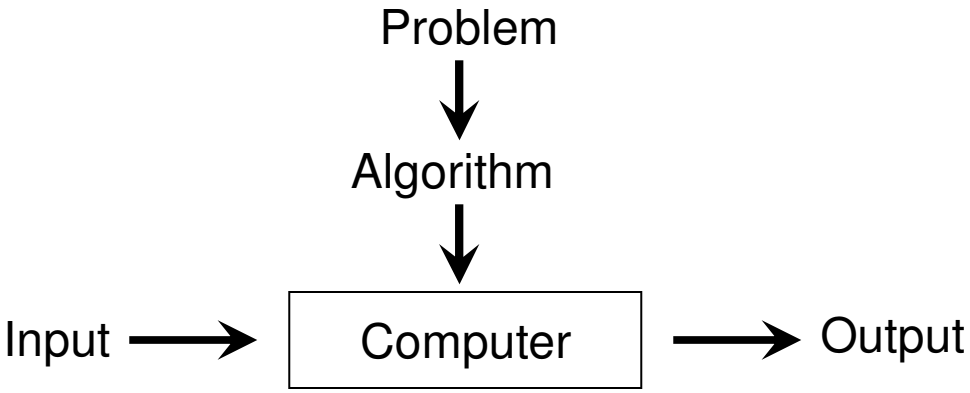
Module 1: Analyzing the Efficiency of Algorithms

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Based on Chapter 2 in the Textbook

What is an Algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



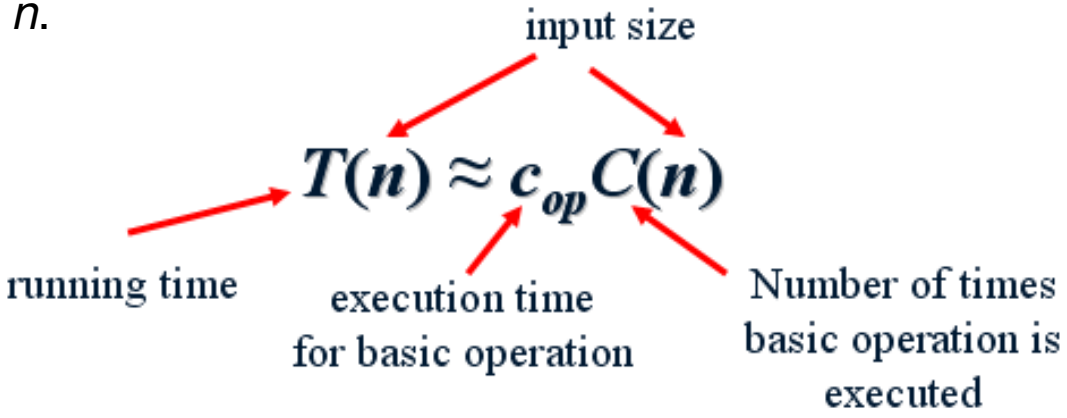
- Important Points about Algorithms
  - The non-ambiguity requirement for each step of an algorithm cannot be compromised
  - The range of inputs for which an algorithm works has to be specified carefully.
  - The same algorithm can be represented in several different ways
  - There may exist several algorithms for solving the same problem.
    - Can be based on very different ideas and can solve the problem with dramatically different speeds

The Analysis Framework

- **Time efficiency (time complexity):** indicates how fast an algorithm runs
- **Space efficiency (space complexity):** refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output
- Algorithms that have non-appreciable space complexity are said to be ***in-place***.
- The time efficiency of an algorithm is typically as a function of the input size (one or more input parameters)
  - Algorithms that input a collection of values:
    - The time efficiency of sorting a list of integers is represented in terms of the number of integers (*n*) in the list
    - For matrix multiplication, the input size is typically referred as n*n.
    - For graphs, the input size is the set of Vertices (V) and edges (E).
  - Algorithms that input only one value:
    - The time efficiency depends on the magnitude of the integer. In such cases, the algorithm efficiency is represented as the number of bits $1+\lfloor \log_2 n \rfloor$ needed to represent the integer n

Units for Measuring Running Time

- The running time of an algorithm is to be measured with a unit that is independent of the extraneous factors like the processor speed, quality of implementation, compiler and etc.
  - At the same time, it is not practical as well as not needed to count the number of times, each operation of an algorithm is performed.
- Basic Operation: The operation contributing the most to the total running time of an algorithm.
  - It is typically the most time consuming operation in the algorithm's innermost loop.
    - **Examples:** Key comparison operation; arithmetic operation (division being the most time-consuming, followed by multiplication)
  - We will count the number of times the algorithm's basic operation is executed on inputs of size *n*.



Examples for Input Size and Basic Operations

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer *n*	*n*'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Orders of Growth

- We are more interested in the order of growth on the number of times the basic operation is executed on the input size of an algorithm.
- Because, for smaller inputs, it is difficult to distinguish efficient algorithms vs. inefficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and n^2 respectively, then
  - if $n = 3$, then we may say there is not much difference between requiring 3 basic operations and 9 basic operations and the two algorithms have about the same running time.
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is n or n^2.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Exponential-growth functions

Source: Table 2.1
From Levitin, 3rd ed.

Best-case, Average-case, Worst-case

- For many algorithms, the actual running time may not only depend on the input size; but, also on the specifics of a particular input.
  - For example, sorting algorithms (like insertion sort) may run faster on an input sequence that is *almost-sorted* rather than on a randomly generated input sequence.

- **Worst case:** $C_{worst}(n)$ – maximum number of times the basic operation is executed over inputs of size n
- **Best case:** $C_{best}(n)$ – minimum # times over inputs of size n
- **Average case:** $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Example for Worst and Best-Case Analysis: Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

/* Assume the second condition will not be executed if the first condition evaluates to false */

- Worst-Case: $C_{worst}(n) = n$
- Best-Case: $C_{best}(n) = 1$

Probability-based Average-Case Analysis of Sequential Search

- If *p* is the probability of finding an element in the list, then (1-*p*) is the probability of not finding an element in the list.
- Also, on an *n*-element list, the probability of finding the search key as the i^{th} element in the list is *p/n* for all values of $1 \le i \le n$

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

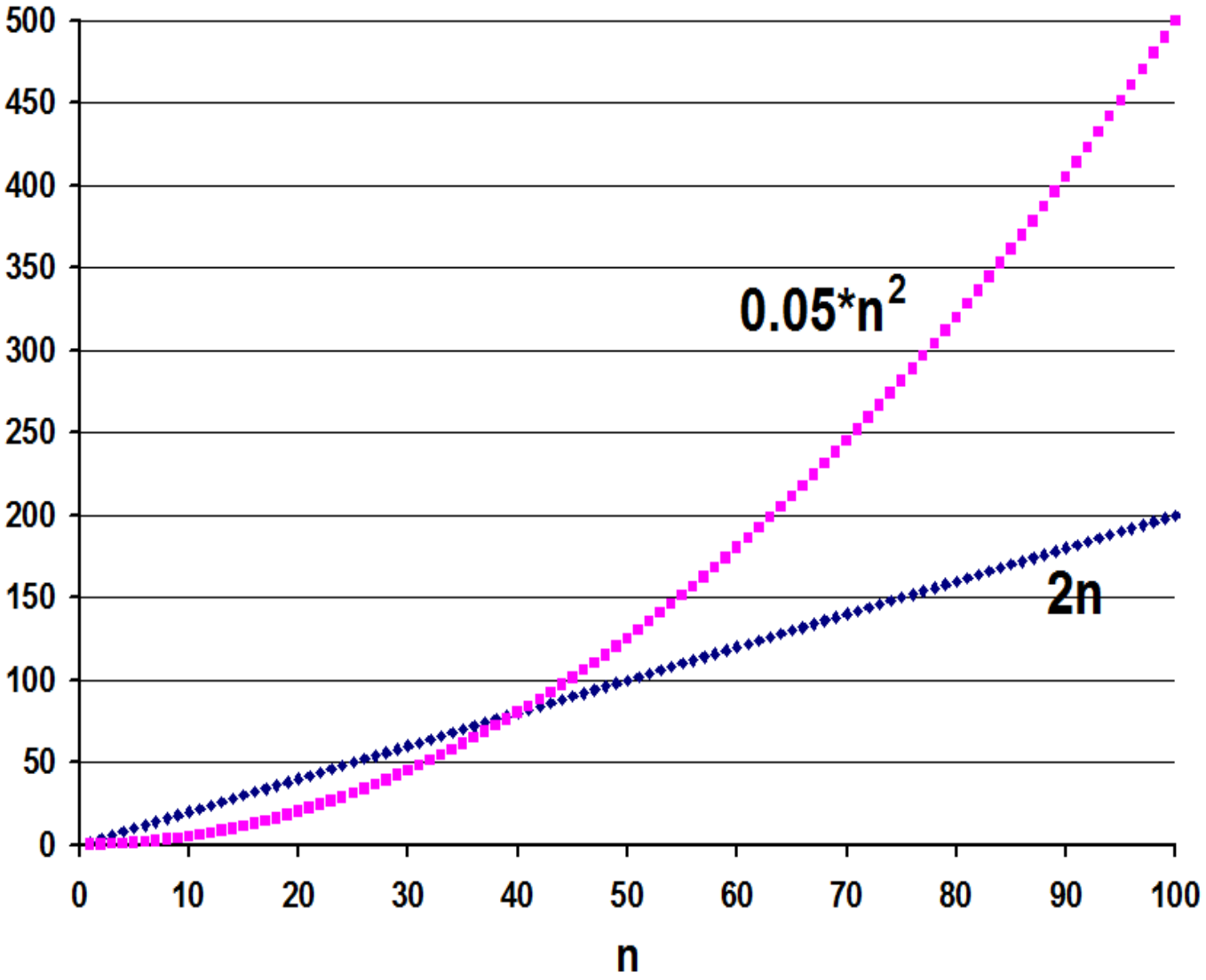
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

- If p = 1 (the element that we will search for always exists in the list), then $C_{avg}(n) = (n+1)/2$. That is, on average, we visit half of the entries in the list to search for any element in the list.
- If p = 0 (all the time, the element that we will search never exists), then $C_{avg}(n) = n$. That is, we visit all the elements in the list.

YouTube Link: https://www.youtube.com/watch?v=8V-bHrPykrE

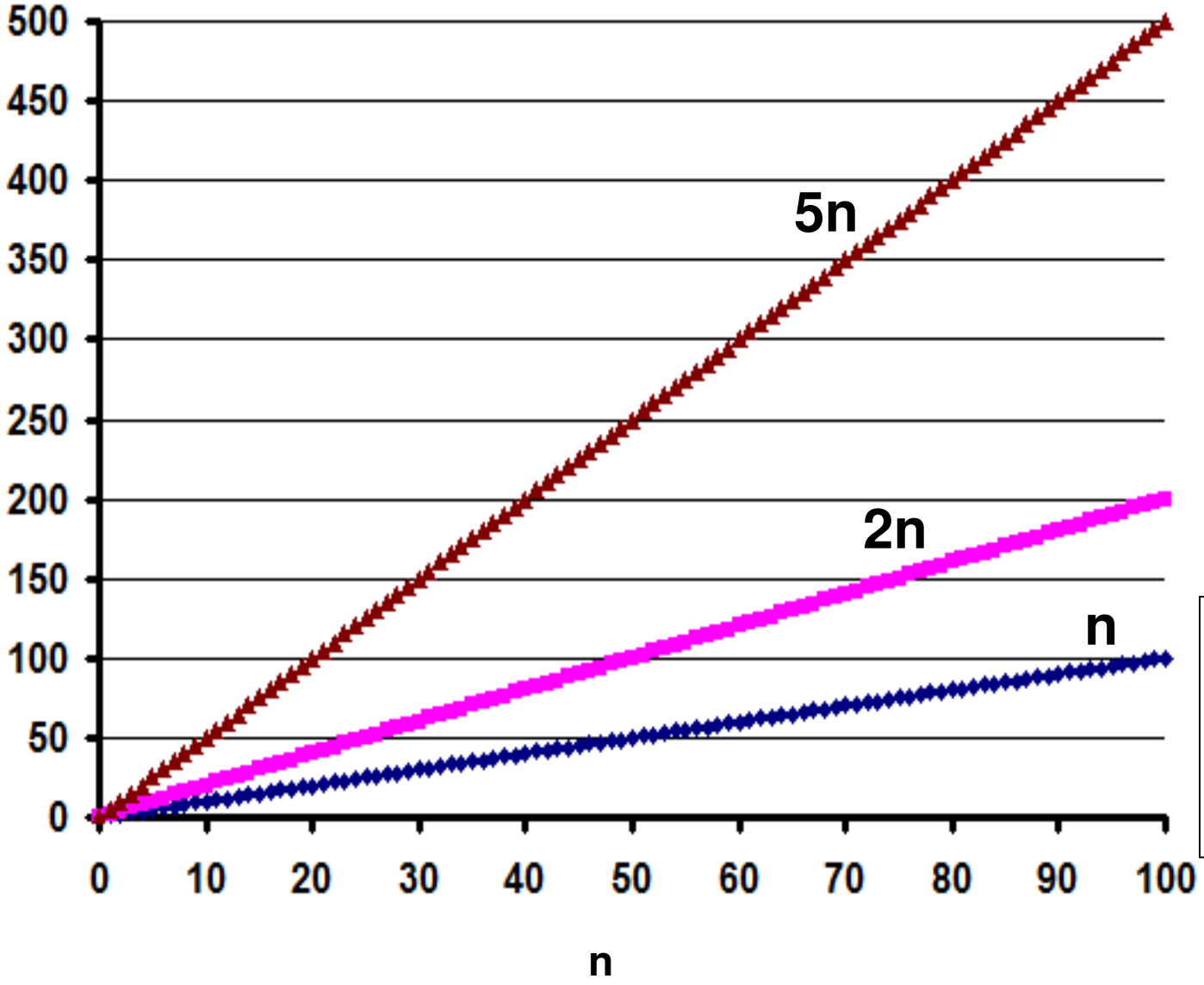
Asymptotic Notations: Intro



$2n \leq 0.05\ n^2$
for $n \geq 40$
$2n = O(n^2)$

$0.05n^2 \geq 2n$
for $n \geq 40$
$0.05n^2 = \Omega(n)$

Asymptotic Notations: Intro

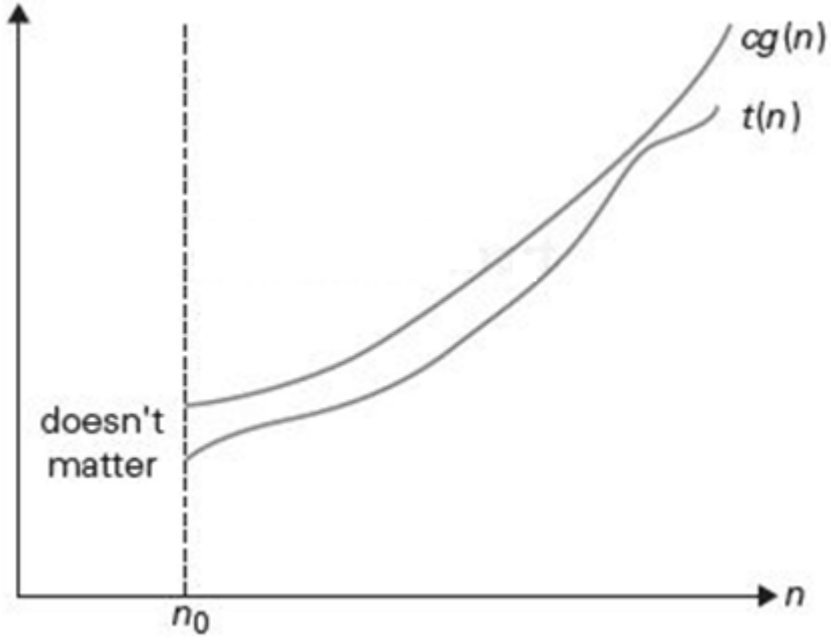


$2n \leq 5n$
for $n \geq 1$
$2n = O(n)$

$2n \geq n$
for $n \geq 1$
$2n = \Omega(n)$

As $2n = O(n)$ and $2n = \Omega(n)$, we say $2n = \Theta(n)$

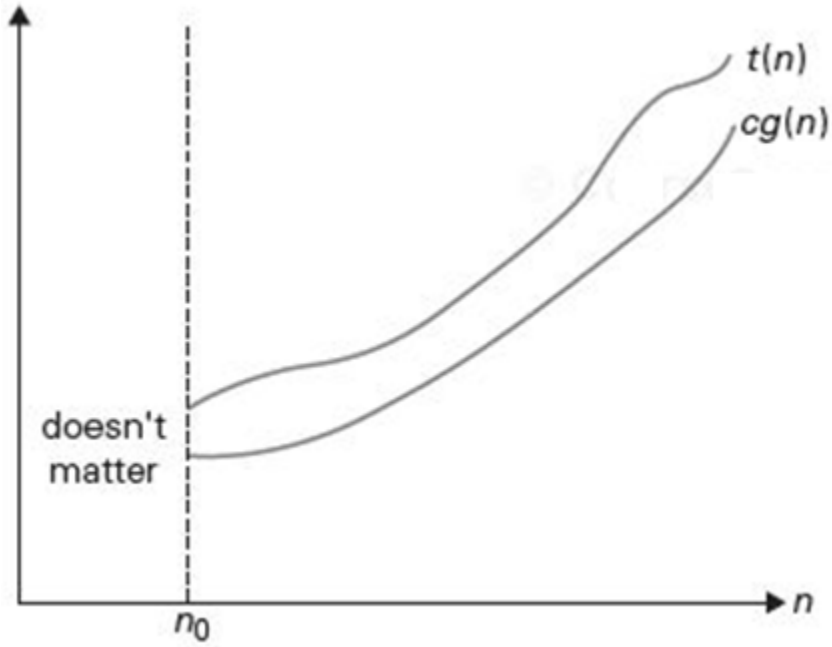
Asymptotic Notations: Formal Intro



$t(n) = O(g(n))$

$t(n) \leq c*g(n)$ for all $n \geq n_0$

c is a positive constant (> 0)
and n_0 is a non-negative integer



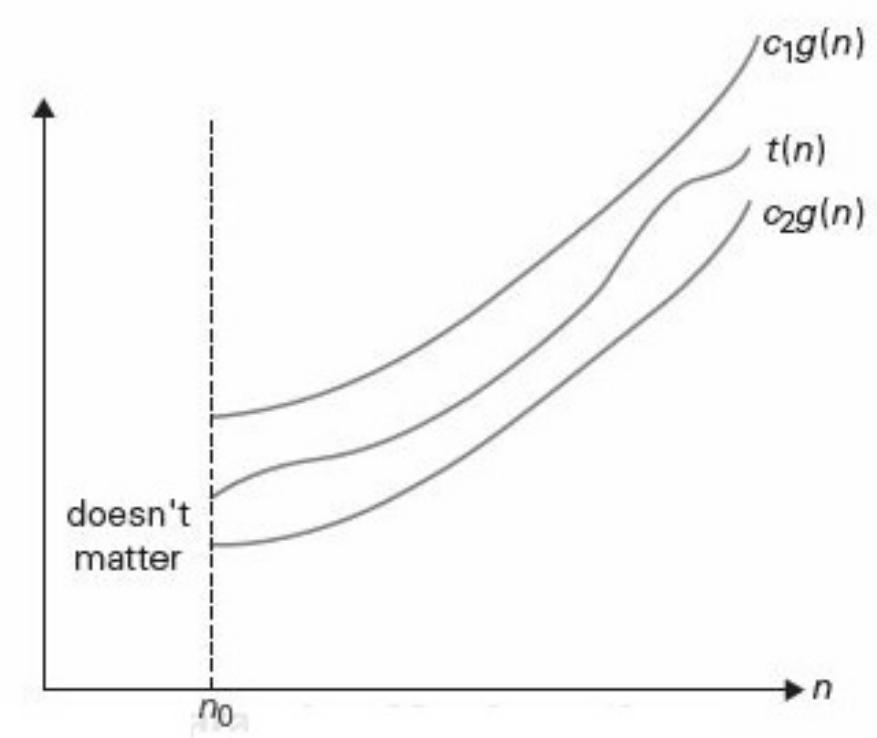
$t(n) = \Omega(g(n))$

$t(n) \geq c*g(n)$ for all $n \geq n_0$

c is a positive constant (> 0)
and n_0 is a non-negative integer

Note: If $t(n) = O(g(n))$ ➔ $g(n) = \Omega(t(n))$; also, if $t(n) = \Omega(g(n))$ ➔ $g(n) = O(t(n))$

Asymptotic Notations: Formal Intro



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

$$c2*g(n) \leq t(n) \leq c1*g(n) \text{ for all } n \geq n_0$$

c1 and c2 are positive constants (> 0)
and n_0 is a non-negative integer

Asymptotic Notations: Examples

- Let t(n) and g(n) be any non-negative functions defined on a set of all real numbers.
- We say t(n) = O(g(n)) for all functions t(n) that have a lower or the same order of growth as g(n), within a constant multiple as n → ∞.
  - **Examples:** $n \in O(n)$, $n \in O(n^2)$, $100n + 5 \in O(n^2)$, $\frac{1}{2}n(n-1) \in O(n^2)$

    $n^3 \notin O(n^2)$, $0.00001n^3 \notin O(n^2)$, $n^4 + n + 1 \notin O(n^2)$

- We say t(n) = Ω(g(n)) for all functions t(n) that have a higher or the same order of growth as g(n), within a constant multiple as n → ∞.
  - **Examples:** $n \in \Omega(n)$, $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, $100n + 5 \notin \Omega(n^2)$
- We say t(n) = Θ(g(n)) for all functions t(n) that have the same order of growth as g(n), within a constant multiple as n → ∞.
  - **Examples:** $an^2 + bn + c = \Theta(n^2)$;

    $n^2 + \log n = \Theta(n^2)$

Useful Property of Big-Oh Notation

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

For any four arbitrary real numbers, a_1, b_1, a_2 and b_2 such that $a_1 \leq b_1$ and $a_2 \leq b_2$,
We have $a_1 + a_2 \leq 2 \max\{ b_1, b_2 \}$

Since $t_1(n) \in O(g_1(n))$, then there exists some constant c_1 and non-negative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1$$

Since $t_2(n) \in O(g_2(n))$, then there exists some constant c_2 and non-negative integer n_2 such that

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2$$

Let $c_3 = \max\{ c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &= c_3\{ g_1(n) + g_2(n)\} \\ &\leq 2 c_3 \max\{ g_1(n), g_2(n)\} \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{ g_1(n), g_2(n)\}$, with the constants c and n_0 required by the O definition being $2 c_3 = 2 \max\{ c_1, c_2\}$ and $\max\{n_1, n_2\}$ respectively.

Note: The above property implies that if an algorithm comprises of two or more consecutively executed parts, the overall efficiency of the algorithm is determined by the part with the highest order of growth.

Extending the Property

- The property that we proved in the previous slide can be extended directly for the Θ notation:
- If $t_1(n) \in \Theta(g_1(n))$ and $t_2(n) \in \Theta(g_2(n))$, then

$$t_1(n) + t_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$$

- The property can be applied for the Ω notation with a slight change: Replace the Max with the Min.
- If $t_1(n) \in \Omega(g_1(n))$ and $t_2(n) \in \Omega(g_2(n))$, then

$$t_1(n) + t_2(n) \in \Omega(\min\{g_1(n), g_2(n)\})$$

Using Limits to Compare Order of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

The first case means t(n) = O(g(n)
if the second case is true, then t(n) = Θ(g(n))
The last case means t(n) = Ω(g(n))

L'Hopital's Rule $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

Note: t'(n) and g'(n) are first-order derivatives of t(n) and g(n)

Stirling's Formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Example 1: To Determine the Order of Growth

Find the class $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$ for the following function:

(a) $(n^2+1)^{10}$.

$O(g(n))$: Let $g(n) = n^{21}$

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{21}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}.n} = \lim_{n\to\infty} \frac{1}{n}\left[\frac{n^2+1}{n^2}\right]^{10} = \lim_{n\to\infty} \frac{1}{n}\left[1+\frac{1}{n^2}\right]^{10}$$

$$= 0 * 1 = 0 \Rightarrow (n^2+1)^{10} = O(n^{21})$$

Example 1: To Determine the Order of Growth (continued…)

(b) $\Theta(g(n))$: Let $g(n) = n^{20}$

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}}$$

$$= \lim_{n\to\infty} \left[\frac{n^2+1}{n^2}\right]^{10} = \lim_{n\to\infty} \left[1+\frac{1}{n^2}\right] = 1$$

$$\Rightarrow \underline{\underline{(n^2+1)^{10} = \Theta(n^{20})}}$$

(c) $\Omega(g(n))$: Let $g(n) = n^{10}$.

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{10}} = \lim_{n\to\infty} \left[\frac{n^2+1}{n}\right]^{10} = \lim_{n\to\infty} \left[n+\frac{1}{n}\right]^{10} = \lim_{n\to\infty} n^{10} = \infty$$

$$(n^2+1)^{10} = \Omega(n^{10})$$

Example 2: To Determine the Order of Growth

Find the class $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$ for the following functions:

(b) $\sqrt{3n^2+7n+4}$ $\rightarrow t(n)$

$O(g(n))$: $\sqrt{3n^2+7n+4} \leq \sqrt{n^2} = n$.

Pick $g(n) = n^2 = \sqrt{n^4}$ [degree of $g(n)$ greater than $t(n)$]

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n^4}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n^4}} = \lim_{n\to\infty} \sqrt{\frac{3}{n^2}+\frac{7}{n^3}+\frac{4}{n^4}}$$

$$= 0$$

$$\Rightarrow \sqrt{3n^2+7n+4} = O(n^2)$$

Example 2: To Determine the Order of Growth (continued…)

$\Theta(g(n))$:

Pick $g(n)$ to be of the same degree as $f(n) = \sqrt{3n^2+7n+4}$ $\Rightarrow \sqrt{n^2} = n$.

$g(n) = \sqrt{n^2} = n$.

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n^2}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n^2}} = \lim_{n\to\infty} \sqrt{3+\frac{7}{n}+\frac{4}{n^2}}$$

$$= \sqrt{3}$$

$$\sqrt{3n^2+7n+4} = \Theta(n)$$

$\Omega(g(n))$:

Pick $g(n)$ to be of a lower degree than $f(n) = \sqrt{3n^2+7n+4}$ $\Rightarrow n$.

So, pick $g(n) = n^{1/2} = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\sqrt{3n^2+7n+4}}{\sqrt{n}} = \lim_{n\to\infty} \sqrt{\frac{3n^2+7n+4}{n}} = \lim_{n\to\infty} \sqrt{3n+7+\frac{4}{n}} = \infty.$$

So, $\sqrt{3n^2+7n+4} = \Omega(\sqrt{n})$

Examples to Compare the Order of Growth

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2.

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}.$$

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$.

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$$

$$\log_2 n \in O(\sqrt{n}).$$

Example 3: Compare the order of growth of $\log^2 n$ and $\log n^2$.

$$\lim_{n->\infty} \frac{\log^2 n}{\log n^2} = \lim_{n->\infty} \frac{\log n * \log n}{\log(n*n)} = \lim_{n->\infty} \frac{\log n * \log n}{\log n + \log n} = \lim_{n->\infty} \frac{\log n * \log n}{2\log n} = \lim_{n->\infty} \frac{\log n}{2} = \infty$$

Hence, $\log^2 n = \Omega(\log n^2)$ That is, $\log n^2 = O(\log^2 n)$

Some More Examples: Order of Growth

For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.

a. $(n^2+1)^{10}$ **b.** $\sqrt{10n^2+7n+3}$

c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$ **d.** $2^{n+1} + 3^{n-1}$

- a) $(n^2+1)^{10}$: Informally, $= (n^2+1)^{10} \approx n^{20}$.

Formally,

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}|}{(n^2)^{10}} = \lim_{n\to\infty} \left(\frac{n^2+1}{n^2}\right)^{10} == \lim_{n\to\infty} \left(1+\frac{1}{n^2}\right)^{10} = 1.$$

Hence $(n^2+1)^{10} \in \Theta(n^{20})$.

b) Informally, $\sqrt{10n^2+7n+3} \approx \sqrt{10n^2} = \sqrt{10}n \in \Theta(n)$. Formally,

$$\lim_{n\to\infty} \frac{\sqrt{10n^2+7n+3}}{n} = \lim_{n\to\infty} \sqrt{\frac{10n^2+7n+3}{n^2}} = \lim_{n\to\infty} \sqrt{10+\frac{7}{n}+\frac{3}{n^2}} = \sqrt{10}.$$

Hence $\sqrt{10n^2+7n+3} \in \Theta(n)$.

c) $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2} = 2n2 \lg(n+2) + (n+2)^2(\lg n - 1)$

$\in \Theta(n \lg n) + \Theta(n^2 \lg n) = \Theta(n^2 \lg n)$.

d) $2^{n+1} + 3^{n-1} = 2^n 2 + 3^n \frac{1}{3} \in \Theta(2^n) + \Theta(3^n) = \Theta(3^n)$

Some More Examples: Order of Growth

List the following functions according to their order of growth from the lowest to the highest:

$$(n-2)!,\quad 5\lg(n+100)^{10},\quad 2^{2n},\quad 0.001n^4+3n^3+1,\quad \ln^2 n,\quad \sqrt[3]{n},\quad 3^n$$

$(n-2)! \in \Theta((n-2)!)$

$5\lg(n+100)^{10} = 50\lg(n+100) \in \Theta(\log n)$

$2^{2n} = (2^2)^n \in \Theta(4^n)$

$0.001n^4 + 3n^3 + 1 \in \Theta(n^4)$

$\ln^2 n \in \Theta(\log^2 n)$

$\sqrt[3]{n} \in \Theta(n^{\frac{1}{3}})$

$3^n \in \Theta(3^n)$

The listing of the functions in the increasing Order of growth is as follows:

$$5\lg(n+100)^{10},\quad \ln^2 n,\quad \sqrt[3]{n},\quad 0.001n^4+3n^3+1,\quad 3^n,\quad 2^{2n},\quad (n-2)!$$

$$\lim_{n\to\infty} \frac{\log^2 n}{n^{1/3}} = \lim_{n\to\infty} \frac{2\log n}{n^*(1/3)n^{(-2/3)}} = \frac{6\log n}{n^{(1/3)}} = \frac{6}{n^*(1/3)n^{(-2/3)}} = \lim_{n\to\infty} \frac{18}{n^{(1/3)}} = 0$$

Hence, $\log^2 n = O(n^{1/3})$

Time Efficiency of Non-recursive Algorithms: *General Plan for Analysis*

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*, if the number of times the basic operation gets executed varies with specific instances (inputs)
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Useful Summation Formulas and Rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i \qquad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

Examples on Summation

- $1 + 3 + 5 + 7 + \ldots + 999$

$$= [1 + 2 + 3 + 4 + 5 + \ldots 999] - [2 + 4 + 6 + 8 + \ldots + 998]$$

$$= \frac{999 * 1000}{2} - 2[1 + 2 + 3 + \ldots + 499]$$

$$= 999 * 500 - 2\left[\frac{499 * 500}{2}\right] = 999 * 500 - 499 * 500$$

$$= 500*(999\text{-}499) = 500*500 = 250{,}000$$

- $2 + 4 + 8 + 16 + \ldots + 1024$

$$= 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}$$

$$= [2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}] - 1$$

$$= \left[\sum_{i=0}^{10} 2^i\right] - 1 = [2^{11} - 1] - 1 = \mathbf{2046}$$

$$\sum_{i=3}^{n+1} 1 = [(n+1) - 3 + 1] = n+1 - 2 = n-1 = \Theta(n)$$

$$\sum_{i=3}^{n+1} i = 3 + 4 + ... + (n+1) = [1 + 2 + 3 + 4 + ... + (n+1)] - [1 + 2]$$

$$= \frac{(n+1)(n+2)}{2} - 3 = \Theta(n^2) - \Theta(1) = \Theta(n^2)$$

$$\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} i^2 + i = \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i$$

$$= \left[\frac{[n-1][(n-1)+1][2(n-1)+1]}{6}\right] + \left[\frac{[n-1][(n-1)+1]}{2}\right]$$

$$= \left[\frac{[n-1][n][2n-1]}{6}\right] + \left[\frac{[n-1][n]}{2}\right]$$

$$= \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$$

$$\sum_{i=0}^{n-1}(i^2+1)^2 = \sum_{i=0}^{n-1}(i^4+2i^2+1) = \sum_{i=0}^{n-1} i^4 + 2\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} 1$$

$$\in \Theta(n^5) + \Theta(n^3) + \Theta(n) = \Theta(n^5)$$

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$$

$$= \sum_{i=0}^{n-1}[\sum_{j=0}^{i-1} i + \sum_{j=0}^{i-1} j]$$

$$= \sum_{i=0}^{n-1}[i^2 + \frac{(i-1)i)}{2}]$$

$$= \sum_{i=0}^{n-1}[\frac{3}{2}i^2 - \frac{1}{2}i]$$

$$= \frac{3}{2}\sum_{i=0}^{n-1} i^2 - \frac{1}{2}\sum_{i=0}^{n-1} i$$

$$\in \Theta(n^3) - \Theta(n^2) = \Theta(n^3)$$

Example 1: Finding Max. Element

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

- The basic operation is the comparison executed on each repetition of the loop.
- In this algorithm, the number of comparisons is the same for all arrays of size n.
- The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n-1 (inclusively). Hence,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Example 2: Sequential Key Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

/* Assume the second condition will not be executed if the first condition evaluates to false */

- Worst-Case: $C_{worst}(n) = n$
- Best-Case: $C_{best}(n) = 1$

Depending on the value of the Key K, the number of comparisons performed can range from 1 to n.

In such cases, it would be more appropriate to denote the overall time complexity for the basic operation as **O(n)**, rather than **Θ(n).**

Example 3: Finding Prime

```
Input: Integer n
Output: True (is prime) or False (not prime)

for i = 2 to n -1 do
    if (n mod i = 0) then
        return false
    end if
end for

return true
```

The number of division operations could range anywhere from 2 to n-1.
For example, if n = 9, the number of divisions is only 2 (for i = 2 and 3)
If n = 541, a prime the number of divisions is 539 (for i = 2, 3, …, 540)

Thus, for a given n, the number of divisions could be as small as 1,
and as large as n – 2. In such cases, it would be more appropriate to denote the overall time complexity for the basic operation as $O(n)$, rather than $\Theta(n)$.

Rule of Thumb

- We will use the notation Θ(f(n)), when the number of times the basic operation will be executed is tightly bound to f(n).
  - Like in Example 1 – To find the maximum element in an array
  - In general: When the best case / worst case ratio tends to a constant (which was 1 for Example 1), as n grows to ∞.
- We will use the notation O(f(n)), when the number of times the basic operation will be executed is bounded at most (i.e., the upper bound) to f(n) and the lower bound could be anything arbitrary (like 1).
  - Like in Example 3 – To find whether an integer is prime or not.
  - In general: When the best case / worst case ratio tends to 0, as n grows to ∞. For Example 3, the best case / worst case ratio is 1/(n-2), and this ratio evaluates to 0, as n grows to ∞.
- **Comparison of O and Θ: Which is better?**
- If two algorithms A and B for a particular problem incur overall time complexities O(f(n)) and Θ(f(n)), then, it would be better to choose algorithm A with complexity O(f(n)) as the algorithm incurs at most f(n) complexity for certain inputs; where as, algorithm B incurs f(n) complexity on all inputs.

Revisiting Examples 1 and 3

- Example 1: To find the max. element
  - Best-case: n-1 comparisons
  - Worst-case: n-1 comparisons

$$\lim_{n->\infty} \frac{Best-case}{Worst-case} = \lim_{n->\infty} \frac{n-1}{n-1} = \lim_{n->\infty} 1 = 1 \quad \text{(a constant)}$$

After discarding the constants and lower order terms, the overall time complexity is Θ(worst-case comparisons) = **Θ(n).**

- Example 3: To find whether an integer is prime or not
  - Best-case: 1 division
  - Worst-case: n-2 divisions

$$\lim_{n->\infty} \frac{Best-case}{Worst-case} = \lim_{n->\infty} \frac{1}{n-2} = 0$$

After discarding the constants and lower order terms, the overall time complexity is O(worst-case comparisons) = **O(n).**

Example 4: Element Uniqueness Problem

```
ALGORITHM UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct
    //          and "false" otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

Best-case situation:

If the two first elements of the array are the same, then we can exit after one comparison. Best case = 1 comparison.

Worst-case situation:

- The basic operation is the comparison in the inner loop. The worst-case happens for two-kinds of inputs:
  - Arrays with no equal elements
  - Arrays in which only the last two elements are the pair of equal elements

Example 4: Element Uniqueness Problem

- For these kinds of inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits i+1 and n-1; and this is repeated for each value of the outer loop i.e., for each value of the loop's variable i between its limits 0 and n-2. Accordingly, we get,

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

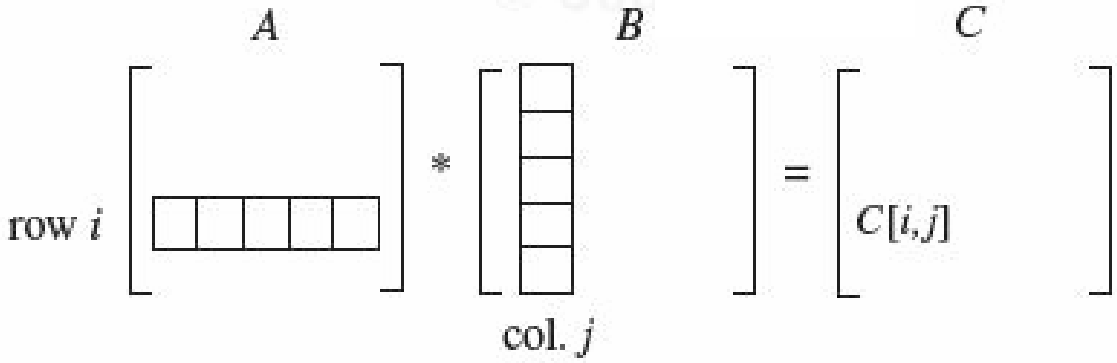
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

$$\lim_{n->\infty} \frac{Best-case}{Worst-case} = \lim_{n->\infty} \frac{1}{n^2/2} = \lim_{n->\infty} \frac{2}{n^2} = 0$$

Hence, overall time Complexity is $O(n^2)$

After discarding the constants and lower order terms, if any

Example 5: Matrix Multiplication



$$C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$$

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Example 5: Matrix Multiplication

- The innermost loop has two arithmetic operations – addition and multiplication – executed once, each time the loop is entered.
- Basic operation - the multiplication operation in the innermost loop.
- Since the number of basic operations depends only on the size of the input matrices and not on the value of the elements, the best-case, average-case and worst-case scenarios are one and the same.

- The number of multiplications made for every pair of specific values of variables i and j is:

$$\sum_{k=0}^{n-1} 1$$

- The total number of multiplications is expressed as follows:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

We do the same number of multiplications all the time.

Hence, overall time complexity is $\Theta(n^3)$.

Time Efficiency of Recursive Algorithms: *General Plan for Analysis*

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Recursive Evaluation of n!

Definition: $n! = 1 * 2 * \ldots * (n\text{-}1) * n$ for $n \geq 1$ and $0! = 1$

- Recursive definition of $n!$: $F(n) = F(n\text{-}1) * n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

$M(0) = 0$

the calls stop when $n = 0$ ← → no multiplications when $n = 0$

M(n-1) = M(n-2) + 1; M(n-2) = M(n-3)+1

M(n) = [M(n-2)+1] + 1 = M(n-2) + 2 = [M(n-3)+1+2] = M(n-3) + 3
= M(n-n) + n = n

Overall time Complexity: Θ(n)

YouTube Link: https://www.youtube.com/watch?v=K25MWuKKYAY

Counting the # Bits of an Integer

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return *BinRec*$(\lfloor n/2 \rfloor) + 1$

 # Divisions $A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$

Since the recursive calls end when *n* is equal to 1 and there are no divisions made, the initial condition is: A(1) = 0.

1	1 bit
2-3	2 bits
4-7	3 bits
8-15	4 bits
16-31	5 bits
32-63	6 bits

Solution Approach: If we use the backward substitution method (as we did in the previous two examples, we will get stuck for values of n that are not powers of 2).
We proceed by setting $n = 2^k$ for $k \geq 0$.

New recurrence relation to solve:

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Counting the # Bits of an Integer

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \cdots \\
&\cdots \\
&= A(2^{k-i}) + i \\
&\cdots \\
&= A(2^{k-k}) + k.
\end{aligned}
$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

Examples for Solving Recurrence Relations

1)

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

$$= \Theta(2^n)$$

2) X(n) = X(n-1) + 5, for n > 1, X(1) = 0

$$
\begin{aligned}
x(n) &= x(n-1)+5 \\
&= [x(n-2)+5]+5 = x(n-2)+5\cdot 2 \\
&= [x(n-3)+5]+5\cdot 2 = x(n-3)+5\cdot 3 \\
&= \dots \\
&= x(n-i)+5\cdot i \\
&= \dots \\
&= x(1)+5\cdot(n-1) = 5(n-1).
\end{aligned}
$$

= Θ(n)

3) X(n) = 3*X(n-1) for n > 1, X(1) = 4

$$
\begin{aligned}
x(n) &= 3x(n-1) \\
&= 3[3x(n-2)] = 3^2x(n-2) \\
&= 3^2[3x(n-3)] = 3^3x(n-3) \\
&= \dots \\
&= 3^ix(n-i) \\
&= \dots \\
&= 3^{n-1}x(1) = 4\cdot 3^{n-1}.
\end{aligned}
$$

= (4/3)3^n = Θ(3^n)

4)

$X(n) = X(n-1) + n \text{ for } n > 0, X(0) = 0$

$$
\begin{aligned}
x(n) &= x(n-1)+n \\
&= [x(n-2)+(n-1)]+n = x(n-2)+(n-1)+n \\
&= [x(n-3)+(n-2)]+(n-1)+n = x(n-3)+(n-2)+(n-1)+n \\
&= \ldots \\
&= x(n-i)+(n-i+1)+(n-i+2)+\cdots+n \\
&= \ldots \\
&= x(0)+1+2+\cdots+n = \frac{n(n+1)}{2}.
\end{aligned}
$$

$X(n) = \Theta(n^2)$

5)

X(n) = X(n/2) + n, for n > 1, X(1) = 1 [Solve for n = 2^k]

$$
\begin{aligned}
x(2^k) &= x(2^{k-1}) + 2^k \\
&= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\
&= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
&= \dots \\
&= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \cdots + 2^k \\
&= \dots \\
&= x(2^{k-k}) + 2^1 + 2^2 + \cdots + 2^k = 1 + 2^1 + 2^2 + \cdots + 2^k \\
&= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1.
\end{aligned}
$$

X(n) = Θ(n)

6)

X(n) = X(n/3) + 1 for n > 1, X(1) = 1 [Solve for n = 3^k]

$$
\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= \dots \\
&= x(3^{k-i}) + i \\
&= \dots \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}
$$

X(n) = Θ(logn)