CS 240 – Data Structures and Data Management

Module 1: Introduction and Asymptotic Analysis

Leili Rafiee Sevyeri   Éric Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2023

version *2023-05-08 15:00*

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

Course Objectives: What is this course about?

- Much of Computer Science is *problem solving*: Write a program that converts the given input to the expected output.
- 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.

  Motivating examples: Digital Music Collection, English Dictionary

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

Course Objectives: What is this course about?

- We will consider various **abstract data types** (ADTs) and how to realize 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
- 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

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: *Size*(I) is a positive integer which is a measure of the size of the instance I.

Example: Sorting problem

Algorithms and Programs

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 Π if, for every instance I of Π, 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

Pseudocode: a method of communicating an algorithm to another person.

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

Pseudocode

- 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 Π, we can have several algorithms.

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

Algorithms in practice: Given a problem Π

1. Design an algorithm $\mathcal{A}$ that solves Π. $\rightarrow$ **Algorithm Design**
2. Assess *correctness* and *efficiency* of $\mathcal{A}$. $\rightarrow$ **Algorithm Analysis**
3. If acceptable (correct and efficient), implement $\mathcal{A}$.

Outline

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. $\rightarrow$ **Running Time**
- We also may be interested in the *amount of additional memory* the program requires. $\rightarrow$ **Auxiliary space**
- The amount of time and/or memory required by a program will depend on *Size(I)*, the size of the given problem instance 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: *hardware* (processor, memory), *software environment* (OS, compiler, programming language), and *human factors* (programmer).
- We cannot test all inputs; what are good *sample inputs*?
- We cannot easily compare two algorithms/programs.

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 *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* (this is called the *complexity* of the algorithm).

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 $10n^2$?

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 (Matrix multiplication, approximately):** What is larger: $4n^3$, $300n^{2.807}$, or $10^{67}n^{2.373}$?

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 (Matrix multiplication, approximately):** What is larger: $4n^3$, $300n^{2.807}$, or $10^{67}n^{2.373}$?

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

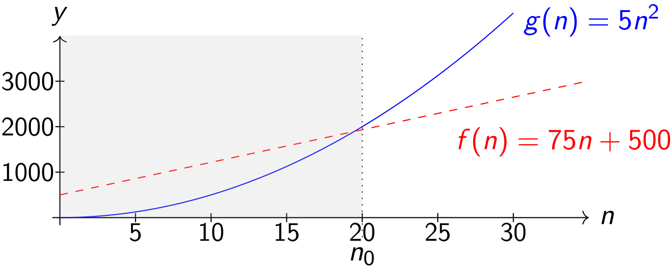
Outline

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:** $f(n) \in O(g(n))$ (f is *asymptotically bounded above* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ 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.

Example 1: 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:

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

note that not all choices of c and n_0 will work.

Aymptotic 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: $f(n) \in \Omega(g(n))$ (f is *asymptotically bounded below* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ such that $c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$.

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

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

Examples 2-4: Order Notation

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

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.

Strictly smaller/larger asymptotic bounds

- We have $f(n) = 2n^2 + 3n + 11 \in \Theta(n^2)$.
- How to express that $f(n)$ grows slower than n^3?

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

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

- Main difference to O, Ω is the quantifier for c.
- Rarely proved from first principles.

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:

- $f(n) + g(n) \in O(\max\{f(n), g(n)\})$
- $f(n) + g(n) \in \Omega(\max\{f(n), g(n)\})$

Proof: $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2\max\{f(n), g(n)\}$

Techniques for Order Notation

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

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)} \quad \text{(in particular, the limit exists).}$$

Then

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

Note that this result gives *sufficient* (but not necessary) conditions for the stated conclusion to hold.

Example 5: Polynomials

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 $c_d > 0$.

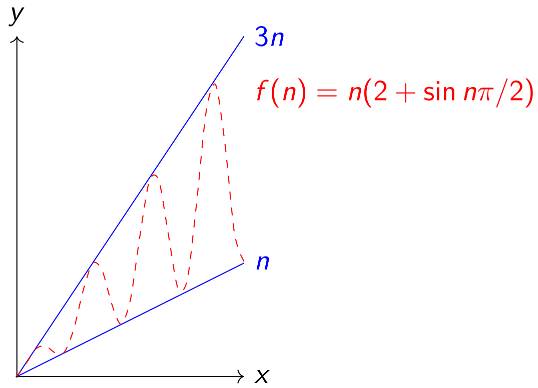
Then $f(n) \in \Theta(n^d)$:

Example 6: Sine

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.

Example 6: Sine

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.



Examples 7-8

Compare the growth rates of $f(n) = \log n$ and $g(n) = n$ using *l'Hôpital's rule*

Now compare the growth rates of $f(n) = (\log n)^c$ and $g(n) = n^d$ (where $c > 0$ and $d > 0$ are arbitrary numbers).

Growth rates

- If $f(n) \in \Theta(g(n))$, then the *growth rates* of $f(n)$ and $g(n)$ are the *same*.
- If $f(n) \in o(g(n))$, then we say that the growth rate of $f(n)$ is *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 *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.

Common Growth Rates

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

- $\Theta(1)$ (*constant*),
- $\Theta(\log n)$ (*logarithmic*),
- $\Theta(n)$ (*linear*),
- $\Theta(n \log n)$(*linearithmic*),
- $\Theta(n \log^k n)$, for some constant k (*quasi-linear*),
- $\Theta(n^2)$ (*quadratic*),
- $\Theta(n^3)$ (*cubic*),
- $\Theta(2^n)$ (*exponential*).

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: $T(n) = c\,2^n$

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: $T(n) = c$ $\rightsquigarrow$ $T(2n) = c$.
- 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: $T(n) = c\, 2^n$

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: $T(n) = c$ $\rightsquigarrow$ $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow$ $T(2n) = T(n) + c$.
- 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: $T(n) = c\, 2^n$

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: $T(n) = c$ $\rightsquigarrow$ $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow$ $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ $\rightsquigarrow$ $T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\,2^n$

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: $T(n) = c$ $\rightsquigarrow$ $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow$ $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ $\rightsquigarrow$ $T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$ $\rightsquigarrow$ $T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\,2^n$

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: $T(n) = c$ $\rightsquigarrow$ $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow$ $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ $\rightsquigarrow$ $T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$ $\rightsquigarrow$ $T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$ $\rightsquigarrow$ $T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\,2^n$

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

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

Relationships between Order Notations

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

- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

Order Notation Summary

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

Outline

Techniques for Run-time Analysis

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

```
Test1(n)
    sum ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← sum + (i − j)^2
    return 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*.

Two techniques for Run-time Analysis

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

Strategy II: Prove a O-bound and a *matching* Ω-bound *separately*.
Use upper bounds (for O-bounds) and lower bounds (for Ω-bound) early and frequently.
This may be easier because upper/lower bounds are easier to sum.

```
Test2(A, n)
    max ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← A[k]
    return max
```

Complexity of Algorithms

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

```
Test3(A, n)
A: array of size n
    for i ← 1 to n − 1 do
        j ← i
        while j > 0 and A[j] < A[j − 1] do
            swap A[j] and A[j − 1]
            j ← j − 1
```

Let $T_{\mathcal{A}}(I)$ denote the running time of an algorithm $\mathcal{A}$ on instance I.

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

Average-case complexity of an algorithm: average over I

Complexity of Algorithms

Worst-case (best-case) complexity of an algorithm: The *worst-case (best-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 (shortest)* running time for any input instance of size n:

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

$$T_{\mathcal{A}}(n)^{best} = \min\{T_{\mathcal{A}}(I) : Size(I) = n\}$$

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:

$$T_{\mathcal{A}}^{avg}(n) = \frac{1}{|\{I : Size(I) = n\}|} \sum_{\{I : Size(I) = n\}} T_{\mathcal{A}}(I)$$

O-notation and Complexity of Algorithms

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

- For example, suppose algorithm $\mathcal{A}_1$ and $\mathcal{A}_2$ both solve the same problem, $\mathcal{A}_1$ has worst-case run-time $O(n^3)$ and $\mathcal{A}_2$ has worst-case run-time $O(n^2)$.

- Observe that we *cannot* conclude that $\mathcal{A}_2$ is more efficient than $\mathcal{A}_1$ for all input!
  1. The worst-case run-time may only be achieved on some instances.
  2. O-notation is an upper bound. $\mathcal{A}_1$ may well have worst-case run-time $O(n)$. If we want to be able to compare algorithms, we should always use Θ-notation.

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

Design Idea for MergeSort

Input: Array A of n integers

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

MergeSort

```
MergeSort(A, n, ℓ ← 0, r ← n − 1, S ← NIL)
A: array of size n, 0 ≤ ℓ ≤ r ≤ n − 1
    if S is NIL   initialize it as array S[0..n − 1]
    if (r ≤ ℓ) then
        return
    else
        m = ⌊(r + ℓ)/2⌋
        MergeSort(A, n, ℓ, m, S)
        MergeSort(A, n, m + 1, r, S)
        Merge(A, ℓ, 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 a parameter.

Merge

Merge(A, ℓ, 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

```
copy A[ℓ..r] into S[ℓ..r]
(i_L, i_R) ← (ℓ, m + 1);
for (k ← ℓ; k ≤ r; k++) do
    if (i_L > m) A[k] ← S[i_R++]
    else if (i_R > r) A[k] ← S[i_L++]
    else if (S[i_L] ≤ S[i_R]) A[k] ← S[i_L++]
    else A[k] ← S[i_R++]
```

Merge takes time $\Theta(r - \ell + 1)$, i.e., $\Theta(n)$ time for merging n elements.

Analysis of MergeSort

Let $T(n)$ denote the time to run *MergeSort* on an array of length n.

- Step 1 (initialize array) takes time $\Theta(n)$
- Step 2 (recursively call *MergeSort*) takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
- Step 3 (call *Merge*) takes time $\Theta(n)$

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

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

It suffices to consider the following *exact recurrence*, with constant factor c replacing Θ's:

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

Analysis of MergeSort

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

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ c & \text{if } n = 1. \end{cases}$$

- The exact and sloppy recurrences are *identical* when n is a power of 2.
- The recurrence can easily be solved by various methods when $n = 2^j$. 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.

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Sometimes, it's convenient to abuse notation and treat it like a value:
  - $f(n) = n^2 + \Theta(n)$
  - $f(n) = n^2 + O(n)$
  - $f(n) = n^2 + O(1)$
  - $f(n) = n^2 + o(1)$

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Sometimes, it's convenient to abuse notation and treat it like a value:
  - $f(n) = n^2 + \Theta(n) \rightsquigarrow f(n)$ is a quadratic function plus a linear term.
  - $f(n) = n^2 + O(n)$
  - $f(n) = n^2 + O(1)$
  - $f(n) = n^2 + o(1)$

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Sometimes, it's convenient to abuse notation and treat it like a value:
  - $f(n) = n^2 + \Theta(n) \rightsquigarrow f(n)$ is a quadratic function plus a linear term.
  - $f(n) = n^2 + O(n) \rightsquigarrow \ldots$ plus a term that grows at most linearly.
  - $f(n) = n^2 + O(1)$
  - $f(n) = n^2 + o(1)$

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Sometimes, it's convenient to abuse notation and treat it like a value:
  - $f(n) = n^2 + \Theta(n) \rightsquigarrow f(n)$ is a quadratic function plus a linear term.
  - $f(n) = n^2 + O(n) \rightsquigarrow \ldots$ plus a term that grows at most linearly.
  - $f(n) = n^2 + O(1) \rightsquigarrow \ldots$ plus a constant term.
  - $f(n) = n^2 + o(1)$

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Sometimes, it's convenient to abuse notation and treat it like a value:
  - $f(n) = n^2 + \Theta(n) \rightsquigarrow f(n)$ is a quadratic function plus a linear term.
  - $f(n) = n^2 + O(n) \rightsquigarrow \ldots$ plus a term that grows at most linearly.
  - $f(n) = n^2 + O(1) \rightsquigarrow \ldots$ plus a constant term.
  - $f(n) = n^2 + o(1) \rightsquigarrow \ldots$ plus a vanishing term.

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

Some Recurrence Relations

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify (*)
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	Selection (*)
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search (*)
$T(n) = T(\sqrt{n}) + \Theta(\sqrt{n})$	$T(n) \in \Theta(\sqrt{n})$	Interpol. Search (*)
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpol. Search (*)

- 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.

(*) These will be studied later in the course.

Useful Sums

Arithmetic sequence:

$\sum_{i=0}^{n-1} i = ???$ $\qquad$ $\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \quad \text{if } d \neq 0.$

Geometric sequence:

$\sum_{i=0}^{n-1} 2^i = ???$ $\qquad$ $$\sum_{i=0}^{n-1} a\, r^i = \begin{cases} a\dfrac{r^n - 1}{r-1} & \in \Theta(r^{n-1}) & \text{if } r > 1 \\ na & \in \Theta(n) & \text{if } r = 1 \\ a\dfrac{1-r^n}{1-r} & \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$

Harmonic sequence:

$\sum_{i=1}^{n} \frac{1}{i} = ???$ $\qquad$ $H_n := \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$

A few more:

$\sum_{i=1}^{n} \frac{1}{i^2} = ???$ $\qquad$ $\sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1)$

$\sum_{i=1}^{n} i^k = ???$ $\qquad$ $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \quad \text{for } k \geq 0$

Useful Math Facts

Logarithms:

- $c = \log_b(a)$ means $b^c = a$. e.g. $n = 2^{\log n}$.
- $\log(a)$ (in this course) means $\log_2(a)$
- $\log(a \cdot c) = \log(a) + \log(c)$, $\log(a^c) = c\log(a)$, $\log x \le x$
- $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$, $a^{\log_b c} = c^{\log_b a}$
- $\ln(x) =$ natural log $= \log_e(x)$, $\frac{\mathrm{d}}{\mathrm{d}x} \ln x = \frac{1}{x}$

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

- $E[X]$ is the expected value of X.
- $E[aX] = aE[X]$, $E[X+Y] = E[X] + E[Y]$ (linearity of expectation)