# Plan to become a nice programmer

Well, I am not a good programmer. I always try to use google to find basic solution. For example, if I need to convert string into integer, I might have to use google to fund the name of the function and what I need to put in the arguments. This is bad. If a mathematician don’t know if he needs to use calculus or geometry to solve this math problem, then he is a fraud, right?

So, my plan is to memorize the techniques and functions so I can easily implement the solution quickly. It’s like when I am trying to say something, I focus on the meaning of the word, I don’t worry about how to express that meaning because I know the language.

So, the goal is to learn functions and techniques to the point where I can easily **write code as I think** with correct syntax and parameters. If there’s an error, it’s going to be about logic error, not syntax or invalid argument error.

On the other hand, it takes me hours to come up with a solution or create algorithm. I also realize that when I look at the problem, I cannot figure out which method I would use to solve this problem. Like, I don’t know if it’s an array problem, or HashMap problem. This problem changed a bit when I learned the power of HashMap. I was able to solve the “find double duplicates and anagram” problems easily and quickly. So, I need to increase my brainpower and knowledge. so I can come up with solutions more quickly and identify what type of problem am I dealing with.

First of all, I am going to use chat GPT to get all the basic functionalities and operations I can perform for arrays, HashMap, strings, and more.

Then I am going to look at 10 easy and 10 medium leetcode problem solution as well as solutions from my previous programming challenge solution, and get the juicy part of the code. For example, How to count the number of occurrence.

I am also going to take notes and write down the algorithm behind it.

**✅ 1. Build a Strong Foundation (Syntax + Patterns)**

Think of this phase like mastering grammar and vocabulary in a new language.

**📌 How to Upgrade:**

* **Pick one language to master first** (Python, Java, or C++ depending on your goals).
* **Create a "Cheat Sheet" notebook**:
  + Data types (int, str, list, dict, set, etc.)
  + Common operations (convert str to int, slice arrays, etc.)
  + File I/O, loops, conditionals
  + HashMap, sorting, searching, etc.
* **Build muscle memory** with these:
  + Write 20 small scripts **from memory** (e.g., reverse a string, count frequency, remove duplicates from array).
  + No copy-paste. Type it. Then try again without looking.
* 🔁 **Daily 20-minute drills**: Code 2-3 small problems by heart every day.

**🧠 2. Learn Problem Patterns (Thinking Like a Programmer)**

This is where real growth happens. You learn to “see” the shape of a problem.

**📌 Key Patterns to Master:**

1. **Sliding Window**
2. **Two Pointers**
3. **Fast & Slow Pointers**
4. **HashMap / HashSet**
5. **Stack / Queue**
6. **Binary Search**
7. **Backtracking**
8. **Dynamic Programming (Basic)**

**🧩 How to Practice:**

* **Group LeetCode problems** by pattern (e.g., Blind 75 or Neetcode).
* Do 5-10 problems per pattern.
* For each one:
  + Write down **"How I knew to use this pattern"**
  + Write a short summary of the algorithm
  + Memorize common edge cases

**🔎 3. Master Debugging & Logic Building**

You want to reach a point where the only bugs you face are **logic errors**, not syntax.

**📌 Practice:**

* **Deliberately break your code** and try to fix it.
* Use a **Rubber Duck Debugging** approach: explain your logic out loud (or to me).
* Build the habit of printing variables and tracing logic manually before running.

**📚 4. Create a “Code Arsenal” Notebook**

A small, personal collection of:

* Templates (DFS, BFS, Binary Search)
* Code snippets you often use (sorting with a lambda, count occurrences)
* Mistakes you made and fixed
* New tricks or patterns you found

💡 Tip: Use Notion, Obsidian, or a paper notebook for spaced repetition.

**🧱 5. Real-World Projects & Challenges**

Once your basics are strong, start applying your skills:

* Contribute to open-source (even fixing typos or testing first)
* Build a simple project with React + Firebase (if web dev interests you)
* Try coding games (like CodinGame) or [Advent of Code](https://adventofcode.com/)

**🧘‍♂️ 6. Build Mental Stamina & Speed**

This is where your plan shines: fast recall + problem classification.

* **Flashcards**: Make flashcards for functions, time complexities, problem types.
* **Mock Interviews**: Use [Pramp](https://www.pramp.com/) or practice aloud with me.
* **Time yourself**: 1 problem in 20–30 minutes max. Review what slowed you down.

**💥 Timeline (Flexible Example)**

| **Week** | **Focus** |
| --- | --- |
| 1-2 | Syntax mastery + Daily drills + String/Array problems |
| 3-4 | HashMap + Sliding Window + 10-15 LeetCode (easy-medium) |
| 5-6 | Binary Search, Stack, Queue + small project start |
| 7-8 | Mock interviews + 2 patterns + review previous problems |
| 9+ | Daily practice + real-world project + open source |

**📈 Track Your Growth**

Create a simple tracker:

* ✅ Syntax (1–10 scale)
* ✅ Problem pattern mastery
* ✅ # of problems solved
* ✅ Weekly review: “What did I learn? What did I struggle with?”

**Notes: Key Coding Patterns for Problem Solving**

1. **Sliding Window Pattern**
   * **Description**: Used to process a series of data elements (e.g., lists, strings) by looking at smaller parts (windows) that slide one step at a time.
   * **When to Use**: To find a subset of elements in a linear data structure (array, string, linked list) that satisfies a given condition (e.g., longest or shortest substring/subarray).
   * **Example Problem**: "Longest Substring with K Unique Characters" (LeetCode).
2. **Subset Pattern**
   * **Description**: Used to find all possible combinations of elements from a given set. Repetitions may or may not be allowed.
   * **When to Use**: When you need to explore all possible arrangements of elements from a set.
   * **Example Problem**: Permutations problem on LeetCode.
   * **Approach**: Build subsets iteratively, level by level, similar to BFS.
3. **Modified Binary Search Pattern**
   * **Description**: The core idea of binary search remains the same (divide the search space in half repeatedly) but is adjusted to solve specific problems.
   * **When to Use**: When dealing with sorted arrays that may have been rotated or contain duplicates.
   * **Example Problem**: "Search in Rotated Sorted Array" (LeetCode).
   * **Improvement Tip**: Understand the core binary search algorithm well and practice with Python’s bisect module.
4. **Top-K Elements Pattern**
   * **Description**: Used to select the K most important elements from a larger dataset.
   * **When to Use**: When the problem asks for the top-ranking elements from a dataset.
   * **Example Problem**: Finding the Kth largest number in an array.
   * **Approach**: Use a heap to efficiently track and remove the smallest of the largest K elements encountered.
5. **Binary Tree DFS Pattern**
   * **Description**: Depth-First Search (DFS) for binary trees explores nodes one branch at a time.
   * **When to Use**: To explore a binary tree deeply, going down each branch before backtracking.
   * **Example Problem**: "Maximum Depth of Binary Tree" (LeetCode).
   * **Approach**: Use recursion to explore left and right subtrees, updating the maximum depth when a new deeper node is encountered.
6. **Topological Sort**
   * **Description**: Used to arrange elements in a specific order based on their dependencies (often in directed acyclic graphs).
   * **When to Use**: When there are prerequisites or dependency chains between elements.
   * **Example Problem**: "Course Schedule" (LeetCode).
7. **Binary Tree BFS Pattern**
   * **Description**: Breadth-First Search (BFS) explores all nodes at the same level first, using a queue to keep track.
   * **When to Use**: When you need to explore nodes level by level.
   * **Example Problem**: "Level Order Reversal of a Binary Tree" (LeetCode).
   * **Approach**: Use a queue to manage the nodes, process them, and then add their children to the queue.
8. **Two-Pointer Pattern**
   * **Description**: Used to iterate through a sorted array with two pointers, typically for efficient one-pass solutions.
   * **When to Use**: For problems involving a sorted array where you can narrow down possibilities with two pointers.
   * **Example Problem**: "Two Sum" (LeetCode).
   * **Approach**: Start one pointer at the beginning and the other at the end of the array. Move pointers toward each other based on the sum comparison with the target.

**Final Advice:**

* To master these patterns, it's essential to have a strong understanding of data structures and algorithms (DSA).
* Practice by solving problems from LeetCode that fit these patterns.
* Consider signing up for a free DSA crash course to strengthen your foundational knowledge.

# Things to study for

String

Array

Hashmap

recursion

Heap

2 pointers

Linked List

Binary search

Sliding windows pattern

Subset pattern

Time complexity

2D arrays