CSE 310
Data Structures and Algorithms

Asymptotic Notations

- Time complexity of an algorithm revisited
  - Worst-case running time
  - Asymptotic running time

- Asymptotic notations

- Review of commonly-used functions & notations

Time complexity of an algorithm

- We saw the analysis of the Insertion Sort algorithm: its best-case and worst-case time complexity.

- Sometimes we want to know the lower bound on the running time: the best-case complexity

- We often care more about the worst-case complexity of an algorithm.

  - The worst-case running time is an ***upper bound*** on running time of an algorithm on ***any*** input.

  - For some algorithms, the worst case occurs fairly often (e.g., searching for absent data in a database)

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

- The "average case" is often roughly as bad as the worst case.
  - Look at the Insertion Sort algorithm again

    "On average", we check half of the sub-array $A[1\ldots j-1]$
    → $t_j = j/2$ ("on average")
    → Average-case running time is a quadratic function on n
    -- the same as the worst case

  - Further, often the "average case" is difficult to compute

- So, unless stated explicitly otherwise, when we talk about "time complexity", we mean the worse case by default.

- Also, we care more about the complexity when *n* becomes very huge ➔ asymptotic complexity
  - To further simplify the analysis, we are concerned with only the *order of growth* rather than the exact running time
    - E.g., $3n^2 + 2n + 9$ ➔ $3n^2$ ➔ n^2
- We will define some formal notations for such analysis
  - O, Ω, Θ-notations

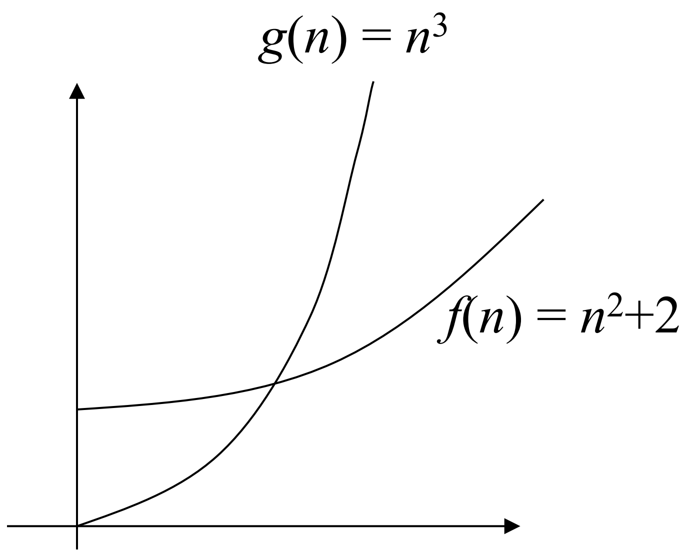
O-notation

- This is used to denote the asymptotic **upper bound (to within a constant factor)** on a function, in particular on the running time of an algorithm.
  - For a given $g(n)$, $O(g(n))$ is the set of functions

$$O(g(n)) = \{f(n)\text{: there exist positive constants } c, n_0 \text{ such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0\}.$$

- While it is a little abuse of notation, we often write something like $f(n) = O(n^2)$, or $t(n) = O(g(n))$, *etc*.
  - $f(n) = O(n^2)$ means $f(n)$ is a member of the set $O(n^2)$
    - Or simply $f(n)$ is upper bounded by n^2.
  - $t(n) = O(g(n))$ means $t(n)$ is a member of the set $O(g(n))$
    - Or simply $t(n)$ is upper-bounded by $g(n)$.

O-notation illustrated



$f(n) = O(g(n)) = O(n^3)$

Questions:

Can we write $f(n) = O(n^4)$?

Can we write $f(n) = O(n^2)$?

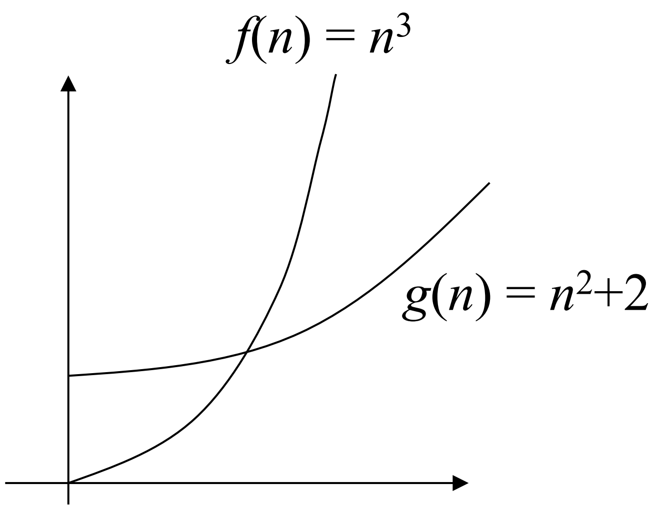
Ω-notation

- This is used to denote the asymptotic **lower bound (to within a constant factor)** on a function, in particular on the running time of an algorithm.
  - For a given $g(n)$, $\Omega(g(n))$ is the set of functions

$$\Omega(g(n)) = \{f(n)\text{: there exist positive constants } c,\ n_0 \text{ such that } 0 \le c\, g(n) \le f(n) \text{ for all } n \geq n_0\}.$$

- Similar to before, we often write things like $f(n) = \Omega(n^2)$, or $t(n) = \Omega(g(n))$, etc.
  - $f(n) = \Omega(n^2)$ means $f(n)$ is a member of the set $\Omega(n^2)$
    - Or simply $f(n)$ is lower-bounded by n^2
  - $t(n) = \Omega(g(n))$ means $t(n)$ is a member of the set $\Omega(g(n))$
    - Or simply $t(n)$ is lower-bounded by $g(n)$

Ω-notation illustrated



$f(n) = \Omega(g(n)) = \Omega(n^2)$

Questions:

Can we write $f(n) = \Omega(n^3)$?

Can we write $f(n) = \Omega(n)$?

Yes, but not very practically useful in the latter case

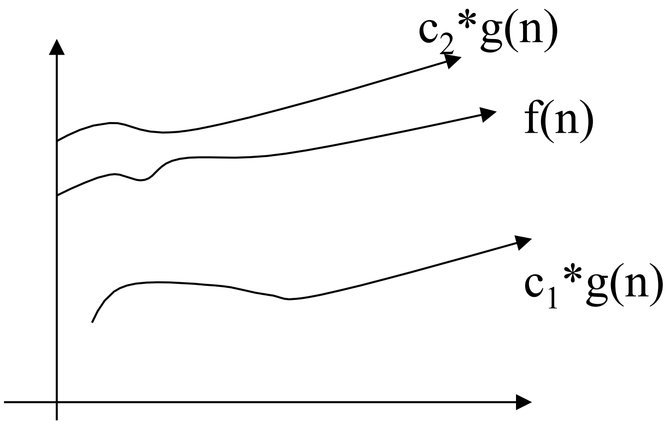
Θ-notation

- This is used to denote asymptotic tight **bound (to within a constant factor)** on a function, in particular on the running time of an algorithm.
  - For a given $g(n)$, $\Theta(g(n))$ is the set of functions

$$\Theta(g(n)) = \{f(n)\text{: there exist positive constants } c_1, c_2, n_0 \text{ such that } 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \geq n_0\}.$$

- Similar to before, we often write things like $f(n) = \Theta(n^2)$, or $t(n) = \Theta(g(n))$, etc.
  - $f(n) = \Theta(n^2)$ means $f(n)$ is a member of the set $\Theta(n^2)$
    - Or simply $f(n)$ has the same order of growth as n^2
  - $t(n) = \Theta(g(n))$ means $t(n)$ is a member of the set $\Theta(g(n))$
    - Or simply $t(n)$ has the same order of growth as $g(n)$

Θ-notation illustrated



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

<u>Theorem 3.1</u>

$f(n)$ is $\Theta(g(n))$ if and only if (iff)

$$f(n) = O(g(n)) \text{ and}$$
$$f(n) = \Omega(g(n))$$

Usually, people use the O-notation even when we have a tight bound.

Some helpful tricks for O, Ω, Θ

- Assume the following:

$$f(\mathrm{n}) = O(s(n))$$

$$g(n) = O(r(n))$$

Then

$$c{*}f(n) = O(s(n)) \qquad \text{(for any constant } c>0\text{)}$$

$$f(n)+c = O(s(n)) \qquad \text{(for any constant } c\text{)}$$

$$f(n) + g(n) = O(s(n) + r(n))$$

$$f(n){*}g(n) = O(s(n){*}r(n))$$

Some helpful tricks (cont'd)

- Consider only the leading term of a formula (drop lower-order terms).

ex: $f(n) = n^3 + 2n^2 - n + 5 = O(n^3)$
$= \Omega(n^3)$
$= \Theta(n^3)$
(However, $f(n) = \Omega(n^2)$ but $f(n) \neq O(n^2)$
$f(n) = O(n^9)$ but $f(n) \neq \Omega(n^9)$)

- Ignore the leading term's coefficient

ex: $f(n) = 3n^3 - 2n^2 = O(n^3)$
$= \Omega(n^3)$ (therefore also $\Theta(\mathrm{n}^3)$)
$f(n) = 3*2^n + \log n = O(2^n)$
$= \Omega(2^n)$ (therefore also $\Theta(2^n)$)

Efficiency of Algorithms

- So what can we do with those fancy notations?
  - They help us to analyze and compare algorithms

- We consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.
  - This evaluation may not be true for small inputs, but is true for large enough inputs ➔ Asymptotic performance.

Example: An algorithm with $\Theta(n^2)$ worst-case running time will run more quickly than a $\Theta(n^3)$ worst-case running time algorithm, for large enough inputs.

- Usually we say that an algorithm is efficient if it runs in polynomial time (or less) --- non-polynomial e^n, 2^n, n^n, …

Optimal Algorithms

- An optimal algorithm for solving a certain problem is one that has minimum asymptotic running time among all possible algorithms for solving the problem.

usually not easy to determine

- Note: an optimal algorithm defined as such is not necessarily one that finds an *optimal solution* to the given problem; similar definitions can be made for space optimality.

Example:

Merge-Sort, Heap-Sort are optimal algorithms for the problem of sorting by comparison, since their running time is $O(n \log n)$ and we can show a lower bound of $\Omega(n \log n)$ for sorting by comparison.

Common Functions & Properties

- Refer to Section 3.2 for reviewing the following concepts, definitions, or functions:
  - Monotonicity
  - Floors and ceilings: $\lfloor x \rfloor$, $\lceil x \rceil$
  - Modular arithmetic: $a \bmod n = a - n \lfloor a/n \rfloor$
  - Polynomials
  - Exponentials
  - Logarithms
  - Factorials: $n! = 1*2*3*\ldots*n$
    - Stirling's approximation

Summary & Reading Assignment

- The time complexity of an algorithm depends on
  - Input size (e.g., 6 elements v.s. 6000 elements)
  - Input itself (e.g., partially sorted or not.)
- Analysis of an algorithm: best-case, worst-case, average-case
- Performance bounds: upper, lower, tight
- Analysis via asymptotic notations: **O, Ω, Θ**

Again: usually, people use the O-notation even when we have a tight bound ➔ Make sure you at least know what t(n) = O(g(n)) means.

- Read Chapter 3 to review topics covered in this set of slides (except the o-notation (small o) and ω-notation (small ω)).

- To prepare for next week, read Section 2.3, Introduction of Chapter 4 (before Sect. 4.1), Sections 4.3, 4.4, 4.5.