

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

12: Insertion Sort



Tom Yeh
he/him/his

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  
    default void remove(); //optional, better avoid it altogether  
}
```

Iterator Example

```
List<String> myList = new ArrayList<String>();  
//... operations on myList
```

```
Iterator listIterator = myList.iterator();
```

```
while(listIterator.hasNext()){  
    String elt = listIterator.next();  
    System.out.println(elt);  
}
```

TEXT

Rules of the game - Comparing

- ▶ `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`.
- ▶ Java classes such as `Integer`, `Double`, `String`, `File` all implement `Comparable`.

Two useful 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;  
}
```

SELECTION 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

Selection sort

```
public static void 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);  
    }  
}
```

← In iteration i

← Find the index min of the smallest remaining array

← swap $a[i]$ and $a[\text{min}]$

► **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

Selection sort: mathematical analysis for worst-case

```
public static void 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:** $1 + 2 + \dots + (n - 2) + (n - 1) \sim n^2/2$, 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.
- ▶ **In-place**, requires almost no additional memory.
- ▶ **Not stable**, think of the array $[5_a, 3, 5_b, 1]$ which will end up as $[1, 3, 5_b, 5_a]$.

SELECTION SORT

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



Lecture 12: Sorting Fundamentals

- ▶ Introduction
- ▶ Selection sort
- ▶ Insertion sort

INSERTION SORT

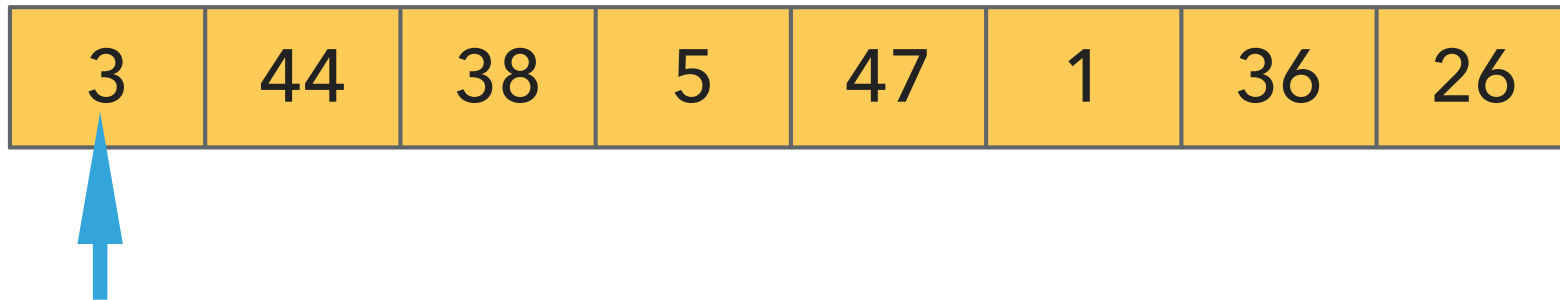
Insertion sort

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.
 - ▶ Move subarray boundaries one element to the right.

INSERTION SORT

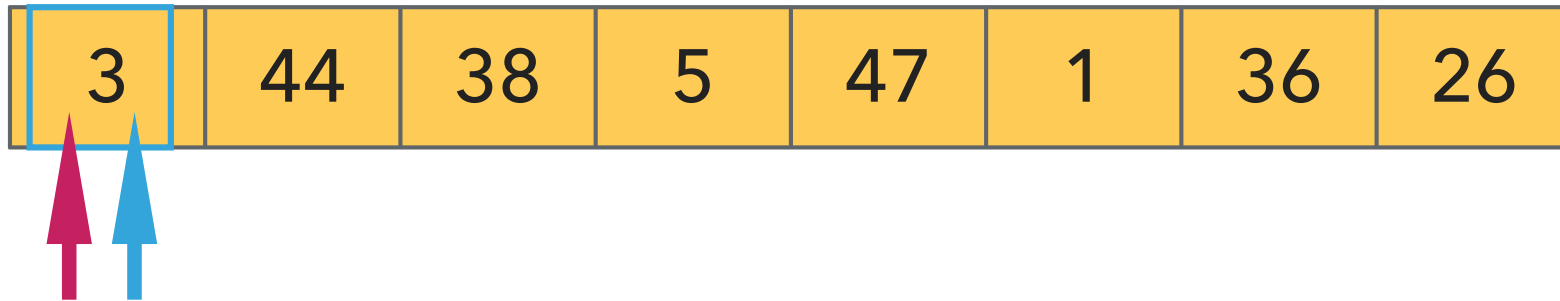
Insertion sort



- ▶ 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.

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.

INSERTION SORT

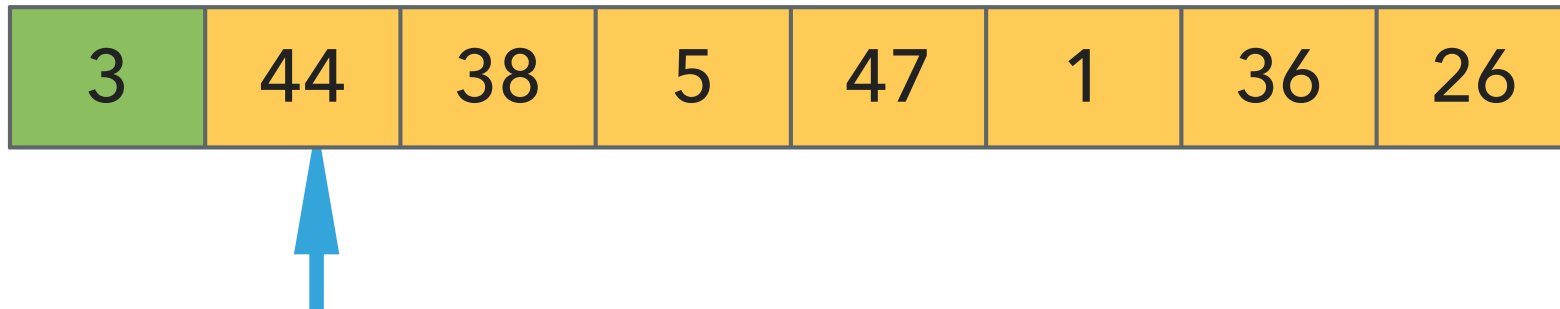
Insertion sort

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

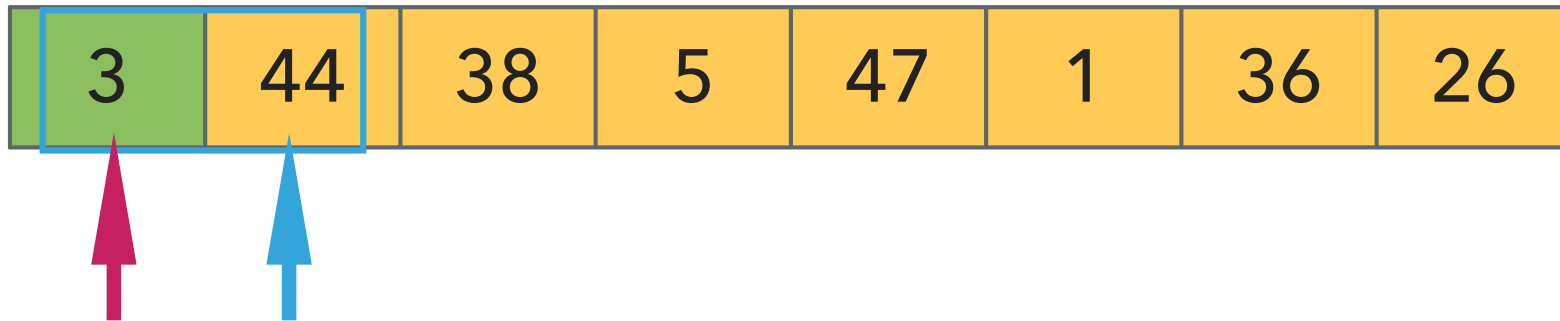
Insertion sort



- ▶ 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



► 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

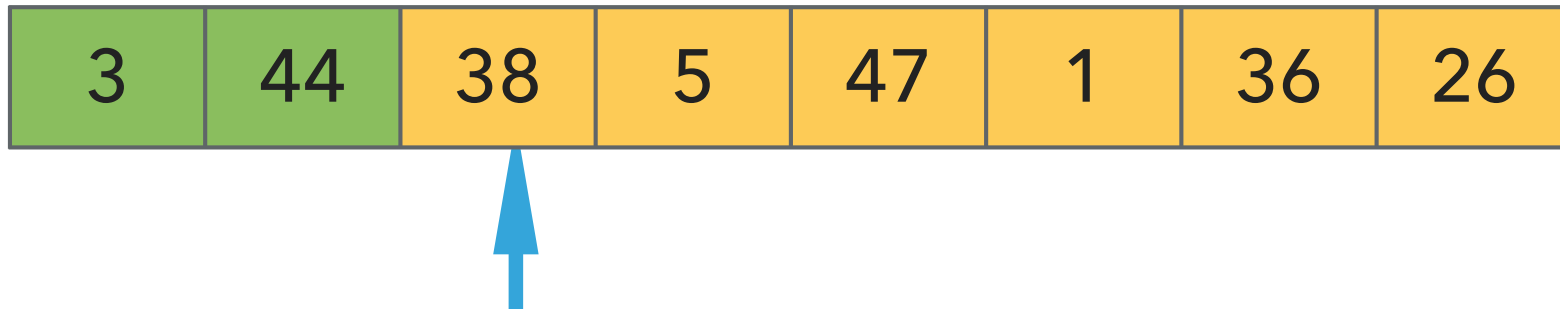
Insertion sort

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

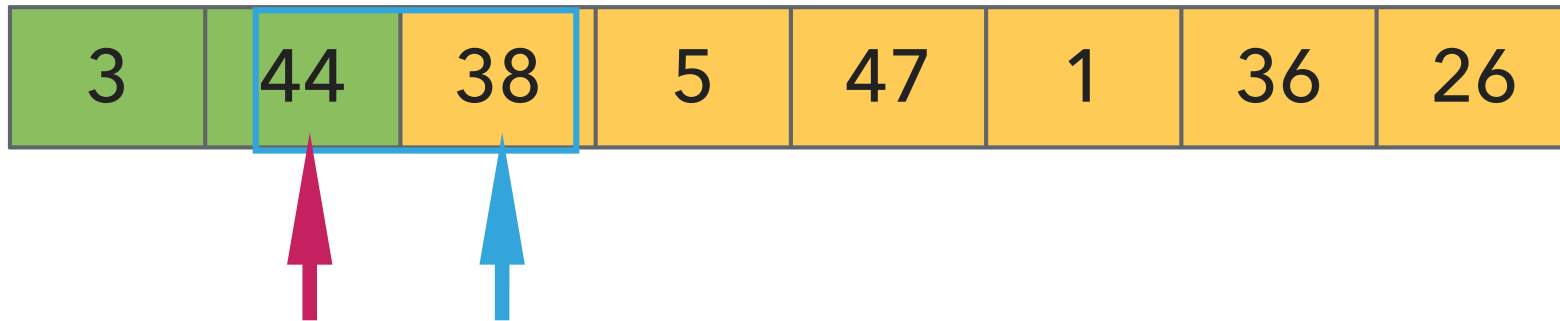
Insertion sort



- ▶ 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

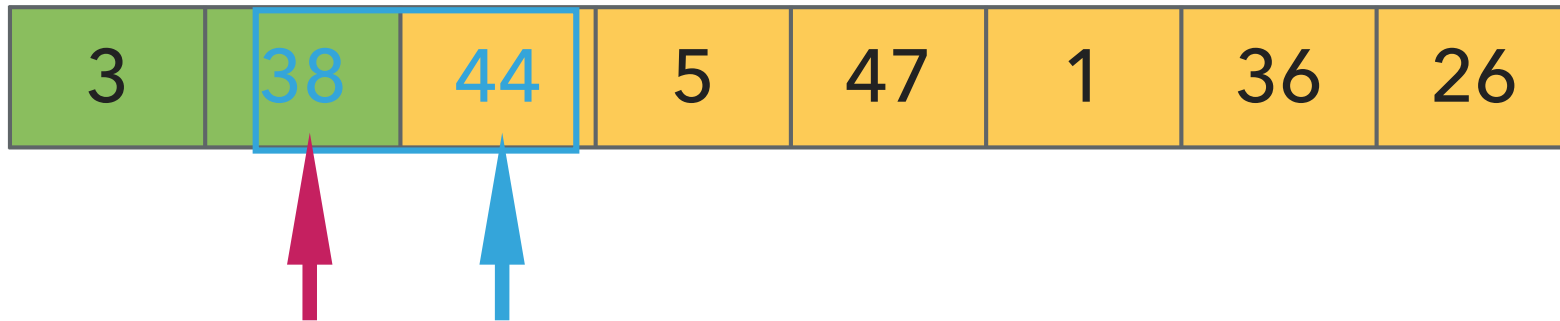
Insertion sort



- ▶ 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

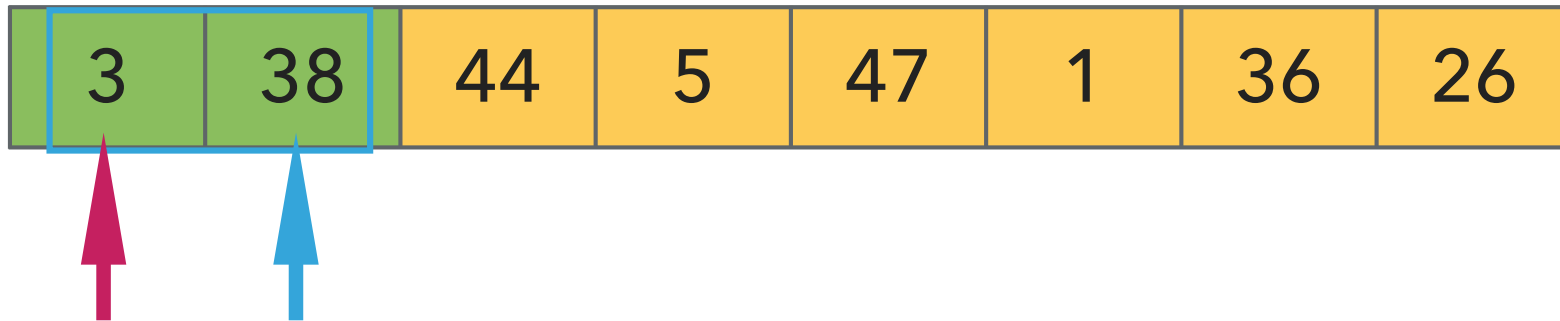
Insertion sort



- ▶ 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



- ▶ 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

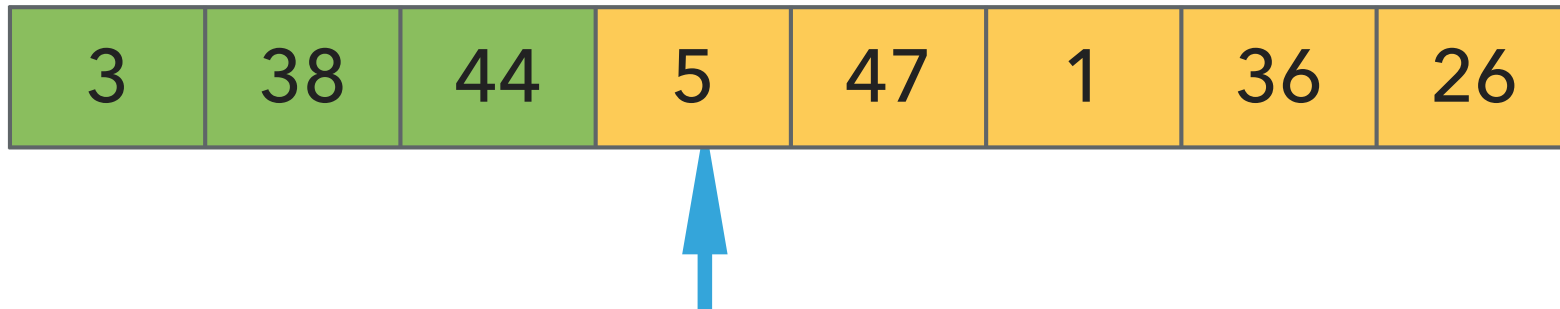
Insertion sort

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

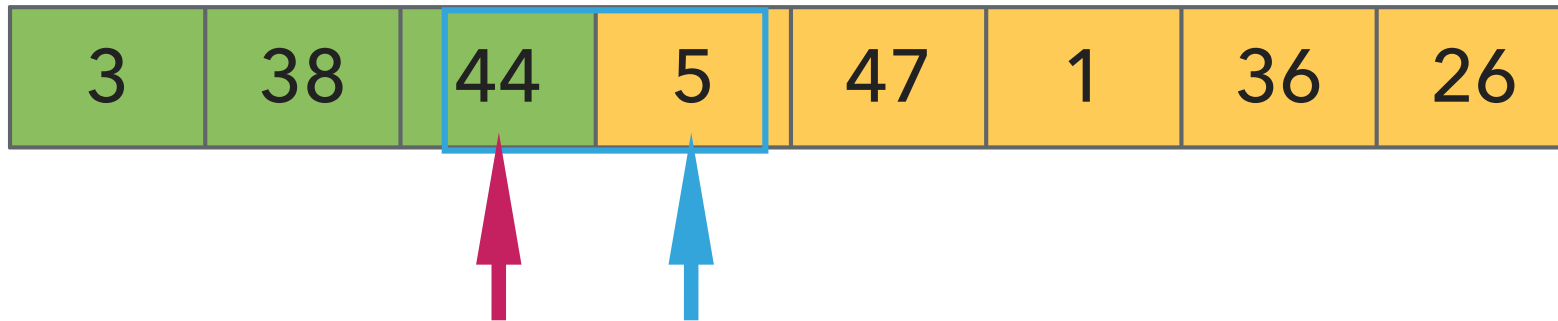
Insertion sort



- ▶ 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

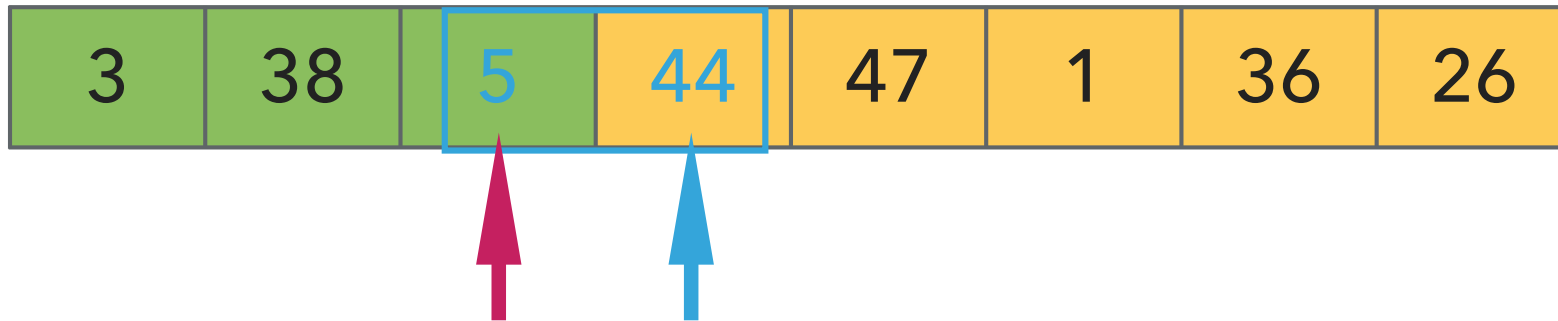
Insertion sort



- ▶ 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

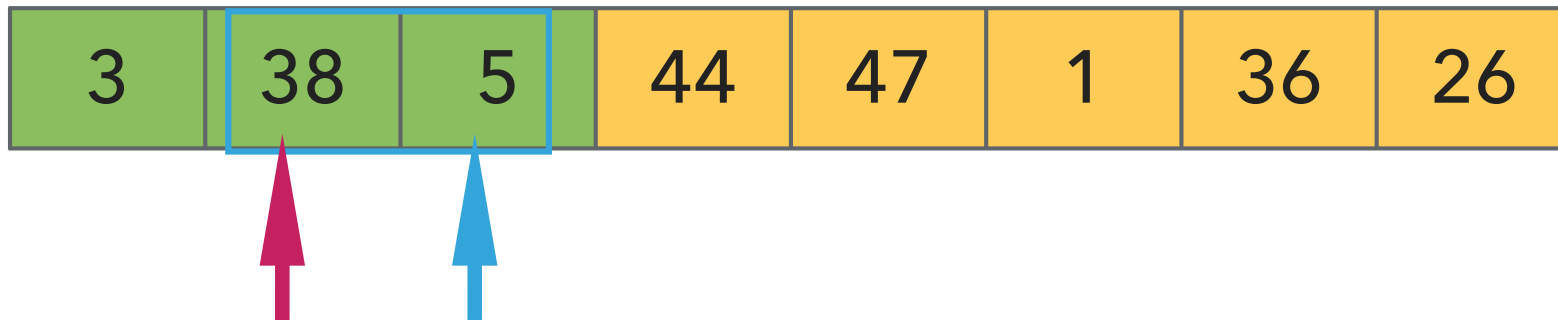
Insertion sort



- ▶ 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

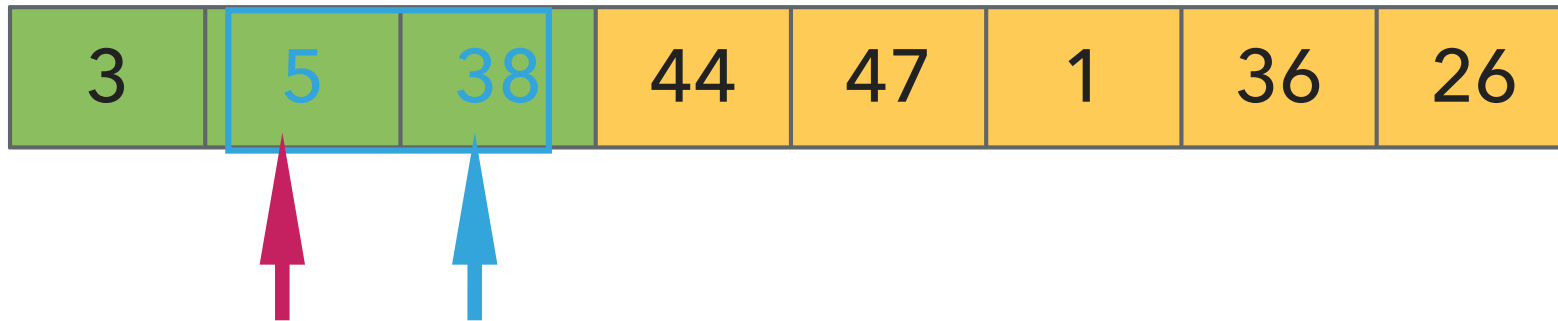
Insertion sort



- ▶ 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

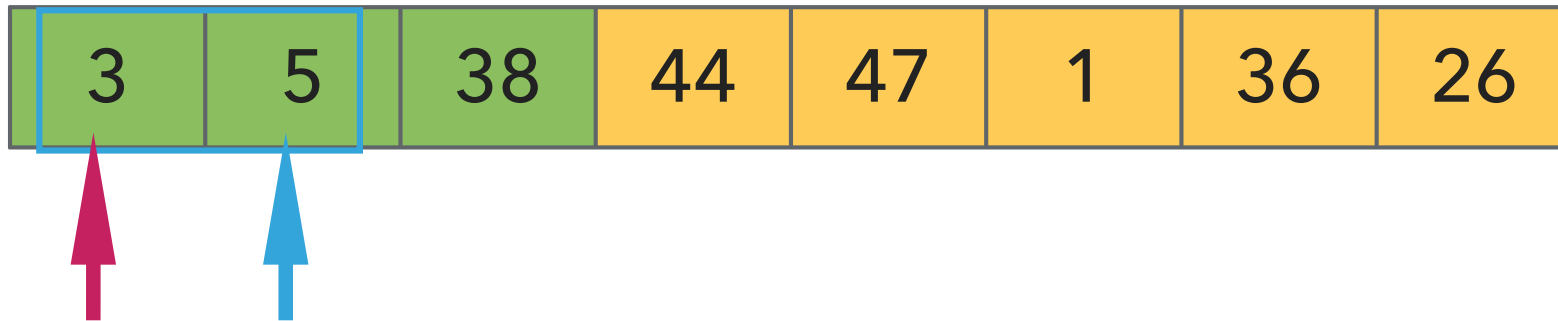
Insertion sort



- ▶ 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



► 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

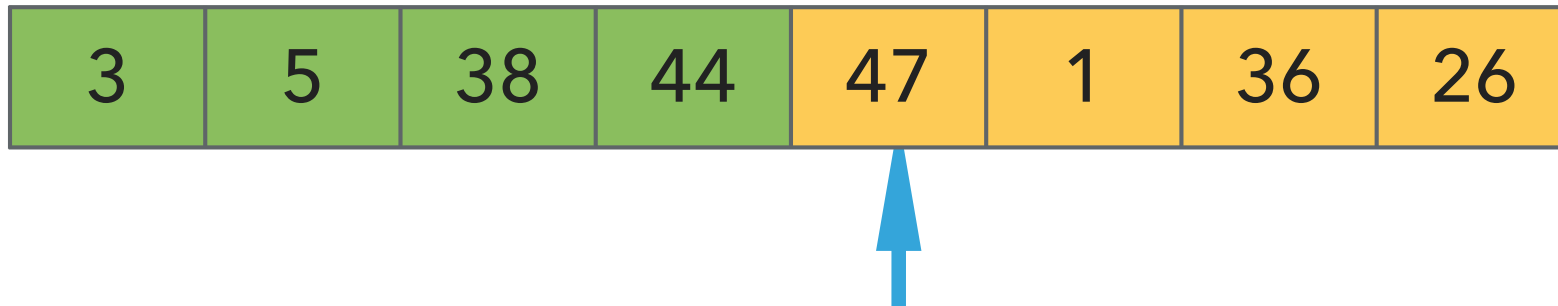
Insertion sort

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

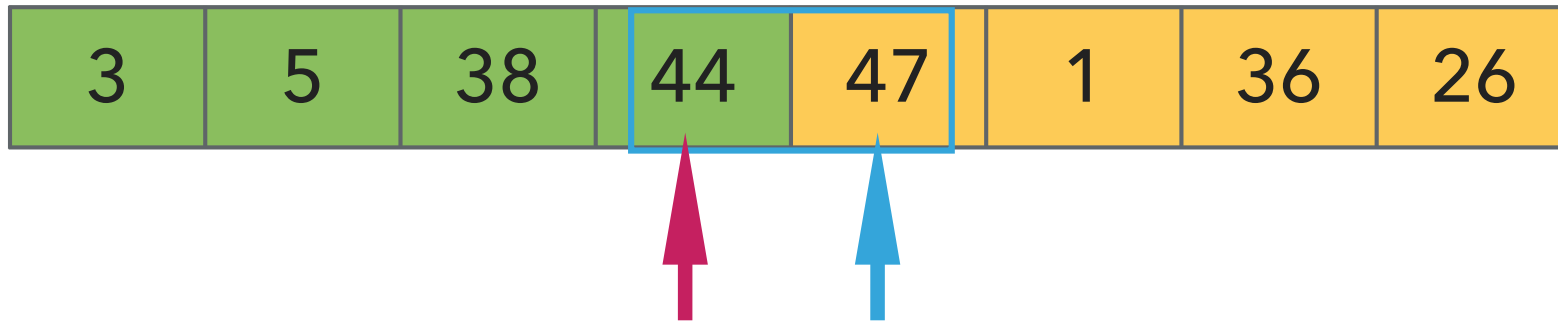
Insertion sort



- ▶ 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



- ▶ 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

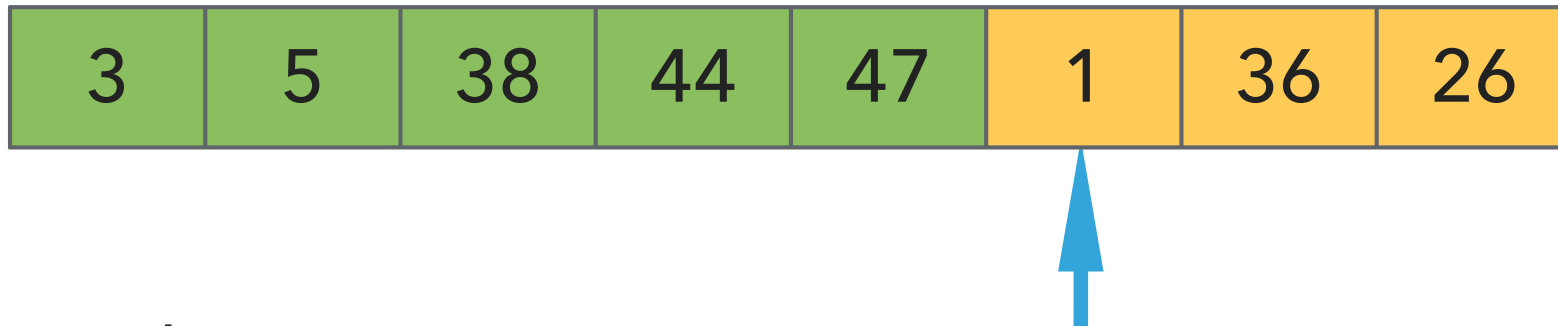
Insertion sort

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

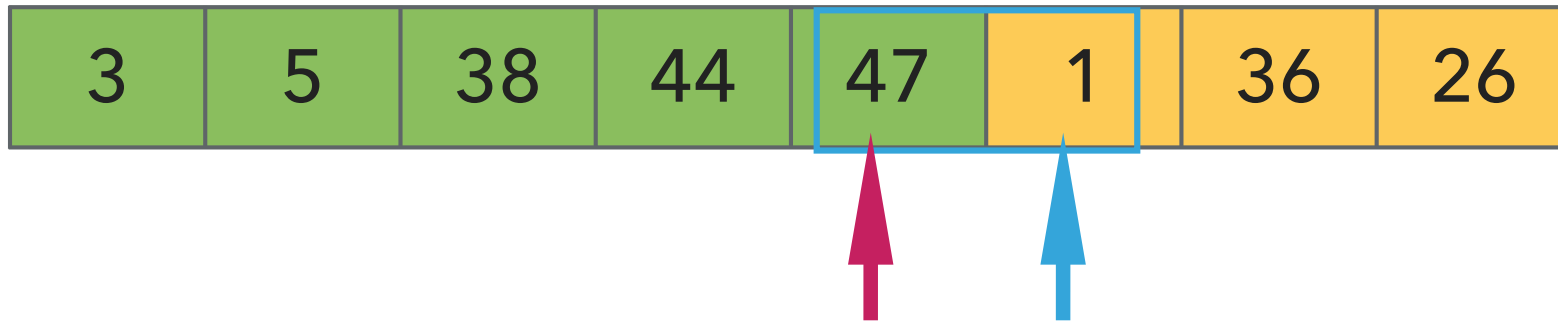
Insertion sort



- ▶ 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

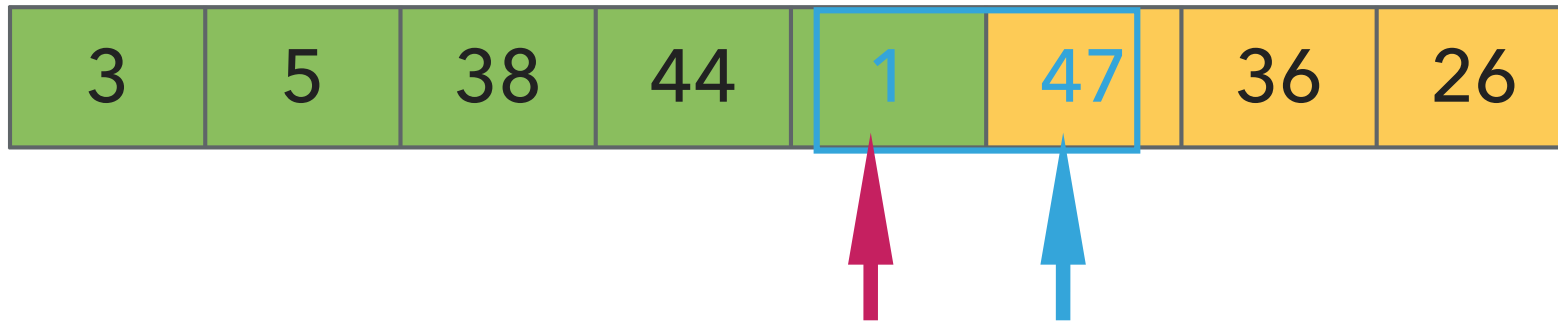
Insertion sort



- ▶ 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

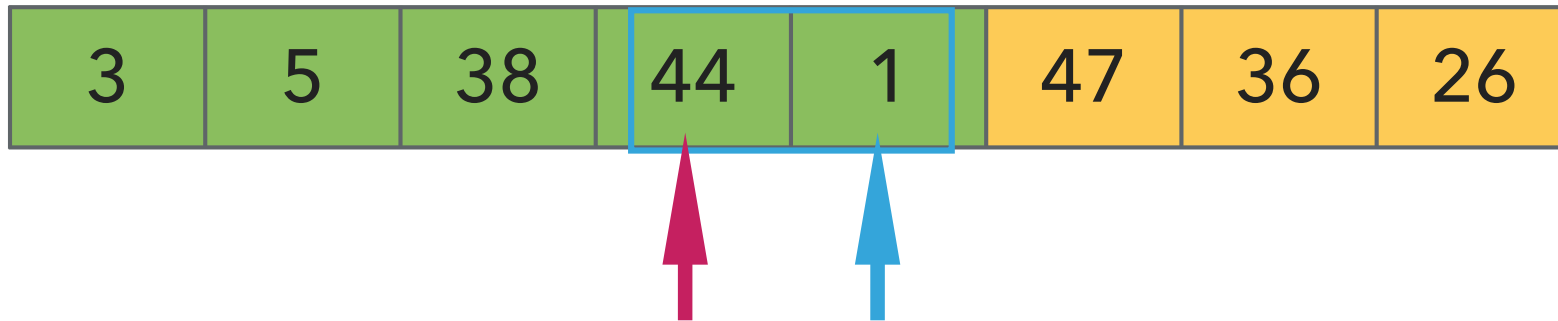
Insertion sort



- ▶ 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

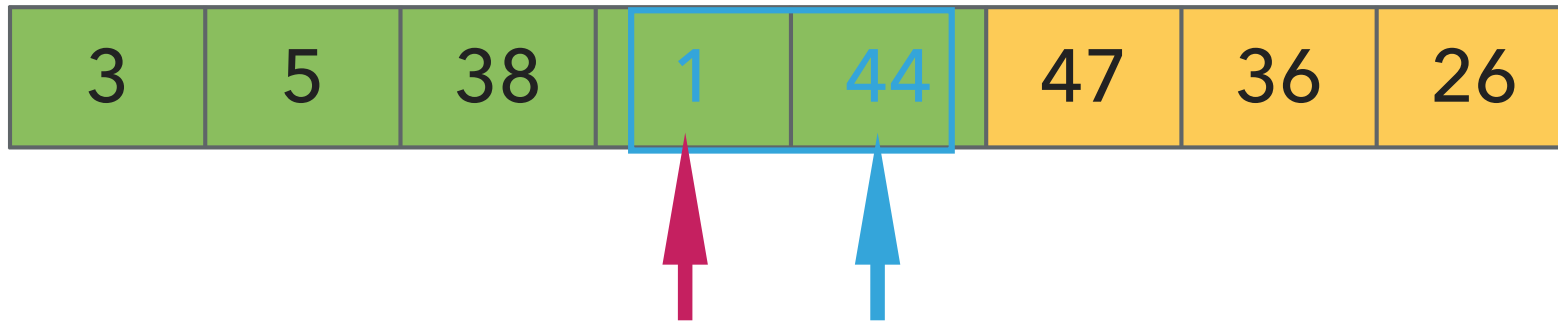
Insertion sort



- ▶ 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

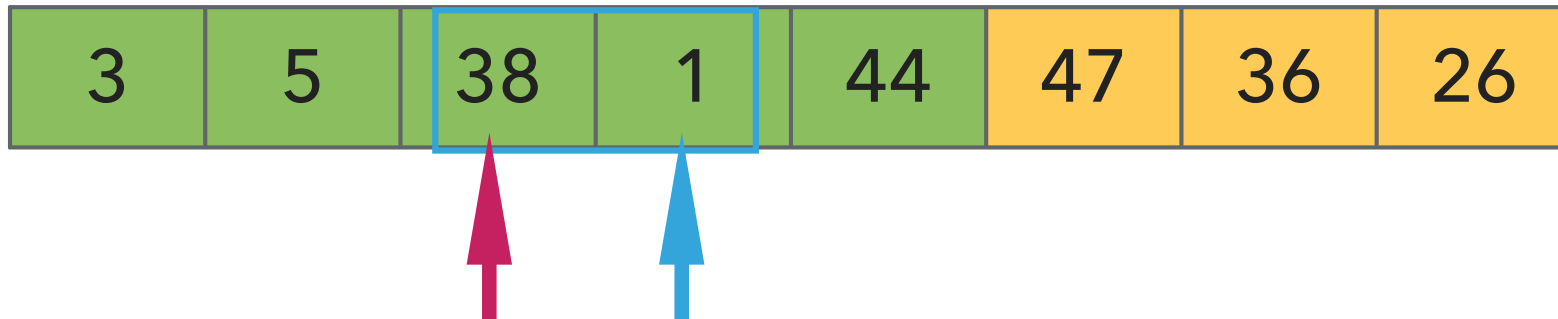
Insertion sort



- ▶ 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

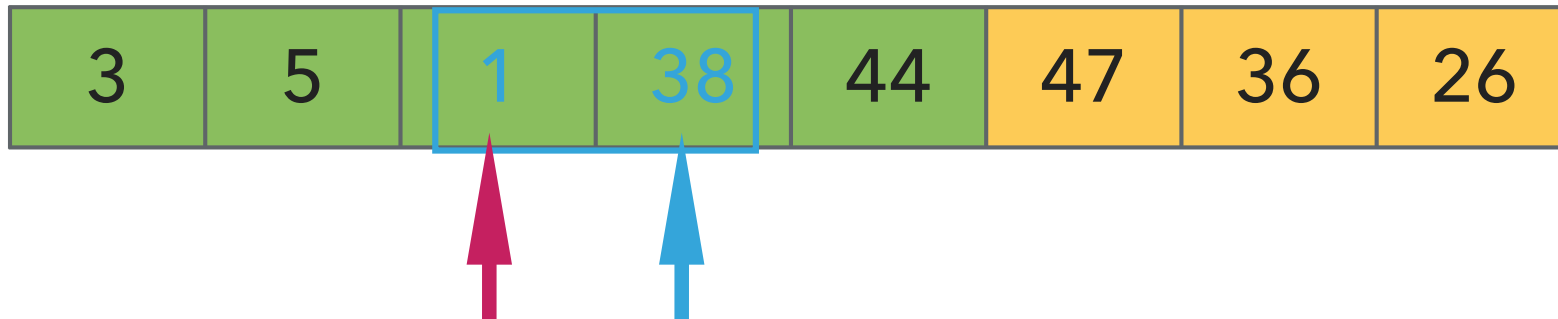
Insertion sort



- ▶ 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

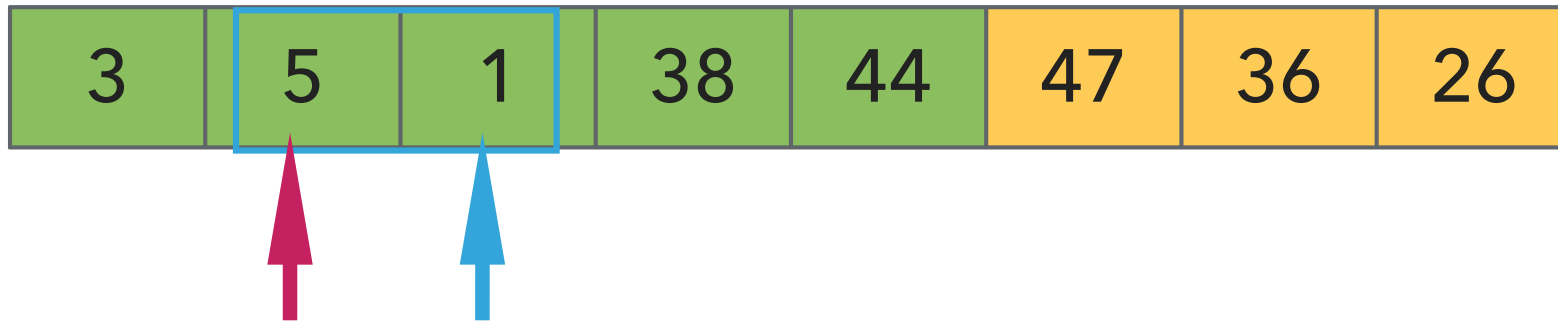
Insertion sort



- ▶ 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

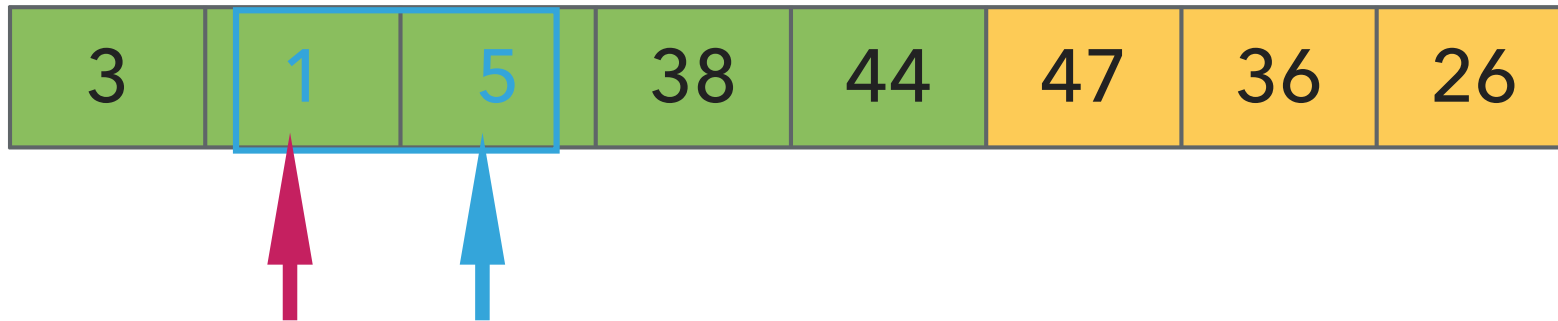
Insertion sort



- ▶ 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

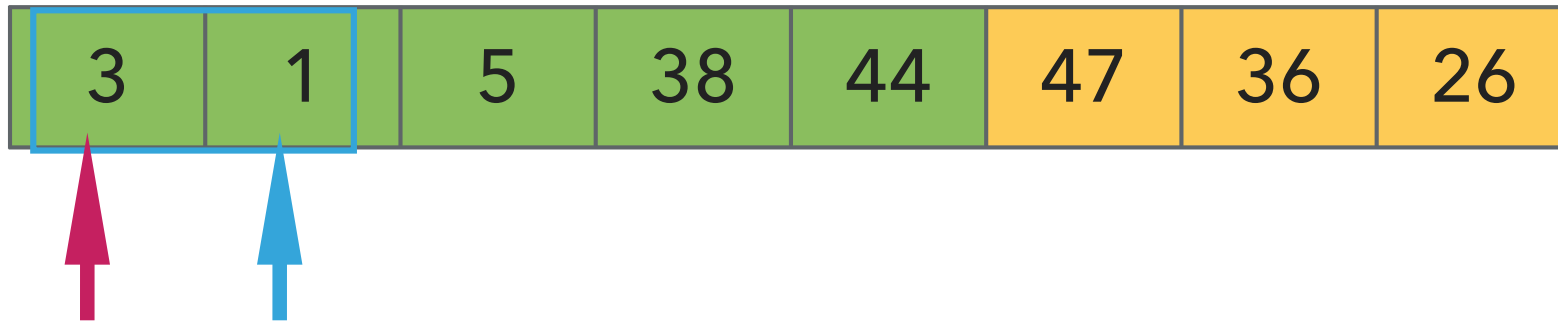
Insertion sort



- ▶ 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

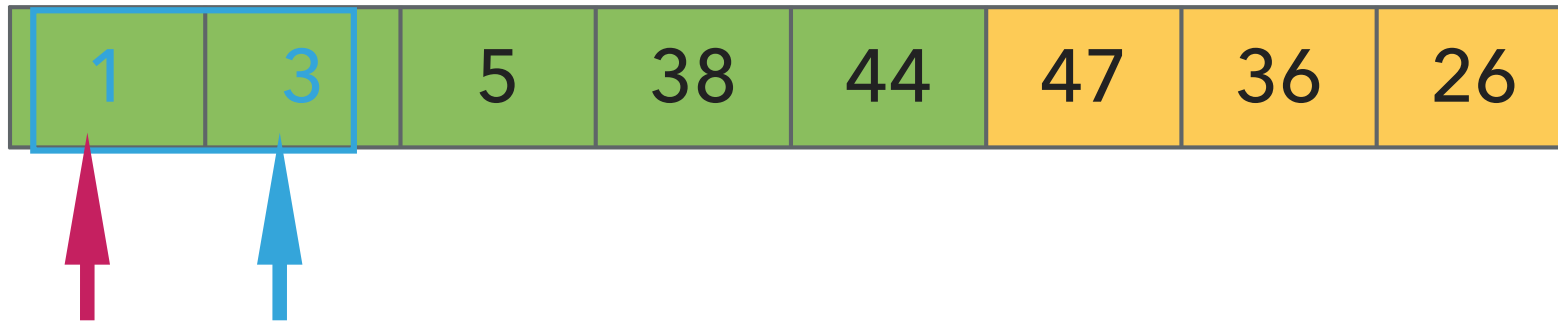


► Repeat:

- 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

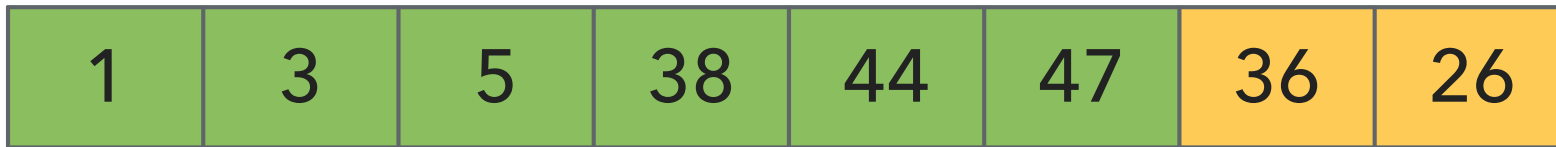
Insertion sort



- ▶ 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

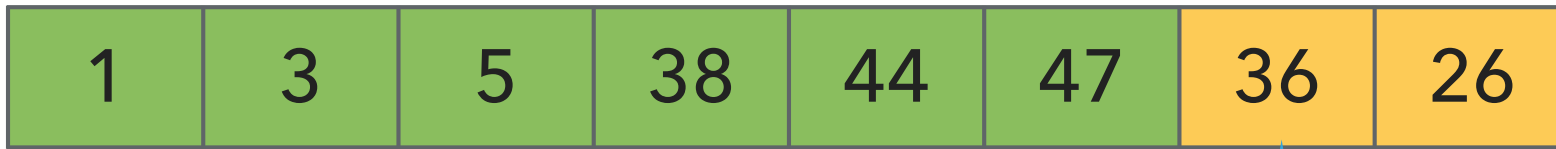
Insertion sort



- ▶ 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

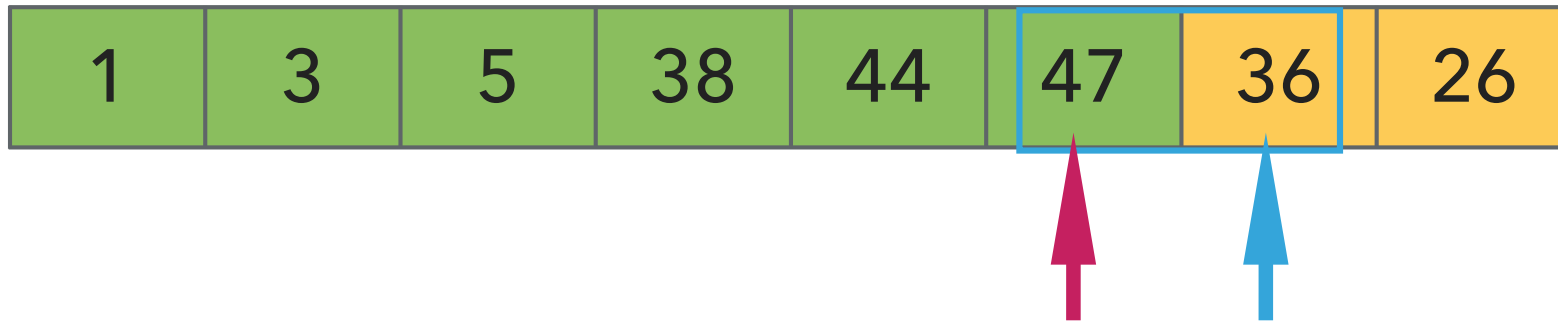
Insertion sort



- ▶ 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

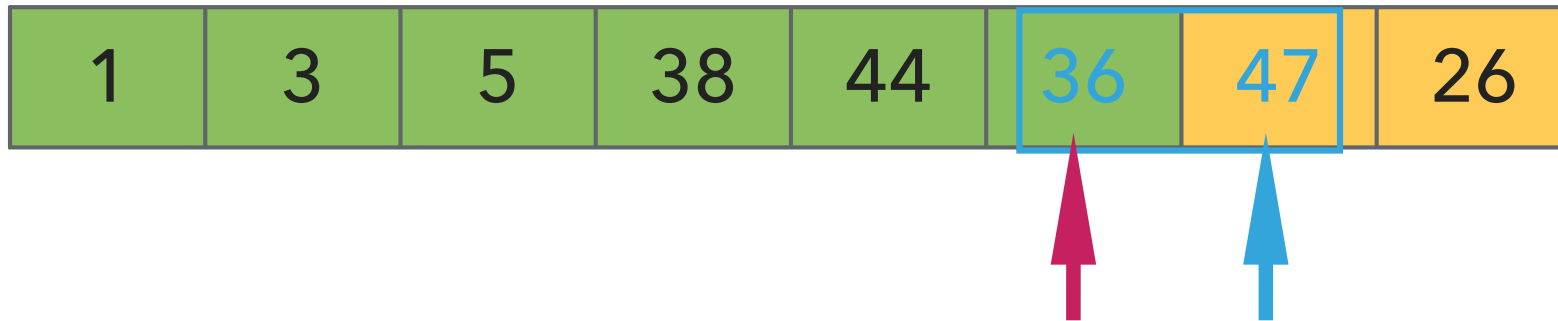
Insertion sort



- ▶ 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

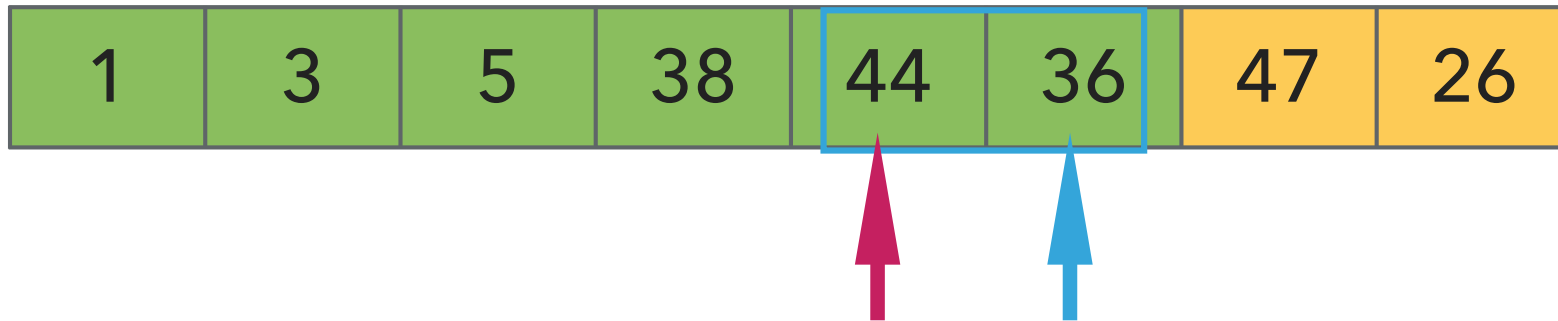
Insertion sort



- ▶ 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

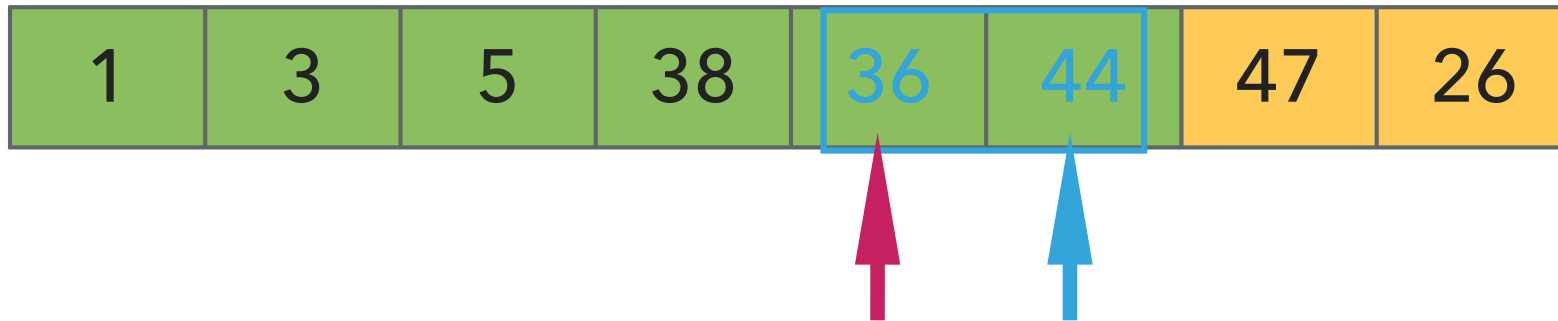
Insertion sort



- ▶ 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

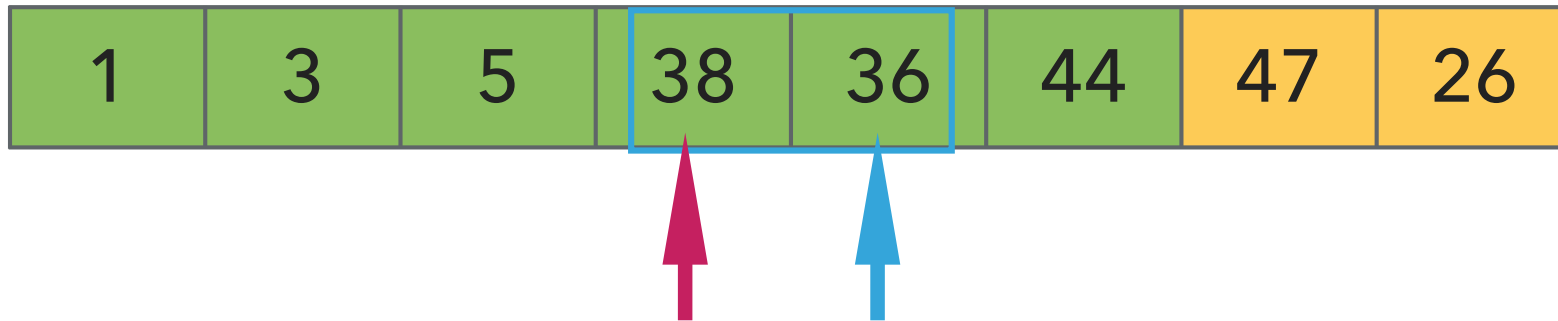
Insertion sort



- ▶ 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

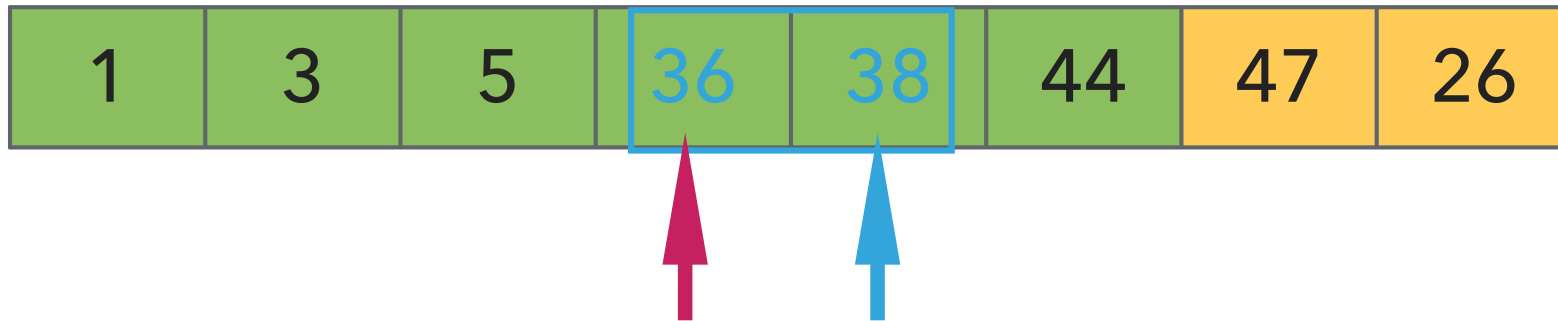
Insertion sort



- ▶ 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

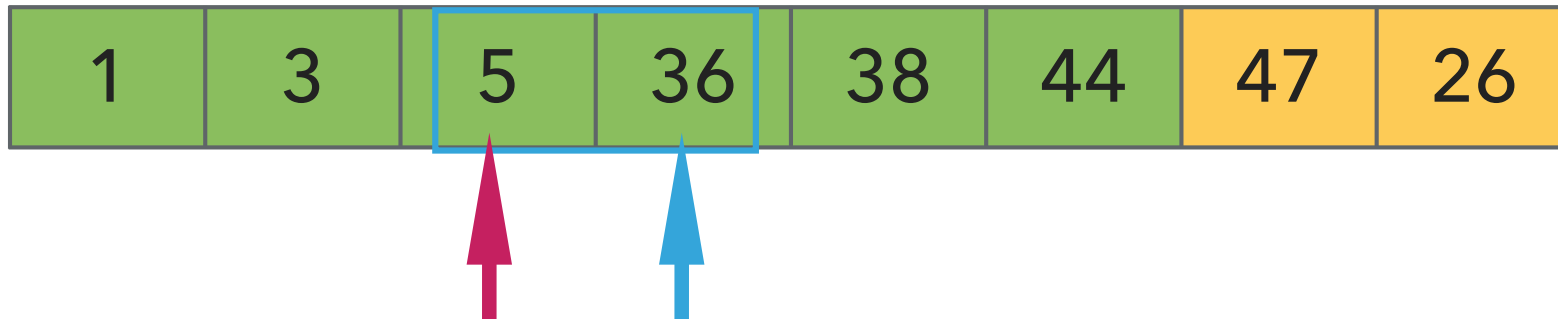
Insertion sort



- ▶ 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

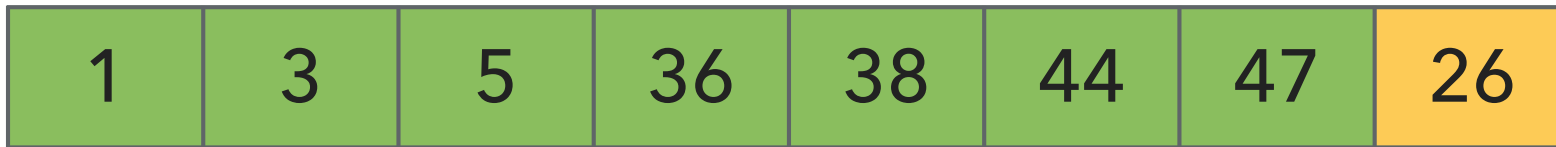
Insertion sort



- ▶ 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

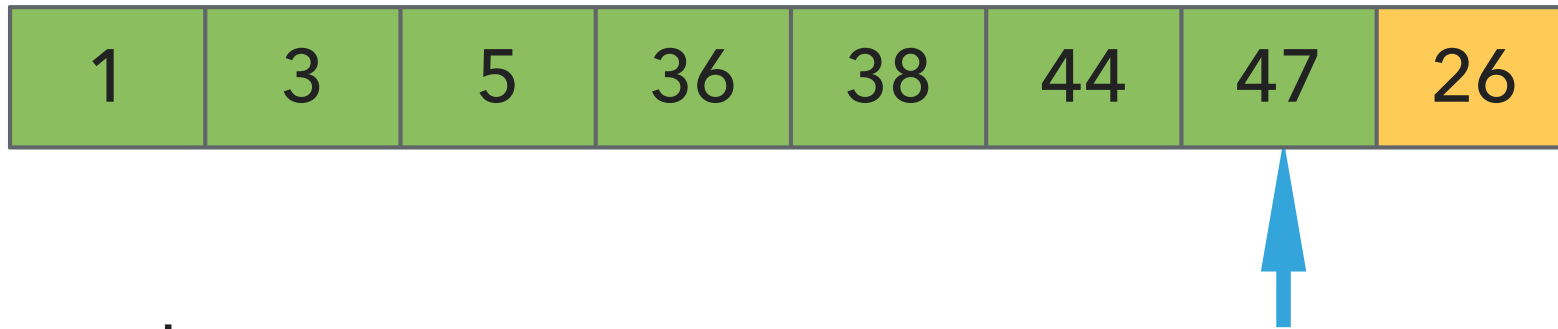
Insertion sort



- ▶ 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

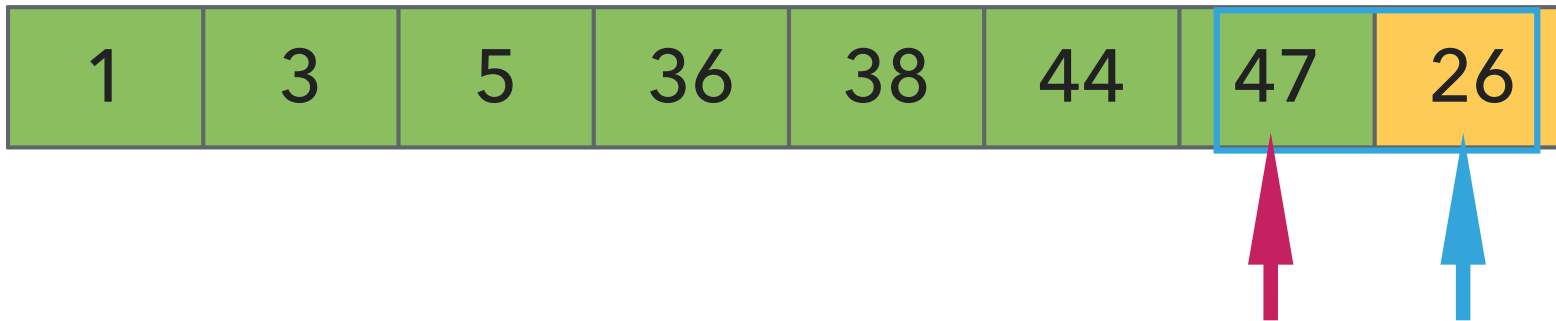
Insertion sort



- ▶ 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

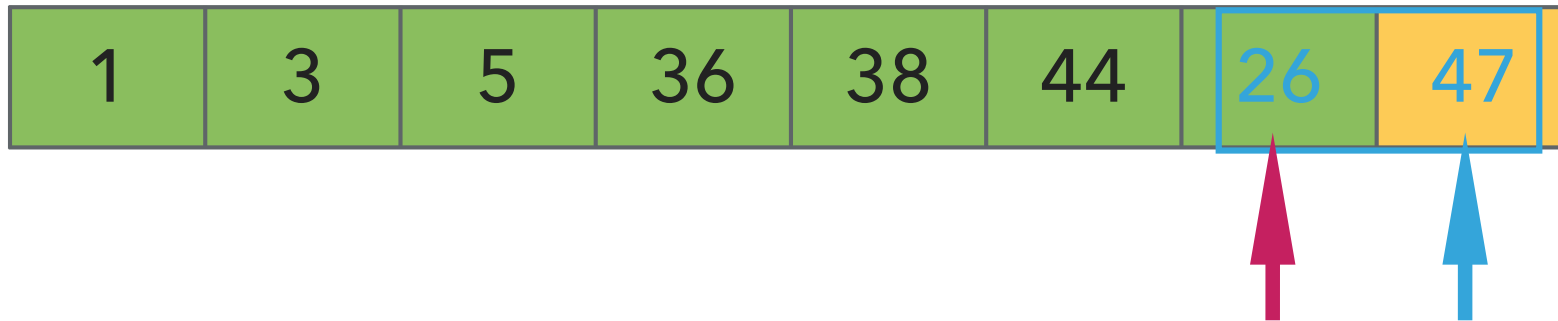
Insertion sort



- ▶ 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

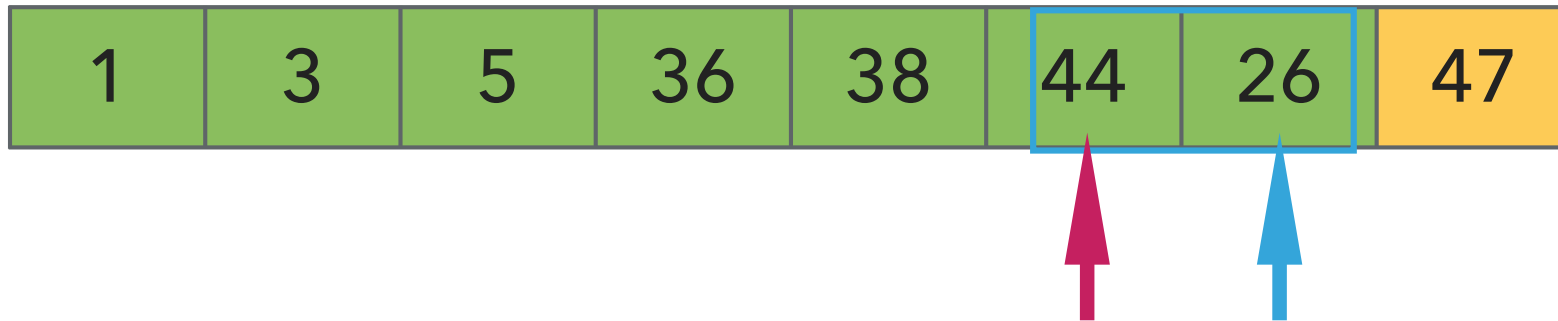
Insertion sort



- ▶ 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

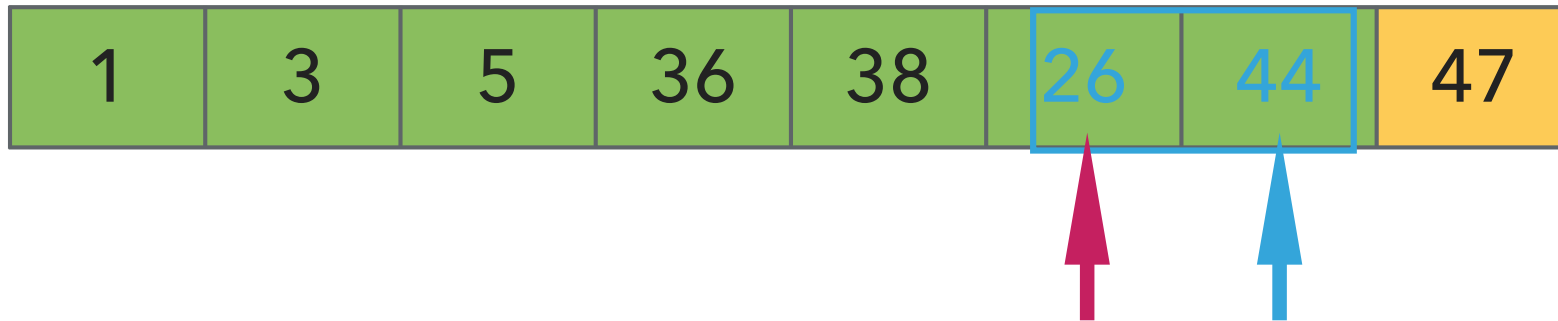
Insertion sort



- ▶ 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

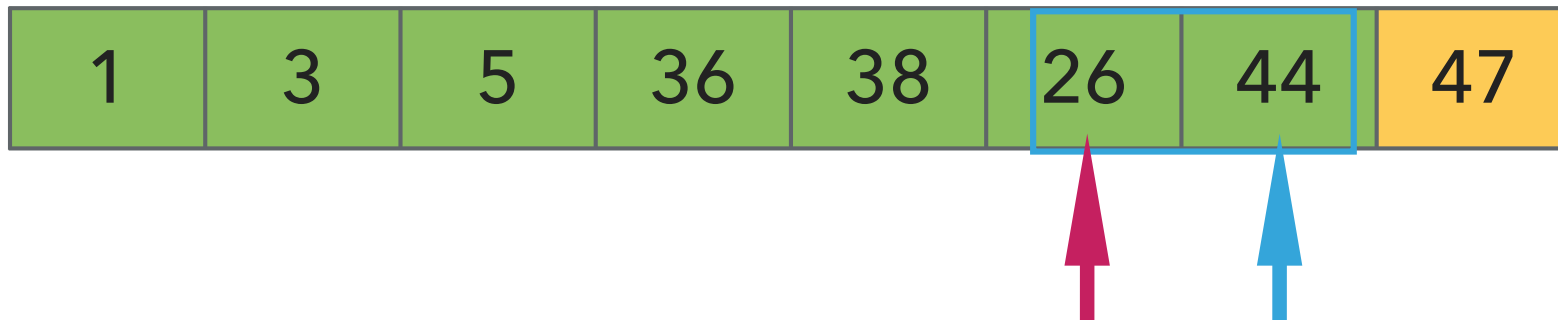
Insertion sort



- ▶ 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

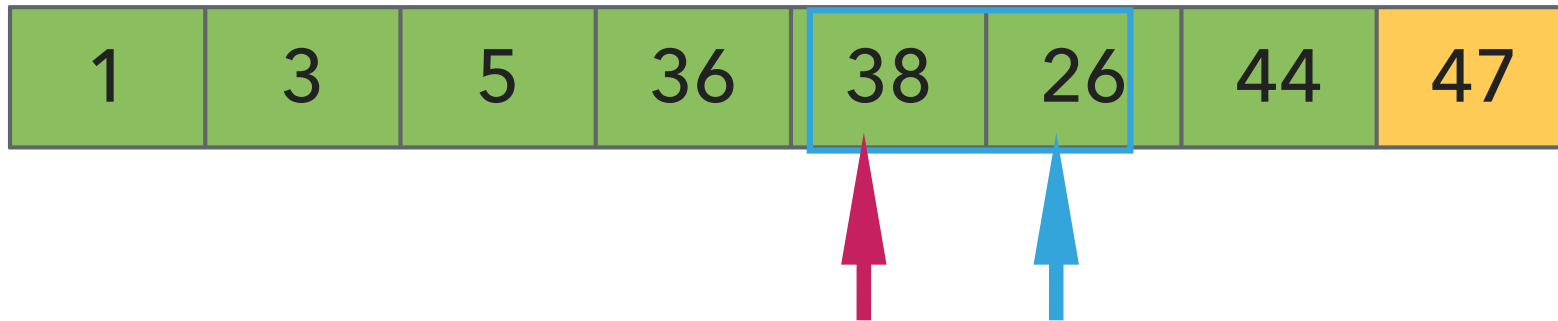
Insertion sort



- ▶ 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

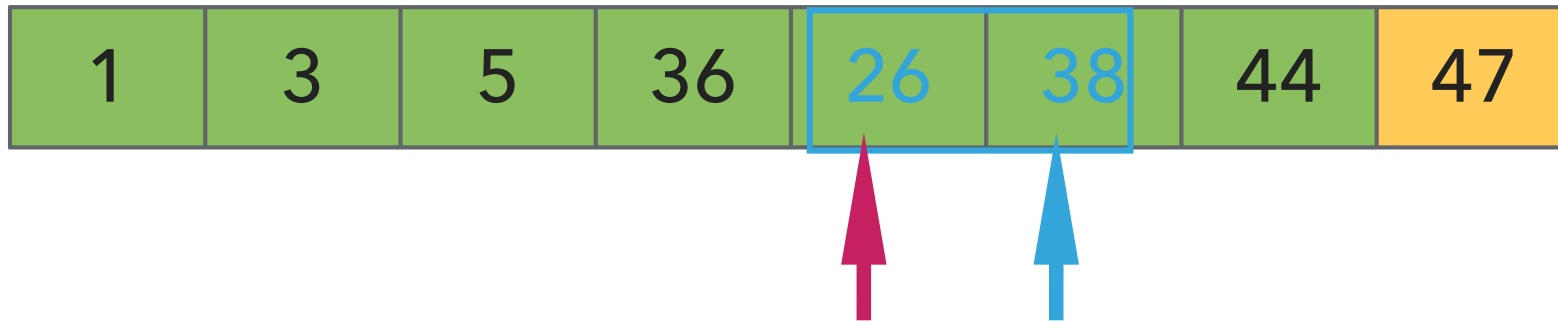
Insertion sort



- ▶ 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

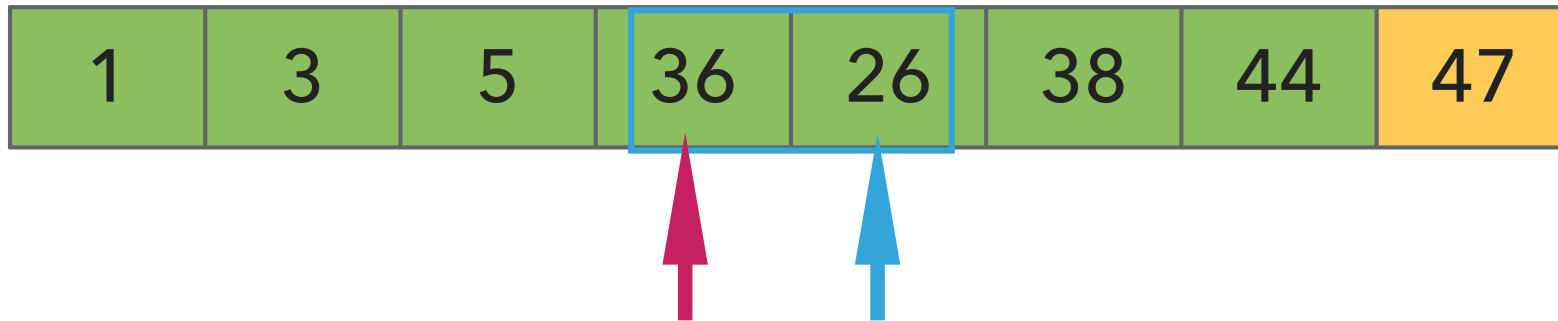
Insertion sort



- ▶ 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

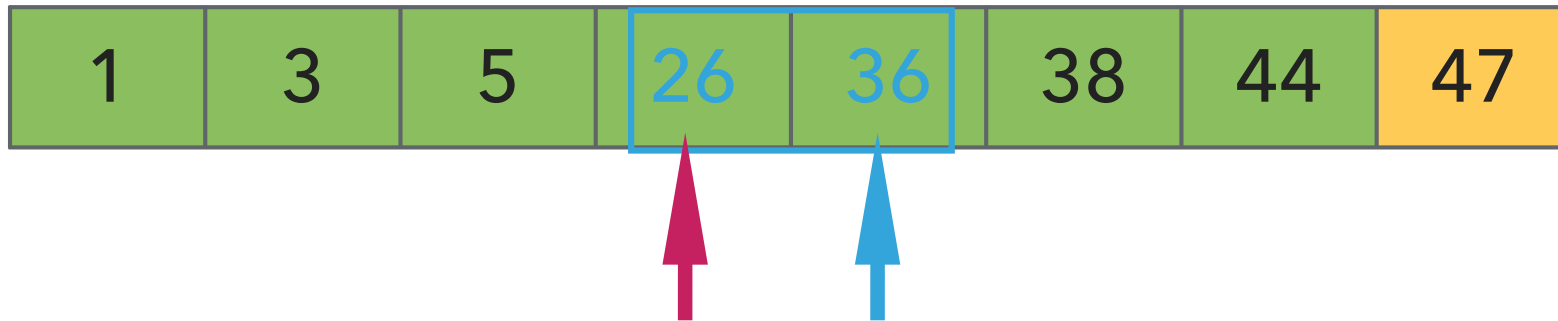
Insertion sort



- ▶ 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

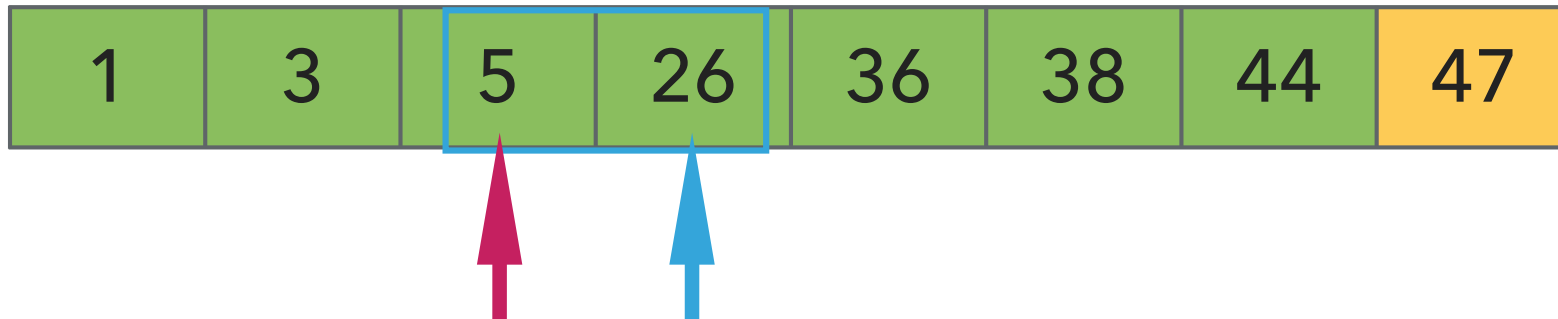
Insertion sort



- ▶ 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



- ▶ 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.



<http://algs4.cs.princeton.edu>

2.1 INSERTION SORT DEMO

INSERTION SORT

In case you didn't get this...

- <https://www.youtube.com/watch?v=ROaIU379I3U>

INSERTION SORT

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;  
        }  
    }  
}
```

► **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

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:** ?

► $0 + 1 + 2 + \dots + (n - 2) + (n - 1) \sim n^2/2$, that is $O(n^2)$.

► **Exchanges:** ?

► $0 + 1 + 2 + \dots + (n - 2) + (n - 1) \sim n^2/2$, that is $O(n^2)$.

► Worst-case running time is **quadratic**.

► **In-place**, requires almost no additional memory.

► **Stable**

INSERTION SORT

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;  
        }  
    }  
}
```

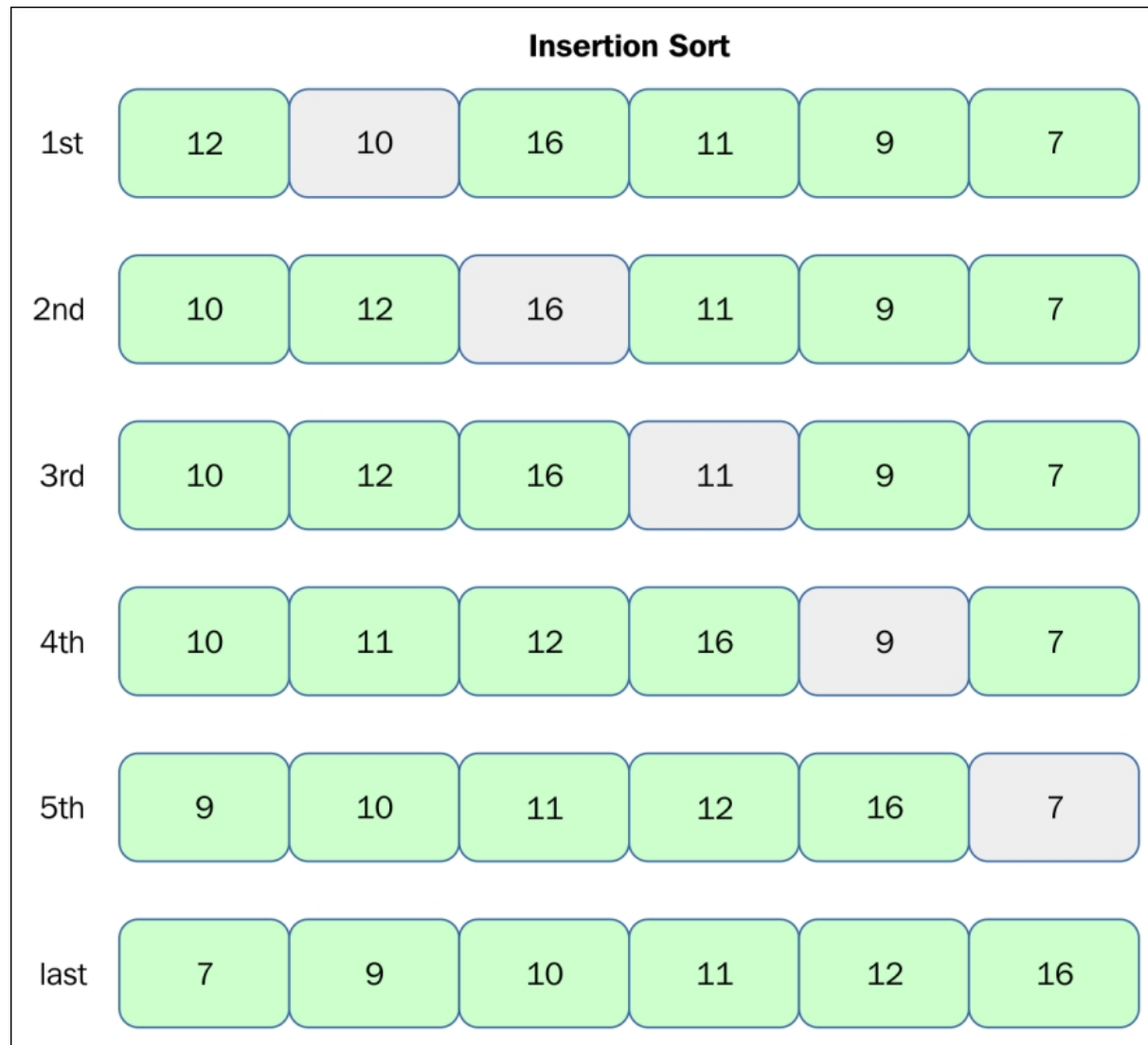
- ▶ **Average case:** quadratic for both comparisons and exchanges $\sim n^2/4$ when sorting a randomly ordered array.
- ▶ **Best case: When? Number of comparisons?**
- ▶ $n - 1$ comparisons and 0 exchanges for an already sorted array.

Practice Time

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

INSERTION SORT

Answer



Readings:

- ▶ Textbook:
 - ▶ Chapter 2.1 (pages 244–262)
- ▶ 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>

Practice Problems:

- ▶ 2.1.1–2.1.8

TEXT

Mergesort

► Mergesort

MERGESORT

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

79

Basics

Mergesort overview

- ▶ Invented by John von Neumann in 1945
- ▶ Algorithm sketch:
 - ▶ Divide array into two halves.
 - ▶ Recursively sort each half.
 - ▶ Merge the two halves



Merging two already sorted halves into one sorted array

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {  
    for (int k = lo; k <= hi; k++)  
        aux[k] = a[k];  
  
    int i = lo, j = mid+1;  
    for (int k = lo; k <= hi; k++) {  
        if (i > mid) //ran out of elements in the left subarray  
            a[k] = aux[j++];  
        else if (j > hi) //ran out of elements in the right subarray  
            a[k] = aux[i++];  
        else if (less(aux[j], aux[i]))  
            a[k] = aux[j++];  
        else  
            a[k] = aux[i++];  
    }  
}
```

Merging Example - 1st step - copying to auxiliary array

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid				hi			

Array a

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {  
    for (int k = lo; k <= hi; k++)  
        aux[k] = a[k];    // Copy a to aux array  
  
    int i = lo, j = mid+1;  
    for (int k = lo; k <= hi; k++) {  
        if (i > mid) //ran out of elements in the left subarray  
            a[k] = aux[j++];  
        else if (j > hi) //ran out of elements in the right subarray  
            a[k] = aux[i++];  
        else if (less(aux[j], aux[i]))  
            a[k] = aux[j++];  
        else  
            a[k] = aux[i++];  
    }  
}
```

Merging Example - k=0

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
i						j			
k									

case: $\text{aux}[i] < \text{aux}[j]$
 $a[0] = \text{aux}[0]$
 $i++;$

Array a

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=1

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
i		j							
k									

case: $\text{aux}[i] < \text{aux}[j]$
 $a[1] = \text{aux}[1]$
 $i++;$

Array a

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left half
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right half
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=2

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i			j				
k									

case: $\text{aux}[i] > \text{aux}[j]$
 $\text{a}[2] = \text{aux}[5]$
 $j++;$

Array a

A	G	H	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=3

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i			j				
k									

case: $\text{aux}[i] > \text{aux}[j]$
 $\text{a}[3] = \text{aux}[6]$
 $j++;$

Array a

A	G	H	I	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=4

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i			j				
k									

case: $\text{aux}[i] < \text{aux}[j]$
 $\text{a}[4] = \text{aux}[2]$
 $i++;$

Array a

A	G	H	I	L	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left half
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right half
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=5

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i			j				
k									

case: $\text{aux}[i] > \text{aux}[j]$
 $\text{a}[5] = \text{aux}[7]$
 $j++;$

Array a

A	G	H	I	L	M	I	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=6

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i						j	
k									

case: $\text{aux}[i] < \text{aux}[j]$
 $\text{a}[6] = \text{aux}[3]$
 $i++;$

Array a

A	G	H	I	L	M	O	M	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=7

Array aux									
A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
		i						j	
k									

case: $\text{aux}[i] < \text{aux}[j]$
 $\text{a}[7] = \text{aux}[4]$
 $i++;$

Array a									
A	G	H	I	L	M	O	R	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=8

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
						i	j		
						k			

case: $i > \text{mid}$
 $a[8] = \text{aux}[8]$
 $j++;$

Array a

A	G	H	I	L	M	O	R	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Merging Example - k=9

Array aux

A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo		mid						hi	
						i	j		
k									

case: $i > \text{mid}$
 $a[9] = \text{aux}[9]$
 $j++;$

Array a

A	G	H	I	L	M	O	R	S	T
0	1	2	3	4	5	6	7	8	9

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left array
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right array
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

2.2 MERGING DEMO



<http://algs4.cs.princeton.edu>

Practice time

How many calls does `merge()` make to `less()` in order to merge two already sorted subarrays, each of length $n/2$ into a sorted array of length n ?

A. $\sim 1/4n$ to $\sim 1/2n$

B. $\sim 1/2n$

C. $\sim 1/2n$ to n

D. $\sim n$

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left subarray
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right subarray
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Answer

How many calls does `merge()` make to `less()` in order to merge two already sorted subarrays, each of length $n/2$ into a sorted array of length n ?

C. $\sim(1/2)n$ to n , that is at most $n - 1$ or $O(n)$

Best case example

Merging $[1,2,3]$ and $[4,5,6]$ requires 3 calls to `less()`
(1 with 4, 2 with 4, 3 with 4). - exhaust 1 side = $(1/2)n$

Worst case example

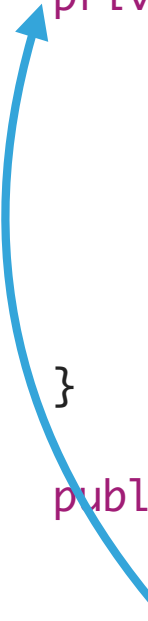
Merging $[1,3,5]$ and $[2, 4, 6]$ requires 5 calls to `less()`
(1 with 2, 2 with 3, 3 with 4, 4 with 5, 5 with 6)

TEXT

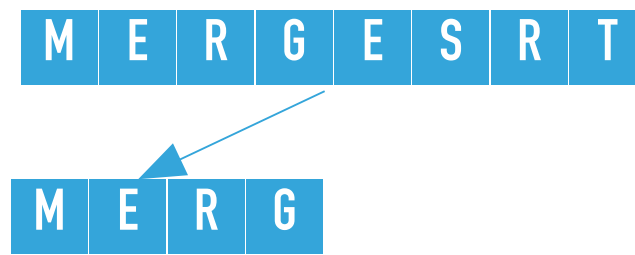
Mergesort - the quintessential example of divide-and-conquer

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}  
  
public static void sort(Comparable[] a) {  
    Comparable[] aux = new Comparable[a.length];  
    sort(a, aux, 0, a.length - 1);  
}
```

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}  
  
public static void sort(Comparable[] a) {  
    Comparable[] aux = new Comparable[a.length];  
    sort(a, aux, 0, a.length - 1);  
}
```

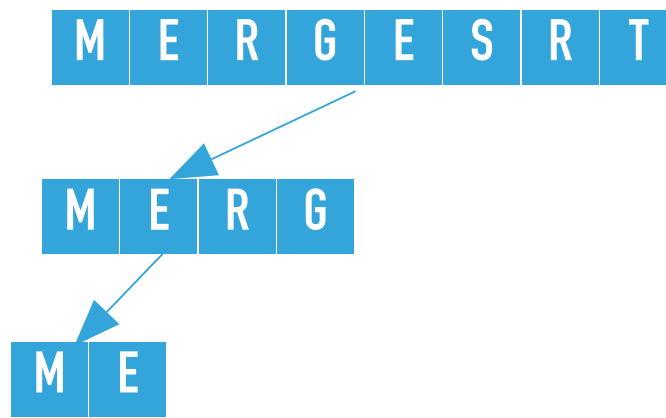


sort([M, E, R, G, E, S, R, T]) calls
sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0,
7) where the array of nulls is the auxiliary array, lo = 0 and hi = 7.



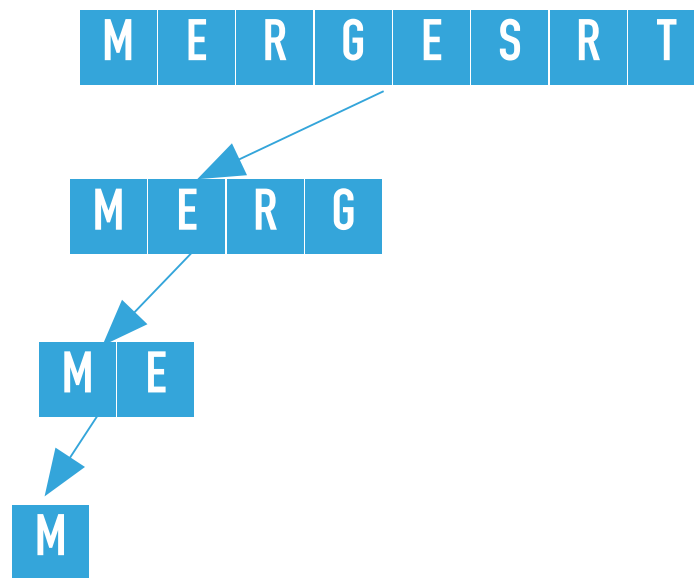
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 7)` calculates the `mid = 3` and calls recursively `sort` on the left subarray, that is `sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 3)`, where `lo = 0, hi = 3`



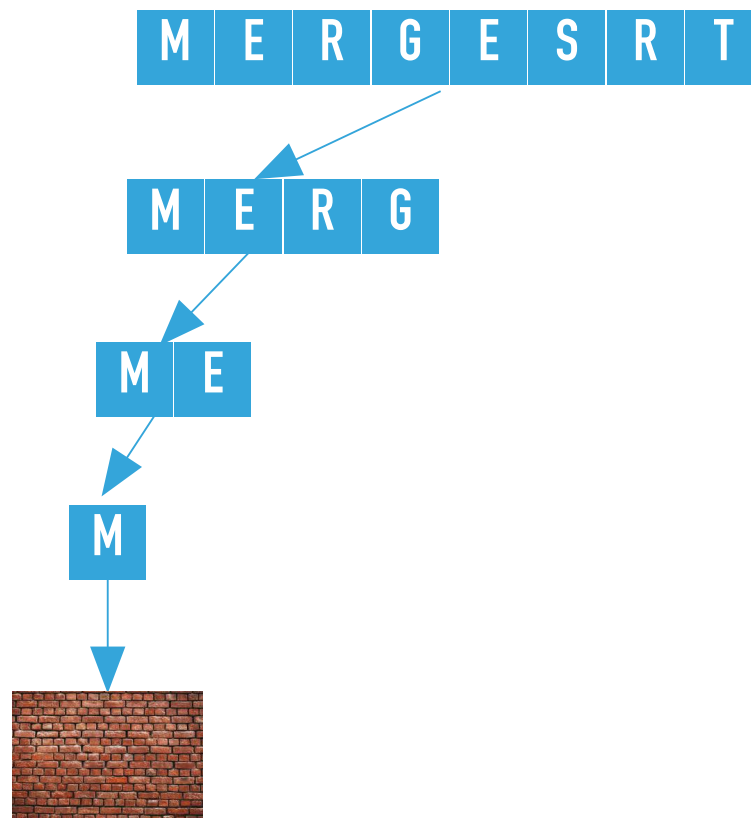
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 3)` calculates the `mid = 1` and calls recursively `sort` on the left subarray, that is `sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1)`, where `lo = 0, hi = 1`



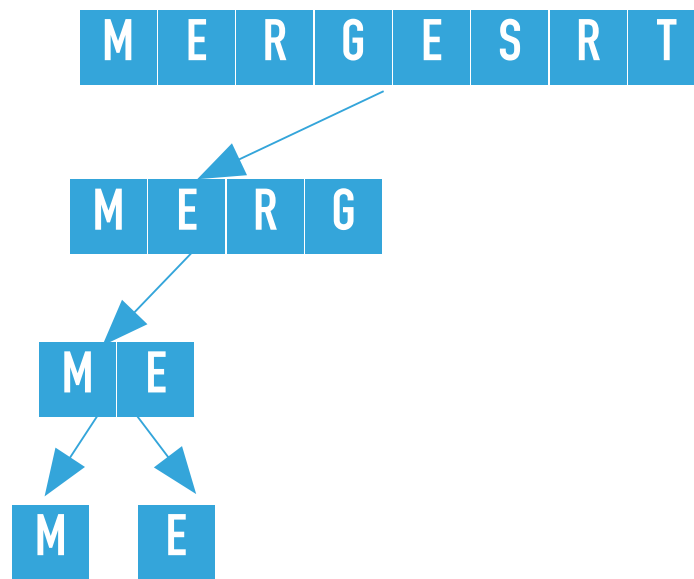
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1)` calculates the `mid = 0` and calls recursively `sort` on the left subarray, that is `sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 0)`, where `lo = 0, hi = 0`



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

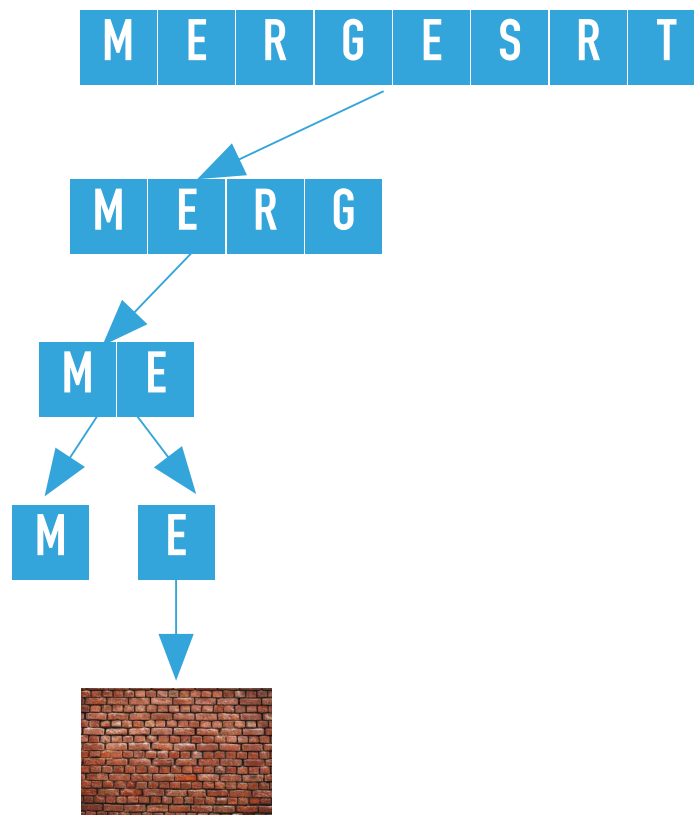
`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 0)` finds `hi <= lo` and returns.



```

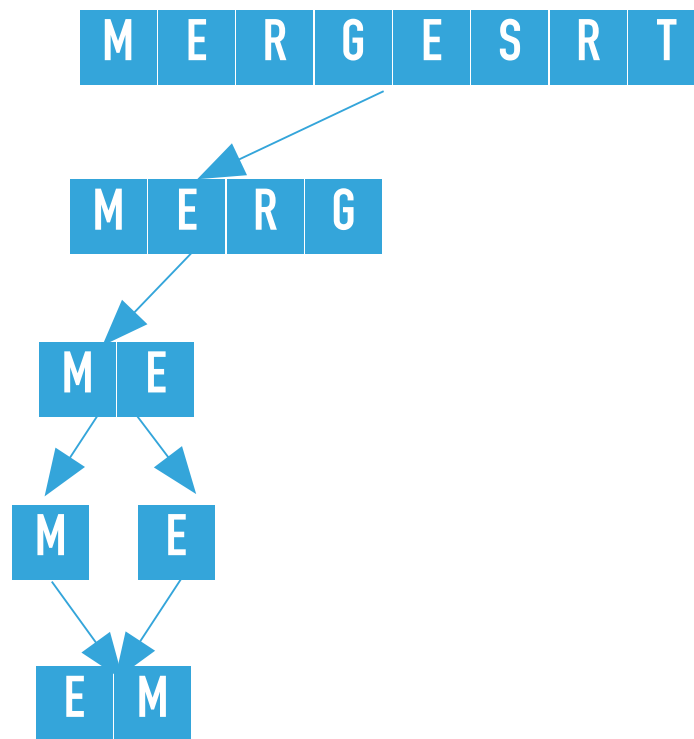
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
  
```

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1) calls recursively sort on the right subarray, that is sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 1, 1), where lo = 1, hi = 1



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 1, 1)` finds `hi <= lo` and returns.

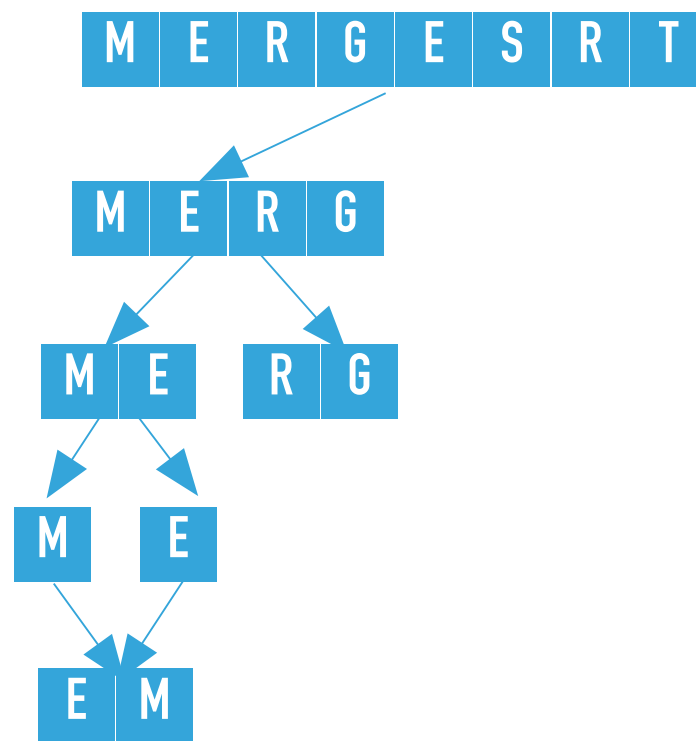


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

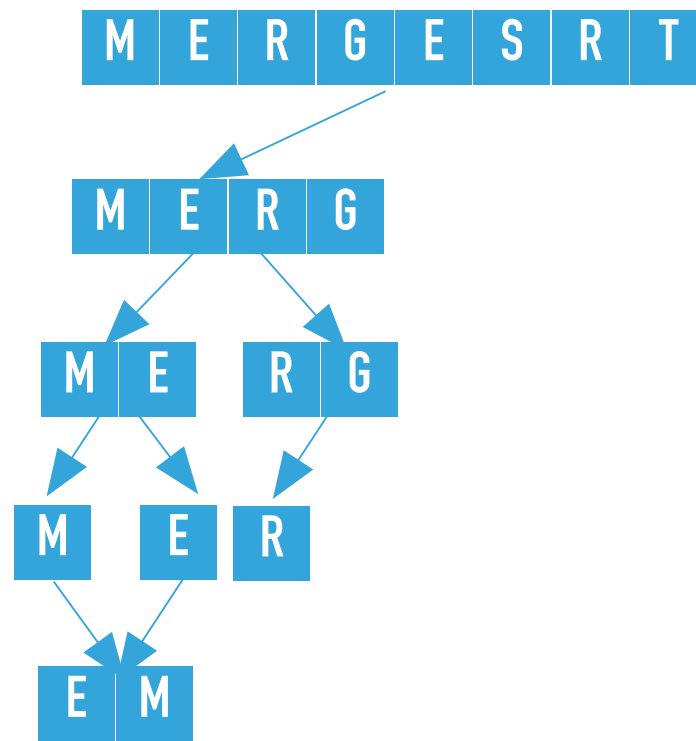
```

`sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1)` merges the two subarrays that it calls `merge([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 0, 1)`, where `lo = 0`, `mid = 0`, and `hi = 1`. The resulting partially sorted array is `[E, M, R, G, E, S, R, T]`.



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 0, 3)` calls recursively `sort` on the right subarray, that is `sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 3)`, where `lo = 2, hi = 3`

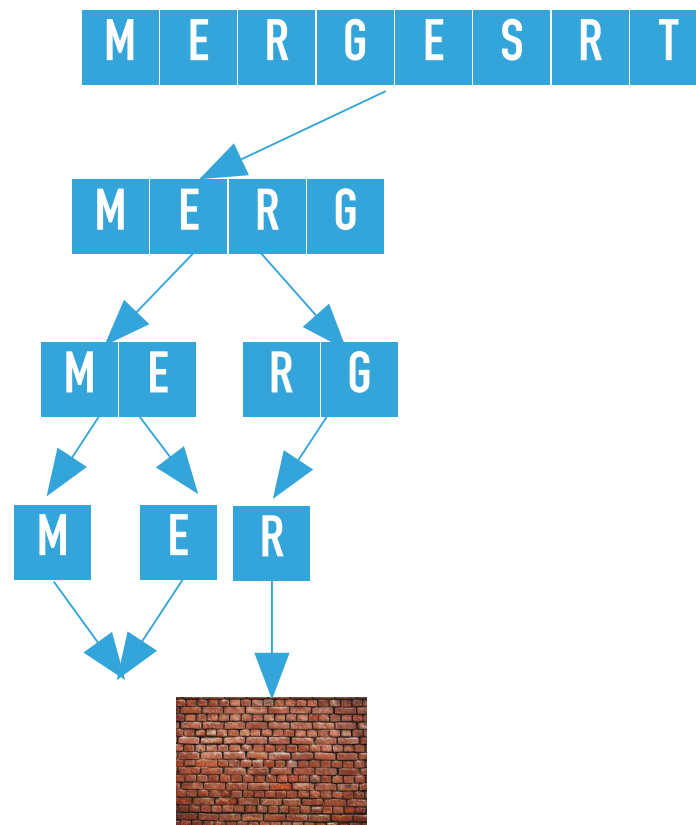


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 3)
 calculates the `mid = 2` and calls recursively `sort` on the left subarray, that is `sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 2)`, where `lo = 2, hi = 2`

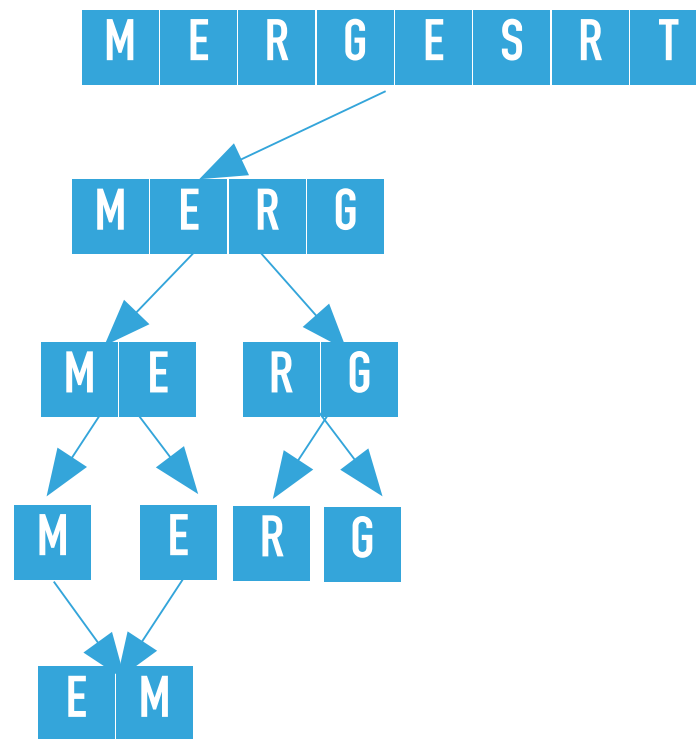


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 2) finds $hi \leq lo$ and returns.

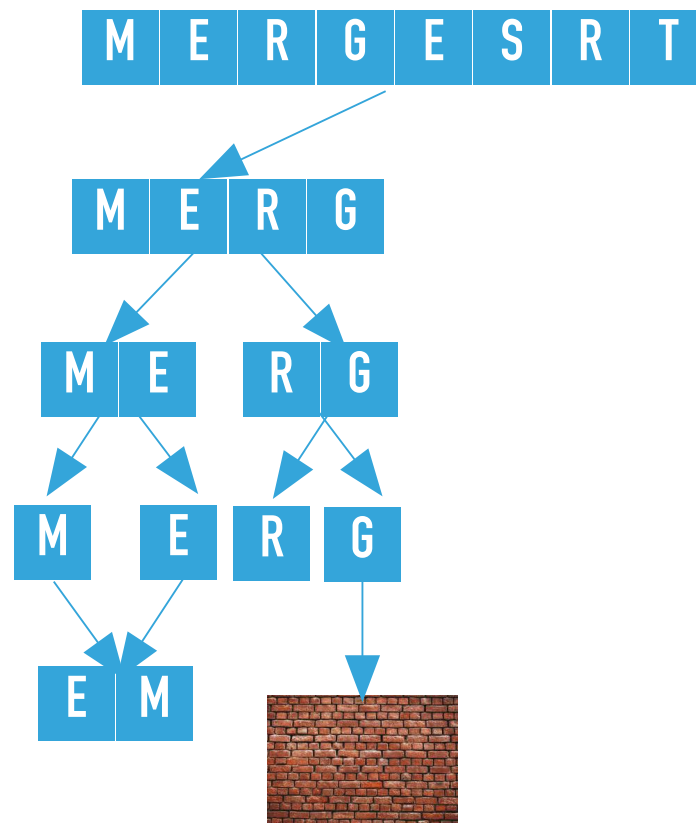


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

`sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 3)` calls recursively `sort` on the right subarray, that is `sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 3, 3)`, where `lo = 3, hi = 3`

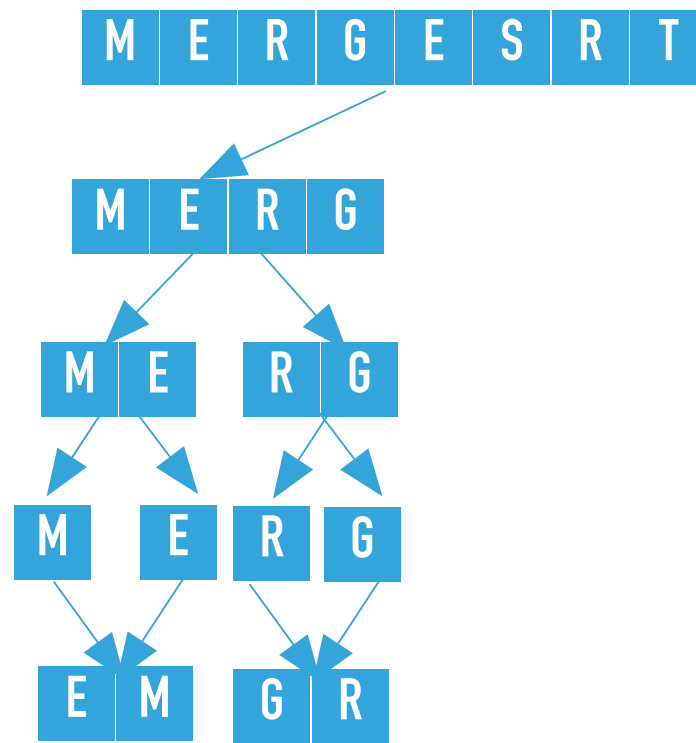


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 3, 3) finds $hi \leq lo$ and returns.

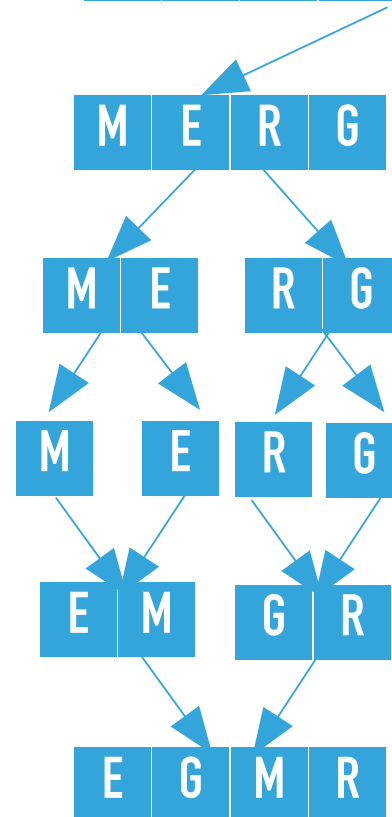


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

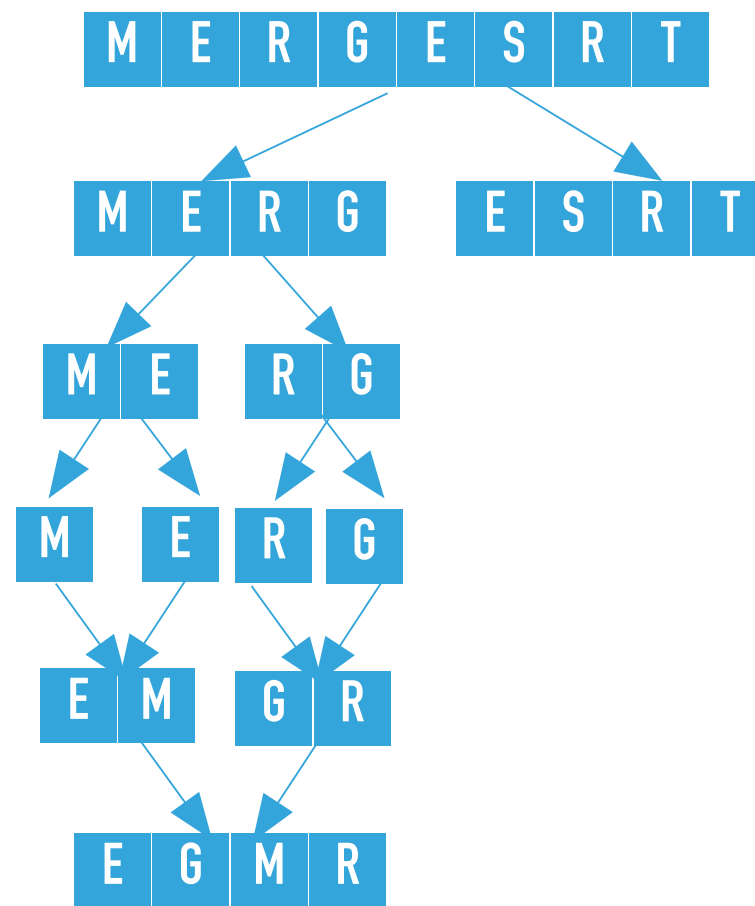
```

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 3)
 merges the two subarrays that is calls merge([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 2, 3), where lo = 2, mid = 2, and hi = 3. The resulting partially sorted array is [E, M, G, R, E, S, R, T].



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`sort([E, M, G, R, E, S, R, T], [M, E, R, G, null, null, null, null], 0, 3)` merges the two subarrays that is calls `merge([E, M, G, R, E, S, R, T], [M, E, R, G, null, null, null, null], 0, 1, 3)`, where `lo = 0`, `mid = 1`, and `hi = 3`. The resulting partially sorted array is `[E, G, M, R, E, S, R, T]`.



```

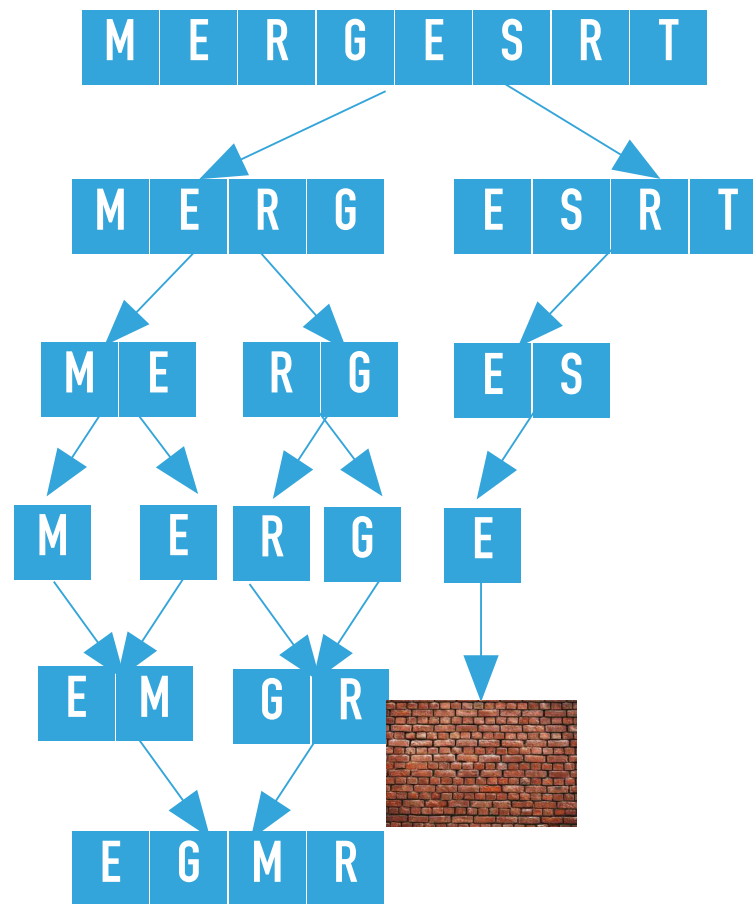
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 0, 7)` calls recursively `sort` on the right subarray, that is `sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 7)`, where `lo = 4, hi = 7`

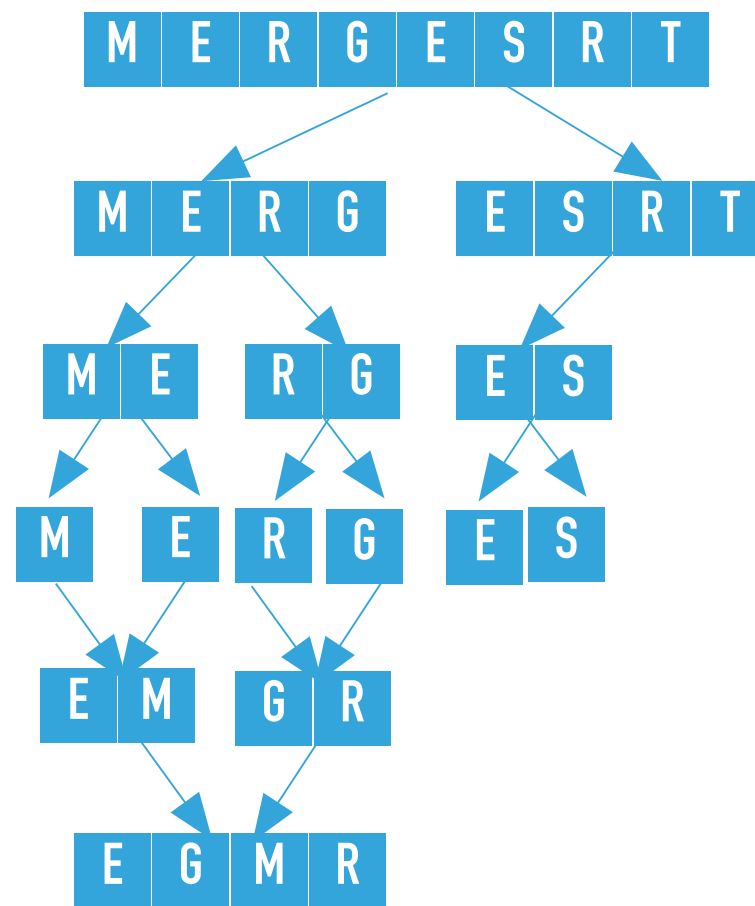
`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 7)` calculates the `mid = 5` and calls recursively `sort` on the left subarray, that is `sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)`, where `lo = 4, hi = 5`.

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` calculates the `mid = 4` and calls recursively `sort` on the left subarray, that is `sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)`, where `lo = 4, hi = 4`.



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)` finds `hi <= lo` and returns.



```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

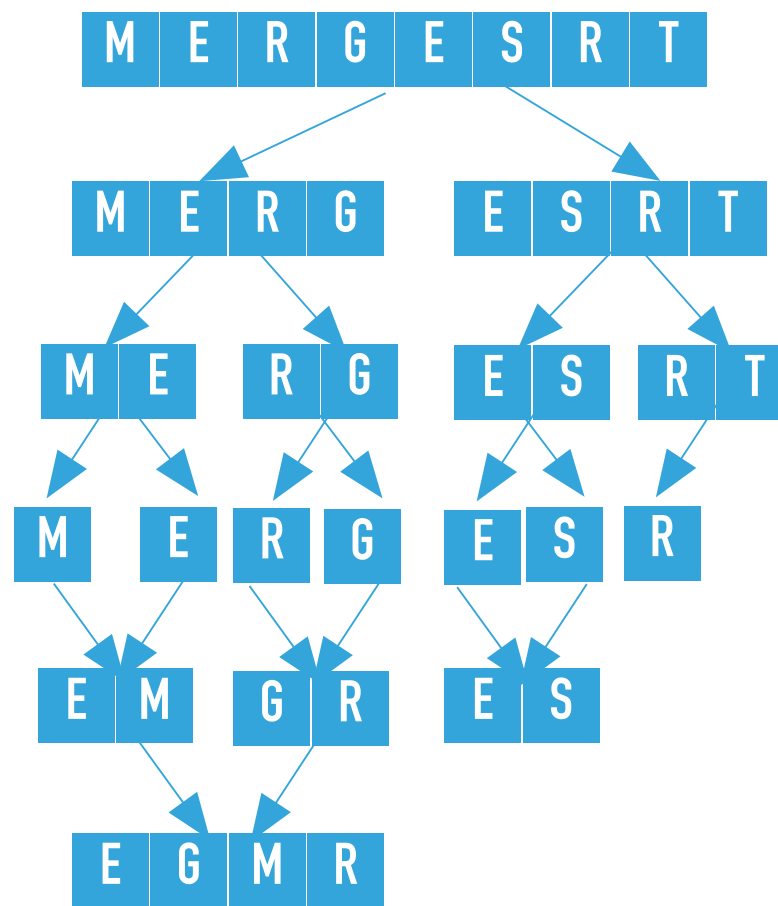
```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` calls recursively `sort` on the right subarray, that is `sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 5, 5)`, where `lo = 5, hi = 5`

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 5, 5)` finds `hi <= lo` and returns.

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` merges the two subarrays that is calls `merge([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4, 5)`, where `lo = 4`, `mid = 4`, and `hi = 5`. The resulting partially sorted array is `[E, G, M, R, E, S, R, T]`.

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 4, 7)` calls recursively `sort` on the right subarray, that is `sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 7)`, where `lo = 6, hi = 7`

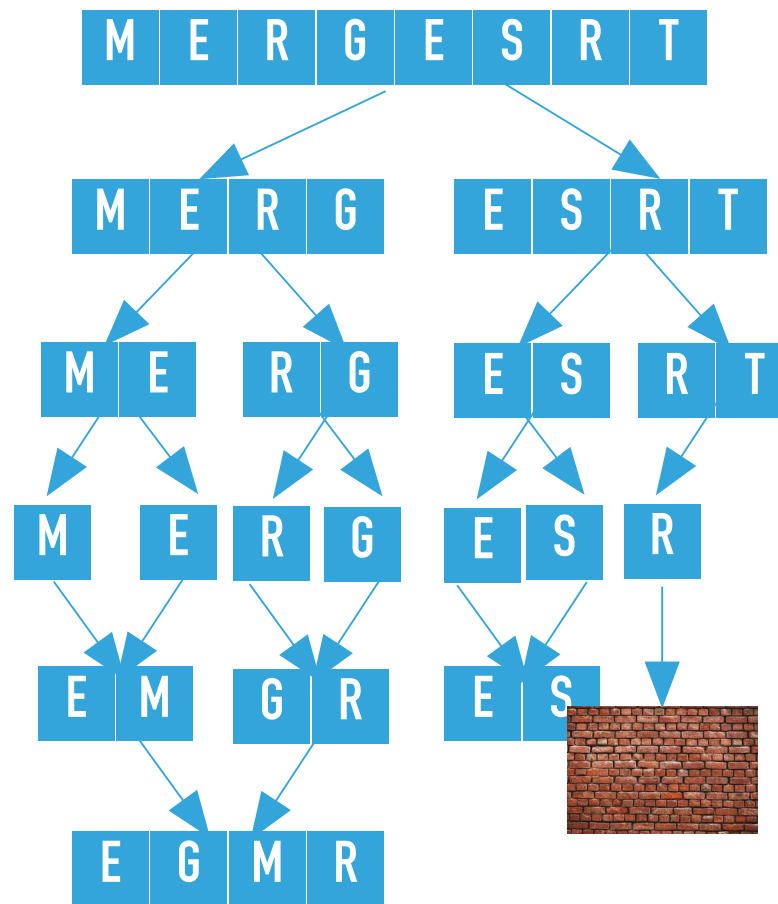


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 7) calculates the mid = 6 and calls recursively sort on the left subarray, that is sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 6), where lo = 6, hi = 6.

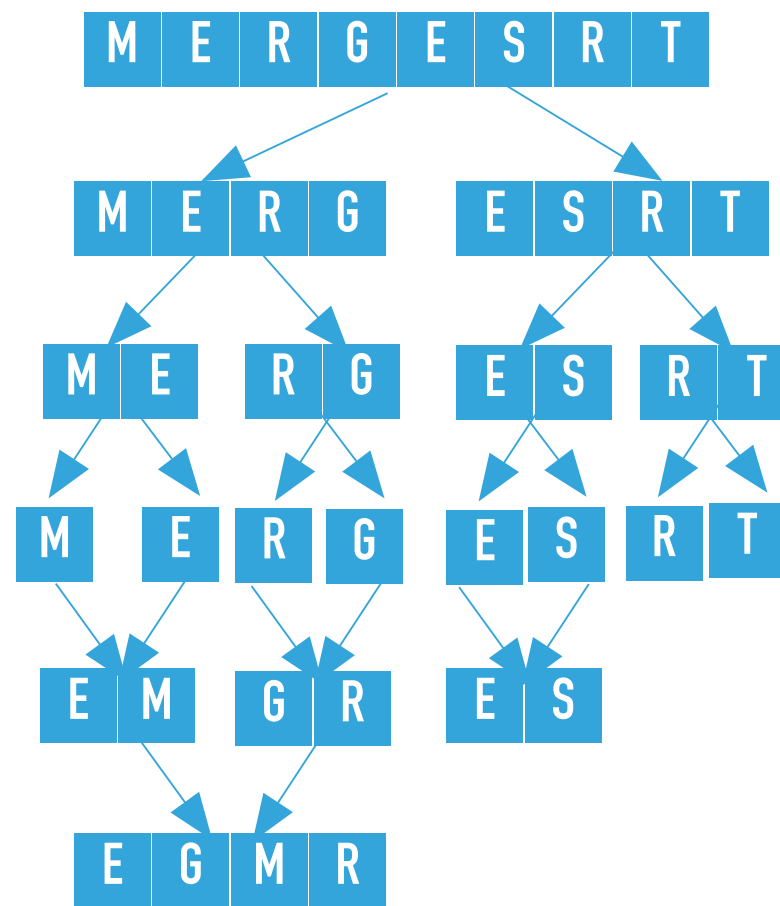


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 6)` finds `hi <= lo` and returns.

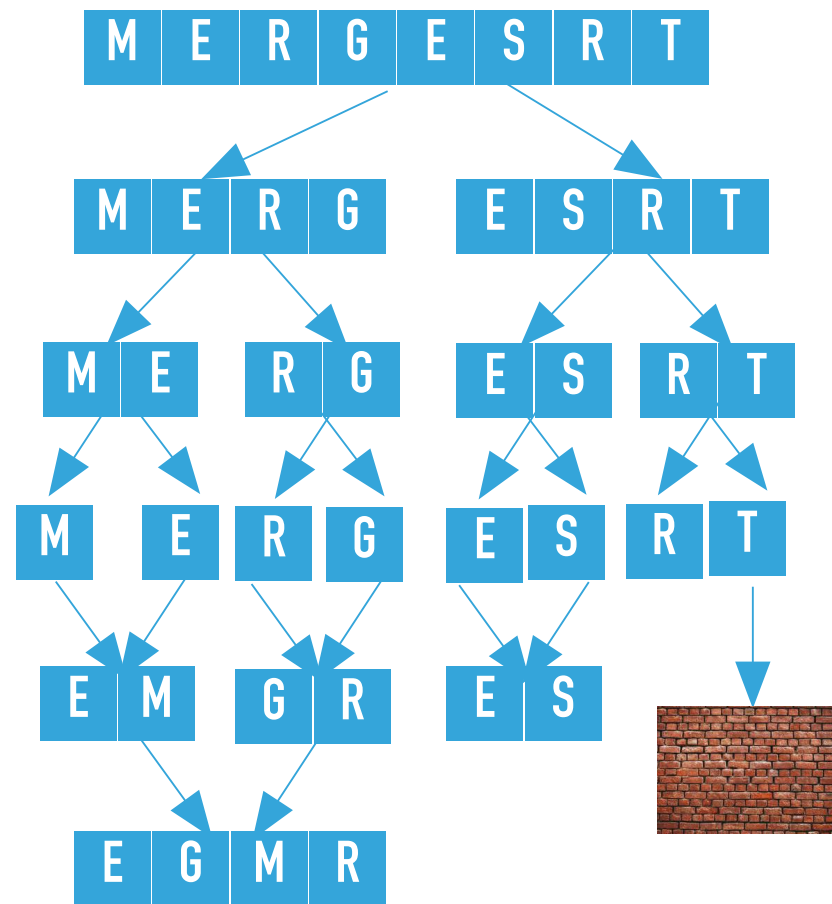


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 7) calls recursively sort on the right subarray, that is sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 7, 7), where lo = 7, hi = 7

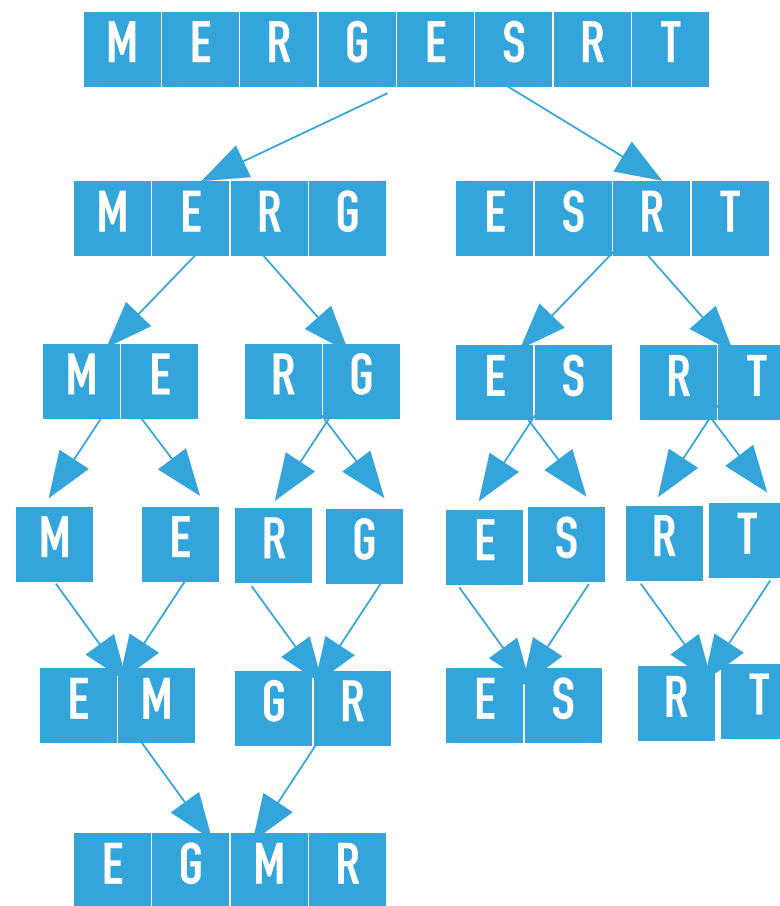


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 7, 7) finds $hi \leq lo$ and returns.

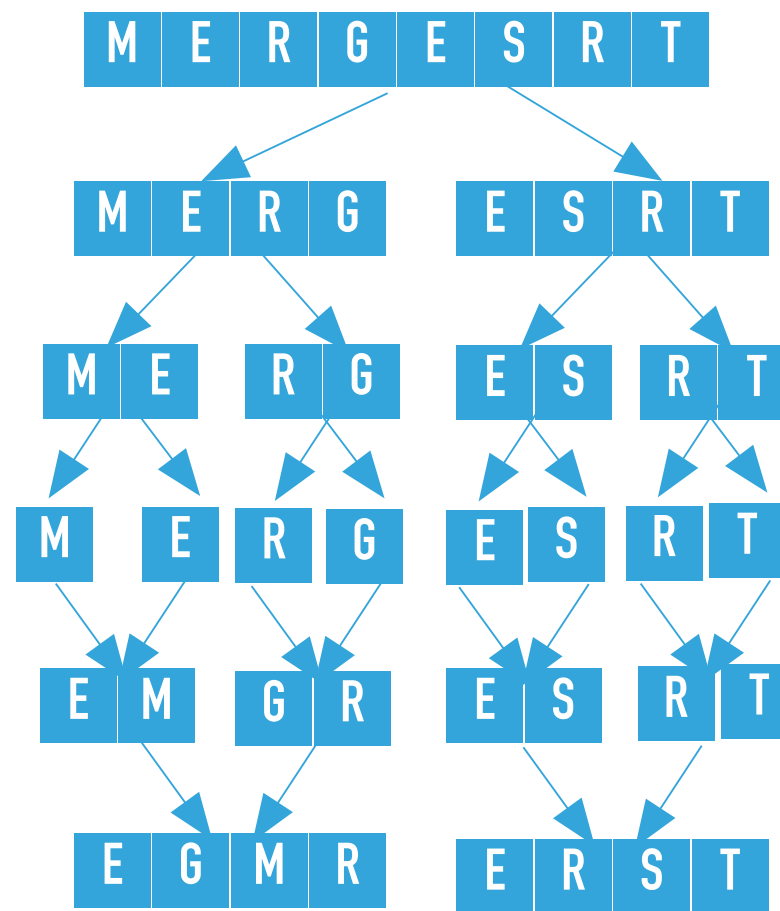


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 7) merges the two subarrays that is calls merge([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 6, 7), where lo = 6, mid = 6, and hi = 7. The resulting partially sorted array is [E, G, M, R, E, S, R, T].

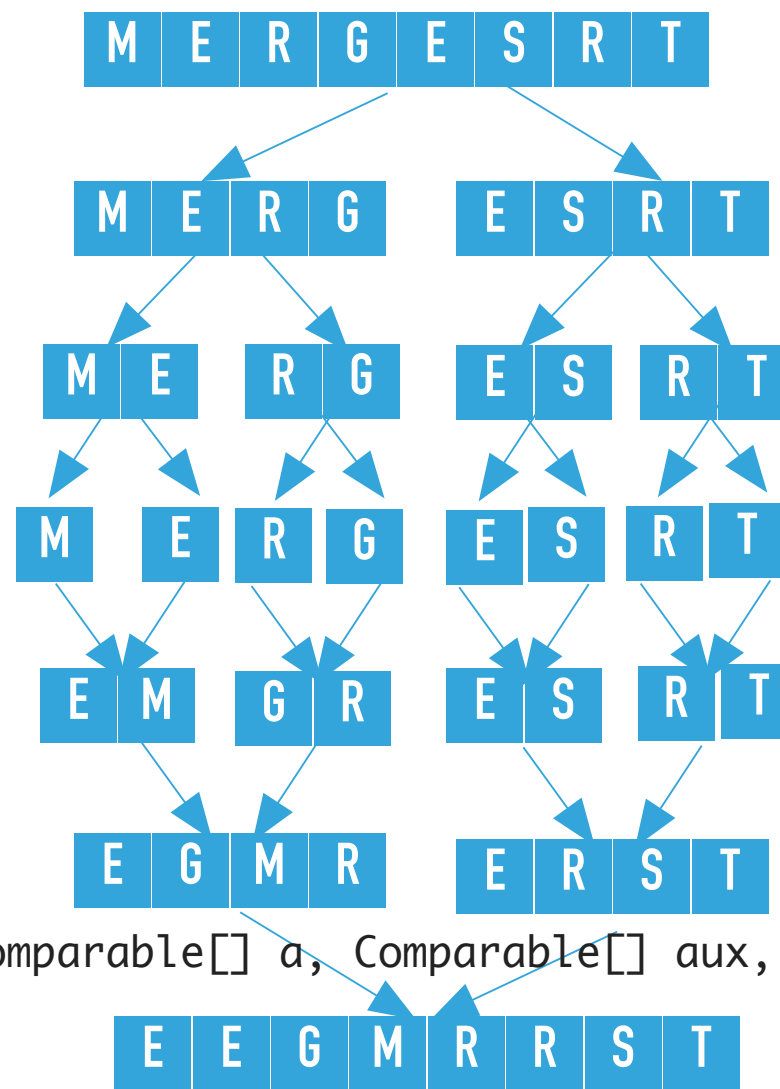


```

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, R, T], 4, 7)` merges the two subarrays that is calls `merge([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, R, T], 4, 5, 7)`, where `lo = 4`, `mid = 5`, and `hi = 7`. The resulting partially sorted array is `[E, G, M, R, E, R, S, T]`.



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`sort([E, G, M, R, E, R, S, T], [E, M, G, R, E, S, R, T], 0, 7)` merges the two subarrays that is calls `merge([E, G, M, R, E, R, S, T], [E, M, G, R, E, S, R, T], 0, 3, 7)`, where `lo = 0`, `mid = 3`, and `hi = 7`. The resulting sorted array is `[E, E, G, M, R, R, S, T]`.

Practice time

Which of the following subarray lengths will occur when running mergesort on an array of length 10?

- A. { 1, 2, 3, 5, 10 }
- B. { 2, 4, 6, 8, 10 }
- C. { 1, 2, 5, 10 }
- D. { 1, 2, 3, 4, 5, 10 }

Answer

Which of the following subarray lengths will occur when running mergesort on an array of length 10?

A. { 1, 2, 3, 5, 10 }

Good algorithms are better than supercomputers

- ▶ Your laptop executes 10^8 comparisons per second
- ▶ A supercomputer executes 10^{12} comparisons per second

	Insertion sort			Mergesort		
Computer	Thousand inputs	Million inputs	Billion inputs	Thousand inputs	Million inputs	Billion inputs
Home	Instant	2 hours	300 years	instant	1 sec	15 min
Supercomputer	Instant	1 second	1 week	instant	instant	instant

Analysis of comparisons

- ▶ We will assume that n is a power of 2 ($n = 2^k$, where $k = \log_2 n$) and the number of comparisons $T(n)$ to sort an array of length n with merge sort satisfies the recurrence:
 - ▶ $T(n) = T(n/2) + T(n/2) + (n - 1) = O(n \log n)$
- ▶ Number of array accesses (rather than exchanges, here) is also $O(n \log n)$.

Mergesort uses $\leq n \log n$ compares to sort an array of length n

If $n = 4$, 2 levels

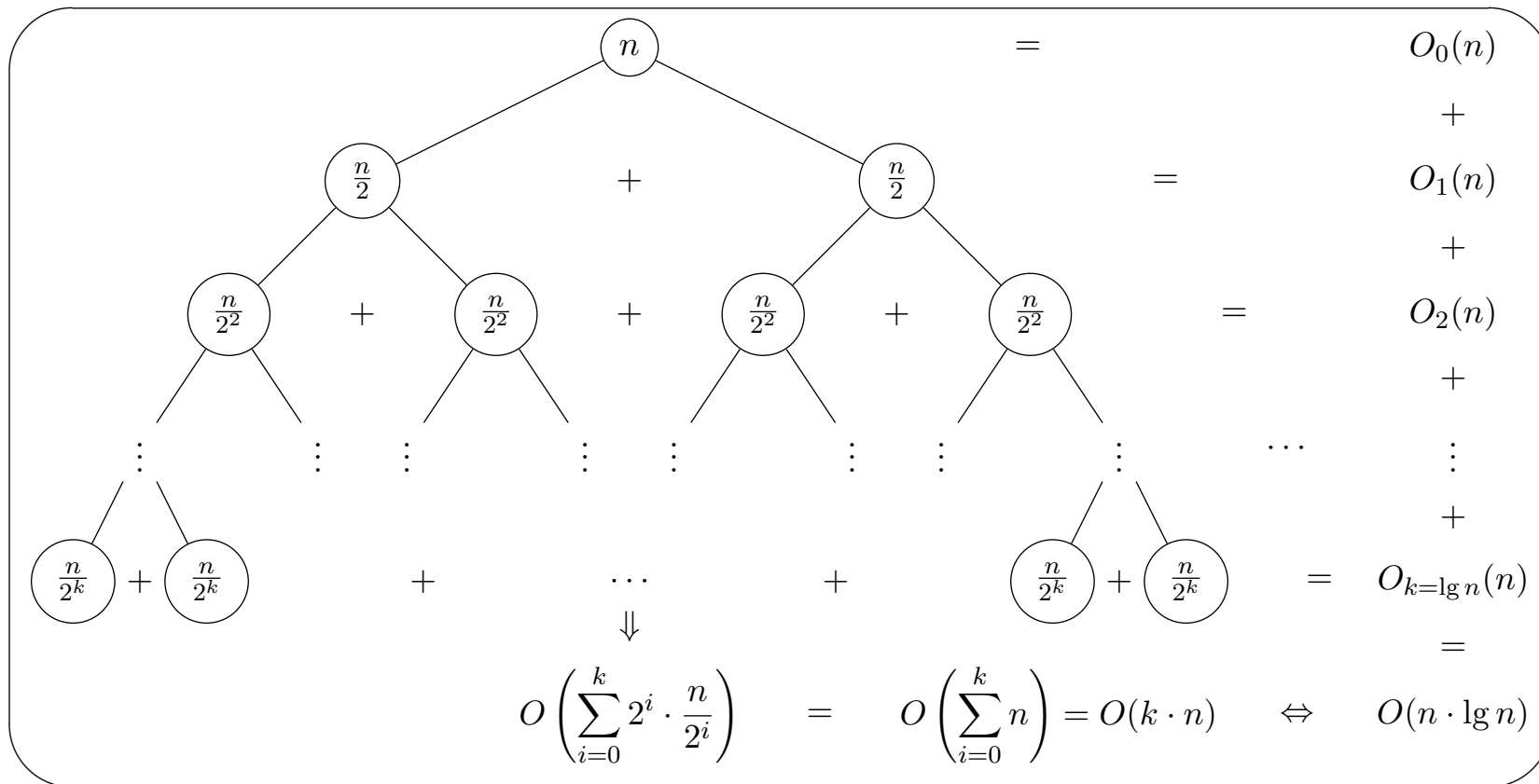
If $n = 8$, 3 levels

If $n = 16$, 4 levels

...

If $n = 2^k$, k levels,

or $k = \log_2 n$



Any algorithm with the same structure takes $n \log n$ time

```
public static void f(int n) {  
    if (n == 0)  
        return;  
    f(n/2);  
    f(n/2);  
    linear(n);  
}
```

Mergesort basics

- ▶ Auxiliary memory is proportional to n , as `aux[]` needs to be of length n for the last merge.
- ▶ At its simplest form, merge sort is **not an in-place algorithm**.
- ▶ There are modifications for halting the size of the auxiliary array but in-place merge is very hard.
- ▶ **Stable**: Look into `merge()`, if equal keys, it takes them from the left subarray.
 - ▶ So is insertion sort, but not selection sort.

Practical improvements for Mergesort

- ▶ Use insertion sort for small subarrays.
- ▶ Stop if already sorted.
- ▶ Eliminate the copy to the auxiliary array by saving time (not space).

```
private static void sort(Comparable[] src, Comparable[] dst, int lo, int hi) {  
    if (hi <= lo + 7) {  
        insertionSort(dst, lo, hi);  
        return;  
    }  
  
    int mid = lo + (hi - lo) / 2;  
    sort(dst, src, lo, mid);  
    sort(dst, src, mid+1, hi);  
  
    if (!less(src[mid+1], src[mid])) {  
        for (int i = lo; i <= hi; i++) dst[i] = src[i];  
        return;  
    }  
  
    merge(src, dst, lo, mid, hi);  
}
```

- ▶ For years, Java used this version to sort Collections of objects.

Sorting: the story so far

	In place	Stable	Best	Average	Worst	Remarks
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$O(n)$	$O(n^2)$	$O(n^2)$	Use for small arrays or partially
Merge sort		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; stable

Lecture 13: Mergesort

- ▶ Mergesort

Readings:

- ▶ Textbook:
 - ▶ Chapter 2.2 (pages 270-277)
- ▶ Website:
 - ▶ Mergesort: <https://algs4.cs.princeton.edu/22mergesort/>
 - ▶ Code: <https://algs4.cs.princeton.edu/22mergesort/Merge.java.html>

Practice Problems:

- ▶ 2.2.1-2.2.2, 2.2.11