Recursion Notes

AP Computer Science A
[AI generated from my notes outline]

1. Recursive Sequences

A recursive sequence defines each term using previous terms.

Example: $a_1 = 2$, $a_n = a_{n-1} + 3$

Find the 3rd term:

$a_1 = 2$

$a_2 = a_1 + 3 = 2 + 3 = 5$

$a_3 = a_2 + 3 = 5 + 3 = 8$

Find the 10th term: Keep going! $a_4 = 11$, $a_5 = 14$, ... $a_{10} = 29$

Or notice the pattern: $a_n = 2 + 3(n-1) = 3n - 1$

2. Fibonacci Sequence

Definition: $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Find F_{10}:

$F_1=1,\ F_2=1,\ F_3=2,\ F_4=3,\ F_5=5,\ F_6=8,\ F_7=13,\ F_8=21,\ F_9=34,\ F_{10}=55$

We computed this bottom-up: start from known values, build forward.

3. Recursive Functions

Recursive functions work the same way: a function that calls itself with a smaller input.

Every recursive function needs two things:

1. **Base case:** When to stop (return a value without recursing)
2. **Recursive case:** Call itself with a simpler input

4. A Simple Recursive Function

```
public int f(int n) {
    if (n <= 0) return 0;       // base case
    return 3 + f(n - 1);        // recursive case
}
```

Trace f(4):

```
f(4) = 3 + f(3) = 3 + 3 + f(2) = 3 + 3 + 3 + f(1) = 3 + 3 + 3 + 3 + f(0)
f(0) = 0  →  total = 12
```

Why n <= 0 instead of n == 0? Defensive programming. If someone calls f(-5), we don't want infinite recursion.

5. Fibonacci as a Function

```
public int fib(int n) {
    if (n <= 2) return 1;
```

```
    return fib(n-1) + fib(n-2);
}
```

Top-down evaluation is slow: fib(10) calls fib(9) and fib(8). But fib(9) also calls fib(8). We recompute the same values over and over!

Bottom-up is smart: When tracing by hand, start from the base cases and work up. This is how we found F_{10} = 55 quickly.

Tip: When tracing recursion, unwind in your head from base case back up.

6. Recursion with Division

```
public int g(int n) {
    if (n < 1) return 0;
    return 1 + g(n / 2);
}
```

Trace g(16):

```
g(16) = 1 + g(8) = 1 + 1 + g(4) = 1 + 1 + 1 + g(2) = 1 + 1 + 1 + 1 + g(1)
g(1) = 1 + g(0) = 1 + 0 = 1
Total: 1 + 1 + 1 + 1 + 1 = 5
```

This counts how many times you can divide by 2 (roughly $\log_2 n$).

7. Print Before Recursion (Tail Recursive Style)

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

countdown(5) outputs: 5 4 3 2 1

The print happens on the way DOWN the recursive calls.

8. Print After Recursion

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

countup(5) outputs: 1 2 3 4 5

The print happens on the way BACK UP from the recursive calls.

9. The Key Difference

Print before recursive call: Output happens going DOWN (forward order)

Print after recursive call: Output happens coming BACK UP (reverse order)

Think of it like walking into a cave:

```
Print first  →  you shout on the way IN
Print second →  you shout on the way OUT
```

This pattern applies to any work, not just printing: processing data before vs. after the recursive call changes the order of operations.

Summary

✓ Every recursive function needs a base case and a recursive case

✓ Use n <= 0 rather than n == 0 for safety

✓ Trace by unwinding from base case back up

✓ Action before recursive call → forward order

✓ Action after recursive call → reverse order