CS11313 - **Spring** 2022

Design & Analysis of Algorithms

Divide and Conquer & Merge Sort

Ibrahim Albluwi

Finding the Max

Problem. Given an array of n elements, find the maximum element.

Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Finding the Max

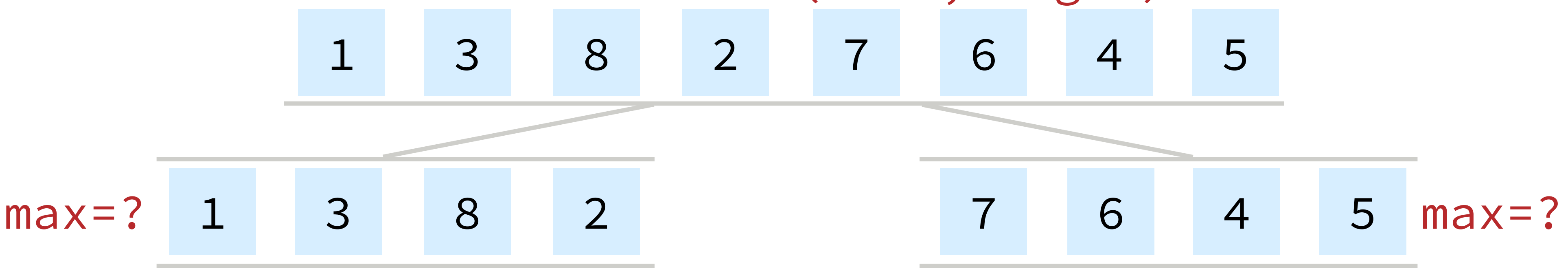
Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!

```
maximum = max(left, right)
```



1 3 8 2 7 6 4 5

max=? 1 3 8 2

7 6 4 5 max=?

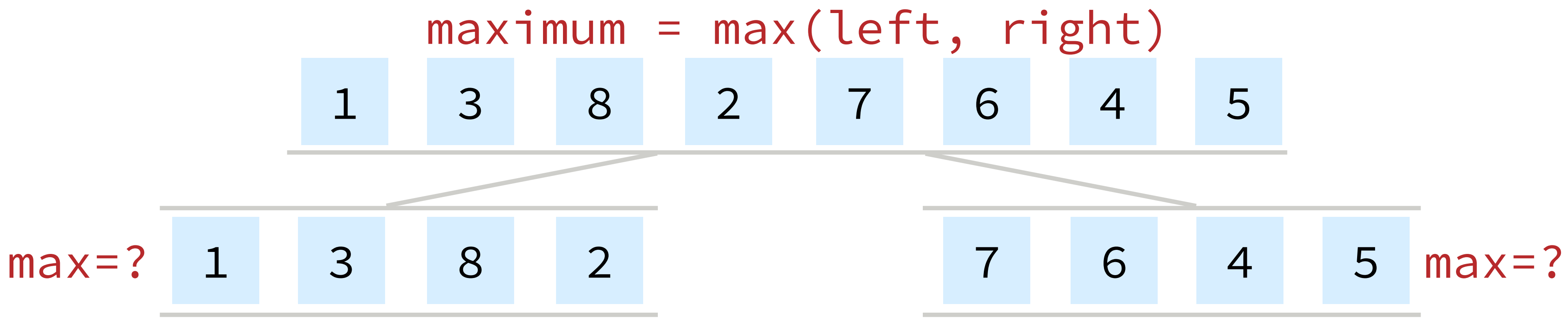
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



maximum = max(left, right)

1 3 8 2 7 6 4 5

max=? 1 3 8 2

7 6 4 5 max=?

You. But how do we find the max of each half? We now have 2 problems instead of one!

Wise man. Do the same!

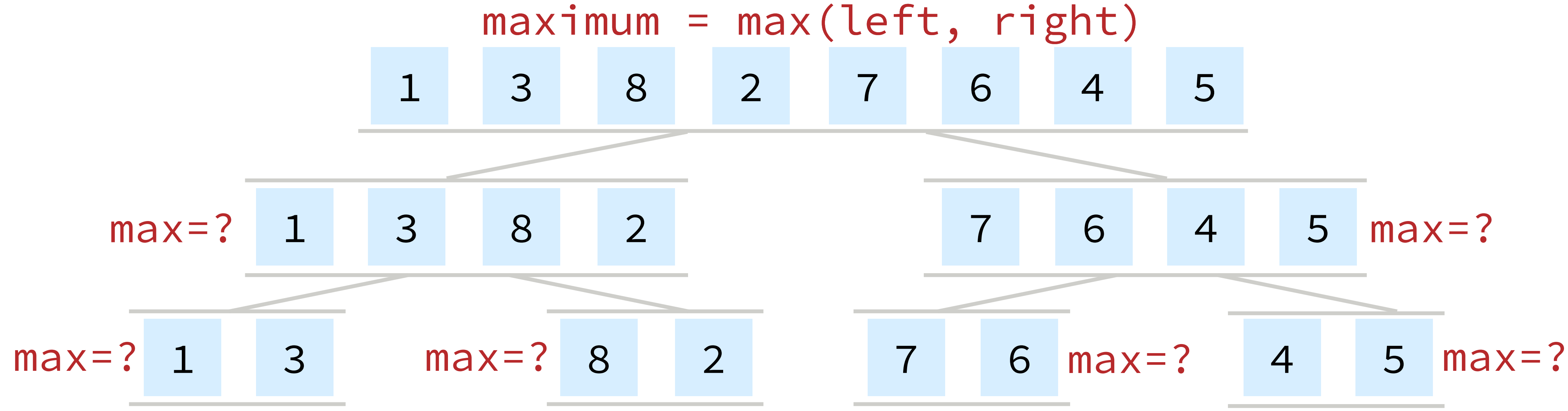
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



```
maximum = max(left, right)
```

1 3 8 2 7 6 4 5

max=? 1 3 8 2

7 6 4 5 max=?

max=? 1 3

max=? 8 2

7 6 max=?

4 5 max=?

You. Now we have 4 problems instead of one!

Wise man. Do the same!

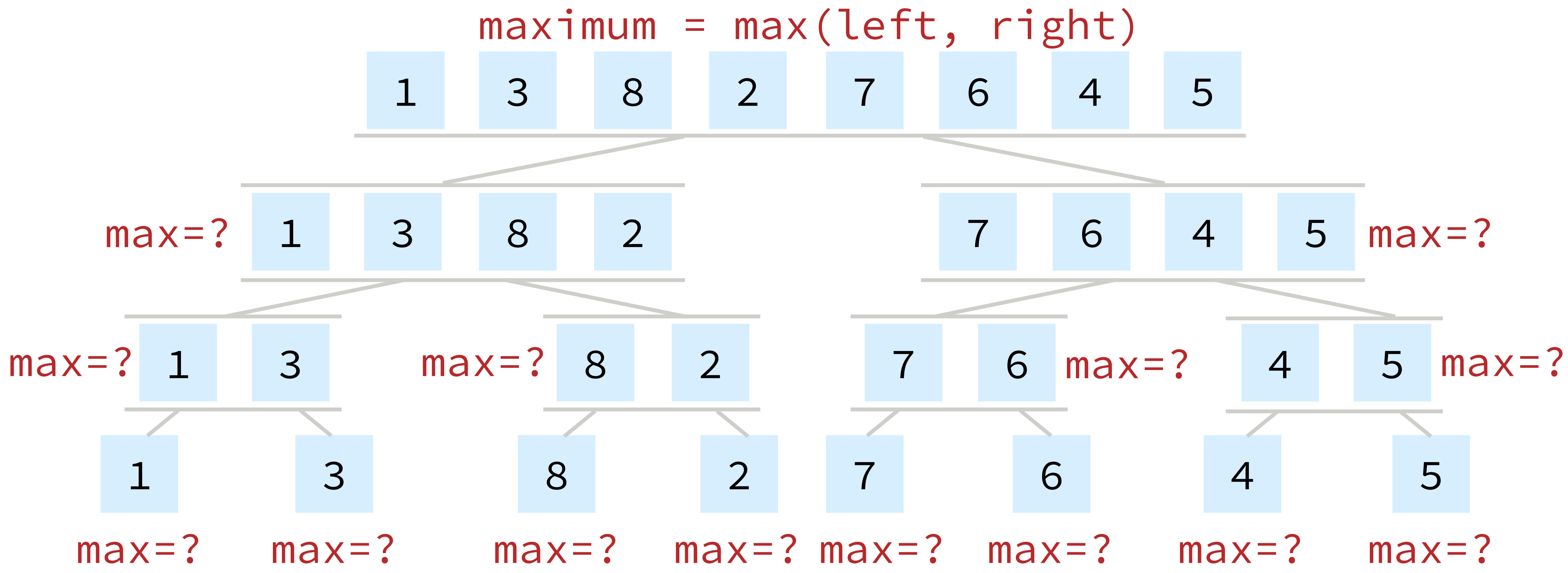
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



You. Now we have 8 problems instead of one!

Wise man. You are a lazy 21st century spoiled kid.

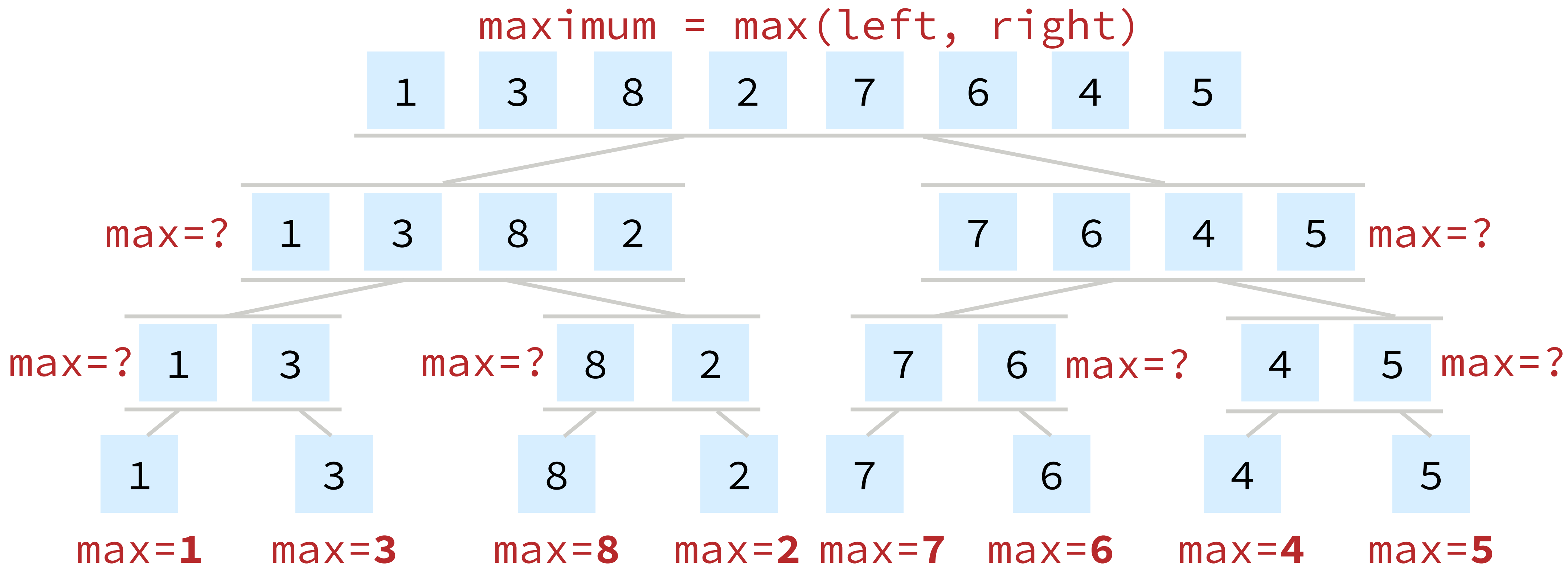
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



maximum = max(left, right)

1 3 8 2 7 6 4 5

max=? 1 3 8 2

7 6 4 5 max=?

max=? 1 3

max=? 8 2

7 6 max=?

4 5 max=?

1 3 8 2 7 6 4 5

max=**1** max=**3** max=**8** max=**2** max=**7** max=**6** max=**4** max=**5**

You. Oops! I know what the maximum of an array of size 1 is!

Wise man. ...

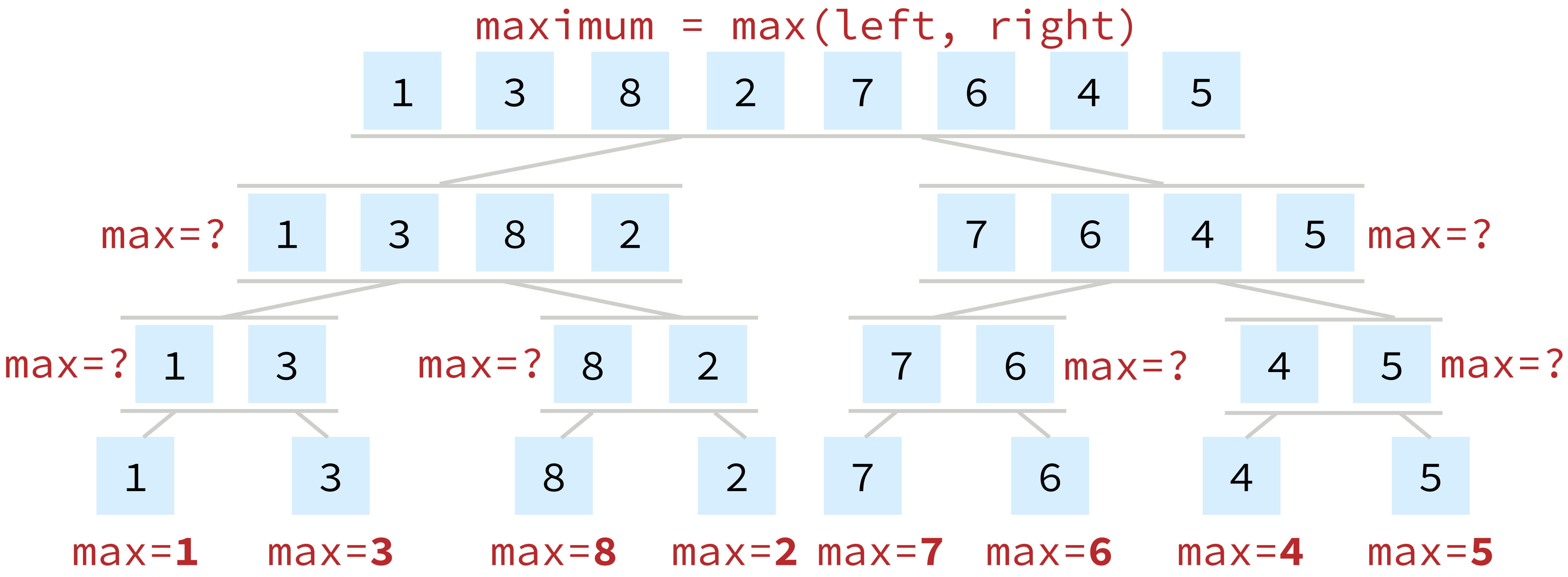
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



`maximum = max(left, right)`

1 3 8 2 7 6 4 5

max=? 1 3 8 2

7 6 4 5 max=?

max=? 1 3

max=? 8 2

7 6 max=?

4 5 max=?

1 3 8 2 7 6 4 5

max=**1** max=**3** max=**8** max=**2** max=**7** max=**6** max=**4** max=**5**

You. … I think I got it!

Wise man. …

Finding the Max

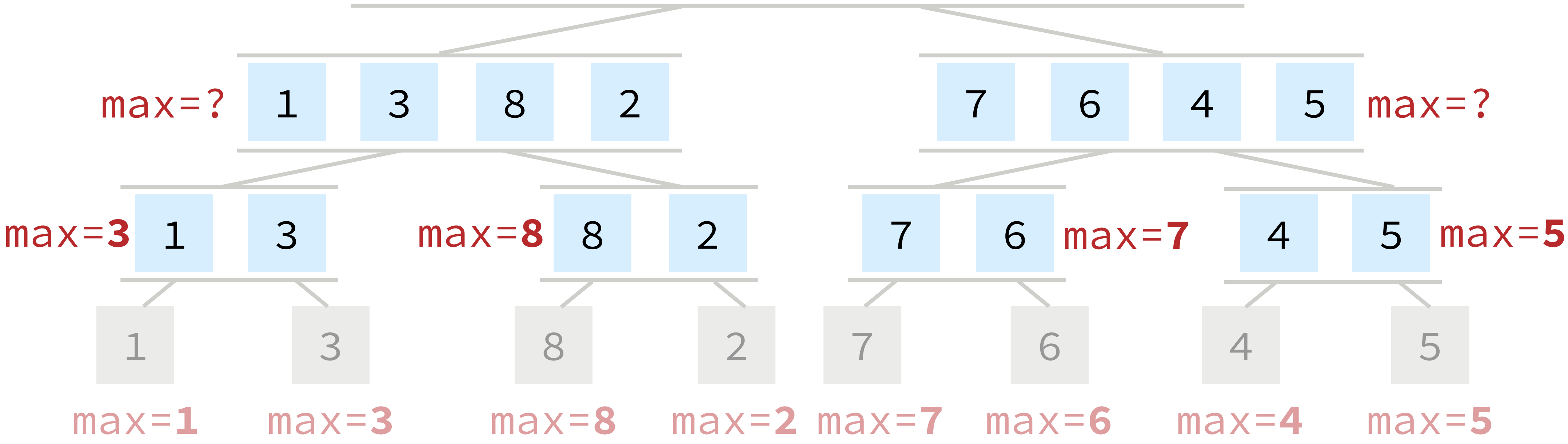
Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!

`maximum = max(left, right)`



1 3 8 2 7 6 4 5

max=? 1 3 8 2   7 6 4 5 max=?

max=3 1 3   max=8 8 2   7 6 max=7   4 5 max=5

1 3 8 2 7 6 4 5

max=1 max=3 max=8 max=2 max=7 max=6 max=4 max=5

You. … I think I got it!

Wise man. …

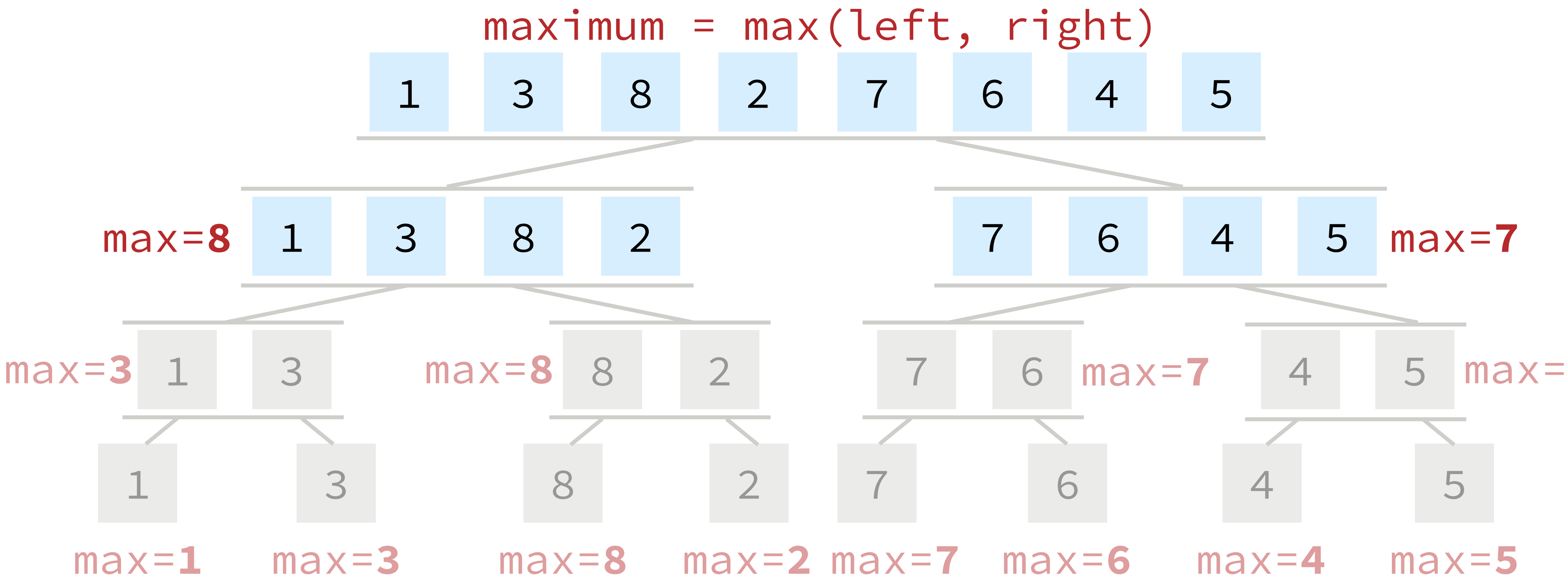
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



You. … I think I got it!

Wise man. …

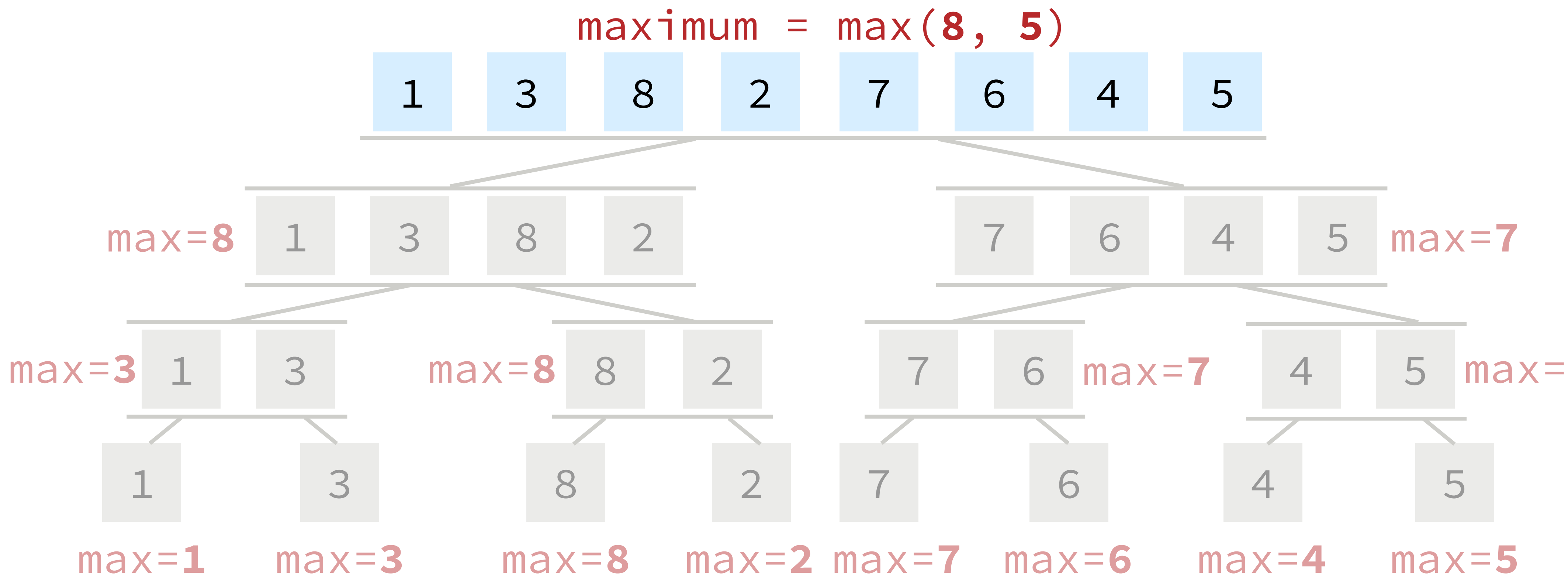
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



maximum = max(8, 5)

1 3 8 2 7 6 4 5

max=8 1 3 8 2

7 6 4 5 max=7

max=3 1 3

max=8 8 2

7 6 max=7

4 5 max=5

1 3 8 2 7 6 4 5

max=1 max=3 max=8 max=2 max=7 max=6 max=4 max=5

You. ... I think I got it!

Wise man. ...

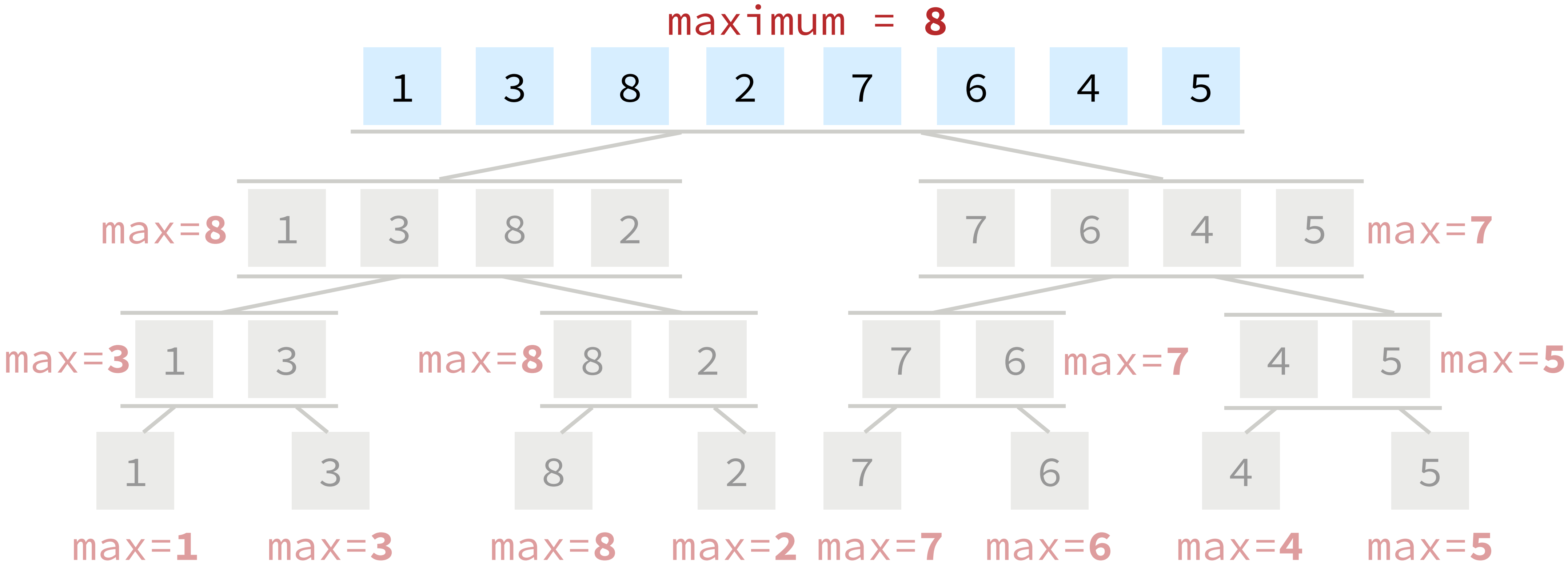
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



You. …

Wise man. The max is **8**

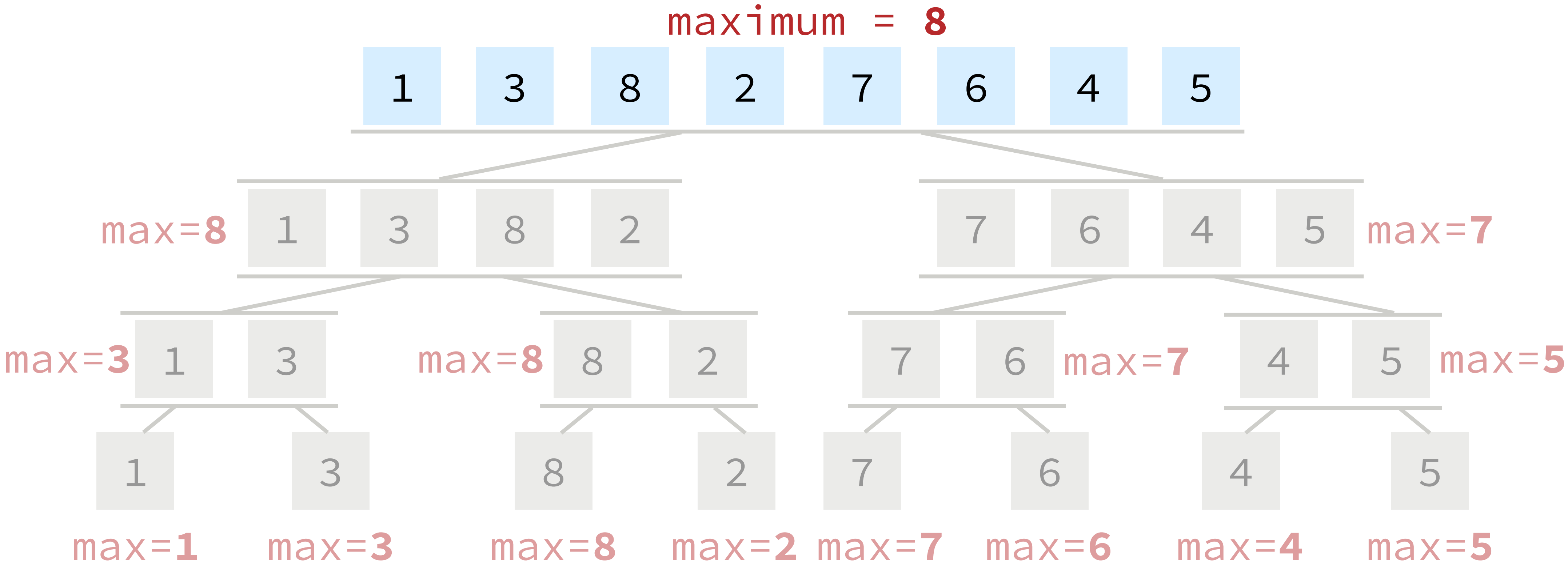
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



You. But this requires a lot of comparisons.

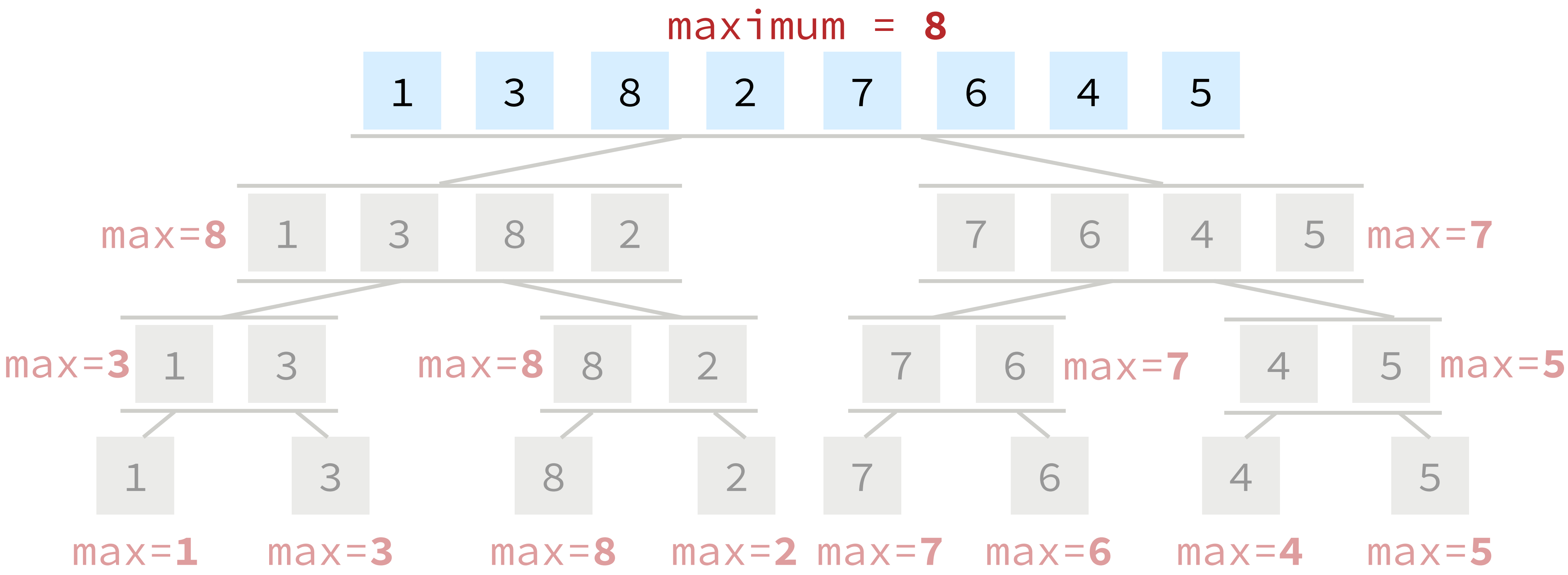
Finding the Max

Problem. Given an array of n elements, find the maximum element.

Solution 1. Assume the first element is the max, compare the current max to each element in the array and update it if a larger element is found.

Number of compares = $n - 1$.

Solution 2 (by the old wise man). If you give me the max of the left half and the max of the right half, I can tell you the max of the whole array!



maximum = 8

1 3 8 2 7 6 4 5

max=8 1 3 8 2

7 6 4 5 max=7

max=3 1 3

max=8 8 2

7 6 max=7

4 5 max=5

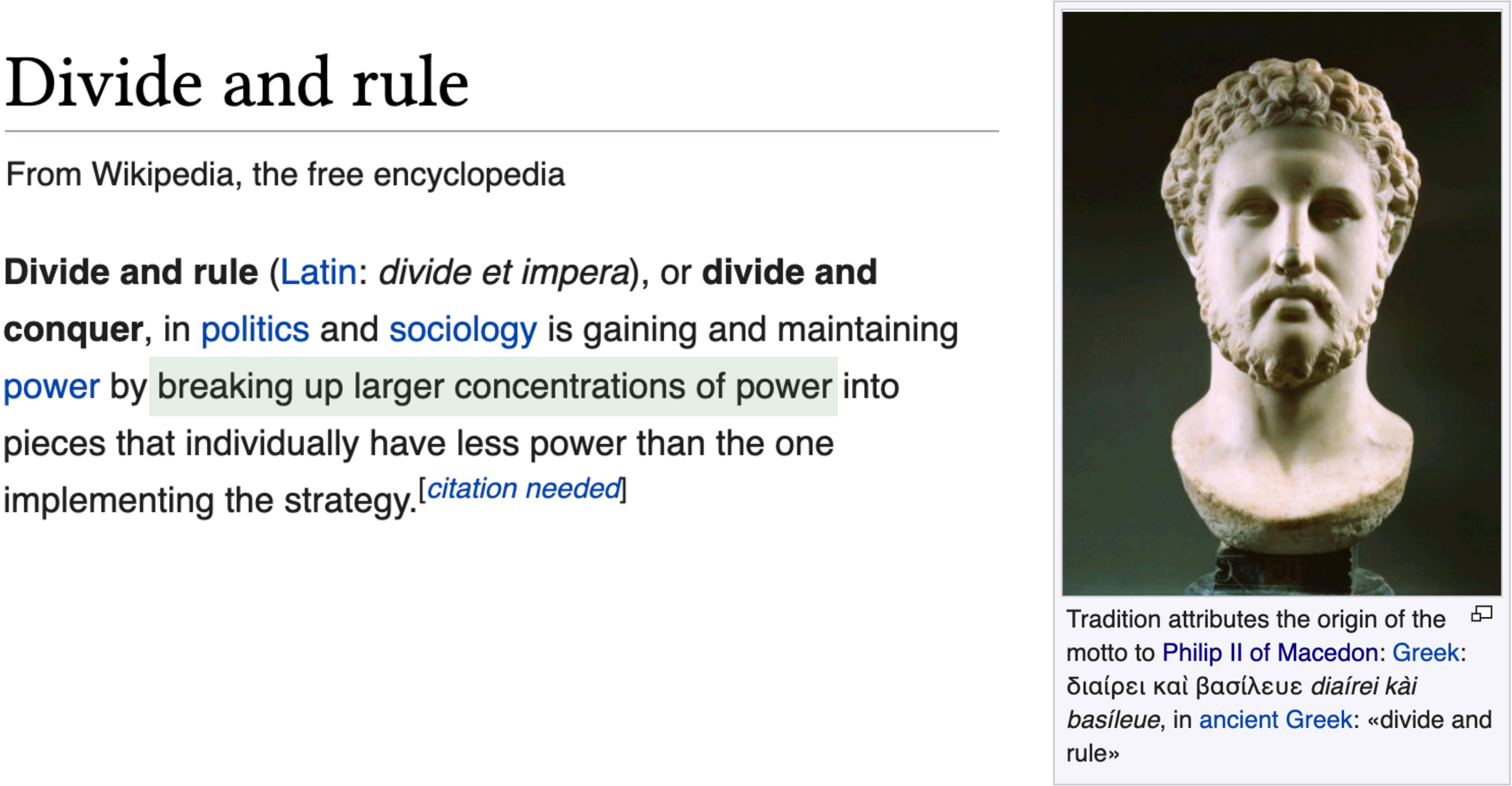
1 3 8 2 7 6 4 5

max=1 max=3 max=8 max=2 max=7 max=6 max=4 max=5

You. But this requires a lot of comparisons.

Wise man. This requires $\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\ldots+\frac{n}{n} \;\leq\; n(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{n}) \;\leq\; n$ compares!

Divide & Conquer



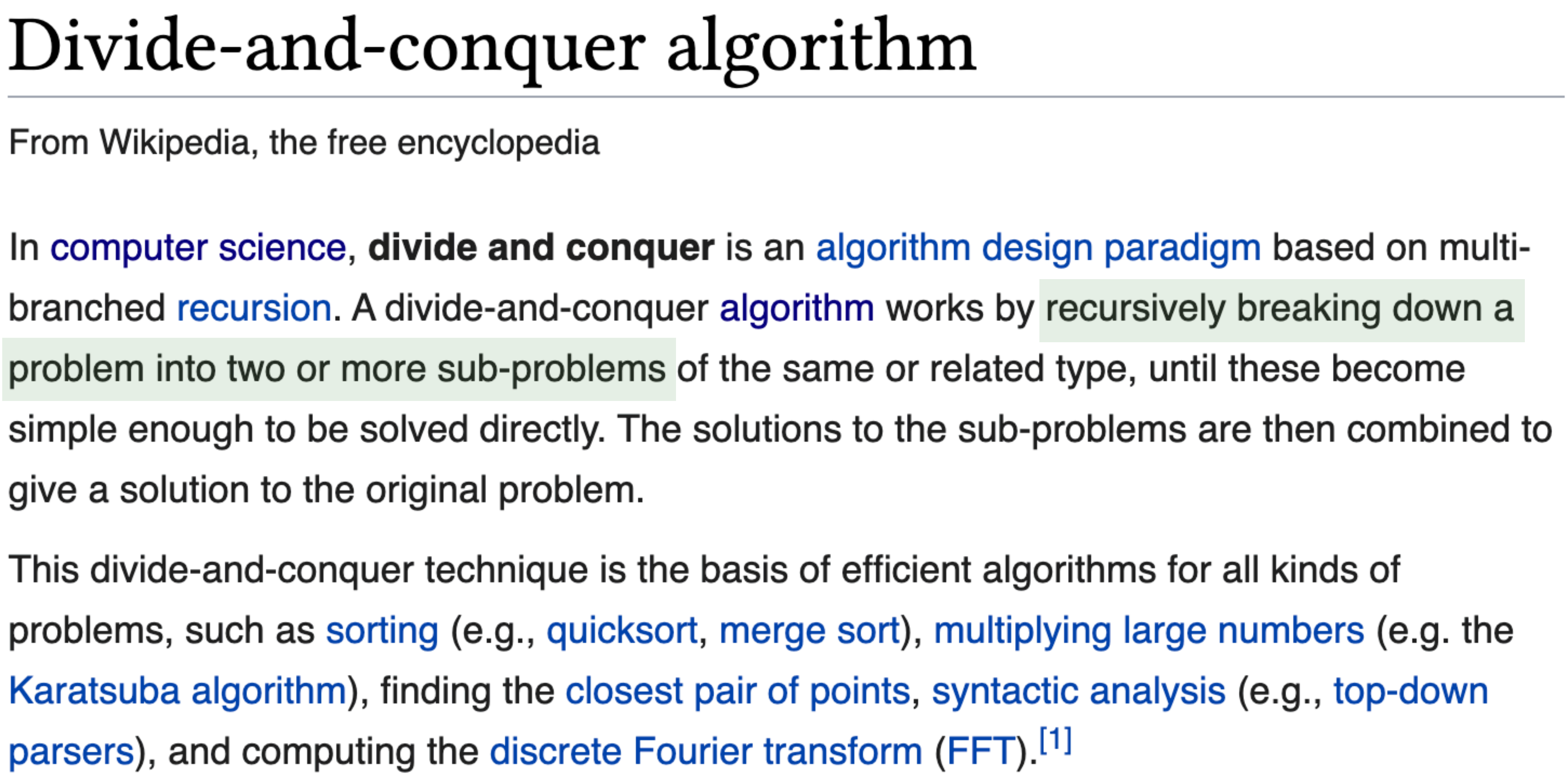
Divide and rule

From Wikipedia, the free encyclopedia

Divide and rule (Latin: *divide et impera*), or **divide and conquer**, in politics and sociology is gaining and maintaining power by breaking up larger concentrations of power into pieces that individually have less power than the one implementing the strategy.[*citation needed*]

Tradition attributes the origin of the motto to Philip II of Macedon: Greek: διαίρει καὶ βασίλευε *diaírei kài basíleue*, in ancient Greek: «divide and rule»

Divide & Conquer



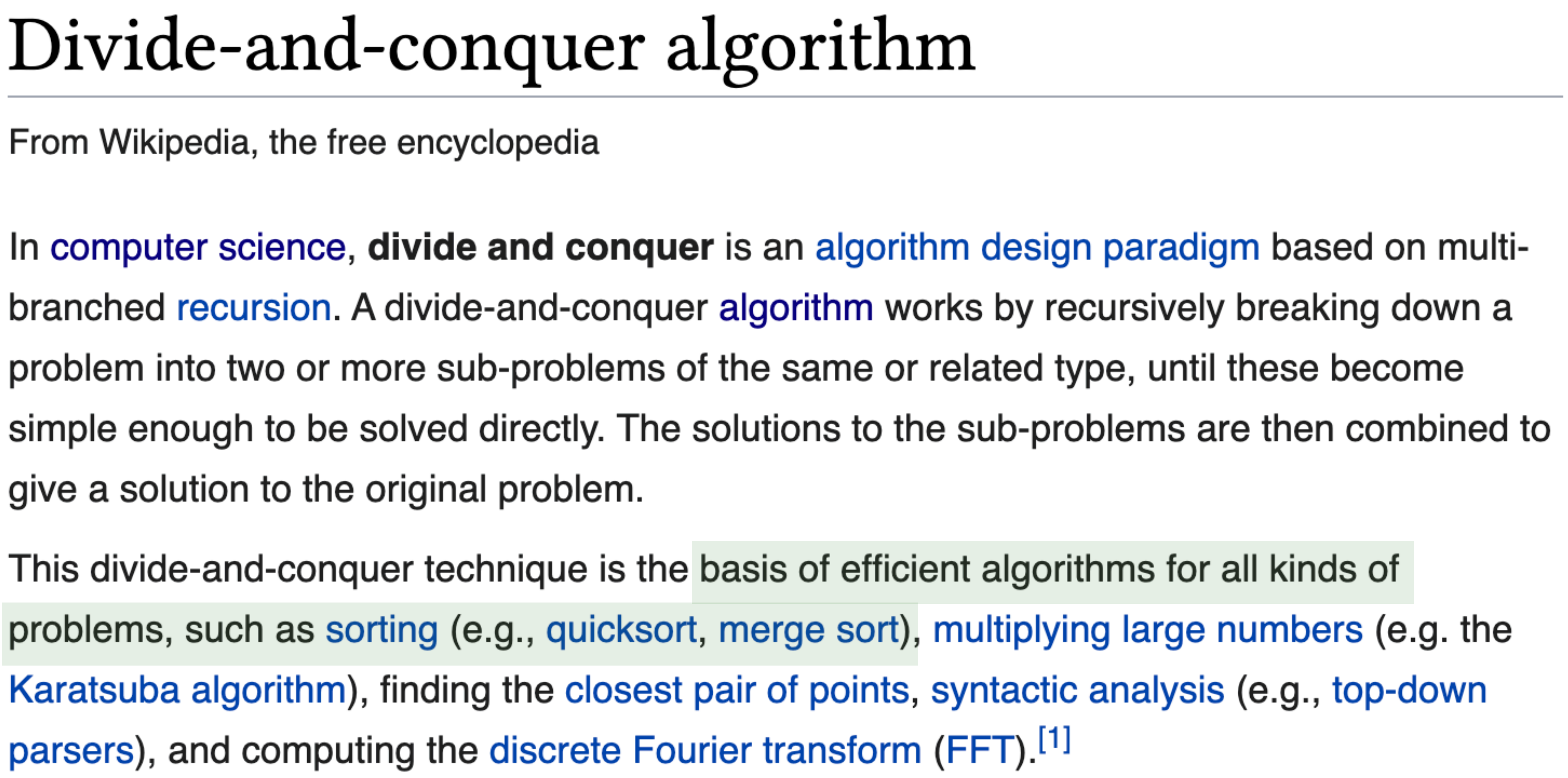
Divide-and-conquer algorithm

From Wikipedia, the free encyclopedia

In computer science, **divide and conquer** is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide-and-conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).[1]

Divide & Conquer



Divide-and-conquer algorithm

From Wikipedia, the free encyclopedia

In computer science, **divide and conquer** is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide-and-conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).[1]

Merge Sort

Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

Merge Sort

Basic Plan:

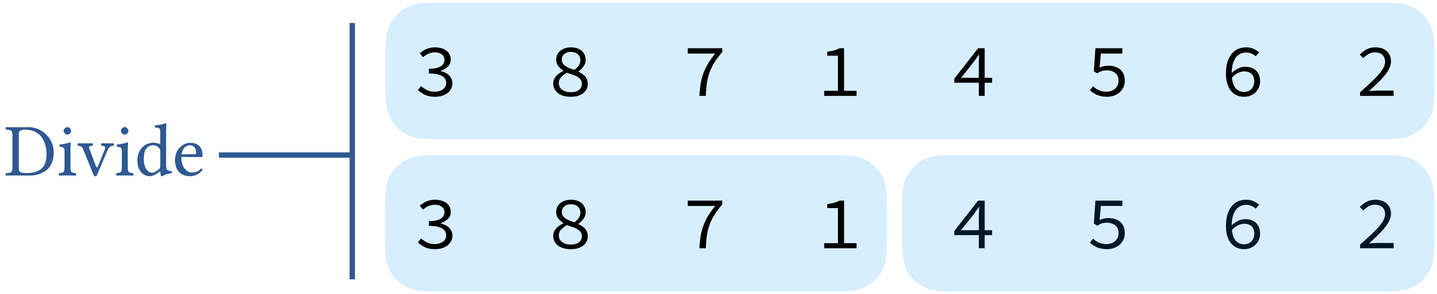
- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

3	8	7	1	4	5	6	2

Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Divide

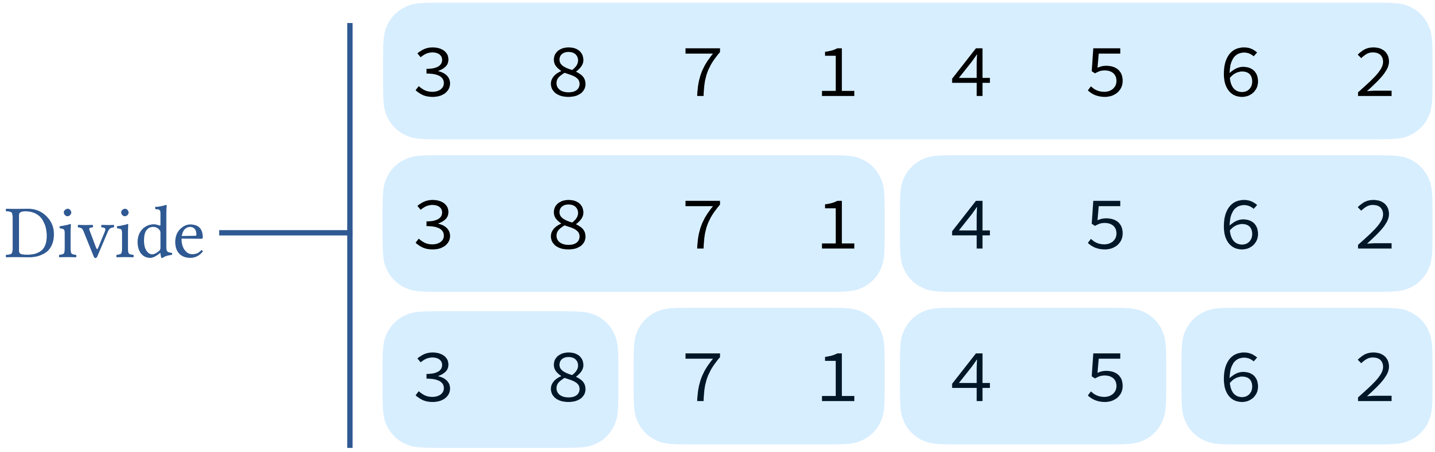
3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

Merge Sort

Basic Plan:

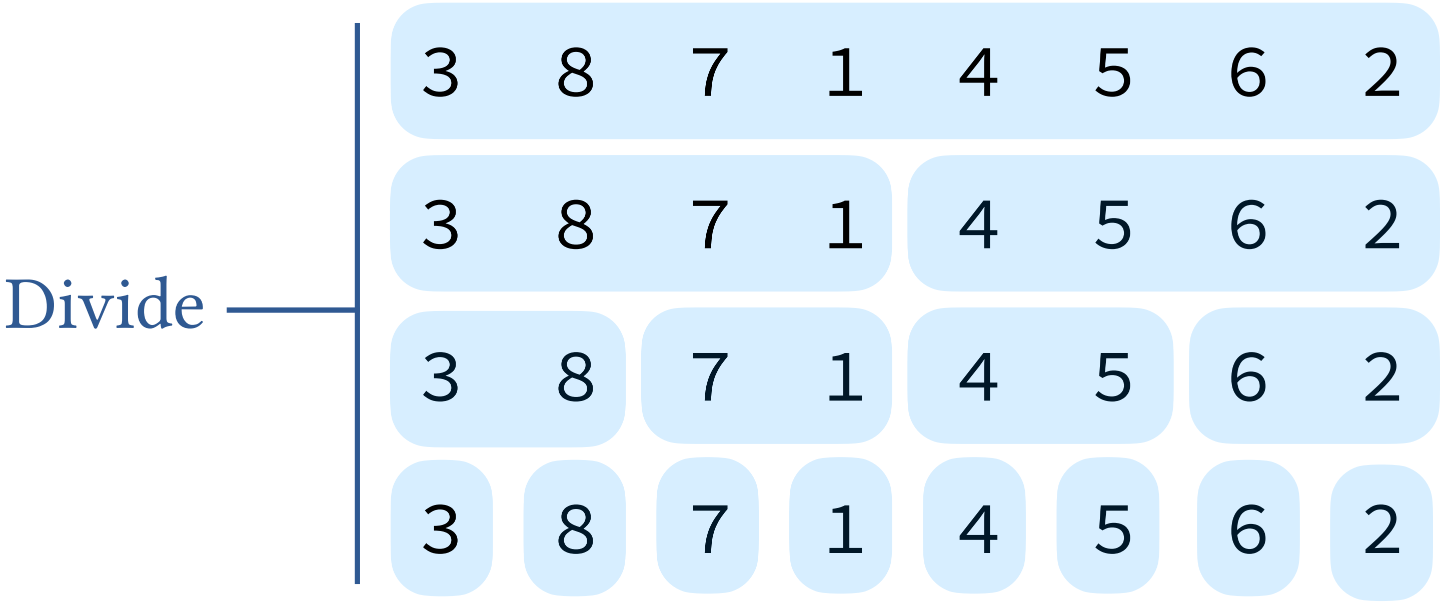
- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Divide

3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

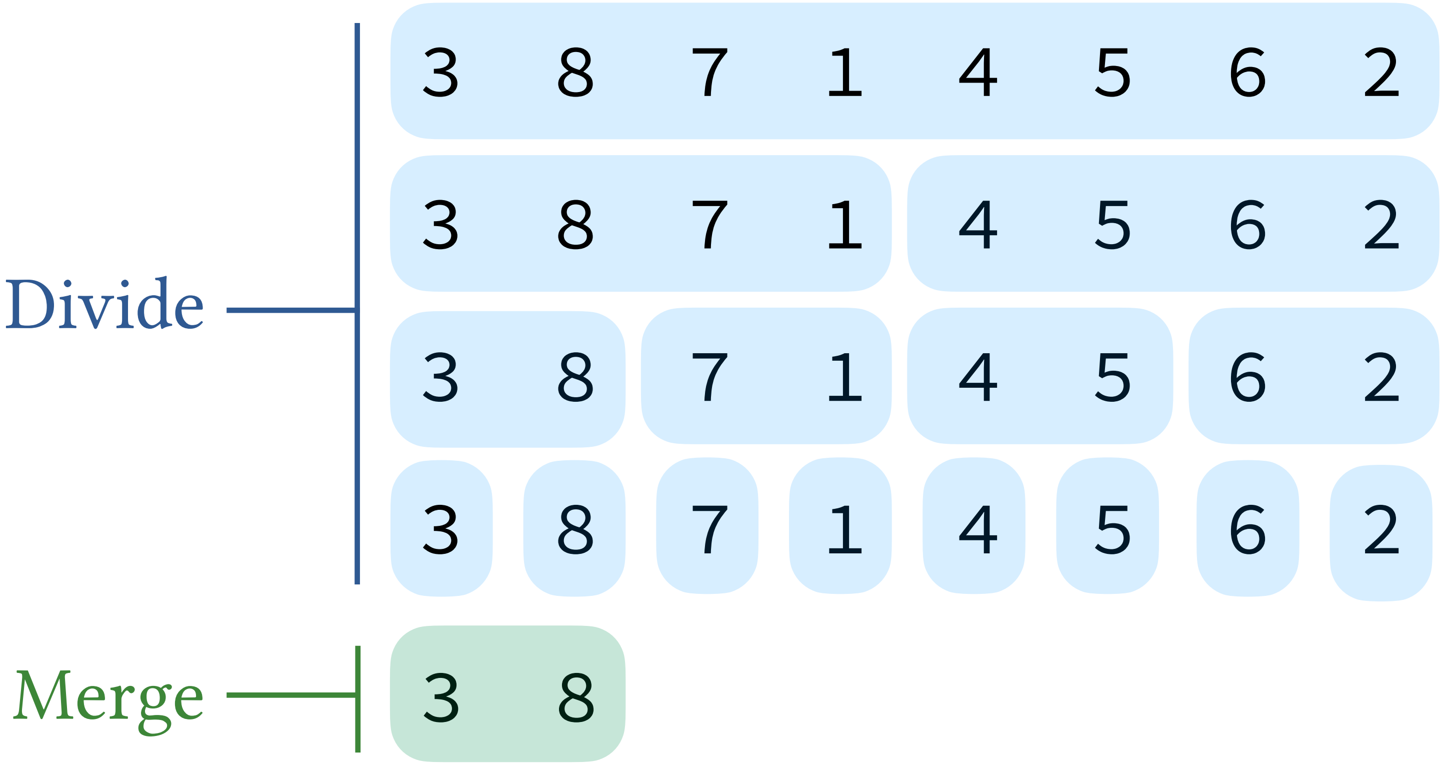
3 8 | 7 1 | 4 5 | 6 2

3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Divide

3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

3 8 | 7 1 | 4 5 | 6 2

3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

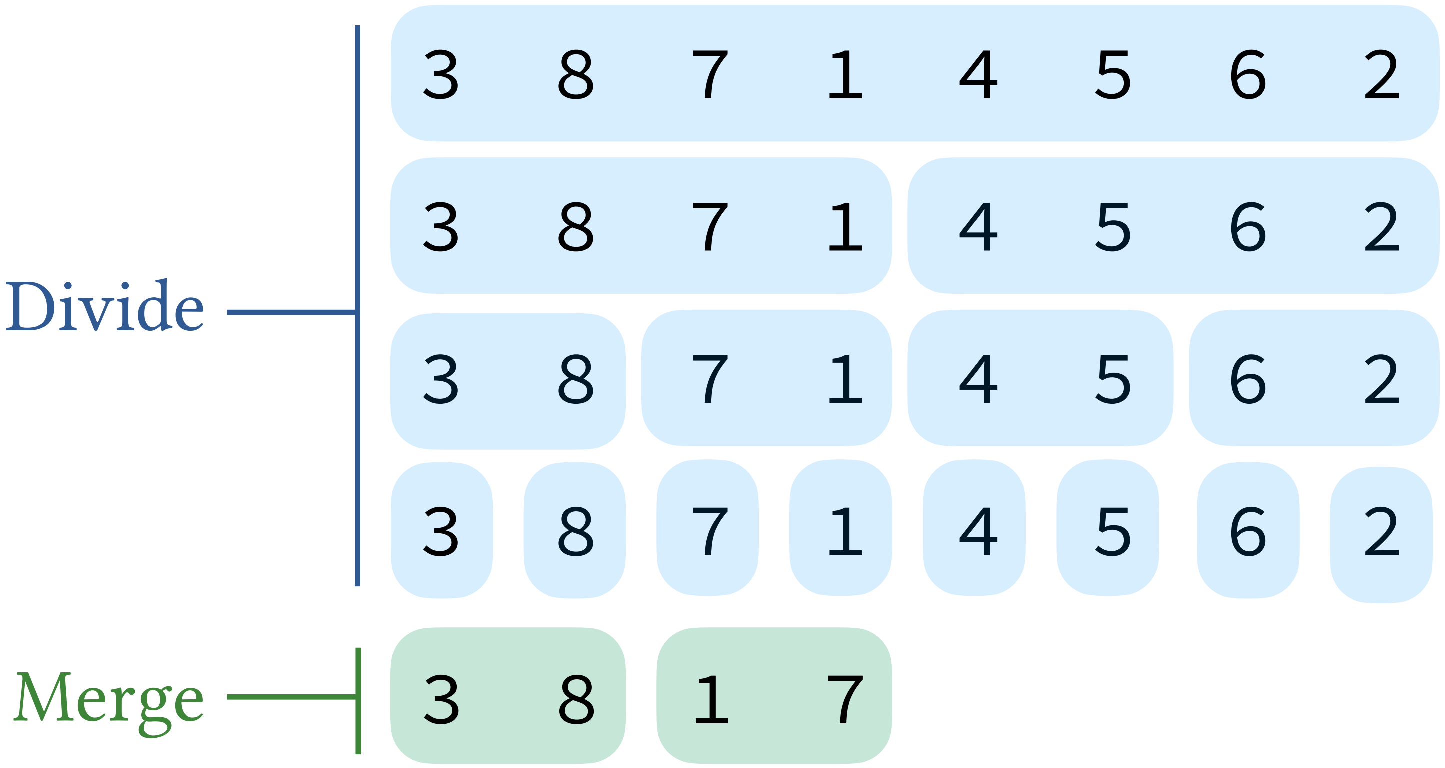
Merge

3 8

Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

3 8 | 7 1 | 4 5 | 6 2

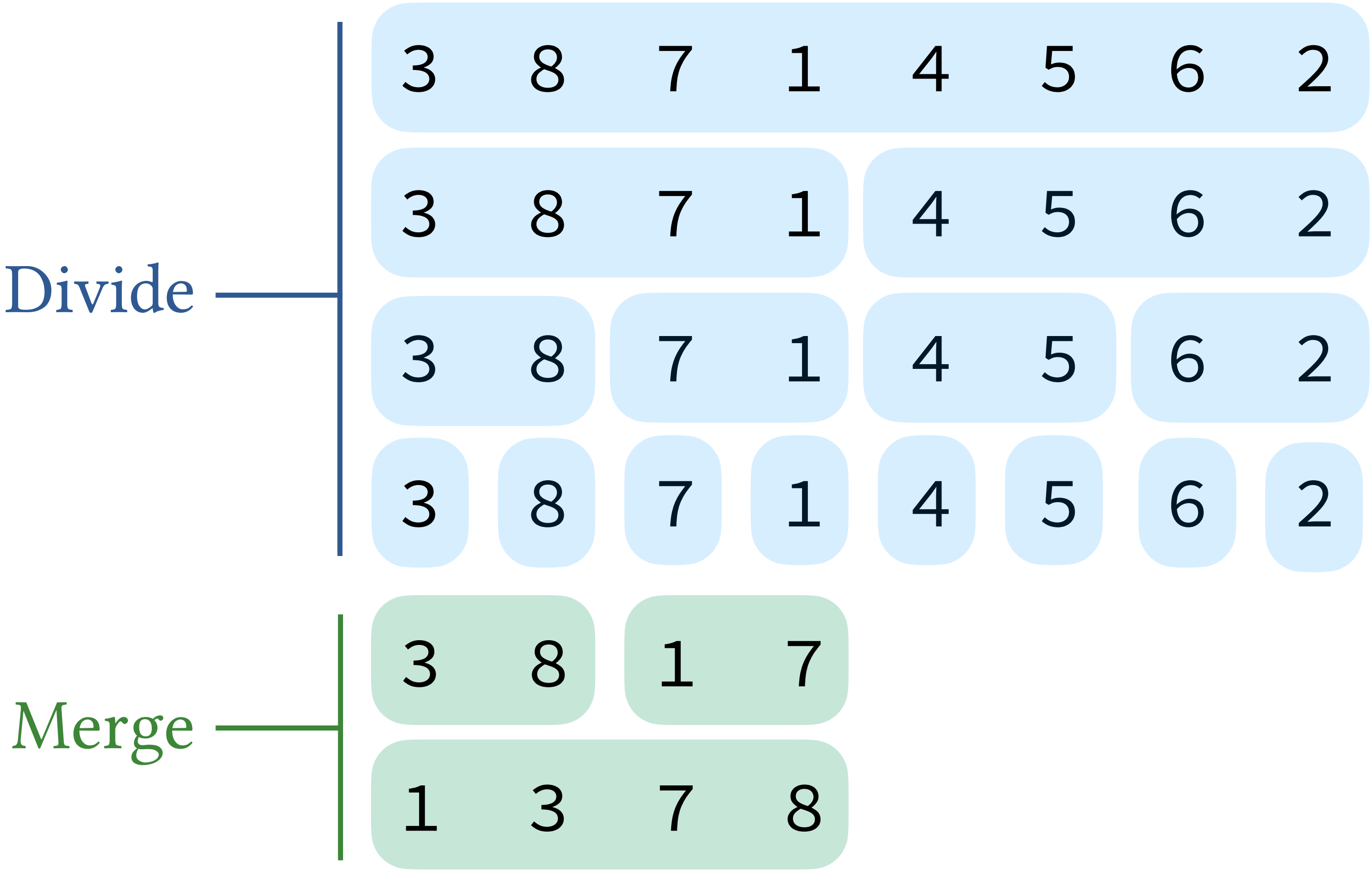
3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

3 8 | 1 7

Merge Sort

Basic Plan:

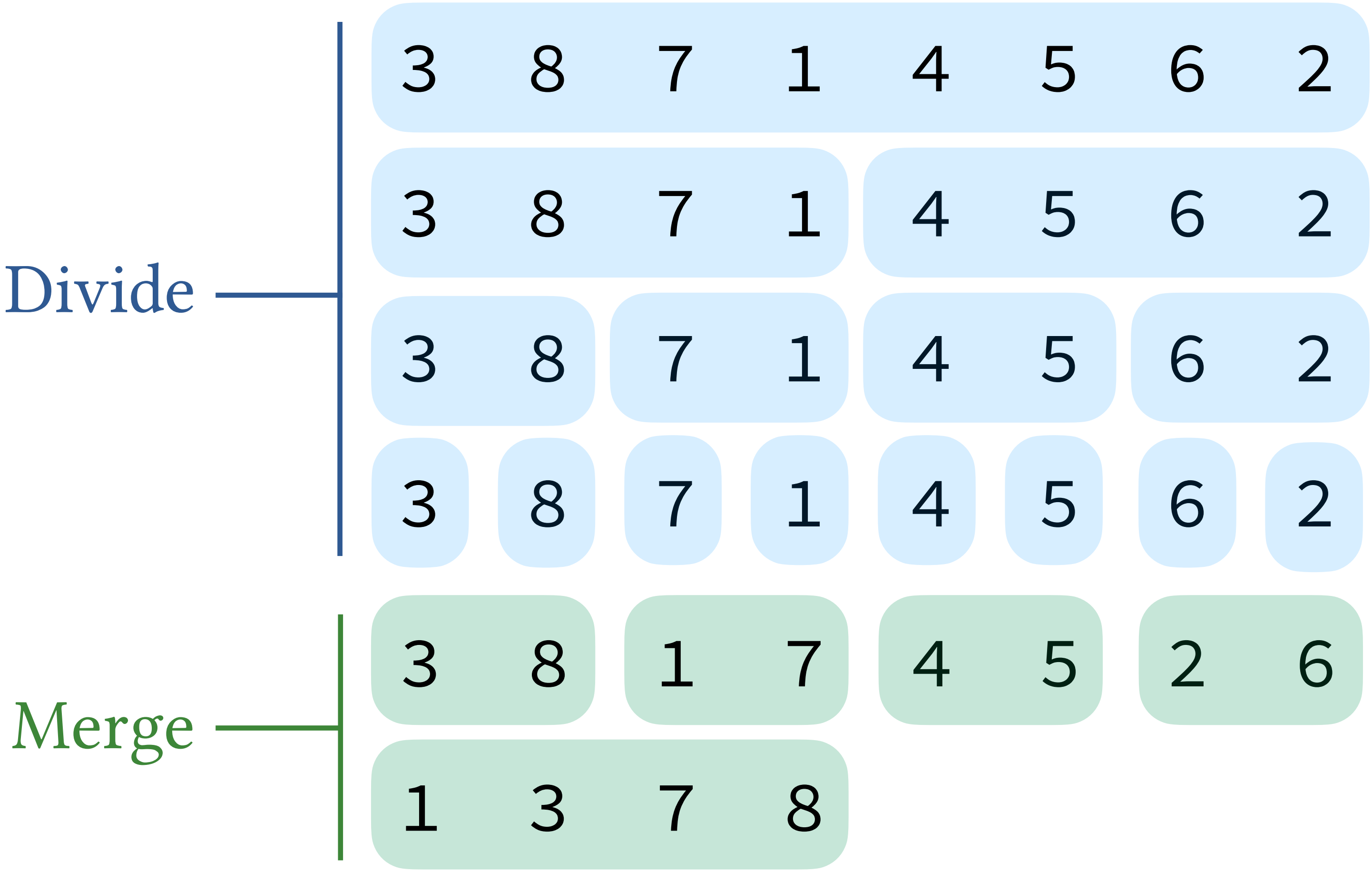
- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Divide

3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

3 8 | 7 1 | 4 5 | 6 2

3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

Merge

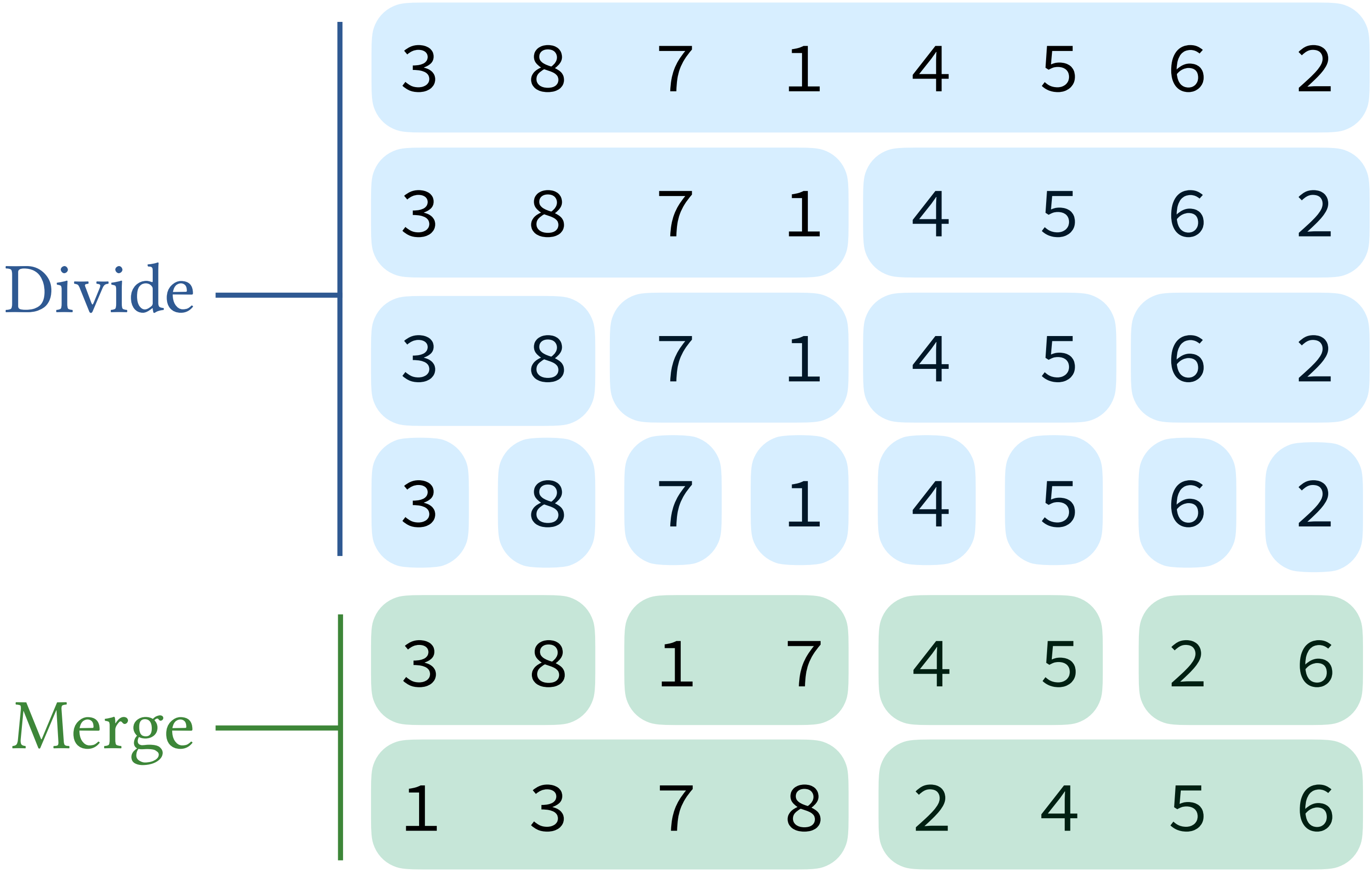
3 8 | 1 7 | 4 5 | 2 6

1 3 7 8

Merge Sort

Basic Plan:

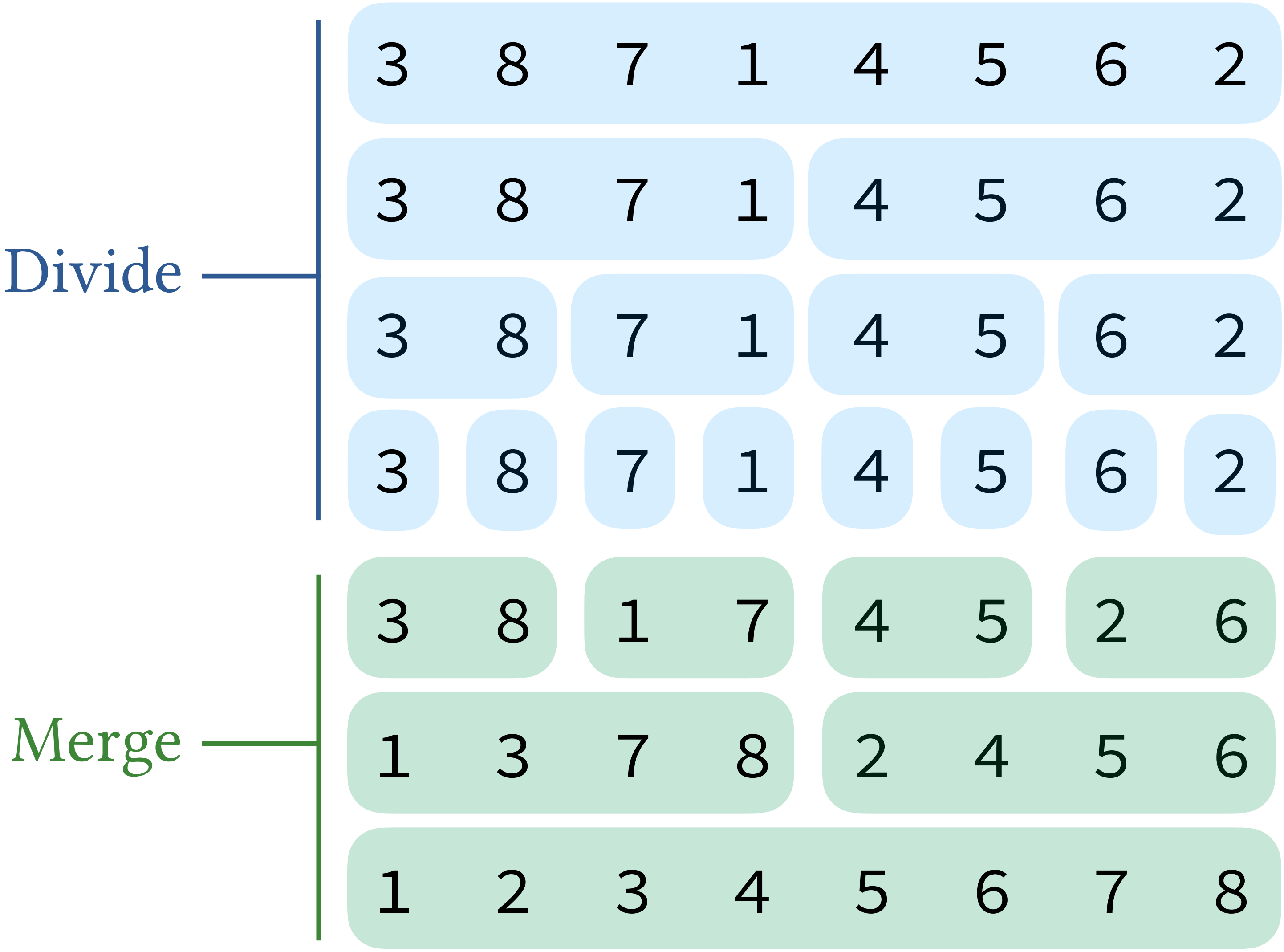
- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Merge Sort

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.



Divide

3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

3 8 | 7 1 | 4 5 | 6 2

3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

Merge

3 8 | 1 7 | 4 5 | 2 6

1 3 7 8 | 2 4 5 6

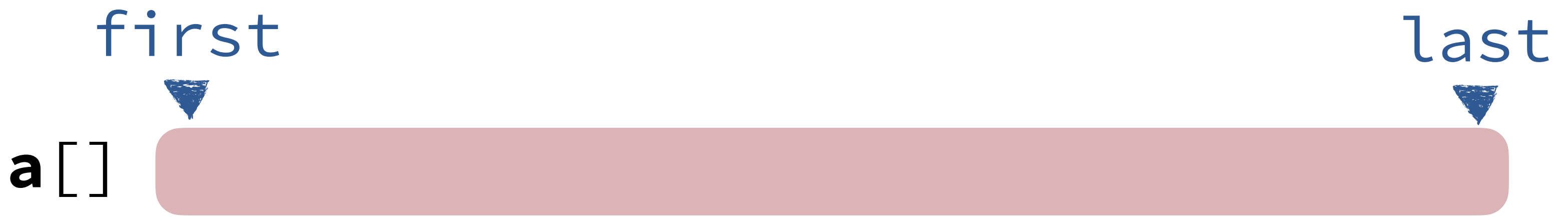
1 2 3 4 5 6 7 8

Merge Sort Algorithm

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

```
MERGE-SORT(a[], first, last)
```



Merge Sort Algorithm

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

```
MERGE-SORT(a[], first, last)

if first >= last
  return
```

if the range size <= 1
it is already sorted

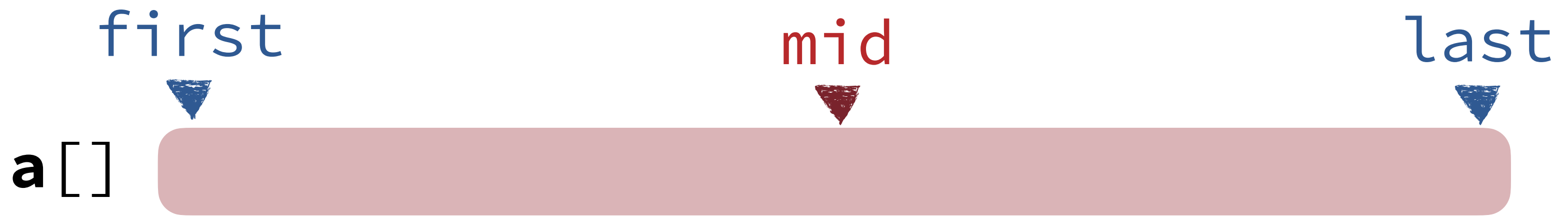
first
last
a[]

Merge Sort Algorithm

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2
```



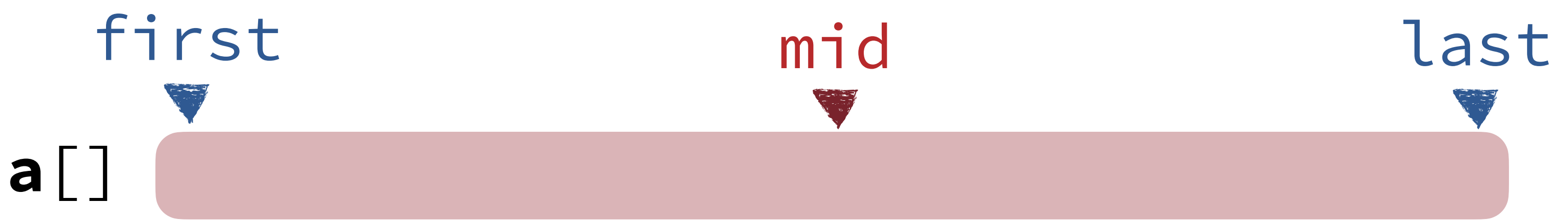
Merge Sort Algorithm

Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)
```

recursively sort the left and right halves



Merge Sort Algorithm

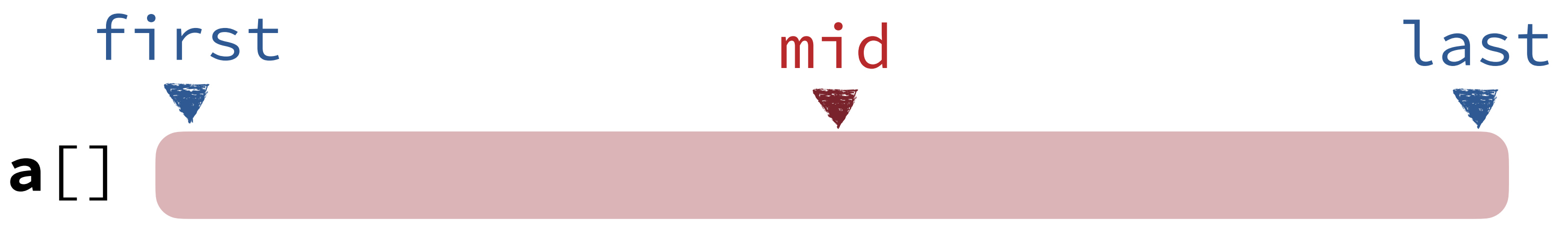
Basic Plan:

- Divide the array into two halves.
- Sort each half.
- Merge the two sorted halves.

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```

merge the sorted halves

assuming a[] *is passed by reference as in C++*



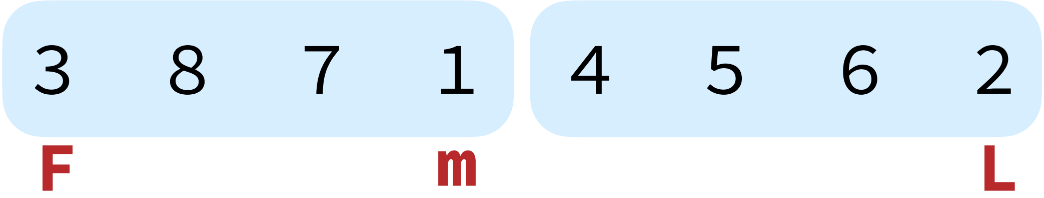
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
    return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```

3	8	7	1	4	5	6	2
F							L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```

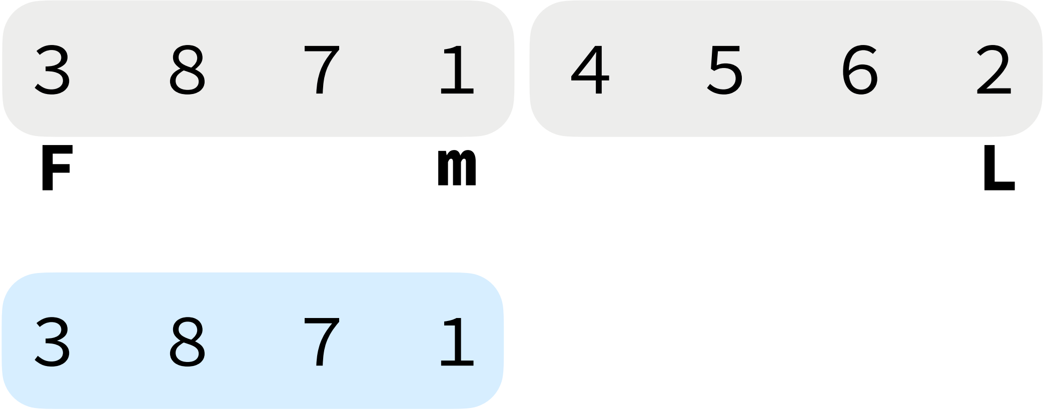


3 8 7 1 4 5 6 2

F m L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



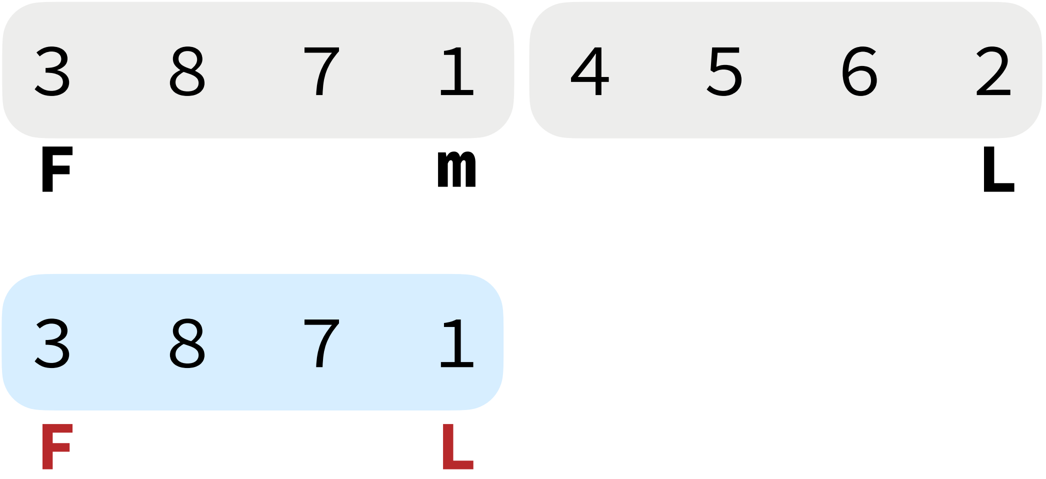
3 8 7 1 4 5 6 2

F m L

3 8 7 1

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

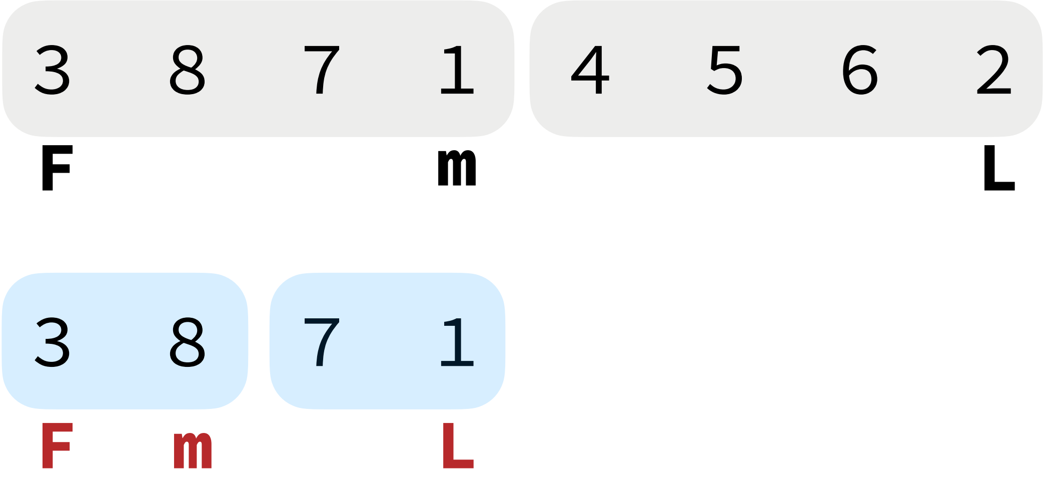
F m L

3 8 7 1

F L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
    return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 | 4 5 6 2

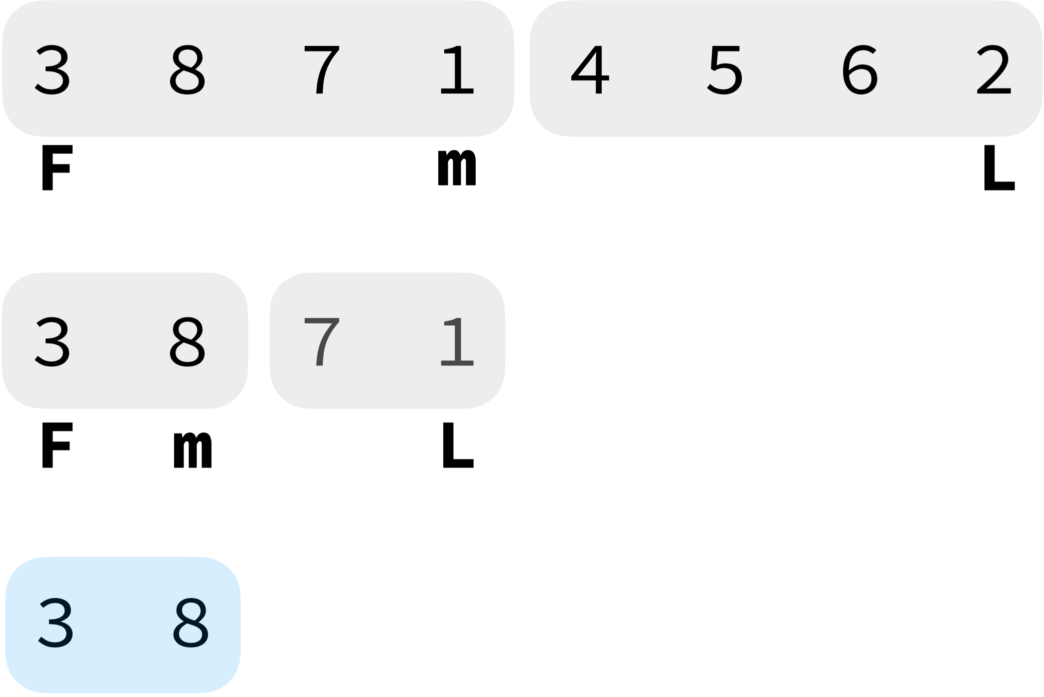
F m L

3 8 | 7 1

F m L

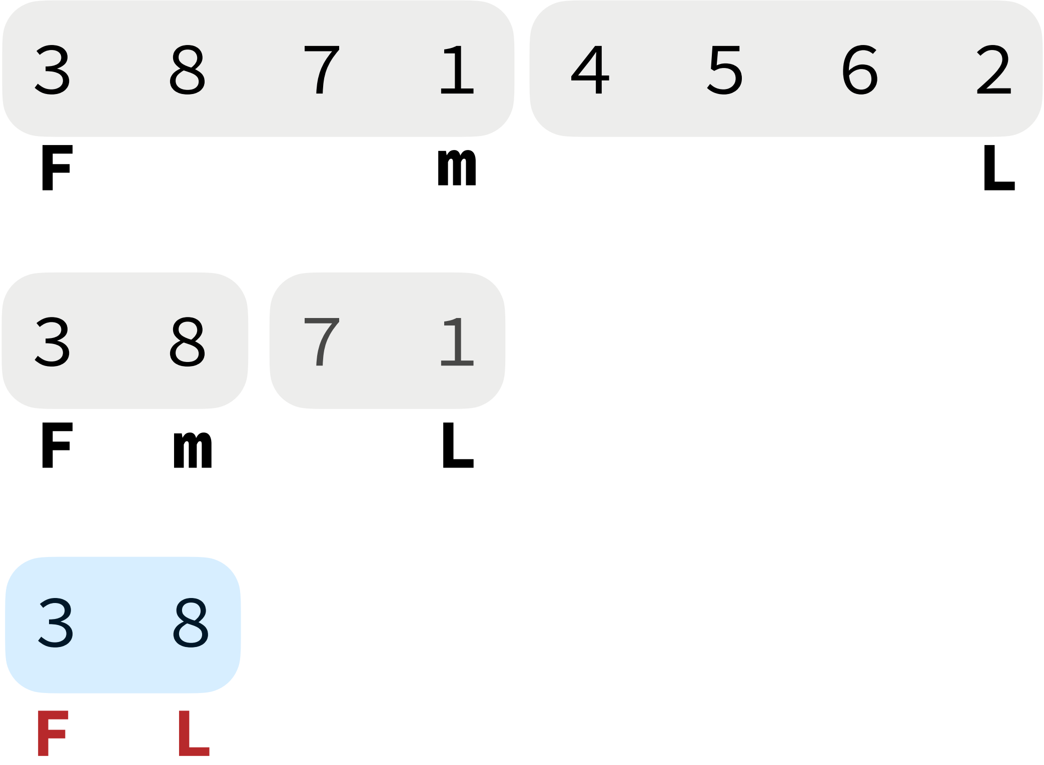
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



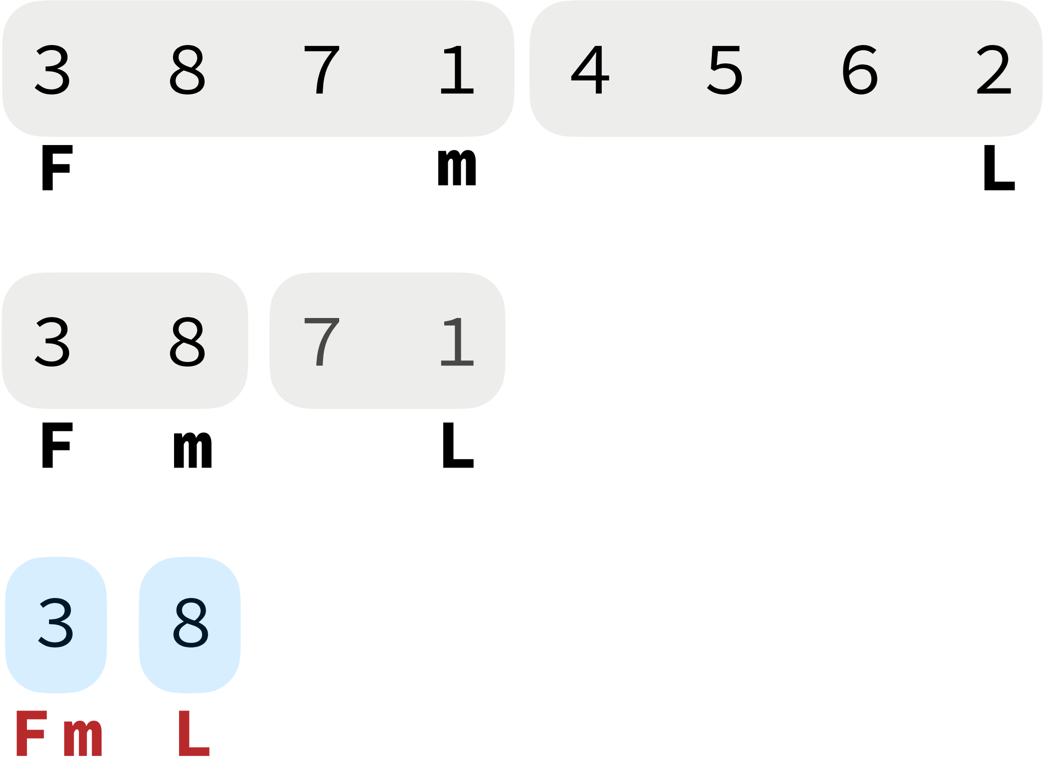
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
    return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

F m L

3 8 7 1

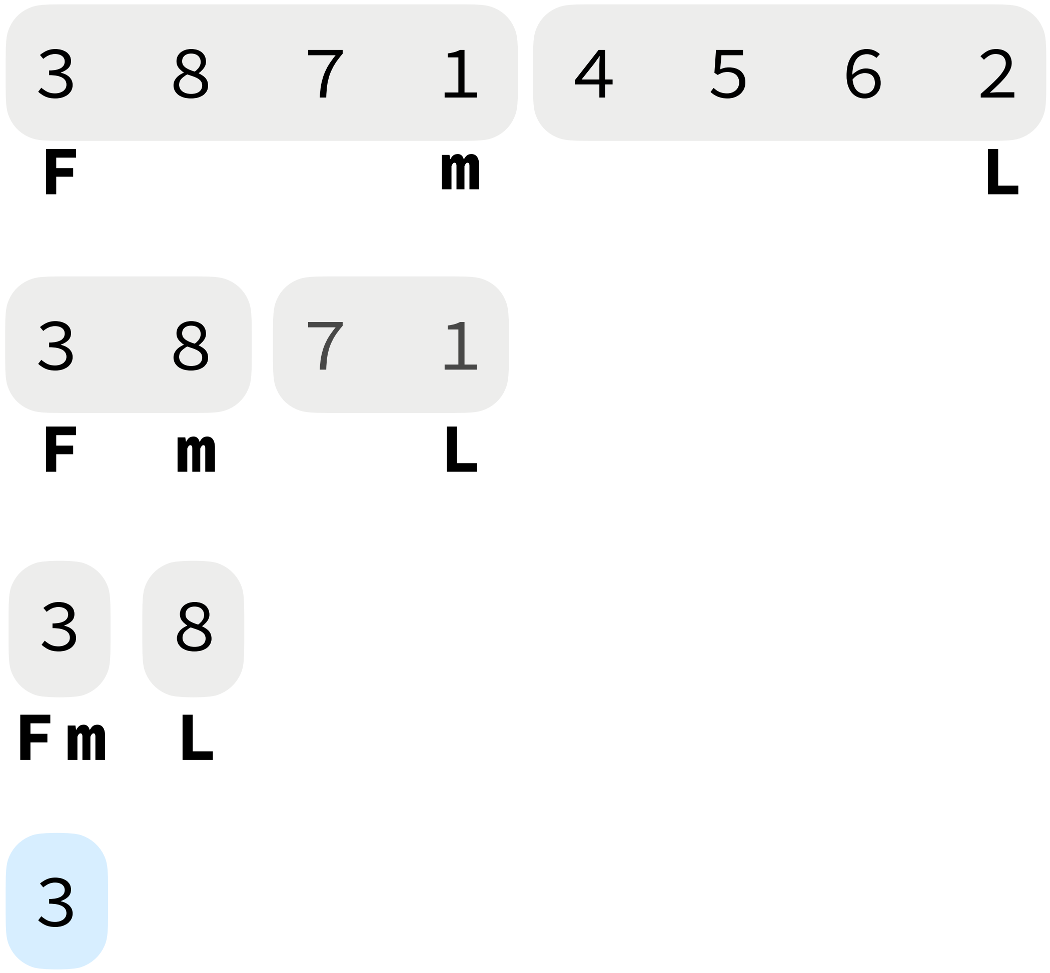
F m L

3 8

F m L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2
F m L

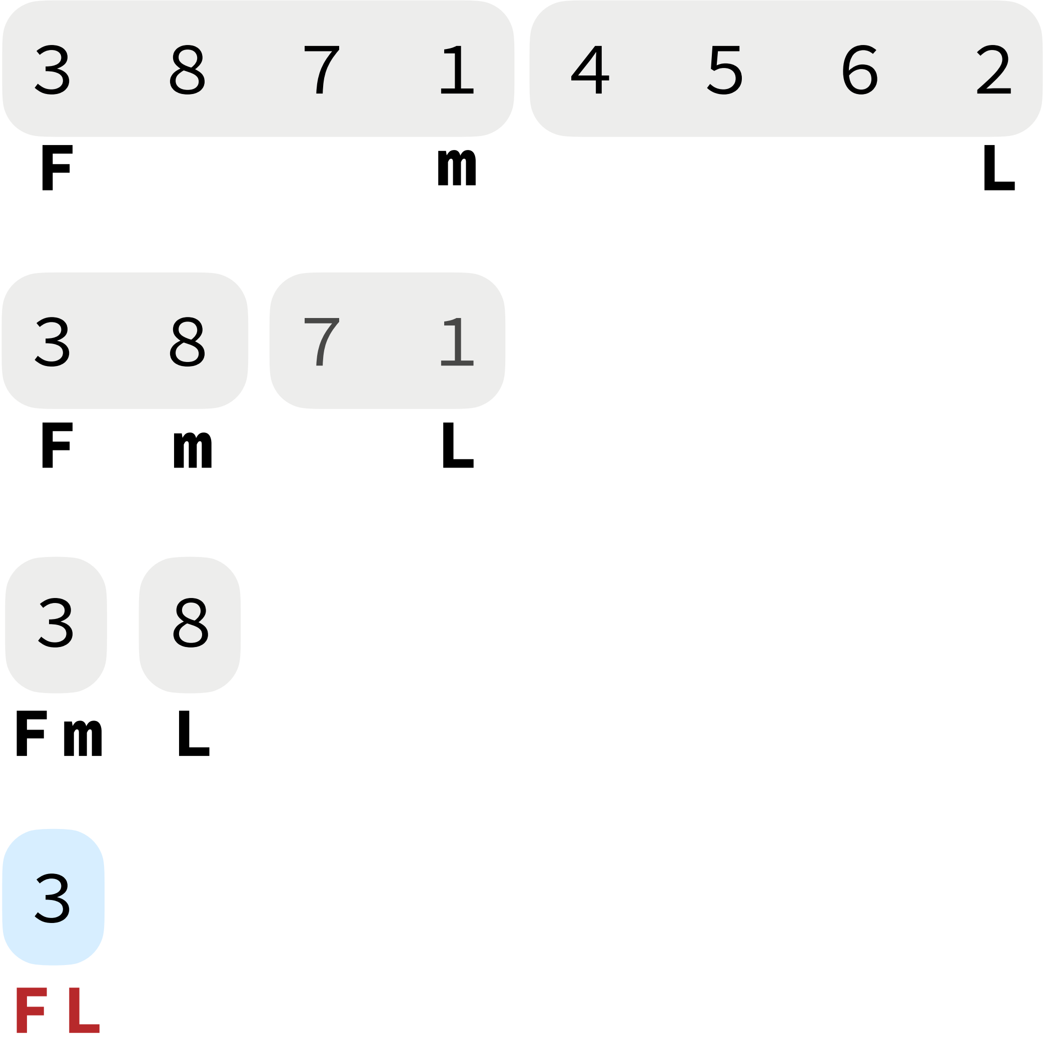
3 8 7 1
F m L

3 8
F m L

3

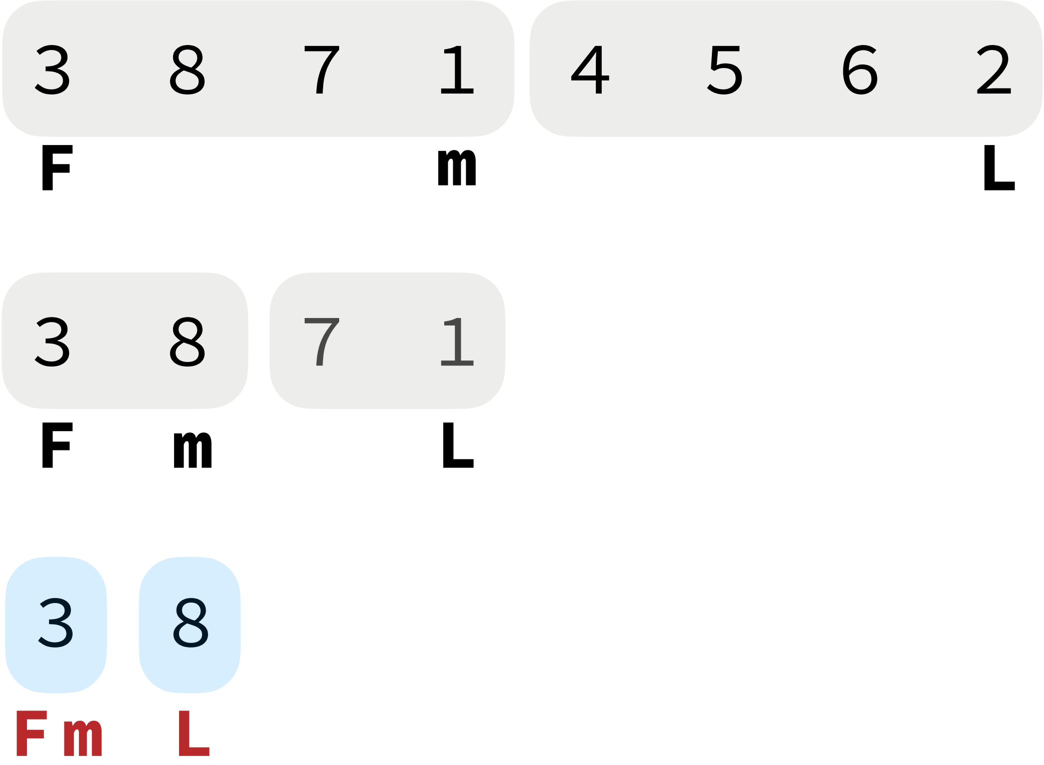
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
    return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



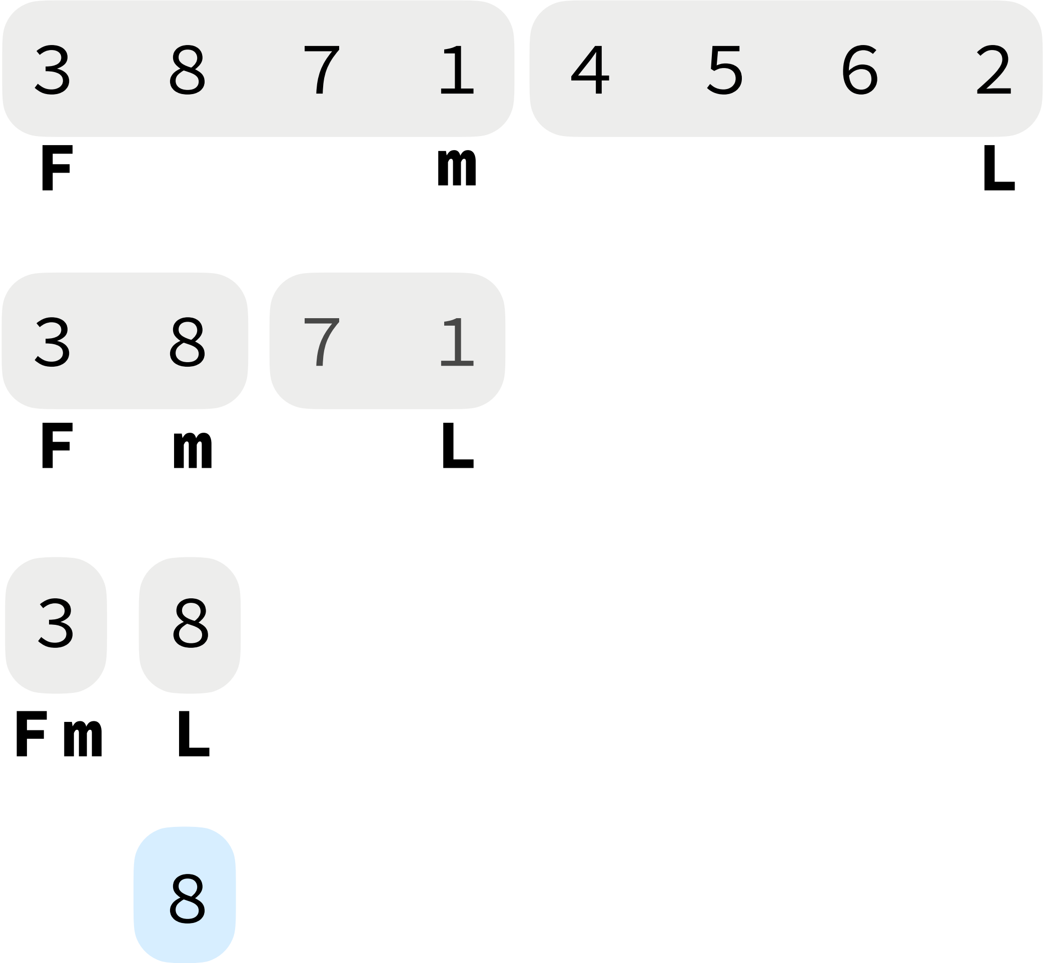
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



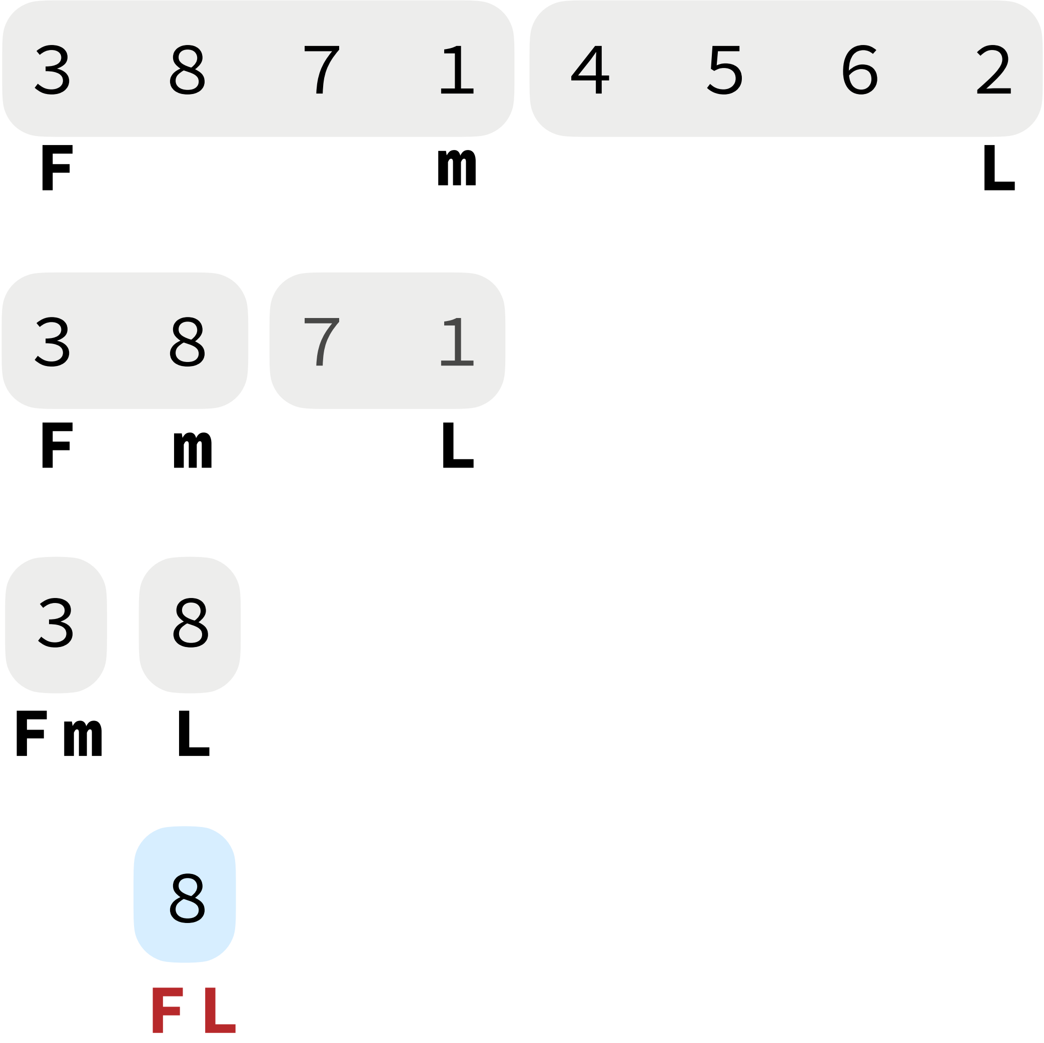
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



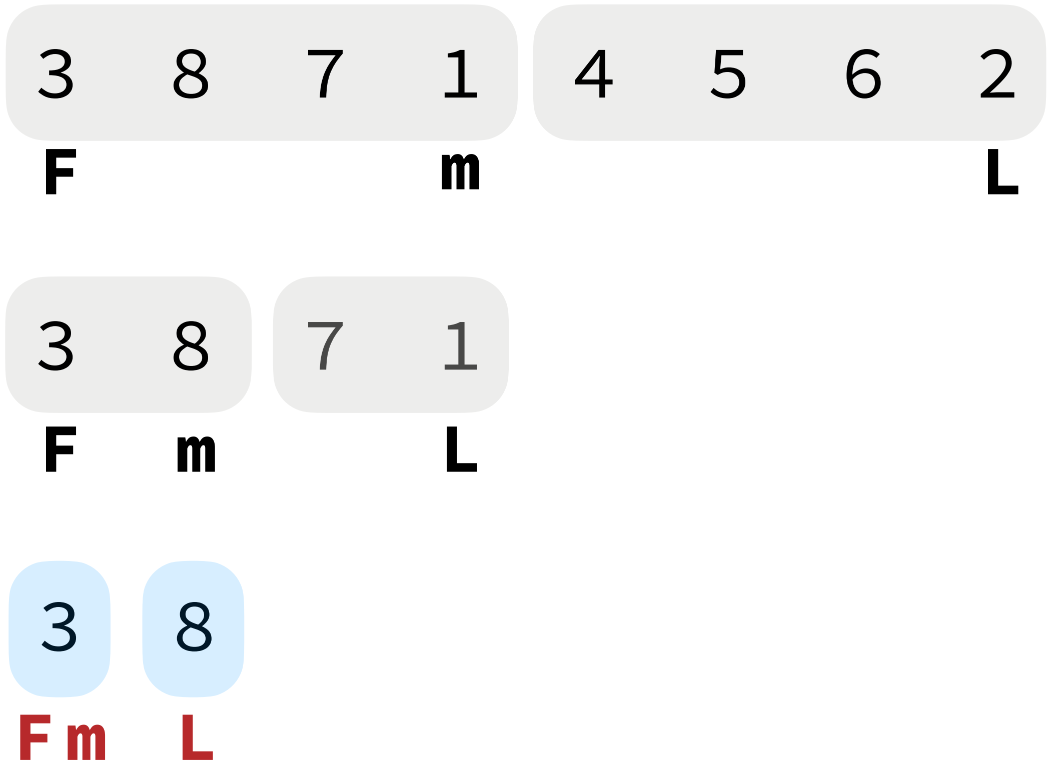
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



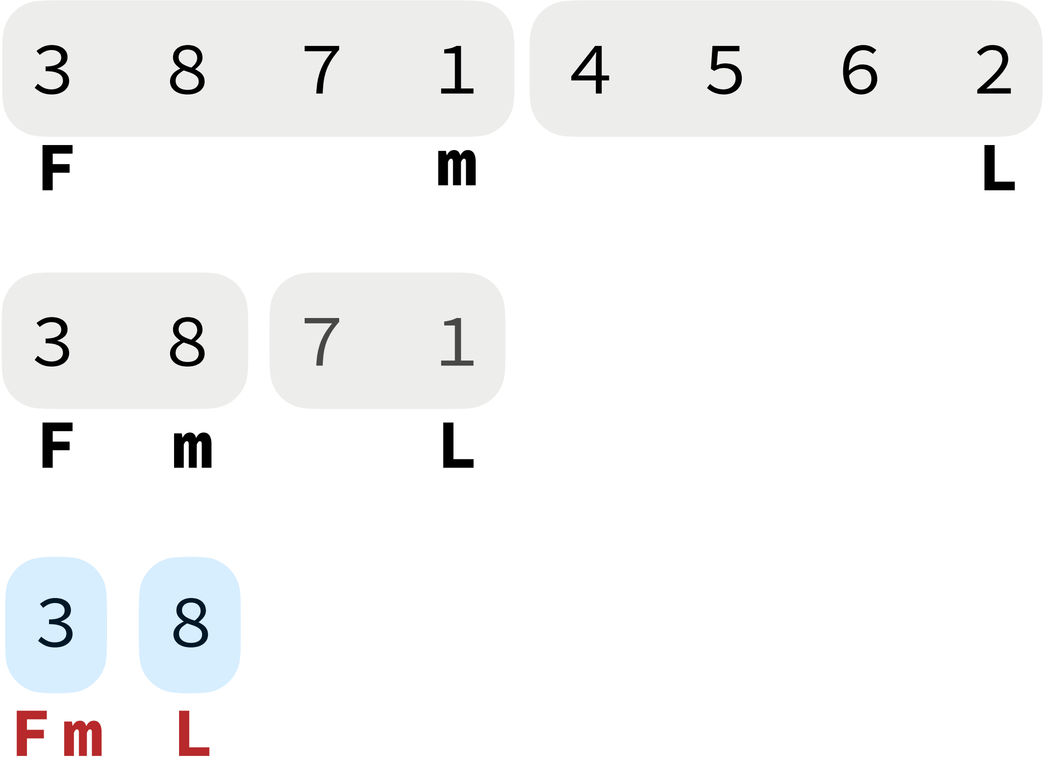
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
   return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

F m L

3 8 7 1

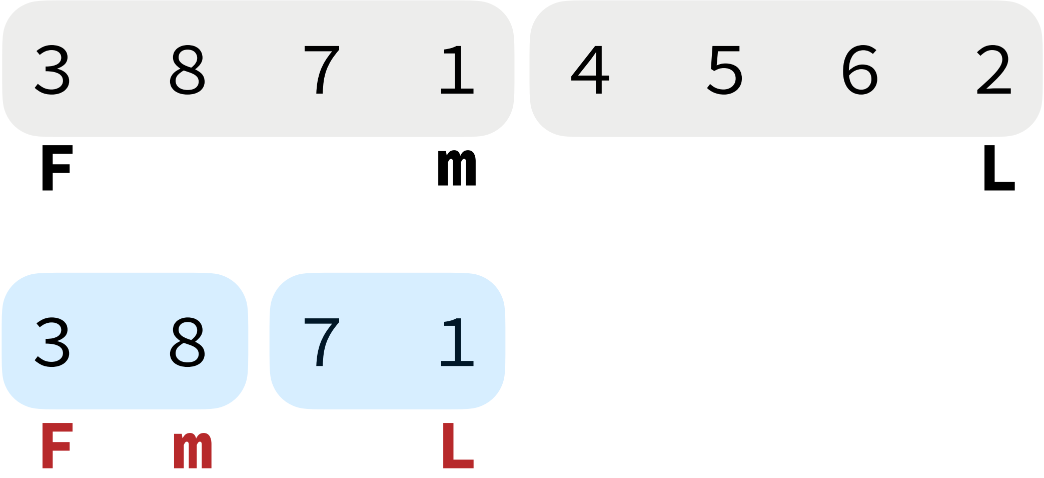
F m L

3 8

Fm L

Merge Sort Trace

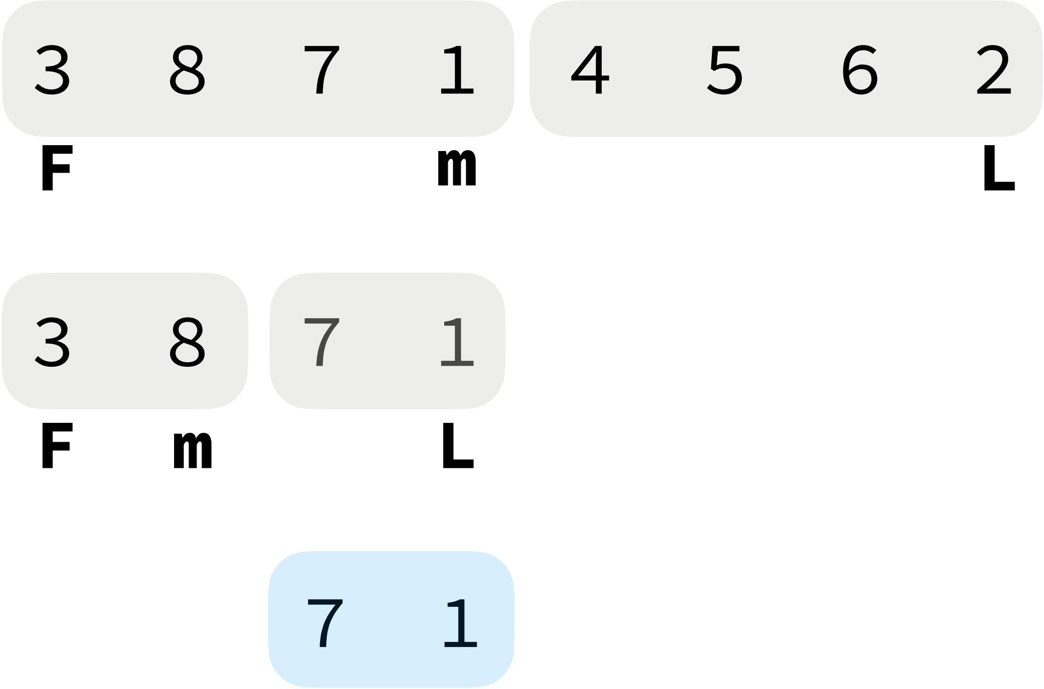
```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



```
3   8   7   1   4   5   6   2
F           m               L

3   8   7   1
F   m       L
```

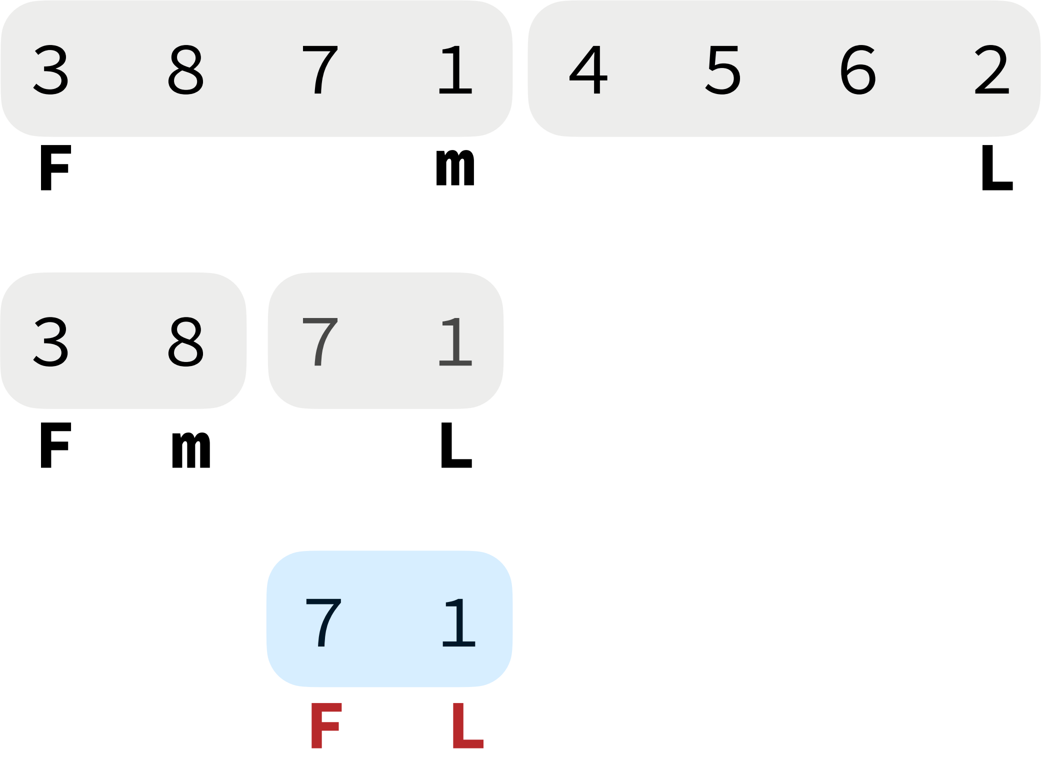
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

F m L

3 8 7 1

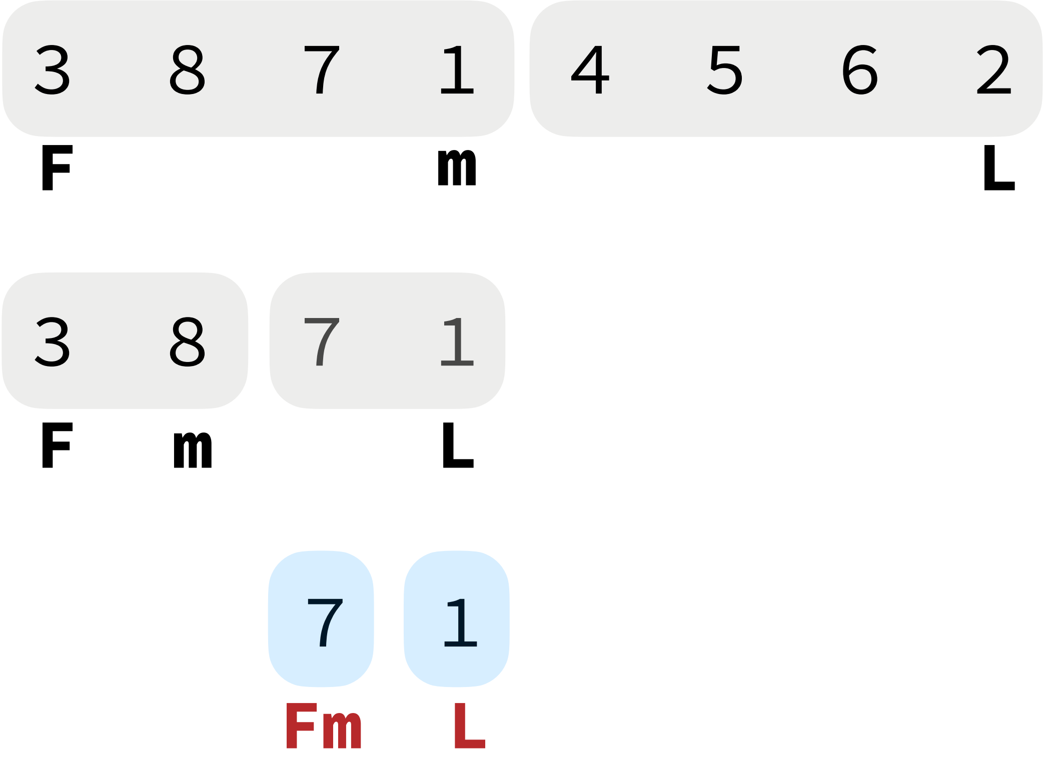
F m L

7 1

F L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

F m L

3 8 7 1

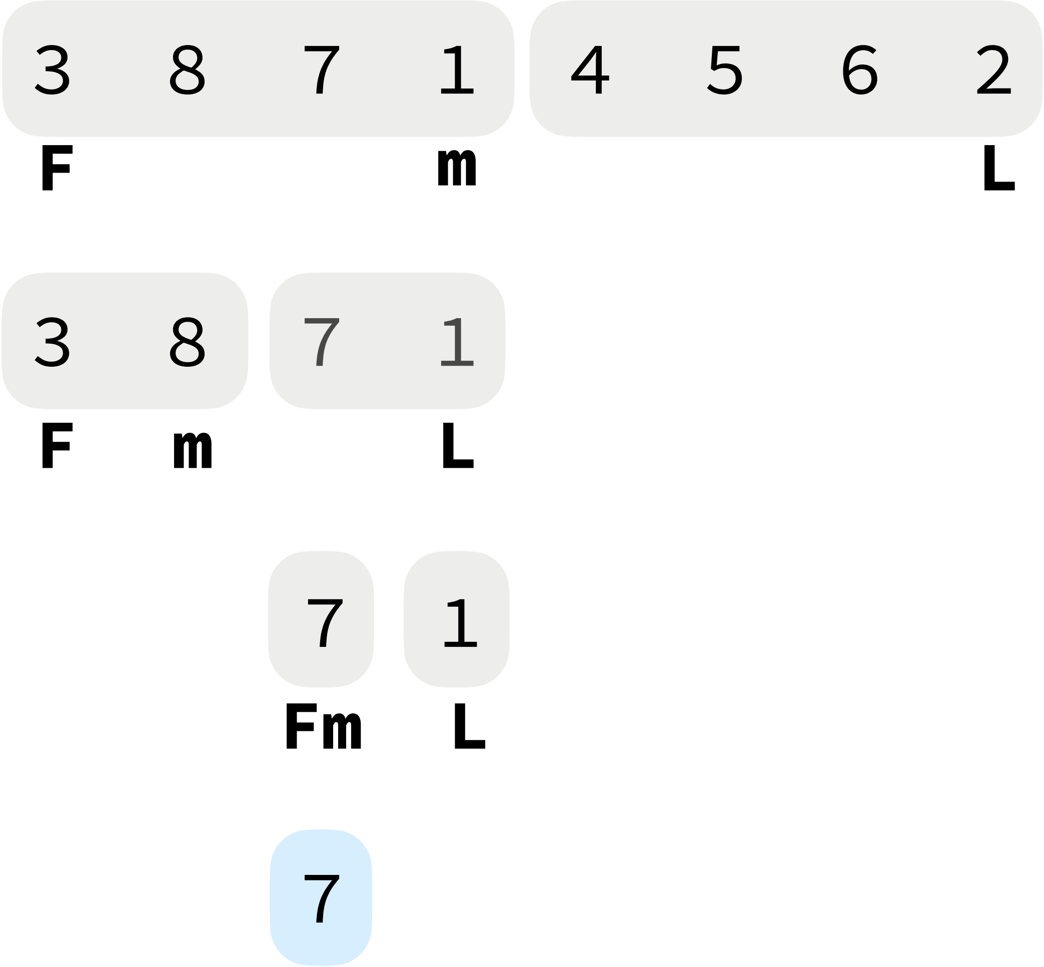
F m L

7 1

Fm L

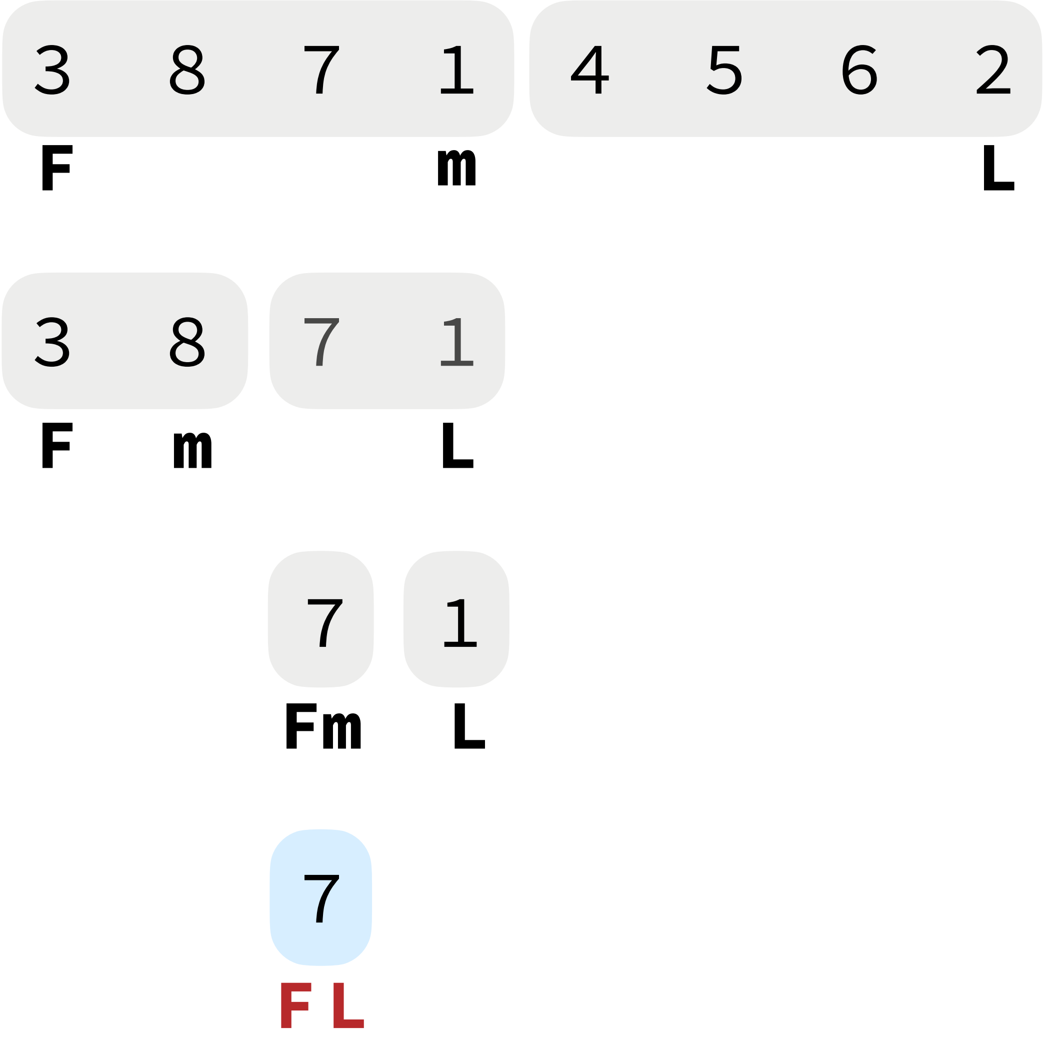
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2
F m L

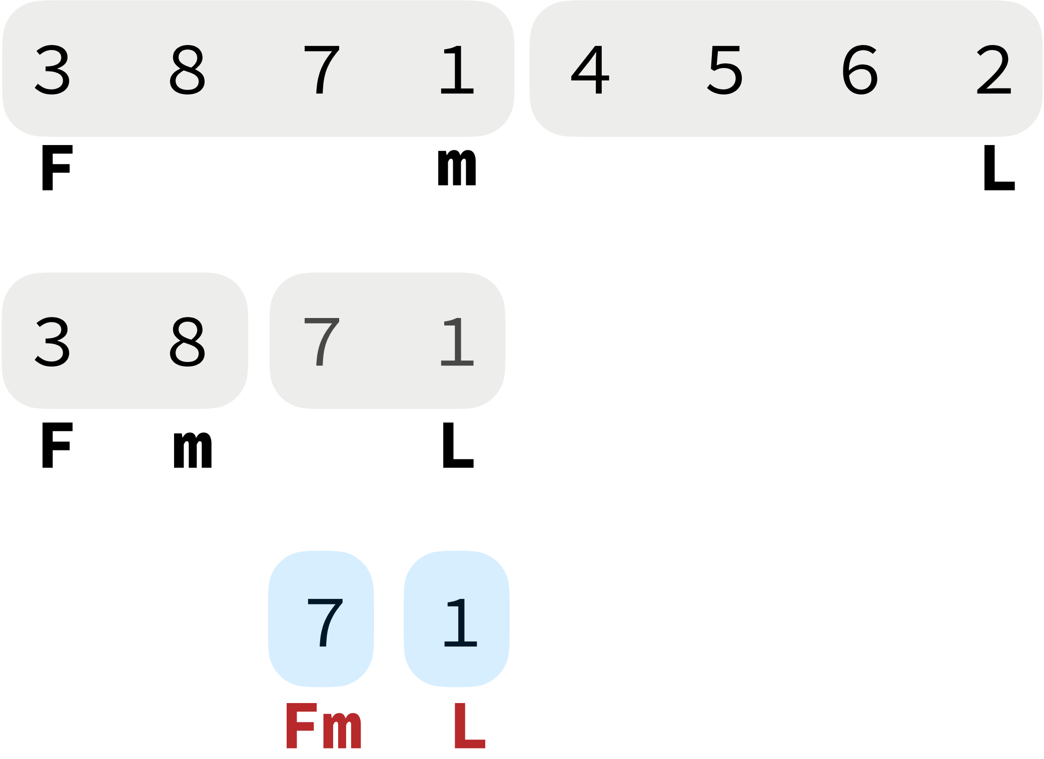
3 8 7 1
F m L

7 1
Fm L

7
F L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



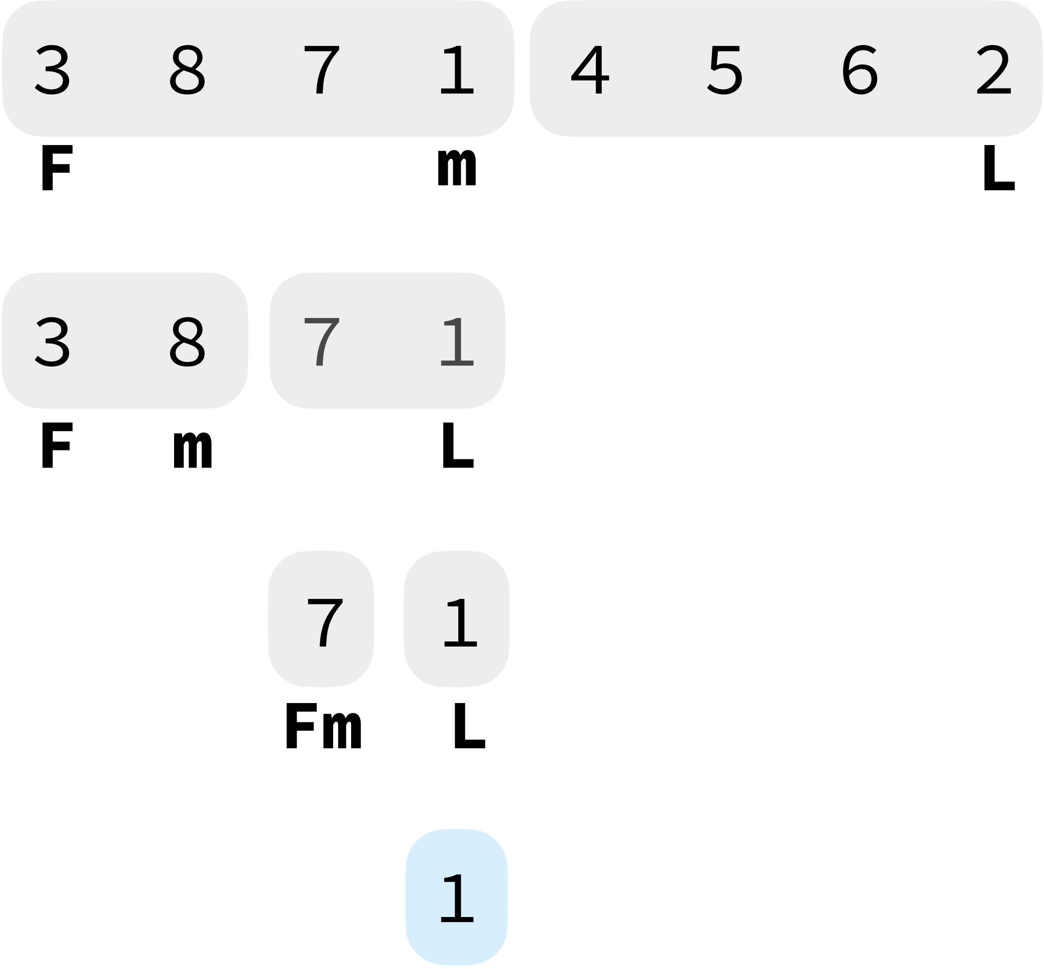
3 8 7 1 4 5 6 2
F m L

3 8 7 1
F m L

7 1
Fm L

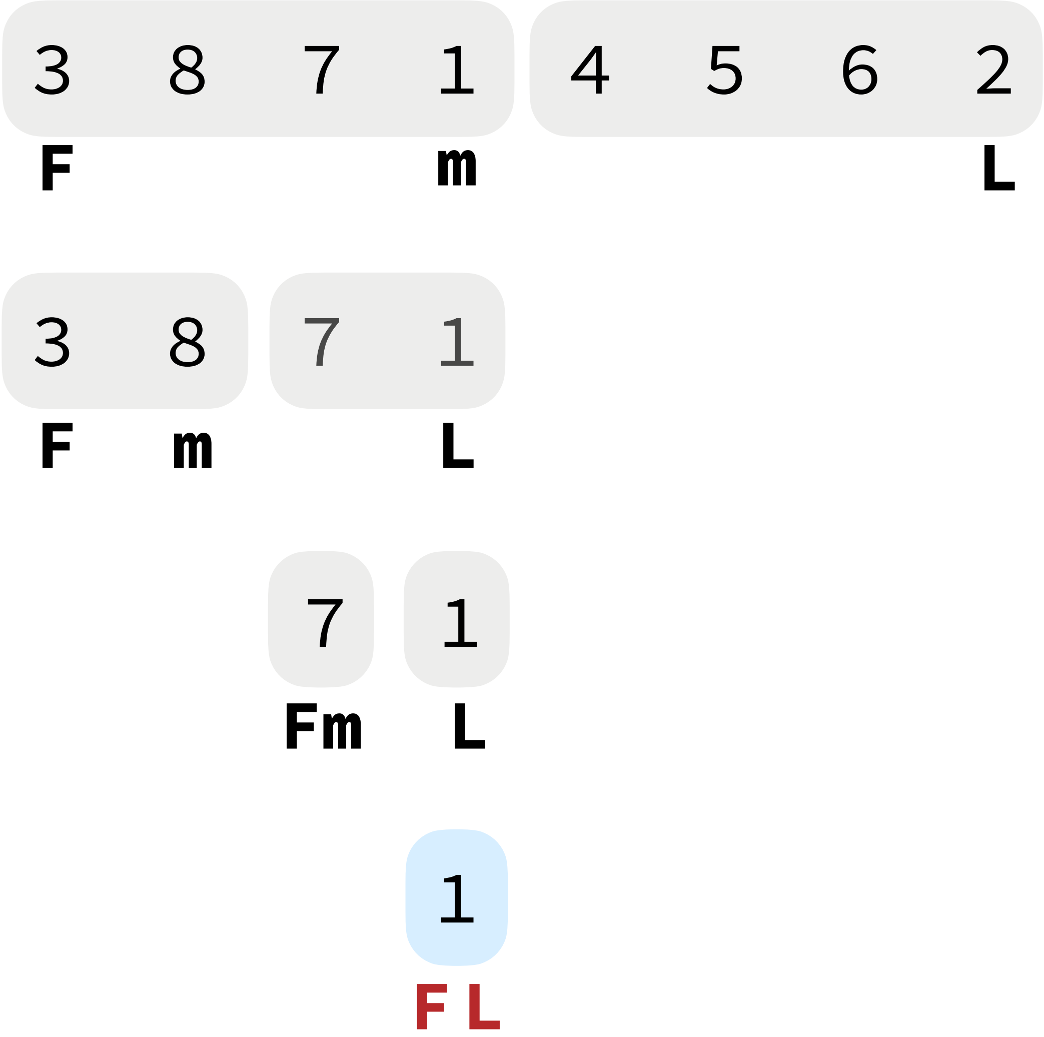
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



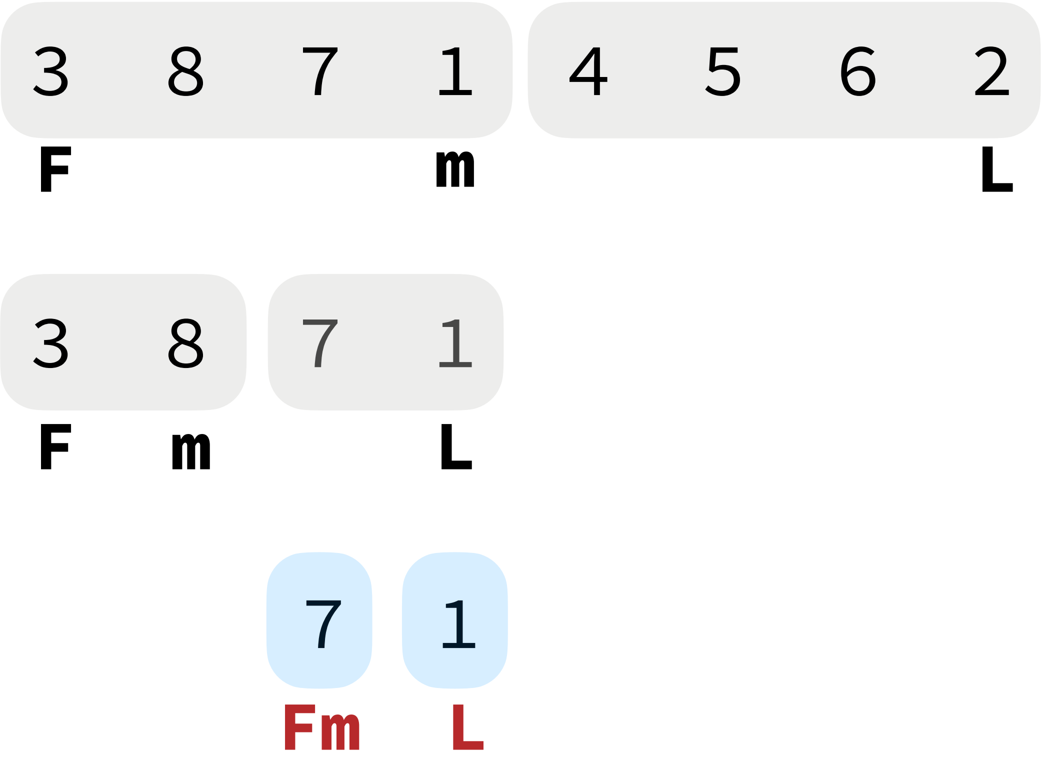
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
   return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



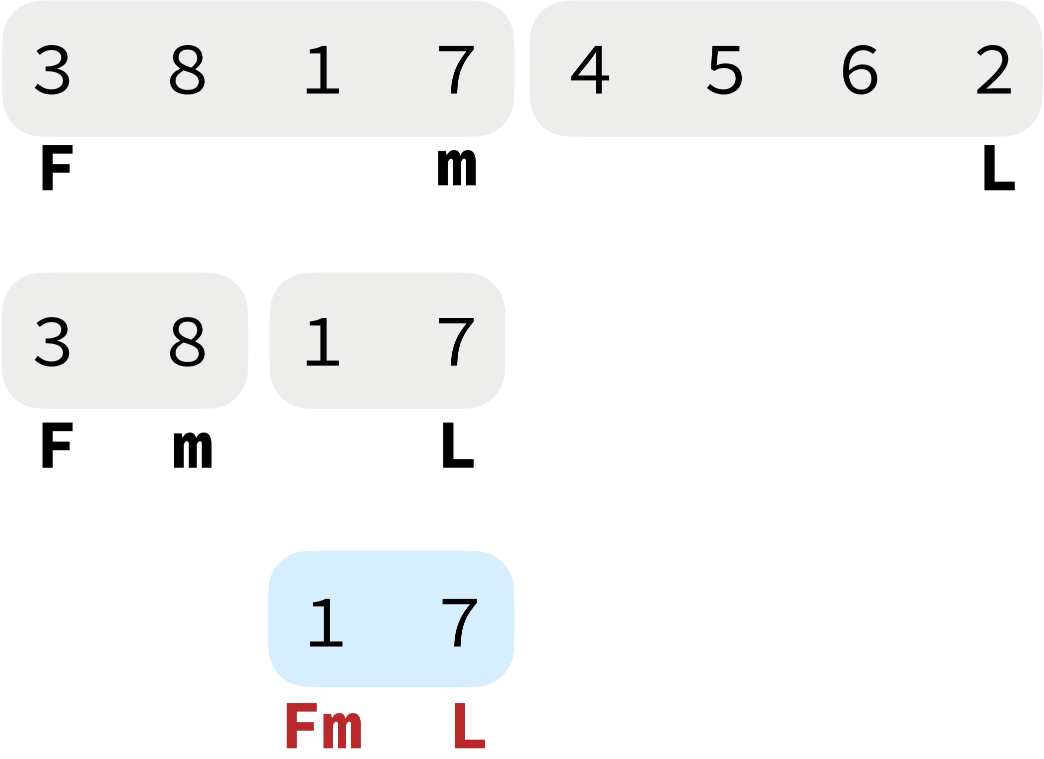
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 1 7 4 5 6 2

F m L

3 8 1 7

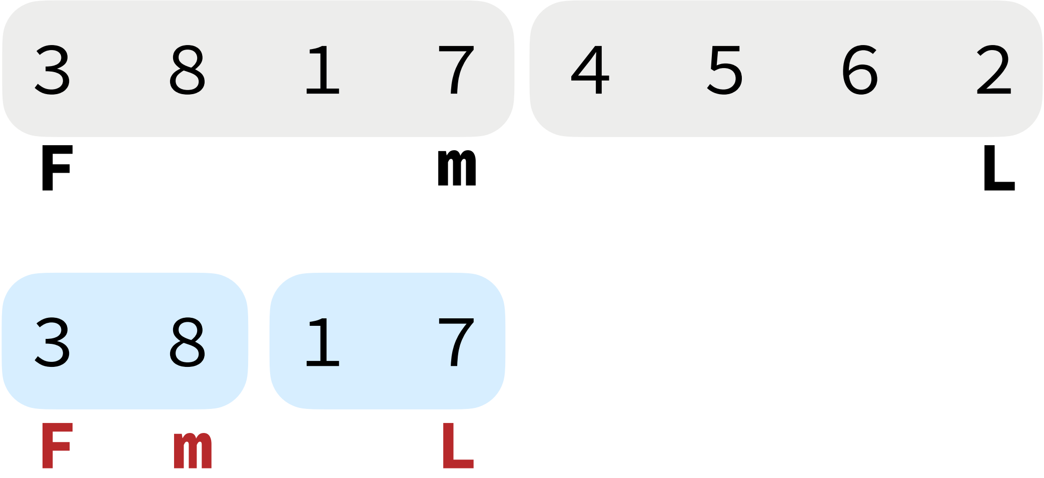
F m L

1 7

Fm L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 1 7 | 4 5 6 2

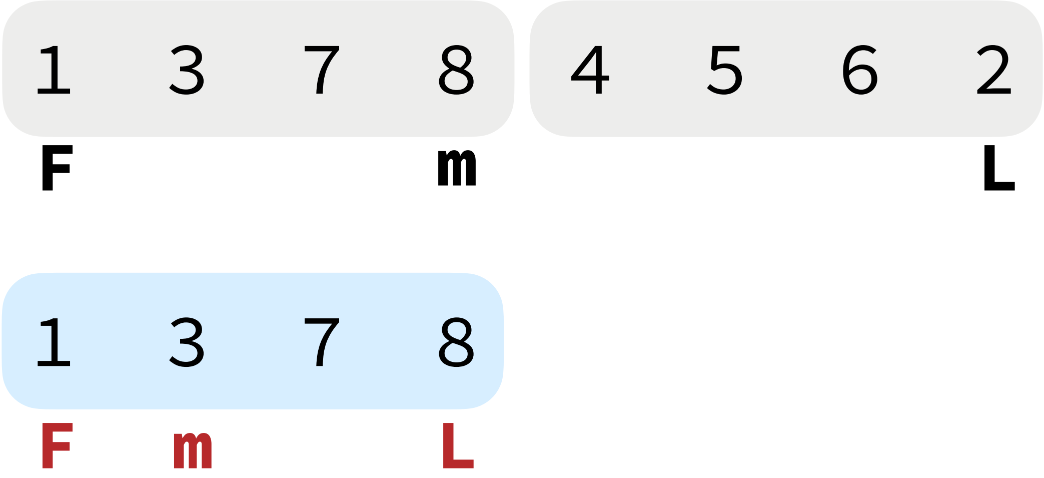
F m L

3 8 | 1 7

F m L

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



1 3 7 8 4 5 6 2

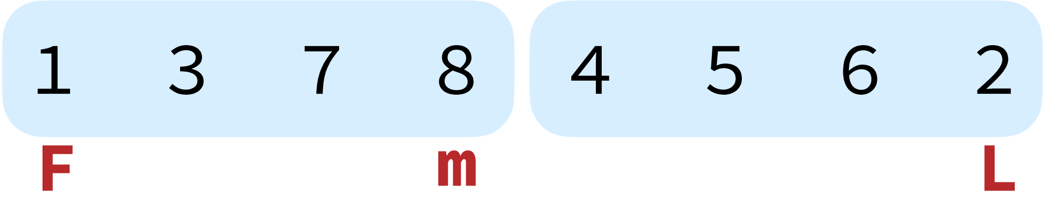
F m L

1 3 7 8

F m L

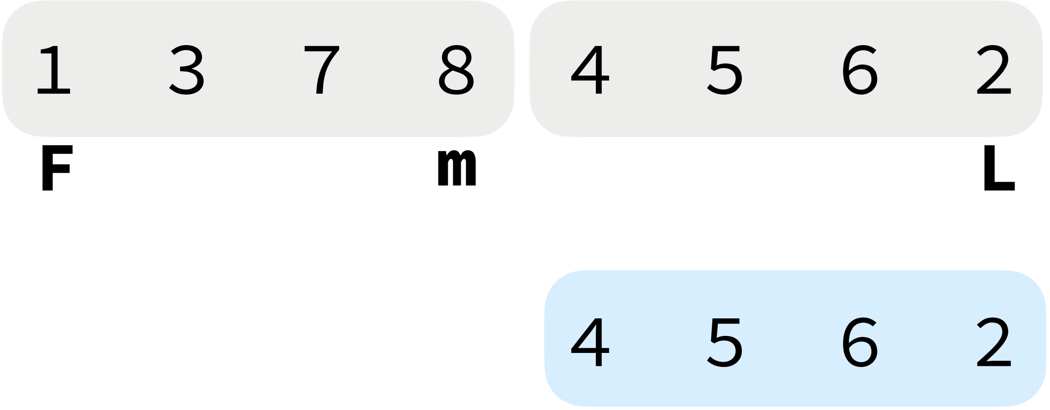
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
    return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



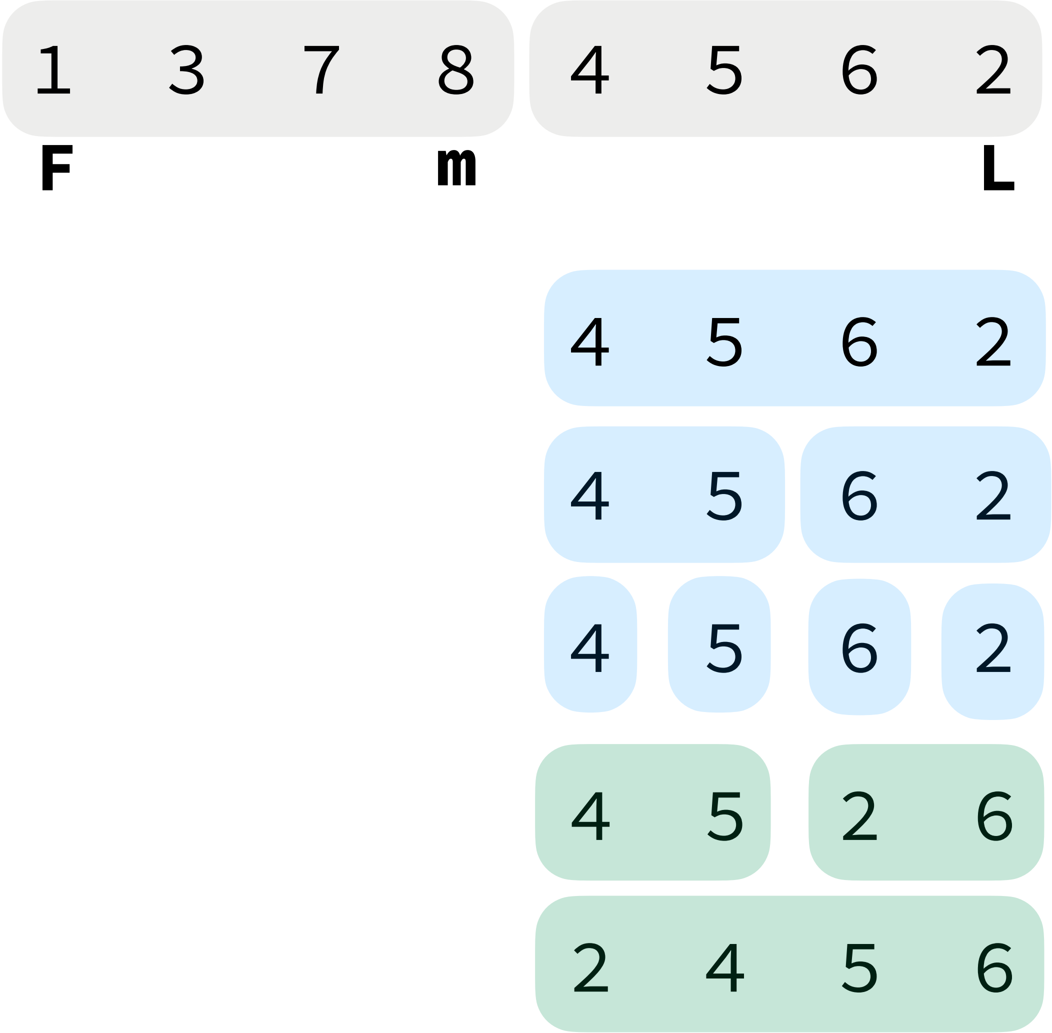
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



1 3 7 8 | 4 5 6 2

F m L

4 5 6 2

4 5 | 6 2

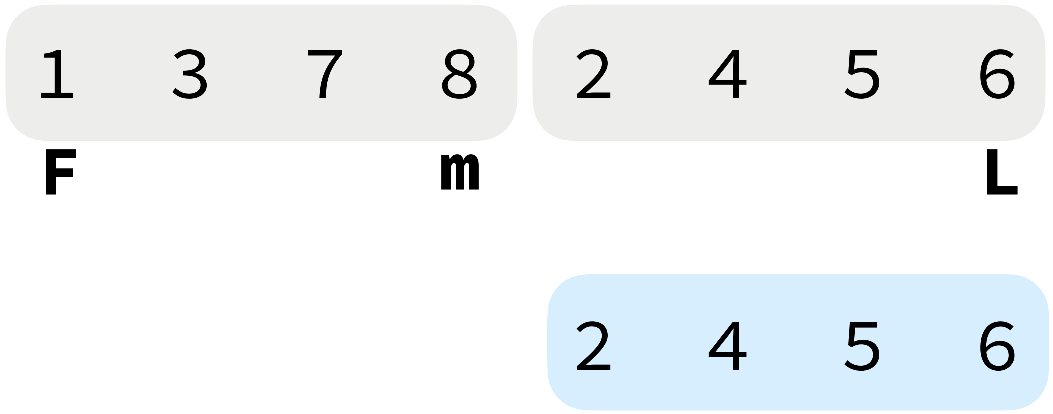
4 | 5 | 6 | 2

4 5 | 2 6

2 4 5 6

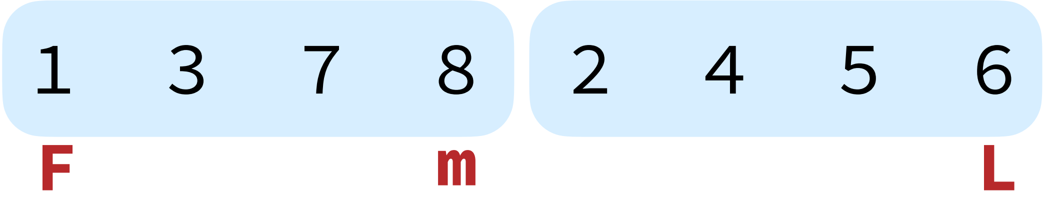
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



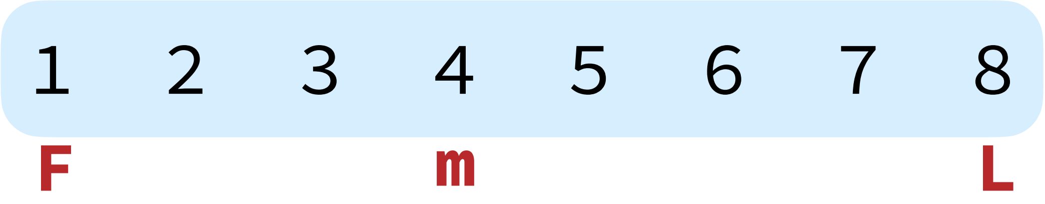
Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
   return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Merge Sort Trace

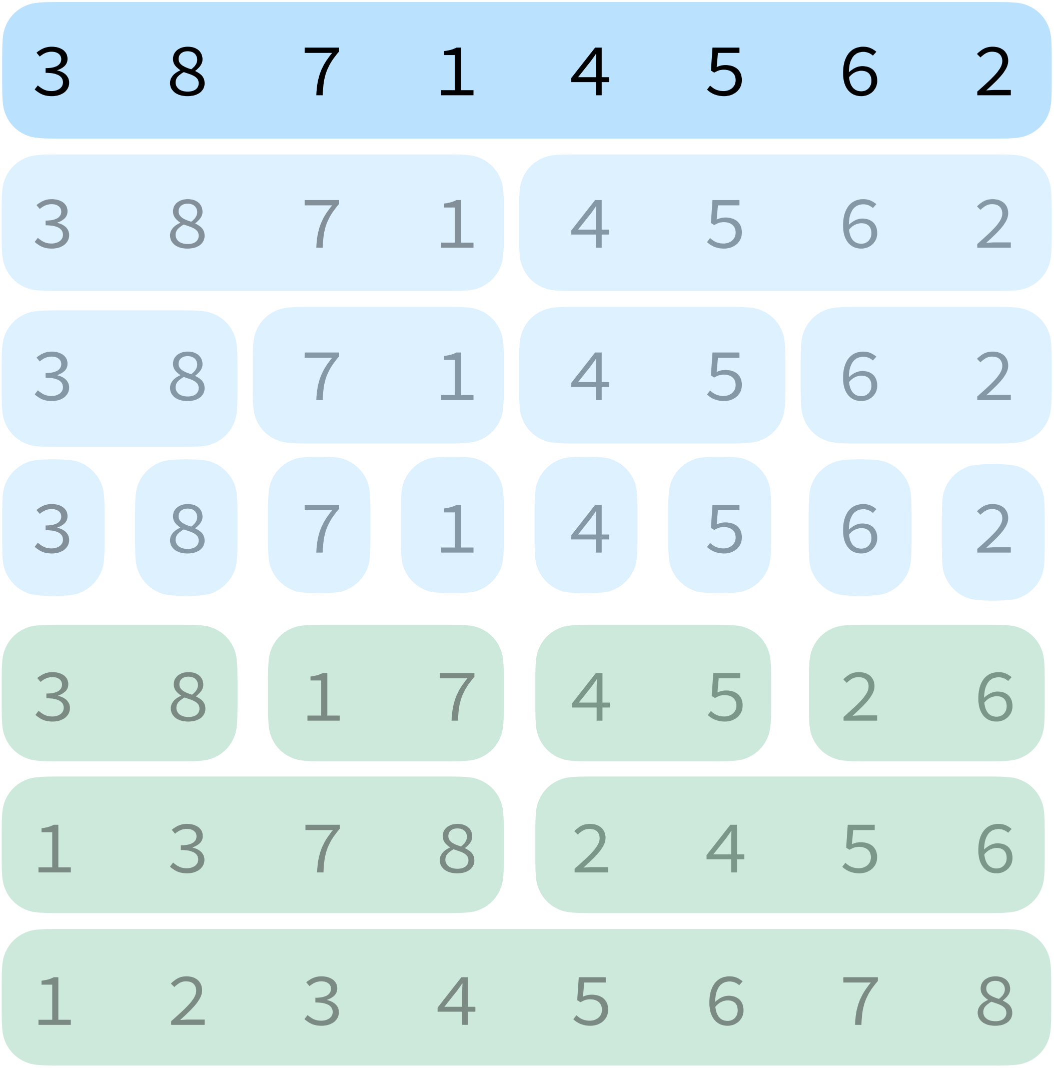
```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



```
1  2  3  4  5  6  7  8
F        m           L
```

Merge Sort Trace

```
MERGE-SORT(a[], first, last)

if first >= last
  return

mid = first + (last - first) / 2

MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



3 8 7 1 4 5 6 2

3 8 7 1 | 4 5 6 2

3 8 | 7 1 | 4 5 | 6 2

3 | 8 | 7 | 1 | 4 | 5 | 6 | 2

3 8 | 1 7 | 4 5 | 2 6

1 3 7 8 | 2 4 5 6

1 2 3 4 5 6 7 8

Merging

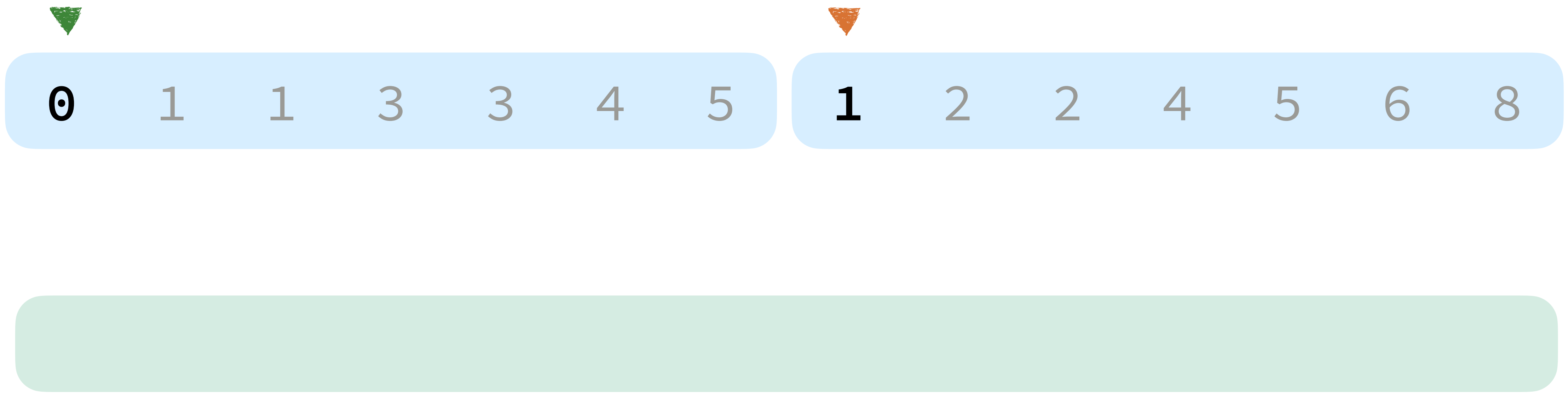
Sorted Arrays

Merging Sorted Arrays

```
0   1   1   3   3   4   5     1   2   2   4   5   6   8
```

Merged sorted array

Merging Sorted Arrays

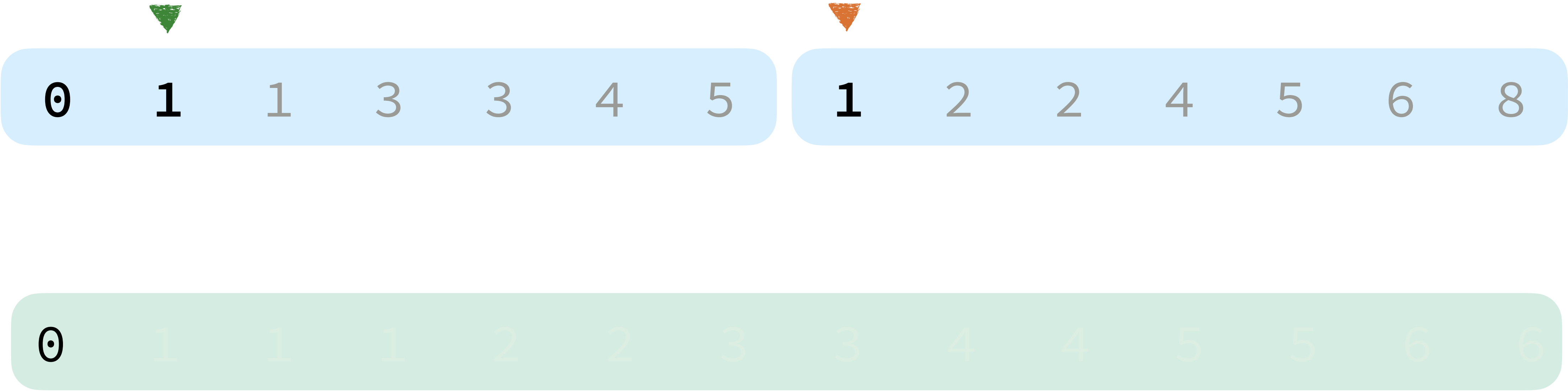


0 1 1 3 3 4 5

1 2 2 4 5 6 8

Merged sorted array

Merging Sorted Arrays



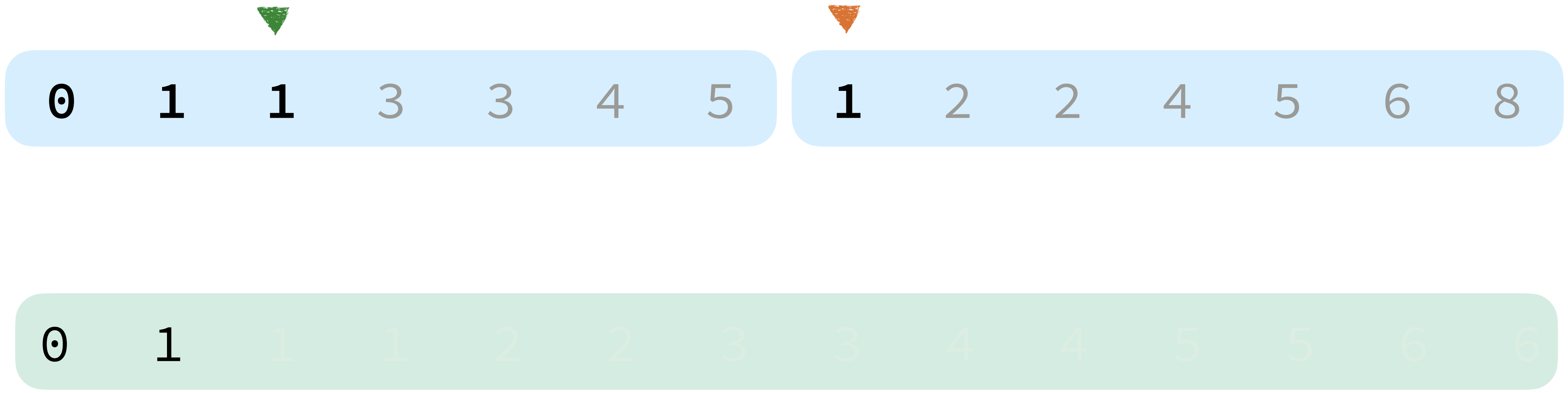
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



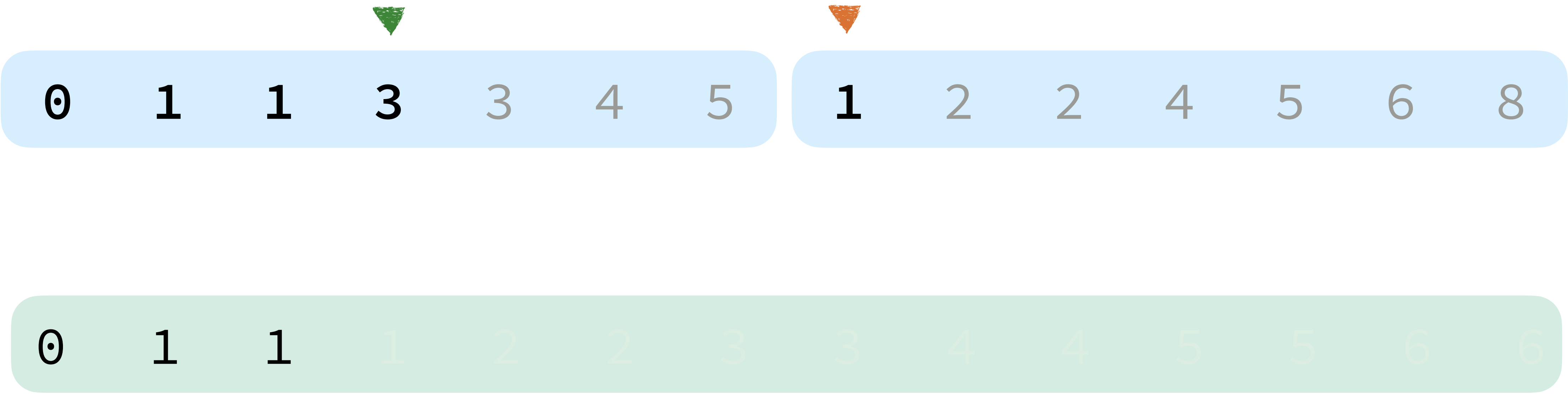
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

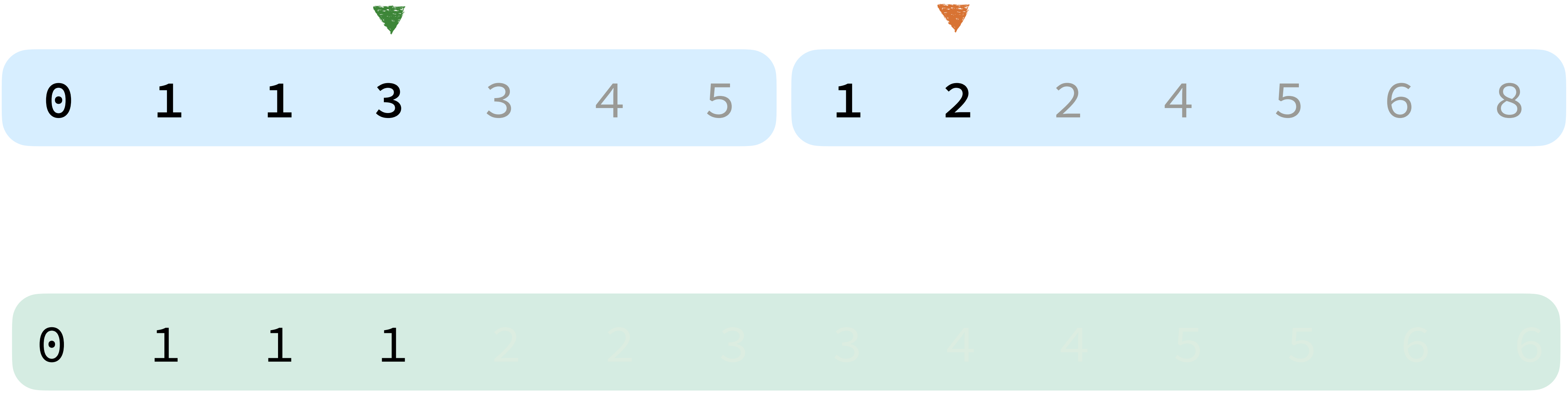
Merged sorted array

Merging Sorted Arrays



Merged sorted array

Merging Sorted Arrays



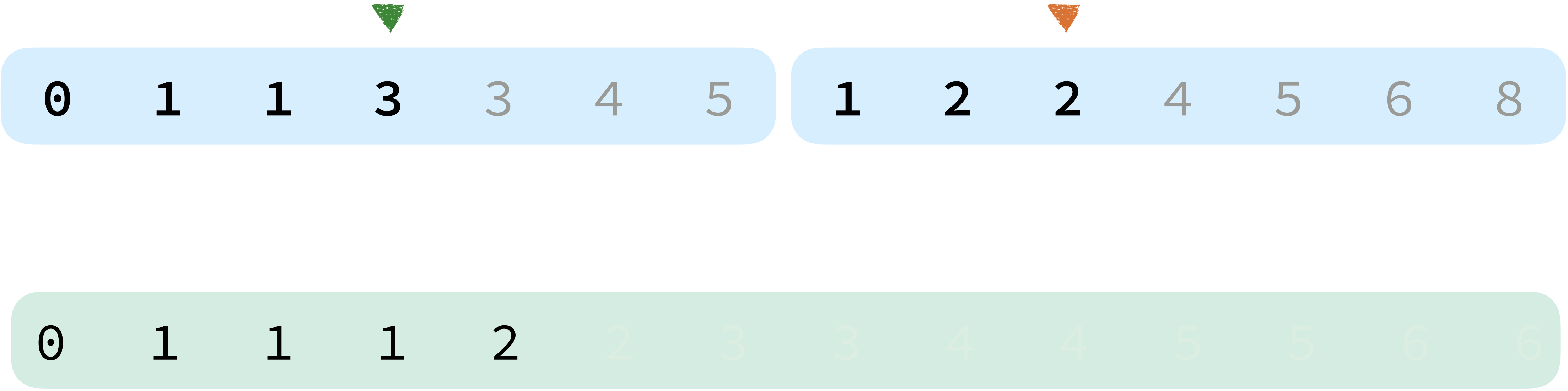
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



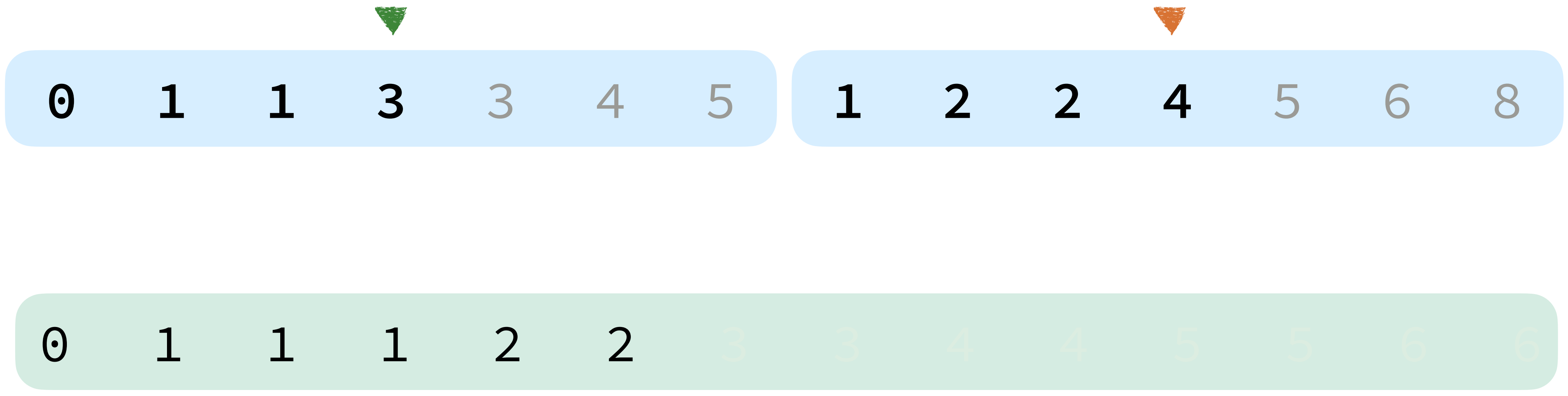
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



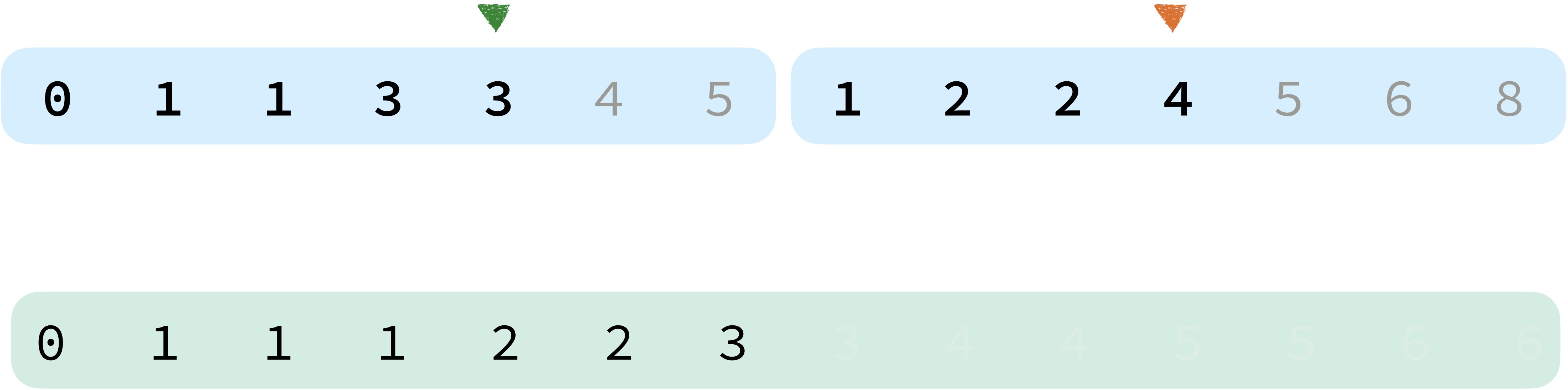
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



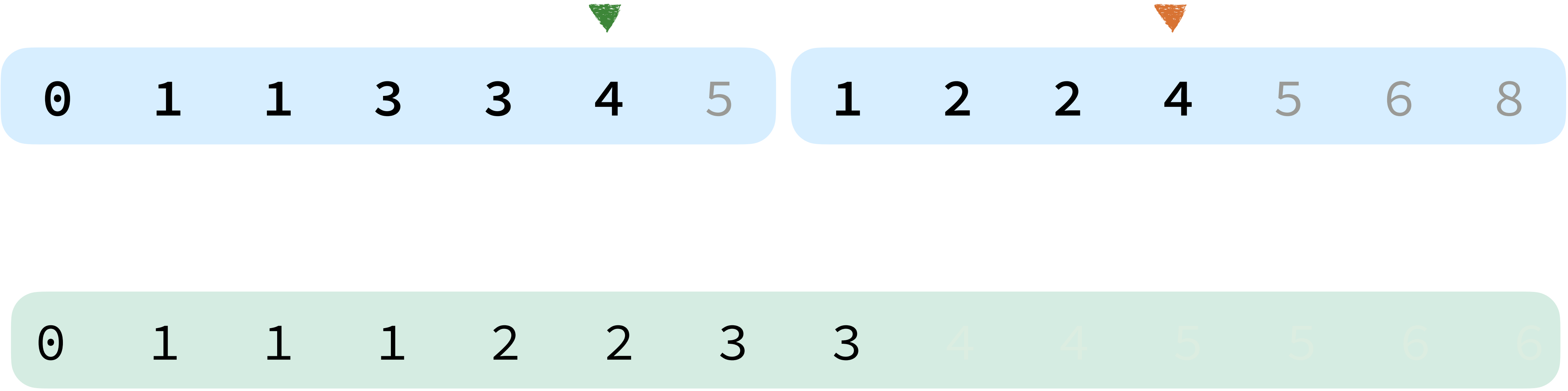
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



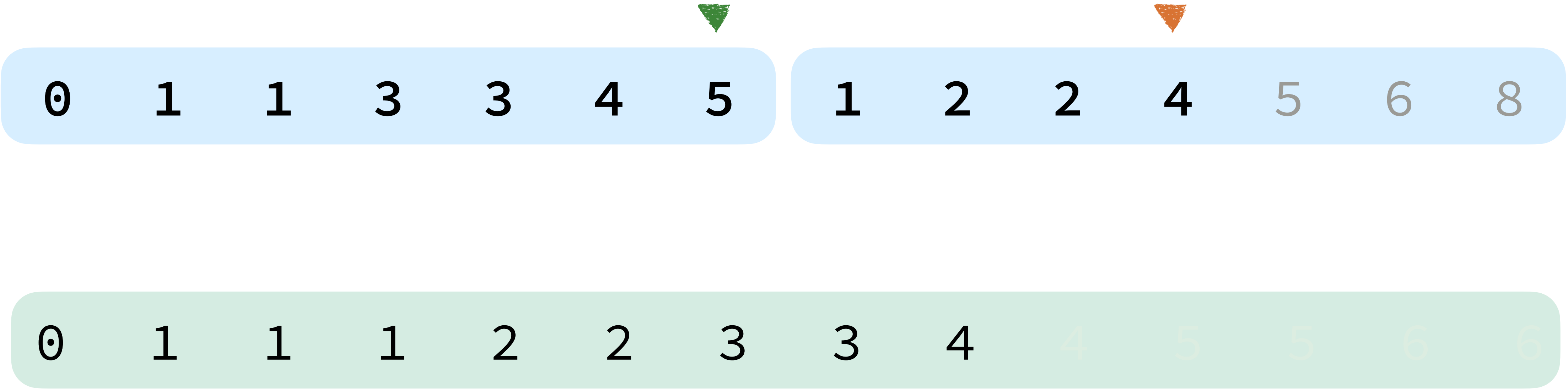
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



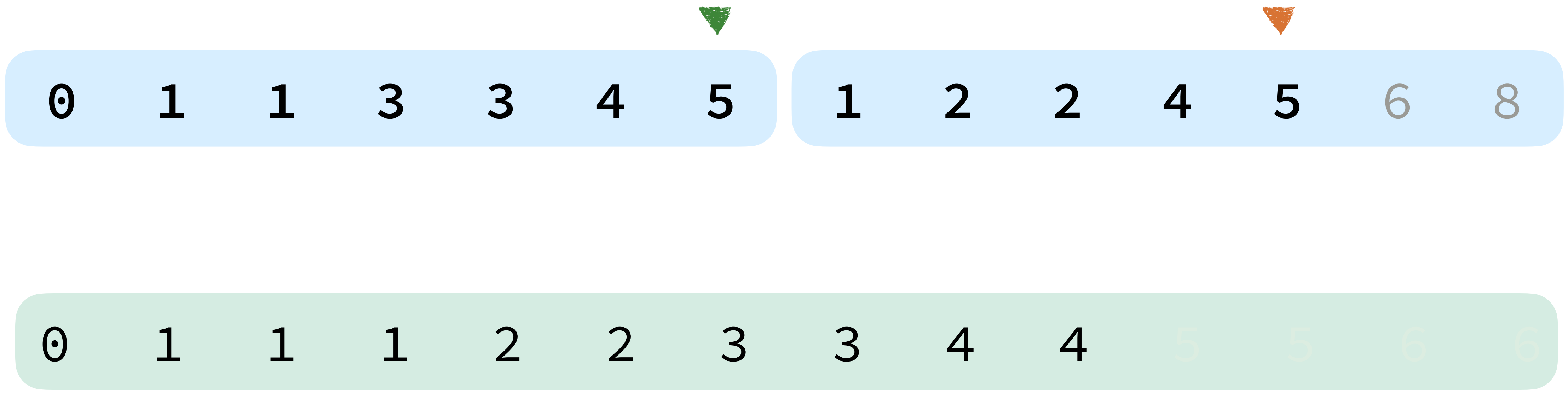
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



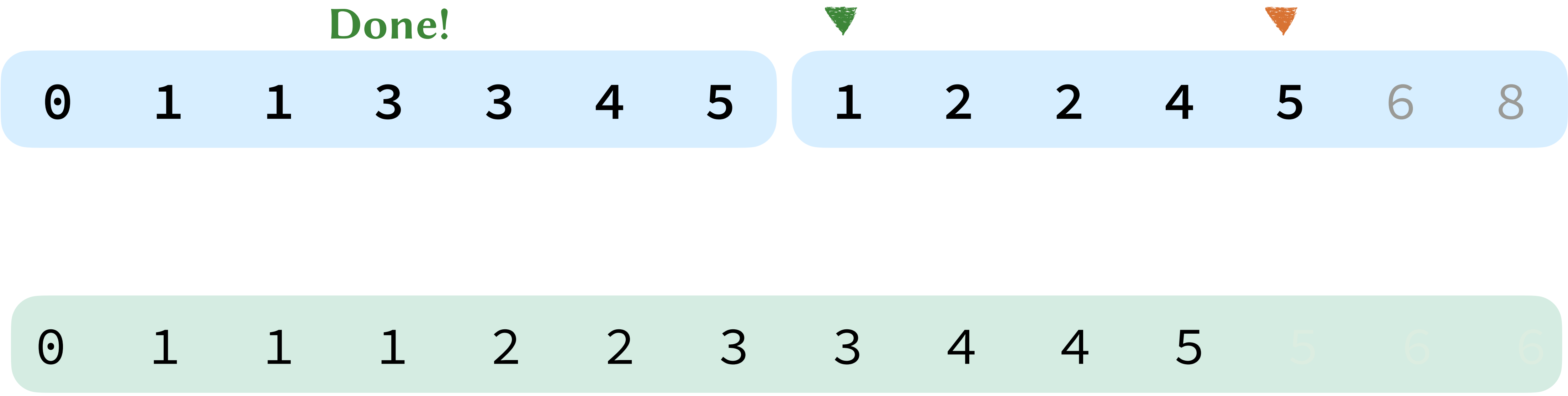
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



Done!

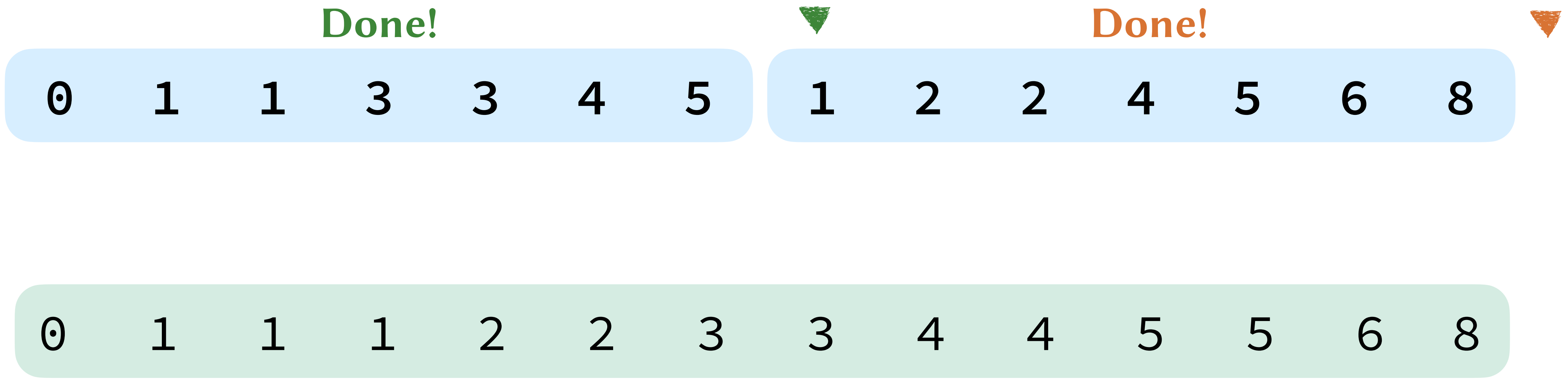
0 1 1 3 3 4 5

1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 6

Merged sorted array

Merging Sorted Arrays



Done! Done!

0 1 1 3 3 4 5

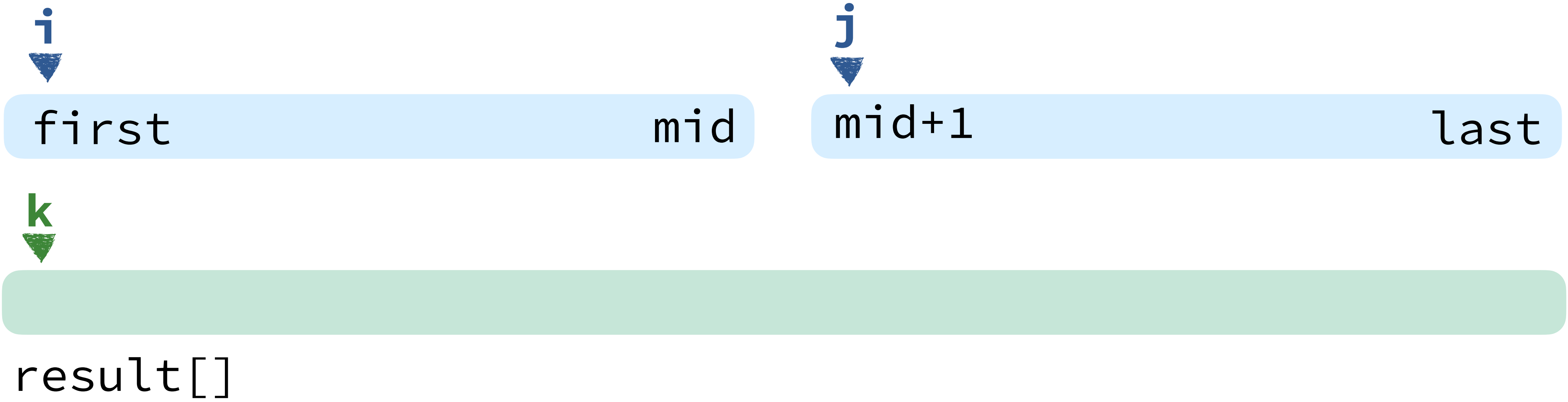
1 2 2 4 5 6 8

0 1 1 1 2 2 3 3 4 4 5 5 6 8

Merged sorted array

Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

 create array result[] of size (last - first + 1)
 i = first,  j = mid+1
```



i

j

first

mid

mid+1

last

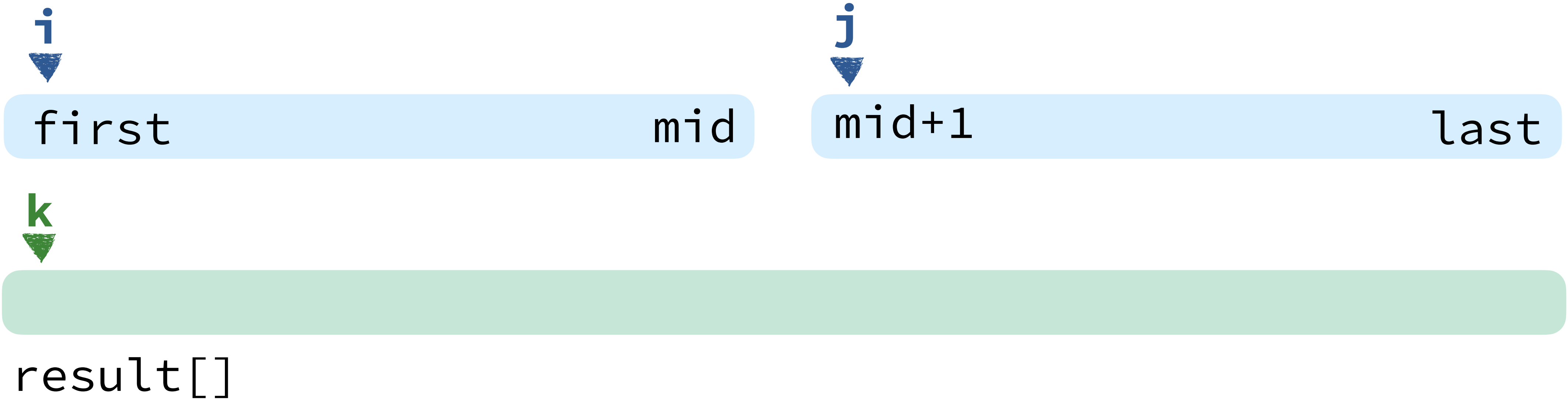
k

result[]

Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

 create array result[] of size (last - first + 1)
 i = first,   j = mid+1

 for (k = 0;   k < size;   k++):
```

we know exactly how many elements will be copied



i

j

first mid

mid+1 last

k

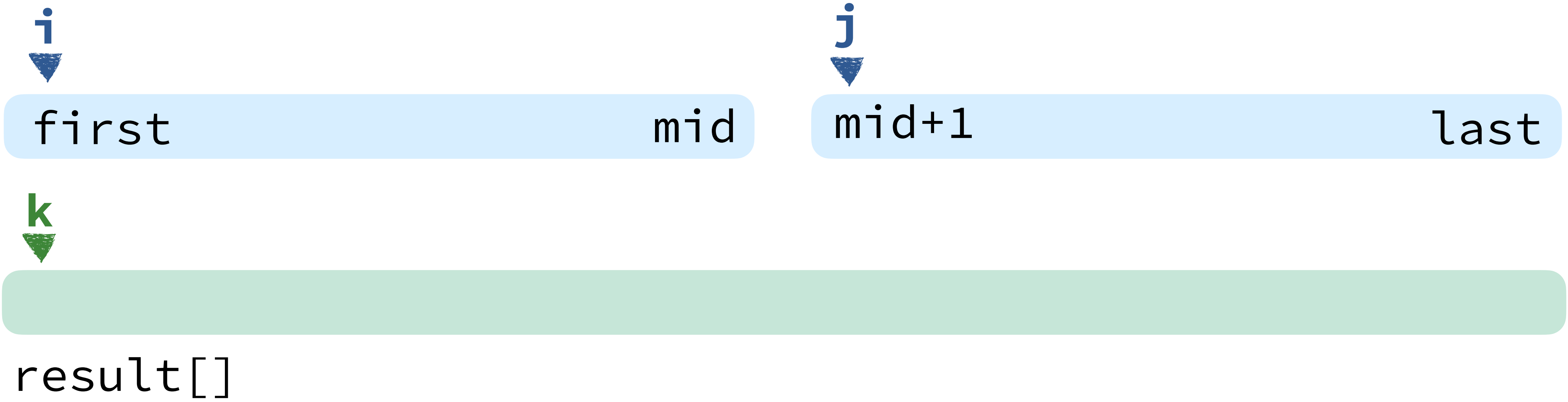
result[]

Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

create array result[] of size (last - first + 1)
i = first,   j = mid+1

for (k = 0;  k < size;  k++):
    if         i > mid:         result[k] = a[j++]
```

no more elements in the left half

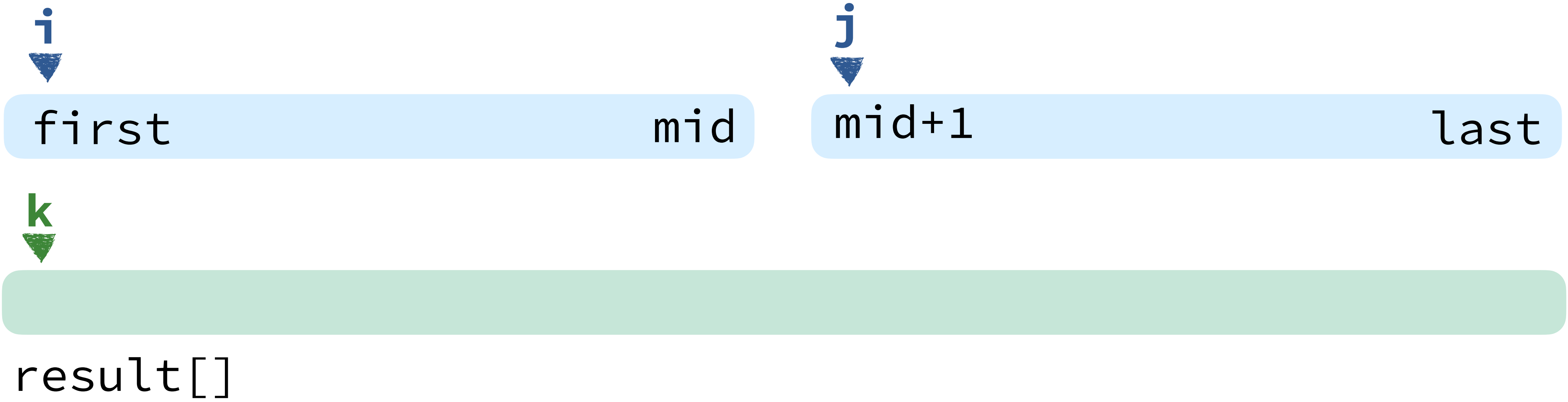
assuming that j++ performs a post-increment as in C++



Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

create array result[] of size (last - first + 1)
i = first,  j = mid+1

for (k = 0;  k < size;  k++):
    if        i > mid:        result[k] = a[j++]
    else if   j > last:       result[k] = a[i++]
```

no more elements in the right half



i

j

first mid

mid+1 last

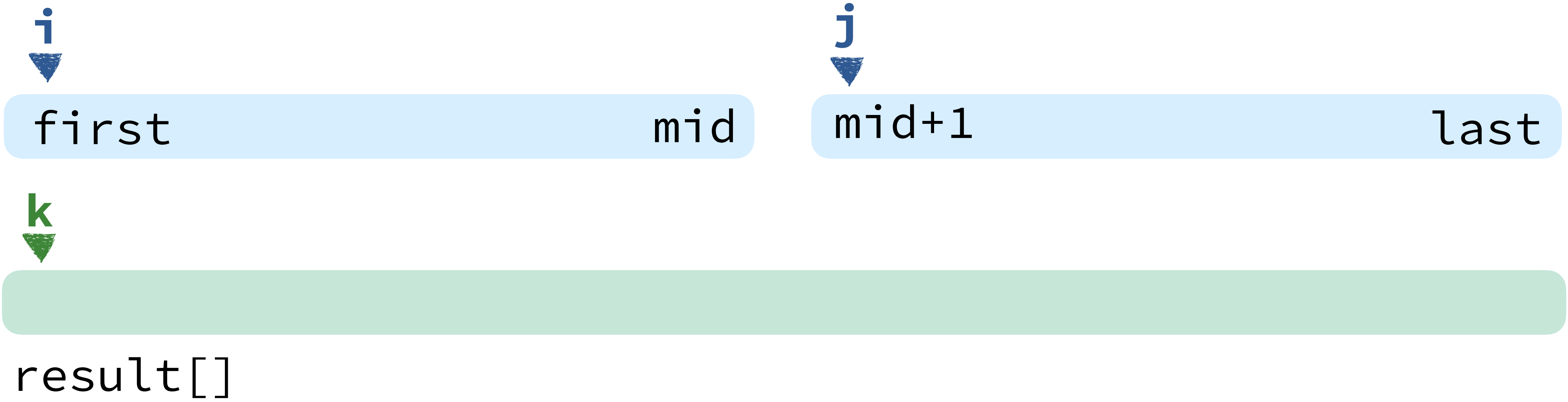
k

result[]

Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

create array result[] of size (last - first + 1)
i = first,  j = mid+1

for (k = 0;  k < size;  k++):
    if        i > mid:        result[k] = a[j++]
    else if   j > last:       result[k] = a[i++]
    else if   a[i] <= a[j]:   result[k] = a[i++]
    else:                     result[k] = a[j++]
```

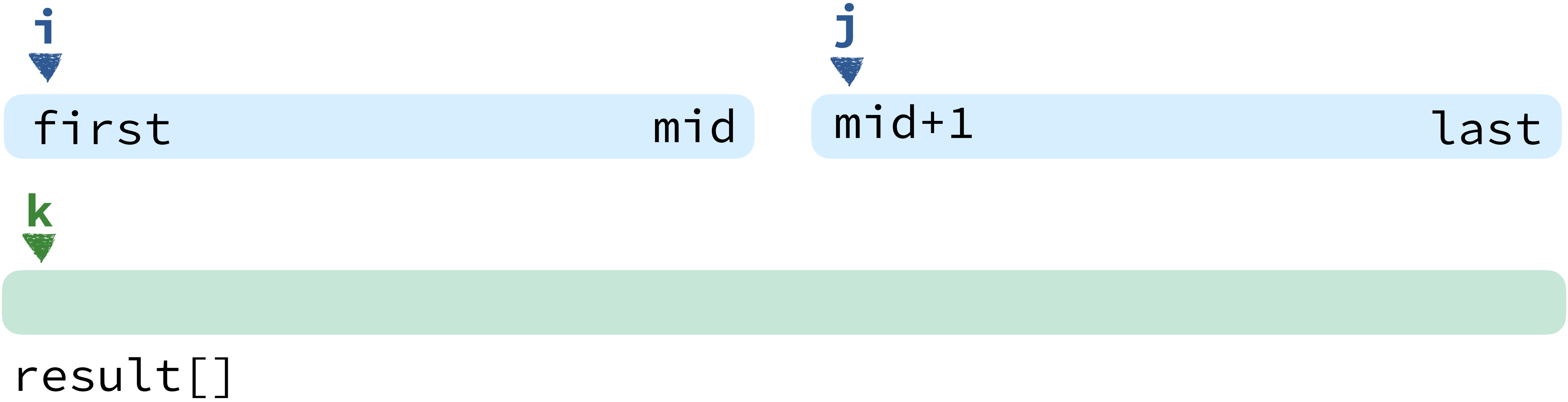
compare the elements and copy the smaller


i
j
first
mid
mid+1
last
k
result[]

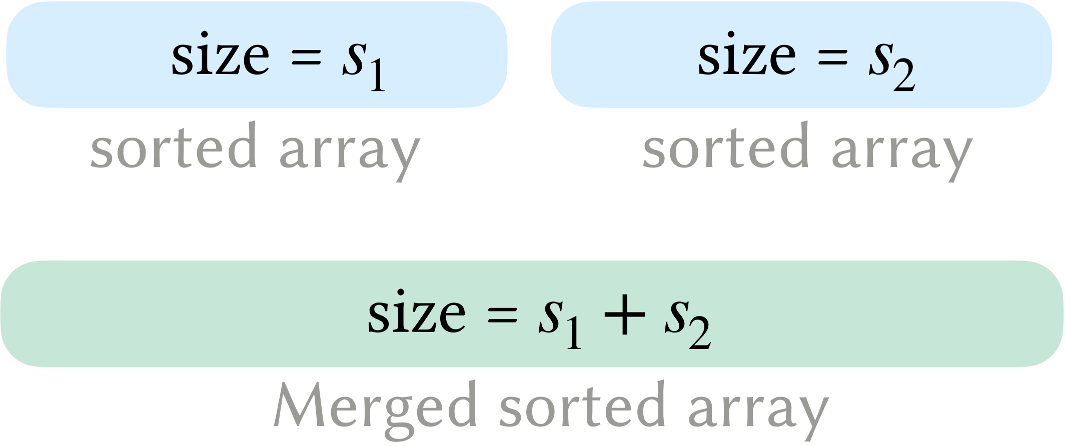
Merging Sorted Arrays

```
MERGE(a[], first, mid, last)

create array result[] of size (last - first + 1)
i = first,  j = mid+1

for (k = 0;  k < size;  k++):
    if         i > mid:         result[k] = a[j++]
    else if    j > last:        result[k] = a[i++]
    else if    a[i] <= a[j]:    result[k] = a[i++]
    else:                       result[k] = a[j++]

copy result[] into a[first ... last]
```

we assume the array result is local to the function and is deleted once the function terminates



Merging Sorted Arrays (Analysis)



size = s_1

sorted array

size = s_2

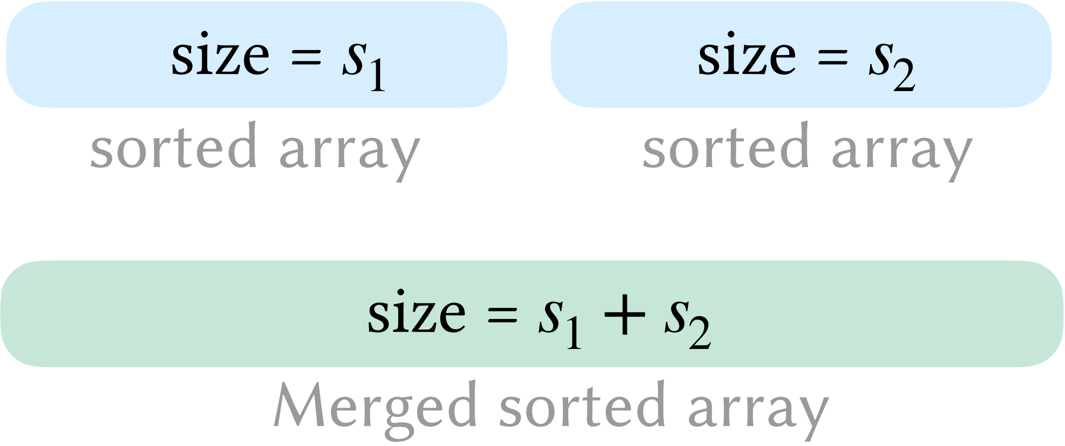
sorted array

size = $s_1 + s_2$

Merged sorted array

Number of Data Moves:

Merging Sorted Arrays (Analysis)



size = s_1

sorted array

size = s_2

sorted array

size = $s_1 + s_2$

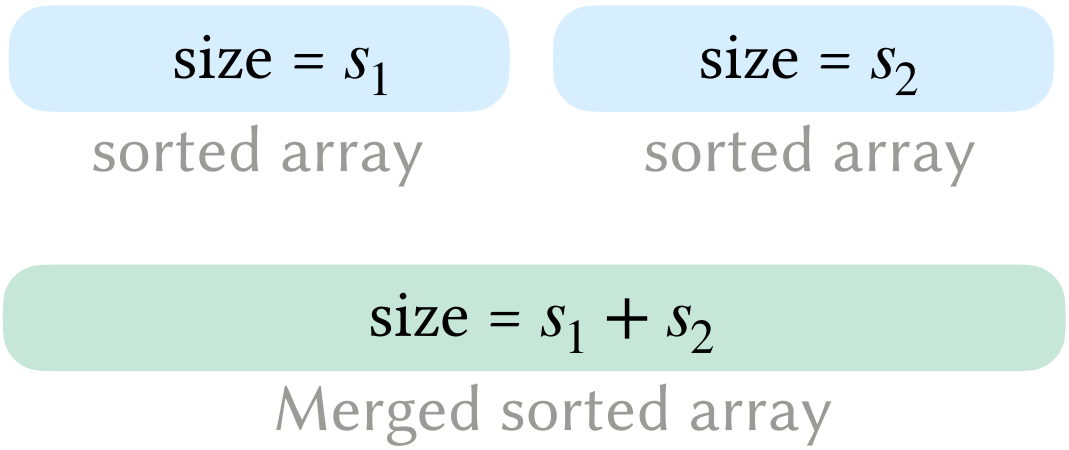
Merged sorted array

Number of Data Moves:

- Worst case: $2(s_1 + s_2)$ data moves.
- Best case: $2(s_1 + s_2)$ data moves.

all elements in both subarrays have to be copied to the merged array and then back to the original array

Merging Sorted Arrays (Analysis)



size = s_1

sorted array

size = s_2

sorted array

size = $s_1 + s_2$

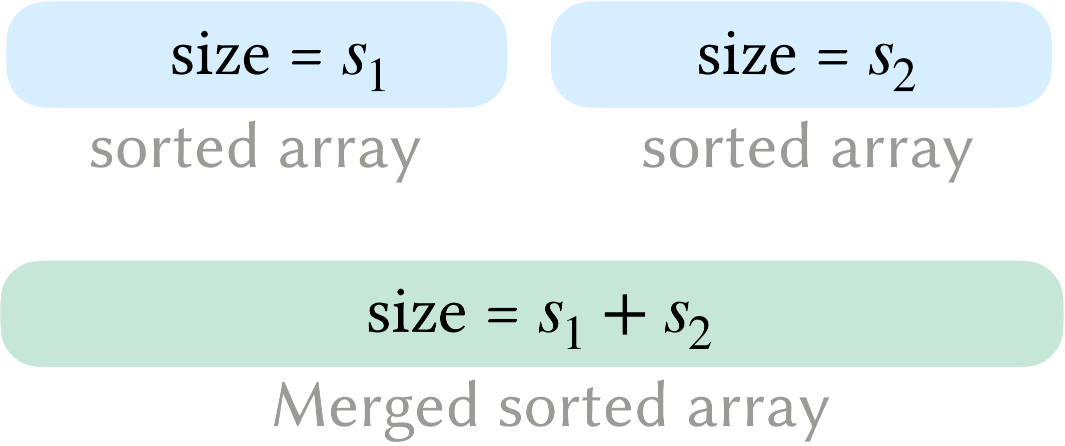
Merged sorted array

Number of Data Moves:

- Worst case: $2(s_1 + s_2)$ data moves.
- Best case: $2(s_1 + s_2)$ data moves.

Number of Data Compares:

Merging Sorted Arrays (Analysis)



size = s_1

sorted array

size = s_2

sorted array

size = $s_1 + s_2$

Merged sorted array

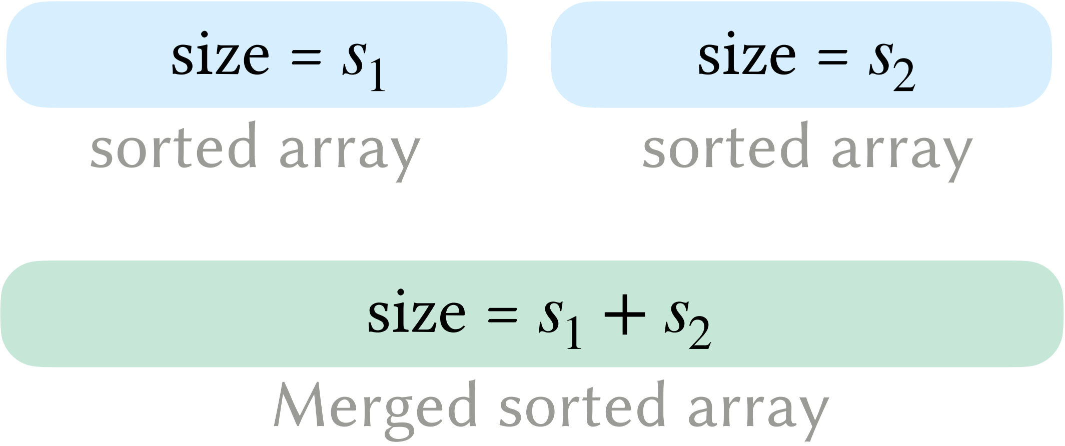
Number of Data Moves:

- Worst case: $2(s_1 + s_2)$ data moves.
- Best case: $2(s_1 + s_2)$ data moves.

Number of Data Compares:

- Worst case: $s_1 + s_2 - 1$ compares (e.g. merge [1, 3, 5] with [0, 2, 4]).
- Best case: $\min(s_1, s_2)$ compares (e.g. merge [7, 8, 9, 10] with [0, 2]).

Merging Sorted Arrays (Analysis)



size = s_1

sorted array

size = s_2

sorted array

size = $s_1 + s_2$

Merged sorted array

Number of Data Moves:

- Worst case: $2(s_1 + s_2)$ data moves.
- Best case: $2(s_1 + s_2)$ data moves.

Number of Data Compares:

- Worst case: $s_1 + s_2 - 1$ compares (e.g. merge [1, 3, 5] with [0, 2, 4]).
- Best case: $\min(s_1, s_2)$ compares (e.g. merge [7, 8, 9, 10] with [0, 2]).

For Merge Sort

$\Theta(n)$ work is needed to merge two sorted arrays of size $\frac{n}{2}$ each.

(considering data compares and moves)

Merge Sort Analysis

Number of Compares: $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n - 1$

(in the worst case)

Merge Sort Analysis

Number of Compares: $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n - 1$

(in the worst case)

time to sort an array of size *n*

Merge Sort Analysis

Number of Compares: (in the worst case)

$$T(n) \quad = \quad T(\lceil \tfrac{n}{2} \rceil) \quad + \quad T(\lfloor \tfrac{n}{2} \rfloor) \quad + \quad n - 1$$

time to sort an array of size *n*

time to sort the *left* half

Merge Sort Analysis

Number of Compares:
(in the worst case)

$$T(n) \quad = \quad T(\lceil \tfrac{n}{2} \rceil) \quad + \quad T(\lfloor \tfrac{n}{2} \rfloor) \quad + \quad n-1$$

time to sort an
array of size *n*

time to sort
the *left* half

time to sort
the *right* half

Merge Sort Analysis

Number of Compares:
(in the worst case)

$$\underbrace{T(n)}_{\text{time to sort an array of size } n} \quad = \quad \underbrace{T(\lceil \tfrac{n}{2} \rceil)}_{\text{time to sort the } \textit{left} \text{ half}} \quad + \quad \underbrace{T(\lfloor \tfrac{n}{2} \rfloor)}_{\text{time to sort the } \textit{right} \text{ half}} \quad + \quad \underbrace{n - 1}_{\text{time to } \textit{merge} \text{ two sorted arrays of size } \frac{n}{2} \text{ each}}$$

Merge Sort Analysis

Number of Compares:
(in the worst case)

$$\begin{aligned} T(n) &= T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + n - 1 && \text{if } n > 1 \\ &= 0 && \text{if } n \leq 1 \end{aligned}$$

Merge Sort Analysis

Number of Compares:
(in the worst case)

$$\begin{aligned} T(n) &= T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + n - 1 && \text{if } n > 1 \\ &= 0 && \text{if } n \leq 1 \end{aligned}$$

For simplicity. We will assume that the array size is a power of two and that the worst case number of compares to merge the two sorted halves = n:

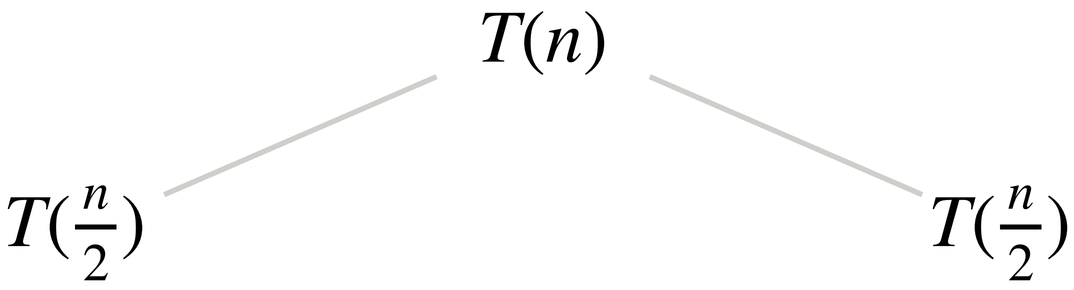
$$T(n) = \begin{cases} 2T(\tfrac{n}{2}) + n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$

(These assumptions do not affect the correctness of the analysis)

Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } \; n > 1 \\ 0 & \text{if } \; n \leq 1 \end{cases}$$

Recursion Tree


$T(n)$

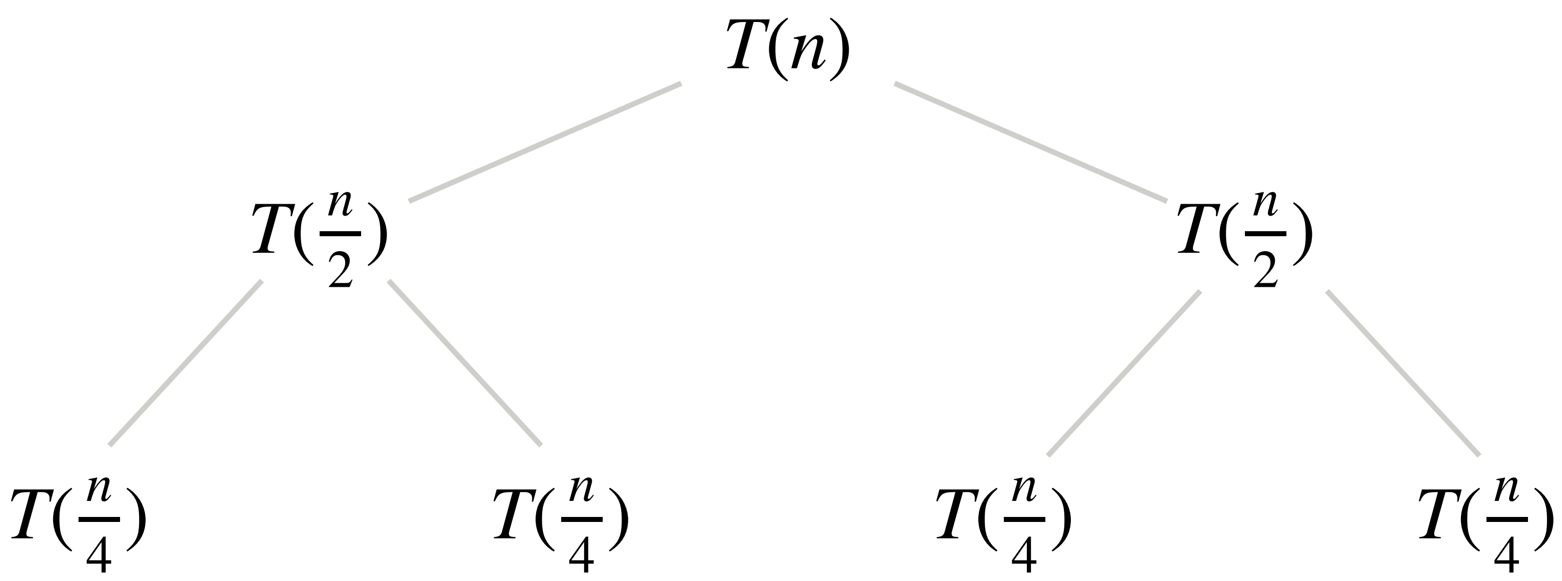
$T(\frac{n}{2})$

$T(\frac{n}{2})$

Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$

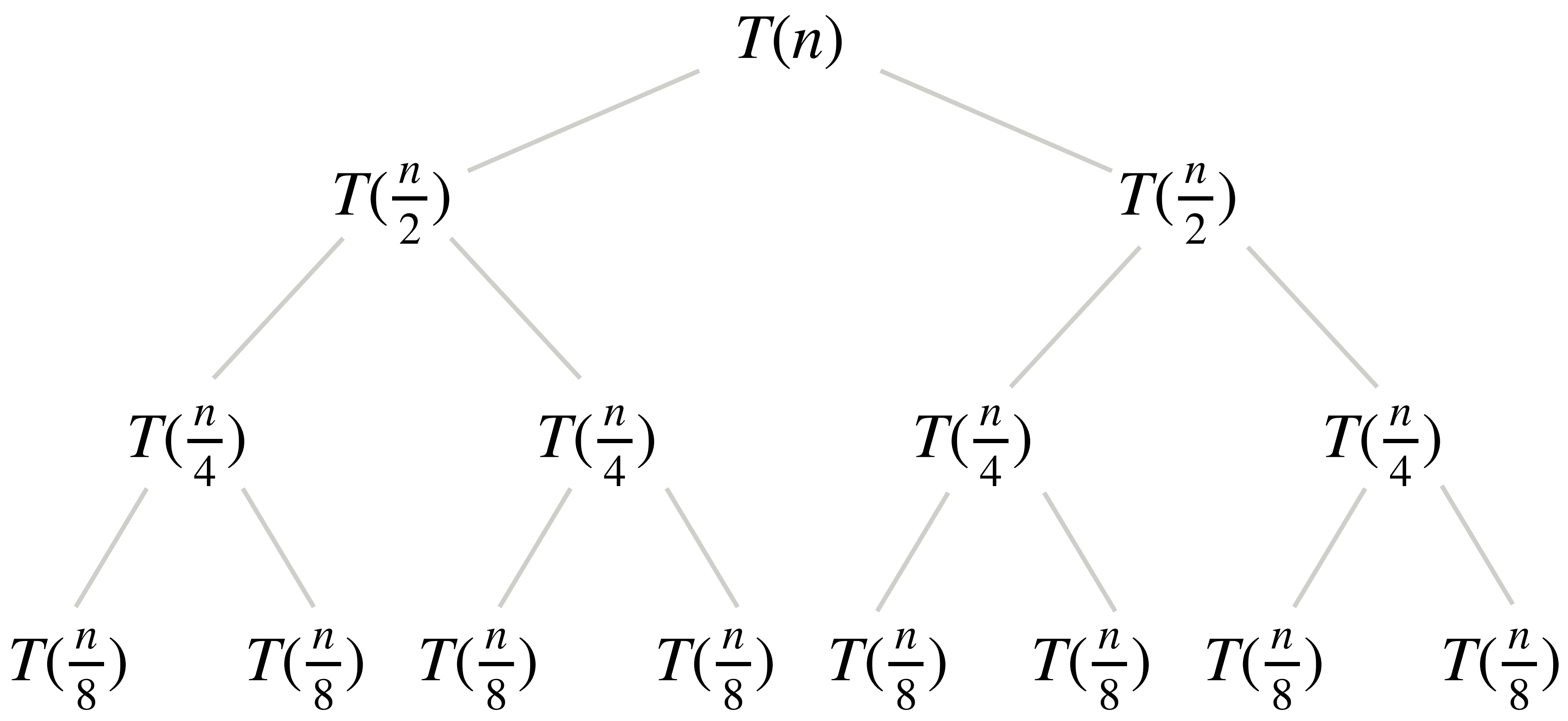
Recursion Tree



Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$

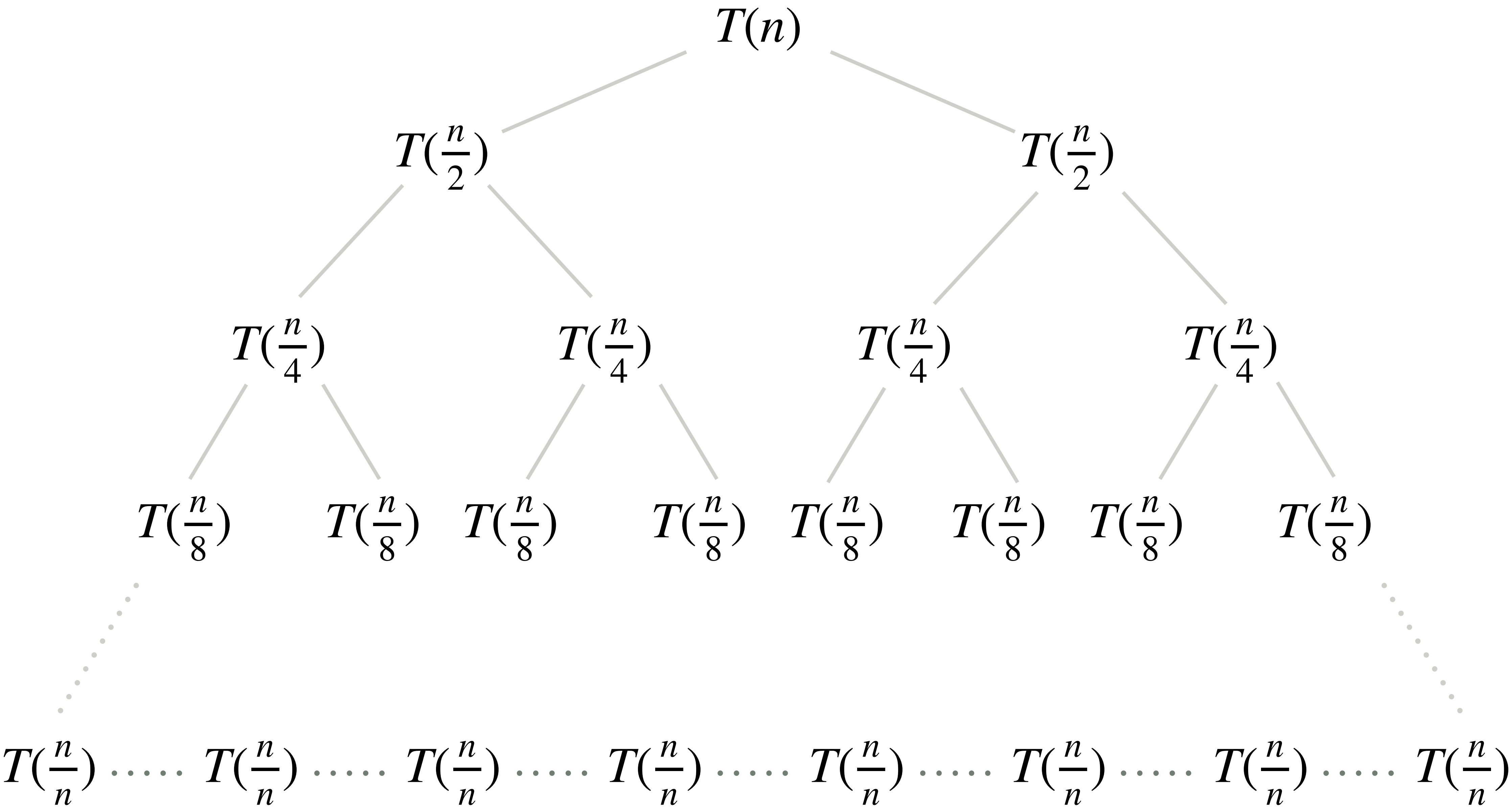
Recursion Tree



Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } \ n > 1 \\ 0 & \text{if } \ n \le 1 \end{cases}$$

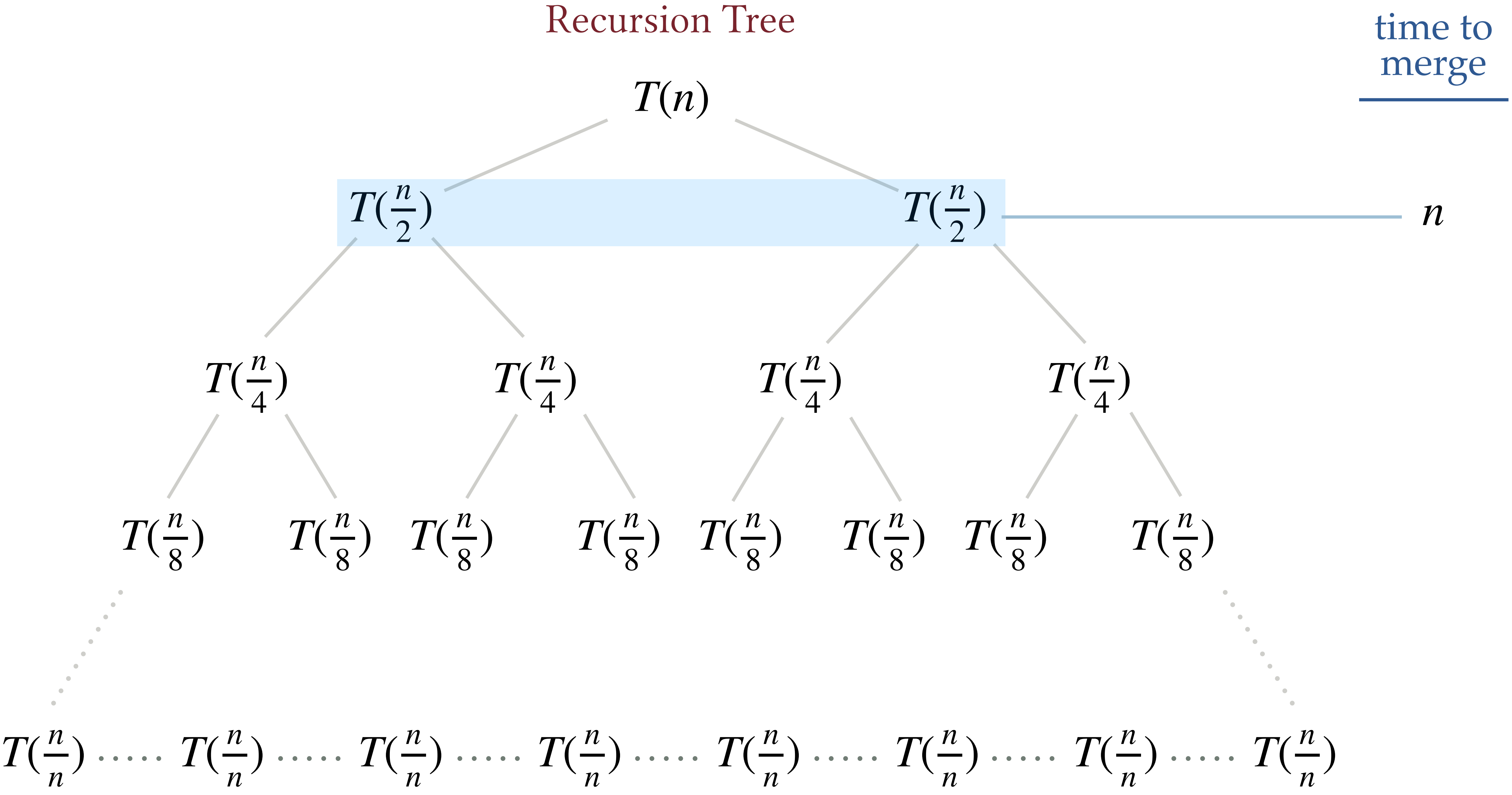
Recursion Tree



$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
$T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n}) \cdots\cdots T(\frac{n}{n})$

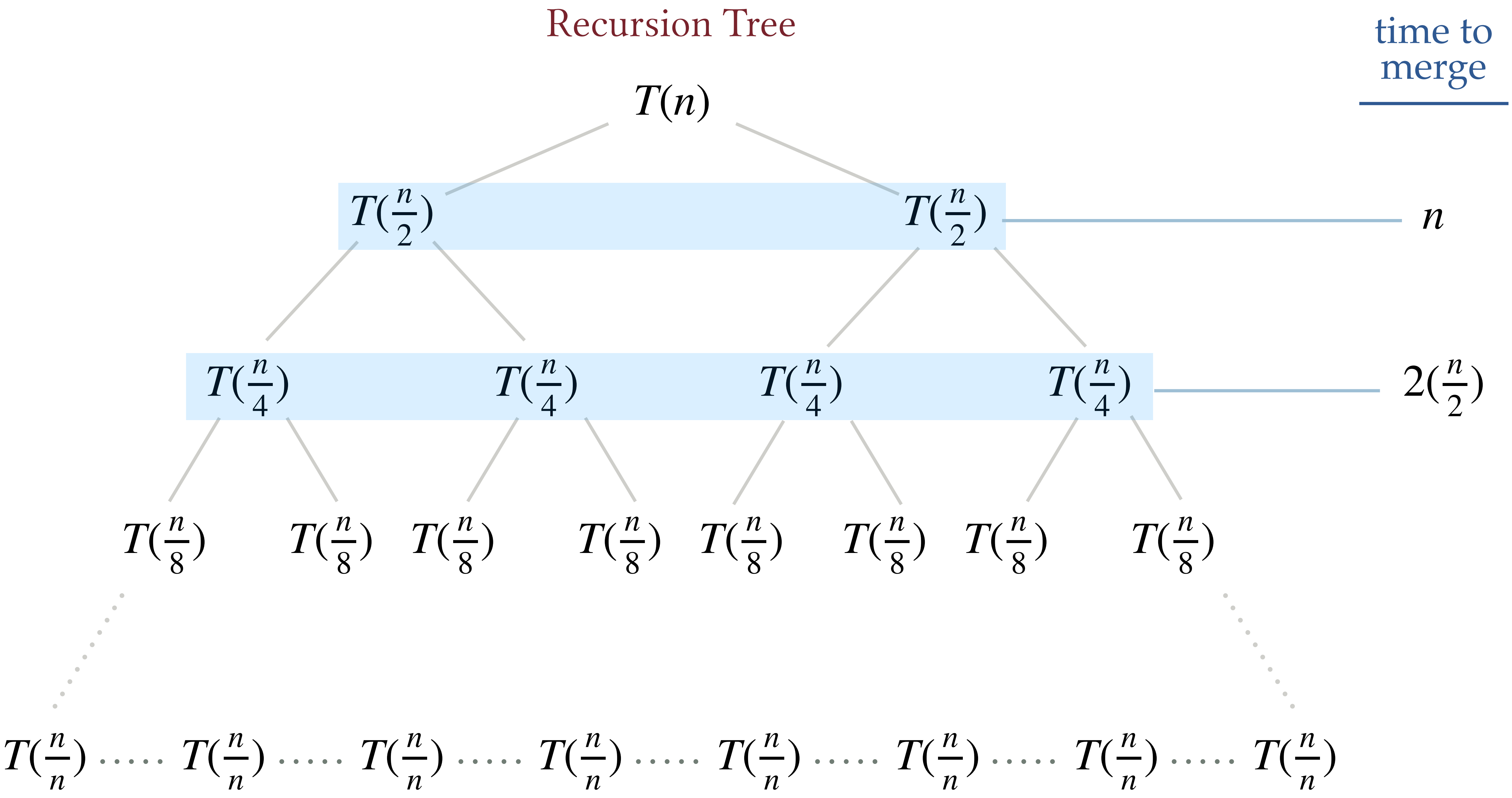
Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } n > 1 \\ 0 & \text{if } n \le 1 \end{cases}$$



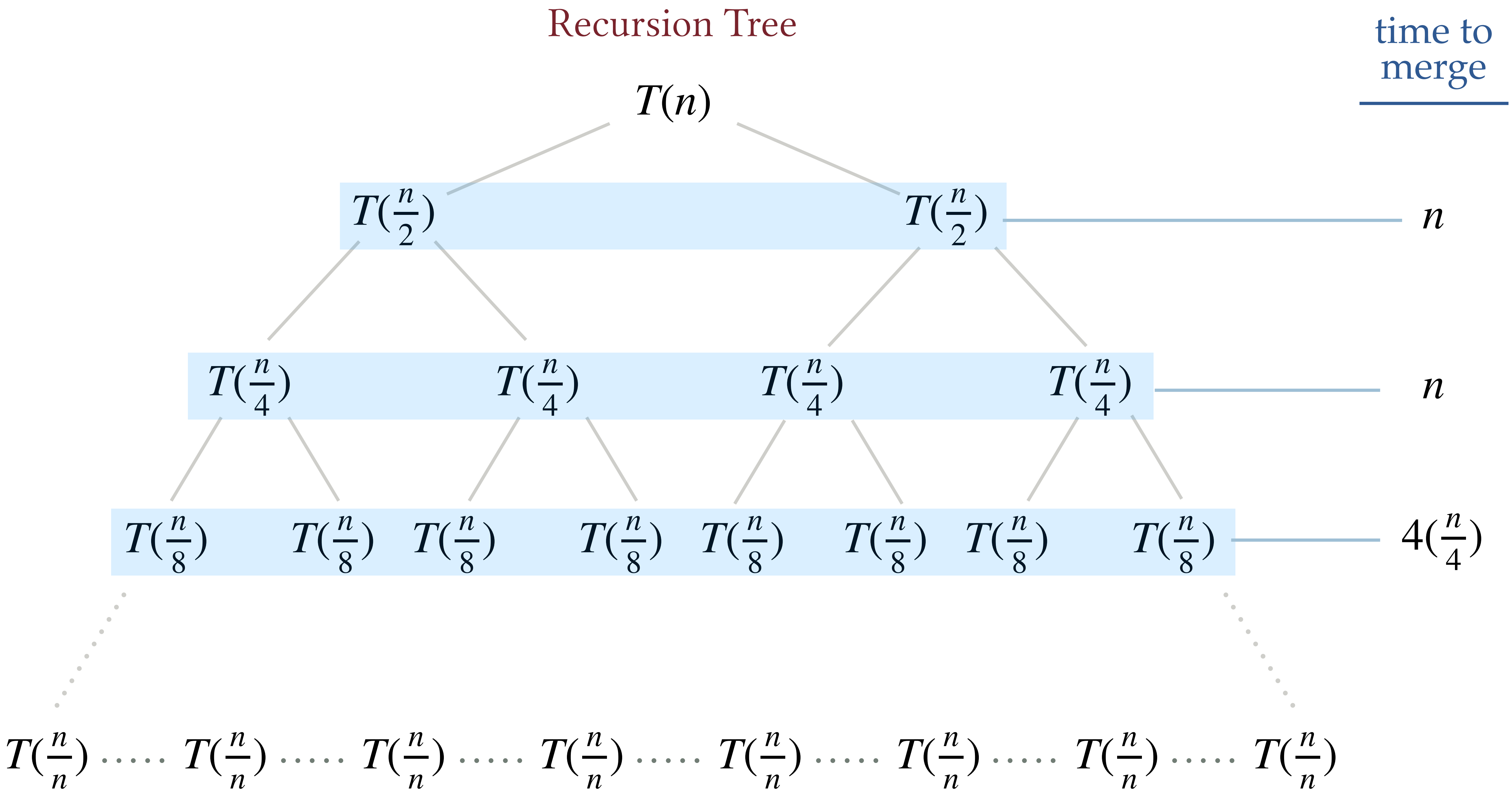
Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } \ n > 1 \\ 0 & \text{if } \ n \le 1 \end{cases}$$


Recursion Tree
time to merge
$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
n
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
$2(\frac{n}{2})$
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
$T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$

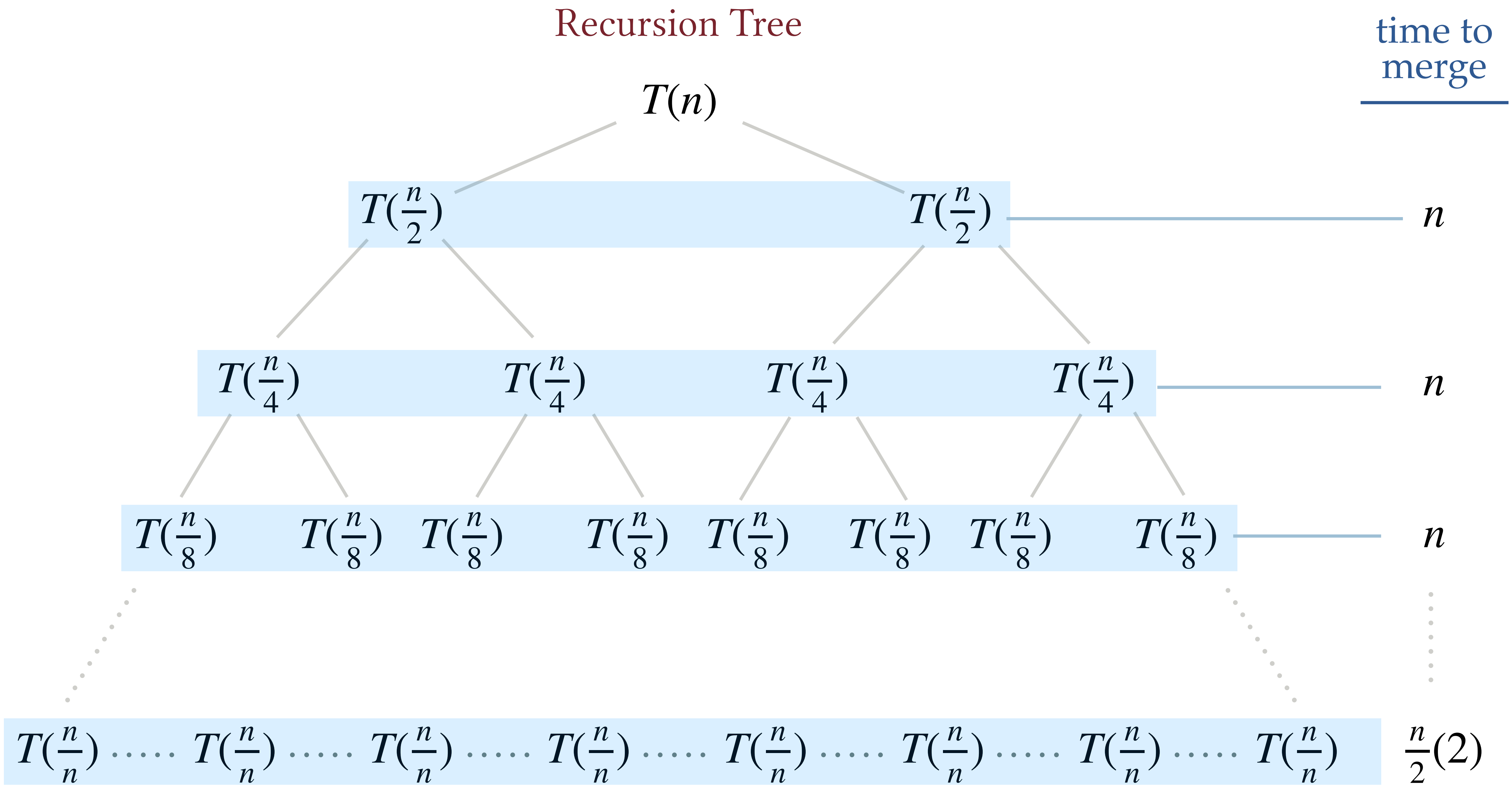
Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } \; n > 1 \\ 0 & \text{if } \; n \leq 1 \end{cases}$$


Recursion Tree
time to merge
$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
n
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
n
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
$4(\frac{n}{4})$
$T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$ ····· $T(\frac{n}{n})$

Merge Sort Analysis

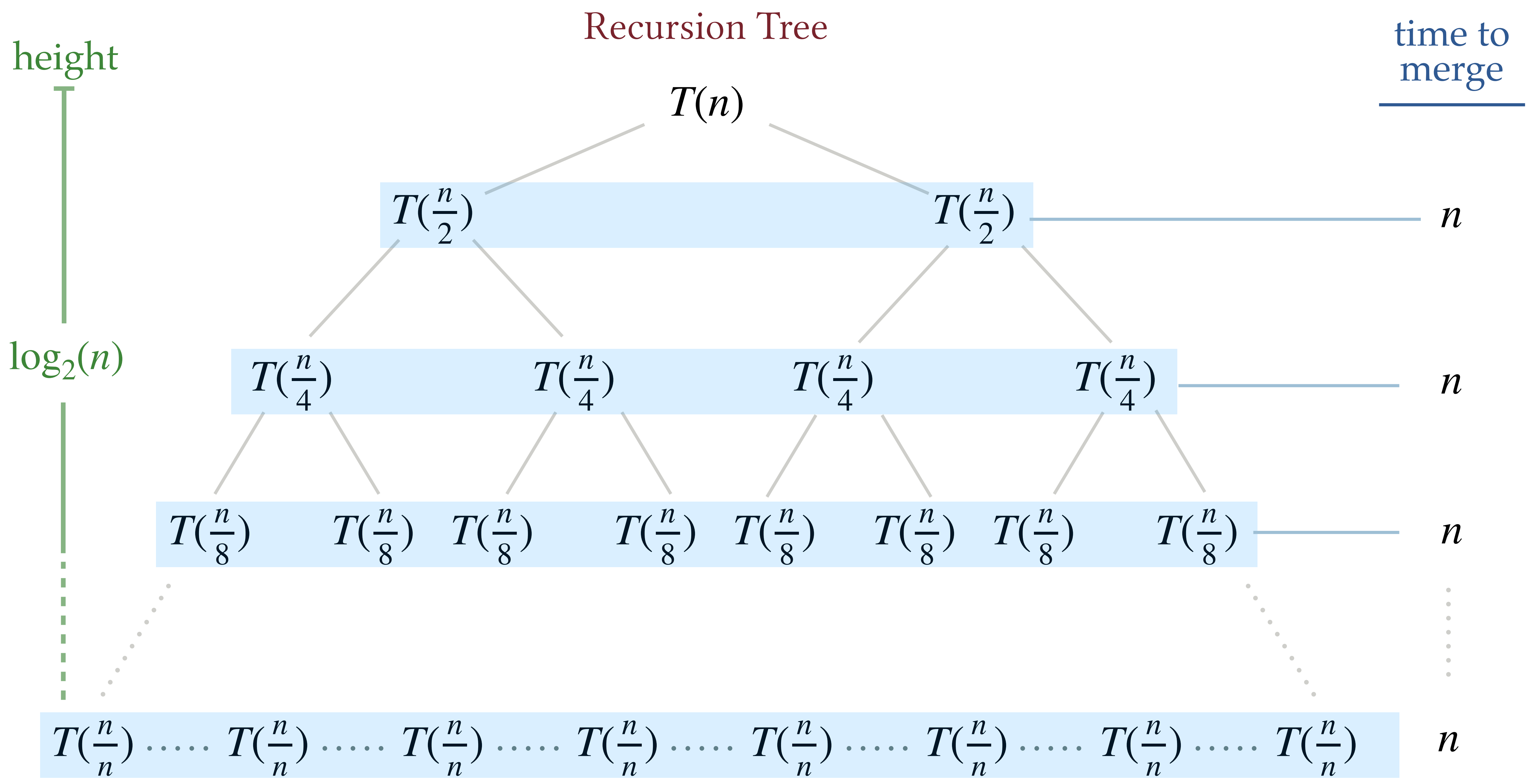
$$T(n) = \begin{cases} 2T(\frac{n}{2}) \quad + \quad n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$



Recursion Tree
time to merge
$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
n
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
n
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
n
$T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$
$\frac{n}{2}(2)$

Merge Sort Analysis

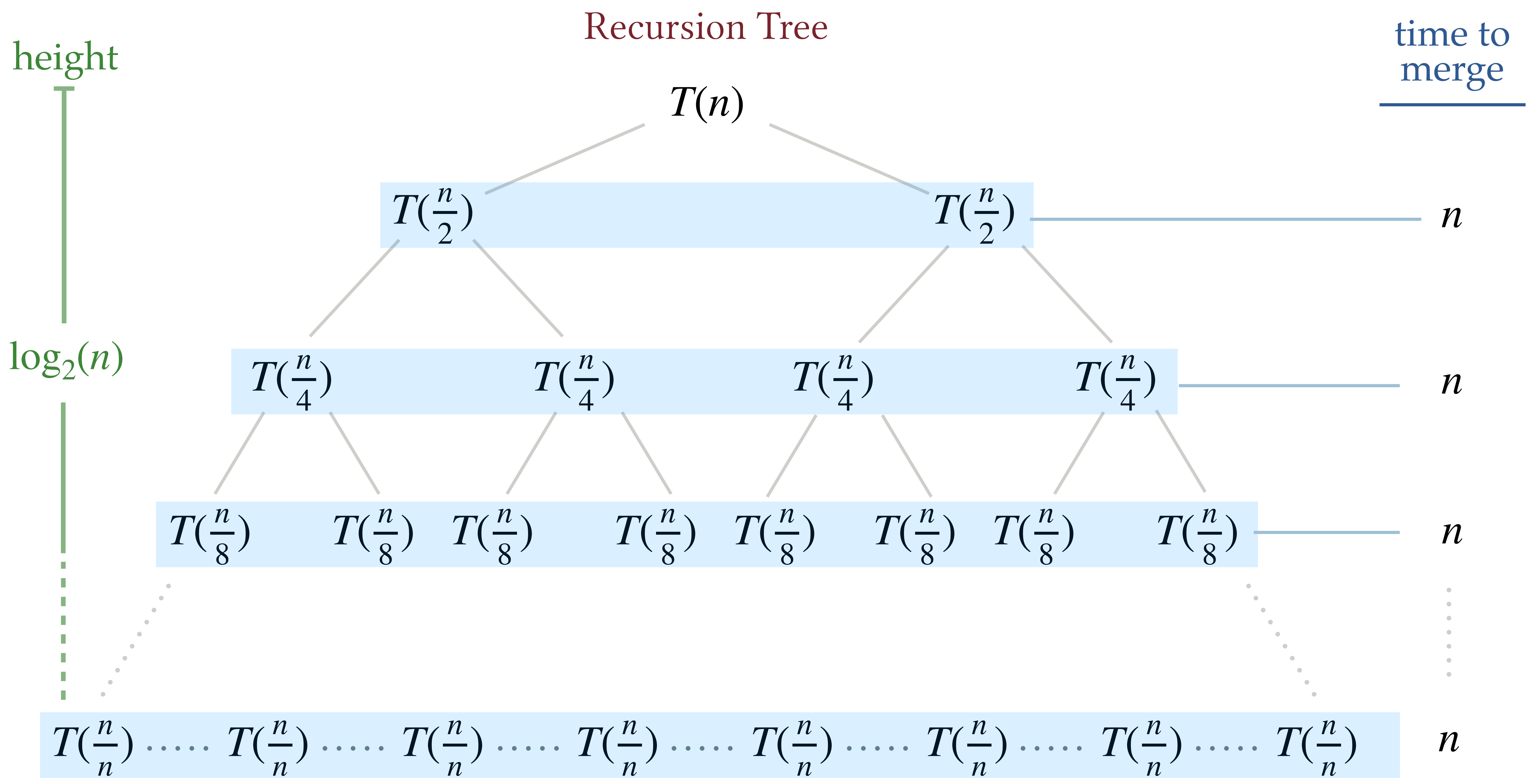
$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$


Recursion Tree
height
$\log_2(n)$
time to merge
$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
n
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
n
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
n
$T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$
n

Merge Sort Analysis

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$

Total Time $= n \log_2(n)$



Recursion Tree
height
$\log_2(n)$
time to merge
$T(n)$
$T(\frac{n}{2})$ $T(\frac{n}{2})$
n
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
n
$T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$
n
$T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$ ..... $T(\frac{n}{n})$
n

Merge Sort Analysis

Data Compares:

- $\sim n \log_2(n)$ in the worst case
- $\sim \frac{1}{2} n \log_2(n)$ in the best case

Data Moves: $\sim 2n \log_2(n)$ in the best, worst and average case

Total amount of work: $\Theta(n \log n)$ in the best case, worst case and average case.

Merge Sort Analysis

Data Compares:

- $\sim n\log_2(n)$ in the worst case
- $\sim \frac{1}{2}n\log_2(n)$ in the best case

make sure you understand why and can do the analysis!

Data Moves: $\sim 2n\log_2(n)$ in the best, worst and average case

Total amount of work: $\Theta(n\log n)$ in the best case, worst case and average case.

Merge Sort Analysis

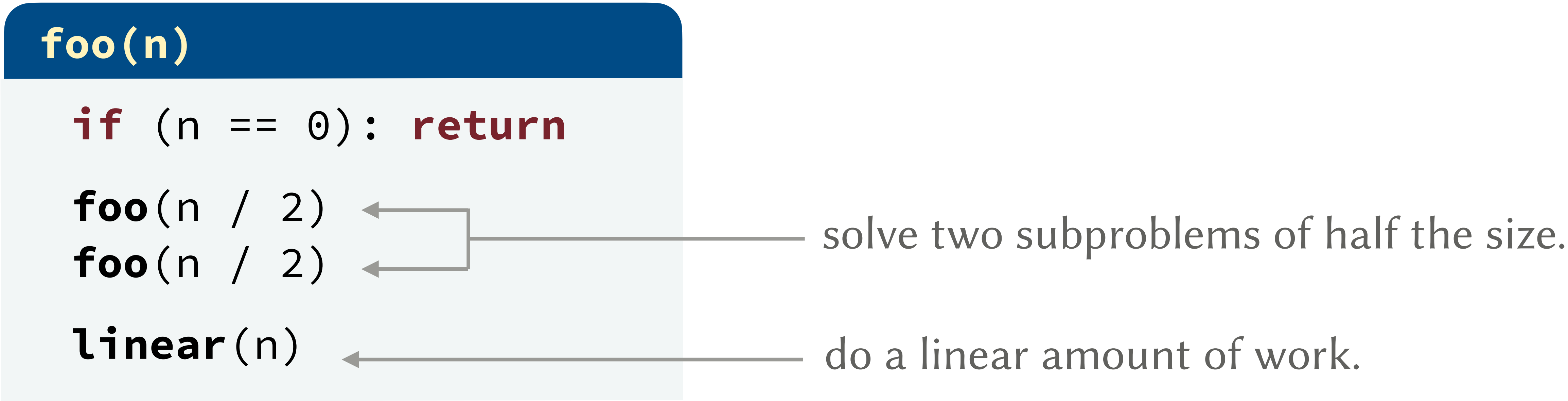
Data Compares:

- $\sim n\log_2(n)$ in the worst case
- $\sim \frac{1}{2}n\log_2(n)$ in the best case

Data Moves: $\sim 2n\log_2(n)$ in the best, worst and average case

Total amount of work: $\Theta(n\log n)$ in the best case, worst case and average case.

Generally: Code that follows the pattern below has a running time of $\Theta(n\log n)$



```
foo(n)
  if (n == 0): return

  foo(n / 2)
  foo(n / 2)

  linear(n)
```

solve two subproblems of half the size.

do a linear amount of work.

Empirical Analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (n^2)			mergesort ($n \log n$)		
computer	**thousand**	**million**	**billion**	**thousand**	**million**	**billion**
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

By Kevin Wayne

Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
(when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

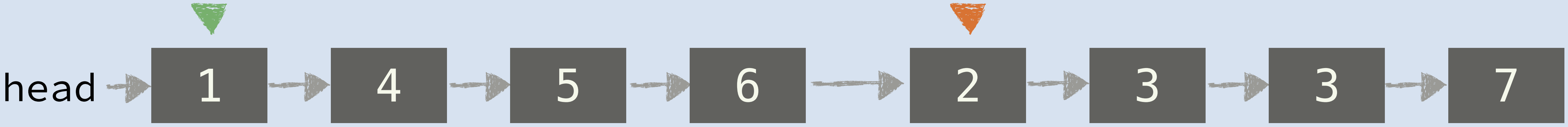
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

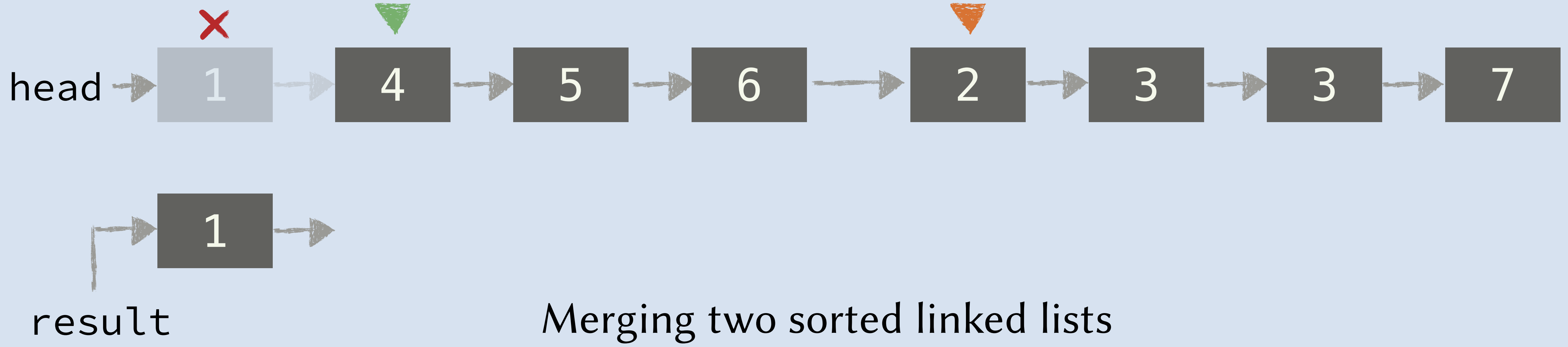
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation. (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

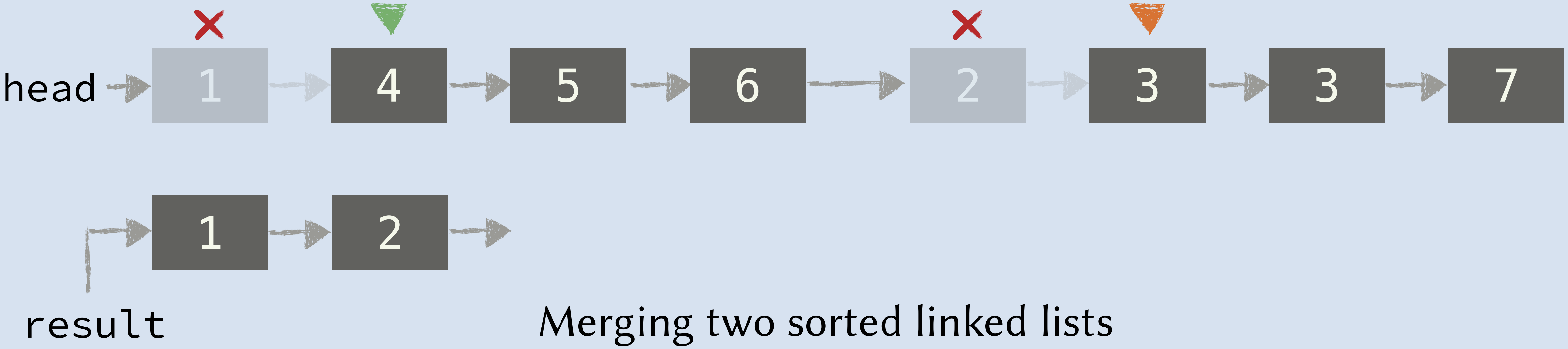
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

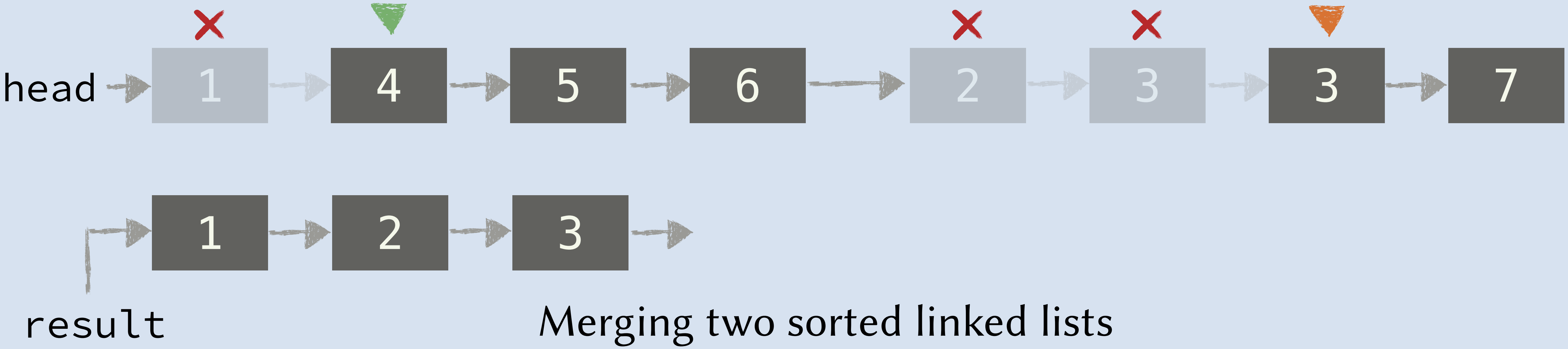
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

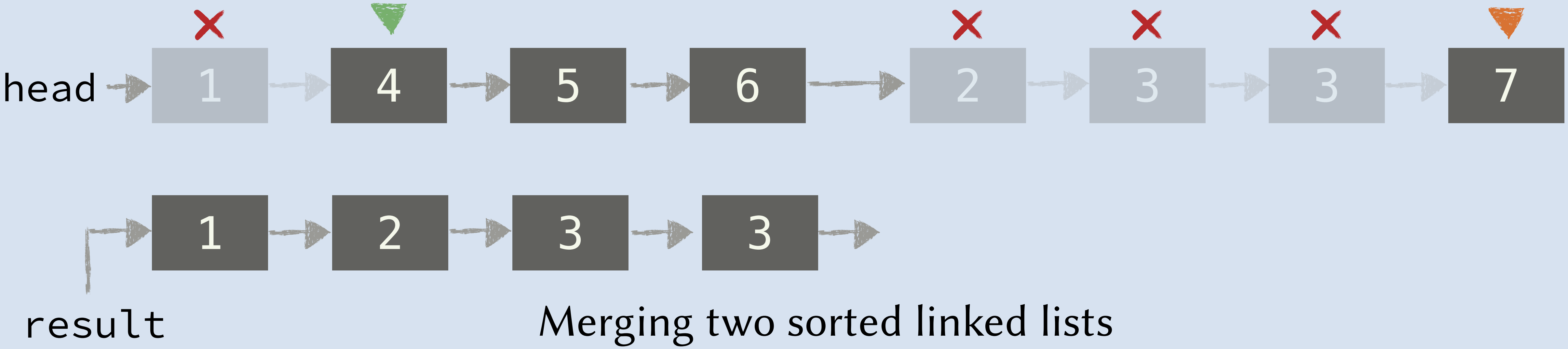
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

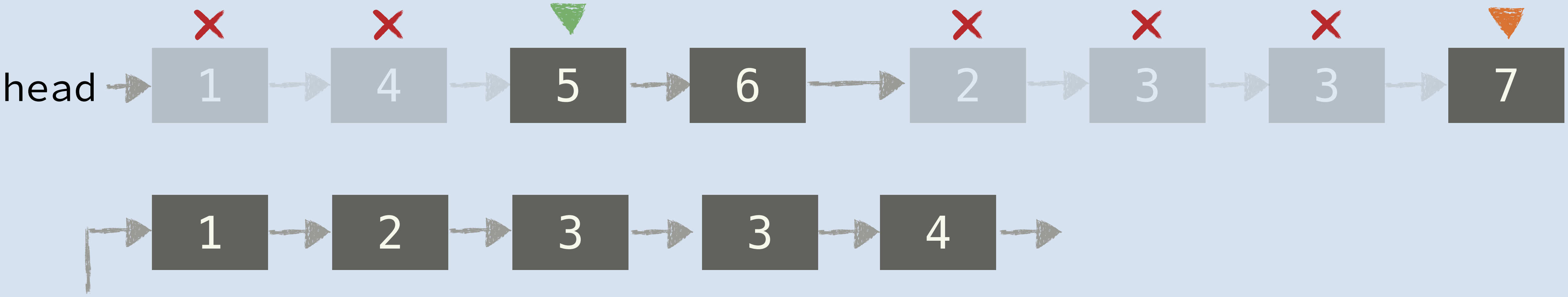
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

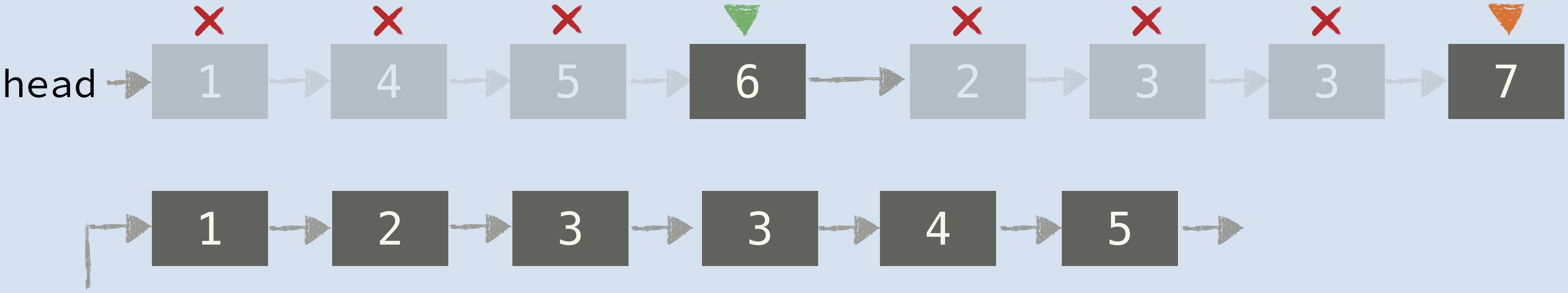
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

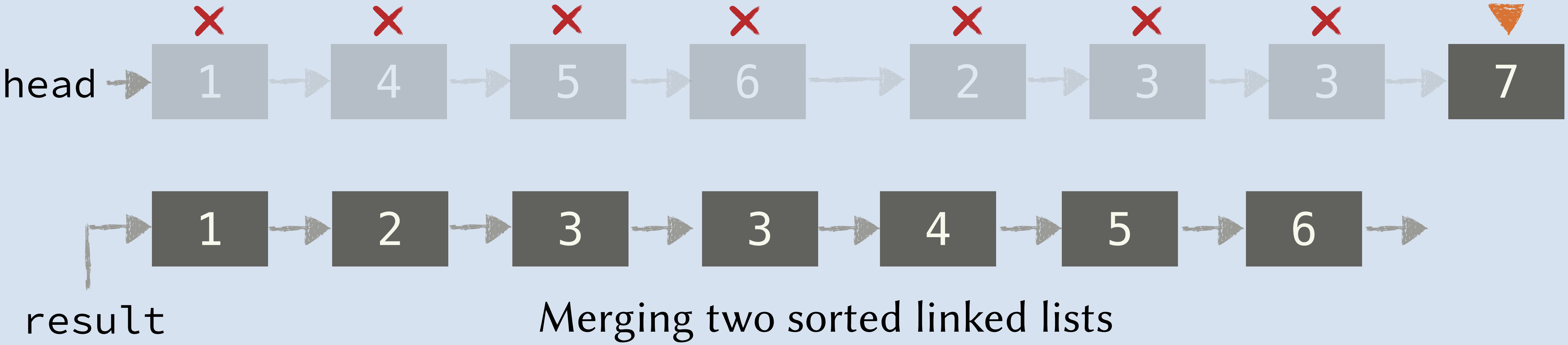
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation. (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

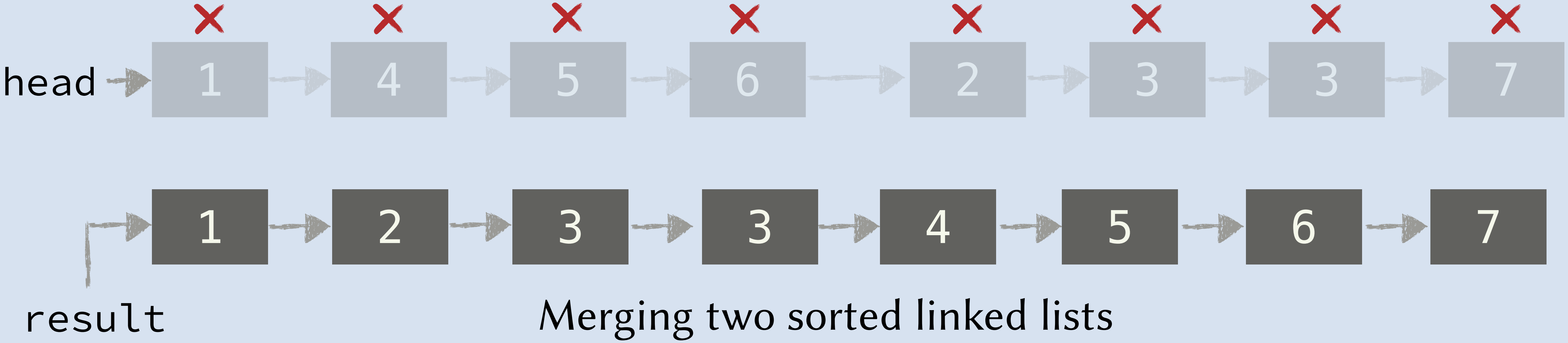
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation. (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

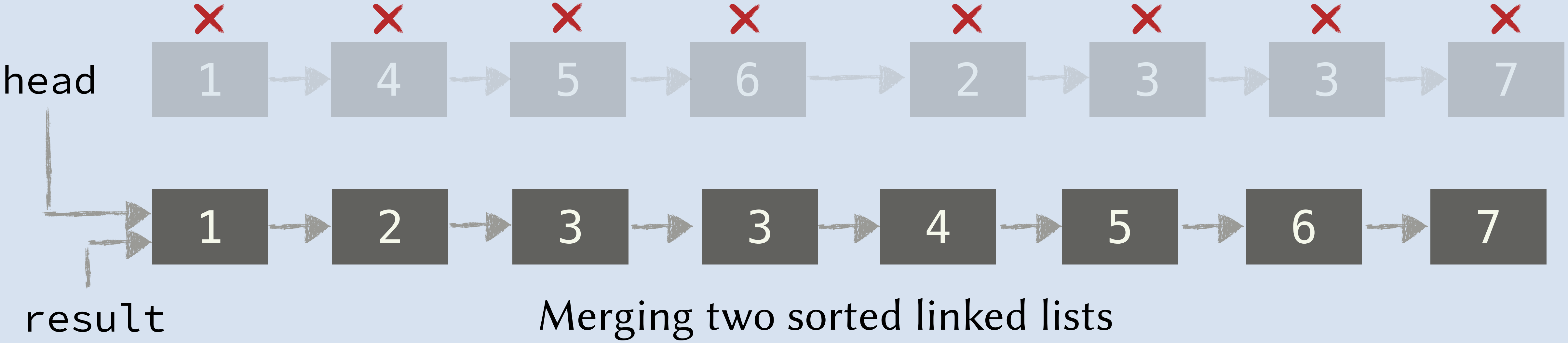
Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.



Merging two sorted linked lists

Merge Sort Analysis

Definition. A sorting algorithm is **in-place** if it uses $O(\log n)$ extra space.

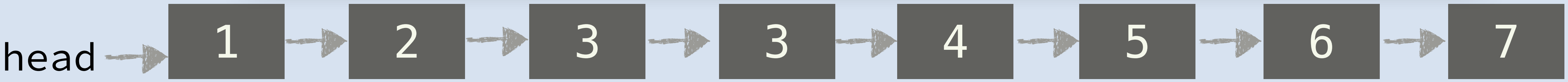
Memory Analysis. Merge Sort is not in-place:

- It requires $\Theta(n)$ extra space for the merge operation.
  (when merging the two halves of size $\frac{n}{2}$ each)
- It also requires $\Theta(\log n)$ extra memory for the recursion stack.
- An in-place implementation of Merge Sort is possible but difficult and probably not worth it!

Sorting a Linked List. Can be done using Merge Sort in $\Theta(n \log n)$ time and using $O(\log n)$ extra memory.

💪 Extra

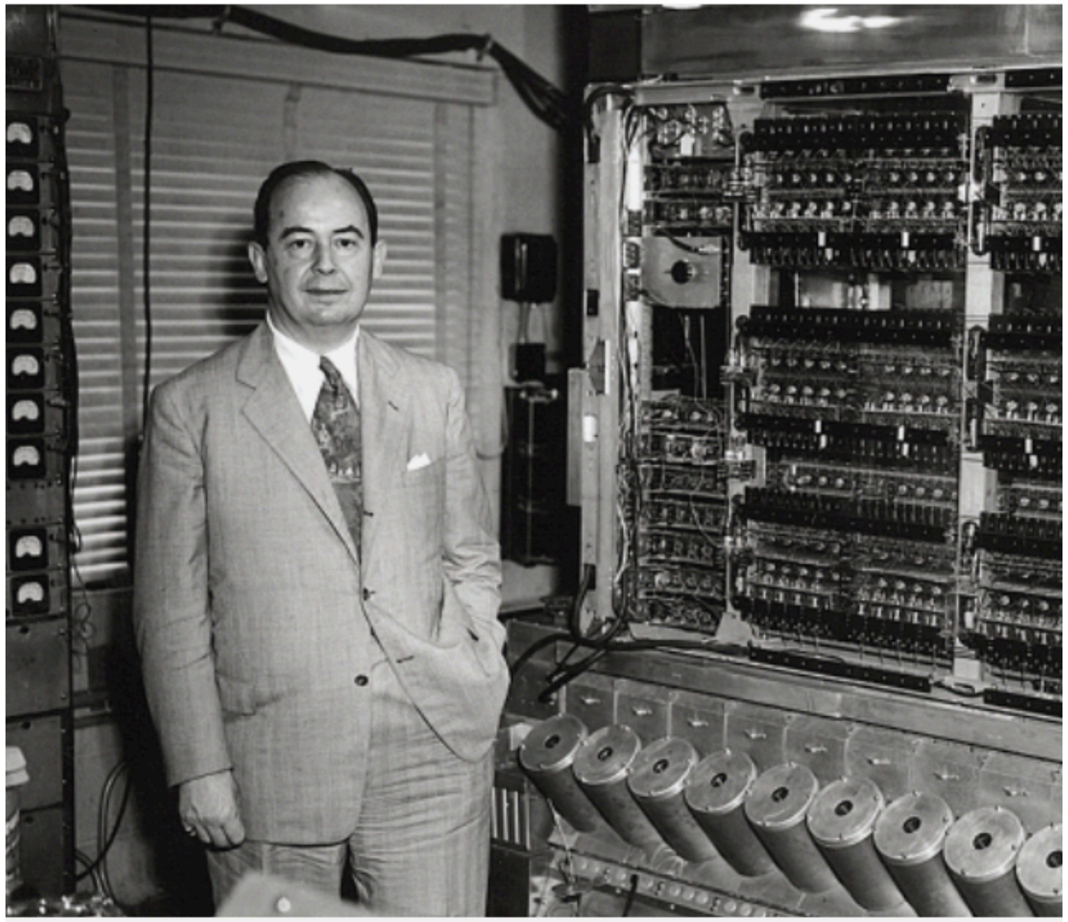
write a C++ program that merges two sorted linked lists without allocating any new node or deleting any node.



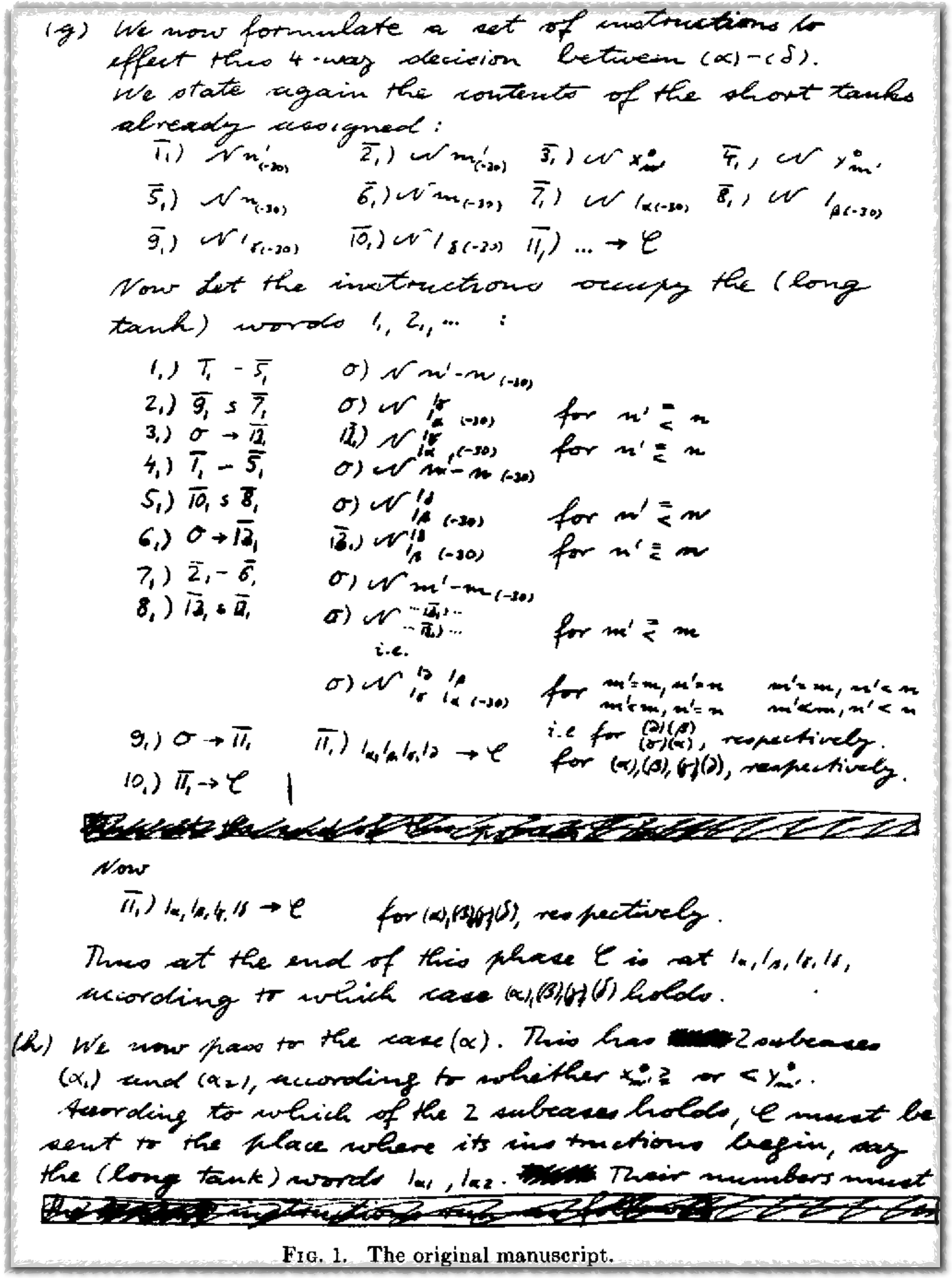
Merging two sorted linked lists

Merge Sort History

Introduced by John von Neumann in 1948 as an example of the algorithms that could be executed on his newly designed machine (EDVAC)



The image to the right is John von Neumann's handwritten code of merge sort in the manuscript titled "A First Draft of a Report on the EDVAC" (as reported by Knuth in the 1970 report titled "Von Neumann's First Computer Program")



(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:

$\bar{1}_1$) $N n'_{(-30)}$ $\bar{2}_1$) $w\, m'_{(-30)}$ $\bar{3}_1$) $w\, x^0_{m'}$ $\bar{4}_1$) $w\, y^0_{n'}$

$\bar{5}_1$) $N n_{(-30)}$ $\bar{6}_1$) $N m_{(-30)}$ $\bar{7}_1$) $w\, l_{\alpha(-30)}$ $\bar{8}_1$) $w\, l_{\beta(-30)}$

$\bar{9}_1$) $N l_{\gamma(-30)}$ $\overline{10}_1$) $w\, l_{\delta(-30)}$ $\overline{11}_1$) $\ldots \to \mathcal{C}$

Now let the instructions occupy the (long tank) words $1_1, 2_1, \ldots$:

1_1) $\bar{1}_1 - \bar{5}_1$ σ) $N\, n'-n_{(-30)}$

2_1) $\bar{9}_1\ s\ \bar{7}_1$ σ) $w \substack{l_\gamma \\ l_\alpha}{}_{(-30)}$ for $n' \substack{\geq \\ <} n$

3_1) $\sigma \to \overline{12}_1$ $\overline{12}_1$) $N \substack{l_\gamma \\ l_\alpha}{}_{(-30)}$ for $n' \substack{\geq \\ <} n$

4_1) $\bar{1}_1 - \bar{5}_1$ σ) $w\, m'-m_{(-30)}$

5_1) $\overline{10}_1\ s\ \bar{8}_1$ σ) $w \substack{l_\delta \\ l_\beta}{}_{(-30)}$ for $m' \substack{\geq \\ <} m$

6_1) $\sigma \to \overline{13}_1$ $\overline{13}_1$) $w \substack{l_\delta \\ l_\beta}{}_{(-30)}$ for $n' \substack{\geq \\ <} m$

7_1) $\bar{2}_1 - \bar{6}_1$ σ) $N\, m'-m_{(-30)}$

8_1) $\overline{13}_1\ s\ \overline{12}_1$ σ) $N \substack{\cdots \overline{13}_1 \cdots \\ \cdots \overline{12}_1 \cdots}$ for $m' \substack{\geq \\ <} m$

i.e.

σ) $N \substack{l_\delta \\ l_\gamma}\ \substack{l_\beta \\ l_\alpha}{}_{(-30)}$ for $\substack{m' \geq m,\ n' \geq n \\ m' < m,\ n' \geq n}$ $\substack{m' \geq m,\ n' < n \\ m' < m,\ n' < n}$

i.e. for $\substack{(\delta)(\beta) \\ (\gamma)(\alpha)}$, respectively.

9_1) $\sigma \to \overline{11}_1$ $\overline{11}_1$) $l_\alpha, l_\beta, l_\gamma, l_\delta \to \mathcal{C}$ for (α),(β),(γ)(δ), respectively.

10_1) $\overline{11}_1 \to \mathcal{C}$

Now

$\overline{11}_1$) $l_\alpha, l_\beta, l_\gamma, l_\delta \to \mathcal{C}$ for (α),(β),(γ)(δ), respectively.

Thus at the end of this phase $\mathcal{C}$ is at $l_\alpha, l_\beta, l_\gamma, l_\delta$, according to which case (α),(β),(γ)(δ) holds.

(h) We now pass to the case (α). This has 2 subcases (α_1) and (α_2), according to whether $x^0_{m'} \geq$ or $< y^0_{n'}$. According to which of the 2 subcases holds, $\mathcal{C}$ must be sent to the place where its instructions begin, say the (long tank) words $l_{\alpha 1}, l_{\alpha 2}$. Their numbers must

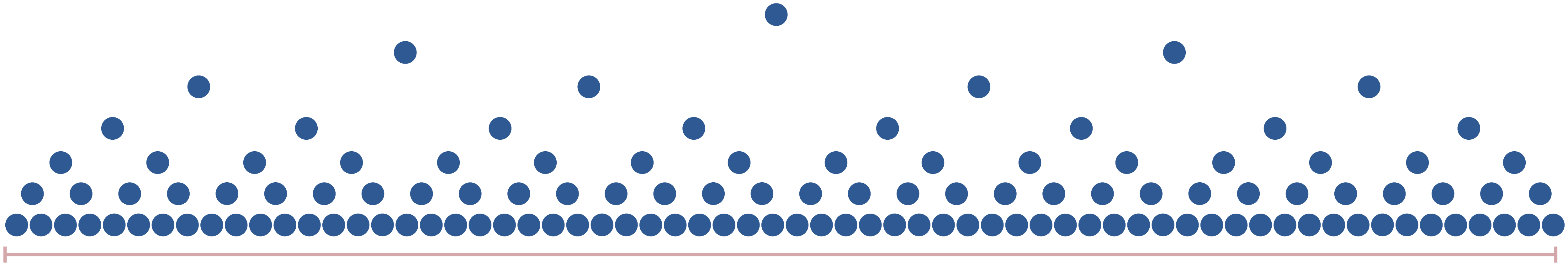
FIG. 1. The original manuscript.

Optimizations

Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

```
MERGE-SORT(a[], first, last)

if last - first + 1 <= CUTOFF:
    insertion-Sort(a, first, last)
    return

mid = first + (last - first) / 2
MERGE-SORT(a, first, mid)
MERGE-SORT(a, mid + 1, last)

MERGE(a, first, mid, last)
```



Too many recursive calls at the leafs

Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

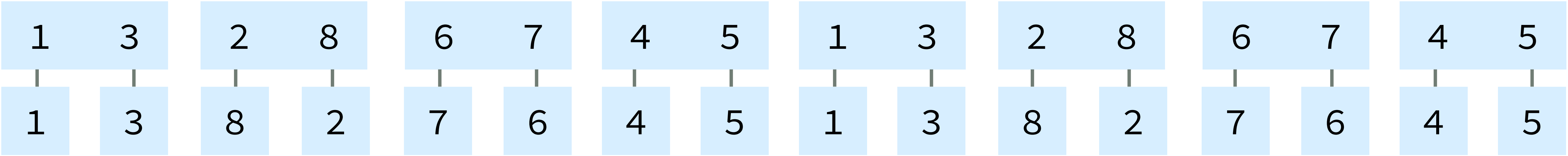
1	3	8	2	7	6	4	5	1	3	8	2	7	6	4	5

Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.

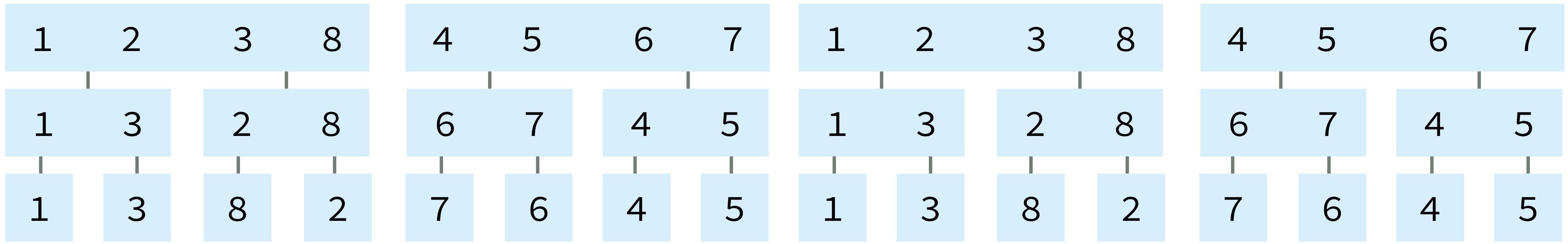


Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.

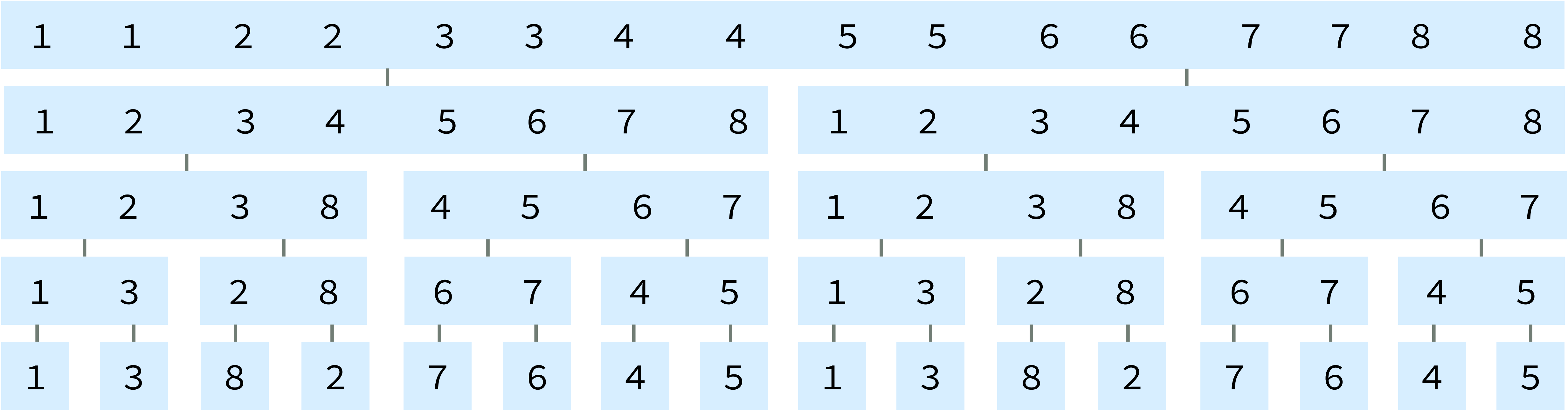


Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.



1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8

1 2 3 4 5 6 7 8 | 1 2 3 4 5 6 7 8

1 2 3 8 | 4 5 6 7 | 1 2 3 8 | 4 5 6 7

1 3 | 2 8 | 6 7 | 4 5 | 1 3 | 2 8 | 6 7 | 4 5

1 | 3 | 8 | 2 | 7 | 6 | 4 | 5 | 1 | 3 | 8 | 2 | 7 | 6 | 4 | 5

Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.

1	3	8	9	2	4	6	5	1	3	8	2	7	4	5	6

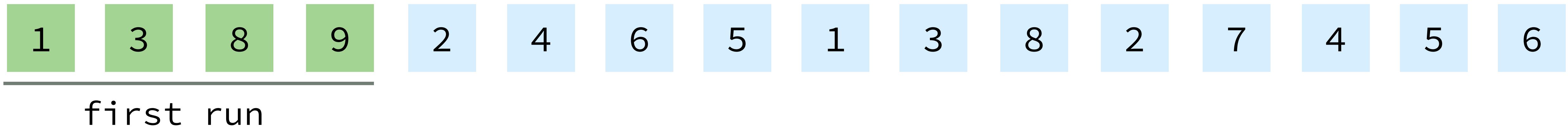
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



1	3	8	9	2	4	6	5	1	3	8	2	7	4	5	6

first run

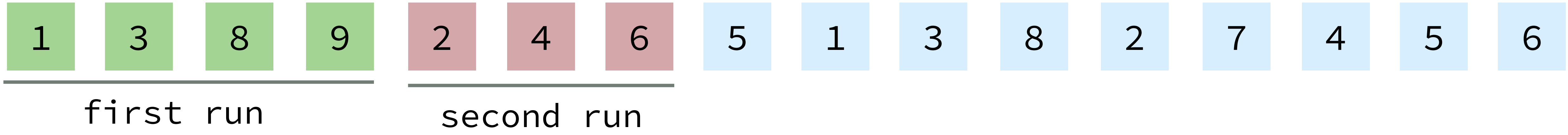
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



1	3	8	9	2	4	6	5	1	3	8	2	7	4	5	6

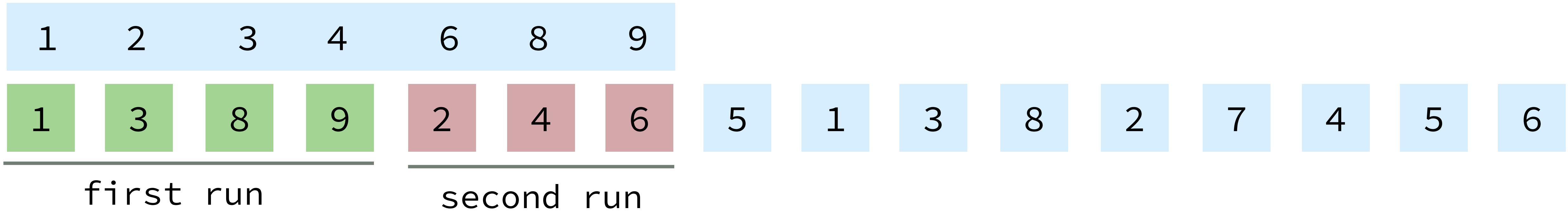
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



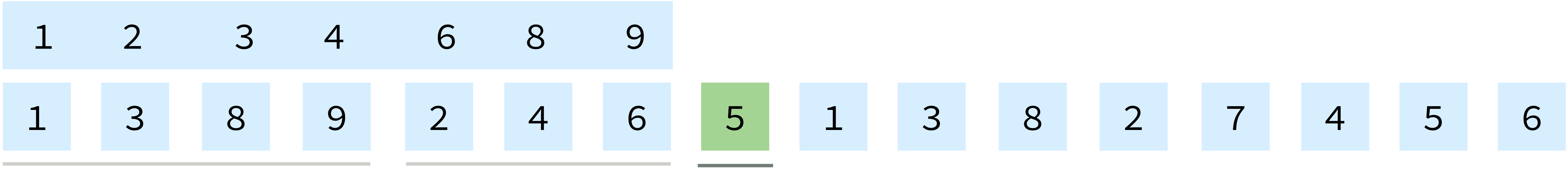
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



1 2 3 4 6 8 9

1 3 8 9 2 4 6 5 1 3 8 2 7 4 5 6

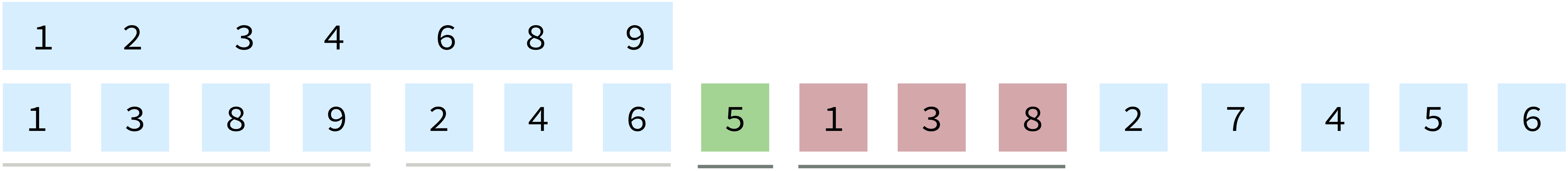
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



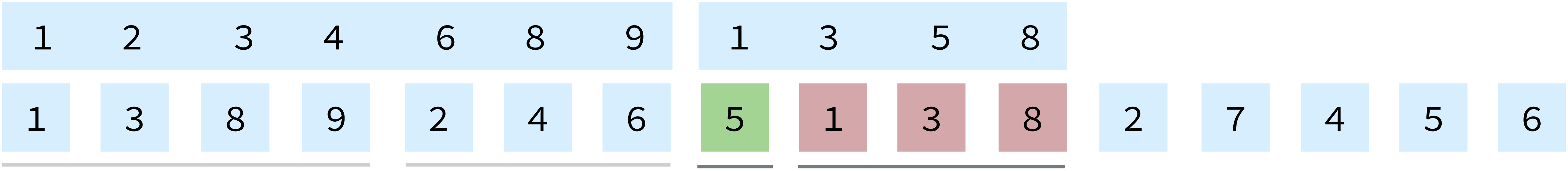
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



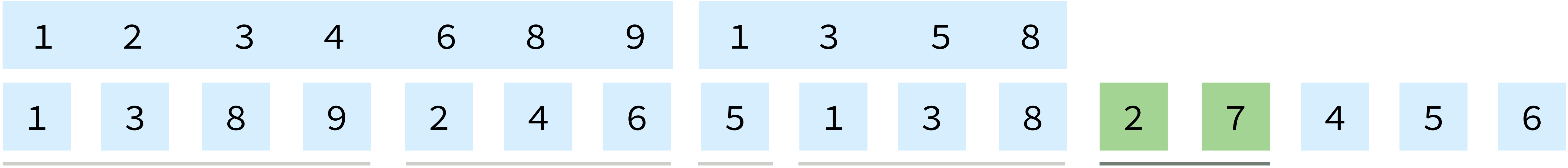
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



1 2 3 4 6 8 9 1 3 5 8

1 3 8 9 2 4 6 5 1 3 8 2 7 4 5 6

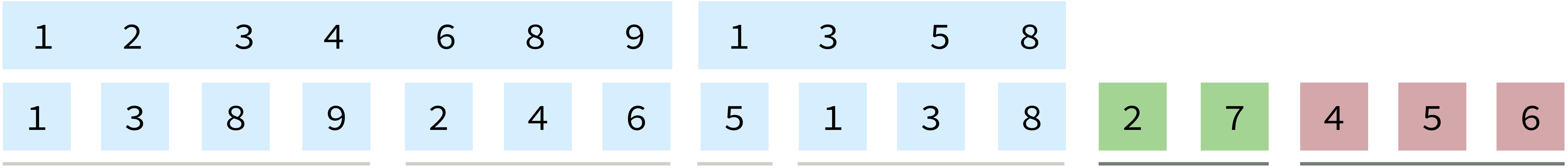
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



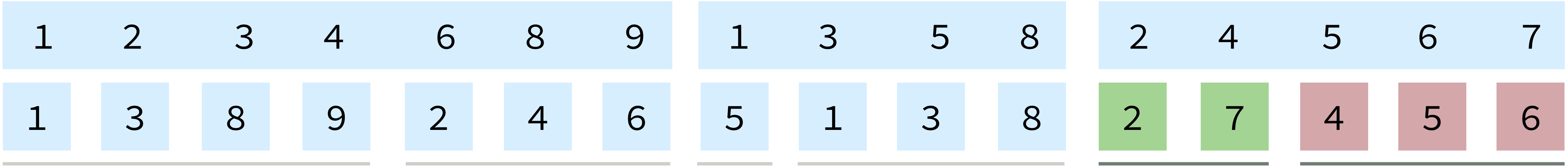
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



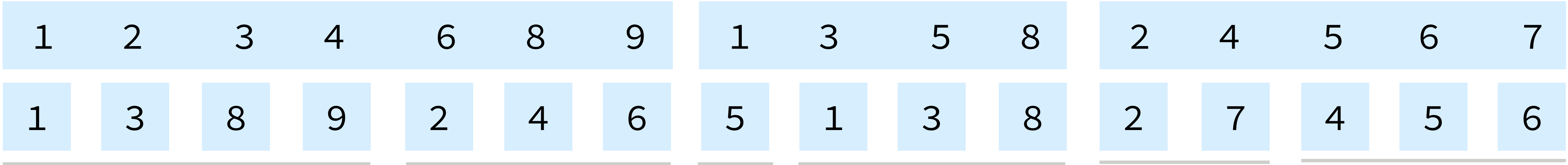
Optimizations

Use *insertion sort* for small Arrays. Avoids spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.

Avoid recursion altogether! (Bottom-up Merge Sort)

- Iteratively merge all subarrays of size 1, to get sorted arrays of size 2 each.
- Iteratively merge all the sorted arrays of size 2 to get sorted arrays of size 4.
- Repeat for arrays size 4, 8, 16, etc.

Exploit Natural Runs. Do not sort subarrays that are already sorted.



1 2 3 4 6 8 9 | 1 3 5 8 | 2 4 5 6 7

1 3 8 9 | 2 4 6 | 5 | 1 3 8 | 2 7 | 4 5 6

- Improves the performance in the best case.
- Performs well on partially sorted data and other special types of data.
- Requires extra data compares to identify the sorted runs.

Timsort

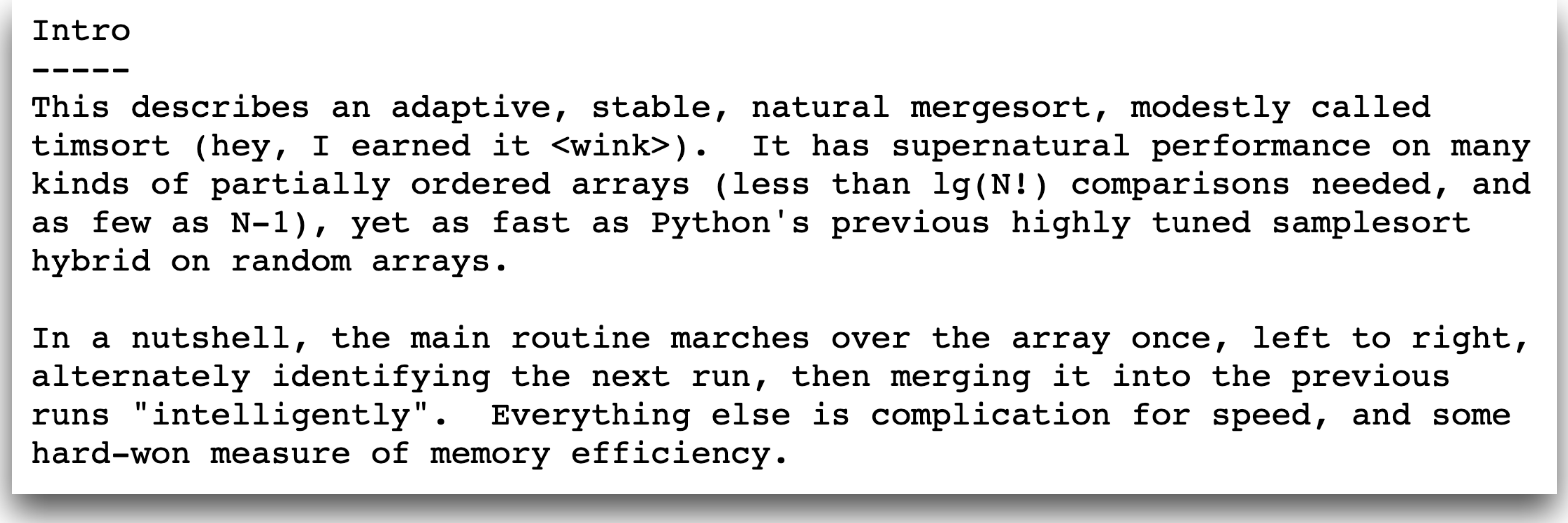
- Introduced by Tim Peters in 2002 for use in the Python Programming Language.

- Bottom-up Merge Sort that exploits natural runs, uses insertion sort in addition to other optimizations.

- Performs well on many kinds of real-world data.

- Very widely used.

python android Java V8 Rust swift octave



```
Intro
-----
This describes an adaptive, stable, natural mergesort, modestly called
timsort (hey, I earned it <wink>).  It has supernatural performance on many
kinds of partially ordered arrays (less than lg(N!) comparisons needed, and
as few as N-1), yet as fast as Python's previous highly tuned samplesort
hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right,
alternately identifying the next run, then merging it into the previous
runs "intelligently".  Everything else is complication for speed, and some
hard-won measure of memory efficiency.
```

https://bugs.python.org/file4451/timsort.txt

Analysis Question

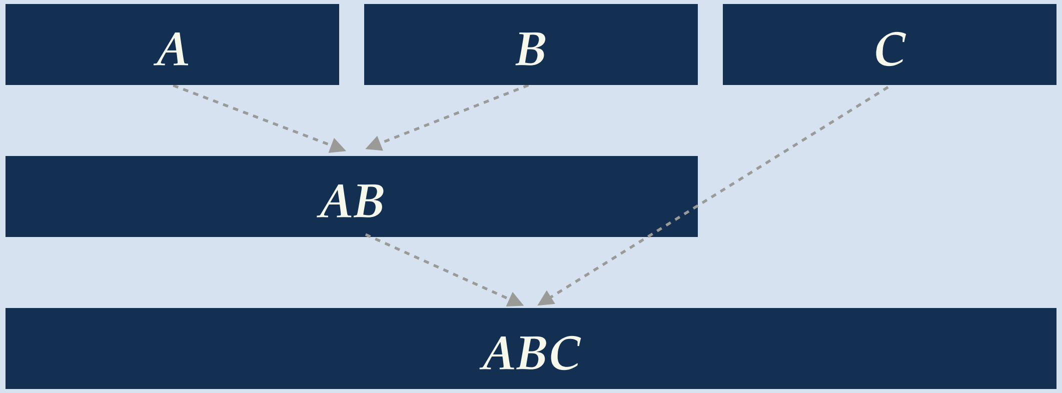
How can we *merge* three sorted arrays into one sorted array?

Analysis Question

How can we *merge* three sorted arrays into one sorted array?

Solution 1

Given three sorted arrays *A*, *B* and *C* of size $\frac{n}{3}$ each, merge *A* and *B* into a new array named *AB* and then merge *AB* and *C*.

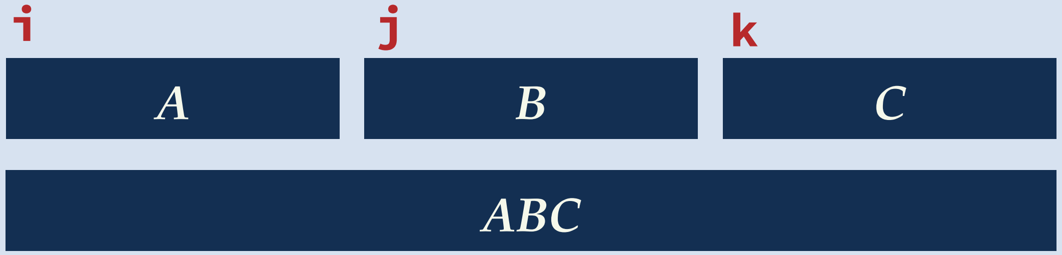


Worst case number of compares:

- To merge A and B: $\frac{1}{3}n+\frac{1}{3}n = \frac{2}{3}n$
- To merge AB and C: $\frac{2}{3}n+\frac{1}{3}n = n$
- Total = $\frac{2}{3}n+\frac{3}{3}n = \frac{5}{3}n$

Solution 2

Use three pointers to implement an algorithm similar to the one described before for merging two sorted arrays.



- Two comparisons are needed to find the minimum of three numbers.
- In the worst case, no array will be completely copied much earlier than the other two arrays.
- The total worst case number of compares $\sim 2n$

Analysis Question

Which requires less comparisons in the worst case: 2-way merge sort or 3-way merge sort?

Analysis Question

Which requires less comparisons in the worst case: 2-way merge sort or 3-way merge sort?

Solution.

- **2-way** merge sort requires $\sim n \log_2(n)$ compares in the worst case.
- **3-way** merge sort requires in the worst case:
  - If **solution 1** is used:

$$\sim \frac{5}{3} n \log_3(n) = \frac{5}{3} n \frac{\log_2(n)}{\log_2(3)} = 1.05 n \log_2(n)$$

  - If **solution 2** is used:

$$\sim 2n \log_3(n) = 2n \frac{\log_2(n)}{\log_2(3)} \approx 1.26 n \log_2(n)$$

2-way merge sort requires less comparisons!
In fact, the number of compares done by 2-way merge sort is optimal.

Interview Question

How can we shuffle a Linked List in $O(n \log n)$ time and using $O(\log n)$ extra memory?

Goal. Rearrange the elements in the linked list such that all possible $n!$ permutations are equally likely.

Interview Question

How can we shuffle a Linked List in $O(n \log n)$ time and using $O(\log n)$ extra memory?

Goal. Rearrange the elements in the linked list such that all possible $n!$ permutations are equally likely.

Shuffle when sorting

Use Merge Sort. Instead of comparing elements during the *merge* operation to decide on which list to copy from, flip a coin to pick randomly a list to copy from.

Interview Question

How can we shuffle a Linked List in $O(n \log n)$ time and using $O(\log n)$ extra memory?

Goal. Rearrange the elements in the linked list such that all possible $n!$ permutations are equally likely.

Shuffle when sorting

Use Merge Sort. Instead of comparing elements during the *merge* operation to decide on which list to copy from, flip a coin to pick randomly a list to copy from.

Note. If shuffling does not have to be in-place, we can copy the elements to an array, use Knuth's Shuffle to shuffle the array (runs in $\Theta(n)$), and then copy the elements back to the linked list.

```
for i = last down to 1:
    j = random integer (0 <= j <= i)
    swap a[i] with a[j]
```

These slides are partially based on:
https://www.cs.princeton.edu/courses/archive/fall21/cos226/lectures/22Mergesort.pdf

Images:
https://static01.nyt.com/images/2012/05/06/books/review/06POUNDSTONE/06POUNDSTONE-superJumbo.jpg?quality=75&auto=webp
https://miro.medium.com/max/1400/0*Vm6RJ1W0oroOuNEw.jpg
http://public.callutheran.edu/~reinhart/CSC521MSCS/Week5/KnuthVonNeumann.pdf
https://image.shutterstock.com/image-vector/cartoon-character-old-wise-man-260nw-600200147.jpg