

Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

About Me

Ekkapot Charoenwanit

Office:

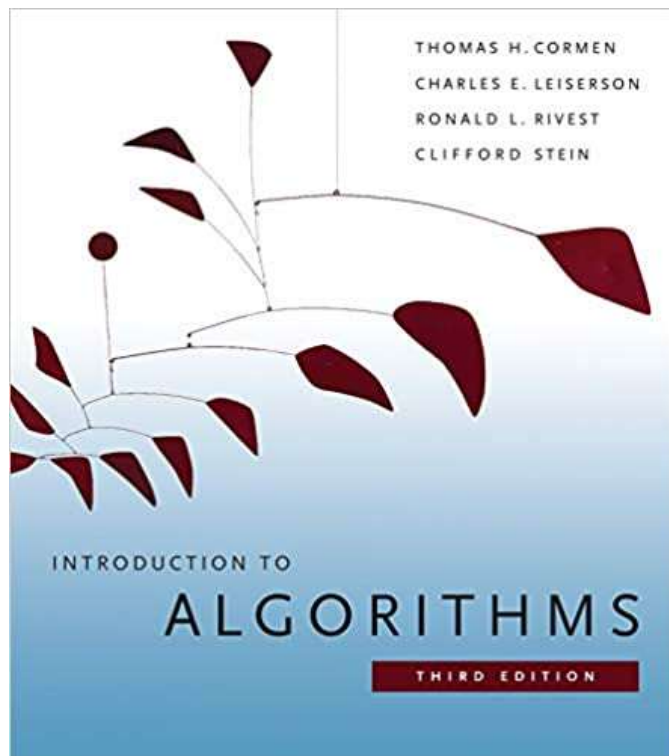
802/806

you will more likely find me in 806

Office Hours:

Wednesday Afternoon 13:00-16:30

Textbook known as *CLRS*



The *third edition* is recommended, but the *second edition* is also fine.

***Available at the main library

Course Logistics

Weekly Assignments	50%
Problem Sets	
Programming Labs	
Midterm Exam	20%
Final Exam	30%

*****Turning in an assignment late = 10% off per day**



Grading Scale:

A [80-100]

B [70-80)

C [60-70)

D [50-60)

F $(-\infty, 50)$

Course Contents

- Asymptotic Analysis
- Induction and Recurrence Relations
- Data Structures
- Searching and Sorting Algorithms
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Graph Algorithms
- State Space Search
- NP-Completeness
- Approximation Algorithms
- Randomised Algorithms
- Linear Programming
- More advanced topics if time allows

The Hierarchy of Abstraction in Computing



This course spans these 3 abstraction layers:

- Problem
- Algorithm
- Programming Language

Our focus will be on the **Algorithm** layer.

Lecture 1: Algorithm Analysis and Asymptotic Notations

Definition of Algorithms

An algorithm is a finite, unambiguous description for a sequence of computational steps to solve a computational problem.

- ***Finite*** in this context means the algorithm must eventually terminate.

Algorithmic Complexity

How *efficient* an algorithm is can be measured by its *algorithmic complexity*

- Time Complexity (Temporal)

How many computational step units are required?

How much time does the algorithm need?

- Space Complexity (Spatial)

How much space does the algorithm need?

This course will focus more on Time Complexity.

Performance Analysis of Algorithms

There are generally **two** methods for analyzing algorithms' performance:

- **Experimental Analysis:**
 - Run code on a computer
 - Measure the running time for different problem sizes
 - Plot the result as a graph
- **Mathematical Analysis:**
 - Express the number of elementary steps parameterized by the problem size

Selection Sort

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

Experimental Analysis

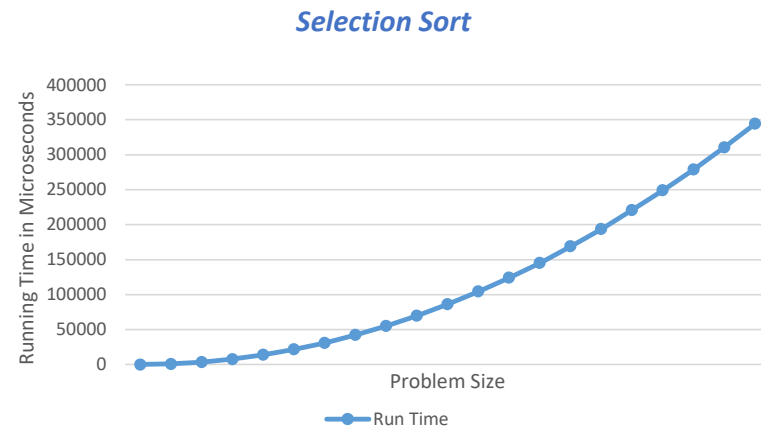
Running a C++ implementation of selection sort for $n = 0$ to **20000** on my workstation:

Ubuntu 18

Intel Core-i7-7700 CPU @3.60GHz

with 16 GB of RAM

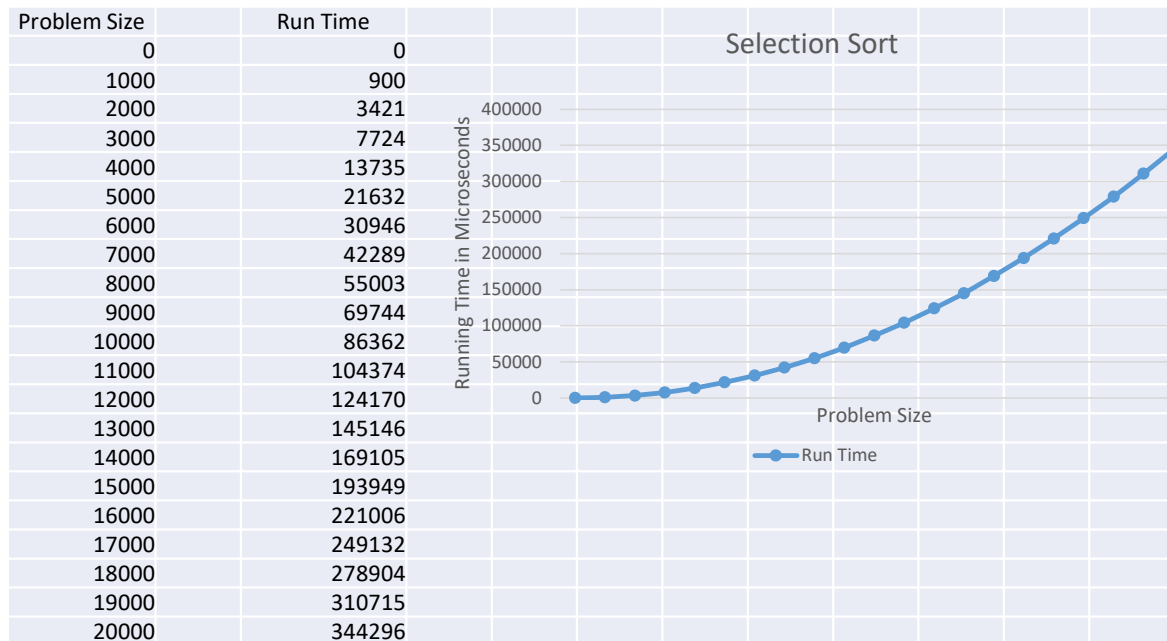
yielded the plot on the right.



The running time appears to be quadratic in problem size:

$$T(n) = 8.60 \times 10^{-4}n^2$$

Experimental Analysis



- The code was run for different numbers of elements n between 0 and 20,000 with a step increase of 1,000.
- Each problem size was run for 100 times and the running times were averaged.

Mathematical Analysis

The time complexity of an algorithm can be determined by the total number of *elementary operations* executed.

$$Time \propto \#Elementary\ Operations$$

Elementary Operations are operations whose execution time is bounded by a *constant*, which depends on

- Programming Language
- Compiler
- Machine

Approaches to Counting Elementary Operations

- Count every elementary operations
 - impractical
- Count only **representative** elementary operations
 - executed the most number of times
 - called **barometer operations**

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

What are the barometer operations in the code of selection sort shown on the right?

Approaches to Counting Elementary Operations

- Count every elementary operations
 - impractical
- Count only **representative** elementary operations
 - executed the most number of times
 - called **barometer operations**

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

Barometer : $d[i] > d[maxI]$

Approaches to Counting Elementary Operations

Count the number of times $d[i] > d[maxI]$ is executed

Inner Loop

$$\sum_{i=2}^k 1$$

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

Approaches to Counting Elementary Operations

Count the number of times $d[i] > d[maxI]$ is executed

Outer Loop

$$\sum_{k=2}^n$$

Inner Loop

$$\sum_{i=2}^k 1$$

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

Approaches to Counting Elementary Operations

Count the number of times $d[i] > d[maxI]$ is executed

$$\sum_{k=n}^2 \sum_{i=2}^k 1 = \sum_{k=1}^{n-1} \sum_{i=1}^{k-1} 1 = \sum_{k=1}^{n-1} (k-1) = \frac{(n-1)(n)}{2}$$

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

Therefore, the running time of selection sort is proportional to $\frac{(n-1)(n)}{2}$.

Complexity Growth of Selection Sort

```
1: procedure SELECTIONSORT( $d, n$ )
2:   for  $k = n$  ;  $k > 1$  ;  $k --$  do
3:      $maxI = 1$ 
4:     for  $i = 2$  ;  $i \leq k$  ;  $i ++$  do
5:       if  $d[i] > d[maxI]$  then
6:          $maxI = i$ 
7:        $d[k] \iff d[maxI]$ 
```

We say that the time complexity of selection sort exhibits a ***quadratic growth*** in the number of elements n .

Complexity Growth

The complexity of an algorithm is generally represented as a function of its *input size* (and possibly the values of other parameters).

Such functions are restricted to *real-valued functions* $f(n): \mathbb{N} \mapsto \mathbb{R}$ defined on the *non-negative integers* that are *eventually* positive as there exists an integer n_0 such that $f(n) > 0$ for all $n \geq n_0$.

The growth function $f(n)$

- gives a simple characterization of the algorithm's efficiency
- gives a simple relative efficiency comparison with other algorithms

Asymptotic Analysis

Benefits of Asymptotic Analysis

- Provides machine-independent analysis
- Abstracts away from implementation details
- Focuses only on the dominating factors

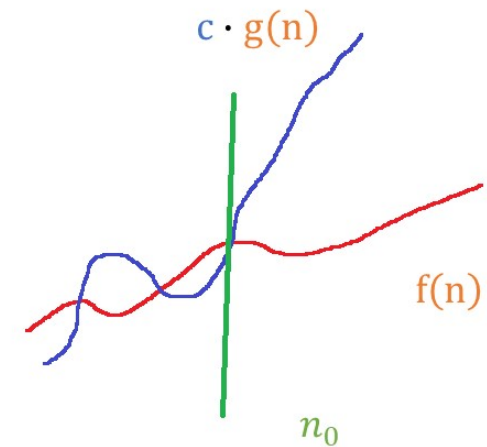
For *sufficiently large* input sizes, a linear-time algorithm with a moderately big constant overhead will eventually run faster than a quadratic-time one with a relatively small constant overhead.

Definition of Big O

A function $f(n)$ is in $O(g(n))$ if and only if

$$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \mapsto f(n) \leq c g(n))$$

$f(n)$ is bound from above by $g(n)$ up to a constant factor c for all sufficiently large n beyond n_0 .



Big-O Notation

$O(g(n))$ is a set of functions taking a *real natural number* as input and returning a *real number*.

$$O(g(n)) = \{f: \mathbb{N} \mapsto \mathbb{R} : \\ \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \rightarrow f(n) \leq c g(n))\}$$

Therefore,

$$f(n) = O(g(n)) \text{ actually means } f(n) \in O(g(n))$$

But it is also common in the literature to use $f(n) = O(g(n))$.

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

Proving that $f(n) = O(g(n))$ is about intelligently picking c and n_0 .

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

Converting the above notation into the corresponding inequality gives:

$$2n^2 \leq cn^2 \text{ for all } n \geq n_0$$

All we have to do is to show that there is at least a pair of (c, n_0) that satisfies the inequality above.

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

$$2n^2 \leq cn^2 \text{ for all } n \geq n_0$$

Since we know that n^2 is non-negative, choosing $n > 0$ and dividing both sides by n^2 gives:

$$c \geq 2$$

Therefore, the above inequality will hold if $n > 0$ and $c \geq 2$

Therefore, we can choose $n_0 = 1$ and $c = 2$ ■

NB: we could have chosen other values such as $n_0 = 2$ and $c = 10$.

A little harder claim

Show that $2n^2 + 3 = O(n^2)$.

Converting the notation into the corresponding inequality gives:

$$2n^2 + 3 \leq cn^2 \text{ for all } n \geq n_0$$

The easy but *impulsive* way:

Try $c = 5$ and solve for n :

$$2n^2 + 3 \leq 5n^2$$

$$3 \leq 3n^2$$

$$1 \leq n^2$$

$$1 \leq n \quad (\text{NB: the negative values are ignored.})$$

Therefore, we choose $c = 5$ and $n_0 = 1$. ■

Yet another little harder claim

The more systematic way:

Solve for n :

$$2n^2 + 3 \leq cn^2$$

Rearranging gives

$$3 \leq (c - 2)n^2$$

Assuming $c - 2 > 0 \rightarrow c > 2$,

$$\frac{3}{c-2} \leq n^2$$

$$\sqrt{\frac{3}{c-2}} \leq n$$

Now we can choose $c > 2$ that makes n look simple.

$c = 5$ leads to: $1 \leq n$

Therefore, the most obvious choice of n_0 is $n_0 = 1$.

Therefore, we choose $c = 5$ and $n_0 = 1$ ■

Not hard enough?

Show that $7n^2 + 1000n = O(n^2)$.

$$7n^2 + 1000n \leq cn^2 \text{ for all } n \geq n_0$$

$$(c-7)n^2 \geq 1000n$$

Assume $c - 7 > 0 \rightarrow c > 7$.

Consider only $n > 0$ and divide both sides by n .

$$(c - 7)n \geq 1000$$

$$n \geq \frac{1000}{c-7}$$

Therefore, we choose $c = 8$ and $n_0 = 1000$ ■

Not hard enough?

Show that $7n^2 - 1000n = O(n^2)$.

$$\begin{aligned} 7n^2 - 1000n &\leq cn^2 \text{ for all } n \geq n_0 \\ -1000n &\leq (c - 7)n^2 \end{aligned}$$

Assume $c - 7 > 0 \rightarrow c > 7$.

Consider only $n > 0$ and divide both sides by n .

$$\begin{aligned} -1000 &\leq (c - 7)n \\ n &\geq \frac{-1000}{c-7} \quad (\text{NB: the sign does not flip b/c } c - 7 > 0) \\ n &\geq \frac{1000}{7-c} \end{aligned}$$

Therefore, we choose $c = 8$, leading to $n \geq -1$.

Since $n > 0$, we choose $c = 8$ and $n_0 = 1$ ■

Disproving $f(n) \notin O(g(n))$

We must show that the negation of $\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \rightarrow f(n) \leq cg(n))$ holds.

The negation is:

$$\forall c \in \mathbb{R}^+ \forall n_0 \in \mathbb{N} (\exists n \in \mathbb{N} : n \geq n_0 \wedge f(n) > cg(n))$$

Disproving $f(n) \notin O(g(n))$

Show that $n^2 \notin O(n)$

$$\forall c \in \mathbb{R}^+ \forall n_0 \in \mathbb{N} (\exists n \in \mathbb{N} : n \geq n_0 \wedge n^2 > c \cdot n)$$

Solve for n :

$$n \geq n_0 \text{ and } n^2 > c \cdot n$$

$$n \geq n_0 \text{ and } n > c$$

$$n > \max(n_0, c)$$

Therefore, we can choose $n = \max(n_0, \lceil c \rceil) + 1$ ■

Comparing Polynomial Functions

For a **polynomial** $f(n)$, it is easy to figure out a Big-O class $f(n)$ belongs to.

- It is the term with the **highest degree** !!!
 - $f(n) = 3n^5 + 40n^2 + 100n + 2 \in O(n^5)$
- It is perfectly valid to use a more slacking upper bound
 - $f(n) \in O(n^6)$
 - $f(n) \in O(n^{1,000,000})$

*****Exercise:** Show that $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \in O(n^k)$ if $a_k > 0$ and $k > 0$.

Comparing Polynomial Functions

n	$n/2$	$n^2/2$	$\frac{(n-1)(n)}{2}$	n^2
10	5	50	45	100
1,000	500	500,000	499,500	1,000,000
10,000	5000	50,000,000	49,995,000	100,000,000
100,000	50,000	5,000,000,000	4,999,950,000	10,000,000,000

Table comparing linear and quadratic growth

Properties of Big-O

- $f(n) \in O(cf(n))$
 - Big-O is conserved under multiplicative constant.
- $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
 - Big-O is transitive.

and many more ...

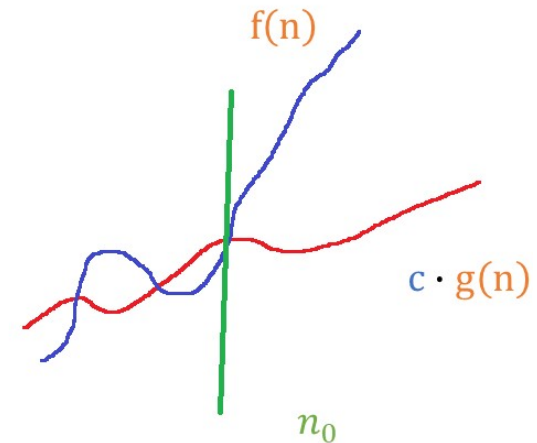
*****Exercise:** Show that the properties above always hold.

Definition of Big Omega

A function $f(n)$ is in $\Omega(g(n))$ if and only if

$$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$$

$f(n)$ is bound from below by $cg(n)$ for all sufficiently large n beyond n_0 .



Big-Omega Notation

$\Omega(g(n))$ is a set of functions taking a *real natural number* as input and returning a *real number*.

$$\Omega(g(n)) = \{f: \mathbb{N} \mapsto \mathbb{R} : \\ \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \rightarrow f(n) \geq c g(n))\}$$

Therefore,

$$f(n) = \Omega(g(n)) \text{ actually means } f(n) \in \Omega(g(n))$$

But it is also common in the literature to use $f(n) = \Omega(g(n))$.

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

Proving that $f(n) = \Omega(g(n))$ is about intelligently picking c and n_0 .

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

Converting the above notation into the corresponding inequality gives:

$$2n^2 \geq cn \text{ for all } n \geq n_0$$

All we have to do is to show that there is at least a pair of (c, n_0) that satisfies the inequality above.

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

$$2n^2 \geq cn \text{ for all } n \geq n_0$$

Assuming $n > 0$ and dividing both sides by n gives:

$$\begin{aligned} 2n &\geq c \\ n &\geq \frac{c}{2} \end{aligned}$$

Therefore, the above inequality will hold if $n \geq \max(1, \lceil \frac{c}{2} \rceil)$ and $c > 0$.

Therefore, we can choose $n_0 = 1$ and $c = 2$ ■

NB: we could have chosen other values such as $n_0 = 5$ and $c = 10$.

Big-Omega of Polynomials

***Exercise:

$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \in \Omega(n^k)$ if $a_k > 0$
and $k > 0$.

Properties of Big-Omega

- $f(n) \in \Omega(cf(n))$
 - Big-Omega is conserved under multiplicative constant.
- $f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$
 - Big-Omega is transitive.

and many more ...

*****Exercise:** Show that the properties above always hold.

Definition of Big Theta

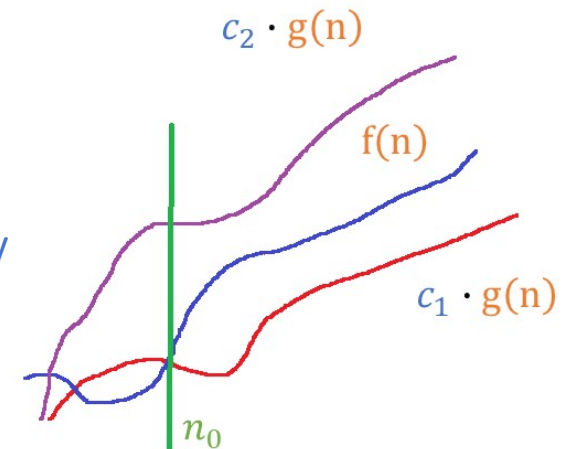
A function $f(n)$ is in $\Theta(g(n))$ if and only if

$$\exists c_1 \in \mathbb{R}^+ \exists c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} (\forall n \in \mathbb{N} : n \geq n_0 \rightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$$

$f(n)$ is sandwiched
between $c_1 g(n)$ and
 $c_2 g(n)$ for all sufficiently large n beyond n_0

$f(n)$ grows at the same rate as $g(n)$ in the sense that $f(n)$ is eventually squeezed between two constant multiples of $g(n)$.

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n))$$



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

Show that $\frac{n^2}{2} - 3n = \Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2 \text{ for all } n \geq n_0$$

Dividing both sides by $n^2 > 0$ gives:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

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

Lower Bound:

$$c_1 \leq \frac{1}{2} - \frac{3}{n}$$

$$c_1 \leq \frac{n-6}{2n}$$

$$n(1 - 2c_1) \geq 6$$

Assuming $1 - 2c_1 > 0 \rightarrow c_1 < \frac{1}{2}$

$$n \geq \frac{6}{1-2c_1}$$

Choose $c_1 = \frac{1}{4}, n_{01} = 12$.

We have just proven $f(n) = \Omega(g(n))$ as a by-product.

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

Upper Bound:

$$\frac{1}{2} - \frac{3}{n} \geq c_2$$

$$c_2 \geq \frac{n-6}{2n}$$

$$n(1 - 2c_2) \geq 6$$

Assuming $1 - 2c_2 < 0 \rightarrow c_2 > \frac{1}{2}$

$$n \geq \frac{6}{1-2c_2}$$

Choose $c_2 = 1, n_{02} = 1$.

We have just proven $f(n) = O(g(n))$ as a by-product.

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

Therefore, $c_1 = \frac{1}{4}$, $c_2 = 1$, $n_0 = \max(n_{01}, n_{02}) = \max(12, 1) = 12$ ■

Properties of Big Theta

- $f(n) \in \Theta(f(n))$ (Reflexive)
 - $f(n)$ has the same order as itself.
- $f(n) \in \Theta(g(n)) \Rightarrow g(n) \in \Theta(f(n))$ (Symmetric)
 - $f(n)$ has the same order as $g(n)$, then $g(n)$ has the same order as $f(n)$.
- $f(n) \in \Theta(g(n)) \wedge g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$ (Transitive)
 - $f(n)$ has the same order as $g(n)$, and $g(n)$ has the same order as $h(n)$, then $f(n)$ has the same order as $h(n)$.

Orders of Growth

$$\begin{aligned} O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \\ \subset O(n^3) \subset O(2^n) \subset O(3^n) \subset O(n!) \end{aligned}$$

Proofs involving ***non-polynomial*** functions require ***mathematical induction***.

We will cover induction in the next lecture.