

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

13: Mergesort



Alexandra Papoutsaki
she/her/hers



Tom Yeh
he/him/his

Lecture 13: Mergesort

- ▶ Quiz Review
- ▶ Mergesort

QUIZ - ITERATOR REVIEW

- ▶ hasNext(): if done, return false. If not done, return true.
- ▶ next(): Give the next item in the iteration.

```
public boolean hasNext() {           // Need to work for count is 0 and length
of 1
    return count * 2 < myElts.length; // Can add check if element is not null
}
```

```
public String next() {
    int savedIndex = count * 2;      // Can inc count by 1 and multiply by 2
    count++;                          // Or increment count by 2
    return myElts[savedIndex];      // Need to save index before updating
}
```

```
for (String s : stack)
    StdOut.println(s);
```

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

ITERATOR FOR ARRAY - ITERATES THROUGH ARRAY IN REVERSE ORDER (STACK)

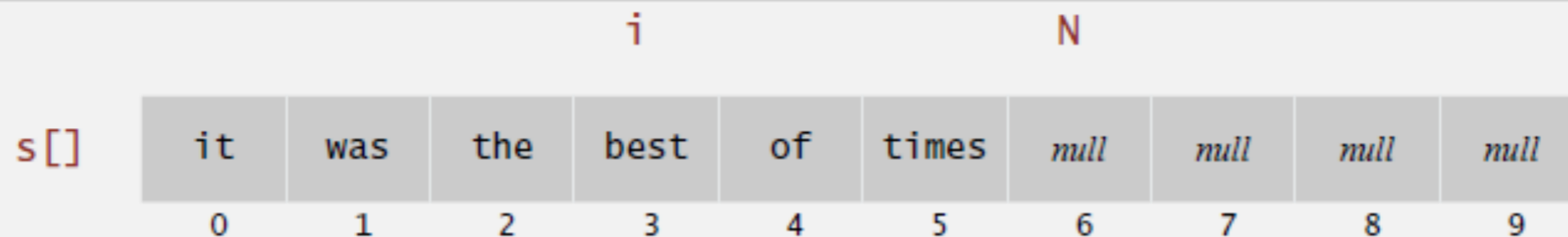
```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove() { /* not supported */ }
        public Item next() { return s[--i]; }
    }
}
}
```



LINKED LIST ITERATOR

```
import java.util.Iterator;

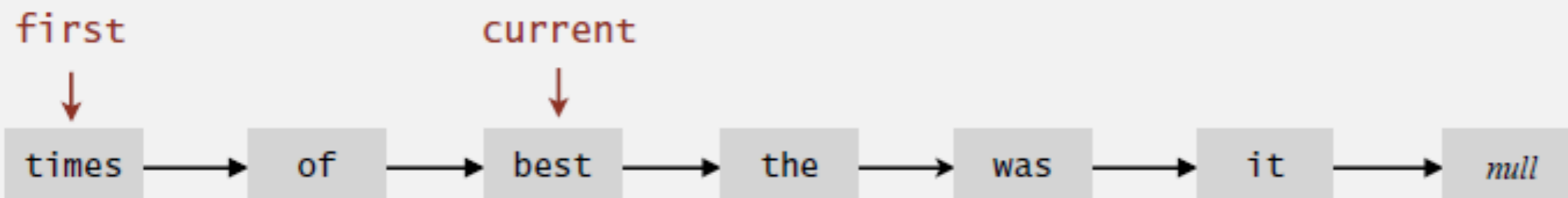
public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }
        public void remove()     { /* not supported */ }
        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

throw UnsupportedOperationException
throw NoSuchElementException
if no more items in iteration



Helper Functions: Refer to data through less and exchange

- ▶ For exchanges, make sure we know which elements are being exchanged. Selection sort vs Insertion sort

Less. Is item v less than w ?

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

Exchange. Swap item in array $a[]$ at index i with the one at index j .

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

Comparable

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

How to make your class T comparable?


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

Comparable interface (built in to Java)

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

sort implementation

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



Comparator

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

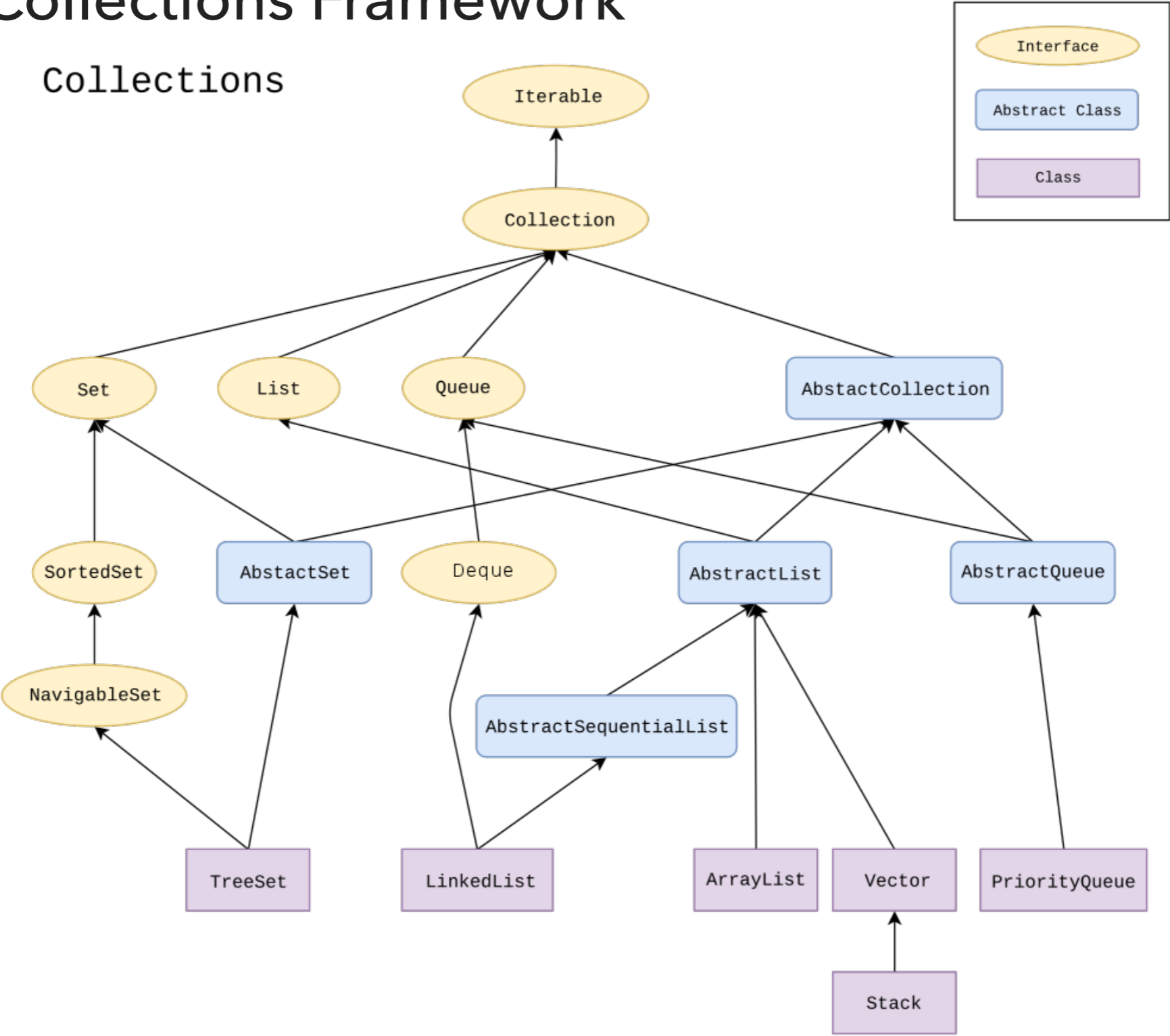
How to define an alternative ordering for your class T?

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

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

// Example
public static Comparator<Employee> ageComparator = new Comparator<Employee>()
{
    @Override // indicates method overriding the superclass' method
    public int compare(Employee e1, Employee e2) {
        {
            return Integer.valueOf(e1.getAge()).compareTo(Integer.valueOf(e2.getAge()));
        }
        // getAge returns an int. Create an Integer object with Integer.valueOf(int)
    }
};
```

The Java Collections Framework



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

Alternative sorting of Collections

- ▶ Collections class contains:
 - ▶ `static <T> void sort(List<T> list, Comparator<? super T> c)`
- ▶ `Collections.sort(list)`
 - ▶ This method will use the natural order `compareTo` method
- ▶ `Collections.sort(list, someComparator);`
 - ▶ `Collections.sort(list, T.InnerAnonymousClass);`
 - ▶ Example: `Collections.sort(list, Employee.nameComparator);`
 - ▶ If list's elements do not implement `Comparable` or cannot be compared with `Comparator`, throw `ClassCastException`.

Example: Natural and alternative sorting for Employees

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

Lecture 13: Mergesort

- ▶ Mergesort

MERGESORT

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

15

Basics

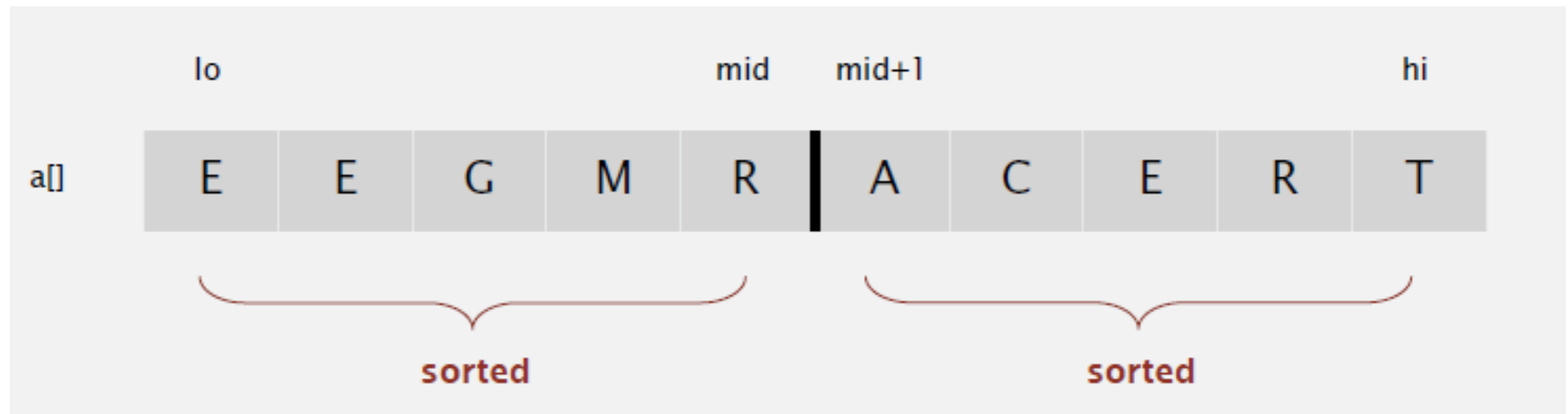
Mergesort overview

- ▶ Invented by John von Neumann in 1945
- ▶ Basic sort for many languages (Java)
- ▶ 1st non trivial alg on a computer
- ▶ Algorithm sketch:
 - ▶ Divide array into two halves.
 - ▶ **Recursively** sort each half.
 - ▶ Merge the two halves



1st step - Understand how merging works

Merge two already sorted halves into one sorted array



Copy to auxiliary array

Merging Example - copying to auxiliary array

Array a

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

Array aux

0	1	2	3	4	5	6	7	8	9

Merging Example - copying to auxiliary array

Array a

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

Array aux

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

Merging Example - copy elements back to original array in order

Maintain 3 indices: 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

k $k = \text{current entry on the sorted result}$

Array aux

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

i j

$i = \text{current entry on the left half}$

$j = \text{current entry on the right half}$

Merging Example - copy elements back to original array in order

Compare minimum in each subarray

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices i and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices i and k

Array a (sorted result)

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

k

Array aux

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

i j

Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices j and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices j and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices i and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices j and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices i and k

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

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

k

Array aux

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

i

j

Merging Example - copy elements back to original array in order

Done

Array a (sorted result)

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

k

Array aux

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

i

j

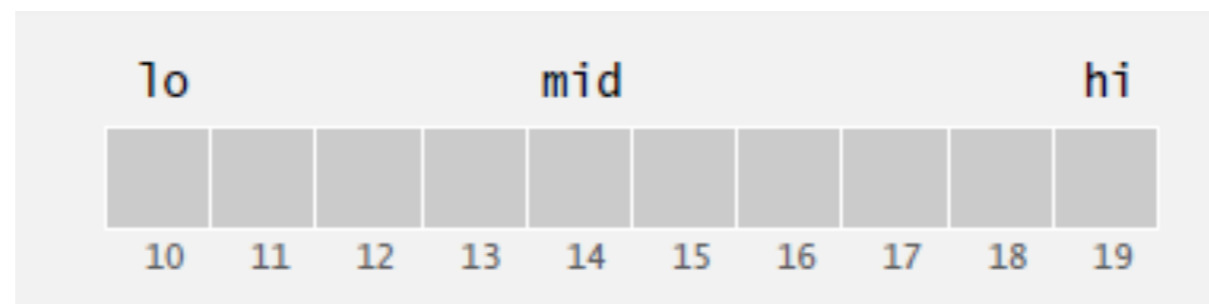
2.2 MERGING DEMO



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

Merging two already sorted halves into one sorted array

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {  
  
    for (int k = lo; k <= hi; k++) // copy to aux array  
        aux[k] = a[k];  
  
    int i = lo, j = mid+1;  
        // lo and mid+1 are the start of the 2 sorted halves  
  
    for (int k = lo; k <= hi; k++) {  
        if (i > mid) //ran out of elements in the left subarray  
            a[k] = aux[j++];  
        else if (j > hi) //ran out of elements in the right subarray  
            a[k] = aux[i++];  
        else if (less(aux[j], aux[i])) // Compares left and right subarray  
            a[k] = aux[j++]; // right element smaller  
        else  
            a[k] = aux[i++]; // left element smaller  
    }  
}
```



Practice time

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

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

B. $\sim 1/2n$

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

D. $\sim n$

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int
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)
            a[k] = aux[j++];
        else if (j > hi)
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```

Answer

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

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

Best case example

When 1 subarray's elements are all smaller than the other subarray's 1st element.

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

Worst case example

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

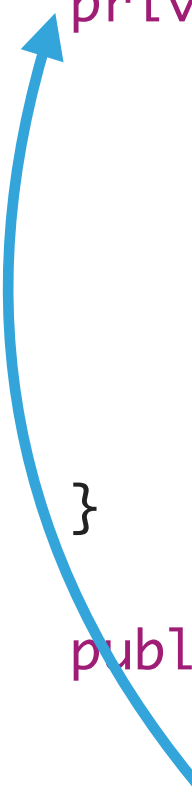
Mergesort - the quintessential example of **divide-and-conquer**

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) // Base case, return
        return;
    int mid = lo + (hi - lo) / 2; // Computes midpoint
    sort(a, aux, lo, mid); // Sort the first half
    sort(a, aux, mid+1, hi); // Sort the second half
    merge(a, aux, lo, mid, hi); // Merge the 2 halves
}
```

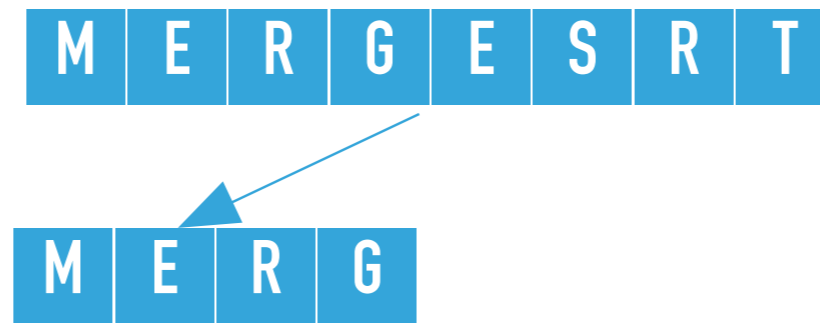
```
public static void sort(Comparable[] a) {
    Comparable[] aux = new Comparable[a.length]; // Create aux array
    sort(a, aux, 0, a.length - 1); // Recursively call sort
}
```

// Important to create auxiliary array outside the recursive sort method.

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

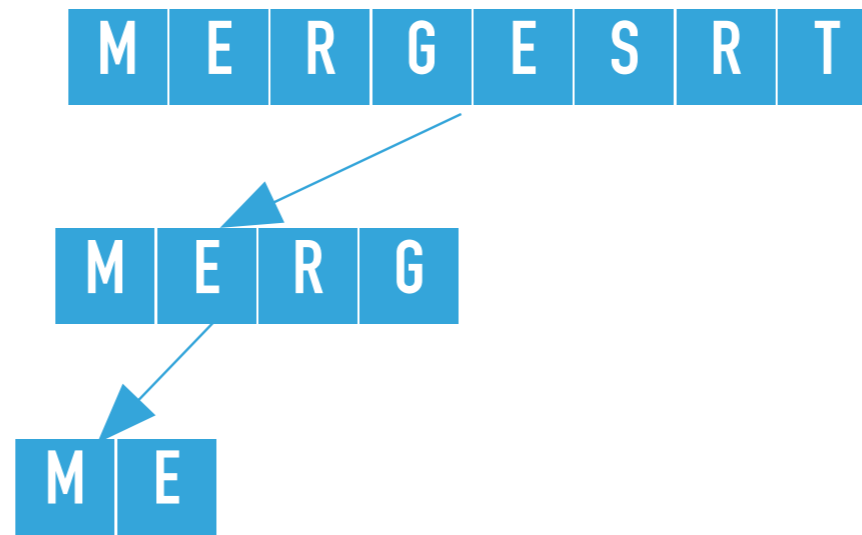


sort([M, E, R, G, E, S, R, T]) calls
sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0,
7) where the array of nulls is the auxiliary array, **lo = 0 and hi = 7.**



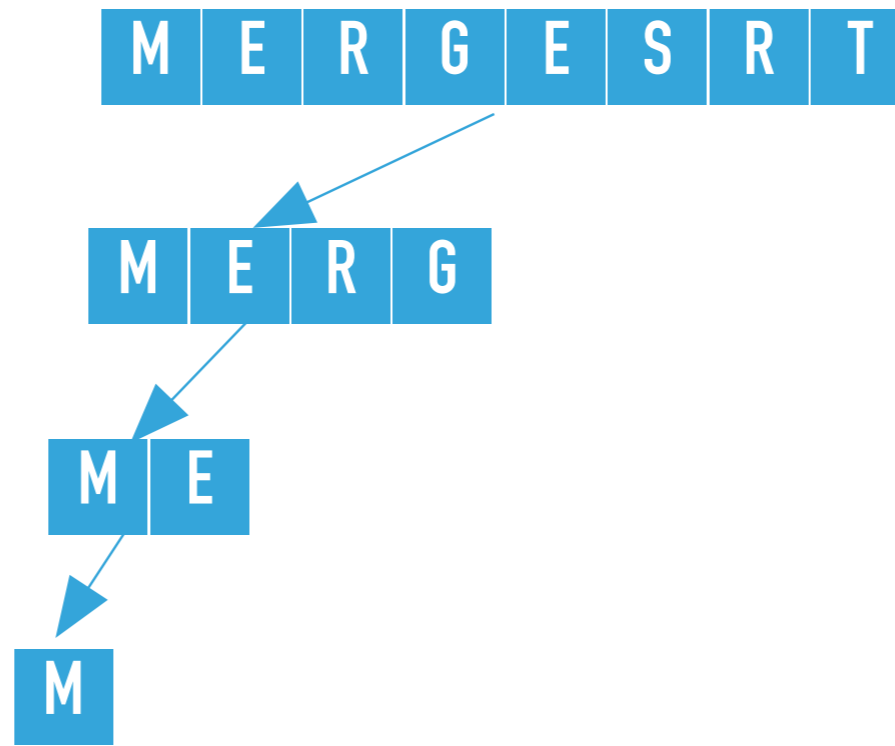
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo)  
        return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

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



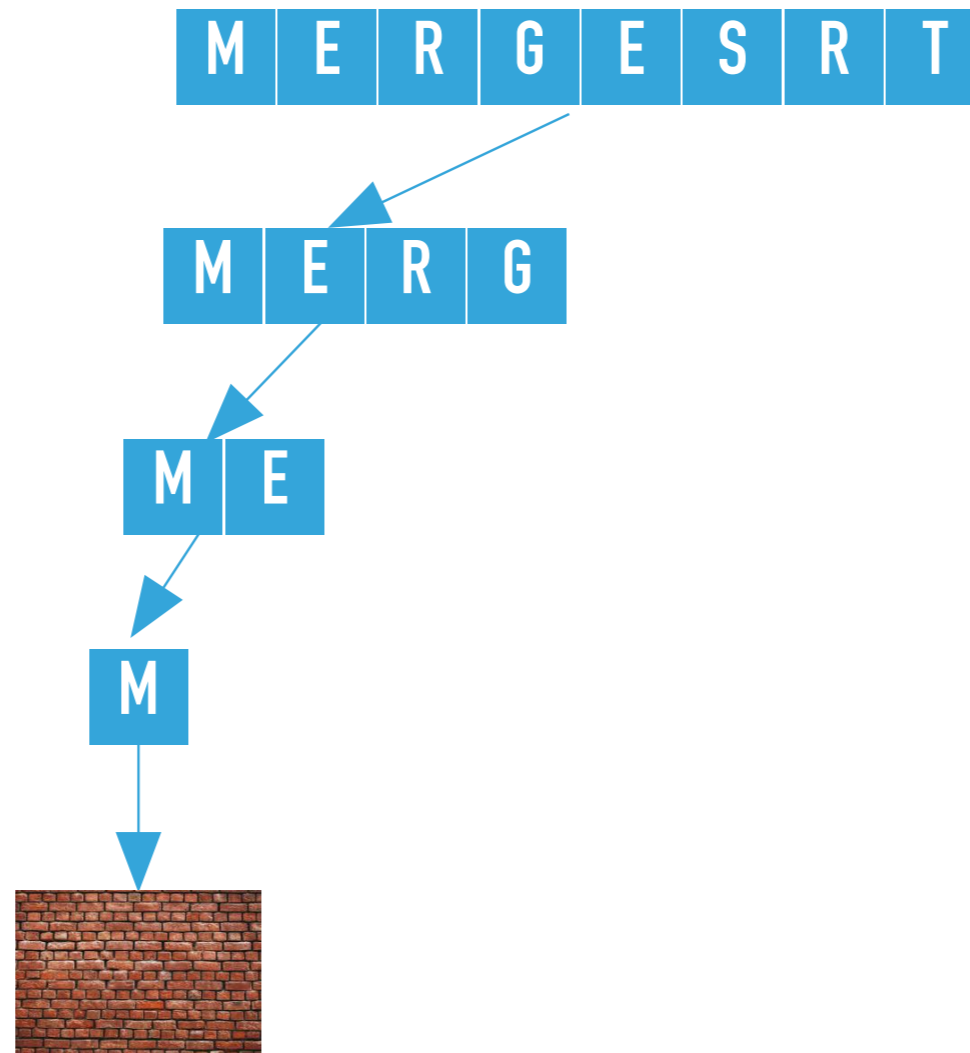
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



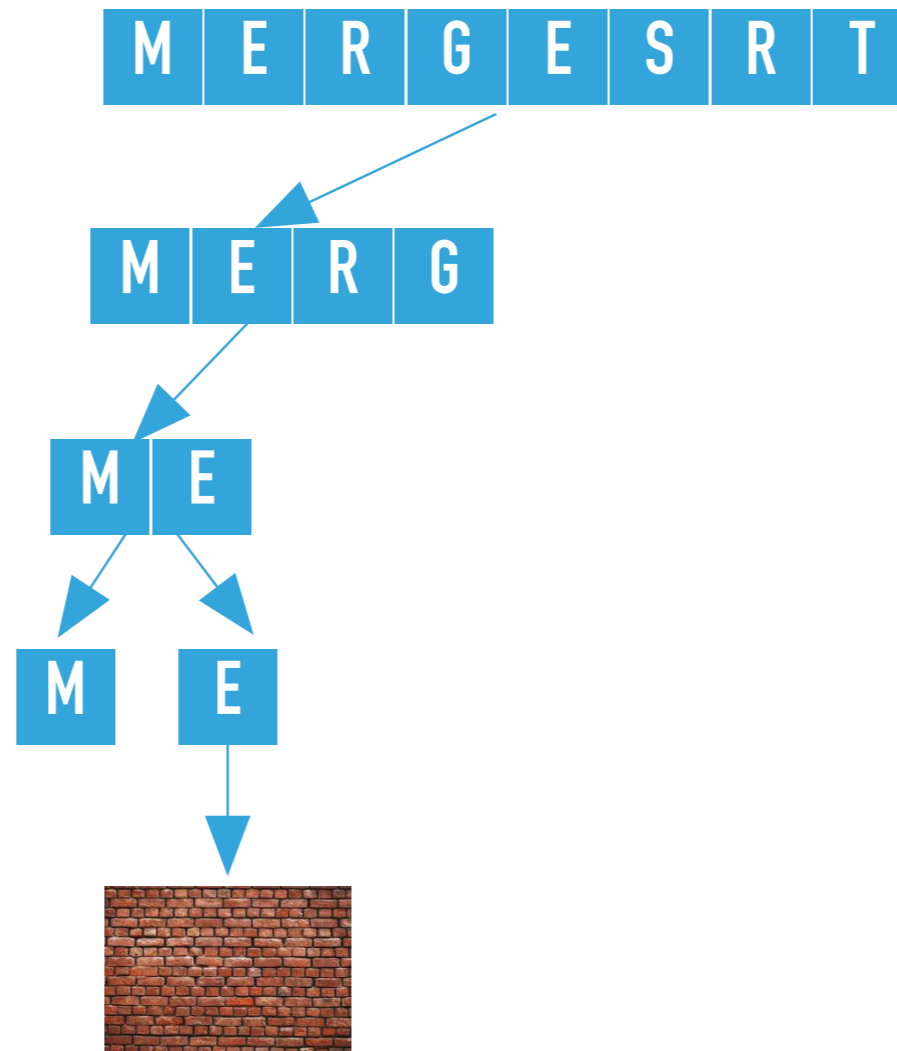
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



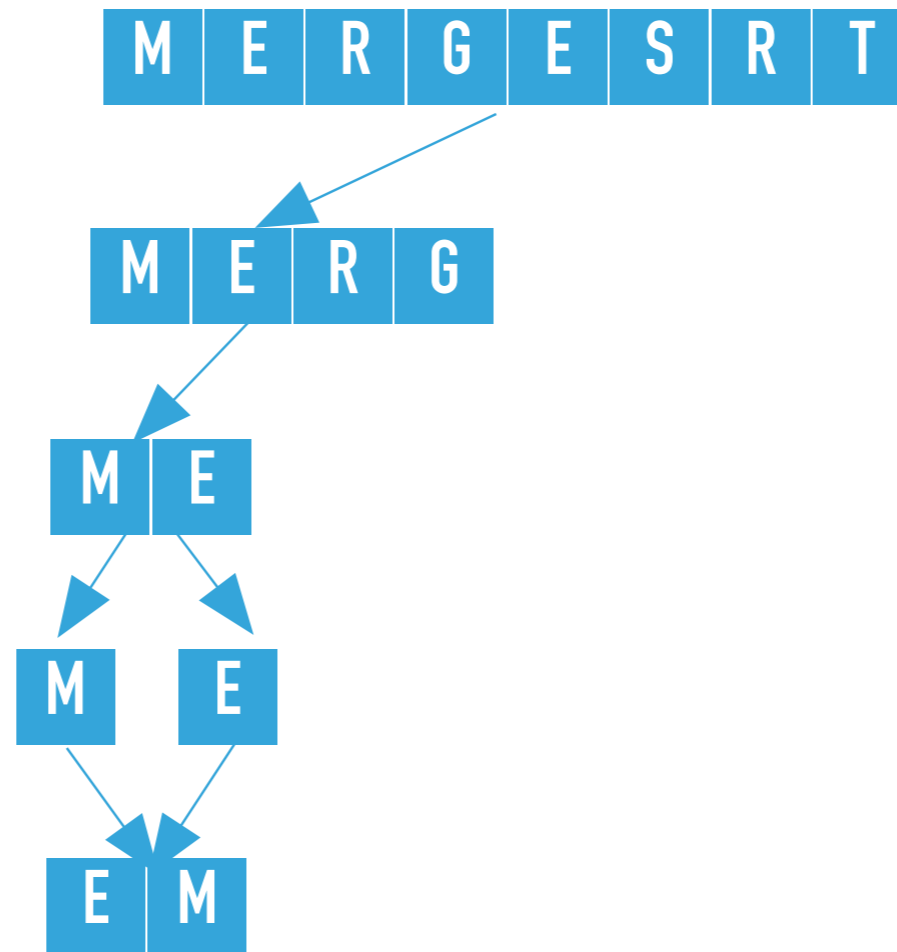
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



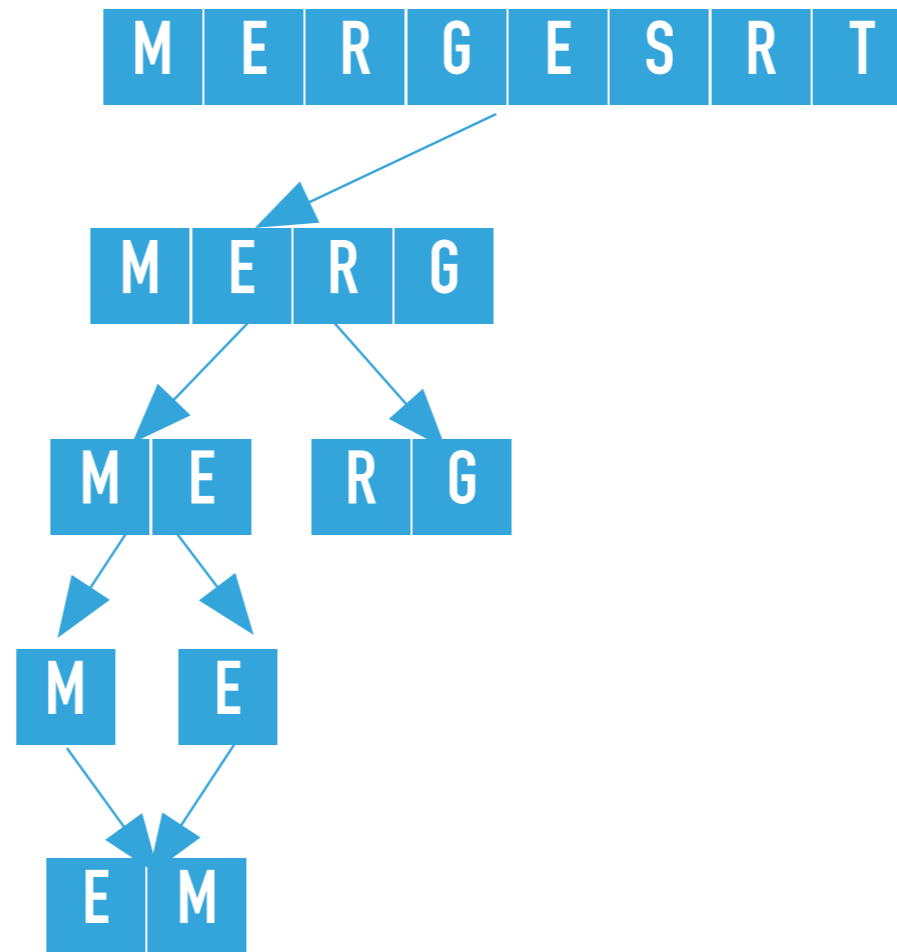
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



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

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

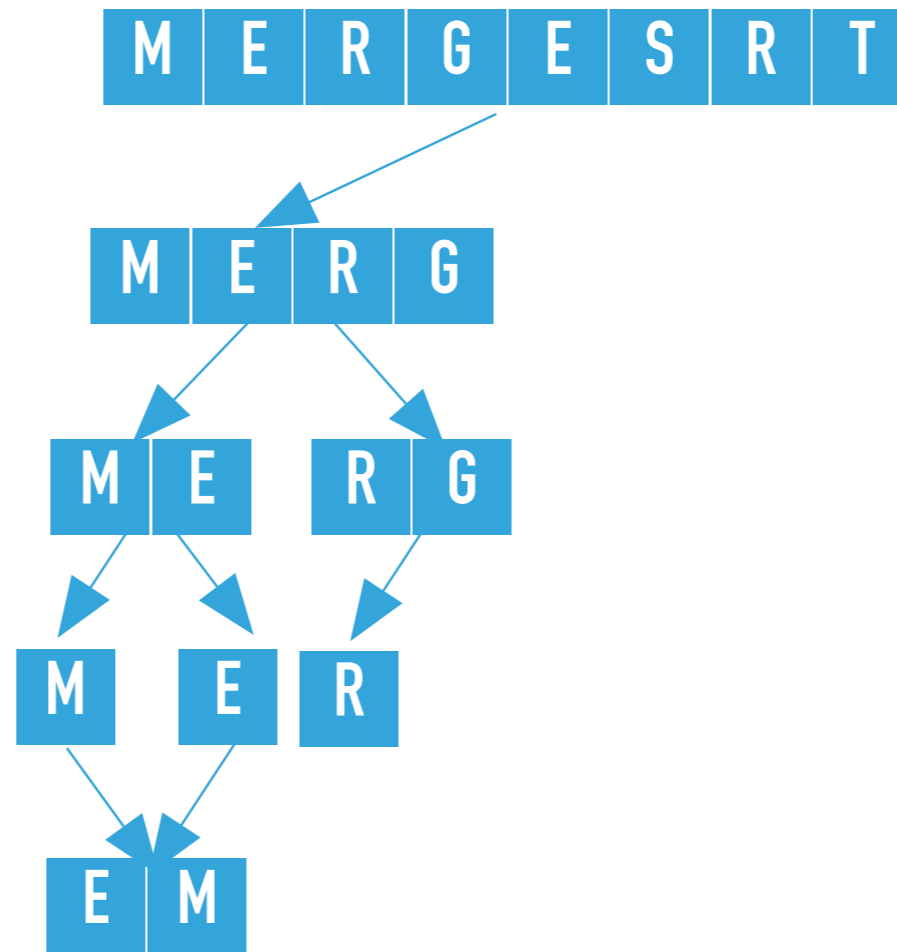


```

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

```

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

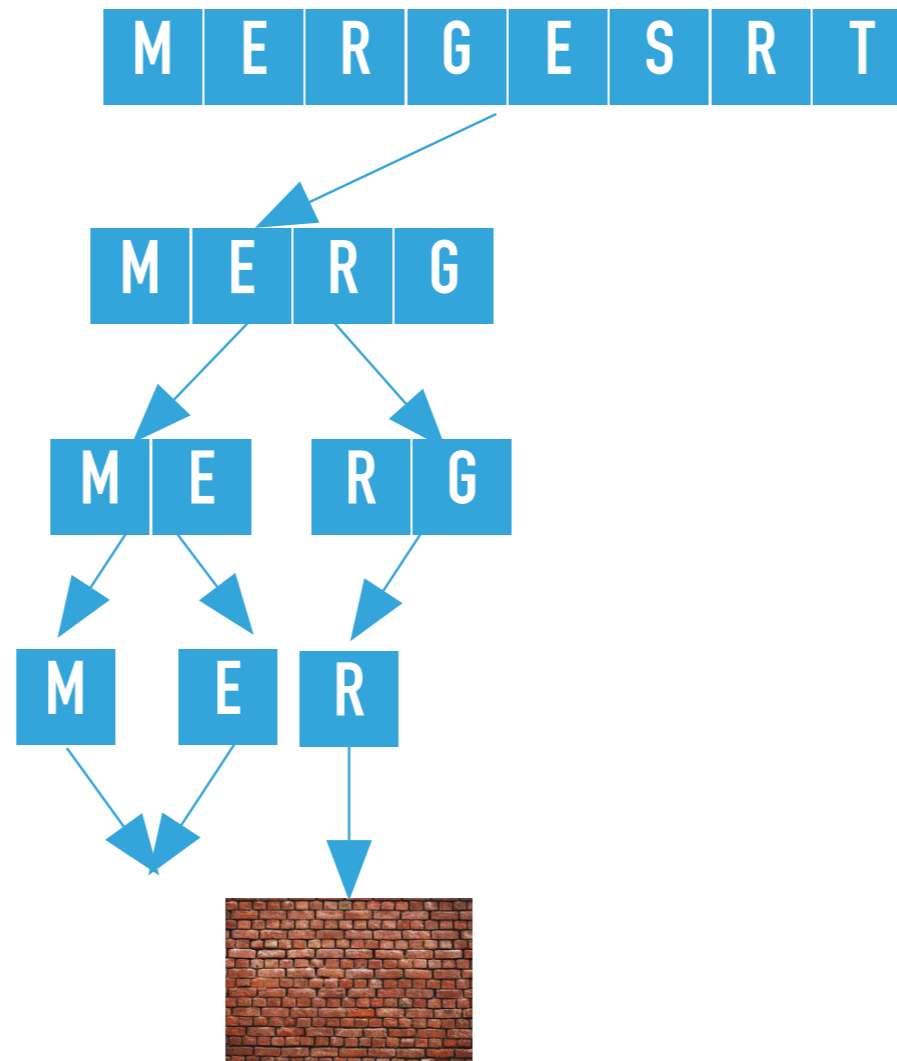


```

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

```

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

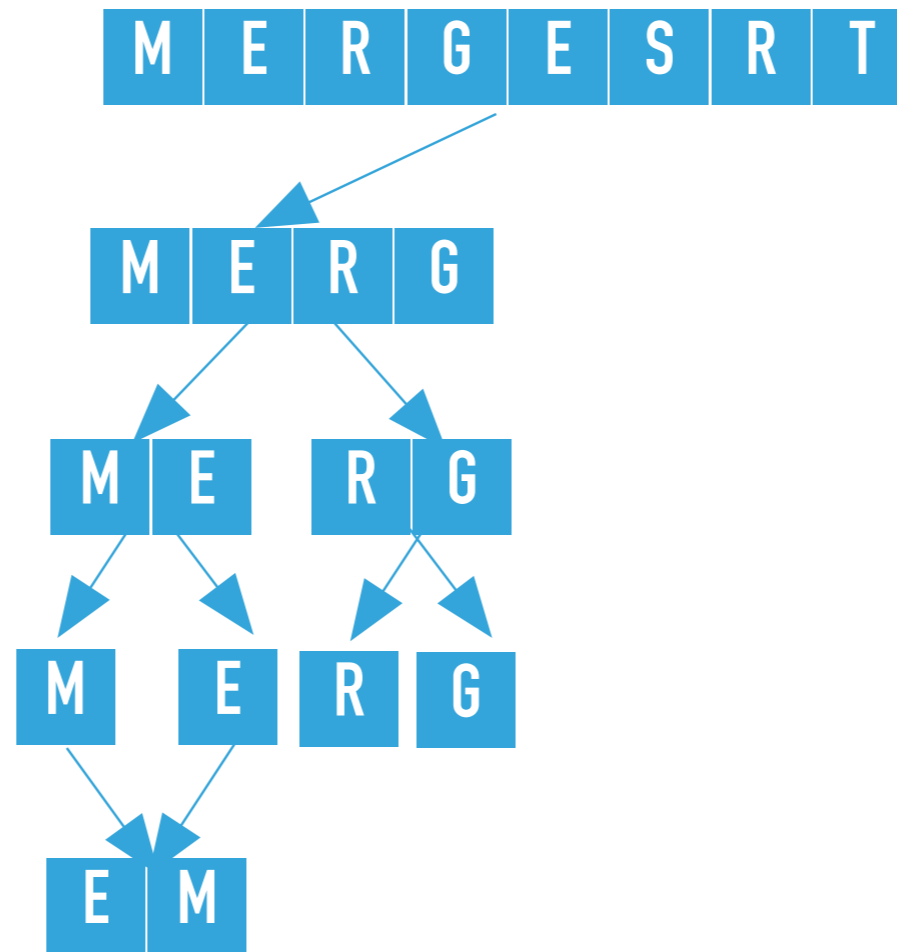


```

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

```

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

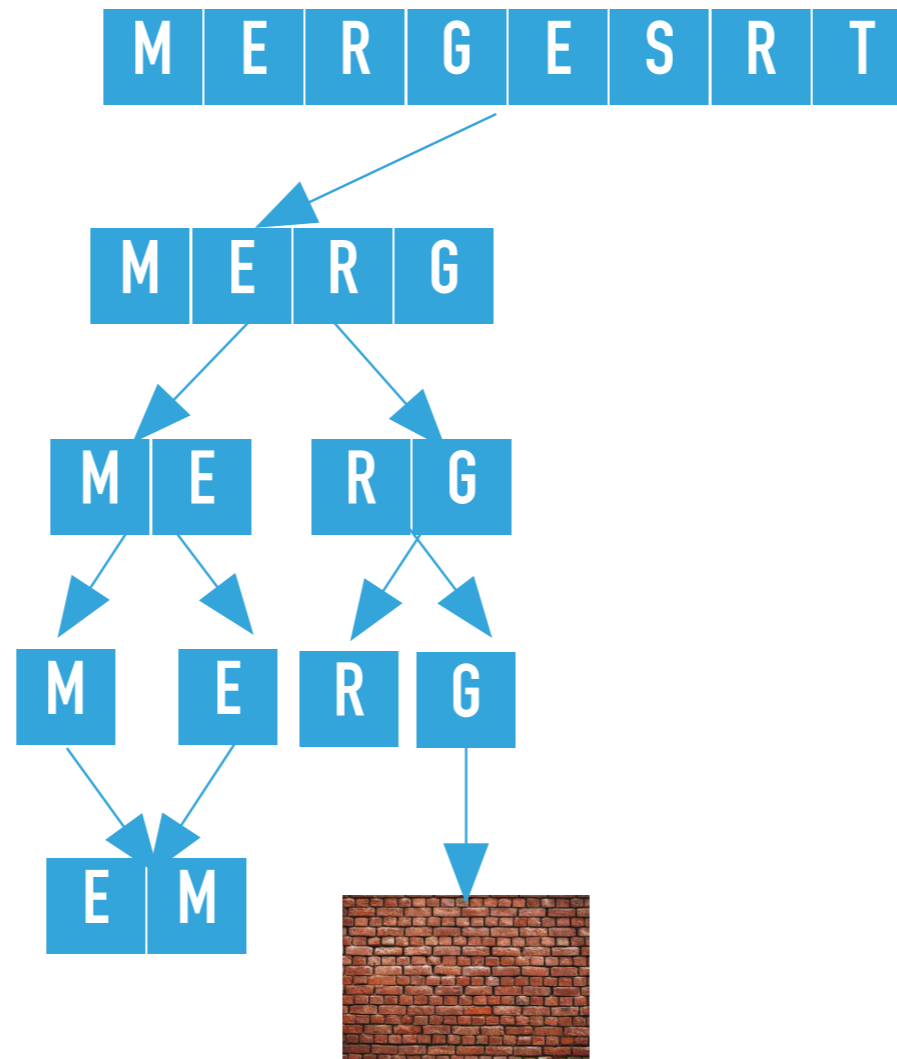


```

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

```

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

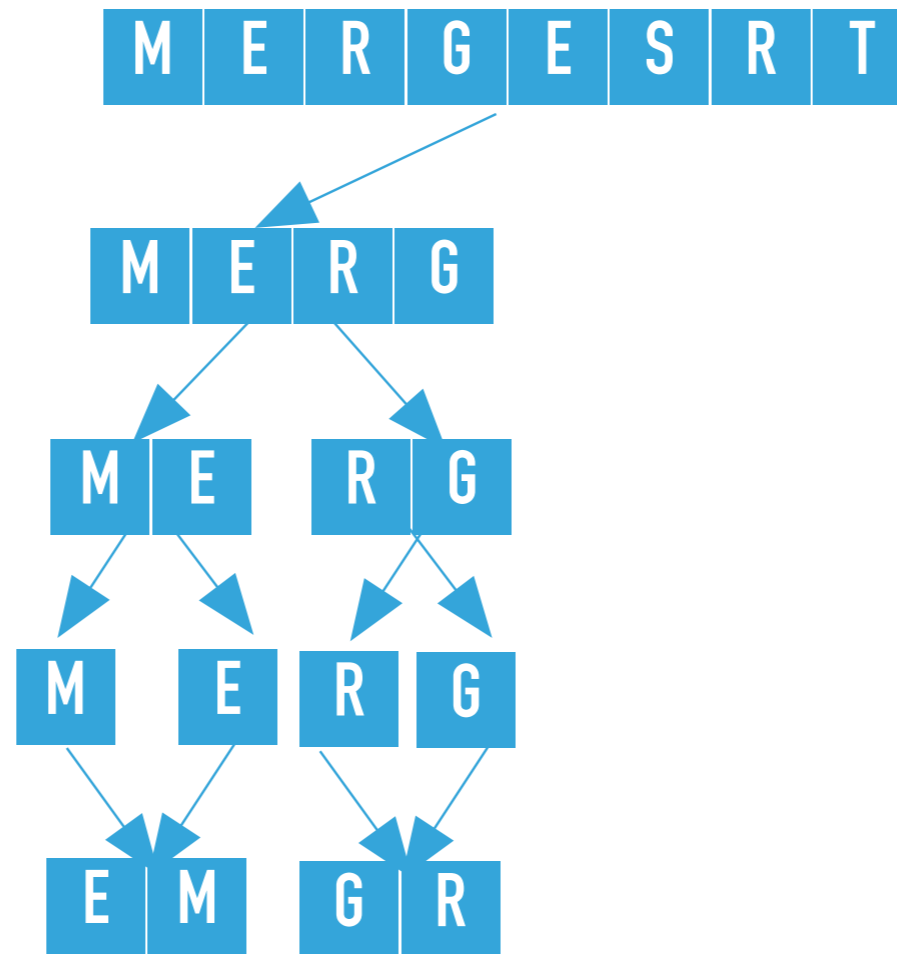


```

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

```

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

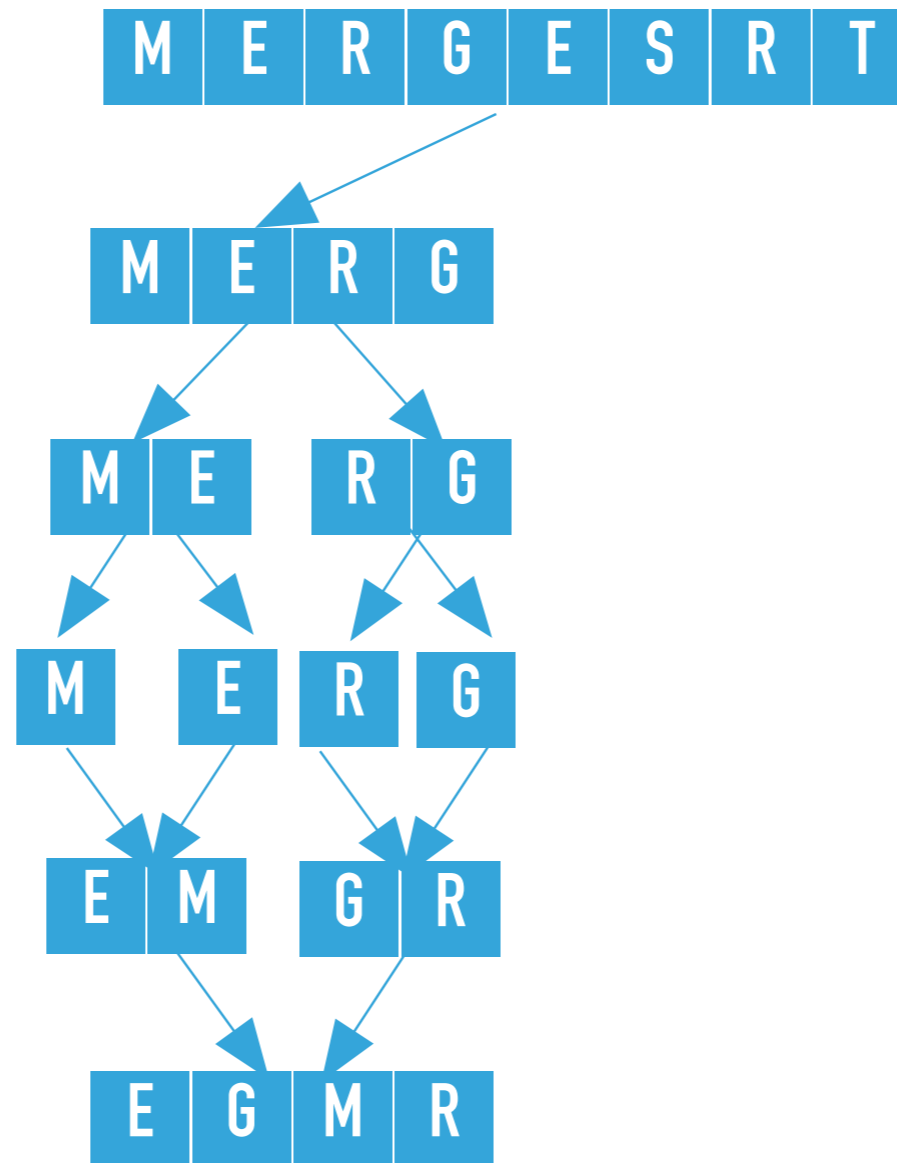


```

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

```

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

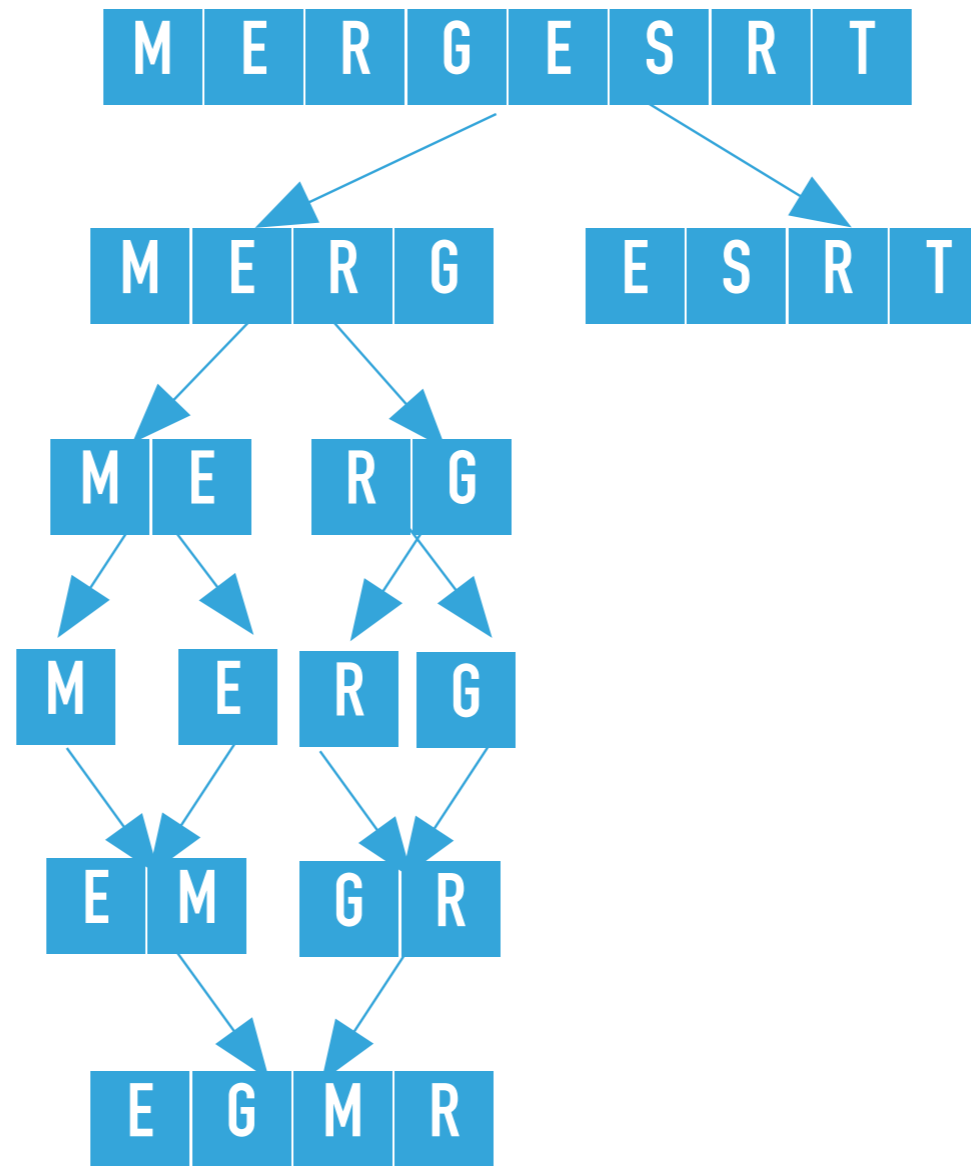


```

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

```

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

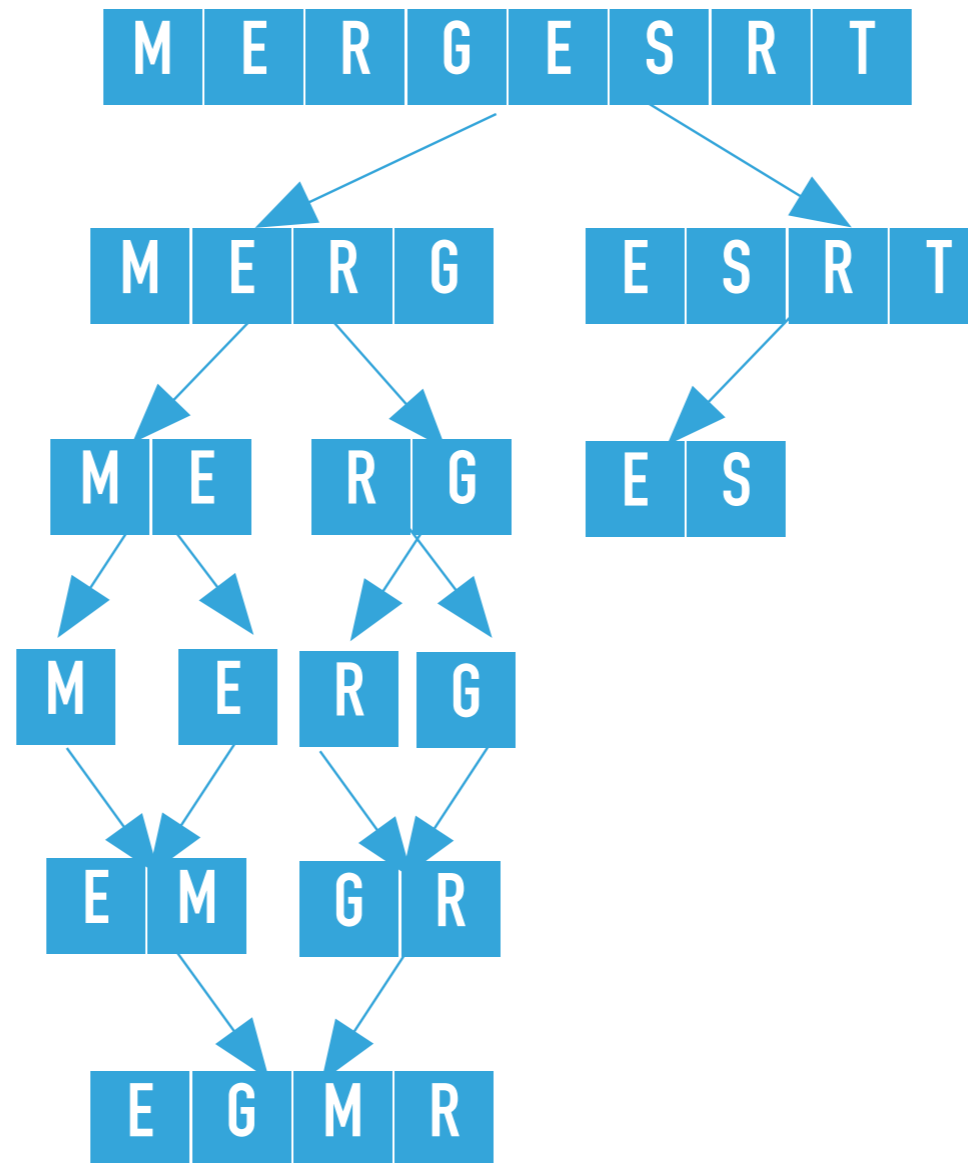


```

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

```

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

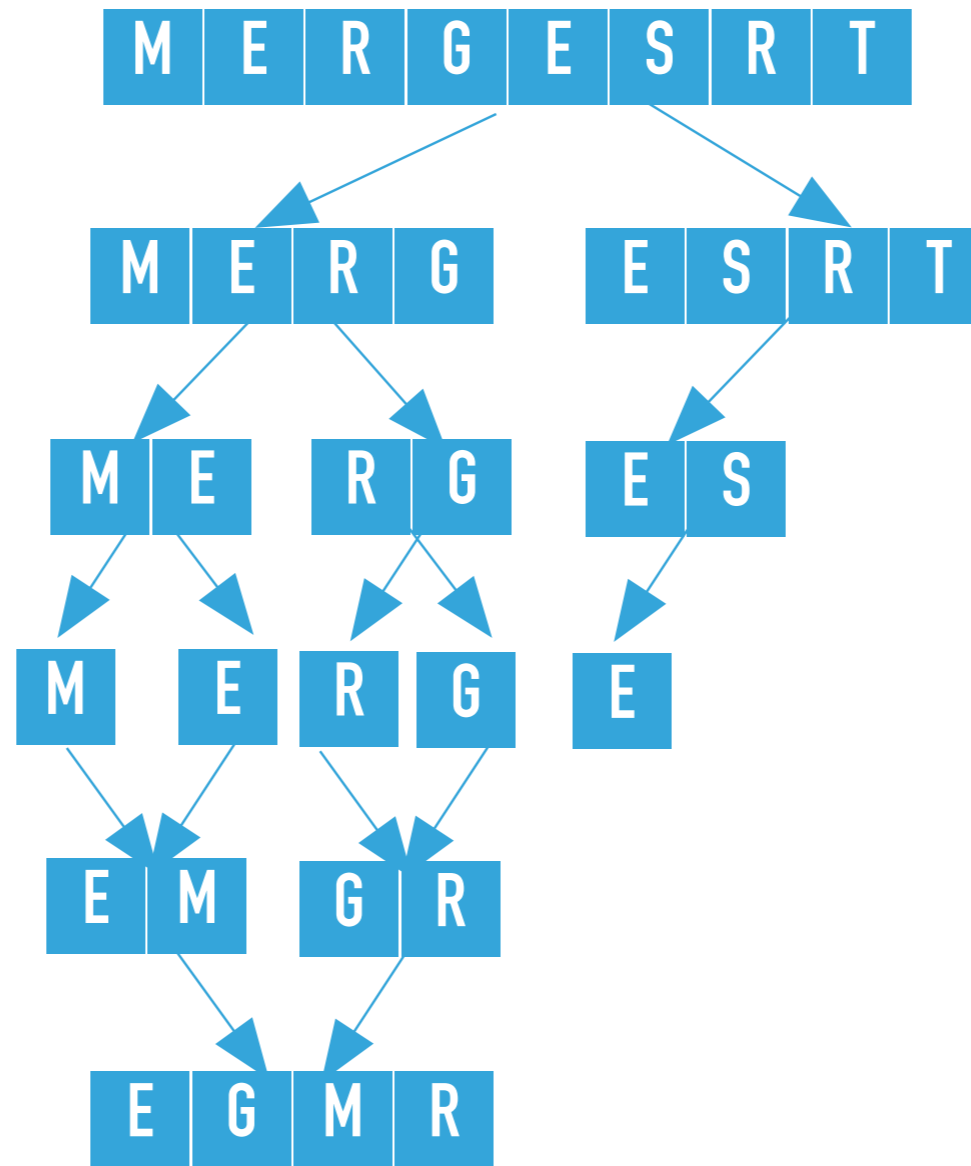


```

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

```

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

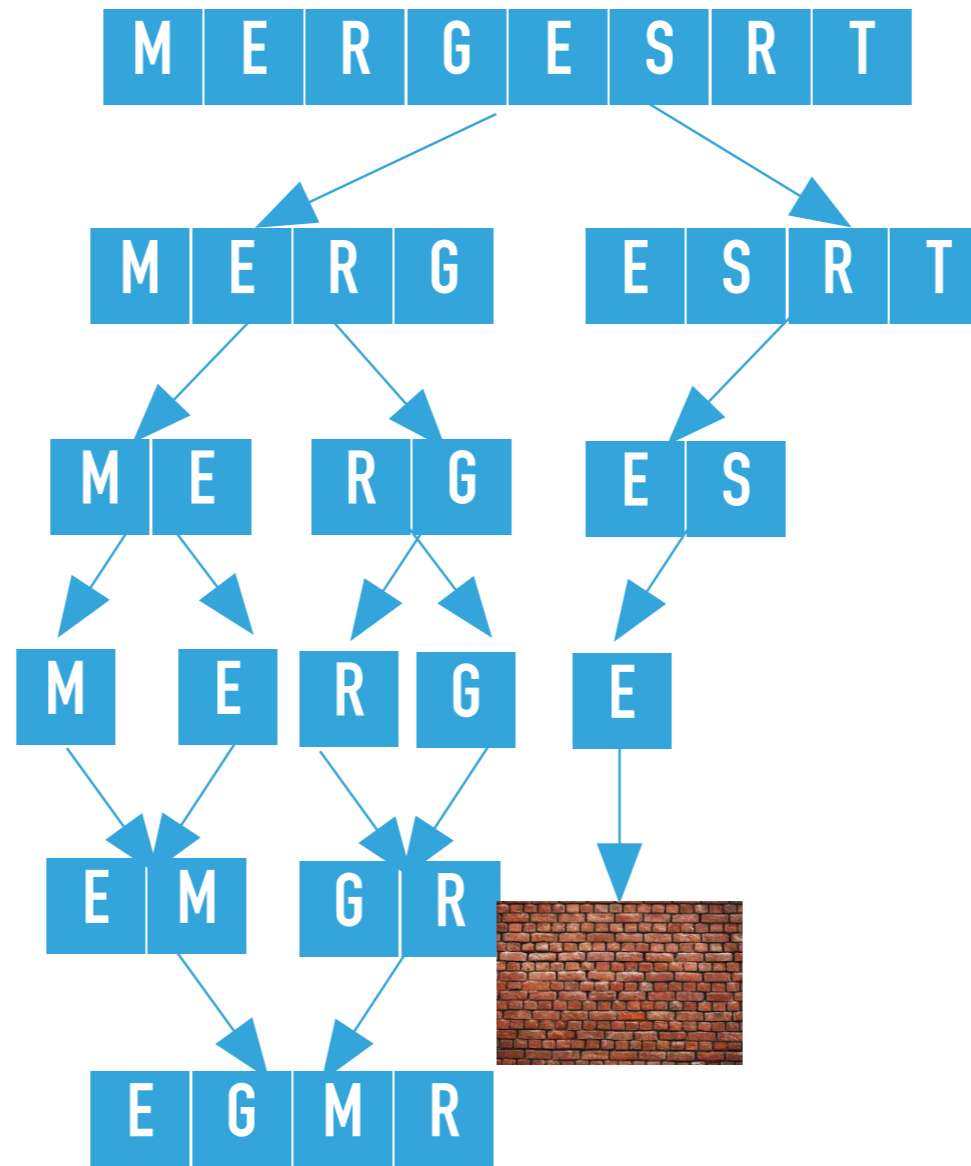


```

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

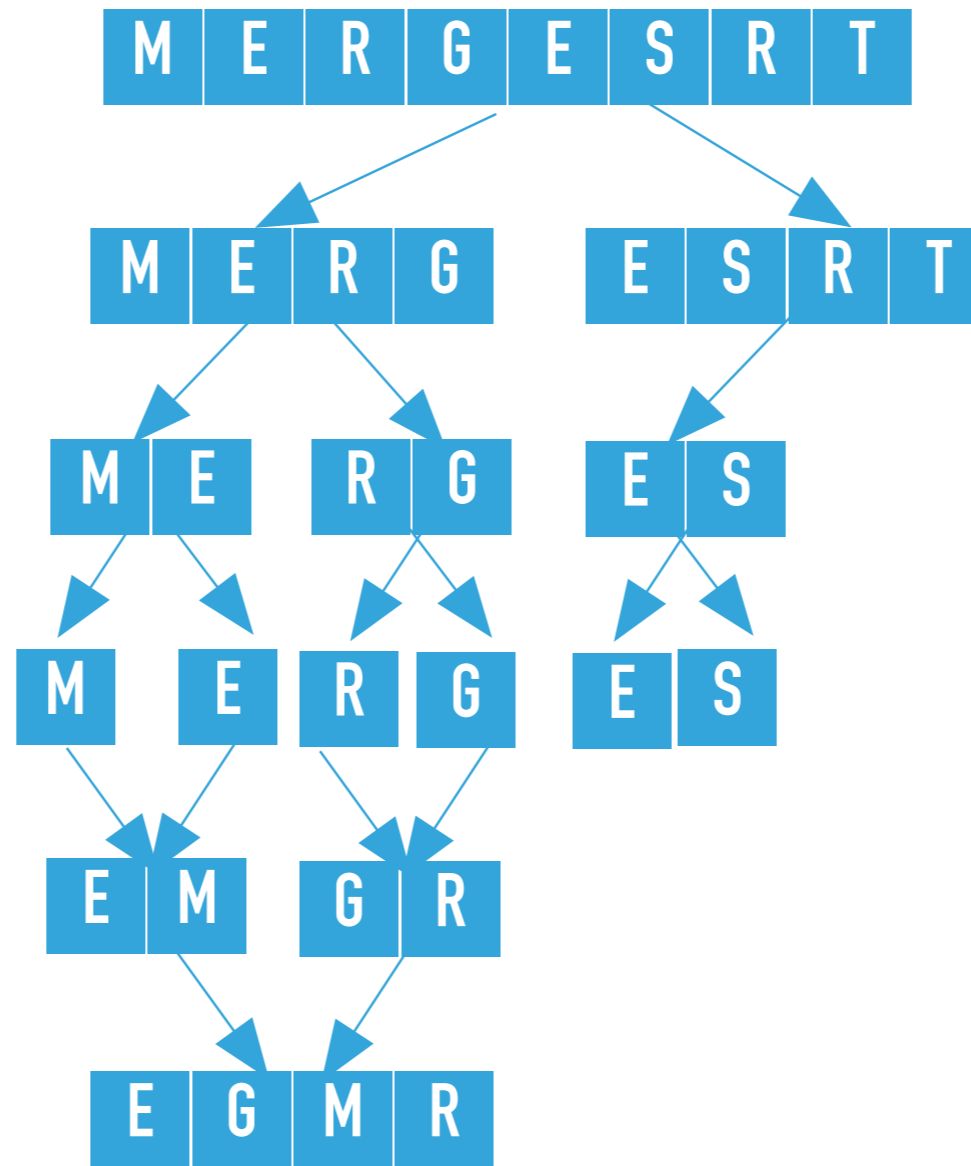
```

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



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

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

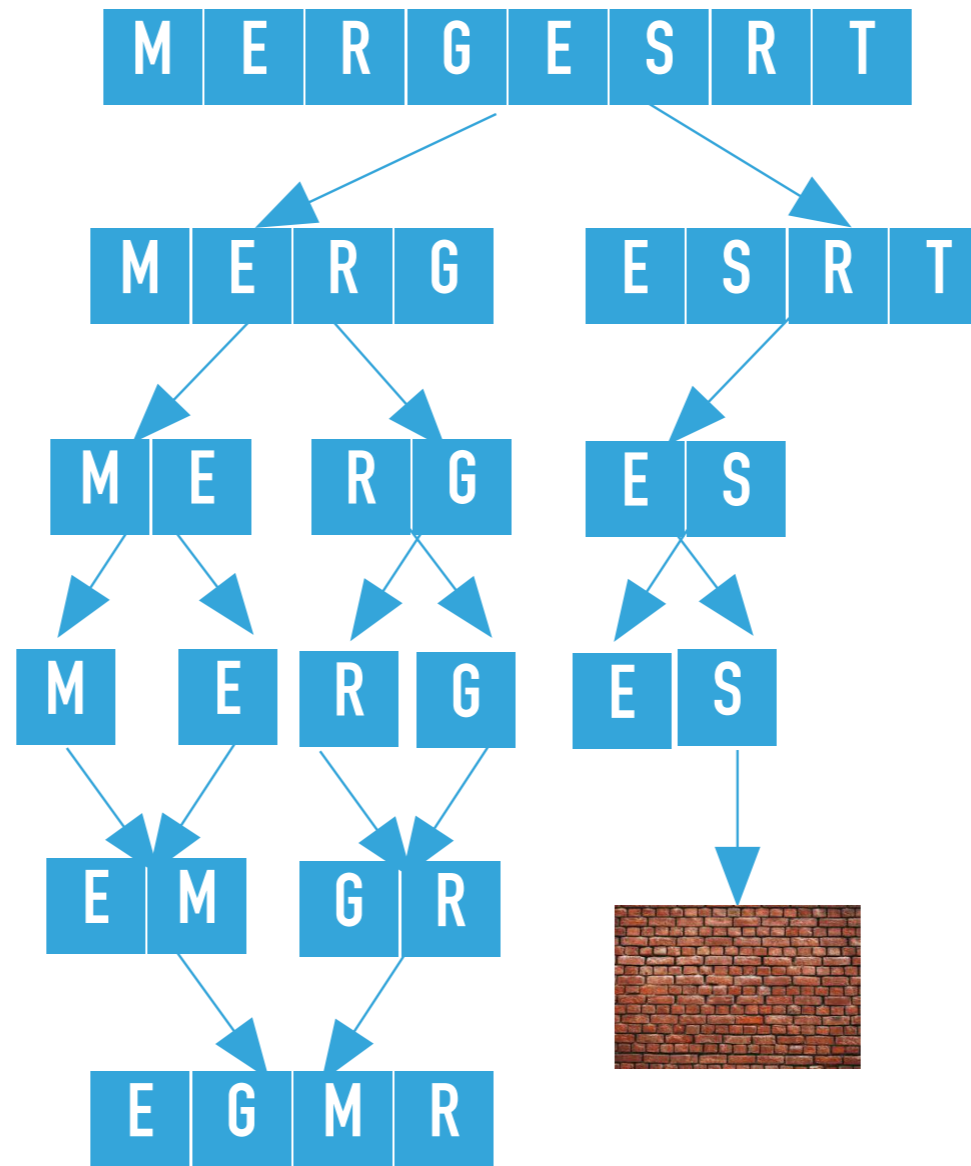


```

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

```

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

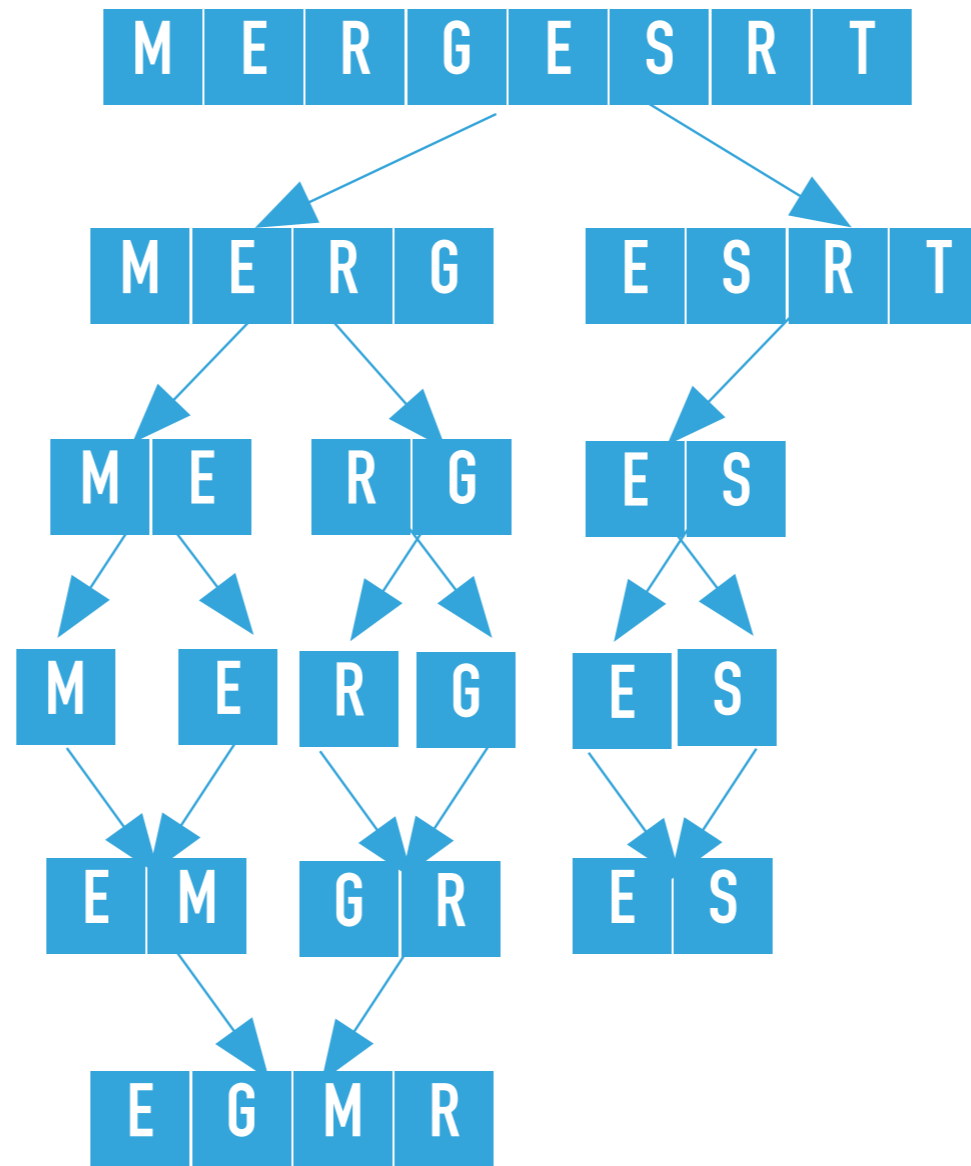


```

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

```

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

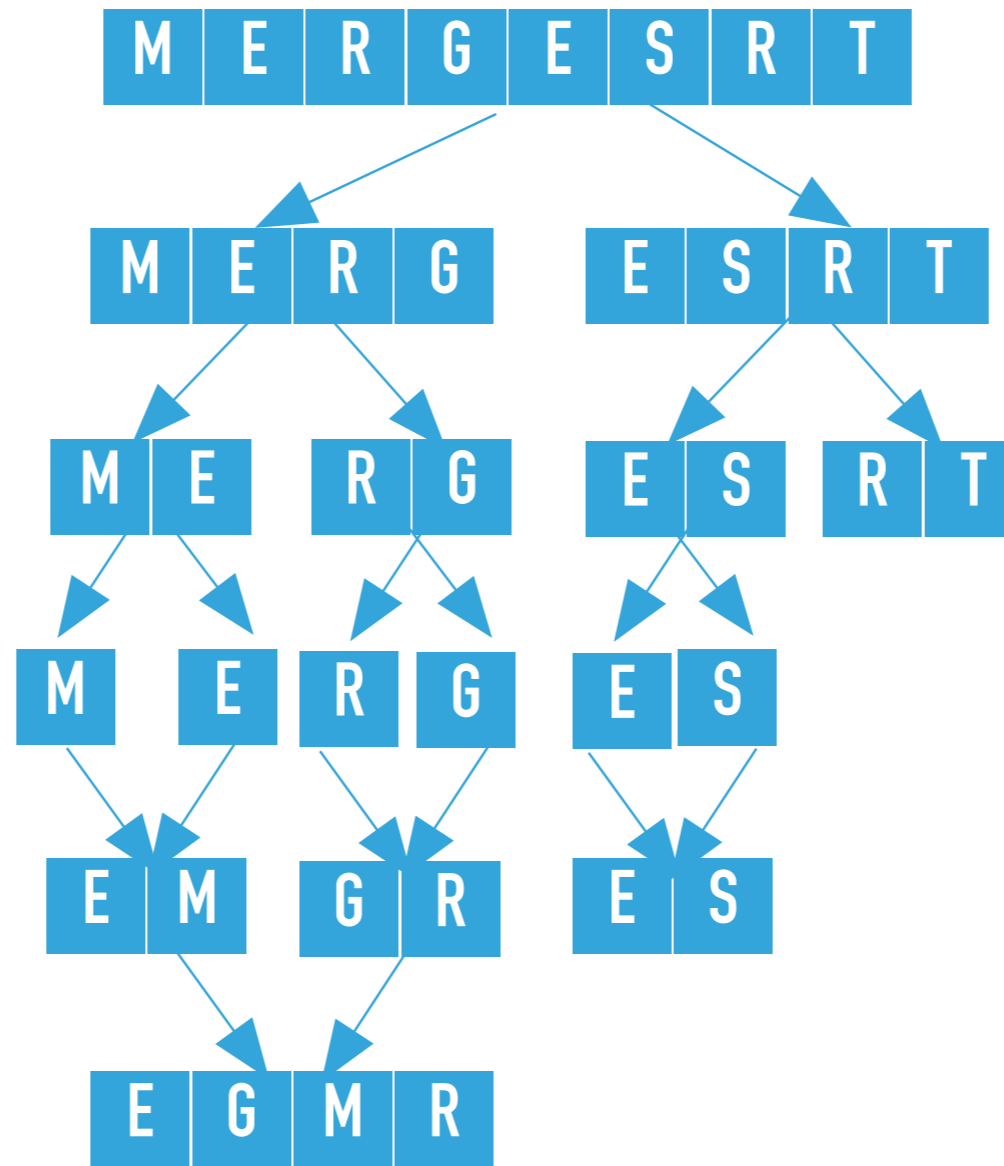


```

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

```

`sort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` 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]`.

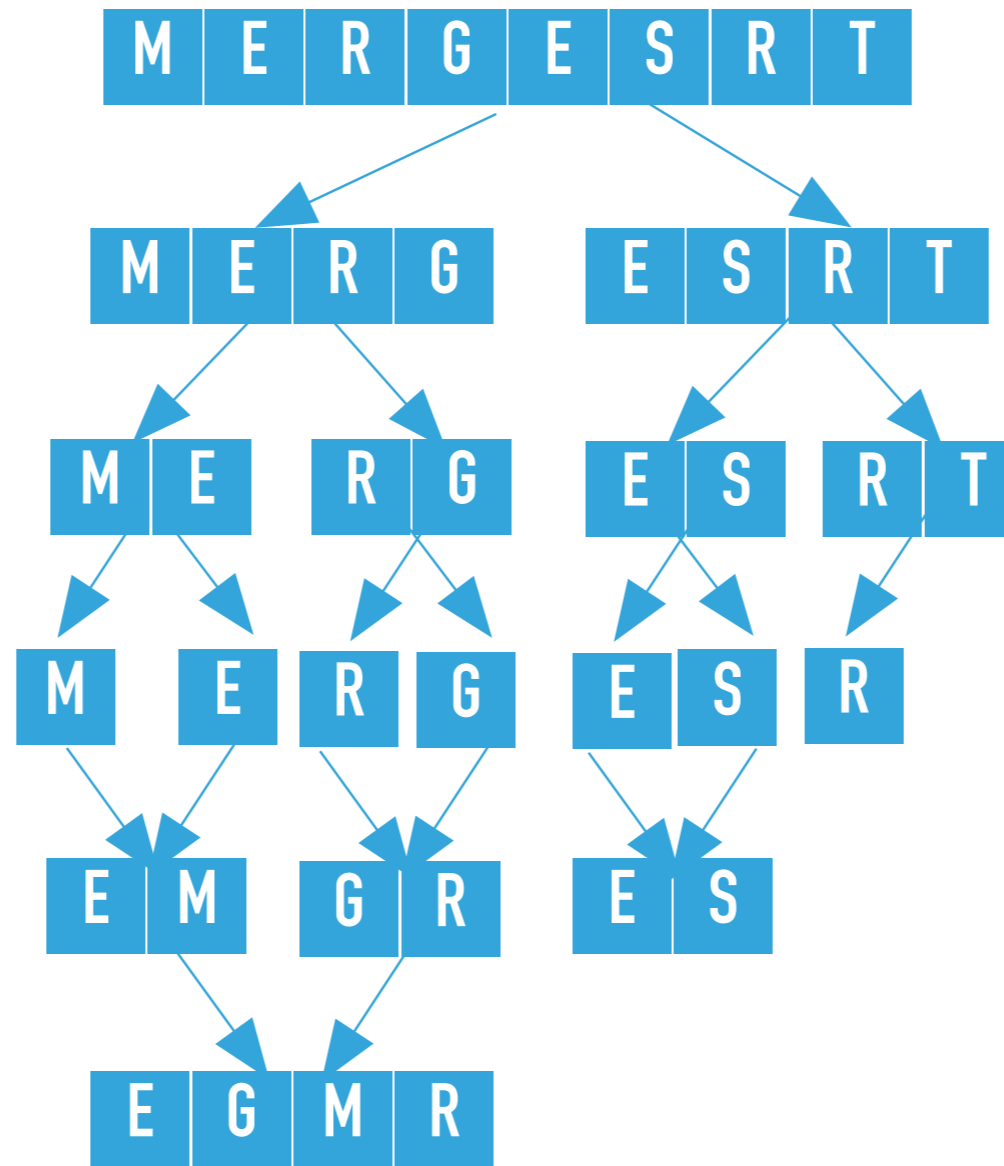


```

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

```

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

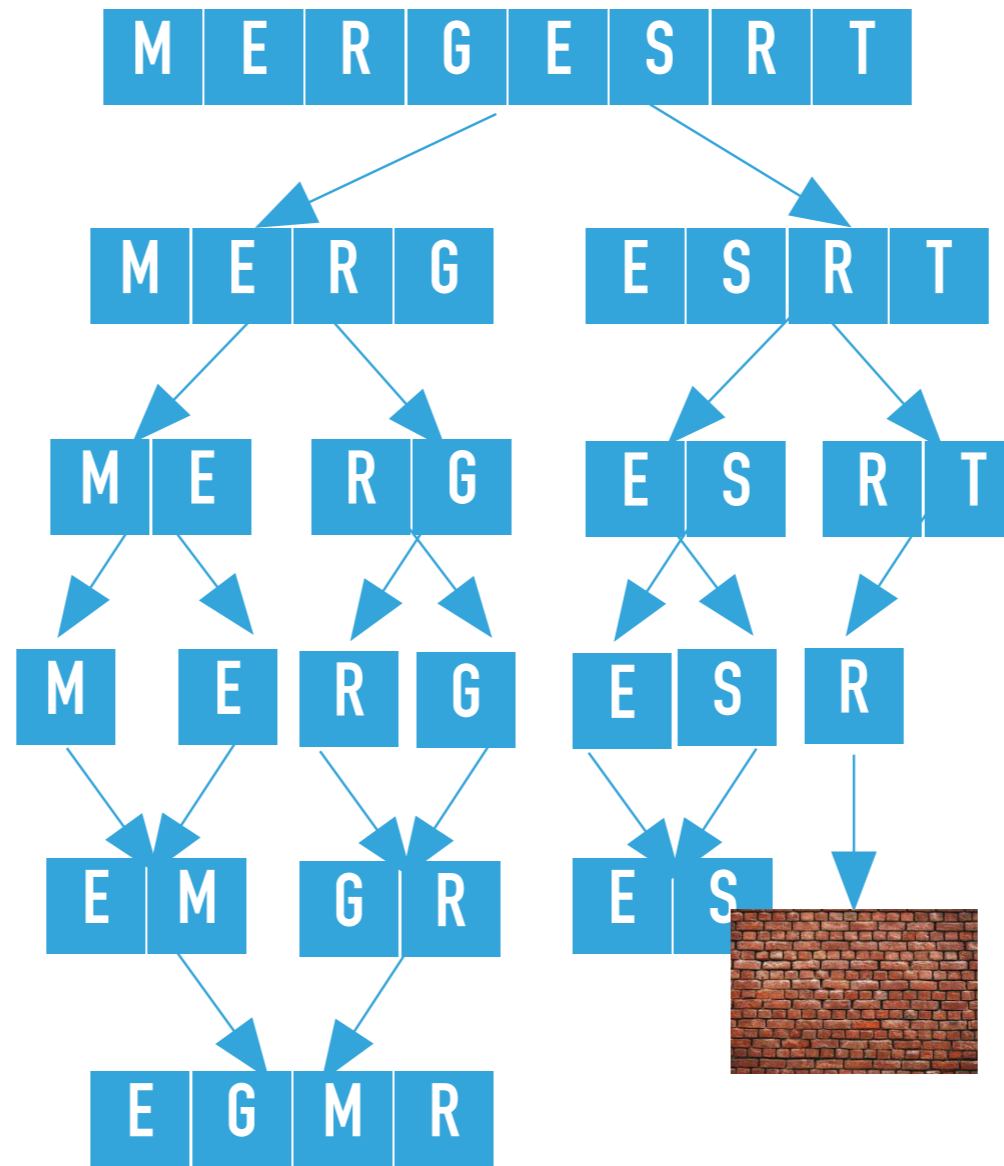


```

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

```

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

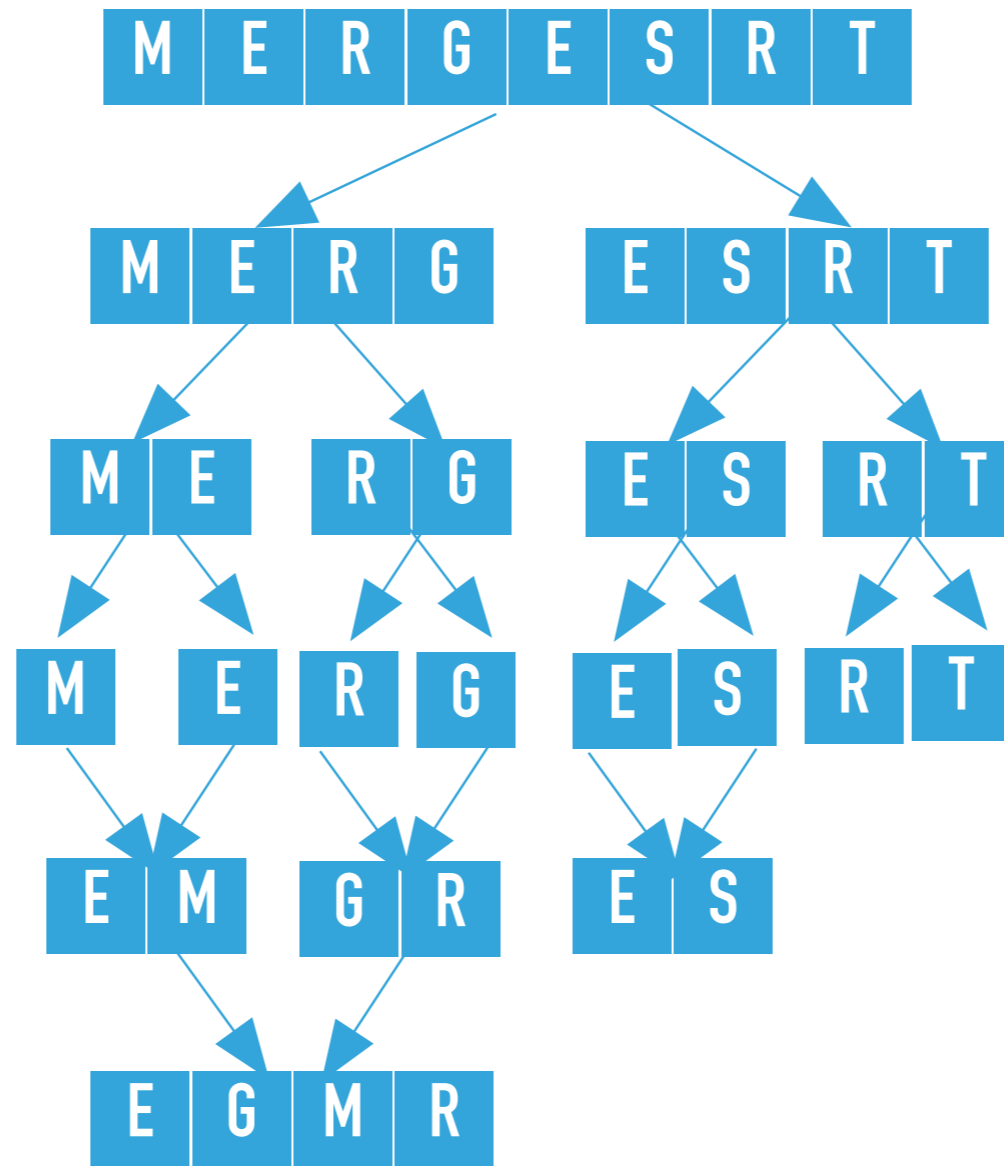


```

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

```

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

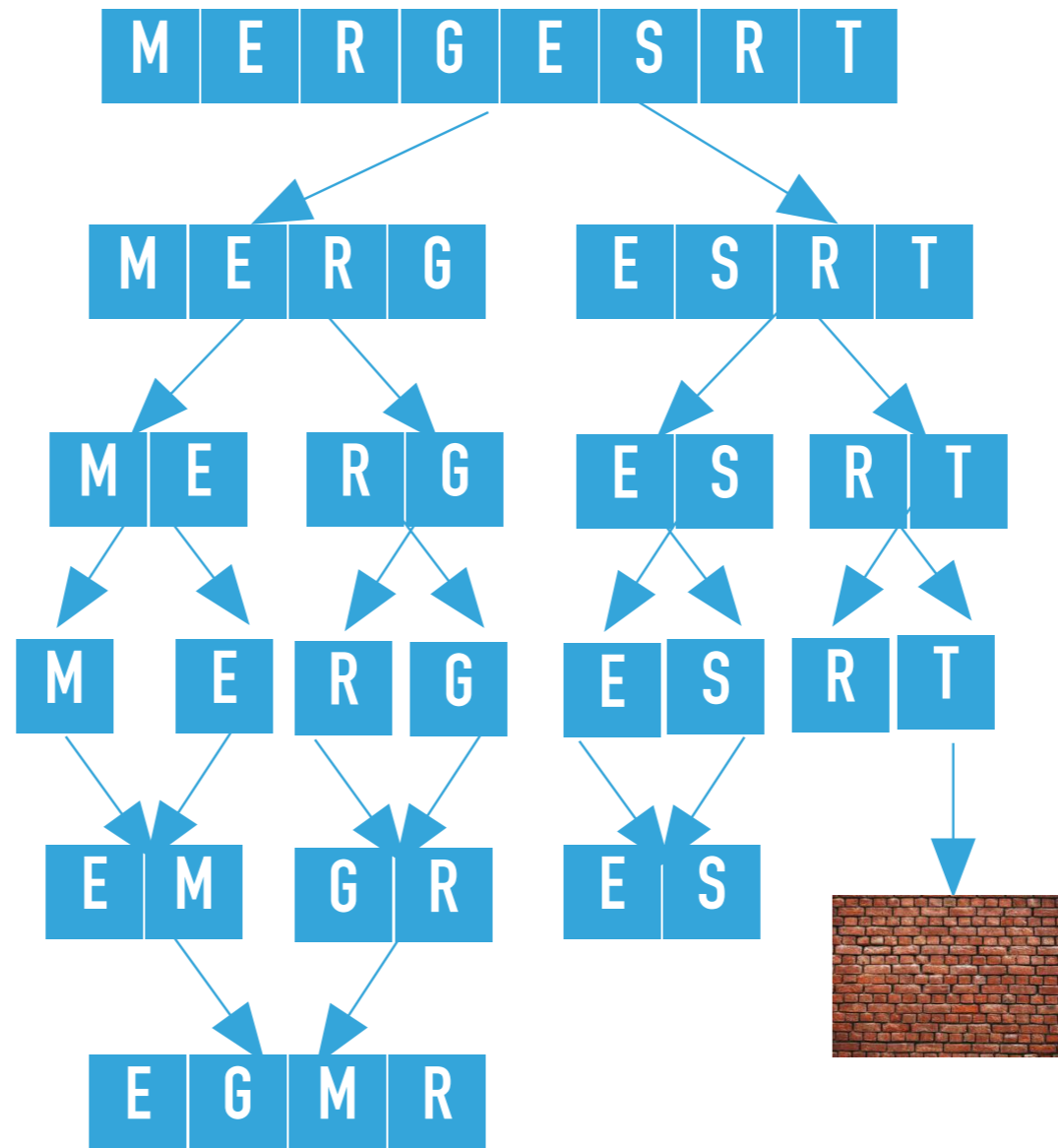


```

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

```

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

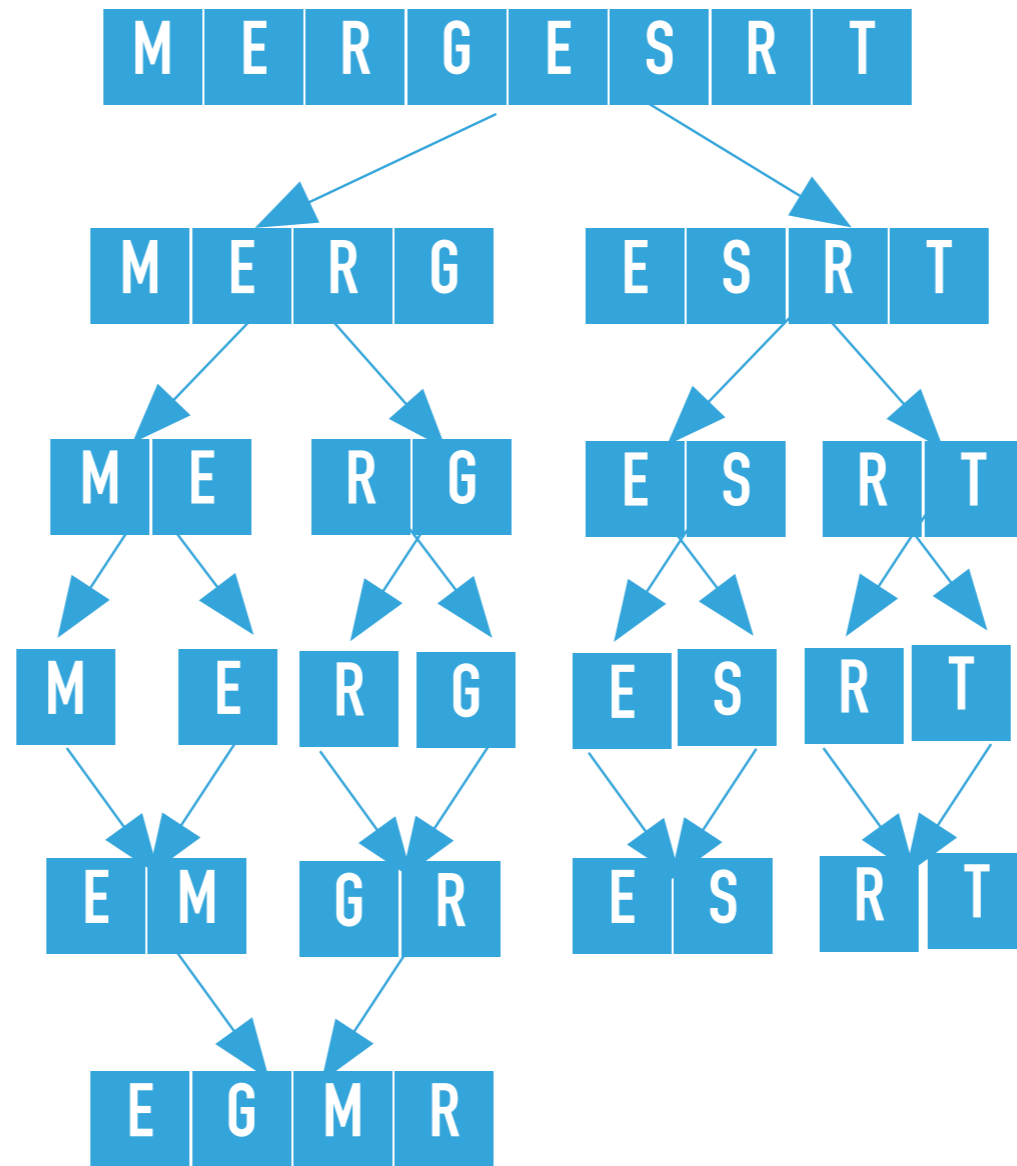


```

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

```

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

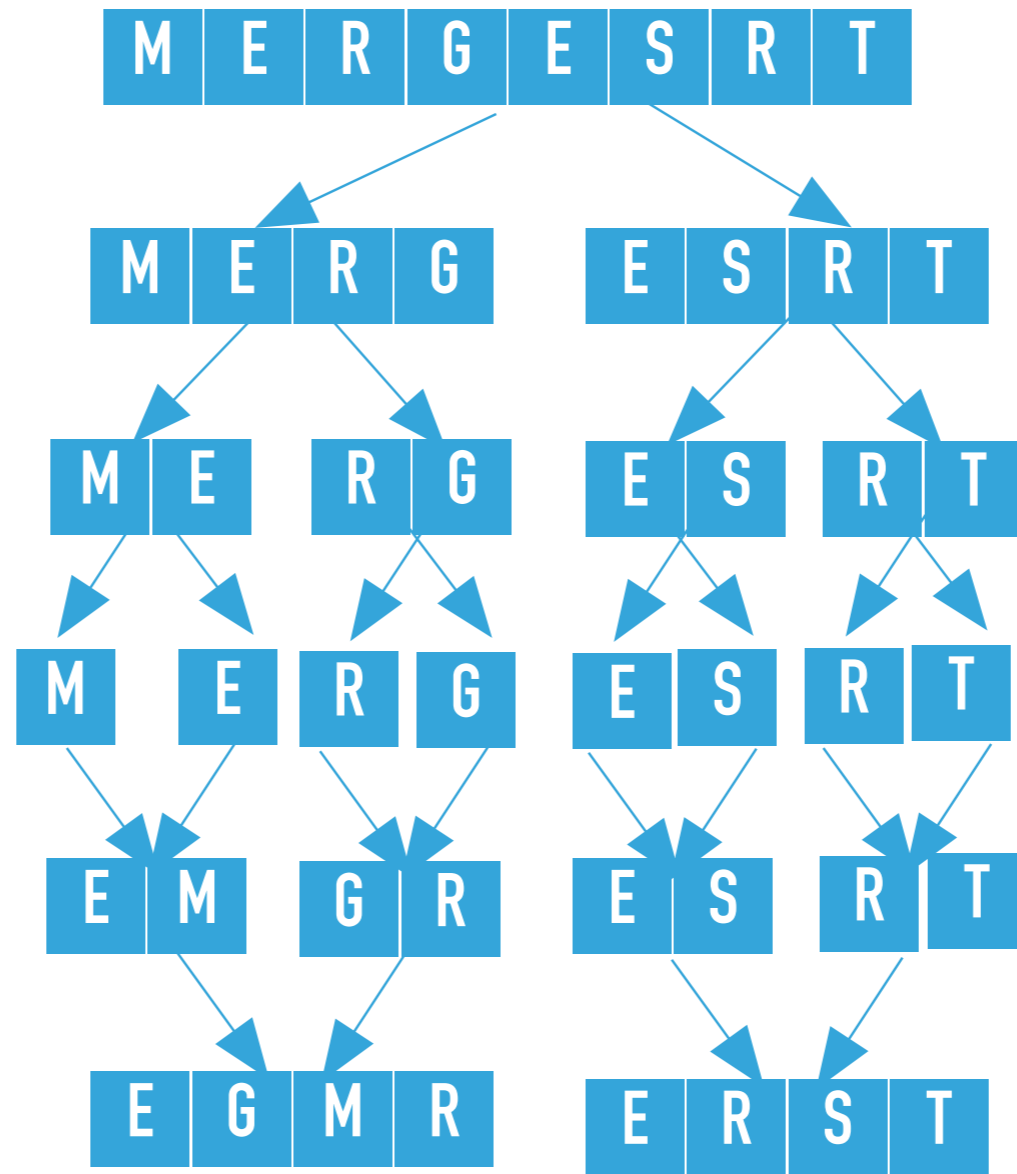


```

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

```

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

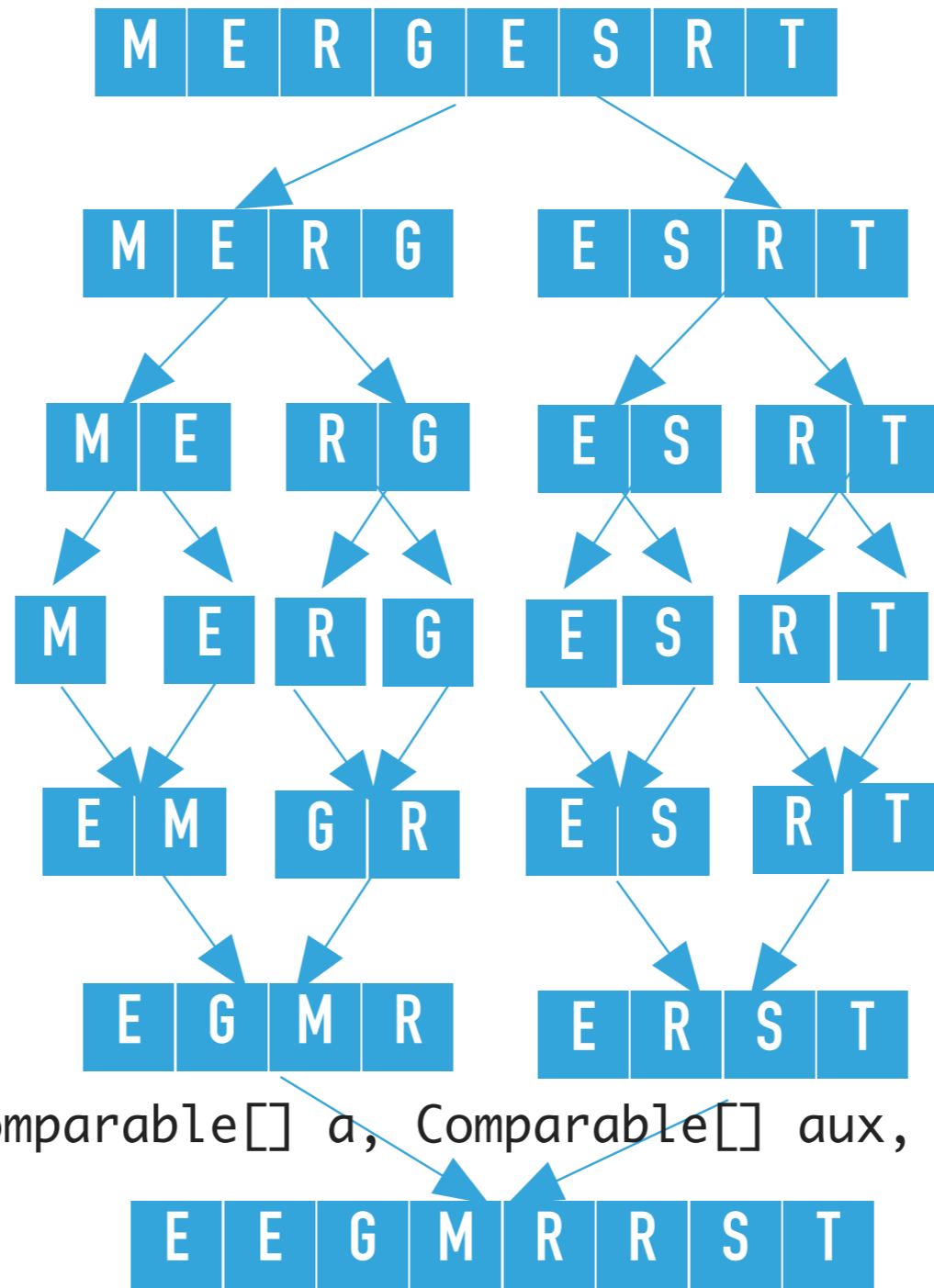


```

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

```

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



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

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

Practice time

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

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

B. { 2, 4, 6, 8, 10 }

C. { 1, 2, 5, 10 }

D. { 1, 2, 3, 4, 5, 10 }

Answer

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

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

Good algorithms are better than supercomputers

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

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

Analysis of comparisons

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

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

If $n = 4$, 2 levels

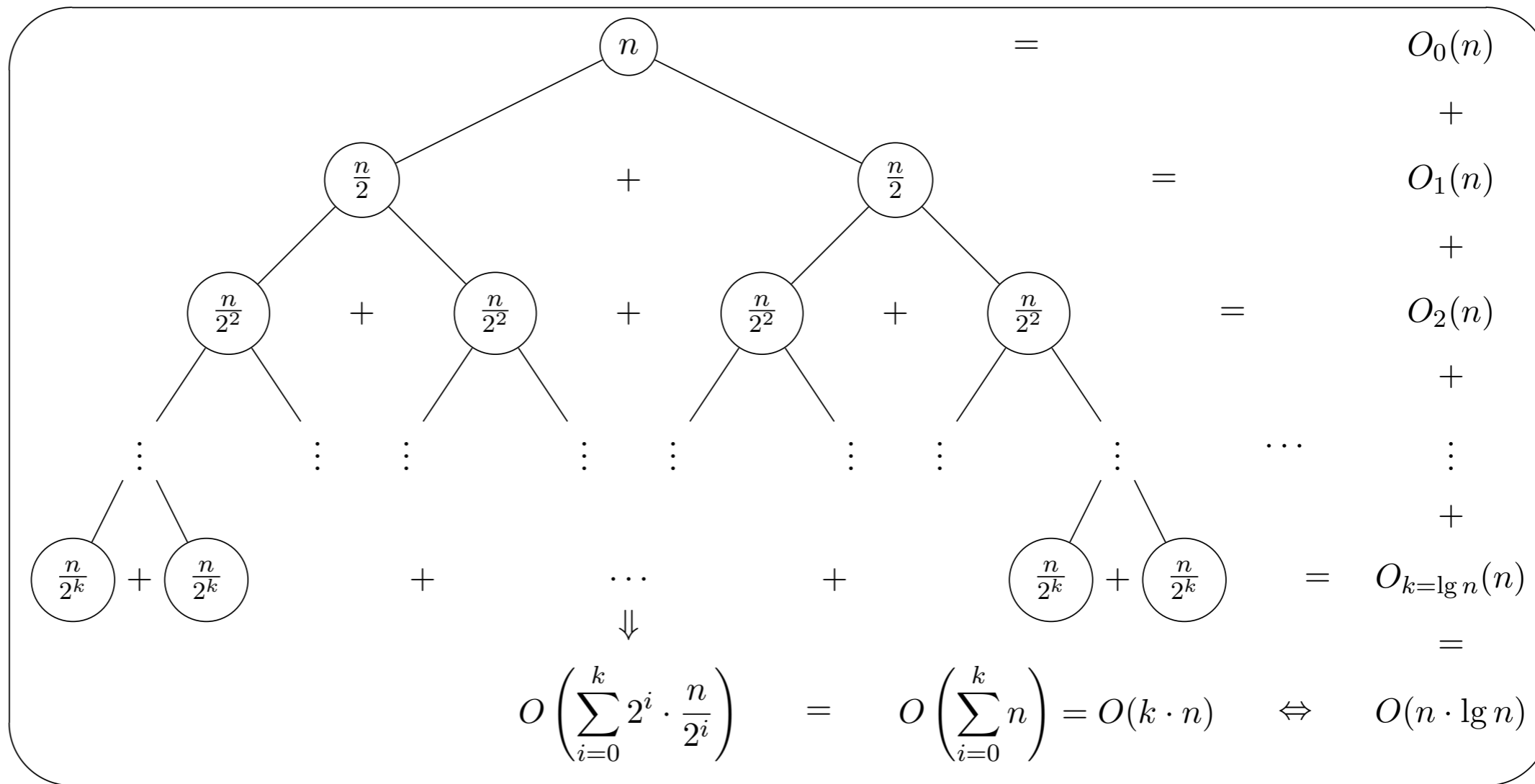
If $n = 8$, 3 levels

If $n = 16$, 4 levels

...

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

or $k = \log_2 n$



(log n levels) x (n comparisons) = O(n log n)

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

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

Mergesort basics

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

Practical improvements for Mergesort

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

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

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

Sorting: the story so far

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

Lecture 13: Mergesort

- ▶ Mergesort

Readings:

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

Practice Problems:

- ▶ 2.2.1-2.2.2, 2.2.11

Readings:

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

Practice Problems:

- ▶ 2.1.1-2.1.8