CS11212 - Spring 2022

Data Structures & Introduction to Algorithms

Analysis of Algorithms

Searching & Sorting: Part 2

Ibrahim Albluwi

Sorting: A Fundamental Problem

Problem. Given a list of n elements, order them in non-decreasing (or ascending) order.
Common variant. Order the elements in descending order.

Sorting: A Fundamental Problem

Problem. Given a list of *n* elements, order them in non-decreasing (or ascending) order.
Common variant. Order the elements in descending order.

Requirement. The meaning of `<, >, ==` for the element type must be defined.
In C++, it is defined for `int`, `double`, `char`, `string`, etc.,
but not for user defined types (e.g. What does `car1 > car2` mean?)

Sorting: A Fundamental Problem

Problem. Given a list of n elements, order them in non-decreasing (or ascending) order.
Common variant. Order the elements in descending order.

Requirement. The meaning of `<`, `>`, `==` for the element type must be defined.
In C++, it is defined for `int`, `double`, `char`, `string`, etc.,
but not for user defined types (e.g. What does `car1 > car2` mean?)

Too many ways to sort!

Bubble Sort
Selection Sort
Insertion Sort
Exchange Sort
Cocktail Sort
Stooge Sort
Comb Sort

Quicksort
Heapsort
Timsort
Merge Sort
Shell Sort
BST Sort
Cycle Sort

MSD Radix Sort
LSD Radix Sort
Counting Sort
Bucket Sort
Bitonic Sort
Bogo (Stupid) Sort
...

Sorting: A Fundamental Problem

Problem. Given a list of *n* elements, order them in non-decreasing (or ascending) order.
Common variant. Order the elements in descending order.

Requirement. The meaning of `<`, `>`, `==` for the element type must be defined.
In C++, it is defined for `int`, `double`, `char`, `string`, etc.,
but not for user defined types (e.g. What does `car1 > car2` mean?)

Inefficient, but easy to analyze!
(*covered in this course*)

Too many ways to sort!

```
Bubble Sort
Selection Sort
Insertion Sort
Exchange Sort
Cocktail Sort
Stooge Sort
Comb Sort

Quicksort
Heapsort
Timsort
Merge Sort
Shell Sort
BST Sort
Cycle Sort

MSD Radix Sort
LSD Radix Sort
Counting Sort
Bucket Sort
Bitonic Sort
Bogo (Stupid) Sort
...
```

Sorting: A Fundamental Problem

Problem. Given a list of *n* elements, order them in non-decreasing (or ascending) order.
Common variant. Order the elements in descending order.

Requirement. The meaning of `<`, `>`, `==` for the element type must be defined.
In C++, it is defined for `int`, `double`, `char`, `string`, etc.,
but not for user defined types (e.g. What does `car1 > car2` mean?)

Efficient, but harder to analyze!
(*covered in the Algorithms course*)

Too many ways to sort!

Bubble Sort
Selection Sort
Insertion Sort
Exchange Sort
Cocktail Sort
Stooge Sort
Comb Sort

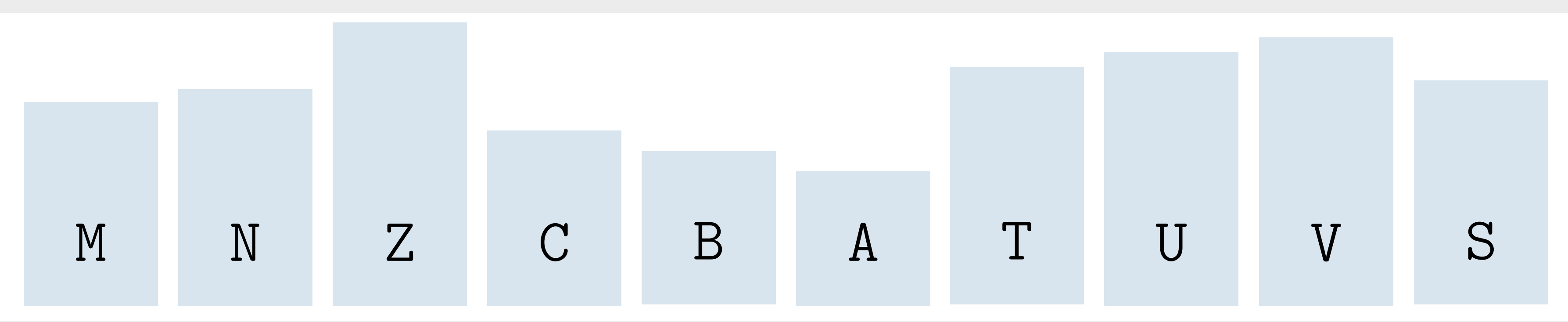
Quicksort
Heapsort
Timsort
Merge Sort

Shell Sort
BST Sort
Cycle Sort

MSD Radix Sort
LSD Radix Sort
Counting Sort
Bucket Sort
Bitonic Sort
Bogo (Stupid) Sort
...

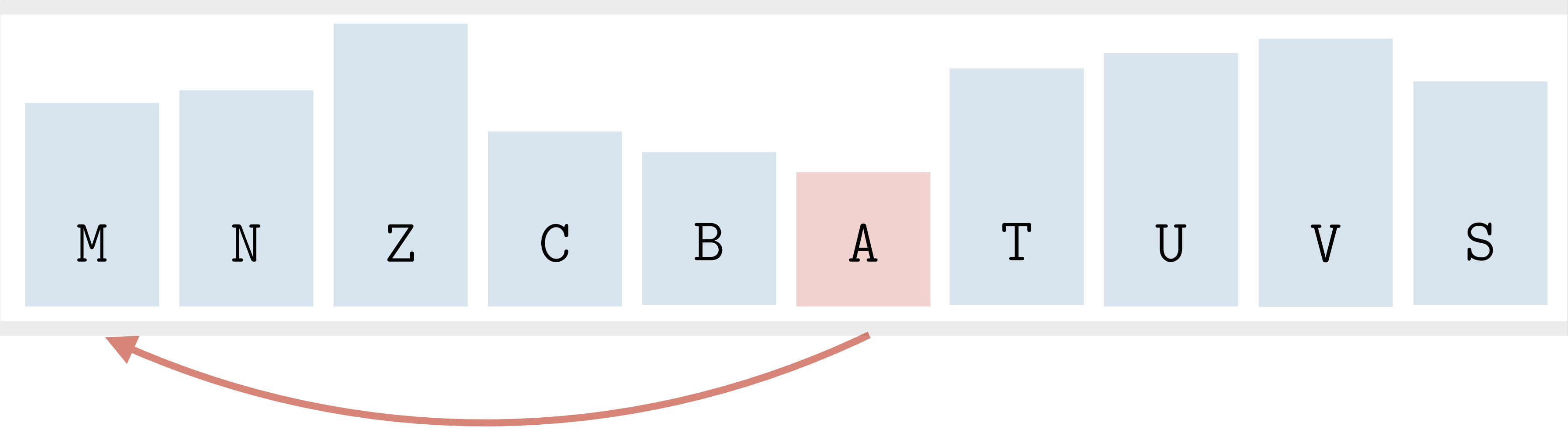
Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Sorting Warmup

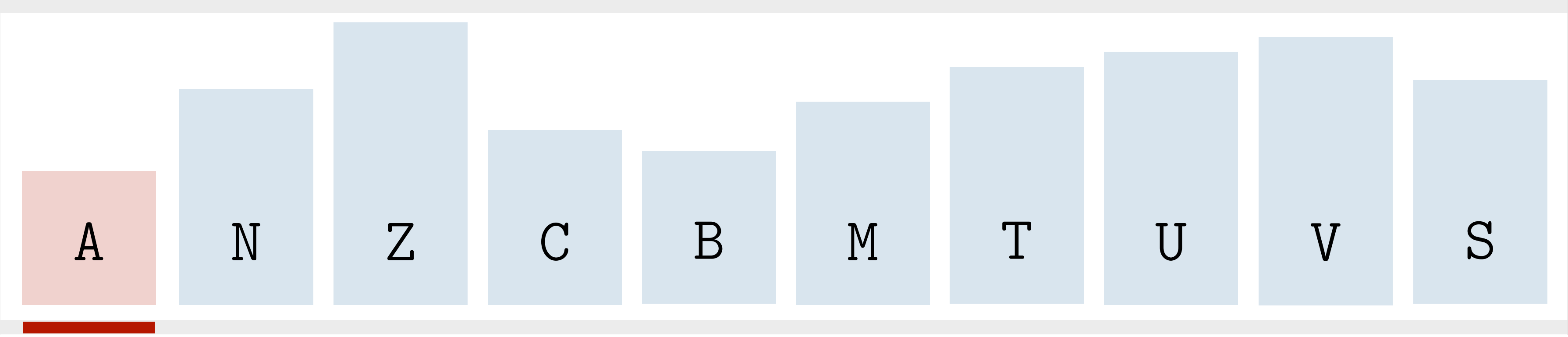
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

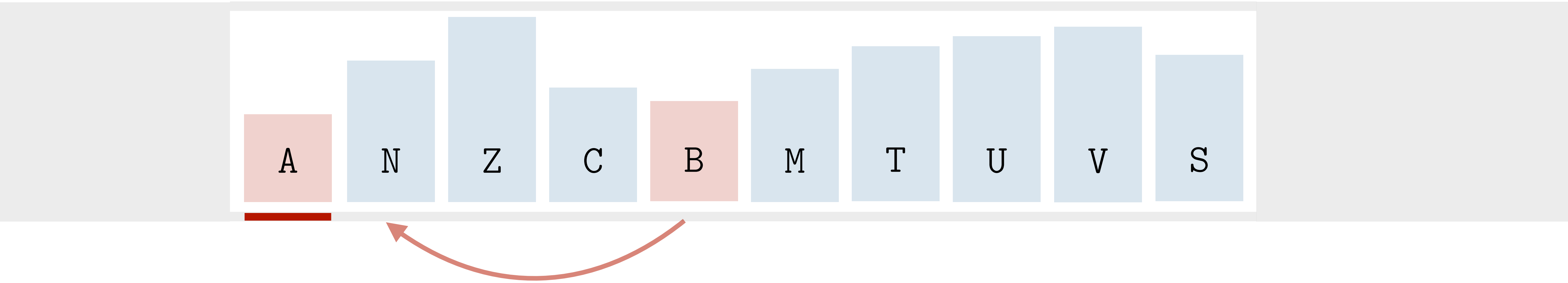
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

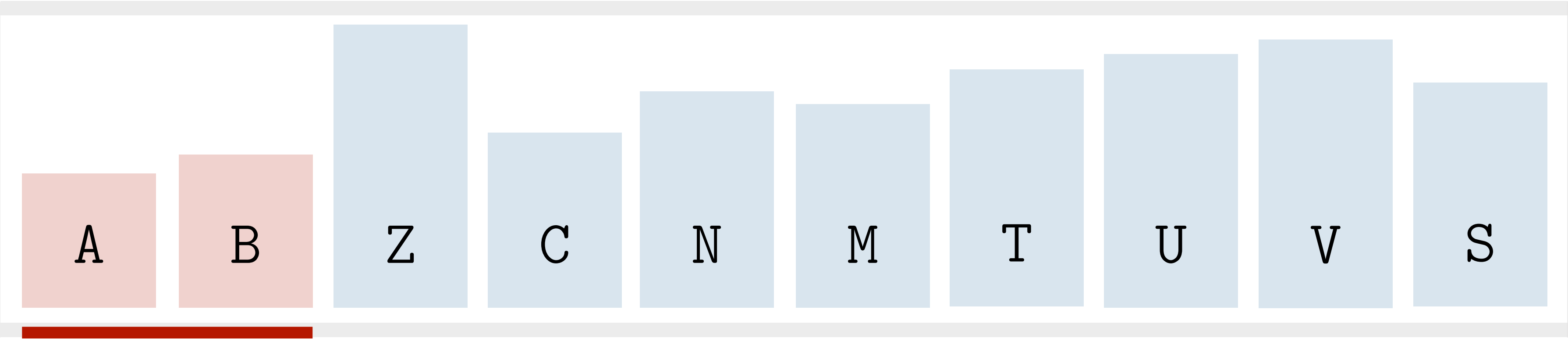
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

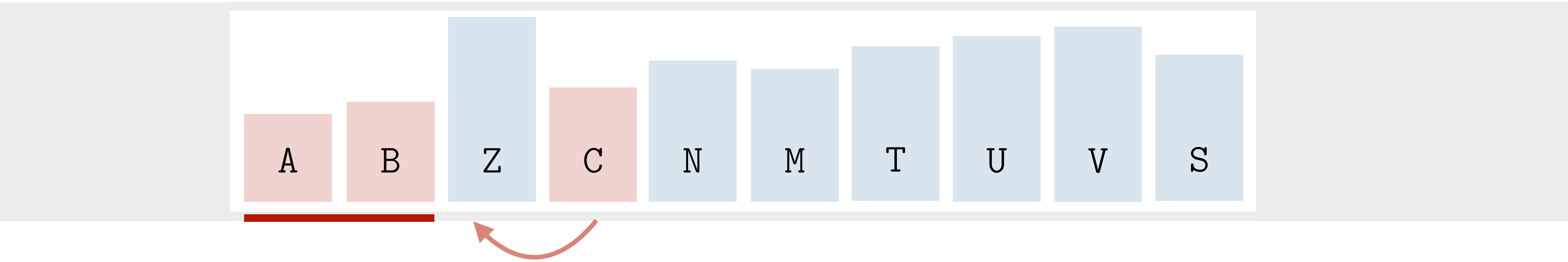
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

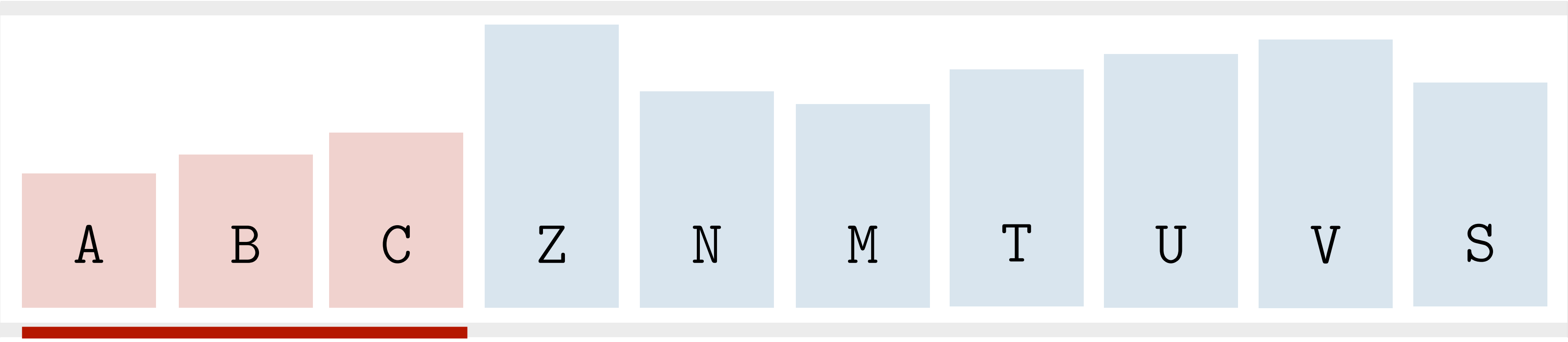
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

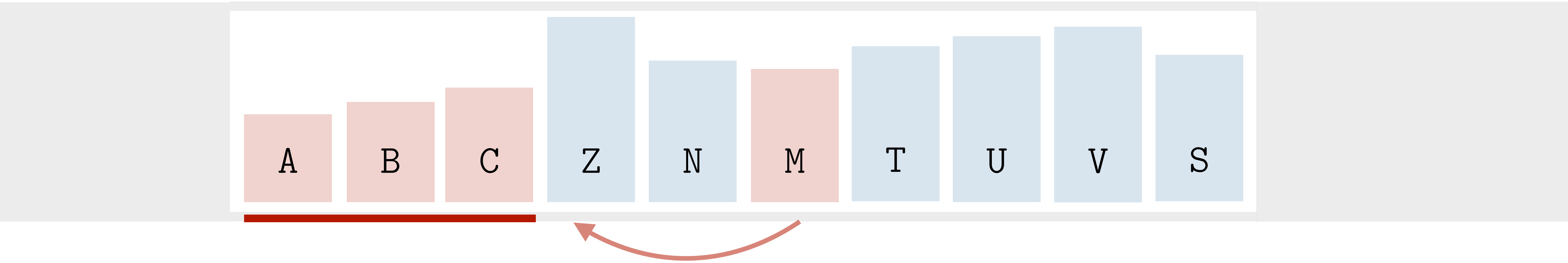
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

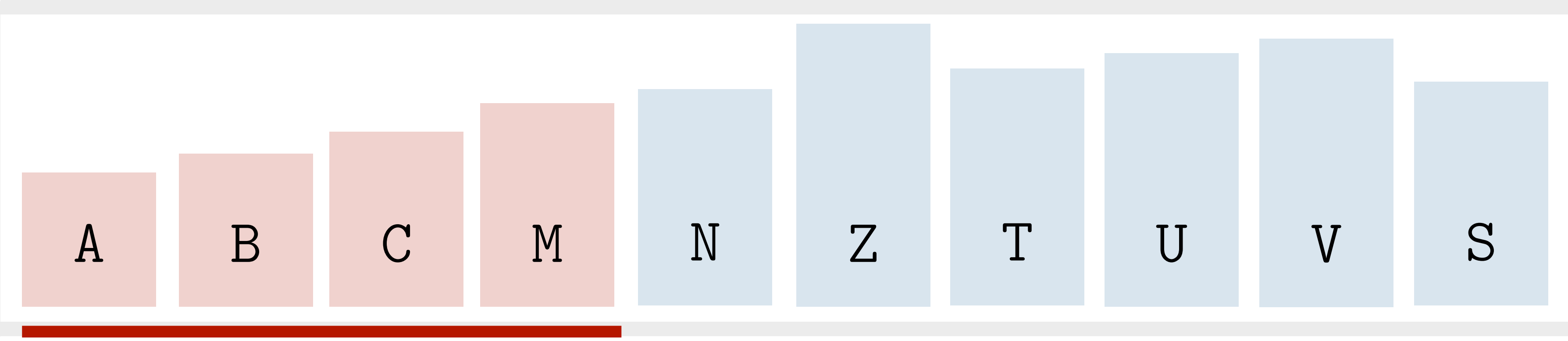
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

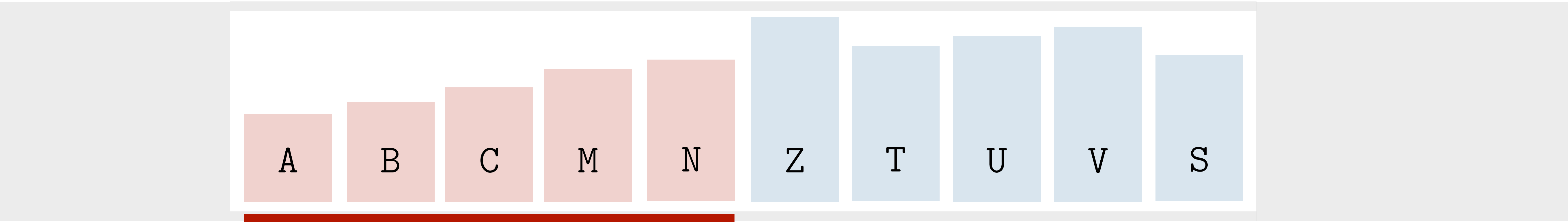
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

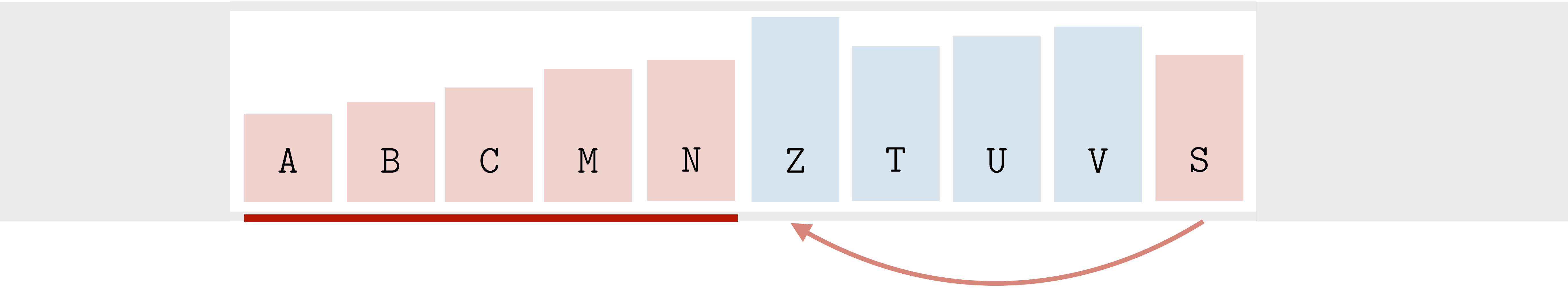
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

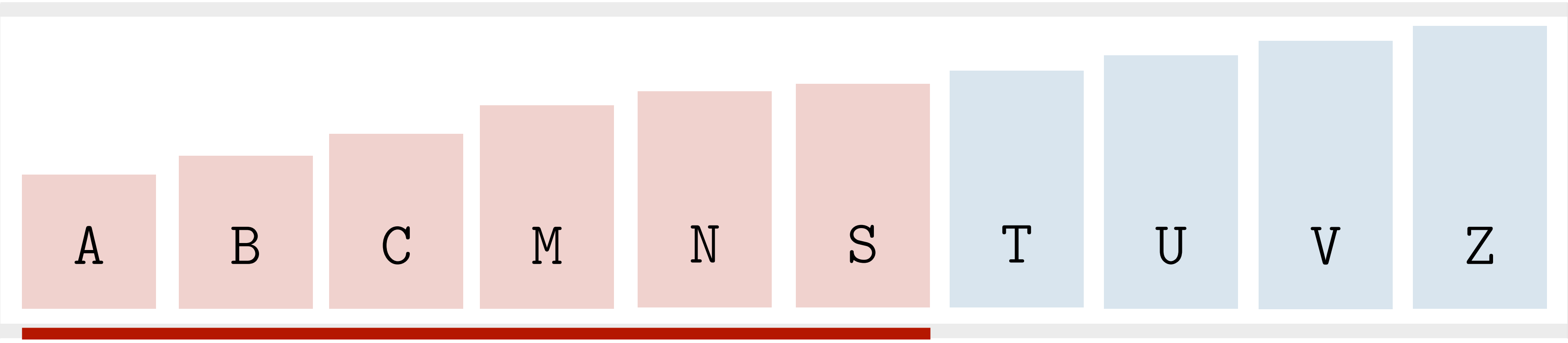
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

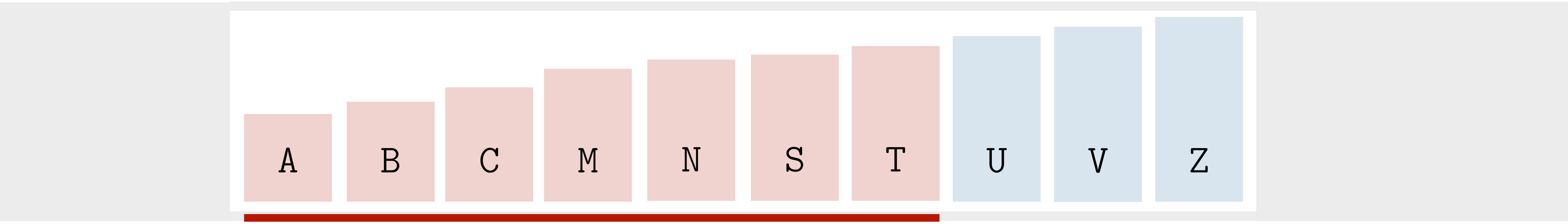
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

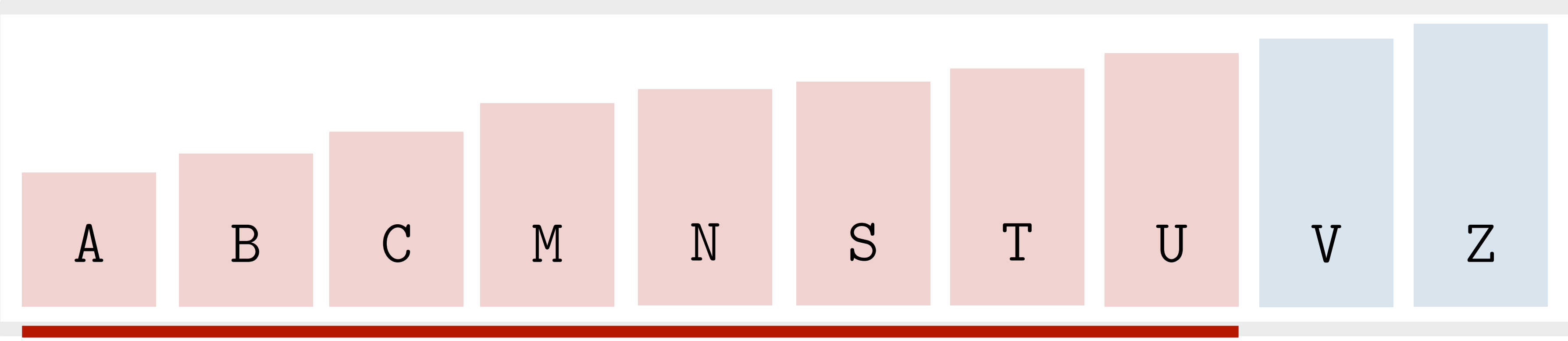
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

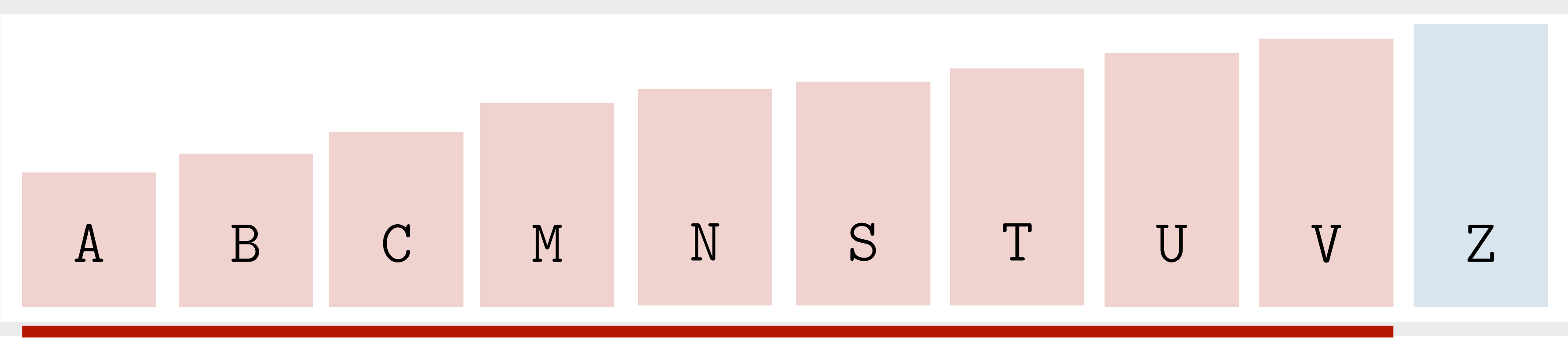
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

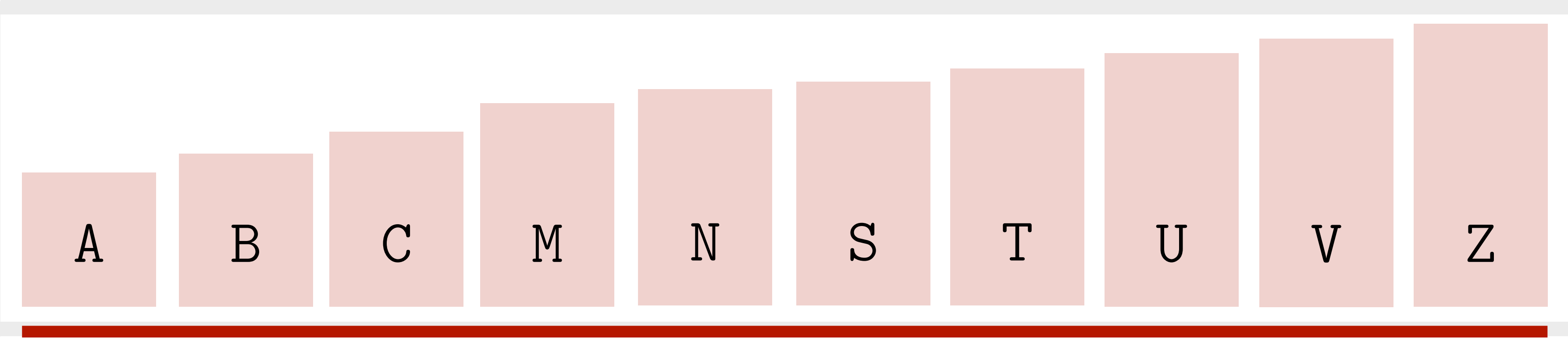
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

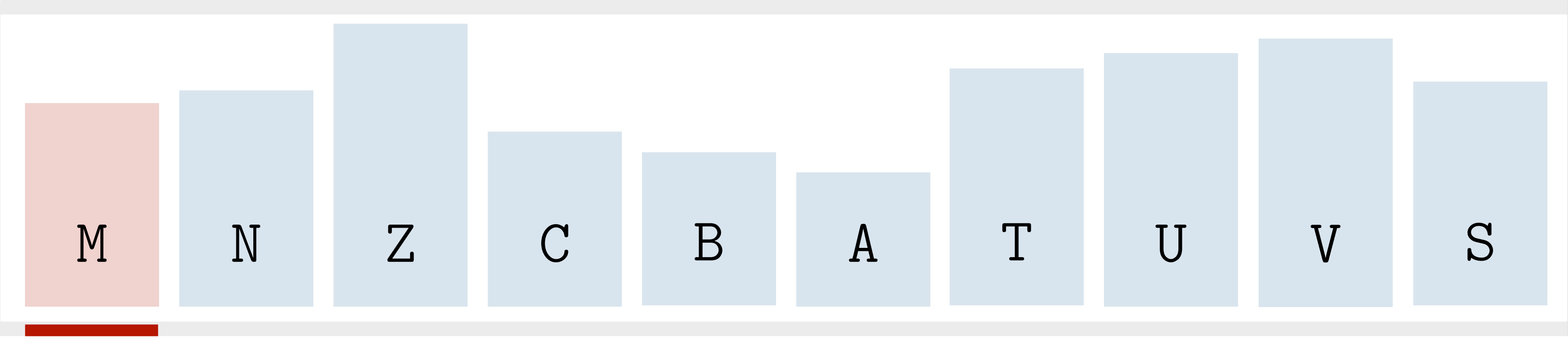
Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

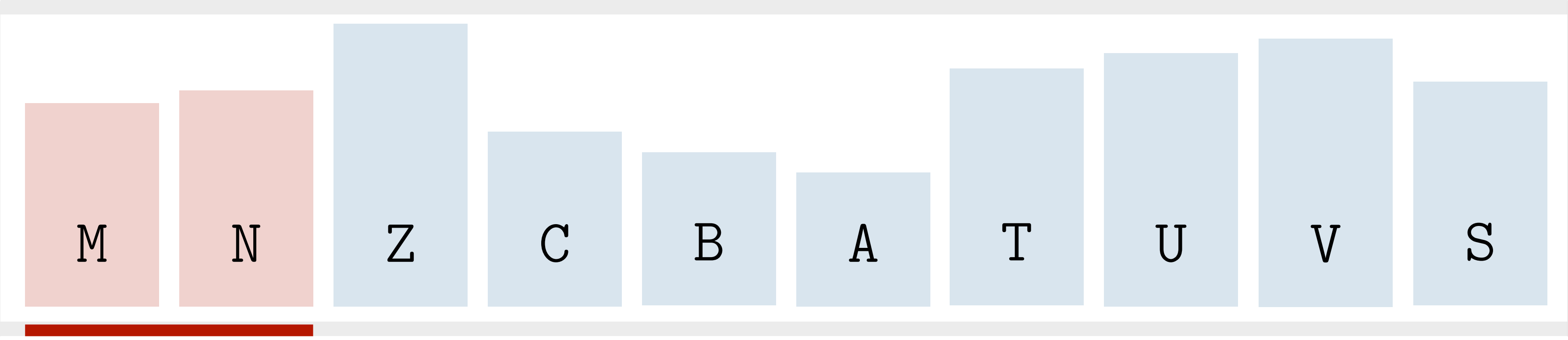


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

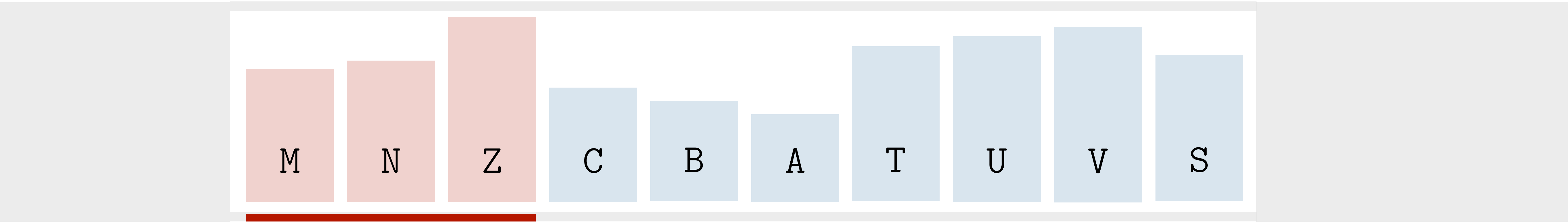


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

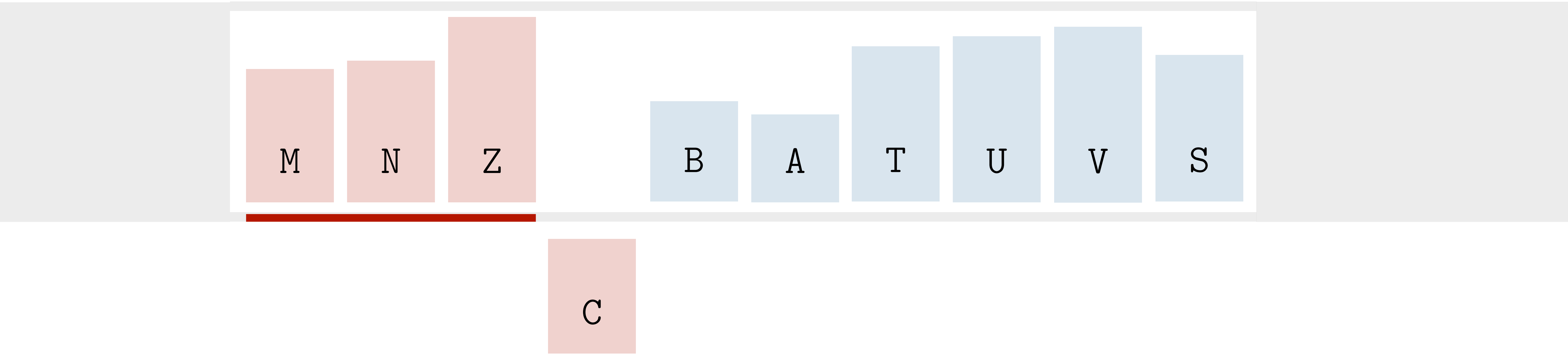


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

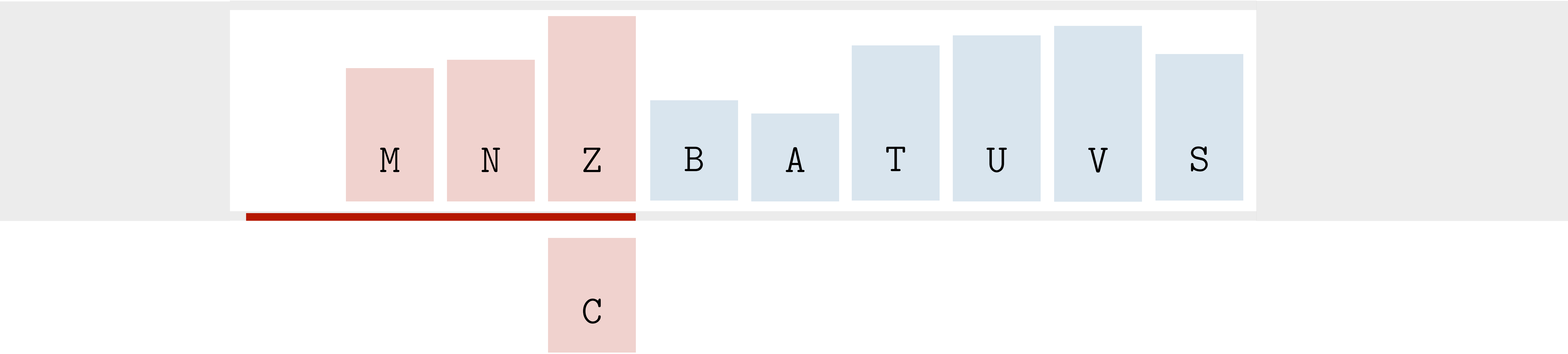


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

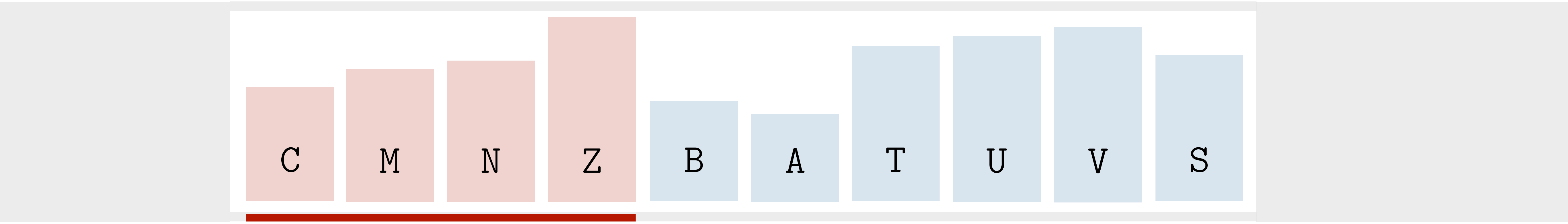


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

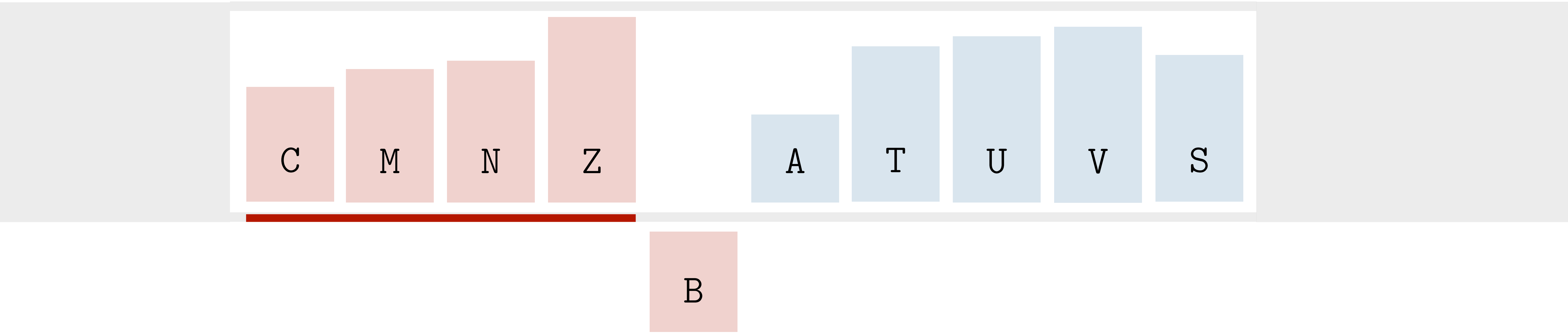


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

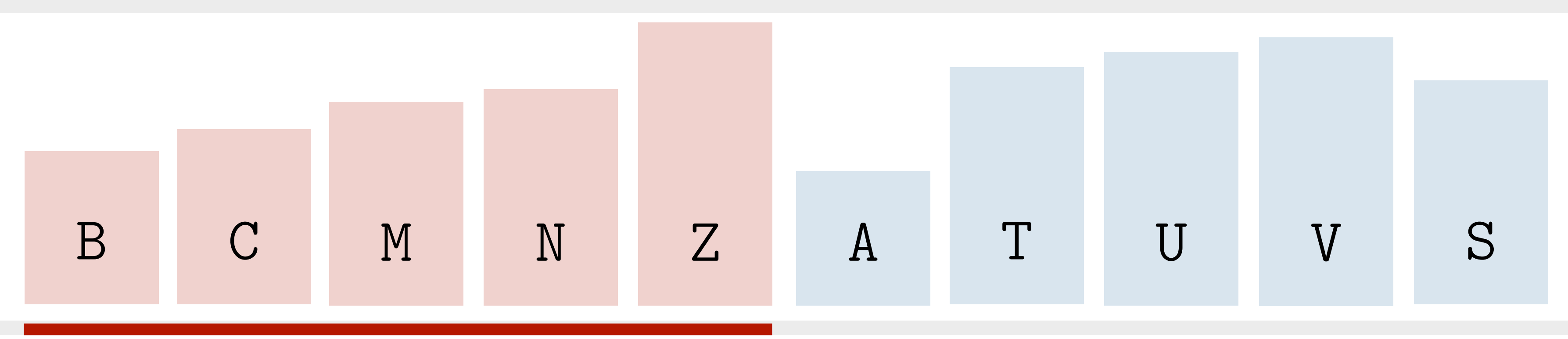


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

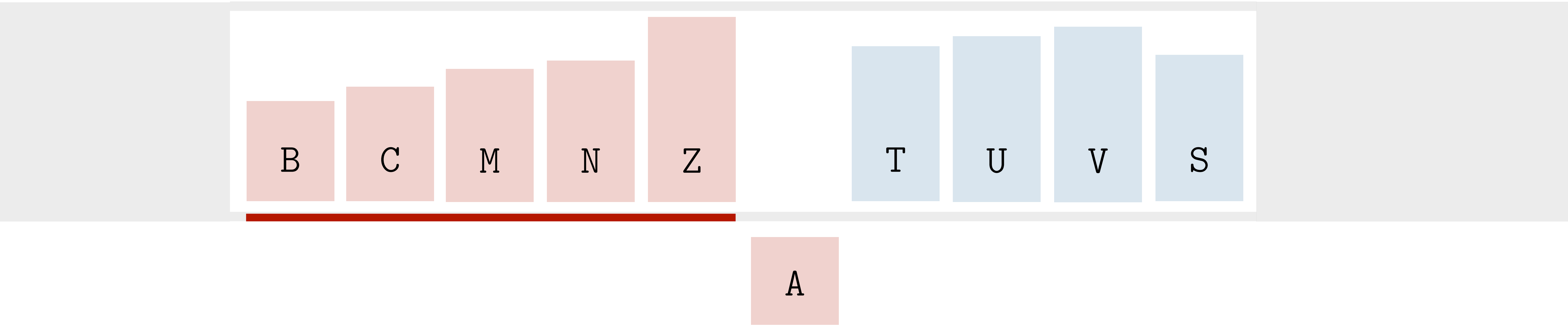


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

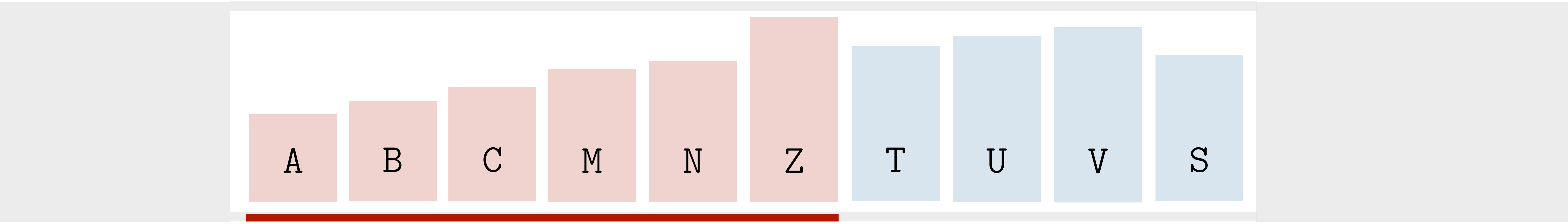


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

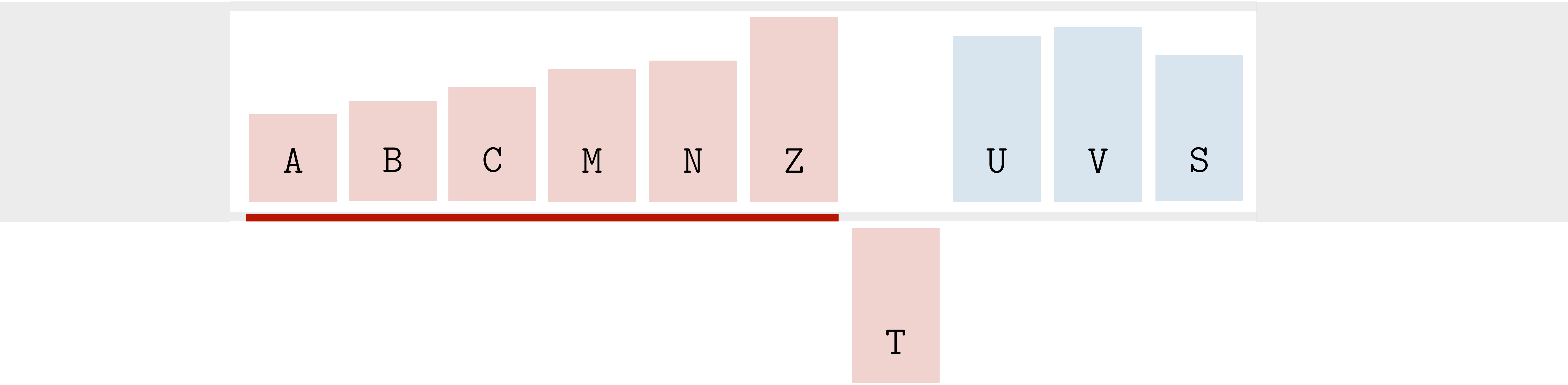


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

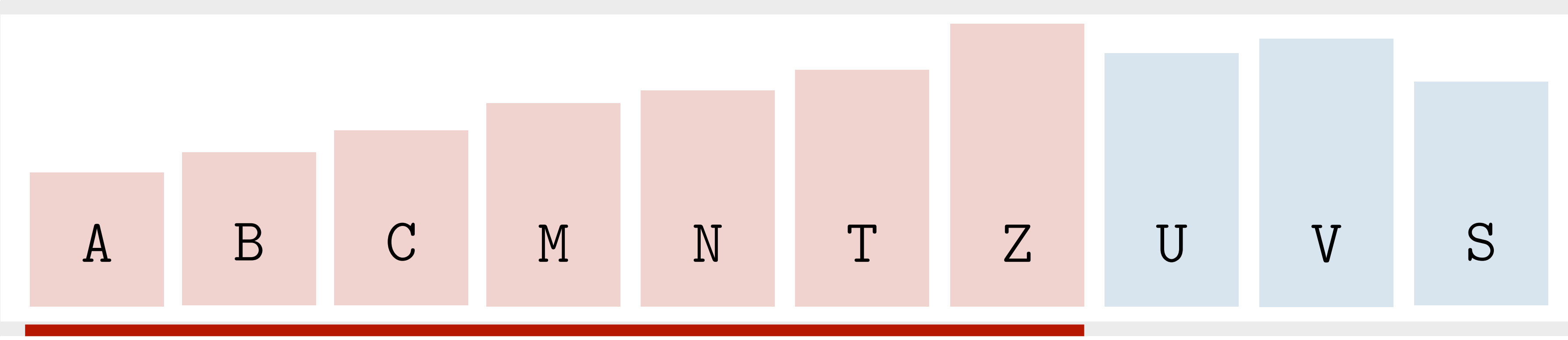


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

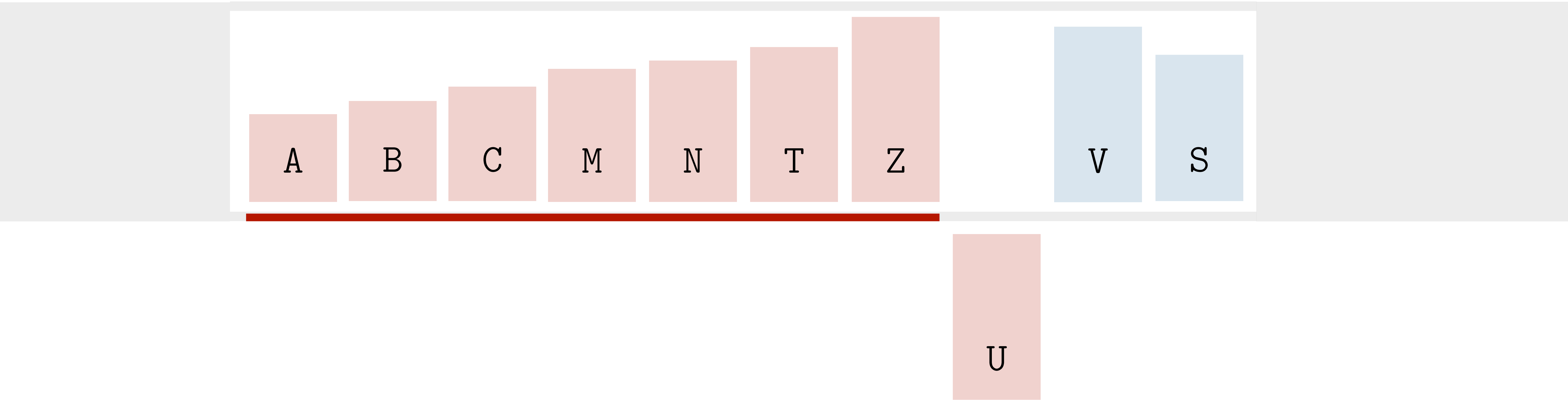


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

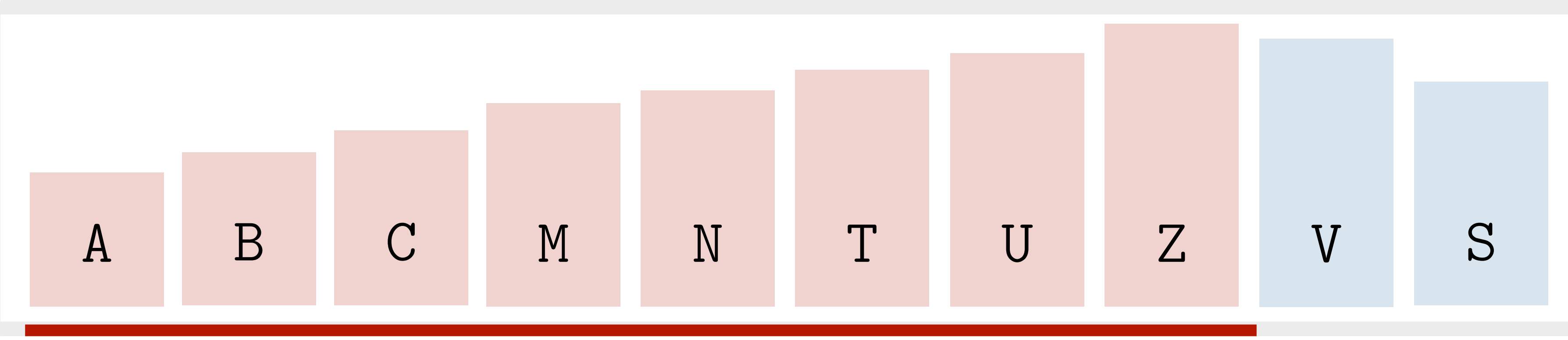


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

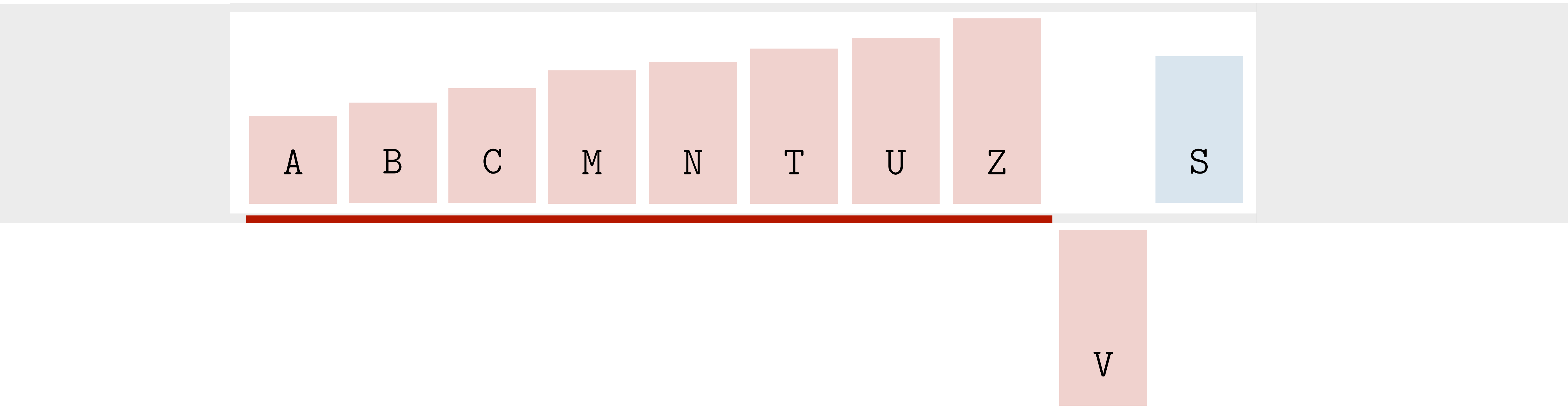


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

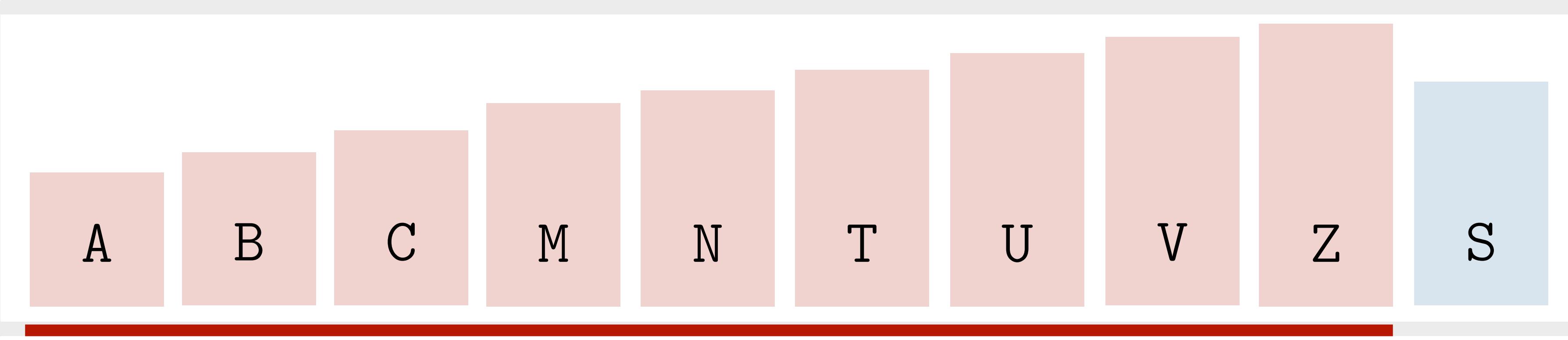


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

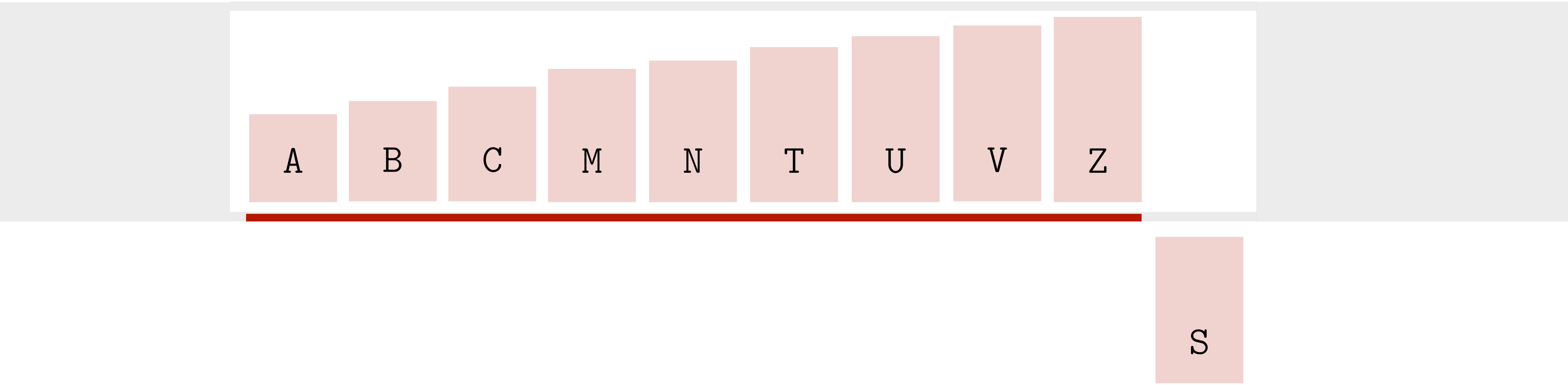


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

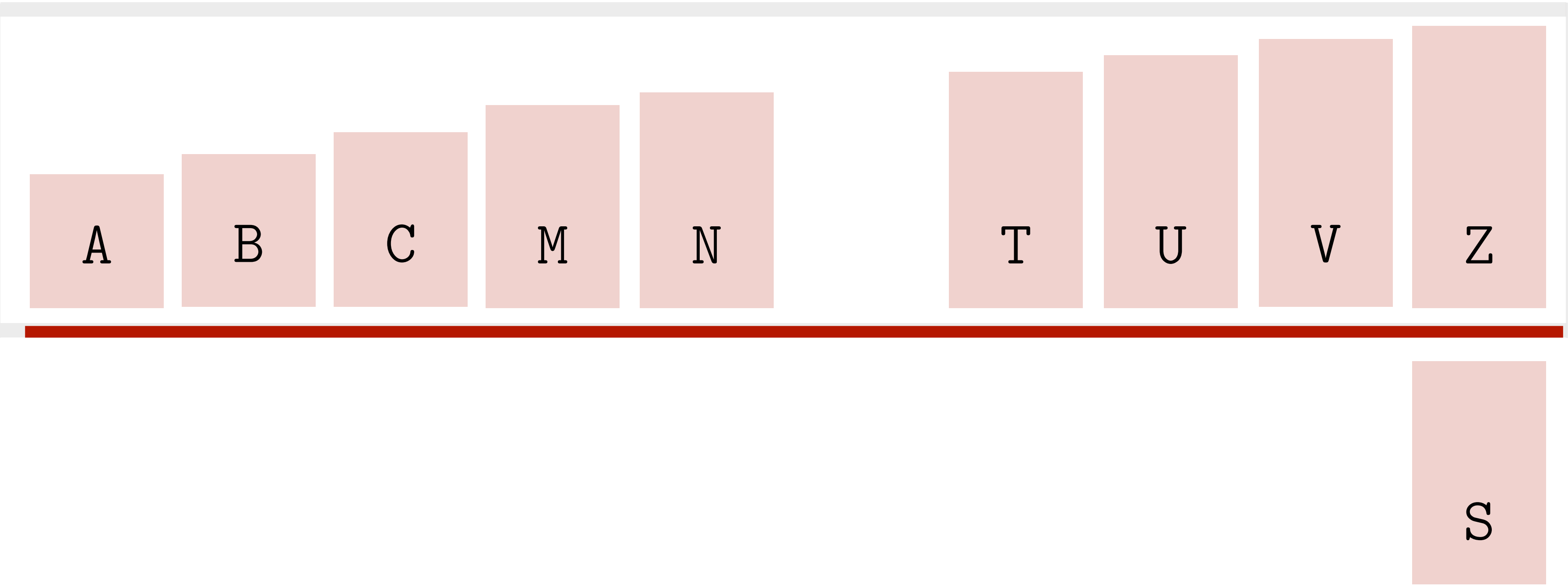


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

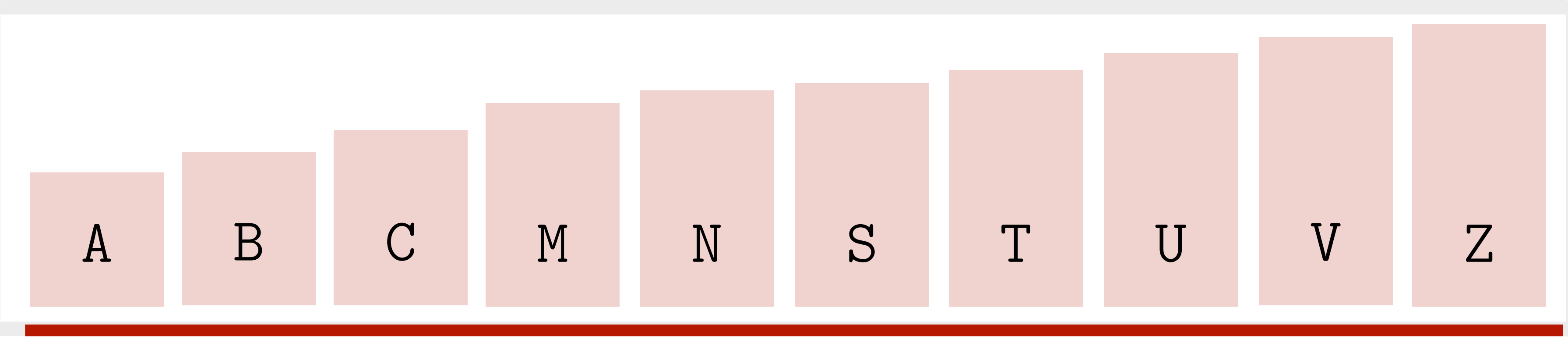


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.

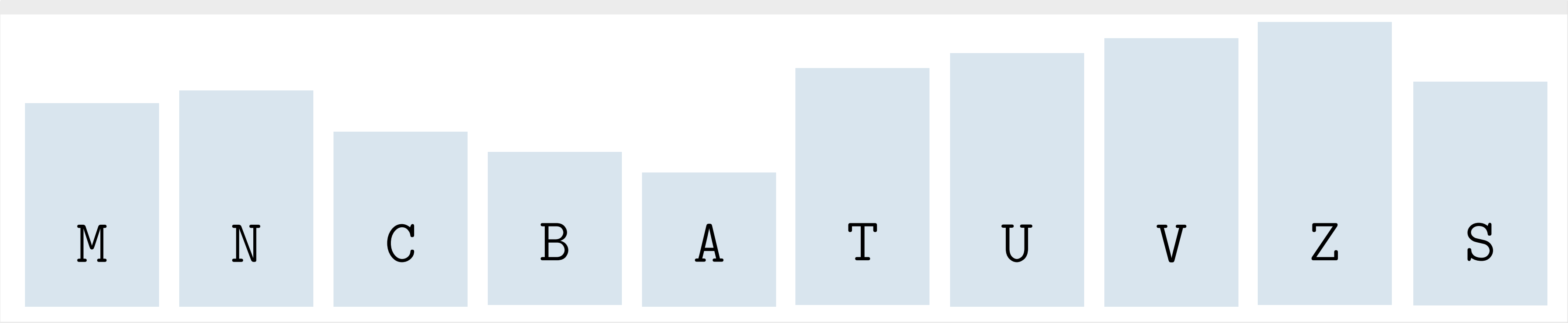


Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



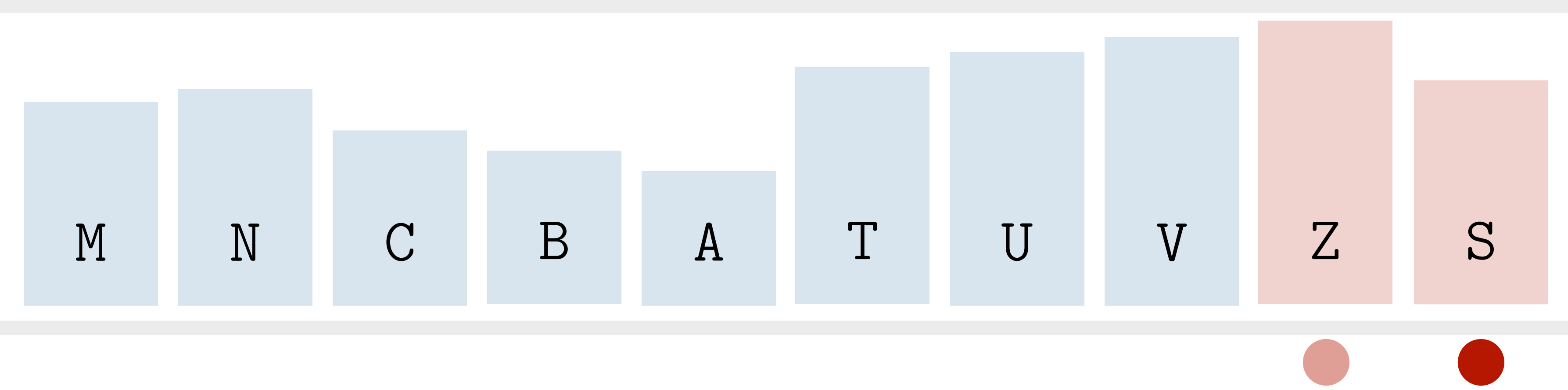
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



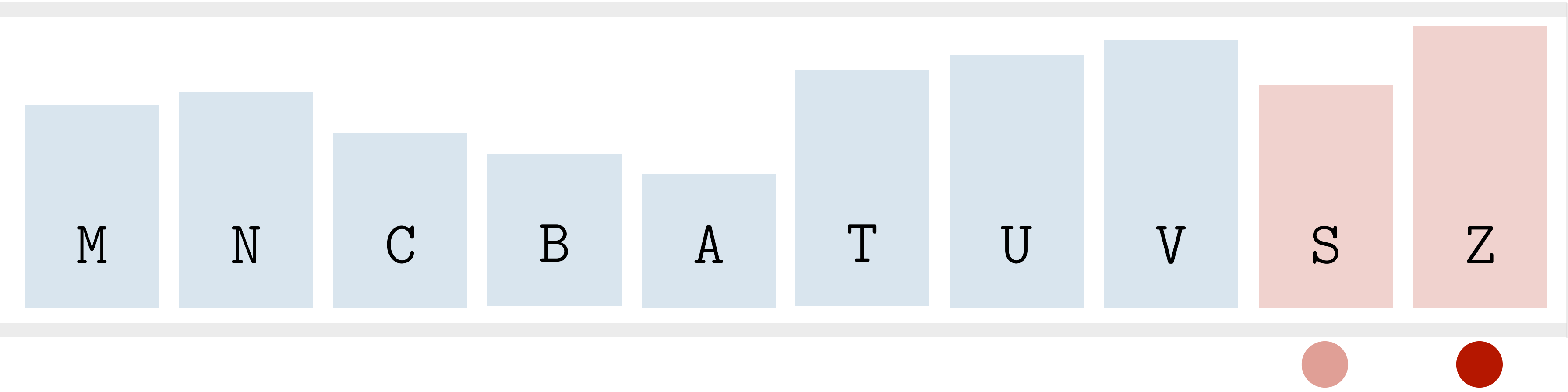
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



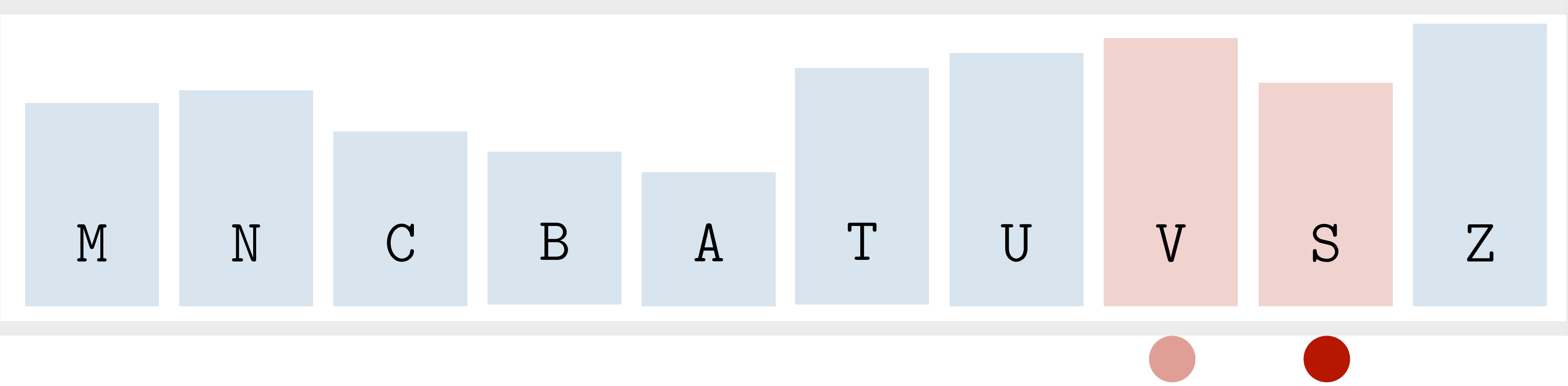
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



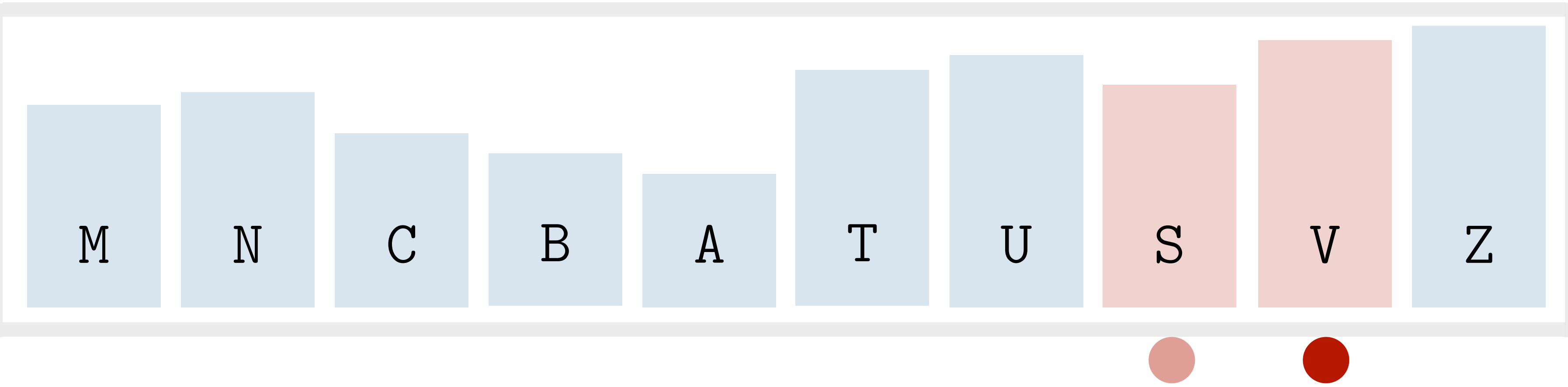
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



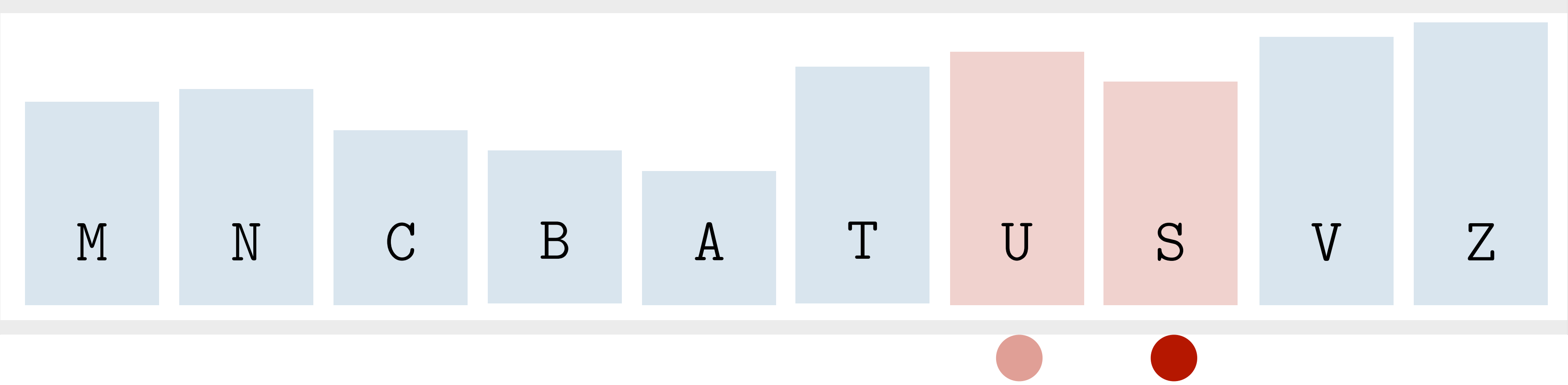
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



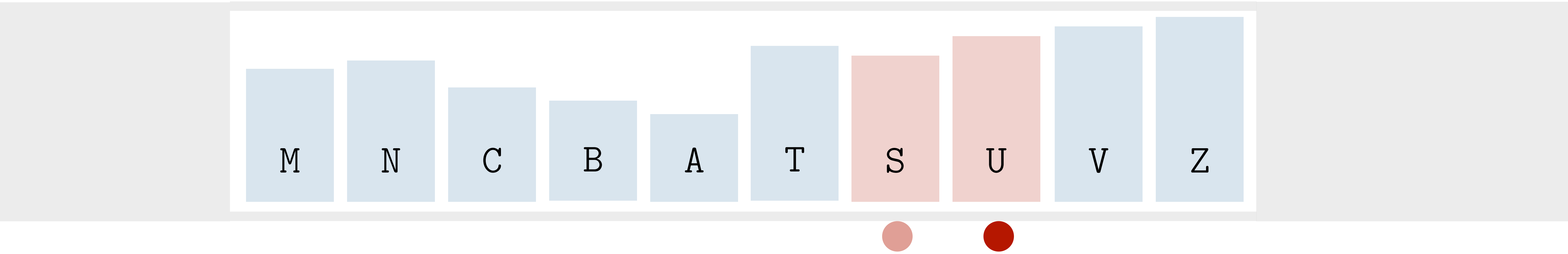
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



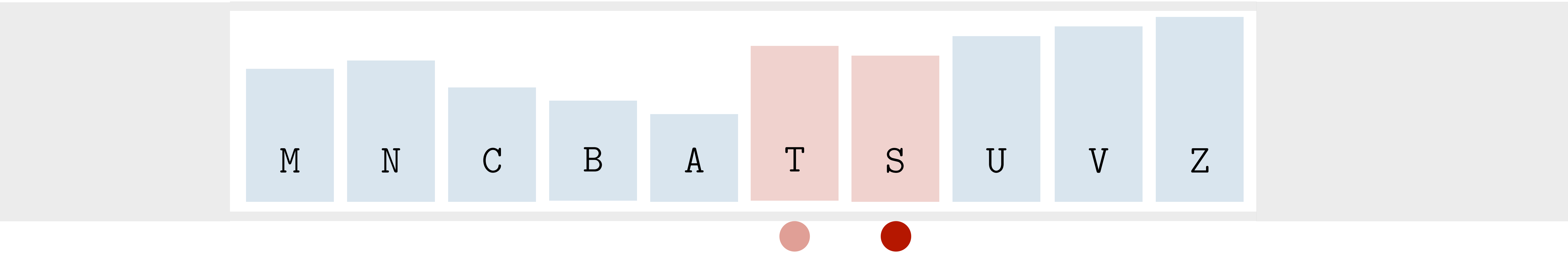
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



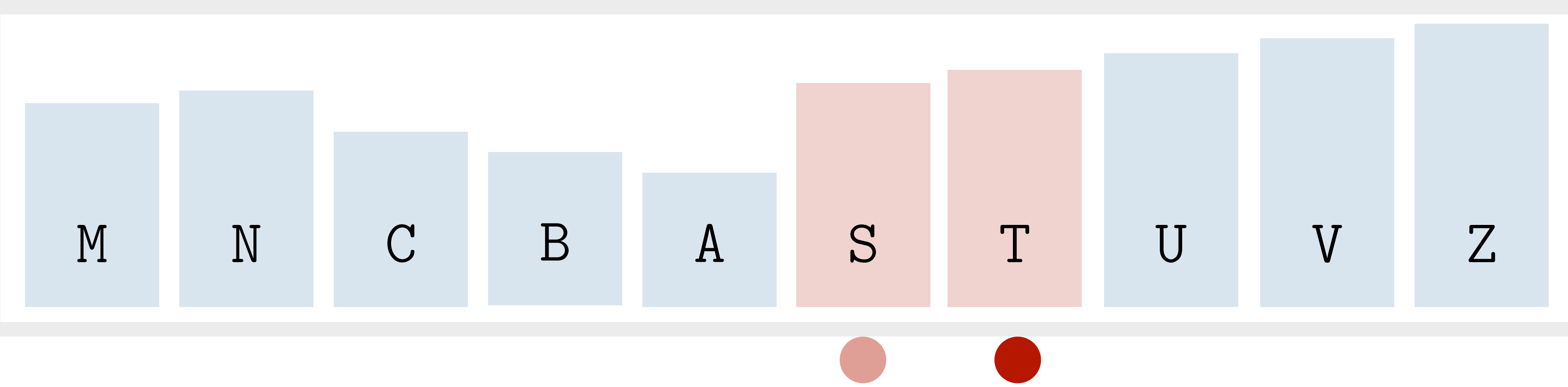
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



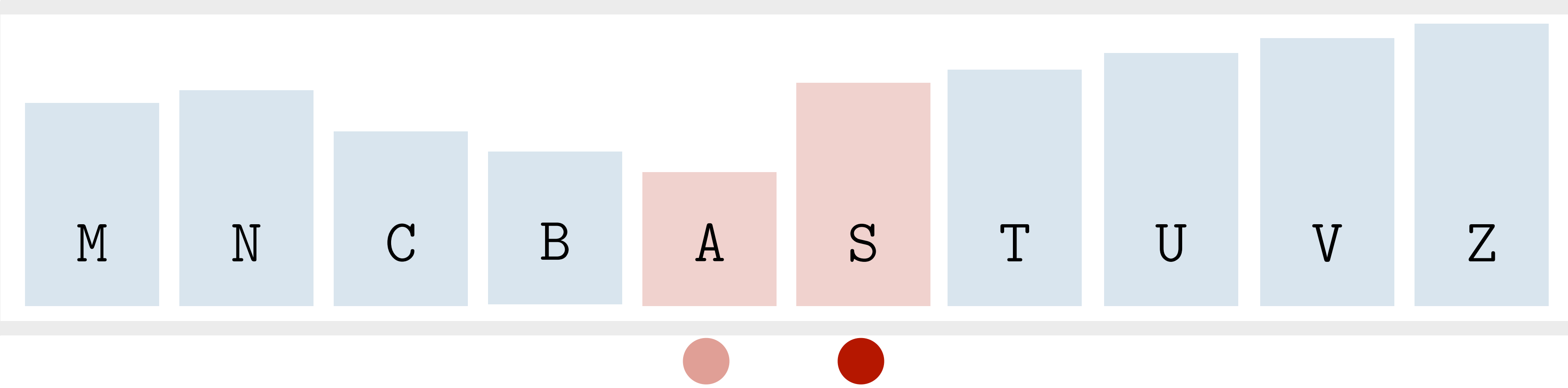
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



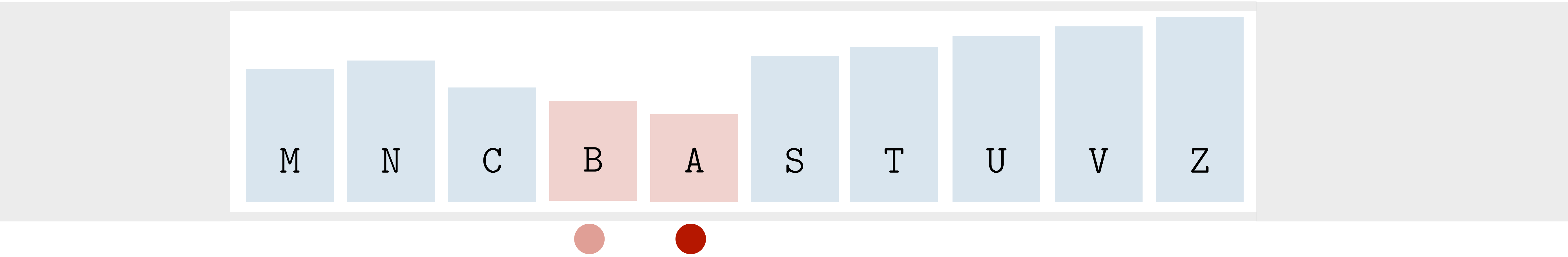
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



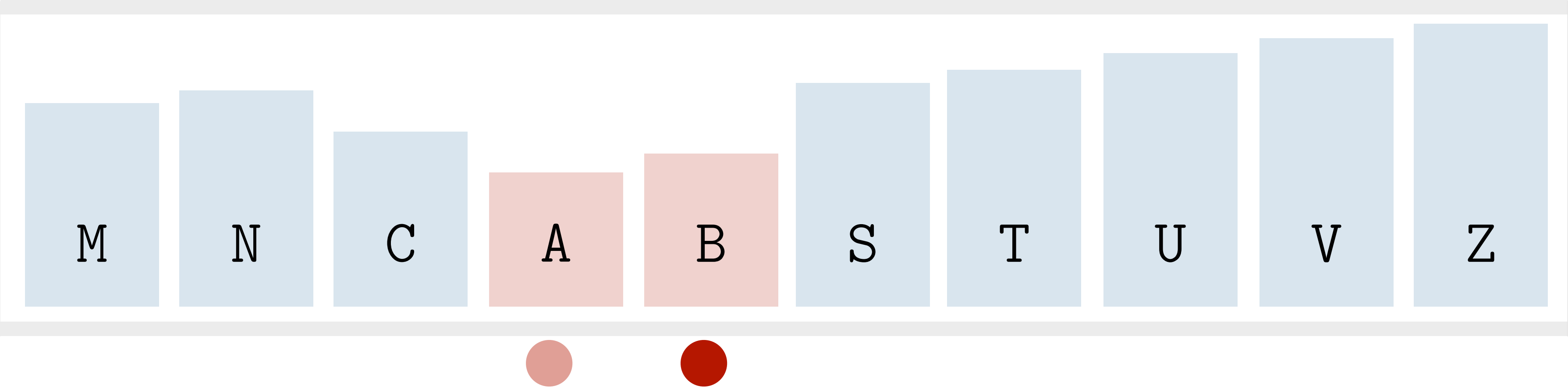
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



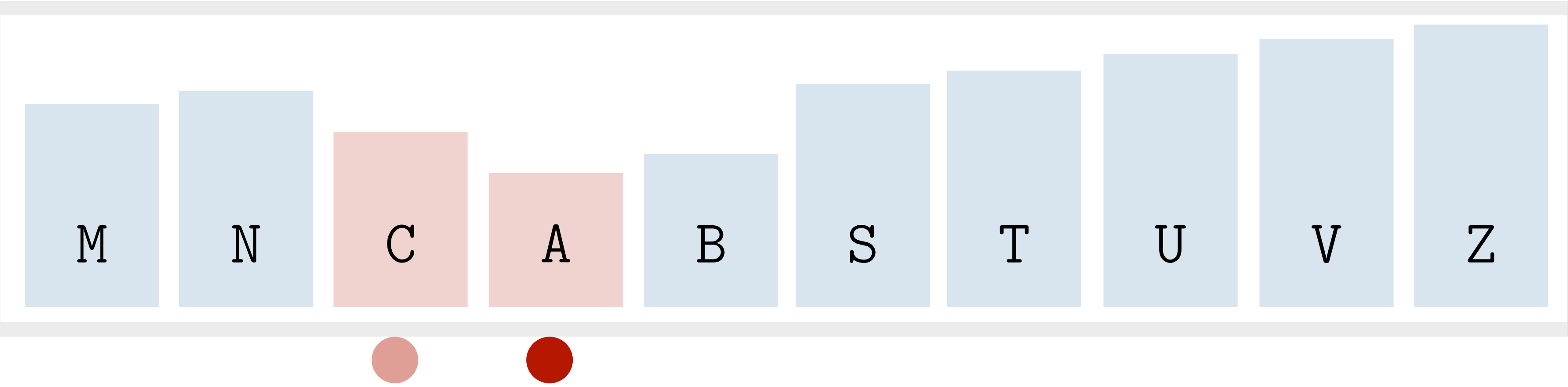
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



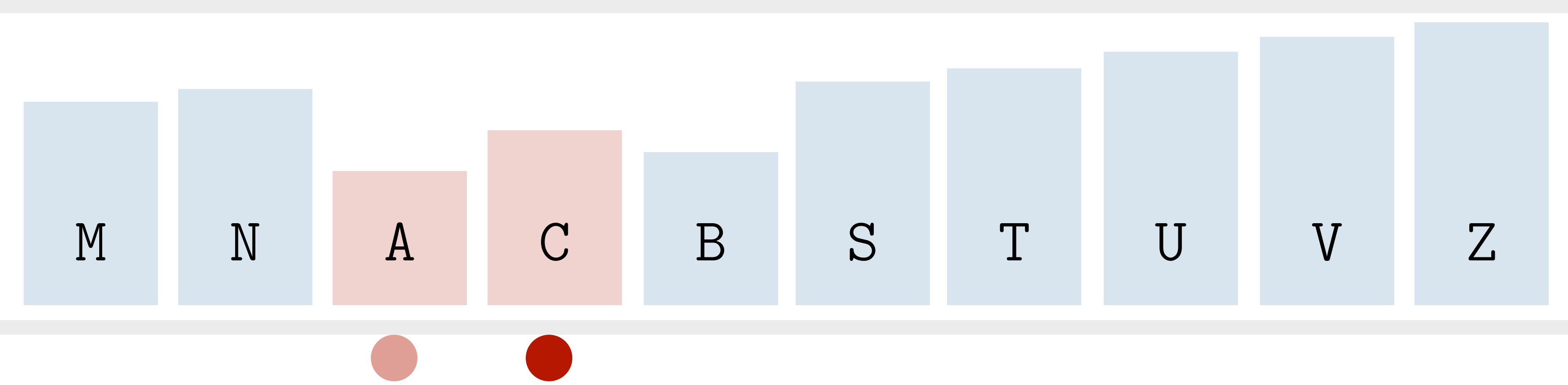
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



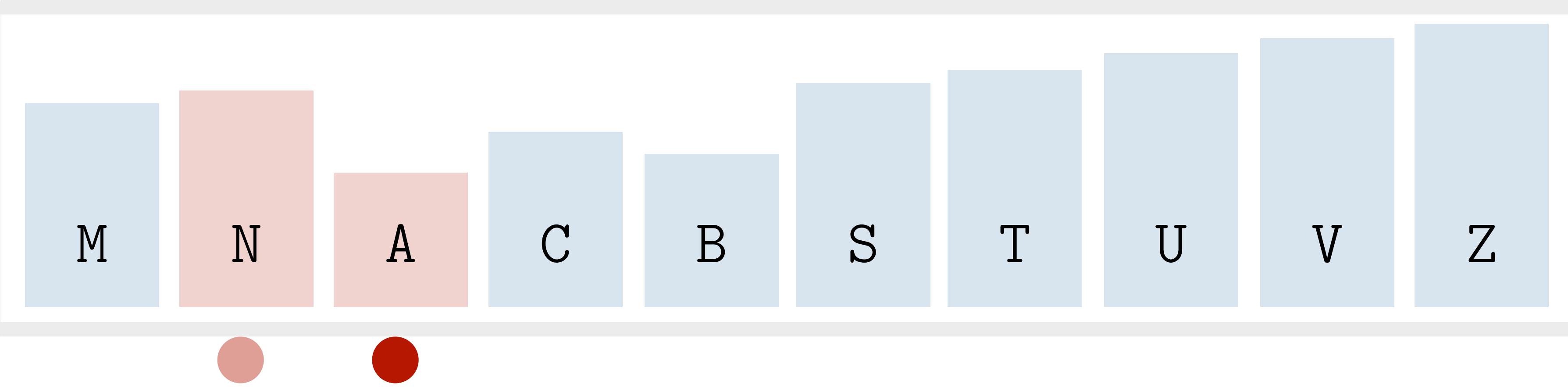
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



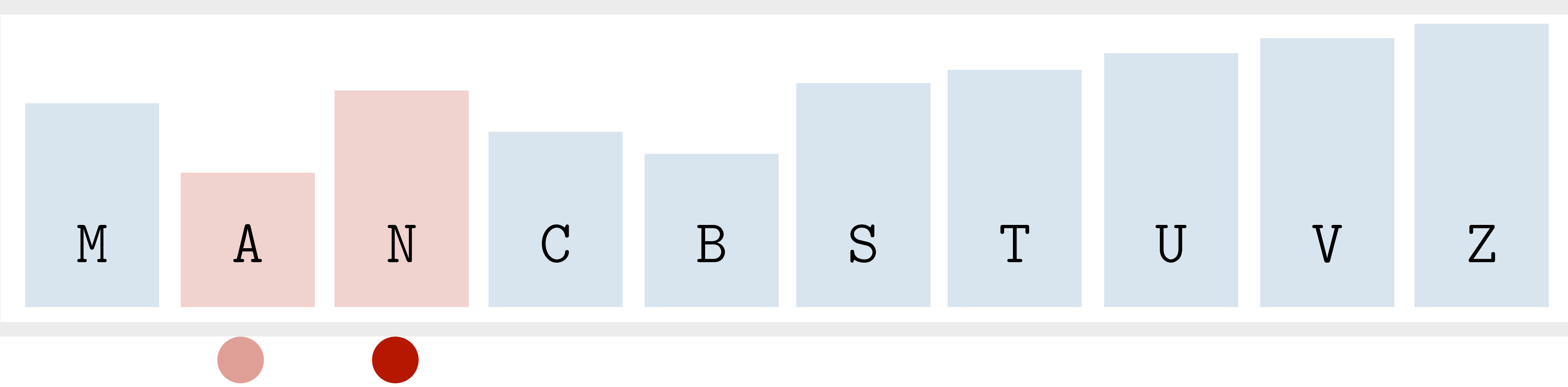
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



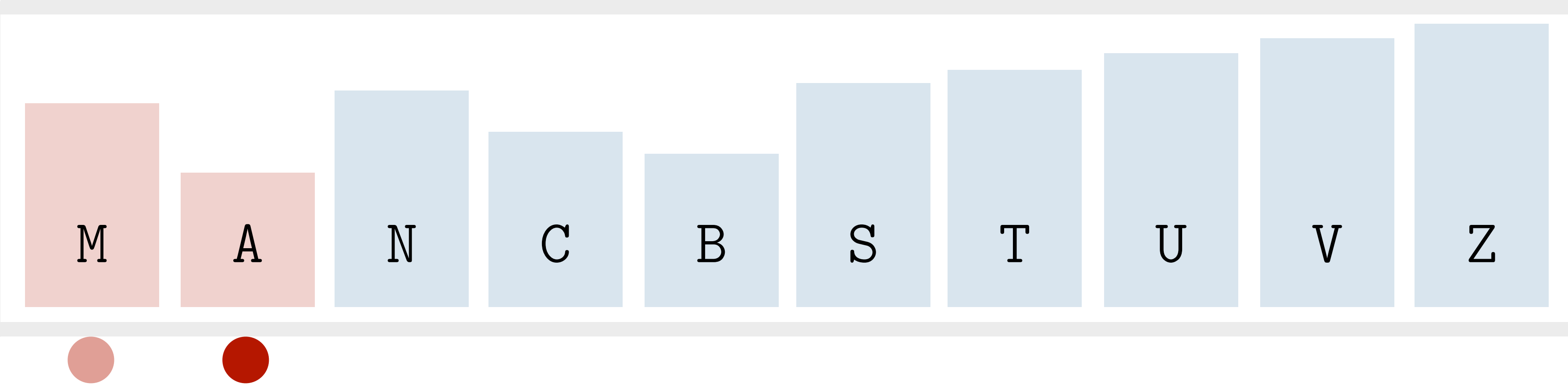
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



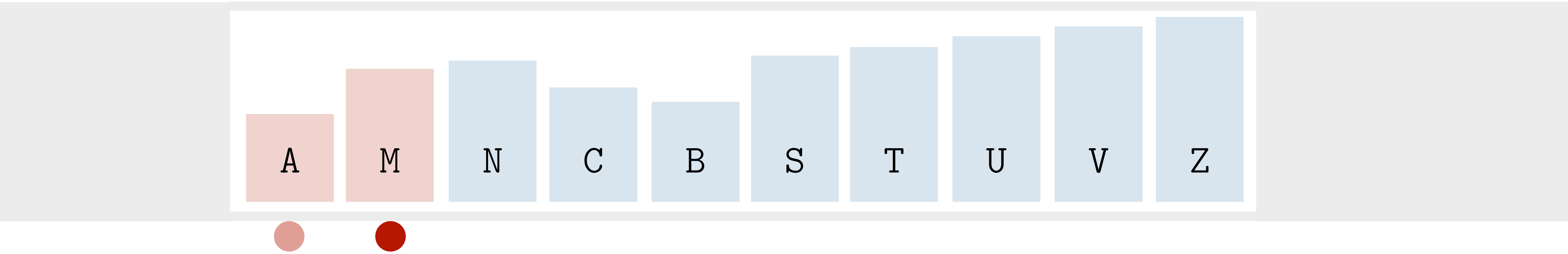
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



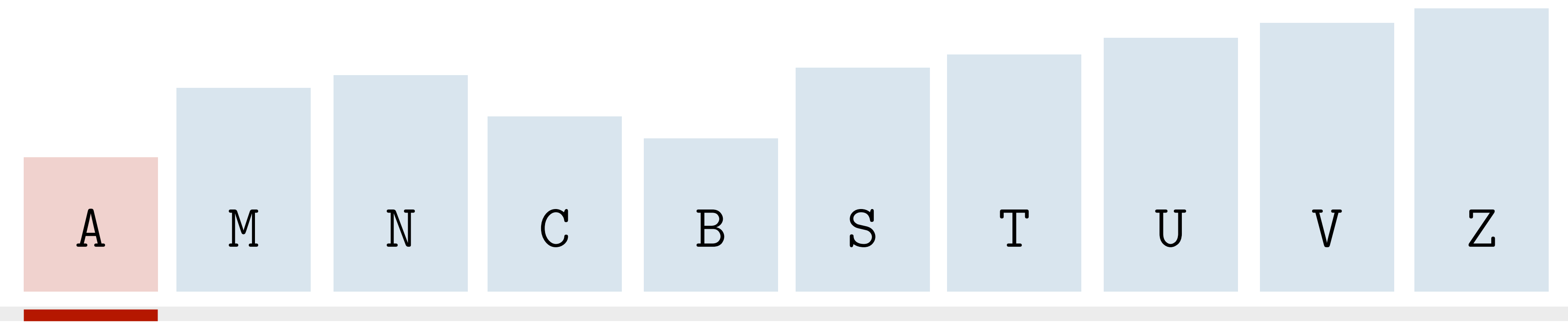
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



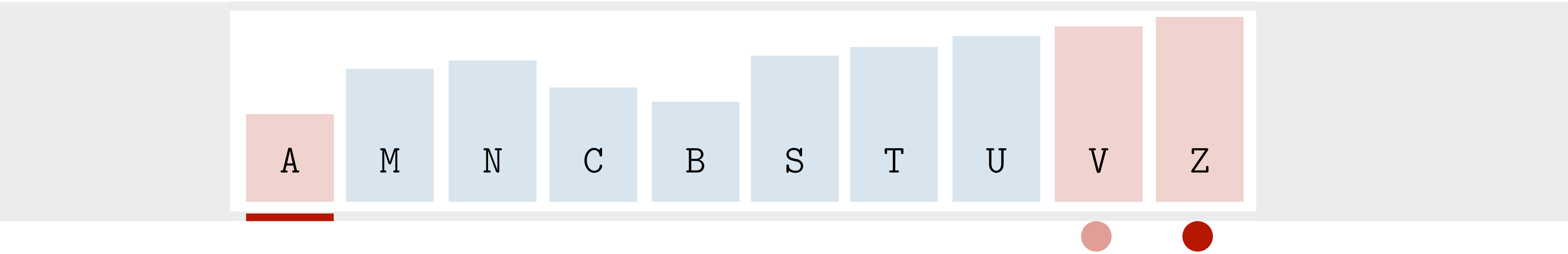
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



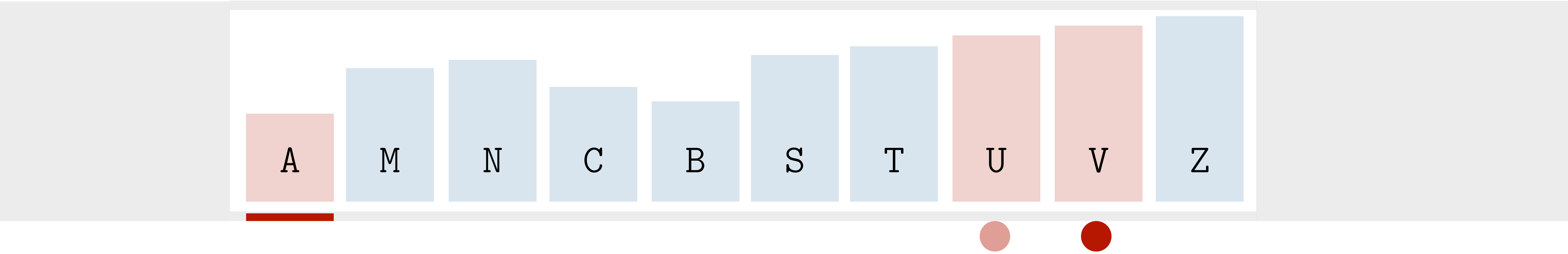
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



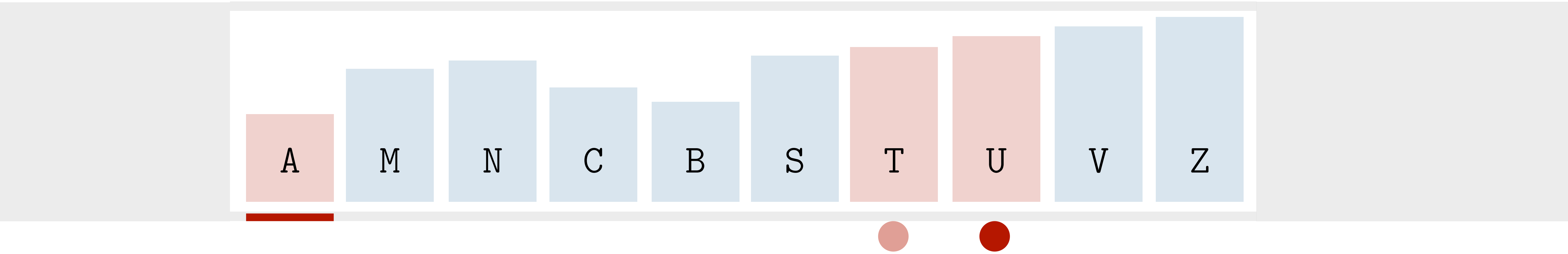
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



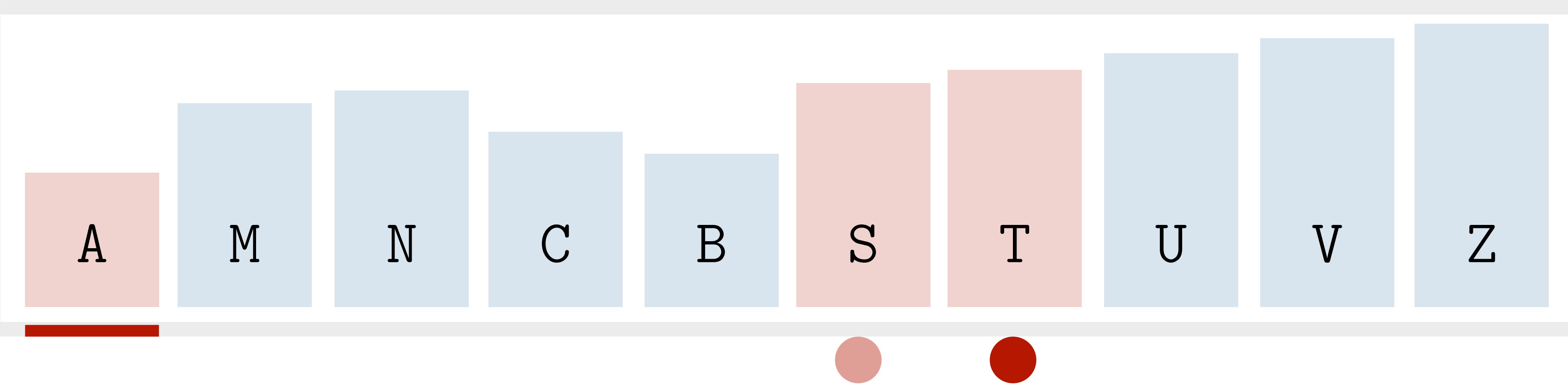
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



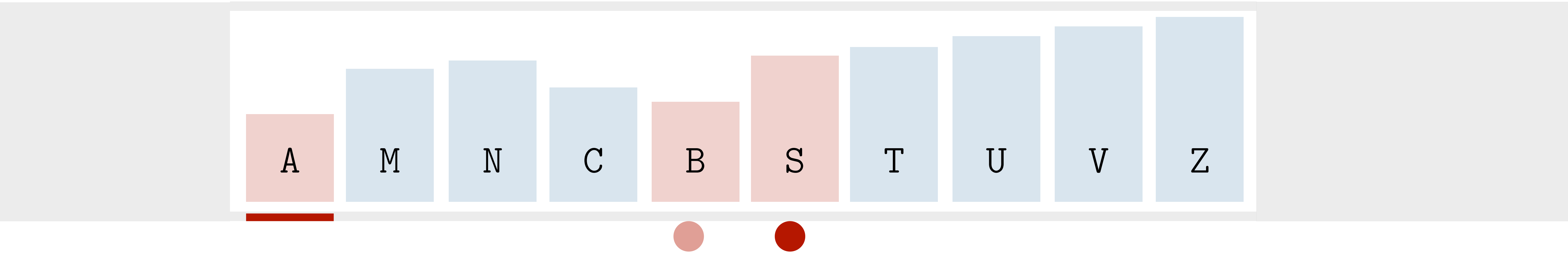
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



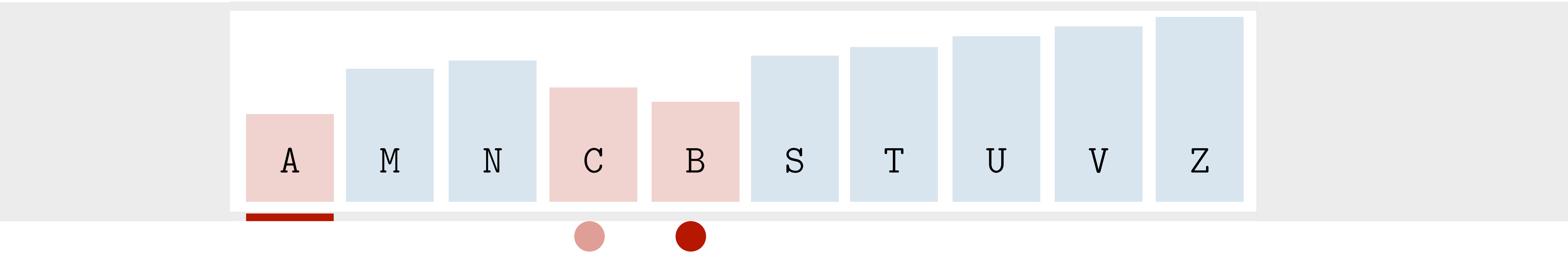
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



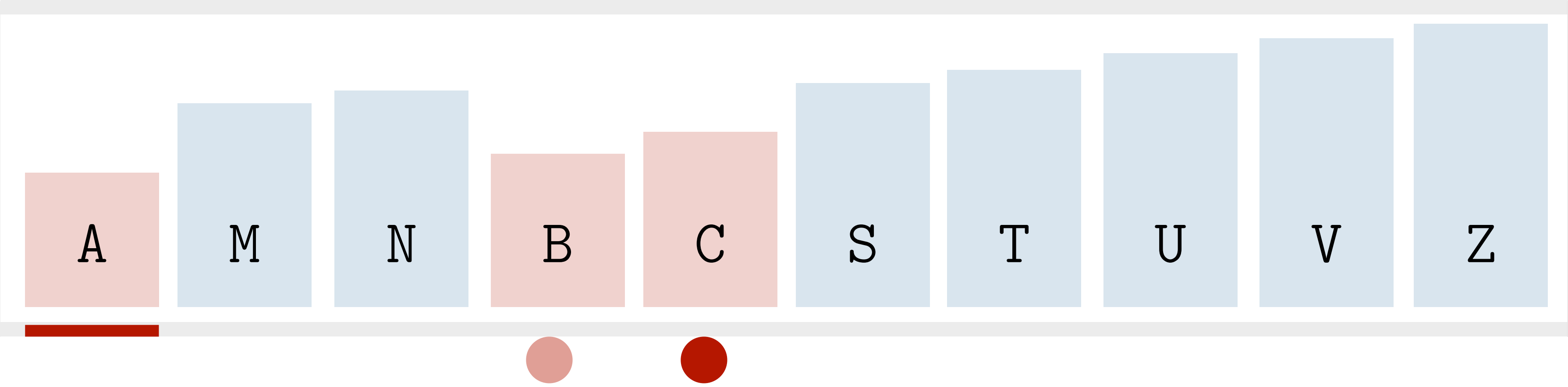
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



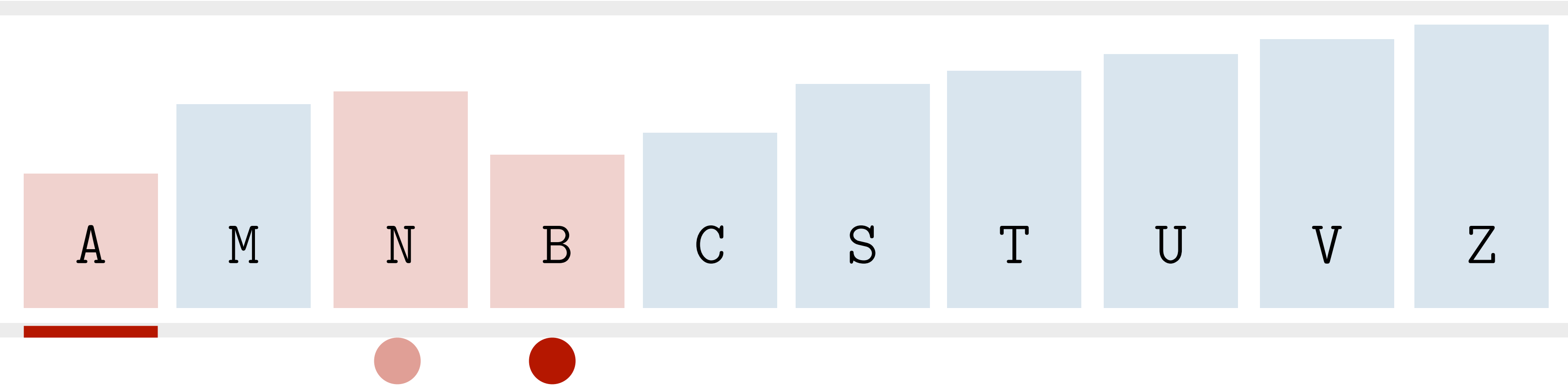
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



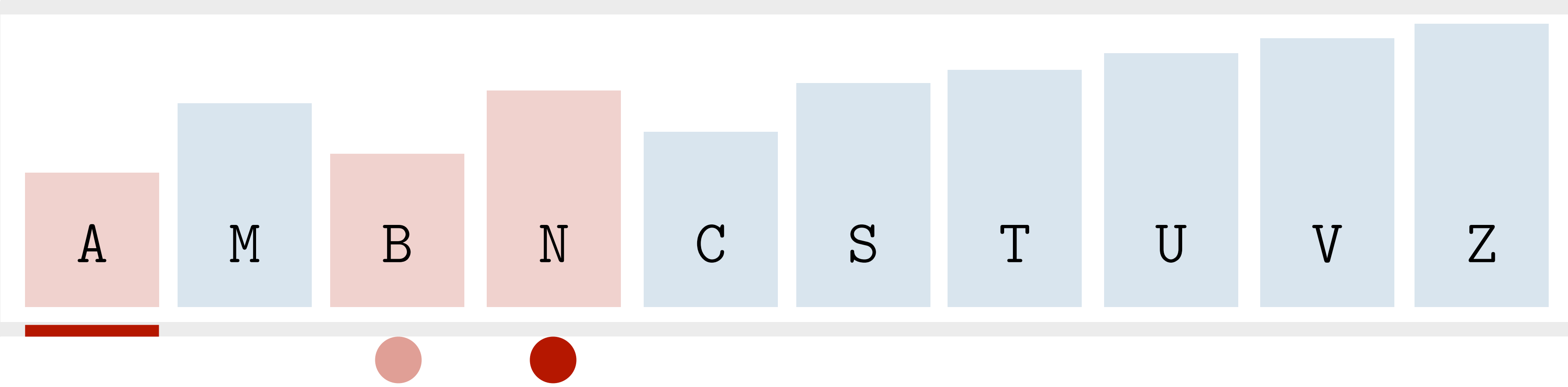
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



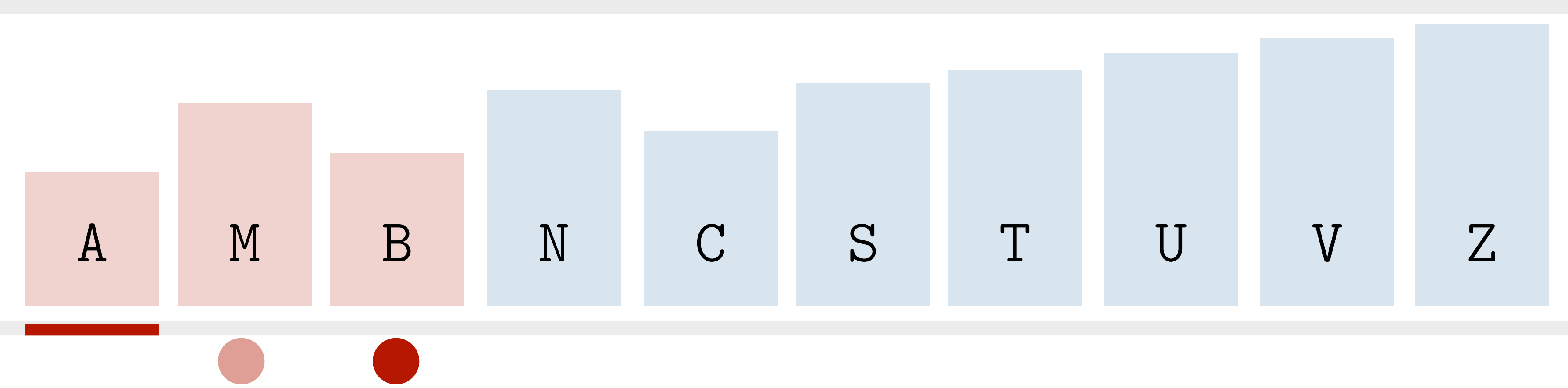
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



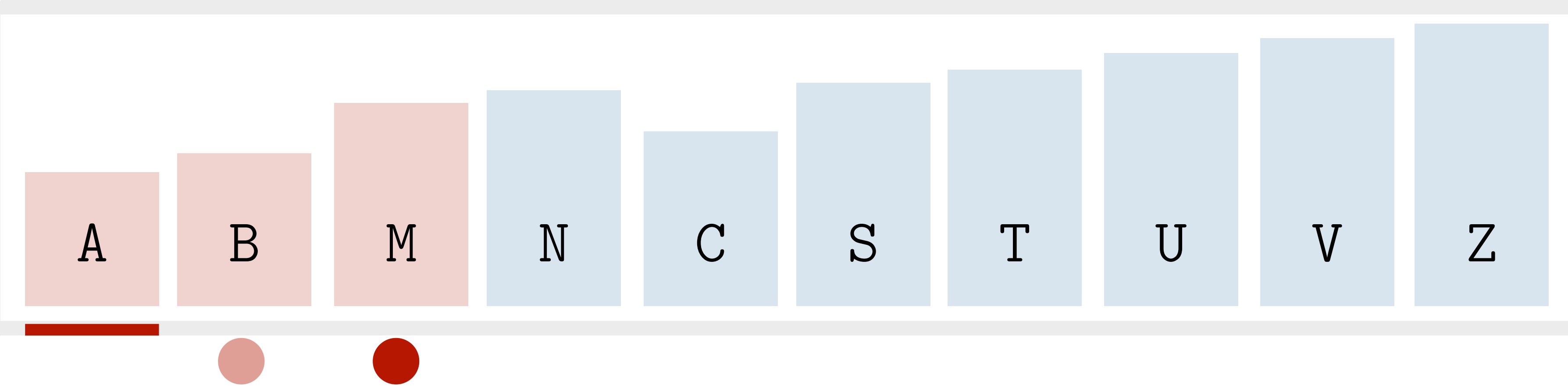
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



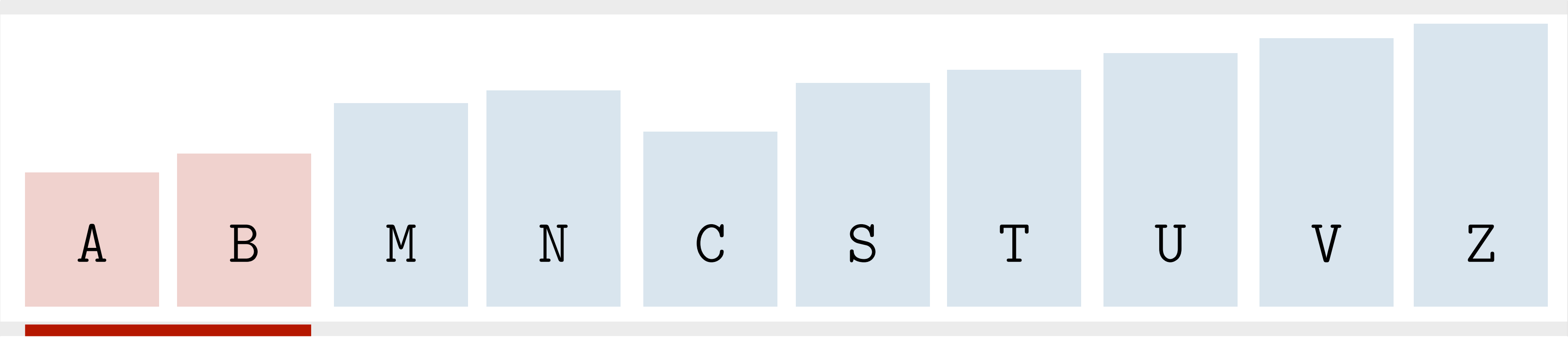
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



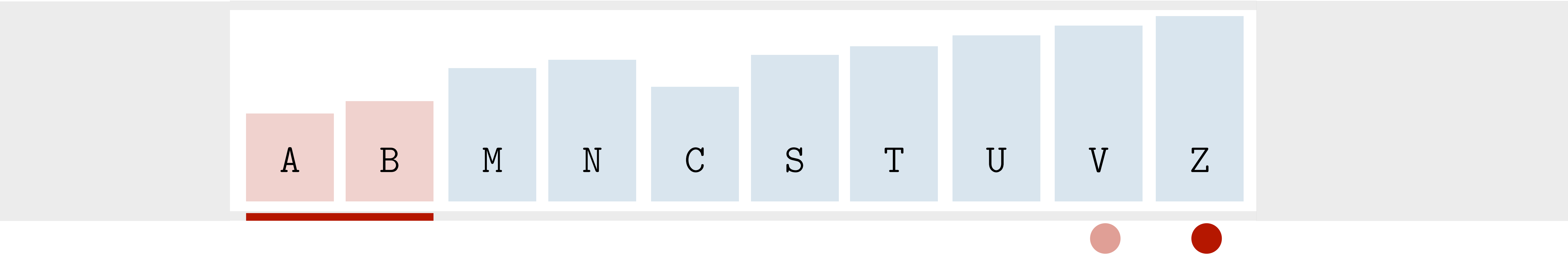
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



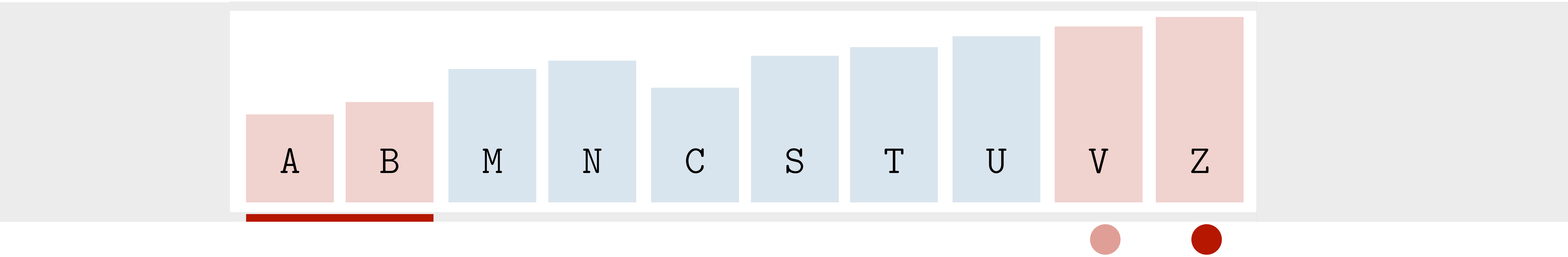
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



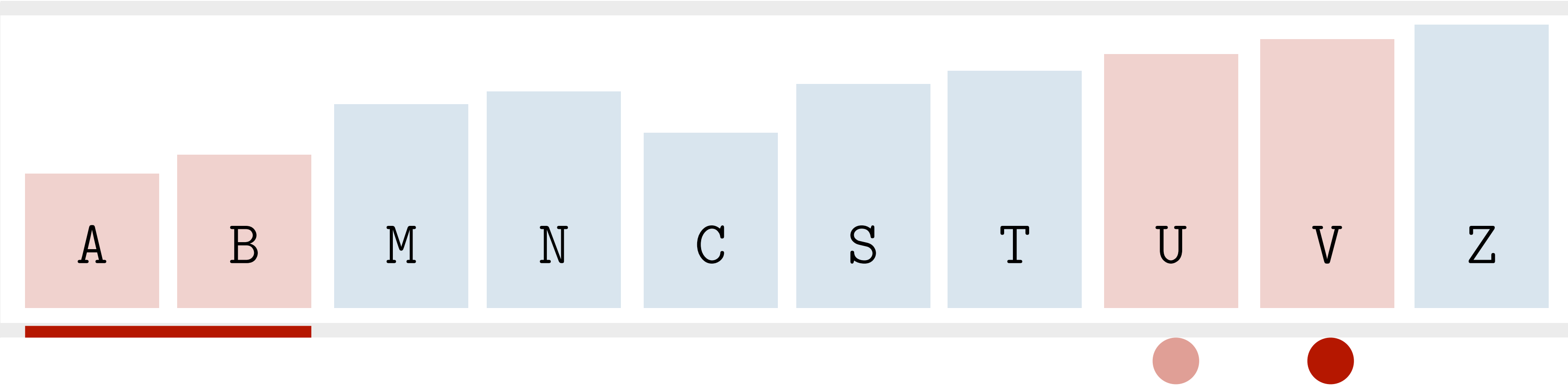
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



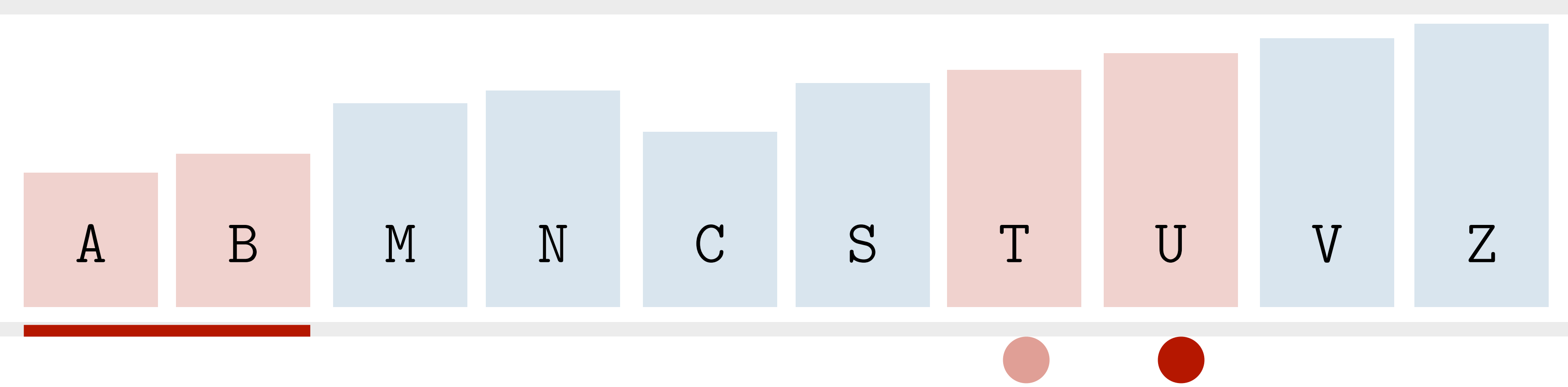
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



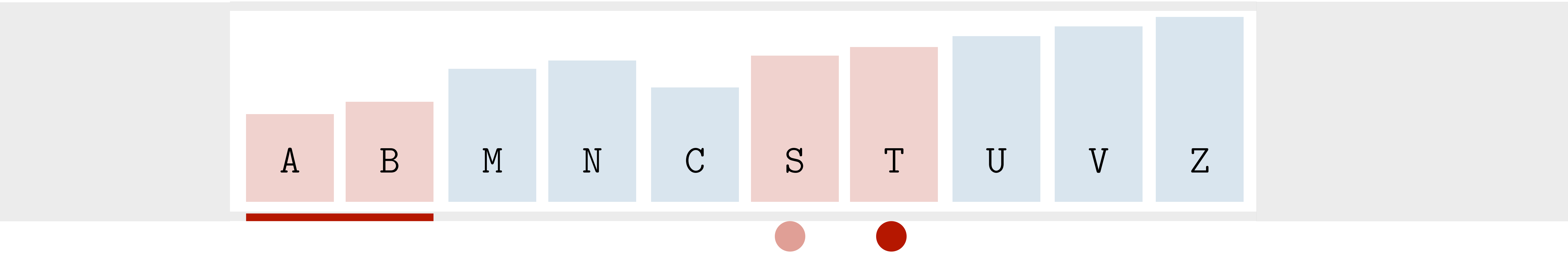
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



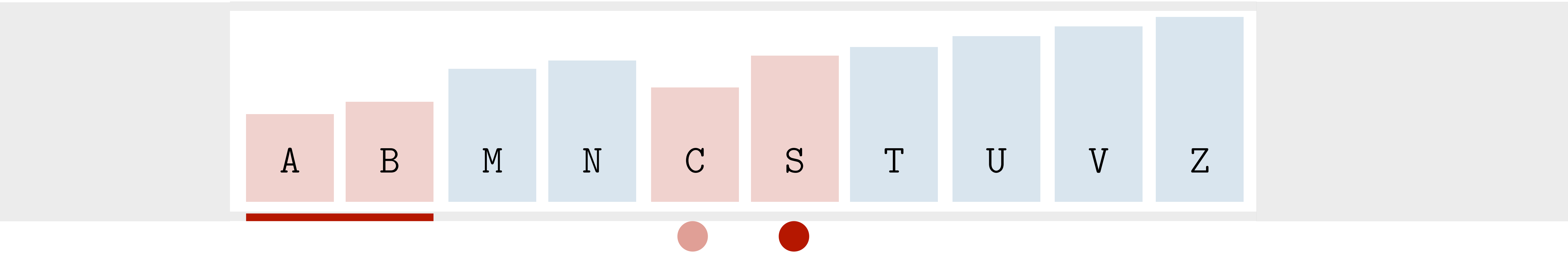
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



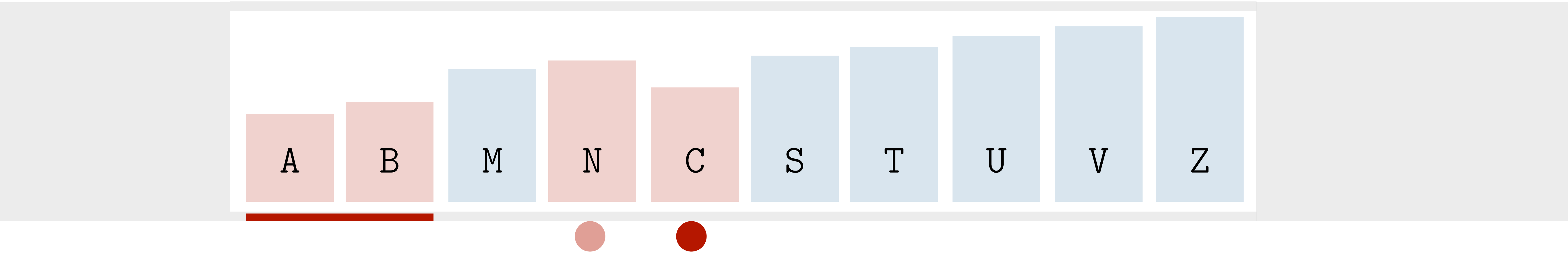
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



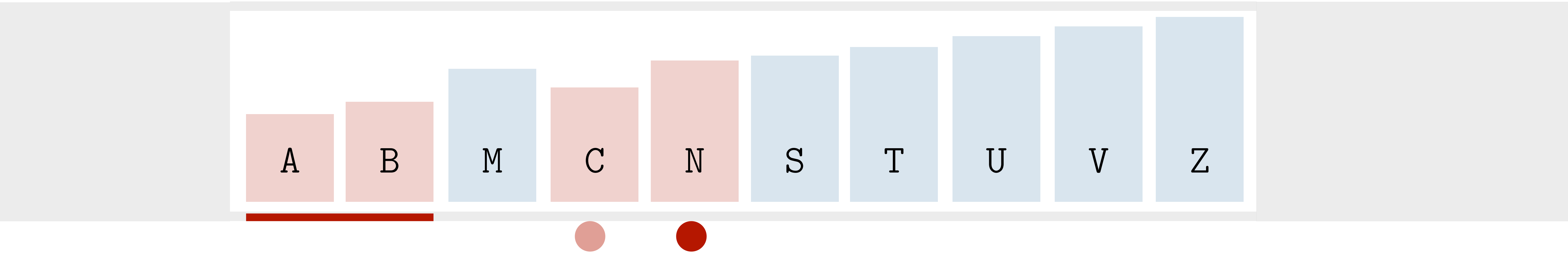
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



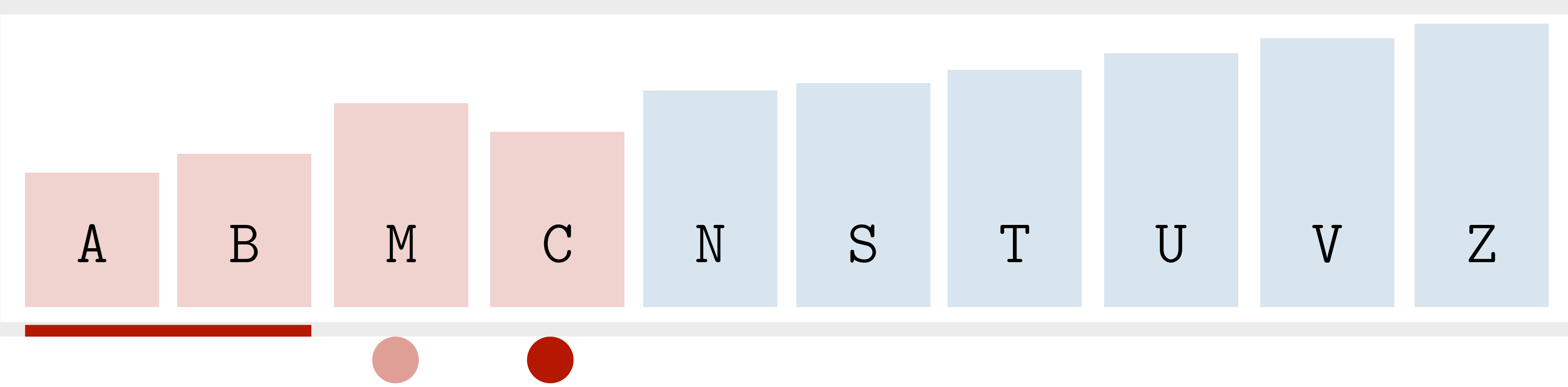
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



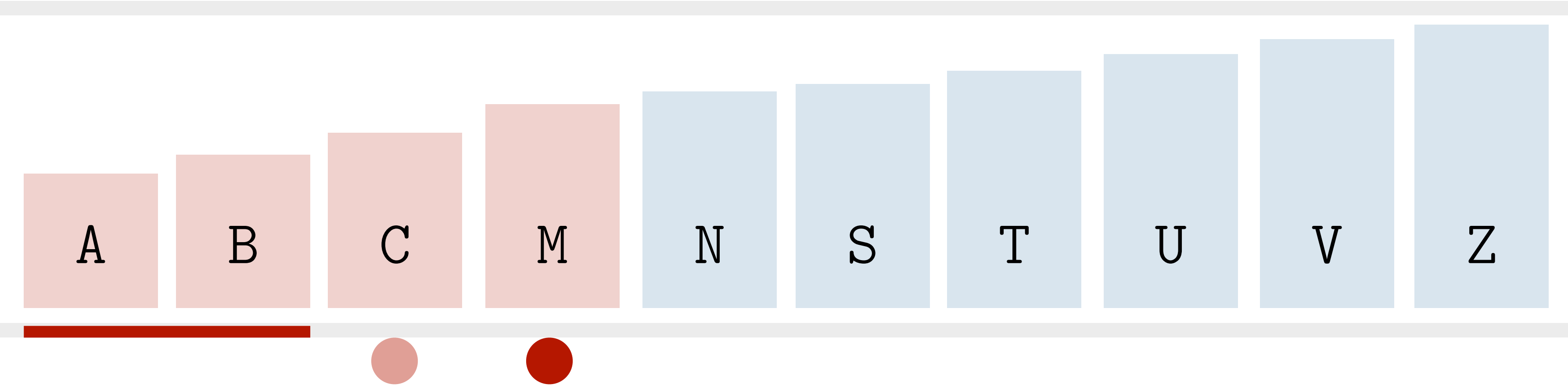
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



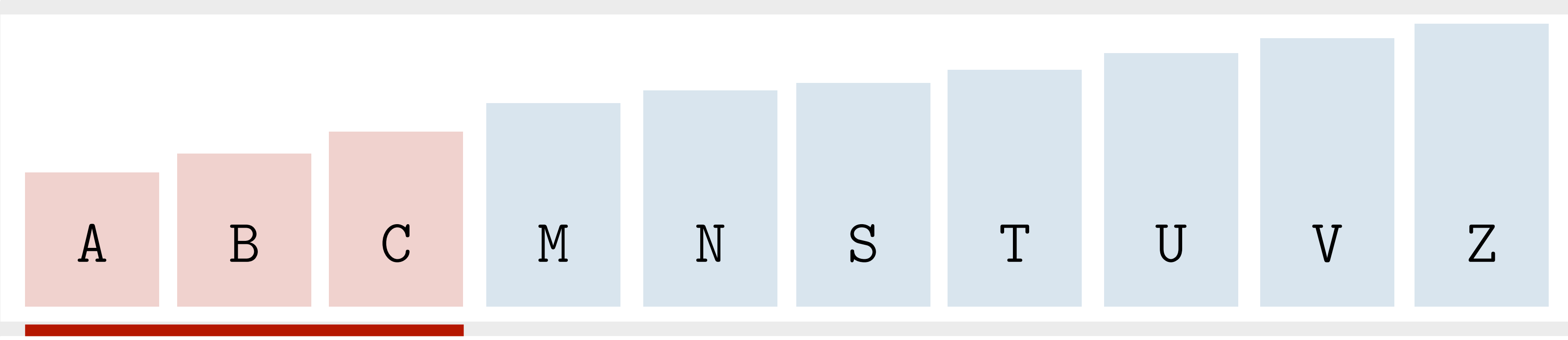
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



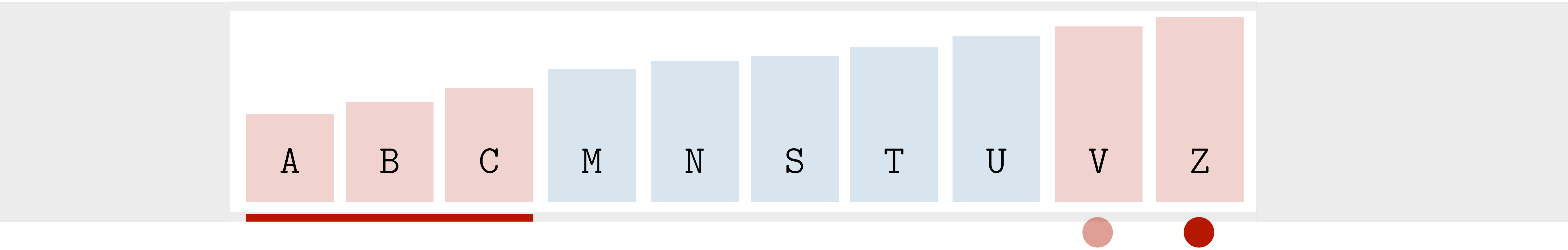
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



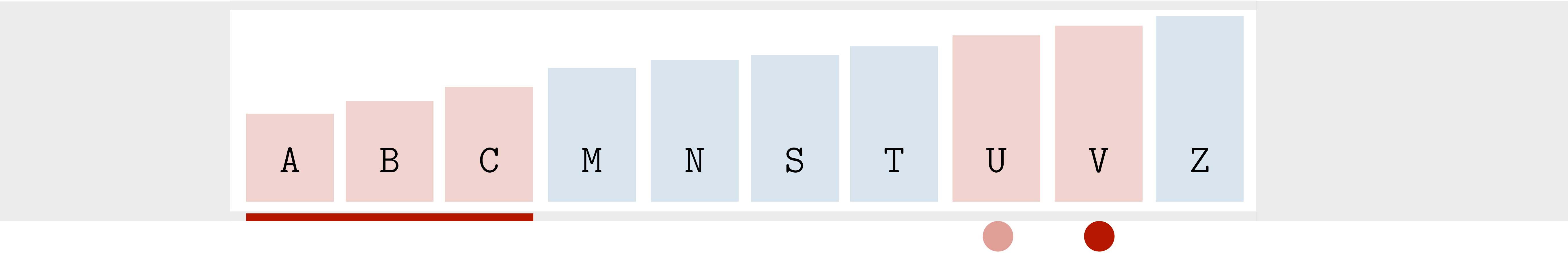
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



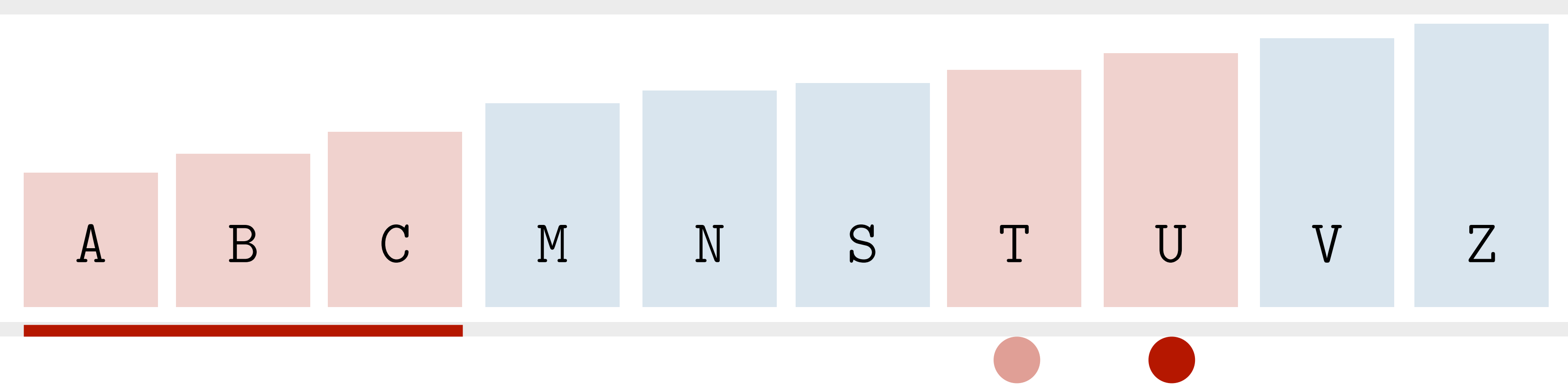
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



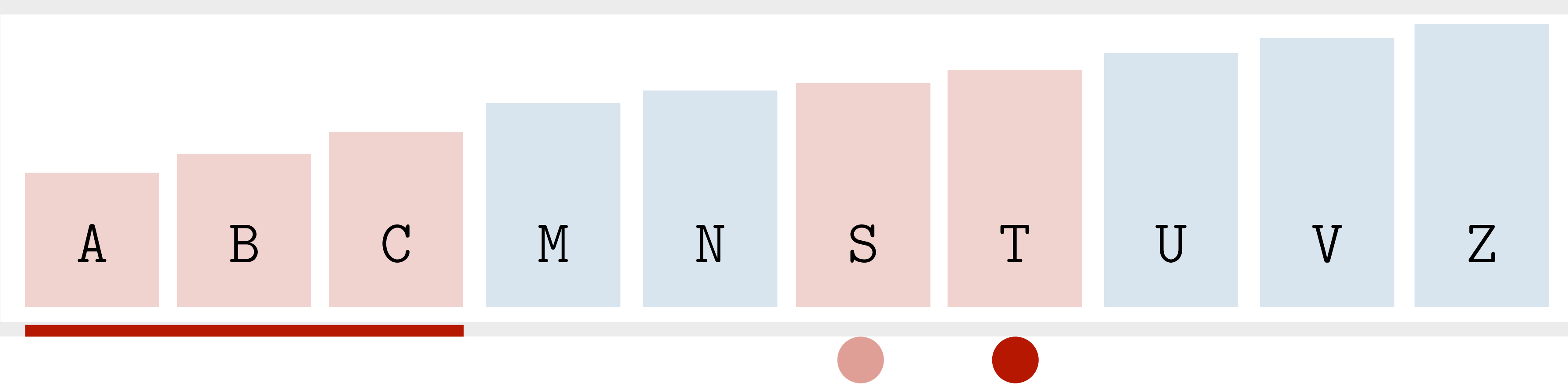
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



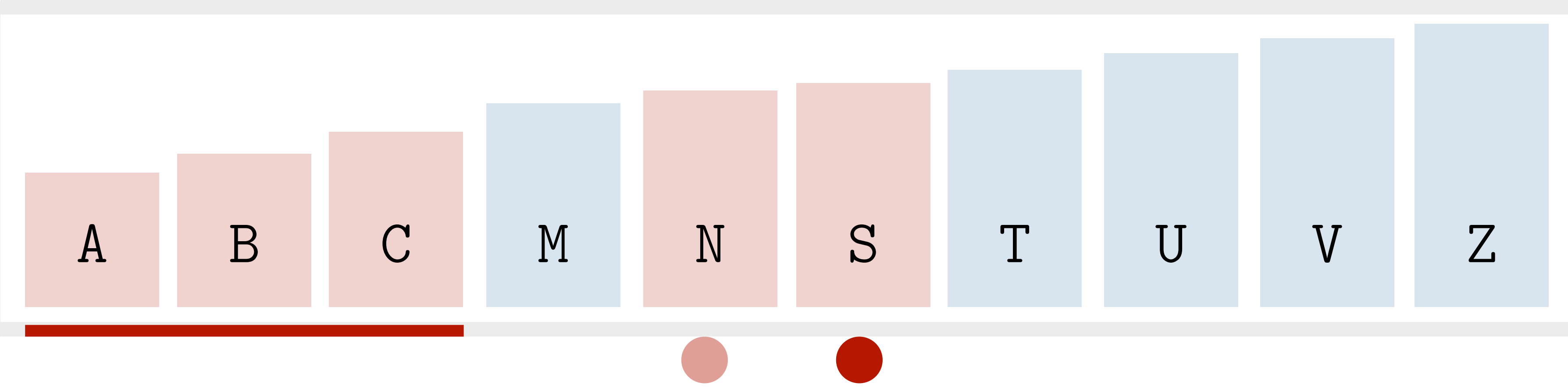
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



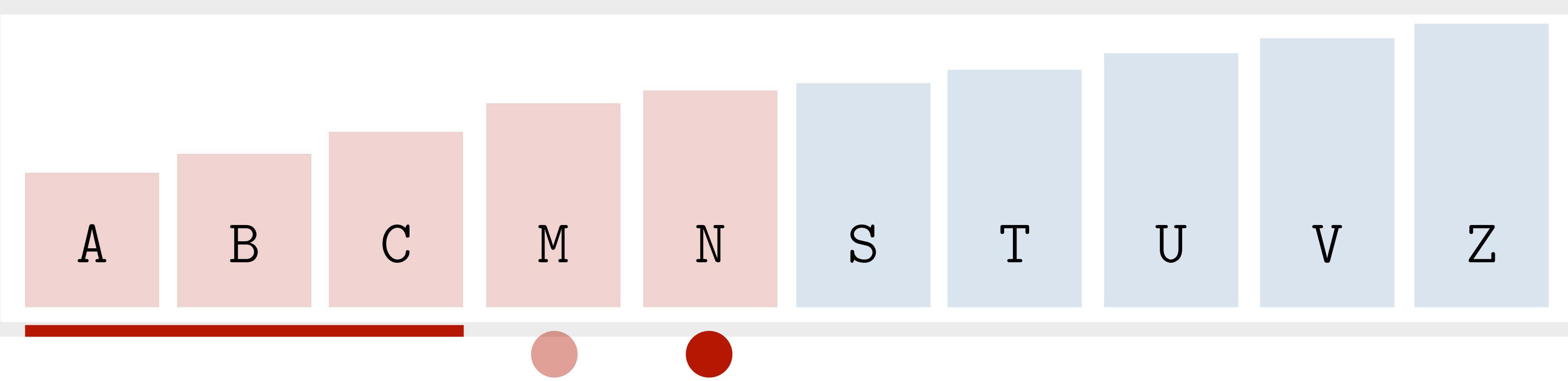
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



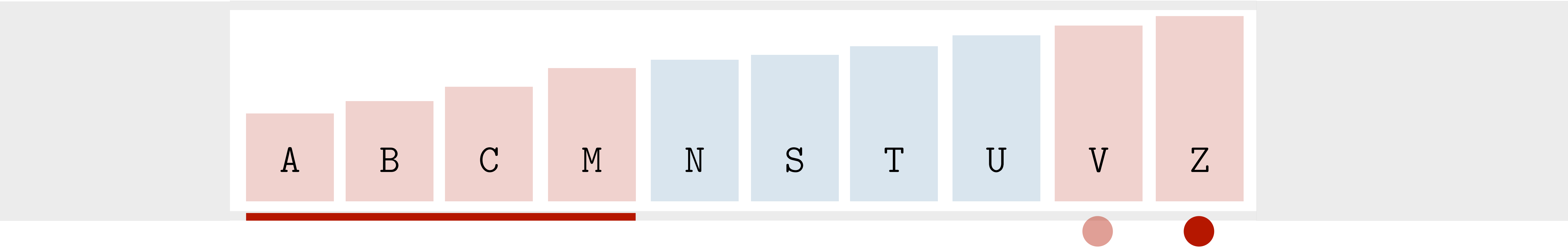
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



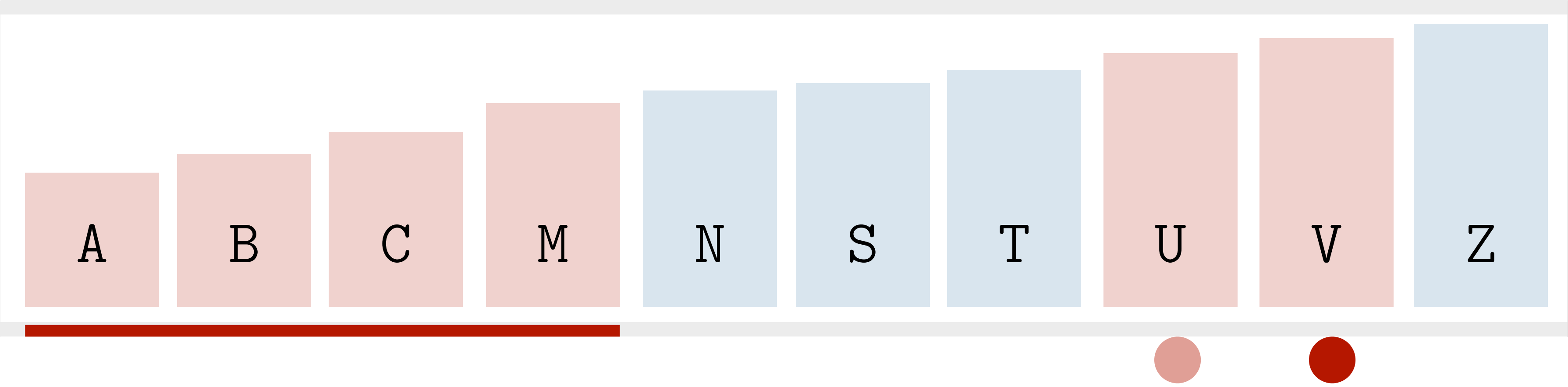
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



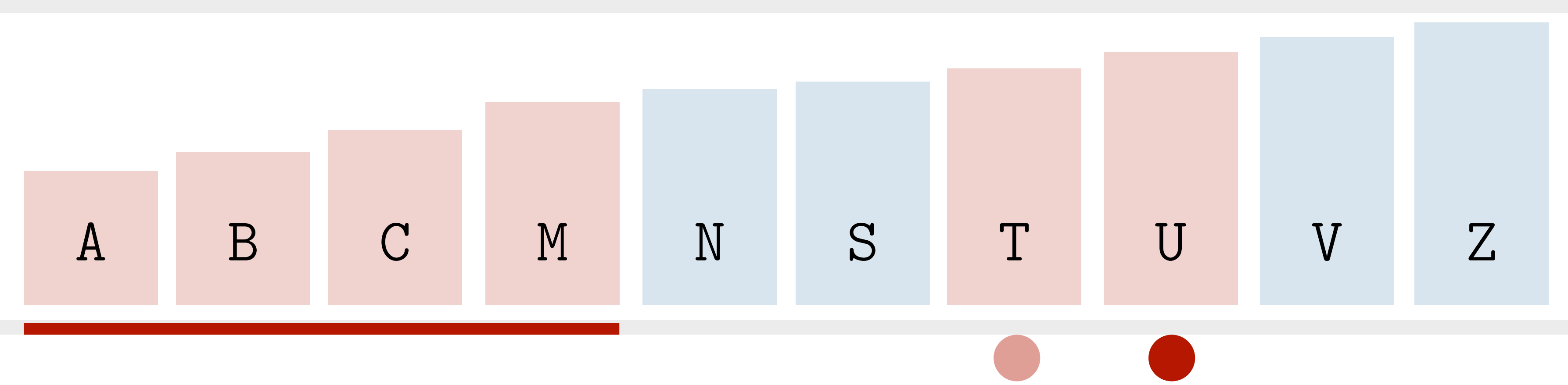
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



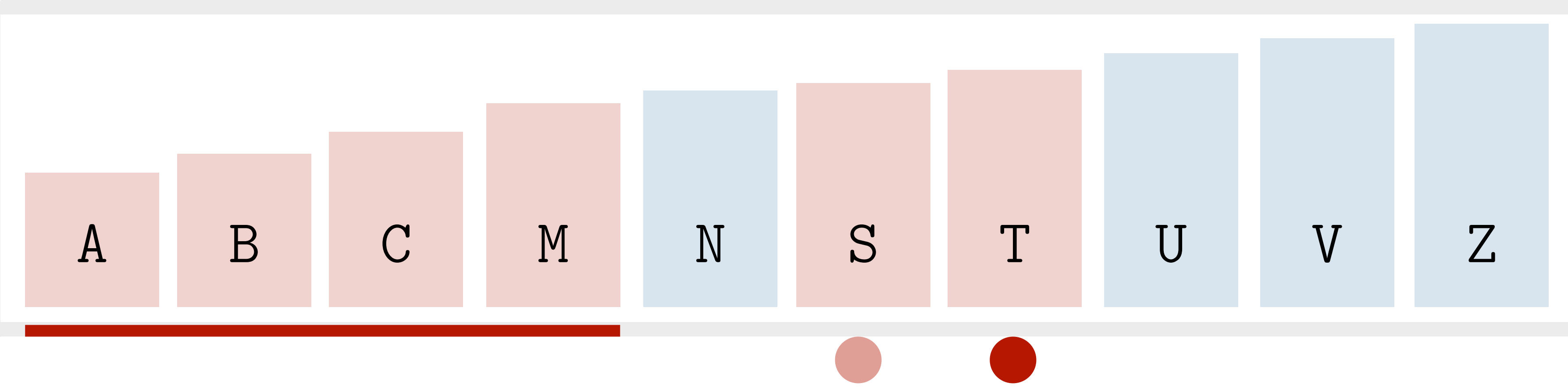
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



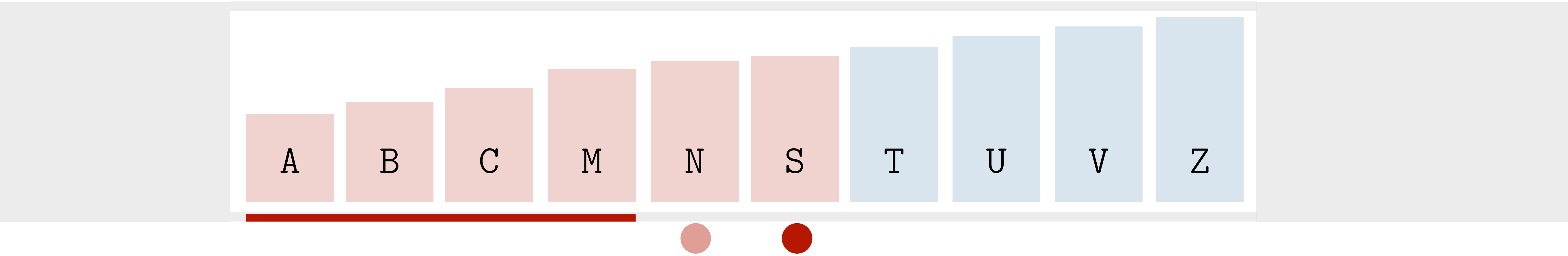
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



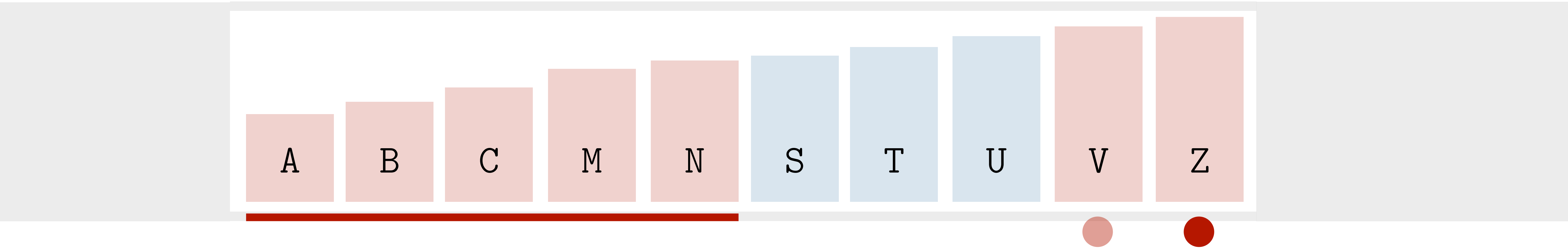
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



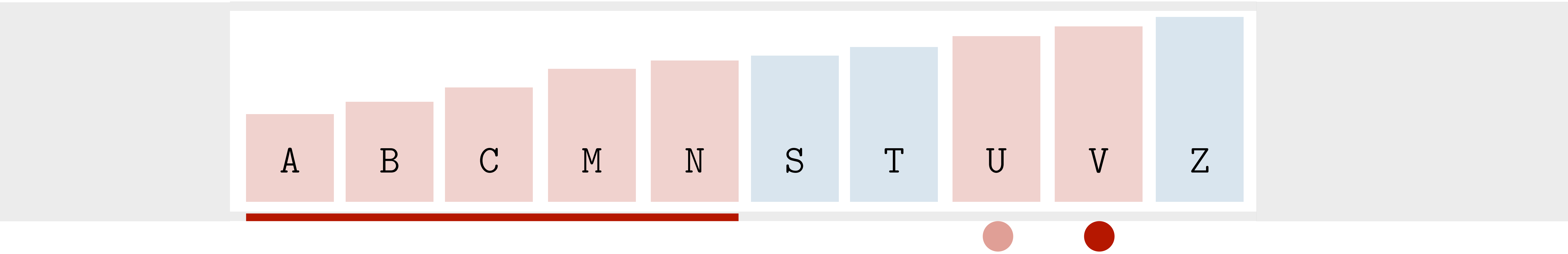
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



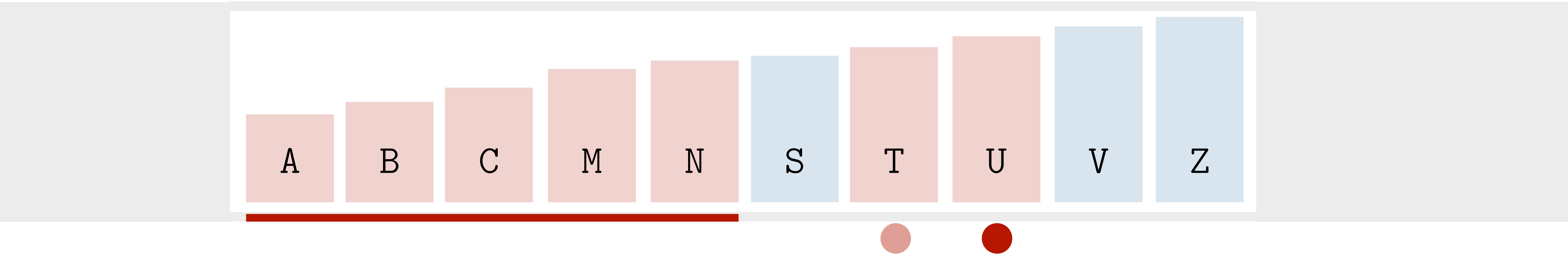
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



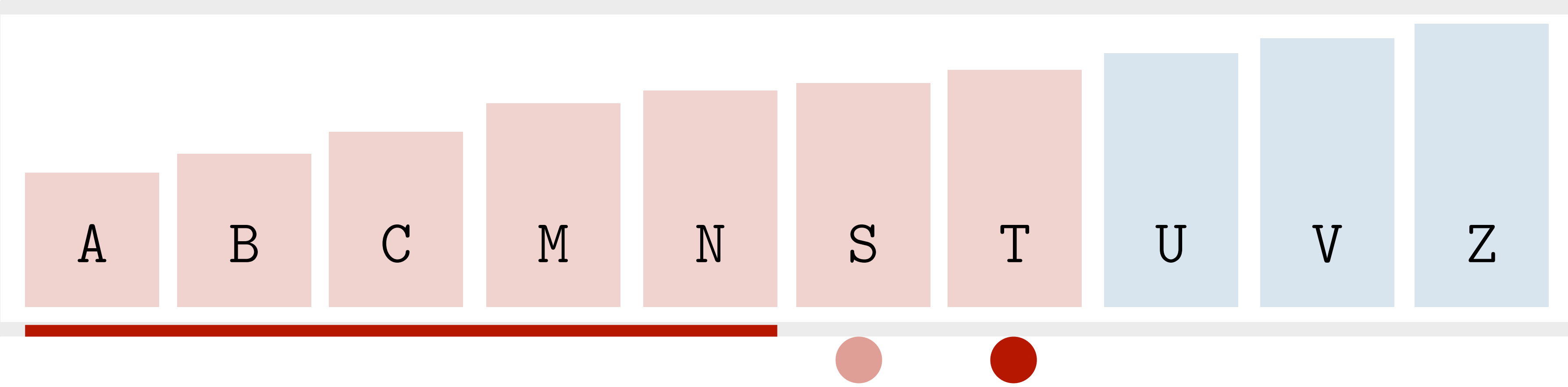
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



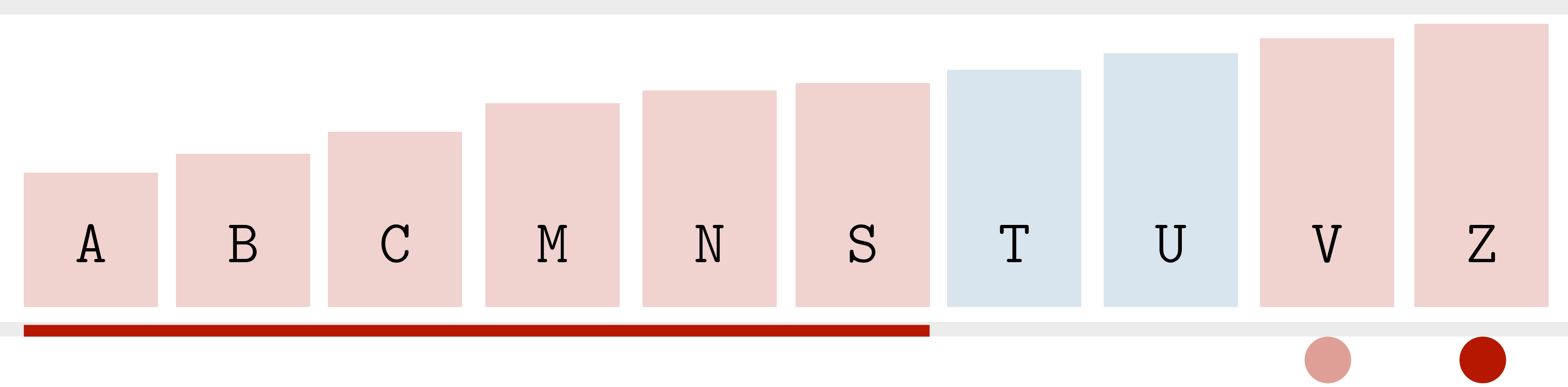
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



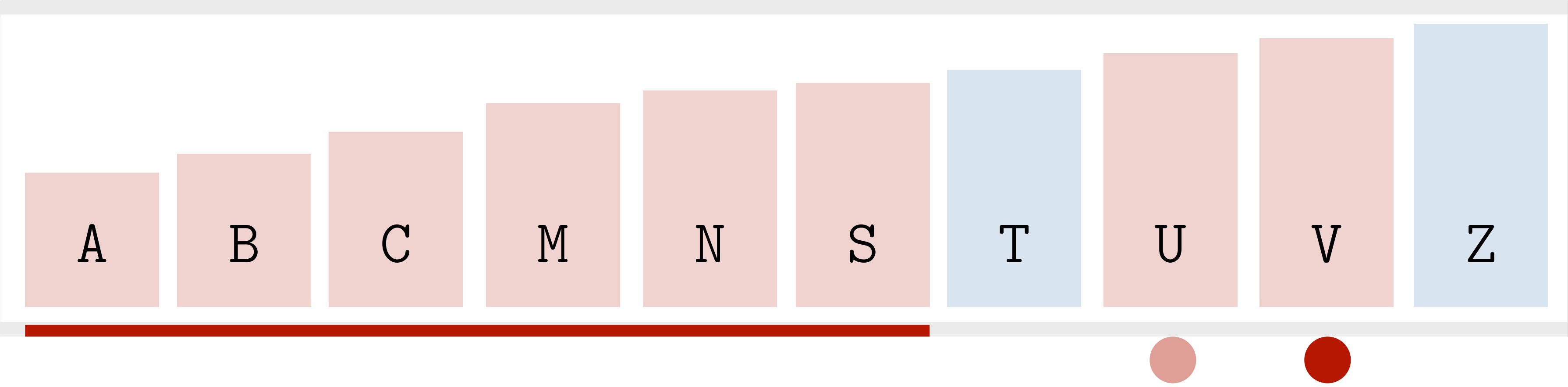
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



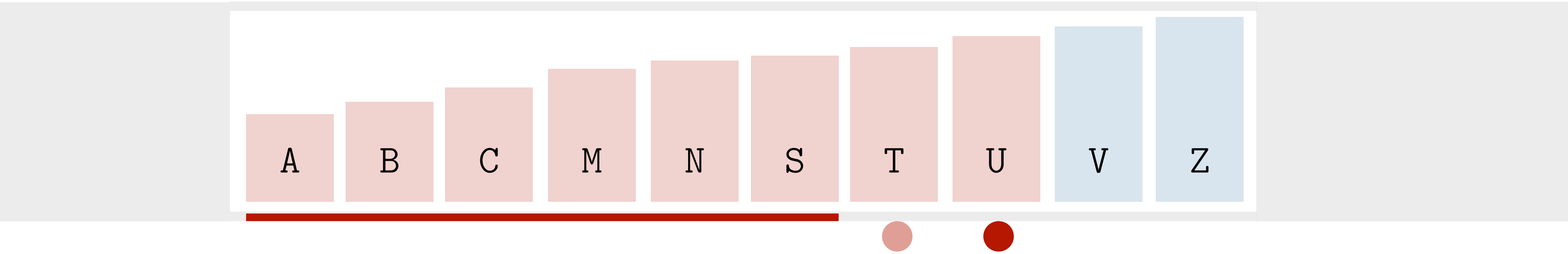
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



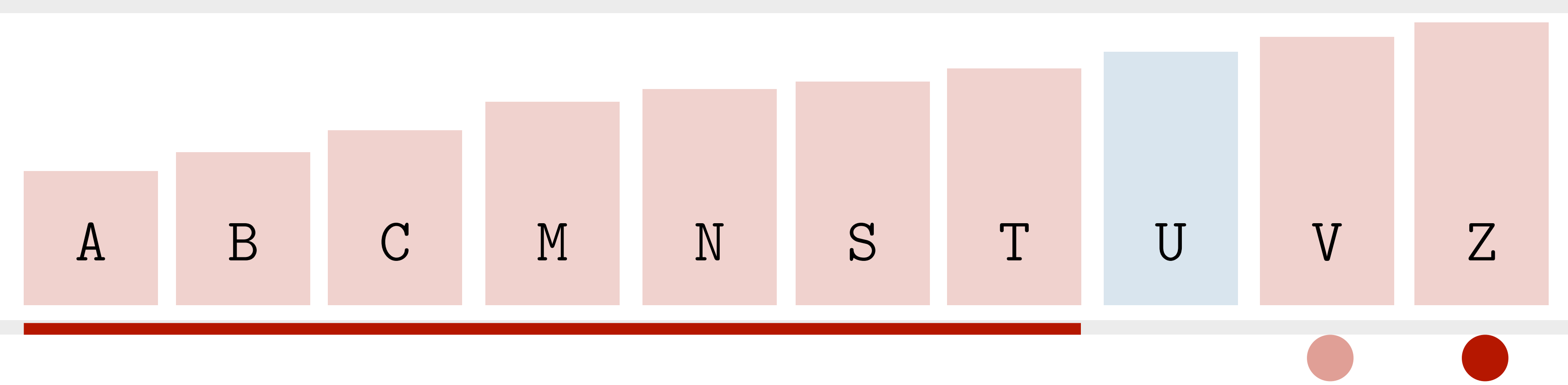
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



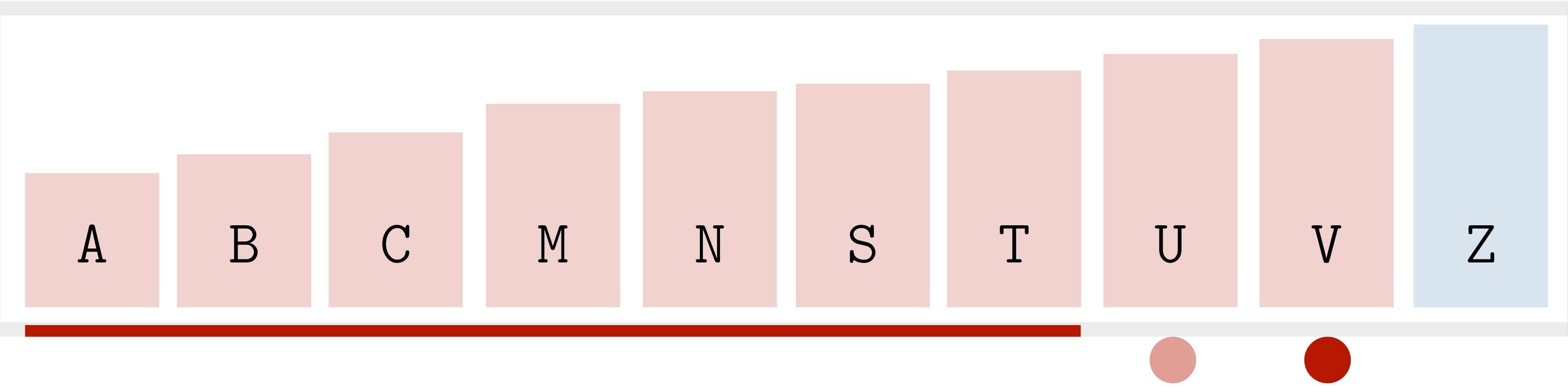
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



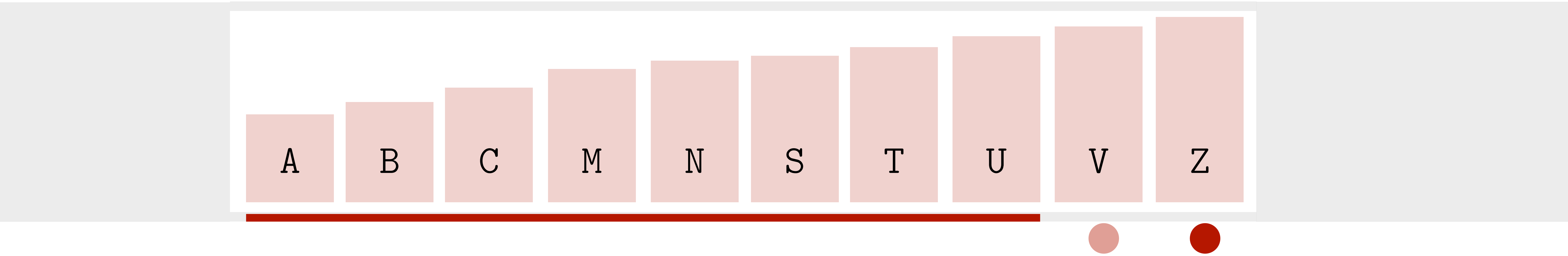
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



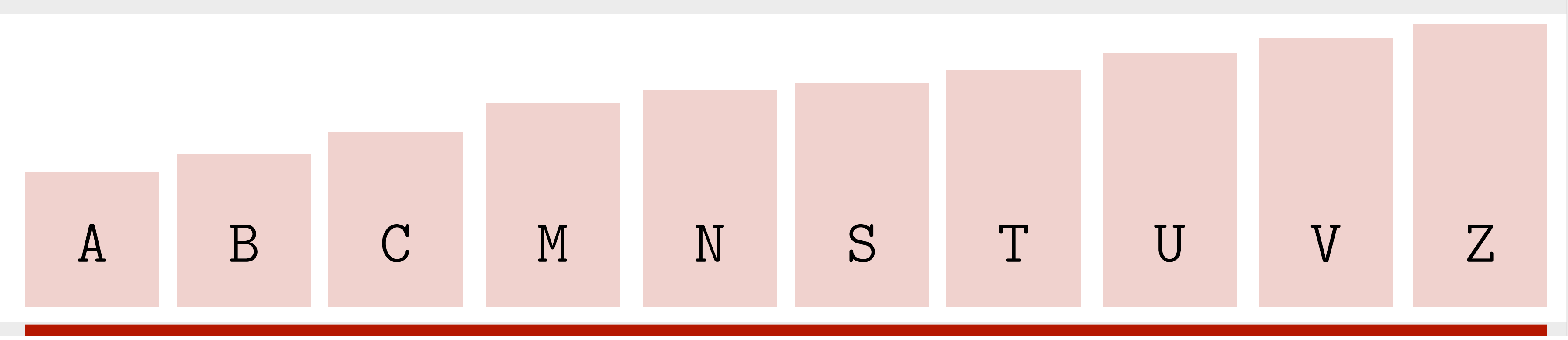
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



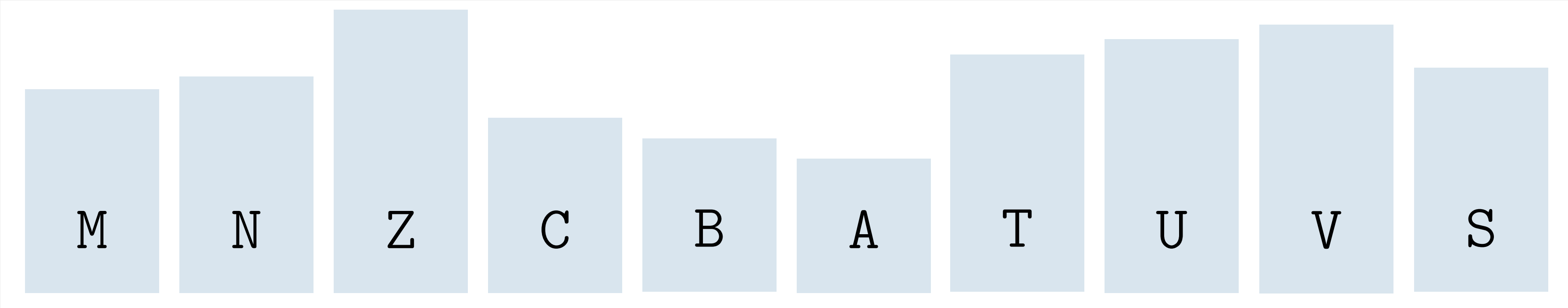
Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

Sorting Warmup

Problem. Sort a list of books alphabetically.
Restrictions. Can't place any book anywhere outside the shelf while sorting.



Which one is the best?

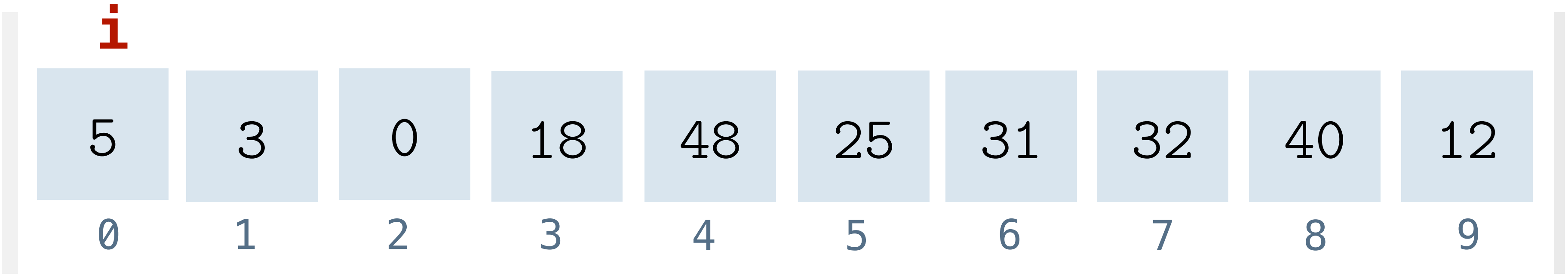
Let's count operations*!*

Idea 1. ***Select*** the min and place it in its correct position, then the second min, etc.

Idea 2. Go through each element and ***insert*** it in its correct position relative to its left.

Idea 3. ***Bubble*** Sort!

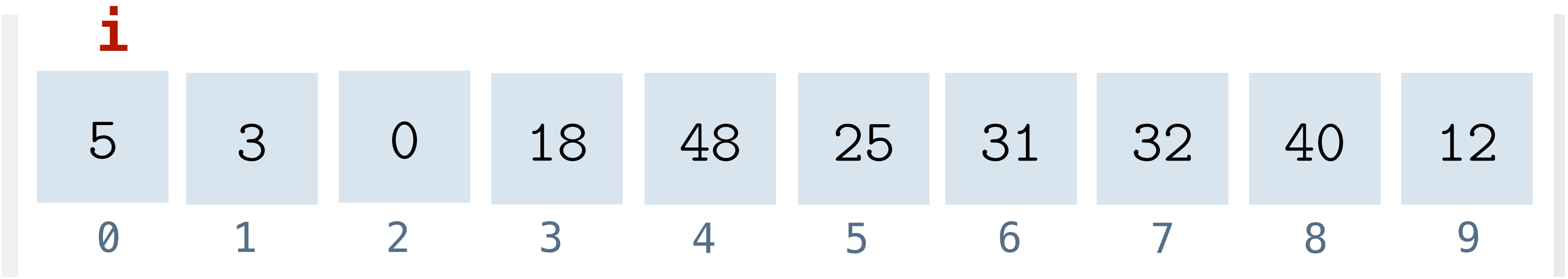
Selection Sort: Implementation



i									
5	3	0	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0;          ; i++) {

           find the index of the
           minimum in a[i, n-1]

           place the minimum
           in its right position

    }
}
```

Selection Sort: Implementation

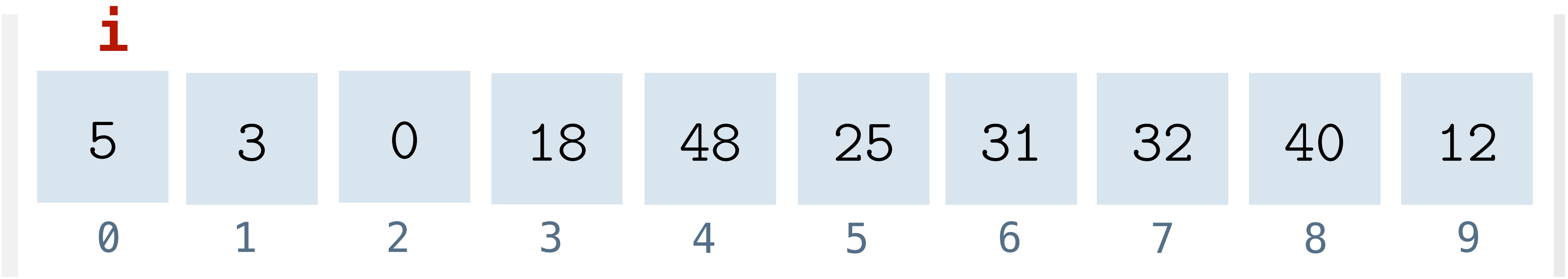


5	3	0	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {

        find the index of the
        minimum in a[i, n-1]

        place the minimum
        in its right position

    }
}
```

Search for the minimum *n*−1 times

Selection Sort: Implementation

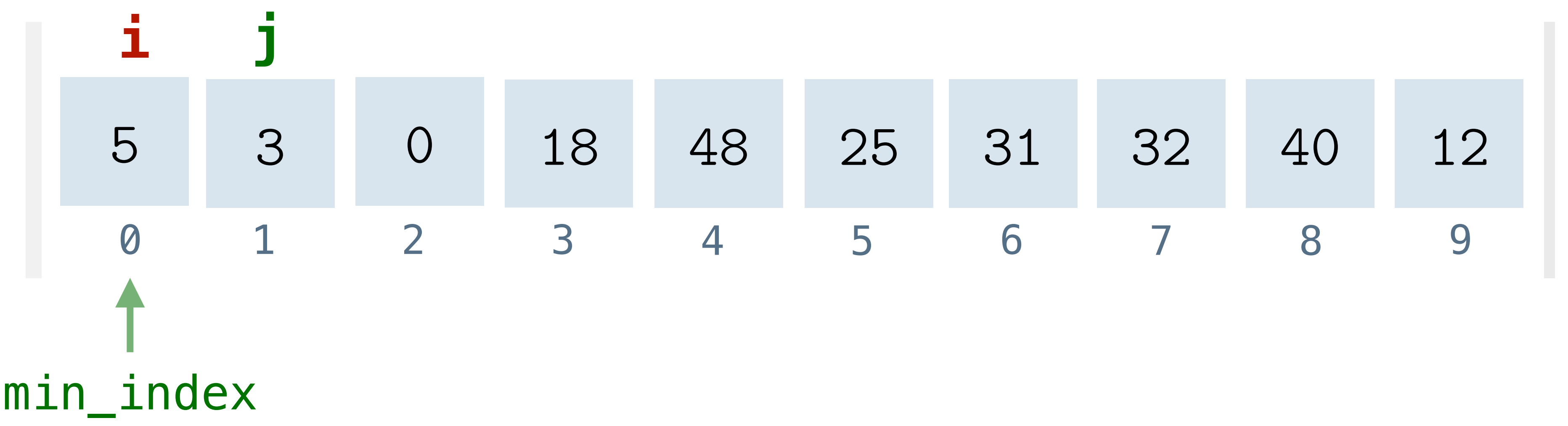


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {



    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Selection Sort: Implementation

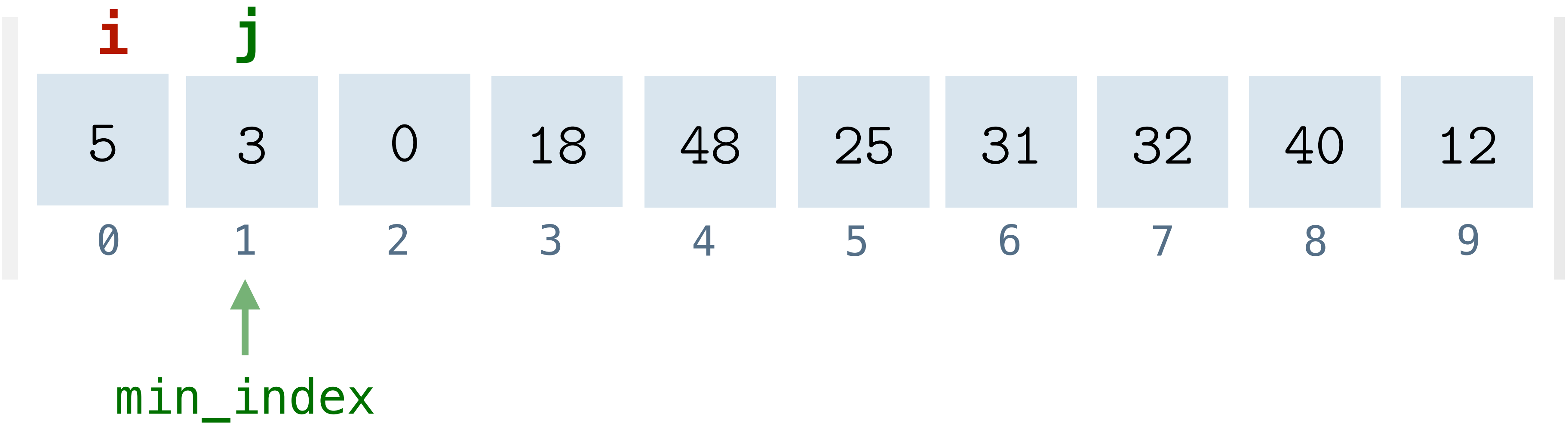


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Selection Sort: Tracing

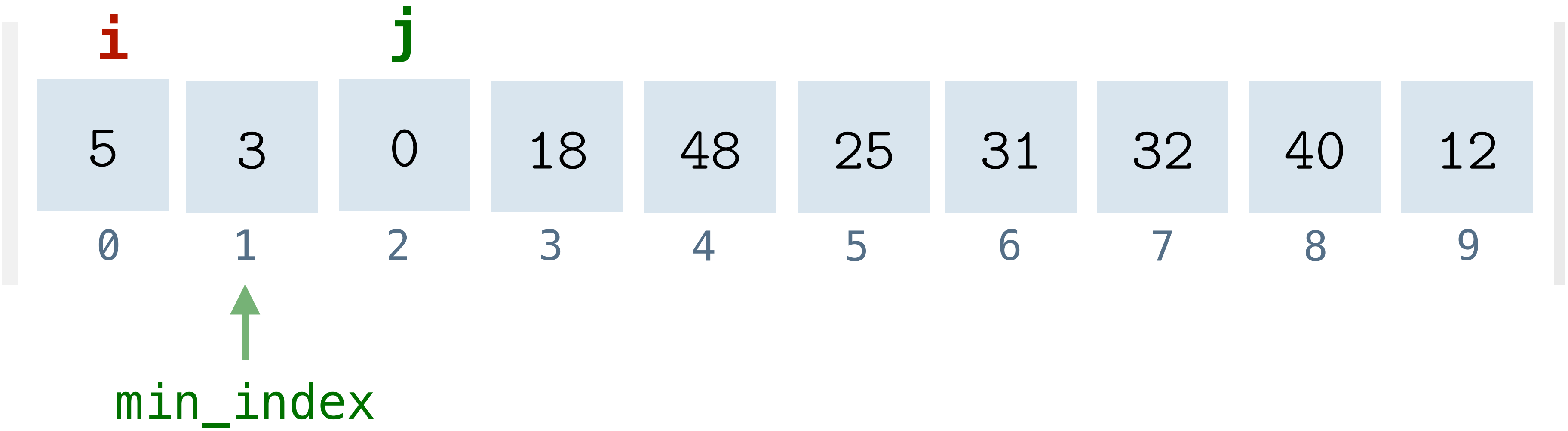


0	1	2	3	4	5	6	7	8	9
5	3	0	18	48	25	31	32	40	12

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

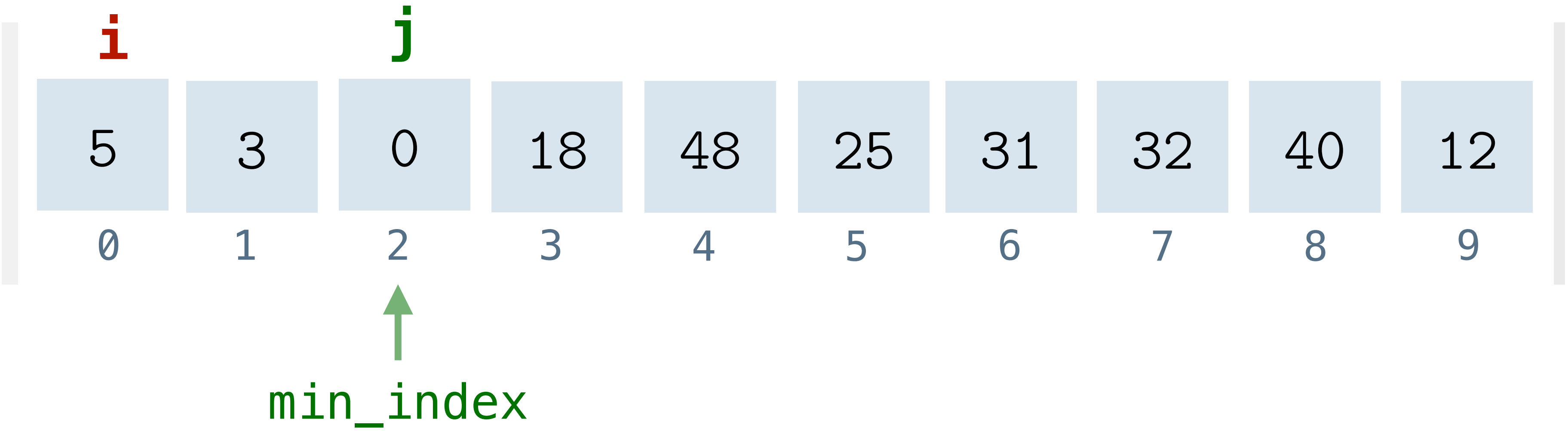
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing



i j

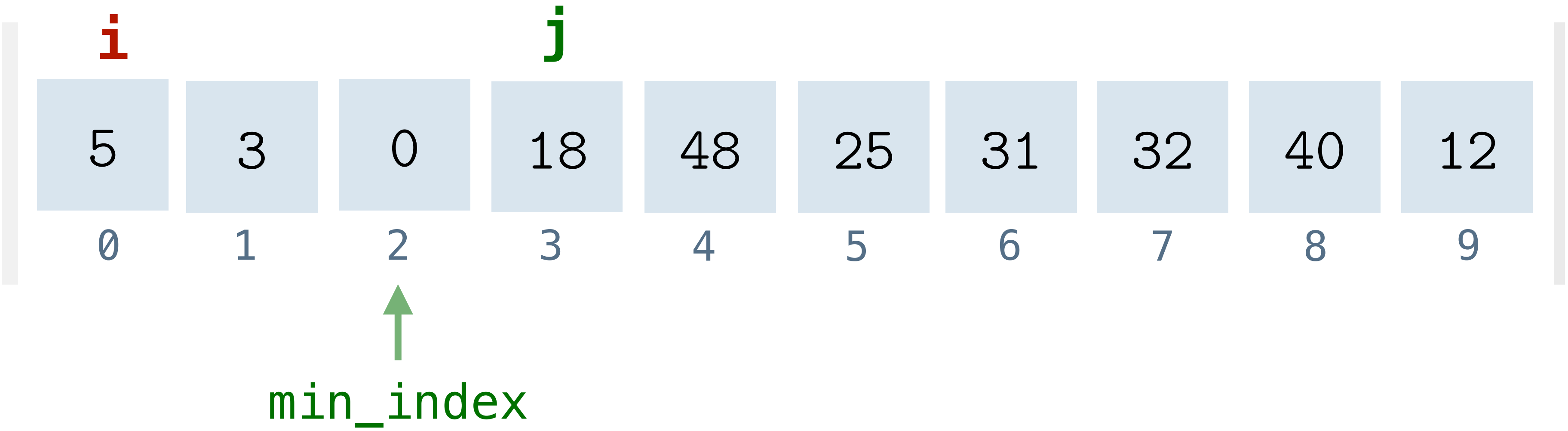
5	3	0	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

min_index

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

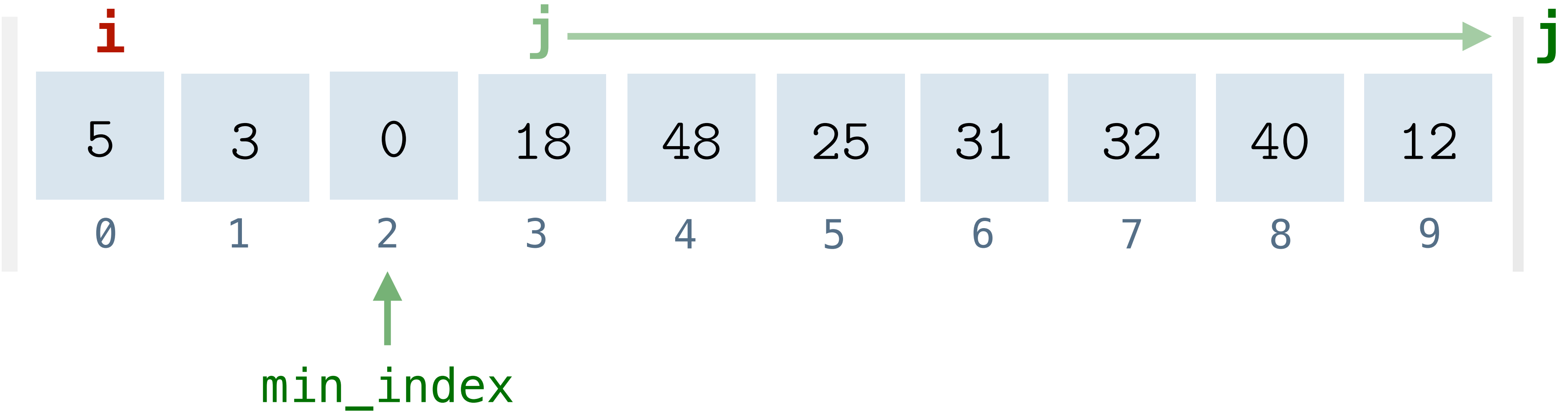


5	3	0	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

Selection Sort: Tracing

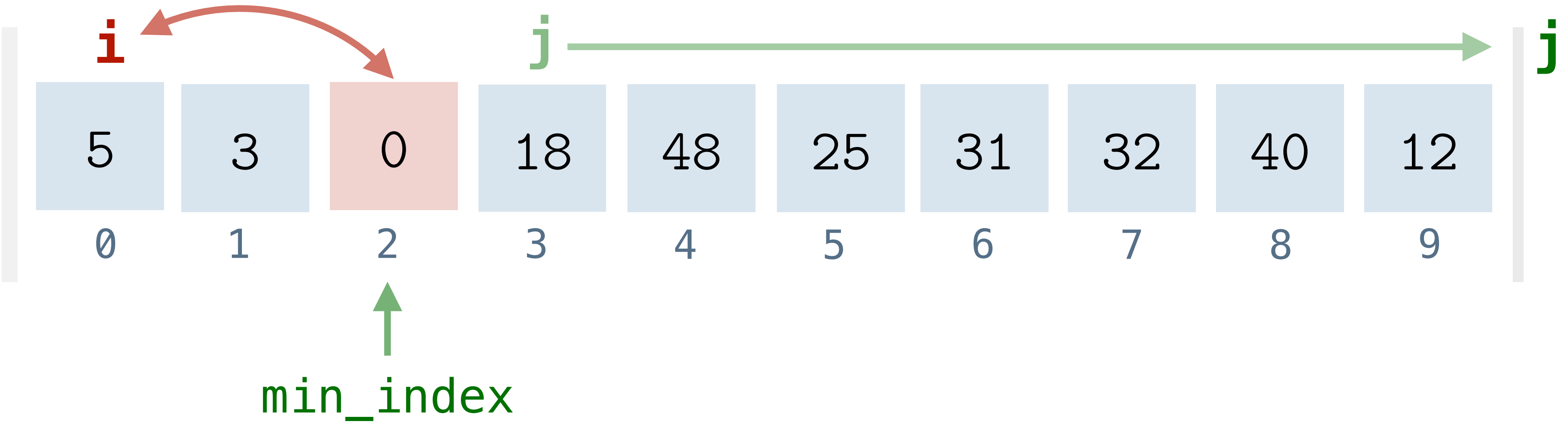


5	3	0	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

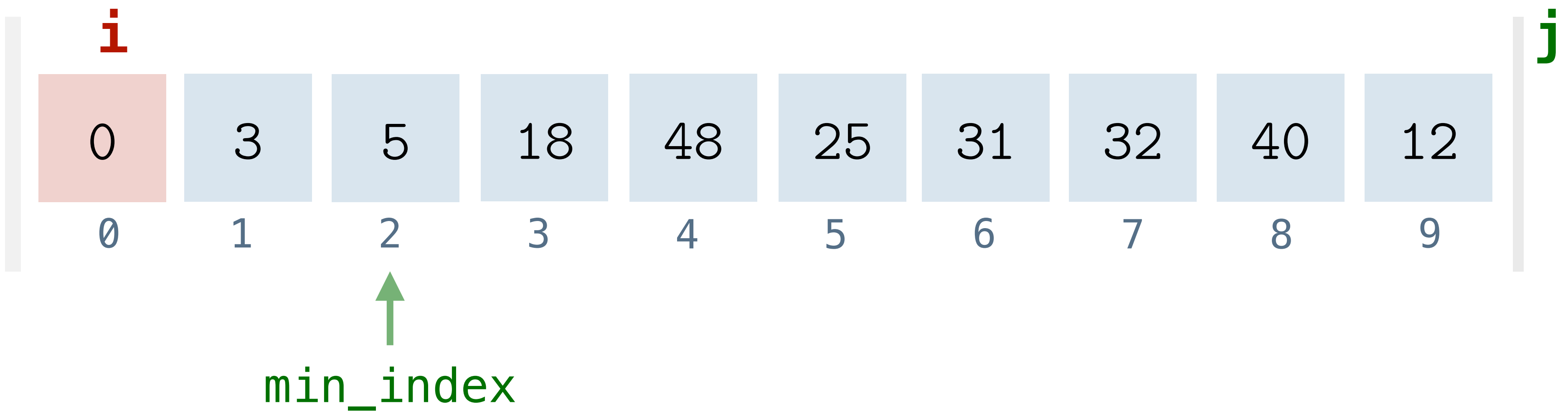
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

place the minimum
in its right position

Selection Sort: Tracing

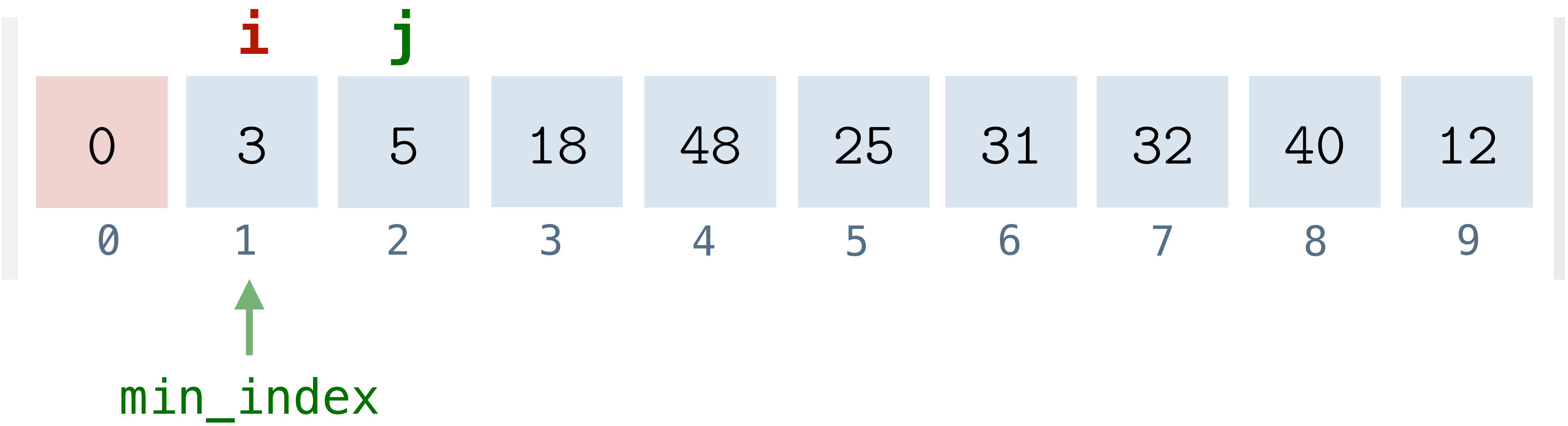


0	1	2	3	4	5	6	7	8	9
0	3	5	18	48	25	31	32	40	12

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

place the minimum
in its right position

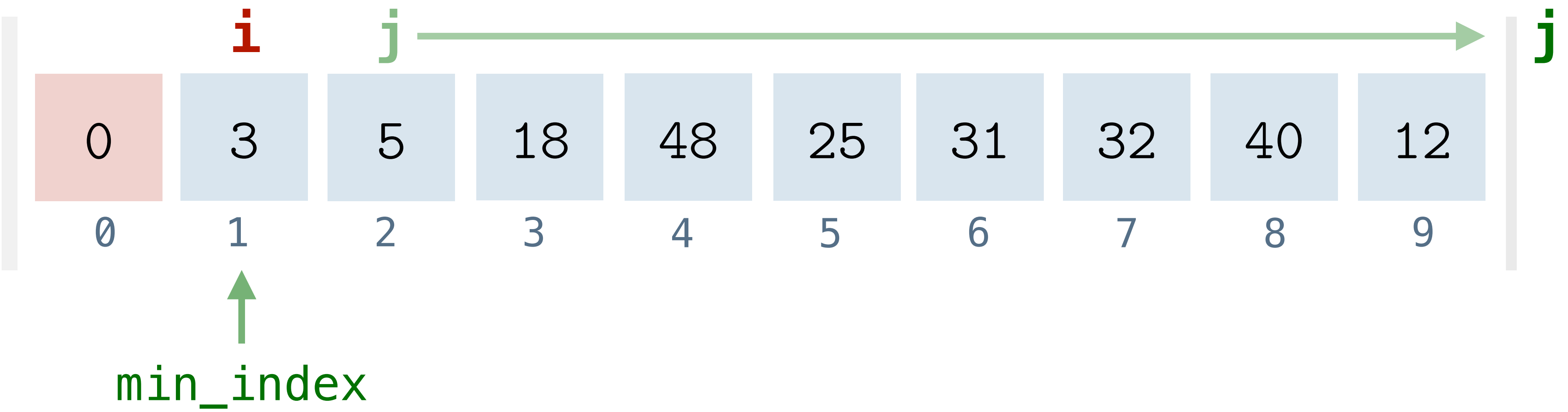
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

Selection Sort: Tracing

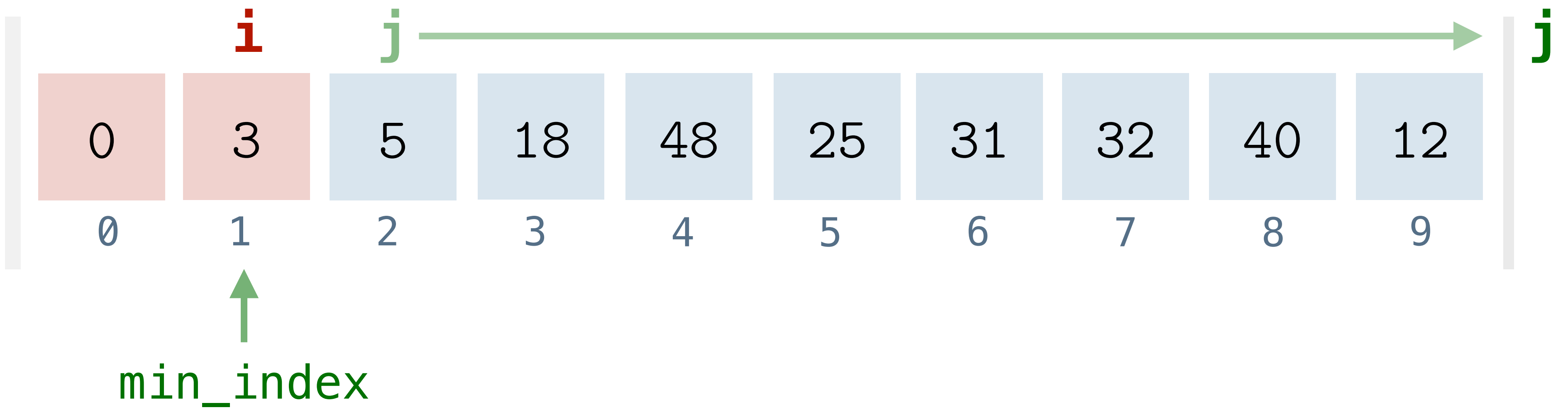


0	3	5	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing



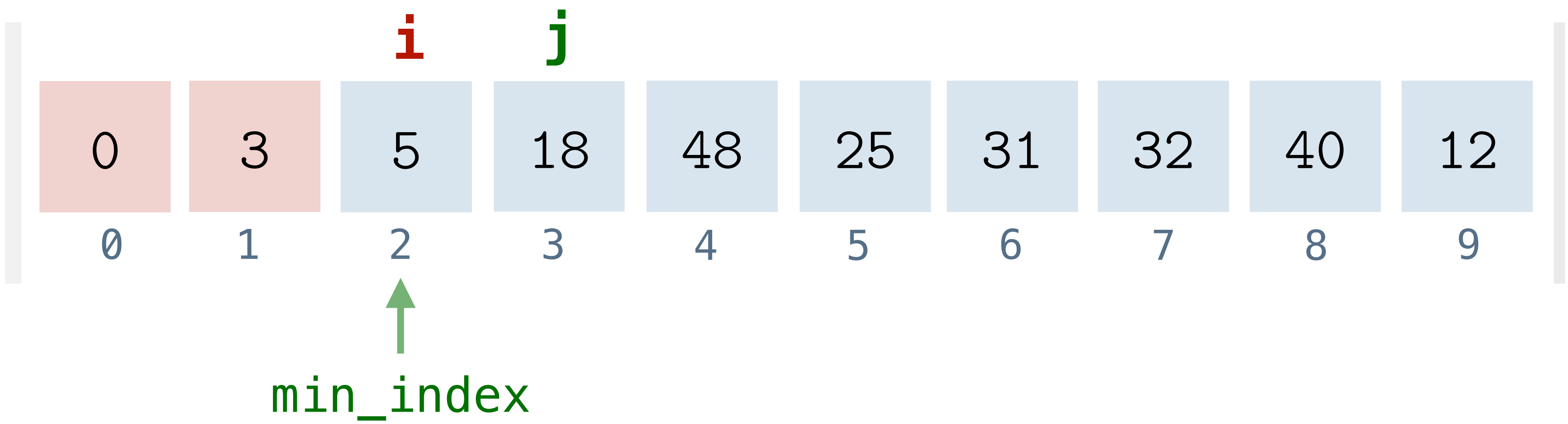
0	3	5	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

Selection Sort: Tracing

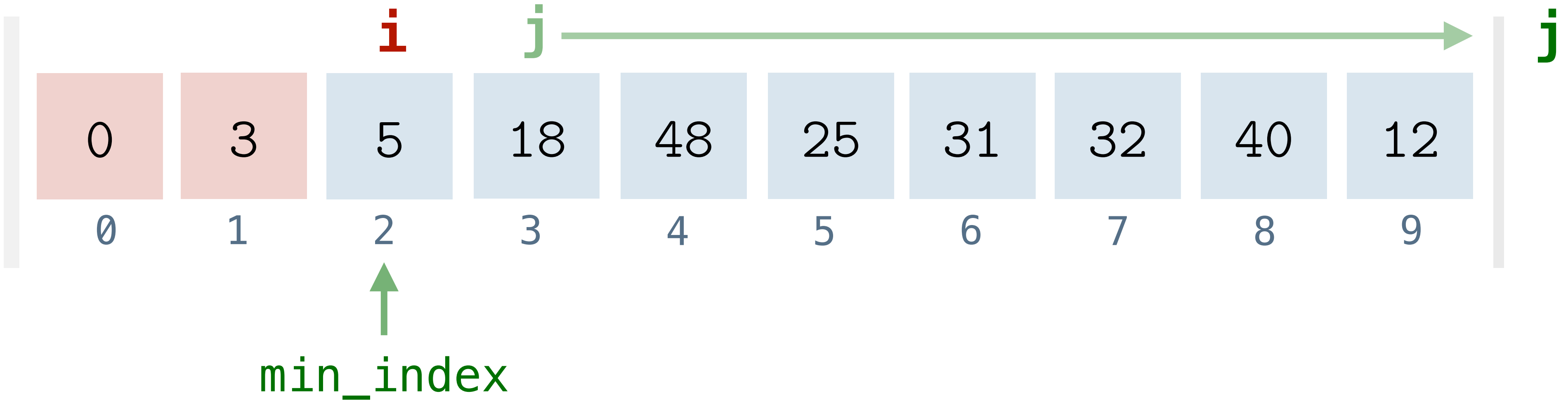


0	3	5	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

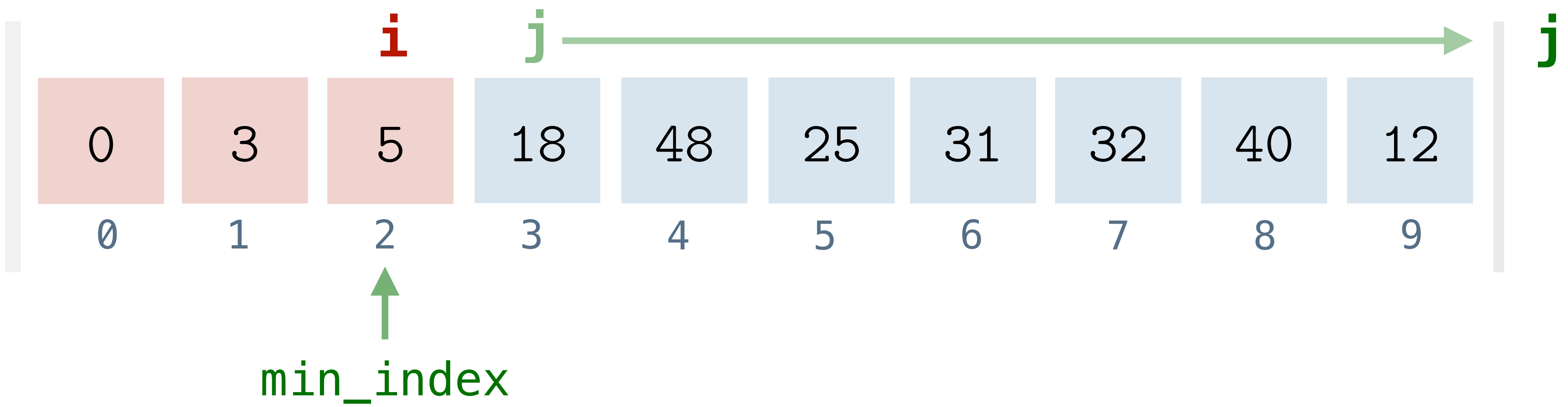
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

Selection Sort: Tracing

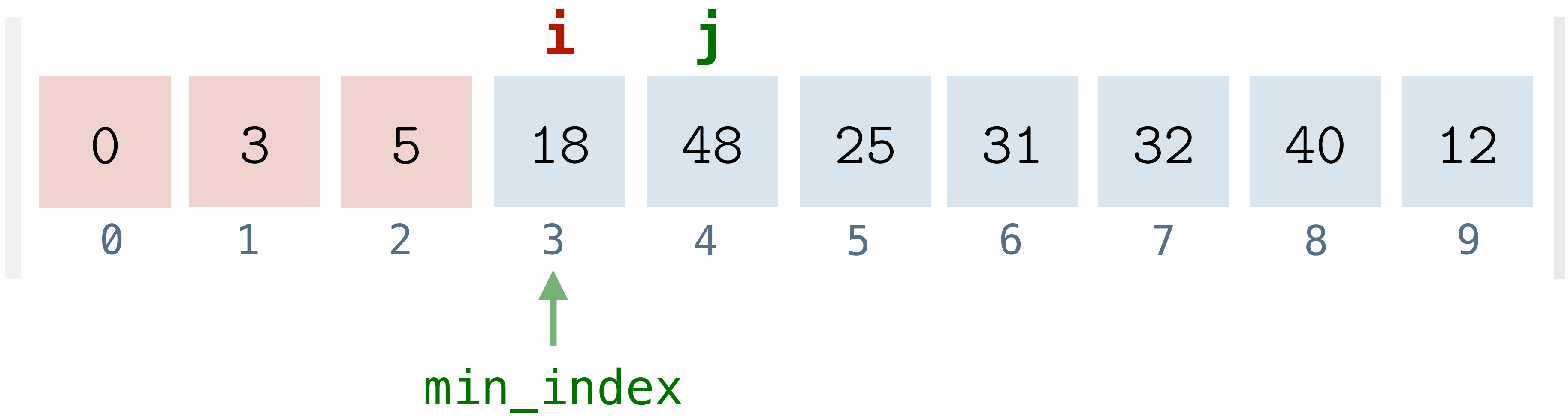


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

Selection Sort: Tracing

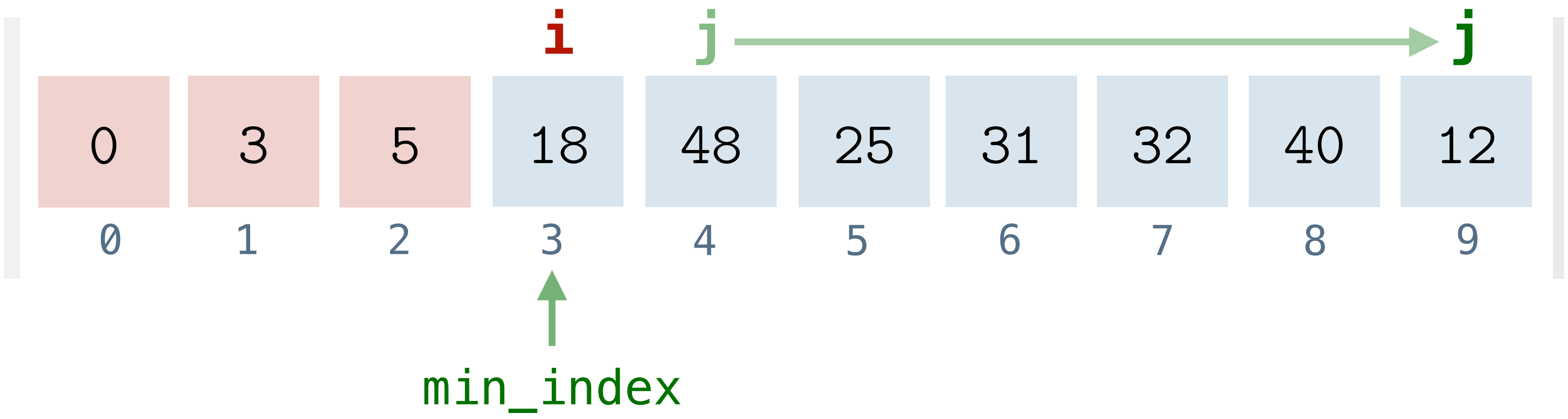


				i	j				
0	3	5	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

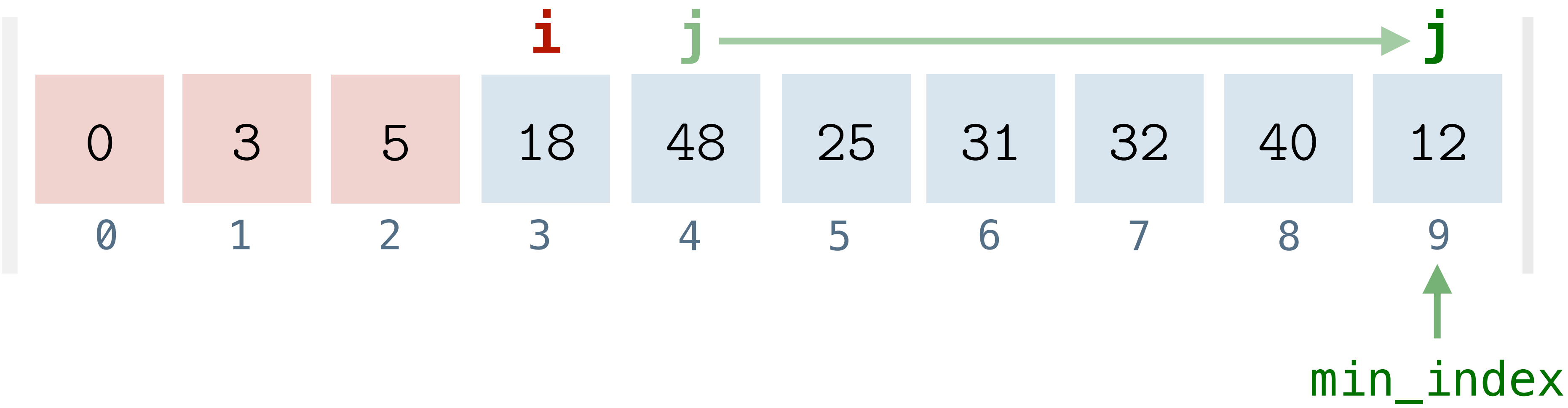
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

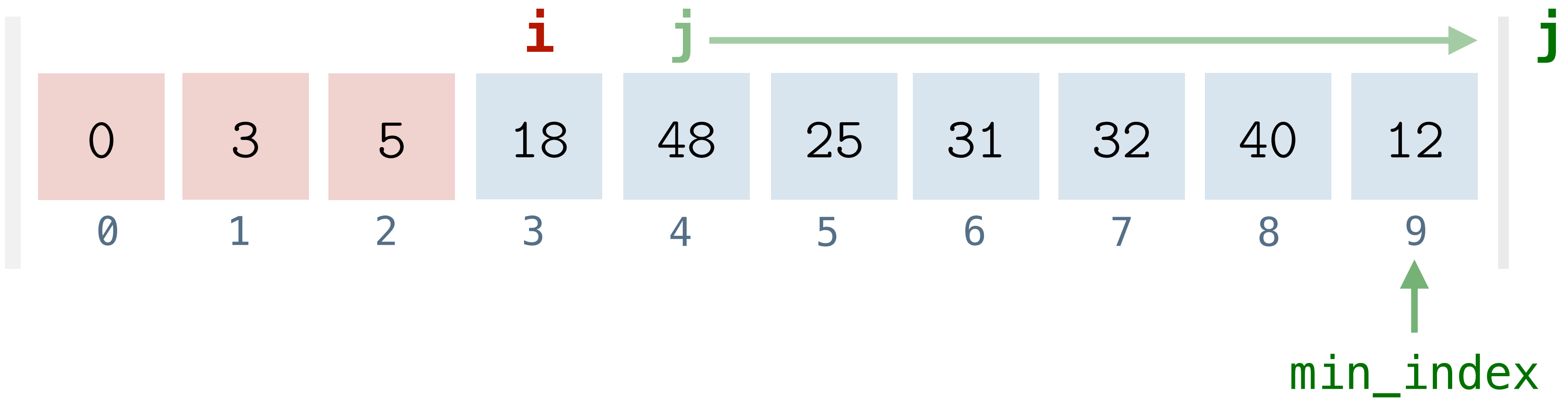
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

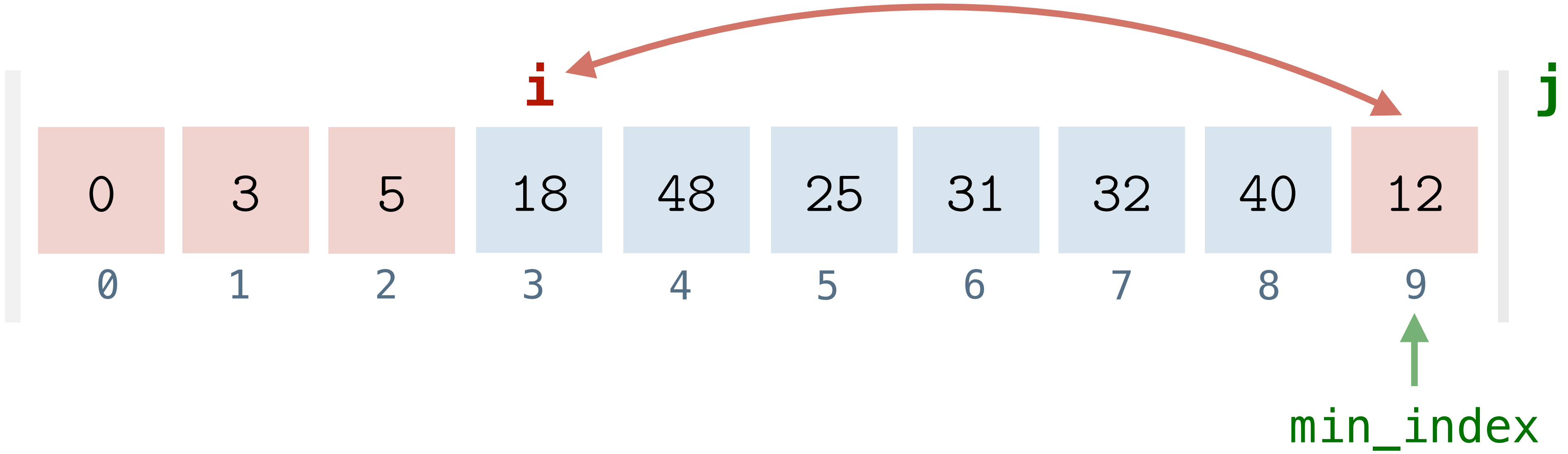
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

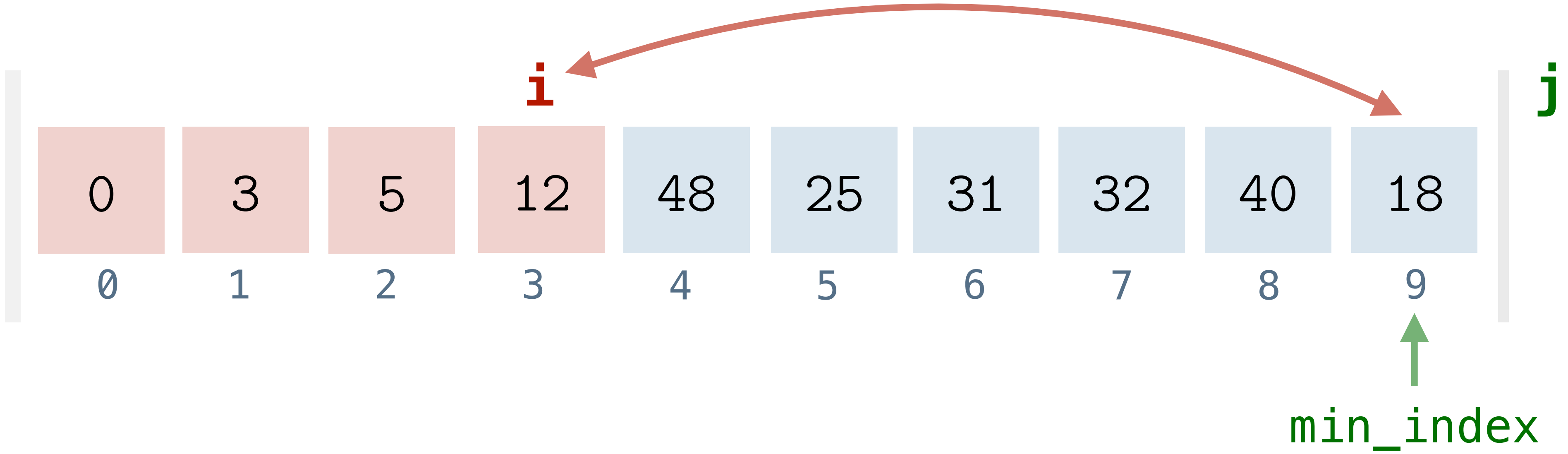


0	3	5	18	48	25	31	32	40	12
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

place the minimum
in its right position

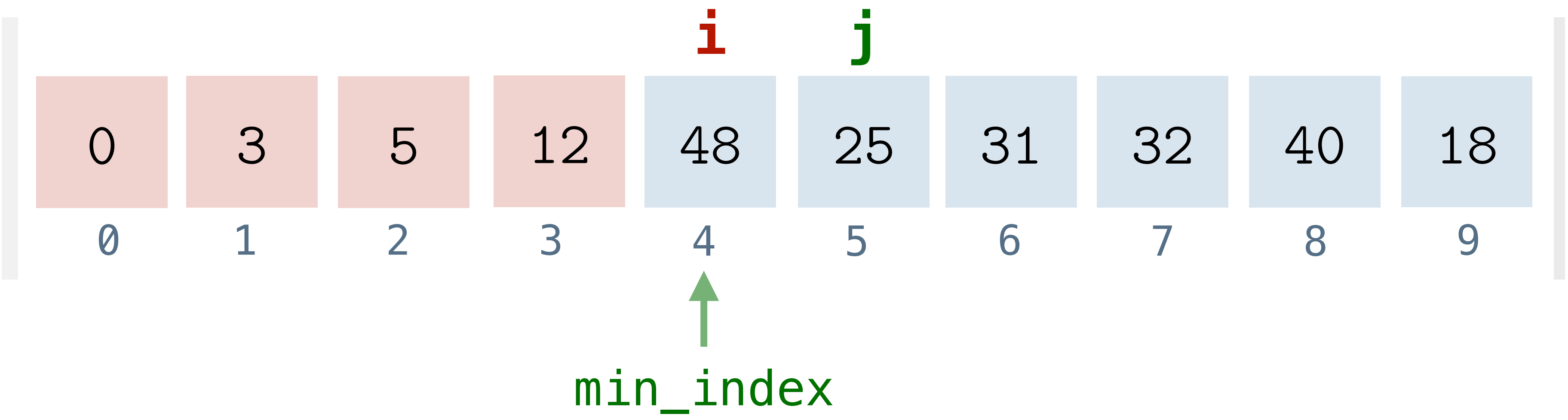
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

place the minimum
in its right position

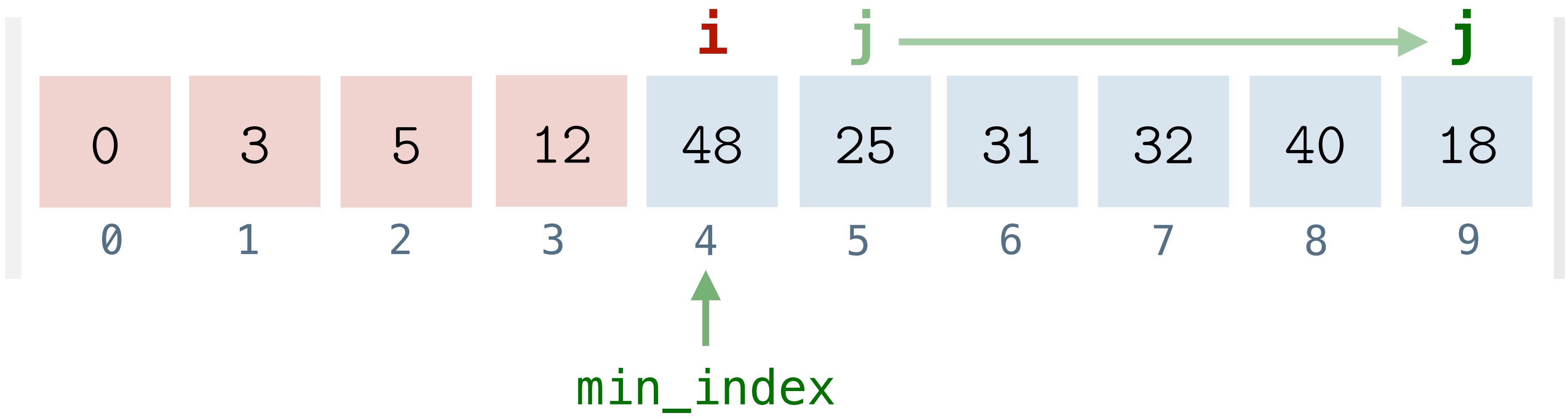
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

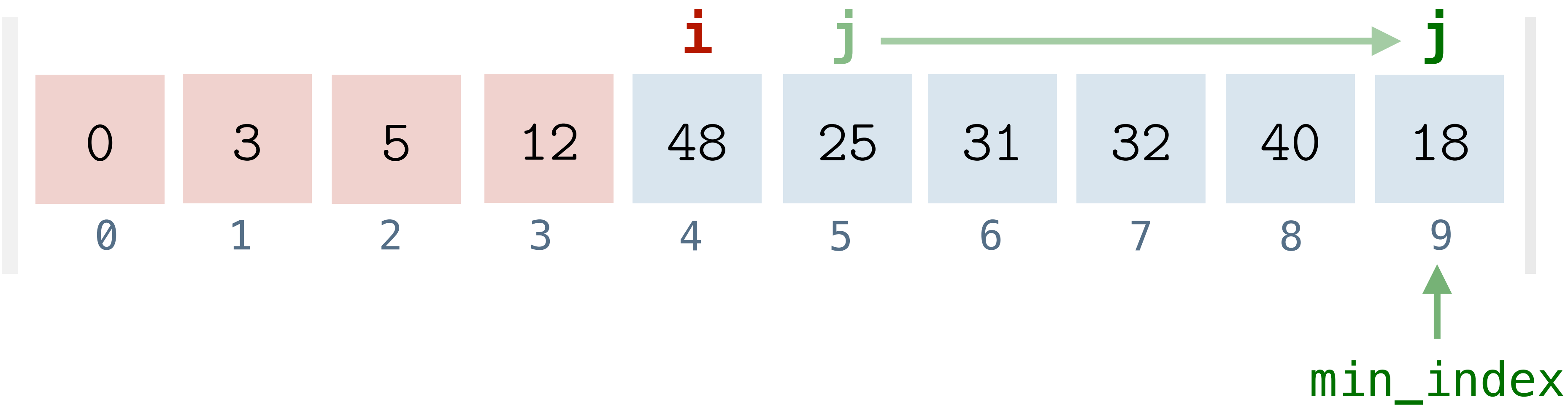


0	1	2	3	4	5	6	7	8	9
0	3	5	12	48	25	31	32	40	18

```
void selection(int a[], int n) {

    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

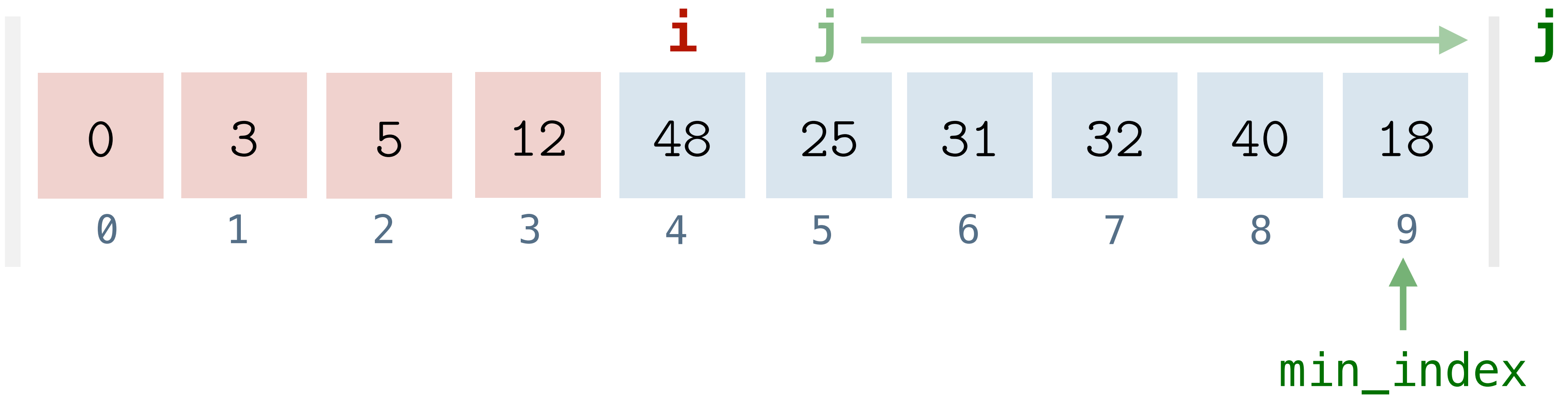


0	1	2	3	4	5	6	7	8	9
0	3	5	12	48	25	31	32	40	18

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

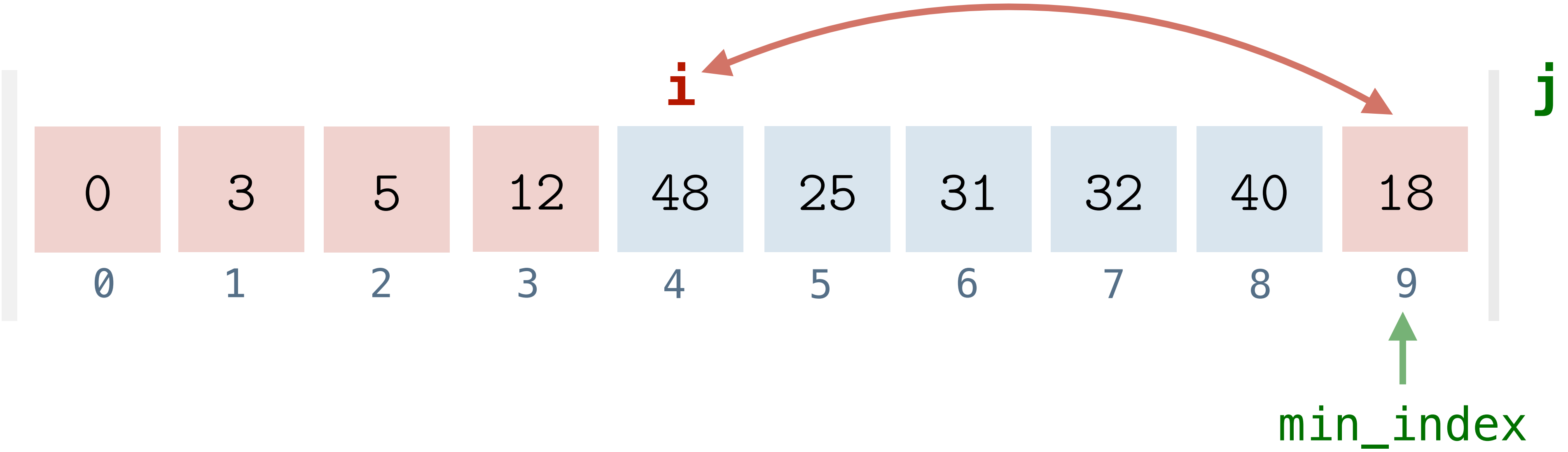


0	1	2	3	4	5	6	7	8	9
0	3	5	12	48	25	31	32	40	18

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

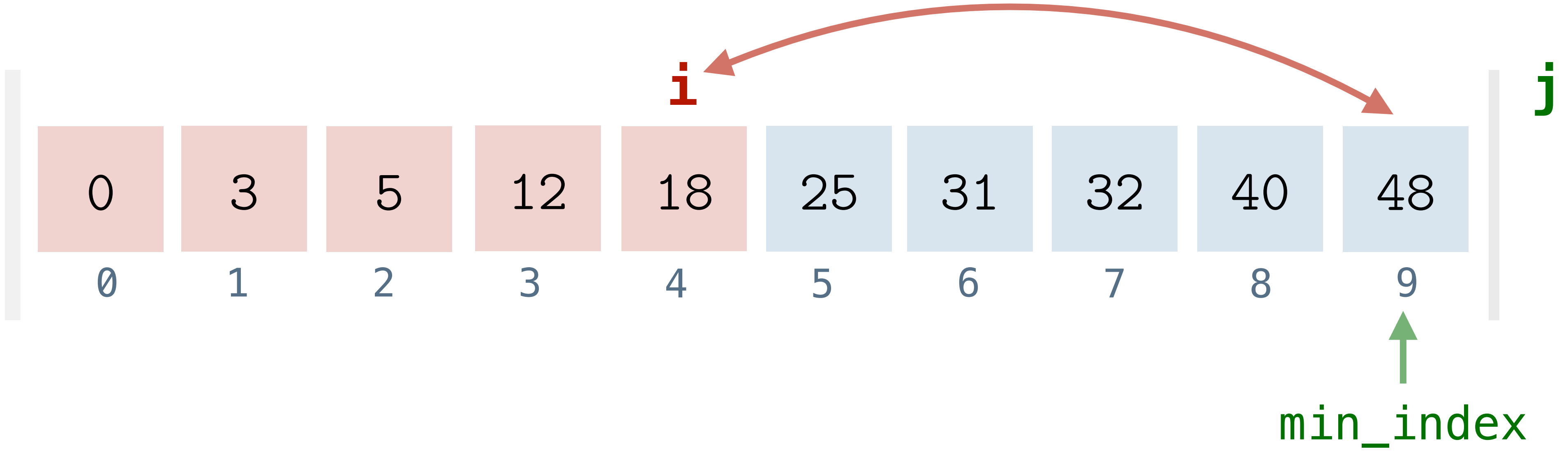
Selection Sort: Tracing



```
void selection(int a[], int n) {

    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

place the minimum
in its right position

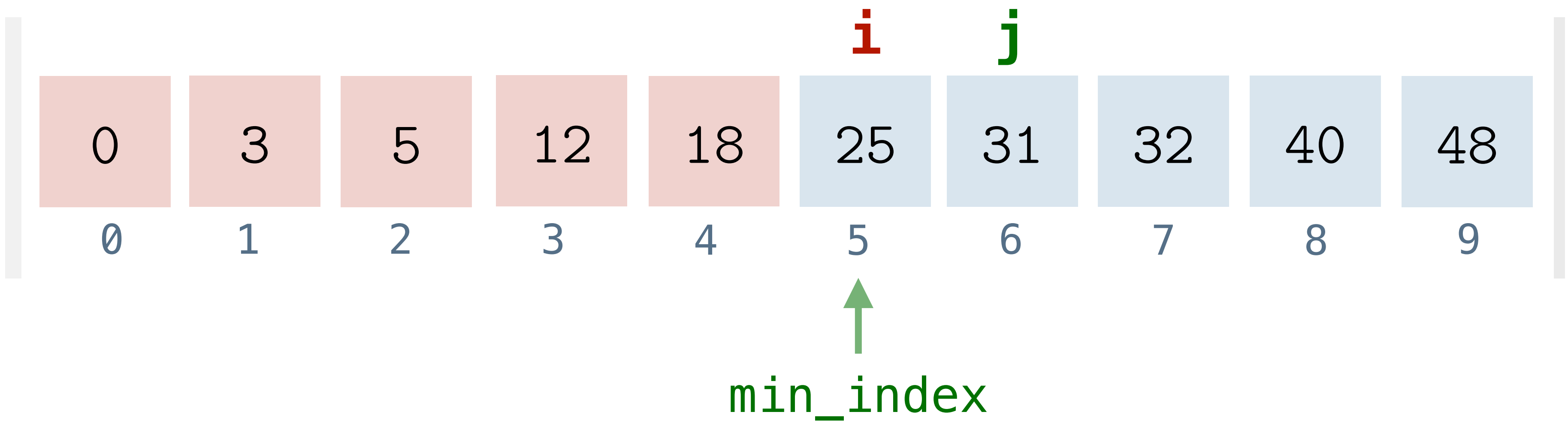
Selection Sort: Tracing



```
void selection(int a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min_index = i;
    for (int j = i+1; j < n; j++)
      if (a[j] < a[min_index])
        min_index = j;

    if (i != min_index)
      swap(a[i], a[min_index]);
  }
}
```

place the minimum
in its right position

Selection Sort: Tracing

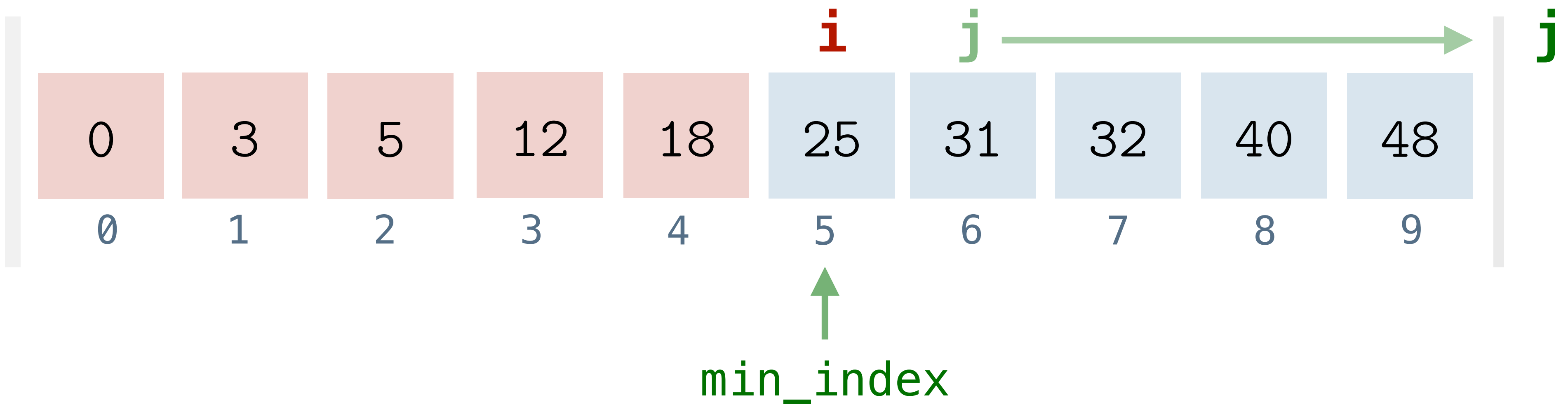


0	3	5	12	18	25	31	32	40	48
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

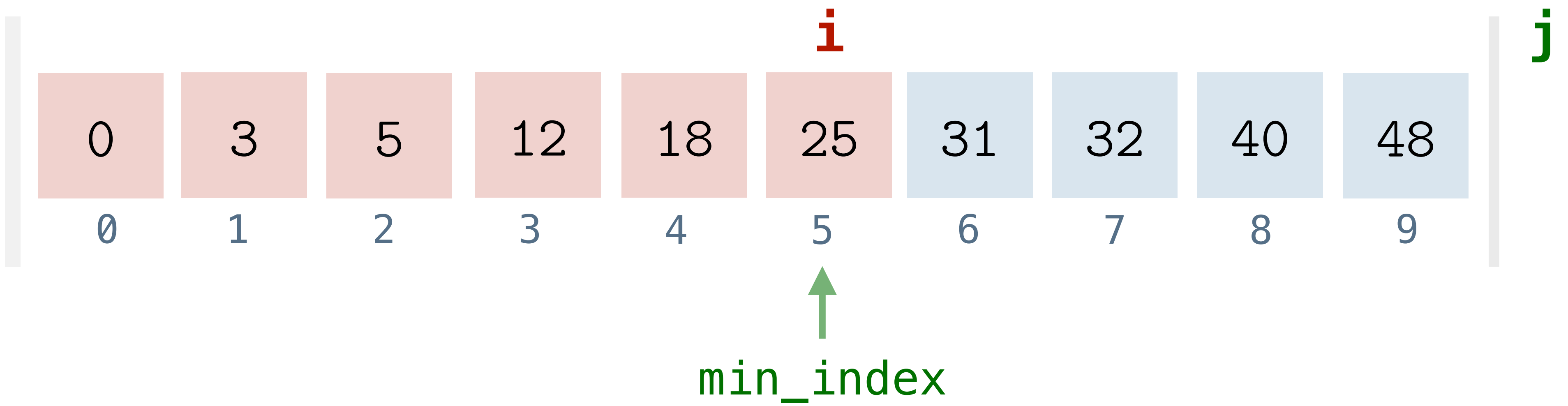
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
      int min_index = i;
      for (int j = i+1; j < n; j++)
          if (a[j] < a[min_index])
              min_index = j;

      if (i != min_index)
          swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

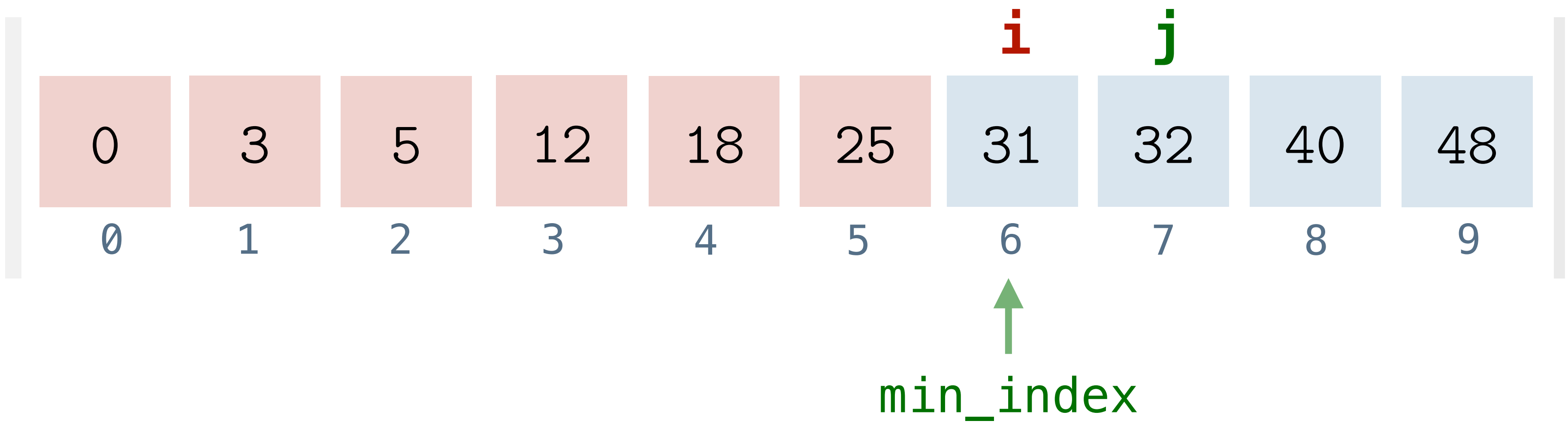


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

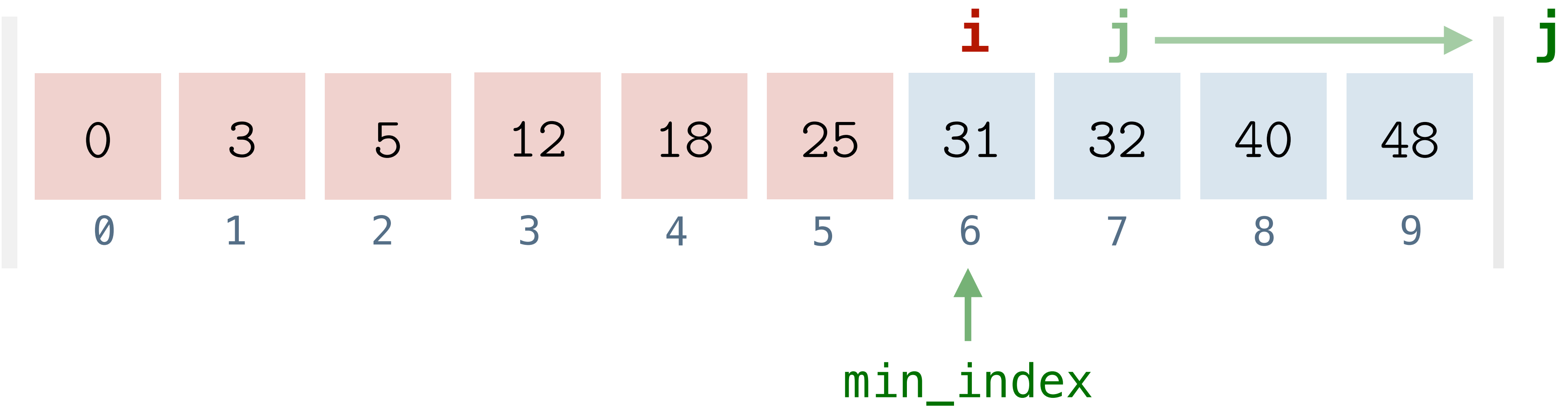
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

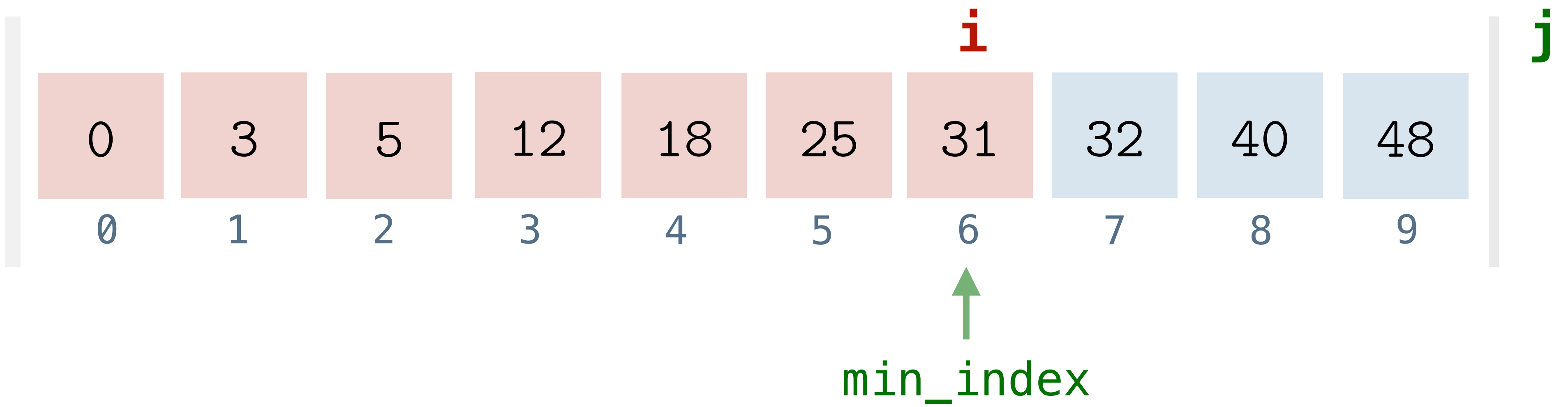
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

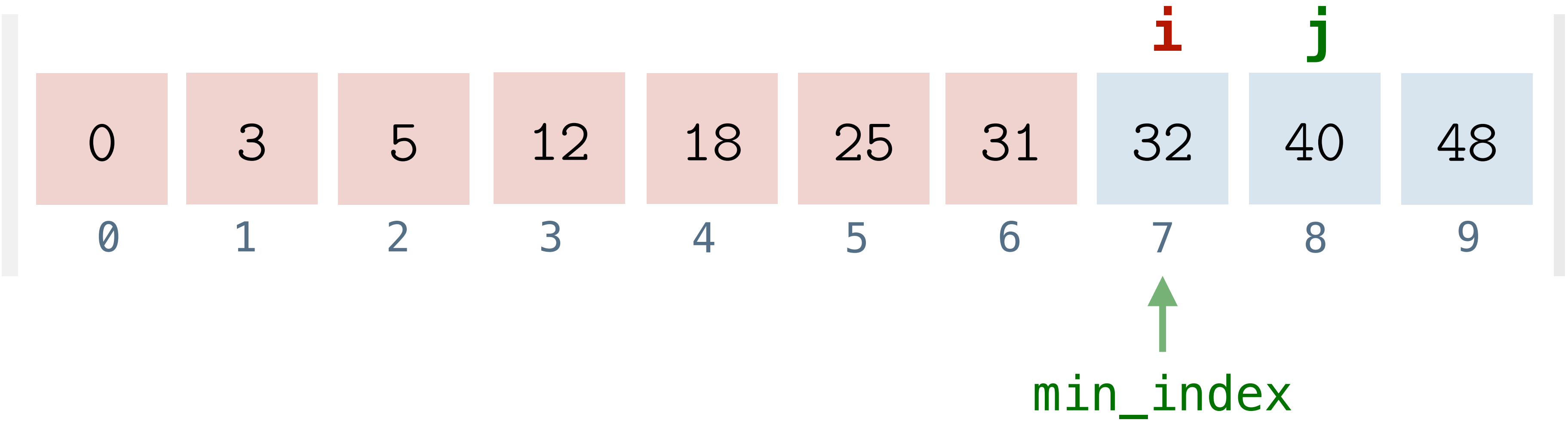


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

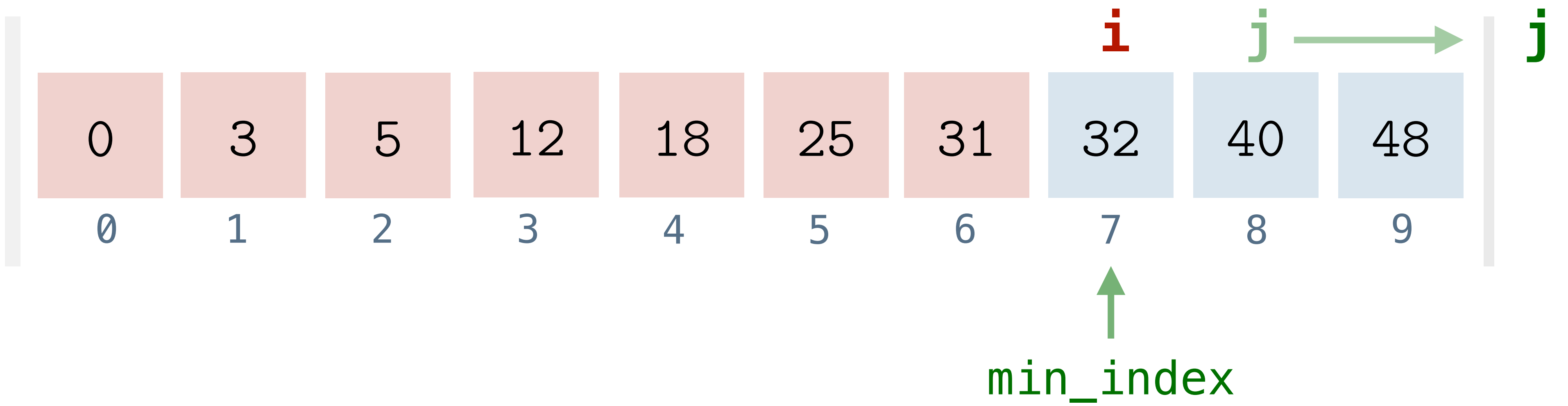
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

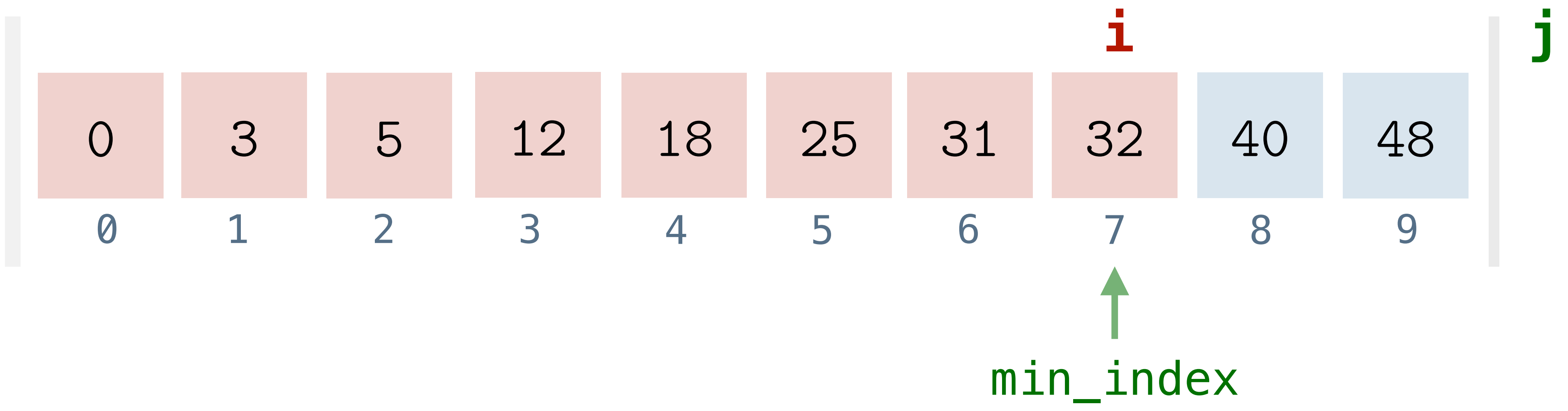
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing

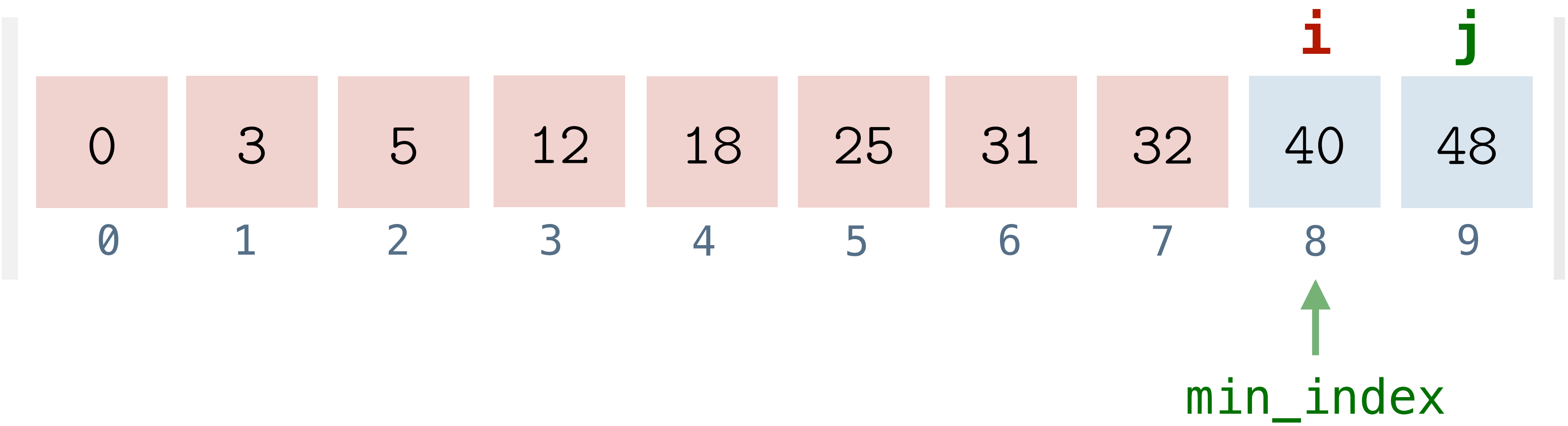


```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

Selection Sort: Tracing

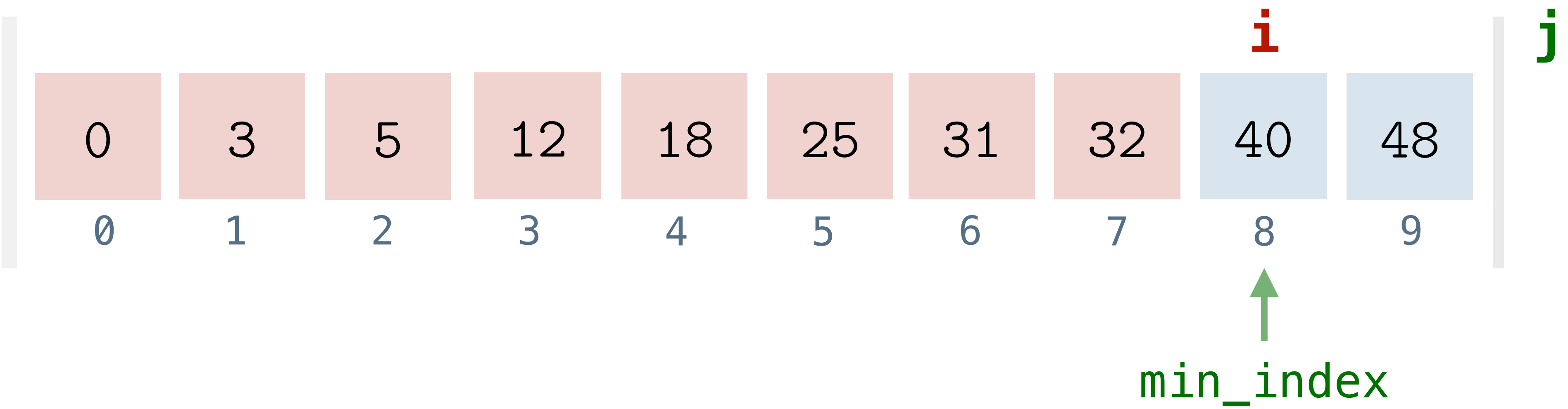


								i	j
0	3	5	12	18	25	31	32	40	48
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in `a[i, n-1]`

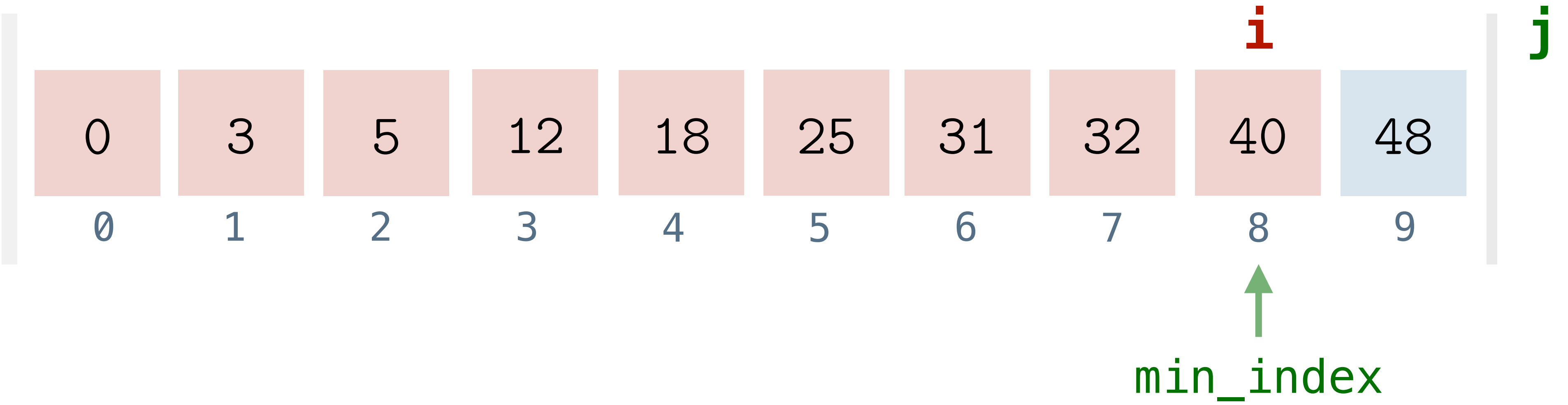
Selection Sort: Tracing



```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

Selection Sort: Tracing



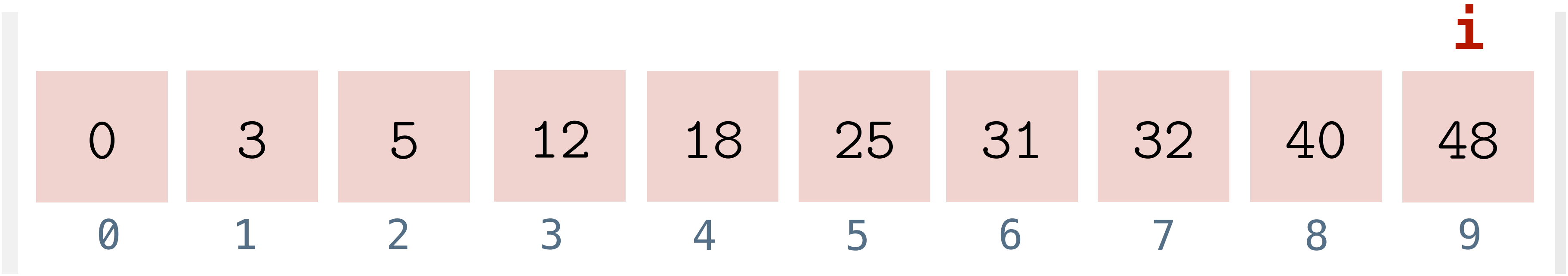
0	3	5	12	18	25	31	32	40	48
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

No swap!

place the minimum
in its right position

Selection Sort: Tracing



0	3	5	12	18	25	31	32	40	48
0	1	2	3	4	5	6	7	8	9

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares.

Counting only **comparisons** between **array elements**

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares. The algorithm is insensitive to the arrangement of the elements in the array.

$$= 1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \text{ data compares}$$

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares. The algorithm is insensitive to the arrangement of the elements in the array.

$$= 1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \text{ data compares}$$

Data Moves.

Worst case.

Best case.

Counting only **movements** of **array elements**

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares. The algorithm is insensitive to the arrangement of the elements in the array.

$$= 1 + 2 + 3 + \dots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \text{ data compares}$$

Data Moves.

Worst case. One swap per iteration, a total of $n-1$ swaps (= $3(n-1)$ data moves).

Best case. No swaps if the array is already sorted.

Counting only **movements** of **array elements**

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares. The algorithm is insensitive to the arrangement of the elements in the array.

$$= 1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \text{ data compares}$$

Data Moves.

Worst case. One swap per iteration, a total of $n - 1$ swaps (= $3(n-1)$ data moves).

Best case. No swaps if the array is already sorted.

Think!

Can you come up with an array of size 6 that leads to 5 swaps?

Analysis

```
void selection(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_index = i;
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min_index])
                min_index = j;

        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

find the index of the minimum in a[i, n-1]

place the minimum in its right position

Data compares. The algorithm is insensitive to the arrangement of the elements in the array.

$$= 1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \text{ data compares}$$

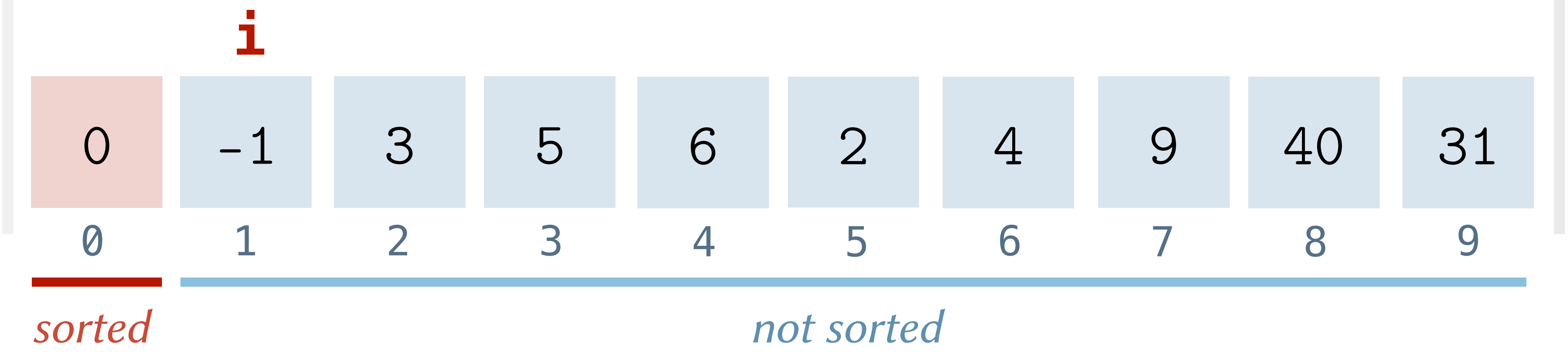
Data Moves.

Worst case. One swap per iteration, a total of $n-1$ swaps (= $3(n-1)$ data moves).

Best case. No swaps if the array is already sorted.

Total. $O(n^2)$ operations in the best case and the worst case.

Insertion Sort: Implementation

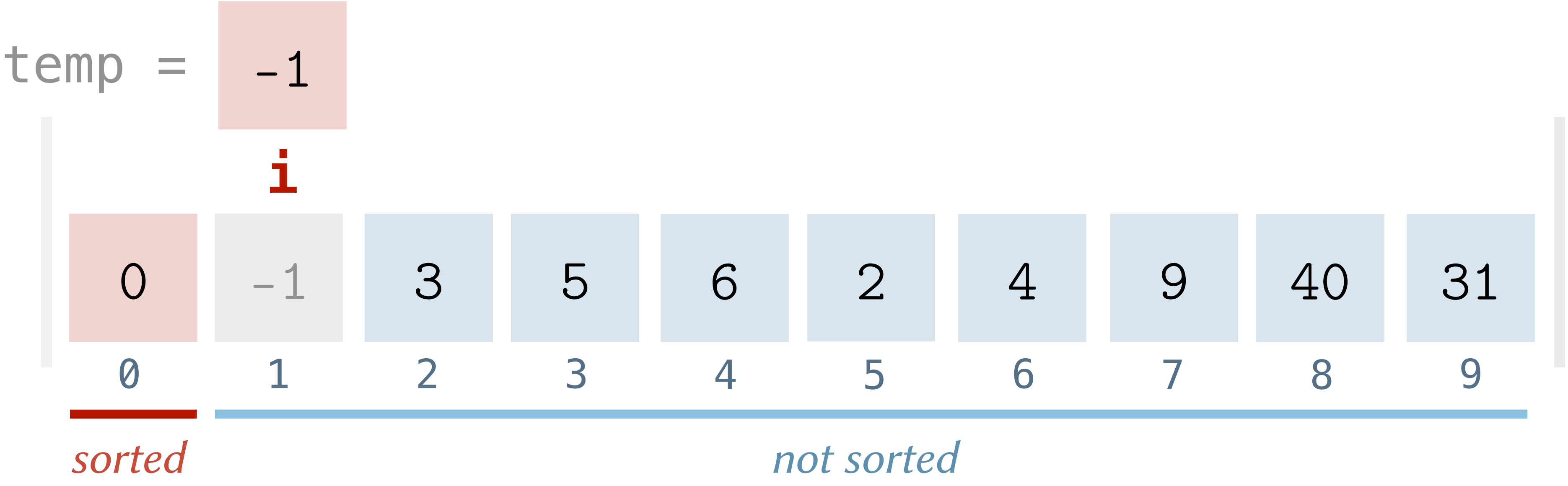


0	-1	3	5	6	2	4	9	40	31
0	1	2	3	4	5	6	7	8	9

```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {







    }
}
```

Insert every element from the ***unsorted*** part into its correct position in the ***sorted*** part

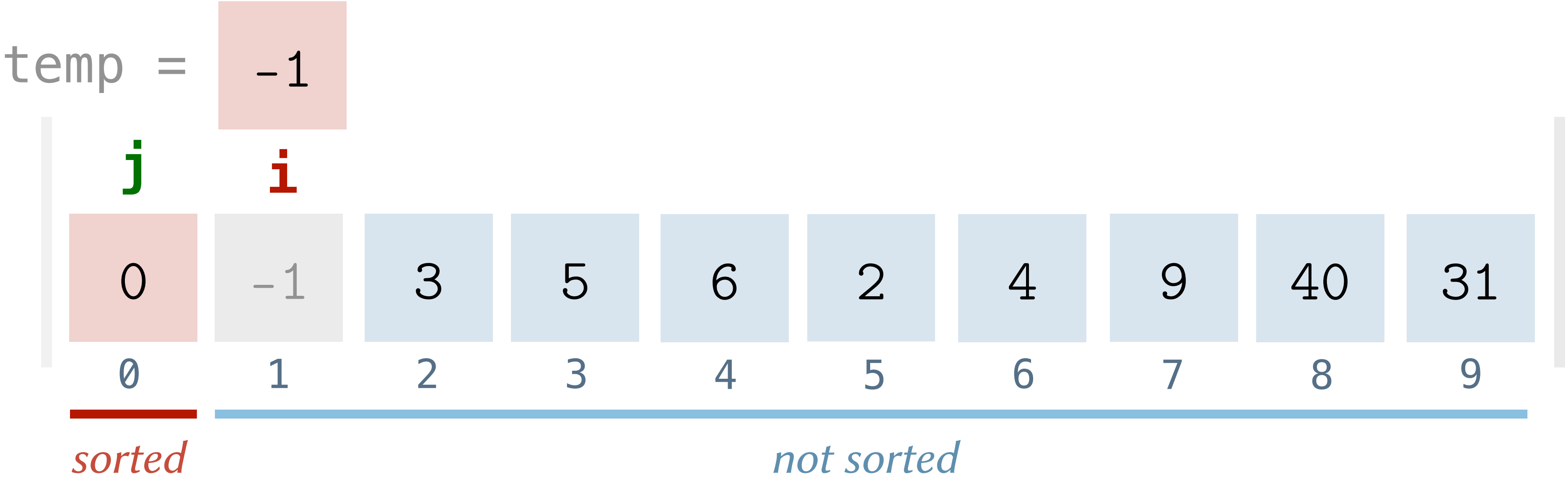
Insertion Sort: Implementation



```
void insertion(int a[], int n) {

    for (int i = 1; i < n; i++) {
        int temp = a[i];










    }
}
```

store element *i*

Insertion Sort: Implementation

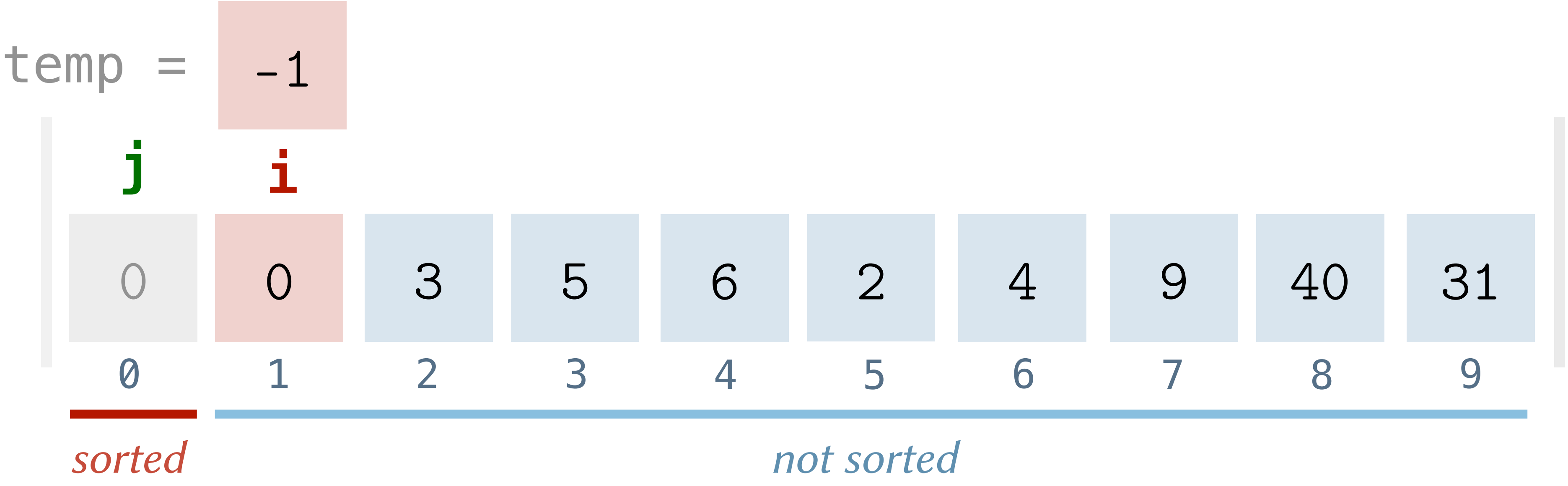


```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }


    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

Insertion Sort: Implementation

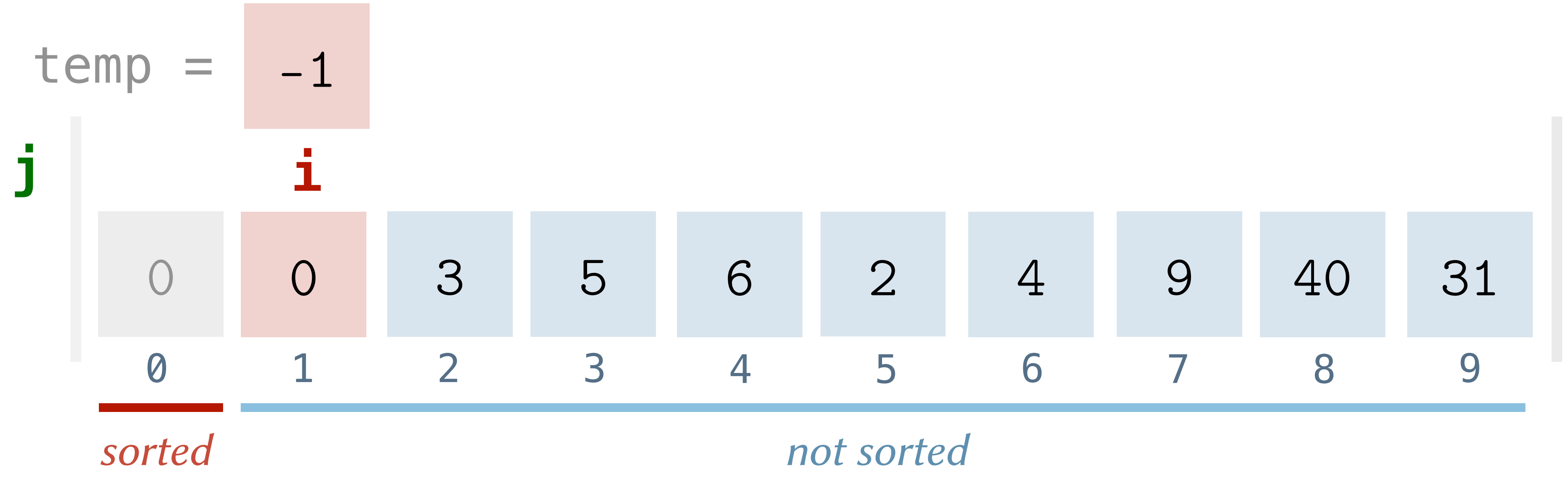


```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

Insertion Sort: Implementation

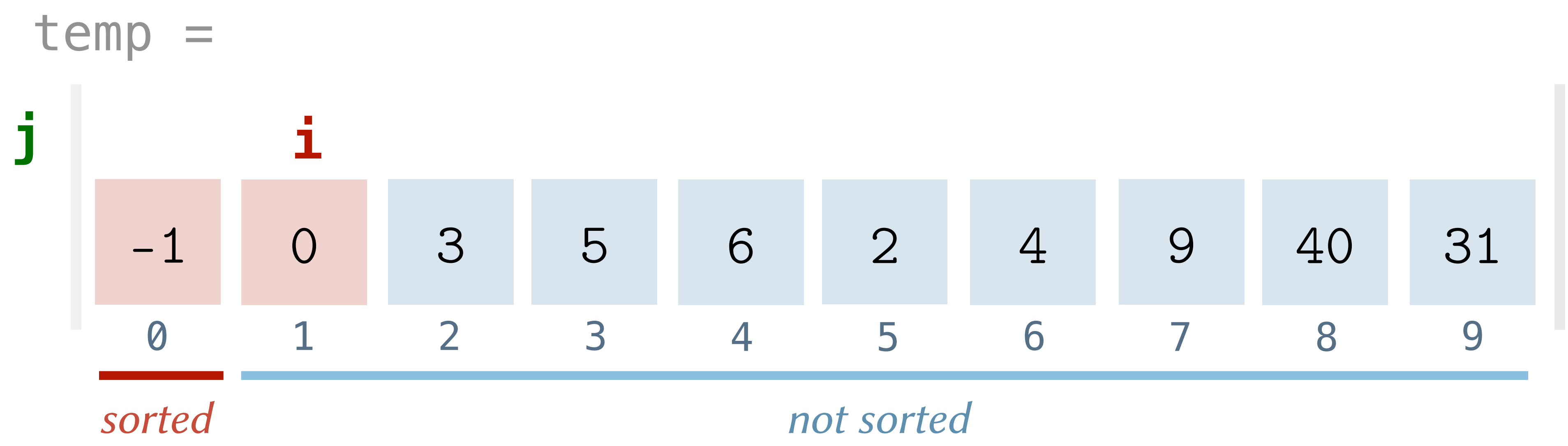


```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

    }
}
```

store element *i*

shift the elements until
the right position of
element *i* is found

Insertion Sort: Implementation



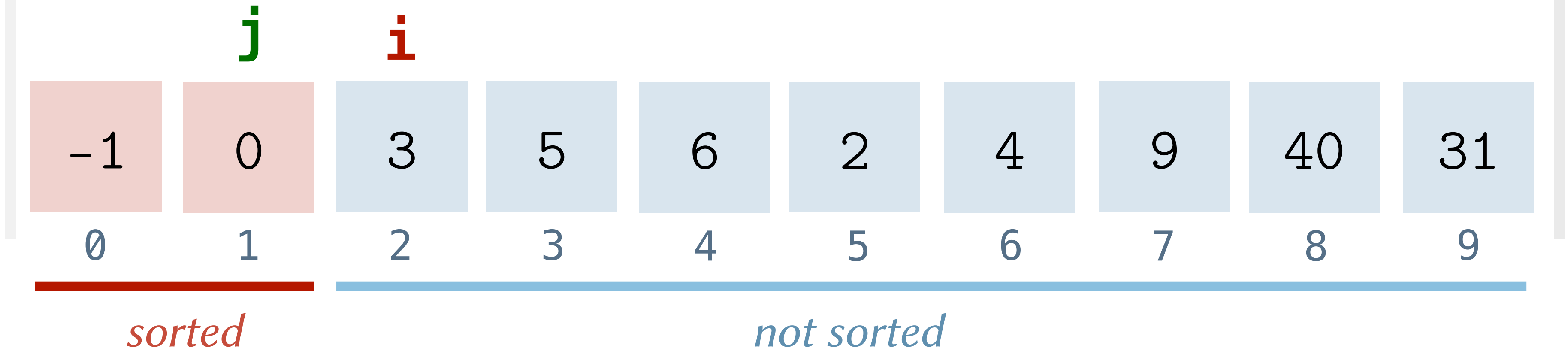
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



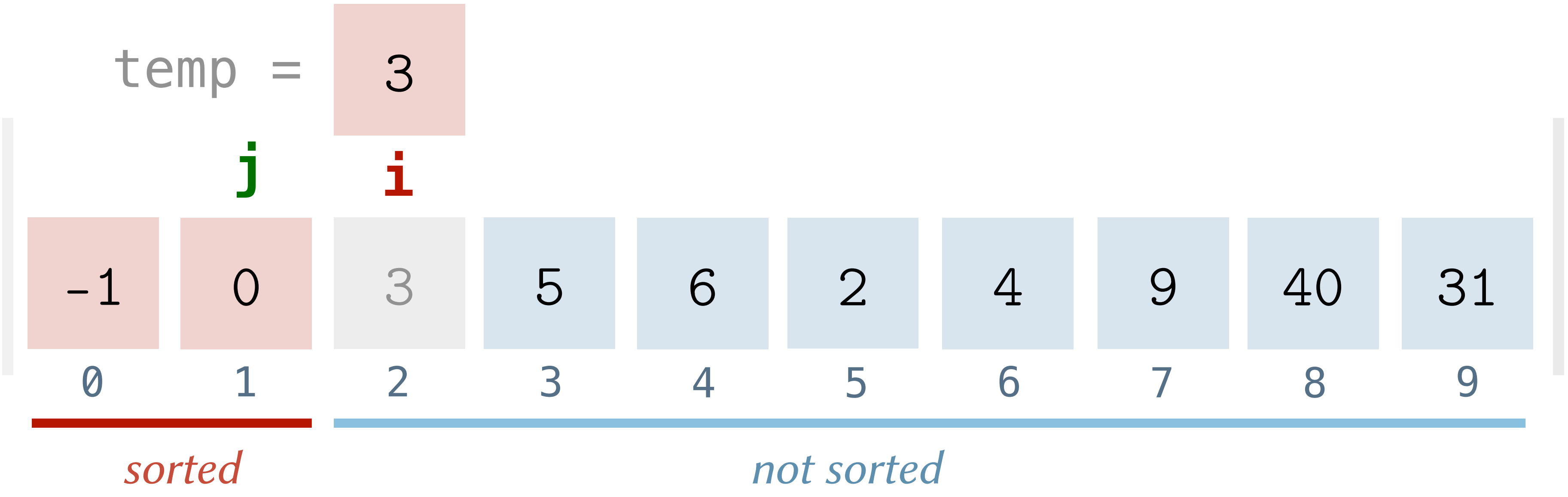
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



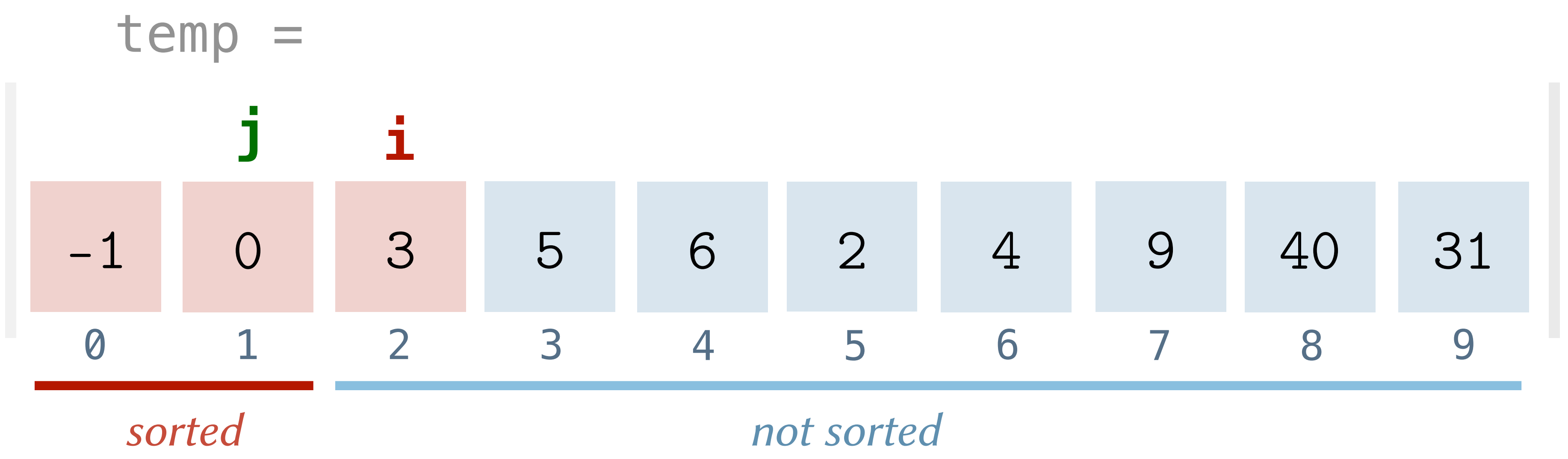
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



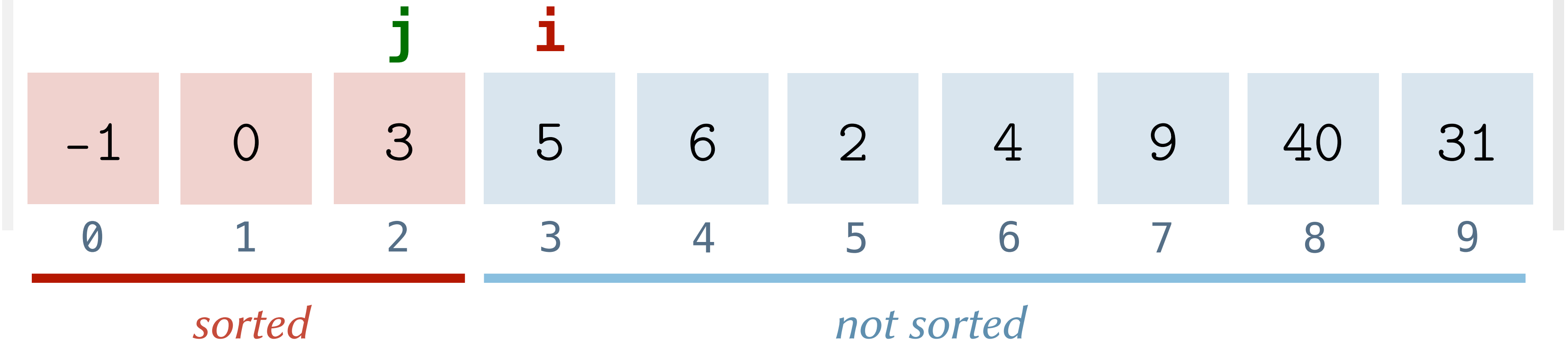
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



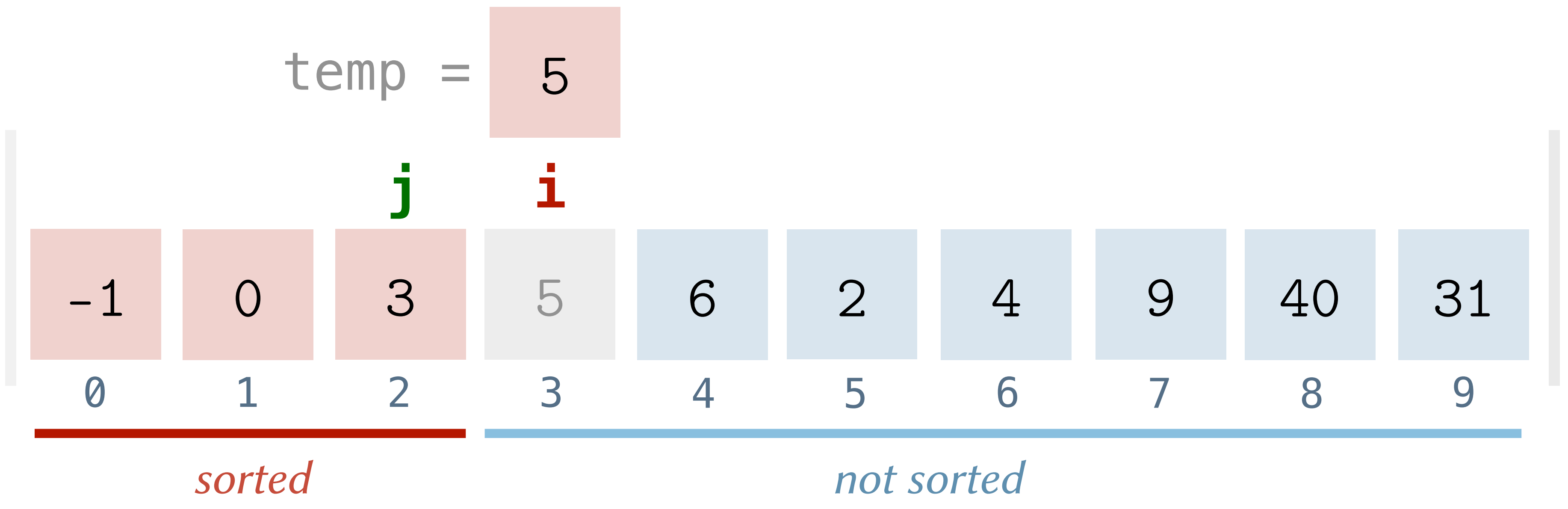
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



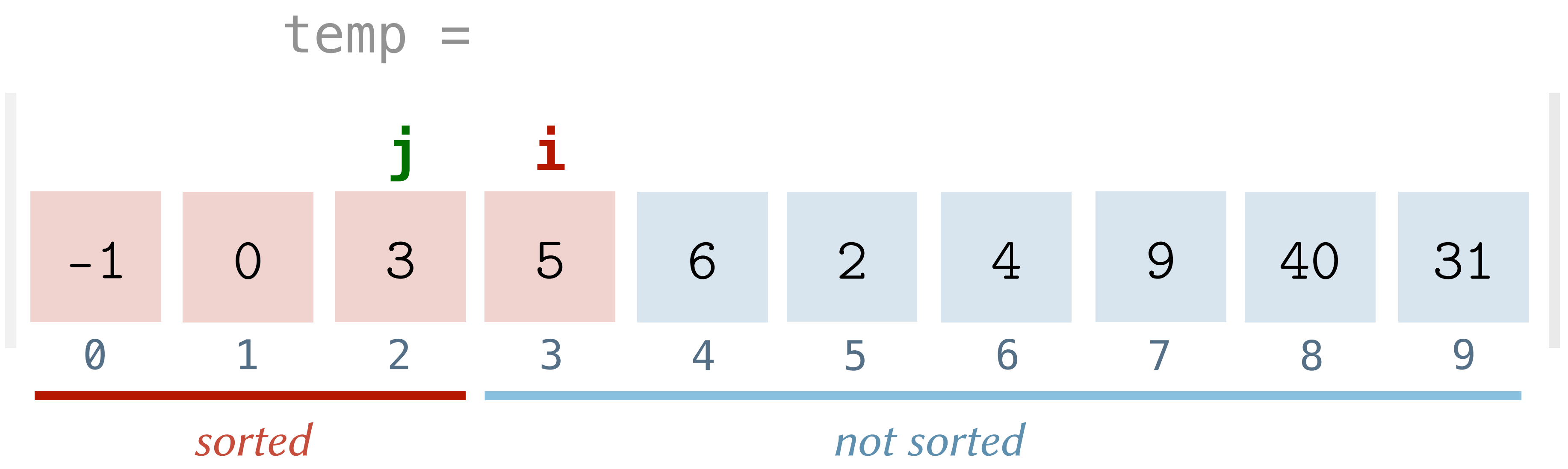
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



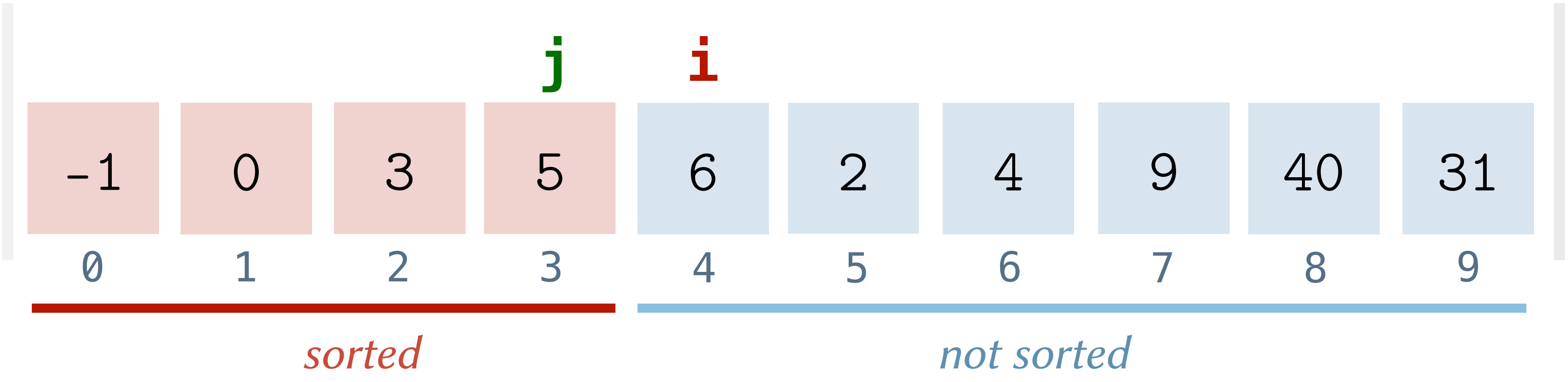
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



j i

-1	0	3	5	6	2	4	9	40	31
0	1	2	3	4	5	6	7	8	9

sorted *not sorted*

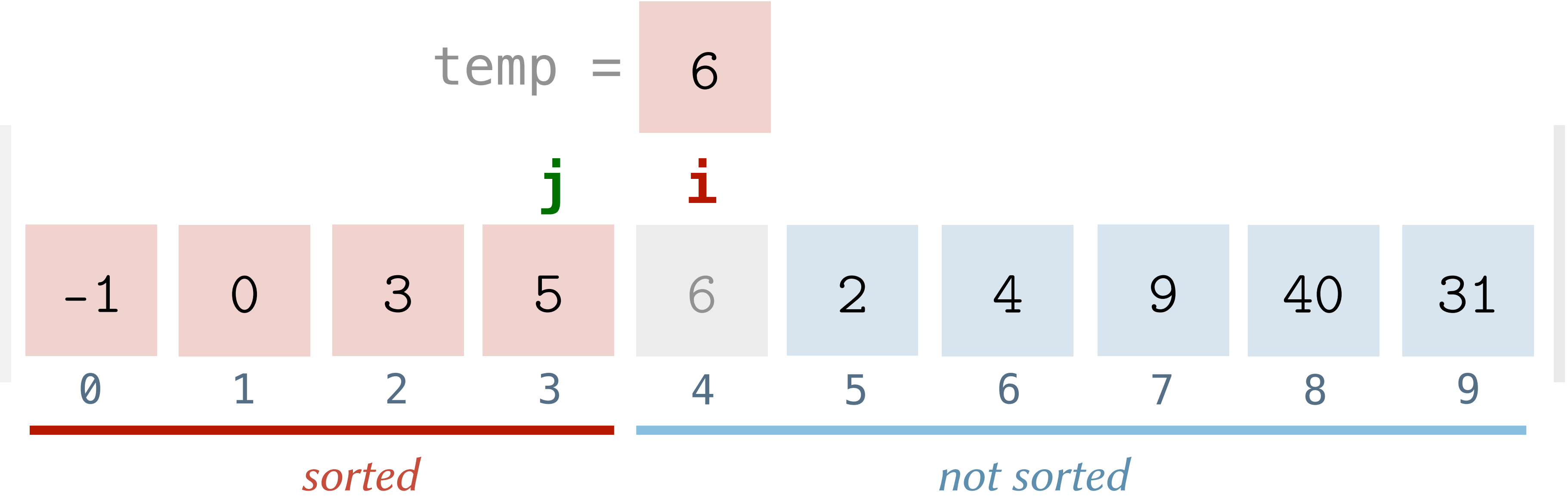
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



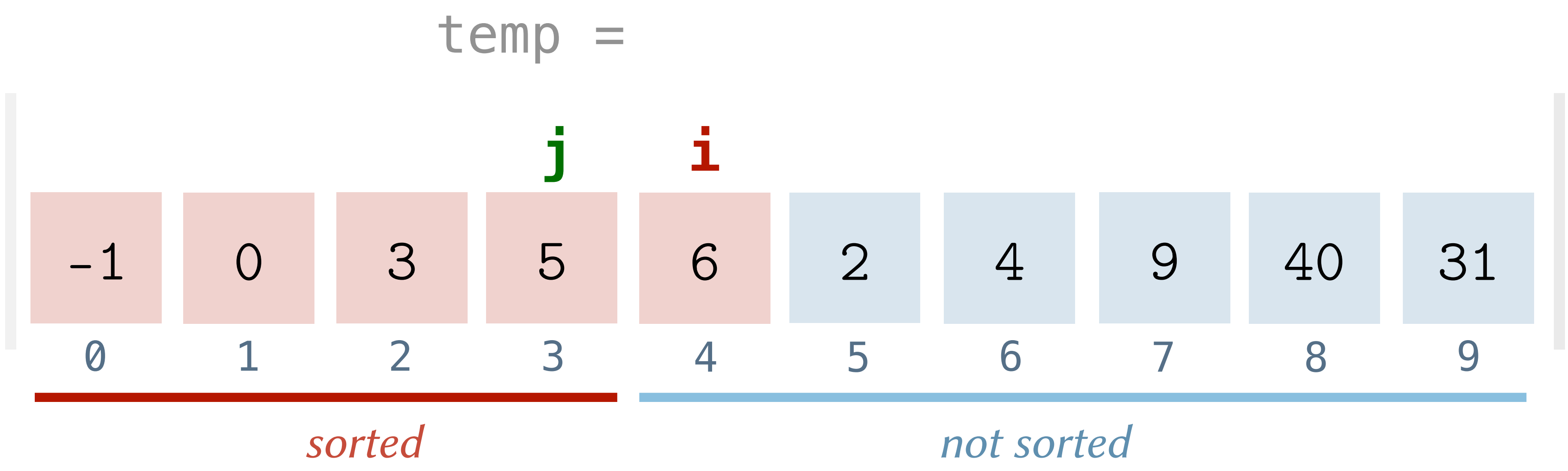
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

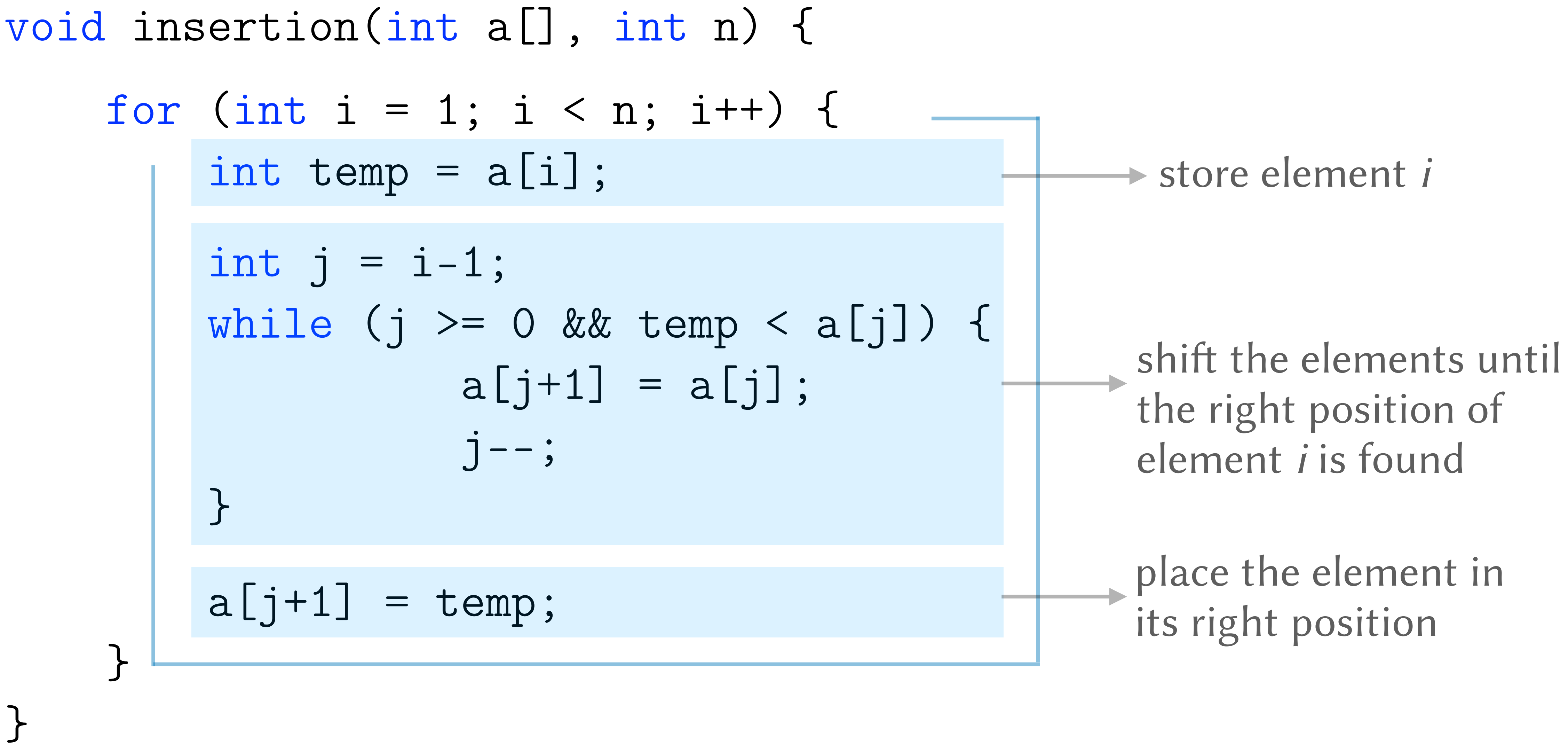
shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



0	1	2	3	4	5	6	7	8	9
-1	0	3	5	6	2	4	9	40	31



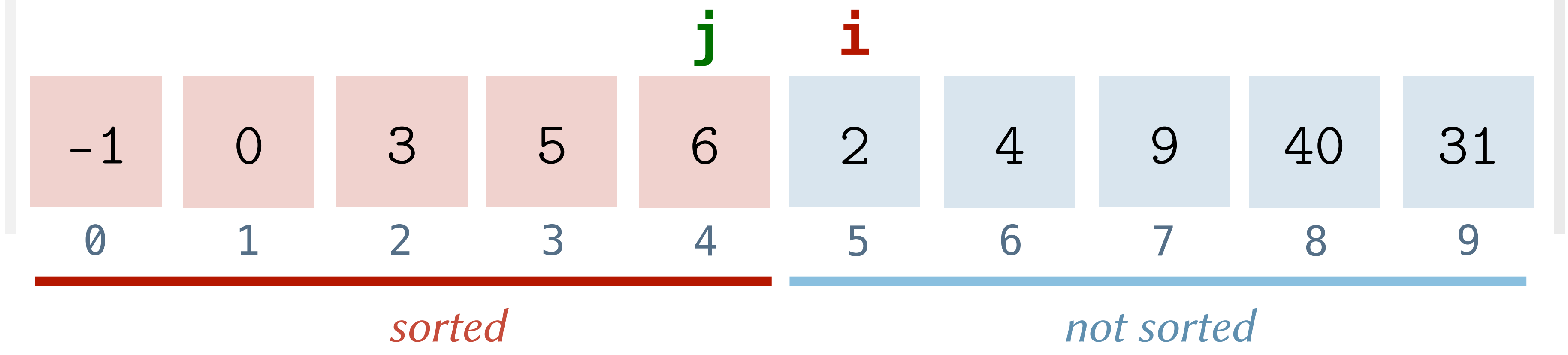
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



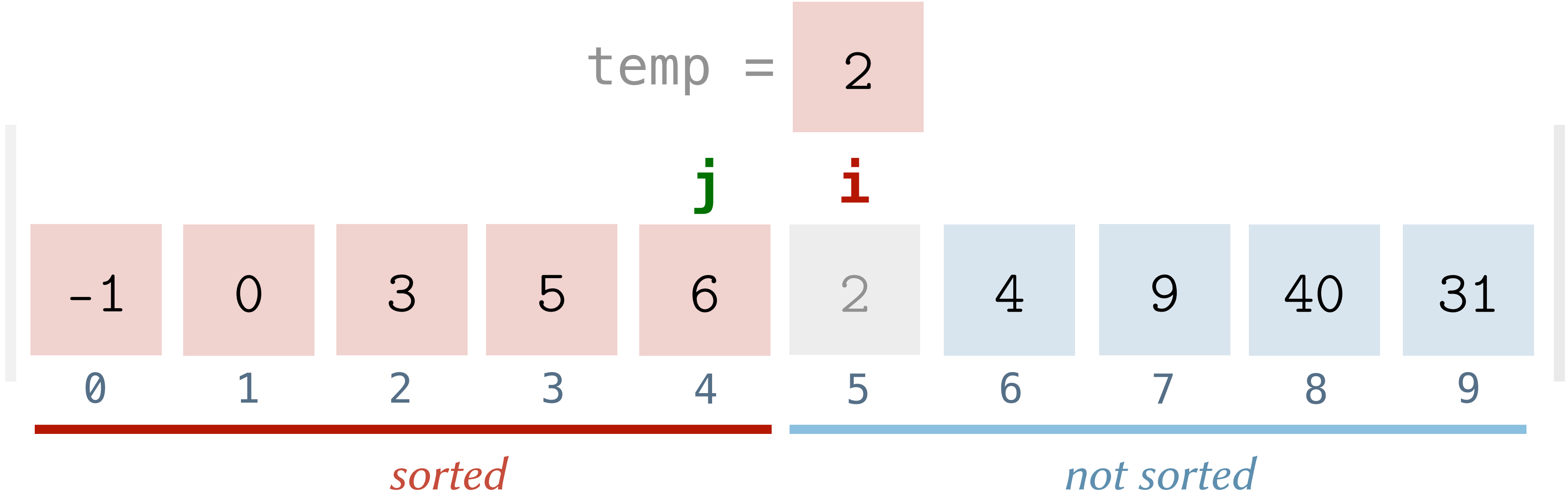
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



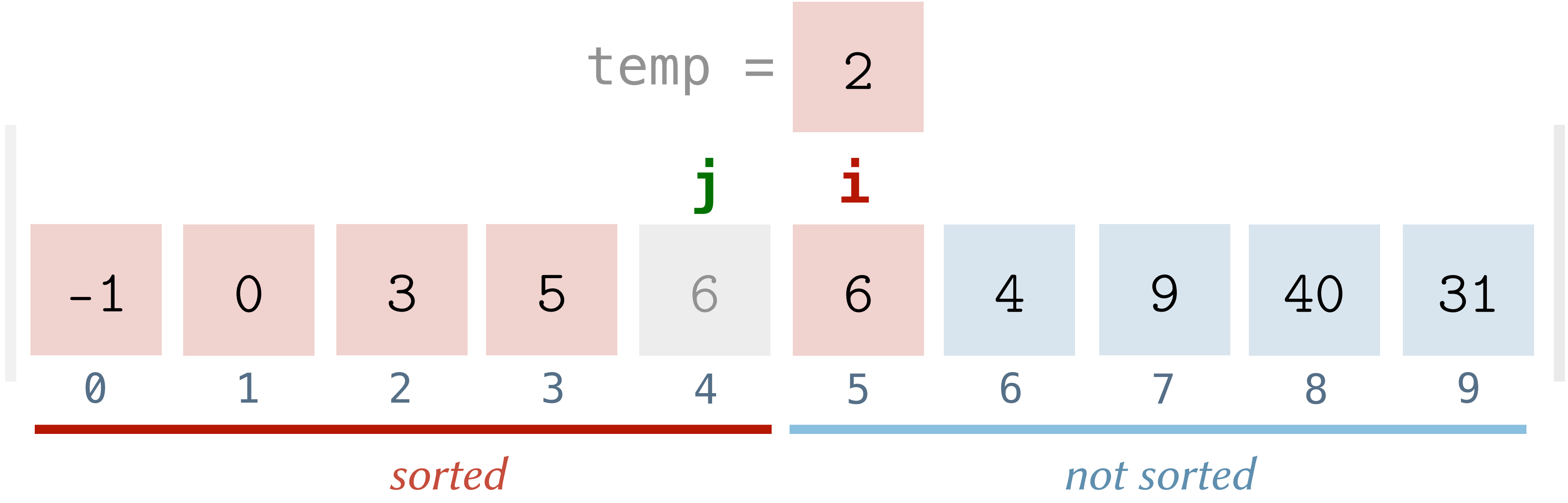
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



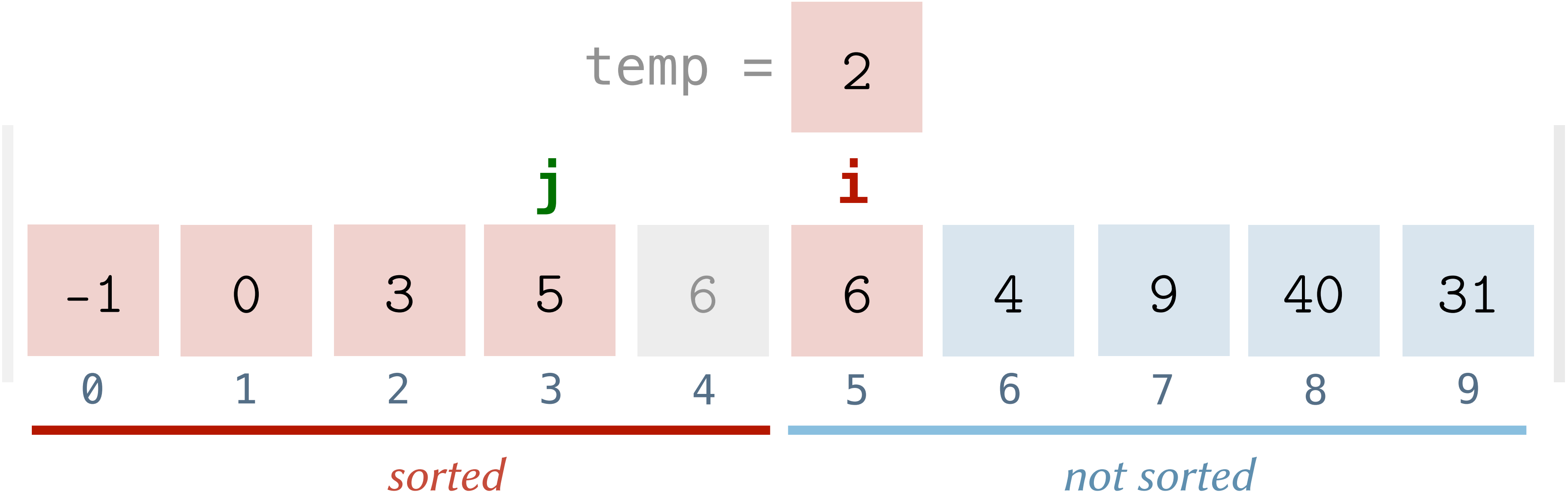
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



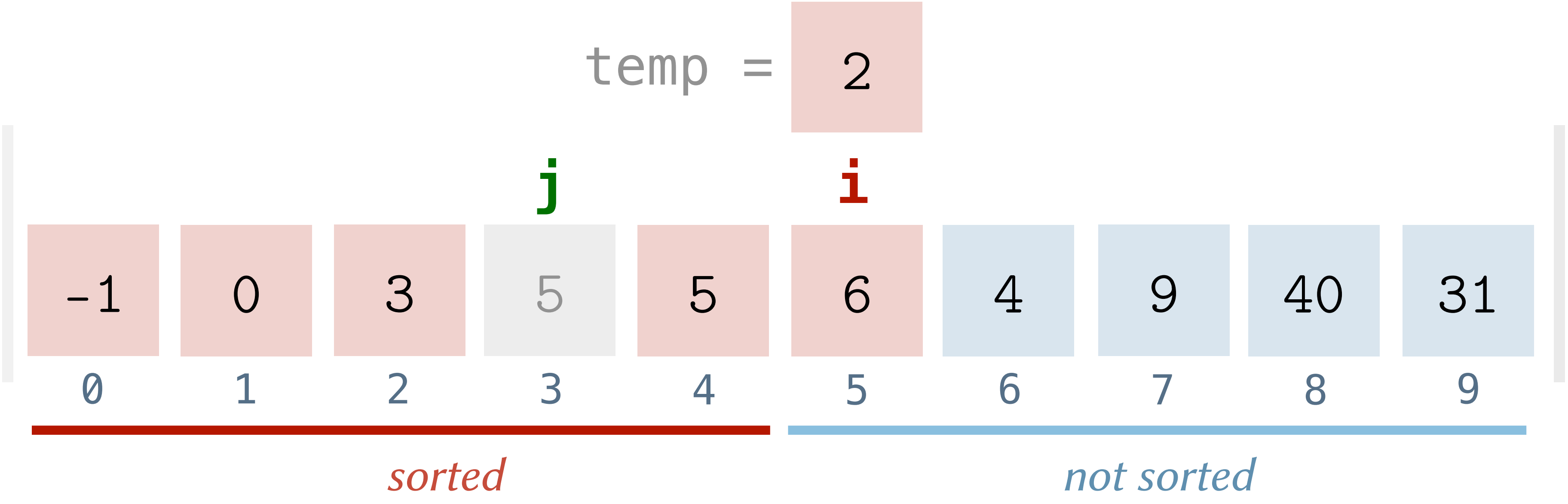
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



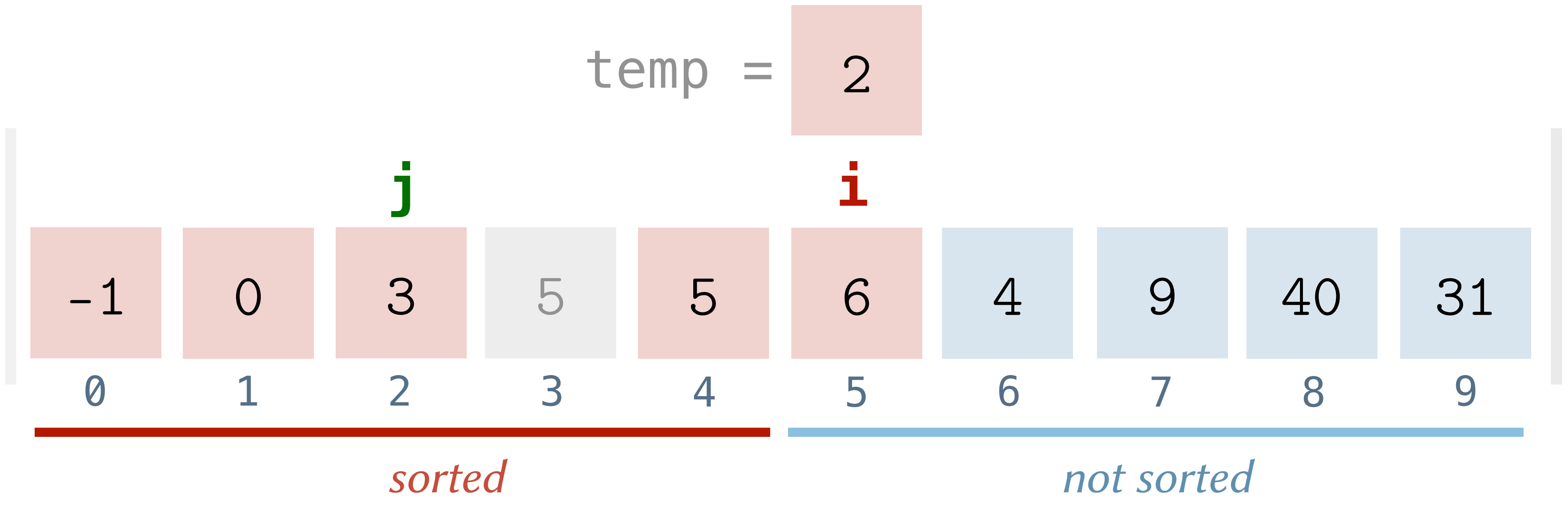
```
void insertion(int a[], int n) {

    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



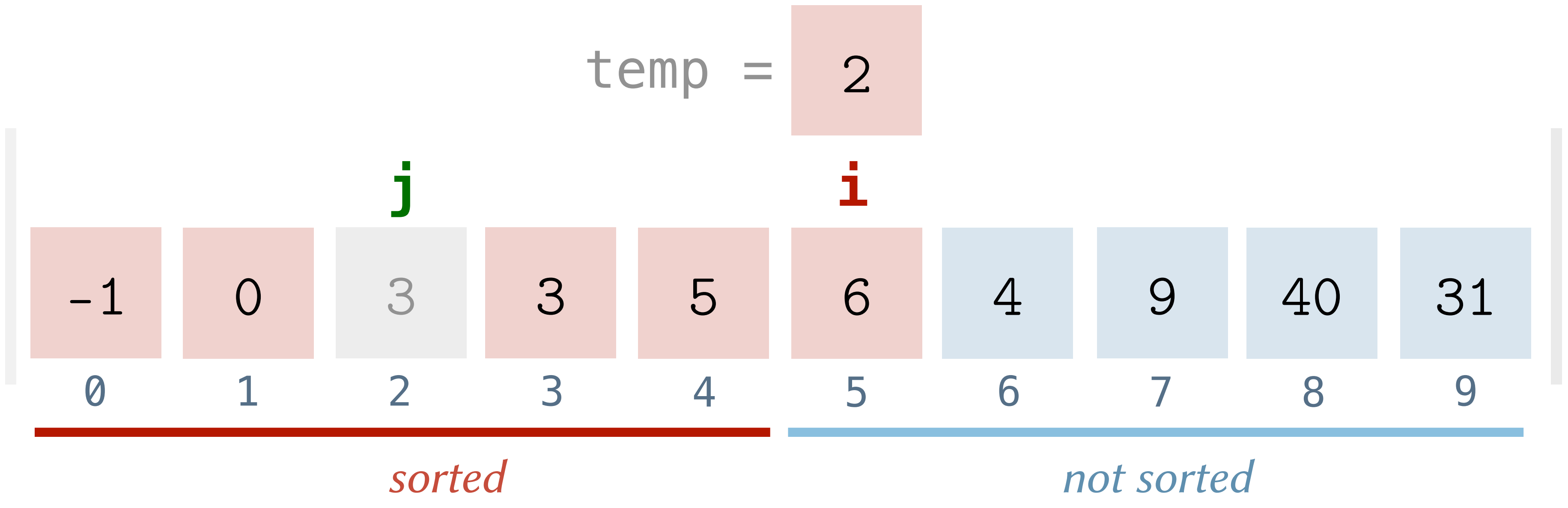
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



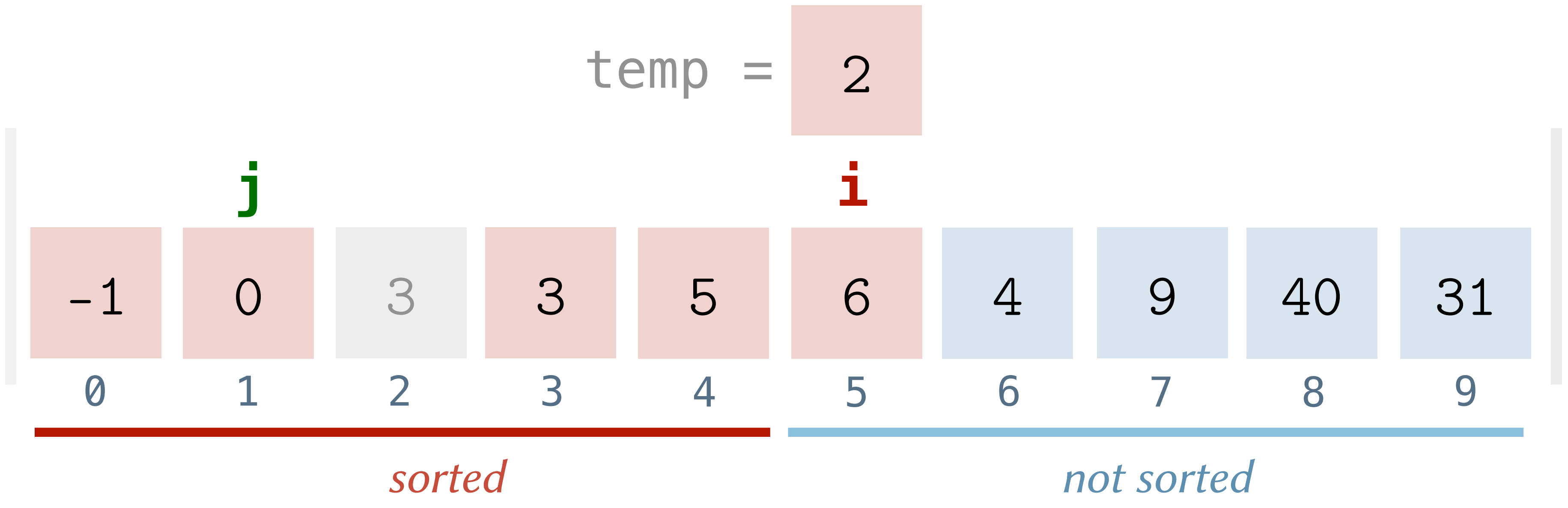
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



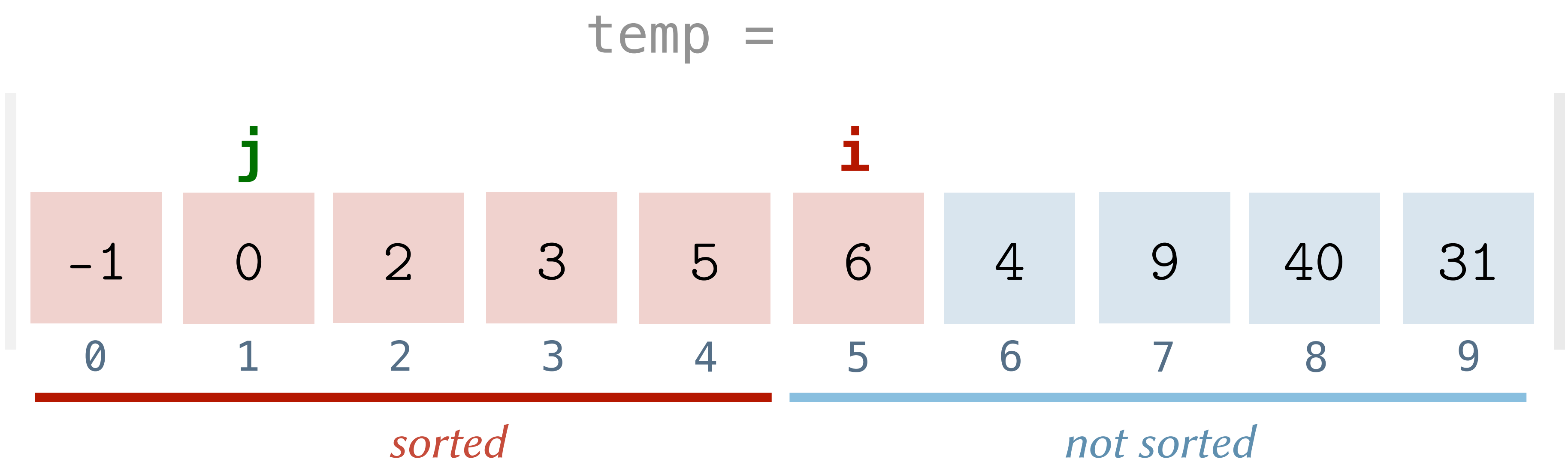
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



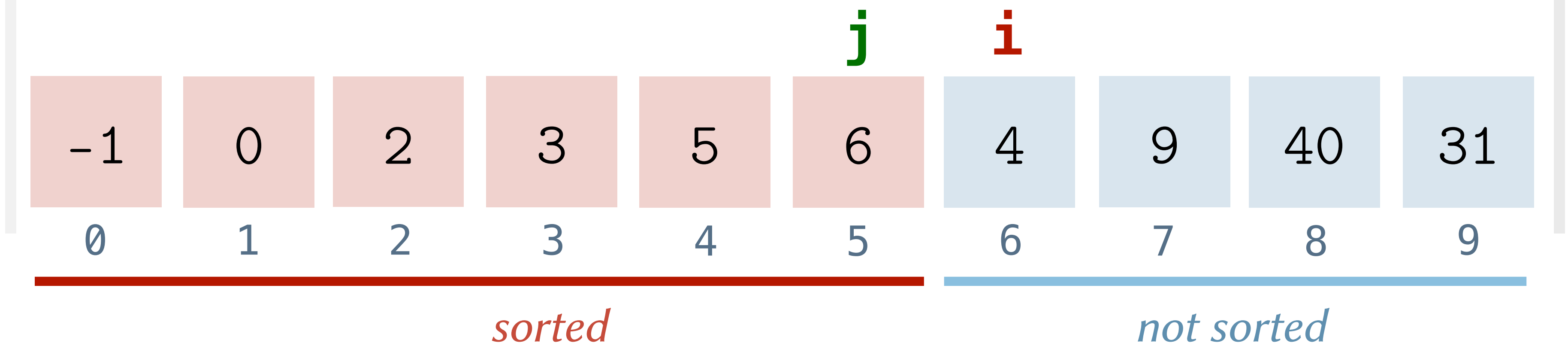
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

→ store element *i*

→ shift the elements until the right position of element *i* is found

→ place the element in its right position

Insertion Sort: Implementation



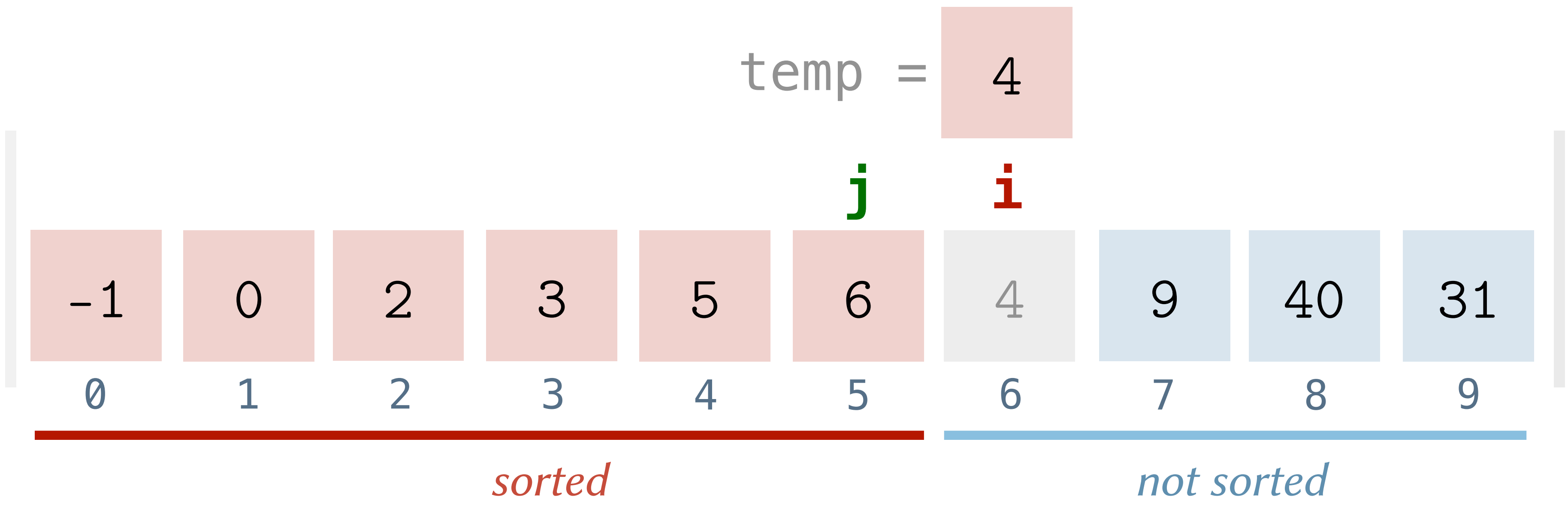
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



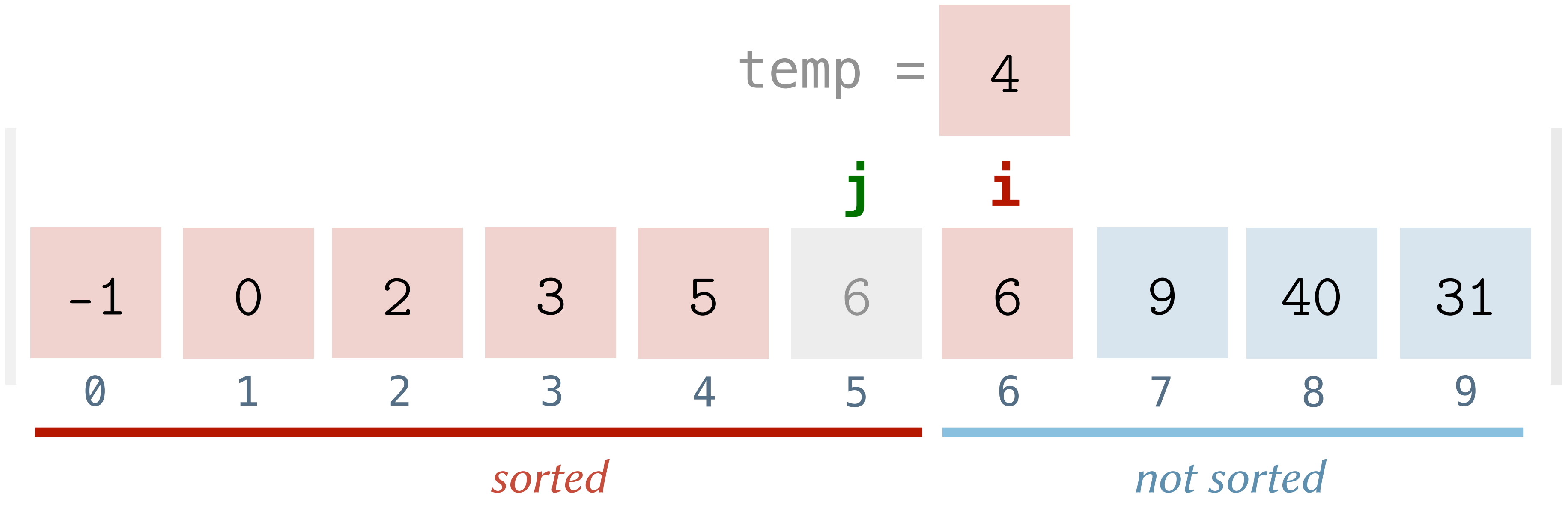
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



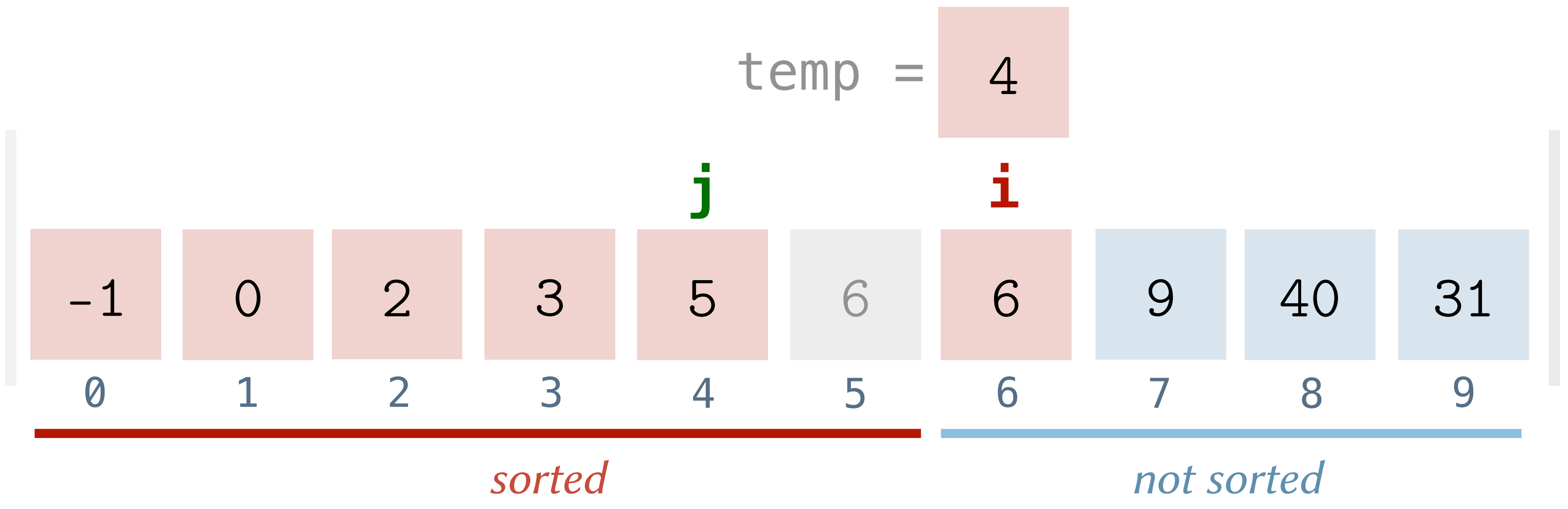
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



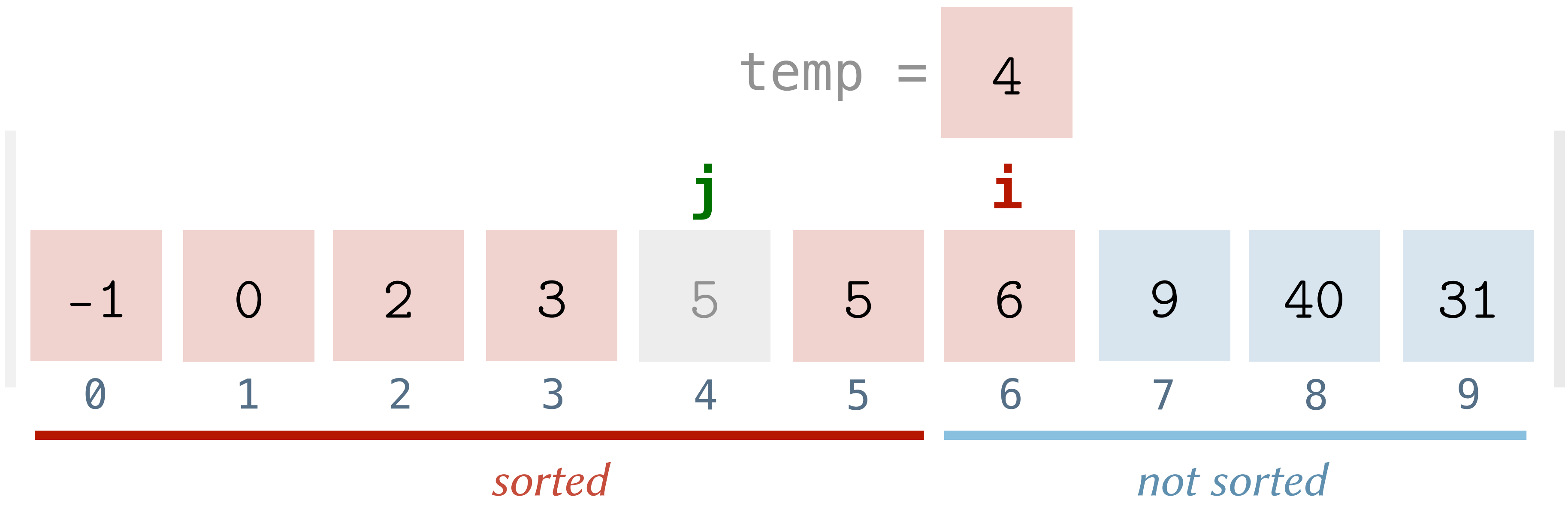
```
void insertion(int a[], int n) {

    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



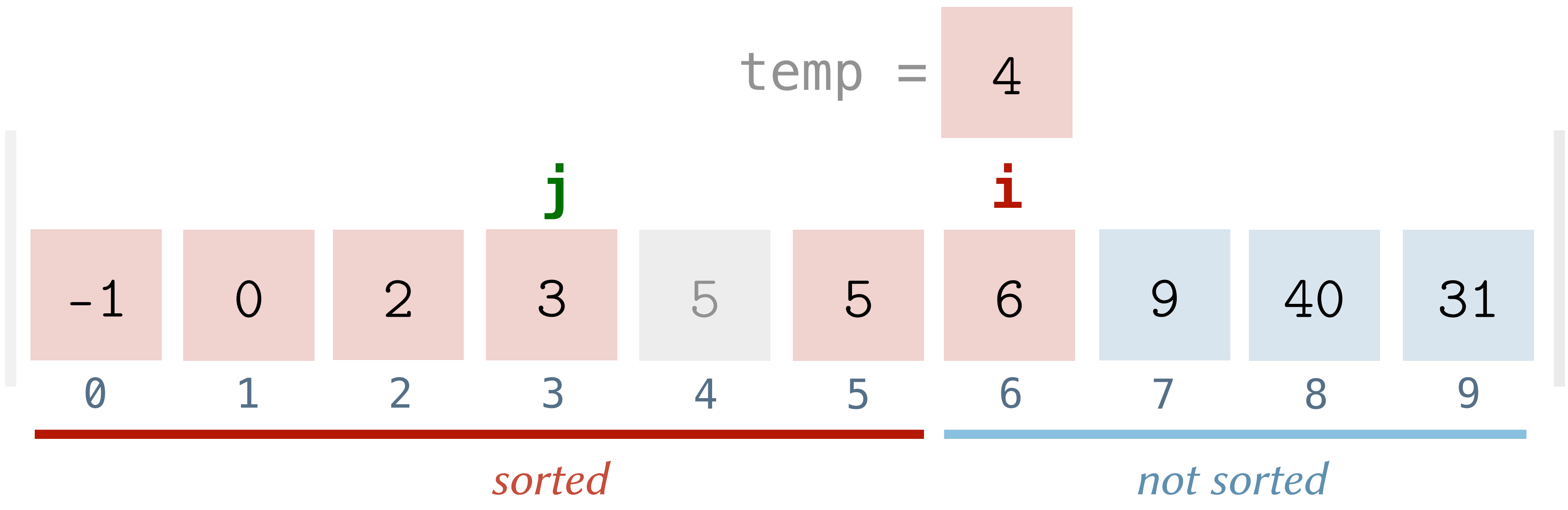
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



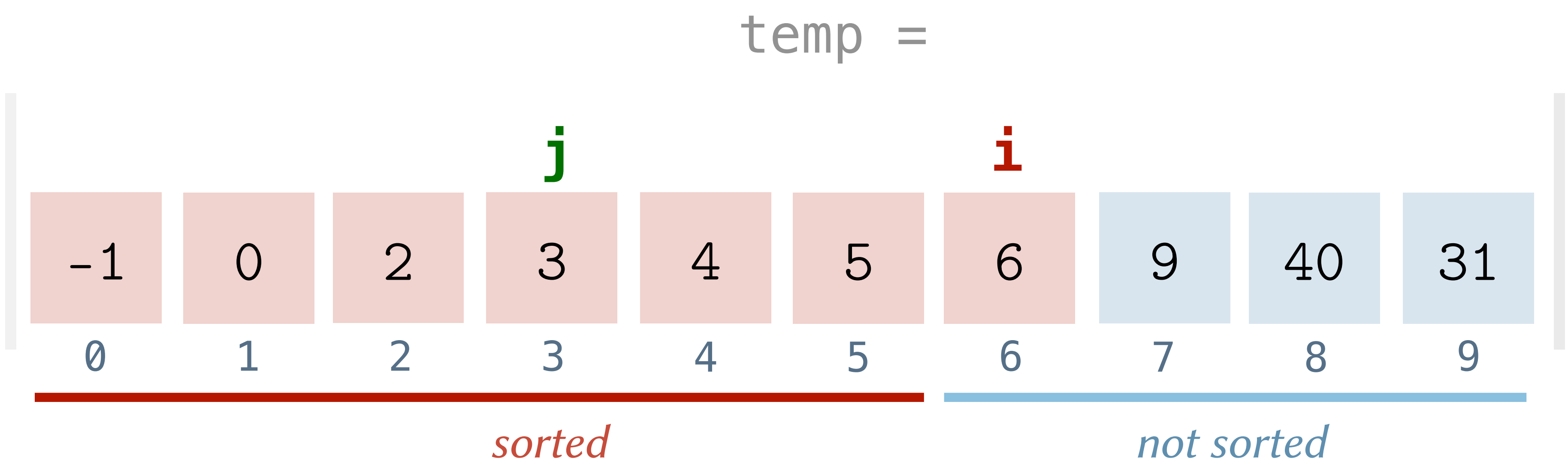
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



0	1	2	3	4	5	6	7	8	9
-1	0	2	3	4	5	6	9	40	31

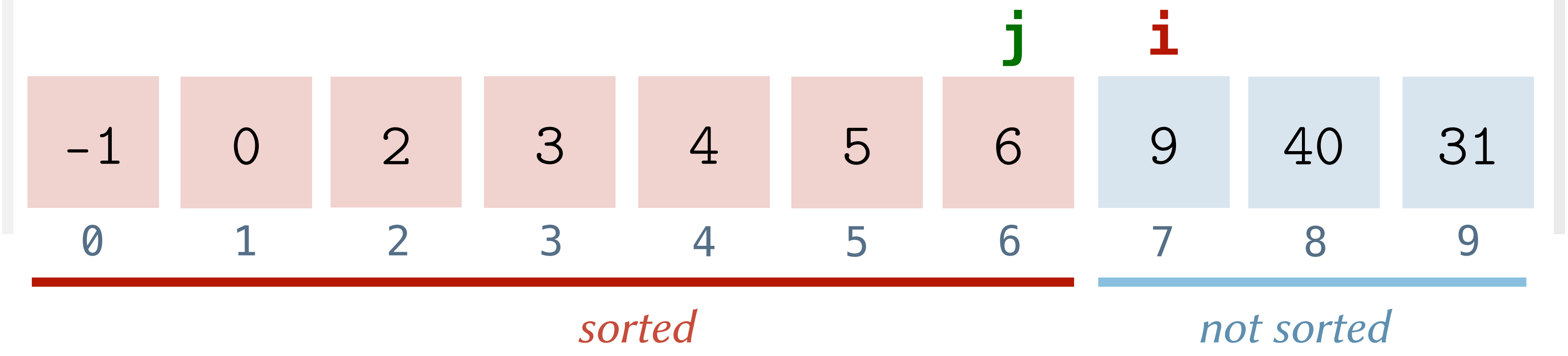
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



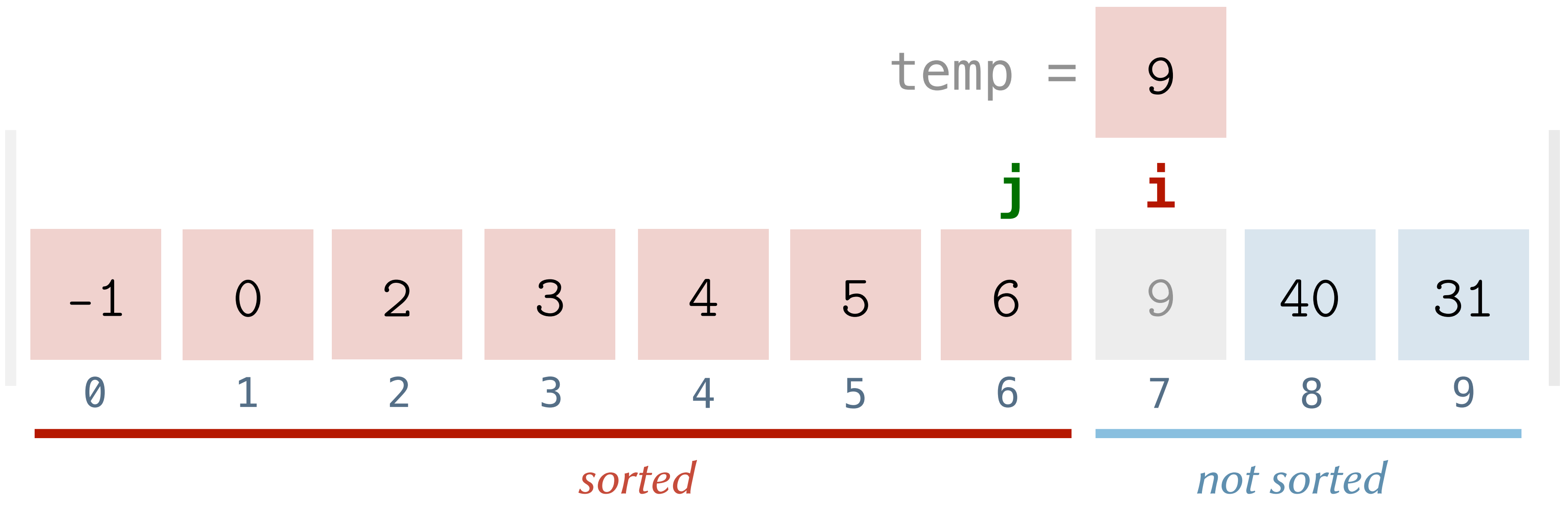
```c
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



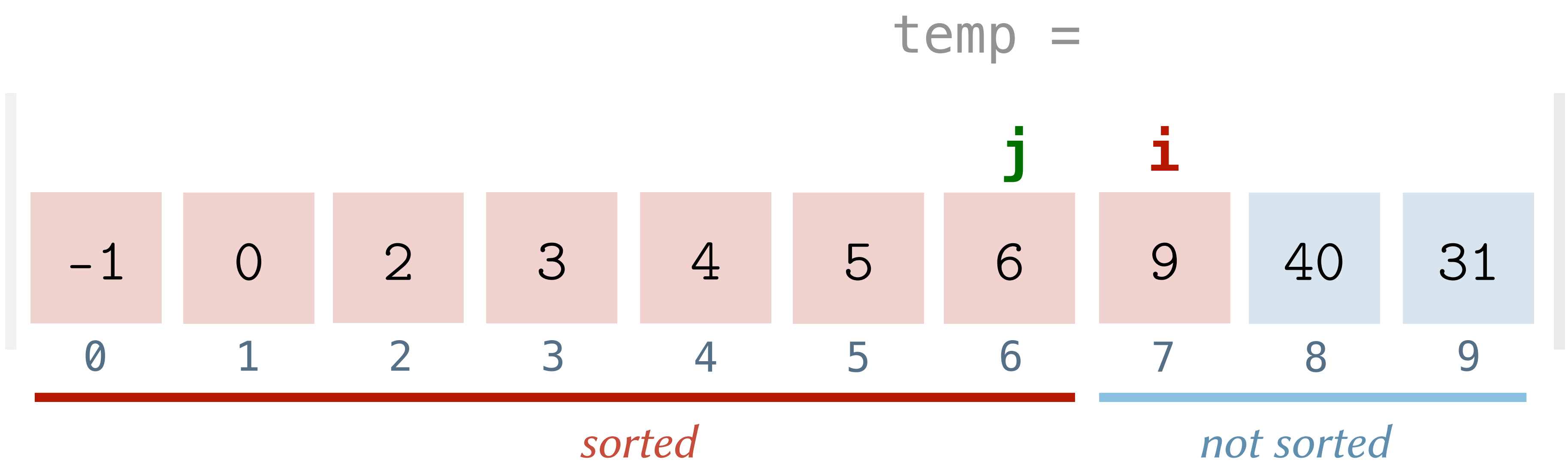
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



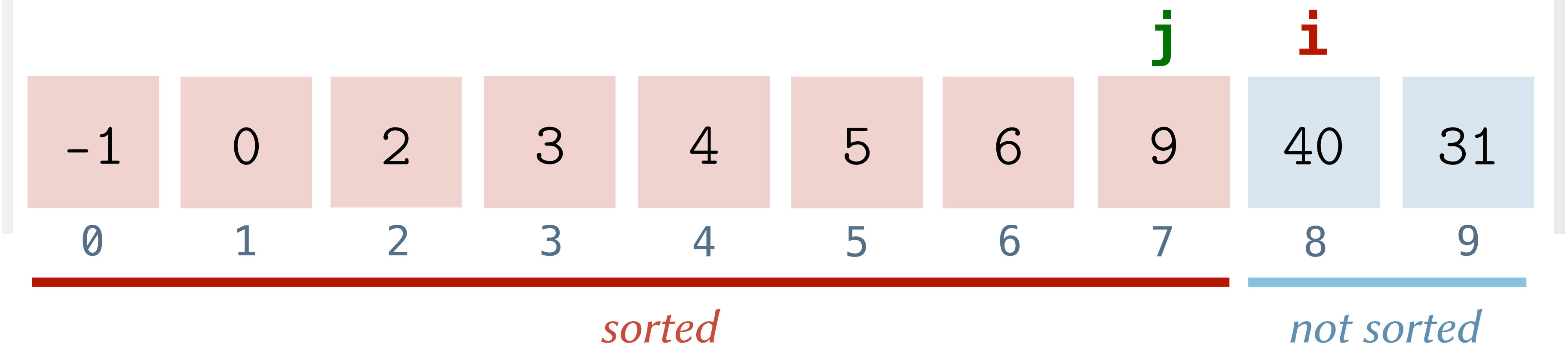
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



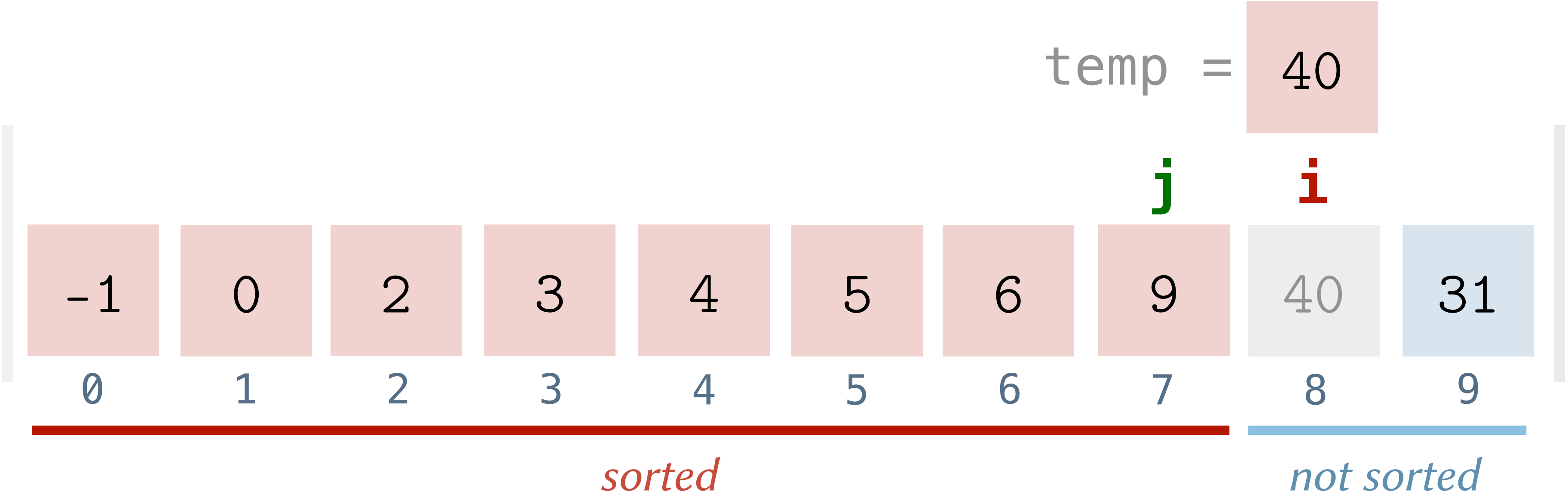
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



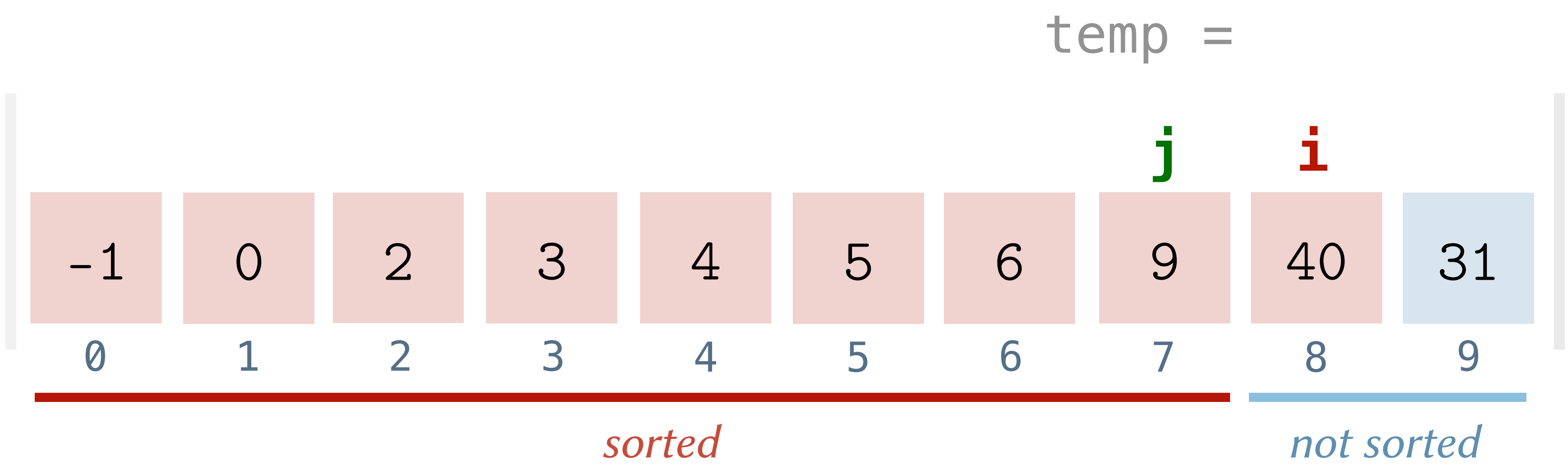
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



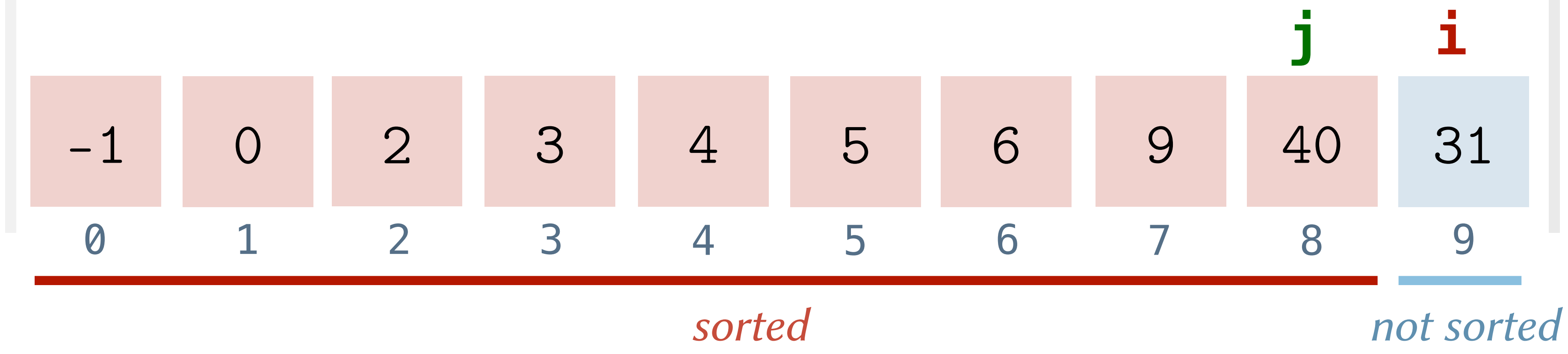
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



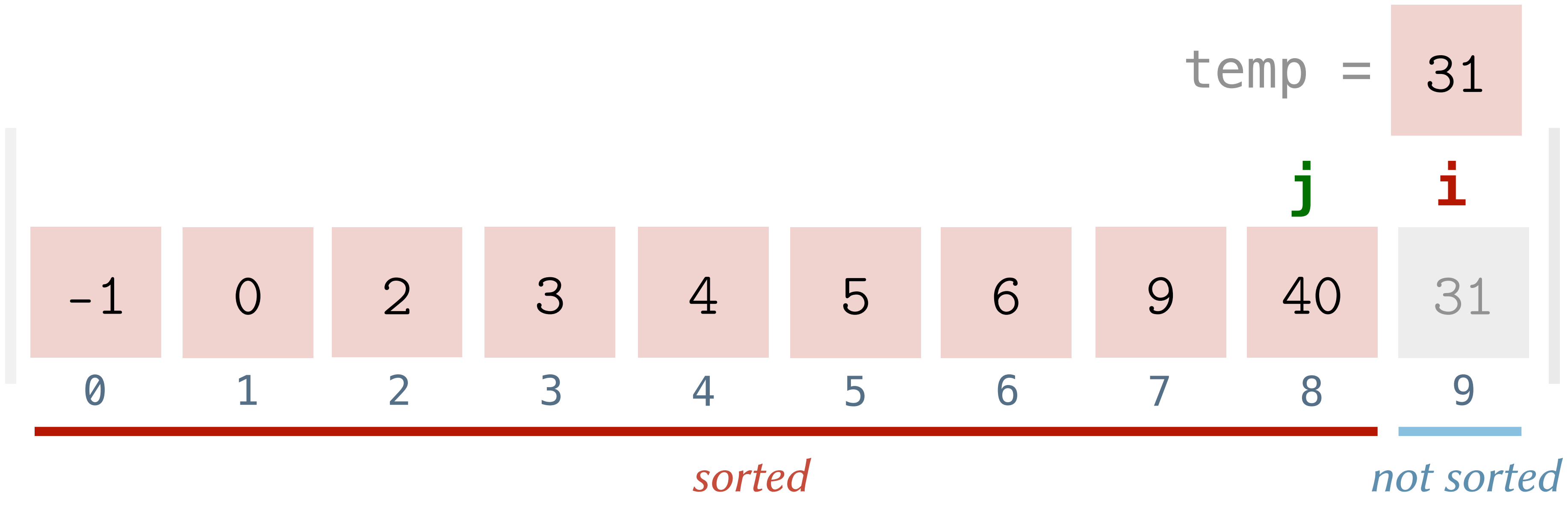
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



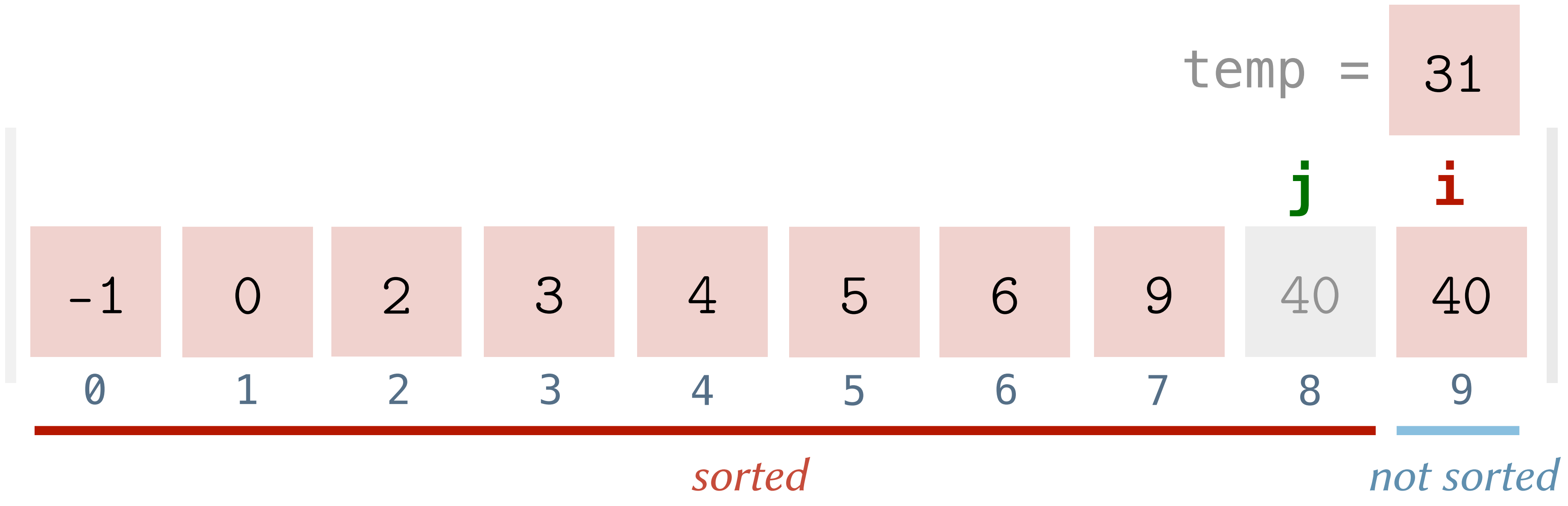
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

→ store element *i*

→ shift the elements until the right position of element *i* is found

→ place the element in its right position

Insertion Sort: Implementation



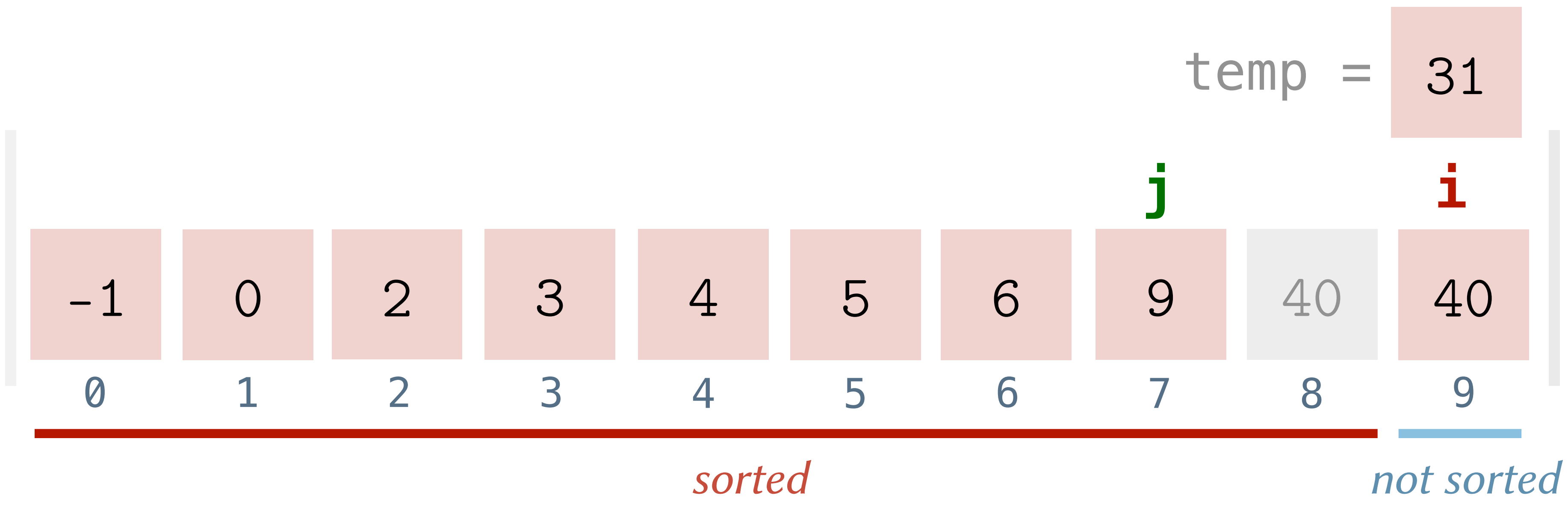
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                    a[j+1] = a[j];
                    j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



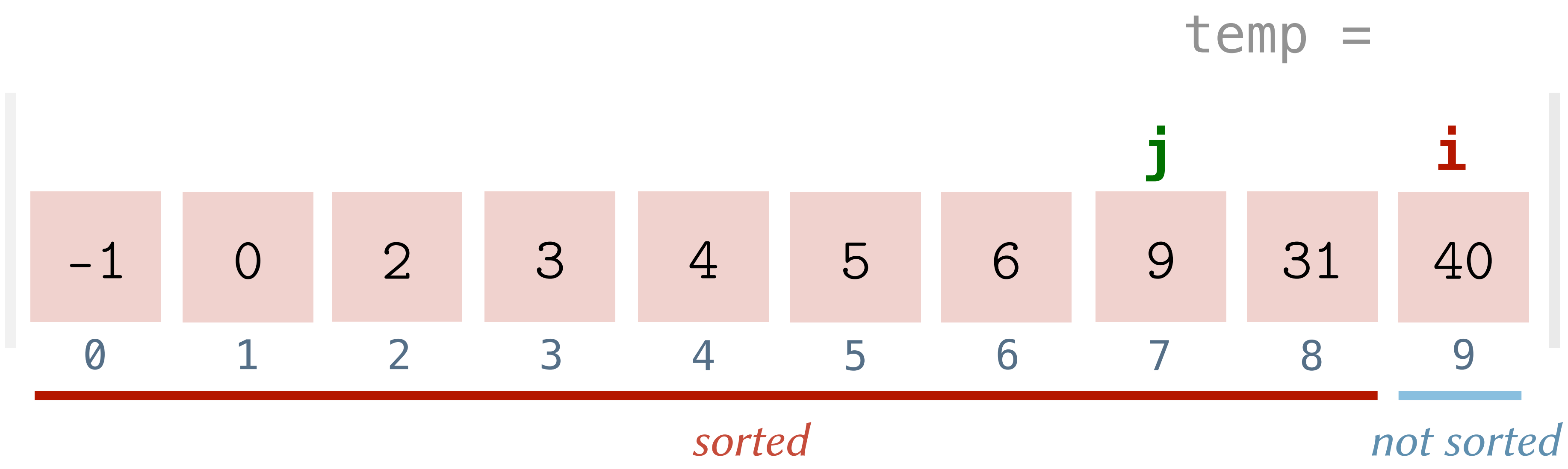
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



temp =

							j		i
-1	0	2	3	4	5	6	9	31	40
0	1	2	3	4	5	6	7	8	9

sorted

not sorted

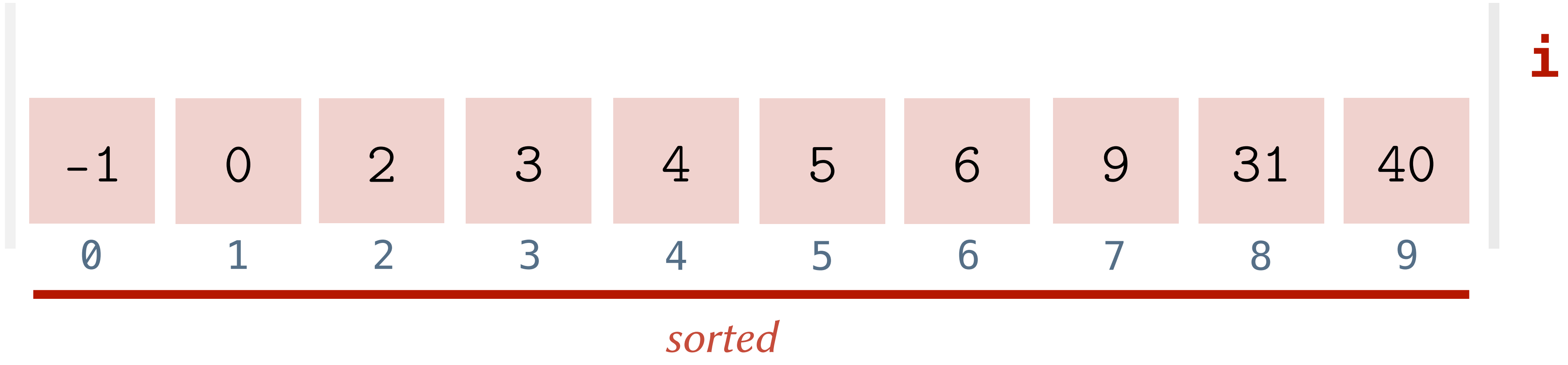
```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                a[j+1] = a[j];
                j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Insertion Sort: Implementation



-1	0	2	3	4	5	6	9	31	40
0	1	2	3	4	5	6	7	8	9

```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = a[i];

        int j = i-1;
        while (j >= 0 && temp < a[j]) {
                  a[j+1] = a[j];
                  j--;
        }

        a[j+1] = temp;
    }
}
```

store element *i*

shift the elements until the right position of element *i* is found

place the element in its right position

Analysis

```
void insertion(int a[], int n) {

   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

Worst Case.

Analysis

```
void insertion(int a[], int n) {
    for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
          a[j+1] = a[j];
      a[j+1] = temp;
    }
}
```

5 4 3 2 1

Worst Case. Reversely sorted arrays.

Analysis

```
void insertion(int a[], int n) {
   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

5 4 3 2 1

Insert **4**:

5 → 5 3 2 1 | **1** shift

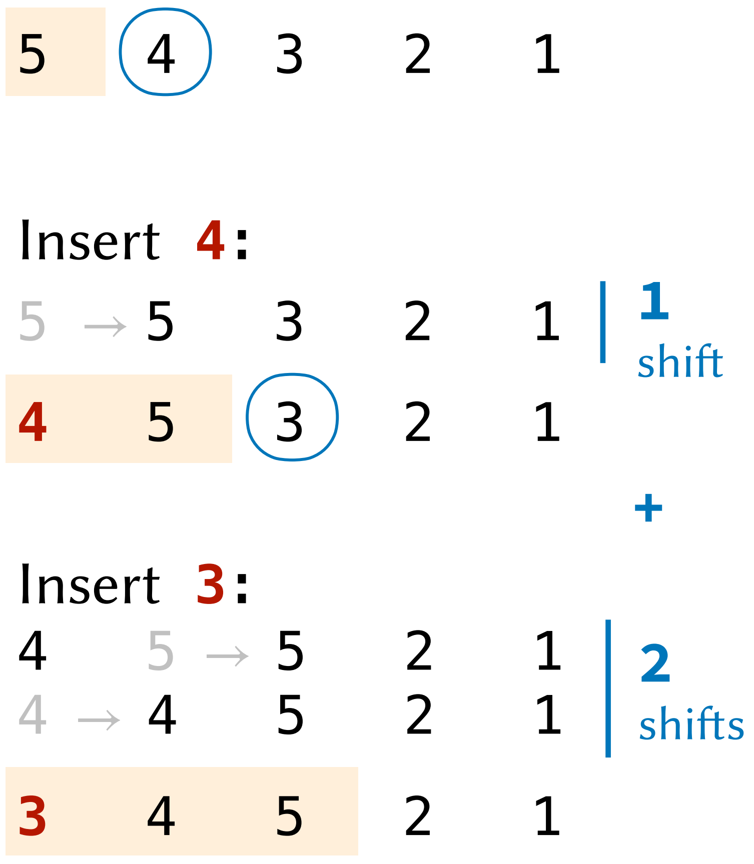
4 5 3 2 1

Worst Case. Reversely sorted arrays.

Analysis

```
void insertion(int a[], int n) {

   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

Worst Case. Reversely sorted arrays.


5 4 3 2 1

Insert 4:

5 → 5 3 2 1 | 1 shift

4 5 3 2 1

+

Insert 3:

4 5 → 5 2 1

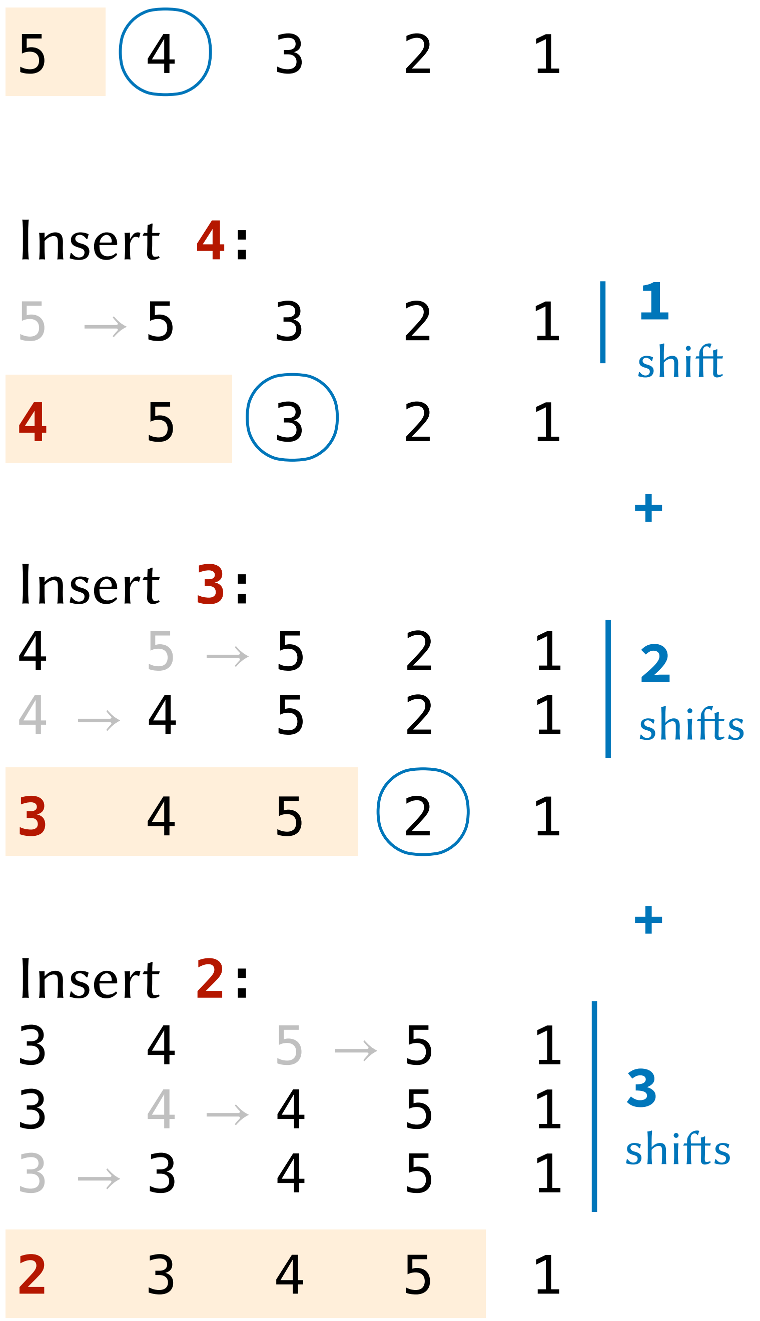
4 → 4 5 2 1 | 2 shifts

3 4 5 2 1

Analysis

```
void insertion(int a[], int n) {

   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

Worst Case. Reversely sorted arrays.



5 4 3 2 1

Insert **4**:

5 → 5 3 2 1 | **1** shift

4 5 3 2 1

+

Insert **3**:

4 5 → 5 2 1

4 → 4 5 2 1 | **2** shifts

3 4 5 2 1

+

Insert **2**:

3 4 5 → 5 1

3 4 → 4 5 1

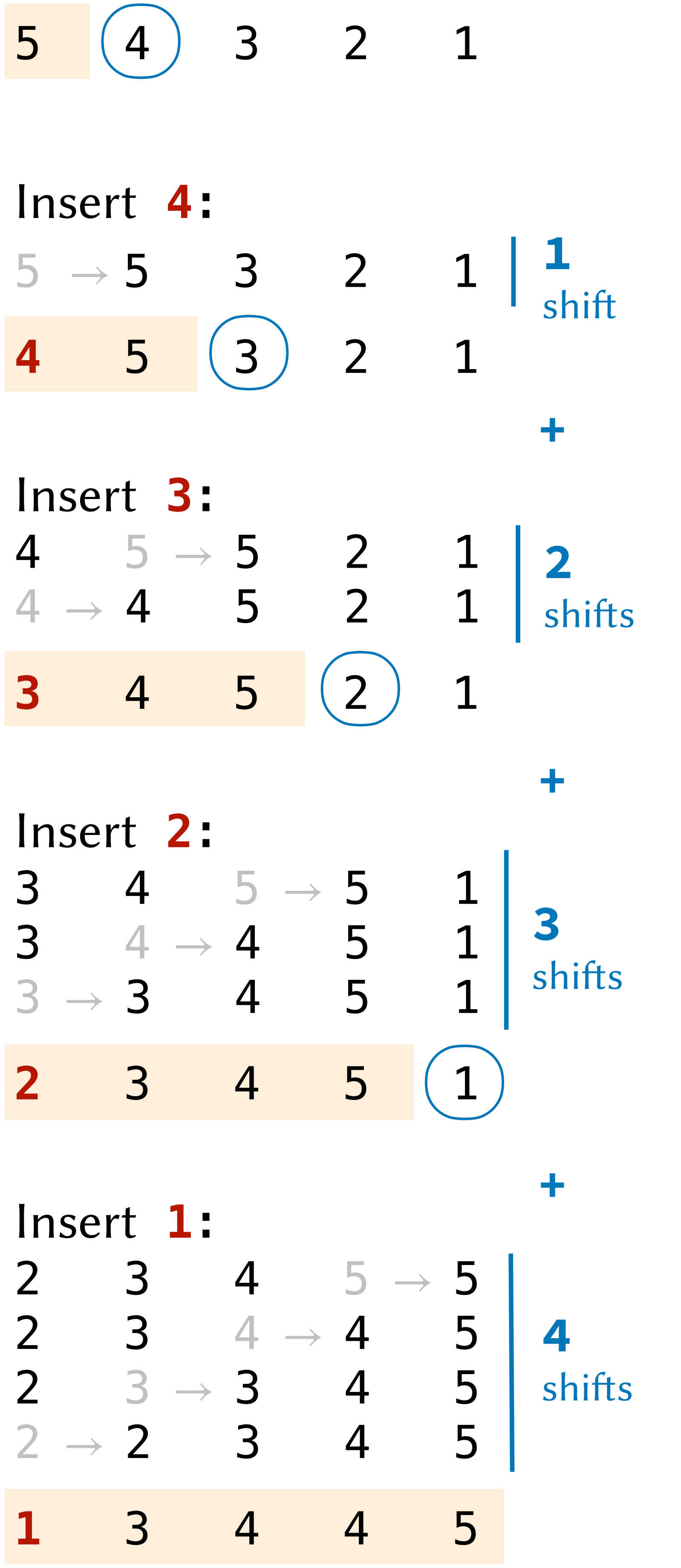
3 → 3 4 5 1 | **3** shifts

2 3 4 5 1

Analysis

```
void insertion(int a[], int n) {

   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
          a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

Worst Case. Reversely sorted arrays.



```
5  4  3  2  1

Insert 4:
5 → 5  3  2  1    1 shift
4  5  3  2  1
                  +
Insert 3:
4  5 → 5  2  1
4 → 4  5  2  1    2 shifts
3  4  5  2  1
                  +
Insert 2:
3  4  5 → 5  1
3  4 → 4  5  1
3 → 3  4  5  1    3 shifts
2  3  4  5  1
                  +
Insert 1:
2  3  4  5 → 5
2  3  4 → 4  5
2  3 → 3  4  5
2 → 2  3  4  5    4 shifts
1  3  4  4  5
```

Analysis

```
void insertion(int a[], int n) {

   for (int i = 1; i < n; i++) {
      int temp = a[i];
      int j;
      for (j = i-1; j >= 0 && temp < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = temp;
   }
}
```

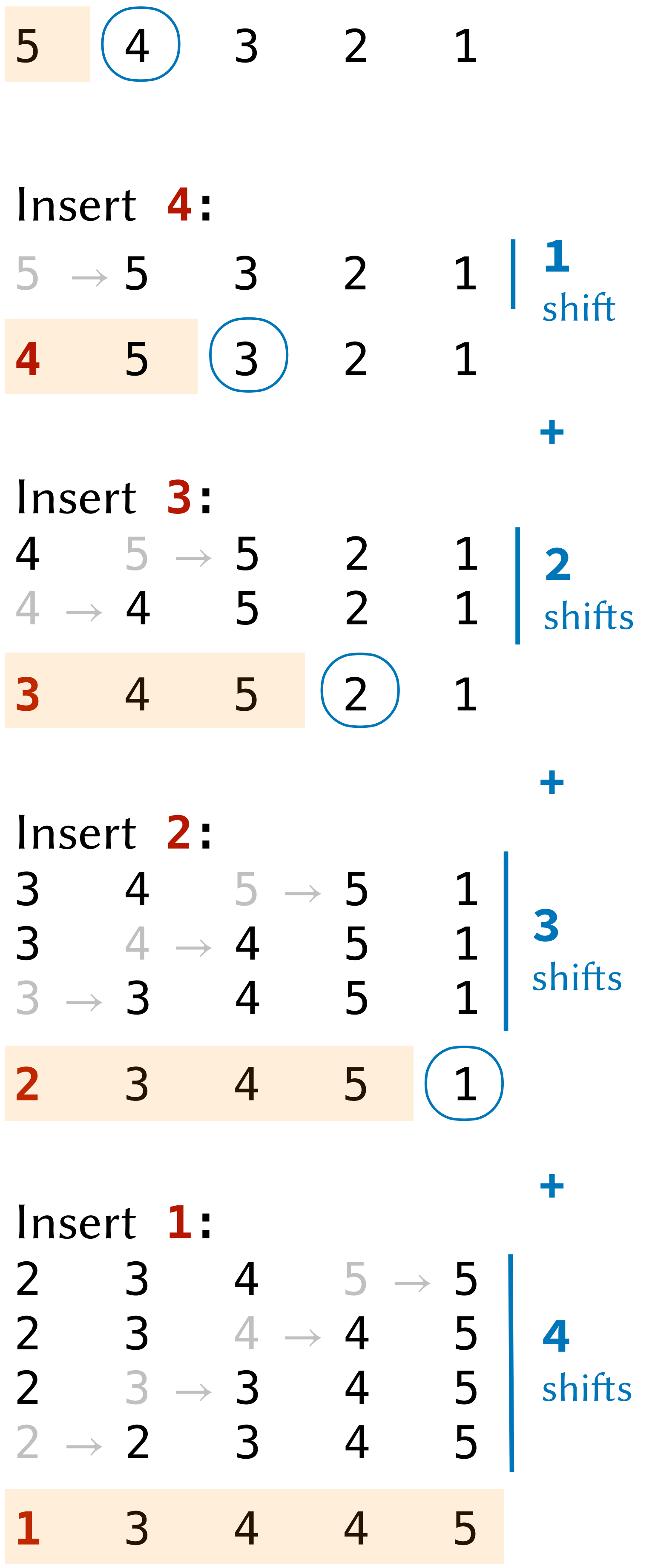
Worst Case. Reversely sorted arrays.

Data compares. $$1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$$

Number of shifts. $$1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$$

Data moves. Number of shifts + $2(n-1)$
For moving `a[i]` to `temp` and then `temp` to `a[j+1]`

Total. $O(n^2)$



5 4 3 2 1

Insert **4**:
5 → 5 3 2 1
4 5 3 2 1
1 shift

+

Insert **3**:
4 5 → 5 2 1
4 → 4 5 2 1
3 4 5 2 1
2 shifts

+

Insert **2**:
3 4 5 → 5 1
3 4 → 4 5 1
3 → 3 4 5 1
2 3 4 5 1
3 shifts

+

Insert **1**:
2 3 4 5 → 5
2 3 4 → 4 5
2 3 → 3 4 5
2 → 2 3 4 5
1 3 4 4 5
4 shifts

Analysis

Best Case.

Analysis

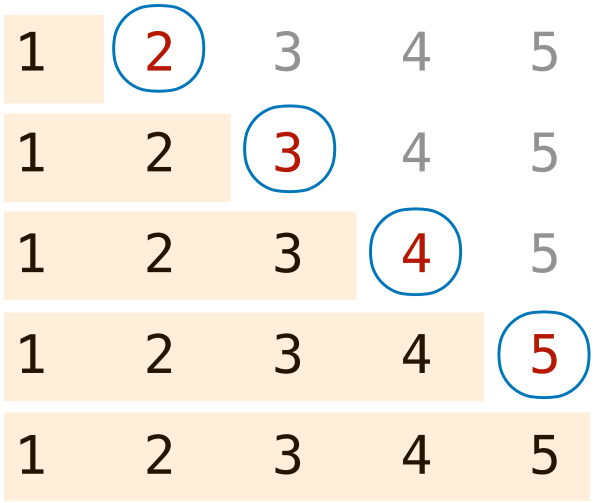
Best Case. Sorted arrays.

Data compares. $n - 1$ (each element is compared to the one to its left)

Number of shifts. 0 (all elements are in their place)

Data moves. Number of shifts + $2(n - 1)$
For moving `a[i]` to `temp` and then back to its place.

Total. $O(n)$



1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

Analysis

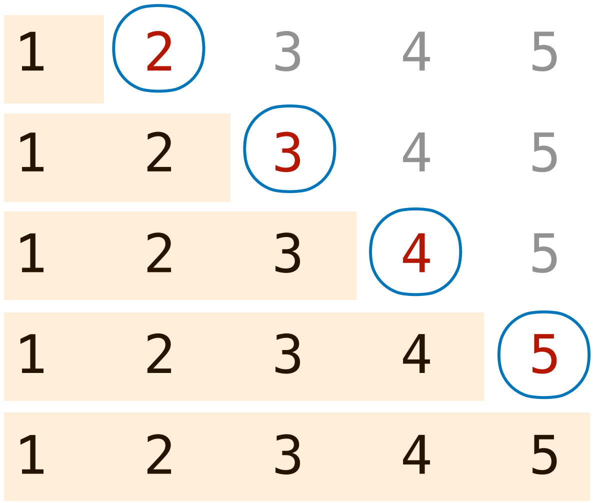
Best Case. Sorted arrays.

Data compares. $n - 1$ (each element is compared to the one to its left)

Number of shifts. 0 (all elements are in their place)

Data moves. Number of shifts + $2(n - 1)$
For moving `a[i]` to `temp` and then back to its place.

Total. $O(n)$



1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

A Good Case. Partially sorted arrays

Total. $O(n)$

Intuition. If every element is either in its correct position or only a few steps away from it, we need a few data compares and moves for every element, which makes the total $O(n)$.

Example

1 2 3 5 ← 4 6 7 10 ← 8 ← 9 11 13 ← 12

[Optional Info] Insertion sort performs a number of shifts that is equal to the number of inversions. A sorted array has 0 inversions, a partially sorted array has a number of inversions that is linear in the size of the array and a reversely sorted array has $\frac{1}{2}n(n-1)$ inversions.

Analysis

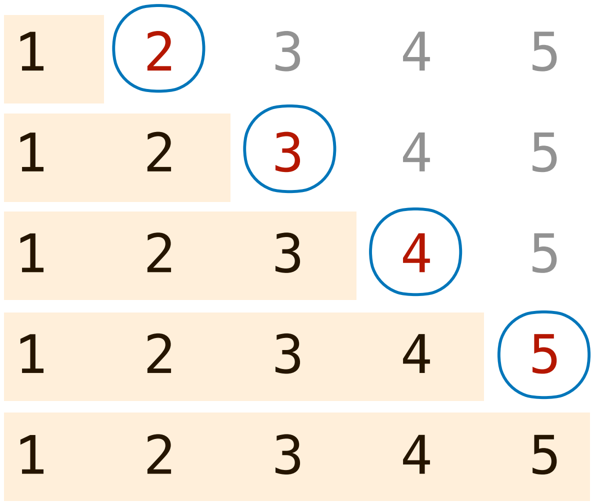
Best Case. Sorted arrays.

Data compares. $n - 1$ (each element is compared to the one to its left)

Number of shifts. 0 (all elements are in their place)

Data moves. Number of shifts + $2(n - 1)$
For moving `a[i]` to `temp` and then back to its place.

Total. $O(n)$



1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

A Good Case. Partially sorted arrays

Total. $O(n)$

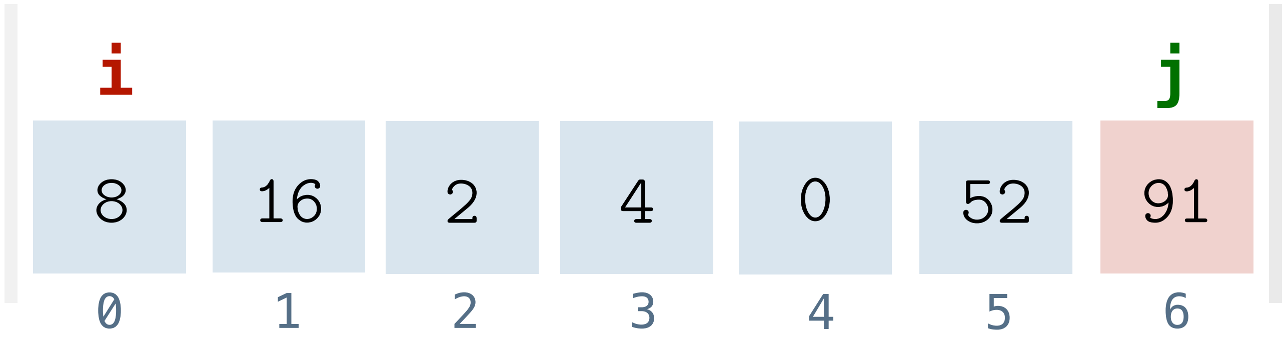
Intuition. If every element is either in its correct position or only a few steps away from it, we need a few data compares and moves for every element, which makes the total $O(n)$.

Average Case. Random arrays.

Claim. Insertion sort requires for sorting a random array around half the amount of data moves and data compares it needs for sorting a reversely sorted array.

Intuition. If elements are random, then each element moves around half the elements to its left before being inserted in its position. I.e. $\frac{1}{2}(1)+\frac{1}{2}(2)+\frac{1}{2}(3) + \ldots + \frac{1}{2}(n-1) = \frac{1}{4}n(n-1)$ shifts.

Bubble Sort: Implementation

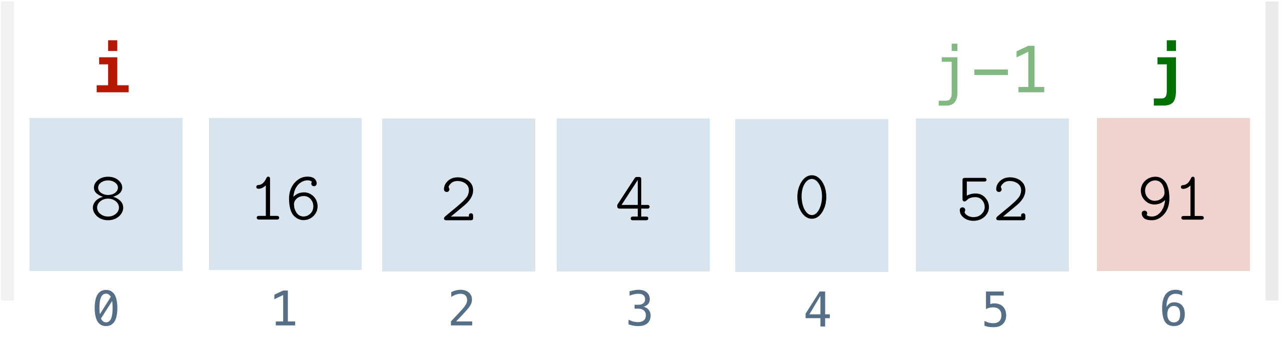


8	16	2	4	0	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {




        }


    }
}
```

compare adjacent elements and swap if not in order

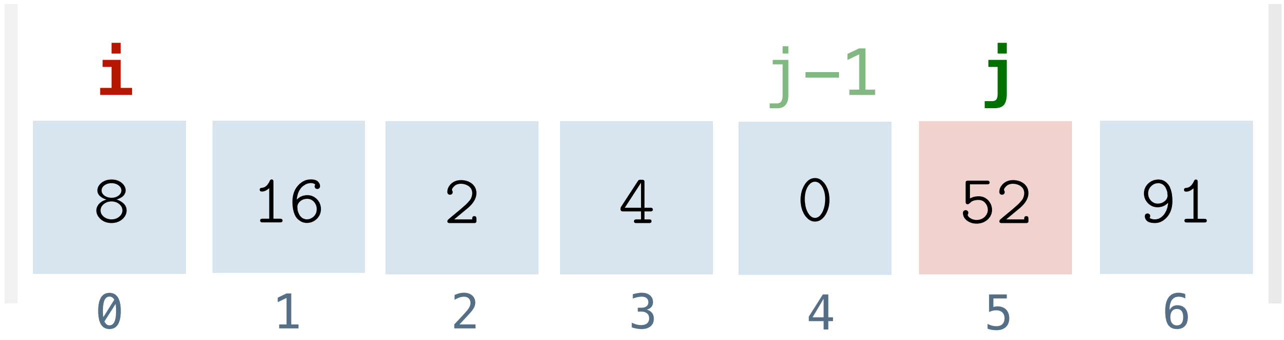
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }



    }
}
```

compare adjacent elements and swap if not in order

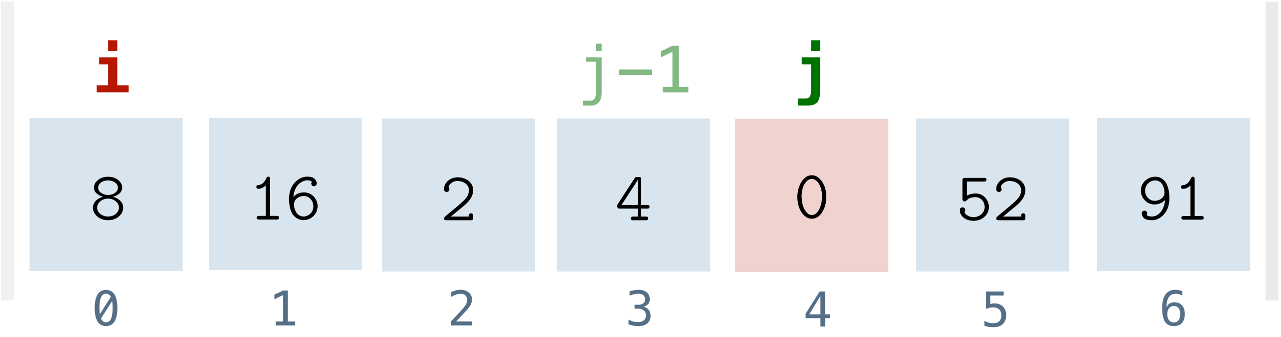
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

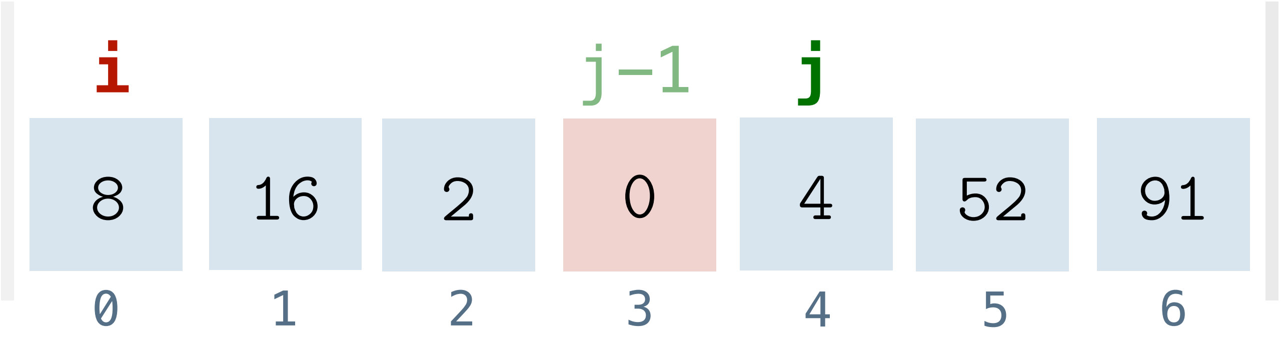
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }
    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

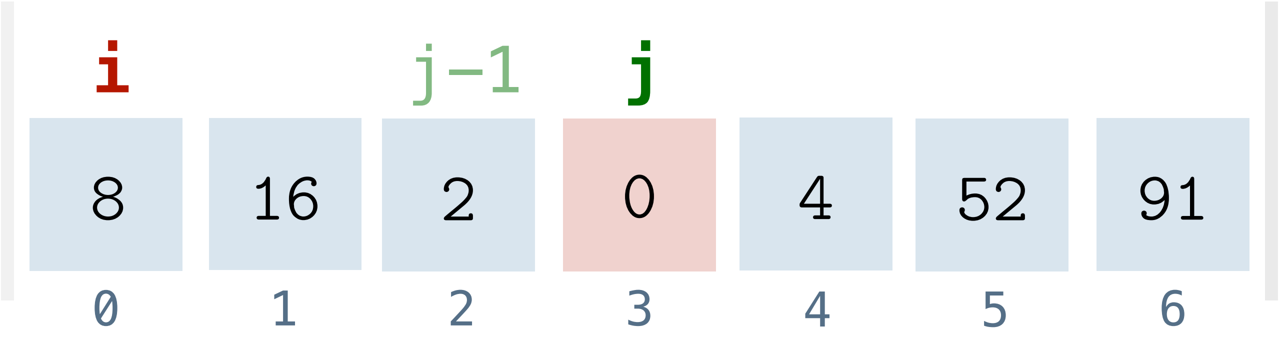


i			j-1	j		
8	16	2	0	4	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

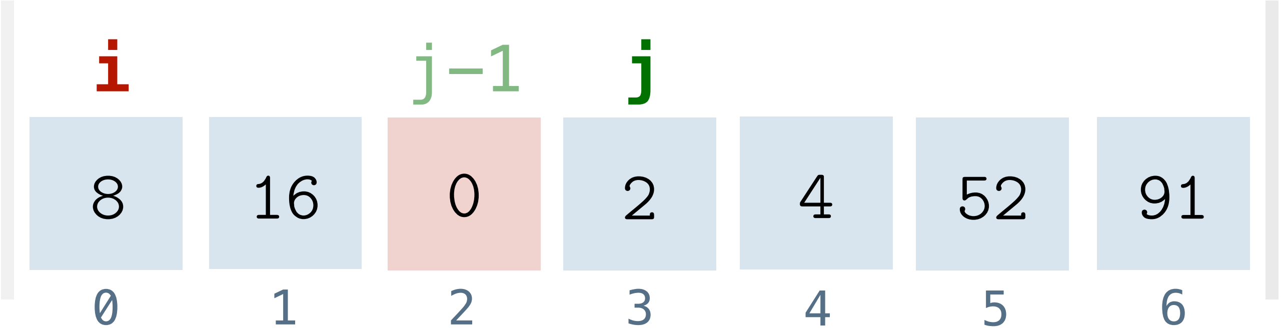


i		j-1	j			
8	16	2	0	4	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

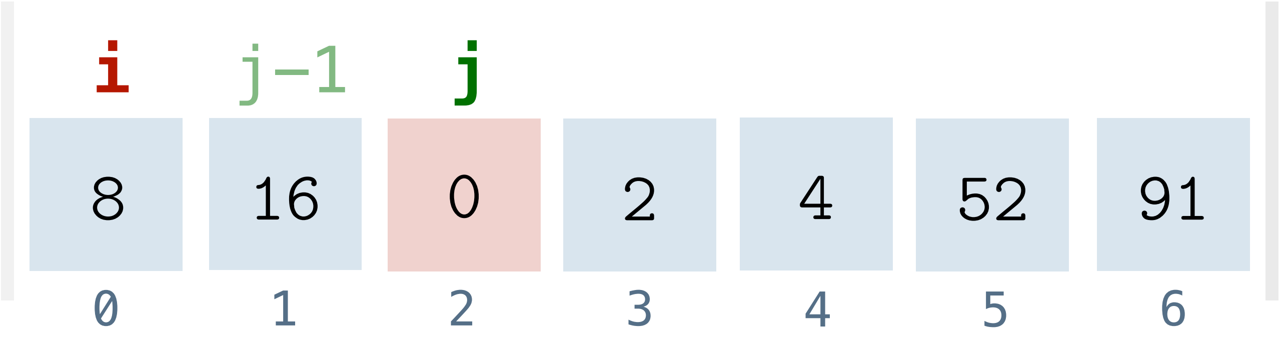
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

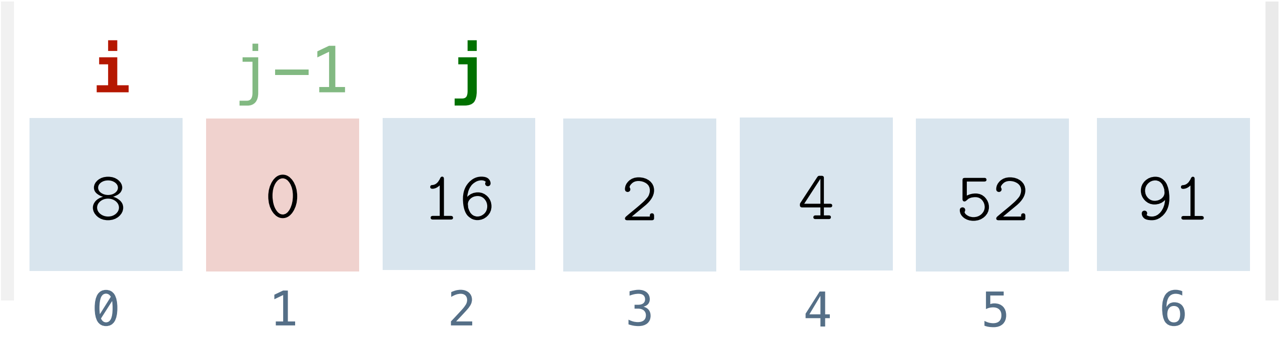
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }



    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

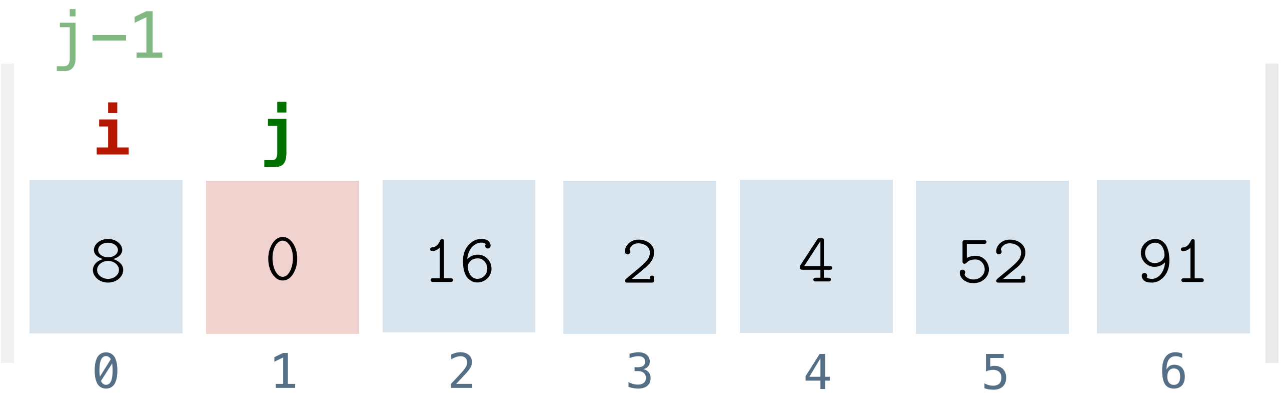


i	j-1	j				
8	0	16	2	4	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

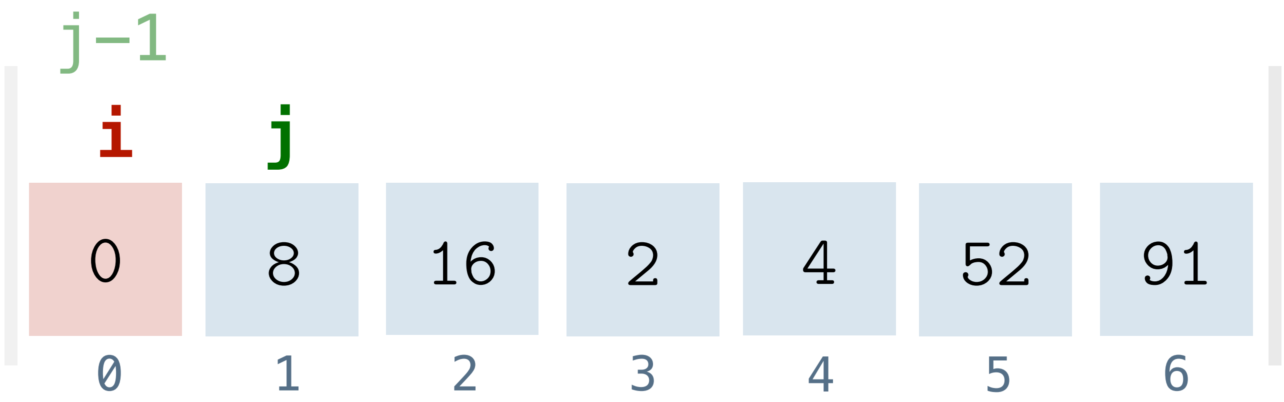
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }
    }
}
```

compare adjacent elements and swap if not in order

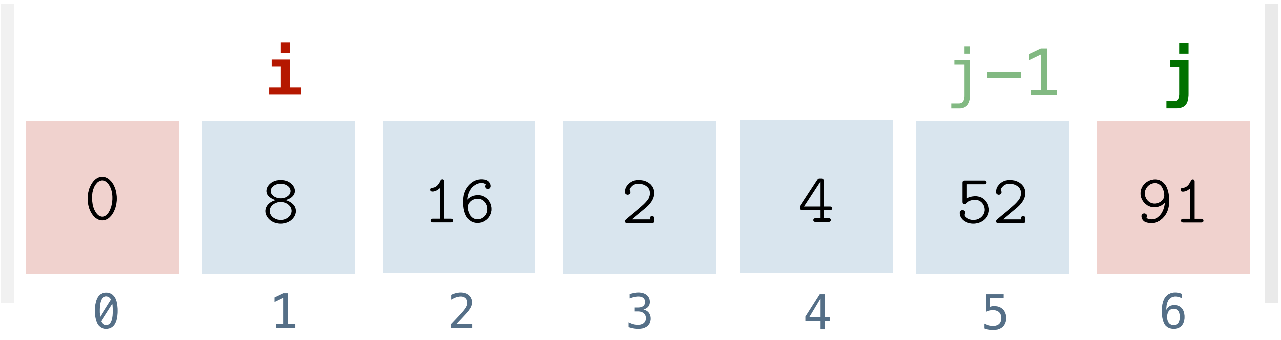
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

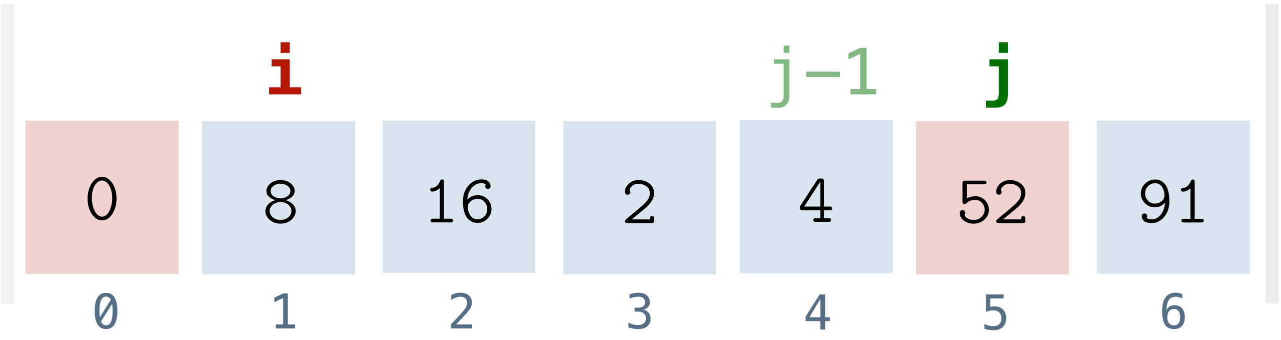
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

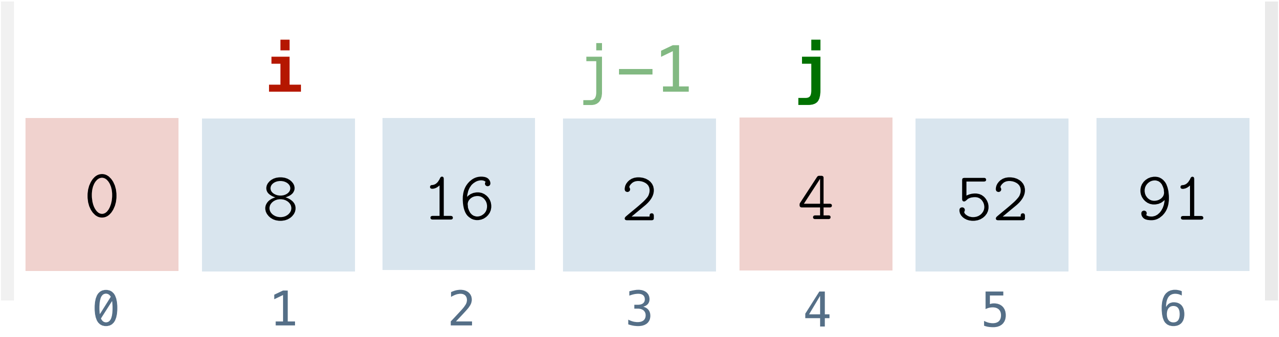
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

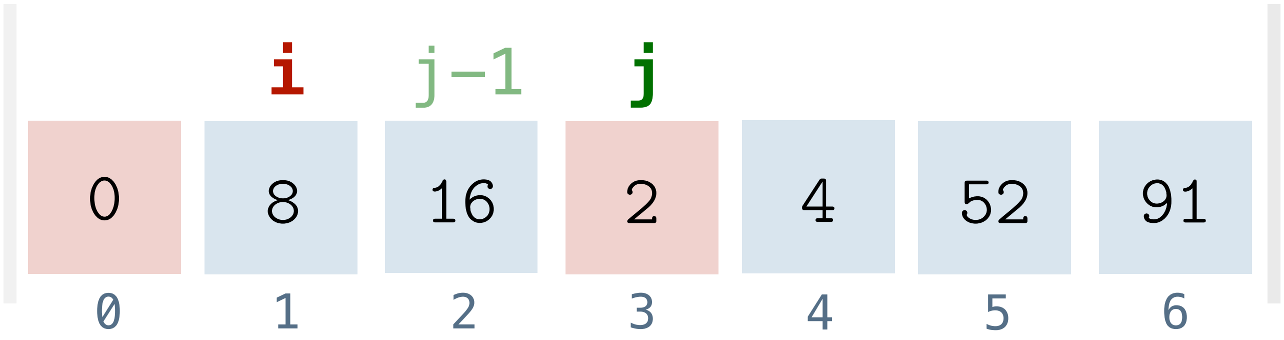


0	1	2	3	4	5	6
0	8	16	2	4	52	91

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

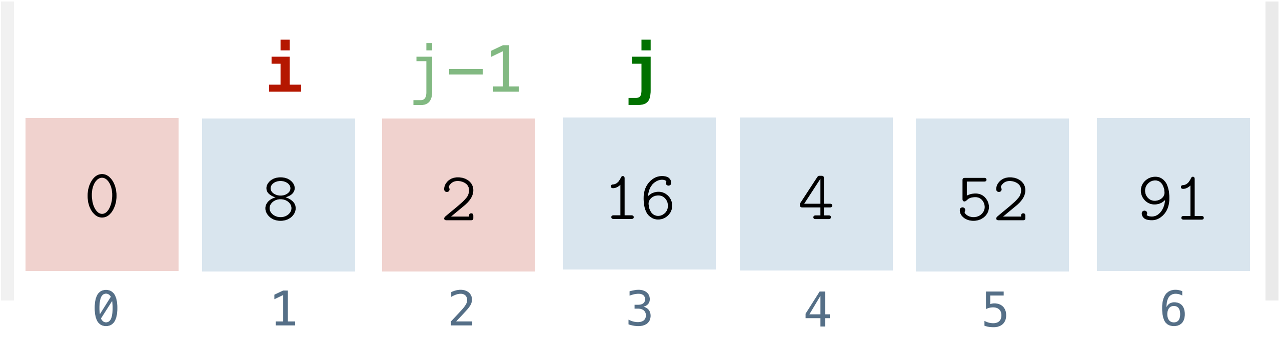
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }
    }
}
```

compare adjacent elements and swap if not in order

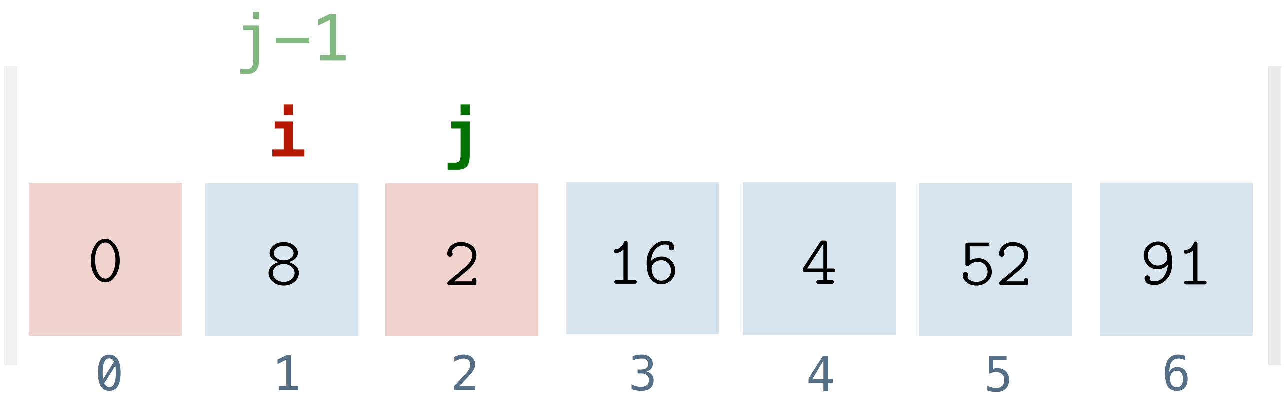
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

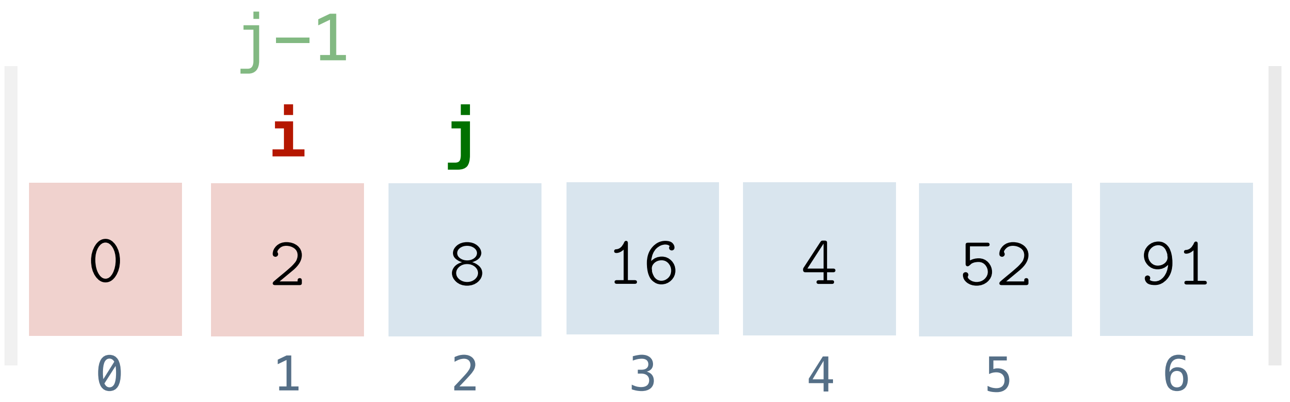
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

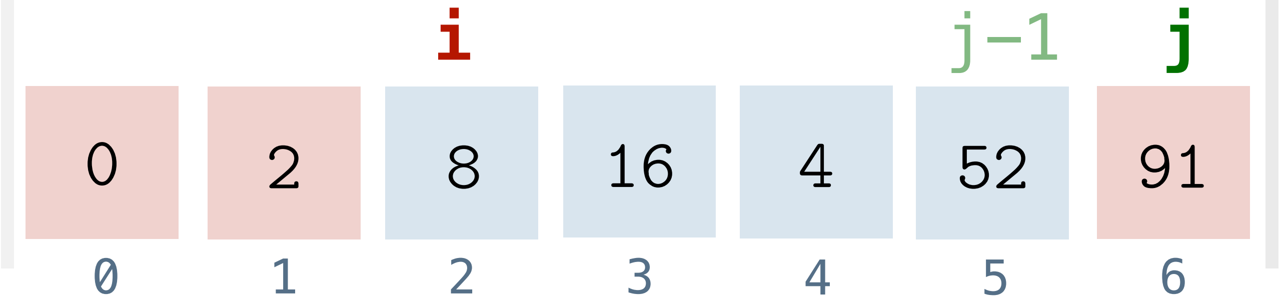
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

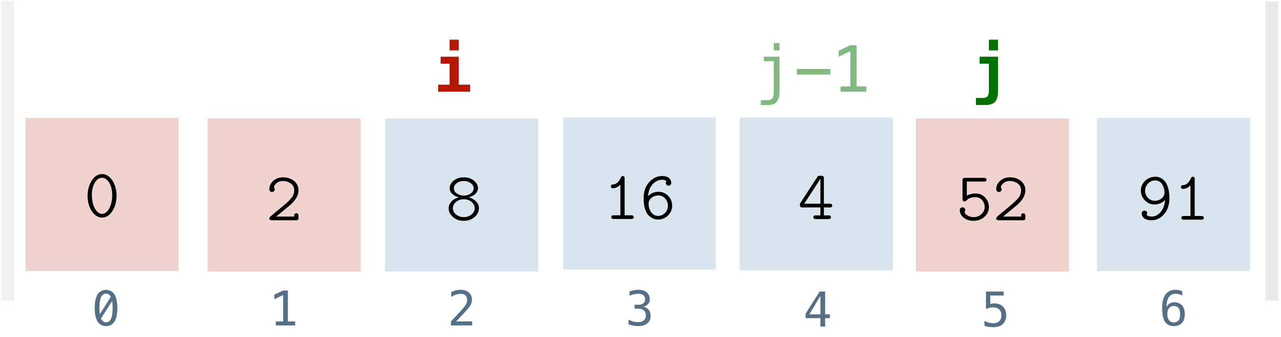
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }



    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

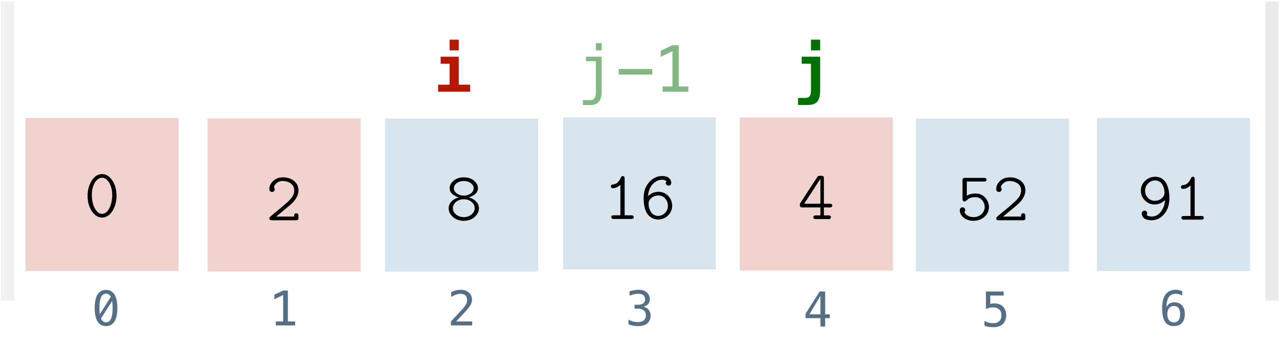


		i		j-1	j	
0	2	8	16	4	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

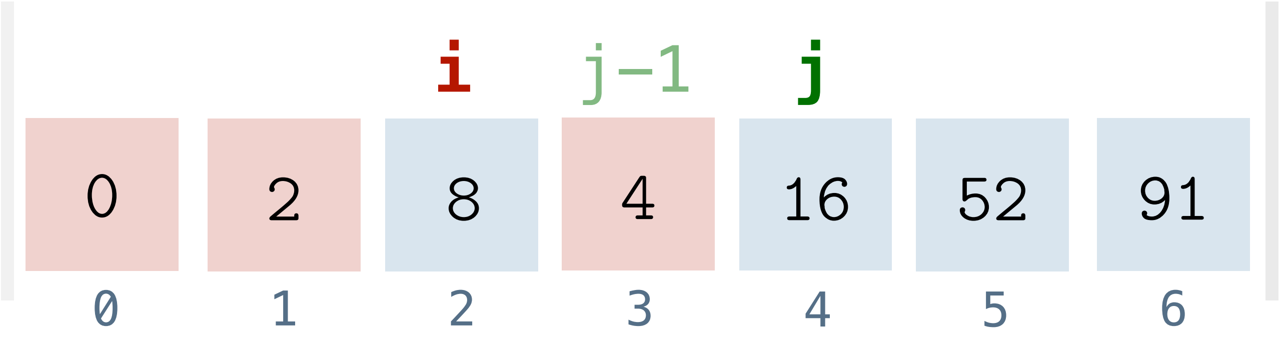
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

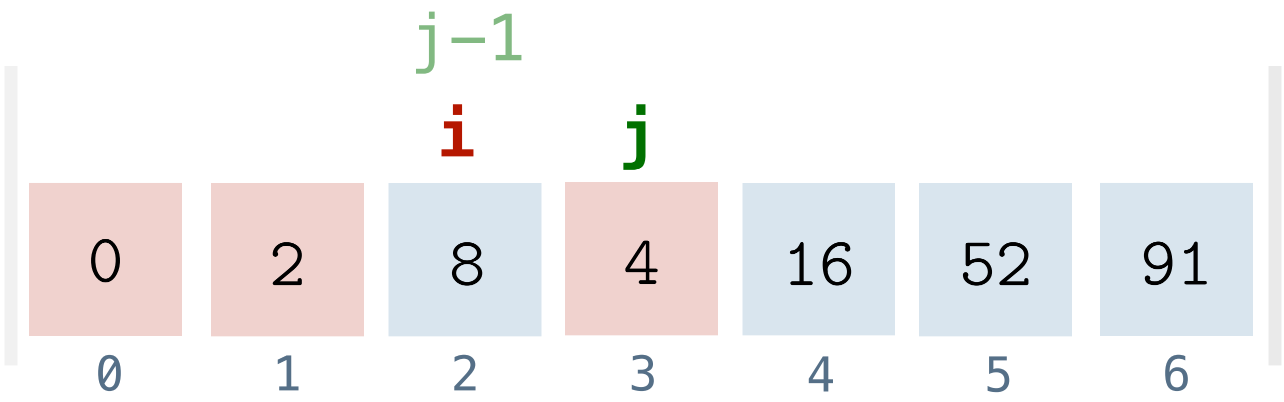


		i	j-1	j		
0	2	8	4	16	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

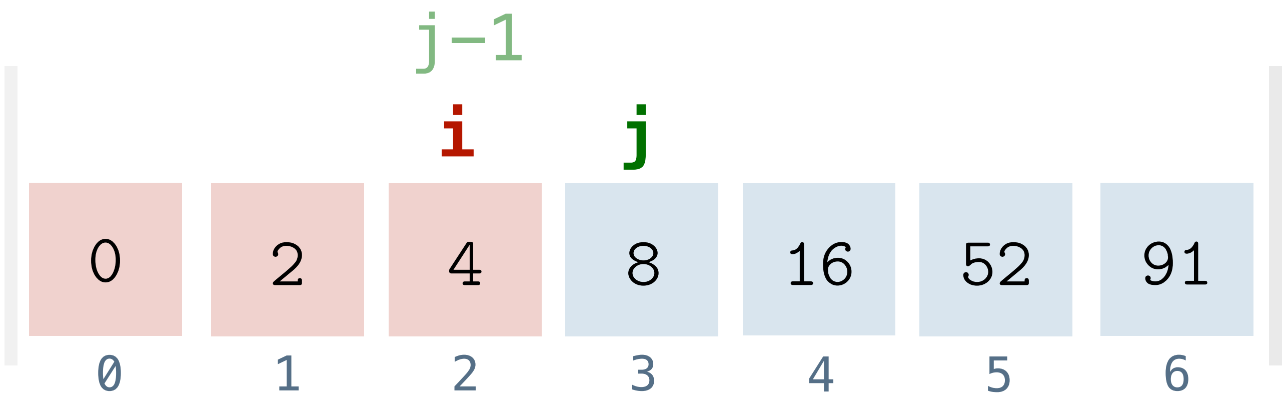
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }

    }
}
```

compare adjacent elements and swap if not in order

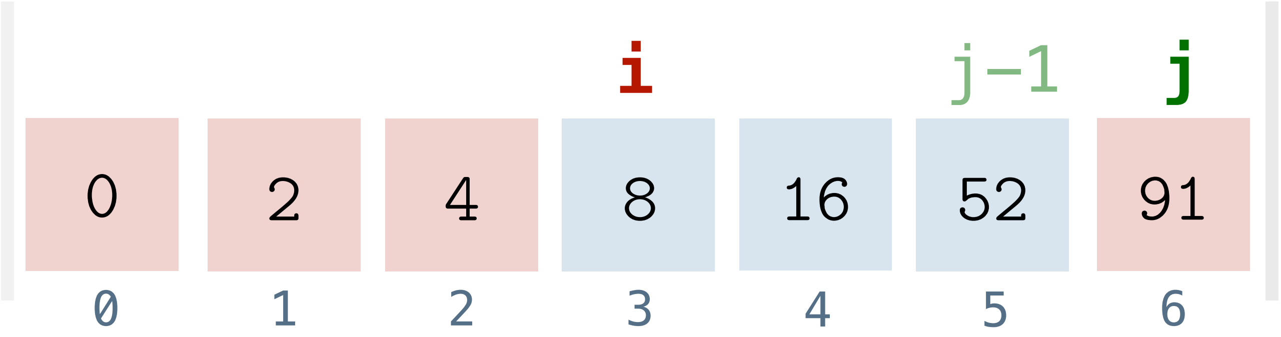
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }
    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

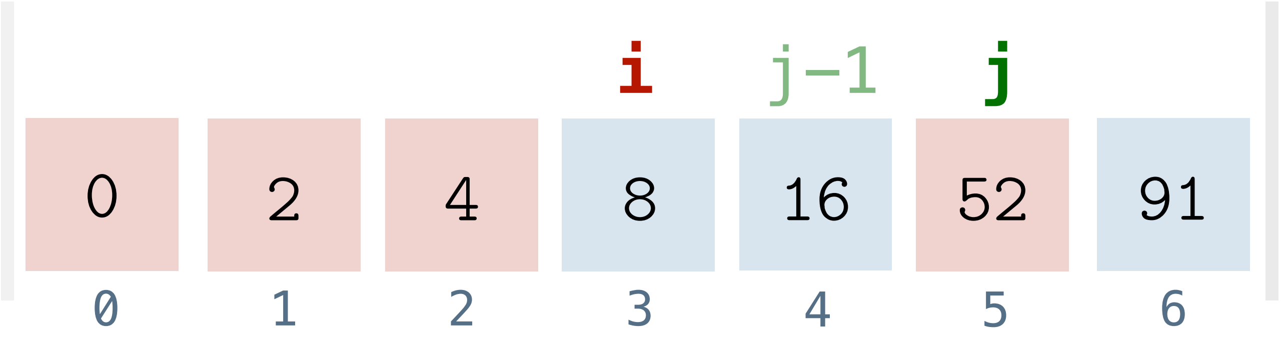


			i		j-1	j
0	2	4	8	16	52	91
0	1	2	3	4	5	6

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }
    }
}
```

compare adjacent elements and swap if not in order

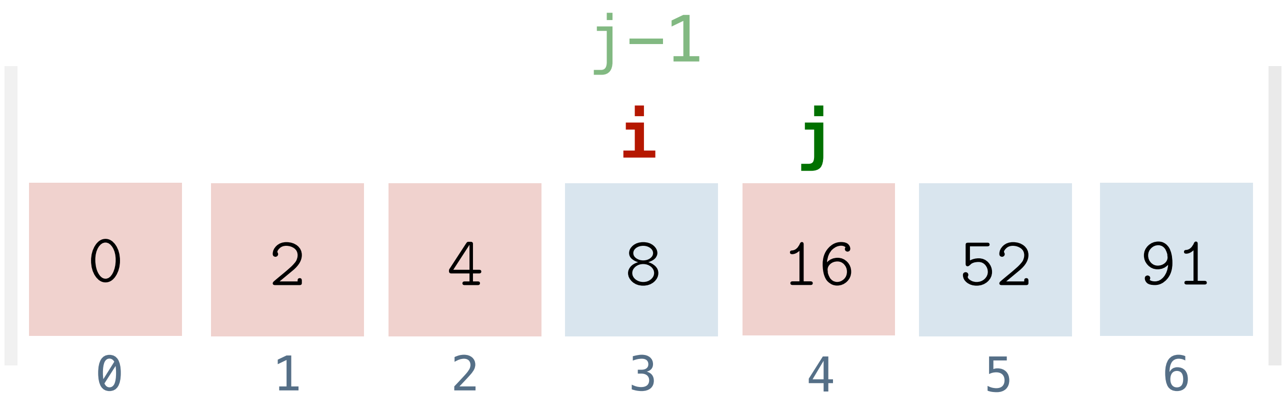
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

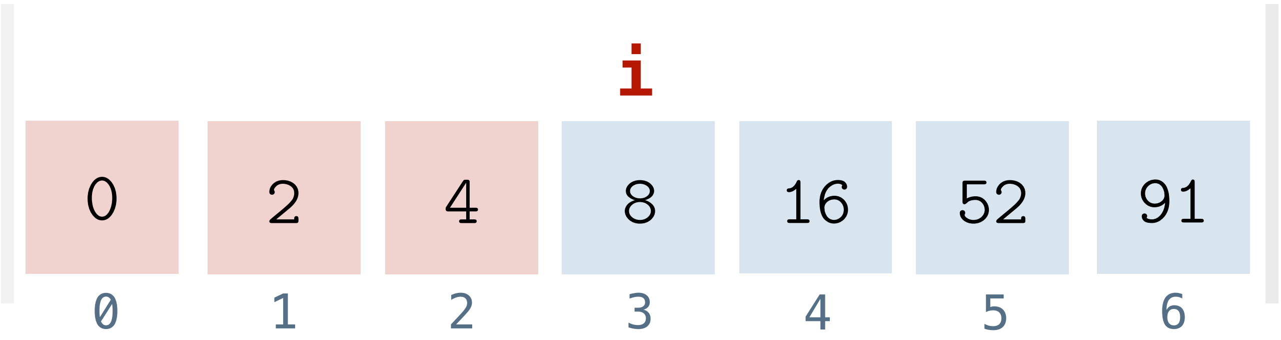
Bubble Sort: Implementation



```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }


    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation

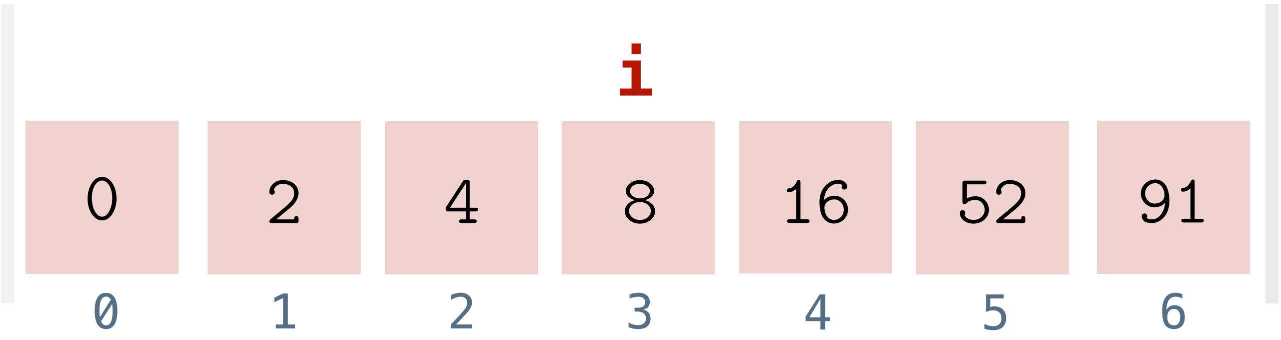


No Swaps!
This means that the remaining elements are already sorted

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {

        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);

            }
        }



    }
}
```

compare adjacent elements and swap if not in order

Bubble Sort: Implementation



No Swaps!
This means that the remaining elements are already sorted

```
void bubble(int a[], int n) {

    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }

        if (!swapped)
            break;
    }
}
```

compare adjacent elements and swap if not in order

Analysis

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Worst Case.

Analysis

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Worst Case. Reversely sorted arrays.

Data compares. $(n-1) + (n-2) + \ldots\ + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$

Data moves.

Analysis

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Worst Case. Reversely sorted arrays.

Data compares. $(n-1) + (n-2) + \ldots + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$

Data moves. Swap with every compare = $3 \times \frac{1}{2}n(n-1)$

Total. $O(n^2)$

Analysis

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Worst Case. Reversely sorted arrays.

Data compares. $(n-1)+(n-2)+\ldots+3+2+1=\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$

Data moves. Swap with every compare = $3\times\frac{1}{2}n(n-1)$

Total. $O(n^2)$

Best Case.

Analysis

```
void bubble(int a[], int n) {
    for (int i = 0;  i < n-1;  i++) {
        bool swapped = false;
        for (int j = n-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                swap(a[j], a[j-1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Worst Case. Reversely sorted arrays.

Data compares. $(n-1)+(n-2)+\ldots+3+2+1=\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$

Data moves. Swap with every compare = $3\times\frac{1}{2}n(n-1)$

Total. $O(n^2)$

Best Case. Sorted arrays.

Only one iteration of the outer loop (0 swaps and $n-1$ data compares) = $O(n)$

	Best		Worst		Random Data		Partially Sorted	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble								
Insertion								
Selection								

	Best		**Worst**		**Random Data**		**Partially Sorted**	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble	$O(n)$	$O(1)$	$O(n^2)$	$O(n^2)$	$\frac{1}{2}n(n-1)$	$\frac{3}{4}n(n-1)$	No general answer. It depends on when the *swapped* flag remains `false`	
	$O(n)$ Sorted Arrays assuming the *swapped* flag is used		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$			
Insertion	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$\frac{1}{4}n(n-1)$	$\frac{1}{4}n(n-1)$ shifts	$O(n)$	$O(n)$
	$O(n)$ Sorted Arrays		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$		$O(n)$	
Selection	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	$\frac{1}{2}n(n-1)$	$O(n)$	$O(n^2)$	$O(n)$
	$O(n^2)$ Sorted Arrays		$O(n^2)$		$O(n^2)$		$O(n^2)$	

	Best		Worst		Random Data		Partially Sorted	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble	$O(n)$	$O(1)$	$O(n^2)$	$O(n^2)$	$\frac{1}{2}n(n-1)$	$\frac{3}{4}n(n-1)$	No general answer. It depends on when the *swapped* flag remains `false`	
	$O(n)$ Sorted Arrays assuming the *swapped* flag is used		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$			
Insertion	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$\frac{1}{4}n(n-1)$	$\frac{1}{4}n(n-1)$ shifts	$O(n)$	$O(n)$
	$O(n)$ Sorted Arrays		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$		$O(n)$	
Selection	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	$\frac{1}{2}n(n-1)$	$O(n)$	$O(n^2)$	$O(n)$
	$O(n^2)$ Sorted Arrays		$O(n^2)$		$O(n^2)$		$O(n^2)$	

The overall running time for all of these algorithms is *asymptotically the same in the worst case*

	Best		Worst		Random Data		Partially Sorted	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble	$O(n)$	$O(1)$	$O(n^2)$	$O(n^2)$	$\frac{1}{2}n(n-1)$	$\frac{3}{4}n(n-1)$	No general answer. It depends on when the *swapped* flag remains false	
	$O(n)$ Sorted Arrays assuming the *swapped* flag is used		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$			
Insertion	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$\frac{1}{4}n(n-1)$	$\frac{1}{4}n(n-1)$ shifts	$O(n)$	$O(n)$
	$O(n)$ Sorted Arrays		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$		$O(n)$	
Selection	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	$\frac{1}{2}n(n-1)$	$O(n)$	$O(n^2)$	$O(n)$
	$O(n^2)$ Sorted Arrays		$O(n^2)$		$O(n^2)$		$O(n^2)$	

Insertion Sort is expected to be a bit more efficient *on random data*

	Best		Worst		Random Data		Partially Sorted	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble	$O(n)$	$O(1)$	$O(n^2)$	$O(n^2)$	$\frac{1}{2}n(n-1)$	$\frac{3}{4}n(n-1)$	No general answer. It depends on when the *swapped* flag remains false	
	$O(n)$ Sorted Arrays assuming the *swapped* flag is used		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$			
Insertion	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$\frac{1}{4}n(n-1)$	$\frac{1}{4}n(n-1)$ shifts	$O(n)$	$O(n)$
	$O(n)$ Sorted Arrays		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$		$O(n)$	
Selection	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	$\frac{1}{2}n(n-1)$	$O(n)$	$O(n^2)$	$O(n)$
	$O(n^2)$ Sorted Arrays		$O(n^2)$		$O(n^2)$		$O(n^2)$	

Selection Sort is the only algorithm that does a *linear number of data moves* in the worst case.

	Best		Worst		Random Data		**Partially Sorted**	
	DC	DM	DC	DM	DC	DM	DC	DM
Bubble	$O(n)$	$O(1)$	$O(n^2)$	$O(n^2)$	$\frac{1}{2}n(n-1)$	$\frac{3}{4}n(n-1)$	No general answer. It depends on when the *swapped* flag remains `false`	
	$O(n)$ Sorted Arrays assuming the *swapped* flag is used		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$			
Insertion	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$\frac{1}{4}n(n-1)$	$\frac{1}{4}n(n-1)$ shifts	$O(n)$	$O(n)$
	$O(n)$ Sorted Arrays		$O(n^2)$ Reversely Sorted Arrays		$O(n^2)$		$O(n)$	
Selection	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	$\frac{1}{2}n(n-1)$	$O(n)$	$O(n^2)$	$O(n)$
	$O(n^2)$ Sorted Arrays		$O(n^2)$		$O(n^2)$		$O(n^2)$	

Insertion Sort is the winner on *partially sorted data*

💪 Advanced Exercises

Exercise # 1

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many data compares does selection sort perform if run on such an array of size $2m$?

Exercise # 1

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{\texttt{1 2 3 4 5 6 7 8}}_{m}\ \underbrace{\texttt{8 7 6 5 4 3 2 1}}_{m}$$

How many data compares does selection sort perform if run on such an array of size $2m$?

Answer. Selection sort always does $\frac{1}{2}n(n-1)$ data compares if the array is of size n, regardless of how the elements are ordered in the array.

The size of the array is $2m$. Therefore, selection sort performs $\frac{1}{2}2m(2m-1)$ $= m(2m-1)$ $= 2m^2 - m$ data compares.

Exercise # 2

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad\underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many swaps does bubble sort perform if run on such an array of size $2m$?

Exercise # 2

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many swaps does bubble sort perform if run on such an array of size $2m$?

Answer.

The 1st pass swaps the right-most 1 with $2m - 2$ elements.
The 2nd pass swaps the right-most 2 with $2m - 4$ elements.
The 3rd pass swaps the right-most 3 with $2m - 6$ elements.

Exercise # 2

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad\underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many swaps does bubble sort perform if run on such an array of size $2m$?

Answer.

The 1st pass swaps the right-most 1 with $2m - 2$ elements.
The 2nd pass swaps the right-most 2 with $2m - 4$ elements.
The 3rd pass swaps the right-most 3 with $2m - 6$ elements.
…
The right-most 6 is swapped with 4 elements.
The right-most 7 is swapped with 2 elements.
The right-most 8 is swapped with 0 elements. All the remaining elements will not need extra swaps for them to get to their positions
(swaps from the previous passes of the algorithm get them to their positions).

Exercise # 2

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8}_{m} \quad \underbrace{8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1}_{m}$$

How many swaps does bubble sort perform if run on such an array of size $2m$?

Answer.

The 1st pass swaps the right-most 1 with $2m - 2$ elements.
The 2nd pass swaps the right-most 2 with $2m - 4$ elements.
The 3rd pass swaps the right-most 3 with $2m - 6$ elements.
…
The right-most 6 is swapped with 4 elements.
The right-most 7 is swapped with 2 elements.
The right-most 8 is swapped with 0 elements. All the remaining elements will not need extra swaps for them to get to their positions
(swaps from the previous passes of the algorithm get them to their positions).

The total is: $0 + 2 + 4 + 6 + \ldots + (2m - 6) + (2m - 4) + (2m - 2)$

Exercise # 2

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad\underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many swaps does bubble sort perform if run on such an array of size $2m$?

Answer.
The 1st pass swaps the right-most 1 with $2m-2$ elements.
The 2nd pass swaps the right-most 2 with $2m-4$ elements.
The 3rd pass swaps the right-most 3 with $2m-6$ elements.
…
The right-most 6 is swapped with 4 elements.
The right-most 7 is swapped with 2 elements.
The right-most 8 is swapped with 0 elements. All the remaining elements will not need extra swaps for them to get to their positions
(swaps from the previous passes of the algorithm get them to their positions).

The total is:

$$\begin{aligned}&0+2+4+6+\ldots+(2m-6)+(2m-4)+(2m-2)\\ &=2(0+1+2+3+\ldots+(m-3)+(m-2)+(m-1))\\ &=2(\tfrac{1}{2}m(m-1)) \quad = m(m-1) = m^2-m \ \text{ swaps}\end{aligned}$$

Exercise # 3

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many data compares does insertion sort perform if run on such an array of size $2m$?

Exercise # 3

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many data compares does insertion sort perform if run on such an array of size $2m$?

Answer.

First half: `m-1` compares. Each element is compared to the one to its left.

Exercise # 3

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{\texttt{1 2 3 4 5 6 7 8}}_{m}\ \underbrace{\texttt{8 7 6 5 4 3 2 1}}_{m}$$

How many data compares does insertion sort perform if run on such an array of size $2m$?

Answer.

First half: `m-1` compares. Each element is compared to the one to its left.

Second half: The `8` is compared to the `8` to its left (1 compare).
The `7` is compared to the `7, 8, 8` to its left (3 compares).
The `6` is compared to the `6, 7, 7, 8, 8` to its left (5 compares).
…
Finally, the `1` is compared to all the remaining `2m-1` elements.

Exercise # 3

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many data compares does insertion sort perform if run on such an array of size $2m$?

Answer.

First half: `m-1` compares. Each element is compared to the one to its left.

Second half: The `8` is compared to the `8` to its left (1 compare).
The `7` is compared to the `7, 8, 8` to its left (3 compares).
The `6` is compared to the `6, 7, 7, 8, 8` to its left (5 compares).
…
Finally, the `1` is compared to all the remaining `2m-1` elements.
The total is:

$$\begin{aligned}
& 1 + 3 + 5 + \ldots + 2m-1 \\
= {} & (0+1) + (2+1) + (4+1) + \ldots + 2m-2+1 \\
= {} & m + 0 + 2 + 4 + \ldots + 2m-2 \\
= {} & m + 2(0 + 1 + 2 + \ldots + m-1) \\
= {} & m + m(m-1) = m^2
\end{aligned}$$

Exercise # 3

Q. Consider an organ-pipe array made of two equal halves of size m each, where elements increase then decrease:

$$\underbrace{1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8}_{m}\quad \underbrace{8\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1}_{m}$$

How many data compares does insertion sort perform if run on such an array of size $2m$?

Answer.

First half: m-1 compares. Each element is compared to the one to its left.

Second half: The 8 is compared to the 8 to its left (1 compare).
The 7 is compared to the 7, 8, 8 to its left (3 compares).
The 6 is compared to the 6, 7, 7, 8, 8 to its left (5 compares).
...
Finally, the 1 is compared to all the remaining 2m-1 elements.

The total is:

$$\begin{aligned}
& 1 + 3 + 5 + \dots + 2m-1 \\
={} & (0+1) + (2+1) + (4+1) + \dots + 2m-2+1 \\
={} & m + 0 + 2 + 4 + \dots + 2m-2 \\
={} & m + 2(0 + 1 + 2 + \dots + m-1) \\
={} & m + m(m-1) = m^2
\end{aligned}$$

Adding the compares from the first half, we get a total of $m^2 + m - 1$ compares.

Exercise # 4

Q. Assume that selection sort knows how to find the minimum in a range of size m in $\log_2 m$ comparisons only. What would be the order of growth of the running time of selection sort if run on an array of size n ?

A. $O(n^2 \log n)$

B. $O(n \log n)$

C. $O(n \log m)$

D. It is impossible to find the minimum in logarithmic time.

```
selection-sort(a[], n):
  for every i from 0 to n-1:
      find the minimum from i to n-1
        place the minimum at index i
```

Exercise # 4

Q. Assume that selection sort knows how to find the minimum in a range of size m in $\log_2 m$ comparisons only. What would be the order of growth of the running time of selection sort if run on an array of size n ?

A. $O(n^2 \log n)$

[✓ **B.** $O(n \log n)$]

C. $O(n \log m)$

D. It is impossible to find the minimum in logarithmic time.

```
selection-sort(a[], n):
  for every i from 0 to n-1:
      find the minimum from i to n-1
        place the minimum at index i
```

$$\text{Total} = \log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ldots + \log_2(3) + \log_2(2) + \log_2(1)$$

$$\leq \log_2(n!) = O(n \log n)$$

Exercise # 5

Q. Assume that insertion sort uses binary search to find the insertion position in the sorted portion of the array. Does this affect the worst case running time of the algorithm?

```
insertion-sort(a[], n):
  for every i from 1 to n-1:
    insert a[i] in the range 0 to i-1
    using linear search and shifts
```

```
binary-insertion-sort(a[], n):
  for every i from 1 to n-1:
    pos = binary_search(a, a[i], 0, i-1)
    insert(a, a[i], pos, i-1)
```

A. No.

B. Affects the actual running time but not the asymptotic running time.

C. Affects both the actual and asymptotic running times.

Exercise # 5

Q. Assume that insertion sort uses binary search to find the insertion position in the sorted portion of the array. Does this affect the worst case running time of the algorithm?

```
insertion-sort(a[], n):
  for every i from 1 to n-1:
    insert a[i] in the range 0 to i-1
    using linear search and shifts
```

```
binary-insertion-sort(a[], n):
  for every i from 1 to n-1:
    pos = binary_search(a, a[i], 0, i-1)
    insert(a, a[i], pos, i-1)
```

A. No.

[✓ **B.** Affects the actual running time but not the asymptotic running time.]

C. Affects both the actual and asymptotic running times.

Number of data compares becomes: $O(\lg(1) + \lg(2) + \lg(3) + \ldots + \lg(n-1)) = O(n \log n)$
Number of data moves remains $O(n^2)$

Total = $O(n \log n) + O(n^2) = O(n^2)$ instead of $O(n^2) + O(n^2) = O(n^2)$