# CSCI 132:
# Basic Data Structures and Algorithms

Sorting (Merge Sort)

Reese Pearsall & Iliana Castillon

Fall 2024

# Announcements

Lab 11 due **tomorrow @ 11:59 PM**

Program 4 Due **Friday @ 11:59 PM**

Reese’s office hours are 2PM – 3PM tomorrow

**Merge Sort** is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the subarrays back together to form the final sorted array

Merge sort is a **Divide and Conquer** algorithm, which involves dividing the problem into smaller sub-problems (divide), recursively solving the smaller problems (conquer), and combining the sub problems to get the final solution for the original problem

Merge sort and the next sorting algorithm we will discuss next week are rather complex. I don’t expect you to memorize the code, and if you don’t fully understand the code, *that is fine!*

You should, however, be able to describe how merge sort works from a high level, and be able to draw out the steps if given an example array

You should also know the time complexity of the sorting algorithms that we talk about

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

Goal: Divide array into two subarrays using recursion:

Base Case:

If an array is of size 1, **return**

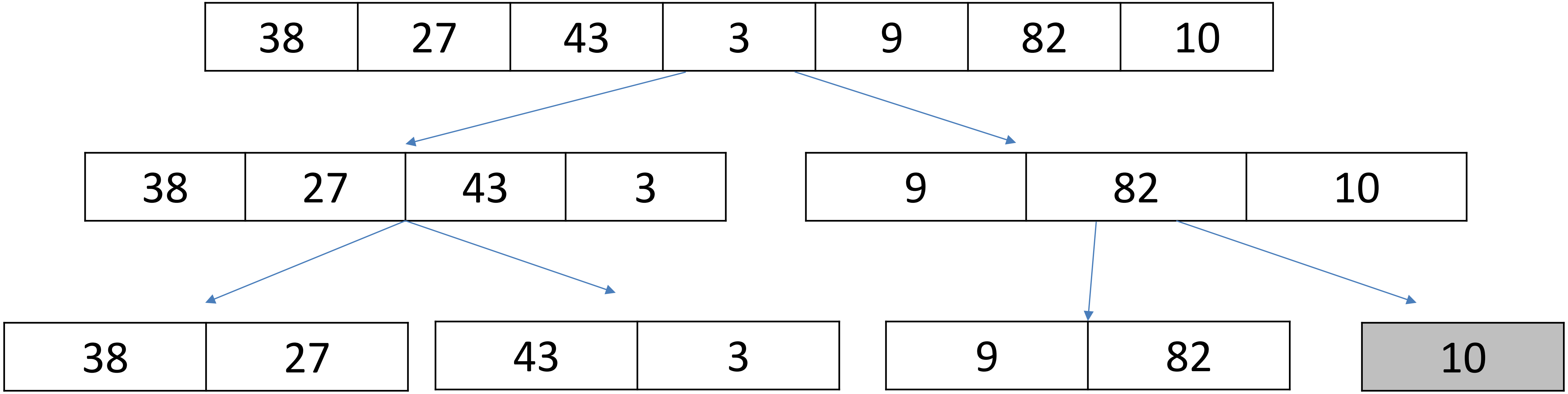
Recursive Case:

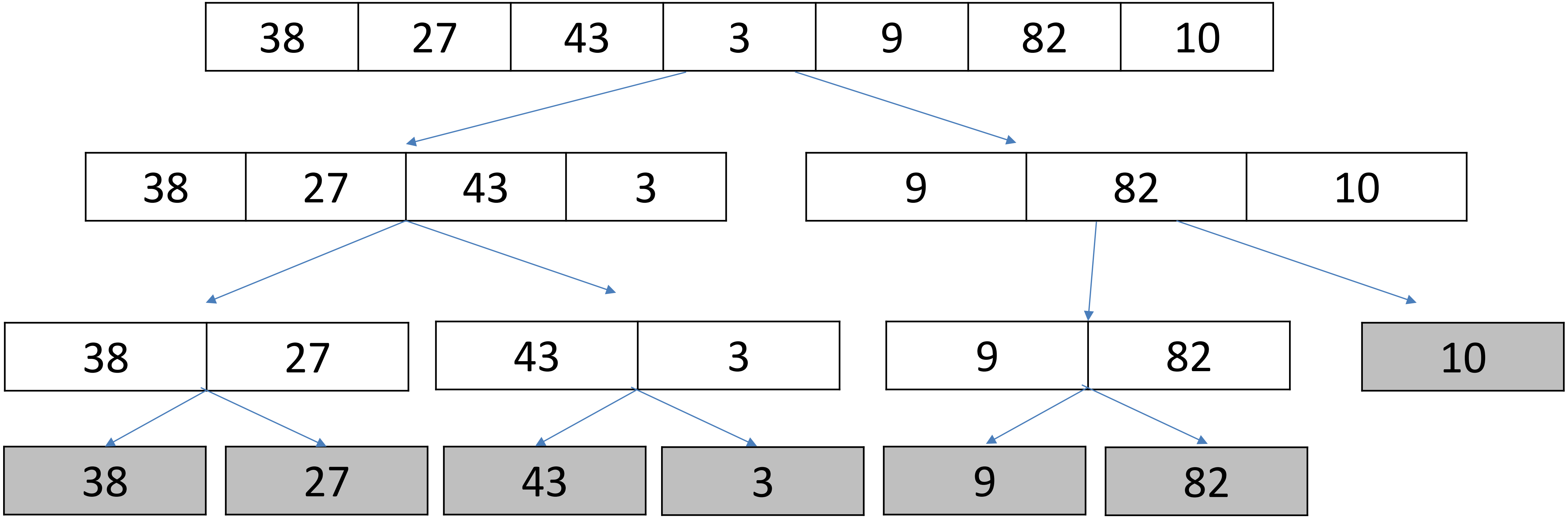
Generate two subarrays `leftArray`, and `rightArray`
`mergeSort(leftArray), mergeSort(rightArray)`

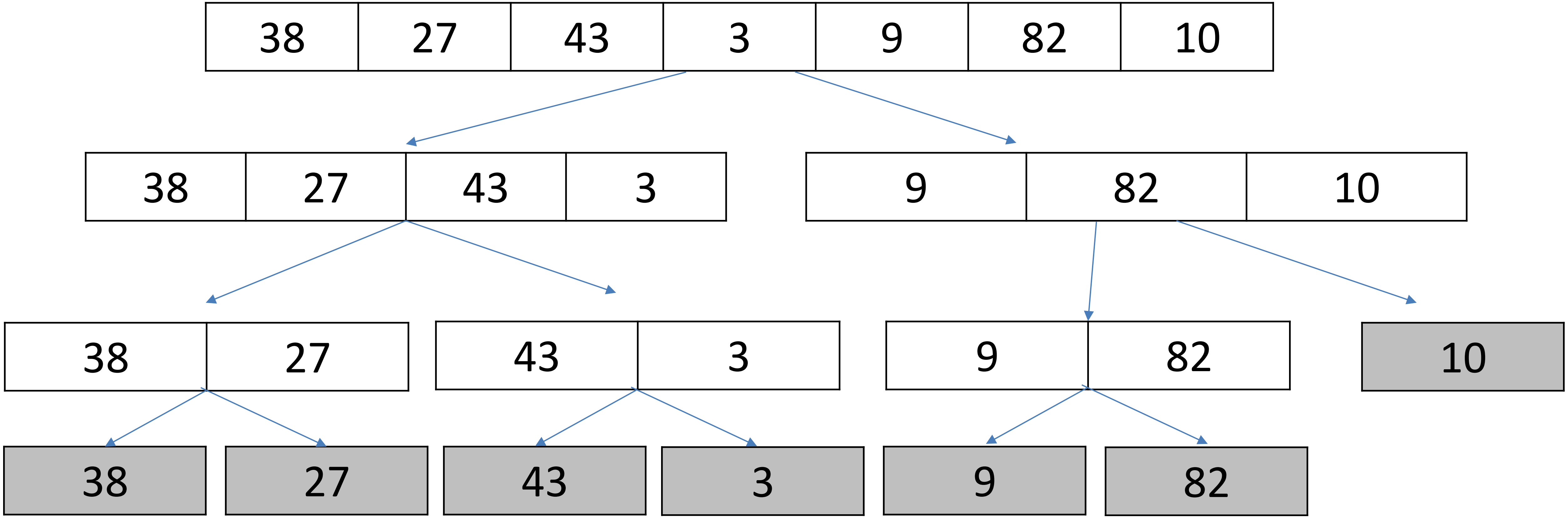
| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | |
|---|---|---|
| 9 | 82 | 10 |

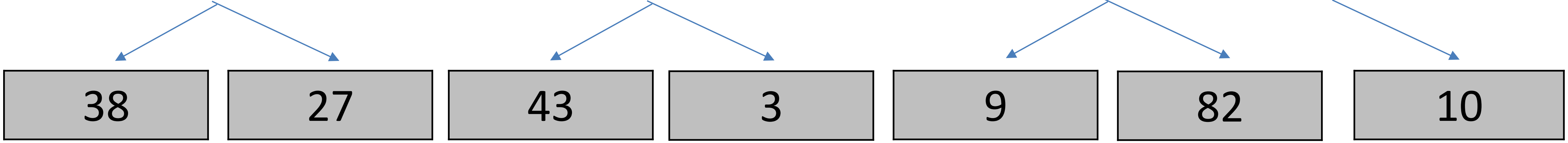

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10


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



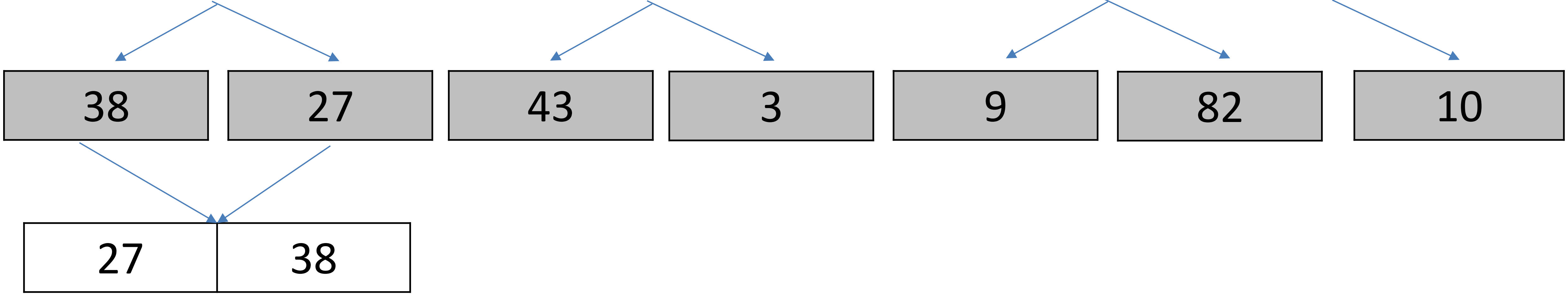

We’ve hit all our base cases (arrays of size 1), now we will begin to **merge** the subarrays

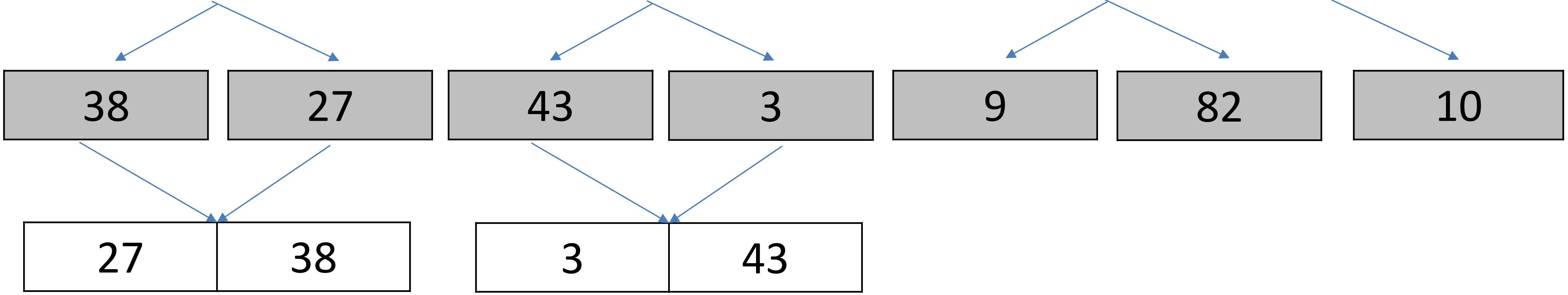
When we merge, our merged array will be sorted

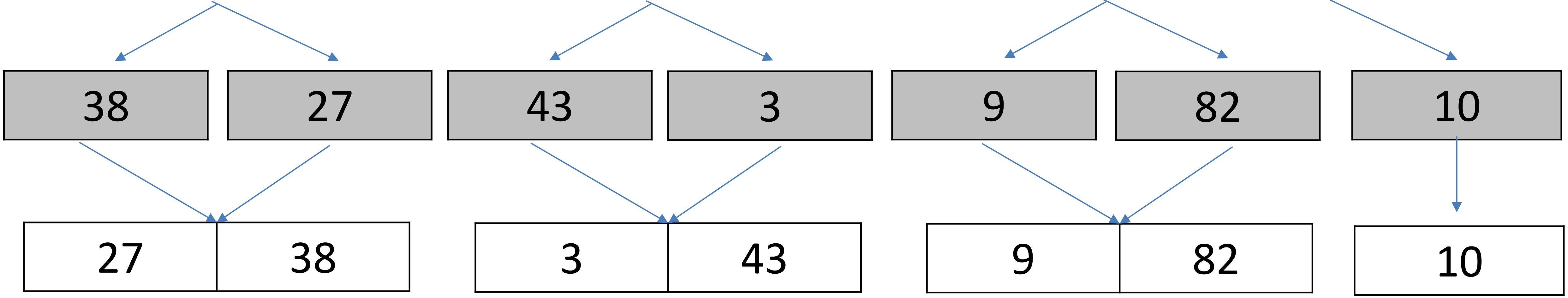


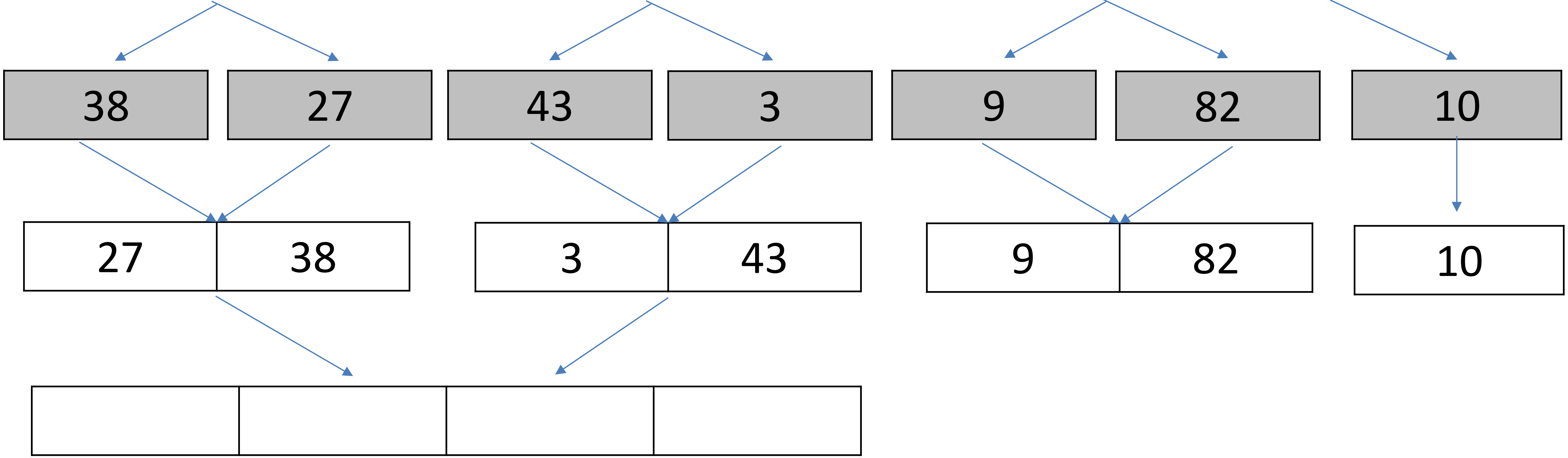

We've hit all our base cases (arrays of size 1), now we will begin to **merge** the subarrays

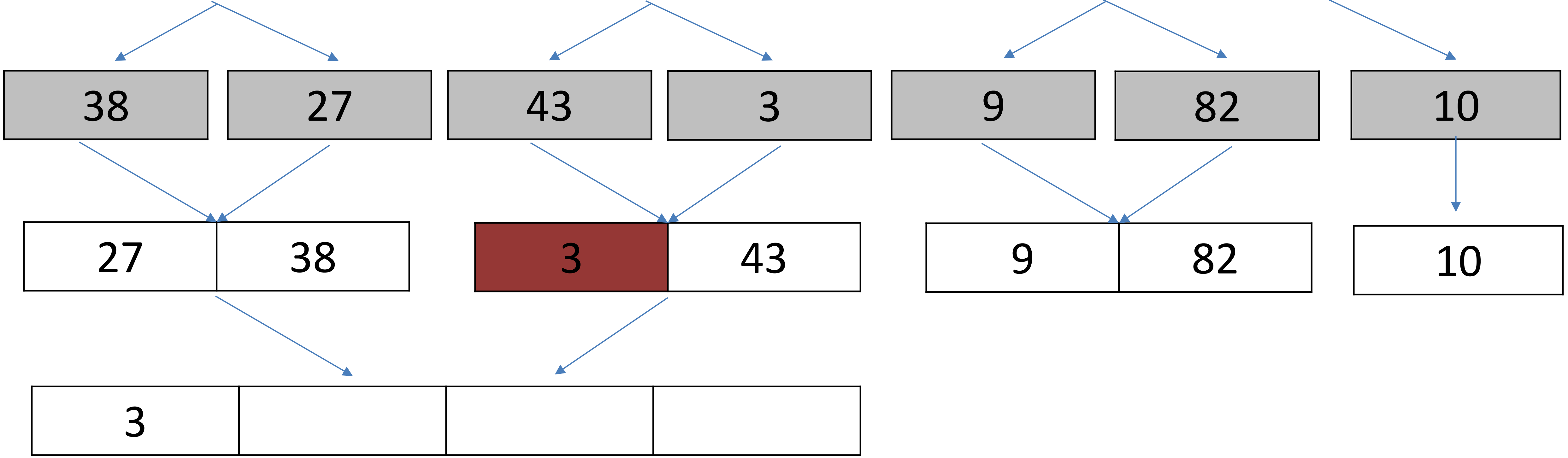
When we merge, our merged array will be sorted

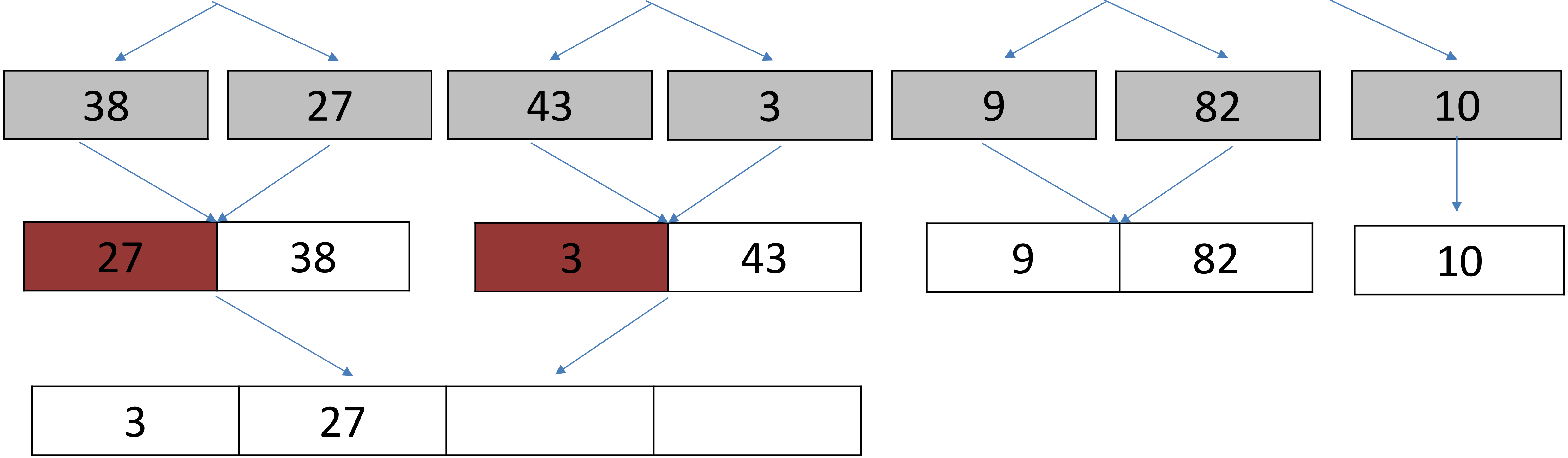

38
27
43
3
9
82
10
27
38

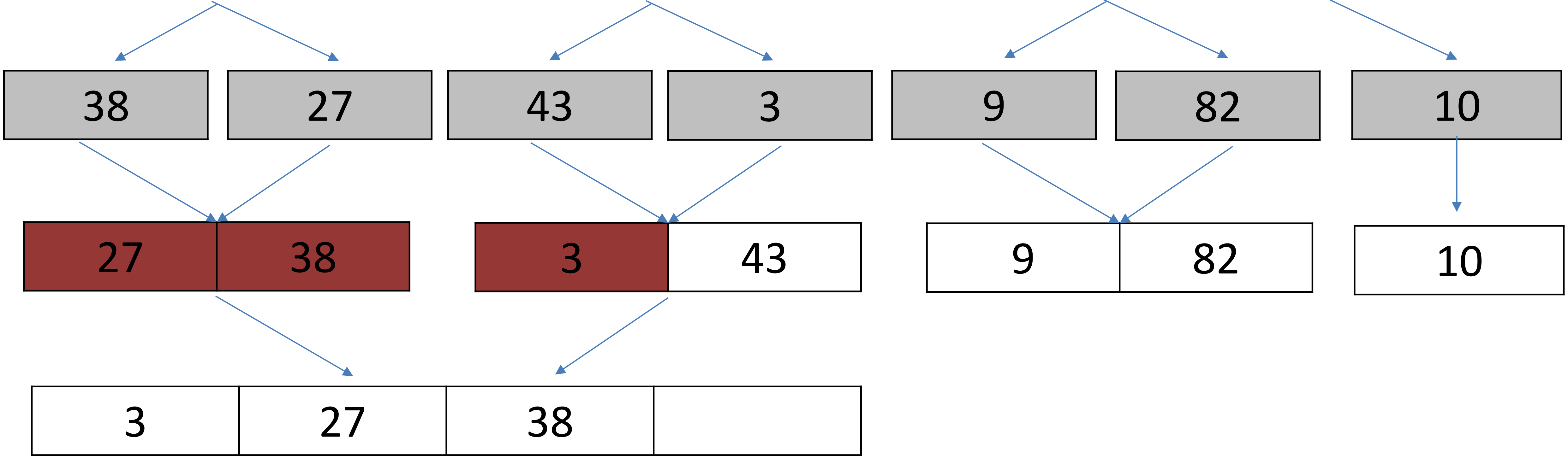

38
27
43
3
9
82
10
27
38
3
43

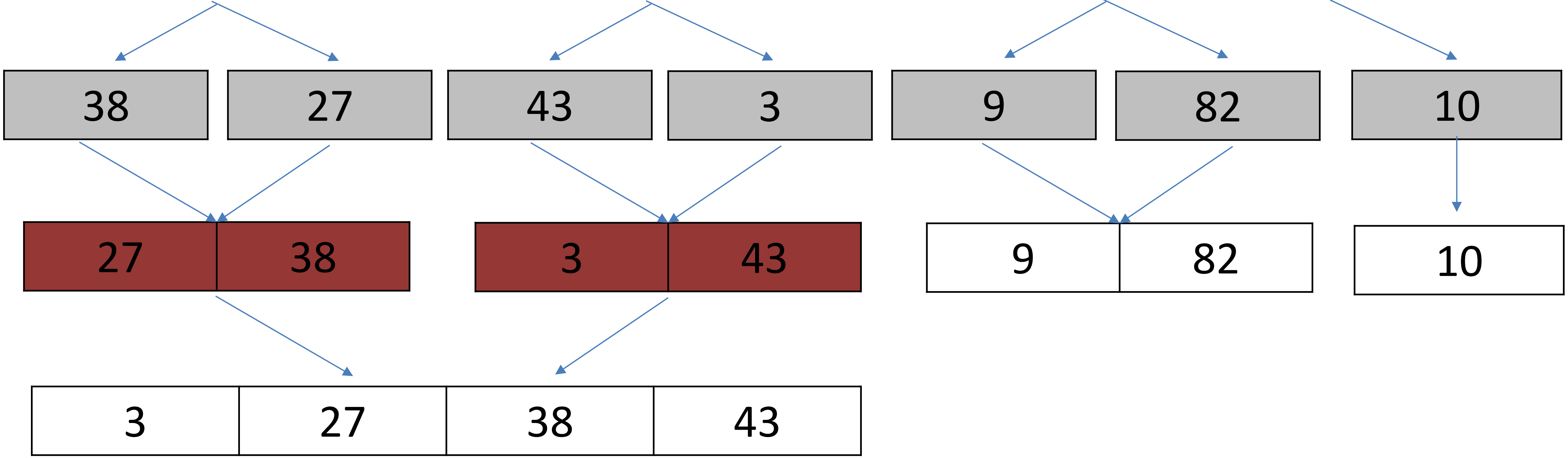

38
27
43
3
9
82
10
27
38
3
43
9
82
10

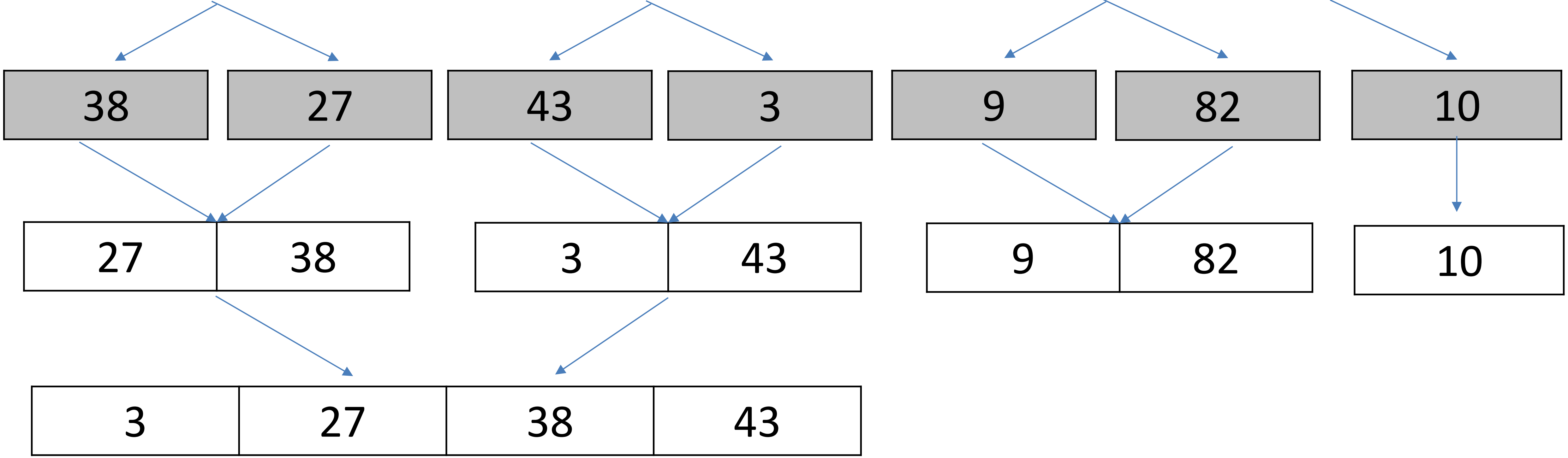

38
27
43
3
9
82
10
27
38
3
43
9
82
10

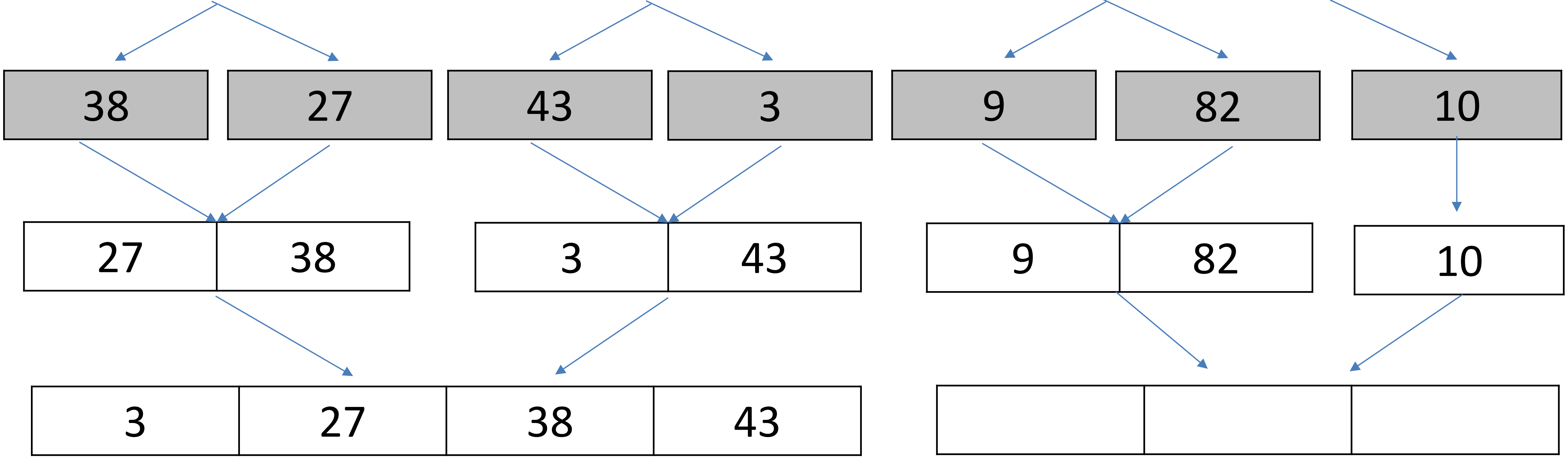

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3

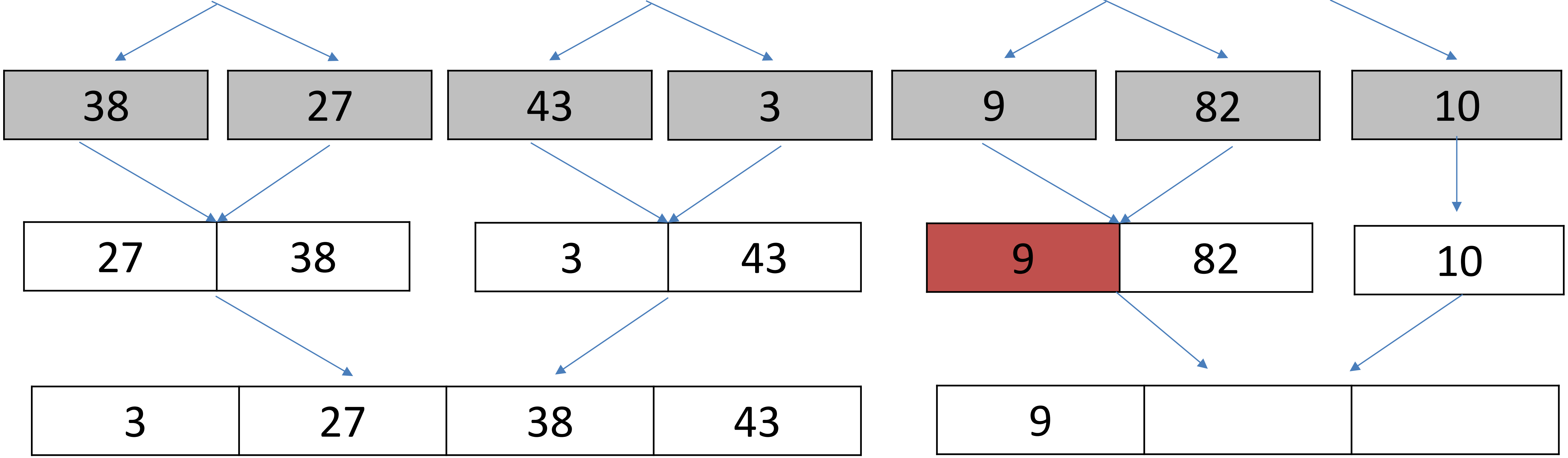

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3
27

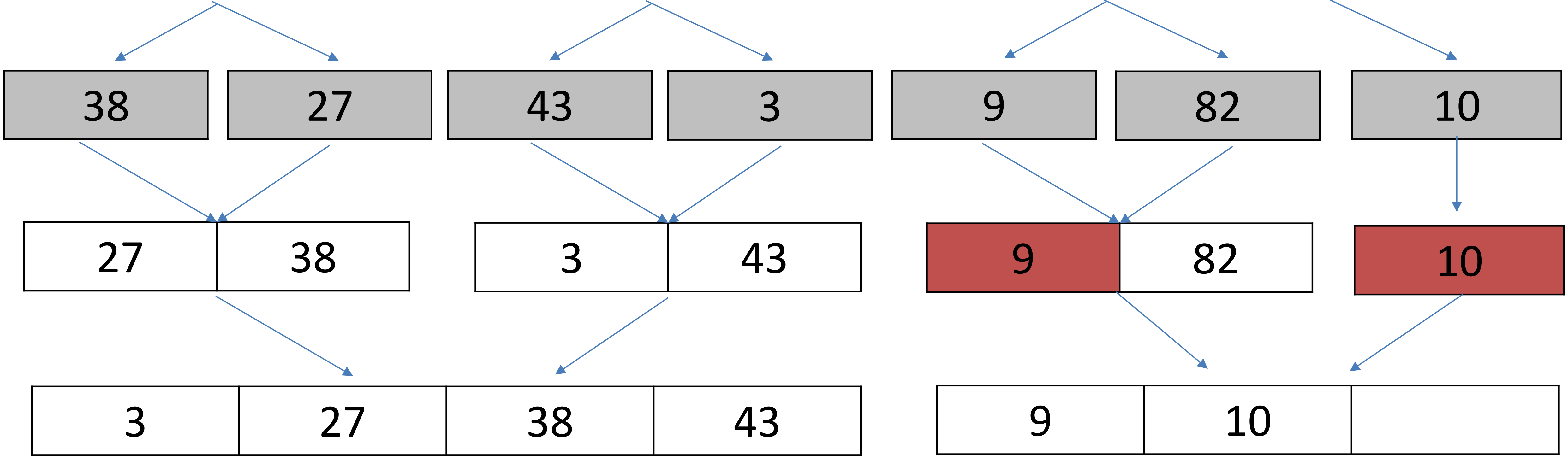

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3
27
38

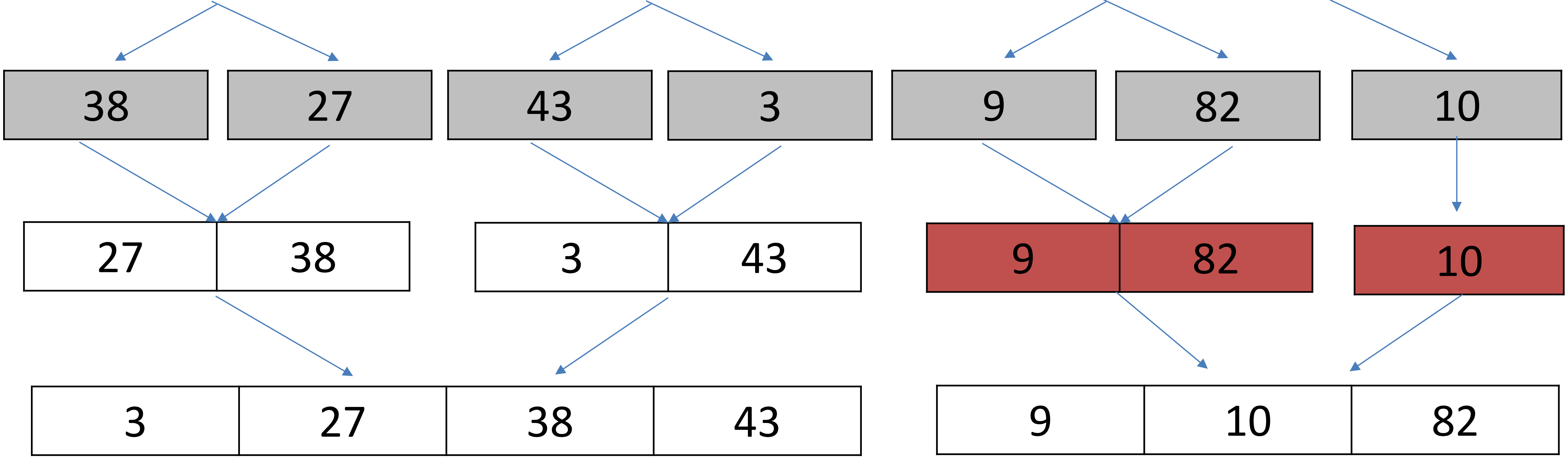

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3
27
38
43

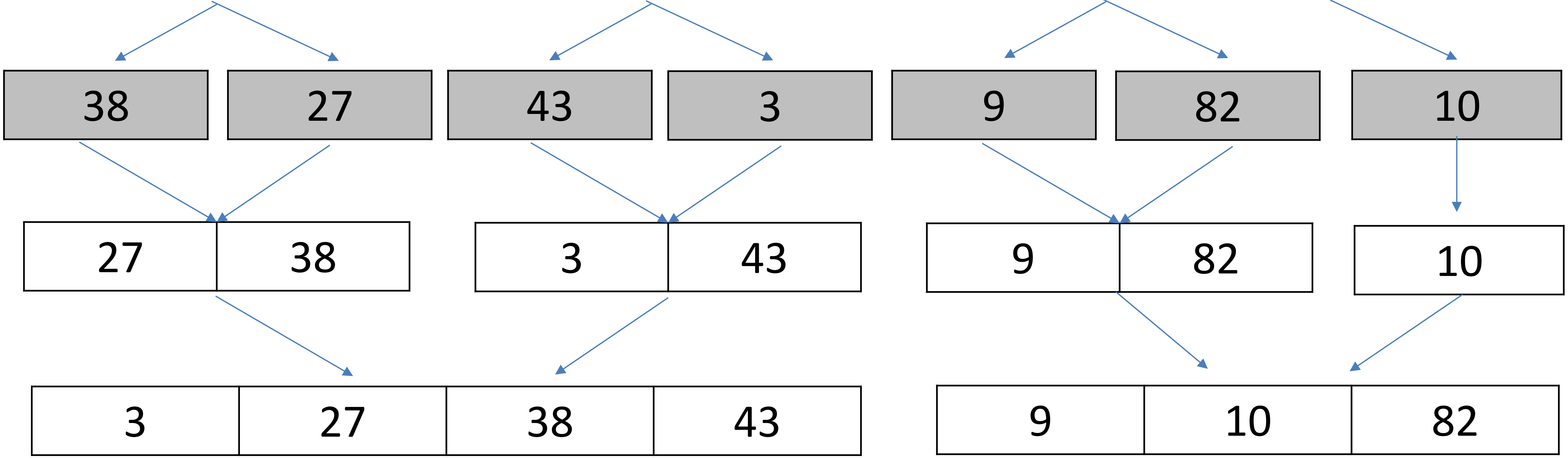

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3
27
38
43

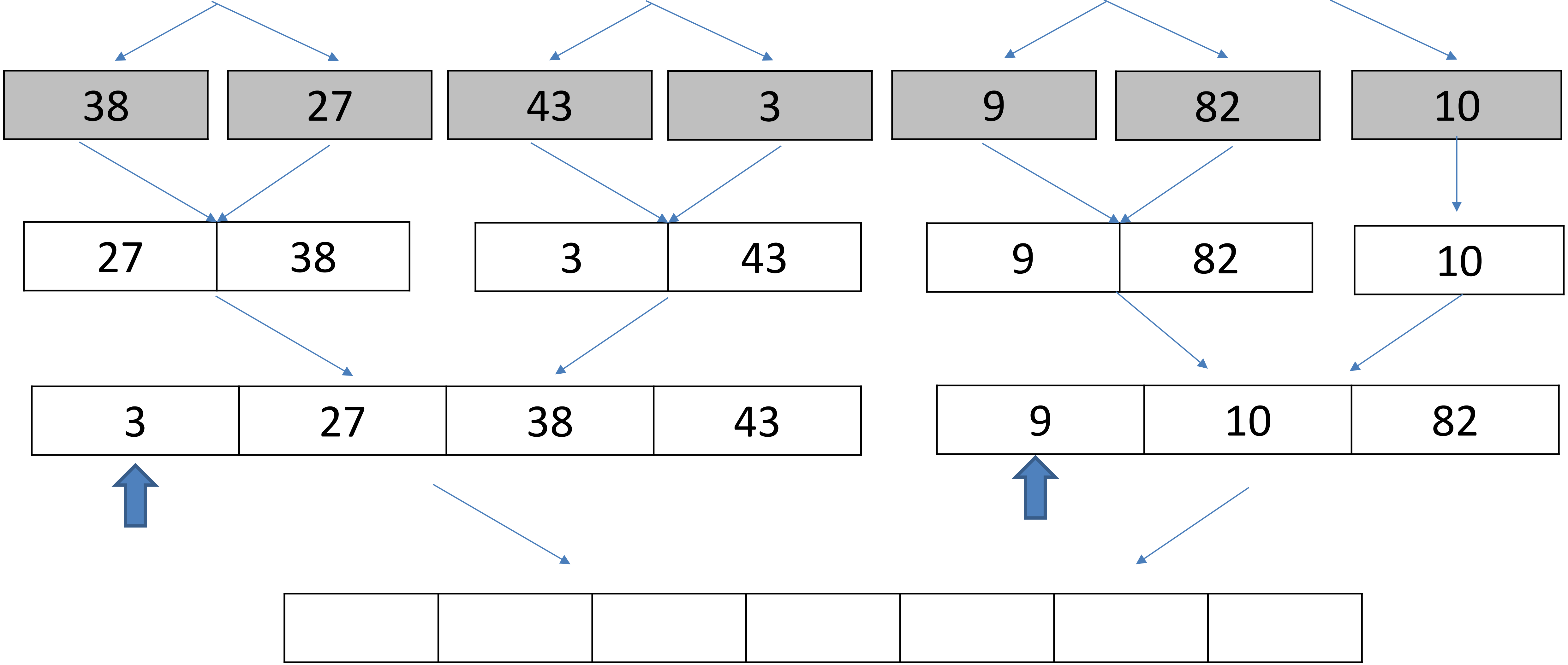

38
27
43
3
9
82
10
27
38
3
43
9
82
10
3
27
38
43


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

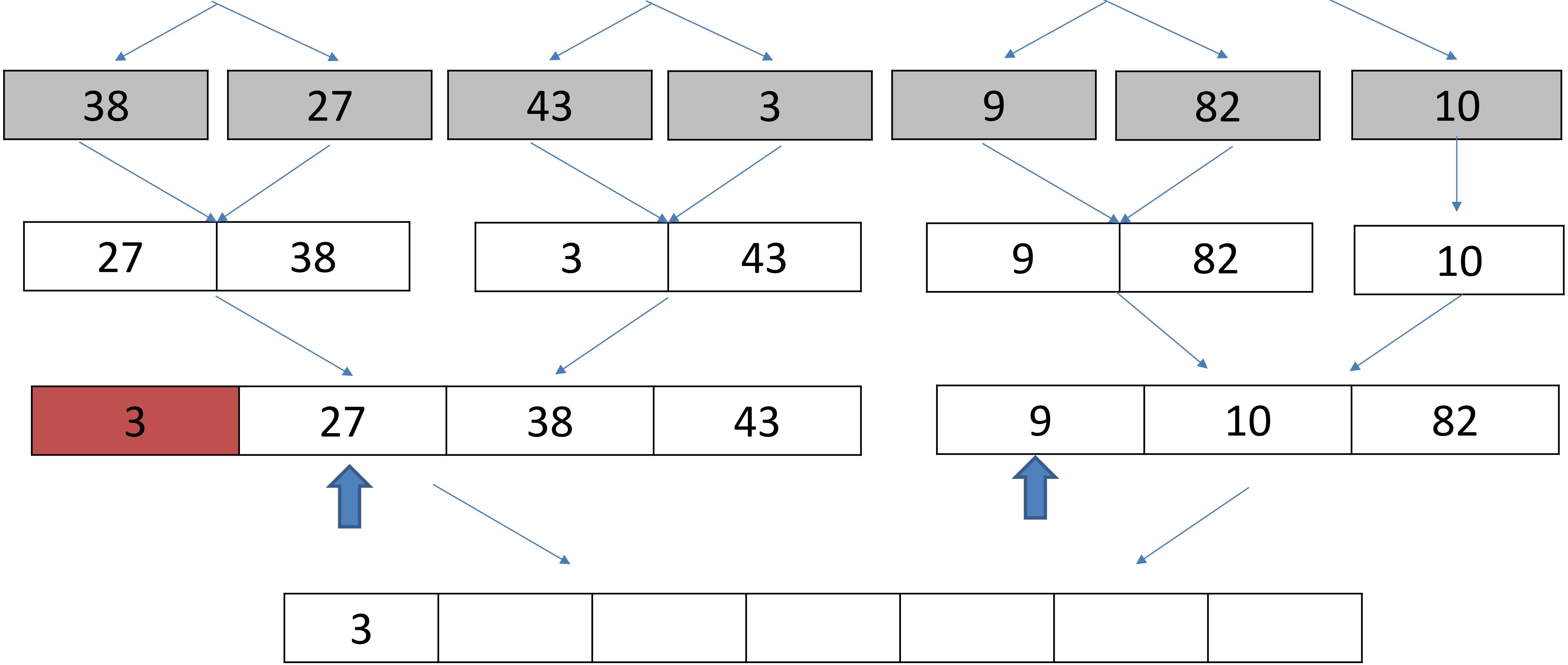

38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10


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

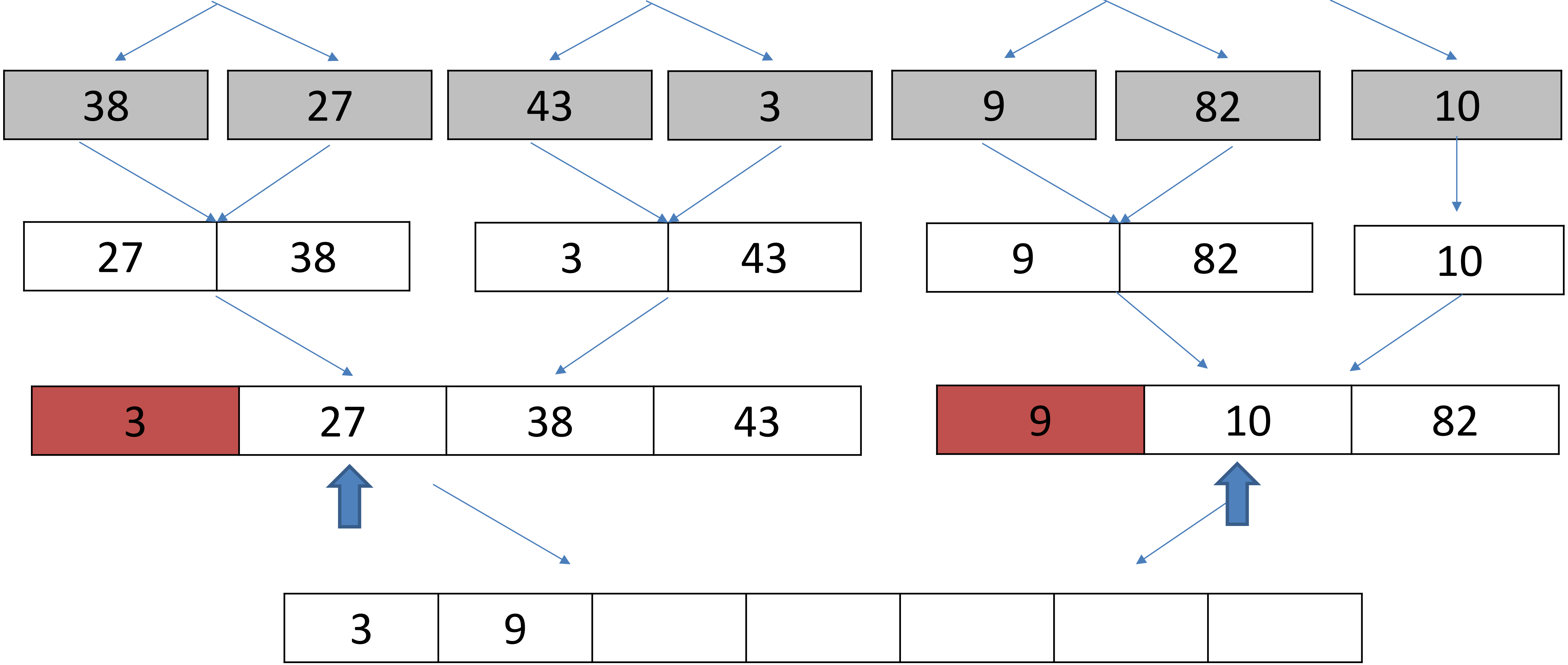

38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82



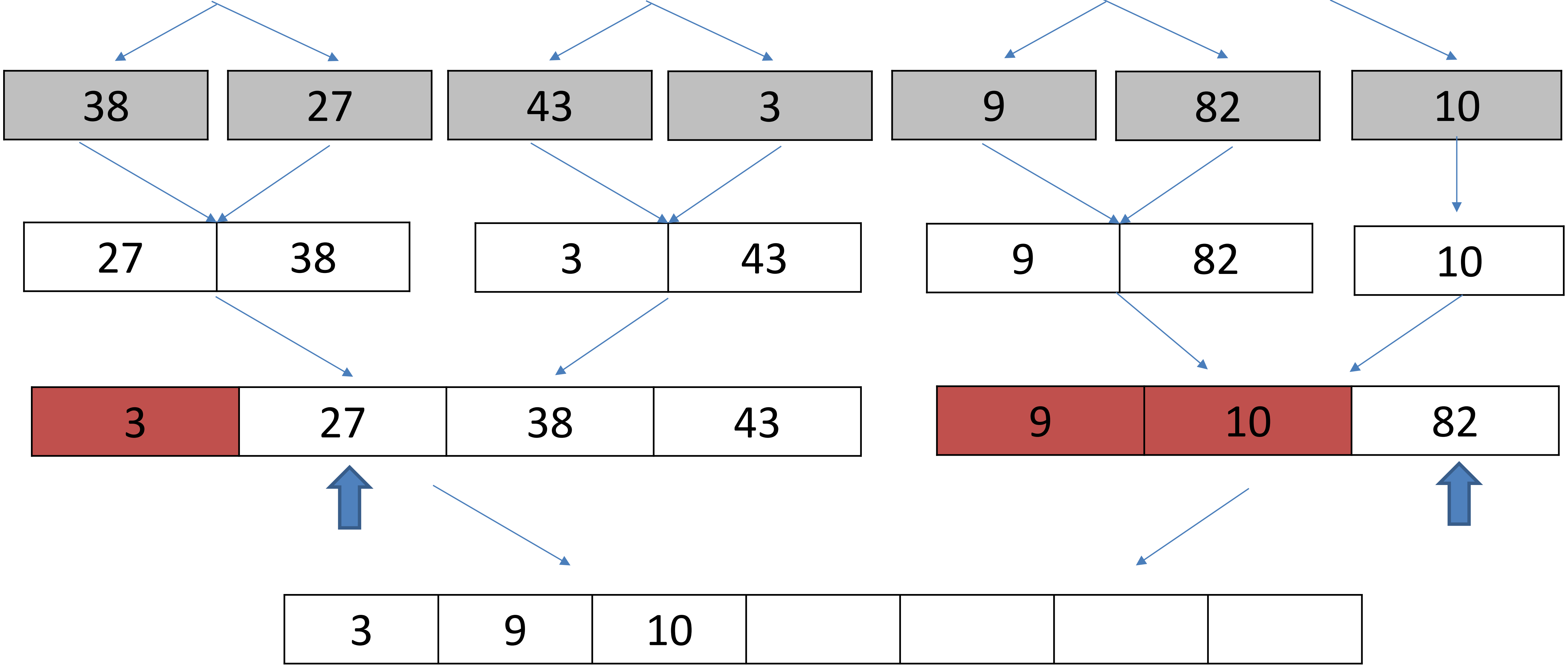

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index



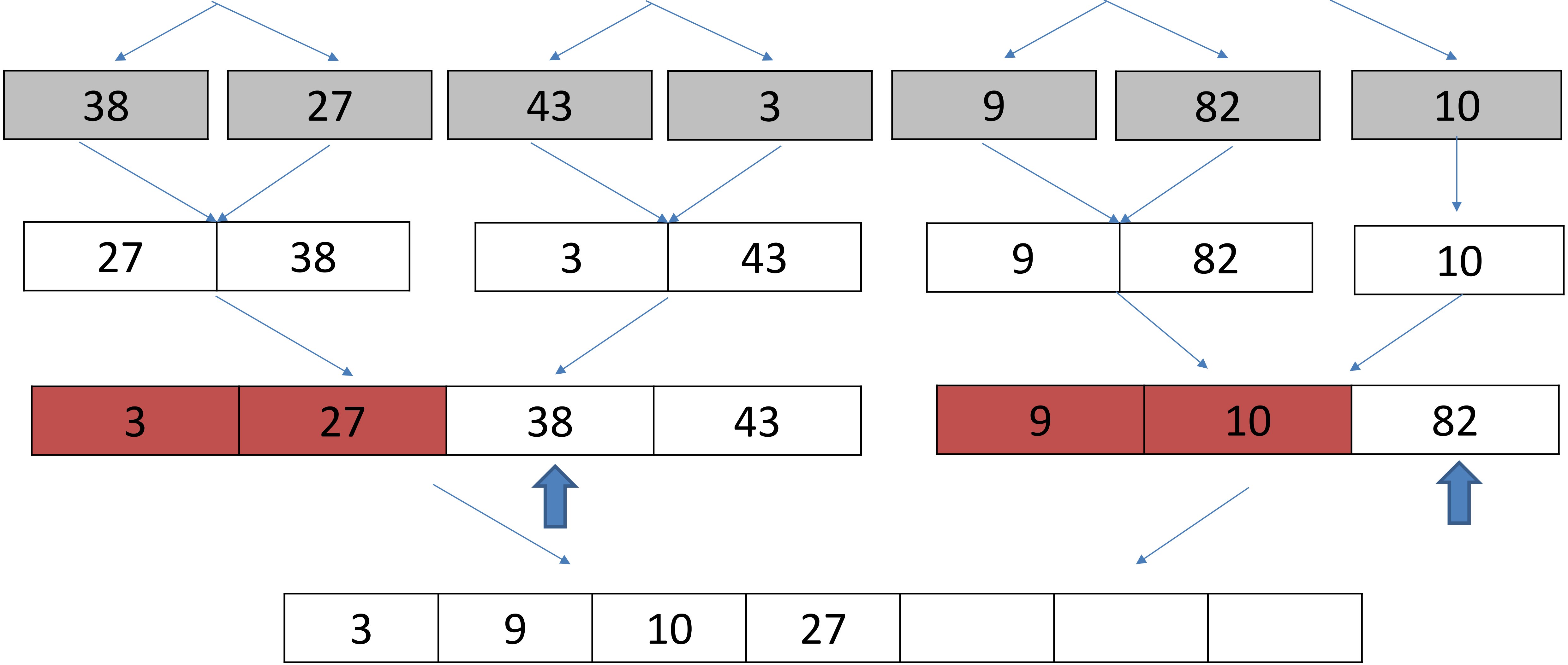

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index



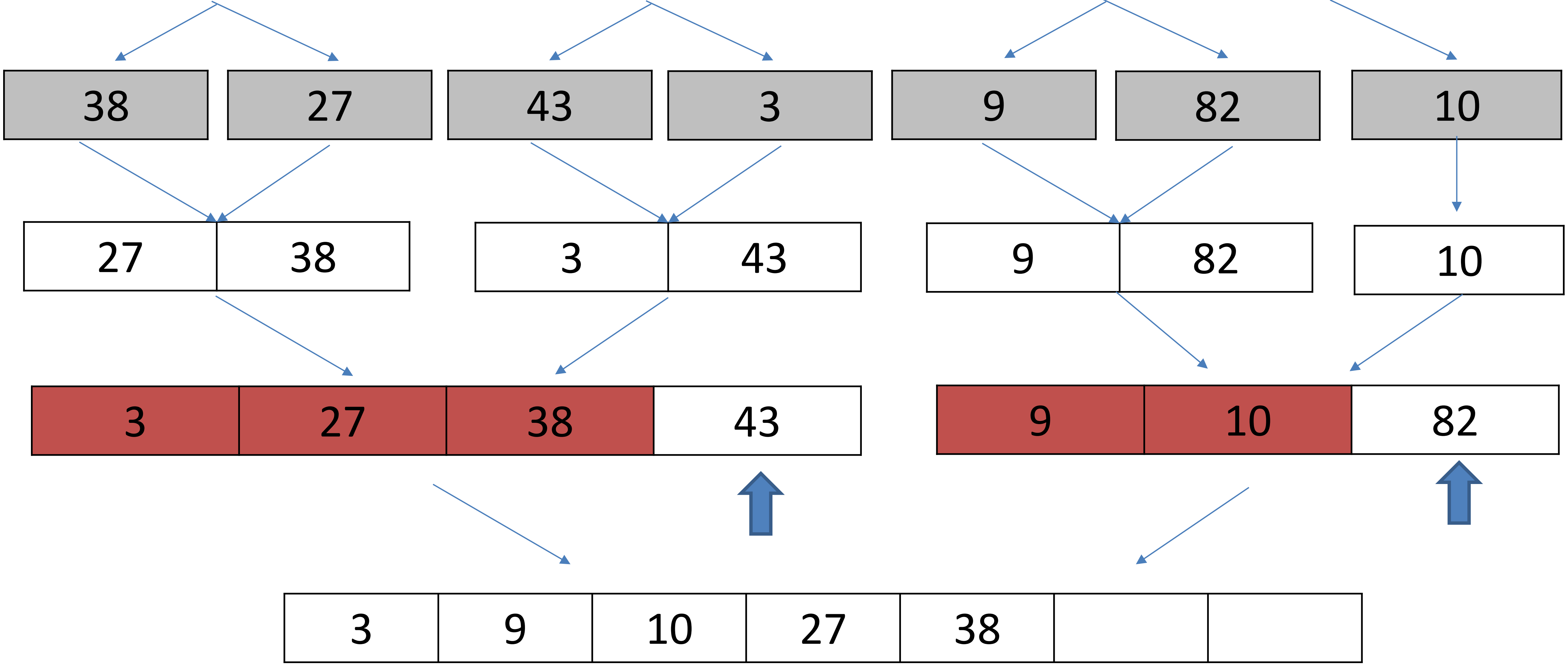

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index



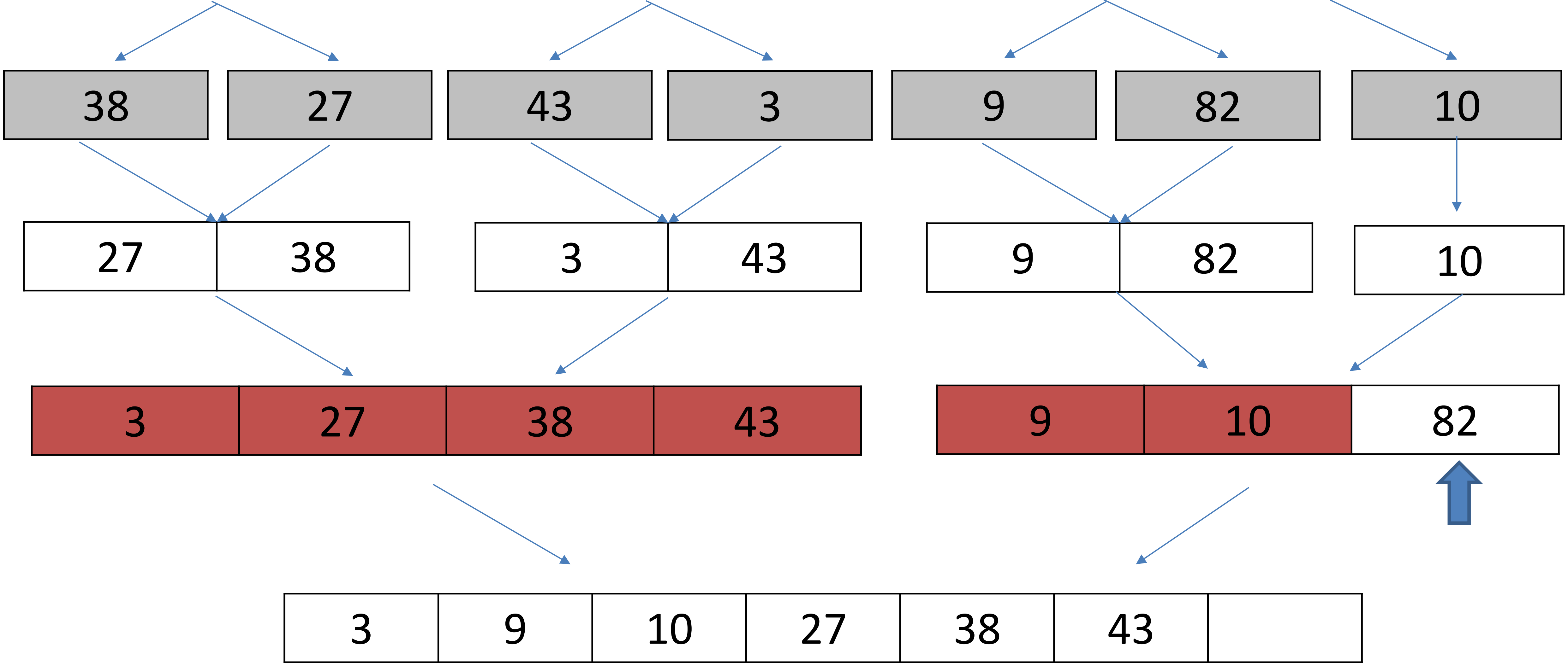

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index



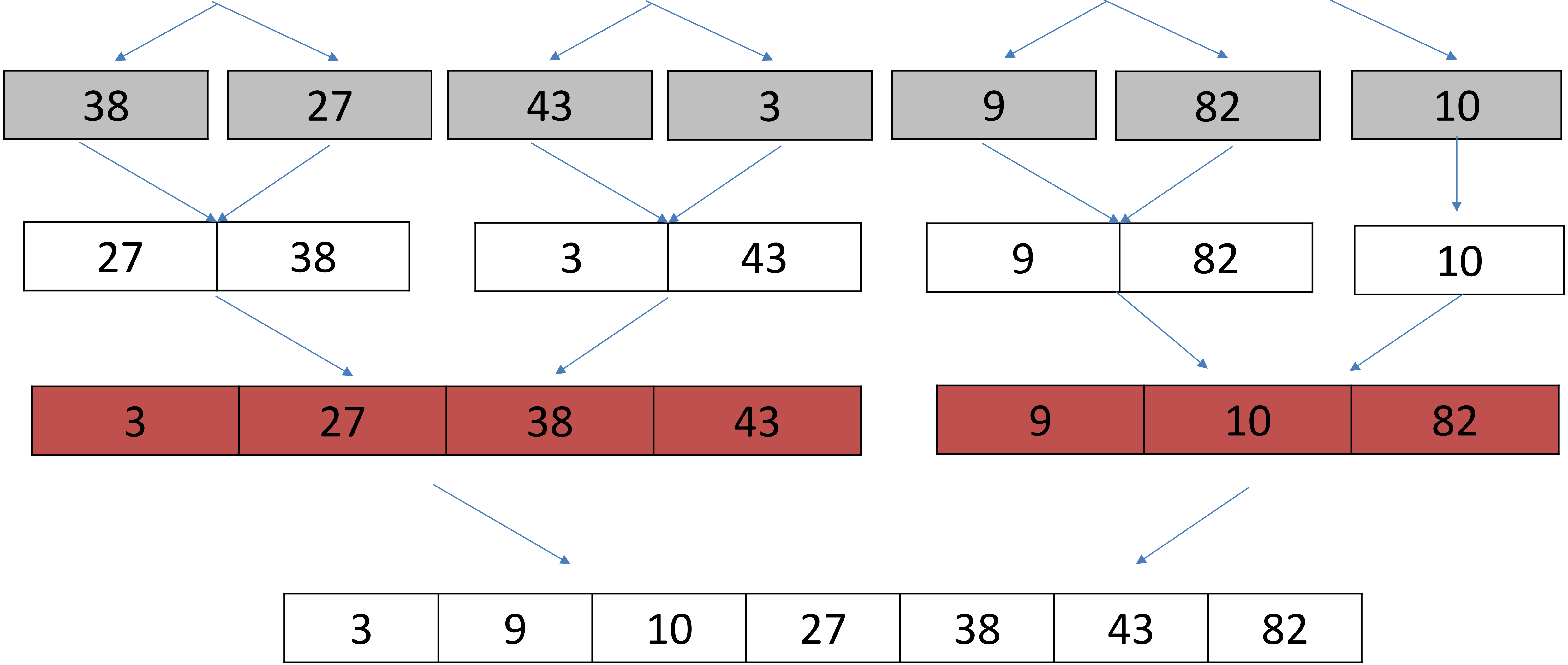

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index



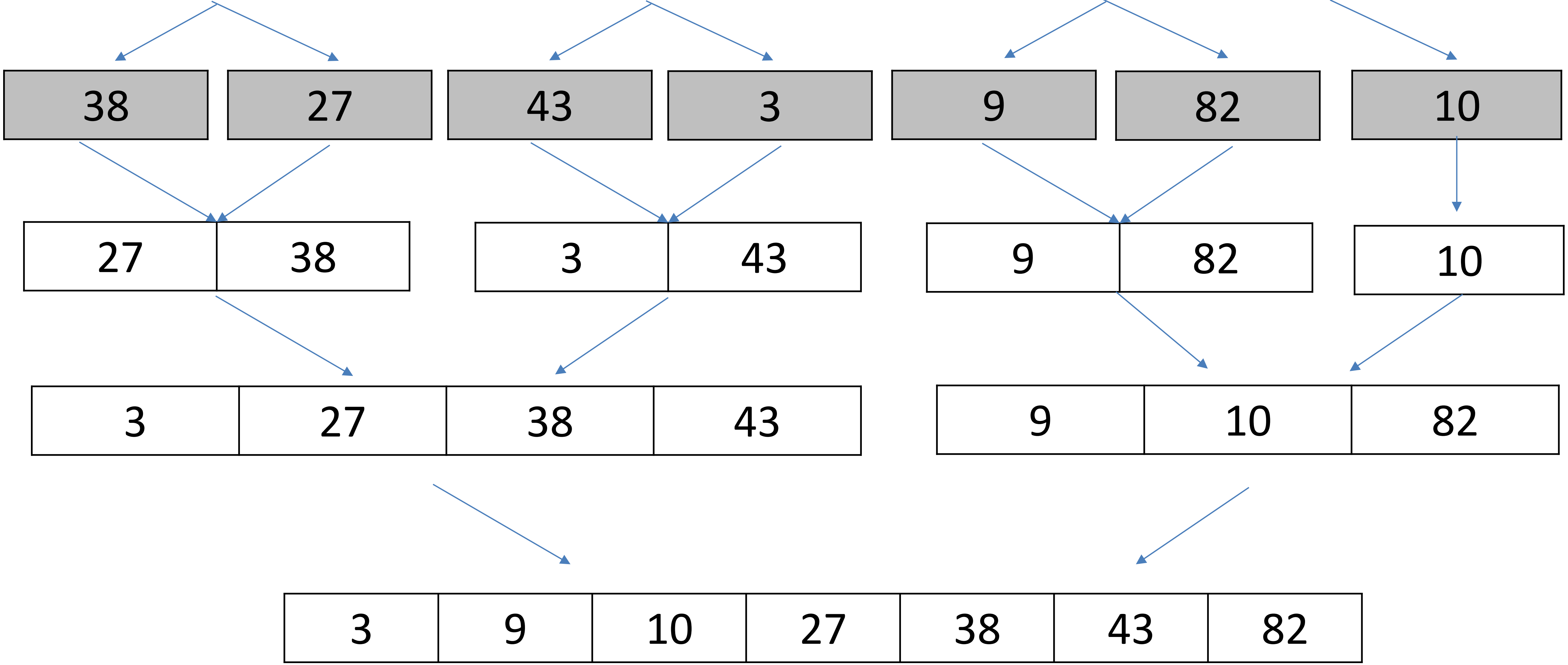

Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index




Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index

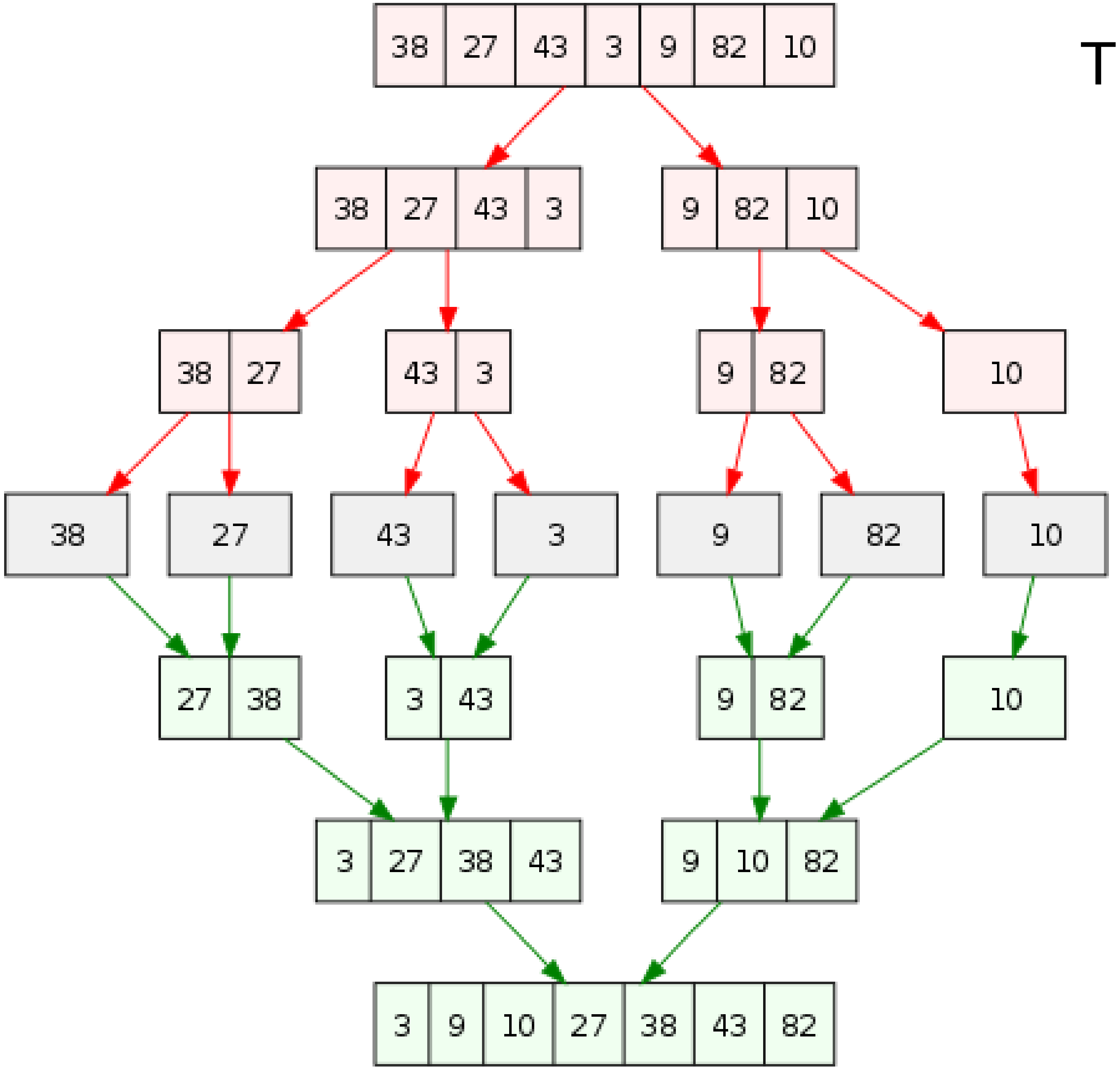



Because we know the subarray that we are merging are already sorted, the smallest element will always be at the first index




Our original array is now sorted!!

# The entire merge sort process




Divide

*(base case)*

Merge

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

In our Java code, this will actually be the order of how things are done…

In practice, we will always prioritize solving the “left” tree first

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| |
|---|
| 38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 27 | 38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| | |
|---|---|
| 27 | 38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| | |
|---|---|
| 27 | 38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

38

27

43

3

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | |
|---|---|---|
| 9 | 82 | 10 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | |
|---|---|---|
| 9 | 82 | 10 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| | |
|---|---|
| 9 | 82 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | |
|---|---|---|
| 9 | 82 | 10 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| | |
|---|---|
| 9 | 82 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

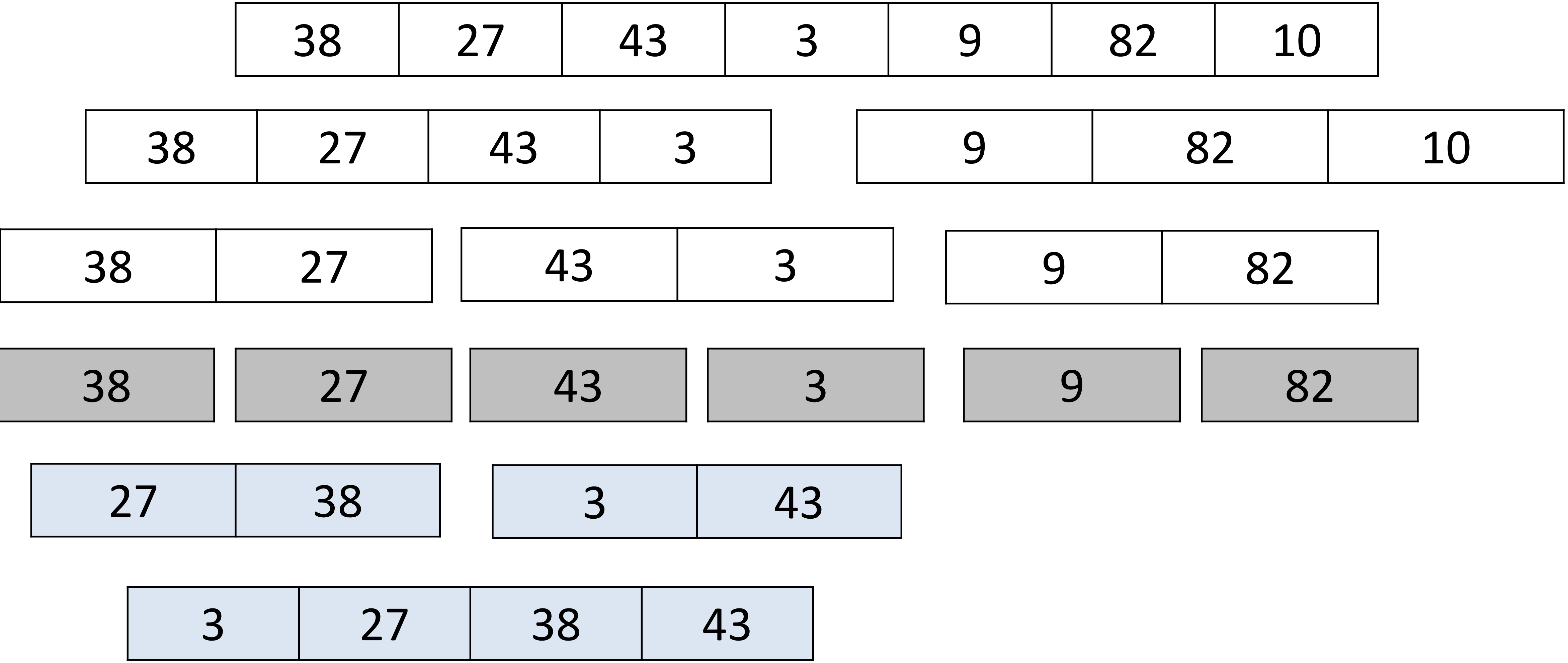
| |
|---|
| 3 |

| |
|---|
| 9 |

| | |
|---|---|
| 27 | 38 |

| | |
|---|---|
| 3 | 43 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |



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

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

| | | | |
|---|---|---|---|
| 38 | 27 | 43 | 3 |

| | | |
|---|---|---|
| 9 | 82 | 10 |

| | |
|---|---|
| 38 | 27 |

| | |
|---|---|
| 43 | 3 |

| | |
|---|---|
| 9 | 82 |

| |
|---|
| 38 |

| |
|---|
| 27 |

| |
|---|
| 43 |

| |
|---|
| 3 |

| |
|---|
| 9 |

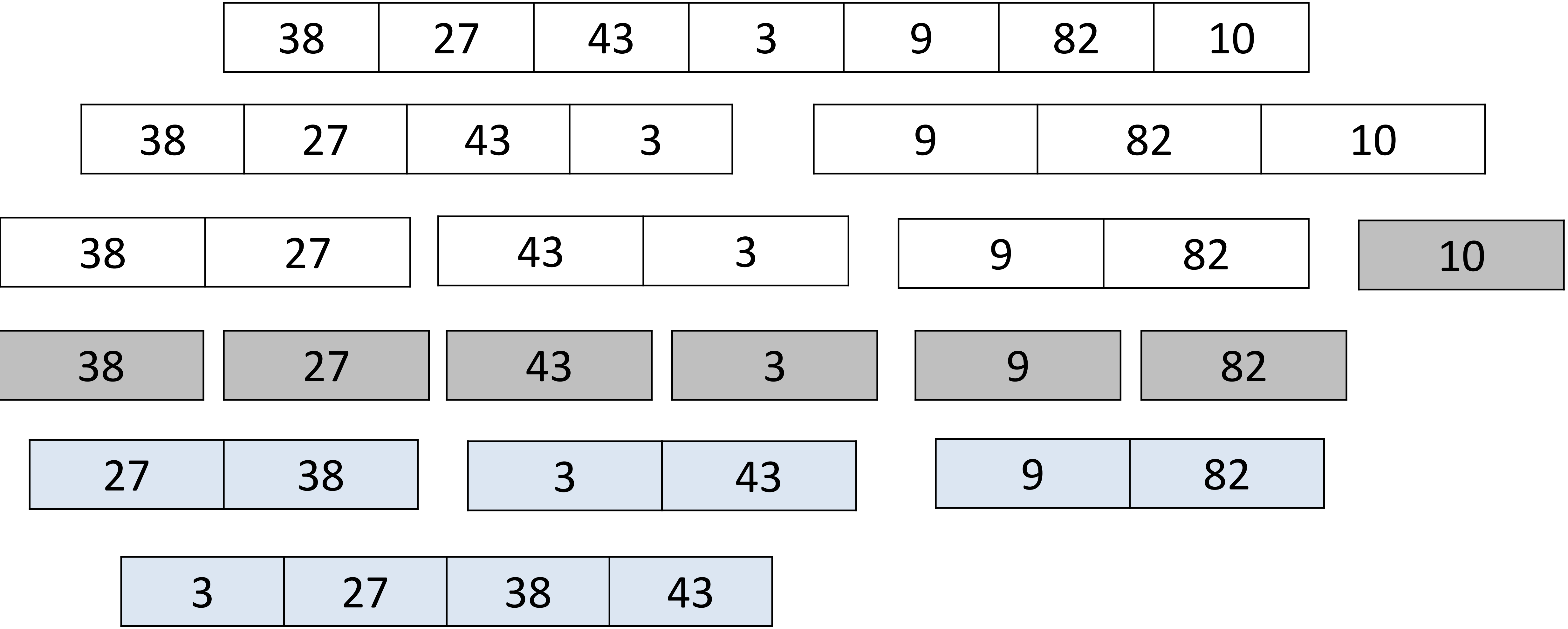
| |
|---|
| 82 |

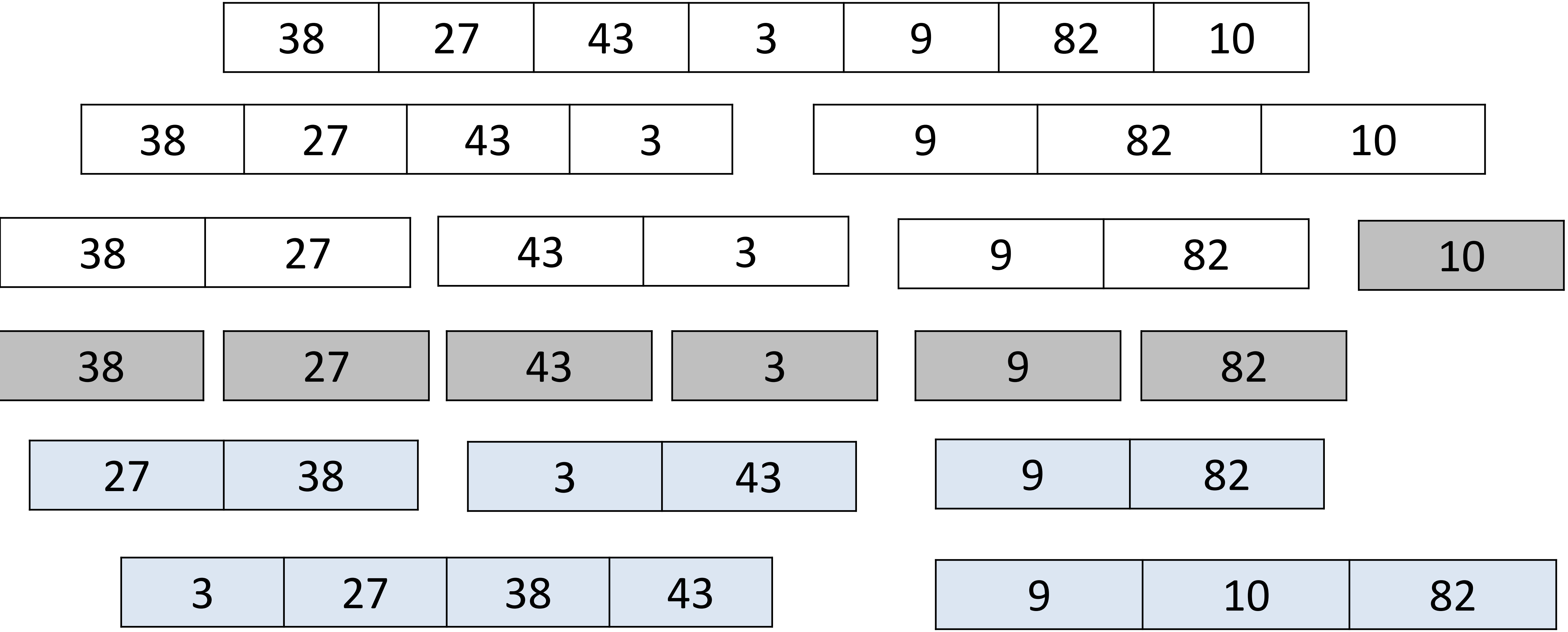
| | |
|---|---|
| 27 | 38 |

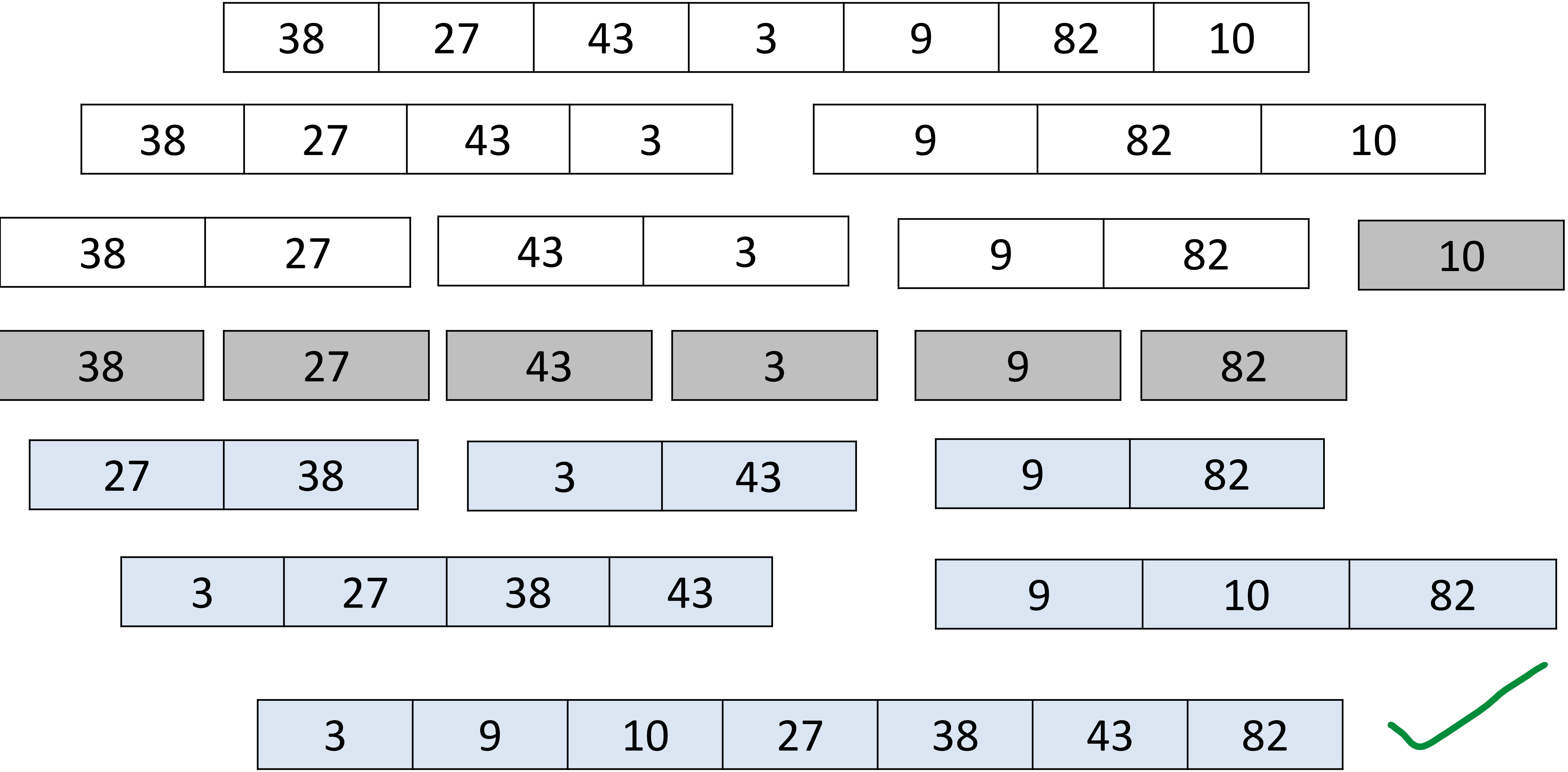
| | |
|---|---|
| 3 | 43 |

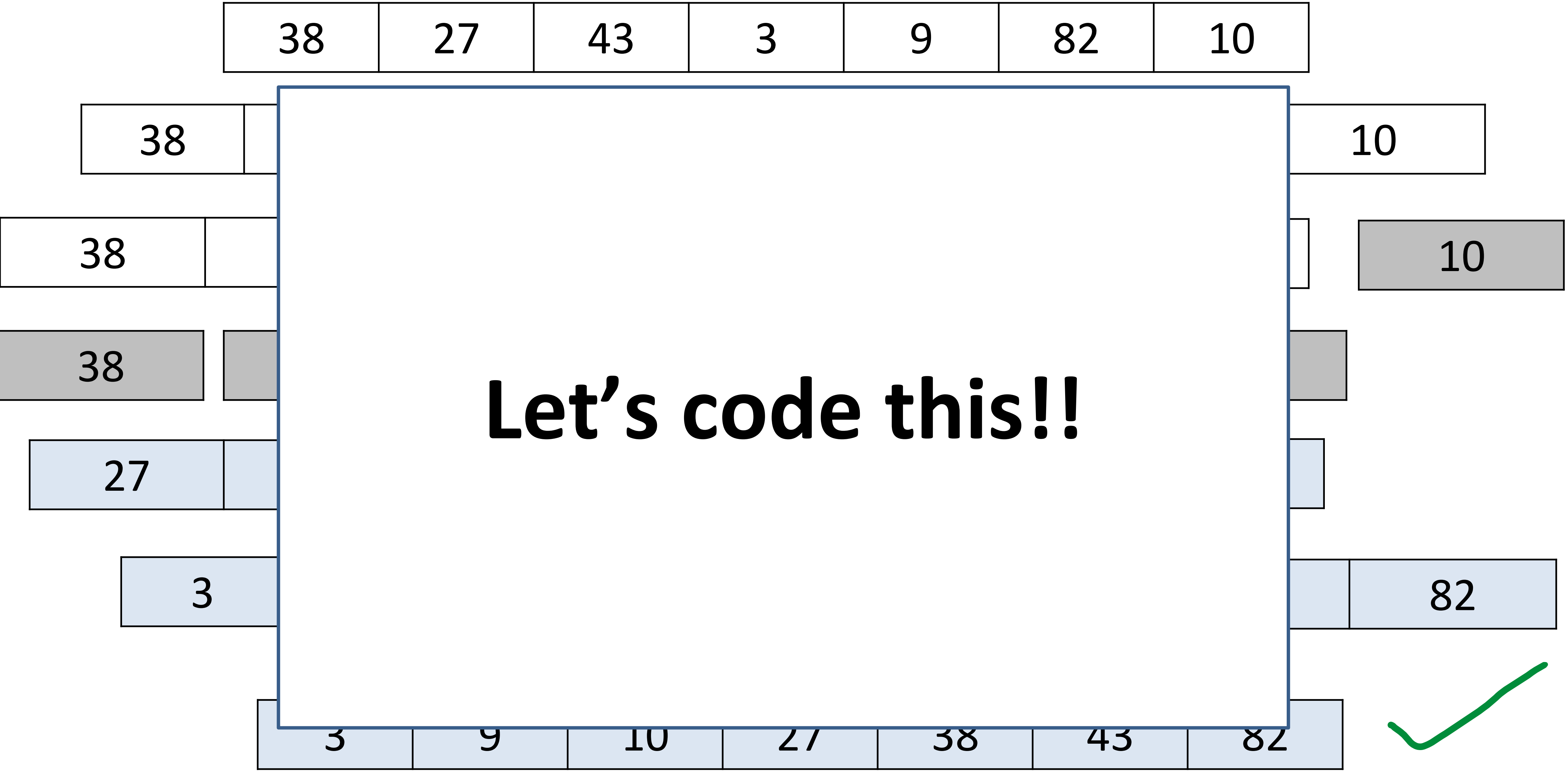
| | |
|---|---|
| 9 | 82 |

| | | | |
|---|---|---|---|
| 3 | 27 | 38 | 43 |


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
27 38
3 43
9 82
3 27 38 43


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


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




# Let’s code this!!

# Running time of merge sort??

# Running time of merge sort??

Running time = number of recursive calls made* $\cdot$ amount of work done in each call

*for merge sort, this won't lead us to the correct answer

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length;
  if (inputLength < 2) {
    return inputArray;
  }
  int midIndex = inputLength / 2;
  int[] leftHalf = new int[midIndex];
  int[] rightHalf = new int[inputLength - midIndex];
  for (int i = 0; i < midIndex; i++) {
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) {
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf);
  merge_sort(rightHalf);
  inputArray = merge(inputArray, leftHalf, rightHalf);
  return inputArray;
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;   O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex];
  int[] rightHalf = new int[inputLength - midIndex];
  for (int i = 0; i < midIndex; i++) {
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) {
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf);
  merge_sort(rightHalf);
  inputArray = merge(inputArray, leftHalf, rightHalf);
  return inputArray;
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;   O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) {
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) {
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf);
  merge_sort(rightHalf);
  inputArray = merge(inputArray, leftHalf, rightHalf);
  return inputArray;
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;   O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) { O(n/2)
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) { O(n/2)
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf);
  merge_sort(rightHalf);
  inputArray = merge(inputArray, leftHalf, rightHalf);
  return inputArray;
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;   O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) { O(n/2)
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) { O(n/2)
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf); O(1)
  merge_sort(rightHalf); O(1)
  inputArray = merge(inputArray, leftHalf, rightHalf);  O(???)
  return inputArray;
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;  O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) { O(n/2)
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) { O(n/2)
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf); O(1)
  merge_sort(rightHalf); O(1)
  inputArray = merge(inputArray, leftHalf, rightHalf);  O(???)
  return inputArray;
}
```

amount of work done in each call?

```java
private static int[] merge (int[] inputArray, int[]leftHalf, int[] rightHalf) {
    int leftSize = leftHalf.length;
    int rightSize = rightHalf.length;
    int i = 0, j = 0, k = 0;
    while (i < leftSize && j < rightSize) {
        if (leftHalf[i] <= rightHalf[j]) {
            inputArray[k] = leftHalf[i];
            i++;
        }
        else {
            inputArray[k] = rightHalf[j];
            j++;
        }
        k++;
    }
   while (i < leftSize) {
      inputArray[k] = leftHalf[i];
      i++;
      k++;
   }
   while (j < rightSize) {
      inputArray[k] = rightHalf[j];
      j++;
      k++;
   }
   return inputArray;
}
```

amount of work done in each call?

```
private static int[] merge (int[] inputArray, int[]leftHalf, int[] rightHalf) {
   int leftSize = leftHalf.length;  O(1)
   int rightSize = rightHalf.length; O(1)
   int i = 0, j = 0, k = 0;
   while (i < leftSize && j < rightSize) {  O(n)
      if (leftHalf[i] <= rightHalf[j]) {
         inputArray[k] = leftHalf[i];
         i++;                                  O(1)
      }
      else {
         inputArray[k] = rightHalf[j];
         j++;                                  O(1)
      }
      k++;
  }
  while (i < leftSize) {
     inputArray[k] = leftHalf[i];
     i++;
     k++;
  }
  while (j < rightSize) {
     inputArray[k] = rightHalf[j];
     j++;
     k++;
  }
  return inputArray;
}
```

# amount of work done in each call?

```
private static int[] merge (int[] inputArray, int[]leftHalf, int[] rightHalf) {
   int leftSize = leftHalf.length;  O(1)
   int rightSize = rightHalf.length; O(1)
   int i = 0, j = 0, k = 0;
   while (i < leftSize && j < rightSize) { O(n)
      if (leftHalf[i] <= rightHalf[j]) {
         inputArray[k] = leftHalf[i];   O(1)
         i++;
      }
      else {
         inputArray[k] = rightHalf[j];  O(1)
         j++;
      }
      k++;
  }
  while (i < leftSize) {       O(n/2)
     inputArray[k] = leftHalf[i];
     i++;                     O(1)
     k++;
  }
  while (j < rightSize) {      O(n/2)
     inputArray[k] = rightHalf[j];
     j++;                     O(1)
     k++;
  }
  return inputArray;  O(1)
}
```

$O(n) + O(n/2) + O(n/2) = O(2n)$

Running time of `merge` subroutine

## $O(n)$

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;  O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) { O(n/2)
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) { O(n/2)
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf); O(1)
  merge_sort(rightHalf); O(1)
  inputArray = merge(inputArray, leftHalf, rightHalf);  O(n)
  return inputArray; O(1)
}
```

amount of work done in each call?

```
public static int[] merge_sort(int[] inputArray) {
  int inputLength = inputArray.length; O(1)
  if (inputLength < 2) {
    return inputArray;   O(1)
  }
  int midIndex = inputLength / 2; O(1)
  int[] leftHalf = new int[midIndex]; O(n/2)
  int[] rightHalf = new int[inputLength - midIndex]; O(n/2)
  for (int i = 0; i < midIndex; i++) { O(n/2)
    leftHalf[i] = inputArray[i];
  }
  for (int i = midIndex; i < inputLength; i++) { O(n/2)
    rightHalf[i - midIndex] = inputArray[i];
  }
  merge_sort(leftHalf); O(1)
  merge_sort(rightHalf); O(1)
  inputArray = merge(inputArray, leftHalf, rightHalf);  O(n)
  return inputArray; O(1)
}
```

$O(n) + O(n/2) + O(n/2) + O(n/2) + O(n)$

Total running time of a single merge_sort call:

**$O(n)$**

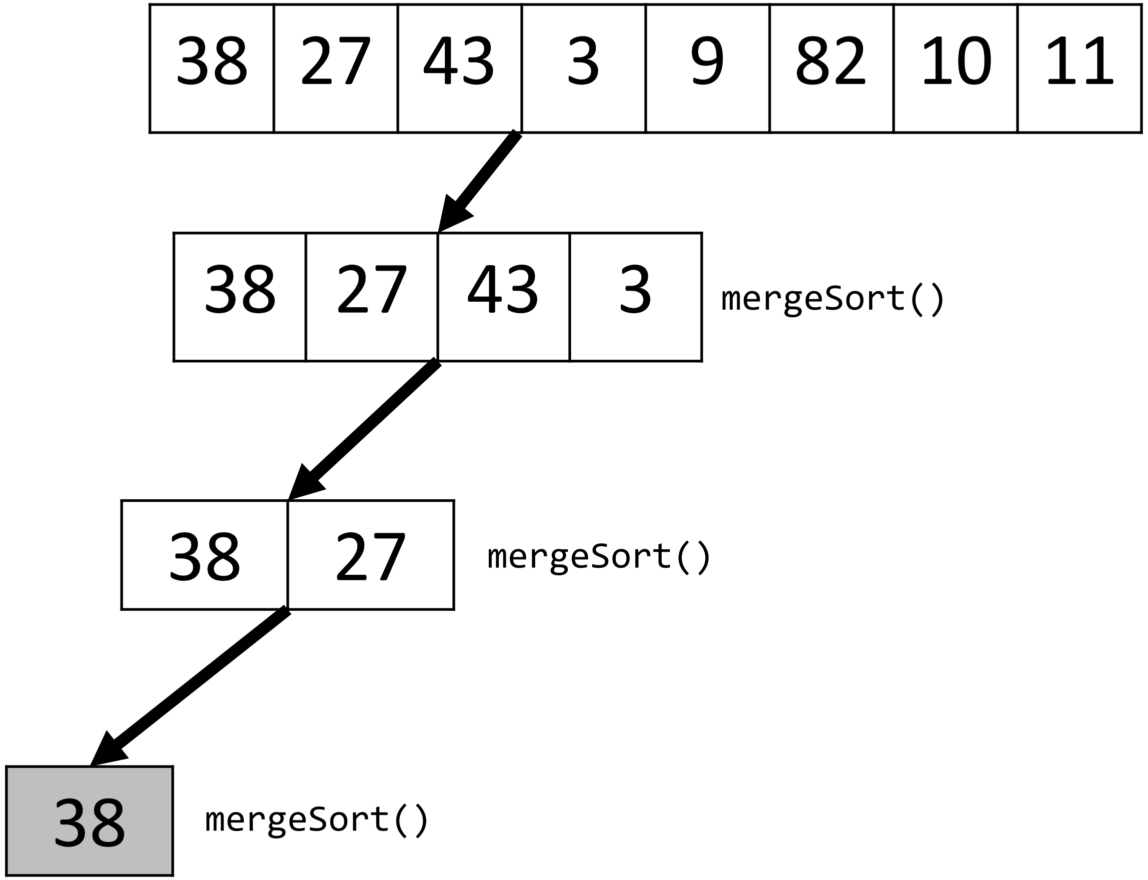
# Running time of merge sort??

Running time = number of recursive calls made* · amount of work done in each call

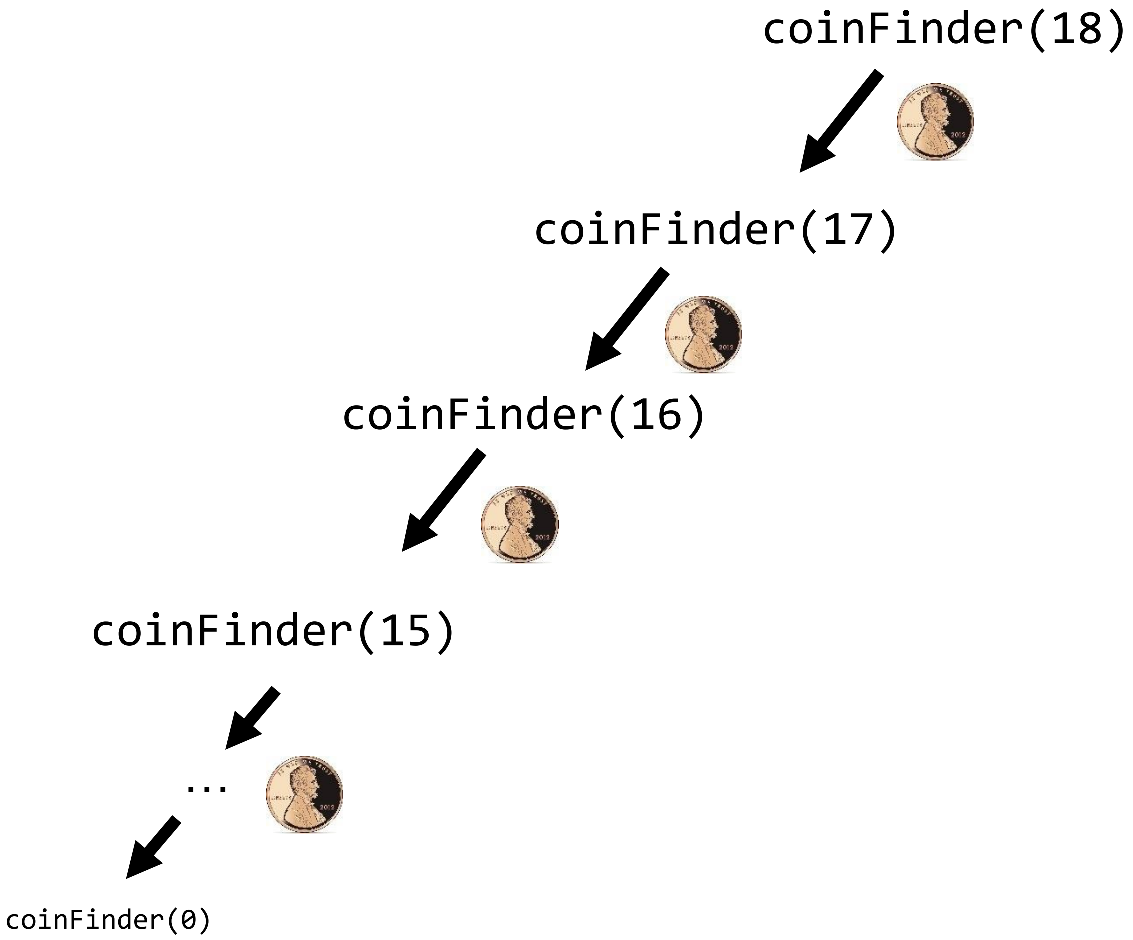
Running time = ??? * **O(n)**

*for merge sort, this won't lead us to the correct answer

# Merge Sort




# Change Making (coinFinder)

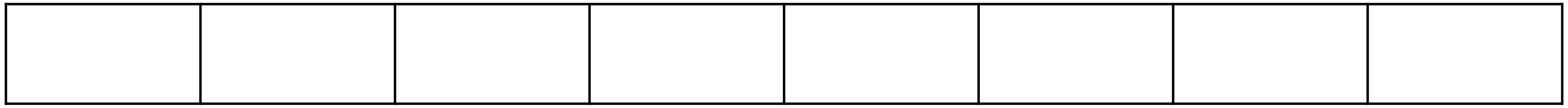


# Running time of merge sort??

Running time = number of recursive calls made* · amount of work done in each call

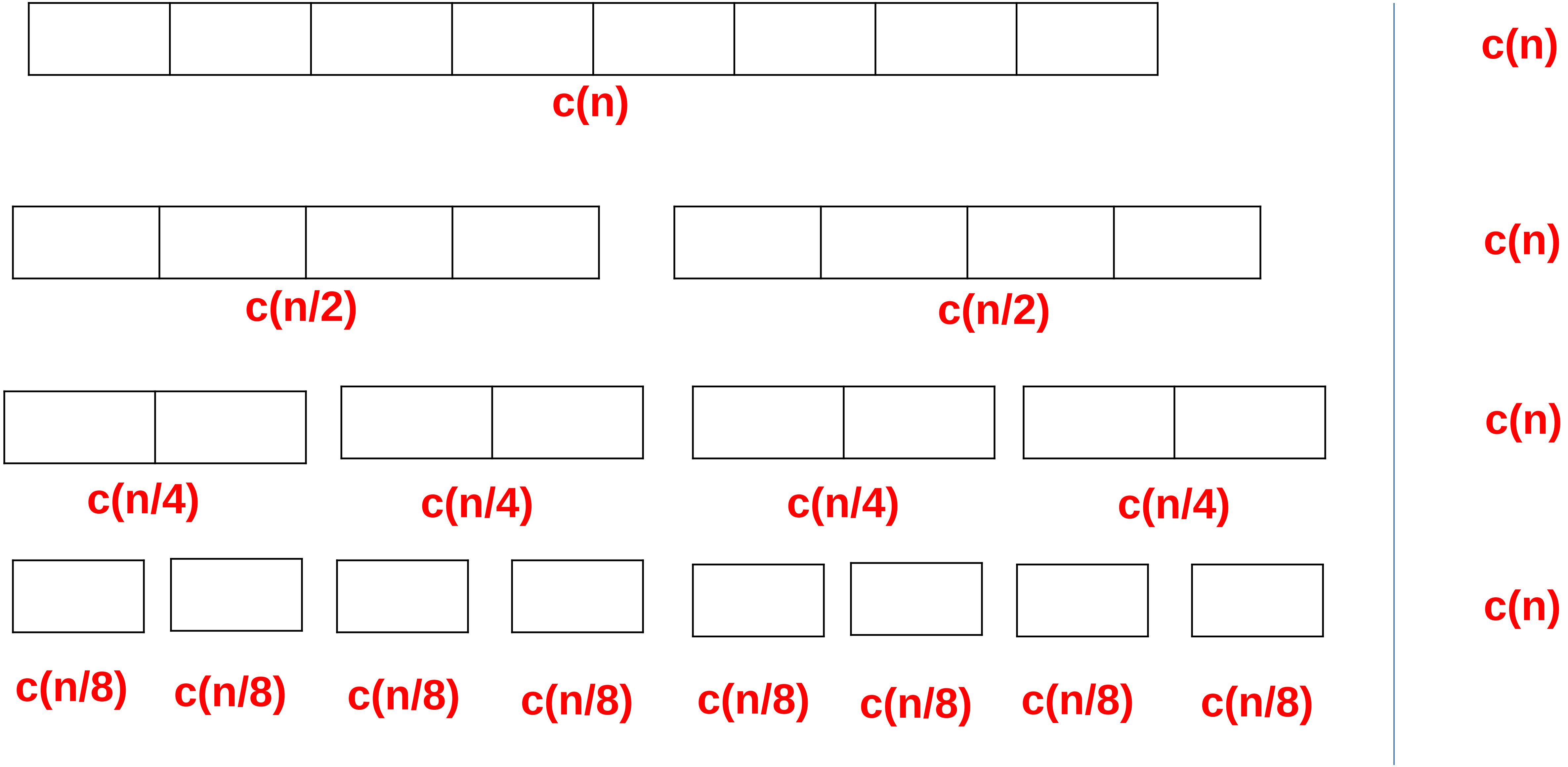
Running time = ??? * **O(n)**

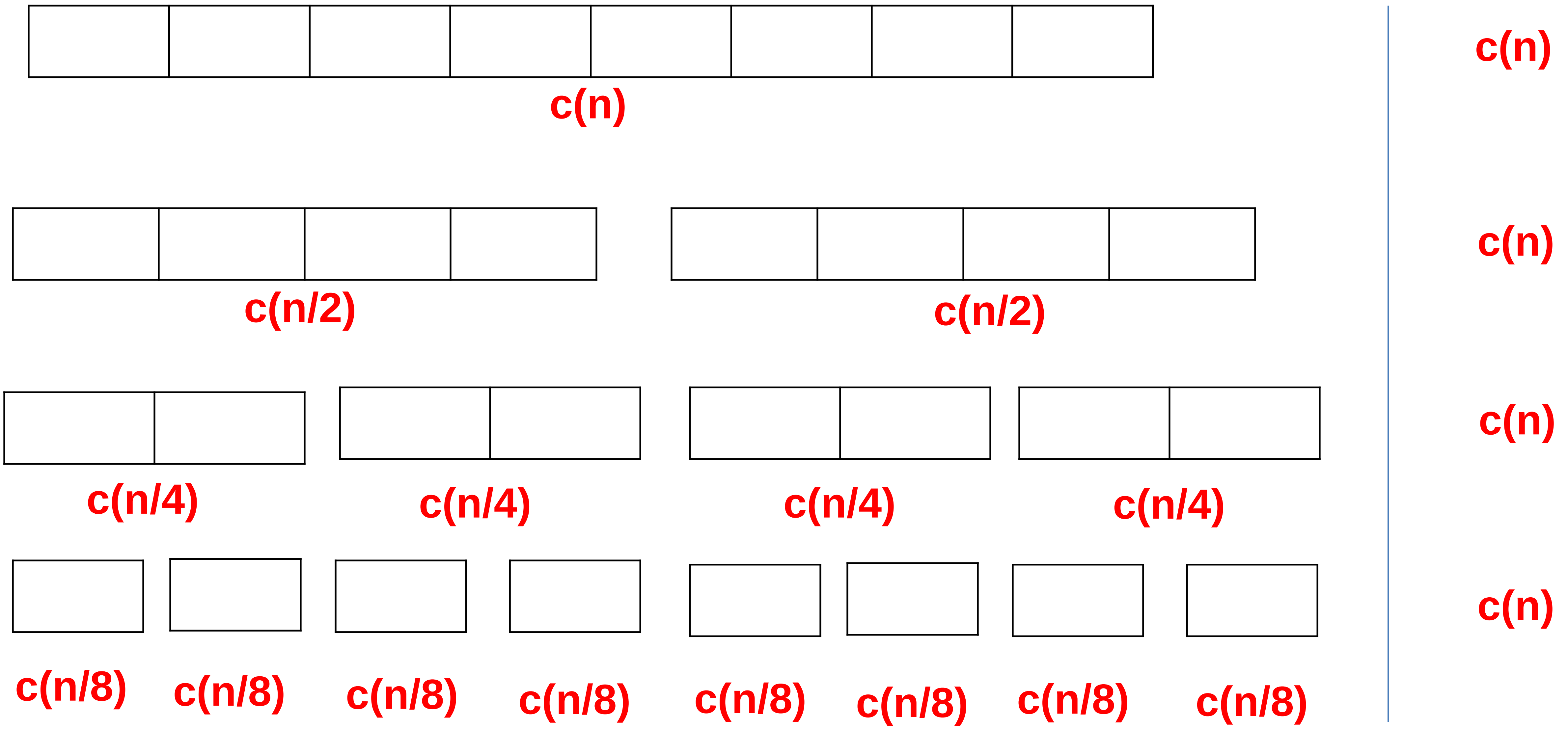
When we recursively call our method when dividing, we give a problem **that is half the size** of the original problem



c(n)

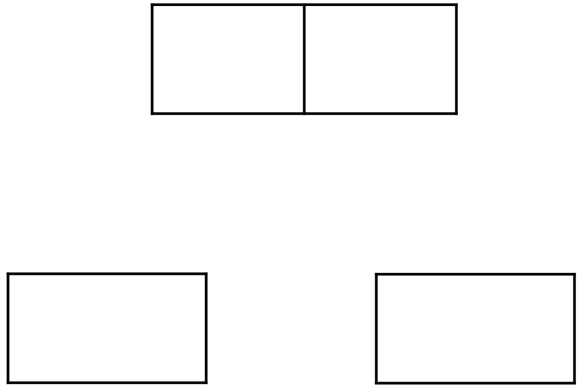
Suppose that the cost of solving a problem of size n can be expressed as c(n)


c(n)
c(n)
c(n/2)
c(n/2)
c(n)
c(n/4)
c(n/4)
c(n/4)
c(n/4)
c(n)
c(n/8)
c(n/8)
c(n/8)
c(n/8)
c(n/8)
c(n/8)
c(n/8)
c(n/8)
c(n)

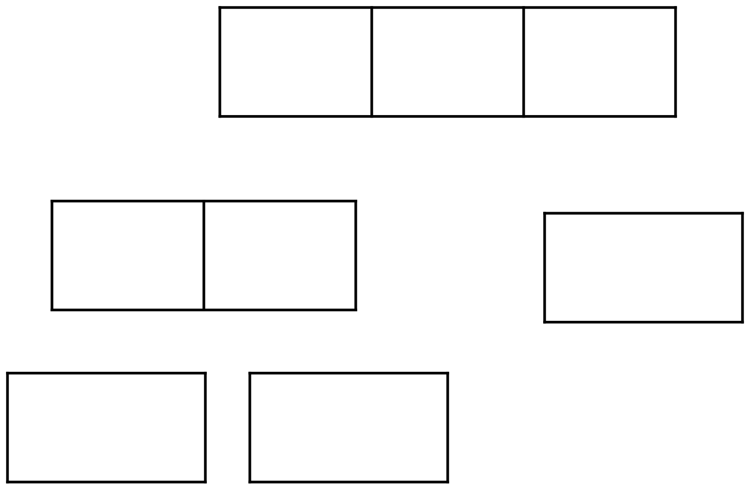



How much do we divide (*in regards to n*)?  AKA **what is the height of the recursion tree**?

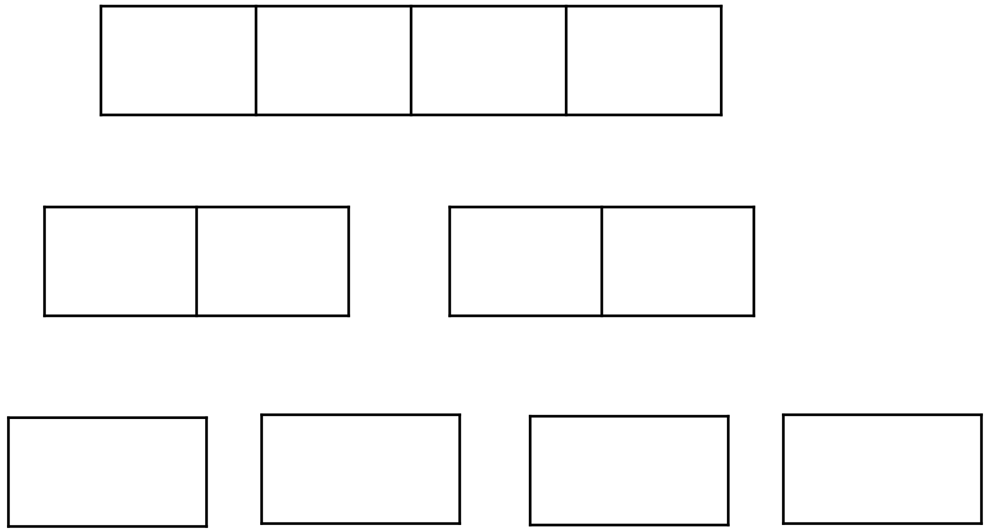
When n = 2, the height is 2



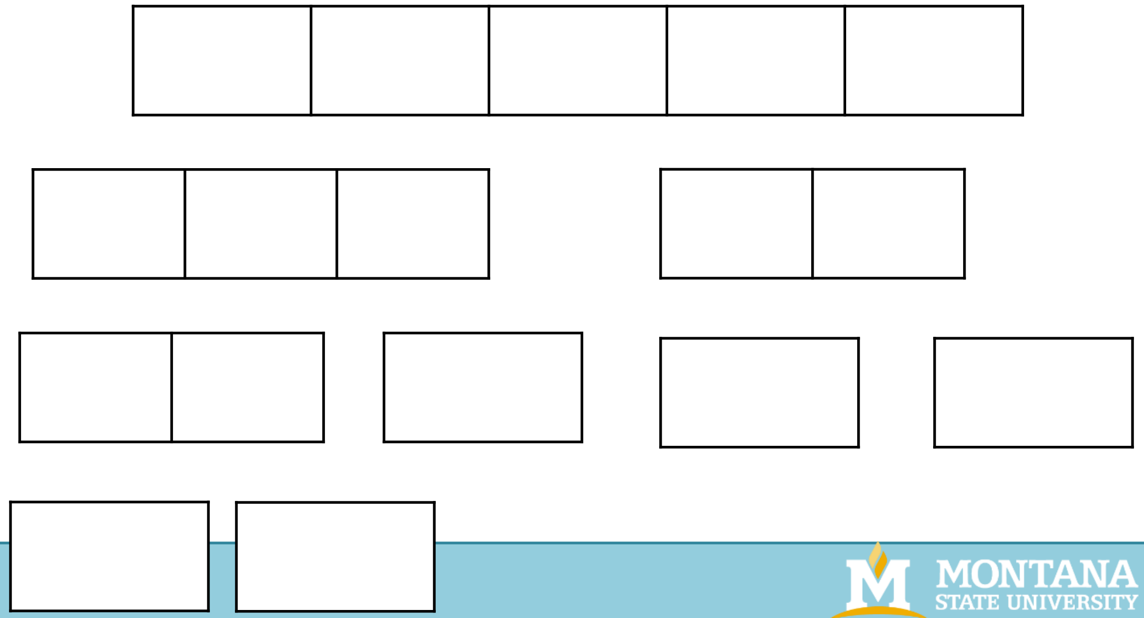
When n = 3, the height is 3



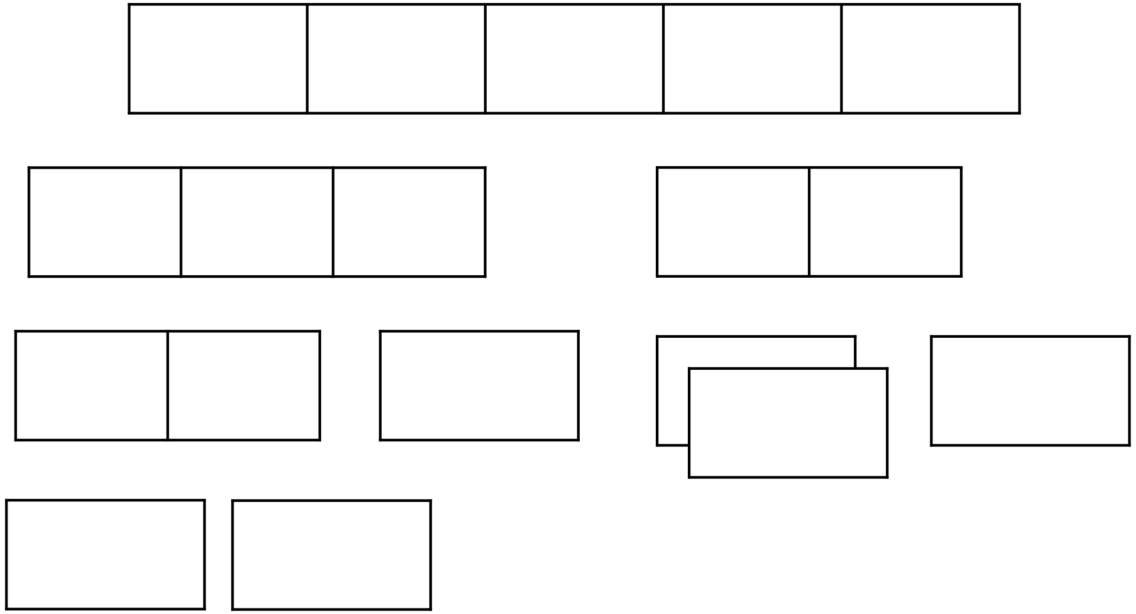
When n = 4, the height is 3



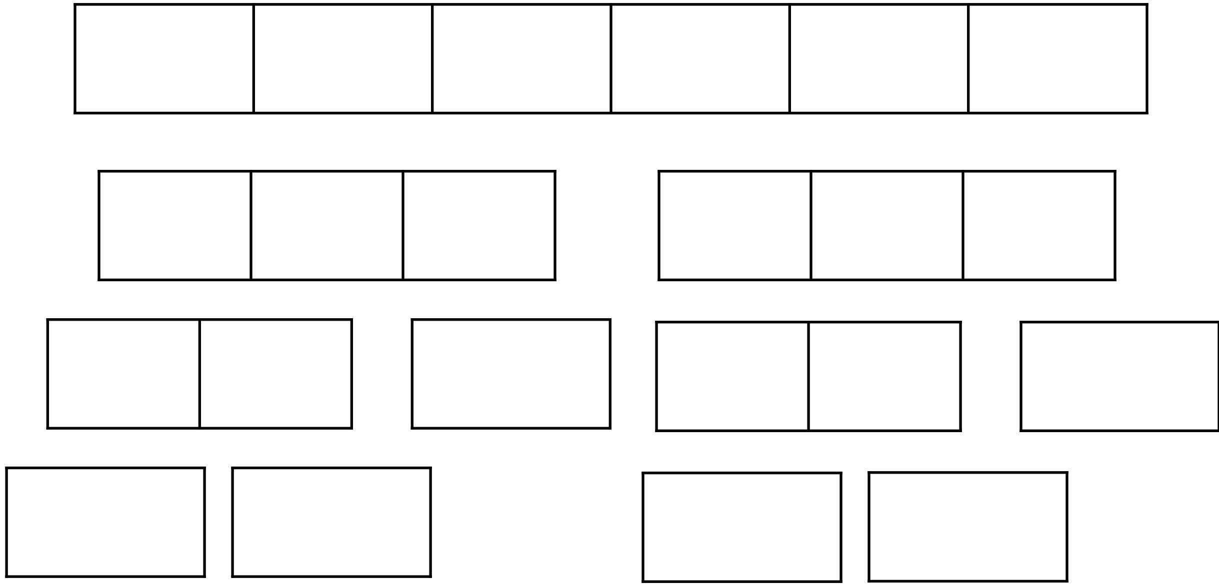
When n = 5, the height is 4



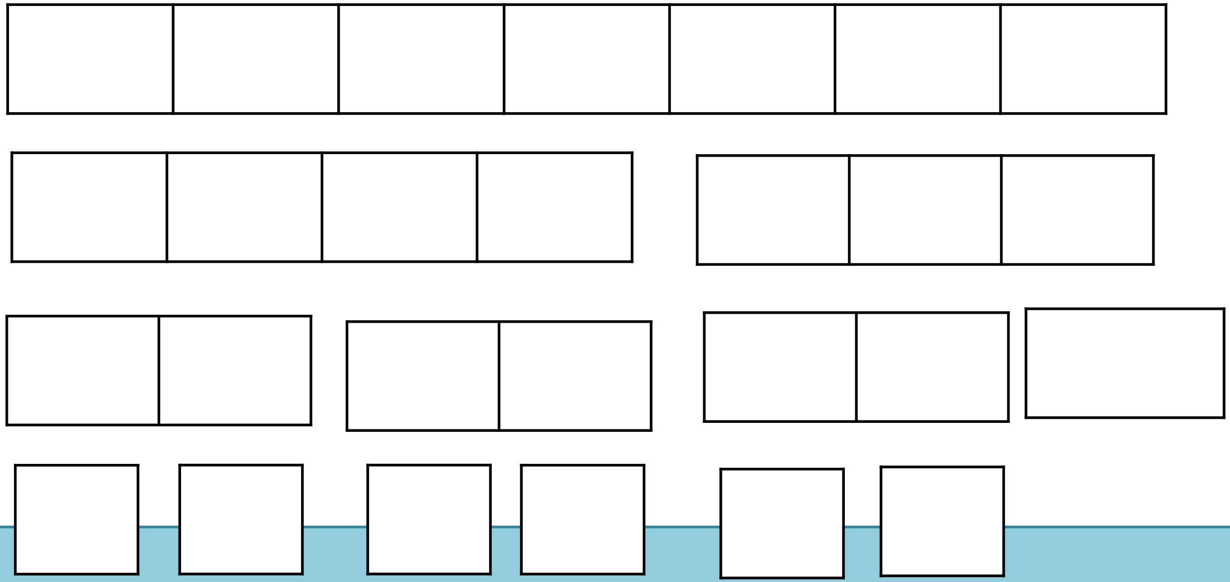
When n = 5, the height is 4



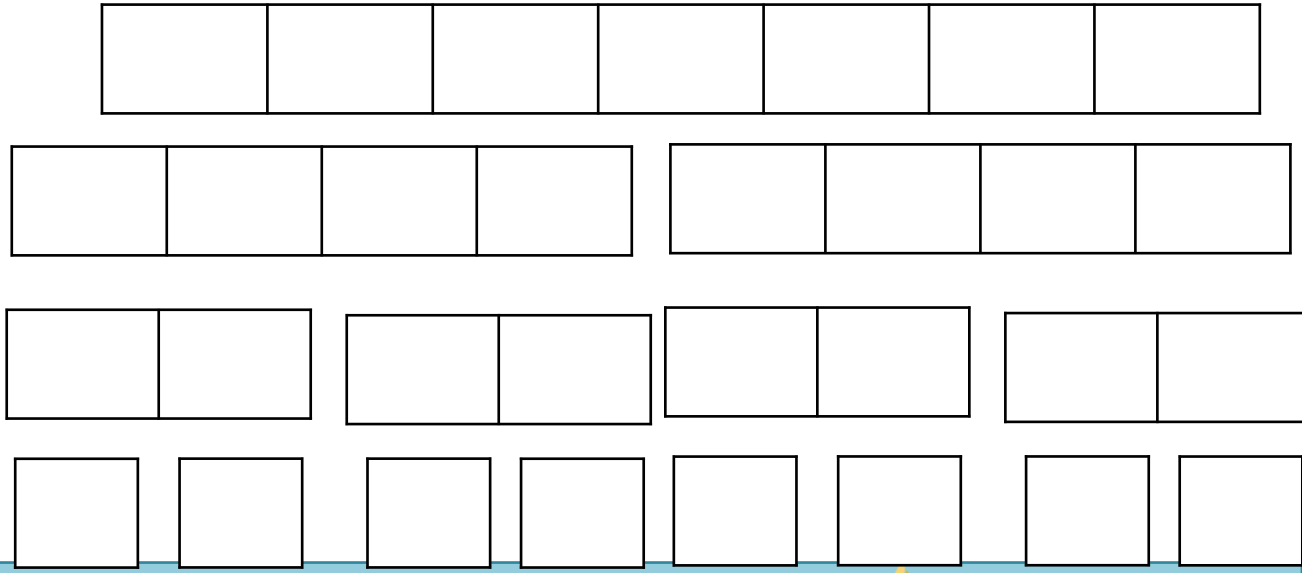
When n = 6, the height is 4



When n = 7, the height is 4

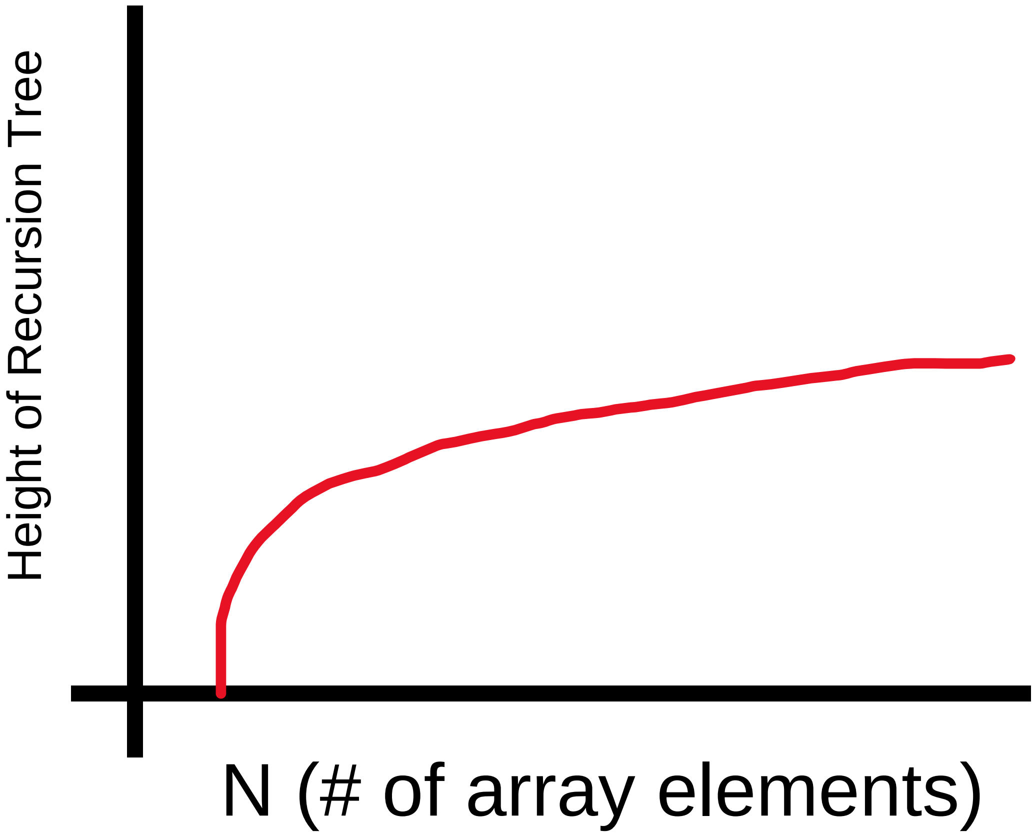


When n = 8, the height is 4



What is the growth rate of the height of our recursion tree (the # of recursive calls made) ?

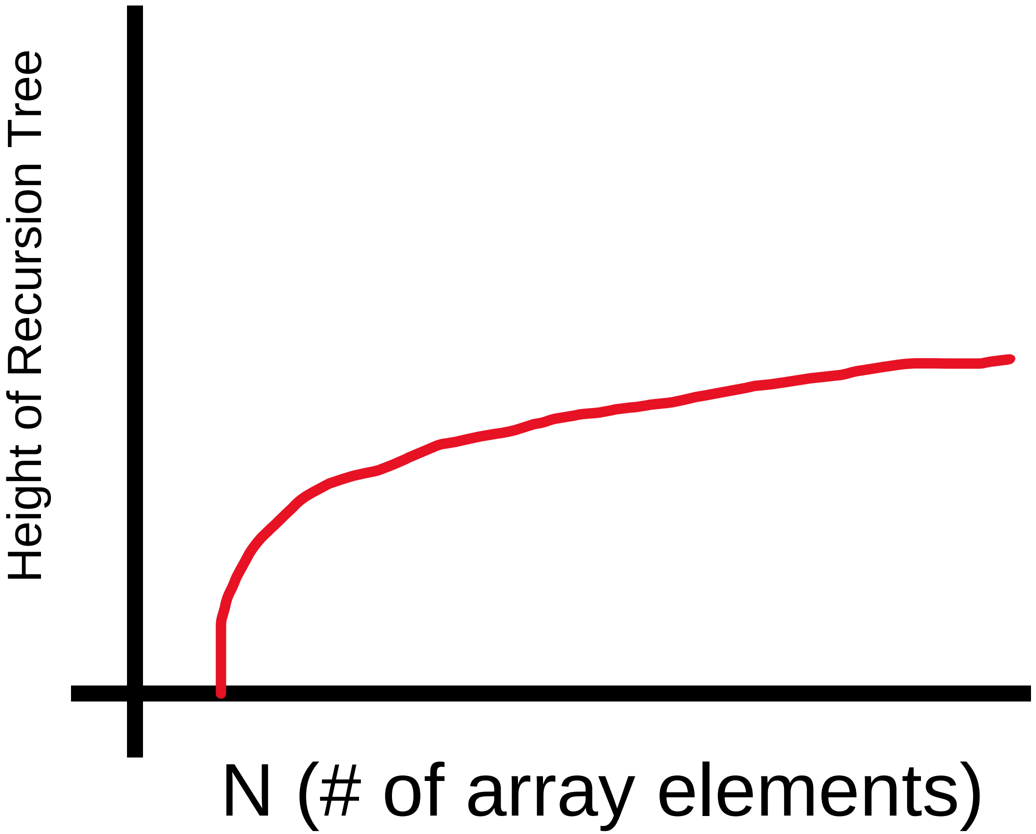
If we were to continue this counting, the graph would look something like this:




???

What is the growth rate of the height of our recursion tree (the # of recursive calls made) ?

If we were to continue this counting, the graph would look something like this:

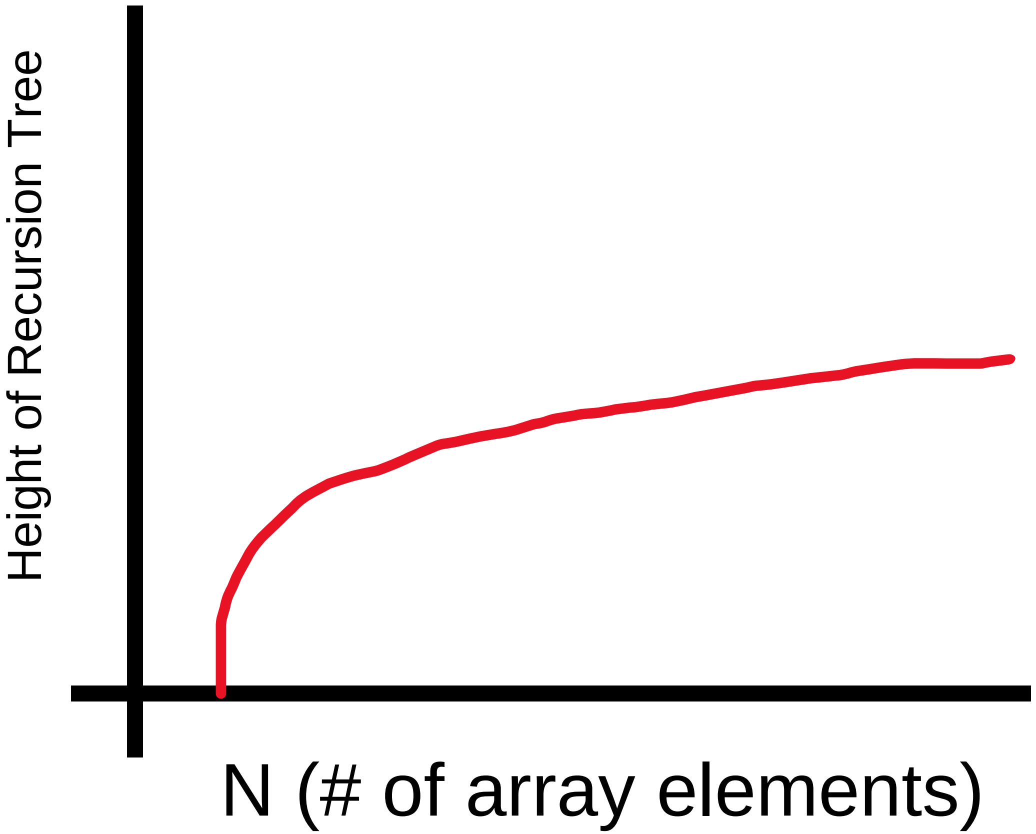



Logarithmic !!

log(n)

What is the growth rate of the height of our recursion tree (the # of recursive calls made) ?

If we were to continue this counting, the graph would look something like this:




# Logarithmic !!

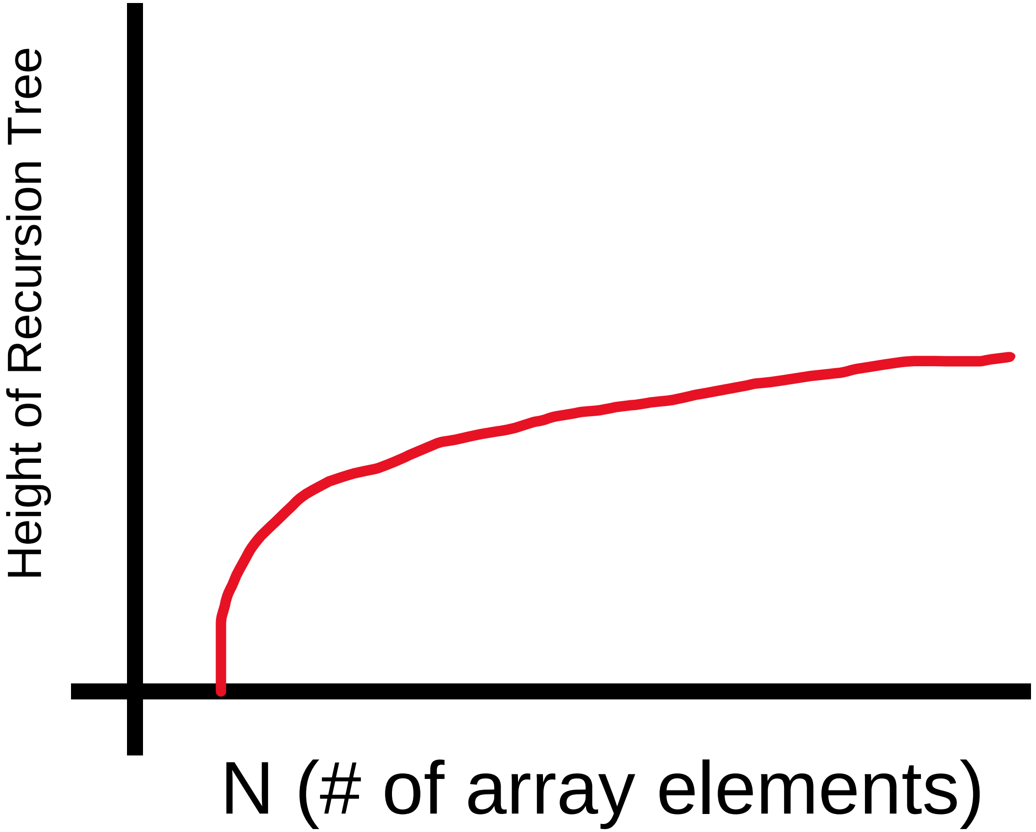
$$\log_2(n)$$

It will actually be log base 2, because we are dividing our array in half in each recursive call

However, in computer science, all logarithms are to the base 2 unless specified otherwise

What is the growth rate of the height of our recursion tree (the # of recursive calls made) ?

If we were to continue this counting, the graph would look something like this:



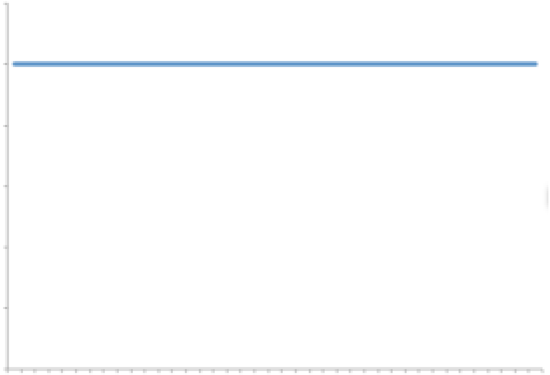

Logarithmic !!

log(n)

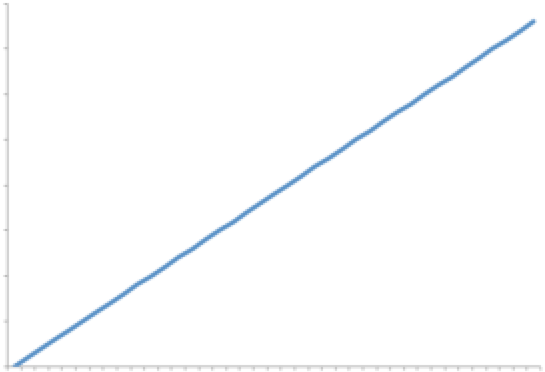
# Growth Rates

We have a new member of the family!

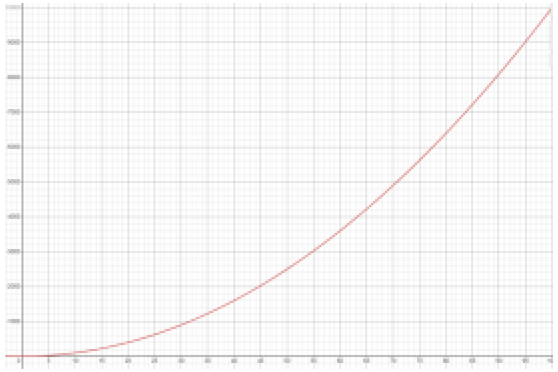
Constant



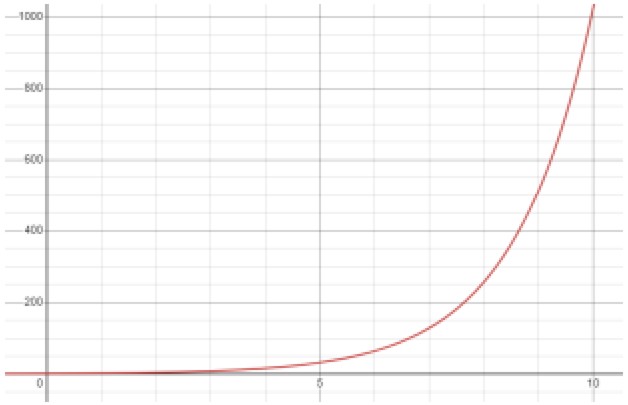
Linear



Quadratic



Exponential

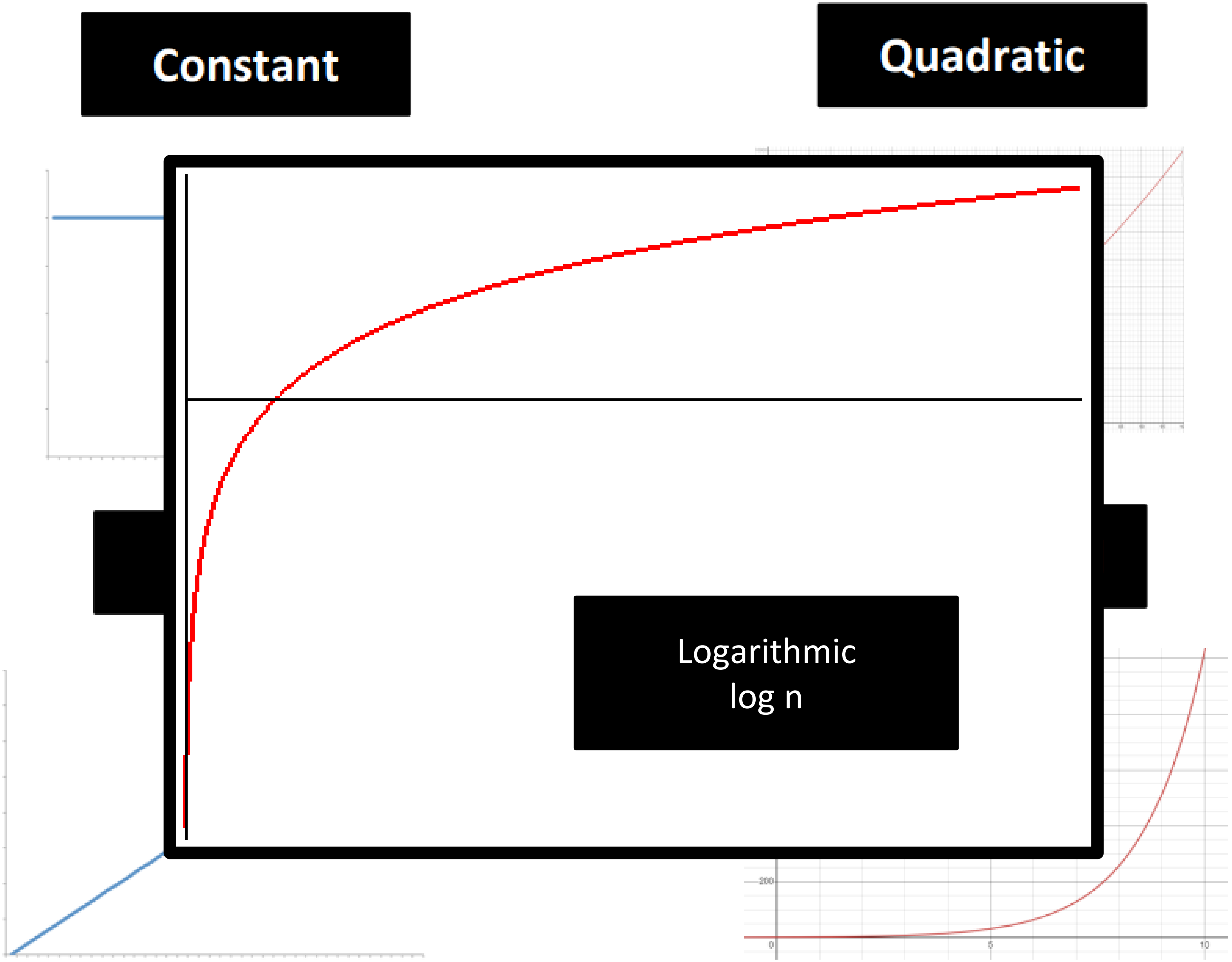


# Growth Rates

We have a new member of the family!

log n is *smaller* than n

algorithms that run in **O(log n)** time are good!



# Running time of merge sort??

Height of recursive tree

Running time = ~~number of recursive calls made~~ $\cdot$ amount of work done in each call

Running time = ??? * **O(n)**

# Running time of merge sort??

Height of recursive tree

Running time = ~~number of recursive calls made~~ · amount of work done in each call

Running time = O(log n) * **O(n)**

Running time of merge sort = **O(n * log n)**

# Running time of merge sort??

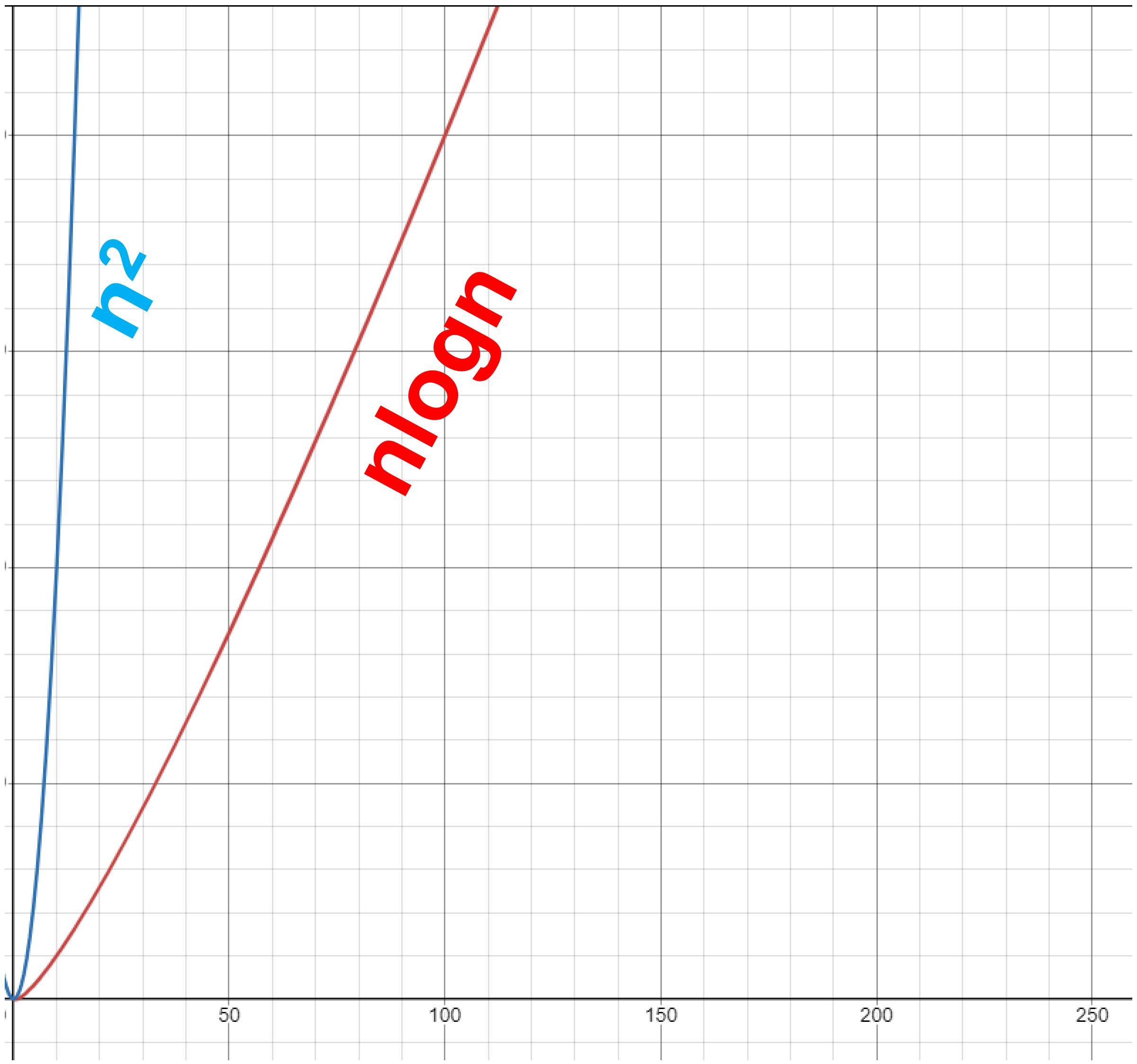
Height of recursive tree

Running time = ~~number of recursive calls made~~ · amount of work done in each call

Running time = O(log n) * **O(n)**

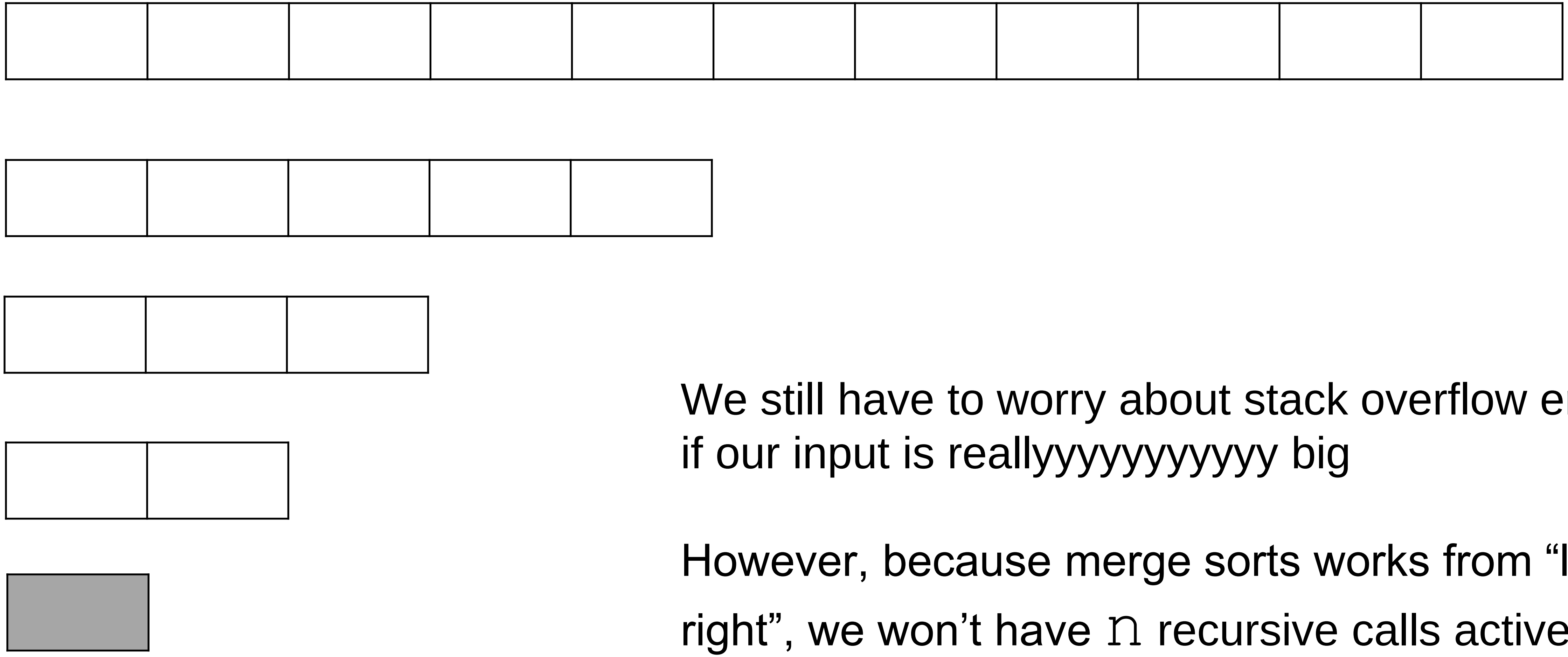
Running time of merge sort = **O(n * log n)**

This is **much** faster than O(n^2)



n²
nlogn
50
100
150
200
250

# What about stack overflow errors?



We still have to worry about stack overflow errors if our input is reallyyyyyyyyyyy big

However, because merge sorts works from “left to right”, we won’t have $n$ recursive calls active, and we are much more efficient with how many method calls we put on call stack