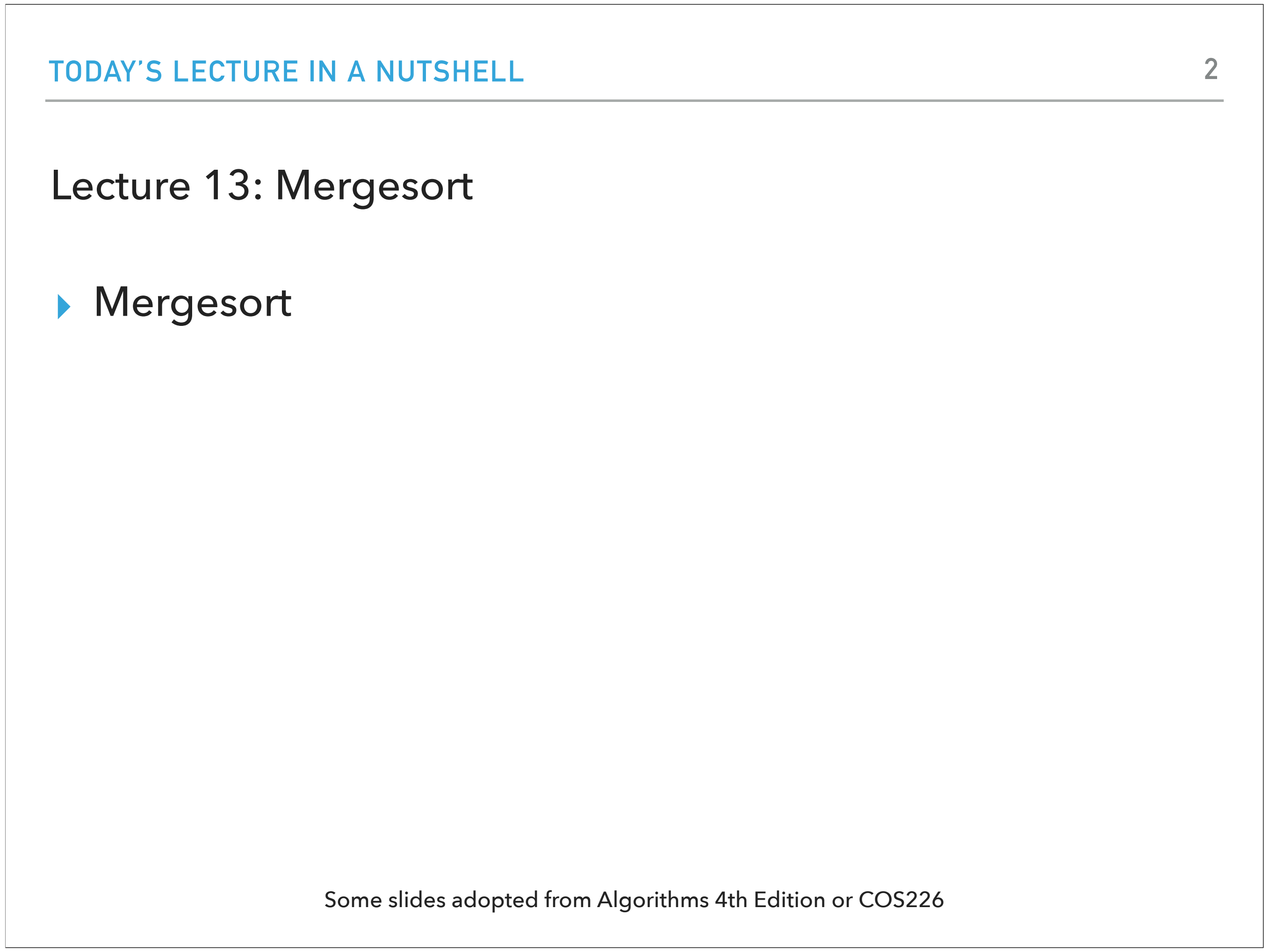
CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

14: Mergesort

Alexandra Papoutsaki
she/her/hers

Today we'll talk about merge sort, a very popular sorting algorithm



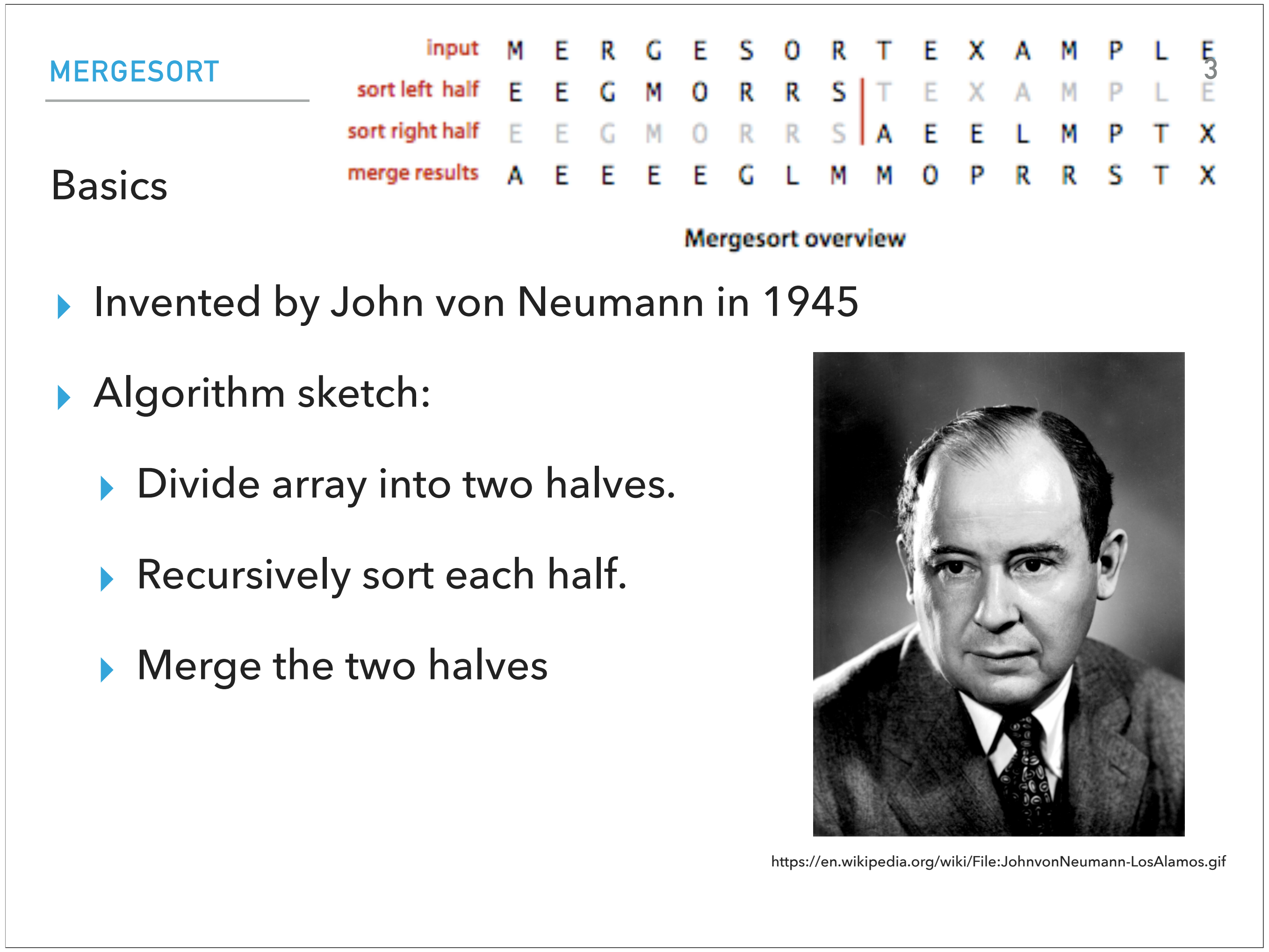
TODAY'S LECTURE IN A NUTSHELL

Lecture 13: Mergesort

- Mergesort

Some slides adopted from Algorithms 4th Edition or COS226

You probably have seen or at least heard of merge sort.



MERGESORT

Basics

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

Mergesort overview

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

https://en.wikipedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif

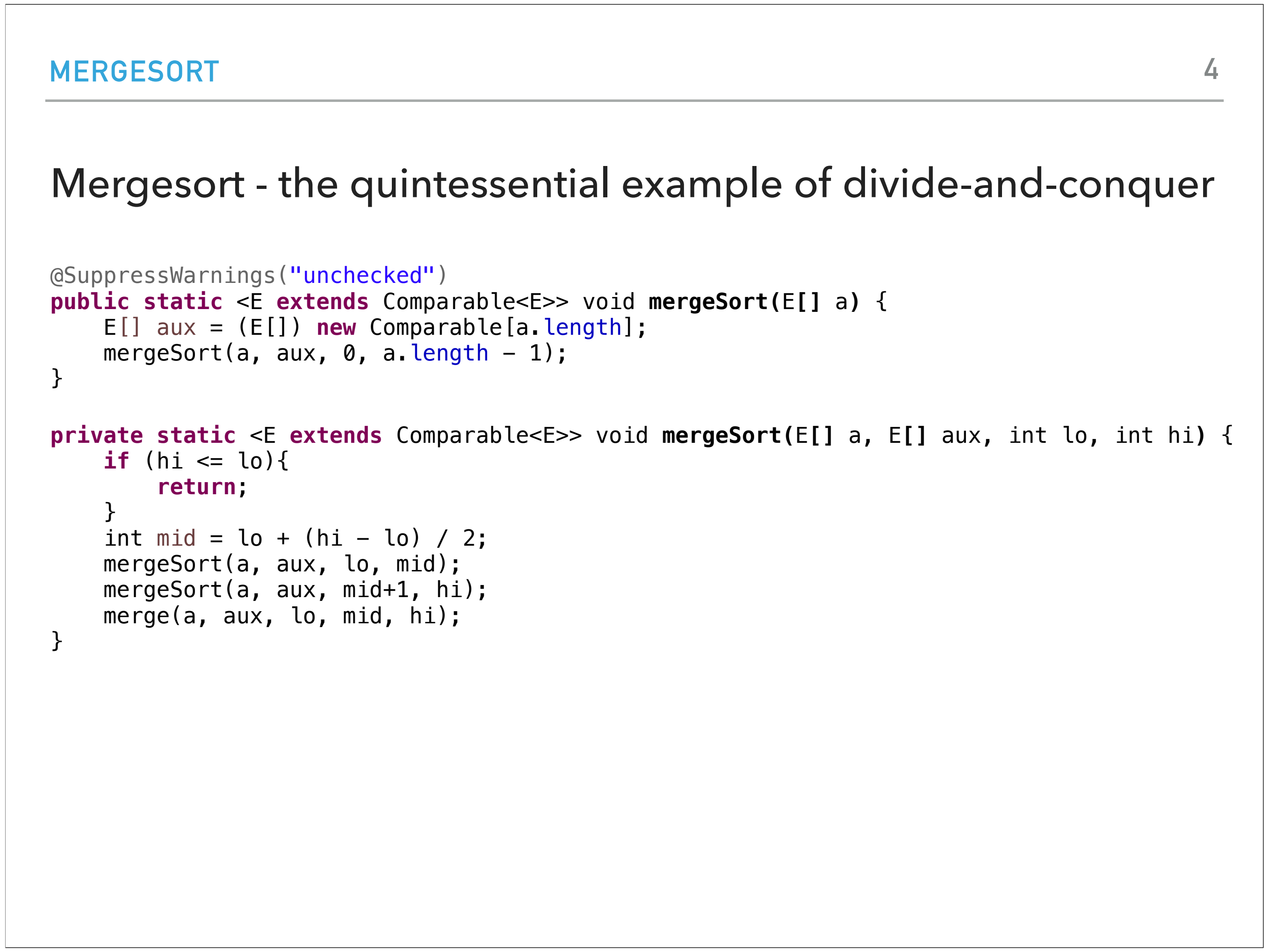
Mergesort was invented by John von Neumann, a Hungarian-American mathematician, physicist, computer scientist, and polymath. Von Neumann was generally regarded as the foremost mathematician of his time and said to be "the last representative of the great mathematicians” who integrated both pure and applied sciences. His contributions to math, physics, cs, and engineering are numerous so it's a person worth knowing about.

Mergesort is based on the following idea. Given an array to sort:

Divide array into two halves.

Recursively sort each half.

Merge the two halves



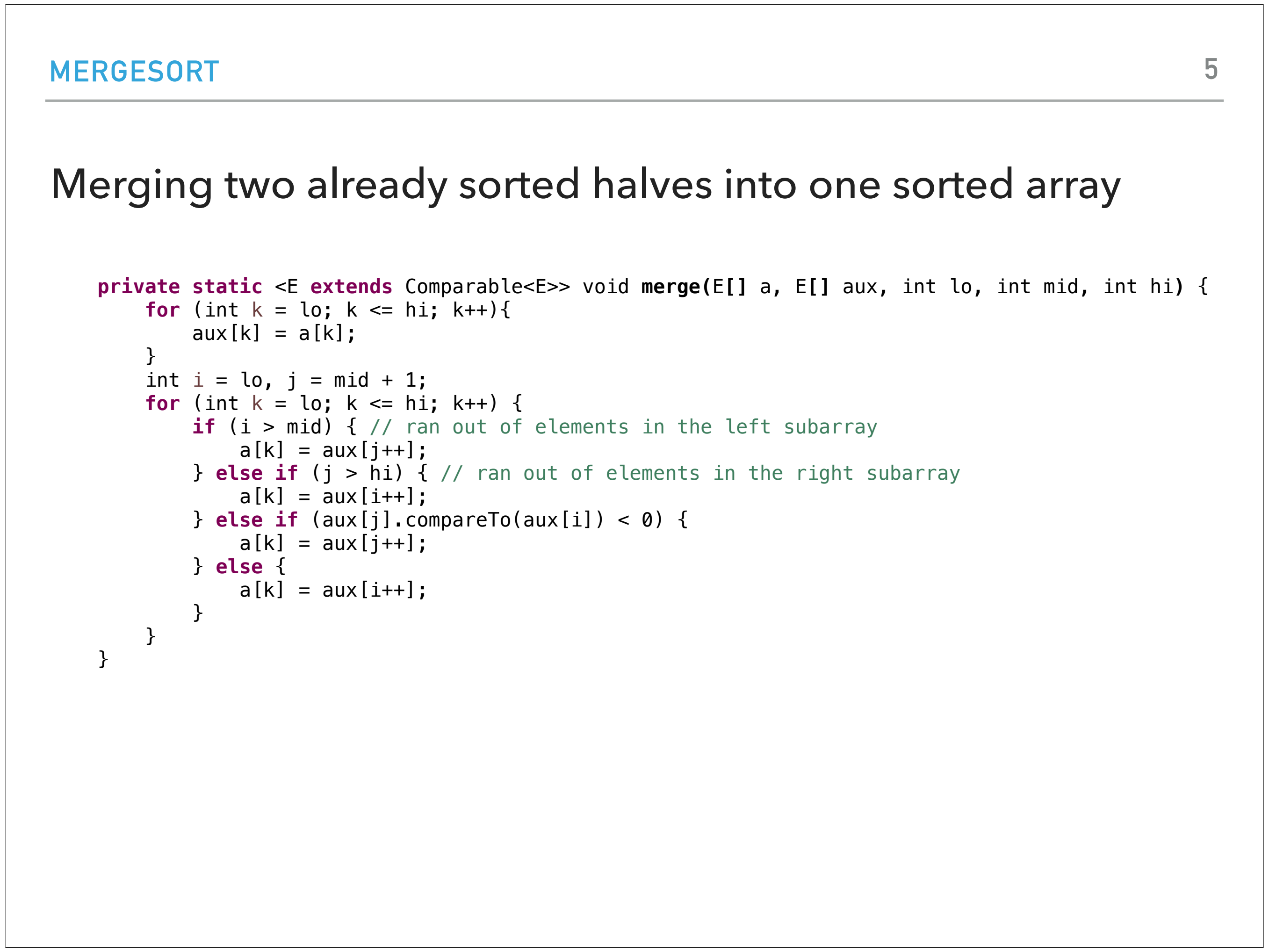
Mergesort - the quintessential example of divide-and-conquer

```
@SuppressWarnings("unchecked")
public static <E extends Comparable<E>> void mergeSort(E[] a) {
    E[] aux = (E[]) new Comparable[a.length];
    mergeSort(a, aux, 0, a.length - 1);
}

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

Let's see how we could translate this idea in Java code. You will notice that there is an overloaded method here, mergeSort. The public method mergeSort is what the client would call by passing an array to be sorted. Internally, that method creates an auxiliary array of the same length with a and then passes the original array and the auxiliary array along with 0 and n-1 (a.length-1) to a private method mergeSort. This private method is recursive and works by recursively sorting each half (this is why we have a mid point!) and then merging them. The merging happens in a separate helper method called merge and we will start with that.

If you are wondering about the way that mid is calculated, (hi + lo)/2 can overflow. lo+(hi-lo)/2 will never overflow if lo, hi are positive integers within range and lo <= hi.



Merging two already sorted halves into one sorted array

```
private static <E extends Comparable<E>> void merge(E[] a, E[] 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 (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

merge takes as parameters the array to sort (a), an auxiliary array (aux), a low (lo), middle (mid), and high(hi) index. The assumption is that a has been sorted so far from a[lo...mid] and a(mid...hi]. Our goal will be to make a[lo...hi] be sorted by merging its two sorted halves.

It starts by copying all the data from the low to the high index from a to aux.

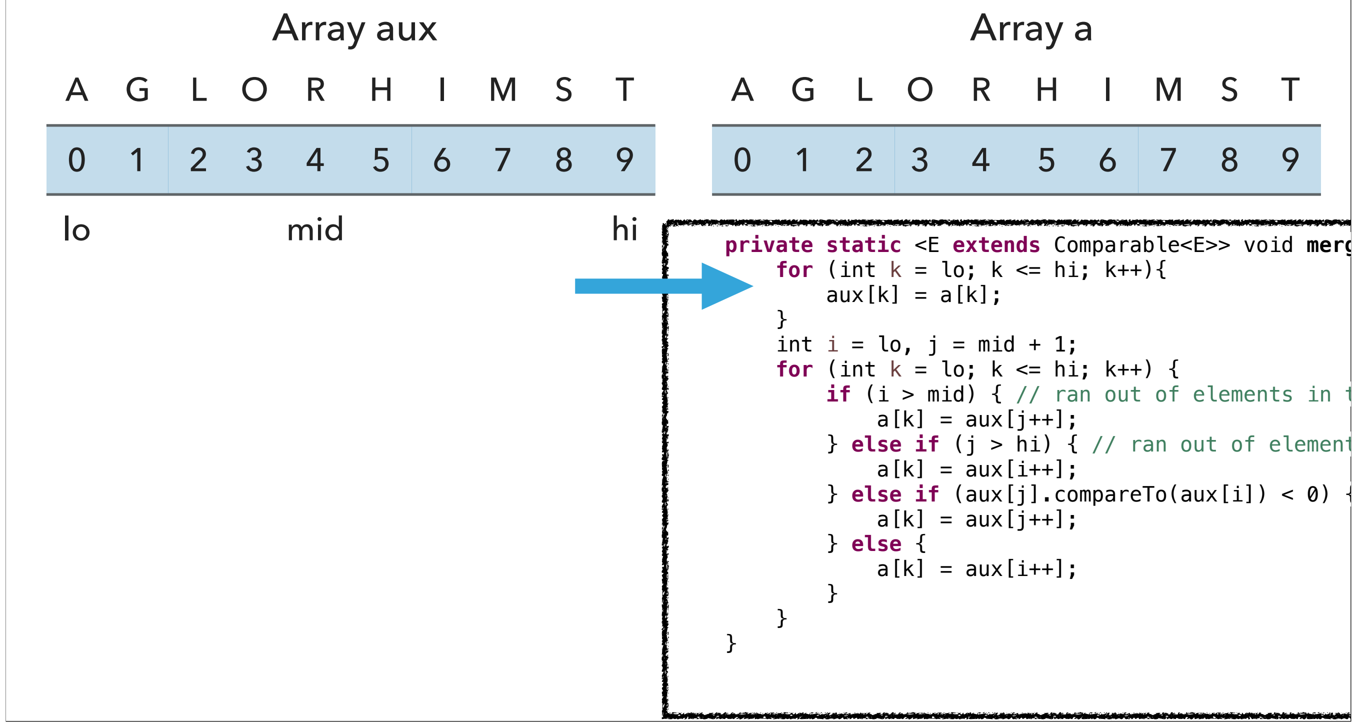
aux will be used to keep the original unsorted subarray, while a[lo...hi] will end up being sorted.

How does this work?

we have two running indices, one, i, that starts at lo, and one, j, that starts at mid+1. We also have k that runs from lo to hi. We repeatedly compare the element at index i with the element at index j. if the i-th element is smaller, we take that one and set it in k and advance i. If it is the j-th element that is smaller, we take that one, and set it in k, and advance j.

If at some point, i exceeds mid (we ran out of elements on the left subarray), or j exceeds high (we ran out of elements on the right subarray), we just transfer what's left which is guaranteed to be larger than what we have already sorted to a.

Merging Example - 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 <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

Let's see that in practice by trying to merge two sorted halves of the array, AGLOR and HIMST. We start by populating the auxiliary array with the contents of a from lo to hi.

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									

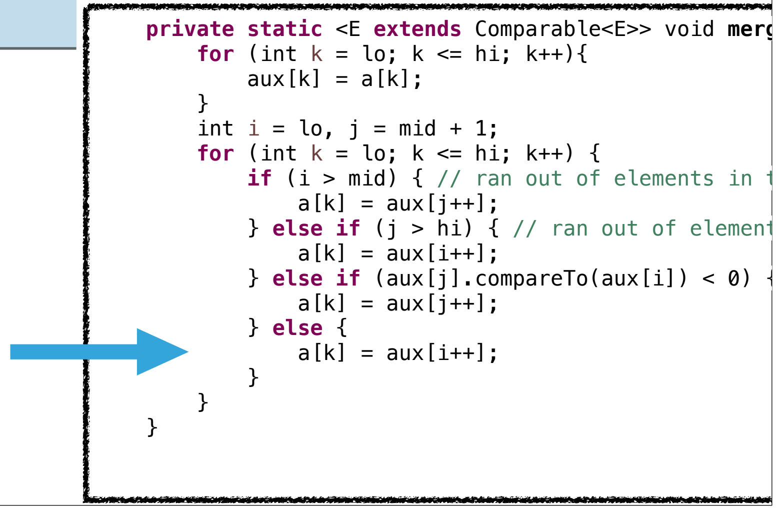
Array a

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

case: aux[i]<aux[j]

a[0]=aux[0]

i++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

Then we repeatedly compare the ith and jth element. Since A<H, we keep A and increase i by one.

Merging Example - k=1

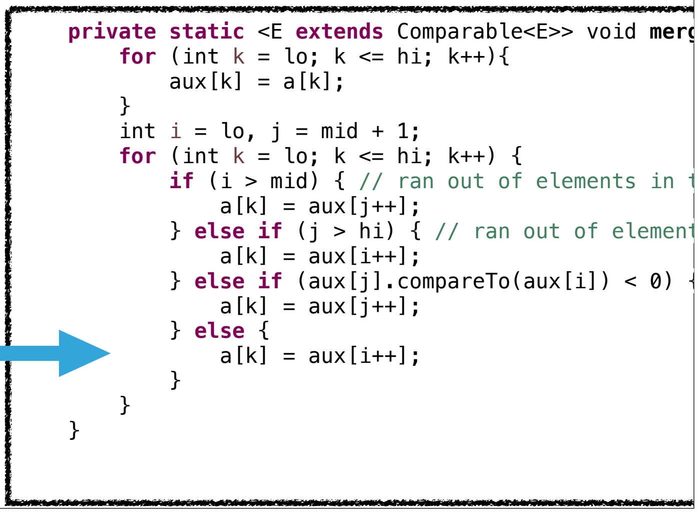
Array aux

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								

Array a

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

case: aux[i]<aux[j]
a[1]=aux[1]
i++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

G<H so again we keep G and increase i by 1.

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							

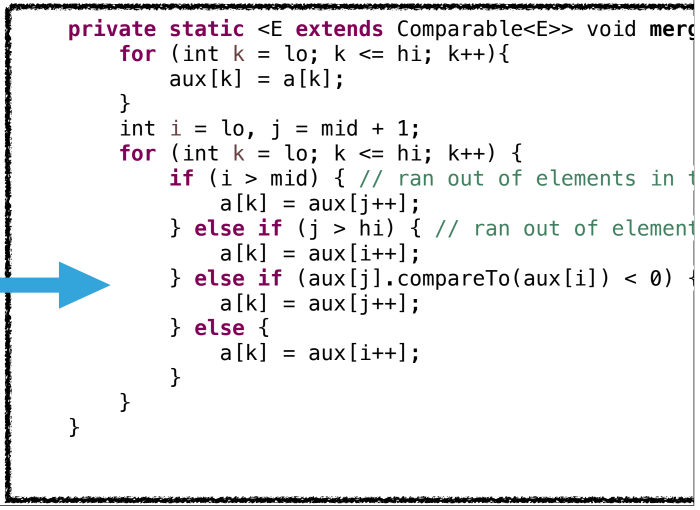
Array a

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

case: aux[i]>aux[j]

a[2]=aux[5]

j++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

L>H so we keep L and increase j by 1.

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						

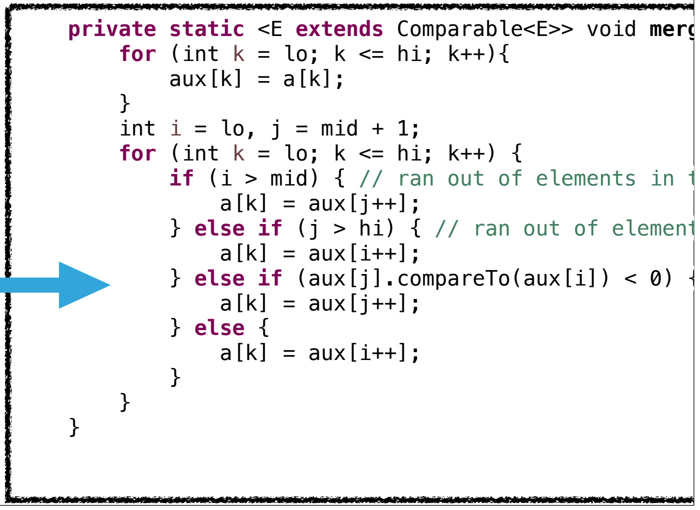
Array a

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

case: aux[i]>aux[j]

a[3]=aux[6]

j++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

I>G so we keep G and increase j by 1.

Merging Example - k=4

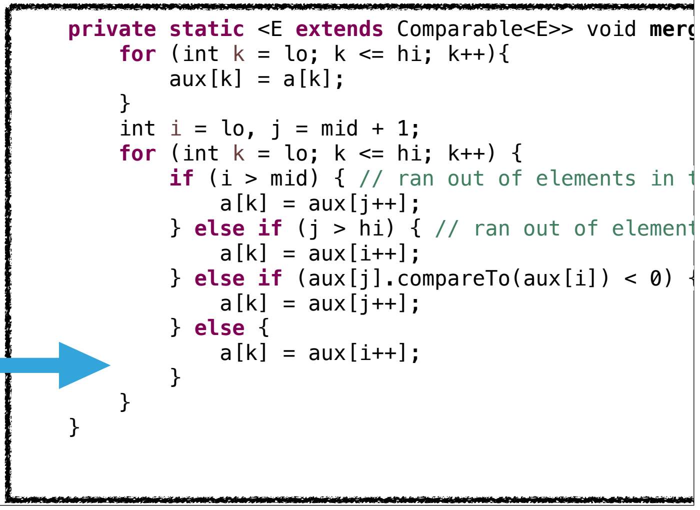
Array aux

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					

Array a

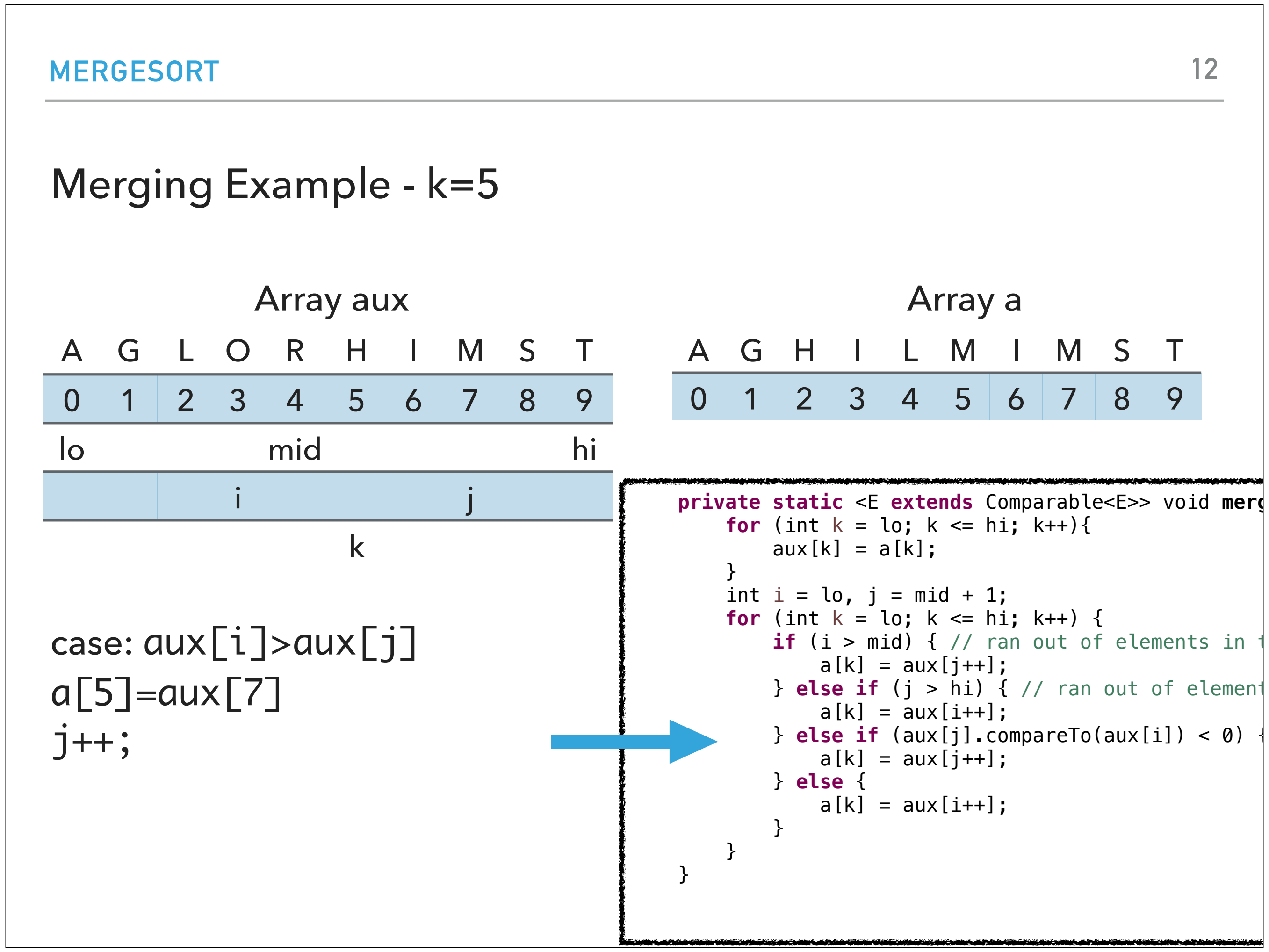
A	G	H	I	L	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9

case: aux[i]<aux[j]
a[4]=aux[2]
i++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

L<M so we keep L and increase j by 1.



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				

Array a

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

case: aux[i]>aux[j]
a[5]=aux[7]
j++;

```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

O>M so we keep M and increase j by 1.

Merging Example - k=6

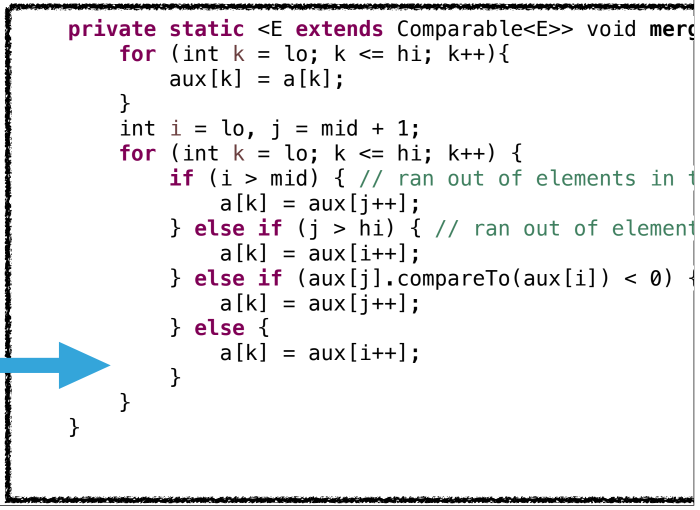
Array aux

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			

Array a

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

case: aux[i]<aux[j]
a[6]=aux[3]
i++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

O<S so we keep O and increase i by 1.

Merging Example - k=7

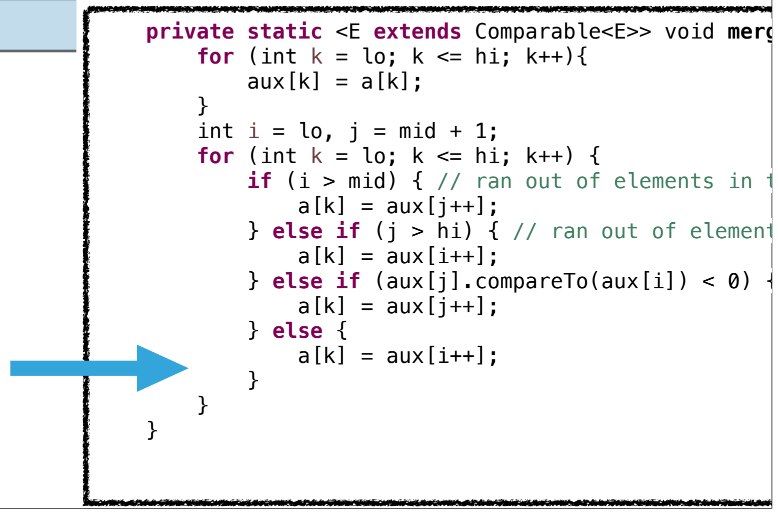
Array aux

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		

Array a

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

case: aux[i]<aux[j]
a[7]=aux[4]
i++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

R<S so we keep R and increase j by 1.

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	

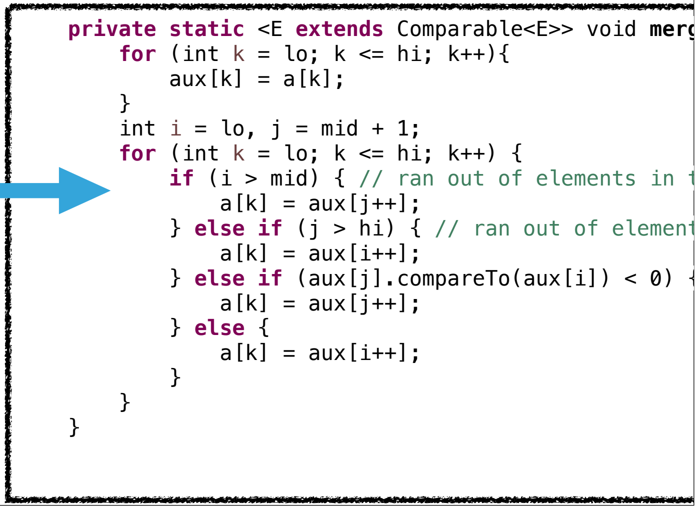
Array a

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

case: i>mid

a[8]=aux[8]

j++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

Since i>mid we keep S and increase j by 1.

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

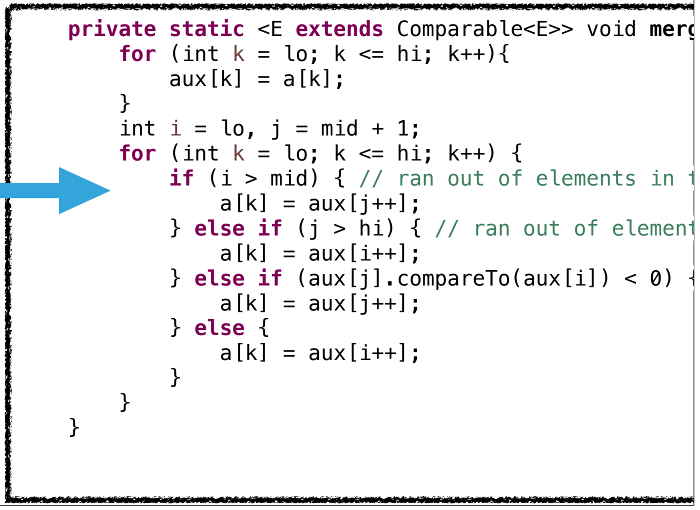
Array a

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

case: i>mid

a[9]=aux[9]

j++;



```
private static <E extends Comparable<E>> void merg
    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 t
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of element
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

Since i>mid we keep T and increase j by 1.

2.2 MERGING DEMO

http://algs4.cs.princeton.edu

https://algs4.cs.princeton.edu/lectures/demo/22DemoMerge.mov

Here is a video demo of how merging works.

Practice time

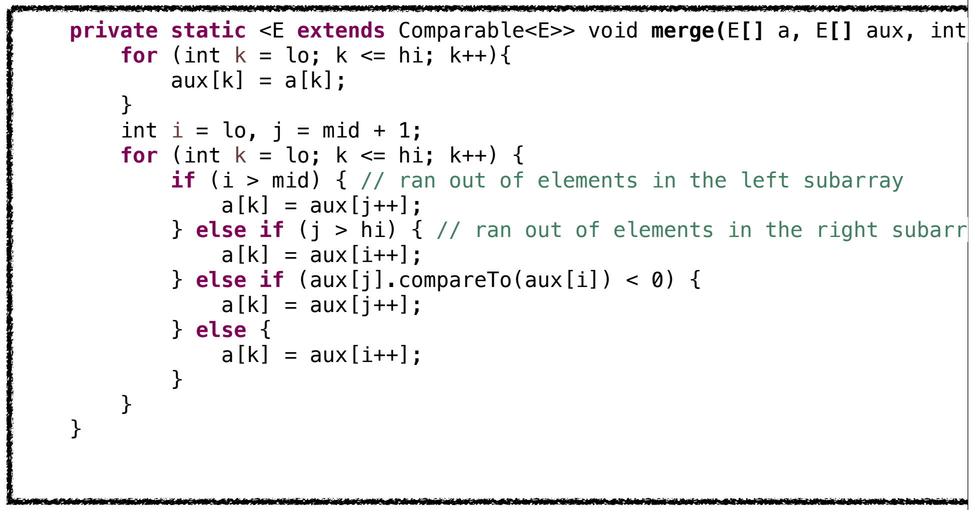
How many calls does `merge()` make to `compareTo()` in order to merge two already sorted subarrays, each of length $n/2$ into a sorted array of length n?

A. ~$1/4n$ to ~$1/2n$

B. ~$1/2n$

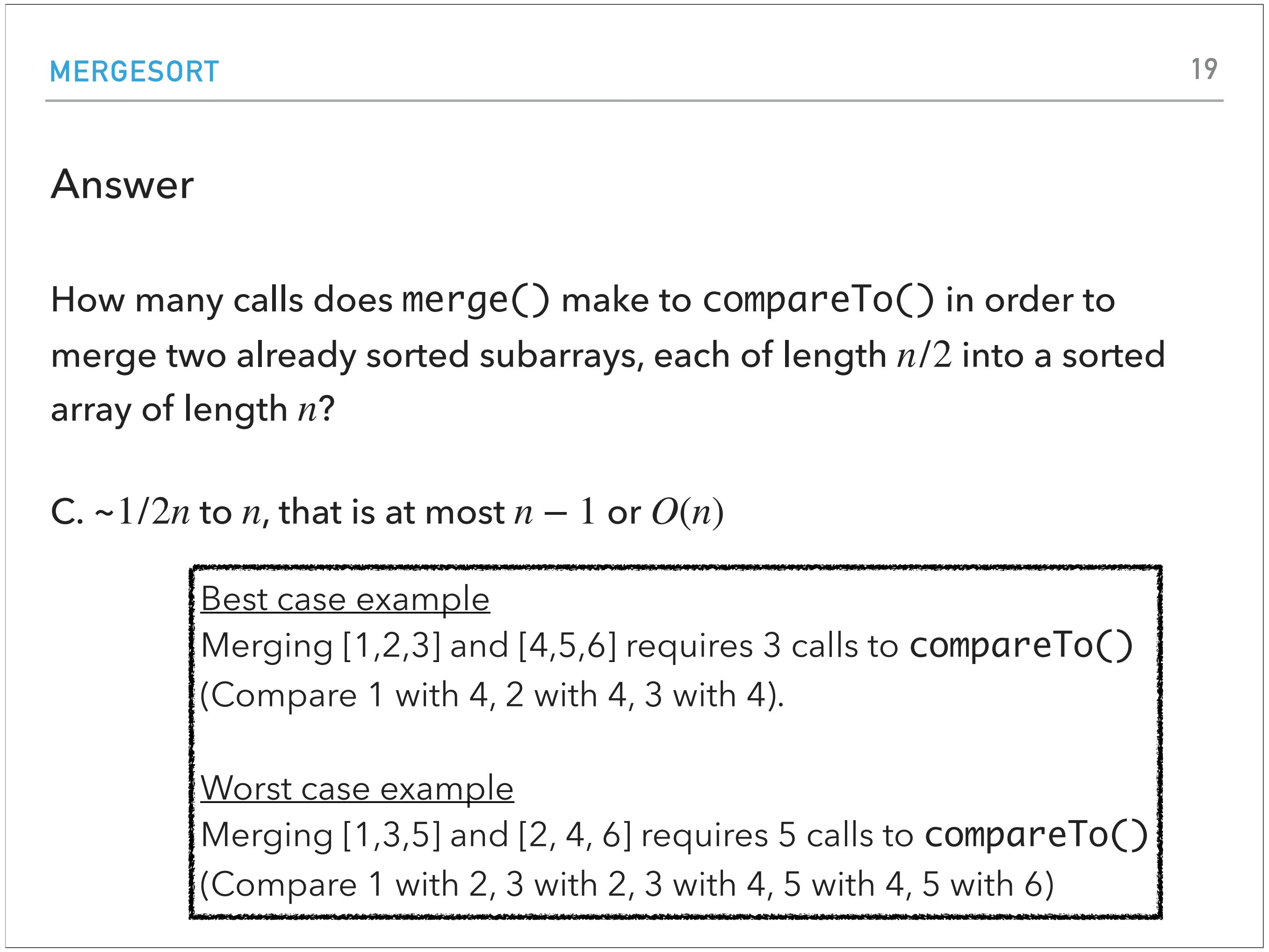
C. ~$1/2n$ to n

D. ~n



```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, int
    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 subarr
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

Let's think of the following question for a moment.



Answer

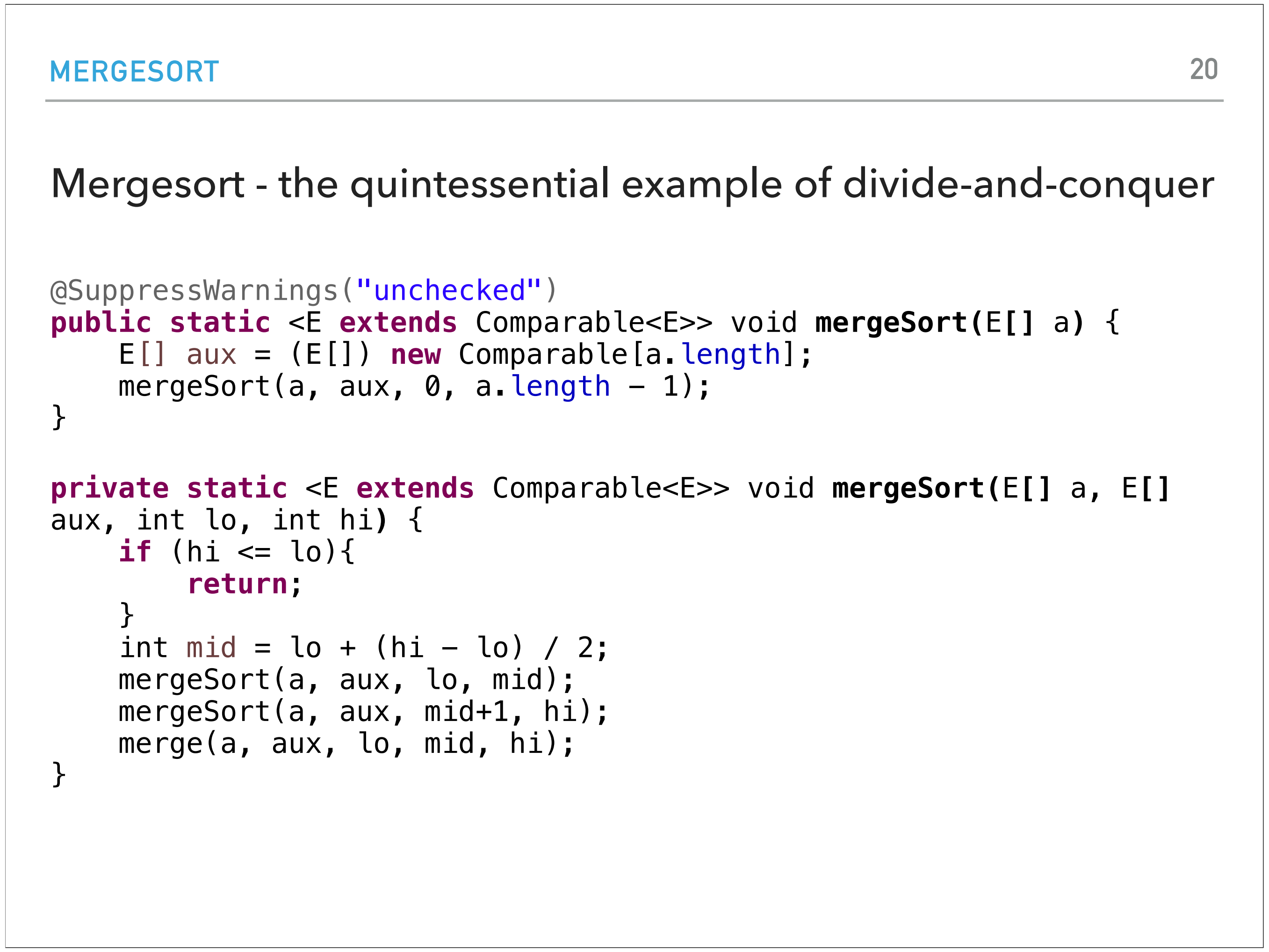
How many calls does `merge()` make to `compareTo()` in order to merge two already sorted subarrays, each of length $n/2$ into a sorted array of length n?

C. ~$1/2n$ 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 `compareTo()` (Compare 1 with 4, 2 with 4, 3 with 4).

Worst case example
Merging [1,3,5] and [2, 4, 6] requires 5 calls to `compareTo()` (Compare 1 with 2, 3 with 2, 3 with 4, 5 with 4, 5 with 6)

It's C. Here is a best and worst case example.

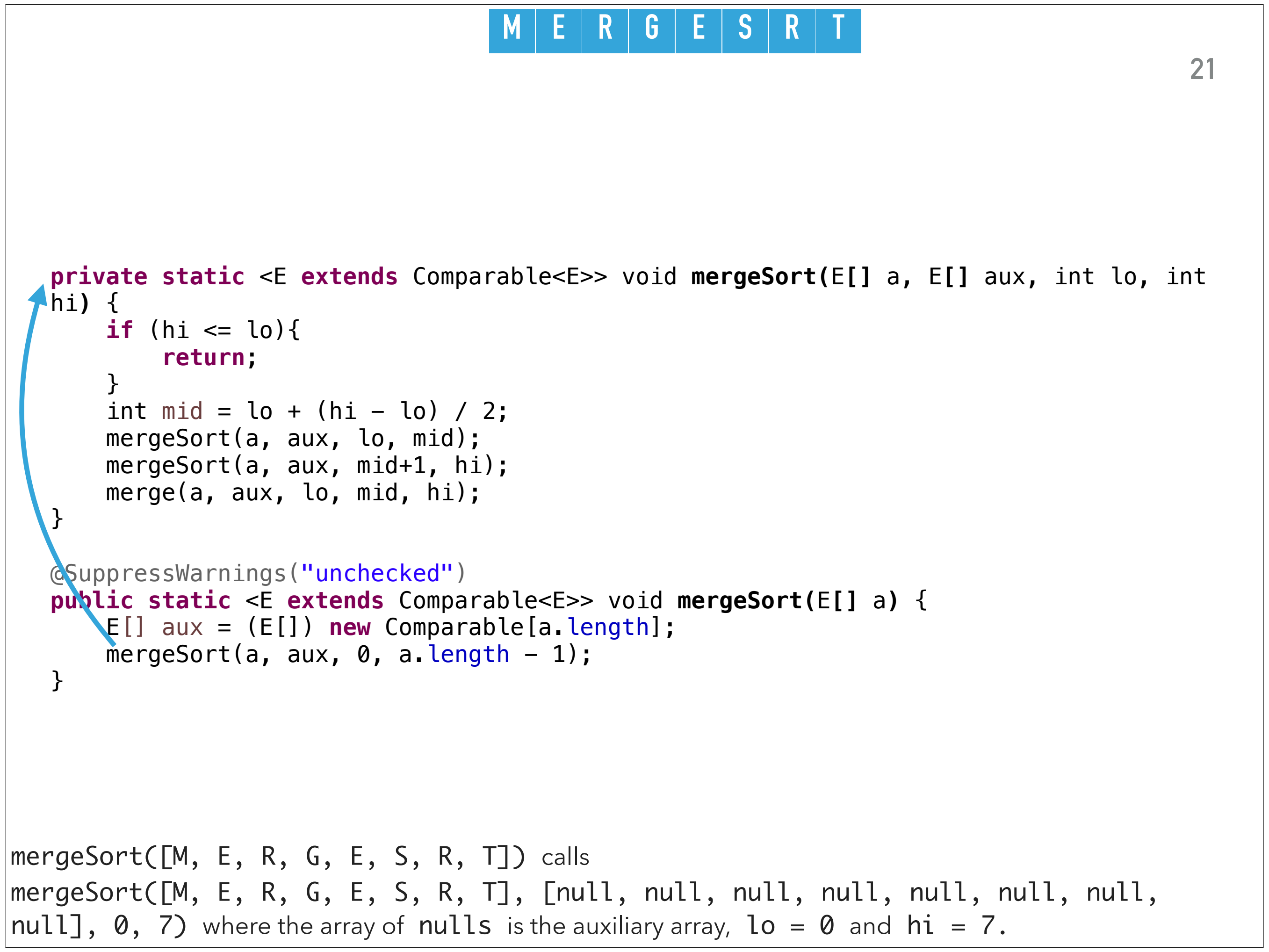


Mergesort - the quintessential example of divide-and-conquer

```
@SuppressWarnings("unchecked")
public static <E extends Comparable<E>> void mergeSort(E[] a) {
    E[] aux = (E[]) new Comparable[a.length];
    mergeSort(a, aux, 0, a.length - 1);
}

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

Now that we saw how merging works, let's go back to how the entire mergeSort works by using a running example.



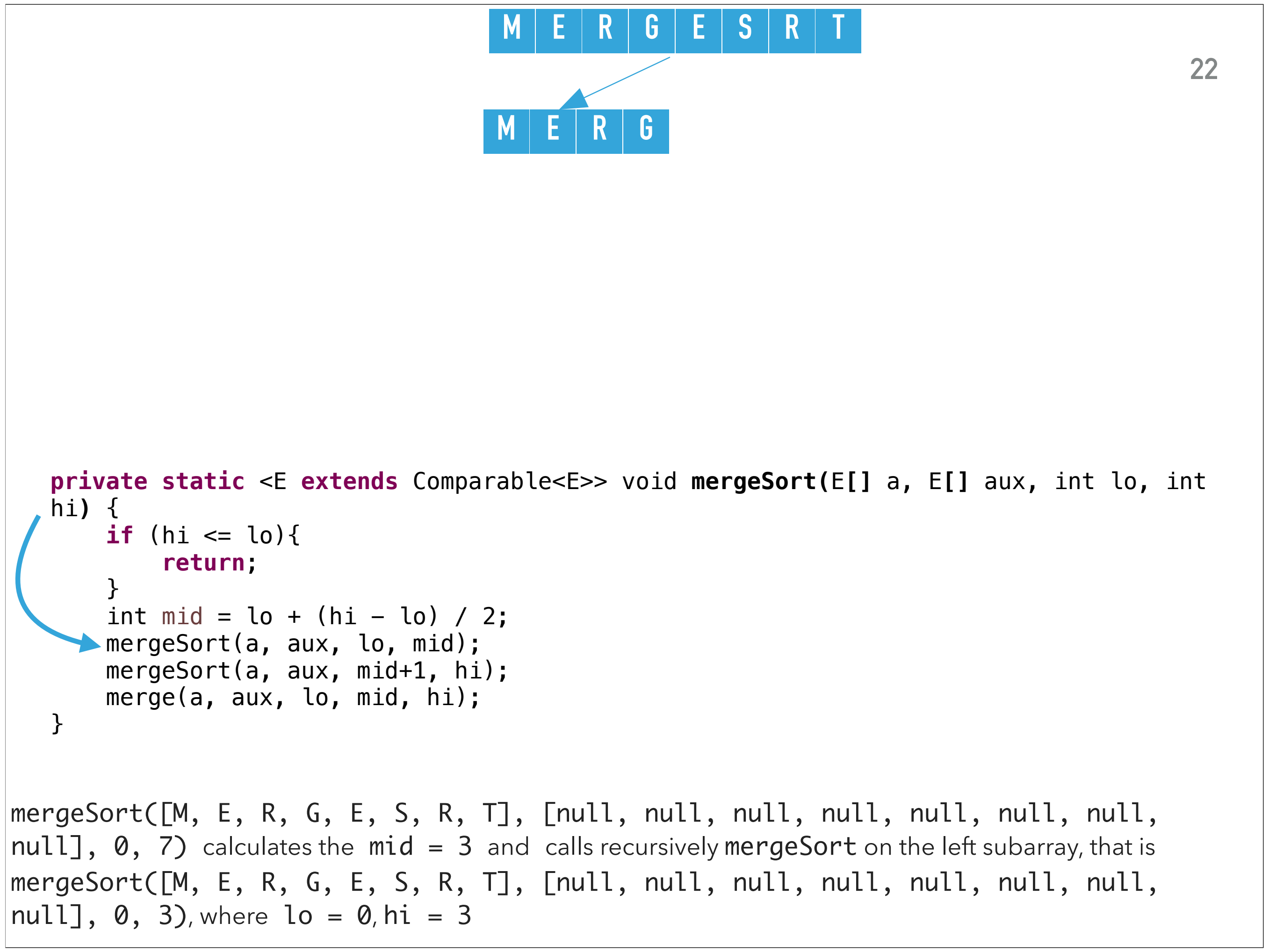
M	E	R	G	E	S	R	T

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

@SuppressWarnings("unchecked")
public static <E extends Comparable<E>> void mergeSort(E[] a) {
    E[] aux = (E[]) new Comparable[a.length];
    mergeSort(a, aux, 0, a.length - 1);
}
```

`mergeSort([M, E, R, G, E, S, R, T])` calls
`mergeSort([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`.

Let's say we call the public method mergeSort on the array [M, E, R, G, E, S, R, T]. This in turn would create an auxiliary array with length 8 and call the private method mergeSort passing it [M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 7. (0 and 7 are the lo and hi indices).



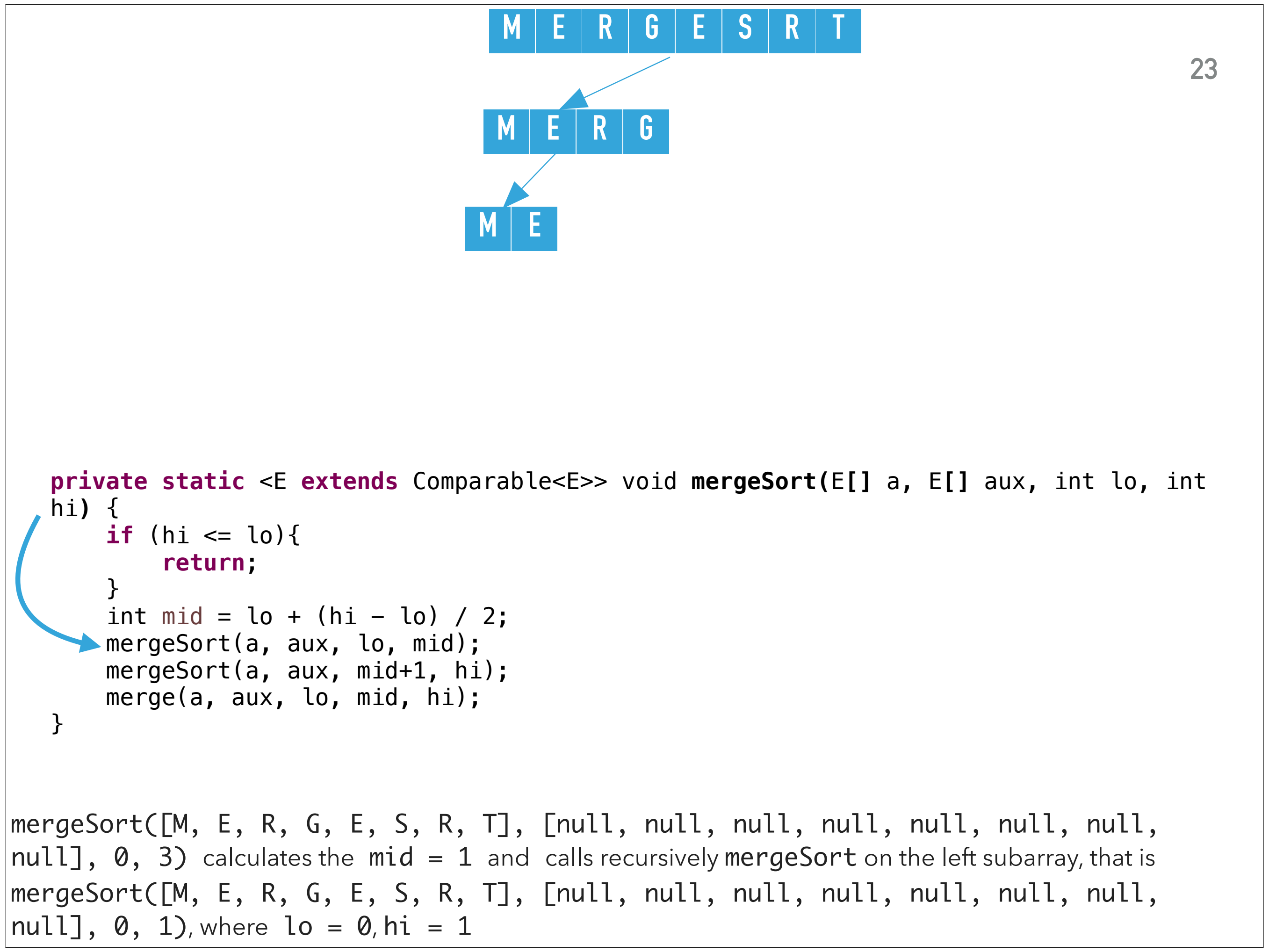
M E R G E S R T

M E R G

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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 3)`, where `lo = 0`, `hi = 3`

mergeSort will calculate mid to be 3 and will recursively call itself on the left subarray [M,E,R,G].



M E R G E S R T

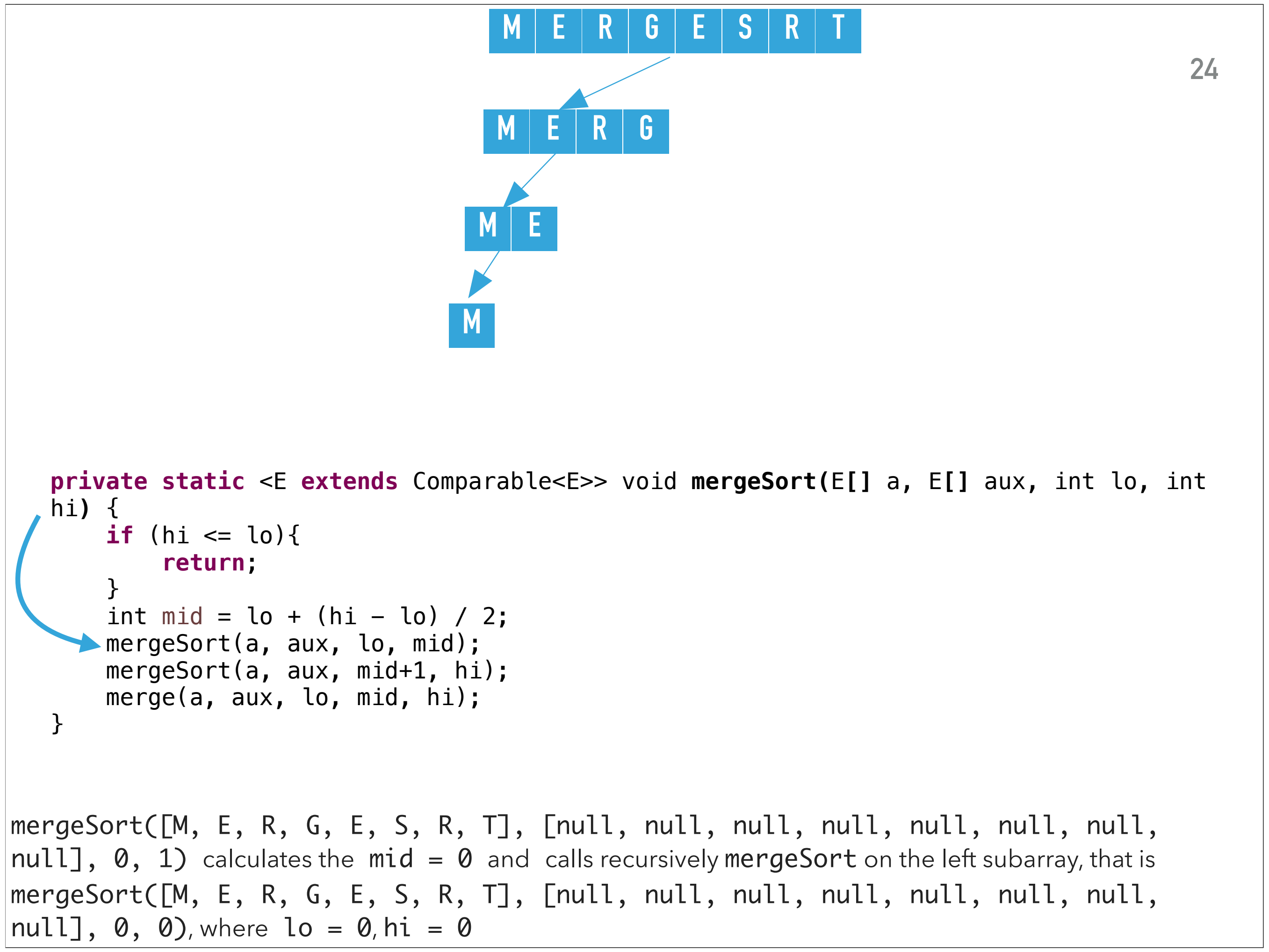
M E R G

M E

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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1)`, where `lo = 0`, `hi = 1`

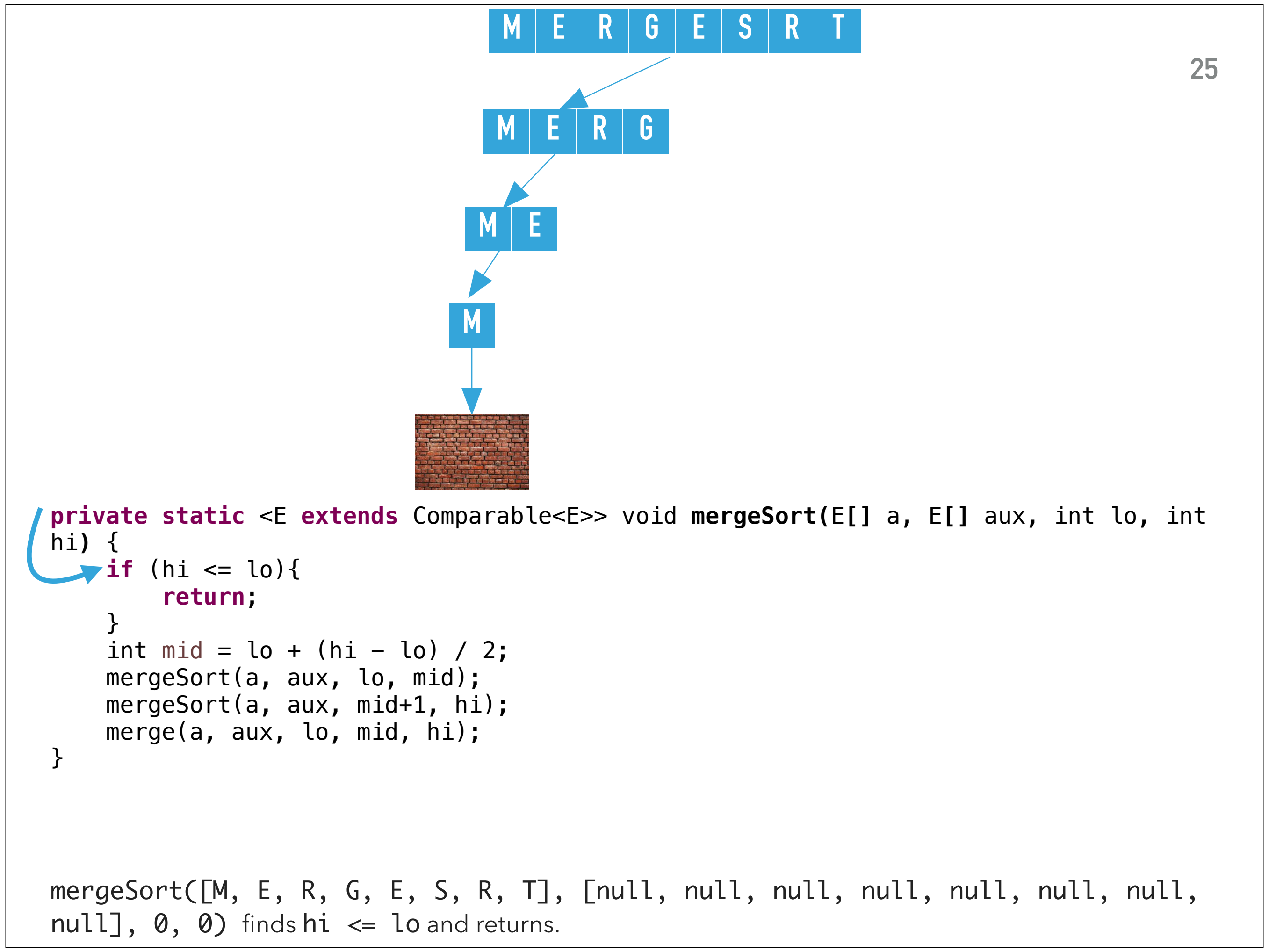
mid will now be 1 and mergeSort will be called recursively on the left subarray [M, E].



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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 0)`, where `lo = 0`, `hi = 0`

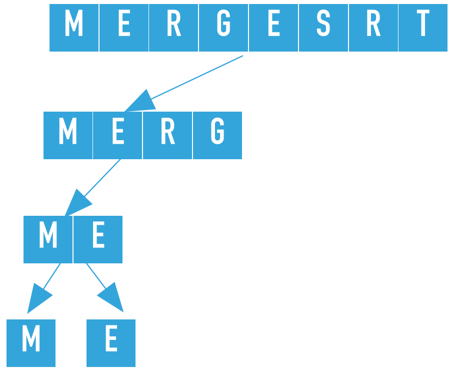
mid is now 0 and mergeSort will call itself recursively on the left subarray, that is [M]



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

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

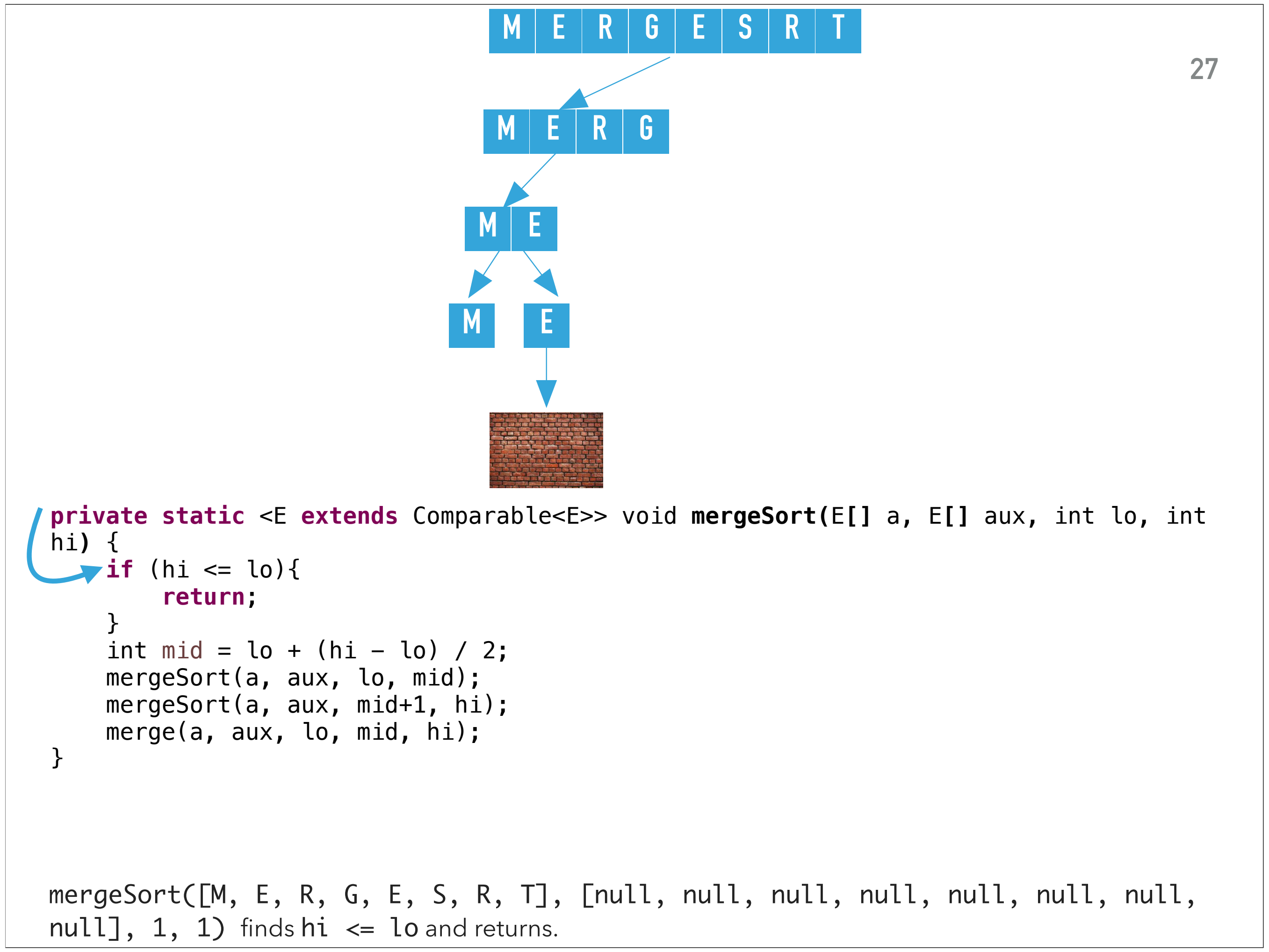
This hits the base case so now we can start unwinding the recursion by returning to the previous call.



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

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

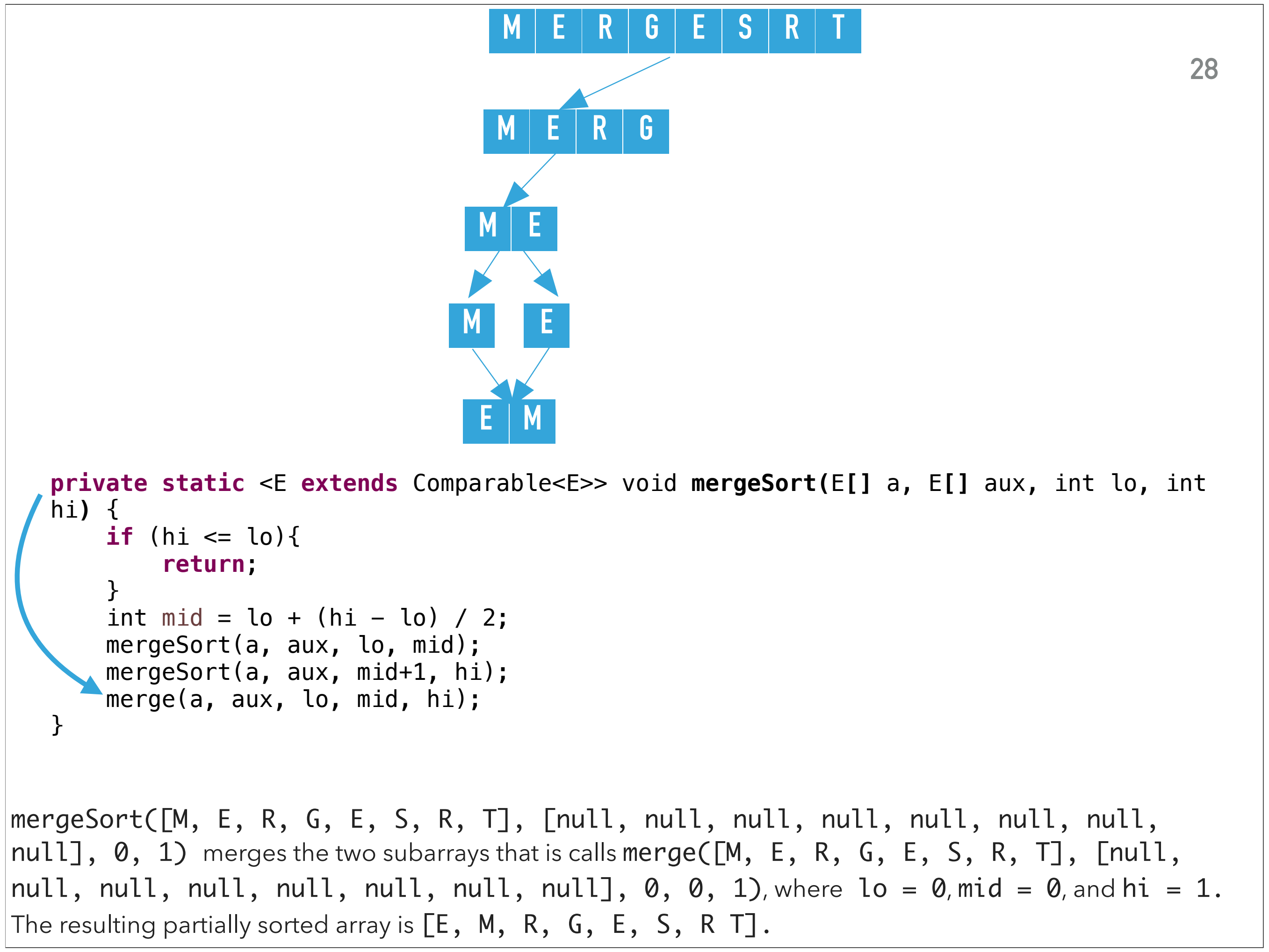
We are done with the left subarray of [M,E] and we will now call mergeSort recursively on the right subarray, that is [E].



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

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

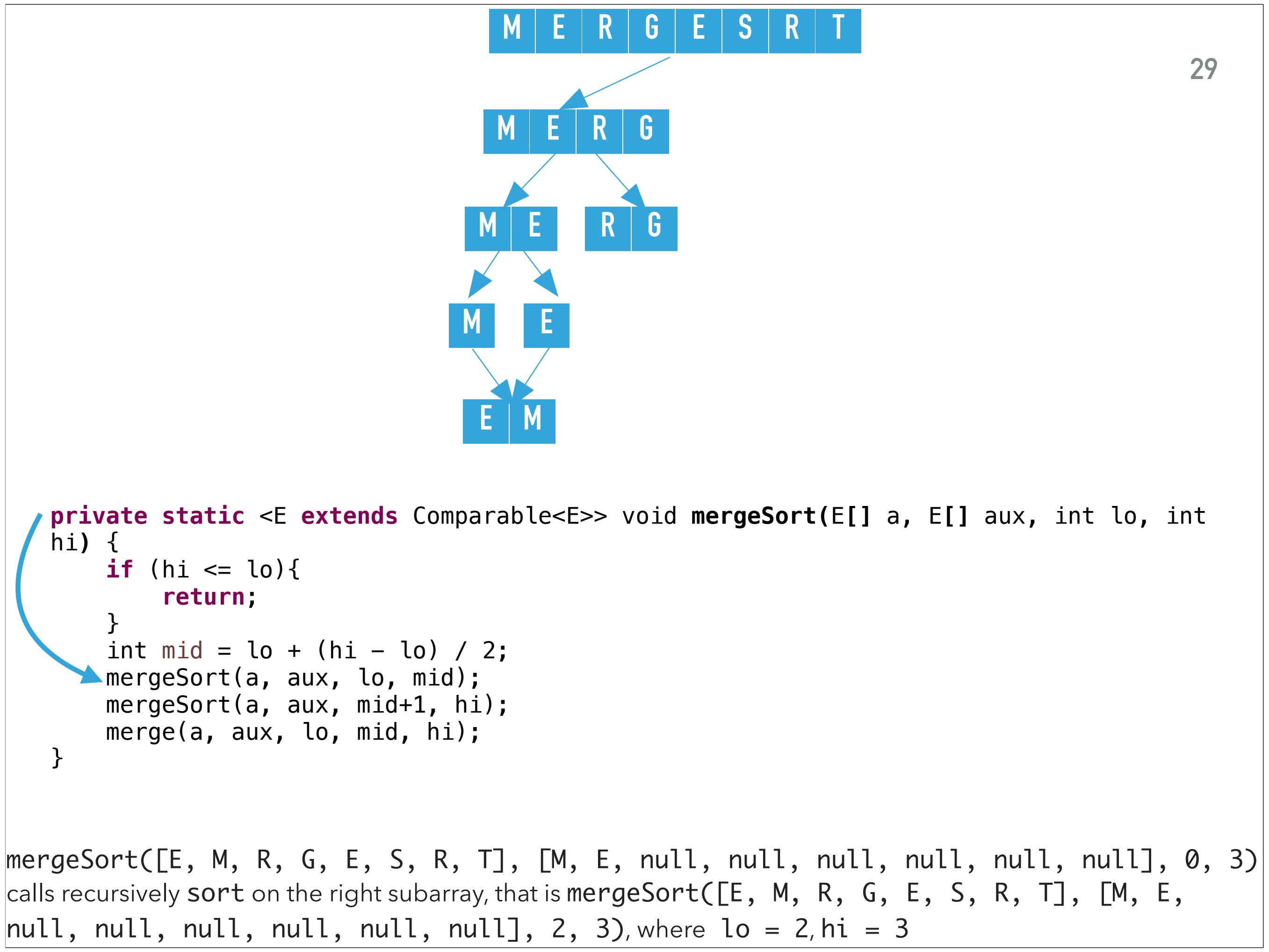
We hit the base case and return.



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

`mergeSort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 1)` merges the two subarrays that is 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]`.

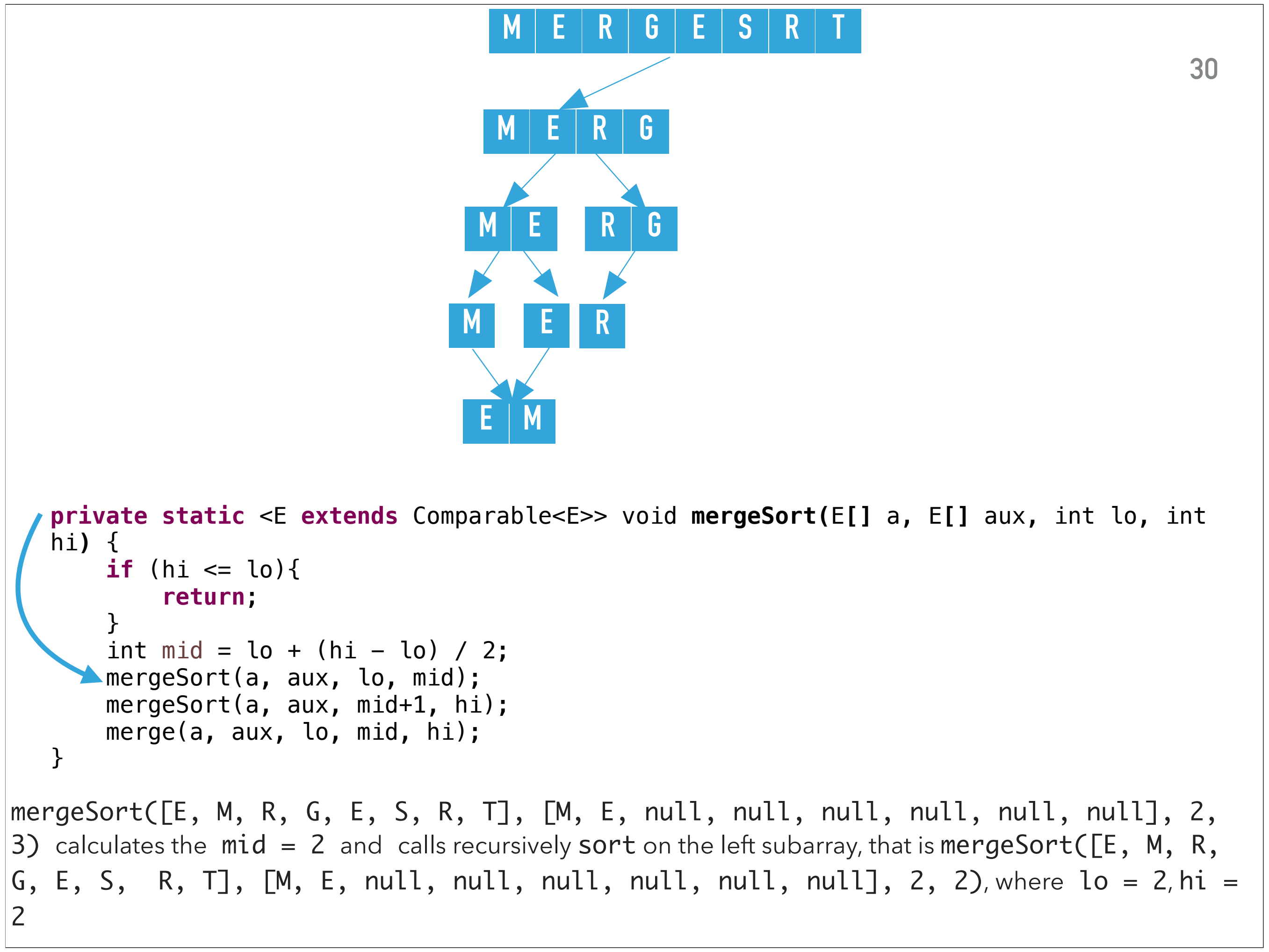
We are now ready to merge M and E which will swap them to E and M resulting to the partially sorted subarray [E,M].



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

`mergeSort([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 `mergeSort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 3)`, where `lo = 2`, `hi = 3`

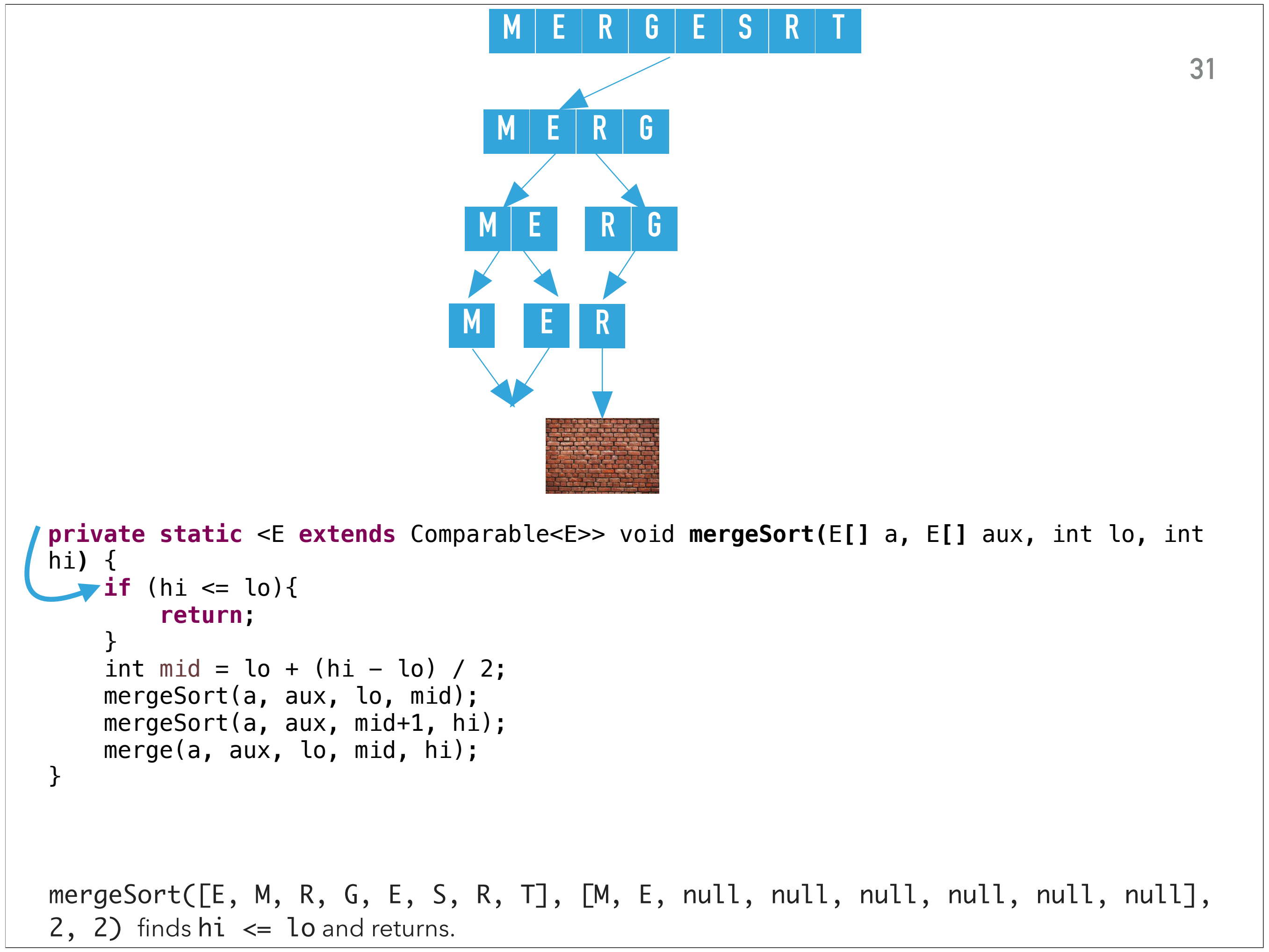
It's time to work on the right subarray of [MERG], [RG].



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

mergeSort([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 mergeSort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 2), where lo = 2, hi = 2

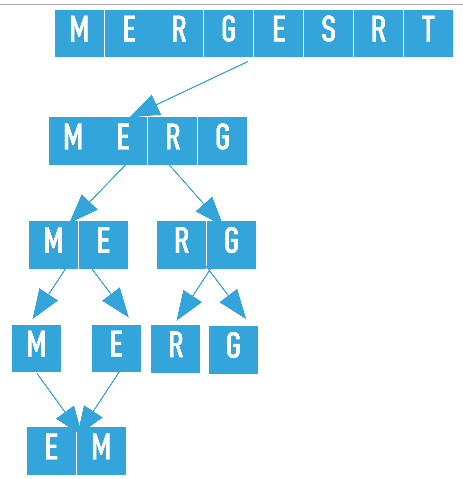
We call mergeSort recursively on the left subarray, [R]



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

`mergeSort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 2, 2)` finds `hi <= lo` and returns.

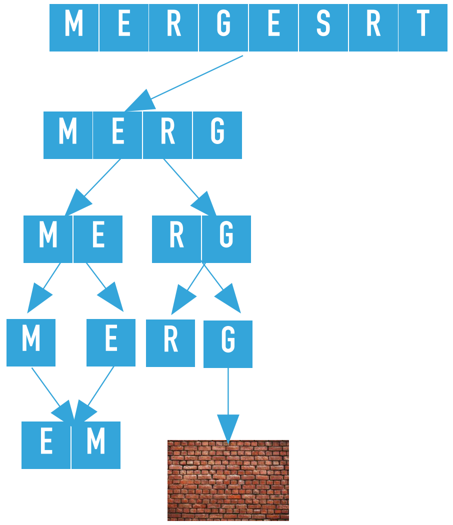
We hit the base case and return.



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

`mergeSort([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 `mergeSort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 3, 3)`, where `lo = 3`, `hi = 3`

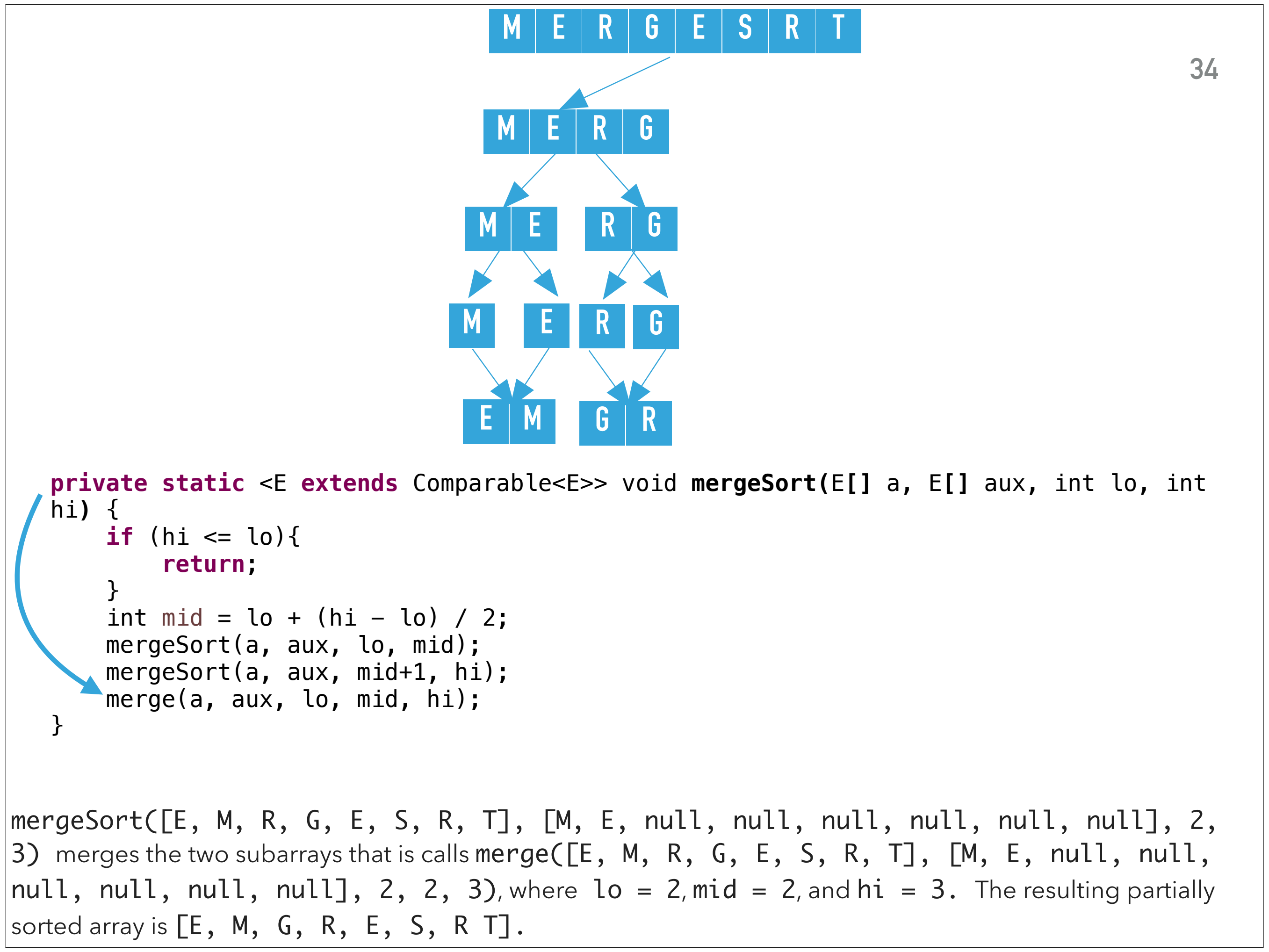
We now call mergeSort on the right subarray of [R,G], [G].



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

`mergeSort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null], 3, 3)` finds `hi <= lo` and returns.

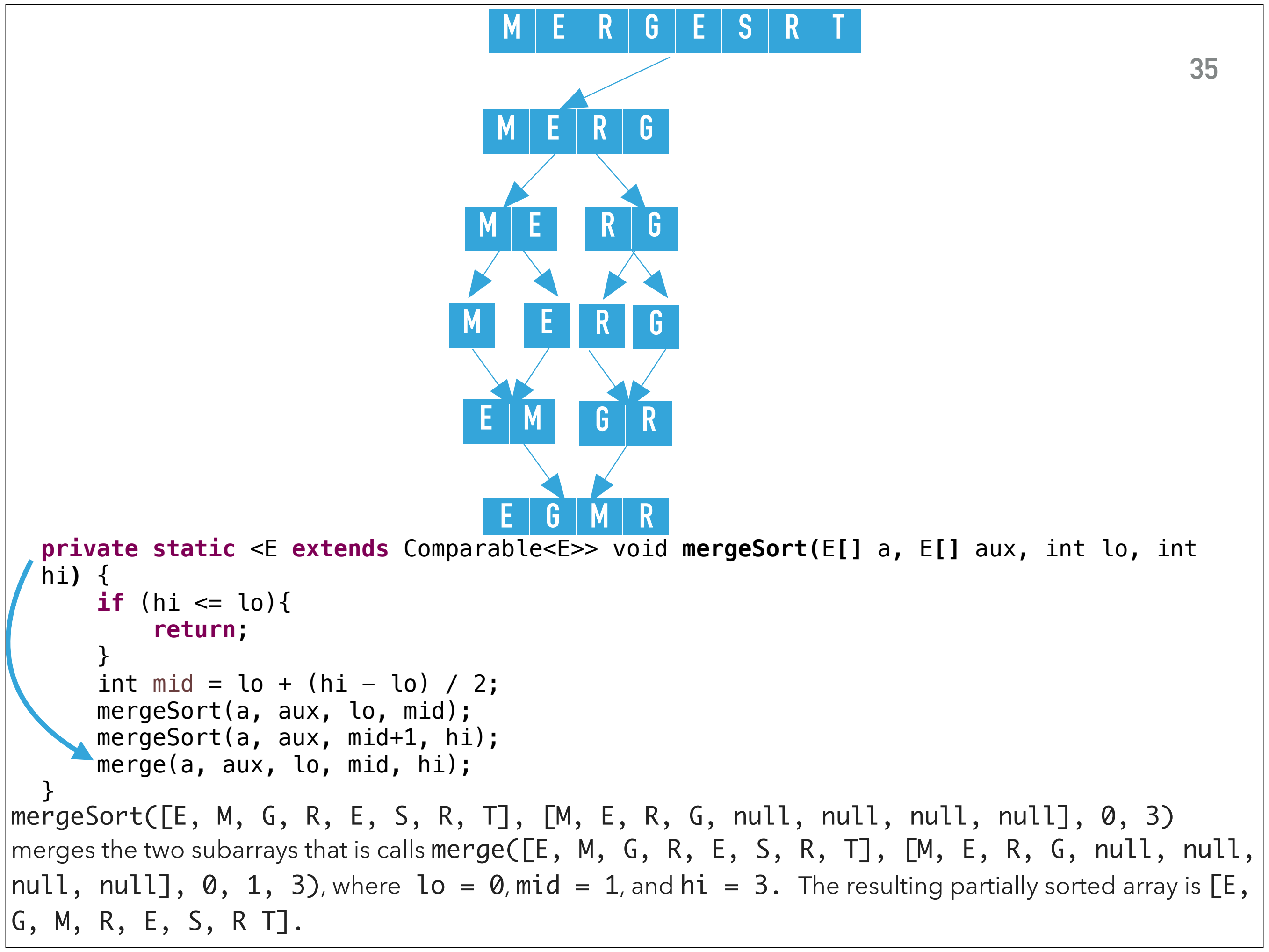
We hit the base case and return.



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

`mergeSort([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]`.

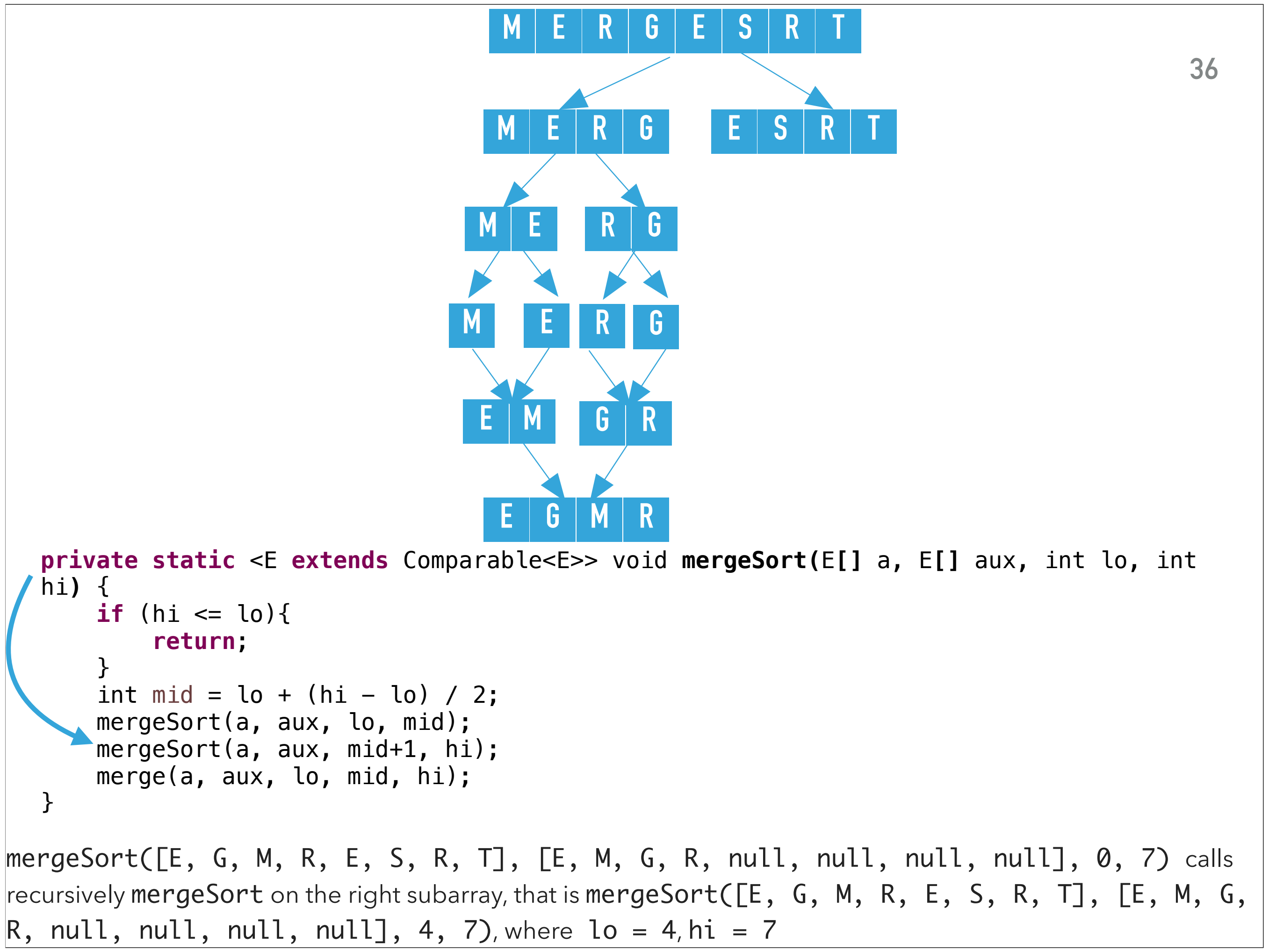
We are now ready to merge R and G and merge swaps them to [G,R].



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

mergeSort([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].

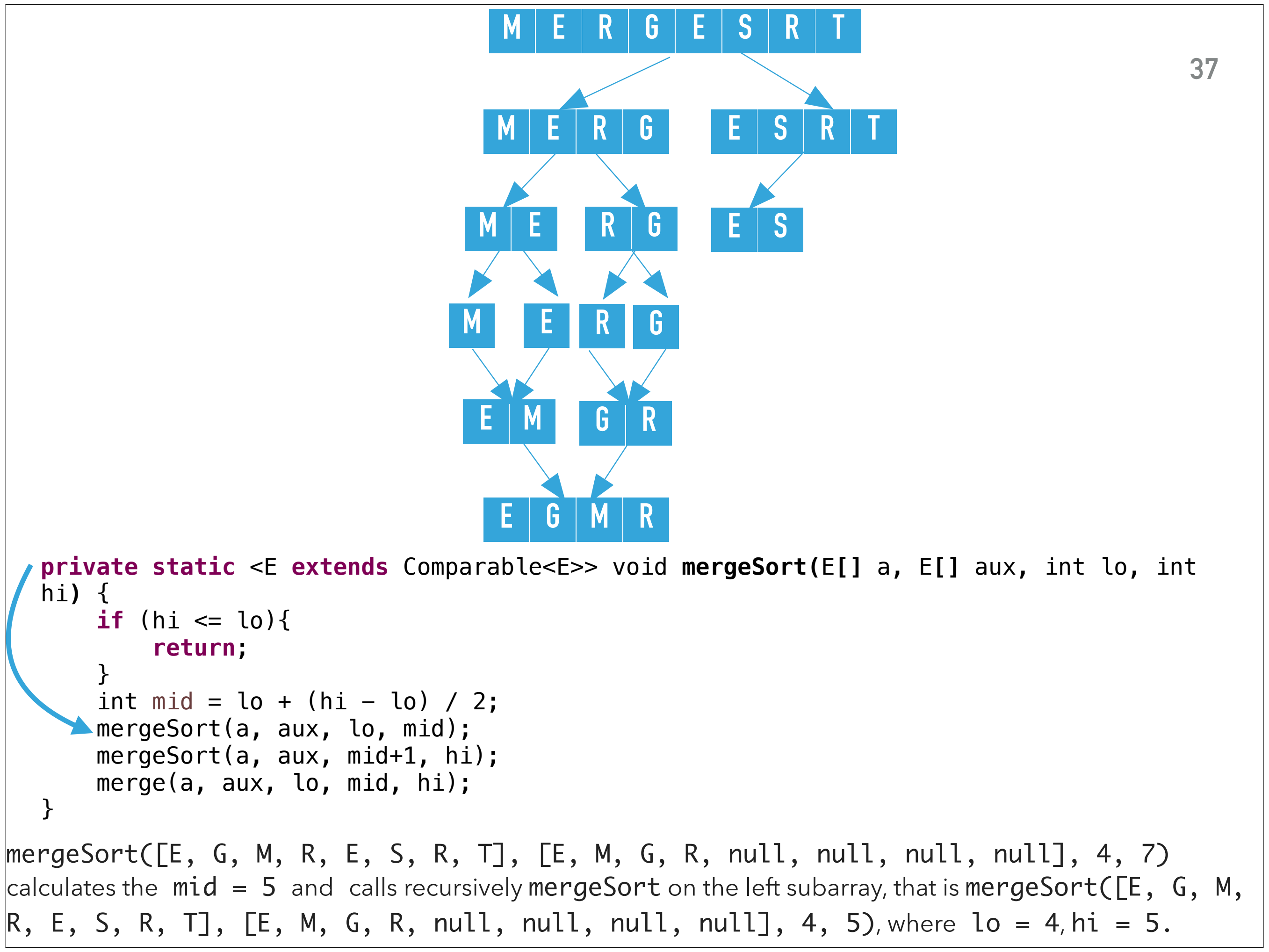
We are now ready to merge the subarrays [E,M] and [G,R], which will result to [E,G,M,R]. (Make sure you know how merge did that!).



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

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

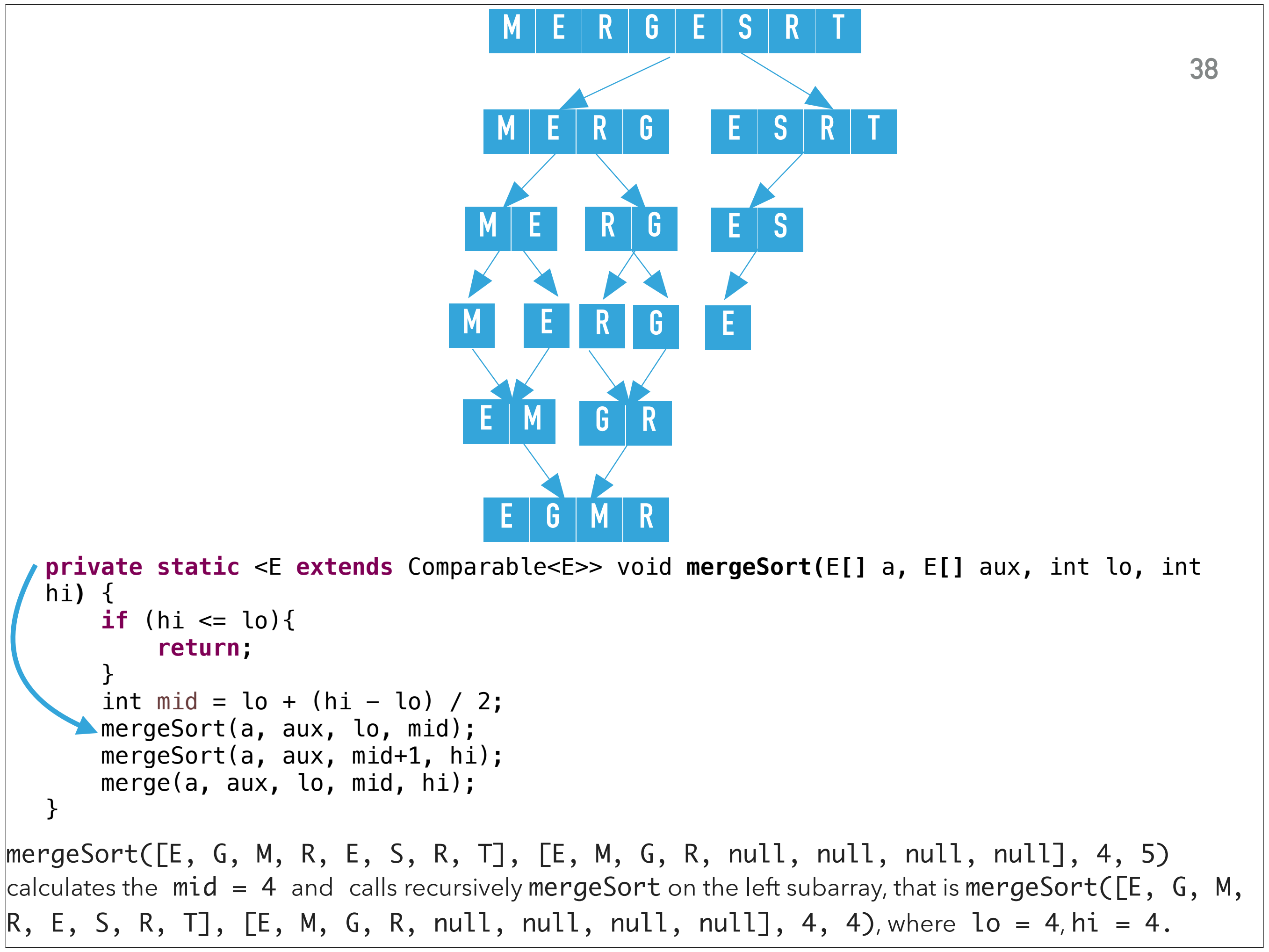
OK, we finished sorting the left subarray of [M,E,R,G,E,S,R,T], time to sort the right subarray, that is [E,S,R,T].



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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)`, where `lo = 4`, `hi = 5`.

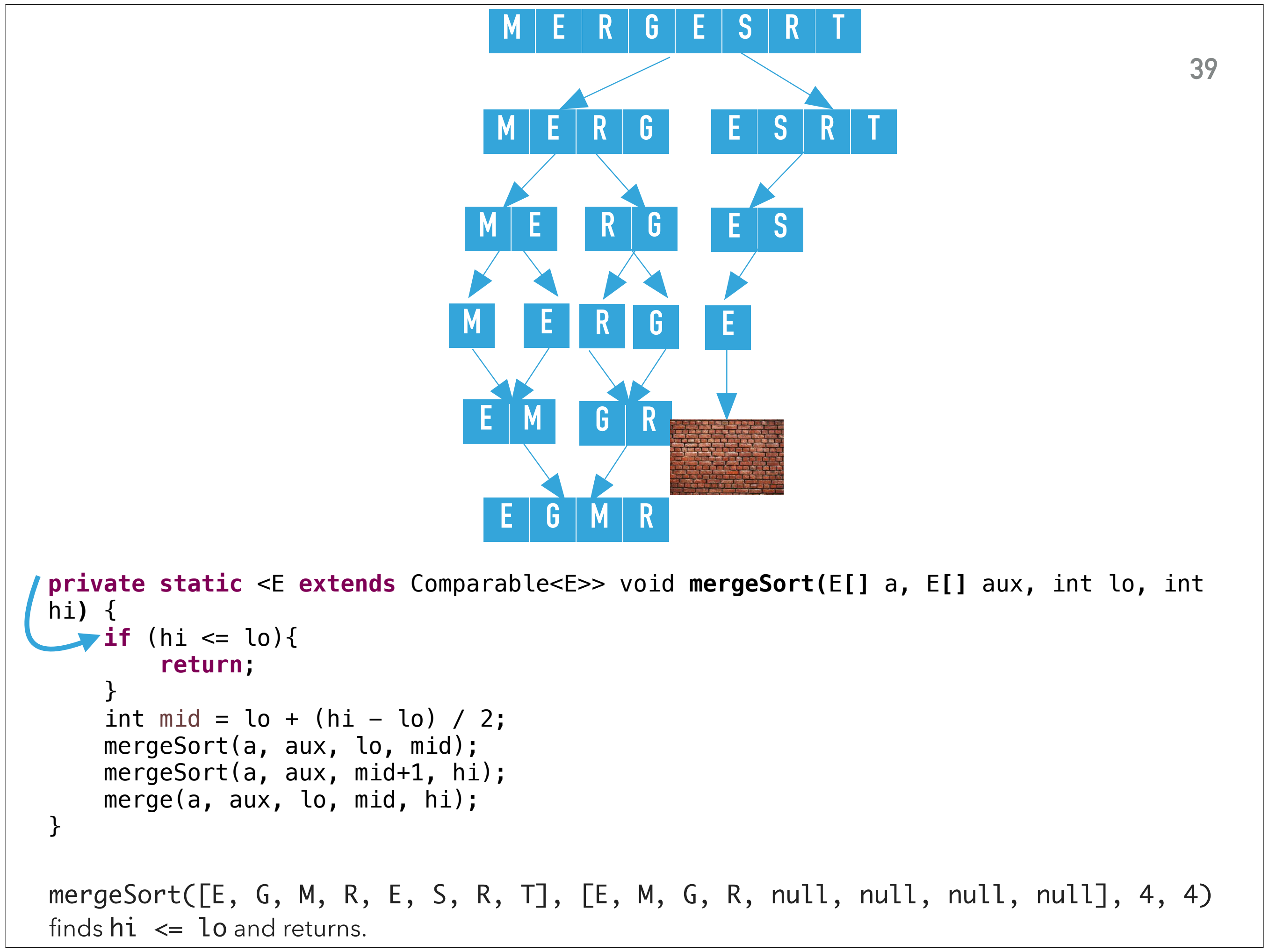
We call merge sort recursively on [E,S].



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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)`, where `lo = 4`, `hi = 4`.

And then on E.

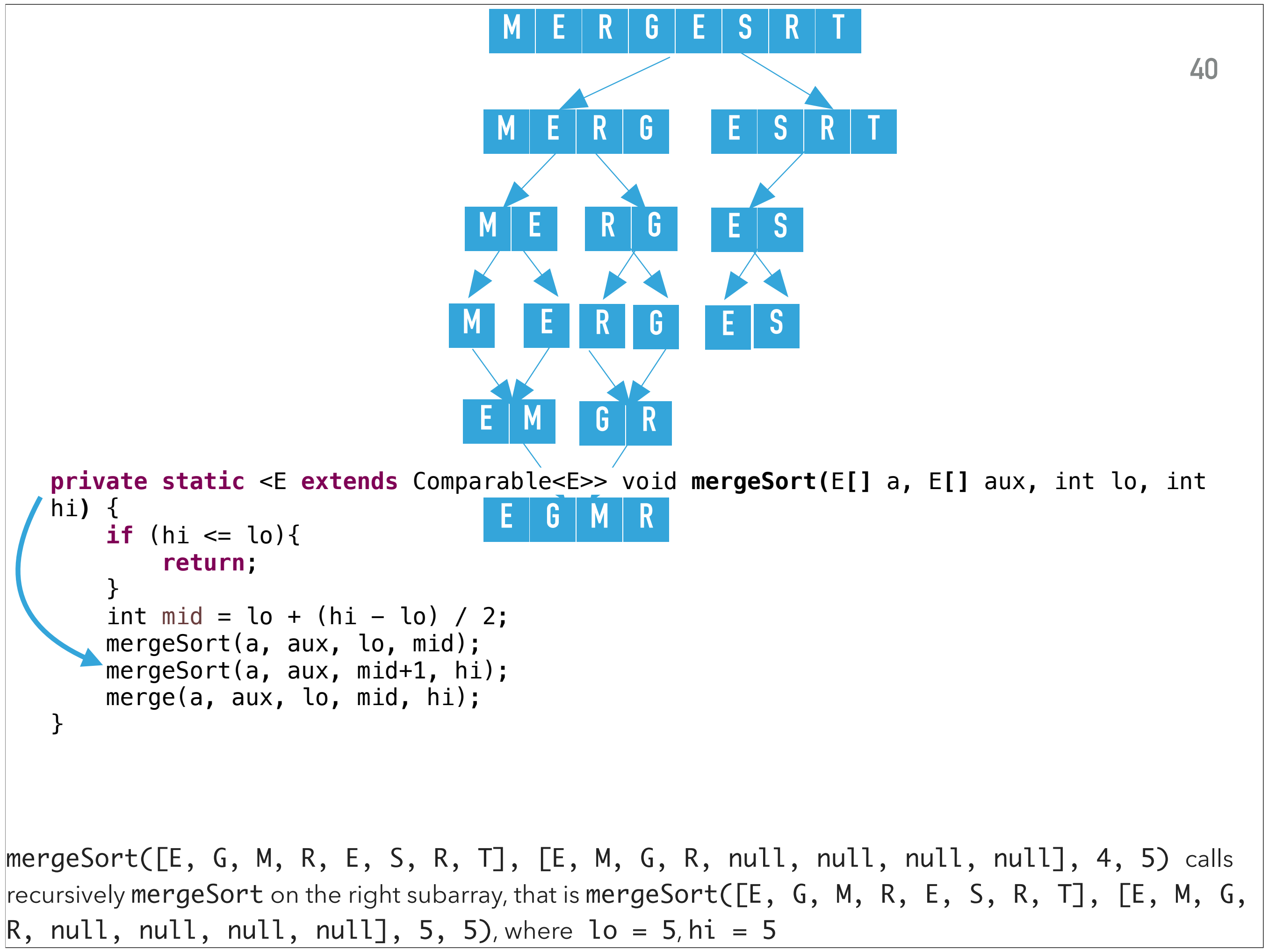


```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

```
mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)
```

finds `hi <= lo` and returns.

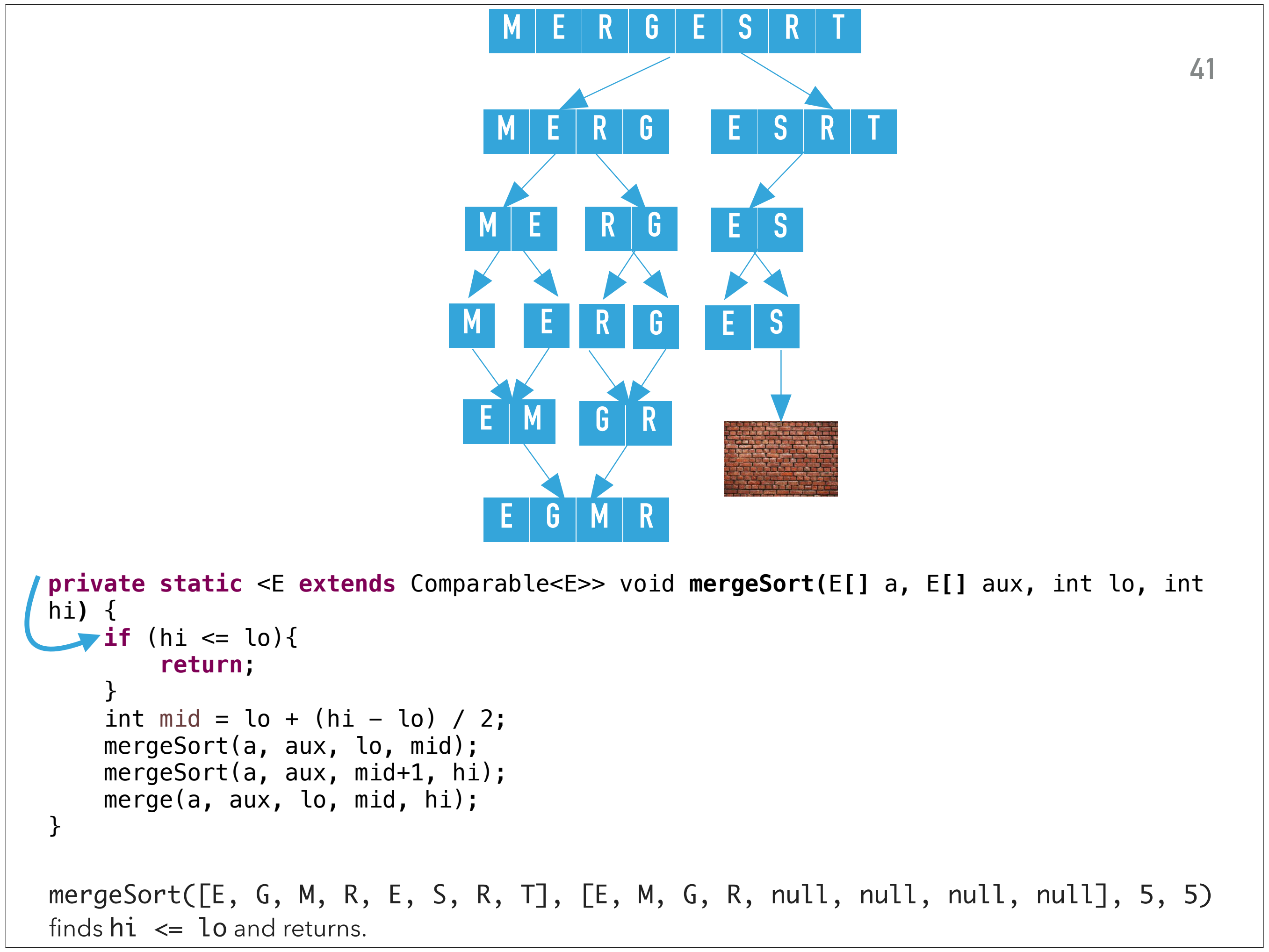
Which then reaches the base case and returns.



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

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

Which then reaches the right subarray for S.

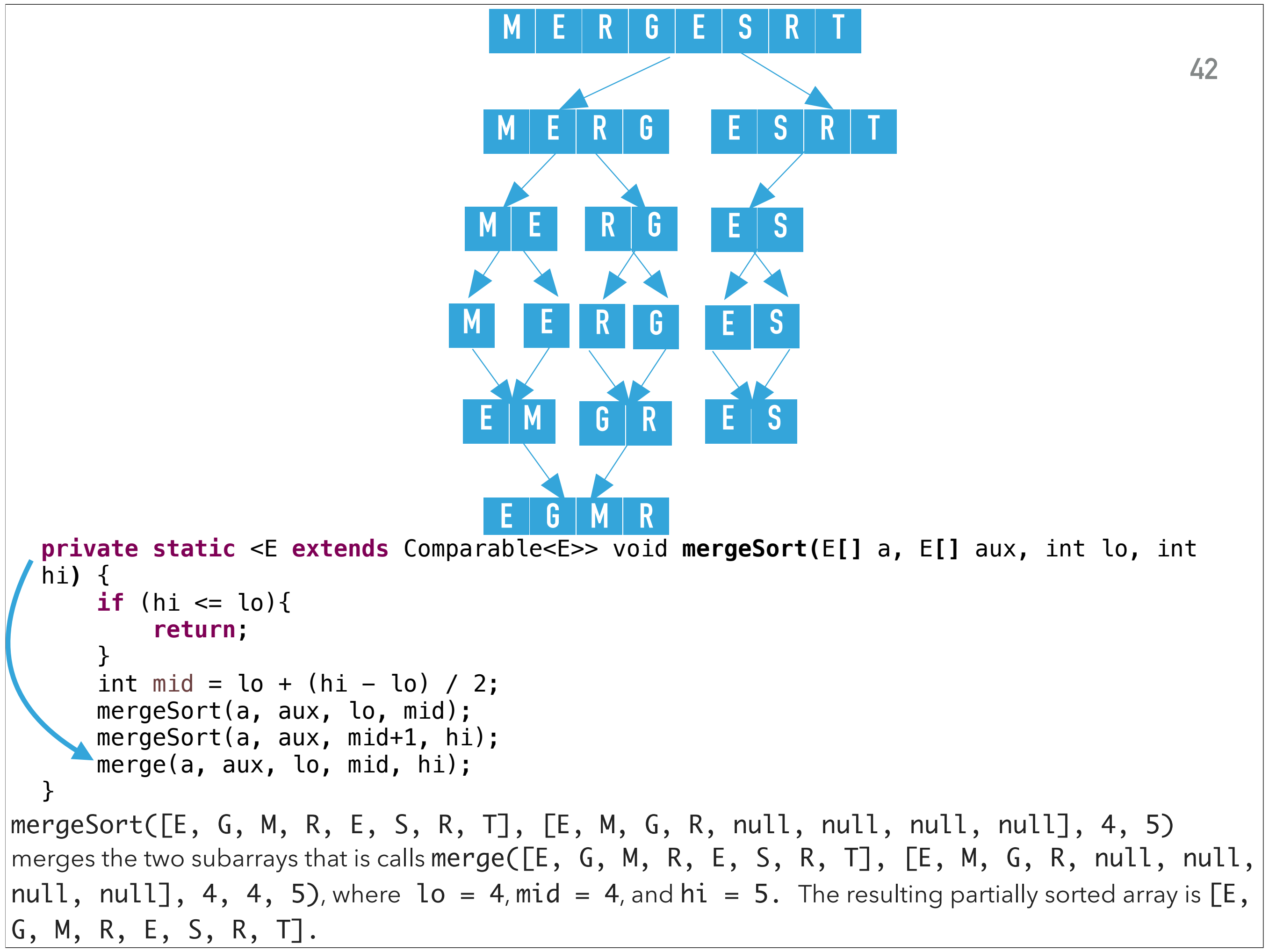


```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

```
mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 5, 5)
```

finds `hi <= lo` and returns.

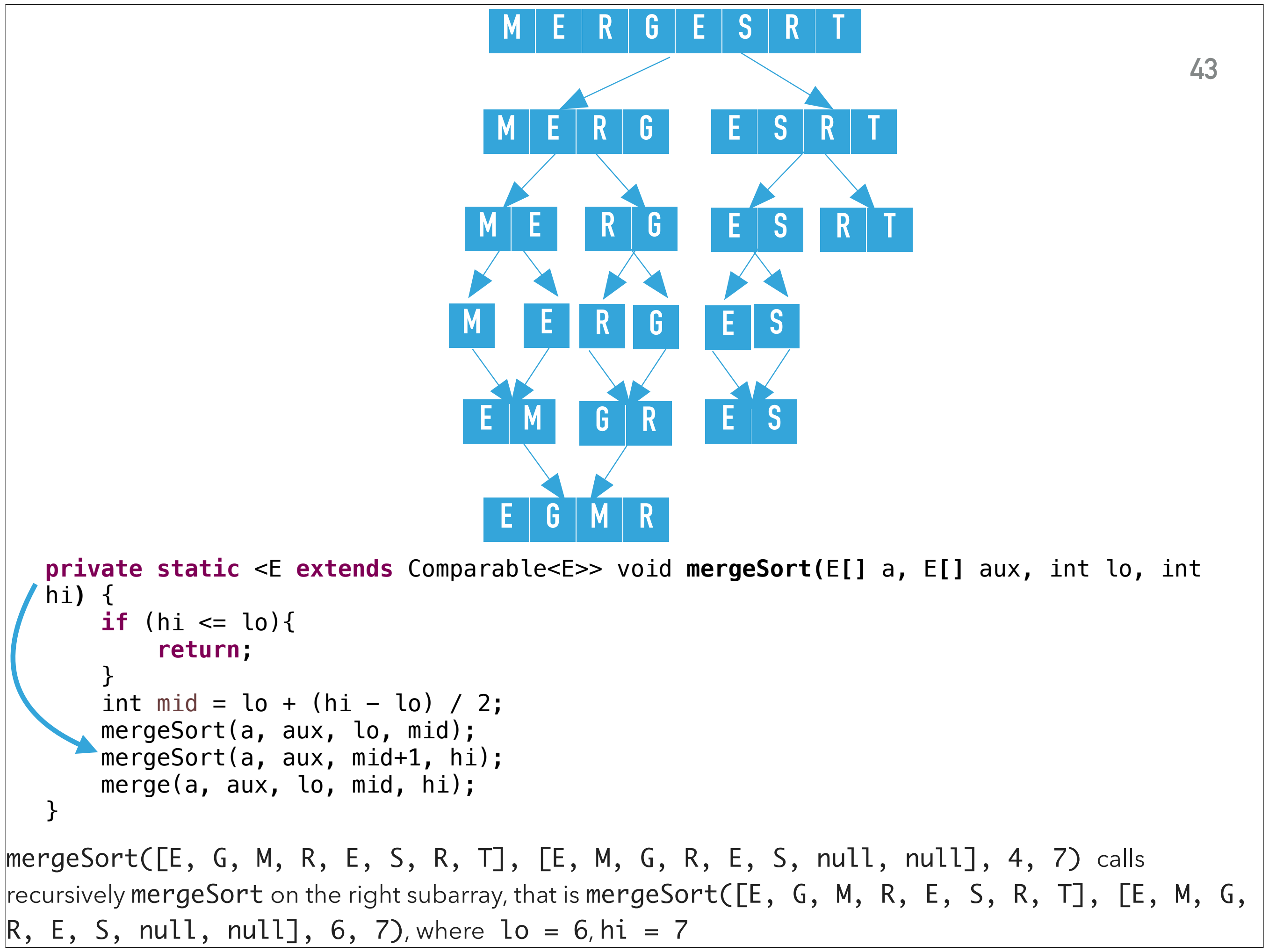
Hits the base case and returns.



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

`mergeSort([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].`

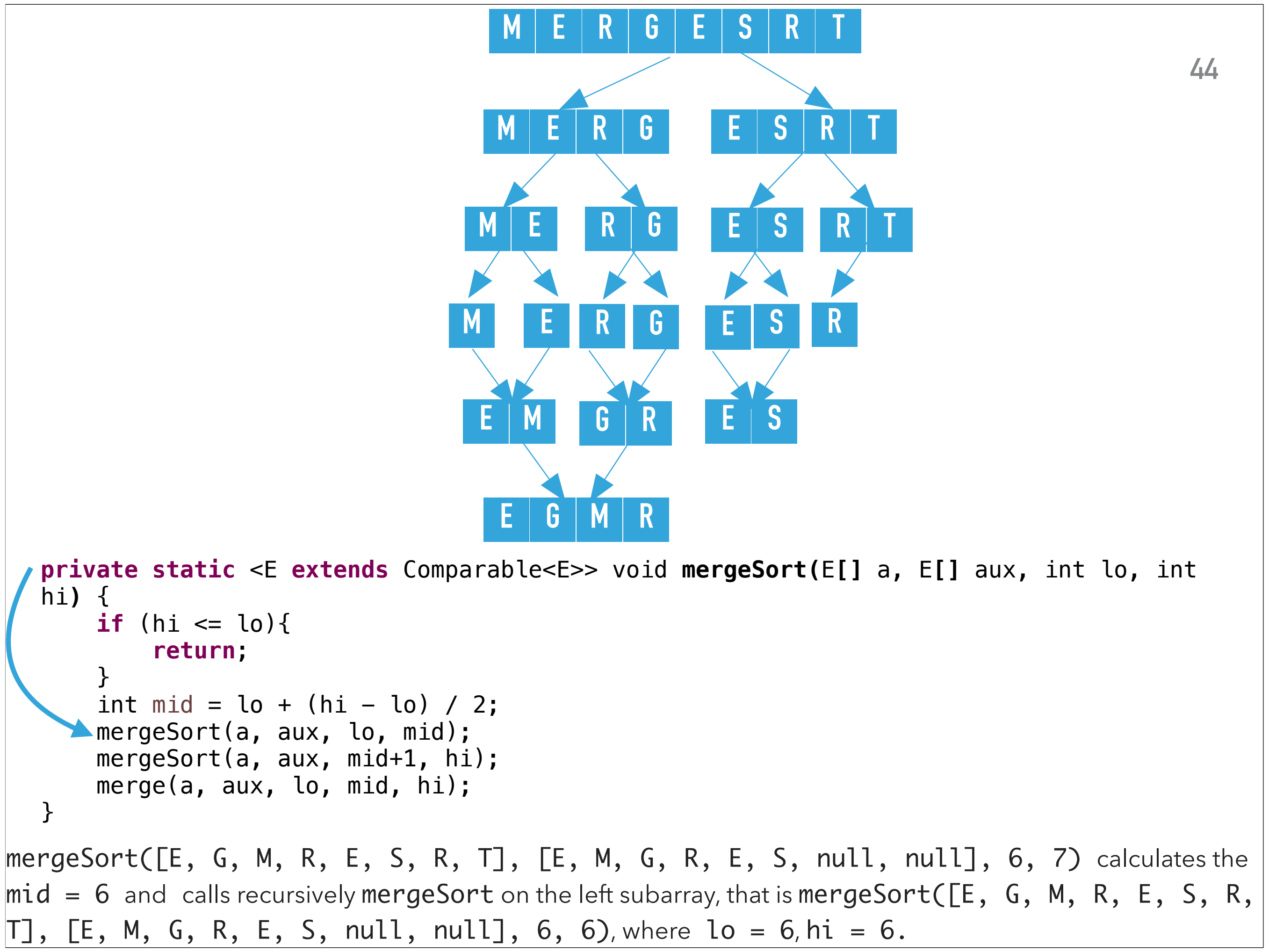
And we now merge E and S into.. E and S, merge didn't change anything here.



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

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

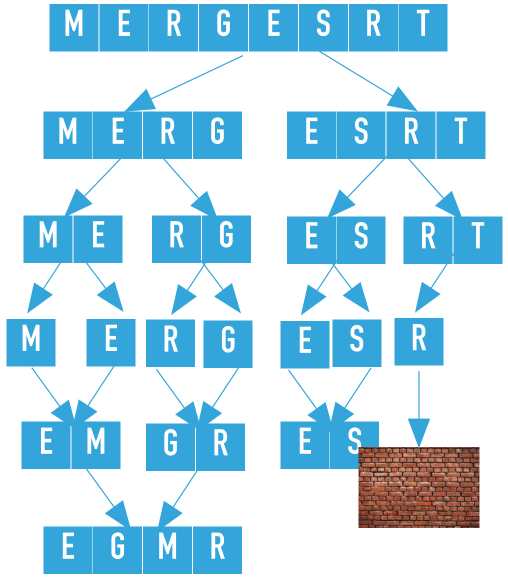
Time to dive into [R,T].



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

`mergeSort([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 `mergeSort` on the left subarray, that is `mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 6, 6)`, where `lo = 6`, `hi = 6`.

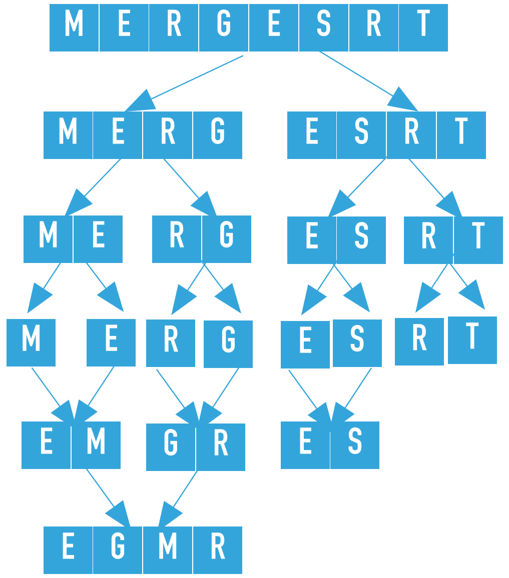
Going into the left subarray R.



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

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

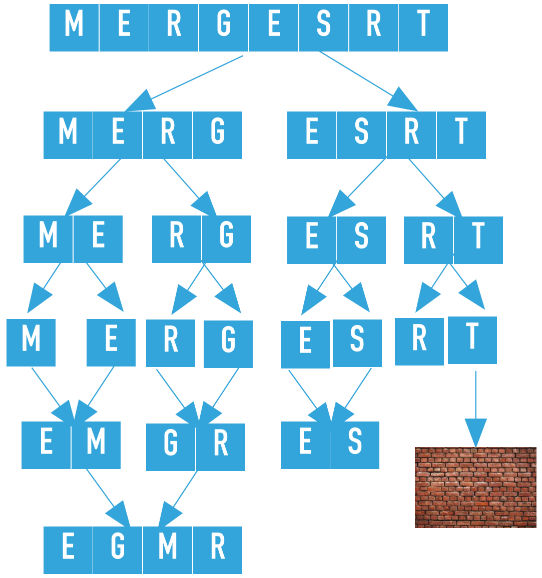
Hitting the base case and returning.



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

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

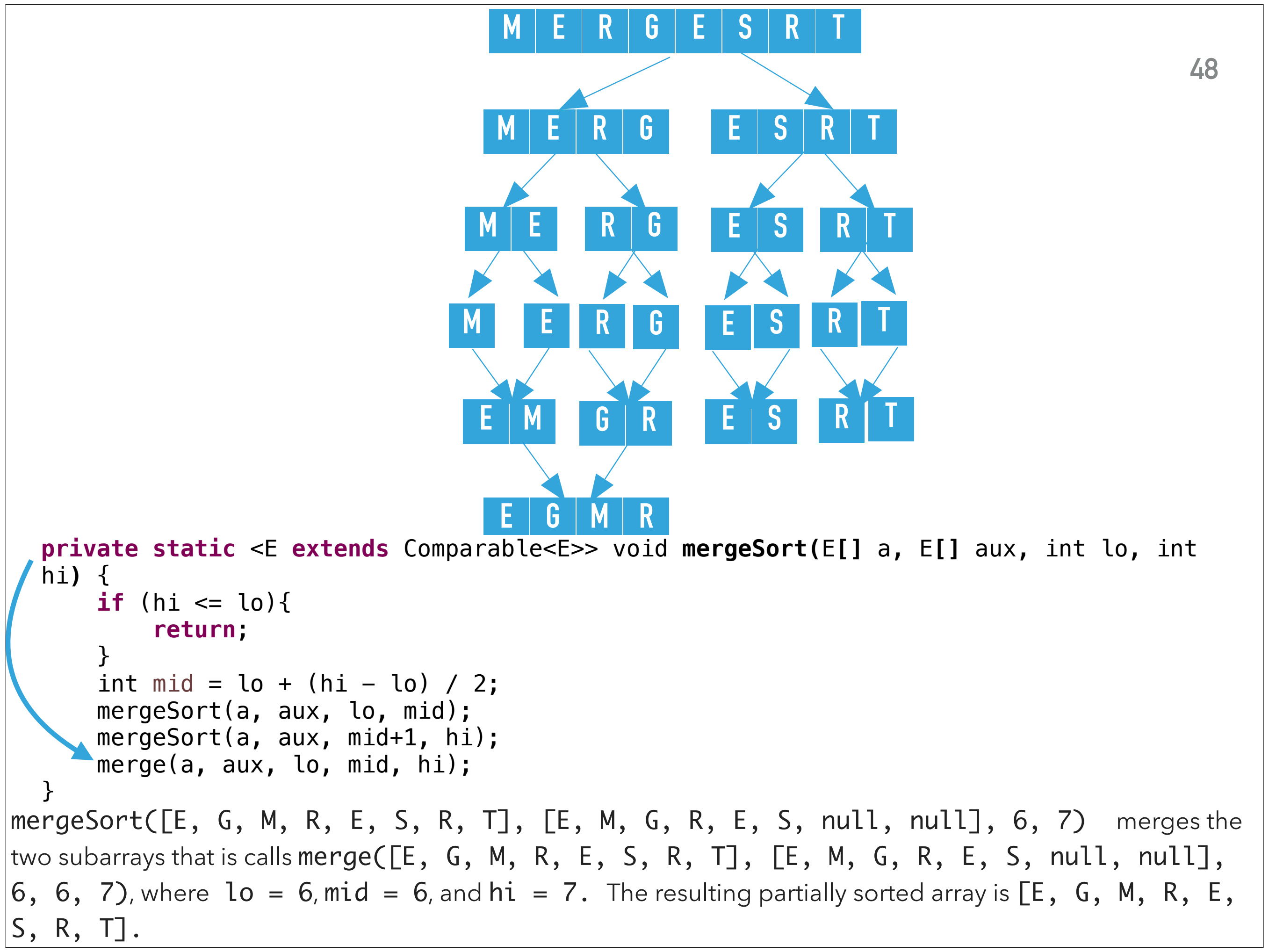
Right subarray T.



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

`mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, E, S, null, null], 7, 7)` finds `hi <= lo` and returns.

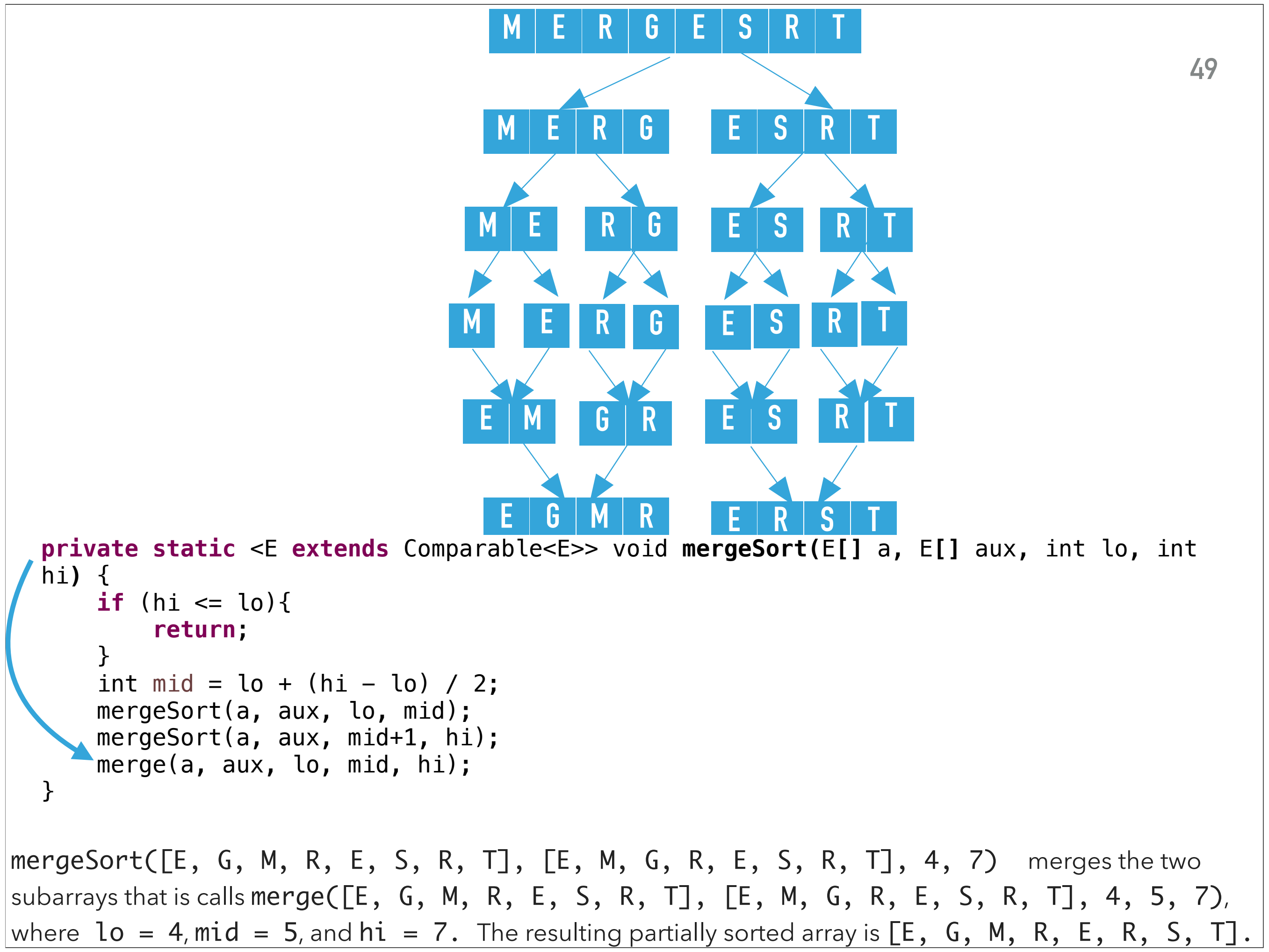
Hitting base case.



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

`mergeSort([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]`.

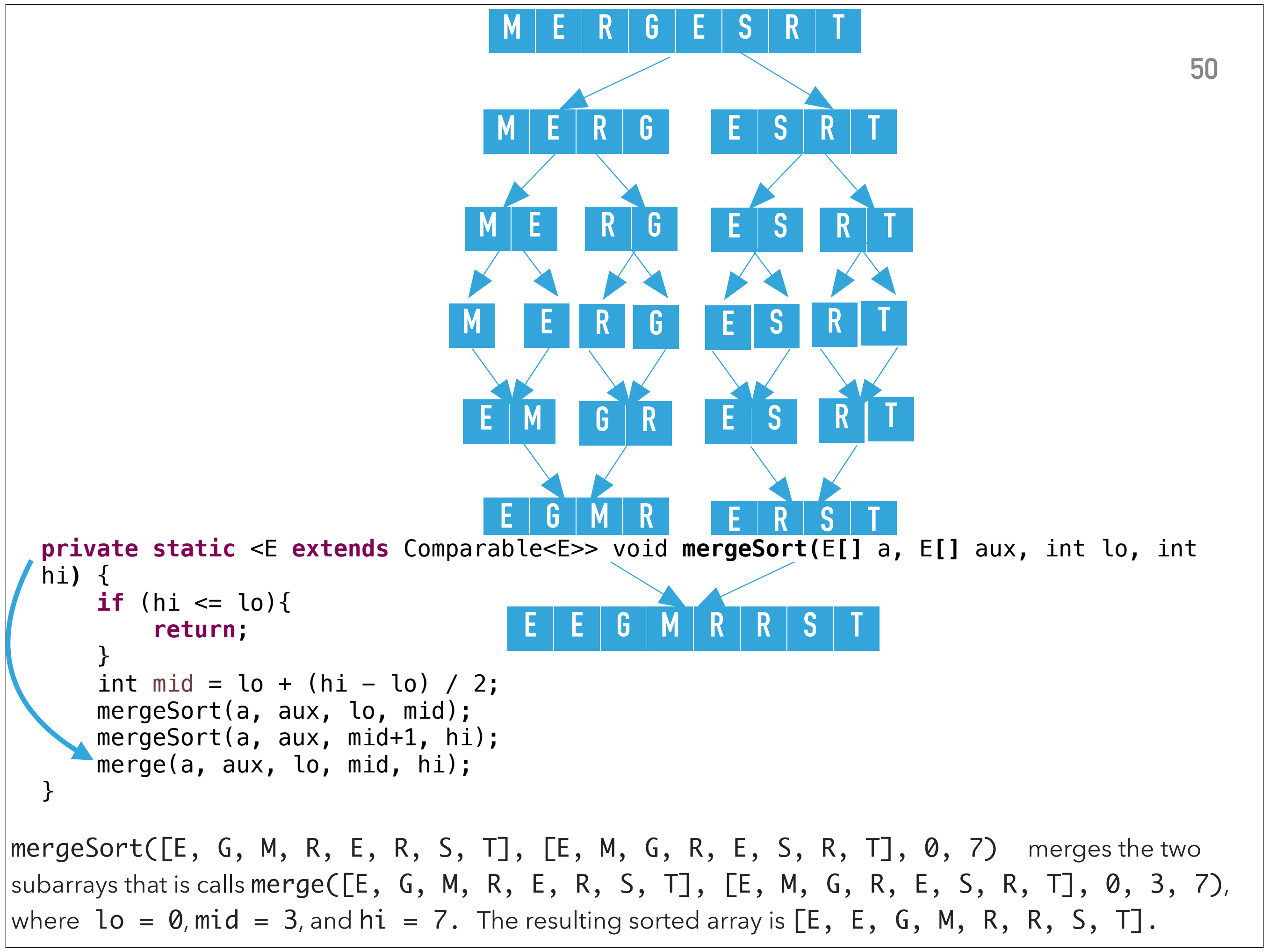
Tine to merge R and T. No swaps.



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

`mergeSort([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]`.

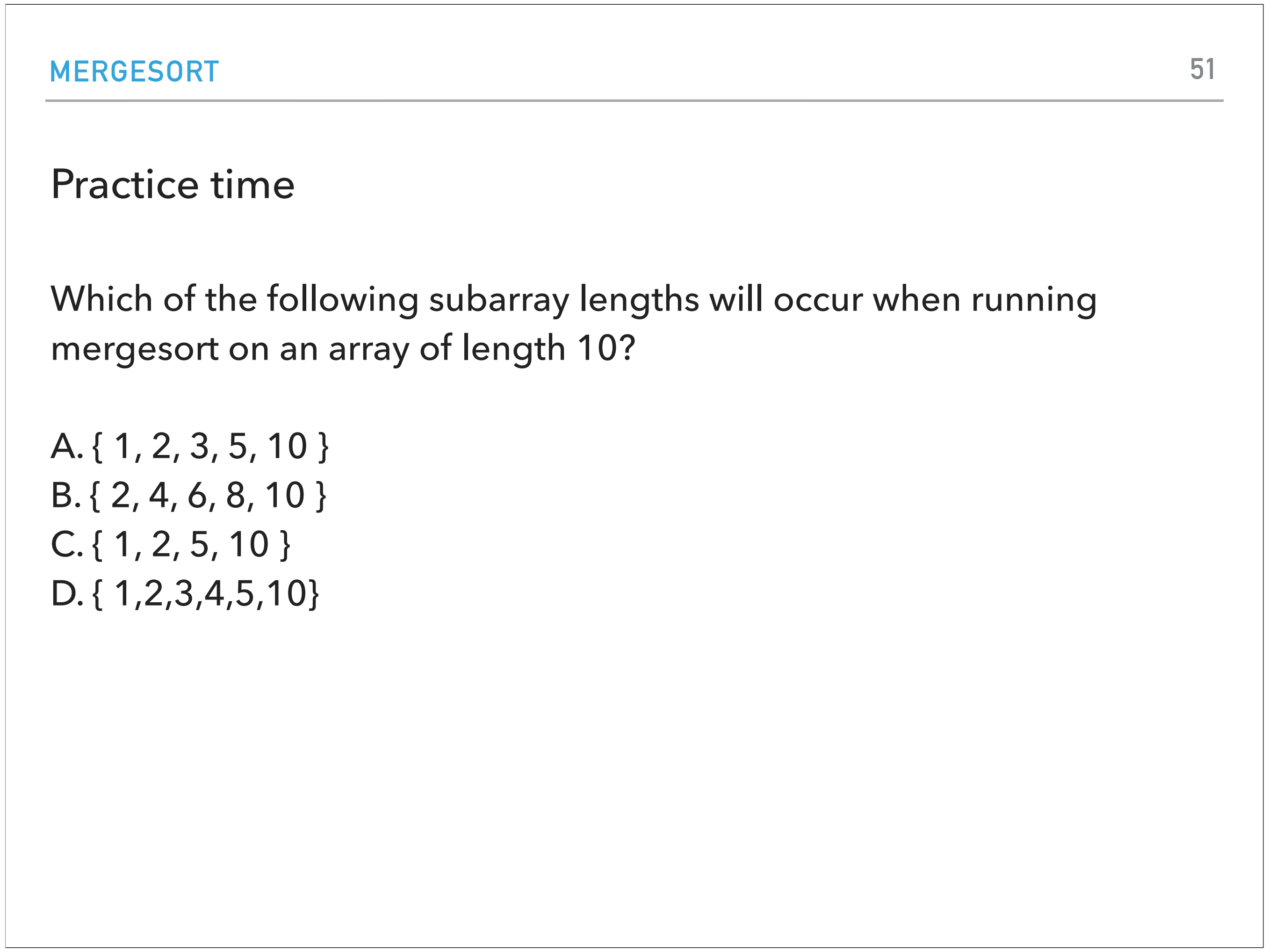
Let's merge [E,S] and [R, T]; that will result to [E,R,S,T].



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

mergeSort([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].

And at last we are at the very last step where we now have two sorted halves, [E,G,M,R] and [E,R,S,T] which we merge to [E,E,G,M,R,S,T]! TADA!



Practice time

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

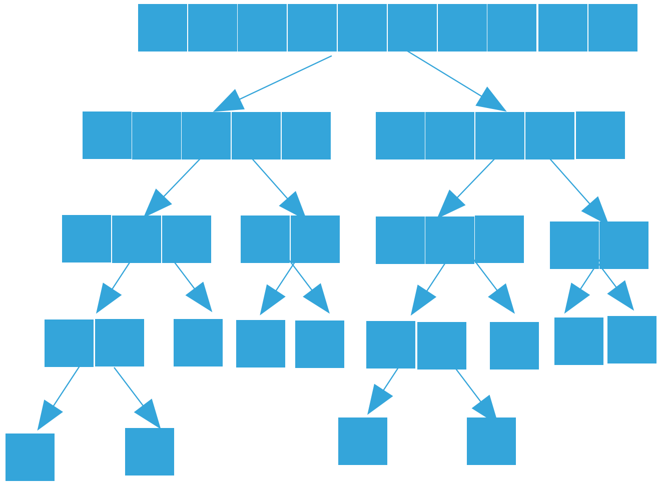
A. { 1, 2, 3, 5, 10 }
B. { 2, 4, 6, 8, 10 }
C. { 1, 2, 5, 10 }
D. { 1,2,3,4,5,10}

Here's a question that focuses on LENGTHS of subarrays.

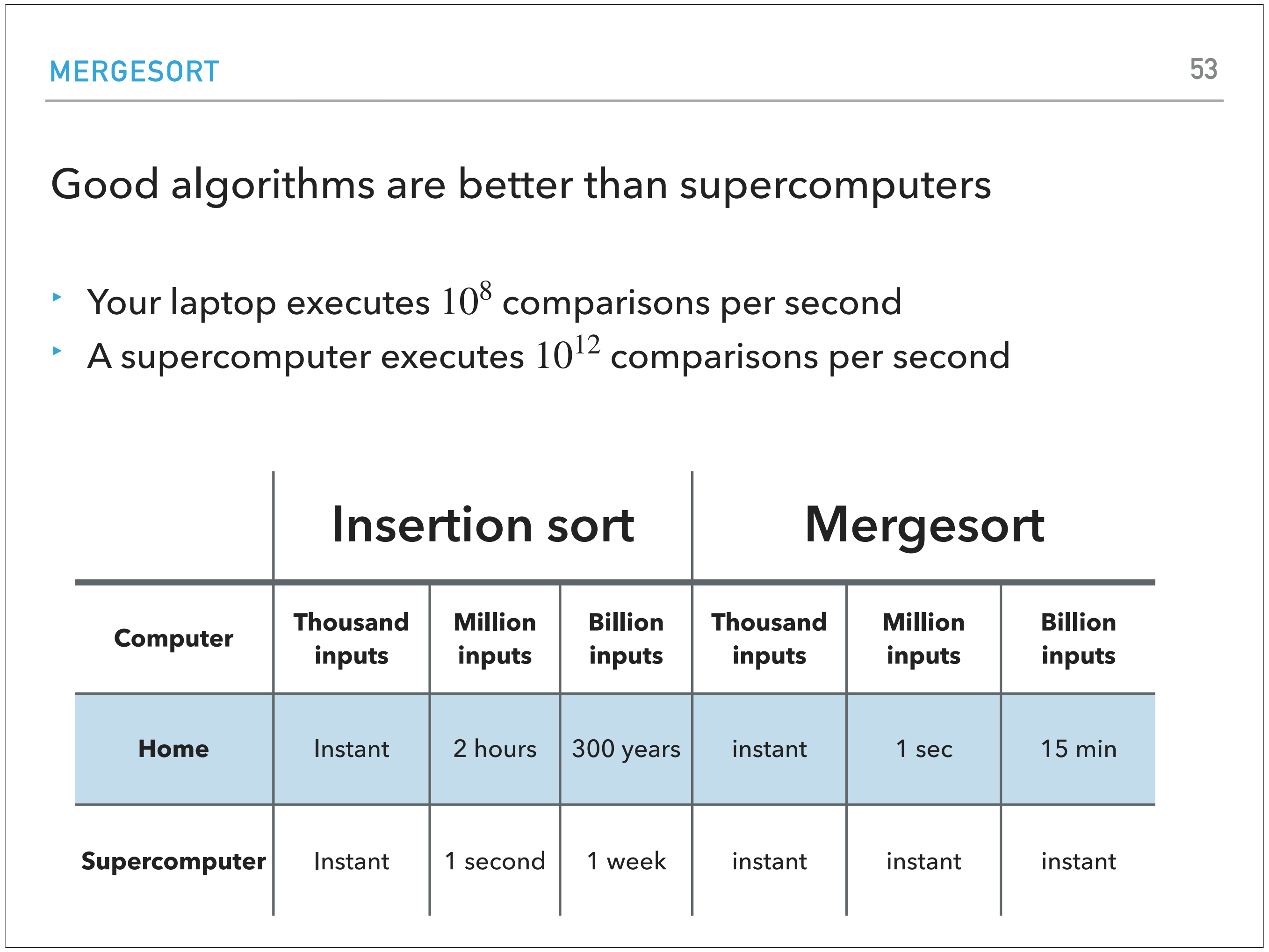
Answer

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

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



The correct answer is A and here is a schematic proof.

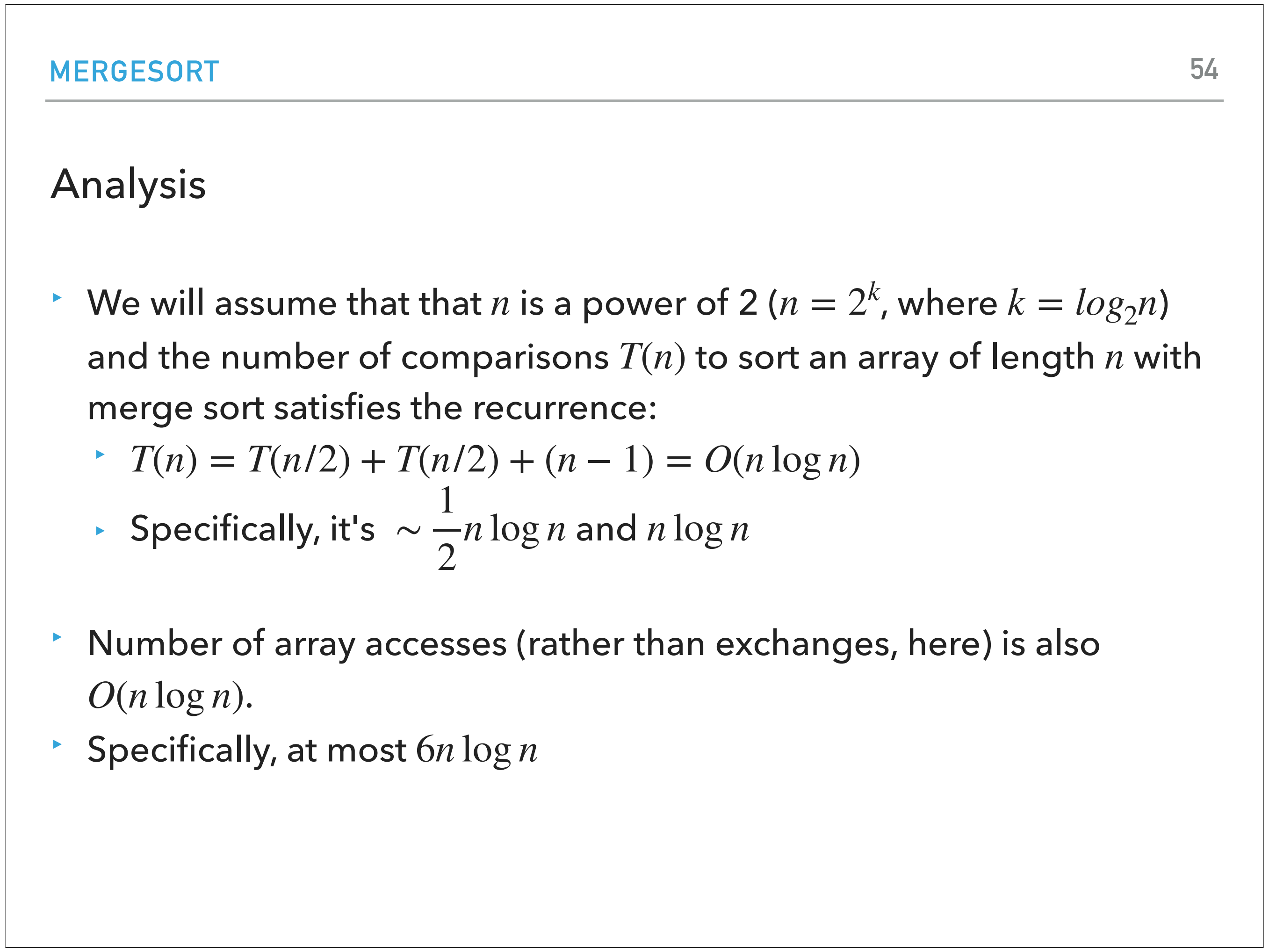


Good algorithms are better than supercomputers

- 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

Let's put in context how good merge sort is by comparing it to insertion sort and seeing how the both fare on a home computer versus a super computer. As you can see, good algorithms are far superior than fancy computers. Mergesort does SIGNIFICANTLY better than insertion sort.



Analysis

- We will assume that 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:
  - $T(n) = T(n/2) + T(n/2) + (n-1) = O(n \log n)$
  - Specifically, it's $\sim \frac{1}{2} n \log n$ and $n \log n$
- Number of array accesses (rather than exchanges, here) is also $O(n \log n)$.
- Specifically, at most $6n \log n$

Beyond the practical evidence of merge sort being fast, there is of course a theoretical analysis that proves that. If we assume that our array has n elements and n is a power of 2 (i,e. n can be written as 2^k; Result holds for all n, but powers of 2 make our life easier), then the number of comparisons we will need for an array of length n will be equal to the number of comparisons for each half plus n-1 for the merging. We won't do the proof, but there is a theorem called the master theorem that shows that this results in linearithmic number of comparisons, i.e. O(nlogn). Specifically it's between 1/2nlogn and nlogn.

The number of array accesses (rather than exchanges) is also linearithmic, specifically at most 6nlogn (2n for the copy to the auxiliary array, 2n for the copy back to the original array, and 2n for comparisons).

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

If $n = 4$, 2 levels

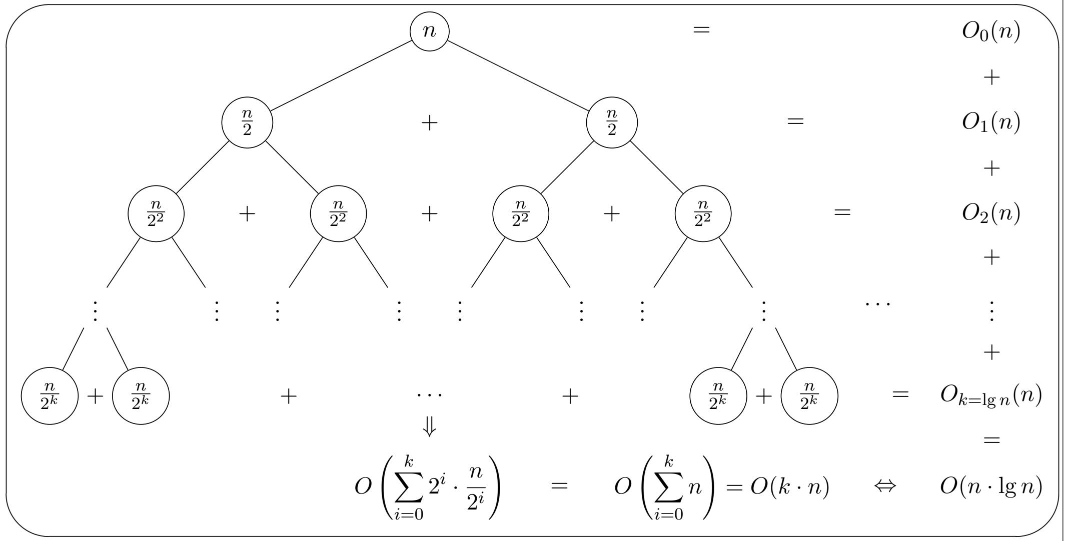
If $n = 8$, 3 levels

If $n = 16$, 4 levels

...

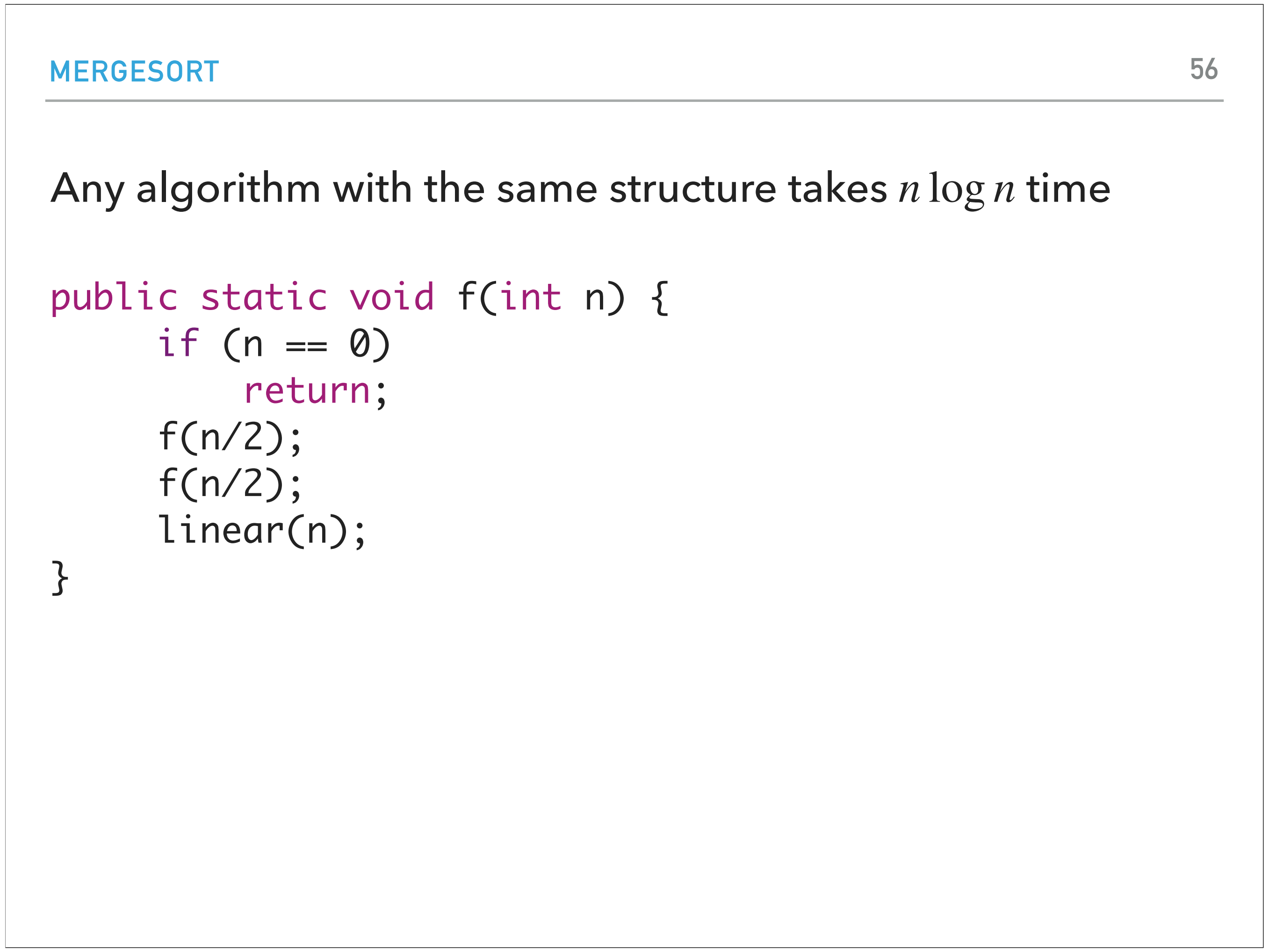
If $n = 2^k$, k levels,

or $k = log_2 n$


n
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^k}$
$=$
$+$
$O_0(n)$
$O_1(n)$
$O_2(n)$
$O_{k=\lg n}(n)$
$\cdots$
$\Downarrow$
$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \lg n)$

http://www.texample.net/media/tikz/examples/PDF/merge-sort-recursion-tree.pdf

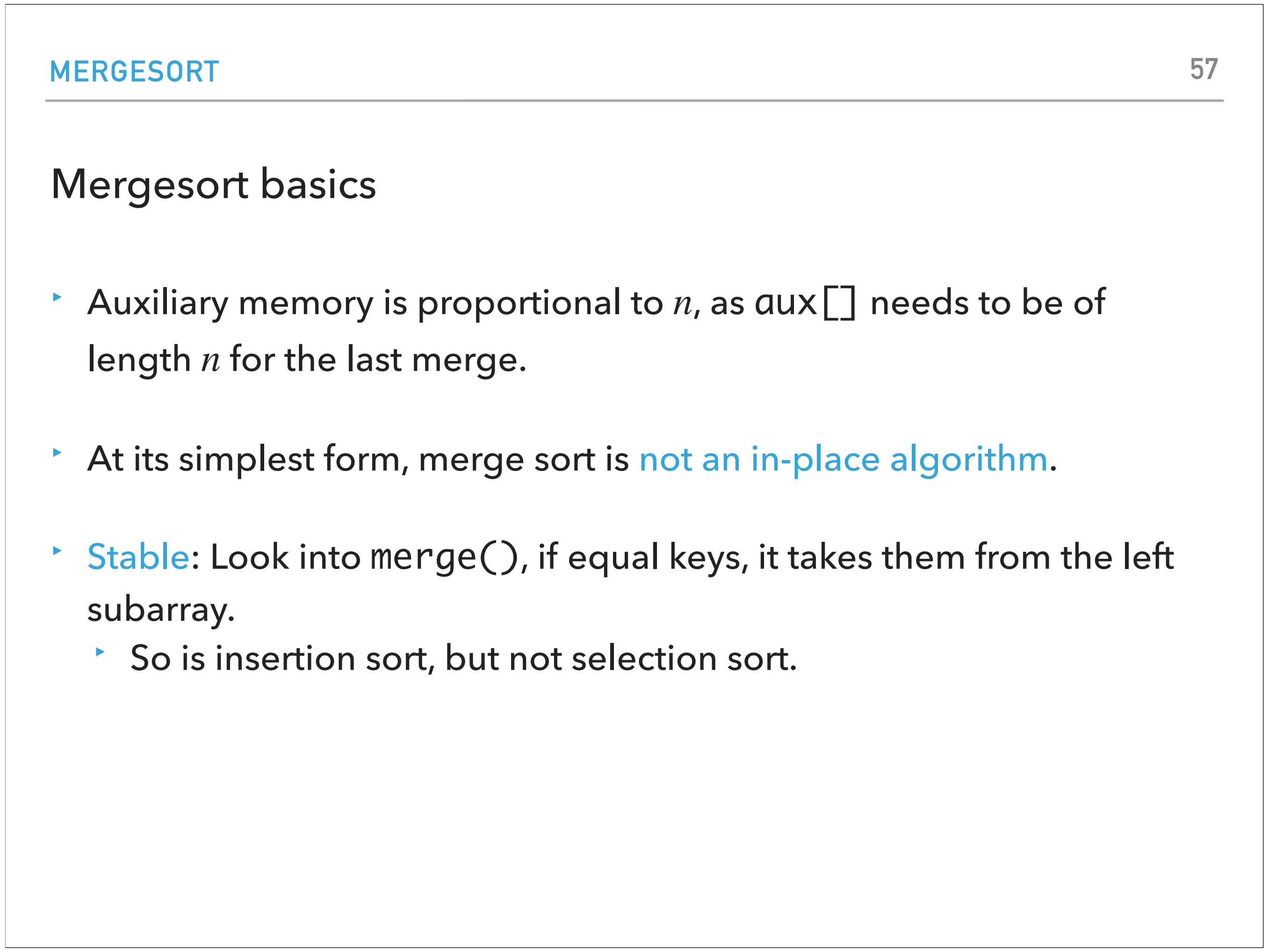
Here is a graphical depiction of this linearithmic complexity. If you think about it, if n=4, we have two levels. If n=8, we have 3 levels. In general, if n=2^k , we have k levels, where k=logn. Since each one of those levels takes O(n) work for merging, we have O(nlogn).



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);
}
```

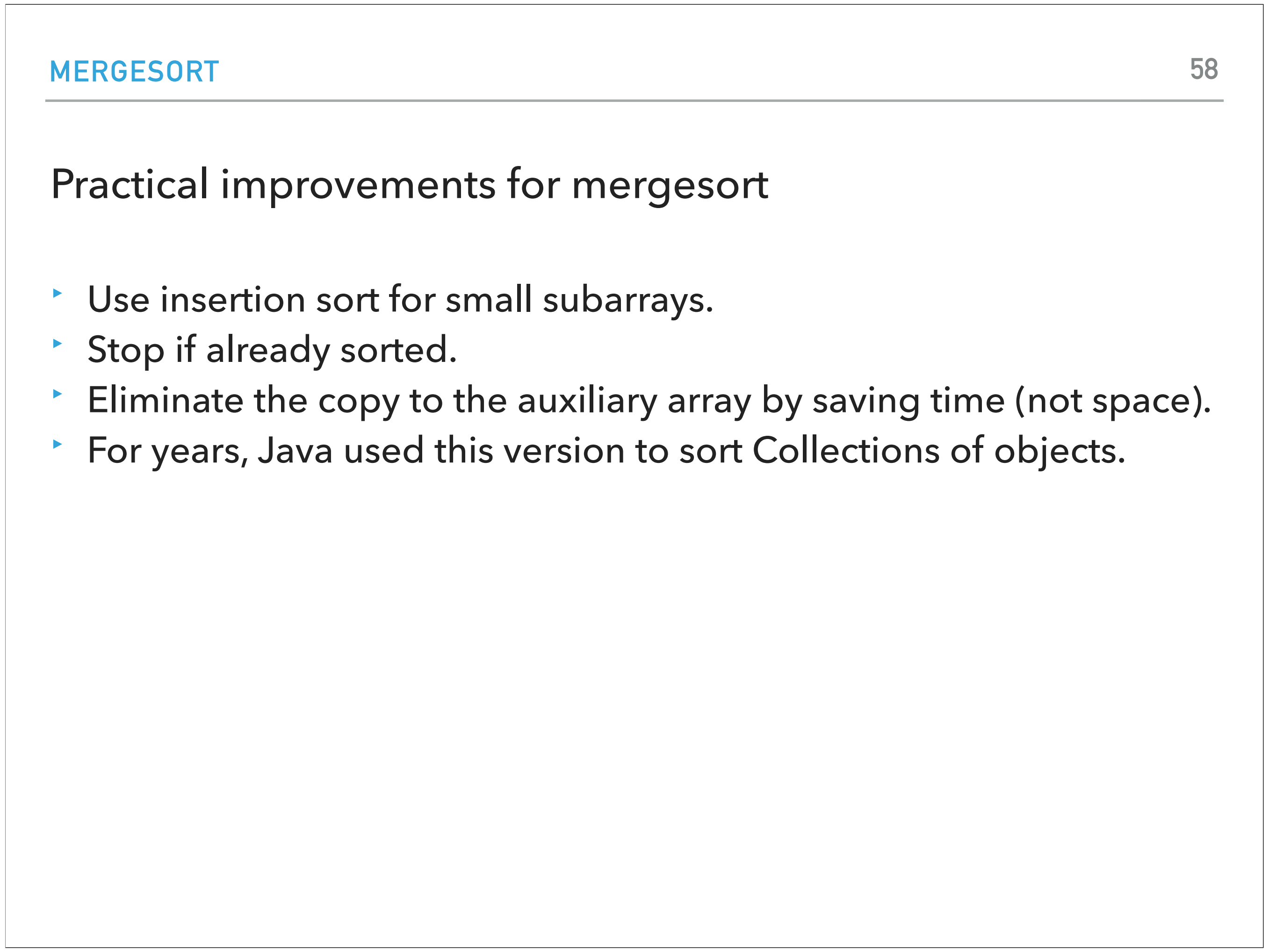
In fact, any algorithm that follows the structure of being called recursively on two problems of half the size of the original and then some linear time work ends up being linearithmic. Notable examples are FFT, hidden-line removal, and Kendall-tau distance algorithms.



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.
- Stable: Look into `merge()`, if equal keys, it takes them from the left subarray.
  - So is insertion sort, but not selection sort.

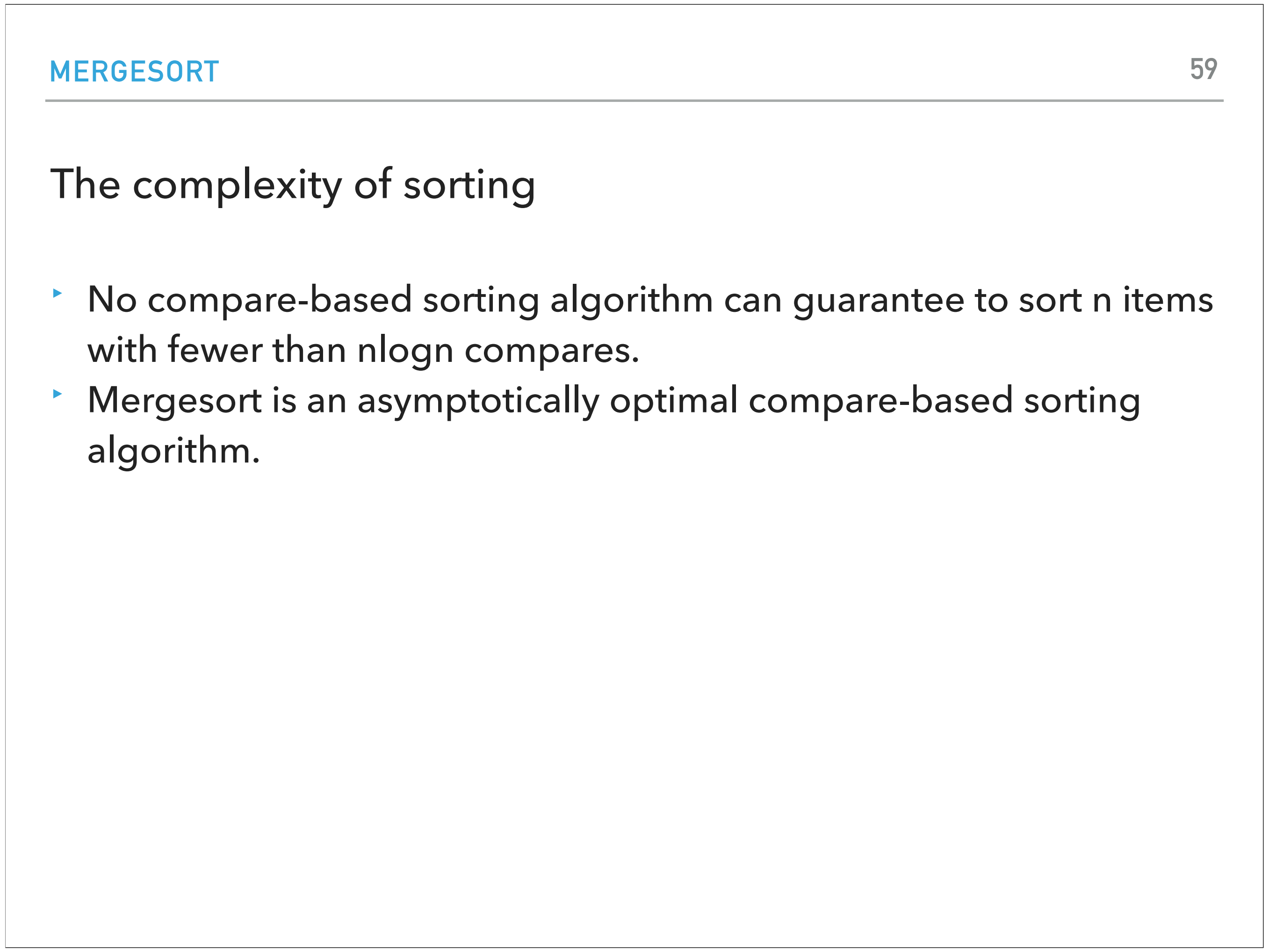
Mergesort is not in place since it requires auxiliary memory proportional to the length of the array to sort. Good news is that it is a stable algorithm since merge takes equal keys from the left subarray.



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

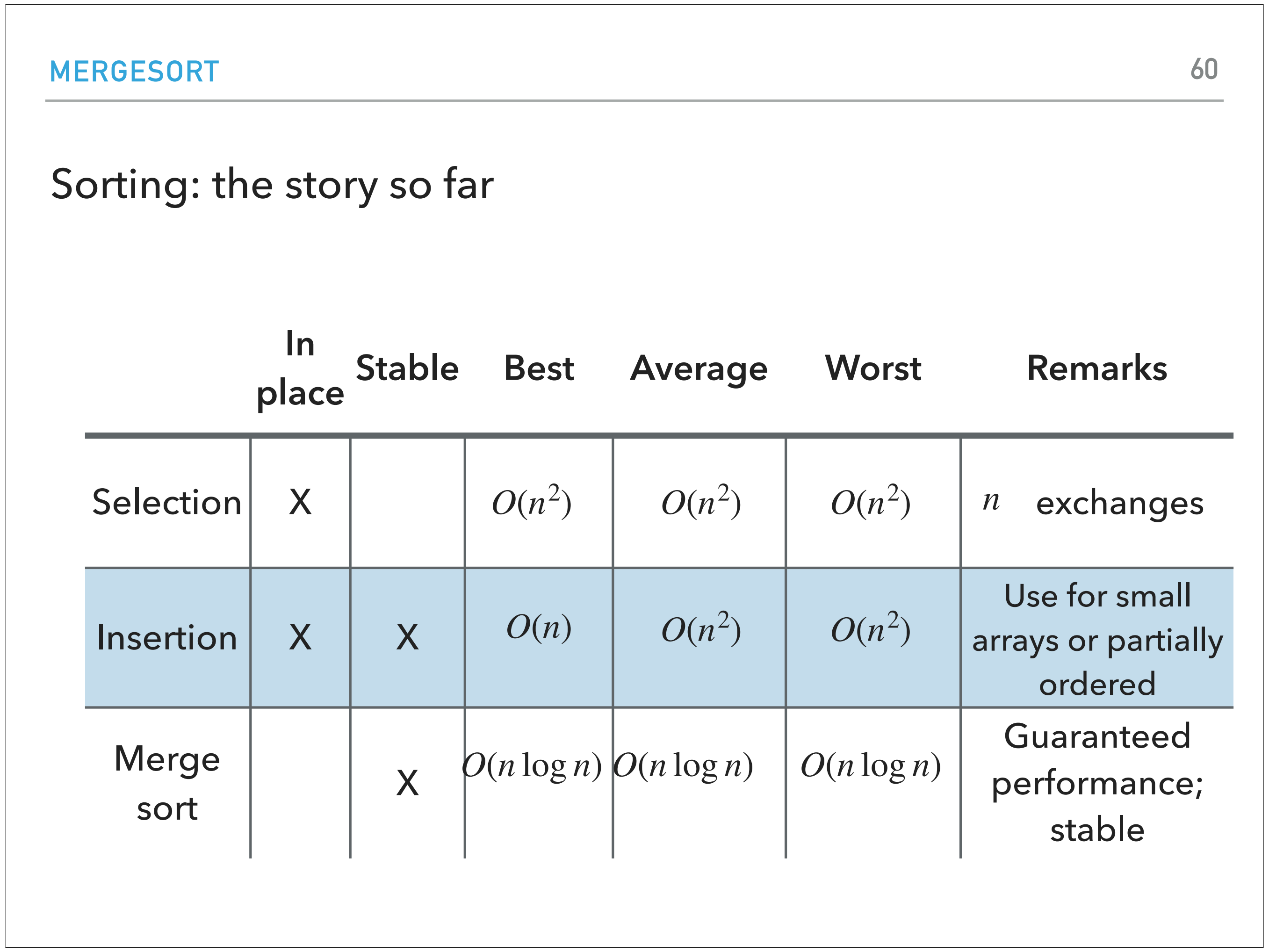
There are practical improvements we can follow. For example, we can use insertion sort for small subarrays. In fact, we can improve most recursive algorithms by handling small cases differently. Switching to insertion sort for small subarrays will improve the running time of a typical mergesort implementation by 10 to 15 percent.
We can also test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.
It is possible to eliminate the time (but not the space) taken to copy to the auxiliary array used for merging. To do so, we use two invocations of the mergesort method, one that takes its input from the given array and puts the sorted output in the auxiliary array; the other takes its input from the auxiliary array and puts the sorted output in the given array. With this approach, in a bit of mindbending recursive trickery, we can arrange the recursive calls such that the computation switches the roles of the input array and the auxiliary array at each level. For years, Java used this version to sort Collections of objects.



The complexity of sorting

- No compare-based sorting algorithm can guarantee to sort n items with fewer than nlogn compares.
- Mergesort is an asymptotically optimal compare-based sorting algorithm.

One important reason to know about merge sort is that we use it as the basis for providing a fundamental result in the field of computational complexity that helps us understand the intrinsic difficulty of sorting. For compare-based algorithms, there is no algorithm that takes fewer than nlogn compares. Which means that mergesort is an asymptotically optimal compare-based sorting algorithm: the number of compares used in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are nlogn.



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 ordered
Merge sort		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; stable

And here is the story so far with sorting. Remember that you need to consider worst, average, best case, as well as whether an algorithm is in place or stable. Merge sort does fantastic in providing guaranteed performance and it is stable but it does poorly in terms of memory performance.

Readings:

- Recommended Textbook:
  - Chapter 2.2 (pages 270-277)
- Recommended Textbook Website:
  - Mergesort: https://algs4.cs.princeton.edu/22mergesort/

Code

- Lecture 14 code

Practice Problem 1 - Recommended textbook 2.2.2

- Give a trace in the style of this lecture, showing how the array [E, A, S, Y, Q, U, E, S, T, I, O, N] would be sorted by mergesort.

Practice Problem 2 - Recommended textbook 2.2.5

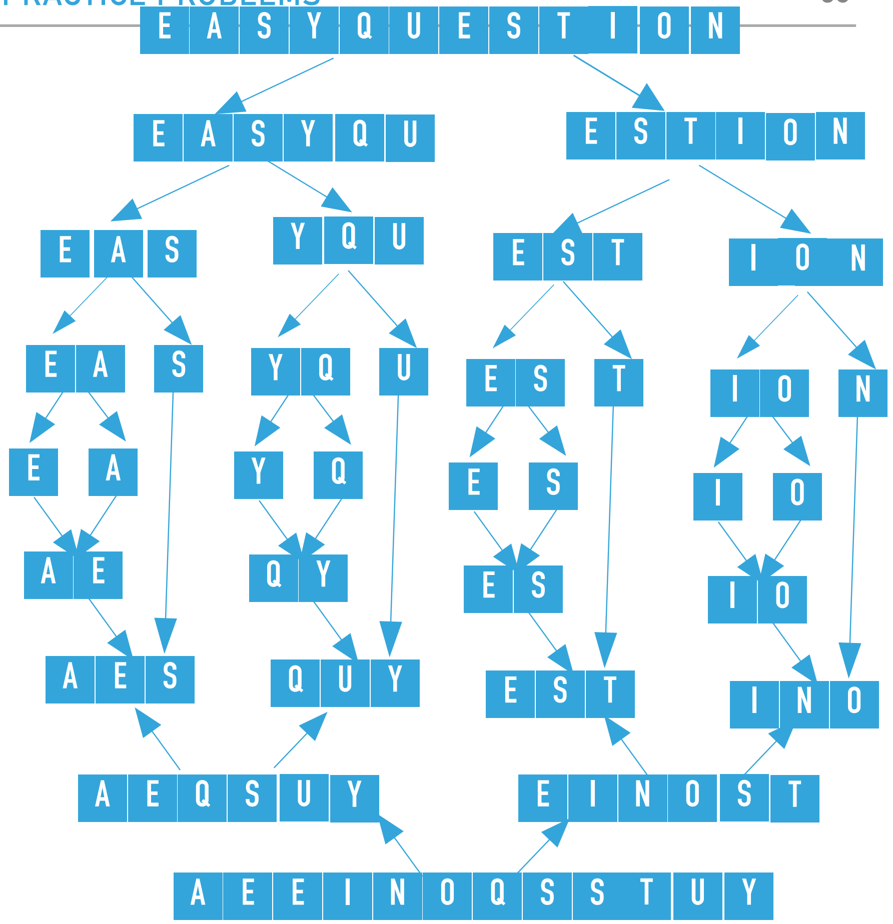
- Give the sequence of subarray lengths in the merges performed by merge sort for n=39.

Practice Problem 3 - Recommended textbook 2.2.6

- Write a program to compute the exact value of the number of array accesses used by merge sort. Use your program to plot the values for n from 1 to 512 and compare the exact values with the upper bound $6n \log n$.

ANSWER 1

- Give a trace in the style of this lecture, showing how the array [E, A, S, Y, Q, U, E, S, T, I, O, N] would be sorted by mergesort.



E A S Y Q U E S T I O N

E A S Y Q U E S T I O N

E A S Y Q U E S T I O N

E A S Y Q U E S T I O N

E A Y Q E S I O

A E Q Y E S I O

A E S Q U Y E S T I N O

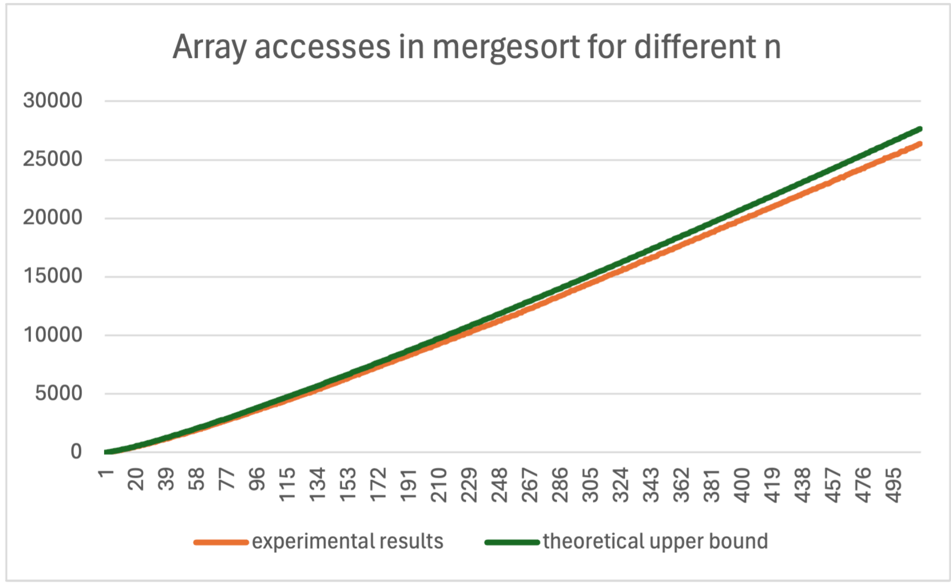
A E Q S U Y E I N O S T

A E E I N O Q S S T U Y

ANSWER 2

- Give the sequence of subarray lengths in the merges performed by merge sort for n=39.
- 39 will be split in 20 and 19. 20 will be split in 10 and 10. 10 will be split in 5 and 5. 5 will be split in 3 and 2. 3 will be split in 2 and 1. Putting this all together it will result to:
- 2, 3, 2, 5, 2, 3, 2, 5, 10, 2, 3, 2, 5, 2, 3, 2, 5, 10, 20, 2, 3, 2, 5, 2, 3, 2, 5, 10, 2, 3, 2, 5, 2, 2, 4, 9, 19, 39

ANSWER 3



On the course Github repo, you can find a Java program that solves the problem and a spreadsheet with a chart that show the experimental results vs the theoretical upper bound.