Lecture #17: Complexity and Orders of Growth

Public-Service Announcement

"Cal Habitat for Humanity is a service club dedicated to giving back to our wonderful community through volunteering. Between building houses, assisting with soup kitchens, gardening, and selling food, we strive to make a positive impact for our neighborhood and beyond. You can commit as little (0) or as many (10,000) hours as you would like to the club, but we could really use the help of volunteers such as you. If this seems like something you'd be interested in, join us at our next meeting on March 7th at 126 Barrows 7pm or visit our website at calhabitat.org."

Complexity

- Certain problems take longer than others to solve, or require more storage space to hold intermediate results.
- We refer to the *time complexity* or *space complexity* of a problem.
- But what does it mean to say that a certain *program* has a particular complexity?
- What does it mean for an *algorithm*?
- What does it mean for a *problem*?

A Direct Approach

- Well, if you want to know how fast something is, you can time it.
- Python happens to make this easy:

```
>>> def fib(n):
...     if n <= 1: return n
...     else: return fib(n-2) + fib(n-1)
...
>>> from timeit import repeat
>>> repeat('fib(10)', 'from __main__ import fib', number=5)
[0.000491..., 0.000486..., 0.000487...]
>>> repeat('fib(20)', 'from __main__ import fib', number=5)
[0.060..., 0.060..., 0.060...]
>>> repeat('fib(30)', 'from __main__ import fib', number=5)
[7.74..., 7.81..., 7.81...]
```

- `repeat(Stmt, Setup, number=N)` says

  Execute Setup (a string containing Python code), then execute Stmt (a string) N times. Repeat this process 3 times and report the time required for each repetition.

A Direct Approach, Continued

- You can also use this from the command line:

```
$ python3 -m timeit --setup='from fib import fib' 'fib(10)'
10000 loops, best of 3: 97 usec per loop
```

- This command automatically chooses a number of executions of fib to give a total time that is large enough for an accurate average, repeats 3 times, and reports the best time.

Strengths and Problems with Direct Approach

- Good: Gives actual times; answers question completely for given input and machine.
- Bad: Results apply only to tested inputs.
- Bad: Results apply only to particular programs and platforms.
- Bad: Cannot tell us anything about complexity of algorithm or of problem.

But Can't We Extrapolate?

- Why not try a succession of times, and use that to figure out timing in general?

```
...# for t in 5 10 15 20 25 30; do
>     echo -n "$t: "
>     python3 -m timeit --setup='from fib import fib' "fib($t)"
> done
5: 100000 loops, best of 3: 8.16 usec per loop
10: 10000 loops, best of 3: 96.8 usec per loop
15: 1000 loops, best of 3: 1.08 msec per loop
20: 100 loops, best of 3: 12 msec per loop
25: 10 loops, best of 3: 133 msec per loop
30: 10 loops, best of 3: 1.47 sec per loop
```

- This looks to be exponential in t with exponent of ≈ 1.6.
- But... what if the program special-cases some inputs?
- ...and this still only works for a particular program and machine.

Worst Case, Best Case, Average Case

- To avoid the problem of getting results only for particular inputs, we usually ask a more general question, such as:
  - What is the *worst case* time to compute f(X) as a function of the size of X, or
  - what is the *average case* time to compute f(X) over all values of X (weighted by likelihood).
- Here, "size" depends on the problem: could be magnitude, length (of list), cardinality (of set), etc.
- Also makes sense to talk about the *best case* over all inputs of the same size (but this is not usually interesting) or the *average case* over all inputs of the same size, weighted by likelihood (but this is hard in general).
- But now we seem to have a harder problem than before: how do we get worst-case times? Doesn't that require testing all cases?
- And when we do, aren't we still sensitive to machine model, compiler, etc.?

Example: Linear Search

- Consider the following search function:

```python
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 comes 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

$$\begin{aligned} & \textit{min-fixed-cost} + M(\text{L}) \times \textit{min-loop-cost} \\ \le\ & \\ & C_{\text{near}}(L) \\ \le\ & \\ & \textit{max-fixed-cost} + M(\text{L}) \times \textit{max-loop-cost} \end{aligned}$$

where $C_{\text{near}}(L)$ is the cost of `near` on a list where the program has to look at $M(L)$ items.

- In the worst case M(L) $==$ len(L) and in the best, M(L) ≤ 1, so

$\textit{min-fixed-cost} \le C_{\text{near}}(L) \le \textit{max-fixed-cost} + \text{len(L)} \times \textit{max-loop-cost}$.

- Simpler, but still clumsy, and the numbers are not going to be precise anyway. Would be nice to have a cleaner notation.

Operation Counts and Scaling

- Instead of getting precise answers in units of physical time, we therefore settle for a proxy measure that will remain meaningful over changes in architecture or compiler.
- Choose some operations of interest and count how many times they occur.
- Examples:
  - How many times does fib get called recursively during computation of fib(N)?
  - How many addition operations get performed by fib(N)?
- You can no longer get precise times, but if the operations are well-chosen, results are *proportional* to actual time for different values of N.
- Thus, we look at how computation time *scales* in the worst case.
- Can compare programs/algorithms on the basis of which scale better.

Asymptotic Results

- Sometimes, results for "small" values are not indicative.
- E.g., suppose we have a prime-number tester that contains a look-up table of the primes up to 1,000,000,000 (about 50 million primes).
- Tests for numbers up to 1 billion will be faster than for larger numbers.
- So in general, we tend to ask about *asymptotic* behavior of programs: as size of input goes to infinity.

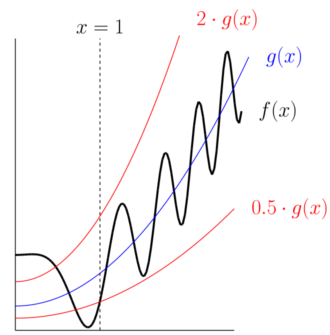
Expressing Approximation

- So, we are looking for measures of program performance that give us a sense of how computation time scales with size of input.
- And we are further interested in ignoring finite sets of special cases that a given program can compute quickly.
- Finally, precise worst-case functions can be very complicated, and the precision is generally not terribly important anyway.
- These considerations motivate the use of *order notation* to express how approximations of execution time or space grow.

The Notation

- Suppose that f is a function of one parameter returning real numbers.
- We use the notation $\Theta(g)$ to mean "the set of all one-parameter functions whose absolute values are *eventually bounded* above and below by some *constant multiples* of g's absolute value."
- So we can write $f \in \Theta(g)$ to mean "whenever n is large enough, $p \cdot |g(n)| \le |f(n)| \le q \cdot |g(n)|$ for some positive constants p and q."
- This notation can be used to express the growth rate of any function.
- In this course, the functions we are interested in are those that express the time a computation takes as a function of problem size.

Illustration



- Here, $f \in \Theta(g)$ because once x is large enough ($x > 1$), $|f(x)|$ is always between two multiples of $|g(x)|$: $0.5 \cdot |g(x)| \le |f(x)| \le 2 \cdot |g(x)|$.

Notational Quirks

- We usually write things like

$$x^2 + 1 \in \Theta(x^2)$$

as shorthand for

$$\lambda x : x^2 + 1 \in \Theta(\lambda x : x^2)$$

- Adding or multiplying sets of functions produces sets of functions. Thus, $x^2 + \Theta(g)$ means "the set of functions of x returning $x^2 + h(x)$, where $h \in \Theta(g)$."
- I prefer $f \in \Theta(g)$ or $f(x) \in x^2 + \Theta(g(x))$ to the traditional $f = \Theta(g)$ or $f(x) = x^2 + \Theta(g(x))$, because '=' makes no formal sense here (the left side denotes a function and the right denotes a set of functions.)

Using Asymptotic Estimates

- Going back to the `near` function,

$$\begin{aligned} &\textit{min-fixed-cost} + M(L) \times \textit{min-loop-cost} \\ \le\ & C_{\text{near}}(L) \\ \le\ & \textit{max-fixed-cost} + M(L) \times \textit{max-loop-cost} \end{aligned}$$

where $M(L)$ is the number of items in L that are examined before the loop terminates.

- *In the worst case,* $M(L) = N$, where N is the length of L.
- So, letting $C^{\text{wc}}_{\text{near}}(N)$ mean "the worst-case value of $C_{\text{near}}(L)$ when N is the length of L:"

$$\begin{aligned} &\textit{min-fixed-cost} + N \times \textit{min-loop-cost} \\ \le\ & C^{\text{wc}}_{\text{near}}(N) \\ \le\ & \textit{max-fixed-cost} + N \times \textit{max-loop-cost} \end{aligned}$$

- Claim: we can state this more cleanly as

$$C^{\text{wc}}_{\text{near}}(N) \in \Theta(N).$$

Using Asymptotic Estimates

- Claim that

  $$\begin{aligned} & \textit{min-fixed-cost} + N \times \textit{min-loop-cost} \\ \leq\; & C^{\rm wc}_{\rm near}(N) \\ \leq\; & \textit{max-fixed-cost} + N \times \textit{max-loop-cost} \end{aligned}$$

  means that

  $$C^{\rm wc}_{\rm near}(N) \in \Theta(N).$$

  Why?

- Well, if we ignore the two fixed costs (assume they are 0), we obviously fit the definition, since for $N \geq 0$,

  $$p \cdot N \leq C^{\rm wc}_{\rm near}(N) \leq q \cdot N,$$

  where p is min-loop-cost and q is max-loop-cost.

- It's easy to see that by tweaking q up a bit—e.g., to q', where

  $$q' = q + \textit{max-fixed-cost}$$

  we can arrange that when N is big enough ($N > 1$ for this particular p'), we cover the necessary range.

Typical $\Theta(\cdot)$ Estimates from Programs

<table>
<tr><th>Bound on Worst-Case Time</th><th>Example</th></tr>
<tr><td>$\Theta(1)$</td><td><pre>x += L[c]</pre></td></tr>
<tr><td>$\Theta(\lg N)$</td><td><pre>while N > 0:
    x, N = x + L[N], N // 2</pre></td></tr>
<tr><td>$\Theta(N)$</td><td><pre>for c in range(N):
    x += L[c]</pre></td></tr>
<tr><td>$\Theta(N \lg N)$</td><td><pre>def sort(L):   # Define N = len(L)
    M = len(L) // 2
    if M == 0: return L # Assume merge takes Θ(N)
    else: return merge(sort(L[:M]), sort(L[M:]))</pre></td></tr>
<tr><td>$\Theta(N^2)$</td><td><pre>for c in range(N):   # Executed N times.
    for d in range(N): # Executed N times for each c
        x += L[c][d]   # Executed N x N times.</pre></td></tr>
<tr><td>$\Theta(2^N)$</td><td><pre>def longMax(A, L, U):   # Define N = U-L; L<=U
    if L == U: return A[L]
    else: return max(longMax(A, L+1, U),
                     longMax(A, L, U-1))</pre></td></tr>
</table>

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.

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