11 Binary Search Written by Irina Galata

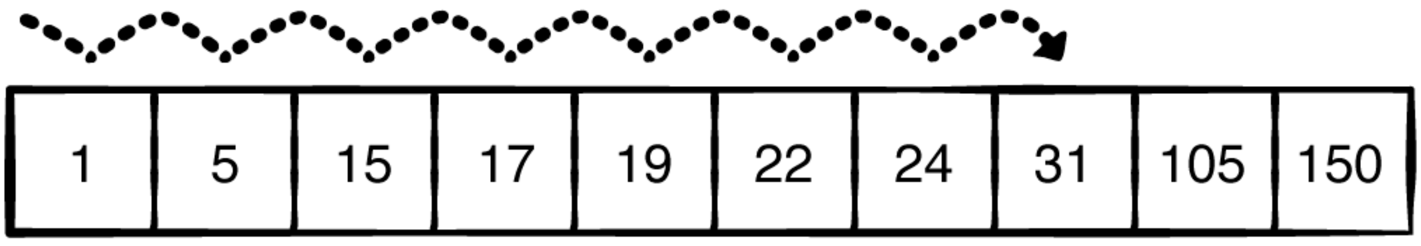
Binary search is one of the most efficient searching algorithms with a time complexity of *O*(*log n*). This is comparable with searching for an element inside a **balanced** binary search tree.

Two conditions need to be met before you can use binary search:

- The collection must be able to perform index manipulation in constant time. Kotlin collections that can do this include the `Array` and the `ArrayList`.
- The collection must be **sorted**.

Example

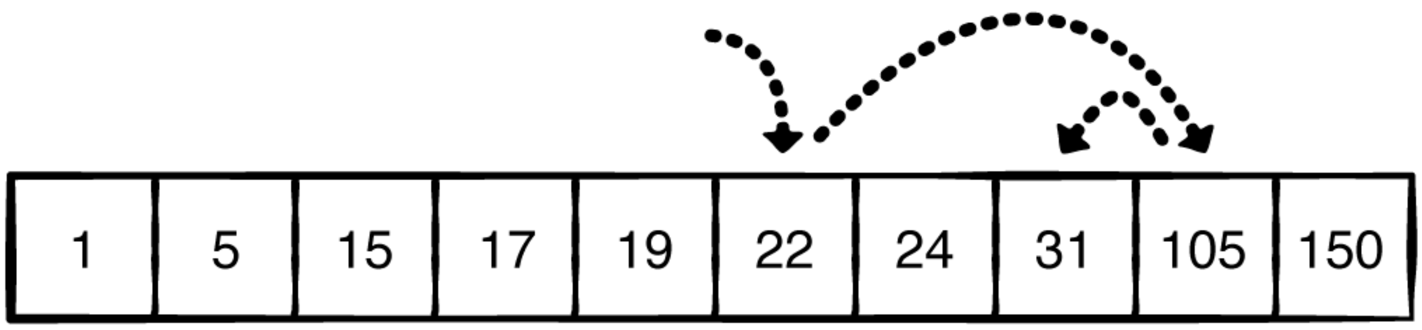
The benefits of binary search are best illustrated by comparing it with linear search. The `ArrayList` type uses linear search to implement its `indexOf()` method. This means that it traverses through the entire collection or until it finds the element.



Linear search for the value 31.

Binary search handles things differently by taking advantage of the fact that the collection is already sorted.

Here's an example of applying binary search to find the value **31**:

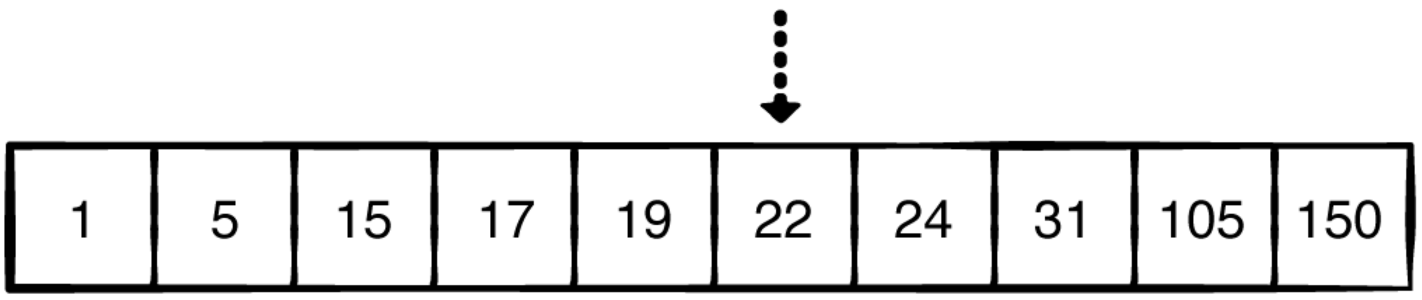


Binary search for the value 31.

Instead of eight steps to find 31, it only takes three. Here’s how it works:

Step 1: Find middle index

The first step is to find the middle index of the collection, like so:



Step 2: Check the element at the middle index

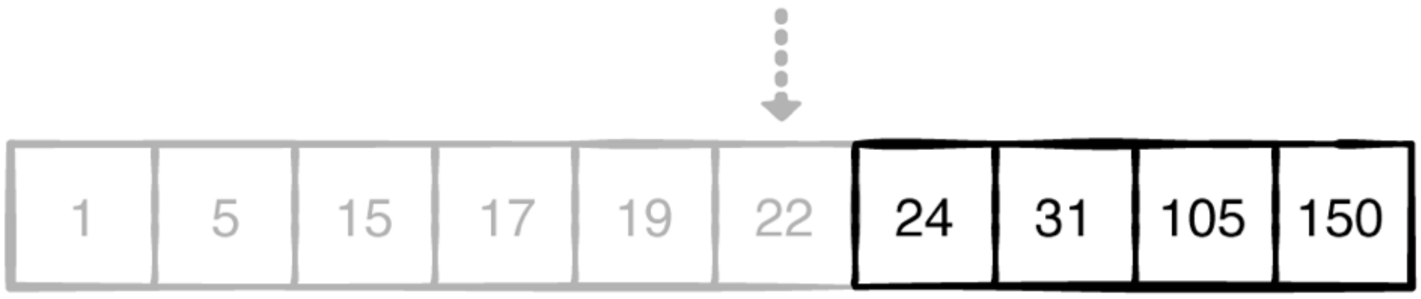
The next step is to check the element stored at the middle index. If it matches the value you’re looking for, you return the index. Otherwise, you’ll continue to Step 3.

Step 3: Recursively call binary Search

The final step is to recursively call binary search. However, this time, you’ll only consider the elements exclusively to the **left** or **right** of the middle index, depending on the value you’re searching for. If the value you’re searching for is less than the middle value, you search the left subsequence. If it’s greater than the middle value, you search the right subsequence.

Each step effectively removes half of the comparisons you would otherwise need to perform.

In the example where you're looking for the value **31** (which is greater than the middle element **22**), you apply binary search on the right subsequence.



You continue these three steps until you can no longer split the collection into left and right halves, or until you find the value inside the collection.

Binary search achieves an *O*(*log n*) time complexity this way.

Implementation

Open the starter project for this chapter. Create a new file named **BinarySearch.kt**. Add the following to the file:

```
// 1
fun <T : Comparable<T>> ArrayList<T>.binarySearch(
    value: T,
    range: IntRange = indices // 2
): Int? {
  // more to come
}
```

Things are fairly simple, so far:

1. You want `binarySearch` to be available on any `ArrayList`, so you define it as a generic extension function.
2. Binary search is recursive, so you need to be able to pass in a range to search. The parameter `range` is made optional by giving it a default value; this lets you start the search without having to specify a range.

In this case, the `indices` property of `ArrayList` is used, which covers all valid indexes of the collection.

Next, implement `binarySearch`:

```
// 1
if (range.first > range.last) {
  return null
}

// 2
val size = range.last - range.first + 1
val middle = range.first + size / 2

return when {
  // 3
  this[middle] == value -> middle
  // 4
  this[middle] > value -> binarySearch(value, range.first until middle)
  else -> binarySearch(value, (middle + 1)..range.last)
}
```

Here are the steps:

1. First, you check if the range contains at least one element. If it doesn't, the search has failed and you return `null`.
2. Now that you're sure you have elements in the range, you find the middle index in the range.
3. You then compare the value at this index with the value you're searching for. If they match, you return the middle index.
4. If not, you recursively search either the left or right half of the collection, excluding the middle item in both cases.

That wraps up the implementation of binary search. Go back to `main()` to test it out:

```
"binary search" example {
  val array = arrayListOf(1, 5, 15, 17, 19, 22, 24, 31, 105, 150)

  val search31 = array.indexOf(31)
  val binarySearch31 = array.binarySearch(31)

  println("indexOf(): $search31")
  println("binarySearch(): $binarySearch31")
}
```

You'll see the following output in the console:

```
---Example of binary search---
indexOf(): 7
binarySearch(): 7
```

This represents the index of the value you're looking for.

Binary search is a powerful algorithm to learn, and it comes up often in programming interviews. Whenever you read something along the lines of "Given a sorted array...", consider using the binary search algorithm. Also, if you're given a problem that looks like it's going to be $O(n^2)$ to search, consider doing some upfront sorting. With upfront sorting, you can use binary searching to reduce complexity to the cost of the sort at $O(n \log n)$.

Challenges

Challenge 1: Find the range

Write a function that searches a **sorted** `ArrayList` and finds the range of indices for a particular element. For example:

```
val array = arrayListOf(1, 2, 3, 3, 3, 4, 5, 5)
val indices = array.findIndices(3)
println(indices)
```

`findIndices` should return the range `2..4`, since those are the start and end indices for the value `3`.

Solution 1

An unoptimized but elegant solution is quite simple:

```
fun <T : Comparable<T>> ArrayList<T>.findIndices(
    value: T
): IntRange? {
  val startIndex = indexOf(value)
  val endIndex = lastIndexOf(value)

  if (startIndex == -1 || endIndex == -1) {
    return null
  }

  return startIndex..endIndex
}
```

The time complexity of this solution is *O*(*n*), which may not seem to be a cause for concern. However, you can optimize the solution to an *O*(_log *n*) time complexity solution.

Binary search is an algorithm that identifies values in a **sorted** collection, so keep that in mind whenever the problem promises a sorted collection. The binary search you implemented in the theory chapter is not powerful enough to reason whether the index is a start or end index. You'll modify that binary search to accommodate for this new rule.

Write the following in **BinarySearch.kt**:

```
fun <T : Comparable<T>> ArrayList<T>.findIndices(
    value: T
): IntRange? {
  val startIndex = startIndex(value, 0..size) ?: return null
  val endIndex = endIndex(value, 0..size) ?: return null
  return startIndex until endIndex
}

private fun <T : Comparable<T>> ArrayList<T>.startIndex(
    value: T,
    range: IntRange
): Int? {
 // more to come
}

private fun <T : Comparable<T>> ArrayList<T>.endIndex(
    value: T,
    range: IntRange
): Int? {
 // more to come
}
```

This time, `findIndices` will use specialized binary searches. `startIndex` and `endIndex` will be the ones that do the heavy lifting with a customized binary search.

You'll modify binary search so that it also inspects whether the adjacent value — depending on whether you're looking for the start or end index — is different from the current value.

Update the `startIndex` function to the following:

```
private fun <T : Comparable<T>> ArrayList<T>.startIndex(
    value: T,
    range: IntRange
): Int? {
  // 1
  val middleIndex = range.start + (range.last - range.start + 1) / 2

  // 2
  if (middleIndex == 0 || middleIndex == size - 1) {
    return if (this[middleIndex] == value) {
      middleIndex
    } else {
      null
    }
  }

  // 3
  return if (this[middleIndex] == value) {
    if (this[middleIndex - 1] != value) {
      middleIndex
    } else {
      startIndex(value, range.start until middleIndex)
    }
  } else if (value < this[middleIndex]) {
    startIndex(value, range.start until middleIndex)
  } else {
    startIndex(value, (middleIndex + 1)..range.last)
  }
}
```

Here's what you do with this code:

1. You start by calculating the middle value of the indices contained in `range`.
2. This is the base case of this recursive function. If the middle index is the first or last accessible index of the array, you don't need to call binary search any further. You'll determine whether or not the current index is a valid bound for the given value.
3. Here, you check the value at the index and make your recursive calls. If the value at `middleIndex` is equal to the value you're given, you check to see if the predecessor is also the same value. If it isn't, you know that you've found the starting bound. Otherwise, you'll continue by recursively calling `startIndex`.

The `endIndex` method is similar. Update the `endIndex` implementation to the following:

```
private fun <T : Comparable<T>> ArrayList<T>.endIndex(
    value: T,
    range: IntRange
): Int? {
  val middleIndex = range.start + (range.last - range.start + 1) / 2

  if (middleIndex == 0 || middleIndex == size - 1) {
    return if (this[middleIndex] == value) {
      middleIndex + 1
    } else {
      null
    }
  }

  return if (this[middleIndex] == value) {
    if (this[middleIndex + 1] != value) {
      middleIndex + 1
    } else {
      endIndex(value, (middleIndex + 1)..range.last)
    }
  } else if (value < this[middleIndex]) {
    endIndex(value, range.start until middleIndex)
  } else {
    endIndex(value, (middleIndex + 1)..range.last)
  }
}
```

Test your solution by writing the following in `main()`:

```
"binary search for a range" example {
  val array = arrayListOf(1, 2, 3, 3, 3, 4, 5, 5)
  val indices = array.findIndices(3)
  println(indices)
}
```

You'll see the following output in the console:

```
---Example of binary search for a range---
2..4
```

This improves the time complexity from the previous $O(n)$ to $O(\log n)$.

Key points

- Binary search is only a valid algorithm on sorted collections.
- Sometimes, it may be beneficial to sort a collection just to leverage the binary search capability for looking up elements.
- The `indexOf` method of arrays uses linear search, which has an $O(n)$ time complexity. Binary search has an $O(log\ n)$ time complexity, which scales much better for large data sets.

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.