

# Lecture 2: Asymptotic Notation

**Steven Skiena**

Department of Computer Science  
State University of New York  
Stony Brook, NY 11794-4400

<http://www.cs.sunysb.edu/~skiena>

## Problem of the Day

---

The *knapsack problem* is as follows: given a set of integers  $S = \{s_1, s_2, \dots, s_n\}$ , and a given target number  $T$ , find a subset of  $S$  which adds up exactly to  $T$ . For example, within  $S = \{1, 2, 5, 9, 10\}$  there is a subset which adds up to  $T = 22$  but not  $T = 23$ .

Find counterexamples to each of the following algorithms for the knapsack problem. That is, give an  $S$  and  $T$  such that the subset is selected using the algorithm does not leave the knapsack completely full, even though such a solution exists.

## Solution

---

- Put the elements of  $S$  in the knapsack in left to right order if they fit, i.e. the first-fit algorithm?
- Put the elements of  $S$  in the knapsack from smallest to largest, i.e. the best-fit algorithm?
- Put the elements of  $S$  in the knapsack from largest to smallest?

# The RAM Model of Computation

---

Algorithms are an important and durable part of computer science because they can be studied in a machine/language independent way.

This is because we use the **RAM model of computation** for all our analysis.

- Each “simple” operation (+, -, =, if, call) takes 1 step.
- Loops and subroutine calls are *not* simple operations. They depend upon the size of the data and the contents of a subroutine. “Sort” is not a single step operation.

- Each memory access takes exactly 1 step.

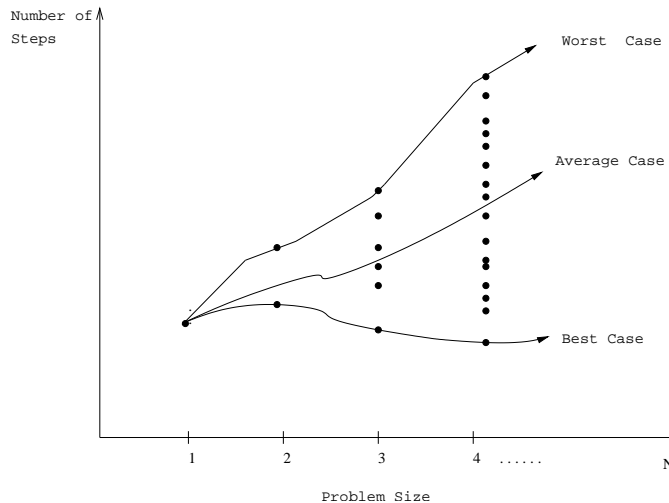
We measure the run time of an algorithm by counting the number of steps, where:

This model is useful and accurate in the same sense as the flat-earth model (which *is* useful)!

# Worst-Case Complexity

---

The *worst case complexity* of an algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ .



## Best-Case and Average-Case Complexity

---

The *best case complexity* of an algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ .

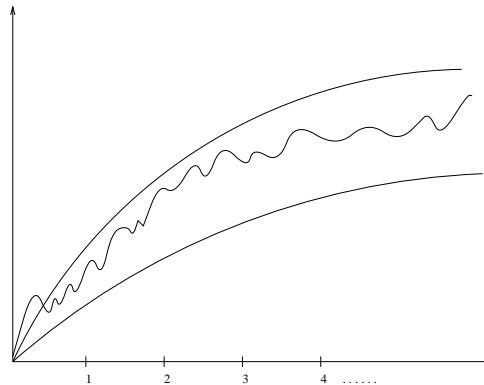
The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size  $n$ .

Each of these complexities defines a numerical function: time vs. size!

# Exact Analysis is Hard!

---

Best, worst, and average are difficult to deal with precisely because the details are very complicated:



It is easier to talk about *upper and lower bounds* of the function. Asymptotic notation ( $O$ ,  $\Theta$ ,  $\Omega$ ) are as well as we can practically deal with complexity functions.



# Names of Bounding Functions

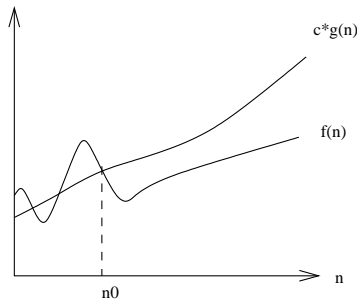
---

- $g(n) = O(f(n))$  means  $C \times f(n)$  is an *upper bound* on  $g(n)$ .
- $g(n) = \Omega(f(n))$  means  $C \times f(n)$  is a *lower bound* on  $g(n)$ .
- $g(n) = \Theta(f(n))$  means  $C_1 \times f(n)$  is an upper bound on  $g(n)$  and  $C_2 \times f(n)$  is a lower bound on  $g(n)$ .

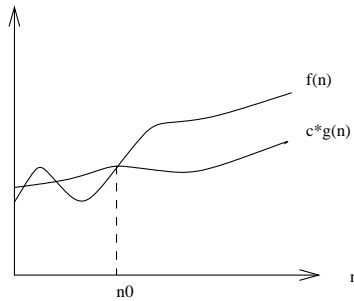
$C$ ,  $C_1$ , and  $C_2$  are all constants independent of  $n$ .

# $O$ , $\Omega$ , and $\Theta$

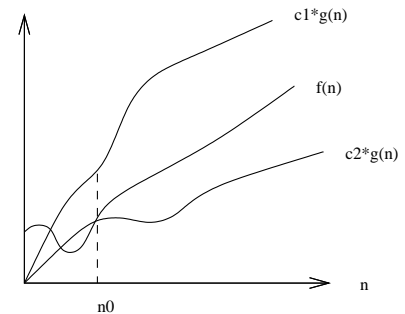
---



(a)



(b)



(c)

The definitions imply a constant  $n_0$  *beyond which* they are satisfied. We do not care about small values of  $n$ .

# Formal Definitions

---

- $f(n) = O(g(n))$  if there are positive constants  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or below  $c \cdot g(n)$ .
- $f(n) = \Omega(g(n))$  if there are positive constants  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or above  $c \cdot g(n)$ .
- $f(n) = \Theta(g(n))$  if there exist positive constants  $n_0$ ,  $c_1$ , and  $c_2$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies between  $c_1 \cdot g(n)$  and  $c_2 \cdot g(n)$  inclusive.

## Big Oh Examples

---

$3n^2 - 100n + 6 = O(n^2)$  because  $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$  because  $.01n^3 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 \neq O(n)$  because  $c \cdot n < 3n^2$  when  $n > c$

Think of the equality as meaning *in the set of functions*.

## Big Omega Examples

---

$3n^2 - 100n + 6 = \Omega(n^2)$  because  $2.99n^2 < 3n^2 - 100n + 6$

$3n^2 - 100n + 6 \neq \Omega(n^3)$  because  $3n^2 - 100n + 6 < n^3$

$3n^2 - 100n + 6 = \Omega(n)$  because  $10^{10^{10}} n < 3n^2 - 100 + 6$

# Big Theta Examples

---

$3n^2 - 100n + 6 = \Theta(n^2)$  because  $O$  and  $\Omega$

$3n^2 - 100n + 6 \neq \Theta(n^3)$  because  $O$  only

$3n^2 - 100n + 6 \neq \Theta(n)$  because  $\Omega$  only

# Big Oh Addition/Subtraction

---

Suppose  $f(n) = O(n^2)$  and  $g(n) = O(n^2)$ .

- What do we know about  $g'(n) = f(n) + g(n)$ ? Adding the bounding constants shows  $g'(n) = O(n^2)$ .
- What do we know about  $g''(n) = f(n) - |g(n)|$ ? Since the bounding constants don't necessarily cancel,  $g''(n) = O(n^2)$

We know nothing about the lower bounds on  $g'$  and  $g''$  because we know nothing about lower bounds on  $f$  and  $g$ .

# Big Oh Multiplication by Constant

---

Multiplication by a constant does not change the asymptotics:

$$O(c \cdot f(n)) \rightarrow O(f(n))$$

$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$

$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$



# Big Oh Multiplication by Function

---

But when both functions in a product are increasing, both are important:

$$O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$$

$$\Omega(f(n)) * \Omega(g(n)) \rightarrow \Omega(f(n) * g(n))$$

$$\Theta(f(n)) * \Theta(g(n)) \rightarrow \Theta(f(n) * g(n))$$