Efficiency

Prof. Clarkson
Fall 2016

Today's music: Opening theme from *The Big O*
(THE ビッグオ)
by Toshihiko Sahashi
Review

Previously in 3110:

• Functional programming
• Modular programming and software engineering
• Interpreters

Today:

• Interlude on **efficiency** of programs
Question

Which of the following would you prefer?

A. $O(n^2)$
B. $O(\log(n))$
C. $O(n)$
D. They're all good
E. I thought this was 3110, not Algo
Which of the following would you prefer?

A. $O(n^2)$
B. $O(\log(n))$
C. $O(n)$
D. They're all good
E. I thought this was 3110, not Algo
What is "efficiency"?

**Attempt #1:** An algorithm is efficient if, when implemented, it runs in a small amount of time on particular input instances.

...problems with that?
What is "efficiency"?

**Attempt #1:** An algorithm is efficient if, when implemented, it runs in a small amount of time on particular input instances

Incomplete list of problems:

• Inefficient algorithms can run quickly on small test cases
• Fast processors and optimizing compilers can make inefficient algorithms run quickly
• Efficient algorithms can run slowly when coded sloppily
• Some input instances are harder than others
• Efficiency on small inputs doesn’t imply efficiency on large inputs
• Some clients can afford to be more patient than others; quick for me might be slow for you
Lessons learned from attempt #1

**Lesson 1:** Time as measured by a clock is not the right metric

- Want a metric that is reasonably independent of hardware, compiler, other software running, etc.

- **Idea:** number of steps taken (say, by small-step semantics) during evaluation of program
  - steps are independent of implementation details
  - but: each step might really take a different amount of time?
    - creating a closure, looking up a variable, computing an addition
  - in practice, the difference isn't really big enough to matter
Lessons learned from attempt #1

Lesson 2: Running time on particular input instances is not the right metric

• Want a metric that can predict running time on any input instance

• Idea: size of the input instance
  – make metric be a function of input size
  – (combined with lesson 1) specifically, the maximum number of steps for an input of that size
  – But: particular inputs of the same size might really take a different amount of time?
    • multiplying arbitrary matrices vs. multiplying by all zeros
  – in practice, size matters more
Lessons learned from attempt #1

Lesson 3: "Small" is too relative

• Want a metric that is reasonably objective; independent of subjective notions of what is fast

• **Okay idea:** beats brute-force search
  
  — *brute force:* enumerate all the answers one by one, check and see whether the answer is right
    
    • the simple, dumb solution to nearly any algorithmic problem
    
    • related idea: guess an answer, check whether correct e.g., bogosort
  
  — but *by how much* is enough to beat brute-force search?
Lessons learned from attempt #1

Lesson 3: "Small" is too relative

• **Better idea:** polynomial time
  – (combined with ideas from previous two lessons) can express maximum number of steps as a polynomial function of the size N of input, e.g.,
    • $aN^2 + bN + c$
  – But: some polynomials might be too big to be quick? e.g. $N^{100}$
  – But: some non-polynomials might be quick enough? e.g. $N^{1+.02 \log N}$
  – in practice, polynomial time really does work
What is "efficiency"?

**Attempt #2**: An algorithm is efficient if its maximum number of steps of execution is polynomial in the size of its input.

*let's give that a try...*
Analysis of running time

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2    key = A[j]
3    // Insert A[j] into the sorted sequence A[1 .. j - 1]
4    i = j - 1
5    while i > 0 and A[i] < key
6        A[i + 1] = A[i]
7        i = i - 1
8    A[i + 1] = key
```

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>n</td>
</tr>
<tr>
<td>c_2</td>
<td>n - 1</td>
</tr>
<tr>
<td>c_3</td>
<td>n - 1</td>
</tr>
<tr>
<td>c_4</td>
<td>n - 1</td>
</tr>
<tr>
<td>c_5</td>
<td>[\sum_{j=2}^{n} t_j]</td>
</tr>
<tr>
<td>c_6</td>
<td>[\sum_{j=2}^{n} (t_j - 1)]</td>
</tr>
<tr>
<td>c_7</td>
<td>[\sum_{j=2}^{n} (t_j - 1)]</td>
</tr>
<tr>
<td>c_8</td>
<td>n - 1</td>
</tr>
</tbody>
</table>

[Cormen et al.  *Introduction to Algorithms*, 3rd ed, 2009]
Analysis of running time

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes \( c_i \) steps to execute and executes \( n \) times will contribute \( c_i n \) to the total running time.\(^6\) To compute \( T(n) \), the running time of INSERTION-SORT on an input of \( n \) values, we sum the products of the cost and times columns, obtaining

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1).
\]

\[\text{[Cormen et al. Introduction to Algorithms, 3rd ed, 2009]}\]
Precision of running time

• Precise bounds are **exhausting to find**
• Precise bounds are to some extent **meaningless**
  – Are those constants $c_1..c_8$ really useful?
  – If it takes 25 steps in high level language, but compiled down to assembly would take 10x more steps, is the precision useful?
  – **Caveat**: if you're building code that flies an airplane or controls a nuclear reactor, you do care about precise, real-time guarantees
## Some simplified running times

<table>
<thead>
<tr>
<th>size of input</th>
<th>N=10</th>
<th>N=100</th>
<th>N=1,000</th>
<th>N=10,000</th>
<th>N=100,000</th>
<th>N=1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>3 hours</td>
</tr>
<tr>
<td>N²</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>12 days</td>
</tr>
<tr>
<td>N³</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>12 days</td>
<td>32 years</td>
<td>10^4 years</td>
</tr>
<tr>
<td>2^N</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>12 days</td>
<td>32 years</td>
<td>very long</td>
</tr>
</tbody>
</table>

max # steps as function of N

assuming 1 microsecond/step

very long = more years than the estimated number of atoms in universe
Simplifying running times

• Rather than $1.62N^2 + 3.5N + 8$ steps, we would rather say that running time "grows like $N^2$"
  – identify broad classes of algorithm with similar performance
• Ignore the low-order terms
  – e.g., ignore $3.5N+8$
  – Why? For big $N$, $N^2$ is much, much bigger than $N$
• Ignore the constant factor of high-order term
  – e.g., ignore $1.62$
  – Why? For classifying algorithms, constants aren't meaningful
    • Code run on my machine might be a constant factor faster or slower than on your machine, but that's not a property of the algorithm
    – Caveat: Performance tuning real-world code actually can be about getting the constants to be small!
• Abstraction to an imprecise quantity
Imprecise abstractions

• OCaml's `int` type is an abstraction of a subset of \( \mathbb{Z} \)
  – don't know which `int` when reasoning about the type of
    an expression
• \( \pm 1 \) is an abstraction of \{1,-1\}
  – don't know which when manipulating it in a formula
• Here's a new one: Big Ell
  – \( L(n) \) represents a natural number whose value is less than
    or equal to \( n \)
  – precisely, \( L(n) = \{m \mid 0 \leq m \leq n\} \)
  – e.g., \( L(5) = \{0, 1, 2, 3, 4, 5\} \)
Manipulating Big Ell

• What is $1 + L(5)$?
• Trick question!
  – Replace $L(5)$ with set: $1 + \{0..5\}$
  – But $+$ is defined on ints, not sets of ints
• We could distribute the $+$ over the set:
  $\{1+0, \ldots, 1+5\} = \{1..6\}$
  – That is, a set of values, one for each possible instantiation of $L(5)$
• Note that $\{1..6\} \subseteq \{0..6\} = L(6)$
• So we could say that $1 + L(5) \subseteq L(6)$
What is \( L(2) + L(3) \)?

*Hint: set of values, one for each possible instantiation of \( L(2) \) and of \( L(3) \)*

A. \( L(2) + L(3) \subseteq L(2) \)
B. \( L(2) + L(3) \subseteq L(3) \)
C. \( L(2) + L(3) \subseteq L(4) \)
D. \( L(2) + L(3) \subseteq L(5) \)
E. \( L(2) + L(3) \subseteq L(6) \)
Question

What is $L(2) + L(3)$?

Hint: set of values, one for each possible instantiation of $L(2)$ and of $L(3)$

A. $L(2) + L(3) \subseteq L(2)$
B. $L(2) + L(3) \subseteq L(3)$
C. $L(2) + L(3) \subseteq L(4)$
D. $L(2) + L(3) \subseteq L(5)$
E. $L(2) + L(3) \subseteq L(6)$
Question

What is $L(2) \ast L(3)$?

A. $L(2) \ast L(3) \subseteq L(2)$
B. $L(2) \ast L(3) \subseteq L(3)$
C. $L(2) \ast L(3) \subseteq L(4)$
D. $L(2) \ast L(3) \subseteq L(5)$
E. $L(2) \ast L(3) \subseteq L(6)$
Question

What is $L(2) \times L(3)$?

A. $L(2) \times L(3) \subseteq L(2)$
B. $L(2) \times L(3) \subseteq L(3)$
C. $L(2) \times L(3) \subseteq L(4)$
D. $L(2) \times L(3) \subseteq L(5)$
E. $L(2) \times L(3) \subseteq L(6)$
A little trickier...

What is $2^{L(3)}$?

- $L(3) = \{0..3\}$
- So $2^{L(3)}$ could be any of $\{2^0, ..., 2^3\} = \{1, 2, 4, 8\}$
- And $\{1,2,4,8\} \subseteq L(8) = L(2^3)$
- Therefore $2^{L(3)} \subseteq L(2^3)$

...we can use this idea of Big Ell to invent an imprecise abstraction for running times
Big Oh, version 1

- **Recall:** we're interested in running time as a function of input size
- **Recall:** $L(n)$ represents any natural number that is less than or equal to a natural number $n$
- "New" imprecise abstraction: Big Oh
  - **Intuition:** $O(g)$ represents any function that is less than or equal to function $g$, for every input $n$
  - Big Oh is a higher-order version of Big Ell: generalize from naturals to functions on naturals
- Why the naturals? We're assuming function inputs and outputs are non-negative:
  - These are functions on input size and running time
  - Those won't be negative
Big Oh, version 1

Definition: \( O(g) = \{ f | \forall n . f(n) \leq g(n) \} \)

e.g.
- \( O(fun n \to 2n) = \{ f | \forall n . f(n) \leq 2n \} \)
- \( (fun n \to n) \in O(fun n \to 2n) \)

Note: these are mathematical functions written in OCaml notation, not OCaml functions
Recall: we want to ignore constant factors

(fun n → n), (fun n → 2n), (fun n → 3n)

...all should be in O(fun n → n)

Revised intuition: O(g) represents any function that is less than or equal to function g times some positive constant c, for every input n
**Big Oh, version 2**

**Definition:** $O(g) = \{ f \mid \exists c>0 . \forall n . f(n) \leq c \ g(n) \}$

e.g.
- $O(fun n \rightarrow n^3) = \{ f \mid \exists c>0 \ \forall n . f(n) \leq cn^3 \}$
- $(fun n \rightarrow 3n^3) \in O(fun n \rightarrow n^3)$
  because $3n^3 \leq cn^3$, where $c = 3$ (or $c=4$, ...)

Recall: we care about what happens at scale

could just build a lookup table for inputs in the range 0..2
Recall: we care about what happens at scale

Revised intuition: $O(g)$ represents any function that is less than or equal to function $g$ times some positive constant $c$, for every input $n$ greater than or equal to some positive constant $n_0$
Big Oh, version 3

Definition:

\[ O(g) = \{ f | \exists c > 0, n_0 > 0 . \forall n \geq n_0 . f(n) \leq c \, g(n) \} \]

this is the important, final definition you should know!

e.g.:

- \[ O(\text{fun } n \rightarrow n^2) = \{ f | \exists c > 0, n_0 > 0 . \forall n \geq n_0 . f(n) \leq c n^2 \} \]
- \( (\text{fun } n \rightarrow 2n) \in O(\text{fun } n \rightarrow n^2) \) because \( 2n \leq cn^2 \), where \( c = 2 \), for all \( n \geq 1 \)
Big Oh Notation: Warning 1

Instead of
\[ O(g) = \{f \mid \ldots \} \]
most authors write
\[ O(g(n)) = \{f(n) \mid \ldots \} \]

• They don't really mean \( g \) applied to \( n \); they mean a function \( g \) parameterized on input \( n \) but not yet applied
• Maybe they never studied functional programming 😊
Big Oh Notation: Warning 2

Instead of

\[(\text{fun } n \rightarrow 2n) \in O(\text{fun } n \rightarrow n^2)\]

all authors write

\[2n = O(n^2)\]

• Your instructor has always found this abusage distressing...
• Yet henceforth he will conform to the convention 😊
• The standard defense is that = should be read here as "is" not as "equals"
• Be careful: one-directional "equality"!
A Theory of Big Oh

- reflexivity: \( f = O(f) \)
- \((no\ symmetry\ condition\ for\ Big\ Oh)\)
- transitivity: if \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \)
- \( c \cdot O(f) = O(f) \)
- \( O(c \cdot f) = O(f) \)
- \( O(f) \cdot O(g) = O(f \cdot g) \)
  \( \text{where } f \cdot g \text{ means } (\text{fun } n \rightarrow f(n) \cdot g(n)) \)

Useful to know these equalities so that you don't have to keep re-deriving them from first principles
What is "efficiency"?

**Final attempt:** An algorithm is efficient if its worst-case running time on input size $N$ is $O(N^d)$ for some constant $d$. 
Running times of some algorithms

- **O(1): constant:** access an element of an array (of length n)
- **O(log n): logarithmic:** binary search through sorted array of length n
- **O(n): linear:** maximum element of list of length n
- **O(n log n): linearithmic:** mergesort a list of length n
- **O(n^2): quadratic:** bubblesort an array of length n
- **O(n^3): cubic:** matrix multiplication of n-by-n matrices
- **O(2^n): exponential:** enumerate all integers of bit length n

...some of these are not obvious, require proof
Asymptotic bounds

Big Oh:

- asymptotic upper bound
- \( O(g) = \{ f | \exists c > 0, n_0 > 0 . \forall n \geq n_0 . f(n) \leq c g(n) \} \)
- intuitions: \( f \leq g \), \( f \) is at least as efficient as \( g \)
Asymptotic bounds

Big Omega

- asymptotic lower bound
- \( \Omega(g) = \{ f \mid \exists c > 0, n_0 > 0 . \forall n \geq n_0 . f(n) \geq c \ g(n) \} \)
- intuitions: \( f \geq g \), \( f \) is at most as efficient as \( g \)
Asymptotic bounds

Big Theta

- asymptotic tight bound
- \( \Theta(g) = O(g) \cap \Omega(g) \)
- \( \Theta(g) = \{ f \mid \exists c_1 > 0, c_2 > 0, n_0 > 0. \forall n \geq n_0 . c_1 g(n) \leq f(n) \leq c_2 g(n) \} \)
- intuitions: \( f = g \), \( f \) is just as efficient as \( g \)
- beware: some authors write \( O(g) \) when they really mean \( \Theta(g) \)
Asymptotic bounds

\[ f(n) = \Theta(g(n)) \quad (a) \]

\[ f(n) = O(g(n)) \quad (b) \]

\[ f(n) = \Omega(g(n)) \quad (c) \]

[Cormen et al. *Introduction to Algorithms*, 3rd ed, 2009]
Alternative notions of efficiency

• **Expected-case** running time
  – Instead of worst case
  – Useful for randomized algorithms
  – Maybe less useful for deterministic algorithms
    • Unless you really do know something about probability distribution of inputs
    • All inputs are probably not equally likely

• **Space**
  – How much memory is used? Cache space? Disk space?

• **Other resources**
  – Power, network bandwidth, ...
Upcoming events

• [this week] nothing

This is efficient.

THIS IS 3110