Recursive Definitions

Table of contents

Recursively defined sequences 1
Recursively defined sets 3
See also 5

Defining a sequence, or a set, by saying how to build its members out of members you already have. See sequences-and-summations[1] for the other ways to define a sequence, and strong-induction-and-the-well-ordering-principle[2] for the proof method that recursive definitions almost always call for.

Recursively defined sequences

To define a sequence recursively, two things are needed:

1. **Initial conditions** — the values of some number of initial terms of the sequence.
2. **A recurrence relation** — a formula specifying how to calculate the value of a term of the sequence from the values of one or more previous terms.

Neither half is optional, and **the same recurrence relation with different initial conditions can give a completely different sequence**:

Definition	Sequence
$a_1 = 2$ and $a_k = 3a_{k-1}$ for each integer $k \geq 2$	$2, 6, 18, 54, ...$
$b_1 = 7$ and $b_k = 3b_{k-1}$ for each integer $k \geq 2$	$7, 21, 63, 189, ...$

The recurrence is identical; only the seed differs, and the two sequences share no terms.

[1] courses/math1061/sequences-and-summations.pdf
[2] courses/math1061/strong-induction-and-the-well-ordering-principle.pdf

Checking that a sequence satisfies a recurrence

Given an explicit formula and a candidate recurrence, the check is a one-step calculation: **assume the formula at $k-1$, substitute it into the recurrence, and see whether what comes out is the formula at k.**

For $b_n = 4n + 3$ and the candidate $b_k = 4 + b_{k-1}$:

$$b_k = 4 + b_{k-1} = 4 + \big(4(k-1) + 3\big) = 4k + 3 \quad \checkmark$$

For the candidate $b_k = 4b_{k-1}$, the same substitution gives

$$b_k = 4b_{k-1} = 4\big(4(k-1) + 3\big) = 16k - 16 + 3 = 16k - 13 \quad \times$$

which is not $4k + 3$, so that recurrence is not satisfied.

Reading a recursive definition off an explicit formula

> This section goes explicit → recursive. The reverse trip — recursive → explicit, which is the harder and more useful one — is solving-recurrence-relations[3].

Going the other way — from an explicit formula to a recursive definition — means computing the first few terms, spotting what is done to each term to get the next, then proving the guess.

Factorial. For $n \in \mathbb{Z}^+$, $n! = n(n-1)\cdots(2)(1)$, and $0! = 1$ **by definition**. Worth memorising: the $0! = 1$ convention is what makes formulas involving $n!$ work at $n = 0$ without a special case.

Example: $a_n = \dfrac{(-1)^n}{n!}$ for each integer $n \geq 0$.

The first five terms, and the factor taking each to the next:

$$\underbrace{1}_{a_0} \xrightarrow{\times -1} \underbrace{-1}_{a_1} \xrightarrow{\times -\frac{1}{2}} \underbrace{\tfrac{1}{2}}_{a_2} \xrightarrow{\times -\frac{1}{3}} \underbrace{-\tfrac{1}{6}}_{a_3} \xrightarrow{\times -\frac{1}{4}} \underbrace{\tfrac{1}{24}}_{a_4}$$

The multiplier at step n is $-\frac{1}{n}$, suggesting $a_n = -\frac{1}{n}a_{n-1}$. Verifying it against the explicit formula:

[3] courses/math1061/solving-recurrence-relations.pdf

Suppose $a_{n-1} = \dfrac{(-1)^{n-1}}{(n-1)!}$. Then

$$a_n = -\frac{1}{n} \cdot a_{n-1} = \frac{(-1)(-1)^{n-1}}{n \cdot (n-1)!} = \frac{(-1)^n}{n!} \quad \checkmark$$

So the recursive definition is

- *initial condition*: $a_0 = 1$
- *recurrence relation*: $a_n = -\dfrac{1}{n} a_{n-1}$ for all $n \geq 1$

Note how the recurrence relation's range starts one past the last initial condition — that is the general rule, and it is what makes the definition cover every term exactly once.

Recursively defined sets

To define a set recursively, **three** things are needed:

1. **Base** — a statement that certain (usually small) objects belong to the set.
2. **Recursion** — rules on how to build new set objects from those objects already known to be in the set.
3. **Restriction** — a statement saying that all objects in the set come from the Base and the Recursion.

The Restriction is the part that is easy to forget and easy to underrate. Without it the Base and Recursion only say what *must* be in the set; the Restriction is what says nothing *else* is, and it is what licenses arguing that a given object is **not** a member.

The set of all binary strings

A **binary string** is a sequence of 0's and 1's. The set B of all binary strings is defined recursively:

- **Base**: the empty string ε is in B.
- **Recursion**: if $s \in B$, then $s0 \in B$ and $s1 \in B$.
- **Restriction**: nothing is in B other than objects defined by the Base and the Recursion.

Every string therefore has a build history from ε, one character at a time — the string 011 is built up as $\varepsilon \to 0 \to 01 \to 011$. Applying the two rules repeatedly generates a binary tree of strings, and a build history is just a path down it from the root:

Two things the picture makes plain that the rules alone do not. Every binary string appears **exactly once** in this tree, at depth equal to its length — so for B a build history is unique,

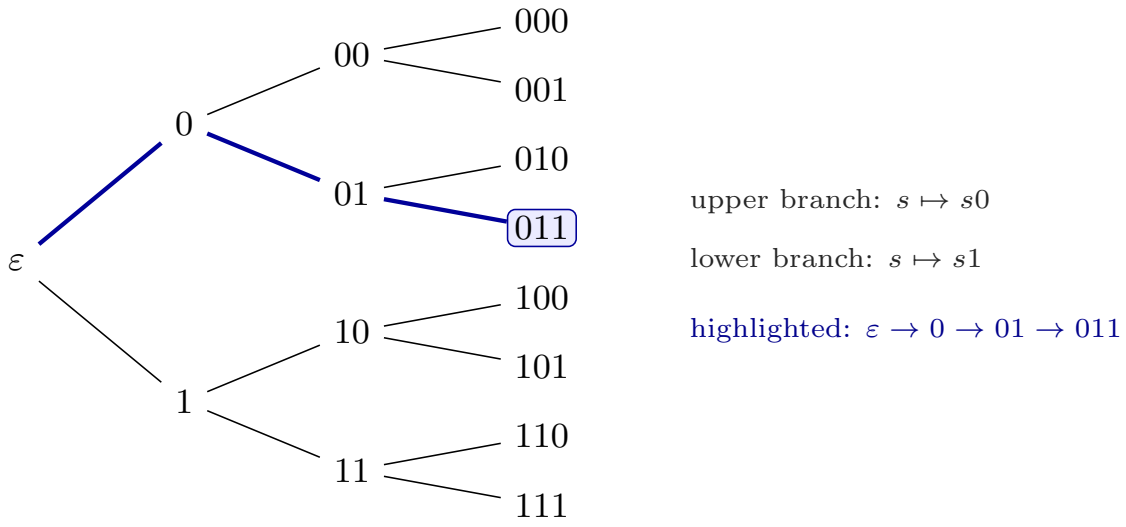


Figure 1: The Recursion applied three times over, drawn the way the lecture drew it: the Base object ε at the root on the left, the two rules fanning rightwards as plain segments with no arrowheads, and each node labelled with the string it is. The upper branch appends a 0 and the lower a 1, so the level a string sits at is its length. **The highlighted path is added here** — the lecture wrote the build history of 011 out as a separate chain rather than marking it on the tree; drawn on the tree it shows what a build history actually is, a route from the Base to the object.

and there is no ambiguity about how a given string was made. And the tree only ever grows: neither rule shortens a string, so a property that holds at the root and survives both rules holds everywhere below. That is the shape of argument the Restriction licenses, and it is what the next section turns into a method.

Arguing about membership

For a recursively defined set, the two directions of argument look quite different:

- **To show an object is in the set**, exhibit a build history — a chain from the Base to the object, with each step naming the rule used.
- **To show an object is not in the set**, find a property that the Base has and that every Recursion rule preserves, and observe the object lacks it. This is an induction over the structure of the set, and it works only because of the Restriction.

See also

- solving-recurrence-relations[4] — the other direction: recursive definition to explicit formula, and proving it
- sequences-and-summations[5] · strong-induction-and-the-well-ordering-principle[6] · mathematical-induction[7]
- counting-principles[8] — $n!$ as the number of arrangements of n objects
- 2026-08-31-recursive-definitions[9] — Lecture 15
- practice-problems[10] — §19 Recursive Definitions, with full worked solutions

[4] courses/math1061/solving-recurrence-relations.pdf
[5] courses/math1061/sequences-and-summations.pdf
[6] courses/math1061/strong-induction-and-the-well-ordering-principle.pdf
[7] courses/math1061/mathematical-induction.pdf
[8] courses/math1061/counting-principles.pdf
[9] courses/math1061/2026-08-31-recursive-definitions.pdf
[10] courses/math1061/practice-problems.pdf