# Loop Invariants

**Proving that an algorithm with a loop is correct**

Last modified: 2026-11-18 21:58:23 (PST)

**Source**

The topics of this chapter follow Kameron Decker Harris's Fall 2026 worksheet on loop invariants:

- what a loop invariant is;
- the three parts of a loop invariant proof: initialization, maintenance and termination;
- proving that a loop stops;
- choosing an invariant strong enough to give the result;
- invariants for a loop inside another loop.

Everything else is new: the algorithms, the exercises, the examples, the theorems and their proofs.

## Preconditions and postconditions

**Exercise 1** (Tracing a square-root loop)**.** The algorithm `root` takes a non-negative integer (Definition: Non-negative integers) $n$:

```
procedure root(n)        // n is a non-negative integer
    r := 0
    while (r + 1) * (r + 1) <= n
        r := r + 1
    return r
```

Run `root(20)` by hand. Each time the condition after `while` is tested, write down the

value of `r` and whether the condition is true, and say which number `root(20)` returns.

Solution

*Solution* 1.

| Test | `r` | $(r+1)^2$ | $(r+1)^2 \leq 20$? |
|---|---|---|---|
| 1 | 0 | 1 | true |
| 2 | 1 | 4 | true |
| 3 | 2 | 9 | true |
| 4 | 3 | 16 | true |
| 5 | 4 | 25 | false |

The condition is false at the fifth test, so the loop stops there, and `root(20)` returns 4. The answer satisfies $4^2 = 16 \leq 20 < 25 = 5^2$: 4 is the largest integer whose square is at most 20.

*Remark* 1 (How to read the pseudocode in this chapter). The algorithms (Definition: Algorithm) in this chapter are written in pseudocode, not in a particular programming language:

- `x := e` sets the variable `x` to the value of the expression `e`;
- the indented lines under `while` are the *body* of the loop, and a `while` loop tests its condition before every pass through the body, running the body when the condition is true and stopping when it is false;
- `for i := a to b` runs its body once for each of $i = a, a+1, \ldots, b$, and not at all if $a > b$;
- `A[1..n]` is an array of $n$ entries `A[1]`, …, `A[n]`;
- text after `//` is a comment, which the algorithm ignores.

In Solution 1, the body of `root` runs four times, and its condition is tested five times: once before each pass through the body, and once more when the loop stops.

**Definition 1** (Floor)**.** The **floor** $\lfloor x \rfloor$ of a real number $x$ (Wikipedia contributors 2026b) is the largest integer that is at most $x$.

**Example 1** (Some floors)**.** By Definition 1,

- $\lfloor 3.7 \rfloor = 3$, since 3 is an integer at most 3.7 and 4 is not;
- $\lfloor 5 \rfloor = 5$, since 5 is already an integer;
- $\lfloor -1.5 \rfloor = -2$, since $-2$ is at most $-1.5$ and $-1$ is not.

**Definition 2** (Precondition)**.** A **precondition** of an algorithm (Definition: Algorithm) is a condition on its input that the algorithm assumes is true when it starts.

**Definition 3** (Postcondition)**.** A **postcondition** of an algorithm is a condition on its input and its output that is required to be true when the algorithm stops, provided that its precondition (Definition 2) was true when it started.

**Example 2** (What `root` promises)**.** For `root` from Exercise 1, the precondition (Definition 2) is

> $n$ is a non-negative integer,

and the postcondition (Definition 3) is

> the returned value $r$ is a non-negative integer with $r^2 \leq n < (r+1)^2$.

For a non-negative integer $r$, the two inequalities $r^2 \leq n < (r+1)^2$ say that $r \leq \sqrt{n} < r+1$, so the postcondition says that $r = \lfloor \sqrt{n} \rfloor$ (Definition 1), the largest integer whose square is at most $n$. Solution 1 checks it for $n = 20$, where $r = 4$.

**Definition 4** (Correct algorithm)**.** An algorithm is **correct** for a given precondition (Definition 2) and postcondition (Definition 3) if, for every input that satisfies the precondition, the algorithm stops, and the postcondition is true when it stops.

**Example 3** (A version of `root` that is not correct)**.** Change `r := 0` in `root` to `r := 1`, and keep the rest. On the input $n = 0$, which satisfies the precondition of Example 2, the condition $(1 + 1)^2 \leq 0$ is false at the first test, so the loop body never runs and the algorithm returns $r = 1$. But $1^2 = 1 > 0$, so the postcondition $r^2 \leq n$ is false. One input is enough to show that this version is not correct (Definition 4), even though it returns the right value for every $n \geq 1$.
To show that an algorithm *is* correct, we need an argument that covers every input at once.

## Loop invariants

**Exercise 2** (One pass through the body of `root`)**.** In `root` (Exercise 1), suppose that at some test of the condition after `while`, `r` is an integer with $r^2 \leq n$, and the condition $(r + 1)^2 \leq n$ is true. Show that $r^2 \leq n$ is true again at the next test.

Solution

*Solution* 2. Since the condition is true, the body runs once and replaces `r` with $r' = r+1$, and then the condition is tested again. At that next test,

$$\begin{aligned} (r')^2 &= (r + 1)^2 \\ &\leq n, \qquad \text{(the condition was true)} \end{aligned}$$

so $r^2 \leq n$ is true again, for the new value of `r`.

**Definition 5** (Guard of a loop)**.** The **guard** of a `while` loop is the condition written after `while`: the loop tests it before each pass through the body, runs the body when it is true, and stops when it is false.

**Example 4** (The guard of a `for` loop)**.** The guard (Definition 5) of `root` (Exercise 1) is $(r+1)^2 \leq n$.
A `for` loop is a `while` loop written more briefly. The loop on the left does the same thing as the loop on the right, provided the body changes neither `i` nor `b`:

```
for i := a to b                i := a
    body                       while i <= b
                                   body
                                   i := i + 1
```

So the guard of `for i := a to b` is $i \leq b$, and the loop stops at the first test where $i = b + 1$, or at once if $a > b$.

**Definition 6** (Loop invariant)**.** A **loop invariant** of a loop is a statement about the loop's variables that is true every time the loop's guard (Definition 5) is tested, before the first pass through the body and after each pass, for every input that satisfies the algorithm's precondition (Definition 2).

**Example 5** (A statement that holds at every test of `root`)**.** In the run of `root(20)` in Solution 1, `r` takes the values 0, 1, 2, 3 and 4 at the five tests of the guard. The statement "$r^2 \leq n$" is true at all five tests, as a loop invariant (Definition 6) must be, and Solution 2 shows, for every $n$, that a pass through the body never turns it from true to false. The statement "$r < 4$", by contrast, is true at the first four tests but false at the fifth, so it is not a loop invariant of `root`.

**Theorem 1** (The loop invariant theorem)**.** *Let $I$ be a statement about the variables of a* `while` *loop whose guard (Definition 5) is $G$. Suppose that*

1. *$I$ is true at the first test of the guard, and*
2. *whenever $I$ and $G$ are both true at a test of the guard, $I$ is true at the next test.*

*Then $I$ is a loop invariant (Definition 6) of the loop. In particular, if the loop stops, then $I$ is true and $G$ is false when it stops.*

Proof

*Proof.* We use induction (Definition: Proof by induction) on the number of passes through the body. For each non-negative integer $k$, let $P(k)$ be the claim

> if the body runs at least $k$ times, then $I$ is true at the test of the guard that follows the first $k$ passes.

The test that follows the first 0 passes is the first test.
**Base case.** $P(0)$ is hypothesis 1.
**Inductive step.** Let $k \geq 1$, suppose $P(k-1)$ holds, and suppose the body runs at least $k$ times. Then it runs at least $k-1$ times, so by $P(k-1)$, $I$ is true at the test that follows the first $k-1$ passes. The $k$th pass comes right after that test, and the body runs only when the guard is true, so $G$ is also true at that test. By hypothesis 2, $I$ is true at the next test, which is the test that follows the first $k$ passes. So $P(k)$ holds.
By induction, $P(k)$ holds for every $k \geq 0$. Every test of the guard follows the first $k$ passes for some $k \geq 0$, so $I$ is true at every test, and $I$ is a loop invariant. The loop stops only at a test where $G$ is false, and $I$ is true at that test too. □

*Remark* 2 (The three parts of a loop invariant proof)*.* A proof that an algorithm with one loop is correct (Definition 4) usually names a statement $I$ and shows three things.
**Initialization.** Assuming the precondition, $I$ is true at the first test of the guard.
**Maintenance.** If $I$ and the guard are both true at a test, then $I$ is true at the next test.
**Termination.** The loop stops, and when it stops, $I$ together with the guard being false implies the postcondition.
Initialization and maintenance are the two hypotheses of Theorem 1, and they play the roles of the base case and the inductive step of a proof by induction (Definition: Proof by induction). The theorem then says that $I$ is a loop invariant, so the proof does not have to repeat the induction each time. Termination is the part an induction proof does not have: the theorem says what is true *if* the loop stops, not that it does.

**Example 6** (`root` is correct)**.** We show that `root` (Exercise 1) is correct (Definition 4) for the precondition and postcondition of Example 2. Let $n$ be a non-negative integer, and let $I$ be the statement

> `r` is a non-negative integer and $r^2 \le n$.

**Initialization.** At the first test, $r = 0$, which is a non-negative integer, and $r^2 = 0 \le n$, since $n \ge 0$.
**Maintenance.** Suppose $I$ and the guard $(r+1)^2 \le n$ are true at a test. The body replaces `r` with $r + 1$, which is again a non-negative integer, and Solution 2 shows that $(r+1)^2 \le n$. So $I$ is true at the next test.
**Termination.** By Theorem 1, $I$ is true at every test. A non-negative integer $r$ satisfies $r \le r^2$ (if $r = 0$, both sides are 0, and if $r \ge 1$, then $r^2 = r \cdot r \ge r \cdot 1$), so $r \le r^2 \le n$ at every test. Each pass through the body increases `r` by 1, starting from 0, so if the body runs at least $k$ times, then $r = k$ at the test after the first $k$ passes, and so $k \le n$. So the body runs at most $n$ times, and the loop stops. When it stops, $I$ is true and the guard is false, so

$$r^2 \le n \quad \text{and} \quad n < (r+1)^2,$$

and `root` returns this $r$, a non-negative integer. That is the postcondition.

## Termination

**Exercise 3** (A loop that halves)**.** The algorithm `halvings` takes a natural number (Definition: Natural numbers (our convention)) $n$ and counts how many times $n$ can be halved, rounding down, before it reaches 1:

```
procedure halvings(n)    // n is a natural number
    k := n
    c := 0
    while k >= 2
        k := floor(k / 2)
        c := c + 1
    return c
```

Here `floor(x)` is the floor $\lfloor x \rfloor$ (Definition 1) of $x$. Explain why the loop stops for every natural number $n$.

Solution

*Solution* 3. First, `k` is a natural number at every test of the guard. At the first test $k = n$, a natural number. If $k$ is a natural number and the guard $k \geq 2$ is true, then $k/2 \geq 1$, so the new value $\lfloor k/2 \rfloor$ is an integer that is at least 1. So, by Theorem 1, "`k` is a natural number" is a loop invariant.
Second, each pass through the body makes `k` smaller: the new value is $\lfloor k/2 \rfloor \leq k/2 < k$, since $k > 0$. Both values are integers, so `k` decreases by at least 1.
So, if the body runs at least $j$ times, then at the test after the first $j$ passes, $1 \leq k \leq n - j$, and so $j \leq n - 1$. The body runs at most $n - 1$ times, and the loop stops.

**Definition 7** (Loop variant)**.** A **loop variant** of a loop is an integer-valued expression in the loop's variables that is at least 0 at every test of the loop's guard (Definition 5), and that is smaller after each pass through the body than it was before that pass, for every input that satisfies the algorithm's precondition (Definition 2).

**Example 7** (Loop variants of `halvings` and `root`)**.** Solution 3 shows that `k` is a loop variant (Definition 7) of `halvings`: it is an integer that is at least 1, and so at least 0, at every test, and each pass through the body makes it smaller. A loop can have more than one loop variant: `k - 1` is another one, since it is at least 0 at every test and decreases by the same amount as `k`.
The termination part of Example 6 is an argument about the loop variant $n - r$ of `root`. It is an integer, it is at least 0 at every test because $r \leq n$ there, and each pass through the body increases `r` by 1, so it decreases $n - r$ by 1.

**Theorem 2** (A loop with a variant stops)**.** *If a loop has a loop variant (Definition 7) whose value at the first test of the guard is $v_0$, then the body runs at most $v_0$ times, and so the loop stops.*

Proof

*Proof.* Write $v$ for the loop variant. For each non-negative integer $k$, let $P(k)$ be the claim

> if the body runs at least $k$ times, then $v \leq v_0 - k$ at the test that follows the first $k$ passes.

We prove $P(k)$ for every $k \geq 0$ by induction (Definition: Proof by induction).
**Base case.** The test that follows the first 0 passes is the first test, where $v = v_0$.
**Inductive step.** Let $k \geq 1$, suppose $P(k-1)$ holds, and suppose the body runs at least $k$ times. By $P(k-1)$, $v \leq v_0 - (k-1)$ at the test that follows the first $k-1$ passes. The

$k$th pass makes the integer $v$ smaller, so it lowers $v$ by at least 1, and $v \le v_0 - k$ at the next test. So $P(k)$ holds.
Now suppose the body runs at least $k$ times. By $P(k)$, $v \le v_0 - k$ at the test that follows the first $k$ passes, and $v \ge 0$ at every test, so $0 \le v_0 - k$, that is, $k \le v_0$. So the body cannot run $v_0 + 1$ times. □

**Example 8** (A countdown timer)**.** The algorithm `countdown` counts down a timer that starts at $x$ minutes and 0 seconds, one second per pass:

```
procedure countdown(x)   // x is a non-negative integer
    y := 0               // x is the minutes, y the seconds
    while x > 0 or y > 0
        if y > 0
            y := y - 1
        else
            x := x - 1
            y := 59
```

Neither `x` nor `y` is a loop variant (Definition 7): `x` stays the same while `y` counts down, and `y` jumps from 0 to 59 when `x` goes down. The total number of seconds left, $60x + y$, is one.
To see this, first let $I$ be the statement "`x` and `y` are integers with $x \ge 0$ and $0 \le y \le 59$". At the first test, $x \ge 0$ and $y = 0$, so $I$ is true. Suppose $I$ and the guard are true at a test. If $y > 0$, the body replaces `y` with $y - 1$, which is between 0 and 58, and leaves `x` alone. Otherwise $y = 0$, so the guard says $x > 0$, and the body replaces `x` with $x - 1 \ge 0$ and `y` with 59. Either way $I$ is true at the next test, so by Theorem 1, $I$ is a loop invariant.
So $60x + y \ge 0$ at every test. Each pass through the body lowers $60x + y$ by exactly 1: if $y > 0$, the new value is $60x + (y - 1)$, and if $y = 0$, the new value is

$$\begin{aligned} 60(x-1) + 59 &= 60x - 1 \\ &= (60x + y) - 1. \quad (\text{since } y = 0) \end{aligned}$$

So $60x + y$ is a loop variant. Write $x_0$ for the input value of `x`. The variant starts at $60x_0$, so by Theorem 2 the body runs at most $60x_0$ times, and the loop stops. When it stops, the guard is false, so $x \le 0$ and $y \le 0$, and $I$ gives $x \ge 0$ and $y \ge 0$, so $x = 0$ and $y = 0$, where $60x + y = 0$. The variant went down by exactly 1 each time, so the body ran exactly $60x_0$ times, once for each second.

*Remark* 3 (A loop nobody knows how to stop). The process of Example: A proposition whose truth value nobody knows is the loop

```
while n != 1
    if n is even
        n := n / 2
    else
        n := 3 * n + 1
```

on a positive integer $n$. Nobody knows whether it stops for every positive integer $n$: that is the Collatz conjecture (Wikipedia contributors 2026a). So nobody has found a loop variant (Definition 7) for it, since Theorem 2 would then prove that it always stops. Termination is a separate claim from what an invariant gives, and it can be the hard part.

## Choosing a strong enough invariant

**Exercise 4** (The two ends of a reversal)**.** The algorithm `reverse` reverses an array `A[1..n]`, where $n \geq 0$, by swapping its outermost entries and moving inward:

```
procedure reverse(A, n)  // A[1..n] is an array, n >= 0
    i := 1
    j := n
    while i < j
        swap A[i] and A[j]
        i := i + 1
        j := j - 1
```

Show that $i + j = n + 1$ is a loop invariant (Definition 6) of its loop.

Solution

*Solution* 4. We check the two hypotheses of Theorem 1.
**Initialization.** At the first test, $i = 1$ and $j = n$, so $i + j = n + 1$.
**Maintenance.** Suppose $i + j = n + 1$ and the guard $i < j$ are true at a test. The swap does not change `i` or `j`, and the body then replaces `i` with $i + 1$ and `j` with $j - 1$. At the next test,

$$\begin{aligned}(i + 1) + (j - 1) &= i + j \\ &= n + 1.\end{aligned}$$

So, by Theorem 1, $i + j = n + 1$ is a loop invariant.

*Remark* 4 (A true invariant can be too weak). The postcondition (Definition 3) of `reverse` is about the entries of `A`: writing `B[1..n]` for the array as it was before the loop, it says that $A[k] = B[n + 1 - k]$ for every $k$ with $1 \le k \le n$. The loop invariant $i + j = n + 1$ of Solution 4 is true, but it says nothing about `A`, so no amount of reasoning from it, and from the guard being false, can reach this postcondition.
The invariant, together with the guard being false, has to imply the postcondition. Here the guard mentions only `i` and `j`, so the invariant itself has to say something about the entries of `A`. A good way to find one is to ask what the loop has finished so far, and what it has not touched yet, at a test partway through the loop.

**Example 9** (`reverse` is correct)**.** Let `B[1..n]` be the array `A` as it was before the loop of Exercise 4. At a test partway through, the loop has swapped the entries outside positions $i$ to $j$, and has not touched the entries from $i$ to $j$. So let $I$ be the statement made of these three parts:

- **(a)** $i + j = n + 1$ and $i \le j + 1$;
- **(b)** $A[k] = B[n + 1 - k]$ for every $k$ with $1 \le k < i$ or $j < k \le n$;
- **(c)** $A[k] = B[k]$ for every $k$ with $i \le k \le j$.

**Initialization.** At the first test, $i = 1$ and $j = n$. Part (a) holds because $1 + n = n + 1$ and $1 \le n + 1$, since $n \ge 0$. No $k$ between 1 and $n$ has $k < 1$ or $k > n$, so (b) says nothing. Nothing has changed yet, so $A[k] = B[k]$ for every $k$, and (c) holds.
**Maintenance.** Suppose $I$ and the guard $i < j$ are true at a test. By (c), $A[i] = B[i]$ and $A[j] = B[j]$ before the swap. By (a), $j = n + 1 - i$ and $i = n + 1 - j$. So after the swap,

$$A[i] = B[j] = B[n + 1 - i] \quad \text{and} \quad A[j] = B[i] = B[n + 1 - j],$$

and no other entry changes. The body then replaces `i` with $i' = i+1$ and `j` with $j' = j-1$.

- Part (a): $i' + j' = i + j = n + 1$, as in Solution 4. Since $i$ and $j$ are integers and $i < j$, we have $i + 1 \leq j$, so $i' \leq j = j' + 1$.
- Part (b): a $k$ with $k < i'$ has $k < i$, where the swap did not change $A[k]$ and (b) already held, or $k = i$, just shown. A $k$ with $k > j'$ has $k > j$ or $k = j$, in the same way.
- Part (c): a $k$ with $i' \leq k \leq j'$ has $i < k < j$, so the swap did not change $A[k]$, and $A[k] = B[k]$ by (c).

So $I$ is true at the next test.
**Termination.** By Theorem 1, $I$ is true at every test. By (a), $i \leq j+1$, so $j-i+1 \geq 0$ at every test, and each pass lowers $j-i+1$ by 2. So $j-i+1$ is a loop variant (Definition 7), and by Theorem 2 the loop stops. When it stops, the guard is false, so $i \geq j$, and with (a), $i = j$ or $i = j + 1$. Now let $1 \leq k \leq n$.

- If $k < i$ or $k > j$, then $A[k] = B[n + 1 - k]$ by (b).
- Otherwise $i \leq k \leq j$, which can only happen when $i = j = k$. Then (a) gives $2k = n + 1$, so $k = n + 1 - k$, and (c) gives $A[k] = B[k] = B[n + 1 - k]$.

So $A[k] = B[n + 1 - k]$ for every $k$, which is the postcondition of Remark 4. The middle entry of an array of odd length is the one that never moves.

## Nested loops

**Exercise 5** (Looking for a repeated entry)**.** The algorithm `has_repeat` decides whether some value appears twice in an array `A[1..n]`, where $n \geq 1$, by comparing every pair of positions:

```
procedure has_repeat(A, n)   // A[1..n] is an array, n >= 1
    found := false
    for i := 1 to n - 1
        for j := i + 1 to n
            if A[i] = A[j]
                found := true
    return found
```

Run `has_repeat` on the array $A = (3, 1, 4, 1)$, with $n = 4$. List the pairs $(i, j)$ that the inner loop compares, in order, and give the first pair at which `found` becomes true.

Solution

*Solution* 5.

| $(i, j)$ | $A[i]$ | $A[j]$ | `found` after the comparison |
|---|---|---|---|
| $(1, 2)$ | 3 | 1 | false |
| $(1, 3)$ | 3 | 4 | false |
| $(1, 4)$ | 3 | 1 | false |
| $(2, 3)$ | 1 | 4 | false |
| $(2, 4)$ | 1 | 1 | true |
| $(3, 4)$ | 4 | 1 | true |

The inner loop compares the six pairs in the table, in that order. `found` first becomes true at $(2, 4)$, since $A[2] = A[4] = 1$, and it stays true, so `has_repeat` returns true.

*Remark* 5 (Proving a nested loop correct). A loop inside another loop gets its own loop invariant (Definition 6). The proof usually goes from the inside out:

1. prove the inner loop's invariant, using the outer loop's invariant at the test where the inner loop starts;
2. read off what the inner loop has done when it stops;
3. use that in the maintenance step of the outer loop's invariant.

Each `for` loop is a `while` loop (Example 4), so each invariant is a statement about the tests of that loop's guard.

**Example 10** (`has_repeat` is correct)**.** In `has_repeat` (Exercise 5), call a pair of positions $(p, q)$ a *match* if $1 \le p < q \le n$ and $A[p] = A[q]$. The postcondition is that `has_repeat` returns true if `A` has a match, and false if it has none. The algorithm never changes `A`. We follow Remark 5.

**The inner loop.** Fix a pass of the outer loop with index $i$, where $1 \le i \le n - 1$, and suppose that, when the inner loop starts,

> `found` is true exactly when there is a match $(p, q)$ with $p < i$,

which is the outer loop's invariant $K$, proved in the next part. Let $J$ be the statement

> $i + 1 \le j \le n + 1$, and `found` is true exactly when there is a match $(p, q)$ with $p < i$, or with $p = i$ and $q < j$.

- Initialization: at the first test, $j = i + 1 \le n$. No $q$ has $i < q < i + 1$, so $J$ says the same as the supposition above.

- Maintenance: suppose $J$ and the guard $j \leq n$ are true at a test. A match with $p < i$, or with $p = i$ and $q < j + 1$, is either one of the matches that $J$ describes or the pair $(i, j)$. If $A[i] = A[j]$, the body sets `found` to true, and $(i, j)$ is such a match. If not, `found` does not change, and $(i, j)$ is not a match. Either way, after `j` becomes $j + 1 \leq n + 1$, $J$ is true at the next test.
- Termination: by Theorem 1, $J$ is a loop invariant, so $n + 1 - j \geq 0$ at every test, and each pass lowers $n+1-j$ by 1. So $n+1-j$ is a loop variant (Definition 7), and the inner loop stops (Theorem 2). It stops when $j \leq n$ is false, so, with $j \leq n + 1$, $j = n + 1$. Every match has $q \leq n < n + 1$, so, when the inner loop stops,

  > `found` is true exactly when there is a match $(p, q)$ with $p < i$ or $p = i$, that is, with $p < i + 1$.

**The outer loop.** Let $K$ be the statement

> $1 \leq i \leq n$, and `found` is true exactly when there is a match $(p, q)$ with $p < i$.

- Initialization: at the first test, $i = 1 \leq n$, since $n \geq 1$, `found` is false, and no match has $p < 1$.
- Maintenance: suppose $K$ and the guard $i \leq n - 1$ are true at a test. Then the inner loop runs, and when it stops, `found` is true exactly when there is a match with $p < i + 1$. After `i` becomes $i + 1 \leq n$, $K$ is true at the next test.
- Termination: by Theorem 1, $K$ is a loop invariant, so $n - i \geq 0$ at every test, and each pass lowers $n - i$ by 1. So $n - i$ is a loop variant, and the outer loop stops. It stops when $i \leq n - 1$ is false, so, with $i \leq n$, $i = n$. Every match has $p < q \leq n$, so $p < n$. So, when the outer loop stops, `found` is true exactly when `A` has a match, and the algorithm returns `found`.

## References


Wikipedia contributors. 2026a. *Collatz Conjecture — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Collatz_conjecture.

Wikipedia contributors. 2026b. *Floor and Ceiling Functions — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Floor_and_ceiling_functions.