Basics of Algorithm Analysis

Rangaballav Pradhan

Asst. Professor, Dept. of CSE, ITER,

SOA Deemed to be University

Analysis of Algorithms

- Analysis: Measures the efficiency of an algorithm
- Gives a theoretical estimation of the resource required by the algorithm to solve a specific computational problem.
- To analyze an algorithm means:
  - developing a formula for predicting how fast an algorithm is, based on the size of the input (time complexity), and/or
  - developing a formula for predicting how much memory an algorithm requires, based on the size of the input (space complexity)
- Usually time is our biggest concern, most algorithms require a fixed amount of space
- Associating the efficient algorithms with specific type of problems

Input size

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be the parameter that most influences the actual time/space required
- Algorithms are analyzed to check the performance for a sufficiently large input instance

Characteristics operation

- In computing time complexity, one good approach is to count the characteristic operations
  - What a “characteristic operation” is depends on the particular problem
  - If searching, it might be comparing two values
  - If sorting an array, it might be:
    - comparing two values
    - swapping the contents of two array locations
    - both of the above
  - Sometimes we just look at how many times the *innermost loop* is executed

Exact vs Asymptotic Analysis

- It is sometimes possible, *in assembly language*, to compute *exact* time and space requirements
  - We know exactly how many bytes and how many cycles each machine instruction takes
  - For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required
- However, often the exact sequence of steps cannot be known in advance
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)
- We go for asymptotic analysis when the input instance is large
- $f(n) = n^2 + p.n + q$ can be represented as "asymptotically equivalent to n^2, as $n \rightarrow \infty$"

Average, best, and worst cases

- **Worst-case**: The maximum number of steps taken by the algorithm on any input instance.
- **Best-case:** The minimum number of steps taken by the algorithm on any input instance.
- **Average case**: An average number of steps taken by the algorithm on any input instance.
- Usually we would like to find the *average* time to analyze an algorithm
- However, sometimes the “average” isn’t well defined
- Often we have to be satisfied with finding the *worst* (longest) time required
- The *best* (fastest) case is seldom of interest

Worst case time

- Worst-case running time is of primary concern
- It is the longest running time for any input of size n.
- It gives us an upper bound on the running time for any input.
- It guarantees that the algorithm will never take any longer.
- Some algorithms primarily incur the worst case time. For example, searching for an absent information.
- The average case is often roughly as bad as the worst case.
- It is measured as the order of growth of the function.

Constant time

- *Constant time* operation means there is some constant k such that this operation always takes k nanoseconds
- A Java statement takes constant time if:
  - It does not include a loop
  - It does not include calling a method whose time is unknown or is not a constant
- If a statement involves a choice (if or switch) among operations, each of which takes constant time, we consider the statement to take constant time
- Even if the running time is not independent of the problem size, rather an upper bound for the running time is bounded independently of the problem size, still it can be called as constant time algorithms.

Linear time

- We may not be able to predict to the nanosecond how long a Java program will take, but do know *some* things about timing:

```
for (i = 1, j = 1; i < = n; i++) {
    j = j * i;
}
```

  - This loop takes time k*n + c, for some constants k and c

    k : How long it takes to go through the loop once
    (the time for j = j * i, plus loop overhead)

    n : The number of times through the loop
    (we can use this as the "size" of the problem)

    c : The time it takes to initialize the loop

  - The total time k*n + c is *linear in n*
  - Finding the minimal value in an unordered array is a linear time operation, taking O(n) time.

Frequency/Step Count method

- Used to determine the time complexity of an algorithm in terms of step count.
- We first determine the cost per execution of each statement (c_i) and the total number of times (i.e., frequency) each statement is being executed ($f_i(n)$) which is a function of input size.
- Combining these two quantities gives us the total contribution of each statement to the total step count ($c_i \times f_i(n)$).
- We then add the contributions of all statements to obtain the step count/time complexity for the entire algorithm as,

$$\sum_i (c_i \times f_i(n))$$

Analyzing Insertion Sort: Frequency Count

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1)$$

$$+ c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)\,.$$

Analyzing Insertion Sort: Frequency Count

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

- Best case: When the input array is already sorted

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

Space Complexity

- Total memory space needed by the algorithm to complete its execution.
- Expressed as a function of input size (in terms of storage).
- Includes the space used for the input as well as any Auxiliary space used.
- Auxiliary Space includes any extra space or temporary space used by an algorithm in addition to that of the input.
- Space Complexity = Fixed Space + Variable Space
- **Fixed Space:** Independent of the characteristics of the inputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- **Variable Space:** depend on the input instance
  - number, size, values of inputs
  - recursive stack space, formal parameters, return address

Space Complexity

- **Instruction space:** The space needed to store the compiled version of the program instructions. The amount of instructions space that is needed depends on factors such as:
  - The compiler used to compile the program into machine code.
  - The compiler options in effect at the time of compilation
  - The target computer.
- **Data space:** The pace needed to store all the constants and variable values. Data space has two components:
  - Space needed by constants and simple variables in program.
  - Space needed by dynamically allocated objects such as arrays and class instances.
- **Environment stack space:** The environment stack is used to save information needed to resume execution of partially completed functions.

Space Complexity

- #Sum Of N Natural Number

sumN(*n*)
sum = 0
for *i* = *n* down to 1
    sum = *sum* + *i*
return *sum*

- Input value *n* takes fixed space of O(1) and auxiliary space is also O(1) because *i* and *sum* are also fixed.
- Hence total space complexity is O(1).

Space Complexity

factorial(*n*)

1. if $n = 0$ then
2. return 1
3. return n * factorial(n-1)

- Here each call add a level to the stack
- Each of these calls is added to call stack and takes up actual memory.
- So it takes O(n) space.

factorial(*n*)

1. *fact* = 1
2. for $i = 1$ to n do
3. *fact* = factor(*fact*, *i*)
4. return *fact*

factor(*fact, i*)

p = *fact***i*
return *p*

- There will be roughly O(n) calls to factor(). However, those calls do not exist simultaneously on the call stack,
- so we only need O(1) space.

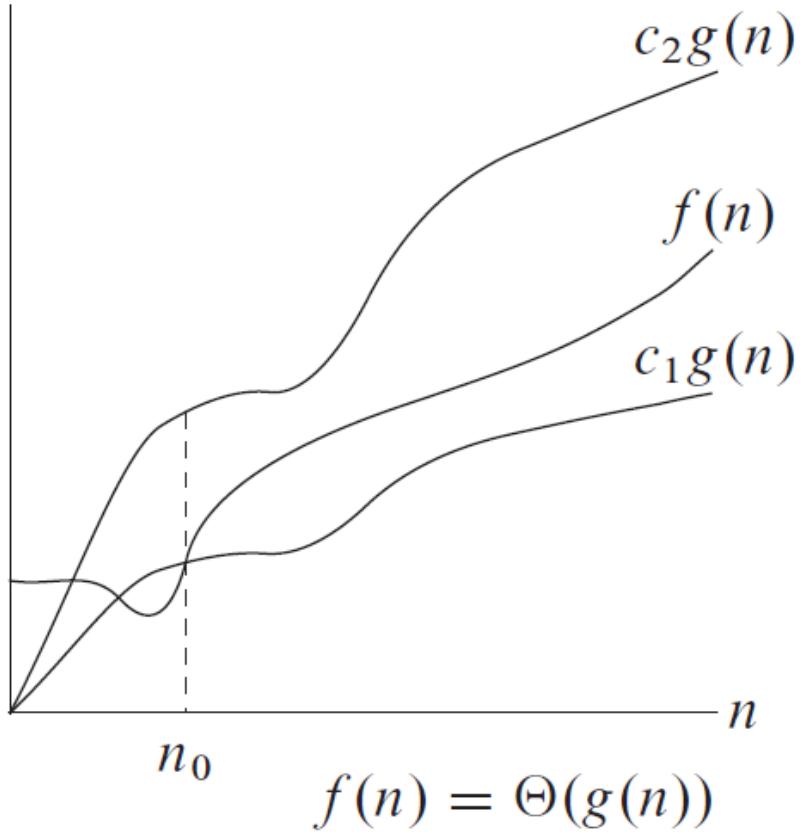
Asymptotic Notations

1. θ – Big theta
2. O – Big Oh
3. Ω – Big omega
4. o – Little Oh
5. ω – Little omega

Θ – Big theta

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.^1$$

- f(n) ∈ $\theta(g\ (n))$ be asymptotically nonnegative

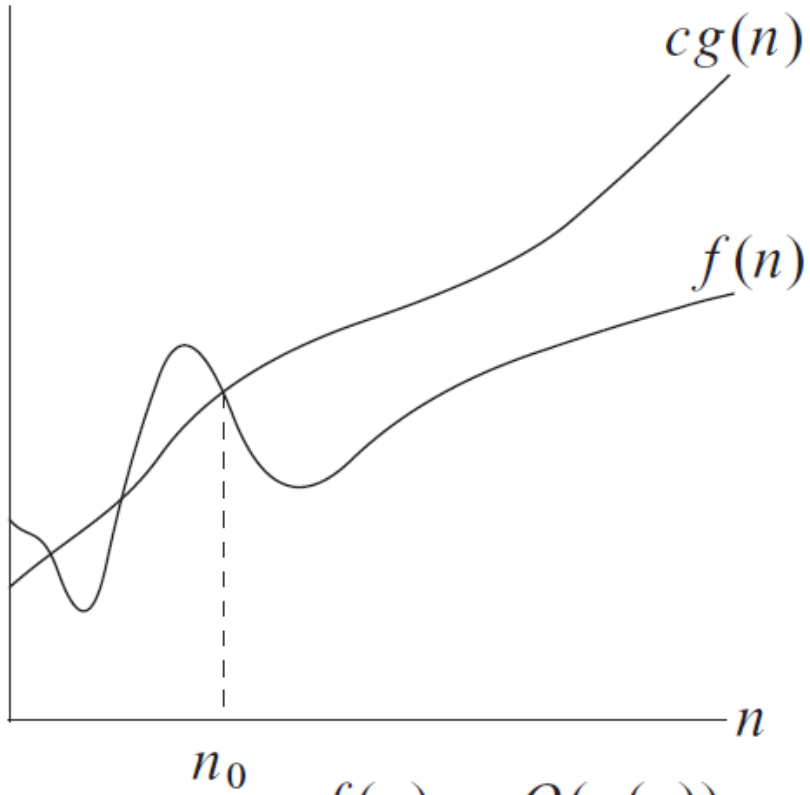


θ - Big theta

1. $3n + 10^{10}$
   $3n \leq 3n + 10^{10} \leq 4n, \forall n \geq 10^{10}$
   $\Rightarrow 3n + 10^{10} = \Theta(n)$
   Note that the first inequality captures $3n + 10^{10} = \Omega(n)$ and the later one captures $3n + 10^{10} = O(n)$.
2. $10n^2 + 4n + 2 = \Theta(n^2)$
   $10n^2 \leq 10n^2 + 4n + 2 \leq 20n^2, \forall n \geq 1, c_1 = 10, c_2 = 20$
3. $6(2^n) + n^2 = \Theta(2^n)$
   $6(2^n) \leq 6(2^n) + n^2 \leq 12(2^n), \forall n \geq 1, c_1 = 6, c_2 = 12$
4. $2n^2 + n \log n + 1 = \Theta(n^2)$
   $2n^2 \leq 2n^2 + n \log_2 n + 1 \leq 5.n^2, \forall n \geq 2, c_1 = 2, c_2 = 5$
5. $n\sqrt{n} + n \log_2(n) + 2 = \Theta(n\sqrt{n})$
   $n\sqrt{n} \leq n\sqrt{n} + n \log_2(n) + 2 \leq 5.n\sqrt{n}, \ \forall n \geq 2, c_1 = 1, c_2 = 5$

O – Big Oh

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



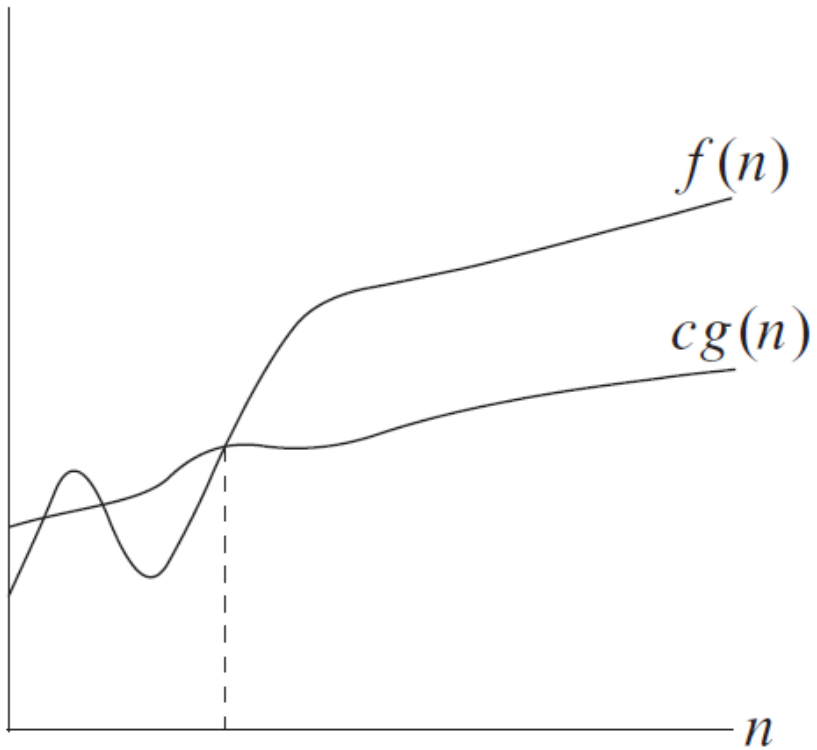
$f(n) = O(g(n))$

O – Big Oh

1. $3n + 2$
   $3n + 2 \leq 4n$, $c = 4$ $\forall n \geq 2$
   $\Rightarrow 3n + 2 = O(n)$
   Note that $3n + 2$ is $O(n^2), O(n^3), O(2^n), O(10^n)$ as per the definition. i.e., it captures all upper bounds. For the above example, $O(n)$ is a tight upper bound whereas the rest are loose upper bounds. Is there any notation which captures all the loose upper bounds?
2. $100n + 6 = O(n)$
   $100n + 6 \leq 101.n$, $c = 101$ $\forall n \geq 6$. One can also write $100n + 6 = O(n^3)$.
3. $10n^2 + 4n + 2 = O(n^2)$
   $10n^2 + 4n + 2 \leq 11.n^2$, $c = 11$ $\forall n \geq 5$
4. $6.2^n + n^2 = O(2^n)$
   $6.2^n + n^2 \leq 7.2^n$, $c = 7$ $\forall n \geq 7$

Ω – Big omega

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

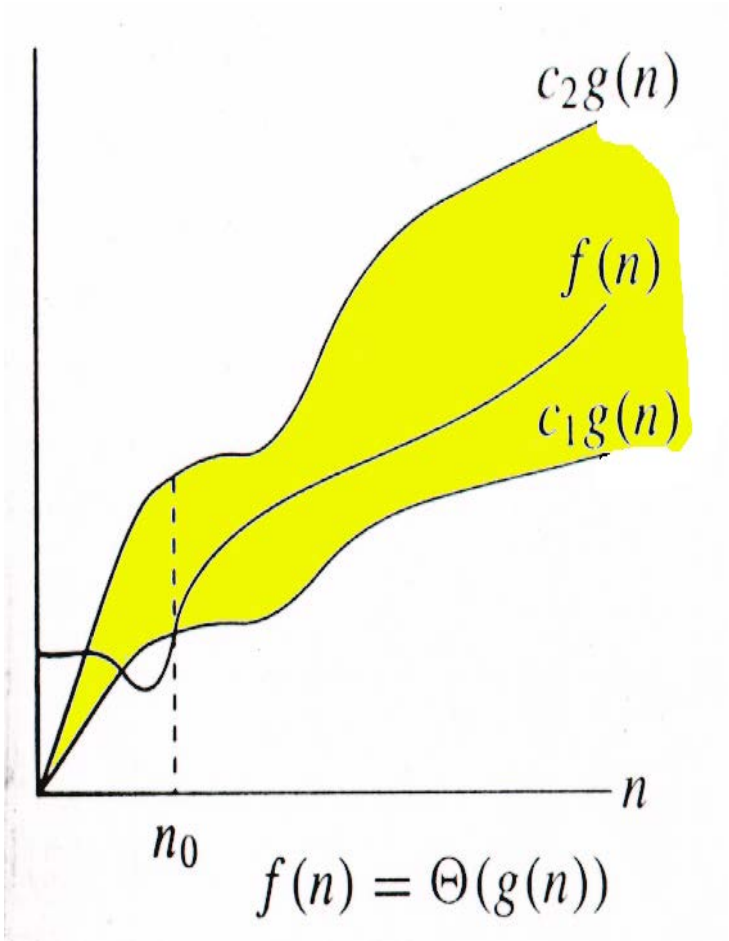


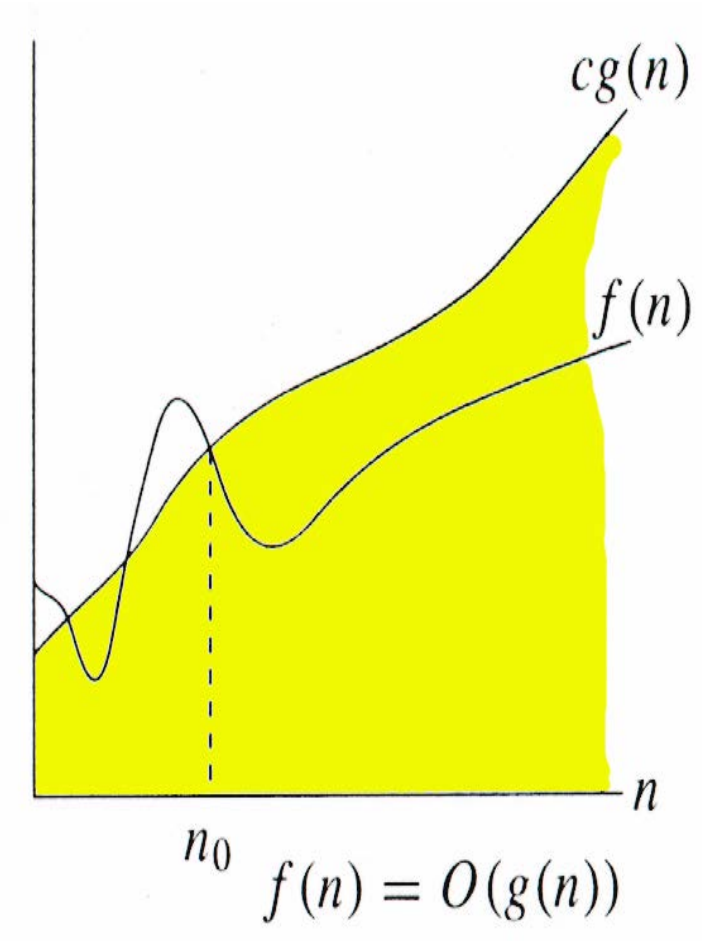
$f(n) = \Omega(g(n))$

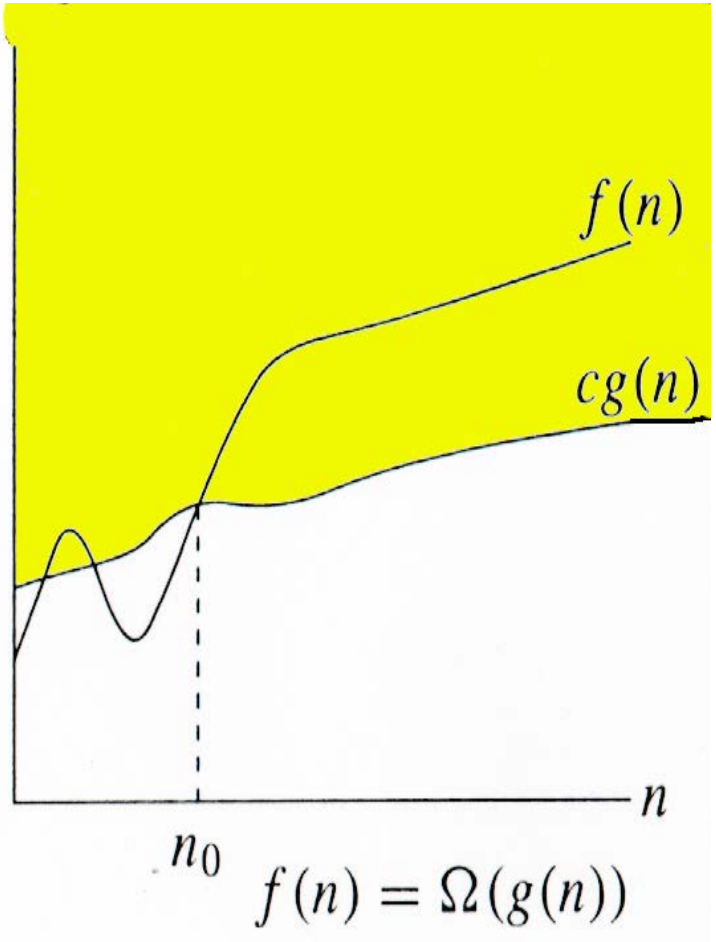
Ω - Big omega

1. $3n + 2$
   $3n + 2 \geq n, \forall n \geq 1$
   $\Rightarrow 3n + 2 = \Omega(n)$
2. $10n^2 + 4n + 2 = \Omega(n^2)$
   $10n^2 + 4n + 2 \geq n^2,\ c = 1\ \forall n \geq 1$
3. $n^3 + n + 5 = \Omega(n^3)$
   $n^3 + n + 5 \geq n^3, c = 1, \forall n \geq 0$
4. $2n^2 + n \log n + 1 = \Omega(n^2)$
   $2n^2 + n \log n + 1 \geq 2.n^2, c = 2, \forall n \geq 1$
5. $6.2^n + n^2 = \Omega(2^n) = \Omega(n^2) = \Omega(n) = \Omega(1)$
   $6.2^n + n^2 \geq 2^n,\ c = 1\ \forall n \geq 1$
   Of the above, $\Omega(2^n)$ is a tight lower bound, while all others are loose lower bounds.

Relation Between Θ, O, Ω







o – Little Oh

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

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

1. $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. Note that here n^2 is polynomially larger than $2n$ by $n^\epsilon, \epsilon = 1$.
2. $100n+6 = o(n^{1.2})$ Here $n^{1.2}$ is polynomially larger than $100n+6$ by $n^\epsilon, \epsilon = 0.2$. For any positive constant c, there exist n_0 such that $\forall n \geq n_0, 100n + 6 \leq c.n^{1.2}$
3. $10n^2 + 4n + 2 = o(n^3)$ Here n^3 is polynomially larger than $10n^2 + 4n + 2$ by $n^\epsilon, \epsilon = 1$
4. $6.2^n + n^2 = o(3^n)$ Note that 3^n is $1.5^n \times 2^n$. So for any $c > 0$, $2^n \leq c.3^n$. The value of c is insignificant as 1.5^n dominates any $c > 0$.
5. $3n + 3 = o(n^{1.00001})$ Here $n^{1.00001}$ is polynomially larger than $3n + 3$ by $n^\epsilon, \epsilon = 0.00001$
6. $n^3 + n + 5 = o(n^{3.1})$ Here $n^{3.1}$ is polynomially larger than $n^3 + n + 5$ by $n^\epsilon, \epsilon = 0.1$

ω - Little omega

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$$

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

1. $n^2 = \omega(n)$ but $n^2 \neq \omega(n^2)$.
2. $3n + 2 = \omega(\log(n))$
3. $10n^3 + 4n + 2 = \omega(n^2)$
4. $5n^6 + 7n + 9 = \omega(n^3)$
5. $2n^2 + n \log n + 1 = \omega(n^{1.9999999})$
6. $15 \times 3^n + n^2 = \omega(2^n) = \omega(n^2) = \omega(n) = \omega(1)$

Properties

- **Reflexivity**

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

- **Symmetry**

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

- **Transitivity**

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

Properties

Transpose symmetry:

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

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$f(n) = O(g(n)) \quad \text{is like} \quad a \leq b\ ,$$
$$f(n) = \Omega(g(n)) \quad \text{is like} \quad a \geq b\ ,$$
$$f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b\ ,$$
$$f(n) = o(g(n)) \quad \text{is like} \quad a < b\ ,$$
$$f(n) = \omega(g(n)) \quad \text{is like} \quad a > b\ .$$

Standard notations and common functions

- **Monotonicity:**
- $f(n)$ is
  - **monotonically increasing** if $m \leq n \Rightarrow f(m) \leq f(n)$.
  - **monotonically decreasing** if $m \leq n \Rightarrow f(m) \geq f(n)$.
  - **strictly increasing** if $m < n \Rightarrow f(m) < f(n)$.
  - **strictly decreasing** if $m < n \Rightarrow f(m) > f(n)$.

Standard notations and common functions

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 .$$

For any integer n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n ,$$

and for any real number $x \geq 0$ and integers $a, b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil ,$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor ,$$

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b} ,$$

$$\left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a - (b-1)}{b} .$$

Standard notations and common functions

- **Exponentials:**

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

- **Exponentials and polynomials**

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Standard notations and common functions

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

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

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

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

Lemma 1. *Let $f(n)$ and $g(n)$ be two asymptotic non-negative functions. Then, $max(f(n), g(n)) = \theta(f(n) + g(n))$*

Proof. Without loss of generality, assume $f(n) \leq g(n)$, $\Rightarrow max(f(n), g(n)) = g(n)$

Consider, $g(n) \leq max(f(n), g(n)) \leq g(n)$

$\Rightarrow g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2}g(n) + \frac{1}{2}g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

From what we assumed,we can write

$\Rightarrow \frac{1}{2}f(n) + \frac{1}{2}g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2}(f(n) + g(n)) \leq max(f(n), g(n)) \leq f(n) + g(n)$

By the definition of θ ,

$max(f(n), g(n)) = \theta(f(n) + g(n))$

Lemma 2. *For two asymptotic functions* $f(n)$ *and* $g(n)$, $O(f(n))+O(g(n)) = O(max(f(n), g(n)))$

Proof. Without loss of generality, assume $f(n) \leq g(n)$

$$\Rightarrow O(f(n)) + O(g(n)) = c_1 f(n) + c_2 g(n)$$

From what we assumed,we can write

$$O(f(n)) + O(g(n)) \leq c_1 g(n) + c_2 g(n)$$

$$\leq (c_1 + c_2) g(n)$$

$$\leq \text{c } g(n)$$

$$\leq \text{c } max(f(n), g(n))$$

By the definition of Big-Oh(O),

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

Observations

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \in \mathbb{R}^+$ then $f(n) = \theta(g(n))$
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c, c \in \mathbb{R}$ (c can be 0) then $f(n) = O(g(n))$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$ then $f(n) = O(g(n))$ and $g(n) \neq O(f(n))$
4. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c, c \in \mathbb{R}(c$ can be $\infty)$ then $f(n) = \Omega(g(n))$
5. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$ then $f(n) = \Omega(g(n))$ and $g(n) \neq \Omega(f(n))$