CS 305
Design and Analysis of Algorithms

09 / 08 / 2022

Instructor: Michael Eckmann

Today's Topics

- Introduce myself
- Start class topics
  - What is an algorithm? What is a computational problem?
  - Pseudocode
  - Analyzing an algorithm
  - Asymptotics
- Reading assignment

Who is your instructor?

- I'm Mike Eckmann, an Associate Professor in the Computer Science Dept. I have taught at Skidmore since 2004. Before coming to Skidmore I was at Lehigh University in PA.
- I studied Mathematics and Computer Engineering and Computer Science all at Lehigh University.
- I was employed as a programmer (systems analyst) for eight years.
- At Skidmore in addition to several required course in CS including CS305 twice, I have taught these elective courses: Computer Graphics, Computer Vision, and Digital Image Processing, among a few others.

Course webpage and policies

- Course webpage is here:
- meckmann.domains.skidmore.edu/2022Fall/cs305
- I sent an email with this link and a short questionnaire
- Syllabus, Class notes (slides only), and Assignments can be found there.
- Assessment: 4-7 HW's, Midterm Exam, Final Exam.
- Class participation and improvement throughout the semester will have a positive influence on your final grade.
- Covid-related --- mask policy, etc.

Algorithm / Problem

- By the end of today's class, you'll get some idea of what this course is about.
- What's an algorithm?
- What's a computational problem?
- What's an instance of a problem?
- What kinds of things do we want to analyze about an algorithm?

Algorithm / Problem

- What's an algorithm?
  - Finite sequence of computational steps that takes input and produces an output
- What's a computational problem?
  - Specified input / output relation (i.e. given certain input, produces certain output)
- What's an instance of a problem?
  - Some particular input
- What kinds of things do we want to analyze about an algorithm?
  - Efficiency (mostly speed, but also space)
  - Correctness

Algorithm / Problem

- Example on board:
  - Sorting problem
  - An algorithm that solves the sorting problem
  - Instance of the sorting problem

Correct Algorithm

- An algorithm is **correct** when if for every instance it halts with correct output.
- A correct algorithm **solves** the computational problem.

FindMax problem and algorithm

- Given a list of numbers, find the maximum number in the list.
- Let's write pseudocode on the board for an algorithm that solves this.
  - Note indentation to indicate blocks
  - Left arrow assignment
  - Among other pseudocode conventions

FindMax problem and algorithm

- **Loop invariants** for determining correctness
- Must show three things about a loop invariant
  - Initialization: some property is true before loop
  - Maintenance: that property is true before each loop iteration
  - Termination: algorithm is correct if loop terminates and the reason the loop terminated and the loop invariant allows us to show that the algorithm is correct
- What's the loop invariant for findMax?

FindMax problem and algorithm

- What is true before the loop and before each subsequent iteration?
- What's the loop invariant for findMax?
  - maxSoFar is largest value in sub list[1 .. i-1]
- Termination: when i reaches n+1 the loop terminates. So the maxSoFar is the largest value in the sub list[1 .. n]
- Together these imply that the maxSoFar is the largest value in the entire list

FindMax problem and algorithm

- Let's analyze it for efficiency (speed).
  - Depends on what?

FindMax problem and algorithm

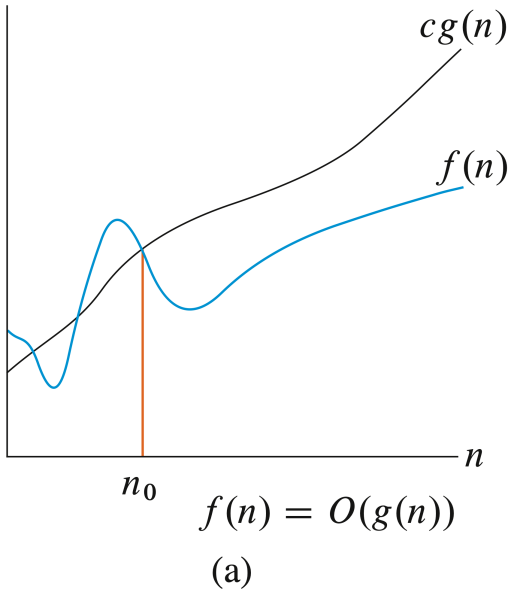
- Let's analyze it for efficiency (speed).
  - Consider
    - best case (best instance)
    - worst case (worst instance)
    - average case also possible – need to know probability distribution of the inputs
- Abstract away from what machine it's being run on.
  - Assume (different) constant time for each line
  - Some lines execute a different number of times
  - let's compute on the board

FindMax problem and algorithm

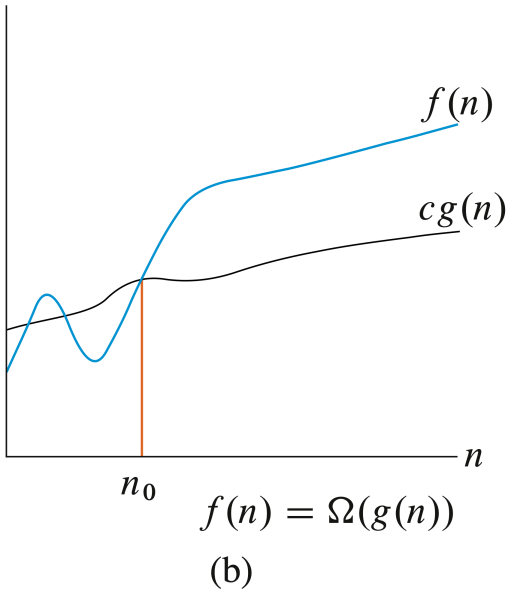
- Let's analyze it for efficiency (speed).
  - we'll focus mostly on worst case running time in this course because
    - Gives an upper bound on running time
    - Worst case can happen often for some algorithms
    - Average is often just as bad
    - Worst case is easier to determine than average

Asymptotic notation

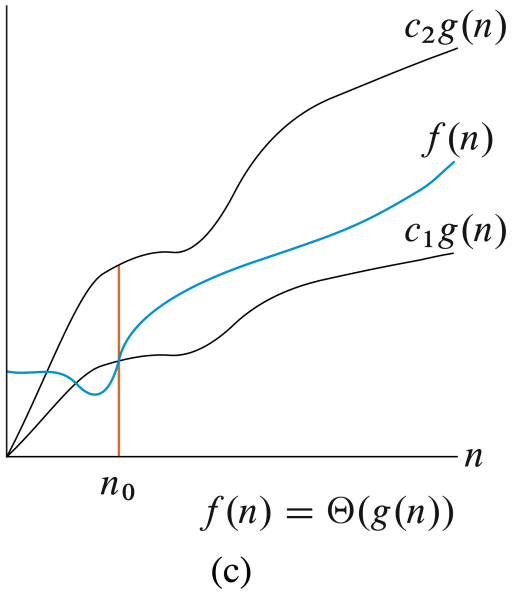
- big O
  - When we say f(n) is O(g(n))
  - g(n) is an upper bound on f(n)
  - Let's look at figure 3.2 in text



(a)



(b)



(c)

Asymptotic notation

- The running time of findMax is O(n) because n is an upper bound on a*n + b (where a and b are constants (not dependent on n)). Note: n is also a lower bound on a*n + b. Together those make n a tight bound on a*n + b.
- Let's prove this using the definition of big O.

- Note:
- a*n + b is also $O(n^2)$, O(n*log n), O(n!), $O(2^n)$
  - Because n^2, n*log n, n!, 2^n are all upper bounds on a*n + b
- a*n + b is NOT O(1), O(log n)

Asymptotic notation

- A reminder of the meaning of "for all" or "for any" vs. "there exists"
- Definitions of Big O, Big Omega and Big Theta on the board.
- Note that these define sets of functions, but we typically say "is" instead of "is in the set"
- Because we cannot do better than n for findMax, the overall (including best and worst cases) running time of findMax is Big Theta (n) – it is an asymptotically **tight bound**
  - Notice that we must compare each element to the maxSoFar and since there are n elements we cannot do better than n-1 compares

Asymptotic notation

- Why use asymptotic behavior for efficiency?
  - In general ...
  - Ignores small inputs (small values of n) because we may be able to create special purpose algorithms if the input is expected to be small
  - Ignore constants
    - Faster machines can combat constant factors
    - Faster machines cannot combat poor asymptotics

Asymptotic notation

- Analyze space of findMax
  - Inputs are the list and the size of the list
  - Additional space for maxSoFar and i
  - Only care about the additional space required beyond the inputs for space analysis
  - Space complexity of findMax is Big Theta (1)

Asymptotic notation

- Let's use the definitions of O and Big Omega and Big Theta in an example to determine the asymptotics of some particular function (other than the simple one for findMax)

Asymptotic notation

- Idea of tight bound vs. not tight bound
- e.g.
- $2*n^2 = O(n^2)$ is asymptotically tight
- $2*n = O(n^2)$ is NOT asymptotically tight (but it is correct to say)
- So, O may or may not be asymptotically tight
- One can use little o to denote an upper bound that is NOT asymptotically tight
- Definition on the board
- Note: $2*n = o(n^2)$ BUT $2*n^2 != o(n^2)$

Asymptotic notation

- Little omega is the corresponding not asymptotically tight bound for lower bounds
- Definition on the board

Reading Homework

- Before Monday's class you should read:
  - in chapter 1 read pages 5-15
  - in chapter 2 read pages 17-33
  - in chapter 3 read pages 49-62

Syllabus

- Course overview
  - A study of techniques used to design algorithms for complex computational problems that are efficient in terms of time and memory required during execution. Students will also learn the techniques used to evaluate an algorithm's efficiency. Topics include advanced sorting techniques, advanced data structures, dynamic programming, greedy algorithms, amortized analysis, graph algorithms, network flow algorithms, and linear programming if time permits.

Syllabus

- Students will
  - Be exposed to a variety of existing algorithms and techniques to develop algorithms
  - Learn to design / develop an algorithm for a given computational problem
    - Maybe by noticing similarity to a learned algorithm/problem or existing technique
  - Learn how to analyze running time and space