

Unit 4: Iteration

Unit 4 is about loops, the code that repeats. It covers while loops and for loops, how to choose between them, how to walk through a string one character at a time, how nested loops behave, and how to count how many times a loop runs. Almost every algorithm you write from here on is a loop, so this unit pays off in every unit after it.

AP Computer Science A

Iteration

About 14 minutes to read

How to use this guide

Read it in order the first time because the topics build on each other. The while loop teaches you how repetition works, the for loop packages that pattern for counting, string algorithms apply loops to real data, nested loops stack them, and code analysis asks you to reason about how much work a loop does. Exam questions almost always give you a loop and ask you to trace it, so do not just read the code samples. Trace each one by hand, writing down the variable values at each pass.

After the first read, use the trap boxes and the tables to review the distinctions that exam questions test 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 4 is about 15 to 17.5 percent of the AP CSA exam, tied with Unit 3 as the heaviest. Loop tracing is the single most tested skill on the multiple-choice section, and every array and ArrayList algorithm in Units 6 through 8 is a loop. If tracing feels slow now, practice it until it is mechanical.

4.1 while Loops

A **while loop** repeats a block of code as long as a condition stays true. The condition is checked *before* each pass, including the first one. If the condition is false on the first check, the body runs zero times. The three parts you must see are the **initialization** before the loop, the **condition** in parentheses, and the **update** somewhere inside the body that moves the condition toward false.

```
int count = 1;           // initialization
while (count <= 5)       // condition, checked before each pass
{
    System.out.println(count);
    count++;             // update: without this, the loop never ends
}
```

Trace it. count starts at 1. The check $1 \leq 5$ is true, so 1 prints and count becomes 2. The check $2 \leq 5$ is true, so 2 prints and count becomes 3. This continues until 5 prints and count becomes 6. The check $6 \leq 5$ is false, so the loop stops. The output is 1, 2, 3, 4, 5. Notice the condition is checked one more time than the body runs.

Trap. An **infinite loop** happens when the condition can never become false, almost always because the update is missing or moves the wrong way. The loop below prints 1 forever because count never changes. When you trace, always confirm the update actually reaches the condition's breaking point.

```
int i = 1;
while (i <= 5)
{
    System.out.println(i);
    // no update: i stays 1, so the loop never ends
}
```

Sentinel values

A **sentinel value** is a special input that ends the loop. It is useful when you do not know in advance how many inputs are coming. The classic pattern reads values until the sentinel appears, and the sentinel itself is not processed.

```
Scanner input = new Scanner(System.in);
int total = 0;
int num = input.nextInt();
while (num != -1)    // -1 is the sentinel: it ends the loop
{
    total += num;
    num = input.nextInt();
}
```

The sentinel value -1 is never added to total. Two things must be true for this pattern to work. The first read happens before the loop so the condition has something real to check, and the read repeats at the bottom of the body so the loop can actually reach the sentinel.

4.2 for Loops

A **for loop** packages the same three parts of a while loop into one header. The **initialization** runs once at the start, the **condition** is checked before each pass, and the **update** runs after each pass, before the condition is checked again. Like a while loop, if the condition is false on the first check, the body runs zero times.

PART	WHAT IT DOES	EXAMPLE IN FOR (INT I = 0; I < 5; I++)
Initialization	Runs once before the loop starts.	int i = 0
Condition	Checked before each pass. False stops the loop.	i < 5
Update	Runs after each pass of the body.	i++

```
for (int i = 0; i < 5; i++)
{
    System.out.println(i);
}
```

Trace it. i starts at 0, 0 < 5 is true, so 0 prints and i becomes 1. The body prints 1, 2, 3, 4. After 4 prints, i becomes 5, 5 < 5 is false, and the loop stops. The output is 0, 1, 2, 3, 4. The loop runs 5 times.

Trap. The **off-by-one error** is the most tested mistake in this unit. With i starting at 0, the condition i < 5 gives 5 passes (0 through 4) while i <= 5 gives 6 passes (0 through 5). When you count iterations, list the actual values of the loop variable instead of guessing from the numbers in the header.

HEADER	VALUES OF I	PASSES
for (int i = 0; i < 5; i++)	0, 1, 2, 3, 4	5
for (int i = 0; i <= 5; i++)	0, 1, 2, 3, 4, 5	6
for (int i = 1; i < 5; i++)	1, 2, 3, 4	4
for (int i = 0; i < 10; i += 2)	0, 2, 4, 6, 8	5

for vs while

Use a **for loop** when you know the number of iterations before the loop starts, which is almost always a counting situation. Use a **while loop** when the stopping condition depends on something that happens inside the loop, like reading input until a sentinel appears or repeating until a value crosses a threshold. Either loop can do the other's job, but the exam rewards the natural choice, and mixing them up usually means the update or the condition is written awkwardly.

Trap. The loop variable declared in the header, like int i, exists only inside the loop. Code after the loop cannot use i. If you need the final value of a counter after the loop ends, declare it before the loop and update it inside.

4.3 Developing Algorithms Using Strings

Loop algorithms on strings come in two flavors: **traversing**, which reads each character in order, and **building**, which assembles a new string one character at a time. Both use the same index pattern. A string of length n has valid indexes 0 through $n - 1$, so the loop runs i from 0 while $i < \text{word.length}()$, and $\text{word.substring}(i, i + 1)$ grabs the single character at position i .

```
String word = "cat";
for (int i = 0; i < word.length(); i++)
{
    System.out.println(word.substring(i, i + 1));
}
```

Trace it. $\text{word.length}()$ is 3, so i takes the values 0, 1, 2. At $i = 0$, $\text{substring}(0, 1)$ is "c". At $i = 1$, $\text{substring}(1, 2)$ is "a". At $i = 2$, $\text{substring}(2, 3)$ is "t". The output is c, a, t on separate lines. The last valid index is $\text{length}() - 1$, which is why the condition uses $<$ and not $\leq$.

Trap. Using $i \leq \text{word.length}()$ throws a `StringIndexOutOfBoundsException`. At $i = \text{word.length}()$, $\text{substring}(i, i + 1)$ asks for a character past the end of the string. The traversal pattern is always $i = 0$ with $i < \text{length}()$. If you see $\leq$ paired with $\text{length}()$ in an exam option, that option is wrong.

Building strings in a loop

To build a string, start with the **empty string** and add one piece per pass. Reversing a string is the standard example, and it runs the index backward.

```
String word = "abc";
String rev = "";
for (int i = word.length() - 1; i >= 0; i--)
{
    rev += word.substring(i, i + 1);
}
```

Trace it. i starts at 2. $\text{substring}(2, 3)$ is "c", so rev becomes "" + "c" = "c". At $i = 1$, $\text{substring}(1, 2)$ is "b", so rev becomes "cb". At $i = 0$, $\text{substring}(0, 1)$ is "a", so rev becomes "cba". Then i becomes -1 , the condition fails, and rev is "cba". Starting from "" matters: if rev started as anything else, that leftover text would sit at the front of the answer.

4.4 Nested Iteration

A **nested loop** is a loop whose body contains another loop. The **outer loop** controls the big steps and the **inner loop** runs to completion for every single pass of the outer loop. That last sentence is the whole topic. For each value of the outer variable, the inner loop starts over from its initialization and runs all of its own iterations.

```
for (int row = 1; row <= 3; row++)
{
    for (int col = 1; col <= 2; col++)
    {
        System.out.print(row + " " + col + " ");
    }
    System.out.println();
}
```

Trace it. The outer loop sets row = 1. The inner loop then runs completely: col = 1 prints "11 ", col = 2 prints "12 ", and col = 3 fails the condition. The println moves to the next line. The outer loop sets row = 2 and the inner loop runs completely again, printing "21 " and "22 ". Same for row = 3. The output is:

```
11 12
21 22
31 32
```

To count total inner-body executions, **multiply** the outer count by the inner count: 3 outer passes times 2 inner passes = 6. When the inner bound depends on the outer variable, multiplication no longer works and you must add up the inner counts pass by pass, as in practice question 8.

Trap. The two most common errors are adding the loop counts instead of multiplying them, and assuming the inner loop picks up where it left off. It does not. The inner loop re-initializes every time the outer body starts. In the example, col resets to 1 for each new row, which is why every row starts with col = 1.

4.5 Informal Code Analysis

Informal code analysis means reasoning about how many times a loop runs without executing it. The exam does not ask for Big-O notation in this unit. It asks questions like "how many times does the body execute" or "which change makes the loop run fewer times." The method is the same as tracing, but you reason about the pattern instead of listing every pass.

Start by identifying what controls the count. A loop that counts i from 0 to $n - 1$ runs about **n times**, no matter what the body does. A nested loop with n outer passes and n inner passes runs about **n times n** , which is n squared. A loop that halves its value each pass, like the one below, runs far fewer times than a loop that steps by one.

```
int n = 32;
int steps = 0;
while (n > 1)
{
    n = n / 2;
    steps++;
}
System.out.println(steps);
```

Trace the halving. n goes 32, 16, 8, 4, 2, 1. The condition $n > 1$ is true for 32, 16, 8, 4, and 2, so steps reaches 5, then n becomes 1 and the loop stops. Doubling the starting value to 64 would add only one more pass. That is the kind of comparison the exam tests: two loops, same body, and you judge which one does more work.

Trap. Count **iterations of the body**, not lines of code inside it. A body with five statements that runs n times does n passes, not $5n$. And watch for a loop variable that is modified inside the body on top of the header update. Each pass then moves the variable twice, which cuts the iteration count roughly in half and surprises anyone who only reads the header.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
< vs <= in the condition	With i starting at 0, < n runs n times and <= n runs n + 1 times. List the actual values of the loop variable to be sure.
for vs while	for counts a known number of iterations. while repeats until something changes, like reaching a sentinel. Either can work, but the wrong choice hides the update.
Missing update vs wrong update	Missing update means the condition never changes, which is an infinite loop. An update that moves the wrong way, like i-- when the condition needs i to grow, is also infinite.
Loop variable scope	int i in the header exists only inside the loop. Using i after the loop is a compile error. Declare the variable before the loop if you need it afterward.
Modifying the loop variable in the body	An extra i++ or i-- inside the body stacks on top of the header update, so the loop steps by 2 or steps backward. Read the whole body, not just the header.
Nested loop counts: add vs multiply	Multiply outer passes by inner passes when the inner bound is fixed. Add the inner counts pass by pass when the inner bound depends on the outer variable.
String traversal bounds	Indexes run 0 to length() - 1. The traversal condition is i < word.length(). Using <= throws StringIndexOutOfBoundsException.
Building strings without ""	Always initialize the result to the empty string. Starting from null or from the original string puts junk at the front of the answer.
Sentinel included in the result	The sentinel ends the loop and is never processed. Check that the loop condition excludes it before the body runs.

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 code segment.

```
int i = 1;
int sum = 0;
while (i <= 4)
{
    sum += i;
    i += 2;
}
System.out.println(sum);
```

What is printed as a result of executing the code segment?

- 1. 6
- 2. 4
- 3. 10
- 4. 1

2. Consider the following code segment.

```
for (int k = 5; k > 0; k -= 2)
{
    System.out.print(k + " ");
}
```

What is printed as a result of executing the code segment?

- 1. 5 3 1
- 2. 5 3 1 0
- 3. 4 2 0
- 4. 5 3 1 -1

3. Consider the following code segment.

```
String s = "rain";
String result = "";
for (int i = 0; i < s.length(); i++)
{
    result = s.substring(i, i + 1) + result;
}
System.out.println(result);
```

What is printed as a result of executing the code segment?

- 1. rain
- 2. niar
- 3. nair
- 4. iar

4. Consider the following code segment.

```
int count = 0;
for (int i = 1; i <= 3; i++)
{
    for (int j = 1; j <= 4; j++)
    {
        count++;
    }
}
System.out.println(count);
```

What is printed as a result of executing the code segment?

1. 7
2. 12
3. 24
4. 4

5. Which of the following code segments will **never terminate** (that is, it is an infinite loop)?

```
(A)
int n = 0;
while (n < 10)
{
    n++;
}

(B)
int n = 10;
while (n > 0)
{
    n -= 2;
}

(C)
int n = 1;
while (n != 0)
{
    n++;
}

(D)
for (int n = 5; n > 0; n--)
{
    n--;
}
```

6. Consider the following code segment.

```
for (int i = 0; i <= 10; i += 2)
{
    System.out.println(i);
}
```

How many times is the body of the loop executed?

- 1. 5
- 2. 6
- 3. 10
- 4. 11

7. Consider the following code segment.

```
int total = 0;
int n = 235;
while (n != 0)
{
    total += n % 10;
    n = n / 10;
}
System.out.println(total);
```

What is printed as a result of executing the code segment?

1. 235
2. 10
3. 532
4. 5

8. Consider the following code segment.

```
int count = 0;
for (int i = 1; i <= 4; i++)
{
    for (int j = 1; j <= i; j++)
    {
        count++;
    }
}
System.out.println(count);
```

What is printed as a result of executing the code segment?

1. 16
2. 10
3. 8
4. 4

Answer Key

- 1. B.** Trace the loop. $i = 1$: sum becomes 1, i becomes 3. $i = 3$: sum becomes 4, i becomes 5. Now $5 \leq 4$ is false, so the loop stops with sum = 4. A assumes i increases by 1 each pass and adds $1 + 2 + 3$, but the update is $i += 2$. C adds $1 + 2 + 3 + 4$, which combines the wrong step with the wrong bound. D is the value of sum after only the first pass, which means the loop was stopped early.
- 2. A.** Trace k . $k = 5$ prints "5 ", $k = 3$ prints "3 ", $k = 1$ prints "1 ", then k becomes -1 and the condition $k > 0$ is false. The output is "5 3 1 ". B adds an extra "0", which would require the condition to allow $k = 0$, but $k > 0$ fails at 0. C starts from the wrong initial value and steps the wrong way. D includes -1 , which mistakes the order of the check: the condition is tested before each pass, so -1 never prints.
- 3. B.** Each new character is placed in front of result, so the string builds backward. $i = 0$ puts "r" first, $i = 1$ makes "ar", $i = 2$ makes "iar", $i = 3$ makes "niar". A is the original string, which would require $\text{result} = \text{result} + \text{char}$ instead of $\text{char} + \text{result}$. C scrambles the middle characters, which comes from not tracking the prepend order carefully. D drops the first character, which would happen only if the loop started at $i = 1$.
- 4. B.** The inner loop runs to completion for every outer pass. The outer loop runs 3 times and the inner loop runs 4 times per outer pass, so $\text{count}++$ executes $3 \times 4 = 12$ times. A adds $3 + 4$ instead of multiplying. C doubles the correct answer, which comes from running the inner loop twice per outer pass by mistake. D counts only the inner passes of a single outer iteration.
- 5. C.** In C, n starts at 1 and increases every pass: 1, 2, 3, and on forever. It never equals 0, so the condition $n != 0$ never becomes false. A terminates when n reaches 10. B terminates when n reaches 0 (10, 8, 6, 4, 2, 0). D looks suspicious because $n--$ appears twice, but trace it: n goes 5, 3, 1, -1 and the condition fails at -1 , so it terminates. The trap in D is reading only the header.
- 6. B.** List the values of i : 0, 2, 4, 6, 8, 10. That is 6 passes. A gets 5 by forgetting that i starts at 0 and counting only 2 through 10. C divides 10 by 1 instead of by the step of 2. D counts the integers from 0 to 10, which ignores the step of 2 entirely.
- 7. B.** This loop sums the digits of 235. $n = 235$: $\text{total} += 5$ gives 5, n becomes 23. $n = 23$: $\text{total} += 3$ gives 8, n becomes 2. $n = 2$: $\text{total} += 2$ gives 10, n becomes 0, and the loop stops. The answer is 10. A prints n instead of total . C is the digits in reverse order, which is what you would get if the code built a string instead of adding. D is only the first digit extracted, which means the loop stopped after one pass.
- 8. B.** The inner bound depends on the outer variable, so multiplication does not work. Count pass by pass: $i = 1$ gives 1 inner pass, $i = 2$ gives 2, $i = 3$ gives 3, $i = 4$ gives 4. The total is $1 + 2 + 3 + 4 = 10$. A multiplies 4×4 , which treats the inner bound as fixed at 4 for every outer pass. C multiplies 2×4 , which has no basis in the code. D counts only the outer passes.

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 AP Computer Science A 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 AP Computer Science A deck and let spaced review bring them back over the next few days.

- Explain what happens before each pass of a while loop, and name the three parts every while loop needs.
- Define a sentinel value and explain why the sentinel is never processed by the loop body.
- Describe two ways a loop becomes infinite.
- State what each part of a for loop header does and when it runs.
- Explain the off-by-one error and show how $< n$ and $\leq n$ differ when i starts at 0.
- State when to choose a for loop and when to choose a while loop.
- Explain why a loop variable declared in a for header cannot be used after the loop.
- Write the standard string traversal loop from memory and explain why the bound is `length() - 1`.
- Explain how to reverse a string with a loop and why the result starts as "".
- Explain what the inner loop does on each pass of the outer loop in a nested loop.
- State when to multiply loop counts and when to add them up pass by pass.
- Explain how to compare two loops informally by counting iterations, without Big-O notation.
- Explain why a loop that halves its value each pass runs far fewer times than one that steps by one.

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 AP Computer Science A deck. 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

while loop, Initialization, Condition, Update, Infinite loop, Sentinel value, for loop, Off-by-one error, Loop variable scope, String traversal, substring, String building, Empty string, Nested loop, Outer loop, Inner loop, Informal code analysis, Iterations.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 4. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.