**DSA Notes**

**🗂️ 1. What is a Data Structure?**

A **data structure** is a way to **organize, store, and manage data** so that it can be used efficiently.

**🔑 Purpose:**

To structure data in memory for efficient access and modification.

**🧱 Common Data Structures:**

| **Type** | **Description** | **Example Use Case** |
| --- | --- | --- |
| **Array** | Contiguous memory elements | Storing list of items |
| **Linked List** | Nodes connected via pointers | Dynamic memory use |
| **Stack** | LIFO (Last In First Out) | Undo functionality |
| **Queue** | FIFO (First In First Out) | Task scheduling |
| **HashMap** | Key-value storage | Caching, lookup tables |
| **Tree** | Hierarchical structure (e.g., BST) | Fast searching |
| **Graph** | Nodes and edges | Maps, networks |

**🧠 2. What is an Algorithm?**

An **algorithm** is a step-by-step **procedure or formula** for solving a problem or performing a task.

**🔑 Purpose:**

To **process** data (usually stored in a data structure) in an efficient way.

**🧮 Common Algorithms:**

| **Type** | **Description** | **Works With** |
| --- | --- | --- |
| **Sorting** | Arrange data (e.g., QuickSort) | Arrays, Lists |
| **Searching** | Find elements (e.g., Binary Search) | Arrays, Trees |
| **Graph traversal** | Explore nodes (e.g., BFS, DFS) | Graphs |
| **Dynamic Programming** | Solve problems by breaking into subproblems | Arrays, Recursion |
| **Greedy Algorithms** | Make local optimal choices | Optimization problems |

🔁 You apply **algorithms** *on* **data structures**.

**🧠 3. What Are DSA Patterns?**

**DSA patterns** are **recurring problem-solving techniques** that appear across many coding problems (especially in interviews and competitive programming). Instead of memorizing individual solutions, learning patterns helps you **recognize problem types and apply strategies quickly**.

**📚 Common DSA Patterns (With Examples)**

Here are the most useful and popular ones:

**1. 🪜 Two Pointers**

Used to traverse arrays or strings with two indices.

* **Use Case:** Sorted arrays, linked lists, or strings
* **Common Problems:**
  + Pair with sum
  + Reverse a string/array in-place
  + Move zeros to end
* **Example:**

cpp

CopyEdit

// Check if array has two numbers that sum to target

bool hasPair(vector<int>& nums, int target) {

int left = 0, right = nums.size() - 1;

while (left < right) {

int sum = nums[left] + nums[right];

if (sum == target) return true;

else if (sum < target) left++;

else right--;

}

return false;

}

**2. 🔁 Sliding Window**

A dynamic window over a subset of data (usually in arrays or strings).

* **Use Case:** Find the max, min, or sum in a subarray of fixed/dynamic size.
* **Common Problems:**
  + Longest substring without repeating characters
  + Maximum sum subarray
* **Example:**

cpp

CopyEdit

// Max sum of subarray of size k

int maxSum(vector<int>& nums, int k) {

int sum = 0, maxSum = 0;

for (int i = 0; i < k; i++) sum += nums[i];

maxSum = sum;

for (int i = k; i < nums.size(); i++) {

sum += nums[i] - nums[i - k];

maxSum = max(maxSum, sum);

}

return maxSum;

}

**3. 🧮 Prefix Sum**

Build a running total to answer range sum queries in constant time.

* **Use Case:** Subarray sums, number of subarrays with given sum
* **Common Problems:**
  + Subarray sum equals k
  + Range sum query
* **Example:**

cpp

CopyEdit

// Sum from index i to j in O(1)

vector<int> prefixSum(vector<int>& nums) {

vector<int> prefix(nums.size() + 1);

for (int i = 0; i < nums.size(); i++)

prefix[i + 1] = prefix[i] + nums[i];

return prefix;

}

**4. 📚 Binary Search**

Used to find a target in sorted data or solve problems with **monotonic behavior**.

* **Use Case:** Sorted arrays, "binary search on answer"
* **Common Problems:**
  + Find element in sorted array
  + Minimum in rotated array
* **Variants:** Lower bound, upper bound, search in rotated array

**5. 🔁 Fast & Slow Pointers (Floyd's Cycle)**

Used for cycle detection in linked lists or arrays.

* **Use Case:** Linked list problems
* **Common Problems:**
  + Detect cycle in a linked list
  + Find cycle start
* **Idea:** Fast pointer moves 2 steps, slow moves 1

**6. 🧩 Backtracking**

Try all possibilities by building the solution incrementally and undoing when needed.

* **Use Case:** Combinatorics, permutations, pathfinding
* **Common Problems:**
  + N-Queens
  + Sudoku solver
  + Subsets, combinations
* **Example:** Generate all subsets

**7. ⚙️ Dynamic Programming (DP)**

Break problems into overlapping subproblems and store their results (memoization/tabulation).

* **Use Case:** Optimal substructure, overlapping subproblems
* **Common Problems:**
  + Fibonacci
  + Knapsack
  + Longest increasing subsequence
* **Variants:**
  + Top-down (recursion + memo)
  + Bottom-up (iterative)

**8. 🧵 Greedy**

Make the locally best decision at each step — assumes it leads to a global optimum.

* **Use Case:** Interval problems, optimization
* **Common Problems:**
  + Activity selection
  + Minimum number of platforms
  + Huffman coding

**9. 🌲 Tree Traversal Patterns**

Used in DFS, BFS, recursion, or iterative stack-based tree problems.

* **Use Case:** Binary Trees, N-ary Trees
* **Common Traversals:**
  + Inorder, Preorder, Postorder
  + Level-order traversal (BFS)

**10. 🔄 Union Find / Disjoint Set**

Used in connected components, graph cycles, Kruskal's MST.

* **Use Case:** Graphs
* **Common Problems:**
  + Detect cycles
  + Number of islands
  + Kruskal's algorithm

**🧠 Why Learn Patterns?**

✅ Helps recognize problem types quickly  
✅ Reduces time spent on interviews and coding challenges  
✅ Allows reusable, flexible thinking  
✅ Works across languages (Python, C++, Java, etc.)

**🚀 Pro Tip:**

👉 **Don't just memorize solutions**.  
Instead, ask:

* What is the **core idea** behind this solution?
* Which **pattern** does it belong to?
* How would I tweak it for similar problems?

**❓ Are patterns and algorithms the same?**

**No, they are not the same** — but they are closely related.

**🔍 Definitions First**

**✅ Algorithm**

An **algorithm** is a **step-by-step set of instructions** to solve a specific problem.

* 🔹 It's concrete.
* 🔹 Has a clear input/output.
* 🔹 It solves a problem directly.

📌 **Examples of algorithms:**

* **Binary Search** (to find an element in a sorted array)
* **QuickSort** (to sort an array)
* **Dijkstra’s Algorithm** (to find shortest path in a graph)
* **Floyd’s Cycle Detection** (to detect loops in linked lists)

**✅ Pattern**

A **pattern** in DSA is a **general approach** or **technique** used to solve a *category* of problems efficiently.

* 🔹 It's more abstract.
* 🔹 It’s like a reusable **template** or **strategy**.
* 🔹 It helps guide the selection or construction of the right algorithm.

📌 **Examples of patterns:**

* **Two Pointers Pattern** (used in Binary Search, reversing arrays, etc.)
* **Sliding Window Pattern** (used to solve max/min subarray problems)
* **Divide and Conquer Pattern** (used in MergeSort, QuickSort, Binary Search)
* **Dynamic Programming Pattern** (used in Fibonacci, Knapsack, etc.)

**🧠 Relationship Between the Two**

| **Aspect** | **Pattern** | **Algorithm** |
| --- | --- | --- |
| **What it is** | General strategy or template | Exact step-by-step procedure |
| **Specificity** | Broad category | Concrete, problem-specific |
| **Used for** | Guiding how to approach a problem | Solving the problem |
| **Example** | Sliding Window | Kadane’s Algorithm |
| **Reusability** | Applied to many problems | Usually solves one specific type |

**🎯 Analogy**

Think of:

* **Pattern** = Blueprint or strategy (like “always push from behind in a race”)
* **Algorithm** = Actual steps taken during a race (e.g., speed up at the last 100m, conserve energy for the start, etc.)

So:

A pattern **guides you toward** the right algorithm.

**💡 TL;DR**

❗**Patterns are strategies. Algorithms are solutions.**  
You **use patterns** to **build or select algorithms**.