Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

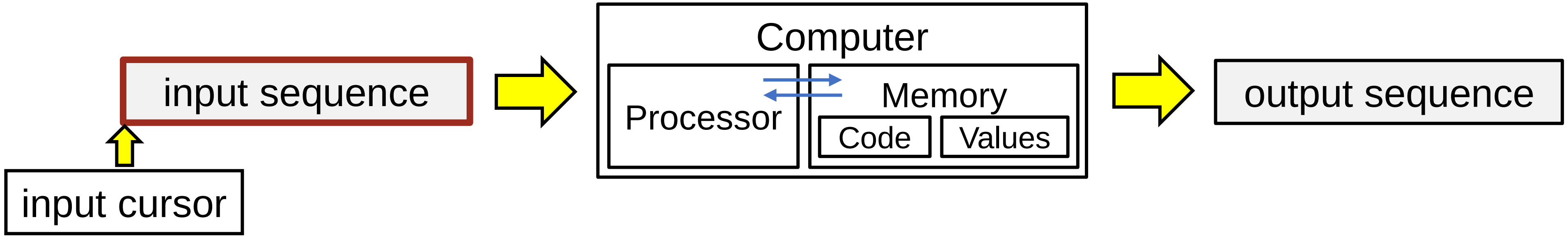
Online Algorithms

We introduce the *online-computation* pattern for processing an unbounded file of input. We use it to:

- Process exam grades
- Compress the file
- Decompress a compressed file

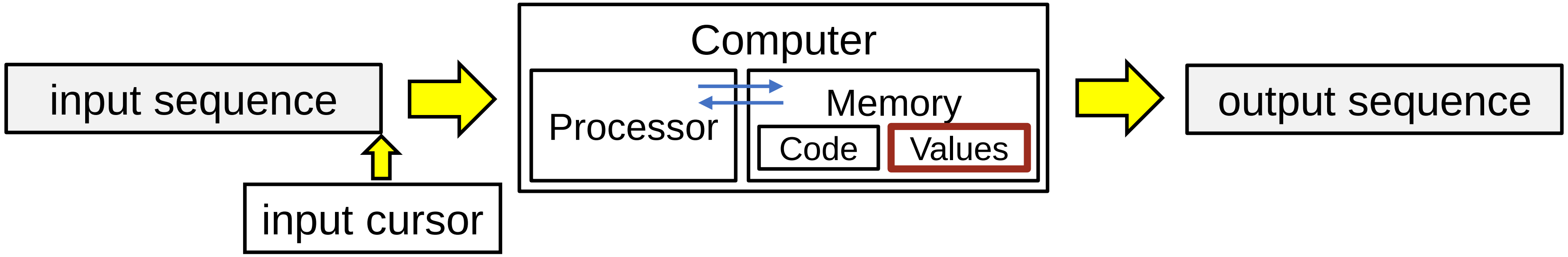
We illustrate many important programming precepts.

Application: Process an input file of unbounded length.



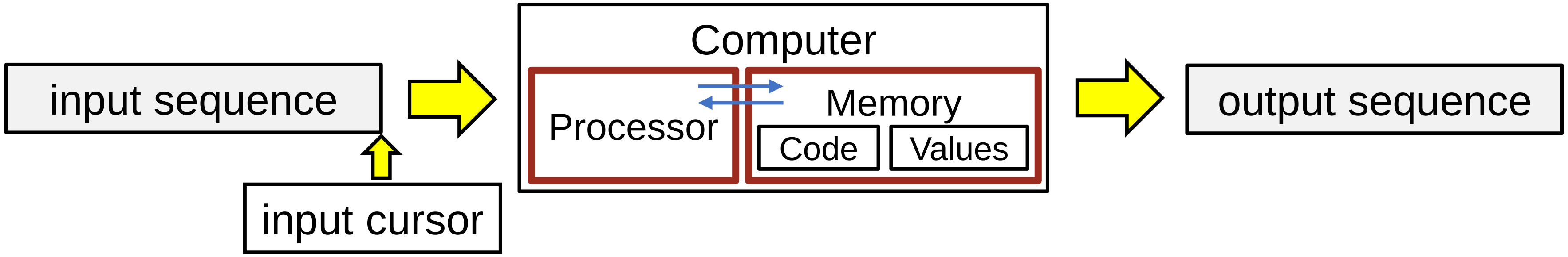
Offline-computation pattern: calls for reading all values first.

```
#.Input.
#.Compute.
#.Output.
```



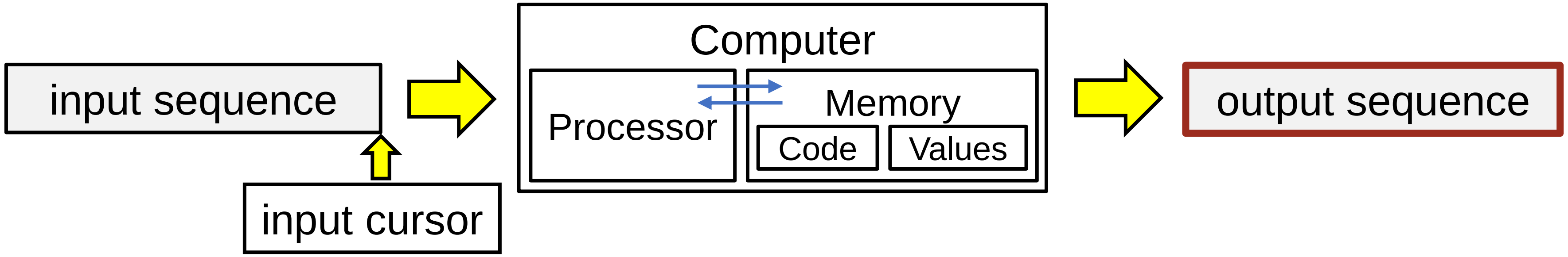
Offline-computation pattern: calls for reading all values first, then processing them.

```
#.Input.
#.Compute.
#.Output.
```



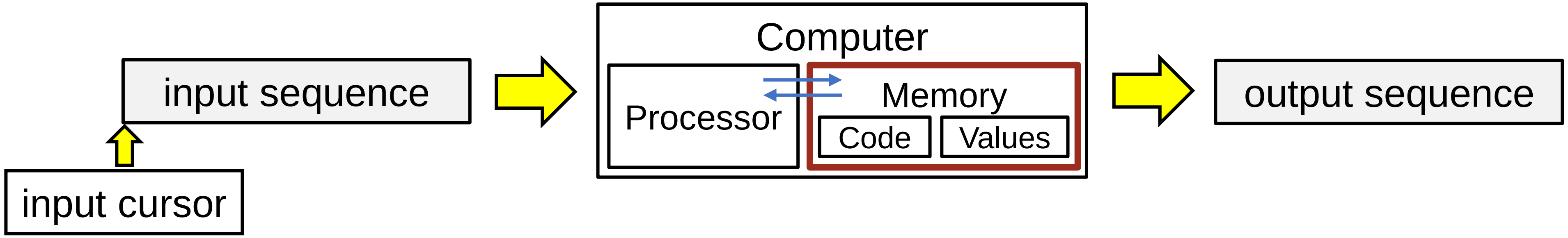
Offline-computation pattern: calls for reading all values first, then processing them, then outputting results.

```
#.Input.
#.Compute.
#.Output.
```



Offline-computation pattern: A mismatch because the memory is finite, but the input is unbounded.

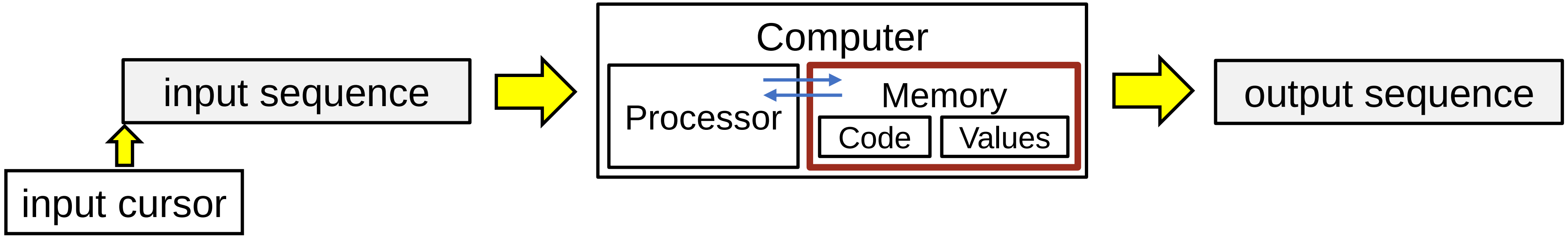
```
#.Input.
#.Compute.
#.Output.
```



Offline-computation pattern: A mismatch because the memory is finite, but the input is unbounded.*

```
#.Input.
#.Compute.
#.Output.
```

*Virtual memory is also effectively unbounded, so the real issue is paging time.



Online-computation pattern: An alternative is to process input values on the fly.

```
v = first-input-value
#.Initialize.
while v != stoppingValue:
    #.Process v.
    v = next-input-value
#.Finalize.
```

Online-computation pattern: A specialization of the general-iteration pattern.

```
v = first-input-value
#.Initialize.
while v != stoppingValue:
    #.Process v.
    v = next-input-value
#.Finalize.
```

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go on to next.
```

Online-computation pattern: Not all problems amenable to online computation.

```
v = first-input-value
#.Initialize.
while v != stoppingValue:
    #.Process v.
    v = next-input-value
#.Finalize.
```

Amenable if:

- Inputs are independent and can be fully processed on the fly.

Online-computation pattern: Not all problems amenable to online computation.

```
v = first-input-value
#.Initialize.
while v != stoppingValue:
    #.Process v.
    v = next-input-value
#.Finalize.
```

Amenable if:

- Inputs are independent and can be fully processed on the fly, or
- Inputs can be summarized on the fly, and the final result computed from those summary values.

Online-computation pattern: Assume inputs are nonnegative integers, followed by -1, each on a separate line.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
#.Initialize.
while v != -1:
    #.Process v.
    v = int(input())
#.Finalize.
```

Online-computation pattern: Parametric in α, β, and γ.

```
v = int(input())        # v is the next integer to be processed, or -1.
#.Initialize. (α)
while v != -1:
    #.Process v. (β)
    v = int(input())
#.Finalize. (γ)
```

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

Data Processing

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

```
90    80    85    90    100    0    85    -1
```

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **There is no shame in reasoning with concrete examples.**

N.B. Recall that the inputs are assumed to each occur on separate lines despite this picture showing them all on the same line.

```
90    80    85    90    100    0    85    -1
```

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **There is no shame in reasoning with concrete examples.**

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

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

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

Application: Process exam grades (in range 0-100).

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Print grades.

Application: Print grades.

☞ **Program top-down, outside-in.**

Application: Print grades.

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

Application: Print grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

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

Application: Print grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Print grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    print(grade)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Print grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
while grade != -1:
    print(grade)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Print grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
while grade != -1:
    print(grade)
    grade = int(input())
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

```
90    80    85    90    100    0    85    -1
```

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Counting the input values.

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = _________       # count is the number of grades processed so far.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Introduce program variables whose values describe “state”.**

A counter count *value* . Establish and maintain its representation invariant.

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = _________       # count is the number of grades processed so far.
#.Initialize. (α)
while grade != -1:
    count += 1
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariant.

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = 0               # count is the number of grades processed so far.
while grade != -1:
    count += 1
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariant.

Application: Count grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = 0               # count is the number of grades processed so far.
while grade != -1:
    count += 1
    grade = int(input())
print(count)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

```
90    80    85    90    100    0    85    -1
```

Application: Average grade.

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

```
90   80   85   90   100   0   85   -1
```

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Remember to do online, not offline, computation.

```
90    80    85    90    100    0    85    -1
```

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Introduce program variables whose values describe "state".**

```
90    80    85    90    100    0    85    -1
```

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = _________       # count is the number of grades processed so far.
sum =   _________       # sum is the sum of the grades processed so far.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Introduce program variables whose values describe "state".**

A counter `count` *value* and a running sum `sum` *value*.

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = _________       # count is the number of grades processed so far.
sum =   _________       # sum is the sum of the grades processed so far.
#.Initialize. (α)
while grade != -1:
    count += 1; sum = sum + count
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariants.

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = 0               # count is the number of grades processed so far.
sum =   0               # sum is the sum of the grades processed so far.
while grade != -1:
    count += 1; sum = sum + count
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariants.

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = 0               # count is the number of grades processed so far.
sum =   0               # sum is the sum of the grades processed so far.
while grade != -1:
    count += 1; sum = sum + count
    grade = int(input())
print(sum / count);
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Average grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
count = 0               # count is the number of grades processed so far.
sum =   0               # sum is the sum of the grades processed so far.
while grade != -1:
    count += 1; sum = sum + count
    grade = int(input())
if count == 0: print("no grades")
else: print(sum/count)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Data Processing

Highest grade

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

90	80	85	90	100	0	85	-1

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Keeping track of highest.

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = _________     # highest is max of the grades processed so far.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Introduce program variables whose values describe "state".**

highest | *value*

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = _________     # highest is max of the grades processed so far.
#.Initialize. (α)
while grade != -1:
    if grade > highest: highest = grade
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariant.

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = _________     # highest is max of the grades processed so far.
#.Initialize. (α)
while grade != -1:
    highest = max(highest, grade)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariant.

Wrong! Need to distinguish between no grades and everyone got a 0;

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = 0             # highest is max of the grades processed so far.
while grade != -1:
    highest = max(highest, grade)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariants.

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = -1            # highest is max of grades processed so far, or -1.
while grade != -1:
    highest = max(highest, grade)
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariants.

Application: Highest grade.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
highest = -1            # highest is max of grades processed so far, or -1.
while grade != -1:
    highest = max(highest, grade)
    grade = int(input())
if highest == -1: print("no grades")
else: print(highest)
```

Coding order	
(1) body	β; `grade=int(input())`
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

```
90    80    85    90    100    0    85    -1
```

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

```
90    80    85    90    100    0    85    -1
```

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Counting the number of occurrences of each grade.

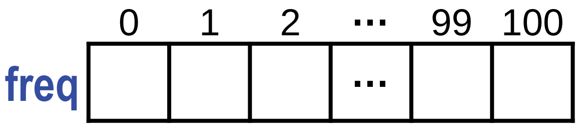
```
90    80    85    90    100    0    85    -1
```

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
freq  = [ ___ ] * 101   # For each k, freq[k] is # of grades of k so far.
#.Initialize. (α)
while grade != -1:
    #.Process v. (β)
    grade = int(input())
#.Finalize. (γ)
```

☞ **Introduce program variables whose values describe “state”.**

Need 101 counters.



Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
freq  = [ ___ ] * 101   # For each k, freq[k] is # of grades of k so far.
while grade != -1:
    freq[grade] += 1
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariant.

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
freq  = [0] * 101       # For each k, freq[k] is # of grades of k so far.
while grade != -1:
    freq[grade] += 1
    grade = int(input())
#.Finalize. (γ)
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariant.

Application: Distribution of grades.

```
grade = int(input())    # grade is the next grade to be processed, or -1.
freq  = [0] * 101       # For each k, freq[k] is # of grades of k so far.
while grade != -1:
    freq[grade] += 1
    grade = int(input())
print("grade frequency")
for g in range(0, 101): print(g, freq[g])
```

Coding order	
(1) body	β; grade=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Compressing a file of integers.

```
10  10  10  10  10  10  1   1   1   1   1   1   1   1   7   7   7   8   9   10  10  10  -1
```

A sequence of equal values is called a run. Each run of *n* instances of *r* can be encoded as a pair of integers, ⟨*r*,*n*⟩.

```
10  6   1   8   7   3   8   1   9   1   10  3   -1  -1
```

A run-encoded file will be shorter if there aren't too many runs of length one.

N.B. Recall that the inputs are assumed to each occur on separate lines despite th spicture showing them all on the same line.

Application: Compressing a file of integers.

```
10  10  10  10  10  10  1  1  1  1  1  1  1  1  7  7  7  8  9  10  10  10  -1
```

A sequence of equal values is called a run. Each run of *n* instances of *r* can be encoded as a pair of integers, ⟨*r*,*n*⟩.

```
10  6  1  8  7  3  8  1  9  1  10  3  -1  -1
```

A run-encoded file will be shorter if there aren't too many runs of length one.

N.B. Recall that the inputs are assumed to each occur on separate lines despite this picture showing them all on the same line.

Application: Compressing a file of integers.

```
10  10  10  10  10  10  1  1  1  1  1  1  1  1  7  7  7  8  9  10  10  10  -1
```

A sequence of equal values is called a run. Each run of *n* instances of *r* can be encoded as a pair of integers, ⟨*r*,*n*⟩.

```
10  6  1  8  7  3  8  1  9  1  10  3  -1  -1
```

A run-encoded file will be shorter if there aren't too many runs of length one.

N.B. We shall emit each ⟨*r*,*n*⟩ on a separate line despite the picture showing them all on the same line.

Application: Write a program to run encode an input file.

- ☞ **Program top-down, outside-in.**
- ☞ **Master stylized code patterns, and use them.**

Use the online-computation pattern.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
#.Initialize. (α)
while v != -1:
    #.Process v. (β)
    v = int(input())
#.Finalize. (γ)
```

☞ **Program top-down, outside-in.**

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

Having by now completely mastered the online-computation pattern, you might just leave blank lines (where placeholders have been) when you instantiate it .

Use the online-computation pattern.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

☞ **Program top-down, outside-in.**

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

Follow the standard coding order.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

```
10  10  10  10  10  10  1  1  1  1  1 | 1  1  1  7  7  7  8  9  10  10  10  -1
```

Stop the music at an arbitrary, but well-chosen, place marked by the bar.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

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

v

10 10 10 10 10 10 1 1 1 1 1 | [1] 1 1 7 7 7 8 9 10 10 10 -1

The state: The next value to be processed is v.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

☞ **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.**

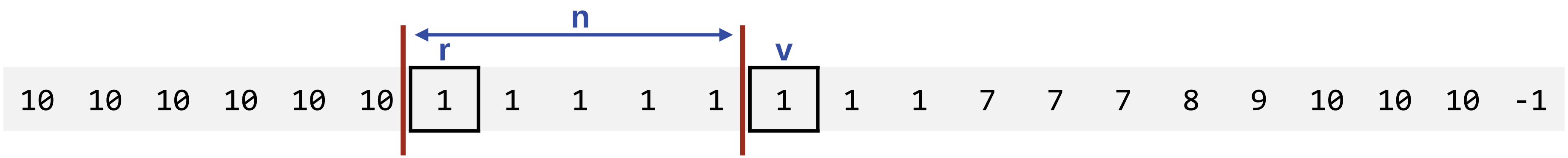
r, v

10 10 10 10 10 10 | [1] 1 1 1 1 | [1] 1 1 7 7 7 8 9 10 10 10 -1

The state: The next value to be processed is v, and we are in a run of r values.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

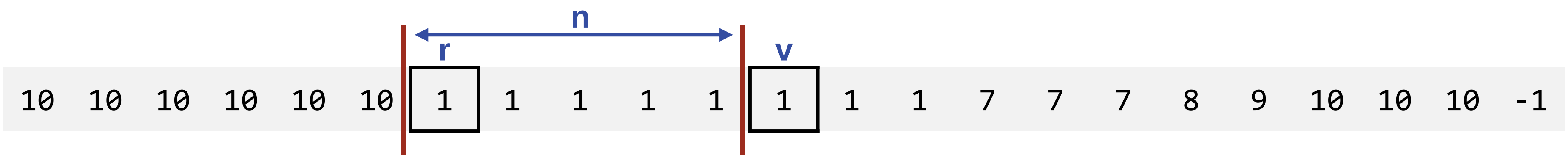
☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.**



The state: The next value to be processed is `v`, and we are in a run of `r` values of length `n`.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

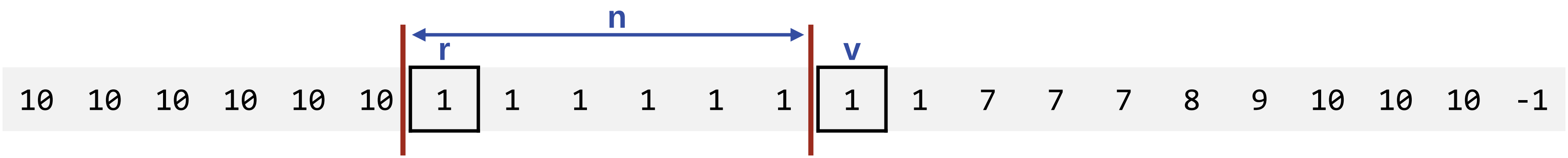
☞ **Body. Do 1st. Play "musical chairs" and "stop the music". Characterize the "program state" when the music stops, i.e., at the instant the loop-body is about to execute yet again.**



The state: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

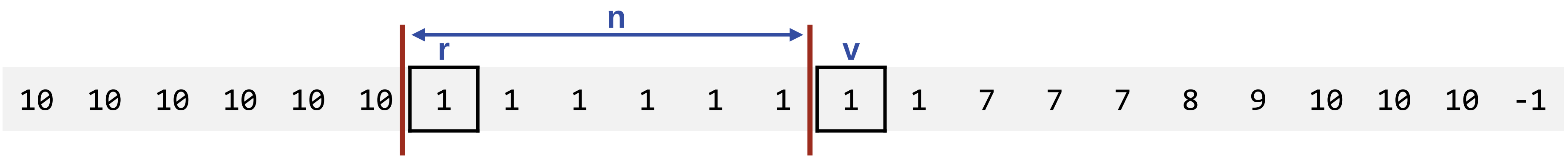
☞ **Body. Do 1st. Play "musical chairs" and "stop the music". Characterize the "program state" when the music stops, i.e., at the instant the loop-body is about to execute yet again.**



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

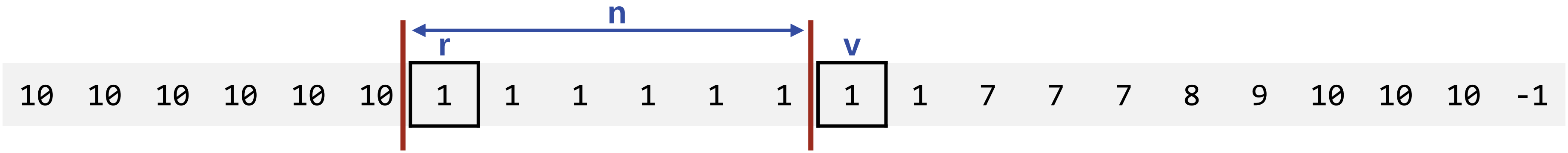
☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the “loop invariant”), and what would have changed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



VARIANT: The number of input values remaining to be processed.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

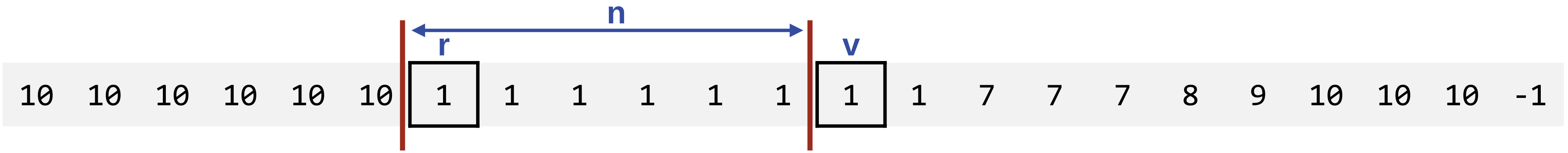
☞ **Body. Do 1st. Play "musical chairs" and "stop the music". Characterize the "program state" when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the "loop invariant"), and what would have changed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



VARIANT: The number of input values remaining to be processed (which the online-computation pattern reduces by 1).

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

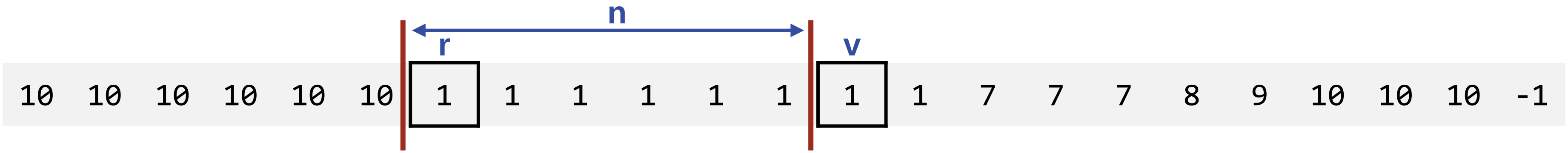
☞ **Body. Do 1st. Play "musical chairs" and "stop the music". Characterize the "program state" when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the "loop invariant"), and what would have changed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



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

☞ **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.**

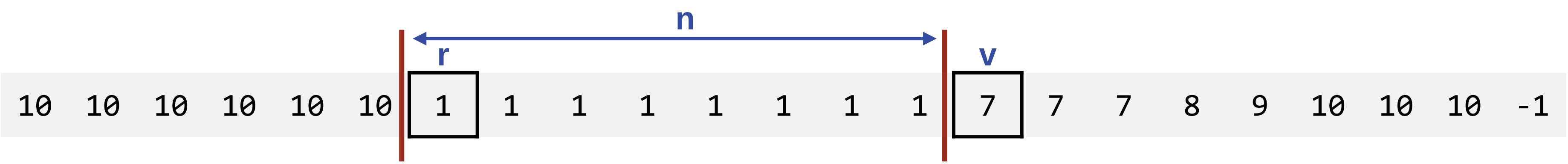


INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    if v == r: n += 1
    else: ___________
    v = int(input())
γ
```

First case: Still in the middle of a run.

☞ **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.**



INVARIANT: The next value to be processed is v, and we are in a run of r values of length n. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        ______________
    v = int(input())
γ
```

First case: Still in the middle of a run.

Second case: Output the now-completed run.

Maintain invariant.

INVARIANT: The next value to be processed is v, and we are in a run of r values of length n. All completed runs seen have been output.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
α
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
γ
```

First case: Still in the middle of a run.

Second case: Output the now-completed run, and begin the next run.

Maintain invariant.

INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
γ
```

First case: Still in the middle of a run.

Second case: Output the now-completed run, and begin the next run.

Completion of the loop body advances v.

Maintain invariant.

INVARIANT: The next value to be processed is v, and we are in a run of r values of length n. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
r: int = ___            # r is first value in the current run.
n: int = ___            # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
γ
```

Establish invariant.

Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

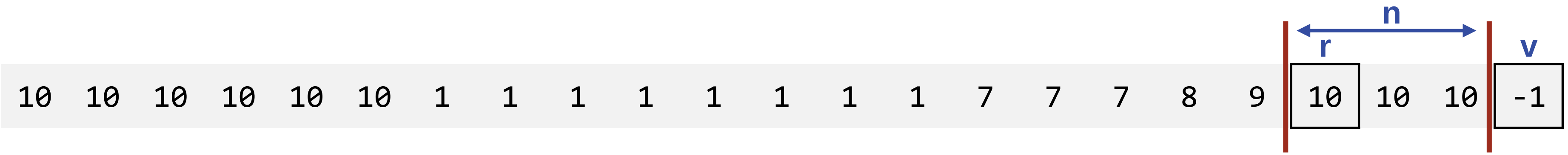
```
n  r
   v
|| 10 | 10  10  10  10  10  1  1  1  1  1  1  1  1  7  7  7  8  9  10  10  10  -1
```

INVARIANT: The next value to be processed is v, and we are in a run of r values of length n. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
r: int = v              # r is first value in the current run.
n: int = 0              # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
γ
```

Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

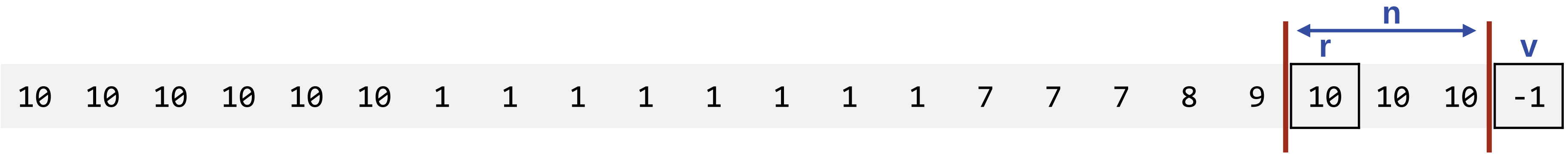
Establish invariant.



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())    # v is the next integer to be processed, or -1.
r: int = v               # r is first value in the current run.
n: int = 0               # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
γ
```

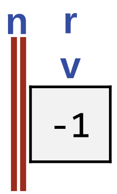
Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
r: int = v              # r is first value in the current run.
n: int = 0              # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
print(r, n)
print("-1 -1")
```

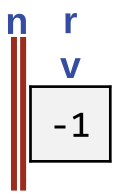
Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
r: int = v              # r is first value in the current run.
n: int = 0              # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
print(r, n)
print("-1 -1")
```

Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case



INVARIANT: The next value to be processed is `v`, and we are in a run of `r` values of length `n`. All completed runs seen have been output.

```
v: int = int(input())   # v is the next integer to be processed, or -1.
r: int = v              # r is first value in the current run.
n: int = 0              # n is length of the current run prefix.
while v != -1:
    if v == r: n += 1
    else:
        print(r, n)
        r = v; n = 1
    v = int(input())
if n != 0: print(r, n)
print("-1 -1")
```

Coding order	
(1) body	β; v=int(input())
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Application: Write a program to decode a run-encoded file.

☞ **Program top-down, outside-in.**
☞ **Master stylized code patterns, and use them.**

Use online-computation pattern

```
v: int = int(input())   # v is the next integer to be processed, or -1.
α
while v != -1:
    β
    v = int(input())
γ
```

☞ **Program top-down, outside-in.**

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

Use online-computation pattern, generalized to read values two inputs at a time.

```
r: int; n: int            # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
#.Assign r and n the values of two integers read from the same line of input.
α
while r != -1:
    β
    #.Assign r and n the values of two integers read from the same line of input.
γ
```

Constructs

statements

To handle two inputs on the same line, we need to extend our base language.

```
variable₁, variable₂ = input().split
variable₁ = int(variable₁)
variable₂ = int(variable₂)
```

Meaning: Read the next line of input, which is assumed to contain two base-10 integers separated by spaces, convert them to their binary fixed-point forms, assign those **int** values to $variable_1$ and $variable_2$, respectively, and advance the input cursor to the beginning of the next line.

N.B. The case of one integer on the line is handled by the assignment statement:

```
variable = int(input())
```

Use online-computation pattern, generalized to read values two at a time.

```
r: int; n: int          # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
r,n = input().split(); r = int(r); n = int(n)
α
while r != -1:
    β
    r,n = input().split(); r = int(r); n = int(n)
γ
```

INVARIANT: Runs have been output for all ⟨r,n⟩ processed so far.

```
r: int; n: int           # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
r,n = input().split(); r = int(r); n = int(n)
α
while r != -1:
    β
    r,n = input().split(); r = int(r); n = int(n)
γ
```

Coding order	

(1) body	β
	`r,n = input().split()`
	`r = int(r); n = int(n)`

(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

INVARIANT: Runs have been output for all ⟨r,n⟩ processed so far.

```
r: int; n: int              # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
r,n = input().split(); r = int(r); n = int(n)
α
while r != -1:
    for k in range(n): print(r)
    r,n = input().split(); r = int(r); n = int(n)
γ
```

Coding order	

(1) body	β
	`r,n = input().split()`
	`r = int(r); n = int(n)`

(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Maintain invariant.

INVARIANT: Runs have been output for all ⟨r,n⟩ processed so far.

```
r: int; n: int          # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
r,n = input().split(); r = int(r); n = int(n)
while r != -1:
    for k in range(n): print(r)
    r,n = input().split(); r = int(r); n = int(n)
γ
```

Coding order	

(1) body	β
	`r,n = input().split()`
	`r = int(r); n = int(n)`

(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Establish invariant.

```
r: int; n: int          # Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
r,n = input().split(); r = int(r); n = int(n)
while r != -1:
    for k in range(n): print(r)
    r,n = input().split(); r = int(r); n = int(n)
print(-1)
```

Coding order	

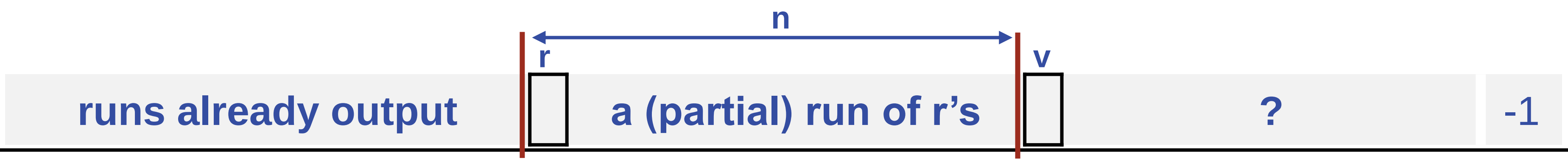
(1) body	β
	`r,n = input().split()`
	`r = int(r); n = int(n)`

(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

Other precepts illustrated

Precepts used without mention.

- ☞ **Write the representation invariant of an individual variable as an end-of-line comment.**
- ☞ **Invent (or learn) vocabulary for concepts that arise in a problem.**
- ☞ **Invent (or learn) diagrammatic ways to express concepts.**
- ☞ **Alternate between using a concrete example to guide you in characterizing "program state", and an abstract version that refers to all possible examples.**



Other precepts illustrated

Precepts used without mention.

- ☞ **Initialization. Do 3rd. Initialize variables so that the loop invariant is established prior to the first iteration. Substitute those initial values into the invariant, and bench check the first iteration with respect to that initial instantiation of the invariant.**
- ☞ **Finalization. Do 4th, but don't forget. Leverage that the looping condition is false, the loop invariant remains true, and the loop variant is 0.**
- ☞ **Boundary conditions. Dead last, but don't forget them.**
- ☞ **Find boundary conditions at extrema, and at singularities, e.g., biggest, smallest, 0, edges, etc.**
