Running Time

Speed

- When writing programs, we often want them to be fast.
- Several things affect this:
  1. the algorithm implemented
  2. the way the algorithm is implemented
  3. programming language used
  4. capabilities of the system running the program
- We won’t worry about 3 or 4.

Implementation

- For a given algorithm, there are many choices in how it's implemented.
  - eg. loop forwards or backwards, order of `if` conditions, how to split into separate methods, what variables to use, lazy/active boolean operators, …
  - Some of these will affect the speed of the program.
- No rules here: programming experience helps.
  - so does knowledge of system architecture, compilers, interpreters, language features, …

Algorithm Analysis

- The inherent running time of the algorithm will almost always overshadow other factors.
  - eg. there's nothing we can do to the Power1 algorithm implementation to make it as fast as Power2
    - … for large values of *y*.
  - eg. for sorted arrays, binary search will be faster
    - … for large arrays, in the worst case.

Measuring Running Time

- To evaluate the efficiency of an algorithm:
  - can’t just time it: different implementations, computers, architectures will affect time.
  - need something that will allow us to generalize
- We will count the number of “steps” required
  - … for an input of “size” n.
- Will be measured in terms of “big-O” notation

Big-O Notation

- Running time will be measured with “big-O” notation.
- Big-O is a way to indicate how fast a function grows.
- eg. “linear search has running time O(n) for an array of length n.”
  - indicates that linear search takes about n steps

Big-O Rules

- Ignore constants:
  - $O(c \cdot f(n)) = O(f(n))$
- Ignore smaller powers:
  - $O(n^a + n^{a-1}) = O(n^a)$
- Logarithms count less than a power
  - Think of $\log n$ as equivalent to $n^{0.00\ldots01}$
  - $O(n^{a+0.1}) > O(n^a \log n) > O(n^a)$
  - eg. $O(n\log n + n) = O(n\log n)$
  - eg. $O(n\log n + n^2) = O(n^2)$

Why Big-O?

- Looks at what happens for large inputs
  - Small problems are easy to do quickly
  - Big problems are more interesting.
  - Larger function makes a **huge** difference for large n.
- Ignores irrelevant details
  - Constants and lower-order terms will depend on the implementation
  - Don’t worry about that until we’ve got a good algorithm.

Function Comparison

n^2

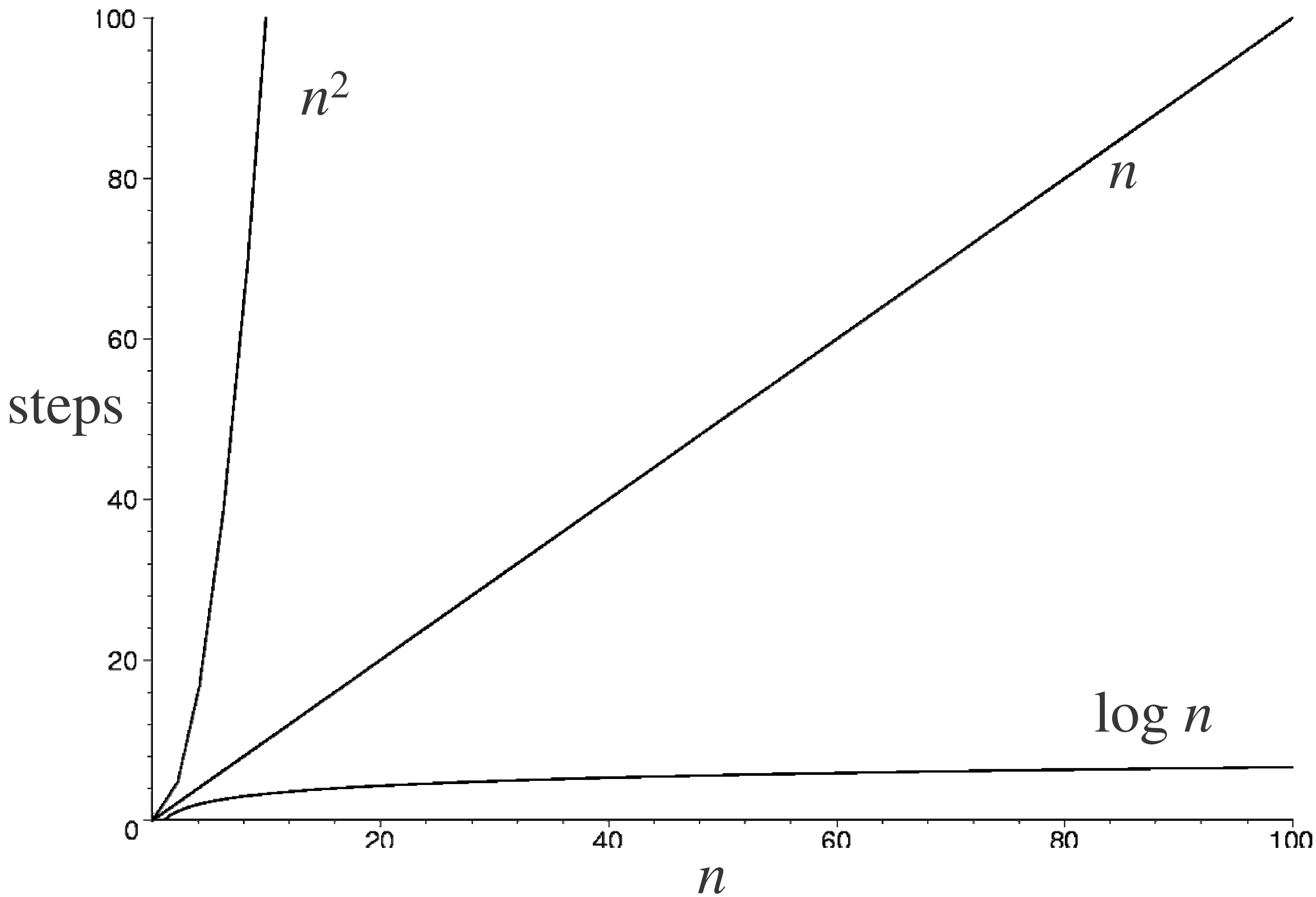
n

steps

$\log n$

n

(printable)



Determining Running Time

- Need to count the number of “steps” taken to complete
  - … in the **worst** case
  - … for input of “size” n.
  - a “step” must take constant ($O(1)$) time.
- Often:
  - iterations of the inner loop × work per loop
  - recursive calls × work per call

Examples 1

- Linear search:
  - checks each element in the array
  - O(n) (or "order n")
- Binary search:
  - Chops array in half with each step.
  - $n \rightarrow n/2 \rightarrow n/4 \rightarrow \ldots \rightarrow 2 \rightarrow 1$
  - takes $\log n$ steps: O($\log n$) (or "order $\log n$")

Examples 2

- Power 1: $x^y \rightarrow x \cdot x^{y-1}$
  - Makes y recursive calls: O(y)
- Power 2: $x^y \rightarrow x^{y/2} \cdot x^{y/2}$
  - Makes log y recursive calls: O(log y)
  - Had to be careful to not calculate $x^{y/2}$ twice
    - Would have created an O(y) algorithm
    - Instead, calculated and stored in a variable