Lecture #17: Complexity and Orders of Growth

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)', globals=globals(), number=5)
[0.00029,..., 0.00027,..., 0.00027,...]
>>> repeat('fib(20)', globals=globals(), number=5)
[0.021..., 0.017..., 0.017...]
>>> repeat('fib(30)', globals=globals(), number=5)
[2.26..., 2.25..., 2.25...]
```

- repeat(Stmt, globals=globals(), number=N) means
  - Execute Stmt (a string) N times. Repeat this process 3 times and report the time required for each repetition. The 'globals' argument here makes sure that 'fib' is available.

A Direct Approach, Continued

- You can also use this from the command line (assuming that 'fib' is defined in file fib.py):

```
$ 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, platforms, and loads.
- **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? Here’s an example using the Unix shell:

```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
```

<table>
<thead>
<tr>
<th>t</th>
<th>Loops</th>
<th>Best Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100000</td>
<td>2.04 usec per loop</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>22.5 usec per loop</td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
<td>256 usec per loop</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>2.75 msec per loop</td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>30.6 msec per loop</td>
</tr>
<tr>
<td>30</td>
<td>10</td>
<td>345 msec per loop</td>
</tr>
</tbody>
</table>

- This is (very roughly) \(1.5t^{1.6}\) usec when \(t \geq 10\).
- But it still applies to a particular program and machine, and only looks at a few possible input values.

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 \(X\)?
  - What is the **average case** time to compute \(f(X)\) as a function of the size of \(X\)?
  - What is the **best case** time to compute \(f(X)\) as a function of the size of \(X\)?
- Here, "size" depends on the problem: could be magnitude of numeric input, number of digits, length (of list), cardinality (of set), etc.
- Average case can be hard and best case uninteresting. Here, we’ll mostly be interested in worst cases.
- 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)?
  - How long does it take to compare numbers \(L\)?
  - 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 \(\text{near}\)?
**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
  $$\min\text{-fixed-cost} + M(L) \times \min\text{-loop-cost} \leq C_{\text{near}}(L) \leq \max\text{-fixed-cost} + M(L) \times \max\text{-loop-cost}$$
  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 $M(L) = \|L\|$ and in the best, $M(L) = 0$, so
  $$\min\text{-fixed-cost} \leq C_{\text{near}}(L) \leq \max\text{-fixed-cost} + \|L\| \times \max\text{-loop-cost}.$$
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 \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 \textit{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 \textit{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 \textit{order notation} to express how approximations of execution time or space grow.

The Notation

- Suppose that \textit{f(n)} and \textit{g(n)} are functions returning real numbers.
- For us, \textit{f(n)} will generally be some kind of cost function—the execution time for a problem of size \textit{n}.
- We use the notation \(\Theta(g(n))\) to mean "the set of all functions whose absolute values are eventually proportional to \textit{g(n)}.
- We write \(f(n) \in \Theta(g(n))\) to mean "whenever \textit{n} is large enough, \(K_1|g(n)| \leq |f(n)| \leq K_2|g(n)|\) where \(0 < K_1 < K_2\) are some constants."
- In other words "\(|f(n)|\) is roughly proportional to \(|g(n)|\)."
- This notation can be used to express the growth rate of any function, but again, we're mostly interested in cost functions.
- (BTW: in most other places, you'll see this written as \(f(n) = \Theta(g(n))\), but I consider that nonsense, since \textit{f(n)} is a function and \(\Theta(g(n))\) is a set of functions.)
Illustration

\[ f(x) = 2 \cdot g(x) \]
\[ g(x) \]
\[ 0.5 \cdot g(x) \]

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

Using Asymptotic Estimates (I)

- Going back to the near function,
  \[
  \text{min-fixed-cost} + M(L) \times \text{min-loop-cost} \\
  \leq C_{\text{near}}(L) \\
  \leq \text{max-fixed-cost} + M(L) \times \text{max-loop-cost}
  \]
  where \( M(L) \) 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 \( C_{\text{near}}^{\text{wc}}(N) \) mean "the worst-case value of \( C_{\text{near}}(L) \) when \( N \) is the length of \( L \):"
  \[
  \text{min-fixed-cost} + N \times \text{min-loop-cost} \\
  \leq C_{\text{near}}^{\text{wc}}(N) \\
  \leq \text{max-fixed-cost} + N \times \text{max-loop-cost}
  \]

Using Asymptotic Estimates (II)

- We can state
  \[
  \text{min-fixed-cost} + N \times \text{min-loop-cost} \\
  \leq C_{\text{near}}^{\text{wc}}(N) \\
  \leq \text{max-fixed-cost} + N \times \text{max-loop-cost}
  \]
  more cleanly as
  \[ C_{\text{near}}^{\text{wc}}(N) \in \Theta(N). \]

- Why?
  - Well, if we ignore the two fixed costs (assume they are 0), we obviously fit the definition, since for \( N \geq 0 \),
    \[ p \cdot N \leq C_{\text{near}}^{\text{wc}}(N) \leq q \cdot N, \]
    where \( p \) is min-loop-cost and \( q \) is max-loop-cost (both constants).
  - It’s easy to see that we can arrange that when \( N > 1 \), we cover the necessary range by tweaking \( q \) up a bit—e.g., to
    \[ q + \text{max-fixed-cost} \]

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

<table>
<thead>
<tr>
<th>Bound on Worst-Case Time</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant time</td>
<td>( x += L[c] )</td>
</tr>
<tr>
<td>( \Theta(1) )</td>
<td></td>
</tr>
<tr>
<td>Logarithmic time</td>
<td>( \text{while } N &gt; 0: ) \n</td>
</tr>
<tr>
<td>( \Theta(\log N) )</td>
<td></td>
</tr>
<tr>
<td>Linear time</td>
<td>( \text{for } c \text{ in range}(N): ) \n</td>
</tr>
<tr>
<td>( \Theta(N) )</td>
<td></td>
</tr>
</tbody>
</table>
Typical $\Theta(\cdot)$ Estimates from Programs (II)

<table>
<thead>
<tr>
<th>Bound on Worst-Case Time</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(N \log N)$</td>
<td><code>def sort(L):</code> # Define N = len(L)</td>
</tr>
<tr>
<td></td>
<td>[M = \lfloor \log_2 N \rfloor \times 2]</td>
</tr>
<tr>
<td></td>
<td>[\text{if } M == 0: \text{ return } L \text{ # Assume merge takes } \Theta(N)]</td>
</tr>
<tr>
<td></td>
<td>[\text{else: return merge(sort(L[:M]), sort(L[M:]))}]</td>
</tr>
</tbody>
</table>

Quadratic time $\Theta(N^2)$

| for c in range(N): \# Executed N times. |
| for d in range(N): \# Executed N times for each c |
| x += L[c][d] \# Executed $N \times N$ times. |

Exponential time $\Theta(2^N)$

| def longMax(A, L, U): \# Define N = U-L; L<=U |
| if L == U: \text{ return A[L]} |
| else: \text{ return max(longMax(A, L+1, U), longMax(A, L, U-1))} |

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.

<table>
<thead>
<tr>
<th>Time ($\mu$sec) for problem size $N$</th>
<th>1 second</th>
<th>Max $N$ Possible in</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 N$</td>
<td>$10^{10}$</td>
<td>$10^{1000000000}$</td>
</tr>
<tr>
<td>$N$</td>
<td>$10^6$</td>
<td>$3.6 \cdot 10^9$</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$63000$</td>
<td>$1.3 \cdot 10^8$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>$1000$</td>
<td>$60000$</td>
</tr>
<tr>
<td>$N^3$</td>
<td>$100$</td>
<td>$1500$</td>
</tr>
<tr>
<td>$2^N$</td>
<td>$20$</td>
<td>$32$</td>
</tr>
</tbody>
</table>

Efficiency and Complexity

- The term efficiency is often misused to describe what I've been discussing here.
- Efficiency is slightly different, however. We can define the efficiency of my program on a problem instance $P$ as the theoretically optimal time to compute result of $P$ over execution time of my program on $P$.
- In the absence of a known theoretical result, we can use the performance of some "good" program.
- But this does raise the question: How does one determine theoretical optimums?

Lower Bounds on Algorithms

- A result that says "No algorithm for problem $P$ can possibly have an asymptotic complexity of less than $\Theta(f(n))$" is called a lower-bound result.
- These are really hard to prove. You are basically saying "No matter how smart you are or how far technology advances, you'll never do better than this bound."
- A few are easy. For example, to add two integers, you cannot do any better than $\Theta(N)$, where $N$ is the total number of digits (why?).
- Many are unsolved. For example, if you can prove that $P \neq NP$, you will win several prestigious prizes and $\$$. (If you prove that instead $P = NP$, you could bring about the end of civilization as we know it, but that's another story.)