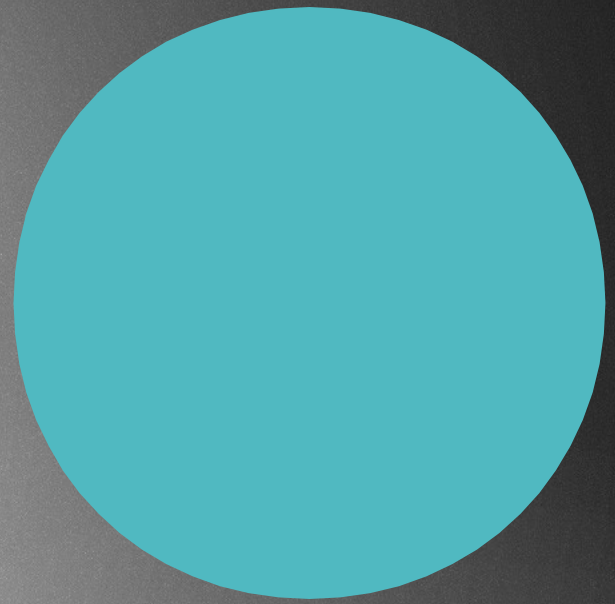
# SORTING ALGORITHMS



# MERGESORT

# Mergesort

- Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945
- Comparison based algorithm with running time complexity **O(N logN)**
- It is a stable sorting algorithm
- Not an in-place algorithm !!!
- Although heapsort has the same time bounds as merge sort → heapsort requires only **Θ(1)** auxiliary space instead of merge sort's **Θ(*n*)**
- Efficient quicksort implementations generally outperforms mergesort
- Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only **Θ(1)** extra space

| | Quicksort | Mergesort |
|---|---|---|
| **In place** | **Yes** | **No** |
| **Stable** | **No** | **Yes** |
| **Time complexity** | **Quadratic sometimes** | **O(N logN)** |

# Mergesort

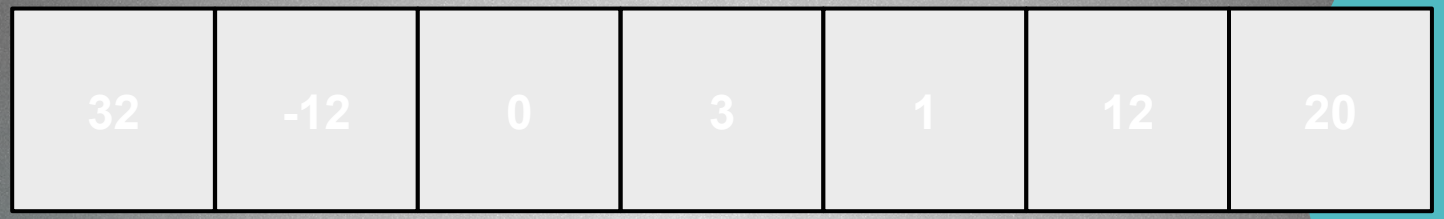
**1.)** divide the array into two subarrays recursively

**2.)** sort these subarrays recursively with mergesort again

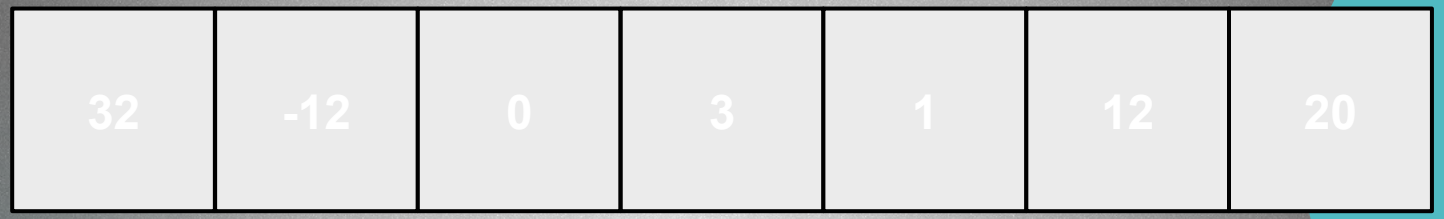
**3.)** if there is only a single item left in the subarray → we consider it to be sorted by definition

**4.)** merge the subarrays to get the final sorted array

# Mergesort „divide”



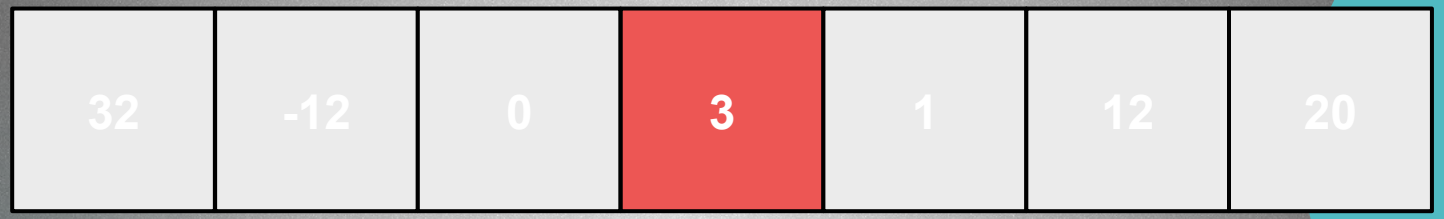
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

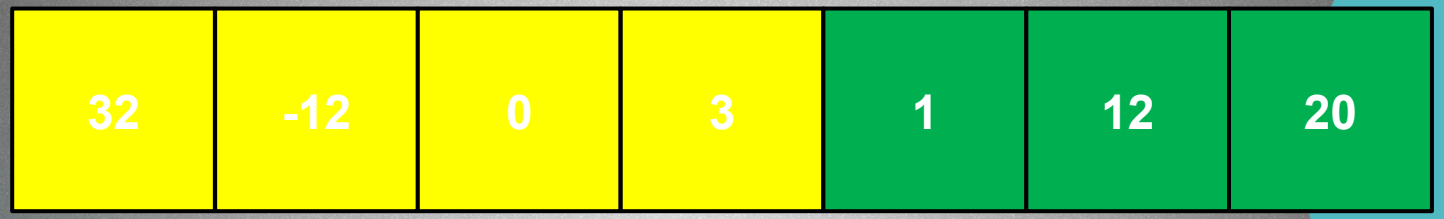
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

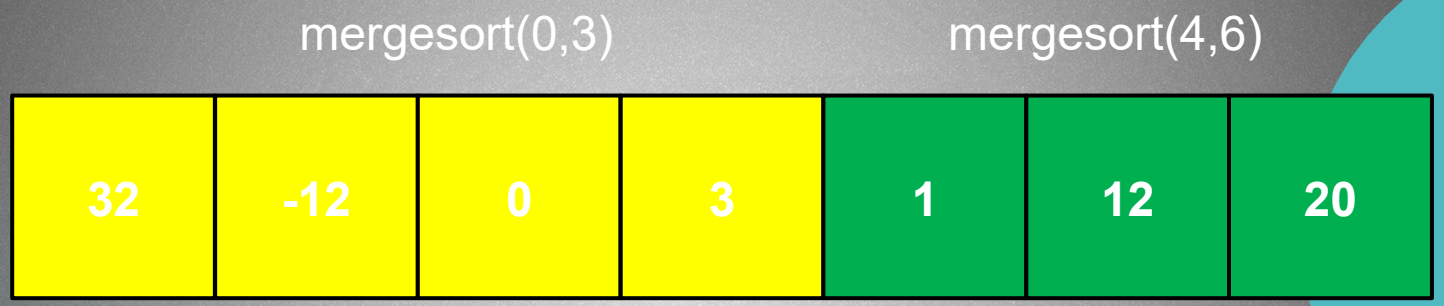
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

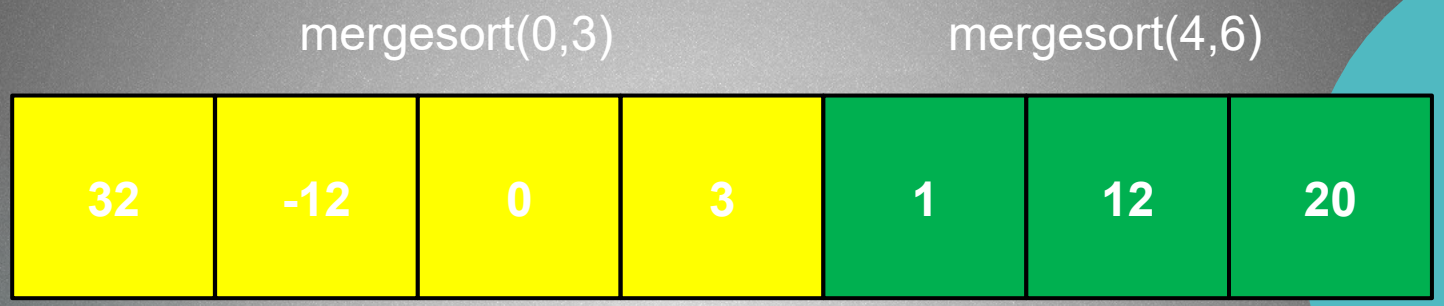
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

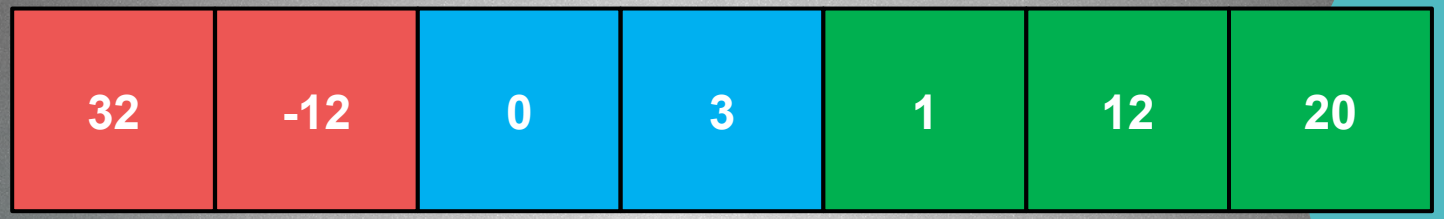
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

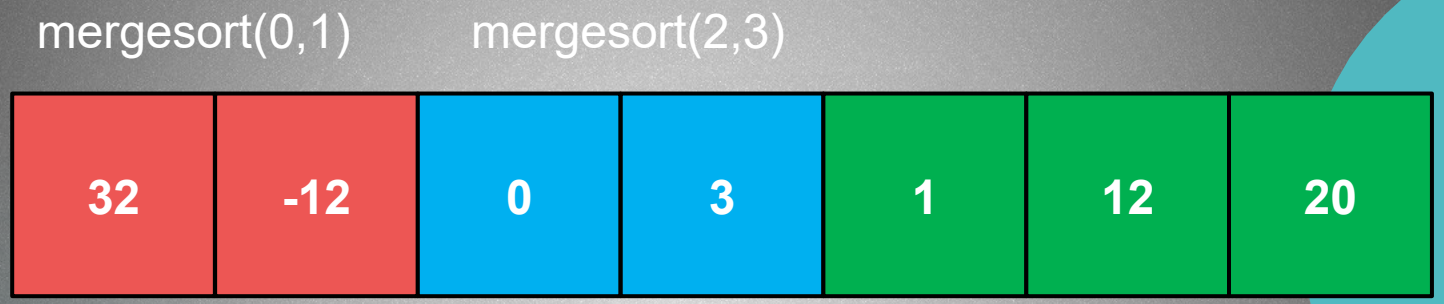
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

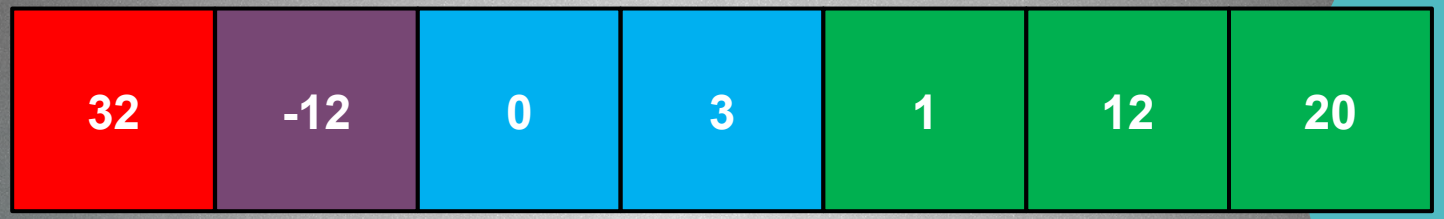
# Mergesort „divide”




We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

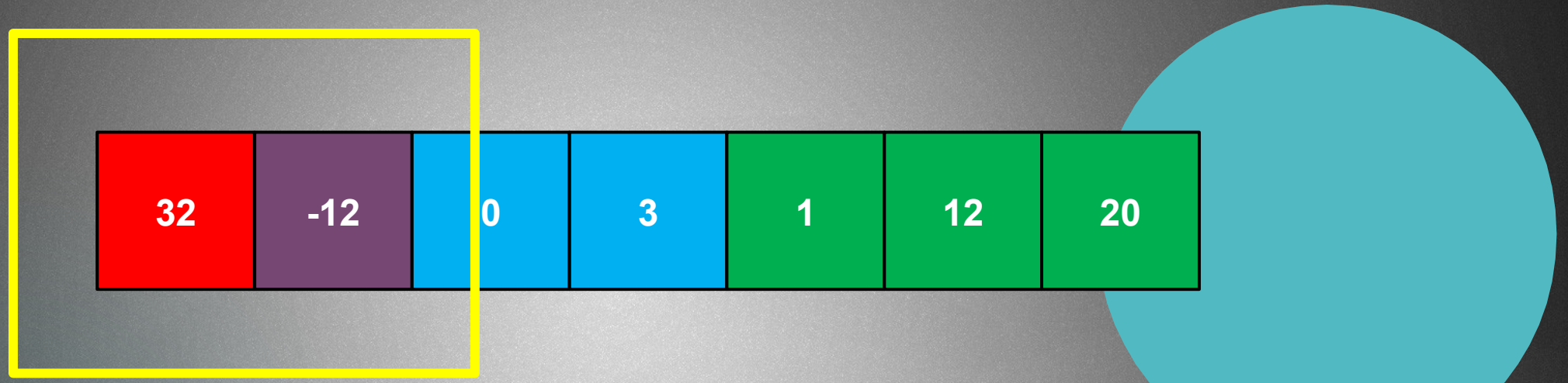
# Mergesort „divide”



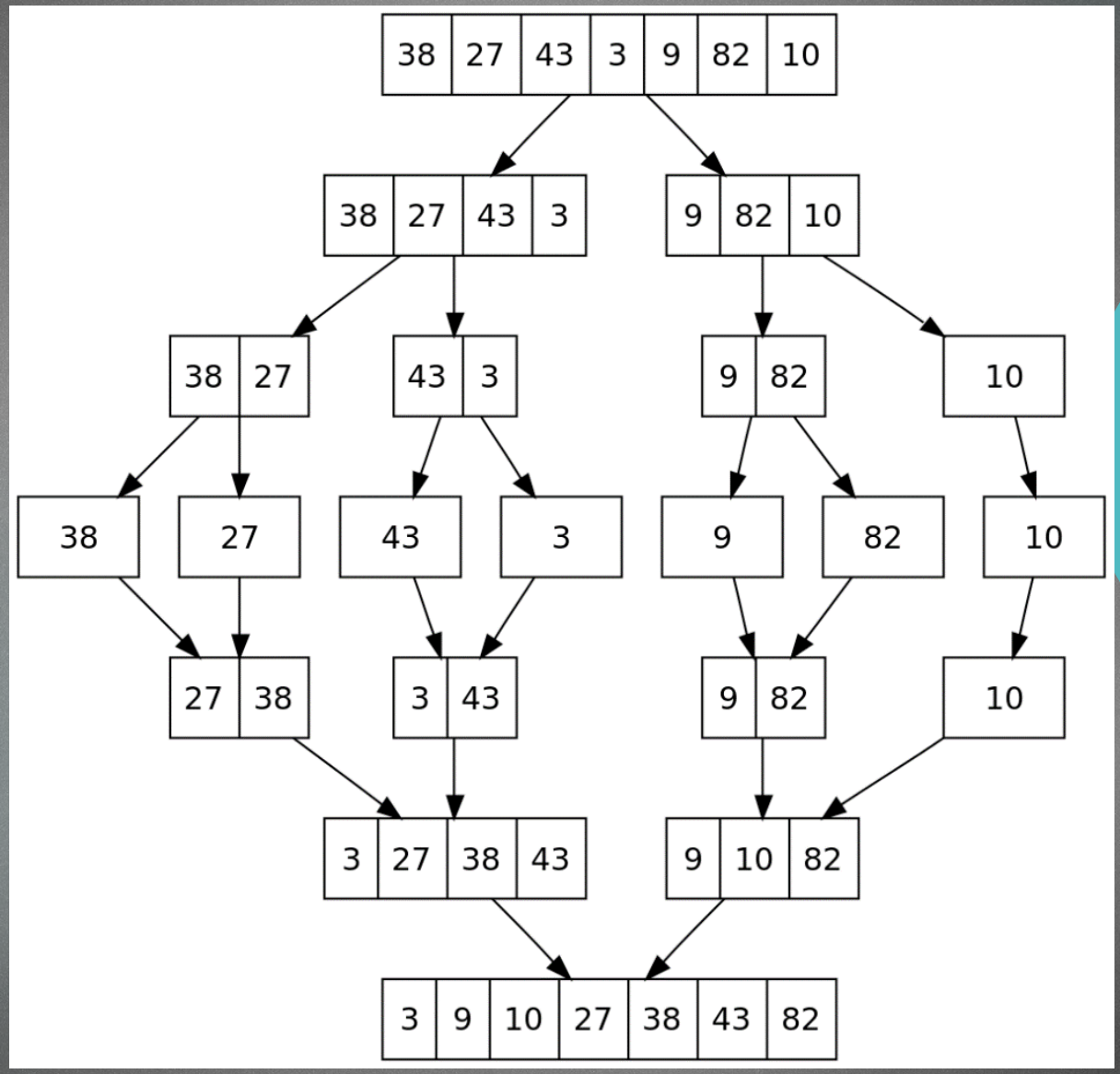

We call the sort method over and over again: mergesort(0,6)

Generate a middle index in order to partition the array: middleIndex = (low+high) / 2

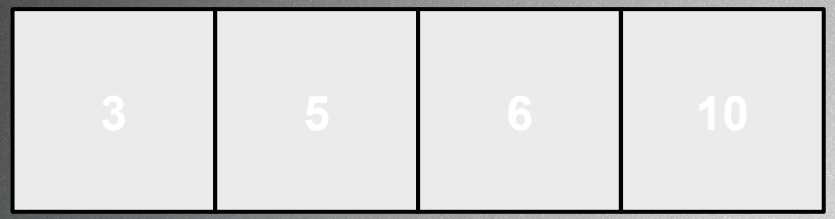
# Mergesort „divide”

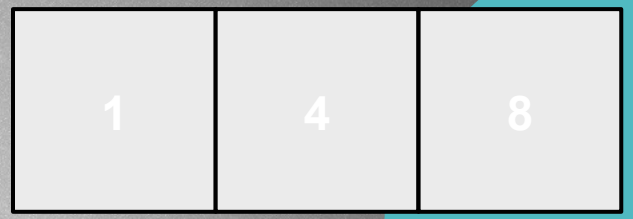



After several recursive method calls: we end with single items,
we consider them sorted by default
~ so we keep merging these already sorted items


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

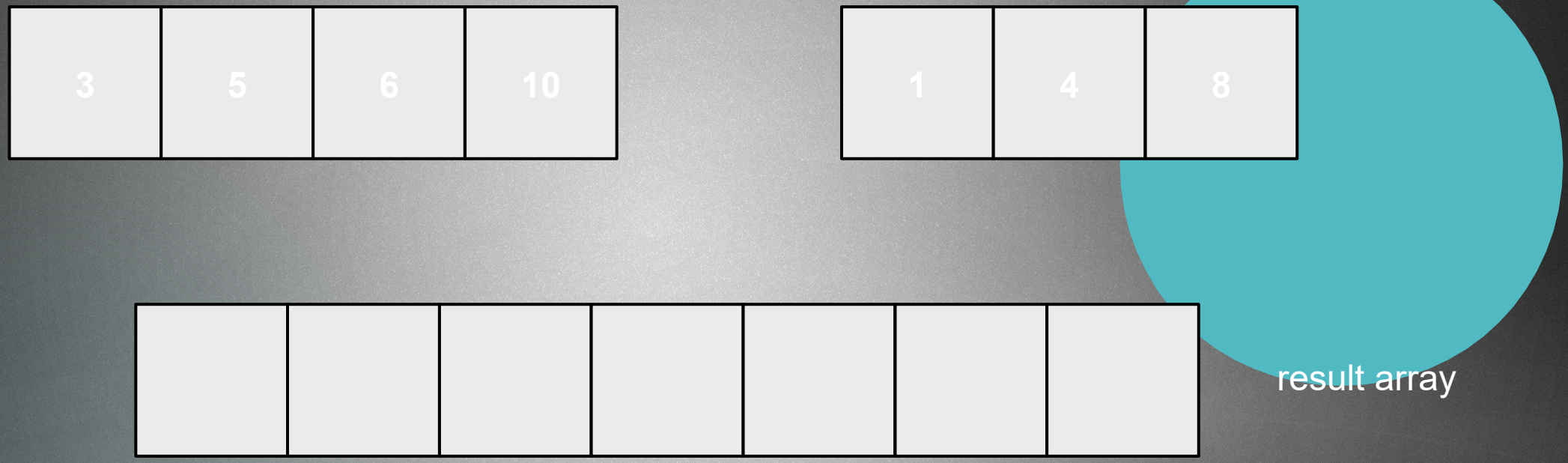
# Mergesort „conquer”



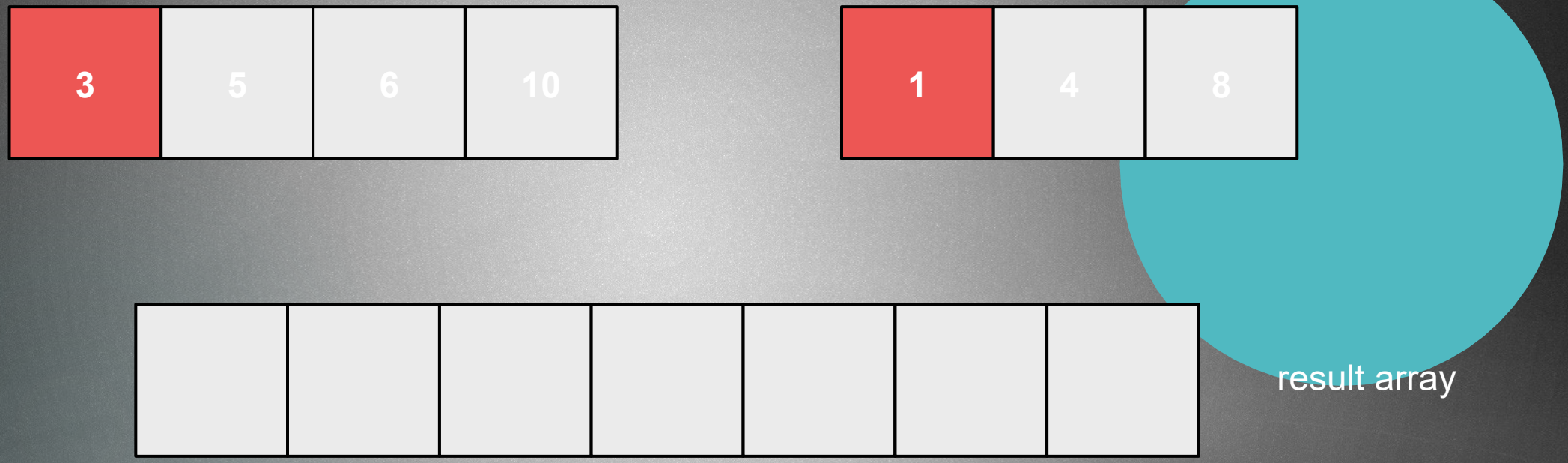




So after the split operations: we have several distinct arrays that are already sorted: we have to merge these arrays into a single one

# Mergesort „conquer”

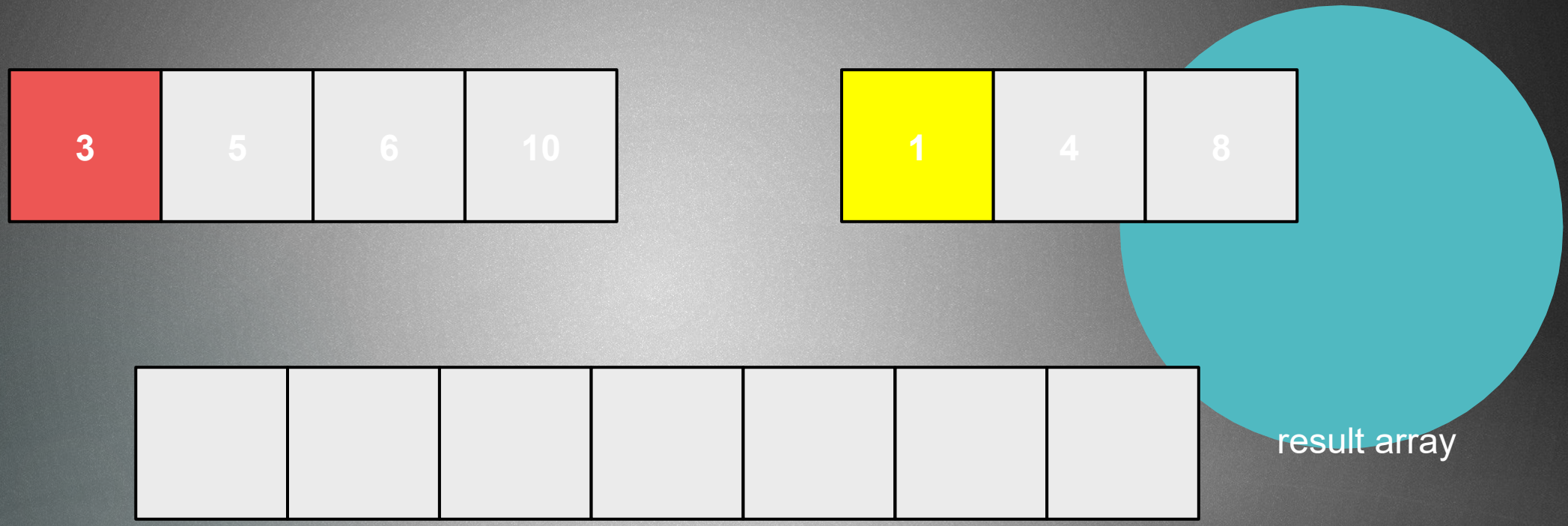


# Mergesort „conquer”



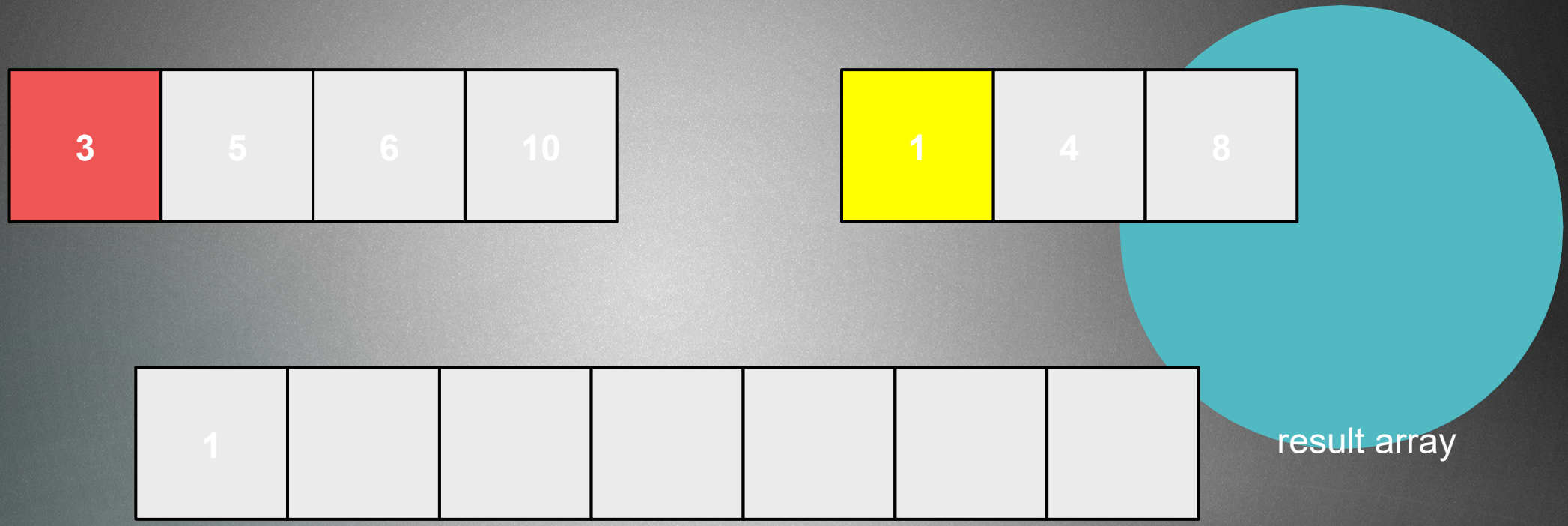

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



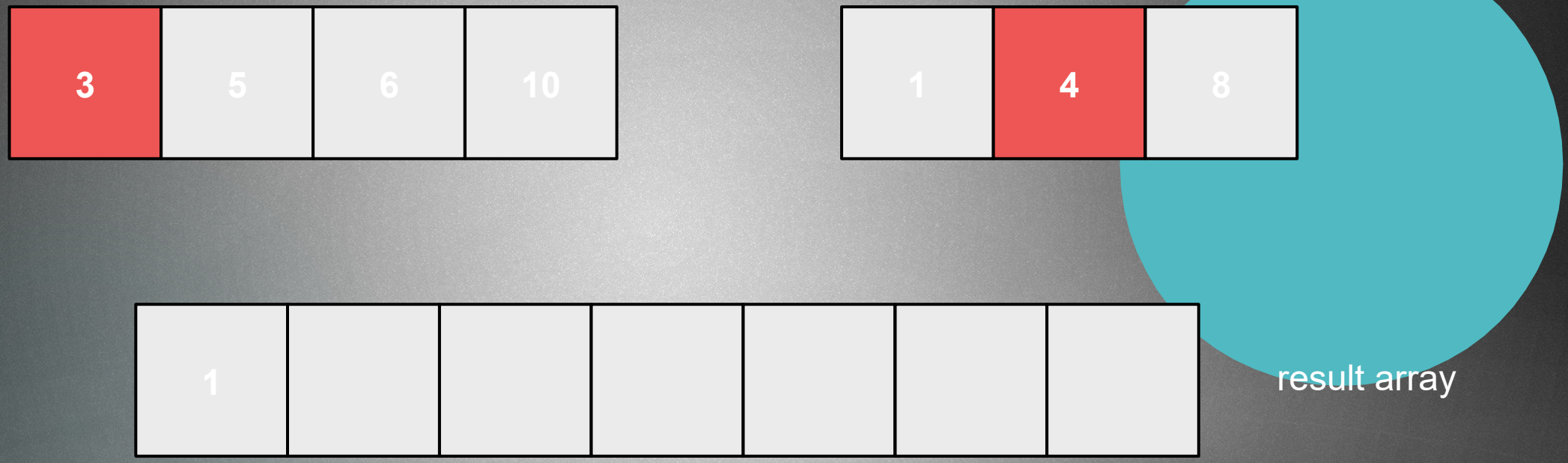

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



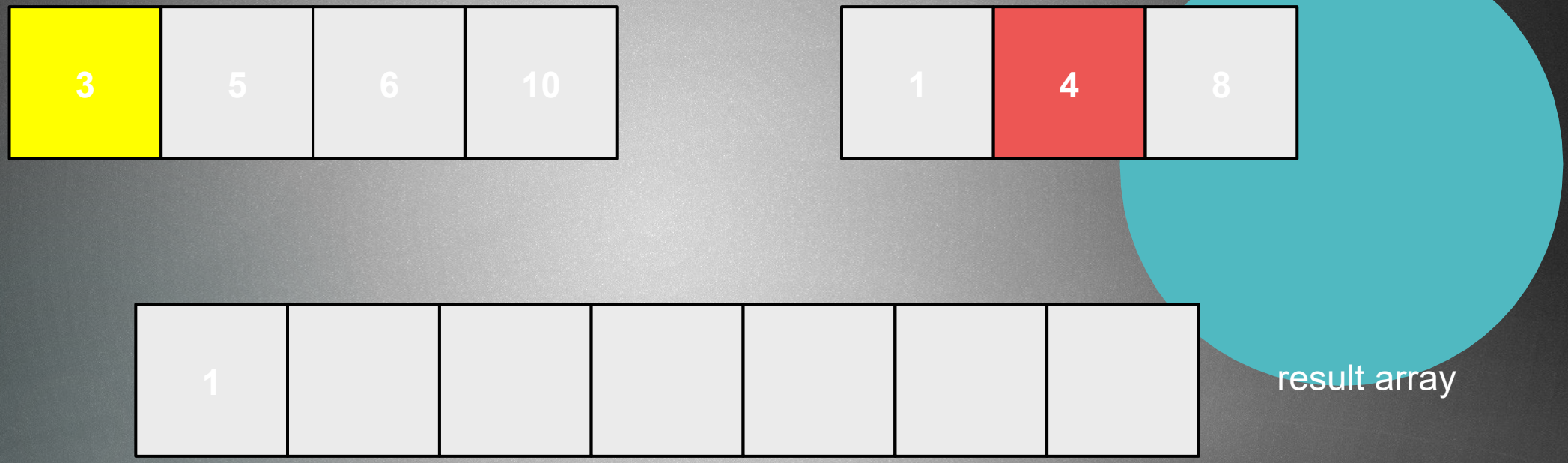

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



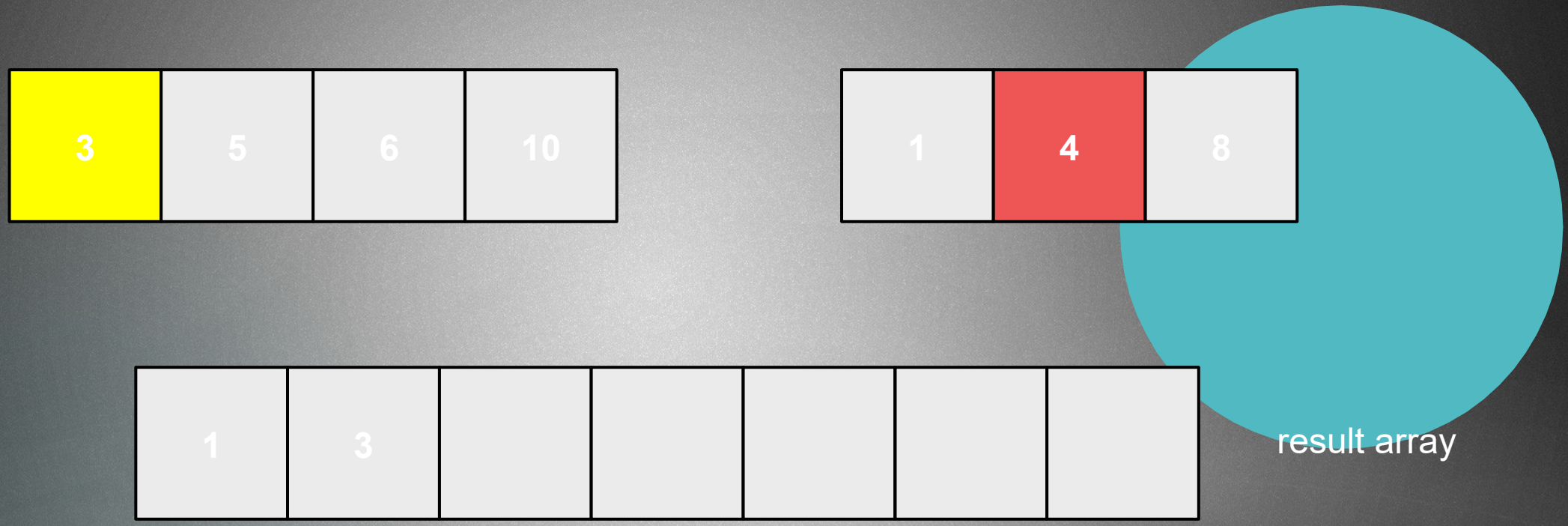

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



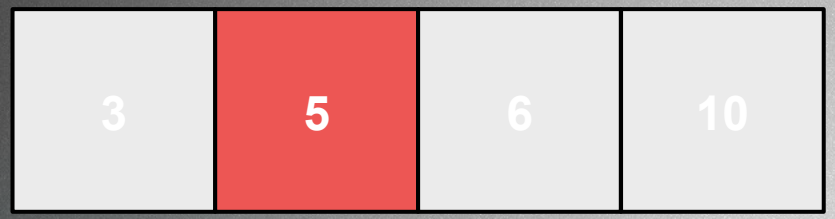

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”




We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”

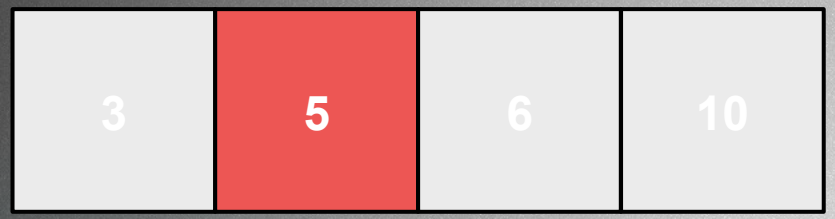





result array

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”

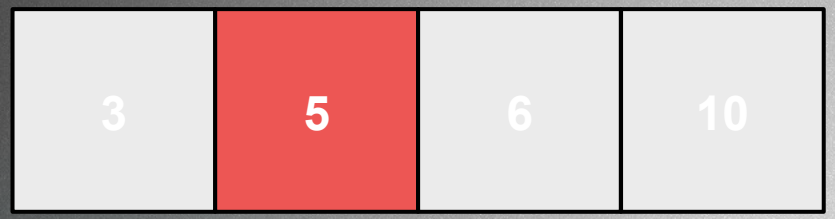


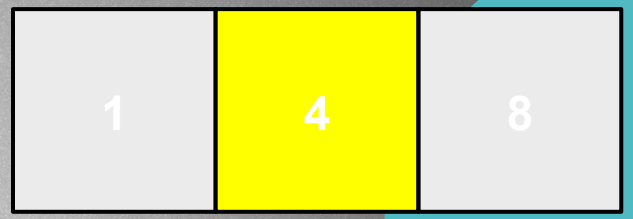



result array

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



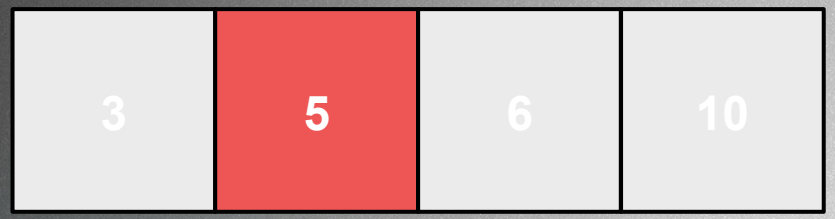



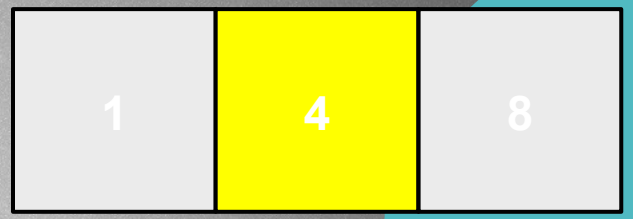


result array

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



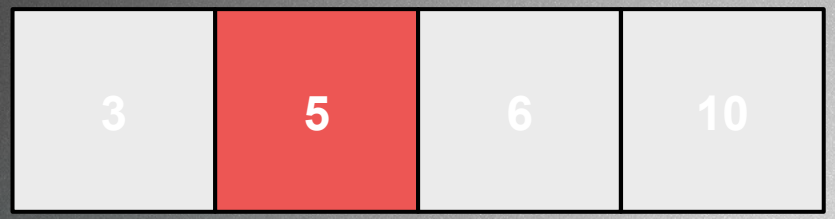



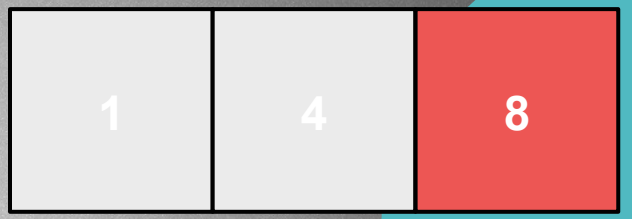


result array

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



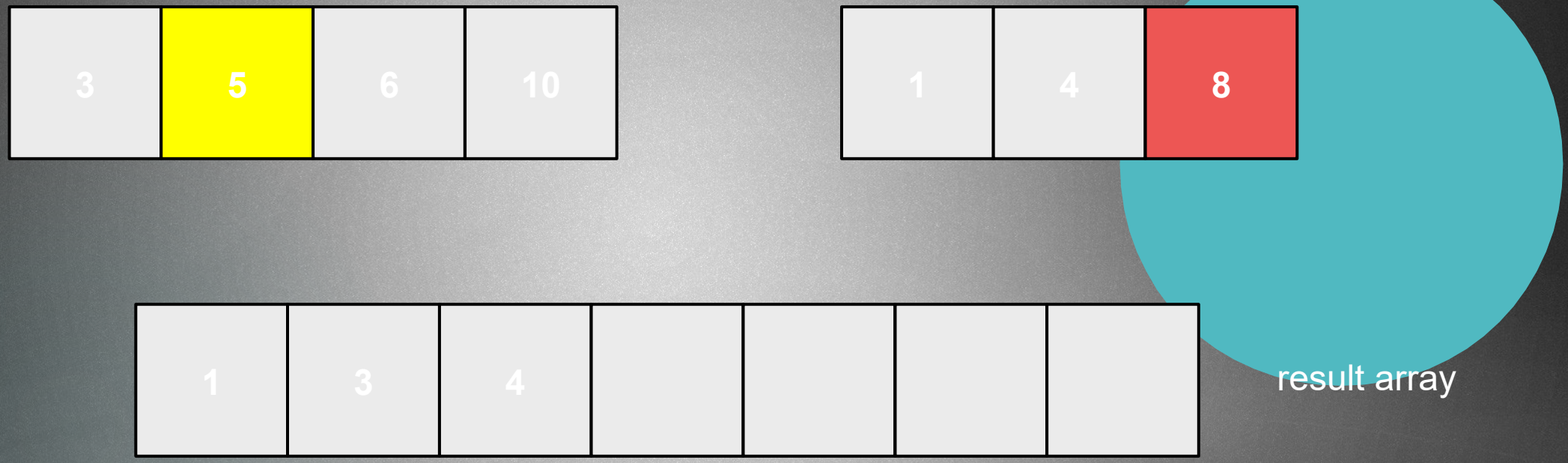





result array

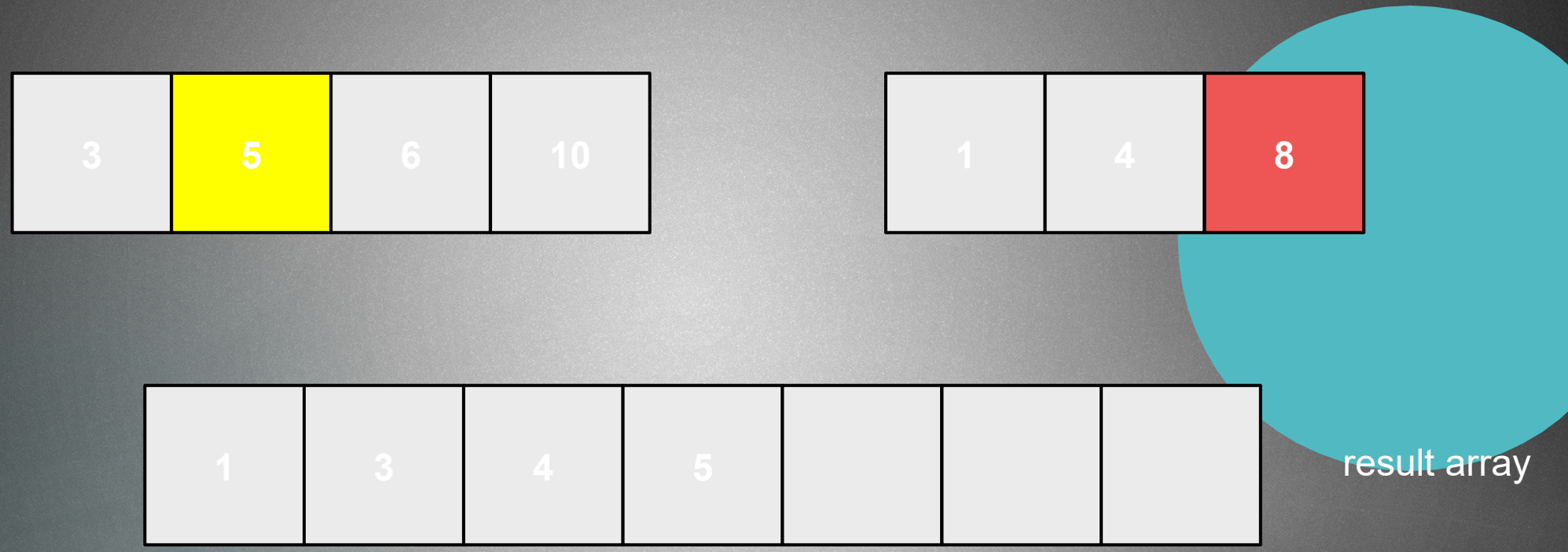
We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



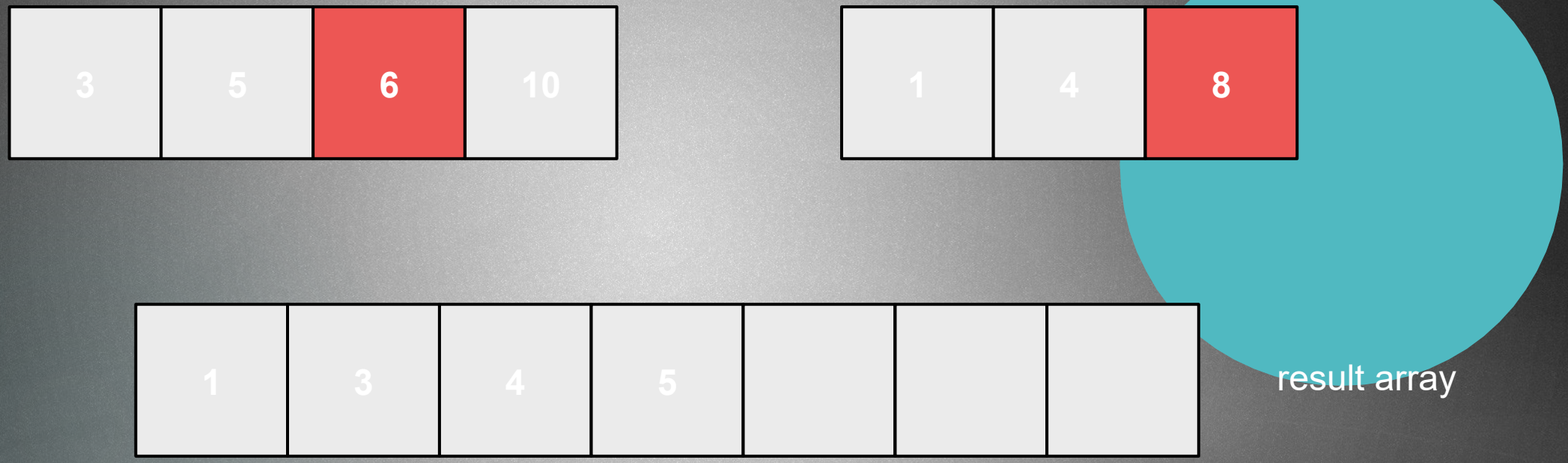

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



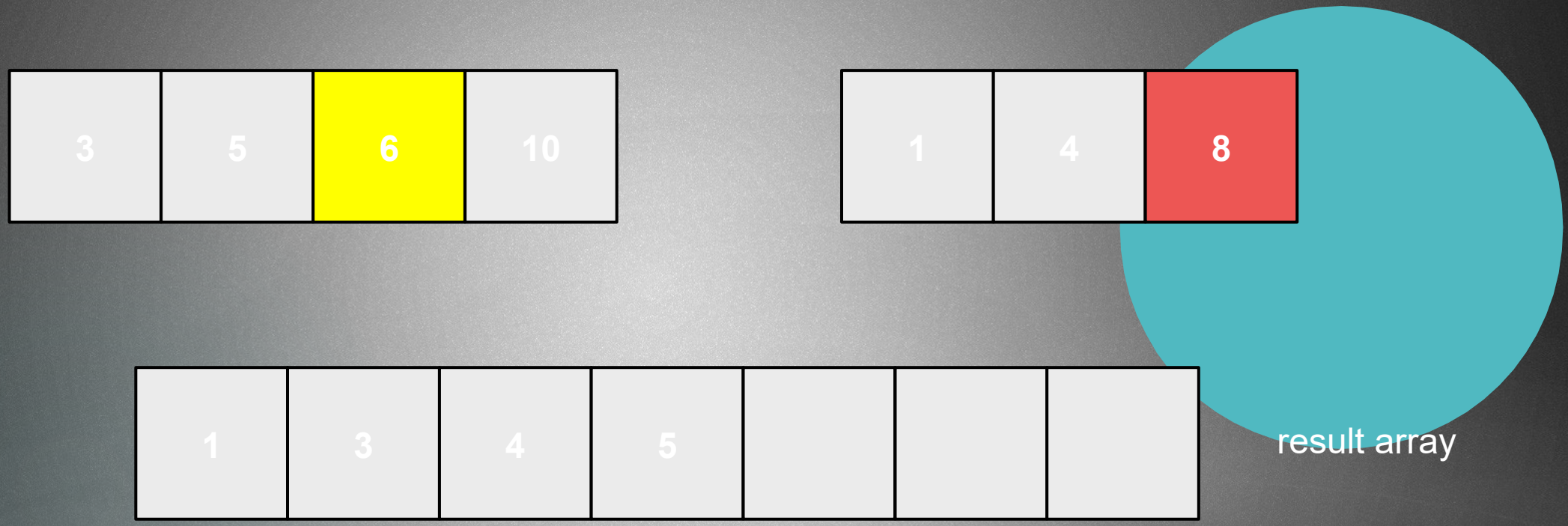

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



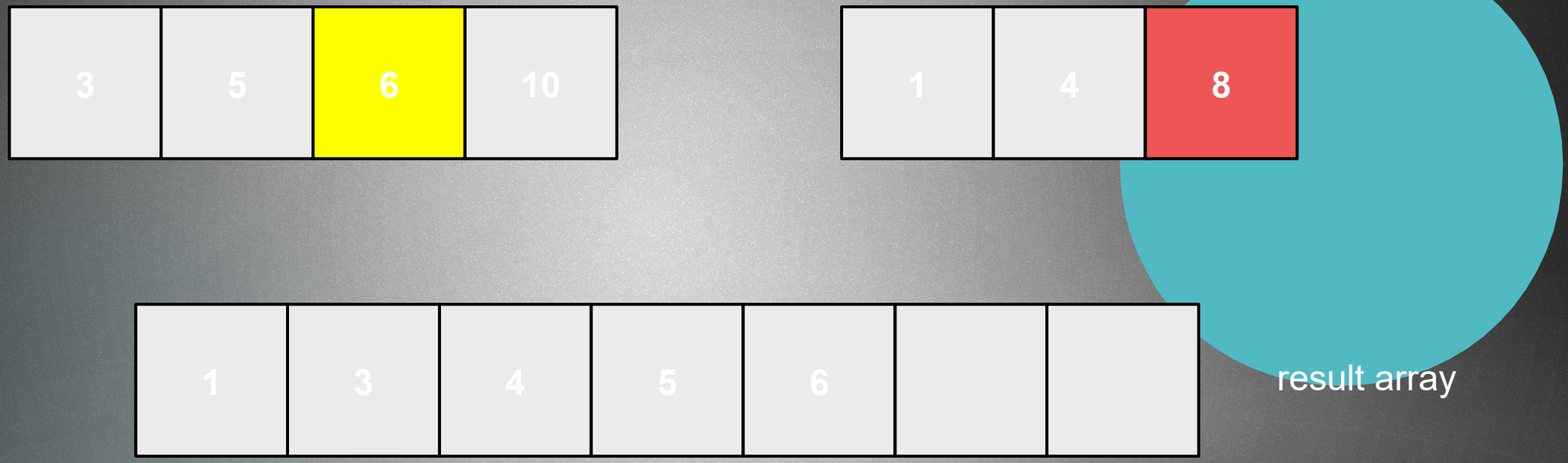

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



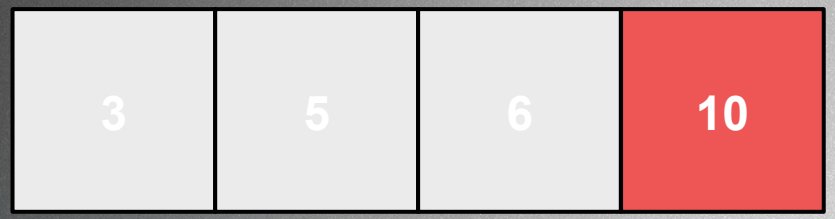

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

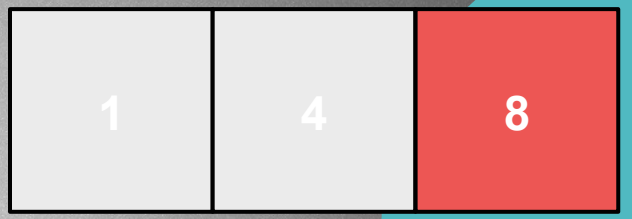
# Mergesort „conquer”

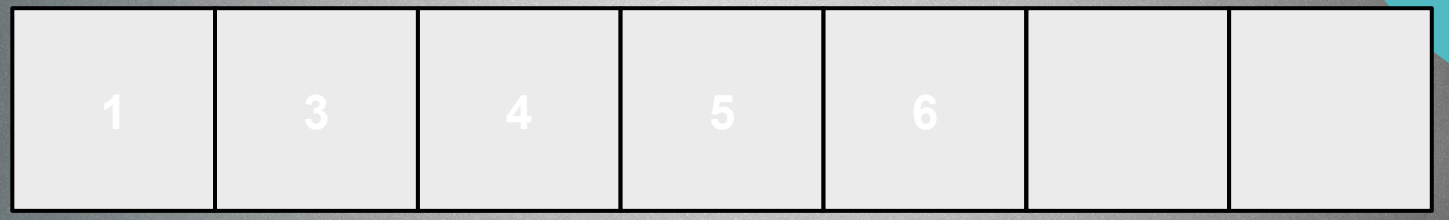



We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



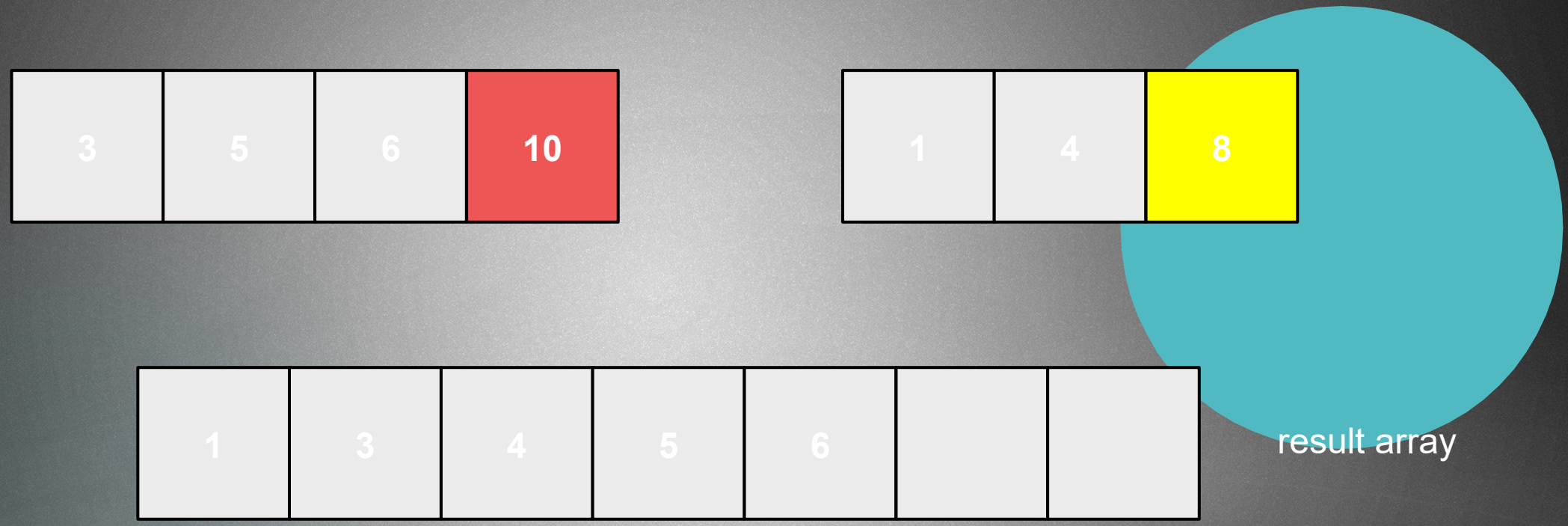




result array

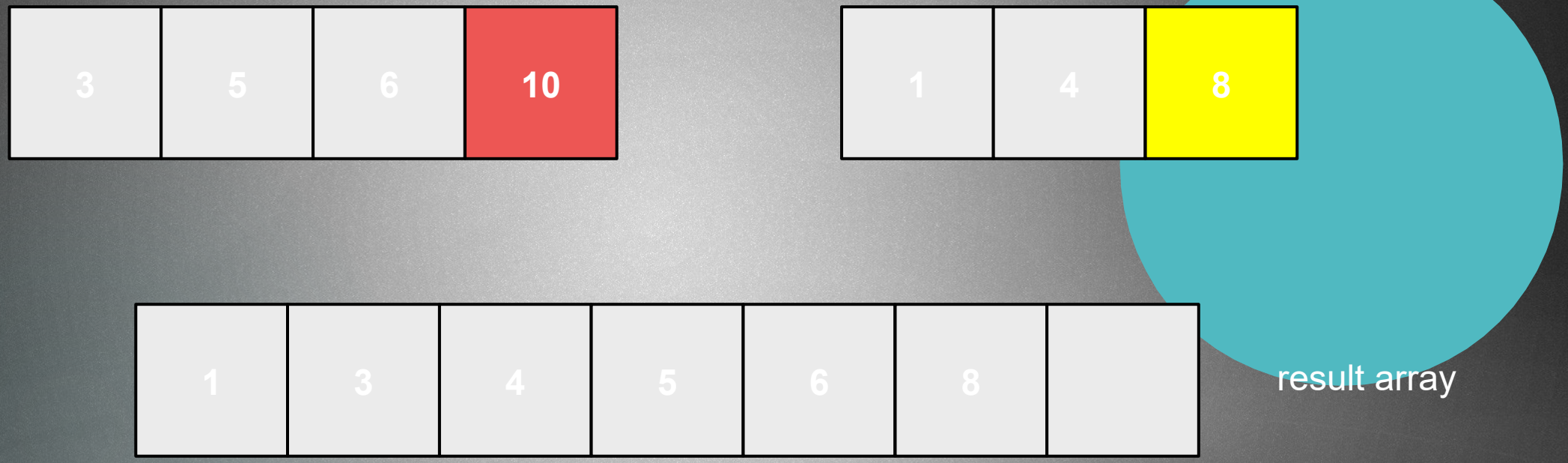
We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



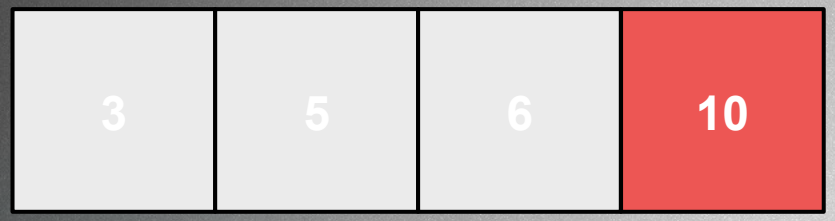

We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”

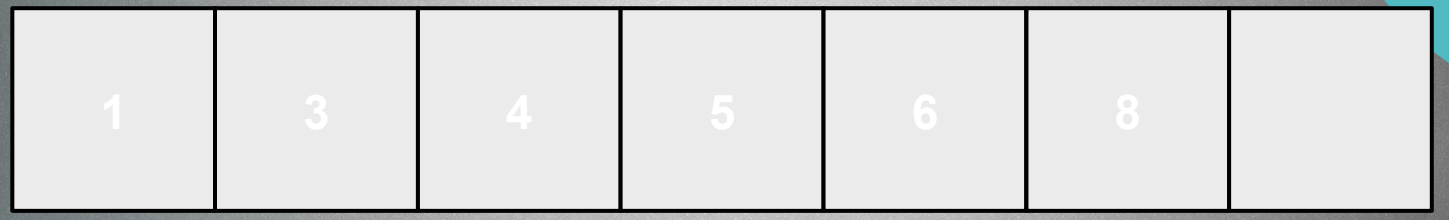



We start at the beginning of the subarrays: we keep comparing them, we insert the smaller into the result array

# Mergesort „conquer”



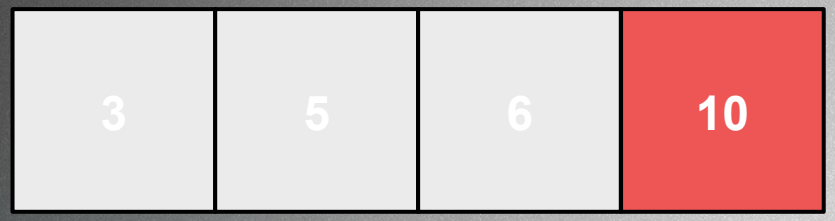




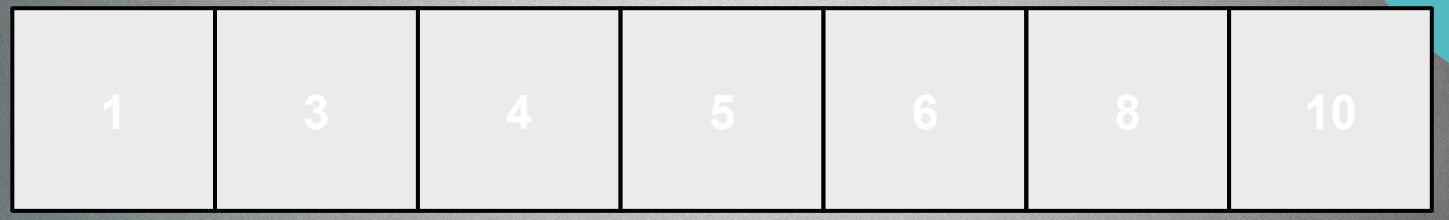

result array

**VERY IMPORTANT**: we have to iterate through the left and right array if there are some more items left → in this case the **10** in the left subarray

# Mergesort „conquer”



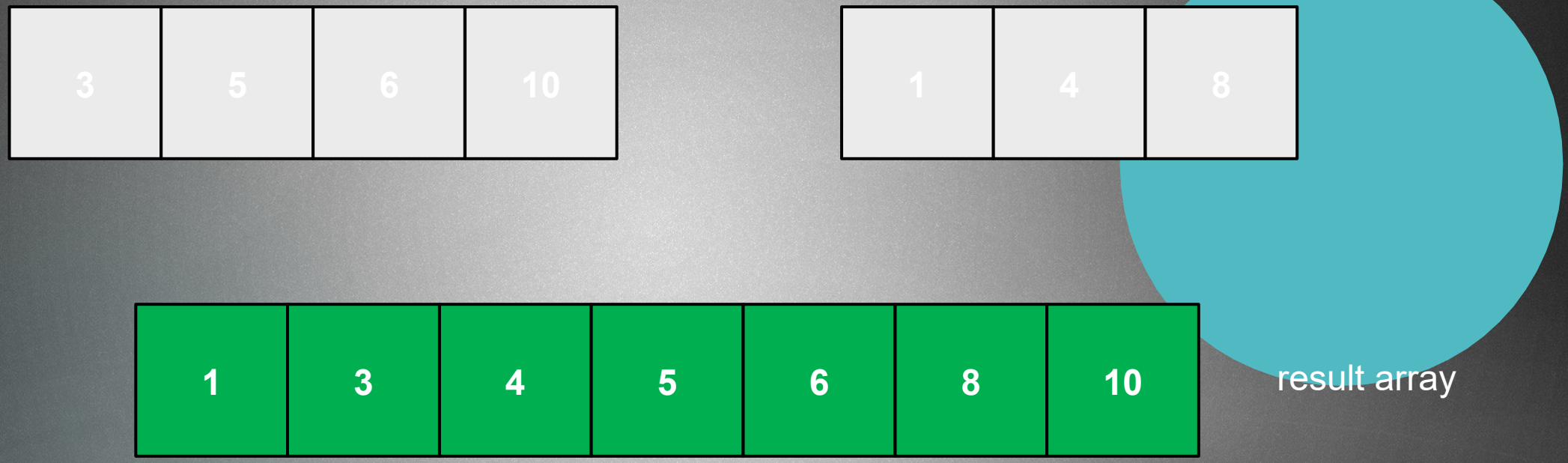





result array

**VERY IMPORTANT**: we have to iterate through the left and right array if there are some more items left → in this case the **10** in the left subarray

# Mergesort „conquer”




**VERY IMPORTANT**: we have to iterate through the left and right array if there are some more items left → in this case the **10** in the left subarray

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
        if  tempArray[i] <= tempArray[j]
            nums[k] = tempArray[i]
            i++
        else
            nums[k] = tempArray[j];
            j++

        k++;

     while i <= middle
            nums[k] = tempArray[i];
            k++;
             i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Base case for recursive method calls, in this situation the sort is over

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

    i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
        if  tempArray[i] <= tempArray[j]
            nums[k] = tempArray[i]
            i++
        else
            nums[k] = tempArray[j];
            j++

        k++;

    while i <= middle
            nums[k] = tempArray[i];
            k++;
            i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

We look for the middle index to partition the array into two equal subarrays

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

      while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Call the mergesort method recursively on the left subarray

IMPORTANT: because of middle, there will be always more items in the left subarray !!!

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

    i = low;
    j = middle + 1;
    k = low;

while i <= middle &&  j <= high
    if  tempArray[i] <= tempArray[j]
        nums[k] = tempArray[i]
        i++
    else
        nums[k] = tempArray[j];
        j++

    k++;

 while i <= middle
        nums[k] = tempArray[i];
        k++;
        i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Call mergesort recursively
on the right subarray

IMPORTANT: because middle+1 there are
at most as many items in the right subarray

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

     while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

The conquer part of the algorithm, we keep merging together the subarrays

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

    i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
        if  tempArray[i] <= tempArray[j]
            nums[k] = tempArray[i]
            i++
        else
            nums[k] = tempArray[j];
            j++

        k++;

    while i <= middle
            nums[k] = tempArray[i];
            k++;
            i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Create a temporary array: size is equal to the size of the input

Thats why mergesort has O(N) memory complexity → we have to use an other array

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

    i = low;
    j = middle + 1;
    k = low;

while i <= middle &&  j <= high
    if  tempArray[i] <= tempArray[j]
        nums[k] = tempArray[i]
        i++
    else
        nums[k] = tempArray[j];
        j++

    k++;

 while i <= middle
        nums[k] = tempArray[i];
        k++;
        i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Create the variables to be able to track the indexes

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

     while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

While we have items in the left and right subarrays

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

      while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

The left subarray item is smaller: we put it to its right / sorted position in the array

Note: we keep merging the items from temp array to the original nums array

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

     while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

The right subarray item is smaller: we put it to its right / sorted position in the array

Note: we keep merging the items from temp array to the original nums array

```
merge(low, middle, high) {

    for i=low to high
            tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
            if  tempArray[i] <= tempArray[j]
                nums[k] = tempArray[i]
                i++
            else
                nums[k] = tempArray[j];
                j++

            k++;

     while i <= middle
                nums[k] = tempArray[i];
                k++;
                 i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

     i = low;
        j = middle + 1;
        k = low;

    while i <= middle &&  j <= high
        if  tempArray[i] <= tempArray[j]
            nums[k] = tempArray[i]
            i++
        else
            nums[k] = tempArray[j];
            j++

        k++;

     while i <= middle
            nums[k] = tempArray[i];
            k++;
             i++;
end
```

## Pseudocode

```
mergeSort(low,high)

    if (low >= high) return

    middle = (low + high) / 2

    mergeSort(low, middle)
    mergeSort(middle + 1, high)
    merge(low, middle, high)
end
```

Sometimes we have items left in the left Subarray: so copy it to the final nums array

~ it is in sorted order so just copy them

IMPORTANT: because we partition our array in a way that the left subarray contains more items → we just have to consider the left subarray

~ so we do not have to copy items from right subarray too !!!

```
merge(low, middle, high) {

    for i=low to high
        tempArray[i] = nums[i];

    i = low;
    j = middle + 1;
    k = low;

while i <= middle &&  j <= high
    if  tempArray[i] <= tempArray[j]
        nums[k] = tempArray[i]
        i++
    else
        nums[k] = tempArray[j];
        j++

    k++;

 while i <= middle
        nums[k] = tempArray[i];
        k++;
        i++;
end
```

```
mergeSort(low,high)

        if (low >= high) return

        middle = (low + high) / 2

        mergeSort(low, middle)
        mergeSort(middle + 1, high)
        merge(low, middle, high)
end
```