

# Unit 10: Recursion

Unit 10 is the shortest unit in AP Computer Science A, and it covers a single idea. A recursive method is a method that calls itself. This guide covers the two parts every recursive method needs, how the call stack handles the nested calls, and how to trace a recursive method by hand. Tracing is the skill the exam tests here, and the habit carries over to every hard tracing question you will meet.

AP Computer Science A

Recursion

About 12 minutes to read

## How to use this guide

Read it in order the first time, because the ideas build on each other. The base case and the recursive case define a recursive method, the call stack explains what happens when it runs, and the tracing habit is what lets you answer exam questions about any of it. Exam questions almost always hand you a method and ask what it returns or prints.

After the first read, use the trap boxes and the tables to review the distinctions the exam tests most often. Finish with the practice questions, then complete the recall check on the last page out loud and note any items you cannot explain yet.

**What this unit is worth.** Unit 10 is about 5 to 7.5 percent of the AP CSA exam. It is the smallest unit, but recursion tracing is a guaranteed MCQ topic and the call-stack reasoning transfers to every hard tracing question. Students who can trace recursion by hand can trace anything.

## 10.1 Recursive Methods

A **recursive method** is a method that calls itself. That sounds circular, and it would be, except that each call works on a smaller version of the problem until it reaches a case simple enough to answer directly. Every recursive method that works has two parts, and every broken one is missing one of them or gets one of them wrong.

### The base case

The **base case** is the input the method can answer without calling itself. It is the exit. In a factorial method, the base case is usually `n <= 1`, which returns 1 directly. Without a base case, the method calls itself until Java runs out of stack memory and throws a `StackOverflowError`.

A base case written in the code is not enough on its own. It has to be reachable. A method whose base case is `n == 0` but whose recursive call passes `n + 1` moves away from the base case on every call, so the base case never runs. A method that steps by 2 toward a base case of `n == 1` can also step right over it when `n` is even. Either way the result is the same: infinite recursion.

### The recursive case

The **recursive case** is the part that calls the method again with a smaller argument and uses the result. In factorial, the recursive case is `return n * factorial(n - 1)`. Three things have to be true about it. It calls the same method. The argument is closer to the base case than the one that came in. And the value that comes back from the recursive call is actually used to build the answer. If the recursive call's return value is thrown away, the method is usually not computing what you think it is.

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

**Trap.** The most common broken recursive method has a base case that looks fine but never runs. Check the direction of the argument. If the base case is at the bottom (`n == 0`) and the call passes `n + 1`, every call moves farther from the exit. Trace two or three calls with real numbers and watch where the argument goes.

## How the call stack works

When a method calls another method, Java pauses the caller and starts the new call. A recursive call is no different, except the "other method" is itself. Each call gets its own **stack frame** holding its own copies of the parameters and local variables, so `factorial(4)` and `factorial(3)` each have their own `n` and never share it. The frames pile up while the calls go deeper. When a call finally hits the base case and returns, the frames **unwind** in reverse: each paused call resumes exactly where it left off, now holding the value its recursive call returned.

```
factorial(3) calls factorial(2)    3 waits with n = 3
  factorial(2) calls factorial(1)  2 waits with n = 2
    factorial(1) hits the base case and returns 1
  factorial(2) resumes: 2 * 1 = 2, returns 2
factorial(3) resumes: 3 * 2 = 6, returns 6
```

If the base case is never reached, the frames keep piling up until Java runs out of stack memory. That is the `StackOverflowError`. It is a runtime error, not a compile-time error. The compiler will not warn you that your recursion never stops.

**Trap.** Students often imagine the recursive calls sharing one `n`. They do not. Each call has its own `n`, its own frame, and its own place to resume. When you trace, give every call its own line and its own value of `n`.

## How to trace a recursive method

Write it out. List each call on its own line, indent one step deeper for each nested call, and write the return value next to each line as the calls finish. Do the way down first, all the way to the base case, then fill in the way back up. For a method that prints, write what prints at the moment it prints, in order. Most tracing errors come from guessing the pattern instead of writing out the calls.

## Common recursive patterns

The exam reuses a small set of shapes. Learn to recognize them, then still trace them by hand.

| PATTERN         | BASE CASE                                     | RECURSIVE CASE                                                    | CHECK WITH                                |
|-----------------|-----------------------------------------------|-------------------------------------------------------------------|-------------------------------------------|
| Factorial       | <code>n &lt;= 1</code> returns 1              | <code>return n * factorial(n - 1)</code>                          | <code>factorial(4)</code><br>= 24         |
| Sum of 1 to n   | <code>n == 1</code> returns 1                 | <code>return n + sum(n - 1)</code>                                | <code>sum(4)</code> = 10                  |
| Countdown print | <code>n == 0</code> returns, printing nothing | print <code>n</code> , then call<br><code>countdown(n - 1)</code> | <code>countdown(3)</code><br>prints 3 2 1 |
| Countup print   | <code>n == 0</code> returns, printing nothing | call <code>countup(n - 1)</code> , then print<br><code>n</code>   | <code>countup(3)</code><br>prints 1 2 3   |

The countdown and countup pair is the single most tested distinction in this unit. The only difference is whether the print comes before or after the recursive call, and the output order flips completely. Statements before the recursive call run on the way down, in call order. Statements after it run on the way up, in reverse order, because each call has to wait for its recursive call to finish before it can continue.

```
// prints 3 2 1 : the print runs BEFORE the recursive call
public static void countdown(int n)
{
    if (n == 0)
    {
        return;
    }
    System.out.print(n + " ");
    countdown(n - 1);
}

// prints 1 2 3 : the print runs AFTER the recursive call
public static void countup(int n)
{
    if (n == 0)
    {
        return;
    }
    countup(n - 1);
    System.out.print(n + " ");
}
```

**Trap.** When the print sits after the recursive call, the first call prints last. Read the method top to bottom, but remember that the recursive call must finish completely before the lines below it run. If you read `countup` as printing 3 2 1, you read the call order instead of the print order.

## Recursion vs iteration

Anything a loop can do, recursion can do, and the reverse is true as well. Recursion is not automatically faster or more elegant. Each recursive call costs a stack frame and the time to set it up, so a recursive solution is usually a little slower and uses more memory than the equivalent loop. It also carries a risk that loops do not have in the same form: runaway recursion ends in a `StackOverflowError`. On the exam you are not asked to pick the better approach. You are asked to trace what a given method does, so the practical move is to be fluent at reading both.

| ASPECT              | RECURSION                                               | ITERATION (LOOP)                   |
|---------------------|---------------------------------------------------------|------------------------------------|
| Speed and memory    | Slower per step; one stack frame per call               | Usually faster; fixed memory       |
| Main runaway risk   | <code>StackOverflowError</code> from a missed base case | Infinite loop from a bad condition |
| What the exam tests | Tracing calls, returns, and print order                 | Tracing loop variables and bounds  |

## Confusions That Cost Points

| PAIR                                        | HOW TO KEEP THEM STRAIGHT                                                                                                                                                                                                                         |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Base case exists vs base case is reached    | A base case in the code means nothing if the argument never gets there. <code>n + 1</code> moves away from a base case of <code>n == 0</code> , and <code>n - 2</code> steps over a base case of <code>n == 1</code> when <code>n</code> is even. |
| Printing before vs after the recursive call | Before the call prints on the way down, in call order: 3 2 1. After the call prints on the way up, in reverse: 1 2 3. The position of the print relative to the call is the whole question.                                                       |
| Off-by-one in the base case                 | <code>sum(0)</code> with a base case of <code>n == 1</code> never stops, because 0 steps to -1, -2, and onward, away from 1. Trace from the actual argument, not from 1 by habit.                                                                 |
| "Recursion is faster or better"             | Nothing about recursion guarantees speed. Each call costs a stack frame, and deep recursion can overflow the stack. Trace the method instead of assuming.                                                                                         |
| One recursive call vs two                   | A method like <code>f(n - 1) + f(n - 2)</code> branches at every level, so the work multiplies. Trace both branches separately and add carefully: <code>f(2) = 1</code> , <code>f(3) = 2</code> , <code>f(4) = 3</code> .                         |

# Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the following method.

```
public static int mystery(int n)
{
    if (n == 0)
    {
        return 0;
    }
    return n + mystery(n - 1);
}
```

What is returned by the call `mystery(4)` ?

- 1. 8
- 2. 12
- 3. 10
- 4. 9

2. Consider the following method.

```
public static void countdown(int n)
{
    if (n == 0)
    {
        return;
    }
    System.out.print(n + " ");
    countdown(n - 1);
}
```

What is printed by the call `countdown(3)` ?

- 1. 3 2 1
- 2. 1 2 3
- 3. 3 2 1 0
- 4. 0 1 2 3

3. Consider the following method.

```
public static void countup(int n)
{
    if (n == 0)
    {
        return;
    }
    countup(n - 1);
    System.out.print(n + " ");
}
```

What is printed by the call `countup(3)`?

- 1. 3 2 1
- 2. 0 1 2 3
- 3. 1 2 3
- 4. 3 2 1 0

4. Consider the following method.

```
public static int f(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * f(n - 2);
}
```

What is returned by the call `f(6)`?

- 1. 48
- 2. 12
- 3. 24
- 4. 720

5. Consider the following method.

```
public static int calc(int n)
{
    if (n == 0)
    {
        return 0;
    }
    return 1 + calc(n + 1);
}
```

Which of the following is true about the call `calc(5)` ?

1. It returns 5.
2. It returns 0.
3. It causes a `StackOverflowError`.
4. It returns -5.

6. Consider the following method.

```
public static int sum(int n)
{
    if (n == 1)
    {
        return 1;
    }
    return n + sum(n - 1);
}
```

What happens when `sum(0)` is called?

1. It returns 0.
2. It causes a `StackOverflowError`.
3. It returns 1.
4. It returns -1.

7. Consider the following method.

```
public static int f(int n)
{
    if (n <= 1)
    {
        return n;
    }
    return f(n - 1) + f(n - 2);
}
```

What is returned by the call `f(4)` ?

1. 5
2. 3
3. 4
4. 6

**8.** Which of the following statements about recursion is true?

1. A recursive method must contain a loop.
2. A recursive method always runs faster than an equivalent loop.
3. A recursive method must call itself exactly once.
4. Any problem solvable with a loop can also be solved with recursion.

## Answer Key

- 1. C.** `mystery(4) = 4 + mystery(3)`; `mystery(3) = 3 + mystery(2)`; `mystery(2) = 2 + mystery(1)`; `mystery(1) = 1 + mystery(0)`; `mystery(0) = 0`. Unwinding:  $1 + 0 = 1$ ,  $2 + 1 = 3$ ,  $3 + 3 = 6$ ,  $4 + 6 = 10$ . A drops a level of the call chain. B multiplies 4 by 3 instead of following the addition in the code. D treats `mystery(1)` as 0 instead of 1, giving  $4 + 3 + 2 + 0 = 9$ .
- 2. A.** `countdown(3)` prints 3, then calls `countdown(2)`, which prints 2 and calls `countdown(1)`, which prints 1 and calls `countdown(0)`. The `n == 0` call hits the base case and returns before printing anything, so the output is 3 2 1. B reads it as if the print came after the recursive call, which would give 1 2 3. C and D print a 0 from the base case, but the base case returns before reaching the print.
- 3. C.** `countup(3)` must finish `countup(2)` before it prints 3; `countup(2)` must finish `countup(1)` before it prints 2; `countup(1)` calls `countup(0)`, which returns at once. The prints then happen on the way back up: 1, 2, 3. A reads the call order instead of the print order. B and D print a 0 from the base case, which returns before printing.
- 4. A.** `f(6) = 6 * f(4)`; `f(4) = 4 * f(2)`; `f(2) = 2 * f(0)`; `f(0) = 1` because  $0 \leq 1$ . Unwinding:  $2 * 1 = 2$ ,  $4 * 2 = 8$ ,  $6 * 8 = 48$ . B stops the trace a level early. C computes  $4 * 3 * 2 * 1$ , which decrements by 1 instead of 2. D computes  $6!$ , ignoring the `n - 2` step entirely.
- 5. C.** `calc(5) = 1 + calc(6)`; `calc(6) = 1 + calc(7)`, and so on. The argument grows on every call, so `n == 0` is never reached. The frames pile up until Java runs out of stack memory and throws a `StackOverflowError`. A assumes the argument counts down to 0, but the call passes `n + 1`. B is the base case's return value, which never runs. D has no basis in the code; nothing negates the result.
- 6. B.** `sum(0)` skips the base case, since 0 is not 1, and calls `sum(-1)`, then `sum(-2)`, and so on. The argument moves away from 1 forever, so the base case never runs and the stack overflows. A is what the method would return if the base case were `n == 0`. C is the base case's return value, but `sum(0)` never reaches the base case. D adds  $0 + (-1)$  and stops, but the recursion does not stop.
- 7. B.** `f(4) = f(3) + f(2)`. `f(3) = f(2) + f(1)`, and each `f(2) = f(1) + f(0) = 1 + 0 = 1`. So `f(3) = 1 + 1 = 2`, and `f(4) = 2 + 1 = 3`. A is `f(5)`, one level deeper than asked. C and D come from miscounting the branches; every `f(2)` must be expanded on its own.
- 8. D.** Recursion and iteration are equivalent in what they can compute; the repeated self-calls replace the loop. A is false because a recursive method needs no loop at all. B is false because each call costs a stack frame, so recursion is often slower, not faster. C is false because a method may call itself more than once, as in `f(n - 1) + f(n - 2)`.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the Recursion deck has flashcards for it and more practice questions use the same kinds of traps.

# One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the Recursion deck and let spaced review bring them back over the next few days.

- Define a recursive method in one sentence.
- Name the two parts every working recursive method needs and say what each one does.
- Explain what a stack frame holds and why each recursive call needs its own.
- Describe what happens, step by step, when a method calls itself.
- Give two ways a recursive method can recurse forever, and name the error Java throws.
- Trace `factorial(4)` by hand, writing every call and every return value.
- State what `countdown(3)` and `countup(3)` print, and explain why the order differs.
- Trace `mystery(4)` for the `n + mystery(n - 1)` method and show the full call sequence.
- Explain why `sum(0)` never terminates when the base case is `n == 1`.
- Explain why `calc(5)` never terminates when the recursive call passes `n + 1`.
- Trace `f(4)` for the `f(n - 1) + f(n - 2)` method, expanding both branches.
- State one real difference between recursion and iteration for speed, memory, and runaway risk.

**Where to go next.** Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the Recursion deck under AP Computer Science A. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

## Key terms for this unit

Recursive method, Base case, Recursive case, Call stack, Stack frame, Infinite recursion, `StackOverflowError`, Tracing, Unwinding, Countdown pattern, Countup pattern.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 10. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.