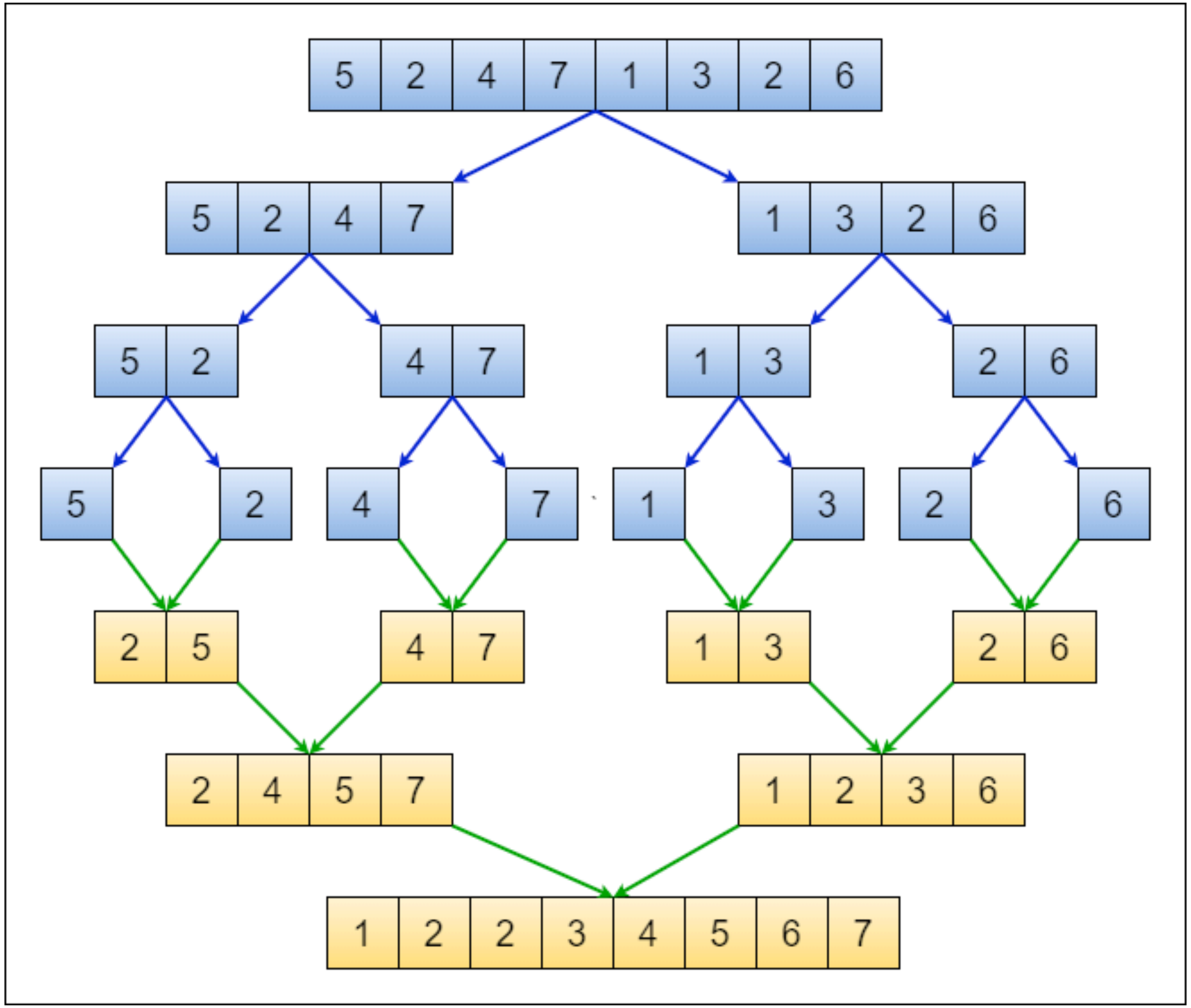
CS62 Class 13: Mergesort

Sorting



Agenda

- Mergesort basics
- Merge walkthrough
- Merge**sort** walkthrough
- Mergesort analysis

Mergesort basics

Basics

input	M	E	R	G	E	S	R	T
sort left half	E	G	M	R	E	S	R	T
sort right half	E	G	M	R	E	R	S	T
merge results	E	E	G	M	R	R	S	T

- Invented by John von Neumann in 1945
  - "Fun" fact: von Neumann was a key player in the Manhattan Project & super influential in the DoD. He was also considered a child prodigy
- Algorithm sketch:
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge the two halves

Mergesort visualization

Merge Sort in 3

Mergesort: the quintessential example of divide-and-conquer

review: a static *generic* method

public is what users call (we are sorting an array)

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

auxiliary array to hold the in-progress sorting

can't have generic (E) array types, so we declare it an array of Comparables, and then we cast it to type E[]

private is the recursive helper method

```
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);
}
```

We do not calculate mid as (hi+lo)/2 as this may overflow. Instead we use lo+(hi-lo)/2, with lo and hi being positive integers within range, and lo<=hi.

two recursive calls for the halves

finally, *aux* will be partially sorted (its left half and its right half will be sorted), so we can merge the halves together into *a*

Merging two already sorted halves into one sorted array

the sorting actually happens in the merge call: we sort a in place while comparing elems in aux

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

k is an index through a

i & j are indices through aux

elem@**j** in aux is smaller, so put the elem@**j** in a & increment j ptr

elem@**i** in aux is smaller (or equal), so put the elem@**i** in a & increment i ptr

Merge walkthrough

Example – Merging two sorted subarrays

- Let's assume that somehow we have already sorted the two halves of the array
  { "M", "E", "R", "G", "E", "S", "R", "T"} into
  { "E", "G", "M", "R", "E", "R", "S", "T"}
  0 1 2 3 4 5 6 7
  - Note that from index lo = 0 to mid = 3, the array is sorted. The same applies from mid+1 = 4 to hi = 7.
- We will now see how we can merge together these two sorted halves into one final sorted array.

Merging Example – copying to auxiliary array

Array a

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7

lo = 0

mid = 3

hi = 7

```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, i
    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 suba
            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							
E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi

Array a							
E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7

lo = 0

mid = 3

hi = 7

```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, i
    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 suba
            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

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
i,k				j			

Array a

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7

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

aux[i] equal to aux[j]

a[0]=aux[0]

i++;

Merging Example - k=1

Array aux

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
	i,k			j			

Array a

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7

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

aux[i] larger than aux[j]
a[1]=aux[4]
j++;

Merging Example - k=2

Array aux

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
	i	k			j		

Array a

E	E	M	R	E	R	S	T
0	1	2	3	4	5	6	7

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

aux[i] smaller than aux[j]
a[2]=aux[1]
i++;

Merging Example - k=3

Array aux

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
		i	k		j		

Array a

E	E	G	R	E	R	S	T
0	1	2	3	4	5	6	7

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

```
aux[i] smaller than aux[j]
a[3]=aux[2]
i++;
```

Merging Example - k=4

Array aux

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
			i	k	j		

Array a

E	E	G	M	E	R	S	T
0	1	2	3	4	5	6	7

```
aux[i] equal to aux[j]
a[4]=aux[3]
i++;
```

```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, i
    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 suba
            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

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
				i	k,j		

Array a

E	E	G	M	R	R	S	T
0	1	2	3	4	5	6	7

```
i > mid
a[5]=aux[5]
j++;
```

```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, i
    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 suba
            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

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
				i		k,j	

Array a

E	E	G	M	R	R	S	T
0	1	2	3	4	5	6	7

```
i > mid
a[6]=aux[6]
j++;
```

```
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, i
    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 suba
            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

E	G	M	R	E	R	S	T
0	1	2	3	4	5	6	7
lo			mid				hi
				i			k,j

Array a

E	E	G	M	R	R	S	T
0	1	2	3	4	5	6	7

```
i > mid
a[7]=aux[7]
j++;
```

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

http://algs4.cs.princeton.edu

2.2 MERGING DEMO

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

Worksheet 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?

Hint: think of a best and worst case example and work through the code

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

B. $\sim 1/2n$

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

D. $\sim n$

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

Worksheet 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. $\sim 1/2n$ 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 walkthrough

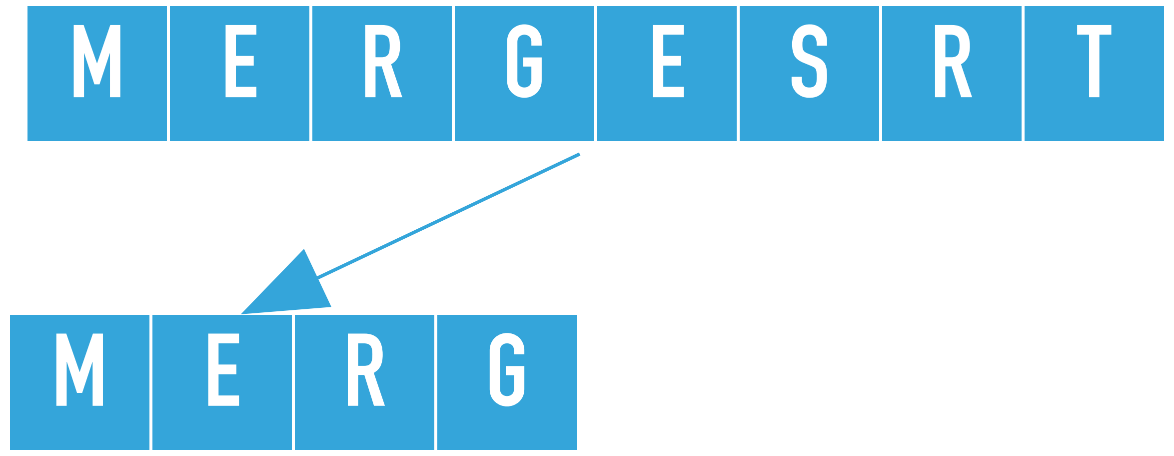
Mergesort: the quintessential example of divide-and-conquer

```
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);
}
```

M	E	R	G	E	S	R	T

```java
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);
}
```

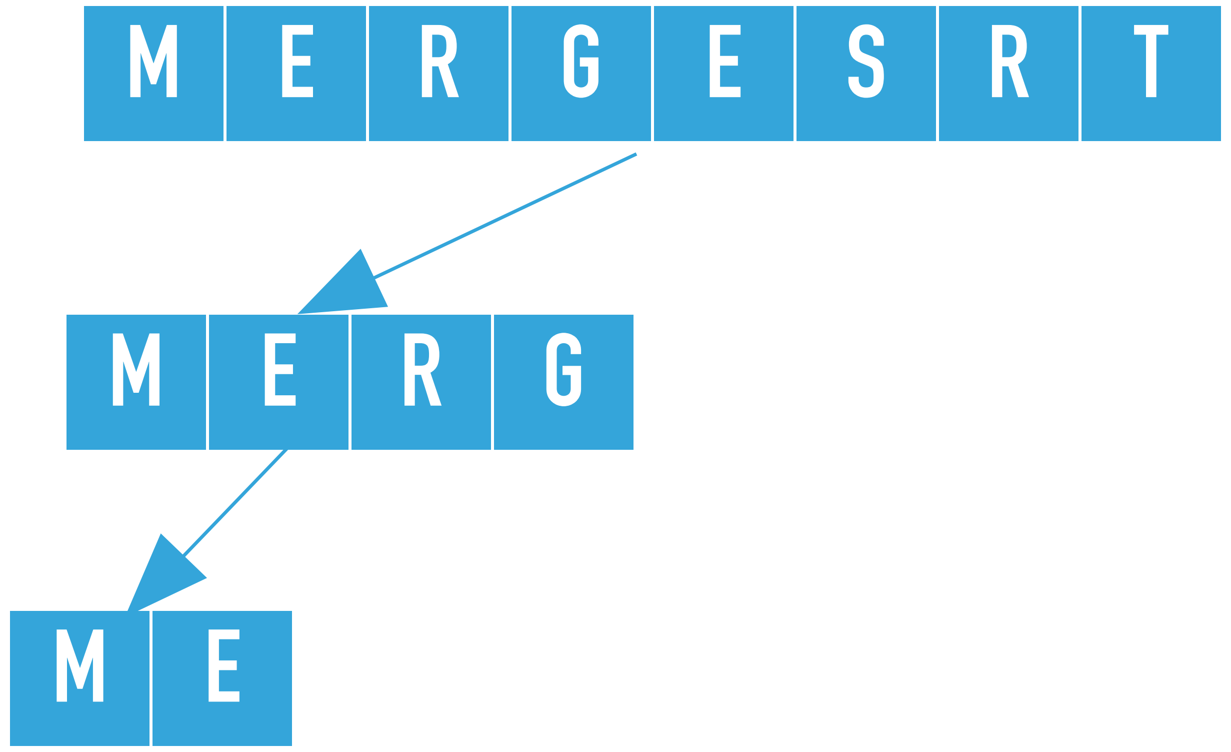
`mergeSort([M, E, R, G, E, S, R, T])` calls `mergeSort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null], 0, 7)` where the array of `nulls` is the auxiliary array, `lo = 0` and `hi = 7`.




```
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);
}
```

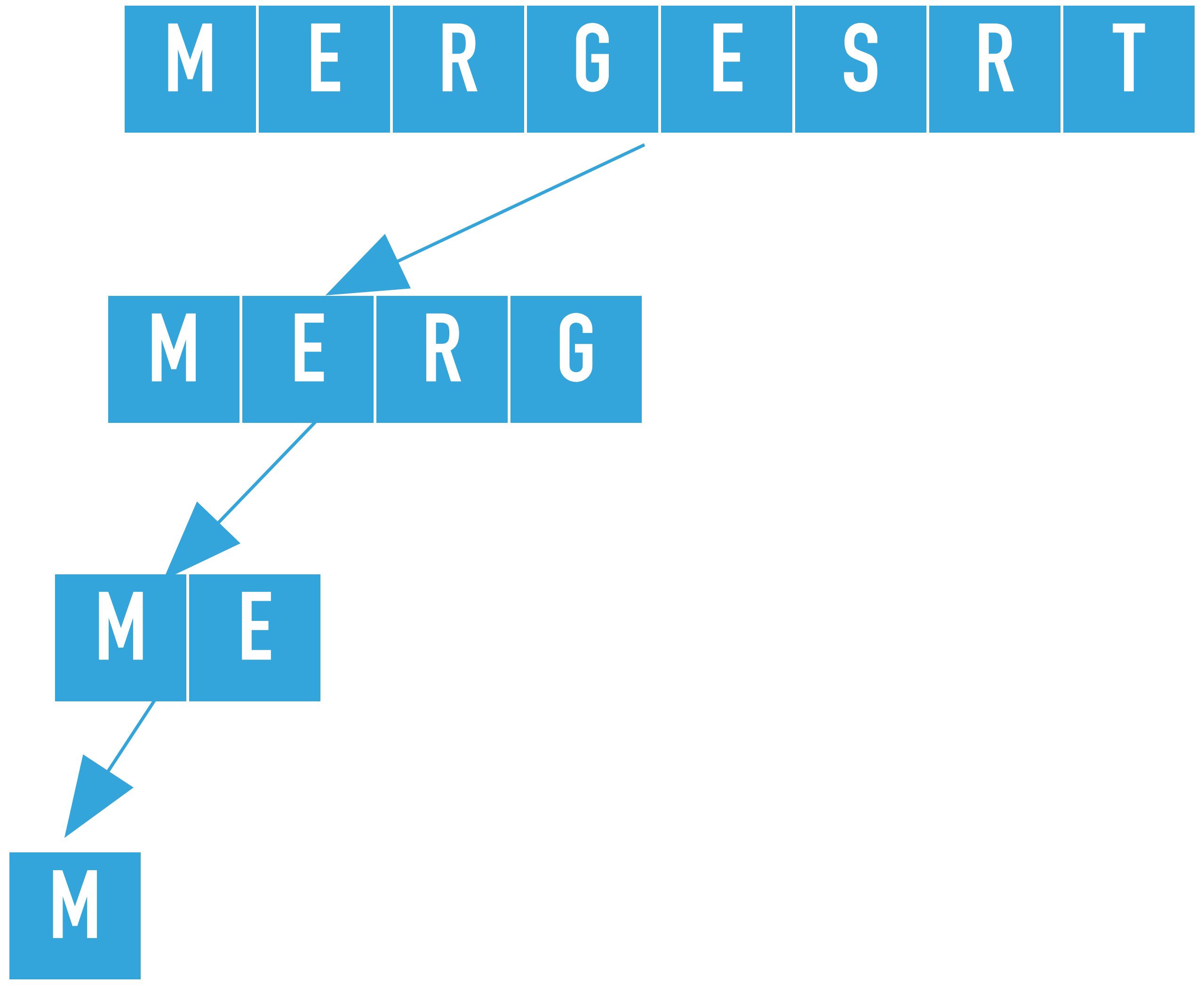
notice that only mid and lo and hi are changing in these recursive calls (not a or aux)

`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`



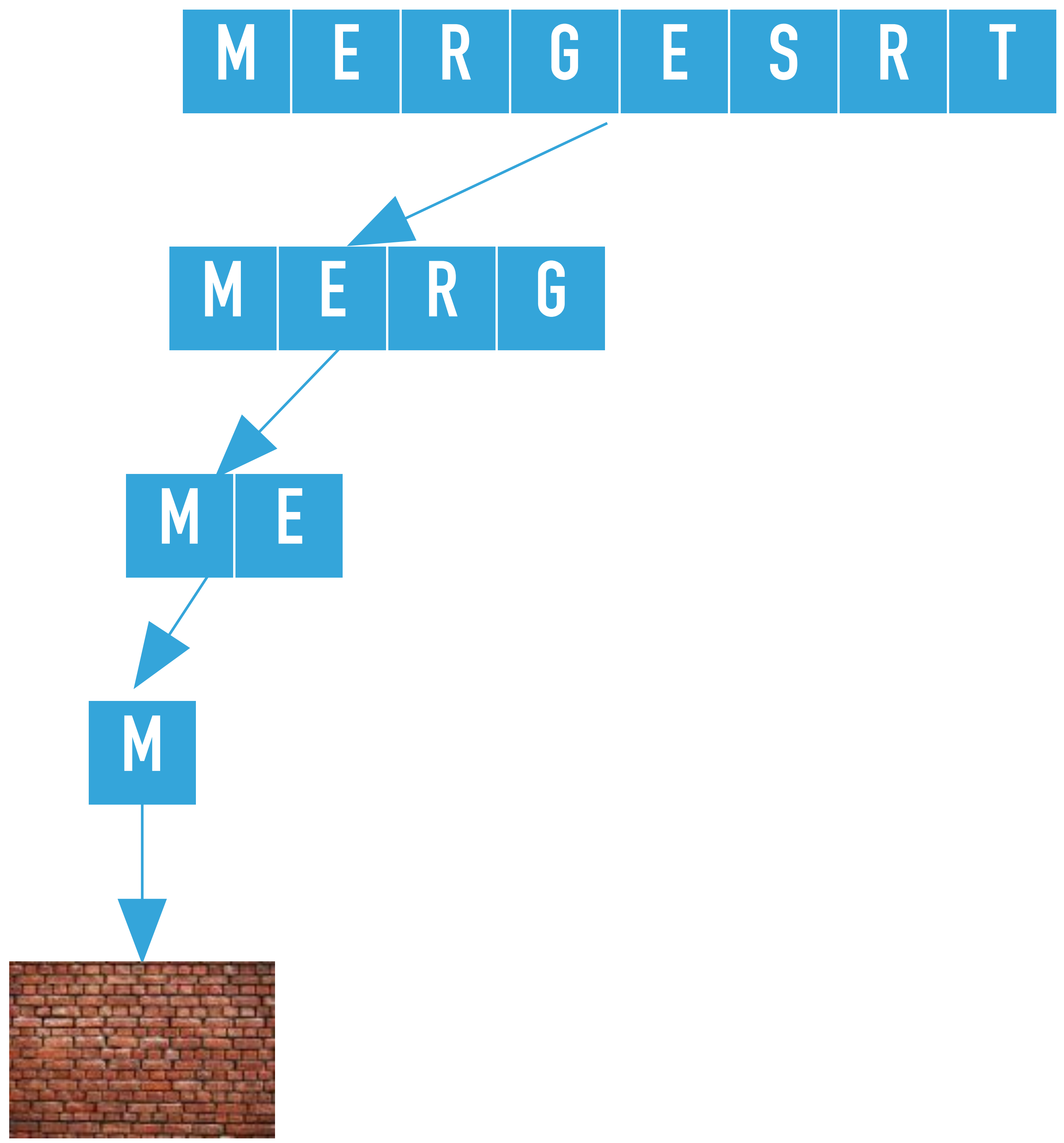

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



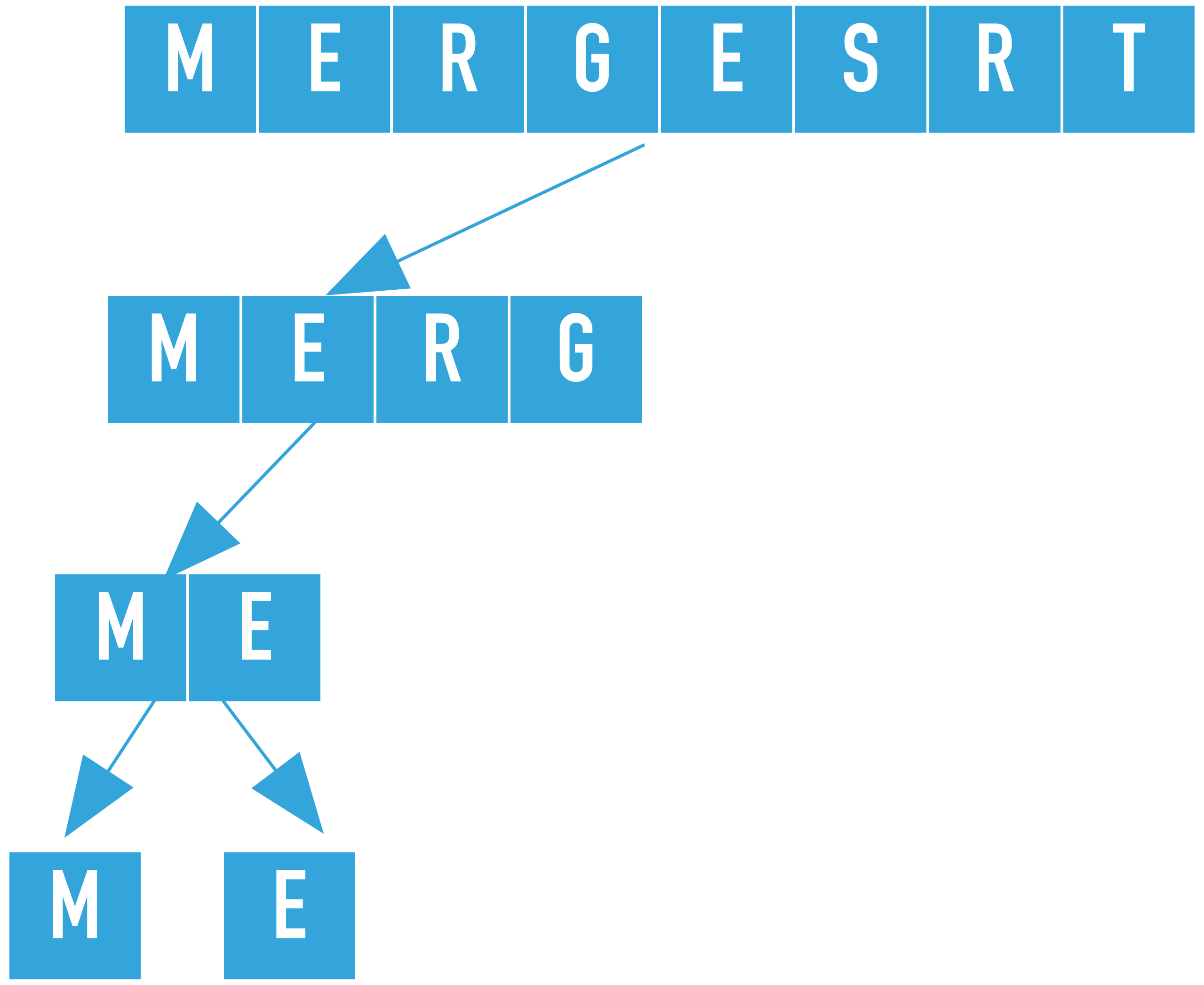

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



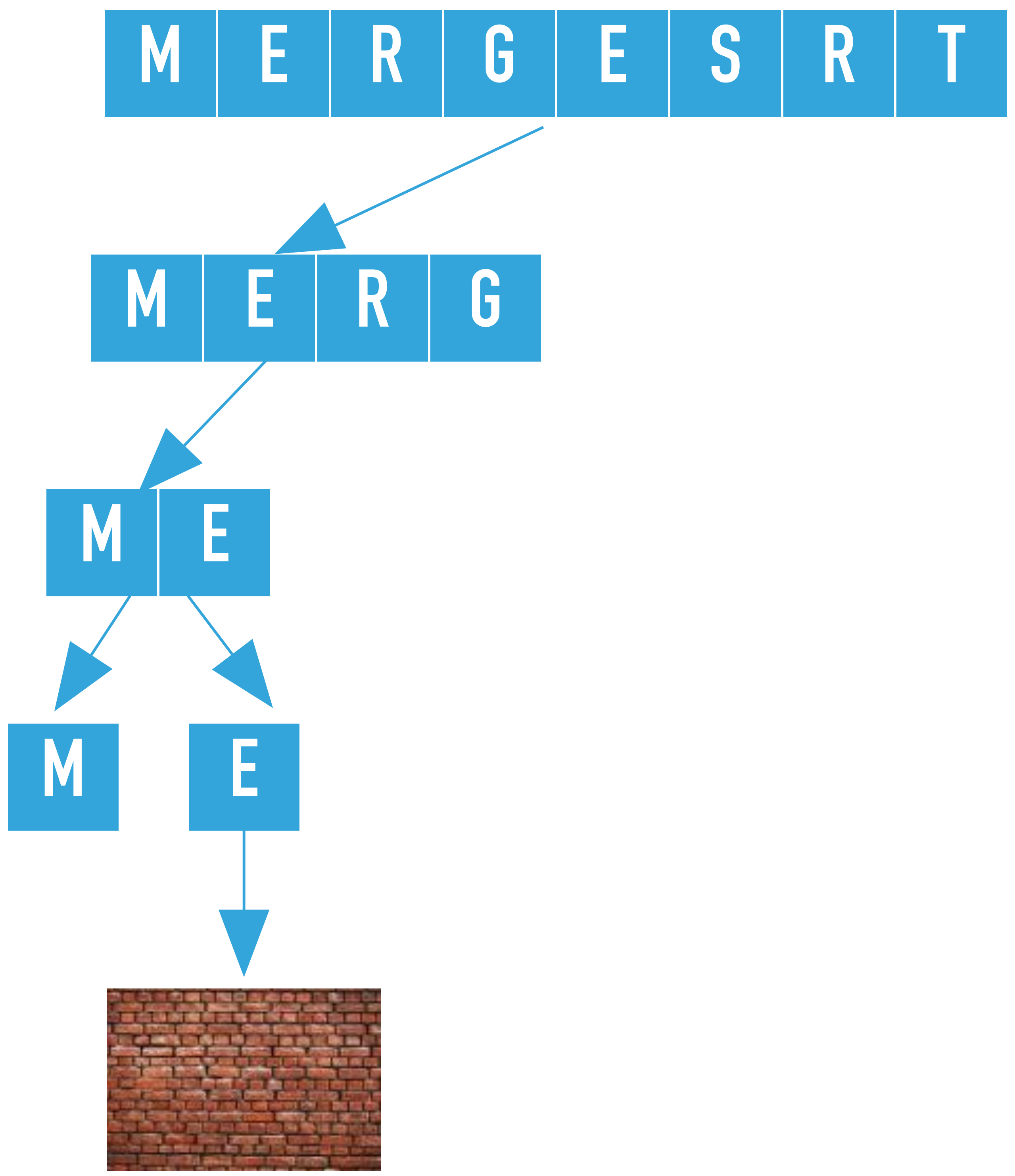

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



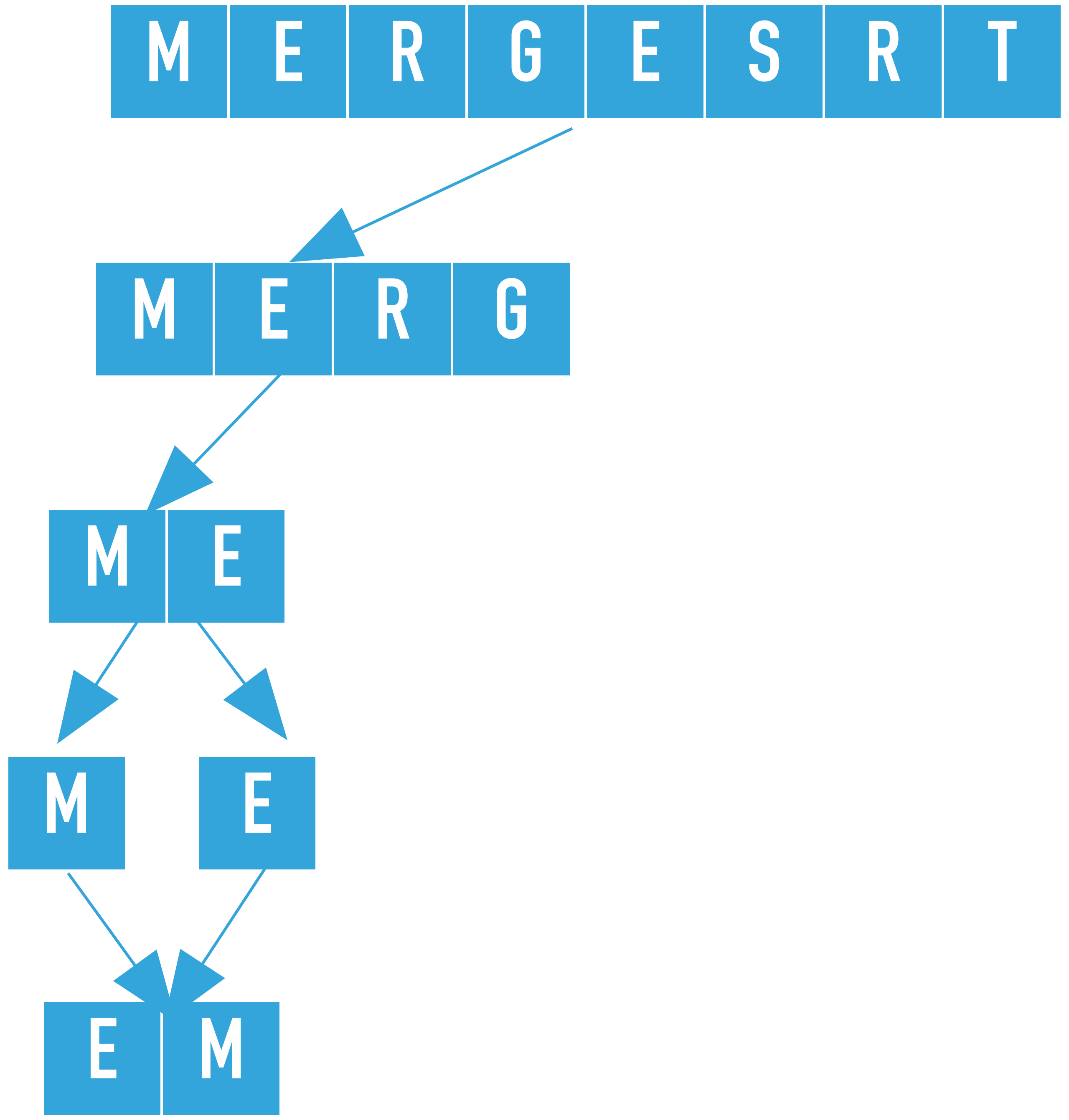

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



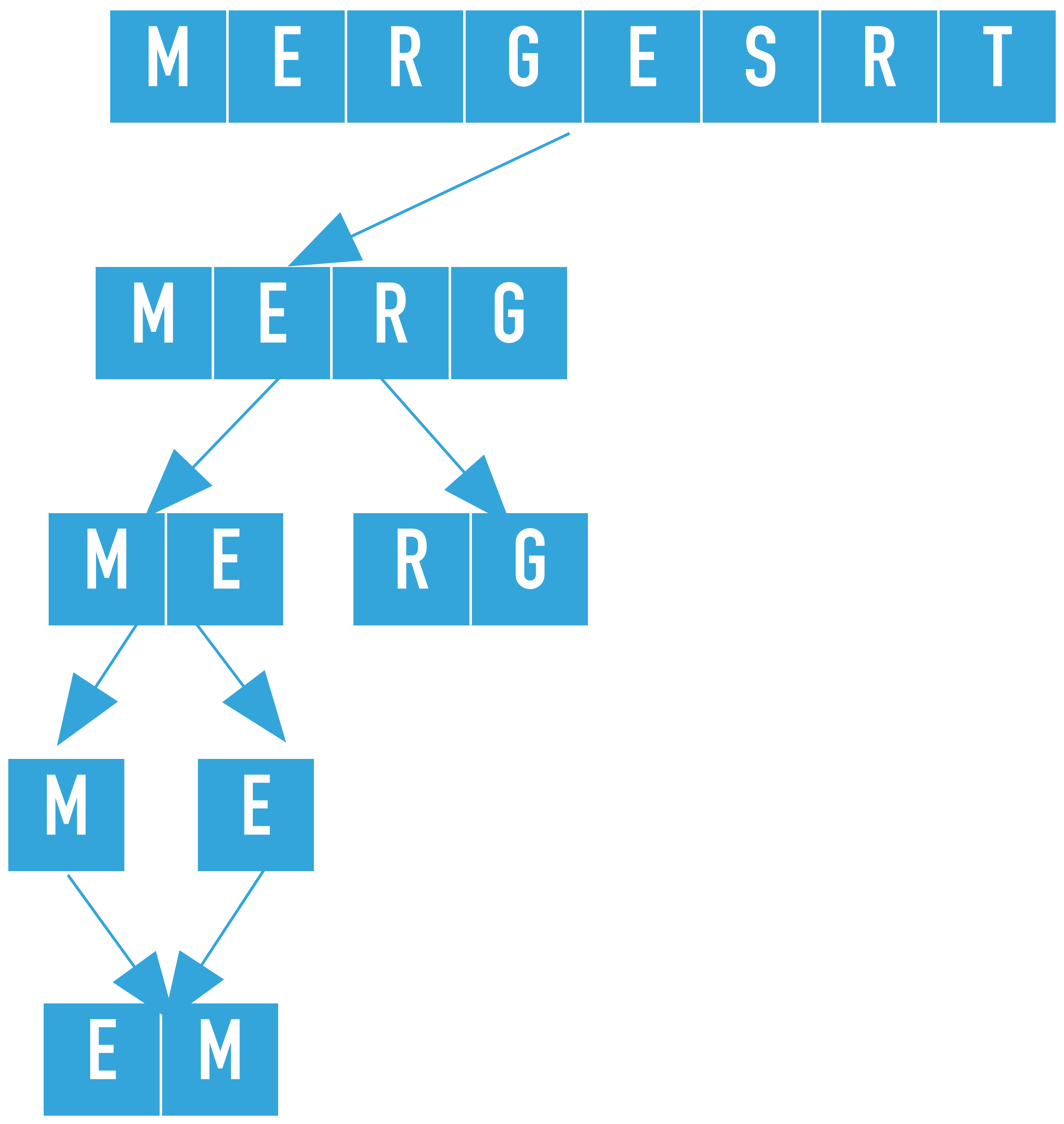

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



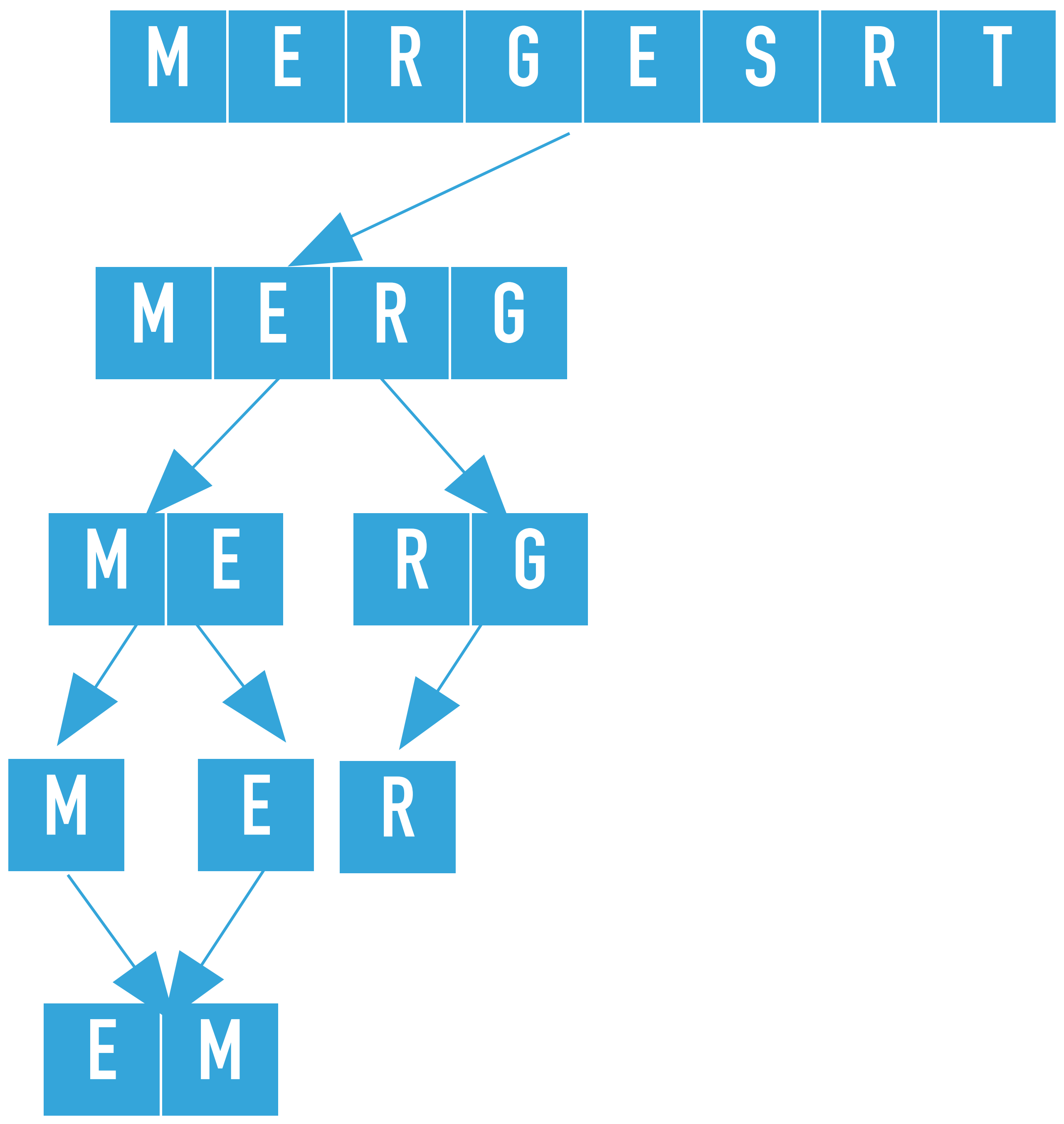

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



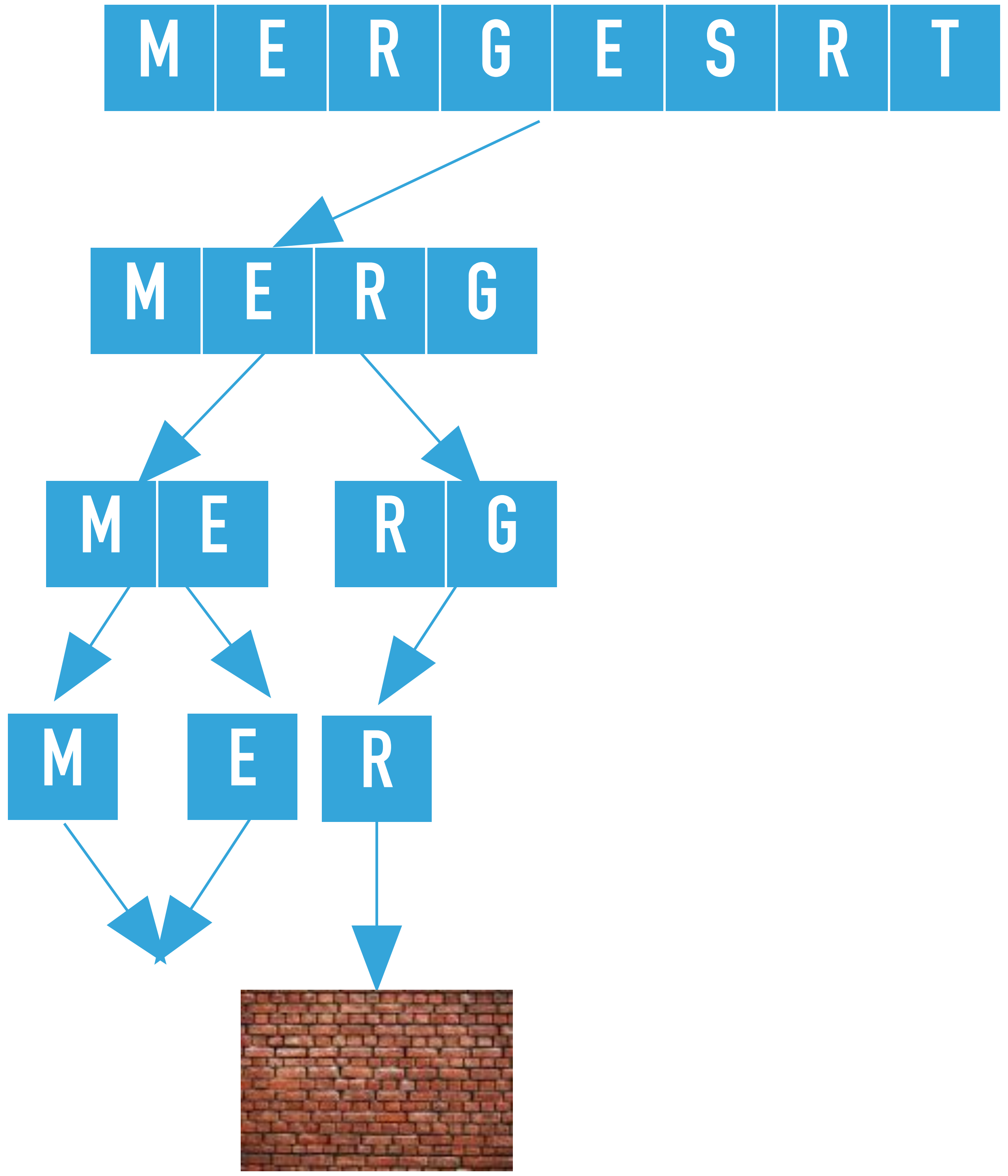

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



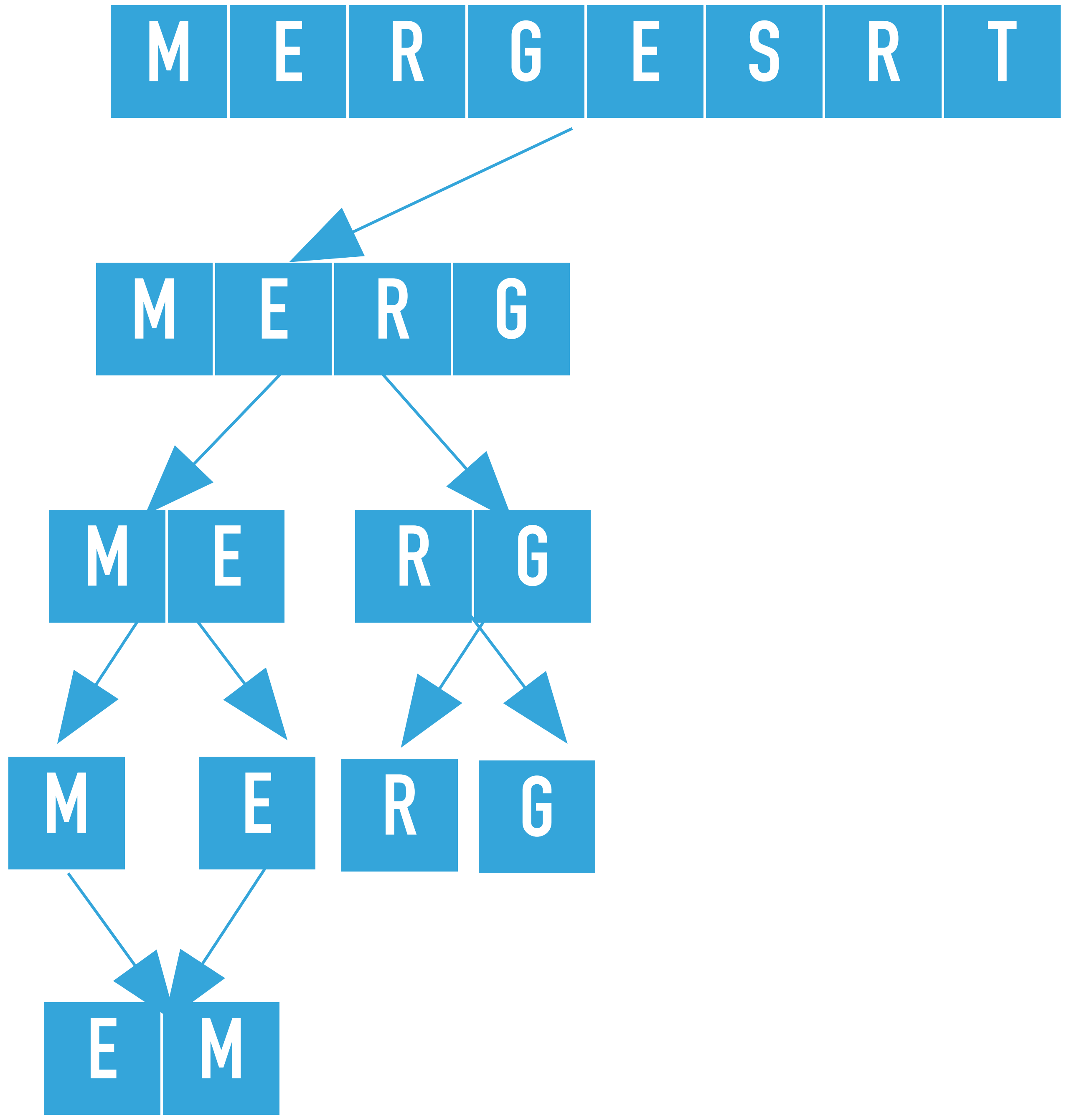

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



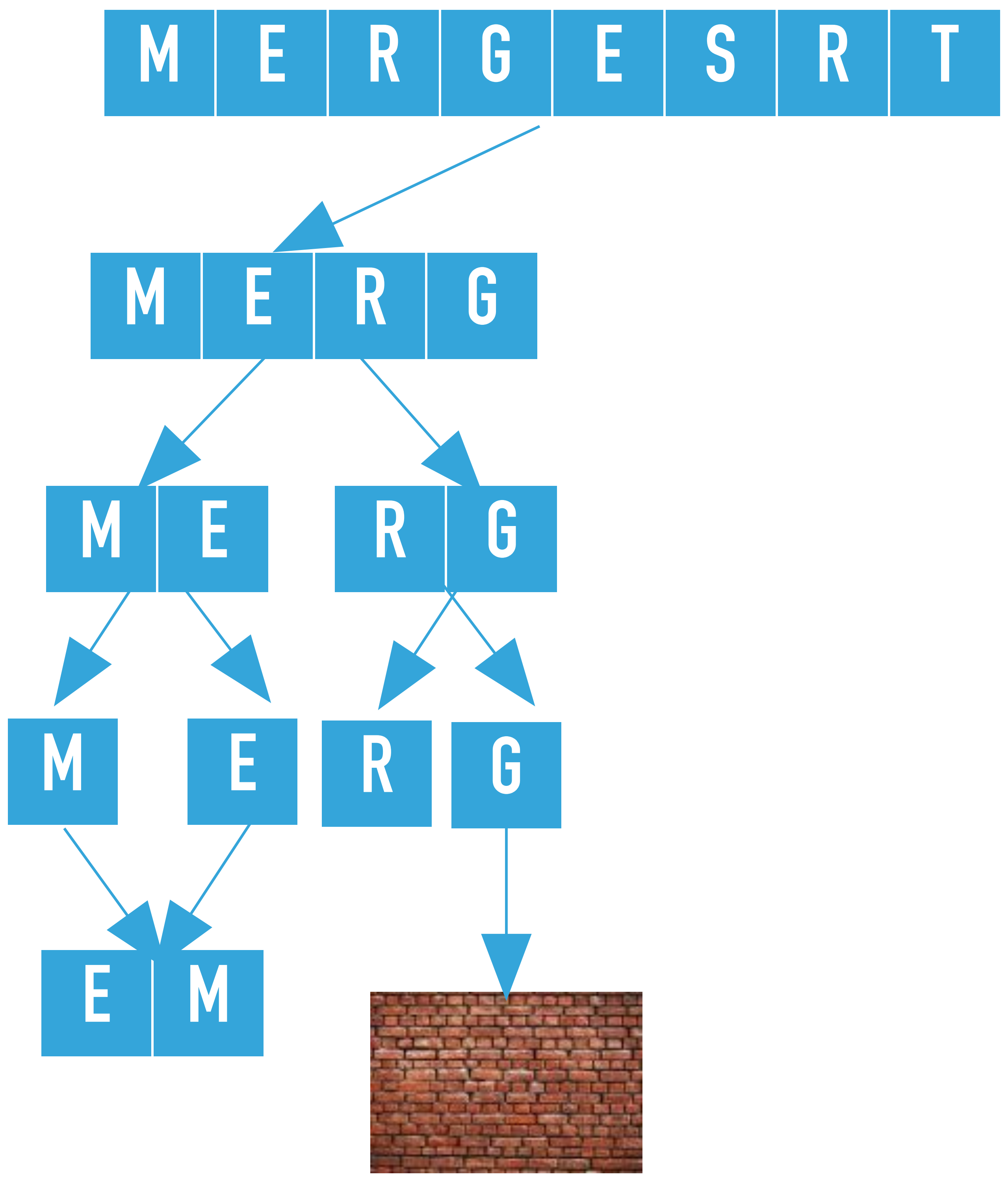

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



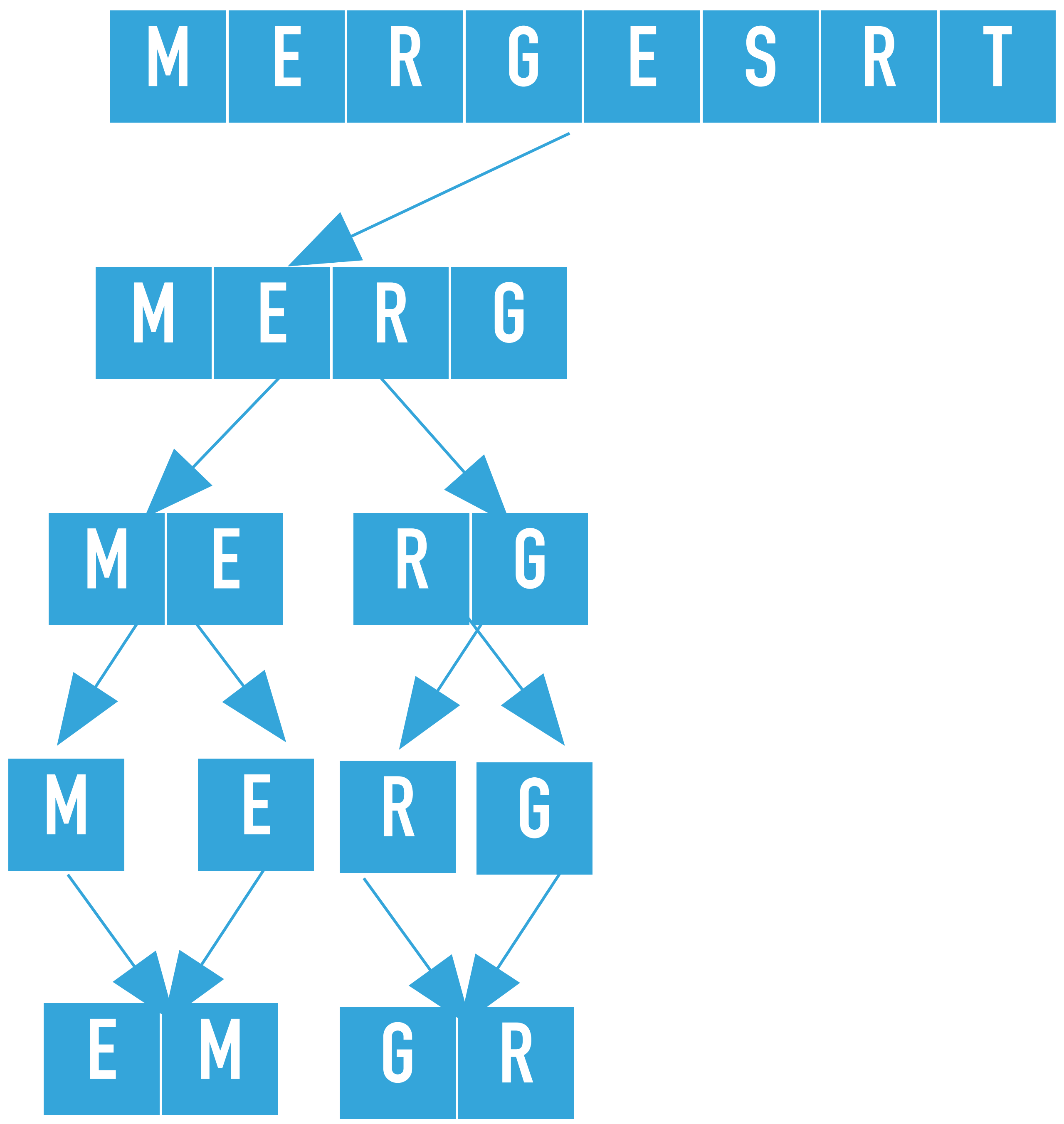

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



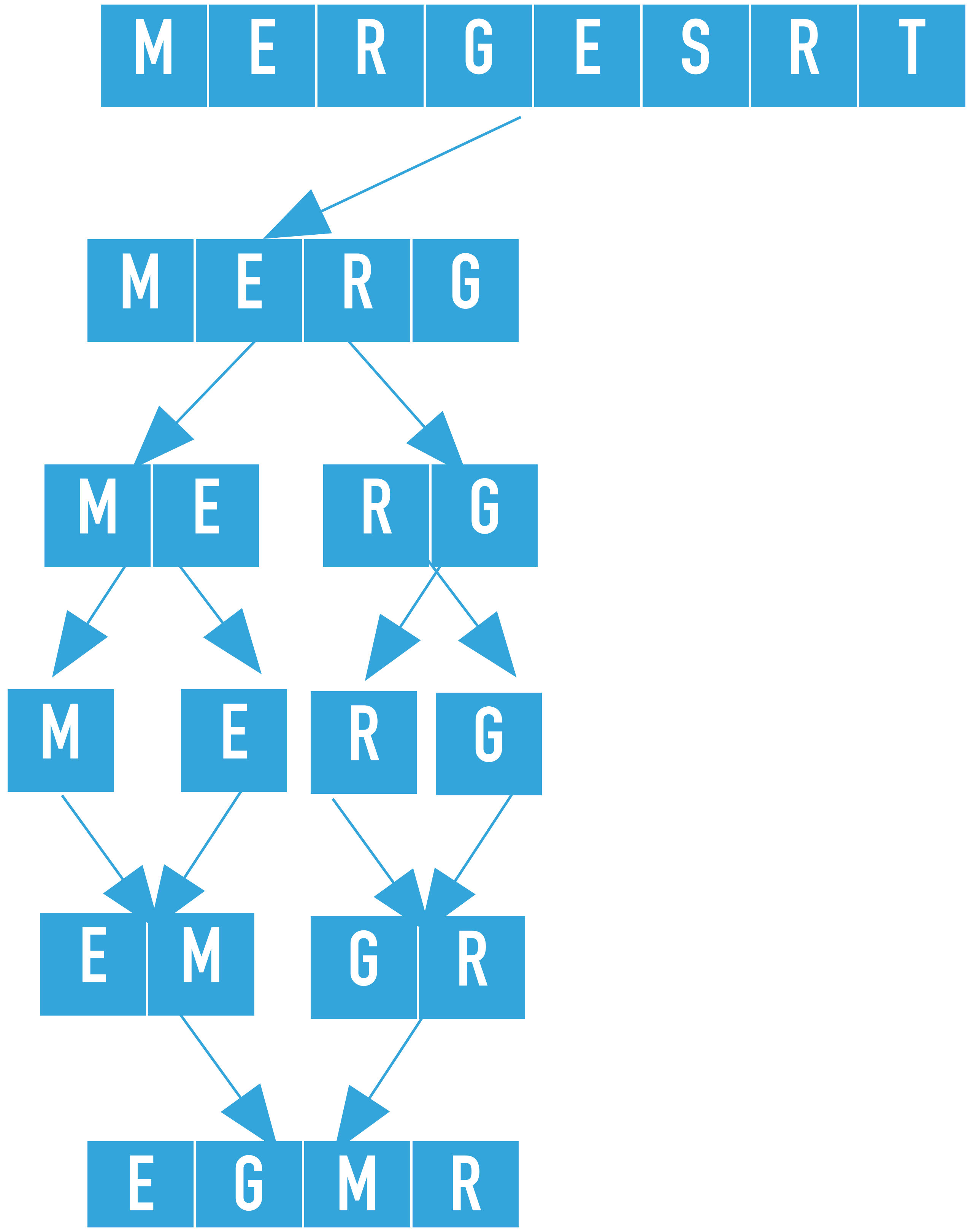

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



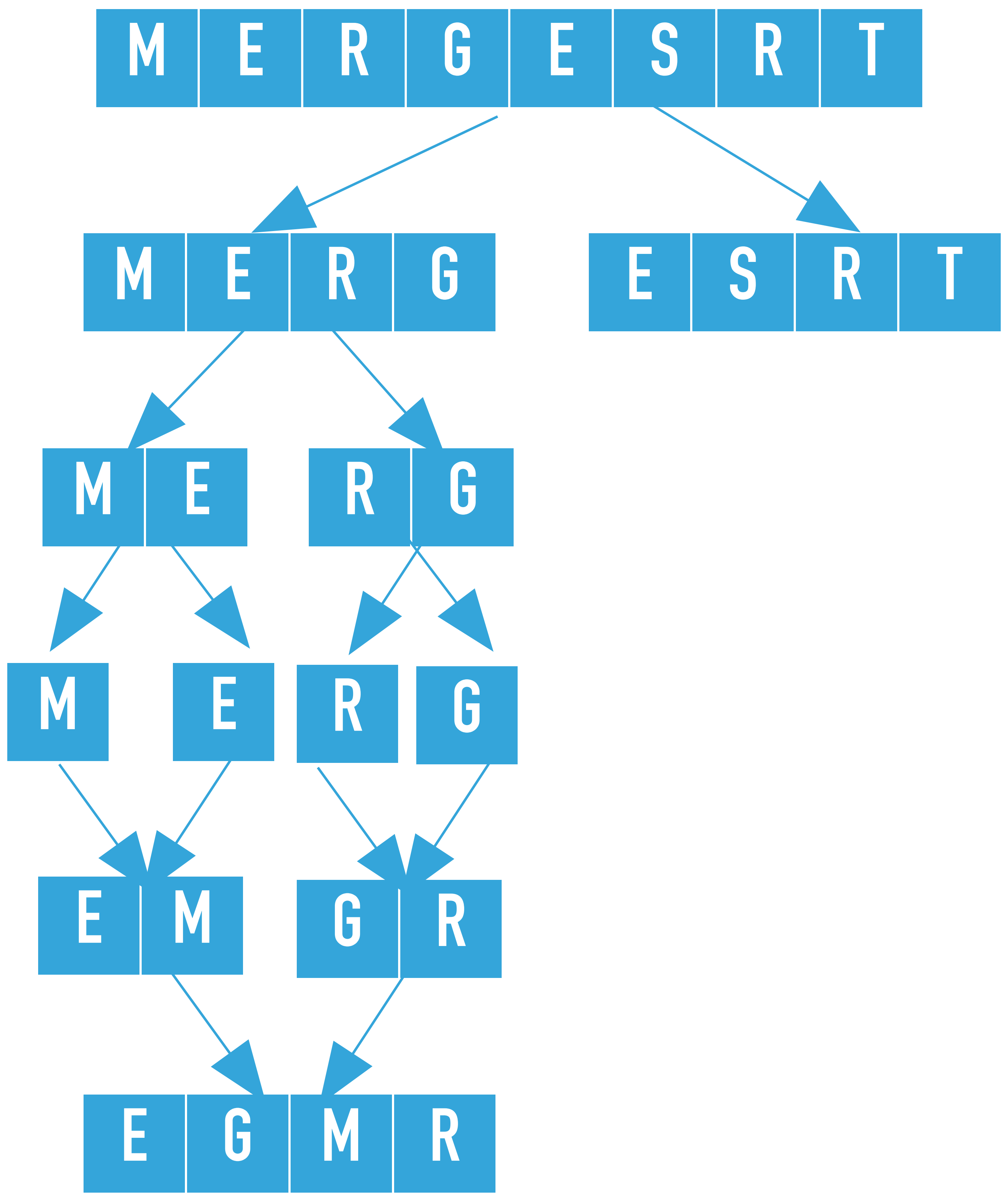

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



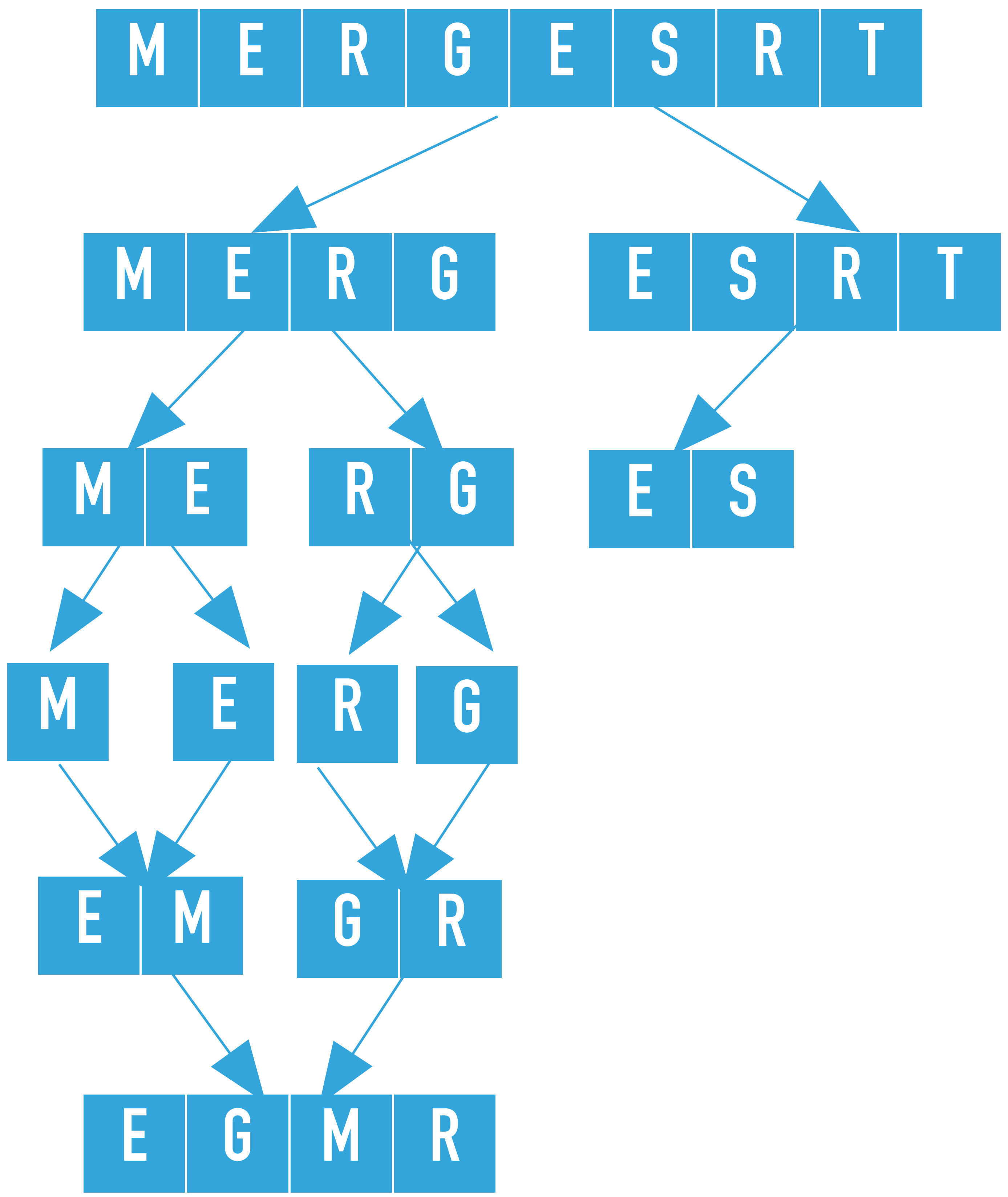

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



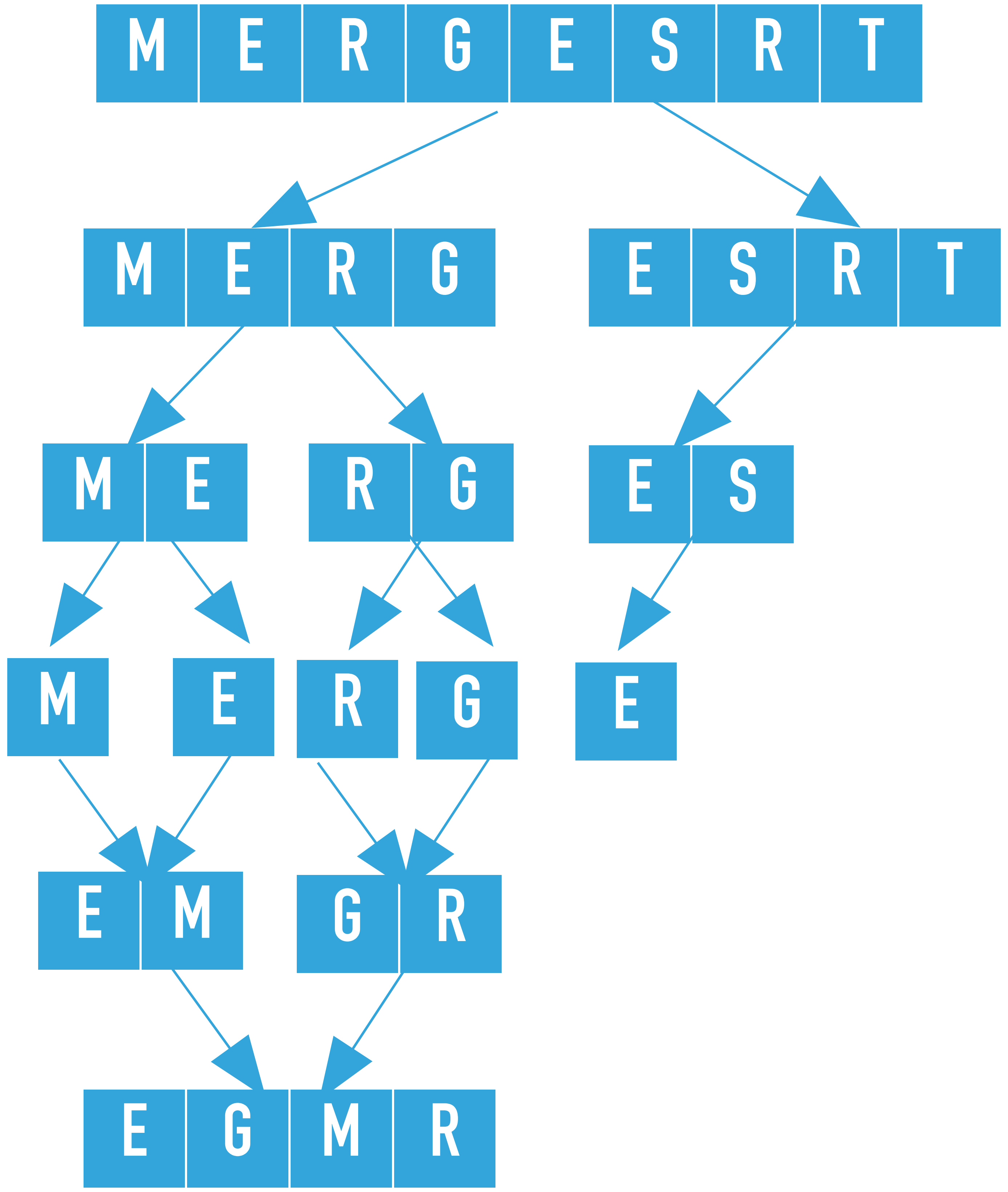

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



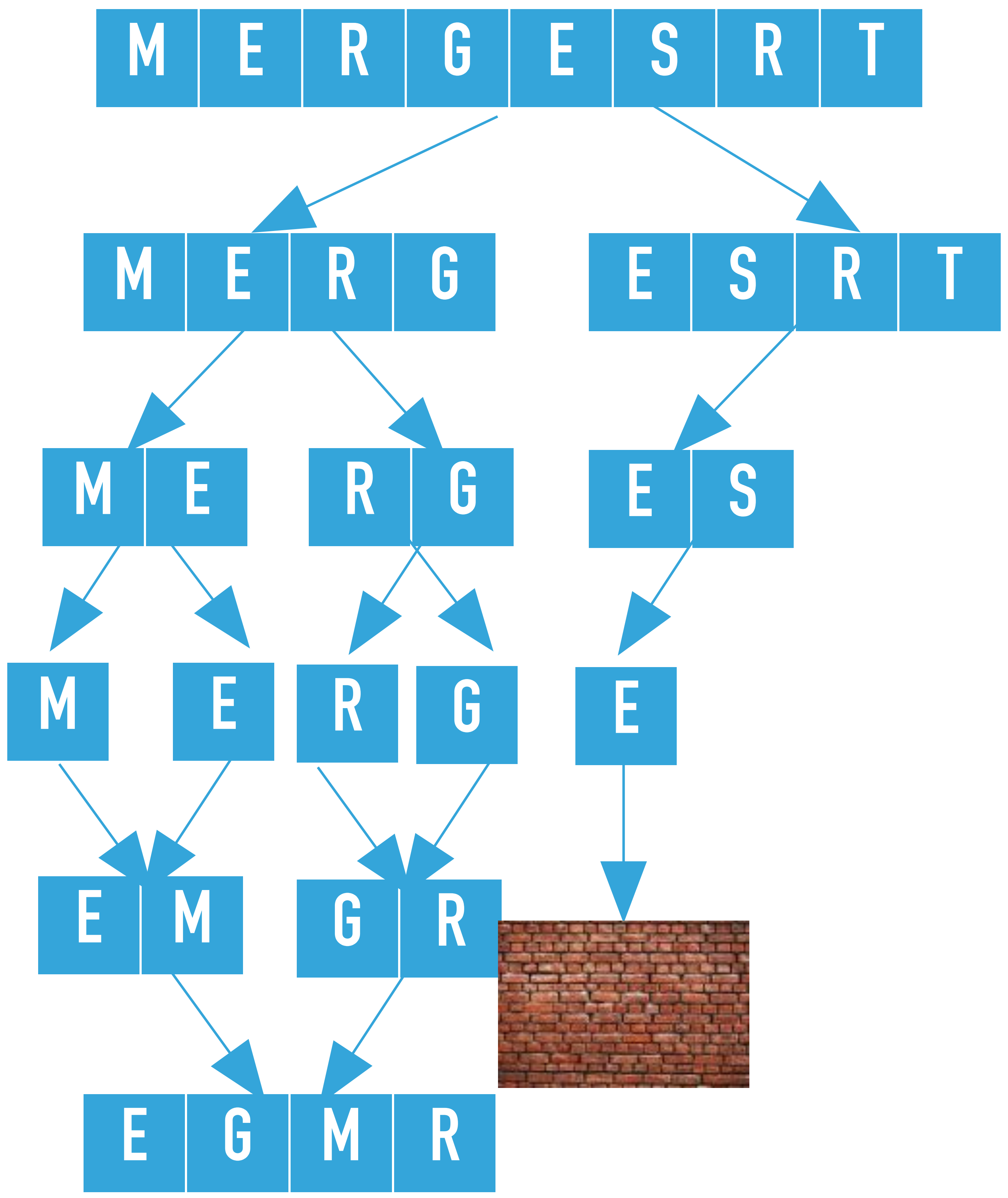

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`.




```
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);
}
```

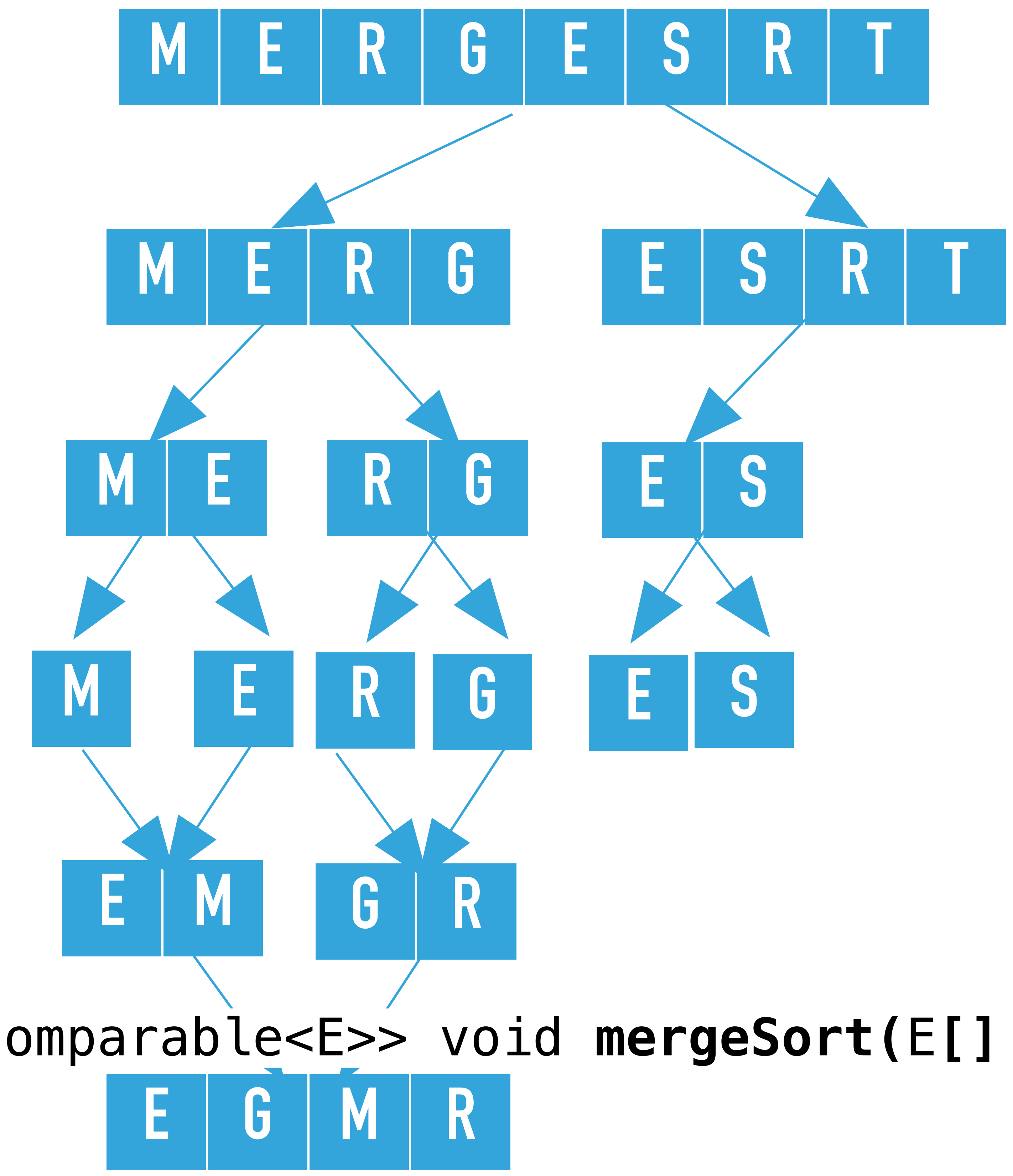
`mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` calculates the `mid = 4` and calls recursively `mergeSort` on the left subarray, that is `mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)`, where `lo = 4`, `hi = 4`.




```
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);
}
```

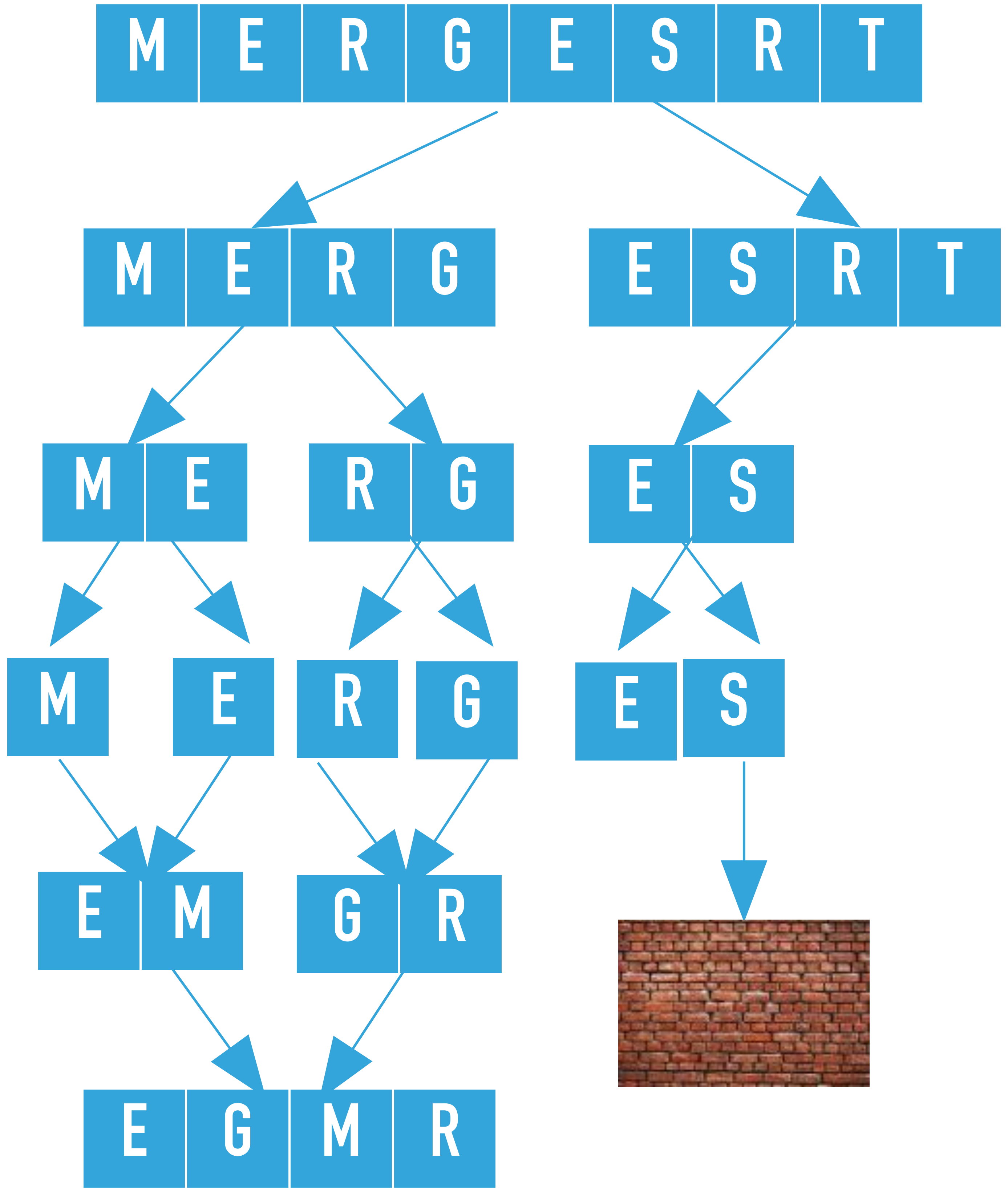
mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 4)

finds `hi <= lo` and returns.




```java
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);
}
```

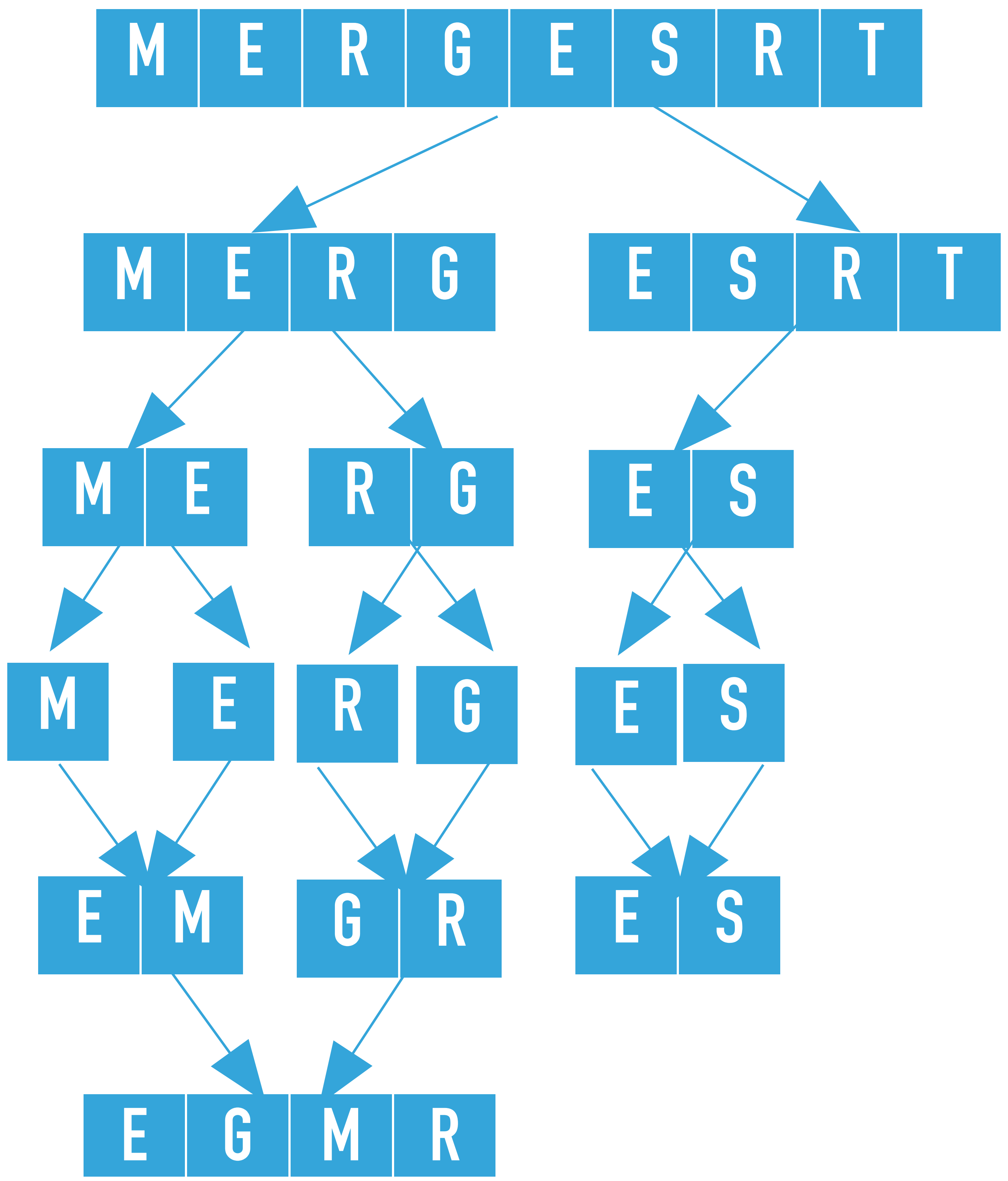
`mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 4, 5)` calls recursively `mergeSort` on the right subarray, that is `mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 5, 5)`, where `lo = 5`, `hi = 5`




```java
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);
}
```

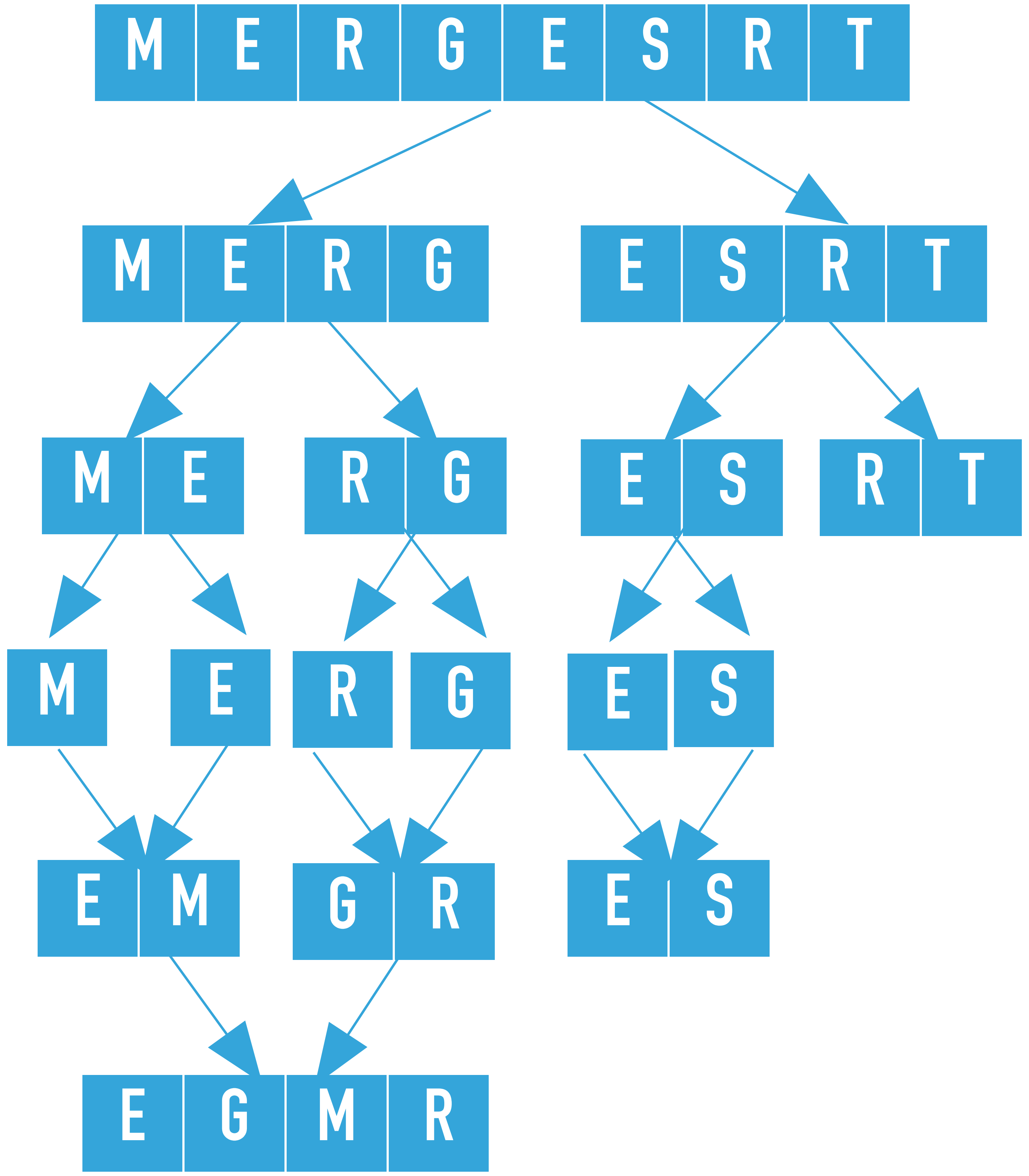
`mergeSort([E, G, M, R, E, S, R, T], [E, M, G, R, null, null, null, null], 5, 5)`

finds `hi <= lo` and returns.



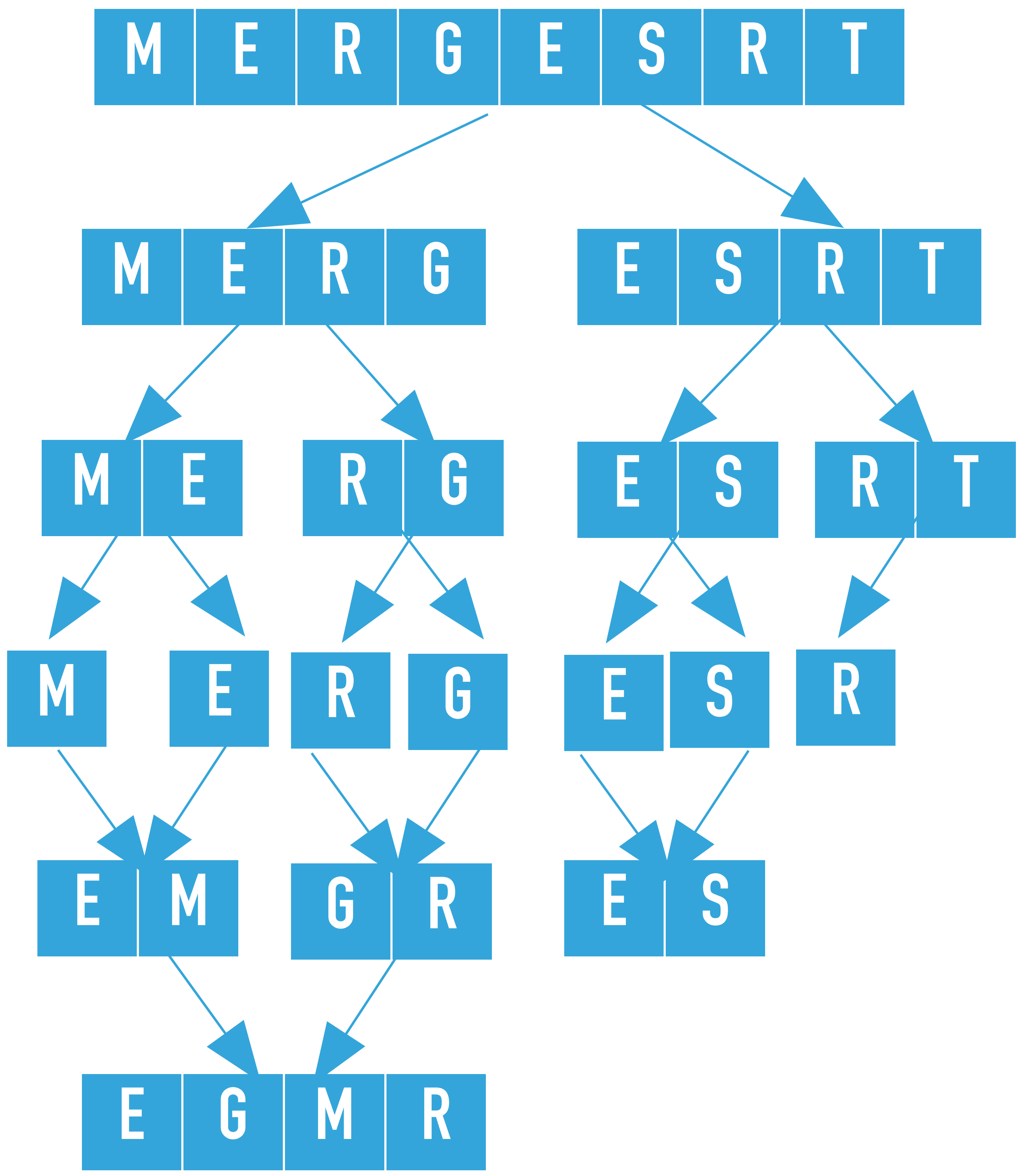

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.



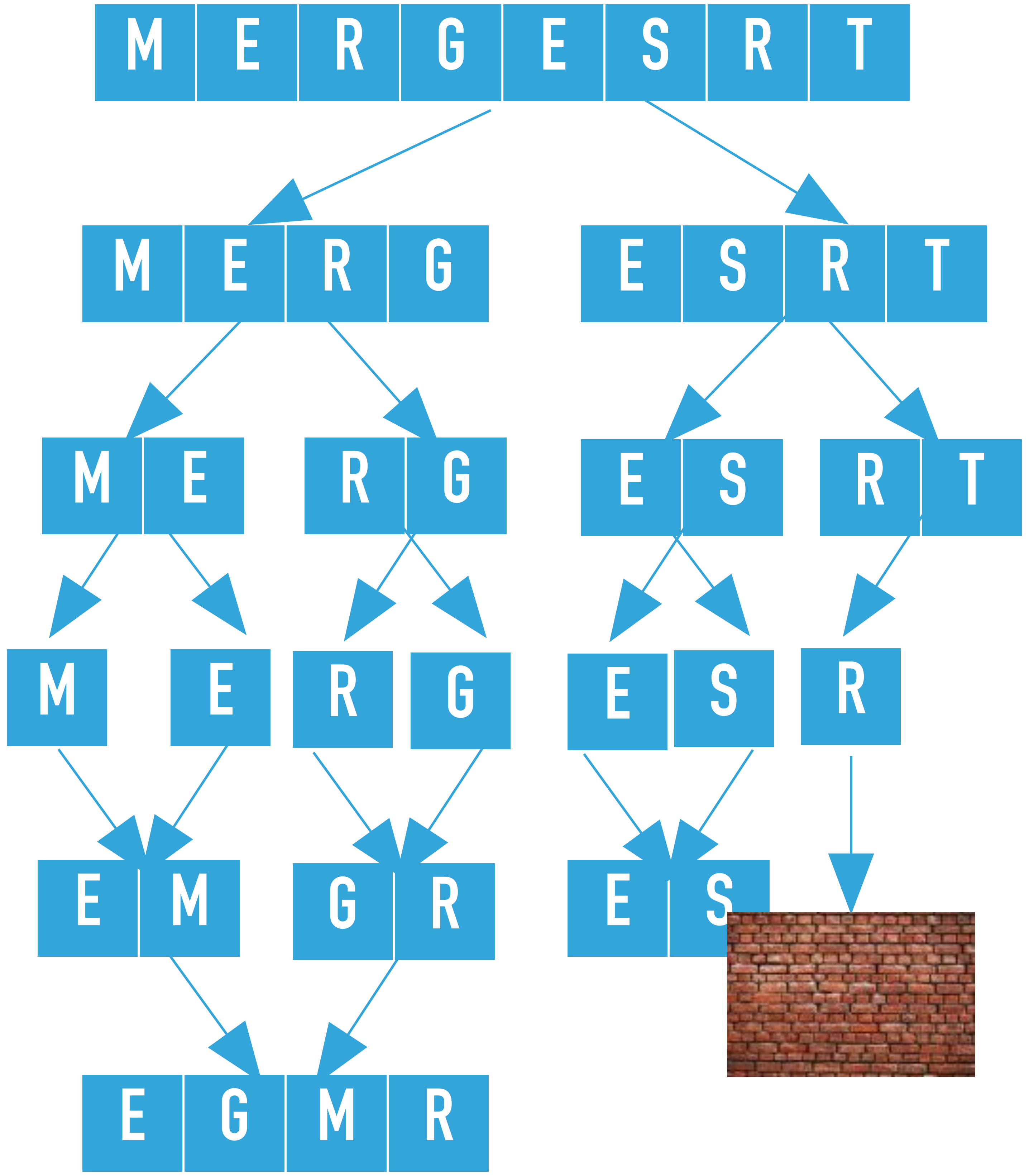

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



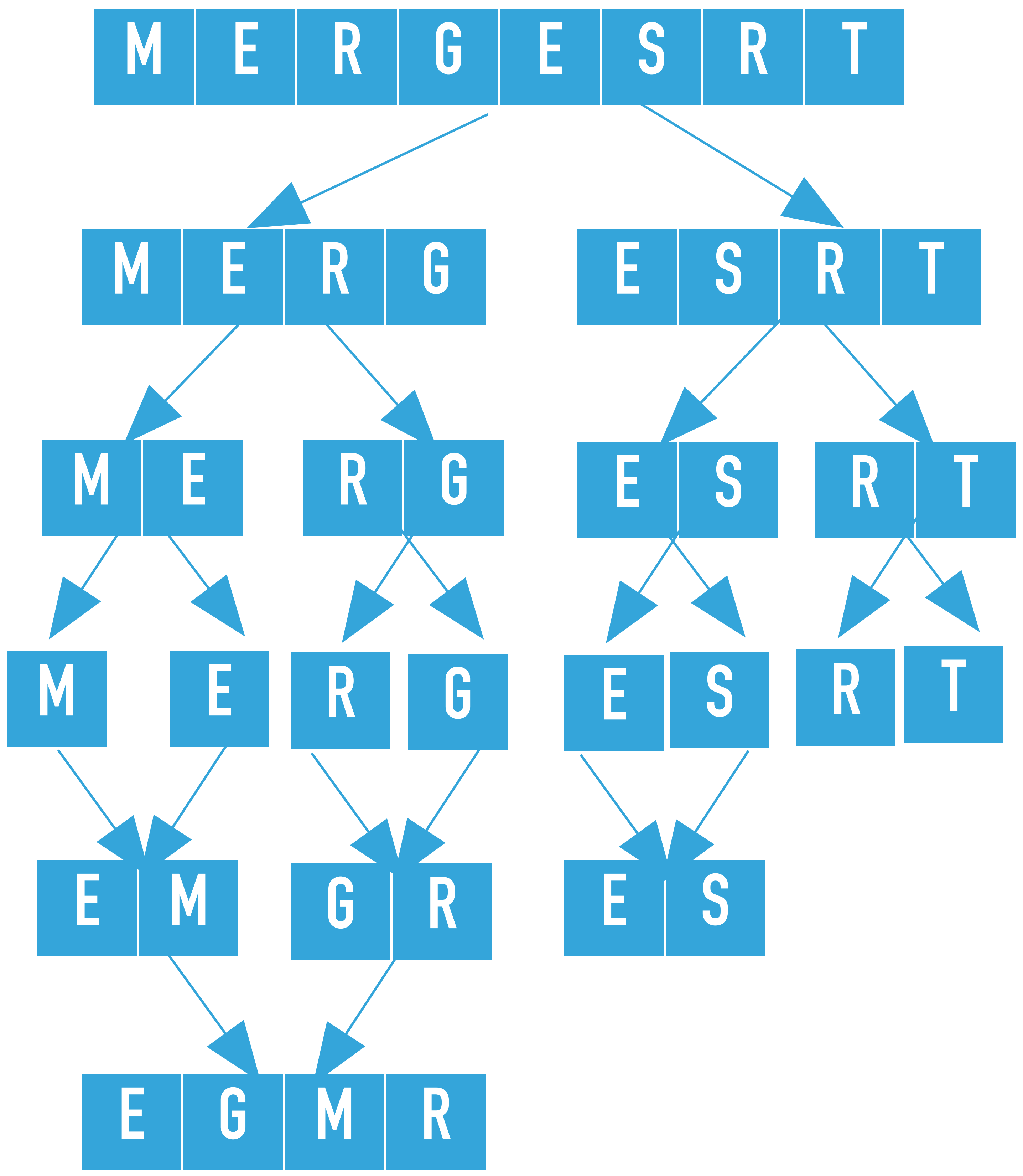

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`.



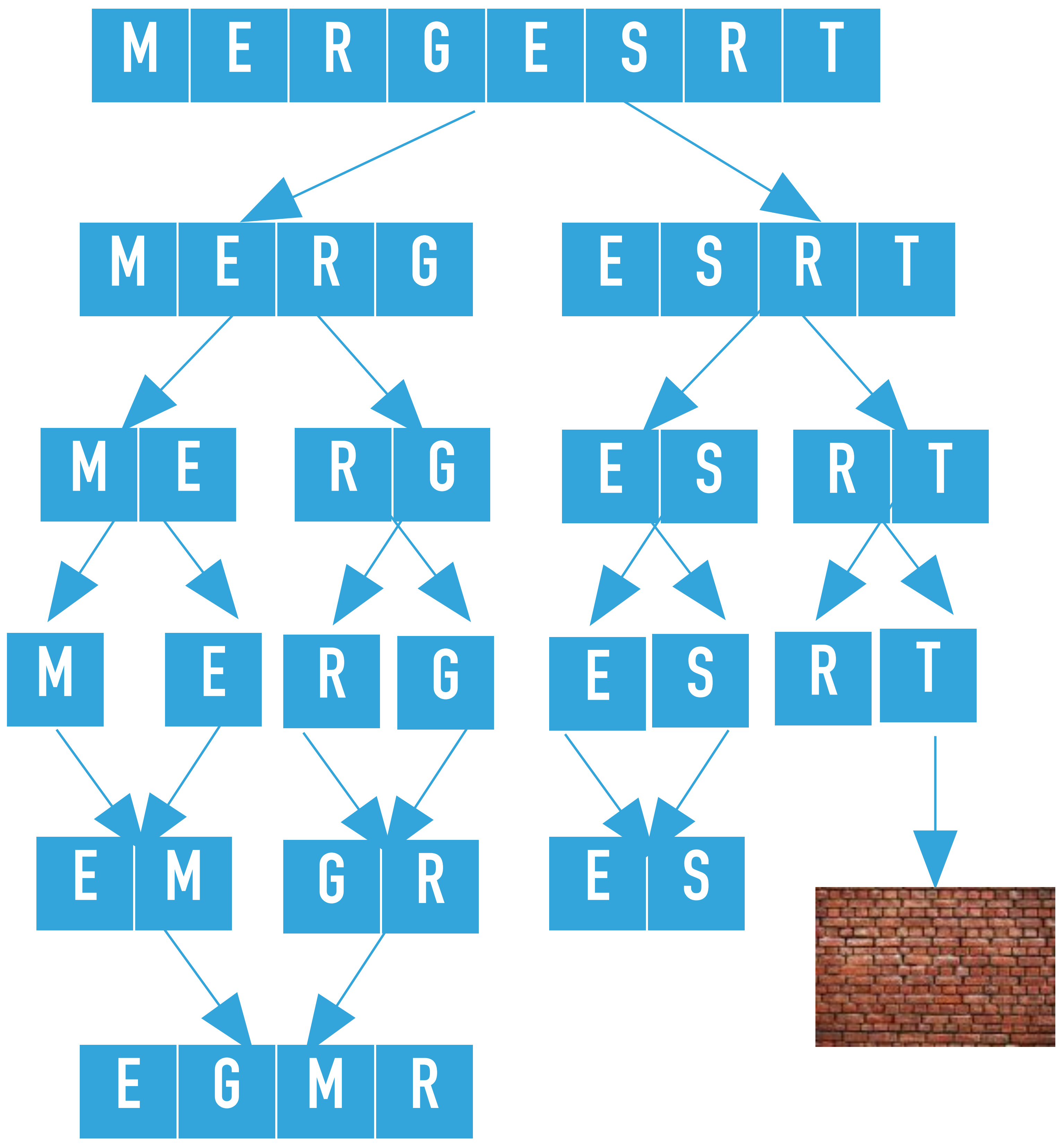

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

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



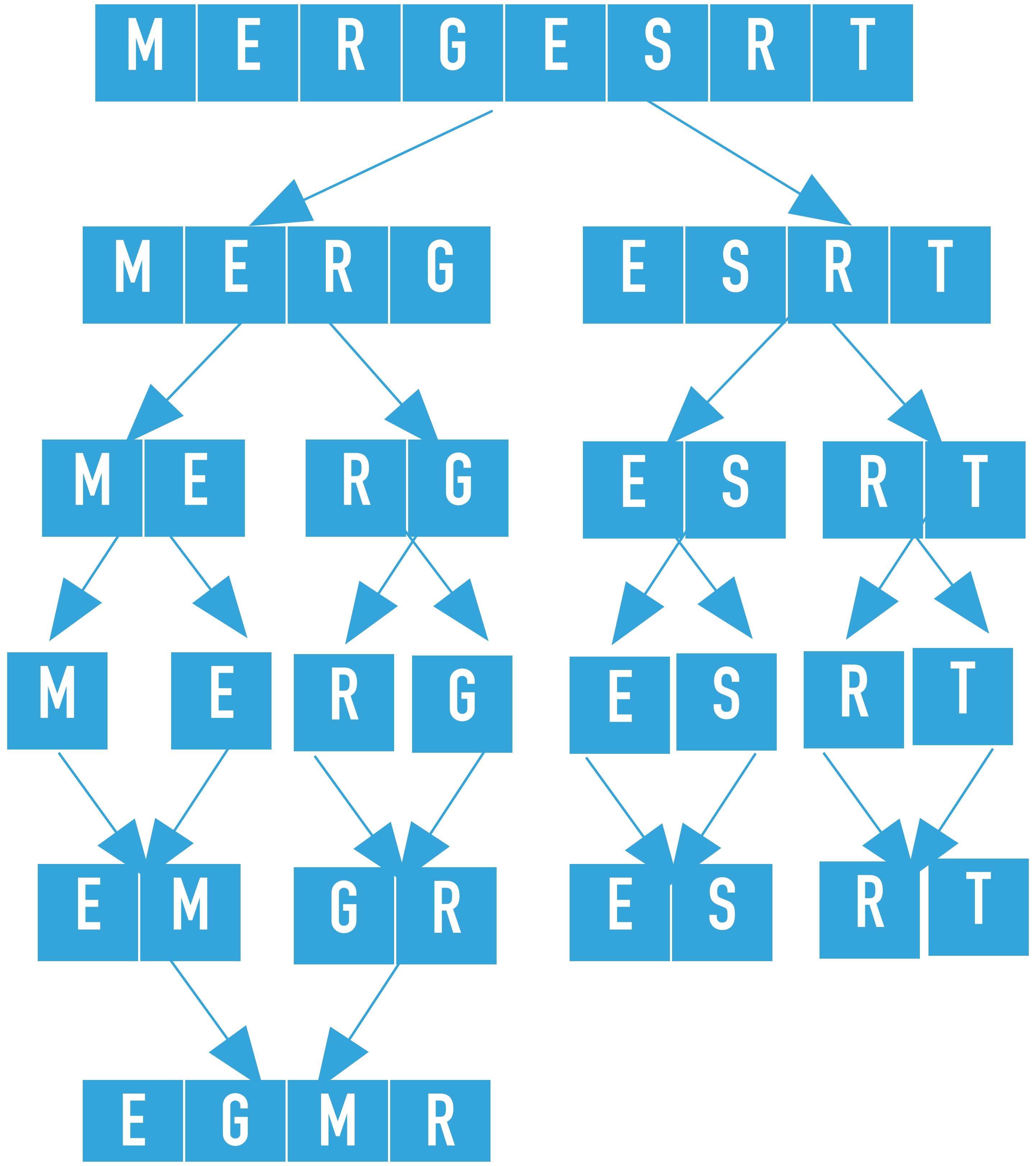

```
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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`



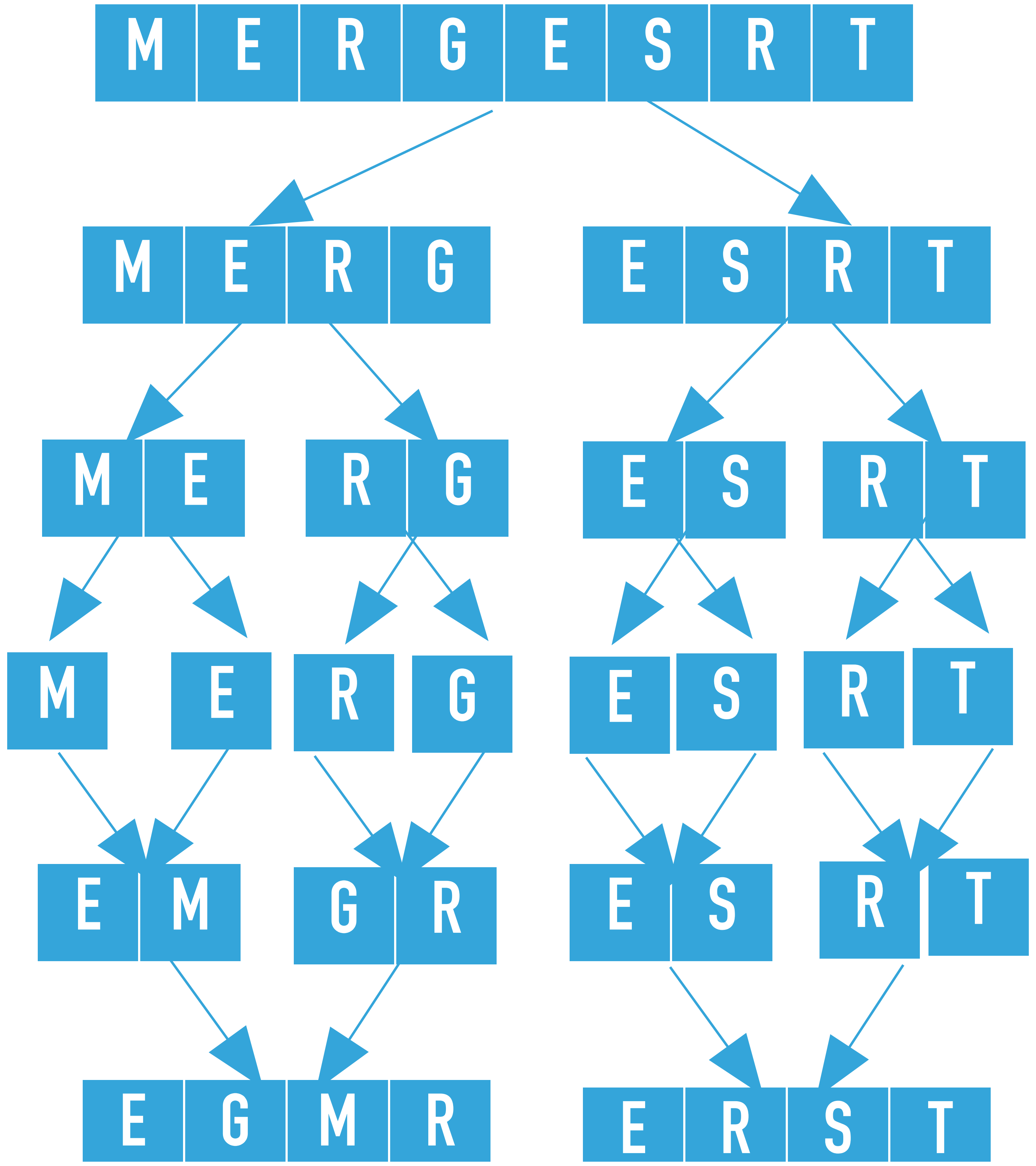

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
     if (hi <= lo){
          return;
     }
     int mid = lo + (hi - lo) / 2;
     mergeSort(a, aux, lo, mid);
     mergeSort(a, aux, mid+1, hi);
     merge(a, aux, lo, mid, hi);
}
```

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



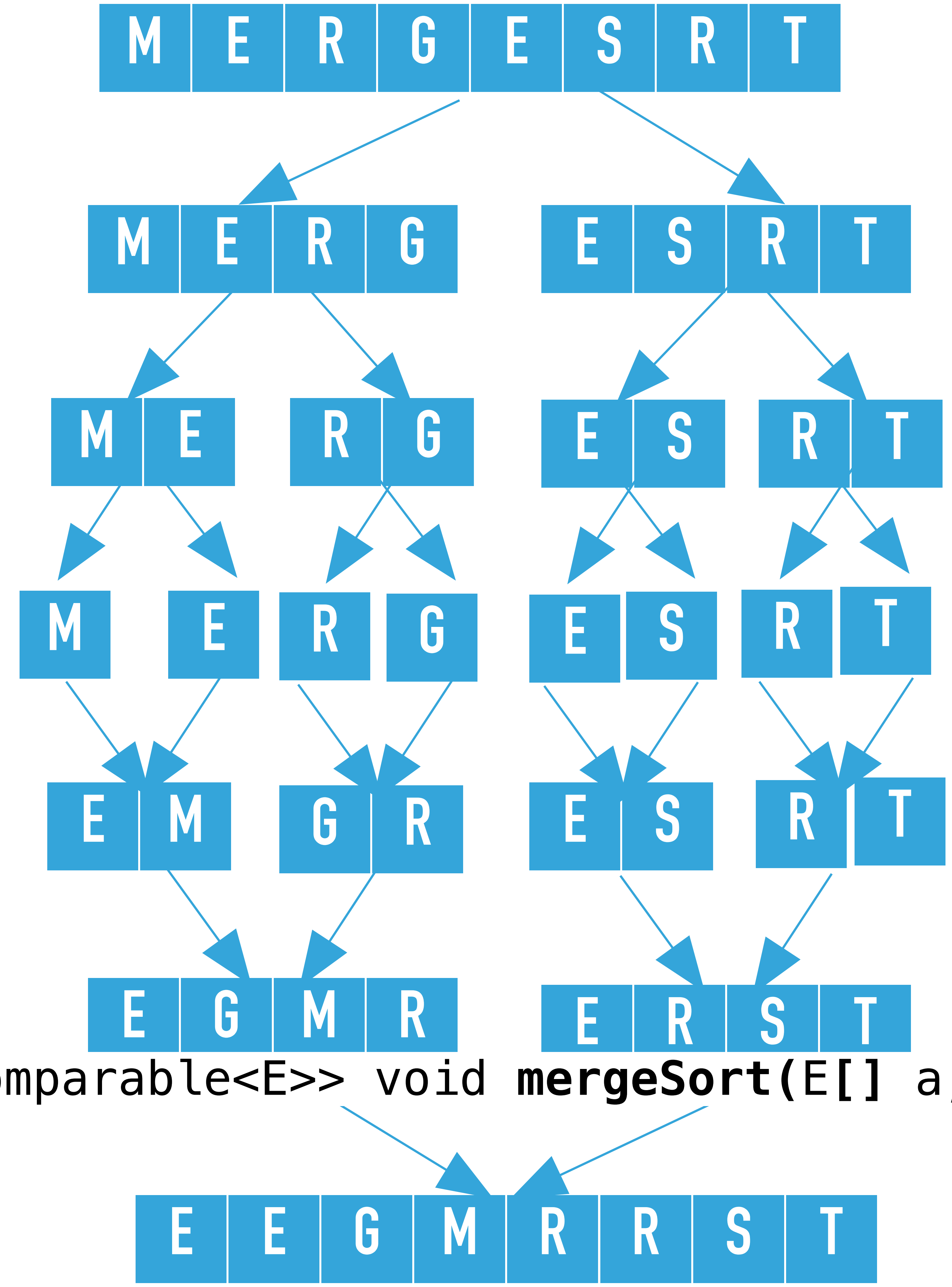

```java
private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi)
{
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

`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]`.

Worksheet 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 }

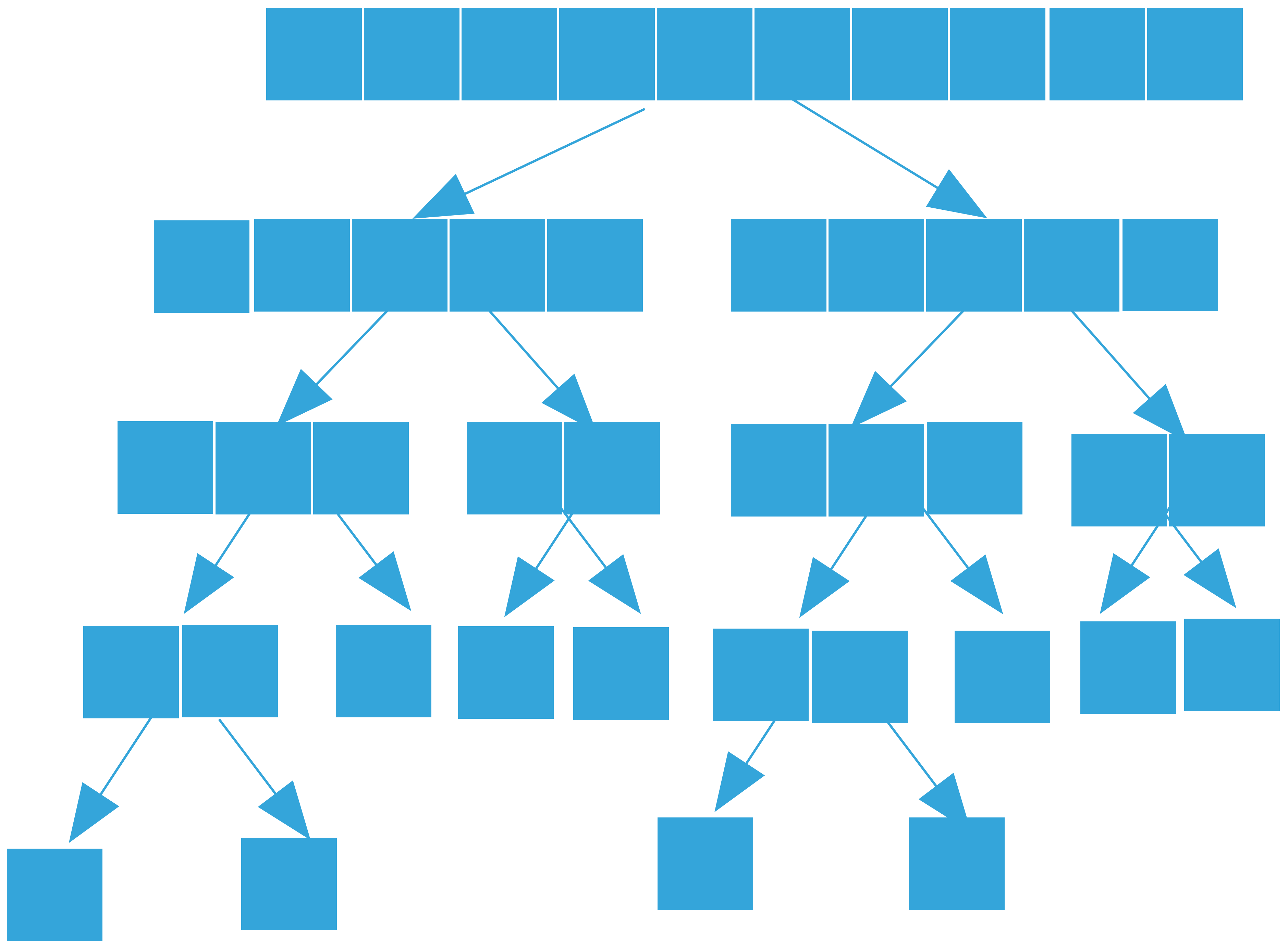
C. { 1, 2, 5, 10 }

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

Worksheet answer

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

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



Analysis of Mergesort

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

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

Log(n) levels, each level takes O(n) time to merge

If $n = 4$, 2 levels

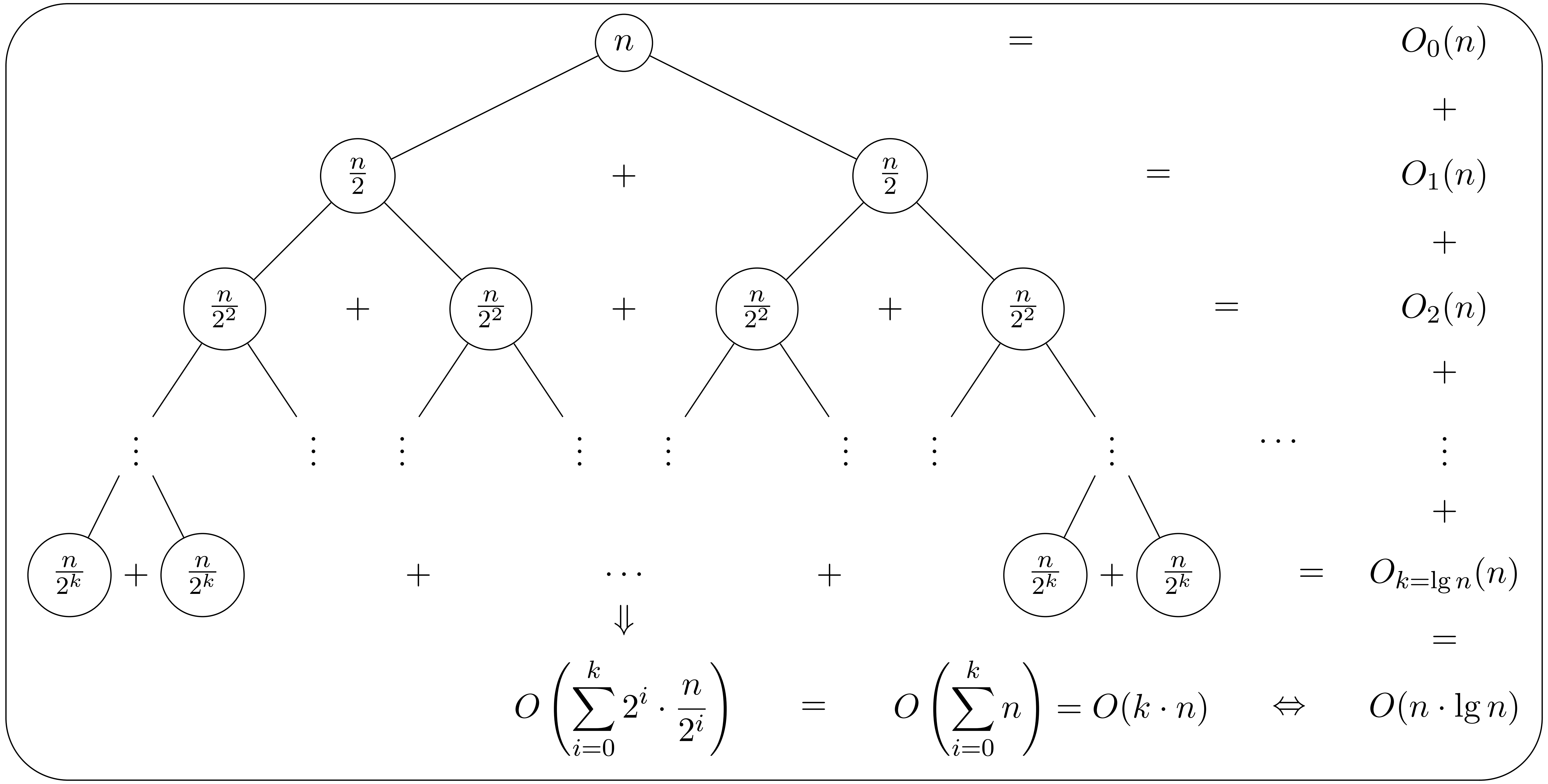
If $n = 8$, 3 levels

If $n = 16$, 4 levels

...

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

or $k = log_2 n$



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

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:
  - $T(n) = T(n/2) + T(n/2) + (n - 1) = O(n \log n)$

  \# comparisons = # comparisons left subarray + # comp. right subarray + n-1 for merge

    - Specifically, it's between $\sim \frac{1}{2} n \log n$ and $n \log n$

- Number of array accesses (rather than exchanges, here) is also $O(n \log n)$.
  - Specifically, at most $6n \log n$

Look at the Master theorem for a proof of this: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Array Accesses

```
    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];      2n to populate aux by copying a
        }
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid) { // ran out of elements in the left subarray
              2 a[k] = aux[j++];
            } else if (j > hi) { // ran out of elements in the right subarray
              2 a[k] = aux[i++];
            } else if (aux[j].compareTo(aux[i]) < 0) {  2
                a[k] = aux[j++];  2
            } else {
                a[k] = aux[i++];  2
            }
        }                        averages out to 4 array accesses per iteration of the second for loop
    }
```

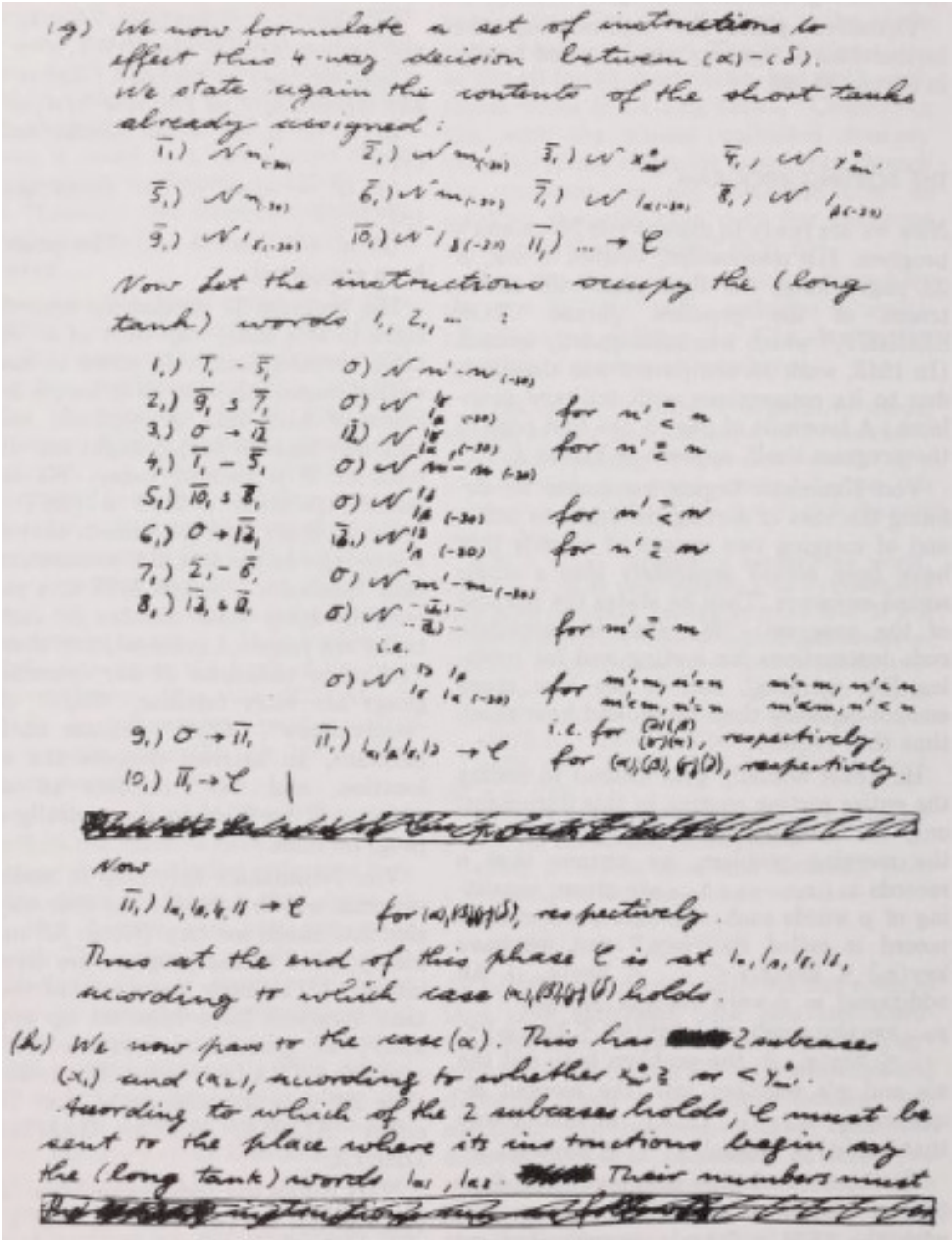
2n copying + 2n comparisons + 2n changing a = ~6n array accesses for each merge() operation, do merge() log(n) times

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);
}
```

History of sorting algorithms (+ Mergesort)

- Computational sorting algorithms were actually first developed in response to hardware constraints!
- Herman Hollerith developed radix sort in the 1800s because of punch card machines (each hole in the punch card was each digit)
- Don Knuth (inventor of run time analysis) found Von Neumann's original Mergesort manuscript

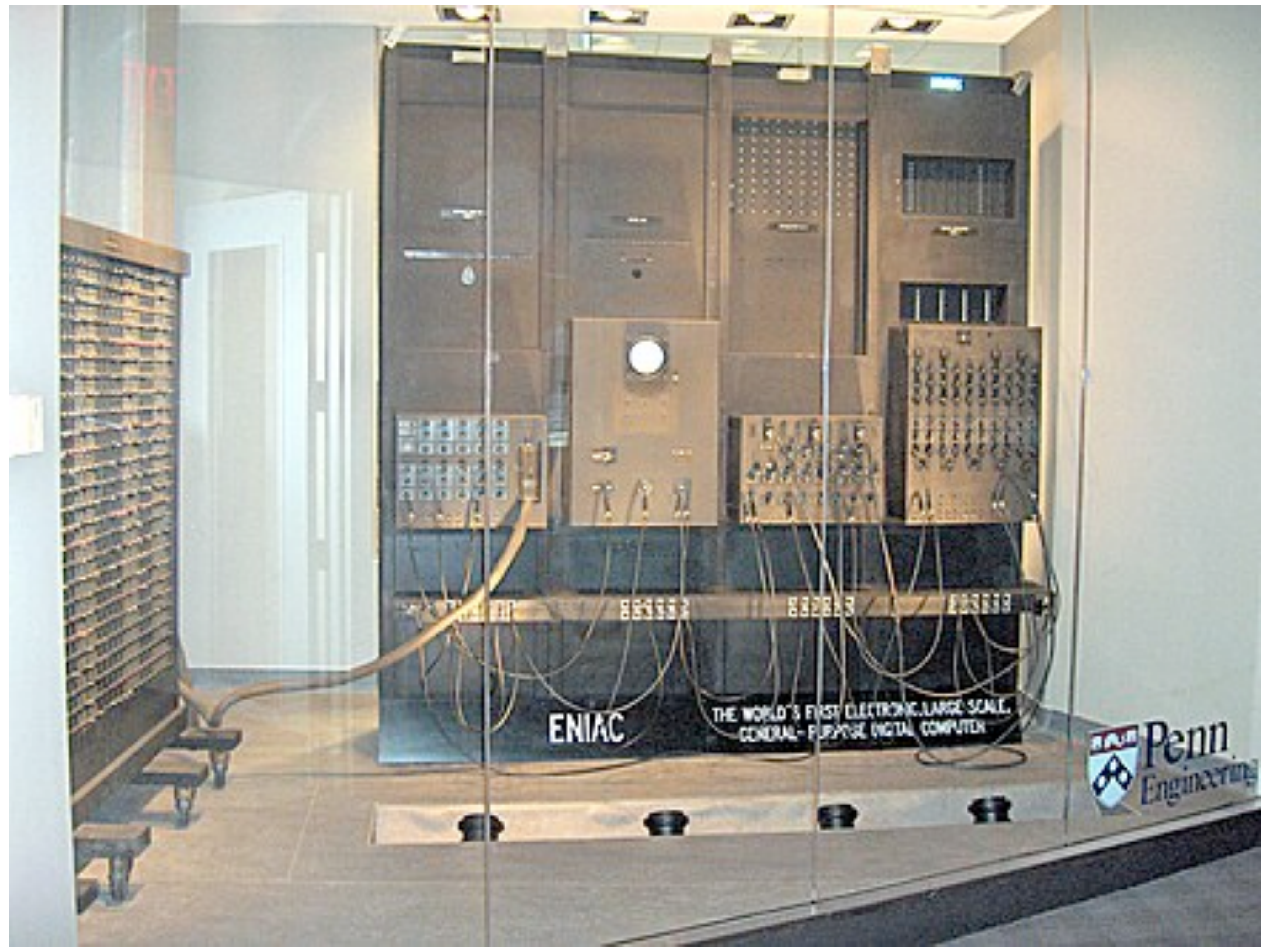

(g) We now formulate a set of instructions to effect this 4-way decision between (α)-(δ). We state again the contents of the short tanks already assigned:

https://cs.pomona.edu/classes/cs62/history/sortingalgs/

https://compileralchemy.substack.com/p/merge-sort-and-its-early-history

History of sorting algorithms (+ Mergesort)

- The ENIAC machine was developed in 1945 at UPenn and considered one of the first computers. 1945? For World War II, of course…to calculate artillery firing tables. (It cost $7 mil in 2023 money to build)
- One advantage of the ENIAC was that it had parallel memory, so it could do different calculations at the same time
- Hence, Mergesort, our divide-and-conquer algorithm!



https://cs.pomona.edu/classes/cs62/history/sortingalgs/

Worksheet time!

Is Mergesort in place?

Is Mergesort stable?

One way to examine ethics in algorithms is the idea of “inconclusive evidence”, in that the data being sorted and .compareTo()’d might not paint the whole picture. Sorting numbers is one task, but in the real world, we sort things like candidates for jobs, search result rankings, financial risk assessments, or movie/song recommendations. Pick one of these “real world” categories (or choose your own) and discuss how we might make the “evidence” more “conclusive”. Should you change what “evidence” gets sorted? Add more evidence? Avoid sorting altogether?

https://doi.org/10.1177/2053951716679679

Worksheet answers

- Auxiliary memory is proportional to n, as `aux[]` needs to be of length n for the last merge.
- At its simplest form, mergesort is **not** an in-place algorithm.
- Mergesort is stable: Look into `merge()`, if equal keys, it takes them from the left subarray.
- There is no singular right answer to the last question, but thinking about how you might design more equitable algorithms is certainly an important skill :)

Practical improvements for Mergesort

- Use insertion sort for small subarrays (improves runtimes by ~10-15%).
- Stop if already sorted (skip the call to merge if already merged - then O(n) best case).
- Eliminate the copy to the auxiliary array by saving time (not space).
- This is called Timsort! It's what Java actually uses when you call Collections.sort() (and Python too!).
  - Tim Peters got banned from the Python community for 3 months in 2024 https://lunduke.locals.com/post/5985667/python-bans-prominent-dev-for-enjoying-the-wrong-old-snl-sketch (This is a pro-Tim biased source)

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

The complexity of sorting

- No comparison-based sorting algorithm can guarantee to sort n items with fewer than O(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		$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Use for small arrays or partially ordered
Merge sort		X	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	Guaranteed performance; stable

Lecture 13 wrap-up

- Exit ticket: https://forms.gle/8pBQiSni7RwPbzhDA
- HW5: Compression part 1 due tonight 11:59pm
  - The autograder results will say -/2 since we're hiding your grade on purpose since we gave lots of JUnit tests
- HW5: Compression part 2 due Tues 11:59pm

Resources

- Reading from textbook: Chapter 2.2 (pages 270–277)
- Online textbook website - https://algs4.cs.princeton.edu/22mergesort/
- Mergesort visualizer (slow stepping) - https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/
- Practice problems behind this slide
- Exercise to the reader: why can comparison-based sorts not be better than O(nlogn)?

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.

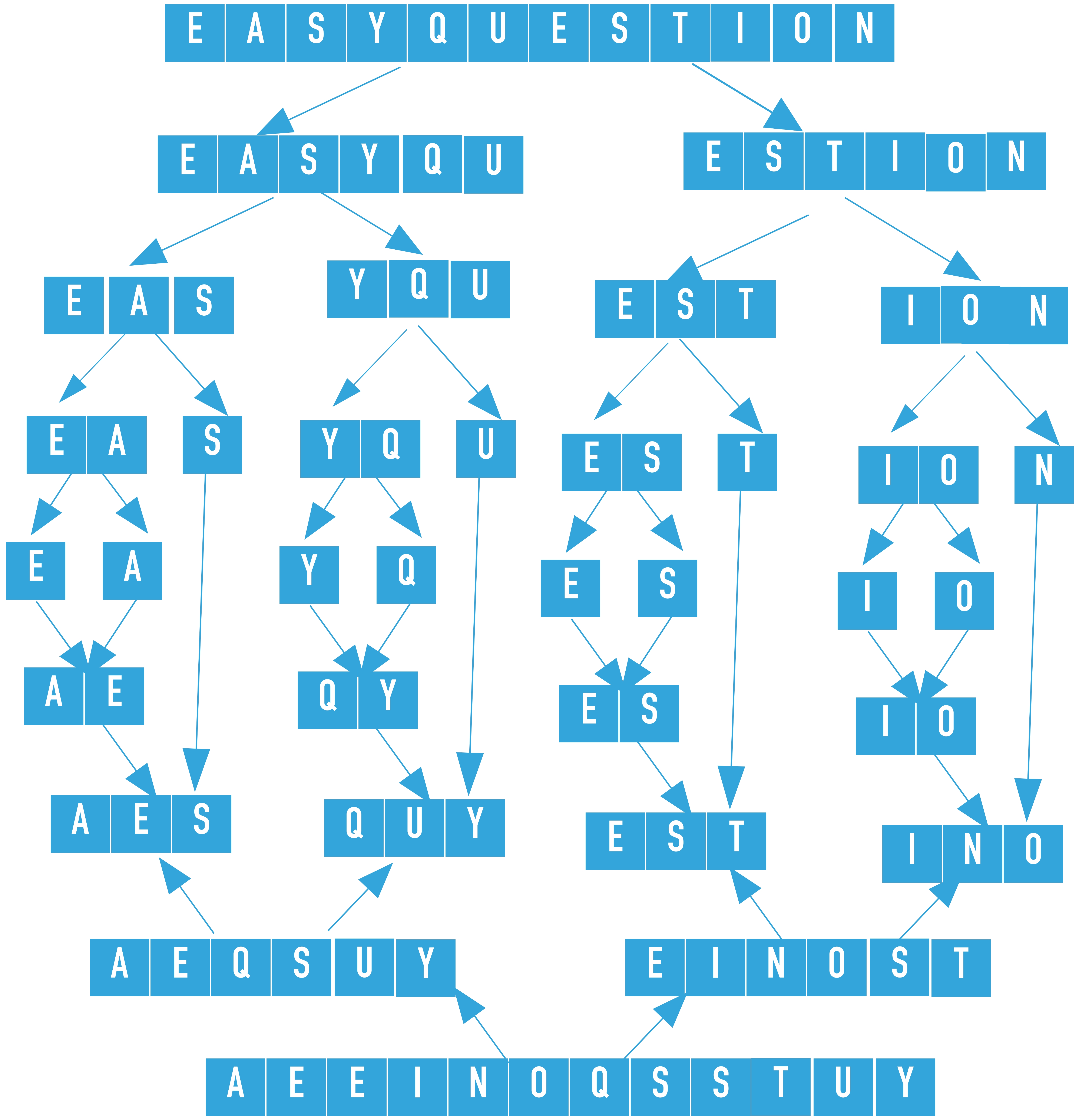
Practice Problem 2 - Recommended textbook 2.2.5

- 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


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

- 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 $6n \log n$
- Code for this is on the Github repo.