

Module 1: Introduction and Asymptotic Analysis

CS 240 - Data Structures and Data Management

Therese Biedl Daniela Maftuleac

Based on lecture notes by many previous cs240 instructors

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

Winter 2015

Course Information

- Instructors

- ▶ Therese Biedl, DC 2341
biedl [at] uwaterloo.ca
- ▶ Daniela Maftuleac, DC 2305C
dmaftule [at] uwaterloo.ca

- Lectures

- ▶ Section 001: TTh 01:00pm-02:20pm in MC 4021
- ▶ Section 002: TTh 11:30am-12:50pm in MC 4061
- ▶ (Section 003: TTh 11:30am-12:50pm in MC 4064)

- Office hours, phone numbers etc.

- ▶ See web page

Course Information

Instructional Support

- Coordinator: Karen Anderson (MC 4010)
kaanders [at] uwaterloo.ca
- Assistant: Joseph Daniel Sitter (MC 4065)
cs240 [at] student.cs.uwaterloo.ca
- Office hours
 - ▶ TBD

Tutorials

- Monday 01:30pm-02:20pm MC 4060
- Monday 02:30pm-03:20pm MC 4058
- Monday 03:30pm-04:20pm MC 4040
- Monday 04:30pm-05:20pm MC 4042
- (Monday 03:30pm-04:20pm MC 4064)

Tutorial next week on \LaTeX

Assignment 0 to learn \LaTeX (5 bonus marks on assignment 1)

Course Information

● Course Webpage

<http://www.student.cs.uwaterloo.ca/~cs240/w15/>

Primary source for up-to-date information for CS 240.

- ▶ Lecture slides
- ▶ Assignments / Solution Sketches
- ▶ Course policies

● Main resource: Lectures

- ▶ Course slides will be available on the webpage before each lecture

● Textbooks

- ▶ Algorithms in C++, by Robert Sedgewick, Addison-Wesley, 1998
- ▶ More books on the webpage under Resources
- ▶ Topics and references for each lecture will be posted on the Webpage

Electronic Communication in CS240

Piazza

<https://piazza.com/uwaterloo.ca/winter2015/cs240>

- A forum that is optimized for asking questions and giving answers.
- You must sign up using your uwaterloo email address.
 - ▶ You can post to piazza using a nickname though.
- Posting solutions to assignments is forbidden.

Email

cs240@student.cs.uwaterloo.ca

- For private communication between students and course staff.
- You should be sending emails from your uwaterloo email address.

Mark Breakdown

- Final 50%
- Midterm 25%
 - ▶ Thursday, February 26, 4:30pm
- Assignments 25%
 - ▶ 5 assignments each worth 5%
 - ▶ Approximately every 2 weeks
 - ▶ Due Wednesday, 8:30am
 - ▶ No lates allowed
 - ▶ Follow the **assignment guidelines**
 - ▶ All assignment to be submitted electronically via MarkUs

Mark Breakdown

- Final 50%
- Midterm 25%
 - ▶ Thursday, February 26, 4:30pm
- Assignments 25%
 - ▶ 5 assignments each worth 5%
 - ▶ Approximately every 2 weeks
 - ▶ Due Wednesday, 8:30am
 - ▶ No lates allowed
 - ▶ Follow the **assignment guidelines**
 - ▶ All assignment to be submitted electronically via MarkUs

Note: You must pass the weighted average of the midterm and the final exam to pass the course

Cheating

- Cheating includes not only copying the work of another person (or letting another student copy your work), but also excessive collaboration.
- Standard penalties: a grade of 0 on the assignment you cheated on, and a deduction of 5% from your course grade. You will also be reported to the Associate Dean of Undergraduate Studies.
- Do **not** take notes during discussions with classmates.

Courtesy

- **Cardinal rule:** Do nothing that keeps your neighbour from learning.
- Please silence cell phones before coming to class.
- Questions are encouraged, but please refrain from talking in class.
- Does a laptop help, or does it distract?

Multiple sections

- Sections 001 and 002:
 - ▶ Same content, same assignments, (roughly) same lecture speed.
 - ▶ Attending other section is allowed while there is space.
 - ▶ No extra chairs in aisles (fire regulation).
- Section 003 **different**:
 - ▶ Enhanced version, higher speed, additional material.
 - ▶ Assignment 1 (essentially) the same.
 - ▶ Switching into it within first weeks only, needs special permission.

Advice

Attend all the lectures and pay attention!

Study the slides before the lectures, and again afterwards.

Read the reference materials to get different perspectives on the course material.

Keep up with the course material! Don't fall behind.

If you're having difficulties with the course, seek help.

Data structure - store & maintain data to support certain operations
operations - insert/delete/search/sort/min/max
ADT - concept → stack {^{push}_{pop}}
DS - implementation → linked list

Course Objectives: What is this course about?

- The objective of the course is to study efficient methods of *storing*, *accessing*, and performing *operations* on 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
- 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 pseudocode and analyzed using order notation (big-Oh, etc.).

Course Topics **Dictionary** - a set of "items" with the following operations: insert, delete, search

- 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)

Problems (terminology)

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 . *Sorting Problem*:

Example: Sorting problem

Given a set/array of numbers, ↴ of size n
arrange them ↴ in increasing order.
↳ operation
↳ input ↳ output

Algorithms and Programs

Algorithm: An algorithm is a *step-by-step process* (e.g., described in pseudocode) for carrying out a series of computations, given an arbitrary problem instance I .

Algorithm 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

Solution \equiv an algorithm
1 problem \equiv m different algorithms

For a problem Π , we can have several algorithms.

For an algorithm A solving Π , we can have several programs (implementations).

Algorithms in practice: Given a problem Π

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

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 memory* the program requires. \rightarrow **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

- We must implement the algorithm.
- 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*.

Running Time Simplifications

Overcome dependency on hardware/software

- Express algorithms using *pseudo-code*
- Instead of time, count the number of *primitive operations*

Random Access Machine (RAM) Model: *uses Random Access Memory*

- The *random access machine* has a set of memory cells, each of which stores one item (word) of data.
- Any *access to a memory location* takes constant time.
- Any *primitive operation* takes constant time.
- The *running time* of a program can be computed to be 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

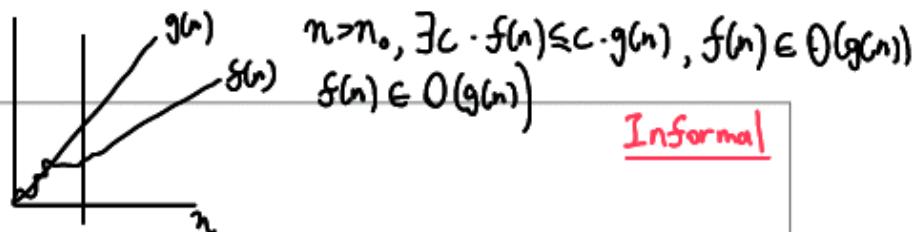
Overcome dependency on hardware/software

- Express algorithms using *pseudo-code*.
- Instead of time, count the number of *primitive operations*.
- Implicit assumption: primitive operations have fairly similar, though different, running time on different systems

Simplify Comparisons

- Example: Compare $1000000n + 200000000000000$ with $0.01n^2$
- Idea: Use *order notation*
- Informally: ignore constants and lower order terms

Order Notation



O-notation: $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$. $f(n) \leq g(n)$ $2n \in O(n)$

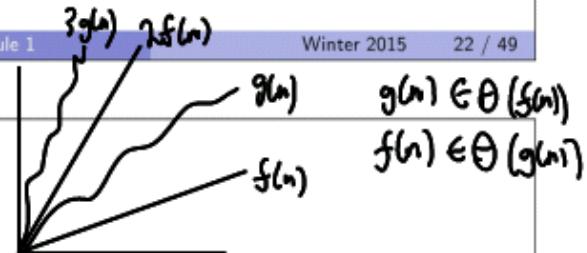
Ω -notation: $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq c g(n) \leq f(n)$ for all $n \geq n_0$. $f(n) \geq g(n)$ $\frac{1}{2}n \in \Omega(n)$

Θ -notation: $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. $f(n) = g(n)$ $\frac{1}{2}n \in \Theta(n)$

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

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

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:

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

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

$$\exists n_0 \exists c \forall n \geq n_0 2n^2 + 3n + 11 \leq c \cdot n^2$$

need to find n_0 and c .

lets say $n_0 = 10$
 $c = 3$

$$\underline{2 + \frac{3}{n} + \frac{11}{n^2} \leq c}, \underline{2 + \frac{3}{10} + \frac{11}{100} \leq 3}$$

$$2n^2 + 3n + 11 \leq 3n^2$$

$3n + 11 \leq n^2$ true for all

$$n > 10$$

Example of Order Notation

Prove that $2010n^2 + 1388n \in o(n^3)$ from first principles.

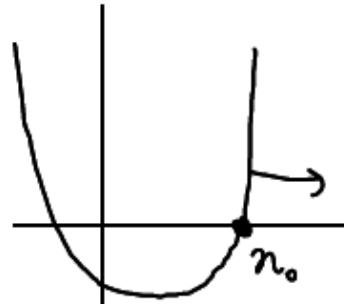
find n_0 st $n \geq n_0$ $2010n^2 + 1388n < c \cdot g(n)$ for all $c > 0$

$$2010n^2 + 1388n < c \cdot n^3 \rightarrow \text{we have to express } n \text{ as a function of } c$$
$$cn^3 - 2010n^2 - 1388n > 0$$

$$cn^3 - 2010n^2 - 1388n > 0$$

$$\Delta = 2010^2 + 4 \times 1388 \times c$$

$$n_0 = \max \left\{ \frac{2010 + \sqrt{2010^2 + 4 \times 1388 \times c}}{2c}, 1 \right\}$$



Complexity of Algorithms

Our goal: Express the running time of each algorithm as a function $f(n)$ in terms of the **input size**.

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

An algorithm can have different running times on input instances of the same size.

Quicksort

Average-case complexity of an algorithm

$n \log n$

Worst-case complexity of an algorithm

n^2

$$f(n) = \underbrace{n}_{\Theta(n)} + \underbrace{2\sqrt{n} \log(n)}_{O(n)} \quad g(n) = n \quad \text{prove } f(n) \in \Theta(g(n))$$

$$\begin{array}{ll} \text{Intuition: } \log(n) \in O(\sqrt{n}) & n + 2\sqrt{n} \log(n) \in O(n+2n) \\ \sqrt{n} \log(n) \in O(\sqrt{n}, \sqrt{n}) & \in O(n) \\ \in O(n) & \end{array}$$

We need to find C_1, C_2, n_0 st for $n \geq n_0$

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$$

$$C_1 \cdot g(n) \leq n + 2\sqrt{n} \log(n) \leq C_2 \cdot n$$

$C_1 = 1$ works :-)

for C_2 :

$$n + 2\sqrt{n} \log(n) \leq C_2 \cdot n \quad \text{E.g. } n_0 = 64 \quad \sqrt{n} = 8$$

$$1 + \frac{2\sqrt{n} \log(n)}{n} \leq C_2 \quad \log 64 = 6$$

$$1 + \frac{2 \cdot 8 \cdot 6}{64} \leq C_2 \rightarrow \text{for that it suffices to have } C_2 = 3$$

$$\therefore C_1 = 1, C_2 = 3, n_0 = 64$$

Intuition	$f(n) \in O(g(n))$	Asymptotically	$x = 9$	$x < 10$	$f(x) = x^9$	$f(n) \in O(n^{10})$
$\in \Omega$	\geq		$y = 2$	$y < 20$	$g(n) = n$	$g(n) \in O(n^{20})$
$\in \Theta$	$=$					
$\in o$	$<$					
$\in w$	$>$					

Complexity of Algorithms

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

$$T_A^{\text{avg}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I : \text{Size}(I) = n\}} T_A(I).$$

Worst-case complexity of an algorithm: The worst-case running time of an algorithm 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 :

$$T_A(n) = \max\{T_A(I) : \text{Size}(I) = n\}.$$

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 complexity*),
- $\Theta(\log n)$ (*logarithmic complexity*),
- $\Theta(n)$ (*linear complexity*),
- $\Theta(n \log n)$ (*linearithmic/pseudo-linear complexity*),
- $\Theta(n^2)$ (*quadratic complexity*),
- $\Theta(n^3)$ (*cubic complexity*),
- $\Theta(2^n)$ (*exponential complexity*).

$$\log^i(n) \in O(n^j) \quad \text{for all } i, j$$

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: $T(n) = cn$
- $\Theta(n \log n)$: $T(n) = cn \log n$
- 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: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$
- linear complexity: $T(n) = cn$
- $\Theta(n \log n)$: $T(n) = cn \log n$
- 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: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$
- $\Theta(n \log n)$: $T(n) = cn \log n$
- 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: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$, $T(2n) = 2T(n)$.
- $\Theta(n \log n)$: $T(n) = cn \log n$
- 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: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$, $T(2n) = 2T(n)$.
- $\Theta(n \log n)$: $T(n) = cn \log n$, $T(2n) = 2T(n) + 2cn$.
- 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 doubles (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$, $T(2n) = 2T(n)$.
- $\Theta(n \log n)$: $T(n) = cn \log n$, $T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$, $T(2n) = 4T(n)$.
- 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., $n \rightarrow 2n$).

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

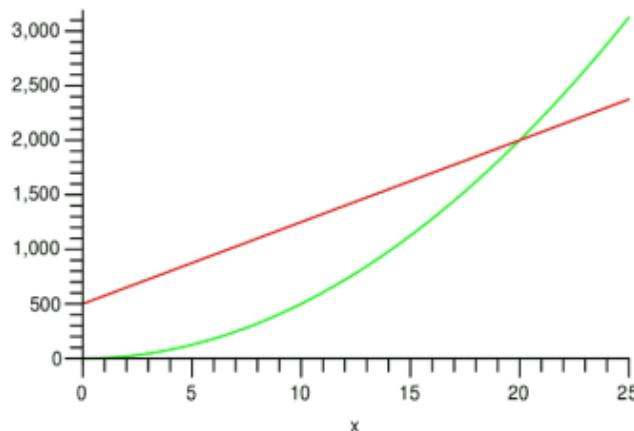
- constant complexity: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
$$\begin{aligned} \log(2n) &= \log 2 + \log n \\ &= 1 + \log n \end{aligned}$$
- linear complexity: $T(n) = cn$, $T(2n) = 2T(n)$.
- $\Theta(n \log n)$: $T(n) = cn \log n$, $T(2n) = 2T(n) + 2cn$.
$$\begin{aligned} 2n \log(2n) &= 2n \log 2 + 2n \log n \\ &= 2n + 2n \log n \end{aligned}$$
- quadratic complexity: $T(n) = cn^2$, $T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$, $T(2n) = 8T(n)$.
- exponential complexity: $T(n) = c2^n$, $T(2n) = (T(n))^2/c$.

Complexity vs. Running Time

- Suppose that algorithms A_1 and A_2 both solve some specified problem.
- Suppose that the complexity of algorithm A_1 is *lower* than the complexity of algorithm A_2 . Then, for sufficiently large problem instances, A_1 will run *faster* than A_2 . However, for small problem instances, A_1 could be slower than A_2 .
- Now suppose that A_1 and A_2 have the *same complexity*. Then we *cannot determine* from this information which of A_1 or A_2 is faster; a more delicate analysis of the algorithms A_1 and A_2 is required.

Example

Suppose an algorithm A_1 with linear complexity has running time $T_{A_1}(n) = 75n + 500$ and an algorithm with quadratic complexity has running time $T_{A_2}(n) = 5n^2$. Then A_2 is faster when $n \leq 20$ (the *crossover point*). When $n > 20$, A_1 is faster.



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 complexity $O(n^3)$ and A_2 has complexity $O(n^2)$.
- The above statements are perfectly reasonable.
- Observe that we *cannot* conclude that A_2 is more efficient than A_1 in this situation! (Why not?)

Techniques for Order Notation

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

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

Then

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

The required limit can often be computed using *L'Hôpital's rule*. Note that this result gives *sufficient* (but not necessary) conditions for the stated conclusions to hold.

An Example

Compare the growth rates of $\log n$ and n^i (where $i > 0$ is a real number).

$$f(n) = \log n \quad g(n) = n^i \quad i > 0$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\log n}{n^i} \quad \underline{\text{take derivative}} \quad \lim_{n \rightarrow \infty} \frac{\frac{1}{n} * C}{in^{i-1}}$$

$$= \lim_{n \rightarrow \infty} \frac{C}{in^{i-1}}$$

Since \lim goes to 0,
 $f(n) \in O(g(n))$

$$\rightarrow 0$$

C is a constant

Example

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

$$f(n) = n(2 + \sin n\frac{\pi}{2})$$

$$g(n) = n$$

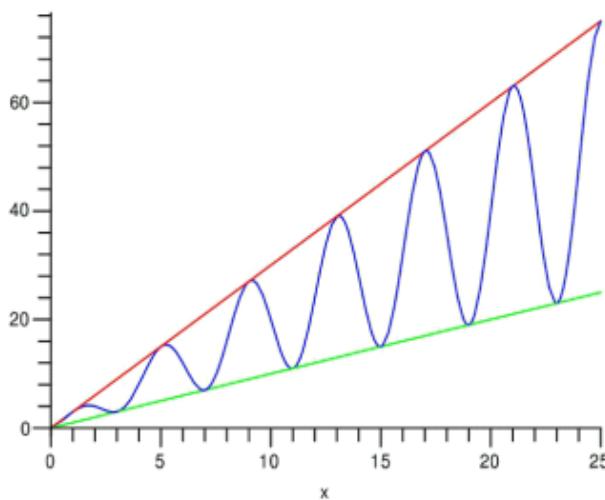
$$n \leq f(n) \leq 3n$$

$c_1 = 1$
 $c_2 = 3$ \Rightarrow follow the definition of Θ
 $n_0 = 1$

$$f(n) \in \Theta(g(n))$$

Example

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



Relationships between Order Notations

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

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

Algebra of Order Notations

“Maximum” rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$.

Then:

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

Transitivity: If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$.

Summation Formulae

$$\sum_{i=1}^n i^2 = 1 + 4 + 9 \dots \in \Theta(n^3)$$

Arithmetic sequence:

$$\sum_{i=1}^n i^3 = 1 + 8 + 27 \dots \in \Theta(n^4)$$

$$\sum_{i=0}^{n-1} (a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

Geometric sequence:

$$\therefore \sum_{i=1}^n i^k \in \Theta(n^{k+1})$$

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

Harmonic sequence:

$$H_n = \sum_{i=1}^n \frac{1}{i} \in \Theta(\log n)$$

More Formulae and Miscellaneous Math Facts

$$\bullet \sum_{i=1}^n i r^i = \frac{nr^{n+1}}{r-1} - \frac{r^{n+1} - r}{(r-1)^2}$$

$$\bullet \sum_{i=1}^{\infty} i^{-2} = \frac{\pi^2}{6}$$

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

$$\bullet \log_b a = \frac{1}{\log_a b}$$

$$\bullet \log_b a = \frac{\log_c a}{\log_c b}$$

$$\bullet a^{\log_b c} = c^{\log_b a}$$

$$\bullet n! \in \Theta(n^{n+1/2} e^{-n})$$

$$\bullet \log n! \in \Theta(n \log n)$$

Techniques for Algorithm Analysis

Two general strategies are as follows.

- Use Θ -bounds *throughout the analysis* and obtain a Θ -bound for the complexity of the algorithm.
- Prove a O -bound and a *matching* Ω -bound *separately* to get a Θ -bound. Sometimes this technique is easier because arguments for O -bounds may use simpler upper bounds (and arguments for Ω -bounds may use simpler lower bounds) than arguments for Θ -bounds do.

Techniques for Loop Analysis

- Identify *elementary operations* that require constant time (denoted $\Theta(1)$ time).
- The complexity of a loop is expressed as the *sum* of the complexities of each iteration of the loop.
- Analyze independent loops separately, and then *add* the results (use “maximum rules” and simplify whenever possible).
- If loops are nested, start with the innermost loop and proceed outwards. In general, this kind of analysis requires evaluation of *nested summations*.

Example of Loop Analysis

$$T(n) = C_1 + \sum_{i=1}^n \sum_{j=i}^n C_2$$

$$\begin{aligned} T(n) &= C_1 + \sum_{i=1}^n C_2 \times (n-i+1) && \text{let } k=n-i+1 \\ &= C_1 + C_2 \sum_{k=1}^n k && \begin{array}{l} 1 \leq i \leq n \\ -n \leq -i \leq -1 \\ 0 \leq n-i \leq n-1 \\ 1 \leq n-i+1 \leq n \end{array} \\ &= C_1 + C_2 \times \frac{n(n+1)}{2} \in \Theta(n^2) && k \end{aligned}$$

Test1(n)

1. $sum \leftarrow 0$
2. **for** $i \leftarrow 1$ to n **do**
3. **for** $j \leftarrow i$ to n **do**
4. $sum \leftarrow sum + (i - j)^2$
5. $sum \leftarrow sum^2$
6. **return** sum

$$A = \sum_{j=i}^n C_2$$

$$= C_2 * (n-i+1)$$

Example of Loop Analysis

Test2(A, n)

1. $max \leftarrow 0$
2. **for** $i \leftarrow 1$ to n **do**
3. **for** $j \leftarrow i$ to n **do**
4. $sum \leftarrow 0$
5. **for** $k \leftarrow i$ to j **do**
6. $sum \leftarrow A[k]$
7. **if** $sum > max$ **then**
8. $max \leftarrow sum$
9. **return** max

$$\begin{aligned}
 T(n) &= C_1 + \sum_{i=1}^n \sum_{j=i}^n \left(C_2 + \sum_{k=i}^j C_3 \right) & A = C_2 + C_3 * (j-i+1) \\
 &= C_1 + \sum_{i=1}^n \sum_{j=i}^n (C_2 + C_3 * (j-i+1)) & \text{let } k = j-i+1 \\
 &= C_1 + \sum_{i=1}^n \sum_{k=1}^{n-i+1} (C_2 + C_3 * k) & i \leq j \leq n \\
 &= C_1 + \sum_{i=1}^n \left(C_2(n-i+1) + C_3 \sum_{k=1}^{n-i+1} k \right) & 0 \leq j-i \leq n-i \\
 &= C_1 + \sum_{i=1}^n C_2(n-i+1) + \underbrace{\sum_{i=1}^n \left(\frac{C_3}{2} (n-i+1)(n-i+2) \right)}_{D} & B = C_3 * \frac{(n-i+1)(n-i+2)}{2} \\
 &\quad \underbrace{\sum_{i=1}^n C_2}_{C} & \text{let } t = n-i+1 \\
 &= C + D & C = \sum_{t=1}^n C_2 t = C_2 * \frac{n(n+1)}{2} \in \Theta(n^2) \\
 &= \Theta(n^2) + \Theta(n^3) \in \Theta(n^3) & D = \frac{C_3}{2} \sum_{q=1}^n q(q+1) \quad \text{let } q = n-i+1 \\
 &&= \frac{C_3}{2} \sum_{q=1}^n q^2 + \frac{C_3}{2} \sum_{q=1}^n q \\
 &&= \frac{C_3}{2} \frac{n(n+1)(2n+1)}{6} + \frac{C_3}{2} \frac{n(n+1)}{2} \\
 &&\in \Theta(n^3)
 \end{aligned}$$

Example of Loop Analysis

$$\begin{aligned}
 T(n) &= C_1 + \sum_{i=1}^n (C_2 + \log_i C_3) \\
 &= C_1 + C_2 n + C_3 \sum_{i=1}^n (\log_i) \\
 &\approx C_1 + C_2 n + C_3 C_4 n \log n \\
 &\in \Theta(n \log n)
 \end{aligned}$$

$$j = i \quad C_3$$

$$j = \frac{1}{2}i \quad C_3$$

$$j = \frac{1}{4}i \quad C_3$$

$$\begin{aligned}
 &\log 1 + \log 2 + \dots + \log n \\
 &\approx \log(1 \times 2 \times 3 \dots \times n)
 \end{aligned}$$

$$\approx \log(n!)$$

$$\in \Theta(n \log n)$$

$$\log n! \approx C_4 n \log n$$

Test3(n)

1. `sum` $\leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $j \leftarrow i$
4. **while** $j \geq 1$ **do**
5. $sum \leftarrow sum + i/j$
6. $j \leftarrow \lfloor j/2 \rfloor$
7. **return** `sum`

Design of 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. This can be done in time $\Theta(n)$.

MergeSort

```
MergeSort(A, n)
1.  if n = 1 then
2.      S ← A
3.  else
4.      nL ← ⌈ $\frac{n}{2}$ ⌉
5.      nR ← ⌊ $\frac{n}{2}$ ⌋
6.      AL ← [A[1], ..., A[nL]]
7.      AR ← [A[nL + 1], ..., A[n]]
8.      SL ← MergeSort(AL, nL)
9.      SR ← MergeSort(AR, nR)
10.     S ← Merge(SL, nL, SR, nR)
11.    return S
```

Analysis of MergeSort

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

- Step 1 takes time $\Theta(n)$
- Step 2 takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
- Step 3 takes time $\Theta(n)$

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

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

Analysis of MergeSort

- The mergesort recurrence is

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

- It is simpler to consider the following *exact recurrence*, with unspecified constant factors c and d replacing Θ 's:

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

Analysis of MergeSort

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

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

- 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.

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
 &= 4T\left(\frac{n}{4}\right) + cn + cn \\
 &= 8T\left(\frac{n}{8}\right) + cn + cn
 \end{aligned}$$

after $\log(n)$ steps $\log n$ times

$$\begin{aligned}
 T(n) &= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \overbrace{cn + cn + cn \dots}^{\log n \text{ times}} \\
 &= \underbrace{n * T(1)}_{\Theta(n)} + \underbrace{\log n * cn}_{\Theta(n/\log n)}
 \end{aligned}$$

$$T(n) \in \Theta(n/\log n)$$