

Data Collections

AP Computer Science A ■ Unit 4

AP Computer Science A

4	8	2	9
---	---	---	---

What we will cover

- 1 Ethics of collecting data
- 2 Arrays
- 3 ArrayList
- 4 2D arrays (grids)
- 5 Searching & sorting
- 6 Recursion

4	8	2	9	5
0	1	2	3	4



The ethics of collecting data

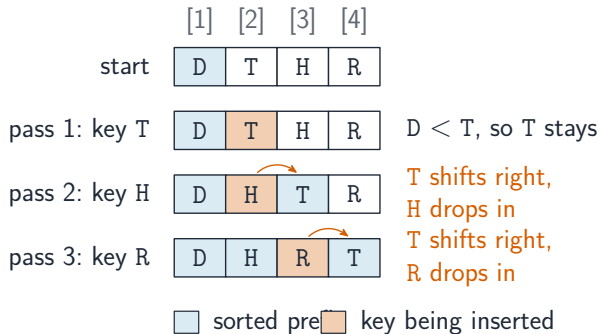
Programs that collect and process personal data carry real responsibility.

Collect only what you need, protect **privacy**, be honest about how data is used, and guard against **bias** in what you gather. Legal is not always ethical.



Data centre

An insertion sort



An insertion sort, shifting each key into place pass by pass

Making and Reading an Array

An **array** is a fixed-size, ordered collection of same-type values.

Indices run from 0 to `length - 1`:

Making and Reading an Array, in detail

index	1	2	3	4	5
scores	90	75	60	88	50

↑
scores[2] is 75

A one-dimensional array (a list) with its indices and bounds

The five standard array algorithms

sum or average

an accumulator declared outside the loop; divide by length at the end

max or min

start from `arr[0]`, not from zero – an array of negatives has no element above zero

count matching

a counter, incremented only when the element passes the test

check for a duplicate

a **nested** loop, with the inner one starting at `i+1` so no pair is checked twice

reverse or copy

build into a **new** array, or swap ends inward for an in-place reverse

Every one of these is a traversal plus a small body. Recognising which one a question wants is most of the work.

Visiting Every Element of an Array

Traverse an array with a `for` loop (gives the index) or an **enhanced `for` / `for-each`** loop (gives each value, read-only):

Wrapping a Number in an Object

An `ArrayList` stores **objects**, not primitives, so a primitive is **wrapped** in an object: `Integer` wraps `int`, `Double` wraps `double`.

Java does this with **autoboxing** (`int` to `Integer`) and **unboxing** (back again) automatically.



Arrays

4	8	2	9	5	7
0	1	2	3	4	5

An **array** stores a fixed number of same-type values, reached by **index** (from **0**). Visit every element with a loop:

```
for (int i = 0; i < a.length; i++) ...
```

Standard algorithms: find the **max/min**, **sum/average**, or count matches by traversing once.



Filing cabinet

Why a data structure at all

the problem

a single variable holds **one** value; real problems need many related values – a class roster, pixels, sensor readings

a data structure

holds many values under one name, reached by **index**

an array

fixed length, set when it is created, and `arr.length` is a field, not a method

the bounds

valid indices run 0 to `length-1`; anything outside throws an `ArrayIndexOutOfBoundsException`

the run-time cost

reaching any element takes the same time whatever the index – that is the point of an array

Off-by-one at the top end is the classic array bug: `i <= arr.length` throws, `i < arr.length` does not.



The ArrayList toolbox

An **ArrayList** grows and shrinks, unlike a fixed array. It stores objects, so numbers are **wrapped** (Integer, Double).

add(x)
append

get(i)
read

set(i,x)
replace

remove(i)
delete

Traverse it with a for or an enhanced for-each loop: `for (int x : list).`

Removing from a list while looping over it

the problem

`remove(i)` shifts every later element down, so the element after the removed one is skipped

fix one

loop backwards, from `size()-1` down to 0 – later shifts cannot affect indices you have passed

fix two

do **not** increment `i` after a removal, so the next pass re-examines the same index

never

remove inside an enhanced for loop – it throws a `ConcurrentModificationException`

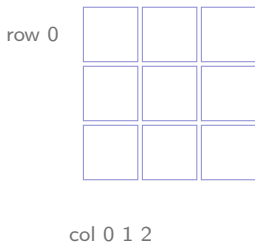
the rest

the array algorithms carry over unchanged, plus **insertion** and **deletion**, which arrays cannot do easily

This is the single most-tested ArrayList point. If a question removes items in a loop, it is testing exactly this.



2D arrays (grids)



A **2D array** is a grid of rows & columns:
`grid[r][c]`.

Walk it with **nested loops** – outer over rows, inner over columns:

```
for r ...  
    for c ...  
        grid[r][c]
```

Grids: Two-Dimensional Arrays

		column		
		1	2	3
row	1			
	2			
	3			

← `grid[2,3]`

A two-dimensional array (a table) with row and column indices

Walking Through a Grid

Visit every cell with **nested loops** – the outer over rows, the inner over columns
(**row-major order**):

The standard grid algorithms

sum a rowfix `r`, loop `c` across `grid[r].length`**sum a column**fix `c`, loop `r` down `grid.length`**max in the grid**nested loops, initialised to `grid[0][0]`**count matching cells**

nested loops with a counter and a test

sum a diagonalone loop, using the cells where `r == c`

Every grid task is a nested traversal with a small body – and `grid.length` is the number of [rows](#), `grid[0].length` the number of columns.



Searching & sorting

Linear search checks every element; **binary search** halves a **sorted** array

Selection & insertion sort order a list; **merge sort** is faster

Binary search needs **sorted** data and does $\sim \log_2 n$ comparisons, far fewer than linear search's n .

Selection sort and insertion sort

selection sort

repeatedly find the **smallest remaining** element and swap it into place

insertion sort

take the next element and **slide it back** into its position among the already-sorted front

both

$O(n^2)$ comparisons in the worst case – fine for small arrays, hopeless for large ones

the difference

insertion sort is much faster on **nearly-sorted** data; selection sort takes the same time whatever the input

the exam skill

be able to state the array's contents **after k passes** of either

Trace, do not memorise. The questions ask what the array looks like partway through, which only tracing answers.

Recursion

A **recursive** method calls itself on a smaller input, stopping at a **base case** .

Factorial

`factorial(n) = n * factorial(n-1)`, with `factorial(0) = 1`.

Each call shrinks n until the base case, then the results multiply back up. **Merge sort** uses the same divide-and-conquer idea.

Reading a file, and the two exceptions it can throw

the import

File and IOException live in `java.io`, so a file-reading program needs `import java.io.*`;

why it can fail

the file might not exist, so Java forces you to [handle or declare](#) the exception

the two ways

wrap the call in `try/catch`, or add `throws IOException` to the method header

typed tokens

`nextInt()`, `nextDouble()` and `nextBoolean()` throw an `InputMismatchException` if the next token is the [wrong type](#)

the usual cause

a line of text where a number was expected, or a stray blank line

File input is the one place a beginner's program meets the outside world, which is exactly why the language insists you say what happens when it goes wrong.

Key points

Arrays are fixed; **ArrayLists** grow & hold objects

A **2D array** is a grid,
walked with nested loops

Binary search & merge sort
beat the simple versions

Recursion solves a problem
via a smaller copy

Worked example – tracing recursion

Trace factorial(4)

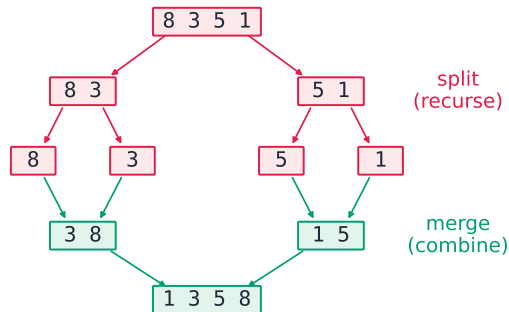
Each call **defers** to a smaller one:

$$4 \cdot f(3) = 4 \cdot 3 \cdot f(2) = 4 \cdot 3 \cdot 2 \cdot f(1)$$

factorial(1) hits the **base case** and returns 1, so the calls **unwind inward**: $2 \cdot 1 = 2$, then $3 \cdot 2 = 6$, then $4 \cdot 6 = 24$.

Writing each call **above** its returned value is the reliable way to trace recursion – and the base case is what stops it.

Merge sort splits the array to single elements



merge sort: $O(n \log n)$ time · needs extra $O(n)$ space

Merge sort splits the array to single elements, then merges sorted halves back up

Binary search halves the range at each step – one

Diagram illustrating the binary search algorithm for finding the letter 'W' in an alphabet (A-Z).

Step 1: low = 0, mid = 12, high = 25. The letter 'M' is compared to 'W'. Since $M < W$, the search range is updated to the right half (low = 13).

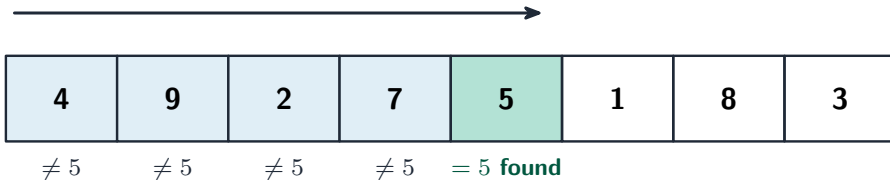
Step 2: low = 13, mid = 19, high = 25. The letter 'T' is compared to 'W'. Since $T < W$, the search range is updated to the right half (low = 20).

Step 3: low = 20, mid = 22, high = 25. The letter 'W' is compared to 'W'. Since $W = W$, the letter is found.

Binary search halves the range at each step

Binary search halves the range at each step – two

search for 5: check each item in turn (left to right)



Linear search checks every element in turn until the target is found

Exam tips

- Weigh both benefits and harms of collecting data – this unit is tested through short written justification, not code.
- Protect **personally identifiable information (PII)** and explain privacy and security risks in context.
- Name real harms: data breaches, surveillance, and algorithmic **bias** from unrepresentative data.
- Respect intellectual property and licensing when you reuse code or data.
- Give a specific, reasoned answer – a vague “it could be bad” earns no marks.



Unit 4 – key takeaways

1

Both sides

benefits and harms of collecting data

2

PII

privacy and security risks in context

3

Harms

breaches, surveillance, algorithmic bias

4

IP

respect licensing; a vague 'it could be bad' scores zero

Check yourself

- 1 factorial(4) returns what?
- 2 What happens to recursion with no base case?
- 3 Binary search needs the array to be ...?
- 4 Array or ArrayList: which resizes itself?

Answers 1. 24 2. it never stops – a StackOverflowError 3. sorted 4. ArrayList