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.

Tower of Hanoi

Goal: Move n disks from peg $A \implies C$ using peg B.
Rules: Only one disk can be moved at a time, and a disk can only be placed on top of a larger disk.
We define the problem as $H(num, from, to, via)$:

1. Move top $n-1$ disks $A \implies B$ via C: $H(n-1, A, B, C)$
2. Move 1 disk $A \to C$
3. Move top $n-1$ disks $B \implies C$ via A: $H(n-1, B, C, A)$

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

Relevant mathematics

1. Geometric series: $S_n = a(1 + r + r^2 + \cdots + r^{n-1}) = a\frac{r^n - 1}{r-1}$
2. Arithmetic series: $S_n = n(\frac{a_1 + a_n}{2}), \quad n = \text{number of terms}, a = \text{first / last term}$
3. Factorial: $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
4. Falling factorial: $n^{\underline{k}} = n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}$
5. Logarithm: $a^k = b \implies k = \log_a b$
6. Logarithm property: $k^{\log a} = a^{\log k}$

The master theorem

The master theorem can give us the order of growth (Big Theta) of a recurrence relation.
For $T(n) = a \cdot T(\frac{n}{b}) + f(n), \quad T(1) = c, \quad a, c > 0,\ b > 1,\ f(n) \in \Theta(n^k),\ d \geq 0$, we have:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

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**: Assume the statement is true for n, then prove 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 $\mathbf{T}(\mathbf{n})$ 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.

General definition of asymptotic notations

For non-strict boundaries ($\leq \geq$)

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

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

For strict boundaries: $<>$

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

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

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

To prove non-strict boundaries, we need to show the existence of constants c, n_0 that satisfies the inequality. 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

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:

- $\log(n!) = \Theta(n \log(n))$ (no need to prove)
- $\Theta(\log_2 n) = \Theta(\log_{10} n)$ as $\log_2 n = \frac{\log n}{\log 2} = c \times \log n$

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.
If the function functuates, there is not a consistent growth rate unless a boundary is specified.

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:

```python
                                        # Times ran O()
def find_max(arr):                      # 1
    if len(arr) == 0:                   # 1
        return None                     # 1
    max_val = arr[0]                    # 1
    for num in arr:                     # n
        if num > max_val:               # n
            max_val = num               # n
    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 subproblems, 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. $O(1)$
- **Insertion & deletion:** Slow, as it involves shifting the surrounding items and creating a new array. $O(n)$

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. $O(n)$
- **Insertion & deletion:** Fast, as we only need to change the pointers of the neighbouring nodes. $O(1)$

Reversing a linked list

We can reverse a linked list in $O(n)$ time by iterating through the list and reversing the direction of the edges using three pointers:

- Make current node point to previous node
- Save previous node
- Move to next node
- Repeat

Example implementation: 4.6

Hare and tortoise algorithm

We can find the middle node of a linked list in $O(n)$ time by using two pointers:

- Move one pointer one node at a time
- Move the other pointer two nodes at a time
- When the fast pointer reaches the end, the slow pointer will be at the middle

Example implementation: 4.7

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 / queue, **push / enqueue (in)** means adding an item, and **pop / dequeue (out)** means removing an item. Only the **out** item can be accessed.

Stacks & Queues

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

Accessing out items is $O(1)$. Accessing / operating on other items is $O(n)$.

3.4 Non-linear structures

3.4.1 Graph

Graph

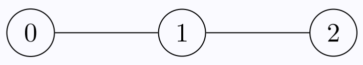
Nodes which can connect to multiple other nodes. (Allow loops)

Implementations:

Adjacency matrix

For a node with n nodes, we have a $n \times n$ matrix. If nodes i, j is connected, then the element M_{ij} is 1. If the connection edge is non-directional, then $M_{ij} = M_{ji} = 1$.

- **Space:** $O(n^2)$ - better for dense graphs (more edges).
- **Access (check):** $O(1)$ for determining if there is a connection between two nodes.
- **Access (process)**: $O(n^2)$ for processing all edges of the graph.

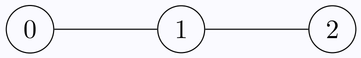


$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Adjacency list

For a node with n nodes, we have an array with n items. Each item in the array is a list of nodes that the current node is connected to.

- **Space:** $O(n + e)$ - better for sparse graphs (less edges).
- **Access (check):** $O(e)$ for determining if there is a connection between two nodes.
- **Access (process)**: Better for sparse graphs as less than n^2 elements.



$[[2], [1, 3], [2]]$

Traversal:

Breadth-first search (BFS)

Visits nodes at the increasing order of distance from the starting node.

- Queue starting node
- Repeat until queue is empty, visit queue node
- Add unqueued adjacent nodes to queue and mark as queued.

Example implementation: 4.8

Depth-first search (DFS)

Visit the furthest nodes and backtrack.

3.4.2 Trees

Trees

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

3.4.3 Hash tables

Hash table

Dynamic set of key-value pairs, where items are stored in an array with **fixed size**. To access an item, the index is found by a hash function.

A **collision** occurs when $h(k_1) = h(k_2)$. We deal with it by different methods.

- **Access:** $\Theta(1)$, $O(n)$
- **Insertion & deletion:** $\Theta(1)$, $O(n)$

Where h is the hash function and k is the key.

Note: Hash tables are a specific implementation of a dictionary. Each item in the hash table table is called a bucket or a slot

Load factor

$$\alpha = \frac{n}{m}$$

Where n is the number of items, and m is the size of the table.

Hash function

A hash function is a function that takes a key and returns an index. It should be in relation to the **size of the table** m.

The most basic hash function is the **division method**:

$$h(k) = k\%m$$

A good value of m should be a **prime number** that is not close to a power of 2.

Collision & resolution

When $h(k_1) = h(k_2)$, we have a **collision**. We can resolve it by:

- **Chaining**: Store items in the same slot by **appending** them to the linked list in the slot.
- **Open addressing**: Store items in another slot.

For opening addressing, we have the following implementations and the slot where the item is stored is:

Method	**Slot**
Linear probing	$A[h(k) + i]$
Quadratic probing	$A[h(k) + i^2]$
Double hashing	$A[h(k) + i \cdot h'(k)]$

Where i is the number of collisions.

Primary clustering

Primary clustering occurs in **open-addressing** hash tables that use **linear probing** for collision resolution, which can lead to clusters of occupied slots.

This can lead to **longer search times** for items that are stored in the same cluster.

Finding the average number of slots inspected for unsuccessful search

General steps:

1. For each slot, assume a collision and resolve until an empty slot is found, and count the steps (no. of slots inspected). We usually count nil slots as well.
2. Sum and divide by m

Notes:

- For double hashing, we **must consider the second hash function**. In other resolution methods, we already assumed a collision so the first hash function does not matter. Refer to example for detailed steps.

Example: 4.9

Average number of slots inspected for successful search

General steps:

1. For each slot, count the number of items in the slot.
2. Sum and divide by m

4 Examples & questions bank

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

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

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

Example 4.2 *Proving with MI 1.2*

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

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

Example 4.6 *Example implementation of reversing a linked list 3.2*

```
def reverse_linked_list(head):
    prev = None
    current = head
    while current:
        next = current.next
        current.next = prev
        prev = current
        current = next
    return prev
```

Example 4.7 *Example implementation of finding the middle node of a linked list 3.2*

```python
def find_middle_node(head):
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Example 4.8 *Example implementation of BFS 3.4.1*

```python
def bfs(mtx, start):
    q = Queue()

    q.enqueue(start) # queue start node
    qd = [start]

    while not q.isEmpty():
        cur = q.dequeue() # visit node
        print(cur)
        for x in mtx[cur]:
            if x not in qd:    # if not queued
                q.enqueue(x)  # put in queue
                qd.append(x)   # mark as queued
```

Example 4.9 *Example of finding the average number of slots inspected for unsuccessful search of built hash table with resolution method double hashing* **??**

$m = 5, \quad h(k) = k \mod 5$ *with double hashing* $f(i) = i \cdot h'(k), \quad h'(k) = 2 - (k \mod 2).$

Slot	***Value***
0	*79*
1	
2	*22*
3	
4	*54*

num \Slot	*0*	*1*	*2*	*3*	*4*
count($k \mod 2 = 0$)	*4*	*1*	*3*	*1*	*2*
count($k \mod 2 = 1$)	*2*	*1*	*2*	*1*	*3*

Therefore the average is $\frac{(4+1+3+1+2)+(2+1+2+1+3)}{5\times 2} = 2$