Fundamental of Computer Science
CS1FC16: Lecture 01

Analysis of Algorithm

Dr Varun Ojha

Department of Computer Science

University of Reading

Learning Objectives

On completion of three parts of this lecture, you will be able to

- Understand algorithm and algorithm’s properties
- Understand how to analyses algorithm in terms of complexity.
- Evaluate the “rate of growth” of standard functions
- Apply knowledge of asymptotic notation to solve complexity expression
- Create a program to plot standard functions

Content of this lecture

- Part I: Introduction
  - Definition Algorithm
  - Properties of Algorithm
  - Complexity
    - Space complexity
    - Time complexity
  - Rate of Growth
- Part – II: Asymptotic Notations
  - Types of Algorithm analysis
  - Asymptotic notation
- Part –III: Examples and Exercise
  - Asymptotic notation examples and proofs
  - Exercise

Fundamental of Computer Science
CS1FC16: Lecture 01, Part – I

Algorithm

Dr Varun Ojha
Department of Computer Science
University of Reading

An algorithm is a finite sequence of instructions, each of which has a clear meaning (is unambiguous) and can be performed with a finite amount of effort in a finite length of time.

Properties of Algorithm

- **Input**: Zero, One or more inputs.
- **Output**: At least one output.
- **Finiteness**: An algorithm should be “finite” (i.e., there MUST NOT be an infinite loop, algorithm MUST terminate)
- **Effectiveness**: Instructions are realised (doable), i.e., they can be performed in a finite amount of time.
- **Definiteness**: Instructions and sequence of instruction clearly defined, i.e., no ambiguities in the instructions and every aspect of performing instruction MUST be specified.

Why study Algorithm

- How to write/create an algorithm.
- How to express an algorithm.
- How to validate an algorithm.
- **How to analyse an algorithm.**
- How to test a program.

How to analyse an algorithm?

Algorithms can be analysed by evaluating the **rate of growth** of **time** or **space** required to solve a problem of size ***n***, which is a measure of the quantity of input data.

- The **time** required by an algorithm expressed as a function of the problem size, ***n*** is called the **time complexity** of the algorithm.
- We also define **space complexity** as a function of problem size ***n***.

Complexity

The complexity of an algorithm $\boldsymbol{A}$ is the function $\boldsymbol{f(n)}$ which gives time and space requirement of the algorithm for input data size $\boldsymbol{n}$.

- Space Complexity
- **Time Complexity**

Random access memory hardware is relatively least expensive and easily manageable these days. Hence, we are more interested in Time Complexity these days.

Space Complexity

The amount of memory space an algorithm needs

Ex.: Algo Sum(A; n)

```
// A is an array of size n
{
 S := 0.0
 for i := 1 to n do
      S := S + A[i]
 return S
}
```

Total space required for Algo Sum is:

```
A     → n words
S     → 1 word
i     → 1 word
n     → 1 word
Total → (n + 3) words
```

Total space required is (n+3) words – 3 remain constant and the ***rate of change*** is dependent on ***n***. Therefore, we are interested in *space complexity* as a function of *n.*

Time Complexity

Time spent by an algorithm to produce one or more output

- **Theoretical analysis**
  - We are interested in evaluating algorithm's *time complexity* in terms of "limiting behaviour" of the complexity as the "size of problem" n increases is called the *asymptotic time complexity*.
- Empirical analysis
  - We are interested in evaluating average wall-clock time an algorithm takes to execute a problem of size n.

Asymptotic Time Complexity

Important Considerations:

- Consider that **one** operation takes **1 unit** of time
- Consider that for a statement `x ← x + y` takes **1 unit** of time

```
x ← x + y
```

```
for i := 1 to n
    x ← x + y
```

```
for i := 1 to n
    for j := 1 to n
        x ← x + y
```

$\mathbf{1}$ unit

$\boldsymbol{n}$ unit

$\boldsymbol{n^2}$ unit

Rate of Growth

(of Standard Functions)

Suppose $\boldsymbol{A}$ is an Algorithm, and $\boldsymbol{n}$ is the size of the input data. Then, the complexity $\boldsymbol{f(n)}$ of $\boldsymbol{A}$ increases proportional to the size of $\boldsymbol{n}$.

It is usually the ***rate of increase*** of $\boldsymbol{f(n)}$ that we want to examine, i.e., we compute $\boldsymbol{f(n)}$ with some standard function, such as

$$\log n, \qquad n, \qquad n\log n, \qquad n^2, \qquad n^3, \qquad 2^n$$

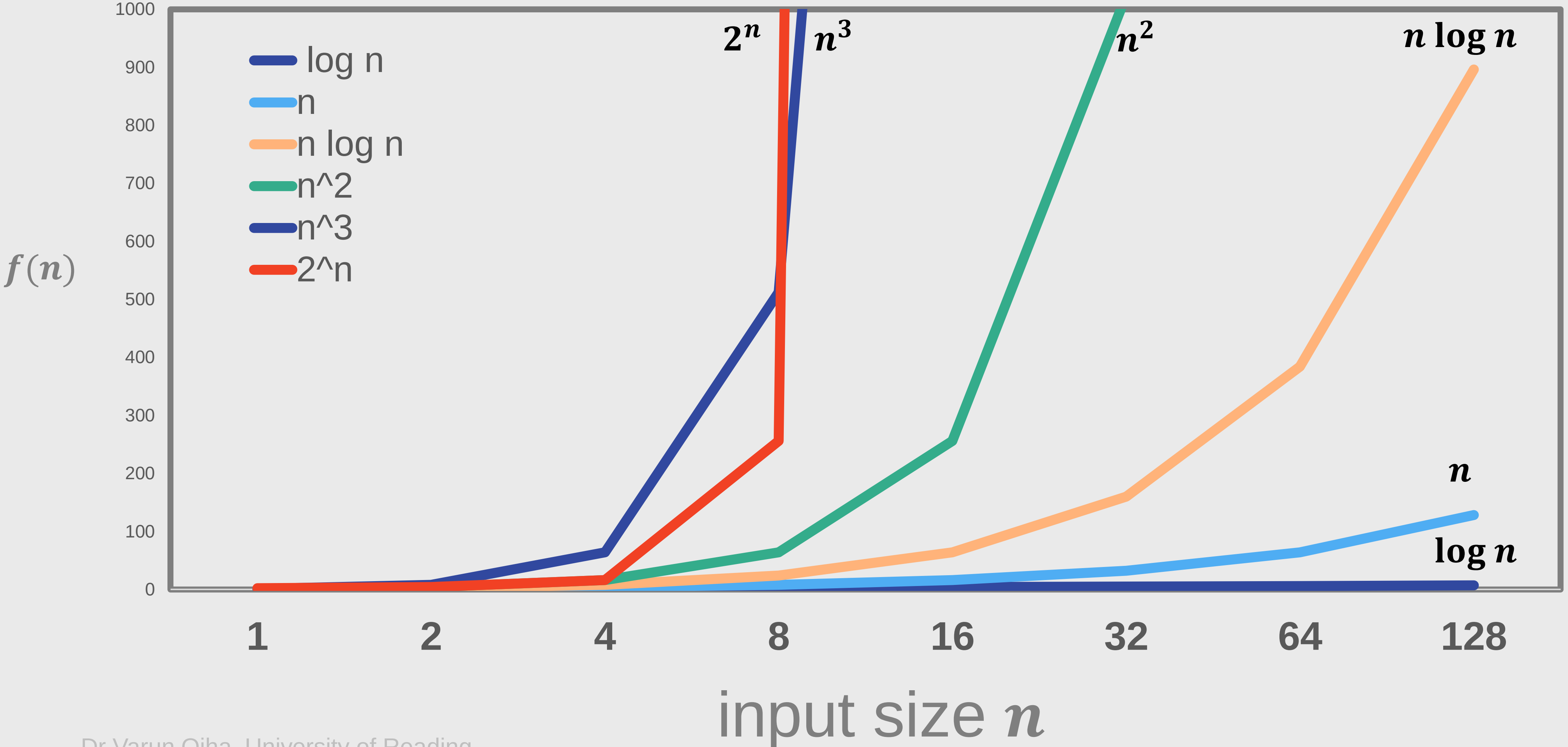
Rate of Growth

Input	Standard functions					
n	$\log n$	n	$n \log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

* values in table are ceiling (nearest upper end integer value)

Rate of Growth

Exercise: write an algorithm to produce this chart



Rate of Growth $\rightarrow$ Algorithm's efficiency

We are interested in the algorithm's behaviours over a large input data size. We call it algorithm's ***asymptotic*** efficiency

Fundamental of Computer Science
CS1FC16: Lecture 01, Part – II

Asymptotic Notation

Dr Varun Ojha
Department of Computer Science
University of Reading

Types of Algorithm Analysis

- **Worst case**
  - Provides a **maximum value** of $f(n)$ for any possible input
  - Provides an upper bound on running time
  - Provides an absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- **Best case**
  - Provides a **minimum value** of $f(n)$ for any possible input
  - Provides a lower bound on running time
  - Answers that for a particular input the algorithm runs the fastest
- **Average case**
  - Provides an **expected** value of $f(n)$
  - Provides a prediction about the running time
  - Assumes that the input is random

Asymptotic Notations

Mathematical notions for analysing ***asymptotic running time complexity*** $\boldsymbol{f(n)}$ of an algorithm based on input size $\boldsymbol{n}$ and a given set of functions $\boldsymbol{g(n)}$ are:

- $\boldsymbol{O}$ **notation:** asymptotic less than

  $\boldsymbol{f(n) = O(g(n))}$ implies: $f(n) \leq g(n)$ (asymptotic upper bound)

- $\boldsymbol{\Omega}$ **notation:** asymptotic greater than

  $\boldsymbol{f(n) = \Omega(g(n))}$ implies: $f(n) \geq g(n)$ (asymptotic lower bound)

- $\boldsymbol{\Theta}$ **notation:** asymptotic equality

  $\boldsymbol{f(n) = \Theta(g(n))}$ implies: $f(n) \approx g(n)$ (asymptotic tight bound)

O notation

(Big Oh and Little Oh - asymptotic upper bound)

Let $\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ be functions that map positive integers to positive real numbers, then we define:

- **Big-Oh, $\boldsymbol{O}(\cdot)$:**

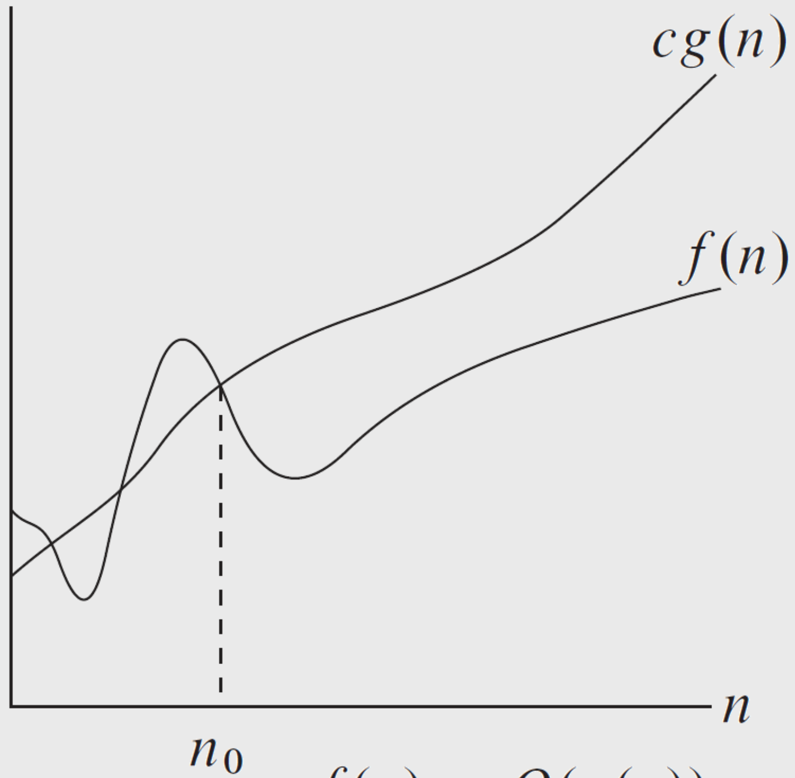
  We say that $f(n)$ is $O(g(n))$ [or $f(n) \in O(g(n))$] if there exists a real constant $c > 0$
  and there exists an integer constant $n_0 \geq 1$
  such that $0 \leq f(n) \leq c \cdot g(n)$ for every integer $n \geq n_0$.

- **Little-Oh, $\boldsymbol{o}(\cdot)$:**

  We say that $f(n)$ is $O(g(n))$ [or $f(n) \in O(g(n))$] if there exists a real constant $c > 0$
  and there exists an integer constant $n_0 \geq 1$
  such that $0 \leq f(n) < c \cdot g(n)$ for every integer $n \geq n_0$.

O notation

$\boldsymbol{O(g(n)) = \{ f(n)|}$ if there exists a real constant $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that $\boldsymbol{0 \leq f(n) \leq c \cdot g(n)}$ for every integer $\boldsymbol{n \geq n_0}$



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

$g(n)$ is an ***asymptotic upper bound*** of $f(n)$

Ω notation

(Big Omega and Little omega - asymptotic lower bound)

Let $\boldsymbol{f}(\boldsymbol{n})$ and $\boldsymbol{g}(\boldsymbol{n})$ be functions that map positive integers to positive real numbers, then we define:

- **Big-Omega, $\boldsymbol{\Omega}(\cdot)$:**

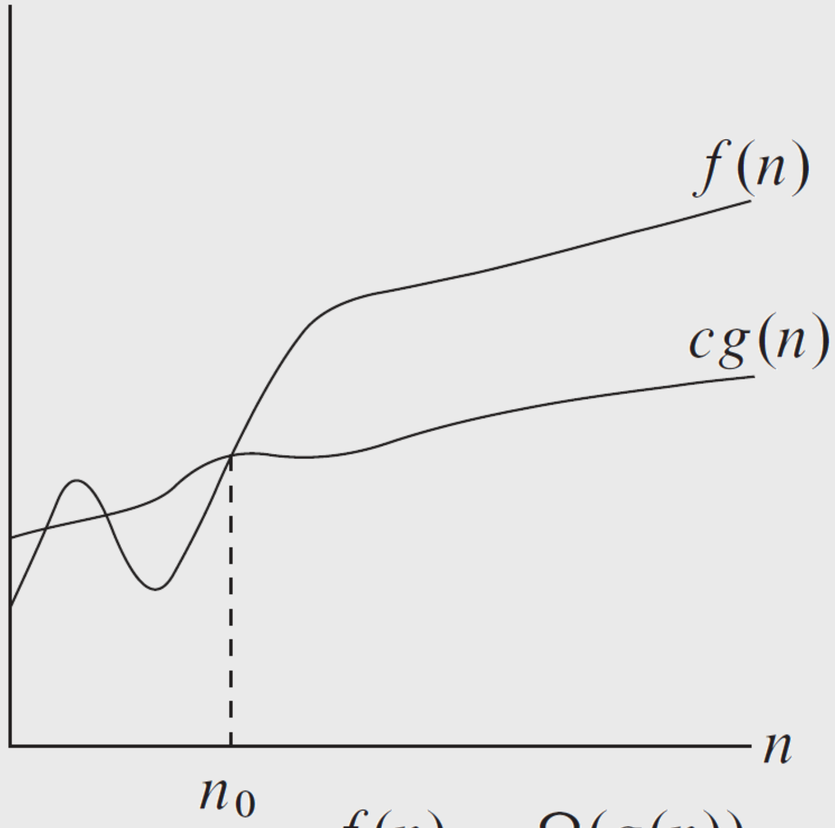
  We say that $f(n)$ is $\Omega(g(n))$ [or $f(n) \in \Omega(g(n))$] if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $0 \leq c \cdot g(n) \leq f(n)$ for every integer $n \geq n_0$.

- **Little-Omega, $\boldsymbol{\omega}(\cdot)$:**

  We say that $f(n)$ is $\omega(g(n))$ [or $f(n) \in \omega(g(n))$] if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $0 \leq c \cdot g(n) < f(n)$ for every integer $n \geq n_0$.

Ω notation

$\boldsymbol{\Omega(g(n)) = \{\, f(n)|}$ if there exists a real constant $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that $\boldsymbol{0 \le c \cdot g(n) \le f(n)}$ for every integer $\boldsymbol{n \ge n_0}$



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

$g(n)$ is an ***asymptotic lower bound*** of $f(n)$

Θ notation
(Theta - asymptotic tight bound)

Let $\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ be functions that map positive integers to positive real numbers.

- **Θ, Θ(·):**

  We say that $f(n)$ is $O(g(n))$ [or $f(n) \in \Theta(g(n))$] if there exists a real constant $c > 0$

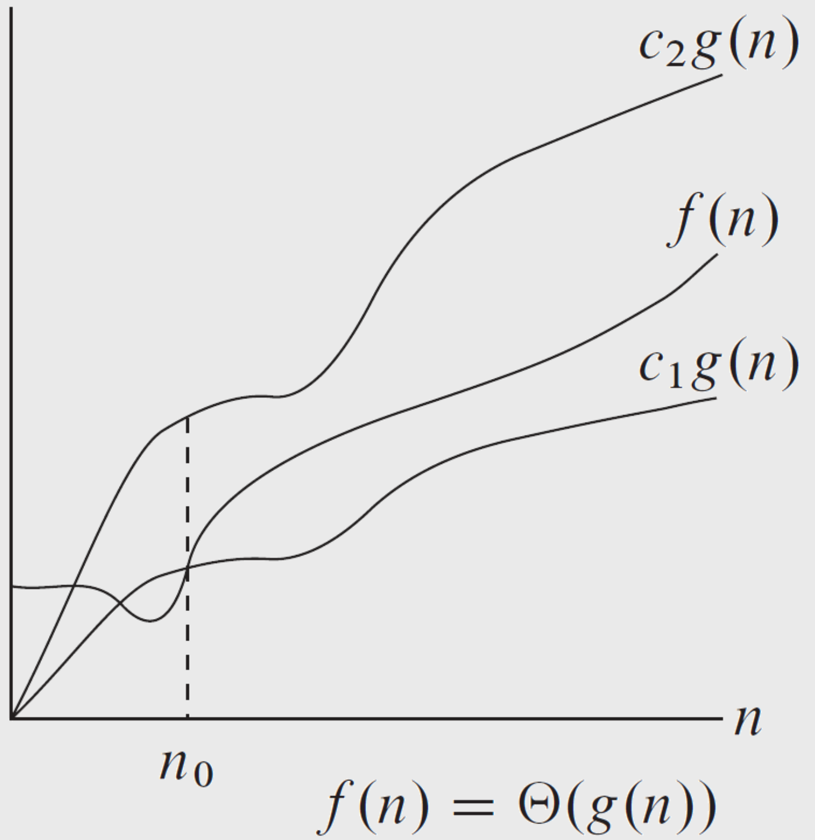
  and there exists an integer constant $n_0 \geq 1$

  such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for every integer $n \geq n_0$.

Θ notation

$\boldsymbol{\Theta(g(n)) = \{ f(n)|}$ if there exists a real constant $\boldsymbol{c}$ and $\boldsymbol{n_0}$such that

$\boldsymbol{0 \leq c_1 \cdot g(n) \leq f(n) \leq \ c_2 \cdot g(n)}$ for every integer $\boldsymbol{n \geq \ n_0}$



$g(n)$ is an ***asymptotic tight bound*** of $f(n)$

Fundamental of Computer Science
CS1FC16: Lecture 01, Part – III

Asymptotic Notation Examples and Exercises

Dr Varun Ojha

Department of Computer Science

University of Reading

Example: Big-Oh Notation

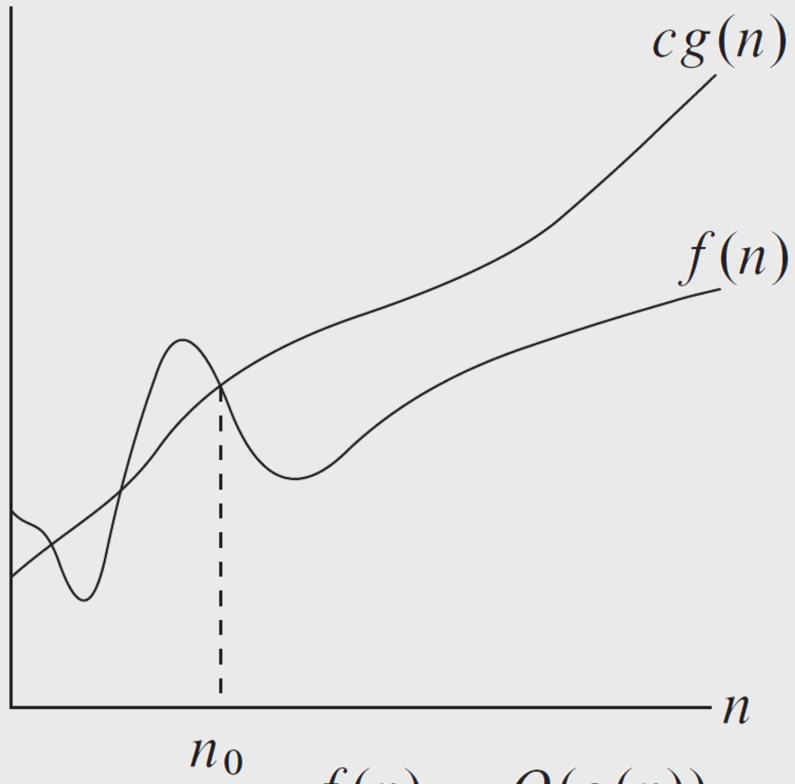
Let $f(n) = 7n + 8$ and $g(n) = n$

Is $\boldsymbol{f(n) \in O(g(n))}$?

Does $f(n)$ belong to $O(g(n))$

O notation (revisit)

$\boldsymbol{O(g(n)) = \{ f(n) |}$ if there exists a real constant $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that $\boldsymbol{0 \leq f(n) \leq c \cdot g(n)}$ for every integer $\boldsymbol{n \geq n_0}$



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

$g(n)$ is an ***asymptotic upper bound*** of $f(n)$

Example: Big-Oh Notation

If $\boldsymbol{f(n) = 7n + 8}$ and $\boldsymbol{g(n) = n}$, is $\boldsymbol{f(n) \in O(g(n))?}$

For $7n + 8 \in O(n)$, we have to find c and n_0 such that $7n + 8 \leq cn,$ for all $n \geq n_0$.

By inspection, it is clear that c must be larger than 7. Let $c = 8$.

Now we need a suitable $n \geq n_0$. Let $n_0 = n = 8$.

In this case, $f(8) = 8 \cdot g(8)$. Because the definition of $O(\cdot)$ requires that $f(n) \leq c \cdot g(n),$ we can select $n_0 = 8$, or any integer above 8, they will all work.

Since we are able to find constants c and n_0 such that $7n + 8$ is $\leq cn$ for every $n \geq n_0$, we can say that $7n + 8\ is\ O(n)$, alternatively $f(n) \in O(g(n))$?.

Q: But how do we know that this will work for every $\boldsymbol{n}$ above 7?

A: We can prove it by induction that $\boldsymbol{7n + 8\ \ 8n}$, $\forall\, \boldsymbol{n \geq 8}$.

Mathematical Induction
Proof of $7n + 8 \leq 8n, \; \forall\, n \geq 8$

Basic Step:

for $n_0 = n = 8,$

$7 \cdot 8 + 8 \leq 64 \qquad (1) \rightarrow$ TRUE

Let $n = k$

$7k + 8 \leq 8k \qquad (2) \rightarrow$ TRUE

Inductive Step:

for $n = k + 1$

- $7(k + 1) + 8 \leq 8(k + 1)$
- $7k + 7 + 8 \leq 8k + 8$
- $(7k + 8) + 7 \leq 8k + 8$
- From (2), we know $7k + 8 \leq 8k$
- $\mathbf{8k + 7 \leq 8k + 8} \quad (3) \rightarrow$ TRUE

Hence, it is proved that $7n + 8 \leq 8n, \; \forall\, n \geq 8$

Example: Big-Omega Notation

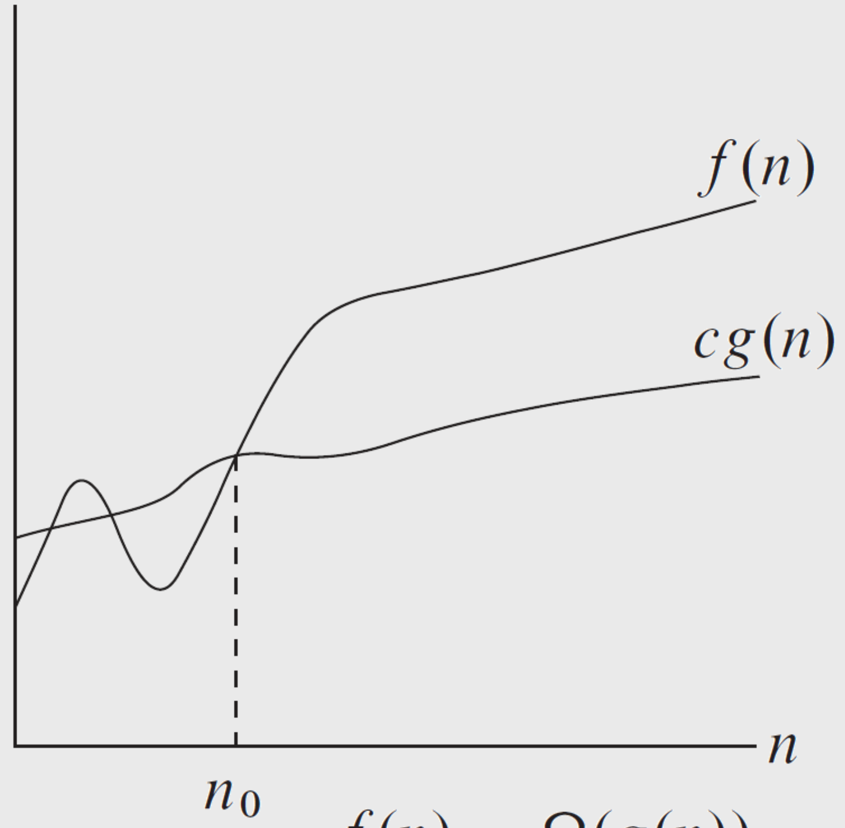
Let $f(n) = \sqrt{n}$ and $g(n) = \log n$

Is $\boldsymbol{f(n) \in \Omega(g(n))}$?

Does $f(n)$ belong to $\Omega(g(n))$

Ω notation (revisit)

$\boldsymbol{\Omega\big(g(n)\big) = \{\, f(n)}|$ if there exists a real constant $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that $\boldsymbol{0 \ \le c \cdot g(n) \ \le f(n)}$ for every integer $\boldsymbol{n \ge \ n_0}$



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

$g(n)$ is an ***asymptotic lower bound*** of $f(n)$

Proof $\sqrt{n} = \Omega(log\, n)$

For $c = 1, n_0 = 16, \quad f(n) = \sqrt{n},$ and $g(n) = \log_2 n$

By definition, we have $f(n) \geq c \cdot (g(n)), \quad \forall\, n \geq n0,$

Replacing values of $c, \quad n0, \quad f(n),$ and $g(n)$ in definition, we get:

$\sqrt{16} \geq 1 \cdot \log_2 16 \rightarrow 4 \geq 1 \cdot 4$ (1)

This gives us $4 \geq 4 \rightarrow$ TRUE

Now check $n = 64,$ i.e., for $n \geq n0$

$\sqrt{64} \geq 1 \cdot \log_2 64 \rightarrow 8 \geq 1 \cdot \log_2 64$ (2)

This gives us $8 \geq 6 \rightarrow$ TRUE

Hence, we get $f(n) \geq c(g(n)),$ i.e., $\sqrt{n} = \Omega(\log n) \rightarrow$ TRUE

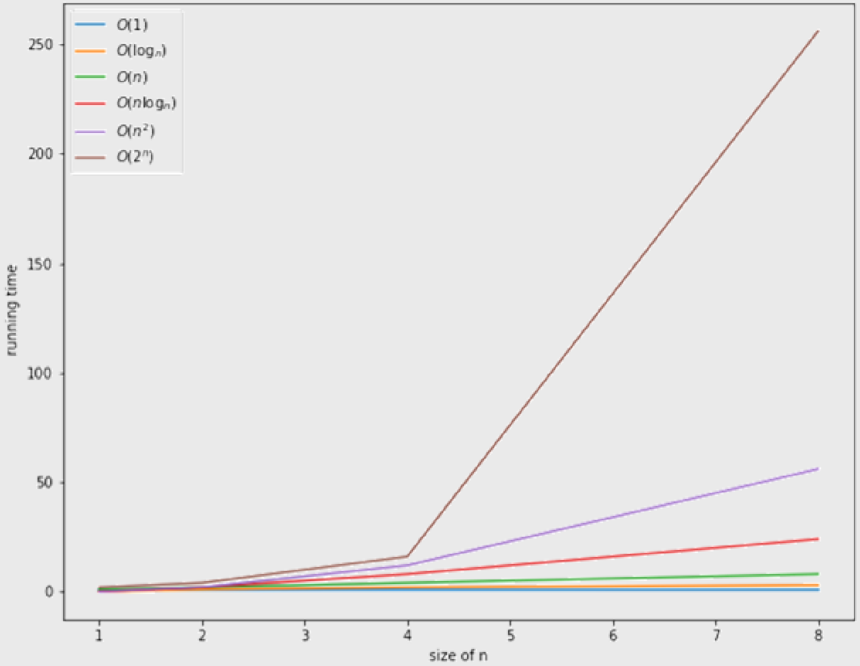
Exercise

Verify and prove:

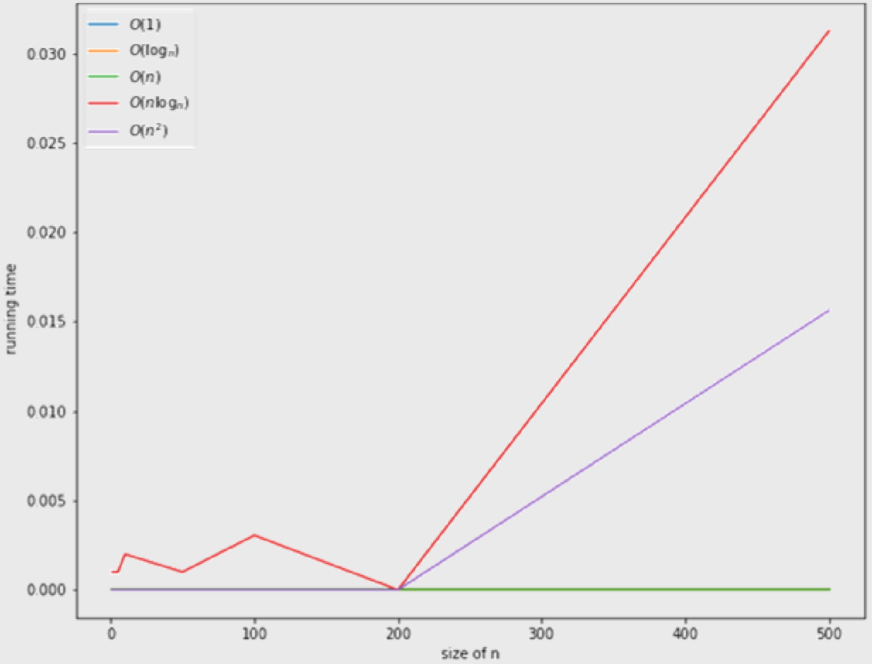
- **Is $2^{n+1} \in \boldsymbol{O}(2^n)$?**
- **Is $2^{n+1} \in \boldsymbol{O}(2^{2n})$?**

Exercise

Write a program to plot theoretical and empirical time of standard functions $O(1), O(\sqrt{n}), O(\log n), O(n), O(n \log n)$, and $O(n^2)$ like the following:



theoretical



empirical