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):

```python
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 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 what it is proportional to.

Bogosort’s efficiency is proportional to the factorial of len(s). (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 len(s) * log_2 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 is problem size, k is constant):
O(k) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3)
constant < logarithmic < linear < log-linear < quadratic < cubic
O(n^k) < O(k^n)
polynomial < exponential
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

Sometimes we can’t win in the worst case ...

<table>
<thead>
<tr>
<th>Board</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x</td>
<td>x</td>
</tr>
<tr>
<td>x x x x x</td>
<td>xx</td>
</tr>
<tr>
<td>x x x x x</td>
<td>xxx</td>
</tr>
<tr>
<td>x x x x x</td>
<td>xxx</td>
</tr>
<tr>
<td>x x x x x</td>
<td>xxx</td>
</tr>
<tr>
<td>x x x x x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

Oh dear ... asymptotic complexity of Boggle is terrible.
Typical of games (chess, checkers, sudoku, etc.)
Solutions are “heuristic”: Not guaranteed to work efficiently.
Live Coding exercise

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