FIT2004 - Algorithms and Data Structures

Seminar 1 - Introduction

Rafael Dowsley

3 March 2025

Agenda

Unit Information

Teaching team

- **Chief examiner/co-lecturer:** Rafael Dowsley
- **Co-lecturer:** Anuja Dharmaratne
- **Malaysia lecturer/coordinator:** Lim (Ian) Wern Han
- **Admin TAs:** Harrison Sloan, Joshua Nung, Mubasshir Murshed, Sachinthana Pathiranage
- **TAs:** David Batonda, Elijah Lewis, Ethan Wills, Jackson Goerner, Klarissa Jutivannadevi, Luhan Cheng, Michael Xue, Nathan Companez, Saman Ahmadi, Satya Jhaveri, Shen-Kit Hia, Susilo Lebang, Thomas Hendrey, Yisong Yu

What is this unit about?

- Solving problems with computers - **efficiently**.
- Developing your algorithm toolbox.
- Training your problem solving skills.
- The unit is not really about programming:
    - Python used for assignments, but the subject is really language agnostic.
    - Algorithms in this unit will be described in English, pseudocode, procedural set of instructions or Python.

Is this unit important?

- Algorithms and Data Structures is a key unit in computer science degrees around the world.
- The subject is very important for careers in the area:
  - Companies actively hunt for people good at algorithms and data structures.
  - Many job interview questions are based on algorithms from this unit.
  - Many applied class questions are in fact very similar to questions you could be asked in job interviews.
  - **You are the future of CS and will make great contributions to field. What you learn in this unit will greatly help you throughout your career.**

Overview of the contents

- Explore some of the most important algorithm design paradigms:
    - Divide-and-Conquer
    - Greedy algorithms
    - Dynamic Programming
    - Network Flow
- Analysis of algorithms.
- Learn important data structures for implementing algorithms efficiently.
- Algorithms for solving important computational problems.

Expectations

- This unit is challenging.
  - If you don't have a good understanding of the contents of prerequisite units (e.g., FIT1008), you need to catch up on them urgently to not get in trouble.
  - You have to be on top of it from Week 1. You will very likely not pass if you think "I can brush up on the material close to the assessment deadlines".
- The minimum expected workload is 144 hours.
  - The best way to spend "the first 60 hours" is by attending and engaging in the Seminars and Applied Classes.
  - Missing Seminars or Applied Classes will require double the efforts to recover.

Good news

- The unit wants you to succeed and understand. Lot of resources available:
  - Notes for all 12 weeks are available in a single PDF file on Moodle (click on “Learning”, then “Additional Information and Resources” and scroll down to “Unit Resources”).
  - Solutions to the Applied Classes’ problems are released.
  - Preparation sheets and solutions are provided.
  - Support on Ed discussion platform and in the consultations.
- Majority of the students that regularly engage with the classes end up getting D/HD.

Support

- Contents and general questions should be clarified on Ed forum or during a consultation.
- For questions involving sensitive matters that cannot be solved on Ed forum, please email **fit2004.clayton-x@monash.edu**.
- **Do not email individual staff members.**

Additional references

- This subject has huge value for your professional development into careers in computer science. Some books you might want to refer to (from time-to-time, even beyond this unit) include:
    - CLRS: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*.
    - KT: Jon Kleinberg, Éva Tardos. *Algorithm Design.*
    - Rou: Tim Roughgarden. *Algorithms Illuminated*.
    - Knu: Donald Knuth. *The Art of Computer Programming*. More advanced, pretty expensive; but iconic CS book!
- No required textbooks. Beware that all of those textbooks contain both less and (far) more than is required for the unit. That is why it is important to not rely solely on these books (i.e., you should watch the seminars, attend applied classes and read the course notes).

Assessment

- In-semester assessments (total of 50 marks):
  - Weekly Quizzes (2 marks per week, 22 marks in total): during each Applied Class; 1 or 2 questions per week; 10 minutes to solve. Bring your own Internet-connected device to answer on Moodle; and also blank working sheets if you plan to use them.
  - Assignment 1 (10 marks) due on Wednesday of Week 7.
  - Assignment 2 (18 marks), due on Wednesday of Week 12.
  - In-semester hurdle: 45% of the in-semester marks (i.e., at least 22.5 out of total 50 marks for Quizzes and Assignments).
- Final exam (50 marks)
  - Exam hurdle: 45% (i.e., at least 22.5 out of the 50 Exam marks).

How to succeed in this unit?

- During Week x, engage with Seminar x.
- During Week x, read the corresponding parts of the unit notes and clarify any questions in the consultations or Ed forum.
- Attend the Applied Classes and engage with them:
  - Applied Class of Week x deepens the understanding of topics presented in Seminar $x - 1$.
  - Do the Preparation x problems before the Applied Class of Week x. They help you practice and self-assess your understanding of the topics of Seminar $x - 1$. Solutions are provided.
  - The Preparation problems will likely be helpful to get you ready for the Quiz as the Quiz covers contents from classes up to Week $x - 1$.
  - The Assignments and the Exam assess the concepts explored in the Applied Classes.

How to get into trouble in this unit?

- Don't engage with classes and focus all effort on completing assignments.
  - The assignments are designed to be done by people with a strong grasp of the contents and are heavily based on the content taught in the unit.
  - If you spend time understanding the content taught in the classes, the assignments will be far easier.
  - Historically, many of those students focusing all their effort on completing assignments will fail the exam hurdle as it covers all contents of the unit.
  - Those students normally struggle a lot with exam questions that are taken almost "as is" from Applied Classes and Seminars.
- Constructive alignment is used in this unit.

Academic integrity

- **Cheating:** Seeking to obtain an unfair advantage in an examination or in other written or practical work required to be submitted or completed for assessment.
- **Collusion:** Unauthorised collaboration on assessable work with another person or persons.
- **Plagiarism:** To take and use another person's ideas and or manner of expressing them and to pass them off as one's own by failing to give appropriate acknowledgement. This includes material from any source, staff, students or the Internet – published and un-published works.
- **Generative AI tools cannot be used for any assessment in this unit.**

How to avoid academic integrity issues

- `https://www.monash.edu/students/study-support/academic-integrity`
- Do not discuss the assessment tasks with other students.
- *High-risk game: every semester a considerable number of academic integrity cases are opened in this unit. After SCC investigations and decisions, most result in a "zero marks for assignment" penalty (which makes it hard to pass the hurdle) or a straight "zero marks for unit" penalty.*
- What can you do? Share test cases! Feel free to post your test cases on the Ed post that will be created for that, and to use other's.

Give us feedback!

- We are continuously trying to improve the unit.
- Tell us about things you dislike/think should be done differently.
- A few examples of changes which were motivated by student feedback:
  - Reduction in exam grade percentage to 50%.
  - Reduction of the number of assignments from 4 to 2.
  - Ed thread specifically for sharing test cases for assignments.
  - Releasing Applied Classes' solutions one week earlier.

Your first algorithm

- What was the first algorithm you learned?
  - As an CS student you already learned some algorithms in the university: binary search, sorting algorithms, etc.
  - But algorithms predate computers by millennia. Even the word 'algorithm' is derived from the name of a 9th century Persian mathematician.
  - In fact, you learned in school an algorithm that is more than 2000 years old: Euclid's algorithm for computing the greatest common divisor.
  - Even in your first school years you already learned the "grade school" multiplication algorithm.

Grade school multiplication algorithm

- Grade school multiplication algorithm using partial products:

$$\begin{array}{cccccc}
 & & & 1 & 2 & 3 \\
\times & & & 3 & 4 & 5 \\
\hline
 & & & 6 & 1 & 5 \\
 & & 4 & 9 & 2 & \\
 & 3 & 6 & 9 & & \\
\hline
 & 4 & 2 & 4 & 3 & 5
\end{array}$$

- Did your teacher talk about the efficiency of this algorithm? Showed the correctness proof?
- Well, back then you were only a user of the algorithm.
- **In the future, understanding the efficiency and correctness of algorithms will be a central skill in your career.**

Grade school multiplication algorithm

- Grade school multiplication algorithm using partial products:

$$\begin{array}{rrrrrr} \times & & & 1 & 2 & 3 \\ & & & 3 & 4 & 5 \\ \hline & & & 6 & 1 & 5 \\ & & 4 & 9 & 2 & \\ & 3 & 6 & 9 & & \\ \hline & 4 & 2 & 4 & 3 & 5 \end{array}$$

- If we consider addition and multiplication of single digit numbers as the basic operations, for n-digit numbers, this algorithm clearly has complexity that is quadratic in n.

- Fundamental question in algorithm design: Can we do it more efficiently?

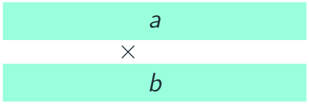
Divide-and-Conquer

Divide-and-Conquer paradigm

- The Divide-and-Conquer algorithm design paradigm works in 3 steps:
    1. Divide the problem into smaller subproblems.
    2. Conquer (i.e., solve) the smaller subproblems.
    3. Combine the solutions of the smaller subproblems to obtain the solution of the bigger problem.
- Analysing the time complexity of a divide-and-conquer algorithm normally involves solving a recurrence relation.
- Normally a polynomial time solution to the problem is already know, and the divide-and-conquer strategy is used to reduce the time complexity to a lower polynomial.

First improvement idea

- **Problem:** multiply two n-digits numbers a and b in sub-quadratic time given addition and multiplication of single digit numbers as the basic operations.



- **Improvement idea:** split the numbers between the $n/2$ most significant digits and the $n/2$ least significant digits; and do recursive calls with them.

- From math we know that:

$$\begin{aligned} a \cdot b &= (a_M \cdot 10^{n/2} + a_L)(b_M \cdot 10^{n/2} + b_L) \\ &= a_M \cdot b_M \cdot 10^n + a_M \cdot b_L \cdot 10^{n/2} + a_L \cdot b_M \cdot 10^{n/2} + a_L \cdot b_L \end{aligned}$$

Are we making progress?

$$a \cdot b = a_M \cdot b_M \cdot 10^n + a_M \cdot b_L \cdot 10^{n/2} + a_L \cdot b_M \cdot 10^{n/2} + a_L \cdot b_L$$

- Reduce 1 instance of the problem of size n to 4 instances of size $n/2$:

a_M	a_M	a_L	a_L
$\times$	$\times$	$\times$	$\times$
b_M	b_L	b_M	b_L

- Are we making progress?

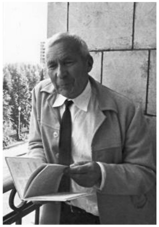
- Not really! Intuition: 4 instances that will each take about $\frac{1}{4}$ of the time that was necessary to solve the original problem, so the overall time stays in the same order. You can later check that by solving the recurrence $T(n) = 4 \cdot T(n/2) + c \cdot n$.

- If we want to follow this approach to improve the efficiency, we should use at most 3 recursive calls.

Are improvements possible?

Conjecture: Sub-quadratic multiplication is impossible!

That is wrong!



Andrey Kolmogorov, one of the greatest mathematicians of the 20th century.

Anatoly Karatsuba, then a 23 y/o student, within one week from hearing that in his seminar.

Karatsuba's algorithm

$$a \cdot b = a_M \cdot b_M \cdot 10^n + (a_M \cdot b_L + a_L \cdot b_M) \cdot 10^{n/2} + a_L \cdot b_L$$

- Do only 3 recursive calls to compute:

$$\begin{array}{ccc} a_M & a_L & a_M + a_L \\ \times & \times & \times \\ b_M & b_L & b_M + b_L \\ = (1) & = (2) & = (3) \end{array}$$

- Given the results of (1), (2) and (3), if we can trivially obtain $(a_M \cdot b_L + a_L \cdot b_M)$ then we are done computing $a \cdot b$ with only 3 recursive calls.

The trick

- **Problem:** Obtain $c = (a_M \cdot b_L + a_L \cdot b_M)$ without further recursive calls when given:

$$\begin{array}{ccc} \boxed{a_M} & \boxed{a_L} & \boxed{a_M + a_L} \\ \times & \times & \times \\ \boxed{b_M} & \boxed{b_L} & \boxed{b_M + b_L} \\ = (1) & = (2) & = (3) \end{array}$$

- Note that

$$\begin{aligned} (a_M + a_L) \cdot (b_M + b_L) &= a_M \cdot b_M + \; a_M \cdot b_L \cdot + a_L \cdot b_M \; + a_L \cdot b_L \\ &= c + a_M \cdot b_M + a_L \cdot b_L \end{aligned}$$

- **Solution:** We obtain c by computing (3) - (1) - (2).

- Where does this trick come from?

Where does this trick come from?

- This trick traces back to Gauss' method for multiplying complex numbers using 3 multiplications of real numbers instead of 4.

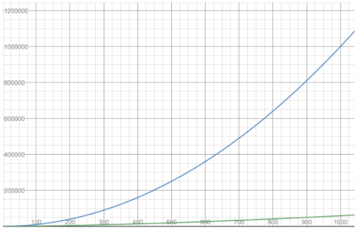
Johann Carl Friedrich Gauss

- **Adapting previous ideas to solve your new problem can be very useful!**

Is that a big improvement?

- The time complexity of Karatsuba's algorithm is $O(n^{1.59})$. To verify that, just solve the recurrence $T(n) = 3 \cdot T(n/2) + c \cdot n$ and use the fact that $\log_2 3 < 1.59$.

- Don't underestimate the difference between n^2 and $n^{1.59}$!



- **This is the algorithm that Python uses for multiplying large numbers.**

Merge Sort

- $O(n \log n)$ sorting algorithm presented by John von Neumann.
- One of the first explicit uses of the Divide-and-Conquer paradigm.
- Python uses as standard Timsort (a hybrid stable sorting algorithm derived from Merge Sort and Insertion Sort).

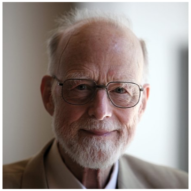
Merge sort

```
function MERGE_SORT($array[lo..hi]$)
    if hi $>$ lo then
        $mid = \lfloor (lo + hi)/2 \rfloor$
        MERGE_SORT(array[$lo..mid$])
        MERGE_SORT(array[$mid + 1..hi$])
        array[$lo..hi$] $=$ MERGE(array[$lo..mid$], array[$mid + 1..hi$])
```

John von Neumann, one of the greatest mathematicians of the 20th century.

Quick Sort

- Sorting algorithm created by Tony Hoare that uses the Divide-and-Conquer paradigm, but the subproblems are not necessarily of the same size.



Tony Hoare (Turing Award 1980)

- Seminar 3 will cover it in detail.

Other Divide-and-Conquer examples

The Divide-and-Conquer paradigm can be applied to get efficient algorithms for a wide range of problems, such as:

- Finding closest pair of points in a plane in $O(n \log n)$.
- Counting inversions in $O(n \log n)$, Applied Class next week.
- Improving matrix multiplication (Strassen's algorithm).
- Fast Fourier Transform: this algorithm published by James Cooley and John Tukey in 1965 is one of the most influential algorithms, with a wide range of applications in engineering, music, science, mathematics, etc.
    - In fact, it can be traced back to unpublished work by Gauss.

Complexity Analysis

Time complexity

- Time complexity is the amount of time taken by an algorithm to run as a function of the input size.
  - Worst-case complexity (our main focus).
  - Best-case complexity.
  - Average-case complexity.

Asymptotic notation

Big-O Notation

It is said that $f(n) = O(g(n))$ if there are constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Big-Ω Notation

It is said that $f(n) = \Omega(g(n))$ if there are constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Big-Θ Notation

It is said that $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Space complexity

- Space complexity is the total amount of space taken by an algorithm as a function of input size.
- Auxiliary space complexity is the amount of space taken by an algorithm *excluding the space taken by the input*.
    - Many textbooks and online resources do not distinguish between the above two terms and use the term "space complexity" when they are in fact referring to auxiliary space complexity.
    - In this unit, we use these two terms to differentiate between them.

In-place algorithm

- An in-place algorithm has $O(1)$ auxiliary space complexity.
  - In other words, it only requires constant space in addition to the space taken by its input.
  - Merging is not an in-place algorithm as it needs to create the output list which is size n.
  - Be mindful that some books use a different definition (e.g., space taken by recursion may be ignored). For the sake of this unit, we will use the above definition.

Time complexity of Binary Search

Binary Search

```
function BINARY_SEARCH($array[1..n]$, $key$)
    $lo = 1$ and $hi = n + 1$
    while $lo < hi - 1$ do
        $mid = \lfloor (lo + hi)/2) \rfloor$
        if $key \geq array[mid]$ then $lo = mid$
        else $hi = mid$
    if $array[lo] = key$ then return $lo$
    else return null
```

- Worst-case time complexity?
  - Search space at start: n
  - After 1st iteration: $n/2$
  - After 2nd iteration: $n/4$
  - ...
  - After x iterations: 1
  - How many iterations?
  - $O(\log n)$
- Best-case time complexity?
  - $O(1)$

Space complexity of Binary Search

Binary Search

```
function BINARY_SEARCH($array[1..n]$, $key$)
    $lo = 1$ and $hi = n + 1$
    while $lo < hi - 1$ do
        $mid = \lfloor (\text{lo} + \text{hi})/2) \rfloor$
        if $key \geq array[mid]$ then $lo = mid$
        else $hi = mid$
    if $array[lo] = key$ then return $lo$
    else return null
```

- Space complexity?
  - $O(n)$
- Auxiliary space complexity?
  - $O(1)$
- It is an in-place algorithm!

What is the time complexity?

- **Problem:** Given a sorted array of n unique numbers and two values x and y, print all numbers that are in between x and y.
- **Algorithm:** Binary search to find the smallest number greater than x. Linear scan from x until next number is $\geq y$.
- What is the time complexity?
- $O(n)$ because all numbers may be between x and y.
- But it seems to really depend on the output ...

Output-sensitive time complexity

- Output-sensitive time complexity is the time-complexity that also depends on the size of the output.

$$x = 23, y = 35$$

1	5	8	17	22	27	31	32	36	41

- **Algorithm:** Binary search to find the smallest number greater than x. Linear scan from x until next number is $\geq y$.

- Let w be the number of values in the range (i.e., in output). What is the output-sensitive time complexity of the algorithm?

- $O(w + \log n)$. Note that w may be n in the worst-case.

- Output-sensitive complexity is only relevant when output-size may vary, e.g., it is not relevant for sorting, finding minimum value etc.

Solving Recurrence Relations

Recurrence relation

- A recurrence relation is an equation that recursively defines a sequence of values, and one or more base cases are given, e.g.:

$$\begin{aligned} T(1) &= b \\ T(n) &= T(n-1) + c \end{aligned}$$

- The complexity of recursive algorithms can be analysed by writing its recurrence relation and then solving it.

Solving a simple recurrence relation for time complexity

Power1

```
function POWER1(x, n)
    if n = 0 then return 1
    else if n = 1 then return $x$
    else return $x \cdot \text{POWER1}(x, n-1)$
```

Goal

Reduce general case to be in terms of the base case.

Solution

$$\begin{aligned} T(n) &= b + c \cdot (n-1) \\ &= c \cdot n + b - c \\ &= O(n) \end{aligned}$$

- Cost when $n = 1$:
  - $T(1) = b$ for constant b
- Cost for general case:
  - $T(n) = T(n-1) + c$ for constant c
  - $T(n) = T(n-2) + 2c$
    $T(n) = T(n-3) + 3c$
  - Pattern?
  - $T(n) = T(n-k) + c \cdot k$
  - Set $k = n - 1$ to get base case.

Checking solution by substitution

Power1

```
function POWER1(x, n)
    if n = 0 then return 1
    else if n = 1 then return $x$
    else return $x \cdot$ POWER1($x, n-1$)
```

Goal

Reduce general case to be in terms of the base case.

Solution

$$\begin{aligned} T(n) &= b + c\cdot(n-1) \\ &= c\cdot n + b - c \\ &= O(n) \end{aligned}$$

- Cost when $n = 1$:
  - $T(1) = b$ for constant b
- We have that:

$$\begin{aligned} T(1) &= c\cdot 1 + b - c \\ &= b \end{aligned}$$

- Cost for general case:
  - $T(n) = T(n-1) + c$ for constant c
- We have that:

$$\begin{aligned} T(n-1) + c &= c\cdot(n-1) + b - c + c \\ &= c\cdot n + b - c \\ &= T(n) \end{aligned}$$

Space complexity of this power function

Power1

```
function POWER1(x, n)
    if n = 0 then return 1
    else if n = 1 then return $x$
    else return $x \cdot \text{POWER1}(x, n-1)$
```

Power2

```
function POWER2(x, n)
    $result = 1$
    for $i \leftarrow 1$ to $n$ do
        $result = result \cdot x$
    return $result$
```

- Space complexity?
  - Total space usage = local space used by the function * maximum depth of recursion
  - $O(n)$
- We will not discuss tail-recursion in this unit because it is language specific, e.g., Python doesn't utilise tail-recursion.
- Auxiliary Space Complexity?
- $\text{POWER1}(x, n)$ is not in-place.
- Iterative version $\text{POWER2}(x, n)$ is in-place.

Yet another power function

Power3

```
function POWER3(x, n)
    if n = 0 then return 1
    else if n = 1 then return $x$
    $y = \text{POWER3}(x \cdot x, \lfloor n/2 \rfloor)$
    if n even then return $y$
    else return $x \cdot y$
```

Goal

Reduce general case to be in terms of the base case.

Solution

$$\begin{aligned} T(n) &= b + c \cdot \log n \\ &= O(\log n) \end{aligned}$$

- Cost when $n = 1$:
  - $T(1) = b$ for constant b
- Cost for general case:
  - $T(n) = T(n/2) + c$ for constant c
  - $T(n) = T(n/4) + 2c$
    $T(n) = T(n/8) + 3c$
  - Pattern?
  - $T(n) = T\left(\frac{n}{2^k}\right) + c \cdot k$
  - Set $k = \log n$ to get base case.

Another check by substitution

Power3

```
function POWER3(x, n)
    if n = 0 then return 1
    else if n = 1 then return $x$
    $y = $ POWER3$(x \cdot x, \lfloor n/2 \rfloor)$
    if n even then return $y$
    else return $x \cdot y$
```

Goal

Reduce general case to be in terms of the base case.

Solution

$$\begin{aligned} T(n) &= b + c \cdot \log n \\ &= O(\log n) \end{aligned}$$

- Cost when $n = 1$:
  - $T(1) = b$ for constant b
- We have that:

$$\begin{aligned} T(1) &= b + c \cdot \log 1 \\ &= b \end{aligned}$$

- Cost for general case:
  - $T(n) = T(n/2) + c$ for constant c
- We have that:

$$\begin{aligned} T(n/2) + c &= b + c \cdot \log(n/2) + c \\ &= b + c \cdot (\log n - \log 2) + c \\ &= b + c \cdot \log n \\ &= T(n) \end{aligned}$$

Recurrence and complexity

- Recurrence relation:

$$\begin{aligned} T(n) &= T(n/2) + c \\ T(1) &= b \end{aligned}$$

- Algorithmic example?
  - Binary search
- Asymptotic complexity?
  - $O(\log n)$

Recurrence and complexity

- Recurrence relation:

$$\begin{aligned} T(n) &= T(n-1) + c \\ T(1) &= b \end{aligned}$$

- Algorithmic example?
  - Linear search
- Asymptotic complexity?
  - $O(n)$

Recurrence and complexity

- Recurrence relation:

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \cdot n \\ T(1) &= b \end{aligned}$$

- Algorithmic example?
  - Merge Sort
- Asymptotic complexity?
  - $O(n \log n)$

Recurrence and complexity

- Recurrence relation:

$$\begin{aligned} T(n) &= T(n-1) + c \cdot n \\ T(1) &= b \end{aligned}$$

- Algorithmic example?
  - Selection Sort
- Asymptotic complexity?
  - $O(n^2)$

Recurrence and complexity

- Recurrence relation:

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + c \\ T(0) &= b \end{aligned}$$

- Algorithmic example?
  - Naive recursive Fibonacci
- Asymptotic complexity?
  - $O(2^n)$

Reading

- Course Notes: Sections 1.2, 1.3 and 1.4, Chapter 2
- Additional resources (not required, but also not necessary covering all topics). Contents related to this seminar and recap of previous units can be found in standard algorithms books such as:
  - CLRS: Chapters 3 and 4
  - KT: Chapters 2 and 5
  - Rou: Chapters 1 to 4

Concluding remarks

- This unit demands your efforts from Week 1.
- The Divide-and-Conquer algorithm design paradigm can be useful for reducing time complexity.
- Coming up next:
  - Analysis of algorithms
  - Non-comparison based sorting (Counting Sort, Radix Sort)
- Preparation required before the next week:
  - Revise computational complexity covered in earlier units.
  - Complete Preparation 1 (in your own time to self-assess your prerequisite knowledge).
  - Complete Preparation 2 before your Applied Class of Week 2.