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
  - 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.

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

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

We'll prove the stronger result:

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

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:

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

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):

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

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.
- Consider the operation.
- Consider the possible data structures.
- Does it matter which data structure we use?

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.

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

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

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		

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

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

1) What's the input size n?

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 scenario for running an input of size n?

Analyzing Code

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

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

$T(n) =$

Is *lines* the right unit?

Analyzing Code

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

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

- Does the "1" matter?
- Does the "2" matter?

Big-O Notation

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

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

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

Asymptotic Notation

- 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 $n \geq n_0$.

- 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 $n \geq n_0$.

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

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 $n \geq n_0$.

- 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 $n \geq n_0$.

Examples

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

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

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

Examples (cont.)

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

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

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

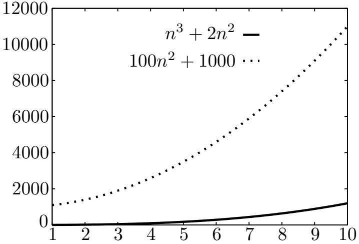
Analyzing Code

```
// 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?)

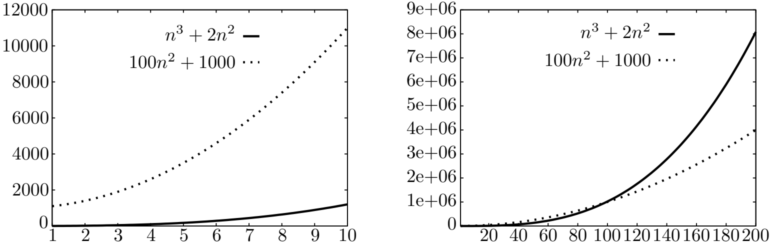
Asymptotically Smaller?

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



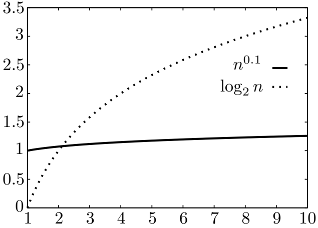
Asymptotically Smaller?

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



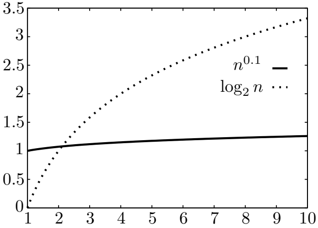
Asymptotically Smaller? (cont.)

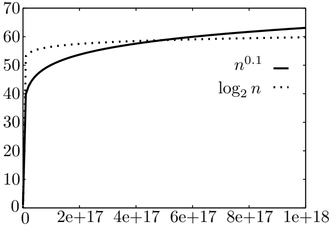
$n^{0.1}$ versus $\log_2 n$



Asymptotically Smaller? (cont.)

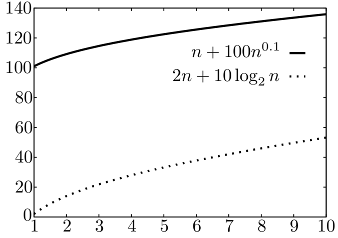
$n^{0.1}$ versus $\log_2 n$





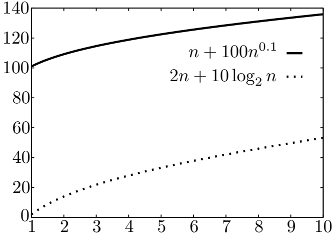
Asymptotically Smaller? (cont.)

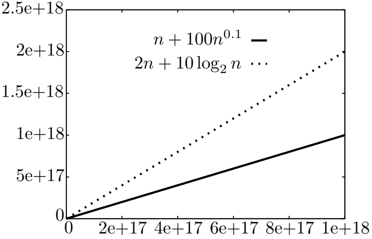
$$n + 100n^{0.1} \qquad \text{versus} \qquad 2n + 10\log_2 n$$



Asymptotically Smaller? (cont.)

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





Typical Asymptotics

Tractable

- 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)

Intractable

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

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

- 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
- Function call: time for the function

Above all, use common sense!

Runtime Example #1

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

Runtime Example #2

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

Runtime Example #3

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

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:

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

Solving the recurrence:

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

$T(n) \in$

Runtime Example #5: Mergesort

Mergesort algorithm:

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

Recurrence relation:

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

Solving recurrence:

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

$T(n) \in$

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:

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

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:

- 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 *memoization* and is closely related to *dynamic programming*.

How much time does this version take?

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:

$A = \texttt{search me}$ $B = \texttt{insane method}$

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

Applications of LCS:
DNA sequencing, revision control systems, `diff`, ...

Runtime Example #8: LCS (cont.)

An Algorithm and Its Analysis:

Example #9

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

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 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:

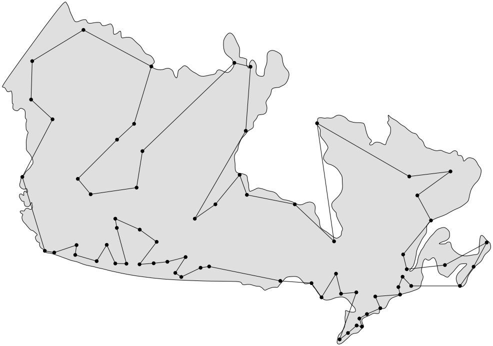
- How many values can you represent with c bits?
- Comparing two values $(x < y)$ gives you one bit of information.
- There are $n!$ possible ways to reorder a list. We could number them: $1, 2, \ldots, n!$
- Sorting basically means choosing which of those reorderings/numbers you'll apply to your input.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

Problem Complexity

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.



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.