Asymptotic notation

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

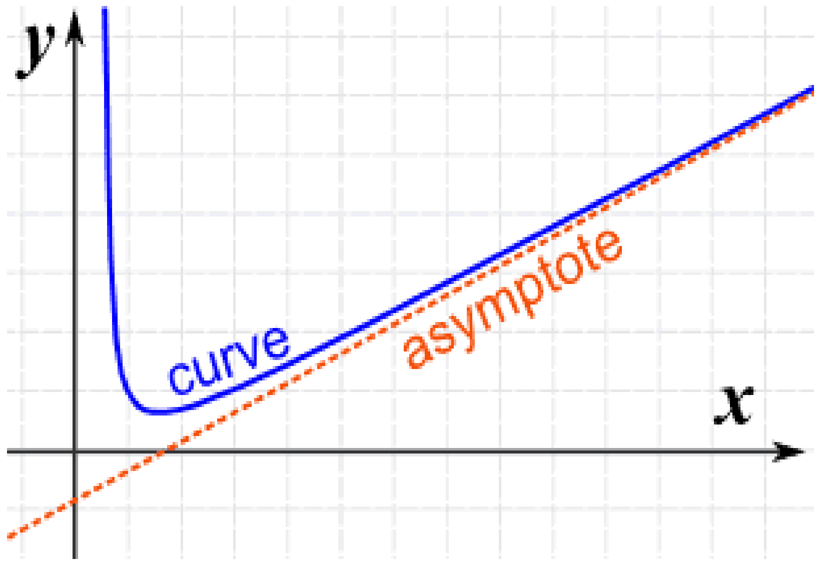
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

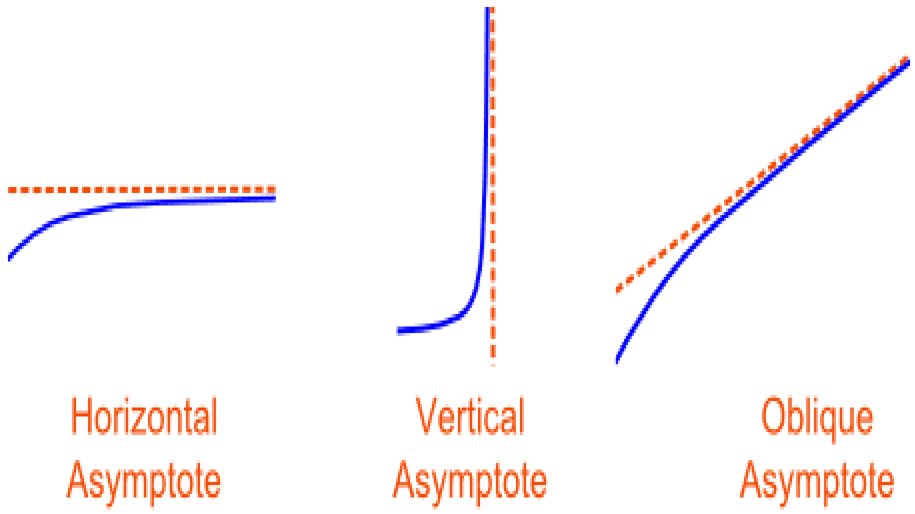
Asymptotic notation

- **Asymptotic Notations** are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's **growth rate**.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for **asymptotic analysis**.
- **Asymptotic analysis** is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- **Asymptotic analysis** refers to computing the running time of any operation in mathematical units of computation

Asymptotic Growth of Functions

- Greek: *asumptotos* - not touching (*a* - not, *sumptotos* - intersecting)
- Math: curves that do not intersect, but come infinitely close

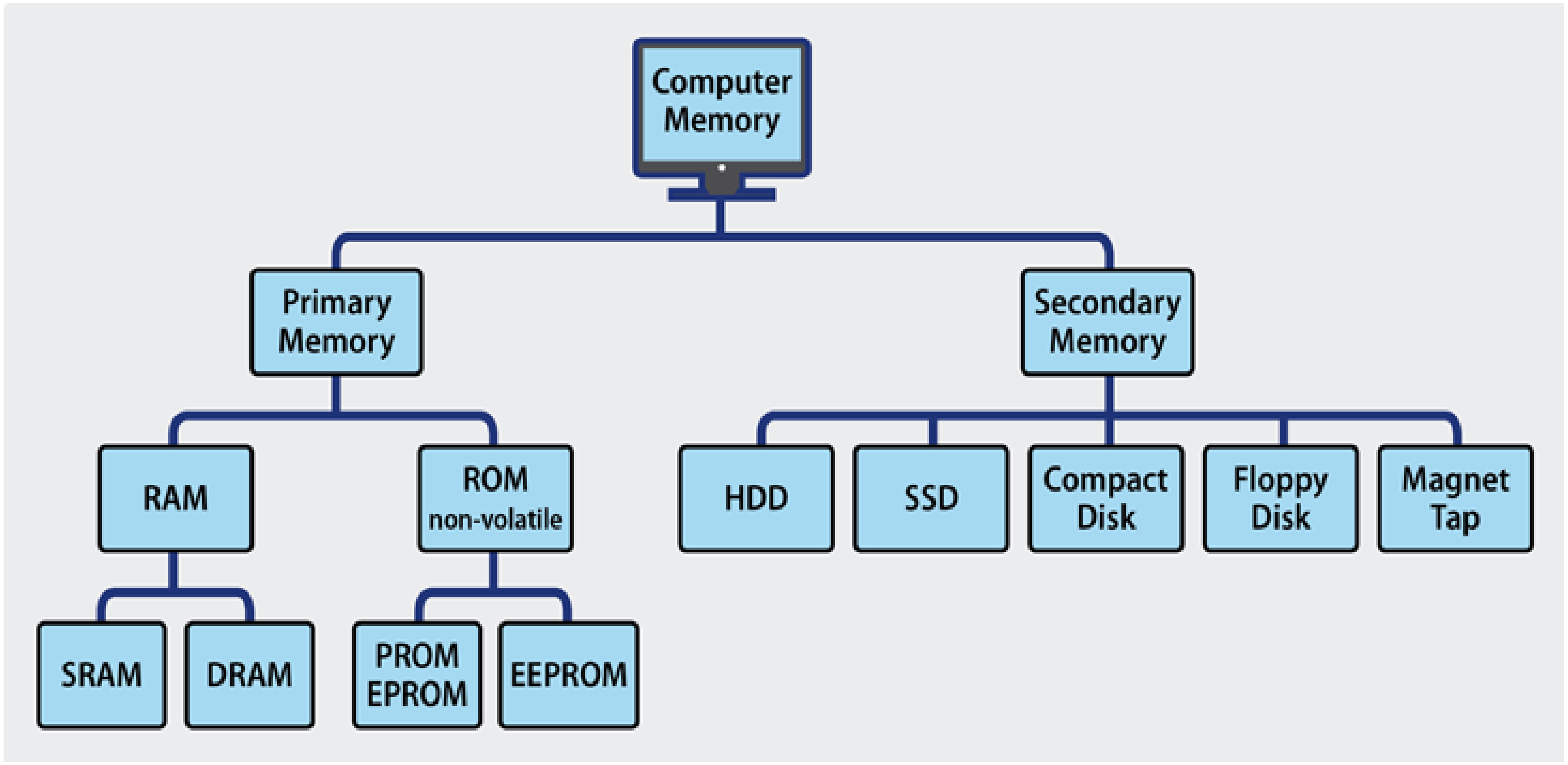




Why Asymptotic notation ?

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on **machine specific constants**, and doesn’t require algorithms **to be implemented** and **time taken by programs** to be compared.
- **Asymptotic Analysis** is the evaluation of the performance of an **algorithm** in terms of just the input size (N), where N is very large.
- It gives you an idea of the limiting behavior of an application, and hence is very **important** to measure the performance of your code.
- **We** calculate, how the time (or **space**) taken by an **algorithm** increases with the input size

Space ?

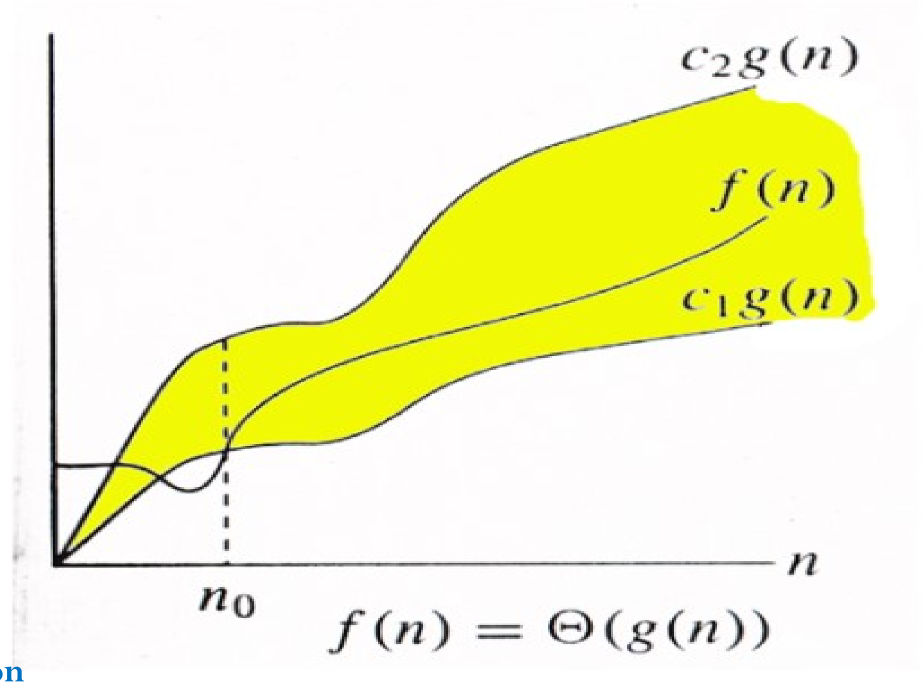

Computer Memory
Primary Memory
Secondary Memory
RAM
ROM non-volatile
HDD
SSD
Compact Disk
Floppy Disk
Magnet Tap
SRAM
DRAM
PROM EPROM
EEPROM

Asymptotic notation

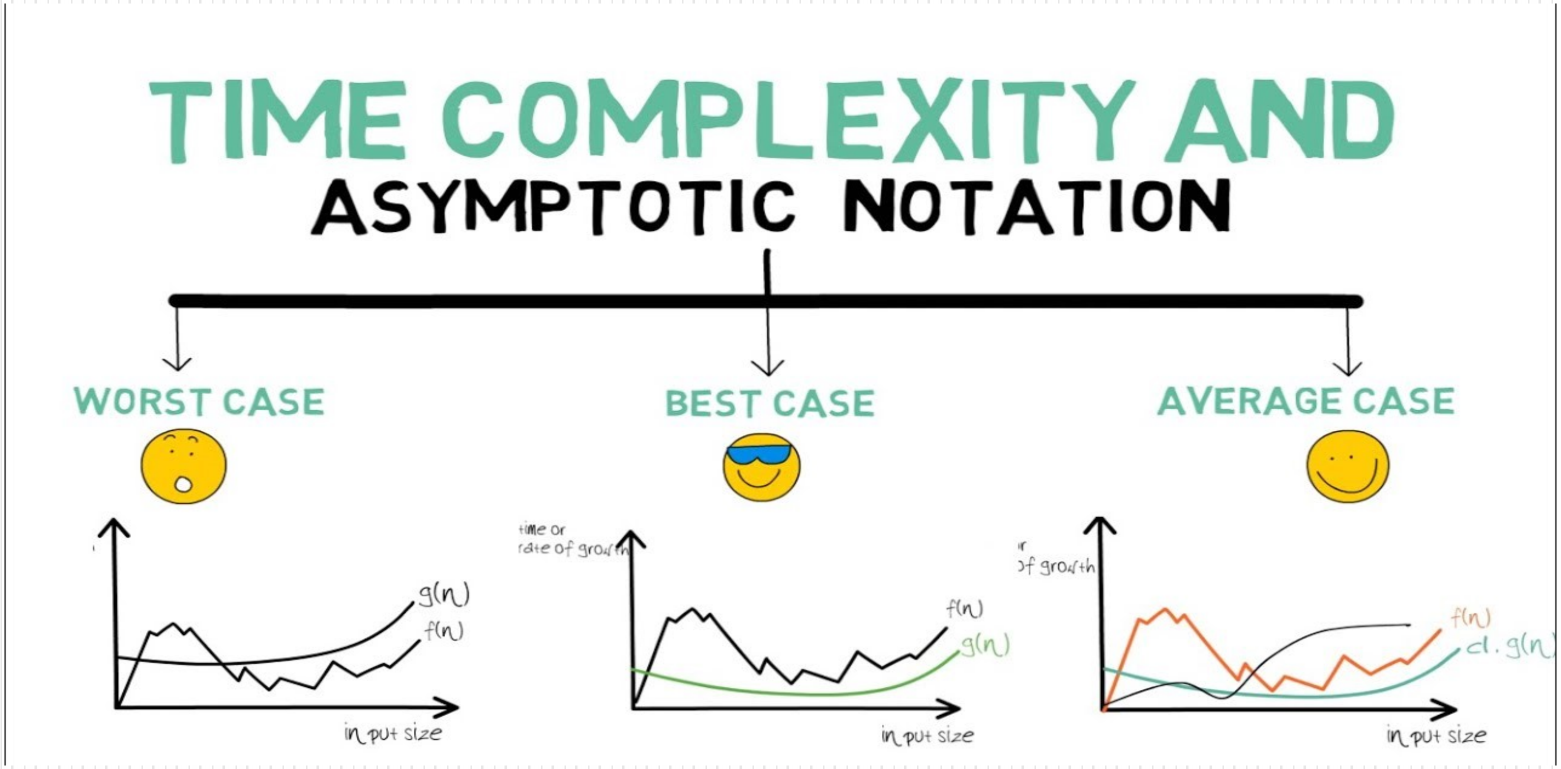
- Asymptotic notations are method used to estimate and represent the efficiency of an algorithm using simple formula.
- This can be useful for separating algorithms that leads to different amounts of work for large inputs.
- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2, \ldots\}$
- Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes.

Asymptotic notation

- Asymptotic notations give a limit (bound) to the growth of a function. This bound could be **upper** or **lower** bound for a function.
- A bound is asymptotically tight if it bounds the function with a constant difference.



Asymptotic notation



Asymptotic notation ?

- **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.
- We use **three** types of **asymptotic notations** to represent the growth of any algorithm, as input increases: Big Theta (Θ) Big Oh(O) Big Omega (Ω).
- **Big-O notation** represents the **upper bound** of the running time of an algorithm. Thus, it gives the **worst-case complexity** of an algorithm.
- **Omega notation** represents the **lower bound** of the running time of an algorithm. Thus, it provides the **best case complexity** of an algorithm.
- **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.

Properties of Asymptotic Notations :

1. General Properties :

- If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant.
- **Example:** f(n) = **$2n^2+5$** is **$O(n^2)$**
  then **7***f(n) = $7(2n^2+5) = 14n^2+35$ is also **$O(n^2)$** .
- Similarly this property satisfies for both Θ and Ω notation.

- We can say
  If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant.
  If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

Properties of Asymptotic Notations

2. Transitive Properties :

- If f(n) is O(g(n)) and g(n) is O(h(n)) then **f(n) = O(h(n))** .
- **Example:** if f(n) = n, g(n) = n^2 and h(n)=n^3
  n is O(n^2) and n^2 is O(n^3)
  then n is O(n^3)
- Similarly this property satisfies for both Θ and Ω notation.
- We can say
  If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
  If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

Properties of Asymptotic Notations

3. Reflexive Properties :

- Reflexive properties are always easy to understand after transitive.
- If f(n) is given then f(n) is O(f(n)). Since *MAXIMUM VALUE OF f(n) will be f(n) ITSELF !*
- Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.
- **Example**: f(n) = n^2 ; O(n^2) i.e O(f(n))

Similarly this property satisfies for both **Θ** and **Ω** notation.

We can say that:

If f(n) is given then f(n) is **Θ**(f(n)).

If f(n) is given then f(n) is **Ω** (f(n)).

Properties of Asymptotic Notations

4. Symmetric Properties :

If f(n) is $\Theta(g(n))$ then g(n) is $\Theta(f(n))$.

- Example: $f(n) = n^2$ and $g(n) = n^2$
  then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$
- **This property only satisfies for Θ notation.**

5. Transpose Symmetric Properties :

- If f(n) is $O(g(n))$ then g(n) is $\Omega(f(n))$.
- *Example:* $f(n) = n$, $g(n) = n^2$
  then n is $O(n^2)$ and n^2 is $\Omega(n)$
- **This property only satisfies for O and Ω notations.**

Properties of Asymptotic Notations

6. Some More Properties :

- 1.) If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$
- 2.) If $f(n) = O(g(n))$ and $d(n)=O(e(n))$

  then $f(n) + d(n) = O(\max(g(n), e(n)))$

  Example: $f(n) = n$ i.e $O(n)$

  $d(n) = n^2$ i.e $O(n^2)$

  then $f(n) + d(n) = n + n^2$ i.e $O(n^2)$

Order of growth

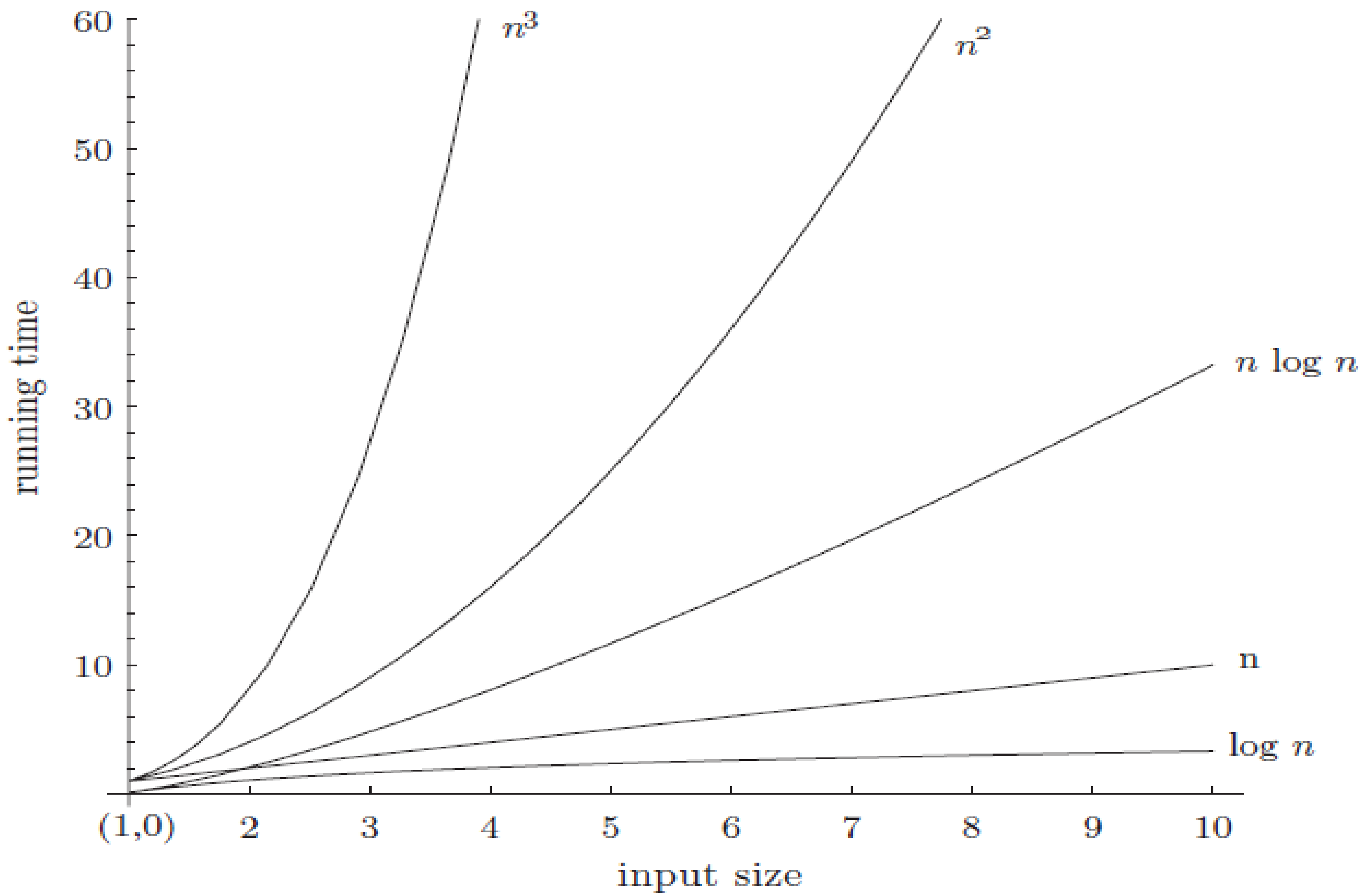
- Lower order item(s) are ignored, just keep the highest order item.
- The constant coefficient(s) are ignored.
- The rate of growth, or the order of growth, possesses the highest significance.
- Use $\Theta(n^2)$ to represent the worst case running time for insertion sort.
- Typical order of growth: $\Theta(1)$, $\Theta(\log n)$, $\Theta(\sqrt{n})$,$\Theta(n)$, $\Theta(n\log n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$
- Asymptotic notations: Θ, O, Ω, o, ω.

Practical Complexity

Analysing the running time of algorithms

- Suppose that we have two algorithms $A1$ and $A2$ of running times in the order of $n \log n$. Which one should we consider to be preferable to the other?
- Technically, since they have the same time complexity, we say that they have the same running time *within a multiplicative constant*, that is, the ratio between the two running times is constant.
- In some cases, the constant may be important and more detailed analysis of the algorithm or conducting some experiments on the behavior of the algorithm may be helpful.
- It may be necessary to investigate other factors, e.g. **space requirements** and **input distribution**.
- The latter is helpful in analyzing the behavior of an algorithm on the average.

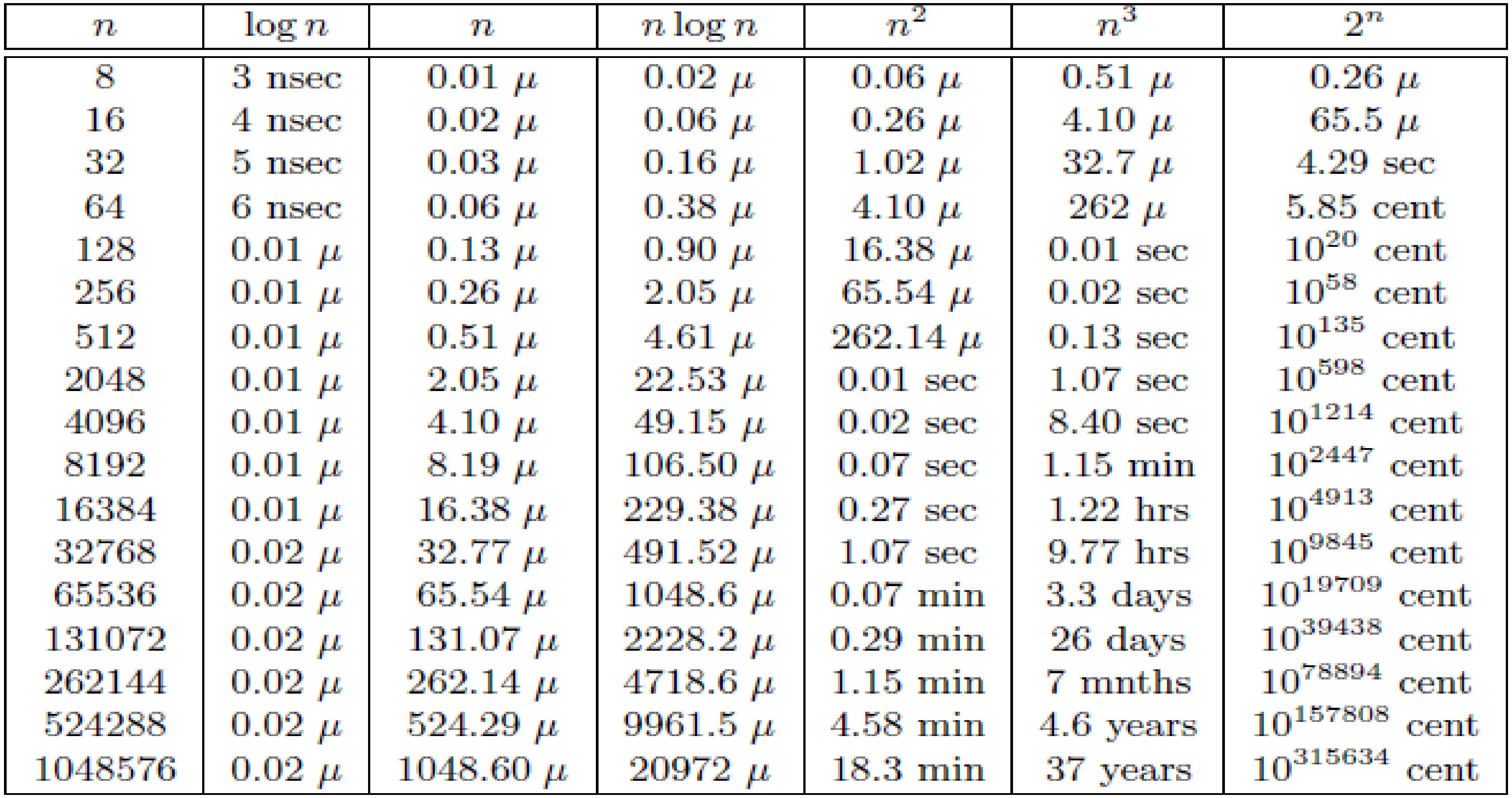
Growth of some typical functions that represent running times.



Observation

- Exponential and hyper-exponential functions grow much faster than the ones shown in the figure (previous slide).
- Higher order functions and exponential and hyper-exponential functions are not shown in the figure, even for moderate values of n.
- Functions of the form $\log k\ n; cn; cn^2; cn^3$ are called, respectively, *logarithmic*, *linear*, *quadratic* and *cubic*.
- Functions of the form n^c or $n^c \log k\ n$; $0 < c < 1$; are called ***sublinear***.
- Functions that lie between linear and quadratic, e.g. $n \log n$ and $n^{1:5}$, are called ***subquadratic***.

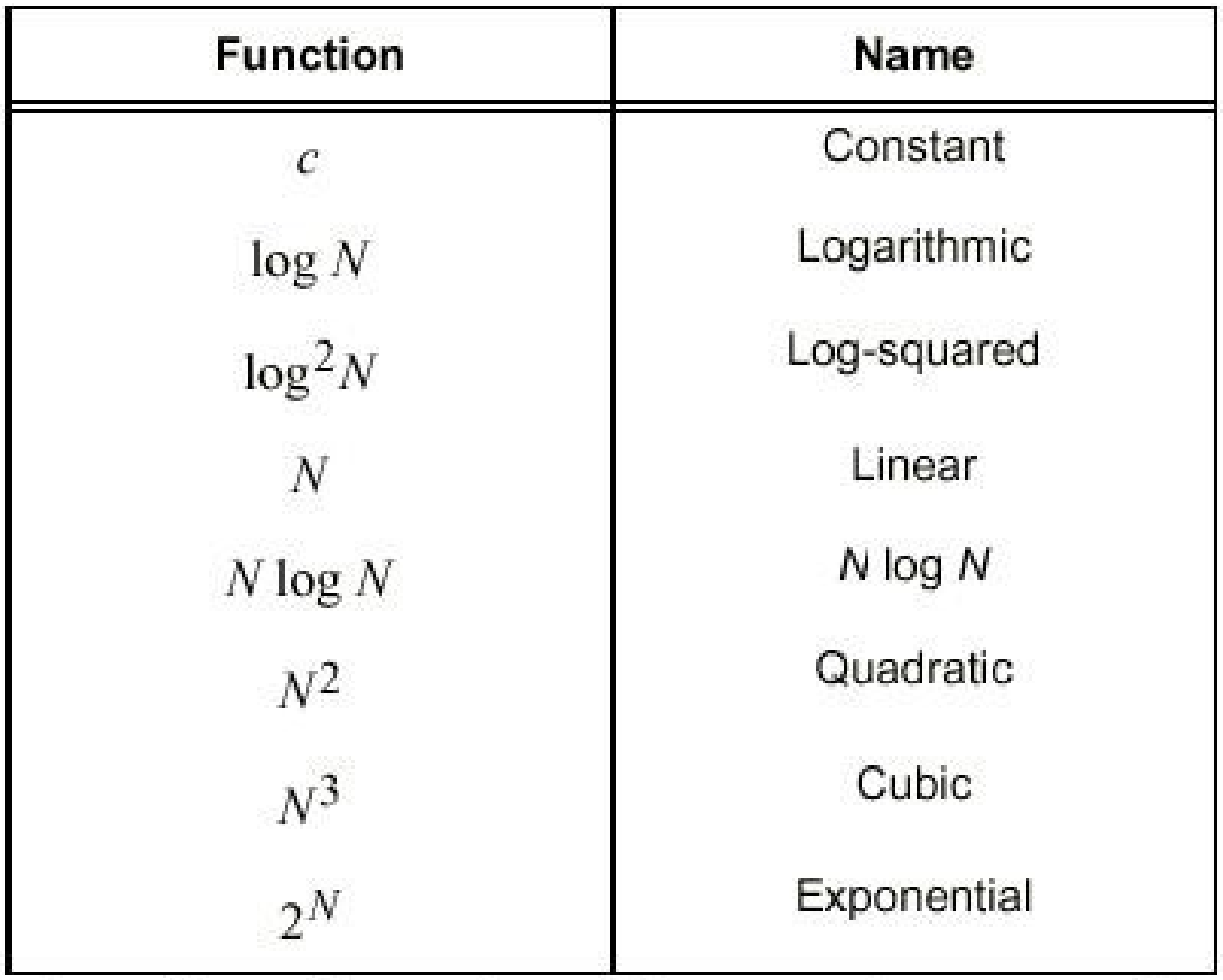
Running times for different sizes of input.



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Note: "nsec" stands for nanoseconds, " μ"is one microsecond and "cent" stands for centuries. The explosive running time (measured in centuries) when it is of the order 2^n.

Functions in order of growth rate



Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Recall the Hierarchy of Growth Rates

Easy

$$\boxed{c < \log^k N < N < N \log N < N^2 < N^3 <}$$

$$\boxed{2^N < 3^N < N! < N^N}$$

Hard

We can make a distinction between problems that have polynomial time algorithms and those that have algorithms that are worse than polynomial time.

Elementary operation

- **Definition:** We denote by an "**elementary operation**" any computational step whose cost is always upper bounded by a constant amount of time regardless of the input data or the algorithm used.
- The following operations on fi*xed-size* operands are examples of elementary operation.

❑ Arithmetic operations: addition, subtraction, multiplication and division.

❑ Comparisons and logical operations.

❑ Assignments, including assignments of pointers when, say, traversing a list or a tree.

- In order to formalize the notions of ***order of growth*** and ***time complexity***, special mathematical notations have been widely used.
- These notations make it convenient to compare and analyze running times with minimal use of mathematics and cumbersome calculations.

Elementary operation

- An **elementary operation** in an algorithm is called a *basic operation* if it is of highest frequency to within a constant factor among all other elementary operations.
- When analyzing searching and sorting algorithms, we may choose **the element comparison operation** *if it is an elementary operation*.
- In matrix multiplication algorithms, we select the operation of scalar multiplication.
- In traversing a linked list, we may select the “operation" of setting or updating a pointer.
- In graph traversals, we may choose the “action" of visiting a node, and count the number of nodes visited.

BIG-OH COMPLEXITY

Big O notation

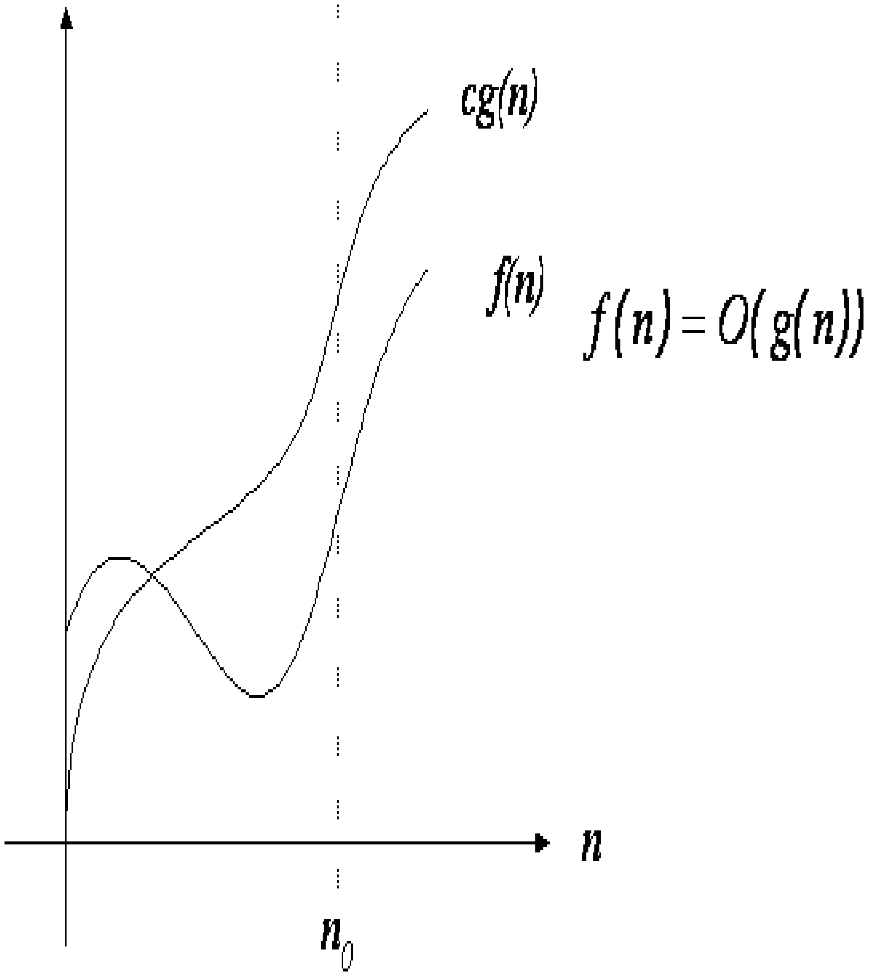
- Big O notation or Big Oh notation, and also **Landau notation** or asymptotic notation, is a mathematical notation used to describe the asymptotic behavior of functions.
- Its purpose is to characterize a function's behavior for very large (or very small) inputs in a simple but rigorous way that enables comparison to other functions.
- More precisely, the symbol O is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function.

Big O notation

- There are also other symbols o, Ω, ω, and Θ for various other upper, lower, and tight bounds.
- Two main areas of application: in mathematics, it is usually used to characterize the residual term of a truncated infinite series, especially an asymptotic series, and in computer science, it is useful in the analysis of the complexity of algorithms.
- Informally, the O notation is commonly employed to describe an asymptotic tight bound, but tight bounds are more formally and precisely denoted by the Θ (capital theta) symbol as described below.

BIG Oh notation

This is a standard notation that has been developed to represent functions which *bound the computing time for algorithm* and it is used to define the worst case running time of an algorithm and concerned with very large values of ***n***.



BIG Oh notation

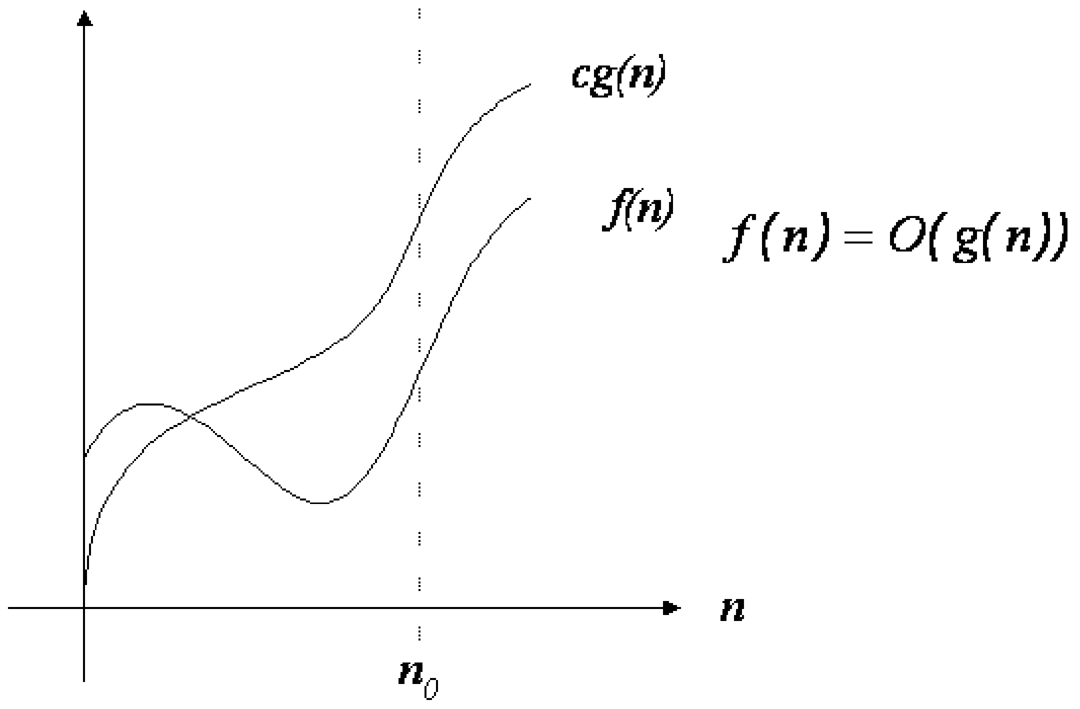
- Let *f* and *g* be functions from positive integers to positive integers. We say *f* is *O(g(n))* (read: "*f* is order *g*") if *g* is an upper bound on *f*: there exists a fixed constant ***c*** and a fixed $\boldsymbol{n_0}$ such that for all $\boldsymbol{n} \geq \boldsymbol{n_0}$,

$$f(n) \leq cg(n).$$

- Equivalently, *f* is *O(g(n))* if the function *f(n)/g(n)* is bounded above by some constant.

Big oh notation(O)

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



Big oh notation(O)

- The meaning of an expression like $O(n^2)$ is really a set of functions: all the functions that are $O(n^2)$.
- When we say that $f(n)$ is $O(n^2)$, we mean that $f(n)$is a member of this set.
- It is also common to write this as $f(n) = O(g(n))$ although it is not really an equality.

Big oh notation(O)

Definition . Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $O(g(n))$ if there exists a natural number n_0 and a constant $c > 0$ such that

$$\forall\ n \geq n_0,\ f(n) \leq cg(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \text{ implies } f(n) = O(g(n)).$$

Informally, this definition says that f grows no faster than some constant times g. The O-notation can also be used in equations as a simplification tool. For instance, instead of writing

$$f(n) = 5n^3 + 7n^2 - 2n + 13,$$

we may write

$$f(n) = 5n^3 + O(n^2).$$

Big oh notation(O)

Examples – ('=' symbol readed as 'is' instead of 'equal')

- The function **$3n+2$** = **O(n)** as **$3n+2 \le 4n$** for all **$n \ge 2$**
- The function $3n+3$ = **O(n)** as $3n+3 \le 4n$ for all **$n \ge 3$**
- The function **$100n$**$+6 = O(n)$ as $100n+6 \le$ **101**n for all $n \ge 6$
- The function **$10n^2$** $+4n+2 = O(n^2)$ as $10n^2 +4n+2 \le$ **$11n^2$** for all **$n \ge 5$**
- The function **$1000n^2$** $+100n-6 = O(n^2)$ as $1000n^2 +100n-6 \le$ **$1001n^2$** for all $n \ge 100$
- The function $6*2^n +n^2 = O(2^n)$ as $6*2^n +n^2 \le 7*2^n$ for all **$n \ge 4$**
- The function $3n+3 = O(n^2)$ as $3n+3 \le 3n^2$, for $n \ge 2$.
- The function $10n^2+4n +2 = O (n^4)$ as $10n^2+4n +2 \le 10n^4$,for $n > 2$
- $3n + 2$ # $O(1)$ as $3n +2$ is not less than or equal to c for any constant c and all $n > n_0$.
- $10n2+4n +2$ # $O(n)$

Big oh notation(O)

- Consider the job offers from two companies. The first company offer contract that will double the salary every year. The second company offers you a contract that gives a raise of Rs. 1000 per year.

This scenario can be represented with Big-O notation as:

- For first company, New salary = Salary X 2n (where n is total service years) Which can be denoted with Big-O notation as O(2n)

- For second company, New salary = Salary +1000n (where n is total service years) Which can be denoted with Big-O notation as O(n)

f(n) is a polynomial in n

Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{aligned} f(n) &\le \textstyle\sum_{i=0}^{m} |a_i| n^i \\ &\le n^m \textstyle\sum_{i=0}^{m} |a_i| n^{i-m} \\ &\le n^m \textstyle\sum_{i=0}^{m} |a_i| \qquad \text{for } n \ge 1 \end{aligned}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed). □

Proving asymptotic bounds

Theorem 1

Let $f(n)$ and $g(n)$ be functions such that

$$\lim_{x\to\infty} \frac{f(n)}{g(n)} = A.$$

Then

1. *If $A = 0$, then $f(n) = O(g(n))$, and $f(n) \neq \Theta(g(n))$.*
2. *If $A = \infty$, then $f(n) = \Omega(g(n))$, and $f(n) \neq \Theta(g(n))$.*
3. *If $A \neq 0$ is finite, then $f(n) = \Theta(g(n))$.*

Proving asymptotic bounds

1. Prove that if $f(x) = O(g(x))$, and $g(x) = O(f(x))$, then $f(x) = \Theta(g(x))$.

   Proof:

   If $f(x) = O(g(x))$, then there are positive constants c_2 and n_0' such that

$$0 \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0'$$

   Similarly, if $g(x) = O(f(x))$, then there are positive constants c_1' and n_0'' such that

$$0 \le g(n) \le c_1'\, f(n) \text{ for all } n \ge n_0''.$$

   We can divide this by c_1' to obtain

$$0 \le \frac{1}{c_1'} g(n) \le f(n) \text{ for all } n \ge n_0''.$$

   Setting $c_1 = 1/c_1'$ and $n_0 = \max(n_0', n_0'')$, we have

$$0 \le c_1 g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0.$$

   Thus, $f(x) = \Theta(g(x))$.

Proving asymptotic bounds

2. Let $f(x) = O(g(x))$ and $g(x) = O(h(x))$. Show that $f(x) = O(h(x))$.

Proof:

If $f(x) = O(g(x))$, then there are positive constants c_1 and n_0' such that

$$0 \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0',$$

and if $g(x) = O(h(x))$, then there are positive constants c_2 and n_0'' such that

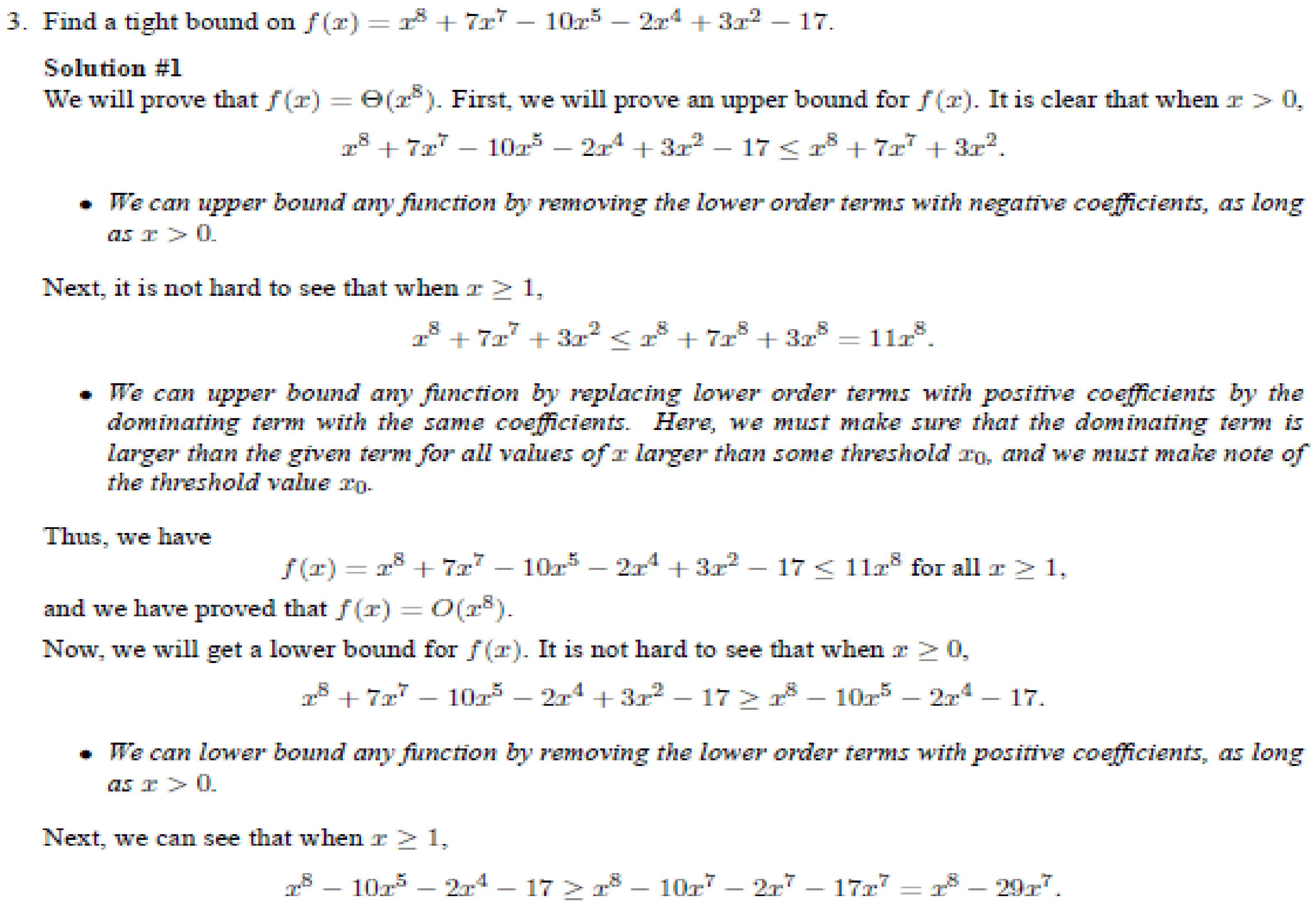
$$0 \leq g(n) \leq c_2\, h(n) \text{ for all } n \geq n_0''.$$

Set $n_0 = \max(n_0', n_0'')$ and $c_3 = c_1\, c_2$. Then

$$0 \leq f(n) \leq c_1\, g(n) \leq c_1\, c_2\, h(n) = c_3\, h(n) \text{ for all } n \geq n_0.$$

Thus $f(x) = O(h(x))$.

Proving asymptotic bounds



3. Find a tight bound on $f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17$.

Solution #1
We will prove that $f(x) = \Theta(x^8)$. First, we will prove an upper bound for $f(x)$. It is clear that when $x > 0$,

$$x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \le x^8 + 7x^7 + 3x^2.$$

- *We can upper bound any function by removing the lower order terms with negative coefficients, as long as $x > 0$.*

Next, it is not hard to see that when $x \ge 1$,

$$x^8 + 7x^7 + 3x^2 \le x^8 + 7x^8 + 3x^8 = 11x^8.$$

- *We can upper bound any function by replacing lower order terms with positive coefficients by the dominating term with the same coefficients. Here, we must make sure that the dominating term is larger than the given term for all values of x larger than some threshold x_0, and we must make note of the threshold value x_0.*

Thus, we have

$$f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \le 11x^8 \text{ for all } x \ge 1,$$

and we have proved that $f(x) = O(x^8)$.

Now, we will get a lower bound for $f(x)$. It is not hard to see that when $x \ge 0$,

$$x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \ge x^8 - 10x^5 - 2x^4 - 17.$$

- *We can lower bound any function by removing the lower order terms with positive coefficients, as long as $x > 0$.*

Next, we can see that when $x \ge 1$,

$$x^8 - 10x^5 - 2x^4 - 17 \ge x^8 - 10x^7 - 2x^7 - 17x^7 = x^8 - 29x^7.$$

Proving asymptotic bounds

- *We can lower bound any function by replacing lower order terms with negative coefficients by a sub-dominating term with the same coefficients. (By sub-dominating, I mean one which dominates all but the dominating term.) Here, we must make sure that the sub-dominating term is larger than the given term for all values of x larger than some threshold x_0, and we must make note of the threshold value x_0. Making a wise choice for which sub-dominating term to use is crucial in finishing the proof.*

Next, we need to find a value $c > 0$ such that $x^8 - 29x^7 \geq cx^8$. Doing a little arithmetic, we see that this is equivalent to $(1-c)x^8 \geq 29x^7$. When $x \geq 1$, we can divide by x^7 and obtain $(1-c)x \geq 29$. Solving for c we obtain

$$c \leq 1 - \frac{29}{x}.$$

If $x \geq 58$, then $c = 1/2$ suffices. We have just shown that if $x \geq 58$, then

$$f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17 \geq \frac{1}{2}x^8.$$

Thus, $f(x) = \Omega(x^8)$. Since we have shown that $f(x) = \Omega(x^8)$ and that $f(x) = O(x^8)$, we have shown that $f(x) = \Theta(x^8)$.

Proving asymptotic bounds

3. Find a tight bound on $f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17$.

Solution #2

We guess (or know, if we read Solution #1) that $f(x) = \Theta(x^8)$. To prove this, notice that

$$\begin{aligned}
\lim_{x\to\infty} \frac{x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17}{x^8} &= \lim_{x\to\infty} \frac{x^8}{x^8} + \frac{7x^7}{x^8} - \frac{10x^5}{x^8} - \frac{2x^4}{x^8} + \frac{3x^2}{x^8} - \frac{17}{x^8} \\
&= \lim_{x\to\infty} 1 + \frac{7}{x} - \frac{10}{x^3} - \frac{2}{x^4} + \frac{3}{x^6} - \frac{17}{x^8} \\
&= \lim_{x\to\infty} 1 + 0 - 0 - 0 + 0 - 0 = 1
\end{aligned}$$

Thus, $f(x) = \Theta(x^8)$ by the Theorem.

Proving asymptotic bounds

4. Find a tight bound on $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21$.

Solution #1

It is clear that when $x \geq 1$,

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \leq x^4 + 12x^2 + 15x \leq x^4 + 12x^4 + 15x^4 = 28x^4.$$

Also,

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \geq x^4 - 23x^3 - 21 \geq x^4 - 23x^3 - 21x^3 = x^4 - 44x^3 \geq \frac{1}{2}x^4,$$

whenever

$$\frac{1}{2}x^4 \geq 44x^3 \Leftrightarrow x \geq 88.$$

Thus

$$\frac{1}{2}x^4 \leq x^4 - 23x^3 + 12x^2 + 15x - 21 \leq 28x^4, \text{ for all } x \geq 88.$$

We have shown that $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21 = \Theta(x^4)$.

Proving asymptotic bounds

4. Find a tight bound on $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21$.

Solution #2

From Solution #1 we already know that $f(x) = \Theta(x^4)$. We verify this by noticing that

$$\begin{aligned}
\lim_{x\to\infty} x^4 - 23x^3 + 12x^2 + 15x - 21 &= \lim_{x\to\infty} \frac{x^4}{x^4} - \frac{23x^3}{x^4} + \frac{12x^2}{x^4} + \frac{15x}{x^4} - \frac{21}{x^4} \\
&= \lim_{x\to\infty} 1 - \frac{23}{x} + \frac{12}{x^2} + \frac{15}{x^3} - \frac{21}{x^4} \\
&= \lim_{x\to\infty} 1 - 0 + 0 + 0 - 0 = 1
\end{aligned}$$

Proving asymptotic bounds

5. Show that $\log x = O(x)$.

Proof:

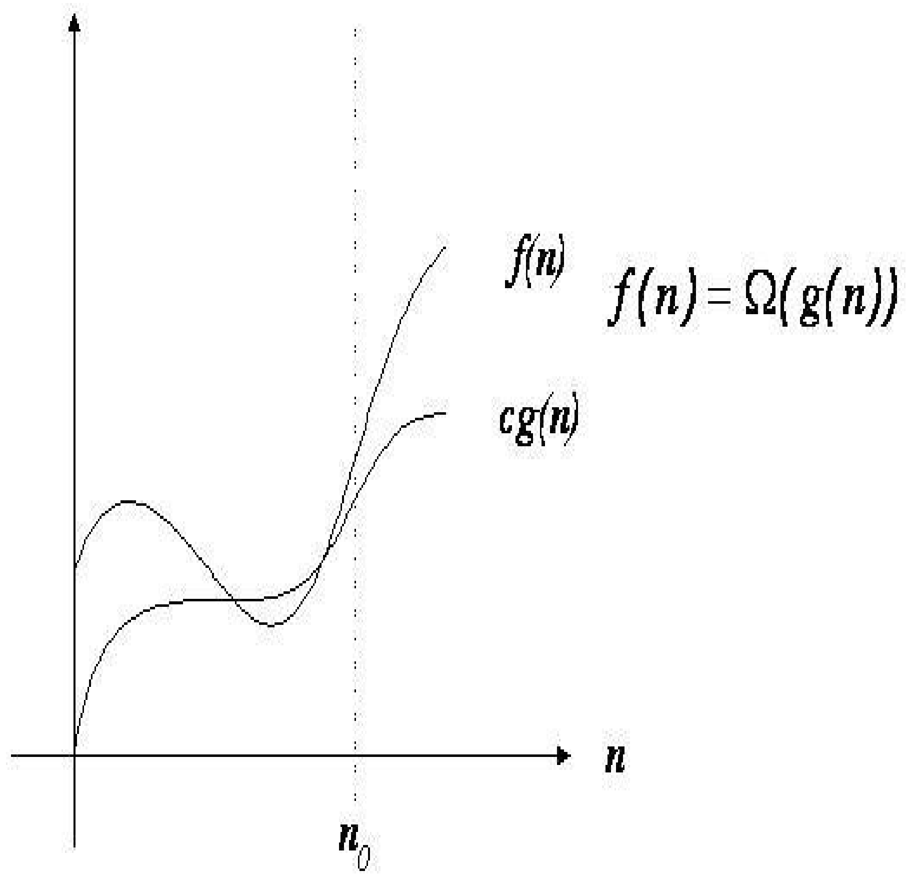
$$\lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{\frac{1}{x}}{1} = \lim_{x\to\infty} \frac{1}{x} = 0$$

Therefore, $\log n = O(n)$.

Lower Bounds: Omega (Ω)

- Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.
- This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.
- In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take at least this much time to complete it's execution. It can definitely take more time than this too.
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

BIG OMEGA notation (Ω)



- This notation is used to describe the **best case running time** of algorithm and concerned with the very large values of n

BIG OMEGA notation (Ω)

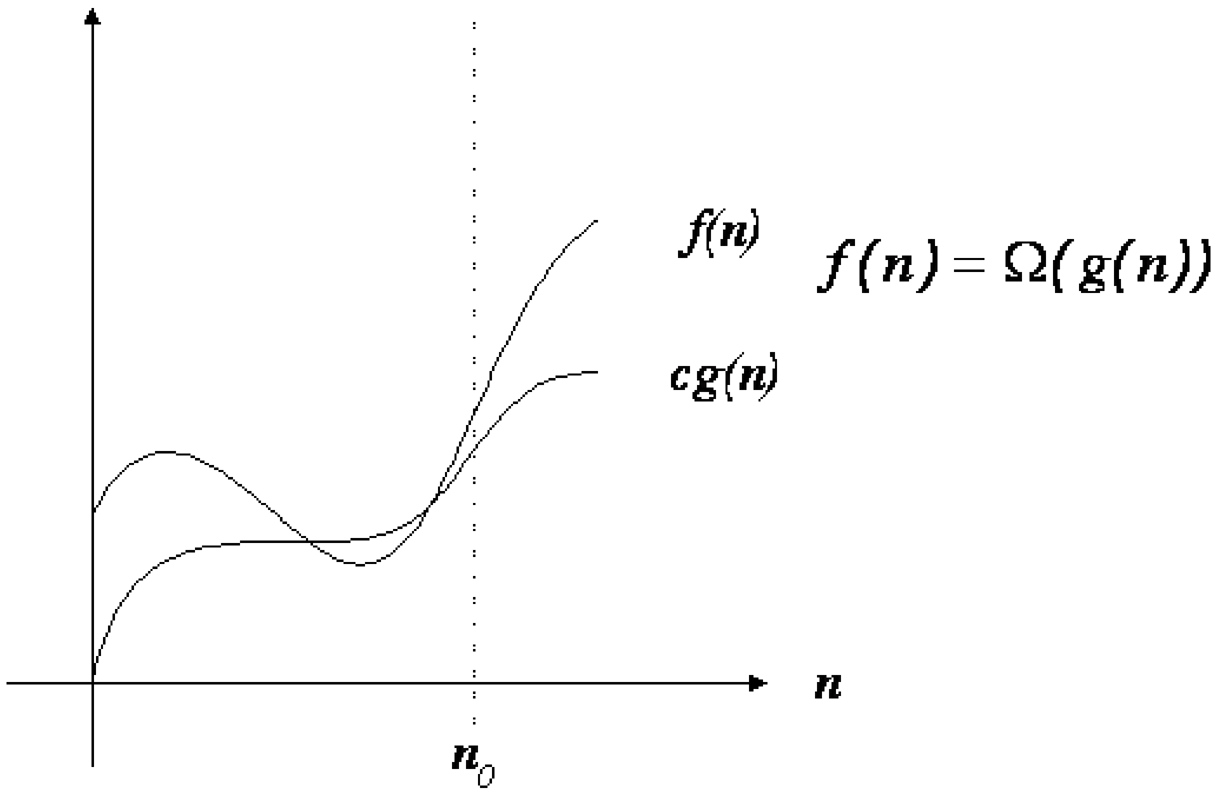
- We say that f is $\Omega(g(n))$ (read: "f is omega of g") if g is a *lower* bound on f for large n. Formally, f is Ω(g) if there is a fixed constant c and a fixed n_0 such that for all $n > n_0$,

$$cg(n) \leq f(n)$$

- For example, any polynomial whose highest exponent is n^k is $\Omega(n^k)$.
- If $f(n)$ is $\Omega(g(n))$ then $g(n)$ is $O(f(n))$. If $f(n)$ is $o(g(n))$ then $f(n)$ is *not* $\Omega(g(n))$.

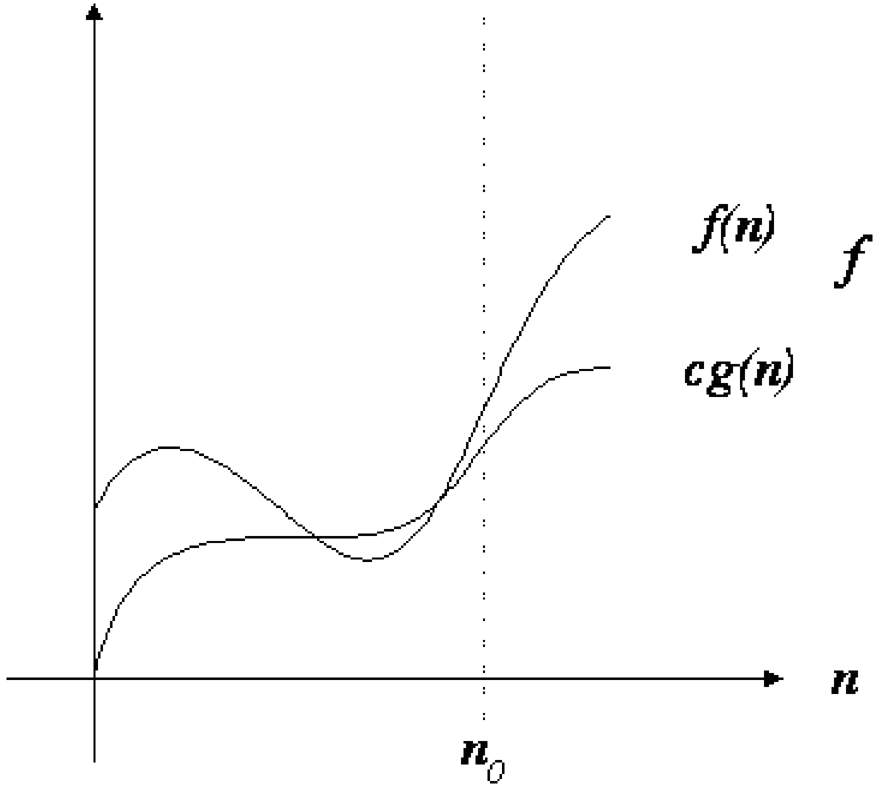
Big omega notation (Ω)

This notation is used to describe the best case running time of algorithm and concerned with the very large values of N.



Definition: Big omega (Ω)

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



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

Big omega (Ω)

Definition : Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Omega(g(n))$ if there exists a natural number n_0 and a constant $c > 0$ such that

$$\forall\ n \geq n_0,\ f(n) \geq cg(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \text{ implies } f(n) = \Omega(g(n)).$$

Informally, this definition says that f grows at least as fast as some constant times g. It is clear from the definition that

$$f(n) \text{ is } \Omega(g(n)) \text{ if and only if } g(n) \text{ is } O(f(n)).$$

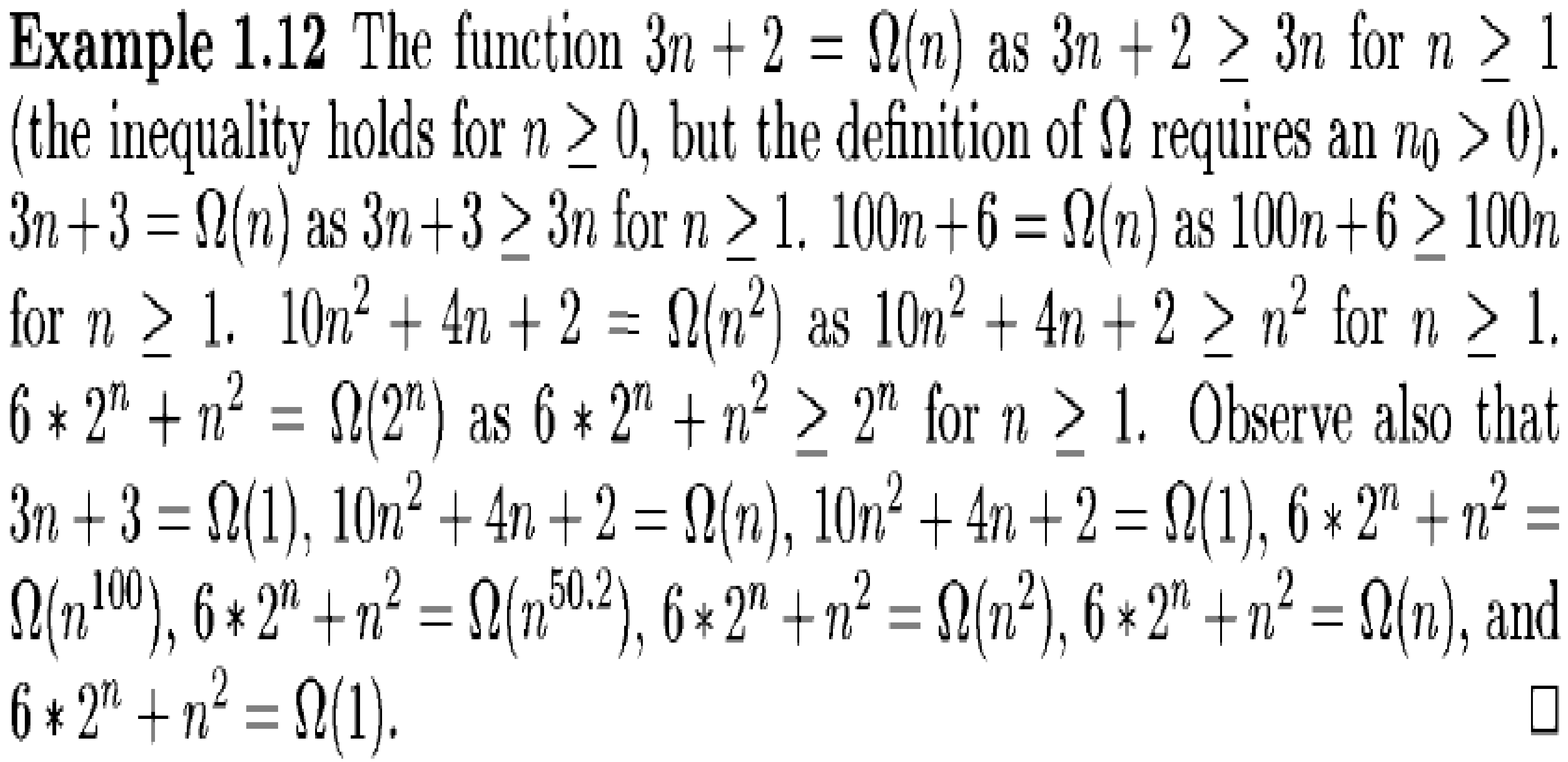
Big omega (Ω)

- The function **3n+2** = Ω(n) as 3n+2 ≥ **3n** for all **n ≥ 1**
- The function 3n+3 = Ω(n) as 3n+3 ≥ **3n** for all n ≥ 1
- The function 100n+6 = Ω(n) as 100n+6 ≥ 100n for all n≥1
- The function $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for all n≥1
- The function $6*2^n + n^2 = \Omega(2^n)$ as $6*2^n + n^2 \geq 2^n$ for all n≥1

Observation

- The function 3n+3 = Ω(1)
- The function 10n2 +4n+2 = Ω(n)
- The function 10n2 +4n+2 = Ω(1)

Big omega (Ω)



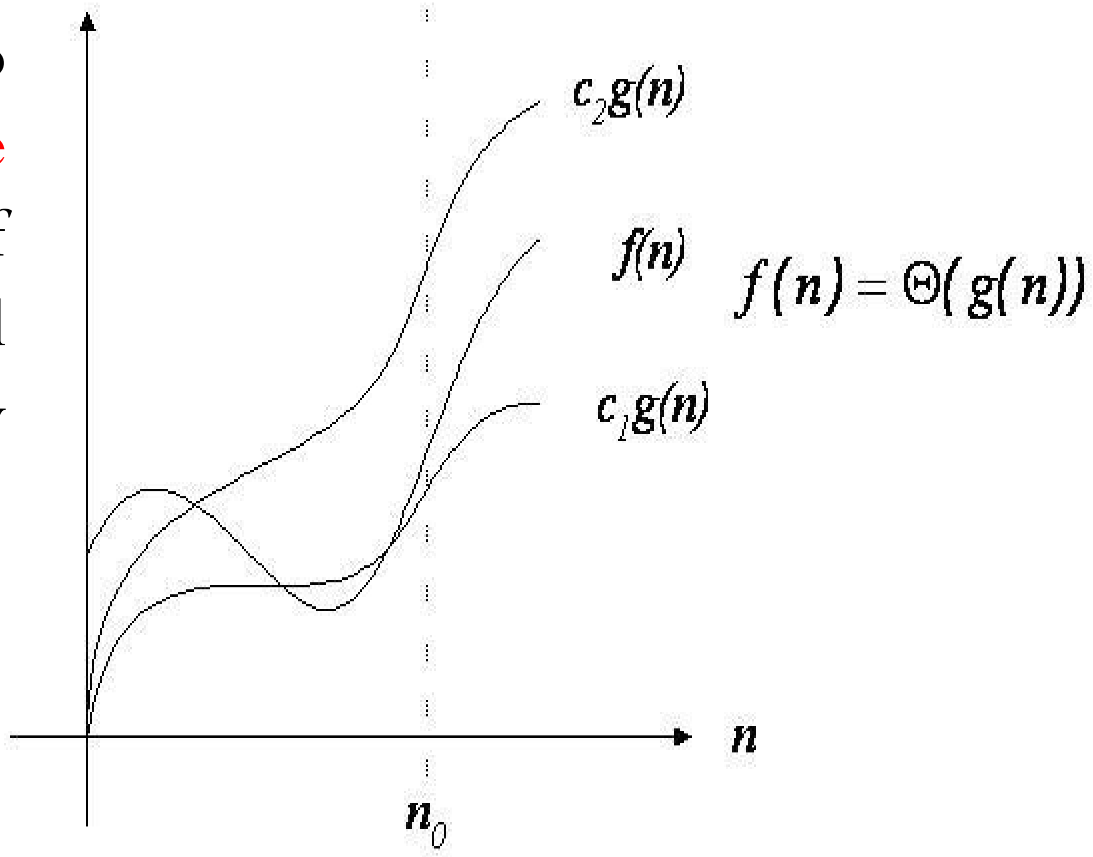
Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$. $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. $\square$

Big omega (Ω)

Theorem 1.3 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

BIG THETA Notation

- This Notation is used to describe the **average case running time** of algorithm and concerned with very large values of n.



The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time

Big theta notation(Θ)

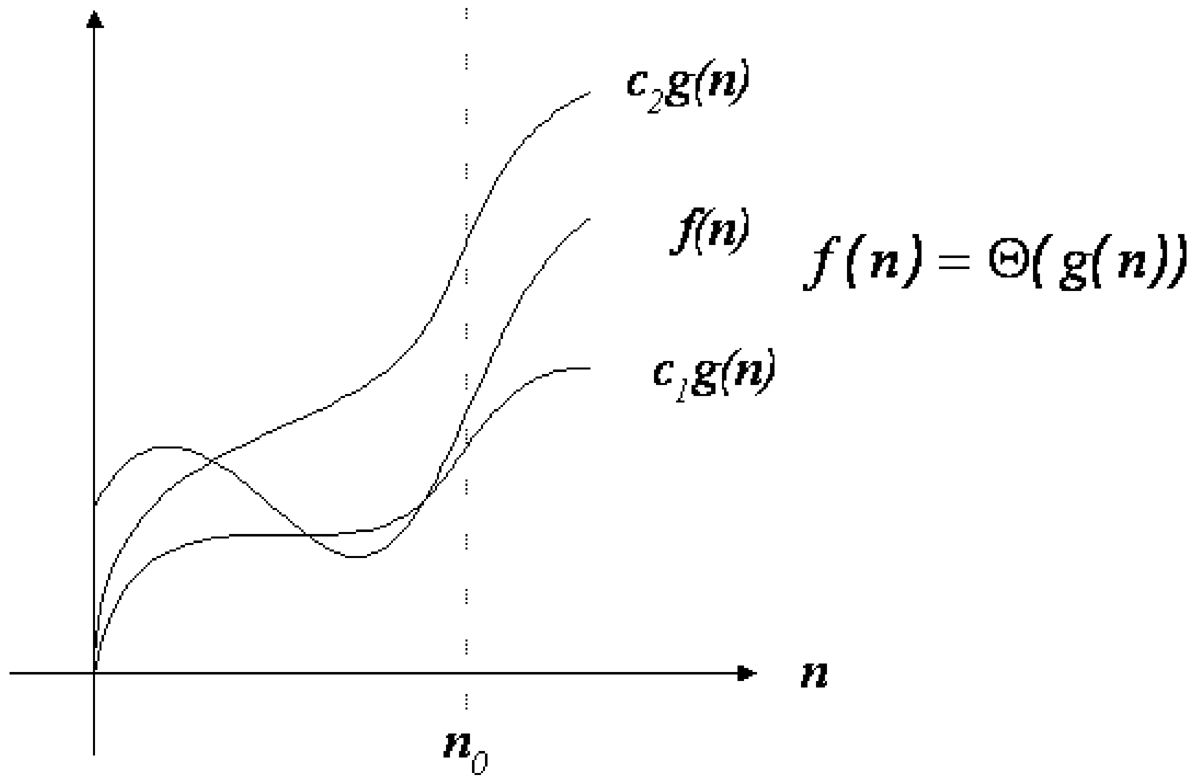
- We say that *f* is $\Theta(g(n))$ (read: "f is theta of g") if *g* is an accurate characterization of *f* for large *n:* it can be scaled so it is both an upper and a lower bound of *f*. That is, *f* is both $O(g(n))$ and $\Omega(g(n))$.
- Expanding out the definitions of Ω and *O*, *f* is $\Theta(g(n))$ if there are fixed constants c_1 and c_2 and a fixed n_0 such that for all $n > n_0$,

$$c_1 g(n) \le f(n) \le c_2\, g(n)$$

- For example, any polynomial whose highest exponent is n^k is $\Theta(n^k)$. If *f* is $\Theta(g)$, then it is $O(g)$ but not $o(g)$.
- Sometimes people use $O(g(n))$ a bit informally to mean the stronger property $\Theta(g(n))$; however, the two are different.

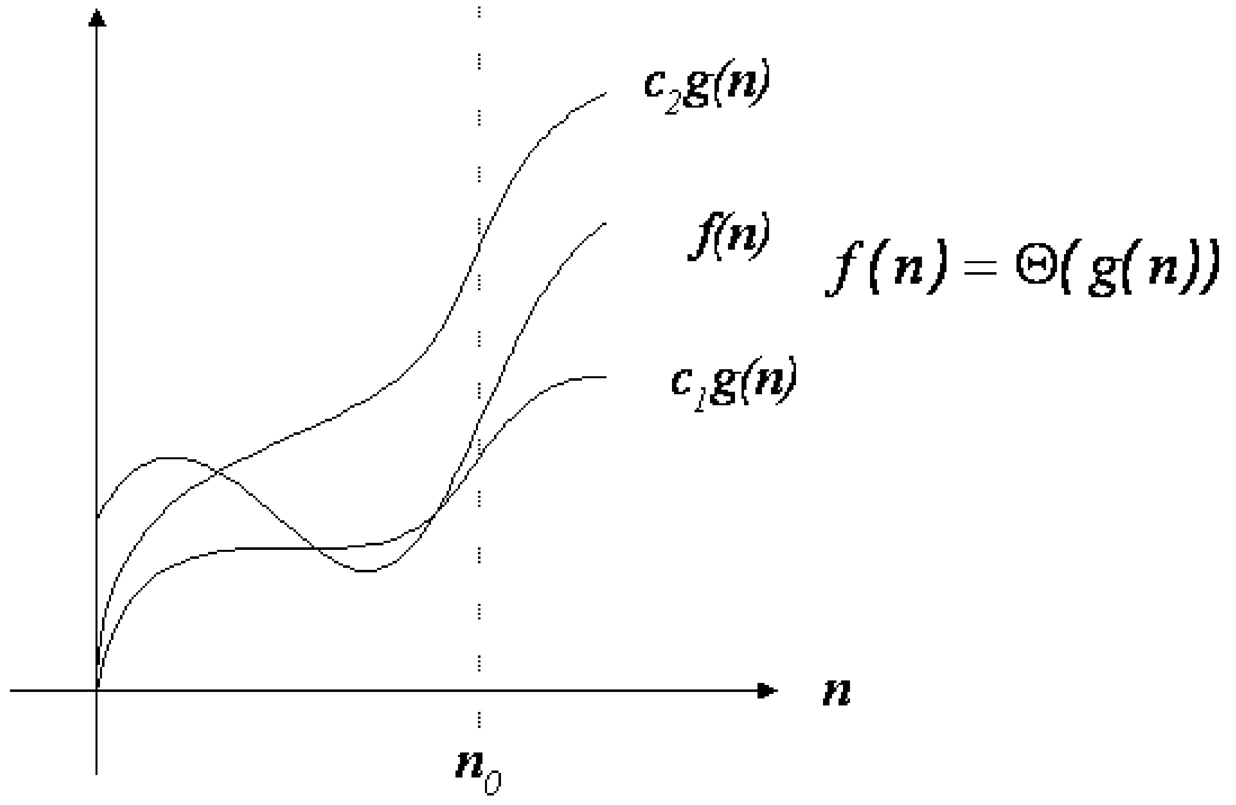
Big theta notation(Θ)

This Notation is used to describe the average case running time of algorithm and concerned with very large values of n.



Definition: Big theta (Θ)

$\Theta(g(n)) = \{f(n)$: 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 $n \ge n_0\}$



Big theta (Θ)

Definition : Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Theta(g(n))$ if there exists a natural number n_0 and two positive constants c_1 and c_2 such that

$$\forall\, n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ implies } f(n) = \Theta(g(n)),$$

where c is a *constant strictly greater than 0*.

An important consequence of the above definition is that

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

Big theta (Θ)

- The function $3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$ and $3n+2 \leq 4n$ for all $n \geq 2$ So, $c1=3$, $c2=4$ and $n0=2$.
- The function $3n+3 = \Theta(n)$
- The function $10n^2 + 4n + 2 = \Theta(n^2)$
- The function $6 * 2^n + n^2 = \Theta(2^n)$

Big theta (Θ)

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

Examples

Example 5 Let $f(n) = 10n^2 + 20n$. Then, $f(n) = O(n^2)$ since for all $n \geq 1$, $f(n) \leq 30n^2$. $f(n) = \Omega(n^2)$ since for all $n \geq 1$, $f(n) \geq n^2$. Also, $f(n) = \Theta(n^2)$ since for all $n \geq 1$, $n^2 \leq f(n) \leq 30n^2$. We can also establish these three relations using the limits as mentioned above. Since $\lim_{n\to\infty}(10n^2 + 20n)/n^2 = 10$, we see that $f(n) = O(n^2)$, $f(n) = \Omega(n^2)$ and $f(n) = \Theta(n^2)$.

Examples

Example 6 In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$. Then, $f(n) = \Theta(n^k)$. Recall that this implies that $f(n) = O(n^k)$ and $f(n) = \Omega(n^k)$.

Example 7 Since

$$\lim_{n\to\infty} \frac{\log n^2}{n} = \lim_{n\to\infty} \frac{2\log n}{n} = \lim_{n\to\infty} \frac{2}{\ln 2} \frac{\ln n}{n} = \frac{2}{\ln 2} \lim_{n\to\infty} \frac{1}{n} = 0$$

(differentiate both numerator and denominator), we see that $f(n)$ is $O(n)$, but not $\Omega(n)$. It follows that $f(n)$ is not $\Theta(n)$.

Example 8 Since $\log n^2 = 2\log n$, we immediately see that $\log n^2 = \Theta(\log n)$. In general, for any *fixed constant* k, $\log n^k = \Theta(\log n)$.

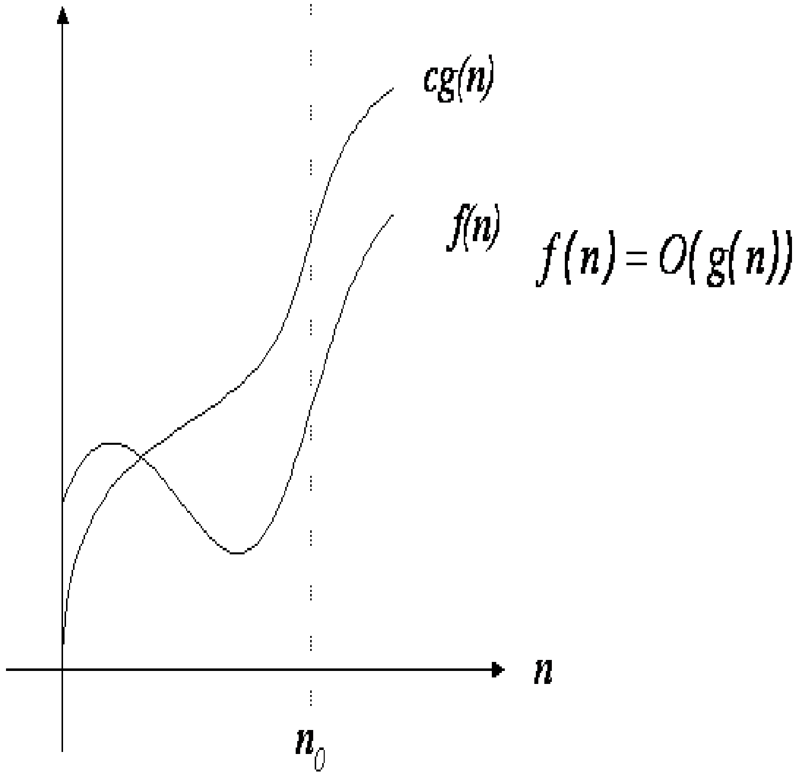
Example 9 Any constant function is $O(1), \Omega(1)$ and $\Theta(1)$.

Example 10 It is easy to see that 2^n is $\Theta(2^{n+1})$. This is an example of many functions that satisfy $f(n) = \Theta(f(n+1))$.

Calculating time complexity

A simple program fragment

1. int
2. sum(int n)
3. {
4. int i, partialsum;
5. partialsum = 0;
6. for(i=1; i<= n; i++)
7. partialsum + = i*i*i;
8. return partialsum
9. }



$6n+4 = O(n)$; as $6n+4 \leq 7n$ for all $n \geq 4$

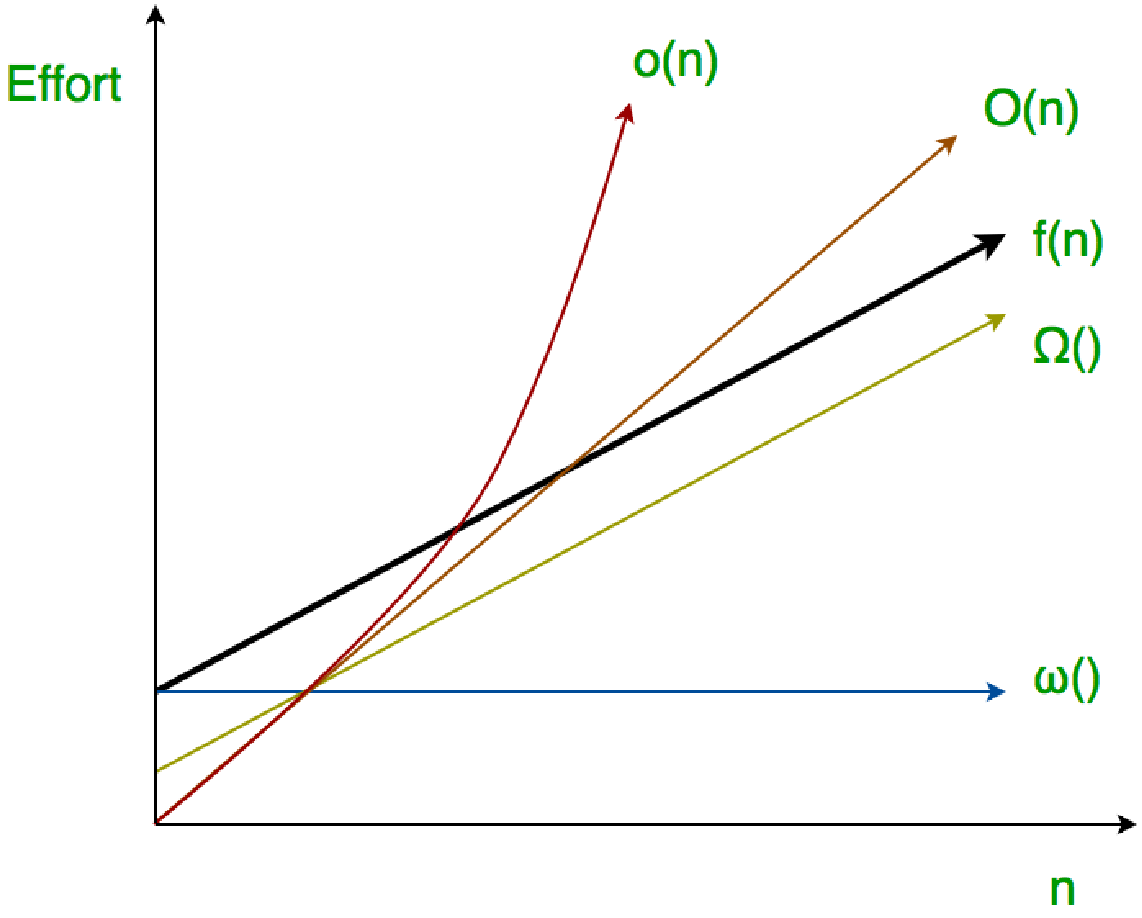
General rules

- *FOR loops*: the running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- *Nested FOR loop*: The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the size of all the for loops.
- Consecutive statements: Just add (which means that the maximum is the one that counts
- IF/ELSE: The running time of an If /else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.
  - **If (condition)**
  - **S1**
  - **Else**
  - **S2**

Little o asymptotic notation

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.
- **Definition :** Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) Ε o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).
- Thus, little o() means **loose upper-bound** of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

Little o asymptotic notation



In mathematical relation,

f(n) = o(g(n)) means

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

Little o asymptotic notation

Definition Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$ if for *every* constant $c > 0$ there exists a positive integer n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$. Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

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

Informally, this definition says that $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. It follows from the definition that

$$f(n) = o(g(n)) \text{ if and only if } f(n) = O(g(n)), \text{ but } g(n) \neq O(f(n)).$$

For example, $n \log n$ is $o(n^2)$ is equivalent to saying that $n \log n$ is $O(n^2)$ but n^2 is *not* $O(n \log n)$.

We also write $f(n) \prec g(n)$ to denote that $f(n)$ is $o(g(n))$. Using this notation, we can concisely express the following hierarchy of complexity classes.

$$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{3/4} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}.$$

Little o asymptotic notation

- **Is $7n + 8 = o(n^2)$?**
  In order for that to be true, for any c, we have to be able to find an n_0 that makes
  $f(n) < c * g(n)$ asymptotically true.
  lets took some example,
  If c = 100,we check the inequality is clearly true. If c = 1/100 , we'll have to use a little more imagination, but we'll be able to find an n_0.
- (Try $n_0 = 1000$.) From these examples, the conjecture appears to be correct.
  then check limits,

$$\lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} (7n + 8)/(n^2) = \lim_{n \to \infty} 7/2n = 0 \text{ (l'hospital)}$$

Little ω asymptotic notation

- **Definition :** Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n_0.
- f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions. In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n).
- The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds.
- Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth. We use ω notation to denote a lower bound that is not asymptotically tight. And, f(n) ∈ ω(g(n)) if and only if **g(n) = o((f(n)).**

Little ω asymptotic notation

- **Prove that $4n + 6 = \omega(1)$;**

  the little omega(ο) running time can be proven by applying limit formula given below.

  if $\lim_{n\to\infty} f(n)/g(n) = \infty$ then functions f(n) is $\omega(g(n))$

  here,we have functions $f(n)=4n+6$ and $g(n)=1$

  $\lim_{n\to\infty} (4n+6)/(1) = \infty$

  and,also for any c we can get n0 for this inequality $0 <= c*g(n) < f(n)$, $0 <= c*1 < 4n+6$

  Hence proved.

Performance Measurement

Performance Measurement

- **Performance measurement** is generally defined as regular **measurement** of outcomes and results, which generates reliable data on the effectiveness and efficiency of algorithm/programs.
- Performance measurement is concerned with obtaining the **space** and **time requirements** of a particular algorithm.
- These quantities depend on the **compiler** and **options** used as well as on the computer on which the algorithm is run.

Performance Measurement

```
Algorithm TimeSearch()
{
    for $j := 1$ to 1000 do  $a[j] := j$;
    for $j := 1$ to 10 do
    {
        $n[j] := 10 * (j - 1)$; $n[j + 10] := 100 * j$;
    }
    for $j := 1$ to 20 do
    {
        $h :=$ GetTime();
        $k :=$ SeqSearch($a, 0, n[j]$);
        $h1 :=$ GetTime();
        $t := h1 - h$;
        write ($n[j], t$);
    }
}
```

```
Algorithm SeqSearch($a, x, n$)
// Search for $x$ in $a[1 : n]$. $a[0]$ is used as additional space.
{
    $i := n$; $a[0] := x$;
    while ($a[i] \neq x$) do $i := i - 1$;
    return $i$;
}
```

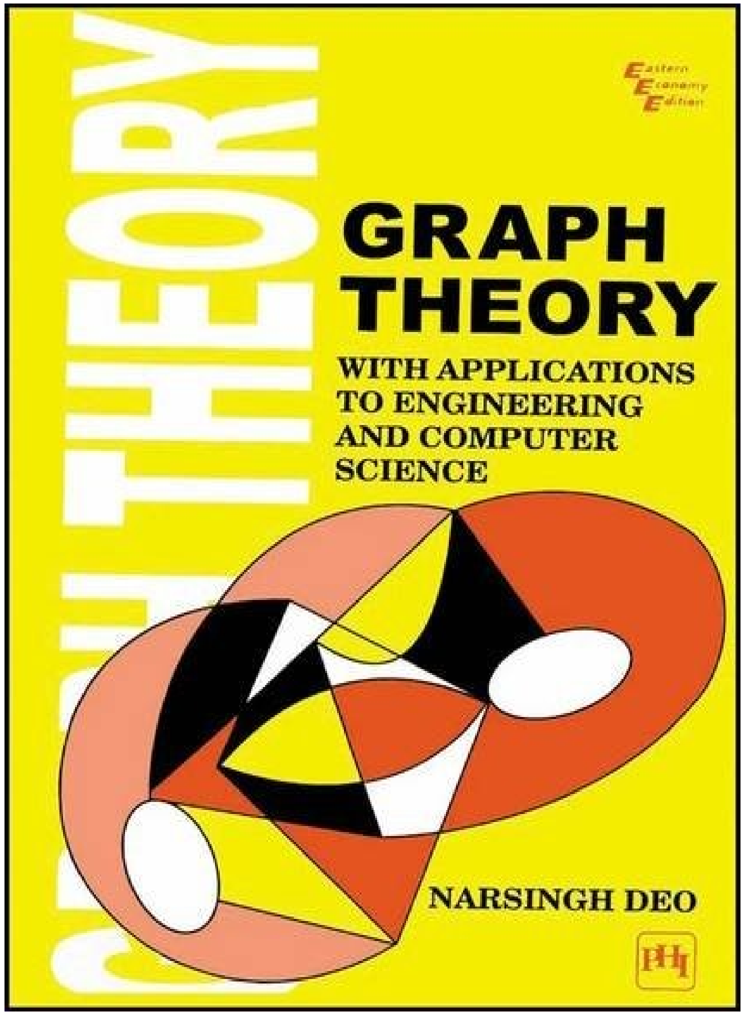
Input Size and Problem Instance

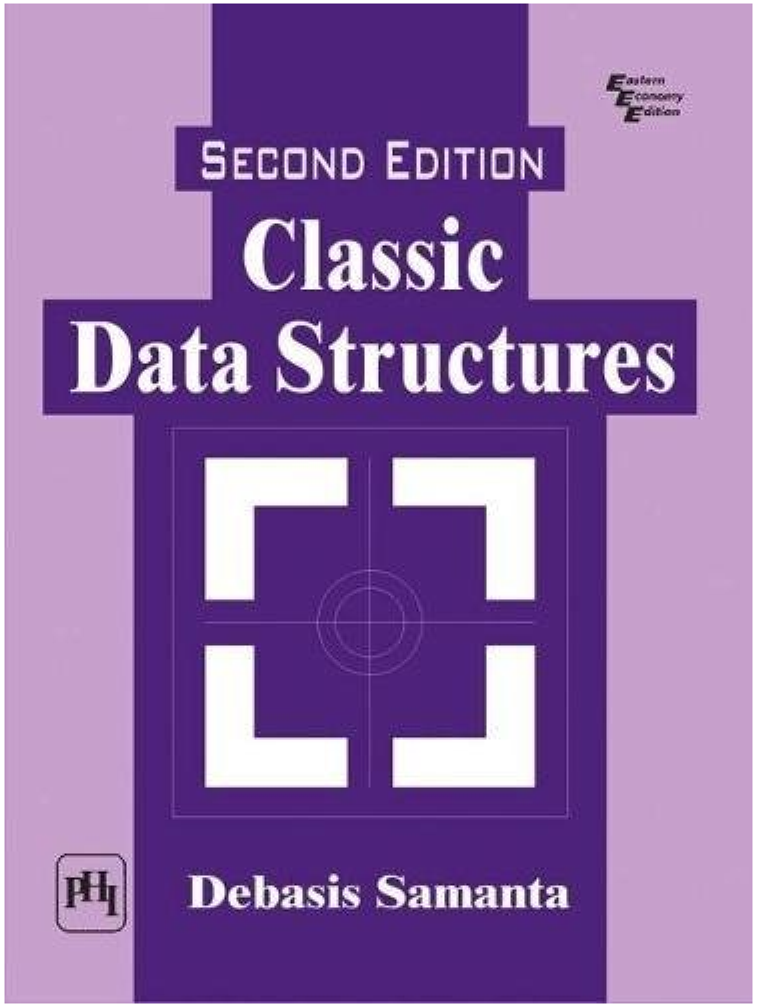
- The **size** of the **instance** of a **problem** is the **size** of the representation of the **input**.
- The input size, as a quantity, is not a precise measure of the input, and its interpretation is subject to the problem for which the algorithm is, or is to be, designed.

Some of the commonly used measures of input size are the following:

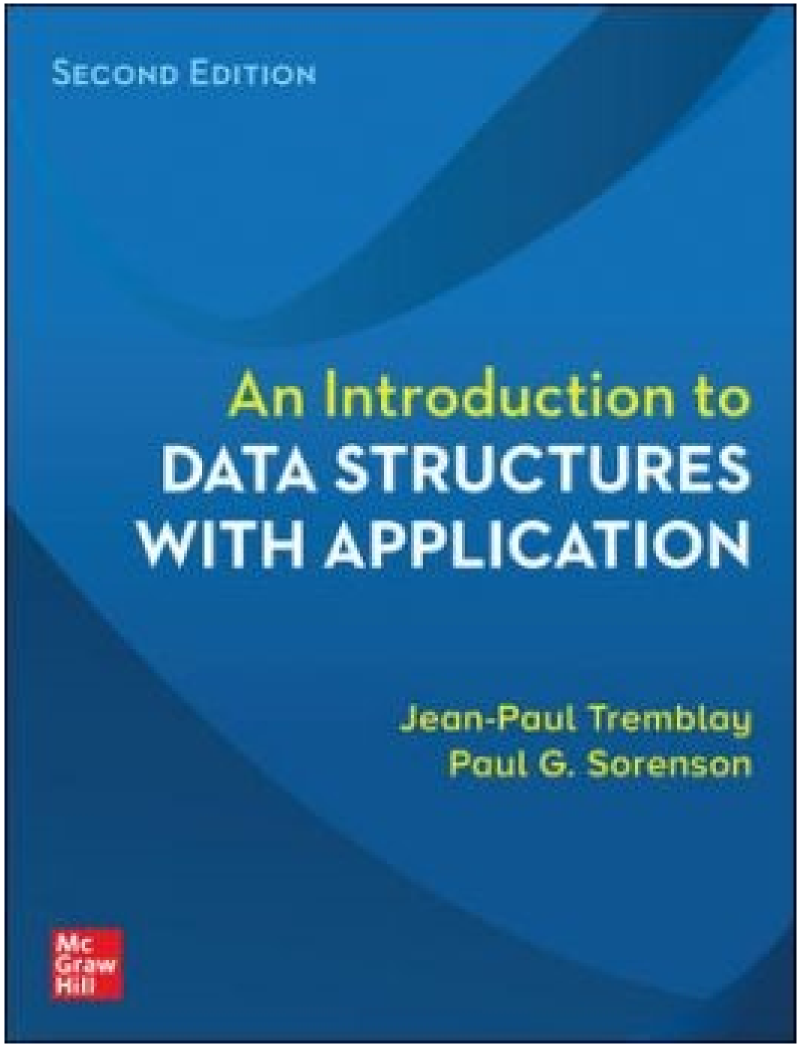
- In **sorting and searching problems**, we use the number of entries in the **array or list** as the input size.
- In **graph algorithms**, the input size usually refers to the number of vertices or edges in the graph, or both.

Reference for Graph Theory & Graph algorithm





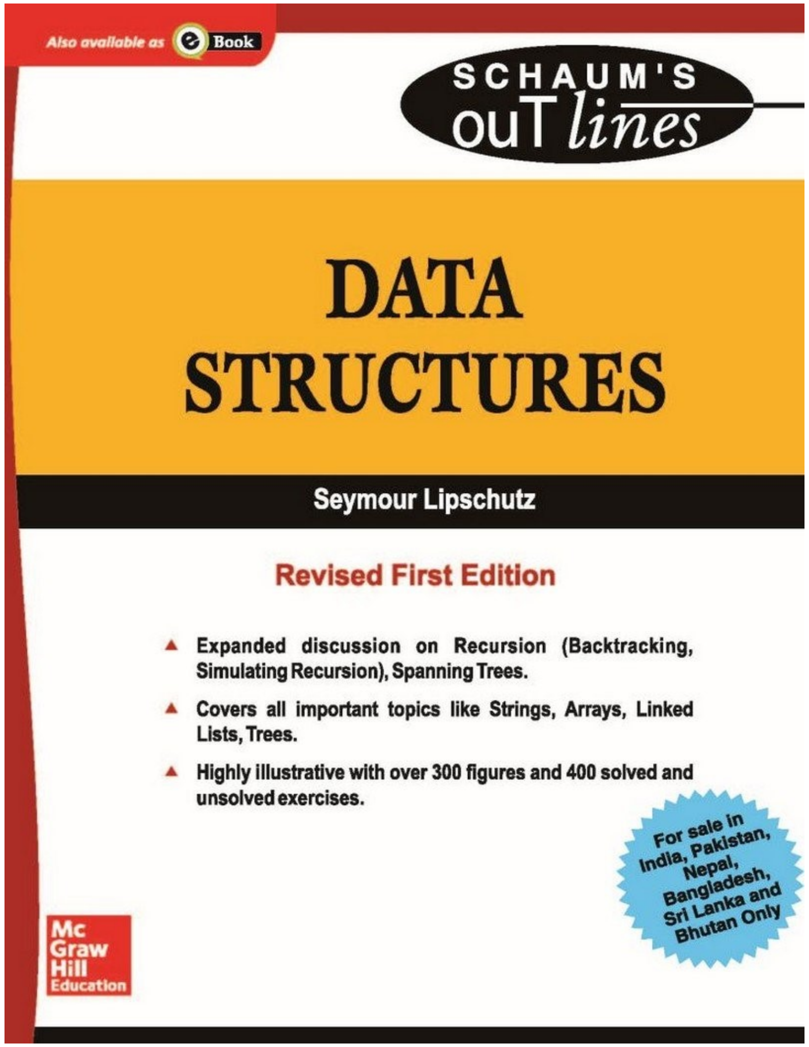
Reference for Graph Theory & Graph algorithm


SECOND EDITION

An Introduction to DATA STRUCTURES WITH APPLICATION

Jean-Paul Tremblay
Paul G. Sorenson

Mc Graw Hill


Also available as e Book

SCHAUM'S ouTlines

DATA STRUCTURES

Seymour Lipschutz

Revised First Edition

- Expanded discussion on Recursion (Backtracking, Simulating Recursion), Spanning Trees.
- Covers all important topics like Strings, Arrays, Linked Lists, Trees.
- Highly illustrative with over 300 figures and 400 solved and unsolved exercises.

For sale in India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan Only

Mc Graw Hill Education

Complexity of some Graph algorithm

A **graph** (sometimes called *undirected graph* for distinguishing from a **directed graph**, or *simple graph* for distinguishing from a multigraph) is a pair $G = (V, E)$, where V is a set whose elements are called *vertices* (singular: vertex), and E is a set of paired vertices, whose elements are called *edges* (sometimes *links* or *lines*).

	Average	Worst
Dijkstra's algorithm	$O(\lvert E\rvert \log \lvert V\rvert)$	$O(\lvert V\rvert^2)$
A* search algorithm	$O(\lvert E\rvert)$	$O(b^d)$
Prim's algorithm	$O(\lvert E\rvert \log \lvert V\rvert)$	$O(\lvert V\rvert^2)$
Bellman–Ford algorithm	$O(\lvert E\rvert \cdot \lvert V\rvert)$	$O(\lvert E\rvert \cdot \lvert V\rvert)$
Floyd-Warshall algorithm	$O(\lvert V\rvert^3)$	$O(\lvert V\rvert^3)$
Topological sort	$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert V\rvert + \lvert E\rvert)$

Input Size and Problem Instance

Some of the commonly used measures of **input size** are the following:

- In **sorting and searching problems**, we use the number of entries in the array or list as the input size.
- In **graph algorithms**, the input size usually refers to the number of vertices or edges in the graph, or both.
- In **computational geometry**, the size of input to an algorithm is usually expressed in terms of the number of points, vertices, edges, line segments, polygons, etc.
- In **matrix operations**, the input size is commonly taken to be the dimensions of the input matrices.
- In **number theory algorithm**s and **cryptography**, the number of bits in the input is usually chosen to denote its length. The number of **words** used to represent a single number may also be chosen as well, as each word consists of a **fixed number of bits**.

Comparison of two algorithms for computing the sum

Algorithm FIRST
Input: A positive integer n and an array $A[1..n]$ with $A[j] = j, 1 \leq j \leq n$.
Output: $\sum_{j=1}^{n} A[j]$.

```
1. $sum \leftarrow 0$
2. for $j \leftarrow 1$ to $n$
3.     $sum \leftarrow sum + A[j]$
4. end for
5. return $sum$
```

Algorithm SECOND
Input: A positive integer n.
Output: $\sum_{j=1}^{n} j$.

```
1. $sum \leftarrow 0$
2. for $j \leftarrow 1$ to $n$
3.     $sum \leftarrow sum + j$
4. end for
5. return $sum$
```

Comparison of two algorithms for computing the sum

Now we will compare two algorithms for computing the sum $\sum_{j=1}^{n} j$. In the first algorithm, which we will call FIRST, the input is an array $A[1..n]$ with $A[j] = j$, for each $j, 1 \leq j \leq n$. The input to the second algorithm, call it SECOND, is just the number n. These two algorithms are shown as Algorithms FIRST and Algorithm SECOND.

Obviously, both algorithms run in time $\Theta(n)$. Clearly, the time complexity of Algorithm FIRST is $\Theta(n)$. Algorithm SECOND is designed to solve a *number problem* and, as we have stated before, its input size is measured in terms of the number of bits in the binary representation of the integer n. Its input consists of $k = \lceil \log (n+1) \rceil$ bits. It follows that the time complexity of Algorithm SECOND is $\Theta(n) = \Theta(2^k)$. In other words, it is considered to be an *exponential* time algorithm. Notice that the number of elementary operations performed by both algorithms is the same.

Example: Time complexity

Algorithm BRUTE-FORCE PRIMALITYTEST

Input: A positive integer $n \geq 2$.

Output: *true* if n is prime and *false* otherwise.

```
1. $s \leftarrow \lfloor \sqrt{n} \rfloor$
2. for $j \leftarrow 2$ to $s$
3.     if $j$ divides $n$ then return false
4. end for
5. return true
```

Consider the brute-force algorithm for primality testing given in Example . Its time complexity was shown to be $O(\sqrt{n})$. Since this is a number problem, the time complexity of the algorithm is measured in terms of the number of bits in the binary representation of n. Since n can be represented using $k = \lceil \log(n+1) \rceil$ bits, the time complexity can be rewritten as $O(\sqrt{n}) = O(2^{k/2})$. Consequently, Algorithm BRUTE-FORCE PRIMALITYTEST is in fact an exponential algorithm.

Plan the experiment: To estimate run time of a Algorithm

1. What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.
2. For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.
3. Are we measuring worst-case or average performance? Suitable test data need to be generated.
4. What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict run times?
5. In case the times are to be used to predict run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola can be used

Shortcomings of asymptotic analysis

- let algorithm ***A*** be asymptotically better than algorithm ***B*.** Here are some common issues with algorithms that have better asymptotic behavior:

Implementation complexity

- Algorithms with better complexity are often (much) more complicated. This can increase coding time and the constants.

Small input sizes

- Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing *B* to outperform *A*.

Worst case versus average performance

- If *A* has better worst case performance than *B*, but the average performance of *B* given the expected input is better, then *B* could be a better choice than *A*. Conversely, if the worst case performance of *B* is unacceptable (say for life-threatening or mission-critical reasons), *A* must still be used.

Testing an algorithm

[1] **Is it efficient?** "Does the algorithm do what it is supposed to do?"

- If algorithm produces the correct output then most likely the algorithm is efficient.
- This can be done by some test cases that consist of several input samples and check if we get the predicted output.

[2] Is it time efficient?

- If the algorithm work and produces correct result with in a acceptable time.

[3] Is it reliable?

- Reliability of a algorithm is the ability to run fault-free for a certain period of time in a given environment.
- If your algorithm crashes when overwhelmed with a lot of input data at the same time, then you better modify it or think about improving it as fast as you can.

Generating Test Data

- Generating a data set that results in the worst-case performance of an algorithm:
- Generating a data set that results in the average-case performance of an algorithm:
- Whether we are estimating worst-case or average-time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment.

Thanks for Your Attention!

Exercises

Problems

1. Suppose a certain algorithm has been empirically shown to sort 100,000 integers or 100,000 floating-point numbers in two seconds. Would we expect the same algorithm to sort 100,000 strings in two seconds? Explain.
2. Suppose you need to search a given ordered array of *n* integers for a specific value. The entries are sorted in non-decreasing order.
   - Give a simple algorithm that has a worst-case complexity of $\Theta(n)$
   - We can do better by using the "binary search" strategy. Explain what it is and give an algorithm based on it. What is the worst-case complexity of this algorithm?
3. Explain the meaning of asymptotic bounds. Explain the importance of asymptotic notation. Determine if an assertion involving asymptotic bounds is valid. Ex: Is $3n^2+3n = O(n)$?
4. Compare the growth rates of two expressions, $2^{\{\lg n\}}$ and $(\lg n)^2$.

Problems

- Explain why the statement, “The running time of algorithm A is at least $O(n^2)$” is meaningless.

Problems

Fill in the blanks with either *true* or *false*:

$f(n)$	$g(n)$	$f = O(g)$	$f = \Omega(g)$	$f = \Theta(g)$
$2n^3 + 3n$	$100n^2 + 2n + 100$			
$50n + \log n$	$10n + \log\log n$			
$50n \log n$	$10n \log\log n$			
$\log n$	$\log^2 n$			
$n!$	5^n			

Express the following functions in terms of the Θ-notation.

(a) $2n + 3\log^{100} n$.
(b) $7n^3 + 1000n \log n + 3n$.
(c) $3n^{1.5} + (\sqrt{n})^3 \log n$.
(d) $2^n + 100^n + n!$.

Express the following functions in terms of the Θ-notation.

(a) $18n^3 + \log n^8$.
(b) $(n^3 + n)/(n + 5)$.
(c) $\log^2 n + \sqrt{n} + \log\log n$.
(d) $n!/2^n + n^{n/2}$.

Problems

Find two monotonically increasing functions $f(n)$ and $g(n)$ such that $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$.

Is $x = O(x \sin x)$? Use the definition of the O-notation to prove your answer.

Prove that $\sum_{j=1}^{n} j^k$ is $O(n^{k+1})$ and $\Omega(n^{k+1})$, where k is a positive integer. Conclude that it is $\Theta(n^{k+1})$.

Let $f(n) = \{1/n + 1/n^2 + 1/n^3 + \ldots\}$. Express $f(n)$ in terms of the Θ-notation. (Hint: Find a recursive definition of $f(n)$).

Show that $n^{100} = O(2^n)$, but $2^n \neq O(n^{100})$.

Show that 2^n is not $\Theta(3^n)$.

Problems

Is $n! = \Theta(n^n)$? Prove your answer.

Is $2^{n^2} = \Theta(2^{n^3})$? Prove your answer.

Carefully explain the difference between $O(1)$ and $\Theta(1)$.

Is the function $\lfloor \log n \rfloor!$ $O(n)$, $\Omega(n)$, $\Theta(n)$? Prove your answer.

Can we use the $\prec$ relation described in Sec. 1.8.6 to compare the order of growth of n^2 and $100n^2$? Explain.

Use the $\prec$ relation to order the following functions by growth rate: $n^{1/100}$, $\sqrt{n}$, $\log n^{100}$, $n \log n$, 5, $\log \log n$, $\log^2 n$, $(\sqrt{n})^n$, $(1/2)^n$, 2^{n^2}, $n!$.

Consider the following problem. Given an array $A[1..n]$ of integers, test each element a in A to see whether it is even or odd. If a is even, then leave it; otherwise multiply it by 2.

(a) Which one of the O and Θ notations is more appropriate to measure the number of multiplications? Explain.

(b) Which one of the O and Θ notations is more appropriate to measure the number of element tests? Explain.

Problems

Consider Algorithm COUNT6 whose input is a positive integer n.

```
Algorithm       COUNT6
   1.  comment: Exercise 1.33
   2.  $count \leftarrow 0$
   3.  for $i \leftarrow 1$ to $n$
   4.      $j \leftarrow \lfloor n/3 \rfloor$
   5.      while $j \geq 1$
   6.          for $k \leftarrow 1$ to $i$
   7.              $count \leftarrow count + 1$
   8.          end for
   9.          if $j$ is even then $j \leftarrow 0$ else $j \leftarrow \lfloor j/3 \rfloor$
  10.      end while
  11.  end for
```

(a) What is the maximum number of times Step 7 is executed when n is a power of 2?

(b) What is the maximum number of times Step 7 is executed when n is a power of 3?

(c) What is the time complexity of the algorithm expressed in the O-notation?

(d) What is the time complexity of the algorithm expressed in the Ω-notation?

(e) Which one of the O and Θ notations is more appropriate to express the time complexity of the algorithm?

Problems

Write an algorithm to find the maximum and minimum of a sequence of n integers stored in array $A[1..n]$ such that its time complexity is

(a) $O(n)$.
(b) $\Omega(n \log n)$.

Let $A[1..n]$ be an array of integers, where $n > 2$. Give an $O(1)$ time algorithm to find an element in A that is neither the maximum nor the minimum.

Consider the element uniqueness problem: Given a set of integers, determine whether two of them are equal. Give an *efficient* algorithm to solve this problem. Assume that the integers are stored in array $A[1..n]$. What is the time complexity of your algorithm?

Questions

1. How does one calculate the running time of an algorithm?
2. How can we compare two different algorithms?
3. How do we know if an algorithm is `optimal'?
4. When we say an algorithm is O(1) what do we mean?
5. You are given list of numbers, obtained by rotating a sorted list an unknown number of times. Write a function to determine the minimum number of times the original sorted list was rotated to obtain the given list. Your function should have the worst-case complexity of O(log N), where N is the length of the list. You can assume that all the numbers in the list are unique.

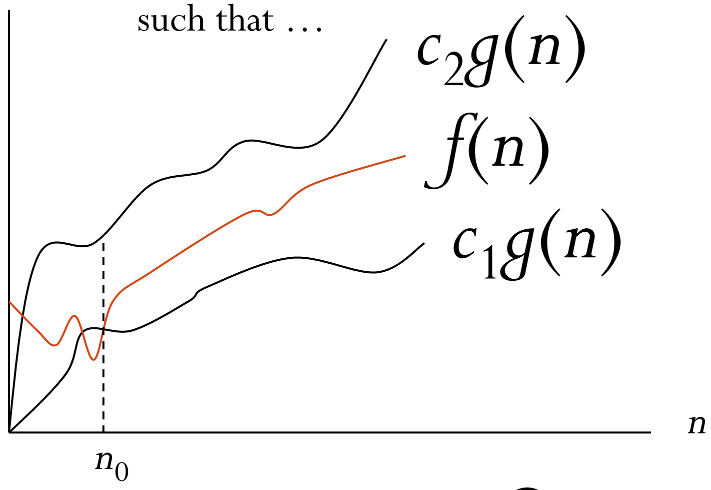
   Example: The list [5, 6, 9, 0, 2, 3, 4] was obtained by rotating the sorted list [0, 2, 3, 4, 5, 6, 9] 3 times.

Questions

6. Suppose that the actual time taken to execute an $O(N^2)$ sorting algorithm on a very fast machine is $\mathbf{2.2x10^{-8}N^2}$ seconds; now suppose that the actual time taken to execute an $O(N \log_2 N)$ sorting algorithm on a very slow machine is $\mathbf{7.2x10^{-4}(N \log_2 N)}$ seconds (here the slow machine is about 10,000 slower than the fast one). For what size arrays will the slower machine running the faster algorithm sort arrays faster than the faster machine running the slower algorithm?
7. Suppose a certain algorithm has been empirically shown to sort 100,000 integers or 100,000 floating-point numbers in two seconds. Would we expect the same algorithm to sort 100,000 strings in two seconds? Explain

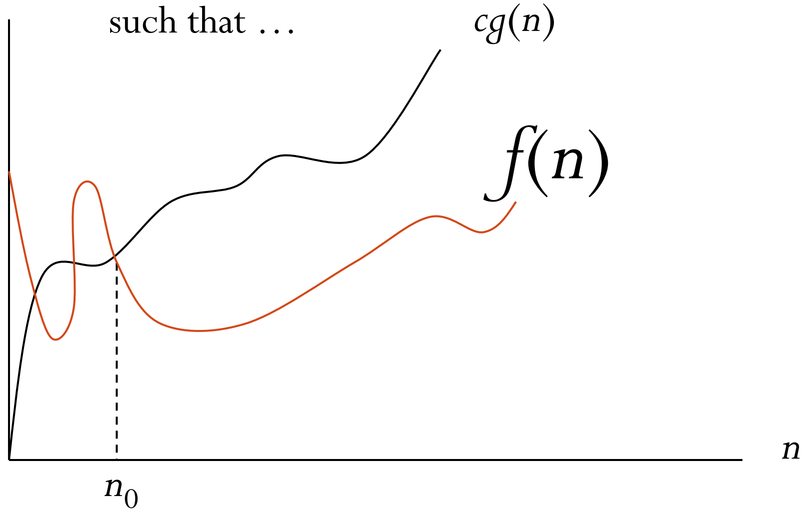
Appendix

There exist positive constants c_1 and c_2
such that there is a positive constant n_0
such that …



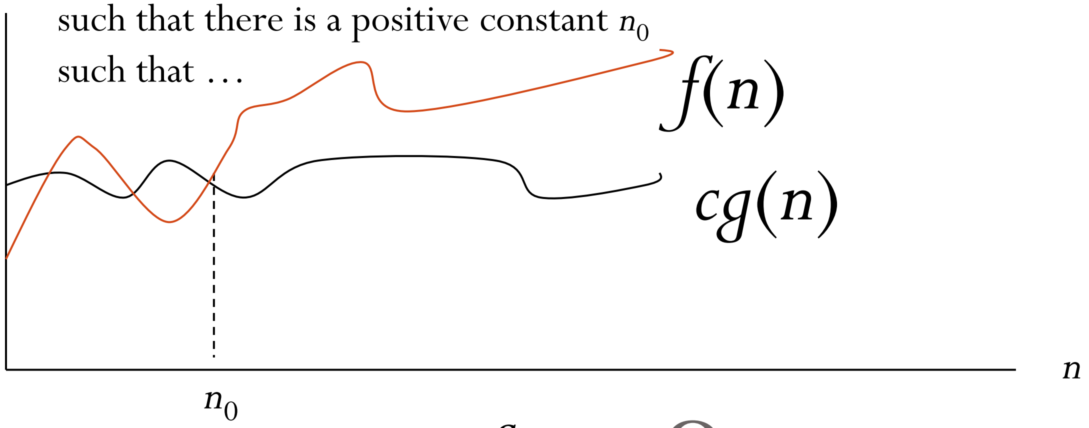
$$f(n) = \Theta(g(n))$$

There exist positive constants c
such that there is a positive constant n_0
such that …



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

There exist positive constants c
such that there is a positive constant n_0
such that …



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

Asymptotic Notation

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Properties

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \;\&\; g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties

- **Symmetry**

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- **Complementarity**

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ iff } g(n) = \omega((f(n))$$