CS 240 – Data Structures and Data Management

Module 1: Introduction and Asymptotic Analysis

T. Biedl  É. Schost  O. Veksler
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2021

References: Goodrich & Tamassia 1.1, 1.2, 1.3
Sedgewick 8.2, 8.3
Outline

1. Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
   - Analysis of Algorithms I
   - Asymptotic Notation
   - Analysis of Algorithms II
   - Example: Analysis of MergeSort
   - Helpful Formulas
Outline

1. Introduction and Asymptotic Analysis
   - CS240 Overview
     - Algorithm Design
     - Analysis of Algorithms I
     - Asymptotic Notation
     - Analysis of Algorithms II
     - Example: Analysis of MergeSort
     - Helpful Formulas
Course Objectives: What is this course about?

- When first learning to program, we emphasize *correctness*: does your program output the expected results?

- Starting with this course, we will also be concerned with *efficiency*: is your program using the computer’s resources (typically processor time) efficiently?

- We will study efficient methods of *storing*, *accessing*, and *organizing* large collections of data.

- Typical operations include: *inserting* new data items, *deleting* data items, *searching* for specific data items, *sorting*.

- **Motivating examples**: Digital Music Collection, English Dictionary
Course Objectives: What is this course about?

- We will consider various **abstract data types** (ADTs) and how to implement them efficiently using appropriate **data structures**.
- There is a strong emphasis on mathematical analysis in the course.
- Algorithms are presented using pseudo-code and analyzed using order notation (big-Oh, etc.).
Course Topics

- big-Oh analysis ✓
- priority queues and heaps ✓
- sorting, selection ✓
- binary search trees, AVL trees, B-trees ✓
- skip lists ✓
- hashing
- quadtrees, kd-trees ✓
- range search ✓
- tries ✓
- string matching
- data compression
CS Background

Topics covered in previous courses with relevant sections in [Sedgewick]:

- arrays, linked lists (Sec. 3.2–3.4)
- strings (Sec. 3.6)
- stacks, queues (Sec. 4.2–4.6)
- abstract data types (Sec. 4-intro, 4.1, 4.8–4.9)
- recursive algorithms (5.1)
- binary trees (5.4–5.7)
- sorting (6.1–6.4)
- binary search (12.4)
- binary search trees (12.5)
- probability and expectations (Goodrich & Tamassia, Section 1.3.4)
Outline

1. Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
     - Analysis of Algorithms I
     - Asymptotic Notation
     - Analysis of Algorithms II
     - Example: Analysis of MergeSort
     - Helpful Formulas
Problems (terminology)

First, we must introduce terminology so that we can precisely characterize what we mean by efficiency.

**Problem:** Given a problem instance, carry out a particular computational task.

**Problem Instance:** *Input* for the specified problem.

**Problem Solution:** *Output* (correct answer) for the specified problem instance.

**Size of a problem instance:** $\text{Size}(I)$ is a positive integer which is a measure of the size of the instance $I$.

**Example:** Sorting problem

- **input:** an array $A$ of integers
- **output:** an array with the same integers in increasing order
- **size of an input:** the length of $A$
Algorithm: An algorithm is a *step-by-step process* (e.g., described in pseudo-code) for carrying out a series of computations, given an arbitrary problem instance $I$.

Solving a problem: An Algorithm $A$ *solves* a problem $\Pi$ if, for every instance $I$ of $\Pi$, $A$ finds (computes) a valid solution for the instance $I$ in finite time.

Program: A program is an *implementation* of an algorithm using a specified computer language.

In this course, our emphasis is on algorithms (as opposed to programs or programming).
 Algorithms and Programs

**Pseudo-code**: a method of communicating an algorithm to another person.

In contrast, a program is a method of communicating an algorithm to a computer.

Pseudo-code

- omits obvious details, e.g. variable declarations,
- has limited if any error detection,
- sometimes uses English descriptions,
- sometimes uses mathematical notation.
Algorithms and Programs

For a problem $\Pi$, we can have several algorithms.

For an algorithm $\mathcal{A}$ solving $\Pi$, we can have several programs (implementations).

Algorithms in practice: Given a problem $\Pi$

1. Design an algorithm $\mathcal{A}$ that solves $\Pi$. $\rightarrow$ Algorithm Design
2. Assess correctness and efficiency of $\mathcal{A}$. $\rightarrow$ Algorithm Analysis
3. If acceptable (correct and efficient), implement $\mathcal{A}$.
1 Introduction and Asymptotic Analysis
   • CS240 Overview
   • Algorithm Design
   • Analysis of Algorithms I
     • Asymptotic Notation
     • Analysis of Algorithms II
     • Example: Analysis of MergeSort
     • Helpful Formulas
Efficiency of Algorithms/Programs

- How do we decide which algorithm or program is the most efficient solution to a given problem?
- In this course, we are primarily concerned with the amount of time a program takes to run. \[\rightarrow\text{Running Time}\]
- We also may be interested in the amount of additional memory the program requires. \[\rightarrow\text{Auxiliary space}\]
- The amount of time and/or memory required by a program will depend on \textit{Size}(I), the size of the given problem instance \textit{I}.
Running Time of Algorithms/Programs

First option: *experimental studies*

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like `clock()` (from `time.h`) to get an accurate measure of the actual running time.
- Plot/compare the results.
Running Time of Algorithms/Programs

Shortcomings of experimental studies

- Implementation may be complicated/costly.
- Timings are affected by many factors: \textit{hardware} (processor, memory), \textit{software environment} (OS, compiler, programming language), and \textit{human factors} (programmer).
- We cannot test all inputs; what are good \textit{sample inputs}?

We want a framework that:

- Does not require implementing the algorithm.
- Is independent of the hardware/software environment.
- Takes into account all input instances.

We need some \textit{simplifications}.
Overview of Algorithm Analysis

We will develop several aspects of algorithm analysis in the next slides. To overcome dependency on hardware/software:

- Algorithms are presented in structured high-level *pseudo-code* which is language-independent.
- Analysis of algorithms is based on an *idealized computer model*.
- Instead of time, count the number of *primitive operations*.
- The efficiency of an algorithm (with respect to time) is measured in terms of its *growth rate*. 
Random Access Machine

Random Access Machine (RAM) model:

- A set of memory cells, each of which stores one item (word) of data. Implicit assumption: memory cells are big enough to hold the items that we store.
- Any *access to a memory location* takes constant time.
- Any *primitive operation* takes constant time. Implicit assumption: primitive operations have fairly similar, though different, running time on different systems.
- The *running time* of a program is proportional to the number of memory accesses plus the number of primitive operations.

This is an idealized model, so these assumptions may not be valid for a “real” computer.
Running Time Simplifications

We will simplify our analysis by considering the behaviour of algorithms for large inputs sizes.

- **Example 1**: What is larger, $100n$ or $10n^2$?
- **Example 2**: What is larger, $1000000n + 20000000000000000$ or $0.01n^2$?

- To simplify comparisons, use **order notation**
- Informally: ignore constants and lower order terms
Outline

1 Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
   - Analysis of Algorithms I
   - Asymptotic Notation
   - Analysis of Algorithms II
   - Example: Analysis of MergeSort
   - Helpful Formulas
Order Notation

**O-notation**: \( f(n) \in O(g(n)) \) if there exist constants \( c > 0 \) and \( n_0 > 0 \) such that \( |f(n)| \leq c |g(n)| \) for all \( n \geq n_0 \).

Example: \( f(n) = 75n + 500 \) and \( g(n) = 5n^2 \) (e.g. \( c = 1, n_0 = 20 \)).

**Note**: The absolute value signs in the definition are irrelevant for analysis of run-time or space, but are useful in other applications of asymptotic notation.
∀n, |f(n)| ≤ |g(n)|
⇒ f(n) ∈ O(g(n))

c=1, \quad n_0 = 1

∀n, |f(n)| ≤ 2|g(n)|
⇒ f(n) ∈ O(g(n))

c=2, \quad n_0 = 1
\[ f(n) = 75n + 500 \quad \text{and} \quad g(n) = 5n^2 \]

for \( n \geq 20 \)

\[ 20 \leq n \leq 5n \]

\[ 100n \leq 5n^2 \quad (\text{iv}) \]

for \( n \geq 20 \)

\[ 20 \leq n \leq 25 \]

\[ 500 \leq 25n^2 \quad (\text{v}) \]

\[ 75n + 500 \leq 100n \quad (\text{vi}) \]

\[ \Rightarrow \text{for } n \geq 20, \quad |f(n)| = 75n + 500 \leq 5n^2 = |g(n)| \]

Taking \( n_0 = 20 \) and \( c = 1 \), this proves that \( f(n) \in O(g(n)) \).
Example of Order Notation

In order to prove that $2n^2 + 3n + 11 \in O(n^2)$ from first principles, we need to find $c$ and $n_0$ such that the following condition is satisfied:

$$0 \leq 2n^2 + 3n + 11 \leq c \cdot n^2 \text{ for all } n \geq n_0.$$ 

note that not all choices of $c$ and $n_0$ will work.
\[ f(n) = 2n^2 + 3n + 11 \quad \quad g(n) = n^2 \]

\[ n > 1 \]

\[ 2n^2 \leq 2n^2 \]

\[ 3n \leq 3n^2 \]

\[ 11 \leq 11n^2 \]

\[ f(n) + 16n^2 = 16g(n) \]

Taking \( n_0 = 1 \) and \( c = 16 \), this proves that \( f(n) \in O(g(n)) \).
Asymptotic Lower Bound

- We have $2n^2 + 3n + 11 \in O(n^2)$
- But we also have $2n^2 + 3n + 11 \in O(n^{10})$
- We want a tight asymptotic bound.

**Ω-notation:** $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $c |g(n)| \leq |f(n)|$ for all $n \geq n_0$.

**Θ-notation:** $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ for all $n \geq n_0$.

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$
Example of Order Notation

Prove that \( f(n) = 2n^2 + 3n + 11 \in \Omega(n^2) \) from first principles.

\[
\forall n \geq 1 \quad 2n^2 + 3n + 11 \geq n^2 \quad \text{So taking } \ a_0 = 1, \ C = 1
\]

This proves that \( f(n) \in \Omega(n^2) \)

Prove that \( \frac{1}{2}n^2 - 5n \in \Omega(n^2) \) from first principles.

Prove that \( \log_b(n) \in \Theta(\log n) \) for all \( b > 1 \) from first principles.

\[
\log_b(n) = \frac{\log(n)}{\log(b)} \quad \text{So taking } \ a_0 = 1 \text{ and } \ C_1 = C_2 = \frac{1}{\log(b)}
\]

This proves that \( \log_b(n) \in \Theta(\log n) \)
\[ f(n) = \frac{1}{2} n^2 - 5n \quad g(n) = n^2 \]

For \( n > 20 \), \( n^2 > 20n \) and \( \frac{1}{4} n^2 \geq 5n \),

\[
\frac{1}{4} n^2 \geq 5n \Rightarrow -5n \geq \left(\frac{1}{4} - 1\right)n^2 + \frac{1}{2} n^2
\]

\[ f(n) \geq \frac{1}{2} n^2 = \frac{1}{2} g(n). \]

Taking \( n_0 = 20 \) and \( c = \frac{1}{2} \), one proves that \( f(n) \in \mathcal{O}(g(n)) \).
Strictly smaller/larger asymptotic bounds

- We have \( f(n) = 2n^2 + 3n + 11 \in \Theta(n^2) \).
- How to express that \( f(n) \) is asymptotically strictly smaller than \( n^3 \)?

\textbf{\( o \)-notation:} \( f(n) \in o(g(n)) \) if for all constants \( c > 0 \), there exists a constant \( n_0 > 0 \) such that \( |f(n)| \leq c |g(n)| \) for all \( n \geq n_0 \).

\textbf{\( \omega \)-notation:} \( f(n) \in \omega(g(n)) \) if \( g(n) \in o(f(n)) \).

- Rarely proved from first principles.

\[ \begin{align*}
\Rightarrow & \quad \forall c > 0 \exists n_0 > 0 \text{ such that} \\
& \forall n > n_0 \quad |g(n)| \leq c |f(n)|
\end{align*} \]
\( f(n) = 2000 n^2 \quad g(n) = n^n \)

Let \( c > 0 \) we need to find \( n_0 \) such that

\[ 2000 n^2 \leq c n^n \quad (x) \]

\( (x) \) is equivalent to \( 2000 \leq c n^{n-2} \)

For \( n > 3 \), \( n \leq n^{n-2} \)

For \( n > 3 \) and \( n \geq \frac{2000}{c} \), \( \frac{2000}{c} \leq n \)

\[ 2000 \leq c n \Rightarrow 2000 \leq c n^{n-2} \]

Taking \( n_0 = \max(3, \frac{2000}{c}) \), we see that \( f(n) \leq o(g(n)) \).
Algebra of Order Notations

**Identity rule:** \( f(n) \in \Theta(f(n)) \)

**Transitivity:**
- If \( f(n) \in O(g(n)) \) and \( g(n) \in O(h(n)) \) then \( f(n) \in O(h(n)) \)
- If \( f(n) \in \Omega(g(n)) \) and \( g(n) \in \Omega(h(n)) \) then \( f(n) \in \Omega(h(n)) \).

**Maximum rules:** Suppose that \( f(n) > 0 \) and \( g(n) > 0 \) for all \( n \geq n_0 \). Then:
- \( O(f(n) + g(n)) = O(\max\{f(n), g(n)\}) \)
- \( \Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\}) \)
Suppose that \( f(n) \in O(g(n)) \) \((\star)\)
\[ f(n) \in O(h(n)) \] \((\star\star)\)

\((\star)\) \(\iff\) \( \exists \ c, n_0 \text{ s.t. } \forall n \geq n_0 \) \( |f(n)| \leq c |g(n)| \)

\((\star\star)\) \(\iff\) \( \exists c', n_0', \text{ s.t. } \forall n \geq n_0' \) \( |g(n)| \leq c' |h(n)| \)

\( n_0'' = \max (n_0, n_0') \). Then, for \( n \geq n_0'' \)
\[ |f(n)| \leq c |g(n)| \leq c c' |h(n)| \]

This proves that \( f(n) \in O(g(n)) \).
\((\ast)\)

\(a(n) \in O(\max(f(n), g(n))) \iff a(n) \in O(f(n) + g(n)).\)

1. Suppose \((\ast)\). \(\exists n_1, c\) such that \(\forall n \geq n_1, |a(n)| \leq c \max(f(n), g(n))\).

2. \(\forall n > \max(n_0, n_1)\)

\[|a(n)| \leq c \max(f(n), g(n))\]

because \(f(n)\) and \(g(n) > 0\), \(\max(f(n), g(n)) \leq f(n) + g(n)\).

[\(\max(\ldots)\) is either \(f(n)\) or \(g(n)\). Suppose it is \(f(n)\).]

Then \(f(n) \leq f(n) + g(n)\) because \(g(n) > 0\).]

\[\Rightarrow |a(n)| \leq c (f(n) + g(n)) = c \ max(f(n) + g(n)).\]
(iv) 
\[ a(n) \in O\left( \max(f(n), g(n)) \right) \iff a(n) \in O(f(n) + g(n)). \]

Suppose (iv). Fix \( n_0 \) such that for \( n > n_0 \),

\[ |a(n)| \leq c |f(n) + g(n)|. \]

For \( n > \max(n_0, n_1) \), \( |a(n)| \leq c (f(n) + g(n)) \)

\[ f(n) + g(n) \leq \max(f(n), g(n)) + \max(f(n), g(n)) \]

\[ = 2 \max(f(n), g(n)) \]

\[ \leq 2 |\max(f(n), g(n))|. \]

\[ |a(n)| \leq 2c |\max(f(n), g(n))|. \]
Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

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

(in particular, the limit exists).

Then

$$f(n) \in \begin{cases} 
  o(g(n)) & \text{if } L = 0 \quad \checkmark \\
  \Theta(g(n)) & \text{if } 0 < L < \infty \quad \checkmark \\
  \omega(g(n)) & \text{if } L = \infty. \quad \checkmark
\end{cases}$$

The required limit can often be computed using \textit{l'\'Hôpital's rule}. Note that this result gives \textit{sufficient} (but not necessary) conditions for the stated conclusions to hold.
Example 1

Let $f(n)$ be a polynomial of degree $d \geq 0$:

$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

for some $c_d > 0$.

Then $f(n) \in \Theta(n^d)$:

1) $f(n) = n^d \left( c_d + \frac{c_{d-1}}{n} + \cdots + \frac{c_1}{n^{d-1}} + \frac{c_0}{n^d} \right)$

\[ \rightarrow 0 \text{ when } n \rightarrow \infty \]

\[ \rightarrow c_d \text{ when } n \rightarrow \infty. \text{ In particular,} \]

$f(n) > 0$ for $n$ large enough.
Example 1

Let \( f(n) \) be a polynomial of degree \( d \geq 0 \):

\[
f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0
\]

for some \( c_d > 0 \).

Then \( f(n) \in \Theta(n^d) \):

\[
2) \quad \frac{f(n)}{n^d} = \frac{c_d}{n^d} + \frac{c_{d-1}}{n^{d-1}} + \cdots + \frac{c_1}{n} + \frac{c_0}{n^d}.
\]

The limit of \( \frac{f(n)}{n^d} \) when \( n \to \infty \) exists and is \( \leq \infty \).

By the limit rule, we get \( f(n) \in \Theta(n^d) \).
Example 2

Prove that \( n(2 + \sin n\pi/2) \) is \( \Theta(n) \). Note that \( \lim_{n \to \infty}(2 + \sin n\pi/2) \) does not exist.

\[
\forall n \geq 1 \quad -1 \leq \sin\left(\frac{n\pi}{2}\right) \leq 1
\]
\[
1 \leq 2 + \sin\left(\frac{n\pi}{2}\right) \leq 3
\]
\[
n \leq n(2 + \sin\left(\frac{n\pi}{2}\right)) \leq 3n
\]

Taking \( n_0 = 1 \) and \( c_1 = 1, c_2 = 3 \), this proves \( n(2 + \sin\left(\frac{n\pi}{2}\right)) \in \Theta(n) \).

But \( \lim_{n \to \infty} \frac{n(2 + \sin\left(\frac{n\pi}{2}\right))}{n} = \lim_{n \to \infty} 2 + \sin\left(\frac{n\pi}{2}\right) \) does not exist.
Example 2

Prove that $n(2 + \sin n\pi/2)$ is $\Theta(n)$. Note that $\lim_{n \to \infty} (2 + \sin n\pi/2)$ does not exist.
Relationships between Order Notations

- \( f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n)) \)
- \( f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n)) \)
- \( f(n) \in o(g(n)) \iff g(n) \in \omega(f(n)) \)

- \( f(n) \in o(g(n)) \implies f(n) \in O(g(n)) \)
- \( f(n) \in o(g(n)) \implies f(n) \notin \Omega(g(n)) \)
- \( f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n)) \)
- \( f(n) \in \omega(g(n)) \implies f(n) \notin O(g(n)) \)
\[ f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n)) \]

Proof of \( \Rightarrow \). By assumption, \( \exists n_0, c_1, c_2 > 0 \) such that
\[ \forall n \geq n_0, \quad c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \]

\[ \Rightarrow \]
\[ c_2 \frac{1}{c_2} |g(n)| \leq |g(n)| \leq c_1 \frac{1}{c_1} |g(n)|. \]

So \( g(n) \in \Theta(f(n)) \).
Growth Rates

- If $f(n) \in \Theta(g(n))$, then the \textit{growth rates} of $f(n)$ and $g(n)$ are the \textit{same}.
- If $f(n) \in o(g(n))$, then we say that the growth rate of $f(n)$ is \textit{less than} the growth rate of $g(n)$.
- If $f(n) \in \omega(g(n))$, then we say that the growth rate of $f(n)$ is \textit{greater than} the growth rate of $g(n)$.
- Typically, $f(n)$ may be “complicated” and $g(n)$ is chosen to be a very simple function.
Example 3

Compare the growth rates of \( \log n \) and \( n \).

Now compare the growth rates of \((\log n)^c\) and \(n^d\) (where \(c > 0\) and \(d > 0\) are arbitrary numbers).
L'Hôpital: if \( \lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty \)

and \( \lim_{n \to \infty} \frac{f'(n)}{g'(n)} = L \) then \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = L \)

1) \( f(n) = \log(n) = \frac{\ln(n)}{\ln(2)} \) \( g(n) = n \)

\( f'(n) = \frac{1}{n} \cdot \frac{1}{\ln(2)} \) \( g'(n) = 1 \)

\( \frac{f'(n)}{g'(n)} = \frac{1}{n \ln(2)} \Rightarrow 0 \text{ when } n \to \infty \)

so \( (L'Hôpital) \frac{f(n)}{g(n)} \to 0 \text{ when } n \to \infty \)

so \( (\text{limit rule}) \ f(n) = o(g(n)) \).
2) \( f(n) = \log(n) \) and \( g(n) = n^a \), \( a > 0 \)

\[
\frac{f'(n)}{g'(n)} = \frac{1}{\frac{n \ln(n)}{a n^{a-1}}} = \frac{a}{n^a \ln(2)} \rightarrow 0 \text{ when } n \rightarrow \infty \quad \log(n) \in o(n^a).
\]

3) \( f(n) = \log(n)^c \) and \( g(n) = n^d \), \( c, d > 0 \)

\[
\frac{f(n)}{g(n)} = \frac{\log(n)^c}{n^d} = \left( \frac{\log(n)}{n^{d/c}} \right)^c \rightarrow \frac{\log(n)}{n^{d/c}} = \lim_{n \rightarrow \infty} \frac{\log(n)}{n^{d/c}} = 0 \quad \text{so } f(n) \in o(g(n)).
\]
Common Growth Rates

Commonly encountered growth rates in analysis of algorithms include the following (in increasing order of growth rate):

- $\Theta(1)$ (constant complexity),
- $\Theta(\log n)$ (logarithmic complexity),
- $\Theta(n)$ (linear complexity),
- $\Theta(n \log n)$ (linearithmic),
- $\Theta(n \log^k n)$, for some constant $k$ (quasi-linear),
- $\Theta(n^2)$ (quadratic complexity),
- $\Theta(n^3)$ (cubic complexity),
- $\Theta(2^n)$ (exponential complexity).
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- **constant complexity**: $T(n) = c$
- **logarithmic complexity**: $T(n) = c \log n$
- **linear complexity**: $T(n) = cn$
- **linearithmic $\Theta(n \log n)$**: $T(n) = cn \log n$
- **quadratic complexity**: $T(n) = cn^2$
- **cubic complexity**: $T(n) = cn^3$
- **exponential complexity**: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c$ \quad \sim \quad T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$
- linear complexity: $T(n) = cn$
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c \quad \leadsto \quad T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n \quad \leadsto \quad T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c$  \quad \Rightarrow \quad T(2n) = c.$
- logarithmic complexity: $T(n) = c \log n$  \quad \Rightarrow \quad T(2n) = T(n) + c.$
- linear complexity: $T(n) = cn$  \quad \Rightarrow \quad T(2n) = 2T(n).$
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance doubles (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c \quad \Rightarrow \quad T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n \quad \Rightarrow \quad T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn \quad \Rightarrow \quad T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n \quad \Rightarrow \quad T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance \textit{doubles} (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$ \quad $\implies$ \quad $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ \quad $\implies$ \quad $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ \quad $\implies$ \quad $T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$ \quad $\implies$ \quad $T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$ \quad $\implies$ \quad $T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$  \[\implies T(2n) = c.\]
- logarithmic complexity: $T(n) = c \log n$  \[\implies T(2n) = T(n) + c.\]
- linear complexity: $T(n) = cn$  \[\implies T(2n) = 2T(n).\]
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$  \[\implies T(2n) = 2T(n) + 2cn.\]
- quadratic complexity: $T(n) = cn^2$  \[\implies T(2n) = 4T(n).\]
- cubic complexity: $T(n) = cn^3$  \[\implies T(2n) = 8T(n).\]
- exponential complexity: $T(n) = c2^n$
How Growth Rates Affect Running Time

It is interesting to see how the running time is affected when the size of
the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c \implies T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n \implies T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn \implies T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n \implies T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2 \implies T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3 \implies T(2n) = 8T(n)$.
- exponential complexity: $T(n) = c2^n \implies T(2n) = (T(n))^2 / c$.

\[
\left(\diamond\right) \quad T(n+1) = c \cdot 2^{n+1} = 2 \cdot T(n)
\]
Outline

1 Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
   - Analysis of Algorithms I
   - Asymptotic Notation
   - Analysis of Algorithms II
   - Example: Analysis of MergeSort
   - Helpful Formulas
Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis.
- Running time of an algorithm depends on the input size $n$.

\[
\text{Test1}(n)
\]
1. $\text{sum} \leftarrow 0$
2. for $i \leftarrow 1$ to $n$ do
3. \hspace{1em} for $j \leftarrow i$ to $n$ do
4. \hspace{2em} $\text{sum} \leftarrow \text{sum} + (i - j)^2$
5. return $\text{sum}$

- Identify primitive operations that require $\Theta(1)$ time.
- The complexity of a loop is expressed as the sum of the complexities of each iteration of the loop.
- Nested loops: start with the innermost loop and proceed outwards. This gives nested summations.
Let $T_i(n)$ be the cost of Test $i(n)$.

$T_i(n) \in \Theta(S_i(n))$, where $S_i(n)$ is the number of times we go through Step 4.

$$S_i(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1.$$  

1. $\sum_{j=i}^{n} 1 = n-i+1 \rightarrow S_i(n) = \sum_{i=1}^{n} (n-i+1) = n^2 - \frac{n(n+1)}{2} + n$

$$= \frac{n^2}{2} + \frac{n}{2} \in \Theta(n^2).$$
\( S_i(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1 \)

\[ \text{big } O: \quad S_i(n) \leq \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n^2 \implies S_i(n) \in O(n^2). \]

\[ \text{big } \Omega: \quad S_i(n) \geq \sum_{i=1}^{\sqrt{n}} \sum_{j=i}^{\sqrt{n}} 1 = \sum_{i=1}^{\sqrt{n}} \sum_{j=\frac{i}{2}+1}^{\frac{n}{2}} 1 = \sum_{i=1}^{\sqrt{n}} \frac{n}{2} = \frac{n^2}{4} \]

\( S_i(n) \in \Omega(n^2) \implies S_i(n) \in \Theta(n^2) \)
Techniques for Algorithm Analysis

Two general strategies are as follows.

**Strategy I:** Use $\Theta$-bounds *throughout the analysis* and obtain a $\Theta$-bound for the complexity of the algorithm.

**Strategy II:** Prove a $O$-bound and a *matching* $\Omega$-bound *separately*.

Use upper bounds (for $O$-bounds) and lower bounds (for $\Omega$-bound) early and frequently.

This may be easier because upper/lower bounds are easier to sum.

---

**Test2**($A$, $n$)
1. $\text{max} \leftarrow 0$
2. $\text{for } i \leftarrow 1 \text{ to } n \text{ do}$
3. $\quad \text{for } j \leftarrow i \text{ to } n \text{ do}$
4. $\quad \quad \text{sum} \leftarrow 0$
5. $\quad \quad \text{for } k \leftarrow i \text{ to } j \text{ do}$
6. $\quad \quad \quad \text{sum} \leftarrow A[k]$
7. $\text{return } \text{max}$
let \( T_2(n) \) be the cost of Test2(4, n)

Then \( T_2(n) \in \Theta(S_2(n)) \), \( S_2(n) \) is the number of times we enter Step 6.

1. \( S_2(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1 = \frac{n(n^2+3n+2)}{6} \in \Theta(n^3) \).

2. \( S_2(n) \leq \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3 \) so \( S_2(n) \in \Omega(n^3) \).

3. \( S_2(n) \geq \sum_{i=1}^{\lfloor n/3 \rfloor} \sum_{j=1}^{n} \sum_{k=1}^{j} 1 \geq \sum_{i=1}^{\lfloor n/3 \rfloor} \sum_{j=\lfloor 2n/3 \rfloor}^{n} \sum_{k=1}^{j} 1 \)
$$S_2(n) \geq \sum_{i=1}^{n/3} \sum_{j=2m/3}^{n} \frac{n}{3}$$

$$\geq \sum_{i=1}^{n/3} \left( \frac{n}{3} \right)^2$$

$$\geq \left( \frac{n}{3} \right)^3$$

$$\Rightarrow S_2(n) \in \Omega(n^3)$$

$$\Rightarrow S_2(n) \in \Theta(n^3).$$
Complexity of Algorithms

- Algorithm can have different running times on two instances of the same size.

\[
\text{Test3}(A, n)
\]

\begin{align*}
A: & \text{ array of size } n \\
1. & \text{ for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
2. & j \leftarrow i \\
3. & \text{ while } j > 0 \text{ and } A[j] > A[j - 1] \text{ do} \\
4. & \quad \text{swap } A[j] \text{ and } A[j - 1] \\
5. & j \leftarrow j - 1.
\end{align*}

Let \( T_A(I) \) denote the running time of an algorithm \( A \) on instance \( I \).

Worst-case complexity of an algorithm: take the worst \( I \)

Average-case complexity of an algorithm: average over \( I \)
Complexity of Algorithms

**Worst-case complexity of an algorithm:** The worst-case running time of an algorithm $\mathcal{A}$ is a function $f: \mathbb{Z}^+ \rightarrow \mathbb{R}$ mapping $n$ (the input size) to the longest running time for any input instance of size $n$:

$$T_{\mathcal{A}}(n) = \max \{ T_{\mathcal{A}}(I) : \text{Size}(I) = n \}.$$

**Average-case complexity of an algorithm:** The average-case running time of an algorithm $\mathcal{A}$ is a function $f: \mathbb{Z}^+ \rightarrow \mathbb{R}$ mapping $n$ (the input size) to the average running time of $\mathcal{A}$ over all instances of size $n$:

$$T_{\mathcal{A}}^{\text{avg}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I : \text{Size}(I) = n\}} T_{\mathcal{A}}(I).$$
O-notation and Complexity of Algorithms

- It is important not to try and make *comparisons* between algorithms using O-notation.

- For example, suppose algorithm $A_1$ and $A_2$ both solve the same problem, $A_1$ has worst-case run-time $O(n^3)$ and $A_2$ has worst-case run-time $O(n^2)$.

- Observe that we *cannot* conclude that $A_2$ is more efficient than $A_1$ for all input!
  1. The worst-case run-time may only be achieved on some instances.
  2. O-notation is an upper bound. $A_1$ may well have worst-case run-time $O(n)$. If we want to be able to compare algorithms, we should always use Θ-notation.
1 Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
   - Analysis of Algorithms I
   - Asymptotic Notation
   - Analysis of Algorithms II
   - Example: Analysis of MergeSort
   - Helpful Formulas
Design of MergeSort

**Input:** Array $A$ of $n$ integers

- **Step 1:** We split $A$ into two subarrays: $A_L$ consists of the first $\left\lfloor \frac{n}{2} \right\rfloor$ elements in $A$ and $A_R$ consists of the last $\left\lfloor \frac{n}{2} \right\rfloor$ elements in $A$.

- **Step 2:** Recursively run *MergeSort* on $A_L$ and $A_R$.

- **Step 3:** After $A_L$ and $A_R$ have been sorted, use a function *Merge* to merge them into a single sorted array.

if $n=1$, return.
MergeSort

\textit{MergeSort}(A, \ell \leftarrow 0, r \leftarrow n - 1, S \leftarrow \text{NIL})

A: array of size \( n \), \( 0 \leq \ell \leq r \leq n - 1 \)

1. \textbf{if} \( S \) is NIL \textbf{ then} initialize it as array \( S[0..n-1] \leftarrow \)
2. \textbf{if} \( r \leq \ell \) \textbf{ then}
   \hspace{1cm} return
3. \textbf{else}
4. \hspace{1cm} \( m = (r + \ell)/2 \)
5. \hspace{2cm} \textit{MergeSort}(A, \ell, m, S)
6. \hspace{2cm} \textit{MergeSort}(A, m + 1, r, S)
7. \hspace{2cm} \textit{Merge}(A, \ell, m, r, S)

Two tricks to reduce run-time and auxiliary space:

- The recursion uses parameters that indicate the range of the array that needs to be sorted.
- The array used for copying is passed along as parameter.
Merge

\[ \text{Merge}(A, \ell, m, r, S) \]

\( A[0..n-1] \) is an array, \( A[\ell..m] \) is sorted, \( A[m+1..r] \) is sorted

\( S[0..n-1] \) is an array

1. copy \( A[\ell..r] \) into \( S[\ell..r] \)
2. int \( i_L \leftarrow \ell \); int \( i_R \leftarrow m+1 \);
3. for \((k \leftarrow \ell; k \leq r; k++ \)) do

4. \( \text{if } (i_L > m) \) \( A[k] \leftarrow S[i_R++] \)
5. \( \text{else if } (i_R > r) \) \( A[k] \leftarrow S[i_L++] \)
6. \( \text{else if } (S[i_L] \leq S[i_R]) \) \( A[k] \leftarrow S[i_L++] \)
7. \( \text{else } A[k] \leftarrow S[i_R++] \)

\text{Merge} \text{ takes time } \Theta(r - \ell + 1), \text{ i.e., } \Theta(n) \text{ time for merging } n \text{ elements.}
\[ A = \begin{bmatrix} 2 & 4 & 7 & 5 & 6 \end{bmatrix} \]

\[ S = \begin{bmatrix} 2 & 4 & 7 & 5 & 6 \end{bmatrix} \]

\[ \uparrow \uparrow \uparrow \uparrow \uparrow \]

\[ i_L = l = 0 \quad i_R = m + 1 \]

\[ h = 0 \quad A = [2 \quad \_ \quad \_ \quad \_ \quad \_] \quad i_L = 1 \]

\[ h = 1 \quad A = [2 \quad 4 \quad \_ \quad \_ \quad \_] \quad i_L = 2 \]

\[ h = 3 \quad A = [2 \quad 4 \quad 5 \quad \_ \quad \_] \quad i_R = 4 \]

\[ h = 4 \quad A = [2 \quad 4 \quad 5 \quad 6 \_ \_] \quad i_R = 5 \]

\[ l = 5 \quad A = [2 \quad 4 \quad 5 \quad 6 \quad 7 \_ \_] \quad i_L = 3 \]
Analysis of MergeSort

Let $T(n)$ denote the time to run $\text{MergeSort}$ on an array of length $n$.
- creating $S$ takes time $\Theta(n)$ ✓
- recursive calls take time $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ ✓
- merging takes time $\Theta(n)$ ✓

The recurrence relation for $T(n)$ is as follows:

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

It suffices to consider the following exact recurrence, with constant factor $c$ replacing $\Theta$'s: (requires proof !)

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn & \text{if } n > 1 \\ c & \text{if } n = 1. \end{cases}$$
Analysis of MergeSort

- The following is the corresponding **sloppy recurrence** (it has floors and ceilings removed):

\[
T(n) = \begin{cases}
2 \frac{T(n/2)}{2} + cn & \text{if } n > 1 \\
c & \text{if } n = 1.
\end{cases}
\]

- The exact and sloppy recurrences are **identical** when \( n \) is a power of 2.

- The recurrence can easily be solved by various methods when \( n = 2^i \). The solution has growth rate \( T(n) \in \Theta(n \log n) \).

- It is possible to show that \( T(n) \in \Theta(n \log n) \) **for all** \( n \) by analyzing the exact recurrence.
Let \( n = 2^k \)

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn \\
T(1) = c
\]

\[
T(2^k) = 2T(2^{k-1}) + c2^k
\]

\[
= 2\left(2T(2^{k-2}) + c\cdot2^{k-1}\right) + c2^k
\]

\[
= 2^2 T(2^{k-2}) + 2c2^k
\]

\[
= 2^2\left(2T(2^{k-3}) + c\cdot2^{k-2}\right) + 2c2^k
\]

\[
= 2^3 T(2^{k-3}) + 3c2^k
\]

\[
= 2^4 T(2^{k-4}) + 4c2^k
\]

\[
= \ldots = 2^k T(2^{k-k}) + k\cdot c2^k
\]
\[ \frac{\tau(n)}{n \log(n)} = \frac{cn \log(n) + \Theta(n)}{n \log(n)} = C + \frac{c}{\log(n)} \]

limit rule \( \Rightarrow \tau(n) \in \Theta(n \log n) \)
Some Recurrence Relations

<table>
<thead>
<tr>
<th>Recursion</th>
<th>resolves to</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + \Theta(1)$</td>
<td>$T(n) \in \Theta(\log n)$</td>
<td>Binary search</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + \Theta(n)$</td>
<td>$T(n) \in \Theta(n \log n)$</td>
<td>Mergesort</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + \Theta(\log n)$</td>
<td>$T(n) \in \Theta(n)$</td>
<td>Heapify (→ later)</td>
</tr>
<tr>
<td>$T(n) = T(cn) + \Theta(n)$ for some $0 &lt; c &lt; 1$</td>
<td>$T(n) \in \Theta(n)$</td>
<td>Selection (→ later)</td>
</tr>
<tr>
<td>$T(n) = 2T(n/4) + \Theta(1)$</td>
<td>$T(n) \in \Theta(\sqrt{n})$</td>
<td>Range Search (→ later)</td>
</tr>
<tr>
<td>$T(n) = T(\sqrt{n}) + \Theta(1)$</td>
<td>$T(n) \in \Theta(\log \log n)$</td>
<td>Interpolation Search (→ later)</td>
</tr>
</tbody>
</table>

- Once you know the result, it is (usually) easy to prove by induction.
- Many more recursions, and some methods to find the result, in cs341.
Outline

1. Introduction and Asymptotic Analysis
   - CS240 Overview
   - Algorithm Design
   - Analysis of Algorithms I
   - Asymptotic Notation
   - Analysis of Algorithms II
   - Example: Analysis of MergeSort
   - Helpful Formulas
Order Notation Summary

**O-notation:** $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $|f(n)| \leq c |g(n)|$ for all $n \geq n_0$.

**Ω-notation:** $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $c |g(n)| \leq |f(n)|$ for all $n \geq n_0$.

**Θ-notation:** $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ for all $n \geq n_0$.

**o-notation:** $f(n) \in o(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $|f(n)| \leq c |g(n)|$ for all $n \geq n_0$.

**ω-notation:** $f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $c |g(n)| \leq |f(n)|$ for all $n \geq n_0$. 

\[
|g(n)| \leq c |f(n)| \quad \text{or} \quad g(n) \leq \frac{1}{c} f(n)
\]
Useful Sums

**Arithmetic sequence:**
\[ \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \quad \sum_{i=0}^{n-1} (a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \quad \text{if } d \neq 0. \]

**Geometric sequence:**
\[ \sum_{i=0}^{n-1} 2^i = 2^n - 1 \quad \sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{r^n - 1}{r - 1} & \in \Theta(r^{n-1}) \quad \text{if } r > 1 \\ na & \in \Theta(n) \quad \text{if } r = 1 \\ \frac{1 - r^n}{1 - r} & \in \Theta(1) \quad \text{if } 0 < r < 1. \end{cases} \]

**Harmonic sequence:**
\[ \sum_{i=1}^{n} \frac{1}{i} = H_n := \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n) \]

**A few more:**
\[ \sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1) \quad \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \quad \text{for } k \geq 0 \]
Useful Math Facts

Logarithms:
- $c = \log_b(a)$ means $b^c = a$. E.g. $n = 2^{\log n}$.
- $\log(a)$ (in this course) means $\log_2(a)$
- $\log(a \cdot c) = \log(a) + \log(c)$, $\log(a^c) = c \log(a)$
- $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$, $a^{\log_b c} = c^{\log_b a}$
- $\ln(x)$ = natural log = $\log_e(x)$, $\frac{d}{dx} \ln x = \frac{1}{x}$
- concavity: $\alpha \log x + (1-\alpha) \log y \leq \log(\alpha x + (1-\alpha)y)$ for $0 \leq \alpha \leq 1$

Factorial:
- $n! := n(n-1)(n-2) \cdots 2 \cdot 1 = \#$ ways to permute $n$ elements
- $\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n \log n)$

Probability and moments: