Big O, Omega, and Theta

Compiled from internet

Thanks to internet community for resources sharing

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

Does it terminate?

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

Is it correct? Can you prove it?

Loop invariant

Loop invariant: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.

Upon termination of the loop, the invariant should help you show something useful about the algorithm.

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

Loop invariant?

Loop invariant

Loop invariant: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.

At the start of each iteration of the for loop of lines 1-7 the subarray *A[1..j − 1]* is the sorted version of the original elements of *A[1..j − 1]*

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2        current ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > current
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← current
```

Proof?

Loop invariant

At the start of each iteration of the for loop of lines 1-7 the subarray *A[1..j − 1]* is the sorted version of the original elements of *A[1..j − 1]*

Proof by induction

- Base case: invariant is true before loop
- Inductive case: it is true after each iteration

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2          current ← A[j]
3          i ← j − 1
4          while i > 0 and A[i] > current
5                  A[i + 1] ← A[i]
6                  i ← i − 1
7          A[i + 1] ← current
```

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

How long will it take to run?

Asymptotic notation

- How do you answer the question: "what is the running time of algorithm *x*?"
- We need a way to talk about the computational cost of an algorithm that focuses on the essential parts and ignores irrelevant details
- You've seen some of this already:
  - linear
  - $n \log n$
  - n^2

Asymptotic notation

Precisely calculating the actual steps is tedious and not generally useful

Different operations take different amounts of time. Even from run to run, things such as caching, etc. cause variations

We want to identify **categories** of algorithmic runtimes

For example…

$f_1(n)$ takes n^2 steps

$f_2(n)$ takes $2n + 100$ steps

$f_3(n)$ takes $3n+1$ steps

Which algorithm is better?

Is the difference between f_2 and f_3 important/significant?

Runtime examples

	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
$n = 100$	< 1 sec	< 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

(adapted from [2], Table 2.1, pg. 34)

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant factor of *g(n)*

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant multiplied by *g(n)*

For some increasing range

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$O(n^2) = \begin{array}{rcc} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n \end{array}$$

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Generally, we’re most interested in big O notation since it is an upper bound on the running time

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\mathrm{W}(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* below by some constant factor of *g(n)*

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$\Omega(n^2) = \begin{array}{rcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n^3 \end{array}$$

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above **and** below by some constant factor of *g(n)* (though different constants)

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Note: A function is theta bounded **iff** it is big O bounded and Omega bounded

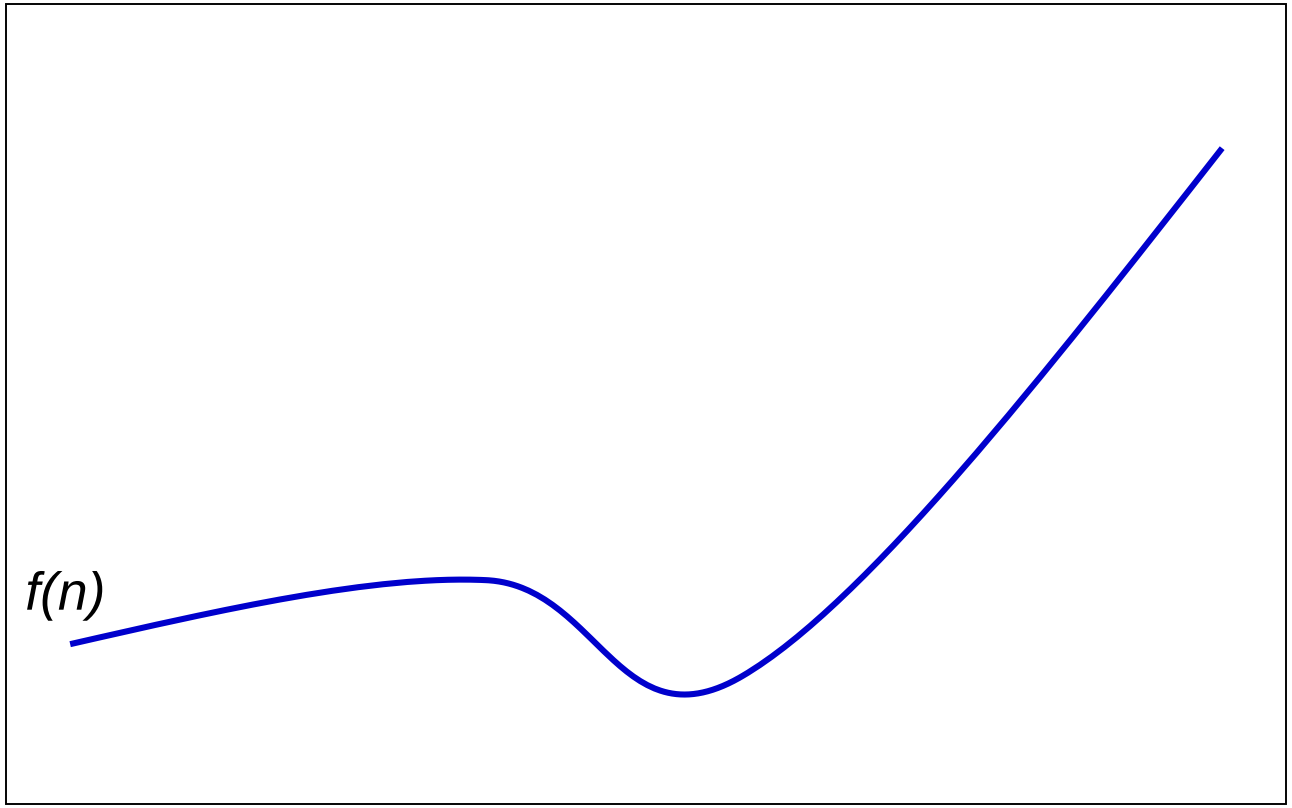
Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

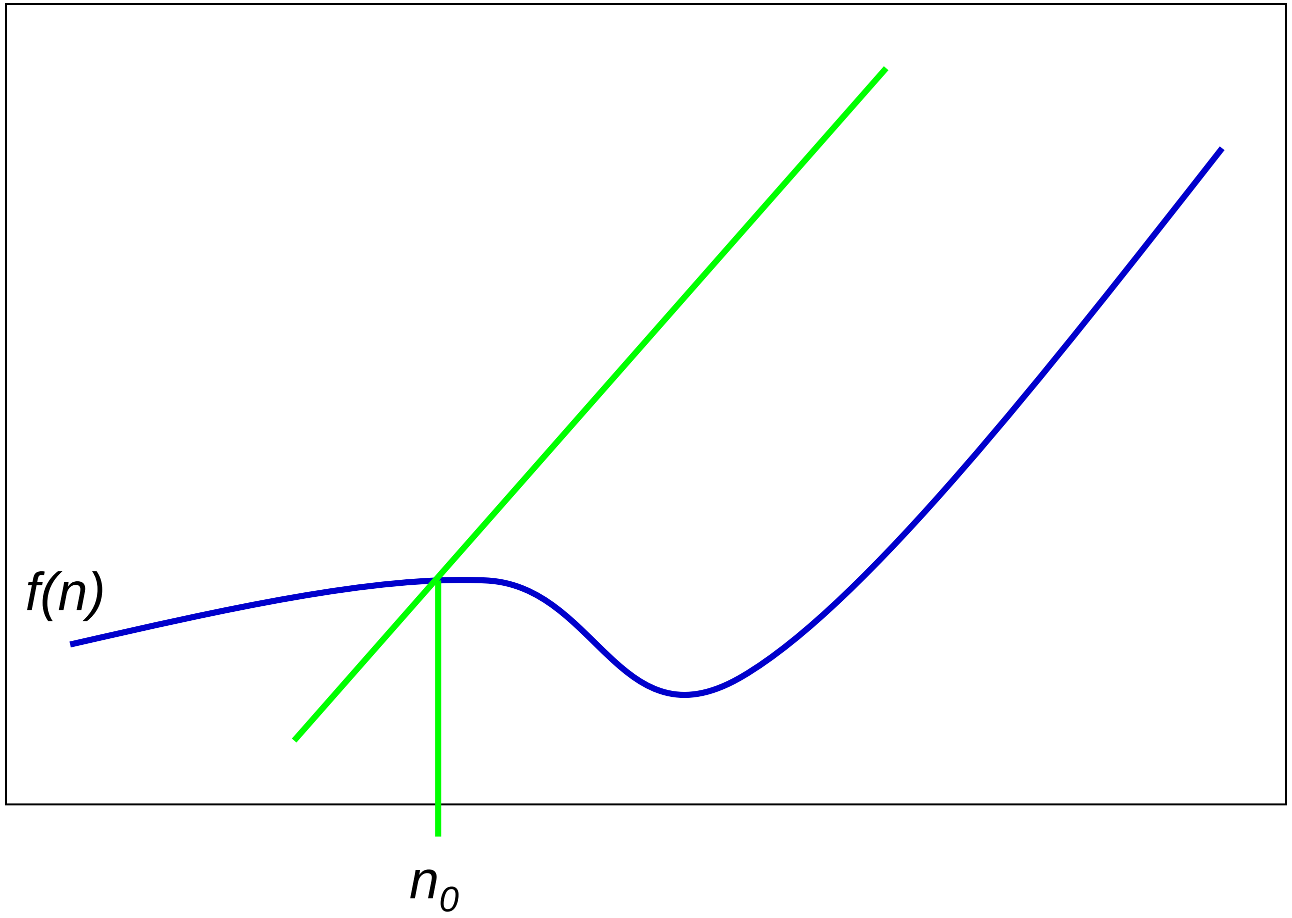
$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$\Theta(n^2) = \begin{array}{lcr} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 3n^2 + n\log n \end{array}$$

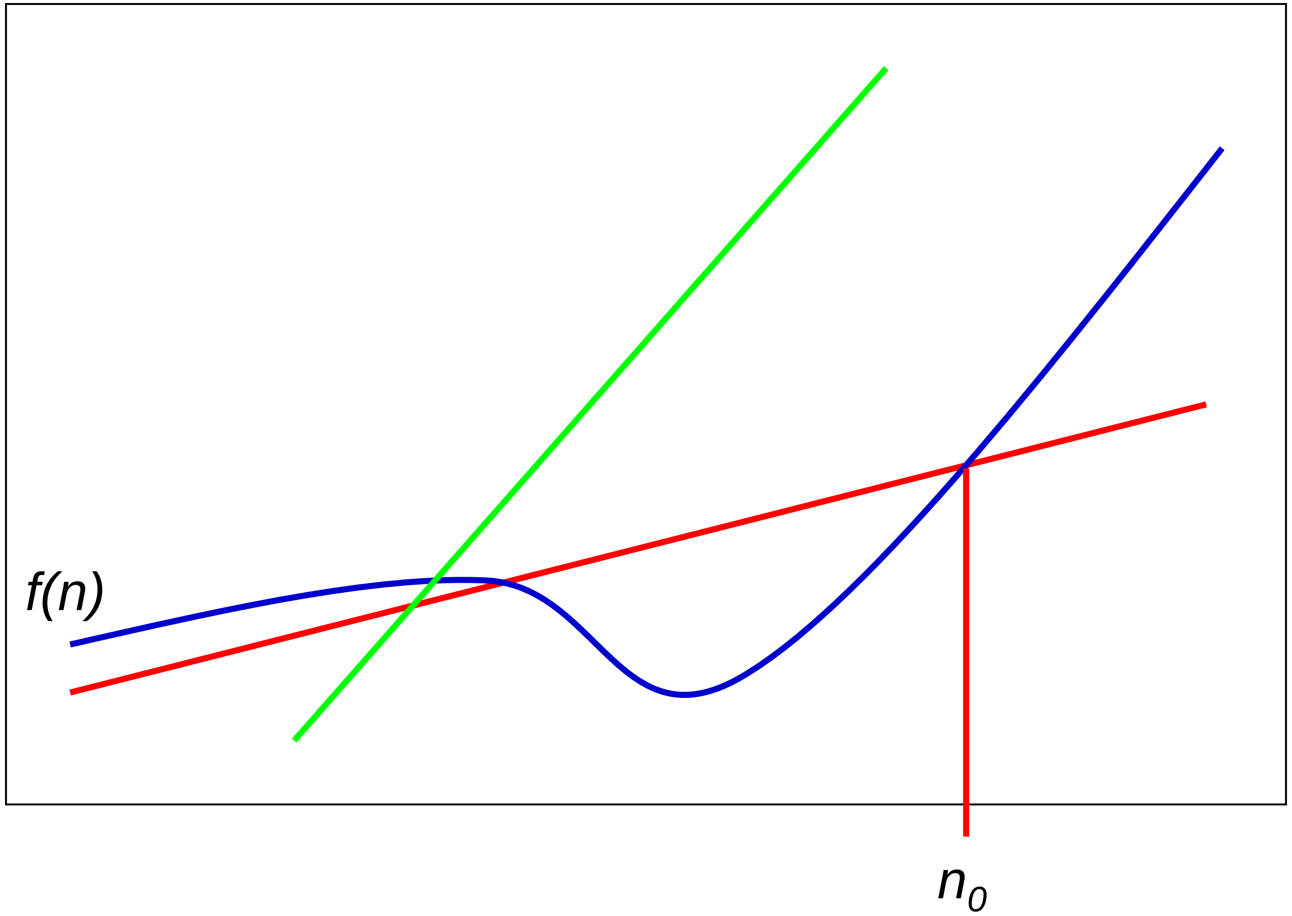
Visually



Visually: upper bound



Visually: lower bound



worst-case vs. best-case vs. average-case

worst-case: what is the worst the running time of the algorithm can be?

best-case: what is the best the running time of the algorithm can be?

average-case: given random data, what is the running time of the algorithm?

Don't confuse this with O, Ω and Θ. The cases above are *situations*, asymptotic notation is about bounding particular situations

Proving bounds: find constants that satisfy inequalities

Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$

Step 1: Prove $O(n^2)$ – Find constants c and n_0 such that
$5n^2 - 15n + 100 \leq cn^2$ for all n > n_0

$$cn^2 \geq 5n^2 - 15n + 100$$

$$c \geq 5 - 15/n + 100/n^2$$

Let n_0 =1 and c = 5 + 100 = 105.
$100/n^2$ only get smaller as n increases and we ignore $-15/n$ since it only varies between -15 and 0

Proving bounds

Step 2: Prove $\Omega(n^2)$ – Find constants c and n_0 such that $5n^2 - 15n + 100 \geq cn^2$ for all n > n_0

$$cn^2 \leq 5n^2 - 15n + 100$$

$$c \leq 5 - 15/n + 100/n^2$$

Let n_0 =4 and c = 5 – 15/4 = 1.25 (or anything less than 1.25). 15/n is always decreasing and we ignore $100/n^2$ since it is always between 0 and 100.

Bounds

Is $5n^2 \; O(n)$? **No**

How would we prove it?

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Disproving bounds

Is $5n^2$ $O(n)$?

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Assume it's true.

That means there exists some c and n_0 such that

$$5n^2 \text{ £ } cn \text{ for } n > n_0$$

$$5n \text{ £ } c$$ contradiction!

Some rules of thumb

Multiplicative constants can be omitted

- $14n^2$ becomes n^2
- 7 log n become log n

Lower order functions can be omitted

- $n + 5$ becomes n
- $n^2 + n$ becomes n^2

n^a dominates n^b if $a > b$

- n^2 dominates n, so n^2+n becomes n^2
- $n^{1.5}$ dominates $n^{1.4}$

Some rules of thumb

a^n dominates b^n if $a > b$

- 3^n dominates 2^n

Any exponential dominates any polynomial

- 3^n dominates n^5
- 2^n dominates n^c

Any polynomial dominates any logorithm

- n dominates log n or log log n
- n^2 dominates n log n
- $n^{1/2}$ dominates *log n*

Do **not** omit lower order terms of different variables ($n^2 + m$) does not become n^2

Big O

$n^2 + n \log n + 50$

$2^n - 15n^2 + n^3 \log n$

$n^{\log n} + n^2 + 15n^3$

$n^5 + n! + n^n$

Some examples

- O(1) – constant. Fixed amount of work, regardless of the input size
  - add two 32 bit numbers
  - determine if a number is even or odd
  - sum the first 20 elements of an array
  - delete an element from a doubly linked list
- O(log *n*) – logarithmic. At each iteration, discards some portion of the input (i.e. half)
  - binary search

Some examples

- O(n) – linear. Do a constant amount of work on each element of the input
  - find an item in a linked list
  - determine the largest element in an array
- O(n log n) log-linear. Divide and conquer algorithms with a linear amount of work to recombine
  - Sort a list of number with MergeSort
  - FFT

Some examples

- $O(n^2)$ – quadratic. Double nested loops that iterate over the data
  - Insertion sort
- $O(2^n)$ – exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming
- O(n!)
  - Enumerate all permutations
  - determinant of a matrix with expansion by minors