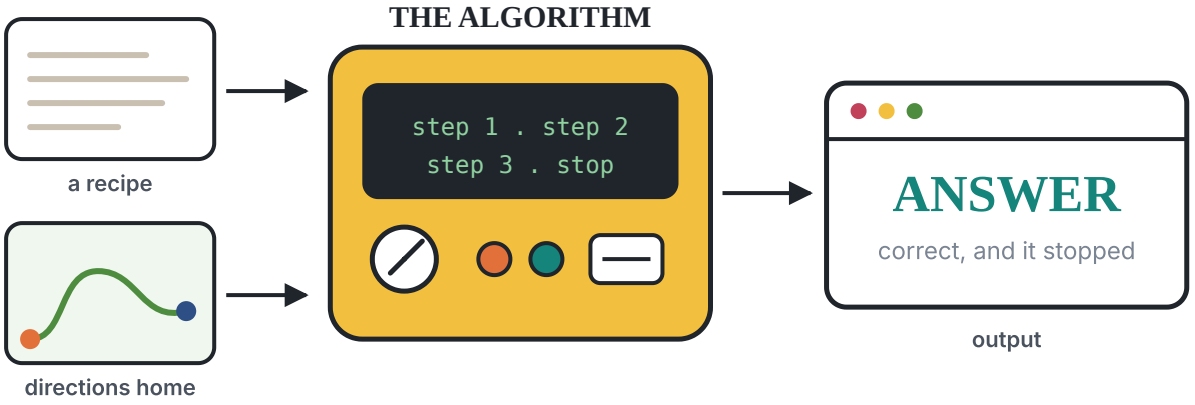
CSC207 *Made Easy*

The full Introduction to Algorithms syllabus, written in plain English. Every difficult idea is explained with an example from ordinary life, and every chapter is illustrated.



Based on the CSC207 Introduction to Algorithms course.
Nothing in the syllabus has been left out.

What is inside

PART 1 · THE IDEA

1 You have been writing algorithms your whole life

2 The five rules an algorithm cannot break

3 Easy, hard, and do not even try

PART 2 · THE THINGS ALGORITHMS TOUCH

4 Data, names and boxes

5 Types: what you are allowed to do with a thing

6 Data structures: picking the right container

7 Algorithm vs program vs language

8 What programming actually is

PART 3 · WRITING IT DOWN

9 Pseudocode: English wearing a tie

10 Flowcharts: the picture version

11 The only three shapes you need

12 Tea, traffic lights and a toll gate

13 Factorial, from idea to running C

14 Euclid's algorithm, the oldest trick in the book

PART 4 · HOW TO ATTACK A PROBLEM

15 Top-down and bottom-up

16 Greedy: grab the best thing in front of you

17 Divide and conquer: break it, solve it, join it

18 Dynamic programming: never solve the same thing twice

19 The rest of the family

PART 5 · IS IT FAST? IS IT FAT?

20 Time complexity and the frequency count

21 Space complexity

22 Best case, worst case, average case

23 Big O notation

24 The exam survival sheet

HOW TO READ THIS BOOK

Every chapter follows the same shape. First the idea in plain words. Then a **real life** box, giving a situation you have already handled yourself. Then a **stick it in your brain** box, holding the one sentence worth carrying into an exam.

The definitions, the pseudocode, the code samples and the complexity rules are all here in full, and worded as the course words them. What has been added around them is the explanation.

PART 1

The idea

Before any code, any Big O, any exam. What is this thing we are actually studying?

1 You have been writing algorithms your whole life

What an algorithm is, and why a program is only one kind of algorithm.

This morning you did something like this. Wake up. Check the time. If you are late, skip breakfast, if not, eat. Find your shoes. Lock the door. Take the bike. Pay 300 francs. Walk the last part.

That is an algorithm in the strict sense, not a loose comparison. It has a starting point, a clear list of steps, a decision in the middle, and a point at which it ends, with you standing in front of your class.

Here is the formal definition. Read it slowly, because every clause in it is doing work:

THE DEFINITION, WORD FOR WORD

An algorithm consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in finite time.

Dromey, *How to Solve it by Computer*, 1982. Woodhouse puts it more briefly: "a finite, definite, effective procedure, with some output."

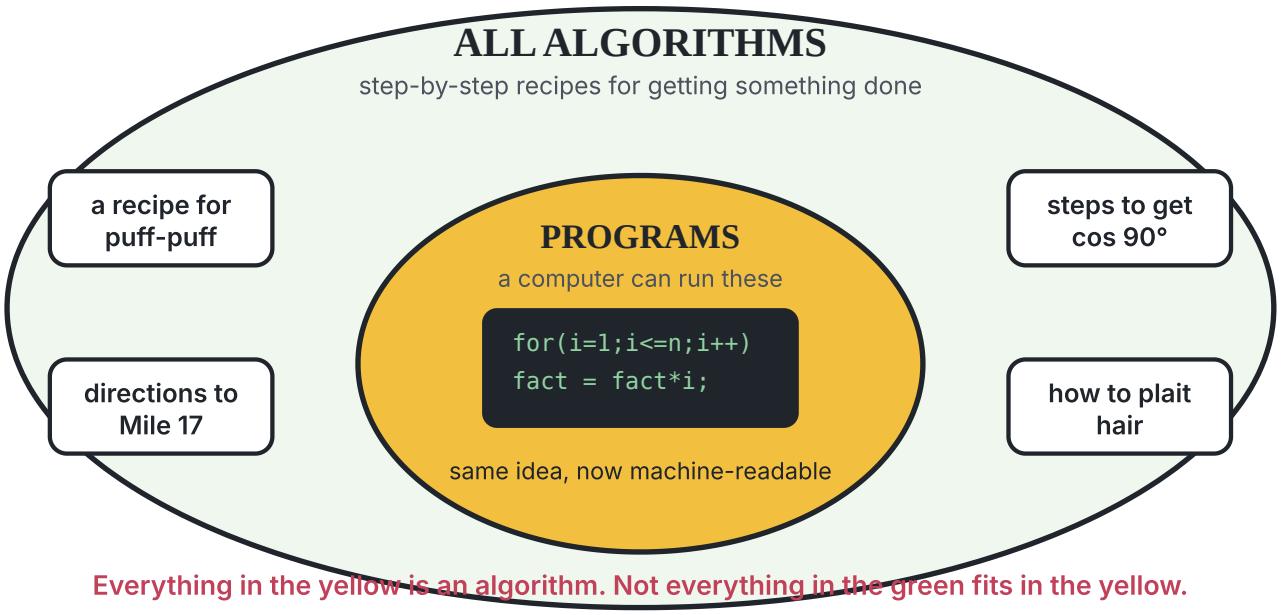
Read that again slowly and translate each piece:

- **Explicit and unambiguous.** No "add a little salt." How much salt?
- **Finite steps.** You can count them. The list ends on paper.
- **Given initial conditions.** You need something to start with. Money, water, a list of numbers.
- **Produce the corresponding output.** Something comes out. A plate of food, an answer, a sorted list.
- **Terminate in finite time.** It actually finishes. It does not run until the world ends.

The part people get wrong in exams

A program is an algorithm. But an algorithm is not always a program.

A program is just an algorithm that has been written in a way a computer can execute. Your mother's recipe for eru is a complete, correct algorithm, but your laptop cannot run it. The steps to compute the cosine of 90 degrees is an algorithm. Directions to your friend's house in Molyko is an algorithm.



The one-way rule. Every program is an algorithm. Most algorithms in the world are not programs. If an exam asks you to explain the relationship, draw this.

REAL LIFE

Think of a tailor in town. The pattern he follows to cut a kaba is the algorithm. It lives in his head and in chalk marks on cloth. The day someone writes that pattern into a cutting machine's software, it becomes a program. The pattern did not change. Only the audience changed, from a human to a machine.

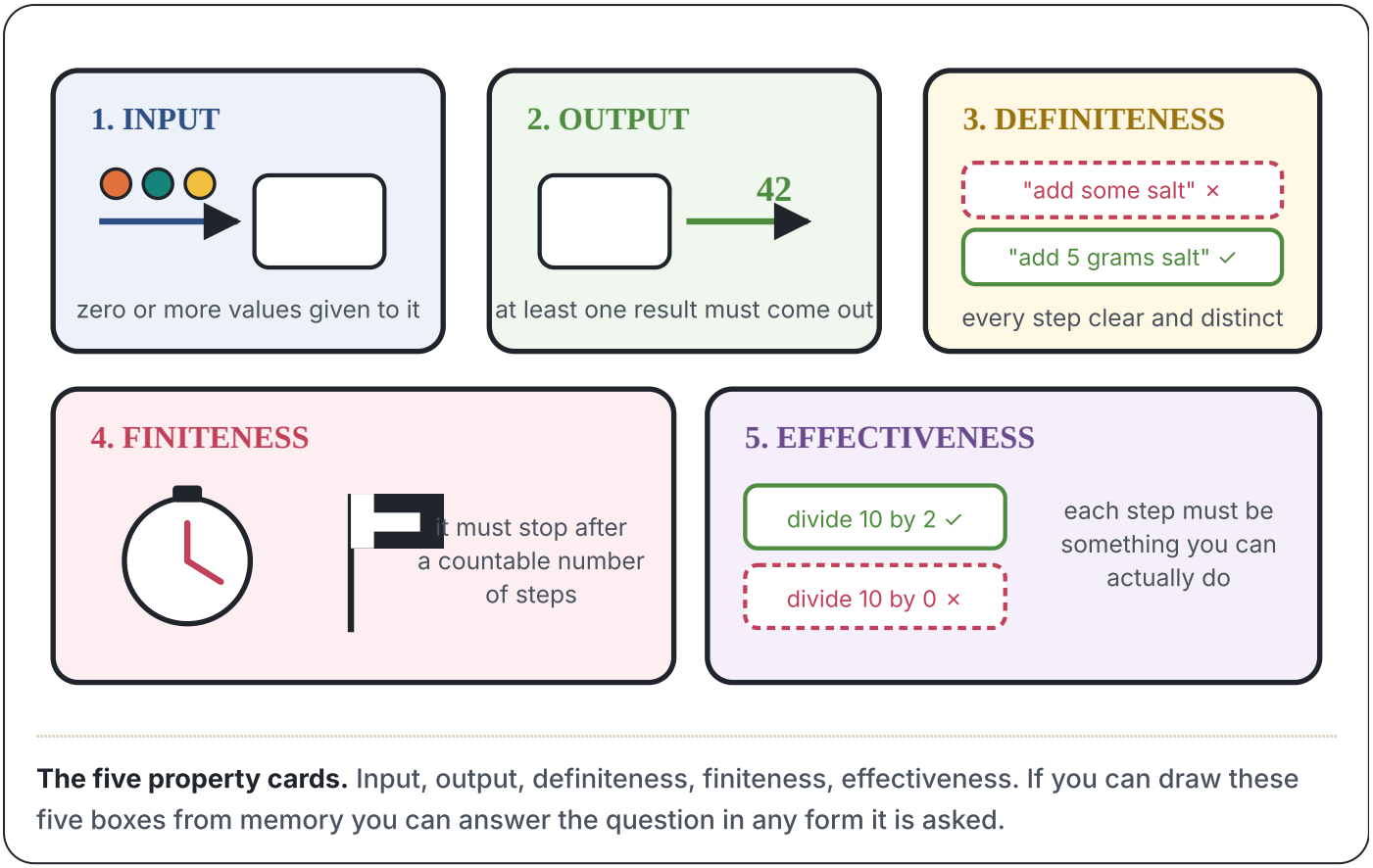
STICK IT IN YOUR BRAIN

An algorithm is a **finite set of precise instructions** for solving a problem. A program is an algorithm the computer can execute. All programs are algorithms. Not all algorithms are programs.

2 The five rules an algorithm cannot break

Input, output, definiteness, finiteness and effectiveness, and what fails without each.

An algorithm has five required properties. Memorising the five words is not enough, because an exam will ask why each one matters. Each is set out below with the failure that follows when it is missing.



The five property cards. Input, output, definiteness, finiteness, effectiveness. If you can draw these five boxes from memory you can answer the question in any form it is asked.

One by one, with the failure

1. Input. An algorithm takes zero or more quantities supplied from outside. Note zero or more, not one or more. An algorithm that just prints "hello" takes no input and is still an algorithm.

2. Output. It must produce one or more results after processing the statements. An algorithm that computes something and tells nobody is useless. Something has to come out.

3. Definiteness. Each instruction must be clear and distinct. "Stir until it is nice" fails. Nice according to who? A second person reading your steps must do exactly what you did.

4. Finiteness. It must terminate after a finite number of steps. This is the rule people break by accident. A loop with a condition that never becomes false is not a slow algorithm. It is not an algorithm at all.

5. Effectiveness. Each operation must be definite and also feasible. Pen-and-paper doable. "Guess the correct password" is not effective. "Try every password from 0000 to 9999" is effective, even if it is slow.

REAL LIFE

A njangi meeting runs on an algorithm, and it hits all five rules. **Input**: who is present and who contributed. **Output**: whose turn it is to collect. **Definiteness**: the order is written in a book, not decided by mood. **Finiteness**: after every member has collected once, the round ends. **Effectiveness**: counting money is something a human can really do.

Now imagine a njangi where the rule is "we pay whoever needs it most." Nobody can agree what that means and the meeting never ends. That rule fails definiteness, and so it fails finiteness too. One broken rule usually breaks another.

WHERE STUDENTS LOSE MARKS

Writing "an algorithm must have input" instead of "zero or more input." The notes say zero or more. Say zero or more.

TRY THIS (THIS IS YOUR HOMEWORK FROM THE NOTES)

You are told to give the result of x squared, where x is any non-negative integer.

1. State the computational problem in prose. Say in general terms what the problem is and its constraints, then its input, then its output.
2. Give two algorithms for it. One using **multiplication** of x. One using only **addition** of x.
3. Which is more efficient, and why?
4. Is each one correct? Does each one terminate? Why or why not?

A hint: run both by hand on small inputs. Then run them on the inputs that are special for integers, which are 0 and 1. If 0 and 1 behave, then argue the general case.

A nudge: addition version is "add x to itself x times." Try x = 0. How many times do you add? Does your loop still stop? That single test is where most wrong answers die.

STICK IT IN YOUR BRAIN

I O D F E. Input, Output, Definiteness, Finiteness, Effectiveness. Zero or more in, one or more out, every step clear, it must stop, every step must be doable.

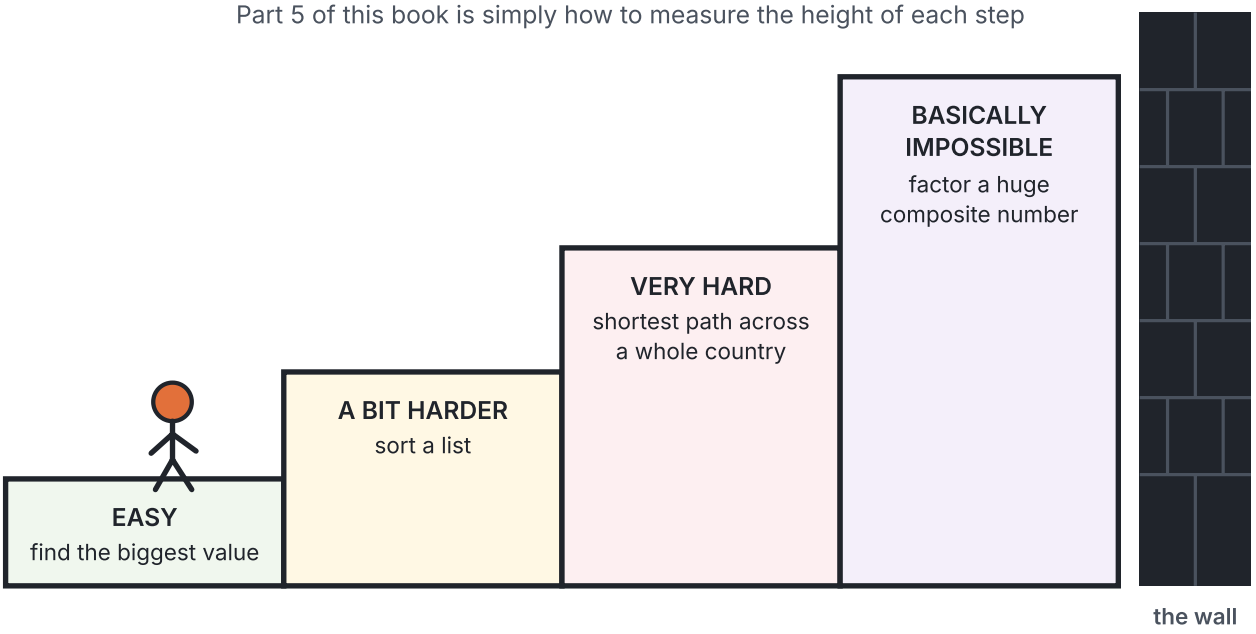
3 Easy, hard, and do not even try

Why some problems cost far more to solve than others.

Not all problems cost the same to solve. They can be arranged on a ladder, and it is worth seeing that ladder early, because the rest of the course is about measuring the height of each step.

The cost of a problem is a step you have to climb

Part 5 of this book is simply how to measure the height of each step



The ladder. The hardness of a problem is not about your skill. It is a property of the problem itself. Factoring a 600-digit number is hard for you, for me, and for every computer on earth. That is why online banking still works.

Easy

Finding the largest or smallest value in a list. Finding one specific value in a list. You look once at each item and you are done.

A bit harder

Sorting a list. Now the items have to be compared against each other, not just looked at.

Very hard

Finding the shortest path between two far-apart cities when there are thousands of possible roads.

Essentially impossible

Factoring large composite numbers. No known method finishes in reasonable time. Modern cryptography is built on exactly this.

REAL LIFE

Finding the tallest person in your classroom is easy. You scan the room once. Lining up the whole class by height is harder, because now you keep comparing people with each other. Finding the cheapest way to visit twelve towns and come back home is the kind of problem that will eat a laptop alive, and that is the famous travelling salesman problem you will meet again in chapter 18.

STICK IT IN YOUR BRAIN

Hardness belongs to the **problem**, not to the programmer. The whole point of algorithm analysis is to put a number on that hardness before you waste a week coding.

PART 2

The things algorithms touch

An algorithm on its own is empty. It needs stuff to work on, and that stuff has to be organised.

4 Data, names and boxes

Variables, addresses, and the assignment operation.

Algorithms manipulate objects, called data or values. Some are taken in as input and some are produced as output. Simple so far.

But data cannot just lie around loose. It has to be **constructed**, meaning organised in a way that can actually be used. Data constructs let you create different kinds of data and encode them so an algorithm can work on them.

Those constructs match mathematical ideas you already met. Cartesian product, which is pairing things up. Disjoint union, which is "this kind or that kind, not both." Functions, which map one thing to another. In practice, these give you **data structures**, which are just relationships between data values.

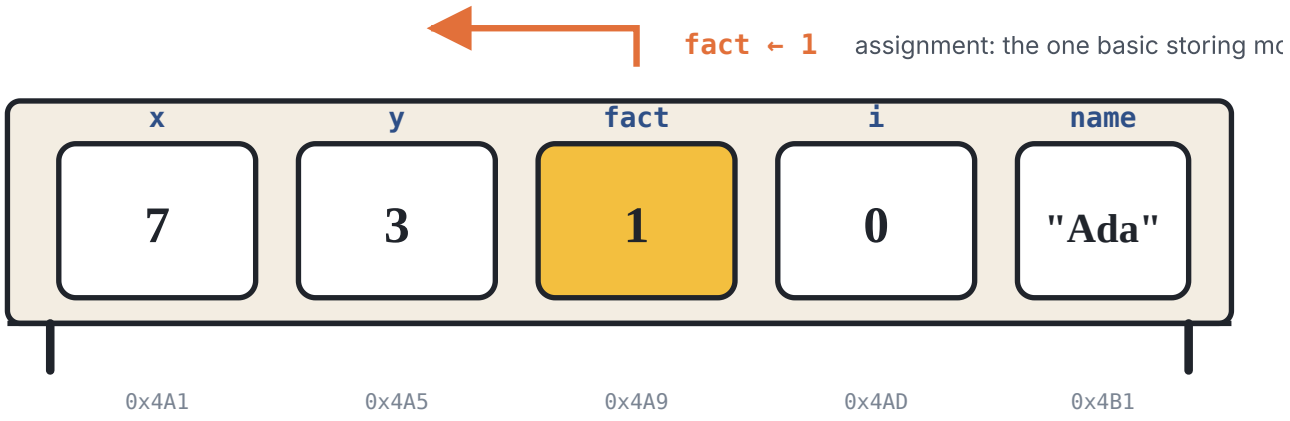
Every value lives somewhere

Data is stored at a location, also called an address. Addresses are numbers, and numbers are miserable to remember, so we give them names. The name is the variable.

And the basic operation that puts data into a location is **assignment**. It is written with a left arrow:

```
R ← L      // put whatever is in L into the box called R
fact ← 1    // put the number 1 into the box called fact
```

Memory is a shelf. A variable is the label, not the box.



the real addresses underneath, which nobody wants to type

The label can move to another box. The value inside can be replaced. They are two different things.

Names are placeholders. In the equation $x^2 + 2y - 2 = 1$, the letters x and y are not the data. They are the names holding the data. Same thing in code.

REAL LIFE

Think of the lockers at a hostel. Each locker has a number stamped on it, which nobody uses in conversation. People say "Ricky's locker." Ricky is the variable name. The stamped number is the address. The contents are the value.

Three separate things, and they can change independently. New contents in the same locker. Same person moved to a different locker. The label and the box are not the same thing, and this is the single idea that makes pointers easy later.

STICK IT IN YOUR BRAIN

Values live at **addresses**. We name addresses so humans can cope, and those names are **variables**. The basic operation for storing a value is **assignment**, written as a left arrow in pseudocode.

5 Types: what you are allowed to do with a thing

A type is a set of values together with the operations allowed on them.

Here is the definition, and it is better than it looks. A **data type** is a set of values together with a set of permissible operations on them. The type decides what actions can be applied.

Numbers allow arithmetic. You can add two integers. Eggs allow whisking, by a baker. You cannot whisk a brick and you cannot add two eggs and expect 2. The operations come attached to the kind of thing.

The thing comes with its own list of legal moves



System-defined (primitive)

Come free with the language.

`int` `float` `char` `double` `bool`

User-defined

You build them out of the primitives.

```
struct Student { int id; char name[40]; }
```

Two families of type. Primitive types are handed to you by the language. User-defined types are the ones you assemble yourself. The practical point: a data type **reduces the coding effort**, because the legal operations are decided once and reused everywhere.

REAL LIFE

Go to the market. Tomatoes are sold by the bucket, cloth is sold by the yard, and credit for your phone is sold by the amount. Each one has its own legal operations. You can halve a bucket of tomatoes. You can cut cloth. You cannot cut 500 francs of airtime in half with a knife.

Nobody ever taught you that rule. The thing carries its own rules, and a data type is exactly that idea written formally.

STICK IT IN YOUR BRAIN

A type is **a set of values plus the operations allowed on them**. Two families: system-defined (primitive) like int, float, char, double, bool, and user-defined, which you build.

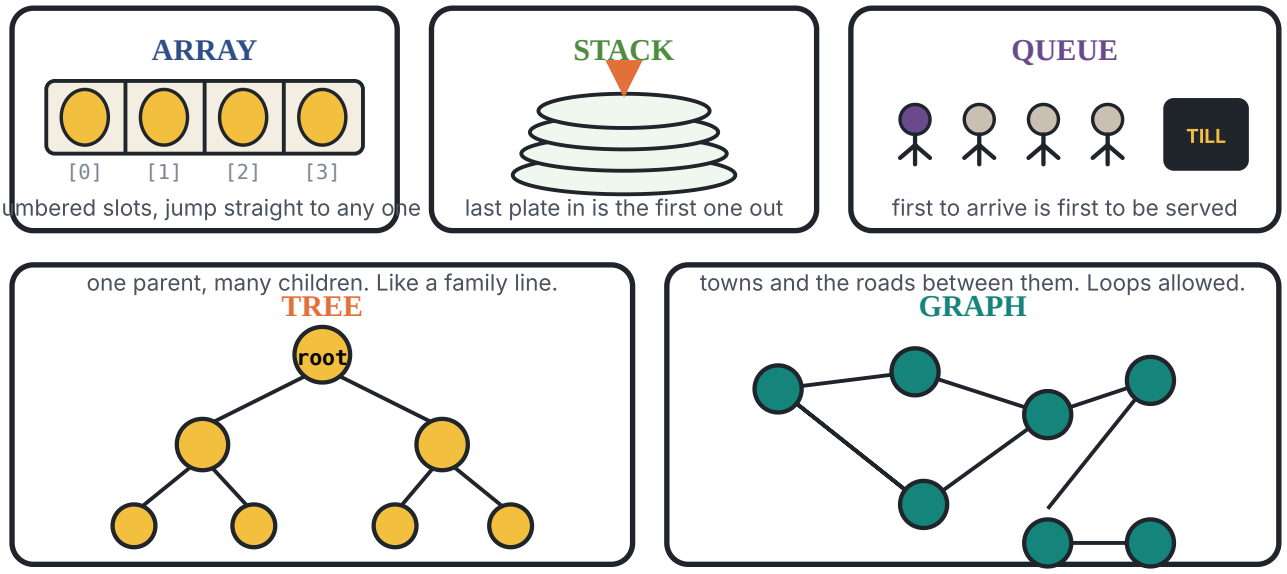
6 Data structures: picking the right container

Arrays, stacks, queues, trees and graphs, and when each one fits.

A **data structure** is a particular way of storing and organising data in a computer so it can be used efficiently. That last word carries the weight. Any structure can hold the data. Only the right one makes the algorithm fast.

The common ones are arrays, files, linked lists, stacks, queues, trees and graphs. Here is what each one is for.

Five containers you already use every day



Choose wrong and a one-second job becomes a one-hour job. Nothing else changed.

Also on the list: files, and linked lists (an array where each item points at the next one instead of sitting in numbered slots, so inserting in the middle is cheap)

The container decides the cost. Finding the 400th item is instant in an array and painful in a linked list. Taking the oldest item is instant in a queue and awkward in a stack. Same data either way.

REAL LIFE

A **stack** is the pile of plates after a party. You wash the top one first, which is the last one that was put down. A **queue** is the line at the bank. First person in is first person served. Try running a bank like a stack and see what happens to you.

A **tree** is your family, going down. A **graph** is the road network around Buea, where many roads meet many roads and you can go round in a circle.

STICK IT IN YOUR BRAIN

A data structure organises data so it can be used **efficiently**. Array, file, linked list, stack, queue, tree, graph. Stack is last in first out. Queue is first in first out.

7 Algorithm vs program vs programming language

Three words that are often confused, and the three layers inside a language.

Three words that get mixed up constantly. The distinction between them is clean:

Word	What it is	Everyday twin
Algorithm	Step-by-step unambiguous instructions to solve a problem	The plan in your head
Program	An algorithm executable on a computer	The plan written so a machine obeys it
Programming language	A collection of primitives (basic actions, control and data constructs), the rules for combining them, and the meanings attached	The language you wrote it in

The three words inside a language

Every language, including English, has these three layers. Learn them here and you will never confuse a syntax error with a logic error again.

Syntax, semantics, pragmatics

SYNTAX

The symbols, and the rules for combining them. The gra

```
if (x > 0) {
// brackets, semicolons, order
```

SEMANTICS

The meaning. What the thing actually does when it runs

"only run this when x is bigger than zero"

PRAGMATICS

Options and techniques for using the language well and

"the clean way to bind variables in a compiler"

Correct syntax with wrong semantics compiles perfectly and gives you the wrong answer.

Syntax, semantics, pragmatics. For a program, the basic actions (primitives) are the instructions the computer or processor recognises. Everything above that is built by combining them.

REAL LIFE

"The goat ate the man" is perfect English. Syntax, clean. Semantics, suspicious. The sentence is grammatically correct and factually mad.

That is exactly a logic bug. The compiler is a grammar teacher, not a meaning teacher. It will pass your sentence and let you fail your own test.

STICK IT IN YOUR BRAIN

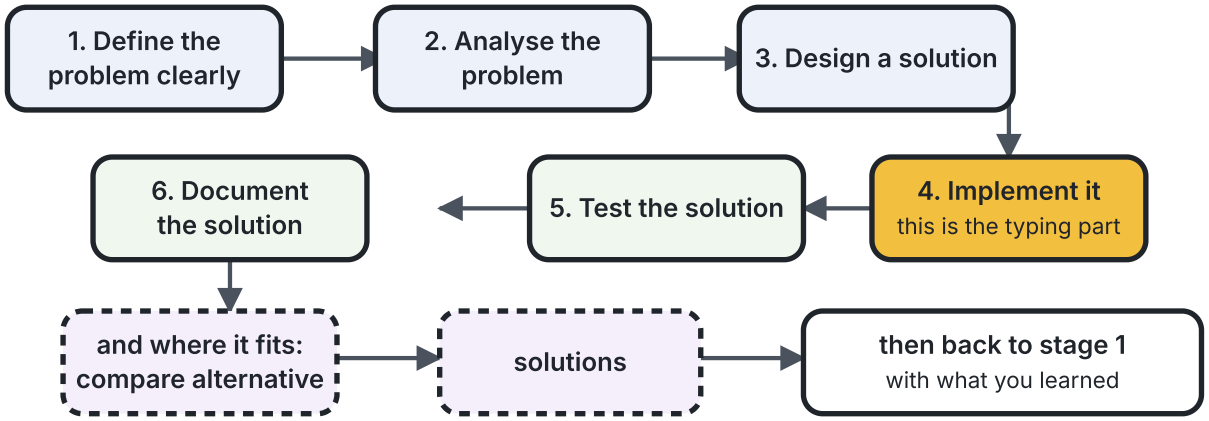
Syntax is form. **Semantics** is meaning. **Pragmatics** is how to use the language well. A compiler checks the first one only.

8 What programming actually is

The six stages of the programming process, only one of which is coding.

Programming is the total creative process, not simply writing code. The code is one stage out of six, and the stages that surround it are where projects live or die.

Only one of these six stages involves a keyboard



Skip stage 1 and you will build a beautiful solution to the wrong problem.

Note the gap between stage 3 and stage 4. Design belongs on paper, where mistakes are free.

That gap is what pseudocode and flowcharts are for, and that is Part 3 of this book.

The six stages. Define, analyse, design, implement, test, document. Plus, where circumstances allow, compare alternative solutions.

REAL LIFE

Nobody builds a house by buying cement first. You find out how many people will live in it, you check the land, you draw a plan, you build, you check the roof does not leak, and you keep the plan somewhere safe for whoever repairs it in ten years.

Stage 6, documentation, is the one everyone skips. It is the plan you leave behind. Future you, at 2am, debugging your own project, is a different person who remembers nothing.

STICK IT IN YOUR BRAIN

Programming is **define, analyse, design, implement, test, document**, and where possible compare alternative solutions. Coding is stage four of six.

PART 3

Writing it down

You have an idea in your head. Now get it onto paper in a form that another human, and later a machine, can follow without guessing.

9 Pseudocode: English wearing a tie

Writing an algorithm in a form that sits between English and code.

The same algorithm can be written in many notations. When you are still discovering the solution, you need a notation that lets you be a bit vague where you are still unsure. The three common ones are **pseudocode**, **flowcharts** and **dataflow diagrams**.

Pseudocode sits between natural language and a real programming language. It lets you think in English while structuring your sentences the way your target language will want them. It is English with some rules of structure and some keywords that make it look a little like code.

One idea, three levels of tie-tightness

PLAIN ENGLISH

Keep multiplying
the numbers from
1 up to n together
and show the result.

anyone understands it
no machine can run it

→

PSEUDOCODE

```
BEGIN
  fact ← 1
  read n
  FOR i = 1 TO n
    fact ← fact × i
  ENDFOR
  display fact
END
```

structure without syntax pain

→

REAL C

```
int n, fact=1, i;
scanf("%d",&n);
for(i=1;i<=n;i++)
  fact=fact*i;
printf("%d",fact);
```

the machine runs this
one missing ; and nothing works

Thinking happens in the middle box. Skip it and you debug your thinking in C, which is torture.

Dataflow diagrams are the third notation: they show how data moves rather than the order of steps.

Why the middle box exists. Pseudocode is not laziness. It is where you catch a logic mistake while it still costs nothing to fix.

The keyword list your course uses

These are the keywords used throughout this course. Write them in capitals. They come in pairs, and the pairs are not optional.

For this	Use these
Start and finish	`BEGIN MAINPROGRAM` … `END MAINPROGRAM`
Initialisation	`INITIALISATION` … `END INITIALISATION`
Subprogram	`BEGIN SUBPROGRAM` … `END SUBPROGRAM`
Selection	`IF`, `THEN`, `ELSE`, `ENDIF`
Multi-way selection	`CASEWHERE`, `OTHERWISE`, `ENDCASE`
Pre-test repetition	`WHILE` … `ENDWHILE`
Post-test repetition	`REPEAT` … `UNTIL`

And four house rules that go with them:

- Keywords are written in **capitals**.
- Structural elements come in **pairs**. Every BEGIN has an END. Every IF has an ENDIF.
- **Indenting** is used to show structure.
- Names of subprograms are **underlined**. That underline is a promise that you will come back and write that subprogram later. It is the mechanism that makes top-down development work.

A full worked algorithm: searching a list

The algorithm below is written out step by step. It is a **linear search**. Keep it in mind, because it comes back in chapter 22 when we talk about best and worst case.

```
1. Start
2. Accept size for an array        // read size
3. Accept array element values from the user
4. Accept the element to be searched   // read Value
5. Set i = 0, flag = 0
6. Compare A[i] with Value
     IF (A[i] is the value)
        Set flag = 1, go to step 8
     ELSE
        move to the next element, i = i + 1
7. IF (i <= n) go to step 6
8. IF (flag = 1) THEN
        print "found" and return i as the position
     ELSE
        print "not found"
9. Stop
```

Look at what `flag` is doing. It is a yes-or-no memory. It starts at 0 meaning "I have not found it." The moment the item is found it becomes 1. At step 8 you no longer need to remember anything about the search, you just ask the flag. That little trick appears everywhere in programming.

REAL LIFE

You lost your phone in the house. You check the bed, then the table, then the chair, then the kitchen. One place at a time, in order, until you find it or you run out of places. That is linear search, and you have run it a hundred times.

The flag is you shouting "found it!" so you stop checking.

Good about pseudocode

Easy to understand even for a complex program. It does not follow programming language syntax. Real programs can be generated from it easily. You grasp the logic fast. It is easy to modify.

Bad about pseudocode

It cannot be compiled or interpreted, so errors cannot be caught by a machine. And because it can be written in any style, the flow of a program can become hard to follow.

STICK IT IN YOUR BRAIN

Pseudocode is **a mix of natural language and language constructs**. Keywords in capitals, structural elements in pairs, indenting shows structure, subprogram names underlined. Its big weakness: no compiler means no error checking.

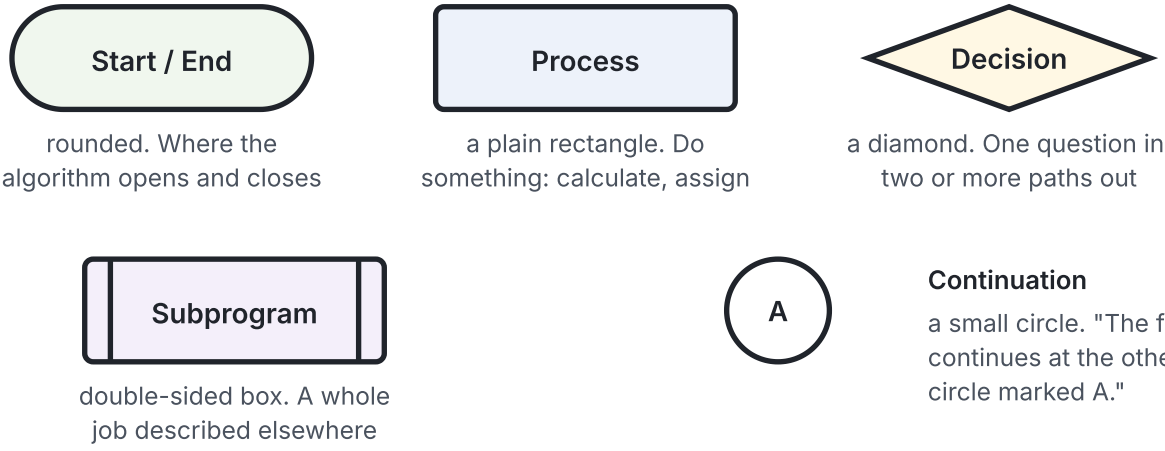
10 Flowcharts: the picture version

The five shapes, and how to read a flowchart.

Flowcharts are a diagrammatic method of representing algorithms. Operations go in boxes, and lines with arrowheads show the flow of control. The standard direction of flow is **top to bottom and left to right**, so you only draw an arrow pointing up or left when something genuinely loops back.

The alphabet is small. These are the box types.

The complete flowchart alphabet



Start / End rounded. Where the algorithm opens and closes

Process a plain rectangle. Do something: calculate, assign

Decision a diamond. One question in, two or more paths out

Subprogram double-sided box. A whole job described elsewhere

Continuation a small circle. "The flow continues at the other circle marked A."

Main flow runs top to bottom, left to right. Arrows going up mean a loop.

Lines carry the arrowheads. The shape tells you what kind of step it is before you read a word.

Five shapes, no more. The reason flowcharts work is that your eye reads the shape faster than it reads the text. A diamond means "a decision happens here" before you have read the question inside it.

WHERE STUDENTS LOSE MARKS

Using a rectangle for a decision, or forgetting to label the branches coming out of a diamond. If a diamond has two lines leaving it and neither says yes or no, the chart is incomplete.

STICK IT IN YOUR BRAIN

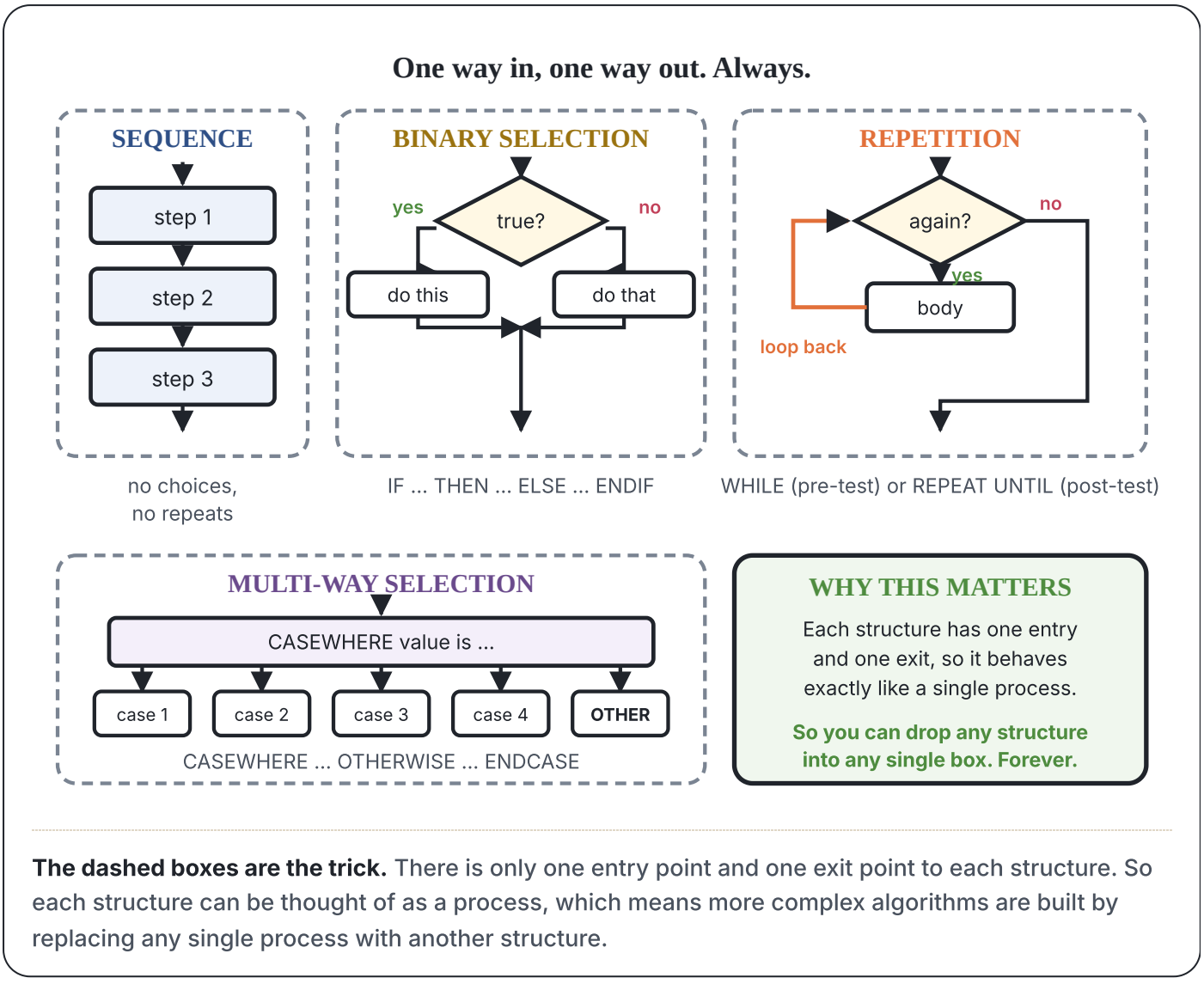
Oval for start and end. Rectangle for process. Diamond for decision. Double-sided rectangle for subprogram. Circle for continuation. Flow is top to bottom, left to right.

11 The only three shapes you need

Sequence, selection and repetition, and why they nest without limit.

This is the most important idea in Part 3. Every algorithm ever written, of any size, is built from three control structures plus one packaging trick.

1. **Sequence**. Do this, then this, then this.
2. **Selection**. Choose a path. Two-way (binary) or multi-way.
3. **Repetition**. Do it again while something is still true.
4. **Subprograms**. Take a chunk of the above, give it a name, and use the name.



The dashed boxes are the trick. There is only one entry point and one exit point to each structure. So each structure can be thought of as a process, which means more complex algorithms are built by replacing any single process with another structure.

REAL LIFE

Making breakfast. **Sequence**: boil water, make tea, drink. **Selection**: if there is bread, eat bread, else drink only tea. **Repetition**: keep stirring until the sugar dissolves. **Subprogram**: "make tea" is a whole set of steps you refer to by name without listing them every time.

You do not think about these categories while cooking. But every instruction you could ever give another person fits in one of them.

Pre-test and post-test loops are not the same

This catches people. `WHILE` tests **before** running the body, so the body can run zero times. `REPEAT UNTIL` tests **after**, so the body always runs at least once.

REAL LIFE

WHILE there is water in the bucket, fetch a cup. If the bucket is already empty you fetch nothing. Zero cups.

REPEAT fetch a cup UNTIL the bucket is empty. You dip in first, then check. If the bucket was already empty you have still reached in once and come out wet.

STICK IT IN YOUR BRAIN

Sequence, selection, repetition, subprograms. One entry, one exit each, which is why they nest without limit. WHILE can run zero times, REPEAT UNTIL always runs at least once.

12 Tea, traffic lights and a toll gate

Four worked examples, from a simple sequence to a full control problem.

1. Sequence: make a pot of tea

Problem: write a set of instructions that describe how to make a pot of tea.

```
BEGIN
   fill a kettle with water
   boil the water in the kettle
   put the tea leaves in the pot
   pour boiling water in the pot
END
```

No decisions, no repeats. Pure sequence. Notice that the order is the entire algorithm. Swap steps two and four and you have leaves floating in cold water.

2. Binary selection: approaching a traffic light

Problem: write a set of instructions to follow when approaching a set of traffic control lights.

```
IF the signal is green THEN
   proceed through the intersection
ELSE
   stop the vehicle
ENDIF
```

Two outcomes only. Green, or not green. Everything that is not green is treated the same way.

3. Multi-way selection: responding to all the signals

Problem: write instructions that describe how to respond to *all* possible signals at a set of traffic lights. Now "not green" is not good enough, because amber and red are different situations, and a broken light is a third.

```
CASEWHERE signal is
   red    : stop the vehicle
   amber  : stop the vehicle
   green  : proceed through the intersection
   OTHERWISE : proceed with caution
ENDCASE
```

NOTICE THE OTHERWISE

That last line is the professional habit. It catches the case where the light is off, flashing, or showing something nobody planned for. Every CASEWHERE needs one. In real code it is the difference between a program that survives surprises and one that crashes.

4. The toll gate, which uses everything at once

Here is the problem as stated. Read it twice before looking at the solution.

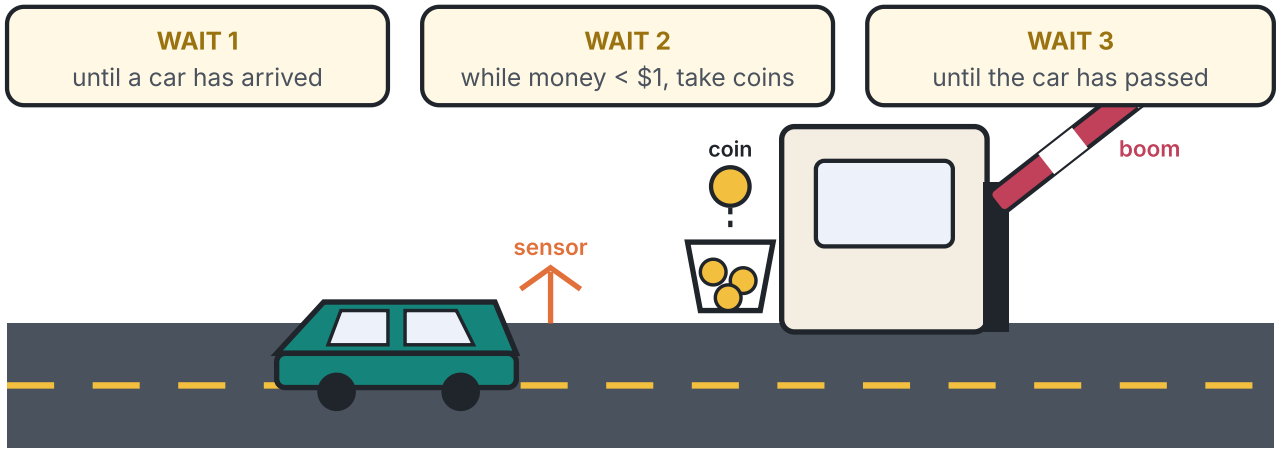
THE PROBLEM

When operational, a toll gate operates by having a boom gate obstructing the road, and a sensor detecting when a vehicle is present. After coins to the value of $1.00 have been deposited in the basket, the boom gate opens and stays open until a vehicle has gone through.

Amounts greater than $1.00 are accepted but no change is given. Individual coins less than 10 cents are ignored.

Write an algorithm to describe the control of the toll gate.

Three waits, one inside the other



And the whole thing sits inside a fourth loop: REPEAT all of that UNTIL the gate is switched off.

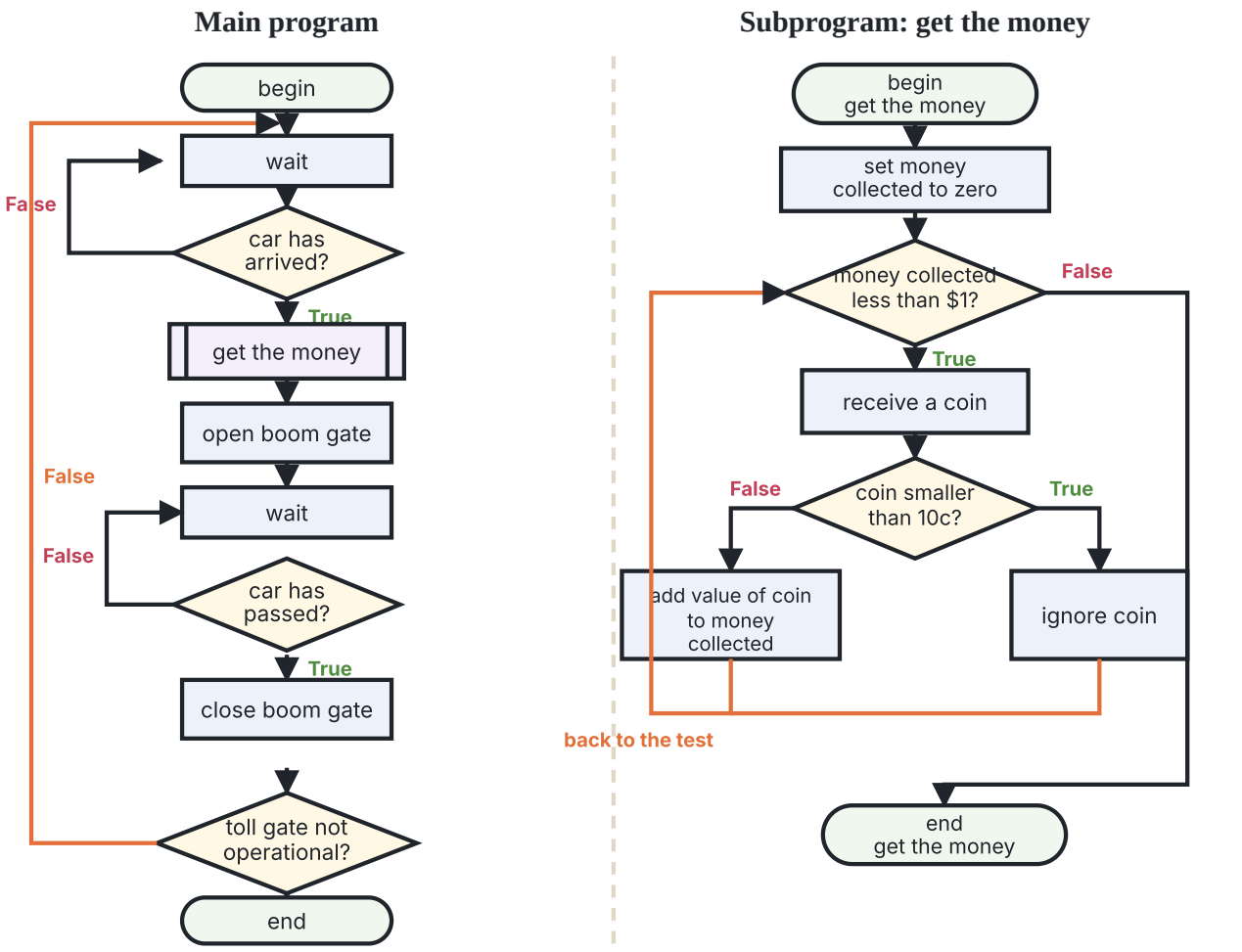
Why this problem is a good exam question. It forces you to notice that a toll gate does nothing but wait, three times over, forever. Spot the waits and the pseudocode writes itself.

Here is the solution. Read the main program first, then the subprogram.

```
BEGIN MAINPROGRAM
   REPEAT
      REPEAT
         wait
      UNTIL car has arrived
      get the money              // underlined: a subprogram, written below
      open boom gate
      REPEAT
         wait
      UNTIL car has passed
      close boom gate
   UNTIL toll gate is not operational
END MAINPROGRAM
```

```
BEGIN SUBPROGRAM get the money
   INITIALISATION
      money collected is set to 0
   END INITIALISATION
   WHILE money collected is less than $1
      receive coin
      IF coin is less than 10 cents THEN
         ignore coin
      ELSE
         add the value of the coin to the money collected
      ENDIF
   ENDWHILE
END SUBPROGRAM get the money
```

The same algorithm drawn as a flowchart. Trace the "False" arrows with your finger. Every one of them goes back up, and that is what a wait looks like in a picture.



Every orange arrow is a loop. Every one of them points back up the page.

That is how you spot repetition in any flowchart, before you have read a single box.

The same algorithm, drawn. The main program on the left, the subprogram on the right. The double-sided box in the main program is where the subprogram gets called, which is exactly what the underline did in the pseudocode.

Three things worth noticing

- **"No change is given" needed no code.** The loop condition is "less than $1", so once you pass $1 it stops and never thinks about the extra. The requirement was handled by choosing the right comparison. This is what good design looks like.
- **"Ignore coins under 10 cents" is a selection inside a repetition.** That is chapter 11 nesting, used for real.
- **The underline is doing work.** The main program stays readable because "get the money" is one line. The messy coin-counting detail lives somewhere else. That is top-down design, which is the next chapter.

STICK IT IN YOUR BRAIN

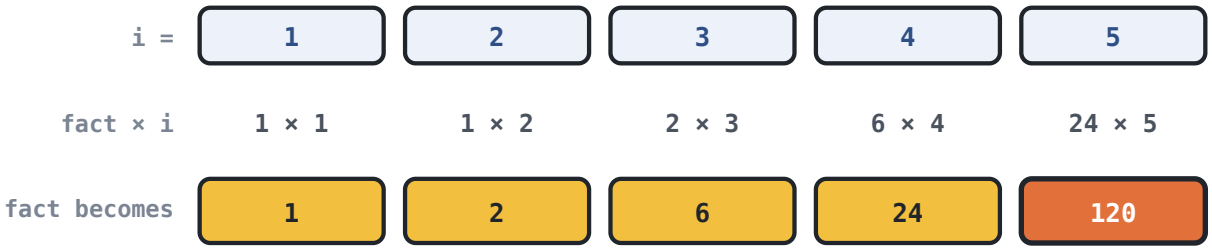
A real control problem is usually **a loop of loops**. Find every point where the machine must wait, give each wait its own REPEAT UNTIL, and wrap everything in a REPEAT UNTIL the machine is switched off.

13 Factorial, from idea to running C

One problem written three ways: numbered steps, pseudocode, then C.

Factorial of n, written n!, means multiply every whole number from 1 up to n. So 5! is 1 × 2 × 3 × 4 × 5, which is 120.

One accumulator, multiplied again and again



fact starts at 1, not 0. Start it at 0 and every multiplication gives 0 forever.

By n = 20 the answer has 19 digits. Factorial grows faster than almost anything you will meet.

The accumulator pattern. One variable holds the running answer and gets updated each time round the loop. Initialising it correctly decides whether the answer is right at all: 1 for multiplying, 0 for adding.

Step 1: the algorithm in numbered steps

```
Step 1: start
Step 2: initialize fact = 1
Step 3: input from the user value n
Step 4: for i = 1 to i <= n repeat the process
Step 5: fact = fact * i
Step 6: i++                 // increment i by one
Step 7: print fact value
Step 8: stop
```

Step 2: the same thing as pseudocode

```
Start program
   Declare fact and n
   Enter number for n
   for i = 1 to i <= n
      Perform fact = fact * i
   Display fact
End program
```

Step 3: the real C program

```c
#include <stdio.h>

void main() {
   int n, fact=1, i;
   printf("enter value for n");
   scanf("%d", &n);
   for(i=1; i<=n; i++) {
      fact = fact * i;
   }
   printf("\n factorial of %d is %d", n, fact);
}
```

REAL LIFE

You are counting the day's takings at a shop. You start with an empty total, which is zero, and you add each note as you pick it up. That is the accumulator pattern with addition.

Now do it with multiplication instead and you must start at one, because starting at zero means your total stays zero no matter how much money is in the bag. Same pattern, different starting value, and getting it wrong silently destroys the answer.

TRY THIS

Run the C program in your head with n = 0. What does it print? Is that the mathematically correct answer for 0! ? Trace the loop condition carefully: how many times does the body execute?

STICK IT IN YOUR BRAIN

Idea, then pseudocode, then code. Never idea straight to code. And for any **accumulator**, initialise to 1 for products and 0 for sums.

14 Euclid's algorithm, the oldest trick in the book

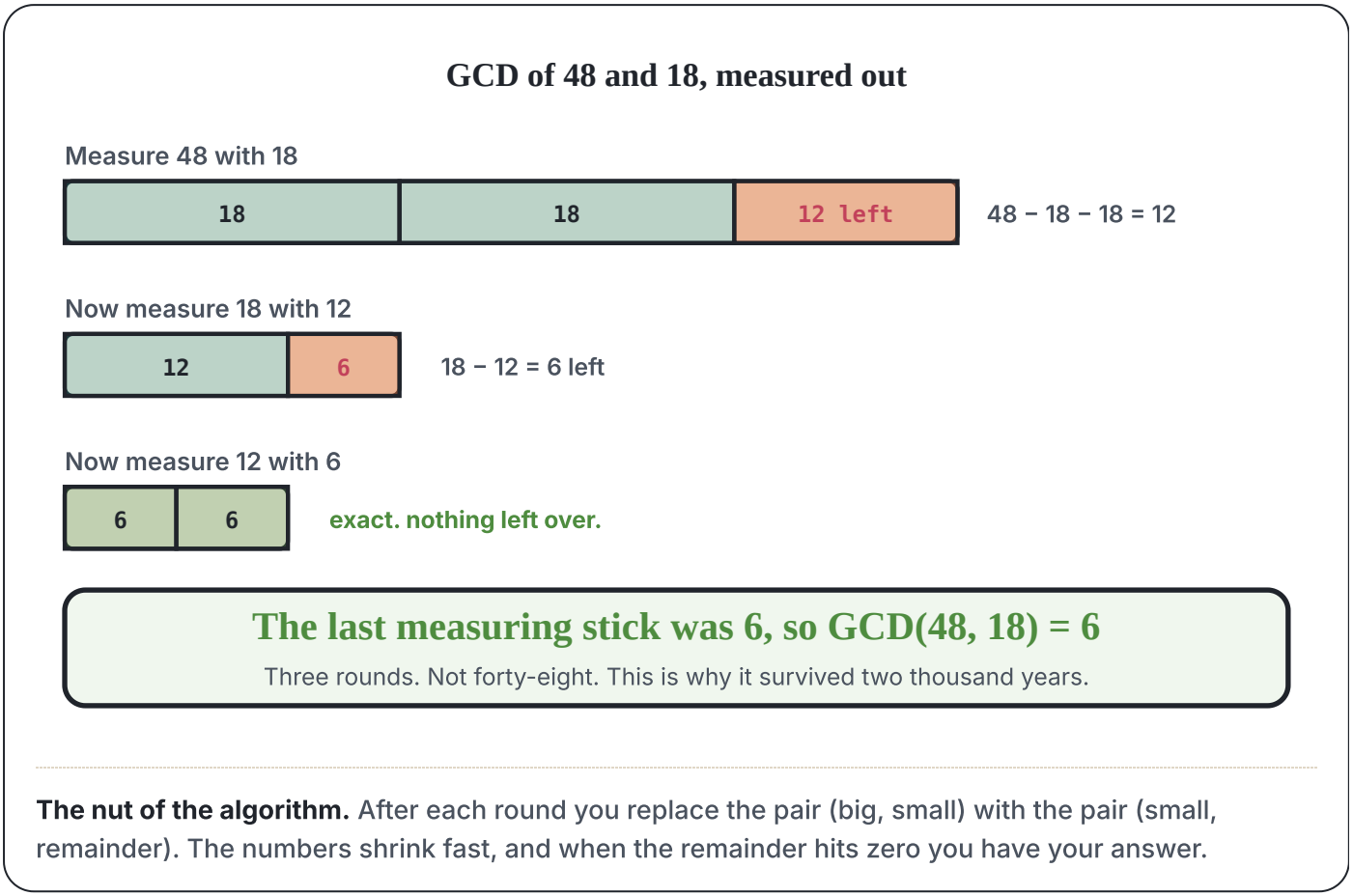
Finding the greatest common divisor by repeated measurement.

Euclid's algorithm finds the **greatest common divisor** of two numbers. The GCD is the largest number that divides both of them with no remainder. It is one of the oldest algorithms still in common use, and it still shows up in ordinary places: reducing fractions to simplest form, and inside number theory and cryptography.

First, forget the numbers. Think about measuring.

Euclid was a geometer, so he thought about lengths, not division. Suppose you have a long plank and a short stick, and you want the longest stick that measures both of them exactly.

You lay the short stick against the plank again and again, cutting off its length each time, until what remains is shorter than the stick. That leftover is the remainder. Now here is the magic move: you throw away the original plank and start again, measuring the *old stick* with the *leftover*. Keep doing this. When a measurement comes out exact, with nothing left over, the stick you were measuring with is your answer.



The nut of the algorithm. After each round you replace the pair (big, small) with the pair (small, remainder). The numbers shrink fast, and when the remainder hits zero you have your answer.

The algorithm, step by step

The pseudocode uses three storage locations: L, S and R. L and S hold the two input lengths, R holds the remaining length. The comments in square brackets explain what each stage is doing.

```
1  INPUT L, S          [put the two lengths into locations L and S]
2  R ← L               [make remaining length r equal to the input length l]

E0: [Ensure r ≥ s]
3  IF R > S THEN GOTO step 7
   ELSE swap the contents of R and S
4  L ← R               [redundant, but useful for later discussion]
5  R ← S
6  S ← L

E1: [Find remainder] Until r in R is less than s in S,
    repeatedly subtract s in S from r in R.
7  IF S > R THEN done measuring, GOTO 10 ELSE measure again
8  R ← R − S
9  GOTO 7              [remainder loop]

E2: [Is the remainder zero?] Either the last measure was exact and R
    holds zero, so we can halt. Or we continue.
10 IF R = 0 THEN GOTO step 15 ELSE continue to step 11

E3: [Interchange s and r] The nut of Euclid's algorithm. Use remainder r
    to measure what was previously the smaller number s. L is a temp.
11 L ← R
12 R ← S
13 S ← L
14 GOTO 7              [repeat the measuring process]

OUTPUT:
15 PRINT S             [S contains the greatest common divisor]
16 HALT, END, STOP
```

WHY SWAPPING NEEDS THREE STEPS

Steps 11, 12 and 13 look clumsy but they are the correct way to swap two values. To exchange what is in two cups you need a third empty cup. L is that third cup.

Try it with two steps: `R ← S` then `S ← R`. The first line already destroyed what was in R, so the second line copies S back onto itself. Both locations now hold the same thing and one value is gone forever. Write that out by hand once and you will never forget it.

The same algorithm in BASIC

Here `LET` is the assignment instruction symbolised by the left arrow.

```
5  REM Euclid's algorithm for greatest common divisor
6  PRINT "Type two integers greater than 0"
10 INPUT A,B
20 IF B=0 THEN GOTO 80
30 IF A > B THEN GOTO 60
40 LET B=B-A
50 GOTO 20
60 LET A=A-B
70 GOTO 20
80 PRINT A
90 END
```

And in C

```c
// Euclid's algorithm for greatest common divisor
int euclidAlgorithm (int A, int B) {
   A = abs(A);
   B = abs(B);
   while (B != 0) {
      while (A > B) {
         A = A-B;
      }
      B = B-A;
   }
   return A;
}
```

Three notations, one algorithm. That is the point of chapter 9 proven in front of you: the idea is the same, only the clothing changes.

REAL LIFE

You have 48 oranges and 18 mangoes and you want to make identical gift bags with no fruit left over and the biggest possible bags. How many bags? Six. Each bag gets 8 oranges and 3 mangoes.

You did not need algebra. You needed GCD(48, 18), and Euclid handed it to you in three rounds.

STICK IT IN YOUR BRAIN

Euclid replaces the pair **(bigger, smaller)** with **(smaller, remainder)**, over and over, until the remainder is zero. The last non-zero value is the GCD. Swapping two values always needs a third temporary location.

PART 4

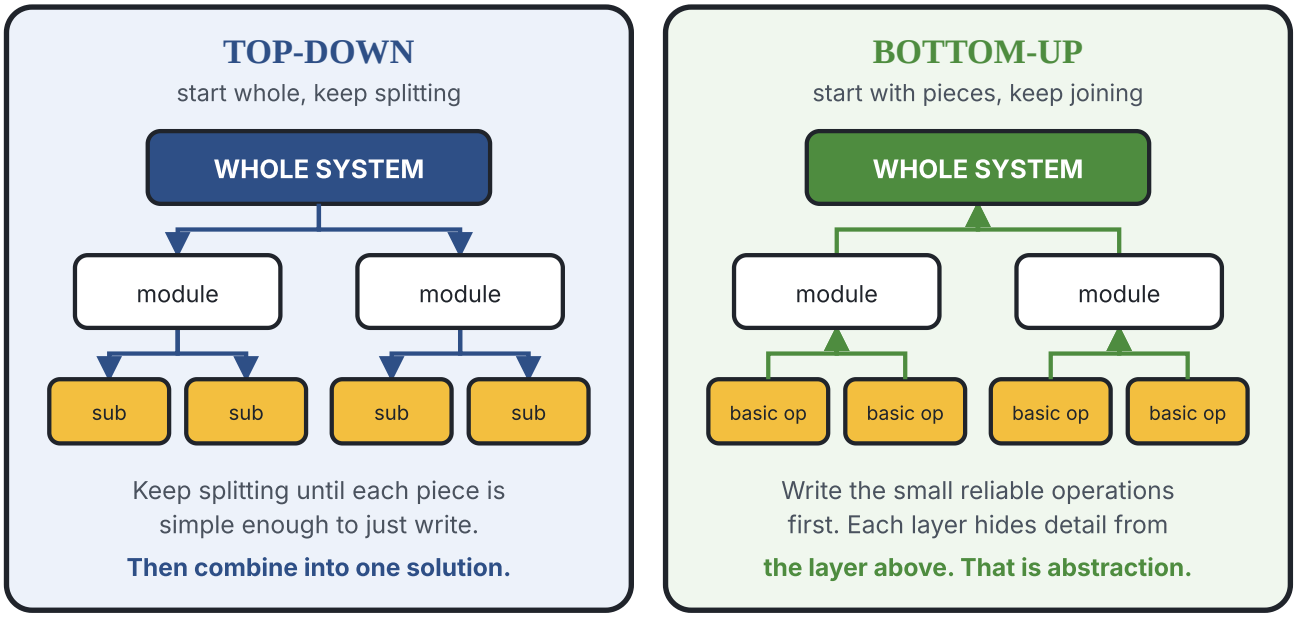
How to attack a problem

You understand the problem. Now, what strategy do you reach for? These are the standard weapons, and each one has a shape of problem it was built for.

15 Top-down and bottom-up

Two directions of design, and when each suits the problem.

Two opposite directions of attack, and both are correct. They just suit different situations.



Same destination, opposite journeys. Top-down identifies major components and decomposes them until each module is simple. Bottom-up designs the most primitive components first and works up, each layer providing abstraction for the next.

REAL LIFE

Top-down is how you plan a wedding. Ceremony, reception, transport. Then reception splits into food, drinks, music, seating. Then food splits into caterer, menu, plates. You never start a wedding by deciding the brand of spoon.

Bottom-up is how a cook learns. First you learn to chop. Then to fry. Then to season. One day those combine into a dish you never studied, because you have the pieces.

WHERE YOU ALREADY SAW THIS

Chapter 12. The toll gate pseudocode had get the money underlined in the main program, with the details written separately. That underline is top-down design in action. The main program stays readable because the detail was pushed down a level.

STICK IT IN YOUR BRAIN

Top-down splits a big problem into subtasks, then subtasks into smaller subtasks, then combines them into a single solution. **Bottom-up** builds basic operations first, combines them into modules, and combines modules into the system.

16 Greedy: grab the best thing in front of you

Taking the best option at each stage, and the cases where that fails.

A greedy algorithm works in stages. At each stage it makes the decision that looks best *right now*, without worrying about future consequences. It picks the local best and assumes the local best adds up to the global best.

Sometimes that assumption is true and provable. Sometimes it is a disaster. Knowing which is which is the entire skill.

The advantage

It is straightforward, easy to understand and easy to code. Once a decision is made you never go back and re-examine it, so you waste no time on already-computed values.

The disadvantage

For many problems there simply is no greedy algorithm. In many cases there is no guarantee that making locally optimal choices gives the global optimum.

Where greedy genuinely works

These are the standard applications. Each has a proof behind it that the local choice is safe.

- **Sorting:** selection sort, topological sort
- **Priority queues:** heap sort
- **Huffman coding** compression
- **Prim's and Kruskal's** algorithms (minimum spanning trees)
- **Dijkstra's** shortest path in a weighted graph
- **Coin change** problem
- **Fractional knapsack** problem
- **Disjoint sets:** union by size, union by height or rank
- **Job scheduling**
- And greedy methods are often used as **approximation algorithms** for complex problems

REAL LIFE

Giving change. Someone owes you 1,750 francs. You hand over the 1,000 first, then the 500, then the 200, then the 50. Biggest note that still fits, every time, never reconsidering. That is greedy, and with normal currency it gives the smallest number of notes.

But change the currency and it breaks. Suppose coins of 1, 7 and 10 and you must pay 15. Greedy takes 10, then 1, 1, 1, 1, 1, which is six coins. The right answer is 7 + 7 + 1, which is three coins. Greedy did not lie to you. It just answered a different question: "what looks best now."

A problem worth testing, and the honest answer

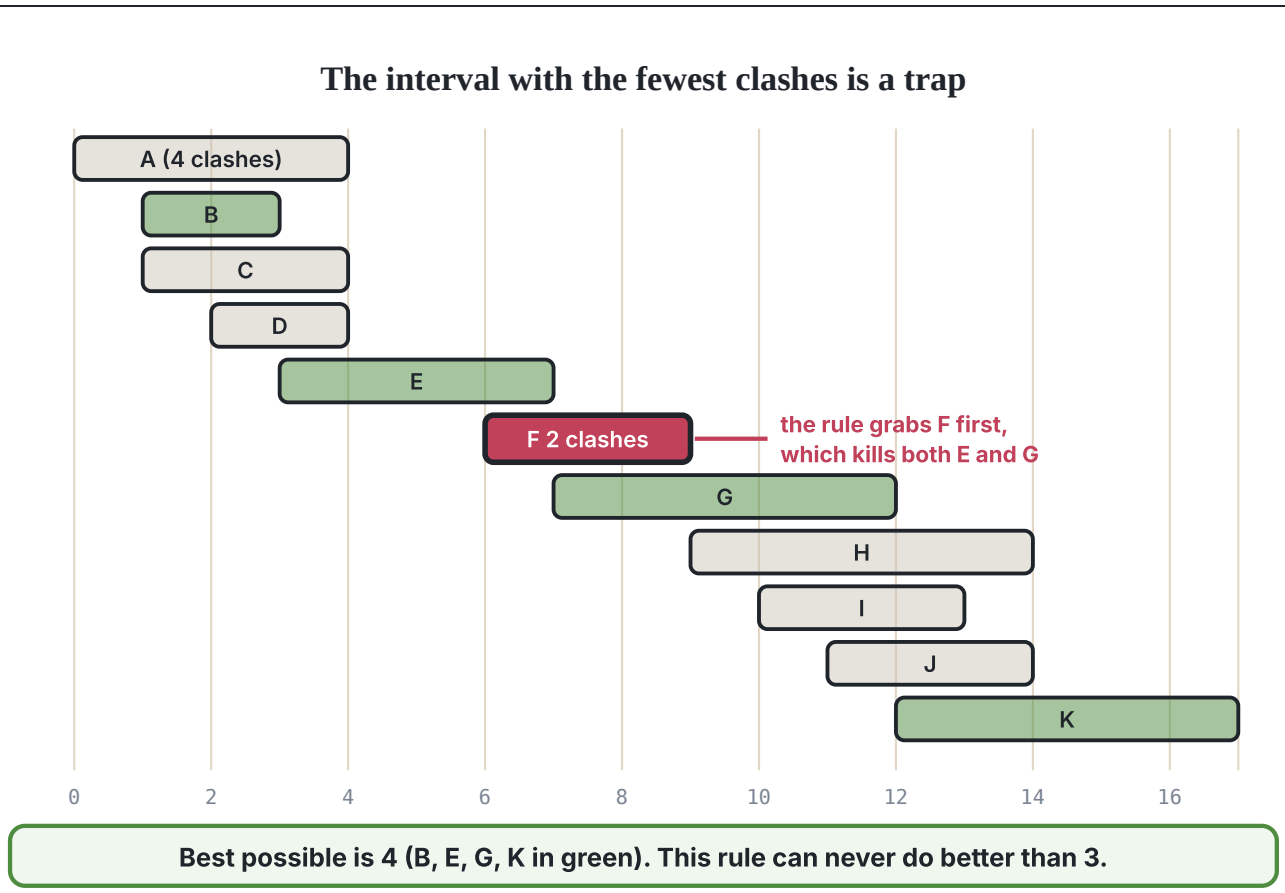
INTERVAL SCHEDULING

Given a set of n intervals $S = \{(start_i, end_j) \mid 1 \le i \le n\}$, find a maximum subset S' of S such that no pair of intervals in S' overlaps. Check whether the following algorithm works or not.

```
while (S is not empty) {
   Select the interval I that overlaps the least number of other intervals.
   Add I to the final solution set S′.
   Remove all intervals from S that overlap with I.
}
```

This rule feels clever. "Take the least troublesome one first" sounds like exactly the right instinct, and on small examples it keeps giving the right answer. That is what makes it a good exam question.

It does not always work. Here is a set of eleven intervals where it fails no matter how you break ties.



Verified counterexample. F clashes with only two intervals, so the rule takes it first. But E and G were the two it destroyed, and those were load-bearing. You end up with three bookings where four were possible. The rule is not wrong by a little. It is simply not guaranteed.

So what is the rule that does work?

Sort by **earliest finishing time**. Take the interval that ends soonest, throw away everything it clashes with, repeat.

The reason it works is almost obvious once you hear it: the interval that finishes earliest leaves the largest possible amount of free time behind it for everything else. No other choice can leave more room. That single sentence is the proof sketch, and it is the sentence to write in an exam.

REAL LIFE

You have one projector and many groups want to book it. To fit in the most groups, you do not ask who has the fewest clashes and you do not ask who asked first. You give it to whoever will **finish earliest**, then look at what is still possible, then repeat. Short bookings early leave the most room.

STICK IT IN YOUR BRAIN

Greedy takes the **local best at each stage and never looks back**. Simple and fast, but only correct when you can prove the local choice is safe. For interval scheduling the safe choice is **earliest finishing time**, not fewest clashes.

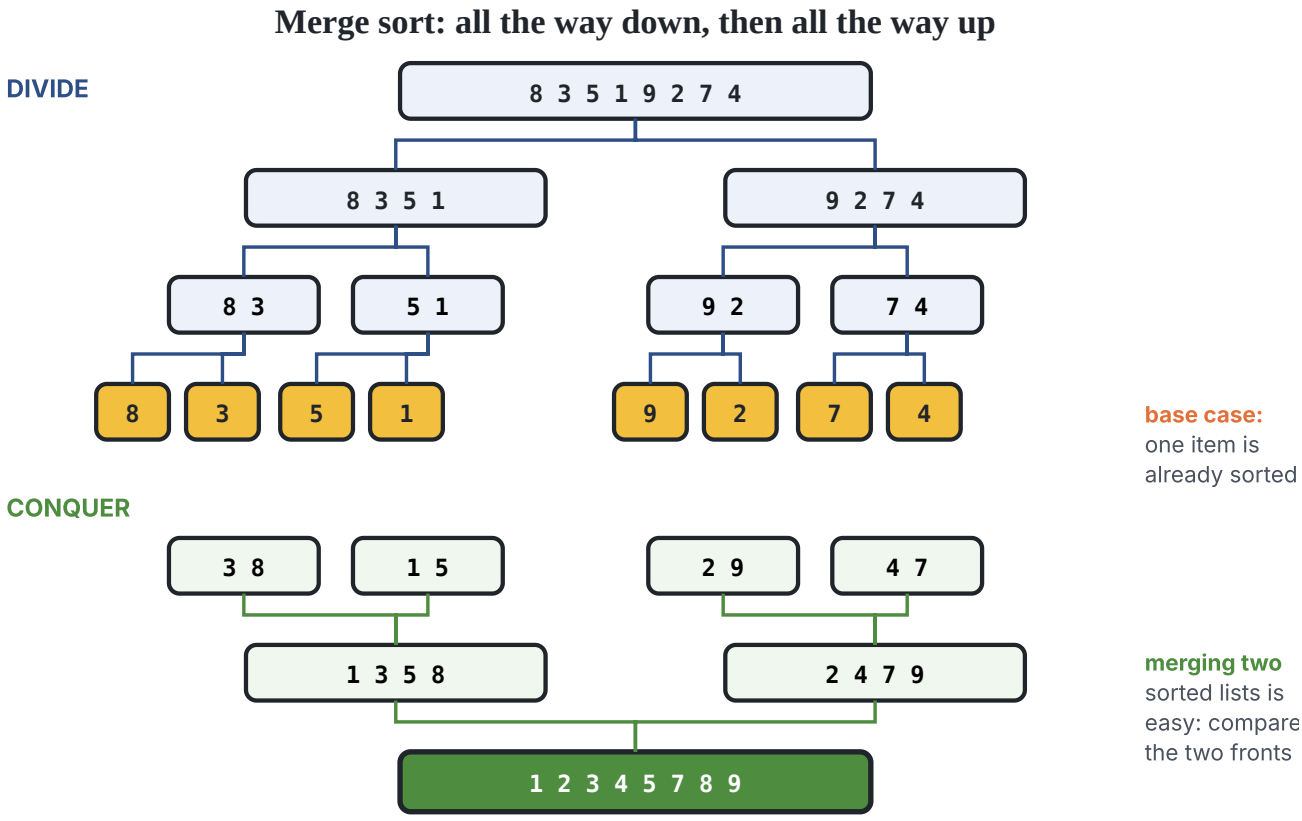
17 Divide and conquer: break it, solve it, join it

Breaking a problem into smaller instances of the same problem.

The strategy has exactly three moves:

1. **Divide.** Break the problem into subproblems that are smaller instances of the *same type* of problem.
2. **Recursion.** Solve those subproblems recursively.
3. **Conquer.** Combine their answers appropriately.

That phrase "the same type of problem" is the whole trick. Half of a sorting problem is still a sorting problem, so the same method handles it.



Notice where the real work happens. Splitting costs nothing. The thinking is all in the merge, and merging two already-sorted lists is simple: look at the front of each, take the smaller, repeat. That is why merge sort is fast.

What you get

- **It cracks hard problems.** The Tower of Hanoi is the classic example. A problem that looks impossible for 20 discs becomes trivial once you see it as "move 19 discs, move 1 disc, move 19 discs."

- **Parallelism.** The subproblems are independent, so different processors can take different subproblems. On shared-memory machines you do not even need to plan the data movement in advance.
- **Memory access.** Divide and conquer naturally makes efficient use of memory caches, because the small subproblems fit in fast memory.

What it costs

- **Recursion is slow.** There is overhead on every repeated subproblem call.
- **It needs stack space** to store the state at each point in the recursion. With large enough base cases the overhead becomes negligible, but it is real.
- **Sometimes it is just more complicated.** A clear example: to add n numbers, a simple loop is far easier than splitting the set in two, adding each half recursively, and adding the sums.

Applications: binary search, merge sort and quick sort, median finding, min and max finding, matrix multiplication, and the closest pair problem.

REAL LIFE

A hundred exam scripts to mark and four lecturers. Nobody marks a hundred scripts. You split into four piles of twenty-five, each person marks their pile, then you stack the four piles together. Divide, solve independently, combine.

And you can feel the disadvantage too. If there were only six scripts, splitting them four ways and handing them around costs more effort than just marking the six yourself. That is the overhead described above.

STICK IT IN YOUR BRAIN

Divide, recursion, conquer. Break into smaller instances of the same problem, solve recursively, combine the answers. The subproblems are **independent**, which is what makes it parallelisable, and is also what separates it from dynamic programming.

18 Dynamic programming: never solve the same thing twice

Recursion, plus a table of the answers already worked out.

There is a formula worth memorising exactly as written:

THE FORMULA

Dynamic Programming = Recursion + Memoization

So what is **memoization**? It is maintaining a table of already-solved subproblems. Before you solve anything, you check the table. If the answer is there, you take it. If not, you solve it and write it in.

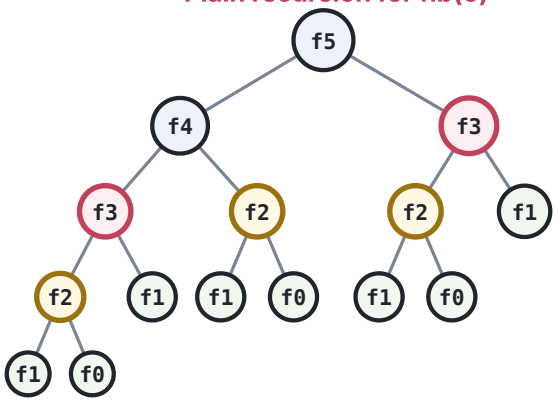
The difference from divide and conquer

This is the exam question, and it answers in one line. In divide and conquer there is **no dependency among the subproblems**. In dynamic programming there is an **overlap** of subproblems.

Overlap means the same subproblem keeps coming back. Without a table you solve it again and again and again, which is where exponential blow-up comes from. With the table, memoization reduces exponential complexity to polynomial complexity, $O(n^2)$, $O(n^3)$ and so on, for many problems.

Same work, done five times, or written down once

Plain recursion for fib(5)



f3 computed twice. f2 three times. f1 five times. For fib(50) this is billions of repeated calls.

The same job with a table

n	fib(n)
0	0
1	1
2	1
3	2
4	3
5	5

Each row written once, read many times. Six calculations instead of fifteen.

The subproblems OVERLAP. That overlap is the signal for dynamic programming.

If subproblems never repeat, memoization buys nothing and DP is the wrong tool. Dynamic programming is not a solution for all problems.

Overlap is the whole signal. Look at how many times f2 appears on the left. Every repeat is work you already did. The table on the right is the entire fix.

REAL LIFE

You are a student in a tough course and people keep asking you the same past question. The first three times you work it out fresh on paper. By the fourth time you are tired, so you write the worked solution in your notebook and from then on you just open the page.

You did not get smarter. You stopped repeating yourself. That notebook is a memoization table, and that is the entire difference between a program that answers in milliseconds and one that answers next week.

The two directions

Bottom-up DP

Evaluate the function starting with the smallest possible input value, then step through the values, slowly increasing. You fill the table from row 0 upward. No recursion at all.

Top-down DP

Keep the recursive structure of the original code, but avoid unnecessary recalculation. Break into subproblems, solve them, and remember the solutions in case they are needed again.

Where it is used

The main ones:

- **String algorithms:** longest common subsequence, longest increasing subsequence, longest common substring, edit distance
- **Graph algorithms:** Bellman-Ford for shortest distance, Floyd's all-pairs shortest path
- Chain matrix multiplication
- Subset sum
- 0/1 knapsack
- Travelling salesman problem, and many more

WHERE STUDENTS LOSE MARKS

Saying "DP is just recursion." It is not. The difference between dynamic programming and plain recursion is **the memoization of recursive calls**. No table, no DP.

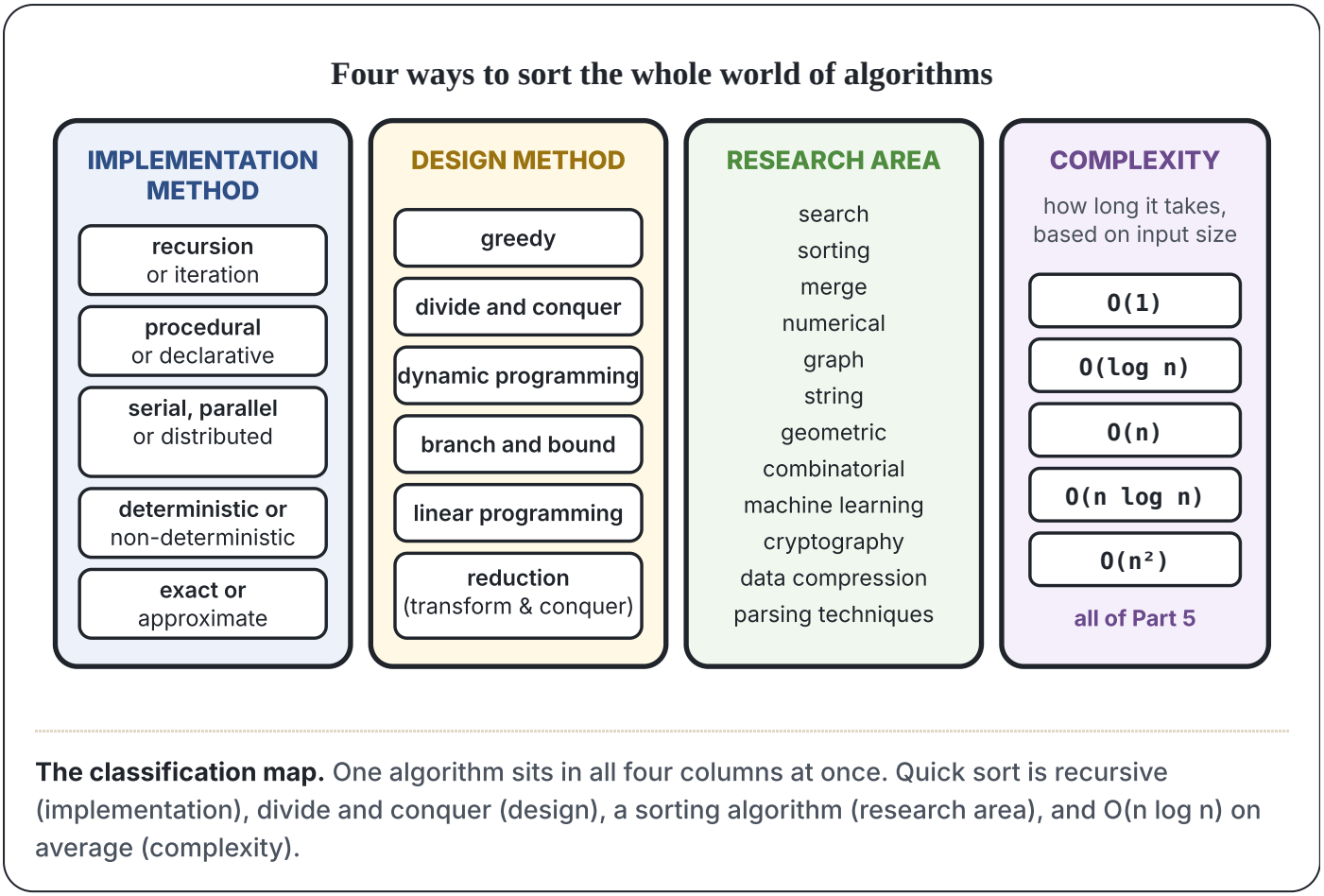
STICK IT IN YOUR BRAIN

DP = recursion + memoization. Divide and conquer has independent subproblems. DP has **overlapping** subproblems. The table turns exponential into polynomial. If nothing repeats, DP helps nothing.

19 The rest of the family

The remaining classifications, collected in one place.

There are many ways to classify algorithms. Four axes matter most: by **implementation method**, by **design method**, by **research area**, and by **complexity**. Chapters 16 to 18 covered the main design methods. Here is everything else.



The classification map. One algorithm sits in all four columns at once. Quick sort is recursive (implementation), divide and conquer (design), a sorting algorithm (research area), and O(n log n) on average (complexity).

Recursion or iteration

A **recursive** algorithm calls itself repeatedly until a base condition is satisfied. An **iterative** algorithm uses loops, and sometimes stacks and queues, to do the same job. Some problems suit recursion, others suit iteration. Iterative algorithms are generally the ones that parallelise well.

REAL LIFE

Recursion is asking the person in front of you in a queue "how many people are ahead of you?", and they ask the person ahead of them, until someone at the front says "nobody." Then the answer comes back down the line, plus one each time.

Iteration is walking to the front of the queue yourself, counting as you go.

Procedural or declarative

In **declarative** languages you say *what* you want without saying *how* to get it. In **procedural** programming you must specify the exact steps. The clearest example is SQL, which is more declarative than procedural, because the queries do not specify the steps used to produce the result. Procedural languages include C, PHP and PERL.

> **REAL LIFE**
>
> Declarative is telling a taxi driver "take me to Mile 17." Procedural is telling him "turn left, go 200 metres, turn right at the junction, continue past the market." Same destination, completely different amount of talking.

Serial, parallel or distributed

Normally we assume computers execute one instruction at a time, which gives **serial** algorithms. **Parallel** algorithms take advantage of computer architectures that process several instructions at a time, dividing the problem into subproblems and serving them to several processors or threads. If those parallel pieces are spread across different machines, we call it **distributed**.

Deterministic or non-deterministic

Deterministic algorithms solve the problem with a predefined process. **Non-deterministic** algorithms guess the best solution at each step through the use of heuristics. A heuristic is an educated guess that is usually right and carries no guarantee.

Exact or approximate

Algorithms for which we can find the optimal solution are **exact**. When we cannot find the optimal solution, we give **approximation** algorithms instead. These are generally associated with NP-hard problems, which are the ones on the top step of chapter 3's ladder.

Randomized algorithms

A few algorithms make choices randomly. For some problems the fastest known solutions must involve randomness. Quick sort is the example, where the pivot can be chosen at random to avoid the worst case.

Linear programming

Here there are inequalities in terms of the inputs, and you maximise or minimise some linear function of those inputs. Many problems, for example maximum flow in directed graphs, can be discussed using linear programming.

REAL LIFE

You sell two kinds of bag. Each small bag takes 1 hour and gives 2,000 profit. Each large bag takes 3 hours and gives 5,000 profit. You have 40 hours and material for 20 bags. How many of each should you make? Those constraints are inequalities, the profit is a linear function, and the whole situation is a linear program.

Reduction, also called transform and conquer

Here you solve a difficult problem by **transforming it into a known problem** for which you already have an optimal algorithm. The goal is to find a reducing step whose cost is not worse than the algorithm you are reducing to.

A clean example: to find the median of a list, first sort the list, then take the middle element. You did not invent a median algorithm. You turned the median problem into a sorting problem, which was already solved.

REAL LIFE

You cannot open a tin because you have no tin opener. Instead of inventing a tin-opening technique, you walk next door and borrow a tin opener. You converted "open this tin" into "find a neighbour," a problem you already know how to solve.

Branch and bound, enumeration and backtracking

These are used in artificial intelligence. You do not need to explore them fully at this level. The one-line idea: explore the tree of possibilities, and abandon a branch as soon as you can prove it cannot beat the best answer you already have.

STICK IT IN YOUR BRAIN

Four axes: **implementation method, design method, research area, complexity**. One algorithm has a position on all four at the same time. If an exam asks you to classify an algorithm, give all four, not one.

PART 5

Is it fast? Is it fat?

Two algorithms can both be correct and one can still be useless. This is how you tell them apart before you write a single line of code.

20 Time complexity and the frequency count

Measuring cost by counting how often the key statement runs.

Time complexity of a program or algorithm is the amount of computer time it needs to run to completion.

But here is the thing: you cannot measure that with a clock, because a clock measures your laptop, not your algorithm. A faster machine gives a smaller number and the algorithm did not improve. So instead we **count**. This is called developing a **frequency count** for all the key statements.

Consider these three algorithms. The key statement in all three is the same single line: `a = a + 1`.

Count the key statement. That count is the answer.

Algorithm A

```
a = a + 1
```

no loop at all, so it executes exactly once whatever n is

= 1

Algorithm B

```
for x = 1 to n
  a = a + 1
```

one loop, running n times, so the key statement runs n times

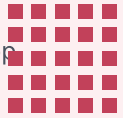


= n

Algorithm C

```
for x = 1 to n
  for y = 1 to n
    a = a + 1
```

the inner loop runs n times for every single run of the outer loop and the outer loop runs n times



= n^2

Why the shape of the loops is everything. Nothing about the statement changed. Only how it was wrapped. One line, one tally. One loop, a row of tallies. A loop inside a loop, a square block of tallies.

Stated plainly:

- Frequency count for **Algorithm A** is **1**, because `a = a + 1` executes only once.
- Frequency count for **Algorithm B** is **n**, because the loop runs n times.
- Frequency count for **Algorithm C** is $\mathbf{n^2}$, because the inner loop runs n times each time the outer loop runs, and the outer loop runs n times.

REAL LIFE

You are greeting people at an event. Greeting the host once is Algorithm A. Walking down a row of n guests and greeting each is Algorithm B. Now imagine every guest must shake hands with every other guest. That is Algorithm C, and with 100 guests it is around ten thousand handshakes. The event does not end.

Nested loops over the same data are worth pausing on. They are not wrong, but they are expensive, and the price is worth knowing before you pay it.

STICK IT IN YOUR BRAIN

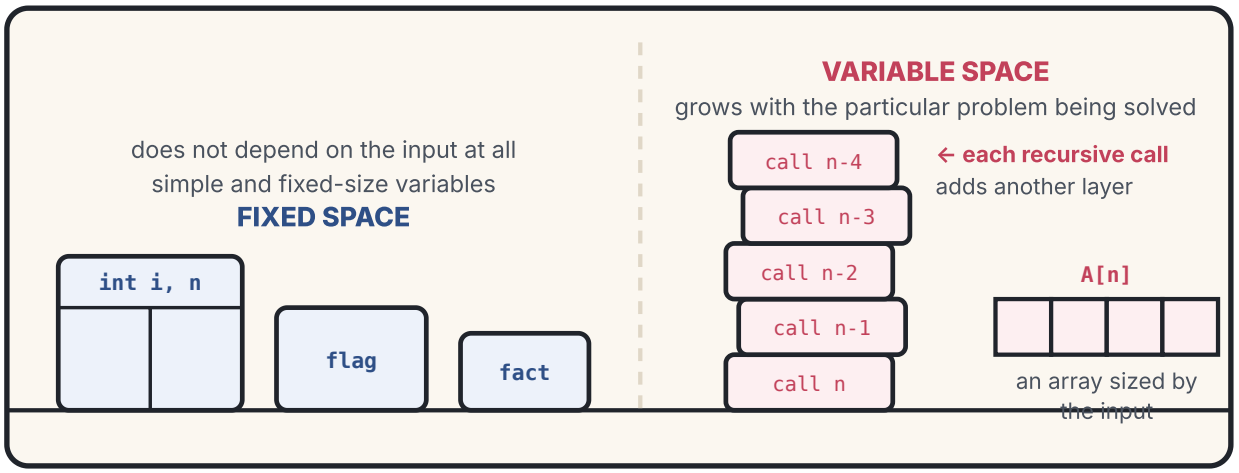
To find time complexity, **count how many times the key statement executes**, in terms of n. No loop gives 1. One loop gives n. A loop inside a loop gives n^2.

21 Space complexity

Fixed and variable memory, including the recursion stack.

Space complexity is the amount of memory a program or algorithm needs to run to completion. The space needed splits into two components, and the split is the important part.

Your memory is a room. Some of it is furniture. Some of it is luggage.



Total space = fixed + variable

The recursion stack is real memory. This is the cost chapter 17 warned about. Deep recursion does not just take time, it takes a layer of memory per call, and when the room is full the program dies.

Fixed space requirements

Not dependent on the characteristics of the input and output. Consists of space for simple variables, fixed-size variables and so on. The same whether n is 5 or 5 million.

Variable space requirements

Space needed by variables whose size depends on the particular problem being solved, plus referenced variables, plus **the stack space required for recursion**. The clearest example is the additional space required when a function uses recursion.

REAL LIFE

Moving house. The sofa and the fridge are the same size no matter how long you stay. That is fixed. The number of suitcases depends on how many people are travelling. That is variable.

And recursion is the sneaky one. Every recursive call is a half-finished job you have set down on the floor, still waiting for its answer. Twenty calls deep means twenty half-finished jobs sitting in your room at once.

STICK IT IN YOUR BRAIN

Space = **fixed** (simple and fixed-size variables, independent of input) **+ variable** (input-dependent variables, referenced variables, and recursion stack space).

22 Best case, worst case, average case

Why the same input size can still take different amounts of time.

One distinction first, because it is easily blurred. There is a difference between a **problem** and an **algorithm**. A problem has a single statement describing it in general terms. There are many ways to solve it, and some solutions are more efficient than others.

Now the subtle part. Running time varies not only for inputs of *different sizes*, but also for inputs of the *same size*. Ten items arranged one way can be much faster than the same ten items arranged another way.

So we give three measures. Remember the linear search from chapter 9, going through a list one item at a time.

Same list. Same algorithm. Three very different days.

BEST CASE the item you want is in the very first slot



1 look

WORST CASE it is in the last slot, or not there at all



n looks

AVERAGE CASE averaged over all the places it could be



about n/2 looks

We design for the worst case, because that is the promise you can actually keep.
A best case of 1 is luck. A worst case of n is a guarantee.

Sorting shows the same effect. If n items are already in sorted order, a sorting algorithm does the least work. Hand it the same n items in reverse order and it needs n^2 operations.

Case	Definition	When you use it
Best case	A measure of the minimum time the algorithm will require for an input of size n.	Rarely. Nice to know, dangerous to plan with.
Worst case	A measure of the maximum time required for input of size n. If n items are supplied in reverse order to a sorting algorithm, it needs n^2 operations.	Almost always. This is the one Big O describes.
Average case	The time required for typical input data of size n. Obtained by averaging the running time over all possible inputs of size n.	When you know the shape of real data and worst case is too pessimistic.

REAL LIFE

How long does it take you to get to campus? **Best case**, the bike is waiting at your gate and the road is clear, so ten minutes. **Worst case**, rain, traffic, no bike, forty-five minutes. **Average case**, about twenty.

Now, which number do you use when deciding what time to leave for an 8am exam? The worst case. Every time. That instinct is exactly why computer scientists quote worst case by default.

STICK IT IN YOUR BRAIN

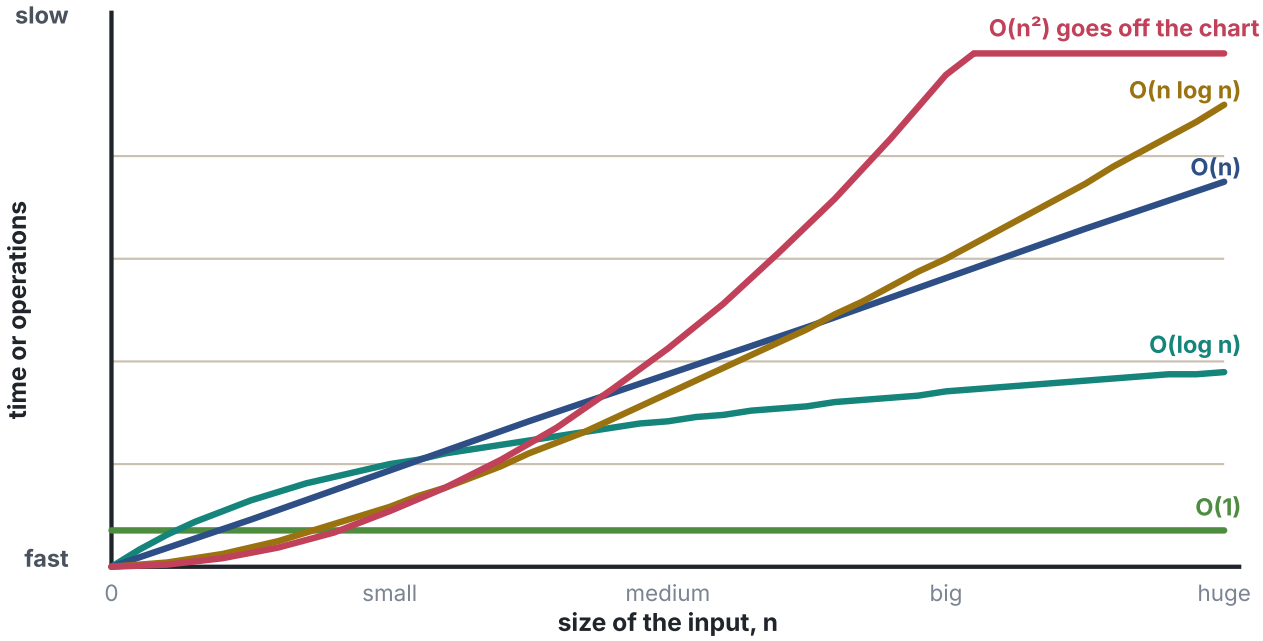
Running time varies for inputs of **different sizes and also for inputs of the same size**. Best is the minimum, worst is the maximum, average is over all inputs of that size. Big O describes the worst case.

23 Big O notation

Describing how an algorithm grows as its input grows.

Big O notation is used in computer science to describe the performance or complexity of an algorithm. Specifically it describes the **worst-case scenario**, and it can describe either the execution time required or the space used, in memory or on disk.

What Big O really answers is one question: when the input gets bigger, how much worse does this get? Not "how many seconds." How it *grows*.



The gap between the green line and the red line is the difference between an app and a frozen screen.

Read the shape, not the numbers. For tiny inputs all five lines are bunched together, which is why a bad algorithm passes your small test and then dies on real data. The lines only separate when n gets large, and real data is always large.

The five you must know

O(1), constant time. Describes an algorithm that will always execute in the same time or space **regardless of the size of the input data set**. The standard example is push and pop on a stack. Taking the top plate off a pile takes the same effort whether the pile has 3 plates or 300.

O(log N), logarithmic time. The iterative halving of the data set described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the data set grows. A common illustration: if 10 items take one second, 100 items take two seconds, and 1000 items take three seconds. Multiply the data by ten, add one second. That is the magic of halving.

O(N), linear time. Performance grows linearly and in direct proportion to the size of the input data set. Double the data, double the time. The standard example is linear search over unsorted data. Remember that Big O always assumes the upper limit.

O(N log N). The more advanced sorting algorithms sit here: quick sort and merge sort. It is slower than linear but dramatically better than quadratic, and it is the best that general comparison-based sorting can do.

O(N^2), quadratic time. Performance is directly proportional to the **square** of the size of the input data set. This is common with algorithms that involve **nested iterations** over the data set. The standard example is comparing two dimensional arrays of size n.

A SMALL NAMING NOTE

Your lecture slide labels both O(log N) and O(N log N) as "Logarithmic Time." O(log N) is the logarithmic one. O(N log N) is usually called **linearithmic** or just "n log n." Say "n log n" in the exam and you are safe either way.

What these feel like at real sizes

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)
10	1	3	10	33	100
100	1	7	100	664	10,000
1,000	1	10	1,000	9,966	1,000,000
1,000,000	1	20	1,000,000	19,931,569	1,000,000,000,000

Look at the bottom row. At one million items, the O(log n) algorithm does twenty steps. The O(n^2) one does a trillion. Same problem, same machine, same programmer. The only difference is which algorithm was chosen.

REAL LIFE

Finding a name in a phone book. **O(n)** is starting at page one and reading every name. **O(log n)** is opening it in the middle, seeing you are past the name, and throwing away half the book. Then half of what remains. Then half again.

A thousand-page book needs about ten of those halvings. Ten. That is what the word logarithmic describes, and it is something you have done since you were a child.

WHERE STUDENTS LOSE MARKS

Writing O(2n) or O(n + 5). Big O throws away constants and smaller terms, because they stop mattering as n grows. O(2n) is just O(n). And $O(n^2 + n)$ is just $O(n^2)$.

STICK IT IN YOUR BRAIN

O(1) same time always. **O(log n)** halving, like binary search. **O(n)** one pass, like linear search. **O(n log n)** good sorting, quick and merge. **$O(n^2)$** nested loops over the same data. Big O describes the **worst case** and ignores constants.

24 The exam survival sheet

The definitions, lists and distinctions, collected for revision.

If you read nothing else again, read this page.

Definitions you may be asked to write out

- **Algorithm:** a finite set of explicit, unambiguous steps which, for given initial conditions, produce output and terminate in finite time.
- **Program:** an algorithm executable on a computer.
- **Programming language:** a collection of primitives, the rules for combining them, and the meanings attributed to them.
- **Data type:** a set of values together with the permissible operations on them.
- **Data structure:** a way of storing and organising data so it can be used efficiently.
- **Time complexity:** the amount of computer time an algorithm needs to run to completion.
- **Space complexity:** the amount of memory an algorithm needs to run to completion.

Lists you should be able to produce cold

Question	Answer
Five properties of an algorithm	Input (zero or more), output (one or more), definiteness, finiteness, effectiveness
Six stages of programming	Define, analyse, design, implement, test, document (plus compare alternatives)
Three representations	Pseudocode, flowcharts, dataflow diagrams
Programming structures	Sequence, selection, repetition, subprograms
Flowchart shapes	Oval start/end, rectangle process, diamond decision, double-sided rectangle subprogram, circle continuation
Two design approaches	Top-down, bottom-up
Four classification axes	Implementation method, design method, research area, complexity
Design methods	Greedy, divide and conquer, dynamic programming, branch and bound, linear programming, reduction
Three time complexity cases	Best, worst, average
Two space components	Fixed, variable (variable includes the recursion stack)

The one-line distinctions that carry marks

Between	The deciding sentence
Algorithm and program	Every program is an algorithm. Not every algorithm is a program.
Syntax and semantics	Syntax is form, semantics is meaning. A compiler only checks form.
WHILE and REPEAT UNTIL	WHILE tests first and can run zero times. REPEAT UNTIL tests after and always runs at least once.
Divide and conquer vs DP	D&C subproblems are independent. DP subproblems overlap.
Recursion vs DP	Dynamic programming is recursion plus memoization.
Greedy vs the rest	Greedy takes the local best and never reconsiders. Correct only when provable.
Procedural vs declarative	Procedural says how. Declarative says what. SQL is declarative, C is procedural.
Exact vs approximate	Exact finds the optimum. Approximate is used when the optimum is out of reach, usually on NP-hard problems.
Deterministic vs non-deterministic	Deterministic follows a predefined process. Non-deterministic guesses using heuristics.

Frequency counting, in three moves

1. Find the key statement, the one doing the real work.
2. Ask how many times it runs, in terms of n. No loop is 1. One loop is n. Nested loops multiply.
3. Drop the constants and the smaller terms. What is left is your Big O.

THE LAST THING TO REMEMBER

Everything in this course comes back to one habit: **before you write code, write the steps**. On paper, in plain words, with every decision visible. Mistakes on paper cost seconds. The same mistakes in code cost nights.

That is the heart of the subject. Most of the rest is vocabulary built on top of it.

This course rests on about twenty ideas. Most of them describe something you already knew how to do.

CSC207 Made Easy

A plain English rewrite of the CSC207 Introduction to Algorithms course, 2023/2024 academic year. Every definition, pseudocode listing, code sample and complexity rule has been kept in full. What has been added is the explanation around them.

Original course material by Drs. Nyanga B., Achenkeng P. and Fotsing C. Definitions quoted in chapter 1 are from Dromey, *How to Solve it by Computer* (1982) and Woodhouse et al, *Computer Science* (1984).