CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

11: Sorting Fundamentals and Comparators

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his

Bio: Professional

- Dr. Tom Yeh
  - Ph.D. - UCLA
  - B.S. - UC Berkeley
- Research Interests
  - Computer Architecture
  - Machine Learning
  - Architectural Acceleration of Machine Learning:

    Ultra-low precision training and inference

- Computer architect by training. Worked on CPU designs at a startup, Intel, Sun Micro.

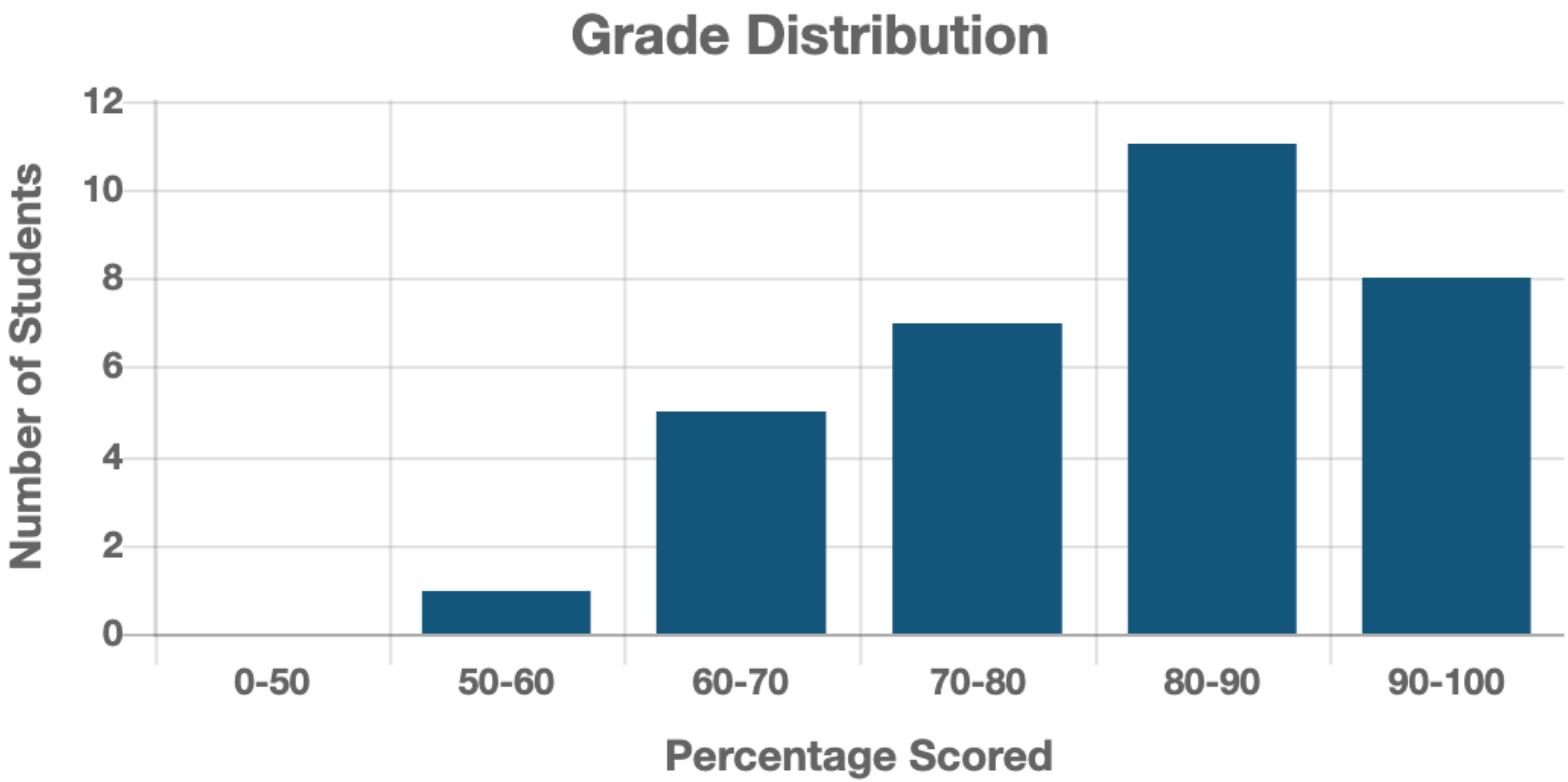
Bio: Personal Interests

Lecture 11: Sorting Fundamentals

- Midterm Grade Distribution
- Iterator and Iterable Interfaces
- Sorting

Some slides adopted from Algorithms 4th Edition or COS226

Grade Statistics for Midterm I



Average (mean) grade	64.94
Median grade	67.00
Standard deviation	8.25
Lowest grade	46.00
Highest grade	77.00
Total graded	32

Maximum = 80

`Iterable` Interface

- What is an **Iterable**?
  - Class with a method that returns an Iterator
- What is an **Iterator**?
  - Class with methods hasNext() and next()
- Why make data structures Iterable?
  - To support elegant code
  - Interface that allows an object to be the target of a for-each loop:

```
// "foreach" statement (shorthand)
for(String s: stack){
  System.out.println(s);
}
```

```
myList.forEach(System.out::println);
```

```
public interface Iterable<Item>
{
    Iterable<Item> iterator();
}
```

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove; // don't use
}
```

```
// equivalent code (longhand)
Iterator<String> i = stack.iterator();
while (i.hasNext()
{
    String s = i.next();
    StdOut.println(s);
}
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

ITERATORS

Example: making ArrayList iterable

- Start with a class, implement iterable, within class you will implement iterator

```
public class ArrayList<Item> implements Iterable<Item> {
     // Have the class implement Iterable

    public Iterator<Item> iterator() {
        // Need this method iterator that returns an iterator
        return new ArrayListIterator();
     }

    // Have this inner class which implements Iterator
    private class ArrayListIterator implements Iterator<Item> {

        private int i = 0;

        public boolean hasNext() {
               return i < n;
        }

        public Item next() {
            return a[i++];
        }

        public void remove() {
                throw new UnsupportedOperationException();

        }
    }
}
```

Example: making Stack iterable (linked-list implementation)

```
public class Stack<Item> implements Iterable<Item> {
     // Have the class implement Iterable

    public Iterator<Item> iterator() {
        // Need this method iterator that returns an iterator
        return new stackIterator();
     }

    // Have this inner class which implements Iterator
    private class stackIterator implements Iterator<Item> {
        //
        private Node current = first;

        public boolean hasNext() {
               return current != null;
        }

        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }

        public void remove() {
                throw new UnsupportedOperationException();

        }
    }
}
```

`Iterator` Interface

- Interface that allows us to traverse a collection one element at a time.

```
public interface Iterator<E> {
  //returns true if the iteration has more elements
  //that is if next() would return an element instead of throwing an exception
  boolean hasNext();

  //returns the next element in the iteration
  //post: advances the iterator to the next value
  E next();

  //removes the last element that was returned by next
  //optional, better avoid it altogether
  // default void remove();
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Traversing `ArrayList`

- Once you implement the Iterable interface, here are some valid ways to traverse ArrayList and print its elements one by one.

```
for(String elt:a1) {
    System.out.println(elt);
}

a1.forEach(System.out::println);

a1.iterator().forEachRemaining(System.out::println);
```

-

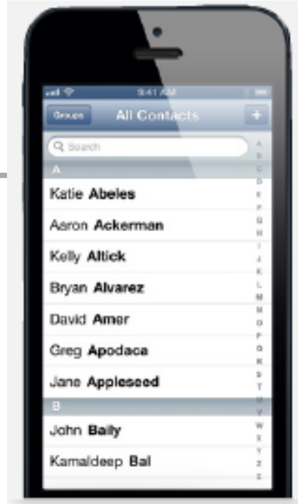
- Iterable: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

Lecture 11: Sorting Fundamentals

- Introduction
- Selection sort
- Insertion sort

Some slides adopted from Algorithms 4th Edition or COS226

Why study sorting?



contacts

FedEx packages

- It's more common than you think: e.g., sorting flights by price, contacts by last name, files by size, emails by day sent, neighborhoods by zipcode, etc.
- Good example of how to compare the performance of different algorithms for the same problem.
- Some sorting algorithms relate to data structures.
- Sorting your data will often be a good starting point when solving other problems (keep that in mind for interviews).

Definitions

- Sorting: the process of arranging n items of a collection in non-decreasing order (e.g., numerically or alphabetically).
  - Rearrange array of N items into ascending order
- Key: assuming that an item consists of multiple components, the key is the property based on which we sort items.
- Goal: sort **any** type of data according to the key

	Chen	3	A	991-878-4944	308 Blair
	Rohde	2	A	232-343-5555	343 Forbes
	Gazsi	4	B	766-093-9873	101 Brown
item →	Furia	1	A	766-093-9873	101 Brown
	Kanaga	3	B	898-122-9643	22 Brown
	Andrews	3	A	664-480-0023	097 Little
key →	Battle	4	C	874-088-1212	121 Whitman

Andrews	3	A	664-480-0023
Battle	4	C	874-088-1212
Chen	3	A	991-878-4944
Furia	1	A	766-093-9873
Gazsi	4	B	766-093-9873
Kanaga	3	B	898-122-9643
Rohde	2	A	232-343-5555

Total order: It must be possible to put items in order

- Sorting is well defined if and only if there is total order.
- Total order: a binary relation $\leq$ on a set C that satisfies the following statements for all v, w, and x in C:
  - Connexity: $v \leq w$ or $w \leq v$.
  - Transitivity: for all v, w, x, if $v \leq w$ and $w \leq x$ then $v \leq x$.
  - Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Ex: standard order for numbers, alphabetical order for strings, chronological order for dates

How many different algorithms for sorting can there be?

- 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
- …

Rules of the game - Comparing

- We will be sorting arrays of n items, where each item contains a key. In Java, **objects** are responsible in telling us how to ***naturally* compare** their keys.
- Let's say we want to sort an array of objects of type T.
- Our class T should implement the `Comparable` interface (more on this in a few lectures). We will need to implement the `compareTo` method to satisfy a total order.
- Sort has no dependence on data type

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
   public int compareTo(Item that);
}
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

sort implementation

```
public static void sort(Comparable[] 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)
            exch(a, j, j-1);
         else break;
}
```

Rules of the game - Comparing

- `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 (positive) if v is greater than `w`.
  - Returns <0 (negative) if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.
  - Is a total order.
- Java classes such as `Integer`, `Double`, `String`, `File` all implement `Comparable`.
- Need to implement the Comparable interface for user-defined comparable types.
- compareTo allows us to use the same sorting algorithms on different data

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Implementing the Comparable interface

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

Two primary sorting abstractions

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

```
private static boolean less(Comparable v, Comparable w) {
     return v.compareTo(w) < 0;
}
```

- Exchange: swap item in array a[] at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j) {
        Comparable swap = a[i];
        a[i]=a[j];
        a[j]=swap;
}
```

- Sort method will use these 2 methods

Which total order property is violated?

```
public class Temperature implements Comparable<Temperature> {
    private final double degrees;

    // Constructor code ....

    public int compareTo(Temperature that) {
        double EPSILON = 0.1;
        if (this.degrees < that.degrees - EPSILON) return -1;
        if (this.degrees > that.degrees + EPSILON) return +1;
        return 0;
    }
    ...
}
```

- Connexity: $v \leq w$ or $w \leq v$.
- Transitivity: for all v, w, x, if $v \leq w$ and $w \leq x$ then $v \leq x$.
- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.

Rules of the game - Cost model

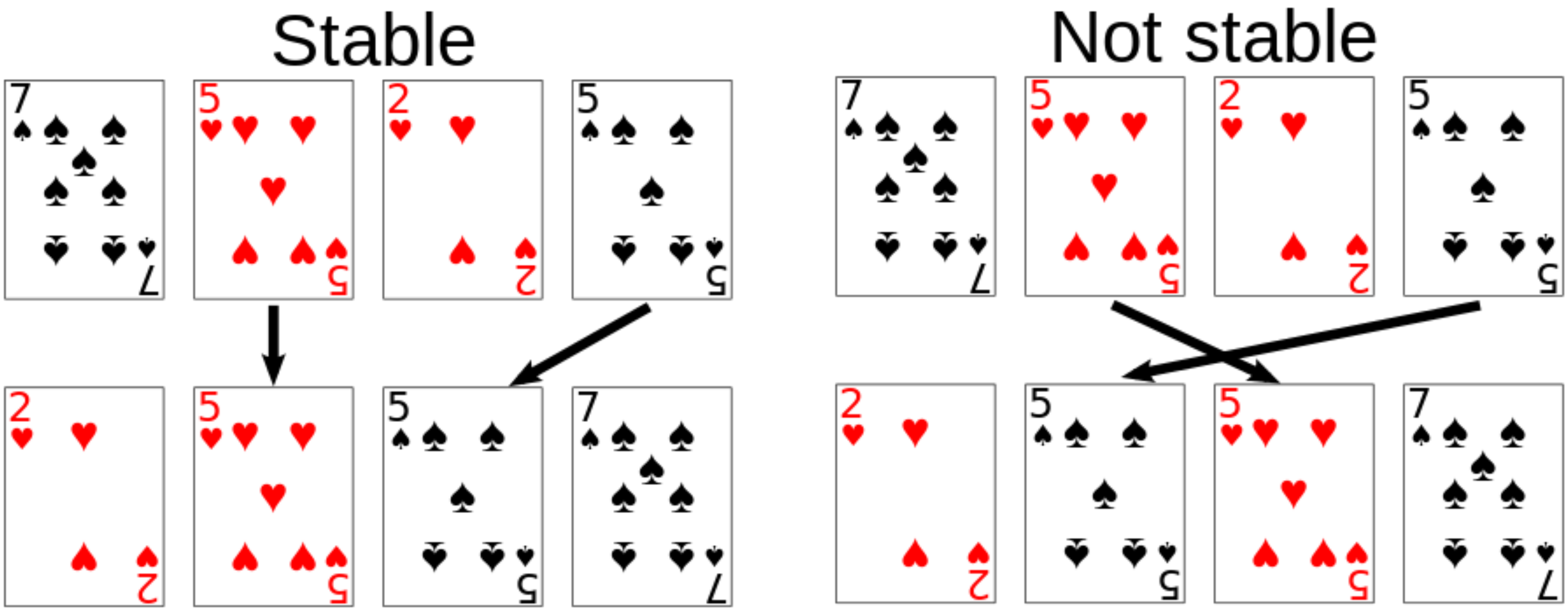
- Sorting cost model: we count compares and exchanges. If a sorting algorithm does not use exchanges, we count array accesses.
- Compares, exchanges, array accesses give us an estimate on the time complexity
- There are other types of sorting algorithms where they are not based on comparisons (e.g., radixsort). We will not see these in CS62 but stay tuned for CS140.

Rules of the game - Memory usage

- Extra memory: often as important as running time. Sorting algorithms are divided into two categories:
  - In place: use constant or logarithmic extra memory, beyond the memory needed to store the items to be sorted.
  - Not in place: use linear auxiliary memory.

Rules of the game - Stability

- Stable: 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

Lecture 11: Sorting Fundamentals

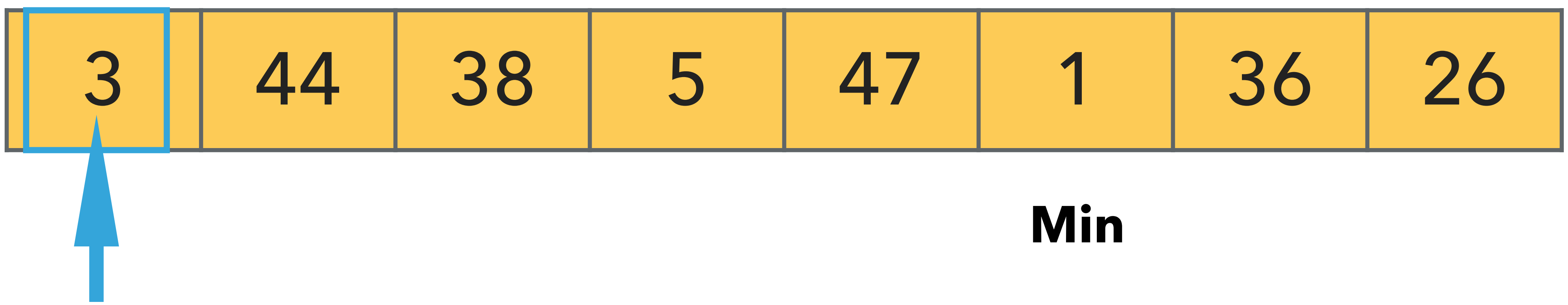
- Introduction
- Selection sort
- Insertion sort

Selection sort

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

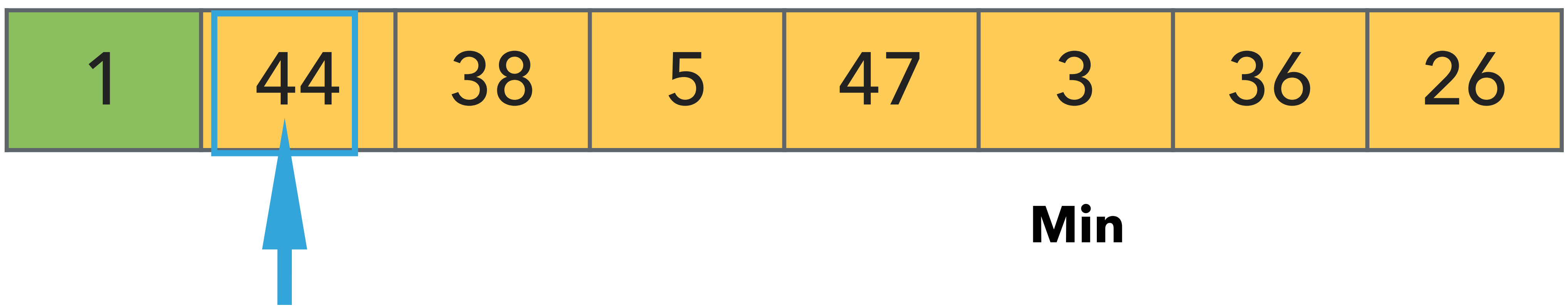
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

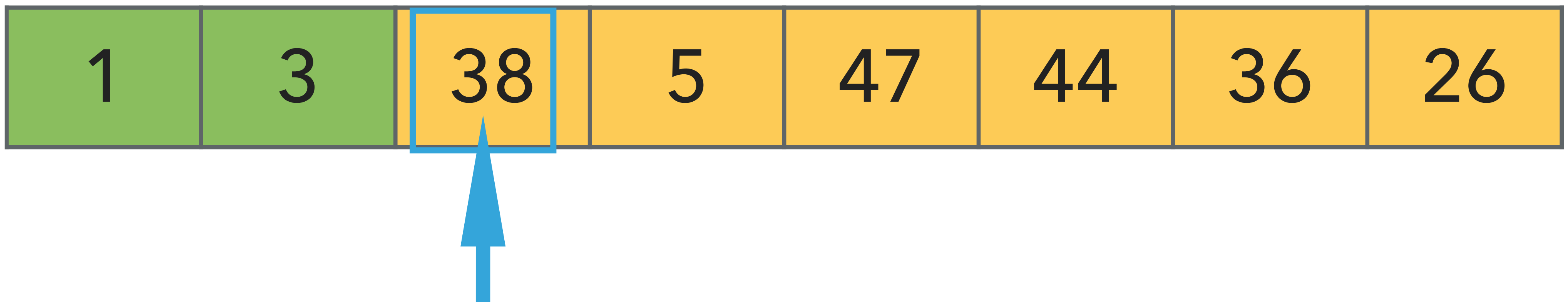
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

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

1	3	5	38	47	44	36	26

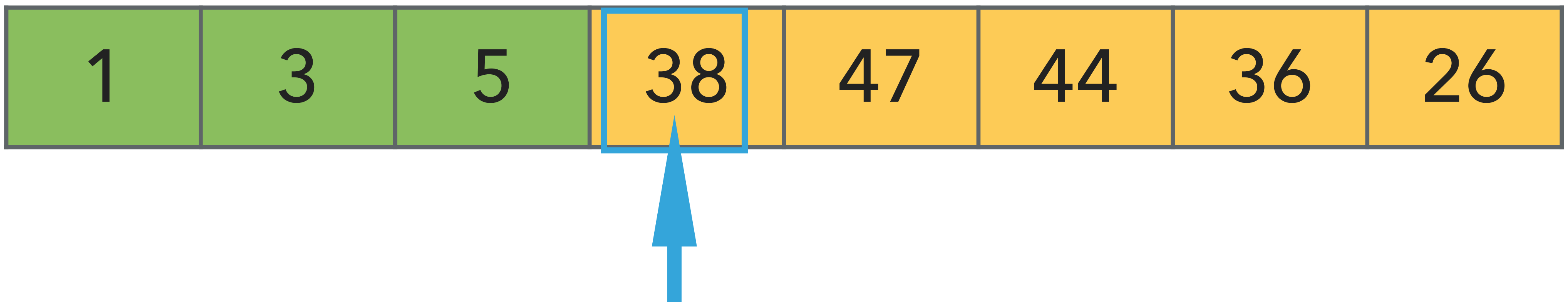
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

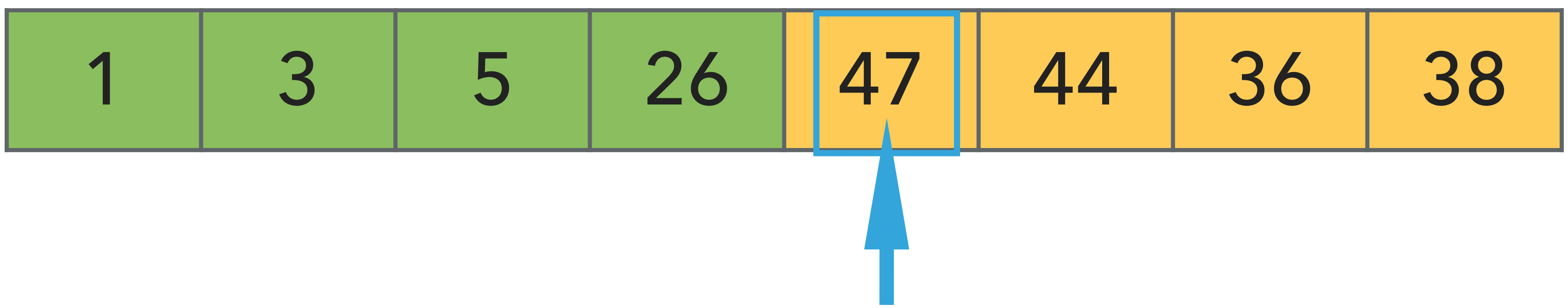
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

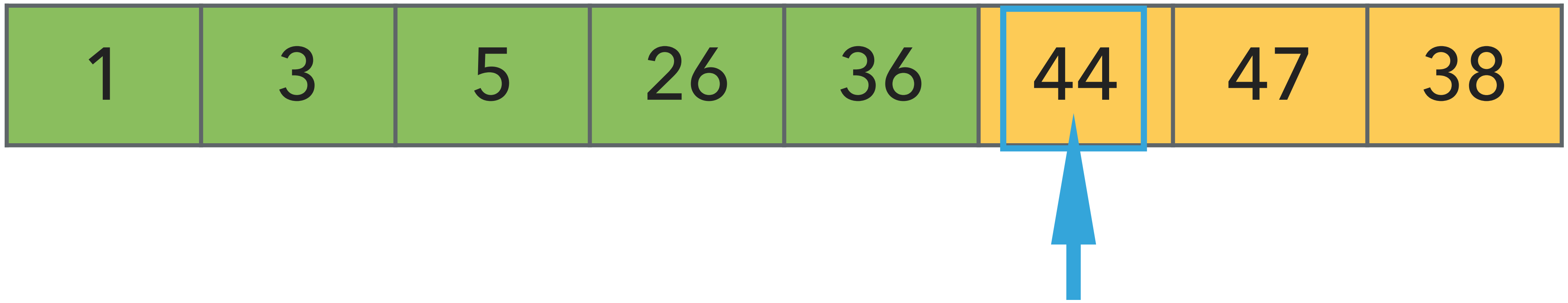
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

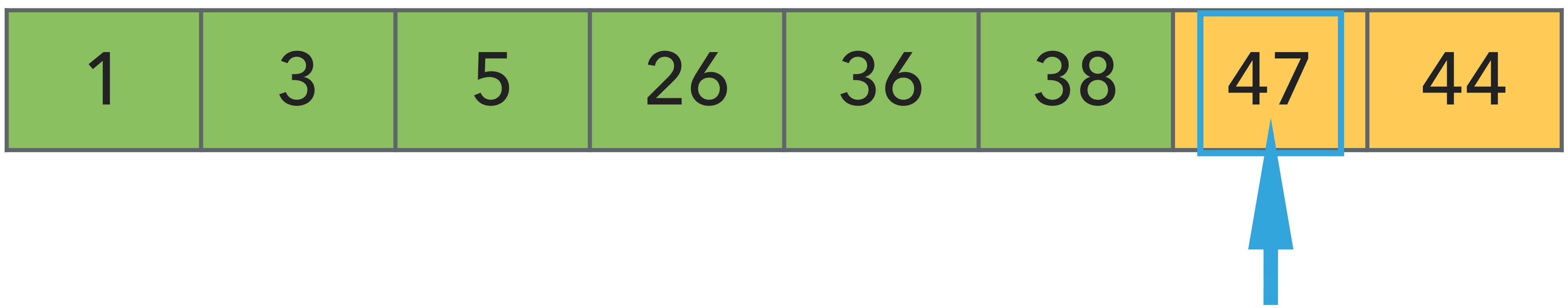
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

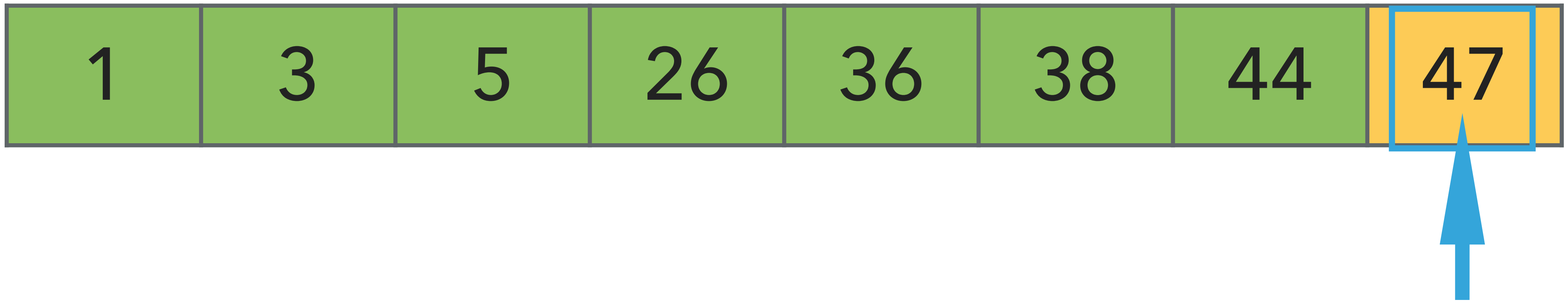
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

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

```
public static void sort(Comparable[] a) {
```

- Move the pointer to the right.

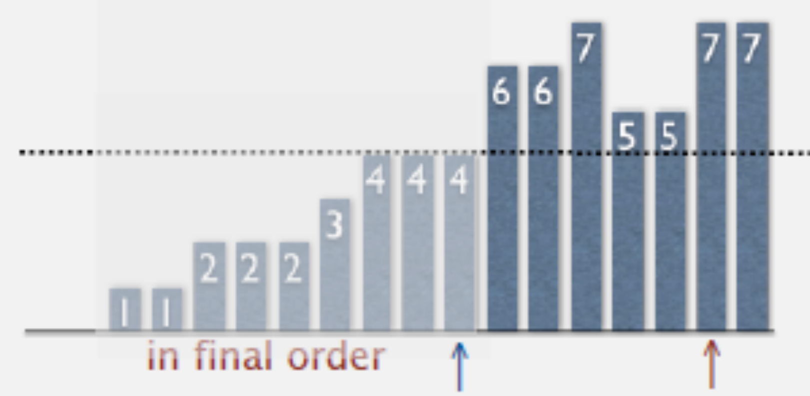
```
i++;
```

- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```




```
}
```

Selection sort

```
public static void selection_sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {        ← In iteration i
            int min = i;
            for (int j = i+1; j < n; j++) {  ← Find the index min of the
                if (less(a[j], a[min]))        smallest remaining array
                    min = j;
            }
            exch(a, i, min);                 ← swap a[i] and a[min]
        }
    }
```

- 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]` is smaller than any entry in `a[0…i]`

Selection sort: mathematical analysis for worst-case

```
public static void selection_sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
        }
    }
```

- Comparisons:
- Exchanges:
- In-place?
- Stable?

Selection sort: mathematical analysis for worst-case

```
public static void selection_sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
        }
    }
```

- Comparisons: $1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, that is $O(n^2)$.
- Exchanges:
- In-place?
- Stable?

Selection sort: mathematical analysis for worst-case

```
public static void selection_sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
       }
   }
```

- Comparisons: $1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, 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. (Does NOT depend on the input)
- In-place?
- Not stable?

Selection sort: mathematical analysis for worst-case

```
public static void selection_sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
        }
    }
```

- Comparisons: $1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, 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. (Does NOT depend on the input)
- 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].

Practice Time

- Using selection sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm.

SELECTION SORT

Answer

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

Lecture 11: Sorting Fundamentals

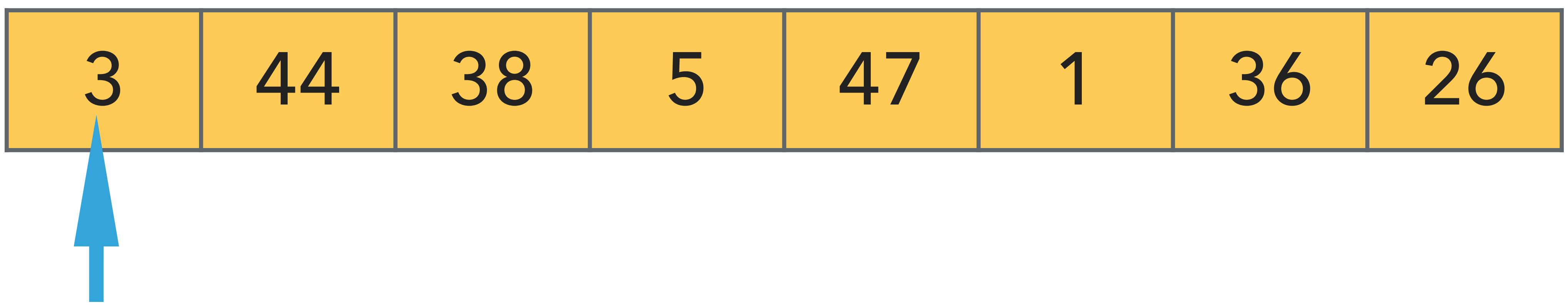
- Introduction
- Selection sort
- Insertion sort

Insertion sort

3	44	38	5	47	1	36	26

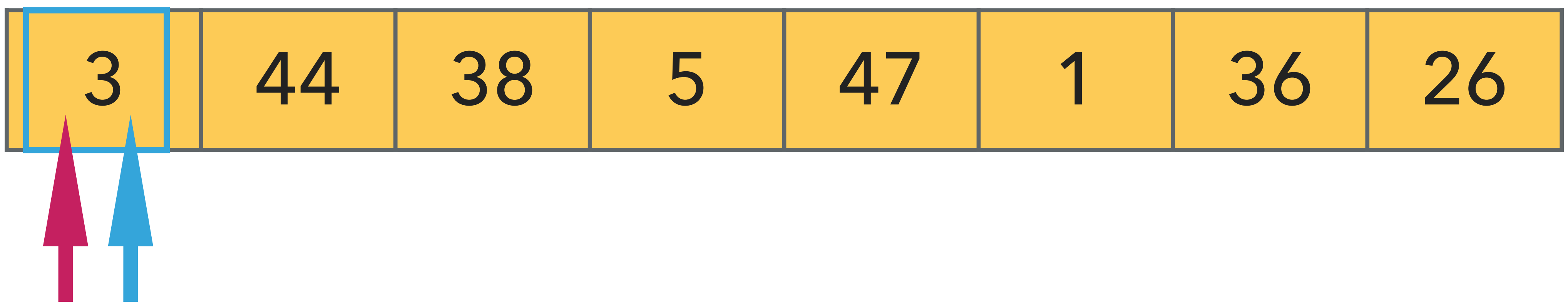
- Keep a *partially* sorted subarray on the left and an unsorted subarray on the right
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there. (exchange with larger entry to the left)
  - Move subarray boundaries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



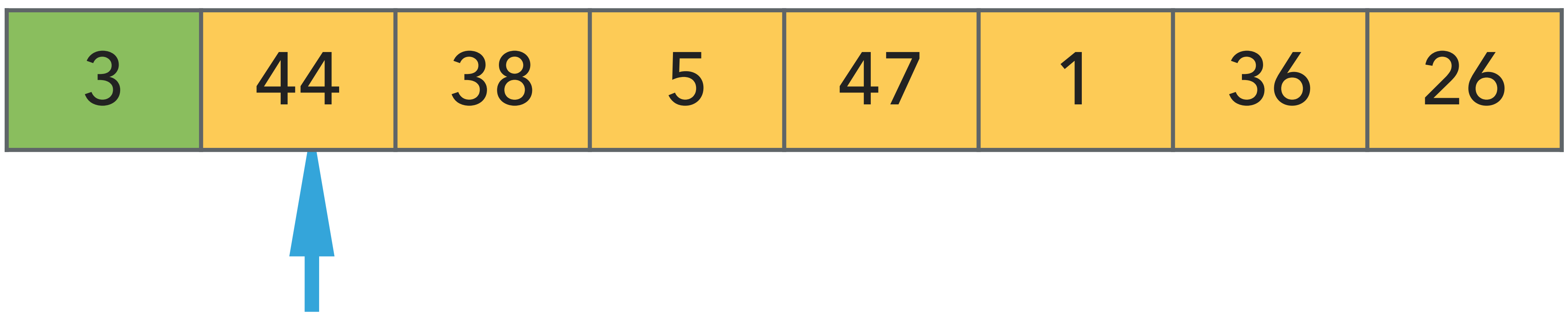
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	44	38	5	47	1	36	26

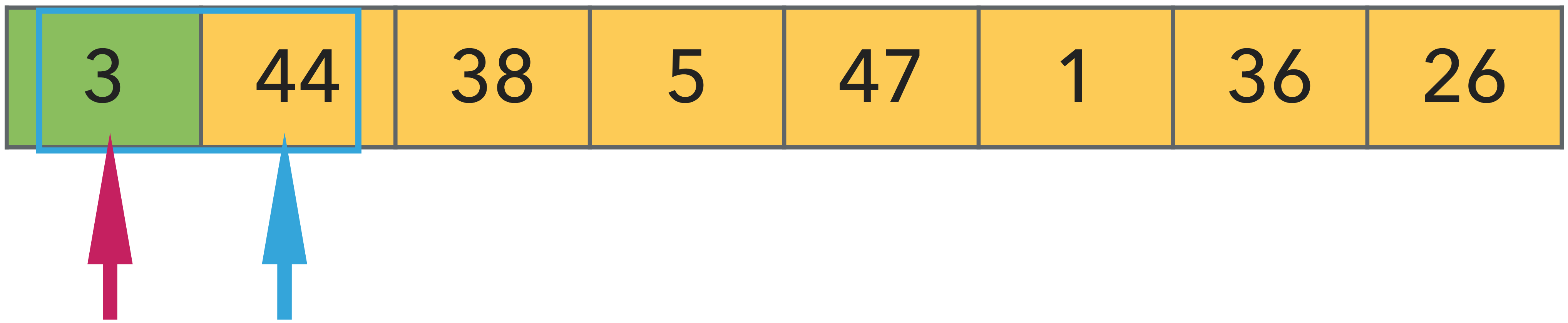
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



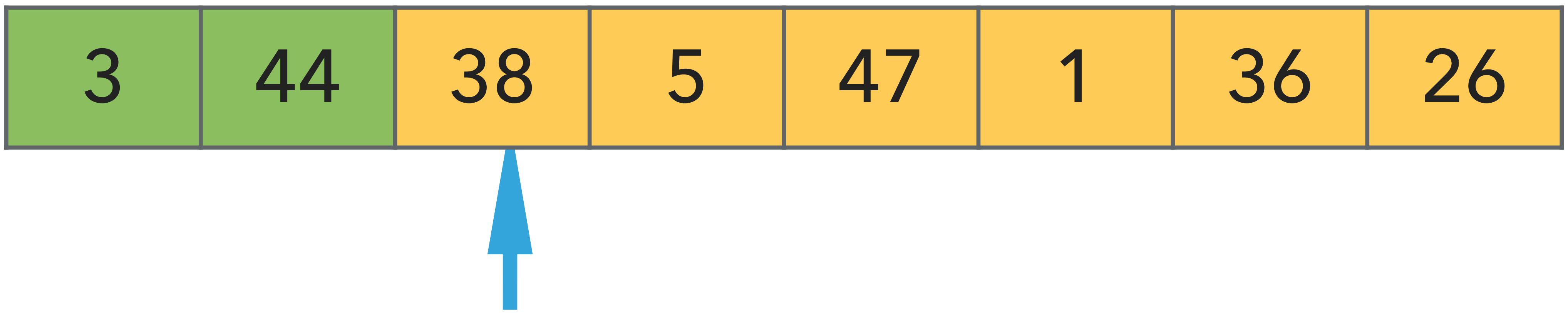
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	44	38	5	47	1	36	26

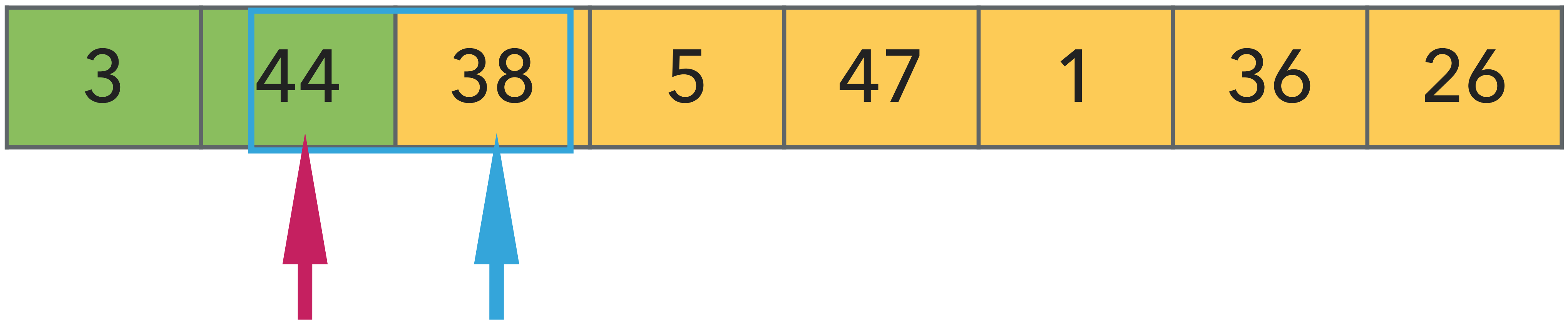
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



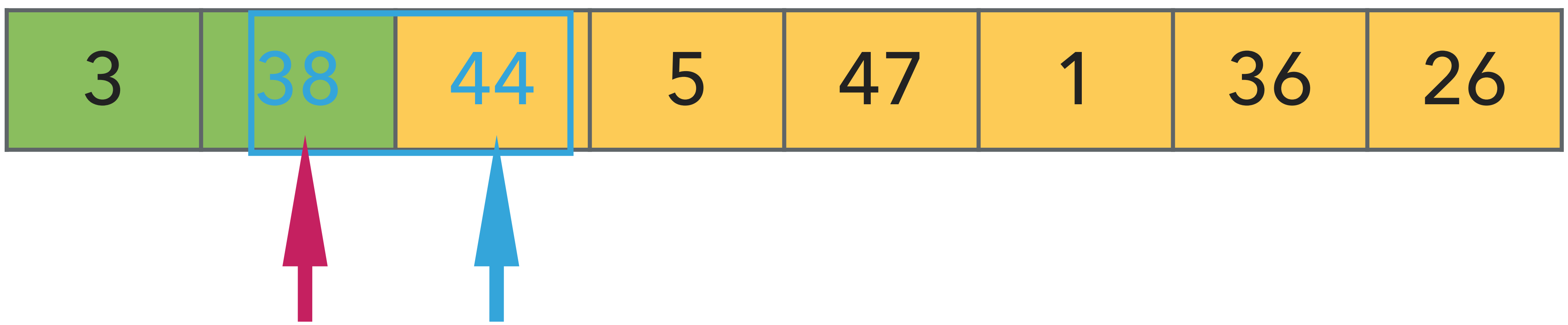
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



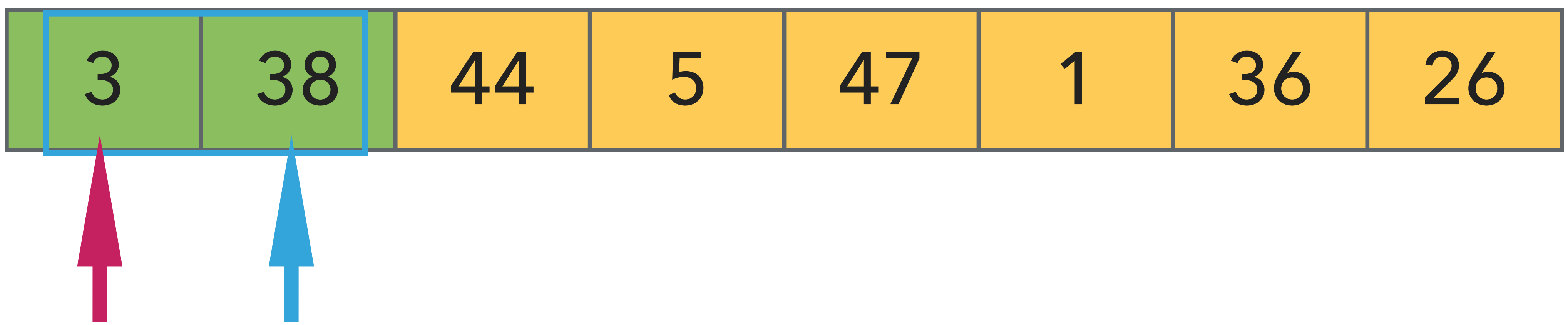
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



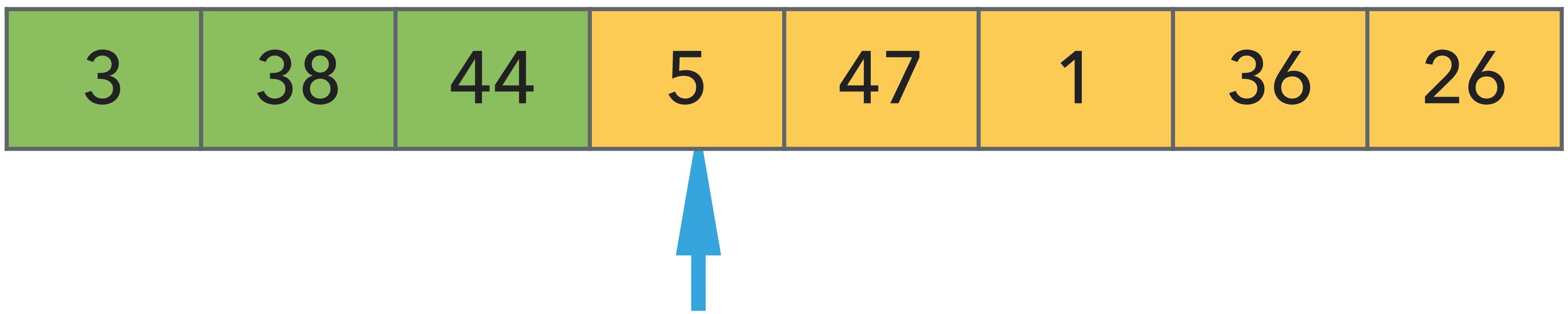
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	38	44	5	47	1	36	26

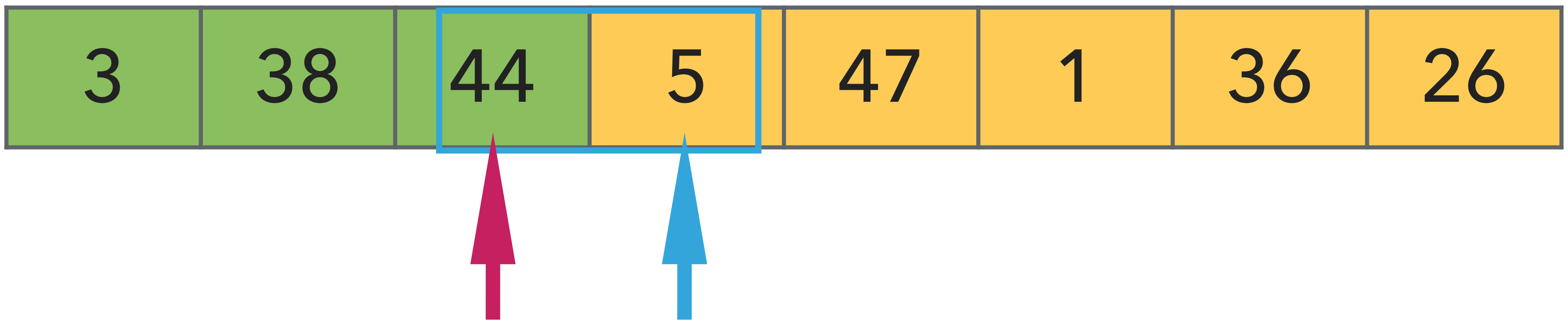
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



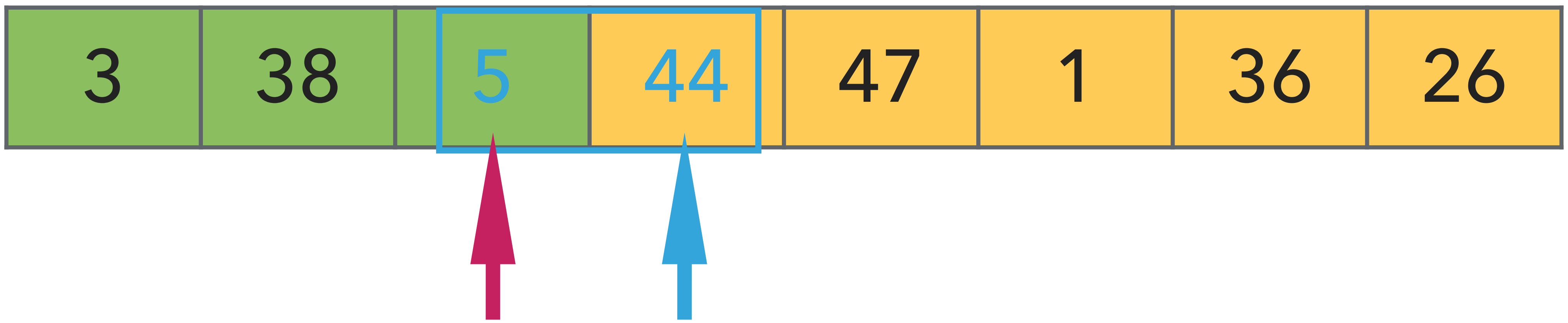
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



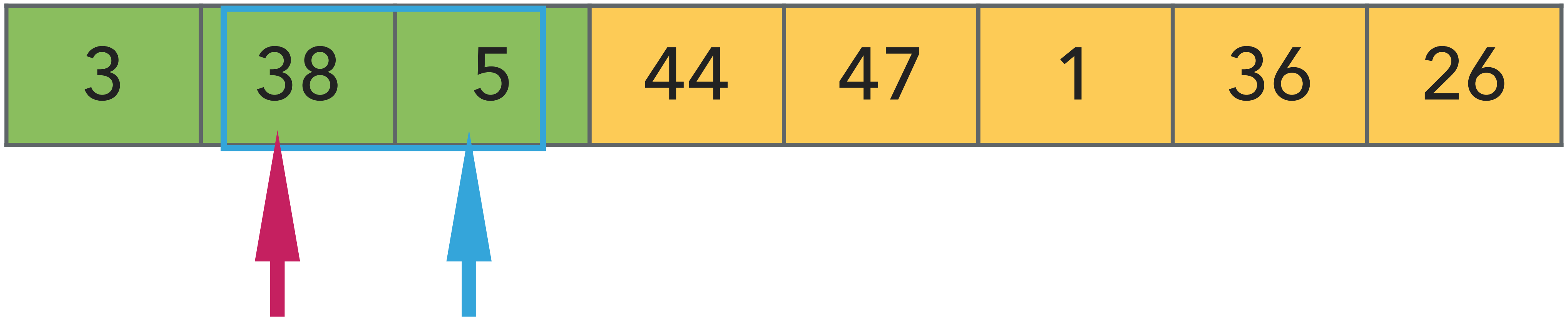
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



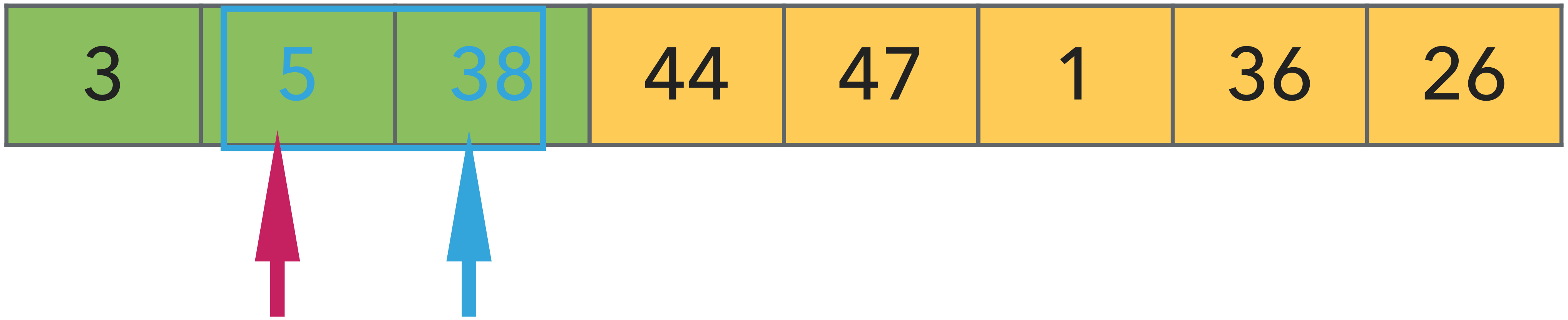
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



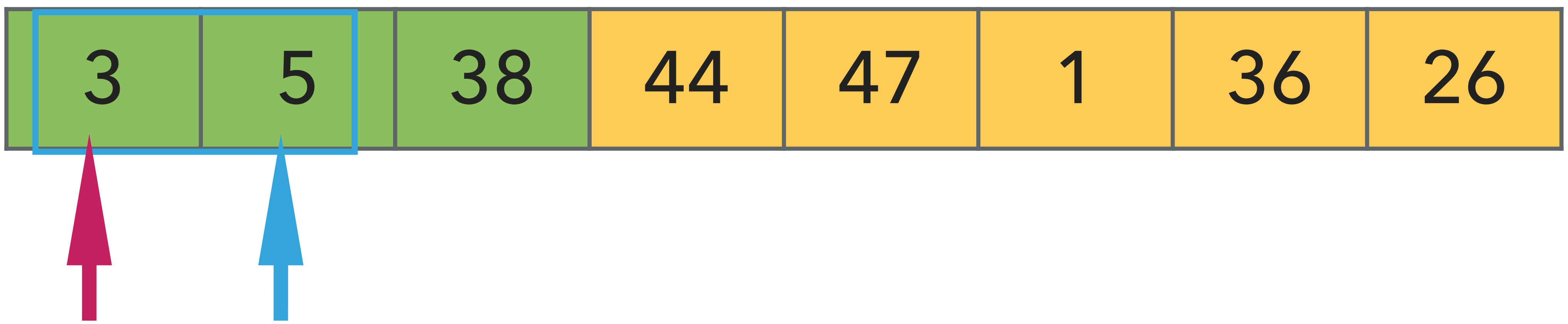
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



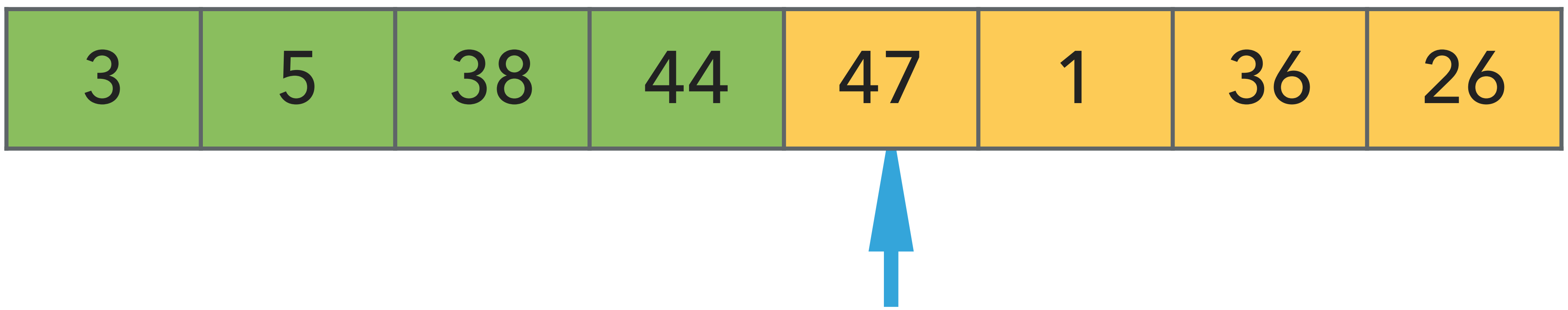
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	5	38	44	47	1	36	26

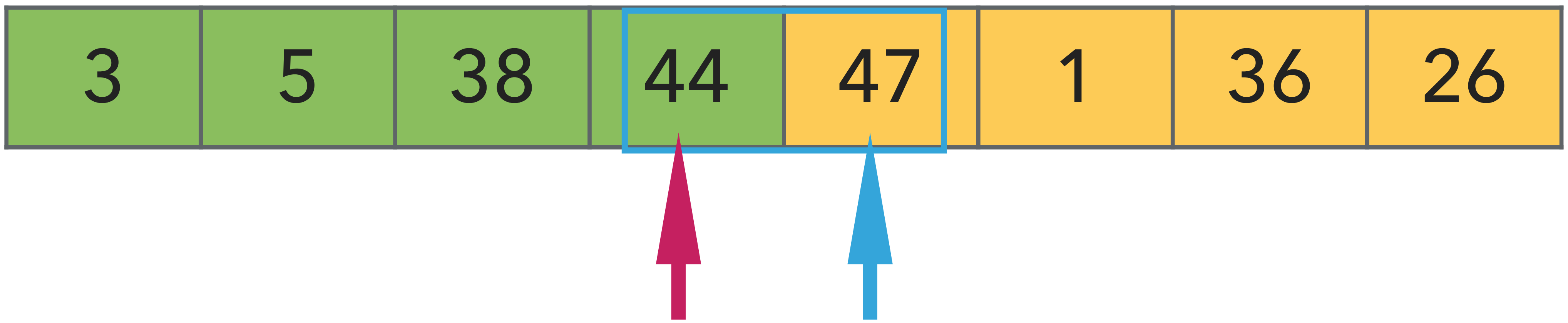
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



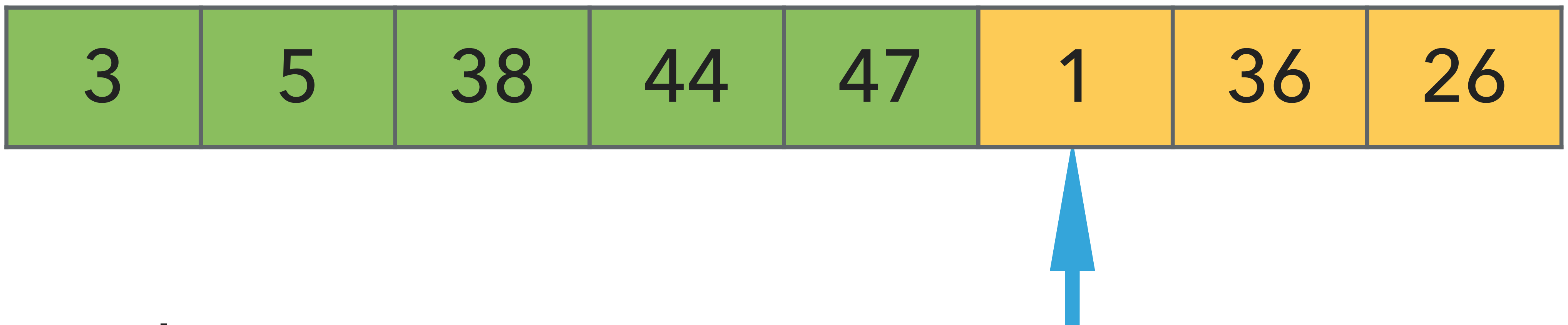
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

3	5	38	44	47	1	36	26

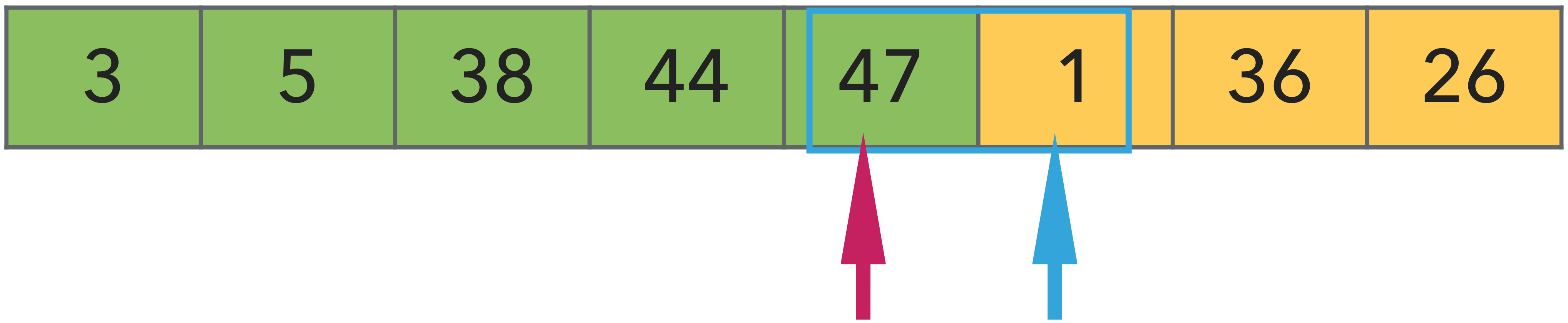
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



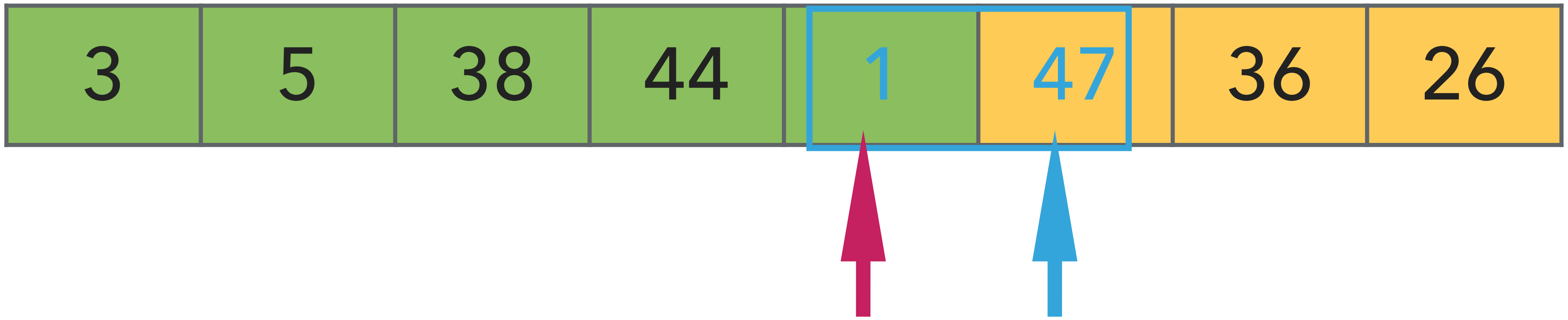
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



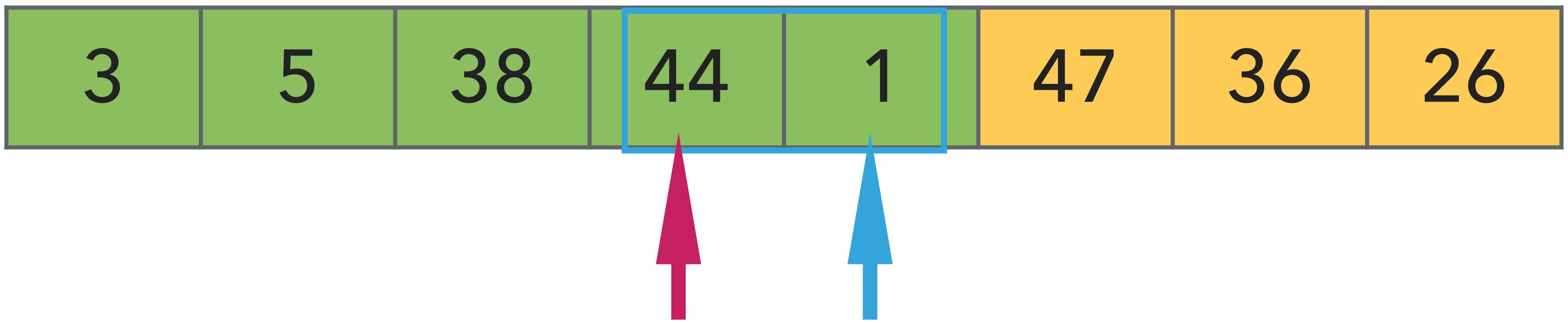
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



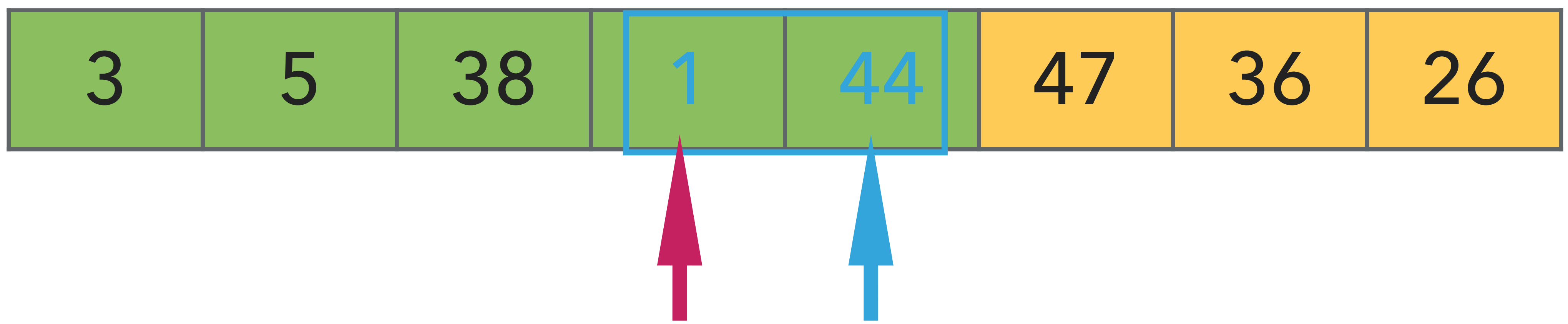
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



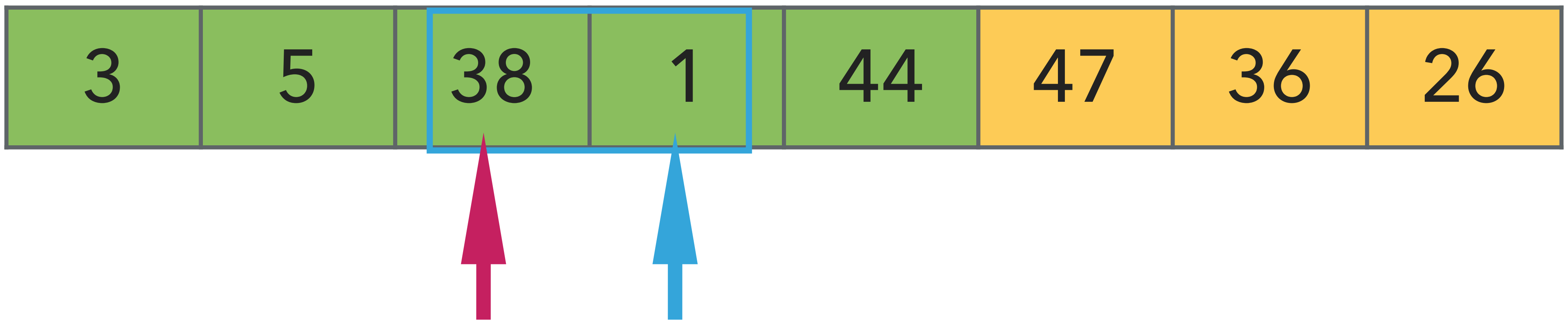
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



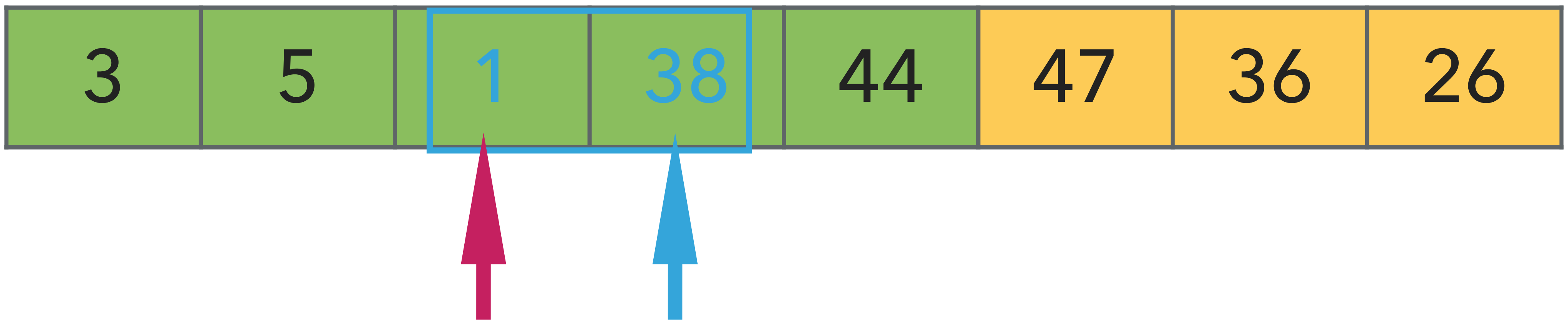
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



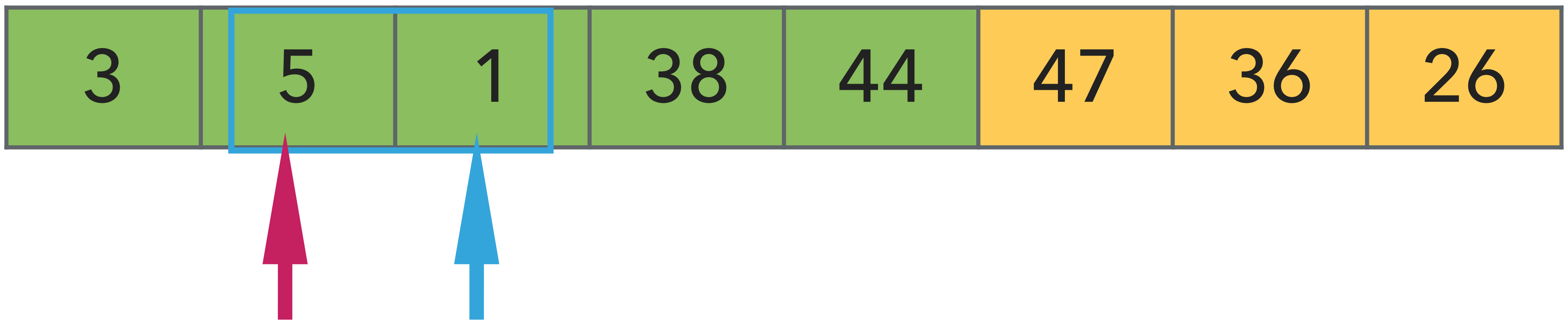
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



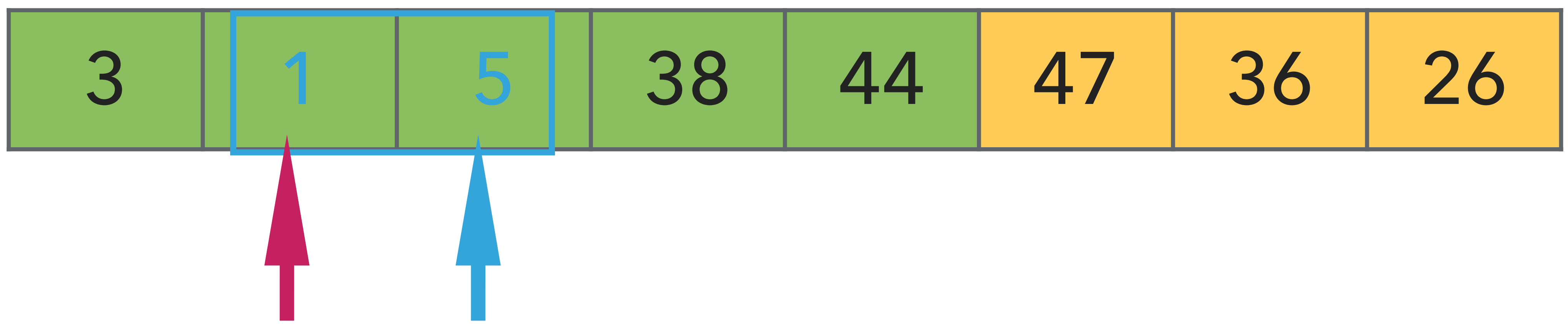
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



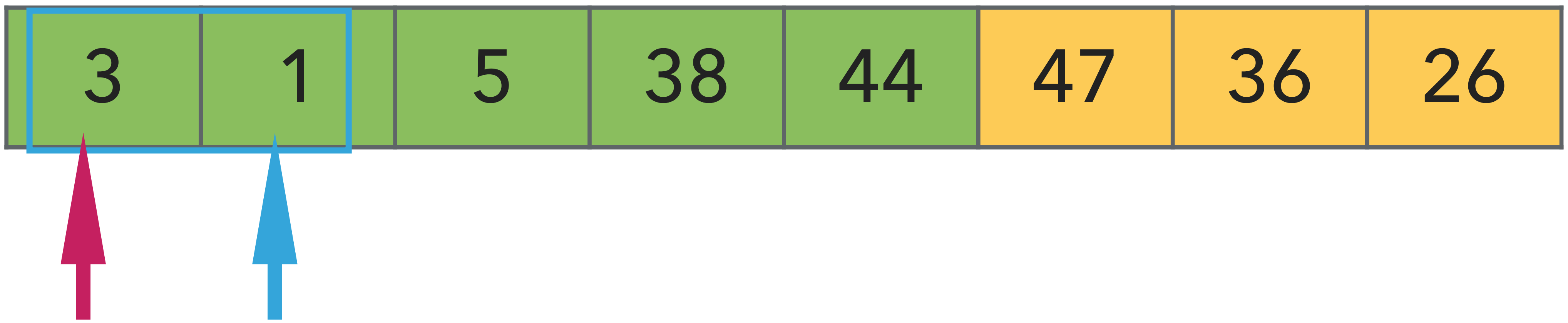
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



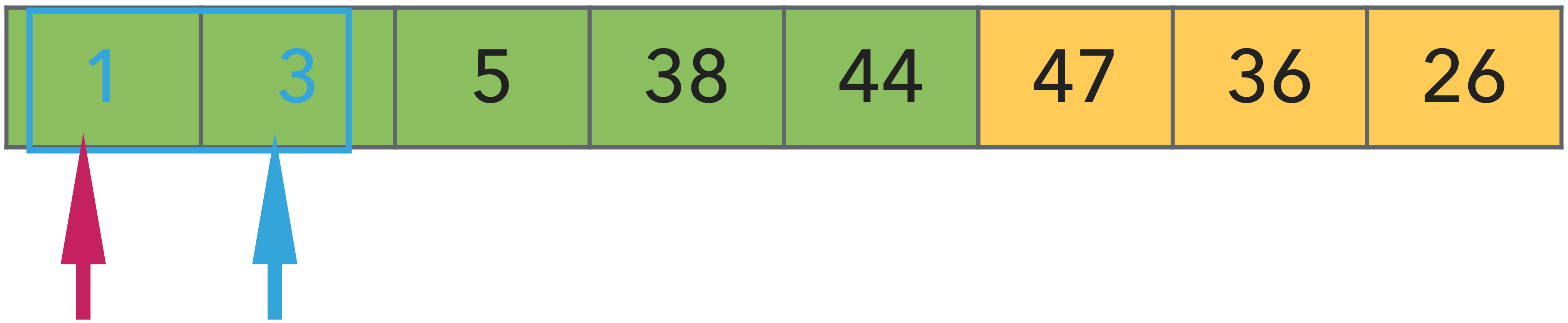
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



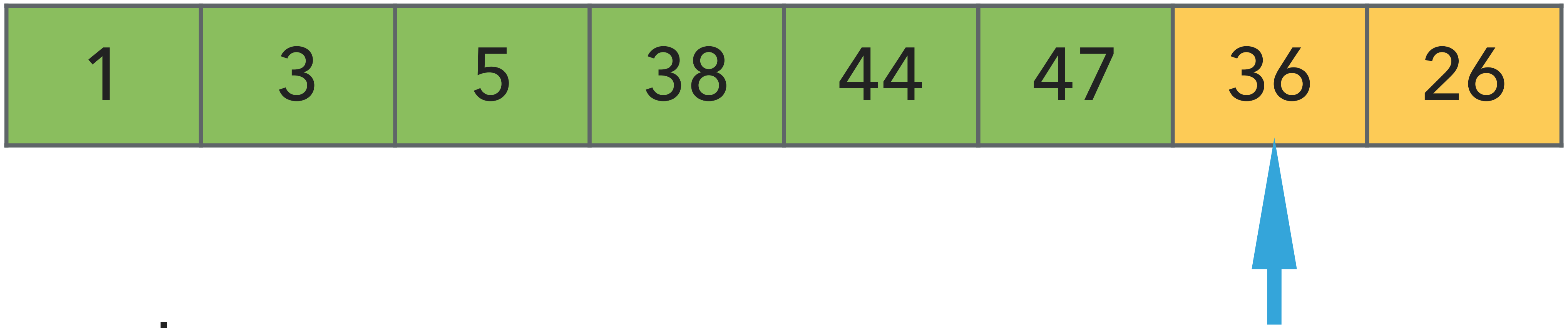
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

1	3	5	38	44	47	36	26

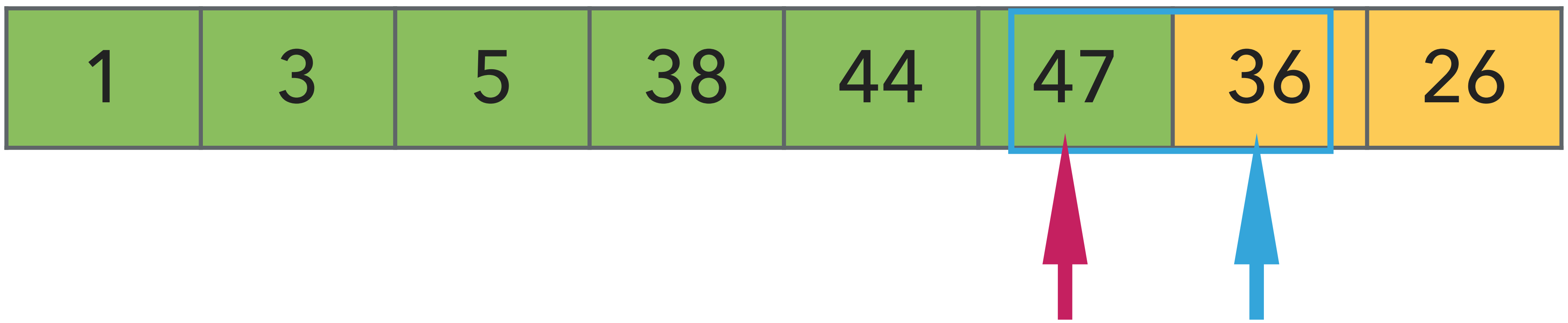
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



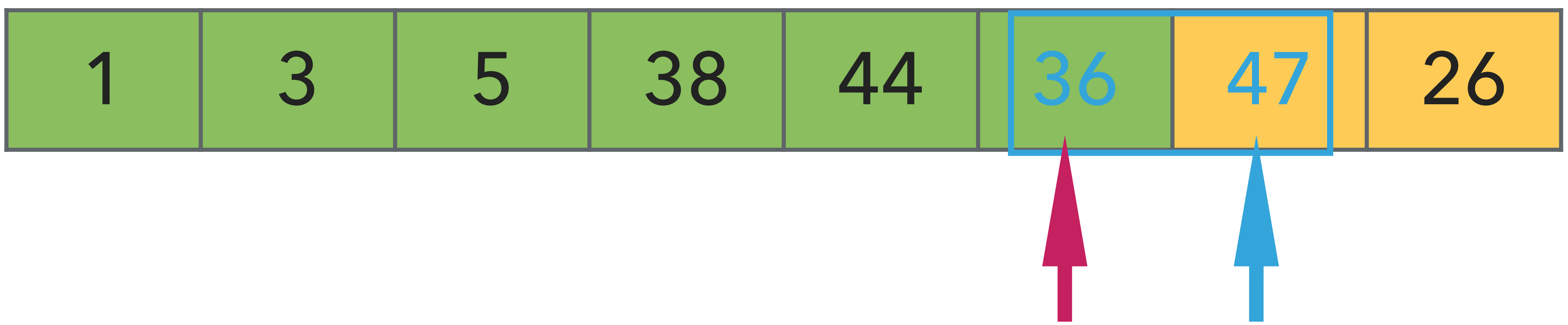
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



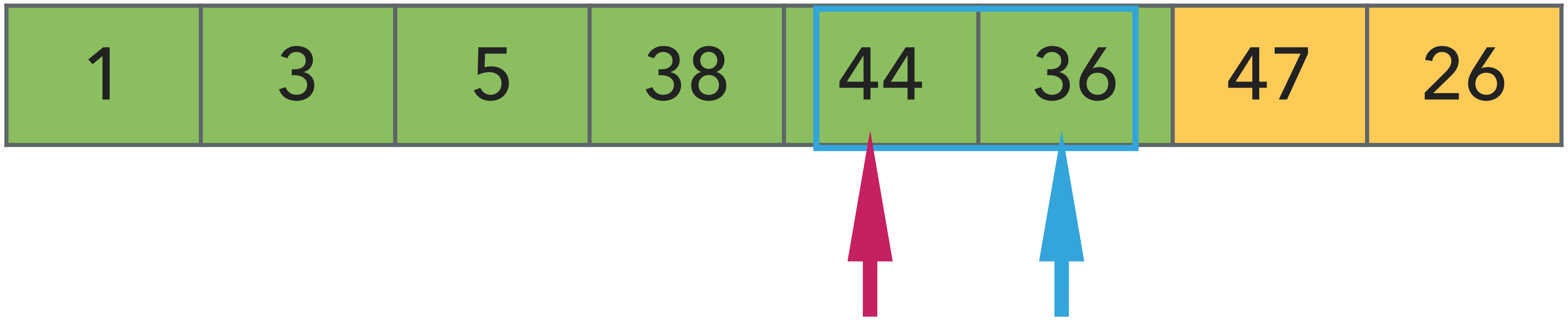
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



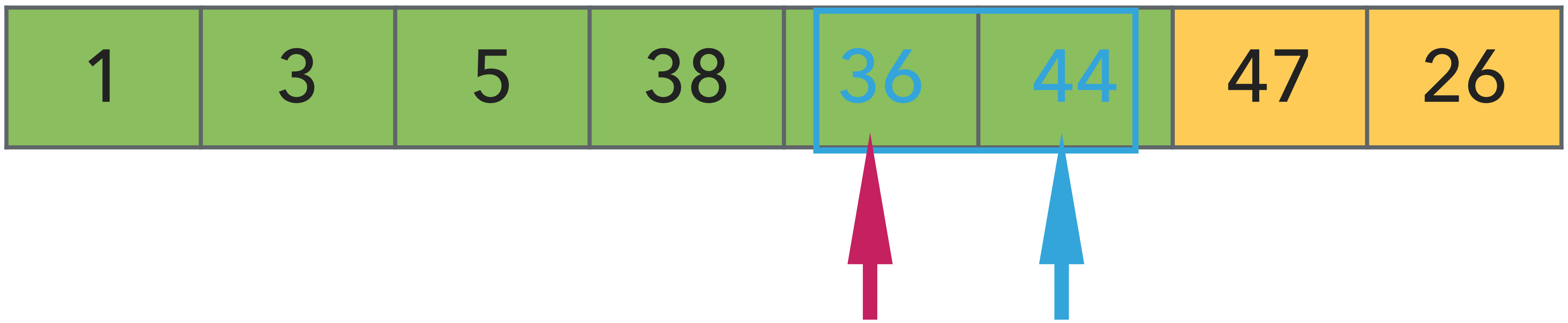
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



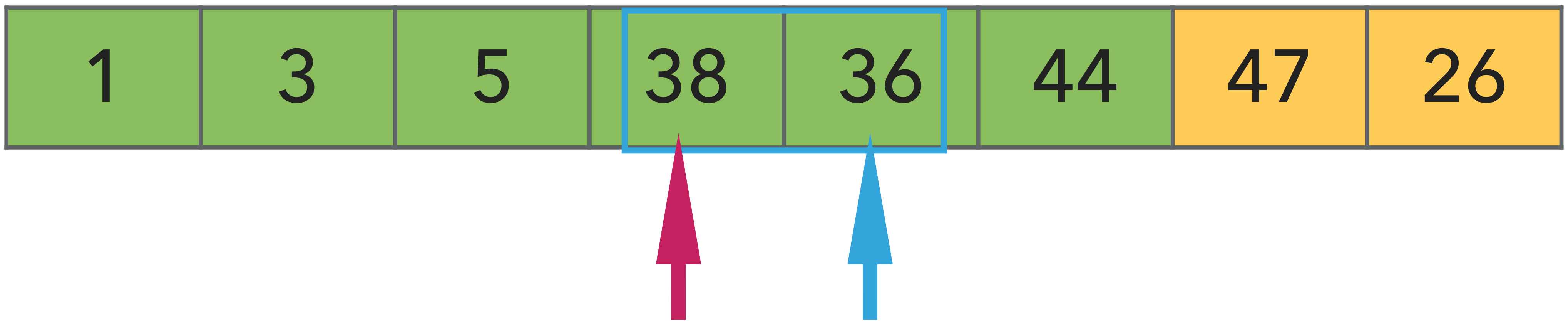
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



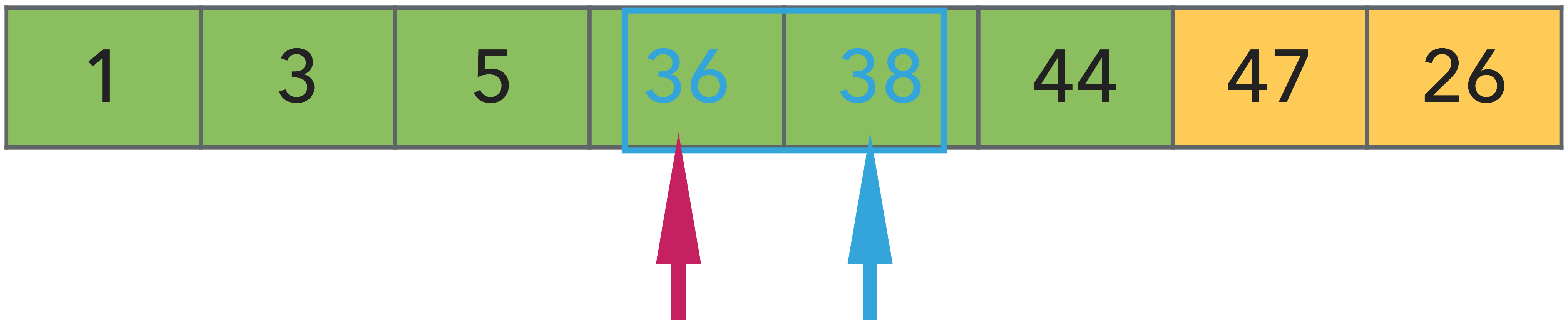
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



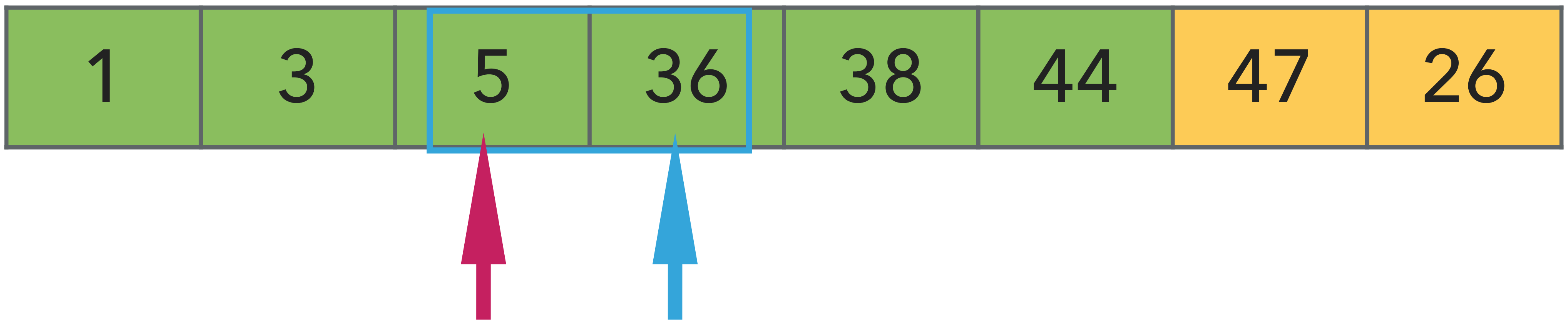
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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



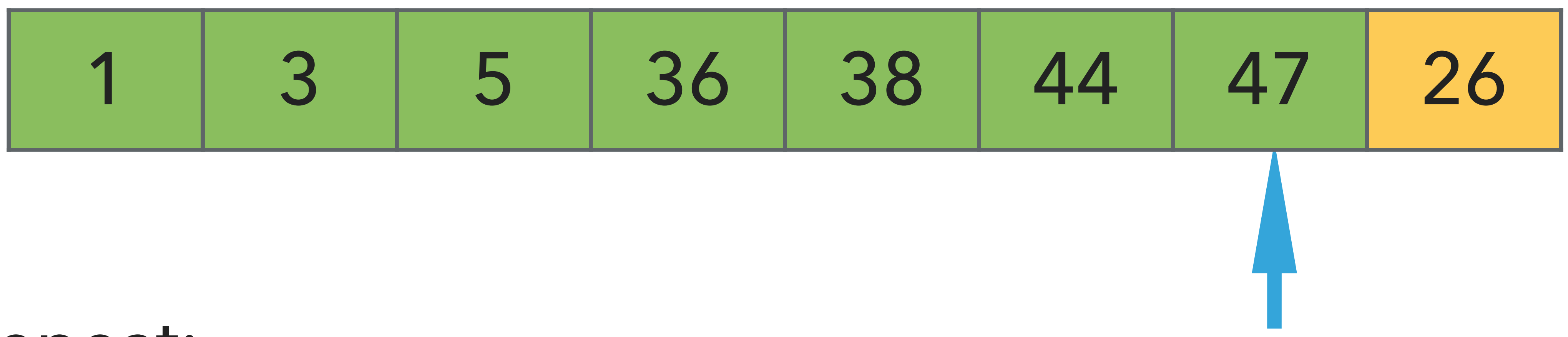
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

1	3	5	36	38	44	47	26

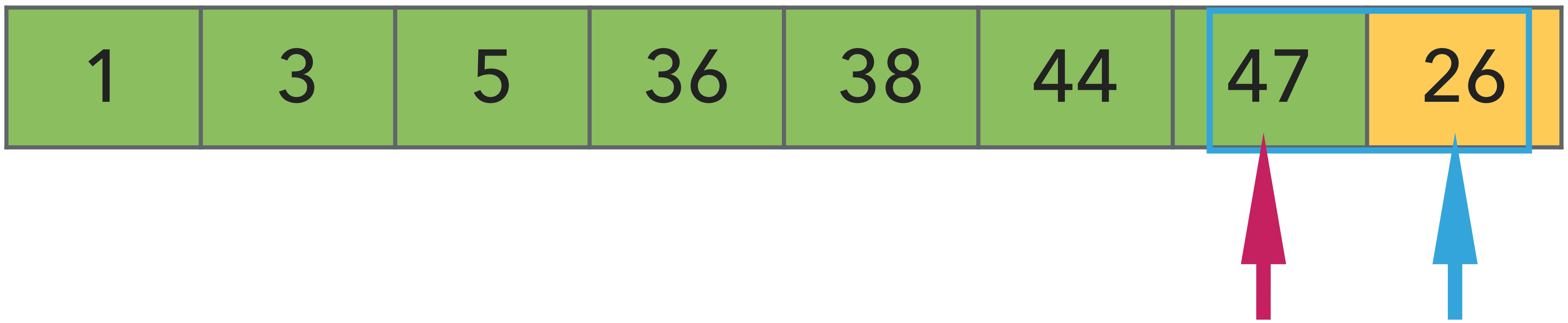
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



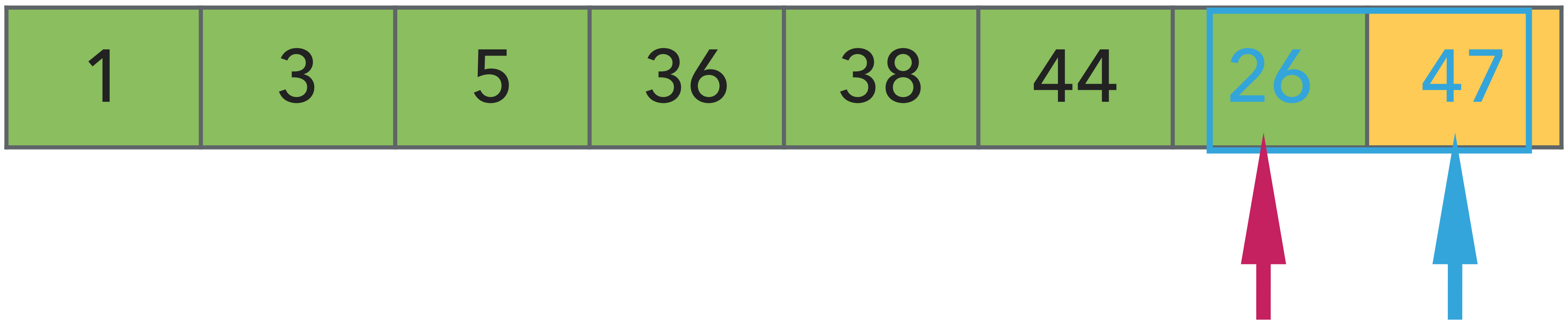
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



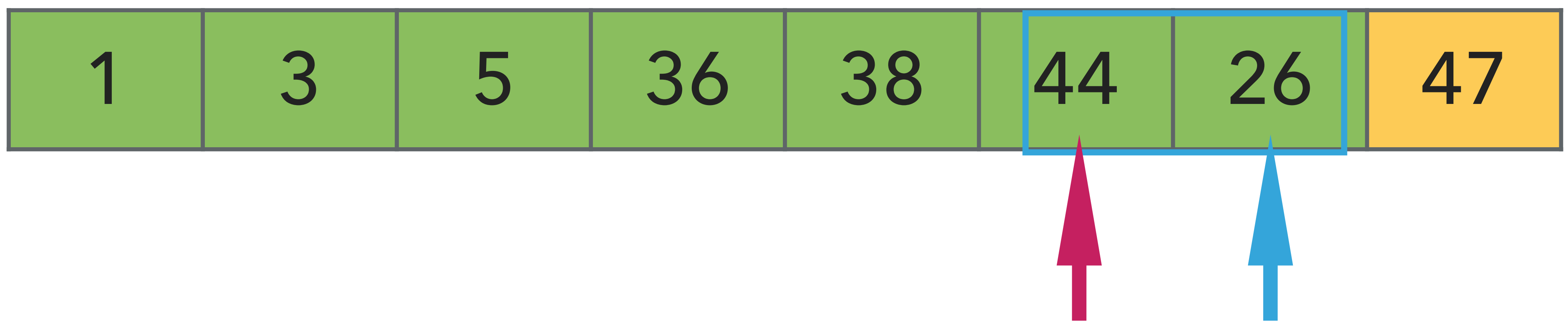
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



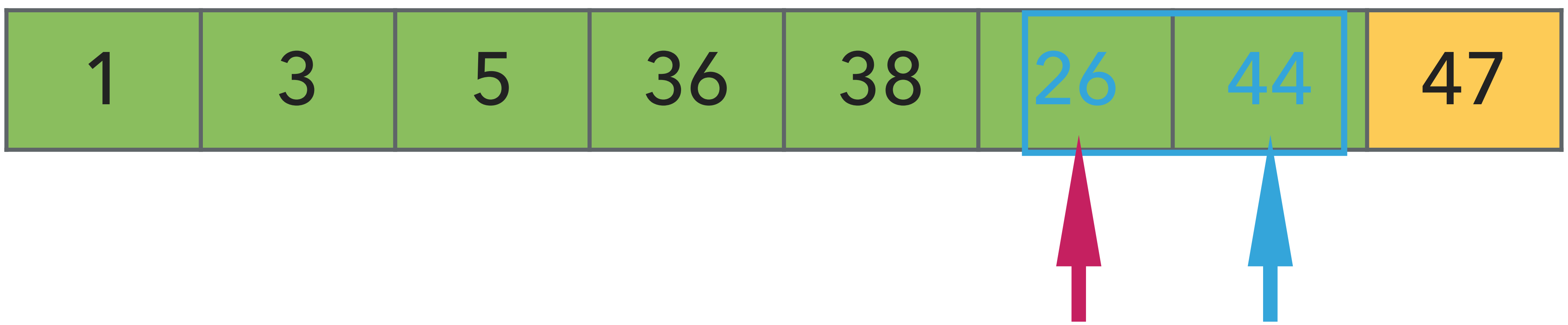
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



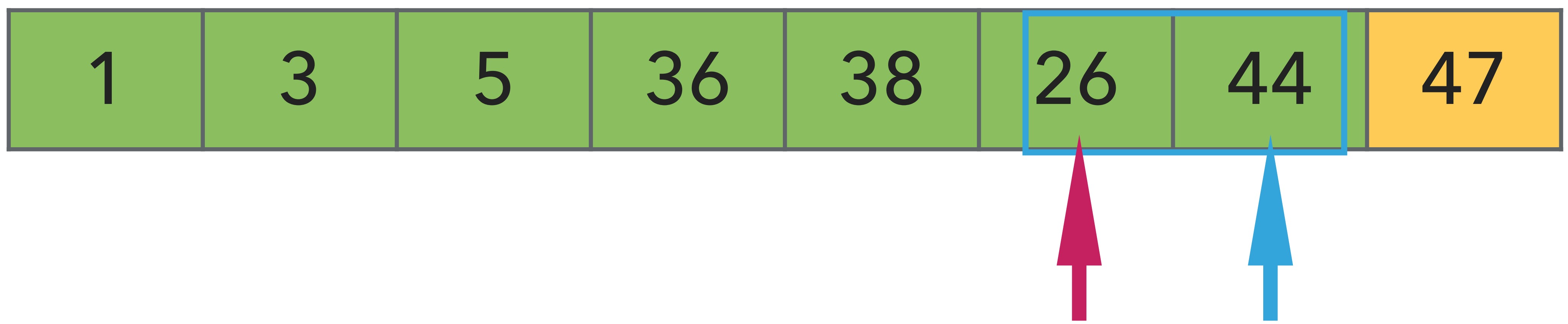
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



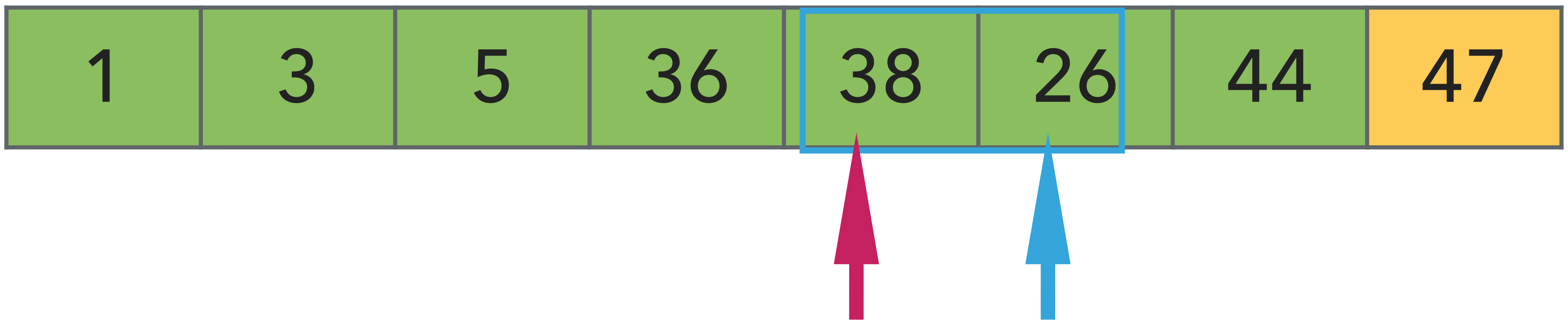
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



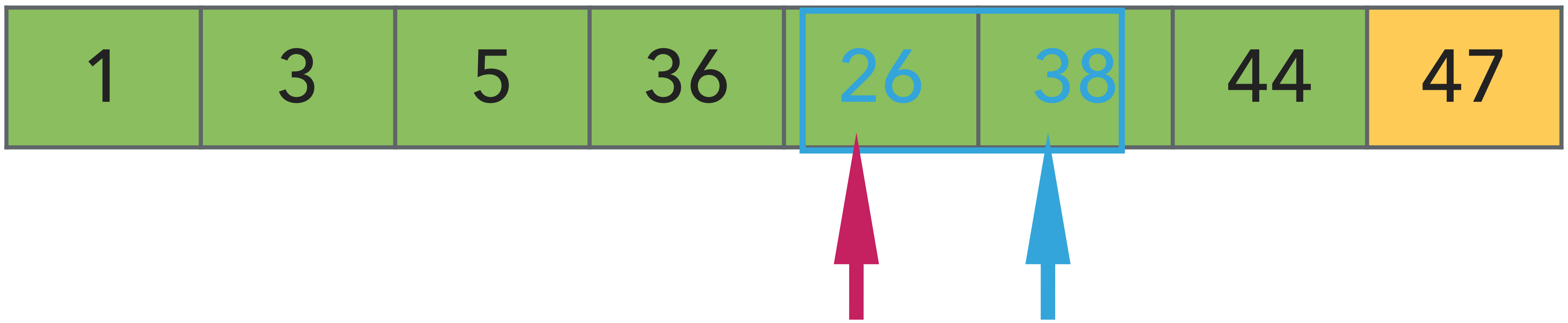
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



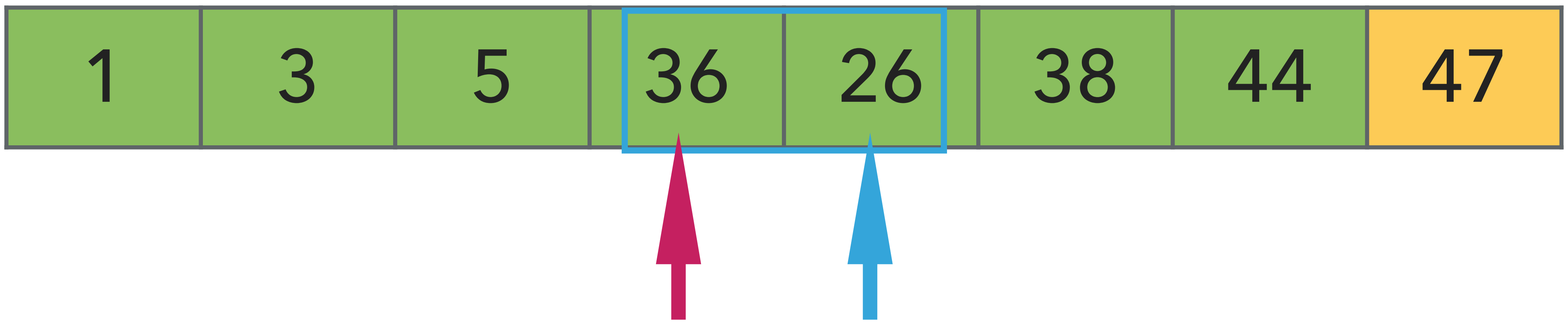
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



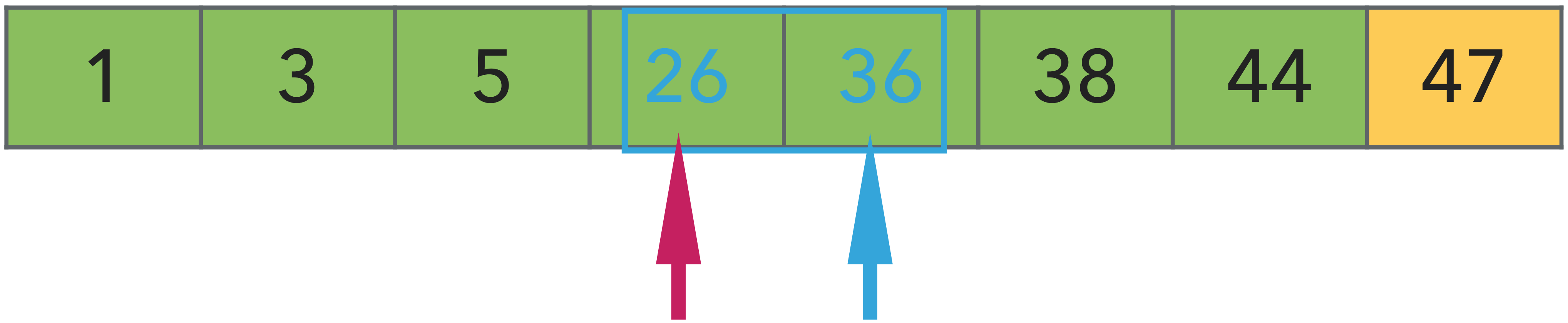
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



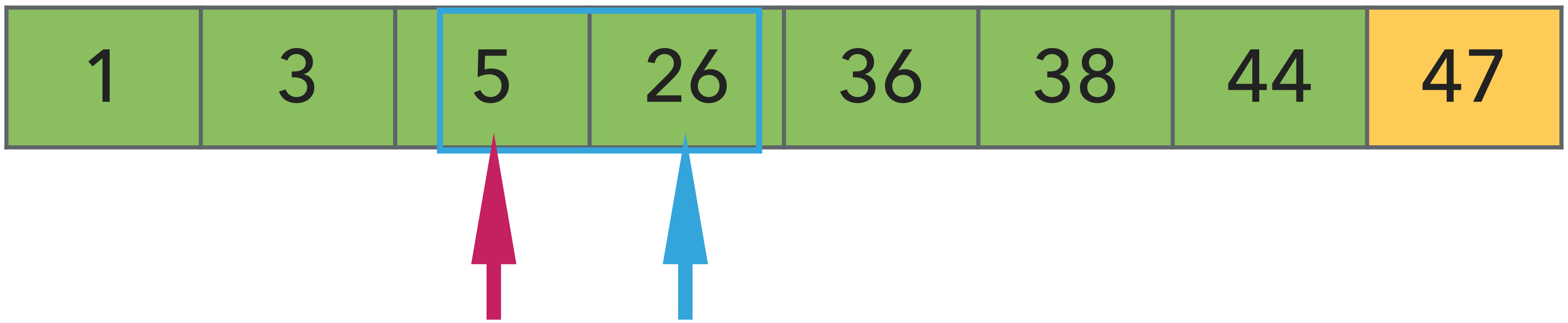
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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

In case you didn't get this…

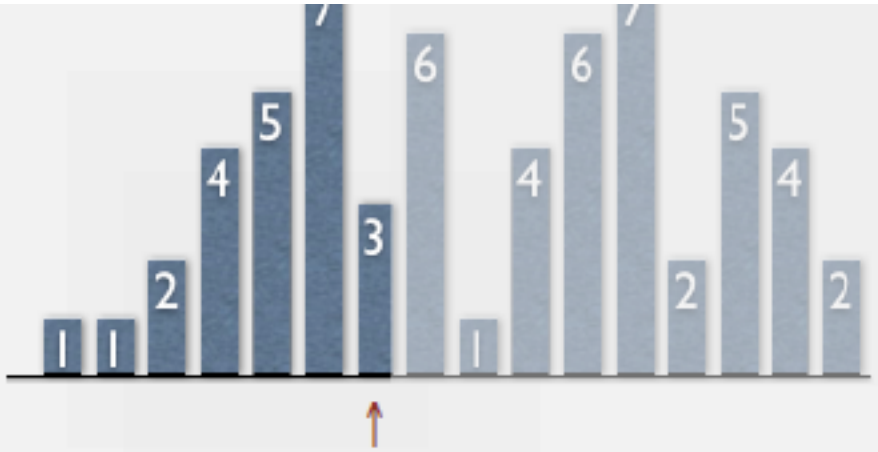
- https://www.youtube.com/watch?v=ROalU379l3U

Insertion sort

```
public static void sort(Comparable[] a) {
```

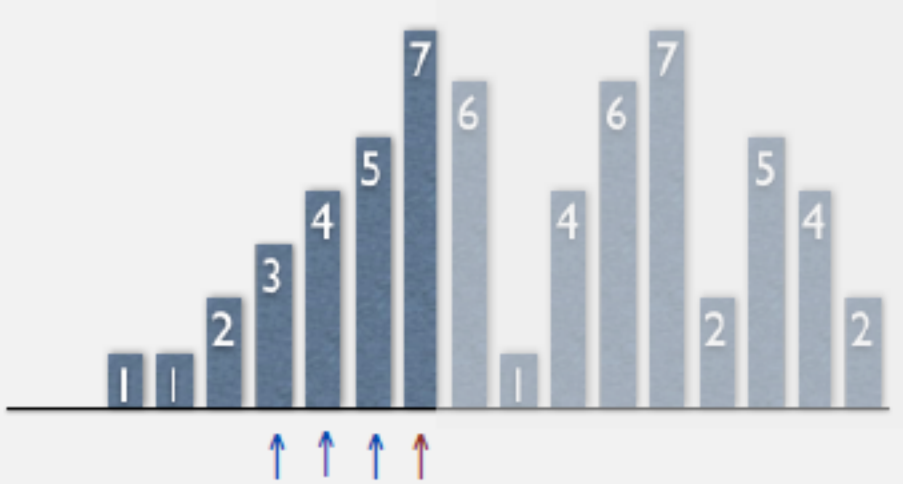
- Move the pointer to the right.

```
i++;
```



- Moving from right to left, exchange `a[i]` with each larger entry to its left.

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```




```
}
```

Insertion sort

```
public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(less(a[j], a[j-1]))
                  exch(a, j, j-1);
               else
                  break;
           }
       }
   }
```

← In iteration i

← Move from right to left, exchange a[i] with entry to the left, if it's larger

- 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 void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(less(a[j], a[j-1]))
                  exch(a, j, j-1);
               else
                  break;
           }
       }
   }
```

- Comparisons: $0 + 1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, that is $O(n^2)$.
- Exchanges: ?
- In-place?
- Stable?

Insertion sort: mathematical analysis for worst-case

```
public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(less(a[j], a[j-1]))
                  exch(a, j, j-1);
               else
                  break;
           }
       }
   }
```

- Comparisons: $0 + 1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, that is $O(n^2)$.
- Exchanges: $0 + 1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, that is $O(n^2)$.
- Worst-case running time is quadratic. Worst case = array sorted in reverse order.
- Every element moves all the way to the left.
- In-place, requires almost no additional memory.
- Stable

Insertion sort: average and best case

```
public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(less(a[j], a[j-1]))
                  exch(a, j, j-1);
               else
                  break;
           }
        }

    }
```

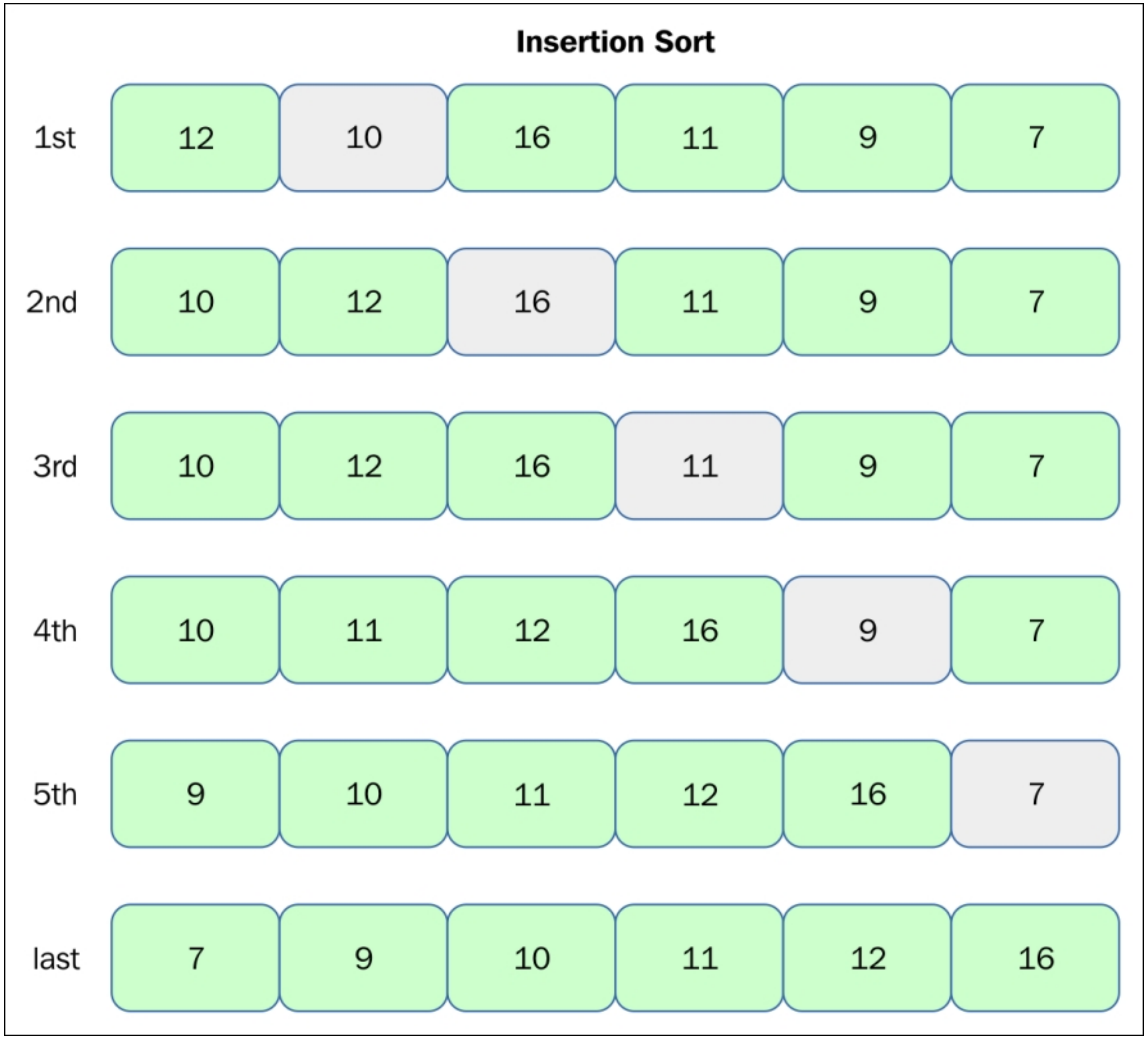
- Average case: quadratic for both comparisons and exchanges $\sim n^2/4$ when sorting a randomly ordered array. (2X faster than selection sort on average)
    - Expect each entry to move halfway back: 0 + 0.5 + 1 +...(n-1)/2 ~ (n/2)*(n/2) ~ n^2/4
- Best case: $n - 1$ comparisons (validate) and 0 exchanges for an already sorted array.

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

Practice Time (cards)

- Using insertion sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm.

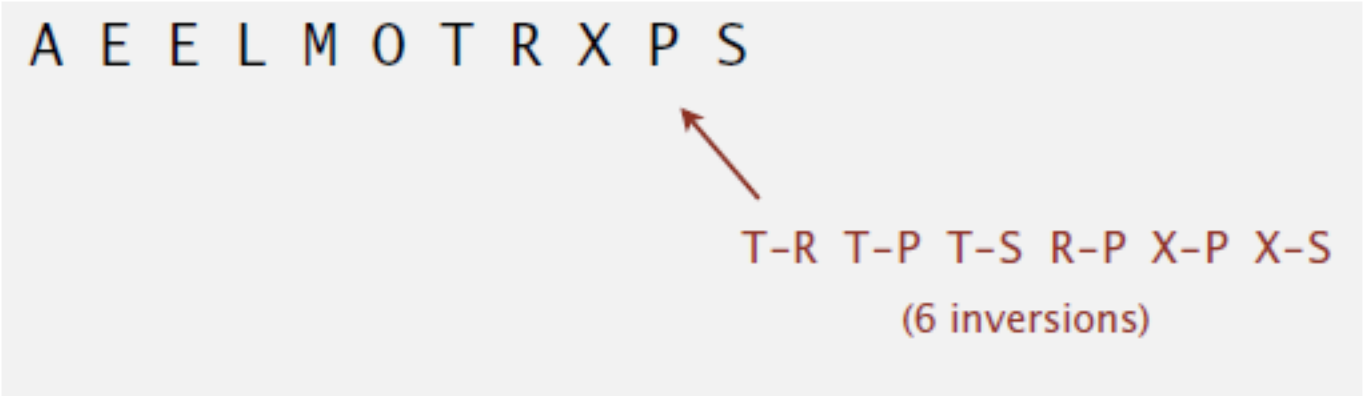
Answer



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

Insertion Sort

- For partially-sorted arrays, insertion sort runs in linear time
- Number of exchanges equals number of inversions
- Inversion = pair of keys that are out of order



- Ex1: Appending a subarray of size 10 to a sorted subarray of size N
- Ex2: An array of size N with only 10 entries out of place

Lecture 11: Sorting Fundamentals and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators

`Comparable`

- Interface with a single method that we need to implement: `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than `w`.
  - Returns <0 if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.
- Corresponds to natural ordering.

How to make your class T comparable?

1. Implement `Comparable<T>` interface.
2. Implement `compareTo(T that)` method to compare `this` T object to `that` based on natural ordering.

Comparator

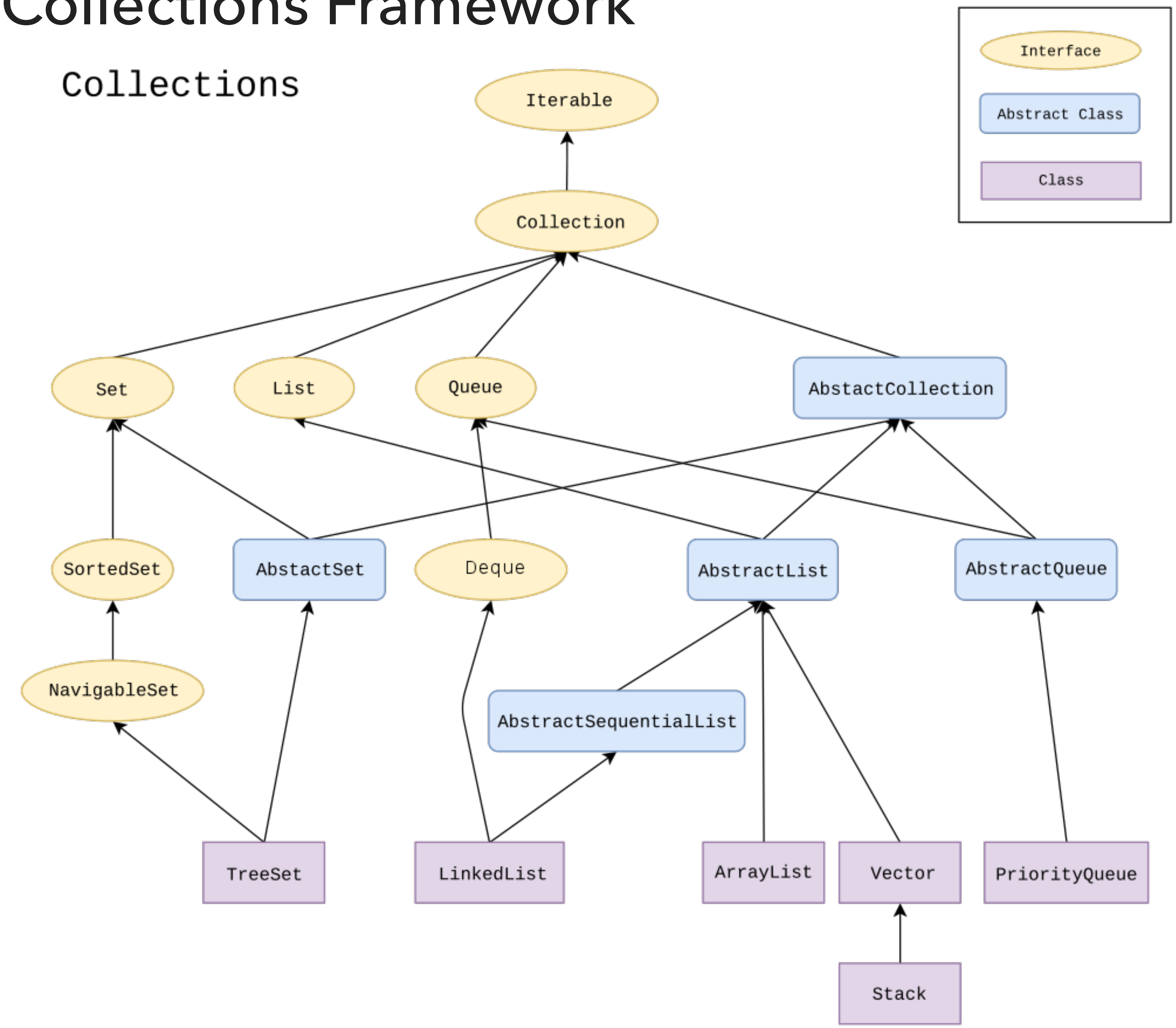
- Sometimes the natural ordering is **not** the type of ordering we want.
- Comparator is an interface which allows us to **dictate what kind of ordering** we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that `compare(v, w)`:
  - Returns >0 if v is greater than `w`.
  - Returns <0 if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.

How to define an alternative ordering for your class T?

1. Make a new class that implements `Comparator<T>` interface.
2. Implement `compare(T t1, T t2)` method to compare t1 object to t2 based on an alternative ordering.
3. Alternatively, implement an anonymous inner class:

```
public static Comparator<T> nameOfComparator = new Comparator<T>()
{
    @Override      // indicates method overriding the superclass' method
    public int compare(T t1, T t2) {
     {
           //return something;
    }
};
```

The Java Collections Framework



https://en.wikipedia.org/wiki/Java_collections_framework

Sorting `Collections`

- Collections class contains:
  - `public static <T extends Comparable<? super T>> void sort(List<T> list)`
  - Generic methods introduce their own type parameters.
  - Use `extends` with generics, even if the type parameter implements an interface.
- The class `T` itself or one of its ancestors implements `Comparable`.
- `Collections.sort(list)`
  - Implemented as optimized mergesort, that is timsort.
  - If list's elements do not implement `Comparable`, throw `ClassCastException`.

Alternative sorting of `Collections`

- Collections class contains:
  - `static <T> void sort(List<T> list, Comparator<? super T> c)`
- `Collections.sort(list, someComparator);`
  - `Collections.sort(list, new ExternalComparatorClass());` or:
  - `Collections.sort(list, T.InnerAnonymousClass);`
  - If list's elements do not implement `Comparable` or cannot be compared with `Comparator`, throw `ClassCastException`.

Example: Natural and alternative sorting for `Employees`

https://github.com/pomonacs622021fa/LectureCode/blob/main/Lecture11/Employee.java

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator

Lecture 11: Sorting Fundamentals and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators

Readings:

- Textbook:
  - Chapter 2.1 (pages 244–262), Chapter 2.1 (Page 247), Chapter 2.5 (Pages 338-339)
- Website:
  - Elementary sorts: https://algs4.cs.princeton.edu/21elementary/
  - Code: https://algs4.cs.princeton.edu/21elementary/Selection.java.html and https://algs4.cs.princeton.edu/21elementary/Insertion.java.html
- Oracle documentation:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Comparable: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
  - Comparator: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Practice Problems:

- 2.1.1-2.1.8