CMPT 225: Data Structures & Programming – Unit 26 – Merge Sort

Dr. Jack Thomas

Simon Fraser University

Spring 2021

Today’s Topics

- The Need for Sorting
- Divide and Conquer
- Merging in Arrays and Lists
- Analysis
- Merge-Sort in Java

Welcome to Sorting Algorithms

- A major topic in Computer Science, both in the theory and in programming practice.
- These cover any process where a data set can be **put into order**, like ordering numbers from smallest to largest, or alphabetical order.
- Many of our data structures take pains to **keep their data sorted**, have a **sorting step**, or depend on **data already being sorted** in order to be efficient.
- Already introduced with **Insertion Sort**, way back in unit 4 on arrays.

Remember This Guy?

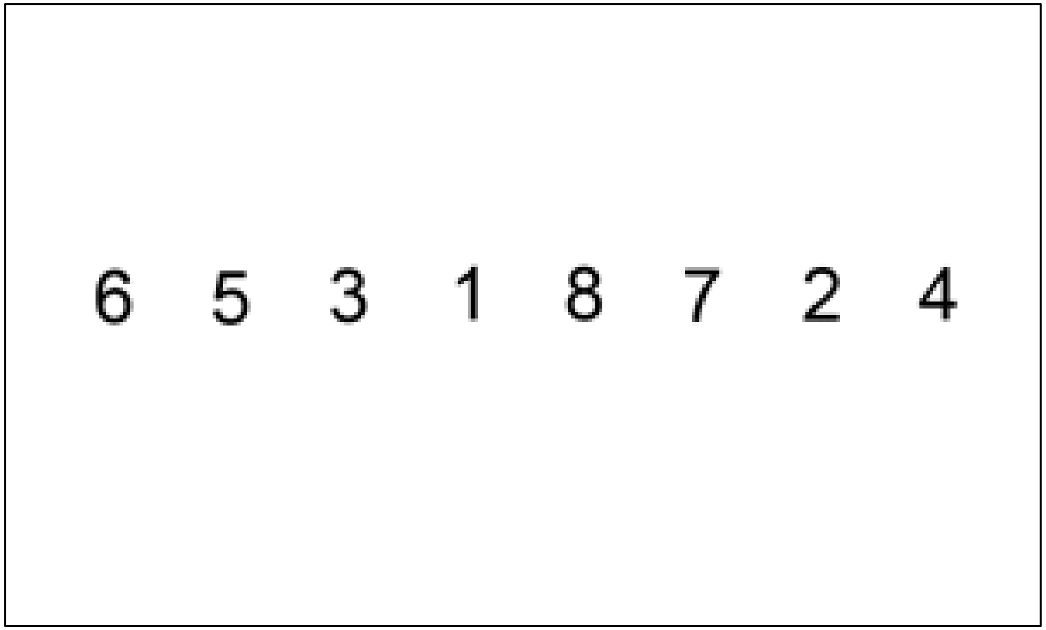


Image credit: https://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

Why Do We Need More Than One?

- **Different sorting methods have different properties** that make them better fits for different circumstances.
- For example, some are only efficient when **paired with certain data structures**, or may **perform more consistently** even if they share a similar O().
- We’ll perform an **initial analysis** of each algorithm as they’re introduced, and then **again once all our sorting options are available**.

Divide-and-Conquer Algorithms

- A common **algorithmic design pattern** we can apply to sorting is **Divide-and-Conquer**.
- We **divide the problem** to be solved into multiple smaller problems that are easier to solve before **combining** them back together again.

The Three Steps of a Divide-and-Conquer Algorithm

1. **Divide**: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution. Otherwise, divide the input data into two or more disjoint subsets.
2. **Recur**: Recursively solve the subproblems associated with the subsets.
3. **Conquer**: Take the solutions to the subproblems and “merge” them into a solution to the original problem.

Sorting Through Divide-and-Conquer

- Sorting a long sequence of values can be imagined as answering many smaller questions **comparing pairs of values** to decide which belongs in front of which.
- **Merge-Sort** will let **us break our sequence down** to a set of those smaller comparison problems, solve those, and then **Merge our smaller sorted sequences** back together again.

Merge-Sort as Divide-and-Conquer

1. **Divide**: If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S1 and S2, each containing about half of the elements of S; that is, S1 contains the first n/2 elements of S, and S2 contains the remaining n/2 elements.
2. **Recur**: Recursively sort sequences S1 and S2.
3. **Conquer**: Put back the elements into S by merging the sorted sequences S1 and S2 into a sorted sequence.

Let’s Get One Of Those Wikipedia Visualizations In Here

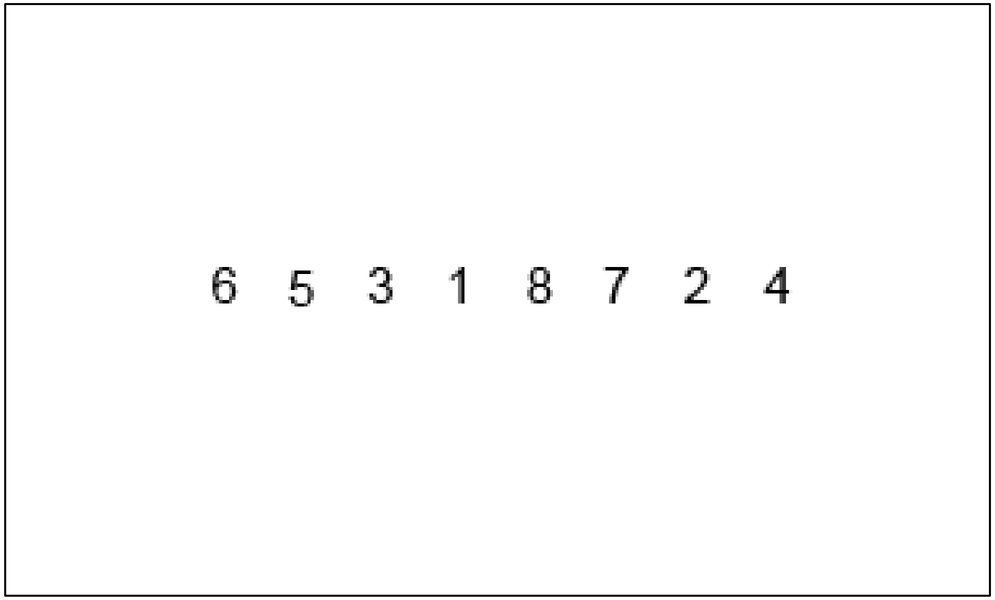
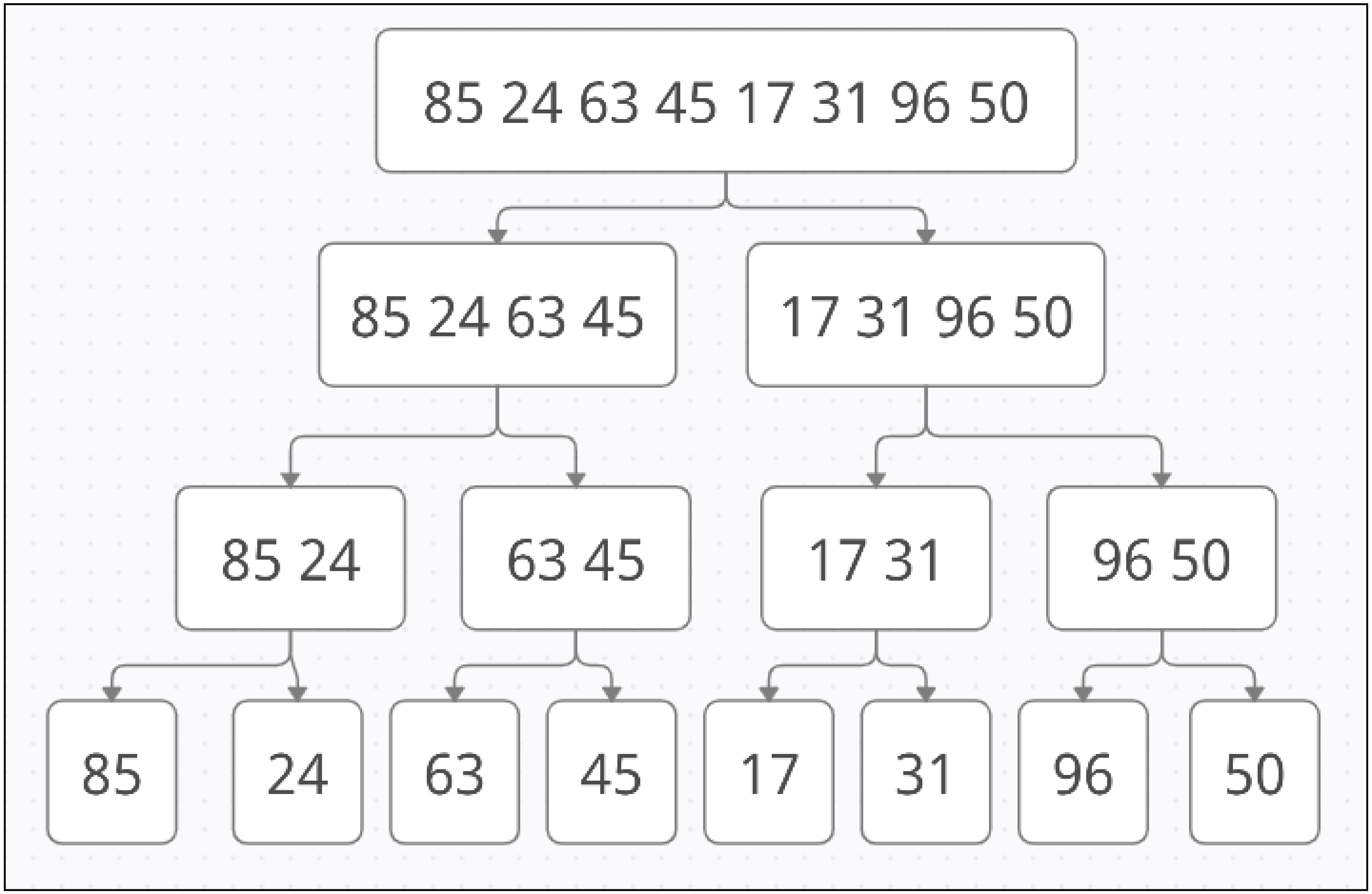
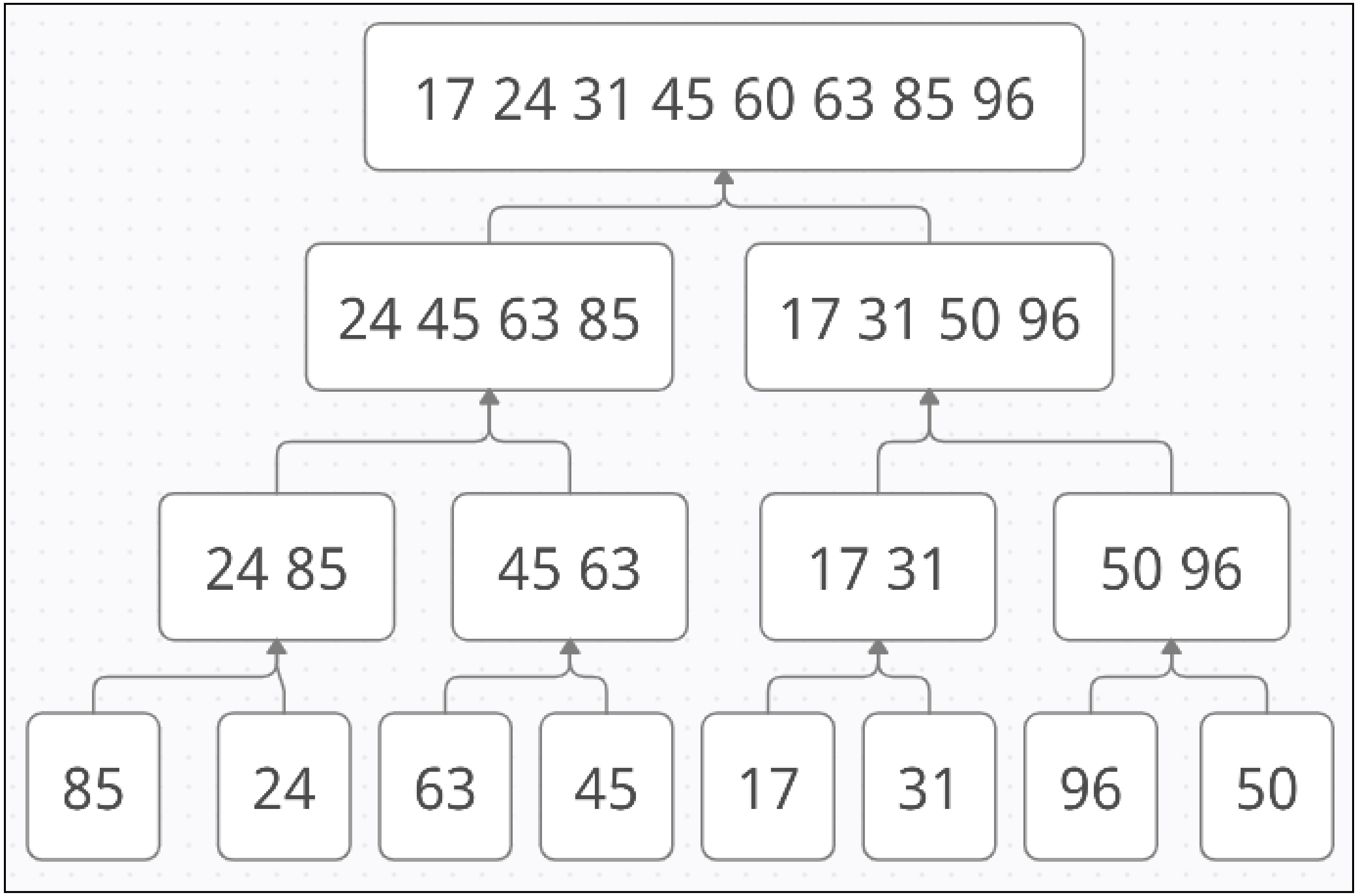


Image credit: https://en.wikipedia.org/wiki/File:Merge-sort-example-300px.gif

Divide…


85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
85
24
63
45
17
31
96
50

…and Conquer



17 24 31 45 60 63 85 96

24 45 63 85

17 31 50 96

24 85

45 63

17 31

50 96

85

24

63

45

17

31

96

50

Ahahaha It’s A Tree Again

- Surprise!
- **Merge-Sort** is performing **Binary Recursion** to implicitly create a **Complete Binary Tree**.
- This is also a **Decision Tree**, because each node of the Tree is essentially a question about which value belongs before which.
- All **Comparison-Based Sorting Algorithms** produce a Decision Tree, and Decision Trees have some common properties (like their height) which will come into play later.

Merging in Arrays and Lists

- The **Merge step** is the most involved to implement, since it depends more on our choice of underlying data structure.
- As such, there are **two slightly different algorithms for merging** depending on whether we're using an **Array** or a **List**, but they're ultimately doing the same thing (combining two smaller sorted sequences into one larger sorted sequence).

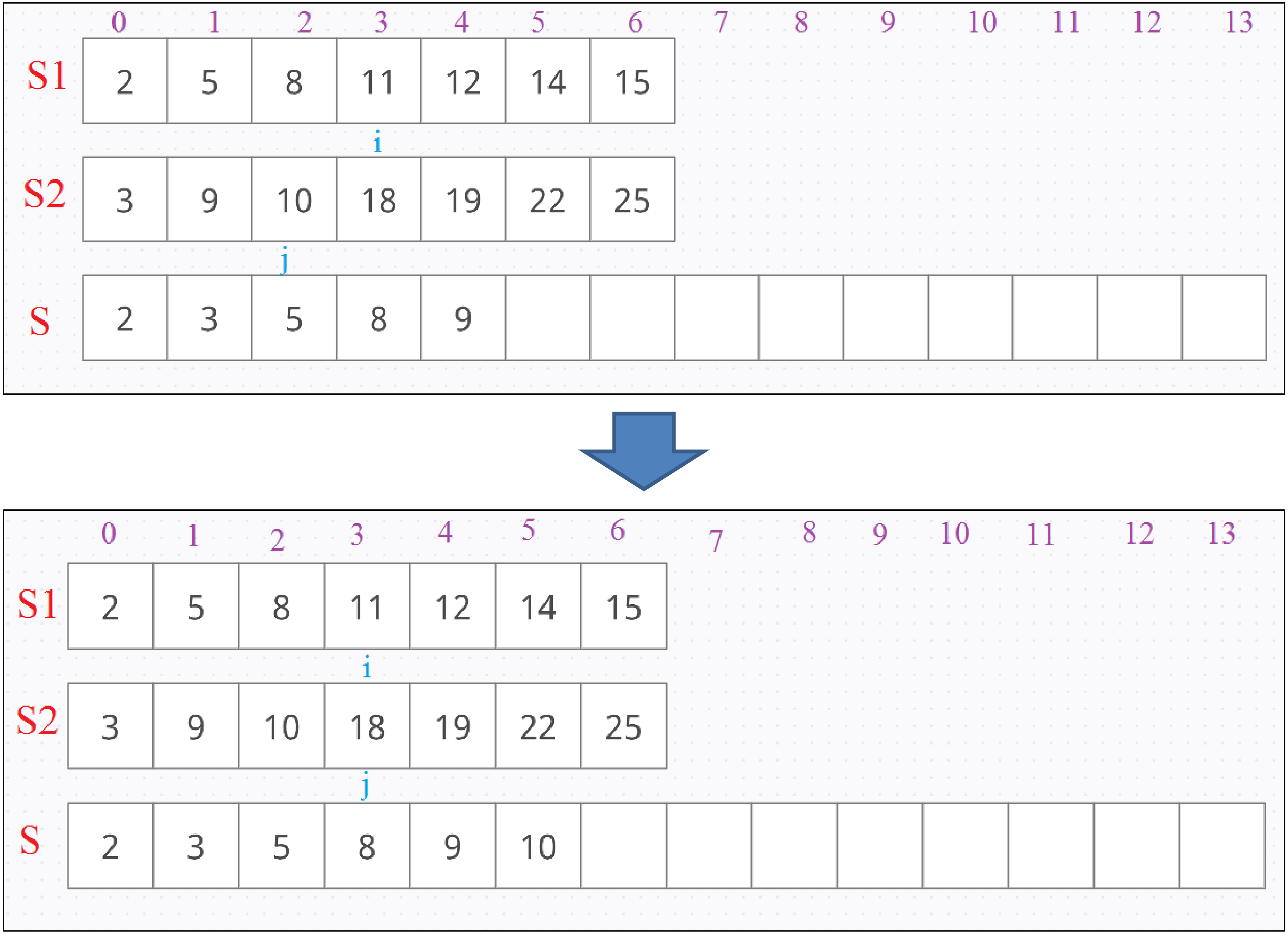
Merge-Sort Algorithm for Arrays

Algorithm merge(S1,S2,S):

Input: Sorted sequences S1 and S2 and an empty sequence S, all of which are implemented as arrays.

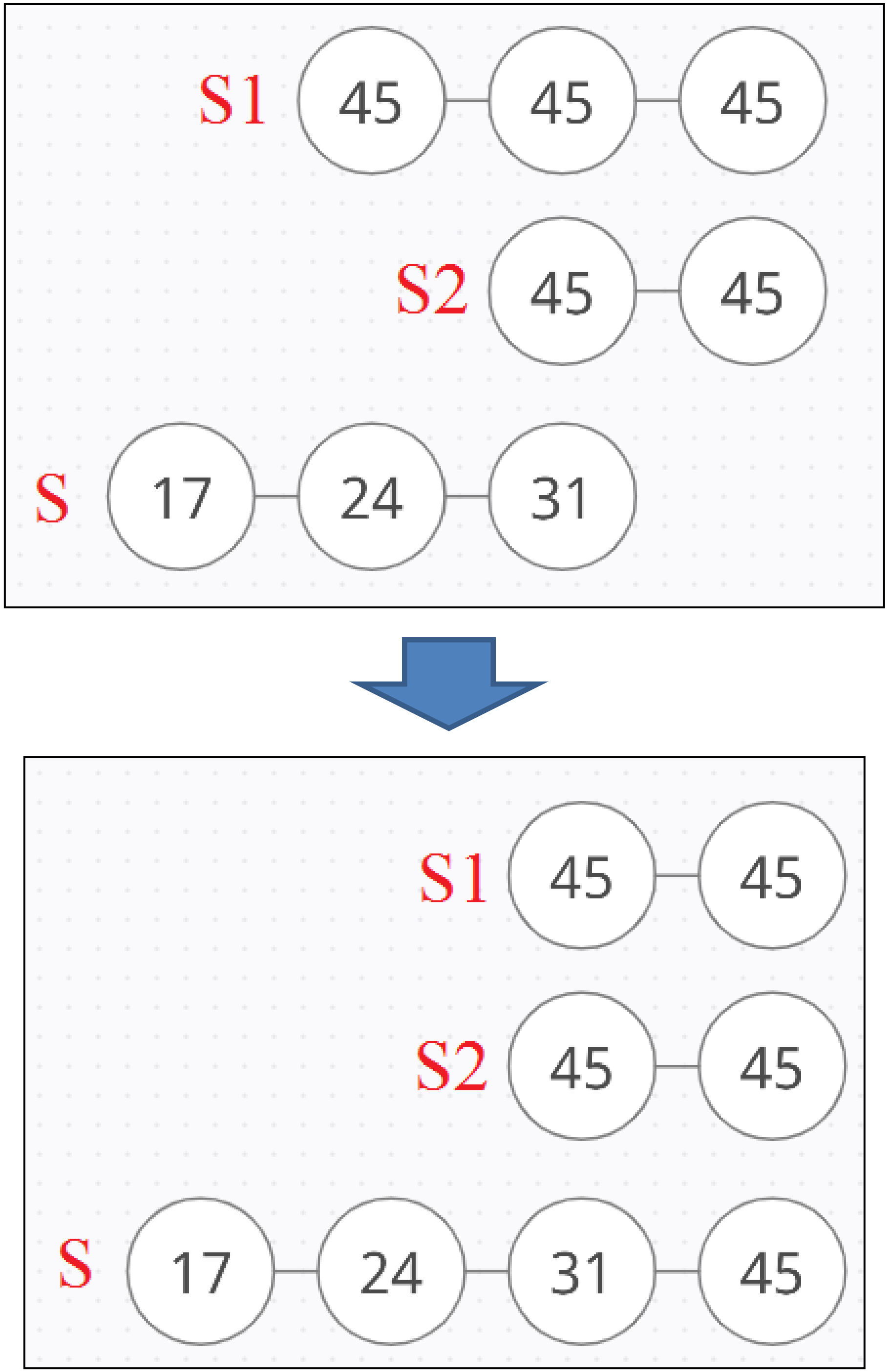
Output: Sorted sequence S containing the elements from S1 and S2.

```
i <- j <- 0
While i < S1.size() and j < s2.size() do
    if S1.get(i) <= S2.get(j) then
        S.addLast(s1.get(i))
        i <- i+1
    else
        S.addLast(S2.get(j))
        j <- j+1
While i < S1.size() do
    S.addLast(S1.get(i))
    i <- i+1
While j< S2.size() do
    S.addLast(S2.get(j))
    j <- j+1
```


0 1 2 3 4 5 6 7 8 9 10 11 12 13
S1
2 5 8 11 12 14 15
i
S2
3 9 10 18 19 22 25
j
S
2 3 5 8 9
0 1 2 3 4 5 6 7 8 9 10 11 12 13
S1
2 5 8 11 12 14 15
i
S2
3 9 10 18 19 22 25
j
S
2 3 5 8 9 10

Merge-Sort Algorithm for Lists

```
Algorithm merge(S1,S2,S):
Input: Sorted sequences S1 and S2 and an empty sequence S,
implemented as linked lists.
Output: Sorted sequence S containing the elements from S1 and S2.
While S1 is not empty and S2 is not empty do
        if S1.first().element() <= S2.first().element() then
                S.addLast(S1.remove(S1.first()))
        else
                S.addLast(S2.remove(S2.first()))
While S1 is not empty do
        S.addLast(S1.remove(S1.first())).
While S2 is not empty do
        S.addLast(S2.remove(S2.first()))
```


S1
45
45
45
S2
45
45
S
17
24
31
S1
45
45
S2
45
45
S
17
24
31
45

Analyzing Merge-Sort

- Each **layer** of the Merge-Sort Decision Tree requires **O(n) comparisons** to resolve, since each layer has n data values in it and every value may need to be compared to find its place in the next round of sequences.
- The **height** of the Tree is **log n**, since it’s a **Complete Binary Tree**.
- With log n layers to resolve at O(n) per layer **Merge-Sort is O(n log n**).
- (Small assumption: this **assumes that comparing two values takes O(1)**, which may not be true of all data, like a complex key.)

Merge-Sort in Java

- Does Java offer built-in Merge-Sort? **No**.
- Well, not directly. Not all the time.
- The **.sort()** function found **in java.util.Collections for Java 7** was a Merge Sort, but was **replaced in Java 8**.
- Sorting in particular is a topic where Java doesn't care to implement every possibility, and will even change the underlying algorithm for common functions based on the current trend or newly developed variants.
- Don't worry, though, I'm sure you can implement Merge-Sort on your own!

Recap – Merging the Key Points Into One Slide

- We introduced the topic of **Sorting Algorithms** and our motivation for exploring them.
- **Merge-Sort** is the first (after **Insertion-Sort**) of our sorting methods, which uses **Divide-and-Conquer** to produce a **Decision Tree**.
- The **Merge step** does most of the actual work, and looks different depending on whether our data structure is **array- or list-based**.
- Merge-Sort runs **in O(n log n)**.
- **Java does not offer Merge-Sort** by default, though it does offer sorting.