Merge Sort

https://cs.pomona.edu/classes/cs140/

Outline

Topics and Learning Objectives

- Learn how the merge sort algorithm operates
- Become aware of the “Divide and Conquer” algorithmic paradigm by analyzing merge sort

Exercise

- Recursion tree

Extra Resources

- CLRS (Cormen Book): Chapter 4
- Algorithms Illuminated: Part 1: Chapter 1

Divide and Conquer

- This is an algorithm design paradigm
- Most divide and conquer algorithms are recursive in nature
- The basic idea is to break the problem into easier-to-solve subproblems

- What's easier to do:
  - Sort 0, 1, or 2 numbers, or
  - Sort 10 numbers

Merge Sort

- This is a “Divide and Conquer”-style algorithm
- Improves over insertion sort in the worst case
- Unlike insertion sort, the best/average/worst case running times of merge sort are all the same

What is the running time of each line?

TTYN

```
FUNCTION MergeSort(array)
    n = array.length
    IF n == 1
        RETURN array

    left_sorted = MergeSort(array[0 ..< n//2])
    right_sorted = MergeSort(array[n//2 ..< n])

    array_sorted = Merge(left_sorted, right_sorted)

    RETURN array_sorted
```

What is the running time of each line?

```
      FUNCTION MergeSort(array)
O(1)    n = array.length
O(1)    IF n == 1
O(1)        RETURN array

O(?)    left_sorted = MergeSort(array[0 ..< n//2])
O(?)    right_sorted = MergeSort(array[n//2 ..< n])

        array_sorted = Merge(left_sorted, right_sorted)

        RETURN array_sorted
```

What is the running time of each line?

```
T(n)    FUNCTION MergeSort(array)
O(1)      n = array.length
O(1)      IF n == 1
O(1)          RETURN array

T(n/2)    left_sorted = MergeSort(array[0 ..< n//2])
T(n/2)    right_sorted = MergeSort(array[n//2 ..< n])

          array_sorted = Merge(left_sorted, right_sorted)

          RETURN array_sorted
```

What is the running time of each line?

```
T(n)    FUNCTION MergeSort(array)
O(1)      n = array.length
O(1)      IF n == 1
O(1)          RETURN array

T(n/2)    left_sorted = MergeSort(array[0 ..< n//2])
T(n/2)    right_sorted = MergeSort(array[n//2 ..< n])

O(?)      array_sorted = Merge(left_sorted, right_sorted)

O(1)      RETURN array_sorted
```

What is the running time of each line?

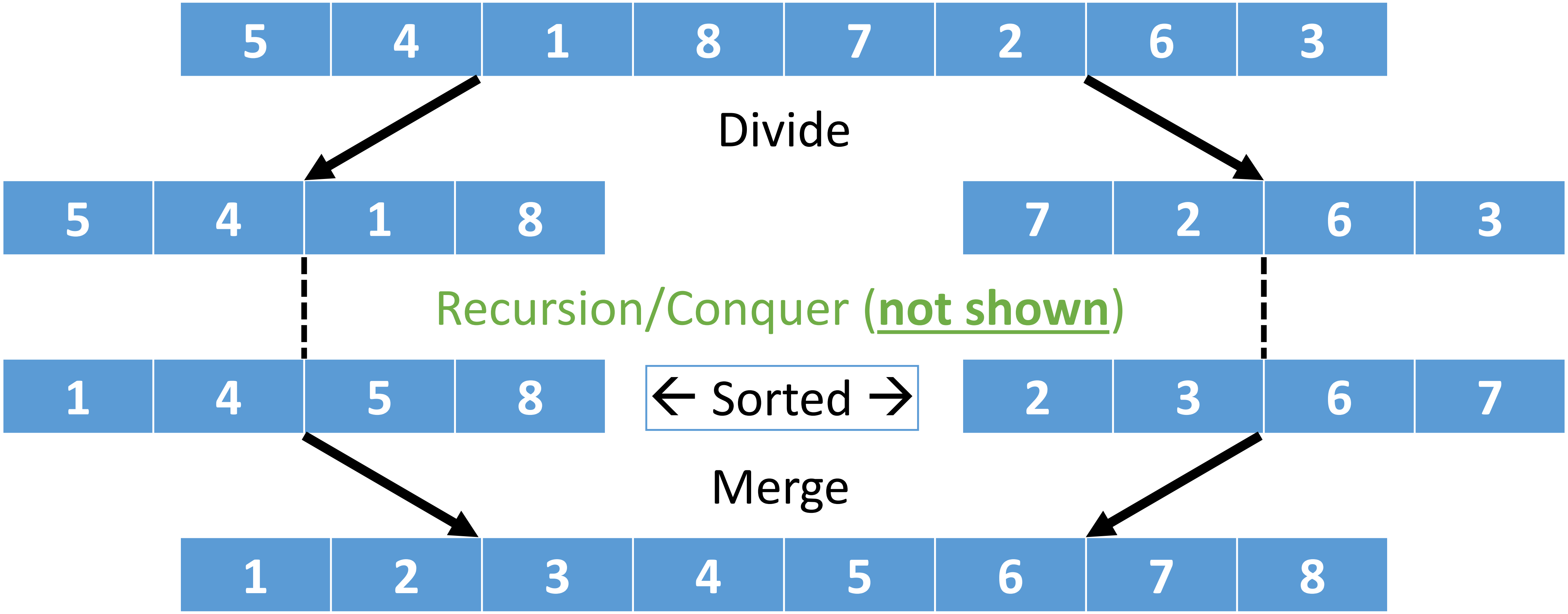
```
T(n)     FUNCTION MergeSort(array)
O(1)       n = array.length
O(1)       IF n == 1
O(1)           RETURN array

T(n/2)     left_sorted = MergeSort(array[0 ..< n//2])
T(n/2)     right_sorted = MergeSort(array[n//2 ..< n])

O(?)       array_sorted = Merge(left_sorted, right_sorted)

O(1)       RETURN array_sorted
```

$$T(n) = 2\,T(n/2) + O(?) + 4\,O(1)$$
$$= 2\,T(n/2) + O(?)$$

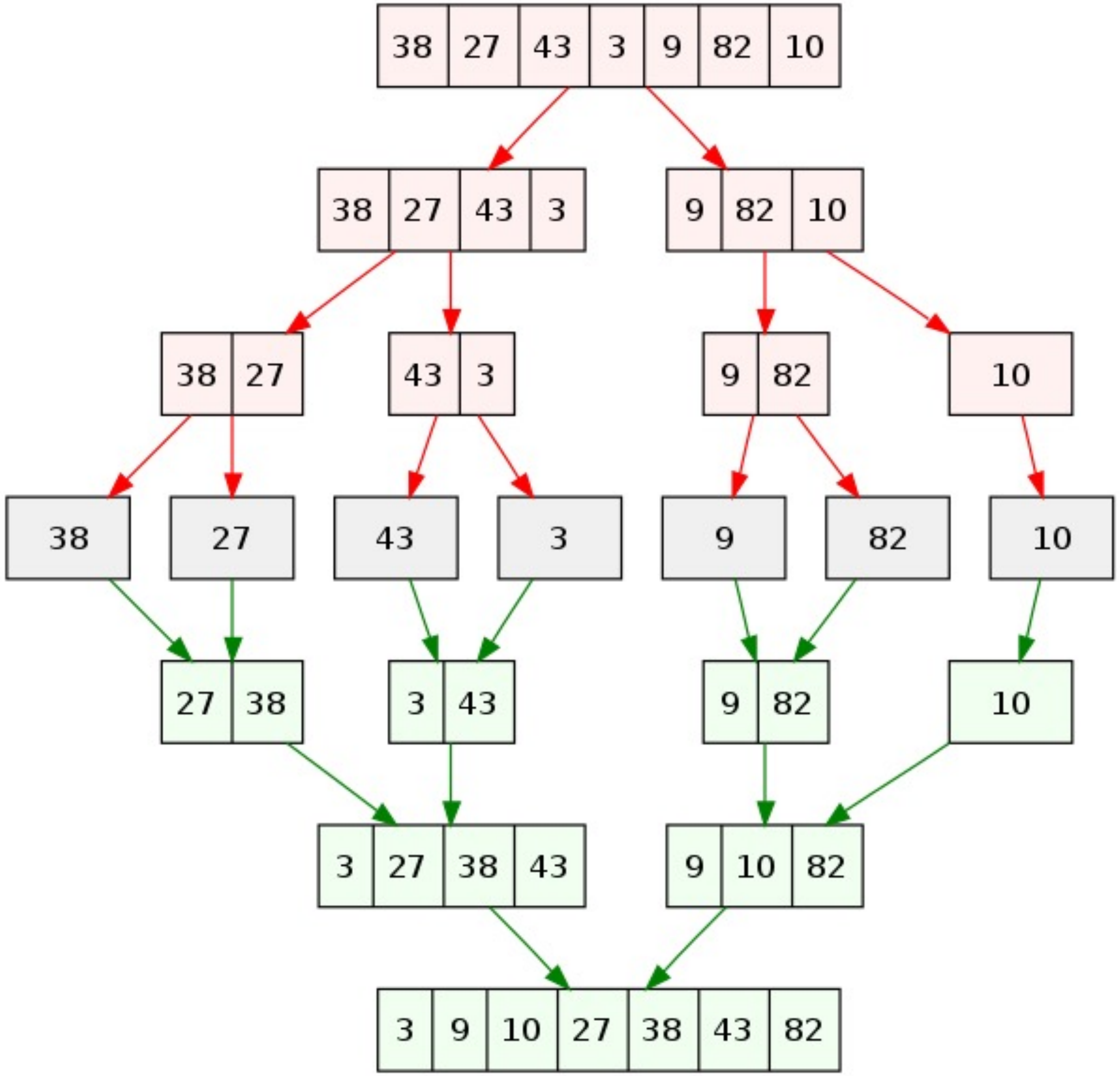
Recurrence Equation

$$
\begin{aligned}
T(n) &= 2\,T(n/2) + O(?) + 4\,O(1) \\
&= 2\,T(n/2) + O(?)
\end{aligned}
$$

Merge Sort



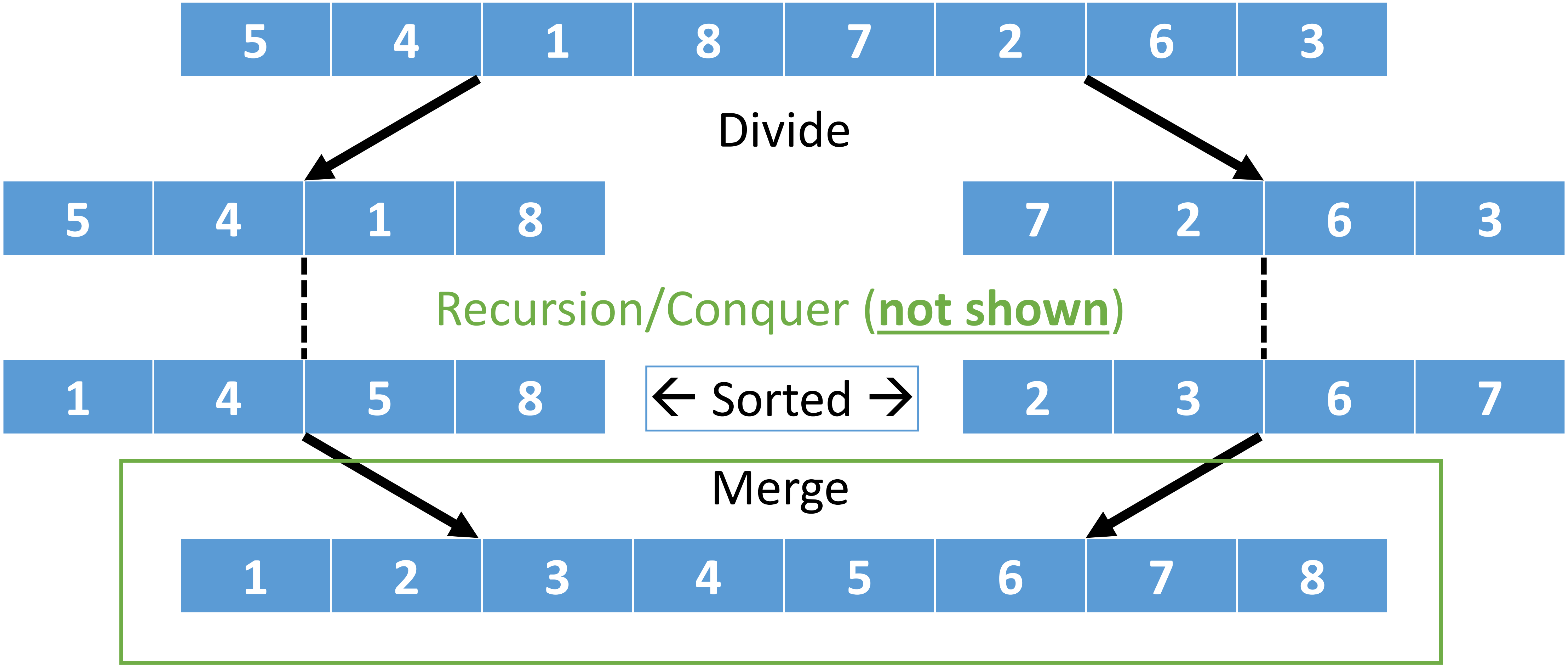
5 4 1 8 7 2 6 3
Divide
5 4 1 8
7 2 6 3
Recursion/Conquer (not shown)
1 4 5 8
← Sorted →
2 3 6 7
Merge
1 2 3 4 5 6 7 8


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

Merge Sort

Write the Merge routine

TTYN



```
FUNCTION Merge(one, two)
    out[one.length + two.length] # Declare array
```

```
FUNCTION Merge(one, two)
    out[one.length + two.length]
    i = j = k = 0
    WHILE k < out.length
        IF one[i] < two[j]
            out[k] = one[i]
            i = i + 1
        ELSE
            out[k] = two[j]
            j = j + 1
        k = k + 1
```

Ignoring invalid indices

What is the total running time?

TTYN

```
FUNCTION Merge(one, two)
    out[one.length + two.length]
    i = j = k = 0
    WHILE k < out.length
        IF one[i] < two[j]
            out[k] = one[i]
            i = i + 1
        ELSE
            out[k] = two[j]
            j = j + 1
        k = k + 1
```

Ignoring invalid indicies

Line	Total Running Time
out[one.length + two.length]	4
i = j = k = 0	3
WHILE k < out.length	2 (m + 1)
IF one[i] < two[j]	3 m
out[k] = one[i]	3 m
i = i + 1	2 m
ELSE	0
out[k] = two[j]	3 m
j = j + 1	2 m
k = k + 1	2 m

$$T_{merge}(m) = 12\ m + 9$$

FUNCTION Merge(one, two)	Total Running Time
out[one.length + two.length]	4
i = j = k = 0	3
WHILE k < out.length	2 (m + 1)
IF one[i] < two[j]	3 m
ou	3 m
i = i + 1	2 m
ELSE	0
out[k] = two[j]	3 m
j = j + 1	2 m
k = k + 1	2 m

Ignoring invalid indicies

Simplifying the running time

- We don't need to be *exactly* correct with the running time of Merge
- We will eventually remove lower order terms anyway
- Let's simplify the expression a bit:

$$T_{merge}(m) = 12m + 9$$

$$T_{merge}(m) \leq 12m + 9m$$

$$T_{merge}(m) \leq 21m$$

Merging

We have an idea of the cost of an individual call to merge:

$$T(m) \leq 21m$$

What else do we need to know to calculate the total time of **MergeSort**?

1. How many times do we merge in total?
2. What is the size of each merge? (In other words: **What is m?)**

What is the running time of each line?

```
T(n)     FUNCTION MergeSort(array)
O(1)       n = array.length
O(1)       IF n == 1
O(1)           RETURN array

T(n/2)     left_sorted = MergeSort(array[0 ..< n//2])
T(n/2)     right_sorted = MergeSort(array[n//2 ..< n])

O(?)       array_sorted = Merge(left_sorted, right_sorted)

O(1)       RETURN array_sorted
```

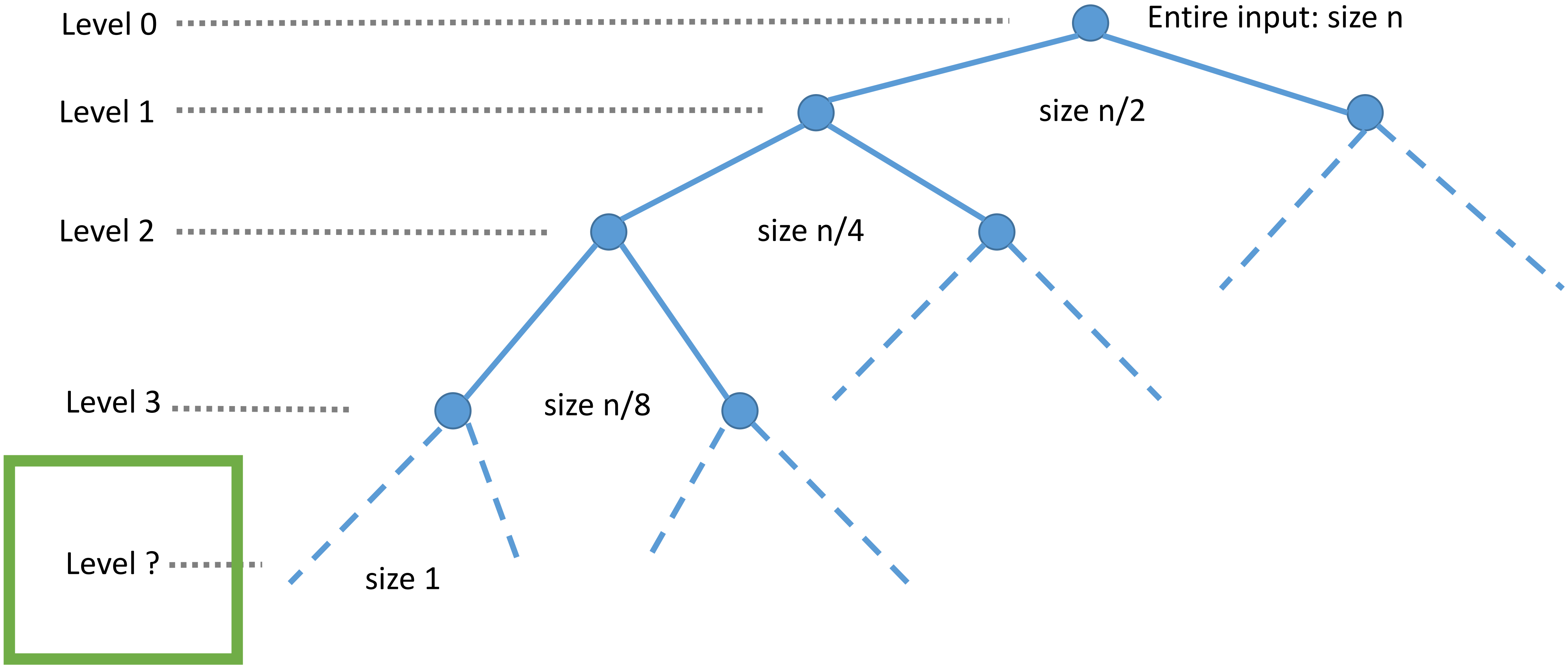
$$
\begin{aligned}
T(n) &= 2\,T(n/2) + O(?) + 4\,O(1) \\
&= 2\,T(n/2) + O(?)
\end{aligned}
$$

What is the running time of each line?

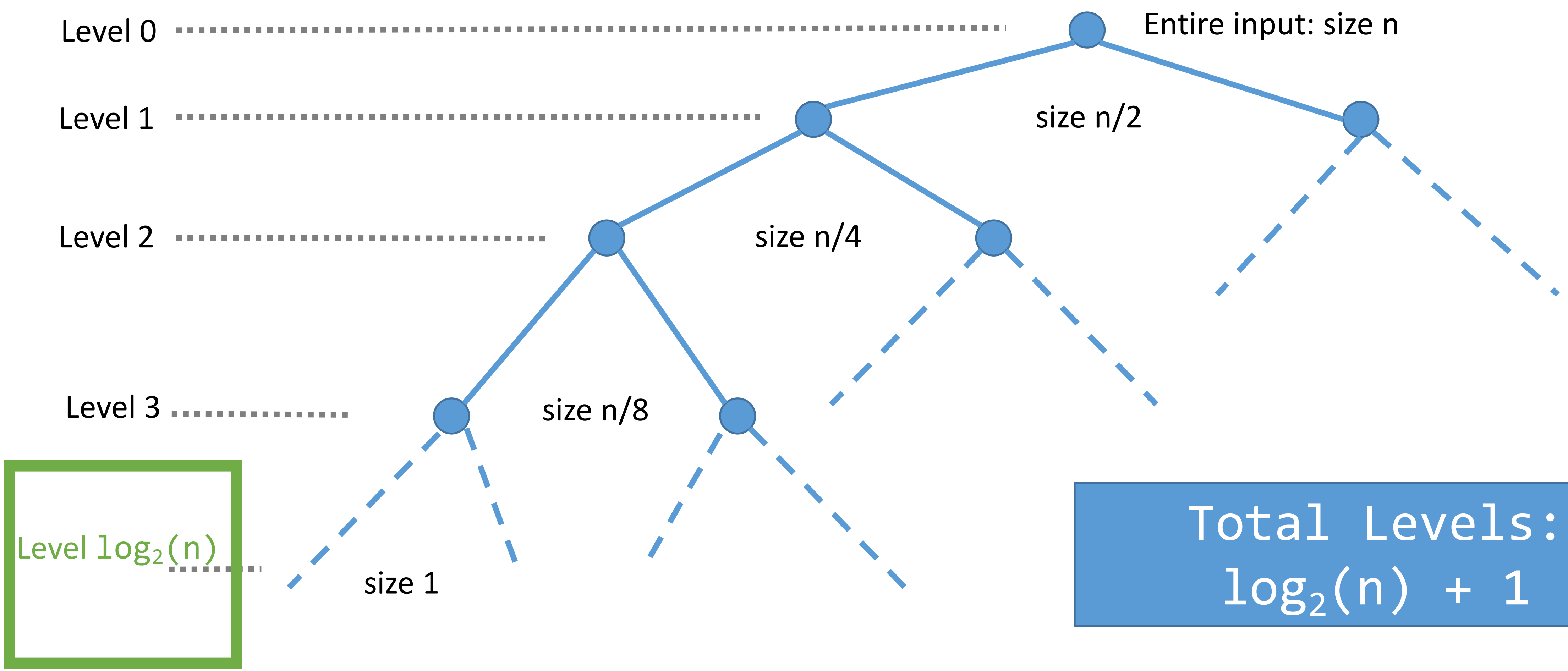
```
T(n)     FUNCTION MergeSort(array)
O(1)       n = array.length
O(1)       IF n == 1
O(1)           RETURN array

T(n/2)     left_sorted = MergeSort(array[0 ..< n//2])
T(n/2)     right_sorted = MergeSort(array[n//2 ..< n])

O(n)       array_sorted = Merge(left_sorted, right_sorted)

O(1)       RETURN array_sorted
```

$$
\begin{aligned}
T(n) &= 2\,T(n/2) + O(n) + 4\,O(1) \\
&= 2\,T(n/2) + O(n)
\end{aligned}
$$

How many times do we call `Merge`?



How many times do we call `Merge`?



Total Levels:
$\log_2(n) + 1$

Exercise

TTYN

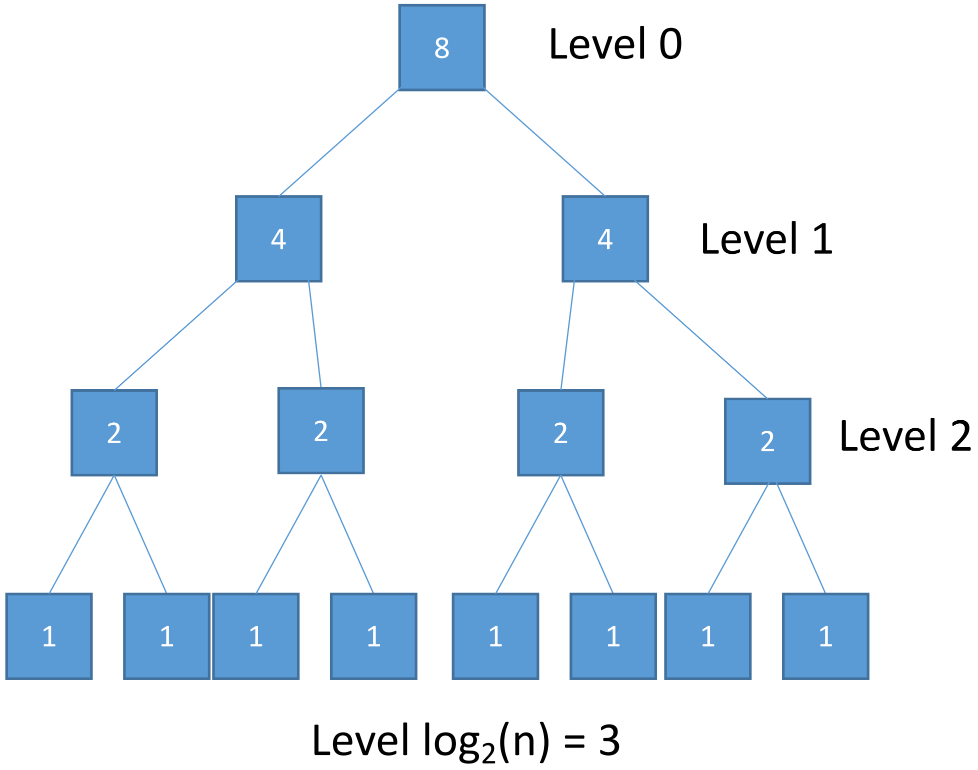
How many sub-problems are there at level L? The top level is Level 0, the second level is Level 1, and the bottom level is Level $\log_2(n)$

Answer: 2^L

How many elements are there for a given sub-problem found in level L?

Answer: $n/2^L$

How many computations are performed at a given level? The cost of a `Merge` was 21m.



Exercise

How many sub-problems are there at level L? The top level is Level 0, the second level is Level 1, and the bottom level is Level $\log_2(n)$

Answer: 2^L

How many elements are there for a given sub-problem found in level L?

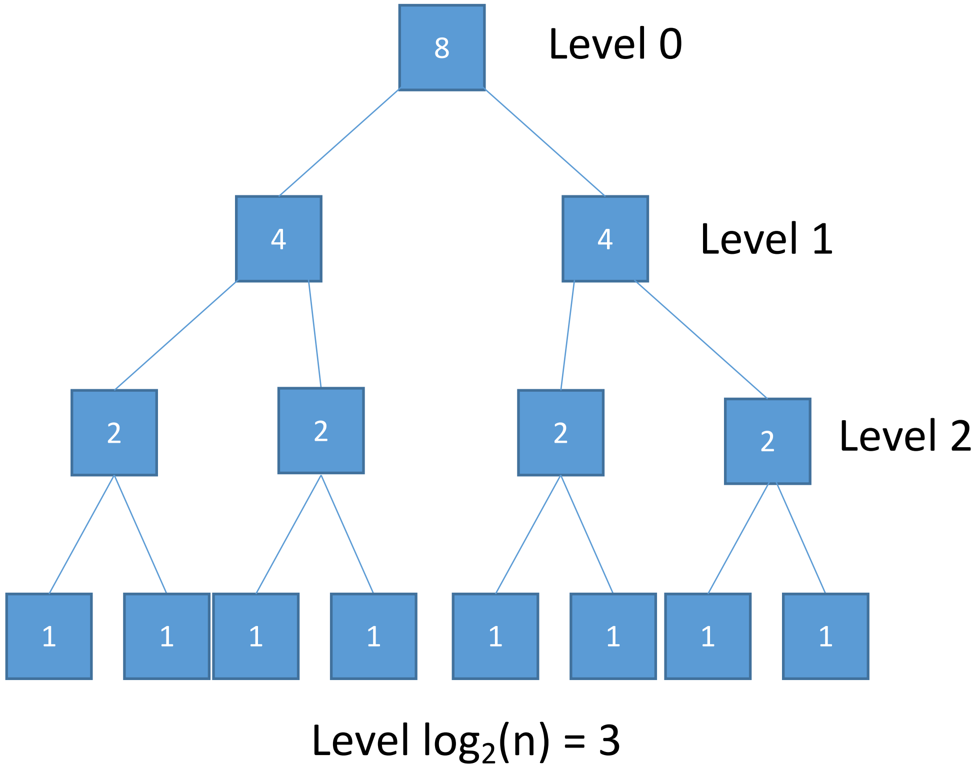
Answer: $n/2^L$

How many computations are performed at a given level? The cost of a `Merge` was 21m.

Answer: $2^L \, 21(n/2^L) \rightarrow 21n$

What is the total computational cost of merge sort?

Answer: $21n\,(\log_2(n) + 1)$



Exercise

How many sub-problems are there at level L? The top level is Level 0, the second level is Level 1, and the bottom level is Level $\log_2(n)$

Answer: 2^L

How many elements are there for a given sub-problem found in level L?

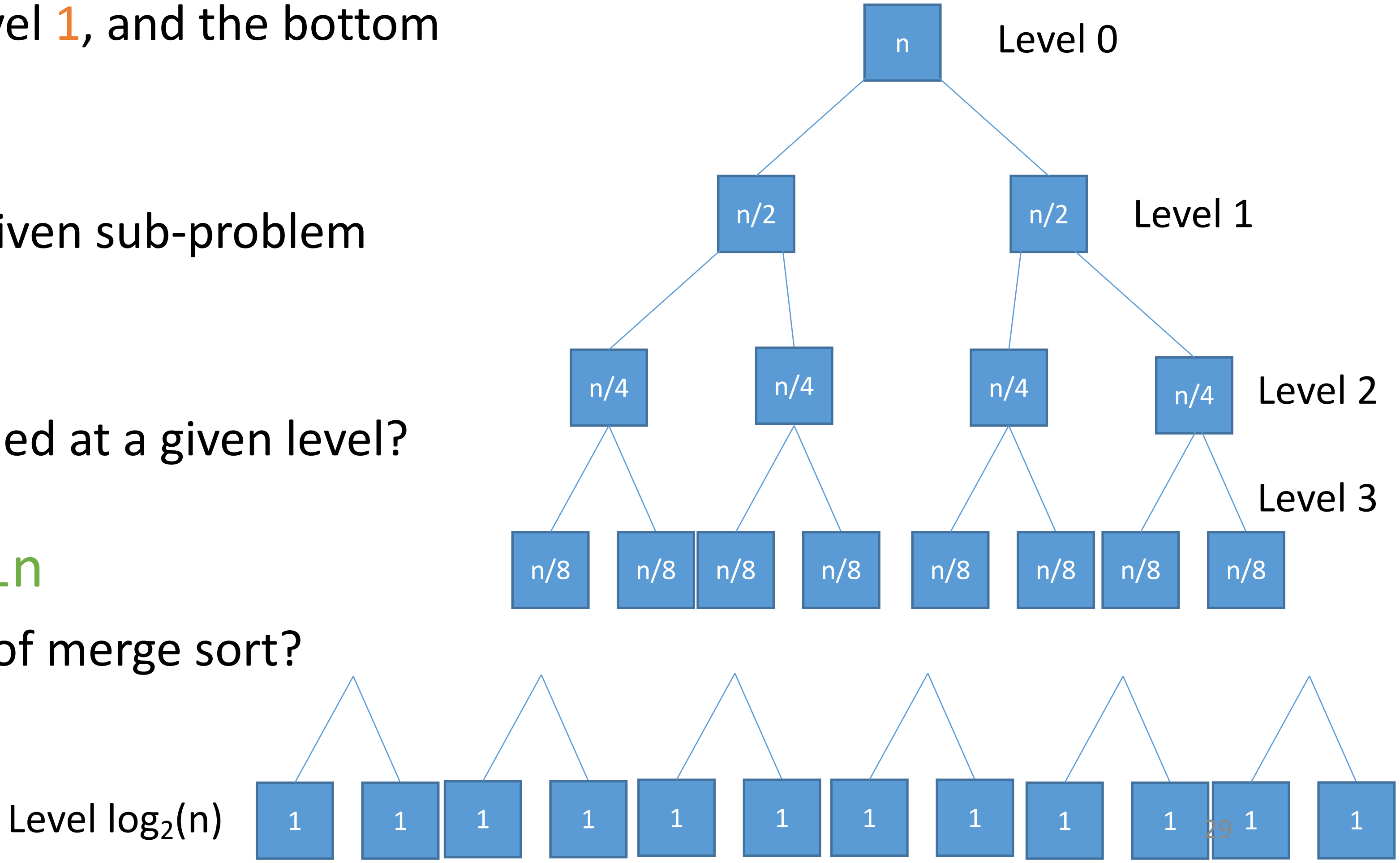
Answer: $n/2^L$

How many computations are performed at a given level? The cost of a `Merge` was 21m.

Answer: $2^L \, 21(n/2^L) \rightarrow 21n$

What is the total computational cost of merge sort?

Answer: $21n\,(\log_2(n) + 1)$



Merge Sort

Divide and Conquer

- constantly halving the problem size and then merging

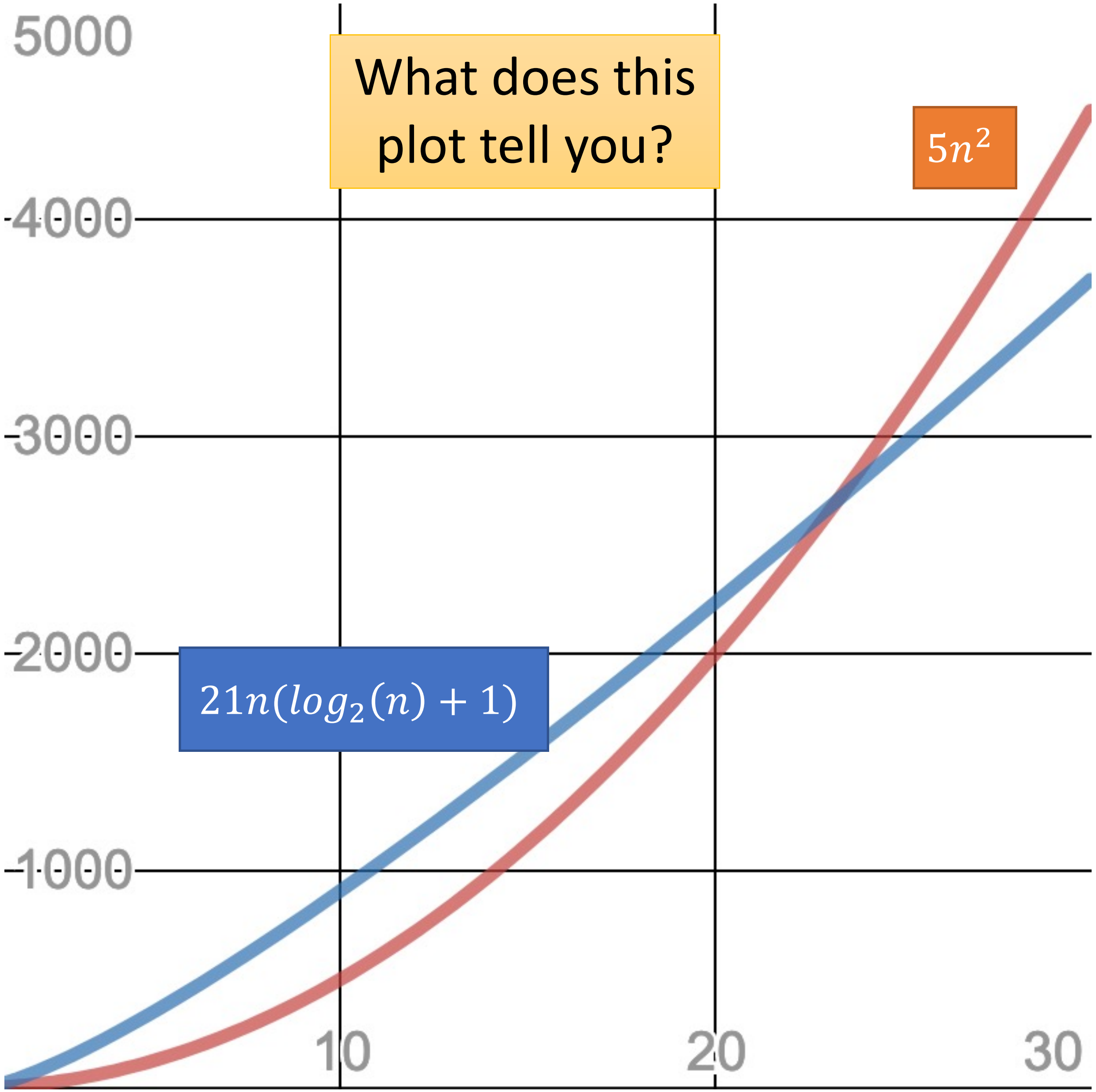
Total running time of roughly $21n \log_2(n) + 21n$

Compared to insertion sort with an average **total running time** of $\frac{1}{2} n^2$

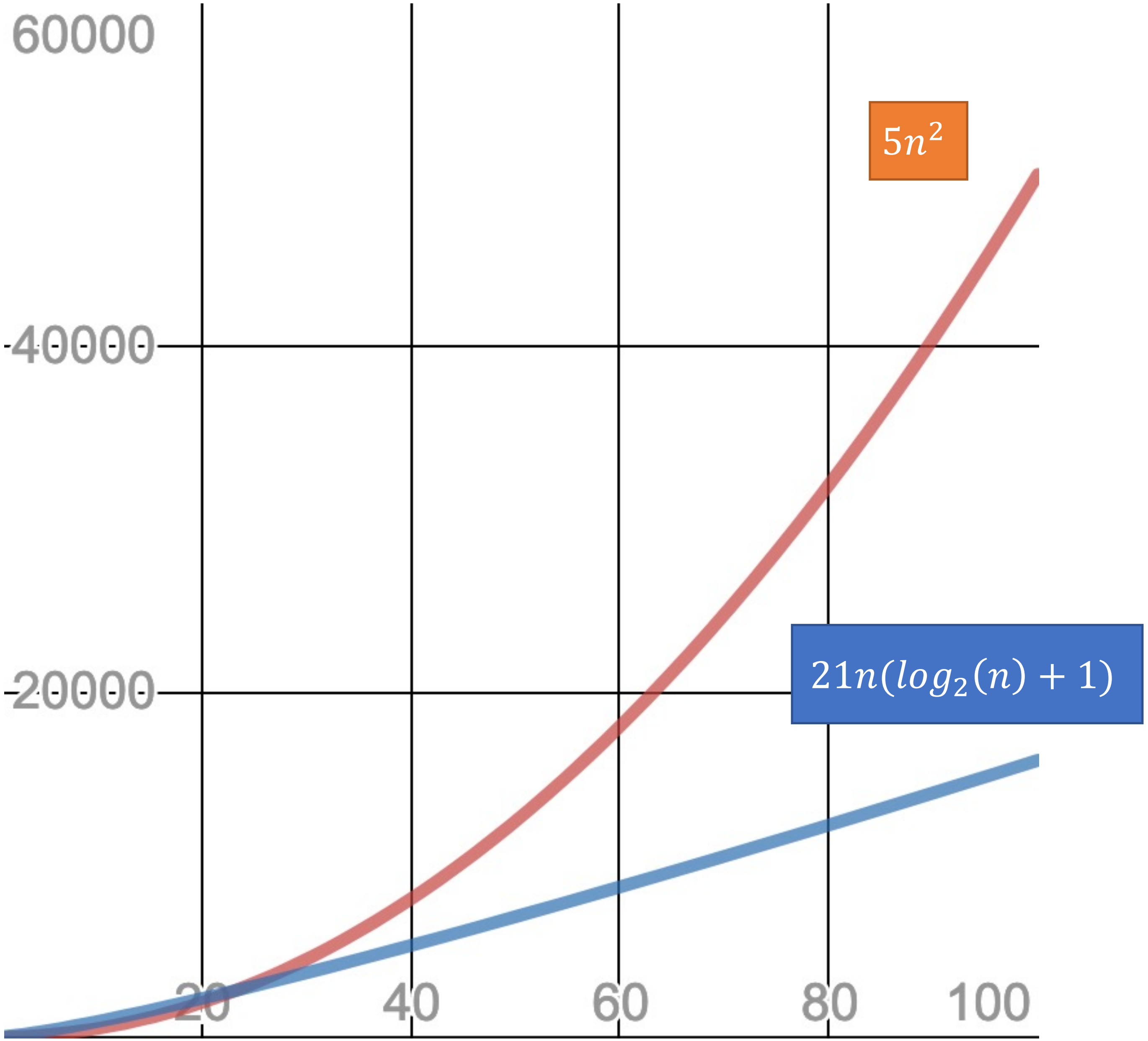
- For small values of n, insertion sort is better

Which algorithm is better?

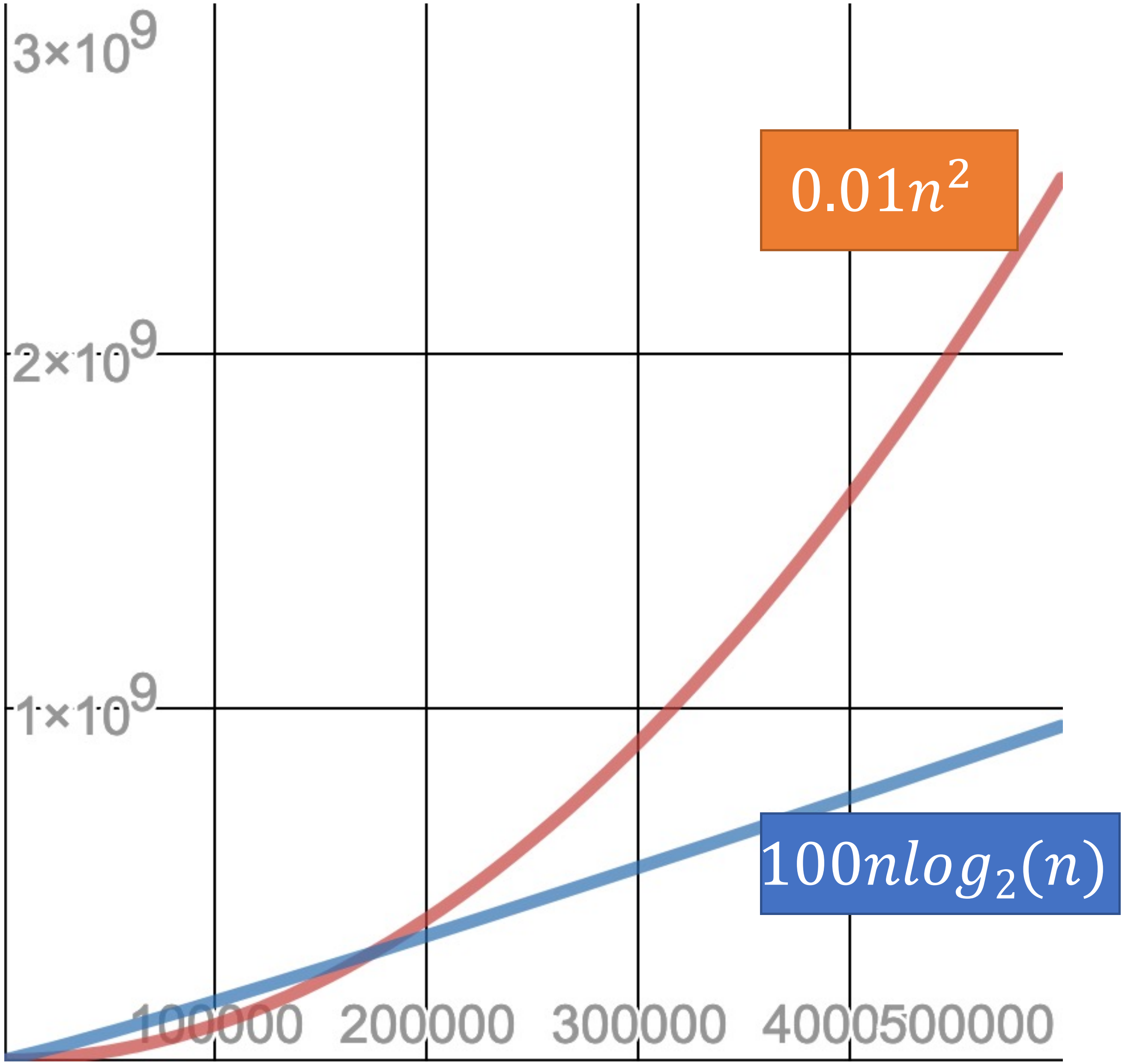
Merge Sort Verse Insertion Sort Worst-Case



Merge Sort Verse Insertion Sort Worst-Case



Constants



3×10^9
2×10^9
1×10^9
100000 200000 300000 400000 500000
$0.01n^2$
$100nlog_2(n)$