03
Algorithm Analysis
Chapter 3

Algorithm

- An ***algorithm*** is a finite set of instructions to be followed to solve a problem that is guaranteed to halt.
  - There can be more than one solution (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using different programming languages on different platforms.
- An algorithm must be correct. It should correctly solve the problem.
  - e.g. For sorting, even if the input is already sorted, or it contains repeated elements, it should work as expected.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Algorithm

```
Step 1: Start
Step 2: Declare variables n, factorial and i.
Step 3: Initialize variables
          factorial ← 1
          i ← 1
Step 4: Read value of n
Step 5: Repeat the steps until i = n
     5.1: factorial ← factorial*i
     5.2: i ← i+1
Step 6: Display factorial
Step 7: Stop
```

Shamelessly borrowed from: https://www.programiz.com/dsa/algorithm

Algorithm

There are *two aspects* of algorithmic performance:

- Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- Space
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

➢ We will focus on time:

- How to estimate the time required for an algorithm
- How to reduce the time required

Analysis of Algorithms

- ***Analysis of Algorithms*** is the area of computer science that provides tools to analyze the efficiency of different methods of solutions.
- How do we compare the time efficiency of two algorithms that solve the same problem?

  Naïve Approach: implement these algorithms in a programming language (e.g., Python, Java, C++), and run them to compare their time requirements. Comparing the programs (instead of algorithms) has difficulties.
  - *How are the algorithms coded?*
    - Comparing running times means **comparing the implementations**.
    - We should not compare implementations, because they are sensitive to programming style that may cloud the issue of which algorithm is inherently more efficient.
  - *What computer should we use?*
    - We should compare the efficiency of the algorithms independently of a particular **computer**.
  - *What data should the program use?*
    - Any analysis must be independent of specific data.

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using **growth functions.**

The Execution Time of Algorithms

Each operation in an algorithm (or a program) has a cost.

Each operation takes a certain amount of time.

`count = count + 1` take a certain amount of time, it is constant.

A sequence of operations:

```
count = count + 1
sum = sum + count
```

`count = count + 1`	Cost: c_1
`sum = sum + count`	Cost: c_2

Total Cost = c1 + c2

The Execution Time of Algorithms

Example: Simple If-Statement

	Cost	Times
`if n < 0:`	c1	1
`    absval = -n`	c2	1
`else:`		
`    absval = n`	c3	1

Total Cost <= c1 + max(c2,c3)

The Execution Time of Algorithms

Example: Simple Loop

Code	Cost	Times
`i = 1`	c1	1
`sum = 0`	c2	1
`while (i <= n):`	c3	n+1
`    i = i + 1`	c4	n
`    sum = sum + i`	c5	n

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

The time required for this algorithm is proportional to **n**.

The Execution Time of Algorithms

Example: Nested Loop

Code	Cost	Times
`i=1`	c1	1
`sum = 0`	c2	1
`while (i <= n):`	c3	n+1
`    j=1`	c4	n
`    while (j <= n):`	c5	n*(n+1)
`        sum = sum + i`	c6	n*n
`        j = j + 1`	c7	n*n
`    i = i + 1`	c8	n

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

The time required for this algorithm is proportional to $\mathbf{n^2}$.

General Rules for Estimation

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: (The running time of the inner loop + other statements in the outer loop) * number of iterations in the outer loop.
- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else**: Never more than the running time of the test plus the larger of running times of "if" and "else" parts.

Algorithm Growth Rates

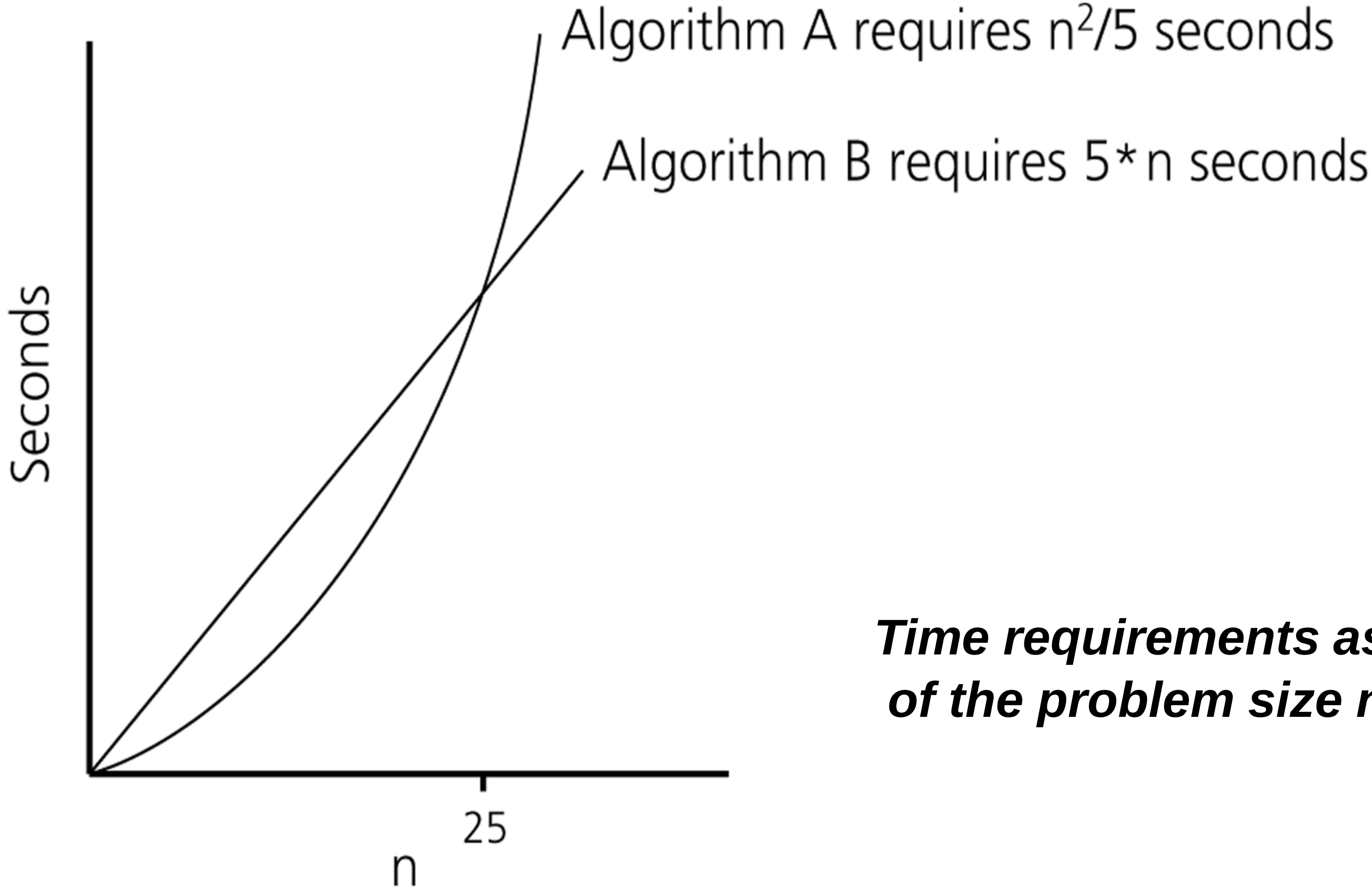
- We measure an algorithm's time requirement as a function of the *problem size*.
  - e.g. number of elements in a list for a sorting algorithm, the number users for a social network search.

- So, for instance, we say that (if the problem size is n)
  - For problem of size **n**, Algorithm A requires $\mathbf{5*n^2}$ time units.
  - For problem of size **n**, Algorithm B requires $\mathbf{7*n^2}$ time units.

Algorithm Growth Rates

- The most important thing to learn is **how quickly** the algorithm's time requirement **grows** as a function of the problem size.
  - Algorithm A requires time proportional to $\mathbf{n^2}$.
  - Algorithm B requires time proportional to **n**.
- An algorithm's proportional time requirement is known as ***growth rate***.
- We can compare efficiencies of two algorithms by comparing their growth rates.

Algorithm Growth Rates

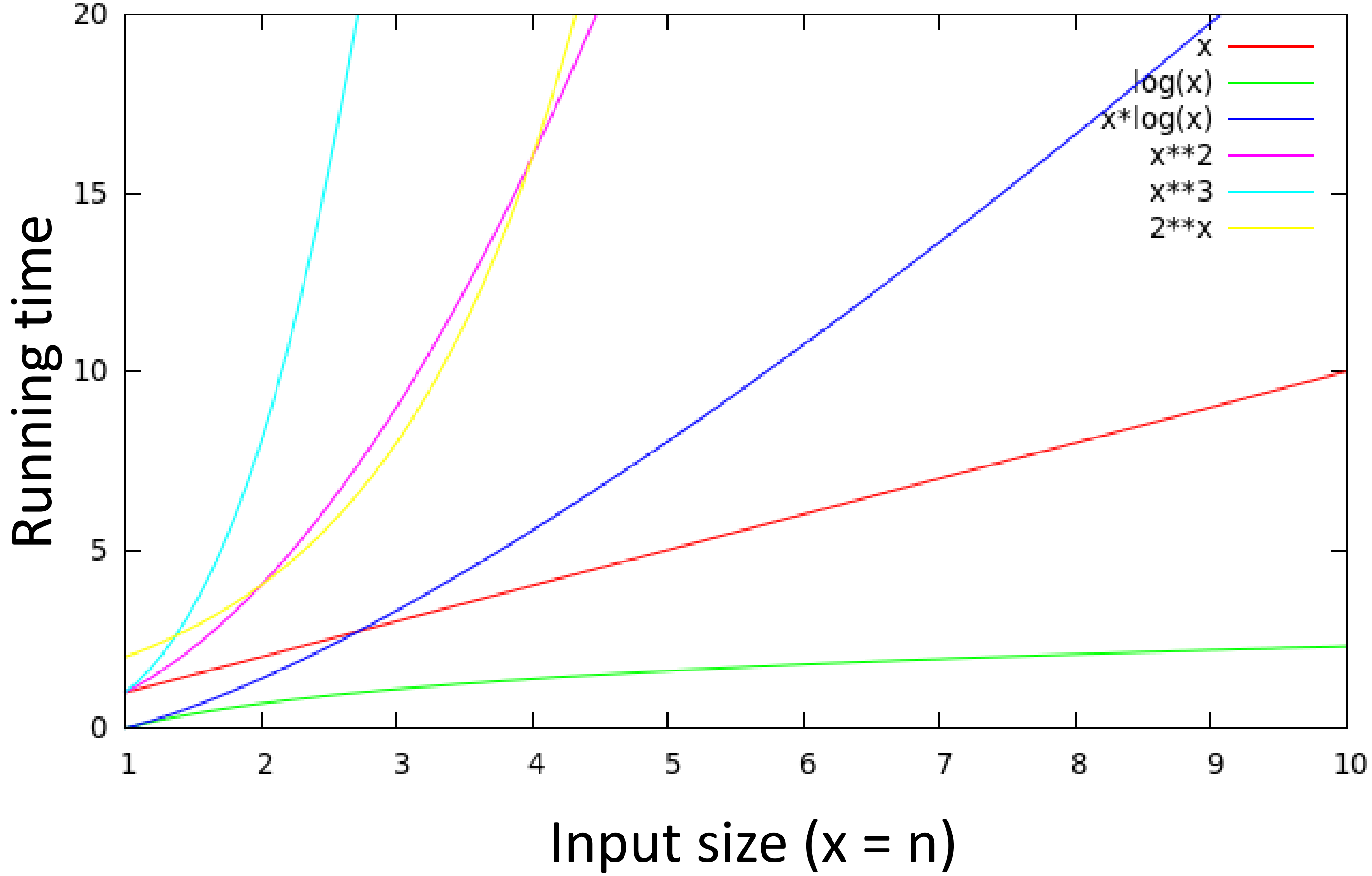


Time requirements as a function of the problem size n

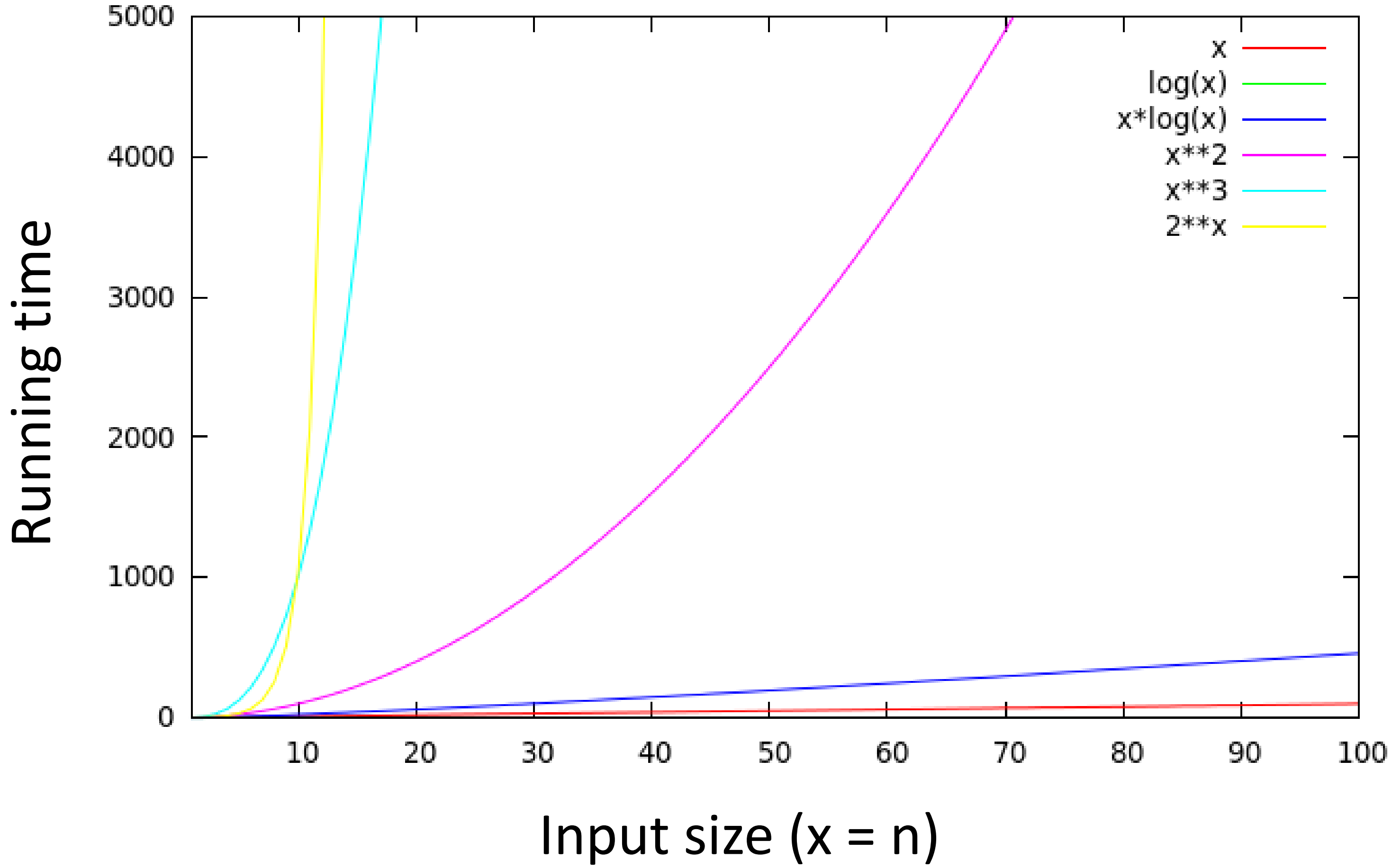
Common Growth Rates

Function	**Growth Rate Name**
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	Log-linear
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Running Times for Small Inputs



Running Times for Large Inputs



Big O Notation

If *Algorithm A requires time proportional to g(n),* Algorithm A is said to be in **order g(n),** and it is denoted as **O(g(n)).**

The **function g(n)** is called the algorithm's **growth-rate function**.

Since the capital O is used in the notation, this notation is called the **Big O notation**.

- If Algorithm A requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$.
- If Algorithm A requires time proportional to **n**, it is **O(n).**

Big O Notation (upper bound)

Definition:

Algorithm A is said to be in order g(n) (denoted as O(g(n))) if constants k and n_0 exist such that A requires no more than k*g(n) time units to solve a problem of size $n \geq n_0$. ➔ $f(n) \leq k*g(n)$ for all $n \geq n_0$

- The requirement of $\mathbf{n \geq n_0}$ in the definition of O(f(n)) formalizes the notion of *sufficiently large problems*.
  - In general, many values of k and n can satisfy this definition.

Order of an Algorithm

- If an algorithm requires $f(n) = n^2 - 3*n+10$ seconds to solve a problem of size n and if constants k and n_0 exist such that

  $k*n^2 > n^2 - 3*n+10$ for all $n \geq n_0$,

  then we can state that the algorithm is order n^2

For k = 3 and n_0 = 2, we have

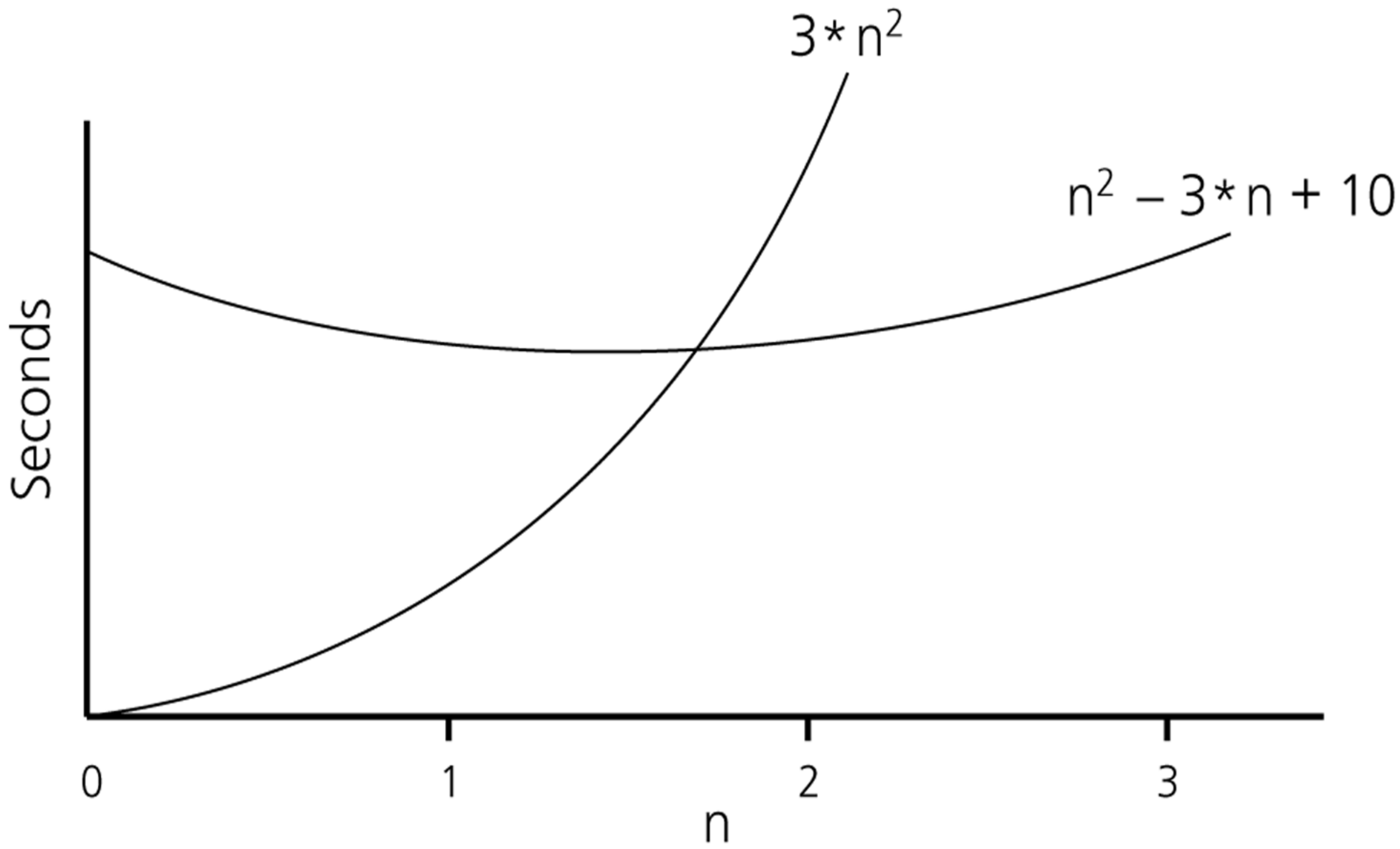
$3*n^2 > n^2 - 3*n+10$ for all $n \geq 2$.

Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$,

So it is **O(n²)**.

O-notation provides an upper bound.

Order of an Algorithm



Small O Notation (strict upper bound)

Definition:

Algorithm A is said to be in order g(n) (denoted as o(g(n))) if constants k and n_0 exist such that A requires no more than k*g(n) time units to solve a problem of size $n \geq n_0$. → $f(n) < k*g(n)$ for all $n \geq n_0$

Order of an Algorithm

- Show $2^x + 17$ is $\mathbf{O(2^x)}$

- $2^x + \boldsymbol{17} \leq 2^x + \boldsymbol{2^x} = 2*2^x$ for $\boldsymbol{x \geq 5}$
- Hence $k = 2$ and $n_0 = 5$

Order of an Algorithm

- Show $2^x + 17$ is **$O(3^x)$**

- $2^x + \boldsymbol{17} \leq k3^x$
- Easy to see that rhs grows faster than lhs over time → k=1
- However when x is small 17 will still dominate → skip over some smaller values of x by using $n_0 = 3$
- Hence $k = 1$ and $n_0 = 3$

Omega Notation (lower bound)

Definition:

Algorithm A is said to be omega g(n) (denoted as $\Omega(g(n))$) if constants k and n_0 exist such that A requires more than k*g(n) time units to solve a problem of size $n \geq n_0$. ➔ $f(n) \geq k*g(n)$ for all $n \geq n_0$

Omega notation provides a lower bound.

Small Omega Notation (strict lower bound)

Definition:

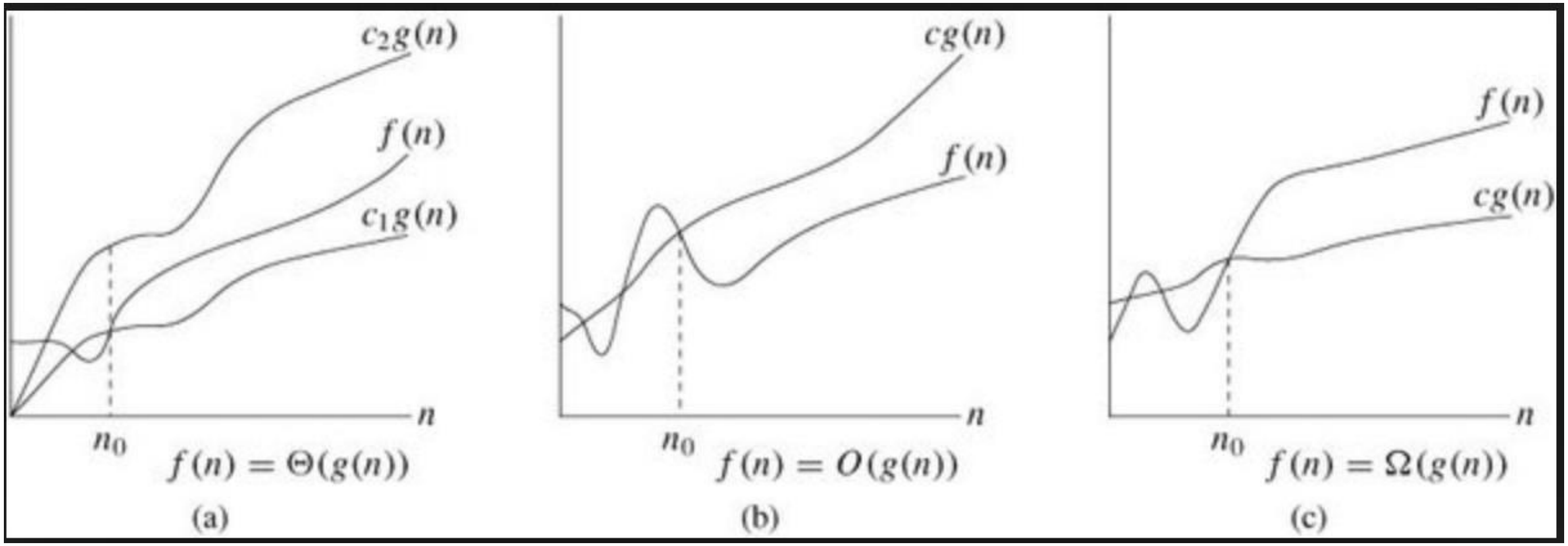
Algorithm A is said to be omega g(n) (denoted as ω (g(n))) if constants k and n_0 exist such that A requires more than k*g(n) time units to solve a problem of size $n \geq n_0$. ➔ $f(n) > k*g(n)$ for all $n \geq n_0$

Theta Notation

Definition:

Algorithm A is said to be theta g(n) (denoted as θ(g(n))) if constants k_1, k_2, and n_0 exist such that

$k_1 * g(n) \leq f(n) \leq k_2 * g(n)$ for all $n \geq n_0$

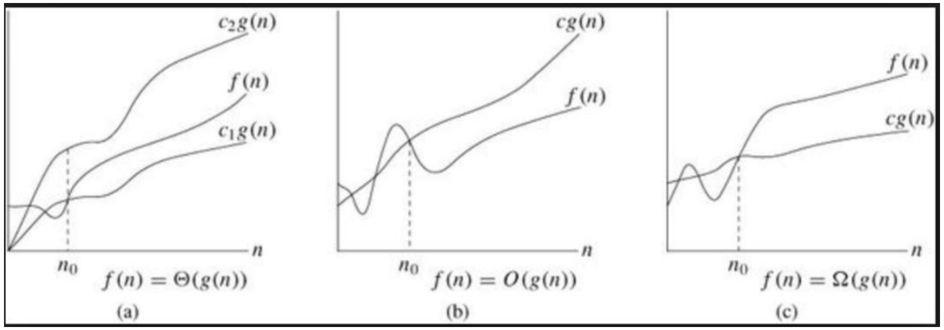


ODTÜ METU

Order of an Algorithm

- Show $f(n) = 7n^2 + 1$ is $\Theta(n^2)$

- You need to show f(n) is $O(n^2)$ and f(n) is $\Omega(n^2)$
- f(n) is $O(n^2)$ because $7n^2 + 1 \le 7n^2 + n^2 \; \forall n \ge 1$ ➔ $k_1 = 8$ $n_0 = 1$
- f(n) is Ω (n^2) because $7n^2 + 1 \ge 7n^2 \; \forall n \ge 0$ ➔ $k_2 = 7$ $n_0 = 0$
- Pick the largest n_0 to satisfy both conditions naturally ➔ $k_1 = 8$, $k_2 = 7$, $n_0 = 1$



A Comparison of Growth-Rate Functions

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Growth-Rate Functions

O(1) Time requirement is **constant**, and it is independent of the problem's size.

$O(\log_2 n)$ Time requirement for a **logarithmic** algorithm increases increases slowly as the problem size increases.

O(n) Time requirement for a **linear** algorithm increases directly with the size of the problem.

$O(n*\log_2 n)$ Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

$O(n^2)$ Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

$O(n^3)$ Time requirement for a c**ubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

$O(2^n)$ As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

$O(1)$	$T(n) = 1$ second
$O(\log_2 n)$	$T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds
$O(n)$	$T(n) = (1*16) / 8 = 2$ seconds
$O(n*\log_2 n)$	$T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds
$O(n^2)$	$T(n) = (1*16^2) / 8^2 = 4$ seconds
$O(n^3)$	$T(n) = (1*16^3) / 8^3 = 8$ seconds
$O(2^n)$	$T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

Properties of Growth-Rate Functions

We can ignore low-order terms in an algorithm's growth-rate function.

- If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
- We only use the higher-order term as algorithm's growth-rate function.

We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.

- If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

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

- We can combine growth-rate functions.
- If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ So, it is $O(n^3)$.
- Similar rules hold for multiplication.

Properties of Growth-Rate Functions

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

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

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

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

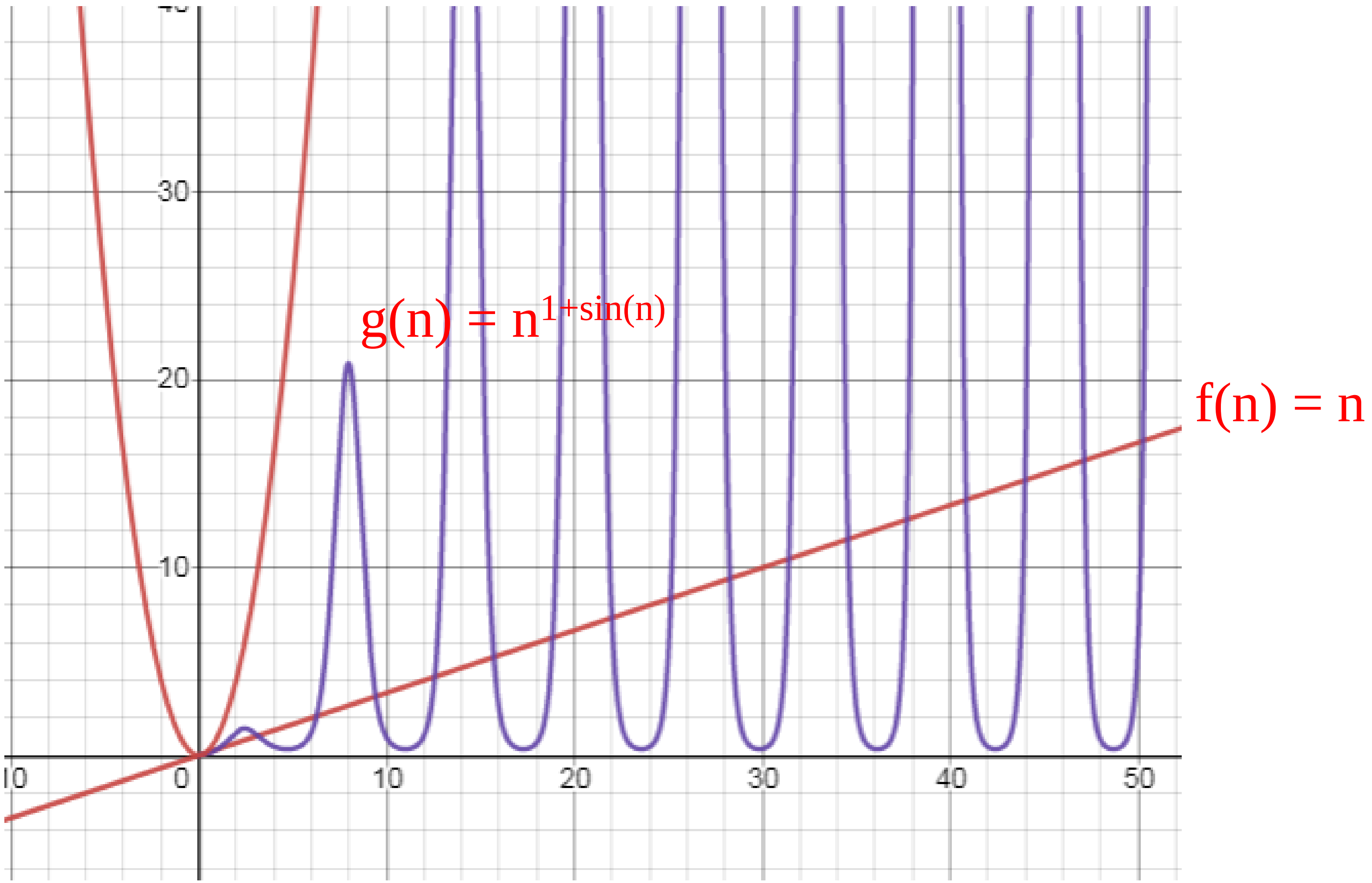
$$f(n) = \Theta(f(n)) \qquad f(n) = O(f(n)) \qquad f(n) = \Omega(f(n))$$

Properties of Growth-Rate Functions

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ implies } f(n) = \Theta(h(n))$$

This still holds when Θ above is replaced by any other asymptotic notation we have introduced (O,Ω,o,ω).

Not All Functions Comparable



ODTÜ METU

Growth-Rate Functions (Example-1)

Code	Cost	Times
`i = 1`	c1	1
`sum = 0`	c2	1
`while (i <= n):`	c3	n+1
`    i = i + 1`	c4	n
`    sum = sum + i`	c5	n

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*c5 \\
&= (c3+c4+c5)*n + (c1+c2+c3) \\
&= a*n + b
\end{aligned}
$$

So, the growth-rate function for this algorithm is O(n)

Growth-Rate Functions (Example-2)

```
for i in range(n):                         n+1
    for j in range(n):                     n*(n+1)
        for k in range(m):                 n*n*(m+1)
            x = x + 1                      n*n*m
```

$T(n) = n+1+n*(n+1)+n*n*(m+1)+n*n*m$

$\quad = 2n^2m+2n^2+2n+1$

For $k=3$, $n_0=3$, and $m_0=3$,

$kn^2m >= 2n^2m+2n^2+2n+1$

Find k, n_0, and m_0 such that for all $n \geq n_0$ and $m \geq m_0$, $kn^2m \geq 2n^2m+2n^2+2n+1$

So, the growth-rate function for this algorithm is **$O(n^2m)$.**

Growth-Rate Functions (Example-3)

Code	Cost	Times
`i=1`	c1	1
`sum = 0`	c2	1
`while (i <= n):`	c3	n+1
`    j=1`	c4	n
`    while (j <= n):`	c5	n*(n+1)
`        sum = sum + i`	c6	n*n
`        j = j + 1`	c7	n*n
`   i = i + 1`	c8	n

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8 \\
&= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3) \\
&= a*n^2 + b*n + c
\end{aligned}
$$

So, the growth-rate function for this algorithm is $O(n^2)$

Growth-Rate Functions (Example-4)

- For nxn matrices, time complexity is: $O(n^3)$

Matrix Multiplication (naïve):

```
for(int i = 0; i < m.length; i++) {
  for(int j = 0; j < m2.length - 1; j++) {
    for(int k = 0; k < m2.length; k++){
      m[i][j] += m[i][k] * m2[k][j];
    }
  }
}
```

- How about this one? $O(\log n)$

```
for (int j = 1; j < n; j = 2 * j)
  sum += j;
```

Growth-Rate Functions (Example-5)

- Fill in the boxes.

$O \quad \Omega \quad \Theta \quad o \quad \omega \quad \sim$

$$6n^2 + 7n - 10 = \boxed{O, \Omega, \Theta}\,(n^2)$$

$O \quad \Omega \quad \Theta \quad o \quad \omega \quad \sim$

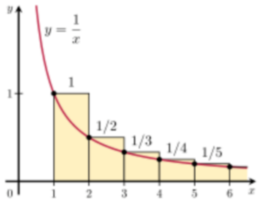
$$6^n = \boxed{\Omega, \omega}\,(n^6)$$

$O \quad \Omega \quad \Theta \quad o \quad \omega \quad \sim$

$$n! = \boxed{O, o}\,(n^n)$$

$O \quad \Omega \quad \Theta \quad o \quad \omega \quad \sim$

$$\sum_{j=1}^{n} \frac{1}{j} = \boxed{O, \Omega, \Theta, \sim}\,(\ln n)$$



$O \quad \Omega \quad \Theta \quad o \quad \omega \quad \sim$

$$\ln(n^3) = \boxed{O, \Omega, \Theta}\,(\ln n)$$