CP312
Algorithm Design and Analysis I

LECTURE 2: ANALYZING ALGORITHMS

Recall: The Sorting Problem

- **Problem:** Sort a sequence of numbers in non-decreasing order
- **Input:** A sequence of numbers $\pi = (a_1, \ldots, a_n)$
- **Output:** A permutation $\pi' = (a_1', \ldots, a_n')$ of π such that

$$a_1' \leq a_2' \leq \cdots \leq a_n'$$

- An algorithm for the sorting problem is a sequence of computational steps with the above input/output specifications.

- Ex: $(8, 10, 1, 6, 2, 7, 3) \Rightarrow (1, 2,3,6,7,8,10)$

Insertion Sort

A	8	10	1	6	2	7	3

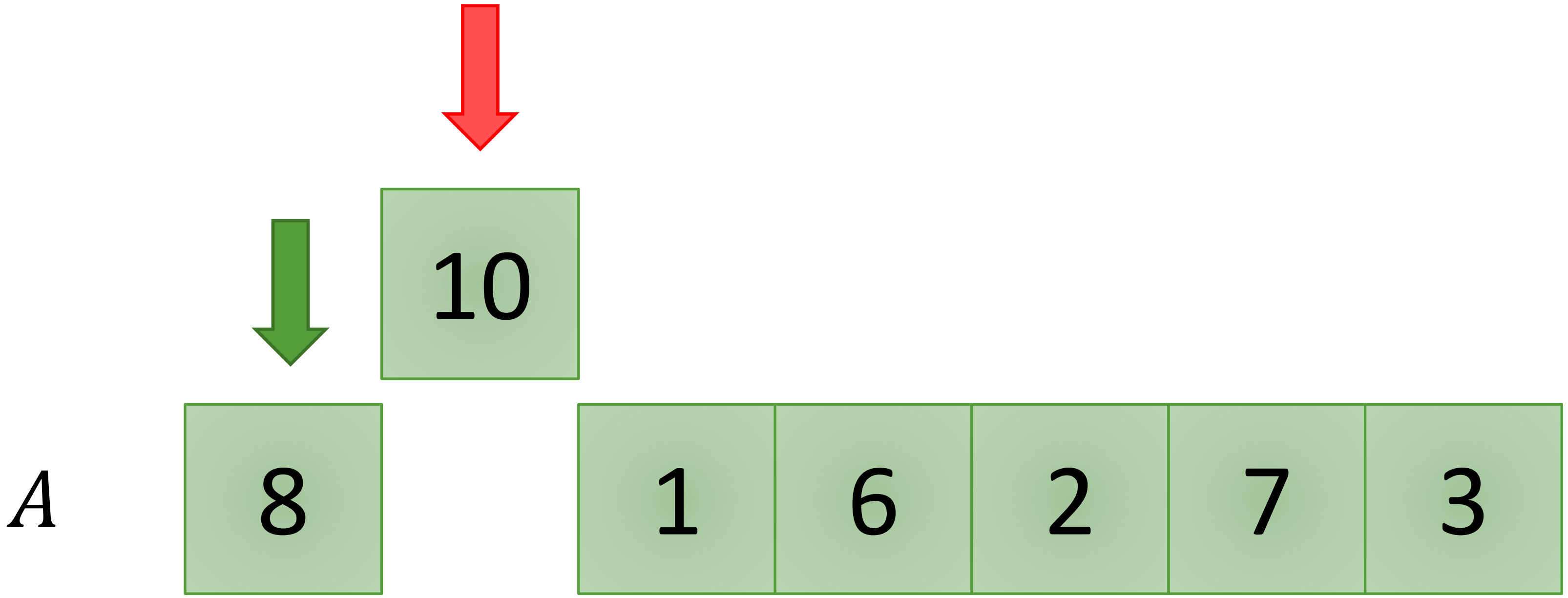
Insertion Sort

A	8	10	1	6	2	7	3

Insertion Sort

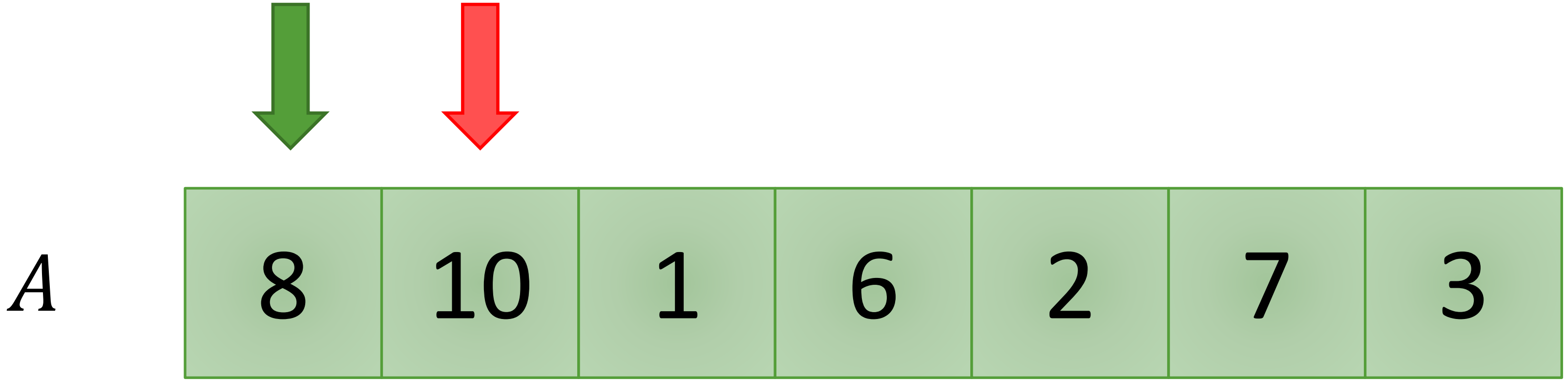
A	8	10	1	6	2	7	3

Insertion Sort

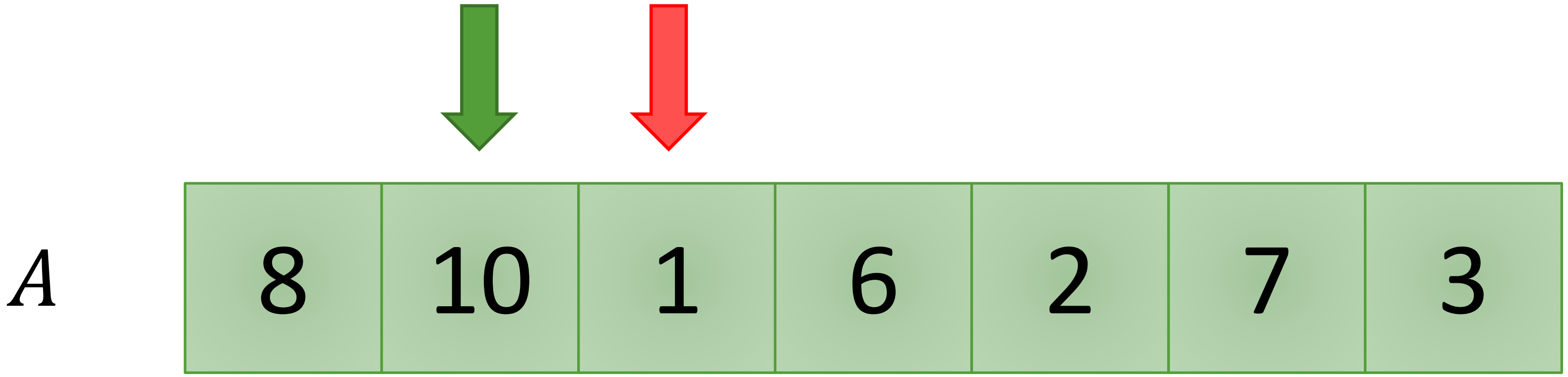


$$8 < 10?$$

Insertion Sort

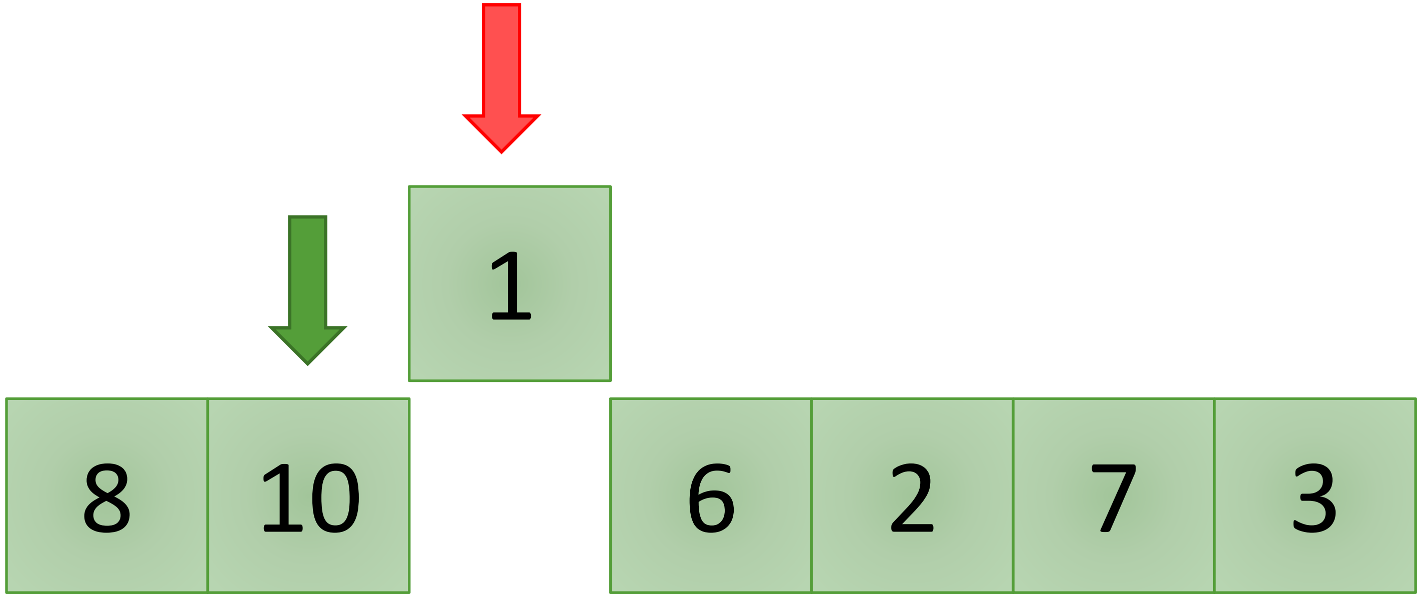


Insertion Sort



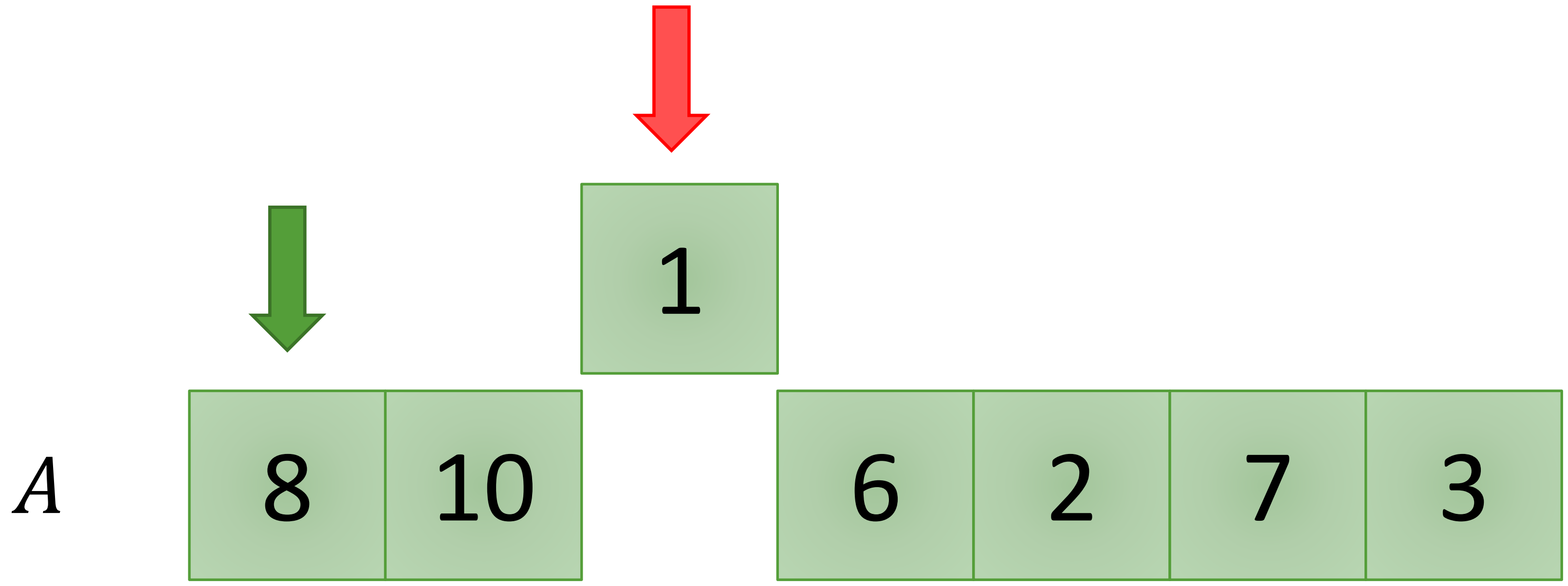
Insertion Sort

A



$$10 < 1?$$

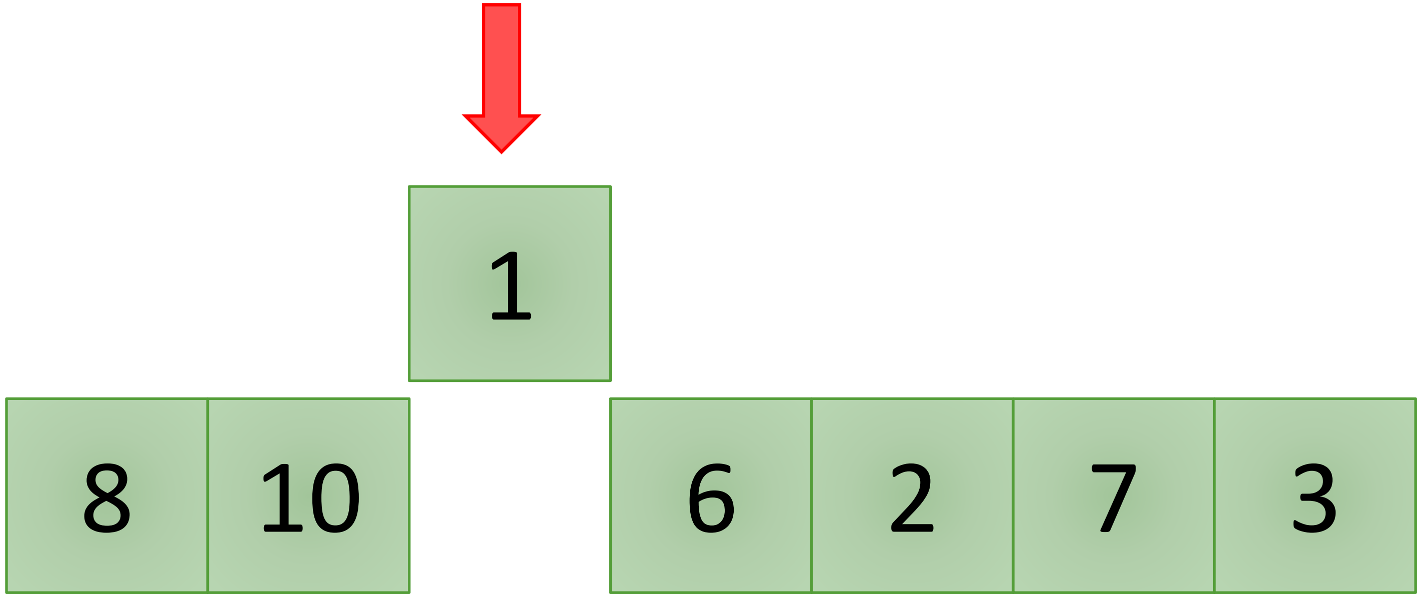
Insertion Sort



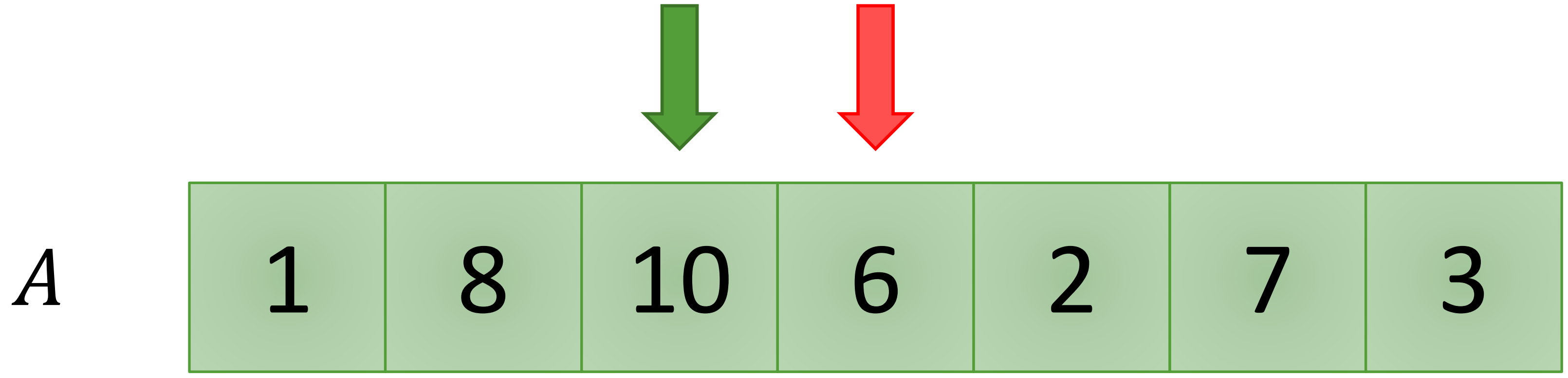
$$8 < 1?$$

Insertion Sort

A

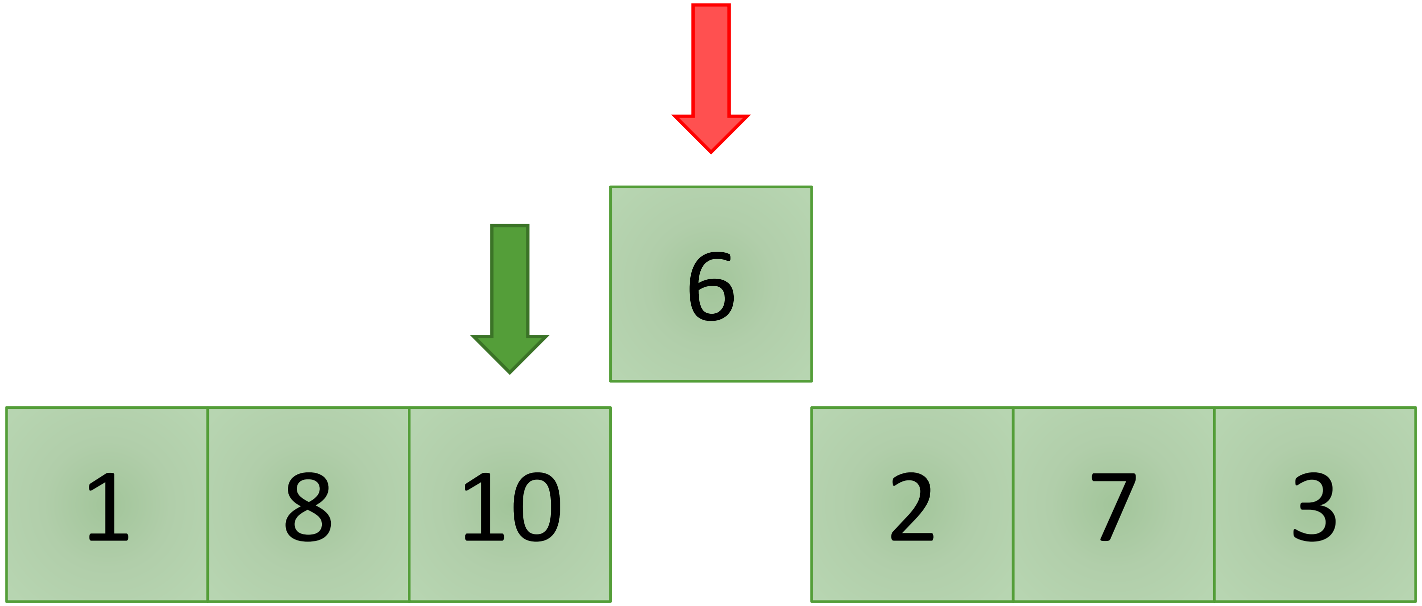


Insertion Sort



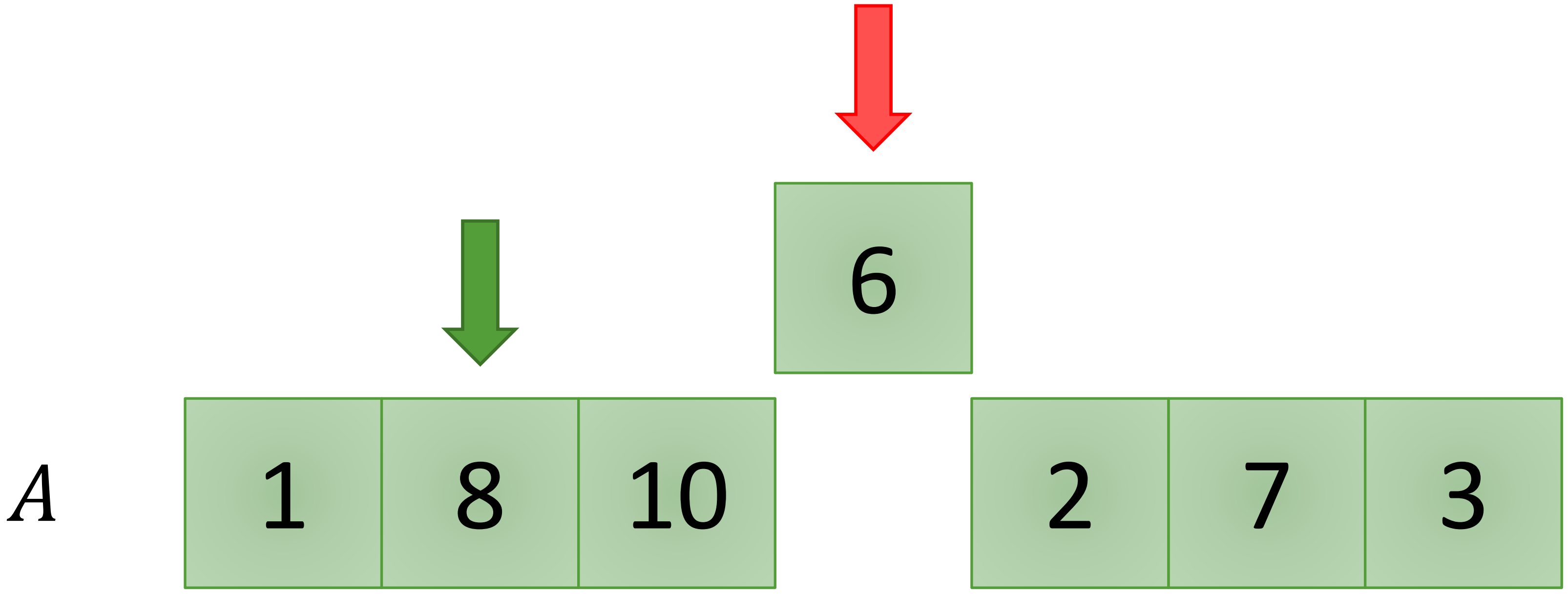
Insertion Sort

A



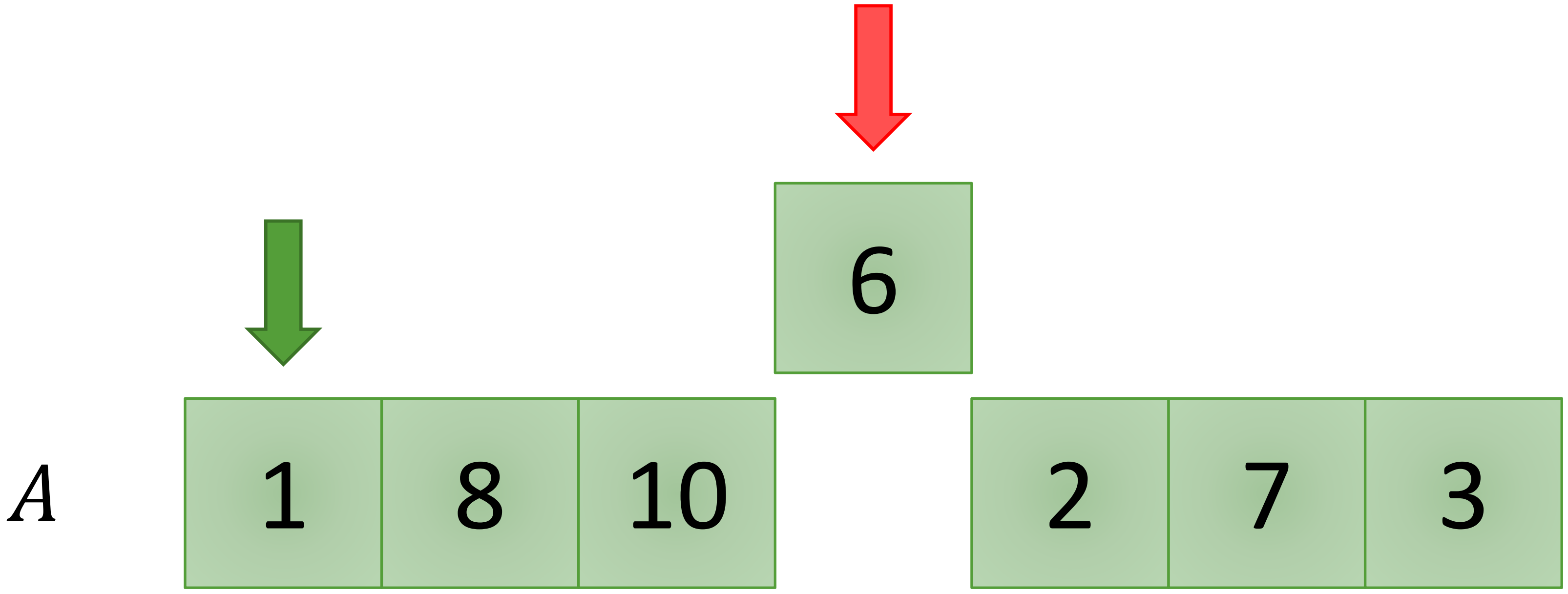
$$10 < 6?$$

Insertion Sort



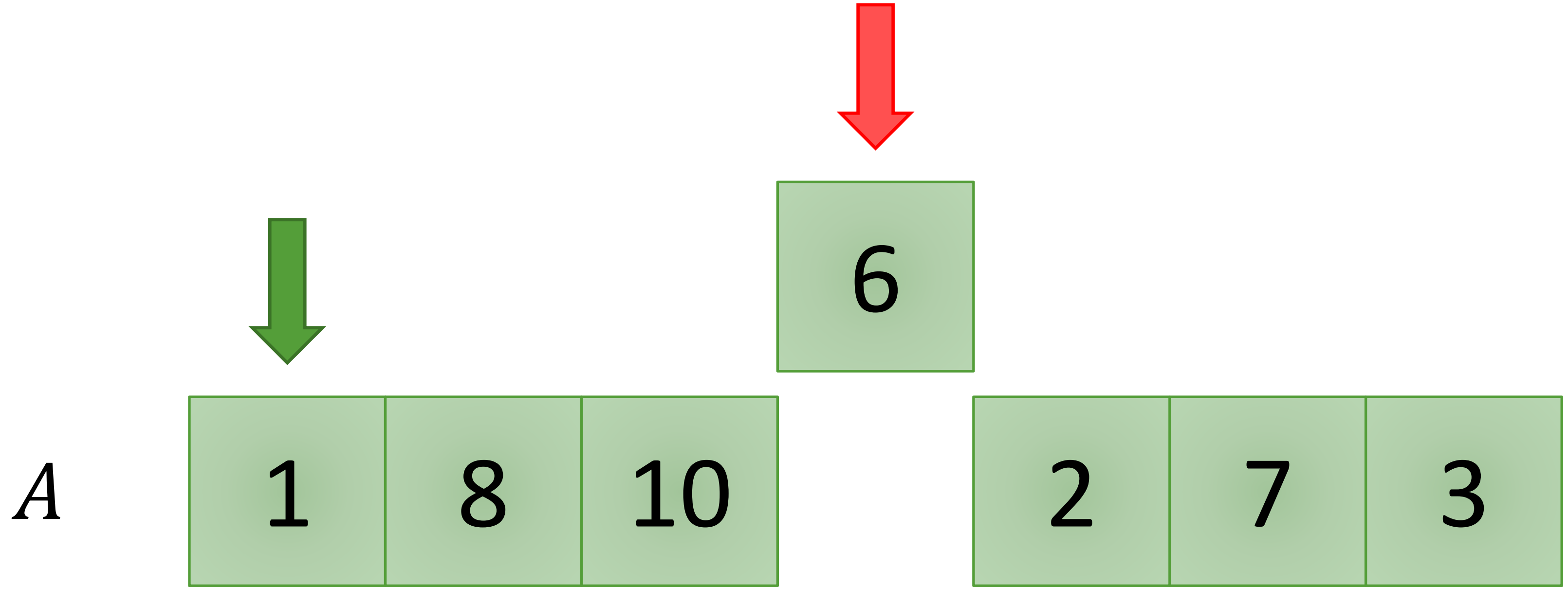
$$8 < 6?$$

Insertion Sort



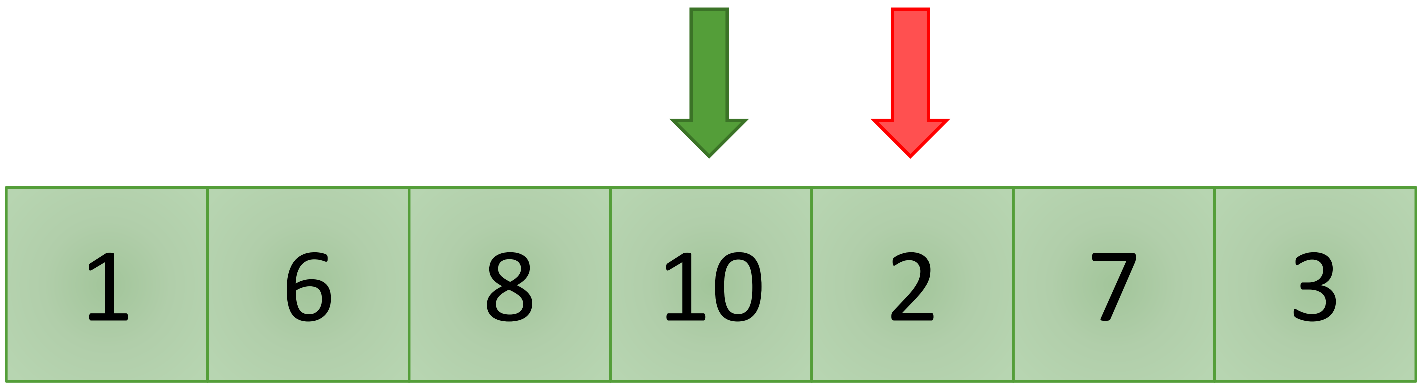
$$1 < 6?$$

Insertion Sort



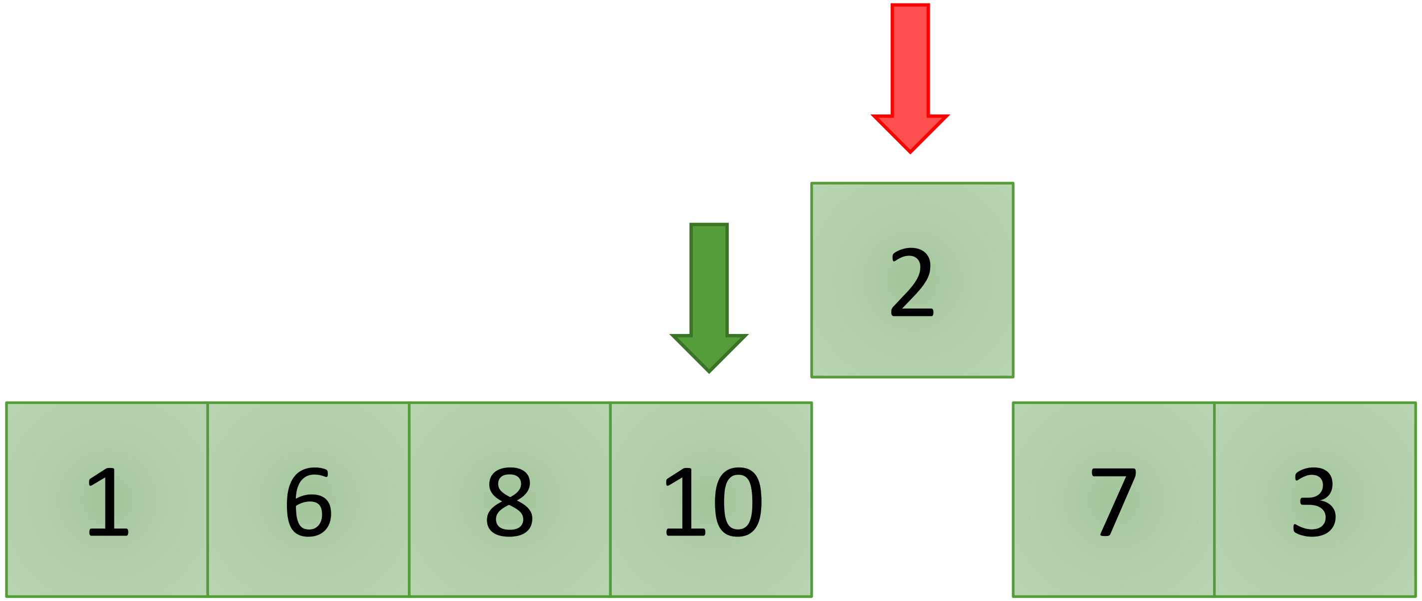
Insertion Sort

A



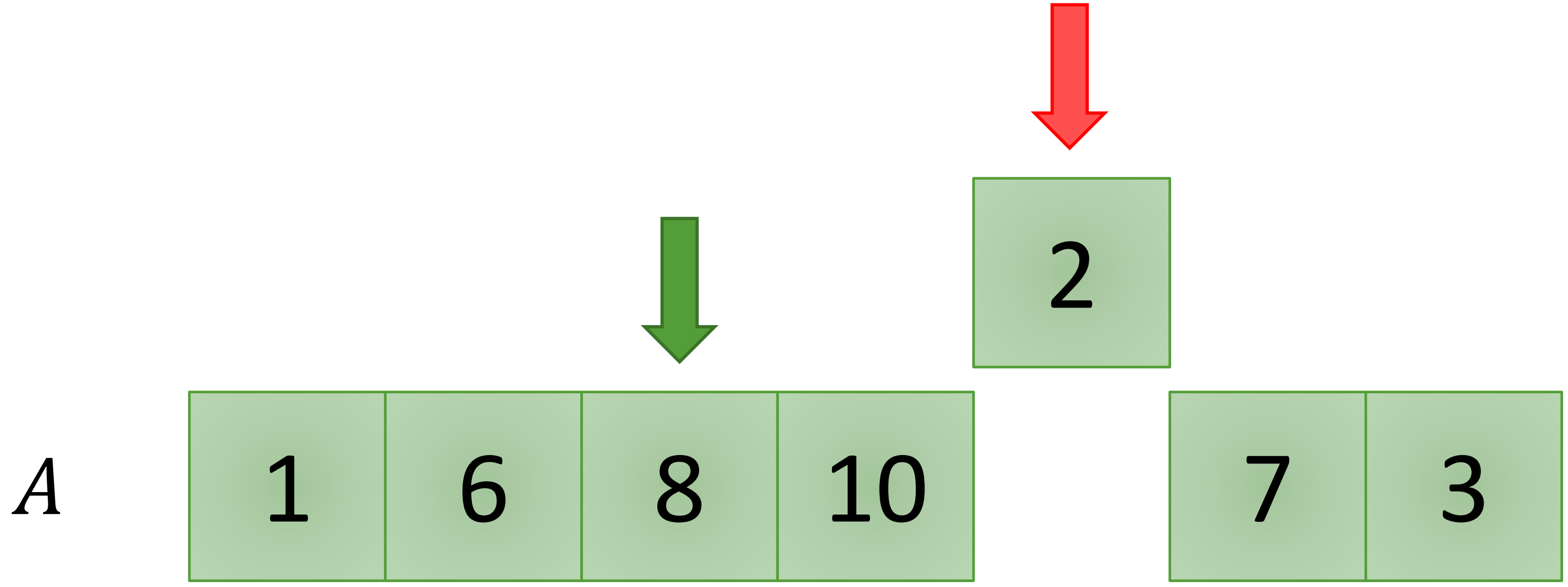
Insertion Sort

A



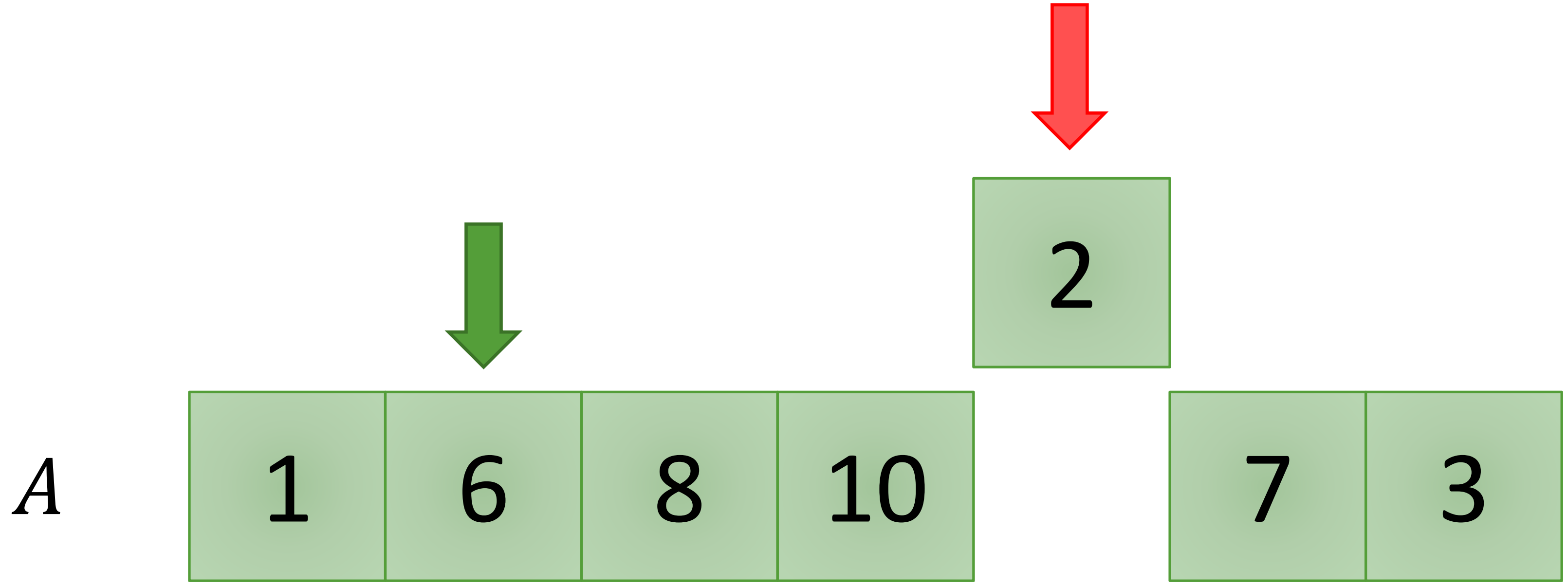
$10 < 2?$

Insertion Sort



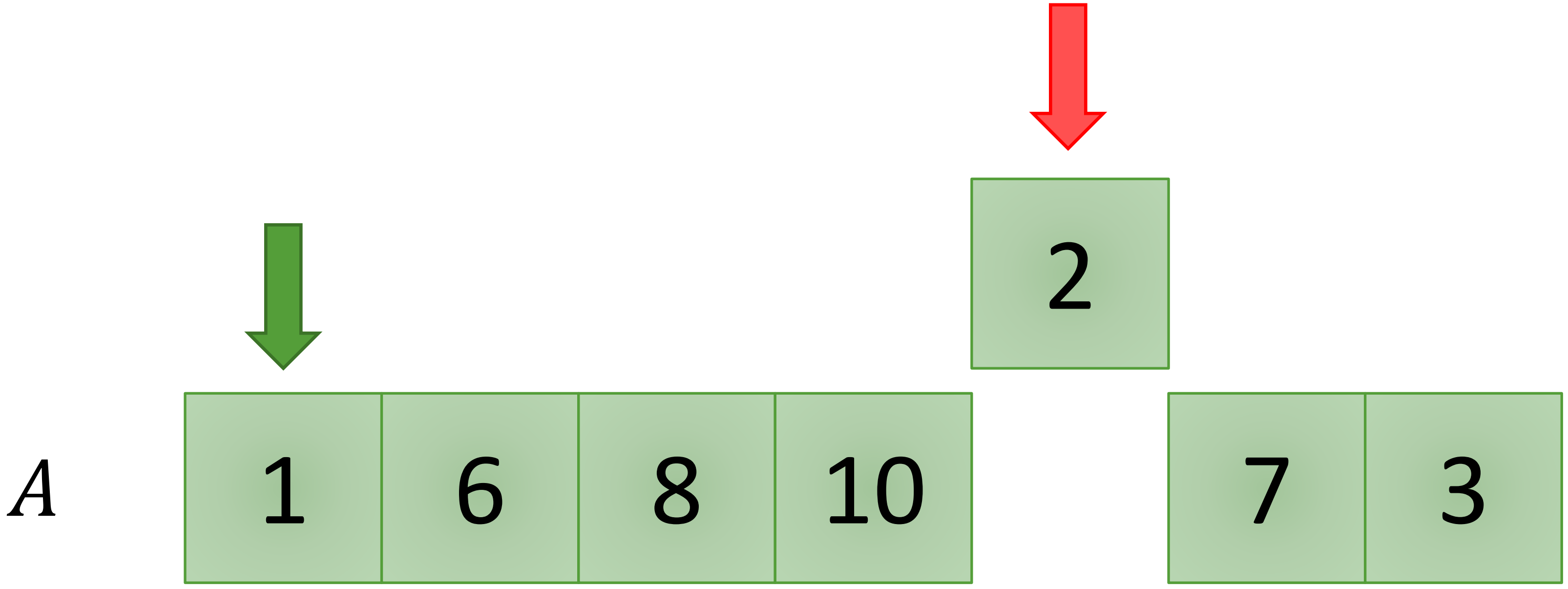
$$8 < 2?$$

Insertion Sort



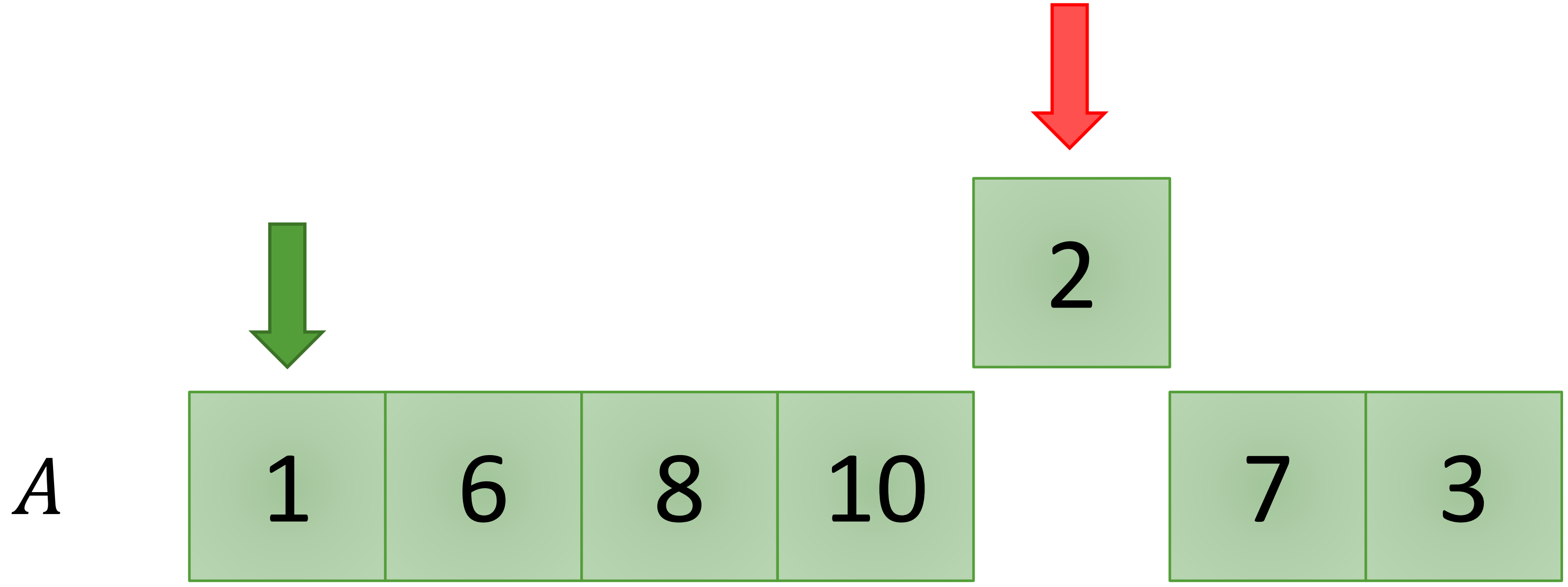
$$6 < 2?$$

Insertion Sort



$1 < 2?$

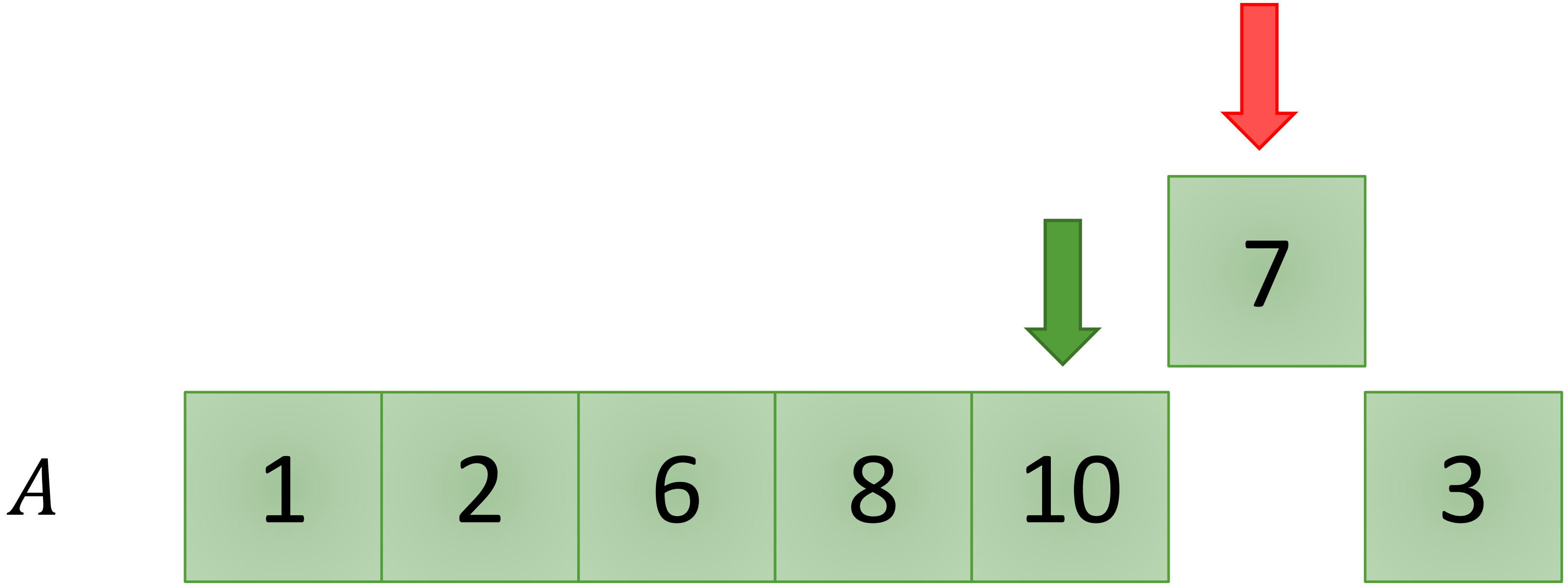
Insertion Sort



Insertion Sort

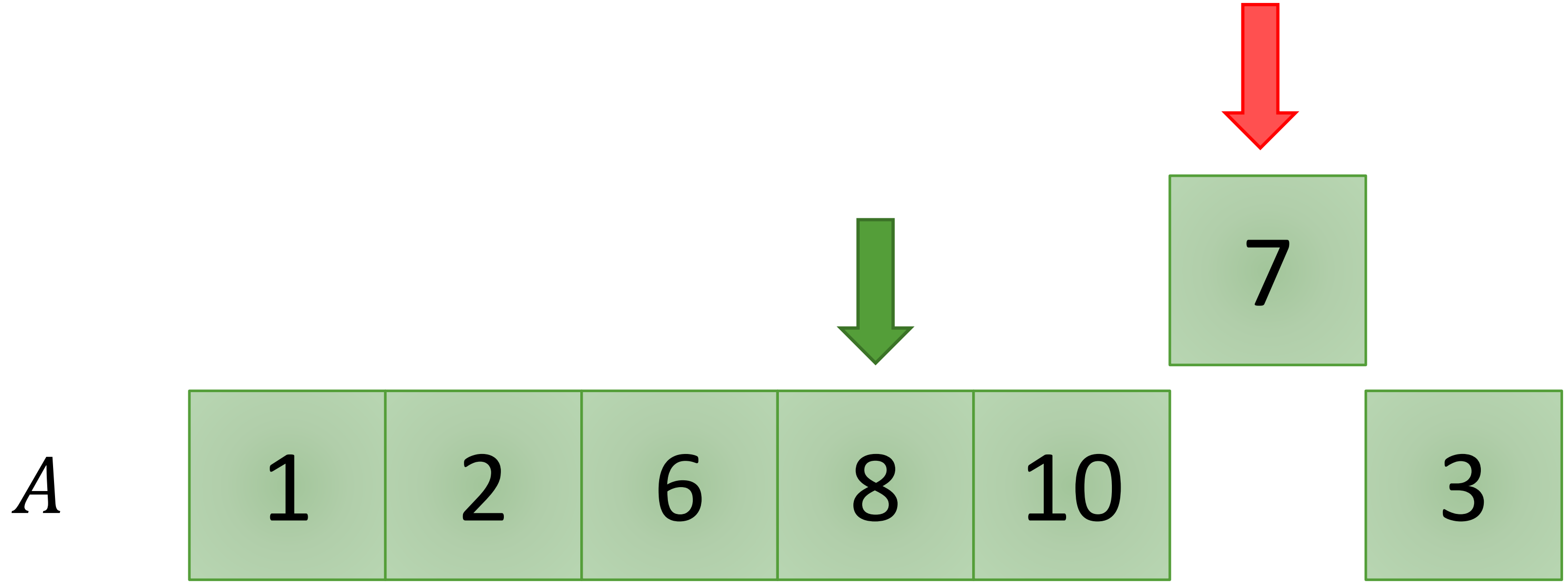
A	1	2	6	8	10	7	3

Insertion Sort



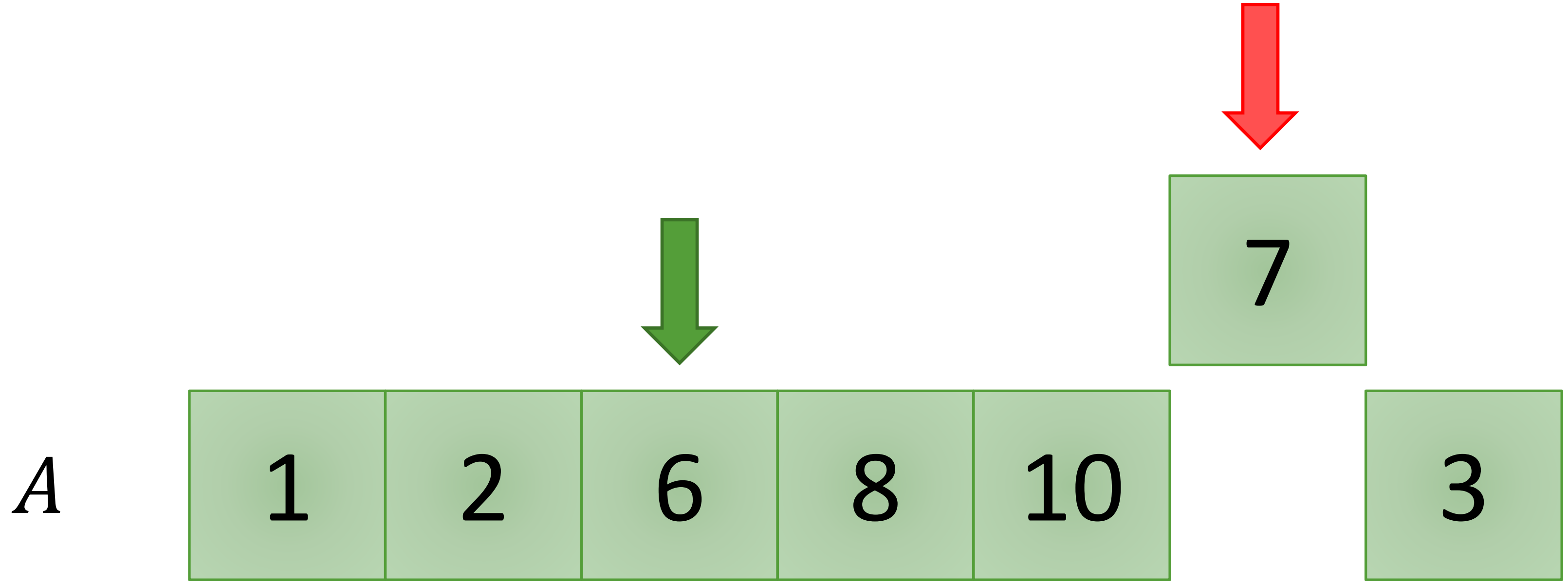
$$10 < 7?$$

Insertion Sort



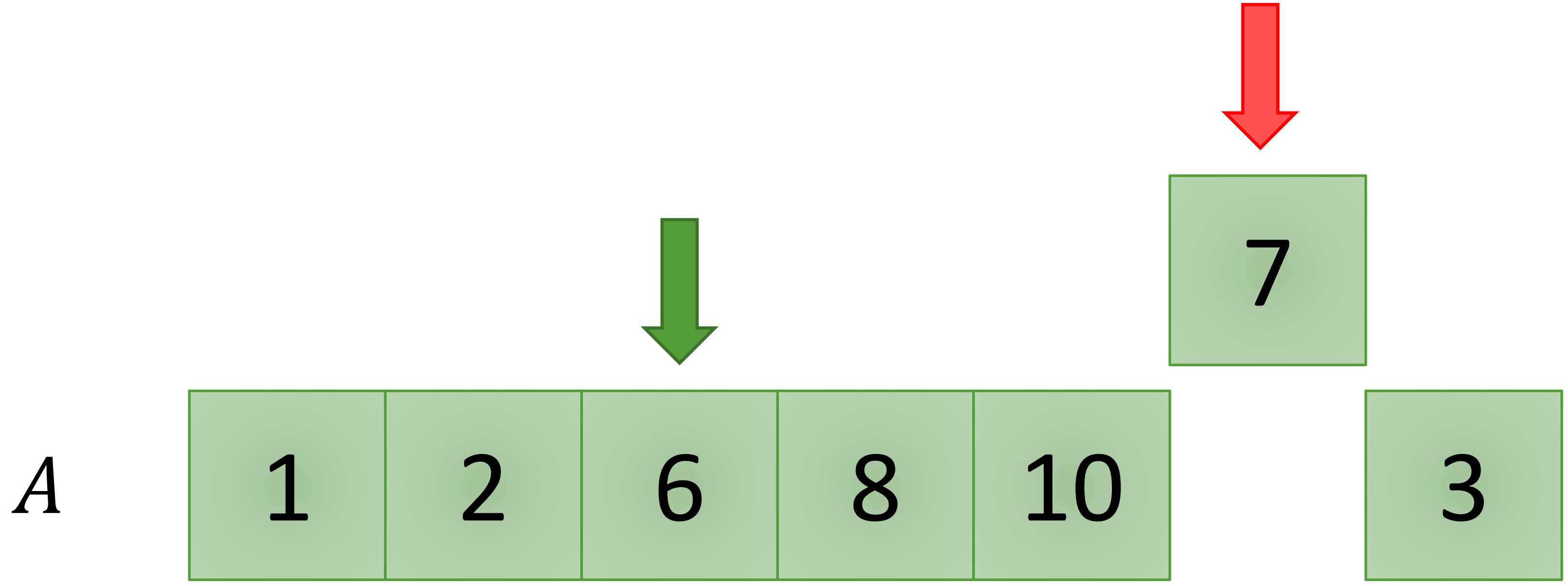
$8 < 7?$

Insertion Sort



$6 < 7?$

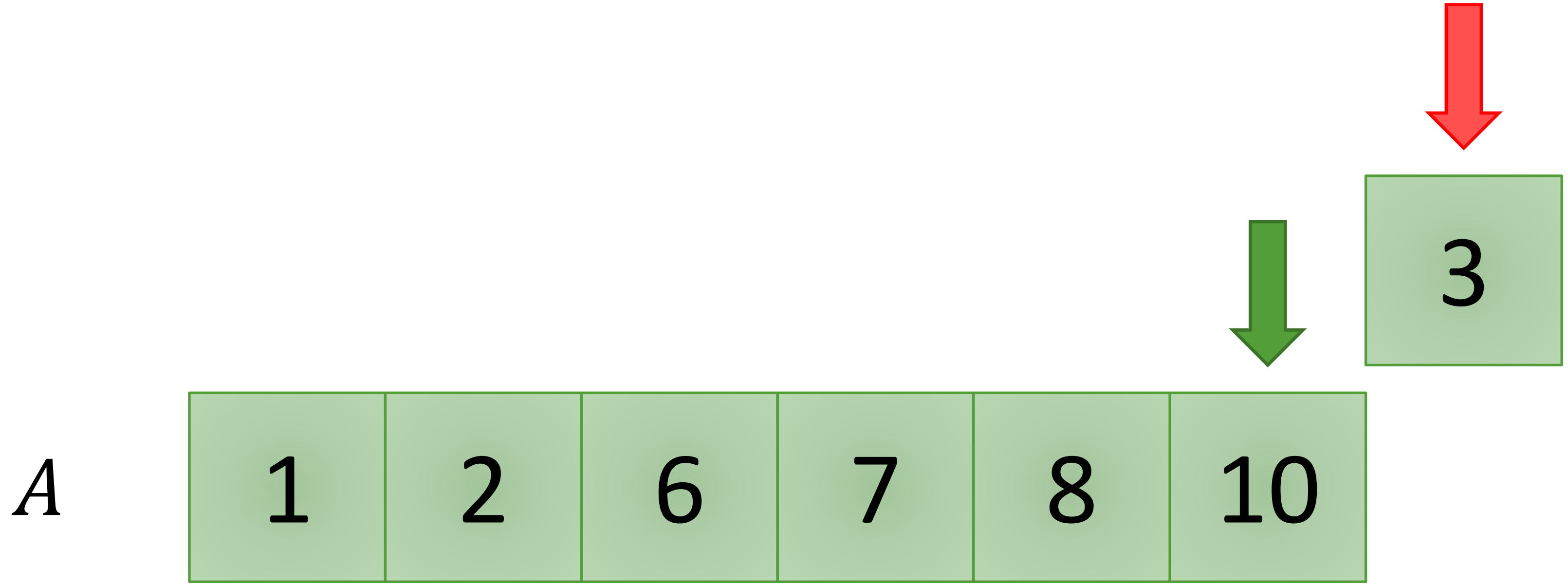
Insertion Sort



Insertion Sort

A	1	2	6	7	8	10	3

Insertion Sort



$10 < 3?$

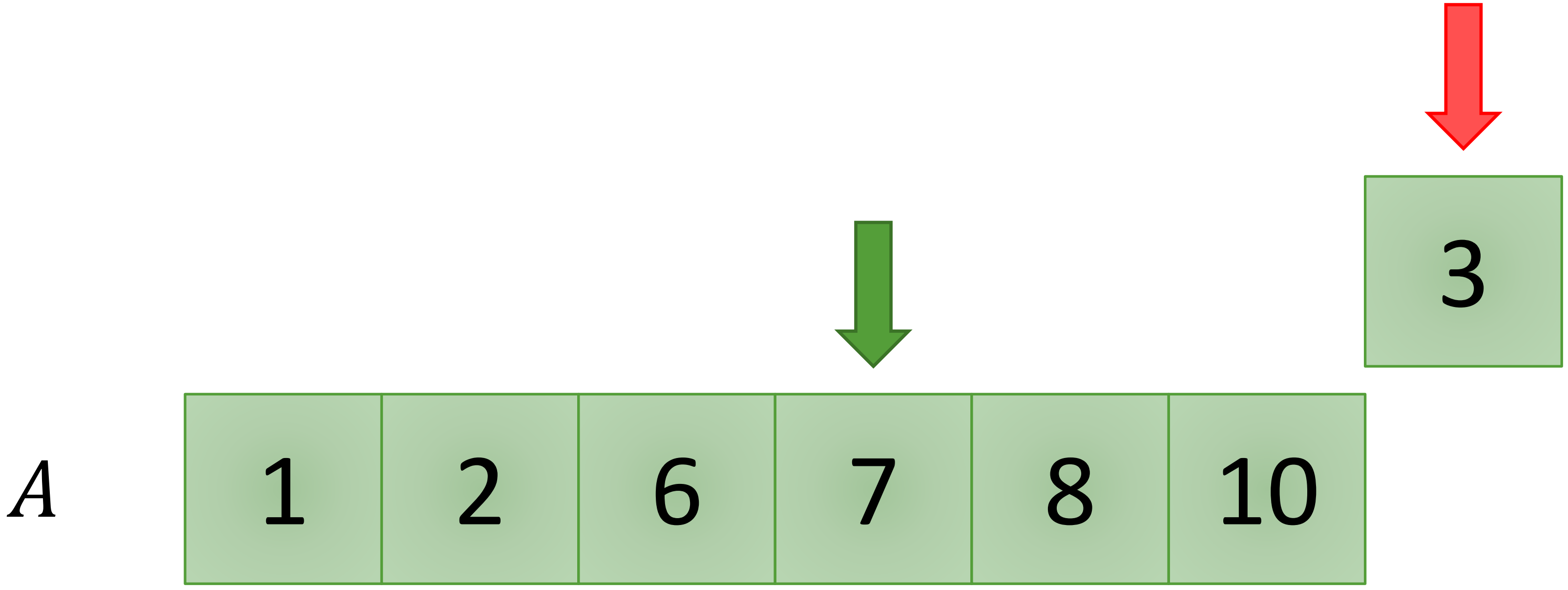
Insertion Sort

3

A	1	2	6	7	8	10

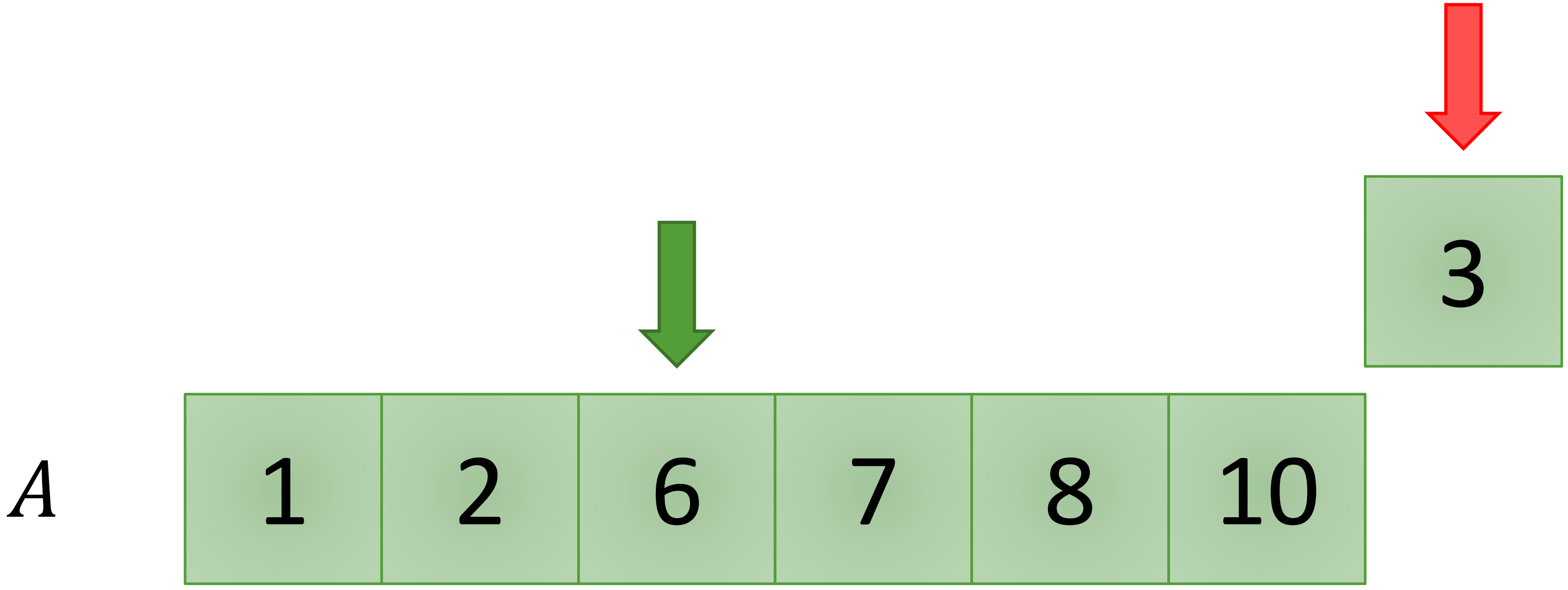
$8 < 3?$

Insertion Sort



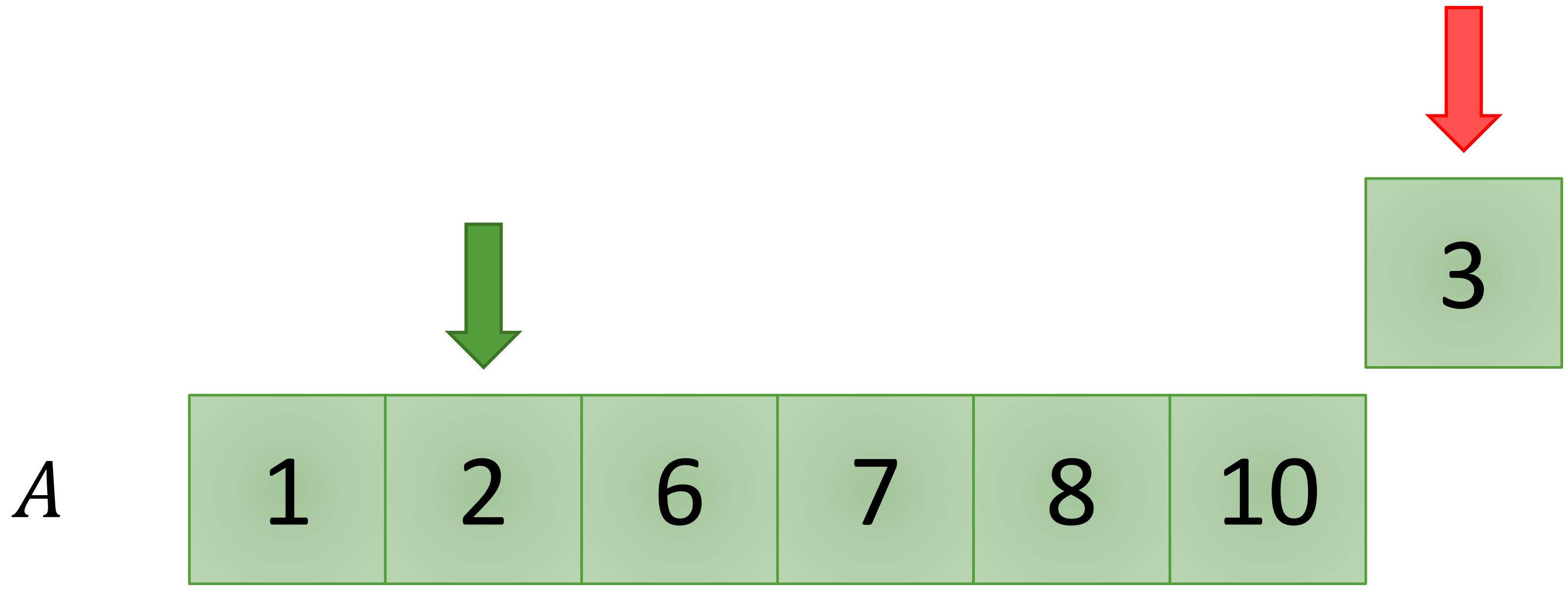
$7 < 3?$

Insertion Sort



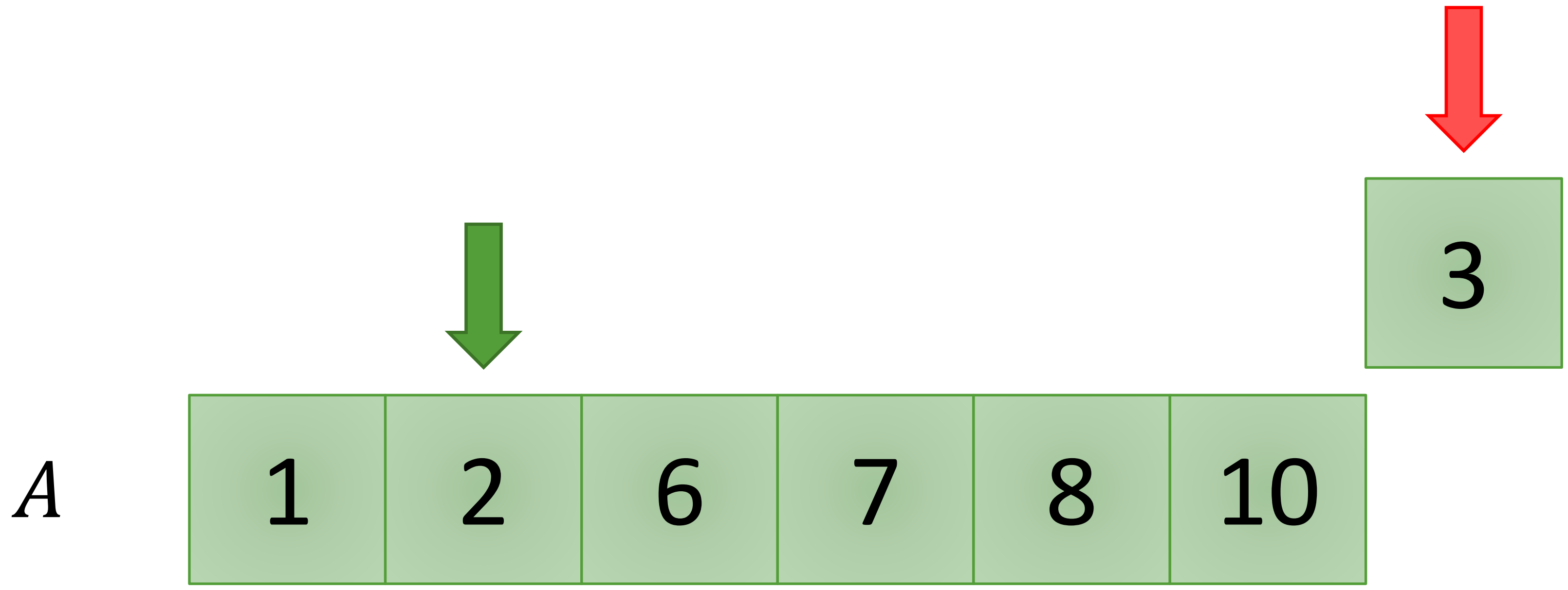
$6 < 3?$

Insertion Sort



$2 < 3?$

Insertion Sort

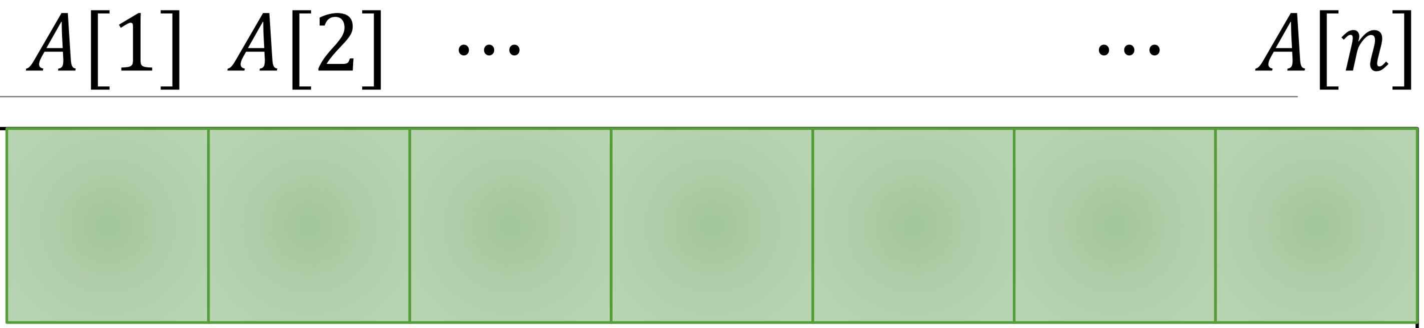


Insertion Sort

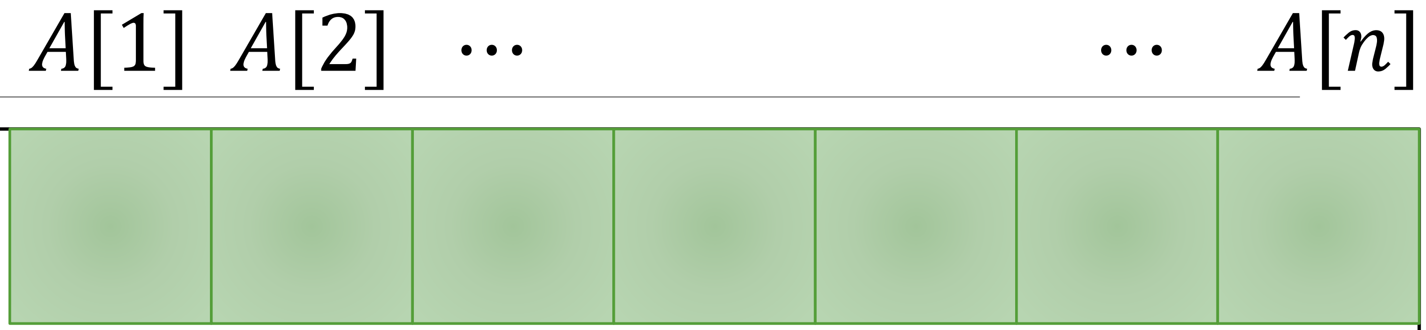
A	1	2	3	6	7	8	10

Pseudocode

Insertion_Sort(A, n):



Pseudocode



Insertion_Sort(A, n):

for $j = 2$ **to** n

    $temp = A[j]$

    $i = j - 1$

    while $i \geq 1$ and $A[i] > temp$

        $A[i + 1] = A[i]$

        $i = i - 1$

    $A[i + 1] = temp$

Pseudocode

Insertion_Sort(A, n):

for $j = 2$ **to** n

$temp = A[j]$

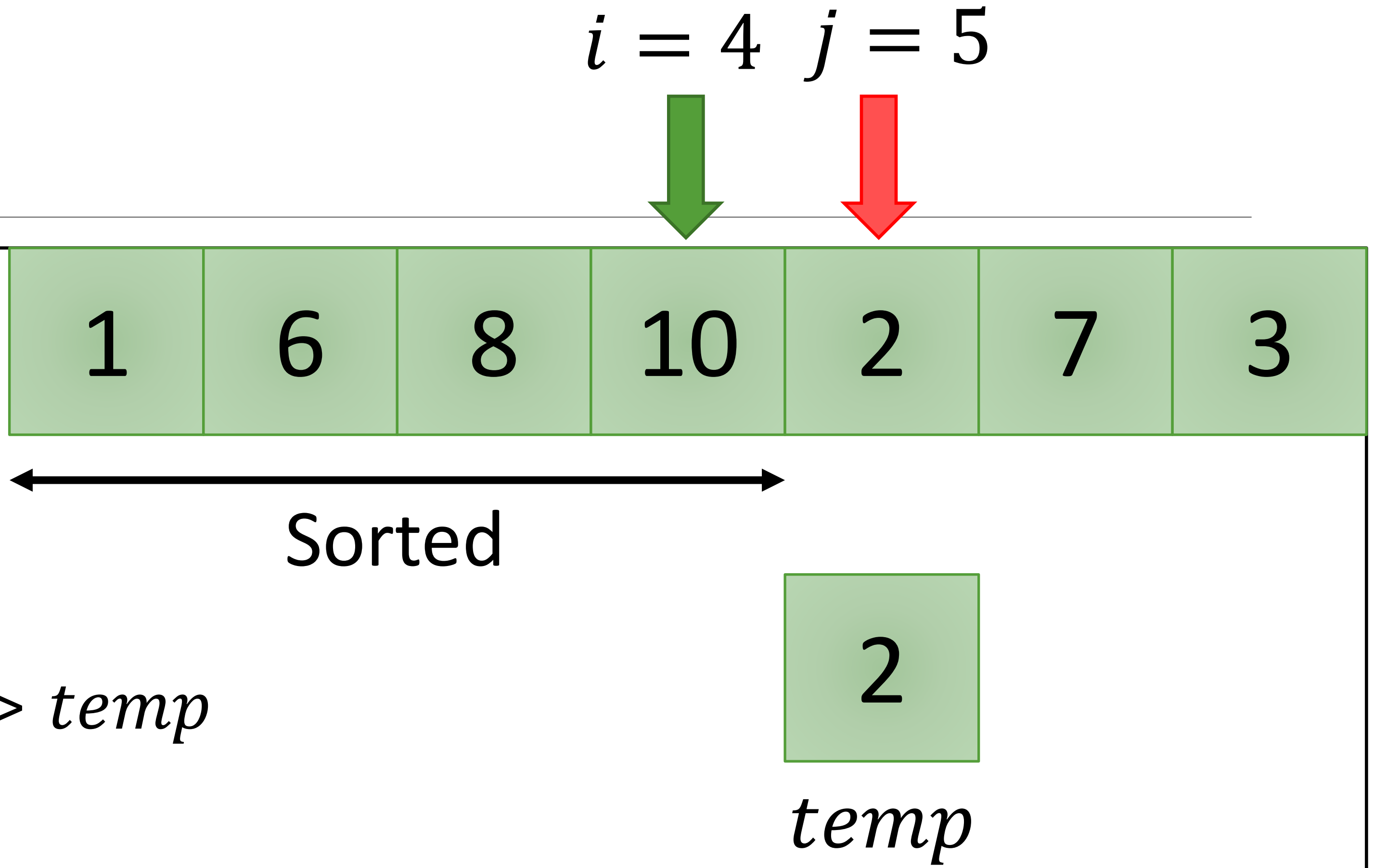
$i = j - 1$

while $i \geq 1$ and $A[i] > temp$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = temp$



Pseudocode

$i = 4$ $j = 5$

1	6	8	10	10	7	3

Insertion_Sort(A, n):

for $j = 2$ **to** n

$temp = A[j]$

$i = j - 1$

while $i \geq 1$ and $A[i] > temp$

$A[i + 1] = A[i]$

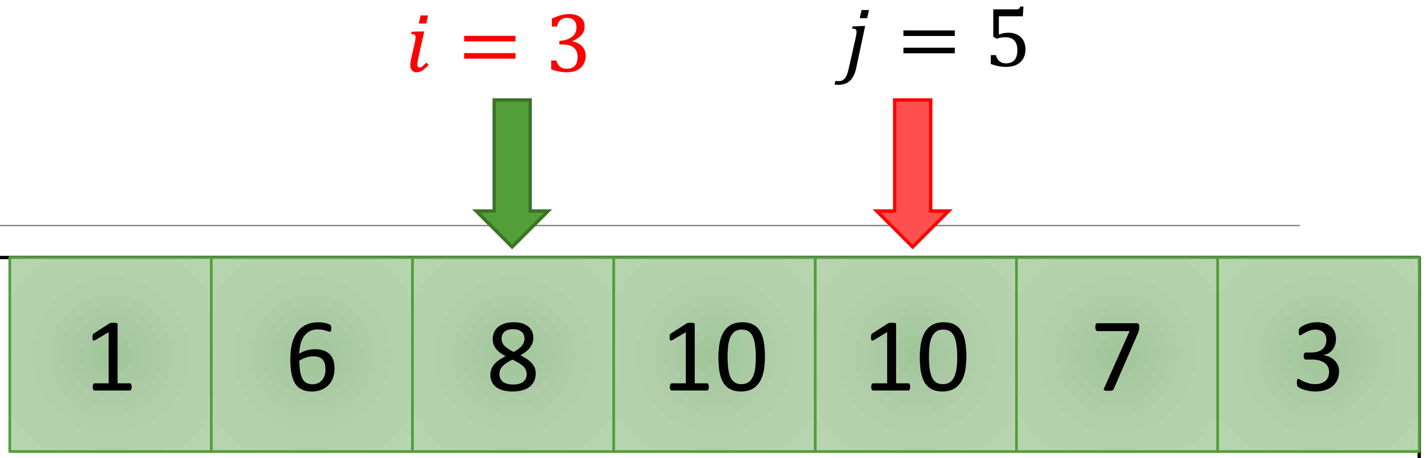
$i = i - 1$

$A[i + 1] = temp$

2

$temp$

Pseudocode



Insertion_Sort(A, n):

for $j = 2$ **to** n

$\quad temp = A[j]$

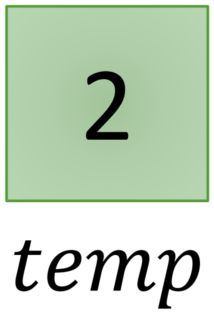
$\quad i = j - 1$

$\quad$ **while** $i \geq 1$ and $A[i] > temp$

$\quad\quad A[i+1] = A[i]$

$\quad\quad i = i - 1$

$\quad A[i+1] = temp$



Pseudocode

$i = 3$ $\quad$ $j = 5$

1	6	8	8	10	7	3

Insertion_Sort(A, n):

for $j = 2$ **to** n

$\quad temp = A[j]$

$\quad i = j - 1$

$\quad$ **while** $i \geq 1$ and $A[i] > temp$

$\quad\quad A[i + 1] = A[i]$

$\quad\quad i = i - 1$

$\quad A[i + 1] = temp$

2

$temp$

Pseudocode

$i = 2$ $j = 5$

1	6	8	8	10	7	3

Insertion_Sort(A, n):

for $j = 2$ **to** n

$temp = A[j]$

$i = j - 1$

while $i \geq 1$ and $A[i] > temp$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = temp$

2

$temp$

Pseudocode

$i = 2$ $\quad$ $j = 5$

1	6	6	8	10	7	3

Insertion_Sort(A, n):

for $j = 2$ **to** n

$\quad temp = A[j]$

$\quad i = j - 1$

$\quad$ **while** $i \geq 1$ and $A[i] > temp$

$\quad\quad A[i + 1] = A[i]$

$\quad\quad i = i - 1$

$\quad A[i + 1] = temp$

2

$temp$

Pseudocode

Insertion_Sort(A, n):

for $j = 2$ **to** n

$\quad temp = A[j]$

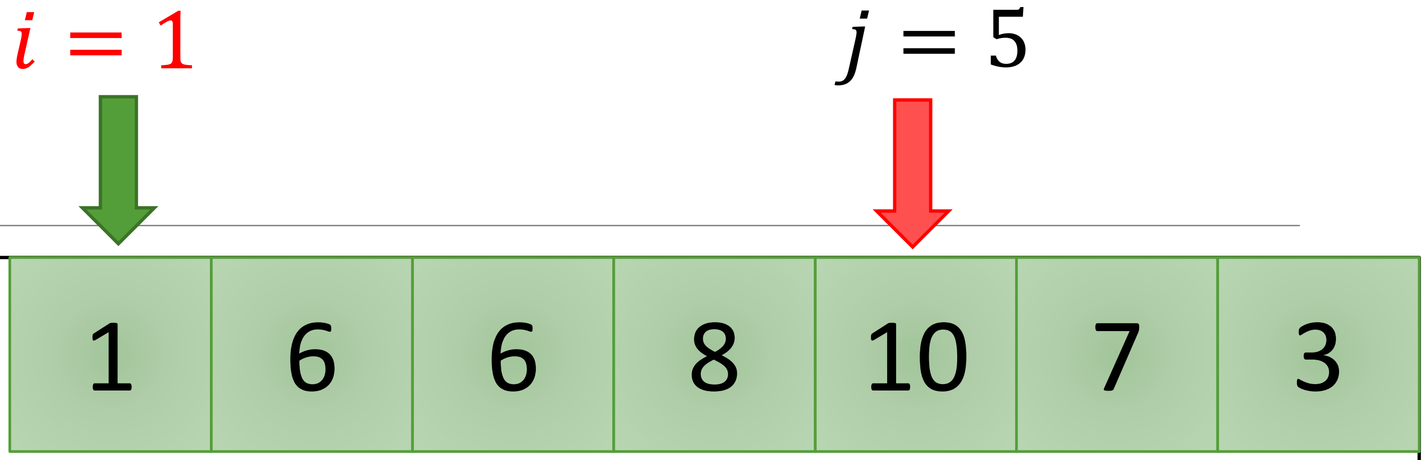
$\quad i = j - 1$

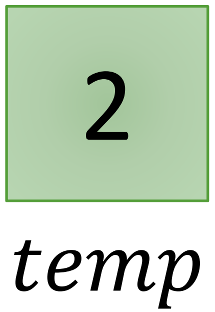
$\quad$ **while** $i \geq 1$ and $A[i] > temp$

$\quad\quad A[i + 1] = A[i]$

$\quad\quad i = i - 1$

$\quad A[i + 1] = temp$





Pseudocode

$i = 1$ $j = 5$

1	2	6	8	10	7	3

Insertion_Sort(A, n):

for $j = 2$ **to** n

$temp = A[j]$

$i = j - 1$

while $i \geq 1$ and $A[i] > temp$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = temp$

2

$temp$

Analyzing Insertion Sort

- We need to show:

1. That the algorithm is **correct** for ANY input. That is, it will always output a sorted array.
2. The **running time** of the algorithm.

- This is the case for all algorithms that we will study.
  - Except we will often put more emphasis on finding the running time

Analyzing Correctness

"Testing can only show the **presence** of errors, not their **absence**"
- E. W. Dijkstra

- How do you prove correctness?
- What were you taught to do in your Software Engineering class?
  - Inspections
  - Walkthroughs
  - Testing
    - Unit Testing
    - Integration Testing
    - Functional/System Testing
- But testing is inconclusive and not a proof that there are no errors.

Analyzing Correctness

- **Definition:** An algorithm is said to be **correct** if it completely satisfies the specification it is meant to accomplish with respect to the problem it is solving and terminates with the expected output.
- We will use **formal verification** to prove that algorithms satisfy certain **properties** or **invariants** that ensure correctness.
- Such a proof is called a **Loop Invariant Proof**.

Correctness: Loop Invariant Proof

- To **prove** that an algorithm is **correct** for ANY input, we will need to show that it is correct (so far) for EVERY iteration of the loop:

1. Formulate a **loop invariant** that defines what correctness means
2. **Initialization**: Show that the loop invariant is true at the start of the first iteration
3. **Maintenance**: Assume loop invariant is true at the start of iteration j. Then prove it is true at the start of iteration $j+1$
4. **Termination**: Show that, when the loop terminates, the algorithm outputs the correct answer.

Analyzing Correctness

Insertion_Sort(A, n):

for $j = 2$ **to** n

  $temp = A[j]$

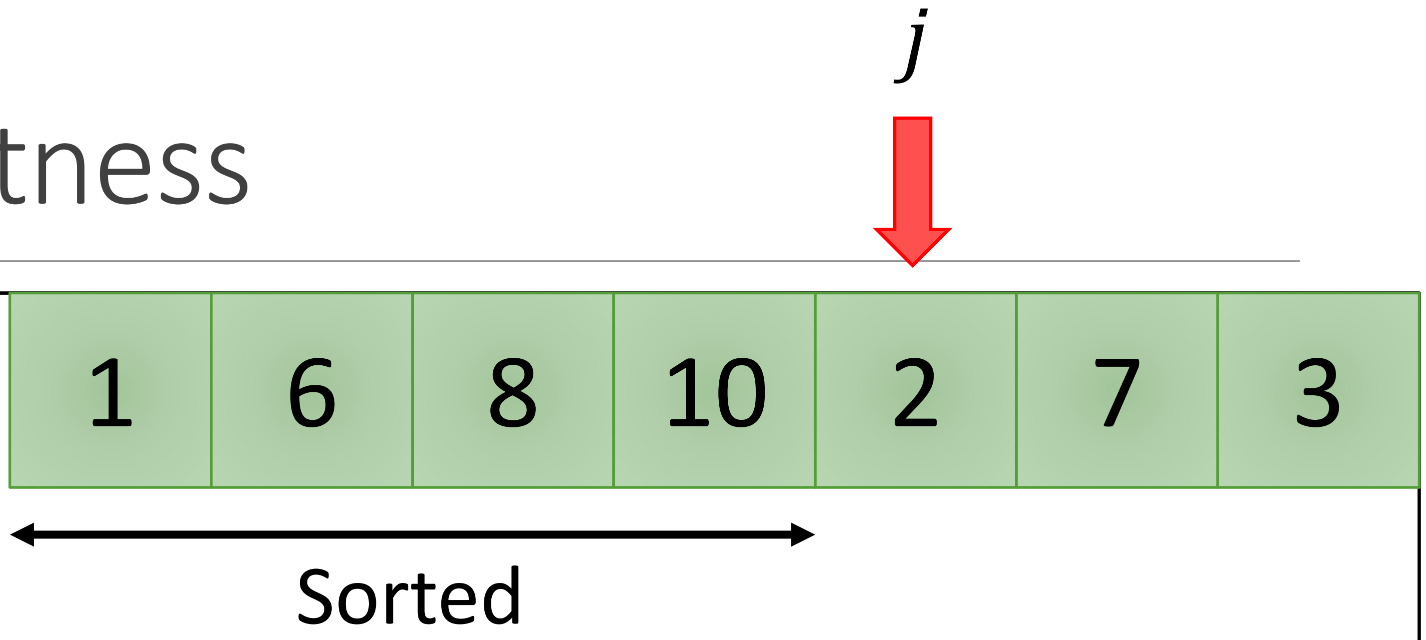
  $i = j - 1$

  while $i \geq 1$ and $A[i] > temp$

    $A[i + 1] = A[i]$

    $i = i - 1$

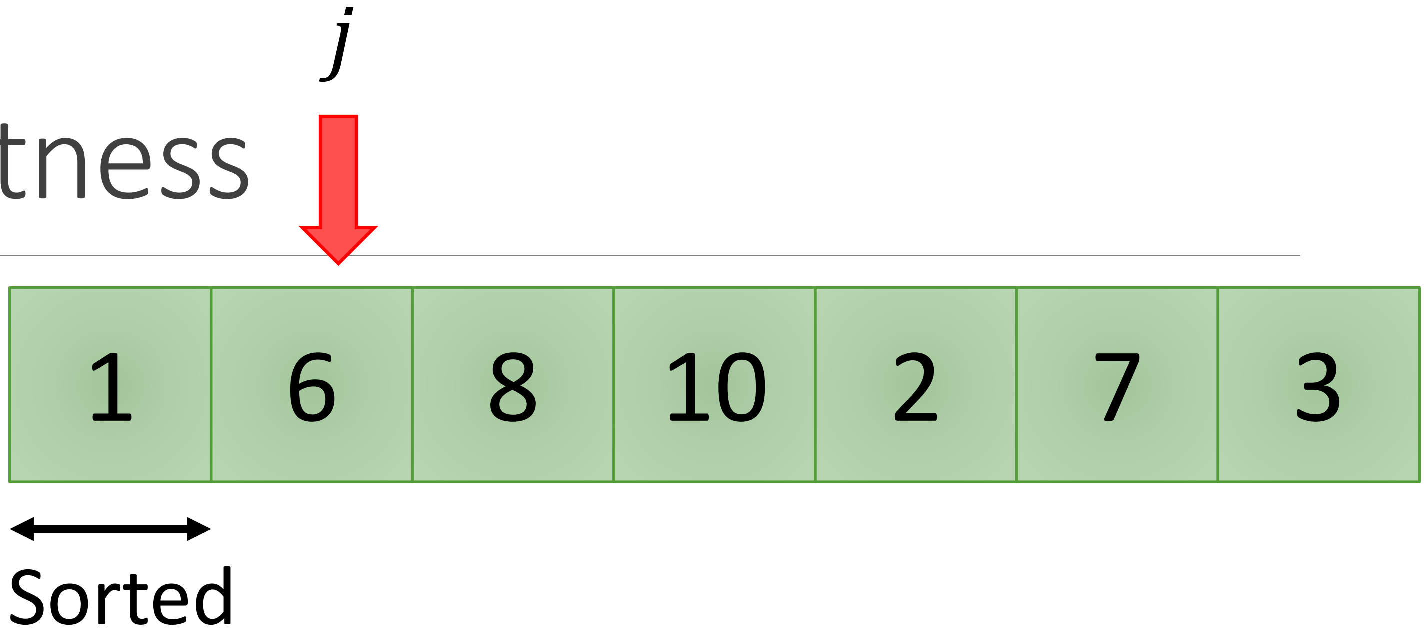
  $A[i + 1] = temp$



Loop Invariant:
At the start of iteration j of the **for** loop, subarray $A[1, \ldots, j - 1]$ is sorted.

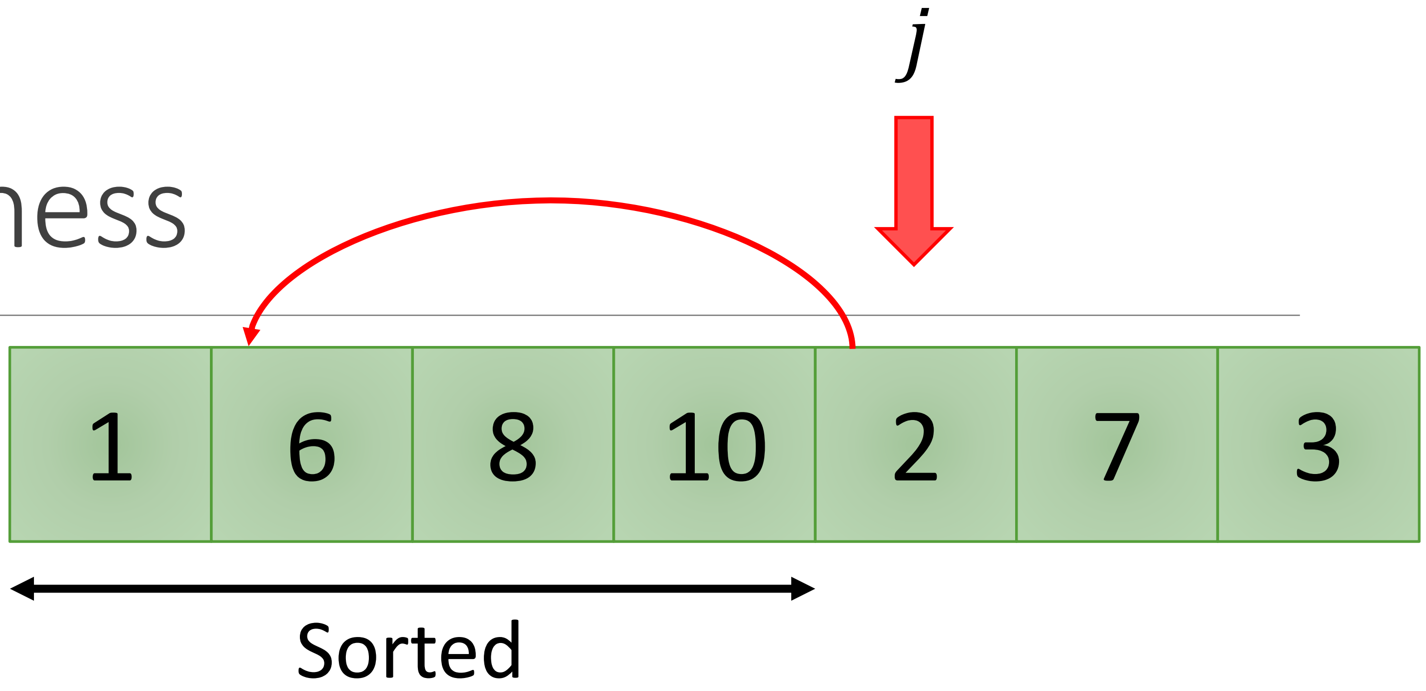
Analyzing Correctness

- **Initialization**: Show that the loop invariant is true in the first iteration
- Is the loop invariant true in the first iteration?
- Yes. Proof: At the start of the first iteration ($j = 2$), the array $A[1, \dots, 1] = A[1]$ is sorted (trivially)



Loop Invariant:
At the start of iteration j of the **for** loop, subarray $A[1, \dots, j-1]$ is sorted.

Analyzing Correctness



- **Maintenance**: Assume loop invariant is true at the start of iteration j. Then prove it is true at the start of iteration $j+1$
- Proof: Informally, if we assume that loop invariant is true at start of iteration j then $A[1, \ldots, j-1]$ is sorted.
  - Then, at start of iteration $j+1$ the element $A[j]$ will be inserted in the proper location within $A[1, \ldots j]$ by the inner while loop so $A[1, \ldots j]$ will also be sorted

Loop Invariant:
At the start of iteration j of the **for** loop, subarray $A[1, \ldots, j-1]$ is sorted.

Analyzing Correctness

j

1	2	3	6	7	8	10

Sorted

- **Termination**: Show that, when the loop terminates, the algorithm outputs the correct answer. ✓

- Proof: The loop terminates when $j = n + 1$. At this point, the loop invariant states that the subarray $A[1, \ldots, (n + 1) - 1]$ is sorted, which is the entire array – so the output is correct.

Loop Invariant:
At the start of iteration j of the **for** loop, subarray $A[1, \ldots, j - 1]$ is sorted.

Analyzing running time

- The running time T must be a **function of the input**; there are inputs for which the algorithm is slow and others for which the algorithm is fast.

- We parametrize the running time of an algorithm with the **input size**. i.e., $T(n)$ where n is the **input size**.

Types of Analyses

- Worst-case:
  - $T(n) =$ **MAXIMUM** running time of the algorithm on input of size n
- Average-case:
  - $T(n) =$ **EXPECTED** running time of the algorithm over **all** inputs of size n
- Best-case:
  - $T(n) =$ **MINIMUM** running time of the algorithm over input of size n

Types of Analyses

- Worst-case:
  - $T(n)$ = **MAXIMUM** running time of the algorithm on input of size n
- Average-case:
  - $T(n)$ = **EXPECTED** running time of the algorithm over **all** inputs of size n
- Best-case:
  - $T(n)$ = **MINIMUM** running time of the algorithm over input of size n

Worst-case running time

- What is insertion's sort worst-case running time?
  - It depends!

- Depends on:
  - CPU clock speed
  - Memory speed
  - Cache size
  - Etc.

- Need a **machine-independent** way if measuring time!

Asymptotic Analysis

- Main Idea:
  - Identify the **primitive** operations and the **data structure** used
  - Ignore machine-dependent **constants** and factors
  - Examine the **growth** of $T(n)$ as $n \to \infty$

$\boldsymbol{\Theta}$ – Notation (informally)

1. Drop low-order terms
2. Ignore leading constants

- E.g. $T(n) = 22n^2 + 41n + 12 = \Theta(n^2)$
- E.g. $T(n) = 4n^2 + 3n^5 + 2n^8 = \Theta(n^8)$

Worst-case running time of insertion sort

Insertion_Sort(A, n):

for $j = 1$ **to** n

$\quad temp = A[j]$

$\quad i = j - 1$

$\quad$ **while** $i > 0$ and $A[i] > temp$

$\quad\quad A[i+1] = A[i]$

$\quad\quad i = i - 1$

$\quad A[i+1] = temp$

10	8	7	6	3	2	1

1. Data structure? **Array**
2. Primitive operations?

Worst-case running time of insertion sort

Insertion_Sort(A, n):
for $j = 1$ **to** n
    $temp = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > temp$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = temp$

10	8	7	6	3	2	1

1. Data structure? Array
2. Primitive operations?
   Compare, Shift, Assign

Worst-case running time of insertion sort

Insertion_Sort(A, n):

for $j = 1$ **to** n

$temp = A[j]$

$i = j - 1$

while $i > 0$ and $A[i] > temp$

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = temp$

10	8	7	6	3	2	1

j	#Compares	#Shifts	#Assign	$T_j(n)$
1	0	0	1	0
2	1	1	1	2+1
3	2	2	1	4+1
…	…	…	…	…
n	n-1	n-1	1	2(n-1)+1

$$\boldsymbol{T(n) = \sum_j T_j(n) = \sum_j 2(j-1)+1 = \Theta(n^2)}$$

Best-case running time of insertion sort

Insertion_Sort(A, n):
for $j = 1$ to n
$temp = A[j]$
$i = j - 1$
while $i > 0$ and $A[i] > temp$
$A[i+1] = A[i]$
$i = i - 1$
$A[i+1] = temp$

1	2	3	6	7	8	10

j	#Compares	#Shifts	#Assign	$T_j(n)$
1	0	0	1	0
2	1	0	1	1+1
3	1	0	1	1+1
…	…	…	…	…
n	1	0	1	1+1

$$T(n) = \sum_j T_j(n) = \sum_j 2 = \Theta(n)$$

Asymptotic Analysis

- What can we say about the relative performance of any two algorithms using Θ −notation?

- A $\Theta(n^2)$ algorithm is always faster than a $\Theta(n^3)$ algorithm **for large enough $\boldsymbol{n}$**.
  - We say the $\Theta(n^2)$ algorithm is asymptotically faster than the $\Theta(n^3)$ one.

- Though sometimes, in practice, we might favor asymptotically slower algorithms if we only care about "smaller" values of n

Try this example: Finding the Maximum

Find_Max(A, n):

$m = A[1]$

for $j = 2$ **to** n

  if $A[j] > m$ **then**

    $m = A[j]$

return m

- Prove **correctness**
  - Define loop invariant
  - Initialization
  - Maintenance
  - Termination
- Find the **running time**

Finding the Maximum: Correctness

- **Loop invariant:** At the start of iteration j, m is the maximum of $A[1, \ldots, j-1]$
- **Initialization:** At start of first iteration $j = 2$, it is true that $m = A[1]$ is clearly the maximum of subarray $A[1, \ldots, 1] = A[1]$

Find_Max(A, n):

$m = A[1]$

for $j = 2$ **to** n

    if $A[j] > m$ **then**

        $m = A[j]$

return m

Finding the Maximum: Correctness

- **Loop invariant:** At the start of iteration j, m is the maximum of $A[1, \ldots, j-1]$
- **Maintenance:** Assume m is the maximum of $A[1, \ldots, j-1]$ at start of iteration j
- Then at iteration $j+1$, either m stays the same or $A[j+1]$ is the new maximum. So m now becomes the maximum of $A[1, \ldots, j]$

Find_Max(A, n):

$m = A[1]$

for $j = 2$ **to** n

  if $A[j] > m$ **then**

    $m = A[j]$

return m

Finding the Maximum: Correctness

- **Loop invariant:** At the start of iteration j, m is the maximum of $A[1, \ldots, j-1]$
- **Termination:** Loop terminates when $j = n+1$ at which point the loop invariant states that m is the maximum of $A[1, \ldots, (n+1)-1] = A[1, \ldots, n]$ which is indeed what is required of this algorithm

```
Find_Max(A, n):
m = A[1]
for j = 2 to n
        if A[j] > m then
                m = A[j]
return m
```