ANALYSIS OF ALGORITHMS

Chapter2/Part2

Prepared by: Enas Abu Samra

KEY POINTS OF CHAPTER2

- Analysis of Algorithms.
- Calculating the Running Time of a program.
- Order of Growth.
- Best Case, Average Case, Worst Case.
- Analyzing the Time Efficiency of non-recursive Algorithms.
- Analyzing the Time Efficiency of recursive Algorithms.

What is the Asymptotic Notations?

TIME COMPLEXITY -ASYMPTOTIC ANALYSIS

- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of **asymptotic notations.**

- An algorithm may **not** have the same performance for different types of inputs. **With the increase in the input size, the performance will change.**

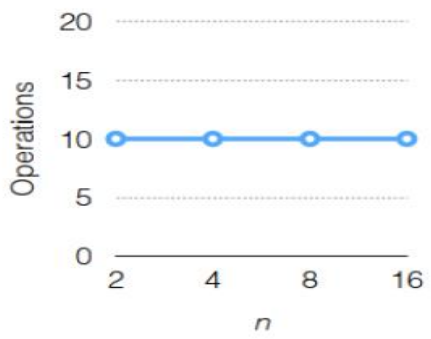
- The study of change in performance of the algorithm with the change in the order of the input size is defined as **asymptotic analysis.**

ASYMPTOTIC ANALYSIS

Example. Assume $n \geq 10$ is the size of an array and we are interested in counting the number of array accesses an algorithm performs.

? How quickly does the number operations performed grows when the input size grows (when the array size grows)?

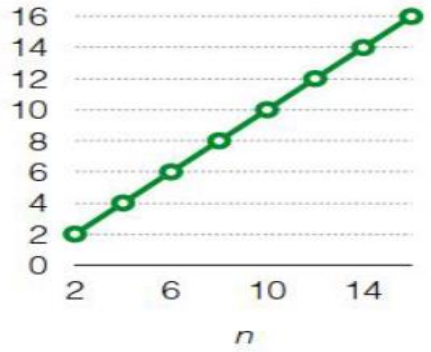
```
for (i=0; i<10; i++)
    sum += a[0];
```



No growth!

Always 10, regardless of the array size

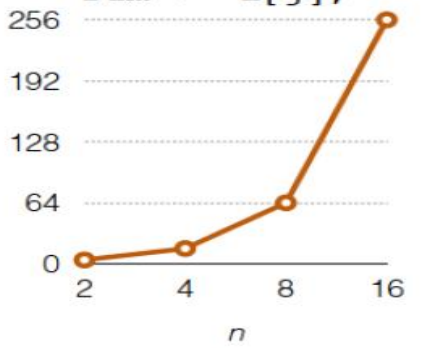
```
for (i=0; i<n; i++)
    sum += a[i];
```



LInear growth!

operations **double** when the array size doubles

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
      sum += a[j];
```



Quadratlc growth!

operations **quadruple** when the array size doubles

ASYMPTOTIC NOTATIONS

- **Asymptotic notations** are the mathematical notations used to describe the **running time of an algorithm** when the input tends towards a particular value or a limiting value.

- There are mainly three asymptotic notations:
  - ✓ Big-O notation → O
  - ✓ Omega notation → Ω
  - ✓ Theta notation → Θ

ASYMPTOTIC NOTATIONS

The most common Asymptotic Notation for calculating Algorithm's complexities are:

➤ **Big-O Notation**

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the **worst-case complexity** of an algorithm.

➤ **Big- Ω Notation**

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the **best-case complexity** of an algorithm.

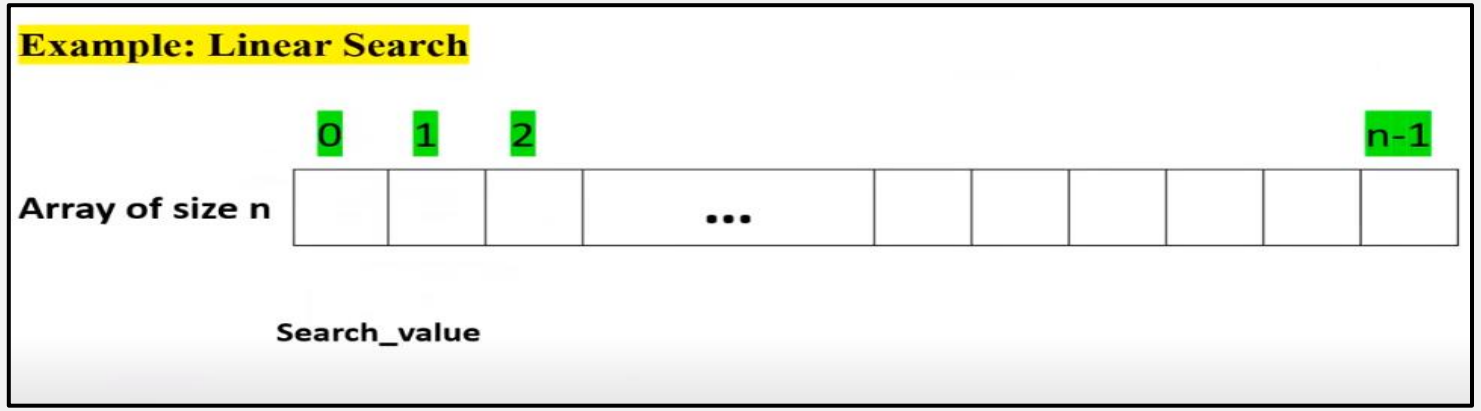
➤ **Big- ΘNotation**

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case complexity** of an algorithm.

Why we use the Asymptotic Notations?

CASE STUDY (1)

- Suppose we have values in this array, and we want to search for **number 3**.



- ✓ If number 3 is in the **first place**, how many steps does it take to reach it? ➔ **1 step.**
- ✓ If number 3 is in the **middle of array**, how many steps does it take to reach it? ➔ **(n-1)/2 steps.**
- ✓ If number 3 is in the **last place**, how many steps does it take to reach it? ➔ **(n-1) steps.**

CASE STUDY(1)

- ✓ If number 3 is in the **first place**, we called this case ➔ **Best Case.** (minimum number of steps).
- ✓ If number 3 is in the **middle of array**, we called this case ➔ **Average Case.** (average number of steps).
- ✓ If number 3 is in the **last place**, we called this case ➔ **Worst Case.** (maximum number of steps).

- **How to represent these cases in algorithm analysis?**

  Using the asymptotic notations.

  Note:

  In the previous case (Linear Search), the complexity is :

  Best case ➔ $\Omega(n)$, **Average case** ➔ $\Theta(n)$, **Worst case** ➔ $O(1)$

CASE STUDY (2) – WHO IS BETTER?

A

```
for (int i=0; i < 50 * n; i++)
  op();
```

$50n$

B

```
for (int i=0; i < n * n; i++)
  op();
```

n^2

We expressed the number of operations performed by each program as $T_A(n) = 50n$ and $T_B(n) = n^2$, which are two functions that have different values depending on the value of the input size n.

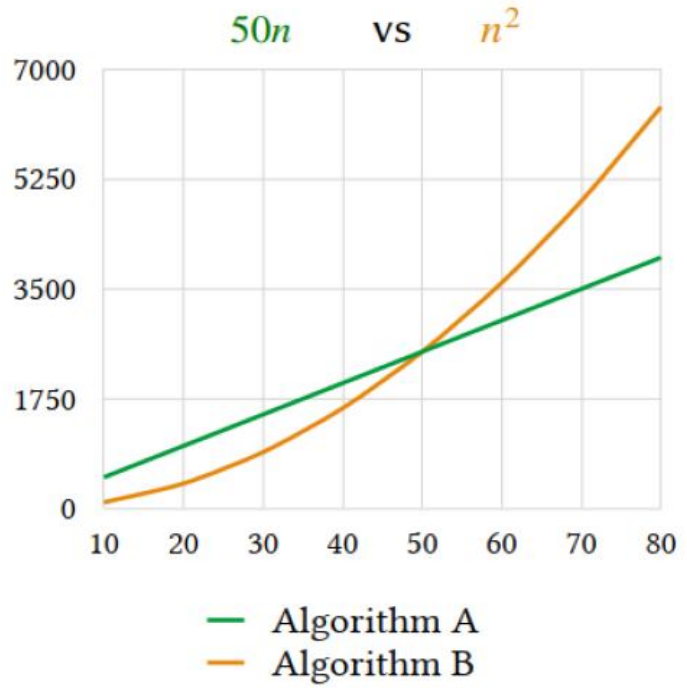
? Which function represents a better running time (less performed operations)?

CASE STUDY (2) – WHO IS BETTER?

	$50n$	n^2
n	Algorithm A	Algorithm B
10	500	100
20	1000	400
30	1500	900
40	2000	1600
50	2500	2500
60	3000	3600
70	3500	4900
80	4000	6400
90	4500	8100

$50n$ is worse

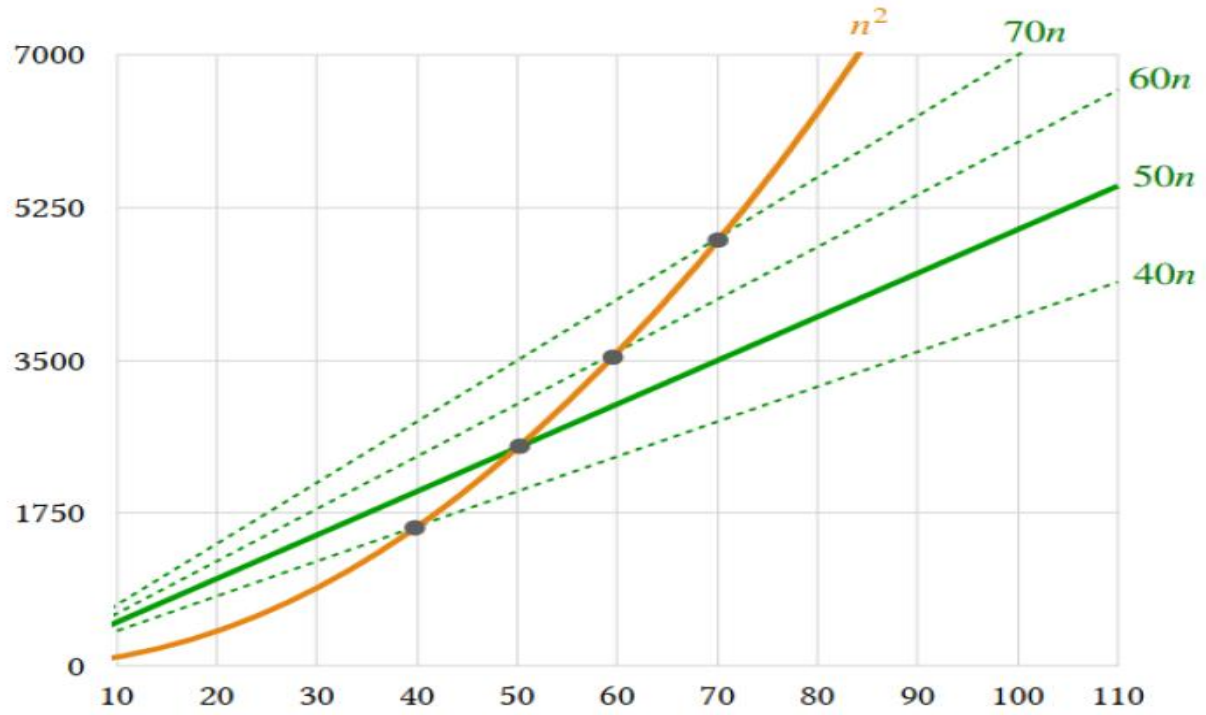
n^2 is worse



! n^2 grows faster than $50n$.

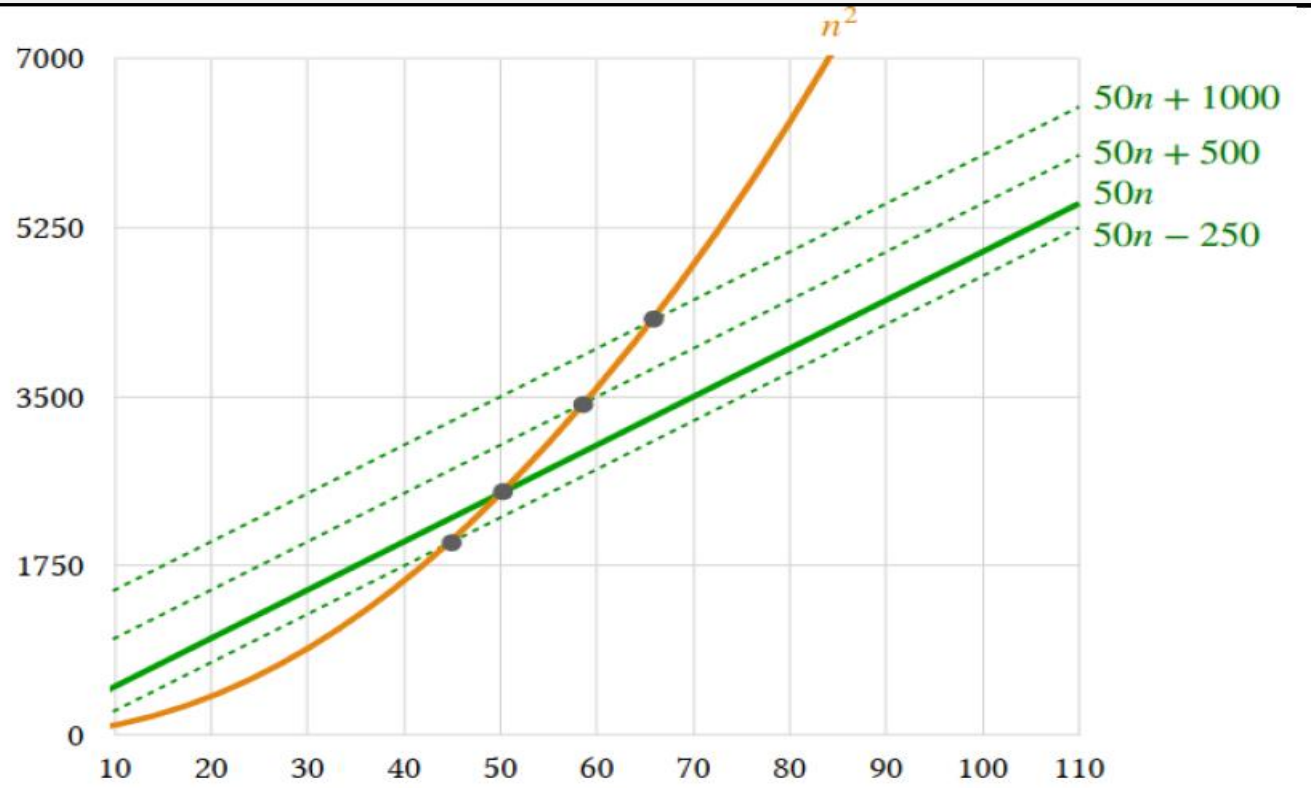
n^2 must at some point become worse (perform more operations) than $50n$ forever (when n > 50 in this case)

ORDER OF GROWTH –EXAMPLE(1)



! n^2 will at some point exceed cn regardless of what the value of c is.

ORDER OF GROWTH-EXAMPLE(2)



! n^2 will at some point exceed $cn + a$ regardless of what the values of c and a are.

ORDER OF GROWTH

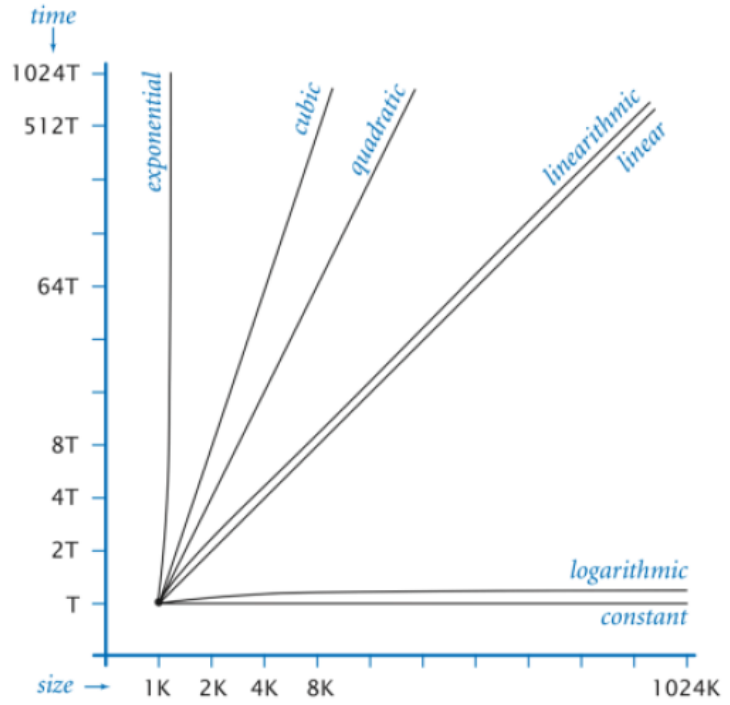
Order of Growth of the running time: How quickly the running time of an algorithm grows as the input size grows.

Examples: log n, n, $n2$, $n3$, $2n$, etc.

	order of growth name	function
	constant	1
good	logarithmic	$\log(n)$
good		$\sqrt{n}$
fine	linear	n
fine	linearithmic	$n \log(n)$
fine		$n\sqrt{n}$
bad	quadratic	n^2
bad	cubic	n^3
horrible	exponential	2^n
horrible	exponential	3^n
horrible	factorial	$n!$

ORDER OF GROWTH

	order of growth name	function
good	constant	1
good	logarithmic	$\log(n)$
good		$\sqrt{n}$
fine	linear	n
fine	linearithmic	$n \log(n)$
fine		$n\sqrt{n}$
bad	quadratic	n^2
bad	cubic	n^3
horrible	exponential	2^n
horrible	exponential	3^n
horrible	factorial	$n!$



Orders of growth (log–log plot)

! constant < logarithmic < polynomial < exponential < factorial < n^n

$\log_b(n)$ n^c $(c > 0)$ c^n $(c > 1)$

COMMON TIME COMPLEXITY

O(c) = 1, c: any constant

O(log logn)

O(logn) ➔ $O(\sqrt{n}) \approx O(n^c)$, $0 < c < 1$ ➔ $O(\sqrt{n})$

O($\sqrt{n}$ logn) ➔ (logn * logn) ➔ $(\log n)^2$

O(n)

O(nlogn) ➔ O(logn!)

$O(n^c)$, $c \geq 2$

$O(c^n)$, $c > 1$

O(n!)

$O(n^n)$

$log_n n = 1$

$\log 1 = 0$

REASONS FOR THE NEED FOR ASYMPTOTIC NOTATIONS

- The algorithm may have more than one time complexity for different cases.

  Example➔ case study (1), three time complexity.

- To compare between the different algorithms.

  Example➔ case study (2), the algorithm A is better/equal/wore than algorithm B.

Description of the Asymptotic Notations

BIG-O NOTATION (WORST CASE)

- Given **f(n)** and **g(n)** - functions defined for positive integers:

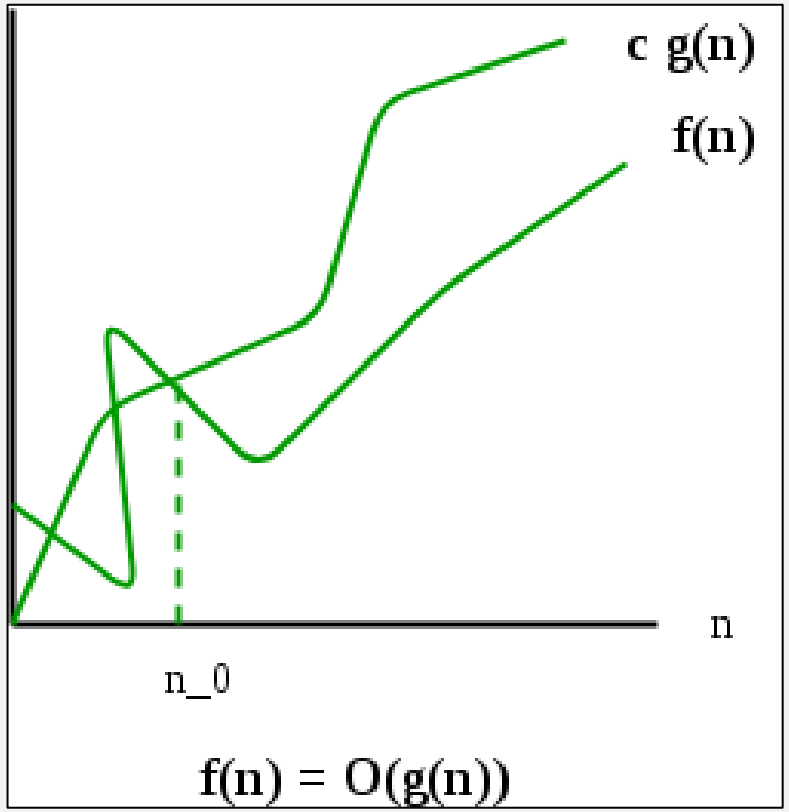
Then f(n) = O(g(n))

- If there exists a positive constant **c (c ≥1)** such that:

f(n) ≤ c*g(n)

for all sufficiently large positive integers n, **(n ≥n_0).**

- The Big O notation defines an **upper bound** of an algorithm, it bounds a function only from above.



Ω NOTATION (BEST CASE)

- Given **f(n)** and **g(n)** - functions defined for positive integers:

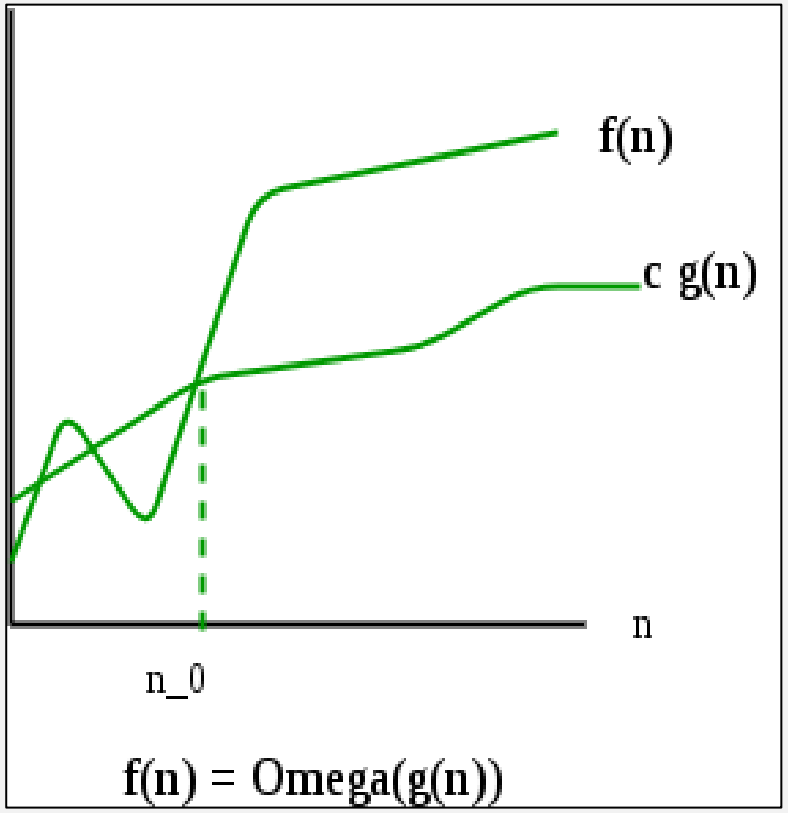
Then f(n) = Ω(g(n))

If there exists a positive constant **c (c ≥1)** such that:

f(n) ≥ c*g(n)

for all sufficiently large positive integers n, **(n ≥n_0).**

- The Omega Ω notation defines an **lower bound** of an algorithm, it bounds a function only from below.
- Since the best-case performance of an algorithm is generally not useful, the Omega notation is **the least used** notation among all three.



THETA-Θ NOTATION

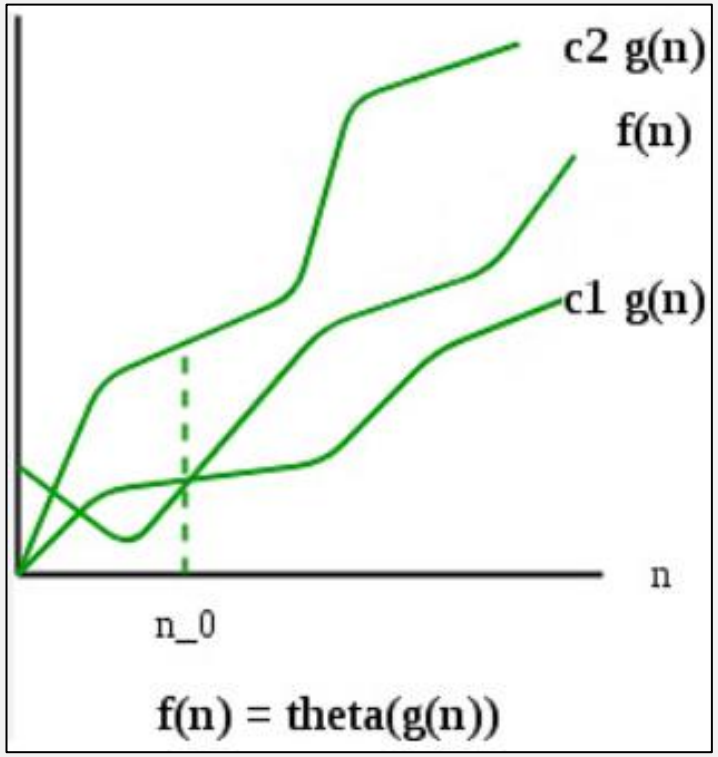
- Given **f(n)** and **g(n)** - functions defined for positive integers:

Then f(n) = Θ(g(n))

If there exists two positive constants **c1 and c2 (c1, c2 ≥1)** such that:
c1*g(n) ≤ f(n) ≤ c2 * g(n)

for all sufficiently large positive integers n, **(n ≥n_0).**

- The above definition means, if f(n) is theta of g(n), then the value f(n) is always between cl*g(n) and c2*g(n) for large values of n (n >= n_0).
- The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
- Combine between the big and omega notations.



Examples of Big Notation

EXAMPLE(1) OF BIG NOTATION

- Assume **f(n)** = 2n + 5, **g(n)** = n, prove that: f(n) = O(g(n))

Hint: c*g(n) = 3n, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \leq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \leq c*g(n)$.

$n_o = 1$, ➔ 2(1) + 5 <= 3(1) ➔ 7<= 3 (False)

$n_o = 2$, ➔ 2(2) + 5 <= 3(2) ➔ 9<= 6 (False)

$n_o = 3$, ➔ 2(3) + 5 <= 3(3) ➔ 11<= 9 (False)

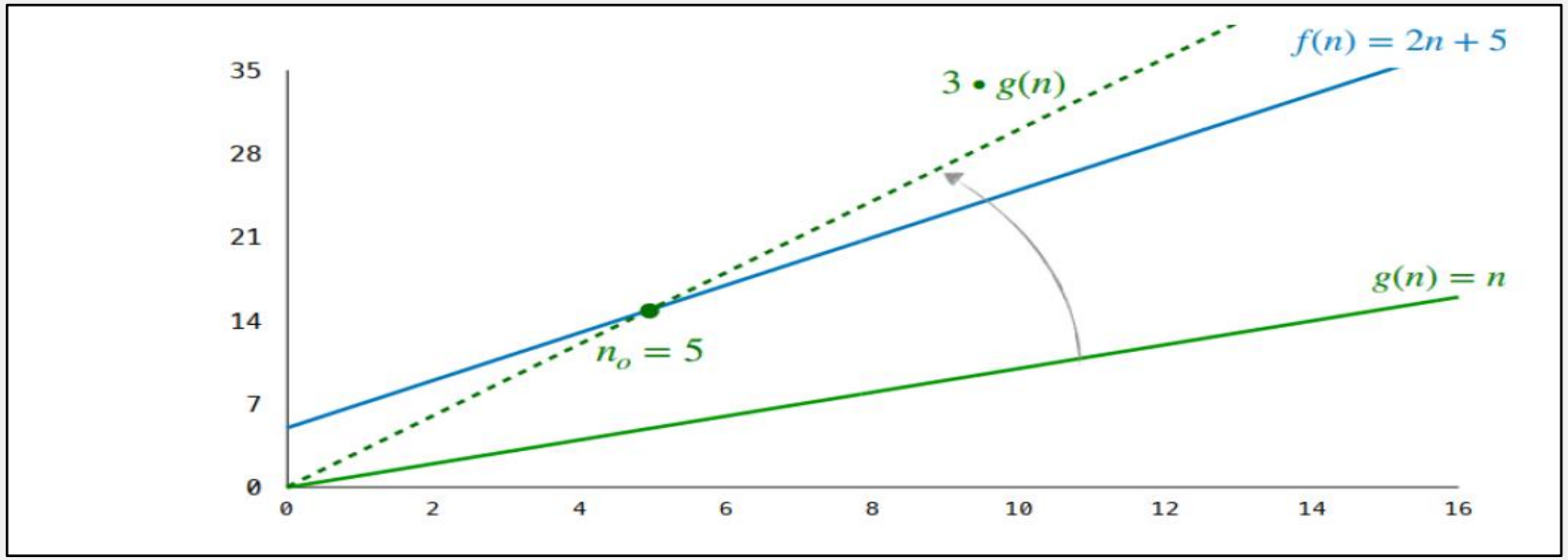
$n_o = 4$, ➔ 2(4) + 5 <= 3(4) ➔ 13<= 12 (False)

$n_o = 5$, ➔ 2(5) + 5 <= 3(5) ➔ 15<= 15 (True)

EXAMPLE(1) OF BIG NOTATION

- When the input is (5), the two algorithms are equal in the operations number, but when the inputs become greater than (5), **c*g(n) will be worse than f(n).**

➔ **We can say that c*g(n) worse that f(n) when n>5.**



EXAMPLE(2) OF BIG NOTATION

- Assume **f(n)** = 2n + 5, **g(n)** = n, prove that: f(n) = O(g(n))

Hint: c*g(n) = 4n, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \leq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \leq c*g(n)$.

$n_o = 1$, ➔ 2(1) + 5 <= 4(1) ➔ 7<= 4 (False)

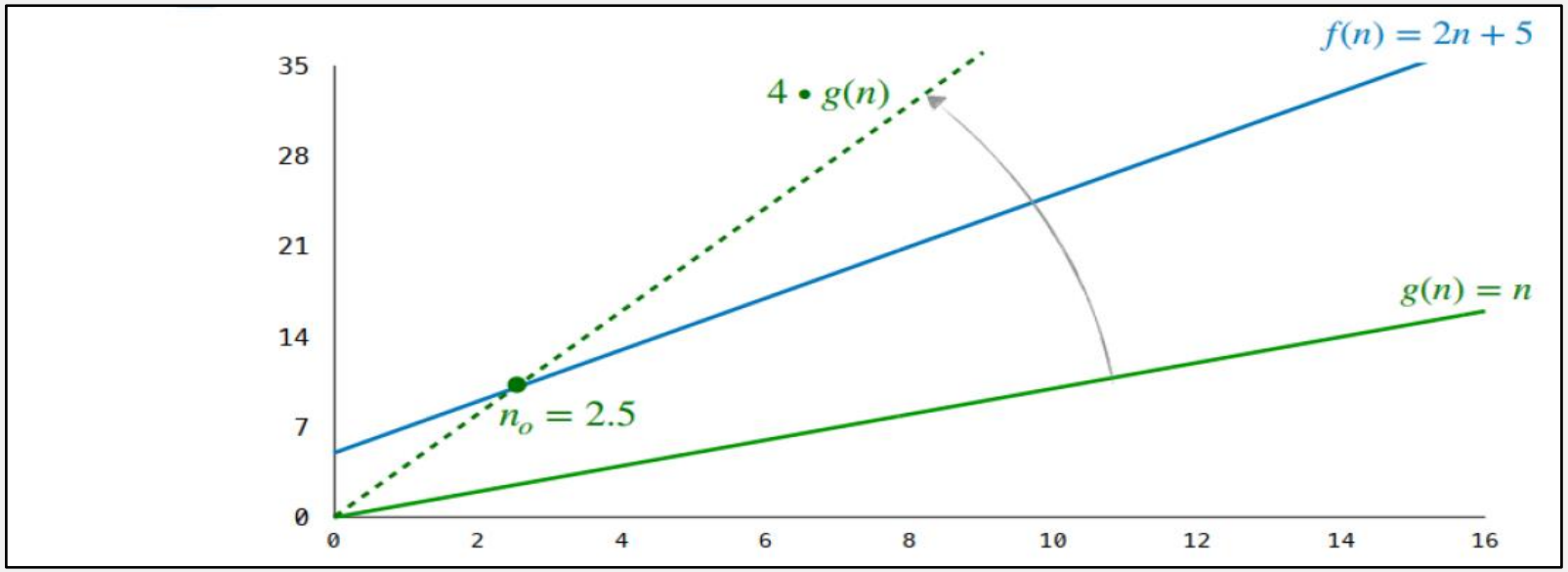
$n_o = 2$, ➔ 2(2) + 5 <= 4(2) ➔ 9<= 8 (False)

$n_o = 2.5$, ➔ 2(2.5) + 5 <= 4(2.5) ➔ 10 <= 10 (True)

EXAMPLE(2) OF BIG NOTATION

- When the input is (2.5), the two algorithms are equal in the operations number, but when the inputs become greater than (5), **c*g(n) will be worse than f(n).**

➔ **We can say that c*g(n) worse that f(n) when n>2.5**



EXAMPLE(3) OF BIG NOTATION

- Assume **f(n)** = 2n + 5, **g(n)** = n, prove that: f(n) = O(g(n))

Hint: c*g(n) = 7n, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \leq c*g(n)}$$

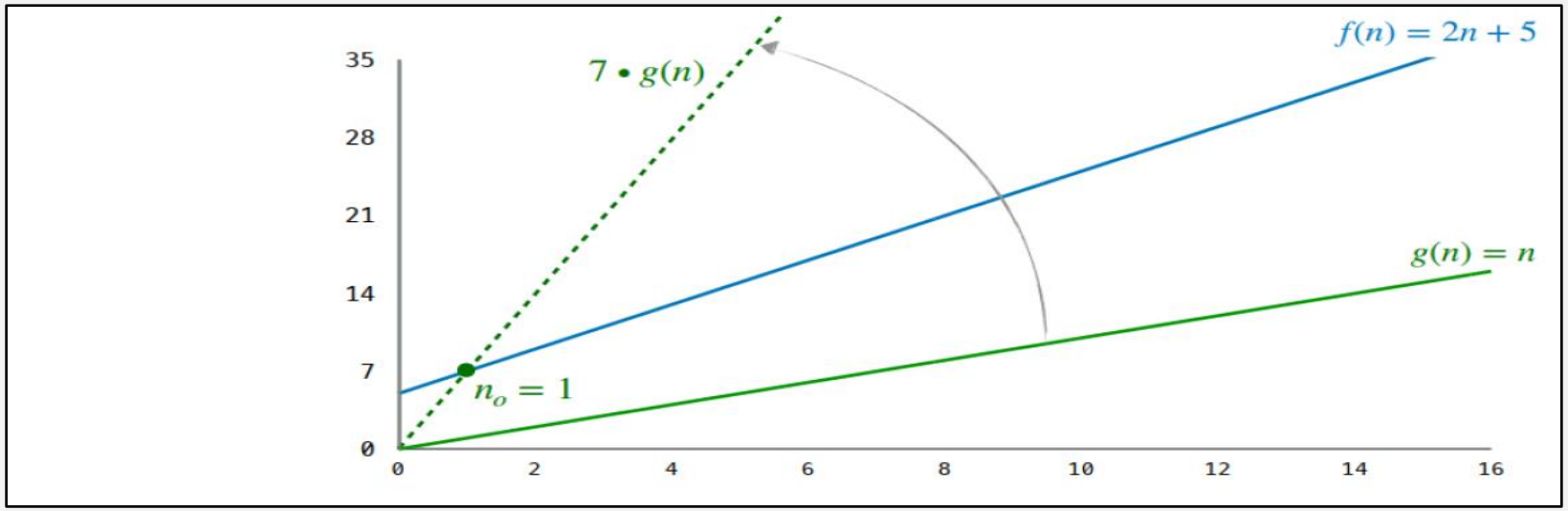
to prove that equation, we must find n_0 that make $f(n) \leq c*g(n)$.

$n_o = 1$, ➔ $2(1) + 5 <= 7(1)$ ➔ $7 <= 7$ (True)

EXAMPLE(3) OF BIG NOTATION

- When the input is (1), the two algorithms are equal in the operations number, but when the inputs become greater than (1), **c*g(n) will be worse than f(n).**

➔ **We can say that c*g(n) worse that f(n) when n>1**



EXAMPLES OF BIG NOTATION

- Prove that these functions f(n) are belong/equal to O(g(n)).

- **f(n)** = 20n, **g(n)** = 2^n , **Hint:** assume **c = 1** **Sol. n_0 = 8**
- **f(n)** = 2n^2 , **g(n)** =n!, **Hint:** assume **c = 1** **Sol. n_0 = 5**
- **f(n)** = log8n , **g(n)** = n^3 , **Hint:** assume **c = 2** **Sol. n_0 = 2**
- **f(n)** = 10n + 1, **g(n)** = n , **Hint:** assume **c = 11** **Sol. n_0 = 1**
- **f(n)** = 5n^2 + 6n, **g(n)** = 2^n , **Hint:** assume **c = 1** **Sol. n_0 = 9**

Examples of Omega Notation

EXAMPLE(1) OF OMEGA NOTATION

- Assume **f(n)** = 6n, **g(n)** = n + 5, prove that: f(n) = Ω(g(n))

Hint: c*g(n) = 2n + 5, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \geq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \geq c*g(n)$.

$n_o = 1$, ➔ 6(1) >= 2(1) + 5 ➔ 6 >= 7 (False)

$n_o = 2$, ➔ 6(2) >= 2(2) + 5 ➔ 12 >= 9 (**True**)

- When the inputs become equal/greater than (2), c*g(n) will be better than f(n).

  ➔ We can say that c*g(n) better that f(n) when n>= 2

EXAMPLE(2) OF OMEGA NOTATION

- Assume **f(n)** = n^2, **g(n)** = nlogn, prove that: f(n) = Ω(g(n))

Hint: c*g(n) = 5nlogn, don't forget that c and n_0 is positive values.

Solution:

$$\mathbf{f(n) \geq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \geq c*g(n)$.

$n_0 = 1$, ➔ 1^(2) >= 5(1)log(1) ➔ 1>= 0 (**True**)

- When the inputs become equal/greater than (1), c*g(n) will be better than f(n).

  ➔ We can say that c*g(n) better that f(n) when n>= 1

EXAMPLE(3) OF OMEGA NOTATION

- Assume **f(n)** = 2n^2 , **g(n)** = n^2 + 3n -1, prove that: f(n) = Ω(g(n))

Hint: c*g(n) = n^2 + 3n -1, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \geq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \geq c*g(n)$.

$n_o = 1$, ➔ 2(1)^2 >= (1)^2 + 3(1) -1➔ 2 >= 3 (False)

$n_o = 2$, ➔ 2(2)^2 >= (2)^2 + 3(2) -1➔ 8 >= 9 (False)

$n_o = 3$, ➔ 2(3)^2 >= (3)^2 + 3(3) -1➔ 18 >= 17 (**True**)

- When the inputs become equal/greater than (3), c*g(n) will be better than f(n).

  ➔ We can say that c*g(n) better that f(n) when n>= 3

EXAMPLE(4) OF OMEGA NOTATION

- Assume **f(n)** = n^2, **g(n)** = n, prove that: f(n) = Ω(g(n))

Hint: c*g(n) = n, don't forget that c and n_o is positive values.

Solution:

$$\mathbf{f(n) \geq c*g(n)}$$

to prove that equation, we must find n_0 that make $f(n) \geq c*g(n)$.

$n_o = 1$, ➔ (1) ^ 2 >= 1 ➔ 1 >= 1 (**True**)

- When the inputs become equal/greater than (3), c*g(n) will be better than f(n).

  ➔ We can say that c*g(n) better that f(n) when n>= 1

Examples of Theta Notation

EXAMPLE OF THETA NOTATION

- Assume **f(n)** = 4n + 8, **g(n)** = n, prove that: f(n) = Θ(g(n))

Hint: c1*g(n) = n, **c2*g(n) =** 11n

Solution:

$$\mathbf{c1*g(n) \leq f(n) \leq c2*g(n)}$$

to prove that equation, we must find n_0 that make the previous equation correct.

➔ **c1*g(n) ≤ f(n)**

$n_0 = 1$, ➔ (1) <= 4(1) + 8 ➔ 1 <= 12 (**True**)

- When the inputs become equal/greater than (1), c1*g(n) will be better than f(n).

  ➔ We can say that c1*g(n) better that f(n) when n>= 1, f(n) = Ω(c1*g(n))

EXAMPLE OF THETA NOTATION

- Assume **f(n)** = 4n + 8, **g(n)** = n, prove that: f(n) = Θ(g(n))

Hint: c1*g(n) = n, **c2*g(n) =** 11n

Solution:

$$\mathbf{c1*g(n) \leq f(n) \leq c2*g(n)}$$

to prove that equation, we must find n_0 that make the previous equation correct.

➔ **f(n) ≤ c2*g(n)**

n_o = 1, ➔ 4(1) + 8 <= 11(1) ➔ 12 <= 11 (False)

no = 2, ➔ 4(2) + 8 <= 11(2) ➔ 16 <= 22 (**True**)

- When the inputs become equal/greater than (2), c2*g(n) will be worse than f(n).

  ➔ We can say that c2*g(n) worse that f(n) when n>= 2, f(n) = O(c1*g(n))

USES OF ASYMPTOTIC NOTATION

1. To represent the best, average, and worst case of algorithms. (will see that later).

2. To compare between the different algorithms. (Here, the notation **will be given**).

- We will prove that **algorithm A** is better, equal, or worse than **algorithm B**, by using the **mathematical equations and finding n_0.**

Note: in some cases **we determine n_0** and **find c** which makes the equation correct.

3.To compare between the different algorithms. (Here, the notation **won't be given**).

- Here, you **must determine** which one of the algorithms is better, equal, or worse than the other then represent it by notations.

EXAMPLE(1) OF BIG NOTATION (FIND C)

- Assume **f(n)** = 2n + 5, **g(n)** = n, prove that: f(n) = O(g(n))

Hint: assume $n_0 = 1$, don't forget that c and no is positive values.

Solution:

$$\mathbf{f(n) \leq c*g(n)}$$

to prove that equation, we must find c that make $f(n) \leq c*g(n)$.

- no = 1, ➔ 2(1) + 5 <= c(1) ➔ 7<= c ➔ c =7
- **g(n) = 7n**

EXAMPLE(2) OF OMEGA NOTATION (FIND C)

- Assume **f(n)** = 5n , **g(n)** = n + 4, prove that: f(n) = Ω(g(n))

Hint: assume $n_0 = 2$, don't forget that c and no is positive values.

Solution:

$$\mathbf{f(n) \geq c*g(n)}$$

to prove that equation, we must find c that make $f(n) \geq c*g(n)$.

- no = 2, ➔ 5(2) >= c(2) + 4 ➔ 10>= 2c + 4 ➔ c = 3
- **g(n) = 3n + 4**

COMPARISON OF FUNCTIONS

Which function is bigger n^2 n^3 ??

- *one option*: $n = 2$ 2^2 2^3

  n=3

$$n^2 < n^3$$

- One option apply **log** to the both sides:

  $\log n^2$ $\log n^3$

  2 log n < 3 log n

LOGARITHMS

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$$\log^k n = (\log n)^k$$

$$\log\log n = \log(\log n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$\log_a x = \log_a b \log_b x$$

$$a^{\log_b x} = x^{\log_b a}$$

COMPARISON OF FUNCTIONS

Which function is bigger $n^2 log\ n > n(logn)^{10}$??

- One option apply **log** to the both sides:

$\log(n^2 * \log n)$ $\log(n * (\log n)^{10})$

$\log n^2 + \log\log n$ $\log n + \log(\log n)^{10}$

2logn + loglogn $\geq$ **logn** + 10(loglogn)

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

COMPARISON OF FUNCTIONS

Which function is bigger $3n^{\sqrt{n}} > 2^{\sqrt{n}\log n}$??

- One option apply change **log** to the both sides:

$\log 3n^{\sqrt{n}}$ $\quad$ $\log 2^{\sqrt{n}\log n}$

$\sqrt{n}\log 3n$ $\quad$ $\sqrt{n}\log n * \log_2 2$

$\sqrt{n}\log 3n \geq \sqrt{n}\log$

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

$3n^{\sqrt{n}}$ $\quad$ $2^{\log n^{\sqrt{n}}}$

$3n^{\sqrt{n}}$ $\quad$ $(n^{\sqrt{n}})^{\log 2}$

$3n^{\sqrt{n}} \geq n^{\sqrt{n}}$

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

COMPARISON OF FUNCTIONS

Which function is bigger $n^{logn} < 2^{\sqrt{n}}$??

- One option apply **log** to the both sides:

$\log n^{\log n}$ $\geq$ $\log 2^{\sqrt{n}}$

$\log n * \log n$ $\geq$ $\sqrt{n} * \log 2$

$\log^2 n$ $\geq$ $\sqrt{n}$

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

COMPARISON OF FUNCTIONS

Which function is bigger $2^n < 2^{2n}$??

- One option apply change **log** to the both sides:

$$\log 2^n \quad \log 2^{2n}$$

$$n\log 2 \quad 2n\log 2$$

$$n \leq 2n$$

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

COMPARISON OF FUNCTIONS

For each of the following pairs of functions, either f(n) is $O(g(n))$, f(n) is $\Omega(g(n))$, or f(n) is $\Theta(g(n))$. Determine which relationship is correct.

- $f(n) = n$; $g(n) = \log n^2$ — $f(n) = \Omega(g(n))$
- $f(n) = \log \log n$; $g(n) = \log n$ — $f(n) = O(g(n))$
- $f(n) = n \log n + n$; $g(n) = \log n$ — $f(n) = \Omega(g(n))$
- $f(n) = 10$; $g(n) = \log 10$ — $f(n) = \Theta(g(n))$
- $f(n) = 2^n$; $g(n) = 10n^2$ — $f(n) = \Omega(g(n))$
- $f(n) = 2^n$; $g(n) = 3^n$ — $f(n) = O(g(n))$

SMALL-O AND SMALL-W

Informal Definition. f is said to be $o(g)$ if it grows strictly slower than g.

Informal Definition. f is said to be $\omega(g)$ if it grows strictly faster than g.

Notation	Order of Growth Relation	Example
$f = O(g)$	$f \leq g$	If $f = O(n^2)$, examples for f could be: n^2, $3n^2 + n$, $5n - 1$, $7n \log n + 5n$, $\sqrt{n}$
$f = o(g)$	$f < g$	If $f = o(n^2)$, examples for f could be: $n^{1.9}$, $5n - 1$, $7n \log n + 5n$, $\sqrt{n}$
$f = \Omega(g)$	$f \geq g$	If $f = \Omega(n^2)$, examples for f could be: n^2, $3n^2 + n$, $5n^3$, $7n^5$, 2^n
$f = \omega(g)$	$f > g$	If $f = \omega(n^2)$, examples for f could be: $n^{2.01}$, $n^2 \log n$, $5n^3$, $7n^5$, 2^n
$f = \Theta(g)$	$f = g$	If $f = \Theta(n^2)$, examples for f could be: n^2, $3n^2$, $5n^2 - n$, $7n^2 + n \log n + 100$

END OF CHAPTER2/PART2