MSMC-204: Design and Analysis of Algorithms

M.Sc. in Mathematics and Computing II-Semester Session 2024-25

LECTURE 2: by Dr. Supriya Chanda

- Asymptotic Analysis

Analyzing Algorithms

- Predict how your algorithm performs in practice
- By analyzing several candidate algorithms for a problem we can identify efficient ones
- Criteria:
    - Running time
    - Space usage
    - Cache I/O
    - Main memory I/O
    - Lines of codes

Asymptotic Analysis

- It is a technique of representing limiting behavior.
- It can be used to analyze the performance of an algorithm for some large data set.
- The asymptotic behavior of a function ***f(n)*** refers to the growth of ***f(n)*** as **n** gets large.
- We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs.
- A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it's not always true.
- $f\ (n) = n^2+3n$

Asymptotic Notations

- used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

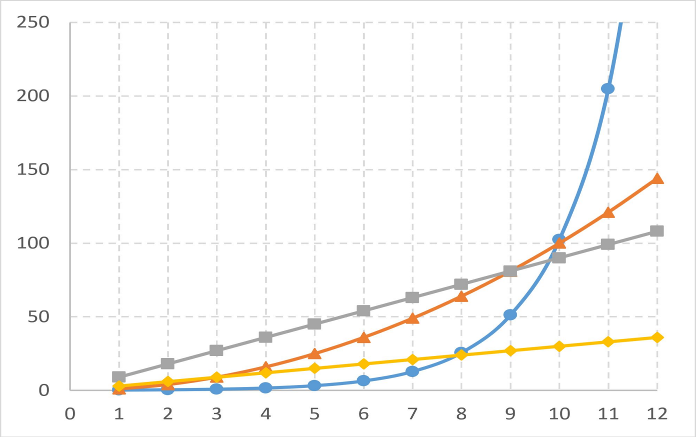
Best/Average/Worst Case Analysis

- We looked at both `best case' (input array was already sorted) and `worst case' (input array was reverse sorted)
- In this course, we shall usually concentrate on worst-case running time
- Major reasons
  - Gives an upper bound on the running time for any input
  - For some algorithms, worst case occurs fairly often, e.g., searching
  - Average case is often roughly as bad as the worst case.

Order of Growth

- In analyzing running time for `insertion sort', we started with constants c_i to represent the cost of each statement
- Then we observed that they give more detail than we need and we discarded them
- We shall go ahead with more simplifying abstraction: **Rate/Order of Growth**
- For the function $f(n)$ we care when n is large enough. When n is small, $f(n)$ is small anyway
- The constant factors and lower order terms doesn't affect the growth of the function
- One algorithm is more efficient than another if its worst-case running time has a lower order of growth

Order of Growth



$g_2(n) = 0.1 \times 2^n$

$g_1(n) = n^2$

$f_2(n) = 9n$

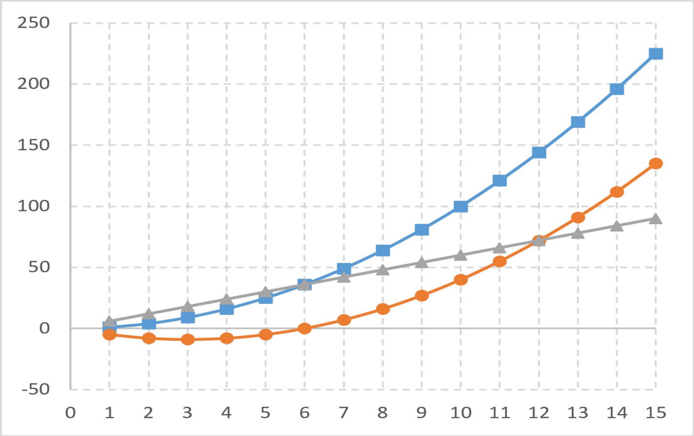
$f_1(n) = 3n$

Omit the constant factors

When n is large enough, $g_1(n)$ will be much larger than $f_1(n)$ or $f_2(n)$

$f_1(n)$ and $f_2(n)$ will have similar growth trend

Order of Growth



$g_1(n) = n^2$

$g_2(n) = n^2 - 6n$

$f(n) = 6n$

Omit the lower-order terms

When n is large enough, $g_1(n)$ or $g_2(n)$ will still be much larger than $f(n)$

$g_1(n)$ and $g_2(n)$ will have similar growth trend because $-6n$ is much smaller compared to n^2

Big Oh, O: Asymptotic Upper Bound

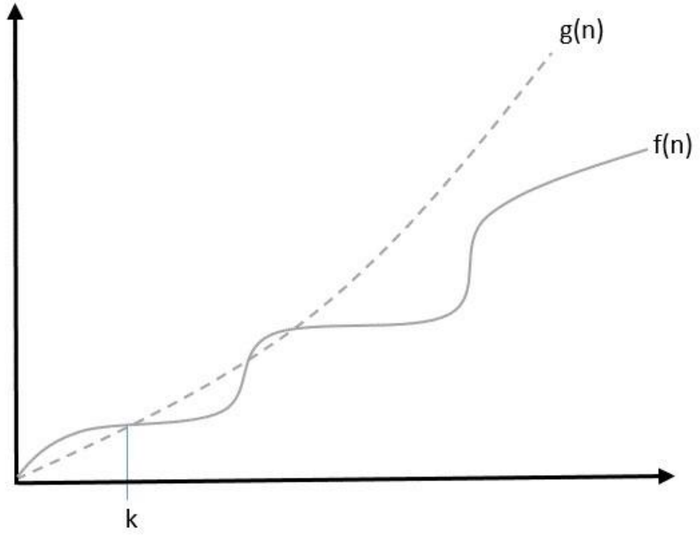
- The notation O(n) is the formal way to express the upper bound of an algorithm's running time.
- It is the most commonly used notation.
- It measures the **worst case time complexity** or the longest amount of time an algorithm can possibly take to complete.

A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

$$f(n) \leqslant c.g(n) \text{ for } n > n_0 \text{ in all case}$$

Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

Big Oh, O: Asymptotic Upper Bound



Example

Let us consider a given function, $f(n) = 4.n^3 + 10.n^2 + 5.n + 1$

Considering $g(n) = n^3$,

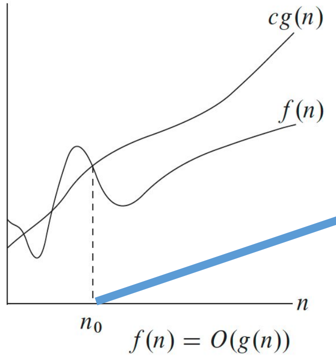
$f(n) \leqslant 5.g(n)$ for all the values of $n > 2$

Hence, the complexity of **f(n)** can be represented as $O(g(n))$, i.e. $O(n^3)$

O (Big-O) [≤]

$$O\big(g(n)\big) = \{f(n): \exists\ c > 0, n_0 > 0, such\ that\ 0 \le f(n) \le cg(n) \forall\ n \ge n_0\}$$

- $f(n) = 3n^2, g(n) = n^2$
- How can we show, $f(n) = O\big(g(n)\big)$
- Let $c = 3, n_0 = 5$
- $cg(n) = 3n^2$, so $f(n) \le cg(n)$
- Similarly, let $c = 10, n_0 = 2$
- $cg(n) = 10n^2$, so $f(n) \le cg(n)$



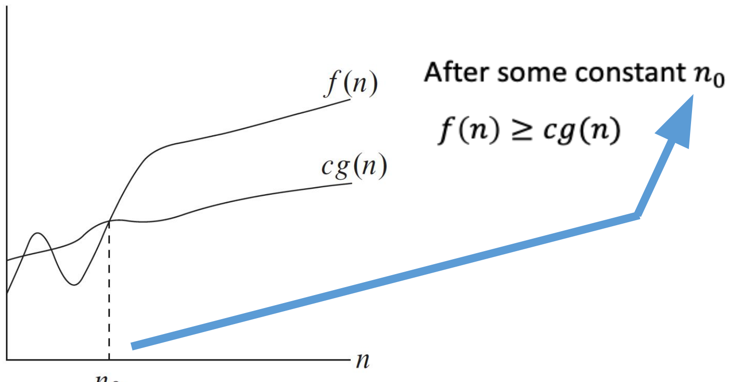
After some constant n_0
$f(n) \le cg(n)$

Always larger than
$n^2 + 2n$

- $f(n) = n^2 + 2n, g(n) = n^3$; How can we show, $f(n) = O\big(g(n)\big)$?
- Let $c = 3, n_0 = 10$; $cg(n) = 3n^3 = n^3 + 2n^3 = n(n^2 + 2n^2)$
- so $f(n) \le cg(n)$

Ω (Big- Ω) [≥]

$$\Omega(g(n)) = \{f(n)\colon \exists\, c > 0, n_0 > 0, such\ that\ 0 \le cg(n) \le f(n)\ \forall\, n \ge n_0\}$$

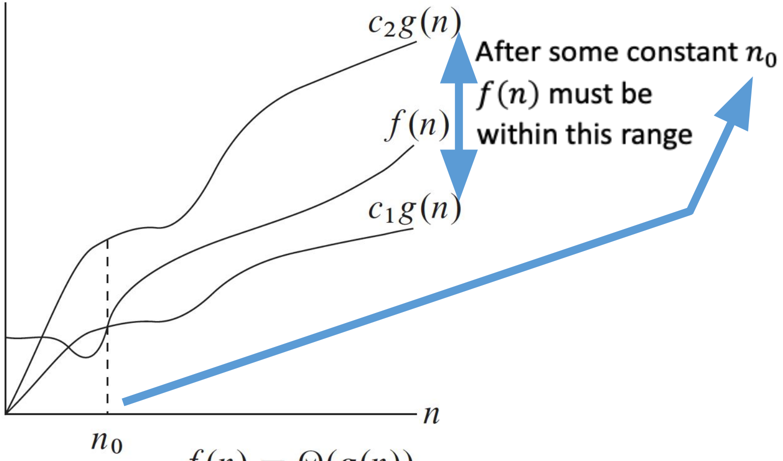


$f(n) = \Omega(g(n))$

- Asymptotic lower bound
- can be of the same order, but can be larger as well

Θ (Big- theta) [=]

$\Theta(g(n)) = \{f(n)\text{: } \exists\ c_1, c_2 > 0, n_0 > 0, s.t.\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n$



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

- Asymptotic tight bound
- Must be of the same order
- $f(n) = \Theta(g(n))$ means $f(n) = O(g(n))$ [≤] and $f(n) = \Omega(g(n))$ [≥], must be =

What This also Means

- $O(g(n))$: class of functions $f(n)$ that grow <u>no faster</u> than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$