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:

```python
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: \( \times \) (multiplication). `square(100)` also requires one. No matter what input \( n \) we pass into `square`, it always takes one operation.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>square(1)</td>
<td>1*1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>square(2)</td>
<td>2*2</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>square(100)</td>
<td>100*100</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>square(n)</td>
<td>n*n</td>
<td>1</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>factorial(1)</td>
<td>1*1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>factorial(2)</td>
<td>2<em>1</em>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>factorial(100)</td>
<td>100<em>99</em>...<em>1</em>1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>factorial(n)</td>
<td>n*(n-1)*...<em>1</em>1</td>
<td>n</td>
</tr>
</tbody>
</table>

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 Θ(n²), 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 \(Θ(n^3)\). As \(n\) gets larger, the lower order terms (10, 5n, and 3n²) 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 \(Θ(n)\). 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:

- \(Θ(1)\) — constant time takes the same amount of time regardless of input size
- \(Θ(log n)\) — logarithmic time
- \(Θ(n)\) — linear time
- \(Θ(n^2), \ Θ(n^3), \ etc.\) — polynomial time
- \(Θ(2^n)\) — 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)

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 bonk(n):
   total = 0
   while n >= 2:
       total += n
       n = n / 2
   return total
5. def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 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 \( \text{foo}(\text{bar}(n)) \)?
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.

```python
>>> 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.
   ```python
def make_item(name, price):
    def get_item_name(item):
    def get_item_price(item):
```

2. Write `purchase`, which calculates the total price of a list of items.
   ```python
def purchase(items):
```
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.

```python
>>> 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.

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

def get_coupon_name(coupon):

def get_coupon_discount(coupon):
```
2. Write `discount_purchase`, which calculates the total price of a list of items after being discounted by a list of coupons.

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

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.

```python
>>> 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.

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

```python
def get_supermarket_items(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.

```python
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
    ""
```