Performance Analysis

Performance Analysis

- There are problems and algorithms to solve them.
- Problems and problem instances.
- Example: Sorting data in ascending order.
  - Problem: Sorting
  - Problem Instance: e.g. sorting data (2 3 9 5 6 8)
  - Algorithms: Bubble sort, Merge sort, Quick sort, Selection sort, etc.
- Which is the best algorithm for the problem? How do we judge?

Performance Analysis

- Two criteria are used to judge algorithms: (i) time complexity (ii) space complexity.
- <u>Space Complexity</u> of an algorithm is the amount of memory it needs to run to completion.
- <u>Time Complexity</u> of an algorithm is the amount of CPU time it needs to run to completion.

Space Complexity

- Why?
  - To know in advance about sufficient memory
  - To select the program with minimum memory requirement
  - To estimate the largest instance it can solve

Components of Space Complexity

1. Instruction space
   - Space needed to store the compiled version of program instruction
   - Depends on…
     - The compiler used to compile the program
       - Ex. a+b + (b*c) + (a+b-c) / (a+b)/4
     - The compiler options in effect
       - Ex. Overlay option
     - The target computer
       - Floating point hardware

Components of Space Complexity

2. Data space
   – Space needed by constants and simple variables
   – Space needed by dynamically allocated objects (arrays, class instance etc.)
3. Environmental stack space
   – The return address
   – All local variables
   – All formal parameters
   – Recursion stack space
     • The space needed by local variables & formal parameters
     • The maximum depth of recursion

Space Complexity

- Memory space S(P) needed by a program P, consists of two components:
  - A fixed part: needed for instruction space (byte code), simple variable space, constants space etc. → c
  - A variable part: dependent on a particular instance of input and output data. → S_p(instance)
- $S(P) = c + S_p(\text{instance})$

Space Complexity: Example 1

```
Algorithm abc (a, b, c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

For every instance 3 computer words required to store variables: a, b, and c. Therefore $S_p()= 3$. $S(P) = 3$.

Space Complexity: Example 2

```
Algorithm Sum(a[], n)
{
      s:= 0.0;
      for i = 1 to n do
            s := s + a[i];
      return s;
}
```

Space Complexity: Example 2.

- Every instance needs to store array a[] & n.
  - Space needed to store n = 1 word.
  - Space needed to store a[ ] = n floating point words (or at least n words)
  - Space needed to store i and s = 2 words
- $S_p(n) = (n + 3)$. Hence $S(P) = (n + 3)$.

Space Complexity: Example 3

```
Algorithm RSum(a[], n)
{
    if (n > 0)
    return Rsum(a,n-1) + a[n-1];
        return 0;
}
```

Space Complexity: Example 3

- Every recursive function call needs to store address of array a[], the value of n & return address.
  - Space needed to store n = 1 word.
  - Space needed to store the address of a[ ] = 1 word
  - Space needed to store return address = 1 words
  - Depth of recursion = n+1
  - Space needed to store a[] = n words
- $S_p(n) = n$. Hence $S(P) = 3(n + 1) + n$.

Space Complexity: Example 4

```
Algorithm Fact(int n)
{
    if (n <= 1)
    return 1;
    else return (n * Fact(n-1));
}
```

Space Complexity: Example 4

- Every recursive function call needs to store the value of n & return address.
  - Space needed to store n = 1 word.
  - Space needed to store return address = 1 words
  - Depth of recursion = n
- Hence S(P) = 2(n)

Time Complexity

- Time required T(P) to run a program P also consists of two components:
  - A fixed part: compile time which is independent of the problem instance → c.
  - A variable part: run time which depends on the problem instance → t_p(instance)
- $T(P) = c + t_p(\text{instance})$

Time Complexity

- How to measure T(P)?
  1. Measure experimentally, using a “stop watch”
     - T(P) obtained in secs, msecs.
  2. Count no. of major operations / instructions.
     - Identify one or more major operations and determine how many times each is executed.
     - Ignore the other minor operations / instructions.
  3. Count program steps → T(P) obtained as a <u>step count.</u>
- Fixed part is usually ignored; only the variable part $t_p()$ is measured.

Operation count : Example 1

```
Algorithm IndexOfMax(int a[], int n)
{
    int index = 0;
    for(int i=1; i<n; i++)
    if (a[index] < a[i])
        index = i;
    return index;
}
```

- Comparison can be considered as major operation in above algorithm.
- Total number of comparison = max{n-1, 0 }
- Hence, T(P) = n-1

Operation count : Example 2

```
Algorithm PolyEval(int coeff[], int n, int x)
{
    int y=1, value=coeff[0] ;
    for(int i=1; i<n; i++)
    {
        y = y * x;
        value = value + y * coeff[i];
    }
}
```

Operation count : Example 2

- Here, additions and multiplications can be identified as major operations.
  - Total number of additions = n
  - Total number of multiplications = 2n
- Hence, total number of operations = 3n
- T(P) = 3n

Step Count

- Considers all executable statements of an algorithm.
- What is a program step?
  - a+b+b*c+(a+b)/(a-b) → one step;
  - comments → zero steps;
  - `while (<expr>) do` → step count equal to the number of times <expr> is executed.
  - `for i=<expr> to <expr1> do` → step count equal to number of times <expr1> is checked.

Step Count : Example 1

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[],n)	0	–	0
2	{	0	–	0
3	S = 0.0;	1		
4	for i=1 to n do	1		
5	s = s+a[i];	1		
6	return s;	1		
7	}	0	–	0

Step Count : Example 1

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[],n)	0	–	0
2	{	0	–	0
3	S = 0.0;	1	1	1
4	for i=1 to n do	1	n+1	n+1
5	s = s+a[i];	1	n	n
6	return s;	1	1	1
7	}	0	–	0
				2n+3

Step Count : Example 2

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[],n,m)	0	-	0
2	{	0	-	0
3	for i=1 to n do;	1		
4	for j=1 to m do	1		
5	s = s+a[i][j];	1		
6	return s;	1		
7	}	0	-	0

Step Count : Example 2

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[],n,m)	0	-	0
2	{	0	-	0
3	for i=1 to n do;	1	n+1	n+1
4	for j=1 to m do	1	n(m+1)	n(m+1)
5	s = s+a[i][j];	1	nm	nm
6	return s;	1	1	1
7	}	0	-	0

2nm+2n+2

Step Count : Example 3

	Statements	S/E	Freq.	Total
1	Algo Transpose(a[],row)	0	–	0
2	{	0	–	0
3	for (i=0; i<row ; i++)	1		
4	for (j=i+1; j<row; j++)	1		
5	swap(a[i][j],a[j][i]);	1		
7	}	0	–	0

Step Count : Example 3

	Statements	S/E	Freq.	Total
1	`Algo Transpose(a[],row)`	0	-	0
2	`{`	0	-	0
3	`for (i=0; i<row ; i++)`	1	row+1	row+1
4	`for (j=i+1; j<row; j++)`	1	row(row+1)/2	row(row+1)/2
5	`swap(a[i][j],a[j][i]);`	1	row(row-1)/2	row(row-1)/2
7	`}`	0	-	0

$\text{row}^2 + \text{row} + 1$

Step Count : Example 4

	Statements	S/E	Freq.	Total
1	Algorithm ABC(a[],b[],n)	0	–	0
2	{	0	–	0
3	for (j=0; j<n ; j++)	1		
4	b[j] = sum(a, j+1);	2j+6		
5	}	0	–	0

Step Count : Example 4

	Statements	S/E	Freq.	Total
1	Algorithm ABC(a[],b[],n)	0	–	0
2	{	0	–	0
3	for (j=0; j<n ; j++)	1	n+1	n+1
4	b[j] = sum(a, j+1);	2j+6	n	n(n+5)
5	}	0	–	0
				$n^2 + 6n + 1$

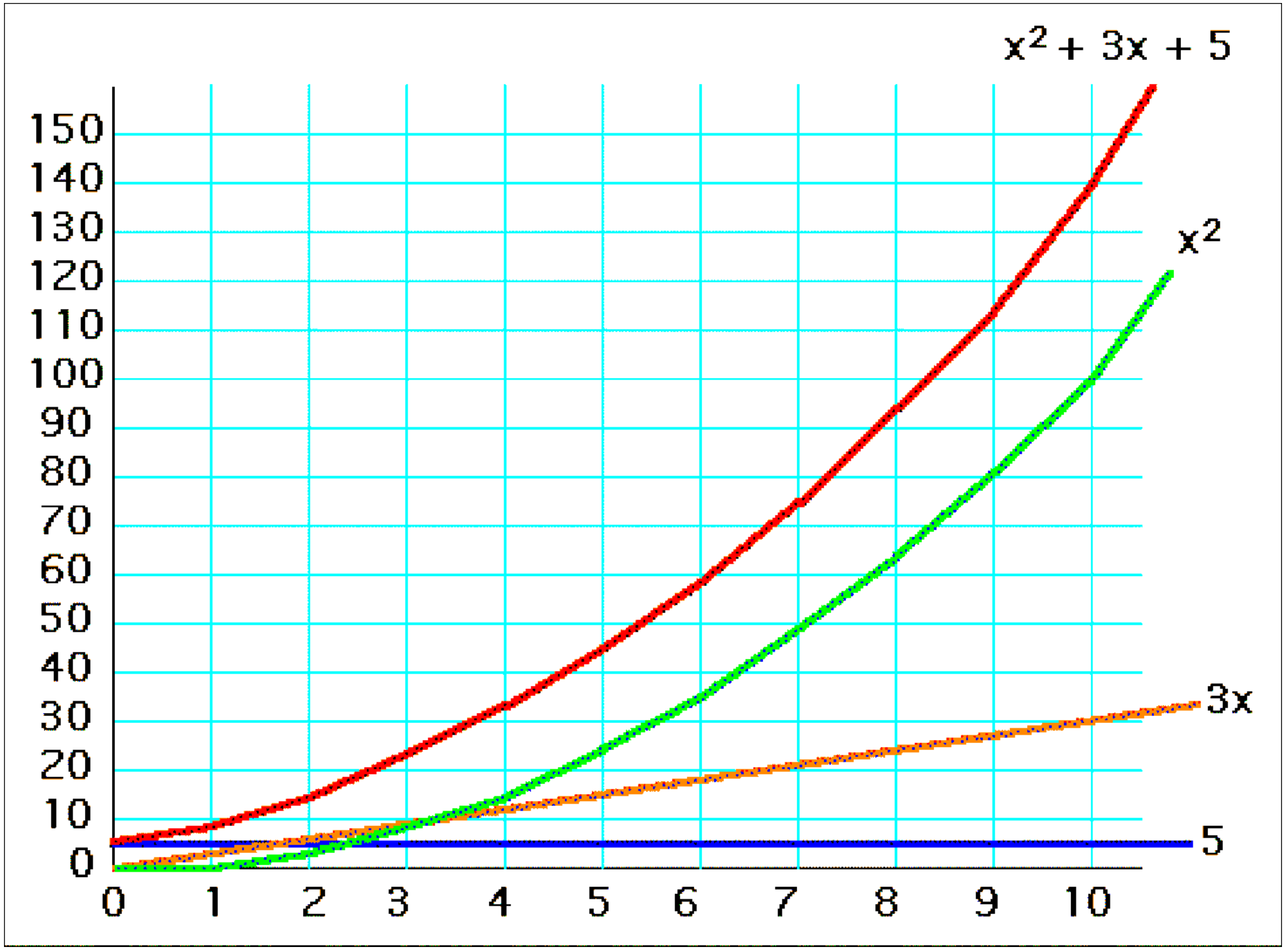
Here, $$\sum_{j=0}^{n-1}(2j+6) = 2\sum_{j=0}^{n-1} j + \sum_{j=0}^{n-1} 6$$

Problem with step count

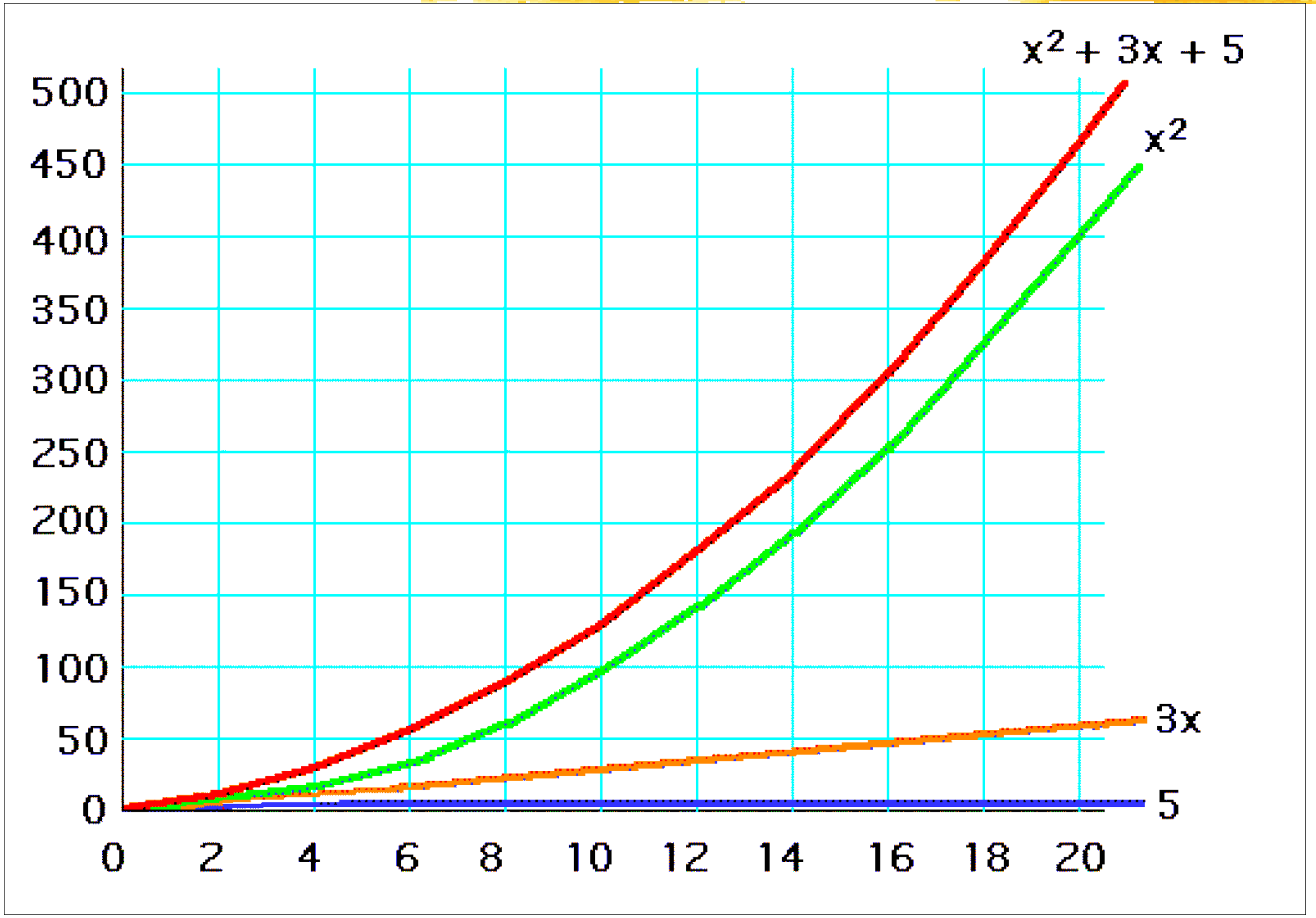
- Doesn't give accurate estimate of time complexity, because step is not well defined
- For ex,
  - $T(P1) = 4n^2 + 6n + 2$ OR
  - $T(P1) = 5n^2 + 7n + 3$ is correct for a given program P1?
- Consider $R = x^2 + 3x + 5$ as x varies:

$x = 0$	$x^2 = 0$	$3x = 0$	$5 = 5$	$R = 5$
$x = 10$	$x^2 = 100$	$3x = 30$	$5 = 5$	$R = 135$
$x = 100$	$x^2 = 10000$	$3x = 300$	$5 = 5$	$R = 10{,}305$
$x = 1000$	$x^2 = 1000000$	$3x = 3000$	$5 = 5$	$R = 1{,}003{,}005$
$x = 10{,}000$	$x^2 = 10^8$	$3x = 3*10^4$	$5 = 5$	$R = 100{,}030{,}005$
$x = 100{,}000$	$x^2 = 10^{10}$	$3x = 3*10^5$	$5 = 5$	$R = 10{,}000{,}300{,}005$

$R = x^2 + 3x + 5$ for $x= 1..10$



$R = x^2 + 3x + 5$ for $x = 1..20$



Observation with step count

- In general, when $F(n) = c_1n^2 + c_2n + c_3$, c_1n^2 is much larger than $c_2n + c_3$.

- Let $r(n) = \dfrac{c_2n + c_3}{c_1n^2} = \dfrac{c_2}{c_1 n} + \dfrac{c_3}{c_1n^2}$

- Now, $\lim_{n \to \infty} \dfrac{c_2}{c_1 n} + \dfrac{c_3}{c_1n^2} = 0$
- Which denotes that c_1n^2 is the dominant term in the F(n).

Observation with step count

- For two programs A & B,

  Analysis of John:

  $t_A(n) = n^2 + 3n \quad t_B(n) = 43n$

  Analysis of Mary:

  $t_A(n) = 2n^2 + 3n \quad t_B(n) = 83n$

For John:

when $n<40$, program A is faster

when $n>40$, program B is faster

For Mary:

when $n<80$, program A is faster

when $n>80$, program B is faster

Observation with step count

- In both the cases, Program A is faster than program B.
- Just the break even point (n=40 or n=80) changes.

- Hence, the value of coefficient is irrelevant.

- Thus, to reflect how one function grows with the growth of another function, study of asymptotic notation is required.

Growth of Functions

- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

- The growth of functions is usually described using the **Asymptotic notations**

- Three most important asymptotic notations are as follows:
  - Big – O notation
  - Omega notation
  - Theta notation

Big – O notation

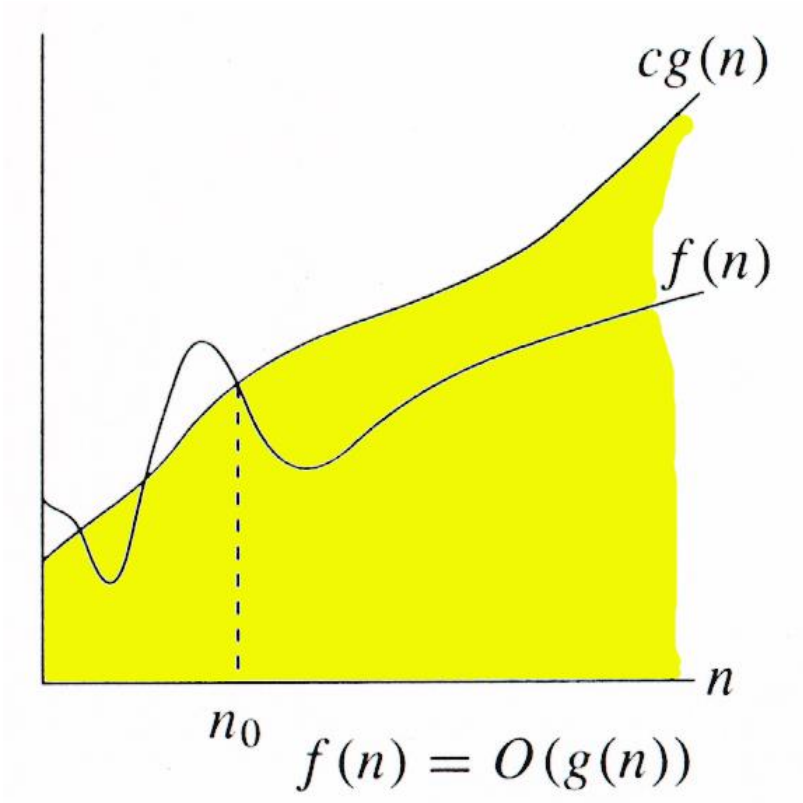
- **Definition:** Let f and g be functions from the integers or the real numbers to the real numbers. We say that f(n) is O(g(n)), if there are exist constants C and n_0 such that,

$$f(n) \leq C\ g(n)$$

- whenever $n >= n_0$.

Big – O notation

- **Graphically, it can be represented as follows:**



Big – O notation

- The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(n) for large n.

- We accept the constant C in the requirement

$$f(n) \leq Cg(n) \quad \text{whenever } n \geq n_0,$$

because **C does not grow with x**

- We are only interested in large n, so it is OK if $f(n) > Cg(n)$ for $n \geq n_0$.

Big – O Examples

- Prove that f(n) = 3n + 5 is O(n).

$$
\begin{aligned}
f(n) &= 3n + 5 \\
&\leq 3n + n \text{ , where } n \geq 5 \\
&\leq 4n \\
&\leq C\, g(n) \text{ , where } C=4 \ \&\ n_0=5
\end{aligned}
$$

- Hence, f(n) = 3n + 5 is O(n) is proved.
- Similarly, Prove for the followings:
  - $f(n) = 27n^2 + 16n$ is $O(n^2)$.
  - $f(n) = 2n^3 + n^2 + 2n$ is $O(n^3)$.
  - $f(n) = 4n^3 + 2n + 3$ is $O(n^3)$.
  - $f(n) = 2^n + 6n^2 + 3n$ is $O(2^n)$.

Big – O Examples – Incorrect bound

- Prove that $f(n) = 7n + 5 \neq O(1)$.

- Proof by contradiction:
- Assume that $7n + 5 = O(1)$. So, we must have

  $7n + 5 \leq C.1$ for $n \geq n0$.

  above statement is not true for large values of n
- Hence, our assumption is false.
- Hence, $f(n) = 3n + 5$ is $O(n)$ is proved.

- Similarly, Prove for the followings:
  – $f(n) = 10n^2 + 7 \neq O(n)$.
  – $f(n) = 27n^2 + 16n + 25 \neq O(n)$.
  – $f(n) = 3n^3 + 4n \neq O(n^2)$.

Big – O notation – Loose bounds

- Question: If f(x) is $O(x^2)$, is it also $O(x^3)$?

- **Yes.** x^3 grows faster than x^2, so x^3 grows also faster than f(x).

- Therefore, we always have to find the **smallest** simple function g(x) for which f(x) is O(g(x)).

- Ex: $f(n) = 2n + 3 = O(n^2)$

Big – Ω notation

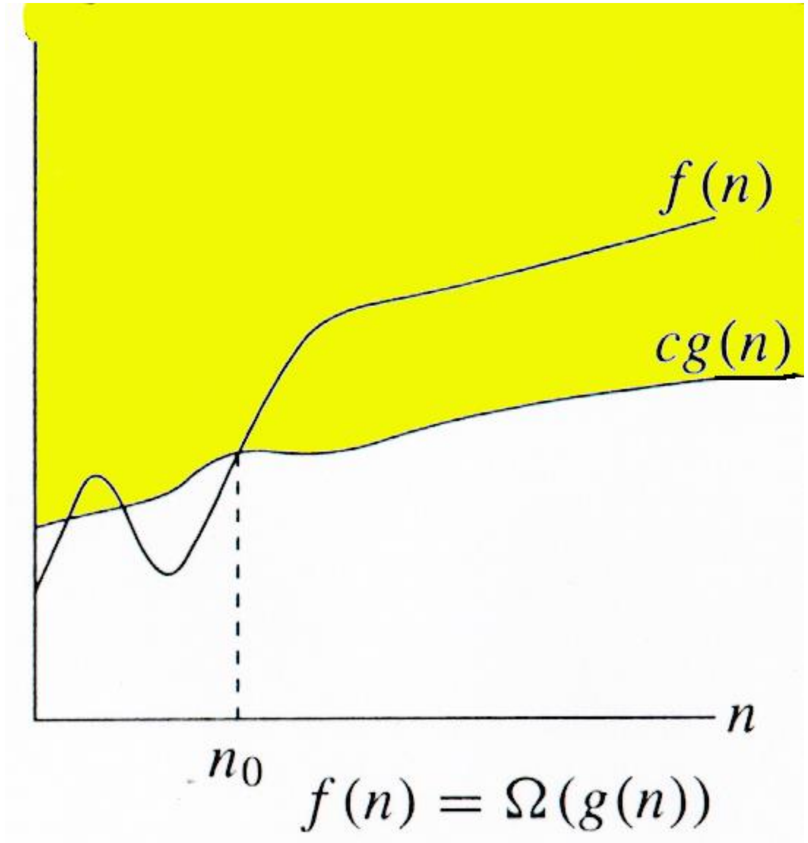
- **Definition:** Let f and g be functions from the integers or the real numbers to the real numbers. We say that f(n) is Ω(g(n)), if there are exist constants C and n_0 such that,

$$f(n) \geq C\, g(n)$$

- whenever n >= n_0 .

Big – Ω notation

- **Graphically, it can be represented as follows:**



Big – Ω notation

- The idea behind the big- Ω notation is to establish an **lower boundary** for the growth of a function f(n) for large n.
- It indicates the best case.

Big – Ω Examples

- Prove that $f(n) = 3n + 5$ is $\Omega(n)$.

$$
\begin{aligned}
f(n) &= 3n + 5 \\
&\geq 3n \text{ , where } n \geq 1 \\
&\geq C\, g(n) \text{ , where } C=3 \;\&\; n_0=1
\end{aligned}
$$

- Hence, $f(n) = 3n + 5$ is $\Omega(n)$ is proved.
- Similarly, Prove for the followings:
  - $f(n) = 27n^2 + 16n$ is $\Omega(n^2)$.
  - $f(n) = 2n^3 + n^2 + 2n$ is $\Omega(n^3)$.
  - $f(n) = 4n^3 + 2n + 3$ is $\Omega(n^3)$.
  - $f(n) = 2^n + 6n^2 + 3n$ is $\Omega(2^n)$.

Big – Ω Examples – Incorrect bound

- Prove that $f(n) = 7n + 5 \neq \Omega(n^2)$.

- Proof by contradiction:
- Assume that $7n + 5 = \Omega(n^2)$. So, we must have

  $7n + 5 \geq C n^2$ for $n \geq n0$.

  $c n^2 / (7n + 5) \leq 1$.

  above statement is not true for n.
- Hence, our assumption is false.
- Hence, $f(n) = 3n + 5$ is $\Omega(n)$ is proved.

- Similarly, Prove for the followings:
  - $f(n) = 10n^2 + 7 \neq \Omega(n^3)$.
  - $f(n) = 27n^2 + 16n + 25 \neq \Omega(n^3)$.
  - $f(n) = 3n^3 + 4n \neq \Omega(2^n)$.

Big – Ω notation – Loose bounds

- Question: If $f(x)$ is $\Omega(x^3)$, is it also $\Omega(x^2)$?
- **Yes.**
- Therefore, we always have to find the **Largest** simple function $g(x)$ for which $f(x)$ is $\Omega(g(x))$.
- Ex: $f(n) = 2n + 3 = \Omega(n^2)$

Big – Θ notation

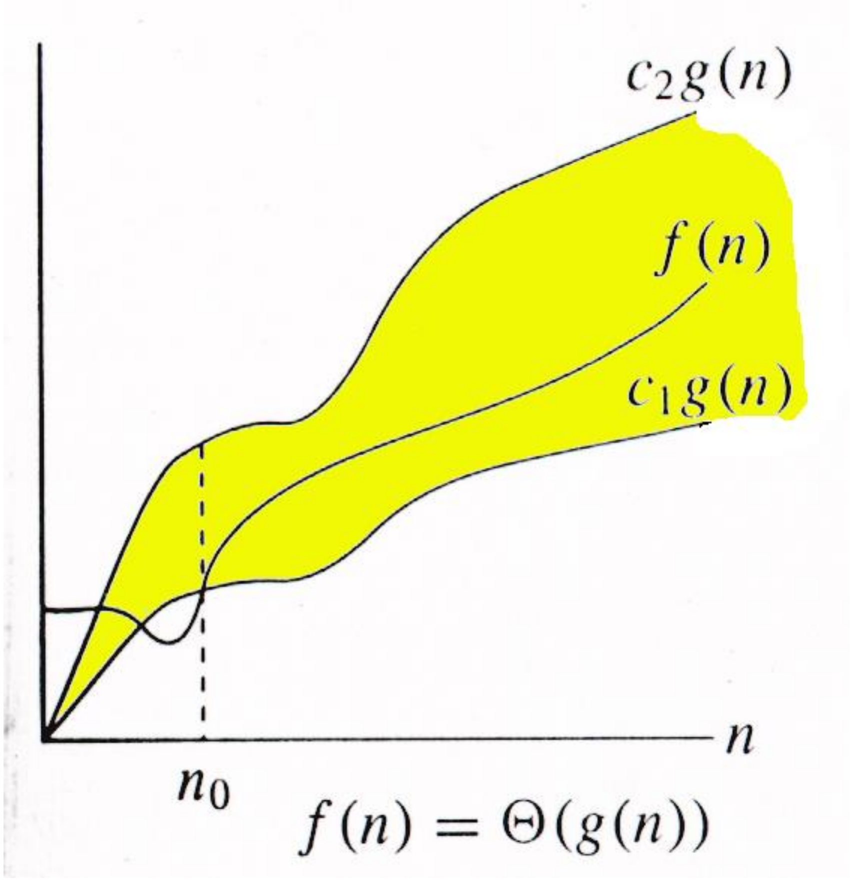
- **Definition:** Let f and g be functions from the integers or the real numbers to the real numbers. We say that f(n) is Θ(g(n)), if there are exist constants C_1, C_2 and n_0 such that,

$$C_1\, g(n) \leq f(n) \leq C_2\, g(n)$$

- whenever $n >= n_0$.

Big – Θ notation

- **Graphically, it can be represented as follows:**

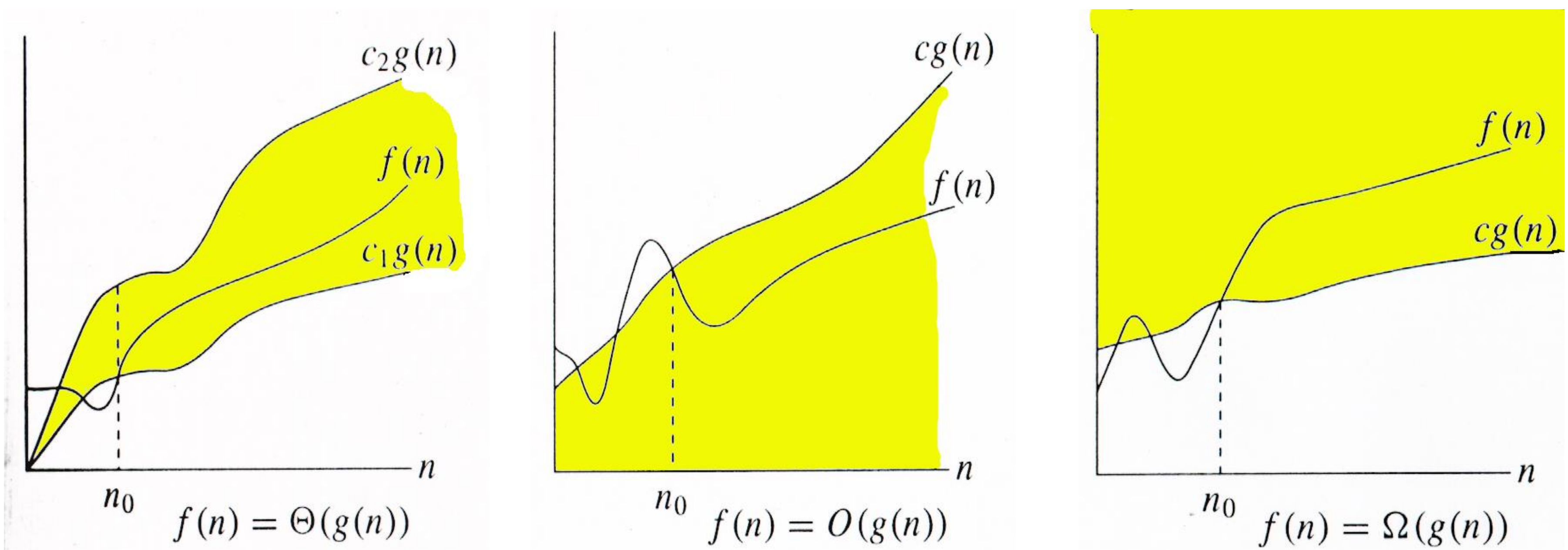


Relation between O, Ω and Θ

For any two functions $g(n)$ and $f(n)$, $\boldsymbol{f(n) = \Theta(g(n))}$ iff

$\boldsymbol{f(n) = O(g(n))}$ **and** $\boldsymbol{f(n) = \Omega(g(n))}$.

I.e., $\boldsymbol{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$



Big – Θ notation - Examples

- Prove that $f(n) = 3n + 5$ is $\Theta(n)$.

- We can say that $3n \leq 3n + 5$
  $C_1 n \leq f(n)$, where $C_1 = 3$

- similarly, $f(n) = 3n + 5$
  $\leq 3n + n$, where $n \geq 5$
  $\leq 4n$
  $\leq C_2\, g(n)$, where $C_2=4$

- Hence, $f(n) = 3n + 5$ is $\Theta(n)$ is proved.
- Similarly, Prove for the followings:
  - $f(n) = 27n^2 + 16n$ is $\Theta(n^2)$.
  - $f(n) = 2n^3 + n^2 + 2n$ is $\Theta(n^3)$.
  - $f(n) = 4n^3 + 2n + 3$ is $\Theta(n^3)$.
  - $f(n) = 2^n + 6n^2 + 3n$ is $\Theta(2^n)$.