Programming Concepts Cheatsheet

A quick reference guide to fundamental programming concepts, covering data structures, algorithms, paradigms, and more. Useful for students and experienced developers alike.

Core Concepts

Data Structures

Array	A collection of elements, each identified by an index or a key. Allows efficient access to elements by index but can be inefficient for insertions and deletions in the middle.
Linked List	A sequence of nodes, each containing data and a link to the next node. Efficient for insertions and deletions but inefficient for random access.
Stack	A LIFO (Last-In, First-Out) data structure. Common operations: push (add to the top), pop (remove from the top), peek (view the top element).
Queue	A FIFO (First-In, First-Out) data structure. Common operations: enqueue (add to the rear), dequeue (remove from the front).
Hash Table/Map	A data structure that uses a hash function to map keys to values, providing efficient key-based lookup. Can suffer from collisions.
Tree	A hierarchical data structure composed of nodes, where each node has a value and links to child nodes. Examples include binary trees, AVL trees, red-black trees.
Graph	A collection of nodes (vertices) and edges, representing relationships between nodes. Used to model networks, relationships, and connections.

Algorithms

Sorting Algorithms	Algorithms that arrange elements in a specific order (e.g., ascending or descending). Examples: Bubble Sort, Merge Sort, Quick Sort.
Searching Algorithms	Algorithms that find a specific element in a data structure. Examples: Linear Search, Binary Search.
Dynamic Programming	An optimization technique that breaks down a problem into smaller subproblems, solves them, and stores the solutions to avoid redundant computations.
Greedy Algorithms	An approach that makes locally optimal choices at each step, hoping to find a global optimum. May not always yield the best solution.
Graph Algorithms	Algorithms for traversing and analyzing graphs. Examples: Depth-First Search (DFS), Breadth-First Search (BFS), Dijkstra's Algorithm.
Divide and Conquer	An algorithmic paradigm that recursively breaks down a problem into smaller subproblems until they become simple enough to be solved directly. The solutions to the subproblems are then combined to solve the original problem.

Programming Paradigms

Paradigms Overview

Programming paradigms are styles or 'ways' of programming. They are not specific languages or tools, but influence the structure and approach to solving problems with code.

Imperative Programming

Description	Focuses on *how* to achieve a result. Uses statements that change a program's state. Relies on sequential execution of commands.
Characteristics	Uses variables, assignment statements, and control flow (loops, conditionals) to modify the program's state.
Examples	C, Pascal, Fortran

Declarative Programming

Description	Focuses on *what* result is desired, rather than *how* to achieve it. Expresses the logic of a computation without explicitly describing the control flow.
Characteristics	Avoids mutable state and side effects. Relies on expressions and functions rather than statements.
Examples	SQL, Prolog, Haskell

Object-Oriented Programming (OOP)

Description	Organizes programs around 'objects', which encapsulate data (attributes) and code (methods) that operate on that data.
Key Concepts	Encapsulation, Inheritance, Polymorphism, Abstraction
Examples	Java, C++, Python, C#

Functional Programming

Description	Treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.
Characteristics	Uses pure functions (no side effects), immutability, recursion, and higher-order functions.
Examples	Haskell, Lisp, Clojure, Scala

Design Patterns

Creational Patterns

Singleton	Ensures that a class has only one instance and provides a global point of access to it.
Factory Method	Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
Abstract Factory	Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
Prototype	Specifies the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structural Patterns

Adapter	Allows interfaces of incompatible classes to work together. Converts the interface of a class into another interface clients expect.
Bridge	Decouples an abstraction from its implementation, so that the two can vary independently.
Composite	Composes objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions uniformly.
Decorator	Dynamically adds responsibilities to an object. Provides a flexible alternative to subclassing for extending functionality.
Facade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Uses sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provides a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility	Avoids coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request. Chains the receiving objects and passes the request along the chain until an object handles it.
Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method	Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Common Algorithms

Searching Algorithms

Linear Search	Simplest search algorithm. It sequentially checks each element of the list until a match is found or the entire list has been searched. **Time Complexity:** O(n)
Binary Search	Efficient algorithm for finding an item from a *sorted* list. It repeatedly divides the search interval in half. **Time Complexity:** O(log n)
Jump Search	Like binary search, but jumps ahead by a fixed number of steps (the 'jump') and then performs a linear search within that block. Useful for large, sorted arrays. **Time Complexity:** O(√n)
Interpolation Search	An improvement over binary search for uniformly distributed data. Instead of dividing the search space in half, it estimates the position of the target value based on its value relative to the values in the search space. **Time Complexity:** O(log log n) on average for uniformly distributed data, O(n) in worst case.

Sorting Algorithms

Bubble Sort	Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. Easy to implement but inefficient for large lists. **Time Complexity:** O(n^2)
Insertion Sort	Builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. **Time Complexity:** O(n^2)
Selection Sort	Divides the input list into two parts: the sorted sublist of items which is built up from left to right at the front (left) of the list and the sublist of the remaining unsorted items that occupy the rest of the list. It repeatedly finds the minimum element from the unsorted part and puts it at the end of the sorted part. **Time Complexity:** O(n^2)
Merge Sort	A divide and conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves. **Time Complexity:** O(n log n)
Quick Sort	A divide and conquer algorithm that picks an element as pivot and partitions the given array around the picked pivot. Although its worst-case time complexity is O(n^2), its average performance is excellent in practice. **Time Complexity:** Average: O(n log n), Worst: O(n^2)
Heap Sort	Based on the heap data structure. It first builds a max-heap from the data, and then repeatedly extracts the maximum element and places it at the end of the sorted portion of the array. **Time Complexity:** O(n log n)

Graph Algorithms

Depth-First Search (DFS)	Traverses a graph by exploring as far as possible along each branch before backtracking. **Common Uses:** Path finding, topological sorting, cycle detection.
Breadth-First Search (BFS)	Traverses a graph level by level, exploring all neighbors of the current node before moving on to the next level. **Common Uses:** Shortest path finding in unweighted graphs.
Dijkstra's Algorithm	An algorithm for finding the shortest paths between nodes in a weighted graph (with non-negative edge weights). **Common Uses:** Navigation, network routing.
*A Search Algorithm**	An informed search algorithm that uses heuristics to guide its search, making it more efficient than Dijkstra's algorithm in many cases. **Common Uses:** Pathfinding, game AI.
Minimum Spanning Tree (MST)	Finds a subset of the edges that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Algorithms include Kruskal's and Prim's. **Common Uses:** Network design, clustering.