CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

14: Mergesort

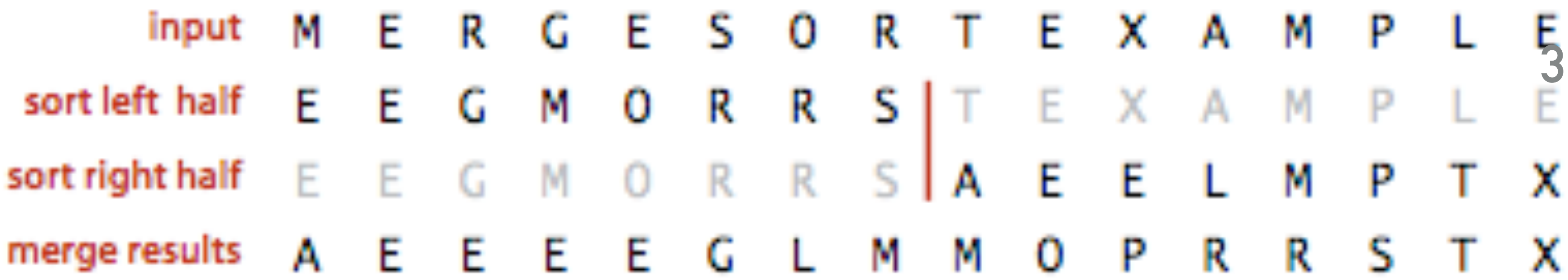
Alexandra Papoutsaki
she/her/hers

Lecture 13: Mergesort

- Mergesort

Some slides adopted from Algorithms 4th Edition or COS226

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 - the quintessential example of divide-and-conquer

```java
@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);
}
```

Merging two already sorted halves into one sorted array

```java
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++];
        }
    }
}
```

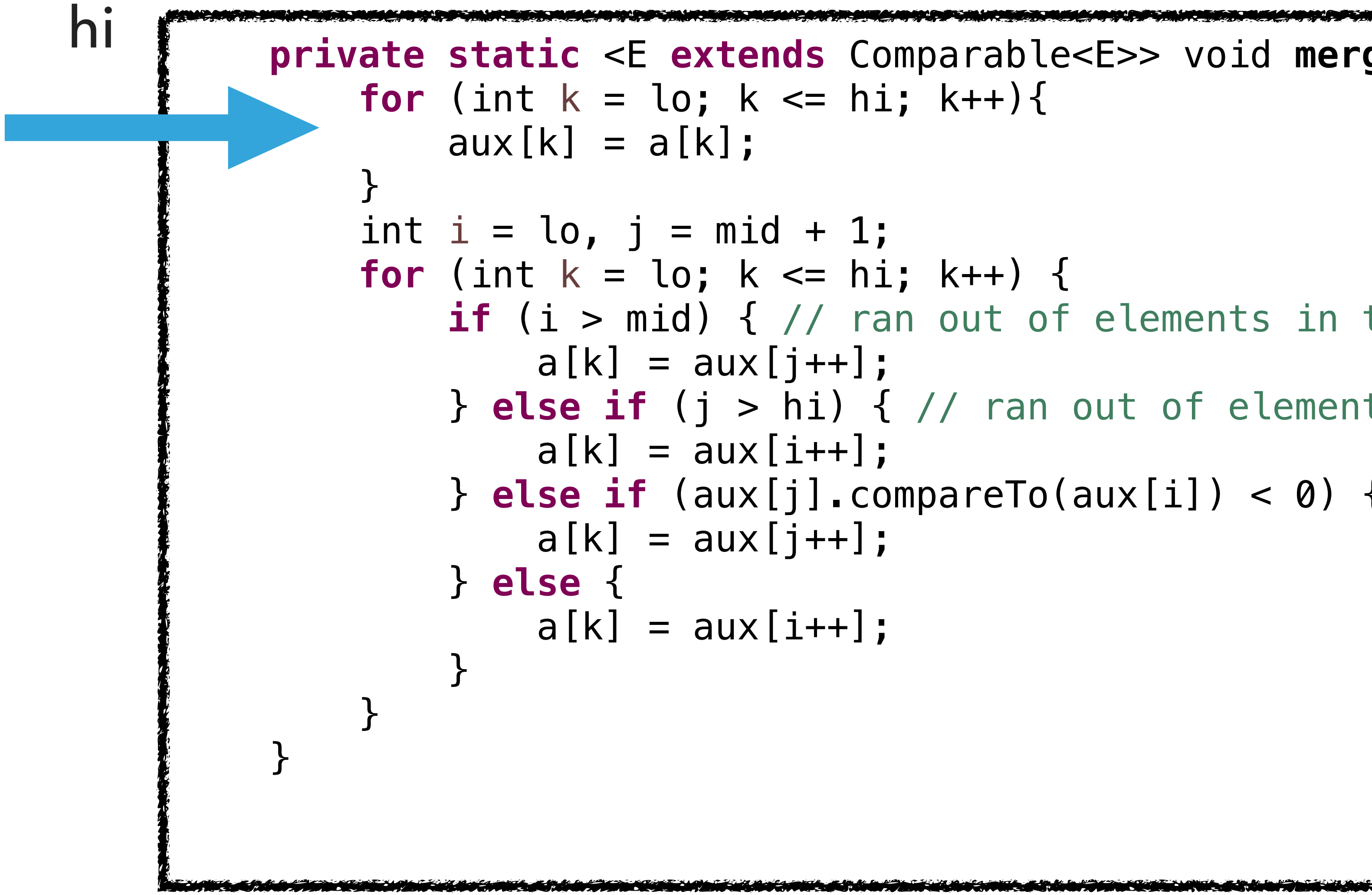
Merging Example - copying to auxiliary array

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++];
        }
    }
}
```

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									

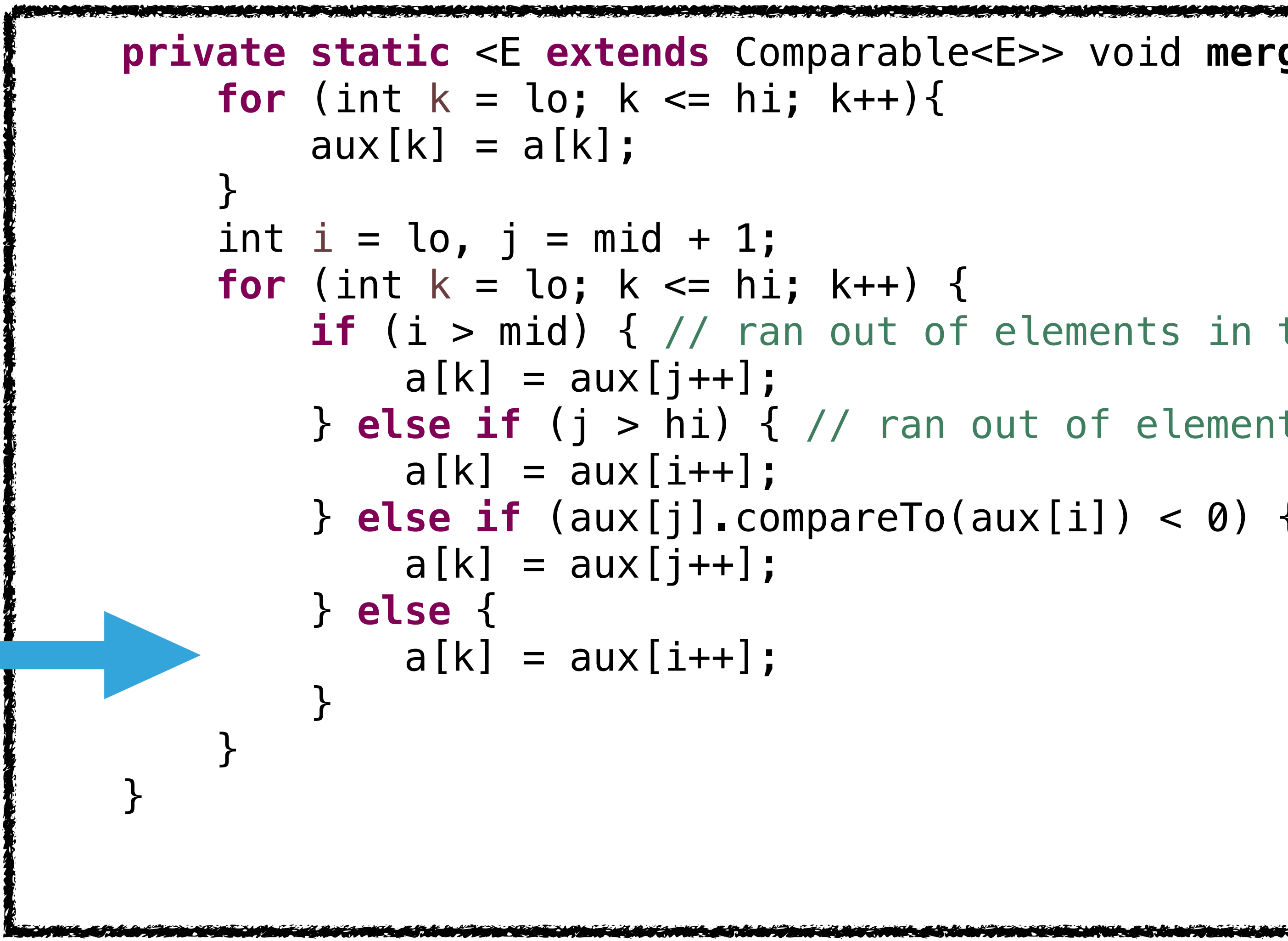
Array a

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++];
        }
    }
}
```

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								

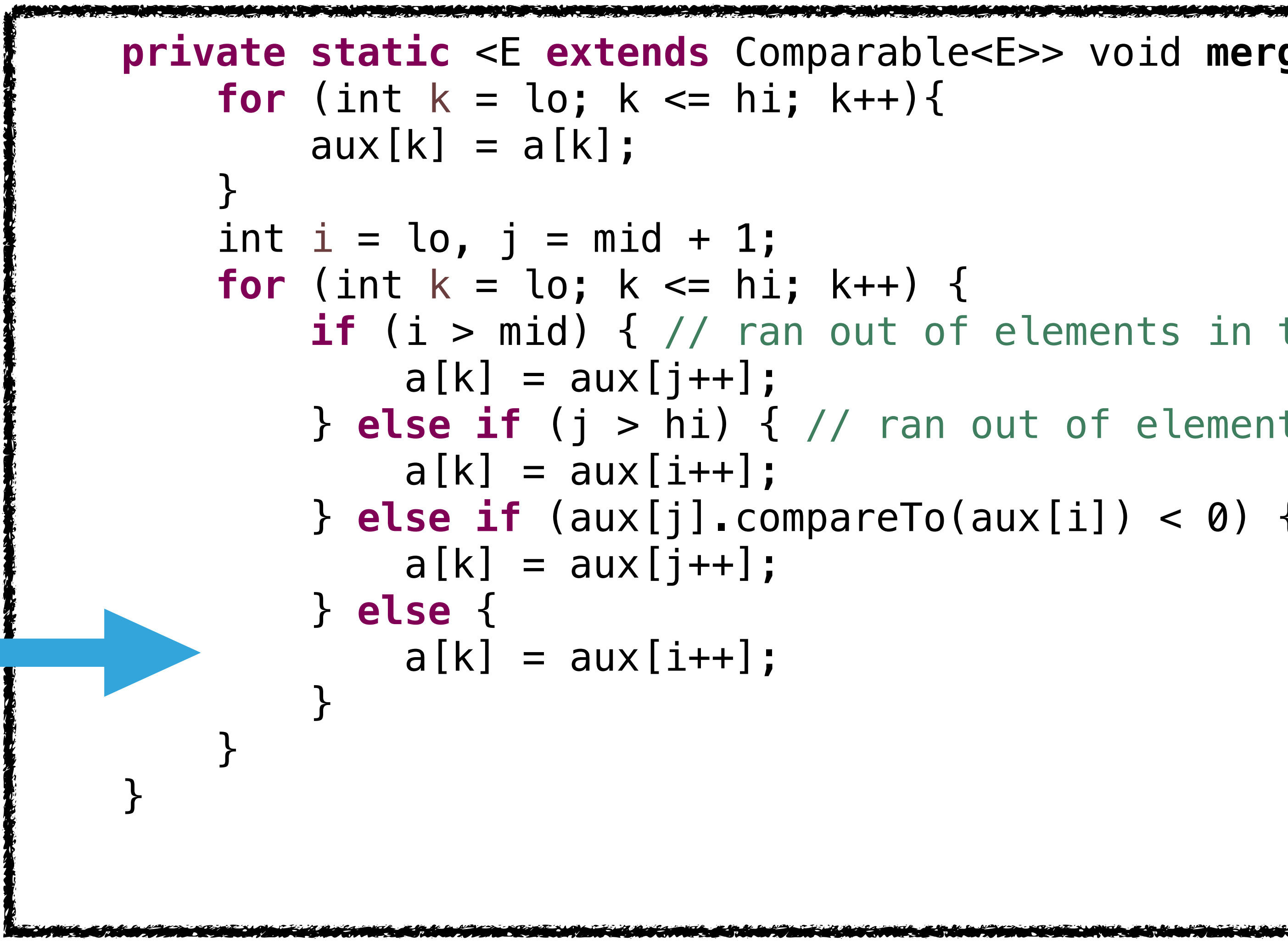
Array a

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++];
        }
    }
}
```

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							

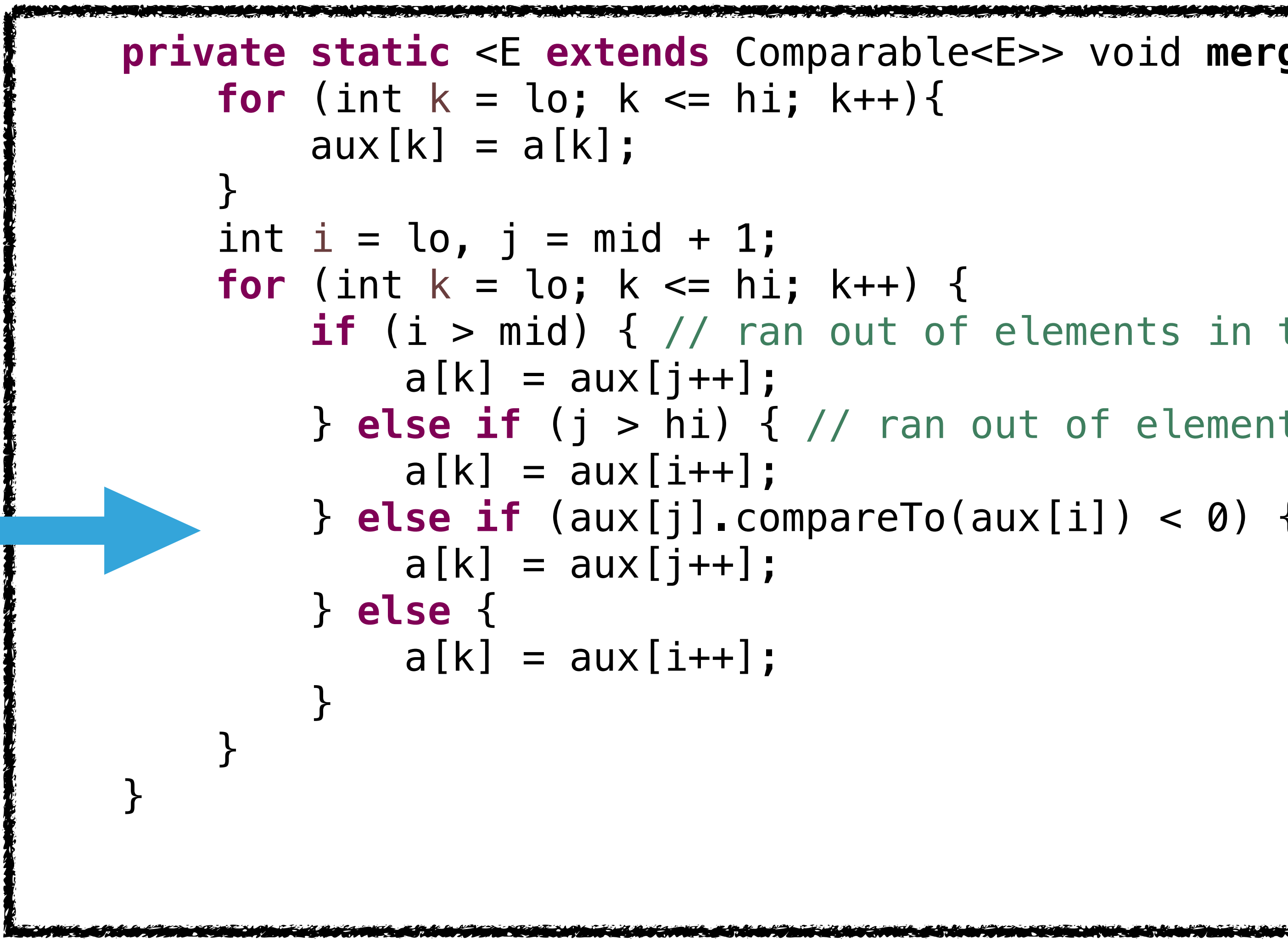
Array a

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++];
        }
    }
}
```

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						

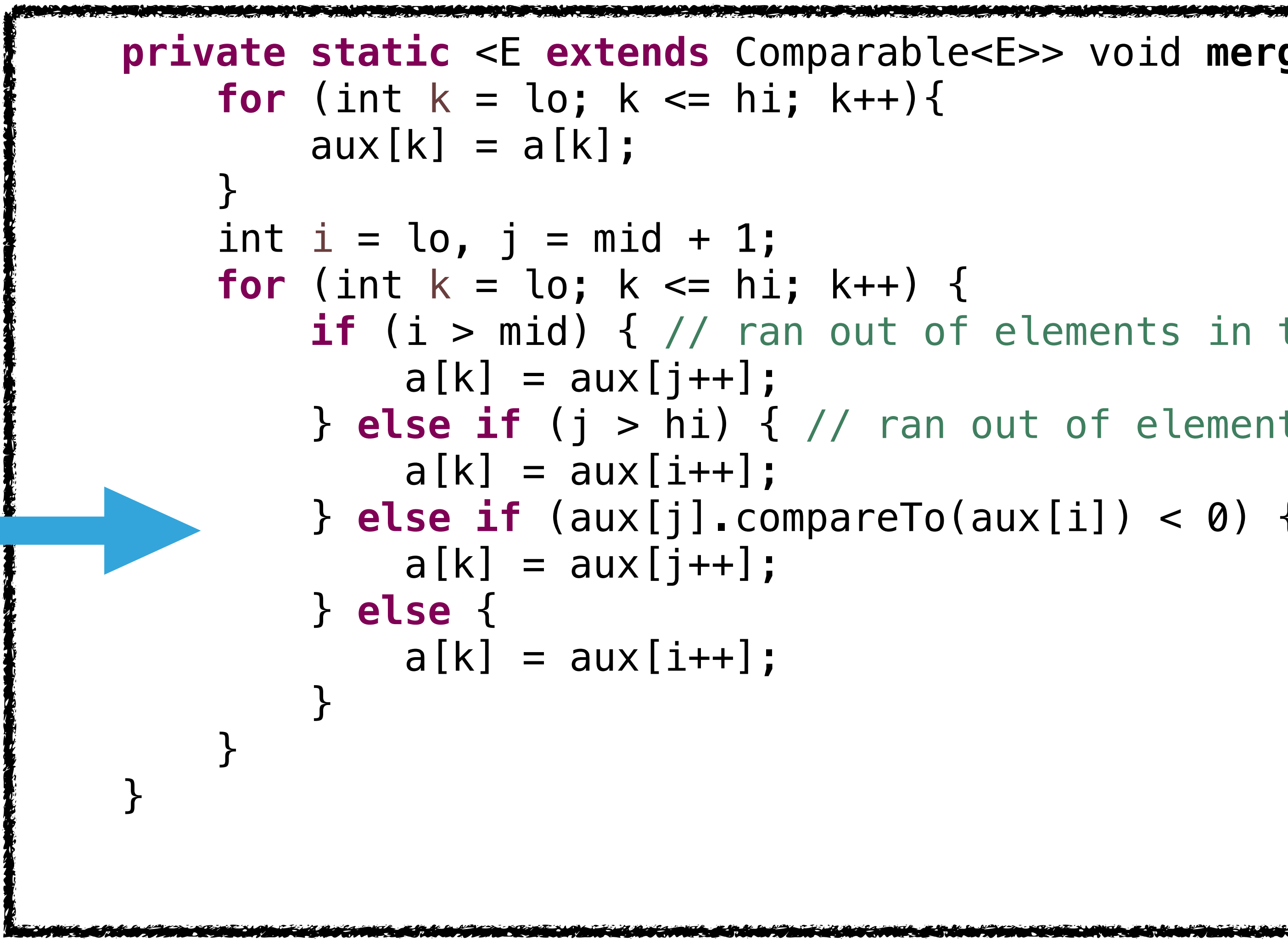
Array a

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++];
        }
    }
}
```

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					

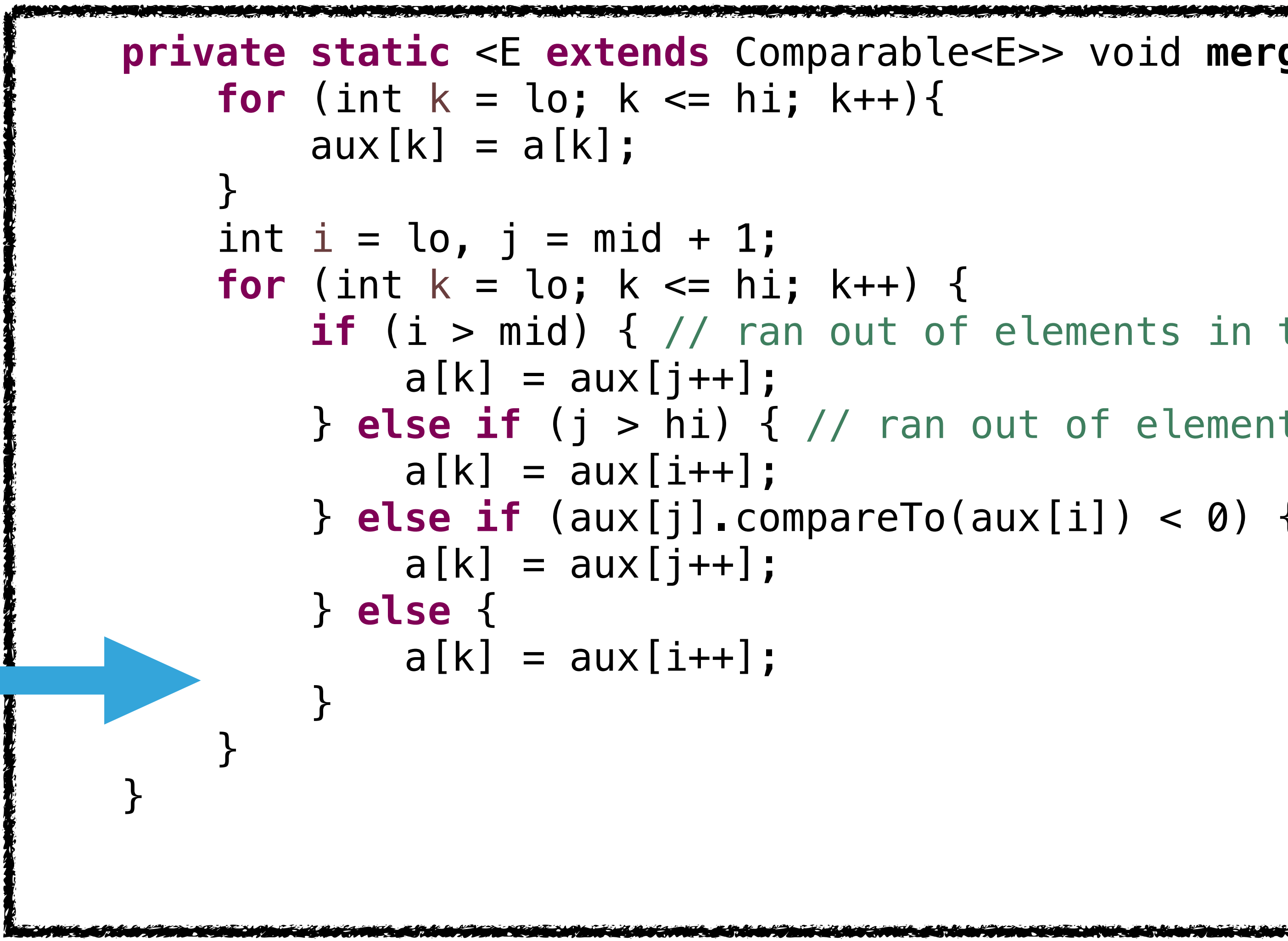
Array a

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++];
        }
    }
}
```

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				

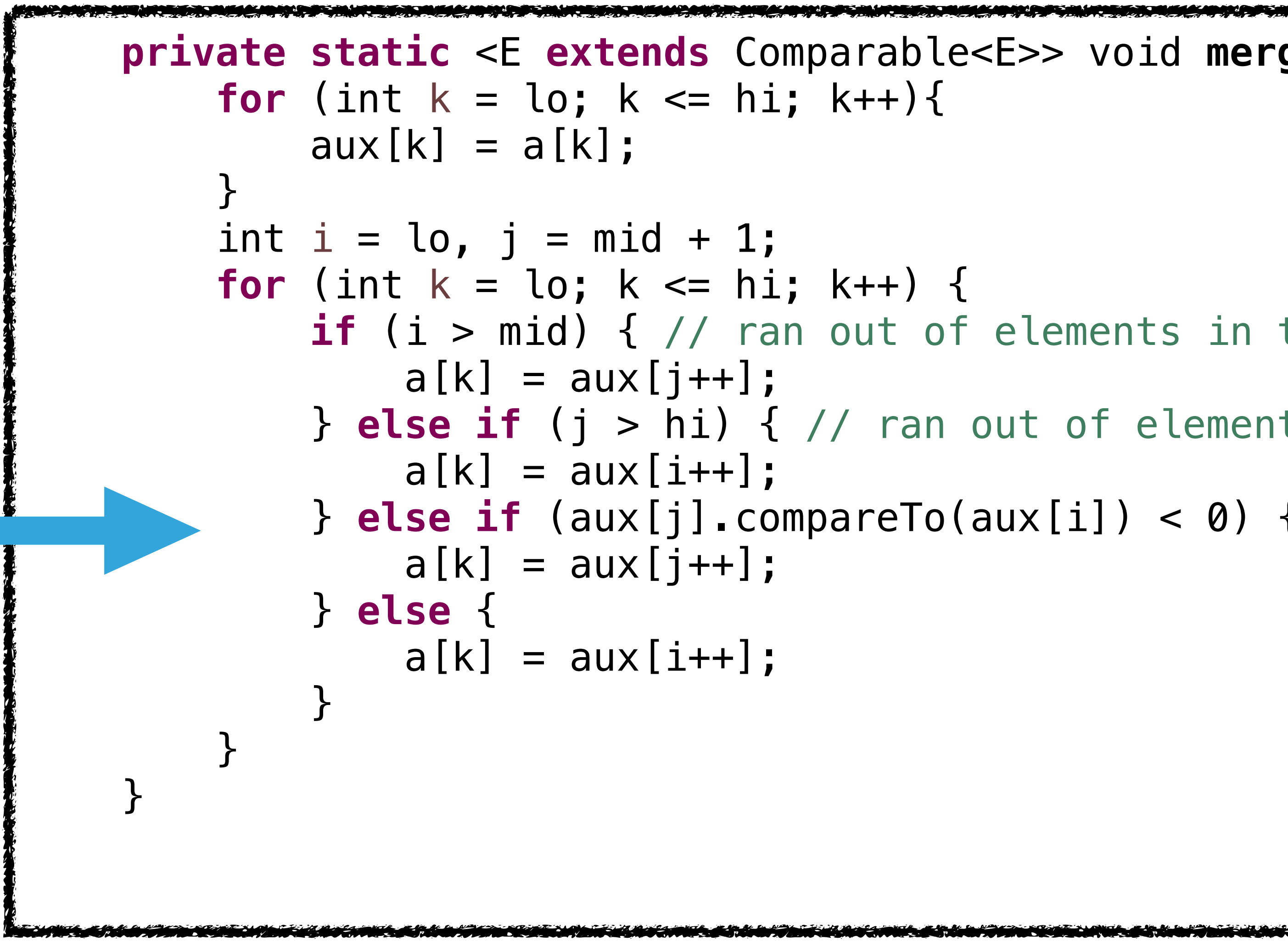
Array a

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++];
        }
    }
}
```

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			

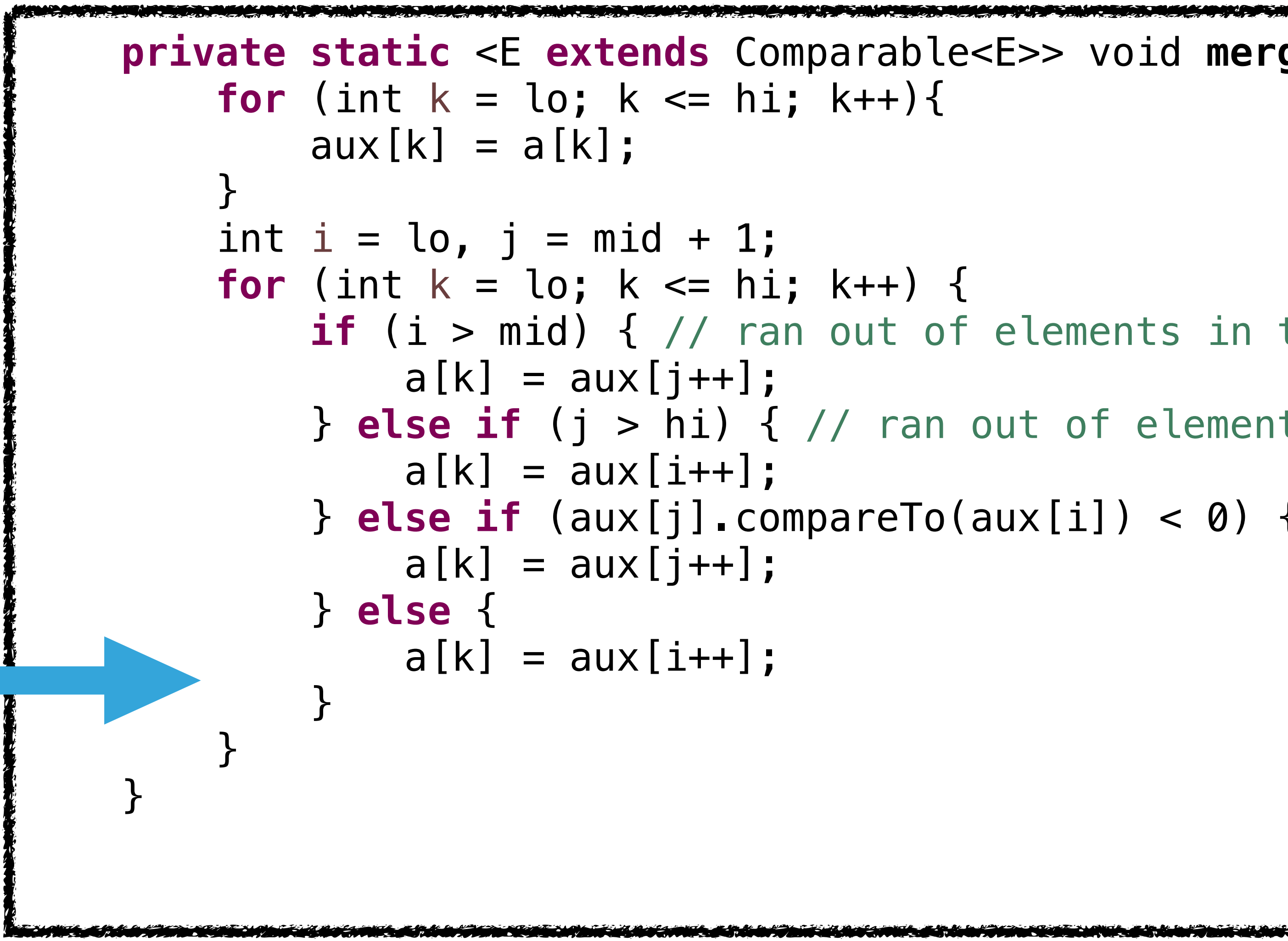
Array a

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++];
        }
    }
}
```

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		

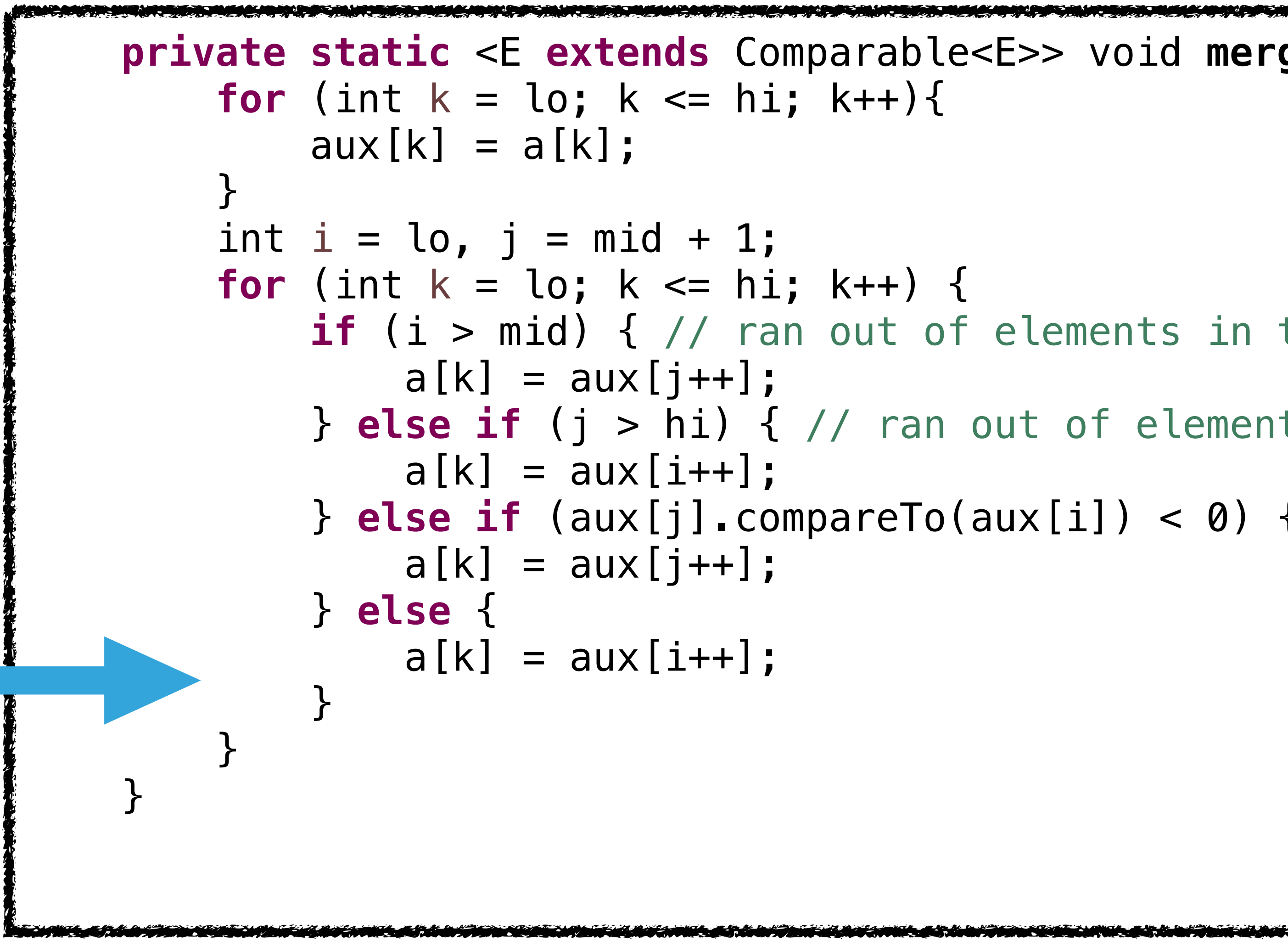
Array a

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]

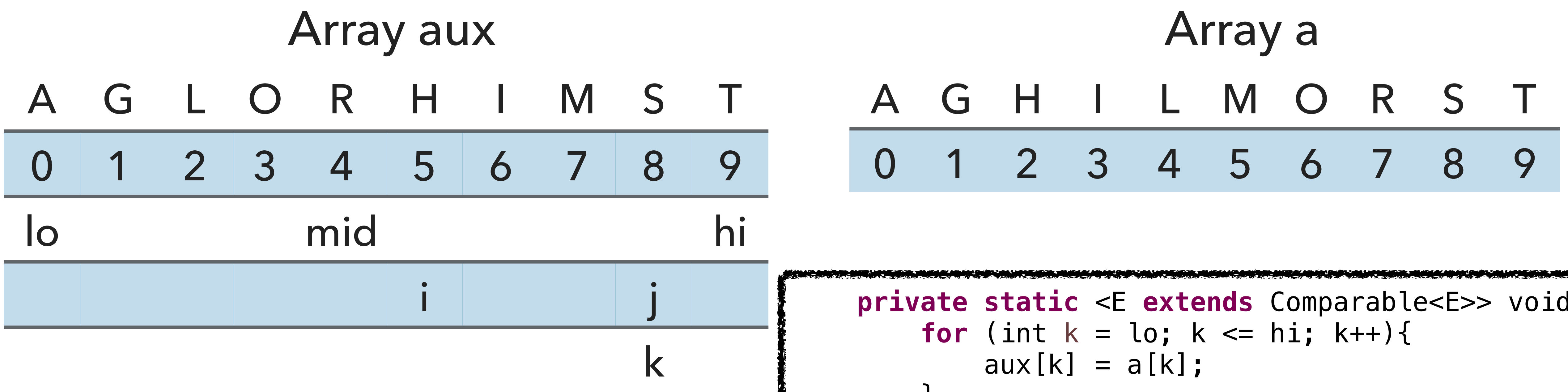
a[7]=aux[4]

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++];
        }
    }
}
```

Merging Example - k=8



Array aux

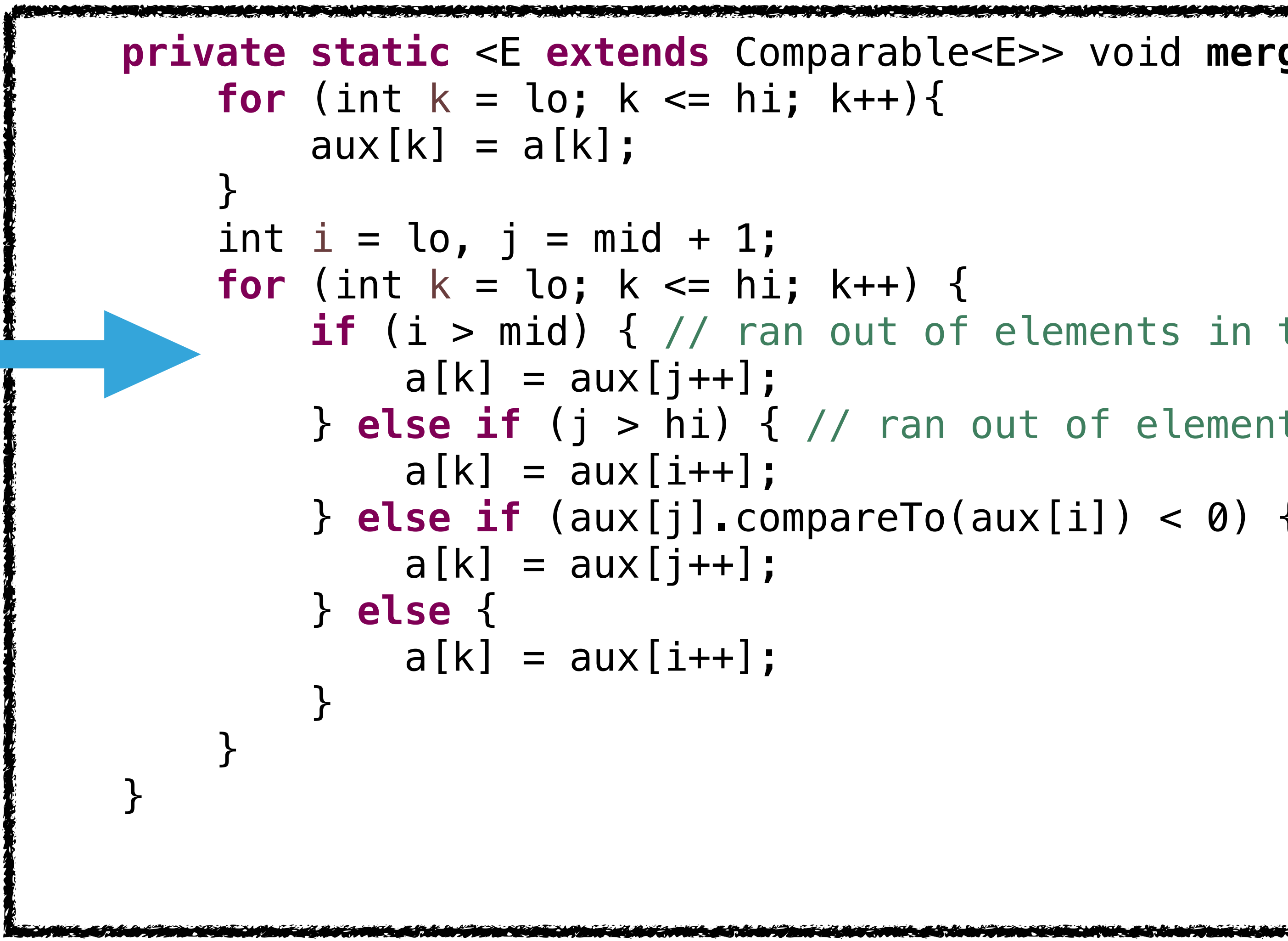
A	G	L	O	R	H	I	M	S	T
0	1	2	3	4	5	6	7	8	9
lo				mid					hi
					i			j	
								k	

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++];
        }
    }
}
```

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

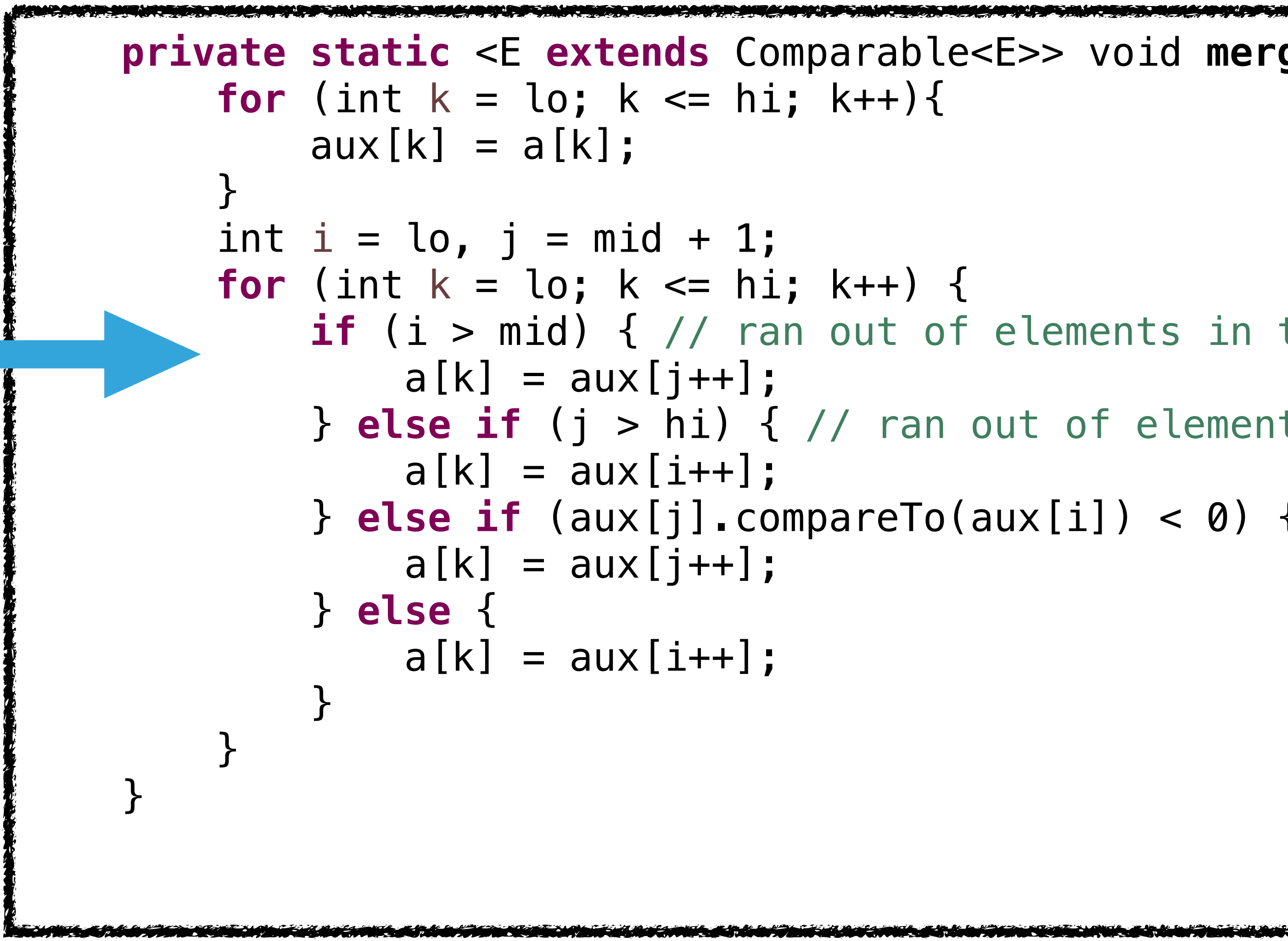
Array a

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++];
        }
    }
}
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

2.2 MERGING DEMO

Algorithms

FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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

Practice time

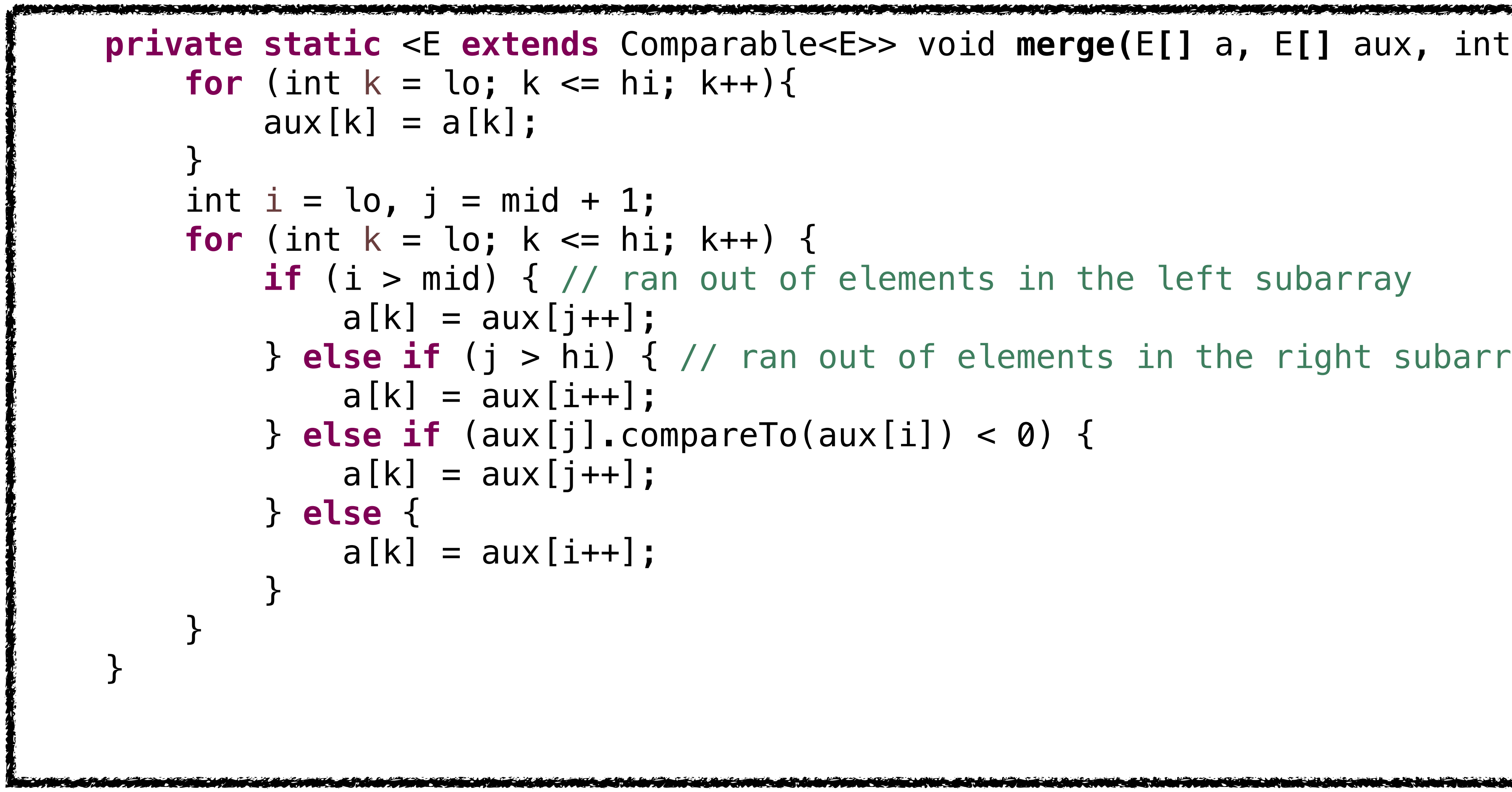
How many calls does `merge()` make to `compareTo()` in order to merge two already sorted subarrays, each of length $n/2$ into a sorted array of length n?

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++];
        }
    }
}
```

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)

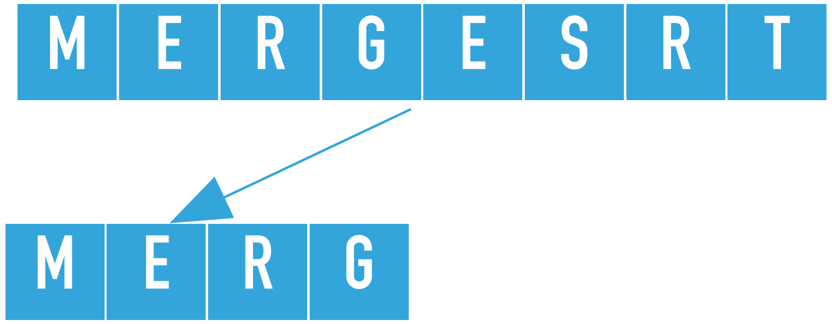
Mergesort - the quintessential example of divide-and-conquer

```java
@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);
}
```

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`.

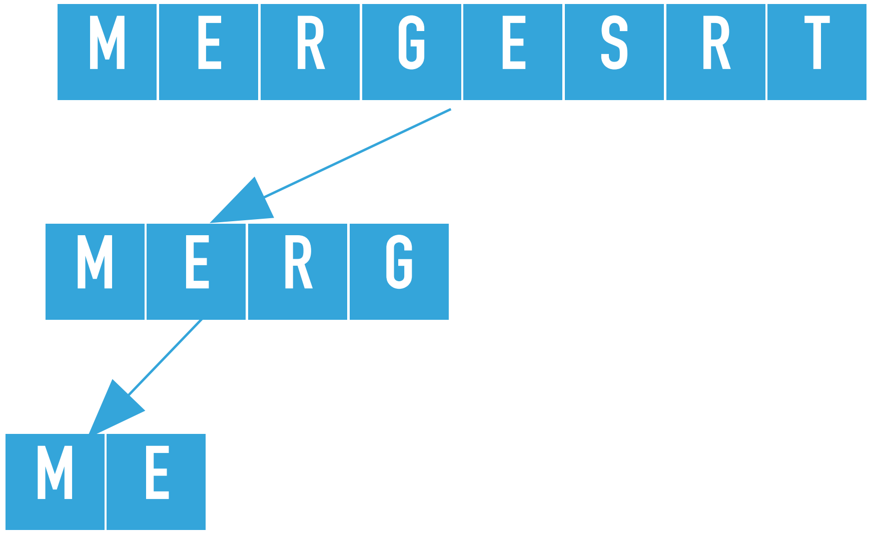


M	E	R	G	E	S	R	T

M	E	R	G

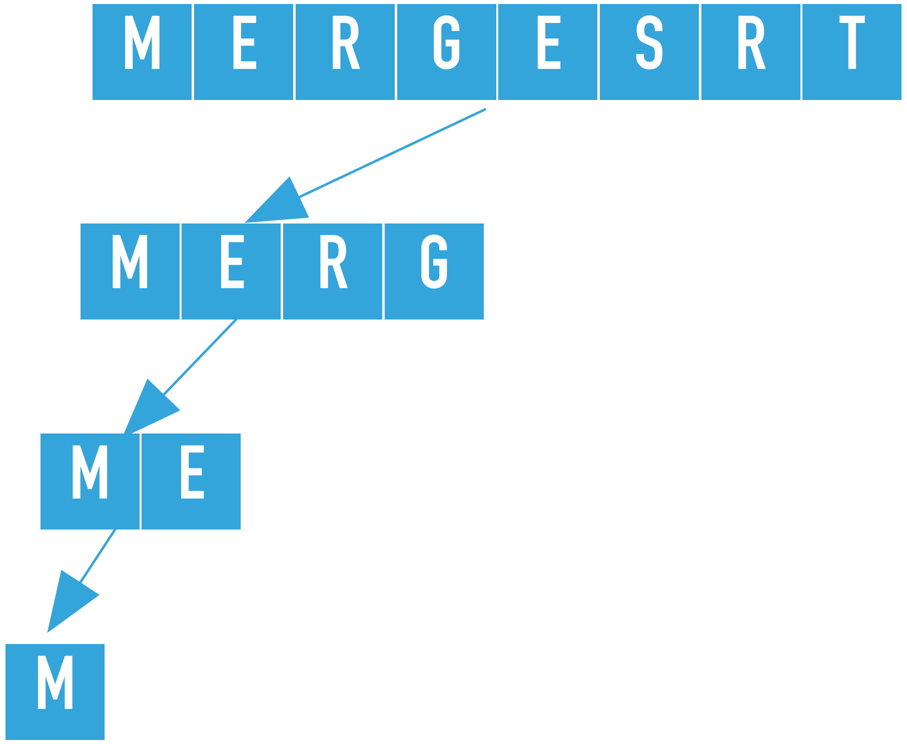
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



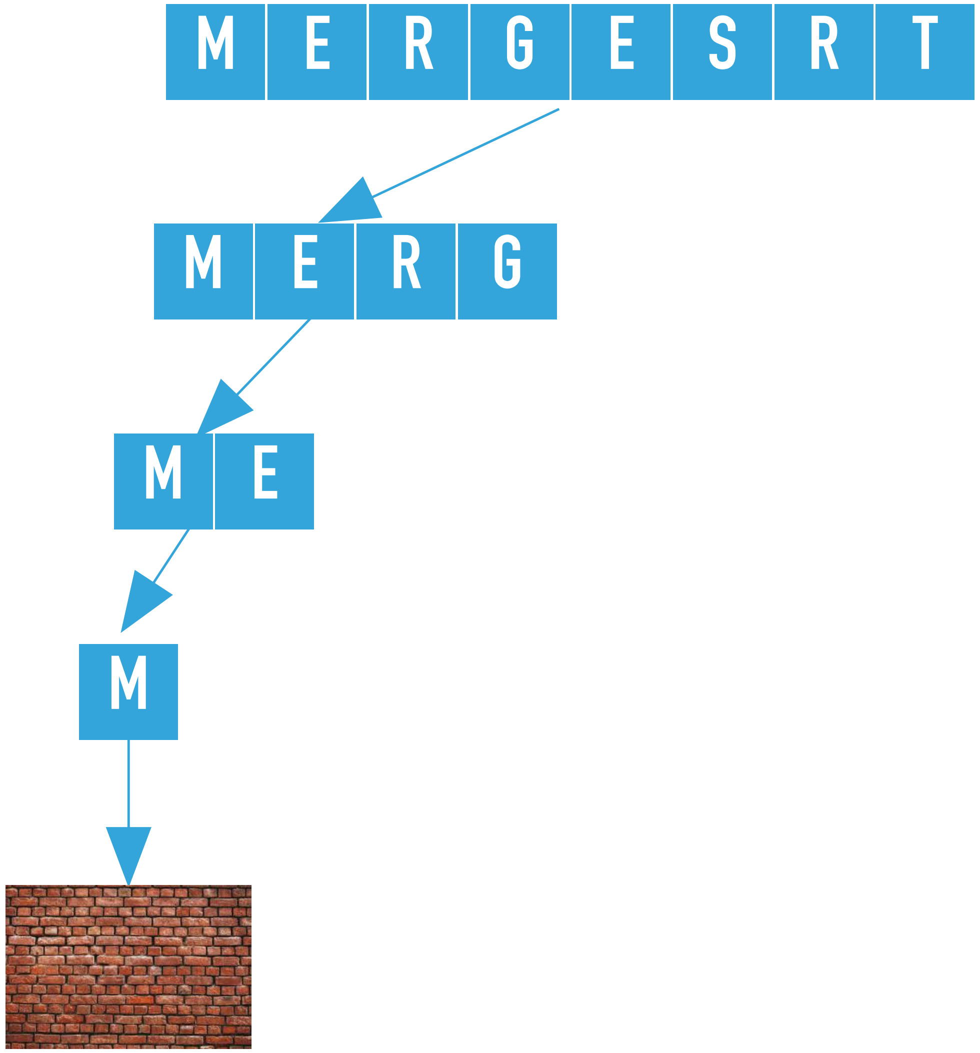
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



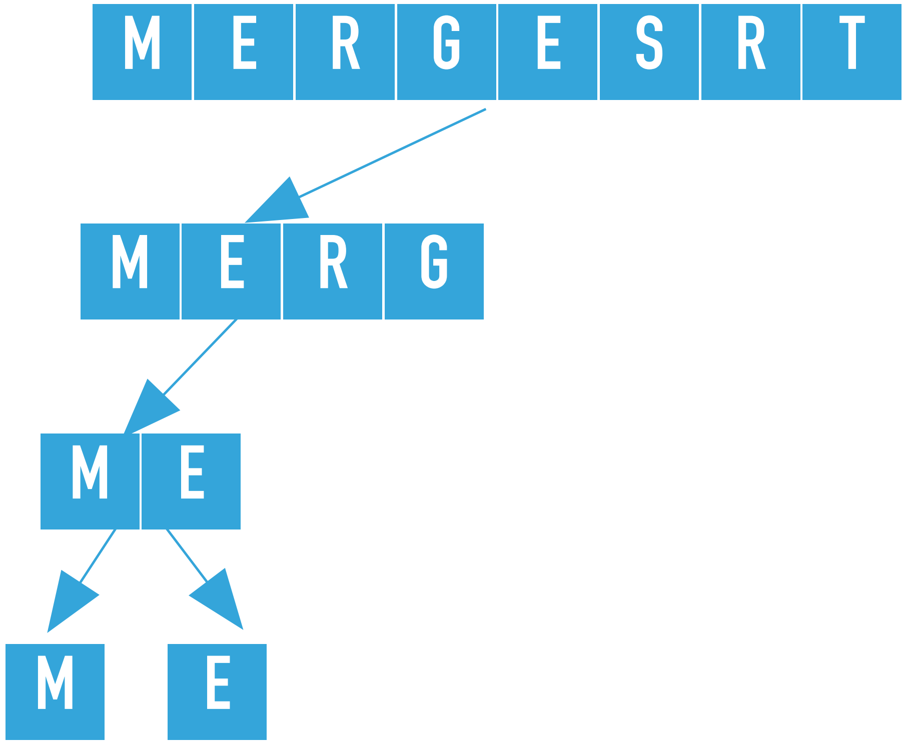
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



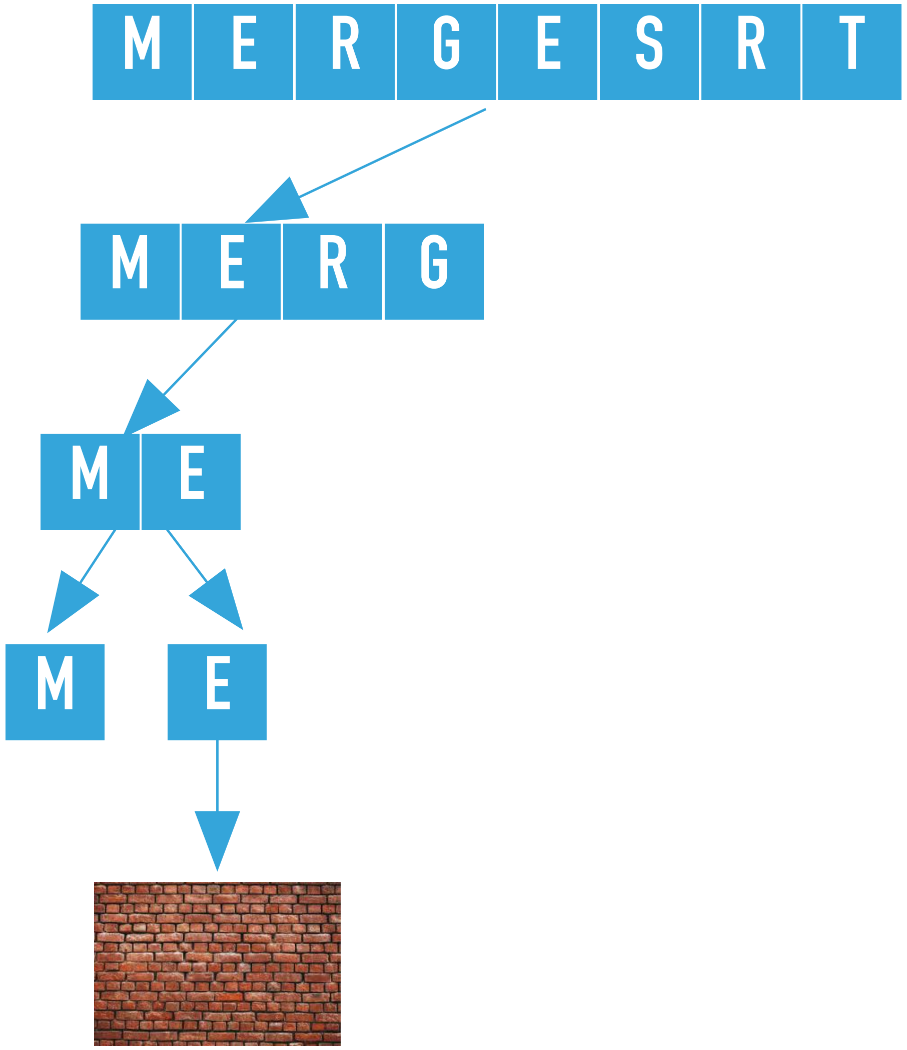
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



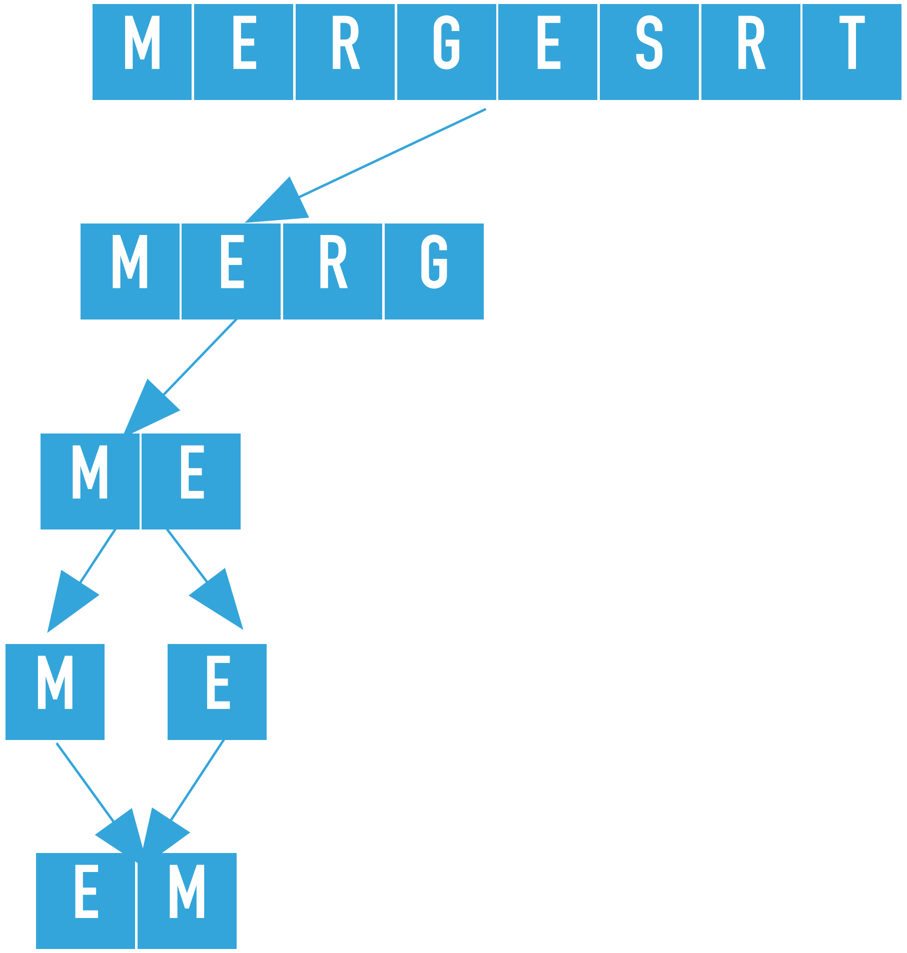
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



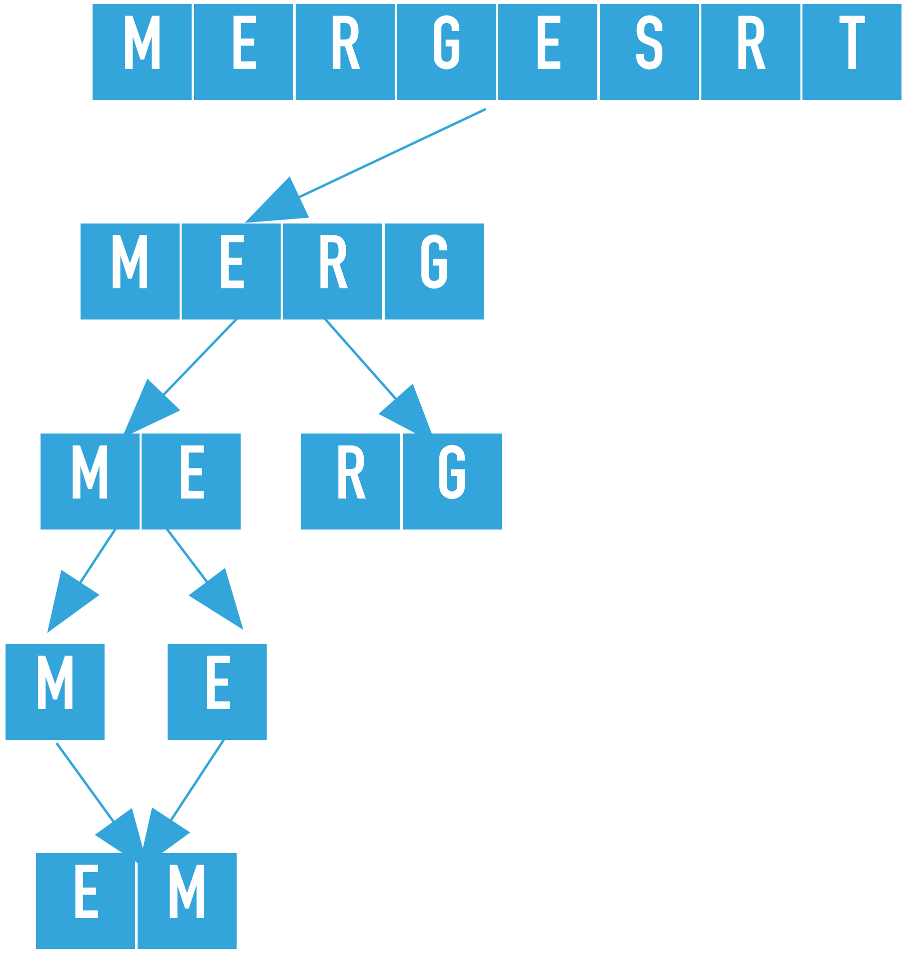
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



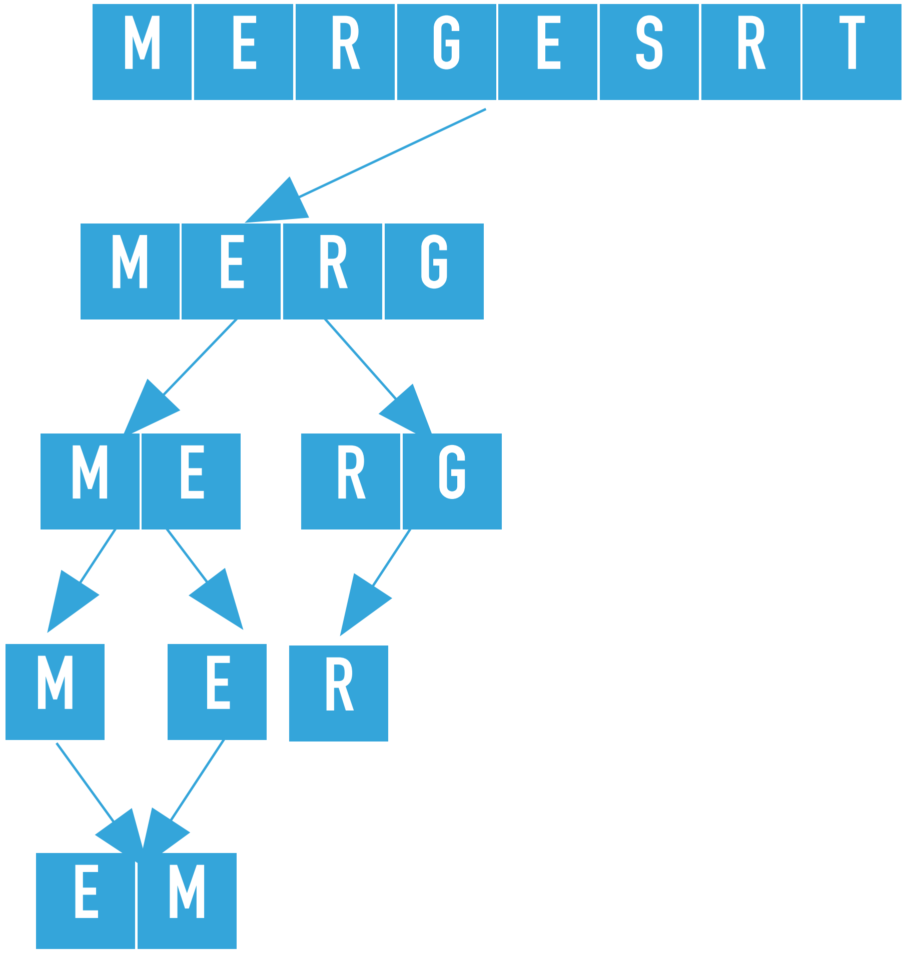
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



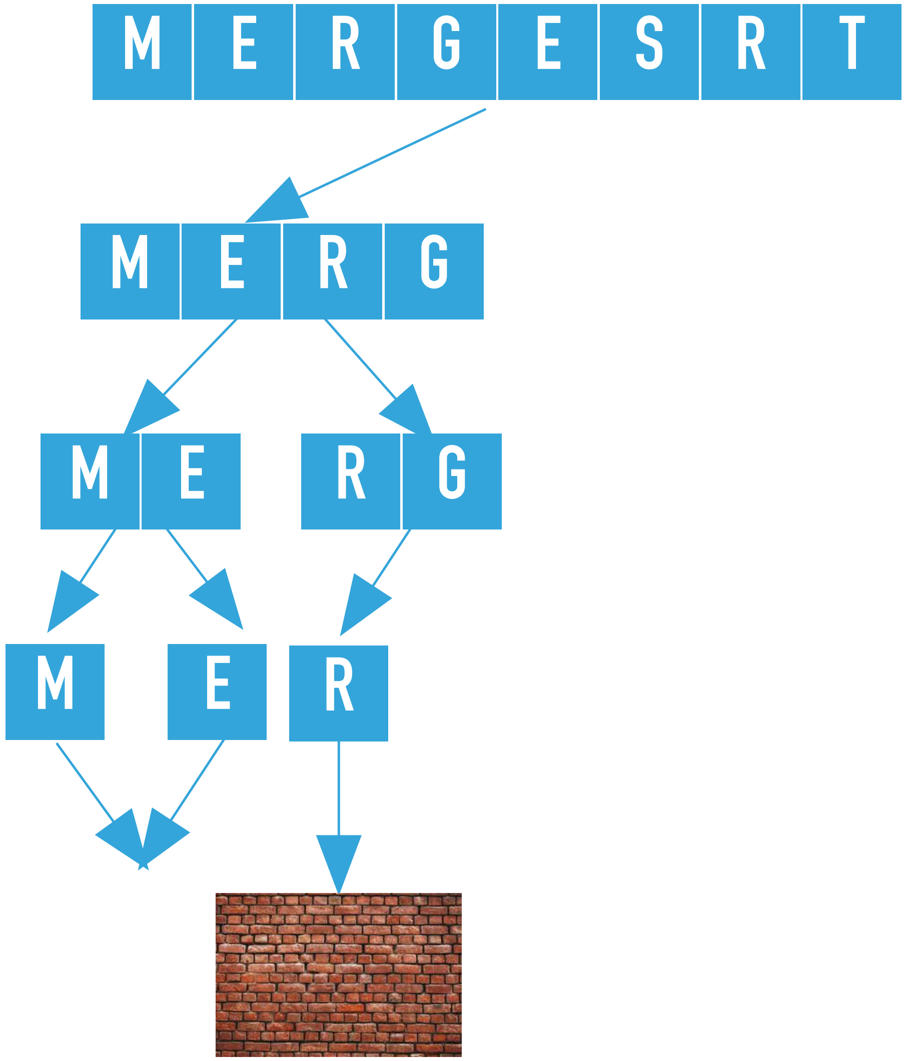
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



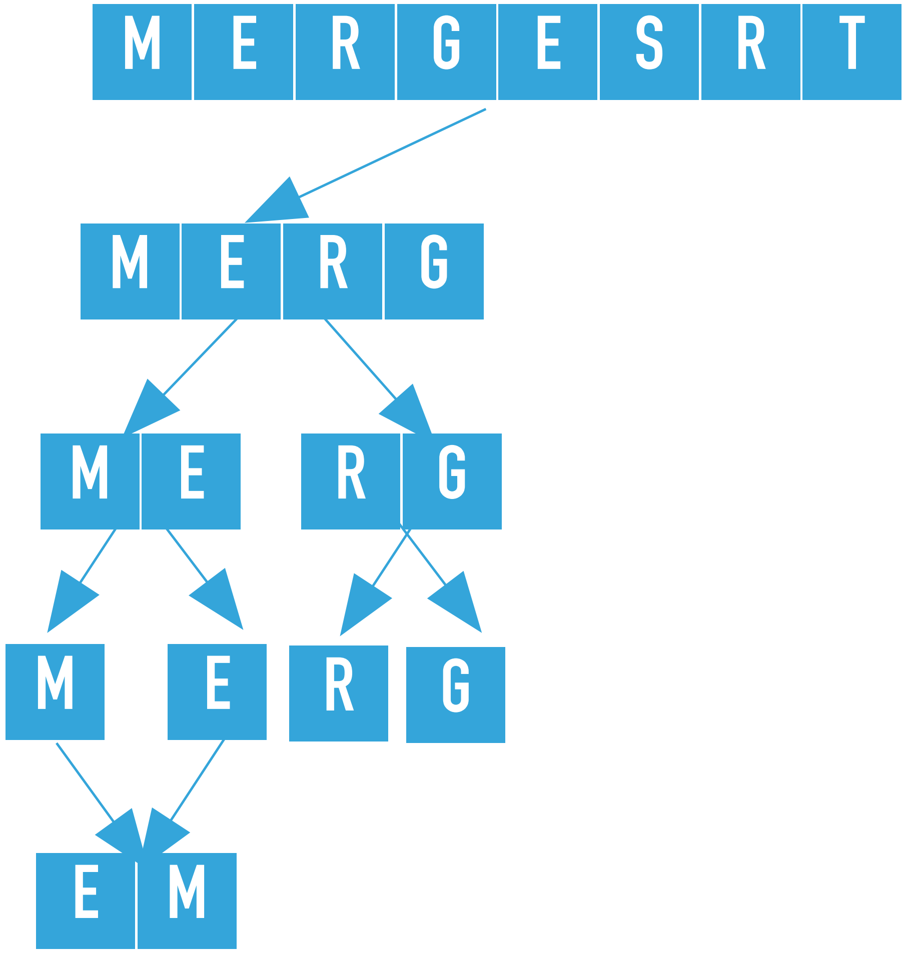
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



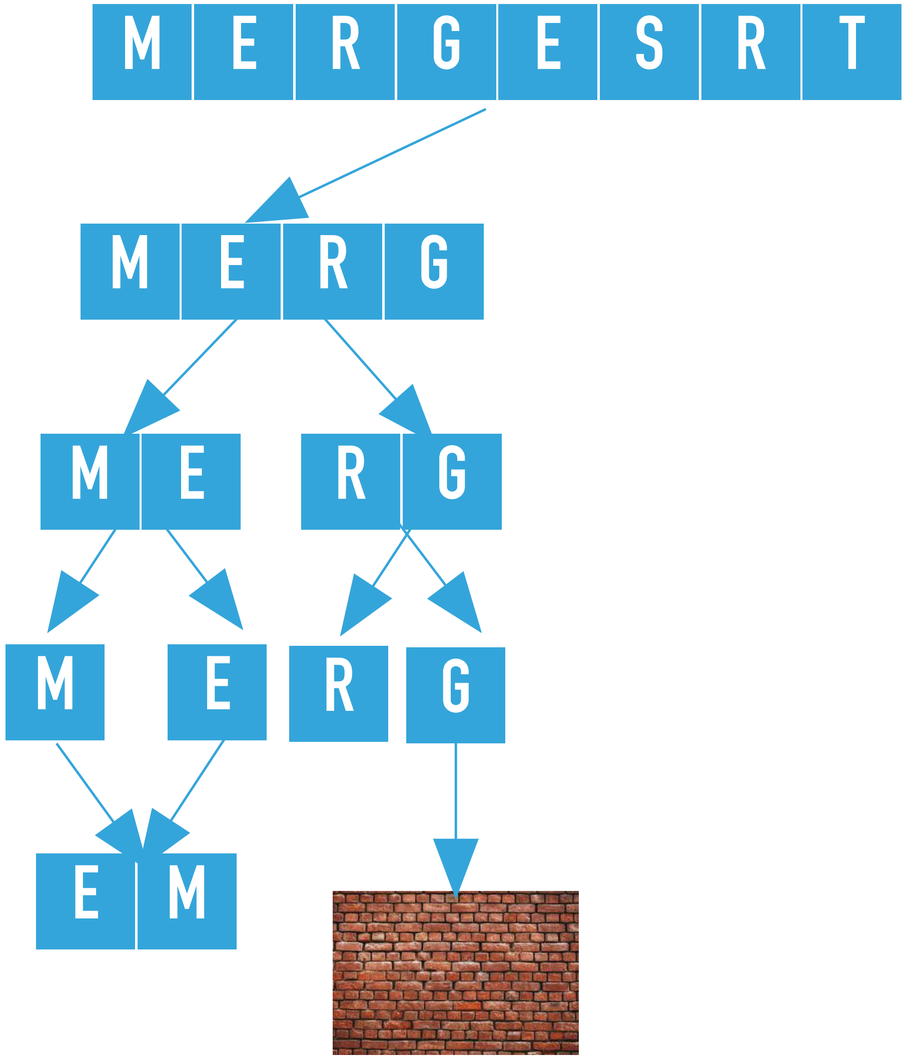
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



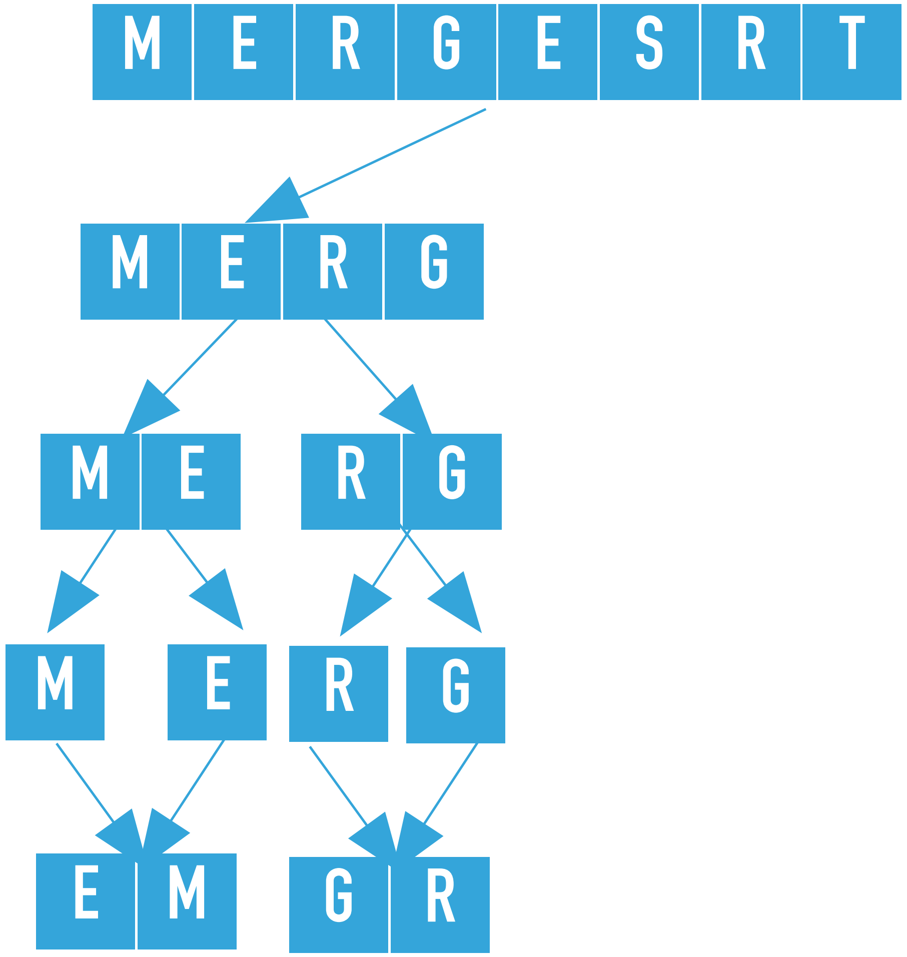
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



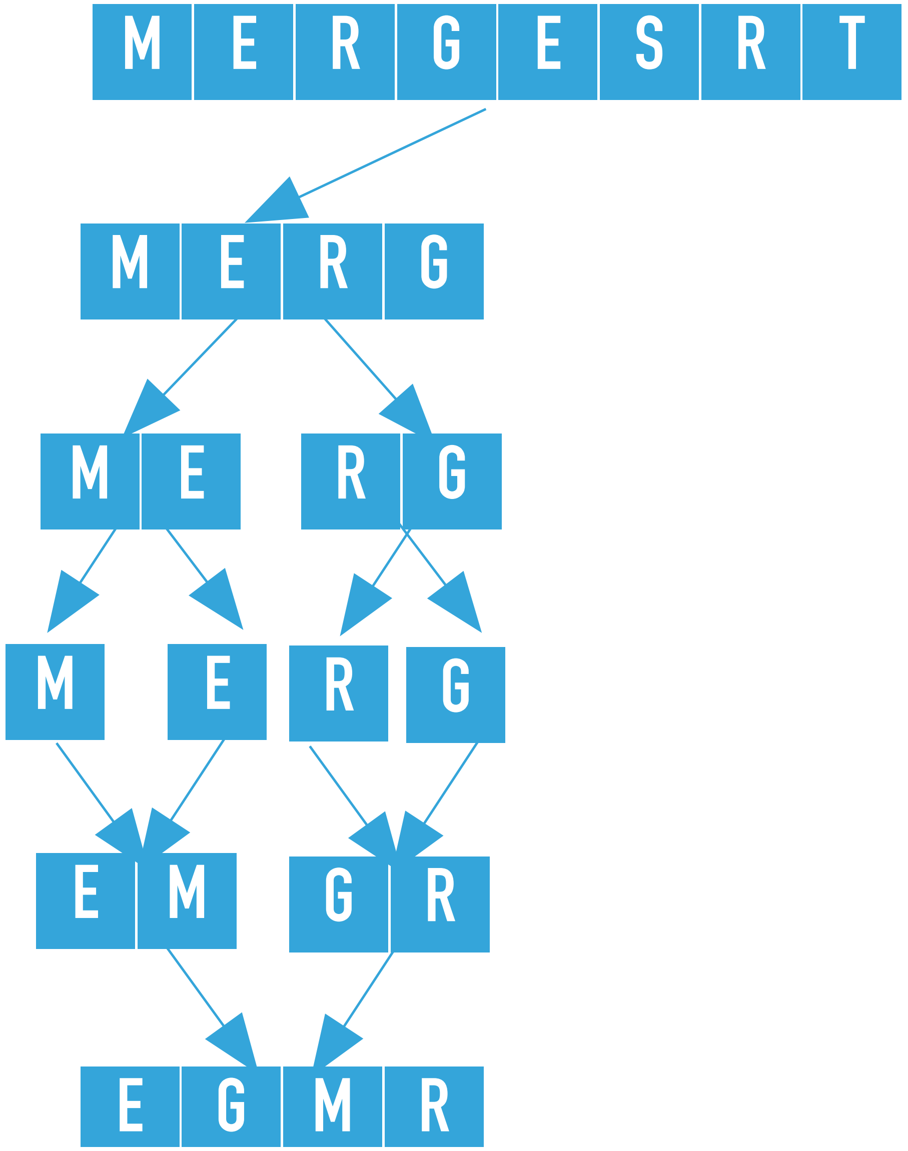
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



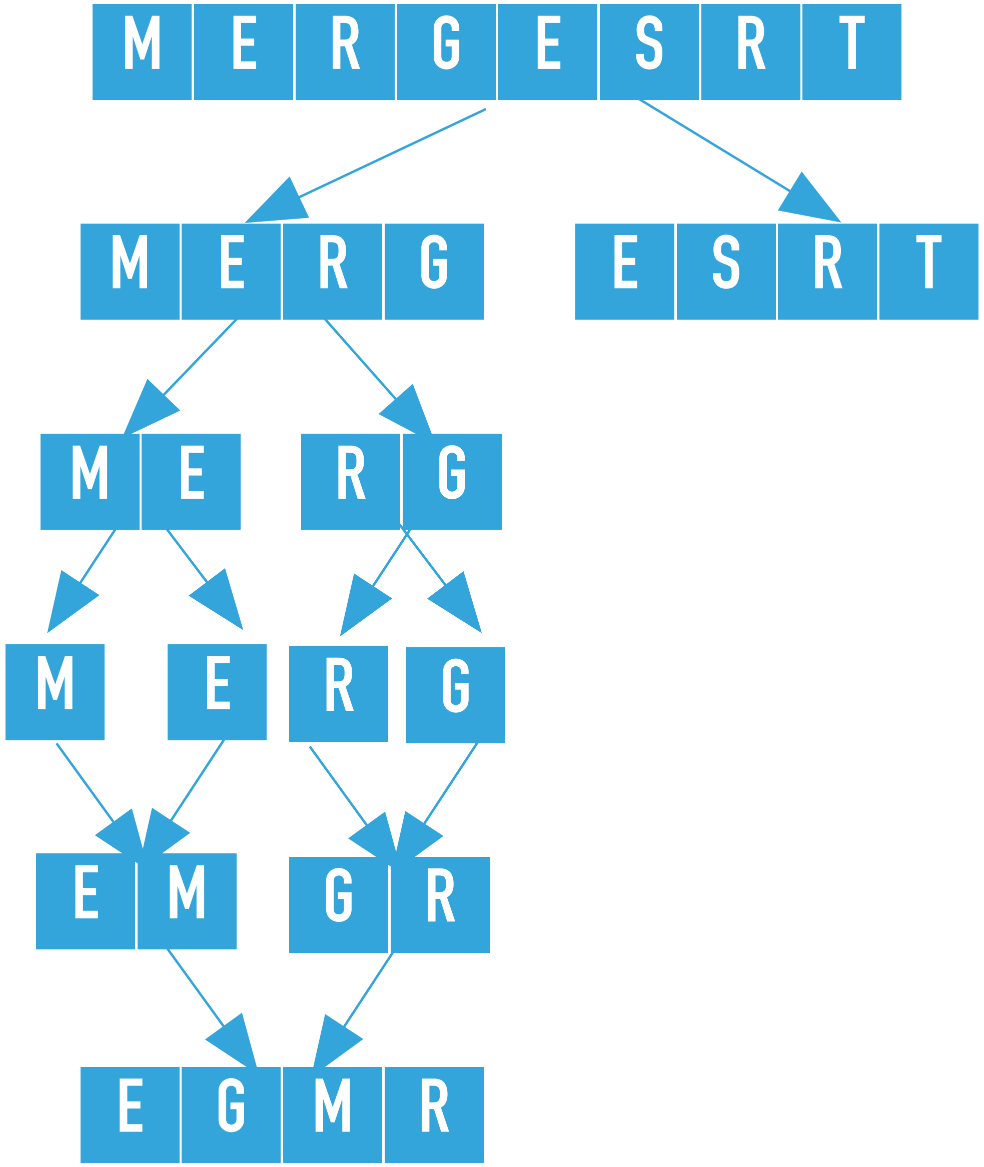
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



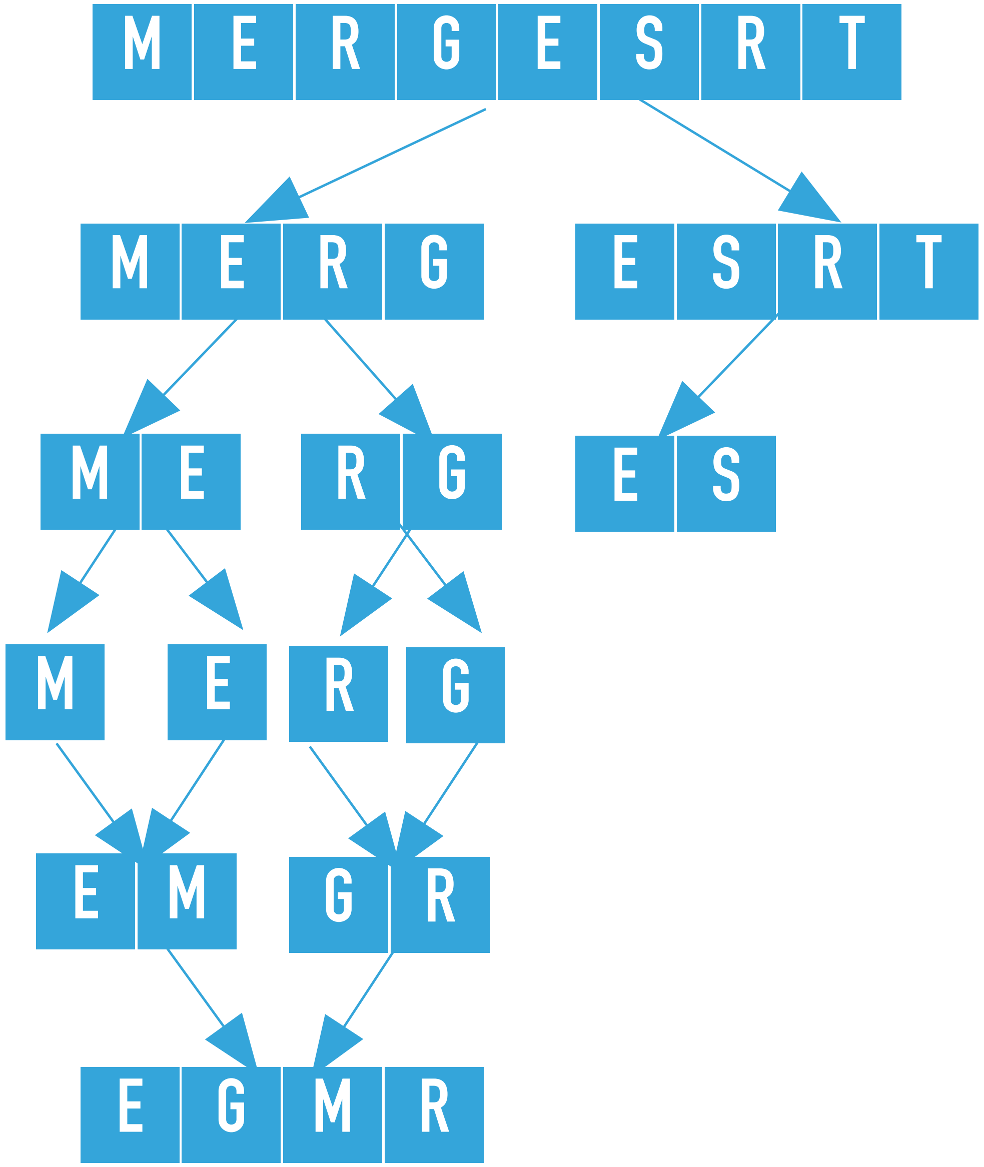
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



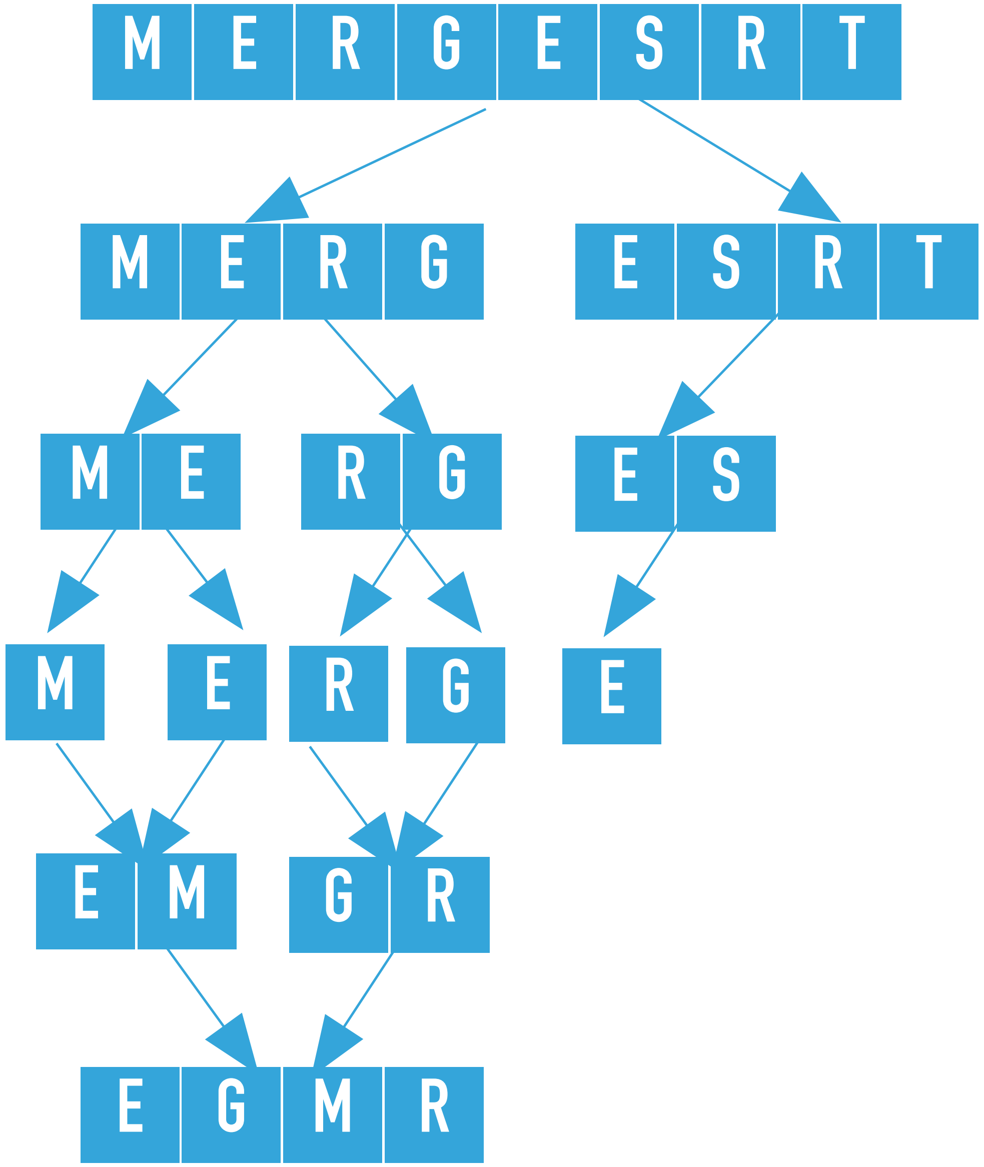
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



```
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`.



M E R G E S R T

M E R G

E S R T

M E

R G

E S

M

E

R

G

E

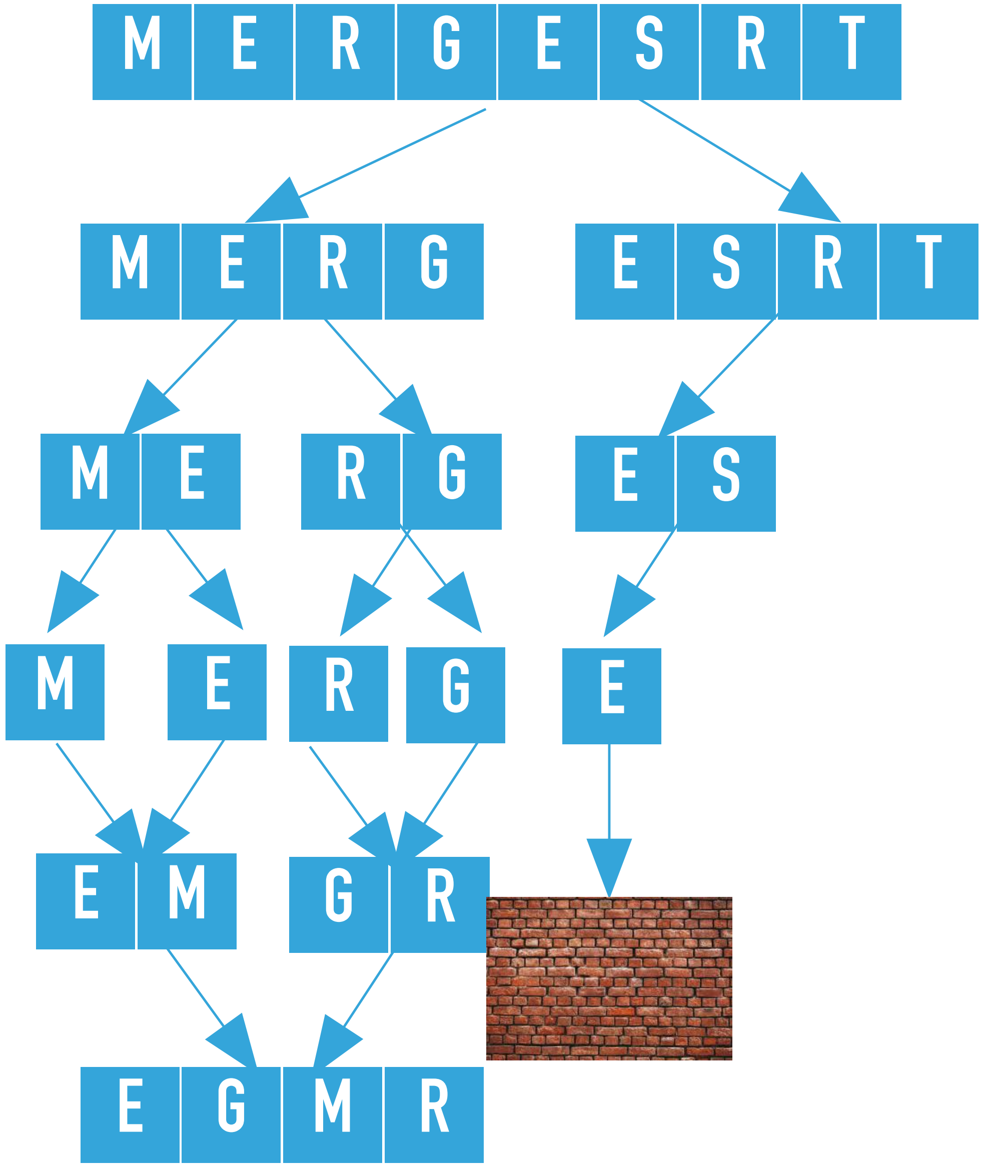
E M

G R

E G M R

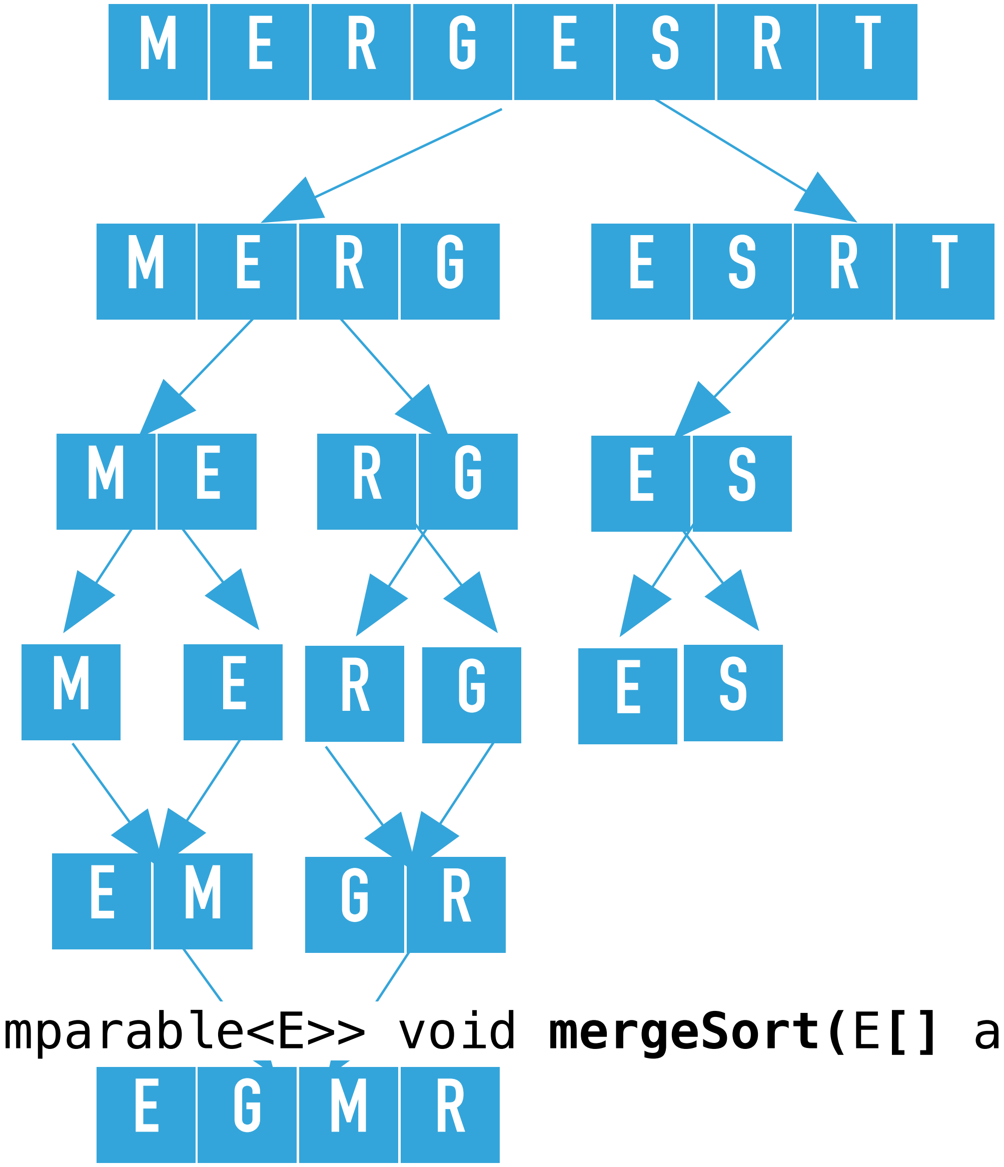
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



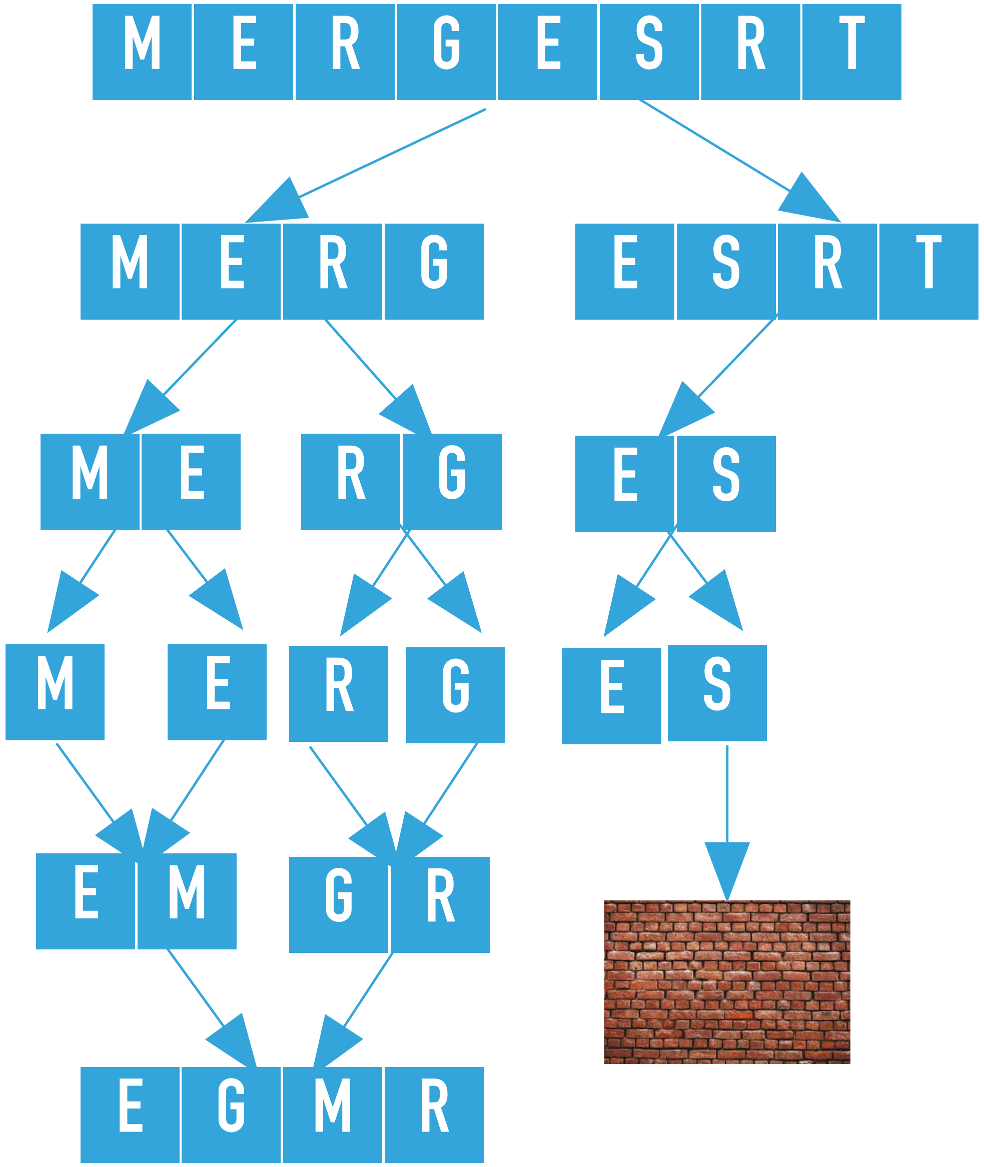
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



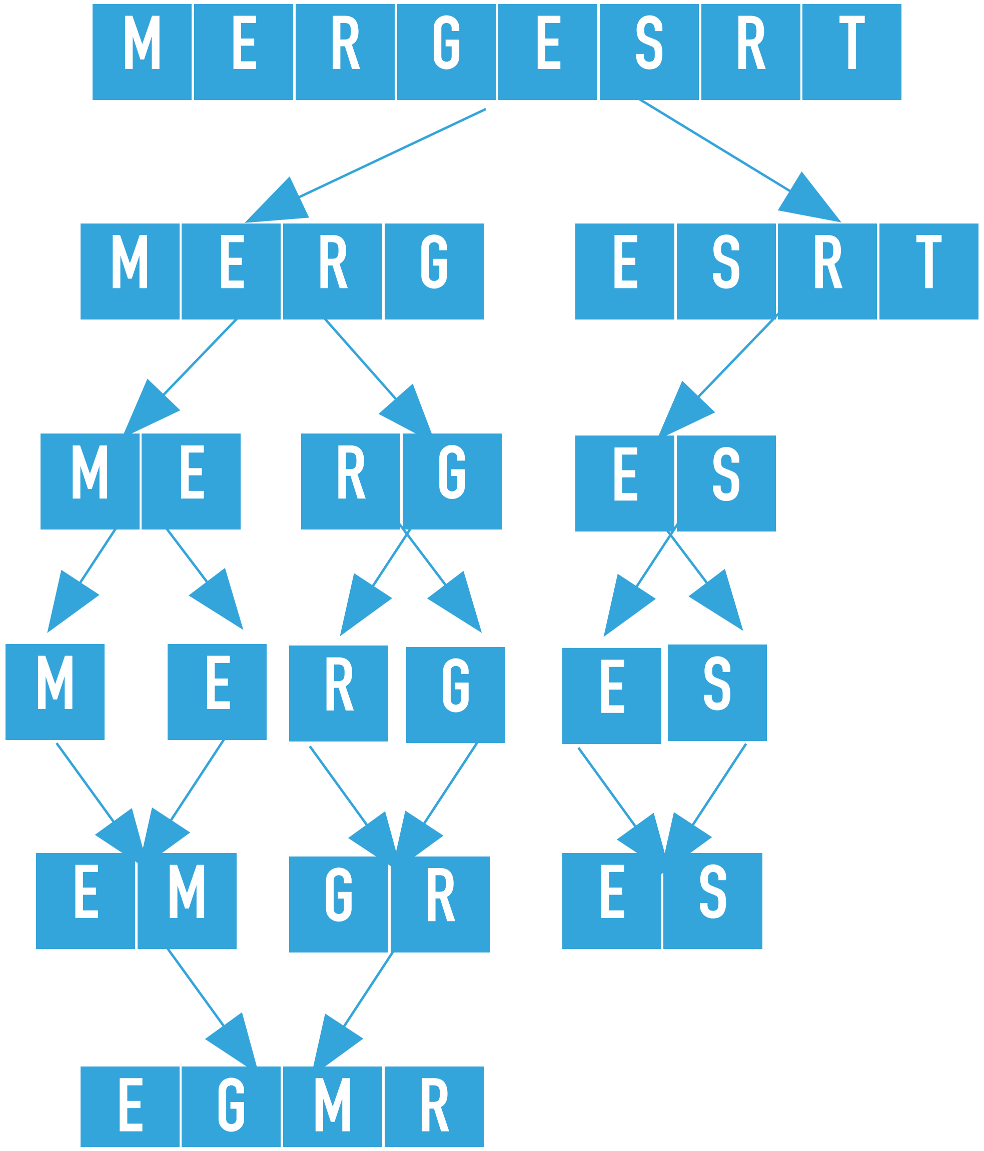
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



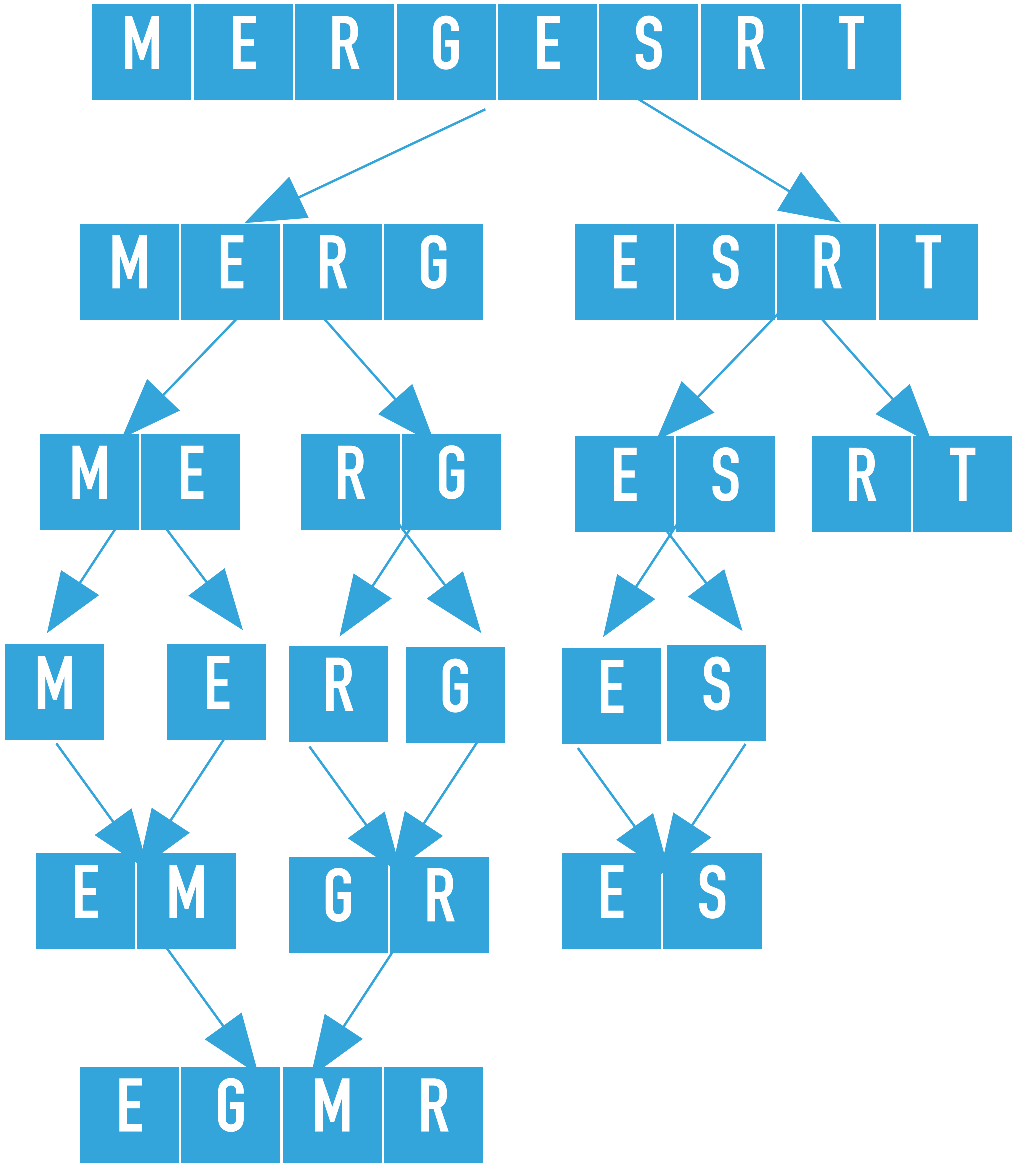
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



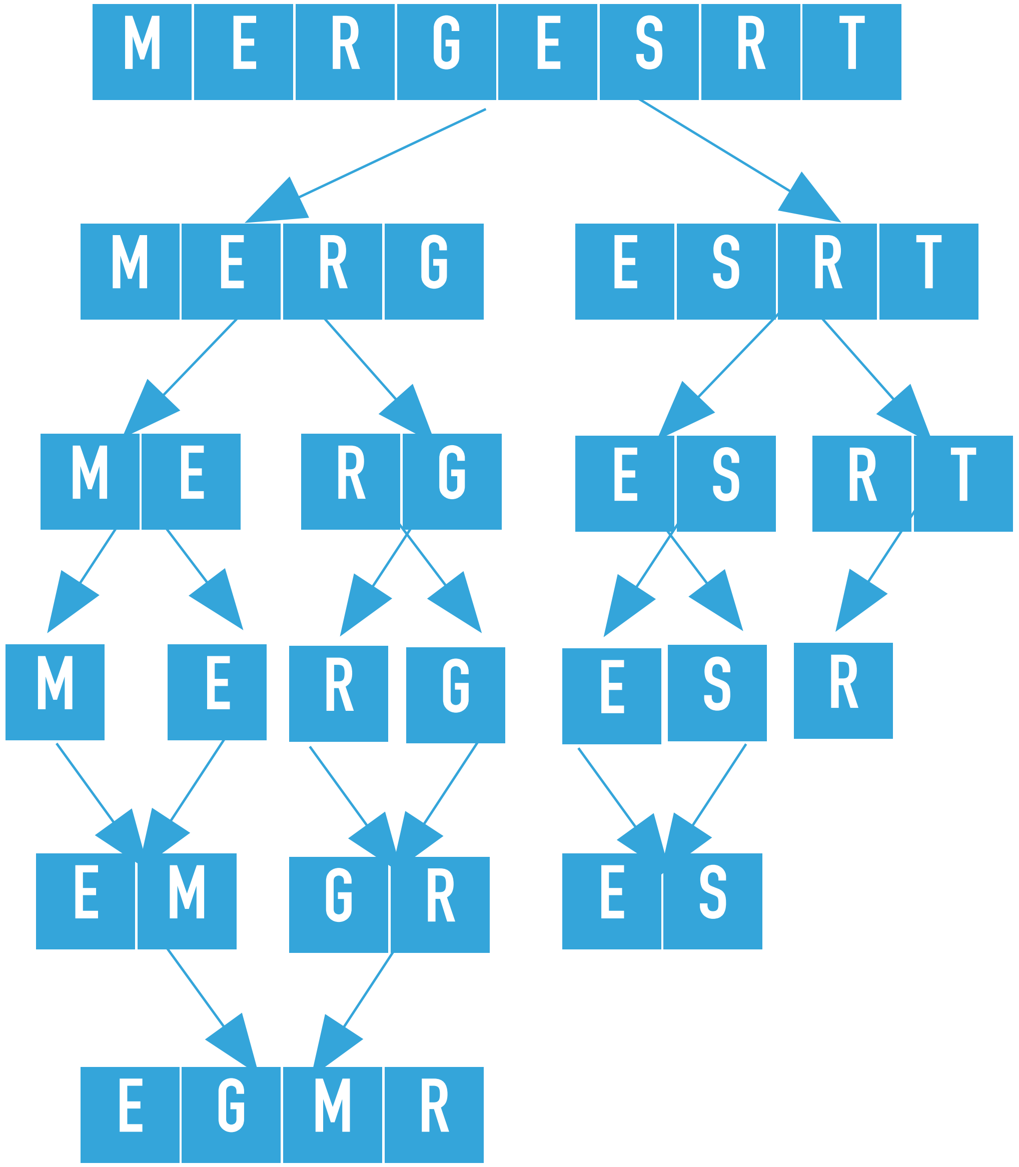
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi – lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



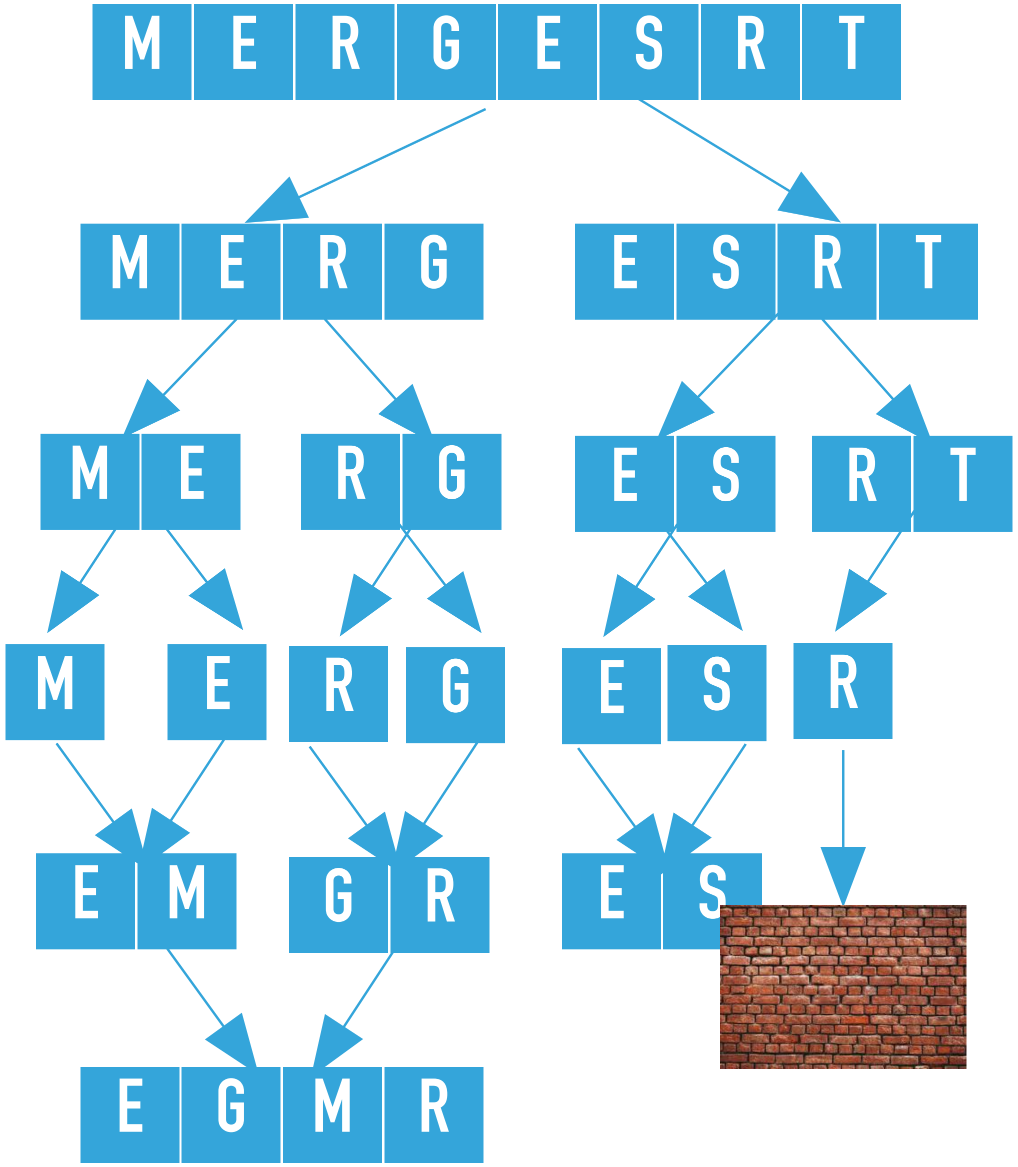
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



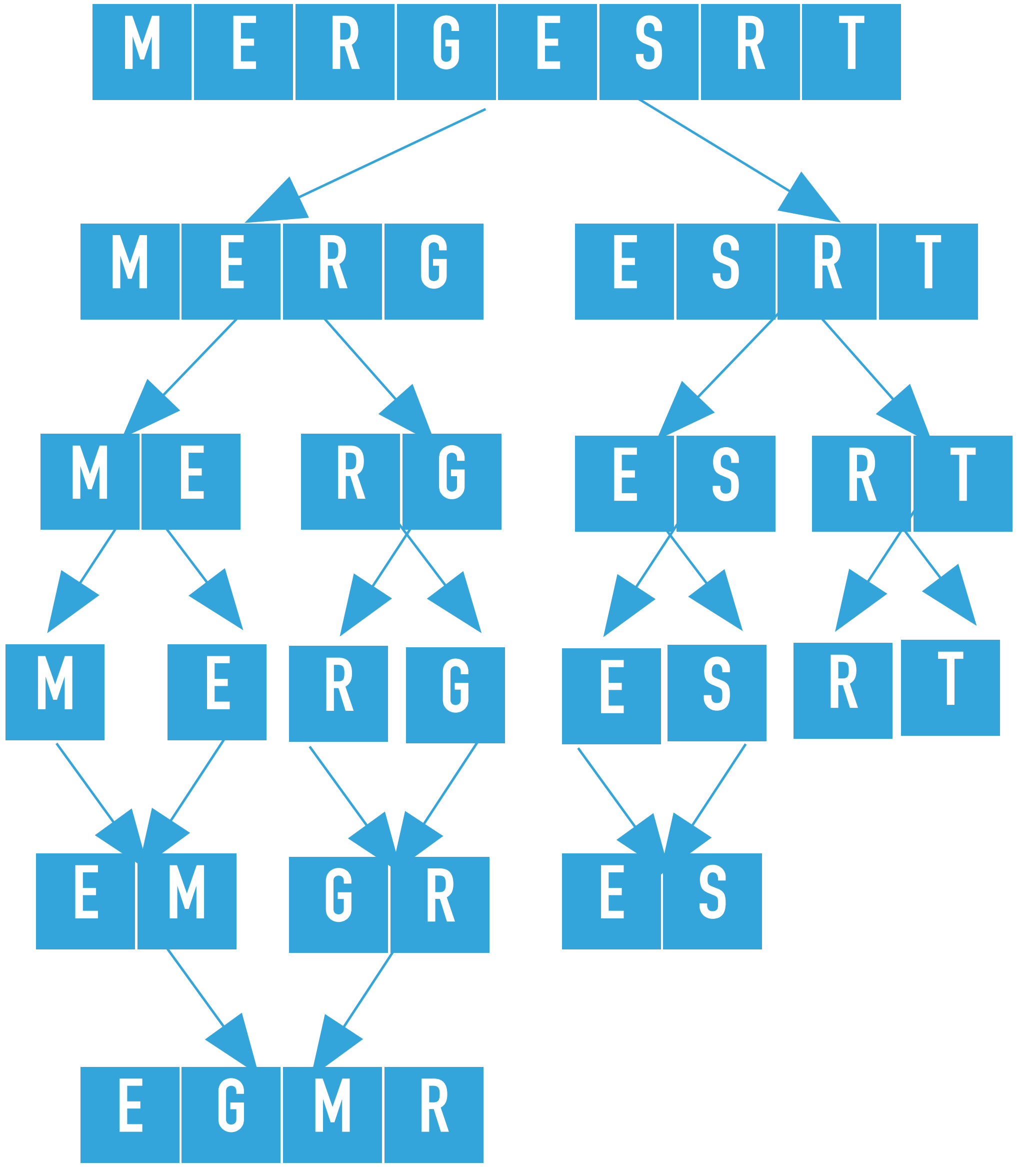
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`.



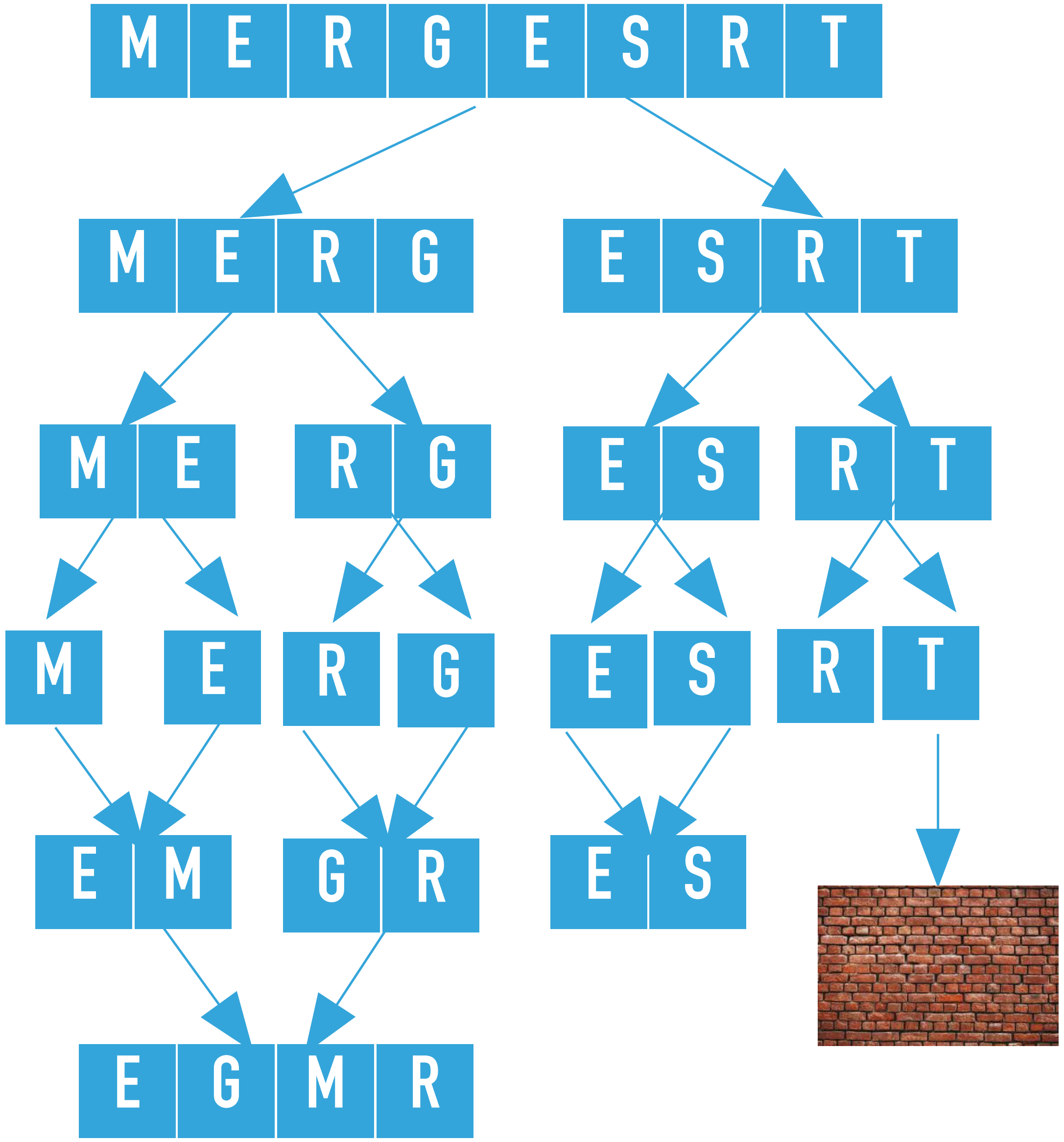
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



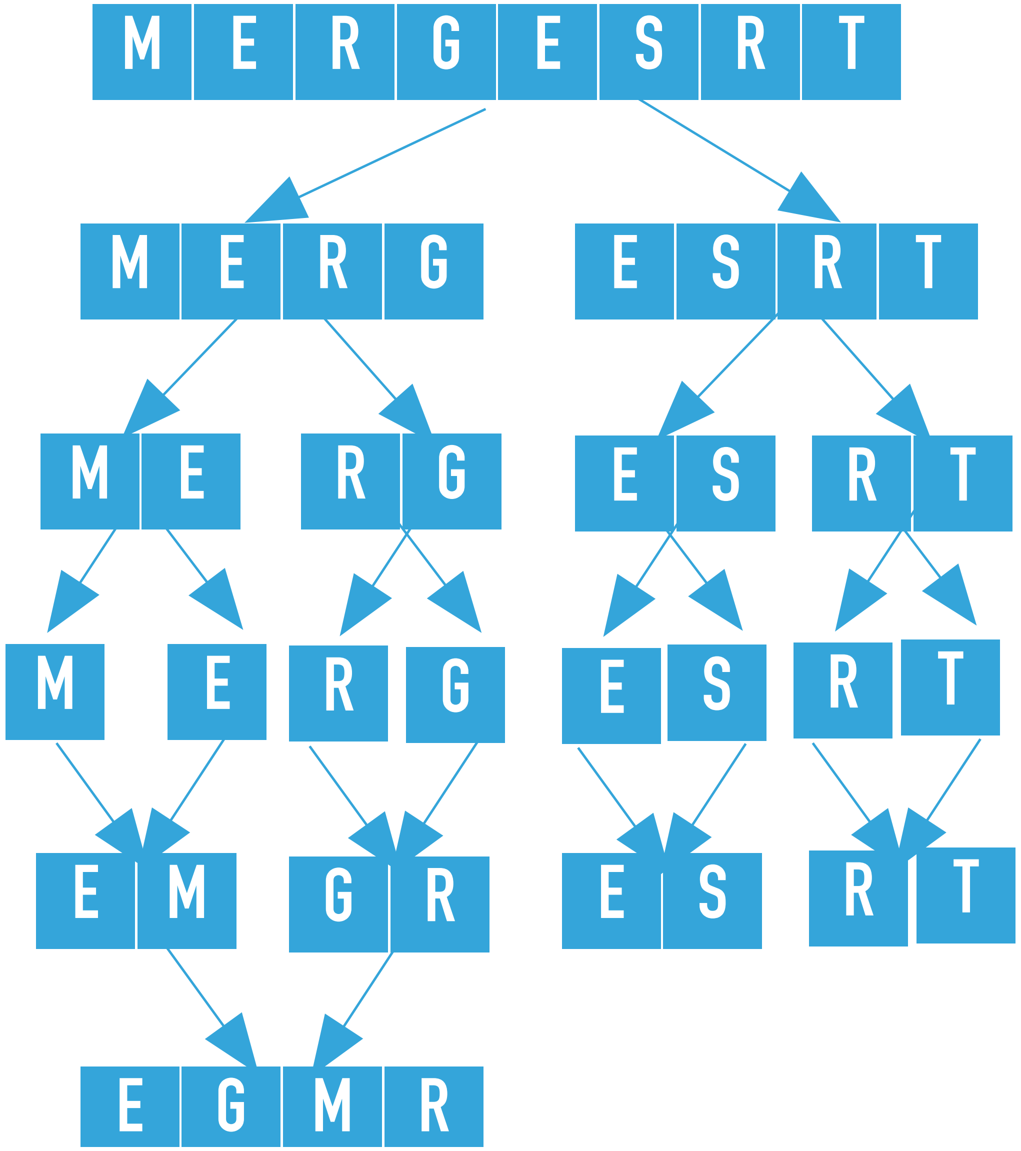
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



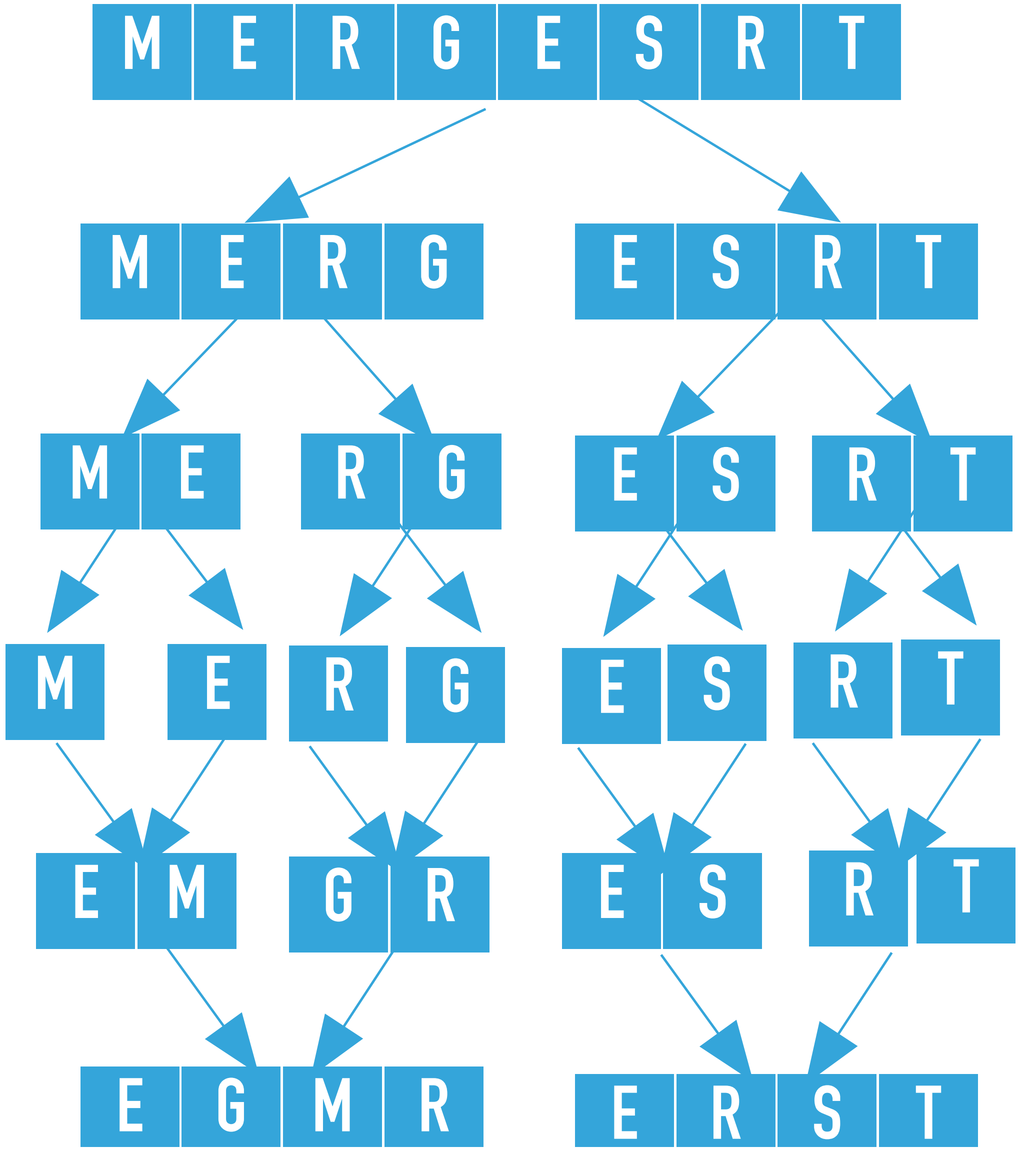
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



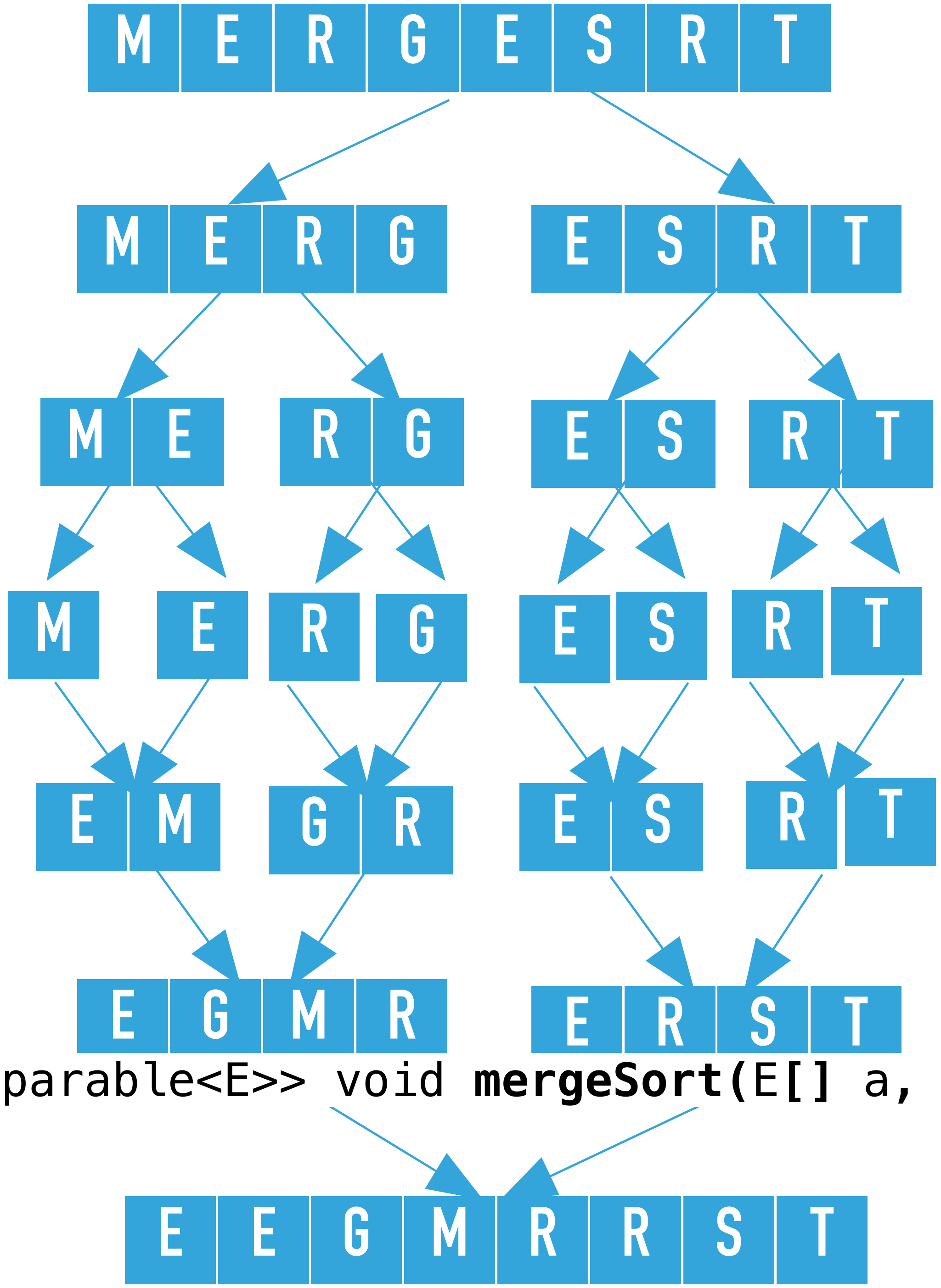
```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int
hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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].`



```
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]`.



```
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].`

Practice time

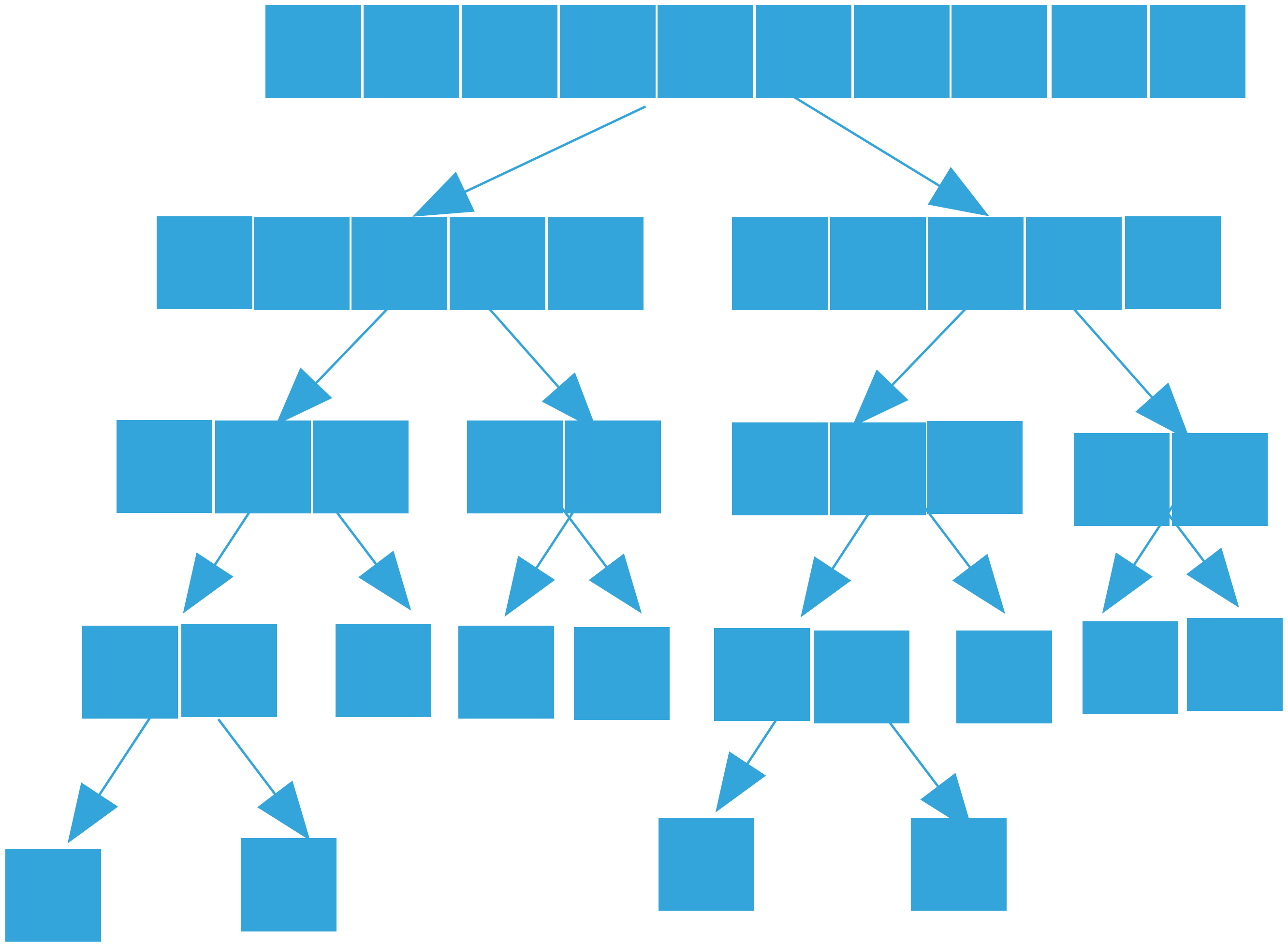
Which of the following subarray lengths will occur when running mergesort on an array of length 10?

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

- 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 $6n \log n$

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

If $n = 4$, 2 levels

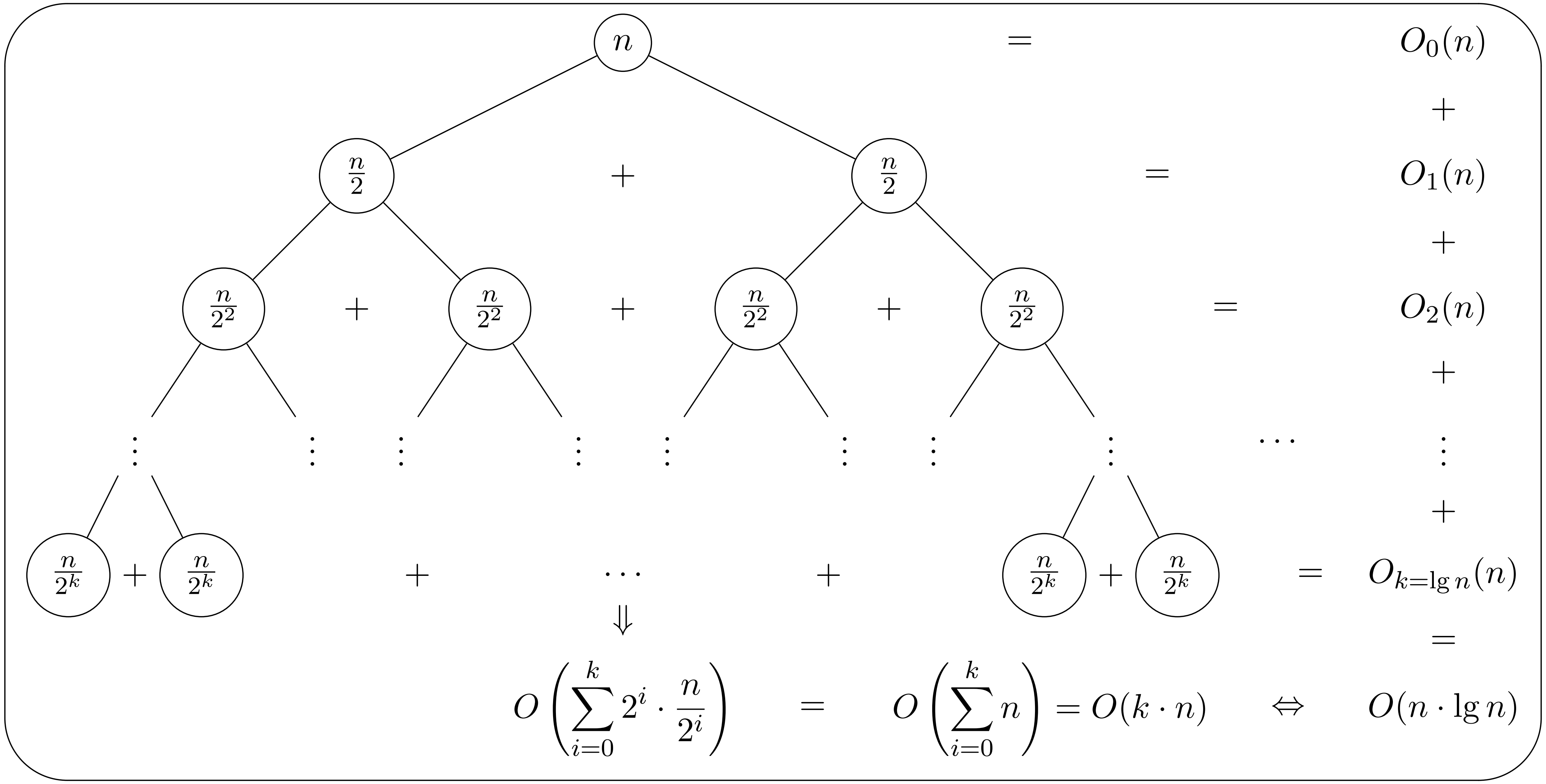
If $n = 8$, 3 levels

If $n = 16$, 4 levels

…

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

or $k = log_2 n$


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

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

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.

- 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).
- 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.

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

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

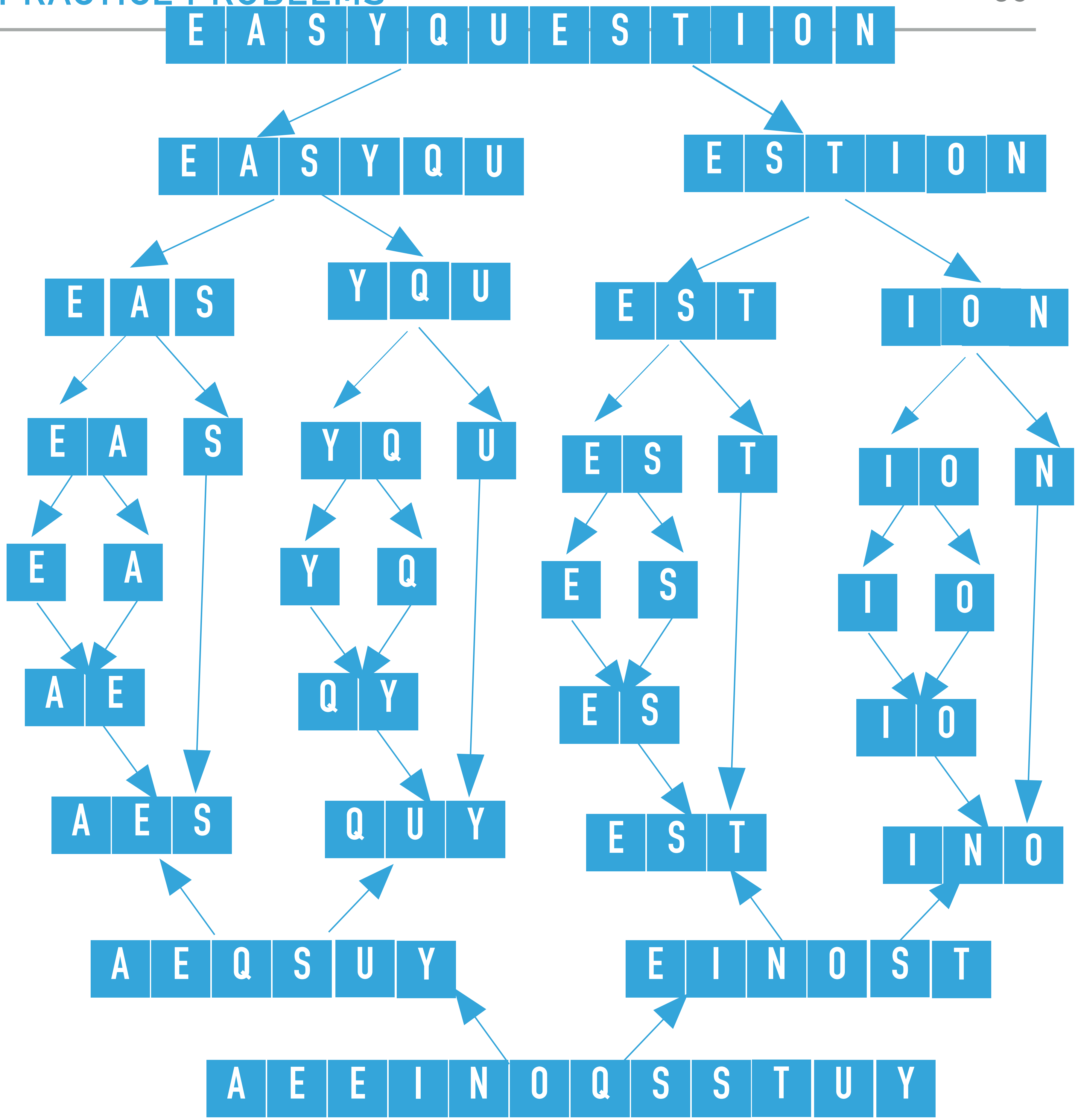
- Give the sequence of subarray lengths in the merges performed by merge sort for n=39.

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

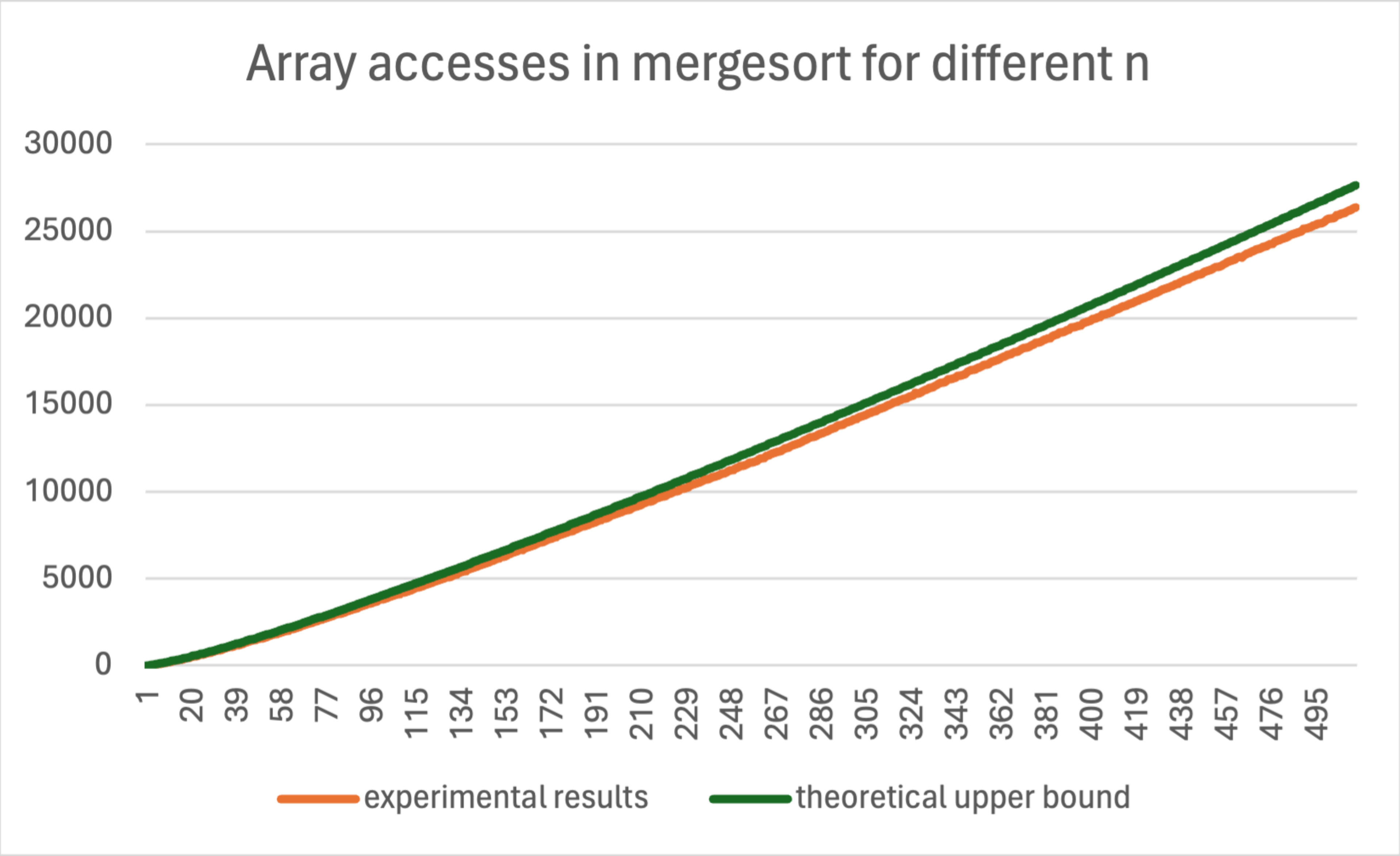
E I N O S T

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


Array accesses in mergesort for different n
30000
25000
20000
15000
10000
5000
0
1
20
39
58
77
96
115
134
153
172
191
210
229
248
267
286
305
324
343
362
381
400
419
438
457
476
495
experimental results
theoretical upper bound