CS240 Notes

Jacky Zhao

June 27, 2020

1 Course Objectives

1.1 Overview

What is this course about?

- When first learning to program, we emphasize correctness
- Starting with this course, we will also be converned with efficiency
- We will 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
- We will consider various abstract data types (ADTs) and how to implemnet 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-O, etc.)

Course Topics:

- big-O 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

Required knowledge:

- arrays, linked lists (3.2- 3.4)
- strings (3.6)
- stacks, queues (4.2 - 4.6)
- abstract data types (4 - intro, 4.1, 4.8 - 4.9)
- recursie 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

1.2 General Terminologies

The core of CS240 is:

Given problem Π, design algorithm A that solves it, and analyze its efficiency

So what is a problem, an algorithms, and how do you quantify efficiency?

Problem

- Given a problem instance, carry out a particular computational task
- Ex. Sorting is a problem

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

Algorithm

- 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

- an implementation of an algorithm using a specified computer language

Pseudocode

- a method of communicating an algorithm to another person
- in contrast, a program is a method of communicating an algorithm to a computer
- General rules of pseudocode:
  - omits obvious details (variable declarations)
  - has limited, if any, error detection
  - sometimes uses English descriptions
  - sometimes usus mathematical notation

1.3 Algorithms and programs

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

2 Analysis of Algorithms I

- **Running Time:** In this course, we are primarily concerned with the amount of time a program takes to run
- **Space:** We also may be interested in the amount of memory the program requires
- The amount of time and/or memory required by a program will depend on $Size(I)$, the size of the given problem instance I

2.1 Running time of Algorithms/Programs

Option 1: Experimental Studies

- Write a program implementing the algorithm
- Run the programs with various sizes of input and measure the actual running time
- Plot/compare the results

Shortcomings:

- Implementation may be complicated/costly
- Timings are affected by many factors: hardware, software environment, and human factors
- 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

Which means, we need some simplifications

We will develop several aspects of algorithm analysis:

- Algorithms are presented in structured high-level pseudocode, which is language-independent
- Analysis of algorithms is based on an idealized computer model
- The efficiency of an algorithm (with respect to time) is measure din terms of its growth rate, aka the complexity of the algorithm

2.2 Simplifications of running time

Overcome dependency on hardware/software

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

Random Access Machine (RAM) model:

- it 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

Simplify Comparisons

- Example: Compare $100n$ with $10n^2$
- Idea: Use order notation
- Informally: ignore constants and lower order terms

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

2.3 Asymptotic 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 $g(n) = 5n^2$, choose $c = 1$ and $n_0 = 20$ can prove $f(n) \in O(g(n))$
- Note: the absolute value signs int eh definition are irrelevant for analysis of run-time or space, but are useful in other application sof asymptotic notation

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:

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

Note that all choices of c and n_0 will work. **Solution:**

Choose $n_0 = 1$.

$$\begin{aligned} n_0 \leq n \to 1 \leq n \to 1 \leq n^2 \to 11 \leq 11n^2 \\ n_0 \leq n \to 1 \leq n \to n \leq n^2 \to 3n \leq 3n^2 \\ \text{We also have: } 2n^2 \leq 2n^2 \end{aligned}$$

So we have:

$$2n^2 + 3n + 11 \leq 2n^2 + 3n^2 + 11n^2 \leq 16n^2$$

So let $c = 16$ and $n_0 = 1$, and we have $|f(n)| < c|g(n)|$ for all $n \geq n_0$.
Thus $2n^2 + 3n + 11 \in O(n^2)$. $\square$

We want a **tight** asymptotic bound. So we have:
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 $n \geq n_0$

Notice:

$$f(n) \in \Theta(g(n)) \longleftrightarrow f(n) \in O(g(n)) \textbf{ and } f(n) \in \Omega(g(n))$$

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

Solution:
Let $n_0 = 20$. We find c.

$$n_0 = 20 \leq n \rightarrow 20n \leq n^2 \rightarrow 5n \leq \frac{1}{4}n^2 \rightarrow 0 \leq \frac{1}{4}n^2 - 5n$$
$$\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \underbrace{\frac{1}{4}n^2 - 5n}_{\geq 0} \geq \frac{1}{4}n^2$$

Since $\frac{1}{2}n^2 - 5n \geq \frac{1}{4}n^2$, we choose $c = \frac{1}{4}$ and we have $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$. $\square$

Quick Summary:

- $O \leftrightarrow$ asymptotically not bigger
- $\Omega \leftrightarrow$ asymptotically not smaller
- $\Theta \leftrightarrow$ asymptotically the same

We have $f(n) = 2n^2 + 3n + 11 \in \Theta(n^2)$

- How do we express that $f(n)$ is **asymptotically strictly smaller** than n^3?

o-notation

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

The o and ω notations are rarely proved from first principles.

2.4 Relationships between Order Notations

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

- $f(n) \in o(g(n)) \rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \rightarrow f(n) \notin O(g(n))$

2.5 Algebra of Order Notations

Identity rule

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

Maximum rules

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

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

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 $f(n) \in \Omega(h(n))$

2.6 Techniques for Order Notation