B2: Orders of Growth

CS1101S: Programming Methodology

Low Kok Lim

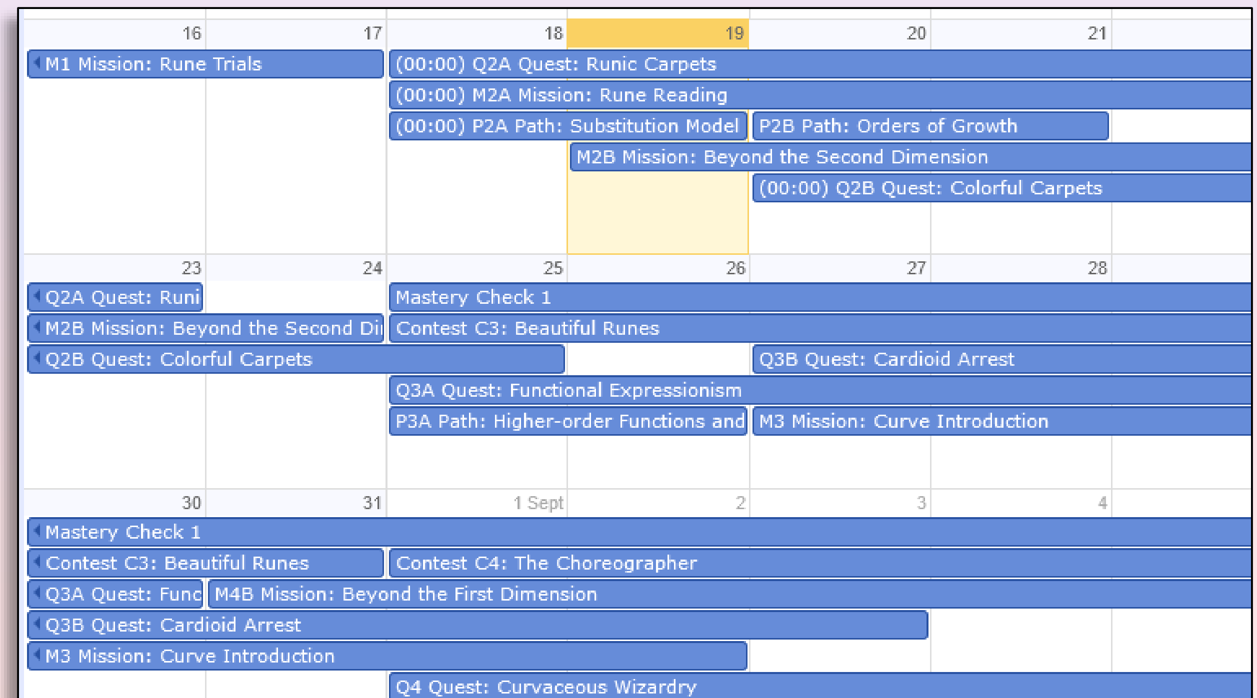
August 20, 2021

Outline

- Assessment Details
- Orders of Growth (Textbook 1.2.3)

Mission and Quest Calendar

- [CS1101S Timelines] calendar
  - Link can be found at **LumiNUS > Module Details > Weblinks**
  - Link for adding it to your Google Calendar



Studio Attendance and Participation

- **Attendance and Participation** (5% of CS1101S)
  - Studios are small communities of learners; your presence in the sessions is needed to be an effective member
    - **3%** for Studio **attendance**
    - **2%** for Studio **participation**; Avengers assess your participation, based on **effort**
  - The **ideal Studio member**
    - prepares well for the meetings, contributes by answering any questions thoughtfully, asks good questions, and tries to help classmates in mastering the material
- **Studio Performance XP**
  - Maximal 500 XP per Studio session
  - Here is where you can shine, by helping others, contributing to discussions, going the extra mile

Reading Assessments

- In-class **MCQ** tests

- **Reading Assessment 1** (6%)
  - **Topics:** Processes, Correctness, Scope
  - Week 4, Friday, 3-Sep-2021, 10am-12pm

- **Reading Assessment 2** (6%)
  - Topics to be announced later
  - Week 10, Friday, 22-Oct-2021, 10am-12pm

- Administered as **E-Exams** (online). Details TBA

Reading Assessment 1

- **Processes:** explain program runs using substitution model, distinguish between iterative and recursive processes (L2)
- **Correctness:** comprehend simple specification and decide whether a given program meets it
- **Scope:** see scope of any name declaration, find declaration that a given name occurrence refers to (L3)

Outline

- Assessment Details
- Orders of Growth (Textbook 1.2.3)

A Closer Look at Performance

- Dimensions of performance
  - **Time:** how long does the program run
  - **Space:** how much memory do we need to run the program

Recall the Factorial Function

- Recursive definition

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n\,(n-1)! & \text{if } n > 1 \end{cases}$$

- **In Source:**

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}

// Example call
factorial(4);
```

Show in Playground

Time for Calculating `factorial(n)`

```
factorial(4)
➔ 4 * factorial(3)
➔ 4 * (3 * factorial(2))
➔ 4 * (3 * (2 * factorial(1)))
➔ 4 * (3 * (2 * 1))
➔ 4 * (3 * 2)
➔ 4 * 6
➔ 24
```

- **Observation:**
  - **Number of operations** *grows* ***linearly*** proportional to *n*

Space for Calculating `factorial(n)`

```
factorial(4)
➔ 4 * factorial(3)
➔ 4 * (3 * factorial(2))
➔ 4 * (3 * (2 * factorial(1)))
➔ 4 * (3 * (2 * 1))
➔ 4 * (3 * 2)
➔ 4 * 6
➔ 24
```

- **Observation:**
  - **Number of deferred operations** *grows* ***linearly*** proportional to *n*
    - Deferred operations need to be “remembered”

Example: Doubling the Argument of `factorial`

```
factorial(2)
➔ 2 * factorial(1)
➔ 2 * 1
➔ 2

factorial(4)
➔ 4 * factorial(3)
➔ 4 * (3 * factorial(2))
➔ 4 * (3 * (2 * factorial(1)))
➔ 4 * (3 * (2 * 1))
➔ 4 * (3 * 2)
➔ 4 * 6
➔ 24
```

- The **number of steps** "roughly" doubles
- The `factorial` function runs in a **time** (**linearly**) proportional to the argument *n*
- The **number of deferred operations** also "roughly" doubles
- It has **space** requirement (**linearly**) proportional to the argument *n*

Fibonacci Numbers

- **Fibonacci sequence**
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
    - Each number is the sum of the previous two

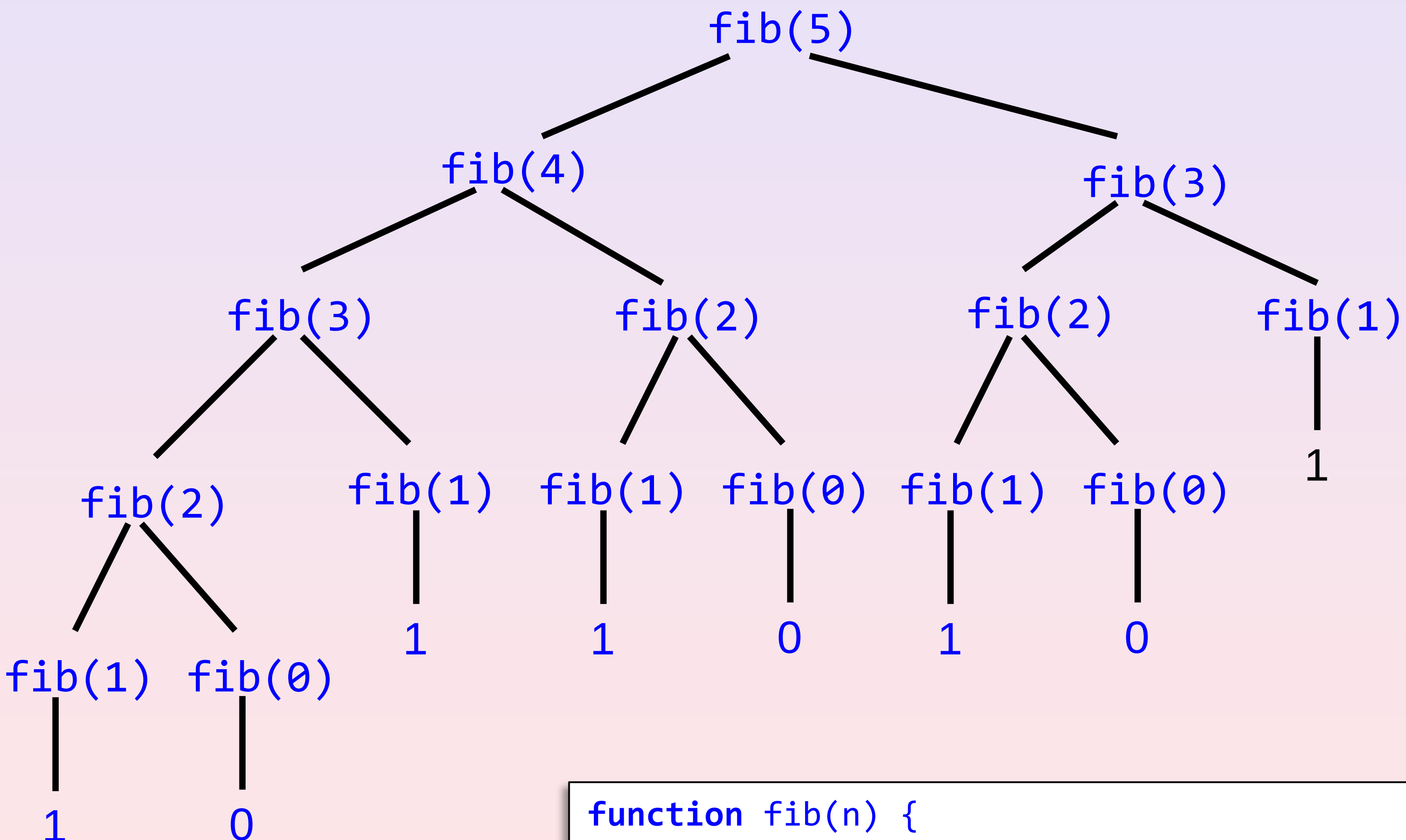
- **Fibonacci function** $F(n)$, such that $F(0) = 0$, $F(1) = 1$, $F(2) = 1$, $F(3) = 2$, $F(4) = 3$, and so on

- An attempt to implement $F(n)$ in Source:

```
function fib(n) {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
```

[Show in Playground]

Evaluating `fib(5)`




```
function fib(n) {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
```

Time for Evaluating `fib(n)`

- **Time** for exploring the tree grows with **size** of tree
- Tree for `fib(n)` has $F(n+1)$ leaves, where

$$F(n) = \left\lfloor \frac{\emptyset^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor \quad \text{and} \quad \emptyset = \frac{1+\sqrt{5}}{2} \approx 1.618$$

- Can we write an ***efficient iterative*** function that computes $F(n)$?
  - Yes, we have seen it in Reflection R2

Example: Computing $F(n)$ using `fib(n)`

- Number of leaves in the recursion tree for `fib(n)` is $F(n + 1)$
  - `fib(10)` needs to visit $F(11)$ = 89 leaves
  - `fib(20)` needs to visit $F(21)$ = 10946 leaves
  - `fib(40)` needs to visit $F(41)$ = 165580141 leaves
  - `fib(100)` needs to visit $F(101)$ = 573147844013817084101 leaves

Orders of Growth

- **Linear growth**
  - The `factorial` function runs in a time (**linearly**) proportional to the argument *n*
- **Exponential growth**
  - The `fib` function runs in a time that grows **exponentially** with the argument *n*
- What exactly do we mean by the above?

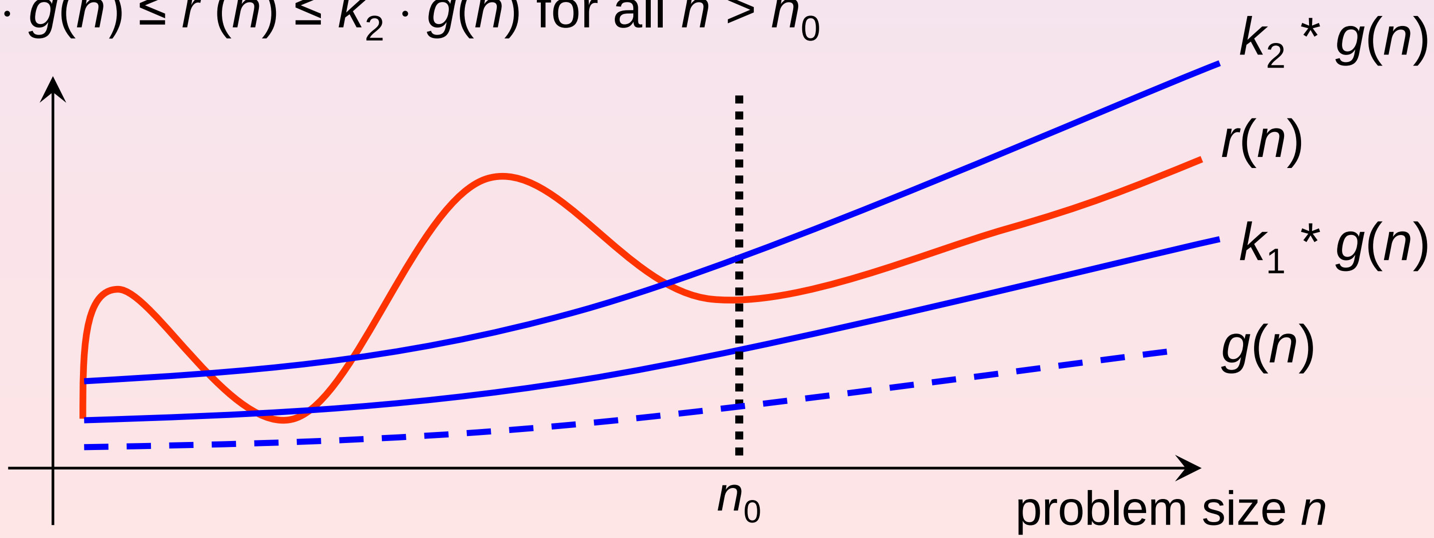
Purpose

- **Rough measure**
  - We are interested in a rough measure of **resources** used by a computational process, with respect to the **problem size**
- **Abstraction**
  - "**Order of growth**" is an abstraction technique
  - We decide to ignore details that we deem irrelevant
    - Examples: the processor speed of the computer, the programming environment, the programming language, or minor differences in programming style

The Θ ("Big Theta") Notation

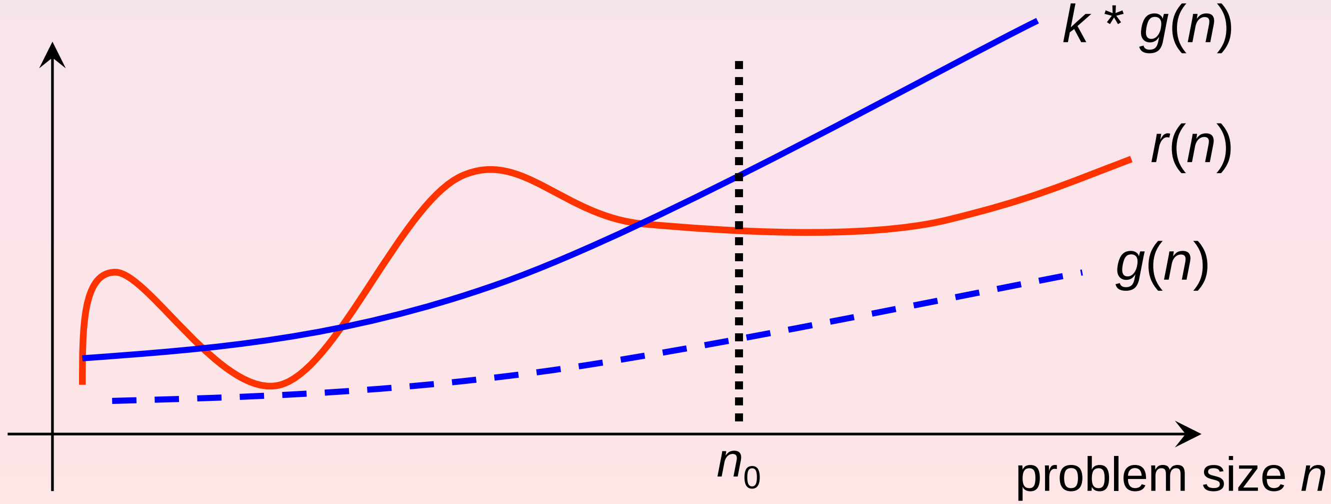
- Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n

- **Definition:**
  - The function r has order of growth $\Theta(g(n))$ if there are positive constants k_1 and k_2 and a number n_0 such that $k_1 \cdot g(n) \le r(n) \le k_2 \cdot g(n)$ for all $n > n_0$



The Big-O ("Big Oh") Notation

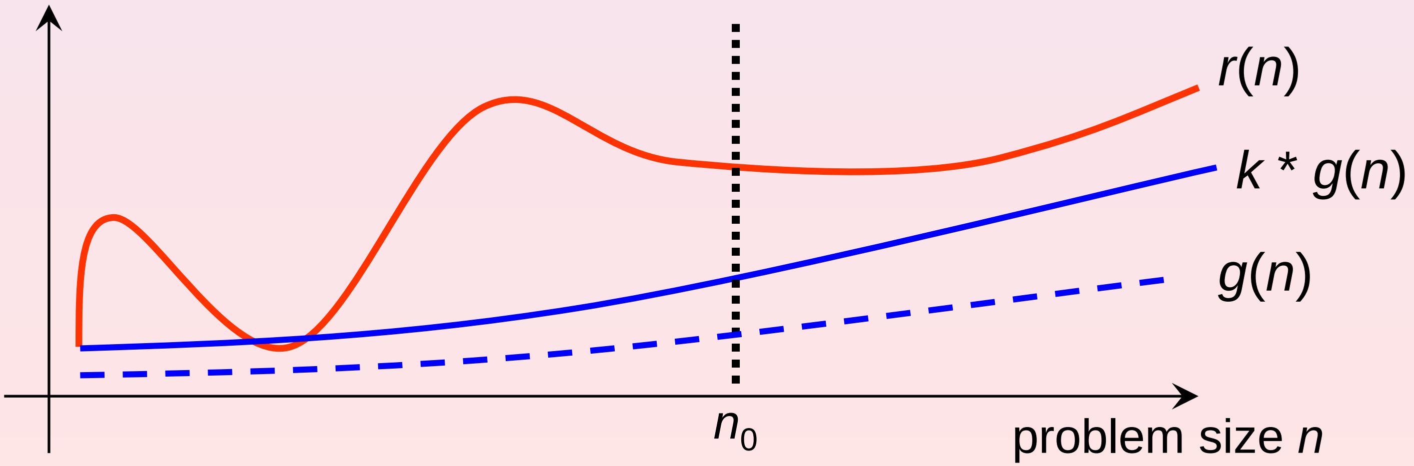
- Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n
- **Definition:**
  - The function r has order of growth $O(g(n))$ if there is a positive constant k and a number n_0 such that $r(n) \leq k \cdot g(n)$ for all $n > n_0$



The Ω ("Big Omega") Notation

- Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n

- **Definition:**
  - The function r has order of growth $\Omega(g(n))$ if there is a positive constant k and a number n_0 such that $k \cdot g(n) \leq r\,(n)$ for all $n > n_0$



Do Constants Matter?

- Let's say *r* has order of growth $\Theta(n^2)$
- Does *r* also have order of growth $\Theta(0.5\, n^2)$?
- Constants don't matter
  - Because we can freely choose k, k_1, k_2

Do Minor Terms Matter?

- Let's say r has order of growth $O(n^2)$
- Does r also have order of growth $O(n^2 + 40n - 5)$?
- Minor terms don't matter
  - Because we can adjust n_0, k, k_1, k_2 such that the minor terms are overruled

Some Common $g(n)$

- 1
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n

Example Orders of Growth

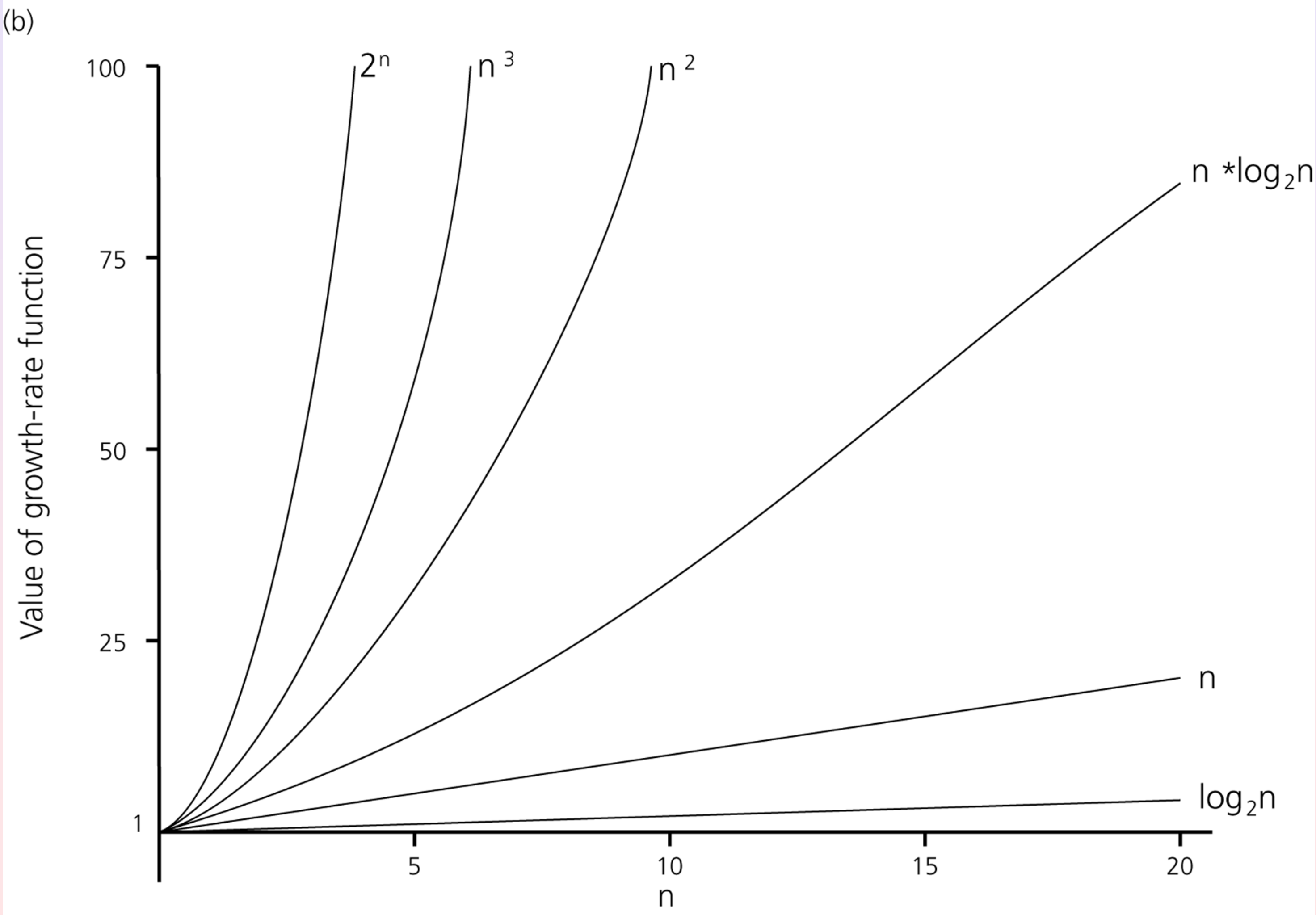
- **Linear time**
  - The `factorial` function runs in a time that has order of growth $\Theta(n)$, where n is the function argument
  - Can we say its order of growth is $O(n)$?
  - Can we say its order of growth is $O(n^2)$?
- **Exponential time**
  - The `fib` function runs in a time that has order of growth $\Theta(\emptyset^n)$, where n is the function argument and $\emptyset = \frac{1+\sqrt{5}}{2} \approx 1.618$

Comparing Orders of Growth

(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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Comparing Orders of Growth



How to Calculate “Big Oh/Theta/Omega”

- Topic of Algorithm Analysis (CS3230)

- For us:
  - Identify the basic computational steps
  - Try a few small values
  - Extrapolate
  - Watch out for “worst case” scenarios

Summary

- Resources for computational processes: **time** and **space**
- Big Theta, Big Oh, Big Omega
  - Provide abstractions or “rough” measures of **resources used** with respect to **problem size**