Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

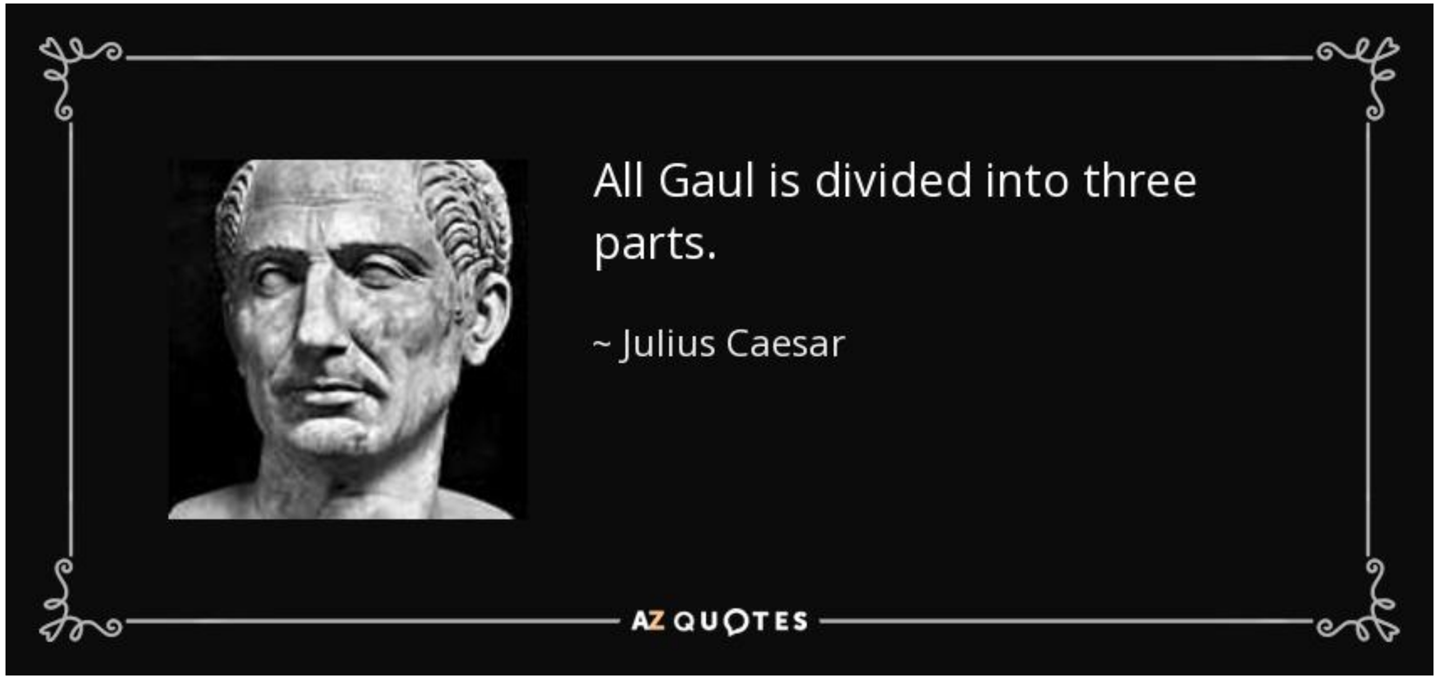
Stepwise Refinement

Introduction

We introduce Stepwise Refinement, a key approach to programming, and illustrate its use in many examples.

- Divide and Conquer
- Sequential Refinement
- Case Analysis
- Iterative Refinement
- Recursive Refinement

Divide and Conquer



All Gaul is divided into three parts.

~ Julius Caesar

AZ QUOTES

Divide and Conquer:

All Gaul is divided into three parts. To conquer Gaul:

First, conquer the first part.

Then, conquer the second part.

Finally, conquer the third part.

A methodology.

Divide and Conquer: Applied to programming

To write a program:
    First, break it into subprograms.
    Then, write each subprogram separately.

the methodology is called programming by *Stepwise Refinement*.

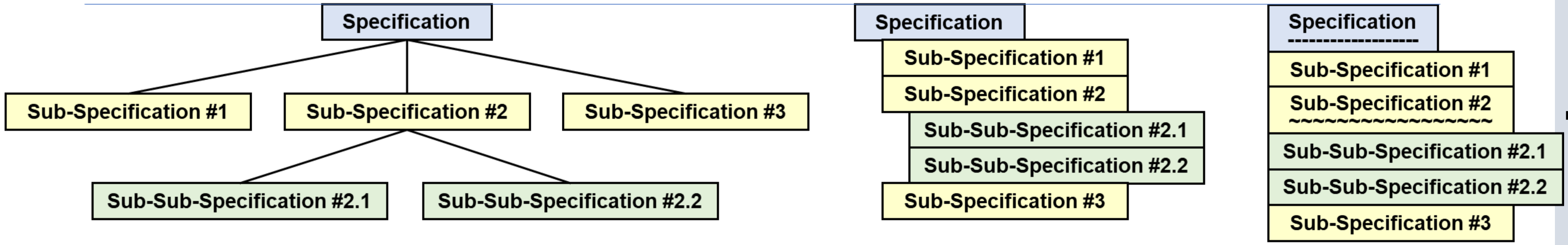
Divide and Conquer: Applied to programming

To write a program:
First, break it into subprograms.
Then, write each subprogram separately.

the methodology is called programming by *Stepwise Refinement*.

☞ **Program top-down, outside-in.**

Stepwise Refinement: Creates a hierarchy of subprograms, each with its own specification.



☞ **Program top-down, outside-in.**

Stepwise Refinement: A “program” to follow as you code.

```
if P is simple to write :
    Write it
else:
    Refine P into simpler subprograms
    Write each subprogram
```

☞ **Program top-down, outside-in.**

where **Refine** is:

Sequential steps
Do one thing after another.
Case analysis
Do one thing or another.
Iteration
Do one thing repeatedly.
Recursion
Do something based on self-similarity.
Selection from a library of patterns
Do some pattern of the previous kinds of refinement.

Stepwise Refinement: Is recursive

```
if P is simple to write :
    Write it
else:
    Refine P into simpler subprograms
    Write each subprogram
```

because it uses itself for writing each subprogram.

Stepwise Refinement: Terminates

```
if P is simple to write :
    Write it
else:
    Refine P into simpler subprograms
    Write each subprogram
```

provided the subprograms get simpler to write.

Stepwise Refinement: Terminates when *P* is so simple that you just write it.

```
if P is simple to write :
    Write it
…
```

This is called the base case of the recursion.

Stepwise Refinement: The subproblems of each refinement must fit together like pieces of a jigsaw puzzle.

We now consider each of the five kinds of refinement.

Sequential Refinement: Implement a specification *P* with a sequence of steps P_1 through P_n executed one after the other.

```
# Specification P.
# ---------------
#.Specification P_1.
#.Specification P_2.
...
#.Specification P_n.
```

where if any "#.Specification P_i" is simple enough, it can be just code.

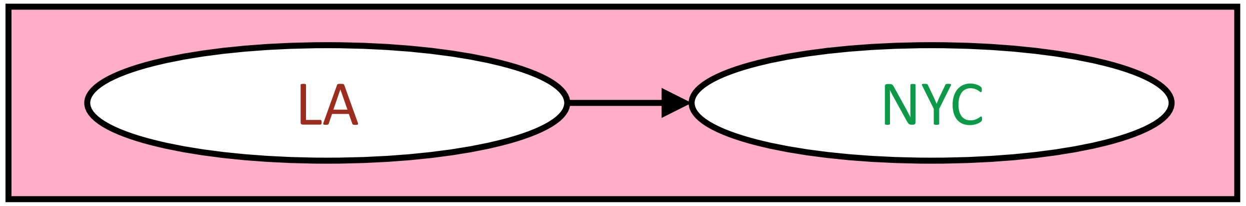
Example 1: A top-level specification

```
#.Drive from LA to NYC.
```

Sequential Refinement Examples

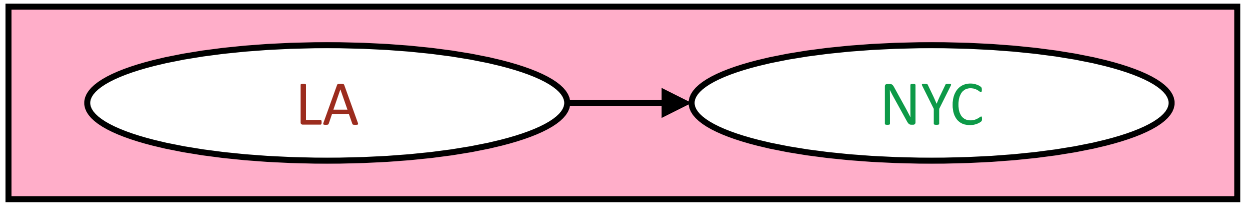
Example 1: A top-level specification that calls for the state-space effect shown.

```
#.Drive from LA to NYC.
```



Example 1: A Sequential Refinement

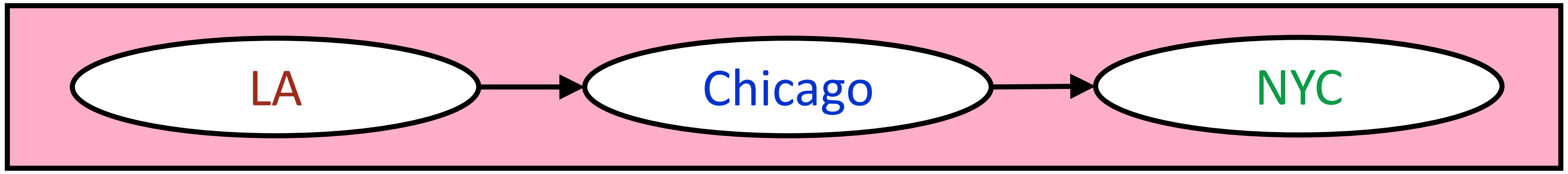
```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```



Sequential Refinement Examples

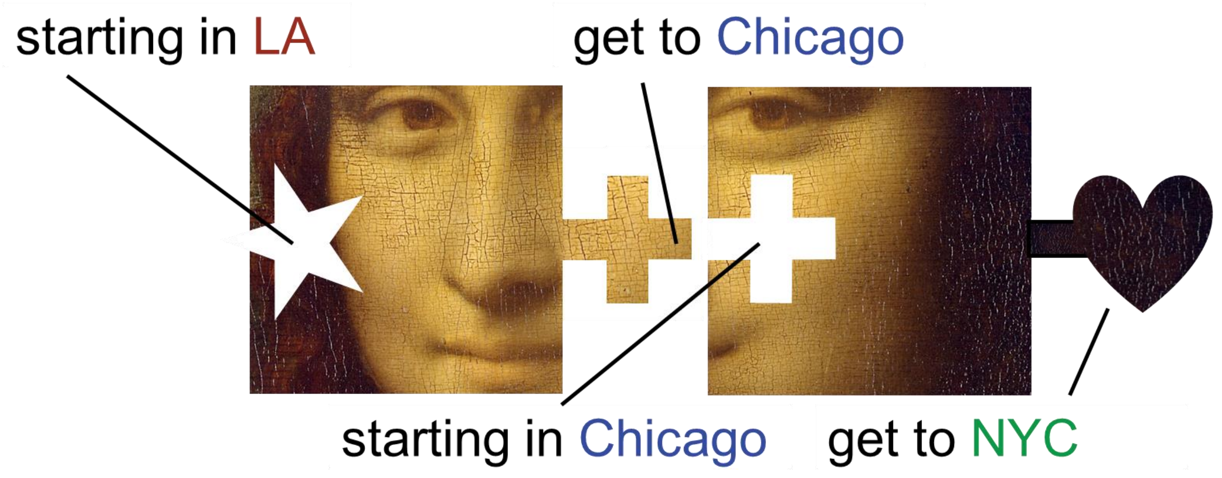
Example 1: A Sequential Refinement

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```



Example 1: A Sequential Refinement

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```



Sequential Refinement Examples

Example 2: A different Sequential Refinement

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to St. Louis.
#.Drive from St. Louis to NYC.
```

Different roads and scenery, but the same net effect (the *external interface*):
If I leave from LA, I will get to NYC.

Example 3: An incorrect Sequential Refinement

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from St. Louis to NYC.
```

The first step does not establish what the second step requires.

Example 3: A corrected Sequential Refinement

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to St. Louis.
#.Drive from St. Louis to NYC.
```

Example 4: An infeasible Sequential Refinement

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Tokyo.
#.Drive from Tokyo to NYC.
```

You can’t drive from LA to Tokyo.

Just because you can express a requirement doesn’t mean that it can be accomplished.

Example 1, continued:

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```

Example 1, continued:

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```

☞ **Refine specifications and placeholders in an order that makes sense for development, without regard to execution order.**

Example 1, continued:

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.

# Drive from Chicago to NYC.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Drive from Chicago to Pittsburgh.
#.Drive from Pittsburgh to NYC.
```

☞ **Refine specifications and placeholders in an order that makes sense for development, without regard to execution order.**

Example 4, continued: Backtrack out of an infeasible Sequential Refinement

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Tokyo.
#.Drive from Tokyo to NYC.
```

You can’t drive from LA to Tokyo.

Example 4, continued: Backtrack out of an infeasible Sequential Refinement

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Tokyo.
#.Drive from Tokyo to NYC.
```

You can’t drive from LA to Tokyo.

☞ **Don’t be wedded to code. Revise and rewrite when you discover a better way.**

Example 4, continued: An infeasible Sequential Refinement undone.

```
#.Drive from LA to NYC.
```

☞ **Don’t be wedded to code. Revise and rewrite when you discover a better way.**

Example 4, continued: An infeasible Sequential Refinement revised.

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Denver.
#.Drive from Denver to NYC.
```

You can drive from LA to Denver and from Denver to NYC.

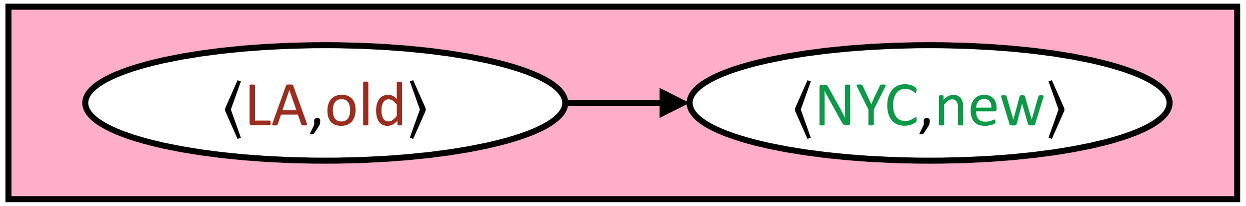
☞ **Don’t be wedded to code. Revise and rewrite when you discover a better way.**

Example 5: A new top-level specification

```
#.Drive from LA to NYC and buy a new car (in any order).
```

Example 5: A new top-level specification

```
#.Drive from LA to NYC and buy a new car (in any order).
```



Sequential Refinement

Examples

Example 5: One possible order

```
# Drive from LA to NYC and buy a new car (in any order).
# -------------------------------------------------------
#.Buy a new car.
#.Drive from LA to NYC.
```

Example 5: Another possible order

```
# Drive from LA to NYC and buy a new car (in any order).
# -------------------------------------------------------
#.Drive from LA to NYC.
#.Buy a new car.
```

Example 5: and its possible refinement.

```
# Drive from LA to NYC and buy a new car (in any order).
# ------------------------------------------------------
# Drive from LA to NYC.
# ~~~~~~~~~~~~~~~~~~~~~
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.

#.Buy a new car (in NYC).
```

Implicitly, unmentioned components of state may not be changed.

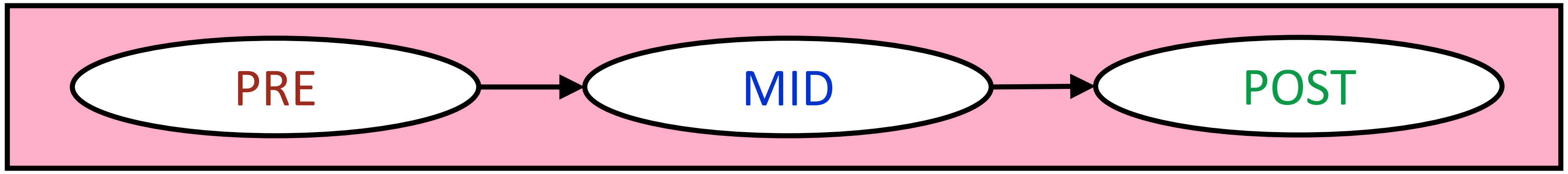
Example 5: and its possible refinement.

```
# Get from ⟨LA,old⟩to ⟨NYC,new⟩.
# ------------------------------
# Get from ⟨LA,old⟩ to NYC,old⟩.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Get from ⟨LA,old⟩ to ⟨Chicago,old⟩.
#.Get from ⟨Chicago,old⟩ to NYC,old⟩.

#.Get from ⟨NYC,old⟩ to ⟨NYC,new⟩.
```

I.e., the convention that unmentioned state components may not be changed implies that the previous version would be as shown above.

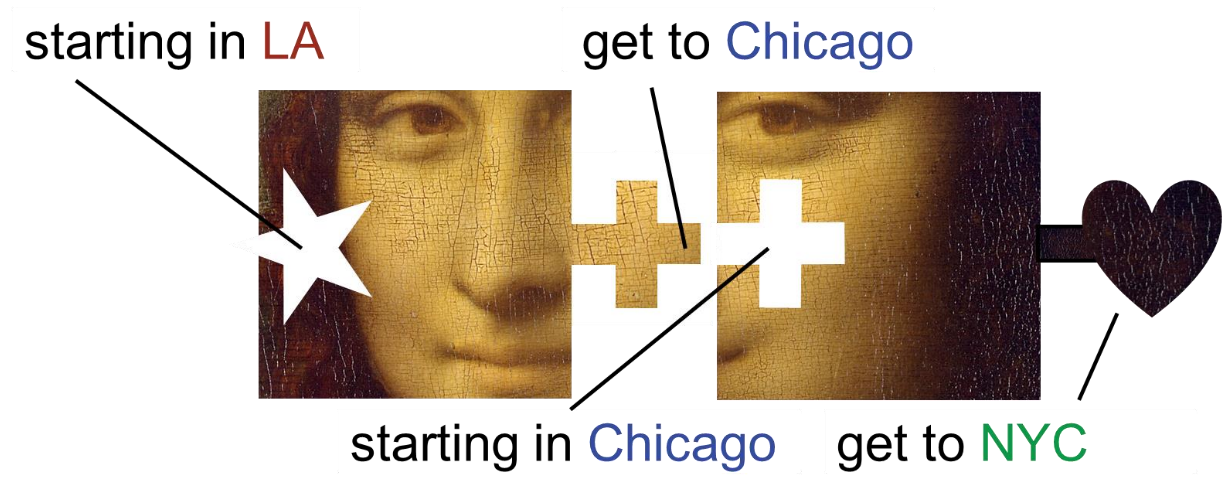
Generalization:

```
# Get from PRE to POST.
# ---------------------
#.Get from PRE to MID.
#.Get from MID to POST.
```



Loosening the Coupling: Between the two sub-steps

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to NYC.
```

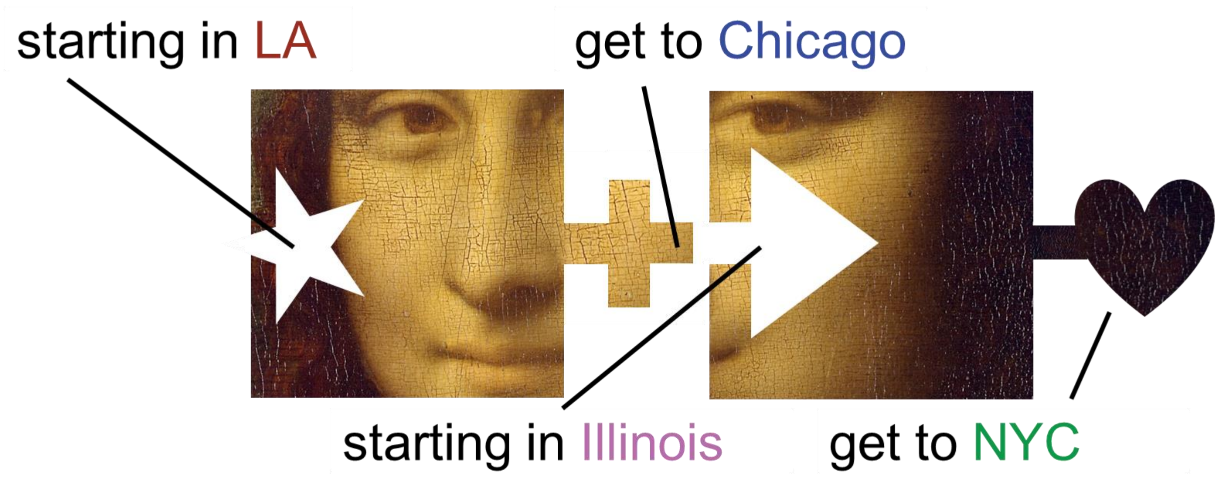


Sequential Refinement

Generalization

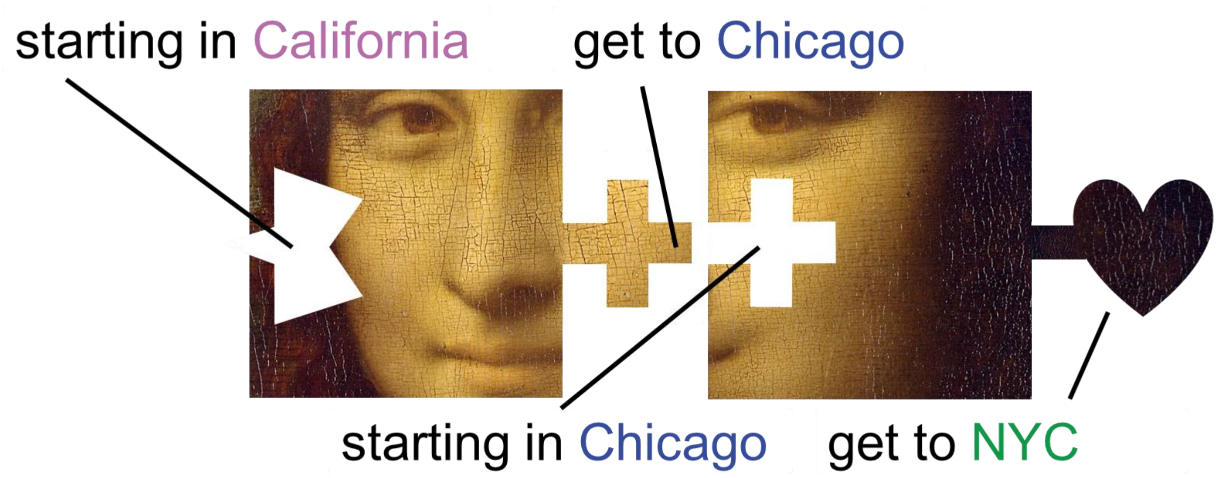
Loosening the Coupling: by weakening a precondition

```
# Drive from LA to NYC.
# ---------------------
#.Drive from LA to Chicago.
#.Drive from Illinois to NYC.
```



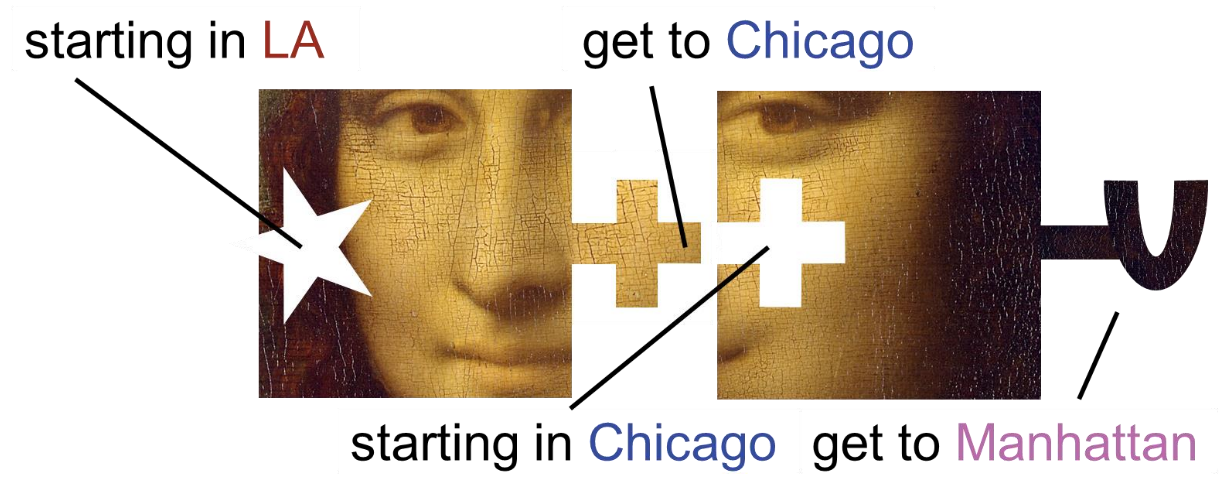
Loosening the Coupling: by weakening a precondition

```
# Drive from LA to NYC.
# ----------------------
#.Drive from California to Chicago.
#.Drive from Chicago to NYC.
```



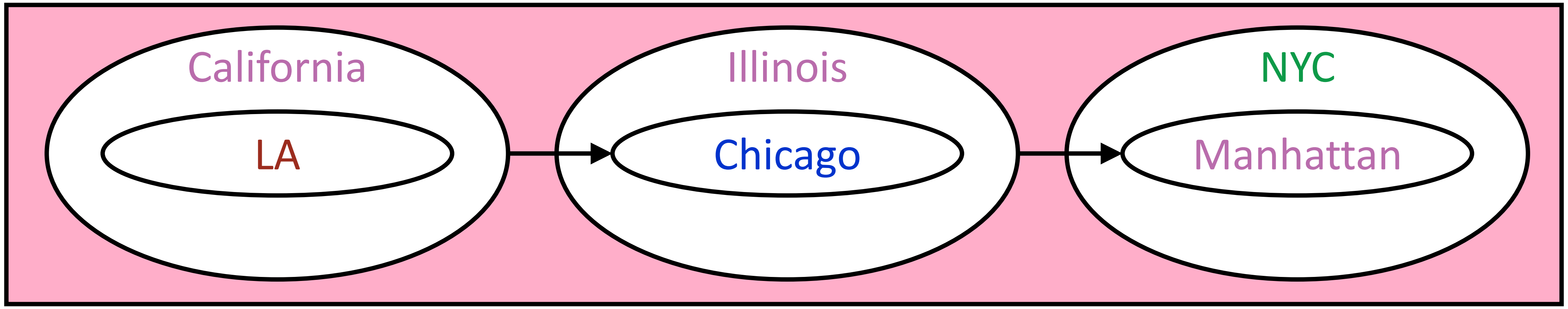
Loosening the Coupling: or by strengthening a postcondition

```
# Drive from LA to NYC.
# ----------------------
#.Drive from LA to Chicago.
#.Drive from Chicago to Manhattan.
```



Loosening the Coupling: or by doing both.

```
# Drive from LA to NYC.
# ---------------------
#.Drive from California to Chicago.
#.Drive from Illinois to Manhattan.
```



2-Step Sequential Refinement: In general

```
# Specification P: Get from PRE to POST.
# ------------------------------------
#.Get from $A_1$ to $B_1$.
#.Get from $A_2$ to $B_2$.
```

where

establishing PRE automatically establishes A_1,
establishing B_1 automatically establishes A_2, and
establishing B_2 automatically establishes POST.+

n-Step Sequential Refinement: In general

```
# Specification P: Get from PRE to POST.
# ------------------------------------
#.Get from $A_1$ to $B_1$.
#.Get from $A_2$ to $B_2$.
...
#.Get from $A_n$ to $B_n$.
```

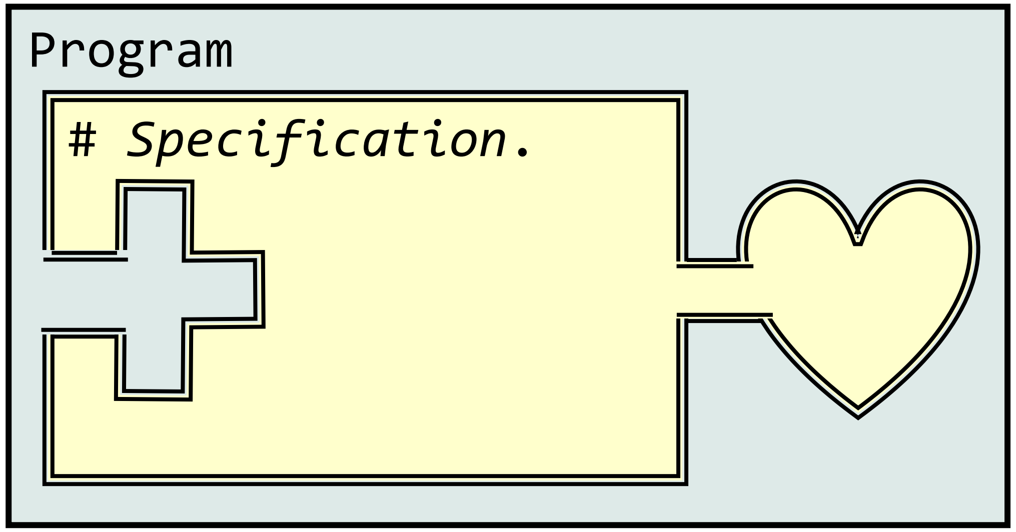
where

establishing PRE automatically establishes A_1,
establishing B_k automatically establishes A_{k+1}, for k from 1 through n-1, and
establishing B_n automatically establishes POST.

Loosening in Practice: Consider an individual specification

```
#.Get from PRE to POST.
```

in the context of a program



Loosening in Practice

Loosening in Practice: The specification

```
#.Get from PRE to POST.
```

can be implemented by any code that satisfies the specification

```
#.Get from PRE’ to POST’.
```

where PRE’ is any weakening of PRE, and POST’ is any strengthening of POST.

Example 1: Precondition is essential, but postcondition can be strengthened

```
# Get from x≥0 to y is a number that when squared equals x.
y = math.sqrt(x)
```

Any weakening of x≥0 would make the specification infeasible for real y, but we are free to choose y as either the positive or negative root of x.

Example 2: Precondition is convenient, but not essential

```
# Get from x≥0 to y is |x|.
y = x
```

The precondition $x \geq 0$ simplifies the code that sets y to the absolute value of x, because in that case the absolute value of x is just x itself.

Example 3: Precondition is irrelevant

```
# Get from x≥0 to y is x squared.
y = x * x
```

because x squared is x*x regardless of whether x is positive or negative.

Example 4: Precondition is customarily ignored

```
#.Get from array A’s elements are unique to A’s elements are
#   numerically ordered.
```

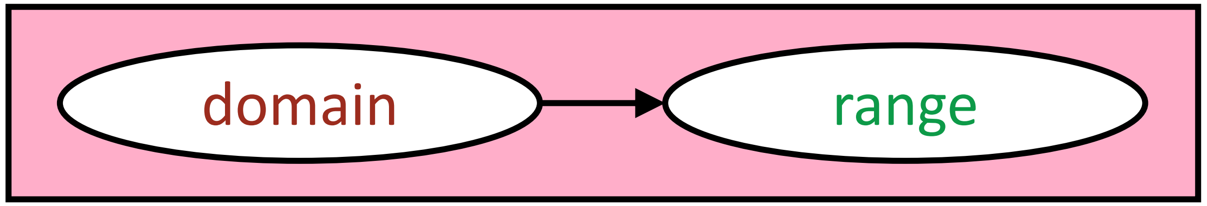
because conventional techniques for establishing the postcondition are more general, and do not rely on the given precondition.

Example 5: Chapter-1 example, revisited

```
# Given n≥0, output the integer part of the square root of n.
# ---------------------------------------------------------------
#.Given n≥0, let r be the integer part of the square root of n≥0.
print(r)
```

Consider the domain and range of the general-purpose output statement

```
print(r)
```



The domain is any state where variable r exists and contains a value, regardless of whether it is the integer square root of n. The range is any state with the additional property that the output ends with the given value.

Conjunctive Normal Form: A condition of the form

$$\boxed{C_1 \textbf{ and } C_2 \textbf{ and … and } C_n}$$

where each C_i is called a *conjunct.*

Example:

x is declared **and** x contains a value **and** x is greater than or equal to 0

state is NY **and** city is NYC

Sequential Refinement

Conjunctive Normal Form

Conjunctive Normal Form: A condition in CNF can be weakened by dropping a conjunct, e.g.,

Replace:

x is declared **and** x contains a value **and** x is greater than or equal to 0

with:

x is declared **and** x contains a value

and can be strengthened by appending an additional conjunct, e.g.,

Replace:

state is NY **and** city is NYC

with:

state is NY **and** city is NYC **and** borough is Manhattan

Implicit Preconditions: In practice, explicit preconditions are often omitted.

```
# Get from LA to NYC.
# --------------------
#.Get to Chicago.
#.Get to St. Louis.
#.Get to NYC.
```

implicitly means

```
# Get from LA to NYC.
# --------------------
#.(Given that we are in LA) Get to Chicago.
#.(Given that we are in Chicago) Get to St. Louis.
#.(Given that we are in St. Louis) Get to NYC.
```

Sequential Refinement

Implicit Preconditions

Implicit Preconditions: The reader of

```
# Get from LA to NYC.
# --------------------
#.Get to Chicago.
#.Get to St. Louis.
#.Get to NYC.
```

must infer the relevant precondition, and scan backwards to confirm that it has been established and survives, i.e., has not subsequently been invalidated.

Sequential Refinement

Implicit Preconditions

Implicit Preconditions: Minimize the distance between code that establishes a precondition, and code that relies on it, if possible.

```
k = 0
# 10 pages of code to do whatever.
k += 1
```

If the 10 pages have nothing to do with variable k, the following is better

```
k = 0
whatever()
k += 1
```

☞ **Many short procedures are better than large blocks of code.**

Implicit Preconditions

Implicit Preconditions: Minimize the distance between code that establishes a precondition, and code that relies on it, if possible, especially if the procedure can be placed outside of the scope of such a variable k.

If the distance remains great, consider an explicit indication of where the precondition was established:

```
#.Given PRE (established at point p in the code), get to POST.
```

☞ **Many short procedures are better than large blocks of code.**

Problem Reduction: A special case of Sequential Refinement

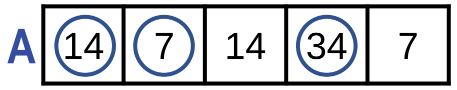
Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?

A	14	7	14	34	7

Problem Reduction: An example

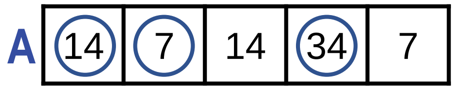
How many distinct values occur in an **`int`** array `A[0..n-1]`?



Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?

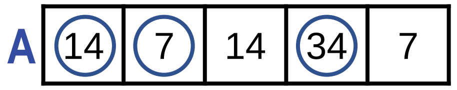


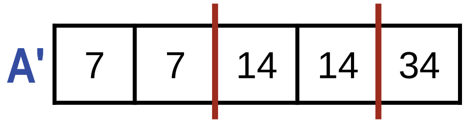
Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

☞ **Solve a different problem, and use that solution to solve the original problem.**

Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?



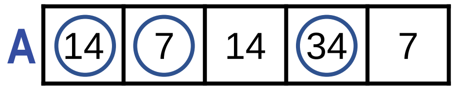


Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

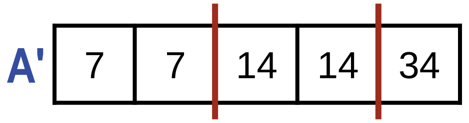
☞ **Solve a different problem, and use that solution to solve the original problem.**

Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?



Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

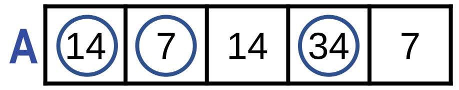


1 + the number of adjacent pairs of unequal elements in `A'`, a version of A rearranged into numerical order.

☞ **Solve a different problem, and use that solution to solve the original problem.**

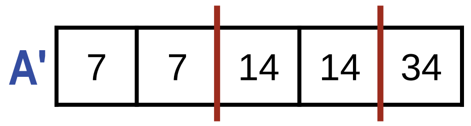
Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?



Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

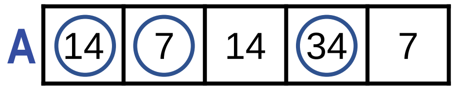
In worst case, running time is proportional to n^2.

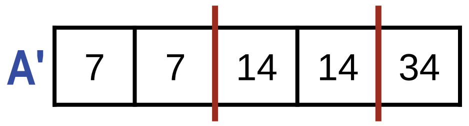


1 + the number of adjacent pairs of unequal elements in `A'`, a version of A rearranged into numerical order.

Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?





Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

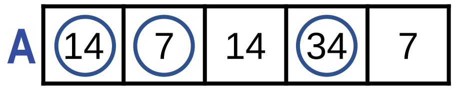
1 + the number of adjacent pairs of unequal elements in `A'`, a version of A rearranged into numerical order.

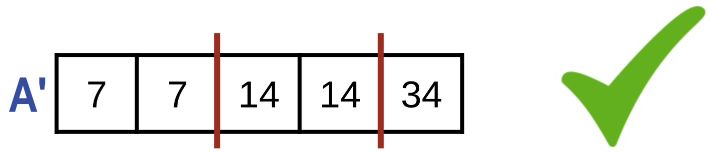
In worst case, running time is proportional to n^2.

In worst case, running time is proportional to n log n, i.e., time to sort an array of length n + time to count the number of unequal adjacent element pairs.

Problem Reduction: An example

How many distinct values occur in an **`int`** array `A[0..n-1]`?





Tally of each first instance of a value i.e., each `A[k]` for which that value doesn't occur in `A[0..k-1]`, for k from 0 through n-1.

1 + the number of adjacent pairs of unequal elements in `A'`, a version of A rearranged into numerical order.

In worst case, running time is proportional to n^2.

In worst case, running time is proportional to n log n, i.e., time to sort an array of length n + time to count the number of unequal adjacent element pairs.

Problem Reduction: In general

```
# Specification P: Get from PRE to POST.
# ----------------------------------------
#.Get from PRE to A.
#.Get from B to POST.
```

where establishing **A** automatically establishes **B**.

Problem Reduction: In general

```
# Specification P: Get from PRE to POST.
# ------------------------------------
#.Get from PRE to A.
#.Get from B to POST.
```

where establishing **A** automatically establishes **B**.

☞ **Solve a different problem, and use that solution to solve the original problem.**

Problem Reduction: In general

```
# Specification P: Get from PRE to POST.
# -------------------------------------
# Get from PRE to A.
# ~~~~~~~~~~~~~~~~~~
#.Define problem P′ based on PRE.
#.Solve problem P′.
#.Establish A from the solution to P′.

#.Get from B to POST.
```

where establishing **A** automatically establishes **B**.

☞ **Solve a different problem, and use that solution to solve the original problem.**

Case Analysis: Implement a specification *P* as a choice of one step to execute from among P_1, … , P_n.

```
# Specification P.
if condition_1:
    #.Specification P_1.
elif condition_2:
    #.Specification P_2.
...
elif condition_{n-1}:
    #.Specification P_{n-1}.
else:
    #.Specification P_n.
```

Case Analysis

Case Analysis: Implement a specification *P* as a choice of one step to execute from among *P_1* and *P_2*.

```
# Specification P.
if condition₁:
    #.Specification P₁.




else:
    #.Specification P₂.
```

Case Analysis: Implement a specification *P* as a choice of one step to execute or not.

```
# Specification P.
if condition_1:
    #.Specification P_1.
```

Case Analysis: Implement a specification *P* as a choice of one step to execute from among P_1, … , P_n.

```
# Specification P.
    if condition_1:
        #.Specification P_1.
    elif condition_2:
        #.Specification P_2.
    ...
    elif condition_{n-1}:
        #.Specification P_{n-1}.
    else:
        #.Specification P_n.
```

Appropriate when distinct program behaviors are required for different situations.

Case Analysis

Case Analysis: Implement a specification *P* as a choice of one step to execute from among $P_1, \ldots, P_n$.

- In the real world: Animal, vegetable, or mineral?
- In a maze: Facing a wall or not?
- After a search: Found or not found?
- In mathematics: Positive or negative? Even or odd? Real or imaginary roots?

Appropriate when distinct program behaviors are required for different situations.

Case Analysis: An example

```
#.Let y be |x|.
```

Appropriate when distinct program behaviors are required for different situations.

Case Analysis: An example

```
# Let y be |x|.
if x >= 0:
    y = x
else:
    y = -x
```

Appropriate when distinct program behaviors are required for different situations.

Case Analysis: An example

```
# Let y be |x|.
if x >= 0:
    y = x
else:
    y = -x
```

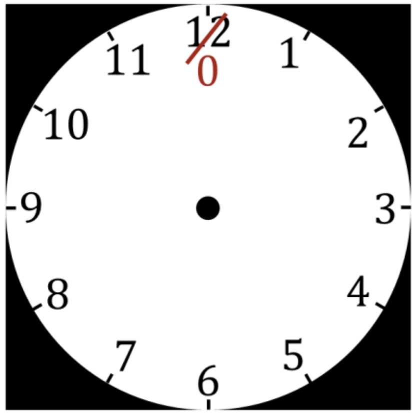
☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: An example

```
# Let y be |x|.
y = abs(y)
```

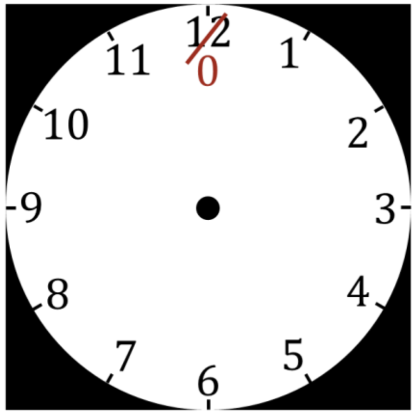
☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: A second example

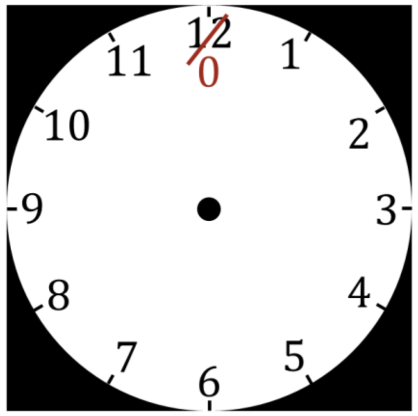


```
#.Advance k to the next hour.
```

Case Analysis: A second example



```
# Advance k to the next hour.
if k == 11:
    k = 0
else:
    k = k + 1
```

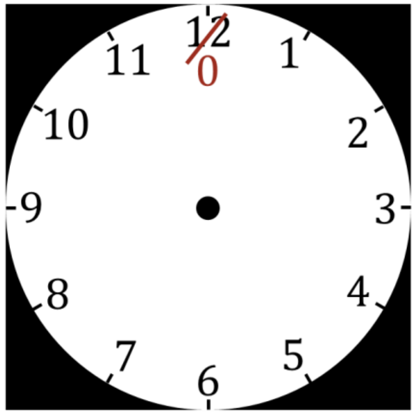


Case Analysis: A second example

```
# Advance k to the next hour.
if k == 11:
    k = 0
else:
    k = k + 1
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

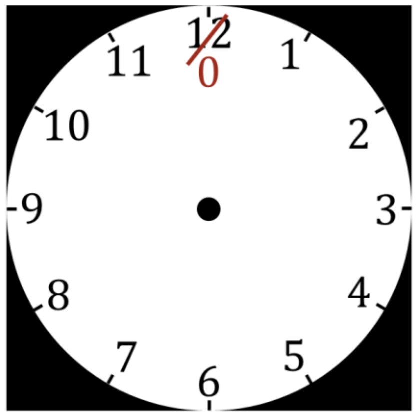
Case Analysis: A second example



```
# Advance k to the next hour.
k = (k + 1) % 12
```

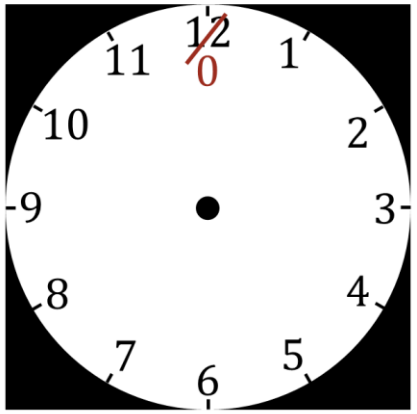
☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: A third example



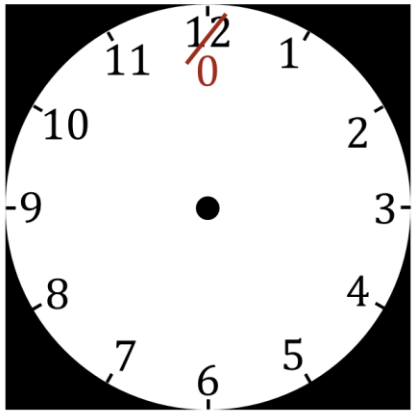
```
#.Advance k to the previous hour.
```

Case Analysis: A third example



```
# Advance k to the previous hour.
if k == 0:
    k = 11
else:
    k = k - 1
```

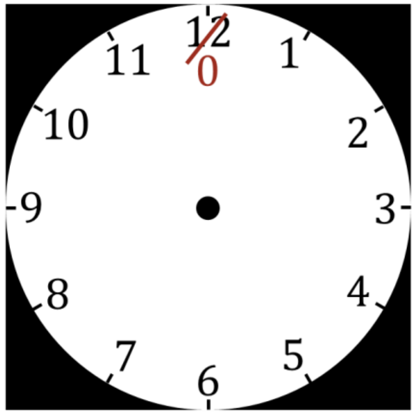
Case Analysis: A third example



```
# Advance k to the previous hour.
if k == 0:
    k = 11
else:
    k = k - 1
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

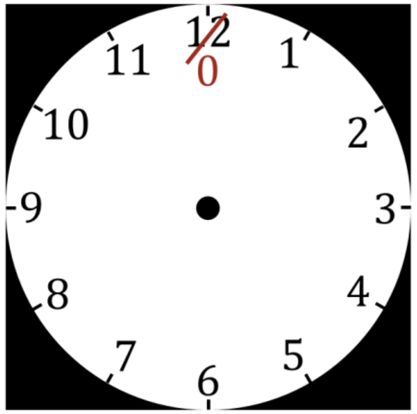
Case Analysis: A third example



```
# Advance k to the previous hour.
k = (k + 11) % 12
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: A third example

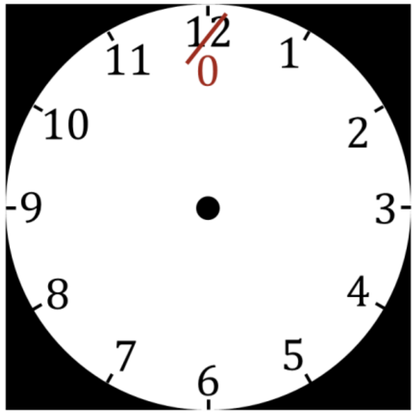


```
# Advance k to the previous hour.
k = (k + 11) % 12
```

Why not `(k-1)%12` ?

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: A third example



```
# Advance k to the previous hour.
k = (k + 11) % 12
```

Why not `(k-1)%12 ?`

This would be fine in Python, but in other languages (e.g., Java), `(k-1)%12` is negative for negative `k-1`. The issue is avoided if we write (k+11)%12

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

☞ **Be alert to high-risk coding steps associated with binary choices: "`==`" or "`!=`", "`<`" or "`<=`", "`x`" or "`x-1`", *condition* or `not(`*condition*`)`, positive or negative, 0-origin or 1-origin, "even integers are divisible by 2, but array segments of odd length have middle elements".**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

☞ **Be alert to high-risk coding steps associated with binary choices.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: parity

```
#.Output whether n is odd or even.
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: parity

```
# Output whether n is odd or even.
if (n % 2) == 1:
    print("odd")
else:
    print("even")
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: parity

```
# Output whether n is odd or even.
if (n % 2) == 1:
    print("odd")
else:
    print("even")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: parity

```
# Output whether n is odd or even.
if (n % 2) == 1:
    print("odd")
else:
    print("even")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

The code is correct in Python, but because in some languages (e.g., Java) n%2 is negative for negative n, the code is fragile.

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: parity, more robust

```
# Output whether n is odd or even.
if (n % 2) == 0:
    print("even")
else:
    print("odd")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: roots, real or imaginary

```
#.Let im be True iff the roots of quadratic Ax2+Bx+C=0 are imaginary.
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax²+Bx+C=0 are imaginary.
im: bool  # Roots are imaginary.
if ((B * B) – (4 * A * C)) < 0:
    im = True
else:
    im = False
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax2+Bx+C=0 are imaginary.
im: bool  # Roots are imaginary.
if ((B * B) – (4 * A * C)) < 0:
    im = True
else:
    im = False
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

<u>Example</u>: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax^2+Bx+C=0 are imaginary.
im: bool  # Roots are imaginary.
if ((B * B) – (4 * A * C)) < 0:
    im = True
else:
    im = False
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Is the case of `((B*B)-(4*A*C))==0` correct?

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax^2+Bx+C=0 are imaginary.
im: bool  # Roots are imaginary.
if ((B * B) – (4 * A * C)) < 0:
    im = True
else:
    im = False
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Is the case of `((B*B)-(4*A*C))==0` correct? Yes.

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

<u>Example</u>: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax^2+Bx+C=0 are imaginary.
im: bool  # Roots are imaginary.
if ((B * B) – (4 * A * C)) < 0:
    im = True
else:
    im = False
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

<u>Example</u>: roots, real or imaginary

```
#.Let im be True iff the roots of quadratic Ax²+Bx+C=0 are imaginary.
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: roots, real or imaginary

```
# Let im be True iff the roots of quadratic Ax²+Bx+C=0 are imaginary.
im: bool = ((B * B) – (4 * A * C)) < 0   # Roots are imaginary.
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

<u>Example</u>: Parallel or intersecting lines

```
#.Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: Parallel or intersecting lines

```
# Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
if (m1 == m2) and (b1 != b2):
    print("parallel")
else:
    print("intersect")
```

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: Parallel or intersecting lines

```
# Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
if (m1 == m2) and (b1 != b2):
    print("parallel")
else:
    print("intersect")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: Parallel or intersecting lines

```
# Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
if (m1 == m2) and (b1 != b2):
    print("parallel")
else:
    print("intersect")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

What if `m1` is 0.0e0 and `m2` is smallest floating-point number?

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: Parallel or intersecting lines

```
# Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
if (m1 == m2) and (b1 != b2):
    print("parallel")
else:
    print("intersect")
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

What if `m1==m2`, but b1 is 0.0e0 and b2 is smallest floating-point number?

Case Analysis: The *condition* in a Case Analysis is often the locus of error.

Example: Parallel or intersecting lines

```
# Output whether lines y=m1·x+b1 and y=m2·x+b2 are parallel or intersect.
if compare-slopes-and-intercepts-wrt-tolerances:
    print("parallel")
else:
    print("intersect")
```

☞ **Never test two floating-point numbers for equality or inequality.**

Iterative Refinement: Implement a specification P by repeatedly executing step *P′*.

```
# Specification P.
# ---------------
#.Setup for P′.
while condition:
    #.Specification P′.
```

Iterative Refinement

Iterative Refinement: Implement a specification P by repeatedly executing step *P′*.

```
# Specification P.
# ----------------
#.Setup for P′.
while condition:
    #.Specification P′.
```

Invariant: The thing that stays the same, and allows *P′* to remain applicable.

Variant: The thing that changes, and eventually causes the loop to stop.

Iterative Refinement: Implement a specification P by repeatedly executing step *P′*.

```
# Specification P.
# ---------------
#.Setup for P′.
while condition:
    #.Specification P′.
```

Invariant: The thing that stays the same, and allows *P′* to remain applicable.

Variant: The thing that changes, and eventually causes the loop to stop.

Infinite loops have their utility, but termination is the norm.

Iterative Refinement

Iterative Refinement: Implement a specification P by repeatedly executing step *P′*.

```
# Specification P.
# ----------------
#.Setup for P′.
while condition:
    #.Specification P′.
```

Invariant: The thing that stays the same, and allows *P′* to remain applicable.

Variant: The thing that changes, and eventually causes the loop to stop.

Infinite loops have their utility, but termination is the norm. For termination, think of the variant as a necessarily nonnegative integer that necessarily decreases by 1 on each iteration. You can only do that a finite number of times.

Iterative Refinement

Iterative Refinement: Implement a specification P by repeatedly executing step *P′*.

```
# Specification P.
# ----------------
#.Setup for P′.
while condition:
    #.Specification P′.
```

A fruitful real-world analogy: Hammering a nail into a block of wood.

Iterative Refinement

Iteration in the Real World

```
#.Drive a nail vertically into a block of wood.
```

Iterative Refinement

Iteration in the Real World

```
# Drive a nail vertically into a block of wood.
# --------------------------------------------
#.Setup: Stabilize the nail vertically, with height≥0. In doing so,
#   establish the invariant (nail vertical, and height≥0) and the
#   initial variant (height of nail).
```

Iteration in the Real World

```
# Drive a nail vertically into a block of wood.
# -------------------------------------------
#.Setup: Stabilize the nail vertically, with height≥0. In doing so,
#   establish the invariant (nail vertical, and height≥0) and the
#   initial variant (height of nail).
while the-head-of-the-nail-sticks-out :
    #.Hit the nail with the hammer squarely. In doing so, maintain the invariant (by
    #   hitting the nail vertically, but not so hard that its height becomes negative),
    #   and reduce the variant (by hitting the nail hard enough to reduce the height
    #   such that a finite number of hits suffices).
```

Iterative Refinement

Iteration in the Real World

```
# Drive a nail vertically into a block of wood.
# --------------------------------------------
#.Setup: Stabilize the nail vertically, with height≥0. In doing so,
#   establish the invariant (nail vertical, and height≥0) and the
#   initial variant (height of nail).
while the-head-of-the-nail-sticks-out :
    #.Hit the nail with the hammer squarely. In doing so, maintain the invariant (by
    #   hitting the nail vertically, but not so hard that its height becomes negative),
    #   and reduce the variant (by hitting the nail hard enough to reduce the height
    #   such that a finite number of hits suffices).
#.The invariant still holds (nail vertical, and height≥0), and the
#   variant is reduced to zero (height==0).
```

Iterative Refinement: What can go wrong?

- Setup doesn’t *establish* the nail’s verticality (the invariant). The very first hammer blow flattens the nail, or begins the process of bending it, even if the loop body is perfectly correct.

Iterative Refinement: What can go wrong?

- Loop body doesn’t *maintain* the nail’s verticality (the invariant) because it hits the nail at a crooked angle. Eventually, the nail is flattened.

Iterative Refinement: What can go wrong?

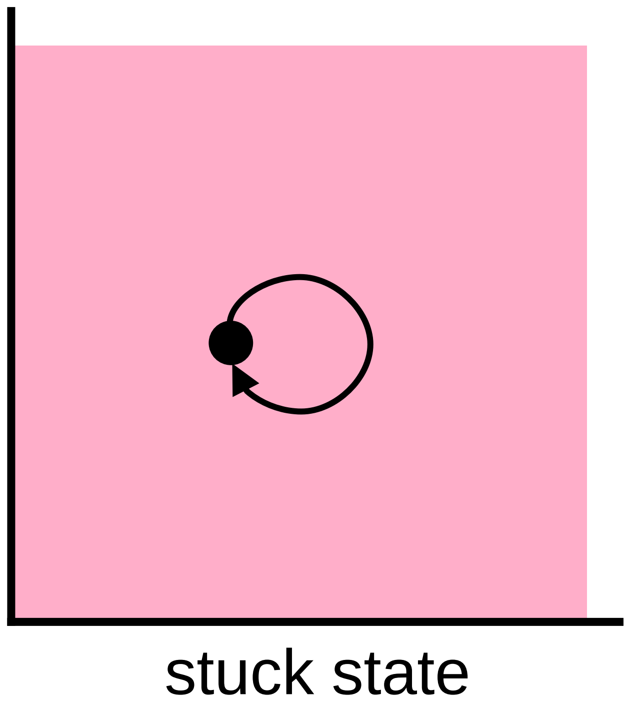
- Loop body doesn’t *maintain* the nail’s nonnegative height (the invariant), splits the wood, and the nail goes into the table top.

Iterative Refinement: What can go wrong?

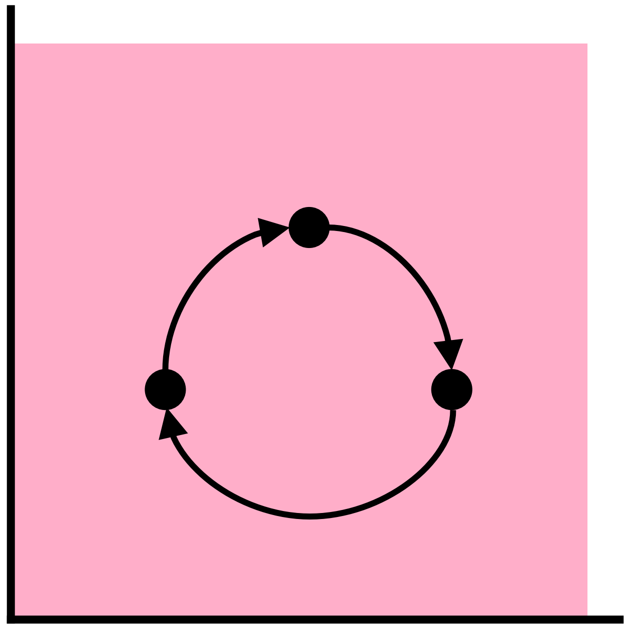
- Loop body makes *insufficient progress* (the variant). The loop runs forever and the nail never gets flush with the surface.

  This can be because the height is an infinite decreasing sequence that doesn’t converge to zero, or because you hit a knot, and stop advancing altogether.

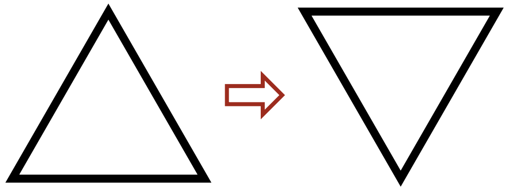
No advancement: Use a feather instead of a hammer, or at a knot.



Cyclic advancement: Movement, but destined to return to a prior state.

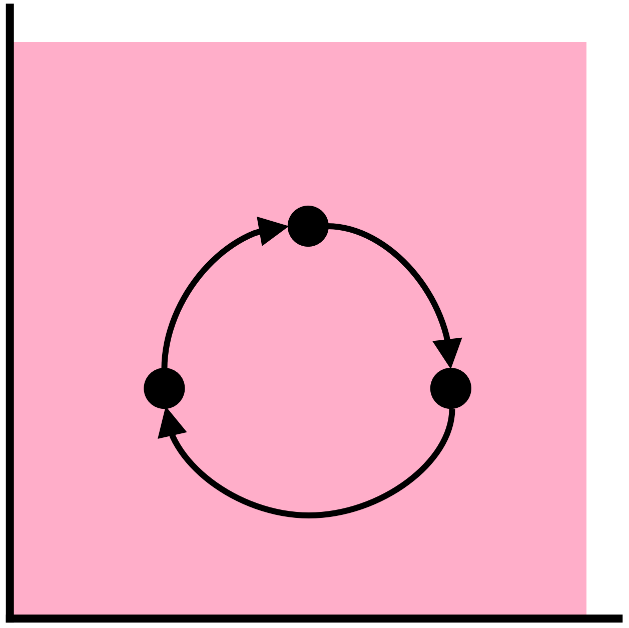


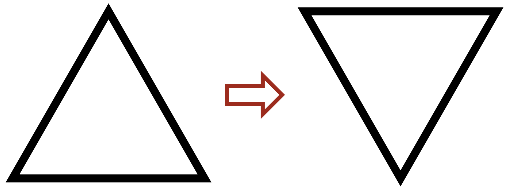
orbit of states



```
#.Make triangle point down.
```

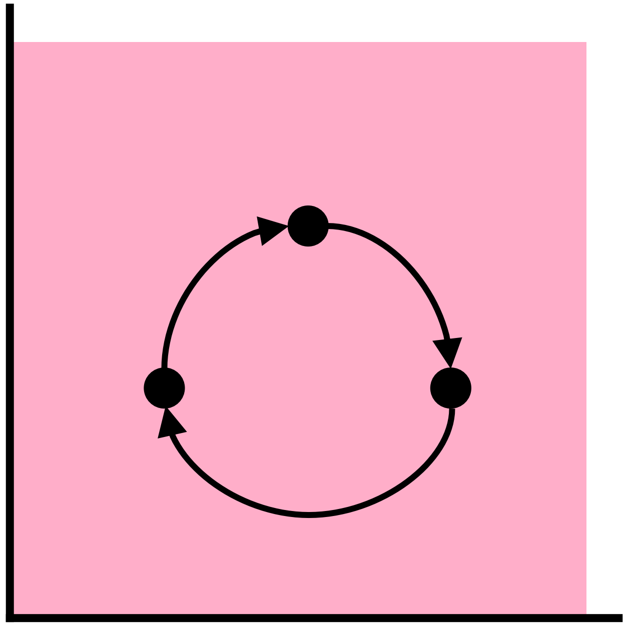
Cyclic advancement: Movement, but destined to return to a prior state.

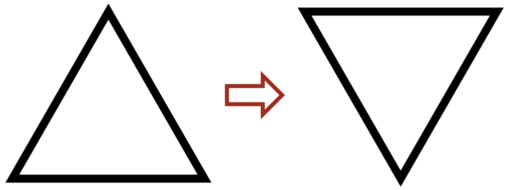

orbit of states



```
# Make triangle point down.
# ---------------------------
#.Compute angle δ.
while not-pointing-down:
    #.Turn angle δ.
```

Cyclic advancement: Movement, but destined to return to a prior state.

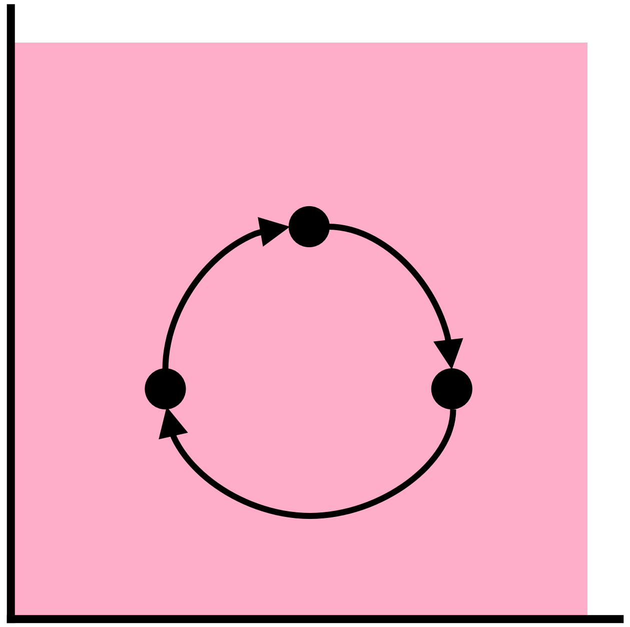




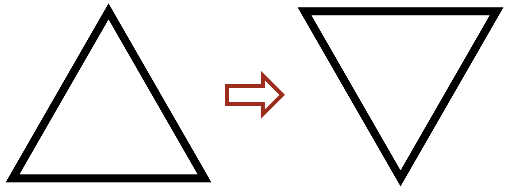
```
# Make triangle point down.
# --------------------------
#.Compute angle δ.
while not-pointing-down:
    #.Turn angle δ.
```

Runs forever if δ is 120°

Cyclic advancement: Movement, but destined to return to a prior state.



orbit of states

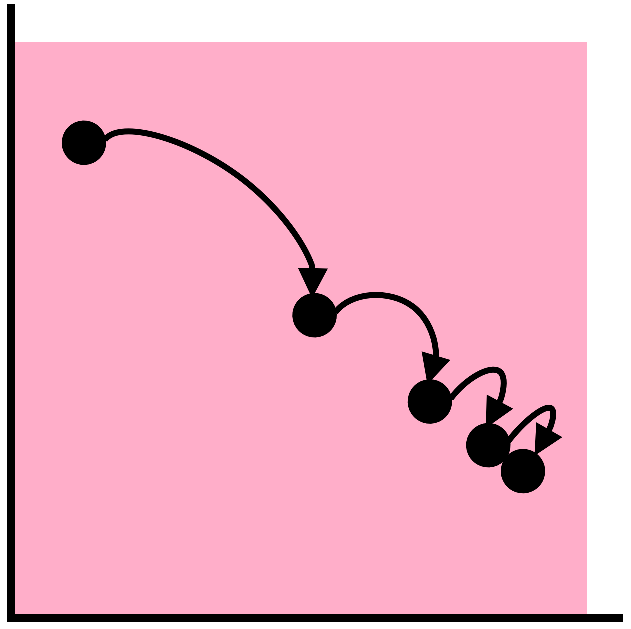


```
# Make triangle point down.
# ---------------------------
#.Compute angle δ.
while not-pointing-down:
    #.Turn angle δ.
```

Runs forever if δ is 120°

(Doesn’t happen in hammering a nail.)

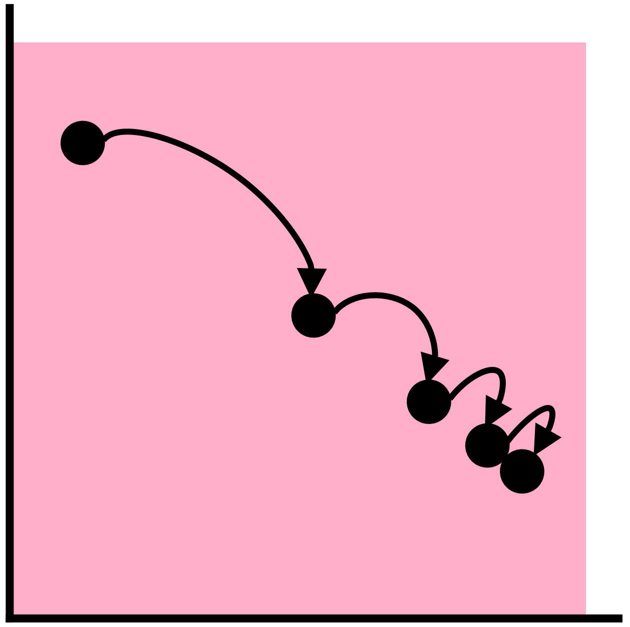
Non-convergent advancement: Variant must be a nonnegative integer that is reduced by at least 1 on each iteration.



```
h: int = 10
while h > 0:
    h = h // 2
```

Terminates

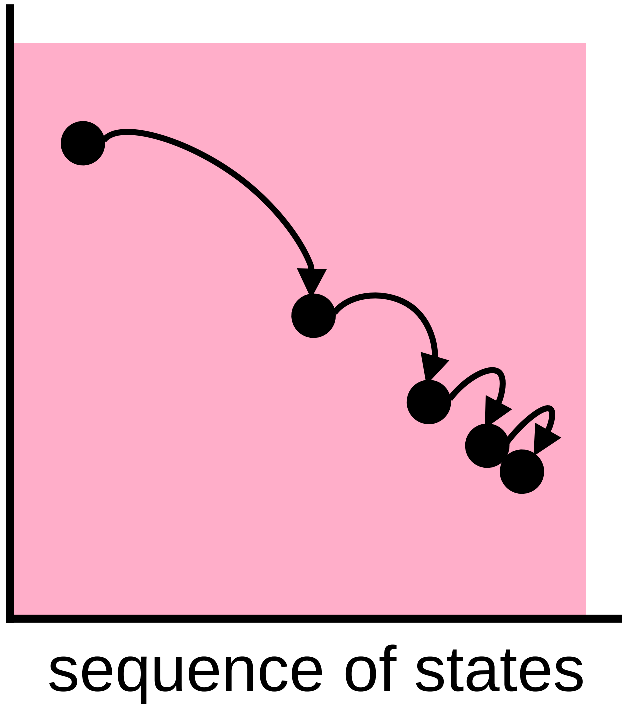
Non-convergent advancement: Variant must be a nonnegative integer that is reduced by at least 1 on each iteration.


sequence of states

```
h: float = 10.0
while h > 0:
    h = h / 2
```

Terminates

Non-convergent advancement: Variant must be a nonnegative integer that is reduced by at least 1 on each iteration.



```
zero: Rational = Rational(0,1)
two:  Rational = Rational(2,1)
h3:   Rational = Rational(10, 1)
while h3 != zero:
    h3 = rational_divide(h3, two)
```

Runs forever (Code explained in Chapter 18.)

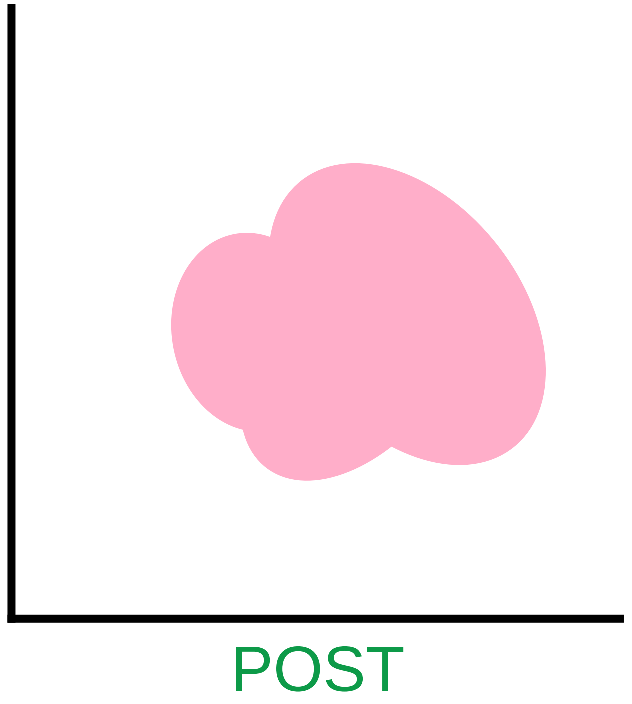
Iterative Refinement: In general

```
# Specification P: Get from PRE to POST.
# ----------------------------------------
#.Setup: Get from PRE to INVARIANT.
while condition:
    #.Get from condition and INVARIANT to INVARIANT.

where not(condition) and INVARIANT entails POST.
```

Iterative Refinement

Finding Loop Invriants

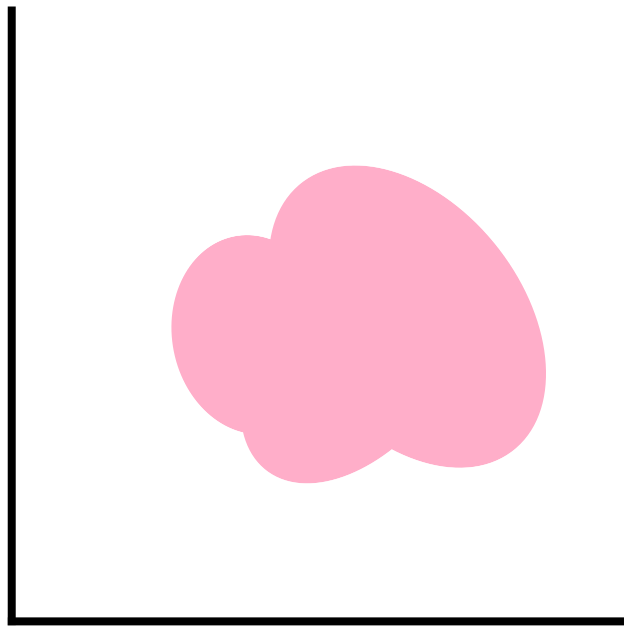
Iteration: To get to **POST** iteratively



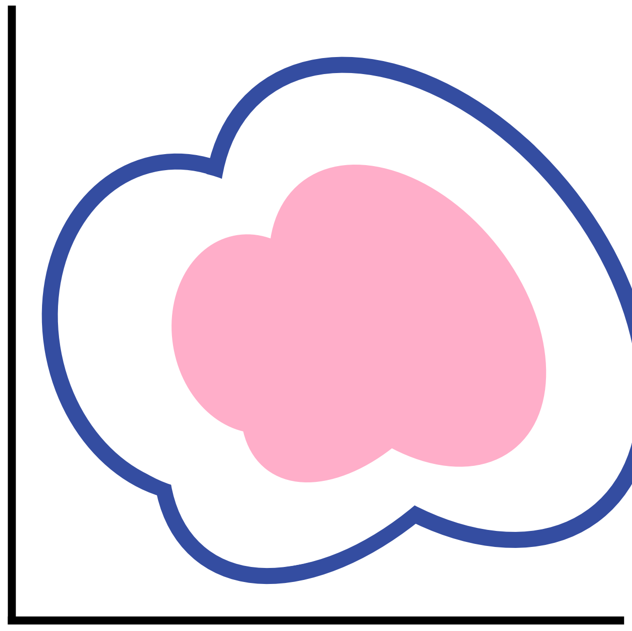
Iterative Refinement

Finding Loop Invriants

Iteration: To get to **POST** iteratively, choose a weakened **POST** as **INVARIANT**



POST

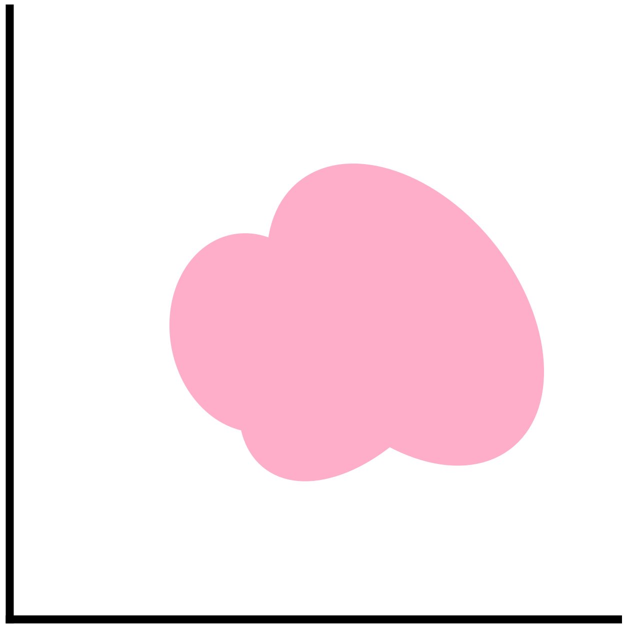


INVARIANT

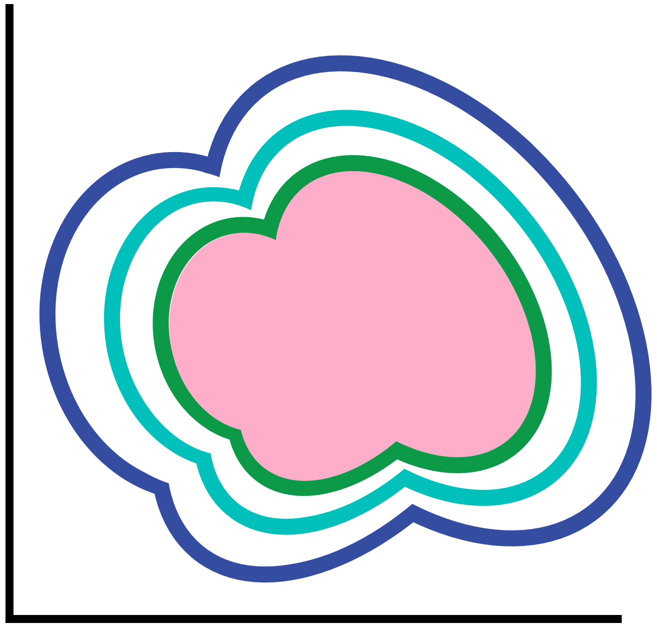
Iterative Refinement

Finding Loop Invriants

Iteration: Then, iteratively change the **INVARIANT**’s parameters.

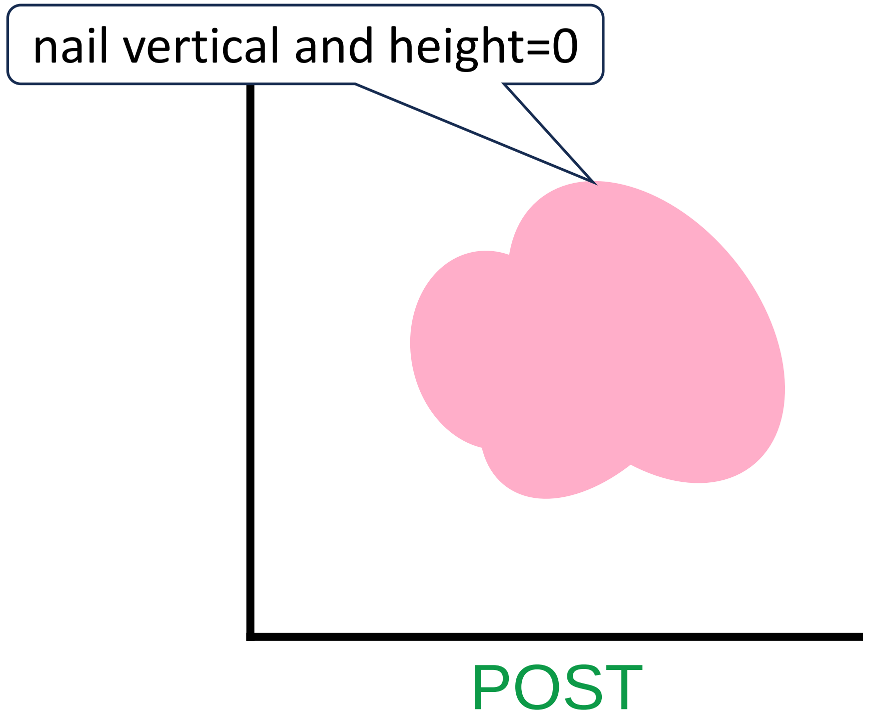


POST

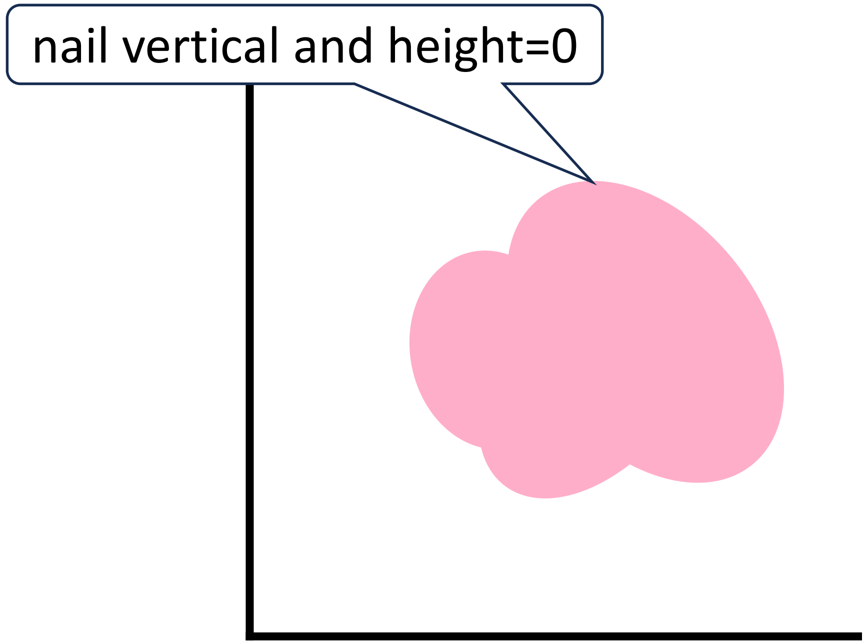


improving approximations

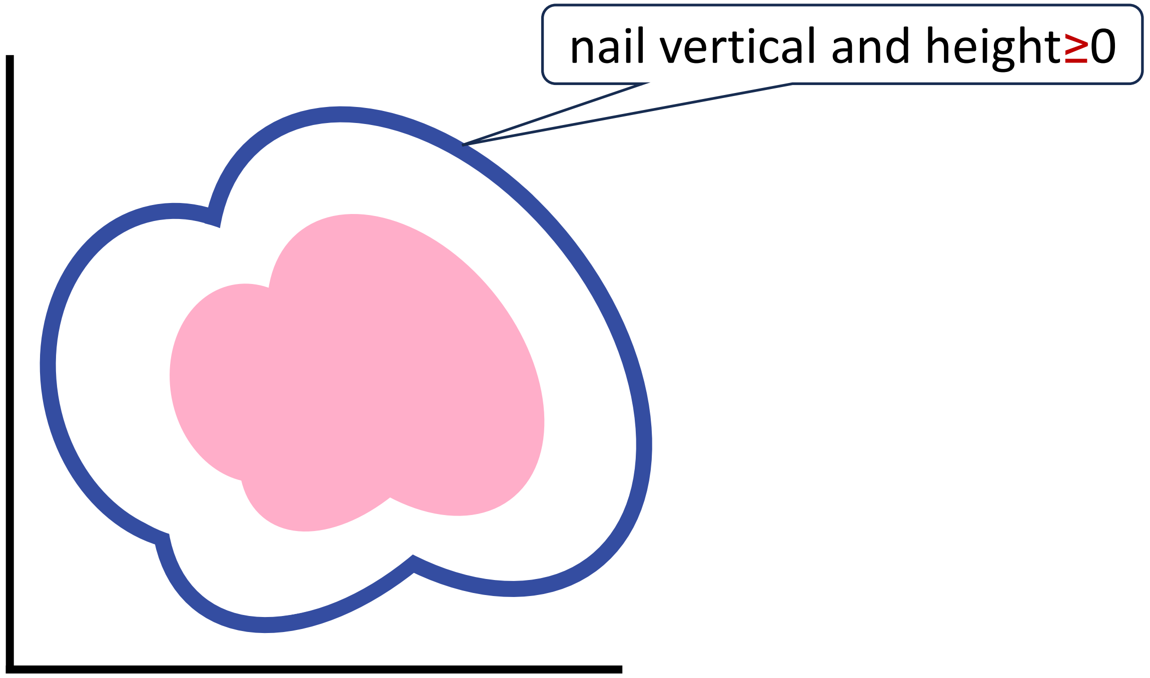
Example: Hammering a nail, the goal



Example: Hammering a nail, set up the **INVARIANT**



POST

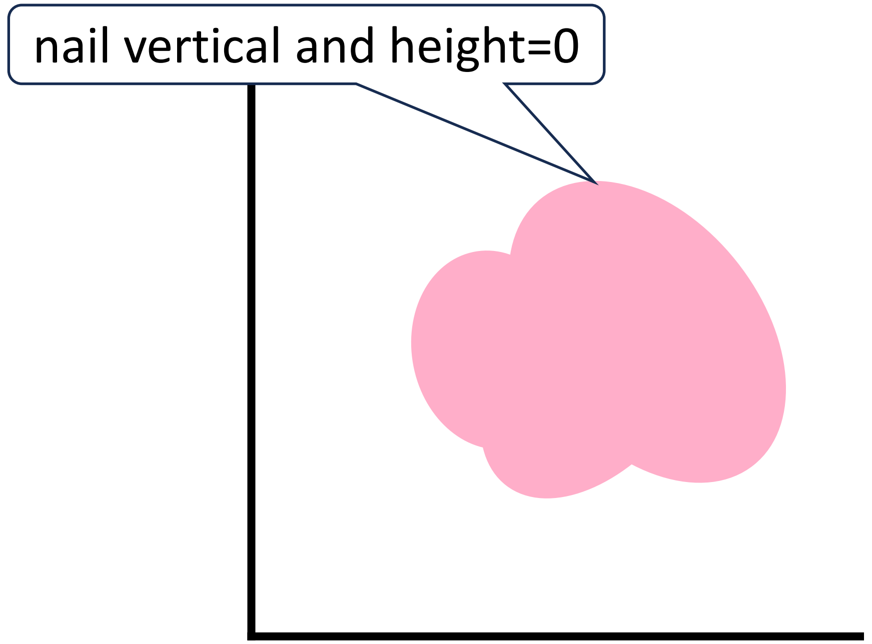


INVARIANT

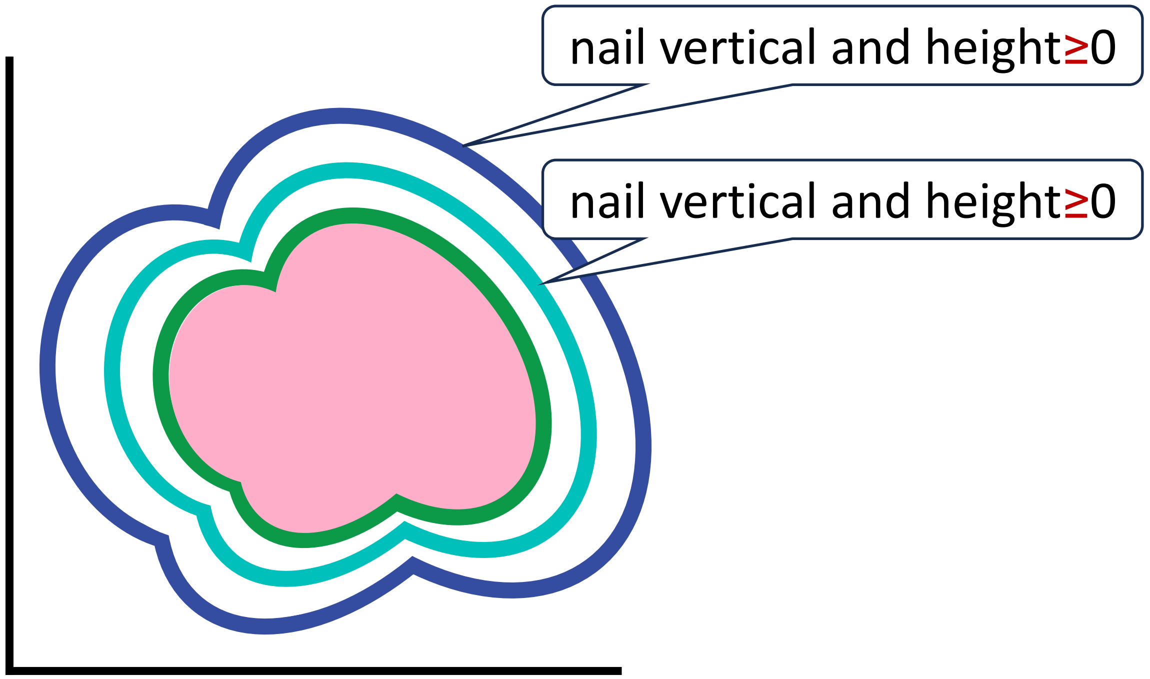
Iterative Refinement

Finding Loop Invriants

Example: Hammering a nail, the process



POST

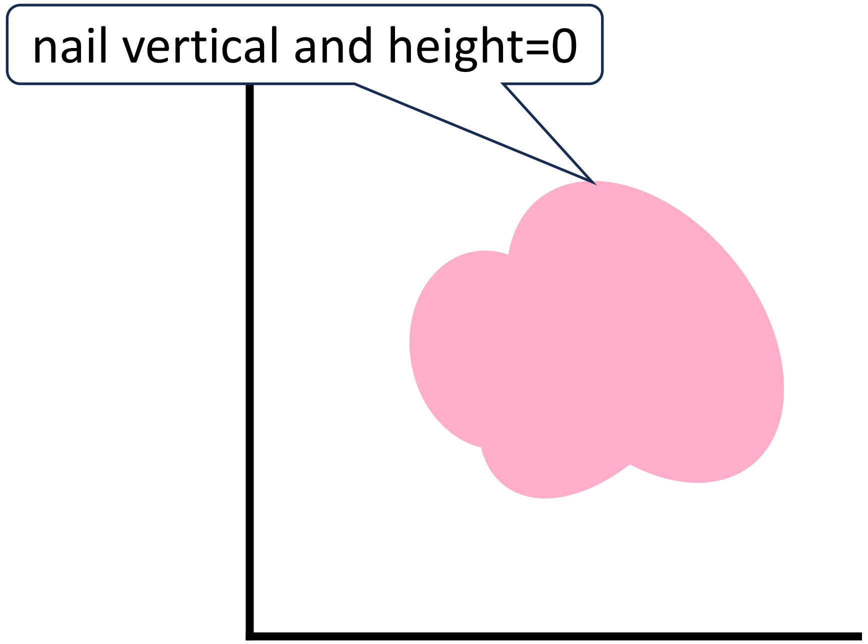


improving approximations

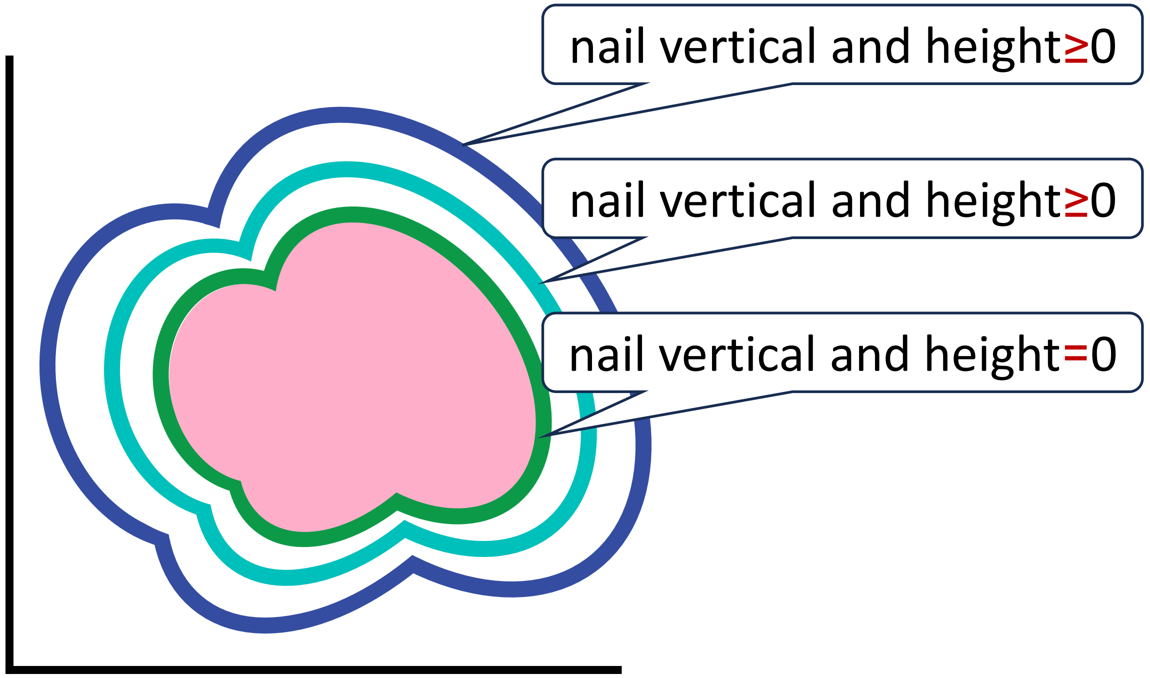
Iterative Refinement

Finding Loop Invriants

Example: Hammering a nail, the process



POST

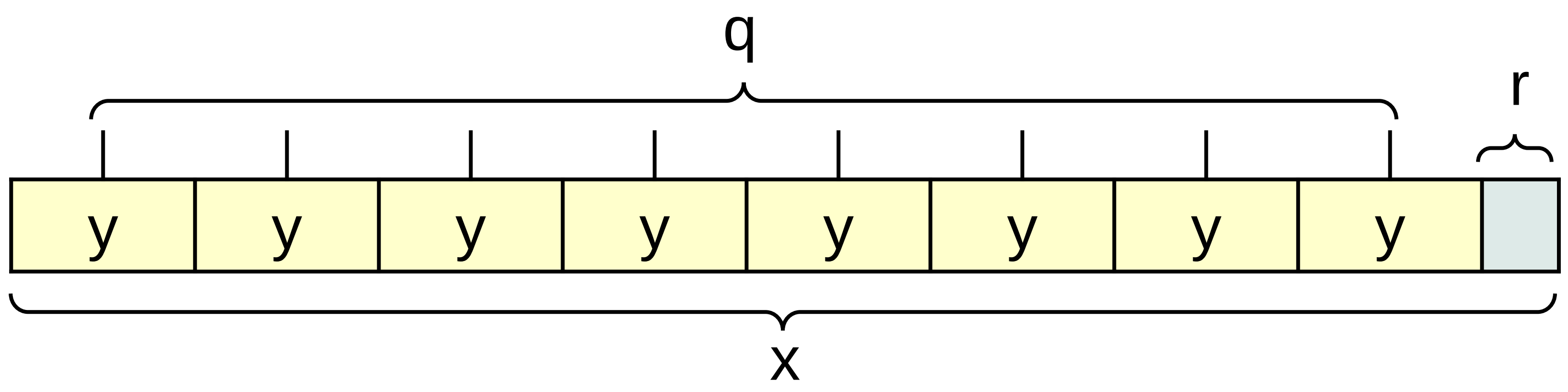

nail vertical and height≥0
nail vertical and height≥0
nail vertical and height=0

improving approximations

PRE: 0≤x, 0<y
POST: x=q*y+r, 0≤r<y, and 0≤q

Example: Integer division. The goal.

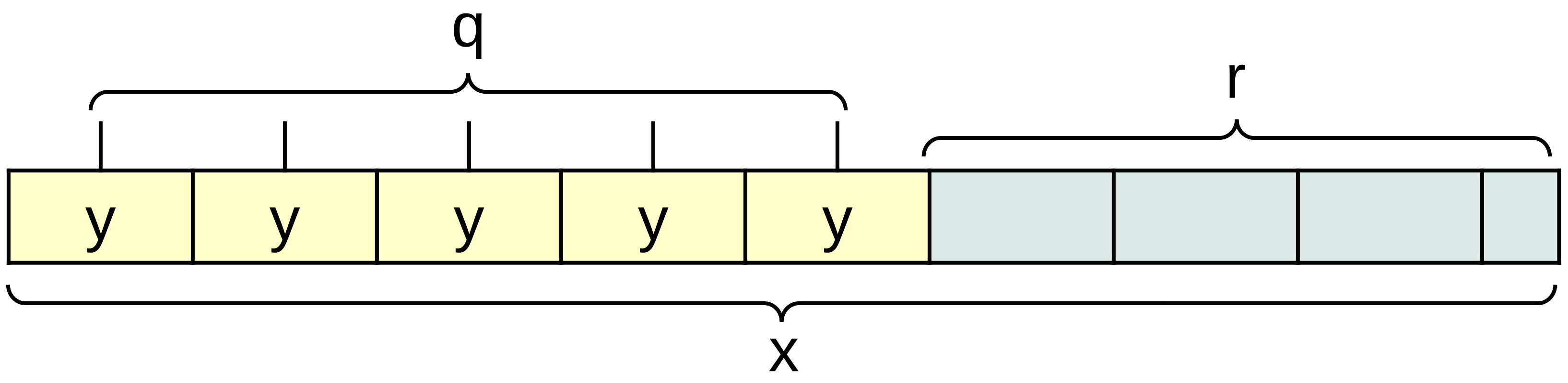
```
#.Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
```



PRE: 0≤x, 0<y
POST: x=q*y+r, 0≤$\boxed{r<y}$, and 0≤q
INVARIANT:

Example: Integer division. The process: Choose the **INVARIANT** …

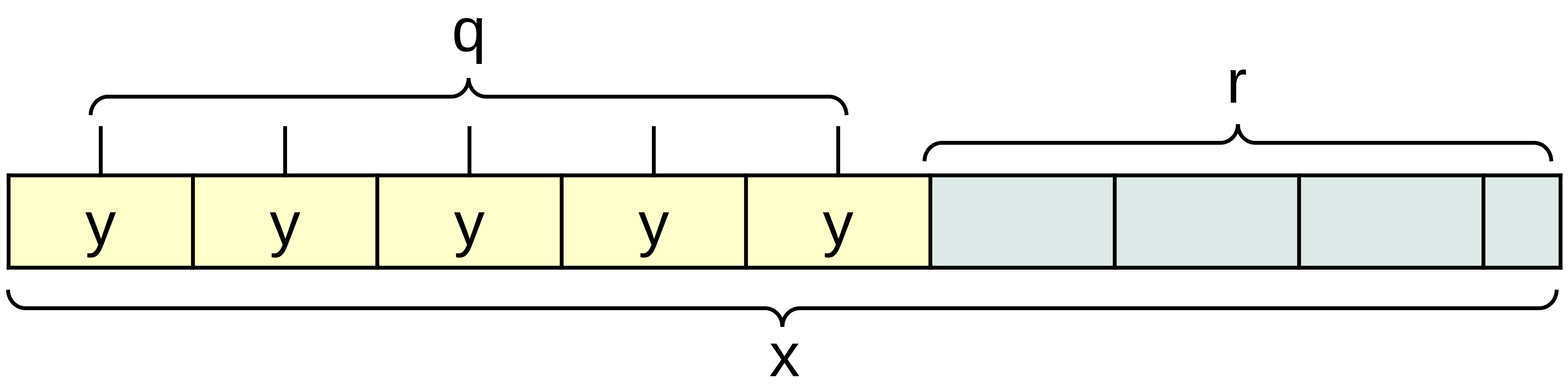
```
#.Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
```



PRE: 0≤x, 0<y
POST: x=q*y+r, 0≤$\boxed{\text{r<y}}$, and 0≤q
INVARIANT: x=q*y+r, 0≤r , and 0≤q

Example: Integer division. … **INVARIANT** as a weakened **POST**.

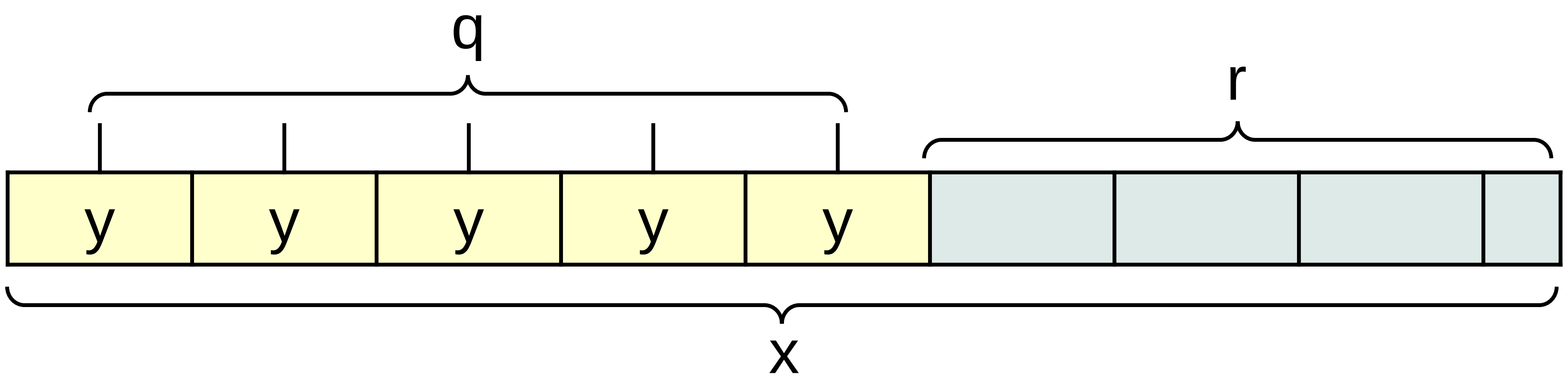
```
#.Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
```



PRE: 0≤x, 0<y
POST: x=q*y+r, 0≤r<y, and 0≤q
INVARIANT: x=q*y+r, 0≤r , and 0≤q

Example: Integer division. The process …

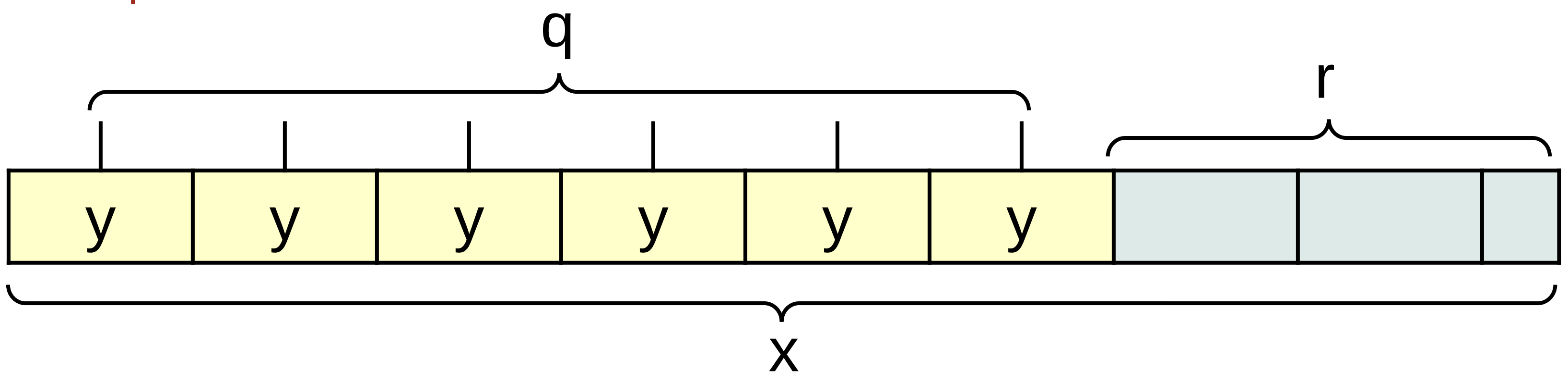
```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = ___
q: int = ___
while condition:
    _____________
```



PRE: $0 \le x$, $0 < y$
POST: $x = q*y + r$, $0 \le r < y$, and $0 \le q$
INVARIANT: $x = q*y + r$, $0 \le r$, and $0 \le q$

Example: Integer division. The process of maintaining the **INVARIANT** …

```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = ___
q: int = ___
while condition:
    r = r - y
    q += 1
```

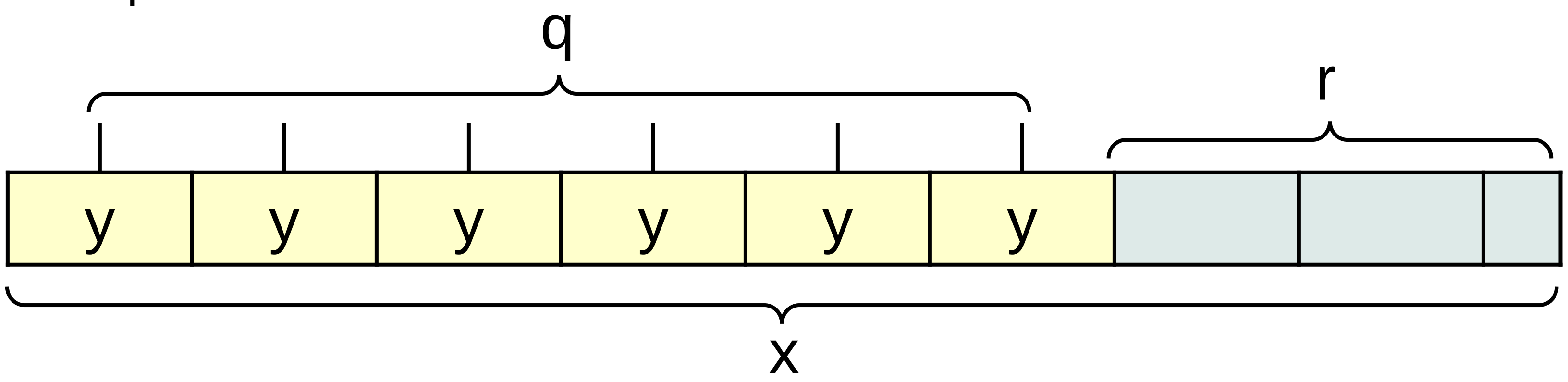


Integer Division

PRE: $0 \le x, 0 < y$
POST: $x = q*y + r, 0 \le r < y$, and $0 \le q$
INVARIANT: $x = q*y + r, 0 \le r$, and $0 \le q$
VARIANT: $x - q*y$

Example: Integer division. … while reducing the **VARIANT** to 0 …

```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = ___
q: int = ___
while r >= y:
    r = r - y
    q += 1
```



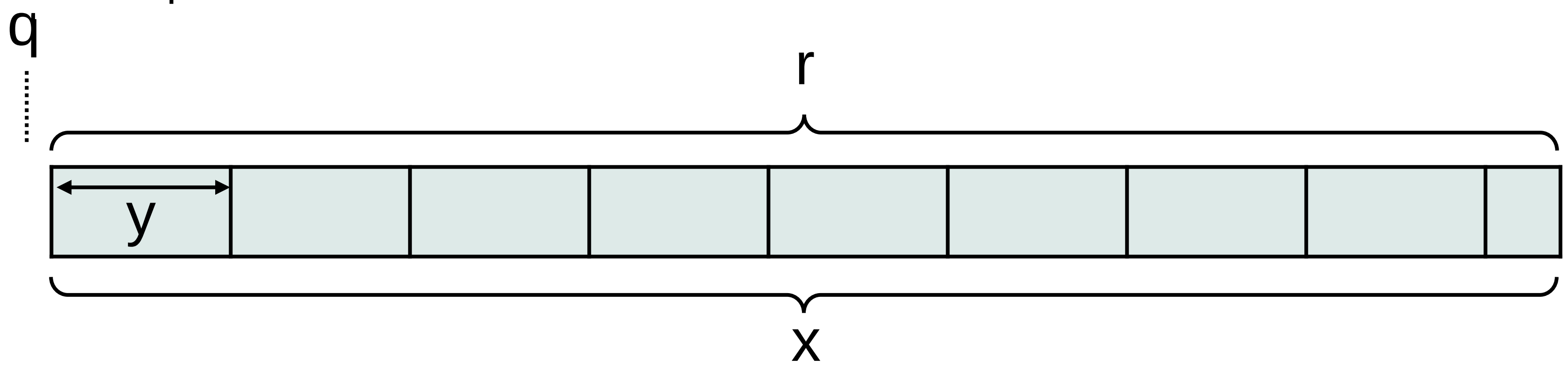
Iterative Refinement

Integer Division

PRE:	0≤x, 0<y
POST:	x=q*y+r, 0≤r<y, and 0≤q
INVARIANT:	x=q*y+r, 0≤r , and 0≤q
VARIANT:	x-q*y

Example: Integer division. … after first having established the **INVARIANT**.

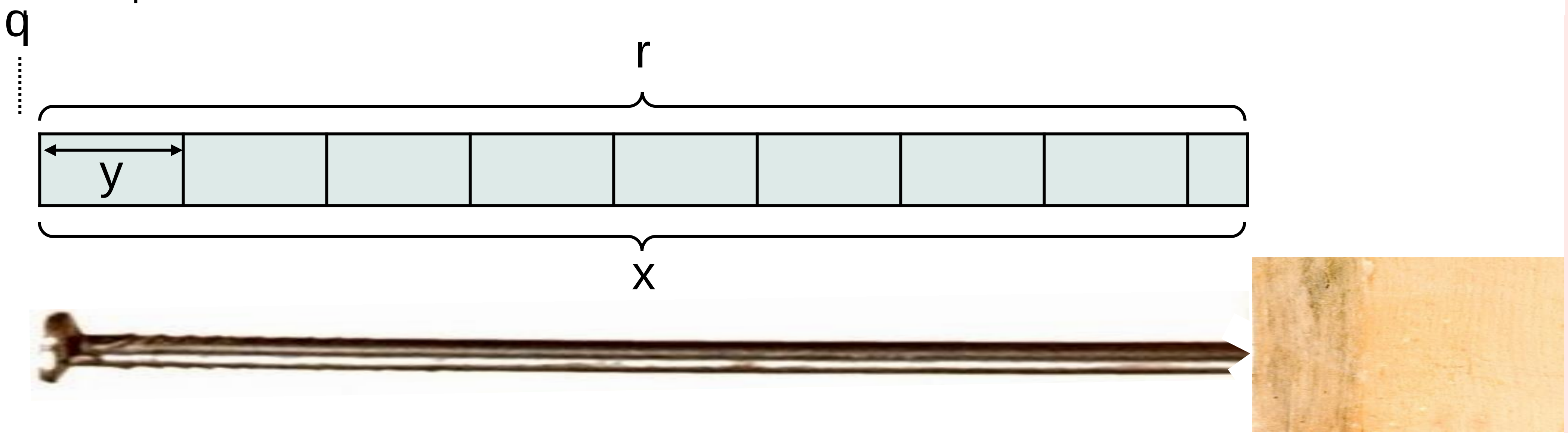
```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = x
q: int = 0
while r >= y:
    r = r - y
    q += 1
```



PRE:	0≤x, 0<y
POST:	x=q*y+r, 0≤r<y, and 0≤q
INVARIANT:	x=q*y+r, 0≤r , and 0≤q
VARIANT:	x-q*y

Example: Played in execution order, with nail and wood analogy: The setup.

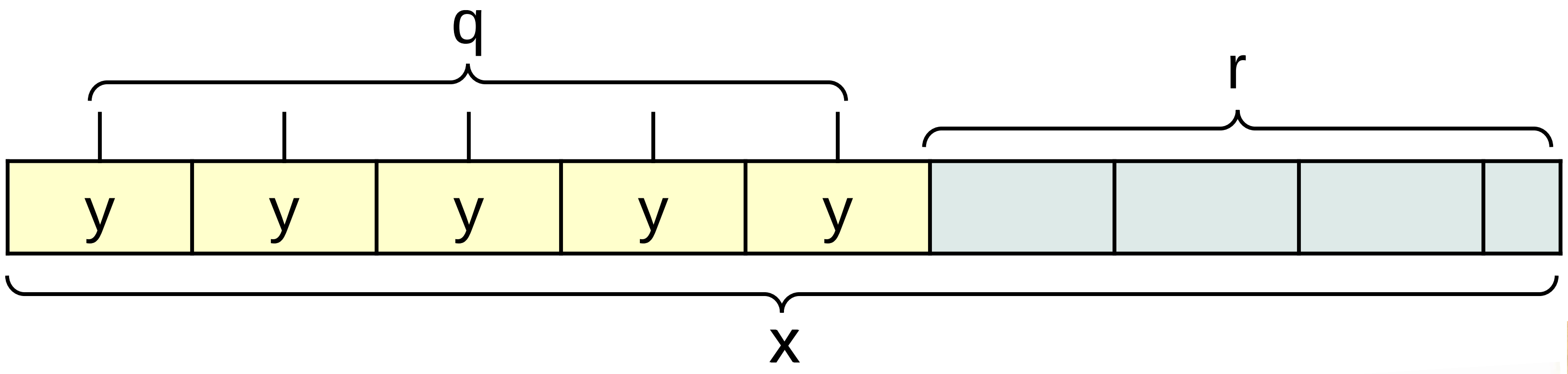
```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = x
q: int = 0
while r >= y:
    r = r - y
    q += 1
```



PRE:	0≤x, 0<y
POST:	x=q*y+r, 0≤r<y, and 0≤q
INVARIANT:	x=q*y+r, 0≤r , and 0≤q
VARIANT:	x-q*y

Example: Played in execution order, with nail and wood analogy: The process.

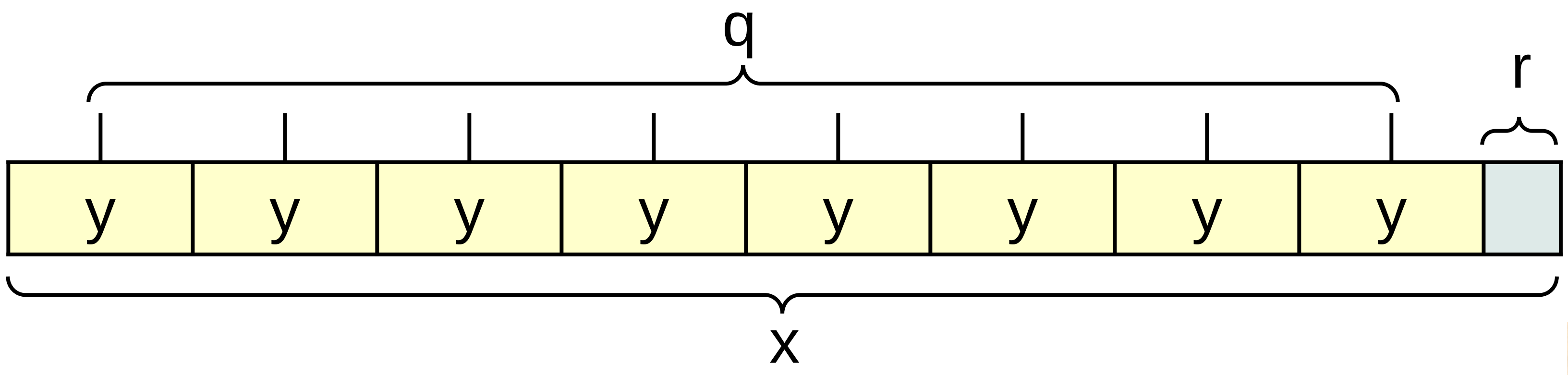
```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = x
q: int = 0
while r >= y:
    r = r - y
    q += 1
```



PRE:	0≤x, 0<y
POST:	x=q*y+r, 0≤r<y, and 0≤q
INVARIANT:	x=q*y+r, 0≤r , and 0≤q
VARIANT:	x-q*y

Example: Played in execution order, with nail and wood analogy: Termination.

```
# Given int x and int y, 0≤x and 0<y, set int q to x//y, and int r to x%y.
r: int = x
q: int = 0
while r >= y:
    r = r - y
    q += 1
```



Example: Euclid's Algorithm, the goal

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y.""“
    _________
    return ____
```

Let *X* and *Y* be positive integers, *GCD* be the mathematical greatest-common-divisor function, and *GCD*(*X*, *Y*) = *d*. When invoked on argument expressions with values *X* and *Y*, the Python function gcd is required to return *d*.

Example: Euclid's Algorithm, the **INVARIANT**

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while condition:
        _________
    return ____
```

We plan to update x or y on each iteration while maintaining the **INVARIANT**

$x > 0$ **and** $y > 0$ **and** $\mathit{GCD}(x, y) = d$

which holds initially because invocation of `gcd` with argument values *X* and *Y* initializes parameters x and y to *X* and *Y*, respectively.

Example: Euclid's Algorithm, the **VARIANT**

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        __________
    return x
```

To assure termination, we will reduce the non-negative **VARIANT** expression x+y by at least 1 on each iteration. You can only do that a finite number of times (while guaranteeing that x+y remains positive) before x+y stops changing. When it does, we must show that x=y. (It remains until after knowing how x and y are updated to show that x+y is reduced by 1 on each iteration.)

The **return** statement implements the observation that the *gcd* of any number and itself is that number.

Example: Euclid’s Algorithm, Case Analysis

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if _____:
          _________
        else:
          _________
    return x
```

There are only two cases: either x > y or x < y (the third possibility of x = y having been ruled out by the loop termination condition).

Example: Euclid's Algorithm, Case Analysis, x > y

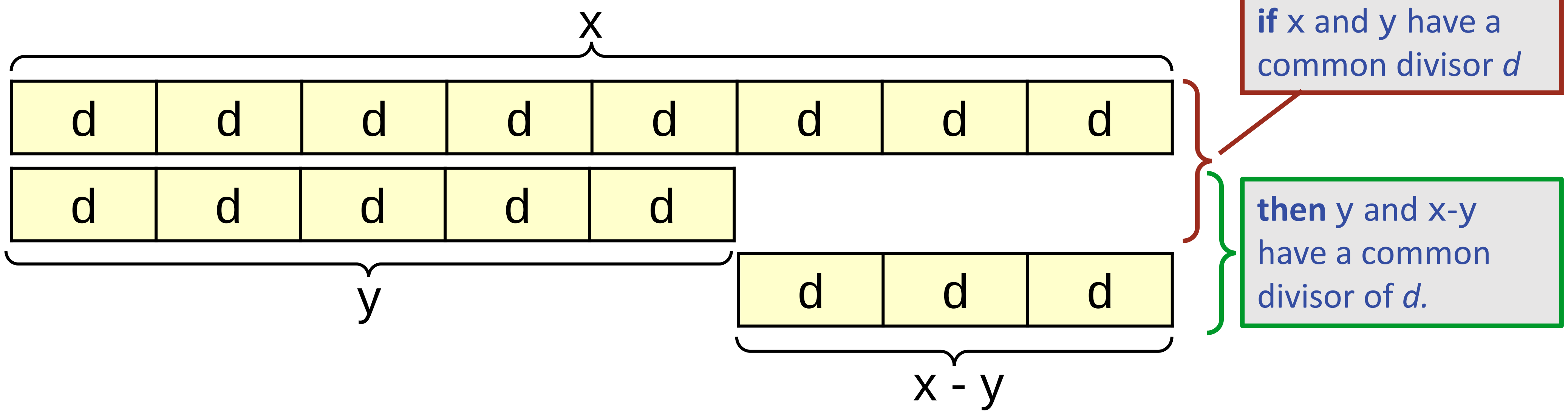
```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if x > y:
            __________
        else:
            __________
    return x
```

First case: x > y. Observe by diagrammatic reasoning (next slide), that x can be replaced by x - y while maintaining the **INVARIANT**, i.e., designating the new value of x by x', then

x' > 0 **and** y > 0 **and** *GCD*(x', y) = *d*

That x' > 0 follows from x > y.

Example: Euclid’s Algorithm, Case Analysis, x > y



if x and y have a common divisor *d*

then y and x-y have a common divisor of *d.*

Example: Euclid's Algorithm, Case Analysis, x > y

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if x > y:
            x = x - y
        else:
            _________
    return x
```

First case: x > y. Observe by diagrammatic reasoning (next slide), that x can be replaced by x - y while maintaining the **INVARIANT**, i.e., designating the new value of x by x', then

$x' > 0$ **and** $y > 0$ **and** $GCD(x', y) = d$

That x' > 0 follows from x > y. This has been done, above.

Example: Euclid's Algorithm, Case Analysis, x < y, by symmetry

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if x > y:
            x = x - y
        else:
            y = y - x
    return x
```

Second case: x < y. Follows by the symmetric argument.

Example: Euclid's Algorithm, correctness

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if x > y:
            x = x - y
        else:
            y = y - x
    return x
```

If the loop stops, the **INVARIANT** and negation of the condition imply that the value returned is correct, i.e.,

$x > 0$ **and** $y > 0$ **and** $GCD(x, y) = d$ **and** $x = y$ **implies** $x = d.$

This is called *partial correctness*.

Example: Euclid's Algorithm, termination

```
def gcd(x:int, y:int) -> int:
    """Given x>0 and y>0, return the greatest common divisor of x and y."""
    while x != y:
        if x > y:
            x = x - y
        else:
            y = y - x
    return x
```

Why does x+y stop changing, with x=y, and therefore the iteration stops?

Because one of x or y is reduced by at least 1 on each iteration, and if x is not equal to y, then in the next iteration x+y can be further reduced.

Partial correctness plus guaranteed termination is called *total correctness.*

Termination: Can be nontrivial, i.e., hard, unknown, or even unknowable

Termination: Are the following two code segments equivalent?

```
# Given input n>0, output “done”.
n = int(input())
print("done")

# Given input n>0, output “done”.
n = int(input())
while n != 1:
    if (n % 2) == 0:
        n = n // 2
    else:
        n = (3 * n) + 1
print("done")
```

Answer turns on whether the loop terminates for every input.

Termination: Are the following two code segments equivalent?

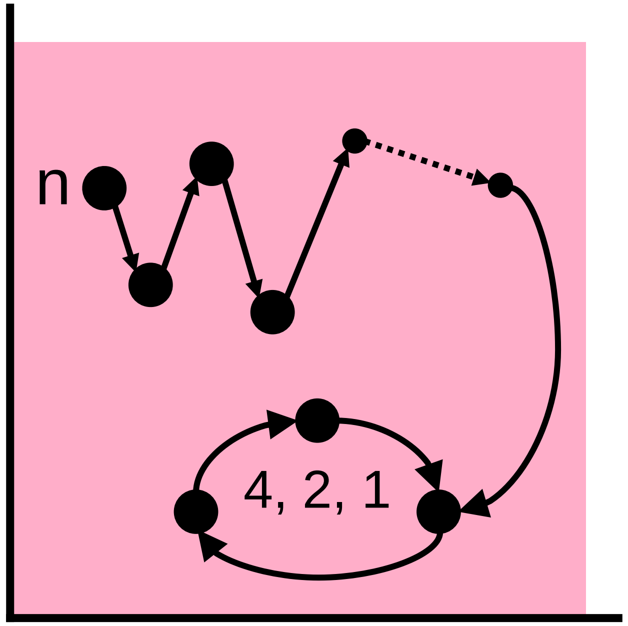
```
# Given input n>0, output “done”.
n = int(input())
print("done")

# Given input n>0, output “done”.
n = int(input())
while n != 1:
    if (n % 2) == 0:
        n = n // 2
    else:
        n = (3 * n) + 1
print("done")
```

Sample input 3:

3 ➔ 10 ➔ 5 ➔ 16 ➔ 8 ➔ 4 ➔ 2 ➔ 1

Termination: Are the following two code segments equivalent?

```
# Given input n>0, output “done”.
n = int(input())
print("done")

# Given input n>0, output “done”.
n = int(input())
while n != 1:
    if (n % 2) == 0:
        n = n // 2
    else:
        n = (3 * n) + 1
print("done")
```



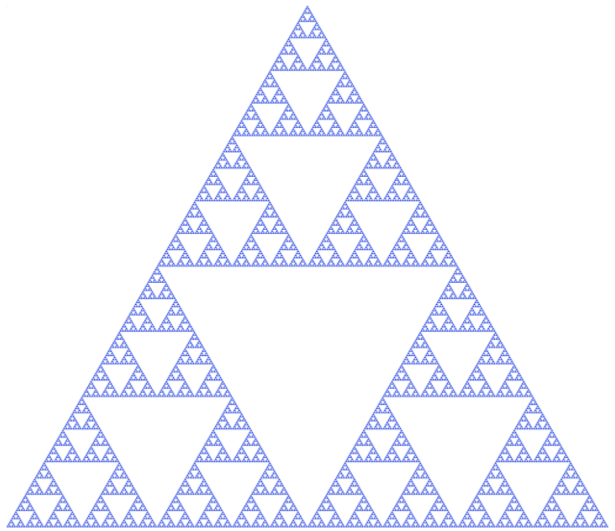
Collatz Conjecture

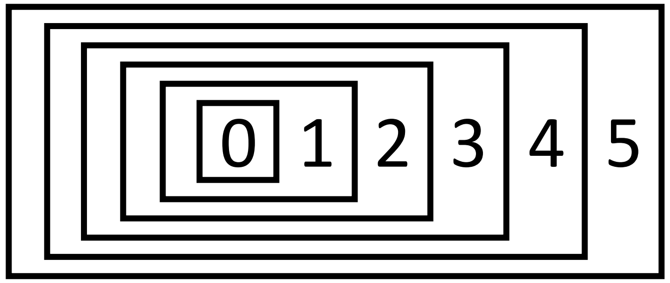
That every such sequence reaches 1 is an open problem in mathematics.

Recursive Refinement: Implement specification *P* by using the very refinement being defined to solve self-similar subproblems.

```
# Specification P.
if base-case :
    #.$P_0$.
else:
    #.Identify smaller instance(s) of P within P itself, apply this
    #   approach to each such instance, and combine the results.
```

Self-similarity: Same or similar structure at every scale





Recursive Refinement: Implement specification *P* by using the very refinement being defined to solve self-similar subproblems.

```
# Specification P.
if base-case :
    #.P_0.
else:
    #.Identify smaller instance(s) of P within P itself, apply this
    #   approach to each such instance, and combine the results.
```

To use the refinement within both the specification and in the refinement itself, define it separately as a procedure, and invoke it by name.

Recursive Refinement

Recursive Refinement: Implement specification *P* by using the very refinement being defined to solve self-similar subproblems.

```
# Specification P.
P(arguments)
```

and elsewhere define:

```
def P(parameters) -> type:
    if base-case :
        #.P_0.
    else:
        #.Identify smaller instance(s) of P within P itself, invoke
        #   P(arguments) on each such instance, and combine the results.
```

To use the refinement within both the specification and in the refinement itself, define it separately as a procedure, and invoke it by name.

Example: 5 4 3 2 1 BLASTOFF

```
# Count down from 5, and say “BLASTOFF” at 0.
countdown(5)
```

and elsewhere define:

```
def countdown(n: int) -> None:
    """Count down from n, and say "BLASTOFF" at zero."""
    if n == 0:
        print("BLASTOFF")
    else:
        print(n)
        countdown(n - 1)
```

A second example: (…(((0+1)+2)+3)+…+100)

```
# Output the sum of 1 through 100.
print(sum(100))
```

and elsewhere define:

```
def sum(n: int) -> int:
    """Return the sum of 0 through n."""
    if n == 0:
        return 0
    else:
        return sum(n - 1) + n
```

A third example: (1+(2+(3+…+(100+0)…)))

```
# Output the sum of 1 through 100.
print(sum(100))
```

and elsewhere define:

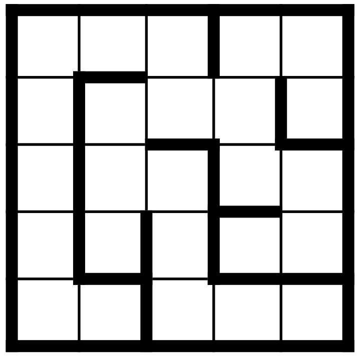
```
def sum(n:int) -> int:
    """Return the sum of 0 through n."""
    return sumAux(n,0)

def sumAux(n: int, acc: int) -> int:
    """Return the sum of 0 through n, and acc."""
    if n == 0:
        return acc
    else:
        return sumAux(n - 1, n + acc)
```

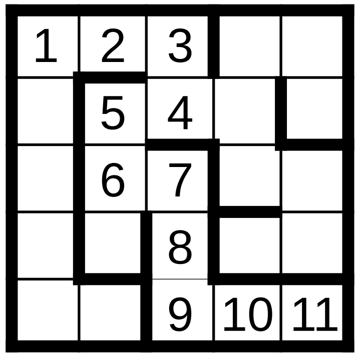
Library of Patterns: Implement specification *P* by using a previously used and analyzed parameterized composition of constructs.

Build your personal library over your lifetime.

Extended Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.
Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs "Unreachable" otherwise. A path is direct if it never visits any cell more than once.

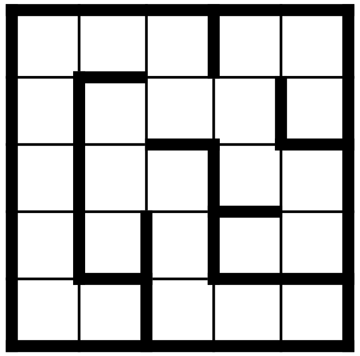


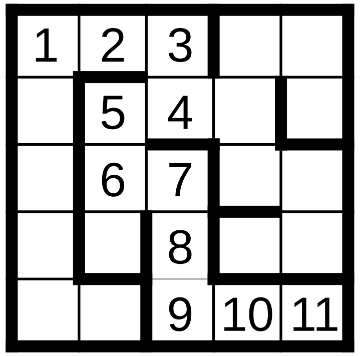


Extended Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.

Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs "Unreachable" otherwise. A path is direct if it never visits any cell more than once.





☞ **Use Stepwise Refinement. Write simple code immediately, otherwise refine the problem statement using: (a) Sequential Refinement, (b) Case Analysis, (c) Iterative Refinement, (d) a known pattern.**

Running a Maze

Specify the goal

```
#.Find path in maze from upper-left to lower-right, if one exists.
```

Refine with an architecture

```
# Find path in maze from upper-left to lower-right, if one exists.
# ------------------------------------------------------------------
#.Input.
#.Compute.
#.Output.
```

☞ **Master stylized code patterns, and use them.**

Refine with an architecture and elaborate

```
# Find path in maze from upper-left to lower-right, if one exists.
# ----------------------------------------------------------------
#.Input a maze of arbitrary size, or output “malformed input” and
#   stop if the input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none.
#   Output format: TBD.
```

☞ **Master stylized code patterns, and use them.**

Ignore Input and Output, and focus on essence

```
# Find path in maze from upper-left to lower-right, if one exists.
# ----------------------------------------------------------------
#.Input a maze of arbitrary size, or output “malformed input” and
#   stop if the input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none.
#   Output format: TBD.
```

Ignore Input and Output, and focus on essence

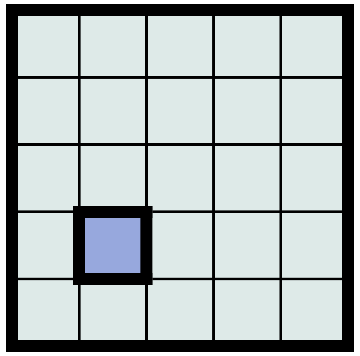
```
#.Compute a direct path through the maze, if one exists.
```

Investigate:

- ☞ **Analyze first.**
- ☞ **Confirm your understanding of a programming problem with concrete examples. Elaborate the expected input/output mapping explicitly.**
- ☞ **There is no shame in reasoning with concrete examples.**
- ☞ **Simple examples may be as good (or better) than complicated ones for guiding you toward a solution.**
- ☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Running a Maze

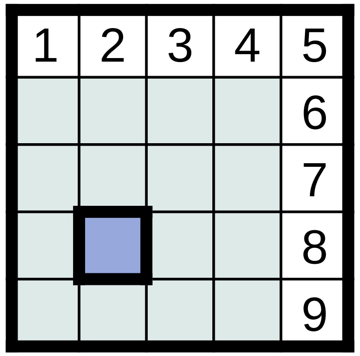
Investigate:


Example 1

Begin with a (near) empty maze

Running a Maze

Investigate:

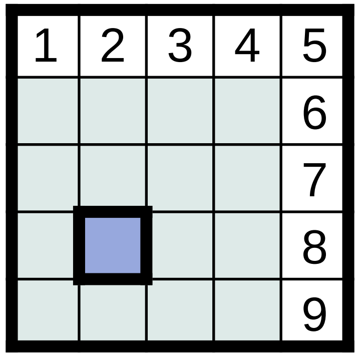


Example 1

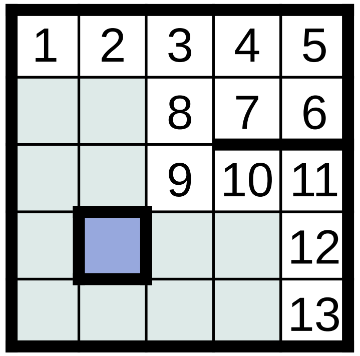
Traverse clockwise along the perimeter

Running a Maze

Investigate:



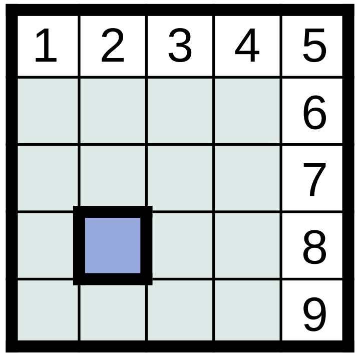
Example 1



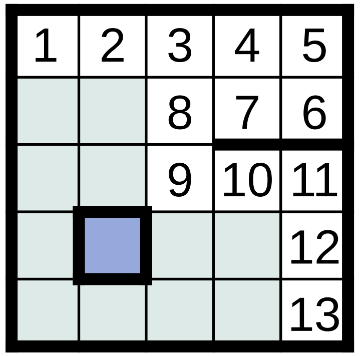
Example 2

Interpose a protruding wall. Continue excursion along it, pirouette to its other size, and continue.

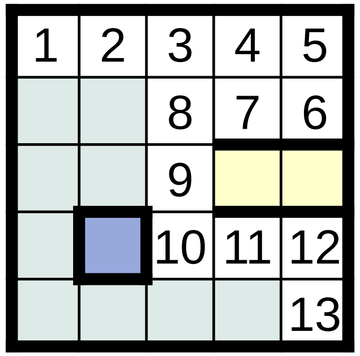
Investigate:



Example 1



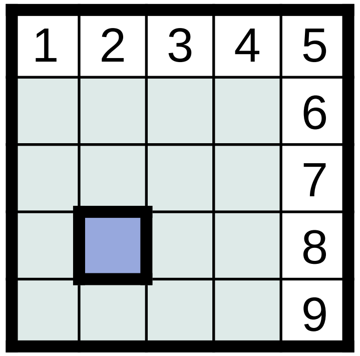
Example 2



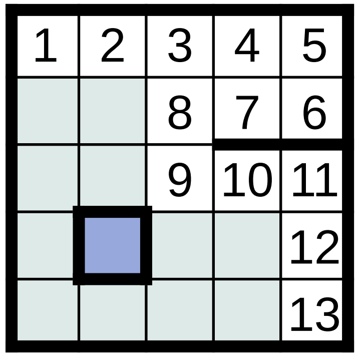
Example 3

Interpose a second protruding wall. Continue excursion along it (effectively backing out of a cul-de-sac), and continue.

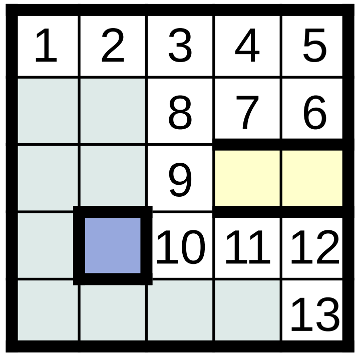
Investigate:



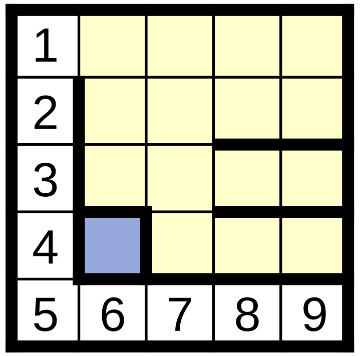
Example 1



Example 2



Example 3

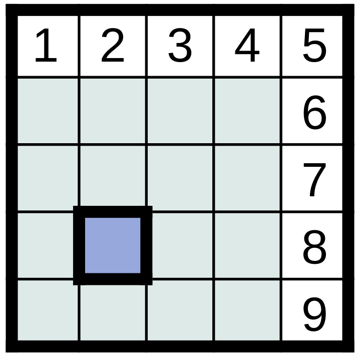


Example 4

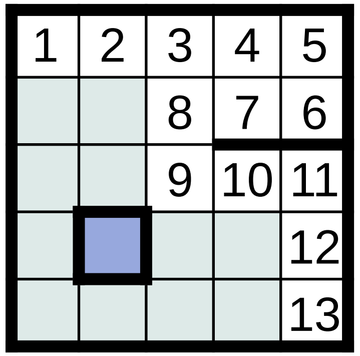
Interpose a third protruding wall. Continue excursion along it (effectively backing out of a room-sized cul-de-sac), and continue.

Running a Maze

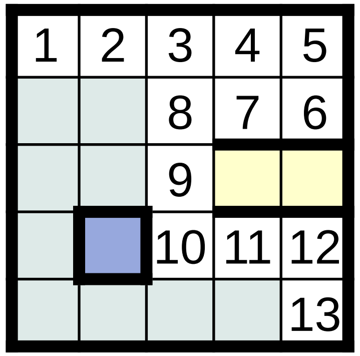
Investigate:



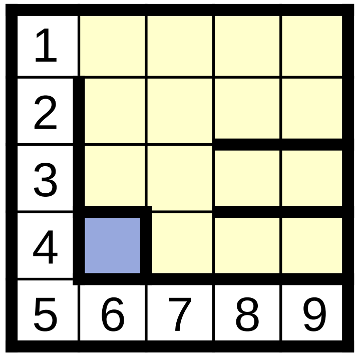
Example 1



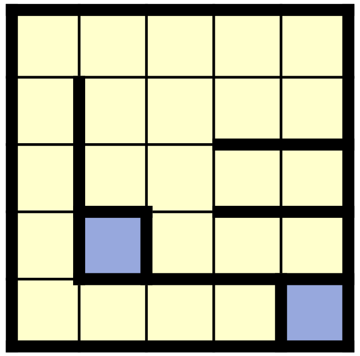
Example 2



Example 3



Example 4


Example 5

Block access to lower-right cell. Continue excursion along bottom and left perimeter, and then stop in upper-left cell.

Return to code

```
#.Compute a direct path through the maze, if one exists.
```

Return to code, and simplify

```
#.Compute a ~~direct~~ path through the maze, if one exists.
```

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Iterative Refinement:

```
# Compute a ~~direct~~ path through the maze, if one exists.
_________
while _________ :
    _________
_________
```

☞ **If you “smell a loop”, write it down.**

Running a Maze

Iterative Refinement:

```
# Compute a ~~direct~~ path through the maze, if one exists.
____3____
while ____2____ :
    ____1____
____4____
```

☞ **Code iterations in the following order: (1) body, (2) termination, (3) initialization, (4) finalization, (5) boundary conditions.**

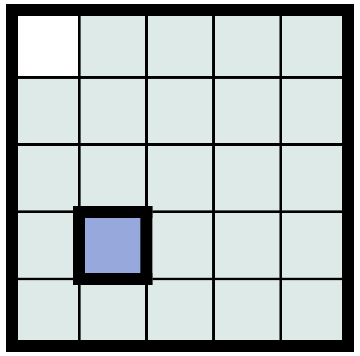
The loop body

```
# Compute a ~~direct~~ path through the maze, if one exists.
____3____
while ____2____ :
    ____1____
____4____
```

☞ **Body. Do 1st.**

Running a Maze

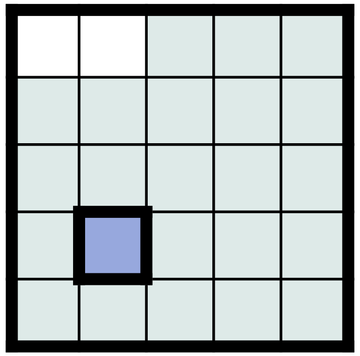
Pick an example, and imagine running the program for a while



Example 1

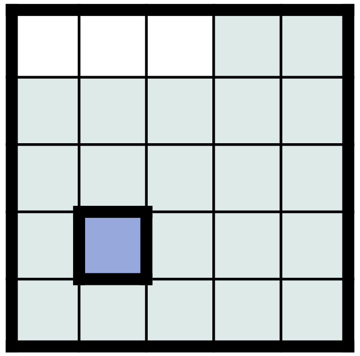
☞ **Body. Do 1st.** Play “musical chairs”

Pick an example, and imagine running the program for a while


Example 1

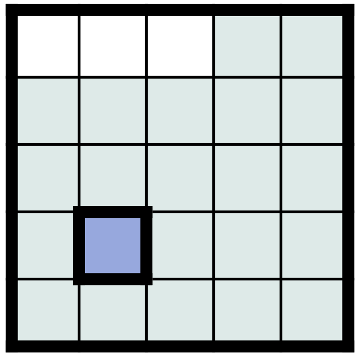
☞ **Body. Do 1st.** Play “musical chairs”

Pick an an example, and imagine running the program for a while


Example 1

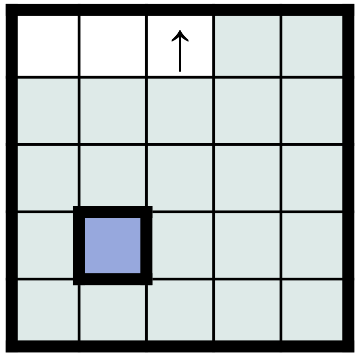
☞ **Body. Do 1st.** Play "musical chairs"

Stop at an arbitrary moment


Example 1

☞ **Body. Do 1st. Play "musical chairs" and "stop the music".**

Characterize the state

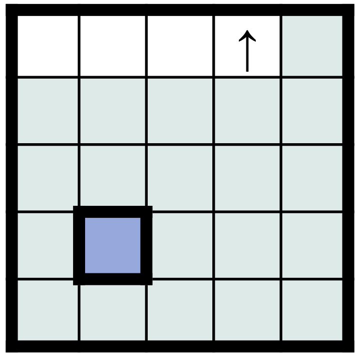


Example 1

Facing a wall

☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.**

Characterize the state, and the state transition

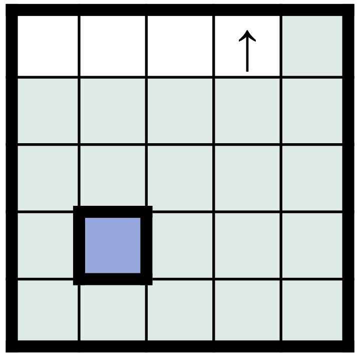


Example 1

INVARIANT: Hand on wall

☞ **Body. Do 1st. Play "musical chairs" and "stop the music". Characterize the "program state" when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the "loop invariant"), and what would have changed (the "loop variant")?**

Characterize the state, and the state transition



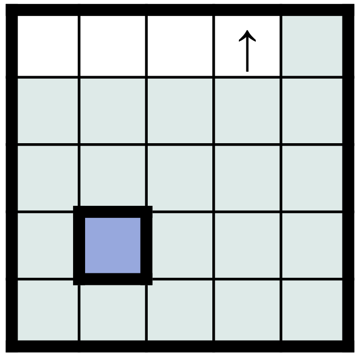
Example 1

INVARIANT: Hand on wall

VARIANT: Distance to goal

☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the “loop invariant”), and what would have changed (the “loop variant”)?**

Characterize the state, and the state transition



Example 1

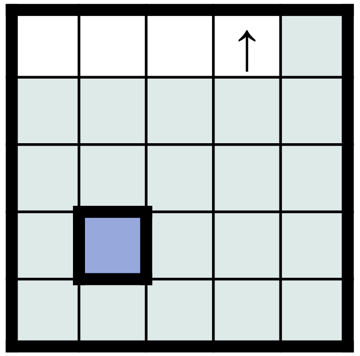
INVARIANT: Hand on wall

VARIANT: Distance to goal

☞ A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.

Running a Maze

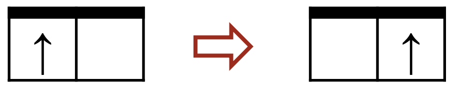
Characterize the state, and the state transition


Example 1

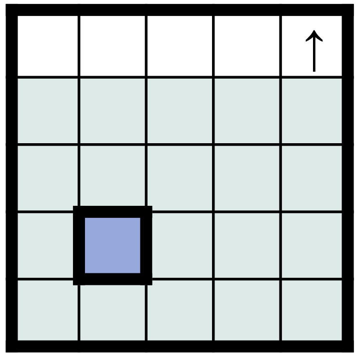
INVARIANT: Hand on wall

VARIANT: Distance to goal

Transition rule

(1) 

Resume playing musical chairs, applying the transition rule

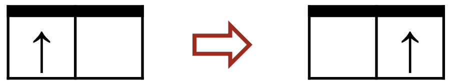


Example 1

INVARIANT: Hand on wall

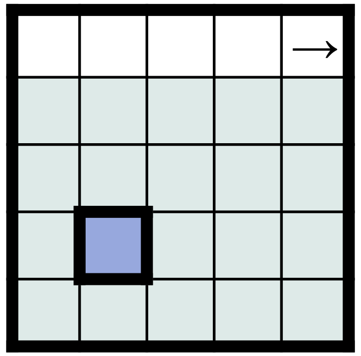
VARIANT: Distance to goal

Transition rule

(1) 

Running a Maze

Introduce a new transition rule when needed

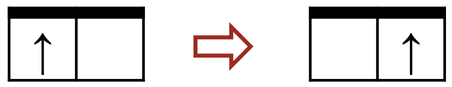


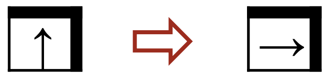
Example 1

INVARIANT: Hand on wall

VARIANT: Distance to goal

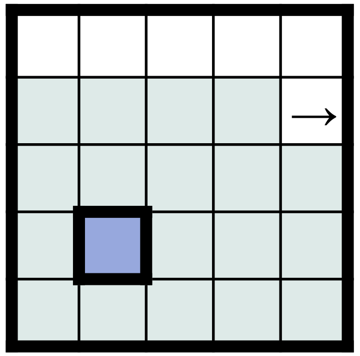
Transition rules

(1) 

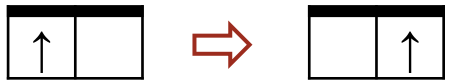
(2) 

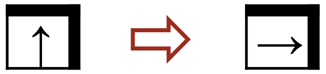
Running a Maze

and resume playing musical chairs

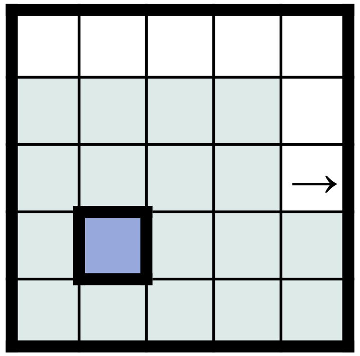

Example 1

Transition rules

(1) 

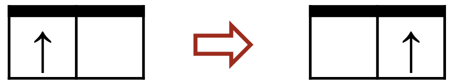
(2) 

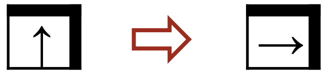
and resume playing musical chairs



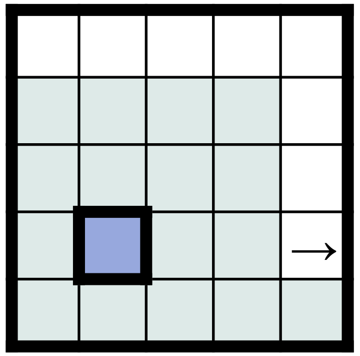
Example 1

Transition rules

(1) 

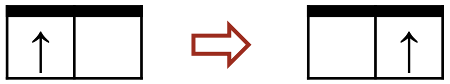
(2) 

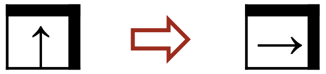
and resume playing musical chairs



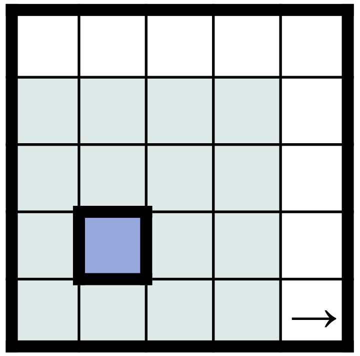
Example 1

Transition rules

(1) 

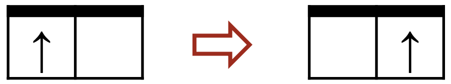
(2) 

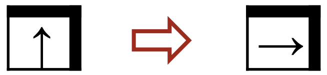
and resume playing musical chairs



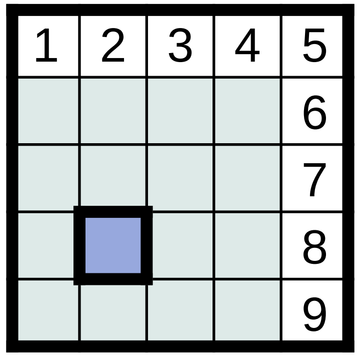
Example 1

Transition rules

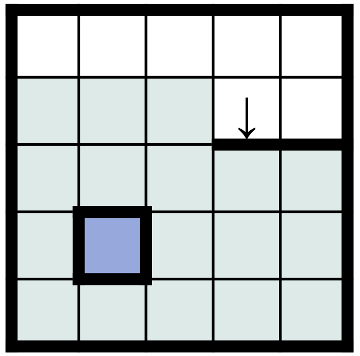
(1) 

(2) 

Try another example

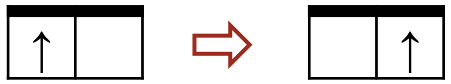


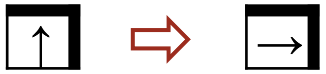
Example 1



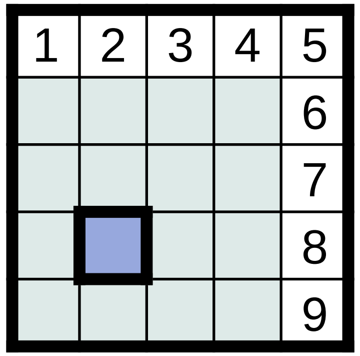
Example 2

Transition rules

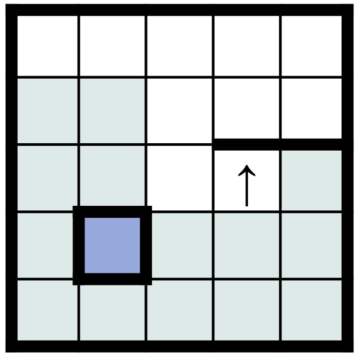
(1) 

(2) 

Try another example, and introduce a new transition rule when needed



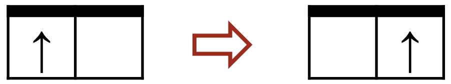
Example 1



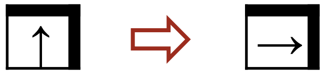
Example 2

Transition rules

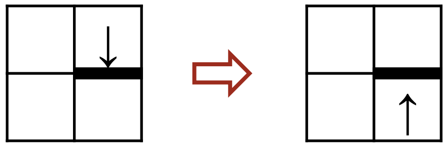
(1)



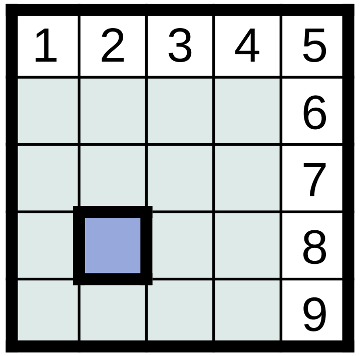
(2)



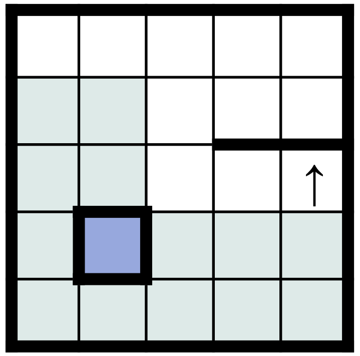
(3)



Then resume playing musical chairs

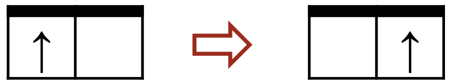


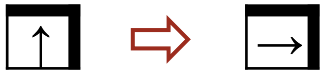
Example 1

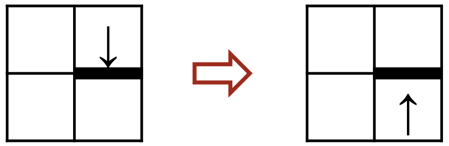


Example 2

Transition rules

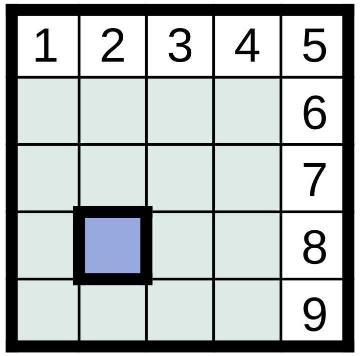
(1) 

(2) 

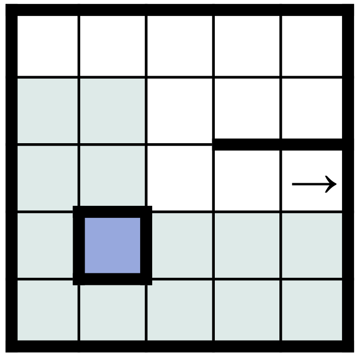
(3) 

Running a Maze

Then resume playing musical chairs

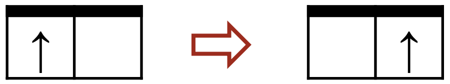


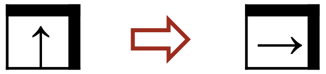
Example 1

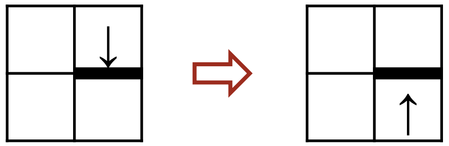


Example 2

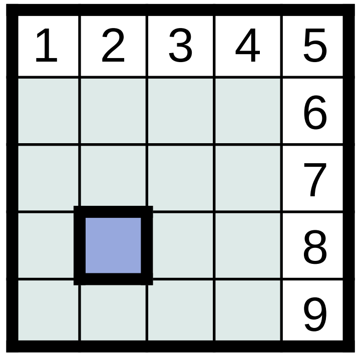
Transition rules

(1) 

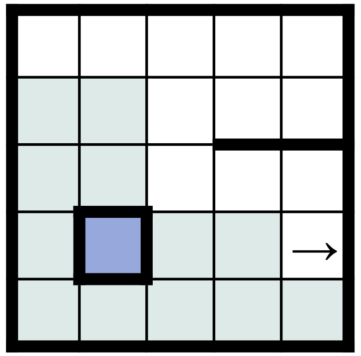
(2) 

(3) 

Then resume playing musical chairs

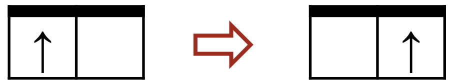


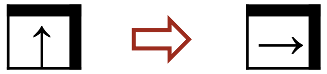
Example 1

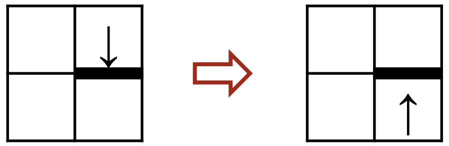


Example 2

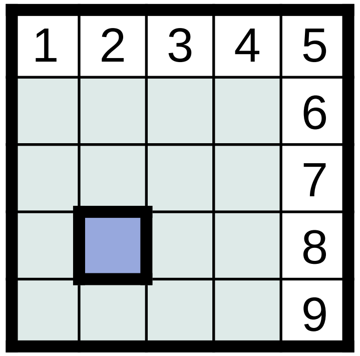
Transition rules

(1) 

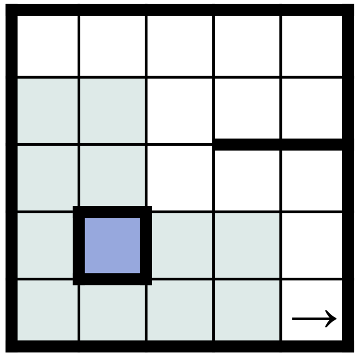
(2) 

(3) 

Then resume playing musical chairs

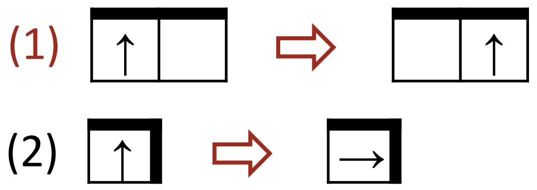


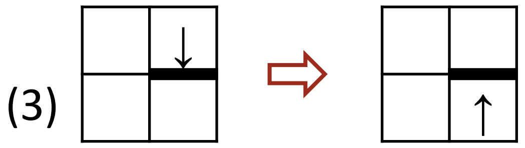
Example 1



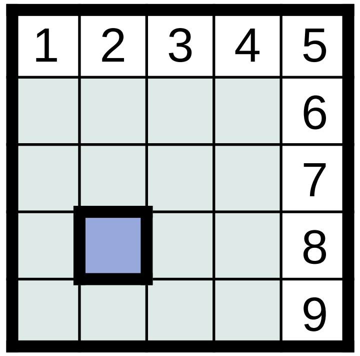
Example 2

Transition rules

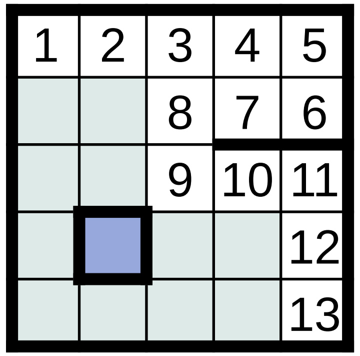




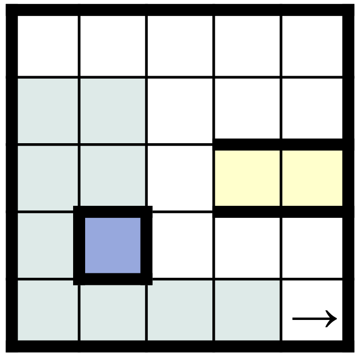
Try another example, and see that the three transition rules suffice



Example 1

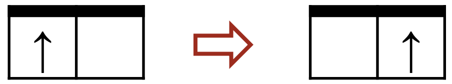


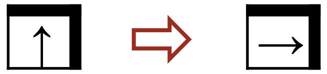
Example 2

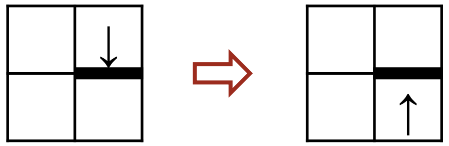


Example 3

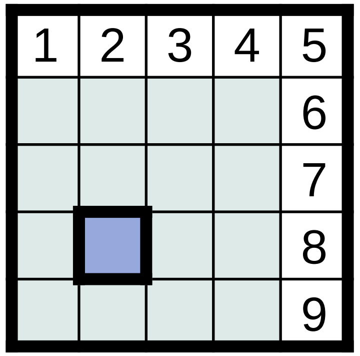
Transition rules

(1) 

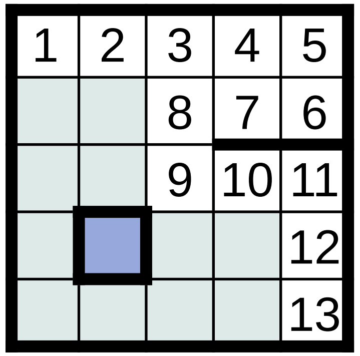
(2) 

(3) 

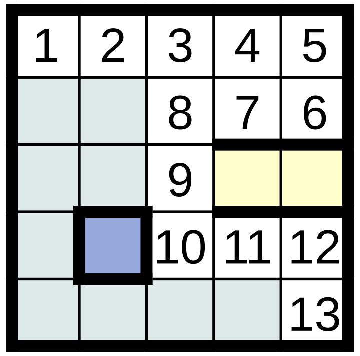
Try another example, and see that the three transition rules get you far



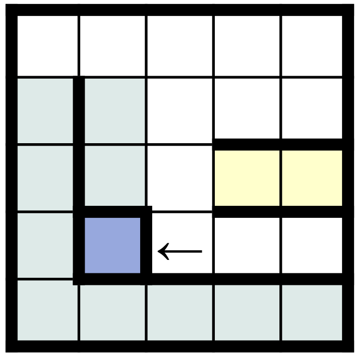
Example 1



Example 2



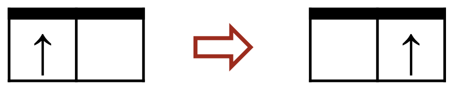
Example 3

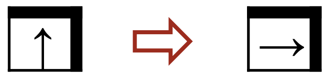


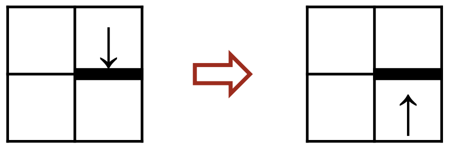
Example 4

Numbering reflects the direct path

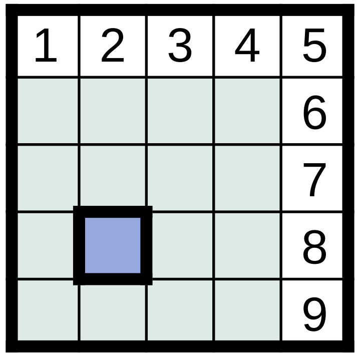
Transition rules

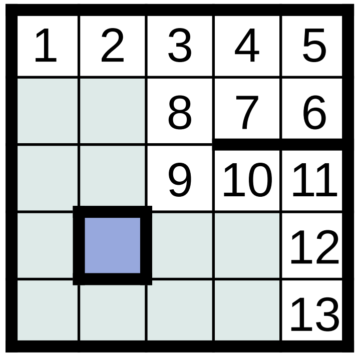
(1)


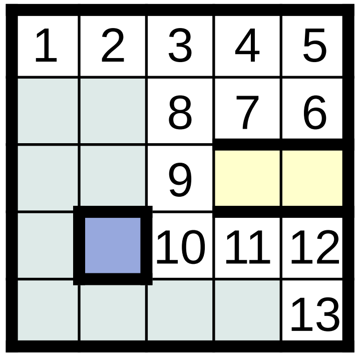
(2)


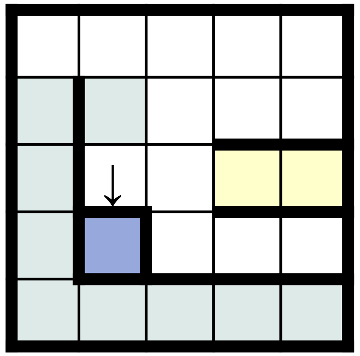
(3)


until a fourth rule is needed

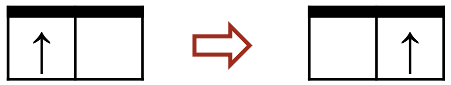

Example 1

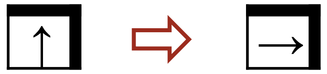

Example 2

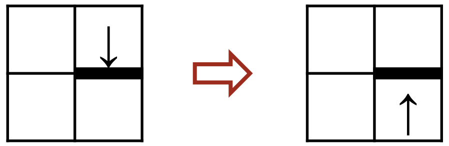

Example 3

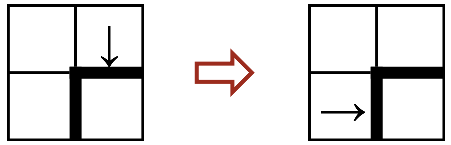

Example 4

Transition rules

(1) 

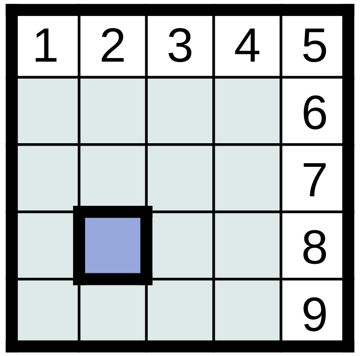
(2) 

(3) 

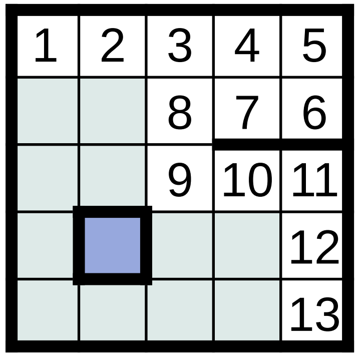
(4) 

Running a Maze

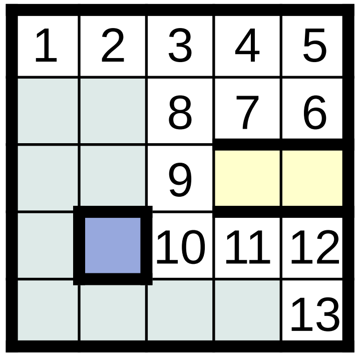
Resume



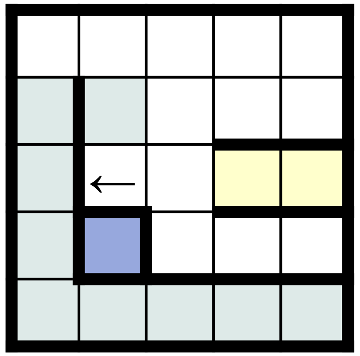
Example 1



Example 2

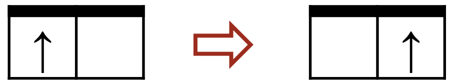


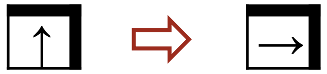
Example 3

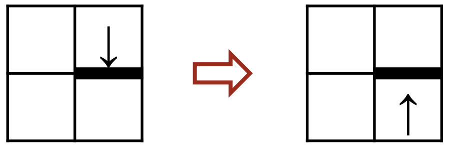


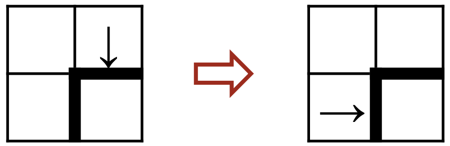
Example 4

Transition rules

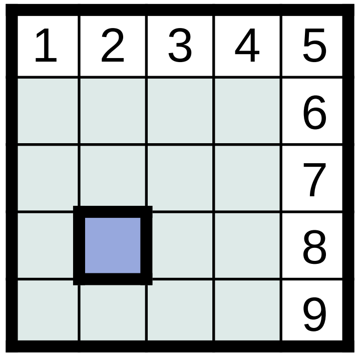
(1) 

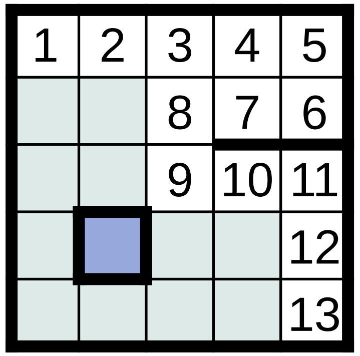
(2) 

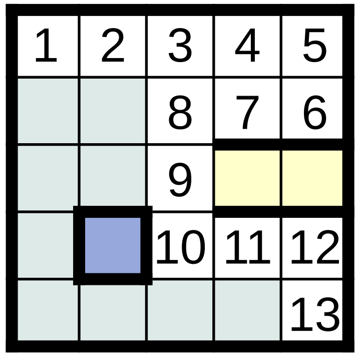
(3) 

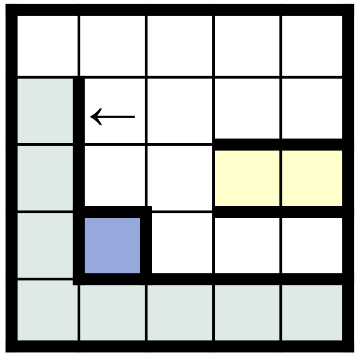
(4) 

Resume

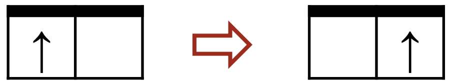

Example 1

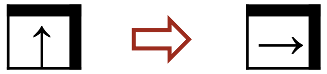

Example 2

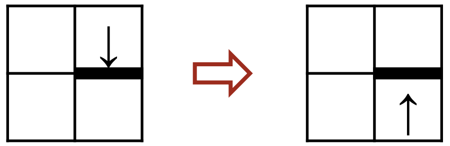

Example 3

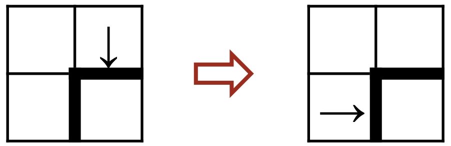

Example 4

Transition rules

(1)


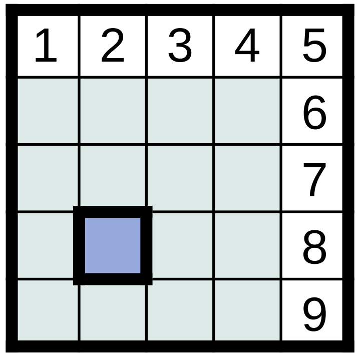
(2)


(3)


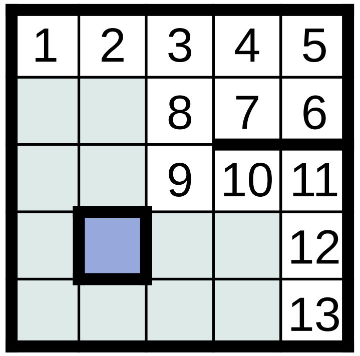
(4)


Running a Maze

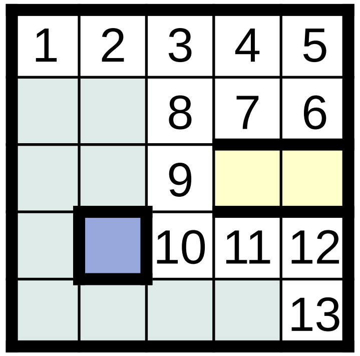
Resume



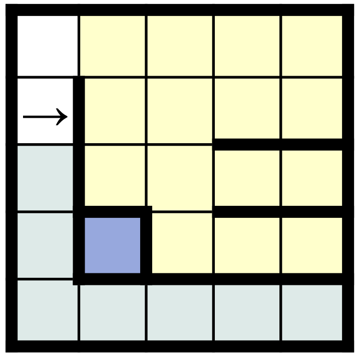
Example 1



Example 2

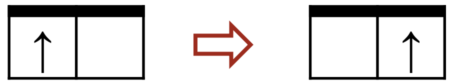


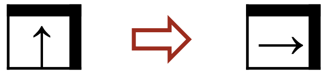
Example 3

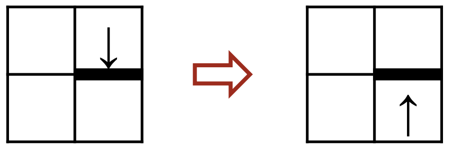


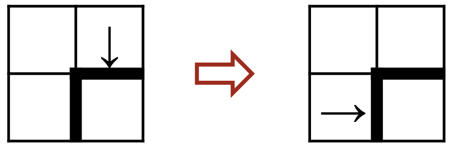
Example 4

Transition rules

(1) 

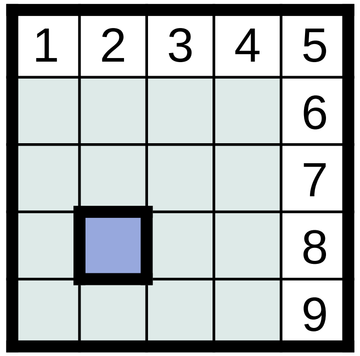
(2) 

(3) 

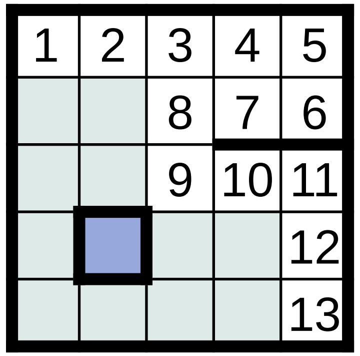
(4) 

Running a Maze

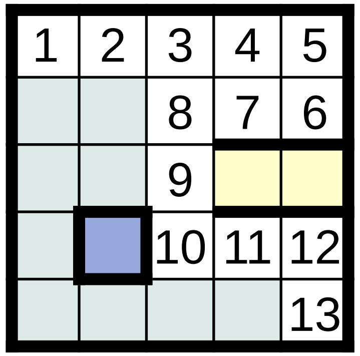
Resume, and go all the way



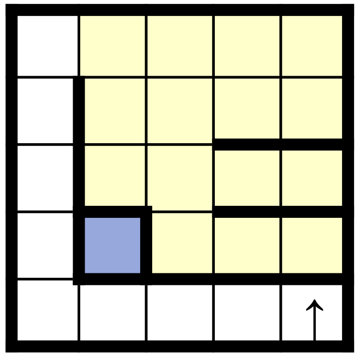
Example 1



Example 2

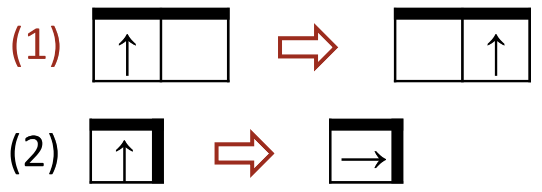


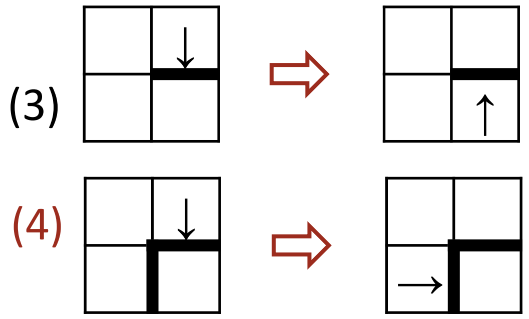
Example 3



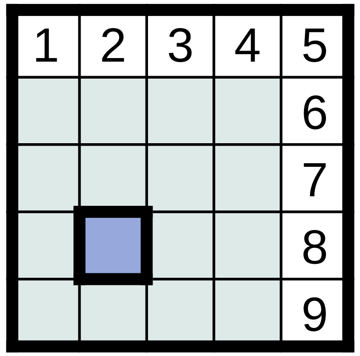
Example 4

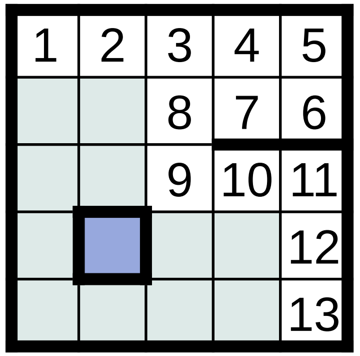
Transition rules

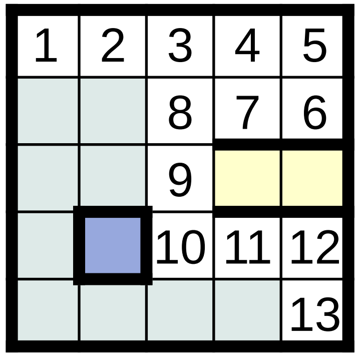


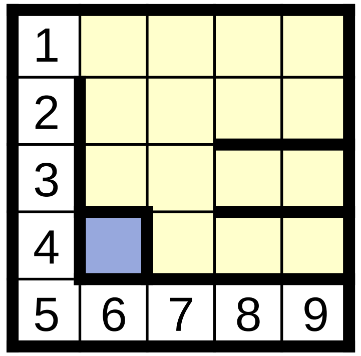


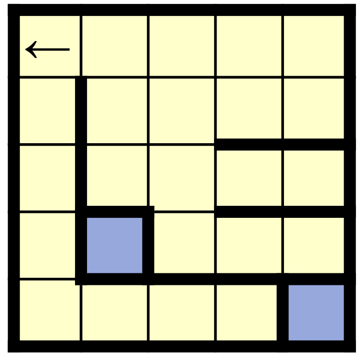
And yet another example


Example 1


Example 2

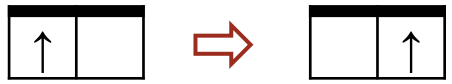

Example 3

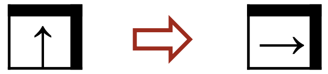

Example 4

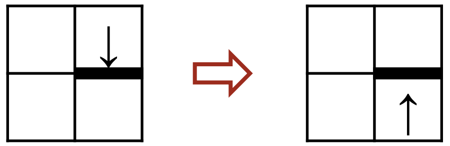

Example 5

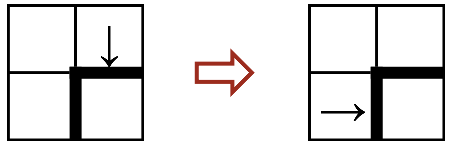
Numbering reflects the direct path

Transition rules

(1)


(2)


(3)


(4)


The loop body: One case for each transition rule

```
# Compute a ~~direct~~ path through the maze, if one exists.
__________
while __________ :
    if __________ :
        __________
    elif __________ :
        __________
    elif __________ :
        __________
    else:
        __________
__________
```

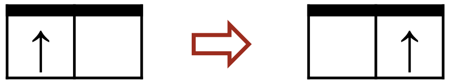
Running a Maze

The loop body: One case for each transition rule, but they are too complex.

```
# Compute a ~~direct~~ path through the maze, if one exists.
_________
while _________ :
    if _________ :
        _________
    elif _________ :
        _________
    elif _________ :
        _________
    else:
        _________
_________
```

Running a Maze

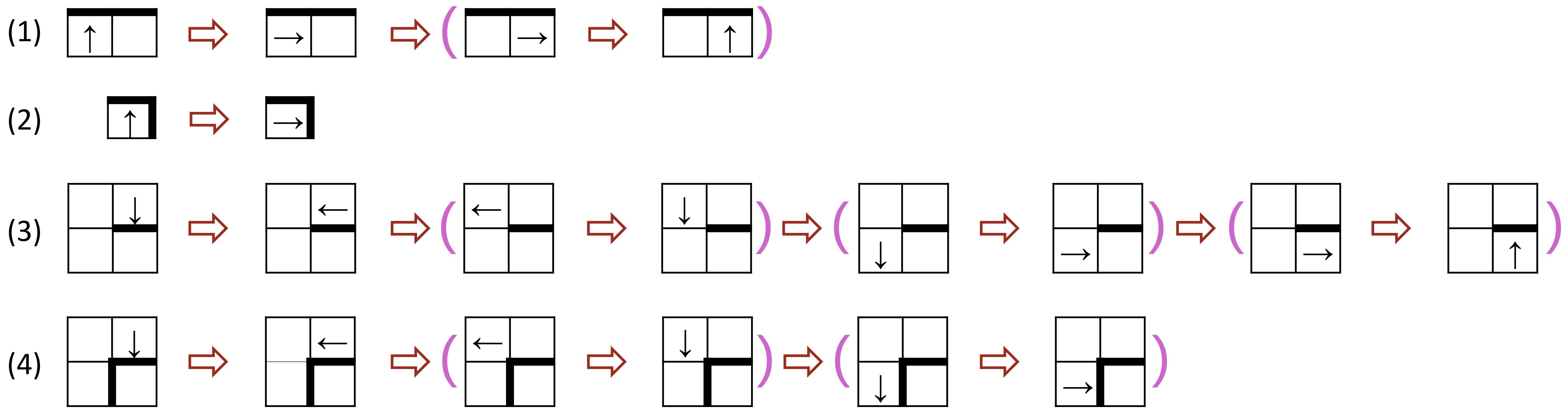
The loop body: One case for each transition rule, but they are too complex.

For example: (1) 

```
# Compute a ~~direct~~ path through the maze, if one exists.
__________
while __________ :
    if two-colinear-walls-not-separated-by-a-perpendicular-wall:
        #.sidestep.
    elif __________ :
        __________
    elif __________ :
        __________
    else:
        __________
__________
```

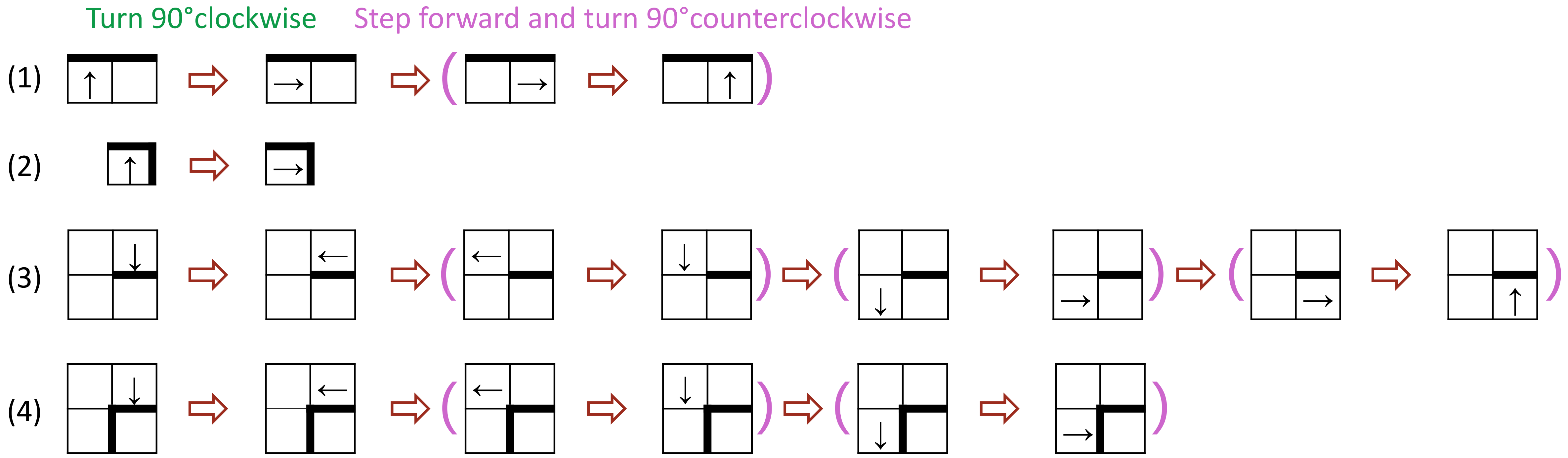
Running a Maze

Idea: Implement coarse-grain transition steps with micro-operations



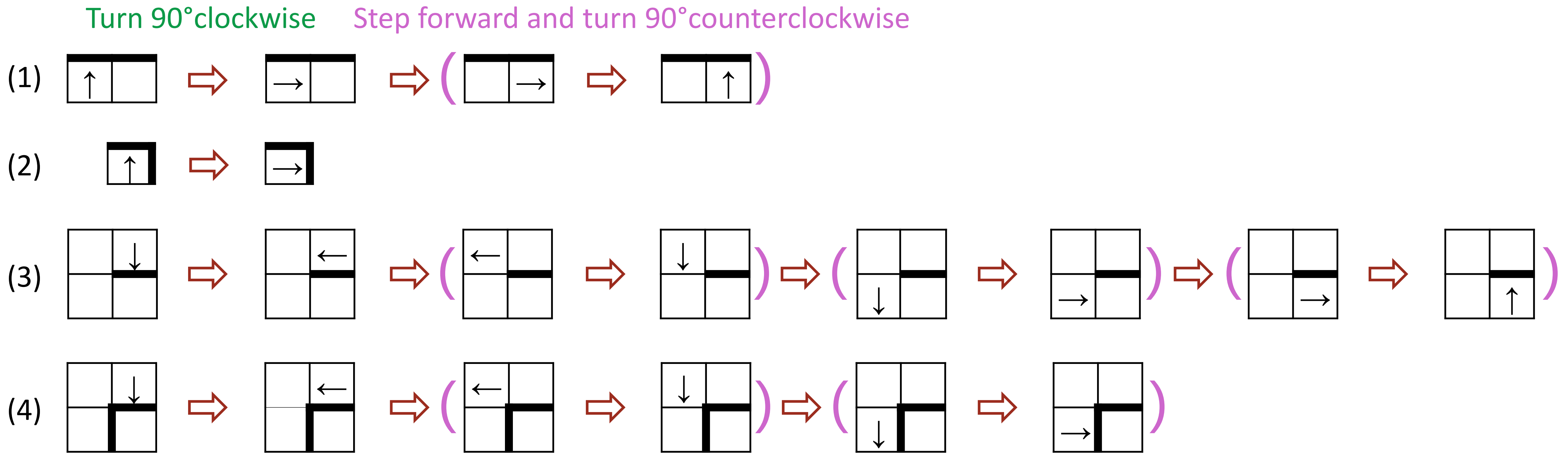
Running a Maze

Idea: Implement coarse-grain transition steps with micro-operations



Running a Maze

Idea: Implement coarse-grain transition steps with micro-operations



new **INVARIANT**: Hand on wall or door

new **VARIANT**: Number of wall segments or doors to goal

The loop body: Now only two simpler cases to consider.

```
# Compute a ~~direct~~ path through the maze, if one exists.
_________
while _________ :
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
_________
```

new **INVARIANT**: Hand on wall or door
new **VARIANT**: Number of wall segments or doors to goal

Iteration: (2) termination

```
# Compute a ~~direct~~ path through the maze, if one exists.
__________
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
__________
```

Iteration: (3) initialization

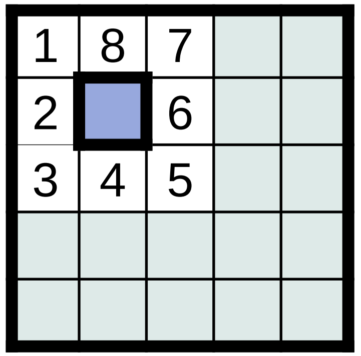
```
# Compute a ~~direct~~ path through the maze, if one exists.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Start in upper-left cell, facing up.
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```

____

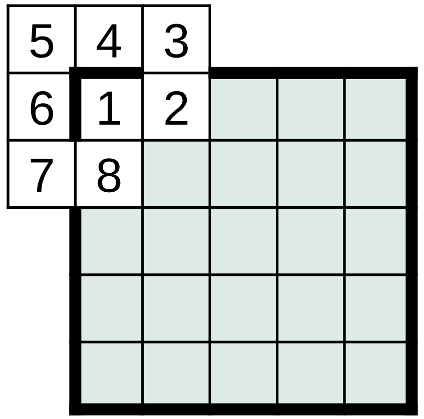
Iteration: Correctness relies on subtle problem constraints

```
# Compute a ~~direct~~ path through the maze, if one exists.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Start in upper-left cell, facing up.
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```

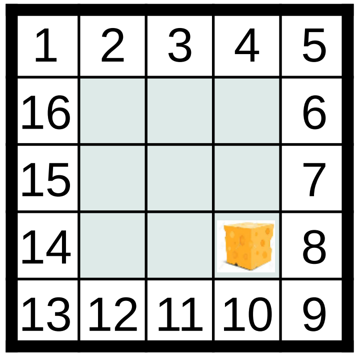
Iteration: Correctness relies on subtle problem constraints



If started facing down, not up



If outer wall not solid



If cheese could be in interior cell

Iteration: (4) finalization (nothing to do)

```
# Compute a ~~direct~~ path through the maze, if one exists.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Start in upper-left cell, facing up.
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```

The core algorithm is in hand

```
# Compute a ~~direct~~ path through the maze, if one exists.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#.Start in upper-left cell, facing up.
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall:
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```