18 Quicksort Written by Márton Braun

In the preceding chapters, you learned to sort a list using comparison-based sorting algorithms, such as merge sort and heap sort.

Quicksort is another comparison-based sorting algorithm. Much like merge sort, it uses the same strategy of **divide and conquer**. One important feature of quicksort is choosing a **pivot** point. The pivot divides the list into three partitions:

```
[ elements < pivot | pivot | elements > pivot ]
```

In this chapter, you'll implement quicksort and look at various partitioning strategies to get the most out of this sorting algorithm.

Example

Open the starter project. Inside **QuicksortNaive.kt**, you'll see a naive implementation of quicksort:

```
fun <T : Comparable<T>> List<T>.quicksortNaive(): List<T> {
  if (this.size < 2) return this // 1

  val pivot = this[this.size / 2] // 2
  val less = this.filter { it < pivot } // 3
  val equal = this.filter { it == pivot }
  val greater = this.filter { it > pivot }
  return less.quicksortNaive() + equal + greater.quicksortNaive() // 4
}
```

This implementation recursively filters the list into three partitions. Here's how it works:

1. There must be more than one element in the list. If not, the list is

considered sorted.
2. Pick the **middle** element of the list as your pivot.
3. Using the pivot, split the original list into three partitions. Elements **less than**, **equal to** or **greater than** the pivot go into different buckets.
4. Recursively sort the partitions and then combine them.

Now, it's time to visualize the code above. Given the **unsorted** list below:

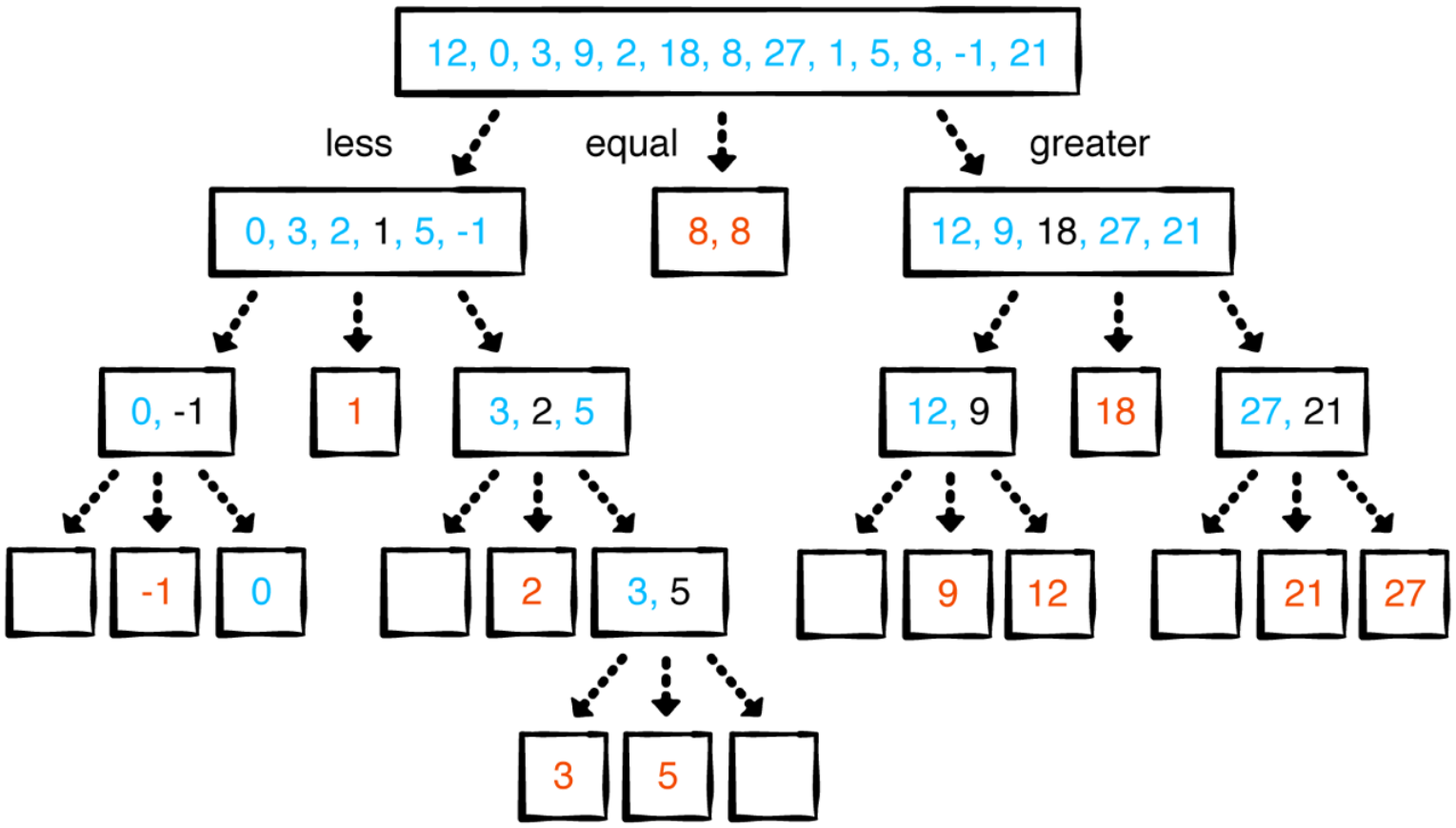
```
[12, 0, 3, 9, 2, 18, 8, 27, 1, 5, 8, -1, 21]
                      *
```

Your partition strategy in this implementation is to always select the **middle** element as the pivot. In this case, the element is **8**. Partitioning the list using this pivot results in the following partitions:

```
less: [0, 3, 2, 1, 5, -1]
equal: [8, 8]
greater: [12, 9, 18, 27, 21]
```

Notice that the three partitions aren't completely sorted yet. Quicksort will recursively divide these partitions into even smaller ones. The recursion will only halt when all partitions have either zero or one element.

Here's an overview of all the partitioning steps:



Each level corresponds with a recursive call to quicksort. Once recursion stops, the leafs are combined again, resulting in a fully sorted list:

```
[-1, 1, 2, 3, 5, 8, 8, 9, 12, 18, 21, 27]
```

While this naive implementation is easy to understand, it raises some issues and questions:

- Calling `filter` three times on the same list is not efficient.
- Creating a new list for every partition isn't space-efficient. Could you possibly sort in place?
- Is picking the middle element the best pivot strategy? What pivot strategy should you adopt?

Partitioning strategies

In this section, you'll look at partitioning strategies and ways to make this quicksort implementation more efficient. The first partitioning algorithm you'll look at is **Lomuto's algorithm**.

Lomuto's partitioning

Lomuto's partitioning algorithm always chooses the **last** element as the pivot. Time to look at how this works in code.

In your project, create a file named **QuicksortLomuto.kt** and add the following function declaration:

```
fun <T : Comparable<T>> MutableList<T>.partitionLomuto(
  low: Int,
  high: Int
): Int {
}
```

This function takes three arguments:

- the receiver (`this`) is the list you are partitioning.
- `low` and `high` set the range within the list you'll partition. This range will get smaller and smaller with every recursion.

The function returns the index of the pivot.

Now, implement the function as follows:

```
val pivot = this[high] // 1

var i = low // 2
for (j in low until high) { // 3
  if (this[j] <= pivot) { // 4
    this.swapAt(i, j) // 5
    i += 1
  }
}
this.swapAt(i, high) // 6
return i // 7
```

Here's what this code does:

1. Set the pivot. Lomuto always chooses the last element as the pivot.

2. The variable `i` indicates how many elements are **less** than the pivot. Whenever you encounter an element that is less than the pivot, you swap it with the element at index `i` and increase `i`.
3. Loop through all the elements from `low` to `high`, but not including `high` since it's the pivot.
4. Check to see if the current element is less than or equal to the pivot.
5. If it is, swap it with the element at index `i` and increase `i`.
6. Once done with the loop, swap the element at `i` with the pivot. The pivot always sits between the **less** and **greater** partitions.
7. Return the index of the pivot.

While this algorithm loops through the list, it divides the list into four regions:

1. `this.subList(low, i)` contains all elements <= `pivot`.
2. `this.subList(i, j)` contains all elements > `pivot`.
3. `this.subList(j, high)` are elements you have not compared yet.
4. `this[high]` is the pivot element.

```
[ values <= pivot | values > pivot | not compared yet | pivot ]
  low         i-1   i          j-1   j         high-1   high
```

Step-by-step

Looking at a few steps of the algorithm will help you get a better understanding of how it works.

Given the **unsorted** list below:

```
[12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
```

First, the last element **8** is selected as the pivot:

```
  0   1  2  3  4  5   6   7   8  9  10  11    12
```

```
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, |  8  ]
  low                                          high
  i
  j
```

Then, the first element **12** is compared to the pivot. It's not smaller than the pivot, so the algorithm continues to the next element:

```
   0  1  2  3  4  5   6   7   8  9  10  11   12
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, |  8  ]
  low                                          high
  i
     j
```

The second element **0** *is* smaller than the pivot, so it's swapped with the element currently at index `i` (**12**) and `i` is increased:

```
  0   1  2  3  4  5   6   7   8  9  10  11   12
[ 0, 12, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, |  8  ]
 low                                          high
      i
         j
```

The third element **3** is again smaller than the pivot, so another swap occurs:

```
  0   1   2  3  4  5   6   7   8  9  10  11   12
[ 0, 3, 12, 9, 2, 21, 18, 27, 1, 5, 8, -1, |  8  ]
 low                                          high
         i
            j
```

These steps continue until all but the pivot element have been compared. The resulting list is:

```
  0   1  2  3  4  5   6   7   8  9  10  11   12
[ 0, 3, 2, 1, 5, 8, -1, 27, 9, 12, 21, 18, |  8  ]
 low                                        high
                        i
```

Finally, the pivot element is swapped with the element currently at index `i`:

```
  0   1  2  3  4  5   6   7   8  9  10  11    12
[ 0, 3, 2, 1, 5, 8, -1 | 8 | 9, 12, 21, 18, |  27  ]
 low                                       high
                        i
```

Lomuto's partitioning is now complete. Notice how the pivot is between the two regions of elements less than or equal to the pivot and elements greater than the pivot.

In the naive implementation of quicksort, you created three new lists and filtered the unsorted lists three times. Lomuto's algorithm performs the partitioning in place. That's much more efficient!

With your partitioning algorithm in place, you can now implement quicksort adding the following to your **QuicksortLomuto.kt** file:

```
fun <T : Comparable<T>> MutableList<T>.quicksortLomuto(low: Int, high: Int)
  if (low < high) {
    val pivot = this.partitionLomuto(low, high)
    this.quicksortLomuto(low, pivot - 1)
    this.quicksortLomuto(pivot + 1, high)
  }
}
```

Here, you apply Lomuto's algorithm to partition the list into two regions. You then recursively sort these regions. Recursion ends once a region has

less than two elements.

You can try Lomuto's quicksort by adding the following to your **Main.kt** file, inside the `main()` function:

```
"Lomuto quicksort" example  {
  val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8)
  println("Original: $list")
  list.quicksortLomuto(0, list.size - 1)
  println("Sorted: $list")
}
```

Hoare's partitioning

Hoare's partitioning algorithm always chooses the **first** element as the pivot. So, how does this work in code?

In your project, create a file named **QuicksortHoare.kt** and add the following function:

```
fun <T : Comparable<T>> MutableList<T>.partitionHoare(low: Int, high: Int):
  val pivot = this[low] // 1
  var i = low - 1 // 2
  var j = high + 1
  while (true) {
    do { // 3
      j -= 1
    } while (this[j] > pivot)
    do { // 4
      i += 1
    } while (this[i] < pivot)
    if (i < j) { // 5
      this.swapAt(i, j)
    } else {
      return j // 6
    }
  }
}
```

Here's how it works:

1. Select the first element as the pivot.
2. Indexes `i` and `j` define two regions. Every index before `i` will be **less than or equal to** the pivot. Every index after `j` will be **greater than or equal to** the pivot.
3. Decrease `j` until it reaches an element that is not greater than the pivot.
4. Increase `i` until it reaches an element that is not less than the pivot.
5. If `i` and `j` have not overlapped, swap the elements.
6. Return the index that separates both regions.

> **Note**: The index returned from the partition does not necessarily have to be the index of the pivot element.

Step-by-step

Given the **unsorted** list below:

```
[  12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8   ]
```

First, **12** is set as the pivot. Then `i` and `j` will start running through the list, looking for elements that are not less than (in the case of `i`) or greater than (in the case of `j`) the pivot. `i` will stop at element **12** and `j` will stop at element **8**:

```
[  12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1,  8  ]
   p
   i                                          j
```

These elements are then swapped:

```
[  8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12 ]
```

```
   i                                   j
```

`i` and `j` now continue moving, this time stopping at **21** and **-1**:

```
[  8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12 ]
                    i                    j
```

Which are then swapped:

```
[  8, 0, 3, 9, 2, -1, 18, 27, 1, 5, 8, 21, 12 ]
                    i                    j
```

Next, **18** and **8** are swapped, followed by **27** and **5**.

After this swap the list and indices are as follows:

```
[  8, 0, 3, 9, 2, -1, 8, 5, 1, 27, 18, 21, 12 ]
                          i      j
```

The next time you move `i` and `j`, they will overlap:

```
[  8, 0, 3, 9, 2, -1, 8, 5, 1, 27, 18, 21, 12 ]
                             j   i
```

Hoare's algorithm is now complete, and index `j` is returned as the separation between the two regions.

There are far fewer swaps here compared to Lomuto's algorithm. Isn't that nice?

You can now implement a `quicksortHoare` function:

```
fun <T : Comparable<T>> MutableList<T>.quicksortHoare(low: Int, high: Int)
```

```
  if (low < high) {
    val p = this.partitionHoare(low, high)
    this.quicksortHoare(low, p)
    this.quicksortHoare(p + 1, high)
  }
}
```

Try it out by adding the following in your **Main.kt**:

```
"Hoare quicksort" example {
  val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8)
  println("Original: $list")
  list.quicksortHoare( 0, list.size - 1)
  println("Sorted: $list")
}
```

Effects of a bad pivot choice

The most important part of implementing quicksort is choosing the right partitioning strategy.

You've looked at three different partitioning strategies:

1. Choosing the middle element as a pivot.
2. **Lomuto**, or choosing the last element as a pivot.
3. **Hoare**, or choosing the first element as a pivot.

What are the implications of choosing a bad pivot?

Starting with the following unsorted list:

```
[8, 7, 6, 5, 4, 3, 2, 1]
```

If you use Lomuto's algorithm, the pivot will be the last element, `1`. This results in the following partitions:

```
less: [ ]
equal: [1]
greater: [8, 7, 6, 5, 4, 3, 2]
```

An ideal pivot would split the elements evenly between the **less than** and **greater than** partitions. Choosing the first or last element of an already sorted list as a pivot makes quicksort perform much like **insertion sort**, which results in a worst-case performance of $O(n^2)$. One way to address this problem is by using the **median of three** pivot selection strategy. Here, you find the median of the first, middle and last element in the list, and use that as a pivot. This prevents you from picking the highest or lowest element in the list.

Create a new file named **QuicksortMedian.kt** and add the following function:

```
fun <T : Comparable<T>> MutableList<T>.medianOfThree(low: Int, high: Int):
  val center = (low + high) / 2
  if (this[low] > this[center]) {
    this.swapAt(low, center)
  }
  if (this[low] > this[high]) {
    this.swapAt(low, high)
  }
  if (this[center] > this[high]) {
    this.swapAt(center, high)
  }
  return center
}
```

Here, you find the median of `this[low]`, `this[center]` and `this[high]` by sorting them. The median will end up at index `center`, which is what the function returns.

Next, you'll implement a variant of Quicksort using this median of three:

```
fun <T : Comparable<T>> MutableList<T>.quickSortMedian(low: Int, high: Int)
  if (low < high) {
    val pivotIndex = medianOfThree(low, high)
    this.swapAt(pivotIndex, high)
    val pivot = partitionLomuto(low, high)
    this.quicksortLomuto(low, pivot - 1)
    this.quicksortLomuto(pivot + 1, high)
  }
}
```

This is a variation on `quicksortLomuto` that adds a median of three as a first step.

Try this by adding the following in your playground:

```
"Median of three quicksort" example {
  val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8)
  println("Original: $list")
  list.quickSortMedian( 0, list.size - 1)
  println("Sorted: $list")
}
```

This is definitely an improvement, but can you do even better?

Dutch national flag partitioning

A problem with Lomuto's and Hoare's algorithms is that they don't handle duplicates really well. With Lomuto's algorithm, duplicates end up in the *less than* partition and aren't grouped together. With Hoare's algorithm, the situation is even worse as duplicates can be all over the place.

A solution to organize duplicate elements is using **Dutch national flag partitioning**. This technique is named after the Dutch flag, which has three bands of colors: red, white and blue. This is similar to how you create three partitions. Dutch national flag partitioning is a good technique to use if you have a lot of duplicate elements.

Create a file named **QuicksortDutchFlag.kt** and add the following function:

```
fun <T : Comparable<T>> MutableList<T>.partitionDutchFlag(
  low: Int,
  high: Int,
  pivotIndex: Int
): Pair<Int, Int> {
  val pivot = this[pivotIndex]
  var smaller = low // 1
  var equal = low // 2
  var larger = high // 3
  while (equal <= larger) { // 4
    if (this[equal] < pivot) {
      this.swapAt(smaller, equal)
      smaller += 1
      equal += 1
    } else if (this[equal] == pivot) {
      equal += 1
    } else {
      this.swapAt(equal, larger)
      larger -= 1
    }
  }
  return Pair(smaller, larger) // 5
}
```

You'll adopt the same strategy as Lomuto's partition by choosing the last element as the `pivotIndex`. Here's how it works:

1. Whenever you encounter an element that is less than the pivot, move it to index `smaller`. This means that all elements that come before this index are less than the pivot.
2. Index `equal` points to the next element to compare. Elements that are equal to the pivot are skipped, which means that all elements between `smaller` and `equal` are equal to the pivot.
3. Whenever you encounter an element that is greater than the pivot,

move it to index `larger`. This means that all elements that come after this index are greater than the pivot.

4. The main loop compares elements and swaps them if needed. This continues until index `equal` moves past index `larger`, meaning all elements have been moved to their correct partition.
5. The algorithm returns indices `smaller` and `larger`. These point to the first and last elements of the middle partition.

Step-by-step

Looking at an example using the **unsorted** list below:

```
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8 ]
```

Since this algorithm is independent of a pivot selection strategy, you'll adopt Lomuto and pick the last element **8**.

> **Note**: I challenge you to try a different strategy, such as median of three.

Next, you set up the indices `smaller`, `equal` and `larger`:

```
[12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
  s
  e
                                            l
```

The first element to be compared is **12**. Since it's larger than the pivot, it's swapped with the element at index `larger` and this index is decremented.

Note that index `equal` is not incremented so the element that was swapped in (**8**) is compared next:

```
[8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
 s
```

```
 e
                                                  l
```

Remember that the pivot you selected is still **8**. **8** is equal to the pivot, so you increment `equal`:

```
[8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
 s
    e
                                                  l
```

0 is smaller than the pivot, so you swap the elements at `equal` and `smaller` and increase both pointers:

```
[0, 8, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
    s
       e
                                                  l
```

And so on.

Note how `smaller`, `equal` and `larger` partition the list:

- Elements in `this.subList(low, smaller)` are smaller than the pivot.
- Elements in `this.subList(smaller, equal)` are equal to the pivot.
- Elements in `this.subList(larger, high)` are larger than the pivot.
- Elements in `this.subList(equal, larger + 1)` haven't been compared yet.

To understand how and when the algorithm ends, let's continue from the second-to-last step:

```
[0, 3, -1, 2, 5, 8, 8, 27, 1, 18, 21, 9, 12]
                   s
                          e
                                                  l
```

```
                        l
```

Here, **27** is being compared. It's greater than the pivot, so its swapped with **1** and index `larger` is decremented:

```
[0, 3, -1, 2, 5, 8, 8, 1, 27, 18, 21, 9, 12]
                  s
                        e
                        l
```

Even though `equal` is now equal to `larger`, the algorithm isn't complete.

The element currently at `equal` hasn't been compared yet. It's smaller than the pivot, so it's swapped with **8** and both indices `smaller` and equal are incremented:

```
[0, 3, -1, 2, 5, 1, 8, 8, 27, 18, 21, 9, 12]
                     s
                           e
                        l
```

Indices `smaller` and `larger` now point to the first and last elements of the middle partition. By returning them, the function clearly marks the boundaries of the three partitions.

You're now ready to implement a new version of quicksort using Dutch national flag partitioning:

```
fun <T : Comparable<T>> MutableList<T>.quicksortDutchFlag(low: Int, high: I
  if (low < high) {
    val middle = partitionDutchFlag(low, high, high)
    this.quicksortDutchFlag(low, middle.first - 1)
    this.quicksortDutchFlag(middle.second + 1, high)
  }
}
```

Notice how recursion uses the `middle.first` and `middle.second` indices to determine the partitions that need to be sorted recursively. Because the elements equal to the pivot are grouped together, they can be excluded from the recursion.

Try out your new quicksort by adding the following in your **Main.kt**:

```
"Dutch flag quicksort" example {
  val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8)
  println("Original: $list")
  list.quicksortDutchFlag( 0, list.size - 1)
  println("Sorted: $list")
}
```

That's it!

Challenges

Challenge 1: Not using recursion

In this chapter, you learned how to implement quicksort recursively. Your challenge is to implement it iteratively. Choose any partition strategy you learned in this chapter.

Solution 1

You implemented quicksort recursively, which means you know what quicksort is all about. So, how might you do it iteratively? This solution uses Lomuto's partition strategy.

This function takes in a list and the range between `low` and `high`. You're going to leverage a stack to store pairs of `start` and `end` values.

```
fun <T : Comparable<T>> MutableList<T>.quicksortIterativeLomuto(low: Int, h
  val stack = stackOf<Int>() // 1
  stack.push(low) // 2
```

```
  stack.push(high)

  while (!stack.isEmpty) { // 3
    // 4
    val end = stack.pop() ?: continue
    val start = stack.pop() ?: continue
    val p = this.partitionLomuto(start, end) // 5
    if ((p - 1) > start) {    // 6
      stack.push(start)
      stack.push(p - 1)
    }
    if ((p + 1) < end) {    // 7
      stack.push(p + 1)
      stack.push(end)
    }
  }
}
```

Here's how the solution works:

1. Create a stack that stores indices.
2. Push the starting `low` and `high` boundaries on the stack to initiate the algorithm.
3. As long as the stack is not empty, quicksort is not complete.
4. Get the pair of `start` and `end` indices from the stack.
5. Perform Lomuto's partitioning with the current `start` and `end` index. Recall that Lomuto picks the last element as the pivot, and splits the partitions into three parts: elements that are less than the pivot, the pivot, and finally elements that are greater than the pivot.
6. Once the partitioning is complete, check and add the lower bound's `start` and `end` indices to later partition the lower half.
7. Similarly, check and add the upper bound's `start` and `end` indices to later partition the upper half.

You're simply using the stack to store a pair of `start` and `end` indices to perform the partitions.

Now, check to see if your iterative version of quicksort works:

```
"Iterative Lomuto quicksort" example {
  val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8)
  println("Original: $list")
  list.quicksortIterativeLomuto( 0, list.size - 1)
  println("Sorted: $list")
}
```

Challenge 2: Provide an explanation

Explain when and why you would use merge sort over quicksort.

Solution 2

- Merge sort is preferable over quicksort when you need stability. Merge sort is a stable sort and guarantees $O(n \log n)$. This is not the case with quicksort, which isn't stable and can perform as bad as $O(n^2)$.
- Merge sort works better for larger data structures or data structures where elements are scattered throughout memory. Quicksort works best when elements are stored in a contiguous block.

Key points

- The naive implementation creates a new list on every filter function; this is inefficient. All other strategies sort in place.
- **Lomuto's** partitioning chooses the last element as the pivot.
- **Hoare's** partitioning chooses the first element as its pivot.
- An ideal pivot would split the elements evenly between partitions.
- Choosing a bad pivot can cause quicksort to perform in $O(n^2)$.
- **Median of three** finds the pivot by taking the median of the first, middle and last element.
- The **Dutch national flag** partitioning strategy helps to organize duplicate elements in a more efficient way.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.