SORTING

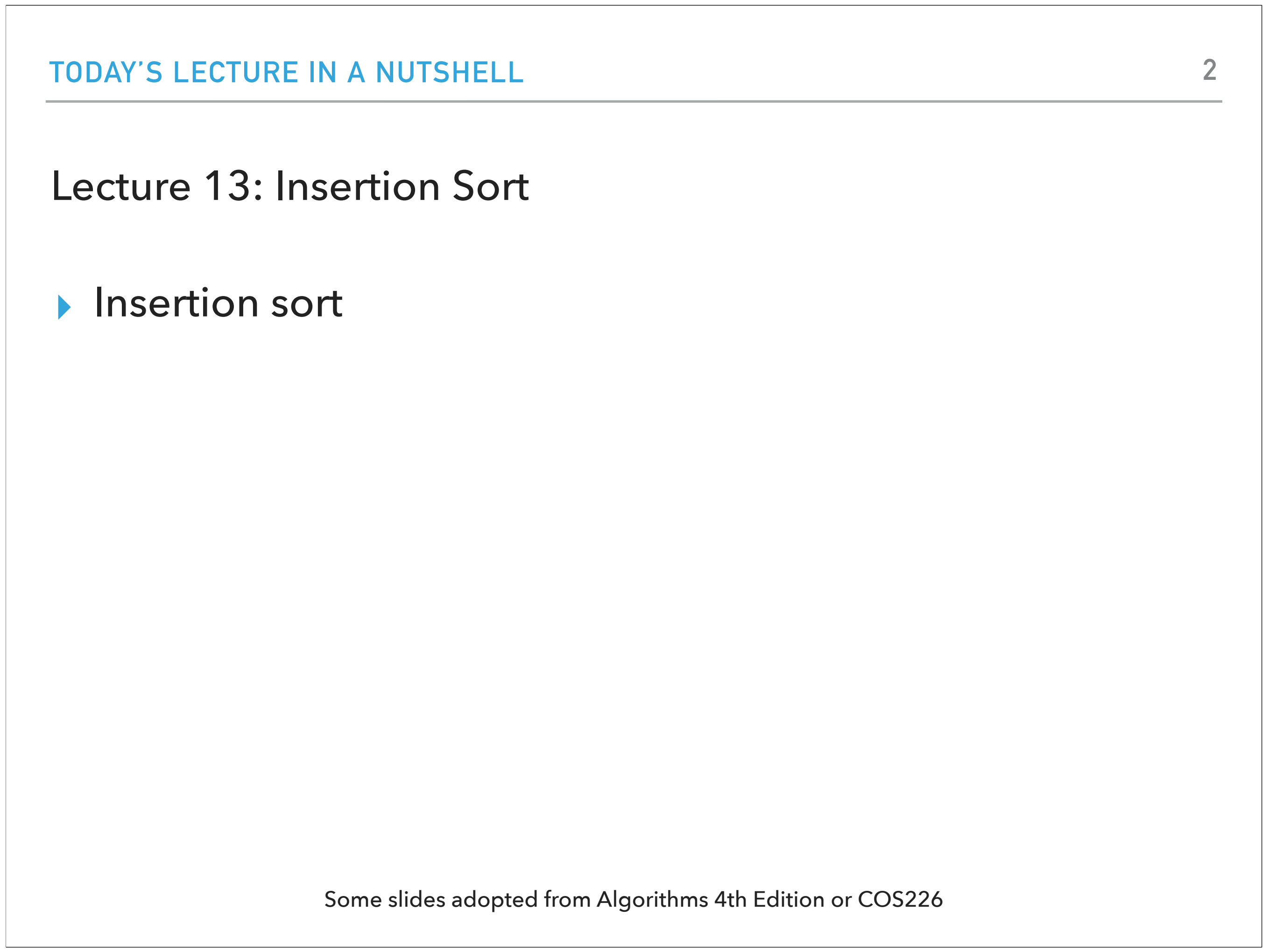
CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

13: Insertion Sort

Alexandra Papoutsaki
she/her/hers

Today we'll continue with our second sorting algorithm, insertion sort.

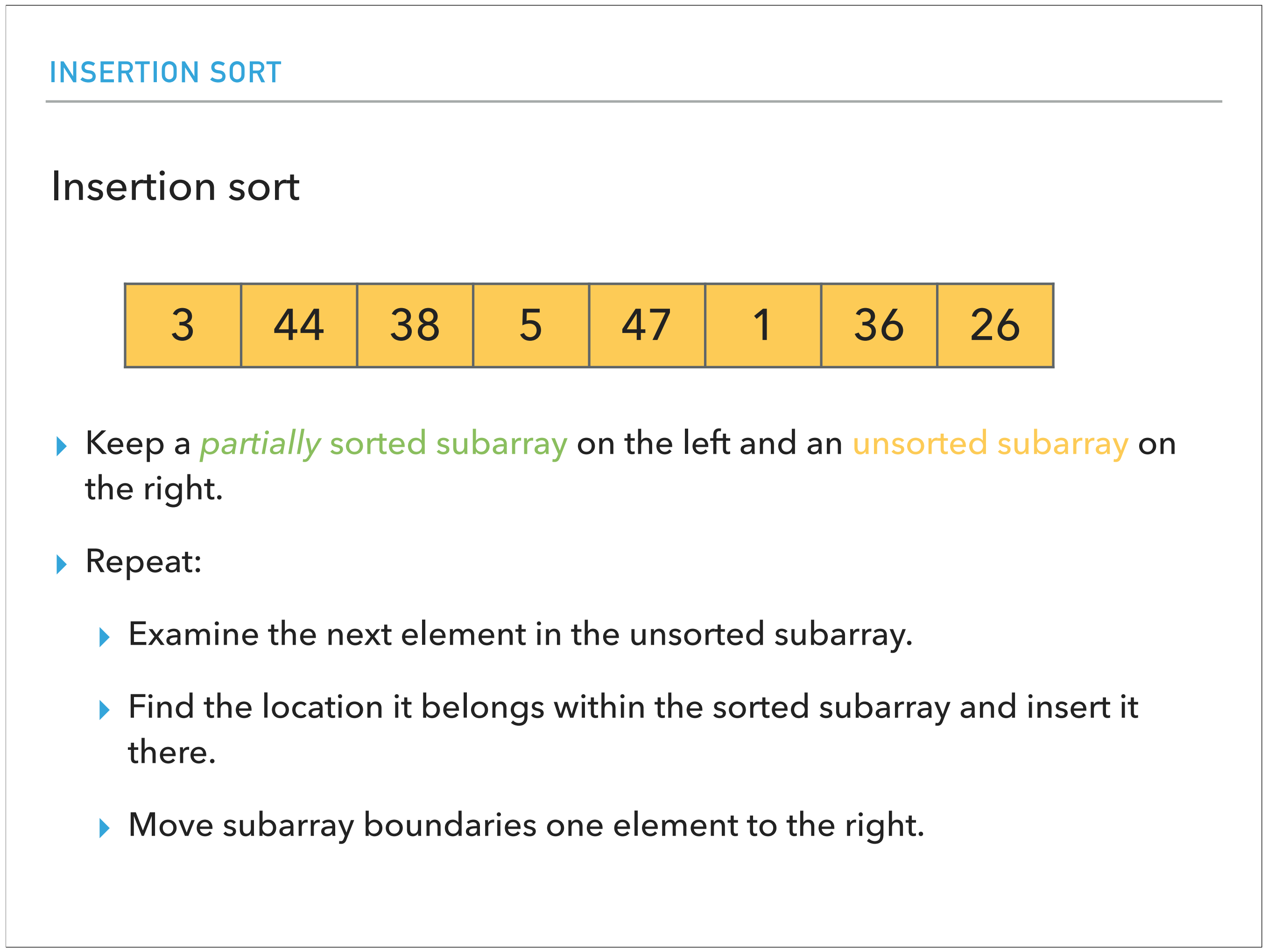


Lecture 13: Insertion Sort

- Insertion sort

Some slides adopted from Algorithms 4th Edition or COS226

If I were to hand you a few playing cards and ask you to sort them, chances are you would come up on your own with insertion sort.



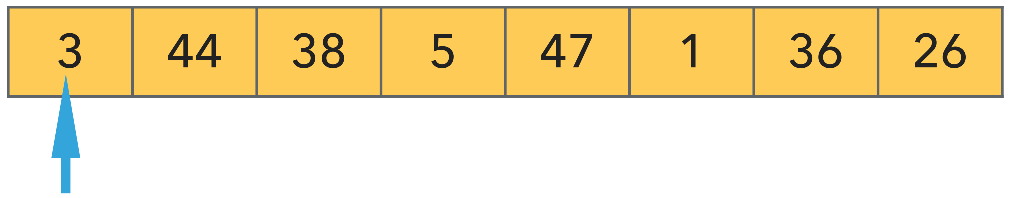
Insertion sort

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's basic idea is that we have a partially sorted subarray on the left (think of it as your left hand) and an unsorted subarray on the right (your right hand). We get to repeat the following steps: 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.
To help with the illustration of how this algorithm works, I have an array of eight numbers. we start with everything marked as yellow since the entire array is unsorted. as we run the algorithm, the left side will become progressively greener since we will be partially sorting it.

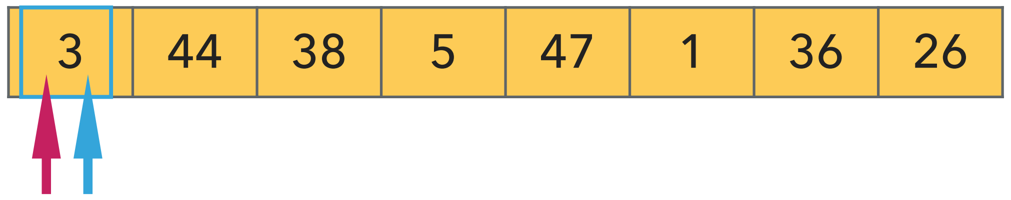
Insertion sort



- 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.

We start with the first element in the array (and unsorted subarray).

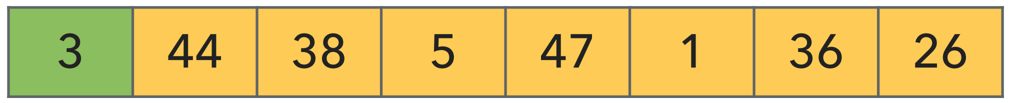
Insertion sort



- 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.

We look for where it fits within the sorted subarray and insert it there. Since there is nothing really in the sorted subarray, we leave it as is.

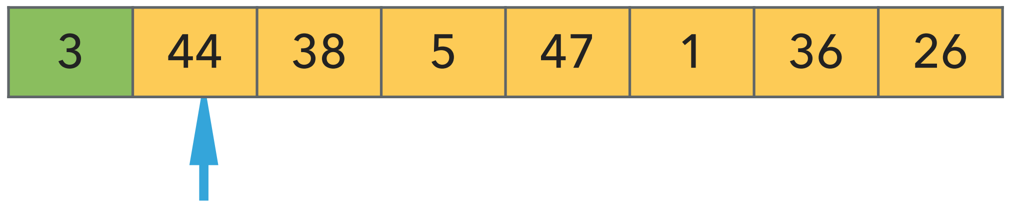
Insertion sort



- 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.

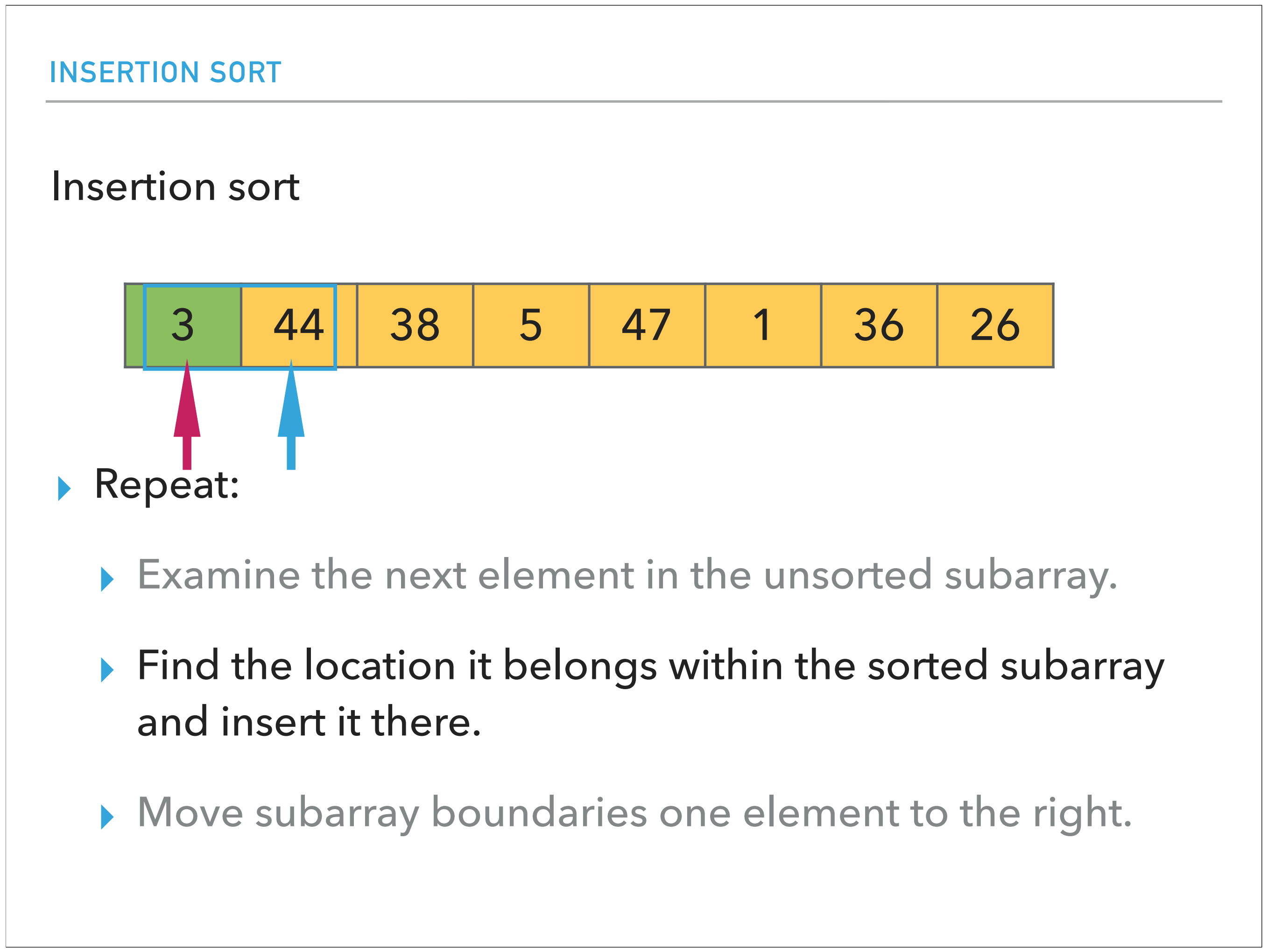
And mark the first index as partially sorted.

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.

Let's examine the second element.



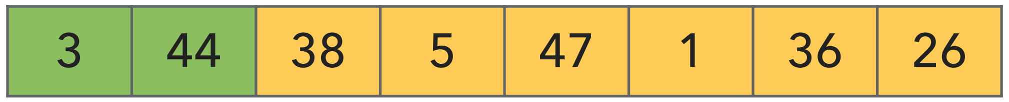
INSERTION SORT

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.

Does it go before or after 3? It stays as is, after it.

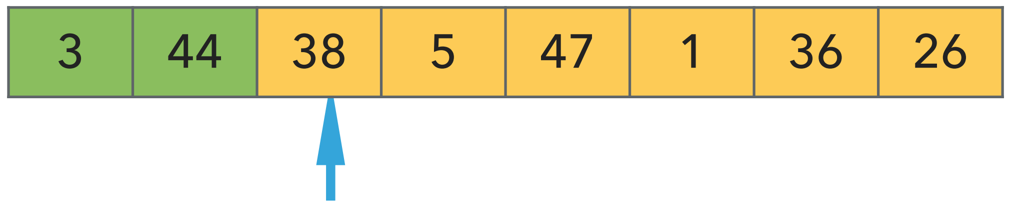
Insertion sort



- 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.

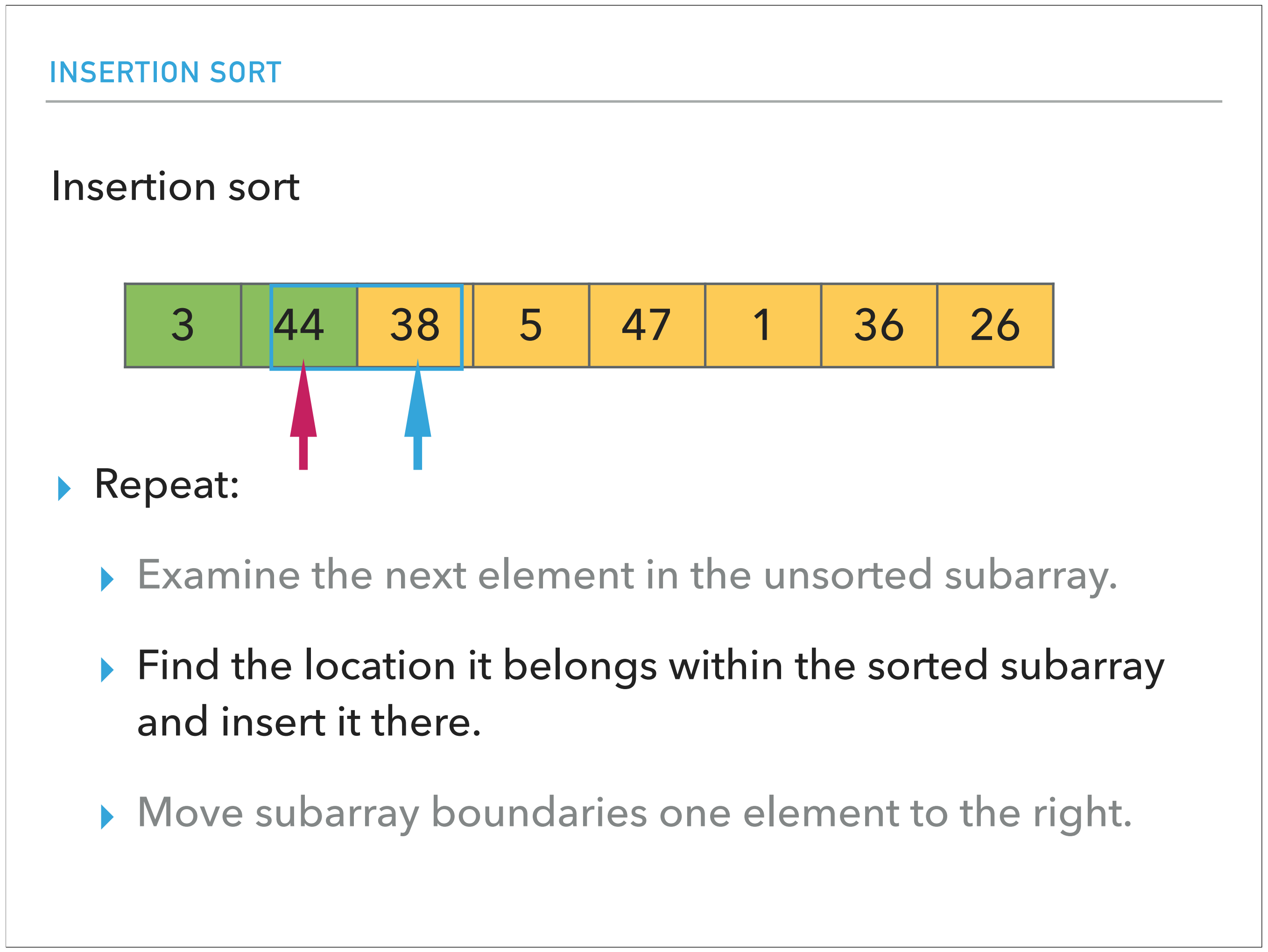
Now 3 and 44 are partially sorted.

Insertion sort

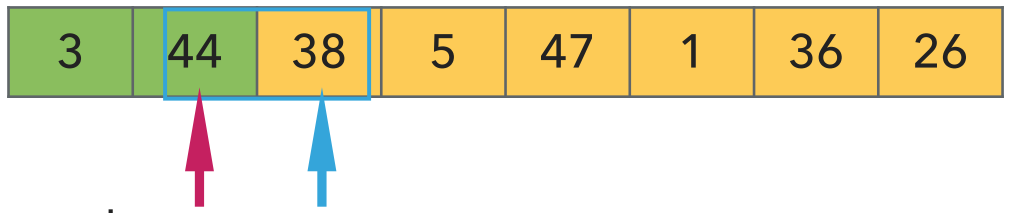


- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Third element is 38.



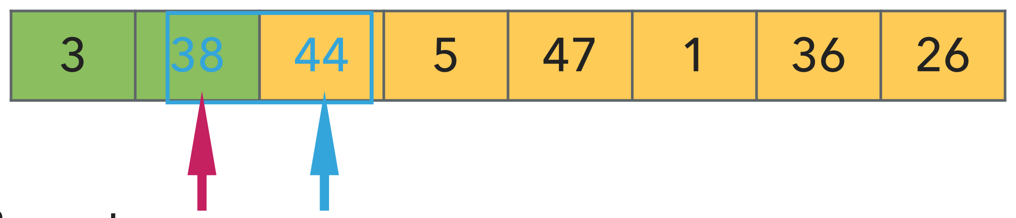
Insertion sort



- 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.

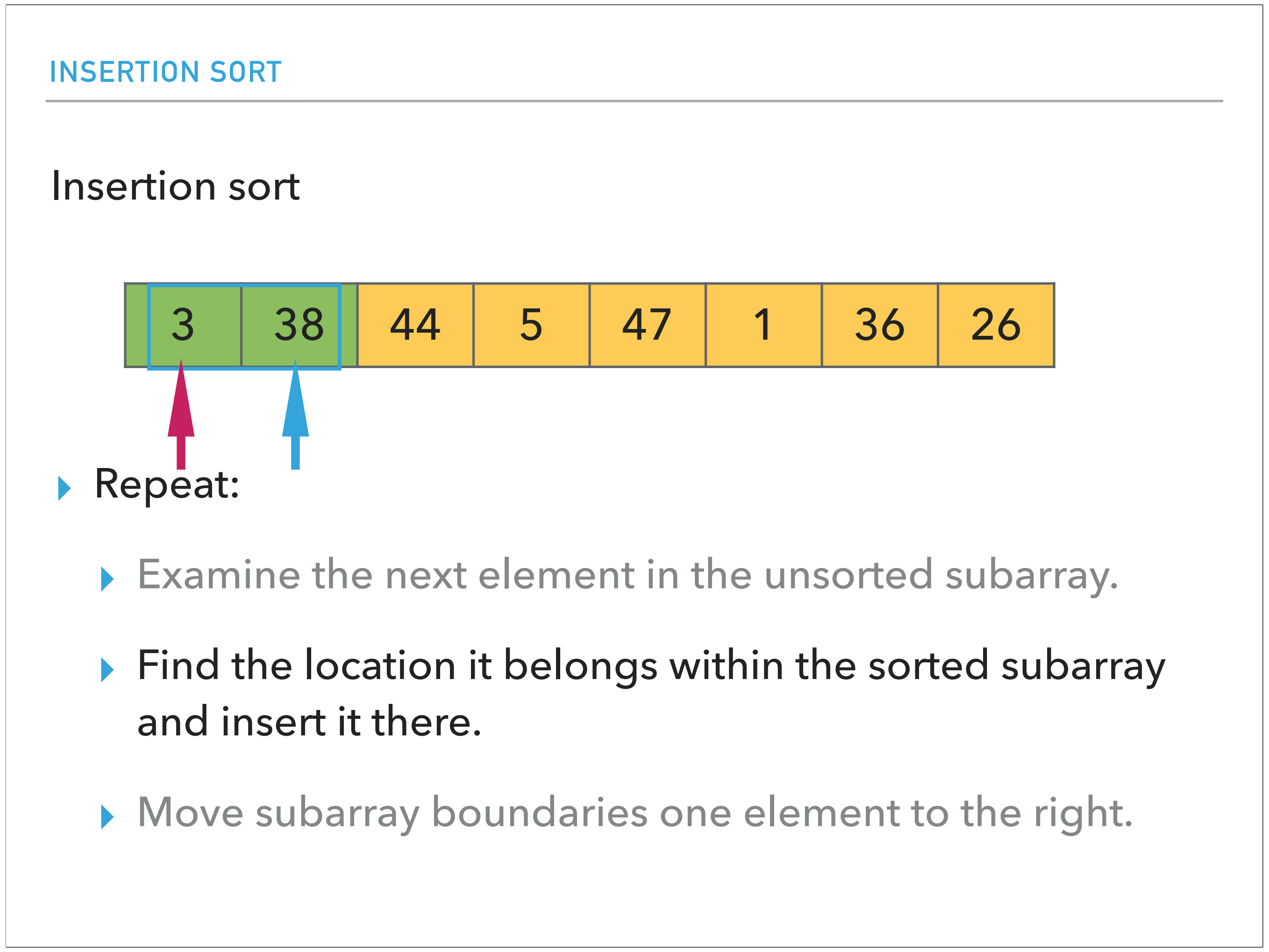
We compare it with 44 and it's smaller/

Insertion sort

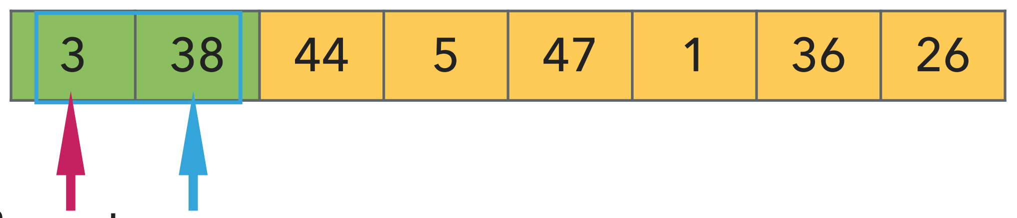


- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

So we swap 38 and 44.



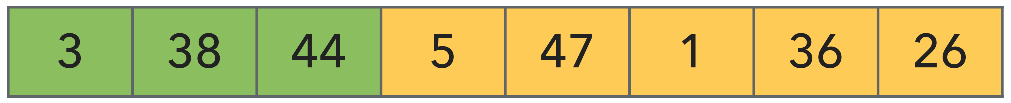
Insertion sort



- 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.

We compare 38 with 3 but it is larger so it stays there.

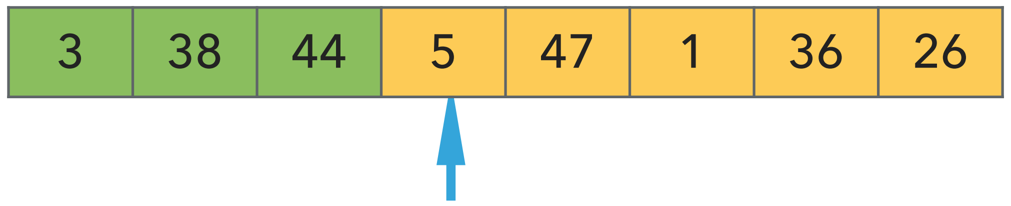
Insertion sort



- 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.

we now have 3, 38, and 44 partially sorted.

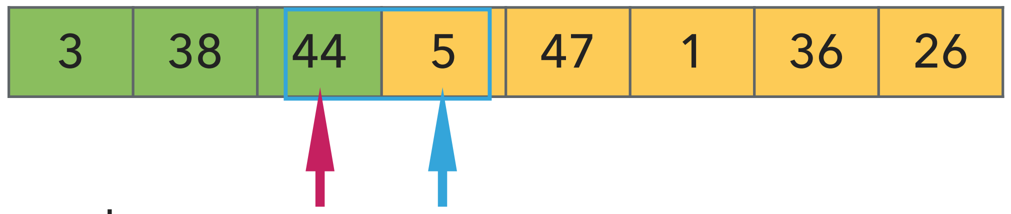
Insertion sort



- 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.

Fourth element is 5.

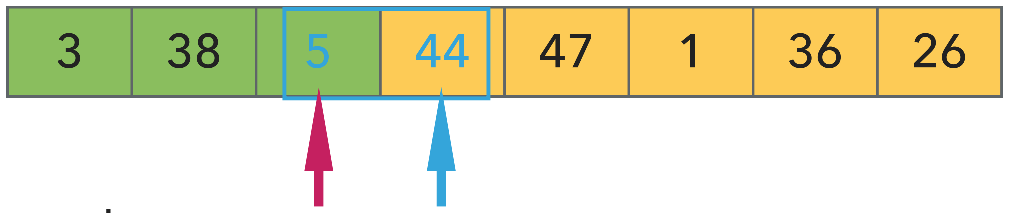
Insertion sort



- 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.

We compare it with 44...

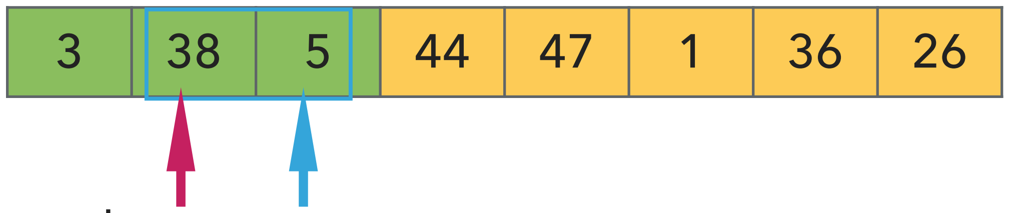
Insertion sort



- 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.

and swap them.

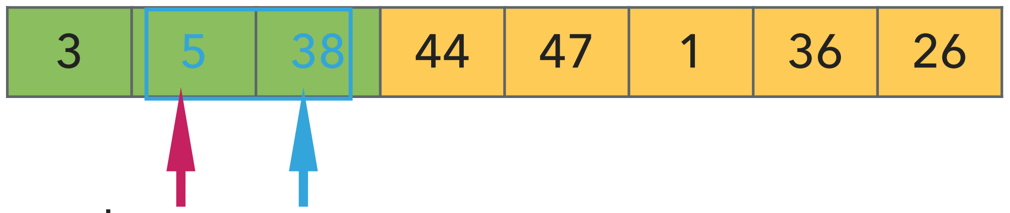
Insertion sort



- 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.

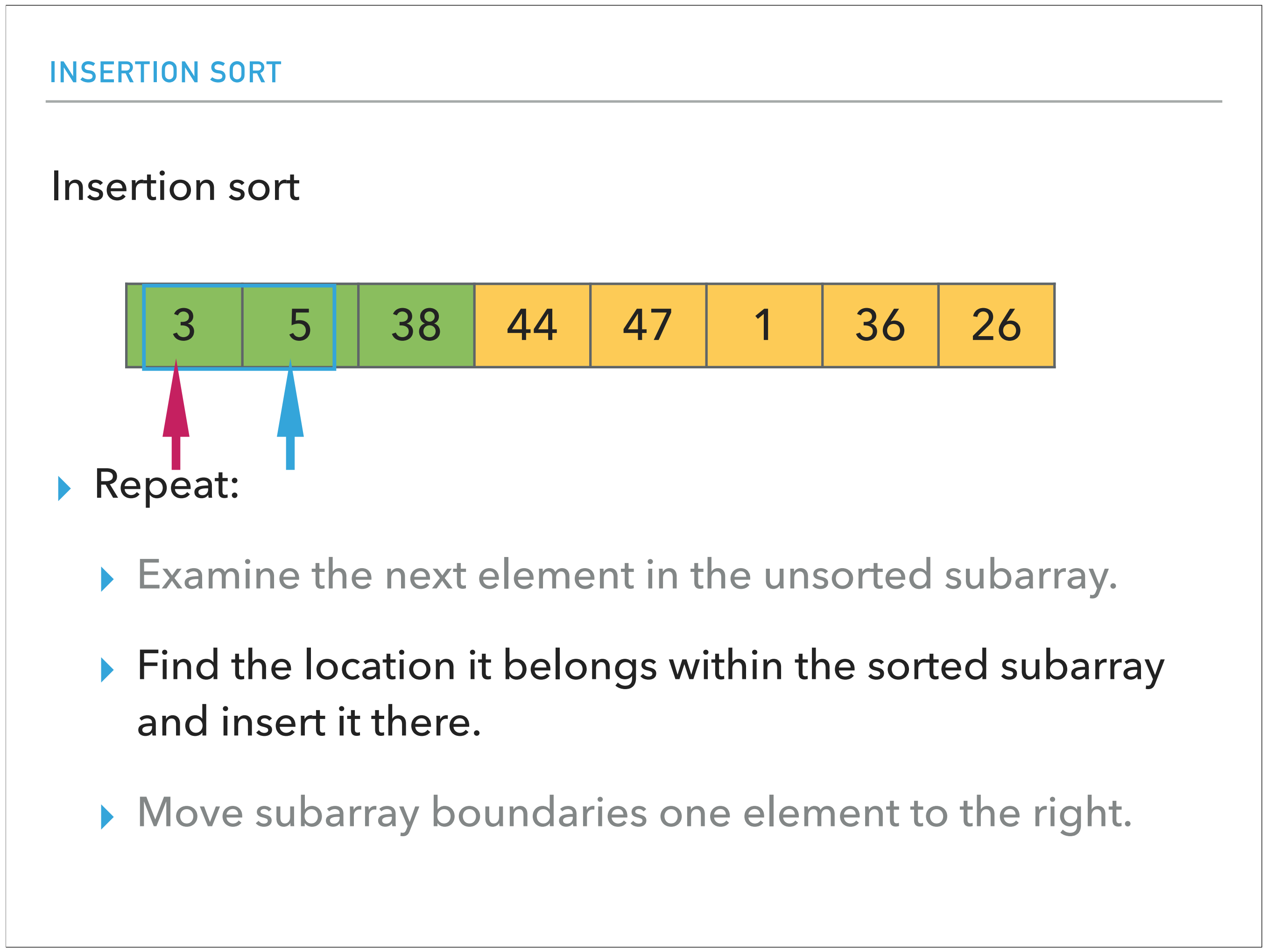
Compare it with 38...

Insertion sort

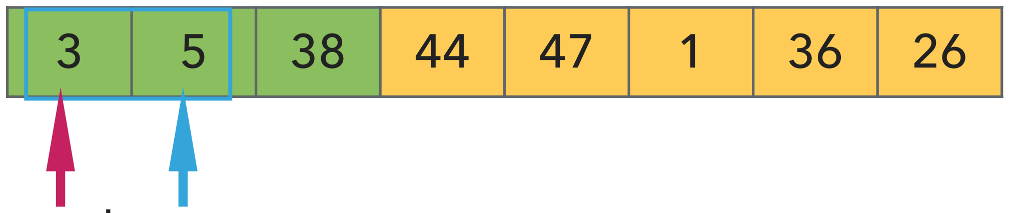


- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

and swap them.



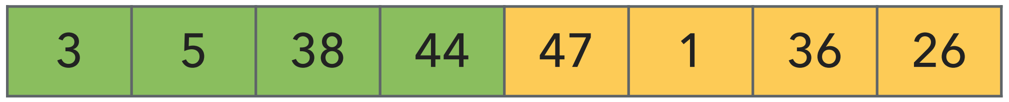
Insertion sort



- 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.

Compare it with 3 and it stays there.

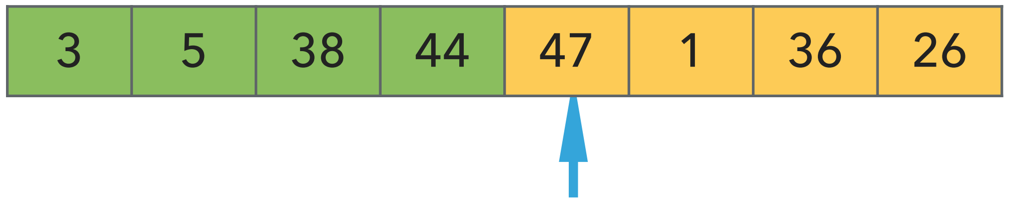
Insertion sort



- 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.

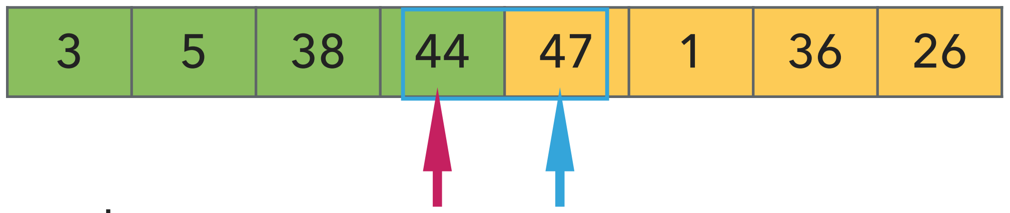
Now 3, 5, 38, 44, are at their final place... The rest of the steps are the same until we move 26 to its final position.

Insertion sort



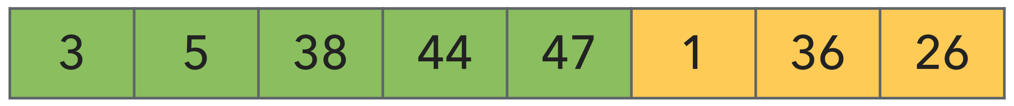
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



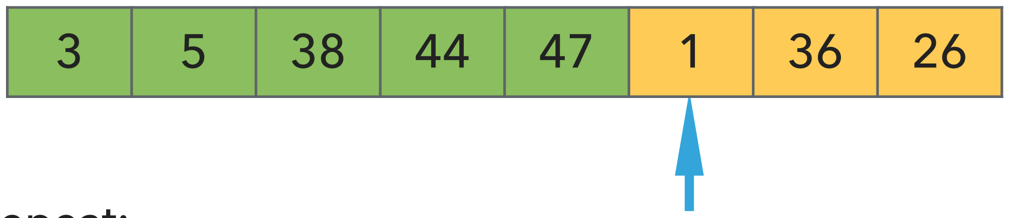
- Repeat:
    - Examine the next element in the unsorted subarray.
    - Find the location it belongs within the sorted subarray and insert it there.
    - Move subarray boundaries one element to the right.

Insertion sort



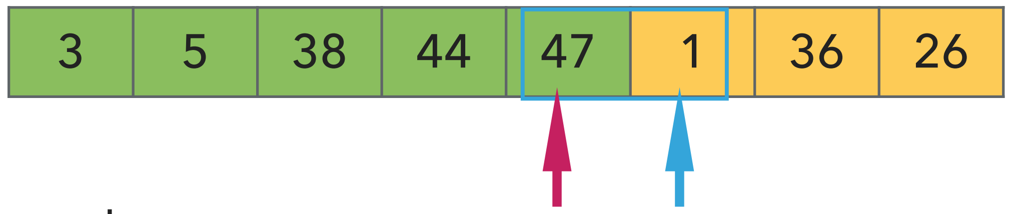
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



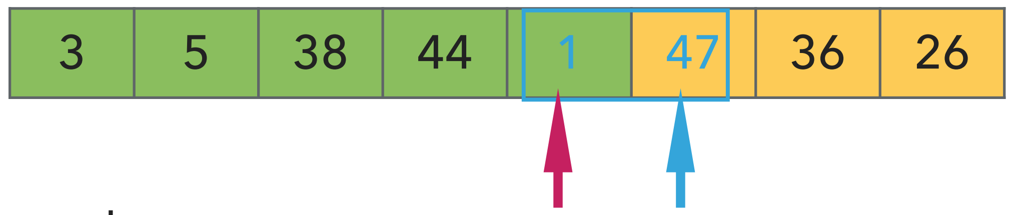
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



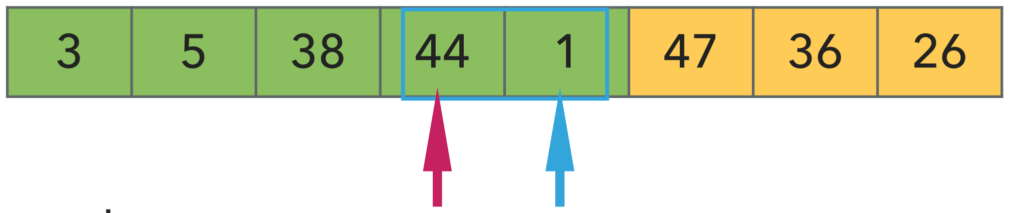
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



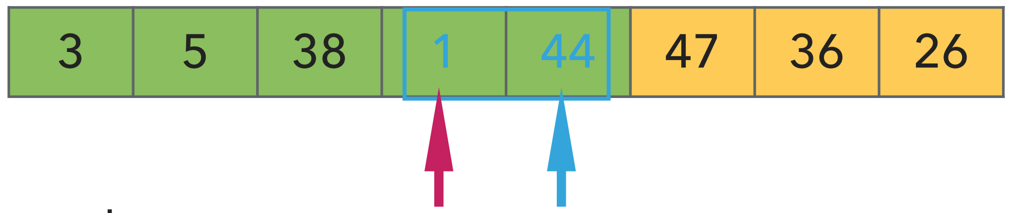
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



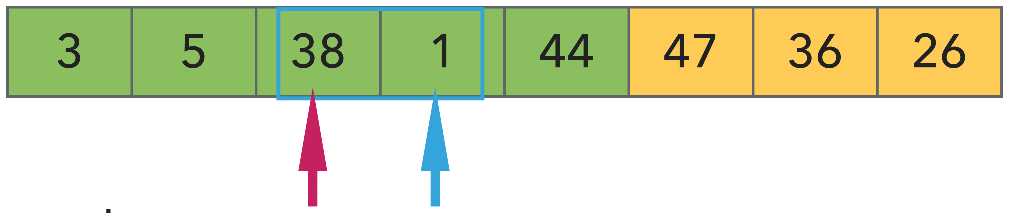
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



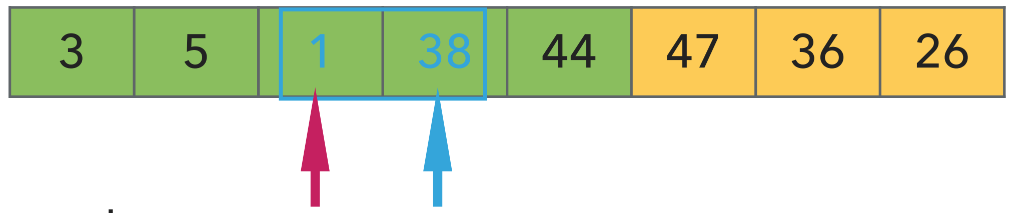
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



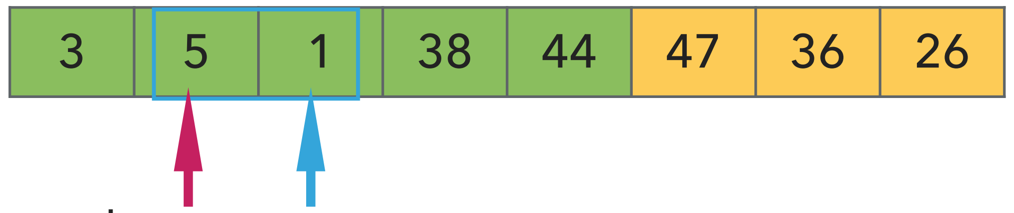
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



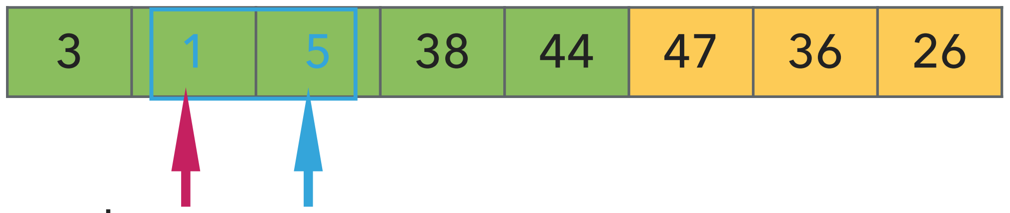
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



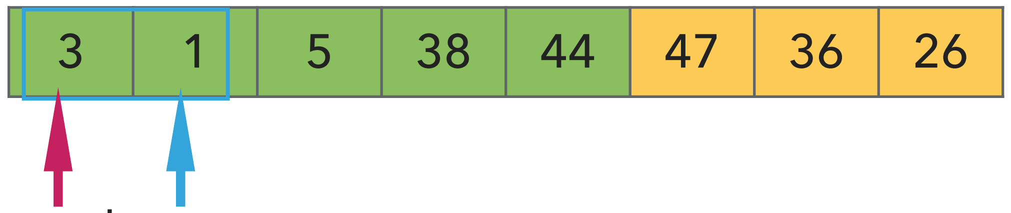
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



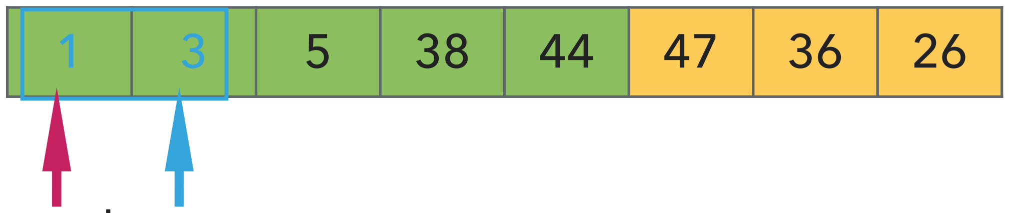
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



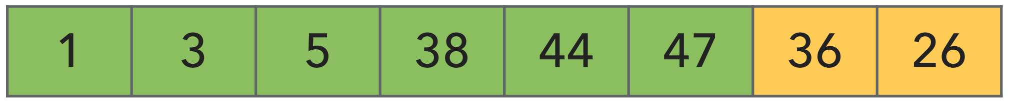
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



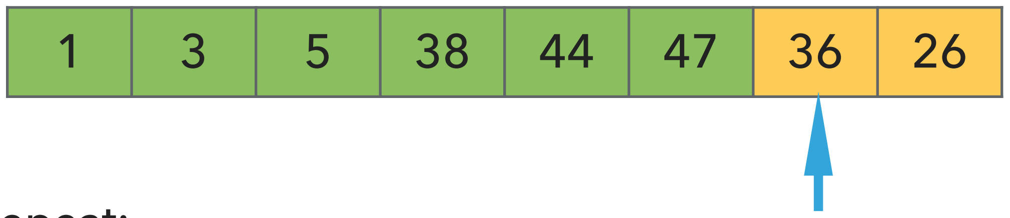
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



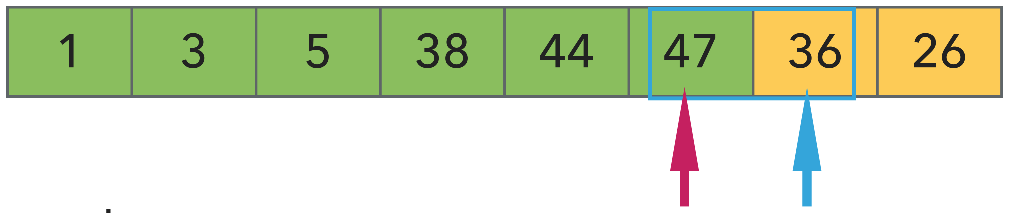
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



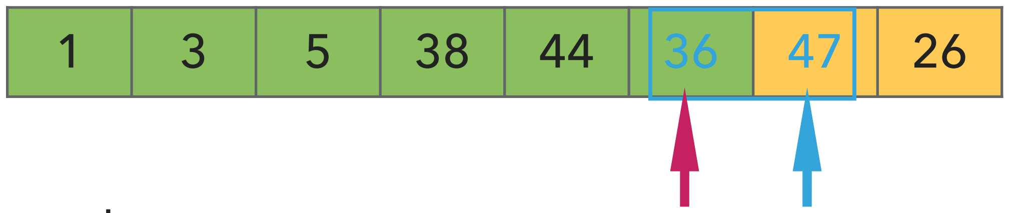
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



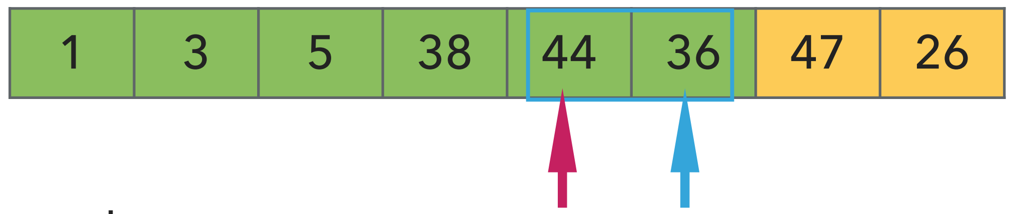
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



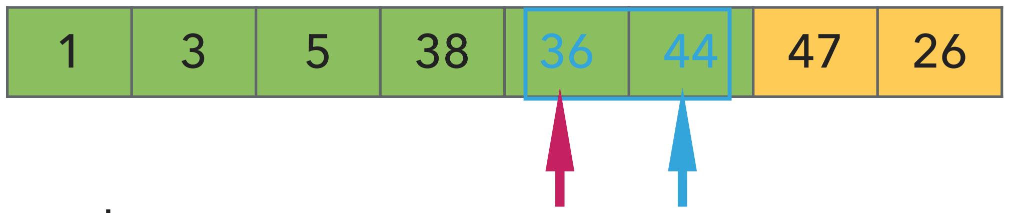
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



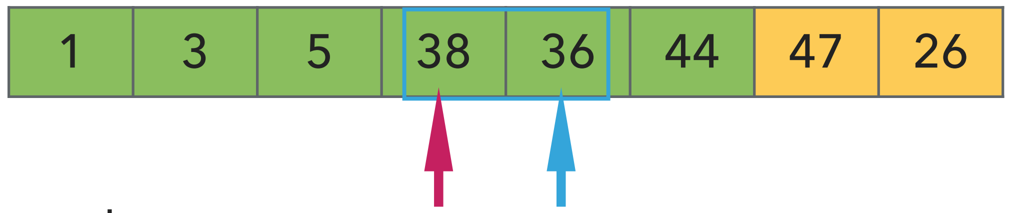
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



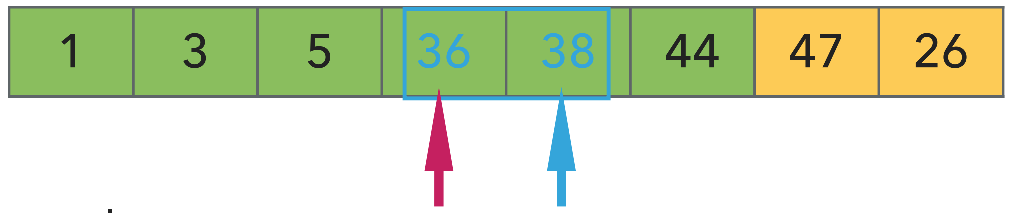
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



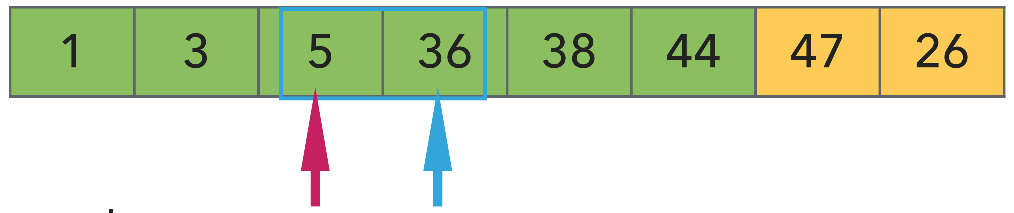
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



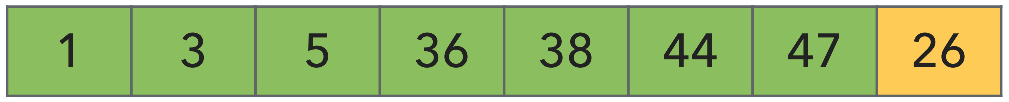
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



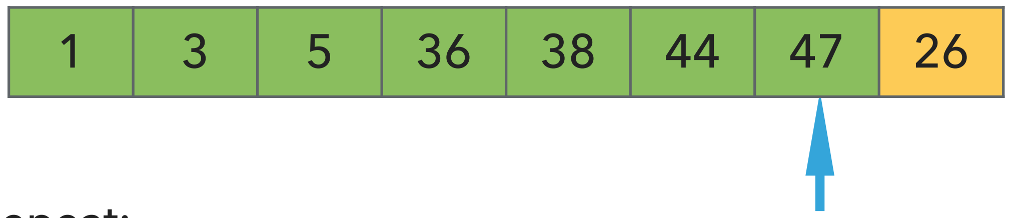
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



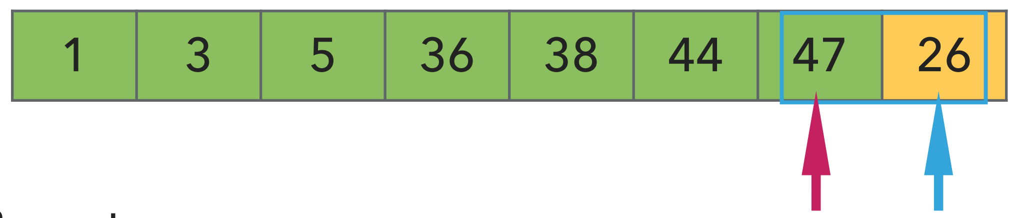
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



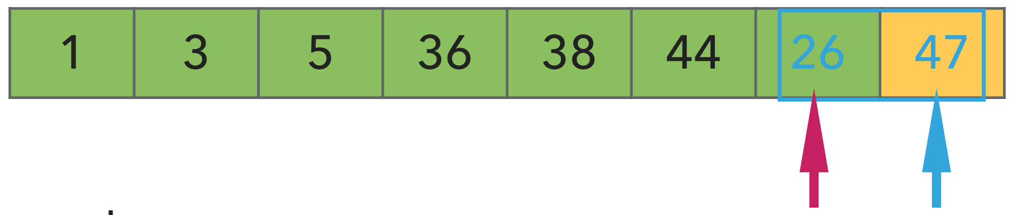
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



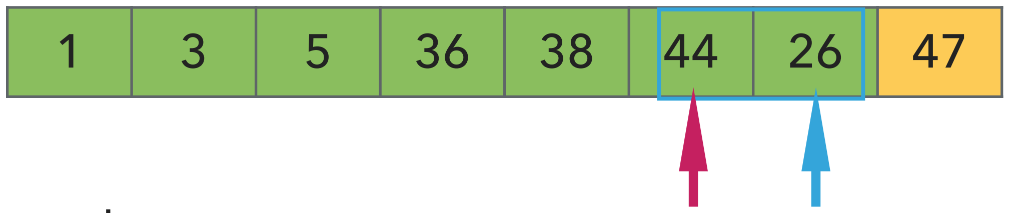
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



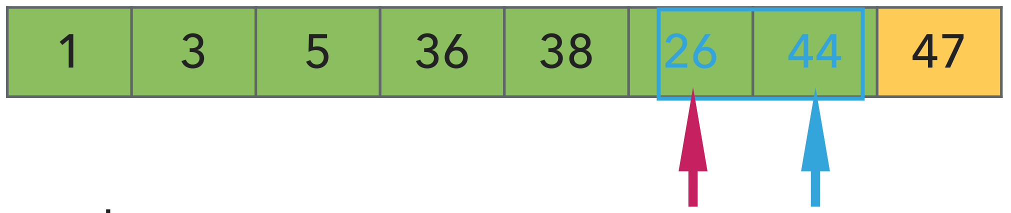
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



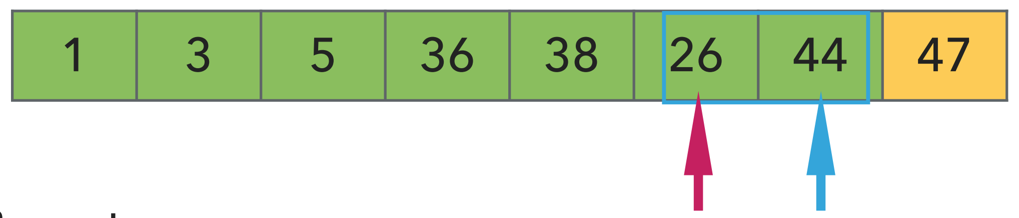
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



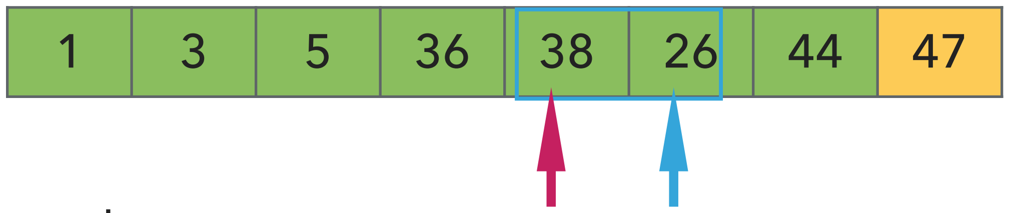
- Repeat:
    - Examine the next element in the unsorted subarray.
    - Find the location it belongs within the sorted subarray and insert it there.
    - Move subarray boundaries one element to the right.

Insertion sort



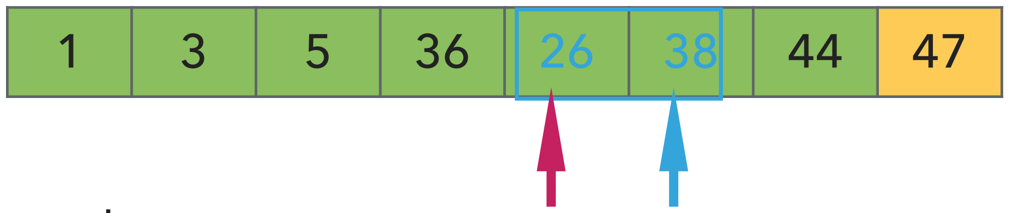
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



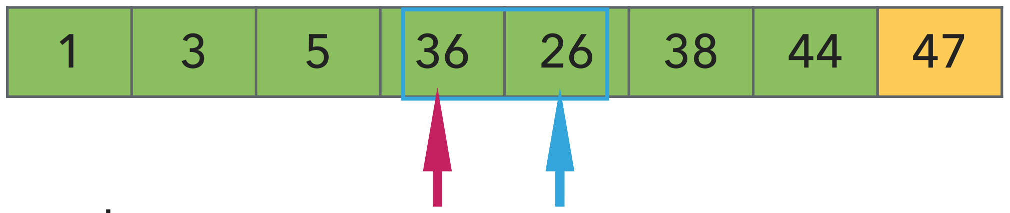
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



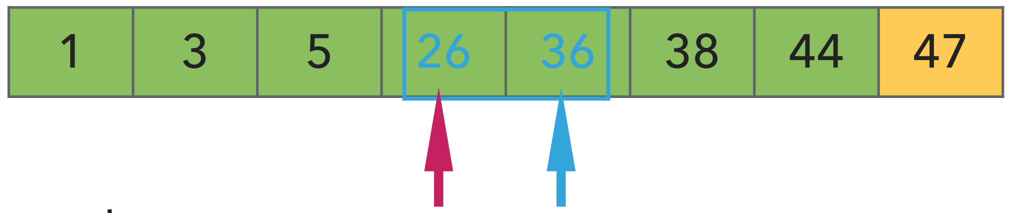
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort



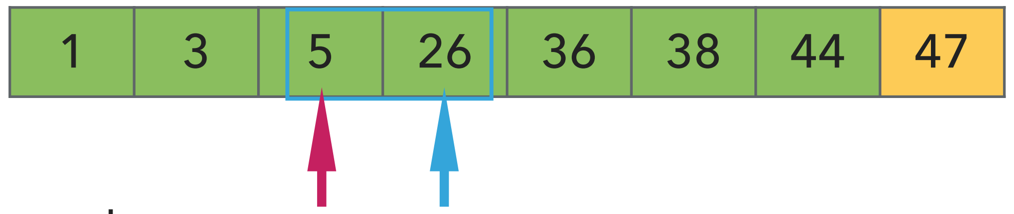
- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

Insertion sort

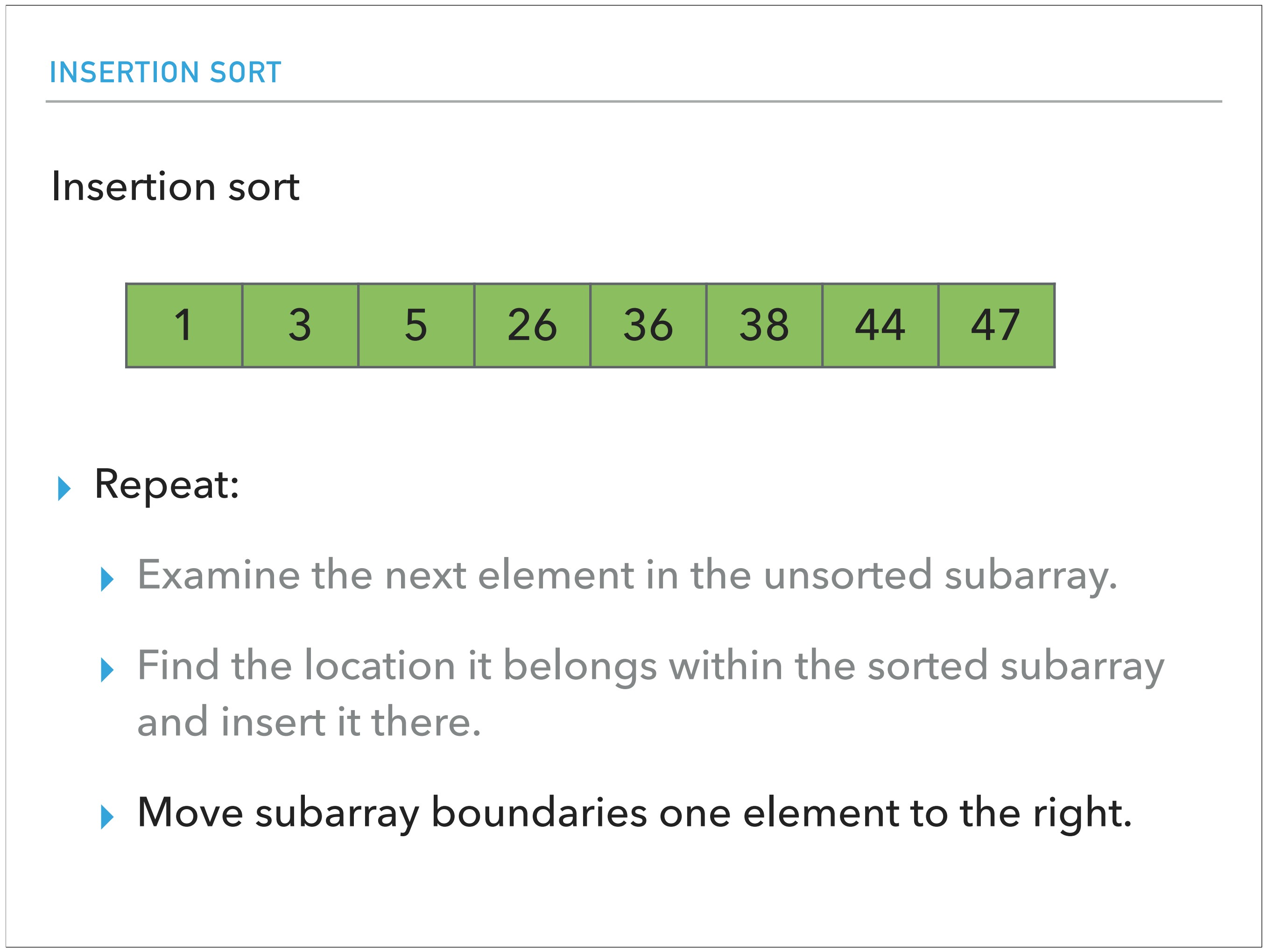


- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.

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.



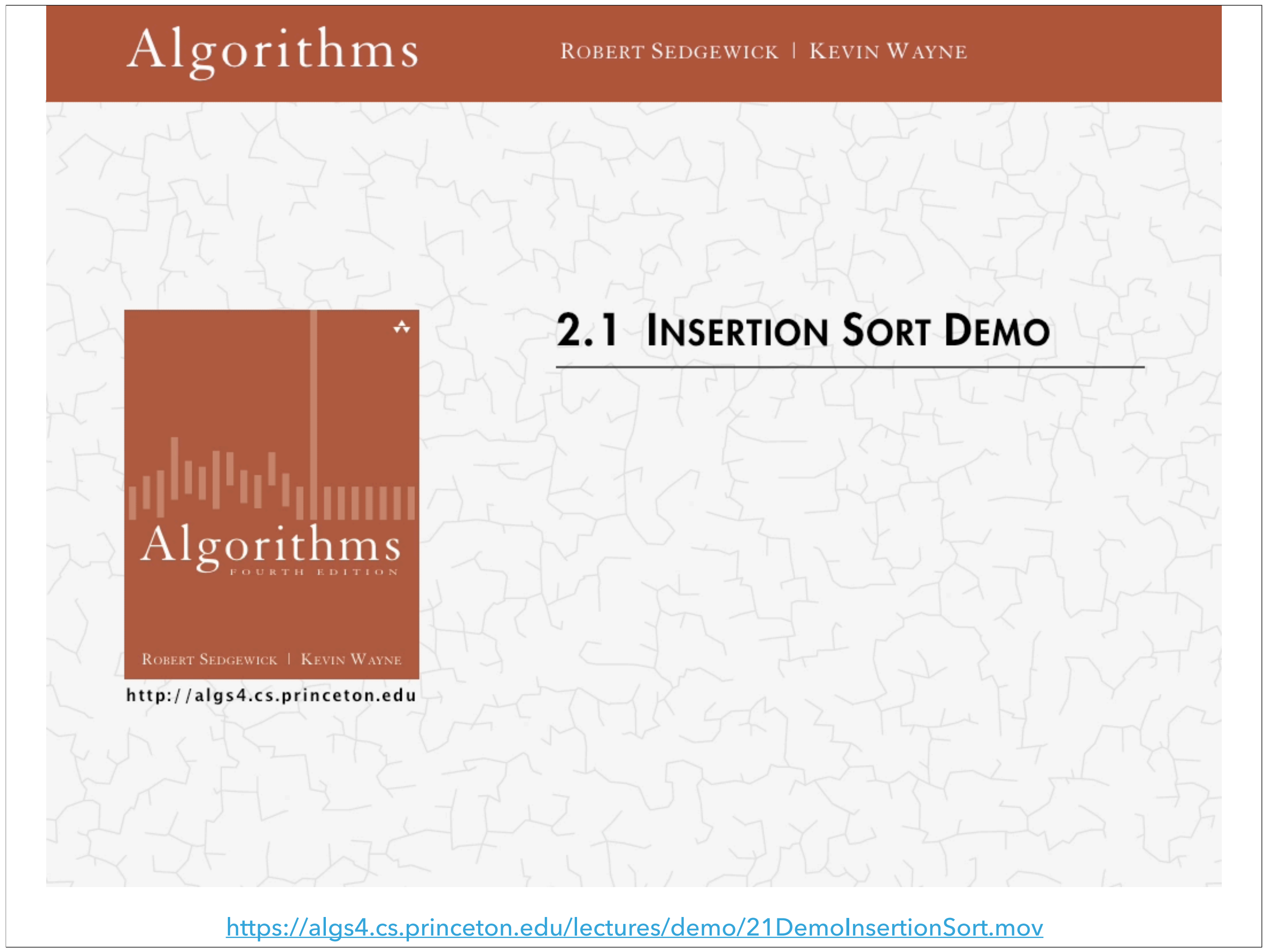
INSERTION SORT

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.

We now have a fully sorted array!



Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

2.1 Insertion Sort Demo

Algorithms FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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

This is a demo of insertion sorts with playing cards.

In case you didn't get this…

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

And here is a funny video with folk dancers that demonstrate insertion sort.

PRACTICE TIME - Implement insertion sort

```
public static <E extends Comparable<E>> void insertionSort(E[] a) {







 }
```

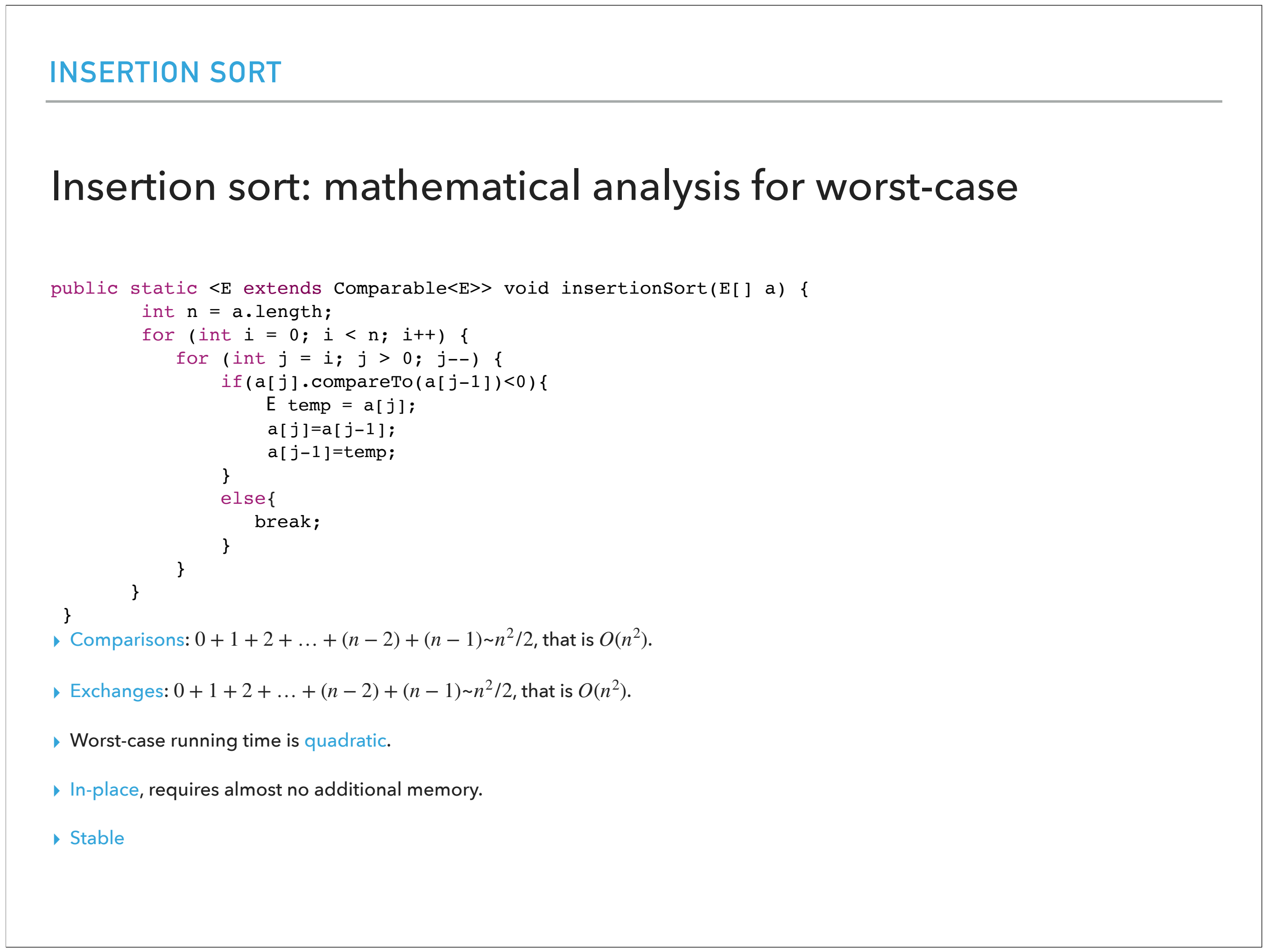
Take a moment and think about what we did so far. How would you implement this method so that you would sort the array a of Comparable items of type E? Remember, generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed. The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

Insertion sort

```
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]`

Presumably you ended with something like this.
At the end of each iteration, the array a is sorted in ascending order for the first i+1 elements a[0…i]



Insertion sort: mathematical analysis for worst-case

```java
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;
               }
           }
       }
 }
```

- 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.
- In-place, requires almost no additional memory.
- Stable

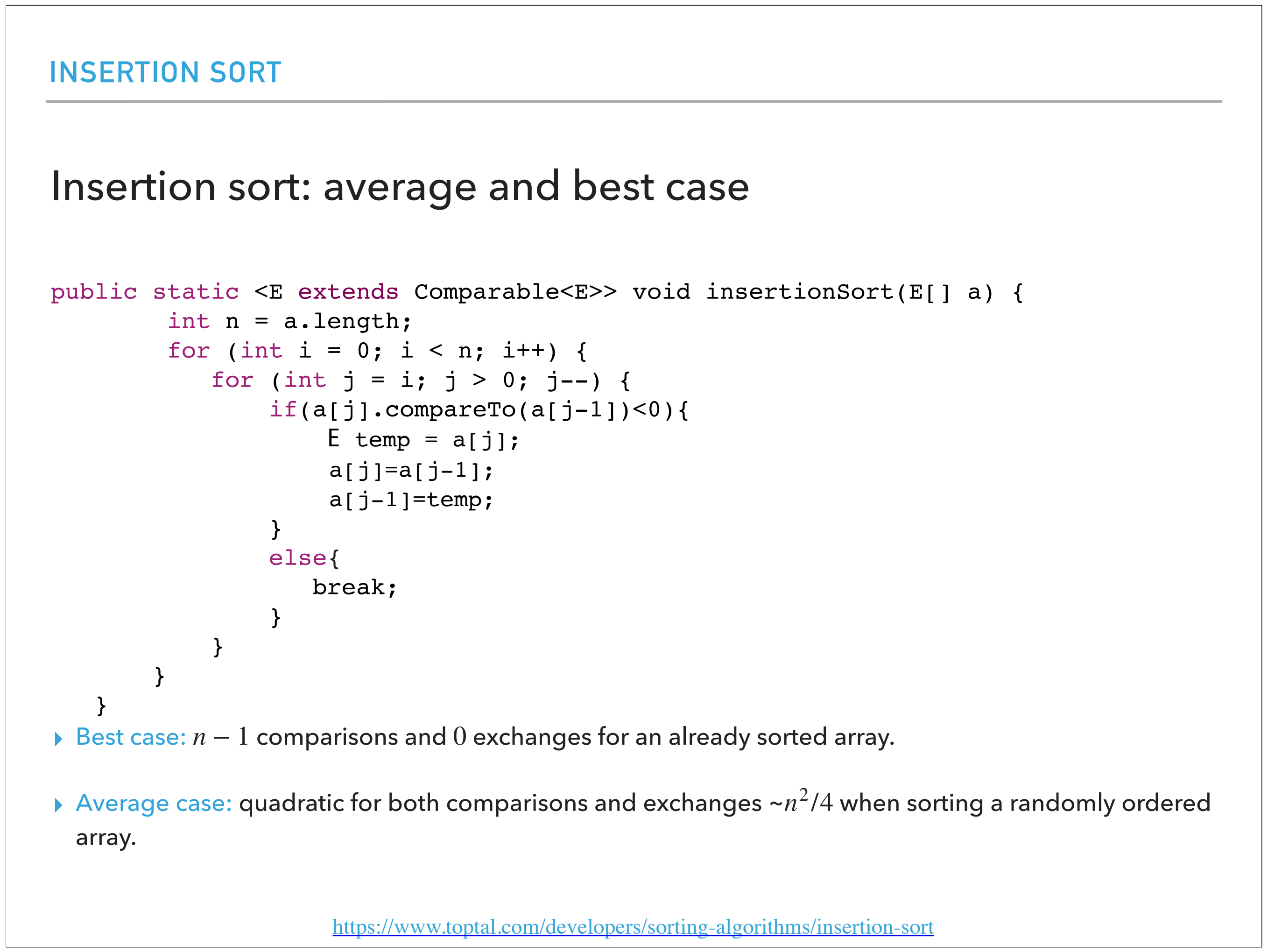
How many comparisons do we make? The simplest worst case input is an array sorted in reverse order

In the worst case:
For i=0 make 0 comparisons
For i = 1 make 1 comparisons
…
For i=n-1 make n-1 comparisons
Total: 1+2+…+n-2+n-1 = n(n-1)/2~O(n^2)

What about exchanges? In the worst case (let's say a reversely ordered array), we would need to again make ~O(n^2) exchanges.

Insertion sort is in place as it requires almost no additional memory except for a handful of temporary variables.

And the good news is that it is stable.



INSERTION SORT

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 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

In the best case:
for each external loop, we only make one comparison, we get in the else statement, and break the inner loop.
That would mean at best case we make n-1 comparisons. Such a scenario would happen if we were given an already sorted array.

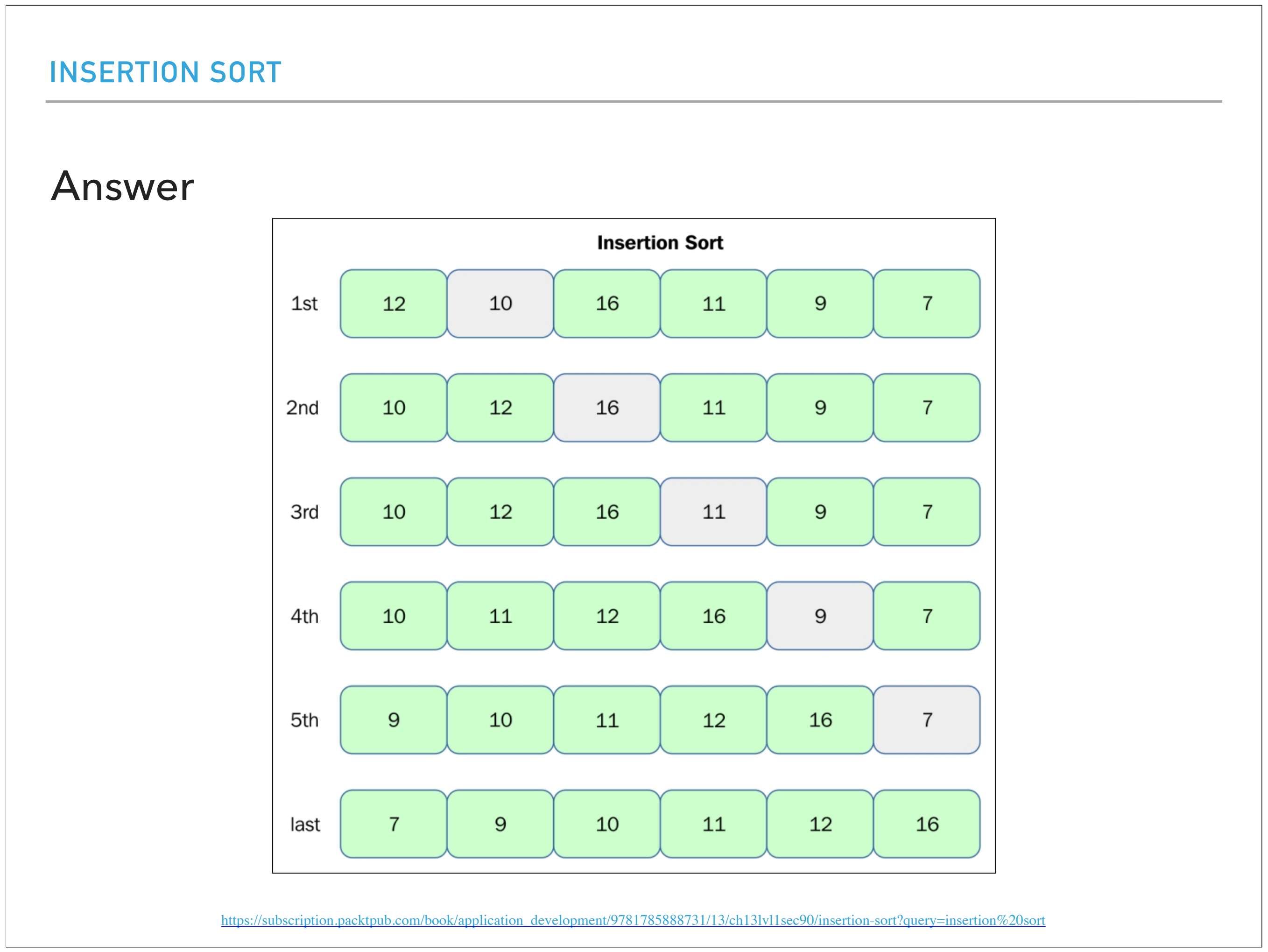
In terms of exchanges, for best case, since none of the if statements would be satisfied, we would have 0 exchanges.

We won't do the proof for average case, but it's quadratic both for comparisons and exchanges when sorting a randomly ordered array.

Practice Time - Worksheet

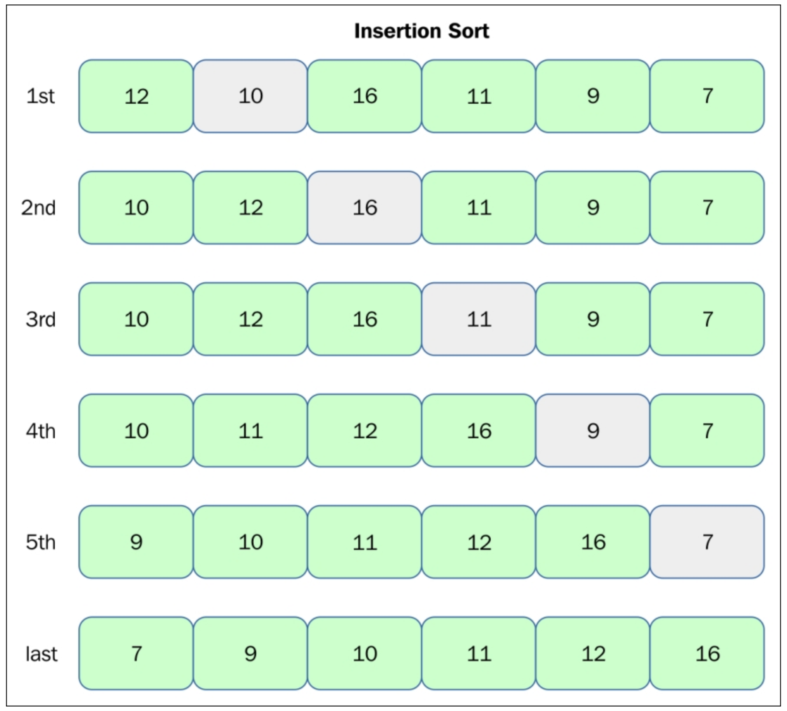
- Using insertion sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm.

Let's practice!



INSERTION SORT

Answer



Insertion Sort

1st	12	10	16	11	9	7
2nd	10	12	16	11	9	7
3rd	10	12	16	11	9	7
4th	10	11	12	16	9	7
5th	9	10	11	12	16	7
last	7	9	10	11	12	16

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

Did you get this answer?

Lecture 13: Insertion Sort

- Insertion sort

To summarize, we saw insertion sort an algorithm that like selection sort has an unsorted and sorted subarray. But here, the sorted array elements are not in their final position until we run the full algorithm. Unlike selection sort, the running time of insertion sort depends on the initial order of the items in the input. For example, if the array is large and its entries are already in or nearly in order, then insertion sort, is MUCH faster than if the entries are randomly ordered or in reverse order. Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. If the array is in order or the keys are all equal, then the total running time is linear. In general, iteration sort works very well for partially sorted arrays, e.g., :
an array where each entry is not far from its final position, or a small array appended to a large sorted array, or an array with only a few entries that are not in place. That makes insertion sort not only excellent for partially sorted arrays but also a fine method for tiny arrays, which will come handy in intermediate stages of advanced sorting algorithms like merge sort which we'll see soon.

Readings:

- Recommended Textbook:
  - Chapter 2.1 (pages 244–262)
  - Chapter 2.5 (Pages 338-339)
- Recommended Textbook Website:
  - Elementary sorts: https://algs4.cs.princeton.edu/21elementary/

Code

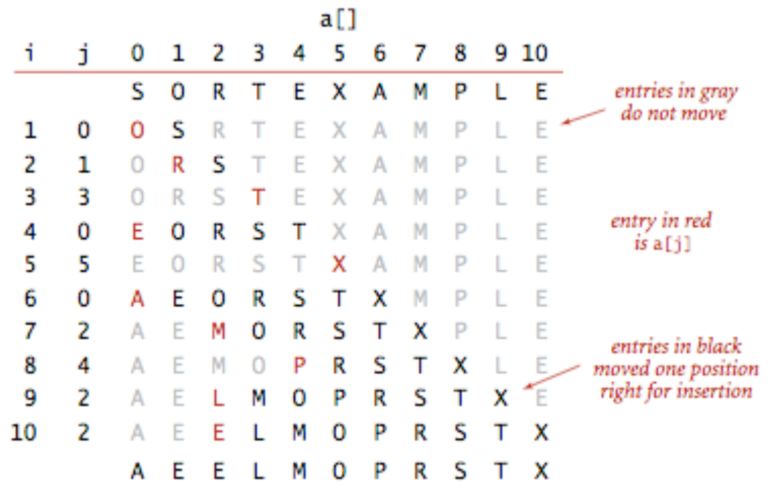
- Lecture 13 code

Worksheet

- Lecture 13 worksheet

Practice Problem 1 - 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.


entries in gray do not move
entry in red is a[j]
entries in black moved one position right for 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

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

Practice Problem 2

- 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 3 - Recommended textbook 2.1.6

- Which method runs faster for an array with all keys identical, selection sort or insertion sort?

Practice Problem 4 - Recommended textbook 2.1.7

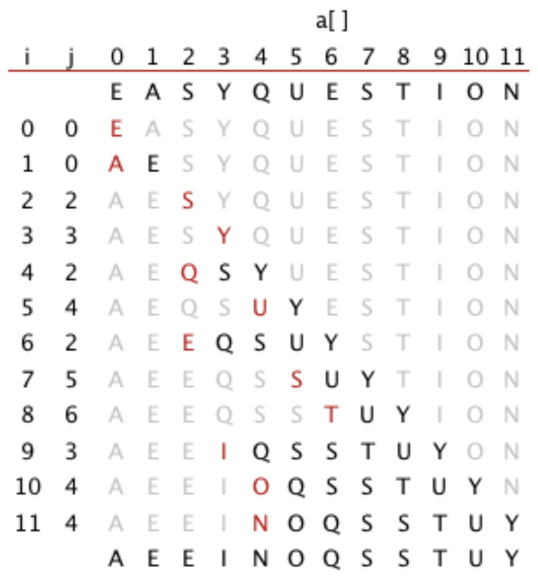
- Which method runs faster for an array in reverse order, selection sort or insertion sort?

Practice Problem 5 - 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 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 2

- 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]
- if statement always true when the array is in reverse order, e.g., [4, 3, 2, 1].

ANSWER 3

- 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 4

- 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 5

- 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.