Personalized Study Guide

Here is a concise, easy-to-understand study guide for your AP Computer Science Principles exam, based only on the provided course material:

Identifying and Correcting Errors

- What you need to know:
- Programs can have different types of errors:
- Logic error*: The program runs, but it behaves incorrectly or unexpectedly. The mistake is in the algorithm itself.
- Syntax error*: The rules of the programming language aren't followed (like a grammar mistake in English).
- Run-time error*: The program stops or crashes while it's running. These are specific to the programming langua
- Overflow error*: The computer tries to handle a number that is too large for its defined range.
- To find and fix errors, you can use:
- Test cases*: Specific inputs used to check if the program gives the expected outcomes.
- Hand tracing*: Manually going through the code step-by-step to see what it does.
- Visualizations*: Using tools that show how the program executes.
- Debuggers*: Tools that help programmers find errors.
- Adding extra output statements*: Temporarily printing values to see what's happening inside the program.
- When testing, use inputs that are at the minimum and maximum extremes of what the program should handle, b
- What you should be able to do:
- Identify different types of errors in an algorithm or program.
- Correct these errors so the algorithm or program works as intended.
- Choose appropriate inputs and determine their expected outputs/behaviors to test if an algorithm or program is c

Conditionals

- What you need to know:
- Selection* is a key part of algorithms that decides which parts of the code run based on whether a condition is tr
- Conditional statements* (like "if-statements") control the flow of a program by executing different code blocks ba
- Basic conditional structures include:
- `IF (condition) { <block of statements> }`: The statements inside the block run only if the condition is true.
- `IF (condition) { <first block> } ELSE { <second block> }`: The first block runs if the condition is true, otherwise th
- Nested conditional statements* mean having conditional statements inside other conditional statements.
- What you should be able to do:
- Express algorithms that involve making decisions (selection) without using a specific programming language (e.g
- Write conditional statements in code.
- Determine what a conditional statement will do, or what its result will be, including with nested conditionals.

Lists

- What you need to know:
- A *list* is an ordered sequence of elements, used to store multiple related items under a single variable. (It migh
- Each item in a list has a unique *index* (position), typically starting from 1 for the first element.
- Basic list operations include:
- Accessing an element by its index (e.g., `aList[i]`).
- Assigning a value to a variable from a list element.
- Assigning a value to a specific element in a list.
- Inserting elements at a given index (shifting others to the right).
- Adding elements to the end of the list (appending).
- Removing elements at a given index (shifting others to the left).
- Determining the *length* (number of elements) of a list.
- Traversing a list* means accessing some or all of its elements, often using *iteration* statements like `FOR EAC

Personalized Study Guide

- Common algorithms that use lists include finding the minimum/maximum value or calculating a sum/average.
- Linear search* (or sequential search) checks each element in order until the desired value is found or the end of
- What you should be able to do:
- Write and evaluate code that uses list indexing and list procedures (operations).
- Write iteration statements to go through a list (traverse it).
- Determine the result of algorithms that involve moving through lists.

Developing Procedures

- What you need to know:
- A *procedure* (also called a method or function) is a named group of programming instructions that can have inp
- Procedural abstraction* is a type of abstraction that gives a name to a process. It allows you to use the procedur
- Procedural abstraction helps manage complexity by:
- Breaking down a large problem into smaller, solvable subproblems (modularity).
- Allowing code to be reused, avoiding duplication.
- Making the code more readable.
- Allowing the internal workings of a procedure to be changed (e.g., to make it faster) without affecting other parts
- Parameters* make procedures more general, allowing them to be reused with different input values (arguments)
- When a procedure is called, the program executes the statements within the procedure before returning to wher
- What you should be able to do:
- Explain how using procedural abstraction helps manage complexity in a program.
- Develop procedural abstractions by writing your own procedures with parameters and clear functionalities.

Simulations

- What you need to know:
- Simulations* are simplified models (*abstractions*) of more complex real-world objects or phenomena. They use
- The purpose of simulations is often to draw conclusions or investigate phenomena without the limitations of the r
- Creating a simulation involves deciding which details to remove or simplify.
- Simulations can contain *bias* based on the real-world elements that were included or left out.
- Simulations help in creating and refining hypotheses.
- Random number generators* can be used in simulations to mimic real-world variability.
- What you should be able to do:
- Explain how computers can represent real-world events or outcomes through simulations.
- Compare simulations with real-world situations, understanding their benefits and limitations.

Undecidable Problems

- What you need to know:
- A *decidable problem* is a decision problem (one with a yes/no answer) for which an algorithm can always be w
- An *undecidable problem* is a problem for which no algorithm can *ever* be constructed that will always provide
- Even if a problem is undecidable, it might still have *some* specific instances that can be solved algorithmically,
- What you should be able to do:
- Explain that there are problems in computer science that cannot be solved by any algorithm, known as undecida