ORDERS OF GROWTH 4

COMPUTER SCIENCE 61A

July 2, 2015

1 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: if the size of the input grows, how does the runtime of the function change? And what do we mean by ”runtime”? Let’s look at the following examples first:

```
def square(n):
    return n * n

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

- `square(1)` requires one primitive operation: `*` (multiplication). `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes one operation.

input	function call	return value	number of operations
1	square(1)	1*1	1
2	square(2)	2*2	1
⋮	⋮	⋮	⋮
100	square(100)	100*100	1
⋮	⋮	⋮	⋮
n	square(n)	n*n	1

- `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases linearly proportional to the input.

input	function call	return value	number of operations
1	factorial(1)	1*1	1
2	factorial(2)	2*1*1	2
⋮	⋮	⋮	⋮
100	factorial(100)	100*99*...*1*1	100
⋮	⋮	⋮	⋮
n	factorial(n)	n*(n-1)*...*1*1	n

Big-O notation is a way to denote an upper bound on the complexity of a function. For example, $O(n^2)$ states that a function's run time will be **no larger than the quadratic** of the input.

- If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input n, then the runtime of this function is $O(n^3)$. As n gets larger, the lower order terms (10, $5n$, and $3n^2$) all become insignificant compared to n^3.
- If a function requires $5n$ operations with a given input n, then the runtime of this function is $O(n)$. The constant 5 only influences the runtime by a constant amount. In other words, the function still runs in linear time. Therefore, it doesn't matter that we drop the constant.

1.1 Kinds of Growth

Here are some common orders of growth, ranked from no growth to fastest growth:

- $O(1)$ — constant time takes the same amount of time regardless of input size
- $O(\log n)$ — logarithmic time
- $O(n)$ — linear time
- $O(n^2)$, $O(n^3)$, etc. — polynomial time
- $O(2^n)$ — exponential time (considered "intractable"; these are really, really horrible)

When using big-O notation, we always want to find the "tightest bound". Recall that `factorial(n)` requires `n` multiplications. Its technically correct to say that `factorial(n)` is in $O(n^2)$, since $n^2 >= n$ for all values of positive values of `n`, but its not very informative. Instead, we want to find the smallest big-O that `factorial(n)` belongs to. Since our implementation of `factorial(n)` must use at least `n` multiplications in all cases, we say its tightest bound is $O(n)$.

1.2 Questions

1. What is the order of growth in time for the following functions? Use big-O notation.

```
def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

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

3.
```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

4.
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

5.
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

6.
```
def bar(n):
    if n % 2 == 1:
```

```
        return n + 1
    return n

def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) + foo(n - 2)
    else:
        return 1 + foo(n - 2)
```

What is the order of growth of `foo(bar(n))`?

1.3 Extra Questions

1. Previously, we looked at the `is_prime` function. Here's the code for it:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

What is the order of growth of `is_prime`?

How can we change `is_prime` so that it runs in $O(\sqrt{n})$?

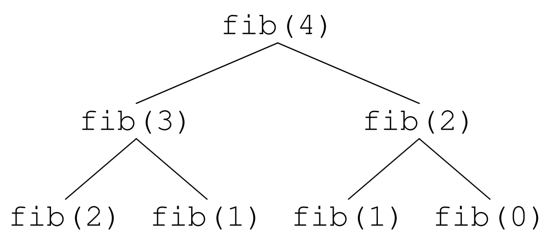
```
def is_prime(n):
```

2 Review of Tree Recursion

Consider a function that requires more than one recursive call. A simple example is a function that computes Fibonacci numbers:

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

This type of recursion is called *tree recursion*, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

2.1 Questions

1. I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume `n` is positive.

   Before we start, what's the base case for this question? What is the simplest input?

   What does `count_stair_ways(n - 1)` represent? What does `count_stair_ways(n - 2)` represent?

   Use those two recursive calls to write the recursive case:

```
def count_stair_ways(n):
```

2. Here's a part of the Pascal's trangle:

```
Item:     0    1    2    3    4    ...
Row 0:    1
Row 1:    1    1
Row 2:    1    2    1
Row 3:    1    3    3    1
Row 4:    1    4    6    4    1
...
```

Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it, use `0` if the entry is empty. Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle.

```
def pascal(row, column):
```

3. The TAs want to print handouts for their students. However, for some unfathomable reason, both the printers are broken; the first printer only prints multiples of `n1`, and the second printer only prints multiples of `n2`. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

   First try to solve without a helper function. Also try to solve using a helper function and adding up to the sum.

```
def has_sum(sum, n1, n2):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 1(5) + 0(3) = 5
    True
    >>> has_sum(11, 3, 5) # 2(3) + 1(5) = 11
    True
    """
```

2.2 Extra Questions

1. The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \leq x \leq 60$, and Printer B prints a random y copies $130 \leq y \leq 140$. The TAs also relax their expectations: they are satisfied as long as they get at least `lower`, but no more than `upper`, copies printed. (More than `upper` copies is unacceptable because it wastes too much paper.)

   Hint: Try using a helper function.

```
def sum_range(lower, upper):
    """
    >>> sum_range(45, 60) # Printer A prints within this range
    True
    >>> sum_range(40, 55) # Printer A can print a number 56-60
    False
    >>> sum_range(170, 201) # Printer A + Printer B will print
    ...         # somewhere between 180 and 200 copies total
    True
    """
```