

Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

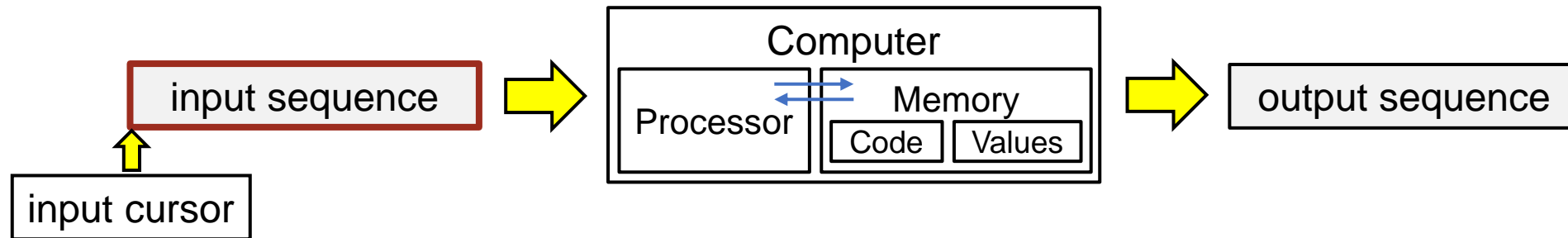
Online Algorithms

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

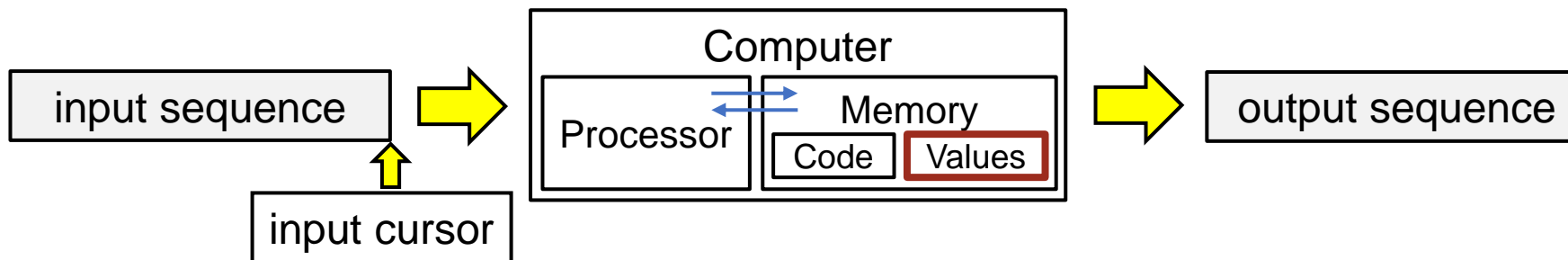
We illustrate many important programming precepts.

Application: Process an input file of unbounded length.



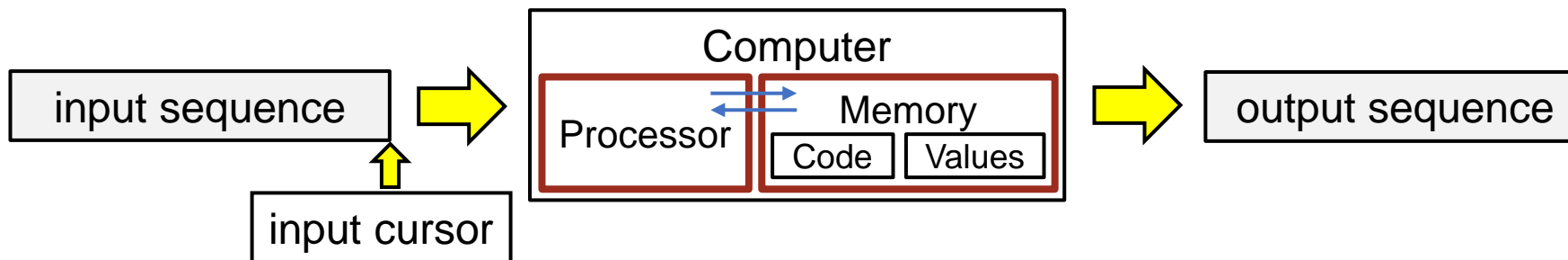
Offline-computation pattern: calls for **reading all values first**.

```
/* Input. */  
/* Compute. */  
/* Output. */
```



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

```
/* Input. */  
/* Compute. */  
/* Output. */
```

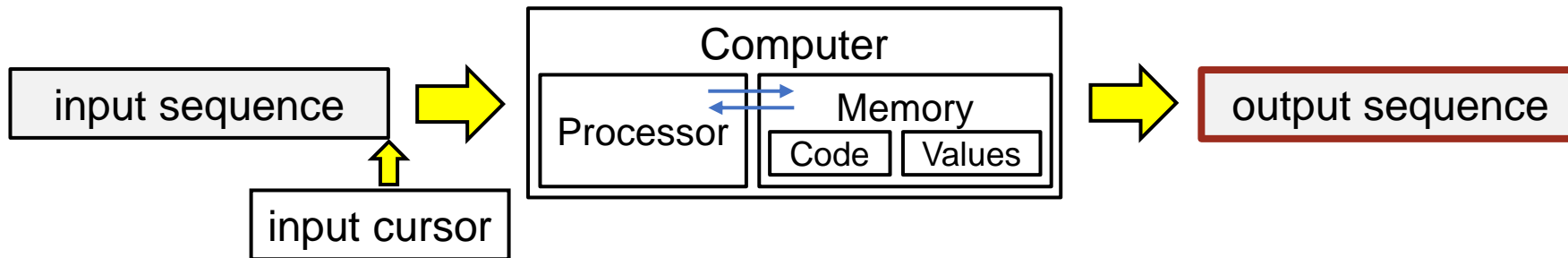


Offline-computation pattern: calls for reading all values first, then processing them, *then outputting results.*

```

/* Input. */
/* Compute. */
/* Output. */

```

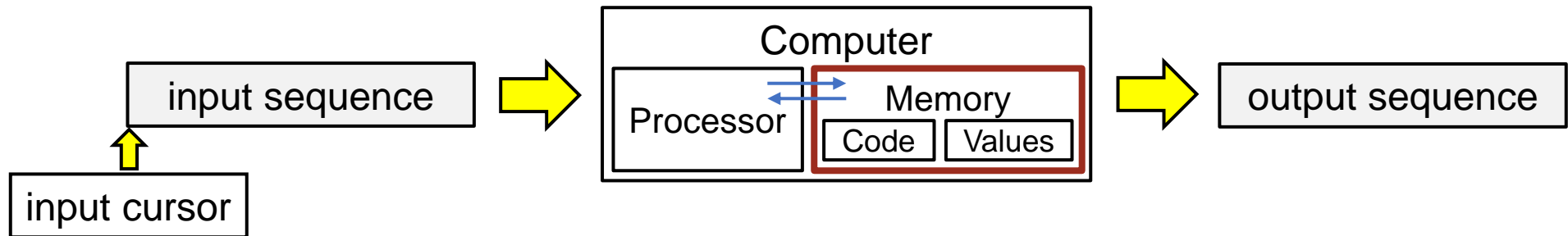


Offline-computation pattern: A **mismatch** because the **memory is finite**, but the **input is unbounded**.

```

/* Input. */
/* Compute. */
/* Output. */

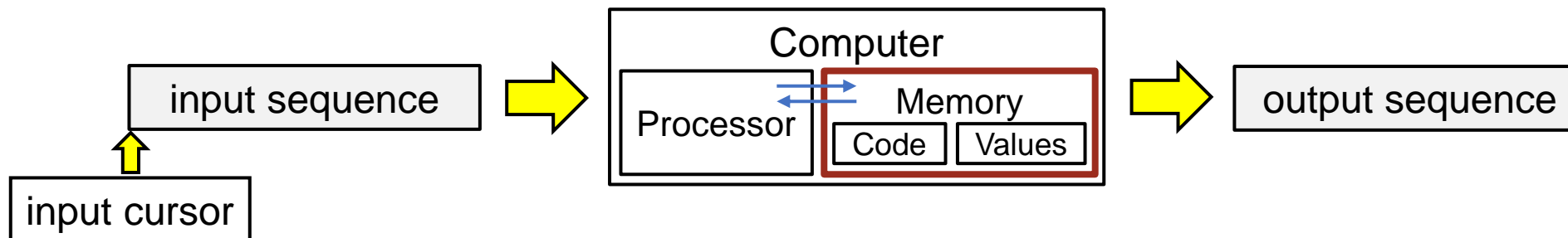
```



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. */
}
```

The diagram illustrates the specialization of the general-iteration pattern into the online-computation pattern. Dotted lines connect the following elements:

- The initialization line `v = /* first input value */;` to `/* Initialize. */`.
- The loop condition `v != /* stoppingValue */` to `/* not finished */`.
- The loop body `/* Process v. */` to `/* Compute. */`.
- The update statement `v = /* next input value */;` to `/* Go on to next. */`.
- The finalization line `/* Finalize. */` to the corresponding line in the right-hand snippet.

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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/* Initialize. */
while ( v != -1 ) {
    /* Process v. */
    v = in.nextInt();
}
/* Finalize. */
```

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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```


90 80 85 90 100 0 85 -1

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

Application	α	β	γ
Print			
Count			
Average			
Highest			
Distribution			


```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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).

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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).

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Coding order	
(1) body	β ; grade=in.nextInt();
(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.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: **Print** grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: Print grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    System.out.println(grade);
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: Print grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
while ( grade != -1 ) {
    System.out.println(grade);
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: Print grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
while ( grade != -1 ) {
    System.out.println(grade);
    grade = in.nextInt();
}
```

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

Application: **Count** grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: Count grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = _____; // Count of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 Introduce program variables whose values describe “state”.

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

Application: Count grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = _____; // Count of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    count++;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariant.

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

Application: Count grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = 0;           // Count of grades processed so far.
while ( grade != -1 ) {
    count++;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Establish invariant.

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

Application: Count grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = 0;           // Count of grades processed so far.
while ( grade != -1 ) {
    count++;
    grade = in.nextInt();
}
```

```
System.out.println(count);
```

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

Application: *Average* grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

90 80 85 90 100 0 85 -1

Application: Average grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 Introduce program variables whose values describe “state”.

90 80 85 90 100 0 85 -1

Application: Average grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = _____; // Count of grades processed so far.
int sum = _____; // Sum of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

A counter count value and a running sum sum value .

Application: Average grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = _____; // Count of grades processed so far.
int sum = _____; // Sum of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    count++; sum = sum+count;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariants.

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

Application: *Average* grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = 0;           // Count of grades processed so far.
int sum = 0;             // Sum of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    count++; sum = sum+count;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Establish invariants.

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

Application: *Average* grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = 0;           // Count of grades processed so far.
int sum = 0;             // Sum of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    count++; sum = sum+count;
    grade = in.nextInt();
}
System.out.println(sum/count);
```

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

Application: *Average* grade.

```

int grade = in.nextInt(); // Next grade to be processed, or -1.
int count = 0;           // Count of grades processed so far.
int sum = 0;             // Sum of grades processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    count++; sum = sum+count;
    grade = in.nextInt();
}
if (count==0)
    System.out.println("no grades");
else System.out.println(sum/count);

```

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

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

90 80 85 90 100 0 85 -1

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = _____; // Highest grade processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

👉 Introduce program variables whose values describe “state”.

highest value

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = _____; // Highest grade processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    if ( grade > highest ) highest = grade;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariant.

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

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = _____; // Highest grade processed so far.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    highest = Math.max(highest,grade);
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariant.

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

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

Application: **Highest** grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = 0; // Highest grade processed so far.
while ( grade != -1 ) {
    highest = Math.max(highest,grade);
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Establish invariants.

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = -1;        // Highest grade processed so far, or -1.
while ( grade != -1 ) {
    highest = Math.max(highest,grade);
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Establish invariants.

Application: Highest grade.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int highest = -1;        // Highest grade processed so far, or -1.
while ( grade != -1 ) {
    highest = Math.max(highest,grade);
    grade = in.nextInt();
}
if (highest==-1)
    System.out.println("no grades");
else System.out.println(sum/count);
```

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

Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **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.

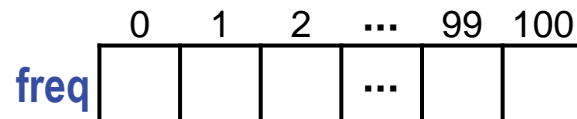
90 80 85 90 100 0 85 -1

Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int freq[] = new int[101] // freq[k] is count of a grade of k.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    /*  $\beta$ : Process v. */
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 Introduce program variables whose values describe “state”.

Need 101 counters.



Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int freq[] = new int[101] // freq[k] is count of a grade of k.
/*  $\alpha$ : Initialize. */
while ( grade != -1 ) {
    freq[grade]++;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariant.

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

Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int freq[] = new int[101] // freq[k] is count of a grade of k; initially 0.
while ( grade != -1 ) {
    freq[grade]++;
    grade = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Establish invariant.

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

Application: Distribution of grades.

```
int grade = in.nextInt(); // Next grade to be processed, or -1.
int freq[] = new int[101] // freq[k] is count of a grade of k; initially 0.
while ( grade != -1 ) {
    freq[grade]++;
    grade = in.nextInt();
}
for(int g=0; g<101; g++)
    System.out.println(g, freq[g]);
```

Coding order	
(1) body	β ; grade=in.nextInt();
(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, $\langle r, n \rangle$.

```
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.

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**.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

Follow the standard coding order.

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Coding order	
(1) body	β ; v=in.nextInt();
(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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

 **Body. Do 1st. Play “musical chairs” and “stop the music”.**

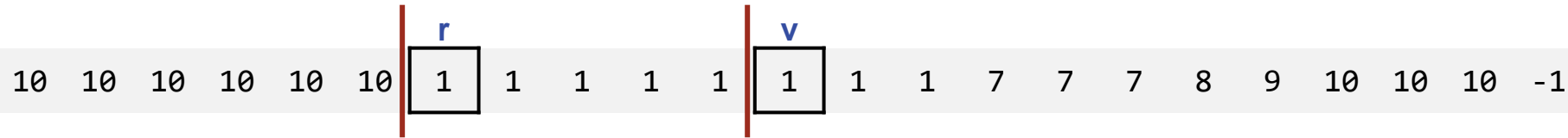
10 10 10 10 10 10 1 1 1 1 1 **1** 1 1 7 7 7 8 9 10 10 10 -1

v

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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

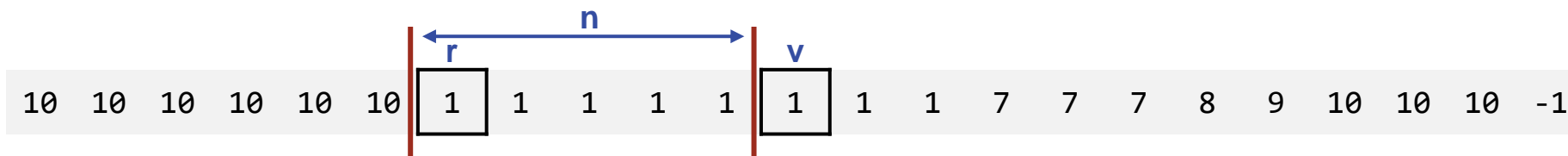
-
-  **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.**
-



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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

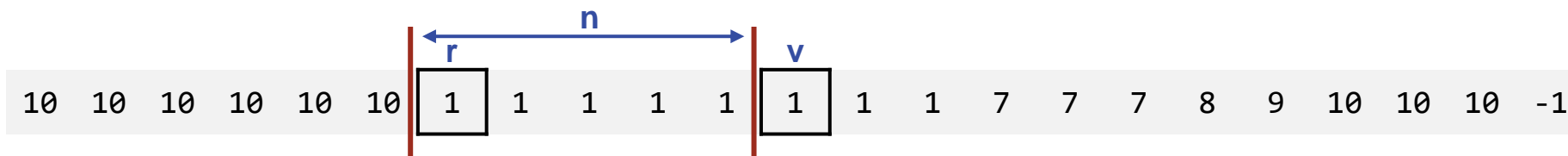
-
- ☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.**
-



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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

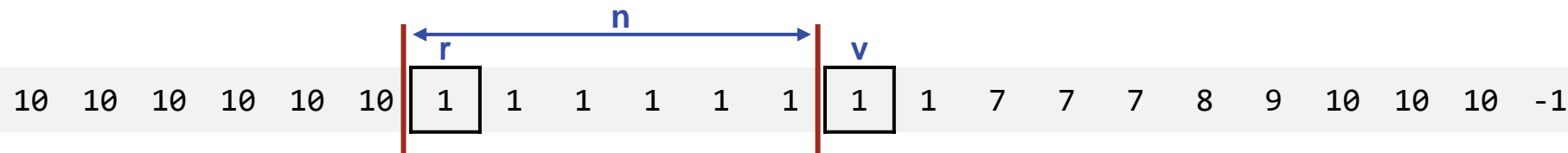
☞ **Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.**



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.**

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

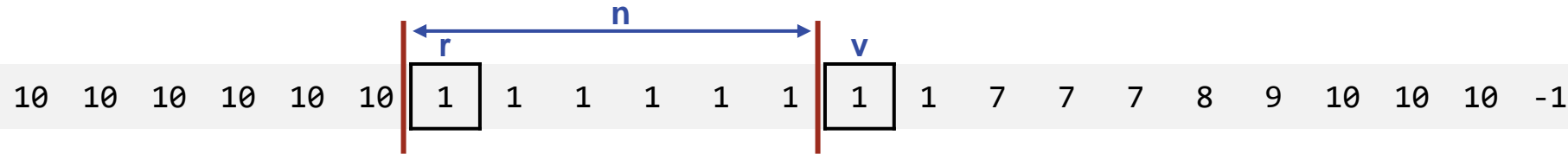
Body. Do 1st. Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again.



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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

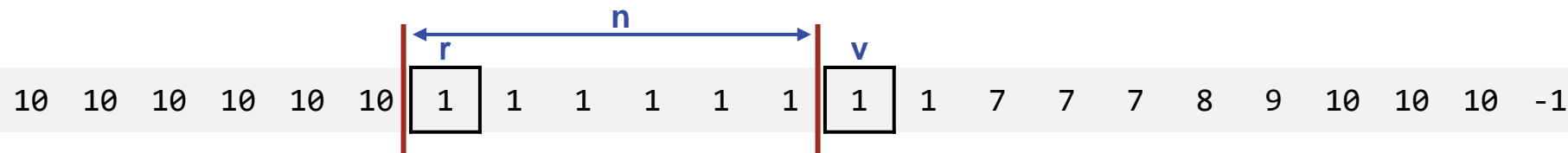
-
- ☞ **Body. Do 1st.** Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again. **If you had stopped one iteration later, what would have looked the same (the “loop invariant”), and what would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



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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

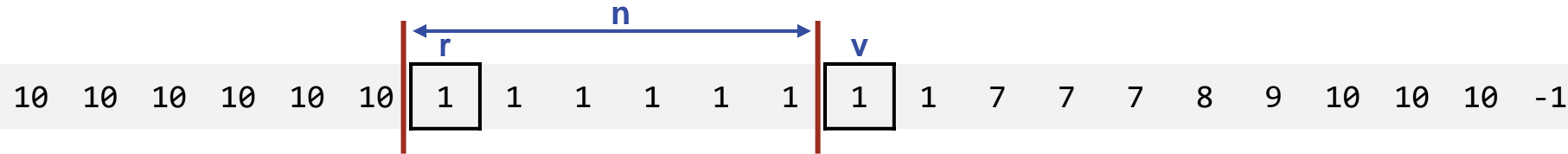
-
- ☞ **Body. Do 1st. Play “musical chairs” and “stop the music”.** Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the “loop invariant”), and **what would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



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

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

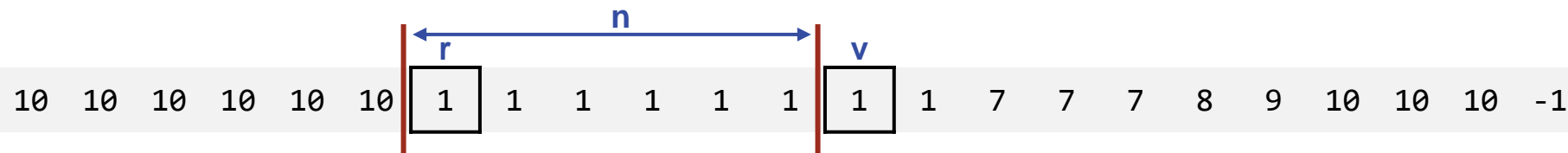
-
- ☞ **Body. Do 1st.** Play “musical chairs” and “stop the music”. Characterize the “program state” when the music stops, i.e., at the instant the loop-body is about to execute yet again. If you had stopped one iteration later, what would have looked the same (the “loop invariant”), and **what would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

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

☞ **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**.**

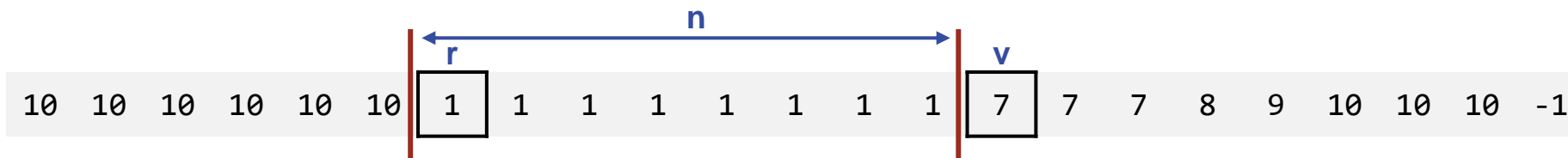


INVARIANT: The next value to be processed is v , and we are in a run of r values of length n . All completed runs seen have been output.

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    if ( v==r ) n++;
    else _____
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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.**

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
    }
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

First case: Still in the middle of a run.

Second case: Output the now-completed run.

Maintain invariant.



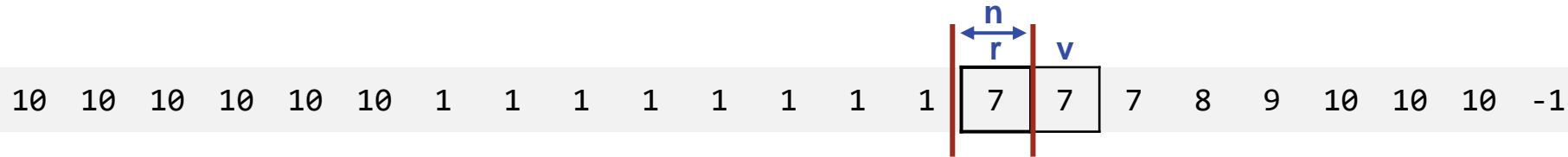
INVARIANT: The next value to be processed is v , and we are in a run of r values of length n . All completed runs seen have been output.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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.

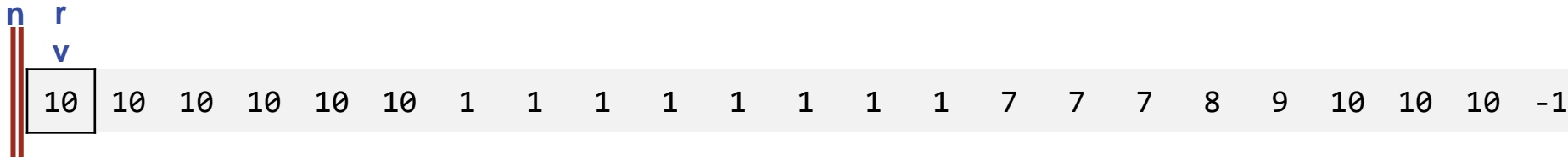
```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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.

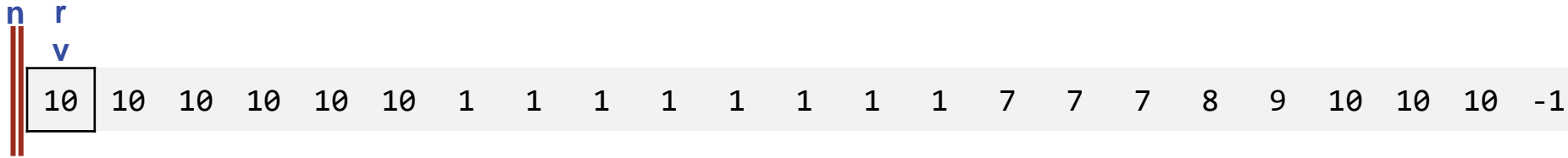


INVARIANT: The next value to be processed is v , and we are in a run of r values of length n . All completed runs seen have been output.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = __; int n = __; // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Establish invariant.

Coding order	
(1) body	β ; $v=in.nextInt()$;
(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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = v;  int n = 0;  // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
/* γ: Finalize. */
```

Establish invariant.

Coding order	
(1) body	β ; <code>v=in.nextInt();</code>
(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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = v;  int n = 0;  // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

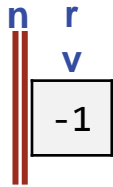
Coding order	
(1) body	β ; $v=in.nextInt()$;
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case



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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = v;  int n = 0;  // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
System.out.print(r + " " + c + " ");
System.out.println("-1 -1");
```

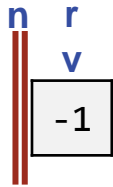
Coding order	
(1) body	β ; $v=in.nextInt()$;
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case



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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = v;  int n = 0;  // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
System.out.print(r + " " + c + " ");
System.out.println("-1 -1");
```

Coding order	
(1) body	β ; $v=in.nextInt()$;
(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.

```
int v = in.nextInt();    // Next integer to be processed, or -1.
int r = v;  int n = 0;  // In run of r values of length n.
while ( v != -1 ) {
    if ( v==r ) n++;
    else {
        System.out.print(r + " " + c + " ");
        r = v; n = 1;
    }
    v = in.nextInt();
}
if ( n!=0 )
    System.out.print(r + " " + c + " ");
System.out.println("-1 -1");
```

Coding order	
(1) body	β ; $v=in.nextInt()$;
(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.

```
int v = in.nextInt();      // Next integer to be processed, or -1.
/*  $\alpha$ : Initialize. */
while ( v != -1 ) {
    /*  $\beta$ : Process v. */
    v = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

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

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

```
int r = in.nextInt(); int n = in.nextInt(); // Next ⟨r,n⟩ to process, or ⟨-1,-1⟩.
/*  $\alpha$ : Initialize. */
while ( r != -1 ) {
    /*  $\beta$ : Process ⟨r,n⟩. */
    r = in.nextInt(); n = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

INVARIANT: Runs have been output for all $\langle r,n \rangle$ processed so far.

```
int r = in.nextInt(); int n = in.nextInt(); // Next  $\langle r,n \rangle$  to process, or  $\langle -1,-1 \rangle$ .
/*  $\alpha$ : Initialize. */
while ( r != -1 ) {
    /*  $\beta$ : Process  $\langle r,n \rangle$ . */
    r = in.nextInt(); n = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Coding order	
(1) body	β ; <code>r=in.nextInt(); n=in.nextInt();</code>
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

INVARIANT: Runs have been output for all $\langle r,n \rangle$ processed so far.

```
int r = in.nextInt(); int n = in.nextInt(); // Next  $\langle r,n \rangle$  to process, or  $\langle -1,-1 \rangle$ .
/*  $\alpha$ : Initialize. */
while ( r != -1 ) {
    for (int k=1; k<=n; k++) System.out.print(r + " ");
    r = in.nextInt(); n = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Maintain invariant.

Coding order	
(1) body	β ; <code>r=in.nextInt(); n=in.nextInt();</code>
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

INVARIANT: Runs have been output for all $\langle r,n \rangle$ processed so far.

```
int r = in.nextInt(); int n = in.nextInt(); // Next  $\langle r,n \rangle$  to process, or  $\langle -1,-1 \rangle$ .
while ( r != -1 ) {
    for (int k=1; k<=n; k++) System.out.print(r + " ");
    r = in.nextInt(); n = in.nextInt();
}
/*  $\gamma$ : Finalize. */
```

Establish invariant.

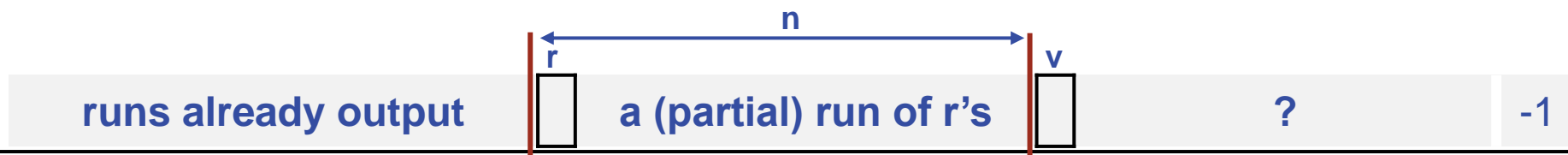
Coding order	
(1) body	β ; <code>r=in.nextInt(); n=in.nextInt();</code>
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

```
int r = in.nextInt(); int n = in.nextInt(); // Next <r,n> to process, or <-1,-1>.
while ( r != -1 ) {
    for (int k=1; k<=n; k++) System.out.print(r + " ");
    r = in.nextInt(); n = in.nextInt();
}
System.out.println (-1);
```

Coding order	
(1) body	β ; <code>r=in.nextInt(); n=in.nextInt();</code>
(2) termination	-
(3) initialization	α
(4) finalization	γ
(5) boundary conditions	exceptions to the general case

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.



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.**
-