Unit #1: Complexity Theory and Asymptotic Analysis

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

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 → for-loops, recursion
  1. Prove the theorem for a base case (e.g., $n = 1$).
  2. Assume that it is true for all $n \leq k$ (for arbitrary k).
  3. Prove it for the next value ($n = k + 1$).
  - Thus, the theorem is *true*.

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

$x = 153$ $\quad 1+5+3 = 9$

x_1 x_2 x_3 $\quad n = 3$

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

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

Sum of all digits

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

We'll prove the stronger result: — not only whether it's divisible by 3, but also remainders are the same.

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

How do we use induction?
↳ induction on the number of digits: n

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

(1) Base Case: → always going to start with base case

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

LHS = RHS

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

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

(LHS: $S(x) \bmod 3$; RHS: $x \bmod 3$)

When $n=1$: $LHS = S(x) \bmod 3 = x \bmod 3 = RHS$

$x = \{0, 1, 2 \ldots, 8, 9\}$

*Remember: n is the number of digits

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

Inductive Hypothesis: (2) Assume this is true

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

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

$x = 3693$

$z = 369$

x_{k+1}

x_1 x_2 x_3

Inductive Step: (3) Show it's true for $n = k+1$

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

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

the first k digits of the number

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

369 %3 = 0

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

sum of all k digits

3+6+9 = 18 %3 = 0

Inductive hypothesis.
Now, do (3) inductive step (k+1)

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

Inductive Step (continued):

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

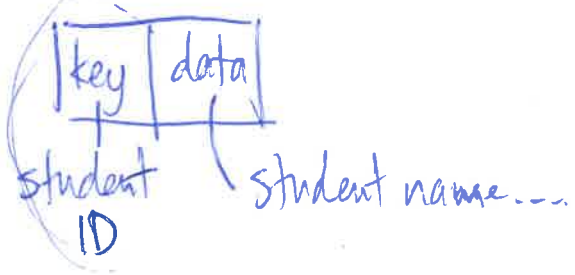
QED (*quod erat demonstrandum*: "what was to be demonstrated")

A Task to Solve and Analyze

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

- Consider the data that you need to store.

  Key-value pair

- Consider the operation.

  Search for key in the data structure, and return corresponding data (name)

- Consider the possible data structures.

  array, linked-list, and many more..

- Does it matter which data structure we use?

  Yes! Many can get the job "done", but very different in terms of efficiency

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. – average speed
  - ↳ total number of instructions
  - ↳ " " of lines of code
  - ↳ " " of disk operations, network accesses

  } aren't always the same

– run time in seconds, total CPU cycles needed

Memory efficiency

Analysis of Algorithms

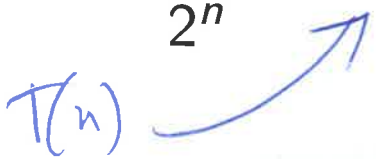
- The analysis of an algorithm can give insight into two important considerations:
  - How long the program runs (time complexity or runtime)
  - How much memory it uses (space complexity)
- Analysis can provide insight into alternative algorithms.
- The *input size* is indicated by a non-negative integer 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:
  - $T(n) = 4n + 5$
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$

If we have n students it will take T(n) amount of time to do...

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
$n \log n$	10ps
n^2	100ps
2^n	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., $T(n) = \log n$):

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

$n^2 \log$

n^3

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

10^{-6}

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		

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

is this the best measure?

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

if search key is found return its index

1) What's the input size n?

⇓

# of elements (students) in the array

Depends on what we're comparing?

- integers,
- Strings
- 10000 x 10000 matrix

Analyzing Code

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

generally → safety-critical system

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

↓ rarely used, unrealistic

Analyzing Code

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

1 (for i = 0 to ...)
1 (if array[i] == key)
1 (return i)
1 (return -1)

} how many times do we loop?

n times

once at the end

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

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

Is *lines* the right unit?

↳ It's often proportional to the right answer

- it will scale about the same for different sizes of n.

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

} we usually throw constants away.

* if the constant is really high, we may try and improve it.

Big-O Notation Asymptotic analysis -

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

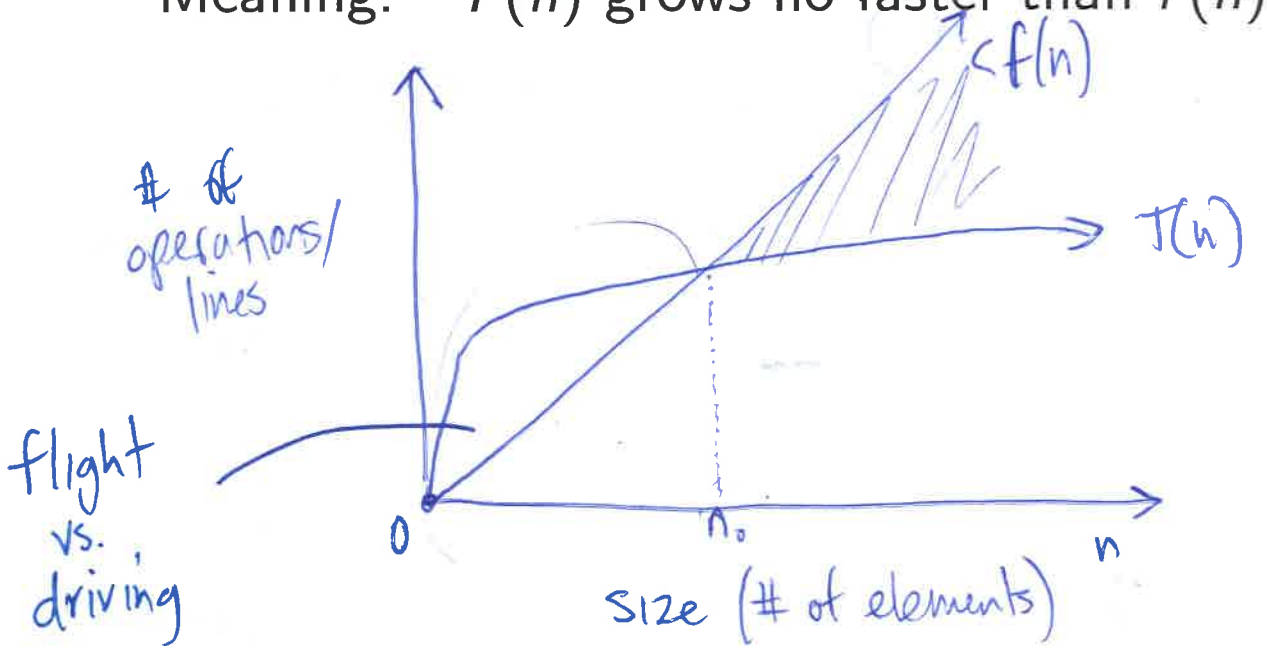
$T(n) \in O(f(n))$ iff there are positive constants c and n_0 such that

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

$T(n)$: grow no faster

$n \geq n_0$: for all n greater than n_0

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



Asymptotic Notation

$T(n) = 2n+1 \leq 2n+1n$ as long as $n \geq 1$.
$\leq 3n$
$\therefore T(n) \in O(n)$ where $c = 3$ and $n_0 = 1$ } witnesses

upper-bound

- 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 $n \geq n_0$.

lower-bound

- 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 $n \geq n_0$.

$T(n) = 2n+1 \geq 2n$ as long as $n \geq 1$

$\therefore T(n) \in \Omega(n)$ with $c = 2$ and $n_0 = 1$

same upper- and lower-bound.

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

$T(n) = 2n+1 \Rightarrow T(n) \in \Theta(n)$ when $n_0 = 1$

Asymptotic Notation (cont.)

- 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 $n \geq n_0$.

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

big-O: $T(n) \leq f(n)$
big-Omega: $T(n) \geq f(n)$
big-Theta: $T(n) = f(n)$
~~bigo~~
little-o: $T(n) < f(n)$
little-omega: $T(n) > f(n)$

Examples

① a) $T(n) = 10000n^2 + 25n \leq 10000n^2 + 25n^2$ when $n \geq 1$

$\leq 10025n^2$

$\therefore T(n) \in O(n^2)$ with $c = 10025$, and $n_0 = 1$

b) $10000n^2 + 25n \leq 10001n^2$

$25n \leq n^2$

$25 \leq n$

$\therefore T(n) \in O(n^2)$ with $c = 10001$ and $n_0 = 25$

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

① $O(n)$ ② $\Omega(n)$

② $T(n) = 10000n^2 + 25n \geq 10000n^2$ when $n \geq 0$

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

③ $\Rightarrow T(n) \in \Theta(n^2)$

Constant

$10^{-10}n^2 \in \Theta(n^2)$ Let $c = \frac{1}{10^{10}}$ and $n_0 = 0$

$T(n) \leq cn^2$ By O

$T(n) \geq cn^2$ By Omega

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

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

$T(n) = n \log n \leq n^2$ when $n \geq 1$

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

Is $T(n)$ $\Omega(n^2)$?

$T(n) = n \log n \geq n^2$

Examples (cont.)

* when using base-2 log we will write it lg

$T(n) = n\log n \geq n$ when $n \geq 2$
$\log n \geq 1$
$\therefore T(n) \in \Omega(n)$

$T(n)$ is $O(n)$
$T(n) = n\log n \leq n$
$\log n \leq 1$ Can't do

$n \log n \in \Omega(n)$ could have done $n \log n$

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

$n^3 + 4 \in \omega(n^2)$ $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{n^3+4}{n^2} = \underbrace{\lim_{n\to\infty} n}_{\infty} + \underbrace{\lim_{n\to\infty} \frac{4}{n^2}}_{0} = \infty$

Analyzing Code

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

n

1

worst-case!

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

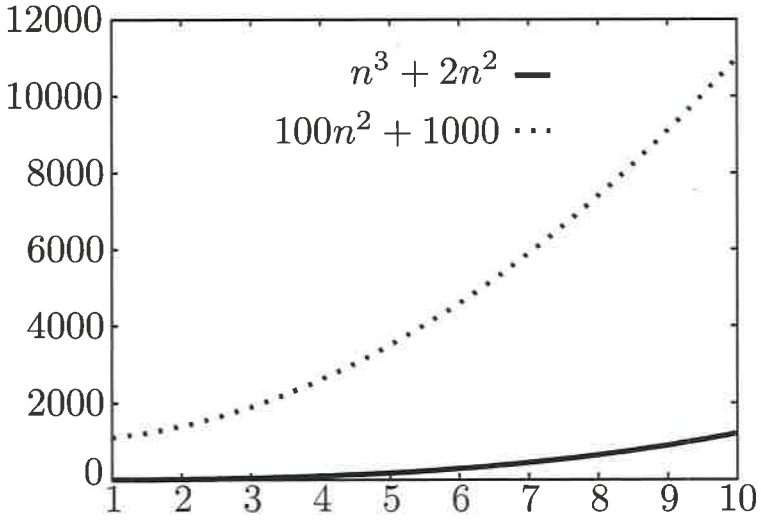
What about if we don't consider worst-case?

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

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

Asymptotically Smaller?

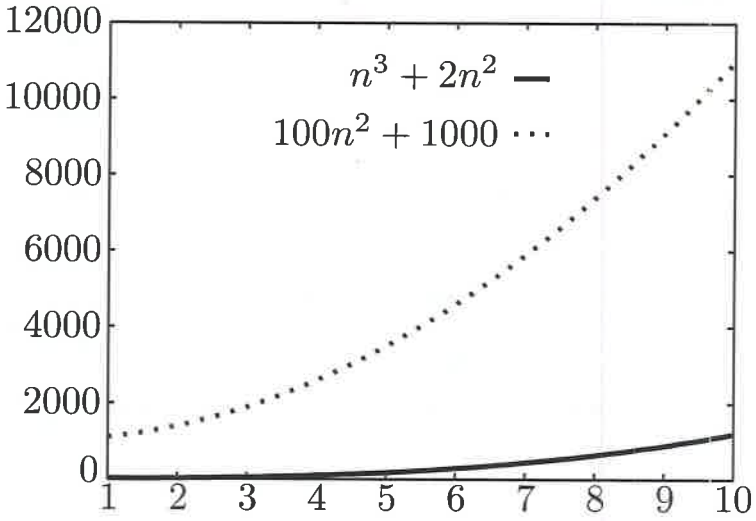
$n^3 + 2n^2$ versus $100n^2 + 1000$

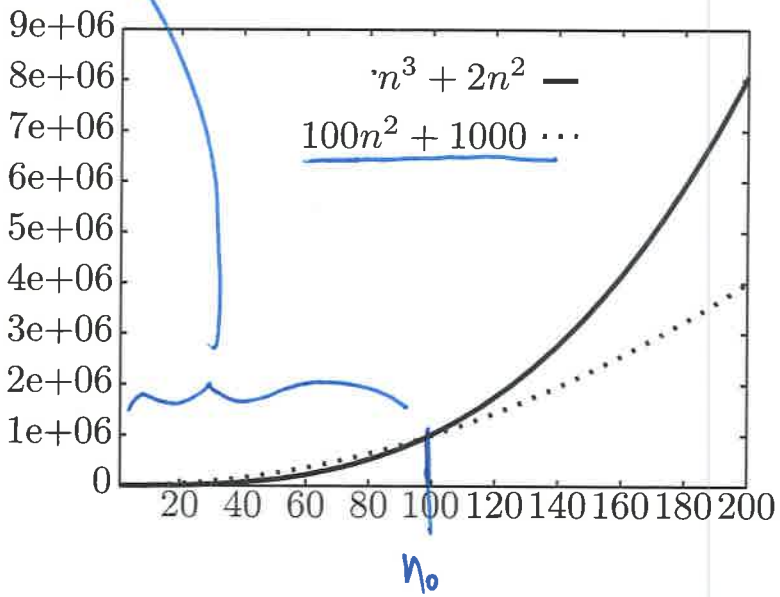


Asymptotically Smaller?

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

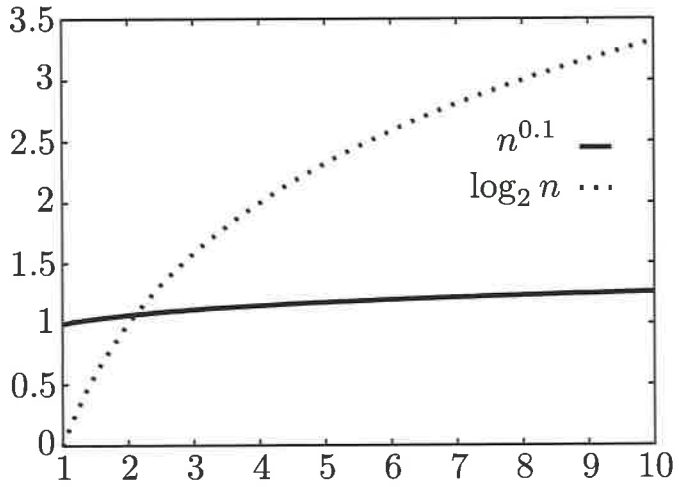
If we are certain $n < n_0$ then $n^3 + 2n^2$ is a good choice





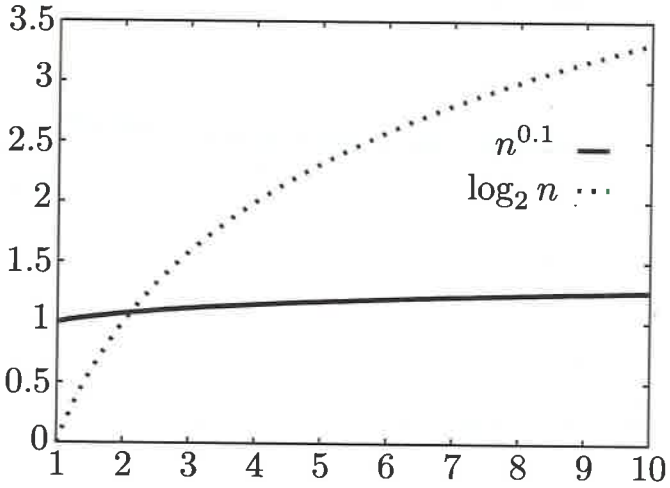
Asymptotically Smaller? (cont.)

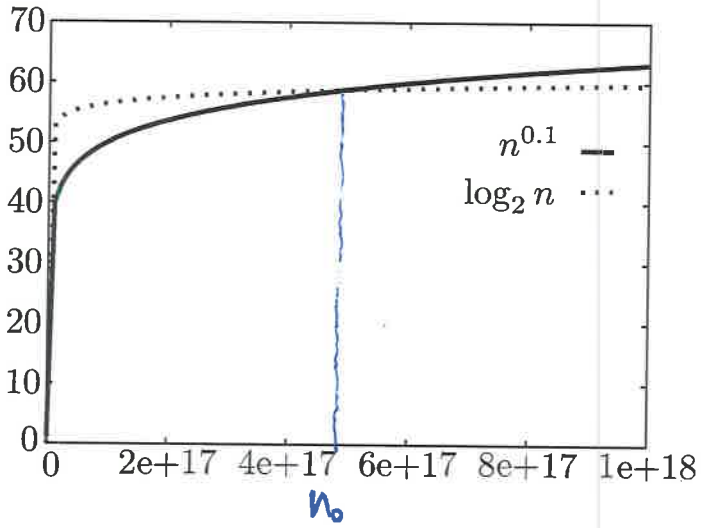
$n^{0.1}$ versus $\log_2 n$



Asymptotically Smaller? (cont.)

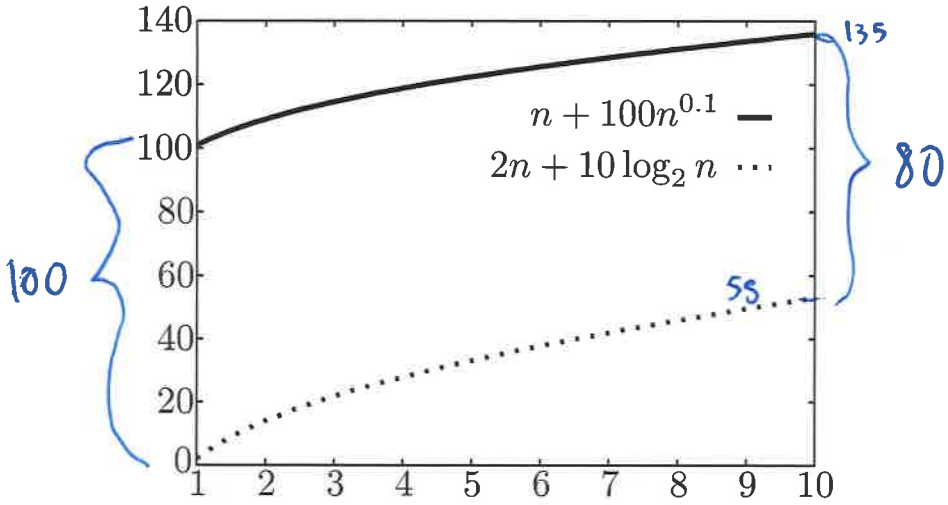
$n^{0.1}$ versus $\log_2 n$





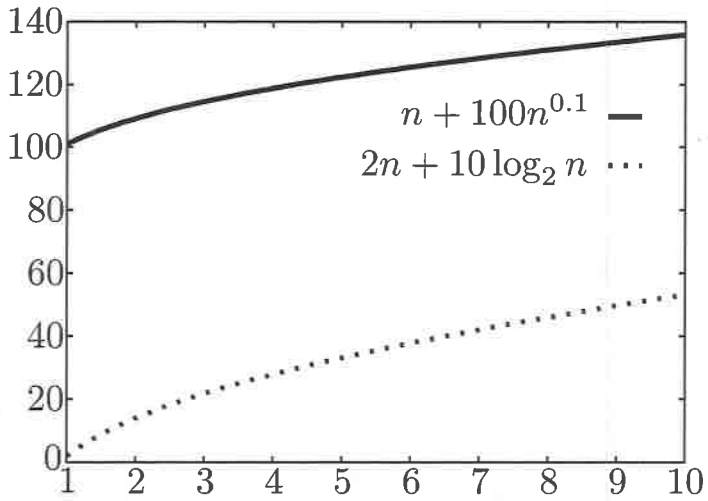
Asymptotically Smaller? (cont.)

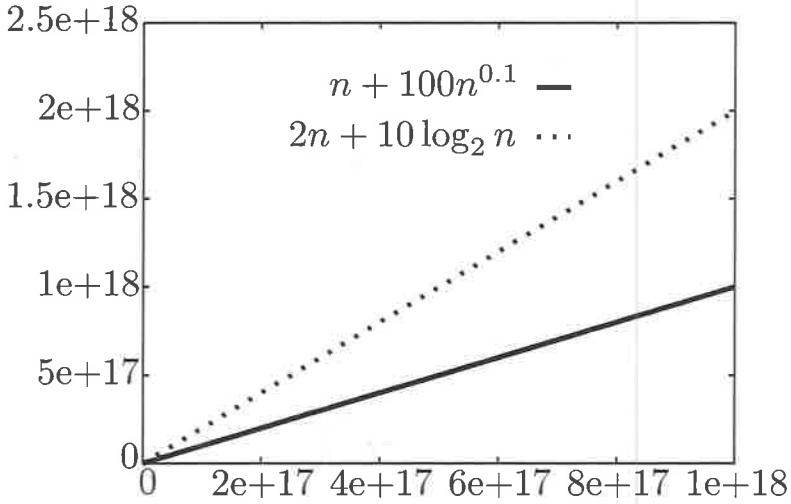
$$n + 100n^{0.1} \quad \text{versus} \quad 2n + 10\log_2 n$$



Asymptotically Smaller? (cont.)

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





Typical Asymptotics

Doable on a computer for "reasonable" sizes of n...

Tractable

- Constant: $\Theta(1)$ — looking up a value in an array, given the index; → push, pop, peek, enqueue, dequeue
- Logarithmic: $\Theta(\log n)$ ($\log_b n$, $\log n^2 \in \Theta(\log n)$) → binary search
- Poly-Log: $\Theta(log^k n)$ ($\log^k n \equiv (\log n)^k$)
- Linear: $\Theta(n)$ → find_min → because we have visit every element
- Log-Linear: $\Theta(n \log n)$ → many sorting algorithms
- Superlinear: $\Theta(n^{1+c})$ (c is a constant > 0)
- Quadratic: $\Theta(n^2)$ → less efficient sorting algorithms
- Cubic: $\Theta(n^3)$
- Polynomial: $\Theta(n^k)$ (k is a constant)

Intractable

Remember from Slide 12
- not doable unless n is really small

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

Sample Asymptotic Relations

$O(n^x)$ $O(n^{100})$

we want as tight a bound as possible.

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

n and 100n

n

Analyzing Code

- Single line 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
- Function call: time for the function

Above all, use common sense!

If we had to write down each line of code that was being executed from program start to finish how many lines of code would we need to write down?

Runtime Example #1

(1)

$\sum_{j=1}^{n} 1 = \underbrace{1+1+1+\dots+1}_{n} = n$ (j=1, j=2, j=3, ..., j=n)

$\sum_{i=1}^{n} n = \underbrace{n+n+n+\dots+n}_{n} = n^2$ (i=1, i=2)

loop n times

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

loop that runs n times

(2)

outer loop	inner loop
i = 1	j = 1 up to j = n ⇒ n times
i = 2	j = 1 to j = n ⇒ n times
i = 3	n times
⋮	
i = n	j = 1 to j = n ⇒ n times

n

$n \times n = n^2$

$T(n) = n^2$

Big-O: $T(n) = n^2 \leq n^2$

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

Big Ω: $T(n) = n^2 \geq n^2$

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

$\Rightarrow \Theta(n^2)$

Runtime Example #2

starting j value depends on the current i value.

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

n-1 times

n times?

the first time it will run n times, but the second time will be n-1, n-2, etc..

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

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

outer	inner
i = 1	j=1 to n ⇒ n times ✓
i = 2	j=2 to n ⇒ n-1 times ✓
	⇒ n-2 times
i = n-1	j=n-1 to n ⇒ 2 times

$n + (n-1) + (n-2) + \dots + 3 + 2$.

$2 + 3 + 4 + 5 + \dots + (n-1) + n = \frac{n(n+1)}{2} - 1$

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

$= \frac{n^2}{2} + \frac{n}{2} \leq \frac{n^2}{2} + \frac{n^2}{2}$

$\leq n^2$

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

Big-Ω:

$T(n) = \frac{n(n+1)}{2}$

$= \frac{n^2}{2} + \frac{n}{2} - 1 \geq \frac{n^2}{4}$ $n_0 \geq 2$

$\frac{4}{2} + \frac{2}{2} - 1 \geq \frac{4}{4}$

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

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

$i = i+1$ $i = 2*i$

1 2 4 8 16

Runtime Example #3

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

outer	inner
$i = 1$	$j = 1$ to $1 \Rightarrow 1$ time
$= 2$	$j = 1$ to $2 \Rightarrow 2$ times
$= 4$	$\Rightarrow 4$ times
$= 8$	
$\vdots$	$\vdots$
$i = 2^k$	$\Rightarrow 2^k$ times

$k \simeq \lg n$

sum of a geometric series

$T(n) = 1 + 2 + 4 + 8 + \ldots + 2^k$

$= 2^k + 2^{k-1} + \ldots + 8 + 4 + 2 + 1$ ← BINARY

$= (1\ 1\ 1\ \ldots\ 1\ 1\ 1\ 1)_2$ → What happens when we add 1 to this number?

$T(n)+1 = (1 + 0 + 0 + 0 + \ldots 0 + 0 + 0)_2 = 2^{k+1}$

① $2^k < n$

② $2^{k+1} \geq n$

$T(n) = 2^{k+1} - 1 = (2 * 2^k) - 1 < 2(n) - 1 = 2n - 1$ $\therefore T(n) \in O(n)$

$T(n) = 2^{k+1} - 1 \geq n - 1$ $\therefore T(n) \in \Omega(n)$ $\Rightarrow T(n) \in \Theta(n)$

Runtime Example #4

What is the max value in the array?

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

array — Base case — max number with problem size 1 shorter — c

Recursion almost always yields a recurrence relation:

base case B.C. — constant time — constant (maybe diff value than b)

$$T(1) \leq b$$
$$T(n) \leq c + T(n-1) \quad \text{if } n > 1$$

$T(n-1) \leq c + T(n-2)$

Solving the recurrence:

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

$cn - c + b$

further show proof by induction.

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

Runtime Example #5: Mergesort

$T(n) = [\#$ of operations to do left half $n/2] + [\#$ of ops to do right half $n/2] +$ # operations to merge 2 sorted arrays

Mergesort algorithm:

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

Recurrence relation:

BC

$$T(1) \leq b$$

merging

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

$T(n/2) \leq 2T(n/4) + cn/2$

Solving recurrence:

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

general case

BC

$bn + cn\lg n \rightarrow$ Big-O analysis: Throw away smaller terms.

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

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)
```

Recurrence Relation: (lower bound)

$$T(0) \geq b$$
$$T(1) \geq b$$
$$T(n) \geq T(n-1) + T(n-2) + c \qquad \text{if } n > 1$$

Claim:

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

n is the exponent ⇒ growth rate of function is exponential

golden ratio

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

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

$$\varphi * \varphi = \left(\frac{1+\sqrt{5}}{2}\right) * \left(\frac{1+\sqrt{5}}{2}\right) = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = \frac{2}{2} + \frac{1+\sqrt{5}}{2} = 1+\varphi$$

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

Claim:

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

Proof: (by induction on n)

Base Case: $T(0) \geq b > b\varphi^{-1}$ 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 $n = k + 1$.

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

$O(n^3) \neq O(n^2)$

Note: $O(3^n) \neq O(2^n)$

even $O(3n) = O(2n) = O(n)$

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

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 (array)
- 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.

→ Can look up and access those values in $O(1)$.

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

How much time does this version take?

$O(n)$ because we compute each value up to n once.

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:

n letters

$A =$ `search me`

m letters

$B =$ `insane method`

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

Applications of LCS:

DNA sequencing, revision control systems, `diff`, ...

"Naive" approach / Brute Force / exhaustive search

Algorithm:

- Get every subsequence of A (2^n)
  - Get every subsequence of B (2^m)
    - Check if they match
    - (Remember the longest match)

Runtime Example #8: LCS (cont.)

An Algorithm and Its Analysis:

How many subsequences can we make?

"abc" ⇒ binary – every letter is either included in the sequence or not

A has n letters

bin	
000	""
001	"c"
010	"b"
011	"bc"
100	"a"
101	"ac"
110	"ab"
111	"abc"

$2^n - 1$ different subsequences for A

B has m letters

$2^m - 1$ different subsequences for B

$2^n + 2^m * \min(n, m)$

Example #9

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

$\lg(x/y) = \lg(x) - \lg(y)$

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

① Big-O: $T(n) = \lg(n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1)$

$= \lg n + \lg(n-1) + \lg(n-2) + \ldots + \lg 2 + \lg 1 = \sum_{i=1}^{n} \lg(i)$

$\leq \lg n + \lg n + \lg n + \ldots + \lg n + \lg n = \sum_{i=1}^{n} \lg(n)$

$= n \lg n \in O(n \lg n)$

② Big-Ω:

$T(n) = \lg n + \lg(n-1) + \lg(n-2) + \ldots + \lg 2 + \lg 1$

$\geq \lg(n/2) + \lg(n/2) + \lg(n/2) + \ldots + 0 + 0 + 0 + \ldots + 0 + 0$ ← setting the second half to 0

$= n/2 \lg(n/2) = n/2 (\lg n - \lg 2)$ ← $\lg 2 = 1$

$= n/2 (\lg n - 1) \geq n/4 \lg n$ for $n_0 \geq 4$, $c = 1/4$ $\therefore \Omega \in O(n \lg n)$

③ Big-Θ

① and 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

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

$\log_2 n = \frac{\log_{10} n}{\log_{10} 2}$

divide by a constant

for constants $b, c > 1$.

"throw" away constants.

Asymptotic Analysis Summary

↳ As n grows towards infinity

↳ above n_0

- 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
  - Average case

quick-sort. nlogn

worst-case: n^2

- 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).

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

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

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

Chew these over:

- How many values can you represent with c bits? 2^c
- 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? $\Rightarrow \lg(n!)$ bits of information to compare $\in \Omega(n \lg n)$ (from Example 9)

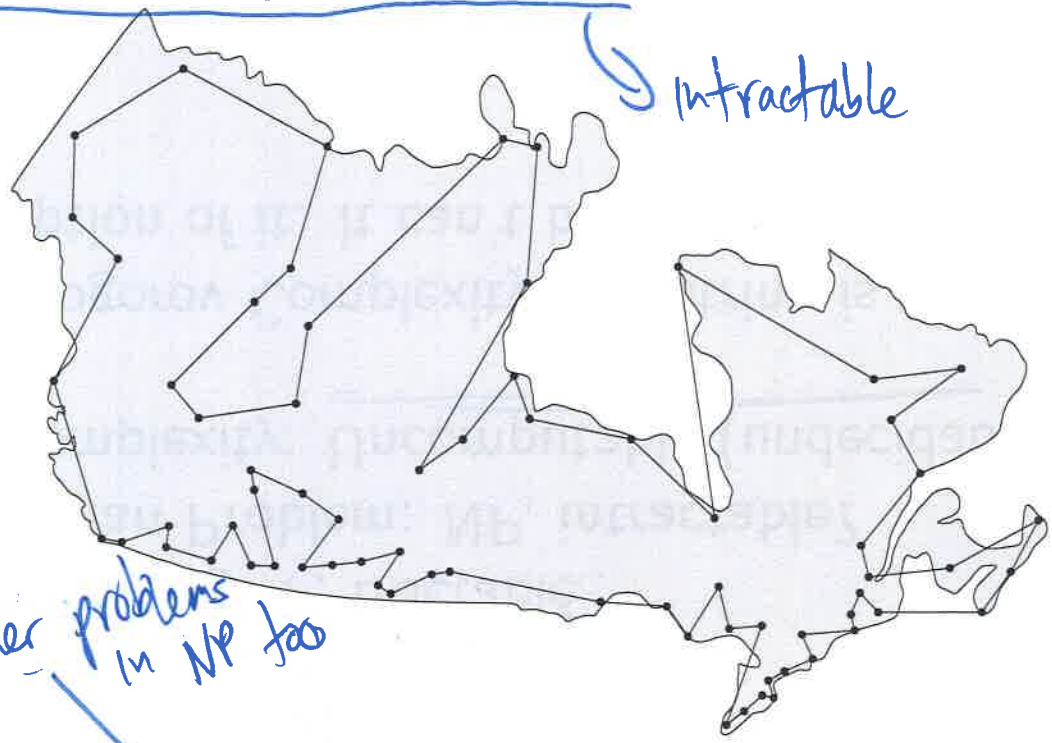
Problem Complexity

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/`

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

intractable



other problems in NP too

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

Can we create a solution to this type (NP-problem) and prove that it works in polynomial time

Problem Complexity

polynomial

- 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.