Lecture #19: Complexity and Orders of Growth, contd.

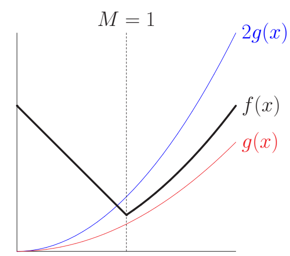
The Notation

- Suppose that f is a one-parameter function on real numbers.
- $O(f)$: functions that *eventually grow no faster than f:*
  - $g \in O(f)$ means that $|g(x)| \le C_g \cdot |f(x)|$ for all $x \ge M_g$
  - where C_g and M_g are constants, generally different for each g.
- $\Omega(f)$: functions that *eventually grow at least as fast as f*:
  - $g \in \Omega(f)$ means that $f \in O(g)$,
  - so that $|f(x)| \le C_f|g(x)|$ for all $x > M_f$, and so
  - $|g(x)| \ge \frac{1}{C_f}|f(x)|$.
- $\Theta(f)$: functions that *eventually grow as g grows:*
  - $\Theta(f) = O(f) \cap \Omega(f)$, so that
  - $g \in \Theta(f)$ means that $\frac{1}{C_f}|f(x)| \le |g(x)| \le C_g \cdot |f(x)|$ for all sufficiently large x.

The Notation (II)

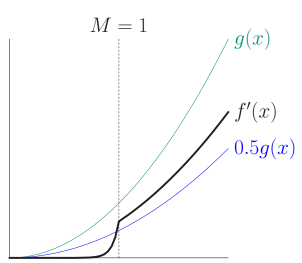
- So $O(f)$, $\Omega(f)$, and $\Theta(f)$ are *sets of functions*.
- If $E_1(x)$ and $E_2(x)$ are two expressions involving x, we usually abbreviate $\lambda x : E_1(x) \in O(\lambda x : E_2(x))$ as just $E_1(x) \in O(E_2(x))$. For example, $n+1 \in O(n^2)$.
- I write $f \in O(g)$ where others write $f = O(g)$, because the latter doesn't make sense.

Illustration



- Here, $f \in O(g)$ ($p = 2$, see blue line), even though $f(x) > g(x)$. Likewise, $f \in \Omega(g)$ ($p = 1$, see red line), and therefore $f \in \Theta(g)$.
- That is, $f(x)$ is eventually (for $x > M = 1$) no more than proportional to $g(x)$ and no less than proportional to $g(x)$.

Illustration, contd.



- Here, $f' \in \Omega(g)$ ($p = 0.5$), even though $g(x) > f'(x)$ everywhere.

Other Uses of the Notation

- You may have seen $O(\cdot)$ notation in math, where we say things like

$$f(x) \in f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + O(x^3), \text{ for } 0 \le x < a.$$

- Adding or multiplying sets of functions produces sets of functions. The expression to the right of $\in$ above means "the set of all functions g such that

$$g(x) = f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + h(x)$$

  where $h(x) \in O(x^3)$."

Example: Linear Search

- Consider the following search function:

```
def near(L, x, delta):
    """True iff X differs from some member of sequence L by no
    more than DELTA."""
    for y in L:
        if abs(x-y) <= delta:
            return True
    return False
```

- There's a lot here we don't know:
  - How long is sequence L?
  - Where in L is x (if it is)?
  - What kind of numbers are in L and how long do they take to compare?
  - How long do abs and subtract take?
  - How long does it take to create an iterator for L and how long does its __next__ operation take?
- So what can we meaningfully say about complexity of near?

What to Measure?

- If we want general answers, we have to introduce some "strategic vagueness."
- Instead of looking at times, we can consider number of "operations." Which?
- The total time consists of
  1. Some fixed overhead to start the function and begin the loop.
  2. Per-iteration costs: subtraction, abs, __next__, <=
  3. Some cost to end the loop.
  4. Some cost to return.
- So we can collect total operations into one "fixed-cost operation" (items 1, 3, 4), plus M_L "loop operations" (item 2), where M_L is the number of items in L up to and including the y that come within delta of x (or the length of L if no match).

What Does an "Operation" Cost?

- But these "operations" are of different kinds and complexities, so what do we really know?
- Assuming that each operation represents some range of possible minimum and maximum values (constants), we can say that

$$\begin{array}{l} \text{min\_fixed\_cost} + M(L) \times \text{min\_loop\_cost} \\ \leq \\ C_{\text{near}}(L) \\ \leq \\ \text{max\_fixed\_cost} + M(L) \times \text{max\_loop\_cost} \end{array}$$

where $C_{\text{near}}(L)$ is the cost of near on list L, and $M(L)$ is the number of items near must look at.

Best/Worst Cases

- We can simplify by not trying to give results for particular inputs, but instead giving summary results for *all inputs of the same "size."*
- Here, "size" depends on the problem: could be magnitude, length (of list), cardinality (of set), etc.
- Since we don't consider specific inputs, we have to be less precise.
- Typically, the figure of interest is the *worst case over all inputs of the same size.*
- Since $M(L) \leq \text{len}(L)$, $C_{\text{near}}(L) \leq \text{len}(L) \times \text{max\_loop\_cost}$.
- So if we let $C_{\text{wc}}(N)$ mean "worst-case cost of near over all lists of size N," we can conclude that

$$C_{\text{wc}}(N) \in O(N)$$

.

Best of the Worst

- But in addition, it's also clear that $C_{\text{wc}}(N) \in \Omega(N)$.
- So we can say, most concisely, $C_{\text{wc}}(N) \in \Theta(N)$.
- Generally, when a worst-case time is not $\Theta(\cdot)$, it indicates either that
  - We don't know (haven't proved) what the worst case really is, just put limits on it, or
    - ∗ Most often happens when we talk about the worst-case for a *problem:* "what's the worst case for the best possible algorithm?"
  - We know what the worst-case time is, but it's not an easy formula, so we settle for approximations that are easier to deal with.

Example: Nested Loop

- Last time, we saw the worst-case $C_{\text{ad}}(N)$ of the nested loop

```
for i, x in enumerate(L):
    for j, y in enumerate(L, i+1): # Starts at i+1
        if x == y: return True
```

is $\Theta(N^2)$ (where N is the length of L).

Example: A Tricky Nested Loop

- What can we say about $C_{\text{iu}}(N)$, the worst-case cost of this function (assume pred counts as one constant-time operation):

```
def is_unduplicated(L, pred):
    """True iff the first x in L such that pred(x) is not
    a duplicate. Also true if there is no x with pred(x)."""
    i = 0
    while i < len(L):
        x = L[i]
        i += 1
        if pred(x):
            while i < len(L):
                if x == L[i]:
                    return False
                i += 1
    return True
```

- ? In this case, despite the nested loop, we read each element of L at most once. So $C_{\text{iu}}(N) \in \Theta(N)$.

Some Useful Properties

In the following, K, k, K_i, and k_i are constants, and $N \geq 0$.

- $\Theta(K_0N + K_1) = \Theta(N)$
- $\Theta(N^k + N^{k-1}) = \Theta(N^k)$
  - ▷ $|N^k| \leq |N^k + N^{k-1}| \leq 2N^k$ for $N > 1$.
- $\Theta(|f(N)| + |g(N)|) = \Theta(\max(|f(N)|, |g(N)|))$
  - ▷ $\max(|f(N)|, |g(N)|) \leq |f(N)| + |g(N)| \leq 2\max(|f(N)|, |g(N)|)$.
- $\Theta(\log_a N) = \Theta(\log_b N)$
  - ▷ $\log_a N = \log_a b \cdot \log_b N$. (As a result, we usually use $\log_2 N = \lg N$ for all logarithms.)
- $\Theta(f(N) + g(N)) \neq \Theta(\max(f(N), g(N)))$
  - ▷ Consider $f(N) = -g(N)$.
- $O(N^{k_1}) \subset O(k_2^N)$, if $k_2 > 1$.
  - ▷ $\lg N^{k_1} = k_1 \lg N$, $\lg k_2^N = (\lg k_2)N$, and $k_1 \lg N < \frac{k_1}{k_2} \cdot k_2 \cdot N$ for $N > 0$.

Fast Growth

- Here's a bad way to see if a sequence appears (consecutively) in another sequence:

```
def is_substring(sub, seq):
    """True iff SUB[0], SUB[1], ... appear consecutively in sequence SEQ."""
    if len(sub) == 0 or sub == seq:
        return True
    elif len(sub) > len(seq):
        return False
    else:
        return is_substring(sub, seq[1:]) or is_substring(sub, seq[:-1])
```

- Suppose we count the number of times is_substring is called.
- Then time depends only on D=len(seq)-len(sub).
- Define $C_{\text{is}}(D)$ = worst-case time to compute is_substring.
- Looking at cases: $D \leq 0$ and $D > 0$:

$$C_{\text{is}}(D) = \begin{cases} 1, & \text{if } D \leq 0 \\ 2C_{\text{is}}(D-1)+1, & otherwise. \end{cases}$$

Fast Growth (II)

- To solve:

$$C_{\text{is}}(D) = \begin{cases} 1, & \text{if } D \leq 0 \\ 2C_{\text{is}}(D-1)+1, & \text{otherwise.} \end{cases}$$

- Expand repeatedly:

$$\begin{aligned} C_{\text{is}}(D) &= 2C_{\text{is}}(D-1)+1 \\ &= 2(2C_{\text{is}}(D-2)+1)+1 \\ &= 2(2(\ldots(D(0)+1)+1)+\ldots+1)+1)+1 \\ &= 2(2(\ldots(1+1)+1)+\ldots+1)+1)+1 \\ &= 2^D + 2^{D-1} + \ldots + 1 \\ &= 2^{D+1} - 1 \\ &\in O(2^D) \end{aligned}$$

Slow Growth

- A perhaps-familiar technique:

```
def binary_search(L, x):
    """Return True iff X occurs in sorted list L."""
    low, high = 0, len(L)
    while low < high:
        m = (low + high) // 2
        if x < L[m]: high = m
        if x > L[m]: low = m+1
        else:        return True
    return False
```

- The value of high-low is halved on each iteration, starting from N, the length of L, so counting loop iterations in the worst case:

$$C_{\text{bs}}(N) = \begin{cases} 0, & \text{if } N \leq 0; \\ 1 + C_{\text{bs}}(N/2), & \text{otherwise.} \end{cases}$$

- So

$$C_{\text{bs}}(N) = 1 + C_{\text{bs}}(N/2) = 1 + 1 + C_{\text{bs}}(N/4) = \cdots \in \Theta(\lg N)$$

Some Intuition on Meaning of Growth

- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size N (assuming perfect scaling and that problem size 1 takes 1μsec).
- Entries show the *size of problem* that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.
- N = problem size

Time (μsec) for problem size N	Max N Possible in 1 second	1 hour	1 month	1 century
$\lg N$	10^{300000}	$10^{1000000000}$	$10^{8 \cdot 10^{11}}$	$10^{9 \cdot 10^{14}}$
N	10^6	$3.6 \cdot 10^9$	$2.7 \cdot 10^{12}$	$3.2 \cdot 10^{15}$
$N \lg N$	63000	$1.3 \cdot 10^8$	$7.4 \cdot 10^{10}$	$6.9 \cdot 10^{13}$
N^2	1000	60000	$1.6 \cdot 10^6$	$5.6 \cdot 10^7$
N^3	100	1500	14000	150000
2^N	20	32	41	51