JYOTHISHMATHI INSTITUTE OF TECHNOLOGY & SCIENCE
Nustulapur, Karimnagar – 505481.

Data Structures through C++

II Year B.Tech. I Sem.

2018-19

N.MAHESH KUMAR
ASSOCIATE PROFESSOR
CSE

Asymptotic Analysis

Why performance analysis?

There are many important things that should be taken care of, like

- user friendliness,
- modularity,
- security,
- maintainability, etc.

Why to worry about performance?

The answer to this is simple, we can have all the above things **only if we have performance**. So performance is like currency through which we can buy all the above things.

Asymptotic notations

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that
  - **Doesn’t depend on machine specific constants,**
  - **Doesn’t require algorithms to be implemented**
  - **Time taken by programs to be compared.**
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Asymptotic Complexity

- Two important reasons to determine operation and step counts
  1. To compare the time complexities of two programs that compute the same function
  2. To predict the **growth** in run time as the instance characteristic changes

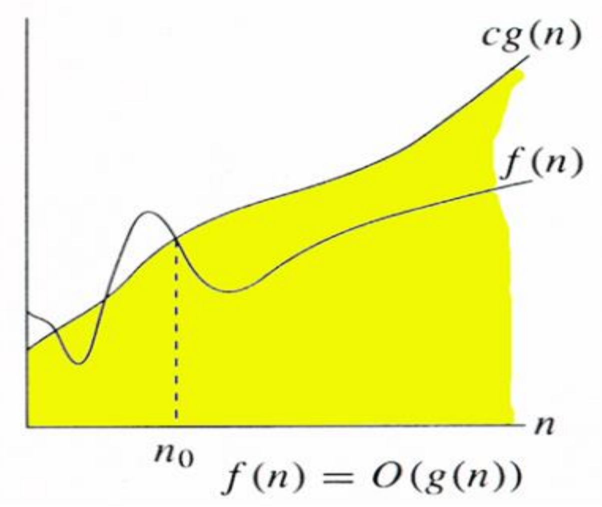
- Neither of the two yield a very accurate measure
  - **Operation counts:** focus on “key” operations and ignore all others
  - **Step counts:** the notion of a step is itself inexact
- **Asymptotic complexity** provides meaningful statements about the time and space complexities of a program

Asymptotic Notation

- Describes the behavior of the time or space complexity for large instance characteristic
- **Big Oh (O)** notation provides an **upper bound** for the function f
- **Omega (Ω)** notation provides a **lower-bound**
- **Theta (Θ)** notation is used when an algorithm can be **bounded both from above and below** by the same function

Big "oh'' --- O

- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above
- **In typical usage, the formal definition of *O* notation is not used directly; rather, the *O* notation for a function *f*(*x*) is derived by the following simplification rules:**
- If *f*(*x*) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If *f*(*x*) is a product of several factors, any constants (terms in the product that do not depend on *x*) are omitted.

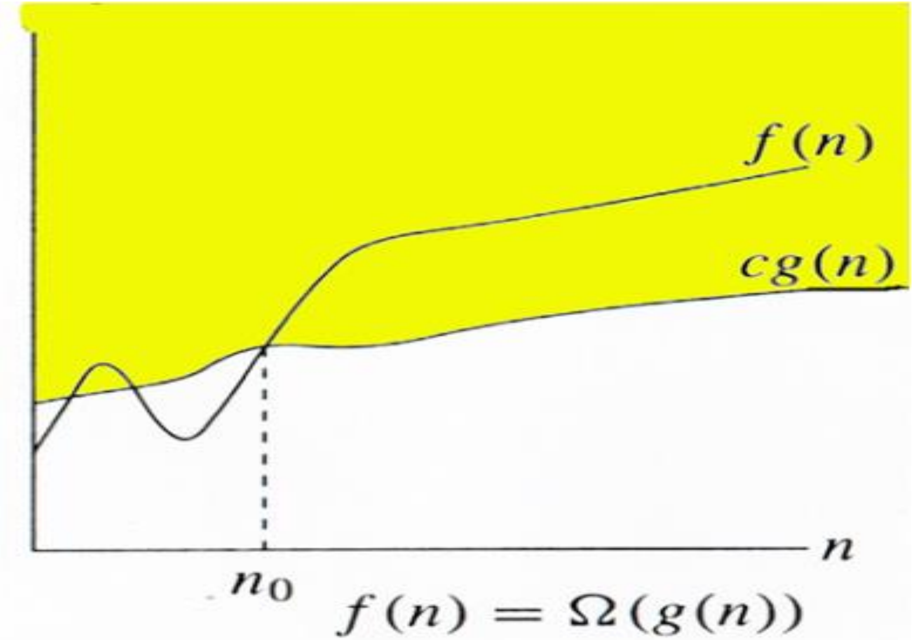


- **Definition**: [Big "oh'']
  - $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.

- **Examples**
  - $f(n) = 3n+2$
    - 3n + 2 <= 4n, for all n >= 2, $\therefore$ 3n + 2 = $O(n)$

Omega - Ω

- Ω notation provides an asymptotic lower bound.
- Omage notation can be useful when we have lower bound on time complexity of an algorithm.
- the Omega notation is the least used notation among all three.

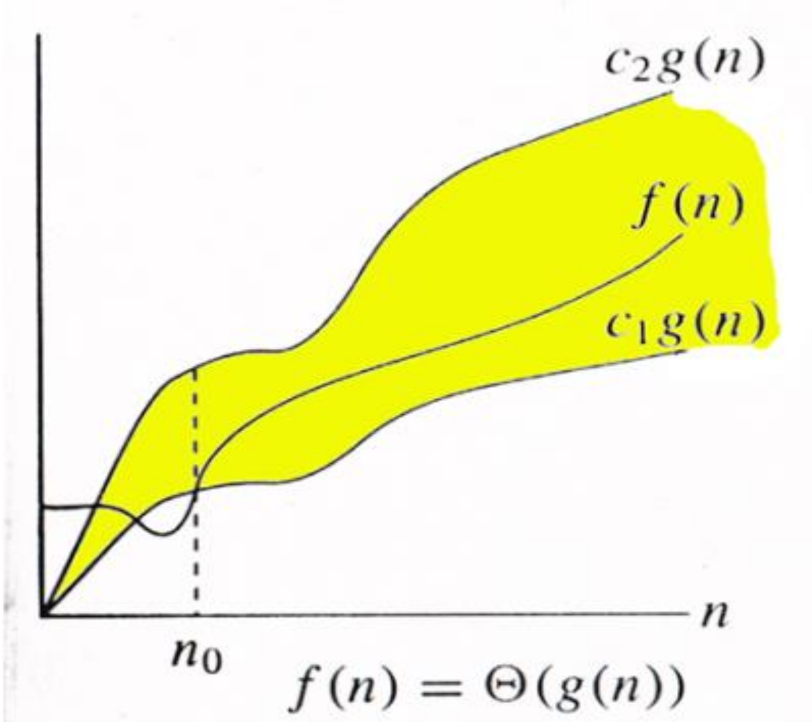


- **Definition:** [Omega]
  - $f(n) = \Omega(g(n))$ (read as "*f* of *n* is omega of *g* of *n*") iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.

- **Examples**
  - $f(n) = 3n+2$
    - 3n + 2 >= 3n, for all n >= 1, $\therefore$ 3n + 2 = $\Omega(n)$

Theta - Θ

- The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
- A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.



Performance Analysis

- **Definition:** [Theta]
  - $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.

- **Examples**
  - $f(n) = 3n+2$
    - $3n <= 3n + 2 <= 4n$, for all $n >= 2$, $\therefore$ $3n + 2 = \Theta(n)$

Performance Analysis

- ***Example :*** **Figure** Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2 $= O(n)$