

Unit 3: Algorithms and Programming

Unit 3 teaches you to think the way a programmer thinks. It starts with variables and the types of data they hold, then moves to the expressions that compute with that data, the conditionals that choose between paths, and the loops that repeat work. Lists let you manage collections of values, and procedures package logic into named, reusable pieces. The unit closes with testing, debugging, a first look at why some algorithms take longer than others, and the Create Performance Task, which is built from exactly these ideas.

[AP Computer Science Principles](#)[Algorithms and Programming](#)[About 17 minutes to read](#)

How to use this guide

Read it in order the first time. The topics stack: variables feed expressions, expressions feed conditionals and loops, loops and conditionals drive list algorithms, and procedures wrap all of it into manageable pieces. Exam questions usually hand you a short program and ask what it displays or what is wrong with it, so trace every example by hand instead of just reading the trace tables. The pseudocode in this guide follows the AP exam reference sheet exactly, including the left arrow for assignment and list indices that start at 1.

After the first read, review with the trap boxes and the confusion table, work the practice questions, and finish with the recall check on the last page. Say the recall answers out loud and note any items you cannot explain yet.

What this unit is worth. Algorithms and Programming is about 30 to 35 percent of the AP Computer Science Principles exam, the largest share of any unit. It also feeds the Create Performance Task, which is 30 percent of your AP score and asks you to write, demonstrate, and explain a program built from these ideas. Mastering this unit therefore prepares you for both parts of the exam.

Variables and Assignment

A **variable** is a named storage location for a value. The name stays the same while the value it holds can change as the program runs. **Assignment** stores a value in a variable and is written with a left arrow: `score ← 10` puts the value 10 into the variable `score`.

Assignment is not equality. The statement `score ← 10` changes `score`. The expression `score = 10` asks whether `score` currently holds 10 and produces true or false. Mixing them up is one of the most common reading errors on the exam, so read the arrow as gets and the equals sign as is equal to.

Variables get their values from constants, from `INPUT()`, from arithmetic, or from other variables. Each assignment replaces the old value completely. Trace this short program:

```
x ← 3
x ← x + 2
DISPLAY(x)
```

The first line stores 3 in `x`. The second line reads the current value of `x`, adds 2, and stores the result back, so `x` becomes 5. The display shows 5.

Trap. In `x ← x + 2`, the `x` on the right is read before the assignment happens. The old value 3 is used in the calculation, then the result 5 is stored back into `x`. A variable can appear on both sides of the arrow in the same statement.

Data Types

Every value has a **type**, and the type decides which operations make sense. The exam works with five types.

TYPE	HOLDS	EXAMPLE
Integer	Whole numbers, positive, negative, or zero	7, -3, 0
Real (float)	Numbers with fractional parts	3.5, -0.25
Boolean	Exactly one of the two values true or false	The result of <code>4 > 1</code>
String	Text written in quotes	"hello"
List	An ordered collection of values under one name	[2, 4, 6]

Types matter because the same symbol can do different work depending on the type. The `+` in `3 + 4` adds numbers and gives 7, while the `+` in `"co" + "de"` joins strings and gives "code". Most exam errors about types come from feeding an operation a type it was not built for.

Trap. A Boolean is not a number in the exam pseudocode. The expression `true + 1` is a type error, not 2, and `DISPLAY(true AND 3)` is a type error, not false. Check that each operator is getting the type it expects.

Arithmetic Expressions and Precedence

An **arithmetic expression** combines values with operators: + for addition, - for subtraction, * for multiplication, / for division, and MOD for the remainder after division. The expression 17 MOD 5 evaluates to 2, because 17 divided by 5 leaves a remainder of 2.

Operators have an order, called **precedence**. Parentheses come first. Then *, /, and MOD are evaluated from left to right. Then + and - are evaluated from left to right. Trace this expression:

```
DISPLAY(2 + 3 * 4 MOD 5)
```

Multiplication and MOD come before addition, so $3 * 4 = 12$ first, then $12 \text{ MOD } 5 = 2$, then $2 + 2 = 4$. The display shows 4.

Trap. MOD is not division. The expression $12 / 5$ gives the quotient 2.4, while $12 \text{ MOD } 5$ gives the remainder 2. Questions about divisibility, even and odd numbers, or cycling through a range are MOD questions in disguise.

RANDOM(a, b) returns an integer from a to b, including both endpoints. **RANDOM(1, 6)** can produce 1, 2, 3, 4, 5, or 6, which is why it appears in dice and lottery examples. A common mistake is assuming the upper bound is excluded. It is not.

Boolean Expressions

A **Boolean expression** evaluates to true or false. Comparisons use =, ≠, >, <, ≥, and ≤. The Boolean operators are NOT, AND, and OR. NOT flips a value, AND is true only when both sides are true, and OR is true when at least one side is true.

Precedence continues here. Comparisons are evaluated before Boolean operators, and among the operators NOT comes first, then AND, then OR. Parentheses override everything. Trace this expression:

```
DISPLAY(NOT (3 > 5) AND (2 + 2 = 4))
```

The comparisons run first: $3 > 5$ is false and $2 + 2 = 4$ is true. Then NOT false gives true, and true AND true gives true. The display shows true.

EXPRESSION	VALUE	WHY
true AND false	false	AND needs both sides true
true OR false	true	OR needs at least one side true
NOT true OR false	false	NOT runs first: false OR false
NOT (true OR false)	false	Parentheses run first: NOT true

Trap. The = inside a condition compares; it never stores. IF ($x = 5$) tests x, while $x \leftarrow 5$ changes x. If a program behaves as if a condition is always true, check whether an assignment slipped into the test.

Strings

A **string** is a sequence of characters written in quotes. Strings can be joined with +, which is called **concatenation**: "area" + "51" gives "area51". Strings can also be compared with = and ≠, which is how programs check names, passwords, and answers.

On the exam reference sheet, LENGTH is defined for lists. Treat strings as values you concatenate and compare, and do not assume list operations apply to them.

Conditionals

A **conditional** runs different code depending on whether a condition is true. The basic form tests one condition, with an optional ELSE covering everything else. An ELSE IF chain tests several conditions in order, and only the first true branch runs. Trace this program:

```
score ← 85
IF (score ≥ 90)
{
    DISPLAY("A")
}
ELSE IF (score ≥ 80)
{
    DISPLAY("B")
}
ELSE
{
    DISPLAY("C")
}
```

The test $85 \geq 90$ is false, so the first branch is skipped. The test $85 \geq 80$ is true, so B is displayed and the ELSE is skipped. The order of the tests matters: if the ≥ 80 test came first, a score of 95 would display B, which is wrong.

Nested Conditionals

A conditional placed inside another conditional is called **nested**. The inner test only runs when the outer test is true, so nesting expresses a condition that depends on another condition. Trace this program:

```
temp ← 75
IF (temp > 32)
{
  IF (temp > 90)
  {
    DISPLAY("hot")
  }
  ELSE
  {
    DISPLAY("mild")
  }
}
ELSE
{
  DISPLAY("freezing")
}
```

The outer test $75 > 32$ is true, so the inner IF runs. The inner test $75 > 90$ is false, so the inner ELSE runs and mild is displayed. The outer ELSE is skipped entirely because the outer condition was true.

Trap. An ELSE always pairs with the nearest unmatched IF above it. When tracing nested conditionals, match each ELSE to its IF before deciding which branch runs.

Iteration: Three Loops

Iteration repeats a block of code. The exam gives you three loops, and choosing the right one is a tested skill.

LOOP	FORM	WHEN TO USE IT
REPEAT n TIMES	REPEAT 4 TIMES { }	The number of repetitions is known before the loop starts.
REPEAT UNTIL	REPEAT UNTIL (done) { }	The loop must continue until some condition becomes true. The count is unknown, and the body always runs at least once.
FOR EACH	FOR EACH item IN list { }	Every element of a list needs the same treatment, exactly once each.

REPEAT UNTIL checks its condition after running the body, so the body executes at least once even if the condition is already true. FOR EACH handles the indexing for you: the loop variable takes each value in turn, and you never touch an index. Trace this loop:

```
n ← 6
REPEAT UNTIL (n < 4)
{
  n ← n - 2
}
DISPLAY(n)
```

CHECK N < 4	BODY RUNS	N AFTER
6 < 4 is false	n ← 4	4
4 < 4 is false	n ← 2	2
2 < 4 is true	Loop ends	2

The display shows 2.

Trap. REPEAT UNTIL stops when its condition becomes true, which is the reverse of a loop that continues while a condition holds. To keep looping while it is raining, write REPEAT UNTIL (NOT raining).

When you need each element's position, pair REPEAT TIMES with an index variable:

```
names ← ["Ana", "Ben", "Cyd"]
index ← 1
REPEAT LENGTH(names) TIMES
{
  DISPLAY(names[index])
  index ← index + 1
}
```

LENGTH(names) is 3, so the body runs three times with index = 1, 2, 3, displaying Ana, Ben, and Cyd. Starting the index at 1 matches the exam rule that lists are indexed from 1.

Lists

A **list** is an ordered collection of values stored under one name. Lists are the exam's tool for managing more data than you could reasonably name one variable at a time.

Indexing starts at 1. In `scores ← [72, 95, 61]`, `scores[1]` is 72, `scores[2]` is 95, and `scores[3]` is 61. There is no `scores[0]`, and an index past the end is also an error. This is the single most tested fact about lists.

OPERATION	WHAT IT DOES
<code>APPEND(list, value)</code>	Adds value to the end. <code>APPEND(scores, 88)</code> makes <code>[72, 95, 61, 88]</code> .
<code>INSERT(list, index, value)</code>	Inserts at the index, shifting later elements right. <code>INSERT(scores, 2, 80)</code> makes <code>[72, 80, 95, 61]</code> .
<code>REMOVE(list, index)</code>	Removes the element at the index, shifting later elements left. <code>REMOVE(scores, 1)</code> makes <code>[95, 61]</code> .
<code>LENGTH(list)</code>	Returns the number of elements. <code>LENGTH(scores)</code> is 3.

Trap. `INSERT` does not replace. `INSERT(scores, 2, 80)` keeps the 95 and moves it to index 3. To overwrite an element, assign to the index directly: `scores[2] ← 80` replaces the 95.

Off-by-one errors are the classic list bug. A loop meant to visit every element must run from index 1 through `LENGTH(list)`. Trace this traversal:

```
data ← [10, 20, 30]
index ← 1
REPEAT UNTIL (index > LENGTH(data))
{
    DISPLAY(data[index])
    index ← index + 1
}
```

`LENGTH(data)` is 3. With `index = 1, 2, and 3` the loop displays 10, 20, and 30, and `index` becomes 4. The check `4 > 3` is true, so the loop ends. If the condition had been `index ≥ LENGTH(data)`, the loop would have stopped with `index = 3` and the 30 would never display.

Common List Algorithms

Most list questions on the exam are variations on a few patterns: visit every element, keep a running value, and update it when an element qualifies. Searching a list from first to last while checking each element is called **linear search**, and each pattern below is a linear search with different bookkeeping. Learn the patterns and you can trace any of them.

Pattern 1: Sum of a list

Initialize a total to 0, add each element, and display the total.

```
numbers ← [4, 9, 2, 7]
total ← 0
FOR EACH num IN numbers
{
    total ← total + num
}
DISPLAY(total)
```

NUM	TOTAL BEFORE	TOTAL AFTER
4	0	4
9	4	13
2	13	15
7	15	22

The display shows 22. The total must start at 0. Starting it at the first element would add that element twice.

Pattern 2: Find the maximum

Initialize the maximum to the first element, then replace it whenever a larger element appears.

```
vals ← [3, 11, 5, 9]
maxVal ← vals[1]
FOR EACH v IN vals
{
    IF (v > maxVal)
    {
        maxVal ← v
    }
}
DISPLAY(maxVal)
```

V	V > MAXVAL	MAXVAL AFTER
3	3 > 3 is false	3
11	11 > 3 is true	11
5	5 > 11 is false	11
9	9 > 11 is false	11

The display shows 11. Initializing maxVal to vals[1] instead of 0 matters: if every value were negative, starting at 0 would wrongly report 0 as the maximum.

Pattern 3: Count values meeting a condition

Initialize a counter to 0 and add 1 for each element that passes the test.

```
scores ← [72, 95, 61, 88, 95]
count ← 0
FOR EACH s IN scores
{
  IF (s ≥ 90)
  {
    count ← count + 1
  }
}
DISPLAY(count)
```

s	s ≥ 90	COUNT AFTER
72	false	0
95	true	1
61	false	1
88	false	1
95	true	2

The display shows 2. The loop and the test are separate ideas: the loop visits all five elements, and only the test decides which ones count. If `count ← count + 1` sat outside the IF, every element would be counted.

Pattern 4: Filter a list

Build a new list by appending each element that passes a test. Start from an empty list.

```
ages ← [16, 21, 14, 19, 12]
adults ← []
FOR EACH a IN ages
{
  IF (a ≥ 18)
  {
    APPEND(adults, a)
  }
}
DISPLAY(adults)
```

A	A ≥ 18	ADULTS AFTER
16	false	[]
21	true	[21]
14	false	[21]
19	true	[21, 19]
12	false	[21, 19]

The display shows [21, 19]. The original list is never modified; filtering copies qualifying elements into a new list. Forgetting to initialize adults to [] is the usual bug, because APPEND needs a list to append to.

Procedures

A **procedure** is a named block of code that performs one task. You define it once with `PROCEDURE` and run it by calling its name. Procedures keep programs organized: each procedure handles one job, and the main program reads as a short list of calls.

Parameters are the placeholders listed in the definition. **Arguments** are the actual values supplied in a call. In the example below, `length` and `width` are parameters; 6 and 4 are arguments.

```
PROCEDURE rectangleArea(length, width)
{
    RETURN length * width
}
area ← rectangleArea(6, 4)
DISPLAY(area)
```

Trace the call: `rectangleArea(6, 4)` assigns 6 to `length` and 4 to `width`, then `RETURN` sends back $6 * 4 = 24$. The call expression evaluates to 24, so `area ← 24`, and the display shows 24.

RETURN ends the procedure immediately and hands a value back to the caller. That value can be stored, displayed, or used in an expression, as above. A procedure with no `RETURN` still runs its statements, but the call produces no value, so assigning the result to a variable is a logic error.

Procedures manage complexity through **abstraction**. The caller needs to know what `rectangleArea` does and what arguments it takes, not how it multiplies. When a program is built from well-named procedures, each piece can be understood, tested, and fixed on its own. This is the same idea the Create Performance Task asks you to explain about your own program.

Trap. Parameters and arguments are different roles, not different things. A question that asks for the parameter wants the name in the definition; a question that asks for the argument wants the value in the call. Swapping the words loses the point.

Debugging and Testing

Testing means running a program on chosen inputs and comparing the actual output to the expected output. A **test case** is one input together with the output you expect. Good test cases include normal inputs and **boundary cases**: an empty list, a list with one element, values sitting exactly on a condition's edge, and the largest input you expect.

For the counting pattern on the previous page, boundary cases would be a list where no score reaches 90, a list where every score does, and a score of exactly 90. If any of these produce a surprise, the bug is usually in the condition or in the initialization.

Debugging is the process of finding and fixing the bug a test exposed. The reliable method is to trace by hand: take the failing test case, walk through the code line by line, and write down each variable's value until the actual behavior diverges from what you expected. Most exam debugging questions are solved this way.

Trap. Testing can show that a bug exists, but it can never prove a program is correct. Passing ten test cases means the program handles those ten cases. The eleventh can still fail. The exam tests this distinction directly.

Algorithm Efficiency: A First Intuition

Some algorithms do more work than others, and the difference shows as inputs grow. Finding the maximum of a list checks each element once, so a list twice as long takes about twice as long. An algorithm with a loop inside a loop does far more work: doubling the input roughly quadruples the running time.

The exact vocabulary of reasonable and unreasonable running times belongs to Unit 4. For now the intuition is enough: work that grows in step with the input stays manageable, while work that multiplies with the input becomes impractical fast. When the exam asks why one approach is better, count how many times the inner statements run.

Simulations

A **model** is a simplified representation of a real system: it keeps the parts you think matter and leaves out the rest. A **simulation** is a program that runs a model over time, applying the model's rules step by step to see what it predicts.

Simulations stand in for experiments you cannot or should not run. Engineers crash virtual cars instead of real ones. Meteorologists run tomorrow's atmosphere on a supercomputer instead of waiting for it. Epidemiologists test disease policies in software instead of on people. Each replaces an experiment that would be too dangerous, too expensive, or too slow.

A simulation is only as good as its model, and every model simplifies reality. The assumptions baked into the model show up in every result, a problem called **simulation bias**. A traffic simulation that assumes identical one-second reactions will praise a light timing that fails on real roads. Simulation output is therefore evidence to weigh, not ground truth to trust. Serious simulations run many trials and check their predictions against real data.

```
heads ← 0
REPEAT 4 TIMES
{
  IF (RANDOM(1, 2) = 1)
  {
    heads ← heads + 1
  }
}
DISPLAY(heads)
```

This simulates four coin flips, counting 1 as heads. Suppose the draws come out 2, 1, 1, 2: the first draw leaves heads at 0, the next two raise it to 2, and the last leaves it at 2. The display shows 2.

Trap. A simulation's output is not a measurement of the real world. It is what the model's assumptions predict, so a model that leaves out an important factor will be confidently wrong.

Undecidable Problems and Heuristics

An **undecidable problem** is a problem no algorithm can solve correctly for every possible input. The classic example is the halting problem. Given any program and any input, decide whether the program will eventually finish or run forever. It has been proven that no algorithm gets this right for every program. That is a mathematical result, not a gap in effort.

Do not confuse undecidable with merely slow. Some problems have correct algorithms that take an unreasonable amount of time as inputs grow, which is what the efficiency section above is about. An undecidable problem is different in kind: no correct-for-all-inputs algorithm exists, no matter how much time you allow.

When an exact answer is out of reach, programmers use a **heuristic**: a practical approach that finds a good-enough answer in reasonable time. Planning delivery routes for a hundred trucks is a famous case. Computing the truly shortest route would take far too long, so a program might always drive to the nearest unvisited stop next, getting a short route quickly without guaranteeing the shortest one.

Trap. Undecidable does not mean the algorithm is still undiscovered. It means no such algorithm exists. A heuristic does not solve an undecidable problem. It sidesteps it with a good-enough answer.

The Create Performance Task

The Create Performance Task is 30 percent of your AP score. You design and build a program, record a short video of it running, and write responses explaining what you built. The code requirements map directly onto this unit, so every section above is preparation.

REQUIREMENT	WHAT IT MEANS IN PRACTICE
Program input	The program takes in data, for example with <code>INPUT()</code> , instead of running on fixed values alone.
Program output	The program produces something visible, for example with <code>DISPLAY()</code> .
At least one list	A list is used to manage data, not just declared. The list should do real work in the program.
At least one procedure	A procedure you wrote is called in the program. Built-in operations alone do not count.
Selection	An IF statement that chooses between paths based on data.
Iteration	A loop that repeats work, such as traversing the list.

The **video** demonstrates the program running. Show the input going in and the output coming out, with the list and the procedure visibly doing their jobs. A screen recording with a brief spoken explanation is the standard format.

The **written responses** ask for four things. Purpose: what the program does and who it is for. Development process: one difficulty you hit while building it and how you resolved it, with specifics. Algorithm: describe your procedure's logic step by step and show two different calls that produce different results, so the reader sees the parameters matter. Abstraction: explain how your list or your procedure manages complexity, meaning what detail it hides and what it lets the rest of the program ignore.

Trap. The written responses are scored on specifics, not adjectives. A response that names the procedure, the iteration, and the selection it uses earns credit. A response that only calls the code efficient and well organized does not.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
<code>←</code> vs <code>=</code>	The arrow stores a value in a variable. The equals sign compares two values and produces true or false. Read them differently every time.
List indices start at 1	<code>list[1]</code> is the first element. There is no <code>list[0]</code> , and <code>list[LENGTH(list)]</code> is the last element.
REPEAT TIMES vs REPEAT UNTIL	REPEAT TIMES runs a fixed count. REPEAT UNTIL runs until its condition becomes true and always executes the body at least once.
FOR EACH vs index loops	FOR EACH hands you each value with no index. If you need positions, use REPEAT TIMES with an index variable starting at 1.
APPEND vs INSERT	APPEND adds to the end. INSERT places a value at an index and shifts the rest right. It never replaces.
Parameter vs argument	The parameter is the name in the PROCEDURE definition. The argument is the value passed in the call.
RETURN vs DISPLAY	RETURN sends a value back to the caller for further use. DISPLAY only shows a value on screen.
MOD vs /	MOD gives the remainder. Division gives the quotient. Divisibility questions are MOD questions.
AND / OR precedence	NOT runs first, then AND, then OR. Parenthesize anything you want read differently.
Testing vs proving	Tests reveal bugs. They never prove a program is correct for all inputs.
Simulation output vs real measurement	A simulation reports what the model predicts. It becomes evidence about the real world only after it is checked against real data.
Undecidable vs slow	A slow problem still has a correct algorithm. An undecidable problem has no algorithm that is correct for every input, no matter how much time you allow.
Heuristic vs exact algorithm	A heuristic finds a good-enough answer fast and guarantees nothing about the best answer. An exact algorithm guarantees the right answer but may take too long to be useful.

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the following program.

```
scores ← [8, 3, 9, 2]
total ← 0
FOR EACH s IN scores
{
  IF (s > 5)
  {
    total ← total + s
  }
}
DISPLAY(total)
```

What is displayed?

1. 17
2. 22
3. 12
4. 5

2. Consider the following program.

```
letters ← ["m", "a", "t", "h"]
DISPLAY(letters[2] + letters[4])
```

What is displayed?

1. th
2. ah
3. mt
4. ma

3. Consider the following program.

```
x ← 17
IF (x MOD 3 = 0)
{
    DISPLAY("A")
}
ELSE IF (x MOD 2 = 0)
{
    DISPLAY("B")
}
ELSE
{
    DISPLAY("C")
}
```

What is displayed?

1. A
2. B
3. C
4. Nothing is displayed

4. Consider the following program.

```
n ← 6
REPEAT UNTIL (n < 4)
{
    n ← n - 2
}
DISPLAY(n)
```

What is displayed?

1. 4
2. 0
3. 6
4. 2

5. Consider the following program.

```
PROCEDURE shipping(weight)
{
  IF (weight > 10)
  {
    RETURN 12
  }
  ELSE
  {
    RETURN 6
  }
}
DISPLAY(shipping(4) + shipping(15))
```

What is displayed?

1. 18
2. 12
3. 6
4. 24

6. Consider the following program.

```
nums ← [5, 10, 15]
APPEND(nums, 20)
INSERT(nums, 2, 7)
DISPLAY(LENGTH(nums))
```

What is displayed?

1. 6
2. 5
3. 4
4. 3

7. A program needs the same block of code in five different places. Which of the following best explains why writing a procedure with parameters is better than copying the block five times?

1. The procedure version runs faster than the copied code.
2. The procedure version uses fewer variables overall.
3. If the logic needs to change, it changes in one place and every call benefits.
4. Writing a procedure removes the need for iteration in the program.

- 8.** Which of the following programs meets the Create Performance Task code requirements?
1. A program that stores vocabulary words in a list and uses a REPEAT loop to DISPLAY each word, with no user input and no procedure.
 2. A program that defines two procedures and calls each one once, with no list, no input, no output, and no iteration.
 3. A program that reads a quiz score with INPUT(), stores it in a variable, and uses an IF statement to display "Pass" or "Fail".
 4. A program that reads ten temperatures into a list with INPUT(), uses a FOR EACH loop containing an IF statement to count how many are below freezing, and calls a procedure report(count) that DISPLAYs the result.

Answer Key

- 1. A.** Trace the loop: $s = 8$ passes ($8 > 5$), so $\text{total} = 8$; $s = 3$ fails; $s = 9$ passes, so $\text{total} = 17$; $s = 2$ fails. The display shows 17. B sums all four values, ignoring the IF entirely. C adds $3 + 9$, which follows no consistent reading of the condition. D adds $3 + 2$, the values that fail the test, applying the condition backwards.
- 2. B.** Indices start at 1, so `letters[2]` is "a" and `letters[4]` is "h", and `+` concatenates them to "ah". C uses zero-based indexing, reading `letters[1] + letters[3] = "mt"`. D reads the first index correctly but slips on the second, giving "ma". A slips on the first index, giving "th".
- 3. C.** $17 \text{ MOD } 3 = 2$, so the first test is false. $17 \text{ MOD } 2 = 1$, so the second test is false. The ELSE runs and C is displayed. A treats 17 as divisible by 3. B treats 17 as even. D forgets that the ELSE guarantees a branch runs.
- 4. D.** The condition is checked after each pass: $6 < 4$ is false, so n becomes 4; $4 < 4$ is false, so n becomes 2; $2 < 4$ is true, so the loop ends with $n = 2$. A stops after one pass, checking the condition before the body runs. B keeps subtracting after the exit condition is already true. C treats REPEAT UNTIL as a loop that skips its body when the condition starts false.
- 5. A.** `shipping(4)`: $4 > 10$ is false, so the procedure returns 6. `shipping(15)`: $15 > 10$ is true, so the procedure returns 12. The display shows $6 + 12 = 18$. B counts only the second call. C counts only the first call. D evaluates both calls as if both weights exceeded 10.
- 6. B.** APPEND gives [5, 10, 15, 20]. INSERT at index 2 shifts the rest right, giving [5, 7, 10, 15, 20]. LENGTH is 5. C treats INSERT as a replacement, leaving four elements. D ignores both operations. A counts an element that was never inserted.
- 7. C.** With the logic in one procedure, a fix or improvement is made once and every call benefits, which is how procedures manage complexity. A is wrong because a procedure call does not execute faster than the same statements written out. B is wrong because parameters are variables too, so the count does not shrink. D is wrong because procedures and iteration solve different problems.
- 8. D.** It takes input (the temperatures), produces output (the displayed count), stores data in a list, calls a procedure, selects with an IF, and iterates with FOR EACH. C has no list, no procedure, and no iteration. A has no input and no procedure. B has no list, no input, no output, and no iteration.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the AP Computer Science Principles deck has flashcards for these algorithms and list operations and more practice questions use the same traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the AP Computer Science Principles deck and let spaced review bring them back over the next few days.

- Explain what $\leftarrow$ does and how it differs from $=$.
- Name the five data types and give an example of each.
- Evaluate $2 + 3 * 4 \text{ MOD } 5$ by hand, showing the precedence order.
- State which values `RANDOM(1, 6)` can return.
- State the precedence order for comparisons, NOT, AND, and OR.
- Explain how an ELSE pairs with its IF in nested conditionals.
- Name the three loops and say when each one is appropriate.
- Explain why REPEAT UNTIL always runs its body at least once.
- State the first and last valid index of a list with n elements.
- Describe what INSERT does to the elements after the insertion point.
- Trace the maximum-finding pattern on `[3, 11, 5, 9]` from scratch.
- Explain why `maxVal` starts at `vals[1]` rather than 0.
- Trace the filter pattern and state what the original list looks like afterward.
- Define parameter and argument, and give an example of each.
- Explain what RETURN does and how it differs from DISPLAY.
- Describe what a boundary test case is and give two examples.
- Explain why passing tests do not prove a program correct.
- Define model and simulation, and say how they differ.
- Explain simulation bias and why results must be checked against real data.
- Define undecidable problem and say how it differs from a merely slow one.
- Define heuristic and explain when a programmer would use one.
- List the six code requirements of the Create Performance Task.
- Explain how a procedure manages complexity through abstraction.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the AP Computer Science Principles deck. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

Variable, Assignment, Integer, Real (float), Boolean, String, List, Arithmetic operators, MOD, RANDOM, Operator precedence, Boolean expression, Comparison operators, NOT, AND, OR, Concatenation, Conditional, Nested conditional, Iteration, REPEAT TIMES, REPEAT UNTIL, FOR EACH, Index, Off-by-one error, APPEND, INSERT, REMOVE, LENGTH, Traversal, Linear search, Procedure, Parameter, Argument, RETURN value, Abstraction, Test case, Boundary case, Debugging, Algorithm efficiency, Simulation, Model, Simulation bias, Undecidable problem, Heuristic, Create Performance Task.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science Principles course framework, Unit 3. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.