CS 209
Data Structures and Mathematical Foundations

02 / 26 / 2024

Instructor: Michael Eckmann

Today’s Topics

- Questions/Comments?
- More Big O, Big Theta, Big Omega discussion and examples

Asymptotic notation

- Recap
- If f(n) is O(g(n)) we say that g is what kind of bound on f?
- If f(n) is Omega(g(n)) we say that g is what kind of bound on f?
- If f(n) is Theta(g(n)) we say that g is what kind of bound on f?

Asymptotic notation

- f is Big O(g) – means g is an upperbound
- f is Big Theta(g) – means g is a tight bound
- f is Big Omega(g) – means g is a lowerbound

Asymptotic notation

- In a multiterm function (terms that are added together)
- Simply select the one that dominates (is O of) the other terms. In other words select the term that grows fastest (as n gets larger).
- Then remove the constant multiplier from this term.
- What remains is a simpler function that is the Big Theta of the original

Asymptotic notation

- Reminder of definitions of Big O, Big Omega and Big Theta on the next slide.
- Note that these define sets of functions, but we typically say “is” instead of “is in the set”

Asymptotic definitions

$f(n)$ is $O(g(n))$ if $\exists$ $c > 0$ and $n_0 > 0$ $\ni$

$$0 \le f(n) \le c\,g(n) \quad \forall\, n \ge n_0.$$

[we say that $g(n)$ is an UPPER BOUND on $f(n)$]

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

[$g(n)$ is a TIGHT BOUND on $f(n)$]

$f(n)$ is $\Omega(g(n))$ if $\exists$ $c > 0$, and $n_0 > 0$ $\ni$

$$0 \le c\,g(n) \le f(n) \quad \forall\, n \ge n_0.$$

[we say that $g(n)$ is a LOWER BOUND on $f(n)$]

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

Asymptotic notation

- Example: 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

Arithmetic Series

Let's prove the sum of all i's with i from 1 to n from earlier is big $O(n^2)$ (n squared is an upper bound on the sum)

Let's also prove that it is Big Omega of n^2 (that n^2 is a lower bound on the sum)

Together, they mean that it is asymptotially tight, that is Big Theta of n^2

Algorithm Analysis

- Functions in increasing order
  - constant functions (e.g. $f(n) = 10$)
  - logarithmic functions (e.g. $f(n) = \log(20n)$)
  - log squared (e.g. $f(n) = \log^2(7n)$)
  - linear functions (e.g. $f(n) = 3n - 9$)
  - N log N (e.g. $f(n) = 2n \log n$)
  - quadratic functions (e.g. $f(n) = 5n^2 + 3n$)
  - cubic functions (e.g. $f(n) = 3n^3 - 17n^2 + (4/7)n$)
  - exponential functions (e.g. $f(n) = 5^n$)
  - factorial functions (e.g. $f(n) = n!$)

Create a table

Let's create a table of best/worst/overall running times for a variety of algorithms that we've analyzed already, including the linked list algorithms, linear search, binary search, insertion sort, selection sort, find max

Note: average running time is also sometimes computed, but it is difficult to determine for some problems.

Algorithm Analysis

- Let's consider 1 problem and 3 ways to solve it (using 3 different algorithms) and we'll analyze the running times of each.
- The Maximum contiguous subsequence problem:
  - Given an integer sequence $A_1, A_2, \ldots, A_N$, find (and identify the sequence corresponding to) the maximum value of $\sum^{j}_{k=i} A_k$. The maximum contiguous subsequence sum is zero if all are negative. Therefore, an empty sequence may be maximum.
- Example input: { -2, **11, -4, 13**, -5, 2 } the answer is 20 and the sequence is { 11, -4, 13 }
- Another: { 1, -3, **4, -2, -1, 6** } the answer is 7, the sequence is { 4, -2, -1, 6 }

Algorithm Analysis

- The simplest is an exhaustive search (brute force algorithm.)
  - that is, simply consider every possible subsequence and compute its sum, keep doing this and save the greatest
  - so, we set the maxSum = 0 (b/c it is at least this big) and we start at the first element and consider every subsequence that begins with the first element and sum them up ... if any has a sum larger than maxSum, save this ...
  - then start at second element and do the same ... and so on until start at last element
- Advantages to this: easy to implement, easy to understand
- Disadvantages to this: slow
- Let's examine the algorithm.
  - decide what is a good thing to count
  - count that operation (in terms of the input size)