Asymptotics

EXAMPLE: INSERTION-SORT

INSERTION-SORT(A, n)

```
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

In-place Algorithm

A sorting algorithm sorts **in-place** if only a constant number of elements of the input array are ever stored outside the array.

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

Correctness

```
INSERTION-SORT(A, n)
1 for i = 2 to n
2     key = A[i]
3     // Insert A[i] into the sorted subarray A[1 : i – 1].
4     j = i – 1
5     while j > 0 and A[j] > key
6         A[j + 1] = A[j]
7         j = j – 1
8     A[j + 1] = key
```

Invariant (**Loop Invariant**)

At the start of each iteration of the for loop (lines 1-8), the sub-array A[1...i-1] is sorted.

Correctness

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Invariant (**Loop Invariant**)

At the start of each iteration of the for loop (lines 1-8), the sub-array A[1...i-1] is sorted.

We use **Loop Invariant** to show correctness.

Loop Invariant (3 aspects)

- Initialization
  - ✓ It is true prior to the first iteration of the loop.
- Maintenance
  - ✓
- Termination
  - ✓

Loop Invariant (3 aspects)

- Initialization
  - ✓ It is true prior to the first iteration of the loop.
- Maintenance
  - ✓ If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination
  - ✓

Loop Invariant (3 aspects)

- Initialization
  - ✓ It is true prior to the first iteration of the loop.
- Maintenance
  - ✓ If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination
  - ✓ When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Loop Invariant (3 aspects)

- Initialization
  - ✓ It is true prior to the first iteration of the loop.
- ✘

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Loop Invariant (3 aspects)

- Initialization
  - ✓ It is true prior to the first iteration of the loop.
- ✗ Consists of A[1]

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i - 1].
    j = i - 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key
```

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Loop Invariant (3 aspects)

- Maintenance
  - ✓ If it is true before an iteration of the loop, it remains true before the next iteration.
  - ✘ the body of the for loop works by
    - ✘

Loop Invariant (3 aspects)

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

- Maintenance
  - ✓ If it is true before an iteration of the loop, it remains true before the next iteration.
  - ✘ the body of the for loop works by
    - ✘ moving A[i-1], A[i-2], and so on, one position to its right, until it finds the proper position for A[i] (lines 4–7),
  - ✘

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Loop Invariant (3 aspects)

- Maintenance
  - ✓ If it is true before an iteration of the loop, it remains true before the next iteration.
  - the body of the for loop works by
    - moving A[i-1], A[i-2], and so on, one position to its right, until it finds the proper position for A[i] (lines 4–7),
  - The subarray A[1..i] then consists of the elements originally in A[1..i] but in sorted order.
  - i=i+1 for the next iteration of the “for” loop preserves the loop invariant.

INSERTION-SORT(A, n)

```
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

Loop Invariant (3 aspects)

- Termination
  - ✓ When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
  - ✘

Invariant (**Loop Invariant**)

At the start of each iteration of the for loop of lines 1-8, the sub-array A[1...i-1] is sorted.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Loop Invariant (3 aspects)

- Termination
  - ✓ When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

- The condition causing the for loop to terminate is that i = n+1.

Invariant (**Loop Invariant**)

At the start of each iteration of the for loop of lines 1-8, the sub-array A[1...i-1] is sorted.

Analyzing time complexity

Exercise: For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

	1 second (10^6 us)	1 minute (6*10^7 us)	1 hour (3.6 * 10^9 us)	1 day (8.64*10^9 us)
log n				
n				
n log n				
n^2				
n^3				
2^n				
n!				

Analyzing time complexity

Exercise: For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

	1 second (10^6 us)	1 minute (6*10^7 us)	1 hour (3.6 * 10^9 us)	1 day (8.64*10^9 us)
log n	2^{10^6}			
n				
n log n				
n^2				
n^3				
2^n				
n!				

Analyzing time complexity

Exercise: For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

	1 second (10^6 us)	1 minute (6*10^7 us)	1 hour (3.6 * 10^9 us)	1 day (8.64*10^9 us)
log n	$2^{10\text{^}6}$			
n	10^6			
n log n	~ 62,746			
n^2	~ 1,000			
n^3	~100			
2^n	~19			
n!	~9			

Analyzing time complexity

Exercise: For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

	1 second (10^6 us)	1 minute (6*10^7 us)	1 hour (3.6 * 10^9 us)	1 day (8.64*10^9 us)
log n	$2^{10\text{^}6}$			$2^{8.64*10\text{^}9}$
n	10^6			
n log n	~62,746			
n^2	~1,000			
n^3	~100			
2^n	~19			~36
n!	~9			~16

Asymptotics

- Refers to the behavior of mathematical functions, algorithms, or models as inputs become large.

- Describes the efficiency of algorithms.

This tells you how algorithms scale as the problem size increases.

Functions

-

$n \log n$ n^2

$n^2 \log n$ $n \log n$

n^2 $n^2 \log n$

O-notation

O-notation

O-notation characterizes an ***upper bound*** on the asympototic behavior of a function: it says that a function grows ***no faster*** than a certain rate. This rate is based on the highest order term.

For example:

$100n^2 + 1000n + 50$ is $O(?)$,

What is the highest order term?

O-notation

O-notation characterizes an ***upper bound*** on the asympototic behavior of a function: it says that a function grows ***no faster*** than a certain rate. This rate is based on the highest order term.

For example:

$100n^2 + 1000n + 50$ is $O(n^2)$, since the highest order term is $100n^2$, and therefore the function grows no faster than n^2.

O-notation

O-notation characterizes an ***upper bound*** on the asympototic behavior of a function: it says that a function grows ***no faster*** than a certain rate. This rate is based on the highest order term.

For example:

$100n^2 + 1000n + 50$ is $O(n^2)$, since the highest order term is $100n^2$, and therefore the function grows no faster than n^2.

Question: $100n^2 + 1000n + 50$ is $O(n^3)$?

O-notation

O-notation characterizes an ***upper bound*** on the asymptototic behavior of a function: it says that a function grows ***no faster*** than a certain rate. This rate is based on the highest order term.

For example:

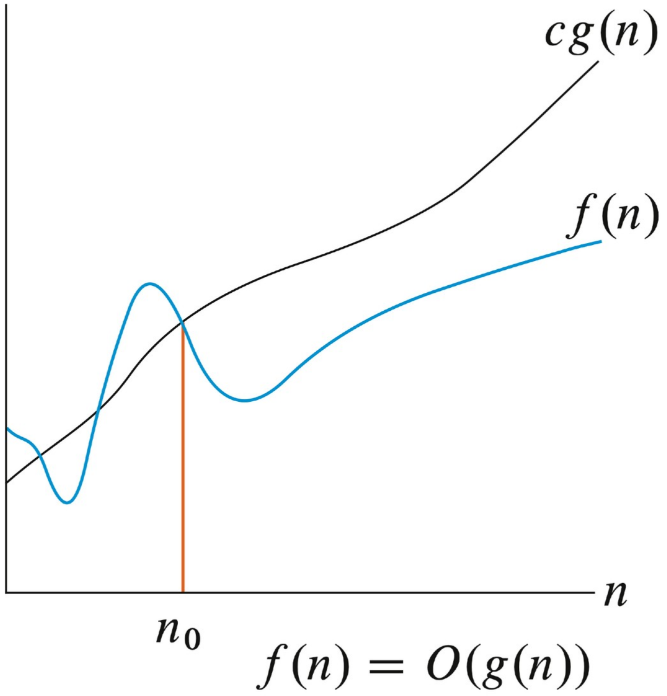
$100n^2 + 1000n + 50$ is $O(n^2)$, since the highest order term is $100n^2$, and therefore the function grows no faster than n^2.

Question: $100n^2 + 1000n + 50$ is $O(n^3)$?

In terms of algorithms?

O-notation

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

O-notation

Is $2n^2 = O(n^3)$?

O-notation

Example

$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

Ω-notation

Ω-notation characterizes a ***lower bound*** on the asymptotic behavior of a function.

For example:

$100n^2 + 1000n + 50$ is Ω(?), since the highest order term is $100n^2$, and it grows at least as fast as ?.

Ω-notation

Ω-notation characterizes a ***lower bound*** on the asymptotic behavior of a function.

For example:

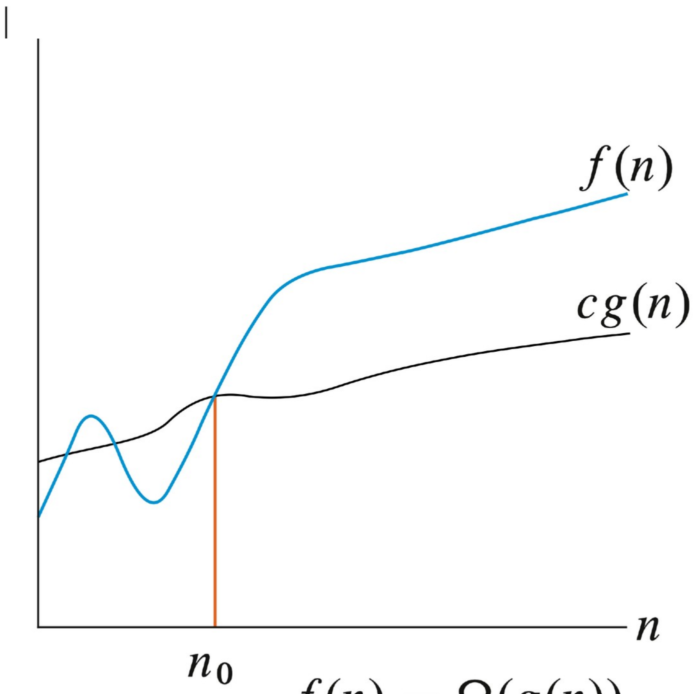
$100n^2 + 1000n + 50$ is $\Omega(n^2)$, since the highest order term is $100n^2$, and it grows at least as fast as n^2.

Question: $100n^2 + 1000n + 50$ is $\Omega(n)$?

Ω-notation

Ω-notation

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



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

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Ω-notation

$Is\ 2n^2 = \Omega(n^3)?$

Ω-notation

Example

$\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg \lg \lg n$
2^{2^n}

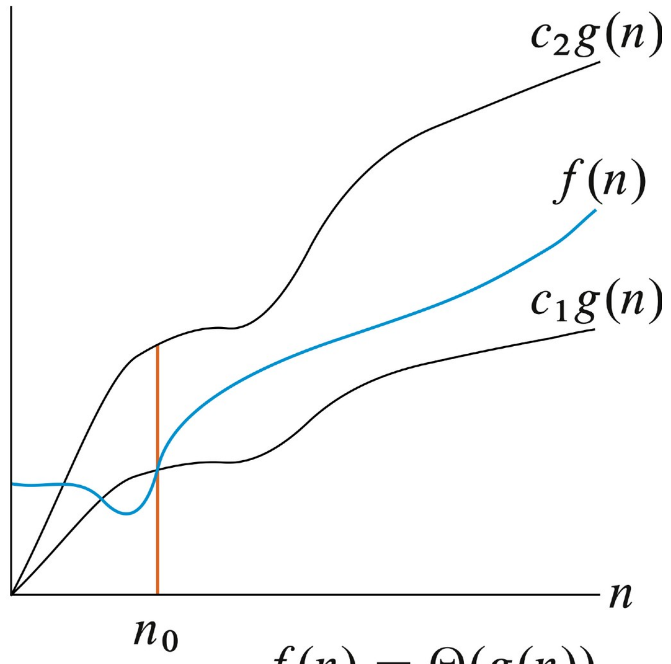
Θ-notation

Θ-notation characterizes a ***tight bound*** on the asympototic behavior of a function: it says that a function grows ***precisely*** at a certain rate, again based on the highest-order term.

If a function is both O(g(n)) and Ω(g(n)), then a function is $\Theta(g(n))$.

Θ-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 $n \ge n_0\}$.



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

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Θ-notation

Is $n^2/2 - 3n = \Theta(n^3)$?

Θ-notation

Is $n^2/2 - 2n = \Theta(n^2)$*?*

Θ-notation

Is $n^2/2 - 2n = \Theta(n^2)$*?*

Example

$n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem

$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Leading constants and low-order terms don't matter.

Important points

Impotant points (constants)

$f(n) = 100n^2, g(n) = n^2;$

- Both are $O(n^2)$ (in asymptotic analysis).
- When analyzing algorithms or functions for large inputs, we care more about how fast something grows rather than the exact value.
- The shape of the growth (quadratic) dominates, not the size of the constant.

Constants do not matter!!!

Impotant points (constants)

$f(n) = 3n$, $g(n) = 100n$;

- Both grow linearly (**O(n)**), even though one is faster in practice. As $n \to \infty$, they behave similarly in shape, so the constant difference becomes less meaningful.

Constants do not matter!!!

Impotant points (constants)

$f(n) = 100n, g(n) = n^2$;

- Even though f(n) is worse for small n, it scales much better than g(n) as n increases.
- So, we say $f(n)=O(n)$, $g(n)=O(n^2)$, which shows f(n) is asymptotically better.

Removing constants allows for easier comparison between algorithms.

Impotant points (constants)

$f(n) \leq 4n + 12 \log n + 57;$

- We simplify to f(n)=O(n)
- Constants clutter notation without adding insight in asymptotic contexts.

This keeps the analysis clear and general.

ASMPTOTIC NOTATION IN EQUATIONS

When on right-hand side:

stands for some anonymous function in the set .

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means $2n^2 + 3n + 1 = 2n^2 + f(n)$ *for some* $f(n) \in \Theta(n)$. In particular, $f(n) = 3n + 1$.

When on left-hand side:

No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning *for all* functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$.

ASMPTOTIC NOTATION IN EQUATIONS
(continued)

Can chain together:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \, . \end{aligned}$$

Interpretation:

- First equation: There exists $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.
- Second equation: For all $g(n) \in \Theta(n)$ (such as the $f(n)$ used to make the first equation hold), there exists $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$.

CHAPTER 3 OVERVIEW

Goals

- A way to describe behavior of functions in the limit. We're studying asymptotic efficiency.
- Describe growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare "sizes" of functions:

$$
\begin{array}{ccc}
O & \approx & \leq \\
\Omega & \approx & \geq \\
\Theta & \approx & = \\
o & \approx & < \\
\omega & \approx & >
\end{array}
$$