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. (e.g., n < n)
  - 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*.

X is true for n=1
If X is true for n=k, then X is true for n=k+1
Therefore,
X is true for n=1, 2, 3, 4, ...

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.

Examples:

x	S(x): sum of digits of x	Divisible by 3
12	3	√
17	8	-
171	9	√
12003	6	√

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.

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

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

"x is divisible by 3" means:
x mod 3 = 0

We'll prove the stronger result:

So we only need to show that:
S(x) mod 3 = 0 iff x mod 3 = 0

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

Above is "stronger results" because we are showing more than needed!

How do we use induction?

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

Inductive Hypothesis:

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

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

Example:

x=876
k=2

z=87

Inductive Step:

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

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

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

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

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

Inductive Step (continued):

e.g.:
x = 876 = 10 · 87 + 6

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

e.g. $\boxed{(6+2) \bmod 3 = 2 \bmod 3 = 2}$

i.e. $\boxed{S(z) \bmod 3 = z \bmod 3}$

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

$\boxed{\text{Because we have proved both the Base Case and Inductive Step.}}$

Induction is used to prove the correctness and 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.

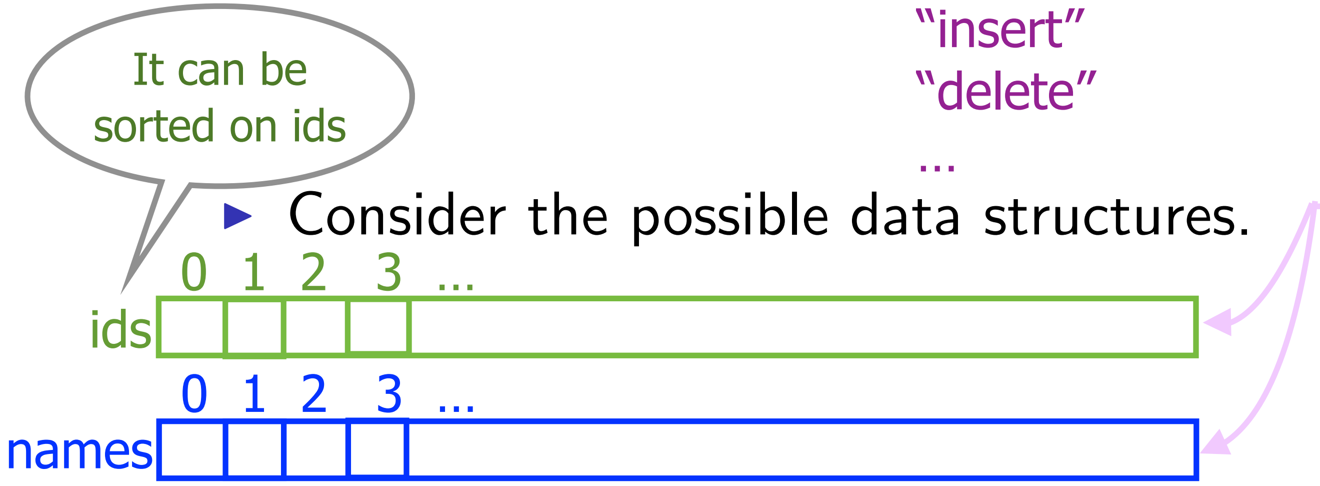
- Consider the data that you need to store.

  id , name

- Consider the operation.

  “search” but also probably needed:
  “insert”
  “delete”
  …

- Consider the possible data structures.



- Two arrays: ids and names
- Array of key-value pairs: <id, name>
- List of key-value pairs: <id, name>
- Dictionary
- Map
- …

- Does it matter which data structure we use?

Yes, because some are faster (more efficient) for this problem when it gets large enough

How can we compare them?

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.

e.g.:
- milliseconds
- number of operations to run
- number of lines of code to run
- amount of memory needed
- ...

We will see that the exact values for such qualities do not matter when using asymptotic notation (O, Θ, Ω, ...)

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:
  - $T(n) = 4n + 5$
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$

E.g., (see slide 9)
n: number of students
T(n): time to find one student

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., $T(n) = \log n$):

T(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., $T(n) = \log n$):

T(n)	n= 10	100
$\log n$	1ps	2ps
n	10ps	100ps
$n \log n$	10ps	200ps
n^2	100ps	10ns
2^n	1ns	$\underline{\text{1Es}}$ "intractable"

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., $T(n) = \log n$):

T(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	$\underline{\text{1Es}}$	$\underline{10^{289}\text{s}}$

“intractable”

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., $T(n) = \log n$):

T(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., $T(n) = \log n$):

T(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., $T(n) = \log n$):

T(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., $T(n) = \log n$):

T(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				

Real life examples:

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

Human genome size:
n = 3 × 10^9 base pairs

Pine tree genome size:
n = 23 × 10^9 base pairs

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 checking for equality == expensive?

1) What's the input size *n*?

n = size of the input "array"

Also, k = size of "key" (optional because we can assume all keys have max key size)

Thus, let's assume that here == takes <u>constant time</u>

Analyzing Code

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

Rarely useful   Sometimes useful

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

Answer: worst-case
because n does not tell us enough about the input.

For example, for an given array with size n:
— The key may be located at array[0]
— The key may not be in the array at all

Thus, the results of worst case analysis is meaningful for any given n — it cannot get worse than the worst case!

Analyzing Code

n = the size of "array"

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

this line runs n times

this line runs n times

this line never runs

this line runs 1 time

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

$T(n) =$ 2n + 1

Is *lines* the right unit? Maybe — Usually

It is often proportional to how much time the algorithm takes
to run given the input size.

Analyzing Code

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

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

- Does the “1” matter?

As n gets bigger, 2n dominates 1. So “no”

- Does the “2” matter?

The time per line changes by constant factor:
— As technology changes
— between different computers

Usually useful to ignore the constant factors
So, “no”

How can we abstract from
things that do not matter?
See next slide!

Big-O Notation

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

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

set of functions

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

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

Example:

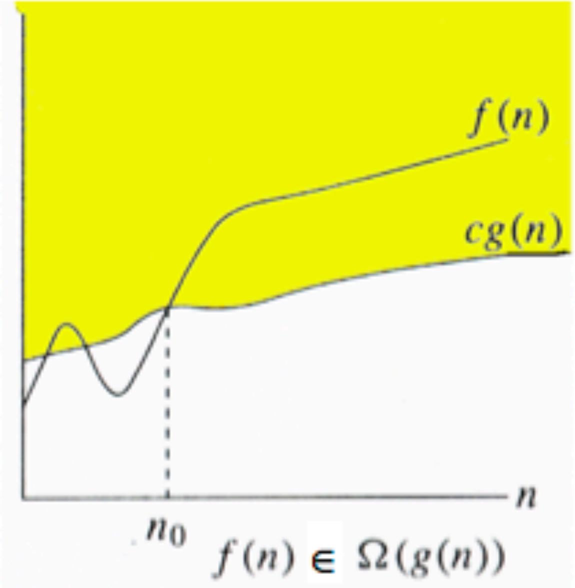
T(n) = 2n+1 f(n) = n

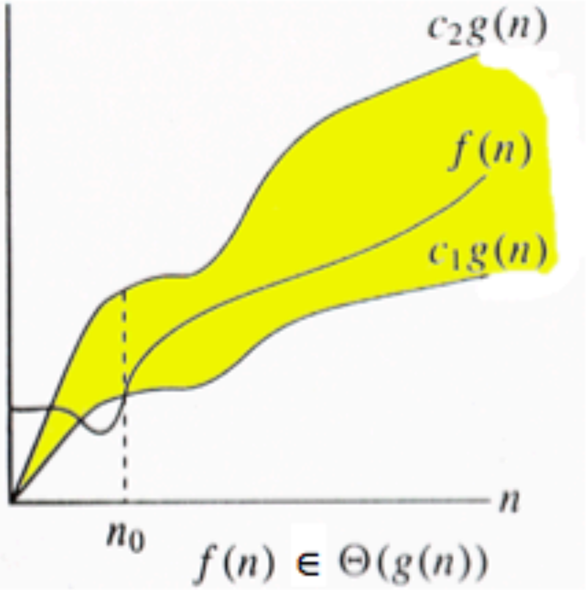
Claim: $2n+1 \in O(n)$

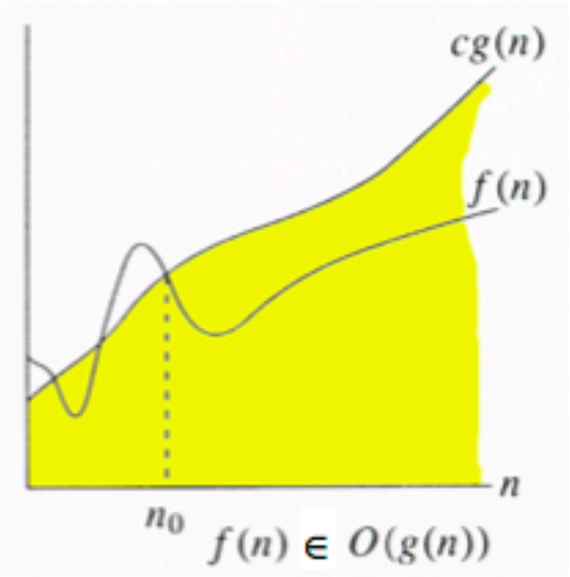
Proof:
$2n+1 \leq 3n$ for $n \geq 1$
$1 \leq n$ for $n \geq 1$ (subtract 2n from both sides)
which is true intuitively

So, we found some c and n0 (c=3 and n0=1) for which the above definition holds.

Thus, $2n+1 \in O(n)$ or $T(n) \in O(f(n))$







Images with minor changes from:
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Asymptotic Notation

Asymptotic notation helps us "compare" functions

- 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 $n \geq n_0$.

  i.e.: "T(n) grows slower or with the same rate as f(n)"

  T(n) "≤" f(n)

- 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 $n \geq n_0$.

  i.e.: "T(n) grows faster or with the same rate as f(n)"

  T(n) "≥" f(n)

  Exercise: show that
  $T(n) \in \Omega(f(n)) \iff f(n) \in O(T(n))$

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

  i.e.: "T(n) grows with the same rate as f(n)"

  T(n) "=" f(n)

Equivalent alternative definition for Θ:
$T(n) \in \Theta(f(n))$ iff there are positive constants c1, c2 and n0 such that $T(n) \geq c1\ f(n)$ and $T(n) \leq c2\ f(n)$ for all $n \geq n0$.

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 $n \geq n_0$.

  i.e.: "T(n) grows strictly slower than f(n)"

  T(n) "<" f(n)

  limit as n→∞, 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 $n \geq n_0$.

  i.e.: "T(n) grows strictly faster than f(n)"

  T(n) ">" f(n)

  limit as n→∞, T(n)/f(n) = ∞

Note that not all pairs of functions are related, for example:

n^(1+sin(n)) vs. n cannot be comparied

Examples

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

10,000 n^2 + 25 n ≤ 10,000 n^2 + 25 n^2 for n ≥ 1
≤ 10,025 n^2 for n ≥ 1

So c=10,025 and n0=1, and thus
10,000 n^2 + 25 n ∊ O(n^2)

10,000 n^2 + 25 n ≥ 10,000 n^2 for n ≥ 1

So c=10,000 and n0=1, and thus
10,000 n^2 + 25 n ∊ Ω(n^2)

Thus:
10,000 n^2 + 25 n ∊ Θ(n^2)

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

Use c=10^-10 and n0=1 for both O(n^2) and Ω(n^2) thus Θ(n^2)

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

n log(n) ≤ c n^2
log(n) ≤ c n
How do we know if this always hold as n grows?

lim as n→∞, log(n) / (c n)
consider their rate of growth (or derivatives)

lim as n→∞, (1/n) / (c) = 0

So c=1 and n0=1, n log(n) ∊ o(n^2) (i.e. even little o)

Thus from now on
you can always
assume that
log(n) ≤ c n
and
log(n) ∊ o(n)

Examples (cont.)

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

n log(n) ≥ c n
log(n) ≥ c

c=1 and n0=1

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

lim as n→∞, (n^3) / (n^4) = 0

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

lim as n→∞, (n^3) / (n^2) = ∞

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?)

$2n + 1 \in O(n)$ … saying the algorithm is fast
$2n + 1 \in \Omega(n)$ … saying the algorithm is slow

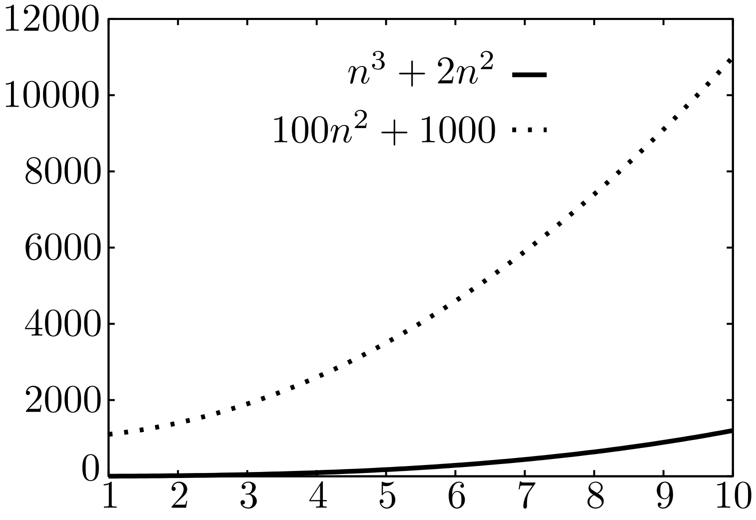
$2n + 1 \in \Theta(n)$

What we can say about best case:

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

Asymptotically Smaller?

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

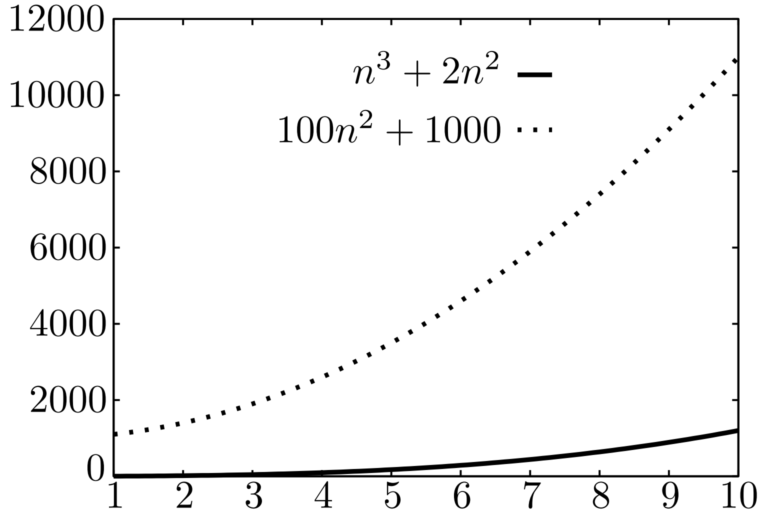


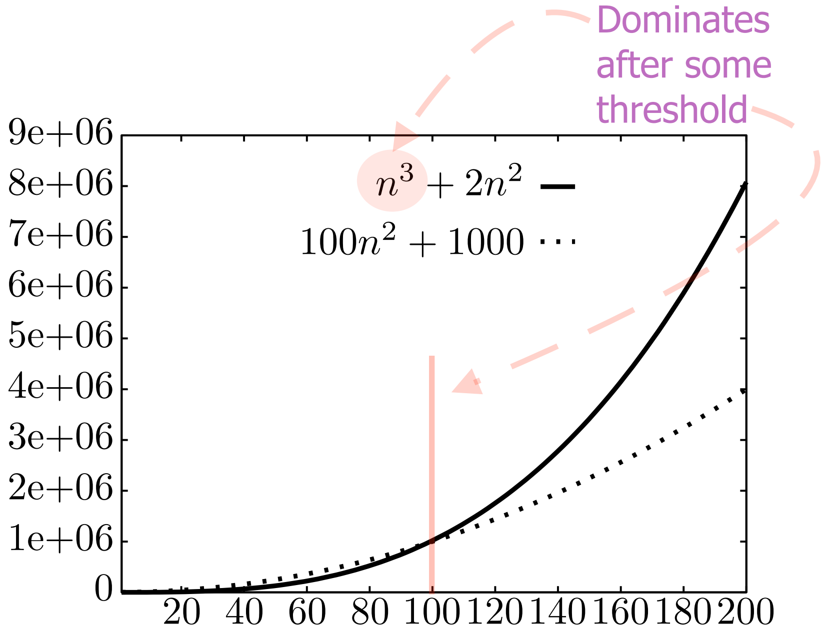
Asymptotically Smaller?

ignore lower terms: (2n^2)   and (1000)
ignore multiplicative constants: 100

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

n^3 + 2n^2 ∈ Ω(100n^2 + 1000)

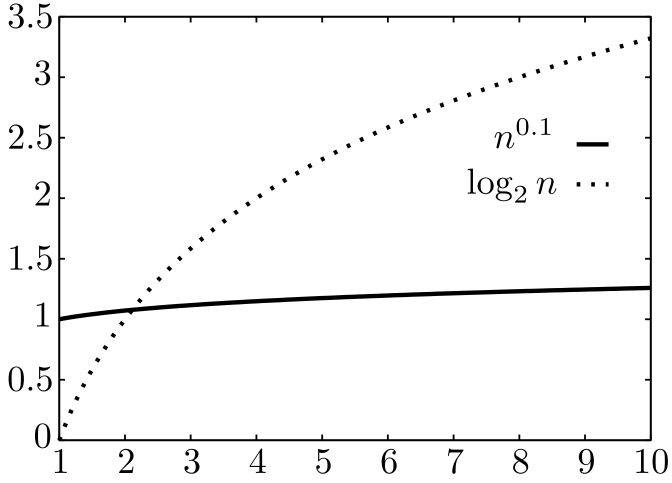



Dominates
after some
threshold

for c=1
and n0 about 100, say
n0=120

Asymptotically Smaller? (cont.)

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



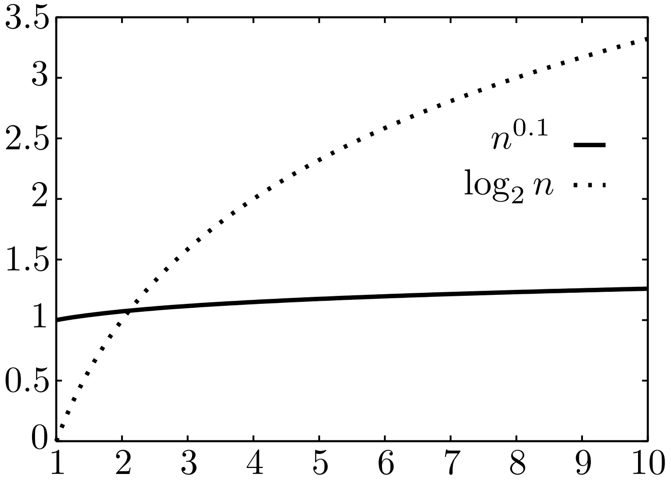
Asymptotically Smaller? (cont.)

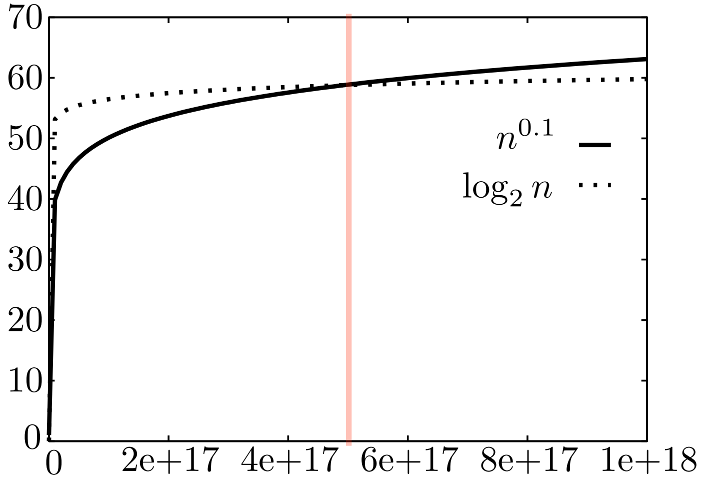
A B

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

A ∈ Ω(B)

also B ∈ O(A)





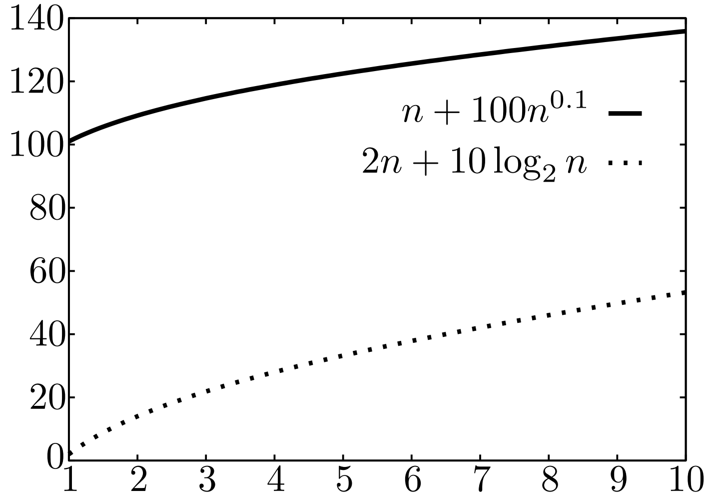
for c=1

and n0 about 5e+17
or anything higher, e.g.

n0=6e+17

Asymptotically Smaller? (cont.)

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



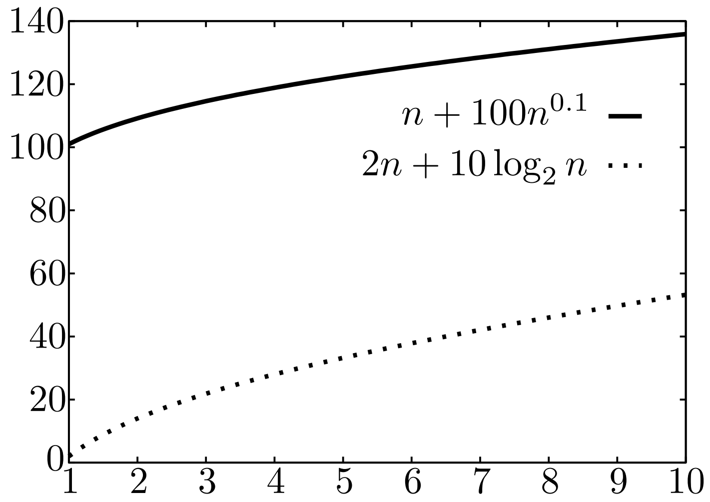
Asymptotically Smaller? (cont.)

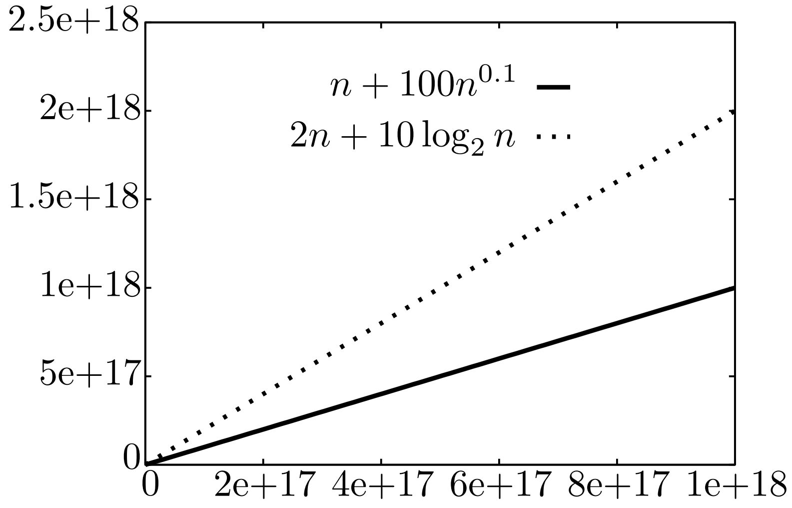
lower order terms can be ignored

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

A

B





Here, you can always come up with some constant c to make cA asymptotically larger than B, or to make cB asymptotically larger than A.

$A \in O(B)$

$A \in \Omega(B)$

thus

$A \in \Theta(B)$

Typical Asymptotics

$\log_b n = \log_a n / \log_a b$

Tractable

sublinear

- Constant: $\Theta(1)$
- Logarithmic: $\Theta(\log n)$ ($\log_b n$, $\log n^2 \in \Theta(\log n)$)
- 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)
- Quadratic: $\Theta(n^2)$
- Cubic: $\Theta(n^3)$
- Polynomial: $\Theta(n^k)$ (k is a constant)

the base of log does not matter

Intractable

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

Sample Asymptotic Relations

Example functions belonging to

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

Analyzing Code

e.g.
checking equality i<=j
or adding values   i+=1;

The sum also constant time
IF all the operations take
constant time

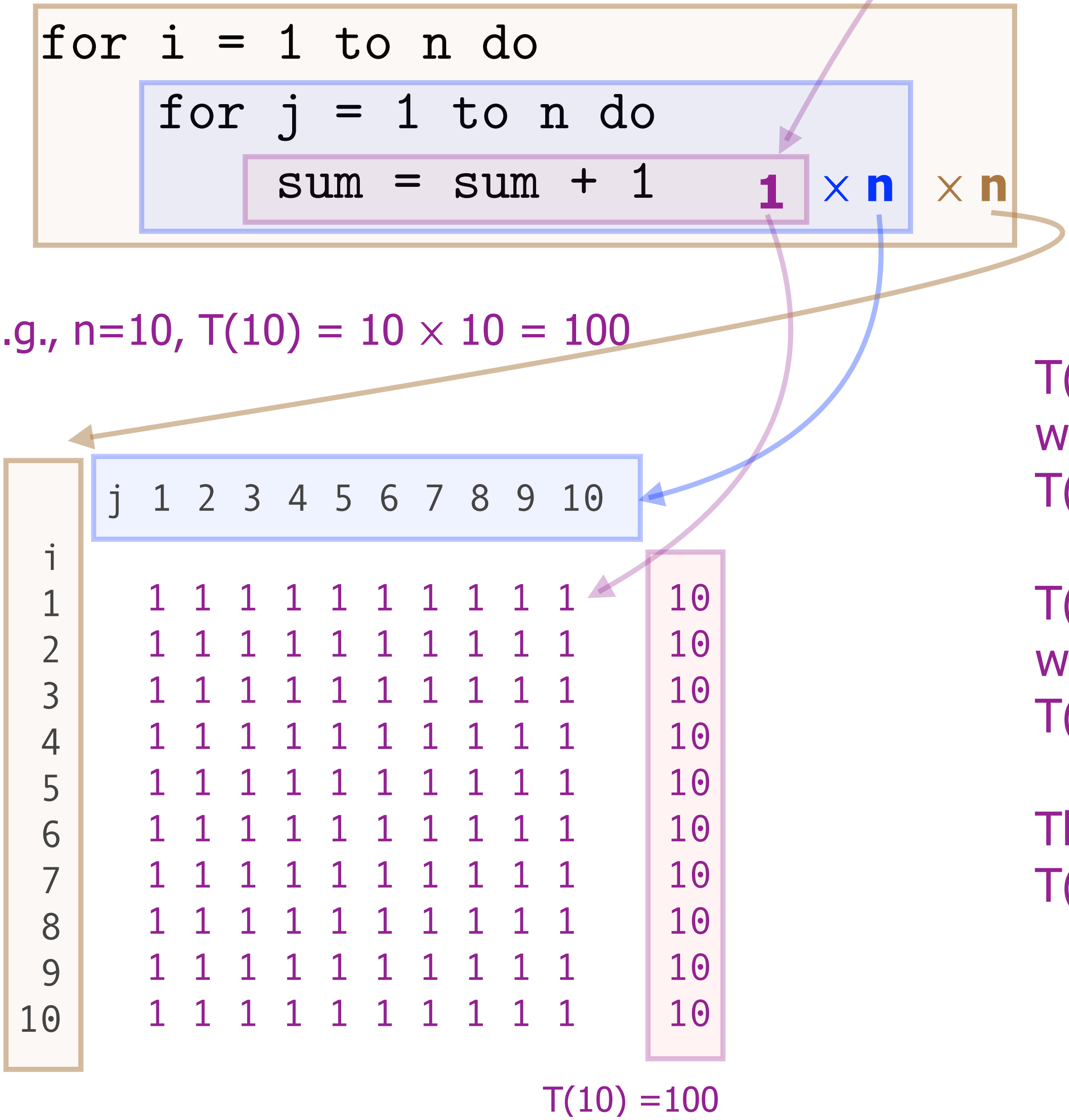
- Single operations: constant time
- Consecutive operations: sum of the operations’ times
- Conditionals: condition time plus the maximum (for worst-case analysis) of the branch times
- Loops: sum of the loop body times   (loop body time × #executed)
- Function call: time for the function

Above all, use common sense!

Runtime Example #1

n is size of input array and T(n) is time to run

The line "sum = sum + 1" takes some constant amount of time to run. The number of times it is executed is proportional to total of time needed.



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

e.g., n=10, T(10) = 10 × 10 = 100

j	1	2	3	4	5	6	7	8	9	10	
i											
1	1	1	1	1	1	1	1	1	1	1	10
2	1	1	1	1	1	1	1	1	1	1	10
3	1	1	1	1	1	1	1	1	1	1	10
4	1	1	1	1	1	1	1	1	1	1	10
5	1	1	1	1	1	1	1	1	1	1	10
6	1	1	1	1	1	1	1	1	1	1	10
7	1	1	1	1	1	1	1	1	1	1	10
8	1	1	1	1	1	1	1	1	1	1	10
9	1	1	1	1	1	1	1	1	1	1	10
10	1	1	1	1	1	1	1	1	1	1	10

T(10) =100

So,
T(n) = 1 × n × n = n^2

T(n) = n^2, so also T(n) ≤ n^2
which means:
T(n) ∈ Ω(n^2)

T(n) = n^2, so also T(n) ≥ n^2
which means:
T(n) ∈ O(n^2)

Thus:
T(n) ∈ Θ(n^2)

Runtime Example #2

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

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

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

e.g., with n=10

i \ j	1	2	3	4	5	6	7	8	9	10	
1	1	1	1	1	1	1	1	1	1	1	10
2		1	1	1	1	1	1	1	1	1	9
3			1	1	1	1	1	1	1	1	8
4				1	1	1	1	1	1	1	7
5					1	1	1	1	1	1	6
6						1	1	1	1	1	5
7							1	1	1	1	4
8								1	1	1	3
9									1	1	2
10										1	1

T(10) = 55

$$T(n) = n + (n-1) + \dots + 3 + 2$$

reverse order

$$T(n) = 2 + 3 + \dots + (n-1) + n$$

add

$$T(n)+T(n) = \underbrace{(n+2) + (n+2) + \dots + (n-2) + (n-2)}_{(n-1)\text{ times}}$$

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

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

$$T(n) = n^2/2 + n/2 - 1$$

$$T(n) \leq n^2/2 + n^2/2 - 1$$

$$T(n) \leq n^2$$

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

$$T(n) = n^2/2 + n/2 - 1$$

$$T(n) \geq n^2/4 \text{ for } n > 1$$

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

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

Runtime Example #3

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

e.g., with n=10, i=1, 2, 4, 8

i	j 1	2	3	4	5	6	7	8	9	10	
1	1	1	1	1	1	1	1	1	1	1	10
2		1	1	1	1	1	1	1	1	1	9
3											
4				1	1	1	1	1	1	1	7
5											
6											
7											
8								1	1	1	3
9											
10											

T(10) = 29

i = 1, 2, 4, 8, 16, …

i = 2^0, 2^1, 2^2, 2^3, … , 2^k

such that 2^k < n ≤ 2^(k+1)

T(n) = ∑ i

T(n) = 2^k + … 2^2 + 2^1 + 2^0 — in reversed order

T(n) = $(1\,1\,1 \dots 1\,1\,1)_2$ — number represented in base 2

T(n) = $(0\,1\,1\,1 \dots 1\,1\,1)_2$ — add a leading 0 (no effect)

T(n)+1 = $(1\,0\,0\,0 \dots 0\,0\,0)_2$ — add 1 to both sides

T(n) = $(1\,0\,0\,0 \dots 0\,0\,0)_2$ - 1

T(n) = 2^(k+1) - 1 — because digit 1 is a bit at position k+1

T(n) ≥ n - 1 — we had above that n ≤ 2^(k+1)

T(n) ∈ Ω(n)

T(n) = 2^(k+1) - 1

T(n) = 2 × 2^k - 1

T(n) < 2 × n - 1 — we had above that 2^k < n

T(n) ∈ O(n)

T(n) ∈ Θ(n)

Runtime Example #4

b = 1 line

c = 2 lines

Usually these exact values for constants b and c do not matter because they will be ignored for asymptotic relations.

base case

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

n

n-1

Recursion almost always yields a recurrence relation:

base case

$$T(1) \le b$$

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

Solving the recurrence:

because
T(n – 1) ≤ c + T(n – 2)

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

T(n – 2) ≤ c + T(n – 3)

means "guessing"

base case

Set k such that:
n – k = 1
to obtain T(1)

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

Runtime Example #5: Mergesort

Takes constant time, so ignored

Mergesort algorithm:

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

T(n/2) T(n/2) c n for some constant c

Recurrence relation:

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

Solving recurrence:

$$T(n) \le 2\underline{T(n/2)} + cn$$

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

$$\le 2(2T(n/4) + cn/2) + cn \qquad \text{(substitution)}$$

$$= 4\underline{T(n/4)} + 2cn$$

$T(n/4) \le 2T(n/8) + cn/4$

$$\le 4(2T(n/8) + cn/4) + 2cn \qquad \text{(substitution)}$$

$$= 8T(n/8) + 3cn$$

$$\le 2^k T(n/2^k) + kcn \qquad \text{(extrapolating } k > 0\text{)}$$

$$= nT(1) + cn \lg n \qquad n/2^k = 1 \quad (\text{for } 2^k = n) \quad k = \lg n$$

$\le nb + cn \lg n$

$T(n) \in$ 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)

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

Claim:

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

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

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

Claim:

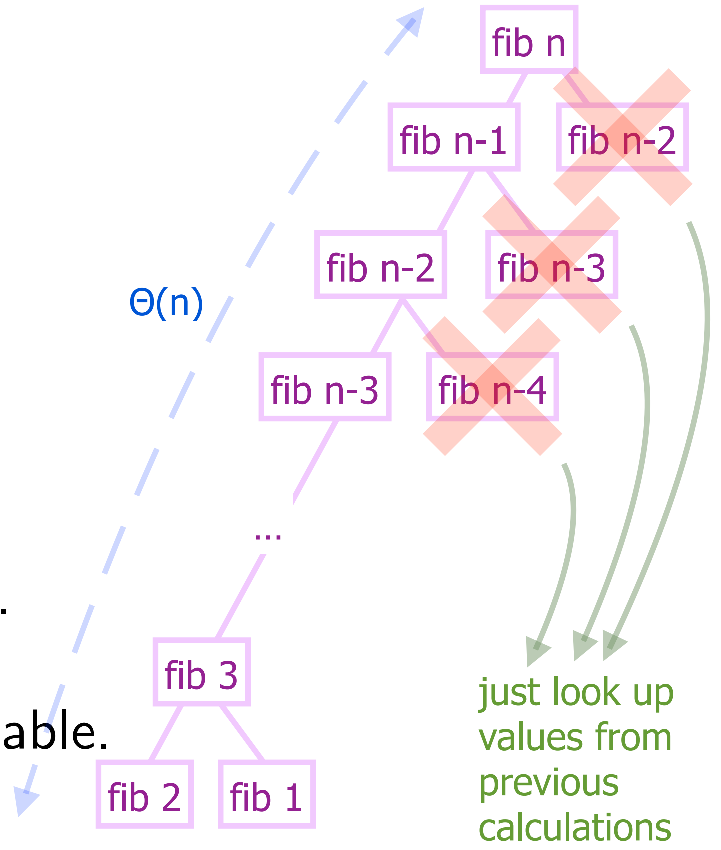
$$T(n) \geq b\varphi^{n-1}$$

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 $n = k + 1$.

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

$T(n) \in$
Why? The same recursive call is made numerous times.

Example #7: Learning from Analysis



To avoid recursive calls:

As each value is computed, store them in an array.

- Store base case values in a table.
- Before calculating the value for *n*:
  - Check if the value for *n* is in the table. Θ(1)
  - If so, return it.
  - If not, calculate it and store it in the table.

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

How much time does this version take?

Θ(n) because we compute all values from 1 to n only 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: | A | = n | B | = m

$A = \underline{\texttt{s}}\texttt{e}\underline{\texttt{a}}\texttt{rch}\ \underline{\texttt{me}}$ $B = \texttt{in}\underline{\texttt{sa}}\texttt{ne}\ \underline{\texttt{me}}\texttt{thod}$

101000 11 001100 110000

A longest common subsequence is “$\underline{\texttt{same}}$”; another is “`seme`”.

(ignoring spaces)

Applications of LCS:

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

Example: subsequences of “abc”

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

Runtime Example #8: LCS (cont.)

$|A| = n$ $|B| = m$

An Algorithm and Its Analysis:

Algorithm 1:

```
For every subsequence S if A
    For every subsequence S' if B
        If S=S'
            Remember the longest so far
```

$\Theta(\min(n,m))$ $\times 2^m$ $\times 2^n$

$T_1(n,m) = \Theta(2^m 2^n \min(n,m))$

Algorithm 2:

```
For every subsequence S if A
    If S is subsequence of B
        Remember the longest so far
```

$\Theta(m)$ $\times 2^n$

$T_2(n,m) = \Theta(2^n \min(n,m))$

Greedy approach:

```
Find first occurrence of S[0] in B
For i=1 to length(S)-1
    Find first occurrence of S[i] in B
    after occurrence of S[i-1]
```

$\Theta(m)$

Example #9

recall: lg a · b = lg a + lg b

Find a $\underline{\overset{\text{meaning } \Theta}{\text{tight bound}}}$ on $T(n) = \lg(n!)$.

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

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

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

Every term lg(i) ≤ lg(n)

$$\begin{aligned} T(n) = \sum_{i=1}^{n} \lg(i) \ge \sum_{i=n/2}^{n} \lg(i) \ge \sum_{i=n/2}^{n} \lg(n/2) &= n/2 \lg(n/2) \\ &= n/2 \lg(n \cdot 1/2) \\ &= n/2 (\lg(n) + \lg(1/2)) \\ &= n/2 (\lg(n) - 1) \\ &= n/2 \lg(n) - n/2 \\ &= n/4 \lg(n) + n/4 \lg(n) - n/2 \\ &\ge n/4 \lg(n) \quad \text{for } n \ge 4 \\ &= 1/4\, n \lg(n) \quad \text{for } n \ge 4 \\ &\in \Omega(n \lg(n)) \quad \text{(i.e., } c=1/4 \text{ and } n0=3) \end{aligned}$$

Removing first half of the terms

Every term lg(i) ≥ lg(n/2)

As long as
$n/4 \lg(n) \ge n/2$
$\lg(n) \ge 2$
$n \ge 4$

$T(n) \in O(n \lg(n))$
$T(n) \in \Omega(n \lg(n))$
So
$T(n) \in \Theta(n \lg(n))$

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 $b, c > 1$.

Asymptotic Analysis Summary

- 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
- 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 $\overset{\Omega}{\text{lower}}$ 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 $\overset{O}{\text{upper}}$ bound on the problem's complexity.

Example 1:

Searching an unsorted list using comparisons takes $\Omega(n)$ time (lower bound).

- Linear search takes $O(n)$ time (matching upper bound).

Example 2:

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 $\overset{\Theta}{O}(n \log n)$ Sort?

Chew these over:

- How many values can you represent with c bits? $z=2^c$ values
  In other words: $c = \lg z$
- Comparing two values ($x < y$) gives you one bit of information.
- There are $n!$ possible ways to reorder a list. We could number them: $1, 2, \ldots, n!$ How many bits do we need to number them all: $\lg(n!)$
- 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?

  How many bits do we need to represent a number to choose between n! possibilities?
  $\geq \lg(n!)$
  $\in \Omega(n \lg n)$ see slide 37

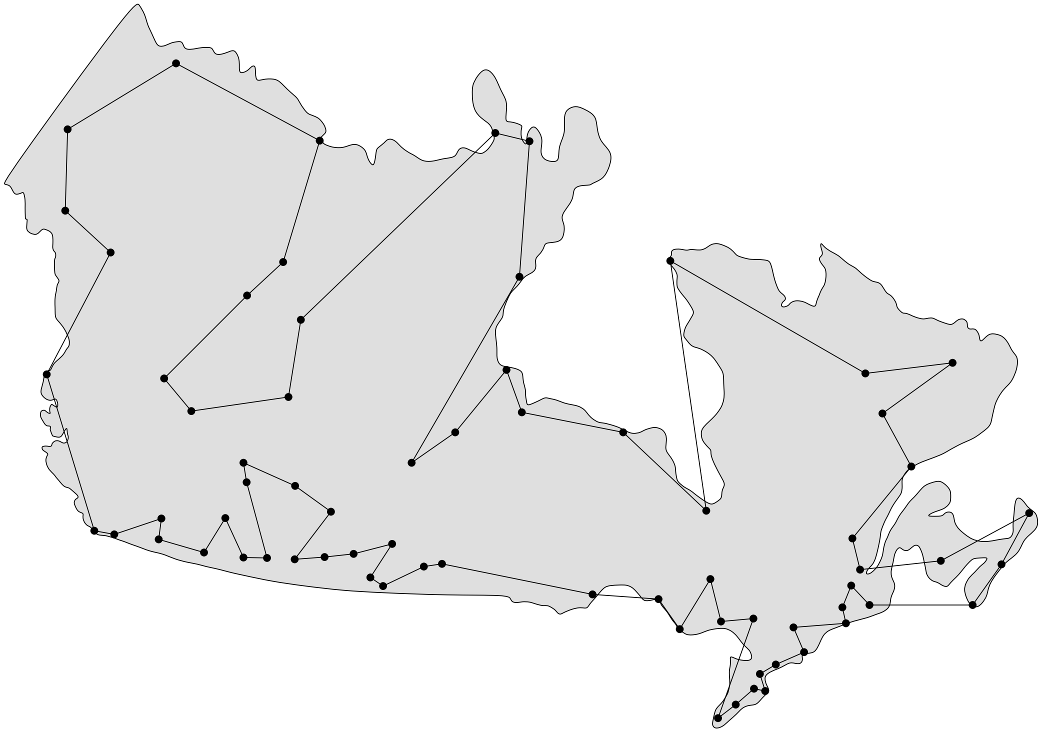
Problem Complexity

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.