Personalized Study Guide

Of course. Here is a rigorous, detailed, and personalized study guide for an AP Computer Science Principles student struggling with Data Abstraction, Strings, and Nested Conditionals.

AP Computer Science Principles Personalized Study Guide

Hello! Welcome to your personalized study guide. As an expert AP Computer Science Principles tutor, I've designed this guide to help you master the topics where you've shown weakness: **Data Abstraction**, **Strings** , and **Nested Conditionals**. This isn't just a summary; it's a deep dive designed to build a strong, lasting foundation. We will break down each concept using the official College Board CED framework, real-world examples, and targeted practice. Let's get started.

Topic 1: Data Abstraction (CED Topic 3.2)

1. Topic Overview

Imagine you're using a TV remote. You press the "Volume Up" button, and the TV gets louder. You don't need to know about the infrared signals, the circuit board inside the remote, or the specific electronic components in the TV that process the signal. You just need to know what the button does. This is the essence of **abstraction**: hiding the complex details and providing a simple interface.

In programming, **Data Abstraction** is a similar idea. It's about creating a single, simple name to represent a complex collection of data. Instead of juggling dozens of individual variables for your friends' phone numbers (`friend1_number`, `friend2_number`, etc.), you can create a single "collection" called `contact_list`. This abstraction makes your code cleaner, easier to understand, and much simpler to manage. It lets you focus on the big picture?what you want to *do* with your data?rather than getting lost in the details of *how* it's stored.

2. Deconstructing the Essential Knowledge

Let's break down the key ideas from the CED.

- **Essential Knowledge (EK) AAP-1.D.1:** *Data abstraction provides a separation between the abstract properties of a data type and the concrete details of its representation.*
- **Elaboration and Explanation:** This means we separate the *idea* of something from *how it's made*. The "abstract properties" are what we care about (e.g., a list of scores has a highest score, a lowest score, and an average). The "concrete details" are how the computer actually stores that list in memory (which we don't need to worry about).
- **Everyday Analogy:** Think of a car. The abstract properties are the steering wheel, gas pedal, and brake. You use these to drive. The concrete details are the engine, the pistons, and the fuel injection system. You don't need to be a mechanic to drive the car.
- **Code Example:** When we use a list to store high scores, we are using a data abstraction.

```
highScores ? [1050, 980, 1120]
```

The *idea* (abstraction) is "a list of high scores." The *details* (concrete representation) are how the computer organizes `1050`, `980`, and `1120` in memory using a list structure.

- **Common Misconception:** "Abstraction just makes things more complicated."
- **Correction:** It's the opposite! Abstraction hides complexity. Imagine managing a playlist of 1,000 songs with 1,000 separate variables. It would be a nightmare. Using one list variable, `myPlaylist`, is far simpler.
- **Essential Knowledge (EK) AAP-1.D.2 & AAP-1.D.3:** *Data abstractions manage complexity in programs by giving a collection of data a name without referencing the specific details of the representation.

Personalized Study Guide

Data abstractions can be created using lists.*

- **Elaboration and Explanation:** This is the core benefit. We use a **list** to group related items. We then give that list a single, meaningful variable name. This act of "naming the collection" is the data abstraction.

- **Everyday Analogy:** A "Shopping List" is a data abstraction. The name "Shopping List" refers to the entire collection of items. You don't say, "I'm going to the store to get item one which is milk, item two which is bread, item three which is eggs..." You just say, "I'm grabbing the shopping list."

- **Code Example:** A program that tracks daily steps for a week.

- Without Abstraction: `stepsMonday ? 7500`, `stepsTuesday ? 8200`, `stepsWednesday ? 6100`, ...

- With Data Abstraction: `weeklySteps ? [7500, 8200, 6100, 9100, 5400, 12000, 7800]`

The name `weeklySteps` manages the complexity of seven different values.

- **Common Misconception:** "I can just use a bunch of variables; it's the same thing."

- **Correction:** While you can store data in separate variables, you lose the ability to easily perform operations on the *collection*. With a list, you can loop through it, find the total, find the average, or add a new day's steps with a single command (`APPEND`). This is impossible with separate variables without writing much more complex code.

- **Essential Knowledge (EK) AAP-1.D.4, AAP-1.D.6, AAP-1.D.7, AAP-1.D.8:** *Developing a data abstraction to implement in a program can result in a program that is easier to develop and maintain. The use of lists allows multiple related items to be treated as a single value. The exam reference sheet provides notation to create and manage lists, which are 1-indexed.*

- **Elaboration and Explanation:**

- **Easier to Maintain:** If you add a new friend to your contacts, you just add one item to your list. You don't have to rewrite your program to add a new variable.

- **List Notation (from reference sheet):**

- `myList ? [val1, val2, ...]` creates a new list.

- `myList ? []` creates an empty list.

- `aList ? bList` copies list `bList` to `aList`.

- **1-Indexed:** This is crucial. The first item in a list is at index `1`, the second at index `2`, and so on. This is different from many real-world programming languages which are 0-indexed.

- **Everyday Analogy (1-Indexing):** Think of a podium for a race. The person in 1st place is the first person, not the "0th" person. The AP CSP reference sheet thinks like this.

- **Code Example:**

```
// Create a list of players
players ? ["Alice", "Bob", "Charlie"]

// Access the second player
// In AP CSP, this is index 2.
currentPlayer ? players[2]  // currentPlayer is now "Bob"

// Add a new player. The program logic doesn't need to change.
APPEND(players, "David") // players is now ["Alice", "Bob", "Charlie", "David"]
```

- **Common Misconception:** "The first item is always at index 0."

- **Correction:** This is true in many languages like Java, Python, and JavaScript, but **NOT** on the AP CSP Exam. On the exam, lists are always 1-indexed. `myList[1]` is the first element. Accessing `myList[0]` would cause an error.

3. Mastering the Learning Objectives

- **Learning Objective (LO) AAP-1.D:** *For data abstraction: a. Develop data abstraction using lists to store multiple elements. b. Explain how the use of data abstraction manages complexity in program code.*
- **Actionable Guidance (How to do this):**

1. **Identify Related Data:** When reading a problem description, look for multiple pieces of data that are the same "kind" of thing (e.g., a list of student names, a collection of sensor readings, a set of menu items).
2. **Choose a Good Name:** Instead of `item1`, `item2`, `item3`, group them. Create a single list with a descriptive name, like `studentRoster`, `sensorReadings`, or `menuItems`. This is you *developing the data abstraction*.
3. **Explain the "Why":** To explain how it manages complexity, use this framework: "By using the list `[your list name]`, I was able to manage complexity. Instead of creating and managing multiple variables (like `var1`, `var2`, `var3`...), I could store all related data in a single structure. This is more efficient because `[choose one or more reasons below]`:"
   - "...I can easily add or remove items without rewriting my code."
   - "...I can use a loop to process every item in the collection with just a few lines of code."
   - "...the code is more readable and easier to understand."

- **Illustrative Scenario:**
- **Problem:** You are creating a simple quiz program. The program needs to store 5 questions. After the user answers all 5, it tells them how many they got right.
- **Applying the LO:**
- **Developing Abstraction:** Instead of `question1 ? "..."`, `question2 ? "..."`, etc., you identify that these are all the same *kind* of data. You create a data abstraction using a list:

```
quizQuestions ? ["What is 2+2?", "What is the capital of France?", "True or false: The sky is blue.", "What is 10/2?", "What is the first letter of the alphabet?"]
```

You would do the same for the answers:

```
quizAnswers ? ["4", "Paris", "True", "5", "A"]
```

- **Explaining Complexity Management:** How does this manage complexity? If we wanted to make it a 10-question quiz, we don't need to add 5 new variables and 5 new `IF` statements. We just add 5 more items to the `quizQuestions` and `quizAnswers` lists. Our loop that runs the quiz will handle it automatically. It makes the program scalable and easier to maintain.

4. Practice Makes Perfect

- **Multiple-Choice Question 1:**

A programmer is creating a program to store the top 10 high scores for a game. Which of the following is the primary benefit of using a single list named `highScores` to store the scores instead of ten separate variables (`score1`, `score2`, etc.)?
(A) Using a list requires less computer memory than using ten separate variables.
(B) Using a list allows the scores to be stored as text, while variables can only store numbers.

(C) The list provides a data abstraction, making it easier to write code that iterates through all the scores to find the lowest or highest value.
(D) The list is automatically sorted from highest to lowest when it is created.

- **Multiple-Choice Question 2:**

Consider the following code segment, which uses a 1-indexed list `fruits`.

```
fruits ? ["apple", "banana", "cherry"]
INSERT(fruits, 1, "apricot")
REMOVE(fruits, 3)
DISPLAY(fruits[2])
```

What is displayed as a result of executing the code segment?
(A) apple
(B) banana
(C) cherry
(D) apricot

- **Short-Answer Question 1:**

In your own words, define "data abstraction" and provide one reason why a programmer would choose to use it.

- **Short-Answer Question 2:**

You are writing a program to help a teacher manage their classroom. The program needs to keep track of which students have turned in their homework. Describe how you could use a data abstraction (specifically, a list) to store this information. What would you name your list?

- **AP-Style Code Analysis Challenge:**

A programmer wrote the following code to calculate the average of three exam scores and display it.

Original Code:

```
score1 ? 85
score2 ? 92
score3 ? 78
sum ? score1 + score2 + score3
average ? sum / 3
DISPLAY("The average is: ")
DISPLAY(average)
```

The programmer now needs to modify the program to handle 25 exam scores.

Your Task:

1. Rewrite the program using a list as a data abstraction to store the 25 scores. You don't need to type out all 25 scores; you can represent them with placeholders (e.g., `s1, s2, ...`).
2. Your new code should use iteration to calculate the sum.
3. Explain how using a list in this scenario manages the complexity of the program, especially if the number of scores changes again in the future.

Solution Walkthrough for Code Analysis Challenge:

1. **Rewritten Program Code:**

```
// Part 1: Storing scores in a list (data abstraction)
examScores ? [85, 92, 78, ..., s25] // List of 25 scores
```

```
    // Part 2: Using iteration to calculate sum
    sum ? 0
    FOR EACH score IN examScores
    {
        sum ? sum + score
    }

    // Calculating the average using the list's length
    average ? sum / LENGTH(examScores)

    DISPLAY("The average is: ")
    DISPLAY(average)
```

2. **Explanation of How Complexity is Managed:**
Using the list `examScores` is a form of data abstraction that significantly manages the program's complexity.

- **Scalability and Maintenance:** In the original code, if we changed from 3 scores to 25, we would need to add 22 new variables. The line calculating the `sum` would become enormous and hard to read (`sum ? score1 + score2 + ... + score25`). With the list, to change the number of scores, we only need to change the items inside the list. The loop (`FOR EACH`) and the average calculation (`sum / LENGTH(examScores)`) work correctly no matter if there are 3 scores or 300 scores. This makes the program much easier to update and maintain.

- **Readability and Simplicity:** The logic of the `FOR EACH` loop is very clear: "for every score in the list, add it to the sum." This is much simpler to read and understand than a single line with 25 variables being added together. The abstraction hides the messy details of the individual scores and lets us focus on the process of calculating the average.

Topic 2: Strings (CED Topic 3.4)
1. Topic Overview

Think of a **String** as a sequence of letters, numbers, and symbols strung together, just like beads on a string. Every text message you send, every username you create, and every sentence you read on a webpage is a string in the world of computer science.
Strings are a fundamental way that computers handle text. They aren't just jumbles of characters; they are *ordered* sequences. This order is important because it allows us to do useful things, like join two strings together to form a new one (e.g., "first name" + "last name") or pull out a piece of a string (e.g., getting the area code from a phone number). Understanding strings is key to creating interactive and user-friendly programs.
2. Deconstructing the Essential Knowledge

- **Essential Knowledge (EK) AAP-1.C.4:** *A string is an ordered sequence of characters.*

- **Elaboration and Explanation:** The key word here is **ordered**. This means the position of each character matters. "cat" is a different string from "act" because the characters are in a different order.

- **Everyday Analogy:** A phone number, `555-1234`, is an ordered sequence of characters. If you scramble the order to `432-1555`, it's a completely different (and useless) phone number.

- **Code Example:**

```
    username ? "KITTEN123"
```

In this string, "K" is the first character, "I" is the second, and so on. In AP CSP (which uses 1-based indexing for strings too), `username[1]` would refer to "K" and `username[5]` would refer to "E".

- **Common Misconception:** "Strings containing numbers are the same as numbers."
- **Correction:** The string `"123"` is not the same as the number `123`. The string is a sequence of three characters: '1', '2', and '3'. The number is a mathematical value. You can do math on the number (`123 + 1` is `124`), but you can't do math on the string. As we'll see next, `"123" + "1"` is `"1231"`.
- **Essential Knowledge (EK) AAP-2.D.1:** *String concatenation joins together two or more strings end-to-end to make a new string.*
- **Elaboration and Explanation: Concatenation** is the fancy word for "gluing strings together." In the AP CSP reference sheet, this is done with the + operator when at least one of the operands is a string.
- **Everyday Analogy:** You have two separate words written on index cards: "ice" and "cream". Concatenation is like taping the cards together to make "icecream".
- **Code Example:**

```
firstName ? "Grace"
lastName ? "Hopper"
fullName ? firstName + " " + lastName // The " " is also a string
// fullName is now "Grace Hopper"
```

- **Common Misconception:** "The + operator always means addition."
- **Correction:** The + operator is context-sensitive. If you use it with two numbers, it performs addition (`5 + 2` is `7`). If you use it with two strings, or a string and a number, it performs concatenation (`"5" + "2"` is `"52"`, and `"hello" + 5` is `"hello5"`).
- **Essential Knowledge (EK) AAP-2.D.2:** *A substring is part of an existing string.*
- **Elaboration and Explanation:** A **substring** is simply a smaller string that is found inside a larger one. For example, "comp" is a substring of "computer".
- **Everyday Analogy:** The word "science" is a substring of the sentence "AP Computer Science is fun."
- **Code Example:** The AP CSP Exam Reference Sheet does not provide a specific function for extracting substrings, but you must understand the concept. If a language had a function like `SUBSTRING(string, start, end)`, you could do this:

```
fullDate ? "2024-10-26"
// A hypothetical function call
year ? SUBSTRING(fullDate, 1, 4) // year would become "2024"
```

- **Common Misconception:** "All parts of a string are substrings."
- **Correction:** This is technically true, but for the concept to be useful, a substring must be a *contiguous* block of characters. In the string "computer", "cop" is not a substring because the 'o' and 'p' are not next to each other in the original string.

3. Mastering the Learning Objectives

- **Learning Objective (LO) AAP-2.D:** *Evaluate expressions that manipulate strings.*
- **Actionable Guidance (How to do this):**

1. **Identify Strings vs. Variables:** Look for quotation marks. Anything inside `" "` is a literal string. Anything without quotes is a variable.

2. **Check the Operator:** If you see a + sign, look at what's on either side of it.
   - `number + number` = Addition.
   - `string + string` = Concatenation.
   - `string + number` = Concatenation (the number is converted to a string).
   - `number + string` = Concatenation.

3. **Go Step-by-Step:** Evaluate expressions in the correct order, just like in math. If you have `varA + varB + varC`, first evaluate `varA + varB`, and then concatenate the result with `varC`.
   - **Illustrative Scenario:**
   - **Problem:** A program needs to generate a user ID from a name and a year of birth. The user ID format should be `firstName_lastName_birthYear`.
   - **Applying the LO:** Let's trace the creation of the ID.

1. Initialize variables:

```
firstName ? "Alan"
lastName ? "Turing"
birthYear ? 1912
```

2. Evaluate the expression to create the `userID`:

```
userID ? firstName + "_" + lastName + "_" + birthYear
```

3. **Step-by-step evaluation:**
   - `firstName + "_"` becomes `"Alan" + "_"` which results in `"Alan_"`.
   - `"Alan_" + lastName` becomes `"Alan_" + "Turing"` which results in `"Alan_Turing"`.
   - `"Alan_Turing" + "_"` becomes `"Alan_Turing_"`.
   - `"Alan_Turing_" + birthYear` becomes `"Alan_Turing_" + 1912`. Since one part is a string, this is concatenation. The number `1912` is treated like the string `"1912"`. The final result is `"Alan_Turing_1912"`.

4. The final value of `userID` is the string `"Alan_Turing_1912"`.

****4. Practice Makes Perfect****

- **Multiple-Choice Question 1:**

Consider the following code segment:

```
a ? 10
b ? "20"
c ? a + b
DISPLAY(c)
```

What is displayed as a result of executing the code segment?
(A) 10
(B) 20
(C) 30
(D) 1020

- **Multiple-Choice Question 2:**

A program is intended to check if a user's entered password, stored in the variable `pass`, is exactly "GoKnights1". Which of the following conditions correctly checks this?
(A) `pass = GoKnights1`
(B) `pass = "GoKnights1"`

(C) `pass + 1 = "GoKnights2"`
(D) `pass = "Go" + "Knights" + "1"`

- **Short-Answer Question 1:**

What is string concatenation? Provide a simple pseudocode example that concatenates a variable and a string literal.

- **Short-Answer Question 2:**

A program has two variables: `city ? "New York"` and `state ? "NY"`. Write a single line of pseudocode that will create a new variable `location` that stores the string `"New York, NY"`.

- **AP-Style Code Analysis Challenge:**

Analyze the following code segment and determine the final value displayed.

Code:

```
part1 ? "pass"
part2 ? "word"
num ? 1

secret ? part1 + part2
secret ? secret + num
num ? num + 1
secret ? secret + num

DISPLAY(secret)
```

Provide a step-by-step trace of your evaluation to show how you arrived at your answer.

Solution Walkthrough for Code Analysis Challenge:

Let's trace the values of the variables line by line.

1. `part1 ? "pass"`
   - `part1` is `"pass"`
2. `part2 ? "word"`
   - `part2` is `"word"`
3. `num ? 1`
   - `num` is the number `1`
4. `secret ? part1 + part2`
   - This is string concatenation: `"pass" + "word"`
   - `secret` becomes `"password"`
5. `secret ? secret + num`
   - This is concatenation between a string and a number: `"password" + 1`
   - `secret` becomes `"password1"`
6. `num ? num + 1`
   - This is numeric addition: `1 + 1`
   - `num` is updated to `2`
7. `secret ? secret + num`

- This is concatenation between a string and a number: `"password1" + 2`
- `secret` becomes `"password12"`

8. `DISPLAY(secret)`
   - The final value displayed is **"password12"**.

Topic 3: Nested Conditionals (CED Topic 3.7)

1. Topic Overview

Imagine you're at a vending machine. Your first decision (**IF**) is, "Do I have enough money?" If the answer is no, you walk away. But if the answer is yes, you then face a second decision (**IF**): "Which snack do I want?" This two-level decision-making process is exactly what a **Nested Conditional** is.
It's a conditional statement (an `IF` statement) placed inside another `IF` or `ELSE` block. Nested conditionals allow programs to handle more complex scenarios where one condition must be true *before* a second, more specific condition can even be checked. They are the key to creating precise and nuanced logic in your algorithms.

2. Deconstructing the Essential Knowledge

- **Essential Knowledge (EK) AAP-2.I.1:** *Nested conditional statements consist of conditional statements within conditional statements.*
- **Elaboration and Explanation:** This simply means you can put an `IF...ELSE` structure inside the code block of another `IF...ELSE` structure. This creates a hierarchy of decisions. The outer `IF` is checked first, and only if its condition is met do you proceed to check the inner `IF`.
- **Everyday Analogy:** Planning a weekend trip. **Outer IF:** `IF (the weather is good)`. If it's not, you stay home. **Inner IF:** `IF (the weather is good)` is true, you then check `IF (we have enough gas)`. You only worry about gas *if* the weather is good enough to go in the first place.
- **Code Example:**

```
score ? 85
passed ? false

IF (score >= 60)
{
    // --- This is the outer block ---
    passed ? true
    DISPLAY("You passed the course.")

    // --- This is the NESTED conditional ---
    IF (score >= 90)
    {
        DISPLAY("Excellent! You earned an A.")
    }
    // --- End of nested conditional ---
}
ELSE
{
    DISPLAY("You did not pass the course.")
}
```

The check for an "A" is *nested* inside the check for passing. You can't get an A if you didn't pass.

- **Common Misconception:** "You have to check all the `IF` conditions in the code."
- **Correction:** You only execute the code in the blocks you enter. In the example above, if `score` was `50`, the outer `IF (score >= 60)` would be false. The program would jump directly to the `ELSE` block. The inner `IF (score >= 90)` would be completely ignored and never evaluated.

3. Mastering the Learning Objectives

- **Learning Objective (LO) AAP-2.I:** *For nested selection: a. Write nested conditional statements. b. Determine the result of nested conditional statements.*
- **Actionable Guidance (How to trace nested conditionals):**

1. **Start at the Top:** Always begin with the outermost `IF` statement.
2. **Evaluate the Outer Condition:** Determine if the outer condition is `true` or `false`.
3. **Choose a Path:**
   - If `true`, enter the `IF` block. Ignore the `ELSE` block entirely. Now, proceed to evaluate any inner conditionals within that `IF` block.
   - If `false`, skip the entire `IF` block and go directly to the `ELSE` block (or to the end of the statement if there is no `ELSE`). Now, proceed to evaluate any inner conditionals within that `ELSE` block.
4. **Repeat for Inner Conditionals:** Follow the same logic for any conditional statements you find inside the chosen path.
5. **Track the Output:** Pay attention to what `DISPLAY` statements are executed along your chosen path.

- **Illustrative Scenario:**
- **Problem:** An app determines the ticket price for an amusement park. Regular price is $50. Kids under 12 get a 50% discount. Seniors (65 and over) get a 20% discount. These discounts do not stack.
- **Applying the LO (Writing the code):** We need a nested structure because the discount logic depends on age.

```
            age ? 10
            price ? 50.00

            // Outer IF checks for any discount eligibility
            IF (age < 12 OR age >= 65)
            {
                // Inner IF determines WHICH discount to apply
                IF (age < 12)
                {
                    price ? price * 0.50 // 50% discount for kids
                }
                ELSE
                {
                    price ? price * 0.80 // 20% discount for seniors
                }
            }
            // If the outer IF is false, price remains $50 (no ELSE needed)

            DISPLAY("Your ticket price is: $")
            DISPLAY(price)
```

- **Applying the LO (Determining the result):** Let's trace with `age ? 10`.

1. **Outer Condition:** `(10 < 12 OR 10 >= 65)` -> `(true OR false)` -> `true`. We enter the outer `IF` block.
2. **Inner Condition:** `(10 < 12)` -> `true`. We enter the inner `IF` block.
3. **Execution:** `price ? 50.00 * 0.50` -> `price` becomes `25.00`. The inner `ELSE` is skipped.
4. **Output:** The program will display "Your ticket price is: $25.00".

****4. Practice Makes Perfect****

- **Multiple-Choice Question 1:**

Consider the following code segment:

```
x ? 10
y ? 5
IF (x > y)
{
    IF (x > 15)
    {
        DISPLAY("A")
    }
    ELSE
    {
        DISPLAY("B")
    }
}
ELSE
{
    DISPLAY("C")
}
```

What is displayed as a result of executing the code segment?
(A) A
(B) B
(C) C
(D) B and C

- **Multiple-Choice Question 2:**

Consider the following code segment:

```
temp ? 75
raining ? false
IF (temp >= 70)
{
    DISPLAY("It's warm. ")
    IF (NOT raining)
    {
        DISPLAY("Let's go to the park.")
    }
}
ELSE
{
    DISPLAY("It's cold. ")
}
```

Personalized Study Guide

Which of the following changes would cause the program to display "It's cold. "?
(A) `temp ? 80`
(B) `temp ? 65`
(C) `raining ? true`
(D) `temp ? 75` and `raining ? true`

- **Short-Answer Question 1:**

In your own words, what is a nested conditional statement? Provide a real-world example (not from this guide) where you might use a nested decision.

- **Short-Answer Question 2:**

A website is offering a discount. Users get 10% off if their order total is over $50. If the user is also a premium member, they get an additional 5% off (on the original price). Write a pseudocode segment using nested conditionals that calculates the final price based on variables `orderTotal` and `isPremiumMember` (a boolean).

- **AP-Style Code Analysis Challenge:**

Analyze the following code segment. Determine the output for each of the three initial value sets provided below.

Code:

```
a ? 0
b ? 0
c ? 0

// Assume inputA, inputB, and inputC are provided
// before this code runs.

IF (inputA > 10)
{
    a ? 5
    IF (inputB > 10)
    {
        b ? 5
    }
    ELSE
    {
        b ? 2
    }
}
ELSE
{
    c ? 5
    IF (inputC > 10)
    {
        a ? 2
    }
}
DISPLAY(a)
DISPLAY(b)
DISPLAY(c)
```

Personalized Study Guide

Determine the final displayed output for each case:

1. `inputA = 12`, `inputB = 5`, `inputC = 15`
2. `inputA = 5`, `inputB = 12`, `inputC = 15`
3. `inputA = 5`, `inputB = 5`, `inputC = 5`

Solution Walkthrough for Code Analysis Challenge:

Case 1: `inputA = 12`, `inputB = 5`, `inputC = 15`

1. Initial values: `a=0, b=0, c=0`.
2. **Outer `IF`:** `(inputA > 10)` -> `(12 > 10)` is **true**. We enter the first main block.
3. `a ? 5`. a is now `5`.
4. **Inner `IF`:** `(inputB > 10)` -> `(5 > 10)` is **false**. We enter the inner `ELSE` block.
5. `b ? 2`. b is now `2`.
6. The outer `ELSE` block is skipped entirely.
7. Final values: `a=5, b=2, c=0`.
8. **Output:** 5 2 0

Case 2: `inputA = 5`, `inputB = 12`, `inputC = 15`

1. Initial values: `a=0, b=0, c=0`.
2. **Outer `IF`:** `(inputA > 10)` -> `(5 > 10)` is **false**. We skip the first main block and enter the outer `ELSE` block.
3. `c ? 5`. c is now `5`.
4. **Inner `IF`:** `(inputC > 10)` -> `(15 > 10)` is **true**. We enter the inner `IF` block.
5. `a ? 2`. a is now `2`.
6. Final values: `a=2, b=0, c=5`.
7. **Output:** 2 0 5

Case 3: `inputA = 5`, `inputB = 5`, `inputC = 5`

1. Initial values: `a=0, b=0, c=0`.
2. **Outer `IF`:** `(inputA > 10)` -> `(5 > 10)` is **false**. We enter the outer `ELSE` block.
3. `c ? 5`. c is now `5`.
4. **Inner `IF`:** `(inputC > 10)` -> `(5 > 10)` is **false**. We skip the inner `IF` block. There is no `ELSE`, so nothing more happens inside this block.
5. Final values: `a=0, b=0, c=5`.
6. **Output:** 0 0 5