Data Collections

AP Computer Science A

The Ethics of Collecting Data

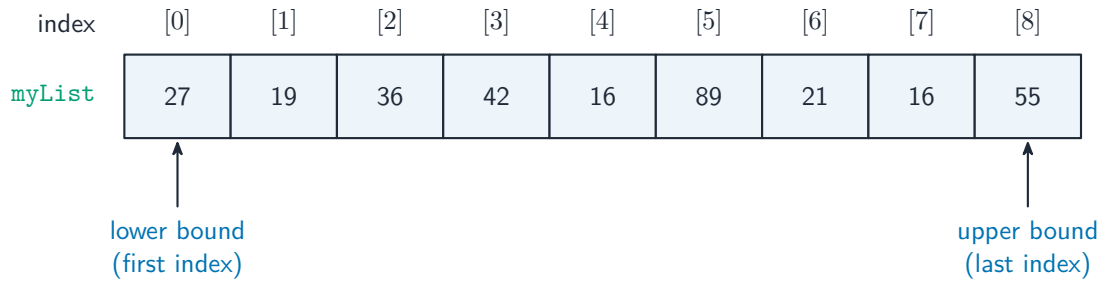
Programs that gather data raise questions of **privacy** 隐私 and **consent** 同意. Collect only what is needed, protect it, and be honest about its use. Data can carry **bias** 偏见 if it does not represent everyone fairly, leading to unfair results –a responsibility that comes with storing information.

Why We Need Data Structures

A single variable holds one value; real problems need to store **many** related values –a class roster, pixels, sensor readings. A **data structure** 数据结构 organizes a collection so we can store, find, and process items efficiently. The AP course uses three: the **array**, the **ArrayList**, and the **2D array**.

Making and Reading an Array

An **array** 数组 is a fixed-size, ordered collection of same-type values. Indices run from `0` to `length - 1`:



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

```
int[] nums = new int[5];        // five zeros
int[] vals = {3, 1, 4, 1, 5};   // initialized
int first = vals[0];            // 3
int n = vals.length;            // 5 (a field, not a method)
```

Accessing an index outside `0..length-1` throws an `ArrayIndexOutOfBoundsException`.

Visiting Every Element of an Array

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

```java
for (int i = 0; i < a.length; i++) { a[i] *= 2; }   // can modify
for (int v : a) { System.out.println(v); }          // read each value
```

Standard Array Algorithms

Master these patterns: compute a **sum** or **average**, find the **max/min**, **count** items meeting a condition, check for a **duplicate**, and **reverse** or **shift** elements. Each is a traversal with a running result:

```java
int sum = 0;
for (int v : a) sum += v;
double avg = (double) sum / a.length;
```

Reading Data from a Text File

A `Scanner` can read a file line by line, using `hasNext...` to test before reading:

```java
Scanner f = new Scanner(new File("data.txt"));
while (f.hasNextLine()) {
    String line = f.nextLine();
}
```

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** 自动装箱 (`int` to `Integer`) and **unboxing** (back again) automatically, so you can write `list.add(5)` and `int x = list.get(0)`.

The ArrayList Toolbox

An **ArrayList** 动态数组 grows and shrinks as you add or remove items. Declare it with the element type in `<>`:

```java
ArrayList<String> names = new ArrayList<String>();
names.add("Amy");           // append
names.add(0, "Bob");        // insert at index
names.get(0);               // read
names.set(1, "Cara");       // replace
names.remove(0);            // delete, shifts the rest left
names.size();               // count (a method, unlike array.length)
```

Visiting Every Element of an ArrayList

Traverse with an index loop or a for-each loop, just like arrays (use `size()` and `get(i)`):

```
for (int i = 0; i < list.size(); i++) { ... list.get(i) ... }
for (String s : list) { ... }
```

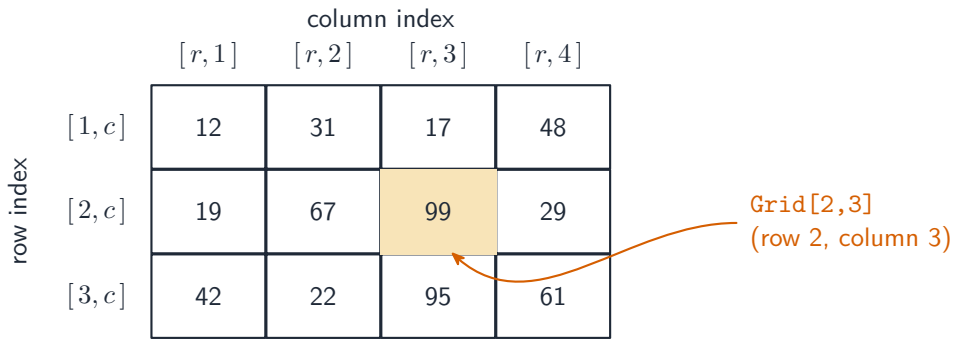
Exam skill: when **removing** items in an index loop, either loop **backwards** or do **not** increment `i` after a removal –otherwise removing shifts elements left and you skip one.

Standard ArrayList Algorithms

The same algorithms as arrays –max/min, count, sum –plus **insertion** and **deletion** that arrays cannot do easily. A common task is to remove all elements matching a condition, handling the index-shift carefully.

Grids: Two-Dimensional Arrays

A **2D array** 二维数组 is a grid (rows and columns) –an array of arrays:



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

```
int[][] grid = new int[3][4];   // 3 rows, 4 columns
grid[r][c] = 7;                 // row r, column c
int rows = grid.length;         // 3
int cols = grid[0].length;      // 4
```

Walking Through a Grid

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

```
for (int r = 0; r < grid.length; r++)
    for (int c = 0; c < grid[0].length; c++)
        System.out.print(grid[r][c]);
```

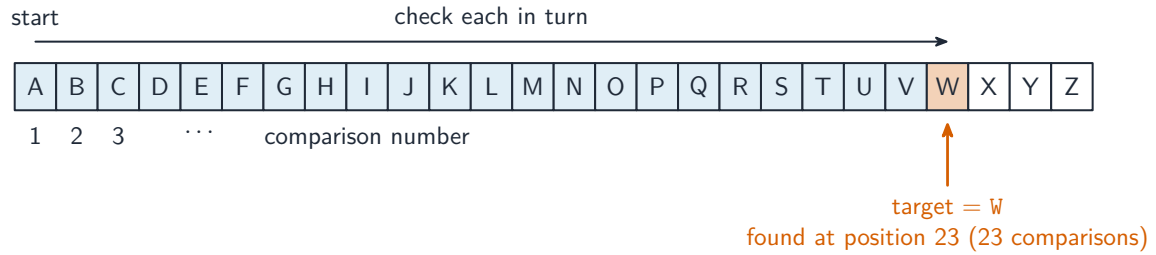
Standard 2D Array Algorithms

Typical grid tasks: sum a row or column, find the max in the grid, count matching cells, or sum a diagonal (where `r == c`). Each is a nested traversal with a running result.

Finding a Value: Linear and Binary Search

- **Linear search** 线性搜索 checks each element in turn –works on any list, taking up to n steps.
- **Binary search** 二分搜索 works only on a **sorted** list: check the middle, then discard the half that cannot contain the target, repeating. It takes about $\log_2 n$ steps –far faster on large data.

Binary search halves the range at each step



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

```
int lo = 0, hi = a.length - 1;
while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (a[mid] == target) return mid;
    else if (a[mid] < target) lo = mid + 1;
    else hi = mid - 1;
}
```

Exam skill: binary search **requires sorted data**; know how many comparisons it makes and how `lo`, `hi`, `mid` update.

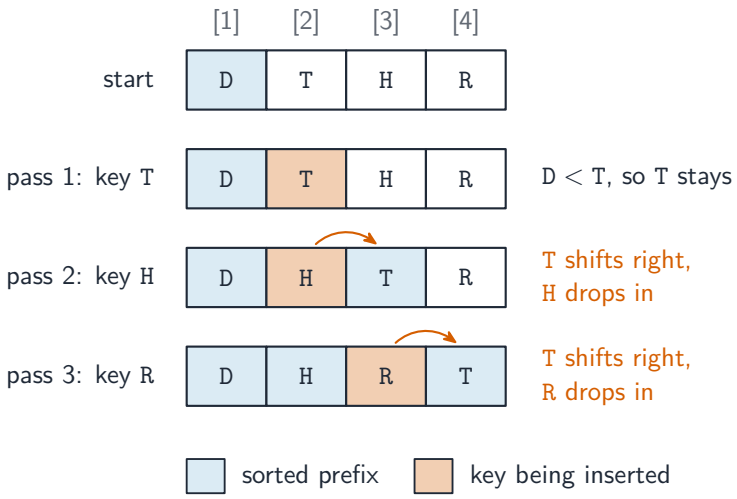
Worked example. Search for `target = 40` in the sorted array `{3, 9, 14, 23, 31, 42, 55}` (indices 0–6). Start `lo=0, hi=6`:

- `mid = (0+6)/2 = 3`, `a[3]=23 < 40`, so `lo = 4`;
- `mid = (4+6)/2 = 5`, `a[5]=42 > 40`, so `hi = 4`;
- `mid = (4+4)/2 = 4`, `a[4]=31 < 40`, so `lo = 5`;
- now `lo (5) > hi (4)`, so the loop ends –`40` is **not present**.

Each step halved the range, so even this miss took only three comparisons.

Putting Data in Order: Selection and Insertion Sort

- **Selection sort** 选择排序 repeatedly finds the smallest remaining element and swaps it into place.
- **Insertion sort** 插入排序 grows a sorted front, inserting each new element where it belongs.



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

Both are simple and take about n^2 steps on average –fine for small arrays. Be able to trace the array **after each pass**.

Methods That Call Themselves: Recursion

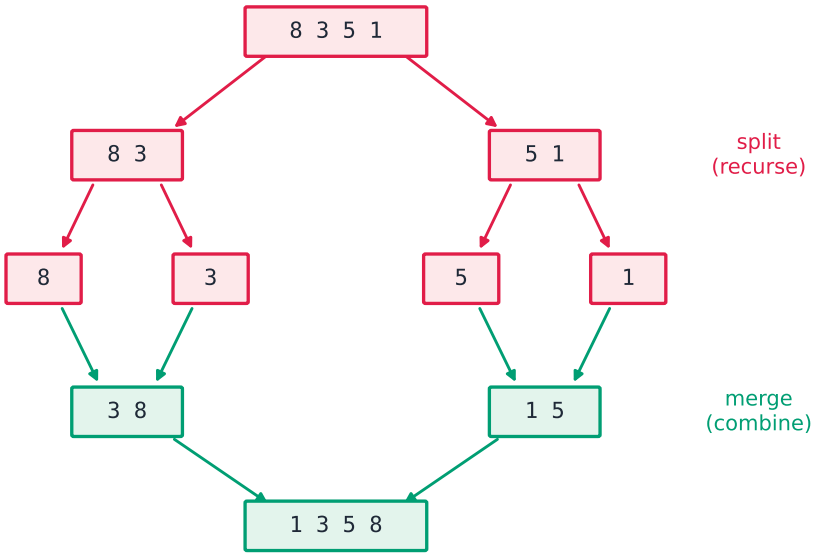
Recursion 递归 is a method that calls itself on a smaller input. It needs a **base case** 基本情况 that stops the calls, and a **recursive case** that moves toward the base:

```
public static int factorial(int n) {
    if (n <= 1) return 1;          // base case
    return n * factorial(n - 1);   // recursive case
}
```

Without a reachable base case, recursion never stops (a stack overflow).

Recursive Search and Merge Sort

Recursion powers efficient algorithms. **Binary search** can be written recursively (search the correct half). **Merge sort** 归并排序 splits the array in half, sorts each half recursively, then **merges** the two sorted halves –taking about $n \log_2 n$ steps, much faster than selection or insertion sort on large data.



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

Worked example. Trace `factorial(4)`. Each call defers to a smaller one: `factorial(4)` = `4 * factorial(3)` = `4 * 3 * factorial(2)` = `4 * 3 * 2 * factorial(1)`. `factorial(1)` hits the **base case** and returns `1`, so the calls unwind inward: `2 * 1 = 2`, then `3 * 2 = 6`, then `4 * 6 = 24`. Writing each call above its returned value is the reliable way to trace recursion.

Exam skill: trace a recursive method by writing out each call and its return value, and know that merge sort's efficiency ($n \log n$) beats the n^2 simple sorts.

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.