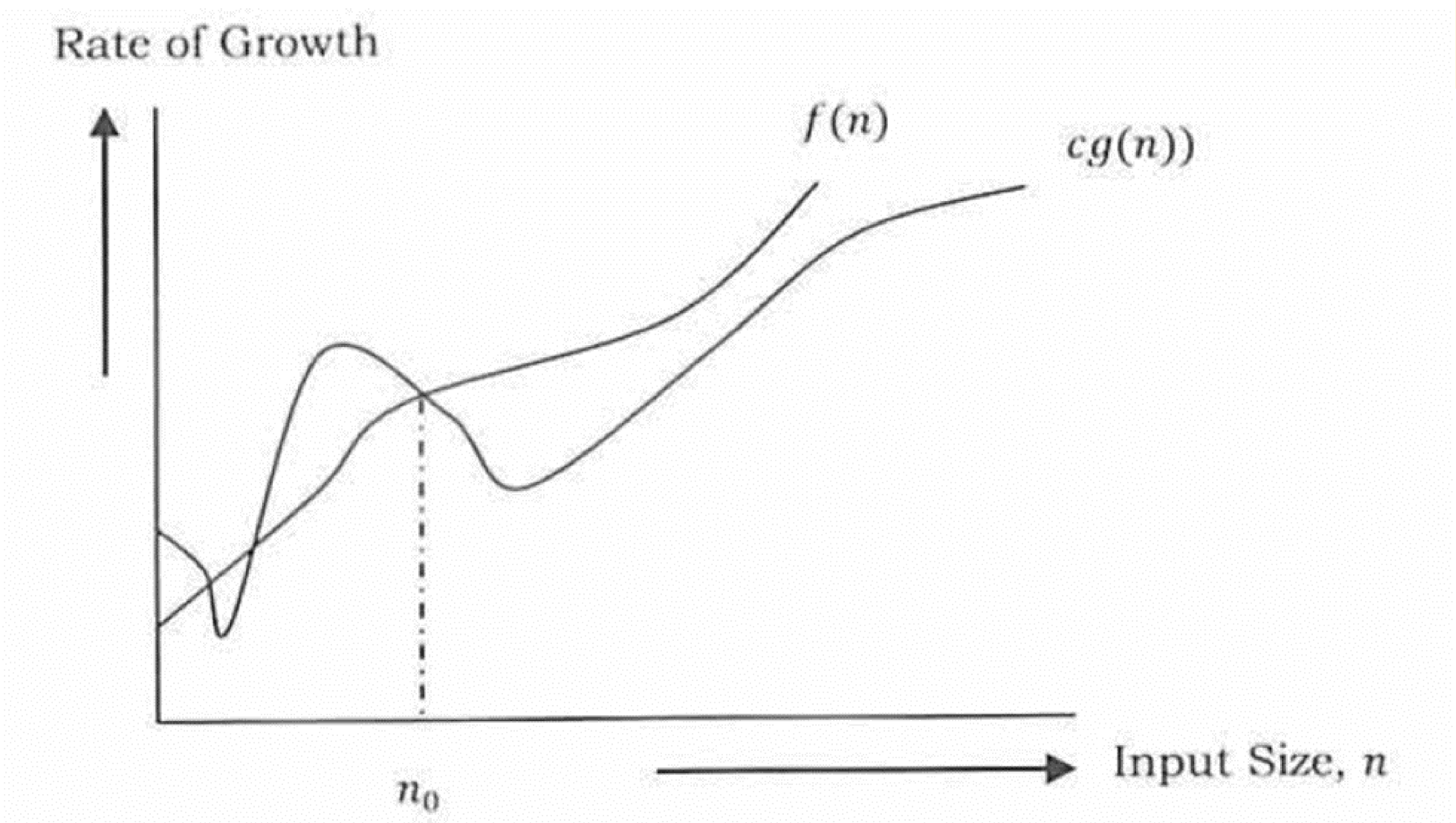
Omega-Ω Notation
Theta- Θ Notation

Data Structure, second stage, computer science department, college of science for women, University of Baghdad.

Omega-Ω Notation

- This notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$.
- That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.


Rate of Growth
f(n)
cg(n))
Input Size, n
n0

Omega-Ω Notation

- The Ω notation can be defined as Ω (g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all n ≥ n_0 }.
- g(n) is an asymptotic tight lower bound for f(n).
- Our objective is to give the largest rate of growth g(n) which is less than or equal to the given algorithm's rate of growth $f(n)$.
- Example -1 Find lower bound for $f(n) = 5n^2$.
- Solution: ∃ c, n_0 Such that: $0 \le cn^2 \le 5n^2$ =:> c= 1 and n_0= 1
- $5n^2 = \Omega(n^2)$ with c = 1 and n_0 = 1

Theta- Θ Notation

- This notation decides whether the upper and lower bounds of a given function (algorithm) are the same.
- The average running time of an algorithm is always between the lower bound and the upper bound.
- If the upper bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth.
- As an example, let us assume that $f(n)$= 10n + n is the expression.
- Then, its tight upper bound $g(n)$ is O(n), The rate of growth in the best case is $g(n)$= $O(n)$.

Theta- Θ Notation

$\Theta(n)$

- In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same.
- For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same.
- In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case).
- Now consider the definition of Θ notation. It is defined as $\Theta(g(n))$ = $\{f(n)$: there exist positive constants c_1, c_2 and n_0 such that 0 ≤ $c_1 g(n)$ ≤ $f(n)$ ≤ $c_2 g(n)$ for all $n \geq n_0\}$.

Theta- Θ Notation

- $g(n)$ is an asymptotic tight bound for $f(n)$.
- $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Guidelines for Asymptotic Analysis

- There are some general rules to help us determine the running time of an algorithm.
- 1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.
- # executes n times
- for i in range(0,n):
  - print 'Current Number:'. i #constant time

  Total time = a constant c x n = c n = O(n).

Guidelines for Asymptotic Analysis

- 2) Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.
- # outer loop executed n times
- for i in range(O,n): n cn^2
- # inner loop executes n times
  - for j in range(O,n): n
    - print 'i value %.d and j value %d' % (i,j) #constant time c
- Total time: = c x n x n = cn^2 = O(n^2).

Guidelines for Asymptotic Analysis

- 3) **Consecutive statements:** Add the time complexities of each statement.
- C_0 n = 100
- #executes n times
- n for i in range(O,n): $O(n)$
  - c print 'Current Number:', I #constant time $O = n + n^2$
- #outer loop executed n times $O = n^2$
- for i in range(O,n): n $O(n^2)$
  - # inner loop executes n times
  - for j in range(O,n): n
    - c print 'i value %d and j value o/od' % (i,j) #constant time
- Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

Guidelines for Asymptotic Analysis

- 4) **If-then-else statements** : Worst-case running time: the test, plus either the *then* part or the *else* part (whichever is the larger).
- if n == 1: #constant time C_0
  - print "Wrong Value"
  - print n
- else:
  - for i in range(0,n): #n times n
    - print 'Curren t Number:', I #constant time C_1
- Total time $= c_0 + c_1 * n = O(n)$.

Guidelines for Asymptotic Analysis

- 5**) Logarithmic complexity**: An algorithm is O($logn$) if it takes a constant time to cut the problem size by a fraction (usually by 1/2). As an example, let us consider the following program:
- def Logarithms(n):
  - i = l
  - while i <= n:
    - i= i * 2
    - print i
- Logarithms(100)
- If we observe carefully, the value of i is doubling every time.

Guidelines for Asymptotic Analysis

- Initially i = l, in next step i = 2, and in subsequent steps i = 4,8 and so on.
- Let us assume that the loop is executing some k times. At k^{th} step 2^k = n and we come out of loop. Taking logarithm on both sides, gives
- log(2^k) = $logn$
- $klog^2$ = $logn$
- $k = logn$ //if we assume base-2
- Total time = O($logn$).

Guidelines for Asymptotic Analysis

- Initially i = l, in next step i = 2, and in subsequent steps i = 4,8 and so on.
- Let us assume that the loop is executing some k times. At k^{th} step 2^k = n and we come out of loop. Taking logarithm on both sides, gives
- log(2^k) = $logn$
- $klog^2$ = $logn$
- $k \ = \ logn$ //if we assume base-2
- Total time = O($logn$).

Guidelines for Asymptotic Analysis

- def Function(n):
  - count= 0 c_0
  - for i in range(n/2, n): #Outer loop execute n/2 times n
    - j = 1 c_1
    - While j + n/2 <= n: #Middle loop executes n/2 times n
      - k=l c_2
      - while k <= n: #tinner loop execute lo9n times n $n\log n$
        - count = count + 1
        - k = k * 2 $\log n$
      - j = j + l
  - print (count) c_3 $n^2 \log n$
- Function(20)
- The complexity of the above function is O(n^2logn).