WGU COMPUTER SCIENCE

DM1 (C959) & DSA1 (C949)

Holy Grail OA Readiness Study Guide

Your complete pass-the-OA reference for both courses
Study section-by-section with ChatGPT as your live tutor

PART 1	**Discrete Mathematics I — C959 (Sections 1–12)**
PART 2	**Data Structures & Algorithms I — C949 (Sections 1–9)**
BACK MATTER	**DM1 Checklist · DSA1 Checklist · OA Readiness Skills List**

Includes:

✓ Complete concept coverage — every OA topic for DM1 and DSA1
✓ Definition boxes · Worked examples · OA wording patterns
✓ Common trap warnings · ChatGPT tutor drill targets
✓ DM2/DSA2 continuation callouts · Complete coverage checklists

TABLE OF CONTENTS

PART 1 — DISCRETE MATHEMATICS I (C959)

DM1 · Section 1 — Propositional Logic & Logical Operators

📖 **DEFINITION**

Proposition: A declarative sentence with exactly one truth value — TRUE (T) or FALSE (F). Never both, never neither.

Compound proposition: Two or more propositions joined by logical operators.

Propositional variable: A letter (p, q, r…) representing a proposition.

1.1 Propositions vs Non-Propositions

IS a proposition (T or F)	Is NOT a proposition
'The sky is blue.' (TRUE)	'Is the sky blue?' (question — no truth value)
'2 + 2 = 5.' (FALSE)	'Sit down!' (command)
'It is raining.' (context-dependent, but T or F)	'x + 1 = 3' (open sentence — truth depends on x)
'Today is Monday.' (T or F)	'This sentence is false.' (self-referential paradox)

⚠️ **COMMON TRAPS**

Open sentences with variables are NOT propositions. "x > 5" is a predicate (Section 3). It only becomes a proposition when x is assigned a value.

1.2 The Six Logical Operators

Operator	Symbol	Read As	TRUE when	FALSE when
NOT (negation)	¬p or ~p	not p	p is FALSE	p is TRUE
AND (conjunction)	p ∧ q	p and q	BOTH TRUE	at least one FALSE
OR (disjunction)	p ∨ q	p or q (inclusive)	at least ONE TRUE	BOTH FALSE
XOR (exclusive or)	p ⊕ q	p exclusive-or q	EXACTLY ONE TRUE	both same value
Conditional (implication)	p → q	if p then q	always TRUE except p=T, q=F	ONLY when p=T and q=F
Biconditional (iff)	p ↔ q	p if and only if q	both SAME truth value	p and q DIFFER

⚠️ **COMMON TRAPS**

★ **THE CONDITIONAL (p → q) — Most-Tested Operator:**

Operator	Symbol	Read As	TRUE when	FALSE when

- FALSE in ONLY ONE case: p is TRUE and q is FALSE.
- If p is FALSE, the whole conditional is TRUE — "vacuous truth."
- "If pigs fly, I am president." → TRUE (false hypothesis = vacuously true).
- Memory: The conditional only makes a promise when p holds. A promise never made cannot be broken.

1.3 Truth Tables — Construction and Classification

Row count formula: For n distinct propositional variables → 2^n rows.

Standard order: for p, first half T / second half F. For q, alternate T/F every quarter. Etc.

p	q	¬p	p ∧ q	p ∨ q	p ⊕ q	p → q	p ↔ q
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

★ MUST KNOW COLD: Classify Any Expression by Its Final Column

Type	Final Column	Example
Tautology	ALL T — every row	p ∨ ¬p (always true — Law of Excluded Middle)
Contradiction	ALL F — every row	p ∧ ¬p (always false)
Contingency	Mixed T and F	p → q (depends on p and q values)

1.4 Logical Equivalences

Logical equivalence (≡): Two expressions are equivalent when their truth tables are IDENTICAL column-for-column.

★ MUST KNOW COLD: All Named Equivalence Laws

Law	AND form	OR form
Identity	p ∧ T ≡ p	p ∨ F ≡ p
Domination	p ∧ F ≡ F	p ∨ T ≡ T
Idempotent	p ∧ p ≡ p	p ∨ p ≡ p
Double Negation	¬(¬p) ≡ p	(applies to both)

Commutative	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
Associative	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive ★	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan's ★	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption	$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$
Negation/Complement	$p \wedge \neg p \equiv F$	$p \vee \neg p \equiv T$
Conditional ★	$p \rightarrow q \equiv \neg p \vee q$	(critical — memorise)
Contrapositive ★	$p \rightarrow q \equiv \neg q \rightarrow \neg p$	(equivalent to original)
Biconditional	$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$	$p \leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$

1.5 The Four Forms of a Conditional

★ MUST KNOW COLD: Converse / Inverse / Contrapositive — Which Is Equivalent?

Form	Written As	Equivalent to Original?
Original	$p \rightarrow q$	YES
Contrapositive	$\neg q \rightarrow \neg p$	✓ YES — always logically equivalent (negate BOTH, flip order)
Converse	$q \rightarrow p$	✗ NO — equivalent only to the Inverse
Inverse	$\neg p \rightarrow \neg q$	✗ NO — equivalent only to the Converse

🔍 **WORKED EXAMPLE**

Example: "If it rains (p), the ground is wet (q)."

Contrapositive: "If ground is NOT wet → it did NOT rain." ✓ equivalent

Converse: "If ground is wet → it rained." ✗ (sprinklers could wet ground)

Inverse: "If it does NOT rain → ground is NOT wet." ✗ (same reason)

1.6 CNF, DNF, Minterms, and Maxterms

Form	Full Name	Outer connective	Inner connective	Memory hook
DNF	Disjunctive Normal Form	OR (disjunction)	AND (conjunction)	D = Disjunction = OR is OUTSIDE

Form	Full Name	Outer connective	Inner connective	Memory hook
CNF	Conjunctive Normal Form	AND (conjunction)	OR (disjunction)	C = Conjunction = AND is OUTSIDE

DNF: **(p ∧ q) ∨ (¬p ∧ r)** ← ANDs inside, OR joins groups

CNF: **(p ∨ q) ∧ (¬p ∨ r)** ← ORs inside, AND joins groups

Term	Normal Form	Corresponds To
Minterm	DNF	One TRUE row in truth table — AND of all variables (negate those that are F in that row)
Maxterm	CNF	One FALSE row in truth table — OR of all variables (negate those that are T in that row)

🔍 WORKED EXAMPLE

Truth table → DNF: 1) Find rows where output = TRUE. 2) For each, AND all variables (¬ if F in that row). 3) OR all groups.

Truth table → CNF: 1) Find rows where output = FALSE. 2) For each, OR all variables (¬ if T in that row). 3) AND all groups.

If truth table has TRUE at p=T,q=T and p=F,q=T:

Minterms: (p ∧ q) and (¬p ∧ q) → DNF: (p ∧ q) ∨ (¬p ∧ q) simplifies to q

★ OA WORDING

- "Which is a proposition?" → look for a statement that is clearly T or F
- "Construct a truth table for…" → remember 2^n rows; work operator by operator left to right
- "Which expression is a tautology?" → final column must be ALL T
- "Apply De Morgan's to ¬(p ∨ q)" → ¬p ∧ ¬q (flip OR → AND, distribute NOT inside)
- "Which is logically equivalent to p → q?" → ¬p ∨ q OR ¬q → ¬p (either is correct)

⚠️ "The converse of p → q is…" → q → p. NOT the contrapositive. Converse ≠ equivalent to original.

⚠️ "If the hypothesis is false, the conditional is…" → TRUE (vacuously true). Most common wrong answer: FALSE.

- "Is p ↔ q a tautology?" → NO — it is a contingency
- "Which form is CNF?" → AND is outer connective, ORs inside each clause

⚠️ "A minterm corresponds to…" → DNF. A maxterm → CNF.

📌 FOUNDATION FOR DM2/DSA2

→ **DM2:** De Morgan's Laws, conditional equivalences, and CNF/DNF are used constantly in DM2 circuit design and formal proofs.

🎯 TUTOR DRILL TARGETS

→ State T/F for each operator given specific p,q values
→ Build a complete 2-variable truth table and classify as tautology/contradiction/contingency
→ Apply De Morgan's Law to any ¬(AND) or ¬(OR)
→ Name all four conditional forms; state which are equivalent
→ Convert a truth table to DNF and CNF
→ Identify CNF vs DNF from a given expression

DM1 · Section 2 — Set Theory & Set Operations

📖 **DEFINITION**

Set: A well-defined, unordered collection of DISTINCT objects (elements). Duplicates are ignored: {1,1,2} = {1,2}.

Roster notation: A = {1, 2, 3} | Set-builder: A = {x | P(x)} (all x such that P(x) is true)

Element notation: $x \in A$ (x is in A) | $x \notin A$ (x is not in A)

Key property: Sets ignore order AND duplicates: {1,2,3} = {3,2,1} = {1,1,2,3}

2.1 Special Sets — Symbols You Must Recognise

Symbol	Name	Members	Notes
$\mathbb{N}$	Natural numbers	0, 1, 2, 3, …	WGU typically includes 0
$\mathbb{Z}$	Integers	…, −2, −1, 0, 1, 2, …	Z from German Zahlen
$\mathbb{Q}$	Rationals	p/q where p,q ∈ $\mathbb{Z}$, q ≠ 0	includes all fractions
$\mathbb{R}$	Reals	all number-line values	includes √2, π, etc.
∅ or {}	Empty set	no elements	|∅| = 0. ∅ ≠ {∅}
U	Universal set	everything in the current context	defined per problem

⚠️ **COMMON TRAPS**

∅ **vs {∅}:** ∅ is empty (0 elements). {∅} contains the empty set (1 element). Different!

2.2 Subset, Proper Subset, Equality

Notation	Meaning	Example A={1,2}, B={1,2,3}
$A \subseteq B$	Every element of A is also in B (A can equal B)	$A \subseteq B$ ✓
$A \subset B$	$A \subseteq B$ AND $A \neq B$ (B has at least one extra)	$A \subset B$ ✓
A = B	$A \subseteq B$ AND $B \subseteq A$ (same elements)	A = B only if identical elements

★ **MUST KNOW COLD: Subset Count Formulas**

# of subsets of A = 2^n (n = |A|) e.g. |A|=3 → 8 subsets | |A|=4 → 16 subsets

# of proper subsets = $2^n - 1$ (excludes A itself)

∅ **is a subset of EVERY set.** Every set is a subset of itself.

2.3 Set Operations — Complete Reference

Operation	Notation	Definition	Example A={1,2,3}, B={2,3,4}
Union	$A \cup B$	Everything in A OR B (or both)	{1,2,3,4}
Intersection	$A \cap B$	Everything in BOTH A and B	{2,3}
Difference	$A - B$	In A but NOT in B	{1}
Difference	$B - A$	In B but NOT in A	{4}
Symmetric Difference	$A \oplus B$ or $A \triangle B$	In EXACTLY ONE (not both)	{1,4}
Complement	A^c or $\bar{A}$	All of U NOT in A	depends on U
Cartesian Product	$A \times B$	All ordered pairs (a,b), $a \in A$, $b \in B$	9 pairs total
Power Set	$\mathcal{P}(A)$ or 2^A	All subsets of A including $\varnothing$ and A	$\mathcal{P}(\{1,2\})=\{\varnothing,\{1\},\{2\},\{1,2\}\}$

2.4 Set Identities (Parallel to Logic Laws)

★ MUST KNOW COLD: De Morgan's for Sets — Most Tested

$(A \cup B)^c = A^c \cap B^c$ ← complement of union = intersection of complements

$(A \cap B)^c = A^c \cup B^c$ ← complement of intersection = union of complements

Distributive: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ | $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Absorption: $A \cup (A \cap B) = A$ | $A \cap (A \cup B) = A$

2.5 Cardinality & Inclusion-Exclusion

★ MUST KNOW COLD: Formulas

$|\mathcal{P}(A)| = 2^n$ where $n = |A|$

$|A \times B| = |A| \cdot |B|$

$|A \cup B| = |A| + |B| - |A \cap B|$ (2-set inclusion-exclusion)

$|A \cup B \cup C| = |A|+|B|+|C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$ (3-set)

🔍 WORKED EXAMPLE

Word problem: 200 CS students, 150 Math students, 50 in both. How many take at least one?

$|A \cup B| = 200 + 150 - 50 = 300$. "At least one" = UNION. "Both" = INTERSECTION.

★ OA WORDING

- "How many subsets does {a,b,c,d} have?" → $2^4 = 16$
- "Find A − B" → elements in A that are NOT in B

⚠️ "Find B − A" vs "Find A − B" — DIFFERENT answers. Order matters for set difference.

- "Symmetric difference A ⊕ B" / "elements in exactly one set" → $(A-B) \cup (B-A) = \{1,4\}$

⚠️ "Is $\varnothing \subseteq A$?" → YES for every set A, including $A = \varnothing$.

- "Apply De Morgan's to $(A \cap B)^c$" → $A^c \cup B^c$
- "200 like X, 150 like Y, 50 like both — how many like at least one?" → $200+150-50 = 300$

⚠️ "At least one" → UNION | "Both" → INTERSECTION | "Exactly one" → symmetric difference.

📌 FOUNDATION FOR DM2/DSA2

→ **DM2:** Inclusion-exclusion expands into advanced counting. Set operations parallel probability rules directly.

🎯 TUTOR DRILL TARGETS

→ List all subsets of a 3-element set
→ Compute union, intersection, difference, symmetric difference for two given sets
→ Apply inclusion-exclusion to a 2-set and 3-set word problem
→ Apply De Morgan's Laws for sets
→ Compute $|\mathcal{P}(A)|$ for sizes 0–4
→ State and identify subset vs proper subset vs equal

DM1 · Section 3 — Predicate Logic & Quantifiers

📖 **DEFINITION**

Predicate P(x): A function that becomes a proposition when x is assigned a value. P(x)="x is even." P(4)=TRUE. P(3)=FALSE. By itself P(x) is not a proposition.

Domain (Universe of Discourse): The set of all possible values x can take. Must be specified — truth can change with domain.

3.1 The Two Quantifiers

Quantifier	Symbol	Read As	TRUE when	FALSE when
Universal	∀x P(x)	For ALL x, P(x)	P(x) true for EVERY x in domain	P(x) false for at least ONE x
Existential	∃x P(x)	There EXISTS at least one x where P(x)	P(x) true for at least ONE x	P(x) false for ALL x

3.2 Negating Quantifiers — The Flip Rule

★ **MUST KNOW COLD: Quantifier Negation — Two Rules to Memorise**

¬(∀x P(x)) ≡ ∃x ¬P(x) "Not for all" = "Some don't"

¬(∃x P(x)) ≡ ∀x ¬P(x) "None exist" = "All fail it"

Memory: NOT pushes THROUGH the quantifier → quantifier FLIPS (∀↔∃) AND predicate gets negated.

Nested — push NOT through one at a time:

¬(∀x ∃y P(x,y)) → ∃x ¬(∃y P(x,y)) → ∃x ∀y ¬P(x,y)

🔍 **WORKED EXAMPLE**

Original	Negation	Plain English
∀x (x>0)	∃x (x≤0)	Some number is not positive
∃x (x is prime)	∀x ¬(x is prime)	No number is prime
∀x ∃y (x+y=0)	∃x ∀y (x+y≠0)	Some x has no additive inverse
All students pass	∃x (Student(x) ∧ ¬Pass(x))	Some student does not pass

3.3 Free vs Bound Variables

Term	Definition	In ∀x P(x,y)
Bound variable	Governed by a quantifier inside the expression	x is BOUND (the ∀ governs it)
Free variable	No governing quantifier — expression is an open sentence	y is FREE

Key: An expression with a free variable is a predicate, not a proposition. Only becomes a proposition when free variables are bound or given values.

3.4 English → Predicate Logic Translation

★ MUST KNOW COLD: Translation Rules

English Phrase	Symbol/Pattern
Everyone / All / Every / For each / For any	∀x
Someone / Some / There exists / At least one	∃x
No one / Nobody / None	∀x ¬P(x) or ¬∃x P(x)
Not everyone / Not all	∃x ¬P(x)
Exactly one (unique existential)	∃!x P(x)
All students pass (∀ + →)	∀x (Student(x) → Pass(x))
Some student passes (∃ + ∧)	∃x (Student(x) ∧ Pass(x))
No student fails	∀x (Student(x) → ¬Fail(x))

⚠ COMMON TRAPS

★ **Critical pattern:** Use → with ∀ and ∧ with ∃.

WRONG: ∀x (Student(x) ∧ Pass(x)) — says EVERYTHING is both a student and passing.

WRONG: ∃x (Student(x) → Pass(x)) — an always-true conditional satisfies this vacuously.

3.5 Nested Quantifiers — Order Matters

Statement	Meaning	Truth (domain = ℤ, P(x,y) = "x+y=0")
∀x ∀y P(x,y)	P true for ALL pairs	FALSE (0+1≠0)
∀x ∃y P(x,y)	For EACH x, find some y	TRUE (y=−x always works)
∃x ∀y P(x,y)	ONE x works for ALL y	FALSE (no single x)

Statement	Meaning	Truth (domain = ℤ, P(x,y) = "x+y=0")
∃ x ∃ y P(x,y)	At least ONE pair works	TRUE (x=1,y=−1)

★ OA WORDING

- "Translate: Every student passed" → ∀ x (Student(x) → Pass(x))
- "Translate: Some cat is black" → ∃ x (Cat(x) ∧ Black(x))
- "Negate: ∀ x P(x)" → ∃ x ¬P(x)
- "Negate: ∃ x P(x)" → ∀ x ¬P(x)

⚠️ "Negate: All primes are odd" → ∃ x(Prime(x) ∧ ¬Odd(x)) i.e. "some prime is not odd." TRUE — 2 is prime and even.

- "Negate: ∀ x ∃ y (x+y=0)" → ∃ x ∀ y (x+y≠0)

⚠️ "Is y free or bound in ∀ x P(x,y)?" → y is FREE. Only x has a quantifier.

⚠️ "Not all dogs bark" ≠ "No dogs bark." "Not all" = some don't. "None" = all don't.

🎯 TUTOR DRILL TARGETS

→ Translate English to predicate logic (use ∀ / ∃ with correct connective)
→ Negate single-quantifier statements
→ Negate nested quantifier statements step by step
→ Identify free vs bound variables in an expression
→ Distinguish ∀ x ∃ y from ∃ x ∀ y with a concrete example

DM1 · Section 4 — Rules of Inference & Proof Techniques

📖 **DEFINITION**

Valid argument: IF all premises are true, the conclusion MUST be true. The structure guarantees truth transfer.

Fallacy: Looks like a valid argument but is not — conclusion does not necessarily follow from premises.

4.1 The Eight Rules of Inference

★ **MUST KNOW COLD: All Eight Rules**

Rule Name	Premises	Conclusion	Memory
Modus Ponens ★	$p \to q$ and p	q	If condition met → result follows
Modus Tollens ★	$p \to q$ and $\neg q$	$\neg p$	Result missing → condition was not met
Hypothetical Syllogism	$p \to q$ and $q \to r$	$p \to r$	Chain two conditionals
Disjunctive Syllogism	$p \vee q$ and $\neg p$	q	One OR option false → other must hold
Addition	p	$p \vee q$	True statement can be OR-ed with anything
Simplification	$p \wedge q$	p (or q)	Extract either half of an AND
Conjunction	p and q	$p \wedge q$	Join two truths with AND
Resolution	$p \vee q$ and $\neg p \vee r$	$q \vee r$	Cancel complementary pair in two ORs

4.2 The Two Fallacies

⚠ **COMMON TRAPS**

Fallacy 1 — Affirming the Consequent (INVALID):

Form: $p \to q$ and $q \vdash p$ ← WRONG

"If it rains, grass is wet. Grass is wet. Therefore it rained." → INVALID (sprinklers!)

Fallacy 2 — Denying the Antecedent (INVALID):

Form: $p \to q$ and $\neg p \vdash \neg q$ ← WRONG

"If it rains, grass is wet. It is not raining. Therefore grass is not wet." → INVALID (sprinklers!)

4.3 Proof Techniques

Technique	Goal	Structure	Best When
Direct Proof	Prove p→q	Assume p. Derive q step by step using definitions/algebra.	Straightforward — try first
Proof by Contrapositive	Prove p→q	Prove ¬q→¬p instead (logically equivalent).	¬q gives more to work with than p
Proof by Contradiction	Prove p	Assume ¬p. Derive a logical impossibility. Conclude p must be true.	Proving irrationality, impossibility, uniqueness
Proof by Cases	Prove p→q	Split p into exhaustive cases; prove q in each.	p has multiple forms (odd/even, etc.)
Mathematical Induction ★	Prove ∀n P(n)	Base case + inductive step. See Section 5.	Any universal integer statement

🔍 WORKED EXAMPLE

Direct Proof — "If n is even, then n^2 is even":

Assume n is even → $n = 2k$ for some $k \in \mathbb{Z}$.

Then $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$. Since $2k^2 \in \mathbb{Z}$, n^2 is even. ✓

Proof by Contradiction — "$\sqrt{2}$ is irrational":

Assume $\sqrt{2} = p/q$ in lowest terms (gcd(p,q)=1).

$2 = p^2/q^2 \rightarrow p^2 = 2q^2 \rightarrow p^2$ even → p even → $p = 2m$.

$(2m)^2 = 2q^2 \rightarrow 4m^2 = 2q^2 \rightarrow q^2 = 2m^2 \rightarrow$ q even.

But p and q both even contradicts gcd(p,q)=1. ✗ Therefore $\sqrt{2}$ is irrational. ✓

★ OA WORDING

- "Which rule? If she studies she passes. She studies. Therefore she passes." → Modus Ponens
- "Which rule? If rain→cancelled. Not cancelled. Therefore no rain." → Modus Tollens
- "Which rule? She is in class OR outside. Not in class. Therefore outside." → Disjunctive Syllogism
- "p is true. Therefore p ∨ q." → Addition

⚠️ "Affirming the consequent" is a FALLACY — not Modus Ponens. The consequent being true does not prove the antecedent.

- "Which technique proves $\sqrt{2}$ irrational?" → Proof by Contradiction

⚠️ "Proof by contrapositive" proves ¬q→¬p. It is NOT the same as contradiction. Know the difference.

🎯 TUTOR DRILL TARGETS

→ Name the rule of inference given a valid argument in any form
→ Identify affirming the consequent or denying the antecedent as a fallacy

→ Choose the correct proof technique for a given statement
→ Trace through a direct proof for an even/odd result
→ Trace through the $\sqrt{2}$ contradiction proof steps

DM1 · Section 5 — Mathematical Induction

📖 **DEFINITION**

Mathematical induction: A proof technique for proving P(n) is true for ALL integers $n \geq n_0$.

Domino analogy: If the first domino falls (base case), and each domino knocks the next (inductive step), ALL dominos fall.

Well-Ordering Principle: Every non-empty set of positive integers has a smallest element. This is what guarantees induction works.

5.1 Standard (Weak) Induction — The Template

★ **MUST KNOW COLD: Two-Step Induction Structure**

STEP 1 — Base Case: Prove $P(n_0)$ is true. Substitute the starting value (usually 1) into both sides and verify.

STEP 2 — Inductive Step:

- **Inductive Hypothesis:** Assume P(k) is true for some arbitrary $k \geq n_0$.
- **Goal:** Prove P(k+1) is true USING the inductive hypothesis.
- **Key move:** Express P(k+1) in terms of P(k) — typically: (sum to k+1) = (sum to k) + (k+1)th term.

🔍 **WORKED EXAMPLE**

Classic: Prove $1+2+3+\ldots+n = n(n+1)/2$

Base (n=1): LHS = 1. RHS = 1(2)/2 = 1. ✓

Inductive Hypothesis: Assume $1+2+\ldots+k = k(k+1)/2$.

Prove for k+1:

$1+2+\ldots+k+(k+1) = k(k+1)/2 + (k+1) = (k+1)(k/2+1) = (k+1)(k+2)/2$ ✓

Divisibility: Prove $3 \mid (n^3-n)$ for all $n \geq 1$

Base (n=1): $1-1=0$, and $3 \mid 0$. ✓

IH: Assume $3 \mid (k^3-k)$. Prove $3 \mid ((k+1)^3-(k+1))$.

Expand: $(k+1)^3-(k+1) = k^3+3k^2+3k+1-k-1 = (k^3-k)+3k^2+3k = (k^3-k)+3k(k+1)$.

(k^3-k) divisible by 3 (IH). $3k(k+1)$ obviously divisible by 3. Sum divisible by 3. ✓

5.2 Strong Induction

Use when: P(k+1) requires knowing P(j) for multiple $j < k$, not just P(k).

Structure: Inductive hypothesis becomes: "Assume P(1), P(2), …, P(k) are ALL true." Then prove P(k+1).

Classic example: Every integer $n > 1$ has a prime factorisation. P(k+1) may need to factor k+1 using knowledge about all numbers smaller than k+1.

★ OA WORDING

- "What are the two steps of induction?" → Base case AND inductive step
- "What do you assume in the inductive step?" → P(k) is true (the inductive hypothesis)
- "What do you prove in the inductive step?" → P(k+1) is true
- "Base case for 1+2+…+n = n(n+1)/2?" → Substitute n=1: LHS=1, RHS=1(2)/2=1. ✓

⚠️ "Strong induction is needed when…" → P(k+1) depends on multiple earlier values, not just P(k).

- "What principle guarantees induction works?" → Well-Ordering Principle

⚠️ "What is the inductive hypothesis?" → "Assume P(k) is true." Not "assume P is true" — it must be for a specific k.

📌 FOUNDATION FOR DM2/DSA2

→ **DM2 HEAVILY:** Induction is the backbone of DM2. Used to prove graph properties, recurrence relations, and combinatorial identities.

🎯 TUTOR DRILL TARGETS

→ State both steps of induction from memory
→ Write the base case for a given summation formula
→ Write the inductive step for 1+2+…+n
→ Explain when strong induction is needed vs weak induction
→ State the Well-Ordering Principle and its role

DM1 · Section 6 — Functions

📖 **DEFINITION**

Function f: A → B: A rule assigning to each element of domain A EXACTLY ONE element in codomain B.

Domain A: The set of allowable inputs.

Codomain B: The set the outputs come from (the "target" set).

Range (image): The set of ACTUAL outputs produced. Range ⊆ Codomain (may be strictly smaller).

Functions vs Relations: Every function is a relation, but a relation can map one input to multiple outputs (not a function).

6.1 Function Classification — Injective, Surjective, Bijective

★ **MUST KNOW COLD: Classification Rules**

Type	Also Called	Definition	Test	f^{-1} Exists?
Injective	One-to-one	f(a)=f(b) implies a=b (distinct inputs → distinct outputs)	No two x values share an output (no y hit twice)	Only if also surjective
Surjective	Onto	Every element of B is the output of at least one input	Range = Codomain exactly (no orphan outputs)	Only if also injective
Bijective	One-to-one correspondence	Both injective AND surjective	Each output hit exactly once; nothing in B missed	YES — inverse always exists

🔍 **WORKED EXAMPLE**

Function (domain/codomain = ℝ)	Injective?	Surjective?	Bijective?
$f(x) = x^3$	✓ YES (strictly increasing)	✓ YES (all reals hit)	✓ YES
$f(x) = x^2$	✗ NO (f(2)=f(−2)=4)	✗ NO (no negative outputs)	✗ NO
f(x) = 2x + 1	✓ YES	✓ YES	✓ YES → $f^{-1}(x) = (x-1)/2$

f: $\mathbb{Z}\to\mathbb{Z}$, f(n)=2n	✓ YES	✗ NO (odd integers never hit)	✗ NO

6.2 Inverse Functions

Inverse f^{-1} exists if and only if f is bijective.

To find $f^{-1}(x)$: Write y = f(x). Swap x and y. Solve for y. That y is $f^{-1}(x)$.

Example: f(x) = 3x − 2. Swap: x = 3y − 2. Solve: y = (x+2)/3. So $f^{-1}(x) = (x+2)/3$.

6.3 Function Composition

(g ∘ f)(x) = g(f(x)): Apply f FIRST, then g. Read right to left: "g of f of x."

Order matters: g ∘ f ≠ f ∘ g in general.

🔍 WORKED EXAMPLE

f(x) = x + 1, g(x) = x^2

(g ∘ f)(3) = g(f(3)) = g(4) = 16

(f ∘ g)(3) = f(g(3)) = f(9) = 10 ← different answer

6.4 Floor and Ceiling Functions

Function	Symbol	Definition	Positive Example	Negative Trap ★
Floor	⌊x⌋	Largest integer ≤ x	⌊3.7⌋ = 3	⌊−3.7⌋ = −4 (NOT −3)
Ceiling	⌈x⌉	Smallest integer ≥ x	⌈3.2⌉ = 4	⌈−3.2⌉ = −3 (NOT −4)

⚠️ COMMON TRAPS

Negative floor trap: ⌊−3.7⌋ = −4, not −3. Floor always rounds TOWARD −∞. Ceiling rounds TOWARD +∞. This is frequently tested.

★ OA WORDING

- "Is $f(x)=x^3$ injective?" → YES — strictly monotone, no two inputs share an output
- "Is $f(x)=x^2$ surjective on $\mathbb{R}\to\mathbb{R}$?" → NO — negatives never appear in range
- ⚠️ "Does f^{-1} exist?" → ONLY if f is bijective. Both injective AND surjective required.
- "Find f^{-1} for f(x)=3x−2" → y=3x−2, swap: x=3y−2, solve: y=(x+2)/3
- "Compute (g∘f)(x)" → apply f FIRST, then g (right to left)
- ⚠️ "(g∘f)(x) vs (f∘g)(x)" — DIFFERENT. Read composition order carefully. ∘ reads right-to-left.
- "⌊−2.3⌋ = ?" → −3 (NOT −2)
- "⌈4.0⌉ = ?" → 4 (already integer; ceiling of integer = itself)

🎯 TUTOR DRILL TARGETS

→ Classify a function as injective / surjective / bijective from its definition or a small table
→ Find the inverse of a linear function
→ Compute f∘g and g∘f for two functions and explain why they differ
→ Evaluate floor and ceiling for positive and negative non-integers
→ Identify the domain, codomain, and range of a described function

DM1 · Section 7 — Relations

📖 **DEFINITION**

Relation R on a set A: A set of ordered pairs (a, b) where a, b ∈ A. Written (a,b) ∈ R or a R b.

Relation from A to B: A subset of A × B. More general than a function — one input may relate to many outputs.

Function vs Relation: A function is a relation where each input has EXACTLY ONE output. Relations allow multiple outputs per input.

7.1 The Four Properties — Test from Ordered Pairs

★ **MUST KNOW COLD: Property Definitions and Tests**

Property	Formal Definition	How to Test	Fails If
Reflexive	(a,a) ∈ R for ALL a ∈ A	Every element must self-relate	ANY element missing its (a,a) pair
Symmetric	(a,b) ∈ R → (b,a) ∈ R	Every pair must have its reverse	Any pair (a,b) present without (b,a)
Antisymmetric	(a,b) ∈ R AND (b,a) ∈ R → a = b	No two DISTINCT elements can relate in BOTH directions	Both (a,b) and (b,a) exist with a ≠ b
Transitive	(a,b) ∈ R AND (b,c) ∈ R → (a,c) ∈ R	Every chain a→b→c must have shortcut a→c	Any chain (a→b→c) without (a,c)

★ **Symmetric ≠ Antisymmetric are NOT opposites.** A relation with ONLY self-pairs (a,a) is BOTH symmetric and antisymmetric.

🔍 **WORKED EXAMPLE**

A = {1,2,3}, R = {(1,1),(1,2),(2,1),(2,2),(3,3)}

Property	Evidence	Result
Reflexive	(1,1)✓ (2,2)✓ (3,3)✓ — all present	YES

Symmetric	(1,2) present AND (2,1) present ✓	YES
Antisymmetric	Both (1,2) and (2,1) exist, but 1≠2	NO
Transitive	(1,2)+(2,1)⟶(1,1)✓ (2,1)+(1,2)⟶(2,2)✓ all chains OK	YES
Type?	Reflexive + Symmetric + Transitive	EQUIVALENCE RELATION

7.2 Special Relation Types

Type	Properties Required	Classic Example
Equivalence Relation	Reflexive + Symmetric + Transitive	= (equality), "same birthday as", modular congruence
Partial Order	Reflexive + Antisymmetric + Transitive	≤ on integers, ⊆ on sets
Strict Order	Irreflexive + Antisymmetric + Transitive	< on integers

7.3 Equivalence Classes and Partitions

Equivalence class [a]: The set of ALL elements related to a under an equivalence relation R.

[a] = {b ∈ A | (a,b) ∈ R}

Partition: An equivalence relation on A divides A into non-overlapping equivalence classes that together cover all of A. Each class is one "block" of the partition.

🔍 **WORKED EXAMPLE**

Relation on ℤ: "a ≡ b (mod 3)" (a and b have the same remainder when divided by 3)

Equivalence classes: [0] = {…,−3,0,3,6,…} [1] = {…,−2,1,4,7,…} [2] = {…,−1,2,5,8,…}

These three classes partition ℤ. Every integer belongs to exactly one class.

★ **OA WORDING**

- "Is R={(1,1),(2,2),(3,3)} an equivalence relation on {1,2,3}?" ⟶ YES (all self-pairs satisfy all three)

⚠️ "Can a relation be both symmetric AND antisymmetric?" ⟶ YES — if all pairs are self-pairs (a,a).

- "The relation ≤ on ℤ is a…" ⟶ Partial order (reflexive, antisymmetric, transitive)
- "Is 'is a sibling of' an equivalence relation?" ⟶ Not reflexive (you are not your own sibling)

⚠️ "Antisymmetric does NOT mean 'not symmetric.'" ⟶ They can co-exist. Check the definition precisely.

- "Which property requires: if (a,b) and (b,c) then (a,c)?" ⟶ Transitive
- "Equivalence classes partition A" ⟶ YES — they are disjoint and their union = A

📌 **FOUNDATION FOR DM2/DSA2**

→ **DM2:** Equivalence relations and partial orders are core to DM2 combinatorics. Relations as ordered pairs directly connect to graph edges.

🎯 TUTOR DRILL TARGETS

→ Given a list of ordered pairs, test all four properties
→ Classify a relation as equivalence / partial order / neither
→ Find equivalence classes for a modular arithmetic relation
→ State the definition of equivalence class and partition

DM1 · Section 8 — Boolean Algebra

📖 **DEFINITION**

Boolean algebra: Algebra of values {0, 1} (or {FALSE, TRUE}). Same laws as propositional logic, different notation.

Notation mapping: Logic $p \wedge q$ ↔ Boolean xy (product) | $p \vee q$ ↔ x+y (sum) | $\neg p$ ↔ x' or $\bar{x}$

8.1 Boolean Laws — Complete Reference

★ **MUST KNOW COLD: All Boolean Identities**

Law	Product (AND) form	Sum (OR) form
Identity	x · 1 = x	x + 0 = x
Domination	x · 0 = 0	x + 1 = 1
Idempotent	x · x = x	x + x = x
Complement	x · x' = 0	x + x' = 1
Double Complement	(x')'= x	(same)
Commutative	xy = yx	x+y = y+x
Associative	x(yz) = (xy)z	x+(y+z) = (x+y)+z
Distributive ★	x(y+z) = xy+xz	x+yz = (x+y)(x+z)
De Morgan's ★	(xy)' = x'+y'	(x+y)' = x'y'
Absorption ★	x(x+y) = x	x+xy = x

8.2 Boolean Simplification

The OA gives you an expression and asks you to simplify it. Work through these steps in order:

- Look for x·x' = 0 or x+x' = 1 — these collapse terms immediately
- Apply absorption: x+xy = x (the xy disappears)
- Apply De Morgan's to break up complements of products or sums
- Distribute or factor to create collapsible patterns
- Substitute back and verify

🔍 **WORKED EXAMPLE**

Simplify: x + x'y

= (x + x')(x + y) [distributive: x + yz = (x+y)(x+z)]

= 1 · (x + y) [complement: x + x' = 1]

= x + y ✓

Simplify: (xy)' using De Morgan's

(xy)' = x' + y' [flip · to +, complement each variable]

Simplify: (x+y)' using De Morgan's

(x+y)' = x'y' [flip + to ·, complement each variable]

★ OA WORDING

- "Simplify x · x'" → 0 (complement product)
- "Simplify x + x'y" → x + y
- "Simplify (xy)'" → x' + y' (De Morgan's)
- "Simplify (x+y)'" → x'y' (De Morgan's)

⚠ De Morgan's in Boolean: flip the operation AND complement each variable. (xy)'→x'+y' | (x+y)'→x'y'

- "Simplify x + xy" → x (absorption — the xy is absorbed into x)

⚠ "Which law gives x(x+y) = x?" → ABSORPTION. The answer is just x.

- "Which expression equals 1 for all inputs?" → x + x' (complement law)
- "AND gate = ?" → · (product) "OR gate = ?" → + (sum) "NOT gate = ?" → x' (complement)

🎯 TUTOR DRILL TARGETS

→ Apply De Morgan's to any (xy)' or (x+y)'
→ Simplify a 2–3 variable Boolean expression step by step
→ Identify which law justifies each simplification step
→ Map logic operator notation to Boolean notation and back

DM1 · Section 9 — Matrices

📖 **DEFINITION**

Matrix: A rectangular array of numbers with m rows and n columns. Always described as m × n (ROWS × COLUMNS, in that order).

Entry notation: a_{ij} = entry in row i, column j. Rows go across (horizontal). Columns go down (vertical).

9.1 Matrix Operations

Operation	Requirement	How to Compute
Addition / Subtraction	SAME dimensions	Add/subtract corresponding entries: (A+B)_{ij} = a_{ij} + b_{ij}
Scalar Multiplication	Any size	Multiply every entry by the scalar k: (kA)_{ij} = k·a_{ij}
Matrix Multiplication AB	Cols(A) = Rows(B)	Entry (i,j) of AB = dot product of row i of A with column j of B
Transpose A^T	Any size	Rows become columns: (A^T)_{ij} = A_{ji}

🔍 **WORKED EXAMPLE**

Matrix multiplication — 2×2 example:

A = [[1,2],[3,4]], B = [[5,6],[7,8]]

AB[1][1] = 1·5 + 2·7 = 5 + 14 = 19

AB[1][2] = 1·6 + 2·8 = 6 + 16 = 22

AB[2][1] = 3·5 + 4·7 = 15 + 28 = 43

AB[2][2] = 3·6 + 4·8 = 18 + 32 = 50

Result: [[19,22],[43,50]]

9.2 Row Operations & Gaussian Elimination

★ **MUST KNOW COLD: The Three Legal Row Operations**

#	Operation	Notation	Example
R1	Row Swap — interchange two rows	$R_i \leftrightarrow R$□	Swap Row 1 and Row 2
R2	Row Scaling — multiply a row by any non-zero constant	$cR_i \longrightarrow R_i$	(1/3)R2 ⟶ R2

R3	Row Replacement — replace row with itself + multiple of another	$R_i + cR_j \longrightarrow R_i$	$R3 - 2R1 \longrightarrow R3$

Row Echelon Form (REF): Staircase pattern of leading entries going left-to-right; zeros BELOW each leading entry.

Reduced REF (RREF): Leading entry in each row = 1; the ONLY non-zero in its entire column.

Gaussian Elimination: Apply row operations to reduce an augmented matrix to REF or RREF to solve a linear system.

9.3 Adjacency Matrices for Graphs

Entry (i,j) = 1 if there is an edge from vertex i to vertex j; otherwise 0.

Undirected graphs: Adjacency matrix is always SYMMETRIC ($A = A^T$) because edge {i,j} = edge {j,i}.

Directed graphs: May NOT be symmetric.

★ OA WORDING

- "Dimensions of AB if A is 3×4 and B is 4×2?" → 3×2 (outer dimensions: rows of A × cols of B)

⚠️ "Can you multiply AB if A is 3×4 and B is 3×2?" → NO. Inner dimensions must match ($4 \neq 3$).

- "Find entry (2,3) of AB" → dot product of row 2 of A with column 3 of B

⚠️ "R3 → R3 − 2R1" means: New Row3 = (old Row3) MINUS (2 × Row1). Not the other way.

- "A matrix in RREF has…" → leading 1s in staircase, each leading 1 is the only non-zero in its column
- "Adjacency matrix of undirected graph is always…" → SYMMETRIC

🎯 TUTOR DRILL TARGETS

→ Multiply two 2×2 matrices
→ Identify which row operation was applied given before/after rows
→ State whether matrix multiplication AB is defined given dimensions
→ Write the adjacency matrix for a small undirected graph
→ State the difference between REF and RREF

DM1 · Section 10 — Sequences & Summations

10.1 Sequence Types

★ MUST KNOW COLD: Recognition Rules

Type	Pattern	nth Term Formula	Sum Formula	Test
Arithmetic	Constant difference d between terms	$a_n = a_1 + (n-1)d$	$S_n = n(a_1+a_n)/2$	Subtract consecutive terms — constant?
Geometric	Constant ratio r between terms	$a_n = a_1 \cdot r^{n-1}$	$S_n = a_1(1-r^n)/(1-r)$ $r \neq 1$	Divide consecutive terms — constant?
Fibonacci / Recursive	Each term = sum of previous two	$F(n) = F(n-1) + F(n-2)$	No simple closed form	1,1,2,3,5,8,13,21,34 …

⚠ COMMON TRAPS

Trap: 2,4,8,16 — differences are 2,4,8 (NOT constant → not arithmetic) but ratios are 2,2,2 (constant → GEOMETRIC r=2). Always check both before classifying.

10.2 Summation Notation and Closed Forms

$\sum_{i=1}^{n} a_i$ means $a_1 + a_2 + \ldots + a_n$. The index i runs from the bottom value to the top.

★ MUST KNOW COLD: Four Closed-Form Sums — All On the OA

Formula	Result	Verify at n=4
$\sum_{i=1}^{n} i$	$n(n+1)/2$	$4(5)/2 = 10$ ✓ (1+2+3+4=10)
$\sum_{i=1}^{n} i^2$	$n(n+1)(2n+1)/6$	$4(5)(9)/6 = 30$ ✓ (1+4+9+16=30)
$\sum_{i=1}^{n} i^3$	$[n(n+1)/2]^2$	$[4(5)/2]^2 = 100$ ✓ (1+8+27+64=100)
$\sum_{i=0}^{n} r^i$ $(r \neq 1)$	$(r^{n+1} - 1)/(r - 1)$	finite geometric series — use this formula directly

★ OA WORDING

- "Next term of 3,7,11,15,…?" → arithmetic d=4, next=19
- "Next term of 2,6,18,54,…?" → geometric r=3, next=162
- "10th term of 5,10,20,40,…?" → a_1=5, r=2: $a_{10} = 5 \cdot 2^9 = 2560$

⚠️ "Evaluate $\Sigma_{i=1}^{10}$ i" → use formula: 10(11)/2 = 55. Do NOT add by hand.

- "Evaluate $\Sigma_{i=1}^{5}$ i^2" → 5(6)(11)/6 = 55

⚠️ "A sequence has differences 2,4,8,16" → NOT arithmetic. Check RATIOS of terms instead.

🎯 TUTOR DRILL TARGETS

→ Classify a sequence as arithmetic / geometric / recursive
→ Find the nth term using the arithmetic or geometric formula
→ Apply all four closed-form summation formulas
→ Evaluate a finite geometric series using the sum formula

DM1 · Section 11 — Graph Theory

📖 **DEFINITION**

Graph G = (V, E): V = set of vertices (nodes). E = set of edges (connections between pairs of vertices).

Undirected: Edge {u,v} = {v,u} — no direction. Directed (digraph): edge (u,v) ≠ (v,u).

Simple graph: No self-loops (vertex connected to itself) and no multiple edges between the same pair.

11.1 Graph Terminology

★ **MUST KNOW COLD: Core Terms**

Term	Definition
Degree of v	Number of edges incident to v. Notation: deg(v). In a digraph: in-degree and out-degree.
Handshaking Lemma ★	$\Sigma \deg(v) = 2\|E\|$ — sum of ALL degrees = twice the number of edges. Sum of degrees is always EVEN.
Path	Sequence of vertices $v_0, v_1, \cdots, v_k$ where each consecutive pair is connected by an edge.
Simple path	Path with no repeated vertices.
Cycle	A path that starts and ends at the same vertex ($v_0 = v_k$) with no other repeated vertices.
Circuit	A closed walk (starts = ends) where edges may not repeat.
Connected graph	A path exists between EVERY pair of vertices.
Complete graph K_n	Every vertex connected to every other. Edges = $n(n-1)/2$.
Bipartite graph	Vertices split into two disjoint sets; edges only go BETWEEN sets, never within.
Subgraph	A graph formed from a subset of V and a subset of E of a larger graph.

11.2 Adjacency Representations

Representation	What It Is	Advantage	Disadvantage
Adjacency Matrix	n×n matrix; entry (i,j)=1 if edge exists	O(1) edge lookup	$O(n^2)$ space — bad for sparse graphs
Adjacency List	Each vertex stores a list of its neighbours	Space-efficient for sparse graphs	O(degree) edge lookup

11.3 Euler and Hamiltonian Paths/Circuits

★ **MUST KNOW COLD: Existence Conditions — Memorise Both**

Type	Definition	Existence Condition	Memory Hook
Euler PATH	Uses every EDGE exactly once	Connected + EXACTLY 2 odd-degree vertices	E = Edges
Euler CIRCUIT	Every EDGE once, returns to start	Connected + ALL vertices have EVEN degree	E = Edges, closed
Hamiltonian PATH	Visits every VERTEX exactly once	No simple general condition (NP-complete)	H = vertices (Ham = Ham-ilton)
Hamiltonian CIRCUIT	Every VERTEX once, returns to start	No simple general condition	H = vertices, closed

⚠ **COMMON TRAPS**

Euler = EDGES. Hamiltonian = VERTICES. Euler has clean existence conditions. Hamiltonian does NOT.

11.4 Graph Isomorphism

Two graphs are isomorphic if there is a one-to-one mapping between their vertices that preserves all edges. They are structurally identical, just drawn differently.

★ **MUST KNOW COLD: Quick Isomorphism Elimination — Check These First**

- Same number of vertices? If NO → not isomorphic.
- Same number of edges? If NO → not isomorphic.
- Same degree sequence (sorted list of all degrees)? If NO → not isomorphic.
- Same number of connected components? If NO → not isomorphic.

If all checks pass: try to find a vertex mapping that preserves every edge (necessary but tedious).

11.5 Minimum Spanning Trees

Algorithm	Strategy	Big-O
Kruskal's	Sort ALL edges by weight. Greedily add cheapest edge that does not form a cycle.	O(E log E)
Prim's	Start at one vertex. Always extend by the cheapest edge to an unvisited vertex.	O(E log V)

Both algorithms produce a spanning tree with exactly n−1 edges and minimum total weight.

★ **OA WORDING**

- "Edges in K_5?" → $5(5-1)/2 = 10$
- "6 vertices each degree 3. How many edges?" → $\Sigma\deg = 18 = 2|E|$, so $|E| = 9$

⚠️ "Sum of degrees is always odd" → FALSE. Handshaking lemma: sum = $2|E|$, ALWAYS EVEN.

- "Does Euler circuit exist?" → check: connected AND all vertices have EVEN degree
- "Does Euler PATH (not circuit) exist?" → connected AND EXACTLY TWO odd-degree vertices

⚠️ "Euler circuit vs Hamiltonian circuit" — Euler visits every EDGE once. Hamiltonian visits every VERTEX once. Completely different.

- "Are these two graphs isomorphic?" → check degree sequences FIRST. Mismatch → NO immediately.
- "Which algorithm adds the globally cheapest edge that doesn't form a cycle?" → Kruskal's
- "A tree with n vertices has how many edges?" → $n-1$, always

📌 FOUNDATION FOR DM2/DSA2

→ **DM2 HEAVILY:** DM2 extends graph theory to planar graphs, graph colouring, chromatic numbers, and Euler's formula $V-E+F=2$.

🎯 TUTOR DRILL TARGETS

→ State degree of each vertex in a small drawn graph
→ Apply handshaking lemma: given degrees find edge count or vice versa
→ Determine whether an Euler path/circuit exists
→ Check isomorphism using degree sequence
→ Trace Kruskal's and Prim's on a small weighted graph

DM1 · Section 12 — Trees

📖 **DEFINITION**

Tree: A connected, undirected graph with NO cycles. Always has exactly n−1 edges for n vertices.

Rooted tree: A tree with one designated vertex as the ROOT. Creates parent/child structure.

Binary tree: Each node has at most 2 children (left and right).

12.1 Tree Terminology

Term	Definition
Root	The designated top-most node; has no parent
Leaf (external node)	A node with no children (degree 1 in rooted tree)
Internal node	A node that has at least one child
Parent / Child	Direct above / direct below relationship in rooted tree
Sibling	Nodes sharing the same parent
Ancestor	Any node on the path from root to the current node
Descendant	Any node reachable by going down from the current node
Height of tree	Length of LONGEST path from root to any leaf (count edges)
Depth / Level of node v	Number of edges from root to v (root is depth 0)
Full binary tree	Every internal node has EXACTLY 2 children
Complete binary tree	All levels fully filled left-to-right (heap shape)

12.2 Spanning Trees and Minimum Spanning Trees

Spanning tree of G: A subgraph that includes ALL vertices, is connected, and has no cycles (i.e., is a tree). Always has exactly n−1 edges.

MST: A spanning tree with minimum total edge weight. Found by Kruskal's or Prim's (see Section 11.5).

★ **OA WORDING**

- "A tree with 15 vertices has how many edges?" → 14 (always n−1)

⚠️ "A connected graph with n vertices and n−1 edges is always a tree" → TRUE. This is the definition.

- "Height of the tree?" → count EDGES on the LONGEST root-to-leaf path
- "Depth of vertex v?" → count edges from root to v

⚠️ "Full binary tree vs complete binary tree" — FULL: every internal node has exactly 2 children. COMPLETE: all levels filled left-to-right. These are DIFFERENT definitions.

- "A spanning tree includes which vertices?" → ALL vertices of the original graph

🎯 **TUTOR DRILL TARGETS**

→ Label root, leaves, internal nodes, parent, child, sibling, ancestor, descendant
→ Compute height and depth of nodes in a drawn tree
→ State the number of edges in a tree with n vertices
→ Distinguish full vs complete binary tree
→ State what a spanning tree is and its edge count

DM1 / C959 — Complete Coverage Checklist

Check each box before attempting the OA. If you cannot answer or explain a topic, go back to that section.

□	Topic	□	Topic
□	Propositions vs non-propositions	□	Conditional equivalence $p \to q \equiv \neg p \vee q$
□	All 6 logical operators (NOT,AND,OR,XOR,→,↔)	□	Converse / Inverse / Contrapositive
□	Truth tables — 2^n rows	□	CNF and DNF — identify and convert
□	Tautology / contradiction / contingency	□	Minterms (DNF) and Maxterms (CNF)
□	Logical equivalences — all named laws	□	Boolean algebra laws (10 identities)
□	De Morgan's laws (logic AND sets AND Boolean)	□	Boolean simplification steps
□	Predicates and domain	□	Matrix dimensions / addition / multiplication
□	Universal ∀ and existential ∃ quantifiers	□	Transpose and row operations (R1,R2,R3)
□	Negating quantifiers (both flip rules)	□	REF vs RREF
□	Nested quantifiers — order matters	□	Arithmetic and geometric sequences
□	Free vs bound variables	□	Fibonacci/recursive sequences
□	Translation English ↔ predicate logic	□	Summation notation and 4 closed forms (Σi, Σi^2, Σi^3, geometric)
□	All 8 rules of inference	□	Graph terminology (degree, path, cycle, connected, K_n, bipartite)
□	Fallacies (affirming consequent, denying antecedent)	□	Handshaking lemma
□	Proof techniques (direct, contrapositive, contradiction, cases, induction)	□	Adjacency matrix and adjacency list
□	Mathematical induction — base case + inductive step	□	Euler path/circuit conditions
□	Strong induction	□	Hamiltonian path/circuit
□	Well-ordering principle	□	Graph isomorphism — degree sequence check
□	Set notation and element notation	□	Minimum spanning trees — Kruskal's and Prim's
□	Special sets ($\mathbb{N},\mathbb{Z},\mathbb{Q},\mathbb{R},\varnothing$,U)	□	Tree terminology (root,leaf,height,depth,parent,child,sibling)
□	Subsets / proper subsets / power set	□	Full vs complete binary tree
□	Set operations ($\cup, \cap, -, \oplus, A^c, \times, \mathscr{P}$)	□	Spanning tree definition and edge count

□	Topic	□	Topic
□	Set identities and De Morgan's for sets	□	Equivalence relation and equivalence classes
□	Inclusion-exclusion (2-set and 3-set)	□	Partial order
□	Functions: domain / codomain / range	□	Reflexive / Symmetric / Antisymmetric / Transitive
□	Injective / surjective / bijective	□	Floor and ceiling functions (including negative values)
□	Inverse functions	□	Function composition

PART 2 — DATA STRUCTURES & ALGORITHMS I (C949)

DSA1 · Section 1 — Algorithms & Complexity Analysis

📖 **DEFINITION**

Algorithm: A finite, step-by-step procedure for solving a problem. Must have five properties:

- **Input:** Zero or more values are provided as input.
- **Output:** At least one value is produced.
- **Finiteness:** The algorithm terminates after a finite number of steps.
- **Definiteness:** Every step is precisely and unambiguously defined.
- **Effectiveness:** Every step is simple enough to be carried out by a person with pen and paper.

1.1 Time and Space Complexity

Term	Definition
Time complexity	How the number of operations grows as input size n increases
Space complexity	How the amount of memory used grows as input size n increases
Best-case	Complexity for the most favourable input (e.g. target is first element in linear search)
Worst-case	Complexity for the least favourable input (e.g. target not in list) ← OA usually tests this
Average-case	Expected complexity averaged over all possible inputs

1.2 Big-O, Big-Ω, Big-Θ

★ **MUST KNOW COLD: Three Asymptotic Notations**

Notation	Read As	Meaning	Use
O(g(n))	Big-O of g(n)	UPPER bound — algorithm runs in AT MOST this rate of growth	Worst-case ceiling ★ most tested
Ω(g(n))	Big-Omega of g(n)	LOWER bound — algorithm runs in AT LEAST this rate of growth	Best-case floor
Θ(g(n))	Big-Theta of g(n)	TIGHT bound — algorithm is EXACTLY this growth rate (O and Ω match)	When upper = lower bound

Practical rule: The OA primarily uses Big-O to describe worst-case complexity. Know what all three mean if asked to distinguish them.

1.3 Big-O Hierarchy — Fastest to Slowest

★ **MUST KNOW COLD: Growth Rate Order**

Big-O	Name	n=1,000 ops	Example
O(1)	Constant	1	Array index lookup, hash table average case
O(log n)	Logarithmic	~10	Binary search
O(n)	Linear	1,000	Linear search, single for-loop over n
O(n log n)	Linearithmic	~10,000	Merge sort, heap sort
$O(n^2)$	Quadratic	1,000,000	Nested loops, bubble/selection/insertion sort worst
$O(n^3)$	Cubic	10^9	Naive matrix multiplication
$O(2^n)$	Exponential	$\sim 10^{301}$	Naive recursive Fibonacci
O(n!)	Factorial	intractable	Brute force permutations

1.4 Simplifying Big-O

- Drop constants: $O(5n) \rightarrow O(n)$ | $O(100) \rightarrow O(1)$
- Drop lower-order terms: $O(n^2 + n + 100) \rightarrow O(n^2)$ | $O(2^n + n^{100}) \rightarrow O(2^n)$
- Keep only the DOMINANT term as $n \rightarrow \infty$

1.5 Reading Big-O from Code Patterns

Code Pattern	Big-O	Why
Single statement, return, or array index	O(1)	No dependency on n
One for-loop from 0 to n	O(n)	Runs n times
Two SEQUENTIAL for-loops over n (not nested)	O(n)	$n + n = 2n \rightarrow O(n)$
Two NESTED for-loops over n	$O(n^2)$	$n \times n$ iterations
Loop that halves the input each iteration	O(log n)	Binary-style reduction
Outer loop O(n) + inner loop O(log n)	O(n log n)	$n \times \log n$ iterations
Three nested loops over n	$O(n^3)$	$n \times n \times n$
Recursive call with 2 sub-calls each time	$O(2^n)$	Naive Fibonacci pattern
Different inputs a and b, sequential	O(a + b)	Two independent sizes

Code Pattern	Big-O	Why
Different inputs a and b, nested	O(a · b)	Two independent sizes, nested

🔍 **WORKED EXAMPLE**

Sequential vs Nested:

```
for i in range(n):   # O(n)
    pass
for j in range(n):   # O(n) — SEQUENTIAL, not nested
    pass
Total: O(n) + O(n) = O(n)   ← NOT O(n²)

for i in range(n):   # outer O(n)
    for j in range(n):   # inner O(n) — NESTED
        pass
Total: O(n) × O(n) = O(n²)
```

1.6 Algorithm Complexity Full Reference Table

Algorithm / Operation	Best	Average	Worst	Space
Array/list index access	O(1)	O(1)	O(1)	—
Linear search	O(1)	O(n)	O(n)	O(1)
Binary search	O(1)	O(log n)	O(log n)	O(1)
Hash table search/insert/delete	O(1)	O(1)	O(n)	O(n)
BST search (balanced)	O(log n)	O(log n)	O(log n)	O(n)
BST search (unbalanced)	O(1)	O(log n)	O(n)	O(n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Heap insert	O(log n)	O(log n)	O(log n)	—
Heap extract min/max	O(log n)	O(log n)	O(log n)	—

★ OA WORDING

- "Simplify $O(4n^3 + 100n^2 + 5000)$" → $O(n^3)$
- "Two for-loops NESTED, each over n" → $O(n^2)$

⚠️ "Two for-loops SEQUENTIAL, each over n" → O(n). NOT $O(n^2)$. Sequential = add. Nested = multiply.

- "Big-O of Python dict lookup?" → O(1) average

⚠️ "Worst case of hash table lookup?" → O(n) — when all keys collide into the same bucket.

- "Which is faster for large n: $O(n^2)$ or O(n log n)?" → O(n log n) grows slower = faster

⚠️ "Best case of selection sort?" → $O(n^2)$. It ALWAYS does n^2 comparisons regardless of input — no early exit.

- "Best case of bubble sort?" → O(n) — already sorted, one pass detects no swaps needed

🎯 TUTOR DRILL TARGETS

→ State the 5 algorithm properties from memory
→ Rank the 8 Big-O orders from fastest to slowest
→ Simplify a given Big-O expression by dropping constants and lower-order terms
→ Determine Big-O for a code snippet with sequential and/or nested loops
→ Distinguish O / Ω / Θ and state which one the OA usually asks about
→ Look up any algorithm in the complexity reference table

DSA1 · Section 2 — ADTs, OOP, and Python Basics

📖 **DEFINITION**

Abstract Data Type (ADT): Defines WHAT operations a data structure supports (the interface) WITHOUT specifying HOW it is implemented.

Data Structure: The concrete implementation — how data is actually stored and operations are carried out.

Example: Stack ADT defines push/pop/peek. It can be implemented with an array OR a linked list — two different data structures, one ADT.

2.1 OOP Vocabulary

★ **MUST KNOW COLD: Object-Oriented Terms — All Tested on OA**

Term	Definition	Python Example
Class	Blueprint/template that defines attributes and methods	class Node:
Object / Instance	A specific instance created from a class at runtime	n = Node(5) ← n is an object/instance
Attribute	A variable belonging to an object	self.value, self.next
Method	A function belonging to a class, operates on the object	def insert(self, x):
Constructor	Special method that runs automatically when an object is created	__init__(self, ...) in Python
Encapsulation	Bundling data and methods together; hiding internal details	Why data structures use classes
Inheritance	A child class derives attributes and methods from a parent class	class BST(Tree):
Polymorphism	Same method name behaves differently in different classes	Multiple sort() implementations
Abstraction	Exposing only necessary interface; hiding implementation details	Stack hides whether it uses array or linked list

2.2 ADT Comparison Table

ADT	Ordered?	Duplicates ?	Key Behaviour	Typical Implementation
Bag (Multiset)	NO	YES	Collect items; no position or indexing	Array or linked list
List	YES	YES	Position-indexed; access by index	Array or linked list
Stack	YES (LIFO)	YES	Push/pop from TOP only	Array or linked list
Queue	YES (FIFO)	YES	Enqueue at back, dequeue from front	Array (circular) or linked list
Hash Table	NO	Keys: NO	Key→value pairs; O(1) average lookup	Array of buckets + hash function
Tree	YES (hierarchical)	YES	Parent-child structure; BST has ordering	Nodes with pointers

2.3 Python Basics — What You Must Read Fluently

★ MUST KNOW COLD: Python Data Types

Type	Syntax	Ordered?	Mutable?	Duplicates?	Key Use
list	[1, 2, 3]	YES	YES	YES	Default ordered sequence
tuple	(1, 2, 3)	YES	NO	YES	Fixed/immutable records
dict	{"a":1}	YES (3.7+)	YES	Keys: NO	Key→value, O(1) lookup
set	{1, 2, 3}	NO	YES	NO	Uniqueness, fast membership O(1)

★ MUST KNOW COLD: Python Syntax to Read Fluently

Variables: x = 5 s = "hello" items = [1,2,3]

Conditionals: if x > 0: elif x == 0: else:

for loop: for i in range(n): ← i runs 0,1,2,…,n−1 (starts at 0, stops BEFORE n)

for with enumerate: for i, x in enumerate(lst): ← gives index AND value

while loop: while condition:

Function: def f(param1, param2=default): return result

Class: class Node: def __init__(self, val): self.val = val; self.next = None

List comprehension: [x*x for x in range(10)] → [0,1,4,9,16,25,36,49,64,81]

2.4 Python Operation Big-O

Operation	Code	Big-O	Note
List index access	lst[i]	O(1)	Direct memory address
List append (end)	lst.append(x)	O(1) amortized	Occasional resize is O(n) but rare
List insert at i	lst.insert(i, x)	O(n)	Shifts all elements after i
List pop from end	lst.pop()	O(1)	
List pop from front	lst.pop(0)	O(n)	Use collections.deque for O(1)
List search (in)	x in lst	O(n)	Linear scan
Dict lookup	d[key]	O(1) avg	Hash table backing
Set membership	x in s	O(1) avg	Hash table backing
Sort	sorted(lst)	O(n log n)	Timsort
deque appendleft/popleft	dq.appendleft(x)/popleft()	O(1)	Best for queue use
heapq push/pop	heapq.heappush/heappop(h,x)	O(log n)	Min-heap

★ OA WORDING

- "ADT vs data structure?" → ADT = interface (what operations). Data structure = implementation (how).
- "The __init__ method is the…?" → constructor (runs when object is created)

⚠️ "Which OOP concept bundles data and methods?" → ENCAPSULATION — not abstraction, not inheritance.

- "list vs tuple?" → list is mutable (can change). tuple is immutable (cannot change).
- "set vs list for 'x in …'?" → set is O(1). list is O(n). Big difference at scale.
- "range(5)?" → 0,1,2,3,4 (NOT 1,2,3,4,5)

⚠️ "lst.pop(0) vs deque.popleft()" → list is O(n), deque is O(1). Use deque for queue operations.

🎯 TUTOR DRILL TARGETS

→ State the difference between an ADT and a data structure with an example
→ Define all 9 OOP terms from the table
→ Classify list/tuple/dict/set by ordered/mutable/duplicates
→ Read a Python for-loop and state what range(n) produces
→ State Big-O for the key Python list operations

DSA1 · Section 3 — Arrays and Linked Lists

3.1 Arrays / Python Lists

📖 **DEFINITION**

Static array: Fixed size allocated at creation. All elements same type. Random access O(1).

Dynamic array (Python list): Resizes automatically. Internally doubles capacity when full. Append is O(1) amortized.

Operation	Big-O	Why
Access by index arr[i]	O(1)	Direct memory calculation: base_address + i × element_size
Search (unsorted)	O(n)	Must scan every element in worst case
Append (end)	O(1) amortized	Rarely triggers resize; resize is O(n) but rare
Insert at position i	O(n)	Must shift elements at i and after to the right
Delete from end	O(1)	pop()
Delete from position i	O(n)	Must shift elements left to fill gap

3.2 Linked Lists

📖 **DEFINITION**

Node: A container holding a data value and a pointer (reference) to the next node.

Head: The first node in the list. Entry point.

Tail: The last node. Its next pointer = None/null.

Singly linked: Each node has ONE pointer: next. Can only traverse forward.

Doubly linked: Each node has TWO pointers: next and prev. Can traverse both directions.

Operation	Singly Linked	Doubly Linked	Note
Access by index	O(n)	O(n)	Must traverse from head — no direct address
Search for value	O(n)	O(n)	Must scan entire list
Insert at head	O(1)	O(1)	Update head pointer
Insert at tail	O(n) without tail ptr / O(1) with tail ptr	O(1) with tail ptr	Depends on whether tail is tracked
Delete at head	O(1)	O(1)	Update head

Delete a specific node (given reference)	O(1)	O(1)	Update surrounding pointers
Delete a specific node (must find it first)	O(n)	O(n)	Must traverse to locate

3.3 Array vs Linked List Tradeoffs

	Array (Python list)	Linked List	Winner
Random access (index)	O(1)	O(n)	Array
Insert/delete at front	O(n)	O(1)	Linked List
Insert/delete at end	O(1) amortized	O(1) with tail	Tie
Memory	Contiguous block	Scattered; extra pointer memory	Array (less overhead)
Cache performance	Excellent (locality)	Poor (scattered)	Array
Dynamic size	Doubles capacity	Grows/shrinks per node	Linked List (no wasted space)

🔍 WORKED EXAMPLE

Singly linked list insert at head:

new_node = Node(value)

new_node.next = head ← new node points to old head

head = new_node ← head now points to new node

Time: O(1) — no traversal needed

★ OA WORDING

- "Big-O of accessing index 500 in a 1000-element list?" → O(1)
- "Big-O of inserting at position 0 in a Python list?" → O(n) (must shift all elements)
- "Insert at head of linked list?" → O(1) (update pointer, no traversal)

⚠️ "Linked list is better for index access" → FALSE. Linked list access is O(n). Array access is O(1).

- "What does a node contain?" → a data value AND a pointer to the next node
- "Difference between singly and doubly linked?" → doubly has PREV pointer; singly has only NEXT

⚠️ "Can you access the middle of a linked list in O(1)?" → NO. Must traverse from head: O(n).

🎯 TUTOR DRILL TARGETS

→ State Big-O for all array operations (access, search, insert at i, append, delete)
→ State Big-O for linked list insert at head vs tail vs middle
→ Describe the node structure of a singly vs doubly linked list
→ Explain when to prefer array over linked list and vice versa
→ Trace a linked list insert-at-head in pseudocode steps

DSA1 · Section 4 — Stacks and Queues

4.1 Stack (LIFO — Last In, First Out)

📖 **DEFINITION**

Stack: An ordered collection where elements are added and removed from the SAME end (the top).

LIFO: Last element added is the FIRST one removed.

★ **MUST KNOW COLD: Stack Operations**

Operation	What It Does	Big-O
push(x)	Add x to the TOP of the stack	O(1)
pop()	REMOVE and RETURN the top element	O(1)
peek() / top()	VIEW the top element WITHOUT removing it	O(1)
isEmpty()	Return True if stack has no elements	O(1)

🔍 **WORKED EXAMPLE**

Trace — push(1), push(2), push(3), pop(), peek():

After push(1): stack = [1] (1 is top)

After push(2): stack = [1, 2] (2 is top)

After push(3): stack = [1, 2, 3] (3 is top)

pop() → returns 3, stack = [1, 2] (last in, first out)

peek() → returns 2, stack UNCHANGED = [1, 2] (just views, does not remove)

Real-world use case	Why a stack works
Function call stack (recursion)	Each function call pushes a frame; return pops it — LIFO
Undo / Redo	Last action is undone first — LIFO
Expression parsing (brackets)	Open brackets pushed; close bracket checks matching pop
DFS traversal (non-recursive)	Uses a stack explicitly instead of recursion

4.2 Queue (FIFO — First In, First Out)

📖 **DEFINITION**

Queue: An ordered collection where elements are added at the BACK and removed from the FRONT.

FIFO: First element added is the FIRST one removed.

★ MUST KNOW COLD: Queue Operations

Operation	What It Does	Big-O
enqueue(x)	Add x to the BACK/REAR	O(1)
dequeue()	REMOVE and RETURN the FRONT element	O(1)
peek() / front()	VIEW the front element WITHOUT removing	O(1)
isEmpty()	Return True if queue has no elements	O(1)

🔍 WORKED EXAMPLE

Trace — enqueue(A), enqueue(B), enqueue(C), dequeue(), peek():

After enqueue(A): queue = [A] (A at front)

After enqueue(B): queue = [A, B]

After enqueue(C): queue = [A, B, C]

dequeue() → returns A, queue = [B, C] (first in, first out)

peek() → returns B, queue UNCHANGED = [B, C]

Python — use collections.deque for O(1) enqueue AND dequeue:

```
from collections import deque
q = deque()
q.append(x)    # enqueue — O(1)
q.popleft()    # dequeue — O(1)   ← list.pop(0) would be O(n)
```

Real-world use case	Why a queue works
Print queue / task scheduling	First job submitted is first to be printed — FIFO
BFS (breadth-first search)	Process nodes level by level — FIFO order
CPU process scheduling	Round-robin: processes wait in line and are served in arrival order

★ OA WORDING

- "Stack: push(5), push(3), pop() — what is returned?" → 3 (last in, first out)
- "Queue: enqueue(X), enqueue(Y), dequeue() — what is returned?" → X (first in, first out)

⚠️ "peek() vs pop()" — peek VIEWS without removing. pop REMOVES and returns. Stack unchanged after peek.

⚠️ "LIFO = Stack, FIFO = Queue" — if a question says 'last added is first removed' → Stack. 'First added is first removed' → Queue.

- "Why use collections.deque for a queue?" → list.pop(0) is O(n). deque.popleft() is O(1).
- "Which data structure is used in DFS?" → Stack (or recursion)
- "Which data structure is used in BFS?" → Queue

🎯 TUTOR DRILL TARGETS

→ Trace a push/pop sequence for a stack and state what is returned
→ Trace an enqueue/dequeue sequence for a queue
→ Distinguish peek from pop
→ State the real-world use cases for stacks and queues
→ State why deque is preferred over list for queue operations in Python

DSA1 · Section 5 — Hash Tables

📖 **DEFINITION**

Hash table: A data structure that maps KEYS to VALUES for average O(1) lookup, insert, and delete.

Hash function: A function h(k) that converts a key k into a bucket index (an integer 0…m−1).

Common hash function: h(k) = k mod m where m = number of buckets.

Bucket (slot): One storage location in the hash table — may hold one item or a chain of items.

5.1 Collisions and How to Resolve Them

Collision: Two different keys produce the SAME bucket index. Must be handled.

Strategy	How It Works	Advantage	Disadvantage
Separate Chaining	Each bucket holds a linked list of all keys that hash there	Simple; handles many collisions gracefully	Extra memory for pointers; poor cache performance
Open Addressing (Linear Probing)	On collision, try next slot: (h(k)+1) mod m, then +2, etc.	Better cache performance (contiguous memory)	Clustering — keys bunch together, degrading performance
Open Addressing (Quadratic Probing)	On collision, try h(k)+1², h(k)+2², h(k)+3², …	Reduces primary clustering	Secondary clustering possible

5.2 Load Factor

Load factor α = n / m where n = number of keys stored, m = number of buckets.

Rule of thumb: Keep α < 0.75 for good performance. When α gets too high, the table is resized (rehashed).

Effect of high load factor: More collisions → longer chains (chaining) or more probing steps (open addressing) → degrades toward O(n).

5.3 Hash Table Complexity

Operation	Average Case	Worst Case	Worst Case Cause
Search / Lookup	O(1)	O(n)	All keys in same bucket (all collide)
Insert	O(1)	O(n)	Same reason
Delete	O(1)	O(n)	Same reason

🔍 **WORKED EXAMPLE**

Hash table with 10 buckets. Keys: 37, 47, 12, 57. h(k) = k mod 10

h(37) = 7 → bucket 7

h(47) = 7 → COLLISION with 37!

Operation	Average Case	Worst Case	Worst Case Cause
Separate chaining: bucket 7 holds linked list: [37] → [47] h(12) = 2 → bucket 2 h(57) = 7 → bucket 7 chain: [37] → [47] → [57]			

★ OA WORDING

- "Where does key 37 go in hash table with 10 buckets?" → h(37) = 37 mod 10 = bucket 7
- "Two different keys hash to the same bucket — what is this?" → a collision
- "When does hash table degrade to O(n)?" → when all keys collide into one bucket
- "What is load factor?" → n/m (number of keys / number of buckets)

⚠️ "Separate chaining vs open addressing" — chaining uses linked lists per bucket; open addressing probes for next empty slot within the same array.

- "Average case for hash table search?" → O(1)

⚠️ "Worst case for hash table search?" → O(n) — common wrong answer is O(1).

🎯 TUTOR DRILL TARGETS

→ Apply h(k) = k mod m to place keys in buckets
→ Trace separate chaining when a collision occurs
→ Define load factor and state the formula
→ Explain why worst-case is O(n)
→ Distinguish average vs worst case for hash table operations

DSA1 · Section 6 — Trees, BSTs, Traversals & Heaps

📖 DEFINITION

Binary tree: Each node has at most 2 children (left, right).

Binary Search Tree (BST): A binary tree where for EVERY node:

- All values in the LEFT subtree are LESS THAN the node's value.
- All values in the RIGHT subtree are GREATER THAN the node's value.
- This applies RECURSIVELY to every node in the tree.

6.1 BST Operations

Operation	How It Works	Balanced Big-O	Unbalanced Big-O
Search	Compare target to node; go left if smaller, right if larger, until found or null	O(log n)	O(n)
Insert	Search for correct position; insert as new leaf	O(log n)	O(n)

Operation	How It Works	Balanced Big-O	Unbalanced Big-O
Delete	Three cases: no children, one child, two children (replace with in-order successor)	O(log n)	O(n)
Inorder Traversal	Visits nodes in SORTED ascending order	O(n)	O(n)

⚠ **COMMON TRAPS**

Balanced vs Unbalanced BST:

Balanced (height ≈ log n): all operations O(log n).

Unbalanced (e.g. inserting sorted input 1,2,3,4,5 → degenerates to a linked list): operations become O(n).

🔍 **WORKED EXAMPLE**

BST Insert — insert 5, 3, 7, 1, 4 (in that order):

Insert 5: root = 5

Insert 3: 3 < 5 → go left. Left of 5 = 3

Insert 7: 7 > 5 → go right. Right of 5 = 7

Insert 1: 1 < 5 → left; 1 < 3 → left. Left of 3 = 1

Insert 4: 4 < 5 → left; 4 > 3 → right. Right of 3 = 4

6.2 The Four Tree Traversal Orders

★ **MUST KNOW COLD: Traversals — Visit Order**

Traversal	Order of Visits	Memory Hook	Key Use	Uses Queue?
Preorder	ROOT → Left → Right	PRE = root BEFORE children	Copy a tree; prefix expressions; DFS	No (uses recursion/stack)
Inorder	Left → ROOT → Right	IN = root IN the middle	SORTED output from BST ★	No
Postorder	Left → Right → ROOT	POST = root AFTER children	Delete a tree; postfix expressions	No
Level-order (BFS)	Level by level, left to right	Breadth-first — widest first	Shortest path; print by levels	YES — uses a QUEUE ★

★ Inorder traversal of a BST always produces values in SORTED (ascending) order.

★ Level-order traversal uses a QUEUE. All others use recursion (or an explicit stack).

🔍 **WORKED EXAMPLE**

BST: root=8, left=3(children:1,6), right=10(right=14(left=13))

Inorder: 1, 3, 6, 8, 10, 13, 14 ← SORTED

Preorder: 8, 3, 1, 6, 10, 14, 13 ← root first

Postorder: 1, 6, 3, 13, 14, 10, 8 ← root last

Level-order: 8, 3, 10, 1, 6, 14, 13 ← row by row

6.3 Heaps and Priority Queues

📖 **DEFINITION**

Heap: A COMPLETE binary tree satisfying the heap property.

Min-heap: Every parent ≤ its children. Root = MINIMUM element always.

Max-heap: Every parent ≥ its children. Root = MAXIMUM element always.

Complete binary tree: All levels fully filled left-to-right. Required for heap structure.

Priority Queue ADT: An ADT where the highest-priority element is always at the front. Almost always implemented with a min-heap.

★ **MUST KNOW COLD: Heap Operations**

Operation	Big-O	How
Access min/max (peek at root)	O(1)	Root is always min/max — direct access
Insert	O(log n)	Add at end of tree; sift UP (swap with parent while violating heap property)
Delete min/max (extract root)	O(log n)	Replace root with last element; sift DOWN (swap with smaller child for min-heap)
Build heap from n elements	O(n)	NOT O(n log n) — this surprises people; bottom-up build is linear

Array representation (1-indexed): Parent of i: ⌊i/2⌋ | Left child: 2i | Right child: 2i+1

⚠️ **COMMON TRAPS**

Min-heap ≠ BST:

- Heap only guarantees parent ≤ children (no left-vs-right ordering).
- BST guarantees left subtree < node < right subtree (full ordering).
- A heap is NOT a BST. Do not confuse them.

★ OA WORDING

- "Inorder traversal of a BST produces…" → values in SORTED ascending order
- "Preorder: first element is always…" → the ROOT
- "Postorder: last element is always…" → the ROOT

⚠️ "Level-order uses which data structure?" → QUEUE (not stack, not recursion).

- "Insert 5 into BST: root=8, left=3, right=10" → 5 < 8 → go left; 5 > 3 → go right; right child of 3 = 5

⚠️ "In a min-heap, root is the…" → MINIMUM element. In a max-heap, root is the MAXIMUM.

- "Build heap from n elements is O(…)" → O(n) (NOT O(n log n))

⚠️ "Is every min-heap also a BST?" → NO. Heap has no left/right ordering rule. Completely different.

🎯 TUTOR DRILL TARGETS

→ Trace BST insert for a sequence of 4–5 values
→ Write all 4 traversal sequences for a given small BST
→ State which traversal gives sorted output and why
→ State which traversal uses a queue
→ Identify min-heap vs max-heap property violation
→ State Big-O for heap insert and extract
→ State the array index formulas for parent and children of node i

DSA1 · Section 7 — Searching Algorithms

Algorithm	Requirement	Best	Average	Worst	How It Works
Linear Search	None — works on any collection	O(1)	O(n)	O(n)	Check each element one by one from start until found or exhausted
Binary Search	SORTED array or list — REQUIRED	O(1)	O(log n)	O(log n)	Compare target to middle; if less go left half, if greater go right half; repeat until found or range empty

🔍 WORKED EXAMPLE

Binary search trace — find 7 in [1,3,5,7,9,11,13]:

low=0, high=6, mid=3. arr[3]=7. Found! ⟶ 7 is at index 3

Binary search trace — find 6 in [1,3,5,7,9]:

low=0, high=4, mid=2. arr[2]=5. 6>5 ⟶ low=3

low=3, high=4, mid=3. arr[3]=7. 6<7 ⟶ high=2

low=3 > high=2 ⟶ NOT FOUND

Binary search pseudocode:

```
low = 0,  high = n−1
while low <= high:
    mid = (low + high) // 2
    if arr[mid] == target: return mid
    elif arr[mid] < target: low = mid + 1
    else: high = mid − 1
return −1  (not found)
```

⚠️ COMMON TRAPS

Binary search REQUIRES a sorted array. Applying binary search to an unsorted array produces WRONG results — it does not "still work but slower."

★ OA WORDING

- "Binary search on sorted array of 1024 elements — max comparisons?" ⟶ $\log_2(1024) = 10$
- "Which search works on unsorted data?" ⟶ Linear search only

⚠️ "Binary search on unsorted array" ⟶ WRONG result, not just slower. The algorithm assumes sorted order.

- "Identify this search from pseudocode: checks middle, halves range" ⟶ Binary search
- "Identify this search: checks each element one by one" ⟶ Linear search

🎯 TUTOR DRILL TARGETS

→ Trace binary search step by step on a small sorted array
→ State the requirement for binary search
→ State best/average/worst Big-O for both search algorithms
→ Identify binary vs linear search from a code snippet

DSA1 · Section 8 — Sorting Algorithms

8.1 Sorting Vocabulary

★ MUST KNOW COLD: Key Terms

Term	Definition	Which Sorts
Stable	Preserves the relative order of elements with EQUAL keys	Bubble ✓, Insertion ✓, Merge ✓ | Selection ✗, Quick ✗, Heap ✗
In-place	Sorts using O(1) extra memory (does not create a separate array)	All EXCEPT Merge sort (Merge needs O(n) extra)
Comparison-based	Sorts by comparing pairs of elements	All six listed here — theoretical lower bound O(n log n)
Internal sorting	All data fits in RAM	Standard assumption for all six sorts below
External sorting	Data too large for RAM; uses disk	Merge sort variants are used for external sorting

8.2 All Six Sorting Algorithms — Complete Reference

Algorithm	Best	Average	Worst	Space	Stable?	In-Place?	Identifying Behaviour
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	✓ YES	✓ YES	Compares and swaps ADJACENT elements repeatedly; large elements "bubble" to end each pass
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	✗ NO	✓ YES	Finds the MINIMUM in the unsorted portion each pass and moves it to sorted position
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	✓ YES	✓ YES	Inserts each element into its correct position in an already-sorted left portion
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	✓ YES	✗ NO	Divide-and-conquer: split in half recursively, then merge sorted halves; consistent O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	✗ NO	✓ YES	Pivot-based: partition around pivot; recursively sort left and right; worst

Algorithm	Best	Average	Worst	Space	Stable?	In-Place?	Identifying Behaviour
							case: sorted input + bad pivot
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	✗ NO	✓ YES	Build max-heap; repeatedly extract max to sorted end; guaranteed O(n log n) AND in-place

🔍 WORKED EXAMPLE

Bubble Sort trace — [5, 3, 1, 4, 2], one pass:

Compare 5,3: swap → [3,5,1,4,2]

Compare 5,1: swap → [3,1,5,4,2]

Compare 5,4: swap → [3,1,4,5,2]

Compare 5,2: swap → [3,1,4,2,5] ← 5 is in final position

Repeat passes until no swaps needed.

Merge Sort trace — [5,3,1,4]:

Split: [5,3] | [1,4]

Split: [5] [3] | [1] [4]

Merge: [3,5] | [1,4]

Merge: [1,3,4,5] ← sorted

⚠️ COMMON TRAPS

Stable vs In-place are INDEPENDENT:

Merge sort: STABLE but NOT in-place. Heap sort: in-place but NOT stable.

Selection sort: ALWAYS $O(n^2)$ — no early exit, no best-case improvement.

Quick sort: $O(n^2)$ WORST CASE when pivot is always the smallest or largest element.

Best way to choose pivot to avoid worst case: random pivot or median-of-three.

★ OA WORDING

- "Which sort has O(n) best case?" → Bubble sort AND Insertion sort (when already sorted)

⚠️ "Which sort ALWAYS runs $O(n^2)$ regardless of input?" → SELECTION SORT — no early-exit mechanism.

- "Which sort needs O(n) extra memory?" → MERGE SORT (out-of-place — needs auxiliary array)
- "Which sorts are STABLE?" → Bubble, Insertion, Merge (Selection, Quick, Heap are NOT)

⚠️ "Stable vs In-place are independent:" → Merge is stable but not in-place. Heap is in-place but not stable.

- "Which sort: guaranteed O(n log n) AND in-place AND NOT stable?" → HEAP SORT
- "Quick sort worst case?" → $O(n^2)$ — when pivot always lands on smallest/largest element
- "Identify this sort from pseudocode: compares adjacent elements, large values move to end" → Bubble sort
- "Identify: finds minimum each pass and places at front" → Selection sort
- "Identify: takes each element and inserts into correct position in sorted left side" → Insertion sort
- "Identify: splits in half, sorts halves, merges" → Merge sort
- "Identify: chooses pivot, partitions around it, recurses" → Quick sort

🎯 TUTOR DRILL TARGETS

→ State best/average/worst/space for all six sorts from memory
→ State which sorts are stable and which are in-place
→ Identify a sort from its pseudocode or described behaviour
→ Trace one pass of bubble sort on a 5-element array
→ Explain why selection sort has no best-case improvement
→ Explain quick sort's $O(n^2)$ worst case

DSA1 · Section 9 — Recursion

📖 **DEFINITION**

Recursive function: A function that calls ITSELF as part of its solution.

Base case: The stopping condition that does NOT call the function recursively. Without this → infinite recursion → stack overflow.

Recursive case: The part that calls the function on a SMALLER version of the problem. Must make progress toward the base case.

Call stack: Each recursive call adds a new frame to the call stack. When the base case is reached, frames resolve in reverse order (LIFO). Python default: ~1000 recursion levels before stack overflow.

9.1 Recursive Function Structure

★ **MUST KNOW COLD: Template for Every Recursive Function**

```
def f(n):
    if n == 0:          # BASE CASE — stopping condition
        return 1        # return directly, no recursive call
    else:               # RECURSIVE CASE
        return n * f(n-1)   # call itself with SMALLER input
```

Both parts are required: Missing base case → infinite recursion. Missing recursive case → not recursive.

9.2 Recursive Factorial

🔍 **WORKED EXAMPLE**

factorial(n):

Base case: factorial(0) = 1

Recursive case: factorial(n) = n × factorial(n−1)

Trace — factorial(4):

factorial(4) = 4 × factorial(3)
= 4 × 3 × factorial(2)
= 4 × 3 × 2 × factorial(1)
= 4 × 3 × 2 × 1 × factorial(0)
= 4 × 3 × 2 × 1 × 1 = 24

Big-O: O(n) — n recursive calls, each O(1) work.

9.3 Recursive Fibonacci — Complexity Example

🔍 **WORKED EXAMPLE**

Naive recursive Fibonacci:

```
def fib(n):
    if n <= 1: return n      # base cases: fib(0)=0, fib(1)=1
    return fib(n-1) + fib(n-2) # two recursive calls!
```

Big-O: O(2^n) — each call spawns TWO sub-calls. Call tree doubles at every level.
fib(5) calls fib(4)+fib(3), fib(4) calls fib(3)+fib(2), etc. Massive redundancy.

Memoized Fibonacci (cache results): O(n) — each sub-problem solved exactly once.

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

Version	Big-O	Why
Naive recursive	O(2^n)	Each call branches into two; exponential call tree with massive redundancy
Memoized	O(n)	Each unique sub-problem (fib(0)…fib(n)) solved exactly once
Bottom-up iterative	O(n)	Loop from 0 to n, store previous two values; no recursion overhead

9.4 Recursive Binary Search

🔍 **WORKED EXAMPLE**

Recursive binary search:

```
def bsearch(arr, low, high, target):
    if low > high: return −1      # base case: not found
    mid = (low + high) // 2
    if arr[mid] == target: return mid   # base case: found
    elif arr[mid] < target:
        return bsearch(arr, mid+1, high, target)  # recurse right
    else:
```

```
    return bsearch(arr, low, mid-1, target)   # recurse left
```

Big-O: O(log n) — halves the search space each call.

9.5 Infinite Recursion Traps

⚠ COMMON TRAPS

- No base case → function calls itself forever → Python raises RecursionError (stack overflow).
- Base case that is never reached → same result. E.g. if n starts negative and base case is n==0.
- Recursive case does not move toward base case → infinite loop. E.g. calling f(n) instead of f(n-1).

★ OA WORDING

- "What is the base case of factorial?" → factorial(0) = 1

⚠ "What happens with no base case?" → infinite recursion → stack overflow (RecursionError in Python).

- "Naive recursive Fibonacci is O(…)" → $O(2^n)$
- "Memoized Fibonacci is O(…)" → O(n)
- "Which traversals are naturally recursive?" → preorder, inorder, postorder (all use recursion)

⚠ "Recursive binary search Big-O?" → O(log n) — halves the range each call. Not O(n).

- "What does the call stack do in recursion?" → stores one frame per active call; resolves LIFO when base case reached

🎯 TUTOR DRILL TARGETS

→ State the two required parts of any recursive function
→ Trace factorial(4) step by step on the call stack
→ State the Big-O of naive Fibonacci and explain WHY it is $O(2^n)$
→ State the Big-O of memoized Fibonacci and explain WHY it is O(n)
→ Identify infinite recursion conditions from a given function
→ Trace recursive binary search on a small array

DSA1 / C949 — Complete Coverage Checklist

Check each box before attempting the OA.

□	Topic	□	Topic
□	Algorithm definition and 5 properties (input/output/finiteness/definiteness/effectiveness)	□	Linked list node structure (data + pointer)
□	Big-O, Big-Ω, Big-Θ — what each means	□	Singly vs doubly linked list (next only vs next+prev)
□	Best / average / worst case complexity	□	Linked list head and tail
□	Time complexity vs space complexity	□	Array vs linked list tradeoffs
□	Big-O hierarchy (O(1) through O(n!))	□	Stack — LIFO; push, pop, peek
□	Simplifying Big-O (drop constants, keep dominant)	□	Stack operation tracing
□	Sequential loops = O(n), nested loops = $O(n^2)$	□	Stack use cases (call stack, undo, DFS)
□	Different input sizes O(a+b) and O(a·b)	□	Queue — FIFO; enqueue, dequeue, front/peek
□	Reading Big-O from code snippets	□	Queue operation tracing
□	ADT vs data structure (interface vs implementation)	□	Queue use cases (BFS, scheduling)
□	OOP: class, object, instance, attribute, method, constructor	□	Hash function h(k) = k mod m
□	Encapsulation, inheritance, polymorphism, abstraction	□	Collision — definition and why it happens
□	Python: list/tuple/dict/set — ordered/mutable/duplicates	□	Separate chaining vs open addressing
□	Python operation Big-O (index O(1), insert/search O(n))	□	Load factor α = n/m
□	range(n) starts at 0, stops BEFORE n	□	Hash table average O(1) vs worst O(n)
□	Array/list index access O(1)	□	BST property: left < node < right
□	Array insert/delete O(n); append O(1) amortized	□	Balanced vs unbalanced BST and impact on Big-O
□	Bag ADT — unordered, allows duplicates	□	4 tree traversals: preorder / inorder / postorder / level-order
□	List ADT	□	Inorder BST = sorted output
□	Static vs dynamic array	□	Level-order uses a QUEUE
□	Binary search — requires sorted; O(log n)	□	Min-heap vs max-heap property

□	Topic	□	Topic
□	Linear search — O(n)	□	Heap operations: O(1) peek, O(log n) insert/extract
□	Bubble sort — stable, in-place, O(n) best, O(n^2) worst	□	Build heap from n items = O(n)
□	Selection sort — NOT stable, O(n^2) always	□	Heap array indexing (parent=i/2, children=2i, 2i+1)
□	Insertion sort — stable, O(n) best, O(n^2) worst	□	Recursive function: base case + recursive case
□	Merge sort — stable, NOT in-place, O(n log n) always	□	Call stack and stack overflow
□	Quick sort — NOT stable, O(n^2) worst case	□	Factorial recursion trace
□	Heap sort — NOT stable, in-place, O(n log n) always	□	Naive Fibonacci O(2^n) vs memoized O(n)
□	Stable vs unstable sorting definition	□	Recursive binary search O(log n)
□	In-place vs extra memory definition	□	Identify sort/search from pseudocode/behaviour

PA / OA Readiness Checklist — Can You Do These WITHOUT Notes?

These are the SKILLS the OA tests. Each item is something you must be able to DO, not just recognise. Go through this list with ChatGPT before scheduling either OA.

DM1 / C959 — Skills Checklist

□	You can DO this without notes
□	Build a complete truth table for any 2-variable compound expression
□	Classify any expression as tautology / contradiction / contingency
□	Apply De Morgan's Laws (logic, sets, and Boolean) without looking them up
□	State all four conditional forms and which two are equivalent to the original
□	Convert a truth table to DNF and to CNF
□	Negate any single or nested quantifier statement
□	Translate English sentences to predicate logic with correct ∀ / ∃ and connective choice
□	Name any of the 8 rules of inference given its argument pattern
□	Identify "affirming the consequent" and "denying the antecedent" as fallacies
□	Write the base case and inductive step for a summation induction
□	Compute union, intersection, difference, and symmetric difference of two sets
□	Apply 2-set and 3-set inclusion-exclusion to a word problem
□	State $\lvert\mathscr{P}(A)\rvert$ for any set size
□	Classify a function as injective / surjective / bijective
□	Find the inverse of a linear function
□	Evaluate $\lfloor x \rfloor$ and $\lceil x \rceil$ for negative non-integer values
□	Test a relation for all four properties from ordered pairs
□	Classify a relation as equivalence relation or partial order
□	Simplify a Boolean expression using the identities
□	Multiply two 2×2 matrices
□	Apply a row operation given its notation
□	State the differences between REF and RREF
□	Find the nth term of an arithmetic and geometric sequence
□	Apply all four closed-form summation formulas
□	State the handshaking lemma and use it to find edge count
□	Determine whether an Euler path or Euler circuit exists

□	You can DO this without notes
□	Check graph isomorphism using degree sequences
□	State height, depth, and edge count for a tree with n vertices

DSA1 / C949 — Skills Checklist

□	You can DO this without notes
□	State the 5 algorithm properties
□	Rank the 8 Big-O orders from fastest to slowest
□	Determine Big-O for a code snippet with sequential or nested loops
□	Simplify $O(5n^2 + 3n + 100)$ to $O(n^2)$
□	State best/average/worst Big-O for all six sorting algorithms
□	State which sorts are stable and which are in-place
□	Identify a sort from its pseudocode or described behaviour
□	Trace one pass of bubble sort
□	State why selection sort has no best-case improvement
□	Explain quick sort's $O(n^2)$ worst case
□	Trace linear search and binary search on a small array
□	State binary search's sorted-array requirement
□	Trace a push/pop sequence for a stack; state what is returned
□	Trace an enqueue/dequeue sequence for a queue
□	Apply h(k) = k mod m to find bucket for a key
□	Explain separate chaining and what happens on a collision
□	State average vs worst case for hash table operations
□	Trace BST insert for 4–5 values
□	Write all 4 traversal sequences for a small BST
□	State which traversal gives sorted output
□	State which traversal uses a queue
□	State min-heap and max-heap root property
□	Trace factorial(4) on the call stack
□	State the Big-O of naive Fibonacci vs memoized and explain why
□	Identify what infinite recursion is and when it occurs

□	You can DO this without notes
□	State the difference between a class and an object
□	State what encapsulation means in OOP
□	List Python list/tuple/dict/set and classify each by ordered/mutable/duplicates