Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Best-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

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

Best-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

- The smallest value of t_j happens when the input array is already sorted; all *t_j's will be 1*

- Running time of INSERTION-SORT will be simplified to

$$14n - 11$$

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
        sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

3	5	7	9	10	12	14	15

Example of the best input

Best-case analysis

- Best-case analysis provides a **lower-bound**, i.e. the **least amount of time** required to sort any input of size n.

- However, this measure can be **misleading** since you can design an algorithm that works well for only one input.

Worst-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

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

Worst-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

- The largest value of t_j happens when the input array is sorted in reverse; all *t_j's will be j*
- Running time of INSERTION-SORT will be simplified to

?

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
        sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

15	14	12	10	9	7	5	3

Example of the worst input

Worst-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

- The largest value of t_j happens when the input array is sorted in reverse; all *t_j's will be j*
- Running time of INSERTION-SORT will be simplified to

$$5n^2 + 9n - 11$$

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

15	14	12	10	9	7	5	3

Example of the worst input

Side note

- For the mathematical background regarding summation formulas read CLRS page 1145-1154 (before approximation by integrals).

Worst-case analysis

- Worst-case analysis provides an **upper-bound**, i.e. the **most amount of time** required to finish on any input of size n.

- This measure is very useful since it provides a **guarantee**.

Average-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

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

Average-case analysis

- Running time of INSERTION-SORT is

$$10\sum_{j=2}^{n} t_j + 4n - 1$$

- If we choose n numbers randomly, on average, half the elements in the subarray A[1 .. j - 1] are bigger than *key*. So, *t_j's will be about* $\frac{j}{2}$
- Running time of INSERTION-SORT will be simplified to

$$2.5n^2 + 6.5n - 6$$

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

Average-case analysis

- Usually, the average running time is **as bad as** the worst-case running time.

- To do an **accurate** average-case analysis we need to consider **probabilities** in the analysis.

- Usually we just do **worst-case analysis**.

Let's compare

best-case

$$T(n) = 14n - 11$$

average-case

$$T(n) = 2.5\,n^2 + 6.5\,n - 6$$

worst-case

$$T(n) = 5n^2 + 9n - 11$$

$\boldsymbol{T}$ stands for **time**. $T(n)$ is running time for input of size n.

Let's compare

	best-case	average-case	worst-case
	$T(n) = 14n - 11$	$T(n) = 2.5\,n^2 + 6.5\,n - 6$	$T(n) = 5n^2 + 9n - 11$
n=2	17	17	27
n=10	129	309	579
n=100	1,389	25,644	50,889
n=1000	**13,989**	**2,506,490**	**5,008,990**

Let's compare

	best-case	average-case	worst-case
	$T(n) = 14n - 11$	$T(n) = 2.5\,n^2 + 6.5\,n - 6$	$T(n) = 5n^2 + 9n - 11$
n=2	17	17	27
n=10	129	309	579
n=100	1,389	25,644	50,889
n=1000	**13,989**	**2,506,490**	**5,008,990**

For large n, value of $T(n)$ is very close to its **largest term**

Let's simplify a little bit

best-case

$$T(n) = 14n - 11$$

average-case

$$T(n) = 2.5\,n^2 + 6.5\,n - 6$$

worst-case

$$T(n) = 5n^2 + 9n - 11$$

Let’s simplify a little bit

best-case

$$T(n) = 14n \cancel{- 11}$$

average-case

$$T(n) = 2.5\,n^2 + \cancel{6.5\,n} \cancel{- 6}$$

worst-case

$$T(n) = 5n^2 + \cancel{9n} - \cancel{11}$$

Lower order terms don’t matter!

Let's simplify a little bit

best-case	average-case	worst-case
$T(n) = 14n$	$T(n) = 2.5\, n^2$	$T(n) = 5n^2$

So, let's see if we can simplify these formulas even more!

Let's simplify a little bit

best-case	average-case	worst-case
$T(n) = 14n$	$T(n) = 2.5\,n^2$	$T(n) = 5n^2$

Which two functions are the most similar?

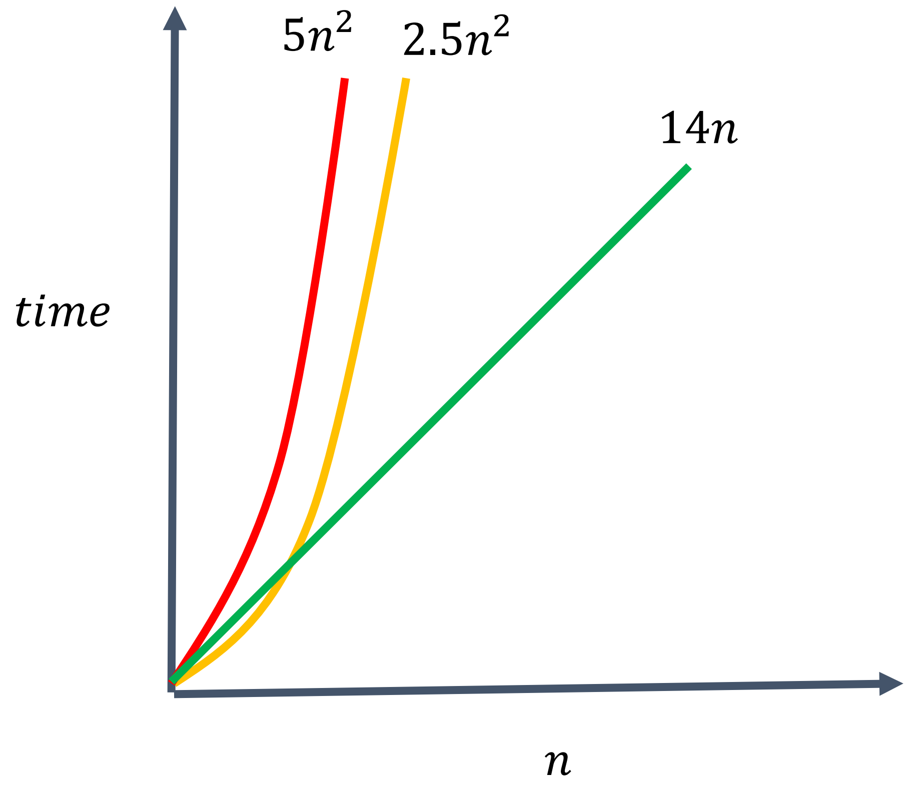
Let's simplify a little bit

best-case	average-case	worst-case
$T(n) = 14n$	$T(n) = 2.5\,n^2$	$T(n) = 5n^2$

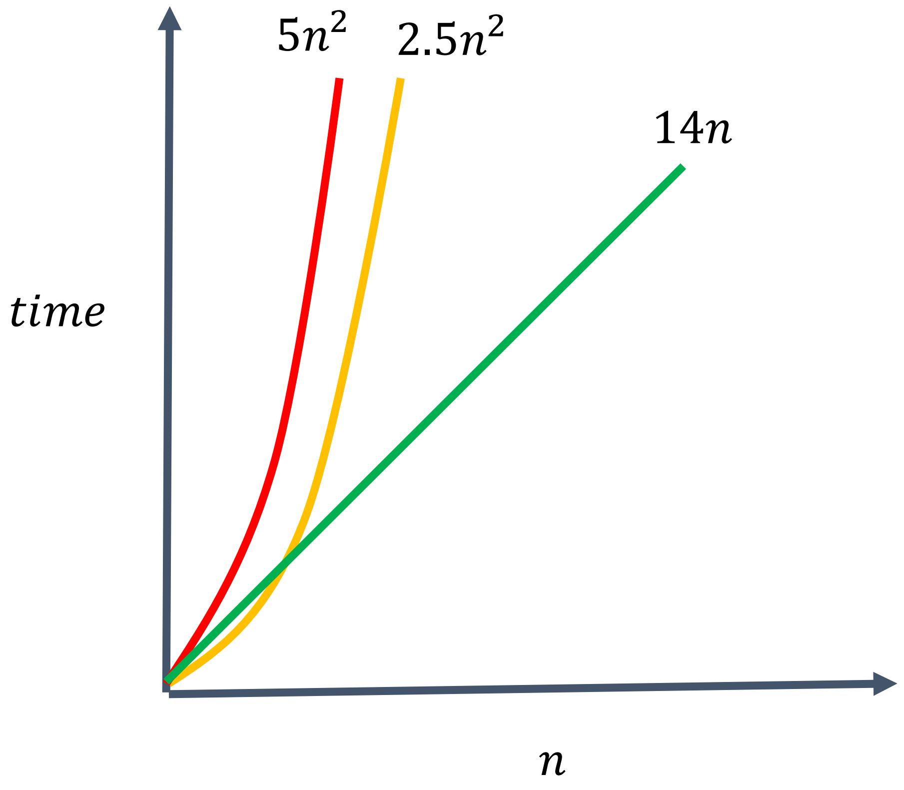
Which two functions are the most similar?

$2.5n^2$ and $5n^2$ are both **quadratic** and grow *almost at the same rate* when n grows. However, $14n$ is **linear** and grows *much slower*.

Let's simplify a little bit



Let's simplify even more!



$T(n) = 14n \qquad \sim \boldsymbol{n}$

$T(n) = 2.5n^2 \qquad \sim \boldsymbol{n^2}$

$T(n) = 5\,n^2 \qquad \sim \boldsymbol{n^2}$

Multiplicative constants don't matter!

Asymptotic notations

- These are notations that allow us to describe the running time in terms of **order of growth**.

- What matters is how fast the running time grows when n grows.

- These notations will also allow us to get rid of **lower order terms** and **multiplicative constants**.

Big-O notation

Big-O Definition

We denote by $O(g(n))$ the set of functions

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

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

Big-O notation

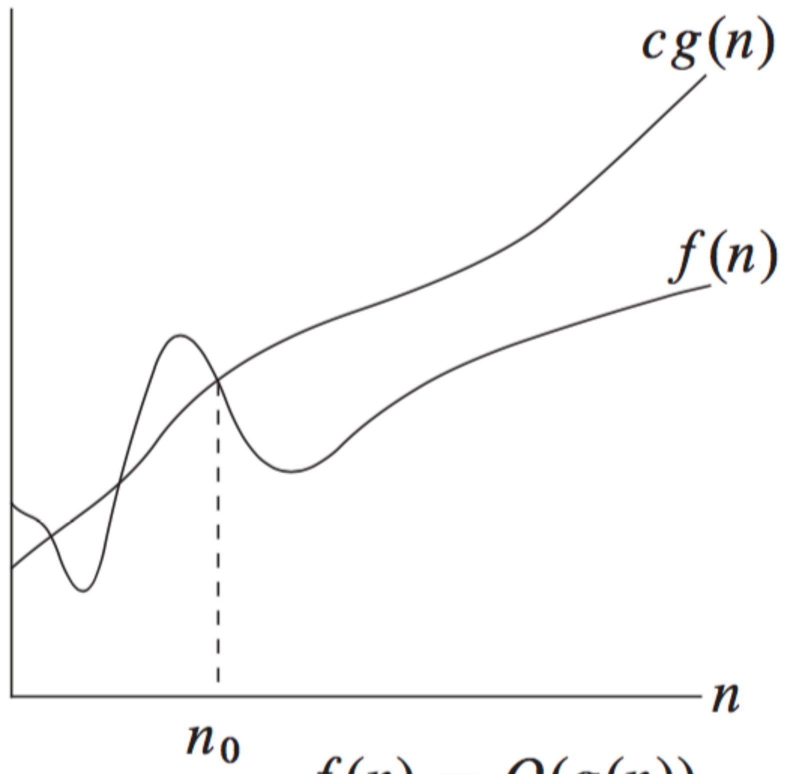
BIG-O DEFINITION

We denote by $O(g(n))$ the set of functions

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

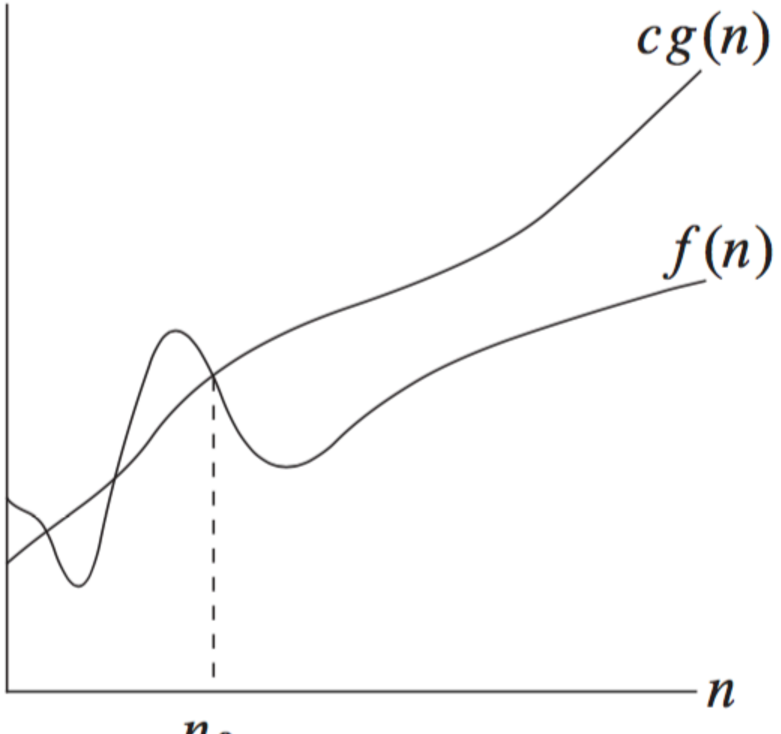
- If function $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Big-O notation



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

Big-O notation

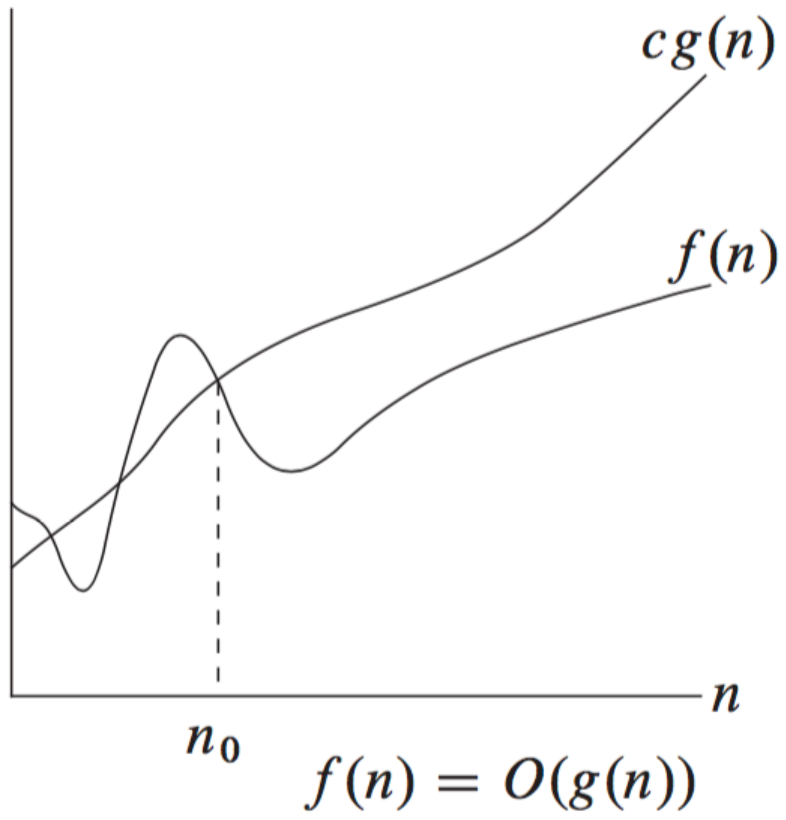


$f(n) = O(g(n))$

We may say that the constant c is helping us to deal with multiplicative factors, and n_0 helps us to deal with lower order terms.

Big-O notation

A simpler function, e.g. n^2



The one I'm interested in, e.g. $5n^2 + 9n - 11$

$$5n^2 + 9n - 11 = O(n^2)$$

Big-O notation

BIG-O DEFINITION

We denote by $O(g(n))$ the set of functions

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

- Note: For our purposes n_0 is an integer but c does not have to an integer. For example, if $c = \frac{1}{2}$ or $c = 2.8$ makes the inequality work, it's fine.

- Show that $T(n) = 5n^2 + 9n - 11 = O(n^2)$

Big-O notation

- O-notation provides **only** an **asymptotic upper bound** on the *function*
- This *function* could be the best-case running time, worst-case running time, or any other function
- Applying the **O-notation** to **worst-case running time** gives an asymptotic upper bound on running time of the algorithm

Big-O notation

- We simply say **"the time complexity of insertion-sort is $O(n^2)$"**

- This means insertion-sort takes **at most** a constant times n^2 on any input of size n, if n is large enough.

- We use the terms "running time" and "time complexity" interchangeably

Meaning of $O(1)$

- All functions that are in $O(1)$ are **constant**. For instance:
  - $f(n) = 10 = O(1)$
  - (we can just pick $c = 10$ in the definition)

Meaning of $O(1)$

- All functions that are in $O(1)$ are **constant**. For instance:
  - $f(n) = 10 = O(1)$
  - (we can just pick $c = 10$ in the definition)

- It doesn't matter how big the constant is. As long as it is **independent from the input size**, it is considered to be $O(1)$.
- When we say an algorithm takes $\boldsymbol{O(1)}$ **time** it means that it takes **constant time independent from input size**.

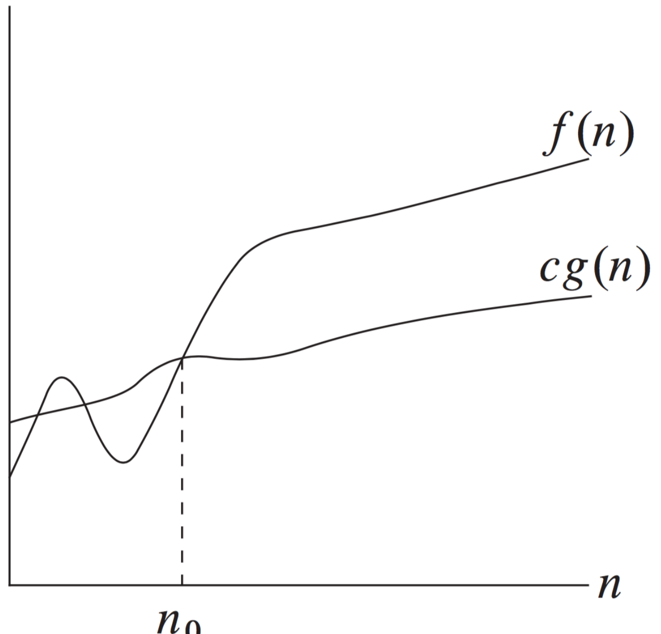
Ω-notation

Ω-NOTATION DEFINITION

We denote by $\Omega(g(n))$ the set of functions

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

- If function $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega(g(n))$



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

Ω-notation

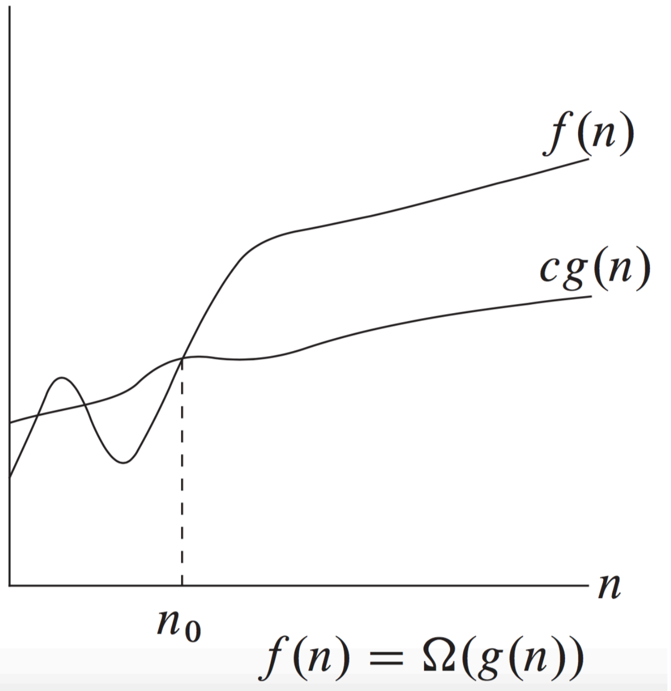
Ω-NOTATION DEFINITION

We denote by $\Omega(g(n))$ the set of functions

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

- If function $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega\big(g(n)\big)$
- **Question:** Which kind of analysis should we apply

the Ω-notation to?



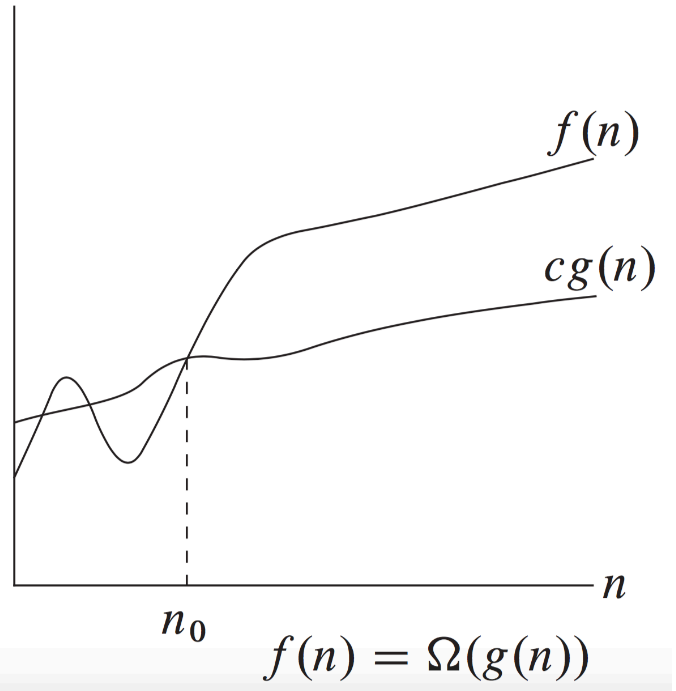
Ω-notation

Ω-NOTATION DEFINITION

We denote by $\Omega(g(n))$ the set of functions

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

- If function $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega\big(g(n)\big)$
- **Question:** Which kind of analysis should we apply the Ω-notation to?
- **Answer:** Best-case analysis since Ω allows us to simplify the lower bound we get by computing best-case running time.



Ω-notation

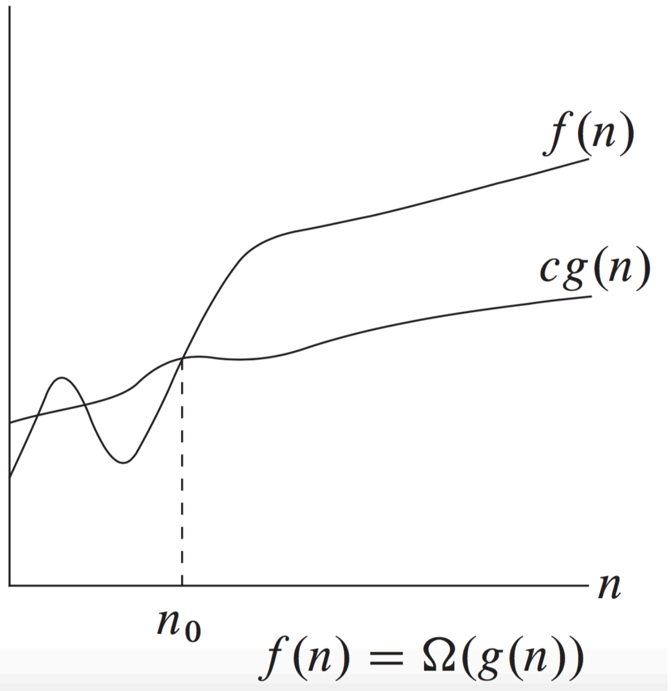
Ω-NOTATION DEFINITION

We denote by $\Omega(g(n))$ the set of functions

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

- If function $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega(g(n))$

- Show that $T(n) = 14n - 11 = \Omega(n)$



Ω-notation

- Ω-notation provides **only** an **asymptotic lower bound** on the *function*
- Applying the **Ω-notation** to **best-case running time** provides an asymptotic lower bound on the running time

- So, we can simply say **"the running time of insertion-sort is $\boldsymbol{\Omega(n)}$"**
- This means insertion-sort takes **at least** a constant times n, when n gets large enough

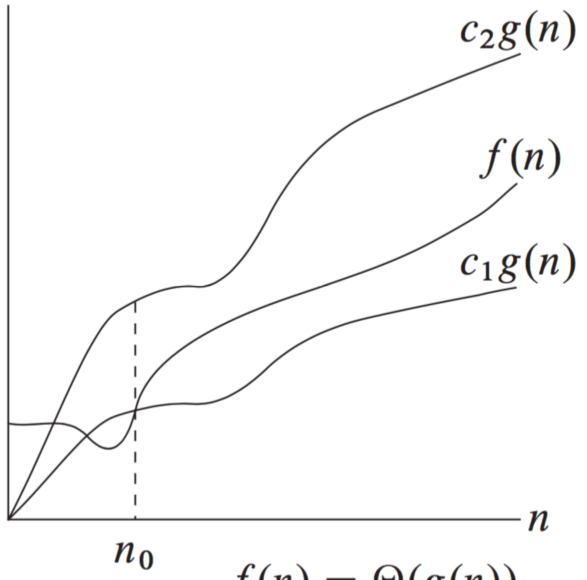
Θ-notation

Θ-NOTATION DEFINITION

We denote by $\Theta(g(n))$ the set of functions

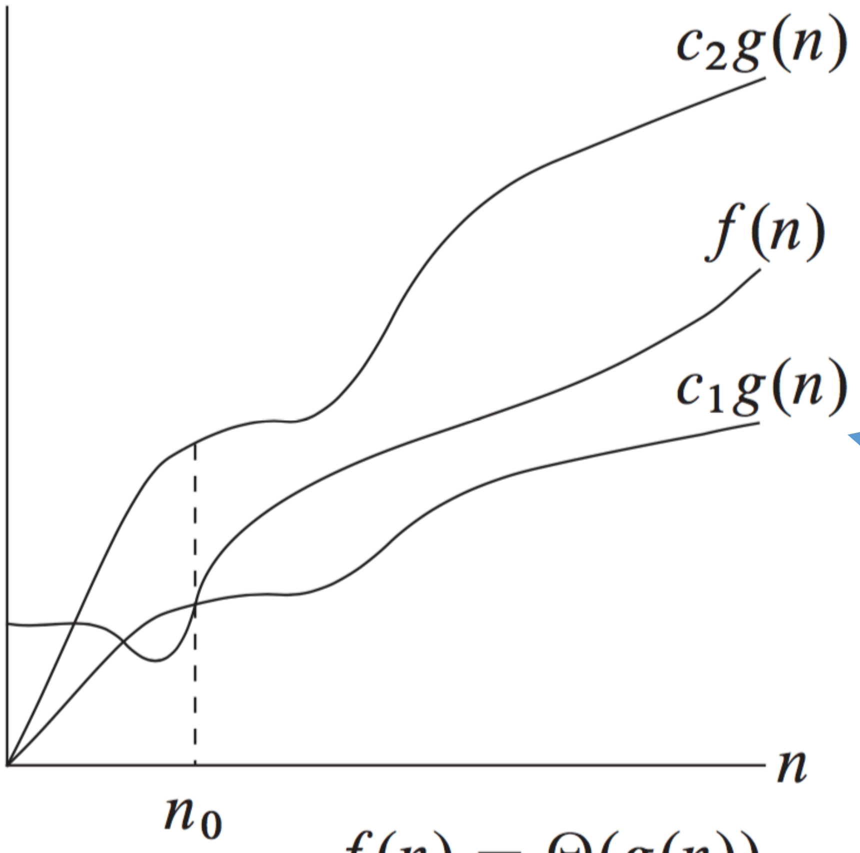
$\Theta\big(g(n)\big) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.

- If function $f(n) \in \Theta(g(n))$, we write $f(n) = \Theta\big(g(n)\big)$



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

Θ-notation



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

$c_2 n^2$

$5n^2 + 9n - 11$

$c_1 n^2$

$$5n^2 + 9n - 11 = \Theta(n^2)$$

Why Θ-notation?

Why Θ-notation?

- Assume the worst-case running time of an algorithm is $5n^2$

- We can show that

$$5n^2 = O(n^3)$$

- However,

$$5n^2 \neq \Theta(n^3)$$

Θ-notation

- Θ-notation provides an **asymptotic tight bound,** i.e. a simultaneous lower and upper bound.

- Θ-notation is a more accurate for describing the running time. But it is not always possible to use it.

Θ-notation

- Θ-notation provides an **asymptotic tight bound,** i.e. a simultaneous lower and upper bound**.**

- Θ-notation is a more accurate for describing the running time. But it is not always possible to use it.

- We cannot say **“the running time of insertion-sort is $\Theta(\boldsymbol{n}^2)$”** since the running time insertion-sorts oscillates between n and n^2 depending on the input.

Θ-notation

- Only if **there is no gap** between the lower bound and the upper bound obtained for the **best-case** and **worst-case** analysis, we can use the Θ-notation.

THEOREM

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta\big(g(n)\big)$ **if and only if**
(1) $f(n) = O\big(g(n)\big)$ and
(2) $f(n) = \Omega\big(g(n)\big)$

Result of the theorem

- So, according to the theorem, if we want to show that for some algorithm A, its running time $T(n) = \Theta(f)$, we should:

1. compute the best-case running time and show that it is $\Omega(f)$
2. compute the worst-case running time and show that it is $O(f)$

Proof of the theorem

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- First, we show $f = \Theta(g) \overset{?}{\Longrightarrow} f = O(g)\ and\ f = \Omega(g)$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- First, we show $f = \Theta(g) \overset{?}{\implies} f = O(g)\ and\ f = \Omega(g)$

$$f = \Theta(g) \implies \exists c_1, c_2, n_0\colon\ c_1 g \le f \le c_2 g$$

Proof

- $f = \Theta(g) \Leftrightarrow f = O(g)\ and\ f = \Omega(g)$

- First, we show $f = \Theta(g) \overset{?}{\Longrightarrow} f = O(g)\ and\ f = \Omega(g)$

$$f = \Theta(g) \Longrightarrow \exists c_1, c_2, n_0\colon \underbrace{c_1 g \leq f}_{\text{Big-}\Omega} \underbrace{\leq c_2 g}_{\text{Big-}O}$$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- First, we show $f = \Theta(g) \overset{?}{\Longrightarrow} f = O(g)\ and\ f = \Omega(g)$

$$f = \Theta(g) \Longrightarrow \exists c_1, c_2, n_0 \colon \underbrace{c_1 g \le f}_{\text{Big-}\Omega} \underbrace{\le c_2 g}_{\text{Big-}O}$$

- Therefore, picking c_2 and n_0 works for $f = O(g)$
- And, picking c_1 and n_0 works for $f = \Omega(g)$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- Second, we show: $f = O(g)\ and\ f = \Omega(g) \overset{?}{\Longrightarrow} f = \Theta(g)$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- Second, we show: $f = O(g)\ and\ f = \Omega(g) \overset{?}{\Longrightarrow} f = \Theta(g)$

$$f = O(g) \Longrightarrow \exists c_1, n_1 : f \leq c_1 g\ (when\ n \geq n_1)$$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- Second, we show: $f = O(g)\ and\ f = \Omega(g) \overset{?}{\Longrightarrow} f = \Theta(g)$

$$f = O(g) \Longrightarrow \exists c_1, n_1 : f \leq c_1 g\ (when\ n \geq n_1)$$

Also,

$$f = \Omega(g) \Longrightarrow \exists c_2, n_2 : c_2 g \leq f\ (when\ n \geq n_2)$$

Proof

- $f = \Theta(g) \iff f = O(g)\ and\ f = \Omega(g)$

- Second, we show: $f = O(g)\ and\ f = \Omega(g) \overset{?}{\Longrightarrow} f = \Theta(g)$

$$f = O(g) \Longrightarrow \exists c_1, n_1 : f \le c_1 g\ (when\ n \ge n_1)$$

Also,

$$f = \Omega(g) \Longrightarrow \exists c_2, n_2 : c_2 g \le f (when\ n \ge n_2)$$

- If $n \geq \max(n_1, n_2)$ both inequalities still hold.

Proof

- $f = \Theta(g) \Leftrightarrow f = O(g)\ and\ f = \Omega(g)$

- Second, we show: $f = O(g)\ and\ f = \Omega(g) \overset{?}{\Longrightarrow} f = \Theta(g)$

$$f = O(g) \Longrightarrow \exists c_1, n_1 : f \leq c_1 g\ (when\ n \geq n_1)$$

Also,

$$f = \Omega(g) \Longrightarrow \exists c_2, n_2 : c_2 g \leq f\ (when\ n \geq n_2)$$

- If $n \geq \max(n_1, n_2)$ both inequalities still hold. So, if $n_0 = \max(n_1, n_2)$

$$c_2 g \leq f \leq c_1 g\ (when\ n \geq n_0) \Longrightarrow f = \Theta(n)$$

Analogy

$$
\begin{array}{llll}
 & O & \leq & \\
f(n) = & \Theta & = & g(n) \\
 & \Omega & \geq &
\end{array}
$$

We mean only in the asymptotic sense!

Analogy

$$
f(n) = \begin{array}{ccc} ? & & < \\ O & & \leq \\ \Theta & & = \\ \Omega & & \geq \\ ? & & > \end{array} \; g(n)
$$

What about strictly less and strictly greater?

Analogy

	little-o	$<$	
	O	$\leq$	
$f(n) =$	Θ	$=$	$g(n)$
	Ω	$\geq$	
	little-ω	$>$	

Two more asymptotic notations

Little-o and Little-ω

$o\big(g(n)\big) = \{f(n)$: for **any constant c** , there exists some constant n_0 such that $\mathbf{0} \leq \boldsymbol{f}(\boldsymbol{n}) < \mathbf{c}\boldsymbol{g}(\boldsymbol{n})$ for all $n \geq n_0\}$.

$\omega\big(g(n)\big) = \{f(n)$: for **any constant c** , there exists some constant n_0 such that $\mathbf{0} \leq \mathbf{c}\boldsymbol{g}(\boldsymbol{n}) < \boldsymbol{f}(\boldsymbol{n})$ for all $n \geq n_0\}$.

Two more asymptotic notations

Alternative Definitions

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Two more asymptotic notations

ALTERNATIVE DEFINITIONS

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Note: The limit definition is easier to use for actual functions but the normal definition is better for proving mathematical properties about notations.

Examples

$$n^{1.99} = o(n^2)$$

$$\lim_{n \to \infty} \frac{n^{1.99}}{n^2} = 0$$

- This shows that even a small difference in the degree of polynomials is very significant in terms of growth.

Examples

$$n^{10} = o(2^n)$$

$$\lim_{n\to\infty} \frac{n^{10}}{2^n} = 0$$

- This shows that an exponential function grows strictly faster than a polynomial.

Examples

$$\log^{10} n = o(n)$$

$$\lim_{n \to \infty} \frac{\log^{10} n}{n} = 0$$

- This shows that a polynomial grows strictly faster than a logarithmic function.

- When we don't write the base of log, it's 2

Examples

$$\log^{10} n = o(n)$$

$$\lim_{n \to \infty} \frac{\log^{10} n}{n} = 0$$

- This shows that a polynomial grows strictly faster than a logarithmic function.
- As a summary,

logarithmic $<$ **polynomial** $<$ **exponential**

Examples

Exercise: Justify that $a^n = o(b^n)$ when a, b are constants and $1 < a < b$. An example is $2^n = o(4^n)$

Examples

Exercise: Justify that $a^n = o(b^n)$ when a, b are constants and $1 < a < b$. An example is $2^n = o(4^n)$

Note: Section 3.2 provides enough mathematical background to understand the growth of functions and some useful limit properties.

A summary

- Functions we usually encounter, c is a constant:

notation	**name**
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Some common functions

- Functions we usually encounter, c is a constant:
- $(\log(n))^c = \log^c n$

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Some properties of asymptotics

- **All** asymptotic notations are **transitive**, e.g.:
- $f = O(g)\ and\ g = O(h) \rightarrow f = O(h)$

Some properties of asymptotics

- **All** asymptotic notations are **transitive**, e.g.:
- $f = O(g)\ and\ g = O(h) \rightarrow f = O(h)$

- $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ are **reflexive**, e.g.:
- $f = \Omega(f)$

Some properties of asymptotics

- **All** asymptotic notations are **transitive**, e.g.:
- $f = O(g)\ and\ g = O(h) \rightarrow f = O(h)$

- $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ are **reflexive**, e.g.:
- $f = \Omega(f)$

- $\boldsymbol{\Theta}$ is **symmetry**, e.g.:
- $f = \Theta(g) \rightarrow g = \Theta(f)$

Some properties of asymptotics

- **All** asymptotic notations are **transitive**, e.g.:
- $f = O(g)\ and\ g = O(h) \to f = O(h)$

- $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ are **reflexive**, e.g.:
- $f = \Omega(f)$

- $\boldsymbol{\Theta}$ is **symmetry**, e.g.:
- $f = \Theta(g) \to g = \Theta(f)$

- **Transpose symmetry:**
- If $f = O(g) \to g = \Omega(f)$ (true for o and ω as well)

Some properties of asymptotics

- **All** asymptotic notations are **transitive**, e.g.:
- $f = O(g)\ and\ g = O(h) \rightarrow f = O(h)$

- $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ are **reflexive**, e.g.:
- $f = \Omega(f)$

- $\boldsymbol{\Theta}$ is **symmetry**, e.g.:
- $f = \Theta(g) \rightarrow g = \Theta(f)$

- **Transpose symmetry:**
- If $f = O(g) \rightarrow g = \Omega(f)$ (true for o and ω as well)

CLRS page 51-52 lists all the properties of asymptotic notations.

Space complexity

- The **space complexity** of an algorithm is **the amount of memory** that the algorithm needs to execute correctly.

- In RAM model we assume each number can ***fit into a single word***.

- Therefore, we can say that **each variable** and **each element in the input** takes **only $O(1)$ memory (or space)**

Space complexity

- **Question:** What is the space complexity of insertion-sort?

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

Space complexity

- **Question:** What is the space complexity of insertion-sort?
- **Answer:** $O(n)$ where $n = A.length$. Because it needs to work with an array of size n, and the number auxiliary variables is constant.

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

Space complexity

- **Question:** What is the space complexity of this for loop?

DUMMY-LOOP(n)

```
1  for i = 1 to n
2      i = i+1
```

Space complexity

- **Question:** What is the space complexity of this for loop?
- **Answer:** $O(1)$ since it is only working with a constant number of variables.

DUMMY-LOOP(n)

1. **for** $i = 1$ **to** n
2. $\quad i = i+1$

Input size

- Stating the input size **depends** on the **problem.**

- For problems like sorting or problems that we receive an array of size n as input, the input size is considered to be n.

- However, for **numerical problems** the **number of bits** required to **represent the input** will determine the input size.

Input size

- **Question:** What is the input size for the following problem?

PRIMALITY-TESTING: Given n, determine if n is prime.

Input size

- **Question:** What is the input size for the following problem?

PRIMALITY-TESTING: Given n, determine if n is prime.

- **Answer:** $\log n$, since we can represent the number with $\log n$ bits