Section 8.13
Divide-and-Conquer Algorithms: Introduction and Mergesort

Divide-and-Conquer Algorithms

- A Divide-and-Conquer algorithm divides a problem into subproblems, solves the subproblems and then combines the solutions to form a solution to the original problem

FindMin

- Example: Use divide-and-conquer to find the minimum of a group of numbers
  - Define a FindMin function that takes a nonempty sequence of numbers as input
    - If the sequence contains only one number, then that number is the minimum
    - Otherwise, divide the sequence into two subsequences of more-or-less equal length
    - Recursively find the minimum of each subsequence
    - The minimum of the two subsequences is the minimum of the original sequence

FindMin

Example: Use divide-and-conquer to find the minimum of a group of numbers

```
Name: FindMin
Input: a sequence of n numbers, a_1…a_n
Output: the minimum of the input

if (n = 1)
  return a_1
else
  mid := ⌊n/2⌋
  min1 := FindMin(a_1…a_mid)
  min2 := FindMin(a_mid+1…a_n)
  if min1 < min2
    return min1
  else
    return min2
  end-if
end-if
```

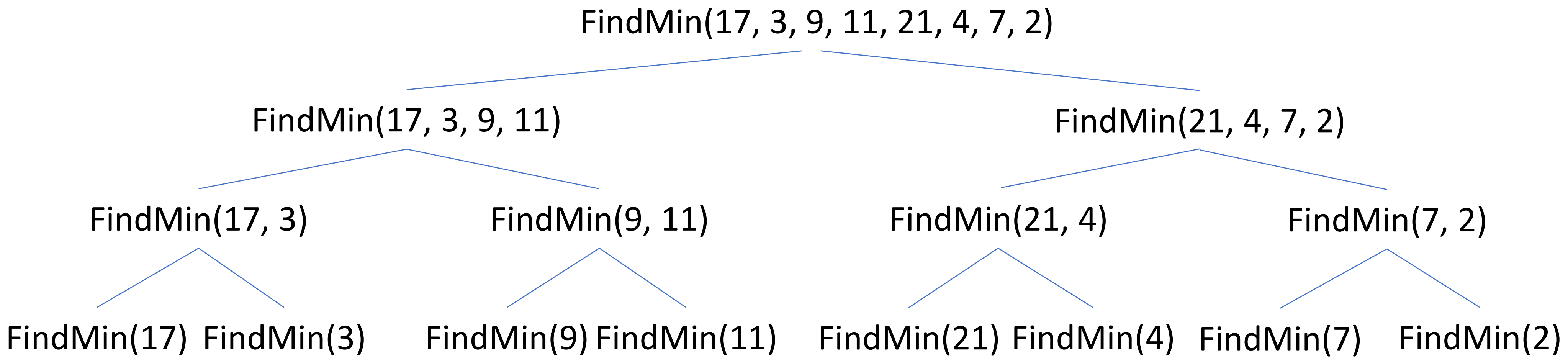
FindMin

A hand trace of FindMin(30, 40, 20, 10)

FindMin

Example: Use divide-and-conquer to find the minimum of a group of numbers

- A tree of recursive calls would look like this:



FindMin

Let $T(n)$ be the number of operations used by FindMin when its input is n numbers

$$T(n) = 2T(n/2) + 8$$

```
Name: FindMin
Input: a sequence of n numbers, a_1…a_n
Output: the minimum of the input

if (n = 1)
  return a_1
else
  mid := ⌊n/2⌋
  min1 := FindMin(a_1…a_mid)
  min2 := FindMin(a_mid+1…a_n)
  if min1 < min2
    return min1
  else
    return min2
  end-if
end-if
```

MergeSort

- Example: Use Divide-and-Conquer to sort a sequence of numbers
  - Define a MergeSort function that takes a nonempty sequence of numbers as input
    - If the sequence contains only one number, then the sequence is sorted
    - Otherwise, divide the sequence into two subsequences of more-or-less equal length
    - Recursively sort the two subsequences
    - Merge the two subsequences to sort the original sequence

MergeSort

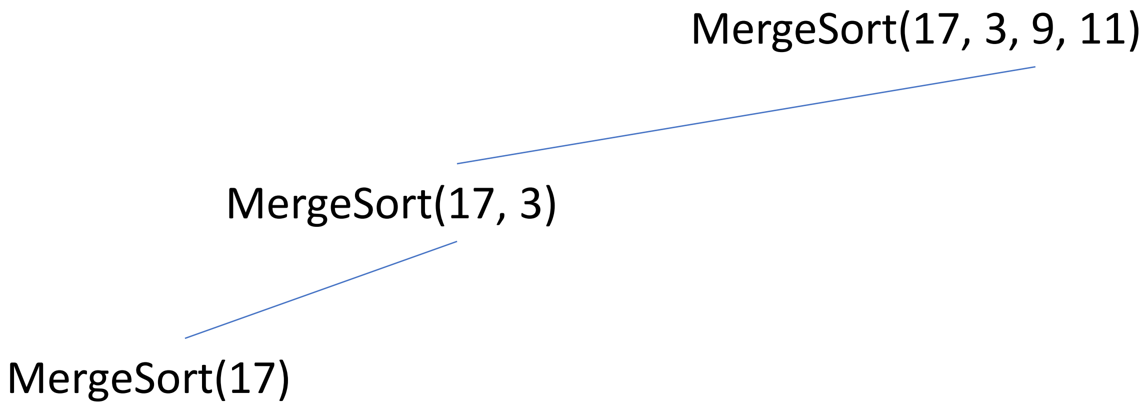
MergeSort(17, 3, 9, 11)

MergeSort

MergeSort(17, 3, 9, 11)

MergeSort(17, 3)

MergeSort



MergeSort

MergeSort(17, 3, 9, 11)

MergeSort(17, 3)

MergeSort(17)=17

MergeSort

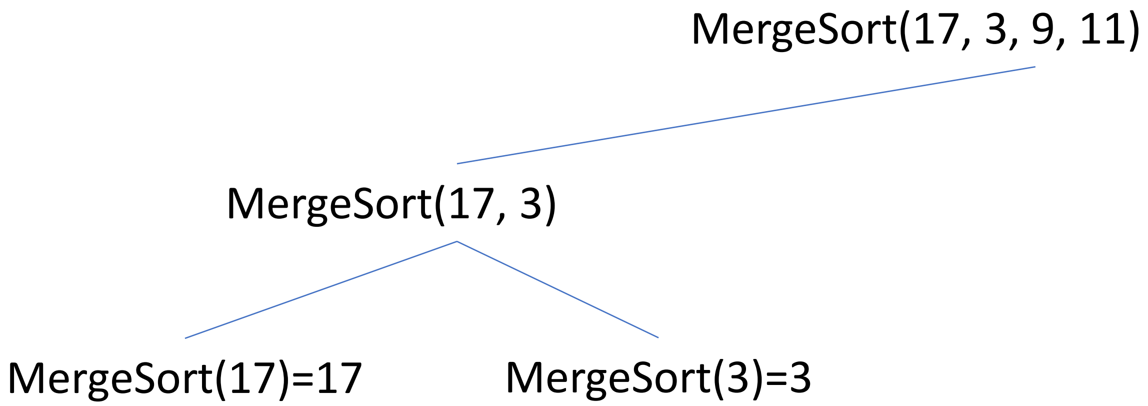
MergeSort(17, 3, 9, 11)

MergeSort(17, 3)

MergeSort(17)=17

MergeSort(3)

MergeSort



MergeSort(17, 3, 9, 11)
MergeSort(17, 3)
MergeSort(17)=17
MergeSort(3)=3

MergeSort

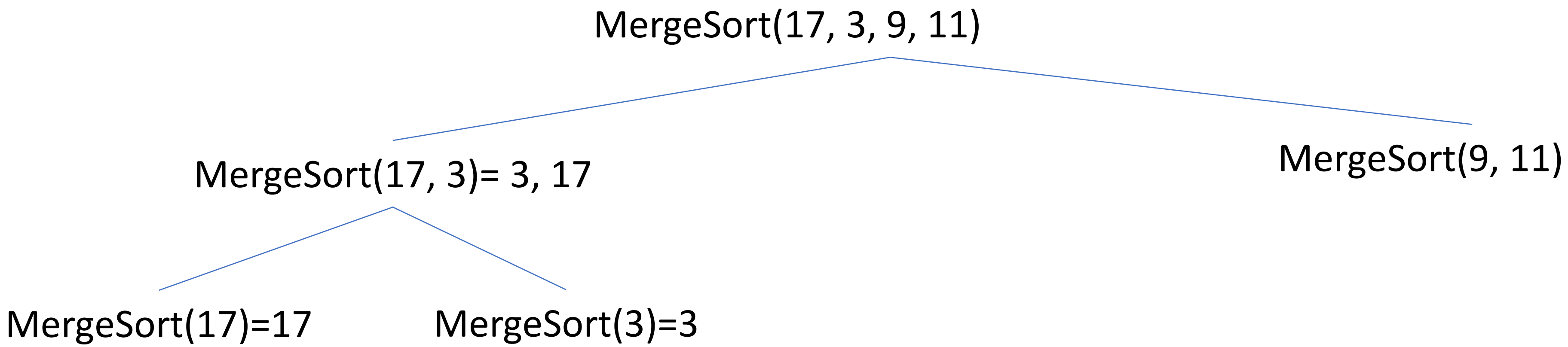
MergeSort(17, 3, 9, 11)

MergeSort(17, 3)= 3, 17

MergeSort(17)=17

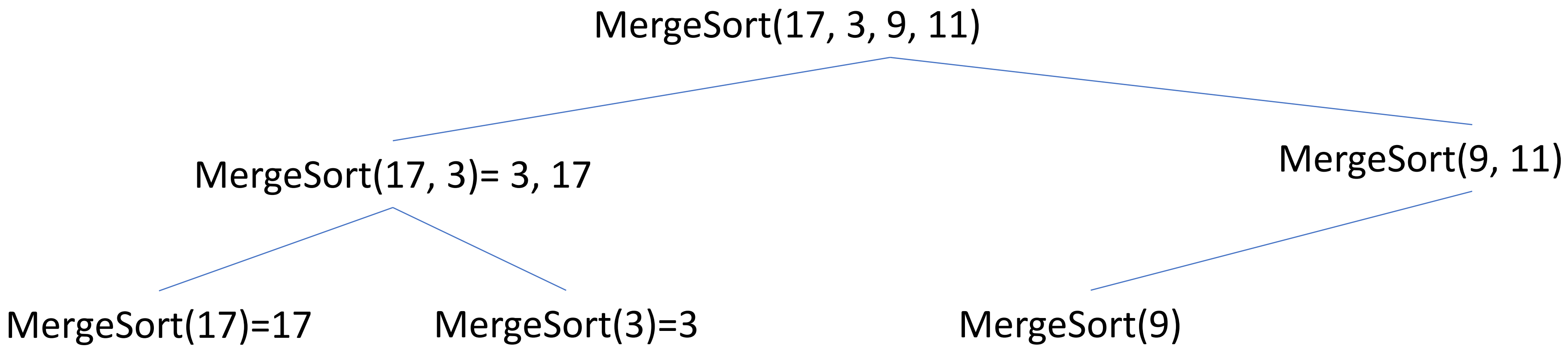
MergeSort(3)=3

MergeSort

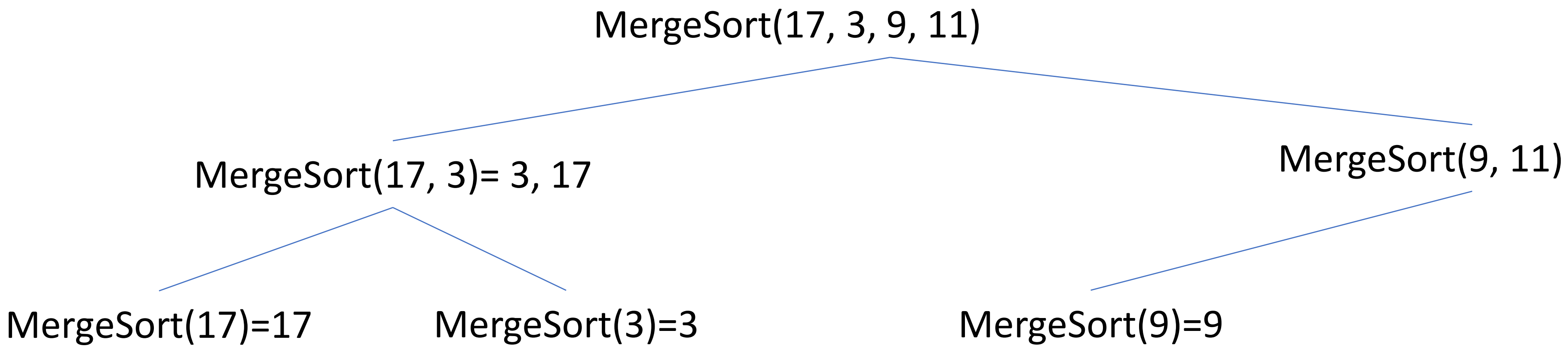


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)
MergeSort(17)=17
MergeSort(3)=3

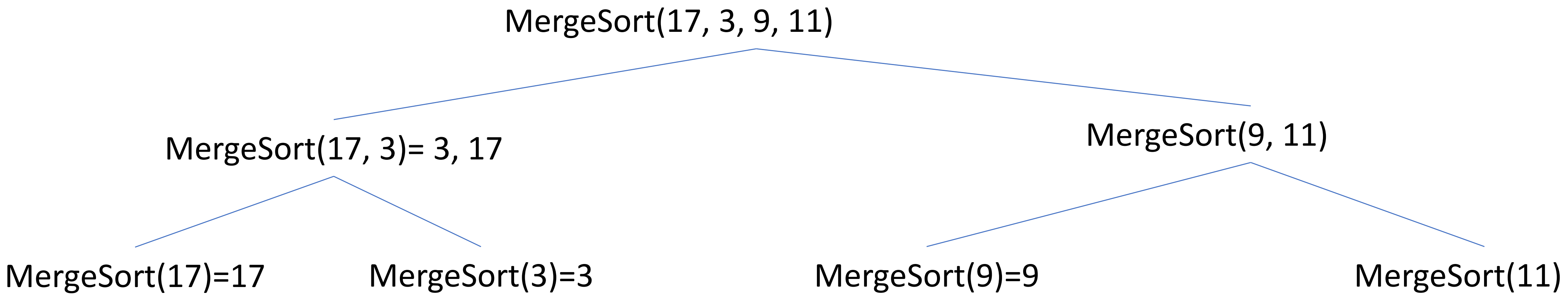
MergeSort


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)

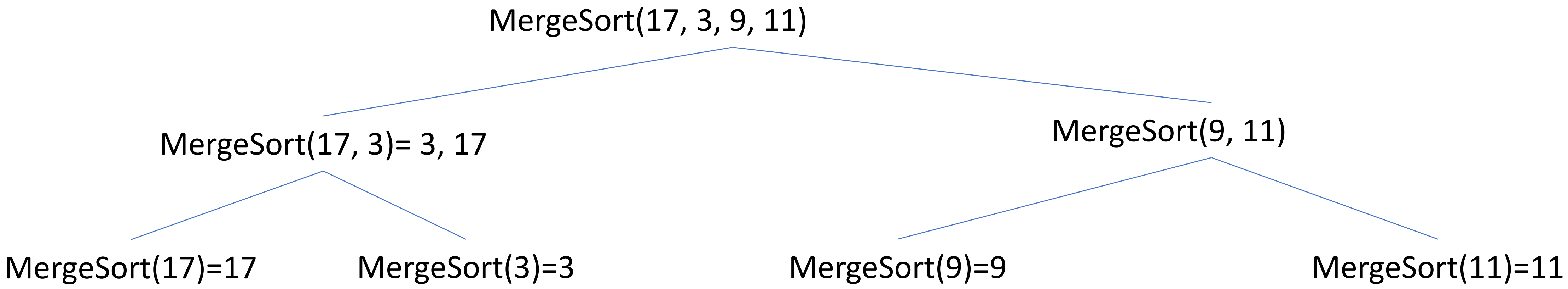
MergeSort


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)=9

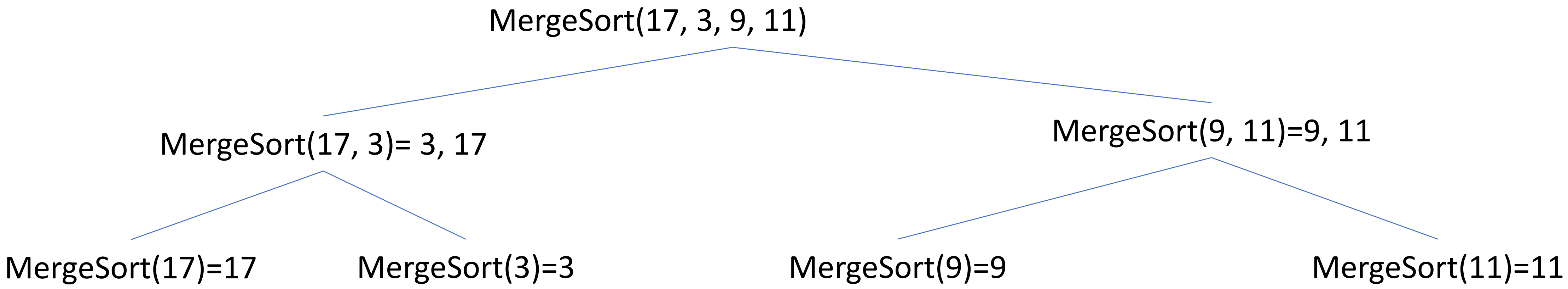
MergeSort


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)=9
MergeSort(11)

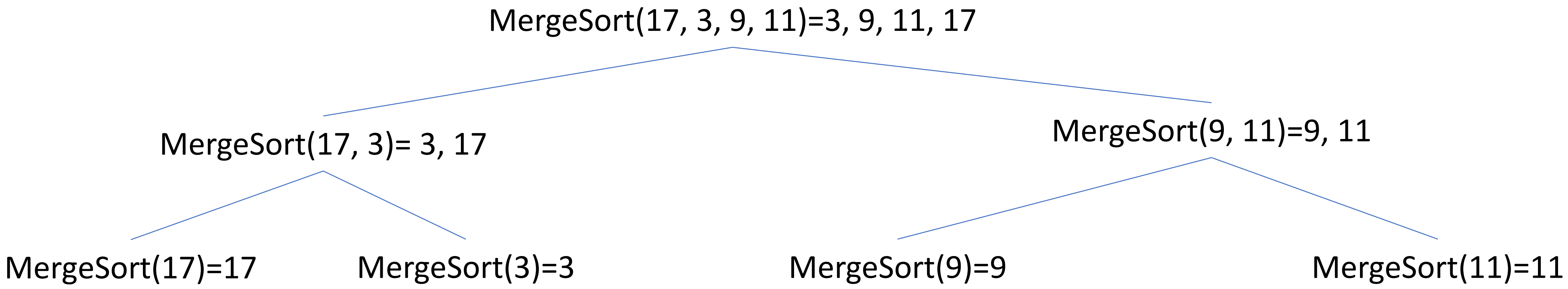
MergeSort


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)=9
MergeSort(11)=11

MergeSort


MergeSort(17, 3, 9, 11)
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)=9, 11
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)=9
MergeSort(11)=11

MergeSort



MergeSort(17, 3, 9, 11)=3, 9, 11, 17
MergeSort(17, 3)= 3, 17
MergeSort(9, 11)=9, 11
MergeSort(17)=17
MergeSort(3)=3
MergeSort(9)=9
MergeSort(11)=11

MergeSort

```
Name: MergeSort(a_1…a_n)
Input: a sequence of n numbers
Output: a sorted sequence of numbers

if (n = 1)
  return a_1
else
   mid := ⌊n/2⌋
   s1 := MergeSort(a_1…a_mid)
   s2 := MergeSort(a_mid+1…a_n)
   s := Merge(s1, s2)
   return s
end-if
```

MergeSort

```
Merge
Input: two sorted sequences of numbers
       a_1…a_m, and b_1…b_n
Output: a sorted sequence of numbers

Allocate a sequence, c, of m+n numbers
indexA := 1
indexB := 1
indexC := 1

while (indexA <= m and indexB <= n)
  if (a_indexA < b_indexB)
    c_indexC := a_indexA
    indexA := indexA + 1
  else
    c_indexC := b_indexB
    indexB := indexB + 1
  end-if
  indexC := indexC + 1
end-while
```

```
while (indexA <= m)
    c_indexC := a_indexA
    indexA := indexA + 1
    indexC := indexC + 1
  end-while
while (indexB <= n)
    c_indexC := b_indexA
    indexB := indexB + 1
    indexC := indexC + 1
  end-while

return c
```

Merge

```
Merge
Input: two sorted sequences of numbers
       a_1…a_m, and b_1…b_n
Output: a sorted sequence of numbers

Allocate a sequence, c, of m+n numbers
indexA := 1
indexB := 1
indexC := 1

while (indexA <= m and indexB <= n)
  if (a_indexA < b_indexB)
    c_indexC := a_indexA
    indexA := indexA + 1
  else
    c_indexC := b_indexB
    indexB := indexB + 1
  end-if
  indexC := indexC + 1
end-while
```

```
while (indexA <= m)
    c_indexC := a_indexA
    indexA := indexA + 1
    indexC := indexC + 1
  end-while
while (indexB <= n)
    c_indexC := b_indexA
    indexB := indexB + 1
    indexC := indexC + 1
  end-while

return c
```

In each of the three loops one element of either of the input sequences is assigned to the output sequence $c_1 \ldots c_{m+n}$. Therefore, the three loops in total are iterated m+n times. Each pass through a loop uses a constant number of operations so the total number of operations is $\Theta(m+n)$

MergeSort

```
Name: MergeSort(a₁…aₙ)
Input: a sequence of n numbers
Output: a sorted sequence of numbers

if (n = 1)
  return a₁
else
   mid := ⌊n/2⌋
   s1 := MergeSort(a₁…a_mid)
   s2 := MergeSort(a_mid+1…aₙ)
   s := Merge(s1, s2)
   return s
end-if
```

If $T(n)$ is the number of operations used by MergeSort to sort n numbers, then:

$$T(n) = 2T(n/2) + \Theta(n)$$

We abuse notation by writing $\Theta(n)$ for estimate of the number of operations used by Merge and the 8 other operations (which are $\Theta(1)$) used by MergeSort outside of the function calls