GROWTH AND DATA ABSTRACTION 4

COMPUTER SCIENCE 61A

July 5, 2016

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	factorial(100)	100*99*...*1*1	100
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	factorial(n)	n*(n-1)*...*1*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.

- 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.
- If a function requires $5n$ operations with a given input n, then the runtime of this function is $\Theta(n)$. 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:

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

1.2 Questions

What is the order of growth for the following functions?

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

> **Solution:** $\Theta(n^2)$, we will call factorial n times with arguments $n, n-1, n-2, ..., 0$. The sum from 0 to n is approximately n^2.

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

> **Solution:** $\Theta(\Phi^n)$, where Φ is the golden ratio. As long as you understand the runtime is exponential in n, we would accept your answer.

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

> **Solution:** $\Theta(n)$, since the while loop executes n times with each iteration taking a constant $\Theta(1)$ time.

4.
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

> **Solution:** $\Theta(\log(n))$, because our while loop iterates at most $\log(n)$ times, due to n being halved in every iteration.

5. ```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

**Solution:** $\Theta(1)$, since at worst it will require 6 recursive calls to reach the base case. So this is $\Theta(6)$, which can be reduced to $\Theta(1)$.

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))`?

Solution: $\Theta(n^2)$

2 Ready. Set. Abstract!

So far, we've only *used* data abstractions. Now let's try *creating* some! In the next section, we'll be looking at two ways of implementing abstract data types: lists and functions.

2.1 Let's Go Shopping

One way to implement abstract data types is with the Python list construct.

```
>>> nums = [1, 2]
>>> nums[0]
1
>>> nums[1]
2
```

We use the square bracket notation to access the data we stored in `nums`. The data is *zero indexed*: we access the first element with `nums[0]` and the second with `nums[1]`.

Let's try to simulate going shopping at the supermarket. Let's first build the *item* abstraction, which represents something you could buy at the supermarket.

1. Write the constructor and selectors for the *item* abstraction. An item consists of a string name and an integer price.

   ```
   def make_item(name, price):
   ```

   Solution:
   ```
       return [name, price]
   ```

   ```
   def get_item_name(item):
   ```

   Solution:
   ```
       return item[0]
   ```

   ```
   def get_item_price(item):
   ```

   Solution:
   ```
       return item[1]
   ```

2. Write `purchase`, which calculates the total price of a list of items.

   ```
   def purchase(items):
   ```

Solution:

```
    total_cost = 0
    for item in items:
        total_cost += get_item_price(item)
    return total_cost

    # List comprehension solution
    return sum([get_item_price(item) for item in items])
```

3 Functional Pairs

A second way of constructing abstract data types is with higher-order functions. We can implement the functions `pair` and `select` to achieve the same result as a list.

```
>>> def pair(x, y):
...     """Return a function that represents a pair of data."""
...     def get(index):
...         if index == 0:
...             return x
...         elif index == 1:
...             return y
...     return get
>>> def select(p, i):
...     """Return the element at index I of pair P."""
...     return p(i)
>>> nums = pair(1, 2)
>>> select(nums, 0)
1
>>> select(nums, 1)
2
```

Note how although using functional pairs is syntactically different from using lists, they accomplish the exact same thing.

3.1 EXTREME Couponing

Coupons are a great way to save money at the supermarket! Intuitively, a coupon associates an item with a discount.

1. Implement the coupon abstraction using functional pairs. A coupon pairs a string item name with an integer amount of discount representing a deduction from the

cost.

```
def make_coupon(name, discount):
```

Solution:

```
    return pair(name, discount)
```

```
def get_coupon_name(coupon):
```

Solution:

```
    return select(coupon, 0)
```

```
def get_coupon_discount(coupon):
```

Solution:

```
    return select(coupon, 1)
```

2. Write discount_purchase, which calculates the total price of a list of items after being discounted by a list of coupons.

```
def discount_purchase(items, coupons):
```

Solution:

```
    total_cost = 0
    for item in items:
        total_cost += get_item_price(item)
        for coupon in coupons:
            if get_coupon_name(coupon) == get_item_name(
               item):
                total_cost -= get_coupon_discount(coupon)
    return total_cost
```

3.2 Data Abstraction Violations

Data abstraction violations happen when we assume we know something about how our data is represented. For example, if we use pairs and we forget to use a selector and instead use the index.

```
>>> butter = make_item('Butter', 2)
>>> print(butter[0]) # violation!!!!
Butter
```

In this example, we assume that `butter` is represented as a list because we use the square bracket indexing. However, we should have used the selector `get_item_name`. This is a data abstraction violation.

Finally, it's time to buy groceries at the supermarket. A supermarket is a list of items.

1. Implement the *supermarket* abstraction.

```
def make_supermarket(items):
```

Solution:

```
    return items
```

```
def get_supermarket_items(supermarket):
```

Solution:

```
    return supermarket
```

2. Write the `shopping` function. Given a grocery list (a list of string item names), a supermarket, and coupon list, calculate the total cost of purchasing every item on the grocery list. You may use any functions you've already written.

```
def shopping(grocery_list, supermarket, coupons):
    """
    >>> grocery_list = ["Butter", "Bread"]
    >>> items = [create_item("Butter", 2),
    ...          create_item("Bread", 3),
    ...          create_item("Soup", 4)]
    >>> supermarket = make_supermarket(items)
    >>> coupons = [create_coupon("Butter", 1),
    ...            create_coupon("Soup", 1)]
    >>> shopping(grocery_list, supermarket, coupons)
    4
    """
```

Solution:

```
    items = []
    for item_name in grocery_list:
        for item in get_supermarket_items(supermarket):
            if item_name == get_item_name(item):
                items.append(item)
    return discount_purchase(items, coupons)
```