Lecture #19: Complexity and Orders of Growth
Annoucements

• Sign up for alternative test #2 times within next two weeks
  http://goo.gl/forms/VhI4rHw3LW.

• Hog Contest Results Up!
  http://cs61a.org/proj/hog_contest/results/.
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:

```python
>>> 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?

```bash
...# 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 \( \approx 1.6 \).

• 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.?
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

\[
\text{min-fixed-cost} + M(L) \times \text{min-loop-cost} \\
\leq \\
C_{\text{near}}(L) \\
\leq \\
\text{max-fixed-cost} + M(L) \times \text{max-loop-cost}
\]

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 \(M(L) = \text{len}(L)\) and in the best, \(M(L) \leq 1\), so

\[
\text{min-fixed-cost} \leq C_{\text{near}}(L) \leq \text{max-fixed-cost} + \text{len}(L) \times \text{max-loop-cost}.
\]

• 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 operation(s) of interest and count how many times they occur.

- **Examples:**
  - How many times does \texttt{fib} get called recursively during computation of \texttt{fib}(N)?
  - How many addition operations get performed by \texttt{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.
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 $1\mu$sec).
- 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

<table>
<thead>
<tr>
<th>Time ((\mu)sec) for problem size $N$</th>
<th>1 second</th>
<th>Max $N$ Possible in 1 hour</th>
<th>1 month</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lg N)</td>
<td>10(^{300000})</td>
<td>10(^{10000000000})</td>
<td>10(^{8\cdot10^{11}})</td>
<td>10(^{9\cdot10^{14}})</td>
</tr>
<tr>
<td>$N$</td>
<td>10(^6)</td>
<td>3.6 \cdot 10(^9)</td>
<td>2.7 \cdot 10(^{12})</td>
<td>3.2 \cdot 10(^{15})</td>
</tr>
<tr>
<td>$N \lg N$</td>
<td>63000</td>
<td>1.3 \cdot 10(^8)</td>
<td>7.4 \cdot 10(^{10})</td>
<td>6.9 \cdot 10(^{13})</td>
</tr>
<tr>
<td>$N^2$</td>
<td>1000</td>
<td>60000</td>
<td>1.6 \cdot 10(^6)</td>
<td>5.6 \cdot 10(^7)</td>
</tr>
<tr>
<td>$N^3$</td>
<td>100</td>
<td>1500</td>
<td>14000</td>
<td>150000</td>
</tr>
<tr>
<td>$2^N$</td>
<td>20</td>
<td>32</td>
<td>41</td>
<td>51</td>
</tr>
</tbody>
</table>
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:
  \[
  O(f) = \{ g \mid |g(x)| \leq p \cdot |f(x)| \text{ when } x > M, \text{ for some constants } p, M \}
  \]

- So we can write \( g \in O(f) \) to mean “whenever \( n \) is large enough, \( |g(n)| \leq p \cdot |f(n)| \) for some constant \( p \).”

- We define
  \[
  f \in \Omega(g) \overset{\text{def}}{=} g \in O(f).
  \]
  or “\( f \) is eventually bounded below by a multiple of \( g \).”

- And finally those bounded both above and below:
  \[
  \Theta(f) = \Omega(f) \cap O(f)
  = \{ g \mid p_1 \cdot |f(x)| \leq p_2 \cdot |f(x)| \text{ when } x > M \}
  \]
  for some constants \( 0 < p_1 \leq p_2, \text{ and } M \).
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.
Notational Quirks

• 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. Thus, $x^2 + O(g)$ means "the set of functions of $x$ returning $x^2 + h(x)$, where $h \in O(g)$.

• Well, to be picky, we really ought to write

$$f \in (\lambda x.f(0) + f'(0)x + \ldots) + O(\lambda x.f'''(0)x^3)$$

but that really gets really tedious...

• ...so if $E(x)$ is some expression involving $x$, we usually abbreviate $\lambda x.E(x)$ as just $E(x)$. Example: $n + 1 \in O(n^2)$

• I prefer $\in$ to the traditional $f(x) = f(0) + \cdots$, since the latter makes no formal sense (the left side is a function and the right is a set of functions.)

• Finally, we will sometimes write $f \in \Theta(g)$ even when $f$ and $g$ are functions of something non-numeric (like lists).
Using Asymptotic Estimates

• Going back to linear search,
  
  \[ \text{min-fixed-cost} + M(L) \times \text{min-loop-cost} \leq C_{\text{near}}(L) \]
  \[\leq \text{max-fixed-cost} + M(L) \times \text{max-loop-cost} \]

• Claim: we can state this more cleanly as \( C_{\text{near}}(L) \in O(M(L)) \) and \( C_{\text{near}}(L) \in \Omega(M(L)) \), or even more concisely: \( C_{\text{near}}(L) \in \Theta(M(L)) \).

• Why? \( C_{\text{near}}(M(L)) \in O(M(L)) \) if \( C_{\text{near}}(M(L)) \leq K \cdot M(L) \) for sufficiently large \( M(L) \), by definition.

• And if if \( K_1 \) and \( K_2 \) are any (non-negative) constants, then \( K_1 + K_2 \cdot M(L) \leq (K_1 + K_2) \cdot M(L) \) for \( M(L) > 1 \).

• Likewise, \( K_1 + K_2 \cdot M(L) \geq K_2 \cdot M(L) \) for \( M > 0 \).

• And we can go even farther. If the sequence, \( L \), has length \( N(L) \), then we know that \( M(L) \leq N(L) \). Therefore, we can say \( C_{\text{near}}(L) \in O(N(L)) \).

• Is \( C_{\text{near}}(L) \in \Omega(N(L)) \)?
Using Asymptotic Estimates

• Going back to linear search,

\[ \text{min-fixed-cost} + M(L) \times \text{min-loop-cost} \leq C_{\text{near}}(L) \]
\[ \leq \text{max-fixed-cost} + M(L) \times \text{max-loop-cost} \]

• Claim: we can state this more cleanly as \( C_{\text{near}}(L) \in O(M(L)) \) and \( C_{\text{near}}(L) \in \Omega(M(L)) \), or even more concisely: \( C_{\text{near}}(L) \in \Theta(M(L)) \).

• Why? \( C_{\text{near}}(M(L)) \in O(M(L)) \) if \( C_{\text{near}}(M(L)) \leq K \cdot M(L) \) for sufficiently large \( M(L) \), by definition.

• And if if \( K_1 \) and \( K_2 \) are any (non-negative) constants, then \( K_1 + K_2 \cdot M(L) \leq (K_1 + K_2) \cdot M(L) \) for \( M(L) > 1 \).

• Likewise, \( K_1 + K_2 \cdot M(L) \geq K_2 \cdot M(L) \) for \( M > 0 \).

• And we can go even farther. If the sequence, \( L \), has length \( N(L) \), then we know that \( M(L) \leq N(L) \). Therefore, we can say \( C_{\text{near}}(L) \in O(N(L)) \).

• Is \( C_{\text{near}}(L) \in \Omega(N(L)) \)? No: can only say \( C_{\text{near}}(L) \in \Omega(1) \).
**Best/Worst Cases**

- We can simplify still further 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.*

- Also makes sense to talk about the *best case* over all inputs of the same size, or the *average case* over all inputs of the same size (weighted by likelihood). These are rarer, though.

- From preceding discussion, since $C_{\text{near}}(N(L)) \in O(N(L))$, it follows that $C_{\text{wc}}(N) \in O(N)$, where $C_{\text{wc}}(N)$ is “worst-case cost of near over all lists of size $N$.”
Best of the Worst

• We just saw that $C_{wc}(N) \in O(N)$.
• But in addition, it’s also clear that $C_{wc}(N) \in \Omega(N)$.
• So we can say, most concisely, $C_{wc}(N) \in \Theta(N)$.
• 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,
    * Most often happens when we talk about the worst-case for a problem: “what’s the worst case for the best possible algorithm?”
  - Or we know what the worst-case time is, but it’s messy, so we settle for approximations that are easier to deal with.