CS 3110

Efficiency

Nate Foster

Spring 2020

Today's music: Patience by Tame Impala

CLICKER QUESTION 1

Review

Previously in 3110:

- Functional programming
- Modular programming and software engineering

New unit of course: Efficiency

Today:

- What it means to be efficient
- Big-Oh notation

WHAT IS EFFICIENCY?

Credit: Kleinberg and Tardos, *Algorithm Design*, chapter 2, 2006.

What is efficiency?

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

Exercise: write down three problems with that definition.

Lessons learned from attempt #1

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

Idea: Use number of "steps" taken during evaluation

What counts as a step?

Steps

- Any kind of primitive unit of computation inside a function
- Should be machine independent
- Examples:
  - Pseudocode: one line
  - Imperative language: assignment, array index, pointer dereference, arithmetic operation, etc.
  - OCaml: apply an arithmetic operator or constructor, substitute a let-binding, choose a branch of if/match, etc.

Lessons learned from attempt #1

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

Idea: Use "size" of the input instance

How to measure size?

Size

- Some representation of how big input is compared to other inputs
- Examples:
  - Number of elements in list or array
  - Number of bits in number
  - Number of nodes and edges in a graph
  - Etc.

Lessons learned from attempt #1

Lesson 3: "Small" is too relative

Okay idea: beats *brute-force* search

Lessons learned from attempt #1

Lesson 3: "Small" is too relative

Better idea: Polynomial time

Number of steps is a polynomial function of the input size:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

Objections to polynomial time

- Some polynomials might be too big?
  e.g. N^{100}
- Some non-polynomials might be fine?
  e.g. $N^{1+.02(\log N)}$

- But in practice, it just works

What is efficiency?

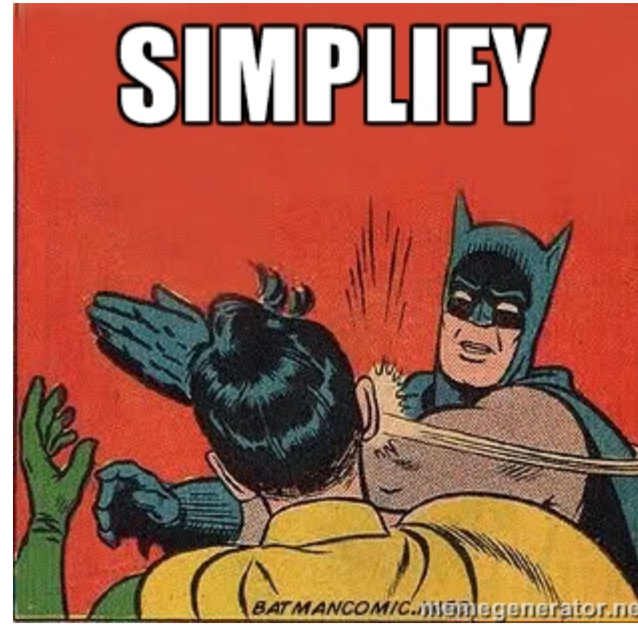
Attempt #2: An algorithm is efficient if its maximum number of steps of execution is polynomial in the size of its input.

let's give that a try...

[Cormen et al. *Introduction to Algorithms*, 3rd ed, 2009]

Analysis of running time

INSERTION-SORT(A)	*cost*	*times*
for j = 2 **to** A.length	c_1	n
key = *A*[*j*]	c_2	$n - 1$
// Insert *A*[*j*] into the sorted sequence *A*[1 .. *j* - 1]	0	$n - 1$
i = *j* - 1	c_4	$n - 1$
while i > 0 and *A*[*i*] < *key*	c_5	$\sum_{j=2}^{n} t_j$
A[*i* + 1] = *A*[*i*]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i = *i* - 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[*i* + 1] = *key*	c_8	$n - 1$



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

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

Precision of running time

- Precise bounds are exhausting to find
- Precise bounds are to some extent meaningless

Caveat: if you're building code that flies an airplane or controls a nuclear reactor, you do care about precise, real-time guarantees

Simplifying running times

- Goal: identify broad classes of algorithms with similar performance

- Don't say: $1.62N^2 + 3.5N + 8$
- Do say: N^2

- Ignore the *low-order terms*
- Ignore the *constant factor* of high-order term

Why ignore low-order terms?

max # steps as function of N

size of input

	N	N^2	N^3	2^N
N=10	< 1 sec	< 1 sec	< 1 sec	< 1 sec
N=100	< 1 sec	< 1 sec	1 sec	10^{17} years
N=1,000	< 1 sec	1 sec	18 min	very long
N=10,000	< 1 sec	2 min	12 days	very long
N=100,000	< 1 sec	3 hours	32 years	very long
N=1,000,000	1 sec	12 days	10^4 years	very long

assuming 1 microsecond/step

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

THINK BIG

Why ignore constant factor?

- For classifying algorithms, constants are irrelevant in practice
  - $1.62N^2$ steps in pseudocode might be 1620 steps in assembly
  - My current laptop might be 2x as fast as last year's
  - ...but those aren't interesting properties of the algorithm
- Caveat: Performance tuning real-world code actually can be about getting the constants to be small!

Imprecise abstraction

- Exact: $1.62N^2 + 3.5N + 8$
- Imprecise abstraction: N^2

Other abstractions

- OCaml's **`int`** type abstracts (subset of) Z
- ±1 is an abstraction of {1,-1}

- Big Oh…

BIG ELL

Credit: Graham, Knuth, and Patashnik, *Concrete Mathematics*, chapter 9, 1989.

Big Ell

$L(n) = \{m \mid 0 \leq m \leq n\}$, where $m,n \in \mathbb{N}$

$L(n)$ represents a natural number less than or equal to n

e.g., $L(5) = \{0, 1, 2, 3, 4, 5\}$

Big Ell

Exercise: what is $1 + L(5)$?

Try to express answer in the form $L(x)$, for some x.

Hint: there are some ambiguities in this question.

CLICKER QUESTION 2

A little trickier…

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

…we can use this idea of Big Ell to invent an imprecise abstraction for running times

BIG OH

Big Oh, version 1

- L(n) represents any natural number that is less than or equal to a natural number n
- A natural function is a function of type $\mathbb{N} \rightarrow \mathbb{N}$
- O(g) represents any natural function that is less than or equal to a natural function g, for every input n
- Big Oh is a higher-order version of Big Ell: generalize from naturals to functions on naturals

Big Oh, version 1

Definition: $O(g) = \{ f \mid \forall n \,.\, f(n) \leq g(n)\}$

e.g.

- $O(\text{fun } n \rightarrow 2n) = \{f \mid \forall n \,.\, f(n) \leq 2n\}$
- $(\text{fun } n \rightarrow n) \in O(\text{fun } n \rightarrow 2n)$

Note: these are mathematical functions written in OCaml notation, not OCaml functions

Big Oh, version 2

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 natural function that is less than or equal to natural function g times some positive constant c, for every input n

Big Oh, version 2

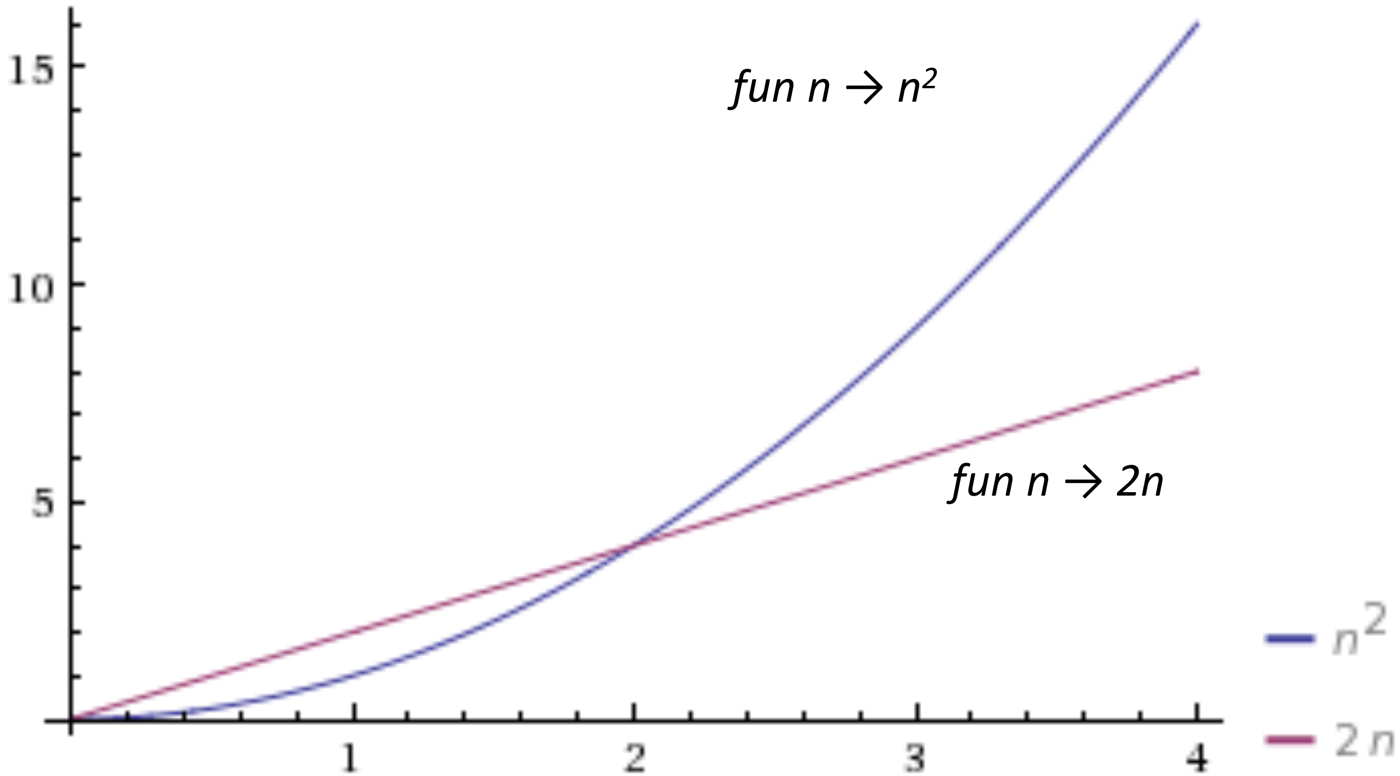
Definition: $O(g) = \{ f \mid \exists c>0 \,.\, \forall n \,.\, f(n) \leq c\, g(n)\}$

e.g.

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

Big Oh, version 3

Recall: THINK BIG



could just build a lookup table for inputs in the range 0..2

Big Oh, version 3

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:

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

this is the important, final definition you should know!

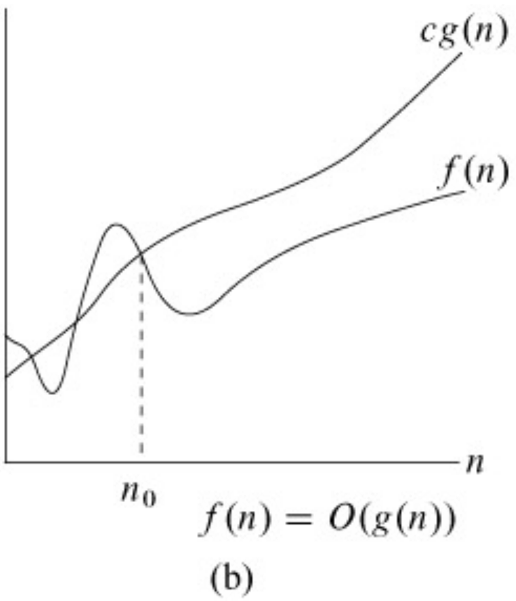
e.g.:

- $O(\text{fun } n \longrightarrow n^2) = \{ f \mid \exists c>0, n_0>0 . \forall n \geq n0 . f(n) \leq cn^2\}$
- $(\text{fun } n \longrightarrow 2n) \in O(\text{fun } n \longrightarrow n^2)$
  because $2n \leq cn^2$, where $c = 2$, for all $n \geq 1$

Asymptotic bound

Big Oh is an asymptotic upper bound

If f ∈ O(g) then f is at least as efficient as g, and might be more efficient



Graph: [Cormen et al. *Introduction to Algorithms*, 3rd ed, 2009]

Big Oh Notation: Warning 1

Instead of

$O(g) = \{f \mid ...$

most authors write

$O(g(n)) = \{f(n) \mid ...$

- 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

$$(\text{fun } n \longrightarrow 2n) \in O(\text{fun } n \longrightarrow n^2)$$

Nearly all authors write

$$2n = O(n^2)$$

- The standard defense is that = should be read here as "is" not as "equals"
- Be careful: one-directional "equality"!

EFFICIENCY

What is efficiency?

Final answer:

An algorithm is efficient if its worst-case running time on input size N is $O(N^d)$ for some constant d.

Upcoming events

- [today] A3 out
- [Monday] R4 due

This is efficient.

THIS IS 3110