CS241 – Analysis of Algorithms Spring 2019

- **Prerequisites**: CMPU102 and CMPU145.
- **Lectures**: M & W @ 9:00 to 10:15 am in SP 105.
- **Textbook**: *Introduction to Algorithms (3rd Edition)*, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (CLRS).
- **Course web page:** https://www.cs.vassar.edu/~cs241/spr19/

  All information on course will be posted on the page above.

Course Assignments/Announcements

- No solutions will be accepted without an excused absence after graded solutions are handed back.
- Check your e-mail frequently for course announcements.

Algorithms

What is an algorithm?

- For the purposes of this class, an *algorithm* is a computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output, and eventually terminates.

Algorithmic Problem

What is an algorithmic problem?

- An algorithmic *problem* is the complete set of possible input *instances* the algorithm may work on and the desired output from each input instance.

Measures of Complexity

What metrics of an algorithm are considered when comparing algorithm complexity?

Time

Space

Number of messages (distributed algorithms)

Power consumption (ad hoc networks)

Algorithm Time Efficiency

Observation: Most algorithms that do anything with their input run longer on larger inputs.

Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

Analyzing Algorithms

Goal: to predict the number of steps executed by an algorithm in a machine- and language-independent way using 2 simplifying assumptions:

Analyzing Algorithms

Simplifying assumption 1 –

We use the RAM model of computation: Single processor with *sequential instruction execution* (no parallel computation).

Running time of algorithm can be described with a mathematical function of the input size n.

Analyzing Algorithms

Simplifying assumption 2 –

We use *asymptotic analysis* of *worst-case* complexity. The asymptotic behavior of a function f(n) refers to the growth of f(n) as n gets very large.

Comparing the asymptotic running time of algorithms lets us ignore constant multiples and lower-order terms in the equation describing the running time.

Different ways to measure algorithm time

1. Implement algorithm and include a system call to count the number of milliseconds it takes to run.
2. Count the exact number of times *each* of the algorithm's operations is executed, assuming each particular line takes a constant amount of time for a data set of size n, and add time of all lines to get a polynomial expression in terms of n.
3. Identify the operations (lines) that contribute *most* to the total running time and count the number of times that operation is executed (best option).

==> the **basic operation (aka dominant operation)**

Algorithm Time Efficiency

Some algorithms take the same amount of time on all input instances. For these algorithms, the running time is given as a constant.

For some algorithms, there is only a worst-case time for all input instances of a particular size, n.

For other algorithms, there are best-case, worst-case, and average-case input instances that depend on other qualities of the input than just the input size.

For algorithm A on all possible inputs of size *n*:

Worst-case: The input(s) for which A executes the most steps.

Best-case: The input(s) for which A executes the fewest steps.

Reading assignment:
Chapters 1-4 in CLRS

Asymptotic Analysis-Ch. 3

Main idea: Running time is measured in the limit as the *input size* grows to infinity.

- focus on calculating algorithm running time in terms of its *rate of growth* with increasing problem size. To make this task easier, we can
  - **identify terms of highest order and ignore lower order terms**
  - **disregard multiplicative constants**

Saying an algorithm has running time $\theta(n^2)$ says that the *order of growth* of the running time is in the set of functions whose running time is n^2, a quadratic function of n

Asymptotic Analysis

- Names for classes of algorithms:

constant	$\theta(n^0) = \theta(1)$	Growth Rate Increasing ↓
logarithmic	$\theta(\lg n)$	
polylogarithmic	$\theta(\lg^k n)$, $k \geq 1$	
linear	$\theta(n)$	
linearithmic	$\theta(n\lg n)$	
quadratic	$\theta(n^2)$	
cubic	$\theta(n^3)$	
polynomial	$\theta(n^k)$, $k \geq 1$	
exponential	$\theta(a^n)$, $a > 1$	

Asymptotic Analysis

Example: As n grows larger, an algorithm with running time of order n^2 will "eventually" run slower than one with running time of order n, which in turn will eventually run slower than one with running time of order (lgn).

Asymptotic analysis in terms of "Big Oh", "Big Omega", and "Theta" are the classification schemes we will use to make these notions precise.

Note: Our conclusions will only be valid "in the limit" or "asymptotically". That is, they may not hold true for small values of n.

"Big Oh" - Upper Bounding Running Time

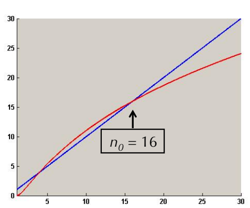
Definition: $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

Intuition:

- **f(n) ∈ O(g(n)) means f(n) is "of order at most", or "less than or equal to" g(n) when we ignore small values of n and constants**
- **some constant multiple of g(n) is an upper bound for f(n) (for large enough n)**

Example: $(\lg n)^2$ is O(n)



$f(n) = (\lg n)^2$

$g(n) = n$

$(\lg n)^2 \leq n$ for all $n_0 \geq 16$, so $(\lg n)^2$ is $O(n)$

$(\lg n)^2 = \lg^2 n$

Asymptotic notation and logarithms:

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

- changing base b changes only constant factor that can be ignored
- When we say $f(n) \in O(\log n)$, the base of the log is unimportant (but it will usually be $\log_2 n$, written as lgn).

"Big Omega" - Lower Bounding Running Time

Definition: $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 1$ such that

$$f(n) \geq cg(n) \quad \text{for all} \quad n \geq n_0.$$

Intuition:

- **$f(n) \in \Omega(g(n))$ means f(n) is "of order at least" or "greater than or equal to" g(n) when we ignore small values of n.**
- **some constant multiple of g(n) is a <u>lower bound</u> for f(n) (for large enough n).**

"Theta" - Tightly Bounding Running Time

Definition: $f(n) \in \theta(g(n))$ if there exist constants c_1, $c_2 > 0$ and $n_0 \geq 1$ such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.$$

Intuition:

- **$f(n) \in \theta(g(n))$ means f(n) is "of the same order as", or "equal to" g(n) when we ignore small values of n.**

<u>Useful way to show "Theta" relationships</u>:

❑ Show both a "Big Oh" and "Big Omega" relationship.

Asymptotic Analysis

- Classifying algorithms is generally done in terms of *worst-case* running time Big Oh or Theta. We rarely express the running time in terms of Big Omega. If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then $f(n) \in \Theta(g(n))$:
  - $O(f(n))$: Big Oh--asymptotic *upper* bound.
  - $\Omega(f(n))$: Big Omega--asymptotic *lower* bound
  - $\Theta(f(n))$: Theta--asymptotic *tight* bound

Little Oh

"Little Oh" notation is used to denote strict upper bounds, (Big-Oh bounds are not necessarily strict inequalities).

Definition: $f(n) \in o(g(n))$ if for *every* $c > 0$, there exists some $n_0 \geq 1$ such that for all $n \geq n_0$, <u>$f(n) < cg(n)$</u>.

Intuition:

- $f(n) \in o(g(n))$ means f(n) is "strictly less than" any constant multiple of g(n) when we ignore small values of n
- f(n) is trapped below any constant multiple of g(n) for large enough n
- For example, if $f(n) \in O(n)$, then $f(n) \in o(n^2)$

Little Omega

"Little Omega" notation is used to denote strict lower bounds (Ω bounds are not necessarily strict inequalities).

Definition: $f(n) \in \omega(g(n))$ if for every $c > 0$, there exists some $n_0 \geq 1$ such that for all $n \geq n_0$, <u>$f(n) > cg(n)$</u>.

Intuition:

- $f(n) \in \omega(g(n))$ means f(n) is "strictly greater than" any constant multiple of g(n) when we ignore small values of n
- f(n) is trapped above any constant multiple of g(n) for large enough n

Using Limits to Compare Orders of Growth

<u>Showing "Little Oh and Little Omega" relationships</u>:

$\lim_{n \to \infty} f(n) / g(n) = 0$ implies that f(n) has a smaller order of growth than g(n)

$\lim_{n \to \infty} f(n) / g(n) = c > 0$ implies that f(n) has the same order of growth as g(n) (c is constant)

$\lim_{n \to \infty} f(n) / g(n) = \infty$ implies that f(n) has a larger order of growth than g(n)

Little Oh and Little Omega

Showing "Little Oh and Little Omega" relationships:

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

$f(n) \in \omega(g(n))$ iff $\lim_{n \to \infty} f(n) / g(n) = \infty$

Showing Theta relationships

$f(n) \in \Theta(g(n))$ iff $\lim_{n \to \infty} f(n) / g(n) = c > 0$

Basic asymptotic efficiency classes

Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	Linearithmic	Some divide and conquer; best sorting time.
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop
2^n	Exponential	Algorithm generates all subsets of n-element set
n!	Factorial	Algorithm generates all permutations of n-element set