# Asymptotic Analysis

S. Halim  YJ. Chang

School of Computing
National University of Singapore

CS3230 Lec01b; Tue, 13 Aug 2024

# Overview

Introduction

Problem-Solving Example: Fibonacci

Model of Computation: RAM

Asymptotic Analysis

- Big O (upper bound)
- $\Omega$ (lower bound)
- New notation $\Theta$ (tight bound)
- Little-o and $\omega$
- Taking Limits

# Algorithm

## Algorithm

"A sequence of unambiguous and executable instructions for solving a problem (given a valid input, obtain a valid output)"

Let's elaborate:

- What are the valid inputs?
- What is the meaning of unambiguous instructions?
- What is the meaning of executable instructions?
- Are all algorithms deterministic?
- Are all algorithms terminate?

# Pseudocode

We can give an algorithm already written in a particular programming language, pros and cons:

- Unambiguous[1]
- Clear
- Quite tedious
- Harder to understand

Alternative: Pseudocode (we will use this going forward)

- Slightly informal
- Still precise enough to understand exactly what instructions are, and how to implement it in some programming language

[1] Unless we do not understand that language

# An Example

In Python (source code)

```
A = [(1, 2, 3), (4, 5, 6)]
[*zip(*A)]
```

Do you know what is this?

In Pseudocode:

# Some Properties of Good Algorithms

There can be many possible algorithms for solving a problem

Given the choices, we prefer:

- Correctness (the most important property)
- Efficiency (time/space/resources)
- Generality: Applicable to a wide range on inputs and not dependent on a particular computer/device
- Usability as a 'subroutine' for other problems
- Simplicity: so that it is easy to code, understand, debug, etc
- Well documented (easy to understand and to extend it)

Some objectives may have trade-offs: simplicity vs efficiency

# Design and Analysis of Algorithms

Designing an algorithm is both science and art
You need to know the relevant techniques
But you also need creativity, intuition, perseverance

# Paradigms

- Complete Search (for example, using brute force, backtracking, branch and bound)
- Divide and Conquer (D&C)
- Dynamic Programming (DP)
- Greedy Algorithm
- Deterministic versus non-deterministic strategies
- Iterative Improvement

# Problem-Solving

The general steps:

1. Understand the problem
2. Design a method to solve the problem
3. Convert it into an algorithm/pseudocode
4. Choose data structures
5. Prove correctness of the algorithm
6. Analyze the complexity of the algorithm (time/space/resources needed)
7. PS: Implement that correct and efficient algorithm

# Fibonacci Numbers

- $Fib(0) = 0$
- $Fib(1) = 1$
- For $n > 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$
- First 10 terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

Problem: Given $n$ as input, compute $Fib(n)$

We will look at two algorithms:

- Recursive algorithm
- Iterative algorithm

PS: Yes, there are other (faster) algorithms

# Recursive algorithm to compute $Fib(n)$

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

Simple, direct recursive implementation from the $Fib(n)$ definition

# Iterative algorithm to compute $F(n)$

```
define IFib(n)
  if n <= 1
    return n
  else
    prev2 = 0
    prev1 = 1
    for i = 2 to n
      temp = prev1
      prev1 = prev1+prev2
      prev2 = temp
  return prev1
```

# Analysis of an Algorithm

We analyze the resources needed by an algorithm:

- Time – in this course, we will mostly concentrate on time
- Space – in this course, we assume all data fits in memory

Sometimes, we do trade-offs:

- If space is not an issue, most of the time, we sacrifice (or use more) space to gain faster time
- For some applications (e.g., Big Data), we may have to sacrifice time so that we are able to process the data

Actual time needed to run an algorithm depends on the machine used, and this is not easy to calculate/measure

# Model of Computation: RAM

Random-Access Machine (RAM) model is simple
and close to how real computers work:

- Each instruction takes a constant amount of time: fetch the instruction, execute, store back the results in the memory
- We count the number of basic instructions needed
- The time complexity is based on input size (more details soon)

# RAM, Continued

- Word is basic unit of memory
  In this course, you can usually assume each number (or relevant item) can be stored in one word
- RAM is an array of words, storing instructions and data
  It takes one unit of time to access *any* word (this is important)
- Each arithmetic or logical operation (+, -, *, /, mod, AND, OR, NOT, etc) takes a constant amount of time (notice that exponent operation is not constant – see D&C lecture later)
- Details of word size and different time taken by different instructions are important, but USUALLY do not have a large impact; so we usually ignore it, unless it makes a difference
- We need to be careful: when numbers are very large (and thus cannot fit in one word), the complexity depends on number of bits/words needed to store the number

# For our $Fib(n)$ and $IFib(n)$ analysis

For large computation of $Fib(n)$,
the resulting number can be very large

To address the above, one can consider computing the Fibonacci numbers modulo some $m$ (for example $2^{wordsize}$)

We omit this detail in our first analysis to simplify discussion

# Analysis of recursive algorithm to compute $Fib(n)$

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

See the recursion tree@VisuAlgo (which is a big tree for large $n$)

Let $T(n)$ be the number of operations done by $Fib(n)$

$T(0) = T(1) = 2$ (if+return)

For $n \geq 2$, $T(n) = T(n-1) + T(n-2) + 6$ (if+else+two function calls+add+return)

So $T(n) \geq Fib(n)$

We can show that $Fib(n) \geq 2^{\frac{n-2}{2}}$ (How?)

$T(n)$ is exponential in $n$

# Analysis of iterative algorithm to compute $F(n)$

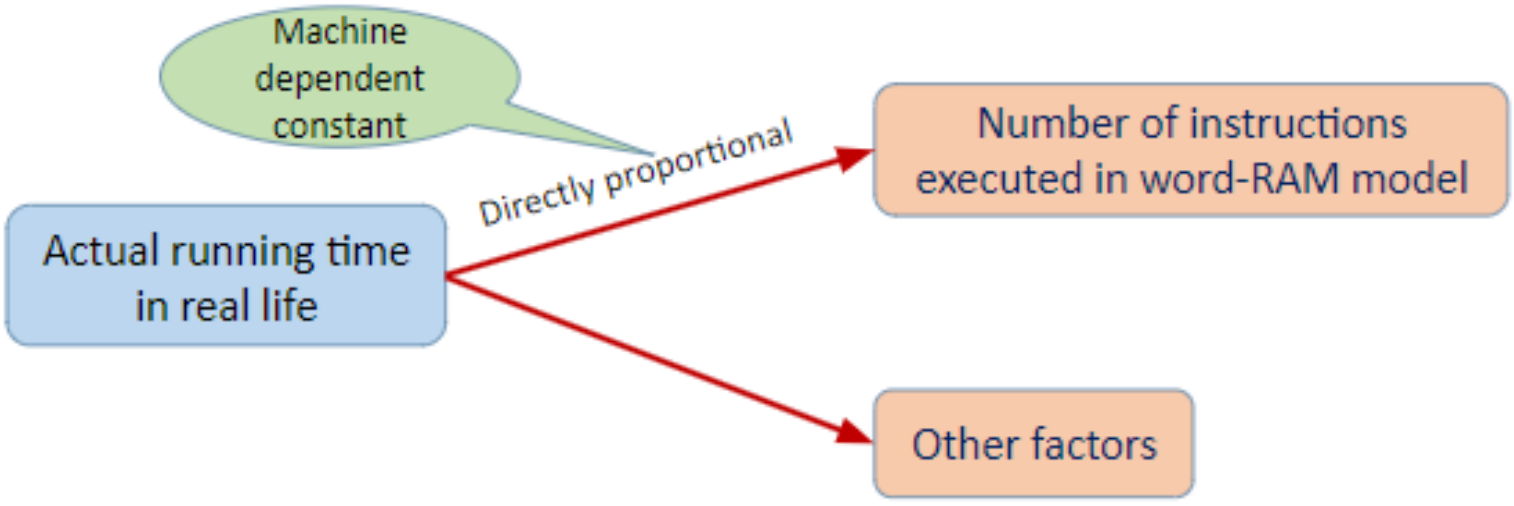
```
define IFib(n)
  if n <= 1
    return n
  else
    prev2 = 0
    prev1 = 1
    for i = 2 to n
      temp = prev1
      prev1 = prev1+prev2
      prev2 = temp
  return prev1
```

For $n \geq 2$,
$T(n) \approx 4 + (n-1) * 5 + 1$
(if+else+two assignments
$+ (n-1)$ iterations,
each takes $\approx 5$ steps
+return)

So $T(n) \approx 5n$, linear in $n$

This is much faster than the recursive version that runs exponential in $n$

# Actual Running Time



# Running Time of an Algorithm

- We often give the running time in terms of the size of the input (usually parameter $n$)
- Size of the input can be the number of items (e.g., sorting $n$ Integers) or length of inputs coded in binary (e.g., Integer $n$ in $Fib(n)$ requires $\log n$ bits encoding – details in the second half)
- We usually perform these analysis:
  - Worst-case analysis: $T(n)$ is the *maximum* time needed for any input of size (at most) $n$
  - Average-case analysis: $T(n)$ is the expected time taken over all inputs of size $n$; either all inputs are equally probable, or we know the probability distribution over the inputs of size $n$
  - We usually do not consider best-case analysis, as inputs that trigger best-case are usually not the typical ones
- It is difficult to compute the exact number of operations (as seen earlier), thus we often give upper bounds instead

# Question 2 at VisuAlgo Online Quiz

Which algorithm is more efficient?

Algorithm 1:
$T1(n) = 100n + 1000$

Algorithm 2:
$T2(n) = n^2 + 5$

# Asymptotic Analysis

Why we do not measure the actual run time:

- Different machines have different speeds,
  i.e., new gaming desktop if fast vs 10-years old laptop is slow
- Different programming languages have different runtimes,
  i.e., C++ is fast vs Python is slow

We prefer to do asymptotic analysis:

- For *large inputs*, how does the runtime behave?
- Comparison of algorithms is based on the asymptotic analysis
- We often ignore lower terms and constant multiplicative factors in the asymptotic analysis

# Most common asymptotic notation: Big O (upper bound)

For the following discussion on asymptotics, assume $f$ and $g$ are functions of *one parameter* $n$

$f \in O(g)$ if there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$

Interpretation: $g$ is an upper bound on $f$

$O(g) = \{f$: there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)\}$

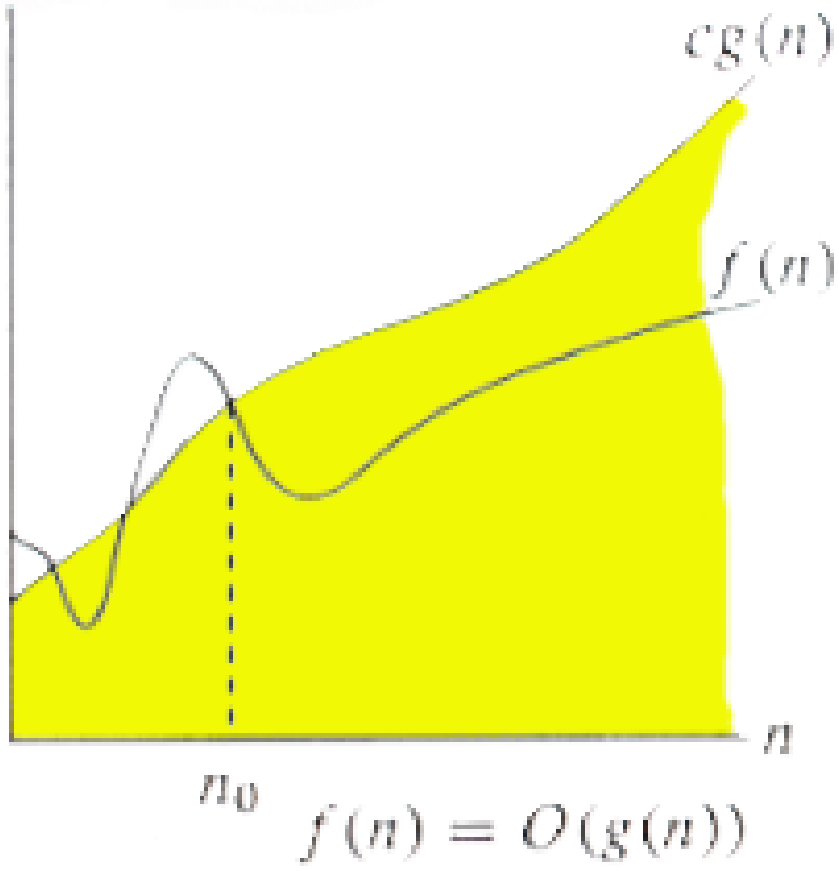
We sometimes also write $f = O(g)$, though not 100% correct

We frequently write $f(n) = O(g(n))$, though technically, $n$ should not have been used (there can be more than one parameter)

Similarly for other asymptotic notations; PS: we **accept** all versions

# Pictorial interpretation of O-notation

Apologize for poor image quality, we plan to redraw these figures




O-notation is an upper-bound notation

So, saying $f(n)$ is *at least* $O(g(n))$ is not correct

# Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive $n$)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)
- $0 \leq 100n + 1000 \leq 101n \leq 101n^2$ (for $n \geq 1000$)
  i.e., we can set $c = 101$ and $n_0 = 1000$

Hence, $100n + 1000 \in O(n^2)$

# Question 3 at VisuAlgo Online Quiz

Let $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$

We want to prove that $f(n) \in O(g(n))$ by showing that
$0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

What should be the appropriate $c$ and $n_0$? (there are $> 1$ answers)

A). $c = 2,\ n_0 = 10$

B). $c = 1,\ n_0 = 11$

C). $c = 5,\ n_0 = 1$

D). $c = 1,\ n_0 = 10$

# New notation $\Omega$ (lower bound)

$f \in \Omega(g)$ if there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq cg(n) \leq f(n)$

Interpretation: $g$ is a lower bound on $f$

# $\Omega$ (lower bound)

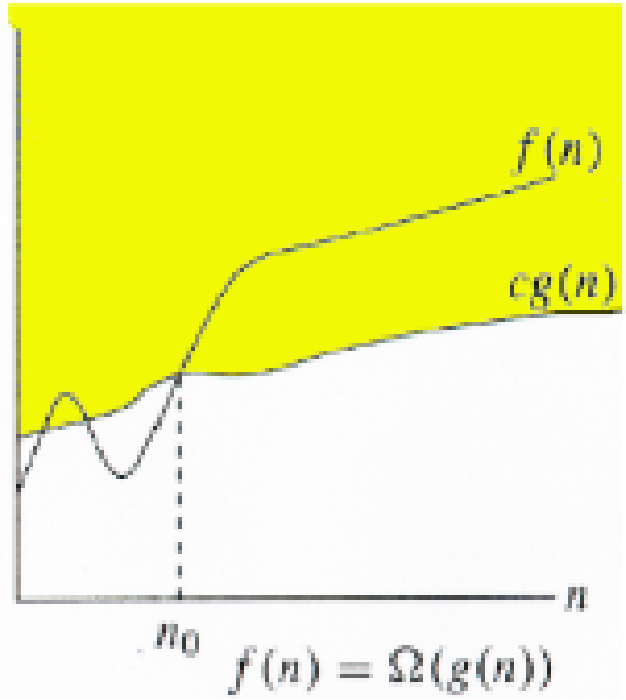
Example: $n^2 \in \Omega(100n + 1000)$
We swap $f(n)$ and $g(n)$ from the earlier Big O example

- $0 \leq \frac{1}{101} \cdot (100n + 1000) \leq n^2$ for $n \geq 1000$
  i.e., we can set $c = \frac{1}{101}$ and $n_0 = 1000$
  just set this $c$ to be the reciprocal of the $c$ in Big O analysis

Again, there are many other possible $c$ and $n_0$

PS: We usually have $f(n)$ as the more complex function and $g(n)$ to be the simpler one, i.e., $7n^2 + 5n + 77 \in \Omega(n^2)$

# Pictorial interpretation of Ω-notation



# New notation $\Theta$ (tight bound)

$f \in \Theta(g)$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Interpretation: $g$ is a tight bound on $f$

We will frequently do $\Theta$ analysis in CS3230

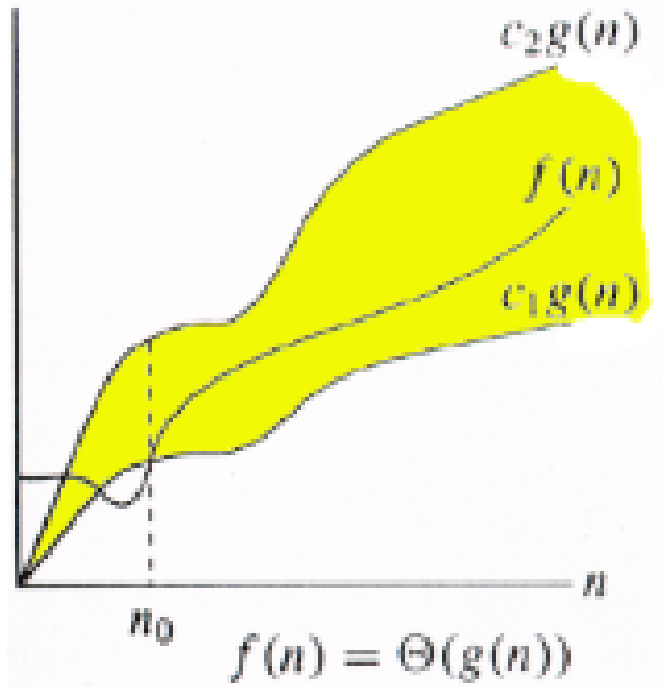
# $\Theta$-notation (tight bound)

Example: $10n^2 + n \in \Theta(n^2)$

- $0 \leq \frac{1}{2}n^2 \leq (10n^2 + n) \leq 11n^2$ for $n \geq 2$
  i.e., $c_1 = \frac{1}{2}$, $c_2 = 11$, and $n_0 = 2$
  again, these are not the only valid constants $c_1$, $c_2$, and $n_0$

Hence, $10n^2 + n \in \Theta(n^2)$

# Pictorial interpretation of Θ-notation



# O, Ω, and Θ

$$\Theta(g) = O(g) \cap \Omega(g)$$

# Little-o (strict upper bound)

$f \in o(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$ (notice **for any constant** $c > 0$ instead of there exists constant $c > 0$ and $<$ instead of $\leq$)

PS: some textbooks define Little-o using $\leq$ instead of $<$
This will only change the chosen $c$ and/or $n_0$

Example: $n \in o(n^2)$
For any constant $c > 0$, let $n_0 = 1 + \frac{1}{c}$ (setting $n_0 = 2$ is also ok)
Then, for $n \geq n_0$, $n < c \cdot n^2$

But $n^2 - n \notin o(n^2)$
Let's say we pick $c = \frac{1}{2}$ (just need to show one counterexample),
for any $n_0$ and large enough $n$, we have:
$n^2 - n > \frac{1}{2}n^2$
$\frac{1}{2}n^2 > n$
$n^2 > 2n$

# $\omega$ (strict lower bound)

$f \in \omega(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$

Example: $n^2 - 36 \in \omega(n)$
For any constant $c > 0$, let $n_0 > \sqrt{36} + c$,
$0 \leq c \cdot n < n^2 - 36$

# Asymptotic Notation: Taking Limits

Assume $f(n), g(n) > 0$, we have:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(g(n))$

It is easier to show $o$, $\Theta$, vs $\omega$ using limits

# $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$

Proof:

By definition of limit, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, means
$\forall \epsilon > 0, \exists n_0 > 0$, such that $\forall n \geq n_0$,
$\frac{f(n)}{g(n)} < \epsilon$

Hence, for any constant $c > 0$ (i.e., we can set $c = \epsilon$), $\exists n_0 > 0$,
such that $\forall n \geq n_0$,
$f(n) < \epsilon \cdot g(n)$, i.e.,
$f(n) < c \cdot g(n)$,
$f(n) \in o(g(n))$

We will prove at least one other during Tut01

# Example

By limit, show that $n^6 + 233n^2 \in \omega(n^2)$

$\lim_{n\to\infty} \frac{n^6+233n^2}{n^2} = \lim_{n\to\infty} \frac{n^4+233}{n^2} = \infty \Rightarrow f(n) \in \omega(g(n))$

# Asymptotic Notation: Some Properties

- Reflexivity: For $O, \Omega$, and $\Theta$,
  $f(n) = O(f(n))$, similarly for $\Omega$ and $\Theta$
- Transitivity: For all five: $O, \Omega, \Theta, o$, and $\omega$
  $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ implies $f(n) \in O(h(n))$
- Symmetry:
  $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$
- Complementary:
  $f(n) = O(g(n))$ iff $g(n) \in \Omega(f(n))$
  $f(n) = o(g(n))$ iff $g(n) \in \omega(f(n))$

We will prove some of these during Tut01

See Asymptotic_Analysis-Useful_Facts.pdf for math refresher

# Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere

List of credits: Erik D. Demaine, Charles E. Leiserson, Surender Baswana, Leong Hon Wai, Lee Wee Sun, Ken Sung, Diptarka Chakraborty, Steven Halim, Sanjay Jain