Lecture #22: Complexity and Orders of Growth

- 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)
...
>>> import timeit
>>> timeit.repeat('fib(10)', 'from __main__ import fib', number=5)
[0.0004911422729492188, 0.0004868507385253906, 0.00048708915710449922]
timeit.repeat('fib(20)', 'from __main__ import fib', number=5)
[0.06009697914123535, 0.06010794639587402, 0.06009793281555176]
>>> timeit.repeat('fib(20)', 'from __main__ import fib', number=5)
[0.06009697914123535, 0.06010794639587402, 0.06009793281555176]
```

- `timeit.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, 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).
- Average case is hard, so we'll let other courses deal with it.
- 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.?

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 operation(s) 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 approximations of execution time or space.

The Notation

- Suppose that f is a function of one parameter returning real numbers.
- We use the notation $O(f)$ to mean "the set of all one-parameter functions whose absolute values are eventually bounded above by some multiple of f's absolute value." Formally:

$$O(f) = \{g \mid \text{there exist } p, M \textit{ such that if } x > M,\ |g(x)| \le p|f(x)|\}$$

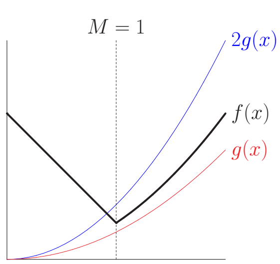
- Similarly, we have "the set of all one-parameter functions whose absolute values are eventually bounded below by some multiple of f's absolute value:"

$$\Omega(f) = \{g \mid \text{there exist } p > 0, M \textit{ such that if } x > M,\ |g(x)| \ge p|f(x)|\}$$

- And finally those bounded both above and below:

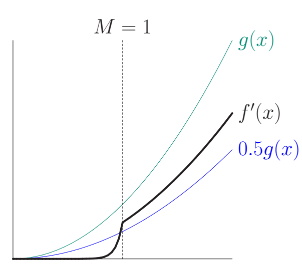
$$\Theta(f) = \Omega(f) \cap O(f)$$

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.

Uses of the Notation

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

$$f(x) \in f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + O(f'''(0)x^3)$$

- Adding or multiplying sets of functions produces sets of functions. The one above means "the set of all functions g(x) such that

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

  where $h(x) \in O(f'''(0)x^3)$.""

- I prefer $\in$ to the traditional $=$, since the latter makes no formal sense.