Unit #1: Complexity Theory and Asymptotic Analysis

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Ed's class:
annotated slides

Unit Outline

- Brief Proof Review
- Algorithm Analysis: Counting the Number of Steps
- Asymptotic Notation
- Runtime Examples
- Problem Complexity

Learning Goals

- Given some code or an algorithm, write a formula that measures the number of steps executed by the code, as a function of the size of the input.
- Use asymptotic notation to simplify functions and to express relations between functions.
- Know and compare the asymptotic bounds of common functions.
- Understand why—and when—to use worst-case, best-case, or average-case complexity measures.
- Give examples of tractable, intractable, and undecidable problems.

Review: Proof by ...

- Counterexample
  - Show an example which does not fit with the theorem.
  - Thus, the theorem is *false*.
- Contradiction
  - Assume the opposite of the theorem.
  - Derive a contradiction.
  - Thus, the theorem is *true*.
- Induction
  - Prove the theorem for a base case (e.g., $n = 1$).
  - Assume that it is true for all $n \leq k$ (for arbitrary k).
  - Prove it for the next value ($n = k + 1$).
  - Thus, the theorem is *true*.

Example: Proof by Induction (Worked Example) 1/4

Theorem:
A positive integer x is divisible by 3 if and only if the sum of its decimal digits is divisible by 3.

e.g., 6 ⇒ divisible by 3
15 ⇒ 1+5=6
and
3|6
i.e., 6%3=0
6÷3=2r0

Proof:
Let $x_1 x_2 x_3 \ldots x_n$ be the n decimal digits of x.
Let the sum of its decimal digits be

e.g., 153 ↑ ↑ ↑ n=3
x_1 x_2 x_3 1+5+3=9

$$S(x) = \sum_{i=1}^{n} x_i$$

We'll prove the stronger result:

$$S(x) \bmod 3 = x \bmod 3.$$

112 ⇒ 1+1+2=4
3∤4
4%3≠0

same remainder

How do we use induction?
Induction on the # of digits: n.

Example: Proof by Induction (Worked Example) 2/4

$S(x) \bmod 3 = x \bmod 3$

Base Case:

Consider any number x with one ($n = 1$) digit (0-9).

$$LHS = S(x) = \sum_{i=1}^{n} x_i = x_1 = x. \quad = RHS \checkmark$$

So, it's trivially true that $S(x) \bmod 3 = x \bmod 3$ when $n = 1$.

When $n = 1$, $LHS = x \bmod 3 = RHS$ for $x \in \{0, 1, \ldots, 9\}$

Example: Proof by Induction (Worked Example) 3/4

Inductive Hypothesis: Assume true for $n = k$ $k \in n$ $k \in \mathbb{Z}^+$

Assume for an arbitrary integer $k > 0$ that for any number x with $n \leq k$ digits:

$$S(x) \bmod 3 = x \bmod 3.$$

Inductive Step: Show true for $n = k + 1$

Consider a number x with $n = k + 1$ digits:

$x = x_1 x_2 \cdots x_k$ e.g., $x = 369 = z$ now suppose $x = 3693$

$$x = x_1 x_2 \ldots x_k x_{k+1}.$$

Let z be the number $x_1 x_2 \ldots x_k$. It's a k-digit number; so, the inductive hypothesis applies:

$$S(z) \bmod 3 = z \bmod 3.$$

$3+6+9 \bmod 3 \rightarrow 3+6+9 \bmod 3$

$18 \bmod 3$

0

0

x_{k+1}

Example: Proof by Induction (Worked Example) 4/4

Inductive Step (continued):

$$
\begin{aligned}
x \bmod 3 &= (10z + x_{k+1}) \bmod 3 && (x = 10z + x_{k+1}) \\
&= (9z + z + x_{k+1}) \bmod 3 \\
&= (z + x_{k+1}) \bmod 3 && (9z \text{ is divisible by } 3) \\
&= (S(z) + x_{k+1}) \bmod 3 && (\text{inductive hypothesis}) \\
&= (x_1 + x_2 + \cdots + x_k + x_{k+1}) \bmod 3 \\
&= S(x) \bmod 3 \\
&= \text{LHS}
\end{aligned}
$$

QED (*quod erat demonstrandum*: "what was to be demonstrated")

adding a digit

369 → 3693

RHS =

$10(369) + 3 = 3693$

z x_{k+1}

$S(z)$

green on slide

because $S(x)$ is the sum of the $k+1$ digits prev.

where x has $k+1$ digits

$$S(x) = \sum_{n=1}^{k+1} x_i$$

See p. 3 of 4

A Task to Solve and Analyze

key	value

Find a student's name in a class given her student ID.

- Consider the data that you need to store.

  ID # and name

  Key-value pair : $\langle k_i, v_i \rangle$

- Consider the operation.

  Search for the key in a data structure and return its corresponding name

- Consider the possible data structures.

  array, map, dictionary, look-up table, "index"

- Does it matter which data structure we use?

  often, yes esp. for larger sizes

Efficiency

Suppose we have two or more algorithms that each solve the same problem.

- Some measure of *efficiency* is needed to determine which algorithm is "better".
- Complexity theory addresses the issue of how *efficient* an algorithm is.
- Suggest some qualities or metrics that we can measure, count, or compare in order to determine the efficiency of an algorithm.

- number of instructions
- wall clock time - CPU time
- memory used
- number of disk operations
- number of function calls
- " " network accesses

Analysis of Algorithms

- The analysis of an algorithm can give insight into two important considerations:
  - How long the program runs (time complexity or runtime)
  - How much memory it uses (space complexity)
- Analysis can provide insight into alternative algorithms.
- The *input size* is indicated by a non-negative integer n (but sometimes there are multiple measures of an input's size).
- Running time can be summarized—and represented—by a real-valued *function* of n such as:
  - $T(n) = 4n + 5$ linear
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$ exponential

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10
$\log n$	1ps
n	10ps
$n \log n$	$\sim$10ps
n^2	100ps
2^n	$\sim$1ns

nanosecond (ns) = one-billionth of a second

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$\log_{10} 100 = 2$

$n =$	10	100
$\log n$	1ps	2ps
n	10ps	100ps
$n \log n$	10ps	200ps
n^2	100ps	10ns
2^n	1ns	1Es

$n^3 = (100)^3 \Rightarrow 1\mu s$ (microsecond)

nanosecond (ns) = one-billionth of a second

Exasecond (Es) = 32 billion years

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10	100	1,000
$\log n$	1ps	2ps	3ps
n	10ps	100ps	1ns
$n \log n$	10ps	200ps	3ns
n^2	100ps	10ns	1μs
2^n	1ns	1Es	10^{289}s

nanosecond (ns) = one-billionth of a second
microsecond (μs) = one-millionth of a second
Exasecond (Es) = 32 billion years

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10	100	1,000	10,000
$\log n$	1ps	2ps	3ps	4ps
n	10ps	100ps	1ns	10ns
$n \log n$	10ps	200ps	3ns	40ns
n^2	100ps	10ns	1μs	100μs
2^n	1ns	1Es	10^{289}s	

nanosecond (ns) = one-billionth of a second
microsecond (μs) = one-millionth of a second
Exasecond (Es) = 32 billion years

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10	100	1,000	10,000	10^5
$\log n$	1ps	2ps	3ps	4ps	5ps
n	10ps	100ps	1ns	10ns	100ns
$n \log n$	10ps	200ps	3ns	40ns	500ns
n^2	100ps	10ns	1μs	100μs	10ms
2^n	1ns	1Es	10^{289}s		

Typical disk access

nanosecond (ns) = one-billionth of a second
microsecond (μs) = one-millionth of a second
Exasecond (Es) = 32 billion years

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10	100	1,000	10,000	10^5	10^6
$\log n$	1ps	2ps	3ps	4ps	5ps	6ps
n	10ps	100ps	1ns	10ns	100ns	1μs
$n \log n$	10ps	200ps	3ns	40ns	500ns	6μs
n^2	100ps	10ns	1μs	100μs	10ms	1s
2^n	1ns	1Es	10^{289}s			

nanosecond (ns) = one-billionth of a second
microsecond (μs) = one-millionth of a second
Exasecond (Es) = 32 billion years

Rates of Growth

Suppose a computer executes 1 operation (op) per picosecond (i.e., trillionth of a second: 10^{-12} s.). Here's how long it would take to run $T(n)$ operations, where $T(n)$ is a function of the input size n (e.g., $T(n) = \log n$):

$n =$	10	100	1,000	10,000	10^5	10^6	10^9
$\log n$	1ps	2ps	3ps	4ps	5ps	6ps	9ps
n	10ps	100ps	1ns	10ns	100ns	1μs	1ms
$n \log n$	10ps	200ps	3ns	40ns	500ns	6μs	9ms
n^2	100ps	10ns	1μs	100μs	10ms	1s	1week
2^n	1ns	1Es	10^{289}s				

nanosecond (ns) = one-billionth of a second
microsecond (μs) = one-millionth of a second
Exasecond (Es) = 32 billion years

Analyzing Code

$T(n)$ = # of lines of code executed

```
// Linear Search
find(key, array):
    for i = 0 to (length(array) - 1) do
        if array[i] == key
            return i
    return -1
```

looping

if search key is found, return its index (subscript)

1) What's the input size n?

# of elements (size) of the array

are you comparing integers? strings? 1000x1000 matrices

key size - does it matter

Analyzing Code

```
// Linear Search
find(key, array):
   for i = 0 to (length(array) - 1) do
      if array[i] == key
         return i
   return -1
```

rarely of interest

2) Should we assume a worst-case, best-case, or average-case scenario for running an input of size n?

depends

depends (often?)

nuclear reactor

Analyzing Code

k++

asm
L R5, k
A R5, 1
ST R5, k
} 3 instr

```
// Linear Search
find(key, array):
   for i = 0 to (length(array) - 1) do
      if array[i] == key
         return i
   return -1
```

1 } loop n times

3) How many lines are executed as a function of n in the worst-case?

$T(n) =$ $n(1+1)+1 = 2n+1$

↳ return

arguably one more for final check

Is *lines* the right unit?

maybe
- often proportional e.g., 3x, 5x scales proportionally

Analyzing Code

The number of lines executed in the worst-case is:

$$T(n) = 2n + 1$$

- Does the "1" matter? not usually
- Does the "2" matter? not usually

Big-O Notation

Asymptotic analysis – Big O is asymptotic notation for functions

Assume that for every integer n, $T(n) \geq 0$ and $f(n) \geq 0$.

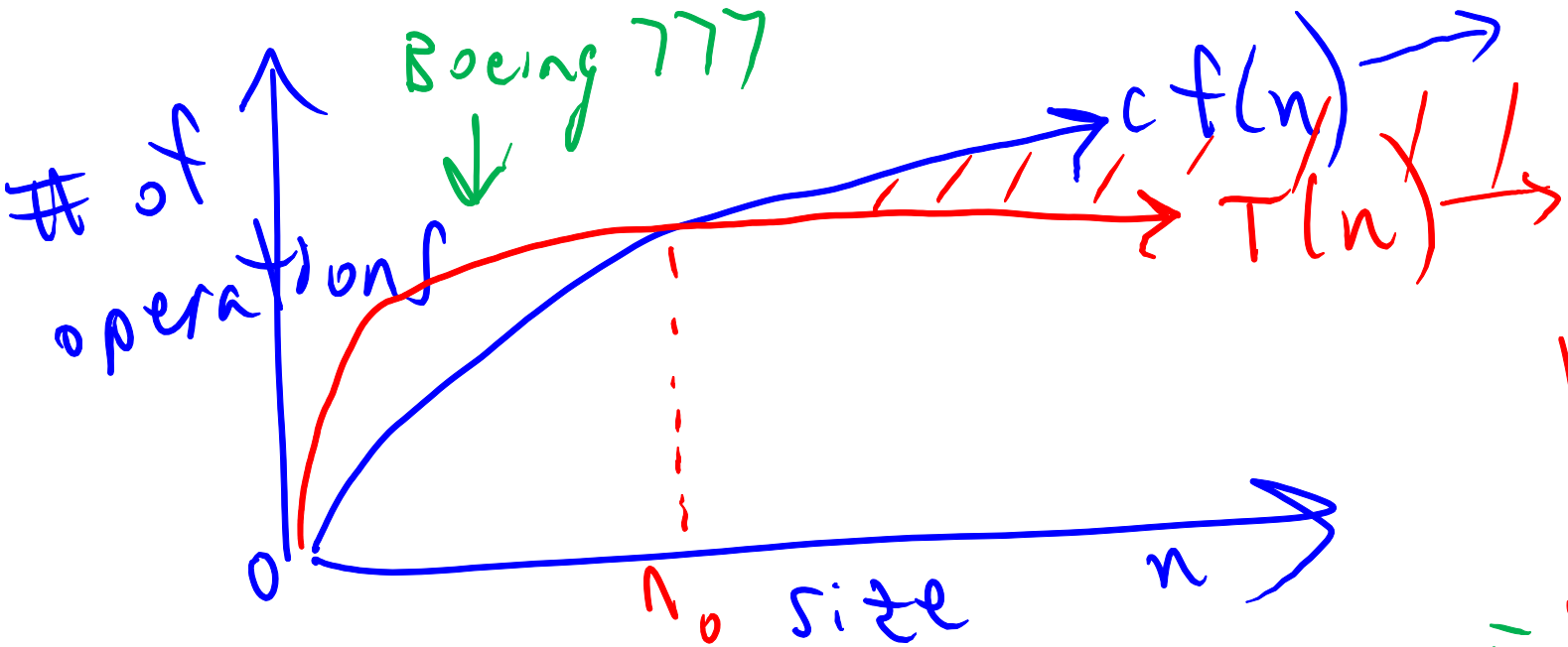
use this

$T(n) \in O(f(n))$ iff there are positive constants c and n_0 such that

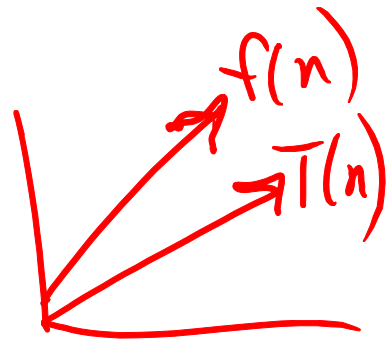
$T(n) = O(n)$

$T(n) \leq cf(n)$ for all $n \geq n_0$.

Meaning: "$T(n)$ grows no faster than $f(n)$"



"meaning something" ish? MATH PHYS



Example →

Asymptotic Notation

$T(n) = 2n+1$ on previous page

$\leq 2n+1n \quad \forall n \geq 1$

Example:

$T(n) = 25n^2 - 16n + 7$

$\leq 25n^2 + 0n^2 + 7n^2$

$= 32n^2$

$c = 32$

$n_0 = 1$

$\therefore T(n) \in O(n^2)$

- Big-O: $T(n) \in O(f(n))$ iff there are positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$.

upper bound

$= 3n \quad c = 3 \quad n =$

witnesses

$\therefore T(n) \in O(n$ where $c = 3$ & $n_0 = 1$

- Big-Omega: $T(n) \in \Omega(f(n))$ iff there are positive constants c and n_0 such that $T(n) \geq cf(n)$ for all $n \geq n_0$.

lower bound

$T(n) = 2n+1$

$\geq 2n \quad \forall n \geq 1 \quad n \in \mathbb{Z}^+$

$\therefore T(n) \in \Omega(n)$

$T(n) = 25n^2 - 16n + 7$

$\geq 20n^2$ when

$T(n) \in \Omega(n^2)$

$5n^2 \geq 16n - 7$ when $n \geq 3$

- Big-Theta: $T(n) \in \Theta(f(n))$ iff $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.

same upper & lower bound

upper & lower bound is the same

$T(n) = 2n+1 \Rightarrow T(n) \in \Theta(n)$

top expr $\geq$ second expression $\rightarrow$

Asymptotic Notation (cont.)

from previous page – witnesses
We want $25n^2 - 16n + 7 \geq 20n^2$ $\quad n \geq 3$
$5n^2 - 16n + 7 \geq 0$ $\quad n=3 \therefore n_0 = 3$
$5n^2 \geq 16n - 7 \Rightarrow 45 \geq 48 - 7$

- Little-o: $T(n) \in o(f(n))$ iff for **any** positive constant c, there exists n_0 such that $T(n) < cf(n)$ for all $n \geq n_0$.

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$$

- Little-omega: $T(n) \in \omega(f(n))$ iff for **any** positive constant c, there exists n_0 such that $T(n) > cf(n)$ for all $n \geq n_0$.

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty$$

Examples

$T(n) \leq 10^{13}$

$= 10^{13}(1)$

$T(n) \in O(1)$

$10,000n^2 + 25n \in \Theta(n^2)$

① $T(n) = 10000n^2 + 25n$

$\leq 10000n^2 + 25n^2$ $c_1 = 10025$

$= 10025n^2$

$\therefore T(n) \in O(n^2)$

② $T(n) \geq 10000n^2$

$\therefore T(n) \in \Omega(n^2)$ $c_2 = 10000$

③ $c_2 g(n) \leq T(n) \leq c_1 g(n)$

$\Rightarrow T(n) \in \Theta(n^2)$

$10^{-10}n^2 \in \Theta(n^2)$

pos. constant

let $c = \frac{1}{10^{10}}$ and $n_0 = 1$

$T(n) \leq cn^2$

$T(n) \geq cn^2$

$T(n) \in \Theta(n^2)$

$n \log n \in O(n^2)$

$T(n) = n \log n$

$\leq n^2 \; \forall n \geq 1$

$T(n) \in O(n^2)$

} true but a tight bound $O(n \log n)$ is better

Examples (cont.)

$n \log n \in \Omega(n)$

$T(n) = n \log n \geq n \quad \forall n \geq 2$

$T(n) \in \Omega(n)$ { tighter fit: $\Omega(n \log n)$

$n^3 + 4 \in o(n^4)$

$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \lim_{n \to \infty} \frac{n^3+4}{n^4} = \lim_{n \to \infty} \frac{1}{n} + \lim_{n \to \infty} \frac{4}{n^4} = 0$

$n^3 + 4 \in \omega(n^2)$

$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \lim_{n \to \infty} \frac{n^3+4}{n^2} = \lim_{n \to \infty} n + \lim_{n \to \infty} \frac{4}{n^2} = \infty$

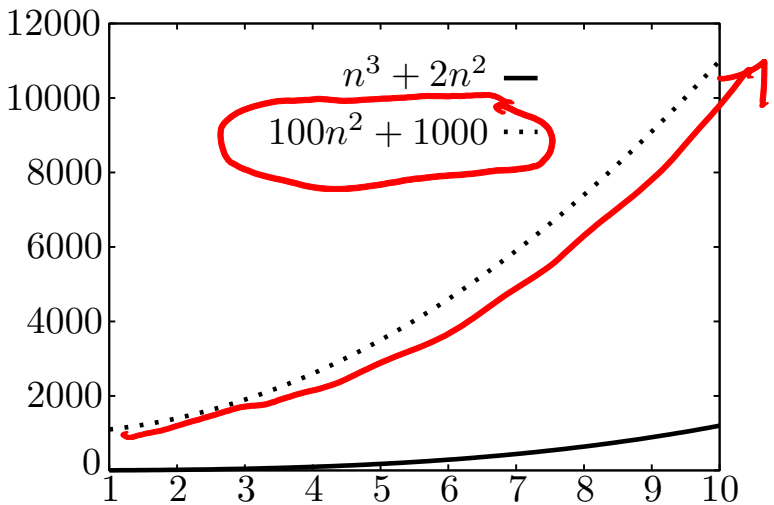
Analyzing Code

```
// Linear Search
find(key, array):
  for i = 0 to (length(array) - 1) do
      if array[i] == key
        return i
    return -1
```

4) How does $T(n) = 2n + 1$ behave asymptotically? What is the appropriate order notation? (O, o, Θ, Ω, ω?)

Asymptotically Smaller?

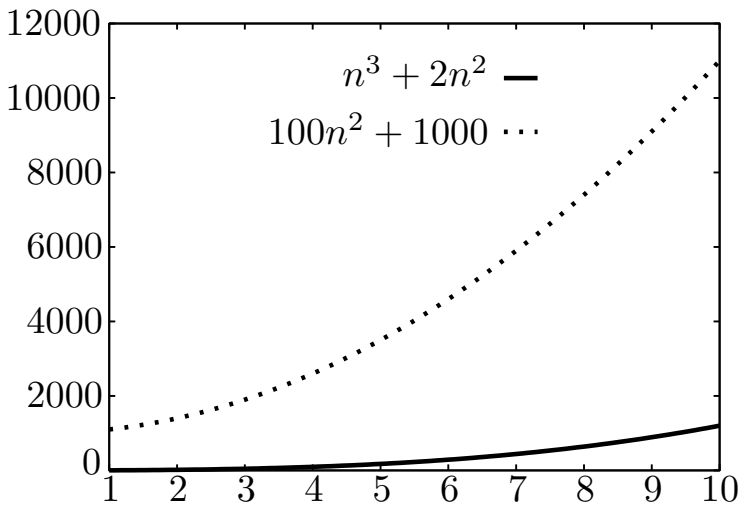
$n^3 + 2n^2$ versus $100n^2 + 1000$

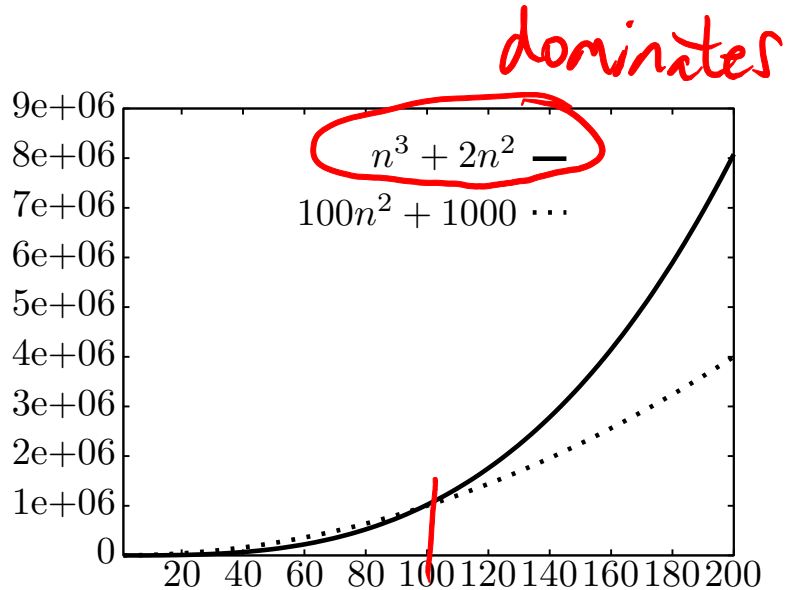


but ...

Asymptotically Smaller?

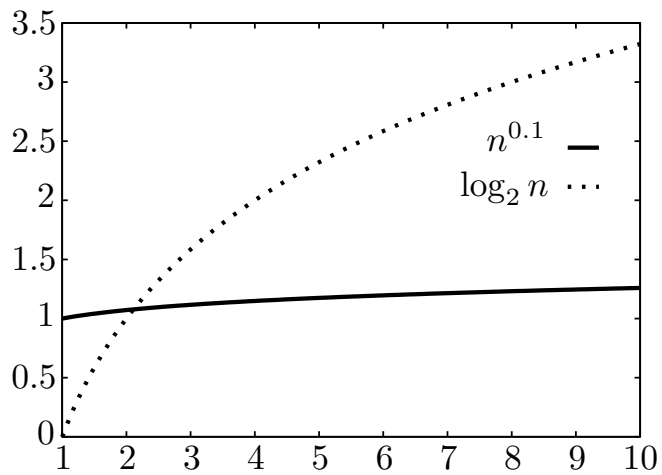
$n^3 + 2n^2$ versus $100n^2 + 1000$





Asymptotically Smaller? (cont.)

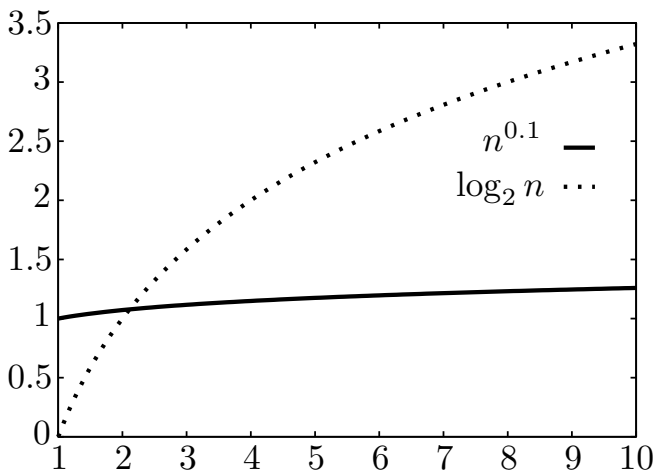
$n^{0.1}$ versus $\log_2 n$

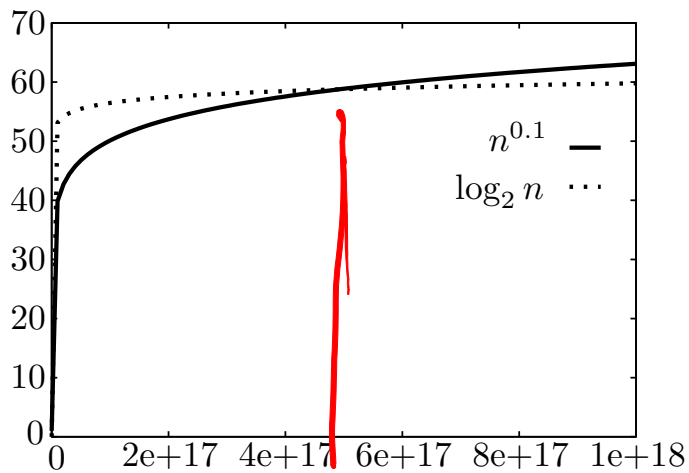


but . . .

Asymptotically Smaller? (cont.)

$n^{0.1}$ versus $\log_2 n$



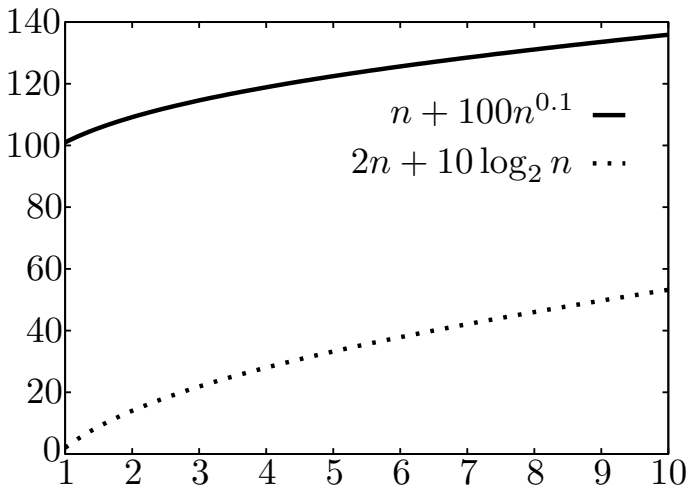


n_0

$\forall n \geq n_0 \ldots$

Asymptotically Smaller? (cont.)

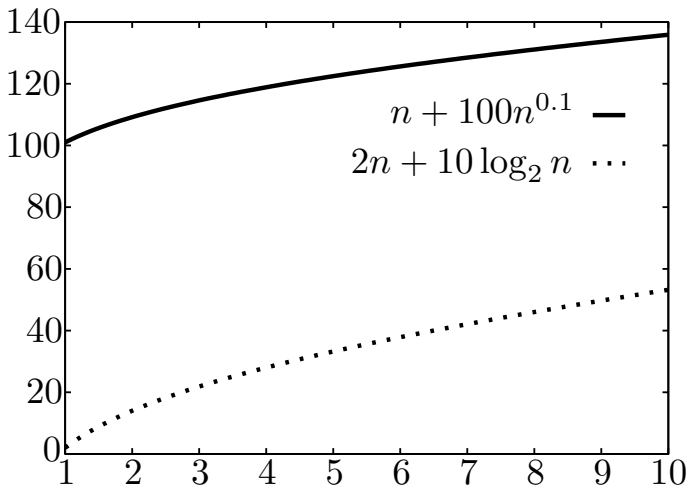
$n + 100n^{0.1}$ versus $2n + 10\log_2 n$

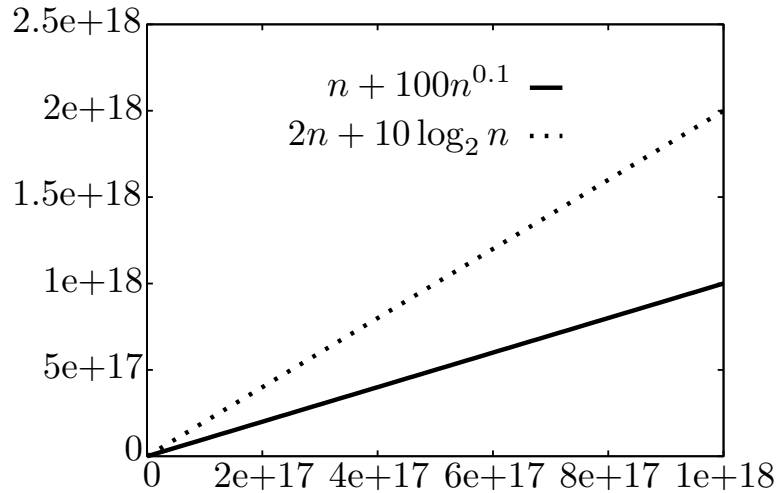


but ...

Asymptotically Smaller? (cont.)

$n + 100n^{0.1}$ versus $2n + 10 \log_2 n$





Typical Asymptotics

Tractable

(do-able on a computer for reasonable n)

- Constant: $\Theta(1)$ hash indexes O(1) expected time
- Logarithmic: $\Theta(\log n)$ ($\log_b n$, $\log n^2 \in \Theta(\log n)$) B+ trees
- Poly-Log: $\Theta(log^k n)$ ($\log^k n \equiv (\log n)^k$)
- Linear: $\Theta(n)$ min. value in an unsorted array
- Log-Linear: $\Theta(n \log n)$ many sorts
- Superlinear: $\Theta(n^{1+c})$ (c is a constant > 0)
- Quadratic: $\Theta(n^2)$
- Cubic: $\Theta(n^3)$
- Polynomial: $\Theta(n^k)$ (k is a constant)

Intractable

(not do-able for all but the smallest n values)

- Exponential: $\Theta(c^n)$ (c is a constant > 1) TSP

Sample Asymptotic Relations

- $\{1, \log n, n^{0.9}, n, 100n\} \subset O(n)$
- $\{n, n \log n, n^2, 2^n\} \subset \Omega(n)$
- $\{n, 100n, n + \log n\} \subset \Theta(n)$
- $\{1, \log n, n^{0.9}\} \subset o(n)$
- $\{n \log n, n^2, 2^n\} \subset \omega(n)$

Analyzing Code

aside:
$f(n) + g(n) \in O(\max\{f(n), g(n)\})$
but how about: $1 + 2 + \ldots + n \in O(\max\{O(1), O(1), \ldots O(n)\}$
- need a constant
$\frac{n(n+1)}{2}$ # of terms
?
No

- Single operations: constant time
- Consecutive operations: sum of the operations' times
- Conditionals: condition time plus the maximum (for worst-case analysis) of the branch times
- Loops: sum of the loop body times } n times through the loop?
- Function call: time for the function

Above all, use common sense!

$50n$ { for $i = 1$ to n [< 50 operations]

$O(\lg n) \Rightarrow$ loop is $O(n \lg n)$ operations

Runtime Example #1

```
for i = 1 to n do
    for j = 1 to n do
        sum = sum + 1
```

$\sum_{i=1}^{n} n = n + n + \ldots + n$ (n times) $= n^2 \Rightarrow O(n^2)$

① $\sum_{j=1}^{n} 1 = 1 + 1 + \ldots + 1 = n$ (n times)

②

outer loop # times	inner loop
$i = 1$	$j = 1$ to $n \Rightarrow n$ times
2	n
$\vdots$	
n	n
$\sum = n(n) = n^2$	

Big O:
$T(n) = n^2$
$\therefore T(n) \in O(n^2)$

Big Ω:
$T(n) = n^2$
$\geq n^2$ trivial
$\therefore T(n) \in \Omega(n^2)$

Big Θ:
$\therefore T(n) \in \Theta(n^2)$

Runtime Example #2

```
i = 1
while i < n do
    for j = i to n do
        sum = sum + 1
    i++
```

① n times; ? times; n times? No

$\sum_{j=i}^{n} 1 = n - i + 1$

$\sum_{i=1}^{n-1} (n - i + 1)$

OR use ②

outer	inner (sum is executed)
1	$j = 1$ to $n \Rightarrow n$ times
2	2 to $n \Rightarrow n-1$
3	3 to $n \Rightarrow n-2$
⋮	⋮
$n-1$	$j = n-1$ to $n \Rightarrow 2$

$\text{sum} = \sum = n + (n-1) + (n-2) + \ldots + 2$

sum of pos. ints $\Rightarrow = \frac{n(n+1)}{2} - 1$

②b

Big O: $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n - 1 \leq \frac{1}{2}n^2 + \frac{1}{2}n^2 = n^2$

$\therefore T(n) \in O(n^2)$

Big Ω: $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n - 1 \geq \frac{1}{4}n^2 \quad n \geq 2$

$\therefore T(n) \in \Omega(n^2)$

$\Rightarrow T(n) \in \Theta(n^2)$

Runtime Example #3 (harder)

```
i = 1
while i < n do
    for j = 1 to i do
        sum = sum + 1
    i += i
```

i timer

not 1

sum of a geometric series

① outer	inner
1	j=1 to 1 ⇒ 1
2	1 to 2 ⇒ 2
4	⇒ 4
8	⇒ 8
⋮	⋮
break from loop → 2^k	1 to 2^k ⇒ 2^k
k = ?	$k \approx \lg n$

actually, $k = \lceil \lg n \rceil - 1$

e.g., $n = 32 \Rightarrow k = 4$

$n = 33 \Rightarrow k = 5$

Sum geometric series

$$T(n) = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

$$= 2^{\lceil \lg n \rceil} - 1$$

② Big-O

$$\leq 2^{\lceil \lg n \rceil} - 1$$

$$< 2^{\lg n + 1}$$

$$= 2^{\lg n} \cdot 2^1$$

$$= 2^{\lg n} \cdot 2$$

$$= 2n$$

$\therefore T(n) \leq 2n \rightarrow \therefore T(n) \in O(n)$

③ Big Ω

since $n = 2^{\lg n}$

$n - 1 = 2^{\lg n} - 1$

$\leq 2^{\lceil \lg n \rceil} - 1$

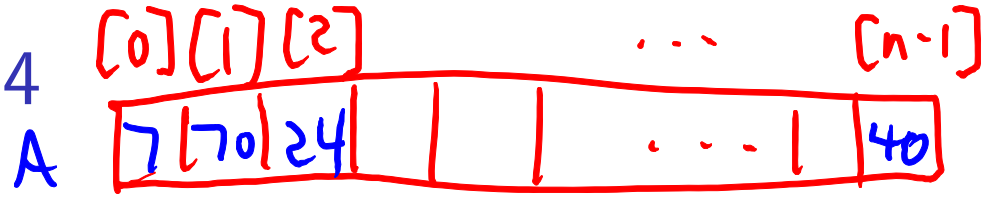
$\therefore T(n) \geq n - 1$

$\geq \frac{1}{2} n \quad \forall n \geq 2$

$\therefore T(n) \in \Omega(n)$

$\therefore T(n) \in \Theta(n)$

Runtime Example #4




```
int max(A, n):
    if( n == 1 ) return A[0]
    return larger of A[n-1] and max(A, n-1)
```

Base Case

40

1 shorter

Recursion almost always yields a recurrence relation:

B.C. $T(1) \leq b$ constant

recursive step $T(n) \leq c + T(n-1)$ if $n > 1$

Solving the recurrence:

$$
\begin{aligned}
T(n) &\leq c + \underline{c + T(n-2)} && \text{(substitution)} \\
&\leq c + c + \underline{c + T(n-3)} && \text{(substitution)} \qquad k < n \\
&\leq kc + T(n-k) && \text{(extrapolating } k > 0\text{)} \\
&= (n-1)c + T(1) && \text{(for } k = n-1\text{)} \\
&\leq (n-1)c + b && \text{B.C.}
\end{aligned}
$$

shown by induction

$T(n) \in O(n) = cn - c + b$

Runtime Example #5: Mergesort

Mergesort algorithm:

Split list in half, sort first half, sort second half, merge together

Recurrence relation:

$$T(1) \leq b \quad \text{(B.C.)}$$

$$T(n) \leq 2T(n/2) + cn \quad \text{if } n > 1$$

split into 2 smaller problems

merge (e.g., Prog. Proj. #1)

Solving recurrence:

$$\begin{aligned}
T(n) &\leq 2T(n/2) + cn \\
&\leq 2(2T(n/4) + cn/2) + cn && \text{(substitution)} \\
&= 4T(n/4) + 2cn \\
&\leq 4(2T(n/8) + cn/4) + 2cn && \text{(substitution)} \\
&= 8T(n/8) + 3cn \\
&\leq 2^k T(n/2^k) + kcn && \text{(extrapolating } k > 0\text{)} \\
&= nT(1) + cn \lg n && \text{(for } 2^k = n\text{)}
\end{aligned}$$

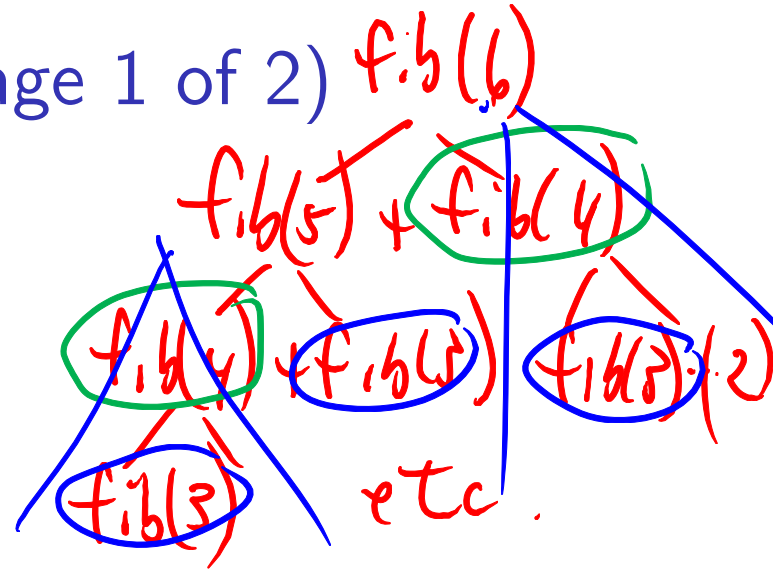
$T(\frac{n}{2}) \leq 2T(\frac{n}{4}) + \frac{cn}{2}$ by defn.

general case

$T(n) \in$ $\leq bn + cn\lg n$ (B.C.) (Big O - throw away smaller terms)

$O(n \lg n)$

Runtime Example #6: Fibonacci (page 1 of 2)



Recursive Fibonacci:

```
int fib(n)
  if( n == 0 or n == 1 ) return n
  return fib(n-1) + fib(n-2)
```

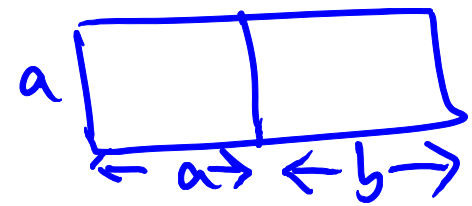
B.C. / recur case

Recurrence Relation: (lower bound) Big-Ω

to show "at least as slow as..."

$$\begin{cases} T(0) \geq b \\ T(1) \geq b \\ T(n) \geq T(n-1) + T(n-2) + c & \text{if } n > 1 \end{cases}$$

Golden Ratio



$\frac{a}{b} = \frac{a+b}{a} = \phi$

Claim:

$$T(n) \geq b\varphi^{n-1}$$ } growth is exponential

Golden Ratio

→ where $\varphi = (1+\sqrt{5})/2 \approx 1.618$

Note: $\varphi^2 = \varphi + 1$

$\phi \cdot \phi = \left(\frac{1+\sqrt{5}}{2}\right)\left(\frac{1+\sqrt{5}}{2}\right) = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} \Leftarrow \frac{1+\sqrt{5}}{2} + \frac{2}{2}$

Runtime Example #6: Fibonacci (page 2 of 2)

Claim:

$$T(n) \geq b\varphi^{n-1}$$

Proof: (by induction on n)

Base Cases: $T(0) \geq b > b\varphi^{-1}$ and $T(1) \geq b = b\varphi^{0}$.

Inductive Hypothesis: Assume $T(n) \geq b\varphi^{n-1}$ for all $n \leq k$.

Inductive Step: Show that it's true for $n = k + 1$.

$$\begin{aligned}
T(n) &\geq T(n-1) + T(n-2) + c \\
&\geq b\varphi^{n-2} + b\varphi^{n-3} + c && \text{(by inductive hypothesis)} \\
&= b\varphi^{n-3}(\varphi + 1) + c \\
&= b\varphi^{n-3}\varphi^2 + c \\
&\geq b\varphi^{n-1}
\end{aligned}$$

$T(n) \in \Omega(\phi^{n-1}) \Rightarrow \Omega(\phi^n)$

Why? The same recursive call is made numerous times.

Example #7: Learning from Analysis

To avoid recursive calls:

- Store base case values in a table.
- Before calculating the value for n:
  - Check if the value for n is in the table.
  - If so, return it.
  - If not, calculate it and store it in the table.

as we generate each answer

This strategy is called *memoization* and is closely related to *dynamic programming*.

Look up the stored value to a previously computed result.

How much time does this version take?

$O(n)$ because we compute each value once

$O(1)$

Runtime Example #8: Longest Common Subsequence

Problem: Given two strings (A and B), find the longest sequence of characters that appears, in order, in both strings.

Example: $|A| = n = 9$ $|B| = m = 13$

A = `search me` B = `insane method`

A longest common subsequence is "`same`"; another is "`seme`".

Applications of LCS:

DNA sequencing, revision control systems, `diff`, ..., diff. between 2 files, MS word compare, MOSS, TurnItIn(?)

Runtime Example #8: LCS (cont.)

An Algorithm and Its Analysis:

Brute force (exhaustive) search

for each possible subsequence of A

[search for it in B

if (found & ≥ longest subsequence so far)

[record it

Let $|A| = n$

# of possible subsequences is 2^n

because A = search me n=9

1 0 1 0 0 0 0 1 1

2(2)(2)...(2) ← ↑ choose it or don't (0)

$\Rightarrow 2^9$ possibilities

For each one of these, test it against B. } try all possibilities starting with pos: 0, 1, ..., m

$T(n,m) \in$ brute force $\Omega(2^n \cdot m \cdot n)$ at least $2^n \cdot m \cdot n$

aside: best = dynamic programming $\Theta(nm)$ $= T(n,m)$

Example #9

$\lg(xy) = \lg(x) + \lg(y)$

Θ

Find a tight bound on $T(n) = \lg(n!)$.

① Big O $T(n) = \lg(n(n-1)(n-2)\cdots(2)(1))$

$= \lg n + \lg(n-1) + \lg(n-2) + \cdots + \lg 2 + \lg 1 =$

$\leq \lg n + \lg n + \lg n + \cdots + \lg n \quad (+\lg n) \quad \sum_{k=1}^{n} \lg k$

$= n \lg n$

$\Rightarrow T(n) \in O(n \lg n)$

② Big Ω $T(n) = \lg n + \lg(n-1) + \lg(n-2) + \cdots + \lg 2 + \lg 1$

$\geq \lg \frac{n}{2} + \lg \frac{n}{2} + \lg \frac{n}{2} + \cdots \lg \frac{n}{2} + 0 + 0 + 0$ throw away the bottom half

$\frac{n}{2} \lg \frac{n}{2} = \frac{n}{2}(\lg n - \lg 2)$

$= \frac{n}{2}(\lg n - 1)$

$\geq \frac{n}{4} \lg n \quad \forall n \geq 4$

$c = \frac{1}{4} \quad n_0 = 4$

$\therefore T(n) \in \Omega(n \lg n)$

③ ① & ② $\Rightarrow T(n) \in \Theta(n \lg n)$

Review: Logarithms

$\log_b x$ is the exponent that b must be raised to, in order for it to equal x.

- $\lg x \equiv \log_2 x$ (base 2 is common in CS)
- $\log x \equiv \log_{10} x$ (base 10 is common for humans)
- $\ln x \equiv \log_e x$ (the natural log)

Note: $\Theta(\lg n) = \Theta(\log n) = \Theta(\ln n)$ because

$$\log_b n = \frac{\log_c n}{\log_c b}$$

for constants $b, c > 1$.

$\lg n$

$\log_2 n = \frac{\log_{10} n}{\log_{10} 5}$

$\lg(xy) = \lg x + \lg y$

$\lg\left(\frac{x}{y}\right) = \lg x - \lg y$

Asymptotic Analysis Summary

as "n" grows without bound

- Determine the input size.
- Express the resources (time, memory, etc.) that an algorithm requires as a function of its input size.
  - Worst case ✓✓
  - Best case – rarely
  - Average case ✓
- Use asymptotic notation (O, Ω, Θ) to express the function simply.

Problem Complexity

The **complexity of a problem** is the complexity of the best algorithm to solve that problem.

- We can sometimes prove a lower bound on a problem's complexity. To do so, we must show a lower bound on any *possible* algorithm to solve it.
- A correct algorithm establishes an upper bound on the problem's complexity.

Searching an unsorted list using comparisons takes $\Omega(n)$ time (lower bound).
- Linear search takes $O(n)$ time (matching upper bound).

$\Theta(n)$

Sorting a list using comparisons takes $\Omega(n \log n)$ time (lower bound).
- Mergesort takes $O(n \log n)$ time (matching upper bound).

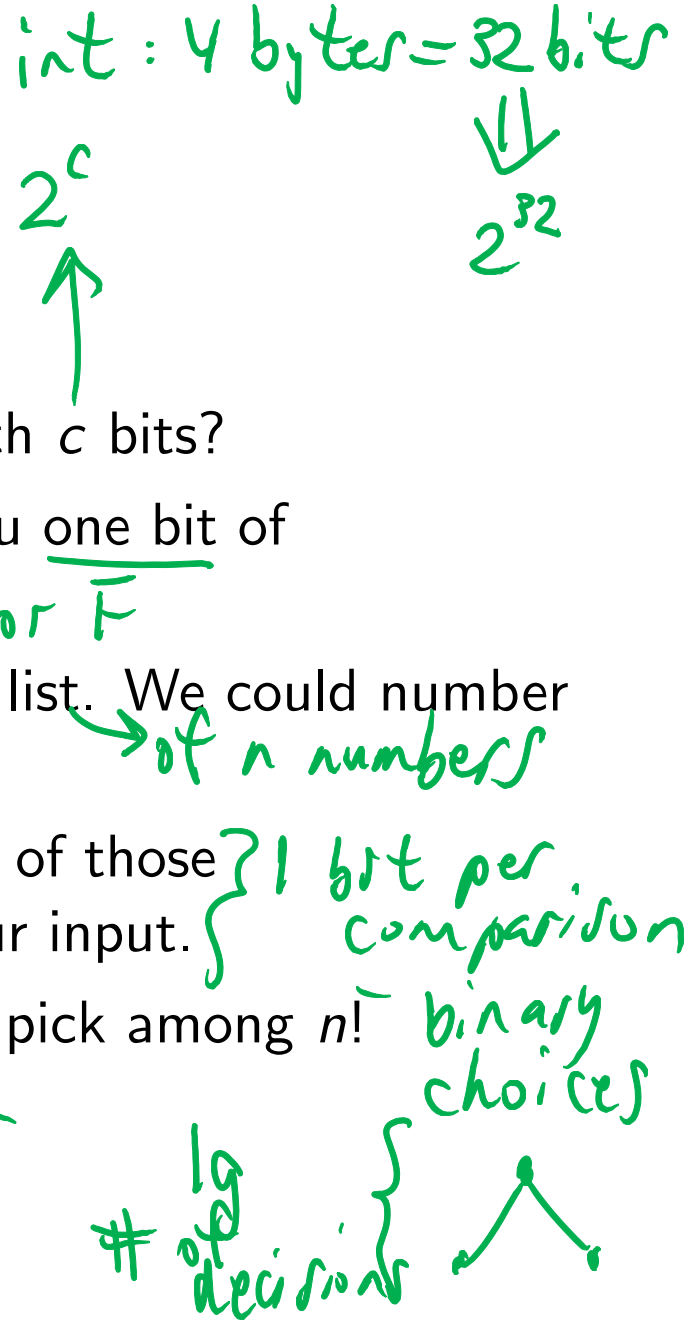
$\Theta(n \lg n)$

Aside: Who Cares About $\Omega(\lg(n!))$?

Can You Beat $O(n \log n)$ Sort?

Chew these over:

- How many different values can you represent with c bits?
- Comparing two values ($x < y$) gives you one bit of information.
- There are $n!$ possible ways to reorder a list. We could number them: $1, 2, \ldots, n!$
- Sorting basically means choosing which of those reorderings/numbers you'll apply to your input.
- How many comparisons does it take to pick among $n!$ numbers?



Problem Complexity

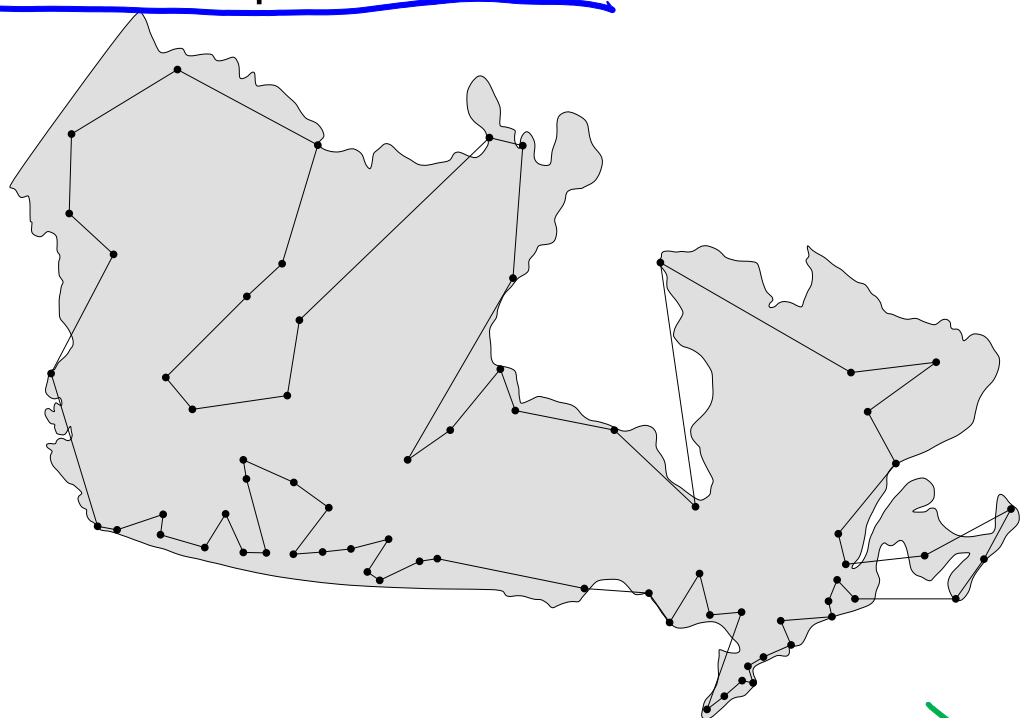
P

Sorting: Solvable in polynomial time, tractable

Traveling Salesman Problem (TSP): In 1,290,319 km, can I drive to all the cities in Canada and return home? `www.math.uwaterloo.ca/tsp/`

NP

Checking a solution takes polynomial time. Current fastest way to find a solution takes exponential time in the worst case.



Are problems in NP really in P? $1,000,000 prize

Problem Complexity

Searching and Sorting: P, tractable
Traveling Salesman Problem: NP, intractable?
Kolmogorov Complexity: Uncomputable (undecidable)

FYI: The Kolmogorov Complexity of a string is the length of the shortest description of it. It can't be computed (e.g., Berry Paradox).

FYI: Also uncomputable: the Halting Problem.

See Google or Wikipedia for more information, if you're interested.