Efficiency

What it means

When to care

What to do about it
Two levels

**Asymptotic efficiency**

“shape” of the performance as function of problem size

**Performance tuning**

depending on machine, common problems, ...

*Both matter, but we address them at different points in software development*
Efficiency

Not quite the same as speed ...

I can increase speed by buying a faster computer
I want to judge efficiency of a program or an algorithm

I often want to know how speed relates to problem size
Example: Sorting

bogosort(s):
  for m in permutations of s:
    if m is in the correct order:
      return m

This is “correct”, but it’s the worst sorting algorithm I can imagine. How would you describe just how stinky it is?

(It works fine for really short lists)
Efficiency is a measure of time (or memory) required \textit{as a function of problem size} (ex: relation of operations in bogosort to size of the list to be sorted)

We don’t (usually) care about the exact time
We ask \textit{what it is proportional to}.

Bogosort’s efficiency is proportional to the \textbf{factorial of len(s)}. \textit{(Really, really bad.)}
A somewhat better sorting algorithm

insertion sort(s):

  t = empty list

for elem in s:

  insert s into t at the correct place

  (moving other items if necessary)

Operations in insertion sort are proportional to

len(s)^2. (We call it “quadratic”)

Much better than factorial. Still not good.
Linear search vs Binary search

Linear search: Operations proportional to the length of the dictionary ("linear")
Binary search: Operations proportional to $\log_2$ of length of dictionary

What if we sort it once at the beginning?

The best sorting algorithms take time proportional to \( \text{len}(s) \times \log_2 \text{len}(s) \)

(and of course Python uses one of those good algorithms)
Asymptotic complexity:

*Only worry about the dominant factor in the worst case, ignoring coefficients*

*Example:* \( 0.5x^2 + 200x + 5000 = O(x^2) \)

Common classes \((n \text{ is problem size, } k \text{ is constant})\):

\[
O(k) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)
\]

*constant \(< \text{logarithmic \(< \text{linear \(< \text{log-linear \(< \text{quadratic \(< \text{cubic}\)}}}

\[
O(n^k) < O(k^n)
\]

*polynomial \(< \text{exponential}\)*
# Examples

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log \log n )</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( \log n )</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>664</td>
<td>9,966</td>
<td>132,877</td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>10</td>
<td>32</td>
<td>100</td>
</tr>
<tr>
<td>( n )</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>10,000</td>
<td>1,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>1,000,000</td>
<td>1,000,000,000</td>
<td>1,000,000,000,000</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>1.E+30</td>
<td>1.E+301</td>
<td>(big)</td>
</tr>
</tbody>
</table>
What about smaller improvements?

“First make it run, then make it fast”
(variation: premature optimization is the root of all evil, or at least the root of many bugs)

Don’t make confusing code to speed it up
Do look for simple improvements that keep the code readable and understandable
Tuning vs. Asymptotic complexity

Asymptotic complexity first
  Initial algorithm design, for the worst case.

Tuning
  For the common case, based on measured performance, on available machines

Example: Python’s sort() function is optimized for “nearly sorted” list
  log-linear worst case, linear common case
Tuning vs. Asymptotic complexity

Asymptotic complexity first

- Initial algorithm design, for the worst case.

Tuning

- For the common case, based on measured performance, on available machines

Example: Python’s sort( ) function is optimized for “nearly sorted” list

- log-linear worst case, linear common case
Sometimes we can’t win in the worst case ...

Oh dear ... asymptotic complexity of Boggle is terrible. Typical of games (chess, checkers, sudoku, etc.) Solutions are “heuristic”: Not guaranteed to work efficiently.
Sudoku is NP-hard* (but small enough)

The best algorithm for solving Sudoku takes time $O(2^n)$
where $n$ is board size ... but the board size is fixed and small, so solutions can be “good enough”

* NP-hard: if there is a polynomial time algorithm for Sudoku, then there must be a polynomial time algorithm for many other problems whose best algorithm is currently exponential. (Most computer scientists believe all these problems are intractable, i.e., have no polynomial time solution.)
Live Coding exercise

Class mergeable
  wraps a sorted list
  provides a sorted insert operation
  optimized for insertion at the end
  provides an operation \( s.\text{merge}(t) \)
  with performance \( O(s + t) \)
  (linear)