CS 305
Design and Analysis of Algorithms

09 / 12 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions/Comments about anything from 1st class or reading assignment?
- Course overview slides that I didn't have time to do last time
- Review Big O, Big Omega, Big Theta
- Little o, little omega
- Loop invariants
- Recap on running time of FindMax, analyze space for FindMax
- Arithmetic series
- Selection sort analysis
- Insertion sort pseudocode and analysis

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 its running time

Asymptotic notation

- Which is upperbound, lowerbound, tight bound
- Big O
- Big Theta
- Big Omega

- Recall little o and little omega

$f(n)$ is $o(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

$f(n)$ is $O(g(n))$ iff $\exists\ c > 0,\ n_0 > 0\ \ni$
$0 \leq f(n) \leq c\, g(n)\ \not\forall\ n \geq n_0$

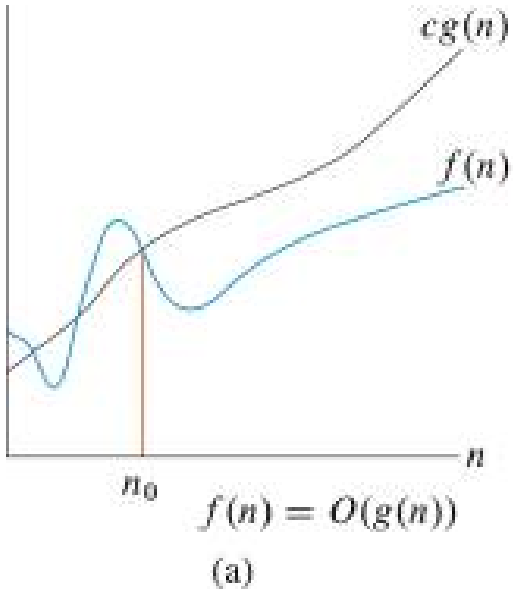
$f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ AND
$f(n)$ is $\Omega(g(n))$.

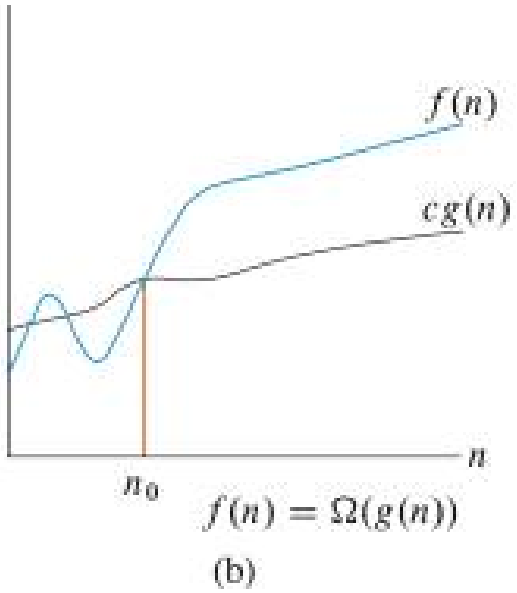
$f(n)$ is $\Omega(g(n))$ iff $\exists\ c > 0,\ n_0 > 0\ \ni$
$0 \leq c g(n) \leq f(n)\ \not\forall\ n \geq n_0$

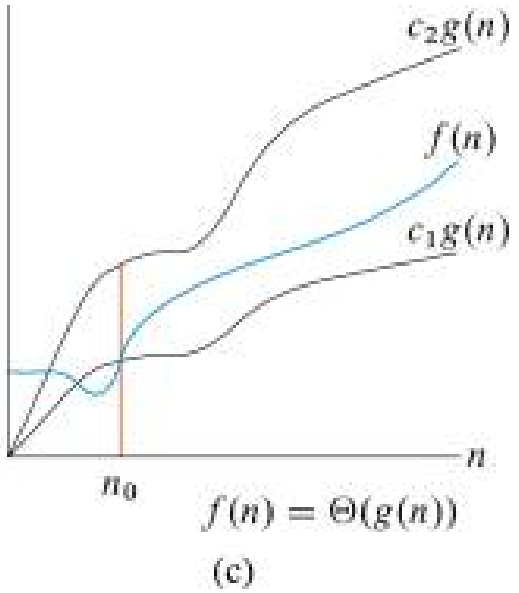
$f(n)$ is $\omega(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

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







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

- **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
- Let me rewrite FindMax on the board
- 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

- Last time we analyzed it for efficiency (speed).
  - Considered
    - 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

FindMax problem and algorithm

- 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

- 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)

Arithmetic Series

- Let me introduce Arithmetic Series on the board.
- This is one of 3 sums you should memorize.
- The other two being Geometric and Harmonic.
- See Appendix A of our text (pgs. 1141, 1142)

Arithmetic Series

Let's prove the sum is big $O(n^2)$ on the board

I leave you to prove that it is also Big Omega of n^2

Which means that it is Big Theta of n^2

Sorting algorithms

- Reminder of what Selection Sort and Insertion Sort do (online animation).
- Let's discuss Selection Sort.
- Keep current "first".
- Continually find min element index in list and swap with the current "first".
  - How long does 1 find min take?
  - How many times do we do find min?
  - Are there any best or worst case situtations? All the same?

Sorting Algorithms

- Insertion Sort pseudocode on the board with run time analysis.
- Are there any best or worst case situtations? All the same?