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

7.1

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 $n + 1$.

7.2

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.

Back to top

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. 7.3 7.4 7.5

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):

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

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

- $\log(n!) = \Theta(n \log(n))$ (prove not needed, related to lower bound of sorting algorithms)
- $\Theta(\log_2 n) = \Theta(\log_{10} n)$ 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.

7.6

2.2.1 Big Theta of resursive relations

When tasked to find find the asymptotic notation of a recursive relation, we are required to give the Big Theta notation of the relation.

The straight forward way is to find the closed-form expression and identify the growth rate. However, this is not always possible.

Useful growth rates

- **Sum of square series**: $\sum_{i=1}^{k} f(n,i)^2 \in \Theta(n^3)$
- **Sum of series**: $\sum_{i=1}^{k} f(n,i) \in \Theta(n^2)$

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:

```
                                    # Times ran 0()
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 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**
- **Search**

We use Ω, Θ, O to describe the best, average and worst case time complexity of an algorithm.

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 / Direct access table

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), and can have weights (values).

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: 7.7

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: 7.8

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 $O(n)$.

Priority queues are a special type of queue, know more about them in the Heaps section.

3.4 Non-linear structures

3.4.1 Graph

Graph

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

- **Directed**: Only one direction is allowed.
- **Undirected**: Both directions are allowed.

- **Weighted**: Each edge has a weight.
- **Unweighted**: Each edge has no weight.

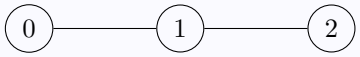
- **Number of nodes:** n
- **Number of edges:** m
- **Degree of node:** d Number of edges connected to the node.
- **Maximum degree of graph:** d_{max} Maximum degree of all nodes.

Implementations:

Adjacency matrix

A $n \times n$ matrix. $A[i]$ are the list of nodes that node i is connected to.

- **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.

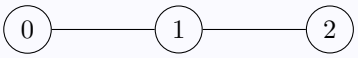


$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Adjacency list

An array with n items, each item in the array is a list of nodes that the current node is connected to.

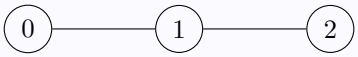
- **Space:** $O(n + m)$ - better for sparse graphs (less edges).
- **Access (check):** $O(m)$ for determining if there is a connection between two nodes.
- **Access (process)**: Better for sparse graphs as less than n^2 elements.



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

Edge list

An array with m items, each item is a tuple of the two nodes that the edge is connecting.



$[(0, 1), (1, 2)]$

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: 7.9

Depth-first search (DFS)

Visit the furthest nodes and backtrack.

3.4.2 Heaps

Priority queues

A priority queue is a queue where each item has a priority, and the item with the highest priority is accessed / removed first. The most common implementation is the **heap**.

- **Access:** $O(1)$ (size, peek)
- **Insertion:** $O(\log n)$ (enqueue, dequeue)

Heap

A (max) heap is a binary tree where each node has a value greater than or equal to its children. The root node has the highest value.

- **Max heap**: The root node has the highest value.
- **Min heap**: The root node has the lowest value.

Thouhg a tree-based data structure, we usually implement heaps as arrays, which is useful such as in the case of heap sort.

- Root at index 0
- Left child at $2i + 1$
- Right child at $2i + 2$
- Parent at $(i - 1)/2$

Heapify and building heaps

We can build (max) heaps using **heapify**, which converts a binary tree into a heap, given that its children are already heaps. This takes $O(\log n)$ time.

1. If largest node is not the root node, swap the largest node and the root
2. Heapify the new root node recursively

Due to the limitation of heapify, we must build the heap from the **bottom up**. Heapifying leaf-nodes does nothing. Observe that the last non-leaf node is at index $n/2 - 1$, we just have to heapify all nodes before that index to build the complete heap. As items further down the array is deeper in the heap:

1. Iterate through the array from the last non-leaf node to the root node.
2. Heapify the current node.

This takes $O(n)$ time.

Finding k largest and smallest elements

Heaps are especially useful for finding the largest and smallest elements. To find the largest k elements:

1. Put first k elements in a min-heap.
2. For each remaining element, peek the min-heap and check if the current element is greater than the peeked element. If so, remove the peeked element and add the current element to the min-heap.

Merging sorted arrays

We can also use heaps to merge sorted arrays.

1. Put the first element of each array into a min-heap.
2. Pop the min-heap to the result, and add the next element of the array that the popped element came from.
3. Repeat until all elements are added.

Time complexity: $O(n \log k)$ where n is the total number of elements and k is the number of arrays. We had inserted n elements into the heap, and each insertion takes $O(\log k)$ time.
Space complexity: $O(n + k)$ as the heap have at most k elements, and result array has n elements.

3.4.3 Trees

Trees

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

Definitions

- **Root**: The top node of the tree.
- **Internal node**: A node with children.
- **Leaf / external node**: A node with no children.
- **Sibilings**: Nodes with the same parent.
- **Height**: The number of edges from the root to the furthest leaf.
- **Depth**: The number of edges from the root to the current node.
- **Sub tree**: A tree that is a child of a node.
- **Degree of node**: The number of immediate children of the node.
- **Degree of tree**: The maximum degree of all nodes.
- **(Proper) ancestor**: Node that has the current node as a child. If not proper, a node can be an ancestor of itself.
- **(Proper) Descendant**: Node that has the current node as a parent. If not proper, a node can be a descendant of itself.

3.4.4 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**:

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

A good value of $prime$ 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: 7.10

Finding the 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

Averages and spaces

For chaining:

- $S_n = 1 + \frac{a}{2}$
- $U_n = \alpha$
- Space $= mp_s + n(p_s + e_s)$

For open addressing:

- $S_n = \frac{1}{2}(1 + \frac{1}{1-\alpha})$
- $U_n = \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
- Space $= ne_s$

Where S_n is the average number of slots inspected, U_n is the average number of slots inspected for successful search, m is the size of the table, n is the number of items, p_s is the space for a pointer, e_s is the space for an element, and α is the load factor.

4 Searching

When we search for data, we want to find the data as quickly as possible.

Searching through unorganized data must be $O(n)$, as we have to look at every element to find the data. This is called **linear / sequential search**.

This could be **not true for organized data**, where we can **gain information** about the data **without** looking at every element, such as with a sorted list.

We will focus on searching algorithms on a **sorted list** in this section.

Jump search

Works by:

- Access linearly every floor($\sqrt{n}$) elements
- If the element is greater than the target, go back to the previous jump
- Linearly search the elements in the jump

Amount of comparisons $= \frac{n}{\sqrt{n}} + \sqrt{n} - 1$
Time complexity: $O(\sqrt{n})$

Binary search

Works by:

- Start at the middle of the list
- If the element is greater than the target, search the left half
- If the element is less than the target, search the right half

Time complexity: $O(\log n)$
Remember that the middle must be recalculated each time.

Interpolation search

An improved version of binary search. Instead of searching the middle, we search the **interpolated position** (probe) based on the target.

$$\text{probe} = \text{low} + \frac{(\text{high} - \text{low})}{(\text{A[high]} - \text{A[low]})} \times (\text{target} - \text{A[low]})$$

Time complexity: $\Theta(\log \log n) O(n)$
Note: This is only effective when the data is **evenly distributed** (linear). Otherwise, it is not effective, such as when the data is exponential, resulting in $O(n)$ time complexity.

5 Tree searching

We can use a tree to search for data. BST and AVL trees guarentee $\Theta(\log n)$ time complexity, as we don't have to access every element to gain information about them.

This is built upon the idea of binary search.

5.1 Binary search tree

Binary search tree (BST)

A binary tree (two child nodes per node) that satisfies the following properties:

- Left node: key $\leq$ root.key
- Right node: key $>$ root.key

The height of a BST is $\log n$.

Time complexities

Consider the height of a BST:

- Min height: $\log_2 n + 1$
- Max height: n (All nodes are in a straight line)

We can then conclude the time complexities for insert, search and delete is: $\Theta(\log n)O(n)$

Traversal methods

The following are three recursive methods to traverse a tree:

- Pre-order: root→left→right
- In-order: left→root→right
- Post-order: left→right→root

Where you access the root and **recursively** trasverse the left and right subtrees. When we consider trees, we always think recursively. These are all **DFS** methods.

To search for a node in a BST, we can simply captialized on the properties of the tree:

1. Start at the root
2. If the key is less than the root, go left
3. If the key is greater than the root, go right
4. Return if equal

Remember, we always consider trees recursively.

5.2 Balancing trees

The balance factor

$$B(n) = \text{left.height} - \text{right.height}$$

A perfectly balanced tree has $B(n) = 0$.

AVL tree

A self-balancing binary search tree where $|B(n)| \leq 1$.

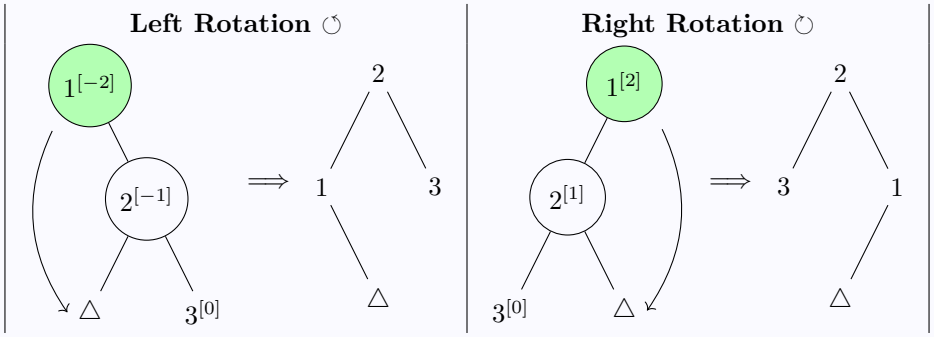
Self-balancing means that when modifying the tree, it will **rotate** the tree to *fix imbalance*.

Time complexities

The time complexities for insert, search and delete in an AVL tree is $\Theta(\log n)$.

Rotations

After inserting a node to the correct position, or deleting a node, the tree may become unbalanced.
We need to check for imbalance for each processed node and **rotate** the tree to fix it.
The following are two types of rotation:



Notice that they are **mirrored** of each other.
Consider the following conditions of balance for each processed node for rotations:

$l \leftarrow$ node.left
$r \leftarrow$ node.right
if $B(node) > 1$ **then**
  If $B(l) < 0$: rotateLeft(l)
  rotateRight(node)
end if
if $B(node) < -1$ **then**
  If $B(r) > 0$: rotateRight(r)
  rotateLeft(node)
end if

6 Sorting

We usually sort elements in accending order. A sorting algorithm can have the following proerties:

- **Stable:** Elements with the same value appear in the same order in the sorted list.
- **In place:** The algorithm uses a constant amount ($O(1)$) of extra memory.
- **Non Comparative:** The algorithm doesn't compare elements.

Lower bound (worse case) of comparison based sorting

The **minimum number of comparisons required** to sort an array of n elements in the **worse case** (can't assume sorted) is

$$\log_2(n!) \in \Theta(n \log n)$$

6.1 Algorithms

Selection Sort

For each element, find the minimum value in the subarray after it (including itself), and swap it with the current element.

Time Complexity: $O(n^2)$

Bubble sort

For each element with index i, for elements in array till before the element $n - i - 1$, check if current element is greater than the next, and swap if true. Repeat until no swaps are made for an element.

Time Complexity: $\Theta(n^2), \Omega(n)$

Insertion sort

For each element, for each previous element, if current element smaller than current previous element, swap them.

Time Complexity: $\Theta(n^2), \Omega(n)$

Notes: Efficient for small datasets and nearly sorted arrays. $O(n)$ for when the array is already sorted.

Back to top

Merge Sort

A divide-and-conquer algorithm that breaks the array into subarrays, sorts them, and merges them back.

- **Base case**: Array length is less than 2, return array.
- **Recursion:** Split the array into halves and (merge)sort them.
- Merge the sorted arrays using a helper function, which:
  - Create result array
  - Until one array ran out of elements, compare foremost element in each array, add smaller to list, and increment index.
  - Add remaining elements to the result array.

Time Complexity: $O(n \log n)$

Space Complexity: $O(n)$

Notes: The array is divided into halves, so the time complexity is $O(\log n)$ for the number of divisions and $O(n)$ for the merge step.

- Performs better than simple sorts.
- Requires extra space as it doesn't sort in place

Parition

The process of rearranging the array so that all elements with values less than the pivot come before the pivot, and all elements with values greater than the pivot come after it.

1. Pick index as pivot
2. Initialze swap index as -1
3. For each element, if element $<$ pivot, increment swap, and swap current element with swap index.
4. Pivot element is at $swap + 1$. Move it to the pivot index by swapping.
5. Return pivot index.

Time Complexity: $O(n)$

Back to top

Quick Sort

A divide-and-conquer algorithm that picks a pivot, **partitions** the array, and recursively sorts the subarrays.

- **Base case**: Array length is less than 2
- Get pivot index by paritioning array (randomly)
- **Recursion:** Split and sort the sub-arrays according to pivot index.

Time Complexity: $O(n^2), \Theta(n \log n)$
Space Complexity: $O(\log n)$ (recursion call stack)
Notes: The time complexity depends on the pivot selection. Worst case is $O(n^2)$ when the pivot is the **smallest or largest** element, as the array is not divided.
The pivot can be selected in different ways:

- Randomized quicksort: Randomly select a pivot.
- Median of three: Select the median of the first, middle, and last element.
- First / middle / last element: Select the first, middle, or last element as the pivot.

If we choose **randomized quicksort**, the expected time complexity must be $O(n \log n)$, as we don't expect to choose the largest / smallest element every parition.

Heap sort

Call *create_max_heap* on the array, then for each element from the right, swap the first element with the current element, then call *heapify* on the array excluding the current element.
Time Complexity: $O(n \log n)$.
Notes:

- Both functions convert an array into a max heap, where the **first element must be the largest**. This is why we swap the first element with the current element.
- *create_max_heap* is $O(n)$, and *heapify* is $O(\log n)$, which is ran $n - 1$ times for each element.
- *Heapify* is used to maintain the heap property, so it is more efficient as it assumes the array is already a heap.

Counting sort

A stable, non-comparative sorting algorithm.

1. Create a new counter array of length k (max element in array / number of distinct elements).
2. For each element, increment the counter array at the index of the element.
3. Then, for each counter item, add their previous value to the current value.
4. Create a new result array of length n.
5. For each element in the original array, add it to the result array at the index of the counter array, then decrement the counter array.

Time Complexity: $O(n+k)$

Space Complexity: $O(n+k)$, as we need a counter array of length k and a result array of length n.

Notes: The algorithm is stable as it loops through the original array at the end in order.

Radix sort

Find the maximum number of digits d in the array, then sort the array using **exp** counting sort for each digit.

Instead of using the element value as the key, we use the ith digit (0-based) by $n//10^i \ \% \ 10$.

Time Complexity: $O(d(n+k))$

Space Complexity: $O(n+k)$

Notes: The algorithm is stable as counting sort is stable. Radix sort works best for arrays with elements $0, 1, 2, ..., b^d$ with fixed length d.

6.2 Summary

Algorithm	Average Time	Special Time	Space	Stable	Use when	Ex
Selection Sort	n^2	-	1	N	-	e
Bubble Sort	n^2	$\Omega(n)$	1	Y	Nearly sorted data	e
Insertion Sort	n^2	$\Omega(n)$	1	Y	Nearly sorted data	e
Heap Sort	$n \log n$	-	1	N	Limited space	e
Merge Sort	$n \log n$	-	n	Y	Stable $n \log n$	e
Randomized Quick Sort	$n \log n$	-	$\log n$	N	Best performance	e
Quick Sort	$n \log n$	$O(n^2)$	$\log n$	N	-	e
Counting Sort	$n+k$	-	$n+k$	Y	Small range of integers	e
Radix Sort	$d(n+k)$	-	$n+k$	Y	Fixed-length integers	e

6.3 K-sorted arrays

k-sorted arrays are arrays where each element is at most k positions away from its sorted position.

Sorting k-sorted arrays - modified selection sort

For each element, find the minimum value within the next k elements after (including itself), and swap it with the current element.
Time complexity: $O(nk)$

Sorting k-sorted arrays - heaps

1. Put the first k elements of each array into a min-heap.
2. For each remaining element, pop the min element from heap back to array from the start, then add current to the heap.
3. Pop remaining elements to array.

Time complexity: $O(n \log k)$ time as the heap always has at most k elements, and we processed all n elements to the heap.
Space complexity: $O(k)$ as the heap always has at most k elements, and we used the original array as we process the remaining elements.

6.4 Other related non-sorting algorithms

Quickselect

A selection algorithm that selects the kth smallest element in an unordered list.

- **Base case**: One element only, return it.
- Get pivot index by paritioning array (randomly)
- **Recursive**: Check if pivot index is k, return if true
- If not, see if pivot index is less than k, then search right subarray, else search left subarray.

Time Complexity: $O(n^2), \Theta(n)$.
Notes: Because the pivot index is already sorted, it must be the kth smallest element. But if it's not, we know that the smaller elements are on the left, and larger elements are on the right. If we're unlucky and never get the pivot index as k, the base case will eventually be reached.
The algorithm can achieve $\Theta(n)$ as we only need to search one side of the array.

7 Examples & questions bank

Example 7.1 *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 7.2 *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 7.3 *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 7.4 *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 7.5 *Proving Big Theta notation 2.2*

Consider $T(n) = 4n^2$, to show that $T(n) \in \Theta(0.5n^2 + 10n + 20)$:

We know that $T(n) \leq c_2 \cdot (0.5n^2 + 10n + 20)$ for $c_2 = 8$ easily.

Consider the lower bound:

$$\begin{aligned} 4n^2 &\geq c_1 \cdot (0.5n^2 + 10n + 20) && \forall\ n \geq n_0 \\ n^2 &\geq 0.5n^2 + 10n + 20 && \forall\ n \geq n_0, c_1 = \frac{1}{4} \\ n^2 - 10n - 20 &\geq 0 && \forall\ n \geq n_0 \end{aligned}$$

Therefore we can simply take $n_0 = 10$ and $c_1 = \frac{1}{4}, c_2 = 8$ to satisfy the definition of big Theta.

Example 7.6 *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 7.7 *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 7.8 *Example implementation of finding the middle node of a linked list 3.2*

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

Example 7.9 *Example implementation of BFS 3.4.1*

```
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 7.10 *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 $\frac{(4+1+3+1+2)+(2+1+2+1+3)}{5\times 2} = 2$

Example 7.11 *Example of implementation of selection sort.* *6.2*

```
function selection_sort(array):
    for i in array:
        min_index = i
        for j in array after i:
            if current element < min element:
                min_index = j
        swap(A, i, min_index)
    return A
```

Example 7.12 *Example of implementation of bubble sort.* *6.2*

```
function bubble_sort(array, n):
    for i in 0 to n - 1:
        swapped = False
        for j in 0 to before n - i - 1:
            if current element > next element:
                swap(current, next)
                swapped = True
        if not swapped break
        # if no two elements were swapped, the array is sorted
    return A
```

Example 7.13 *Example of implementation of insertion sort.* *6.2*

```
function insertion_sort(array):
    for i in array after 0:
        for j in array before i reversed:
            if array[j] > array[i]:
                swap(array, i, j)
            else:
                break
    return array
```

Example 7.14 *Example of implementation of heap sort.* 6.2

```
function heap_sort(array):
    create_max_heap(array)

    for i from right in array:
        swap(array, 0, i)
        heapify(array from 0 to i - 1)

    return array
```

Example 7.15 *Example of implementation of merge sort.* 6.2

```
function merge_sort(array):
    function merge(A1, A2):
        result = []
        while A1 and A2:
            if A1[0] < A2[0]:
                result.append(A1.pop(0))
            else:
                result.append(A2.pop(0))
        return result + A1 + A2

    if len(array) < 2 return array

    mid = len(array) // 2
    left = merge_sort(array to mid)
    right = merge_sort(array after mid)

    return merge(left, right)
```

Example 7.16 *Example of implementation of quick sort.* 6.2

```
function quick_sort(array):
    function partition(array):
        pivot = random index from array
        swap = -1

        for j in array:
            if current element < pivot element:
                swap += 1
                swap(arr, swap, current)
```

```
        # move pivot element to its correct position
        pivot_index = swap + 1
        swap(arr, pivot_index, pivot)
        return pivot_index

    if len(array) < 2 return array

    pivot = partition(array)
    quickSort(array before pivot)
    quickSort(array after pivot)
```

Example 7.17 *Example of implementation of count sort.* *6.2*

```
function counting_sort(array):
    k = max(array) # O(n)
    counter = Array(k)
    for i in array:
        counter[i] += 1

    for i in counter after 0:
        counter[i] += counter[i-1]

    result = Array(len(array))
    for i, x in array:
        result[counter[x]-1] = x
        counter[x] -= 1

    return result
```

Example 7.18 *Example of implementation of radix sort.* *6.2*

```
function radix_sort(array, b=10):
    function countin_sort_exp(i):
        function get_key(n):
            return ith digit of n in base b (0 based)
            # (n // b**i) % b

        counter = Array(b)
        for i in array:
            key = get_key(i)
            counter[key] += 1
```

```
        for i in counter after 0:
            counter[i] += counter[i-1]

        result = Array(len(array))
        for x in array:
            key = get_key(x)
            result[counter[key]] = x
            counter[key] -= 1
        array = result

d = len(max(array))

for i from 0 to d - 1:
    counting_sort_exp(i)
```