Visual Thinking and Problem Understanding

Unit 2: Problem Visualisation, Visual Tools, Algorithms & Pseudocode

Dr. Gopal Chandra Jana

Dept. of CSE, Sharda University
Fundamentals of Computing and AI (CSAI1101) — Unit 2

September 2, 2026

Course Page:
https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/

Outline

Learning Outcomes of Unit 2

After this unit, a student will be able to...

1. **Explain** what problem visualisation is and distinguish *mental models* from *external representations*.
2. **Describe** the four pillars of computational thinking and how visuals support them.
3. **Use** the right visual tool — sketch, mind map, list, table, timeline or storyboard — for a given problem.
4. **Define** an algorithm, state its properties, and judge whether a procedure is a valid algorithm.
5. **Write** clear pseudocode for simple problems using a standard problem-solving approach.

Mapping to the Syllabus

A. Problem visualisation: mental models vs external representations. **B.** Visual tools: sketches, mind maps, lists, tables, timelines, storyboards. **C.** Algorithms & pseudocode: the problem-solving approach.

From Unit 1 to Unit 2

Where we are

In **Unit 1** we learnt *what a computer is* and *how it stores data*. But a computer is useless until we tell it **how to solve a problem**.

- Before we *code*, we must *understand* the problem.
- Before we understand, it helps to **see** the problem — to visualise it.
- This unit builds the **thinking skills** that make you a good programmer and AI engineer.

The golden rule

"A problem well understood is a problem half solved." Visual thinking helps us understand first, and code later.

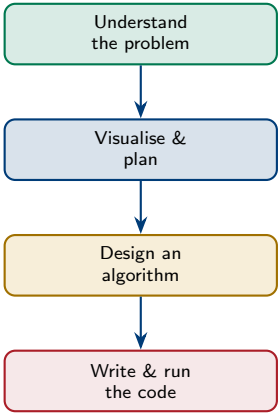


Figure: Coding is the *last* step, not the first.

What is Problem Visualisation?

Seeing a problem before solving it

Problem visualisation means turning a problem into a **picture, diagram or structured form** so we can understand it more clearly and spot the solution.

- Our brain understands *pictures* faster than *walls of text*.
- A good diagram makes hidden structure **visible**.
- It reduces confusion, mistakes and forgotten steps.

Everyday proof

Which is easier to follow — a *written paragraph* of directions, or a *map*? The map wins, because it visualises the problem.

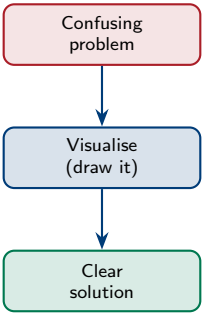


Figure: A picture turns confusion into clarity.

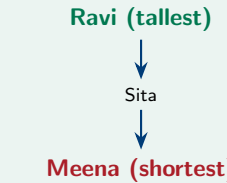
Why a Picture Beats a Paragraph

The problem in words

"Ravi is taller than Sita. Sita is taller than Meena. Meena is shorter than Ravi. Who is the tallest and who is the shortest?"

You have to re-read this a few times...

The same problem as a picture



One glance gives the answer!

Lesson: the information was identical — only the *representation* changed. Good representation makes thinking easy.

Mental Models — The Picture in Your Head

What is a mental model?

A **mental model** is the picture or idea we build **inside our mind** to understand how something works.

- You imagine a *clock's gears* to guess how it keeps time.
- You picture a *map* in your head to reach the canteen.
- A programmer imagines *how data flows* through a program.

Mental models let us *predict*, *reason* and *plan* without touching the real thing.

But mental models have limits

They are *invisible* (others can't see them), *fuzzy*, *easy to forget*, and *limited* by our memory. Two people may picture the same problem differently.

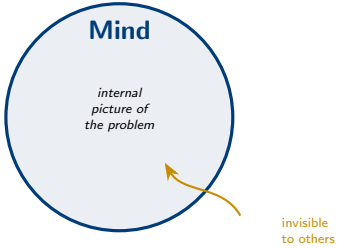


Figure: A mental model lives inside the head.

External Representations — Getting It Out of Your Head

What is an external representation?

An **external representation** is a mental model made **visible on paper or screen** — a sketch, diagram, table, flowchart or written note.

- It is **shareable**: others can see and check it.
- It is **permanent**: it won't fade like memory.
- It **offloads** memory: the paper remembers, so your brain is free to think.
- It reveals **errors** you would miss in your head.

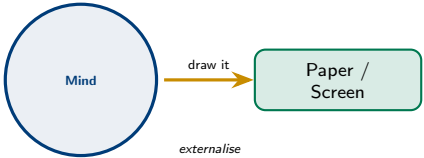


Figure: Externalising = drawing the mental model out.

Mental Models vs. External Representations

Aspect	Mental Model	External Representation
Where it lives	Inside the mind	On paper / screen
Visible to others	No — private	**Yes** — shareable
Permanence	Fades, easily forgotten	**Stays**, can be revisited
Memory load	High — you must hold it	**Low** — paper remembers
Error-spotting	Hard	**Easy** — mistakes become visible
Teamwork	Difficult to share	**Easy** — everyone sees the same thing
Example	Imagining a seating plan	A drawn seating chart

They work together: we *think* with mental models and *externalise* them to check, share and refine.

Computational Thinking — Thinking Like a Problem Solver

The idea (Jeannette Wing, 2006)

Computational thinking is a way of solving problems so clearly and step-by-step that a *human or a computer* could carry out the solution.

- It is a **thinking skill**, not a programming language.
- Useful in *every* field — not just computing.
- It is the bridge from a *messy real problem* to a *clear algorithm*.

Visuals + computational thinking

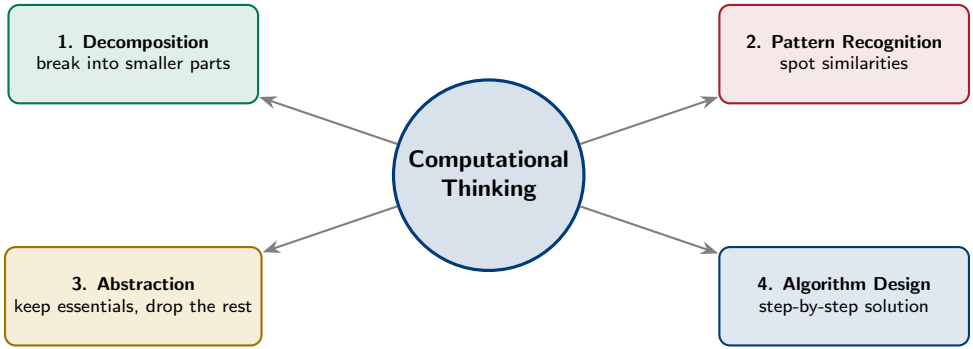
Visual tools make each pillar easier:

- a *mind map* helps **decomposition**,
- a *table* exposes **patterns**,
- a *sketch* supports **abstraction**,
- a *flowchart* shows the **algorithm**.

Remember

Wing said computational thinking is as basic a skill as *reading, writing and arithmetic*.

The Four Pillars of Computational Thinking



Together, these four pillars turn any complex problem into a solvable, step-by-step plan.

The Four Pillars — A Worked Example (Making Tea)

1. Decomposition

Break "make tea" into steps: boil water, add tea leaves, add milk & sugar, strain, pour.

3. Abstraction

Ignore irrelevant details (cup colour, brand of sugar). Keep only what affects the tea.

2. Pattern Recognition

"Boil water" also appears in making coffee, noodles, soup — a *reusable* sub-task.

4. Algorithm Design

Put the steps in the correct *order* to get a repeatable recipe anyone can follow.

The same four pillars solve a coding problem, a maths problem, or a real-life problem.

Benefits of Visualisation for Computational Thinking

What visuals give us

- **Clarity**: complex ideas become simple to grasp.
- **Memory relief**: the paper holds the details.
- **Error detection**: gaps and mistakes stand out.
- **Communication**: the whole team shares one view.
- **Faster planning**: try ideas quickly before coding.

A picture is worth. . .

Studies suggest the brain processes images far faster than text. Around **65%** of people are "visual learners" who understand best when they can *see* an idea.

Design tip

A good visual is *simple*. If a diagram is more confusing than the problem, simplify it — less is more.

Quick Check — Round 1 (Visualisation)

[Q1] Define a "mental model" in one line.

[Q2] Give two advantages of an external representation over a mental model.

[Q3] Name the four pillars of computational thinking.

[Q4] True/False: Computational thinking can only be used for computer programming.

Think for 60 seconds — answers on the next slide.

Answers — Round 1

[A1]

A **mental model** is the internal picture or idea we build in our mind to understand how something works.

[A2]

It is **shareable** (others can see it) and **permanent** (it does not fade); it also *reduces memory load* and *reveals errors* (any two).

[A3]

Decomposition, **Pattern Recognition**, **Abstraction**, **Algorithm Design**.

[A4]

False. It is a general problem-solving skill useful in *every* field — science, business, daily life — not only programming.

A Toolbox of Visual Tools

Choose the right tool for the job

Different problems need different visuals. Here are six everyday tools we will study:

1. **Sketch** — a quick rough drawing.
2. **Mind map** — ideas branching from a centre.
3. **List** — items in order or as a checklist.
4. **Table** — rows & columns of related data.
5. **Timeline** — events along a time line.
6. **Storyboard** — a sequence of frames.

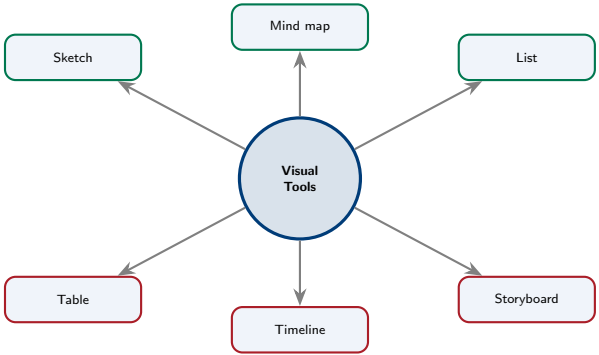


Figure: Six tools, one goal — understand the problem.

Tool 1 — Sketches

A quick, rough drawing

A **sketch** is a fast hand drawing that captures the shape or idea of a problem — neatness does not matter, *clarity* does.

- Great for **spatial** problems: layouts, shapes, how parts fit.
- Perfect for a *first* idea — draw, discard, redraw.
- Used everywhere: app screens (**wireframes**), circuits, room plans.

When to use

"Design the login screen of an app" → sketch the boxes and buttons before writing any code.

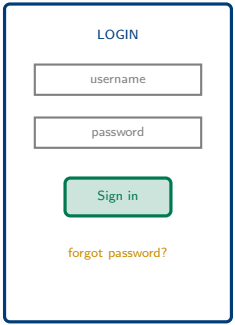


Figure: A sketch (wireframe) of a login screen.

Tool 2 — Mind Maps

Ideas branching from a centre

A **mind map** starts with the main idea in the middle and branches out into sub-ideas — like a tree.

- Excellent for **brainstorming** and **decomposition**.
- Shows how sub-topics *connect* to the main topic.
- Popularised by **Tony Buzan**.

When to use

Planning a project, revising a chapter, or breaking a big task into parts.

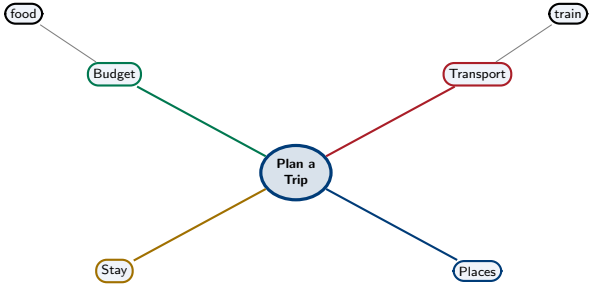


Figure: A mind map for planning a trip.

Tool 3 — Lists

Items, one after another

A **list** arranges items in a line or column. Two common kinds:

- **Ordered list**: sequence matters (steps of a recipe).
- **Unordered list**: sequence does not matter (a shopping list).

Lists are perfect for *checklists*, *to-do items* and *ordered steps*.

Why lists help coding

Ordered lists are the **seed of an algorithm** — a sequence of steps in the right order.

Example

Ordered (steps to log in):

1. Open the app
2. Enter username
3. Enter password
4. Tap "Sign in"

Unordered (things to carry):

- ID card
- Notebook
- Pen

Tool 4 — Tables

Rows and columns of related data

A **table** organises data into **rows** and **columns** so we can compare items at a glance.

- Great for **comparison** and spotting **patterns**.
- Each column is a property; each row is an item.
- Basis of *spreadsheets* and *databases*.

When to use

Comparing phones by price and battery, or timetables of classes.

Phone	Price	Battery	RAM
A	Rs. 12k	4000 mAh	6 GB
B	Rs. 18k	5000 mAh	8 GB
C	Rs. 22k	4500 mAh	12 GB

Figure: A table makes comparison instant — your eye scans down each column.

Tool 5 — Timelines

Events along a line of time

A **timeline** places events in the order they happen along a horizontal line.

- Shows **sequence** and **duration**.
- Great for *history*, *project schedules*, *planning*.
- Helps answer "what comes before / after?"

When to use

A semester plan, a project deadline chart (*Gantt chart*), or the history of computers from Unit 1!

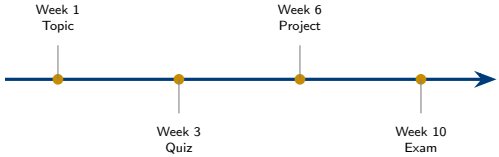


Figure: A timeline of a semester's key events.

Tool 6 — Storyboards

A sequence of frames

A **storyboard** is a series of small pictures (frames) showing **what happens step by step** — like a comic strip.

- Shows a *flow* of actions or screens over time.
- Used for films, animations and **app screen flows**.
- Helps everyone agree on the "story" before building.

When to use

Designing how a user moves through an app: splash → login → home → profile.

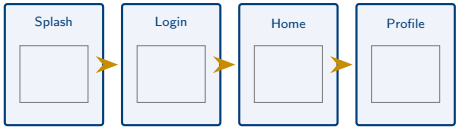


Figure: A storyboard of an app's screen flow.

Which Tool When? — A Quick Guide

Tool	Best for	Example
Sketch	Shapes, layouts, spatial ideas	App screen, room plan
Mind map	Brainstorming, breaking a topic down	Planning a project
List	Steps or checklists	Recipe steps, to-do list
Table	Comparing data, spotting patterns	Phone comparison
Timeline	Order of events over time	Semester schedule
Storyboard	A flow of actions or screens	App user journey

Tip: many real problems use *several* tools together — e.g. a mind map to plan, then a list to order the steps.

Quick Check — Round 2 (Visual Tools)

[Q1] Which tool is best for *comparing* five laptops by price and RAM?

[Q2] What is the difference between an ordered list and an unordered list?

[Q3] Which tool would you use to show the *screen flow* of a mobile app?

[Q4] You must brainstorm all sub-topics of "Healthy Living". Which tool fits best?

Answers — Round 2

[A1]

A **table** — rows for laptops, columns for price and RAM, so you can compare down each column.

[A2]

In an **ordered list** the sequence *matters* (e.g. recipe steps); in an **unordered list** it *does not* (e.g. shopping items).

[A3]

A **storyboard** — a sequence of frames showing screen after screen.

[A4]

A **mind map** — "Healthy Living" at the centre, with branches like diet, exercise, sleep.

What is an Algorithm?

Definition

An **algorithm** is a **finite**, step-by-step set of **unambiguous** instructions that solves a problem or completes a task.

- It takes some *input*, does clear steps, and gives an *output*.
- Every step must be so clear that anyone (or any computer) can follow it the same way.
- The word comes from the 9th-century mathematician **al-Khwarizmi**.

Everyday algorithms

A cooking recipe, directions to a shop, and steps to withdraw cash from an ATM are all algorithms.

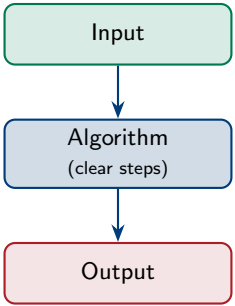


Figure: An algorithm turns input into output through clear steps.

Example: An Everyday Algorithm

Algorithm: Make a cup of tea

1. Start.
2. Boil water in a pan.
3. Add tea leaves.
4. Add milk and sugar.
5. Let it boil for 3 minutes.
6. Strain into a cup.
7. Serve.
8. Stop.

Notice the good qualities

- It has a clear **start** and **stop**.
- Steps are in the correct **order**.
- Each step is **clear** — no confusion.
- It **ends** after a finite number of steps.

A bad step

"Add *some* sugar" is vague. "Add 2 spoons of sugar" is precise. Algorithms must be **precise**.

Characteristics of a Good Algorithm

Knuth's five properties

Every valid algorithm must have:

1. **Finiteness**: it must end after a finite number of steps.
2. **Definiteness**: each step is precise and unambiguous.
3. **Input**: zero or more inputs are supplied.
4. **Output**: it produces at least one output.
5. **Effectiveness**: each step is basic enough to be done, in principle, with pen and paper.

Also desirable

- **Correctness**: gives the right answer for every valid input.
- **Generality**: works for the whole class of problems, not one case.
- **Efficiency**: uses little time and memory.

Not an algorithm

"Keep adding 1 forever" fails *finiteness* — it never stops, so it is **not** an algorithm.

The Problem-Solving Approach

A reliable recipe for any problem

1. **Understand** the problem (what is asked?).
2. Identify **inputs** and **outputs**.
3. **Decompose** into smaller steps.
4. **Design** the algorithm (order the steps).
5. **Write** it as pseudocode / a flowchart.
6. **Test** with examples (dry run).
7. **Refine** and fix any errors.

Polya's four steps

The mathematician **George Polya** summarised problem solving as:

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan.
4. Look back (check the result).

Key habit

Never jump straight to code. **Understand**, then **plan**, then **code**.

What is Pseudocode?

Half-English, half-code

Pseudocode is a way of writing an algorithm using **plain, structured English** that looks like code but ignores the strict rules of any one programming language.

- Easy to read — focuses on *logic*, not syntax.
- Language-independent — any programmer can turn it into code.
- A bridge between the *idea* and the *program*.

Common keywords

`START`, `STOP`, `INPUT`, `READ`, `PRINT`, `IF...THEN...ELSE`, `WHILE`, `FOR`, `SET`.

Algorithm vs. Pseudocode vs. Program

- **Algorithm**: the plan, in any form.
- **Pseudocode**: the plan written code-like, in English.
- **Program**: the plan in a real language (Python, C).

No strict rules

Pseudocode has *no* single official standard — keep it clear, consistent and easy to read.

Pseudocode Example 1 — Add Two Numbers

Pseudocode

```
START
  INPUT a
  INPUT b
  SET sum = a + b
  PRINT sum
STOP
```

Reading it step by step

- Read two numbers a and b.
- Add them and store in sum.
- Print sum.

Dry run: $a = 4$, $b = 6$

sum $= 4 + 6 = 10$. Output: **10**. ✓

Pseudocode Example 2 — Largest of Two Numbers

Pseudocode (decision)

```
START
  INPUT a
  INPUT b
  IF a > b THEN
     PRINT a
  ELSE
     PRINT b
  ENDIF
STOP
```

The IF–THEN–ELSE idea

The program makes a **decision**: it checks a condition and chooses one of two paths.

Dry run

- $a = 7$, $b = 3$: is $7 > 3$? Yes $\rightarrow$ print **7**.
- $a = 2$, $b = 9$: is $2 > 9$? No $\rightarrow$ print **9**.

Pseudocode Example 3 — Sum of First N Numbers (Loop)

Pseudocode (repetition)

```
START
  INPUT n
  SET sum = 0
  FOR i = 1 TO n DO
     SET sum = sum + i
  ENDFOR
  PRINT sum
STOP
```

The loop idea

A **loop** repeats steps without writing them again and again. Here we add $1, 2, \ldots, n$.

Dry run: $n = 4$

i	step	sum
1	$0 + 1$	1
2	$1 + 2$	3
3	$3 + 3$	6
4	$6 + 4$	10

Output: **10**.

The Three Building Blocks of Any Algorithm

Every algorithm uses only three structures

1. **Sequence**: steps one after another, top to bottom.
2. **Selection (decision)**: choose a path with `IF...ELSE`.
3. **Iteration (loop)**: repeat steps with `FOR` / `WHILE`.

Any program — however big — is built from just these three!

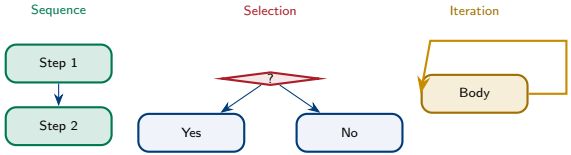


Figure: Sequence, Selection and Iteration.

A Peek Ahead — Flowcharts (Unit 3)

Another way to show an algorithm

A **flowchart** draws an algorithm using standard symbols joined by arrows. We will study it fully in **Unit 3**.

- *Oval* — start / stop
- *Parallelogram* — input / output
- *Rectangle* — a process step
- *Diamond* — a decision

Pseudocode vs. flowchart

Same logic, two views: pseudocode is *text*, a flowchart is a *picture*. Both describe the same algorithm.

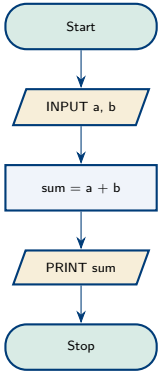


Figure: A flowchart for "add two numbers".

Quick Check — Round 3 (Algorithms & Pseudocode)

[Q1] List any three of Knuth's five properties of an algorithm.

[Q2] Why is "keep adding 1 forever" *not* an algorithm?

[Q3] Name the three basic building blocks of any algorithm.

[Q4] Write pseudocode to read a number and print whether it is positive or negative.

Try Q4 on paper — answers on the next slide.

Answers — Round 3

[A1]

Any three of: **finiteness**, **definiteness**, **input**, **output**, **effectiveness**.

[A2]

It never ends — it fails the **finiteness** property. An algorithm must stop after a finite number of steps.

[A3]

Sequence, **Selection** (decision), **Iteration** (loop).

[A4]

```
START
  INPUT n
  IF n >= 0 THEN
     PRINT "Positive"
  ELSE
     PRINT "Negative"
  ENDIF
STOP
```

Summary of Unit 2

What we covered

1. **Problem visualisation** — turning a problem into a picture makes it far easier to understand and solve.
2. **Mental models vs external representations** — we *think* with internal models and *externalise* them to share, check and remember.
3. **Computational thinking** — decomposition, pattern recognition, abstraction and algorithm design.
4. **Visual tools** — sketches, mind maps, lists, tables, timelines and storyboards, each suited to a kind of problem.
5. **Algorithms** — finite, precise steps with Knuth's five properties; built from sequence, selection and iteration.
6. **Pseudocode** — plain-English, language-independent way to write an algorithm before coding.

Understand first, visualise, then design the algorithm — coding comes last.

Key Terms to Remember

Visualisation & thinking

- **Mental model** = picture in the mind
- **External representation** = on paper/screen
- **Decomposition** = break into parts
- **Pattern recognition** = spot similarities
- **Abstraction** = keep essentials only

Algorithms

- **Algorithm** = finite, precise steps
- **Pseudocode** = code-like English
- **Sequence / Selection / Iteration**
- **Finiteness** = it must stop
- **Definiteness** = no ambiguity

Practice Questions (Exam Oriented)

Short answer (2–5 marks)

1. Differentiate between a mental model and an external representation.
2. List and explain the four pillars of computational thinking.
3. State Knuth's five properties of an algorithm.
4. When would you choose a table over a mind map? Give one example each.
5. Define pseudocode and give two of its advantages.

Long answer (8–10 marks)

6. Explain the problem-solving approach step by step, with an example.
7. Write an algorithm and pseudocode to find the largest of three numbers.
8. Write pseudocode to find the average of `n` numbers, and do a dry run for 3 numbers.
9. Compare sequence, selection and iteration with a small example of each.
10. Explain, with examples, how the six visual tools support the four pillars of computational thinking.

Think & Explore (Take-Home)

Mini tasks

- Draw a *mind map* of "My Week".
- Make a *table* comparing three of your favourite apps.
- Write an *algorithm* for withdrawing cash from an ATM.
- Write *pseudocode* to check whether a number is even or odd, and dry-run it for 7 and 10.

Reading & reflection

- Read Jeannette Wing's short article *"Computational Thinking"* (2006).
- Notice five visual tools you meet in a day (maps, menus, timetables. . .).
- Prepare for Unit 3: *flowcharts, decision tables and dry runs*.

Reminder

Post any doubts on the course page. Practise writing pseudocode — it is a skill you build by doing.

References

Textbooks & sources (as per the course page)

1. Peter Norton, *Introduction to Computers*, Tata McGraw Hill, 6th Edition. **[Main text]**
2. R. G. Dromey, *How to Solve it by Computer*, Pearson.
3. Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, Pearson.
4. G. Polya, *How to Solve It*, Princeton University Press.

Key papers & further reading

5. J. M. Wing, "Computational Thinking," *Communications of the ACM*, 49(3):33–35, 2006.
6. D. E. Knuth, *The Art of Computer Programming, Vol. 1* (algorithm properties).
7. T. Buzan, *The Mind Map Book* (mind mapping).

Course page: https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/

Thank You

Any Questions?

"The best way to understand a problem is to see it."
— A guiding idea of visual thinking

Post your doubts on the course page →
https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/#Post-doubts