Unit #1: Complexity Theory and Asymptotic Analysis

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Brief proof reminder
- Algorithm Analysis: Counting steps
- Asymptotic Notation
- Runtime Examples
- Problem Complexity

Learning Goals

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

Proof by ...

Review

- 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.

example: $n < n$

- Induction
  - prove for a base case (e.g., $n = 1$)
  - assume for all $n \leq k$ (for arbitrary k)
  - prove for the next value ($n = k + 1$)
  - Thus, the theorem is true.

induction step

Example: Proof by Induction (worked) 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.

Believe it.

x	$S(x)$
12	3
23	5
234	9

$\% 3 = 2$

Proof:

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

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

We'll prove the stronger result:

$$\boxed{S(x) \bmod 3 = x \bmod 3.}$$

How do we use induction?

Example: Proof by Induction (worked) 2/4

Base Case:

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

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

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

Example: Proof by Induction (worked) 3/4

Inductive hypothesis:

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

$$\underline{S(x) \bmod 3 = x \bmod 3.}$$

Not a proof!

Example:

$x = 234$

$x_1 = 2, x_2 = 3, x_3 = 4$

$z = 23$

$10z = 230$

$10z + 4 = 234 = x$

Inductive step:

Consider a number $\underline{\underline{x}}$ with $n = \underline{\underline{k+1}}$ digits:

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

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

by i.h.: $$S(z) \bmod 3 = z \bmod 3.$$

$$x = 10 \cdot z + x_{k+1}$$

Example: Proof by Induction (worked) 4/4

Inductive step (continued):

$$
\begin{aligned}
\underline{x \bmod 3} &= (10z + x_{k+1}) \bmod 3 && (x = 10z + x_{k+1}) \\
&= (9z + z + x_{k+1}) \bmod 3 \\
&= (\overset{0+}{z} + x_{k+1}) \bmod 3 && (9z \text{ is divisible by } 3) \\
&= (\overset{\text{i.h.}}{S}(z) + x_{k+1}) \bmod 3 && (\text{induction hypothesis}) \\
&= (x_1 + x_2 + \cdots + x_k + x_{k+1}) \bmod 3 \\
&= S(x) \bmod 3
\end{aligned}
$$

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

Induction is used to prove correctness / running time of algorithms that use loops or recursion.

A Task to Solve and Analyze

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

operations:
- find
- insert
- delete

Student object

ID
name

- array of objects
- Dictionary → sorted by ID's
- (Balanced) Search Tree
- Skiplists
- Linked list
- Hashmap/hashtable

- Does it matter? ∴ YES
- How to compare speed of two algorithms?
  - express runtime as function of the size of the input
  - 2n 1000logn

Analysis of Algorithms

- Analysis of an algorithm gives insight into
  - how long the program runs (time complexity or runtime) and
  - how much memory it uses (space complexity).
- Analysis can provide insight into alternative algorithms
- Input size is indicated by a non-negative integer n (sometimes there are multiple measures of an input's size)
- Running time is a real-valued function of n such as:
  - $T(n) = 4n + 5$
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$

operations

ms

# of lines of code

will not matter when using asymptotic (O, Ω, Θ) notation

Rates of Growth

10^{-12} s

Suppose a computer executes 1op per picosecond (trillionth):

size of input

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

T(n)

nano second

10^{-9}

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

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

Exasecond(Es) $=$ 32 billion years

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

$n =$	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}

Exasecond(Es) $=$ 32 billion years

10^{18}

10^{24} yottasecond

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

$n =$	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	

Exasecond(Es) $=$ 32 billion years

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

$n =$	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		

Exasecond(Es) $=$ 32 billion years

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

$n =$	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			

Exasecond(Es) $=$ 32 billion years

Rates of Growth

Suppose a computer executes 1op per picosecond (trillionth):

$n =$	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				

Exasecond(Es) = 32 billion years

7.5×10^9

DNA 3×10^9

Analyzing Code

T(n) = # 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
```

is this expensive?

→

1) What's the input size, n?

n = size of the array

(k = size of keys)

optional

let's assume == takes constant time

Analyzing Code

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

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

Analyzing Code

n = # size of array

```
// Linear search
find(key, array)
  for i = 0 to length(array) - 1 do    ← n times
    if array[i] == key                  n times
      return i
  return -1    ← 1 time
```

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

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

Are lines the right unit?

Analyzing Code

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

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

- Does the "1" matter? .. as n gets big, 2n term will dominate → NO
- Does the "2" matter?

→ hardware also changes this constant → NO

Big-O Notation .. allows us to abstract from things that don't matter (usually)

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

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

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

for big enough values of n

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

$T(n) = 2n+1$ $\qquad f(n) = n$

"$2n+1$ grows no faster than n".

Claim: $2n+1 \in O(n)$

Proof: $2n+1 \leq \underbrace{3}_{c} \cdot n$ for $n \geq \underbrace{1}_{n_0}$

$\Updownarrow$ if and only if

iff

$1 \leq n$

Asymptotic Notation

Claim:

$T(n) \in \Omega(f(n)) \Leftrightarrow f(n) \in O(T(n))$

big-O

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

$T(n)$ "$\leq$" $f(n)$

big-omega

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

$T(n)$ "$\geq$" $f(n)$

big-theta

$T(n)$ "$=$" $f(n)$

- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.

little-o

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

$T(n)$ "$<$" $f(n)$

little-omega

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \begin{cases} 0 & T(n) \in o(f(n)) \\ 0 < x < \infty & \therefore T(n) \in \Theta(f(n)) \\ \infty & \therefore T(n) \in \omega(f(n)) \end{cases}$$

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

$T(n)$ "$>$" $f(n)$

Example: $n^{1+\sin(n)}$ vs. n $\therefore$ cannot be compared

Examples

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

O: $10{,}000n^2 + 25n \le 10{,}001\, n^2$ (c)

$\Updownarrow$

$25n \le n^2$

$\Updownarrow$

$25 \le n$ (n_0)

$10{,}000n^2 + 25n \le 10025\, n^2$ (c)

$\Updownarrow$

$25n \le 25n^2$

.. for $n \ge 1$ (n_0)

Ω: $10{,}000n^2 + 25n \ge 10{,}000\, n^2$ (c) for $n \ge 1$ (n_0)

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

pos. constant

$c = 10^{-10}$ $n_0 = 1$ (O, Ω)

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

$n \log n \le c \cdot n^2$

$\Updownarrow$

$\log n \le c n$

true for $\begin{cases} c = 1 \\ n_0 = 1 \end{cases}$

to verify consider rates of growth

see next page

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

$n \log n \ge c n$

$\Updownarrow$

$\log n \ge c$ ...... true for $\begin{cases} c = 1 \\ n \ge 2 = n_0 \end{cases}$

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

$\lim_{n\to\infty} \frac{n^3+4}{n^4} = 0$

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

$\lim_{n\to\infty} \frac{n^3+4}{n^2} = \infty$

		x
plot		$\log_2(x)$

will win in long run!

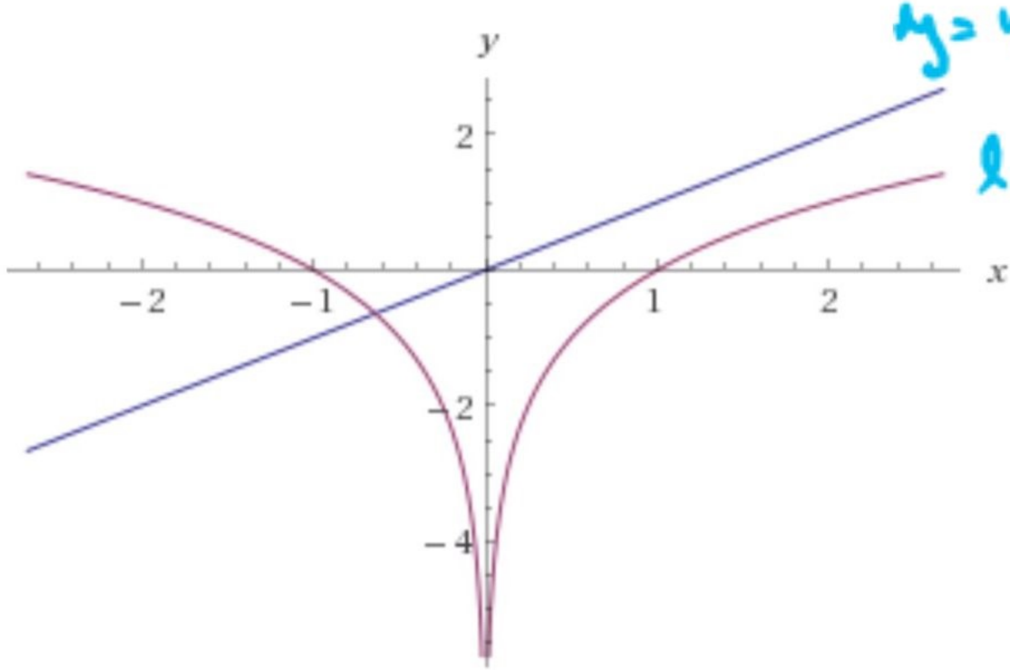
$\log_b(x)$ is

$(x)' = 1$

rate of growth

Plot:

$y = x$



$\log_2 x$

$(\ln x)' = \frac{1}{x}$

rate of growth

(x from −2.664 to 2.664)

— $\mathrm{Re}(x)$

— $\frac{\mathrm{Re}(\log(x))}{\log(2)}$

Analyzing Code

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

worst-case

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

worst-case
$\underline{T(n) \in \Theta(n)}$

$T(n) \in O(n)$ .. my alg. is fast .. upper bound

$T(n) \in \Omega(n)$ .. your alg. is slow .. lower bound

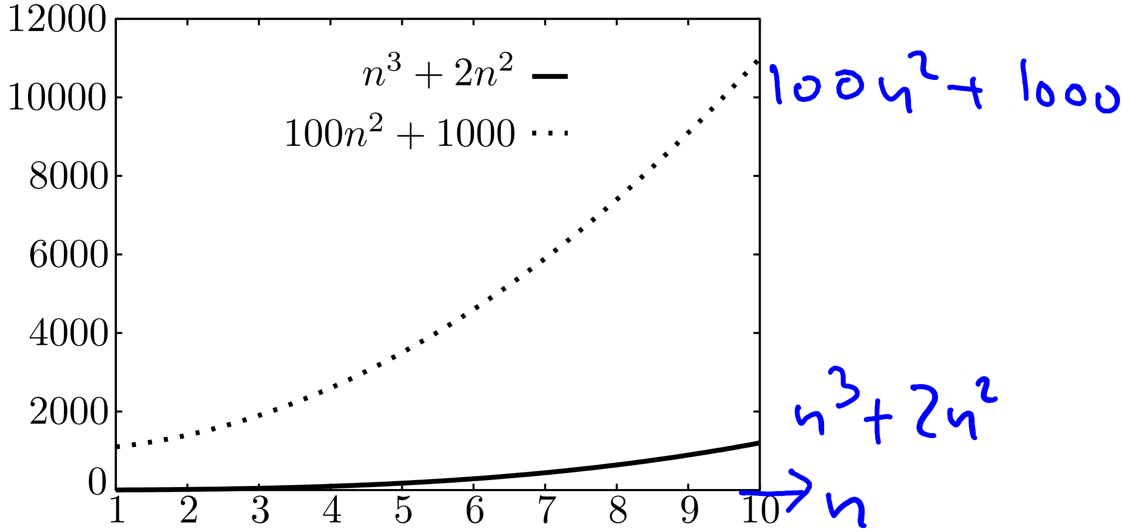
$T(n) \in \Theta(n)$ .. exact answer

Q. what can we say about running time of linear search alg. if we do not consider the worst-case? A. $T(n) \in O(n)$.
$T(n) \in \Omega(1)$.

Asymptotically smaller?

$$n^3 + 2n^2 \qquad \text{versus} \qquad 100n^2 + 1000$$

$\in \; ? \; (\qquad\qquad)$



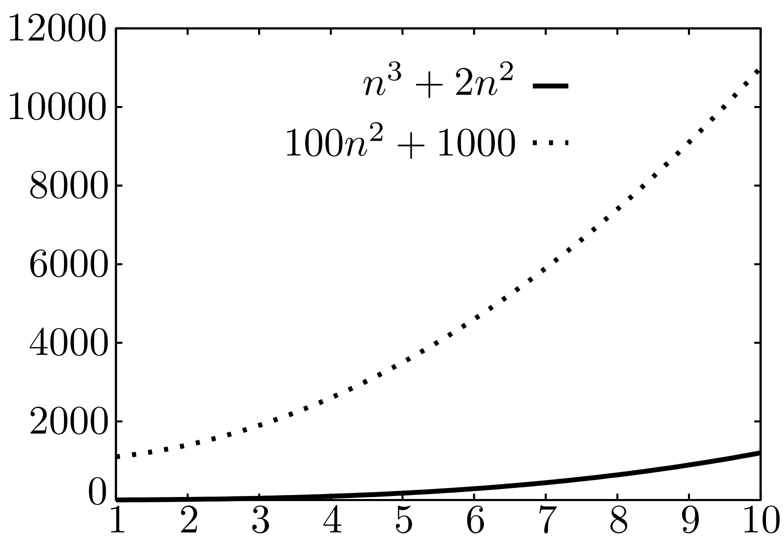
Asymptotically smaller?

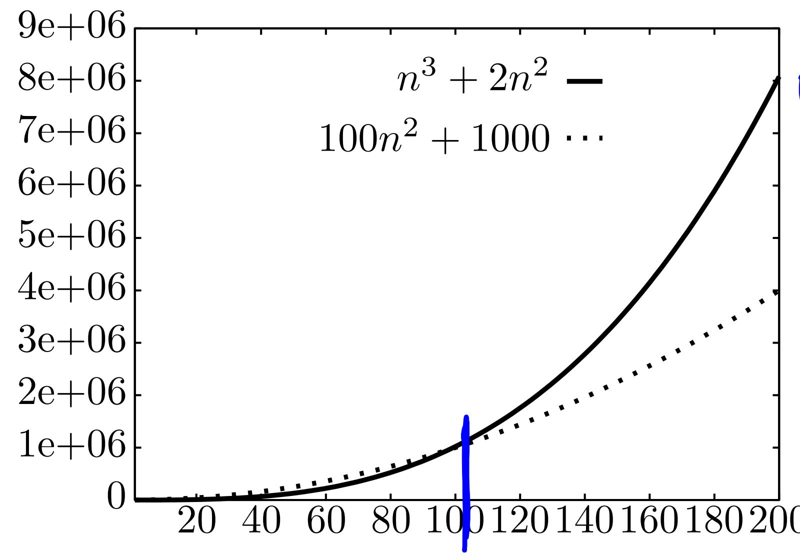
ignore lower order terms

$$n^3 + \cancel{2n^2} \quad \text{versus} \quad \cancel{100}\,n^2 + \cancel{1000}$$

$\in \Omega(\quad)$

ignore multiplicative constant





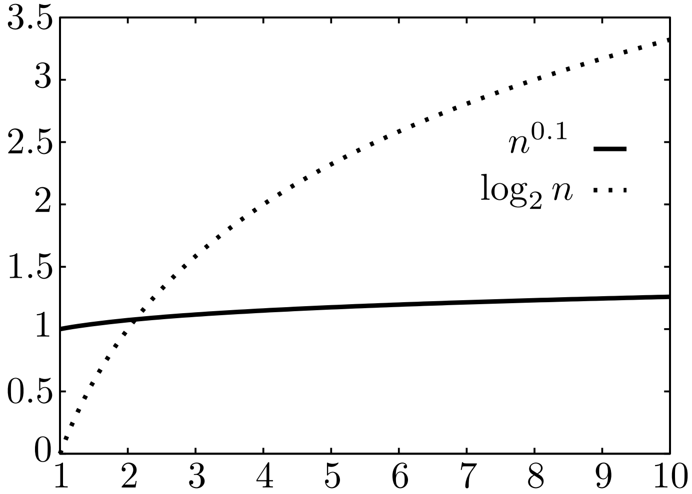
$n^3 + 2n^2$

$c = 1$

$n_0 \sim 100$

Asymptotically smaller?

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



$n^{0.1}$

Asymptotically smaller?

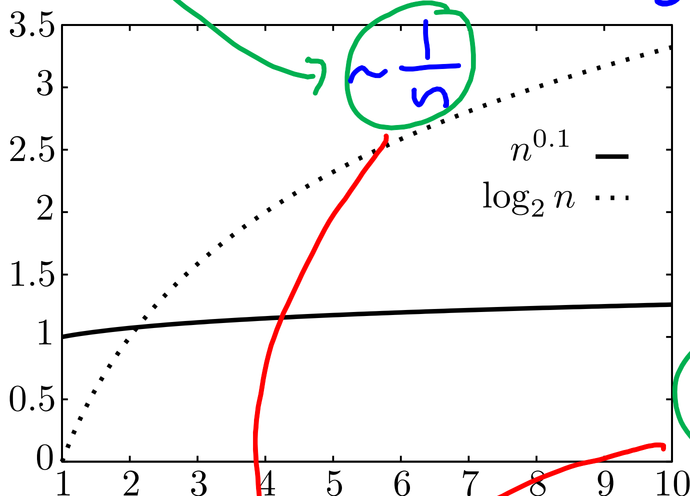
$n^{0.1}$ versus $\log_2 n$

$n^{0.1} \in \Omega(\log_2 n)$ or

$\log_2 n \in O(n^{0.1})$

rate of growth

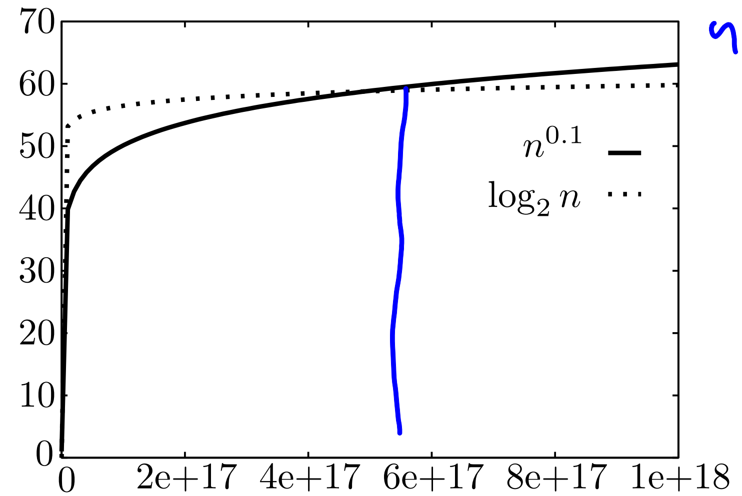
$\sim \frac{1}{n}$



$n^{0.1}$

$\sim \frac{1}{n^{0.9}}$

rate of growth



$n^{0.1}$

$-1 < -0.9$

$\Rightarrow \log_2 n$ is growing slower than $n^{0.1}$

(same as n^2 is growing slower than n^3)

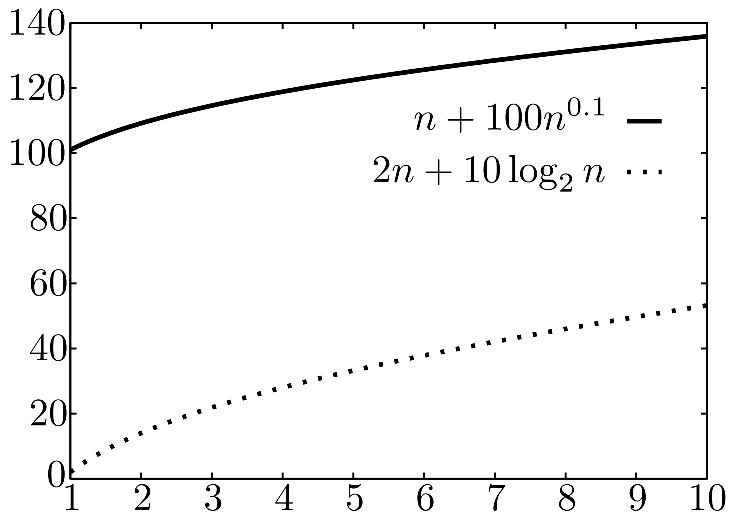
$c = 1$

$n_0 = 5 \times 10^{17}$

Asymptotically smaller?

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

$\in$?

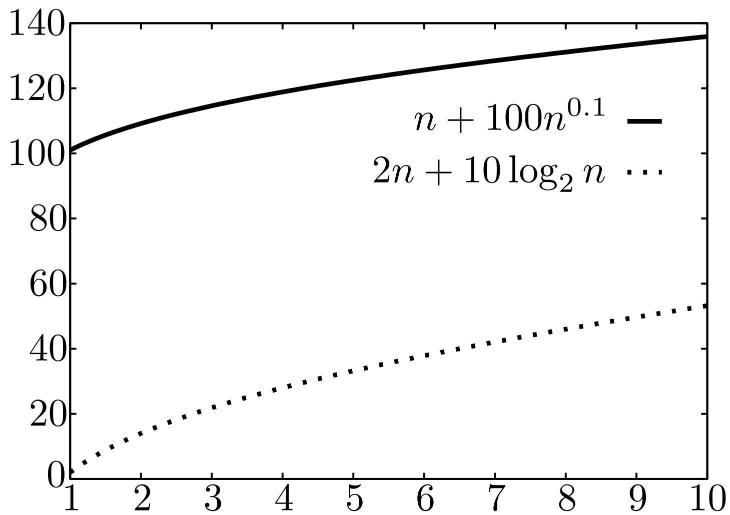


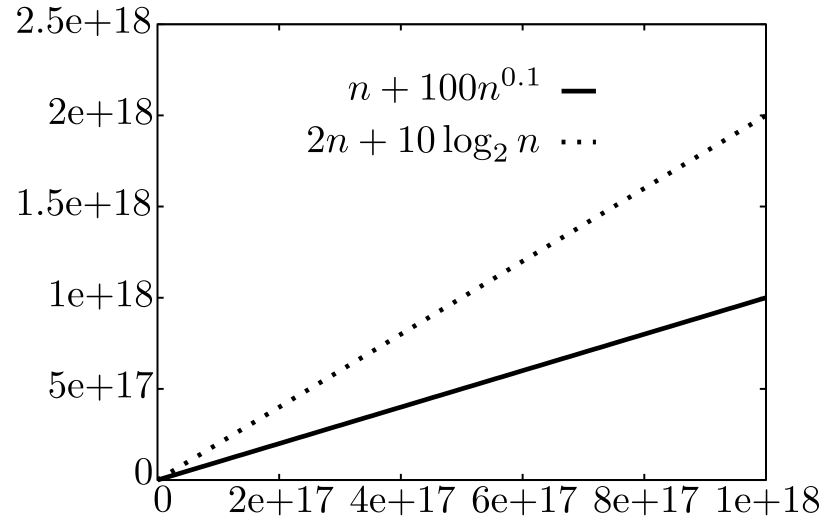
Asymptotically smaller?

lower order terms can be ignored

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

$$\in \Theta(\qquad\qquad)$$





Typical asymptotics

Tractable

sublinear

- constant: $\Theta(1)$
- logarithmic: $\Theta(\log n)$ ($\log_b n$, $\log n^2 \in \Theta(\log n)$)

  $\log_b n = \frac{\log_a n}{\log_a b}$ ← a constant

  ⇒ the base of log doesn't matter
- poly-log: $\Theta(log^k n)$ ($\log^k n \equiv (\log n)^k$)
- linear: $\Theta(n)$
- log-linear: $\Theta(n \log n)$
- superlinear: $\Theta(n^{1+c})$ (c is a constant > 0)

  $\underbrace{n^2 \in O(n^3),\ n^2 \notin \Theta(n^3)}_{n^2 \in o(n^3)}$
- quadratic: $\Theta(n^2)$
- cubic: $\Theta(n^3)$
- polynomial: $\Theta(n^k)$ (k is a constant)

$\Theta(n^{\log n})$ quasipolynomial ... graph alg. for the isomorphism

Intractable

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

  $\underbrace{2^n \in O(3^n), \text{ but } 2^n \notin \Theta(3^n)}_{2^n \in o(3^n)}$

Sample asymptotic relations

- $\{1, \log n, n^{0.9}, n, 100n\} \subset \overset{\text{``}\leq\text{''}}{O}(n)$
- $\{n, n \log n, n^2, 2^n\} \subset \overset{\text{``}\geq\text{''}}{\Omega}(n)$
- $\{n, 100n, n + \underline{\log n}\} \subset \overset{\text{``}=\text{''}}{\Theta}(n)$
- $\{1, \log n, n^{0.9}\} \subset \overset{\text{``}<\text{''}}{o}(n)$
- $\{n \log n, n^2, 2^n\} \subset \overset{\text{``}>\text{''}}{\omega}(n)$

Analyzing Code

- single operations: constant time
- consecutive operations: sum operation times
- conditionals: condition time plus max of branch times
- loops: sum of loop-body times
- function call: time for function

in worst-case

Above all, use your head!

Q

```
string a,b;
  :
if (a == b) ...
  :
```

linear in length of a & b

Runtime example #1

Counting all lines executed!

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

$\leftarrow 1$

$$\sum_{j=1}^{n(1+1)} 2 = 2n$$

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

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

$$2n^2 + n$$

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

Runtime example #2

Let $T(n)$ be how many times sum=sum+1 is executed?

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

1

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

inner loop

sum of arithmetic series

outer loop

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

$$= \frac{n+2}{2} \cdot (n-1) = \frac{n^2}{2} + \frac{n}{2} - 1$$

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

$$S = a + (a+1) + \dots + (b-1) + b$$

$$S = b + (b-1) + \dots + (a+1) + a$$

$$2S = (a+b) + (a+b) + \dots + (a+b)$$

$$S = \frac{a+b}{2} \cdot \#\text{ of terms}$$

how to sum any arithmetic series

Runtime example #3

count sum = sum+1 executions!

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

i

$i = 1, 2, 4, 8, 16, \ldots$

$T(n) = 1 + 2 + 4 + 8 + 16 + \ldots + 2^k$, where $2^k < n$ but $2^{k+1} \geq n$

$k \approx \lg n$ ← apply $\log_2$

$= (1\,1 \ldots 1\,1)_2$ ← binary representation (digits $2^k\, 2^{k-1} \ldots 2^1\, 2^0$)

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

Hence, $T(n) = 2^{k+1} - 1 \geq n - 1$

$T(n) = 2 \cdot 2^k - 1 < 2n - 1$

So: $n - 1 \leq T(n) < 2n - 1 \implies T(n) \in \Theta(n)$ (implies)

If you are curious what the value k is: $k = \lceil \lg n \rceil - 1 = \lfloor \lg(n-1) \rfloor$

Runtime example #4

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

Recursion almost always yields a recurrence relation:

if counting lines:
$b=1$
$c=2$

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

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

Solving recurrence:

Substitution method:

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

≈ guessing

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

Claim: $T(n) \le (n-1)c + b$

Proof by induction on:

- Base case: $n=1$ $\quad T(1) \overset{\text{recurrence}}{\le} b = (1-1)c+b$
- Inductive step:

  Induction hypothesis (i.h.):

  for every $n \le k$: $T(n) \le (n-1)c + b$

  We need to prove the claim for $n = k+1$.

$$T(n) = T(k+1) \overset{\text{recurrence}}{\le} c + T(k) \overset{\text{by i.h.}}{\le} c + (k-1)c + b$$

$$= kc + b = (n-1)c + b.$$

QED

Runtime example #5: Mergesort

Mergesort algorithm:

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

Time: $\Theta(n)$

Recurrence relation:

time for splitting & merging

$$T(1) \le b$$
$$T(n) \le 2T(n/2) + cn \qquad \text{if } n > 1$$

$T(n/2) \le 2T(n/4) + c\,n/2$

Solving recurrence:

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

$\le nb + cn \lg n$

$n/2^k = 1$

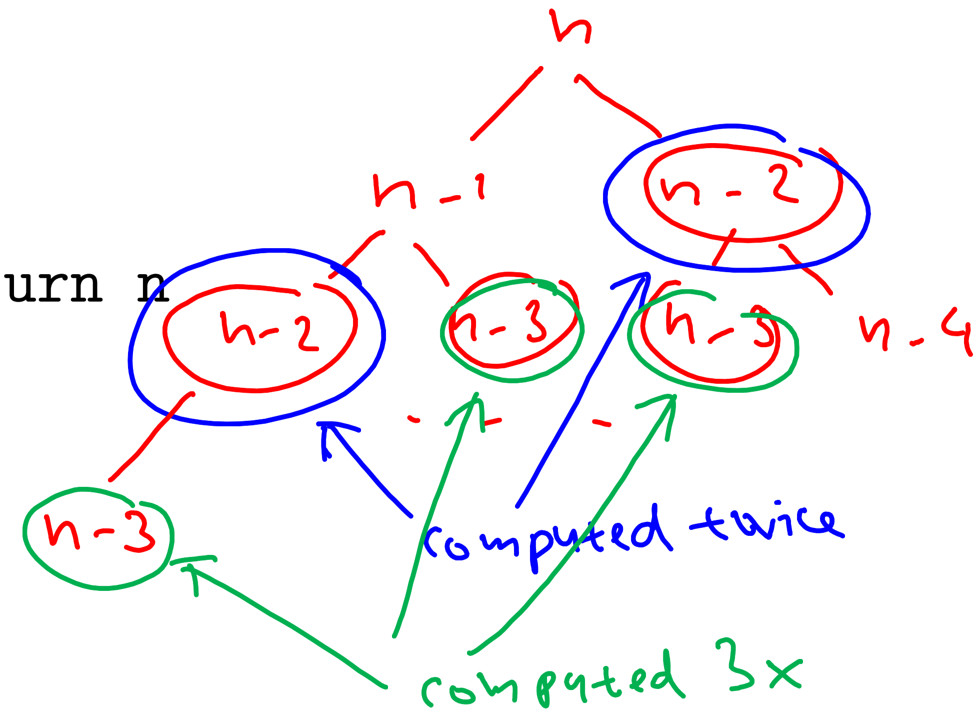
$k = \lg n$

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

Runtime example #6: Fibonacci 1/2

Recursive Fibonacci:

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



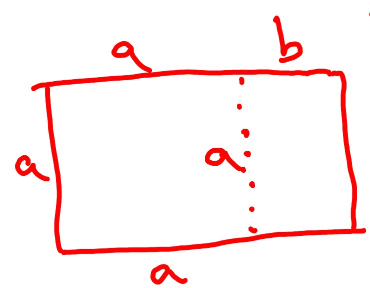
Recurrence relation: (lower bound)

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

⋮ gets worse as we recurse down

Claim:

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



where $\varphi = (1+\sqrt{5})/2$. .. golden ratio
Note: $\varphi^2 = \varphi + 1$. $\Longleftarrow$ $\varphi = \frac{a}{b} = \frac{a+b}{a} = 1 + \frac{b}{a} = 1 + \frac{1}{\varphi}$

Runtime example #6: Fibonacci 2/2

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 hyp: Assume $T(n) \geq b\varphi^{n-1}$ for all $n \leq k$.
Inductive step: Show true for $n = k + 1$.

$$
\begin{aligned}
T(n) &\overset{\text{recurrence}}{\geq} T(n-1) + T(n-2) + c \\
&\geq b\varphi^{n-2} + b\varphi^{n-3} + c && \text{(by } \overset{2x}{\text{inductive}} \text{ hyp.)} \\
&= b\varphi^{n-3}(\underline{\varphi + 1}) + c \\
&= b\varphi^{n-3}\underline{\varphi}^2 + c && \text{golden ratio property} \\
&\geq b\varphi^{n-1}
\end{aligned}
$$

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

Why? Same recursive call is made numerous times.

Example #7: Learning from analysis

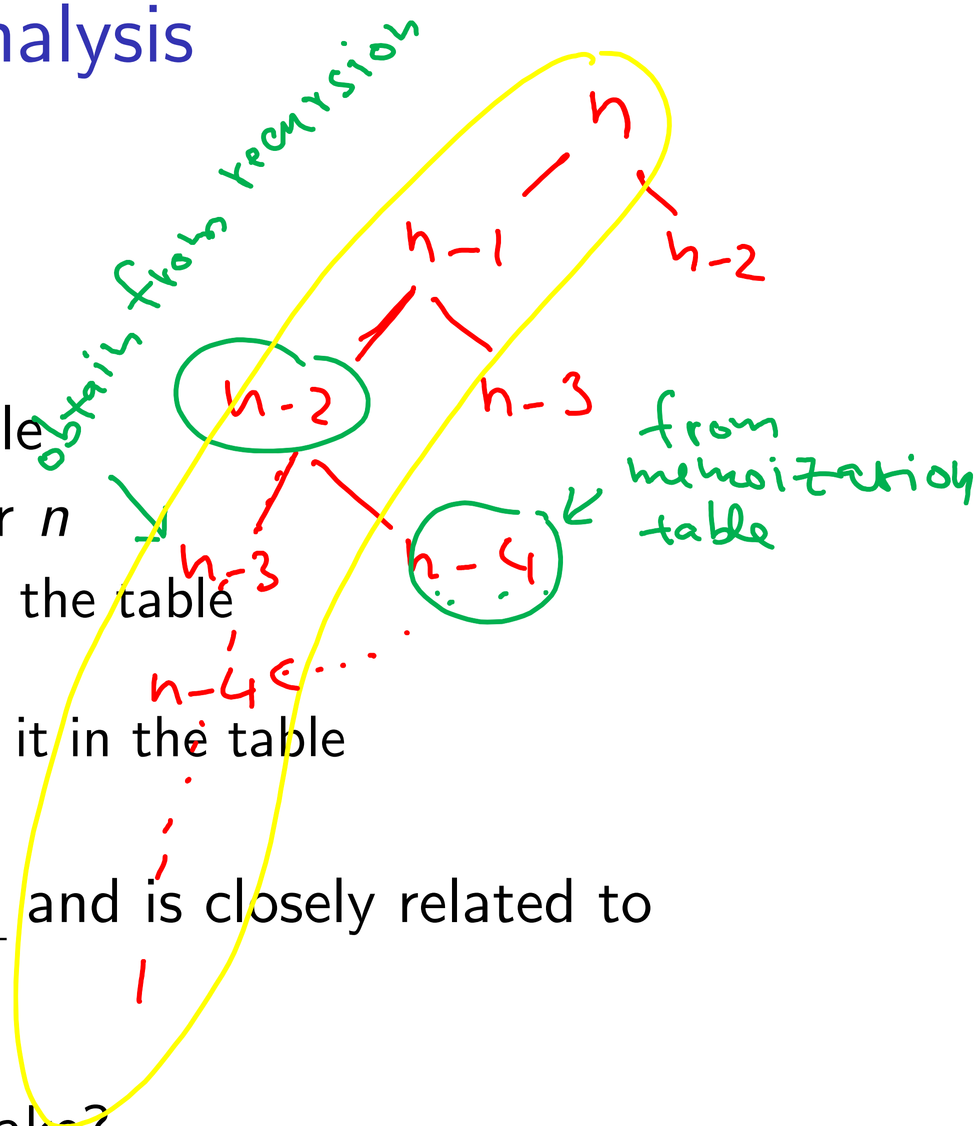
To avoid recursive calls

- 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

This strategy is called <u>memoization</u> and is closely related to <u>dynamic programming</u>.

How much time does this version take?

$\Theta(n)$



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.

$|A| = n$ .. 2^n $|B| = m$

Example:

binary representation → 1 0 1 0 0 0 1 1

to generate all subsequences iterate through numbers from 0 to $2^n - 1$

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

"sa me" ← if we do not ignore the space

A longest common subsequence is "`same`" (so is "`seme`")

"net"

Applications:

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

"worst-case"

Alg. 1:

For every subsequence S of A — $2^n \times$

  For every subsequence S' of B — $2^m \times$

    If $S = S'$ — $\Theta(\min(n,m))$

      remember the longest so far

$T(n) \in \Theta(2^n \cdot 2^m \cdot \min(n,m))$

Example: subsequences of "abc"

$0 .. 2^3 - 1 = 7$

n	bin	subsequence
0	000	""
1	001	"c"
2	010	"b"
3	011	"bc"
4	100	"a"
5	101	"ac"
6	110	"ab"
7	111	"abc"

Best alg. using DP : $\Theta(nm)$

Alg 2.:

For every subsequence S of A $\times 2^n$

if S is a subsequence of B

remember the longest one

$T(n) = \Theta(2^n \cdot m)$

greedy approach:

- find first occ of S[0] in B
- for i=1 to length(S)-1

  find first occ of S[i] in B after occ of S[i-1]

$\Theta(m)$

Example #9

$\lg(a \cdot b) = \lg a + \lg b$

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

$= \lg(n \cdot (n-1)(n-2) \dots 2 \cdot 1)$

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

$$= \sum_{i=1}^{n} \lg(i) \le \sum_{i=1}^{n} \lg(n) = n \lg n \in O(n \log n)$$

$$\ge \sum_{i=n/2}^{n} \lg(i) \ge \sum_{i=n/2}^{n} \lg(n/2)$$

$$= n/2 \cdot \lg(n/2)$$

$$= n/2 \cdot (\lg n - 1) \in \Omega(n \log n)$$

$n \ge 4 \; n_0$

$\Rightarrow$

$\lg n \ge 2$

$$= \frac{1}{2} n \lg n - \frac{n}{2} \ge c\, n \lg n$$

$$\frac{1}{4} n \lg n + \underbrace{\frac{1}{4} n \lg n - \frac{n}{2}}_{\ge 0} \ge \frac{1}{4} n \lg n$$

c

Log Aside

$\log_b x$ is the exponent b must be raised to 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 10 fingered mammals)
- $\ln x \equiv \log_e x$ (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 $b, c > 1$.

Asymptotic Analysis Summary

- Determine what is the input size
- Express the resources (time, memory, etc.) an algorithm requires as a function of input size
  - worst case
  - best case
  - 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 for the 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.)
- 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:

2^c values

- How many 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?

$\geq \lg(n!)$ bits of information

comparisons

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

Problem Complexity

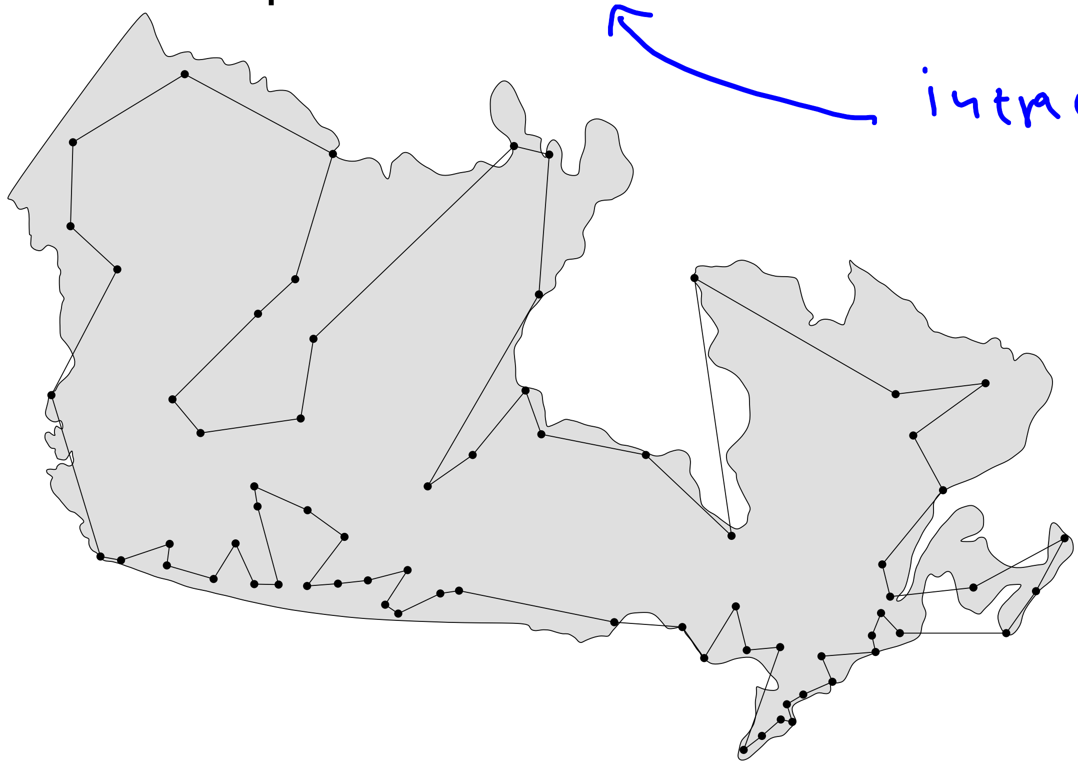
P $\in O(n^c)$

Sorting: solvable in polynomial time, tractable
Traveling Salesman Problem (TSP): In 1,290,319km, 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.

NP

intractable



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

Example 1:
Kolmogorov Complexity of a string is the length of the shortest description of it.

Can't be computed. Pithy but hand-wavy proof: What's:

> *The smallest positive integer that cannot be described in fewer than fourteen words* ← google: Berry Paradox

Example2:
Halting problem : Given a code, decide if it stops.