Engineering Confidence

Selection & Iteration

A worked guide to choosing the next move

AP COMPUTER SCIENCE A · UNIT 2 · EFFECTIVE FALL 2025 · 25–35% OF MCQ

Boolean expressions · if, else & nesting · while & for loops · Standard algorithms & String algorithms · Nested loops & execution counts

A reference for the whole unit — not a single session's homework, but built the way my session guides are built: tool selection first, every step shown, commentary where the thinking usually goes wrong.

engineeringconfidence.one

READ THIS FIRST

About this guide. Unit 2 is the largest unit on the exam — 25–35% of the multiple-choice section, and the whole of the first free-response question — and it is where a program stops being a list of lines and starts making decisions and repeating itself. Every question in it is one of two acts: *read* a segment with an `if` or a loop in it and say exactly what it does, or *write* one to a specification. Both acts have the same enemy, and it is not Java. It is reading code the way you read a paragraph, taking in its general shape, when the machine reads it one condition at a time and does exactly what the condition says.

So this guide is built around two pieces of paper. The **trace table** — one column per variable, one row per statement, and a column for the condition every time one is tested — is how a selection question is answered without guessing. The **loop table** — one row per pass, the condition's value, the body's effect, the variables afterward — is how a loop question is answered without counting on your fingers. The College Board says it in as many words: "have students write program code on paper and trace that code with different sample inputs." Every worked problem below does exactly that.

Three mistakes to learn to catch:

- **The multiway that is not one.** Three separate `if` statements are not an `if-else-if`; each is tested, and the last true one wins. The ordering of the ranges decides the answer, and the exam writes both versions.
- **The loop counted on fingers.** "It runs five times" — or four, or six. Off-by-one errors are the CED's own named failure, and they come from not writing the loop table: the condition is tested *before* every pass, including the first, and the update runs *after* the body.
- **The String traversed one past its end.** `i <= s.length()` throws; `i < s.length() - 1` misses the last character; and a substring of length two needs `i < s.length() - 1`. Every String algorithm in the unit lives or dies on that boundary.

Every topic in the unit, and where it lives. The College Board lists twelve; here is each one and the page that teaches it.

- **2.1** sequencing, selection, repetition — card 2 (p. 8).
- **2.2** Boolean expressions and the relational operators; `==` on primitives against references — card 3 (p. 8); Problems 1 and 3.
- **2.3, 2.4** `if`, `if-else`, nested and multiway selection — card 4 (p. 9); Problems 1 and 2.
- **2.5** compound Boolean expressions and short-circuit evaluation — card 5 (p. 10); Problem 3.
- **2.6** equivalent expressions, De Morgan's law, comparing object references — card 6 (p. 11); Problem 3.
- **2.7** `while` loops, infinite loops, off-by-one — card 7 (p. 12); Problem 4.

- **2.8** `for` loops and their three parts — card 8 (p. 13); Problem 5.
- **2.9** the standard algorithms: divisibility, digits, counting, min and max, sum and average — card 9 (p. 14); Problems 4 and 6.
- **2.10** String algorithms — card 10 (p. 14); Problems 7 and 9.
- **2.11** nested iteration — card 11 (p. 15); Problem 8.
- **2.12** statement execution counts — card 12 (p. 16); Problem 8.

How to use it. The decision tree, the symptom map and the Master Toolbox are reference, not assigned reading. Route the problem, open the matching card, and attempt the prompt — on paper, with the table — before reading the worked solution. No compiler is assumed anywhere in these pages, because none is available on the exam; the paper is the machine. What nine problems *cannot* do is give you enough repetitions; each is the clearest instance of its shape, so a problem you found easy is a signal to go write five more like it.

WHERE THE POINTS GO ON THIS UNIT

In 2026, 25% of the cohort scored a 5 and 23% scored a 1 — more students took a 1 than a 3. The free-response section pays for *production*, not recognition, and Unit 2 is where production starts: the first free-response question, "Methods and Control Structures," asks students "to write two methods or a constructor and one method of a given class," and "the method will require students to write iterative or conditional statements or both," with one of them "calling String methods." Problem 9 is an original task in exactly that form — and the thing to judge it against is its stated contract, boundary cases included, not whether it looks finished.

The trace is the answer. On the multiple-choice section "students will be asked to determine the result of a given program code segment based on specific input," and the distractors are built from the plausible misreadings — the loop that runs one time too many, the branch that a different ordering would have taken. The CED's remedy is the one this guide runs: "write program code on paper and trace that code with different sample inputs." A trace table with the condition's value written in is worth more than any amount of staring.

Early termination is a conditional inside the loop. The CED names the struggle: "students often struggle in situations that warrant variation in the Boolean condition of loops, such as when they want to terminate a loop early," and "if the order of the program statements is incorrect, the early termination may be triggered too early or not at all." The fix is card 7's pattern — a flag or a compound condition, updated at the right line — and the check is the loop table, again.

Off by one is a named error. Topic 2.7 lists it beside the infinite loop: "off by one errors occur when the iteration statement loops one time too many or one time too few." `<` against `<=`, `length()` against `length() - 1`, an update before the test instead of

> after — each is a point on this section, and each is caught by writing the first and last pass of the loop table.
>
> **Explain why it does not work, then fix it.** Skill 3.D — "explain why a code segment will not compile or work as intended and modify the code to correct the error" — is attached to four of this unit's twelve topics. The answer has two halves: the sentence naming what the code does instead, and the corrected line. Problems 2, 4 and 8 each carry a bug to name and mend.

Diagnostic Decision Tree

HOW TO READ A SEGMENT WITH A DECISION OR A LOOP IN IT

Run these *in order*. The first two choose the piece of paper; the rest are the checks to run on it.

1. Am I asked what it prints, returns, or does — or asked to write it? Reading: trace, never predict. Writing: the specification's nouns become variables, its verbs become the loop and the conditions, and the return type is the last line before you start. *Problems 1, 3–5, 8 read; 2, 6, 7, 9 write.*

2. Is there a condition? Write its value — `true` or `false` — in the table for *these* inputs before deciding what runs. Then: separate `if`s are each tested; an `if-else-if` stops at the first true; an `else` belongs to the nearest unmatched `if`; an `if` without braces controls exactly one statement. *Problems 1, 2.*

3. Is the condition compound? Precedence is `!`, then `&&`, then `||`. Evaluate left to right and stop when the answer is known — short-circuit — which is why `x != 0 && y / x > 2` is safe and the reverse order is not. *Problem 3.*

4. Is it == on something that is not a primitive? Then it compares references, not contents. Two Strings built separately can be `.equals` and not `==`. Use `==` for reference identity, including comparison with `null`; use `.equals` for String content. *Problem 3.*

5. Is there a loop? Write the loop table: the condition's value before each pass, the body's effect, the variables after. The condition is tested before the first pass, so a loop can run zero times; the update in a `for` runs after the body and before the next test. Count passes from the table, never from the header. *Problems 4, 5, 8.*

6. Does the body walk a String or an integer's digits? A String's indices run from `0` to `length() - 1`; a substring of length `k` starting at `i` needs `i <= length() - k`. An integer's digits come off the right with `% 10` and `/ 10`, until the number is `0`. *Problems 4, 7, 9.*

7. Is one loop inside another? The inner loop finishes every time the outer loop takes one step. Count executions row by row and add the rows; a triangle of rows is $1 + 2 + \cdots + n$. *Problem 8.*

Where to Look When You're Stuck

FIND THE SENTENCE THAT SOUNDS LIKE YOUR SITUATION

What is happening	First move	Where
"I don't know which branch runs"	Write the condition's value for these inputs. Then: separate `if`s all test; `else-if` stops at the first true.	p. 9, Prob. 1, 2
"My ranges overlap and the wrong label comes out"	Order the multiway from the most restrictive condition down, or make every range two-sided.	p. 9, Prob. 2
"I can't evaluate `!a && b || c`"	`!` first, then &&, then `||`; write the parentheses the precedence implies.	p. 10, Prob. 3
"Is this the same as that expression?"	A truth table, all four rows. Or De Morgan: push the `!` through and flip && with `||`.	p. 11, Prob. 3
"== gave me false for two equal Strings"	It compared references. `.equals` compares contents.	p. 11, Prob. 3
"I don't know how many times the loop runs"	The loop table: condition before each pass, body, variables after. Count the rows.	p. 12, Prob. 4, 5
"It never stops"	The condition never turns false: the variable it tests is not changing, or changes the wrong way.	p. 12, Prob. 4
"I need to leave the loop early"	A conditional inside the body that changes what the loop's condition tests — a flag, or a compound condition — at the right line.	p. 12, Prob. 7
"`for` or `while`?"	A known count: `for`. Until something happens: `while`. Either converts to the other.	p. 13, Prob. 5
"Find the max, sum, count, digits"	The standard algorithms, each three lines; initialize the max from the first value, not zero.	p. 14, Prob. 6
"Walk a String"	`substring(i, i + 1)` from `0` to `length() - 1`; a two-character window stops one earlier.	p. 14, Prob. 7, 9
"How many times does the inner statement run?"	Rows for the outer loop, a count per row for the inner, add.	p. 16, Prob. 8

If none of those is your sentence, the last section of this guide is the longer version, organized by what went wrong.

Master Toolbox — Everything These Problems Use

Use these cards as you work. Name the decision you need to make, then open the relevant card. Each card stands on its own; you do not need to read the whole toolbox before attempting a problem.

1. THE TWO PIECES OF PAPER: THE TRACE TABLE AND THE LOOP TABLE

Every question in this unit is answered on one of two tables, and the first move is to draw it.

The trace table — for straight-line code with decisions in it. One column per variable, one row per statement executed, and when a condition is reached, a row that records its value:

line	condition	t	s
`int t = 75;`		75	
`String s = "cold";`		75	"cold"
`if (t >= 50)`	75 >= 50 → true	75	"cold"
`  s = "mild";`		75	"mild"
`if (t >= 70)`	75 >= 70 → true	75	"mild"
`  s = "warm";`		75	"warm"

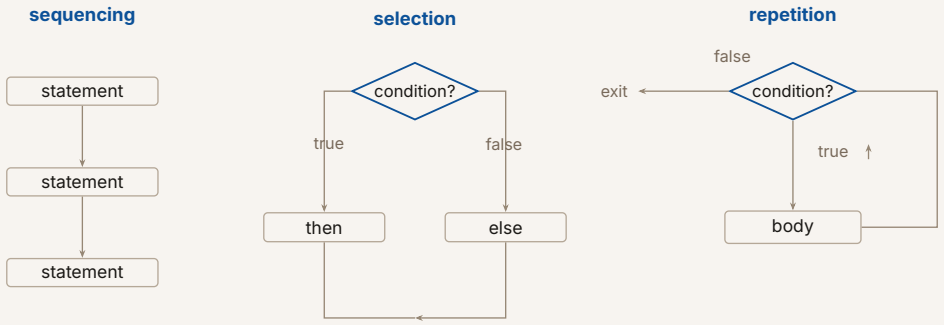
The condition column is the whole point: a branch is taken because a comparison came out `true` for *these* values, and writing the comparison down is what stops you from taking the branch the code's shape suggests.

The loop table — for anything that repeats. One row per pass: the condition's value *before* the pass, what the body did, and every variable's value after. The first row is the test before the first pass (a loop can run zero times); the last row is the test that came out `false`. Count the rows and you have the number of passes, which is the number the exam asks for and the number fingers get wrong.

The verbs. *Determine the output* — the table, then the exact text, spacing and line breaks included. *Describe the behavior* — what the segment does for any input, in one sentence with the variables named. *Explain why it does not work and fix it* — what it does instead, then the corrected line. *Write the method* — the header first, the return last, the loop and its bounds in between.

2. SEQUENCING, SELECTION, REPETITION — THE THREE BLOCKS EVERY ALGORITHM IS BUILT FROM

An algorithm is a step-by-step process, and the CED names its three building blocks. **Sequencing** runs statements one after another (Unit 1). **Selection** chooses between paths on a true-or-false decision. **Repetition** runs a segment again and again until a condition changes. The order in which they are combined is the algorithm; the same three blocks in a different order do something different.



The exam's habit is to hand you a description in words — "a game that reacts to the player," "a filter that keeps only the records that match" — and ask which block it needs. *Reacts to* and *only if* are selection; *for each*, *until*, *keep going* are repetition; *first this, then that* is sequencing.

3. BOOLEAN EXPRESSIONS: RELATIONAL OPERATORS, AND WHAT == ACTUALLY COMPARES

A **relational operator** compares two values and produces a `boolean`: `==` and `!=` ask whether values are the same; `<`, `>`, `<=`, `>=` compare numbers. The result is a value like any other — it can be stored in a `boolean` variable, printed, or returned — and a method whose whole body is `return x > 0;` is complete.

What == compares depends on the type. On primitives (`int`, `double`, `boolean`) it compares the *values*: `7 == 7` is `true`. On reference types it compares the *references* — whether two variables point at the *same object* — and two objects with identical contents built separately are not the same object. For a String's contents the question is `.equals`:

expression	asks	use it for
`a == b` (primitives)	are the values equal?	numbers, booleans
`a == b` (references)	are they the same object?	`null` checks; "is this the same one?"
`a.equals(b)`	do the contents match?	Strings; objects with an `equals`
`a != b`	the opposite of the matching `==`	

Two idioms. `if (s != null && s.length() > 0)` checks for an object before asking it anything, in that order (card 5 says why the order matters). And `if (flag == true)` is `if (flag)`; the comparison adds nothing and is a place to write `=` by accident.

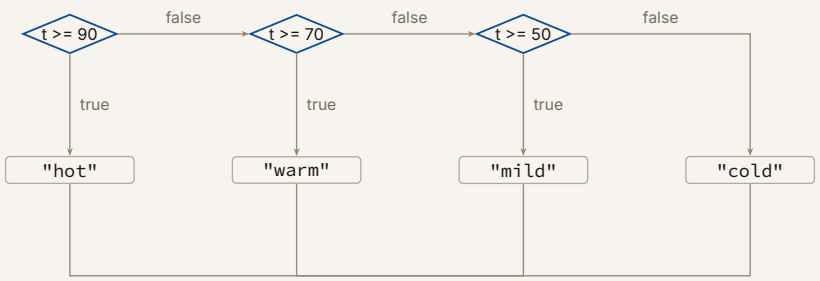
A check first. One question opens each of the next three cards and is answered where that card ends. Answer it before you read on; getting it wrong is the point, because that is what makes the card stick.

1. TRACE EVERY TEST

Start with `x = 1`. Run two *separate* statements — `if (x > 0) x = -x;` then `if (x < 0) x = 0;` — and then run the same two conditions as one `if / else if` chain. What does `x` finish at each time?

4. IF, IF-ELSE, NESTED, MULTIWAY — AND THE RULE THAT THE FIRST TRUE WINS

A **one-way** `if` runs its body when the condition is `true` and skips it otherwise. A **two-way** `if-else` runs exactly one of two bodies. A **nested** `if` sits inside another's body, and its condition is tested *only* if the outer condition was `true`. A **multiway** `if-else-if` is a chain: the conditions are tested in order, the first `true` one's body runs, and *no other body runs* — a trailing `else` catches the case where none was true.



exactly one label is assigned; the chain stops at the first true

Because the chain stops, the second condition is really "`t < 90 && t >= 70`" — the earlier conditions' failures are implied. That is why the order matters: the same four tests as separate `if`s, from the largest threshold down, each overwrite the last, and `t = 95` comes out `"mild"`. Problem 1 traces both.

Three traps in the syntax. An `if` without braces controls exactly the next statement; indent all you like, the second line runs unconditionally. An `else` pairs with the *nearest* unmatched `if` above it, whatever the indentation says — braces are how you say otherwise. And a semicolon after the condition, `if (x > 0);`, ends the `if` with an empty body, so the next line runs every time.

Reading a nested `if` as a multiway. `if (a) { if (b) X else Y } else Z` is the same as "`a && b`: X; `a && !b`: Y; `!a`: Z" — and the exam asks you to rewrite in both directions. Problem 2 does.

CHECK 1 — TRACE EVERY TEST

`0` the first way, `−1` the second. Separate statements each get their own look at the variable *as it stands then*: the first flips 1 to −1, and the second now sees a negative *x*, so it fires too and stores 0. In an `if / else if` chain only the first true branch runs; once `x > 0` has been taken the chain is finished, so −1 survives. Same two conditions, same starting value, different answers — and the difference is not in the conditions at all, it is in whether a taken branch closes the others. Two habits follow. Write the new value of the variable on its own line the moment an assignment changes it, and never evaluate the second condition against the value the first one replaced. And when a chain is what you meant, say `else if` on purpose; the compiler will not tell you which one you wrote.

5. COMPOUND EXPRESSIONS: `!`, `&&`, `||`, AND SHORT-CIRCUIT EVALUATION

The logical operators combine booleans: `!a` flips; `a && b` is `true` only when both are; `a || b` is `true` when either is, or both. Precedence runs `!`, then `&&`, then `||`, so `!a && b || c` means `((!a) && b) || c`. Write the parentheses the precedence implies before evaluating, every time.

a	b	!a	a && b	a || b	!(a && b)
T	T	F	T	T	F
T	F	F	F	T	T
F	T	T	F	T	T
F	F	T	F	F	T

Short-circuit evaluation. Java evaluates a compound left to right and *stops as soon as the answer is known*: in `a && b`, if `a` is `false` the whole thing is `false` and `b` is never evaluated; in `a || b`, if `a` is `true`, `b` is never evaluated. This is not an optimization you can ignore — it is the reason

```
x != 0 && y / x > 2
```

is safe when `x` is `0` (the division never happens) while `y / x > 2 && x != 0` throws an `ArithmeticException` first. The same guard protects a `null`: `s != null && s.length() > 3`. Put the guard on the left.

Reading a compound in words. "Between 10 and 20" is `x >= 10 && x <= 20`; "outside" is `x < 10 || x > 20`. "Not between" is the negation of the first, which card 6 turns into the second. A student who writes `10 <= x <= 20` has written something that does not compile; Java compares two values at a time.

6. EQUIVALENT EXPRESSIONS, DE MORGAN'S LAW, AND COMPARING REFERENCES

Two Boolean expressions are **equivalent** when they evaluate the same in every case, and a **truth table** — every combination of the variables, both expressions evaluated — is the proof. For two variables that is four rows; for three, eight. The exam asks "which of the following is equivalent to ..." and the honest method is the table; the fast method is the law below, checked against one row.

De Morgan's law moves a `!` through a compound by flipping the operator and negating each part:

```
!(a && b) ≡ !a || !b,    !(a || b) ≡ !a && !b.
```

Combined with negating a comparison — `!(x < 5)` is `x >= 5`, `!(x == y)` is `x != y` — it rewrites any negated condition into one with no `!` at all:

```
!(x < 5 || y >= 10) ≡ x >= 5 && y < 10.
```

The error the exam plants is flipping the operator without negating the parts (`!a && !b` for `!(a && b)`), or negating the parts without flipping the operator. Both fail on a row of the table; check one.

Comparing references. Two variables can hold references to the same object; `==` and `!=` ask exactly that. A reference compared with `null` says whether there is an object at all. Classes define their own `equals` to say what "equivalent" means for their objects — usually that the attributes match — and that is the method to call when contents are the question. For Strings this is the difference between `a == b` (same object?) and `a.equals(b)` (same characters?), and the exam writes both.

7. `while` LOOPS: ZERO OR MORE TIMES, AND THE TWO WAYS THEY GO WRONG

A `while` loop tests its condition *before* every pass, including the first; when the condition is `true` the body runs and the test happens again; when it is `false` the loop ends. So a loop whose condition is `false` at the start runs **zero times**, and a loop whose condition never becomes `false` is an **infinite loop** — usually because the body never changes the variable the condition tests, or changes it the wrong way.

The loop table is the method, and here it is for the digit-sum of `n = 4729` with `while (n > 0):`

pass	test	body	sum	n
		(before the loop)	0	4729
1	`4729 > 0` T	`sum += 9; n /= 10`	9	472
2	`472 > 0` T	`sum += 2; n /= 10`	11	47
3	`47 > 0` T	`sum += 7; n /= 10`	18	4
4	`4 > 0` T	`sum += 4; n /= 10`	22	0
	`0 > 0` F	(exit)	22	0

Four passes, five tests. The row that says `F` is the one students forget to write, and it is the row that decides the count.

Off by one. The CED names it: "the iteration statement loops one time too many or one time too few." `while (n > 9)` on the table above stops with `n = 4` unsummed — one pass too few; `i <= s.length()` on a String is one pass too many and throws. The cure is the first and last rows of the table.

Leaving early. A loop that should stop when something is found needs a conditional *inside* the body that changes what the condition tests — a `boolean found` flag set to `true`, with `while (i < n && !found)` — or a compound condition that tests the thing directly. The CED's warning is about order: update the flag at the line where the discovery is made, and the test sees it on the next pass, not this one.

2. COUNT EXECUTIONS, NOT LABELS

For `for (int i = 0; i < 5; i++)`, list the values `i` takes while the body runs. Then a second count: how many times is the *condition* tested, assuming nothing exits early?

8. FOR LOOPS: THREE PARTS, ONE EXACT ORDER

A `for` header has three parts, and the CED spells out when each runs:

```
for ( int i = 0 ;      i < 5      ;      i++      )
      initialization, once   test, before every pass   update, after every body
```

the order on the timeline: initialize once; then test, body, update, test, body, update, ...; the loop ends at the first false test

The variable declared in the header is the **loop control variable**, and it exists only inside the loop. When the loop ends, `i` has the first value that failed the test — for the header above, `5`, not `4`.

The four headers the exam uses, with their values written out — write them out yourself until this is reflex:

header	values of `i`	passes
`for (int i = 0; i < 5; i++)`	0, 1, 2, 3, 4	5
`for (int i = 0; i <= 5; i++)`	0, 1, 2, 3, 4, 5	6
`for (int i = 1; i <= 10; i += 3)`	1, 4, 7, 10	4
`for (int i = 10; i > 0; i -= 3)`	10, 7, 4, 1	4

`for` and `while` are the same loop. Any `for` rewrites as: the initialization before a `while` whose condition is the test, with the update as the last line of the body — and the reverse. Use `for` when the number of passes is known from the start, `while` when the loop runs until something happens; the exam asks you to convert in both directions.

CHECK 2 — COUNT EXECUTIONS, NOT LABELS

The body runs at `i = 0, 1, 2, 3, 4` — five times. The condition is tested **six** times, because the run does not end until a test comes back false, and that is the test at `i = 5`. The two counts are different numbers and Topic 2.12 asks for whichever one the question named, so read the noun: *body executions*, *iterations*, *comparisons* and *tests* are not synonyms. The loop table is what keeps them apart — one row per test, with the value being tested, the verdict, and what the body did if it ran. The last row of an honest loop table always has a false verdict and an empty body column, and if yours does not, you stopped writing one row early.

9. THE STANDARD ALGORITHMS — AND THE INITIALIZATION THAT DECIDES EACH ONE

The CED lists five, and each is three lines once you know the line that comes *before* the loop:

algorithm	the test or the step	the line before the loop
divisible by k?	`n % k == 0`	—
the digits of n	`d = n % 10; n = n / 10;` until `n == 0`	a copy of n, if you still need it
count a criterion	`if (…) count++;`	`int count = 0;`
sum, then average	`sum += x;`	`int sum = 0;` and, for the average, a `double` cast at the end
minimum or maximum	`if (x > max) max = x;`	`max` set to the *first* value, never to `0`

The max initialized to zero is the unit's quietest bug: it is right for positive data and silently wrong the day the values are all negative — `max` stays `0`, a value that was never in the data. Initialize from the first element, or from `Integer.MIN_VALUE` when there is no first element to read.

The average is Unit 1's division. `sum / count` with two `int`s is an `int`; `(double) sum / count` is the average. Cast before dividing, not after.

Digits come off the right. `4729 % 10` is `9`, `4729 / 10` is `472`; repeating gives `2`, `7`, `4`, and the loop ends when the quotient reaches `0`. To get the digits in left-to-right order, count them first or build a String.

3. TEST THE FINAL START

A string has length n and you want to look at every length-3 substring. What is the last valid starting index, should the loop include it, and what happens when $n < 3$?

10. STRING ALGORITHMS: THE TRAVERSAL, THE WINDOW, AND THE BOUNDARY

A String is walked one character at a time with `substring(i, i + 1)` — Unit 1's one-character substring — for `i` from `0` to `length() - 1`. Three things are built on that walk, and the CED names them:

- **does a substring with a property exist** — walk, test each window, set a flag or return early;
- **count the substrings meeting a criterion** — walk, `count++` on each hit;

- **reverse the characters** — walk, and build `rev = ch + rev` so each new character lands in front.

The window's boundary. A window of `k` characters starting at `i` is `substring(i, i + k)`, and it exists only while `i + k <= length()`, that is, `i <= length() - k`. For `k = 2` the loop is `for (int i = 0; i < s.length() - 1; i++)`. Off by one here is not a wrong count; it is a `StringIndexOutOfBoundsException`.

The other traversal: `indexOf`. To walk *words*, find the next space and cut: `int sp = s.indexOf(" "); String w = s.substring(0, sp); s = s.substring(sp + 1);` — and the last word has no space after it, so `indexOf` returns `-1` and the whole remainder is the word. The CED's own sample activity is the *n*th word; Problem 7 writes it.

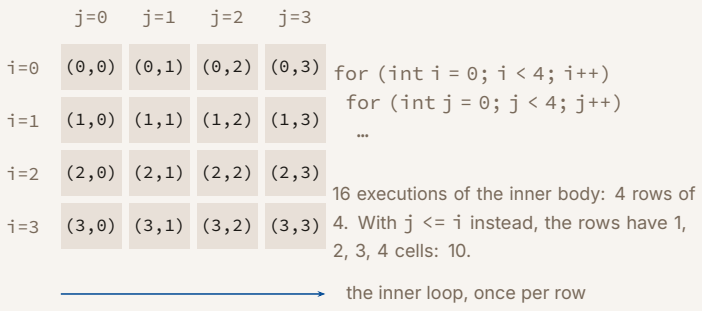
Strings are immutable, so every "change" is a new String assigned back: `s = s.substring(1)` shortens `s`; `s.substring(1)` on its own does nothing you can see.

CHECK 3 — TEST THE FINAL START

The last valid start is $n - 3$, and yes, the loop must include it — so the condition is `i <= n - 3`, or equivalently `i < n - 2`. Get there by the far end rather than by the near one: the window that starts at i ends at `substring(i, i + 3)`, the excluded endpoint may equal the length, so `i + 3 <= n` and $i \le n - 3$ falls straight out. When $n < 3$ that bound is negative, no start is valid, and the loop should simply not run — which it does not, since $0 \le n - 3$ is already false. That is the check worth making on every windowed loop: put in the smallest input the problem allows and confirm the loop body never runs rather than running once and throwing. Writing `i < n` instead is the standard off-by-one here, and it does not give a wrong answer — it gives a `StringIndexOutOfBoundsException` on the last two starts.

11. NESTED ITERATION: THE INNER LOOP FINISHES FIRST

When a loop is inside another loop, the inner loop runs *all* of its passes every time the outer loop takes *one* step. The picture is a grid: the outer variable picks the row, the inner variable walks across it.



Two shapes the exam draws. Independent bounds — a rectangle, $m \times n$ executions. An inner bound that depends on the outer variable, `j <= i` or `j = i`, — a triangle, $1 + 2 + \cdots + n = n(n+1)/2$ executions. Output patterns of stars, digit tables and pairwise comparisons are all one of the two.

Reading the output. A `print` inside the inner loop and a `println()` after it, inside the outer loop, makes one line per outer pass. Trace one full outer pass by hand, then trust the pattern.

12. INFORMAL RUN-TIME ANALYSIS: COUNTING STATEMENT EXECUTIONS

A **statement execution count** is how many times a statement runs, found by tracing the loops around it. For a single loop it is the number of passes; for nested loops, the sum over the outer passes of the inner passes. The exam asks it two ways: "how many times is the statement executed," and "which segment executes the statement more times."

Three counts worth knowing cold:

loop	executions of the body
`for (int i = 0; i < n; i++)`	n
`for (int i = 0; i < n; i += 2)`	$n/2$, rounded up
nested, `i < n` outside and `j < n` inside	n^2
nested, `i < n` outside and `j <= i` inside	$1 + 2 + \cdots + n = n(n+1)/2$

For anything else, the loop table is the method: write the passes and count the rows. "Informal" means a number for this n, not a formula for all of them — the exam gives $n = 10$ and wants 55.

PROBLEM 1

The multiway that is not one: tracing a chain of conditions

Two methods are meant to label a temperature. Segment A:

```
public static String labelA(int t) {
    String s;
    if (t >= 90) {
        s = "hot";
    } else if (t >= 70) {
        s = "warm";
    } else if (t >= 50) {
        s = "mild";
    } else {
        s = "cold";
    }
    return s;
}
```

Segment B:

```
public static String labelB(int t) {
    String s = "cold";
    if (t >= 90) { s = "hot"; }
    if (t >= 70) { s = "warm"; }
    if (t >= 50) { s = "mild"; }
    return s;
}
```

(a) Trace `labelA` for `t` equal to 95, 70, 69 and 49, writing the value of every condition tested. (b) Trace `labelB` for 95, 75, 55 and 20. (c) Explain, in one sentence each, why the two segments disagree and what `labelB` actually computes. (d) Fix `labelB` without changing it into an `else-if` chain.

> BEFORE YOU COMPUTE
>
> Rung 2: conditions, so the trace table has a condition column, and the answer for each input is whatever that column says — not what the shape of the code suggests. Before tracing, decide how each segment *stops*: A is a chain, and the first `true` ends it; B is three separate statements, and every one of them is tested. Write that down before the first row, because it is the whole difference.

WORKING

(a) `labelA`: the chain stops at the first `true`.

t	conditions tested, in order	first true	returns
95	`95 >= 90` T	the first	`"hot"`
70	`70 >= 90` F, `70 >= 70` T	the second	`"warm"`
69	`69 >= 90` F, `69 >= 70` F, `69 >= 50` T	the third	`"mild"`
49	`49 >= 90` F, `49 >= 70` F, `49 >= 50` F	none — the `else`	`"cold"`

Notice `t = 70`: the second condition is `true`, so the third is never tested, even though `70 >= 50` is also `true`. That is what a chain means.

(b) `labelB`: all three are tested, and each `true` one overwrites `s`.

t	>= 90	>= 70	>= 50	s after each, then returned
95	T: `"hot"`	T: `"warm"`	T: `"mild"`	`"mild"`
75	F	T: `"warm"`	T: `"mild"`	`"mild"`
55	F	F	T: `"mild"`	`"mild"`
20	F	F	F	`"cold"`

Ninety-five degrees comes out `"mild"`. The last `true` condition wins, and for any `t >= 50` the last true condition is the third one.

(c) They disagree because A tests its conditions until one is true and stops, while B tests all three and lets each true one overwrite the previous label. What B actually computes is: `"mild"` for every `t >= 50`, and `"cold"` otherwise — the first two `if`s can never survive to the `return`.

(d) Separate `if`s work when the *last* true condition should win, so reverse the order — test the smallest threshold first, so that the largest one true overwrites last:

```
String s = "cold";
if (t >= 50) { s = "mild"; }
if (t >= 70) { s = "warm"; }
if (t >= 90) { s = "hot"; }
return s;
```

Now `t = 95` passes all three, and the last overwrite is `"hot"`. Three tests always run, so this costs more than the chain, but it is correct.

ANSWER

(a) `"hot"`, `"warm"`, `"mild"`, `"cold"` (b) `"mild"`, `"mild"`, `"mild"`, `"cold"` (c) A stops at the first true; B lets the last true overwrite, so every `t >= 50` is `"mild"` (d) test the thresholds in increasing order

WATCH OUT

Reading B as if it were A — "95 is hot, obviously" — is the error this problem exists for, and the exam writes B on purpose. The tell is the missing `else`: without it, nothing stops the later tests. Second, in (a): stopping the trace at the first condition for `t = 70` and writing `"hot"` because 70 "is warm, so the first branch..." — write the comparison, `70 >= 90`, and its value. Third: a trace that skips the condition column and records only `s` looks complete and cannot be checked; the column is where the grader, and you, see the reasoning.

CONNECTION

Segment B's pattern — separate `if`s where a later one is meant to override — is the shape of every "find the largest" and "update the best so far" loop in card 9: the order of the tests is the algorithm. And a chain's implied conditions (the second branch really means `t < 90 && t >= 70`) are card 6's equivalences in disguise; Problem 2 rewrites a nested selection into exactly those compound conditions.

ABOUT THIS EXCERPT

This is the opening of a 41-page guide: the diagnostic tree, the full Master Toolbox, and the first worked problem. 8 more problems follow in the complete guide, each worked the same way — what to notice before you start, every step shown, and the mistake that problem invites. The complete guide is shared with families during the fit conversation.

Engineering Confidence — engineeringconfidence.one