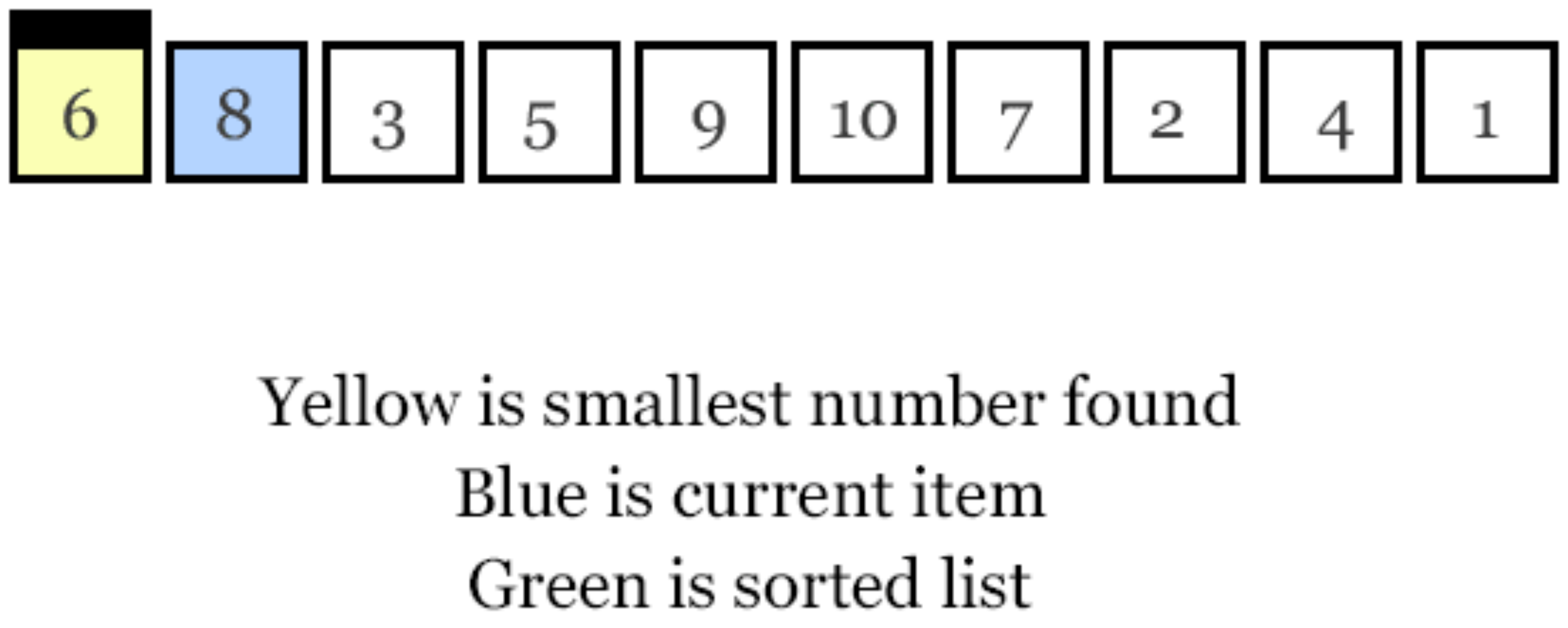
CS62 Class 12: Selection & Insertion sort

Sorting



Selection sort

Insertion sort

Agenda

- More rules of sorting
- Selection sort
- Insertion sort
  - Note: you saw both of these in 51P! (Review slides are on Canvas)

How many different algorithms for sorting can there be?

- Short answer: a ton!

- Adaptive heapsort
- Bitonic sorter
- Block sort
- Bubble sort
- Bucket sort
- Cascade mergesort
- Cocktail sort
- Comb sort
- Flashsort
- Gnome sort
- **Heapsort**
- **Insertion sort**
- Library sort
- **Mergesort**
- Odd-even sort
- Pancake sort
- **Quicksort**
- Radixsort
- **Selection sort**
- Shell sort
- Spaghetti sort
- Treesort
- …

Two useful abstractions

- We will refer to data only through comparisons and exchanges.
- Comparisons: Is v less than w?

  ```
  v.compareTo(w) < 0;
  ```

- Exchanges: Exchanges will result to swapping an element in an array a at index i with one at index j.

  ```
  E temp = a[i];
  a[i]=a[j];
  a[j]=temp;
  ```

Question: How long does a comparison take in big O notation? How about an exchange?

A: Both are O(1) as written above

Rules of the game - Cost model

- Sorting cost model: we count compares and exchanges. If a sorting algorithm does not use exchanges, we count array accesses.
- There are other types of sorting algorithms where they are not based on comparisons (e.g., radixsort, which processes numbers digit by digit into buckets). We will not see these in CS62, but stay tuned for CS140.

Question: There is no sorting algorithm that runs in less than $\Omega(n)$ time. Why?

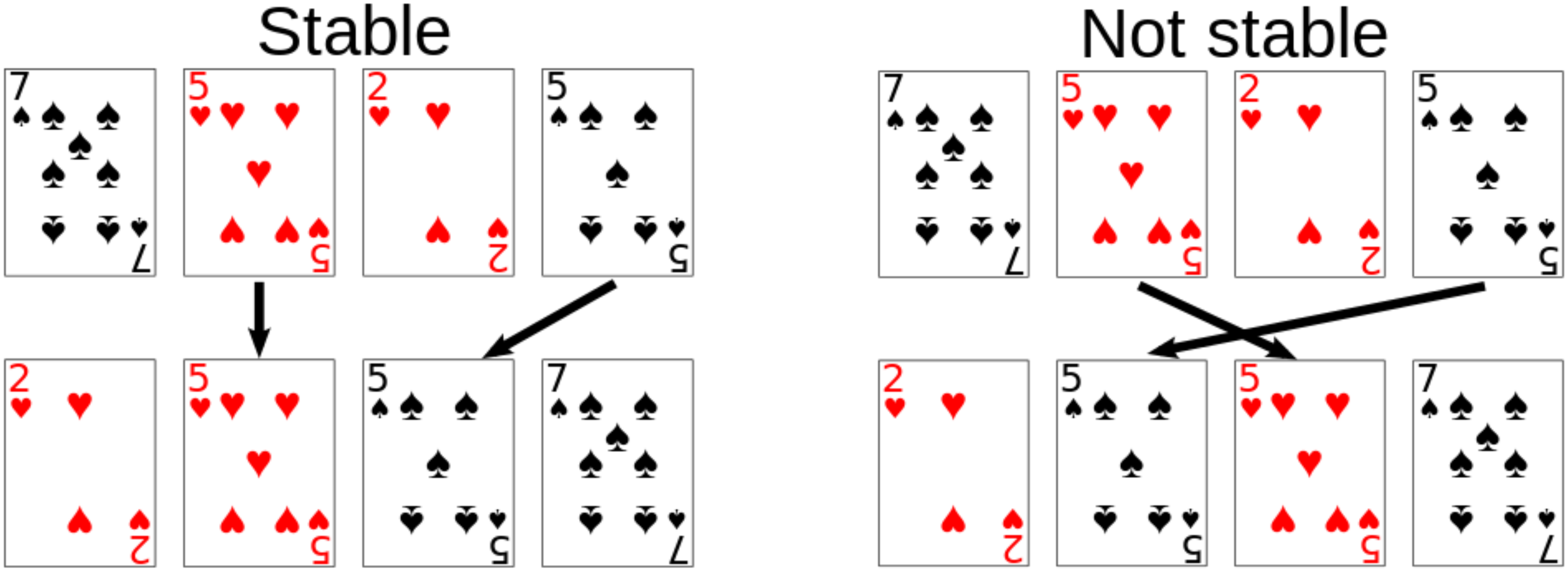
A: We need to see all the data we’re sorting (length n) at least once.

Rules of the game – Memory usage

- Extra memory is often as important as running time. Sorting algorithms are divided into two categories:
  - In place: use constant O(1) or logarithmic O(logn) extra memory, beyond the memory needed to store the elements to be sorted.
  - Not in place: use linear O(n) extra memory.
- Also called space complexity

Rules of the game – Stability

- Stable: sorting algorithms that sort repeated elements in the same order that they appear in the input.



https://en.wikipedia.org/wiki/Sorting_algorithm#/media/File:Sorting_stability_playing_cards.svg

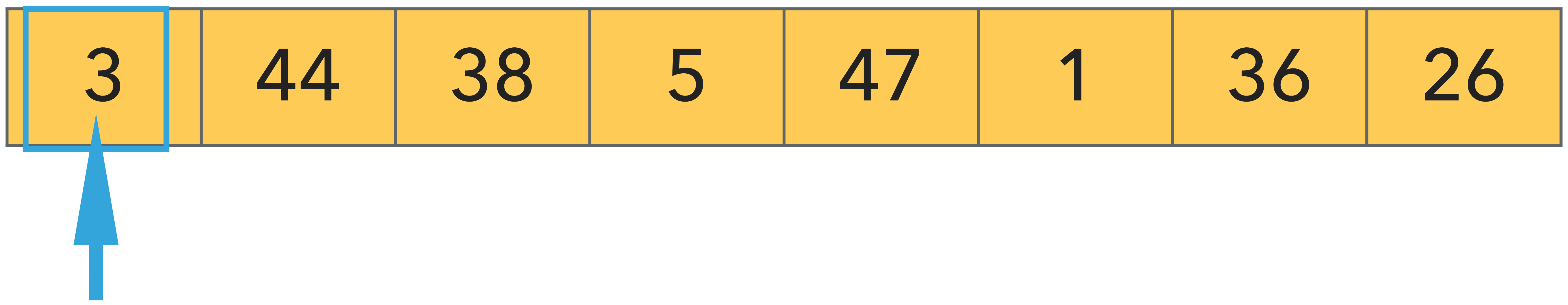
Selection Sort

Selection sort: basic algorithm

3	44	38	5	47	1	36	26

- Divide the array in two parts: a sorted subarray on the left and an unsorted on the right.
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	44	38	5	47	3	36	26

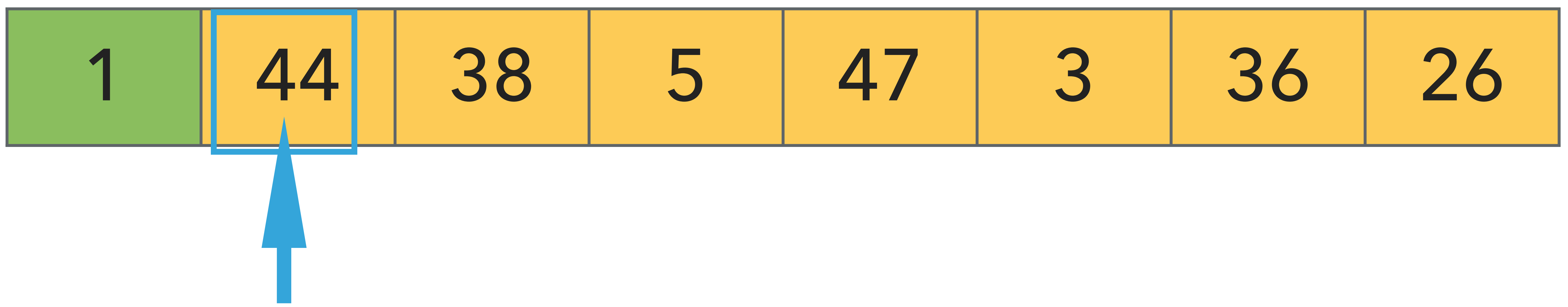
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	44	38	5	47	3	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

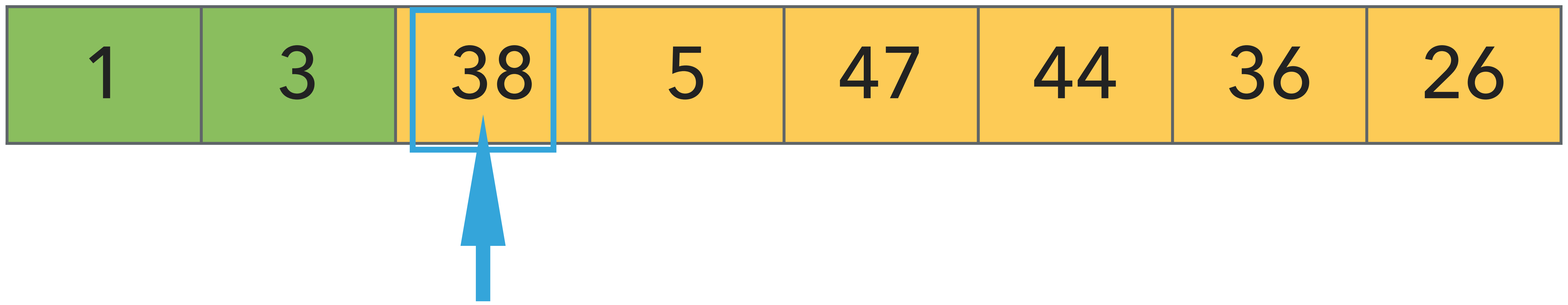
1	3	38	5	47	44	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	38	47	44	36	26

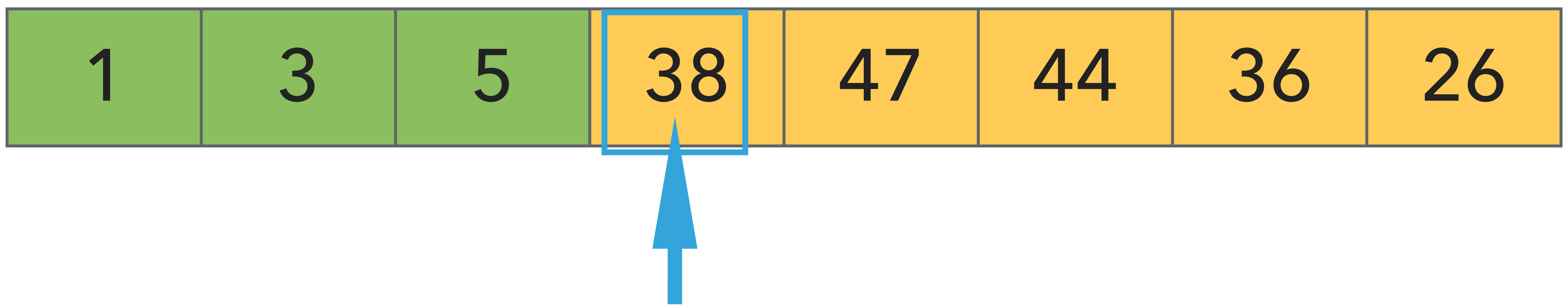
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	38	47	44	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	47	44	36	38

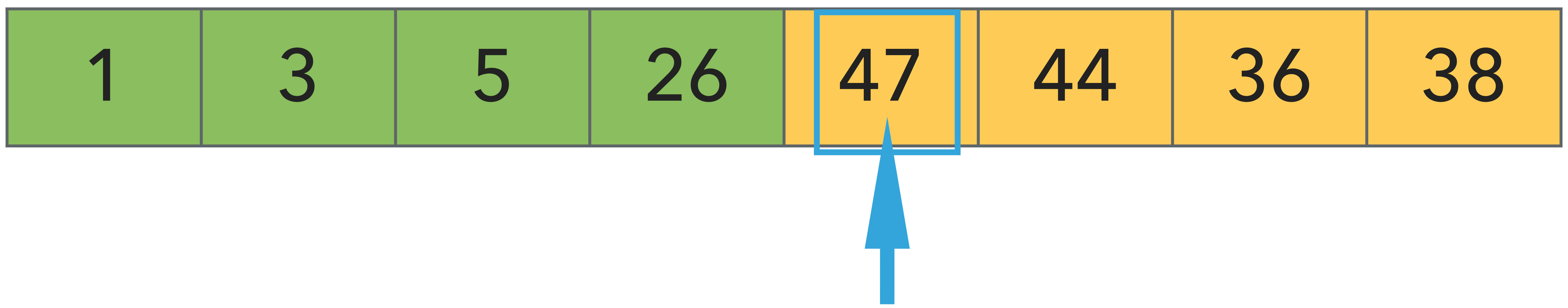
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	47	44	36	38

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	44	47	38

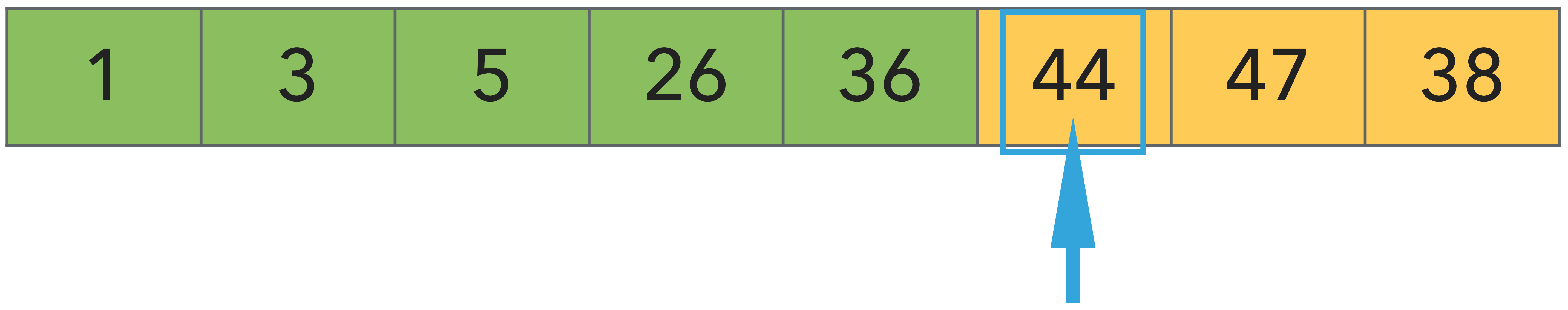
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	44	47	38

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	47	44

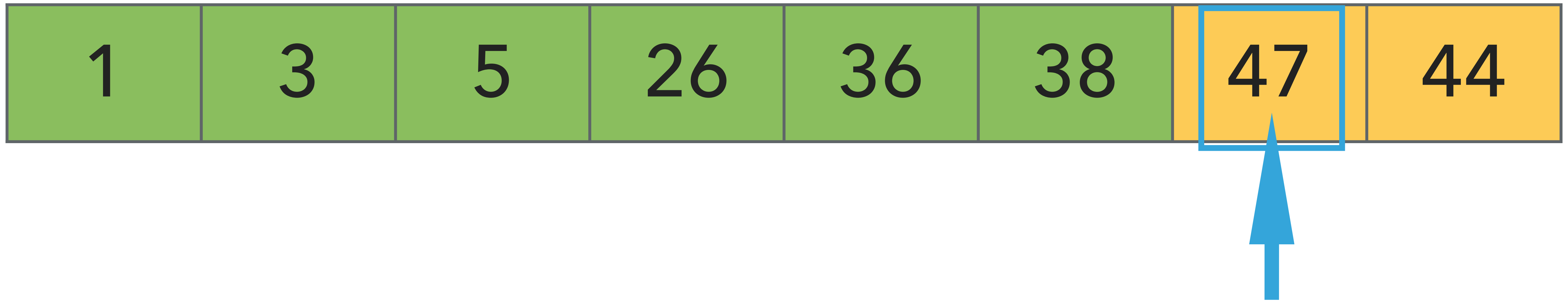
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	47	44

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

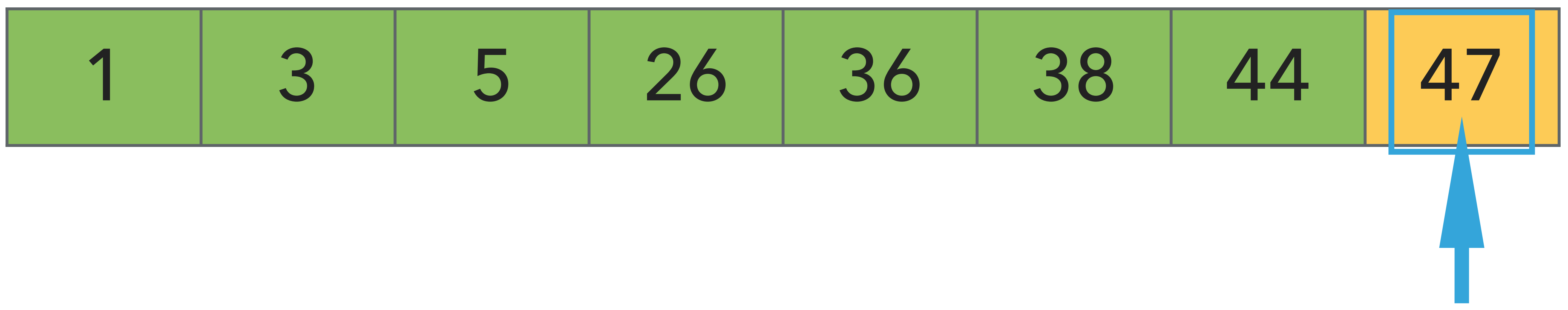
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

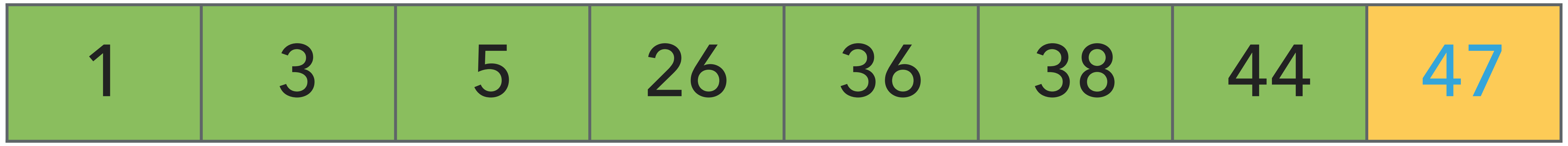
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

http://algs4.cs.princeton.edu

2.1 SELECTION SORT DEMO

https://algs4.cs.princeton.edu/lectures/demo/21DemoSelectionSort.mov

Worksheet time!

- Using selection sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm (draw a new row for each time an element is swapped).

Worksheet answers

https://subscription.packtpub.com/book/application_development/9781785888731/13/ch13lvl1sec89/selection-sort

Worksheet time!

- Fill in the blanks to implement selectionSort:

```
public class SelectionSort {
    public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = _____________;
        for (int i = 0; i < n; i++) {
            int min = ____;
            for (int j = ______ ; ______ ; ____) {
                if (a[j].compareTo(a[min]) __ 0) {
                    min = _____;
                }
            }
            // do the swap
            _______________
            _______________
            _______________
        }
    }
```

Let's talk about this signature: it's a static *generic* method.

<E extends Comparable<E>> - type parameter.
the type of E should implement Comparable
(so .compareTo does not throw a compiler error)

Worksheet answers

```
public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {            ← At iteration i
            int min = i;
            for (int j = i+1; j < n; j++) {      ← Find the index min of the
                if (a[j].compareTo(a[min])<0){    smallest remaining array
                    min = j;
                }
            }
            E temp = a[i];
            a[i]=a[min];                         ← swap a[i] and a[min]
            a[min]=temp;
        }
    }
```

- Invariants: At the end of each iteration i:
  - the array a is sorted in ascending order for the first i+1 elements a[0…i]
  - no entry in a[i+1…n-1] (the remaining array) is smaller than any entry in a[0…i] (the sorted array)

Run time & memory usage & stability

```
public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (a[j].compareTo(a[min])<0){
                    min = j;
                }
            }
            E temp = a[i];
            a[i]=a[min];
            a[min]=temp;
        }
    }
```

Think of it "programmatically": 2 for loops that go to n and increase by a constant #

Think of it "mathematically": each number is how many times the inner loop runs, and each + is for adding them up because of the outer for loop

- Comparisons (calls to compareTo): $1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2$, that is $O(n^2)$.
- Exchanges: n or $O(n)$, making it useful when exchanges are expensive.
- Running time is quadratic, even if input is sorted.
- In-place, requires almost no additional memory.
- Not stable, think of the array [5_a, 3, 5_b, 1] which will end up as [1, 3, 5_b, 5_a].

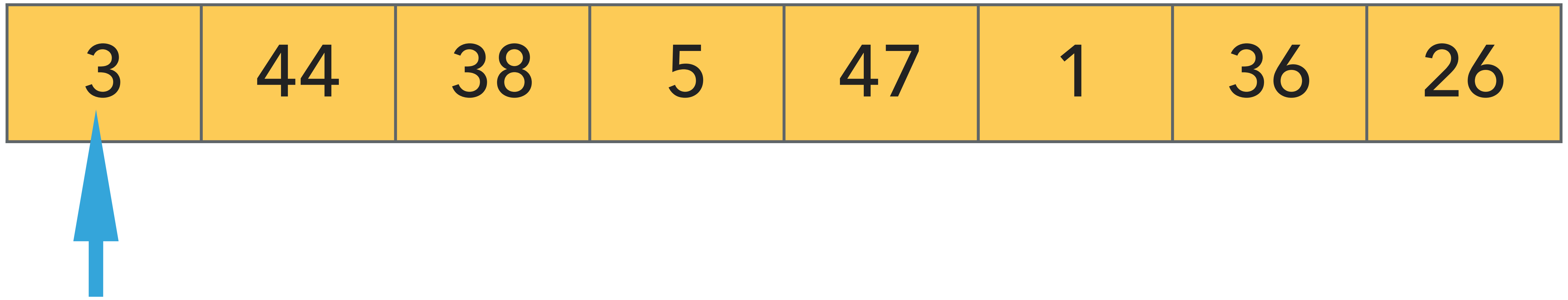
Insertion Sort

Insertion sort

3	44	38	5	47	1	36	26

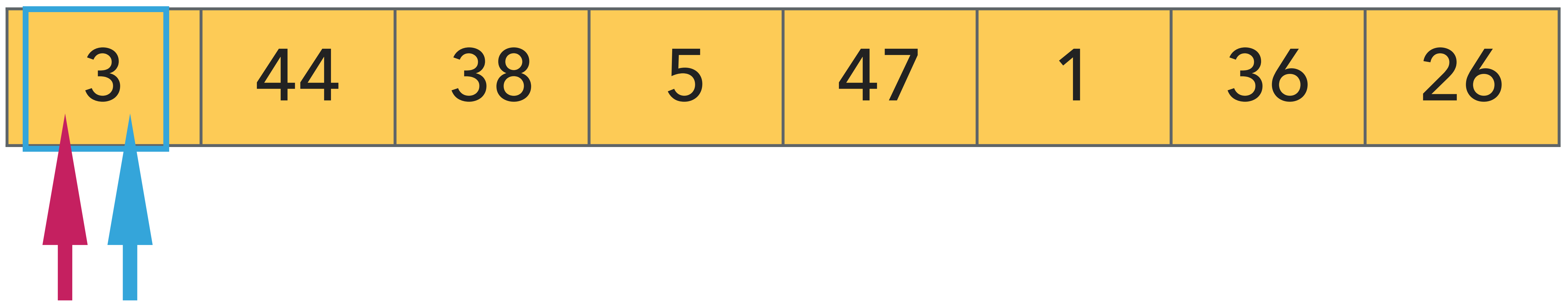
- Keep a *partially* sorted subarray on the left and an unsorted subarray on the right.
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



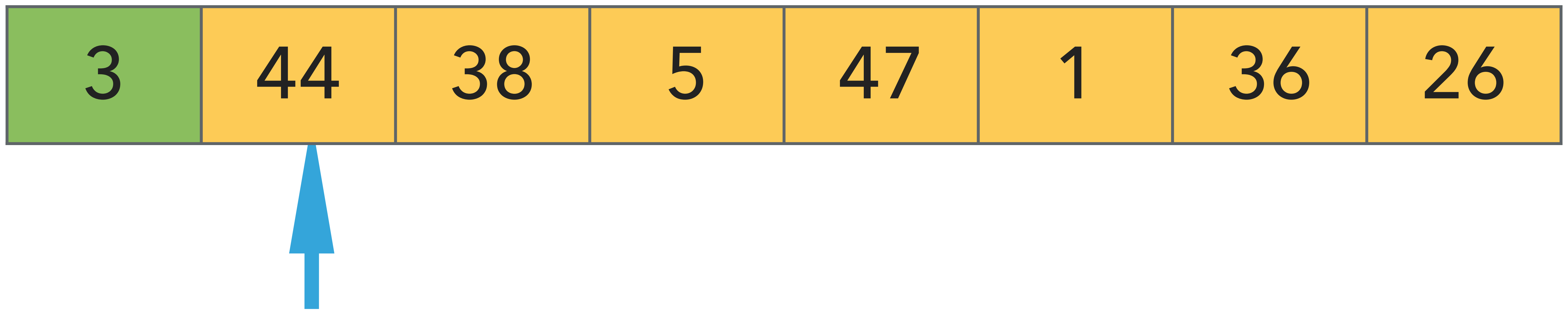
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	44	38	5	47	1	36	26

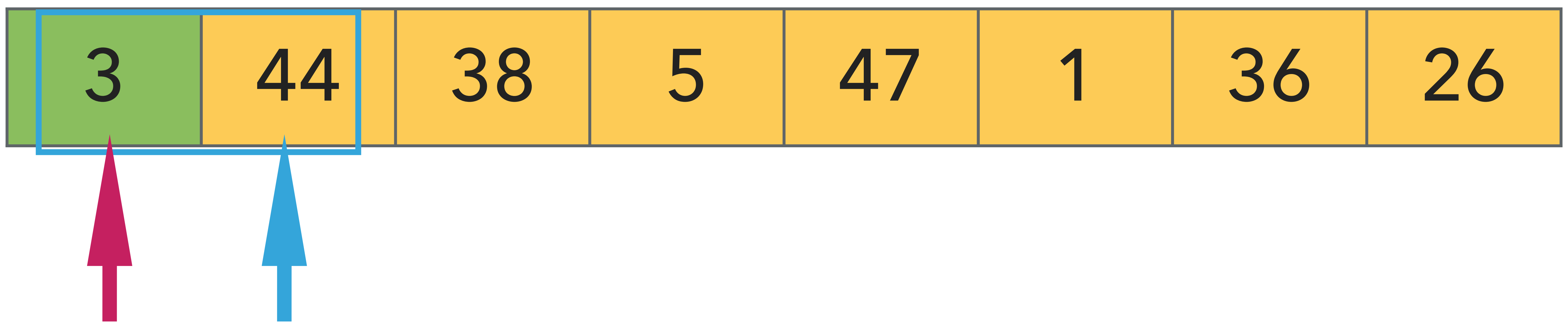
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

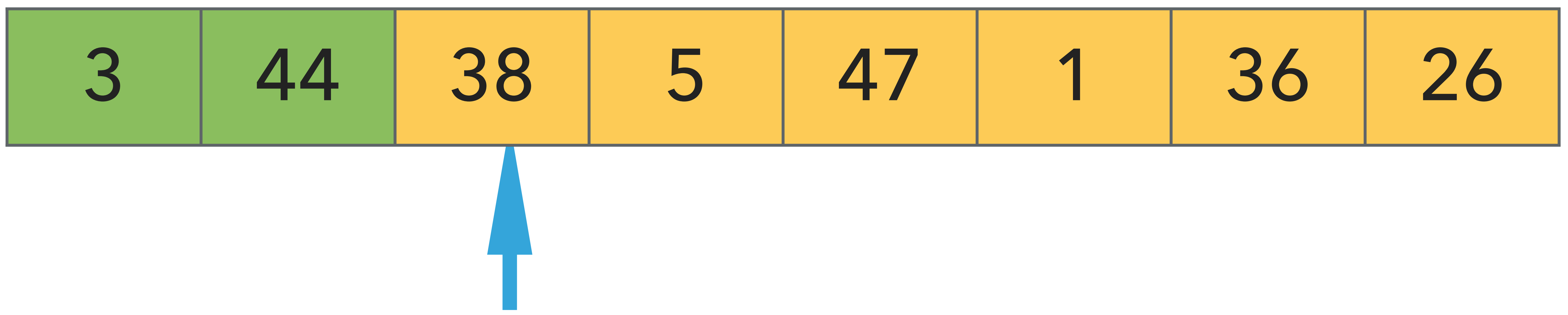


- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

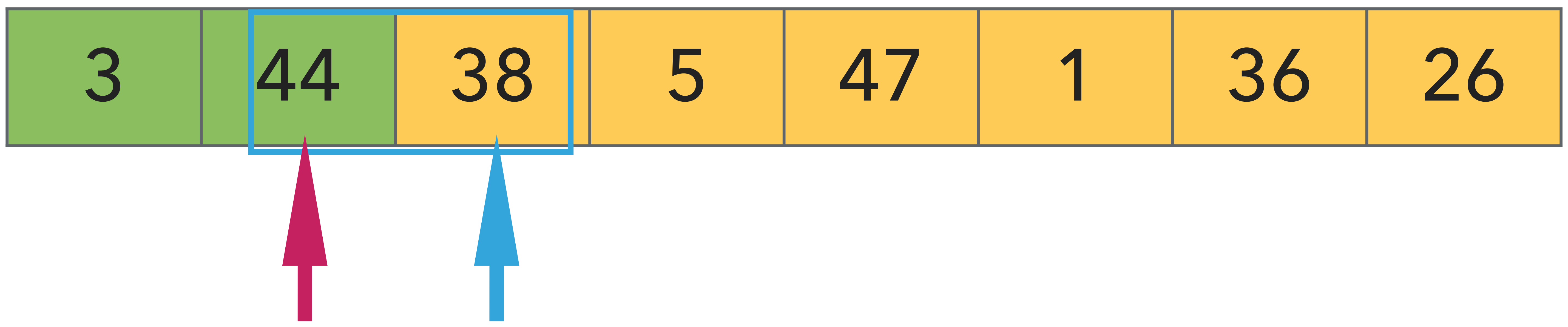
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



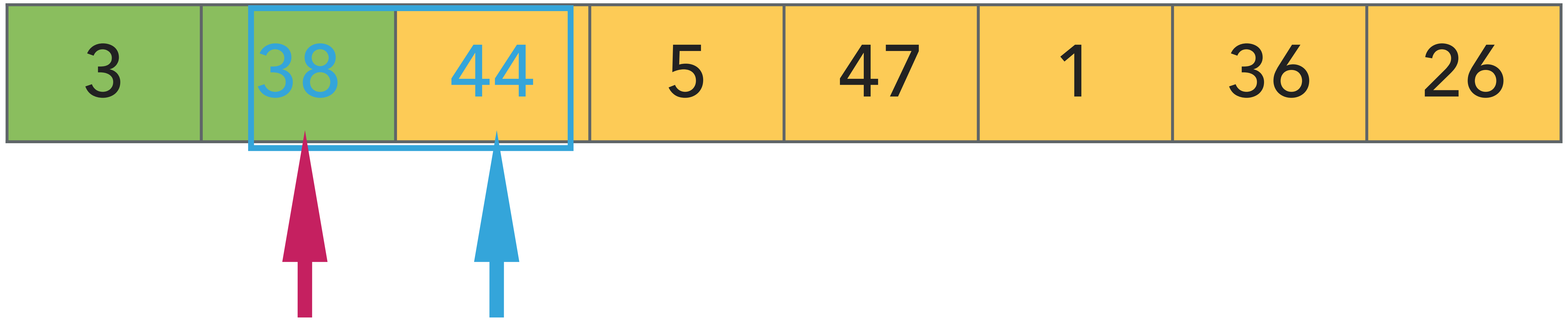
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



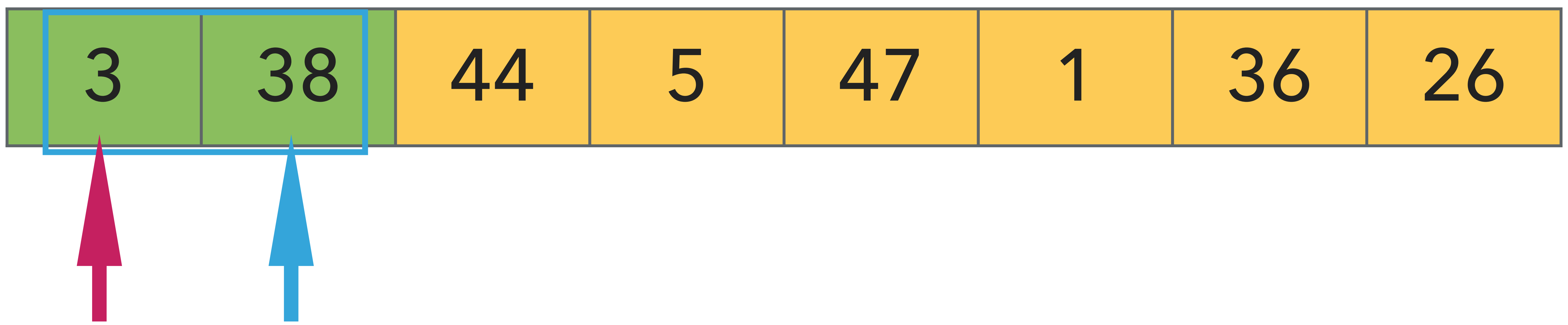
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



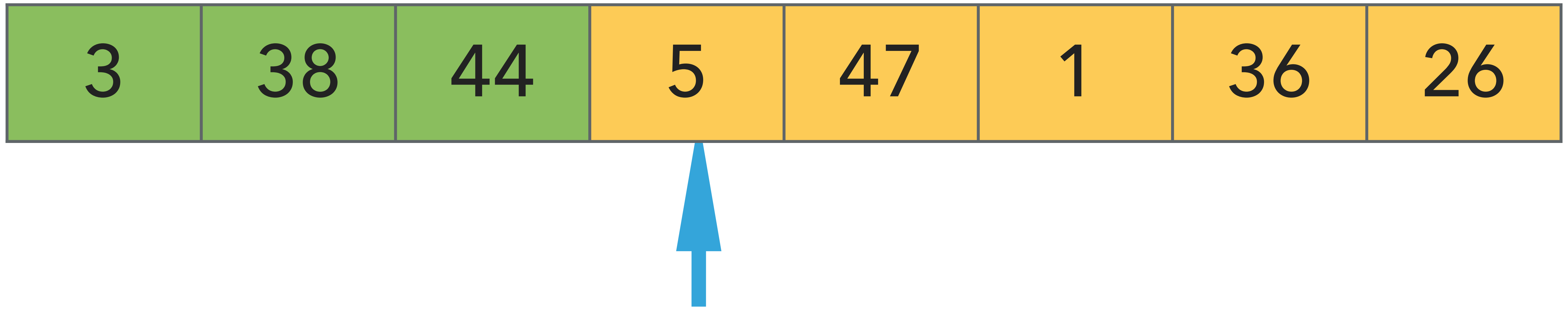
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	38	44	5	47	1	36	26

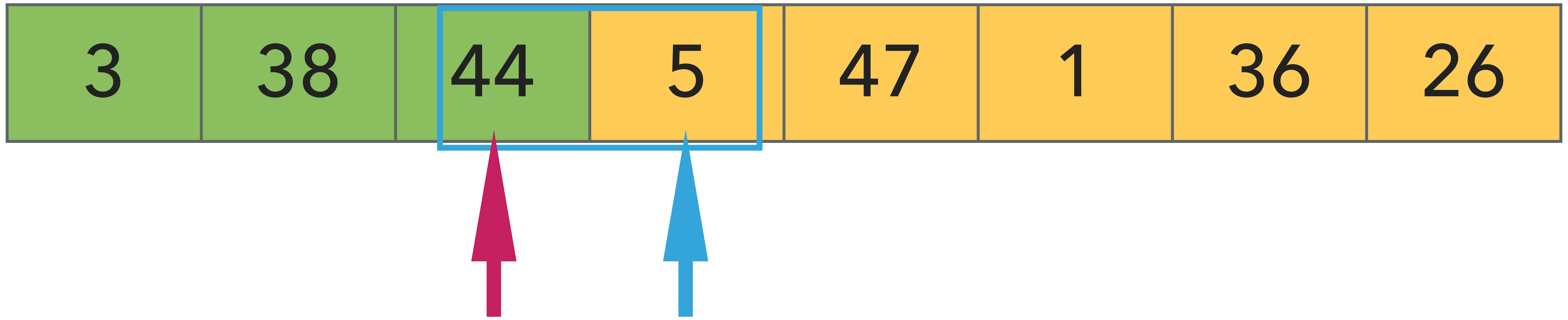
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



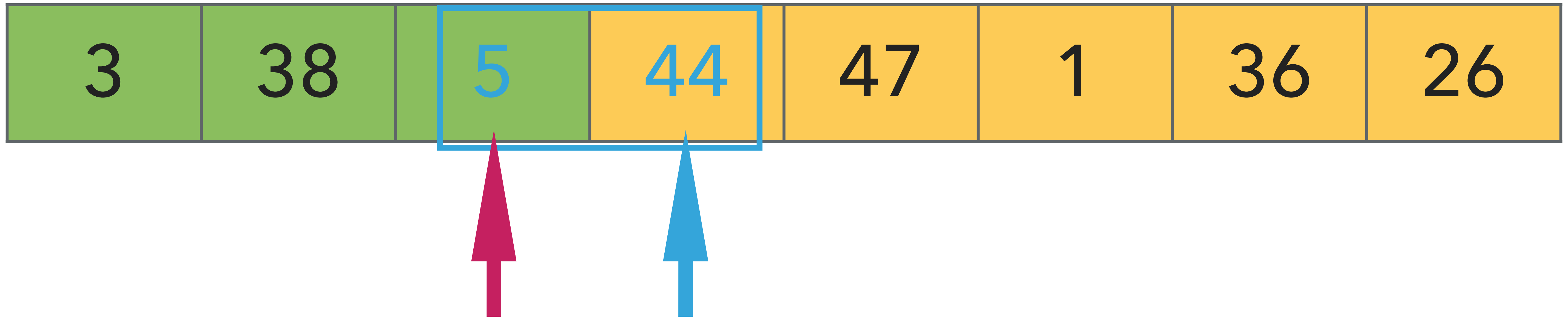
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



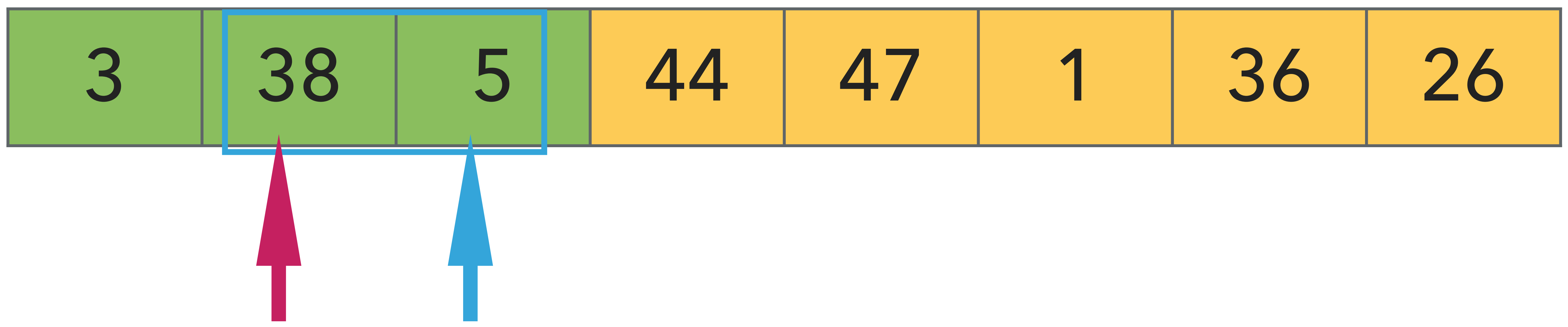
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



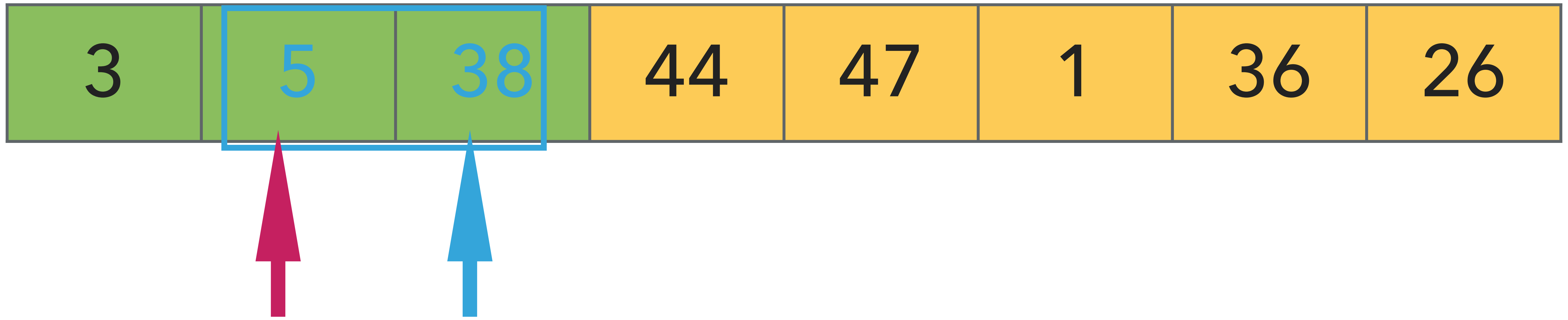
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



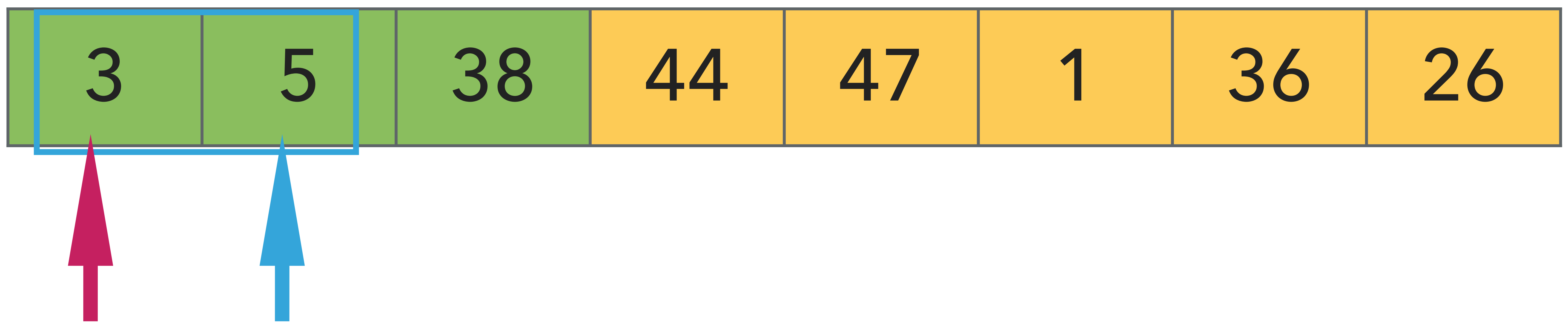
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



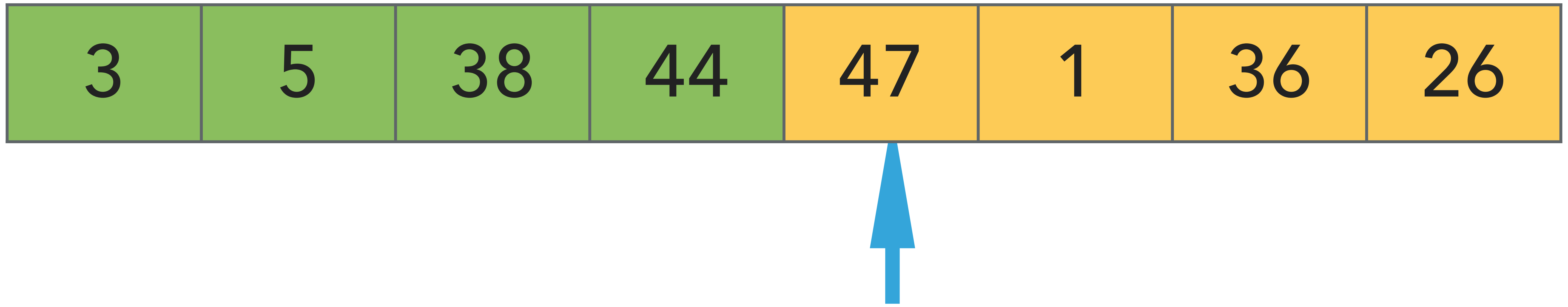
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	5	38	44	47	1	36	26

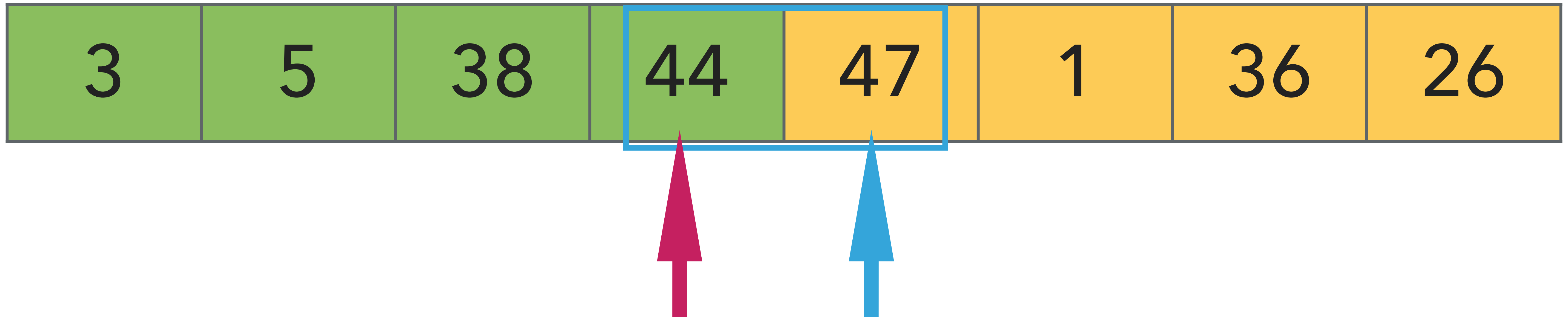
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



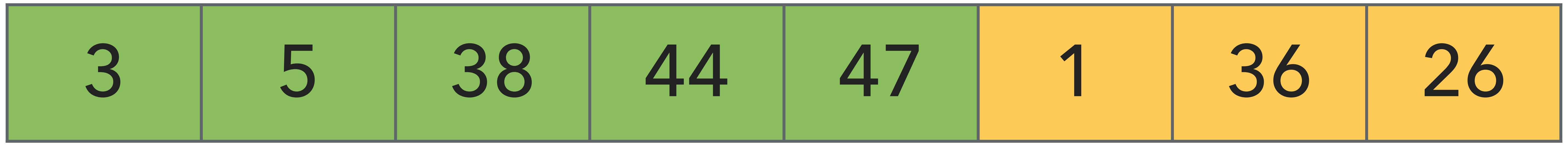
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



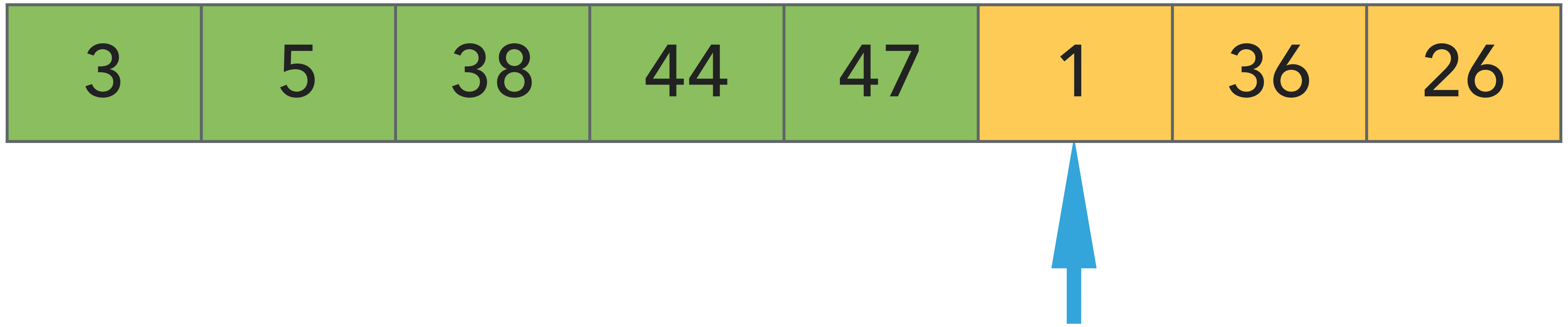
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



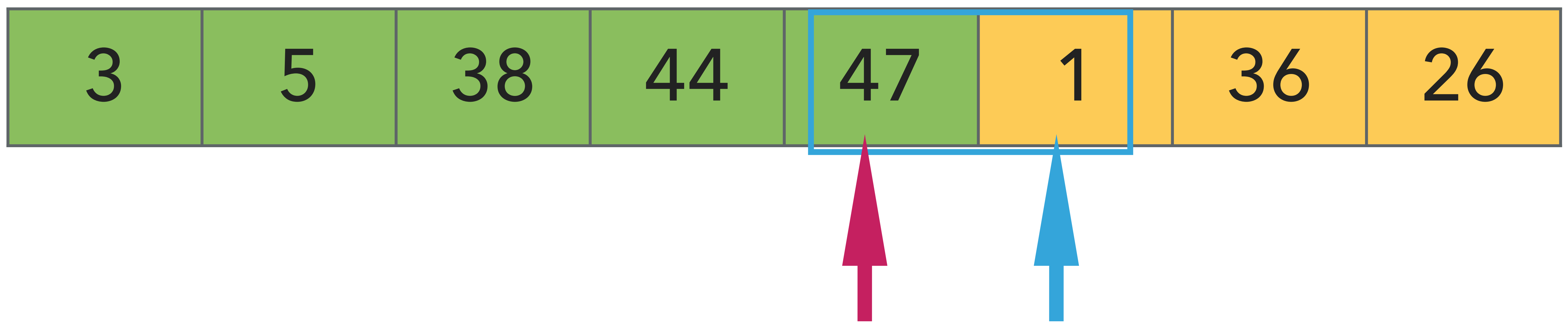
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



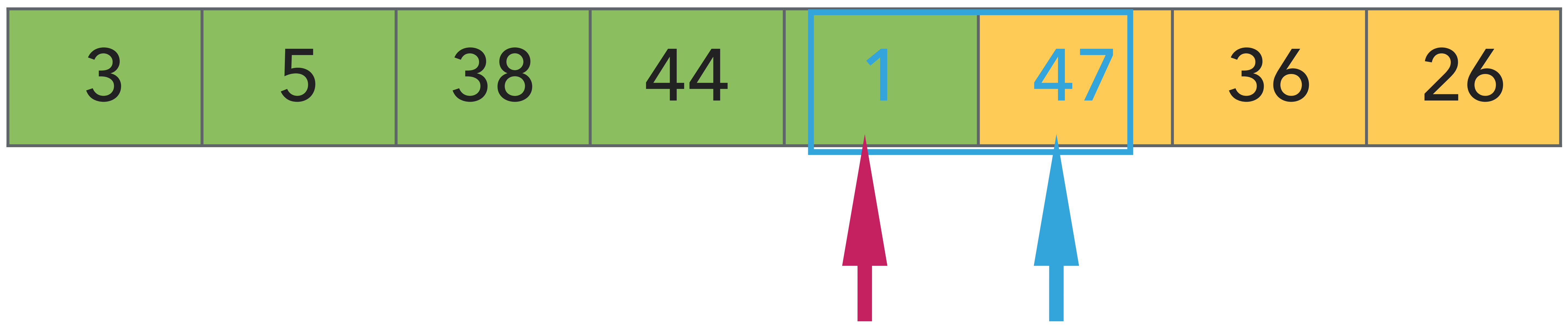
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



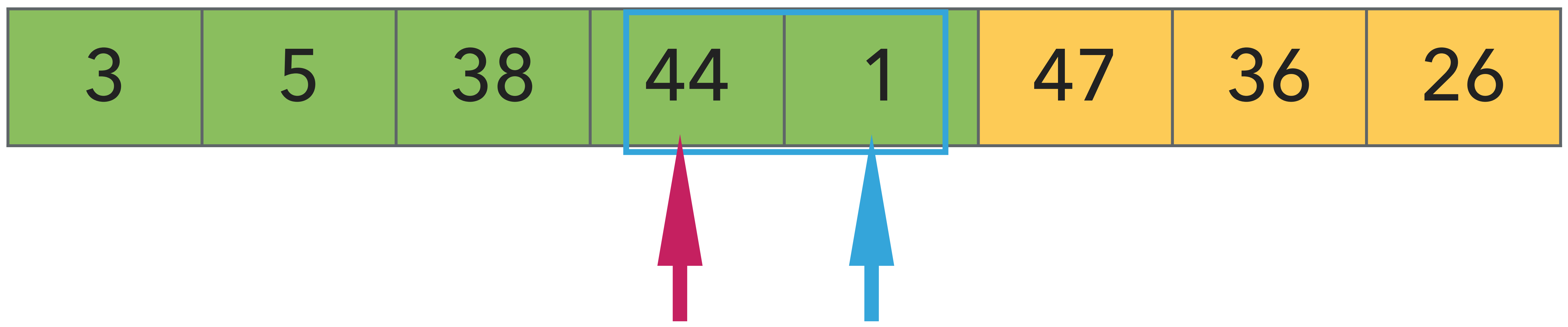
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



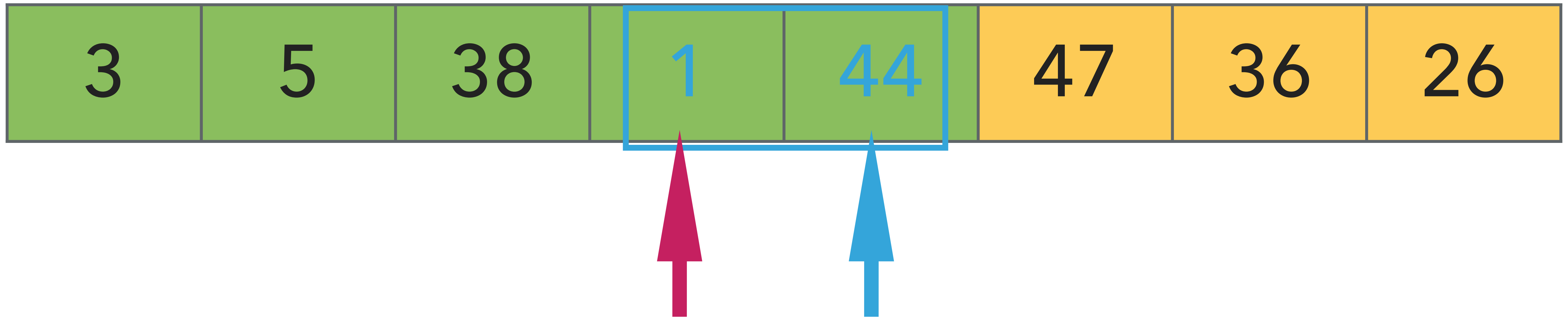
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



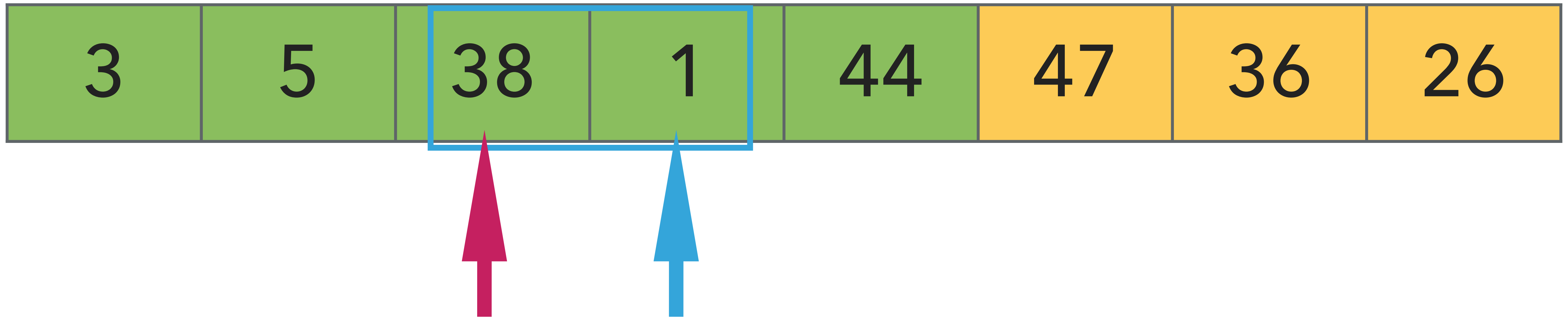
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



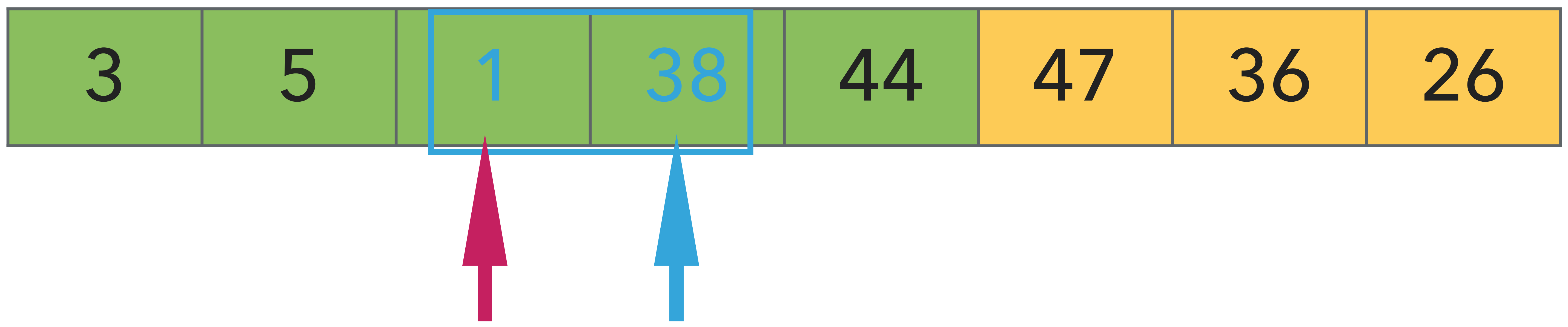
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



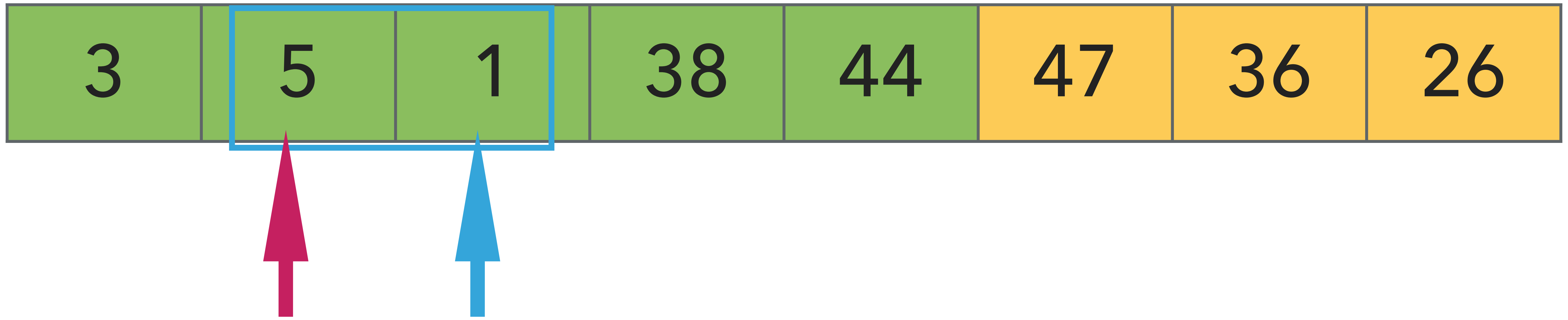
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



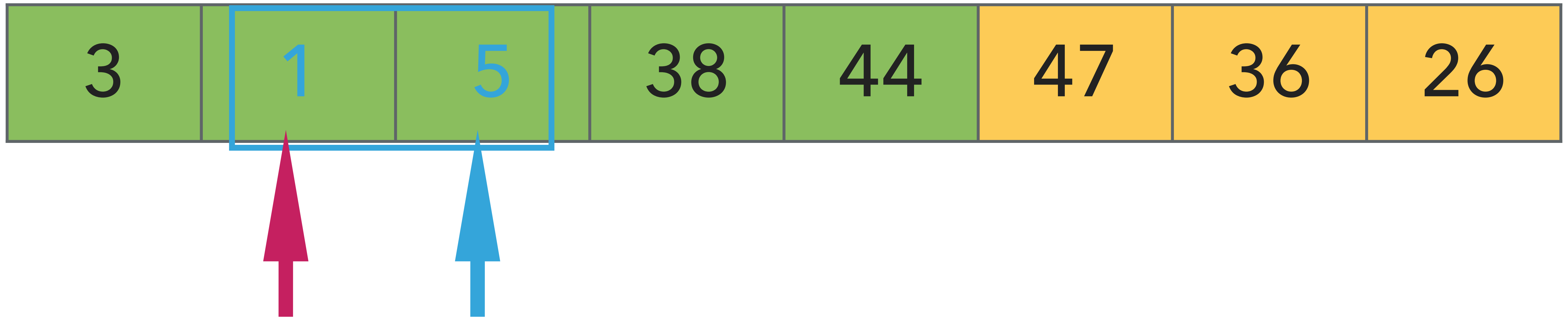
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



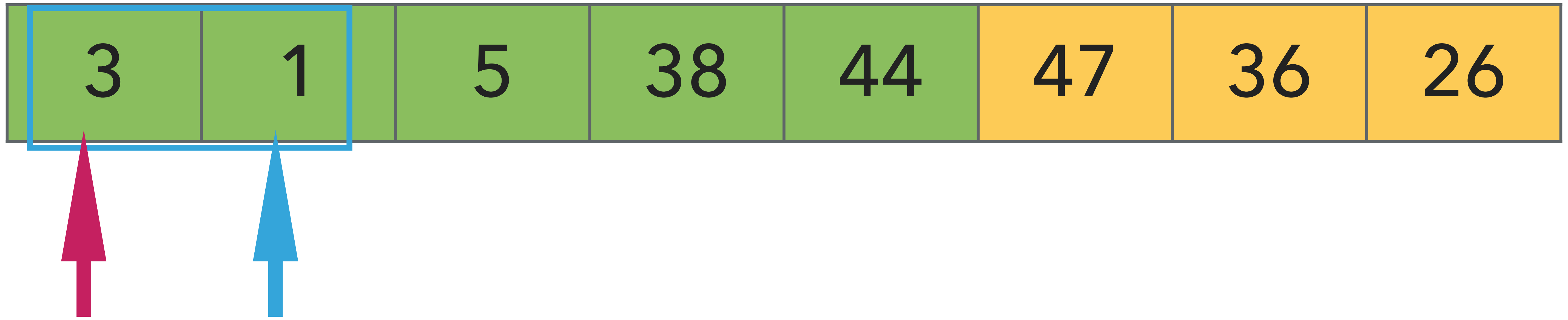
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



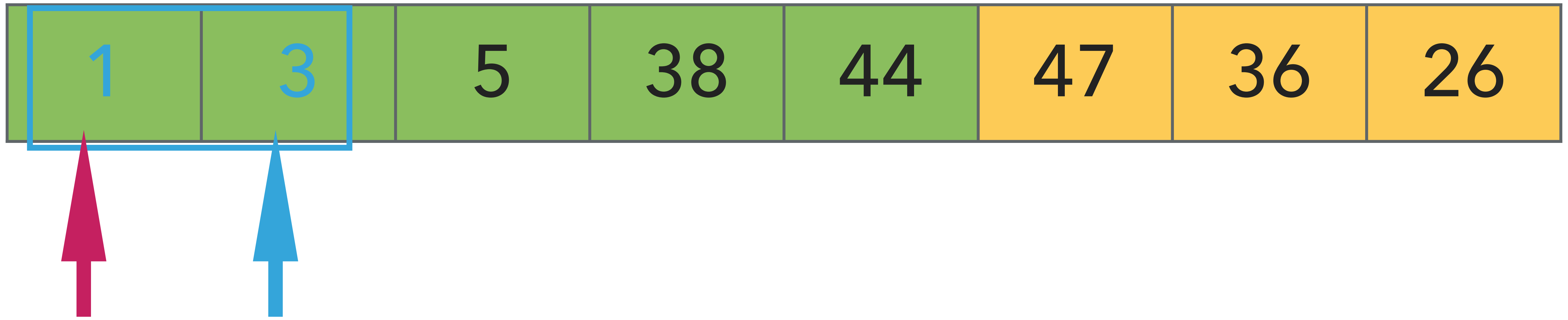
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



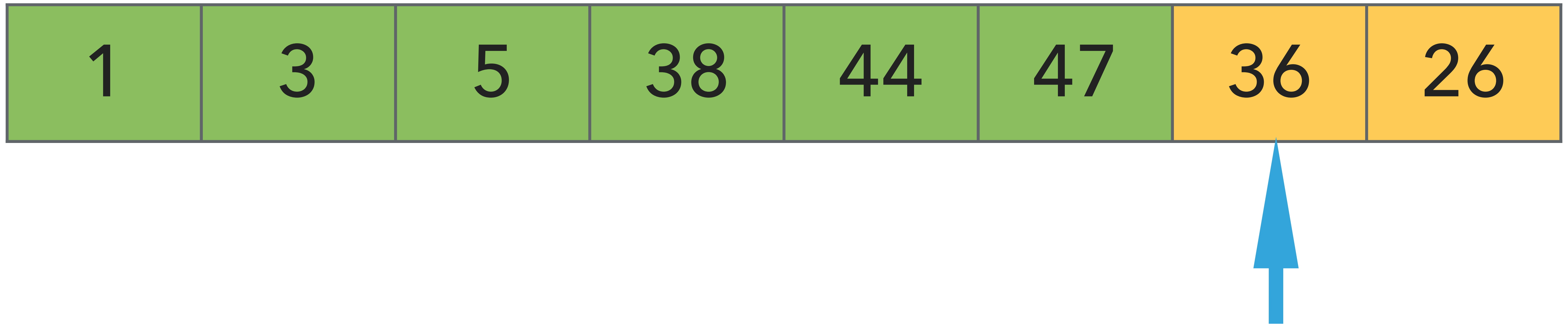
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

1	3	5	38	44	47	36	26

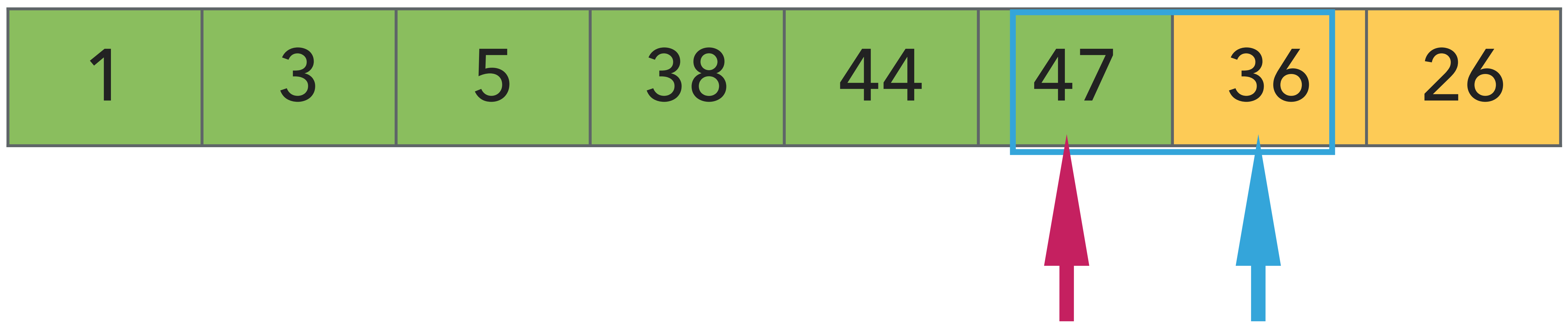
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



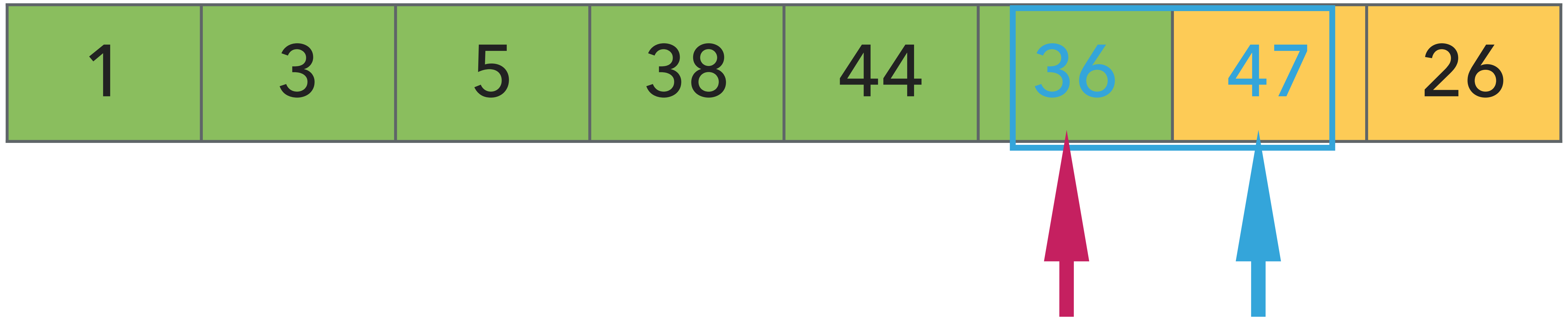
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



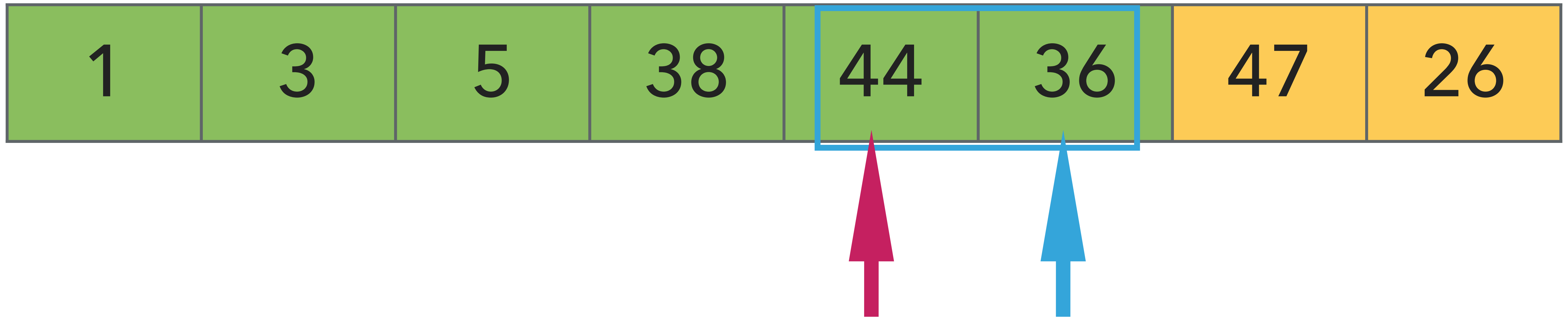
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



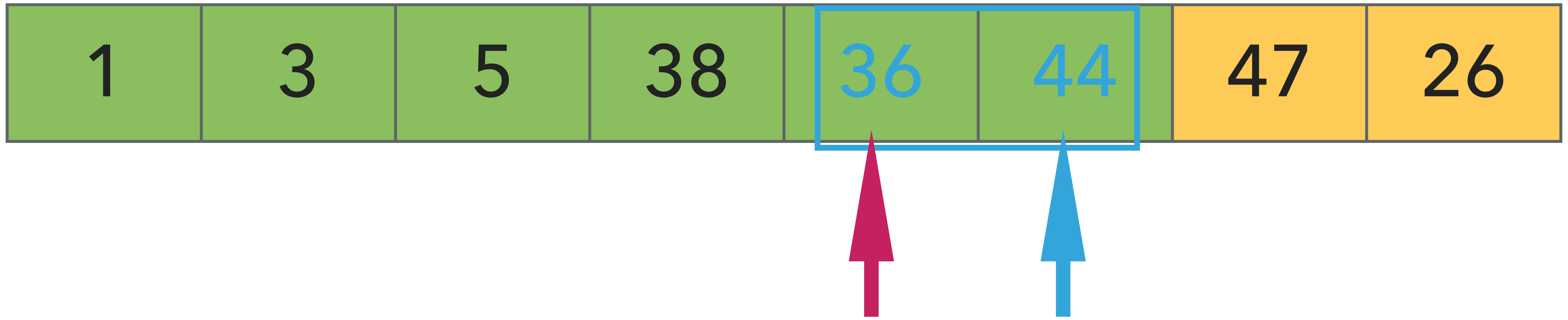
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



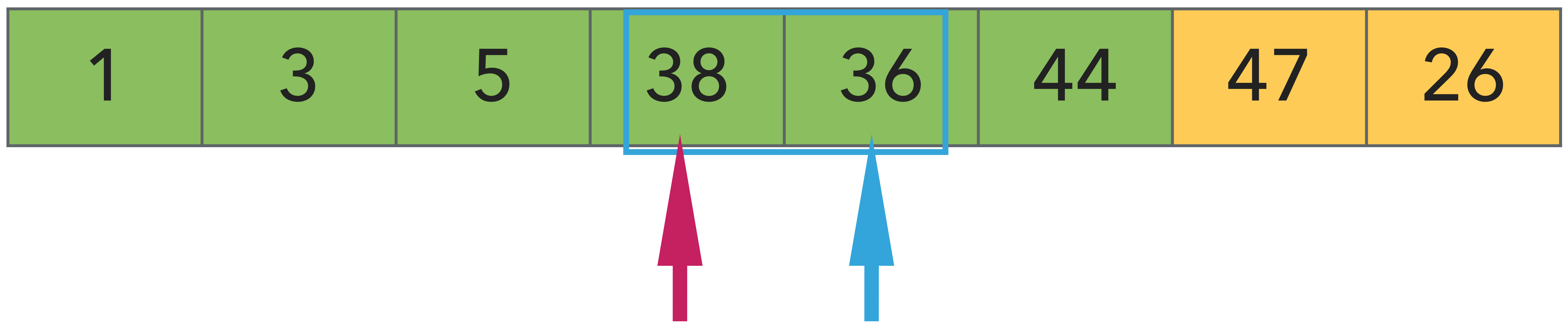
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



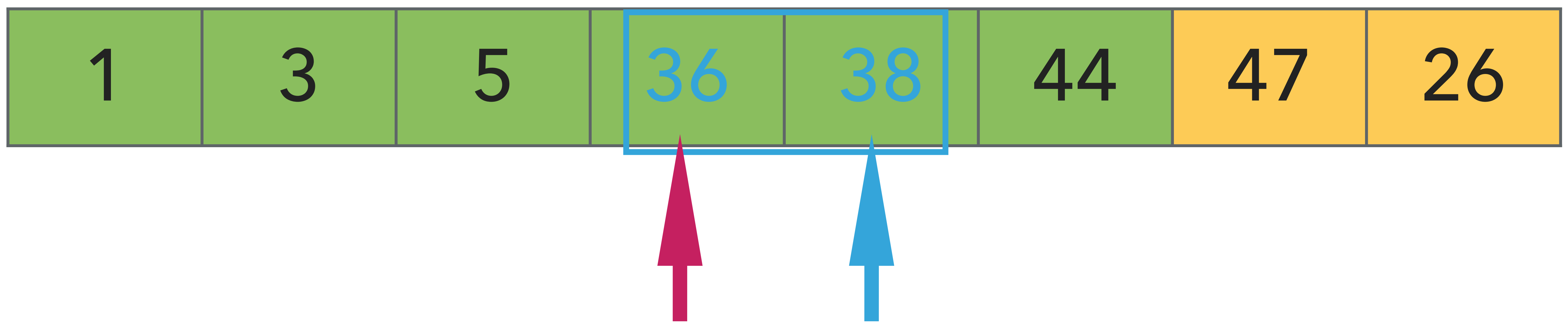
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



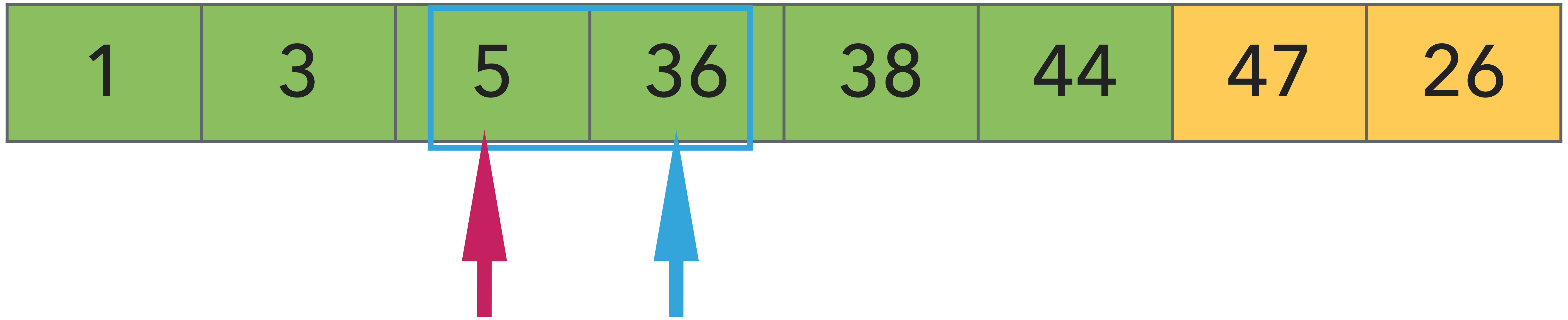
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



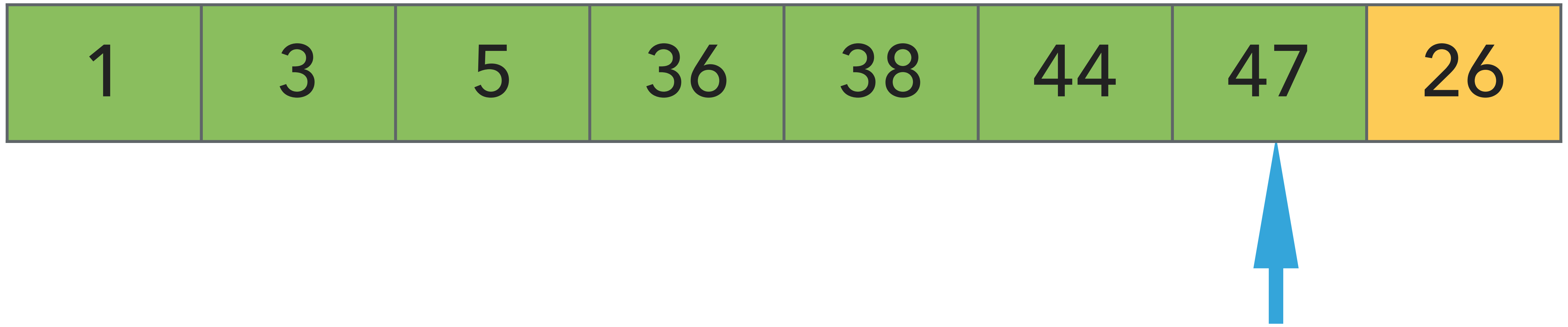
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

1	3	5	36	38	44	47	26

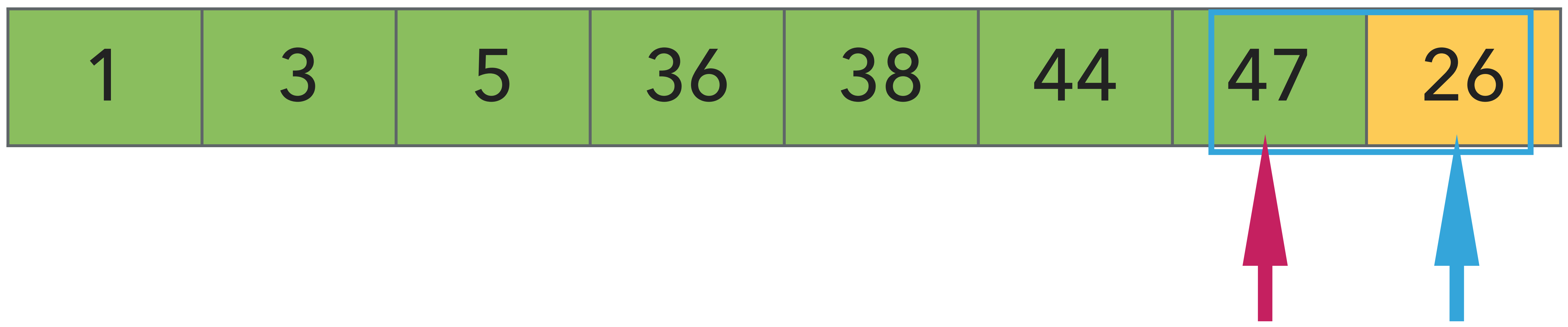
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



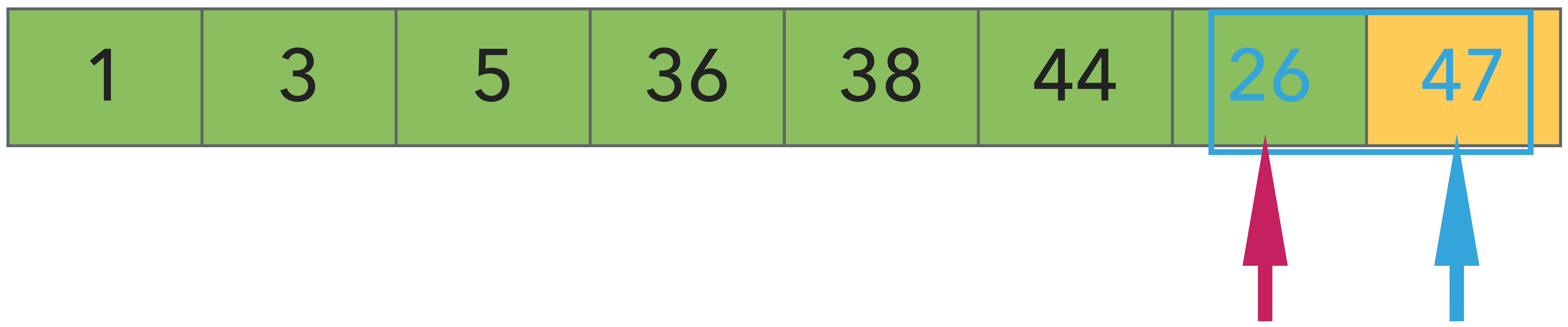
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



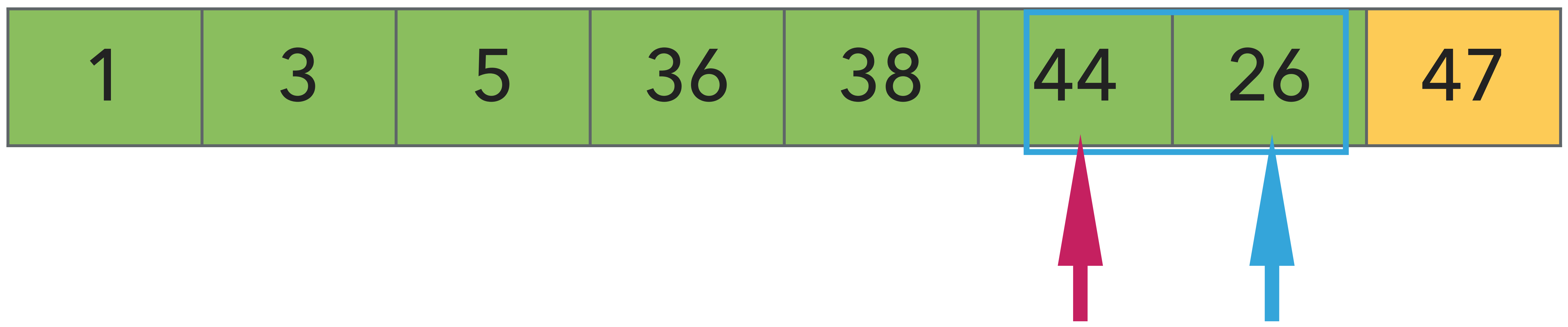
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



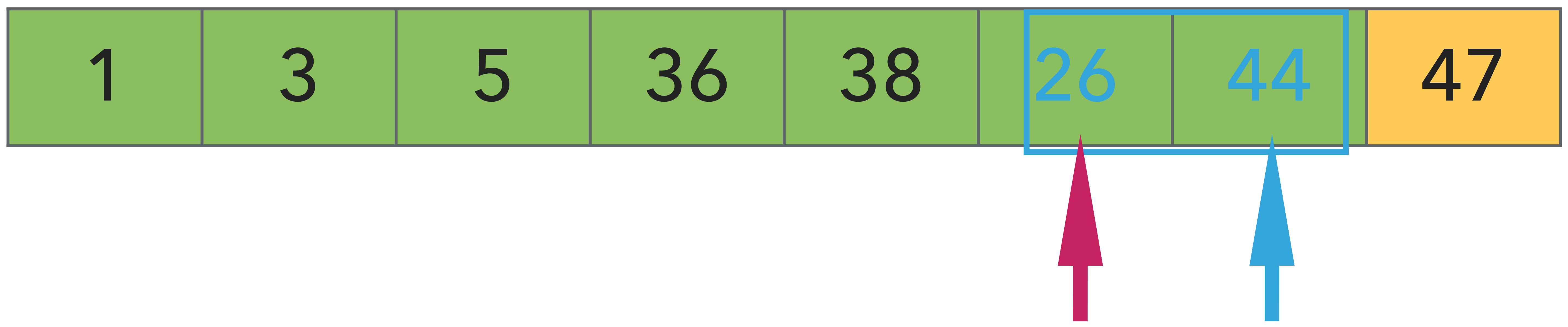
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



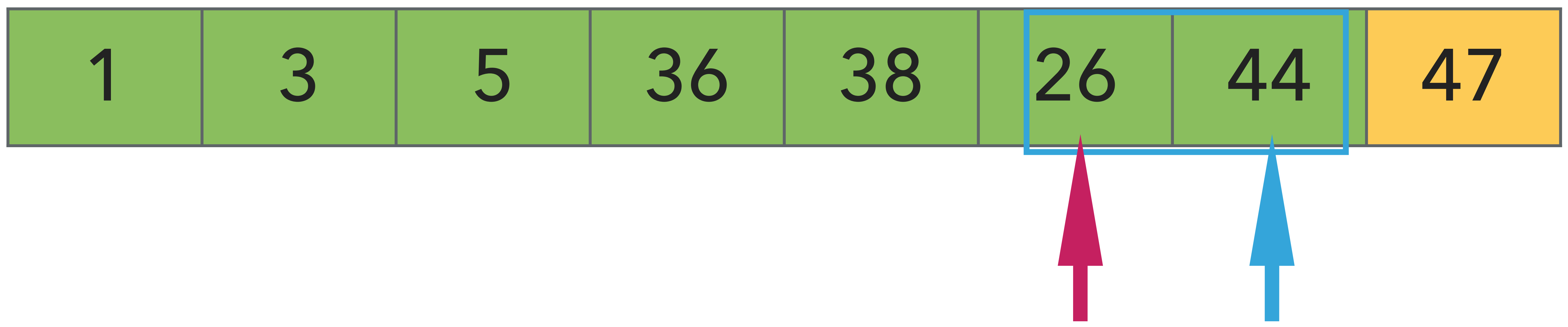
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



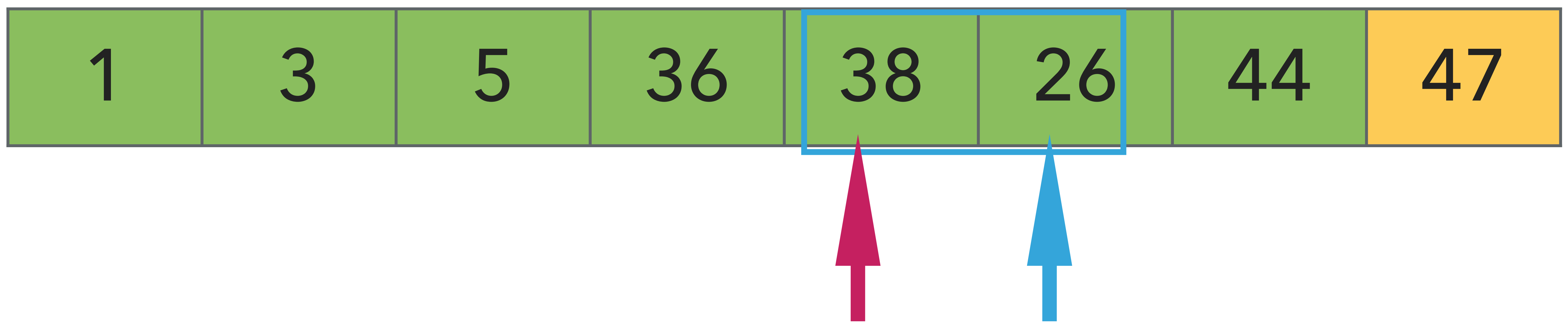
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



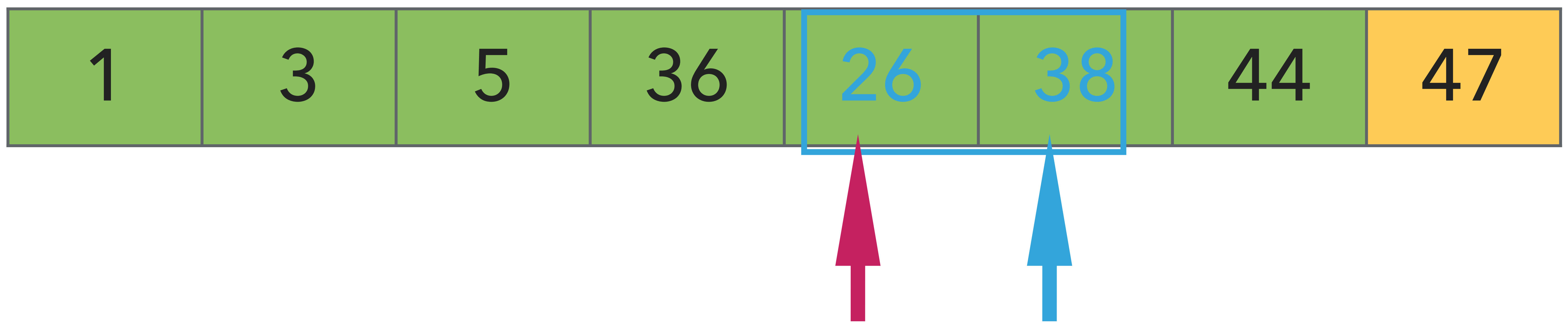
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



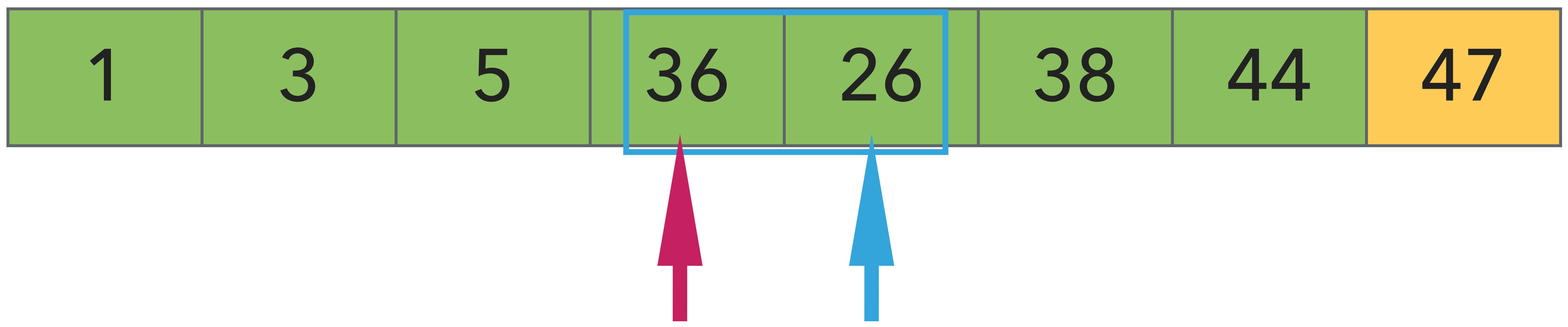
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



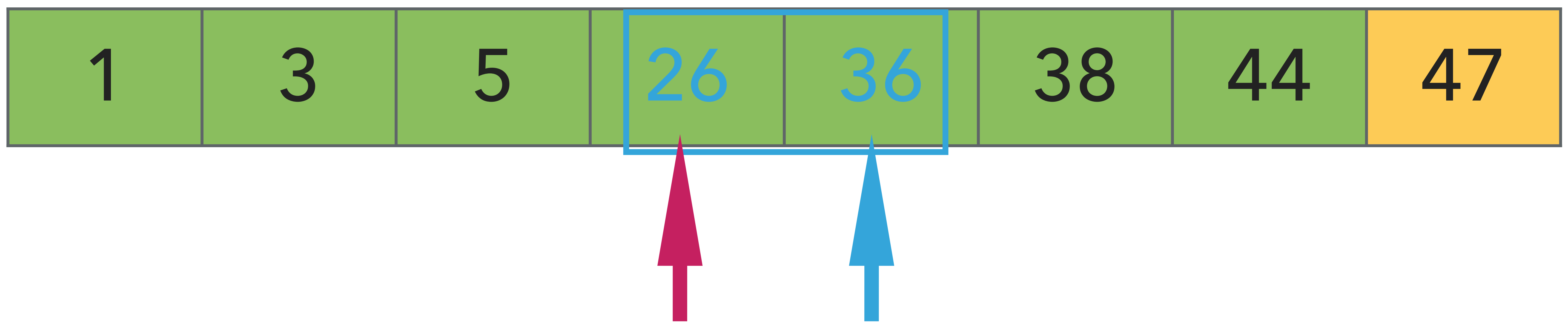
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



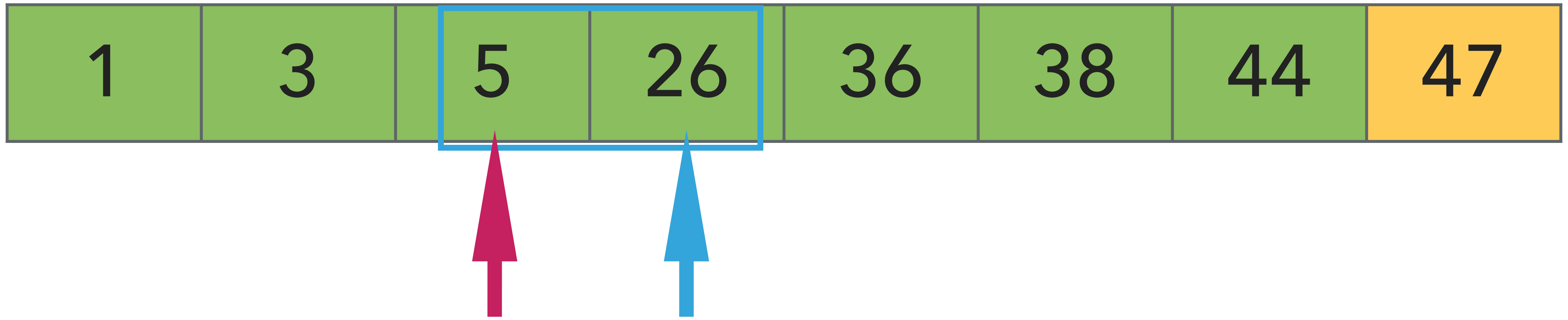
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

1	3	5	26	36	38	44	47

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

http://algs4.cs.princeton.edu

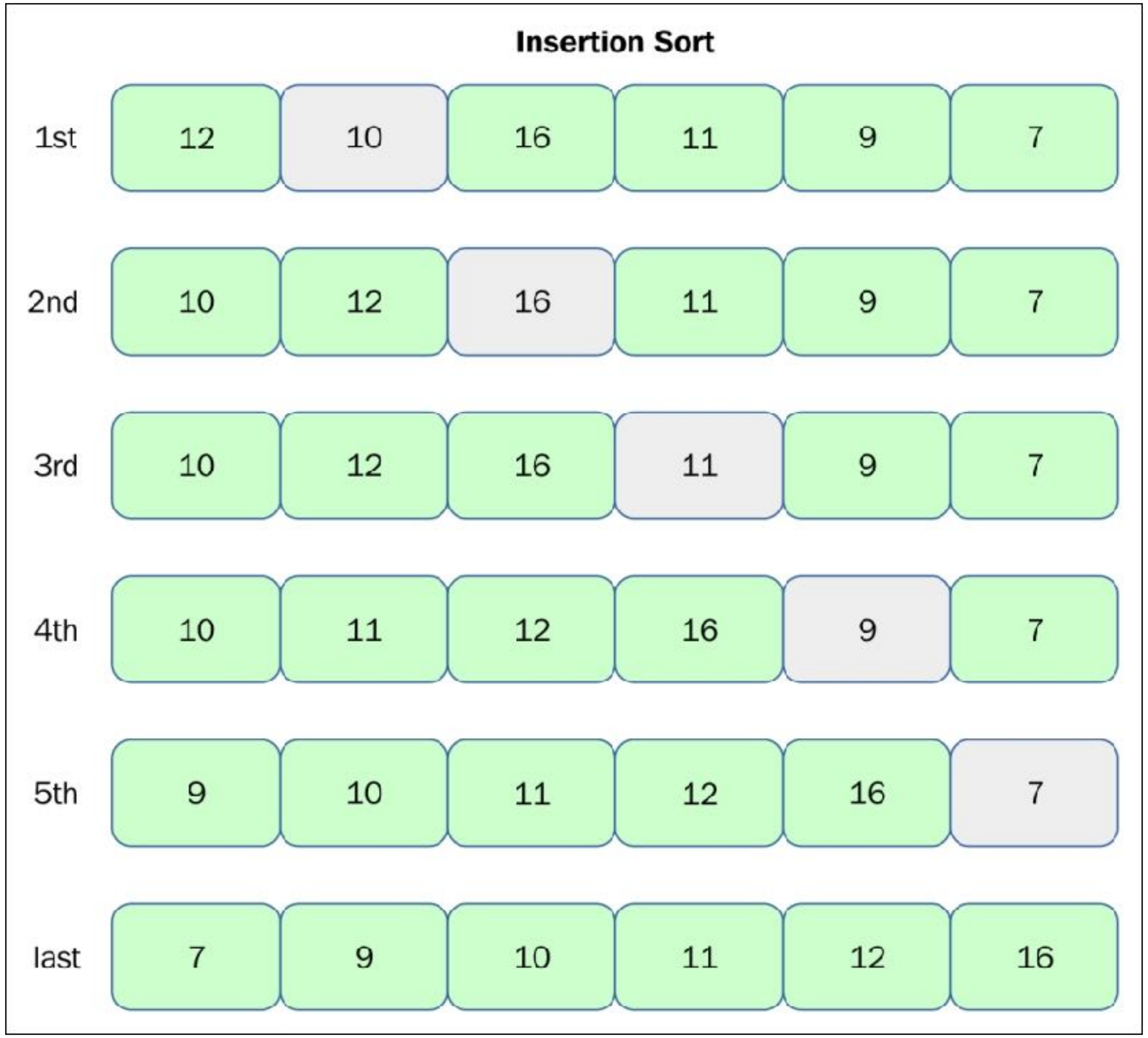
2.1 INSERTION SORT DEMO

https://algs4.cs.princeton.edu/lectures/demo/21DemoInsertionSort.mov https://www.youtube.com/watch?v=EdIKIf9mHk0

Worksheet time!

- Using insertion sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm (draw a new row for each time an element is inserted).

Worksheet answers



https://subscription.packtpub.com/book/application_development/9781785888731/13/ch13lvl1sec90/insertion-sort?query=insertion%20sort

Worksheet time!

- Fill in the blanks to implement insertionSort. If you're done early, answer the other questions:

```
public class InsertionSort {
    public static <E extends Comparable<E>> void insertionSort(E[] a) {
        int n = _______;
            for (int i = 0; i < n; i++) {
                for (int j = ____; ____; ____) {
                    if (a[__].compareTo(a[_______]) < 0) {
                        // do the swap
                        _______________
                        _______________
                        _______________
                    } else {
                        break;
                    }
                }
            }
        }
}
```

Worksheet answers

```
public static <E extends Comparable<E>> void insertionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(a[j].compareTo(a[j-1])<0){
                   E temp = a[j];
                    a[j]=a[j-1];
                    a[j-1]=temp;
               }
               else{
                  break;
               }
           }
       }
 }
```

- Invariants: At the end of each iteration `i`:
  - the array *a* is sorted in ascending order for the first `i+1` elements `a[0…i]`

Insertion sort: mathematical analysis for worst-case

```
public static <E extends Comparable<E>> void insertionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(a[j].compareTo(a[j-1])<0){
                   E temp = a[j];
                   a[j]=a[j-1];
                   a[j-1]=temp;
               }
               else{
                  break;
               }
           }
       }
 }
```

The worst case is the array is in reverse sorted order [5, 4, 3, 2, 1]

- Comparisons: $0 + 1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2$, that is $O(n^2)$.
- Exchanges: $0 + 1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2$, that is $O(n^2)$.
- Worst-case running time is quadratic.
- In-place, requires almost no additional memory.
- Stable

Insertion sort: average and best case

```
public static <E extends Comparable<E>> void insertionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(a[j].compareTo(a[j-1])<0){
                   E temp = a[j];
                   a[j]=a[j-1];
                   a[j-1]=temp;
               }
               else{
                  break;
               }
           }
       }
   }
```

- Best case: $n - 1$ comparisons (go to break statement) and 0 exchanges for an already sorted array.
- Average case: quadratic for both comparisons and exchanges $\sim n^2/4$ when sorting a randomly ordered array.

https://www.toptal.com/developers/sorting-algorithms/insertion-sort

https://stackoverflow.com/questions/17055341/why-is-insertion-sort-%CE%98n2-in-the-average-case

Lecture 12 wrap-up

- Exit ticket: https://forms.gle/3zc3o9ky9LeLjAaj6
- HW5: Compression part 1 due Thurs 11:59pm
- Checkpoint 1 next Tuesday! Please schedule your SDRC exam ASAP if you haven't yet. 1 double sided sheet of handwritten notes allowed.

Resources

- Online textbook website - elementary sorts: https://algs4.cs.princeton.edu/21elementary/
- 8(!) practice problems behind this slide
- Insertion and selection sort are all over the internet since every CS student learns them: Google around for more resources!

Practice Problem 1 – Recommended textbook 2.1.1

- Show all the steps of how selection sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each exchange.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Practice Problem 2 - Recommended textbook 2.1.2

- What is the maximum number of exchanges involving any particular element during selection sort? What is the average number of exchanges involving one specific element x?

Practice Problem 3 - Recommended textbook 2.1.3

- Give an example of an array of n elements that maximizes the number of times the test `a[j].compareTo(a[min])<0` succeeds (and, therefore, `min` gets updated) during the operation of selection sort.

Practice Problem 4 - Recommended textbook 2.1.4

Show all the steps of how insertion sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each insertion.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

Practice Problem 5

- For insertion sort, describe an array of n elements where the if statement in the inner loop is always false and the loop terminates. Now describe an array of n elements where the if statement is always satisfied.

Practice Problem 6 - Recommended textbook 2.1.6

- Which method runs faster for an array with all keys identical (like [1, 1, 1, 1]), selection sort or insertion sort?

Practice Problem 7 - Recommended textbook 2.1.7

- Which method runs faster for an array in reverse order (e.g. [5, 4, 3, 2, 1]), selection sort or insertion sort?

Practice Problem 8 - Recommended textbook 2.1.8

- Suppose that we use insertion sort on a randomly ordered array where items have only one of three values. Is the running time linear, quadratic, or something in between?

ANSWER 1

- Show all the steps of how selection sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each exchange.

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	1	E	A	S	Y	Q	U	E	S	T	I	O	N
1	1	A	E	S	Y	Q	U	E	S	T	I	O	N
2	6	A	E	S	Y	Q	U	E	S	T	I	O	N
3	9	A	E	E	Y	Q	U	S	S	T	I	O	N
4	11	A	E	E	I	Q	U	S	S	T	Y	O	N
5	10	A	E	E	I	N	U	S	S	T	Y	O	Q
6	11	A	E	E	I	N	O	S	S	T	Y	U	Q
7	7	A	E	E	I	N	O	Q	S	T	Y	U	S
8	11	A	E	E	I	N	O	Q	S	T	Y	U	S
9	11	A	E	E	I	N	O	Q	S	S	Y	U	T
10	10	A	E	E	I	N	O	Q	S	S	T	U	Y
11	11	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

ANSWER 2

- What is the maximum number of exchanges involving any particular element during selection sort? What is the average number of exchanges involving one specific element x?
- The maximum number of exchanges is n. See the example below:

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		Z	A	B	C	D	E	F	G	H	I	J	K
0	1	Z	A	B	C	D	E	F	G	H	I	J	K
1	2	A	Z	B	C	D	E	F	G	H	I	J	K
2	3	A	B	Z	C	D	E	F	G	H	I	J	K
3	4	A	B	C	Z	D	E	F	G	H	I	J	K
4	5	A	B	C	D	Z	E	F	G	H	I	J	K
5	6	A	B	C	D	E	Z	F	G	H	I	J	K
6	7	A	B	C	D	E	F	Z	G	H	I	J	K
7	8	A	B	C	D	E	F	G	Z	H	I	J	K
8	9	A	B	C	D	E	F	G	H	Z	I	J	K
9	10	A	B	C	D	E	F	G	H	I	Z	J	K
10	11	A	B	C	D	E	F	G	H	I	J	Z	K
11	11	A	B	C	D	E	F	G	H	I	J	K	Z
		A	B	C	D	E	F	G	H	I	J	K	Z

- The average number of exchanges for a specific element is exactly 2, because there are exactly n exchanges and n items (and each exchange involves two items).

ANSWER 3

- Give an example of an array of n elements that maximizes the number of times the test `a[j].compareTo(a[min])<0` succeeds (and, therefore, `min` gets updated) during the operation of selection sort.
- Any array in reverse order would do, for example, [6, 5, 4, 3, 2, 1].

ANSWER 4

Show all the steps of how insertion sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each insertion.

		a[ ]											
i	j	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	0	E	A	S	Y	Q	U	E	S	T	I	O	N
1	0	A	E	S	Y	Q	U	E	S	T	I	O	N
2	2	A	E	S	Y	Q	U	E	S	T	I	O	N
3	3	A	E	S	Y	Q	U	E	S	T	I	O	N
4	2	A	E	Q	S	Y	U	E	S	T	I	O	N
5	4	A	E	Q	S	U	Y	E	S	T	I	O	N
6	2	A	E	E	Q	S	U	Y	S	T	I	O	N
7	5	A	E	E	Q	S	S	U	Y	T	I	O	N
8	6	A	E	E	Q	S	S	T	U	Y	I	O	N
9	3	A	E	E	I	Q	S	S	T	U	Y	O	N
10	4	A	E	E	I	O	Q	S	S	T	U	Y	N
11	4	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

ANSWER 5

- For insertion sort, describe an array of n elements where the if statement in the inner loop is always false and the loop terminates. Now describe an array of n elements where the if statement is always satisfied.

- if statement always false when the array is already sorted, e.g., [1, 2, 3, 4], and also if all the elements are the same, e.g. [1, 1, 1, 1]
- if statement always true when the array is in reverse order, e.g., [4, 3, 2, 1].

ANSWER 6

- Which method runs faster for an array with all keys identical, selection sort or insertion sort?
- Insertion sort is faster because it will only make one comparison per element (i.e., is linear) and will not need to exchange any elements. Instead, selection sort will still run in quadratic time.

ANSWER 7

- Which method runs faster for an array in reverse order, selection sort or insertion sort?
- Selection sort. Big O says both are quadratic, but selection sort needs only n exchanges, while insertion sort $n^2/2$ exchanges

ANSWER 8

- Suppose that we use insertion sort on a randomly ordered array where items have only one of three values. Is the running time linear, quadratic, or something in between?
- Quadratic. Insertion sort's running time is linear when the array is already sorted or all elements are equal. With three possible values the running time is quadratic.