Module 1: Introduction and Asymptotic Analysis

CS 240 - Data Structures and Data Management

Sajed Haque   Veronika Irvine   Taylor Smith
Based on lecture notes by many previous cs240 instructors

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

Spring 2017

Course Information

- Instructors
  - Veronika Irvine, DC 2124
    `virvine [at] uwaterloo.ca`
  - Taylor Smith, DC 3117
    `tj2smith [at] uwaterloo.ca`
  - Sajed Haque, DC 3117
    `s24haque [at] uwaterloo.ca`
- Lectures (Tuesday, Thursday)
  - 001(R) 10:00-11:20 in MC 2017 (Irvine)
  - 002(R) 10:00-11:20 in MC 2065 (Smith)
  - 003(R) 14:30-15:50 in MC 1056 (Haque)
- Office hours, phone numbers etc.
  - See web page

Course Information

Instructional Support

- Coordinator: Karen Anderson (MC 4010)
  - `kaanders [at] uwaterloo.ca`
- Assistant: Zach Frenette (IA), Wei Sun (ISA), Alex Duong (ISA), Peilin Wang (ISA)
  - `cs240 [at] uwaterloo.ca`
- Office hours
  - See web page

Tutorials (Mondays):

- 101 09:30-10:20M in MC 2054
- 102 10:30-11:20M in MC 2035
- 103 08:30-09:20M in MC 2035
- 104 12:30-01:20M in MC 1056

Tutorial next week on LaTeX
Assignment 0 to learn LaTeX (6 bonus marks on assignment 1)

Course Information

- Course Webpage
  `http://www.student.cs.uwaterloo.ca/~cs240/s17/`
  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/spring2017/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@uwaterloo.ca

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

Mark Breakdown

- Final 50%
- Midterm 25%
  - Tuesday June 20, 4:30-6:20pm
- Assignments 25%
  - 5 assignments each worth 5%
  - Approximately every 2 weeks
  - Due on Wednesdays at 5:00pm
  - 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?

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.

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

- 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: $Size(I)$ 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 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

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 $\mathcal{A}$.

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:

- 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 $0 \le f(n) \le c\,g(n)$ for all $n \ge n_0$.

Ω-notation: $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$.

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

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

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

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:

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

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

Example of Order Notation

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

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.

Average-case complexity of an algorithm

Worst-case complexity of an algorithm

Complexity of Algorithms

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

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

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

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

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*),
- $\Theta(n \log^k n)$, 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., $n \to 2n$).

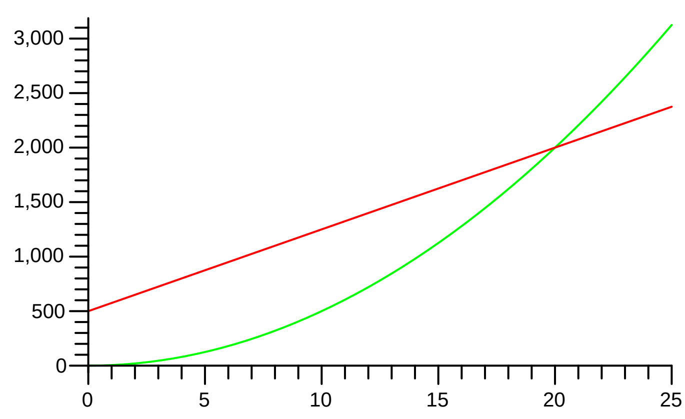
- constant complexity: $T(n) = c$, $T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$, $T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$, $T(2n) = 2T(n)$.
- $\Theta(n \log n)$: $T(n) = cn \log n$, $T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = cn^2$, $T(2n) = 4T(n)$.
- cubic complexity: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

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

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

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).

Example

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

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

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

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

Arithmetic sequence:

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

Geometric sequence:

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

Harmonic sequence:

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

More Formulae and Miscellaneous Math Facts

- $\sum_{i=1}^{n} i\, r^i = \dfrac{nr^{n+1}}{r-1} - \dfrac{r^{n+1}-r}{(r-1)^2}$
- $\sum_{i=1}^{\infty} i^{-2} = \frac{\pi^2}{6}$
- for $k \geq 0$, $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$
- $\log_b a = \frac{1}{\log_a b}$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $a^{\log_b c} = c^{\log_b a}$
- $n! \in \Theta\left(n^{n+1/2}e^{-n}\right)$
- $\log n! \in \Theta(n \log n)$

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

Test1(n)
$sum \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow i$ **to** n **do**
    $sum \leftarrow sum + (i-j)^2$
    $sum \leftarrow sum^2$
return sum

Example of Loop Analysis

Test2(A, n)
$max \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow i$ **to** n **do**
    $sum \leftarrow 0$
    for $k \leftarrow i$ **to** j **do**
      $sum \leftarrow A[k]$
      if $sum > max$ **then**
        $max \leftarrow sum$
return max

Example of Loop Analysis

```
Test3(n)
    sum ← 0
    for i ← 1 to n do
        j ← i
        while j ≥ 1 do
            sum ← sum + i/j
            j ← ⌊j/2⌋
    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)
    if n = 1 then
        S ← A
    else
        n_L ← ⌈n/2⌉
        n_R ← ⌊n/2⌋
        A_L ← [A[1], ..., A[n_L]]
        A_R ← [A[n_L + 1], ..., A[n]]
        S_L ← MergeSort(A_L, n_L)
        S_R ← MergeSort(A_R, n_R)
        S ← Merge(S_L, n_L, S_R, n_R)
    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 $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$
- 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

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

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

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

Analysis of MergeSort

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

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

- 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.