Program Correctness

Spring 2014

Carola Wenk

A Recipe for Computational Tools

Given a problem, we should:

- Specify the input and output.
- Design an algorithm; analyze its performance.

Be precise, mathematical.

- Implement the program to meet specifications; analyze real-world performance.

Use experience and rigorous testing.

This is a very high-level procedure; each particular problem will pose its own challenges in each of these steps.

Sorting

Let's consider the problem of sorting a list of numbers.

<u>Algorithm: </u>[Selection Sort]

1. Find the minimum element in the list.

2. Swap it with the first element.

3. Repeat with the rest of the list.

Why is this algorithm correct? What is the running time?

In the worst case, how many times do we find the minimum?

Algorithm Analysis

This approach to sorting a list is often called “selection” sorting. For a list with i elements, we perform about $c \cdot i$ operations to find the minimum.

Each time we find a minimum, we are reducing the time spent on searching for “future” minima. The list sizes are:

$$n, n-1, n-2, \ldots, 2, 1$$

The corresponding number of operations is:

$$c(n + n - 1 + n - 2 + \cdots + 2 + 1) = c \cdot \sum_{i=1}^{n} i$$

$$c \cdot \sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2} = O(n^2)$$

Specifications

- For selection sort we have verified correctness and worst-case running time.
- But how do we know that our implementation is correct, i.e., that it actually works?
- Why is this important?
- Can we be mathematically precise in defining the input and output?
- It would be nice to come up with a way to answer TRUE/FALSE to whether an input or output meets specifications.

Sorting Specifications

Array → Algorithm/Program → Sorted Array

`A_in = [2,1,9, ...]`

For all `i` in `[0..n-2]`:
`A_out[i] <= A_out[i+1]`

The input is just any array of numbers and the output should be sorted.
Note that there is not always a unique output specification.

Sorting Specifications

Array → Algorithm/ Program → Sorted Array

`A_in = [2,1,9, ...]`

For all `i` and `j` in `[0..n-1]`:
if `i <= j`, then
`A_out[i] <= A_out[j]`.

The input is just any array of numbers and the output should be sorted.
Note that there is not always a unique output specification.

Propositional Logic

We will use logical statements about the input to define the output specification; we can then check whether it is True or False.

<u>And</u>

$x \wedge y$

<u>Or</u>

$x \vee y$

<u>Not</u>

$\neg x$

<u>Commutativity</u>

$$x \vee y = y \vee x$$

$$x \wedge y = y \wedge x$$

<u>Distributivity</u>

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

<u>Associativity</u>

$$x \vee (y \vee z) = (x \vee y) \vee z$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

Propositional Logic

- Implication is written $x \to y$, and is the same as $\neg x \vee y$. When applied to formulas, it tells us that the truth of one formula leads to the truth of another.

- A formula is just a function defined by a logical expression:

$$P(x, y) = (x \wedge y) \vee (x \vee y)$$

Quantification

Universal	Existential
$\forall x, y : \; x + y = y + x$	$\exists x, y : \; x = y$
$\forall x, y, z : \; x = y + z$	$\forall x, y \;\; \exists z : \; x = y + z$

"exists"

"for all"

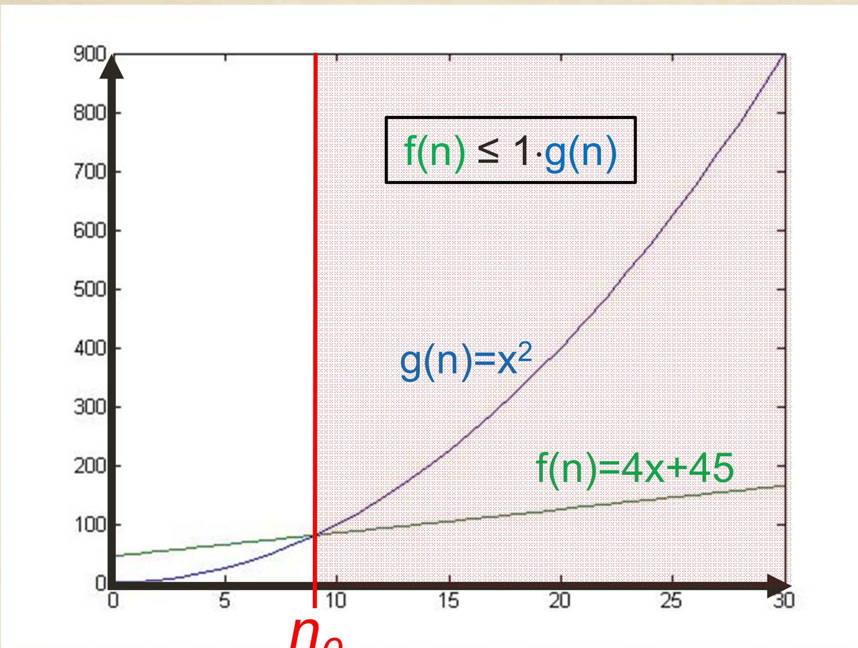
We will construct output specifications using first-order logic on program variables, and prove that the specifications hold after the program has executed.

A note of caution: quantifier order is very important!

Asymptotic Runtime Analysis

Evaluate the abstract runtime of an algorithm by analyzing its <u>asymptotic behavior</u> as a function of the input size.

Time



n_0

Input size

→ How does the algorithm perform if the input grows larger and larger?

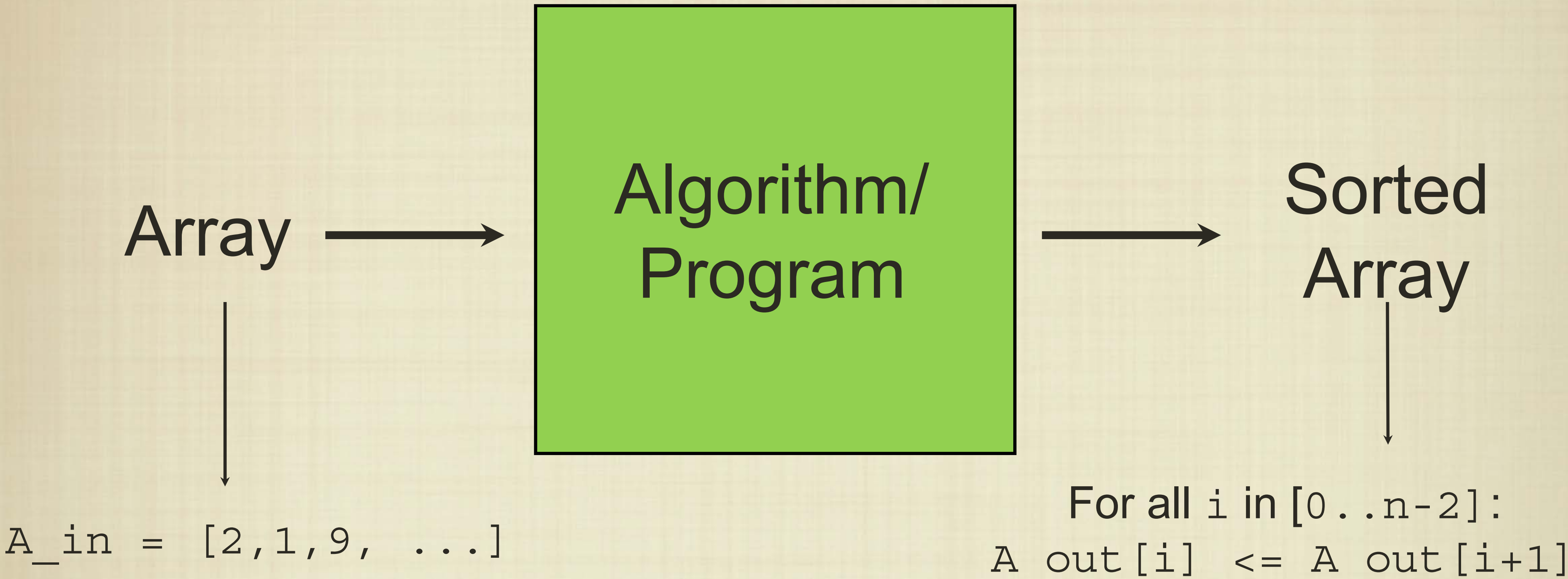
→ Use big-Oh to define classes of functions

$f(n) \in O\big(g(n)\big)$ is defined as

$\exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : \ f(n) \leq c \cdot g(n)$

"exists" "for all"

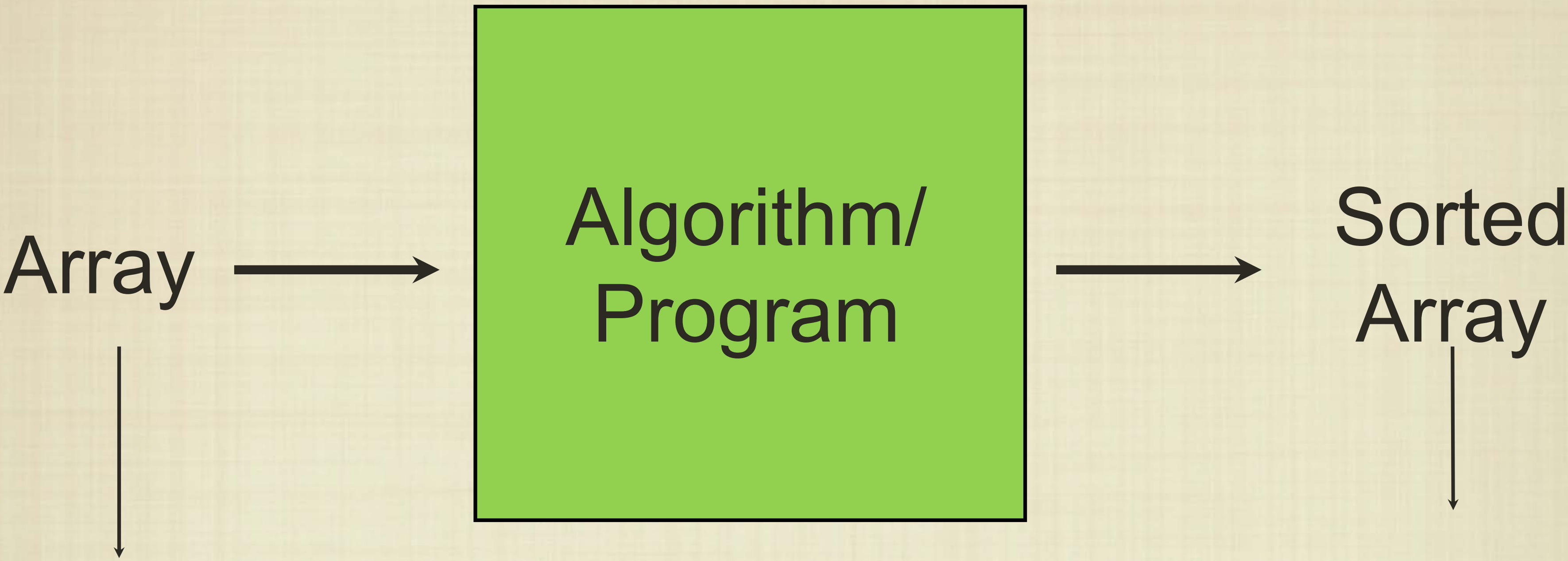
Sorting Specifications



The input is just any array of numbers and the output should be sorted.

Note that there is not always a unique output specification.

Sorting Specifications



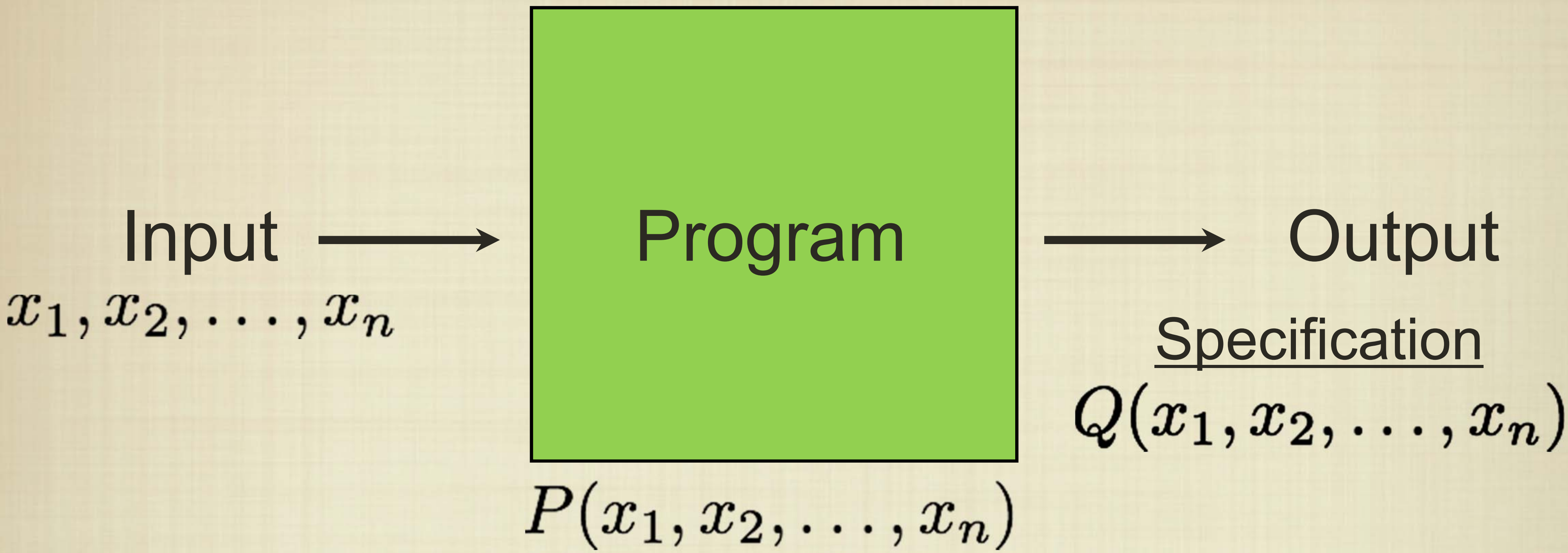
`A_in = [2,1,9, ...]`

$\forall i \in [0, \ldots, n-2]: A_{out}[i] \leq A_{out}[i+1]$

The input is just any array of numbers and the output should be sorted.

Note that there is not always a unique output specification.

Program Execution and Logic



For our purposes, we can view program execution as the application of a (complicated) logical formula to the given input.

When the output specification is guaranteed to follow from <u>any</u> execution (i.e., for all executions), we say the program is <u>correct</u>.

Program Execution and Logic

$$P(x_1, x_2, \ldots, x_n) \stackrel{?}{\rightarrow} Q(x_1, x_2, \ldots, x_n)$$

So, there is a natural connection between a logical specification for the output and the program itself (regardless of the language).

Deriving the formula for a computer program is somewhat cumbersome -- we will use other techniques to prove this implication.

What does testing a program on selected inputs prove?

Putting It All Together

- Given a problem, we should:
  - Specify the input and output in first-order logic.
  - Design an algorithm; analyze its performance.
  - Implement the program to meet specifications; analyze real-world performance.
- These steps are now mathematically precise and empirically concrete. How do we prove that an algorithm/program transforms the input into the output?

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
     int temp = A[i]; // temp = A[i]
     A[i] = A[j];
     A[j] = temp;
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
     int temp = A[i]; // temp = A[i]
     A[i] = A[j];     // (temp = A[i]) ∧ (A_out[i] = A[j])
     A[j] = temp;
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
    int temp = A[i]; // temp = A[i]
    A[i] = A[j];     // (temp = A[i]) ∧ (A_out[i] = A[j])
    A[j] = temp;     // (A_out [j] = temp = A[i]) ∧ (A_out[i] = A[j])
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A, i, j

```
public static void swap(int[] A, int i, int j){
     int temp = A[i]; // temp = A[i]
     A[i] = A[j];     // (temp = A[i]) ∧ (A_out[i] = A[j])
     A[j] = temp;     // (A_out [j] = temp = A[i]) ∧ (A_out[i] = A[j])
}
```

Output:

$$(A_{\text{out}}[i] = A[j]) \wedge (A_{\text{out}}[j] = A[i])$$

For the purposes of correctness, we must distinguish the input from the output.

Program Correctness

- What is the output specification and why is it met?

Input: A

```
public static int my_min(int[] A){
  int z=0;
  for(int i=1; i<A.length; i++){
    if(A[i]<A[z])
      z=i;
  }
  return z;
}
```

Output: $\forall j \in [0, n-1]\colon A[z] \leq A[j]$

How do we reason about loops? Can we argue that we are making progress "toward" the output specification?

Loop Invariants

```
public static int my_min(int[] A){
  int z=0;
  for(int i=1; i<A.length; i++){
     //∀j ∈ [0, i − 1]: A[z] ≤ A[j]
    if(A[i]<A[z])
      z=i;
  }
  return z;
}
```

What do we know before the beginning of each loop iteration?

$$\forall j \in [0, i-1]\colon\ A[z] \leq A[j]$$

The above statement continues to hold for every value of i : it is a loop invariant. When the loop completes we know the output specification is true.

Loop Invariants

```
public static int my_min(int[] A){
  int z=0;
  for(int i=1; i<A.length; i++){
     //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
    if(A[i]<A[z])
      z=i;
  }
  return z;
}
```

What do we know before the beginning of each loop iteration?

Invariant: $\forall j \in [0, i-1]\colon A[z] \leq A[j]$

The above statement continues to hold for every value of i: it is a loop invariant. When the loop completes we know the output specification is true.

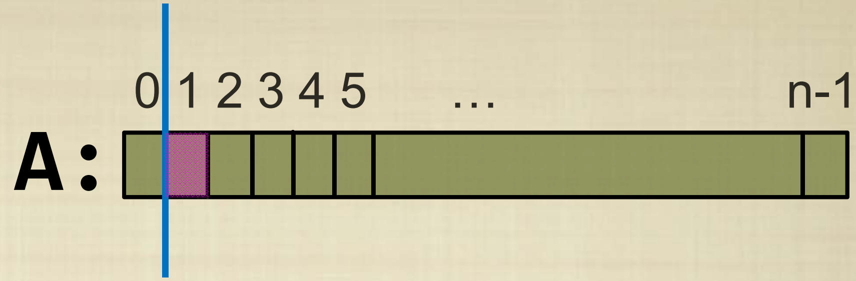
Loop Invariants

Input: A

```
public static int my_min(int[] A){
  int z=0;
  for(int i=1; i<A.length; i++){
     //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
    if(A[i]<A[z])
      z=i;
  }
  return z;
}
```

Output: $\forall j \in [0, n-1]\colon A[z] \leq A[j]$

Loop Invariants

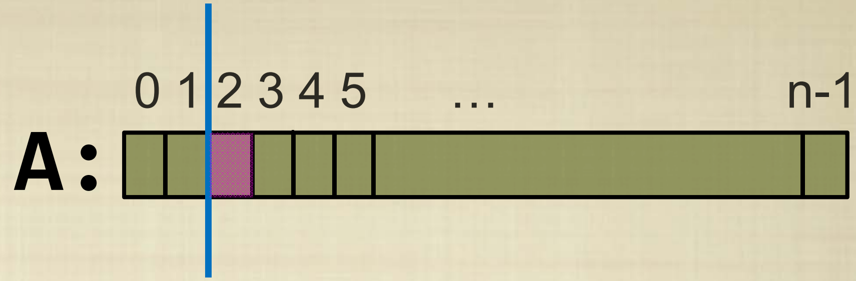


i=1

```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0, 1 − 1]: A[z] ≤ A[j]   // A[z] ≤ A[0]
  if(A[i]<A[z])
    z=i;
}
```

Ensure that A[z]≤A[1]

Loop Invariants

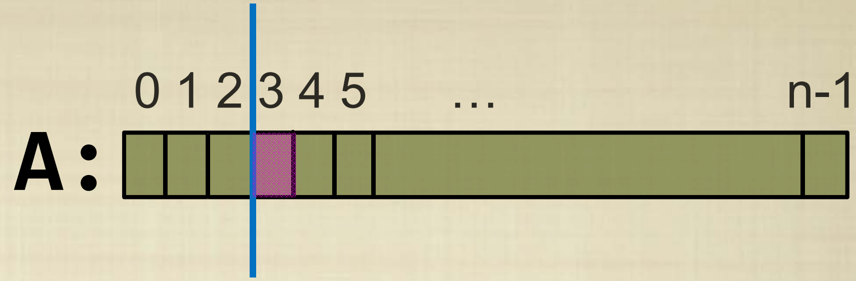


i=2

```
int z=0;
for(int i=1; i<n; i++){
   //Invariant: ∀j ∈ [0, 2 − 1]: A[z] ≤ A[j]   // A[z] ≤ A[0] ∧ A[z] ≤ A[1]
   if(A[i]<A[z])
      z=i;
}
```

Ensure that A[z]≤A[2]

Loop Invariants

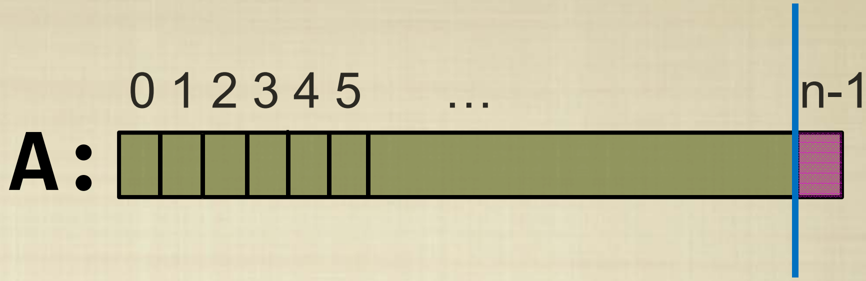


i=3

```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0, 3 − 1]: A[z] ≤ A[j]  // A[z] ≤ A[0] ∧ A[z] ≤ A[1] ∧ A[z] ≤ A[2]
  if(A[i]<A[z])
    z=i;
}
```

Ensure that A[z]≤A[3]

Loop Invariants

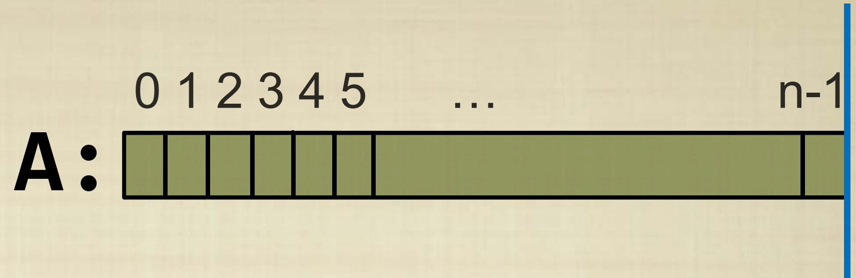


i=n-1

```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0,n-1-1]: A[z] ≤ A[j]
  if(A[i]<A[z])
    z=i;
}
```

Ensure that A[z]≤A[n-1]

Loop Invariants



i=n

```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
  if(A[i]<A[z])
    z=i;
}
//Output: ∀j ∈ [0, n − 1]: A[z] ≤ A[j]
```

Each of these proof steps used the same proof, just for a different value of i.

⟹ Use induction to prove a “generic” proof step.

Induction for Loop Invariants

- Want to prove $P(i)$ for all $i = 1 \dots n$, where
  $P(i) \equiv \ \forall j \in [0, i-1]: A[z] \leq A[j]$
- That means, we want to prove:
  $$P(1), P(2), P(3), P(4), \dots, P(n)$$

- Proceed as follows:

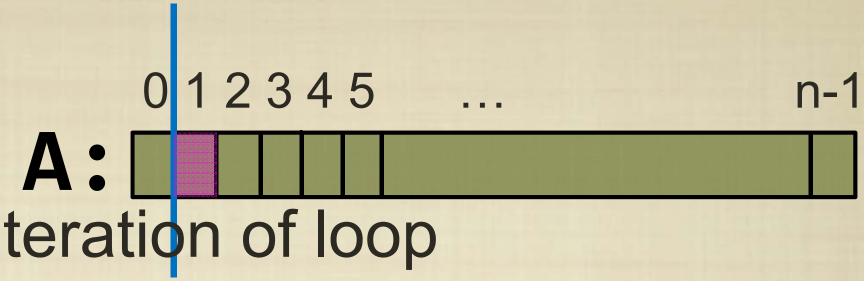
1. Base case (initialization before first iteration of loop)
   - Prove $P(1)$

2. Step (one iteration of the loop)
   - Prove $P(i) \rightarrow P(i+1)$

   (In other words, prove $P(i+1)$, but you are allowed to assume that $P(i)$ has already been proven.)

3. Termination
   - $P(n)$ is true. $P(n)$ can be used to obtain the desired output specification.

Loop Invariants



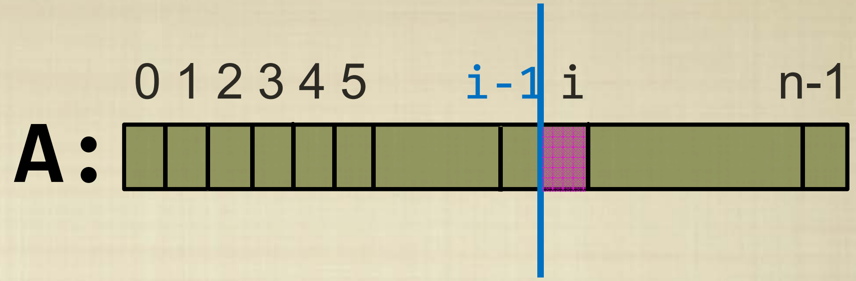
1. Base case (initialization before first iteration of loop

i=1

```
int z=0;
for(int i=1; i<n; i++){
   //Invariant: ∀j ∈ [0, 1 − 1]: A[z] ≤ A[j]   // A[z] ≤ A[0]
   if(A[i]<A[z])
      z=i;
}
```

Just need to prove: A[z] ≤ A[0]

Loop Invariants



2. Step (one iteration of the loop)

i

```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
  if(A[i]<A[z])
    z=i;
}
```

$\equiv P(i)$

≡ A[z] ≤ A[0] ∧ … ∧ A[z] ≤ A[i-1]

Ensure that A[z]≤A[i]

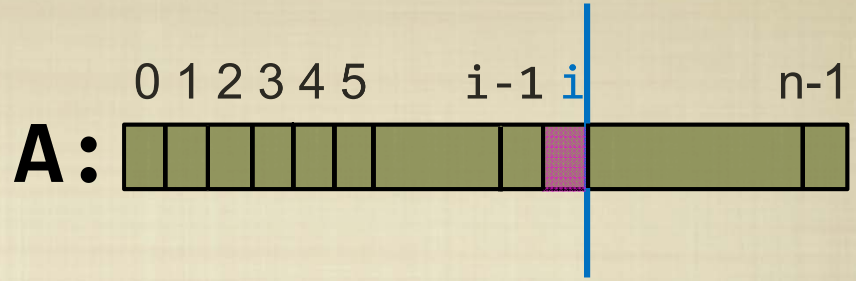
Assume is true

Prove to be true

A[z] ≤ A[0] ∧ … ∧ A[z] ≤ A[i-1] ∧ A[z] ≤ A[i]

$\equiv P(i+1)$

Loop Invariants



2. Step (one iteration of the loop)

i+1

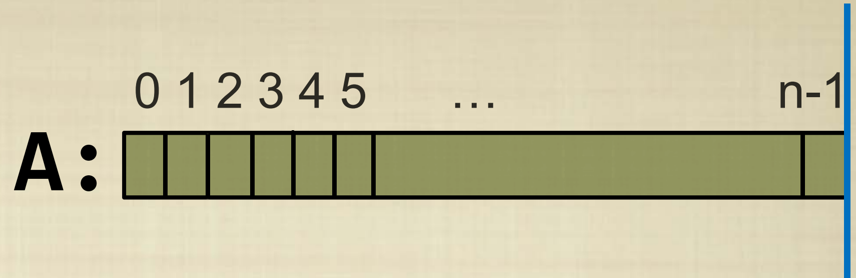
```
int z=0;
for(int i=1; i<n; i++){
  //Invariant: ∀j ∈ [0,i+1- 1]: A[z] ≤ A[j]
  if(A[i]<A[z])
    z=i;
}
```

$\equiv P(i+1)$

≡ A[z]≤A[0] ∧…∧ A[z]≤A[i-1] ∧ A[z]≤A[i]

Has now been proven to be true.

Loop Invariants



3. Termination

i=n

```
int z=0;
for(int i=1; i<n; i++){
   //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
   if(A[i]<A[z])
      z=i;
}
//Output: ∀j ∈ [0,n − 1]: A[z] ≤ A[j]
```

$\equiv P(n)$

Has now been proven to be true.

my_min

```
//Input: A
public static int my_min(int[] A){
  int z=0;
  for(int i=1; i<A.length; i++){
     //Invariant: ∀j ∈ [0, i − 1]: A[z] ≤ A[j]
    if(A[i]<A[z])
      z=i;
  }
  return z;
}
//Output: ∀j ∈ [0, A.length − 1]: A[z] ≤ A[j]
```

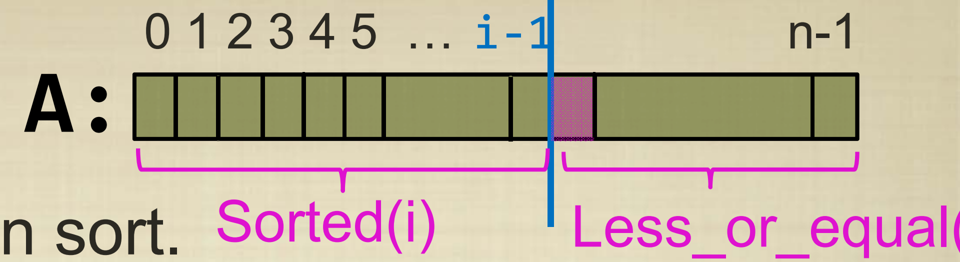
Computes the minimum of A[0...A.length-1]

my_min variant

```
//Input: A, left
public static int my_min(int[] A, int left){
   int z=left;
   for(int i=left+1; i<A.length; i++){
       //Invariant:∀j ∈ [left, i − 1]: A[z] ≤ A[j]
      if(A[i]<A[z])
         z=i;
   }
   return z;
}
//Output:∀j ∈ [left, A.length − 1]: A[z] ≤ A[j]
```

Computes the minimum of A[left...A.length-1]

Selection Sort



- Let's find an invariant for selection sort.

```java
// Input: A
public static void selectionSort(int[] A){
  for(int i=0; i<A.length; i++){
     // Invariant
    int j = my_min(A,i);
    swap(A,i,j);
  }
}
```

n

Handle A[i]

<u>Loop Invariant (at the beginning of each iteration):</u>

Sorted(i)

(1) A[0..i-1] are sorted (in increasing order), and

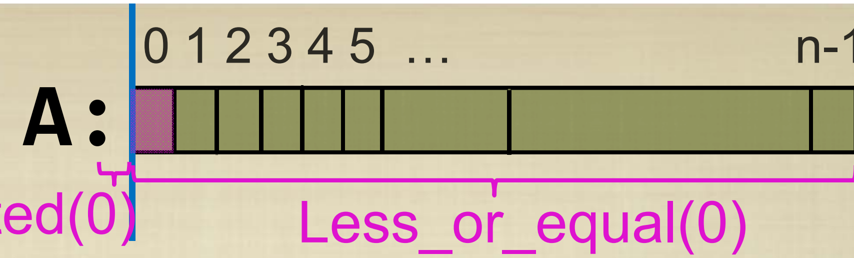
$$\forall k \in [0, i-2]: A_{out}[k] \leq A_{out}[k+1]$$

Less_or_equal(i)

(2) All A[0..i-1] are less or equal to all A[i..n-1]

$$\forall k \in [i, n-1]: A_{out}[i-1] \leq A_{out}[k]$$

Selection Sort



1. Base case (initialization before first iteration of loop

i=0

```
// Input: A
public static void selectionSort(int[] A){
  for(int i=0; i<A.length; i++){
      // Invariant
    int j = my_min(A,i);
    swap(A,i,j);
  }
}
```

$\underbrace{\texttt{A.length}}_{n}$

Handle A[i]

Loop Invariant (at the beginning of each iteration):

Sorted(0)

(1) A[0.. 0-1] are sorted (in increasing order), and

$$\forall k \in [0, 0-2]: A_{out}[k] \le A_{out}[k+1]$$

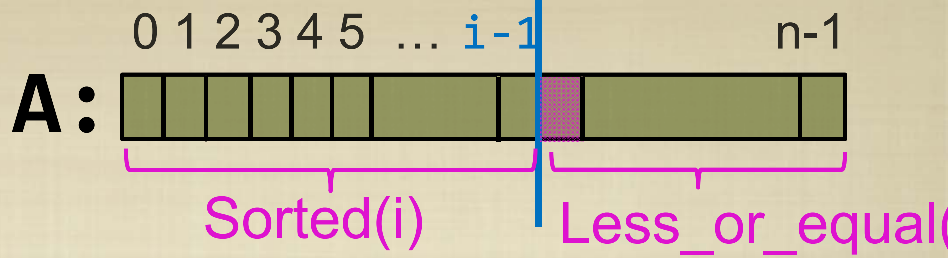
Less_or_equal(0)

(2) All A[0.. 0-1] are less or equal to all A[0 ..n-1]

$$\forall k \in [0, n-1]: A_{out}[0-1] \le A_{out}[k]$$

⇒ Trivially true because Sorted portion is empty.

Selection Sort



2. Step (one iteration of the loop)

a) Assume invariant is true for i (i.e., before one iteration)

i

```
// Input: A
public static void selectionSort(int[] A){
  for(int i=0; i<A.length; i++){
     // Invariant
    int j = my_min(A,i);
    swap(A,i,j);
  }
}
```

n

Handle A[i]

Loop Invariant (at the beginning of each iteration):

Sorted(i)

(1) A[0.. i-1] are sorted (in increasing order), and

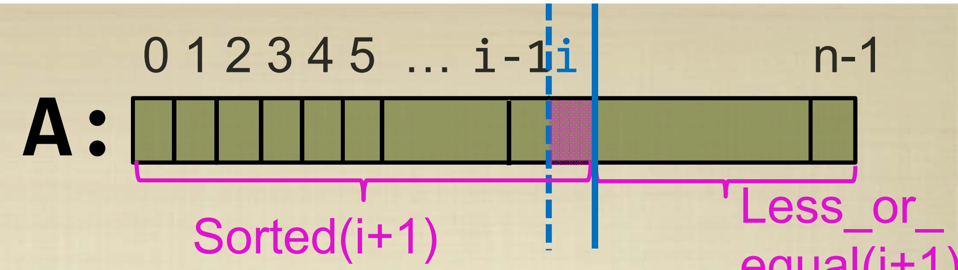
$$\forall k \in [0, i-2]: A_{out}[k] \le A_{out}[k+1]$$

Less_or_equal(i)

(2) All A[0.. i-1] are less or equal to all A[i ..n-1]

$$\forall k \in [i, n-1]: A_{out}[i-1] \le A_{out}[k]$$

Selection Sort



2. Step (one iteration of the loop)

b) Now prove that invariant is true for **i+1** (i.e., after the iteration)

i+1

```
// Input: A
public static void selectionSort(int[] A){
  for(int i=0; i<A.length; i++){
     // Invariant
     int j = my_min(A,i);
     swap(A,i,j);
  }
}
```

n

Loop Invariant (at the beginning of each iteration):

(1) A[0.. **i+1** -1] are sorted (in increasing order), and

Sorted(i+1)

$$\forall k \in [0, \texttt{i+1} - 2]: A_{out}[k] \leq A_{out}[k+1]$$

$\Rightarrow$ True because A[i] is in A[i..n-1], and Less_or_equal(i) is true

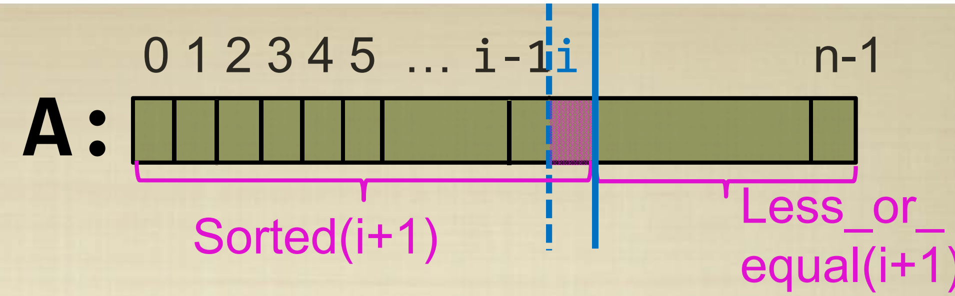
Less_or_equal(i+1)

(2) All A[0.. **i+1** -1] are less or equal to all A[**i+1** ..n-1]

$$\forall k \in [\texttt{i+1}, n-1]: A_{out}[\texttt{i+1} - 1] \leq A_{out}[k]$$

$\Rightarrow$ True because A[i] is min, and Less_or_equal(i) is true

Selection Sort



3. Termination

i=n

```
// Input: A
public static void selectionSort(int[] A){
  for(int i=0; i<A.length; i++){
     // Invariant
    int j = my_min(A,i);
    swap(A,i,j);
  }
}
// Output:
```

n

Loop Invariant:

Sorted(i)

(1) A[0.. n-1] are sorted (in increasing order), and

$$\forall k \in [0, n-2]: A_{out}[k] \le A_{out}[k+1]$$

Output Specification

Less_or_equal(i)

(2) All A[0.. n-1] are less or equal to all A[n ..n-1]

$$\forall k \in [n, n-1]: A_{out}[n-1] \le A_{out}[k]$$