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

3.1–3.3 Variables, Abstraction & Math

Variables & Assignment

Variable: a named box that holds 1 value at a time. Use clear names (score, not x1).

a ← expr: work out the right side 1st, then store it in a. The old value of a is gone.

Trace e.g.: x ← 3, y ← x, x ← 7 → y is still 3 (y got a copy of the value, not a link).

Swap needs a temp: temp ← a, a ← b, b ← temp. Skip temp → both end up w/ b's value.

Data Abstraction (Lists)

List: ordered group of **elements**, each w/ an **index**. AP lists start at **index 1**, not 0.

Data abstraction: 1 name (scores) stands for many values → code is simpler + works for any size.

Math Expressions

Expression	Value	Why
17 MOD 5	2	remainder
20 MOD 4	0	divisible
7 / 2	3.5	real division
2 + 3 * 4	14	* before +

MOD (remainder) ranks w/ * and /. **n MOD 2 = 0** → n is even; n MOD 2 = 1 → n is odd.

Sequencing: statements run 1 at a time, in the exact order written, from top to bottom.

⚠ ← means "store"; = means "compare." x = 5 is a true/false test, not an assignment.

✓ Trace in a table: 1 column per variable. Cross out old values as they change.

3.4–3.5 Strings & Boolean Expressions

Strings

String: ordered sequence of characters, e.g., "AP CSP". Spaces + digits count too.

Concatenation: joins strings end to end → "sun" + "day" = "sunday" (no space added).

Substring: part of an existing string, e.g., "put" in "computer". Read the exam's rule for it.

"5" ≠ 5: digits in quotes are text; joining "5" and "5" gives "55", not the number 10.

Boolean Expressions

Boolean: only true or false. **Compare ops** (=, ≠, >, <, ≥, ≤) each give a true/false result.

A	B	A AND B	A OR B
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

AND: true only if both sides are true. **OR**: true if at least 1 side is true. **NOT** flips it.

Range check: (x ≥ 1) AND (x ≤ 10). With OR it's always true, so every x passes the test.

De Morgan: NOT (A AND B) = (NOT A) OR (NOT B). Flip the operator + negate each side.

Negating: NOT (x > 5) is x ≤ 5, not x < 5. The edge value 5 switches sides when you flip it.

⚠ x = 3 OR 4 is wrong. Write (x = 3) OR (x = 4): each side needs a full comparison.

✓ Test a Boolean answer w/ 1 value that should pass + 1 edge value that should fail.

3.6–3.7 Conditionals & Nested IFs

Selection

Selection: a true/false condition decides which code runs + which code is skipped.

IF (cond) { }: the block runs only if cond is true; if false, skip it + go to the next line.

IF / ELSE: exactly 1 of the 2 blocks runs, every time. ELSE covers every false case.

2 separate IFs can both run (each is tested); an IF / ELSE never runs both of its blocks.

Nested Conditionals

Nested: an IF inside another IF → the inner test runs only if the outer path is reached.

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

Order matters: test score ≥ 70 first → a 95 gets C. Check the strictest condition first.

Trace It

Method: plug in the value, mark each condition T or F, then follow only the true path.

Boundaries: test values right at the edge (89 + 90) b/c off-by-one errors hide there.

⚠ An IF that changes a variable can change which later IF runs. Trace line by line.

✓ Written response: name the condition, the input that makes it true, + what then runs.

3.8 Iteration (Loops)

Loop Types

Iteration: repeating a set of statements a set number of times or until a condition is met.

Loop	Runs
REPEAT n TIMES	exactly n times
REPEAT UNTIL	until cond is true
FOR EACH	1× per element

REPEAT UNTIL (cond): cond is checked **before** each pass. True at start → 0 passes.

Infinite loop: the stop condition never becomes true (e.g., the counter never changes).

Loop counter: i ← 1, then i ← i + 1 inside the loop. Forget the update → infinite loop.

Trace a Loop

Count passes: i ← 1, add 1 each pass: UNTIL (i > 5) → 5 passes; UNTIL (i ≥ 5) → 4.

Accumulator: sum ← 0 before the loop; sum ← sum + x inside. Never reset it inside.

Trace e.g.: x ← 1, REPEAT 3 TIMES {x ← x * 2} → 2, 4, 8. Final x = 8, not 6 (it doubles).

Robot Grid Loops

CAN_MOVE(dir): true if the square in that direction (from the robot's view) is open.

Blocked or off-grid move: robot stays put + the program ends. Rotate = turn 90° in place.

⚠ Track which way the robot faces after every turn. Left/right are the robot's, not yours.

✓ Loop questions: trace the 1st, 2nd + last passes, then check the stop condition.

3.10 Lists

List Operations (Ref Sheet)

Op	Effect	Length
APPEND(L, v)	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

Valid index: 1 to LENGTH(L). Outside that range → error message + the program stops.

INSERT/REMOVE shift later elements → their indices change. Recheck positions.

Traversals

Traversal: visit elements in order. **Complete** = every element; **partial** = only some.

FOR EACH item IN L: item takes each value, 1st to last. item is a value, not an index.

Standard List Algorithms

Sum/average: total ← 0, add each item; average ← total / LENGTH(L). **Count**: add 1 when IF is true.

Max: max ← L[1], then IF (item > max) {max ← item}. Starting max at 0 fails for negatives.

Filter: new ← [], then APPEND each item that passes a test; the original list is unchanged.

Linear search: check each element in order until found; works on unsorted lists.

⚠ No L[0] in AP pseudocode. Java/Python habits (start at 0) cause off-by-one errors.

✓ Written response: say what your list stores + how the code would be harder w/o it.

3.9 & 3.11 Algorithms & Binary Search

Building Blocks

Algorithm: a finite set of steps for a task, built from **sequencing**, **selection** + **iteration**.

Same goal, diff code: different algorithms can solve the same problem; similar ones can differ.

Reuse known algorithms (max, sum, MOD test) → less time to write, test + debug.

Compare 2 algorithms: run both on the same input, incl. edge cases (empty list, ties).

Binary Search

Binary search: data **must be sorted**. Check the middle; drop the half that can't hold it.

Each check halves what's left → about $\log_2(n)$ checks. Linear search: up to n checks.

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

Count checks: smallest k w/ $2^k \geq n + 1$. $2^{10} = 1{,}024$ → a sorted list of 1,000 needs ≤ 10.

Doubling n adds just 1 check to binary search but doubles the checks for linear search.

Trade-off: sorting takes time, so for 1 search of unsorted data, linear may be better.

⚠ Binary search on unsorted data can miss a target that is in the list. Sorted is required.

✓ Sorted + large list → binary search. Unsorted or tiny list → linear search (no sort needed).

3.12–3.14 Procedures & Libraries

Calling Procedures

Procedure: named group of steps w/ **parameters**; it may **RETURN** a value to the caller.

Parameter: the input variable in the definition.
Argument: the value sent in the call.

RETURN(expr): ends the procedure right away + sends the value back to the call.

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 by knowing what it does, not how it does it.

Modularity: split a program into procedures → easier to read, test, fix + reuse.

Generalize: parameters let 1 procedure handle many inputs, e.g., area(w, h) for any room.

Libraries & APIs

Library: prewritten procedures. **API**: the spec for how those procedures work + are used.

Documentation tells you each procedure's name, inputs + return value so you use it right.

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

✓ Written response: explain how your procedure manages complexity (1 call replaces repeats).

3.15–3.18 Random, Sims, Efficiency, Limits

Random Values & Simulations

RANDOM(a, b): random integer from a to b, **inclusive**; all equally likely. RANDOM(1, 6) = die.

Simulation: simplified model of a real process; uses random values for variability.

Pros: test ideas w/o real cost, danger or time.
Cons: simplified, so it may be biased.

Algorithmic Efficiency

Efficiency: how much time or memory an algorithm uses as the input size grows.

Growth	Example	Reasonable?
Linear (n)	linear search	Yes
Polynomial (n^2)	nested loops	Yes
Exponential (2^n)	try all subsets	No

Reasonable: polynomial or slower. **Unreasonable**: exponential → too slow for large n.

Heuristic: finds a good-enough answer when the exact one takes too long to find.

Undecidable Problems

Decidable: an algorithm gives the right yes/no answer for every input (is n even?).

Undecidable: no algorithm gives a correct answer for all inputs, e.g., the **halting problem**.

⚠ Undecidable ≠ slow. A faster computer can never make it solvable for every input.

✓ Efficiency questions: count how often the innermost line runs; nested loops over n → n^2.