Recursion

Recursive definitions, recursive programs and proofs about them

Last modified: 2026-11-12 01:23:00 (PST)

Source

The topics of this chapter follow Kameron Decker Harris's Fall 2026 lecture notes on recursion and recursive functions. These come from those notes:

- the two parts of a recursive definition, a base case and a recursive rule;
- the recursive definitions of the natural numbers and of the strings over an alphabet;
- the three questions to ask when writing a recursive function: the base case, the recursive call and the combination;
- the definition of a recursive algorithm.

Everything else is new: the exercises, the other examples, the Racket functions, the algorithm, the theorems and their proofs.

Recursive definitions

Exercise 1 (Double and subtract one)**.** A sequence (Definition: Sequence) starts with $a_0 = 2$, and each later term is twice the term before it, minus 1. Find a_1, a_2, a_3, a_4 and a_5.

Solution

Solution 1. Each term comes from the one before it:

$$\begin{aligned} a_1 &= 2 \cdot 2 - 1 = 3, \\ a_2 &= 2 \cdot 3 - 1 = 5, \\ a_3 &= 2 \cdot 5 - 1 = 9, \\ a_4 &= 2 \cdot 9 - 1 = 17, \\ a_5 &= 2 \cdot 17 - 1 = 33. \end{aligned}$$

To find a_5 this way, we had to find a_1 through a_4 first.

Definition 1 (Recursive definition)**.** Let n_0 be an integer (Definition: Integers). A **recursive definition** (Wikipedia contributors 2026e) of a function (Definition: Function, map, mapping, transformation) f on the integers $n \geq n_0$ has two parts:

1. *Base case:* the value of f is given directly at n_0, and possibly at a few more of the smallest inputs.
2. *Recursive rule:* $f(n)$ at every other input n is given in terms of the values of f at inputs that are smaller than n and at least n_0.

Example 1 (The double-minus-one sequence, defined recursively)**.** The sequence of Exercise 1 has a recursive definition (Definition 1) with $n_0 = 0$. Its base case is

$$a_0 = 2,$$

and its recursive rule is

$$a_n = 2a_{n-1} - 1 \quad \text{for every integer } n \geq 1.$$

The recursive rule gives a_n from the single value a_{n-1} just before it, so this is a recursively defined sequence (Definition: Recursively defined sequence) with $m = 1$.

Example 2 (Removing the last digit)**.** For a natural number (Definition: Natural numbers (our convention)) n, the floor (Definition: Floor) $\lfloor n/10 \rfloor$ is n with its last decimal digit removed, such as $\lfloor 2026/10 \rfloor = \lfloor 202.6 \rfloor = 202$.

Example 3 (Counting digits)**.** Define a function d on the natural numbers (Definition: Natural numbers (our convention)) by

$$d(n) = \begin{cases} 1 & \text{if } n \leq 9, \\ 1 + d(\lfloor n/10 \rfloor) & \text{if } n \geq 10, \end{cases}$$

where $\lfloor n/10 \rfloor$ is the floor (Definition: Floor) of $n/10$, which is n with its last decimal digit removed. The first line is the base case and the second is the recursive rule (Definition 1). For $n \geq 10$, the input $\lfloor n/10 \rfloor$ is at least 1 and at most $n/10$, which is less than n, so the recursive rule uses a smaller input, but usually not $n - 1$. For example,

$$\begin{aligned} d(2026) &= 1 + d(202) \\ &= 1 + 1 + d(20) \\ &= 1 + 1 + 1 + d(2) \\ &= 1 + 1 + 1 + 1 \\ &= 4, \end{aligned}$$

and 2026 has 4 digits.

Recursively defined sets

Exercise 2 (Building a set of integers)**.** A set (Definition: Set) S of integers is built by two rules:

- $1 \in S$;
- if $x \in S$, then $x + 2 \in S$ and $3x \in S$.

Every element of S comes from these two rules. Find every element of S that is less than 12.

Solution

Solution 2. Starting from 1 and adding 2 again and again gives $1, 3, 5, 7, 9, 11$, so all six are in S. Tripling gives nothing new below 12: $3 \cdot 1 = 3$ and $3 \cdot 3 = 9$ are already in the list, and tripling 5 or more gives at least 15. Both rules turn an odd number into an odd number, so no even number is in S. So the elements of S less than 12 are $1, 3, 5, 7, 9$ and 11.

Definition 2 (Recursively defined set)**.** A **recursively defined set** (Wikipedia contributors 2026e) is a set (Definition: Set) given by two parts, named as in Definition 1:

1. a base case, which names some elements of the set directly;
2. a recursive rule, which says how to build new elements of the set from elements already in it.

The set has no other elements: each of its elements comes from the base case by applying the recursive rule a finite number of times.

Example 4 (The natural numbers, defined recursively)**.** The natural numbers (Definition: Natural numbers (our convention)) are a recursively defined set (Definition 2). The base case is $1 \in \mathbb{N}$, and the recursive rule is that if $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$. The base case gives 1, one application of the rule gives $1 + 1 = 2$, a second gives $2 + 1 = 3$, and so on. The non-negative integers (Definition: Non-negative integers) have the same recursive rule, with the base case $0 \in \mathbb{N}_0$.

Definition 3 (Alphabet)**.** An **alphabet** (Wikipedia contributors 2026a) is a finite, non-empty set (Definition: Set), whose elements are called its symbols. An alphabet is often written Σ, the capital Greek letter sigma.

Example 5 (Three alphabets)**.** Each of these sets is an alphabet (Definition 3):

- $\{0, 1\}$, the two binary digits;
- $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, the four bases of DNA;
- $\{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$, the 26 lowercase letters of English.

The set of all integers is not an alphabet, because it is not finite.

Definition 4 (String)**.** Let Σ be an alphabet (Definition 3). The set Σ^* of **strings** over Σ (Wikipedia contributors 2026f) is the recursively defined set (Definition 2) with

- base case: the *empty string* λ, which has no symbols, is in Σ^*;
- recursive rule: if $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$, where wx is the string w followed by the symbol x.

Racket and Python write the empty string `""`.

Example 6 (Building a string of DNA bases)**.** Let $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. The string `GAT` is in Σ^* (Definition 4), by three applications of the recursive rule:

- $\lambda \in \Sigma^*$ and $\texttt{G} \in \Sigma$, so $\lambda\texttt{G} = \texttt{G}$ is in Σ^*;
- $\texttt{G} \in \Sigma^*$ and $\texttt{A} \in \Sigma$, so $\texttt{GA} \in \Sigma^*$;
- $\texttt{GA} \in \Sigma^*$ and $\texttt{T} \in \Sigma$, so $\texttt{GAT} \in \Sigma^*$.

The string `GAU` is not in Σ^*, since the rule adds only symbols of Σ, and $\texttt{U} \notin \Sigma$.

Definition 5 (Palindrome)**.** A **palindrome** (Wikipedia contributors 2026c) is a string (Definition 4) that reads the same backward as forward: its ith symbol equals its ith symbol from the end, for every i.

Example 7 (Some palindromes)**.** Over the alphabet of lowercase letters, `noon` and `racecar` are palindromes (Definition 5), and so is `abba`, from Exercise: Completing a palindrome. The string `recur` is not a palindrome: backward it reads `rucer`. The empty string and every string of one symbol are palindromes.

Recursion in Racket

Exercise 3 (Tracing a recursive Racket function)**.** The Racket function `binary` takes a non-negative integer (Definition: Non-negative integers) `n` and returns a string:

```
(define (binary n)
  (if (< n 2)
      (number->string n)
      (string-append (binary (quotient n 2))
                     (number->string (remainder n 2)))))
```

Here

- `(if test a b)` returns `a` when `test` is true and `b` when it is false;
- `(quotient n 2)` is $\lfloor n/2 \rfloor$, the floor (Definition: Floor) of $n/2$;
- `(remainder n 2)` is the remainder, 0 or 1, left after dividing `n` by 2;
- `number->string` turns a number into the string of its digits;
- `string-append` joins two strings into one.

Run `(binary 6)` by hand: write down each call of `binary` that it makes, and the string that each call returns.

Solution

Solution 3. Since $6 \geq 2$, `(binary 6)` calls `(binary 3)`, because $\lfloor 6/2 \rfloor = 3$, and puts `"0"`, the remainder of 6 divided by 2, after its result. Since $3 \geq 2$, `(binary 3)` calls `(binary 1)` and puts `"1"` after its result. Since $1 < 2$, `(binary 1)` returns `"1"` without calling `binary` again. Replacing each call with what it returns:

```
(binary 6)
= (string-append (binary 3) "0")
= (string-append (string-append (binary 1) "1") "0")
= (string-append (string-append "1" "1") "0")
= (string-append "11" "0")
= "110"
```

So `(binary 1)` returns `"1"`, `(binary 3)` returns `"11"`, and `(binary 6)` returns `"110"`. That is 6 written in base 2, since $1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 6$.
Racket returns the same values:

```
(binary 1)
; "1"
(binary 3)
; "11"
(binary 6)
; "110"
```

Remark 1 (Three questions for writing a recursive function). A Racket function that calls itself, such as `binary` in Exercise 3, answers three questions:

- **Base case.** Which inputs does it answer directly? For `binary`, the inputs 0 and 1, whose base-2 form is the single digit `"0"` or `"1"`.
- **Recursive call.** Which smaller input does it call itself on? For `binary`, the input $\lfloor n/2 \rfloor$, which is less than n when $n \geq 2$.
- **Combination.** How does it turn the result of the recursive call into its own answer? For `binary`, it puts the last base-2 digit of n, the remainder of n divided by 2, after the result.

The base case is the branch of the `if` that does not call the function, and the recursive call and the combination make up the other branch.

Example 8 (Counting digits in Racket)**.** The recursive definition of d in Example 3 becomes a Racket function line by line:

```
(define (digits n)
  (if (< n 10)
      1
      (+ 1 (digits (quotient n 10)))))
```

Following Remark 1:

- the base case answers 1 for every input less than 10;
- the recursive call is `(digits (quotient n 10))`, on $\lfloor n/10 \rfloor$, which is less than n;
- the combination adds 1 to the result of the recursive call.

`(digits 2026)` calls `(digits 202)`, `(digits 20)` and `(digits 2)` in turn, as in Example 3:

```
(digits 2026)
; 4
```

Lists

Exercise 4 (Building a list with `cons`)**.** In Racket, `empty` is the list with no elements, and `(cons x xs)` is the list whose first element is `x`, followed by the elements of the list `xs`. For example, `(cons 5 empty)` is the one-element list `'(5)`. Write the list `'(7 2 9)` using only `cons`, `empty` and numbers.

Solution

Solution 4. Build the list from its last element forward:

- `(cons 9 empty)` is `'(9)`;
- `(cons 2 (cons 9 empty))` is `'(2 9)`;
- `(cons 7 (cons 2 (cons 9 empty)))` is `'(7 2 9)`.

So `'(7 2 9)` is `(cons 7 (cons 2 (cons 9 empty)))`.
Racket prints it the same way:

```
(cons 7 (cons 2 (cons 9 empty)))
; '(7 2 9)
```

Definition 6 (List)**.** Let A be a set (Definition: Set). The set of **lists** of elements of A (Wikipedia contributors 2026b) is the recursively defined set (Definition 2) with

- base case: the empty list, written `empty`, is a list;
- recursive rule: if $x \in A$ and L is a list, then $\text{cons}(x, L)$ is a list, whose *first* element is x and whose *rest* is L.

Racket writes the empty list as `empty` or `'()`, and $\text{cons}(x, L)$ as `(cons x L)`, and `(first L)` and `(rest L)` return the two parts of a list that is not empty. Racket also has the older names `car` for `first` and `cdr` for `rest`. Python writes the empty list `[]`.

Example 9 (Taking a list apart)**.** Let `L` be the list `'(7 2 9)` (Definition 6), which Solution 4 builds as `(cons 7 (cons 2 (cons 9 empty)))`. Then `first` and `rest` take it apart:

```
(define L (cons 7 (cons 2 (cons 9 empty))))
(first L)
; 7
(rest L)
; '(2 9)
(first (rest L))
; 2
(rest (rest (rest L)))
; '()
```

The last value, `'()`, is how Racket prints `empty`. `(first empty)` raises an error, since the empty list is not built by `cons` and has no first element:

```
(first empty)
; first: contract violation
;   expected: (and/c list? (not/c empty?))
;   given: '()
```

Definition 7 (Length of a list)**.** The **length** $\text{len}(L)$ of a list L (Definition 6) is defined recursively, following the two parts of the definition of a list:

$$\text{len}(\texttt{empty}) = 0,$$
$$\text{len}(\text{cons}(x, L)) = 1 + \text{len}(L).$$

Example 10 (The length of `'(7 2 9)`)**.** In Racket, the definition of length (Definition 7) is

```
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L)))))
```

where `(empty? L)` tests whether `L` is the empty list. Replacing each call with the recursive rule:

```
(len '(7 2 9))
= (+ 1 (len '(2 9)))
= (+ 1 (+ 1 (len '(9))))
= (+ 1 (+ 1 (+ 1 (len '()))))
= (+ 1 (+ 1 (+ 1 0)))
= 3
```

Racket's built-in `length` gives the same values:

```
(len '(7 2 9))
; 3
(length '(7 2 9))
; 3
```

Definition 8 (Appending two lists)**.** The **append** $\mathrm{app}(L, M)$ of two lists L and M (Definition 6), the list of the elements of L followed by the elements of M, is defined recursively on L:

$$\begin{aligned} \mathrm{app}(\texttt{empty}, M) &= M, \\ \mathrm{app}(\mathrm{cons}(x, L), M) &= \mathrm{cons}(x, \mathrm{app}(L, M)). \end{aligned}$$

Example 11 (Appending `'(1 2)` and `'(3)`)**.** In Racket, the definition of append (Definition 8) is

```
(define (app L M)
  (if (empty? L)
      M
      (cons (first L) (app (rest L) M))))
```

The recursion takes `L` apart and leaves `M` alone:

```
(app '(1 2) '(3))
= (cons 1 (app '(2) '(3)))
= (cons 1 (cons 2 (app '() '(3))))
= (cons 1 (cons 2 '(3)))
= '(1 2 3)
```

Racket's built-in `append` gives the same values:

```
(app '(1 2) '(3))
; '(1 2 3)
(append '(1 2) '(3))
; '(1 2 3)
```

Recursive algorithms

Exercise 5 (Is it a palindrome?)**.** The algorithm `pal`, in the pseudocode of Remark: How to read the pseudocode in this chapter, tests whether a string is a palindrome (Definition 5):

```
procedure pal(s)        // s is a string s[1] s[2] ... s[n] of n symbols
    if n <= 1
        return true
    if s[1] != s[n]
        return false
    return pal(s[2..n-1])   // s without its first and last symbols
```

Here `!=` means $\neq$, and `s[2..n-1]` is the string of the symbols `s[2]` through `s[n-1]`, which is the empty string when $n = 2$, since the range `2..1` is empty. Run `pal(level)` by hand: write down each call of `pal` that it makes, and what each call returns.

Solution

Solution 5.

Call	n	First and last symbols	What the call does
`pal(level)`	5	`l`, `l`	calls `pal(eve)`
`pal(eve)`	3	`e`, `e`	calls `pal(v)`
`pal(v)`	1		returns true

`pal(v)` returns true, so `pal(eve)` returns true, and so `pal(level)` returns true: `level` is a palindrome.

Definition 9 (Recursive algorithm)**.** An algorithm (Definition: Algorithm) is **recursive** (Wikipedia contributors 2026d) if it solves a problem by calling itself on smaller inputs of the same problem and using what those calls return. The inputs that it answers without calling itself are its base cases, as in Definition 1.

Example 12 (`pal` is recursive)**.** The algorithm `pal` of Exercise 5 is recursive (Definition 9). On a string of $n \geq 2$ symbols whose first and last symbols match, it calls itself on the string without those two symbols, which has $n-2$ symbols, and returns what that call returns.
It answers directly in two cases: when $n \leq 1$, since the empty string and a single symbol read the same backward, and when the first and last symbols differ. For example, `pal(recur)` finds that the first and last symbols are both `r`, and calls `pal(ecu)`. That call finds `e` and `u`, which differ, and returns false, so `pal(recur)` returns false.

Proofs about recursive definitions

Exercise 6 (Guessing a formula)**.** Compare each term $a_0, a_1, \ldots, a_5$ of the sequence in Solution 1 with the power 2^n, and guess a formula for a_n that does not use the earlier terms.

Solution

Solution 6.

n	0	1	2	3	4	5
a_n	2	3	5	9	17	33
2^n	1	2	4	8	16	32

Each term is one more than the power below it, so a good guess is $a_n = 2^n + 1$. Six terms do not prove the guess for every n.

Theorem 1 (A formula for the double-minus-one sequence)**.** *Let $a_0, a_1, \ldots$ be the sequence with $a_0 = 2$ and $a_n = 2a_{n-1} - 1$ for every integer $n \geq 1$ (Example 1). Then $a_n = 2^n + 1$ for every non-negative integer n.*

Proof

Proof. We use induction (Definition: Proof by induction), following Remark: The shape of a proof by induction with $n_0 = 0$. Let $P(n)$ be the equation $a_n = 2^n + 1$.
Base case. $a_0 = 2 = 2^0 + 1$.
Inductive step. Let $k \geq 0$ be an integer, and suppose $P(k)$: $a_k = 2^k + 1$. The goal $P(k+1)$ is $a_{k+1} = 2^{k+1} + 1$. Since $k + 1 \geq 1$, the recursive rule gives a_{k+1}, and

$$\begin{aligned} a_{k+1} &= 2a_k - 1 && \text{(recursive rule)} \\ &= 2(2^k + 1) - 1 && \text{(induction hypothesis)} \\ &= 2^{k+1} + 2 - 1 \\ &= 2^{k+1} + 1. \end{aligned}$$

So $P(k+1)$ holds, and by induction $a_n = 2^n + 1$ for every non-negative integer n. □

Definition 10 (Structural induction)**. Structural induction** (Wikipedia contributors 2026g) is a way to prove that every element of a recursively defined set (Definition 2) has a property P, in two steps:

1. **Base case.** Show that every element that the set's base case names has P.
2. **Inductive step.** Show that whenever the recursive rule builds a new element from elements that have P, the new element has P too.

Example 13 (Every element of S is odd)**.** Let S be the set of Exercise 2. It is a recursively defined set (Definition 2), with base case $1 \in S$ and recursive rule that if $x \in S$, then $x + 2 \in S$ and $3x \in S$. We use structural induction (Definition 10) to show that every element of S is a positive odd (Definition: Even integer) integer.
Base case. The base case names 1, and $1 = 2 \cdot 0 + 1$ is positive and odd.
Inductive step. Let $x \in S$ be a positive odd integer, so $x = 2j + 1$ for some integer $j \geq 0$. The recursive rule builds $x + 2$ and $3x$ from x, and

$$\begin{aligned} x + 2 &= 2(j + 1) + 1, \\ 3x &= 6j + 3 = 2(3j + 1) + 1, \end{aligned}$$

which are positive and odd, since $j + 1$ and $3j + 1$ are non-negative integers.
So every element of S is a positive odd integer. Every positive odd integer $2j + 1$ is in S too, since it comes from 1 by adding 2 a total of j times. So S is the set of positive odd

integers, as Solution 2 found for the elements less than 12.

Remark 2 (Why structural induction works). Every element of a recursively defined set (Definition 2) comes from the base case by applying the recursive rule some number $k \geq 0$ of times. Let $Q(k)$ be the claim that every element built with k applications of the rule has the property P. Strong induction (Definition: Proof by strong induction) on k proves $Q(k)$ for every $k \geq 0$:

- an element built with 0 applications is named by the base case, so it has P by the base case of Definition 10;
- an element built with $k \geq 1$ applications comes, by its last application, from elements built with fewer applications, which have P by the strong induction hypothesis, so it has P by the inductive step.

So every element of the set has P.

Theorem 2 (Appending adds lengths)**.** *For all lists L and M (Definition 6),*

$$\text{len}(\text{app}(L, M)) = \text{len}(L) + \text{len}(M),$$

where len *is the length (Definition 7) and* app *is the append (Definition 8).*

Proof

Proof. Let M be a list. We use structural induction (Definition 10) on L, with $P(L)$ the equation $\text{len}(\text{app}(L, M)) = \text{len}(L) + \text{len}(M)$.
Base case. The base case of Definition 6 names only the empty list, and

$$\begin{aligned} \text{len}(\text{app}(\texttt{empty}, M)) &= \text{len}(M) && \text{(definition of app)} \\ &= 0 + \text{len}(M) \\ &= \text{len}(\texttt{empty}) + \text{len}(M). && \text{(definition of len)} \end{aligned}$$

Inductive step. The recursive rule builds $\text{cons}(x, L)$ from an element x and a list L. Suppose $P(L)$. Then

$$\begin{aligned} &\text{len}(\text{app}(\text{cons}(x, L), M)) \\ &= \text{len}(\text{cons}(x, \text{app}(L, M))) && \text{(definition of app)} \\ &= 1 + \text{len}(\text{app}(L, M)) && \text{(definition of len)} \\ &= 1 + \text{len}(L) + \text{len}(M) && \text{(induction hypothesis } P(L)) \\ &= \text{len}(\text{cons}(x, L)) + \text{len}(M), && \text{(definition of len)} \end{aligned}$$

which is $P(\text{cons}(x, L))$. By structural induction, $P(L)$ holds for every list L. Since M was any list, the equation holds for all lists L and M. □

Theorem 3 (The function d counts digits)**.** *Let d be the function of Example 3. For every natural number n,*

$$10^{d(n)-1} \leq n < 10^{d(n)},$$

so n has exactly $d(n)$ decimal digits.

Proof

Proof. A natural number n has m digits exactly when $10^{m-1} \leq n < 10^m$, since 10^{m-1}, which is 1 followed by $m-1$ zeros, is the smallest natural number with m digits, and 10^m, which is 1 followed by m zeros, is the smallest with $m+1$. So the inequalities say that n has $d(n)$ digits.

The recursive rule uses $d(\lfloor n/10 \rfloor)$, not $d(n-1)$, so we use strong induction (Definition: Proof by strong induction). Let n be a natural number, and suppose the inequalities hold for every natural number less than n.

If $n \leq 9$, then $d(n) = 1$, and $10^0 = 1 \leq n < 10 = 10^1$.

If $n \geq 10$, let $q = \lfloor n/10 \rfloor$. Then $q \leq n/10 < q+1$, so $10q \leq n < 10q + 10$. Also $q \geq 1$, since $n/10 \geq 1$, and $q < n$, so q is a natural number less than n, and by the strong induction hypothesis

$$10^{d(q)-1} \leq q < 10^{d(q)}.$$

The recursive rule gives $d(n) = 1 + d(q)$. For the lower bound,

$$\begin{aligned} n &\geq 10q \\ &\geq 10 \cdot 10^{d(q)-1} && \text{(strong induction hypothesis)} \\ &= 10^{d(q)} \\ &= 10^{d(n)-1}. && (d(n) = 1 + d(q)) \end{aligned}$$

For the upper bound, q and $10^{d(q)}$ are integers with $q < 10^{d(q)}$, so $q + 1 \leq 10^{d(q)}$, and

$$\begin{aligned} n &< 10(q+1) \\ &\leq 10 \cdot 10^{d(q)} && (q+1 \leq 10^{d(q)}) \\ &= 10^{d(q)+1} \\ &= 10^{d(n)}. && (d(n) = 1 + d(q)) \end{aligned}$$

So the inequalities hold for n, and by strong induction they hold for every natural number n. □

References

Wikipedia contributors. 2026a. *Alphabet (Formal Languages) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Alphabet_(formal_languages).

Wikipedia contributors. 2026b. *List (Abstract Data Type) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/List_(abstract_data_type).

Wikipedia contributors. 2026c. *Palindrome — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Palindrome.

Wikipedia contributors. 2026d. *Recursion (Computer Science) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Recursion_(computer_science).

Wikipedia contributors. 2026e. *Recursive Definition — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Recursive_definition.

Wikipedia contributors. 2026f. *String (Computer Science) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/String_(computer_science).

Wikipedia contributors. 2026g. *Structural Induction — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Structural_induction.