The Ultimate AP® Computer Science Principles Big Idea 3 Cheat Sheet

3.1–3.3 Variables, Abstraction & Math

Variables & Assignment

Variable: named abstraction holding 1 value at a time; meaningful names (score, not x1) → easier to read.
a ← expr: evaluate right side 1st, then store the result in a. Old value of a is overwritten + lost for good.
Data types: number, Boolean, string, list. A variable's value can be any of these + can change type when reassigned.
Trace e.g.: x ← 3, y ← x, x ← 7 → y is still 3 (y got a copy of x's value at that moment, not a link to x).
Swap needs a temp: temp ← a, a ← b, b ← temp. Writing a ← b, b ← a sets both to b's old value (a is lost).

Data Abstraction (Lists)

List: ordered sequence of **elements**; each has an **index**. AP lists start at **index 1**, not 0 as in Python.
Data abstraction: 1 name (scores) for many values → manage complexity; same code works for any size list.
Why lists: no need for score1, score2, score3...; add data w/o rewriting code; separate idea from storage.

Math Expressions

Expression	Value	Note
17 MOD 5	2	remainder
5 MOD 17	5	a < b → a
20 MOD 4	0	divisible
7 / 2	3.5	not cut to 3
2 + 3 * 4	14	* before +

MOD = same precedence as * and /; left to right. **n MOD 2 = 0** → even; = 1 → odd. n MOD 10 → last digit.
Order: ( ) → * / MOD → + −. **Sequencing**: statements run in the order written, top to bottom, 1 at a time.

⚠ ← assigns; = compares. In pseudocode, x = 5 is a Boolean test (true/false), never an assignment.

✓ Trace tables: 1 column per variable, 1 row per line run. Cross out old values; never trust memory.

3.4–3.5 Strings & Boolean Expressions

Strings

String: ordered sequence of characters, e.g., "AP CSP". Spaces, digits + symbols all count as characters.
Concatenation: joins 2+ strings end to end → "sun" + "day" = "sunday". No space is added for you.
Substring: part of an existing string, e.g., "put" in "computer". Exam defines its own string procedures.
Read the procedure def: does it count from 1? Is end index included? Trace w/ the exact rule given.
"5" ≠ 5: a string of digits is text, not a number; joining "5" and "5" gives "55", not 10. Watch types.

Boolean Expressions

Boolean: value is only true or false. **Relational ops**: =, ≠, >, <, ≥, ≤ → each one evaluates to a Boolean.

A	B	A AND B	A OR B	NOT A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

AND: true only if both sides true. **OR**: true if at least 1 side true (or both). **NOT**: flips true ↔ false.
Range check: (x ≥ 1) AND (x ≤ 10). Using OR here is always true → every x passes, even 500 or −3.
De Morgan: NOT (A AND B) = (NOT A) OR (NOT B); NOT (A OR B) = (NOT A) AND (NOT B). Flip op + each side.
Negating >: NOT (x > 5) = x ≤ 5, not x < 5. The boundary value 5 switches sides when you negate.

⚠ x = 3 OR 4 is not valid logic. Write (x = 3) OR (x = 4): each side must be a full comparison.

✓ Test Boolean MCQs by plugging in 1 value that should pass + 1 boundary value that should fail.

3.6–3.7 Conditionals & Nested Conditionals

Selection

Selection: picks which statements run based on a Boolean condition. Code may run or be skipped.
IF (cond) { }: block runs only if cond is true; if false, skip the whole block + go on to the next statement.
IF / ELSE: exactly 1 of the 2 blocks runs, every time. ELSE catches all cases where cond is false.
Separate IFs vs **IF / ELSE**: 2 separate IFs can both run (each is tested); IF / ELSE never runs both blocks.

Nested Conditionals

Nested: an IF inside another IF / ELSE → inner test is checked only if the outer branch holding it is reached.

score	Grade from nested IF
≥ 90	A (1st test true)
80–89	B (1st F, 2nd T)
70–79	C (1st 2 F, 3rd T)
< 70	F (final ELSE)

Order matters: test score ≥ 70 first → a 95 gets C. Check the most restrictive condition (≥ 90) first.
Equivalent forms: IF (a) {IF (b) {X}} = IF (a AND b) {X}. Many conditionals ↔ 1 Boolean expression.

Trace It

Method: plug in the value, evaluate each condition T/F, follow only the true path, then note the output.
Boundaries: test values right at the edge (e.g., 89 + 90) b/c off-by-one logic errors hide right there.
Missing ELSE: if no branch is true, nothing in the structure runs; all variables keep their old values.

⚠ An IF that reassigns a variable used by a later IF can change which branch runs next. Trace in order.

✓ WR 2(a): name the condition, say what input makes it true vs false, + what each branch then does.

3.8 Iteration (Loops)

Loop Types

Iteration: repeats a set of statements a set number of times or until a condition is met. Body = code inside.

Loop	Runs	Watch for
REPEAT n TIMES	exactly n	n = 0 → 0 runs
REPEAT UNTIL	till cond T	cond never T
FOR EACH	1×/element	list length

REPEAT UNTIL (cond): cond checked **before** each pass. If already true at the start → body runs 0 times.
Infinite loop: ending condition never becomes true (e.g., counter never changes) → never stops.
Loop counter: i ← 1, REPEAT UNTIL (i > n) {... i ← i + 1} runs n times. Forget i ← i + 1 → an infinite loop.

Trace a Loop

Count passes: UNTIL (i > 5) from i ← 1 → 5 passes; UNTIL (i ≥ 5) → 4 passes. Check the boundary.
Accumulator: sum ← 0 before loop; sum ← sum + x inside. Resetting sum inside loop → only last x kept.
Trace e.g.: x ← 1, REPEAT 3 TIMES {x ← x * 2} → 2, 4, 8. Final x = 8, not 6 (doubling each pass, not adding).
DISPLAY inside loop → shows a value on each pass; **after loop** → shows only the final value, 1 time.

Robot Grid Loops

CAN_MOVE(dir): true if open square in that dir (left, right, forward, backward) from robot's view.
Off-grid/blocked move: robot stays put + program ends. ROTATE_LEFT/RIGHT turns 90° in place only.

⚠ Robot MCQs: track the direction the triangle faces after every rotate. Left/right are the robot's, not yours.

✓ Loop MCQ: trace the 1st, 2nd + final passes in a table; check the exit condition one last time.

3.10 Lists

List Operations (Ref Sheet)

Op	Effect	Length
APPEND(L, v)	adds v at end	+1
INSERT(L, i, v)	v at i, shift →	+1
REMOVE(L, i)	drop i, shift ←	−1
L[i] ← v	replace at i	same
LENGTH(L)	# elements	same

Index range: valid i is 1 to LENGTH(L). Outside it → error message + program ends (run-time error).
INSERT/REMOVE shift later elements → their indices change. Re-check every position after each call.
Copy: aList ← bList assigns a copy of bList, so changing aList later does not alter bList at all (2 separate lists).

Traversals

Complete traversal: visit every element (FOR EACH item IN L). **Partial**: only some, e.g., the 1st half.
FOR EACH item IN L: item gets each value in order, 1st to last; item is a value, not an index. Can't use L[item].
Index loop: i ← 1, REPEAT LENGTH(L) TIMES {... L[i] ... i ← i + 1} → use when you need the position.

Standard List Algorithms

Sum/avg: total ← 0, add each item; avg ← total / LENGTH(L).
Count: c ← c + 1 each time an IF is true.
Max: max ← L[1], then IF (item > max) {max ← item}. Starting max at 0 fails if all values are < 0 (e.g., temps).
Filter: new ← [], APPEND items that pass a test → the original list stays unchanged + intact for later use.
Remove in loop: REMOVE shifts items left → the next item slides into i; don't add 1 to i that pass.

⚠ L[0] doesn't exist in AP pseudocode. Java/Python habits (index 0) cause off-by-one errors on the exam.

✓ WR 2(c): say what your list stores, + how the program would be harder to write without the list.

3.9 & 3.11 Algorithms & Binary Search

Building Blocks

Algorithm: finite set of instructions for a task. Every one = **sequencing** + **selection** + **iteration** combined.
Same goal, diff code: different algorithms can solve the same problem; similar-looking ones can differ.
Reuse: build on existing correct algorithms (max, sum, MOD test) → less dev time, testing, debugging.
Compare 2 algorithms: run both on the same input incl. edge cases (empty list, ties, negatives, 1 item).
Natural language vs code: steps in English or a flowchart can express the same algorithm as code.

Binary Search

Binary search: data **must be sorted**. Check middle; target smaller → keep left half, else keep right half.
Each check halves what's left → ≈ $\log_2(n)$ checks max.

Linear search: up to n checks (every element).

n items	Linear (worst)	Binary (max)
8	8	4
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20

Count checks: find smallest k w/ $2^k \geq n + 1$. E.g., $2^{10} = 1{,}024$ → a sorted list of 1,000 needs ≤ 10 checks.
Trade-off: sorting costs time; for a 1-time search of unsorted data, linear search may be the better pick.
Doubling n → binary search needs only 1 more check; linear search needs 2× as many checks in the worst case.

⚠ Binary search on unsorted data can miss the target even when it is in the list. Sorted is required.

✓ "Which search?" MCQ: sorted + large → binary. Unsorted or tiny → linear (no sorting step needed).

3.12–3.14 Procedures & Libraries

Calling Procedures

Procedure: named group of instructions w/ **parameters**; may **RETURN** a value (aka method/function).
Parameter: input variable named in the definition. **Argument**: actual value passed in when you call it.
Call: x ← proc(3, 5) → runs body w/ params = 3, 5; then flow returns to the call + x gets the RETURN value.
RETURN(expr): ends the procedure at once + sends the value back; later lines in it are skipped.

Term	In code
Definition	PROCEDURE f(a, b)
Parameters	a, b
Call + arguments	y ← f(2, 9)
Return value	RETURN(a + b)

Procedural Abstraction

Procedural abstraction: use a procedure knowing what it does, not how → manage complexity.
Modularity: split a program into procedures, each a subproblem → easier to read, test, fix + reuse.
Generalize: parameters let 1 procedure work for many inputs, e.g., area(w, h) vs 1 proc per room size.
Change inside: fix or speed up a procedure's body → callers don't change if behavior stays the same.

Libraries & APIs

Library: prewritten procedures to use in new programs. **API**: spec for how they behave + are used.
Docs are needed to use a library right: name, parameters, return value, what each procedure does.

⚠ DISPLAY ≠ RETURN. DISPLAY shows a value; RETURN hands it back so the caller can store or use it.

✓ WR 2(c): say how your procedure manages complexity, e.g., 1 call replaces repeated code w/ diff args.

3.15–3.18 Random, Sims, Efficiency, Limits

Random Values & Simulations

RANDOM(a, b): random integer from a to b, **inclusive**; each value equally likely. RANDOM(1, 6) = die.
Possible outputs: list every value a random call can make; output may differ on every run of same code.
Simulation: abstraction of a real process; drops details; uses random values to model real-world variability.
Sim pros: test ideas w/o real-world cost, danger or time.
Cons: simplified → may be biased or inexact.

Algorithmic Efficiency

Efficiency: computing resources (time, memory) used as input size grows; count steps or loop runs.

Growth	E.g.	Reasonable?
Constant/log	binary search	Yes
Linear (n)	linear search	Yes
Polynomial (n^2)	nested loops	Yes
Exponential (2^n)	try all subsets	No

Reasonable time: polynomial or slower growth.
Unreasonable: exponential/factorial → impractical for big n.
Decision problem: yes/no answer. **Optimization problem**: find the best solution (e.g., shortest path).
Heuristic: approach that finds a good-enough answer when exact is too slow (e.g., traveling salesperson).

Undecidable Problems

Decidable: some algorithm gives a correct yes/no answer for every input (e.g., is n even? → n MOD 2 = 0).
Undecidable: no algorithm can always give a correct yes/no for all inputs, e.g., the **halting problem**.
Some instances of an undecidable problem can still be solved; it's "all cases" that's impossible.

⚠ Undecidable ≠ just slow. Faster computers or more time never make an undecidable problem solvable.

✓ Efficiency MCQ: count how many times the innermost line runs as n grows; nested loop over n → n^2.