CS 61A
Fall 2017

Growth and Nonlocals

Discussion 4: September 27, 2017

1 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by "runtime"?

- `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)	$1 \cdot 1$	1
2	square(2)	$2 \cdot 2$	1
⋮	⋮	⋮	⋮
100	square(100)	$100 \cdot 100$	1
⋮	⋮	⋮	⋮
n	square(n)	$n \cdot n$	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)	$1 \cdot 1$	1
2	factorial(2)	$2 \cdot 1 \cdot 1$	2
⋮	⋮	⋮	⋮
100	factorial(100)	$100 \cdot 99 \cdots 1 \cdot 1$	100
⋮	⋮	⋮	⋮
n	factorial(n)	$n \cdot (n-1) \cdots 1 \cdot 1$	n

For expressing complexity, we use what is called big Θ (Theta) notation. For example, if we say the running time of a function `foo` is in $\Theta(n^2)$, we mean that the running time of the process will grow proportionally with the square of the size of the input as it increases to infinity.

- **Ignore lower order terms:** If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input n, then the runtime of this function is $\Theta(n^3)$. As n gets larger, the lower order terms (10, $5n$, and $3n^2$) all become insignificant compared to n^3.
- **Ignore constants:** If a function requires $5n$ operations with a given input n, then the runtime of this function is $\Theta(n)$. We are only concerned with how the runtime grows asymptotically with the input, and since 5n is still asymptotically linear; the constant factor does not make a difference in runtime analysis.

Kinds of Growth

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

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

In addition, some programs will never terminate if they get stuck in an infinite loop.

Questions

What is the order of growth for the following functions?

1.1
```
def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

1.2
```
def fib_recursive(n):
   if n == 0 or n == 1:
      return n
   else:
      return fib_recursive(n - 1) + fib_recursive(n - 2)
```

1.3
```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

1.4
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

1.5
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

1.6
```
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))`?

2 Nonlocal

Until now, you've been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1 # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with `nonlocal` variables:

- **Global variables** cannot be modified using the `nonlocal` keyword.
- **Variables in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal variable with the same names in a single frame.

Questions

2.1 Draw the environment diagram for the folowing code.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

2.2 (Fall 2016) Draw the environment diagram for the following code.

```
lamb = 'da'
def da(da):
    def lamb(lamb):
        nonlocal da
        def da(nk):
            da = nk + ['da']
            da.append(nk[0:2])
            return nk.pop()
    da(lamb)
    return da([[1], 2]) + 3

da(lambda da: da(lamb))
```

2.3 Write a function that updates and prints a value `x` based on input functions.

```
def memory(n):
    """
    >>> f = memory(10)
    >>> f = f(lambda x: x * 2)
    20
    >>> f = f(lambda x: x - 7)
    13
    >>> f = f(lambda x: x > 5)
    True
    """
```

3 Mutable Lists

Let's imagine you order a mushroom and cheese pizza from La Val's, and that they represent your order as a list.

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all La Val's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
```

```
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i` (does not replace element but adds a new one)
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `pop(i)` removes and returns the element at index `i`

Questions

3.1 Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*. It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

(a)
```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2      # compares each value
```

(b)
```
>>> lst1 is lst2      # compares references
```

(c)
```
>>> lst2 = lst1
>>> lst1.append(4)
>>> lst1
```

(d)
```
>>> lst2
```

(e)
```
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
```

(f)
```
>>> lst1
```

(g)
```
>>> lst2
```

(h)
```
>>> lst2 is lst1
```

3.2 Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

3.3 Reverse a list *in place*, i.e. mutate the given list itself, instead of returning a new list.

```
def reverse(lst):
    """ Reverses lst in place.
    >>> x = [3, 2, 4, 5, 1]
    >>> reverse(x)
    >>> x
    [1, 5, 4, 2, 3]
    """
```