To prove a loop in key, one needs

- a loop invariant; it must be
  - initially valid,
  - inductive, i.e. hold after executing the body if it held before,
  - strong enough to prove the post-condition (use case).
- a modifies set; this must contain all variables changed by the loop body.
- a loop variant (or ranking function); it must be
  - non-negative,
  - strictly decreasing for every execution of the loop body.

The loop variant guarantees that the loop terminates.
The rule while_invariant_with_variant_dec

The rule while_invariant_with_variant_dec takes an invariant $inv$, a modifies set $\{m_1, \ldots, m_k\}$ and a variant $v$. The following cases must be proven.

- Initially Valid: $\implies inv \land v \geq 0$
- Body Preserves Invariant:
  
  $\implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land [\{b = COND; \}])b = \text{true} \implies \langle BODY \rangle inv$

- Use Case:

  $\implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land [\{b = COND; \}])b = \text{false} \implies \langle \ldots \rangle \phi$

- Termination:

  $\implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land v \geq 0 \land [\{b = COND; \}])b = \text{true} \implies \{old := v\}\langle BODY \rangle v \leq old \land v \geq 0$
Example: Multiplication

```java
/*@
  @ requires a >= 0 && b >= 0;
  @ ensures \result == a*b;
  @*/

public static int mul(int a, int b) {
    int sum = 0;
    while (b > 0) {
        sum = sum + a;
        b--;
    }
    return sum;
}
```
One possible loop invariant is \( \text{sum} + a \times b = a \times \text{old}(b) \):

```java
/*@ requires a >= 0 && b >= 0;
 @ ensures \result == a*b;
 @*/
public static int mul(int a, int b) {
    int sum = 0;
    /*@ loop_invariant sum + a*b == a*\old(b);
     @ modifies sum, b;
     @ decreases b;
     @*/
    while (b > 0) {
        sum = sum + a;
        b--;
    }
    return sum;
}
```

This is enough to prove it in KeY (Demo)
Algorithm to check if an array contains an element.

```java
/*@
   @ requires arr != null;
   @ ensures \result == (\exists int k; 0 <= k && k < arr.length;
       arr[k] == elem);
   @*/

public static boolean find(int[] arr, int elem) {
   for (int i = 0; i < arr.length; i++) {
      if (arr[i] == elem)
         return true;
   }
   return false;
}
```
What is the loop invariant?

```java
/*@
    @ loop_invariant !(\exists k; 0 <= k && k < i; arr[k] == elem);
    @ loop_invariant 0 <= i && i <= arr.length;
    @ modifies i;
    @ decreases arr.length - i;
    @*/
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == elem)
        return true;
} return false;
```
Demo: Binary Search

```java
/*@ requires arr != null;
  @ requires (\forall int j,k; 0 <= j \&\& j <= k \&\& k < arr.length;
      @ arr[j] <= arr[k]); // array is sorted
  @ ensures \result == (\exists int k; 0 <= k \&\& k < arr.length;
      @ arr[k] == elem);
  @*/

boolean binary(int[] arr, int elem) {
    int lower = 0, upper = arr.length - 1;
    while (lower <= upper) {
        int mid = (lower + upper) / 2;
        assert lower <= mid && mid <= upper;
        if (arr[mid] == elem) {
            return true;
        } else if (arr[mid] > elem) {
            upper = mid - 1;
        } else {
            lower = mid + 1;
        }
    }

    return false;
}
```
/*@ requires arr != null && arr.length > 0;
   @ ensures (\forall int j,k; 0 <= j && j <= k && k < arr.length;
           arr[j] <= arr[k]); // array is sorted @*/

public static boolean bubblesort(int[] arr) {
    for (int i = arr.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j+1]) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
}
Function **BubbleSort** sorts integer array $arr$

$arr$: unsorted sorted

by “bubbling” the largest element of the left unsorted region of $arr$ toward the sorted region on the right.

Each iteration of the outer loop expands the sorted region by one cell.
Sample execution of BubbleSort

1. \( i = 1 \) and \( j = 2 \)
   - Swap 4 and 1
   - \( i = 2 \) and \( j = 3 \)
   - Swap 3 and 4
   - \( i = 3 \) and \( j = 4 \)
   - Swap 2 and 4
   - \( i = 4 \) and \( j = 5 \)
   - Swap 2 and 5