THE GROWTH OF FUNCTIONS

CS340

Analyzing algorithms

- We have looked at proving the correctness of an algorithm.
- We want to predict the resources that the algorithm requires.
- This is usually running time.
- **Asymptotic analysis**
    - Comparison of functions as inputs approach infinity.
- **Asymptotic efficiency**
    - How the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

Efficiency and Complexity

- How to calculate: “**How long does a program take?**”
- The same algorithm may produce different run times
  - The same program running on different machines will take longer on the slower machine!
  - Different programming languages may take different times
- We need a machine-independent measure

Time complexity

- Each programming statement has a cost
  - **Adding 2 numbers** uses a certain number of processing cycles
  - **Making a comparison** uses some number of processing cycles
- Each programming statement is run **some number of times**
- The **running time** of the algorithm is the sum of running times for each statement executed
  - A statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.

Total Cost = cost x times

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

- t_j denotes the number of times the **while** loop test in line 5 is executed for that value of j.
- The test for a loop is executed one more time than the loop.

Running time for Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Running time for Insertion Sort

- Best case: the **while** loop is never run

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

= *an* + b

= *linear* time based on size of input. Size of input = n.

Running time for Insertion Sort

- Worst case: the **while** loop runs the maximum number of times every time.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

- = $an^2 + bn + c$
- = quadratic time based on size of input

Time complexity

- We will focus on the worst case
  - The worst case is an **upper bound** on the total time
    - What did the best case of insertion sort tell us?
- Why care about the worst case?
  - The worst case *can be* roughly as bad as the average case
    - What is the average case for insertion sort?
    - It is often difficult to figure out what the average case is.
  - For critical applications, you want a **guarantee** on the running time even when you know nothing about the input.

One more abstraction

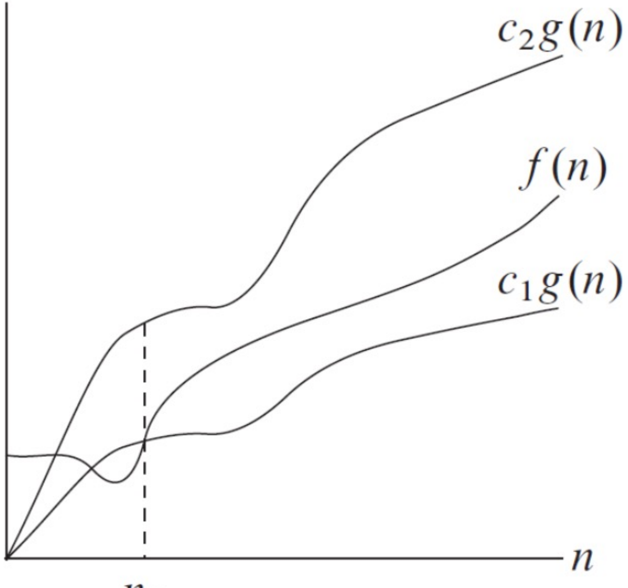
- It is the rate (or order) of growth we are interested in
- We can simplify to the order of n, where n is size of input

  - $an + b \rightarrow \Theta(n)$ (pronounced theta of n) = linear time
  - $an^2 + bn + c \rightarrow \Theta(n^2)$ (pronounced theta of n-squared)

Theta Notation

$\Theta(g(n))$ = { f(n): there exist positive constants c_1, c_2, and n_0 such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}



$f(n) = \Theta(g(n))$

O-Notation

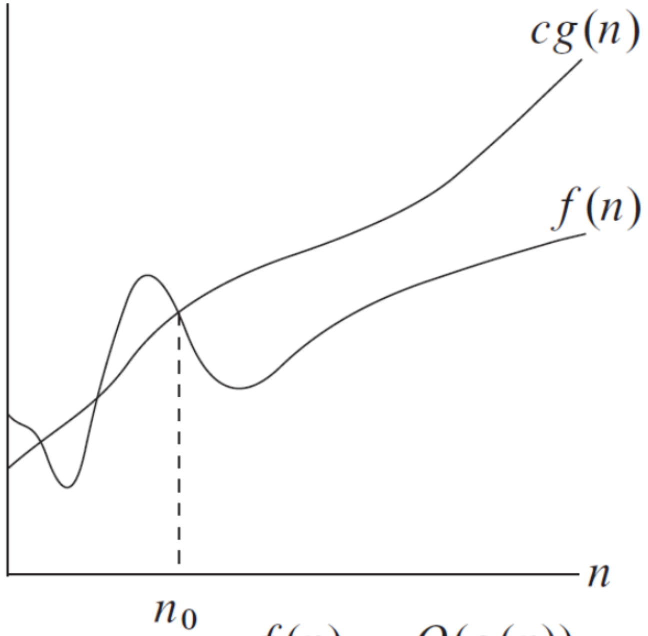
$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

$\Theta(g(n)) \subseteq O(g(n))$

Θ-notation is a **stronger** notion

than O-notation.

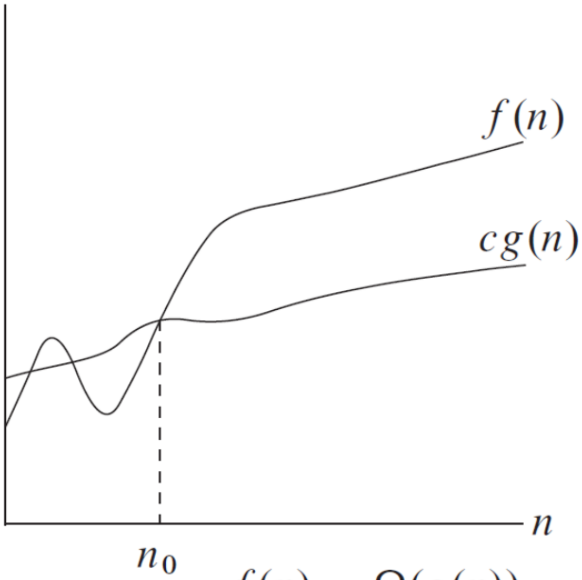


$f(n) = O(g(n))$

Omega-Notation

$\Omega(g(n))$ = { f(n): there exist positive constants c and n_0 such that

$0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$}



$f(n) = \Omega(g(n))$

Ω –notation provides an ***asymptotic lower bound***.