# COMP2119

## Notes for HKU · Fall 2025

**Author:** Jax

**Contact:** enhanjax@connect.hku.hk

MORE notes on my website!

# Contents

# 1 Recursion

Solves a problem by breaking it down into smaller instances of the same problem. If you can solve a smaller instance of the problem, you can solve a larger one.

1. **Base Case(s)**: The simplest instance(s) of the problem that can be solved directly without further recursion.
2. **Recursive Case(s)**: More complex instances of the problem that are solved by breaking them down and making recursive calls.

## 1.1 Recurrence relation

**Recurrence relation / equation**

A mathematical equation which is defined in terms of itself.

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n-1)+1 & n > 1 \end{cases}$$

**Solve a recurrence relation - subsitution method**

We "solve" a recurrence relation by finding a closed-form expression for $f(n)$ in terms of $n$.

1. Expand the relation as $f(n)$ to 3 levels
2. Observe the pattern of $f(n)$
3. Generalize the pattern in terms of the level $k$
4. Using the base case to define $k$ in terms of $n$
5. Subsitute $k$ back into the generalization

4.1

## 1.2 Mathematical induction

**Mathematical induction**

A method of mathematical proof that proves a statement for all natural numbers.

1. **Base case**: Prove the statement for the first natural number in the statement's range.
2. **Inductive step**: Prove that if the statement is true for $n$, then it is also true for $n+1$.

4.2

# 2 Algorithm Analysis

We usually determine the efficiency of an algorithm by analyzing its time and space complexity.

**Size of input**

An **input** is a data structure that is given to an algorithm to solve a problem. The **size** of an input is the number **n** of elements in the input.

## 2.1 How do we assess time complexity?

When we analyize time complexity, we are interested in the efficiency of an algorithm as a function of the size of the input.

As different machines have varying speeds of processors, time is not a good fit for measuring algorithm efficiency.

We can consider the number of operations an algorithm performs relative to the size of the input, which will isolate the algorithm from the performing machines.

However, different operations have varying efficiencies, so instead we consider the **growth rate** of the **total number of operations** as a function of the input size. The analysis of such is called **Asymptotic analysis**.

**Time complexity**

The **rate of growth** of the **total number of operations** an algorithm performs relative to **size of input**.

**Space complexity**

The **rate of growth** of the **total amount of memory** an algorithm uses relative to **size of input**.

## 2.2 Asymptotic notation

**Growth rate / Complexity: The basis of asymptotic notations**

We are defining a certain function $\mathbf{T(n)}$ to represent the **total number of operations** with respect to **n** (the size of the input).
When we say $T(n)$ is $A$ of $(g(n))$ (or $T(n) \in A(g(n))$), we are saying that the **growth rate** of $T(n)$ is bounded by $A(g(n))$ under a specific mathematical inequality defined by $A$.

### Growth rate functions

We use the likeness of $g(n)$ to describe the complexity of $T(n)$. The following gives the common growth rate functions (increasing order of growth):

$$1 < \sqrt{n} < \log n < n < n \log n < n^2 < n^3 < 2^n < n! < n^n$$

Note that the higher the growth rate, the more complex the algorithm.
Note the following growth rate equivilencies:

- $\log(n!) = \Theta(n \log(n))$

### General definition of asymptotic notations

For non-strict boundaries:

$$T(n) \in A(g(n)) \text{ iff } \exists\ c > 0 :$$

$$T(n) \simeq c \cdot g(n)\ \forall\ n \geq n_0 > 0$$

For strict boundaries:

$$T(n) \in a(g(n)) \text{ if } \forall\ c > 0\ \exists\ n_0 \geq 0 :$$

$$T(n) \simeq c \cdot g(n)\ \forall\ n \geq n_0$$

Note that the $\in$ is interchangable with $=$. The meaning does not really matter.

---

4.3 4.4

### Definition of all asymptotic notations

| Notation | Condition | Asymptotic boundary | Name |
|---|---|---|---|
| $O(g)$ | $T(n) \leq c \cdot g(n)$ | Upper | Big O |
| $\Omega(g)$ | $T(n) \geq c \cdot g(n)$ | Lower | Big Omega |
| $\Theta(g)$ | $c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$ | Tight | Big Theta / Order of growth |
| $o(g)$ | $T(n) < c \cdot g(n)$ | Strictly upper | Little o |
| $\omega(g)$ | $T(n) > c \cdot g(n)$ | Strictly lower | Little omega |

### Identifying asymptotic growths

*Non-strict*: Identify **the term with the highest growth rate** in $T(n)$ would be $g(n)$.
*Strict*: Choose the **next / previous** growth rate function with reference to the list depending on if it's upper / lower.
You can use the same logic to determine if some $f(n)$ is $A/a$ of $g(n)$.
Note that as the defintions are specified by an **inequality**, there could be **multiple satisfying** $g(n)$ for the same $T(n)$ and all of them are valid. However, keep in mind the only useful $g(n)$ for analysis would be closest to the condition boundary.

---

4.5

## 2.3 Time complexity of code

**Operation of time complexities**

By definition:

$$O(g) + O(g) = k \times O(g) = O(g)$$

$$O(g) \times O(g) = O(g \times g)$$

$$O(g_1) + O(g_2) = O(g_2), \quad g_2 > g_1 \text{ in terms of order of growth}$$

Our goal is to examine how the runtime grows with respect to the input size. Consider the following example:

```
0                                           # Times ran 0()
1  def find_max(arr):                       # 1
2      if len(arr) == 0:                    # 1
3          return None                      # 1
4      max_val = arr[0]                     # 1
5      for num in arr:                      # n
6          if num > max_val:                # n
7              max_val = num                # n
8      return max_val                       # 1
```

It should be acknowledged that each operation / line would take different amounts of time to compute. However, as we are interested in the runtime $T(n)$'s growth rate, we can treat all operations to take 1 unit of time.

Hence, $T(n) = O(1) + O(1) + O(1) + O(1) + O(n) + O(n) + O(n) + O(1) = O(n)$

**Typical growth rates and common occurance**

| **Growth** | **Occurance** |
|---|---|
| 1 | Statments that run a set number of times with no respect to the input size |
| $\log n$ | Algorithms that solve a big problem by transformation into a series of smaller problems, cutting the problem size by some constant fraction at each step. |
| $n$ | "for" loops that has $n$ amount of iterations per program run |
| $n \log n$ | Algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. |
| $n^k$ | Nested "for" loops that has n amount of iterations per program run |

# 3 Data Structures

The purpose of a programme is to process data *efficiently*. A data structure is a way to group & store data in a computer, and have pros and cons which make them suitable for different scenarios.

**Operations**

A data structure has the following operations:

- **Access**
- **Insertion & deletion** (start, end, middle)
- **Search**
- **Sorting**
- **Merging**

## 3.1 Overview of data structures

**Linear vs. Non-linear**

A Linear Data Structure has each item only relates to it's front and back item.
A Non-linear Data Structure has each item can relate to multiple other items.

**Abstract vs. Concrete data types**

**Abstract data types** (ADT) is a concept which defines the operations that can be performed on a data structure, without specifying the implementation details.
**Concrete data types** (CDT) is the implementation of the ADT.

The following is an overview of the ADTs that is dicussed in the course:

- Core implementations
  - **Arrays**
  - **Nodes / linked list**
- Structure types
  - Linear structures
    - **Linked list**
    - **Stack**
    - **Queues**
  - Non-linear structures
    - **Trees**
    - **Graphs**
    - **Hash tables**

## 3.2 Core implementations

The following ADTs can be used to implement other data structures that will be introduced in the following sections.

**Array**

Set number of items stored in *adjacent* memory locations. Identified by the first item.

- **Access:** Simple access by index.
- **Insertion & deletion:** Slow, as it involves shifting the surrounding items and creating a new array.

**Node**

A node is an item, which can be connected to other nodes by **edges**. In order to access an item in a node-based data structure, we have to traverse the nodes by following the edges.
Nodes can be directional (one-way access) or non-directional (two-way access).

**Linked list**

Nodes that are connected to the next node. Identified by the first node.

- **Access:** Slow, as we have to traverse the list from the start.
- **Insertion & deletion:** Fast, as we only need to change the pointers of the neighbouring nodes.

## 3.3 Linear structures

**Push pop & access**

Stacks and queues are concepts that extend from a linear list of items.
They have the following characteristics:

- **Only one item can be accessed** at an instance of the data structure.
- **Items cannot be inserted or removed** freely.

In a stack or a queue, **push (in)** means adding an item, and **pop (out)** means removing an item. Only the item that will be **popped** can be accessed.

**Stacks & Queues**

- **Stack**: *First In, First Out (FIFO).*
- **Queue**: *Last In, First Out (LIFO).*

## 3.4 Non-linear structures

### 3.4.1 Trees

**Trees**

Nodes which has **one parent** and can connect to **multiple children**.

# 4 Examples & questions bank

**Example 4.1** *Solving a recurrence relation with subsitution method 1.1*

$$
\begin{aligned}
(lvl\ 1): \quad f(n) &= 2f(n-1)+1 \\
(lvl\ 2): \quad f(n-1) &= 2f(n-2)+1 \\
\uparrow \quad f(n) &= 2(2f(n-2)+1)+1 \\
&= 4f(n-2)+2+1 \\
(lvl\ 3): \quad f(n) &= 2(2(2f(n-3)+1)+1)+1 \\
&= 8f(n-3)+4+2+1
\end{aligned}
$$

$$
(lvl\ k): \quad f(n) = 2^k f(n-k) + 2^k - 1
$$

$$
\begin{aligned}
f(1) = f(n-k) = 1 &\implies k = n-1 \\
\therefore f(n) &= 2^{n-1}f(1) + 2^{n-1} - 1 \\
&= 2 \times 2^{n-1} - 1 \\
&= 2^n - 1
\end{aligned}
$$

**Example 4.2** *Proving with MI 1.2*

$$
\begin{aligned}
&\textit{Prove } f(n) = 1\cdot 2^1 + 2\cdot 2^2 + \cdots + n\cdot 2^n = (n-1)\cdot 2^{n+1} + 2 \\
&\textit{Base case: } n = 1 \\
&\textit{LHS: } f(1) = 1\cdot 2^1 = 2 \\
&\textit{RHS: } (1-1)\cdot 2^{1+1} + 2 = 0\cdot 2^2 + 2 = 2 \\
&\textit{Inductive step: Assume } f(n) \textit{ holds true } \forall n \le k: \\
f(k+1): \quad &\textit{LHS } = f(k) + (k+1)\cdot 2^{k+1} \\
&= [(k-1)\cdot 2^{k+1} + 2] + (k+1)\cdot 2^{k+1} \\
&= (k-1)\cdot 2^{k+1} + (k+1)\cdot 2^{k+1} + 2 \\
&= (2k)\cdot 2^{k+1} + 2 \\
&= k\cdot 2^{k+2} + 2 \\
&= ((k+1)-1)\cdot 2^{(k+1)+1} + 2 \\
&= \textit{RHS} \\
\therefore &\textit{ Statement holds true } \forall\ n > 0
\end{aligned}
$$

**Example 4.3** *Disproving Big O notation 2.2*

*Consider $T(n) = 3n^3 + 1$, to show that $T(n) \neq O(n^2)$:*

$$\begin{aligned} 3n^3 + 1 &\leq c \cdot n^2 && \forall\ n \geq n_0 \\ 3n^3 + 1 &\leq c \cdot n^2 && \forall\ n \geq 2 \\ 3n + \frac{1}{n^2} &\leq c && \forall\ n \geq 2 \end{aligned}$$

*As this expression cannot hold true for all $n \geq 2$ for a specific $c$ value, we can conclude that $T(n) \neq O(n^2)$.*

*(Example: if $c = 10$ the equation does not hold when let's say $n = 100$)*

**Example 4.4** *Proving Little o notation 2.2*

*To show that $T(n) \in o(n^4)$:*

$$\begin{aligned} 3n^3 + 1 &< c \cdot n^4 && \forall\ n \geq n_0 \\ \frac{3}{n} + \frac{1}{n^4} &< c && \forall\ n \geq n_0 \end{aligned}$$

*As this express can hold true for any $c > 0$ with sufficiently large $n_0$, we can conclude that $T(n) \in o(n^4)$.*

*(Example: if $c = 1$ the equation holds with $n_0 = 100$)*

**Example 4.5** *Identifying asymptotic growths 2.2*

*Example 1:* $T(n) = 3n^3 + 1 \implies O(n^3)\ \Omega(n^3)\ \Theta(n^3)\ o(n^4)\ \omega(n^2)$

*Example 2:* $n^2 \in O(2^n)$ *($n^2 \leq 2^n$ order of growth. This is one of the many satisfying $g(n)$. The useful $g(n)$ would be: $n^2 \in O(n^2)$)*

*Example 3:* $n \log n \in \Omega(e^{\log n})$ *($n \log n \geq n$)*