Merge Sort

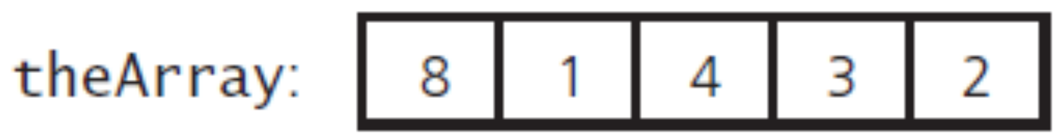
CS110C

Max Luttrell, CCSF

merge sort

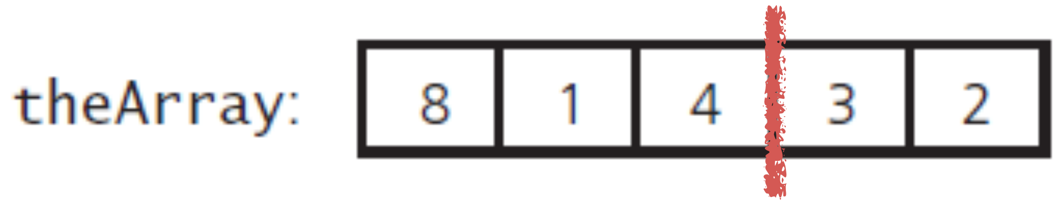
- Divide your array into halves
- Sort each half
- Merge the sorted halves into a temporary array so that the temporary array is sorted
- Copy values from (sorted) temporary array back to original array

merge sort



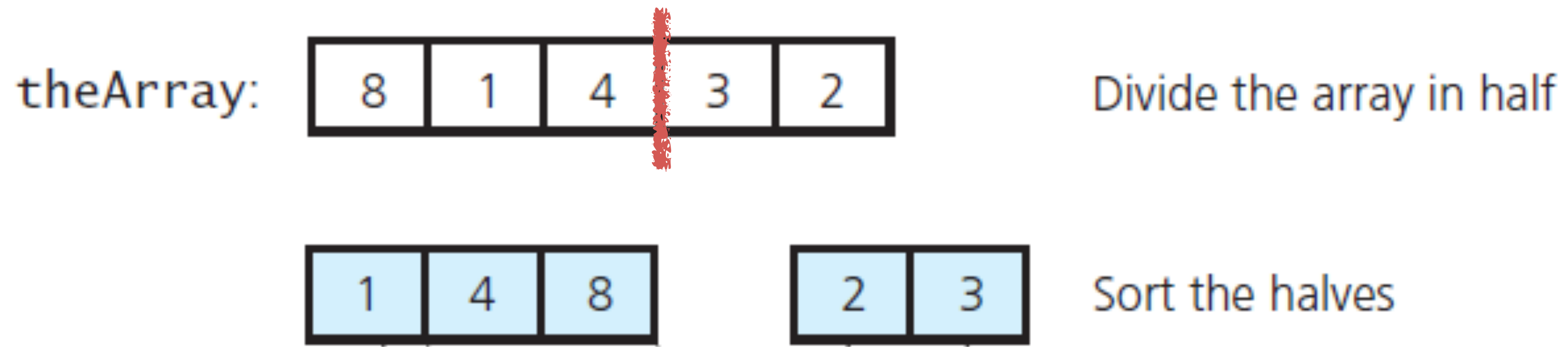
Divide the array in half

merge sort

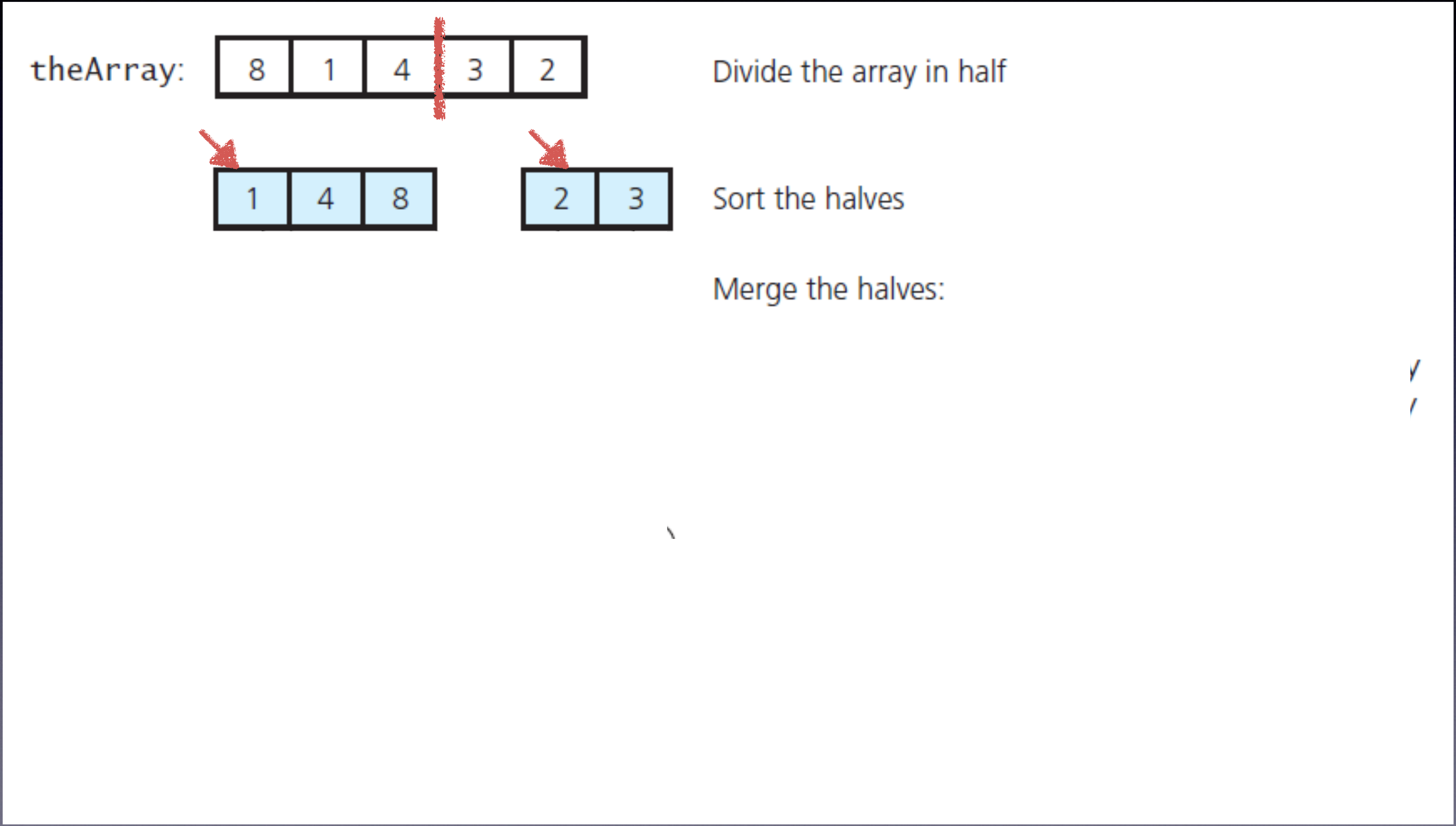


Divide the array in half

merge sort


theArray:
8
1
4
3
2
Divide the array in half
1
4
8
2
3
Sort the halves

merge sort

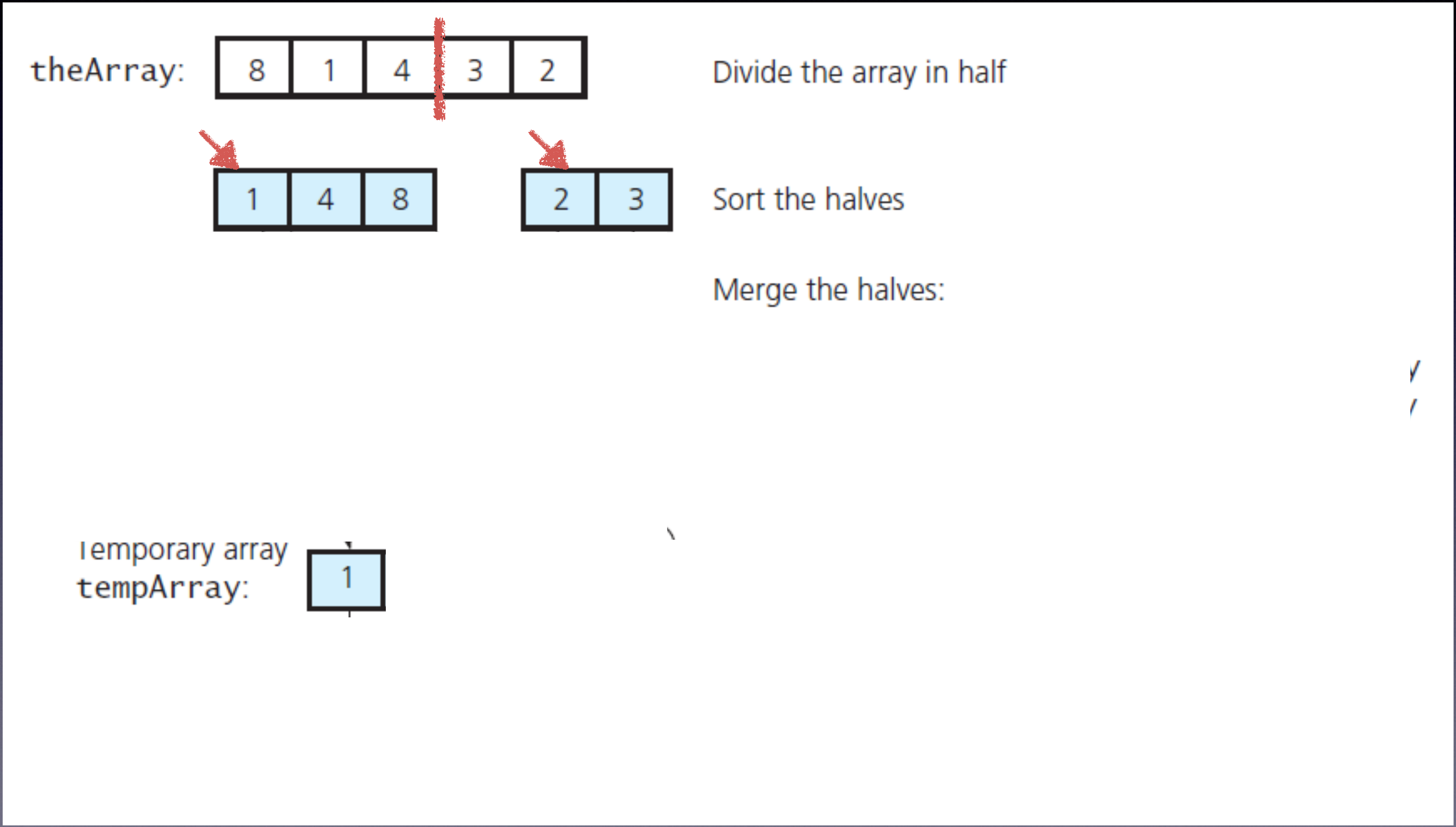


Divide the array in half

Sort the halves

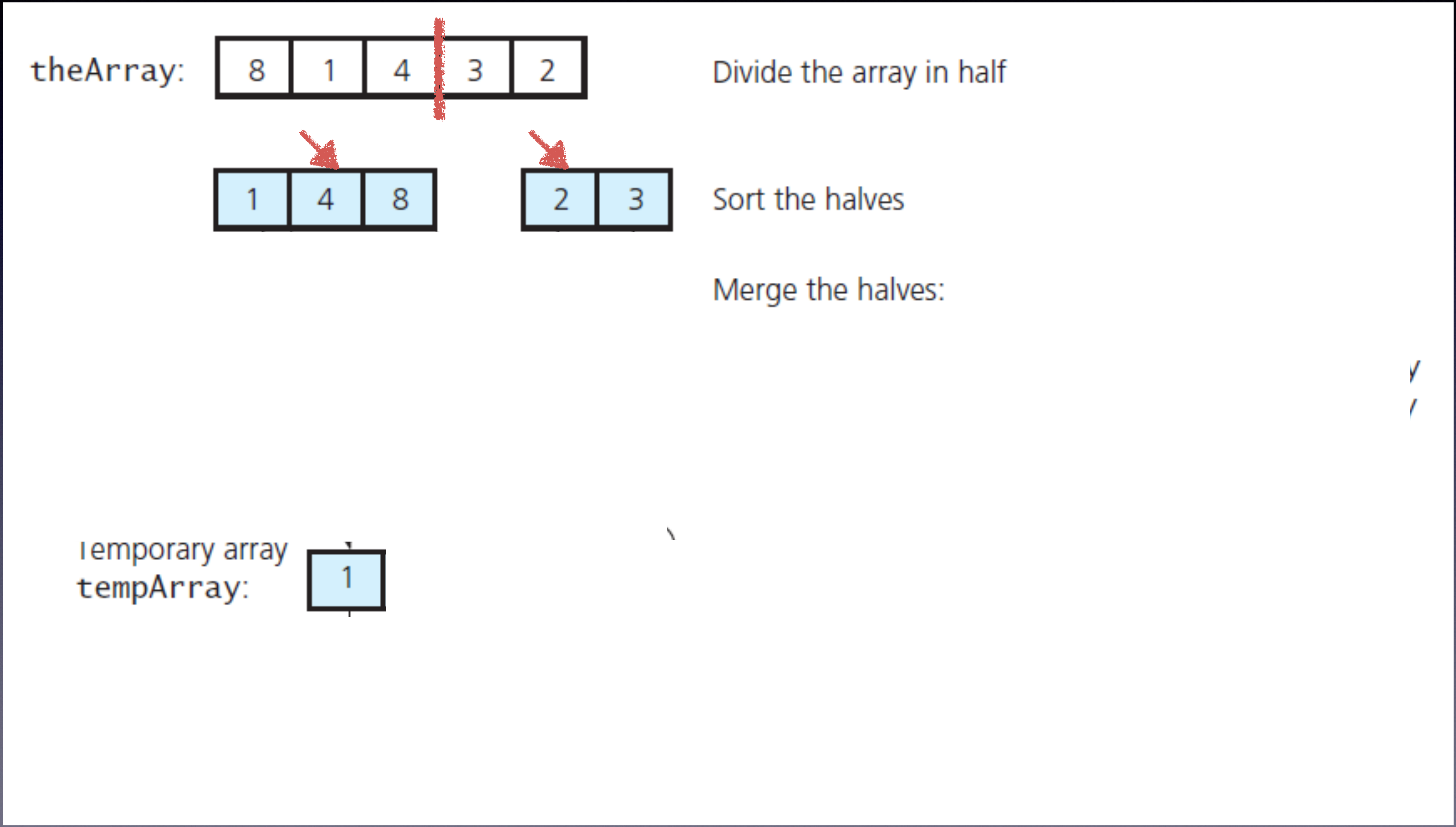
Merge the halves:

merge sort

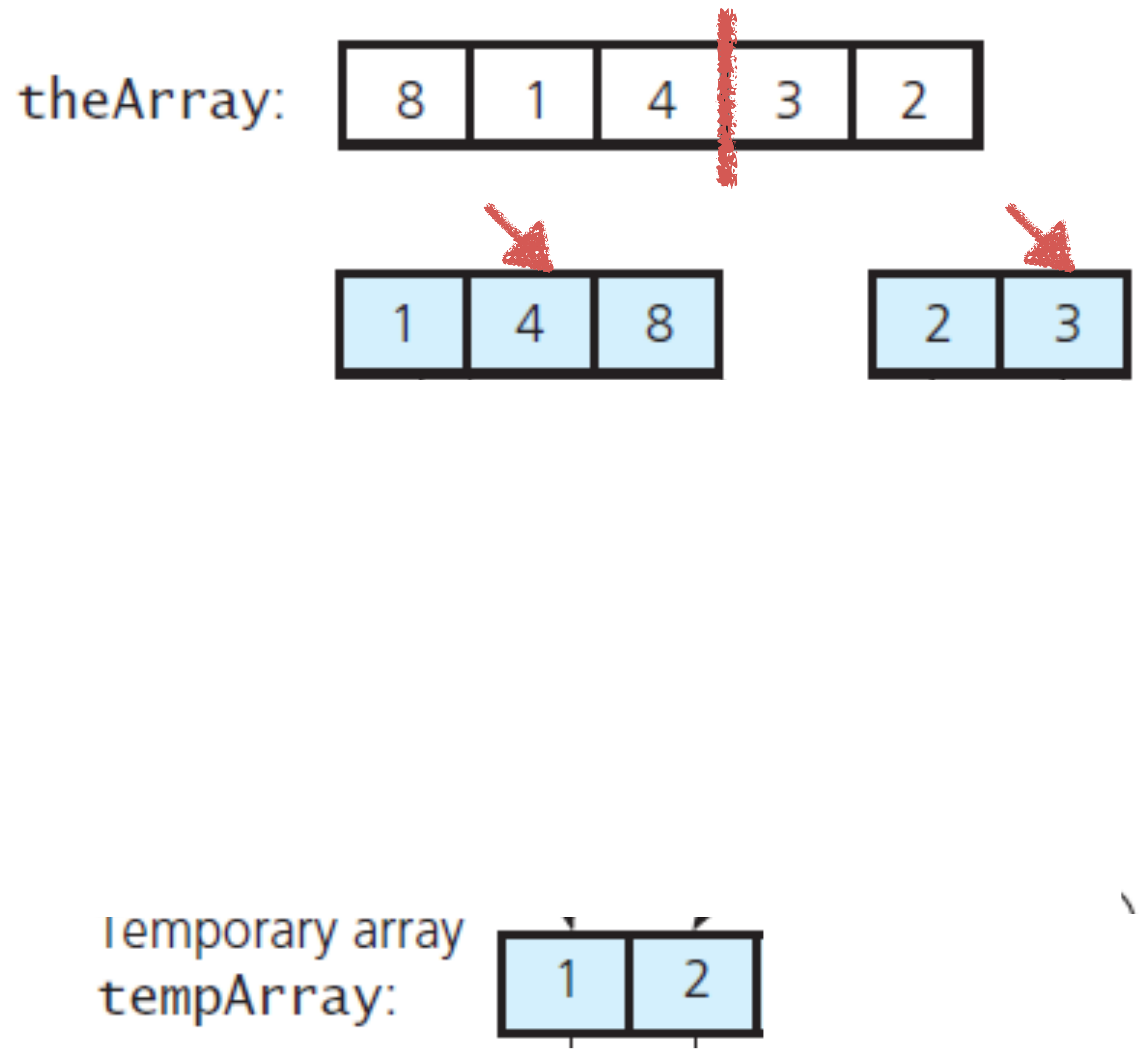


theArray:
8
1
4
3
2
Divide the array in half
1
4
8
2
3
Sort the halves
Merge the halves:
Temporary array
tempArray:
1

merge sort


theArray:
8 1 4 3 2
Divide the array in half
1 4 8
2 3
Sort the halves
Merge the halves:
Temporary array
tempArray:
1

merge sort

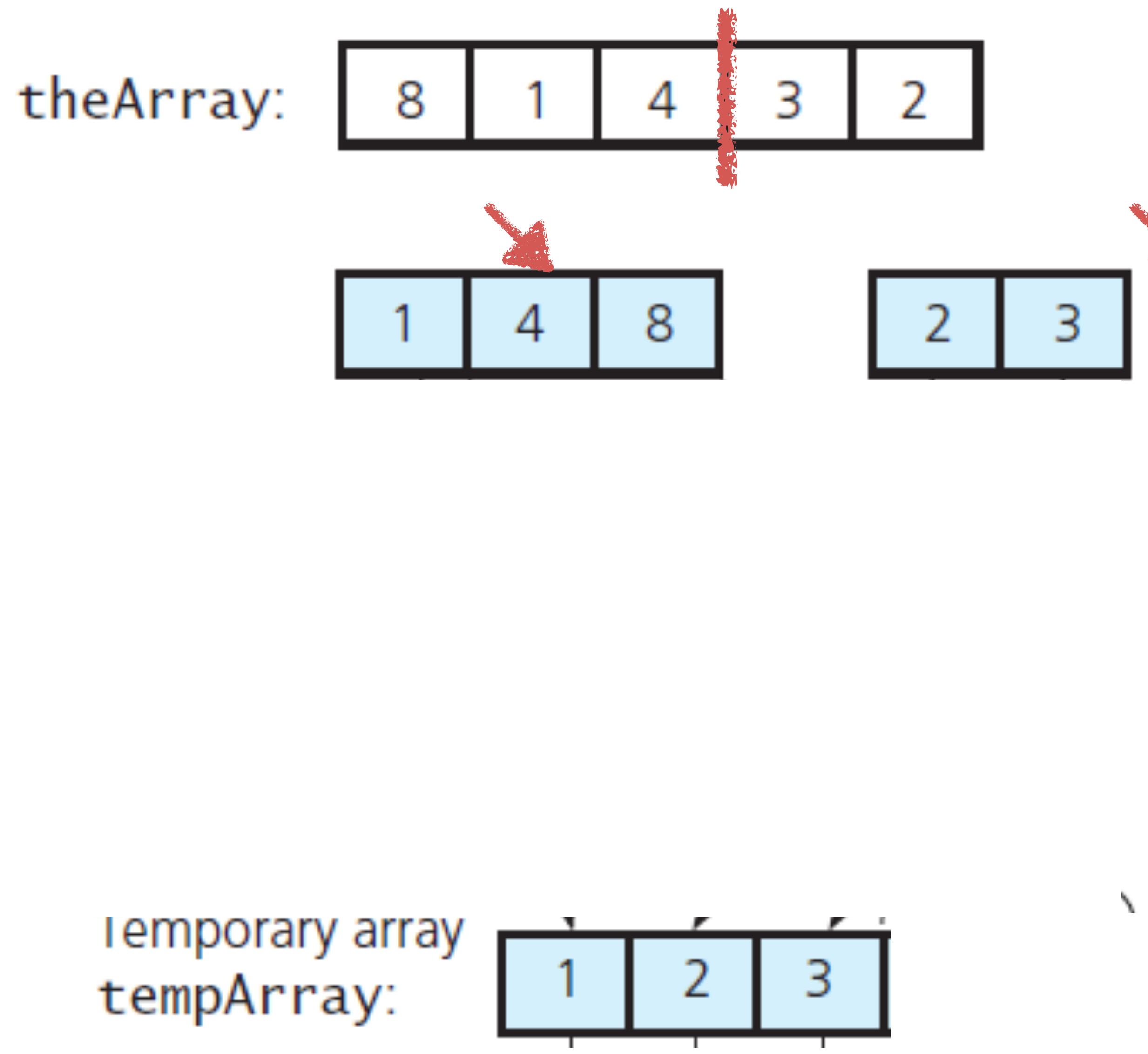


Divide the array in half

Sort the halves

Merge the halves:

merge sort

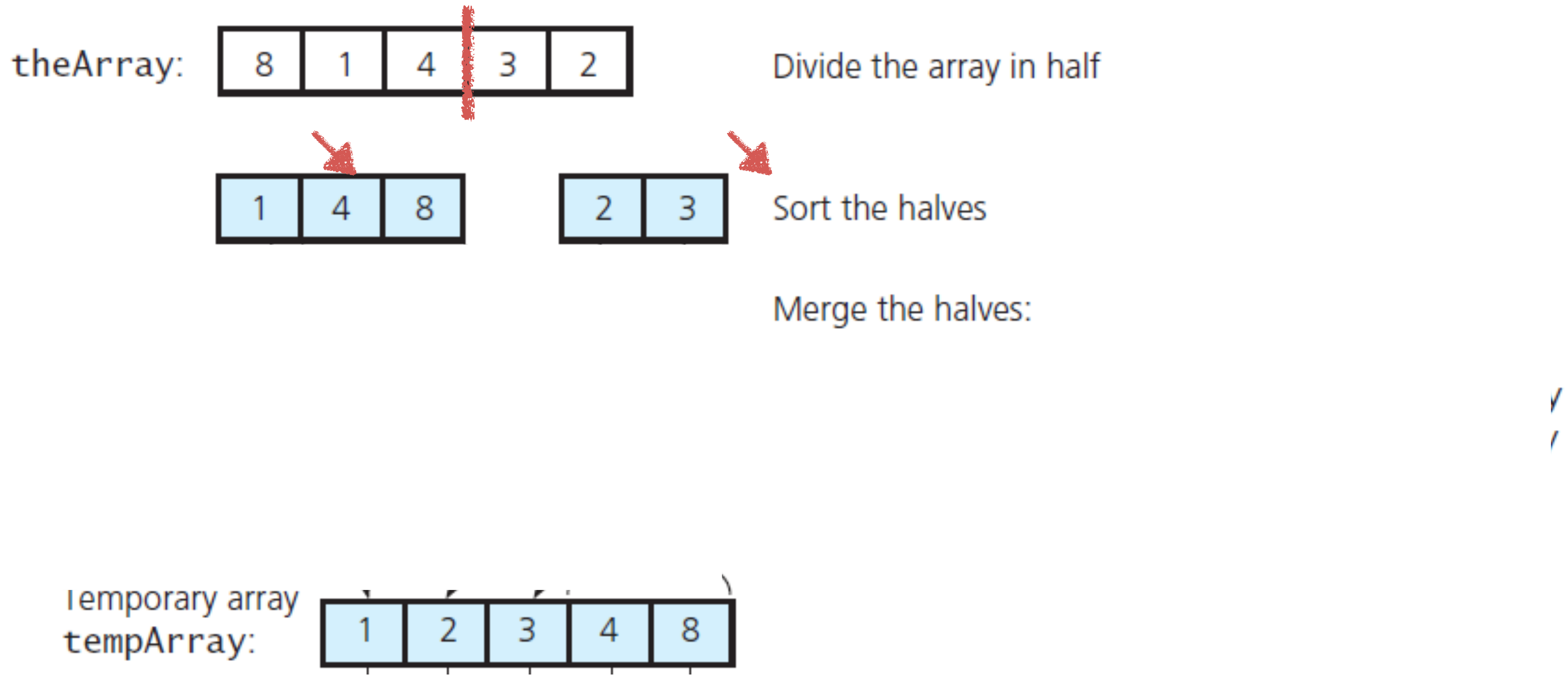


Divide the array in half

Sort the halves

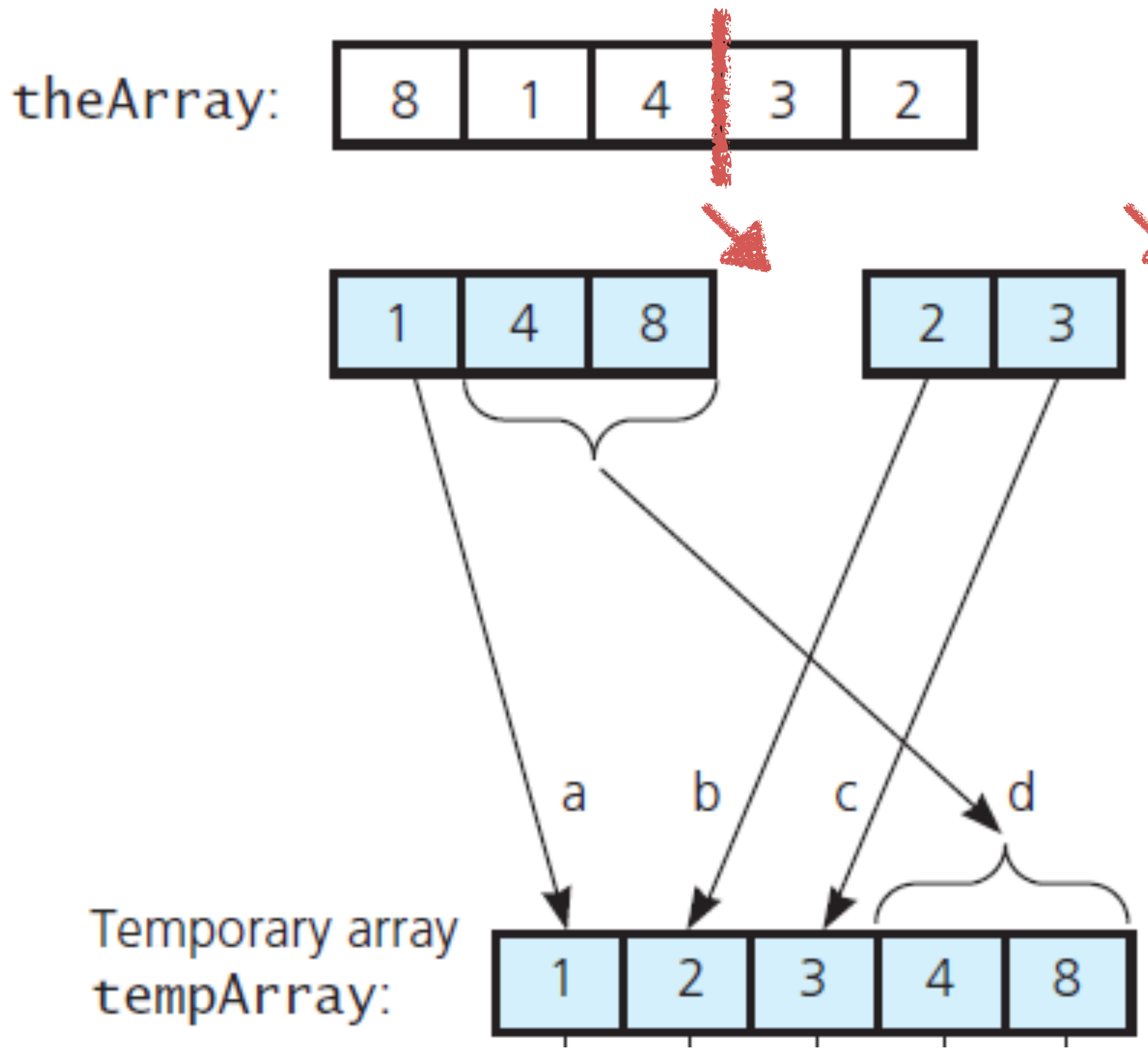
Merge the halves:

merge sort



theArray:
8
1
4
3
2
Divide the array in half
1
4
8
2
3
Sort the halves
Merge the halves:
Temporary array
tempArray:
1
2
3
4
8

merge sort



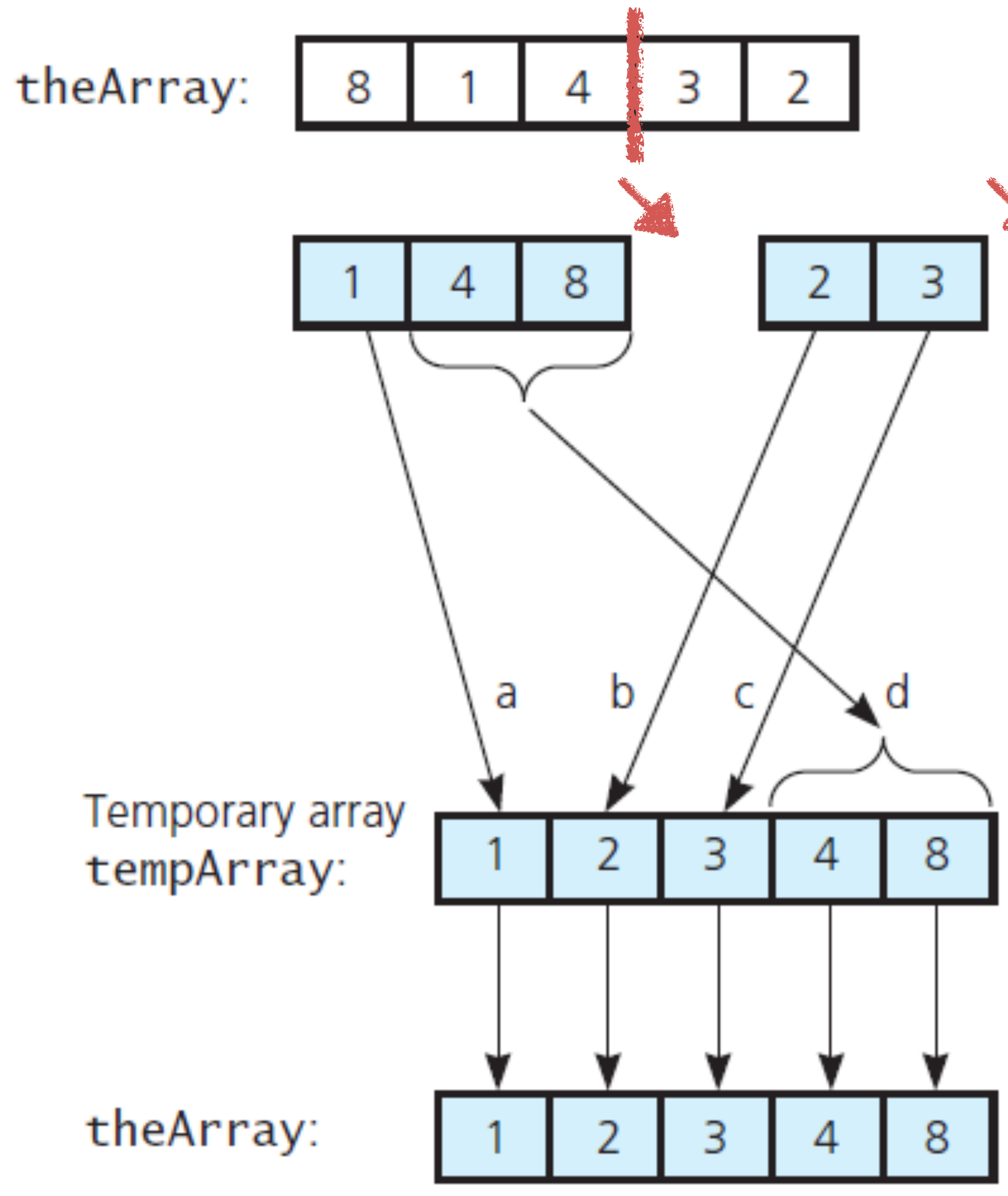
Divide the array in half

Sort the halves

Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

merge sort



Divide the array in half

Sort the halves

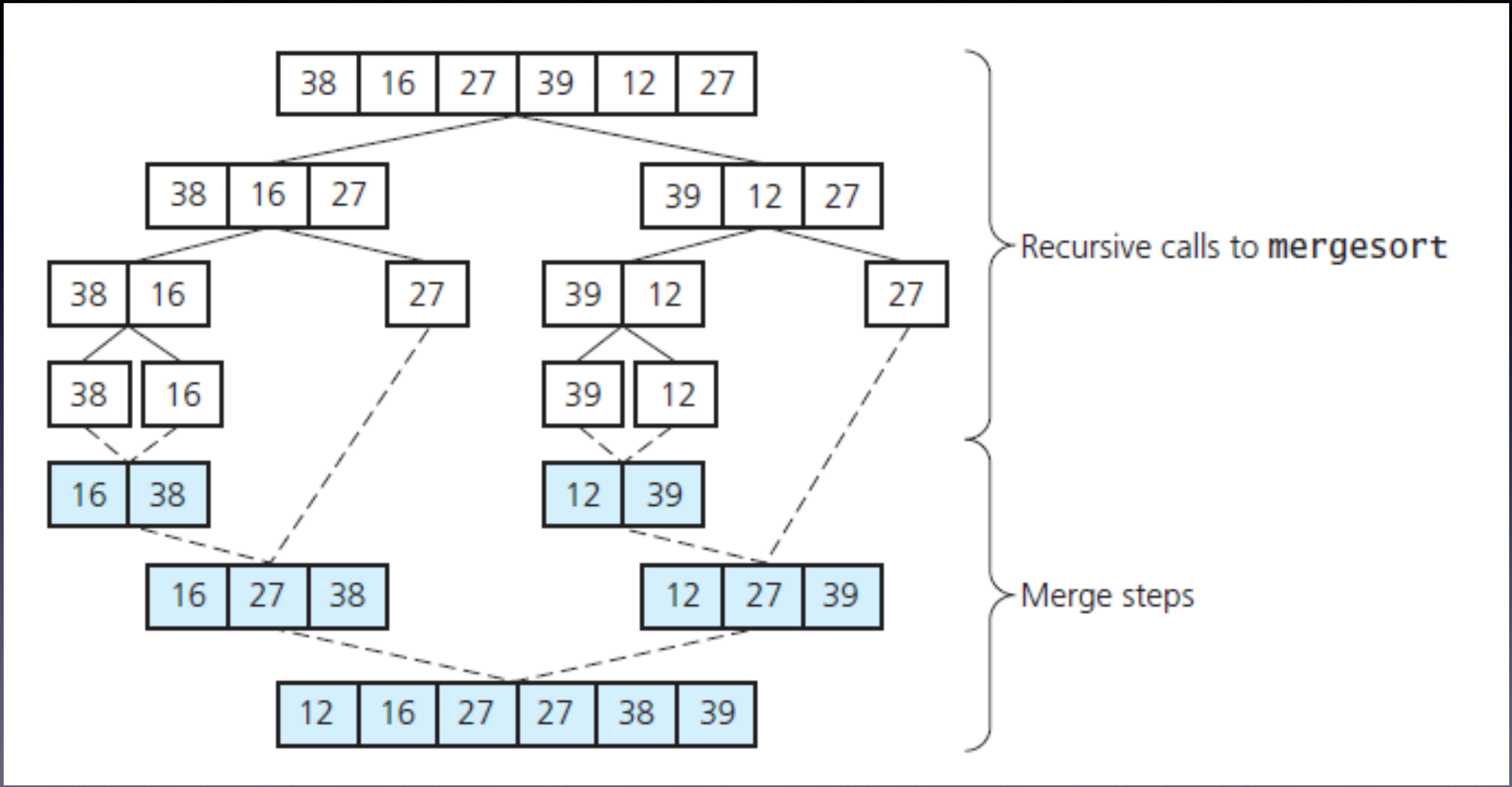
Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

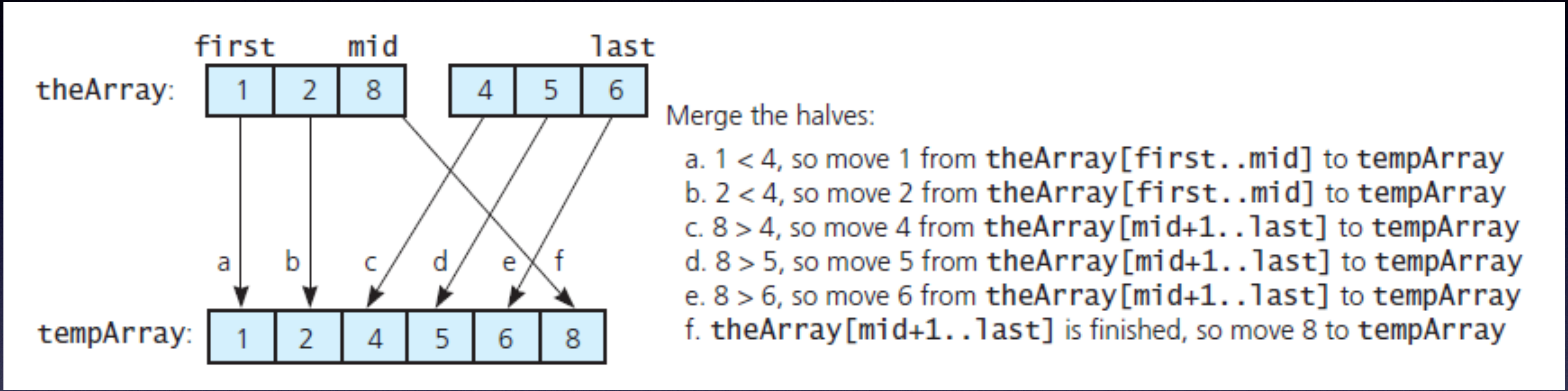
merge sort

```
// perform mergeSort on theArray between indices first and last
mergeSort(theArray: ItemArray, first: integer, last: integer)
{
  if (first<last)
  {
    mid = (first+last) / 2

    // recursively sort first half of array
    mergeSort(theArray, first, mid)

    // recursively sort second half of array
    mergeSort(theArray, mid+1, last)

    // merge sorted halves (first thru mid, and mid+1 thru last)
    merge(theArray, first, mid, last)
  }
}
```

merge sort


38 16 27 39 12 27
38 16 27
39 12 27
38 16
27
39 12
27
38
16
39
12
16 38
12 39
16 27 38
12 27 39
12 16 27 27 38 39
Recursive calls to mergesort
Merge steps

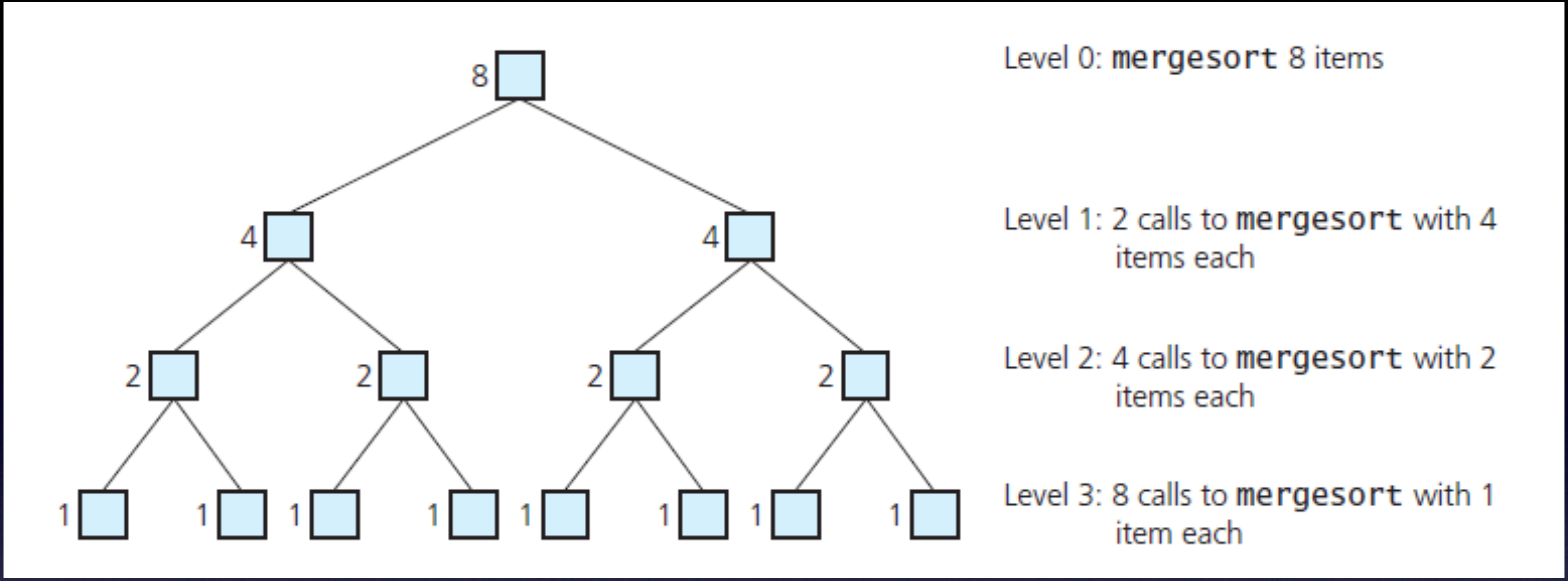
merge analysis



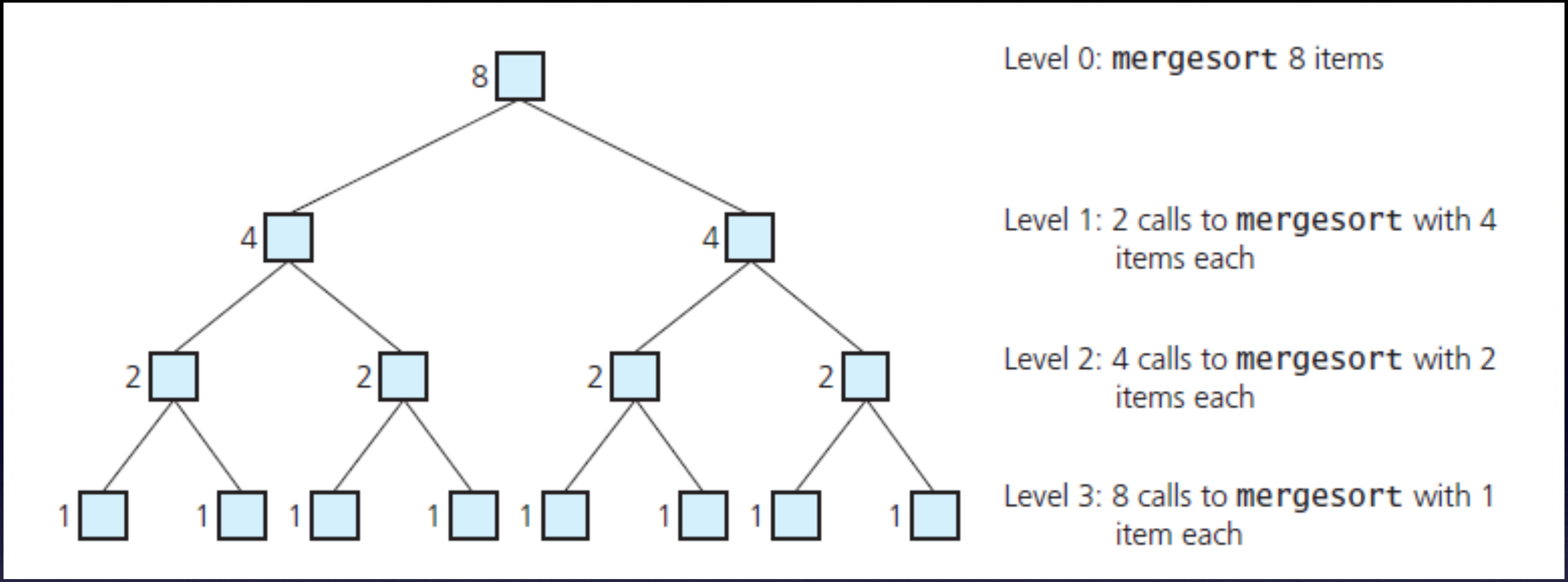
Merge the halves:

a. 1 < 4, so move 1 from theArray[first..mid] to tempArray
b. 2 < 4, so move 2 from theArray[first..mid] to tempArray
c. 8 > 4, so move 4 from theArray[mid+1..last] to tempArray
d. 8 > 5, so move 5 from theArray[mid+1..last] to tempArray
e. 8 > 6, so move 6 from theArray[mid+1..last] to tempArray
f. theArray[mid+1..last] is finished, so move 8 to tempArray

- At most n-1 comparisons, n moves to tempArray, and then n moves from tempArray back to theArray
- Total: 3n-1 operations


8
4
4
2
2
2
2
1
1
1
1
1
1
1
1
Level 0: mergesort 8 items
Level 1: 2 calls to mergesort with 4 items each
Level 2: 4 calls to mergesort with 2 items each
Level 3: 8 calls to mergesort with 1 item each

- Each call to mergeSort halves the array -- first to 2 pieces, then 4 pieces, ... , n pieces. For n=8, there are 3 levels.
- If n is a power of 2, then there are exactly $k = \log_2 n$ levels. If not, then there are at most $k = \log_2 n+1$ levels.


8
4
4
2
2
2
2
1
1
1
1
1
1
1
1
Level 0: mergesort 8 items
Level 1: 2 calls to mergesort with 4 items each
Level 2: 4 calls to mergesort with 2 items each
Level 3: 8 calls to mergesort with 1 item each

- Level 0 calls merge once, making 3n-1 operations. Level 1 calls merge two times, for 3(n/2) - 1 ops each time, or 3n-2 ops.
- On level m, there are 2^m calls to merge, each of which merges $n/2^m$ items, needing $3(n/2^m) -1$ ops each. Together the 2^m calls to merge require $3n-2^m$ ops. So each level requires O(n) ops.
- Since there are $\log_2 n$ or $\log_2 n + 1$ levels, and each level takes O(n) ops, the total performance is **O(n log n)**

n	merge sort O(n log n)	Bubble sort $O(n^2)$
1024	1024*10 (about 10000)	1024*1024 (about 1,000,000)