Lecture 4: Loop Invariants

CS 2110

September 3, 2026

Today's Learning Outcomes

17. Identify the loop invariant of an iterative method involving an array and visualize it using an array diagram.

18. Use an array diagram to develop an iterative method that maintains the visualized invariant.

19. Write precise specifications for methods involving arrays that use range notation.

Loop Anatomy

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
    sum += a[i];
}
```

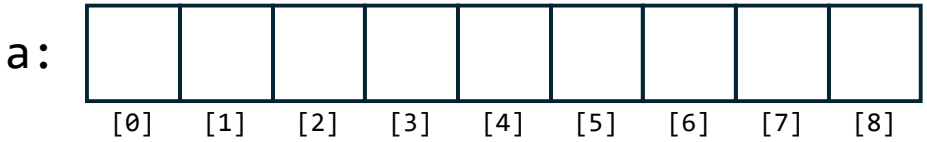
`while` Loops

Built from all the same components as `for` loops.
Every `for` loop can be transformed into a `while` loop.

We'll focus on `while` loops today, but everything also applies to `for` loops.

Range Notation

A **range** of an array is a contiguous subset of its entries.



Range Properties and Array Diagrams

We use **array diagrams** as a tool to visualize properties of array ranges.

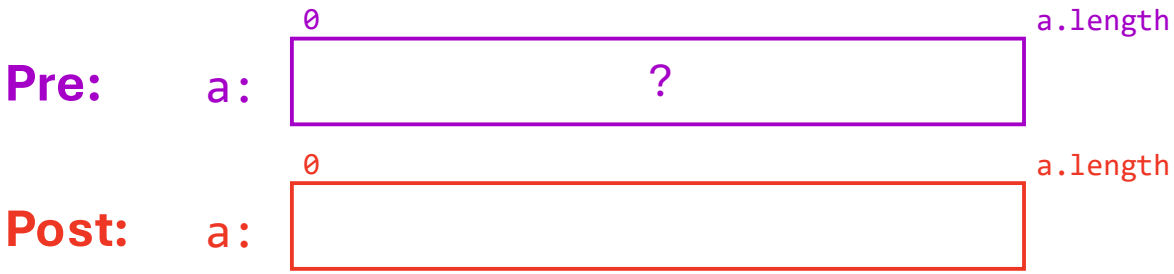
a:

Writing "Loopy" Code

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```

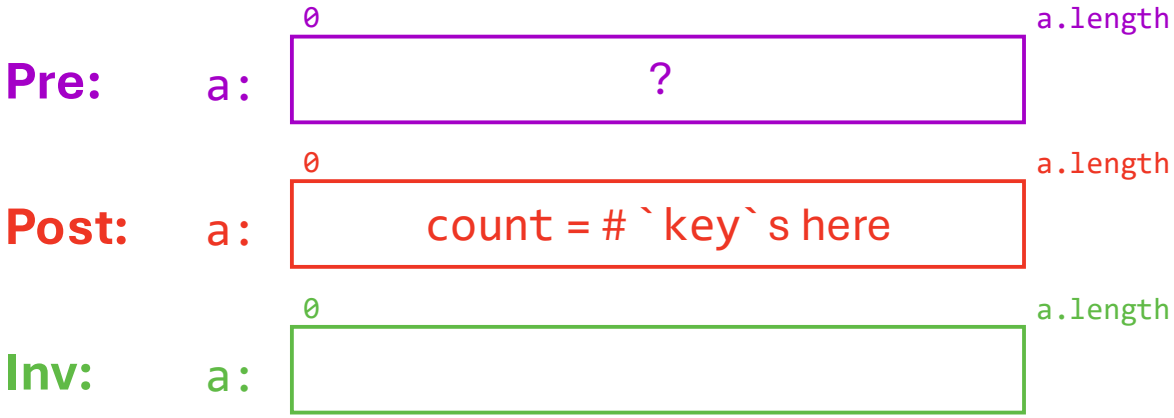
Writing "Loopy" Code

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```



Writing "Loopy" Code

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```



(Loop) Invariants

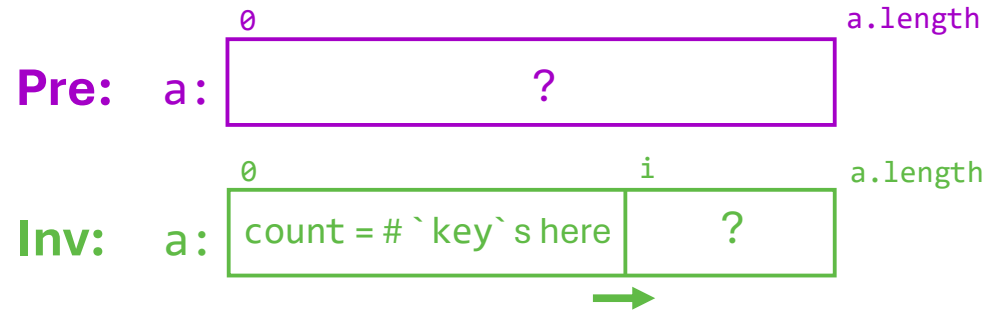
An **invariant** is an assertion about code that is true at multiple predetermined points of its execution.

Once we have a loop invariant diagram, writing its loop becomes mechanical.

Developing a Loop: Initialization

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3
4
5
6    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
7    while (...) { ... }
8  }
```

Initialize local variables to *truthify* the loop invariant when we enter the loop.

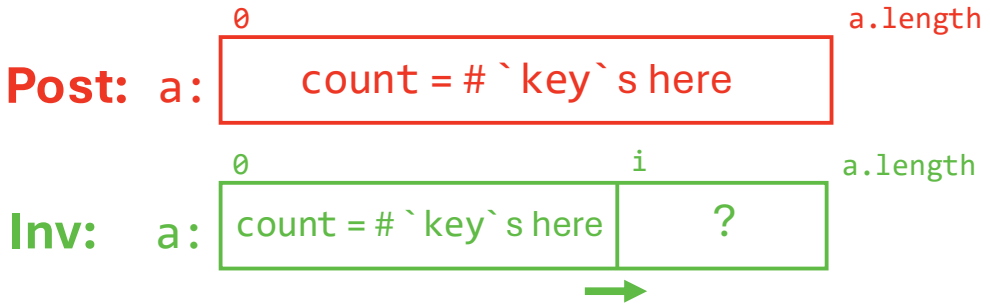


Developing a Loop: Guard + Post-condition

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3    int i = 0; // next index of `a` to check
4    int count = 0;
5    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
6    while (              ) { ... }
7
8  }
```

Guard the loop on a condition that *falsifies* once the loop is done with its work.

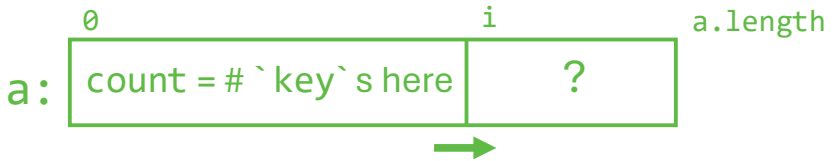
Use loop invariant to meet method post-condition.



Developing a Loop: Body

In each iteration:

- Progress toward the loop's goal
- Restore the loop invariant




```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3    int i = 0;  int count = 0;
4    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
5    while (i < a.length) {
6
7
8
9    }
10   return count;
11 }
```

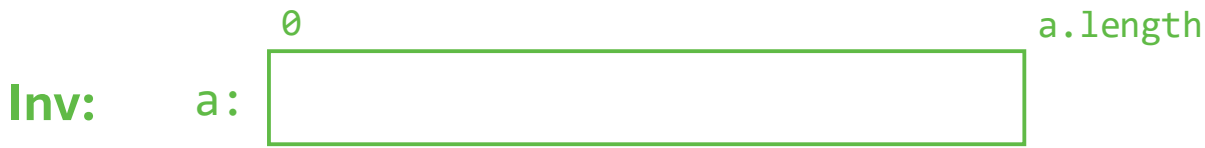
Example 2: `argmin()`

```
1  /** Returns an *index* of the minimum element in `a`.
2   *  Requires that `a.length > 0`. */
3  static int argmin(double[] a) { ... }
```

```
argmin(new double[]{1.0, 3.5, 4.2, 0.7, 6.3, 2.8})
```

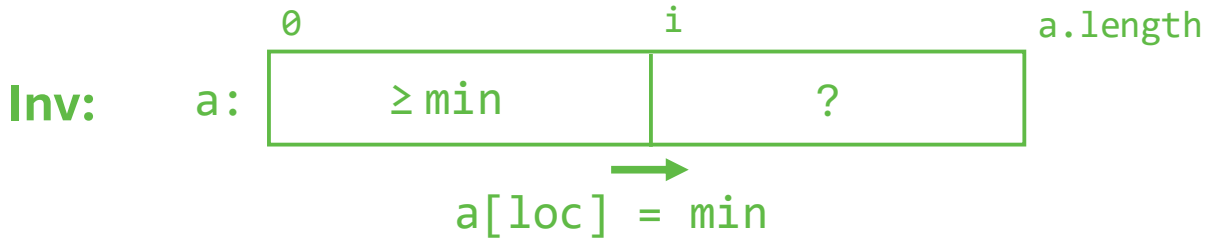
What do we need to keep track of to compute the `argmin()`?

`argmin()` Invariant Diagram



Exercise

Use the array diagram we developed to complete the definition of `argmin()`.

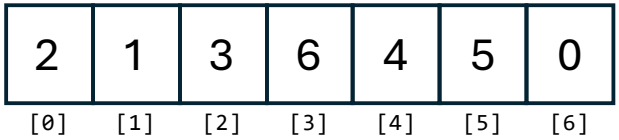



```
1  /** Returns an *index* of the minimum element in `a`.
2   *  Requires that `a.length > 0`. */
3  static int argmin(double[] a) { ... }
```

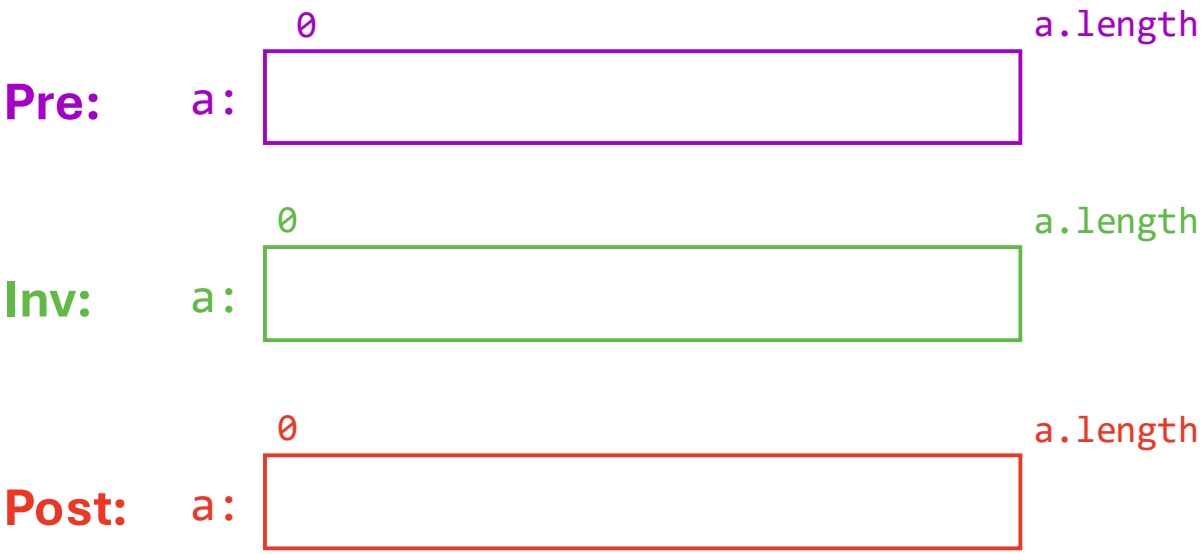
Coding Demo: `argmin()`

Example 3: paritySplit()

```
1  /** Rearranges `a` so that all even elements appear before
2   *  all odd elements. Returns the index of the first odd
3   *  element, or `a.length` if all elements are even. */
4  static int paritySplit(int[] a) { ... }
```



`paritySplit()` Array Diagrams

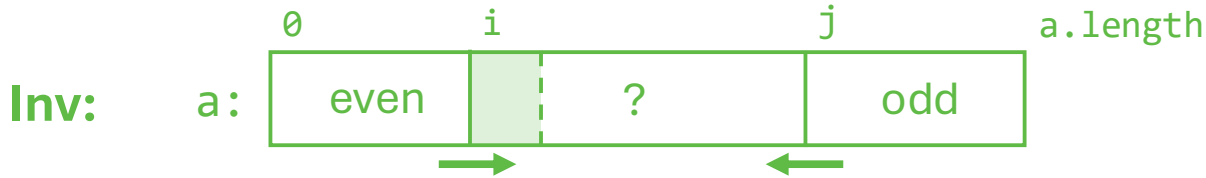


Coding Demo: `paritySplit()`

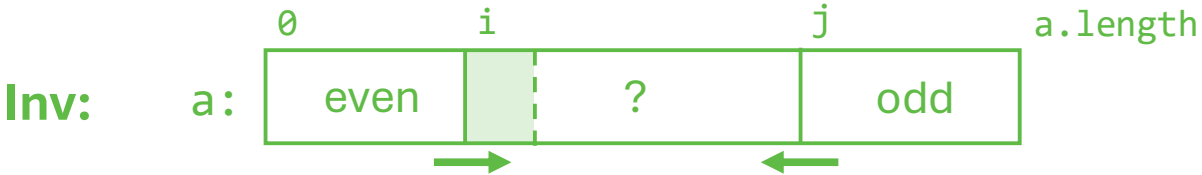
`paritySplit()` Loop Body

Inspect `a[i]` and consider two cases:

1. `a[i]` is even



2. `a[i]` is odd



Review: Steps for Developing Loops

1. Think about loop progress to sketch its invariant.
2. Identify local variables: one per boundary, perhaps others for properties.
3. Use the “Inv” diagram to write the loop invariant comment.
4. Slide the “Inv” boundaries to their "Pre" positions
   - Use overlaps to initialize the loop variables.
   - Use the loop invariant to initialize other local variables.
5. Slide the “Inv" boundaries to their "Post" positions
   - Use overlaps to to determine the loop guard.
   - Use the loop invariant to satisfy the method post-conditions
6. Develop the loop body so that it:
   - Makes progress toward the post-condition.
   - Re-establishes the loop invariant.