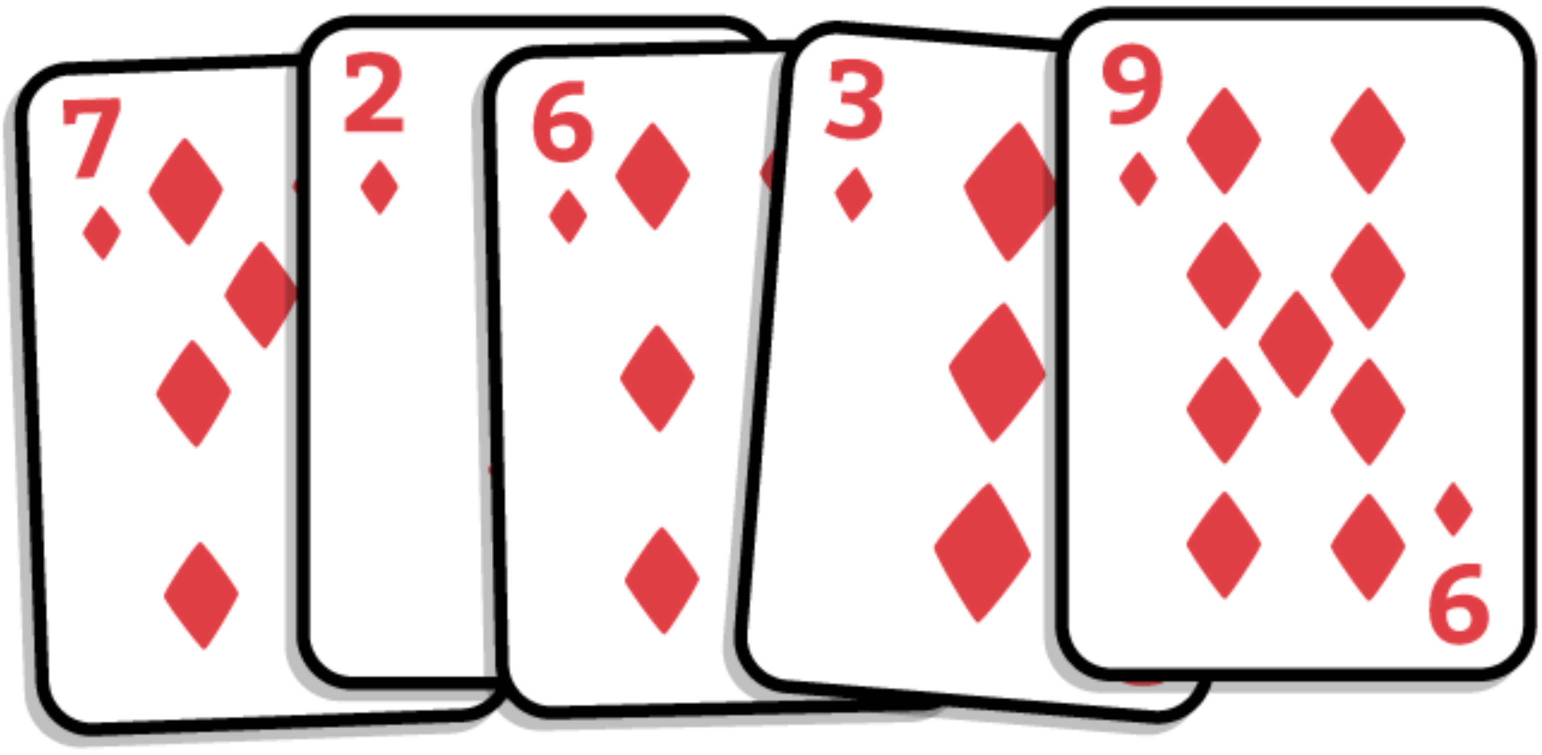
15 Merge Sort Written by Márton Braun

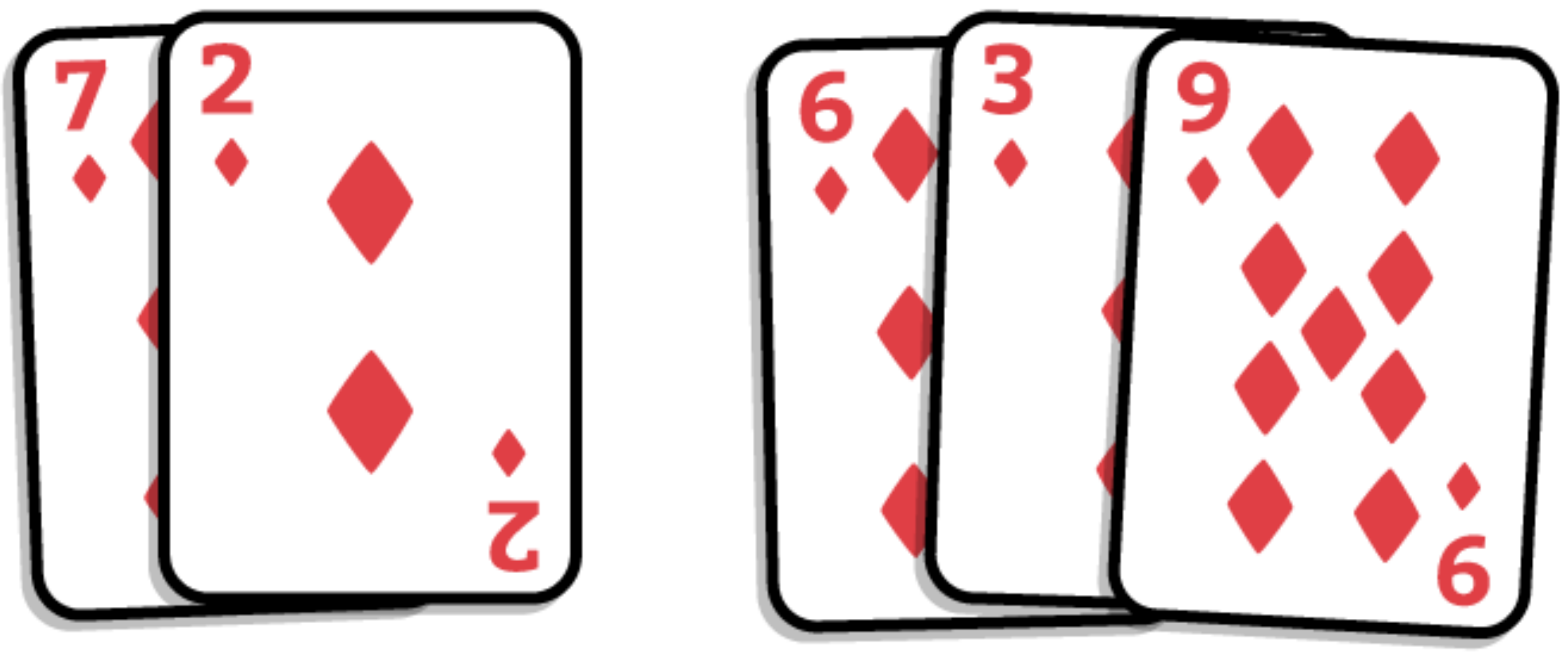
Merge sort is one of the most efficient sorting algorithms. With a time complexity of $O(n \log n)$, it's one of the fastest of all general-purpose sorting algorithms. The idea behind merge sort is **divide and conquer** — to break a big problem into several smaller, easier-to-solve problems, and then combine those solutions into a final result. The merge sort mantra is to *split first* and *merge after*.

As an example, assume that you're given a pile of unsorted playing cards:

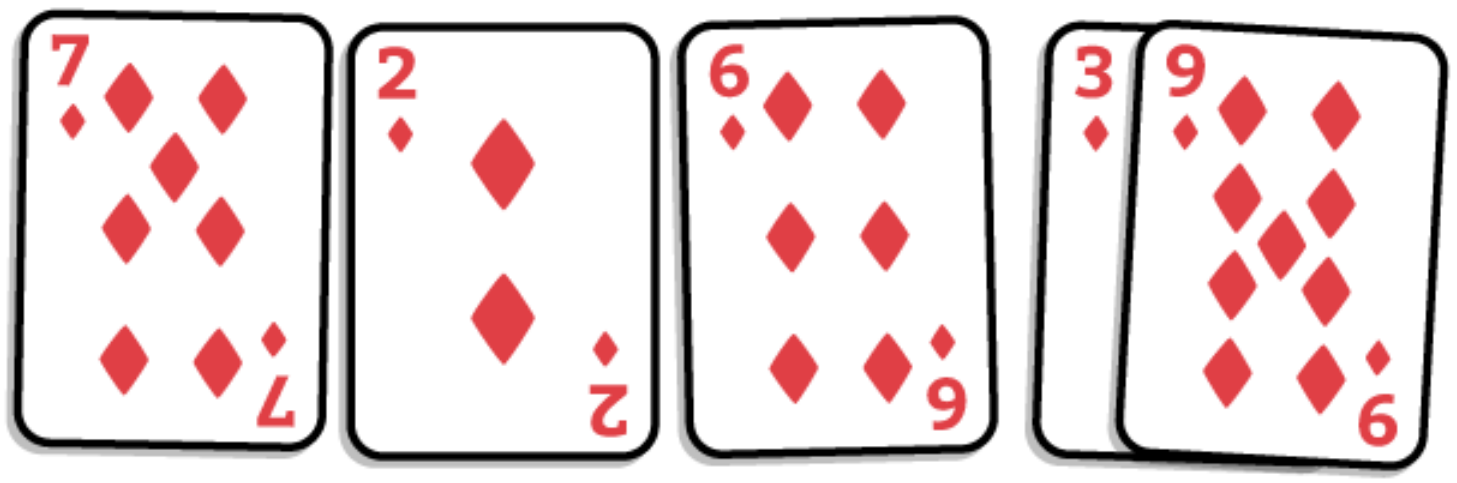


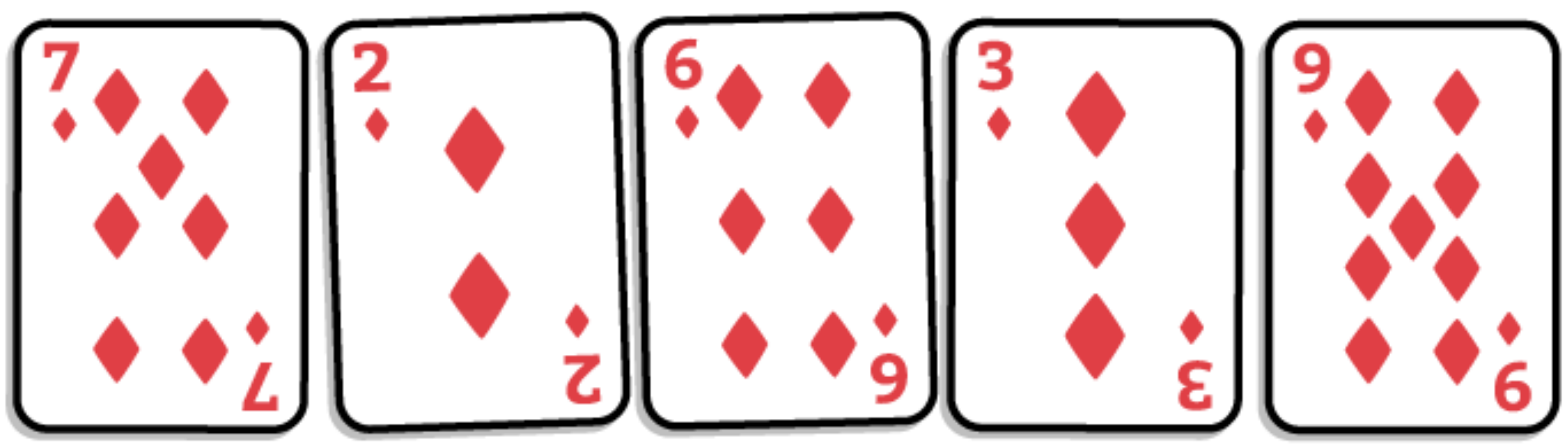
The merge sort algorithm works as follows:

1. Split the pile in half, which gives you two unsorted piles:

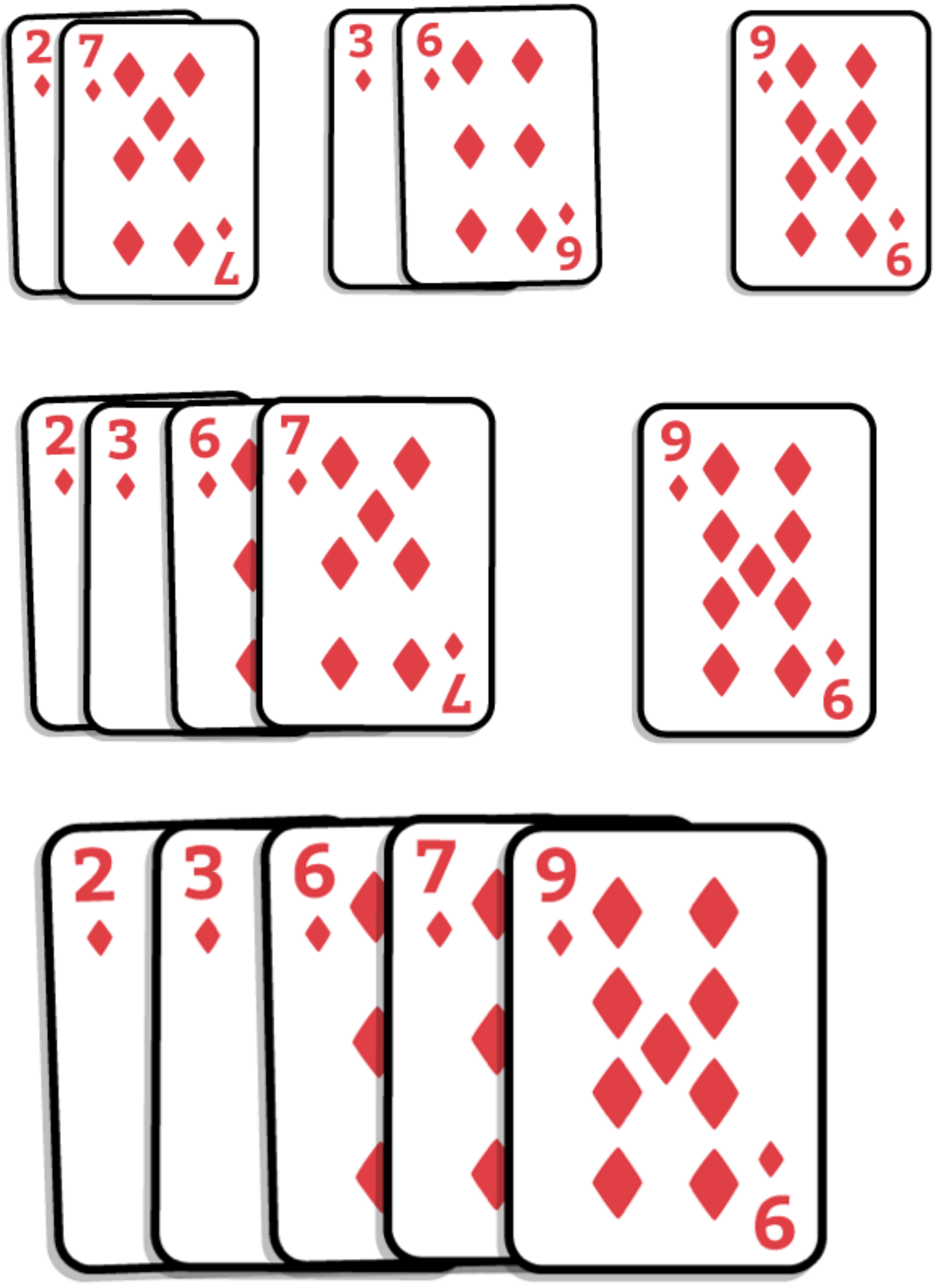


2. Keep splitting the resulting piles until you can’t split them anymore. In the end, you’ll have one card in each pile. Because a single card is always sorted, you now have a bunch of sorted piles:





3. Merge the piles in the reverse order in which you split them. During each merge, put the contents in sorted order. This is easy because each pile has already been sorted. You know that the smallest cards in any pile are on the left side:



In this chapter, you'll implement merge sort from scratch.

Implementation

The Merge sort consists of two main steps: split and merge. To implement them, open the starter project and start editing the **MergeSort.kt** file into the **mergesort** package.

Split

In the **MergeSort.kt** file copy the following code:

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
  if (this.size < 2) return this
  val middle = this.size / 2
  val left = this.subList(0, middle)
  val right = this.subList(middle, this.size)
  // ... more to come
}
```

When it comes to sorting algorithms, any list that’s shorter than two elements is already sorted. That’s why the first `if` is your best chance to exit fast and finish with the sorting.

You then split the list into halves. Splitting once isn’t enough — you have to keep splitting recursively until you can’t split anymore, which is when each subdivision contains only a single element.

To do this, update `mergeSort` as follows:

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
  // 1
  if (this.size < 2) return this
  val middle = this.size / 2

  // 2
  val left = this.subList(0, middle).mergeSort()
  val right = this.subList(middle, this.size).mergeSort()

  // ... still more to come
}
```

Here are the key elements of the code as it looks right now:

1. Recursion needs a **base case**, which you can also think of as an “exit

condition". In this case, the base case is when the list only has one element. Your previous *quick win* is now the cornerstone of the algorithm.
2. You're calling `mergeSort` on each of the sub-lists. This recursion continues to try to split the lists into smaller lists until the "exit condition" is fulfilled. In your case, it will split until the lists contain only one element.

There's still more work to do before your code will compile. Now that you've accomplished the splitting part, it's time to focus on merging.

Merge

Your final step is to merge the `left` and `right` lists. To keep things clean, you'll create a separate `merge` function to handle this.

The sole responsibility of the merge function is to take in two `sorted` lists and combine them while retaining the sort order. Add the following immediately below `mergeSort`:

```
private fun <T : Comparable<T>> merge(left: List<T>, right: List<T>): List<
  // 1
  var leftIndex = 0
  var rightIndex = 0
  // 2
  val result = mutableListOf<T>()
  // 3
  while (leftIndex < left.size && rightIndex < right.size) {
    val leftElement = left[leftIndex]
    val rightElement = right[rightIndex]
    // 4
    if (leftElement < rightElement) {
      result.add(leftElement)
      leftIndex += 1
    } else if (leftElement > rightElement) {
      result.add(rightElement)
      rightIndex += 1
    } else {
      result.add(leftElement)
      leftIndex += 1
      result.add(rightElement)
      rightIndex += 1
    }
  }
  // 5
  if (leftIndex < left.size) {
    result.addAll(left.subList(leftIndex, left.size))
  }
  if (rightIndex < right.size) {
    result.addAll(right.subList(rightIndex, right.size))
  }
  return result
}
```

Here's what's going on:

1. The `leftIndex` and `rightIndex` variables track your progress as you parse through the two lists.
2. The `result` list will house the combined lists.
3. Starting from the beginning, you compare the elements in the `left` and `right` lists sequentially. When you reach the end of either list, there's nothing else to compare.
4. The smaller of the two elements goes into the `result` list. If the elements are equal, they can both be added.
5. The first loop guarantees that either `left` or `right` is empty. Since both lists are sorted, this ensures that the leftover elements are greater than or equal to the ones currently in `result`. In this scenario, you can add the rest of the elements without comparison.

Finishing up

Complete `mergeSort` by calling `merge`. Because you call `mergeSort` recursively, the algorithm will split and sort both halves before merging them.

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
  if (this.size < 2) return this
  val middle = this.size / 2
  val left = this.subList(0, middle).mergeSort()
  val right = this.subList(middle, this.size).mergeSort()

  return merge(left, right)
}
```

This is the final version of the merge sort algorithm. Here's a summary of the key procedures of merge sort:

1. The strategy of merge sort is to *divide and conquer* so that you solve many small problems instead of one big problem.

2. It has two core responsibilities: a method to divide the initial list recursively, as well as a method to merge two lists.

3. The merging function should take two sorted lists and produce a single sorted list.

Finally, it's time to see this in action. Head back to **Main.kt** and test your merge sort.

Add the following code in `main()`:

```
"merge sort" example {
  val list = listOf(7, 2, 6, 3, 9)
  println("Original: $list")
  val result = list.mergeSort()
  println("Merge sorted: $result")
}
```

This outputs:

```
---Example of merge sort---
Original: [7, 2, 6, 3, 9]
Merge sorted: [2, 3, 6, 7, 9]
```

Performance

The best, worst and average time complexity of merge sort is *O*(*n* log *n*), which isn't too bad. If you're struggling to understand where *n* log *n* comes from, think about how the recursion works:

- As you recurse, you split a single list into two smaller lists. This means a list of size 2 will need one level of recursion, a list of size 4 will need two levels, a list of size 8 will need three levels, and so on. If you had a list of 1,024 elements, it would take 10 levels of recursively splitting in two to get down to 1024 single element lists. In general, if you have a list of size *n*, the number of levels is log2(*n*).
- A single recursion level will merge *n* elements. It doesn't matter if there are many small merges or one large one; the number of elements merged will still be *n* at each level. This means the cost of a single recursion is *O*(*n*).

This brings the total cost to *O*(log *n*) × *O*(*n*) = *O*(*n* log *n*).

The previous chapter's sort algorithms were **in-place** and used `swapAt` to move elements around. Merge sort, by contrast, allocates additional memory to do its work. How much? There are log2(*n*) levels of recursion, and at each level, *n* elements are used. That makes the total *O*(*n* log *n*) in space complexity. Merge sort is one of the hallmark sorting algorithms. It's relatively simple to understand, and serves as a great introduction to how to divide and conquer algorithms work. Merge sort is *O*(*n* log *n*), and this implementation requires *O*(*n* log *n*) of space. If you're clever with your bookkeeping, you can reduce the memory required to *O*(*n*) by discarding the memory that is not actively being used.

Challenges

Challenge 1: Iterables merge

Write a function that takes two sorted iterables and merges them into a single iterable. Here's the function signature to start:

```
fun <T : Comparable<T>> merge(
  first: Iterable<T>,
  second: Iterable<T>
): Iterable<T>
```

Solution 1

The tricky part of this challenge is the limited capabilities of `Iterable`. Traditional implementations of this algorithm rely on the abilities of `List` types to keep track of indices.

Since `Iterable` types have no notion of indices, you'll make use of their iterator. The `Iterator` in Kotlin has a slight inconvenience that you need to fix first. If there are no more elements in the iterable and you try to get the next one using `next()`, you'll get a `NoSuchElementException`. To make it friendlier for your algorithm, write the following extension function first:

```
private fun <T> Iterator<T>.nextOrNull(): T? {
  return if (this.hasNext()) this.next() else null
}
```

You can now use `nextOrNull()` to safely get the next element. If the returned value is `null`, this means there's no next element, and the iterable is over. This will be important later on.

Now, for `merge()`. Add the following code to your file:

```
fun <T : Comparable<T>> merge(
    first: Iterable<T>,
    second: Iterable<T>
): Iterable<T> {

  // 1
  val result = mutableListOf<T>()
  val firstIterator = first.iterator()
  val secondIterator = second.iterator()

  // 2
  if (!firstIterator.hasNext()) return second
  if (!secondIterator.hasNext()) return first

  // 3
  var firstEl = firstIterator.nextOrNull()
  var secondEl = secondIterator.nextOrNull()

  // 4
  while (firstEl != null && secondEl != null) {
    // more to come
  }
}
```

Setting up the algorithm involves the following steps:

1. Create a new container to store the merged iterable. It could be any class that implements `Iterable` but a `MutableList` is an easy to use choice, so go with that one. Then, grab the iterators of the first and second iterable. Iterators sequentially dispense values of the iterable via `next()`, but you'll use your own extension `nextOrNull()`.
2. Create two variables that are initialized as the first and second iterator's first value.
3. If one of the iterators didn't have a first value, it means the iterable it came from was empty, and you're done sorting. Simply return the other iterable.
4. This first `while` loop is where all of the comparisons are made to get the resulting iterable ordered. It only works while you still have values

in both iterables.

Using the iterators, you'll decide which element should be appended into the `result` list by comparing the first and second next values. Write the following inside the `while` loop:

```
when {
  // 1
  firstEl < secondEl -> {
    result.add(firstEl)
    firstEl = firstIterator.nextOrNull()
  }
  // 2
  secondEl < firstEl -> {
    result.add(secondEl)
    secondEl = secondIterator.nextOrNull()
  }
  // 3
  else -> {
    result.add(firstEl)
    result.add(secondEl)
    firstEl = firstIterator.nextOrNull()
    secondEl = secondIterator.nextOrNull()
  }
}
```

This is the main component of the merging algorithm. There are three situations possible, as you can see in the `when` statement:

1. If the first value is less than the second, you'll append the first value in `result` and seed the next value to be compared with by invoking `nextOrNull()` on the first iterator.
2. If the second value is less than the first, you'll do the opposite. You seed the next value to be compared by invoking `nextOrNull()` on the second iterator.
3. If they're equal, you append both the `first` and `second` values and seed both next values.

This will continue until one of the iterators runs out of elements to dispense. In that scenario, the iterator with elements left only has elements that are equal to or greater than the current values in `result`.

To add the rest of those values, write the following at the end of `merge()`:

```
while (firstEl != null) {
  result.add(firstEl)
  firstEl = firstIterator.nextOrNull()
}
while (secondEl != null) {
  result.add(secondEl)
  secondEl = secondIterator.nextOrNull()
}

return result
```

Confirm that this function works by writing the following in **Main.kt**:

```
"merge iterables" example {
  val list1 = listOf(1, 2, 3, 4, 5, 6, 7, 8)
  val list2 = listOf(1, 3, 4, 5, 5, 6, 7, 7)

  val result = merge(list1, list2)
  println("Merged: $result")
}
```

You’ll see the following console output:

```
---Example of merge iterables---
Merged: [1, 1, 2, 3, 3, 4, 4, 5, 5, 5, 6, 6, 7, 7, 7, 8]
```

Key points

- Merge sort is in the category of the divide and conquer algorithms.
- There are many implementations of merge sort, and you can have

different performance characteristics depending on the implementation.
- To do a comparison, in this chapter you sorted objects implementing the `Comparable<T>` interface but the same can be done providing a different implementation of `Comparator<T>`.