion (Fractals)

- A recursive operation performed on successively smaller regions.

Sierpinski's Triangle

http://fusionanomaly.net/recursion.jpg
Sierpinski’s Triangle
Sierpinski’s Carpet
(the next slide shows an animation that could give some people headaches)
Mandelbrot set

Fancier fractals