14 O(n²) Sorting Algorithms Written by Márton Braun

$O(n^2)$ time complexity is not great performance, but the sorting algorithms in this category are easy to understand and useful in some scenarios. These algorithms are space efficient, and they only require constant $O(1)$ additional memory space. For small data sets, these sorts compare favorably against more complex sorts. It's usually not recommended to use $O(n^2)$ in production code, but you'll need to start somewhere, and these algorithms are a great place to start.

In this chapter, you'll look at the following sorting algorithms:

- Bubble sort.
- Selection sort.
- Insertion sort.

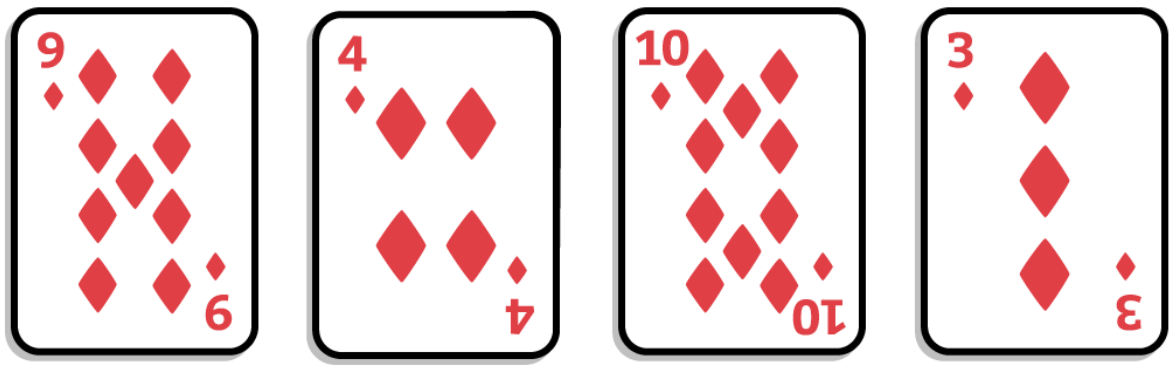
All of these are **comparison-based** sorting methods. In other words, they rely on a comparison method, such as the less than operator, to order elements. You measure a sorting technique's general performance by counting the number of times this comparison gets called.

Bubble sort

One of the simplest sorts is the **bubble sort**. The bubble sort repeatedly compares adjacent values and swaps them, if needed, to perform the sort. The larger values in the set will, therefore, *bubble up* to the end of the collection.

Example

Consider the following hand of cards, and suppose you want to sort them in ascending order:

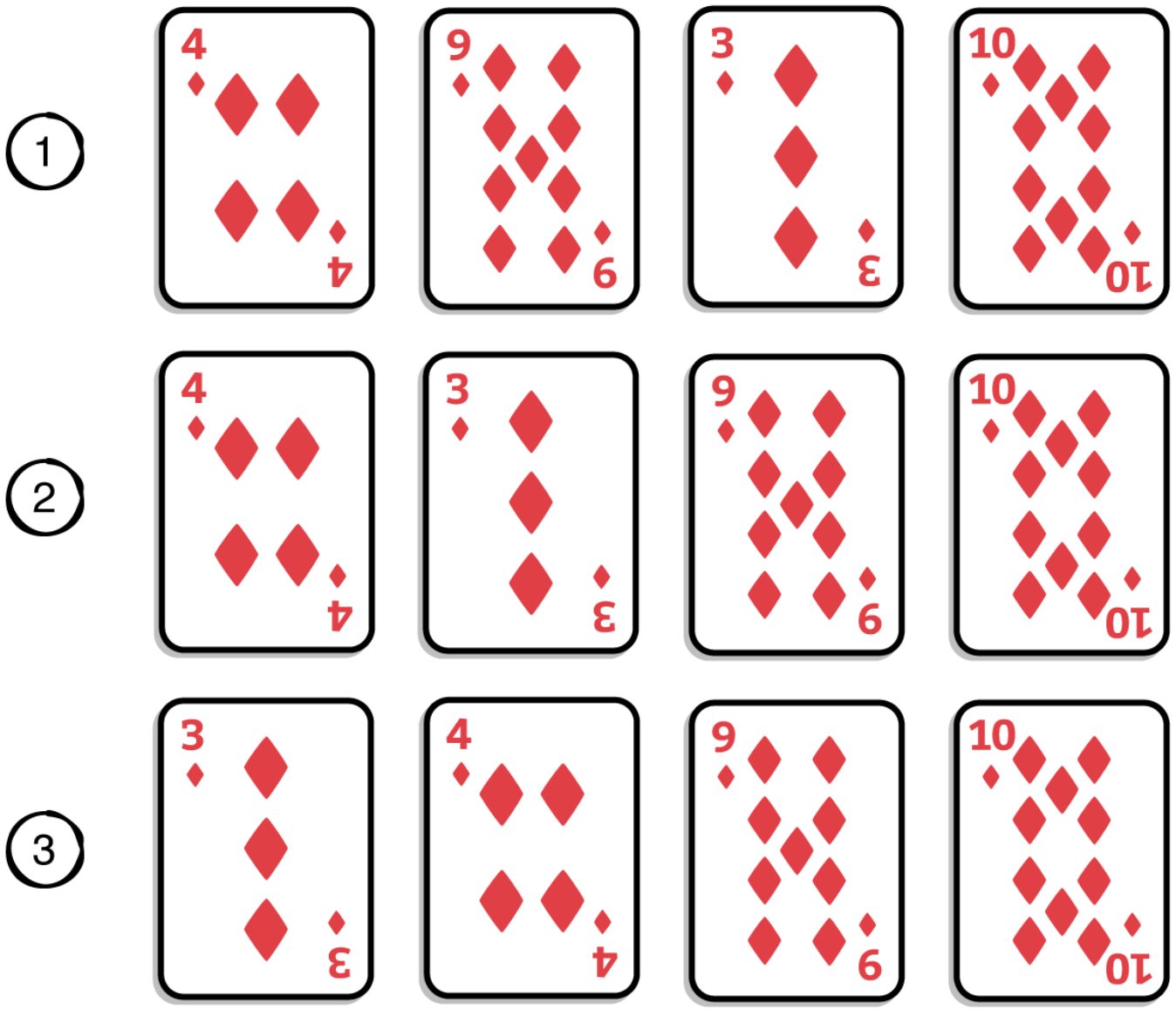


A single pass of the bubble sort algorithm consists of the following steps:

- Start at the beginning of the collection. Compare 9 with the next value in the array, which is 4. Since you're sorting in ascending order, and 9 is greater than 4, you need to swap these values. The collection then becomes `[4, 9, 10, 3]`.
- Move to the next index in the collection. You're now comparing 9 and 10. These are in order, so there's nothing to do.
- Move to the next index in the collection. Compare 10 and 3. You need to swap these values. The collection then becomes `[4, 9, 3, 10]`.
- You'll stop there as it's pointless to move to the next position because there's nothing left to sort.

A single pass of the algorithm seldom results in a complete ordering. You can see for yourself that the cards above are not yet completely sorted. 9 is greater than 3, but the 3 of diamonds still comes before the 9 of diamonds. It will, however, cause the largest value (10) to bubble up to the end of the collection.

Subsequent passes through the collection do the same for 9 and 4 respectively. Here's an illustration of the passes. You can see that after each pass, the collection has fewer cards in the wrong position.



The sort is only complete when you can perform a full pass over the collection without having to swap any values. At worst, this will require *n*-1 passes, where *n* is the count of members in the collection. For the cards example, you had four cards, so you needed three passes to make sure everything's in order.

Implementation

To get started, open the starter project for this chapter. Because you already know that you'll need to do a lot of swapping, the first thing you need to do is write an extension for `ArrayList` to swap elements.

Open **Utils.kt** and add:

```
fun <T> ArrayList<T>.swapAt(first: Int, second: Int) {
  val aux = this[first]
  this[first] = this[second]
  this[second] = aux
```

```
}
```

With this useful addition, start working on the `bubbleSort()` extension.

Since `ArrayList` is mutable, you're free to swap its elements. In **src ▸ bubblesort**, create a new file named **BubbleSort.kt**. Add the following function:

```
fun <T : Comparable<T>> ArrayList<T>.bubbleSort(showPasses: Boolean = false
  // 1
  if (this.size < 2) return
  // 2
  for (end in this.lastIndex downTo 1) {
    var swapped = false
    // 3
    for (current in 0 until end) {
      if (this[current] > this[current + 1]) {
        // 4
        this.swapAt(current, current + 1)
        swapped = true
      }
    }
    // 5
    if(showPasses) println(this)
    // 6
    if (!swapped) return
  }
}
```

Here's how it works:

1. There's no need to sort the collection when it has less than two elements. One element is sorted by itself; zero elements don't require an order.

2. A single-pass will bubble the largest value to the end of the collection. Every pass needs to compare one less value than in the

previous pass, so you shorten the array by one with each pass.

3. This loop performs a single pass starting from the first element and going up until the last element not already sorted. It compares every element with the adjacent value.

4. Next, the algorithm swaps the values if needed and marks this in `swapped`. This is important later because it'll allow you to exit the sort as early as you can detect the list is sorted.

5. This prints out how the list looks after each pass. This step has nothing to do with the sorting algorithm, but it will help you visualize how it works. You can remove it (and the function parameter) after you understand the sorting algorithm.

6. If no values were swapped this pass, the collection is assumed sorted, and you can exit early.

> **Note:** `lastIndex` is a handy extension to get the last valid index of a collection. Its value is always `size - 1`.

Try it out. Open **Main.kt** and write the following inside `main()`:

```
"bubble sort" example {
  val list = arrayListOf(9, 4, 10, 3)
  println("Original: $list")
  list.bubbleSort(true)
  println("Bubble sorted: $list")
}
```

Since you've set the `showPasses` parameter, you'll see the following output:

```
---Example of bubble sort---
Original: [9, 4, 10, 3]
[4, 9, 3, 10]
[4, 3, 9, 10]
```

```
[3, 4, 9, 10]
Bubble sorted: [3, 4, 9, 10]
```

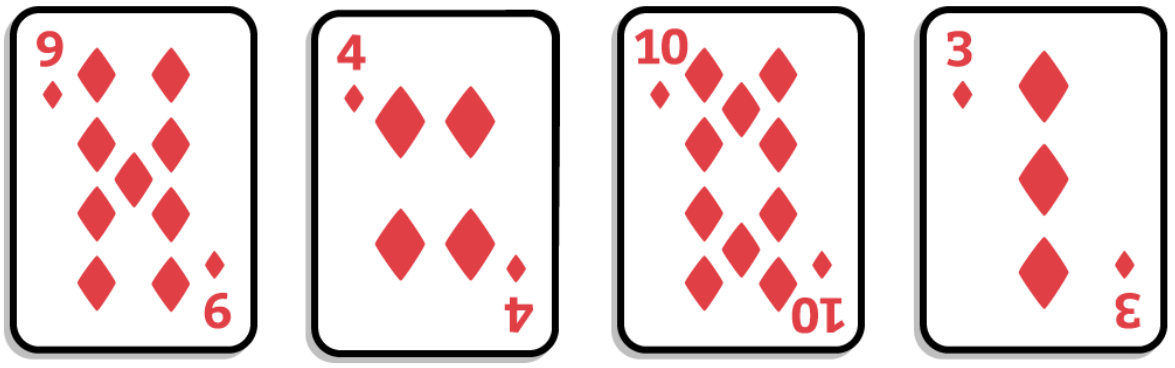
Bubble sort has a *best* time complexity of $O(n)$ if it's already sorted, and a *worst* and *average* time complexity of $O(n^2)$, making it one of the *least* appealing sorts.

Selection sort

Selection sort follows the basic idea of bubble sort but improves upon this algorithm by reducing the number of `swapAt` operations. Selection sort only swaps at the end of each pass. You'll see how that works in the following implementation.

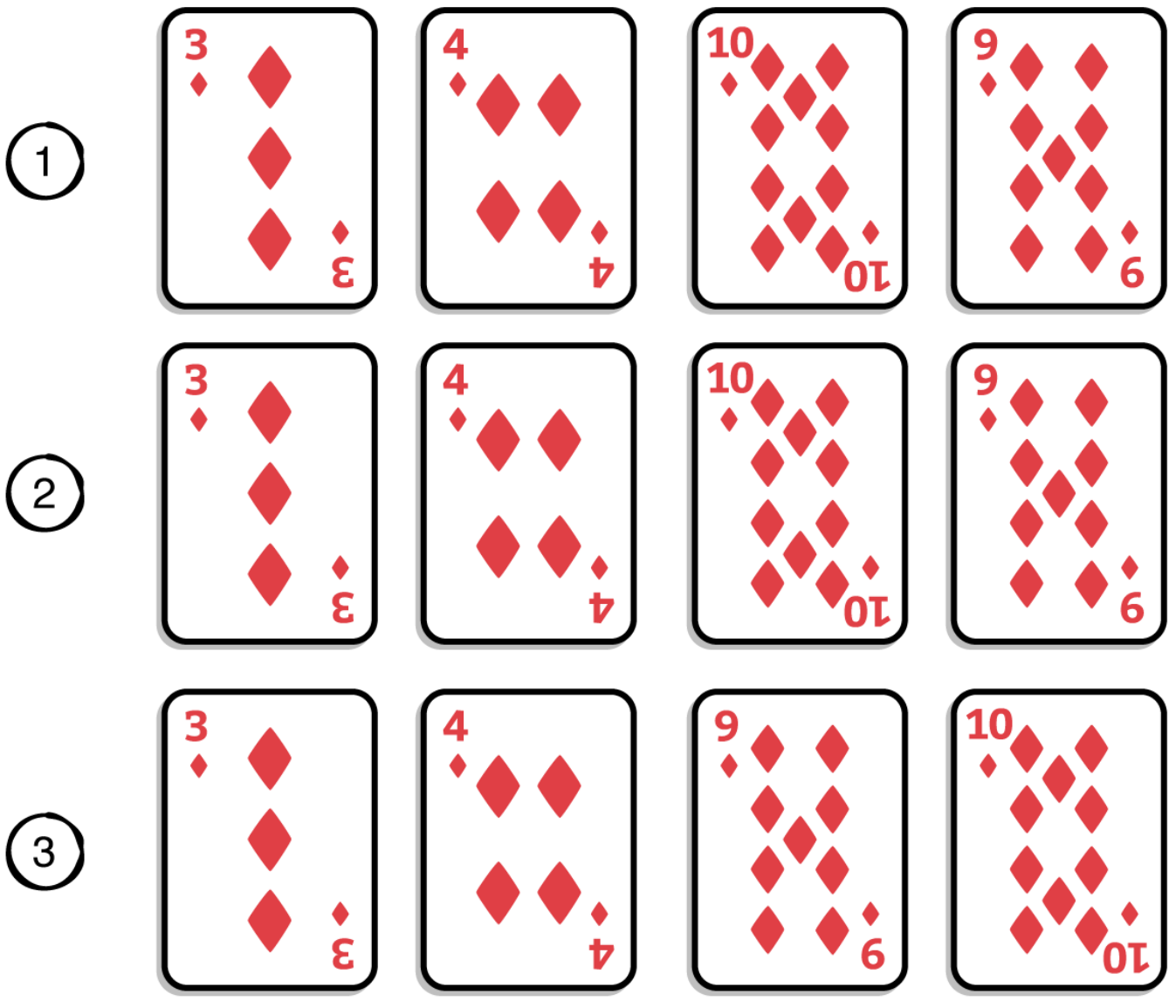
Example

Assume you have the following hand of cards:



During each pass, selection sort finds the lowest unsorted value and swaps it into place:

1. 3 is found as the lowest value, so it's swapped with 9.
2. The next lowest value is 4, and it's already in the right place.
3. Finally, 9 is swapped with 10.



Implementation

In **src ▸ selectionsort**, create a new file named **SelectionSort.kt**. Since you added the `swapAt()` extension for the bubble sort, you'll leverage it here too.

> **Note**: If you didn't already add swapAt(), go back and copy it into **Utils.kt**.

After you confirm that you can swap list elements, write the following inside the file:

```
fun <T : Comparable<T>> ArrayList<T>.selectionSort(showPasses: Boolean = fa
  if (this.size < 2) return
  // 1
  for (current in 0 until this.lastIndex) {
    var lowest = current
    // 2
```

```
    for (other in (current + 1) until this.size) {
      if (this[other] < this[lowest]) {
        lowest = other
      }
    }
    // 3
    if (lowest != current) {
      this.swapAt(lowest, current)
    }
    // 4
    if (showPasses) println(this)
  }
}
```

Here's what's going on:

1. You perform a pass for every element in the collection, except for the last one. There's no need to include the last element because if all other elements are in their correct order, the last one will be as well.
2. In every pass, you go through the remainder of the collection to find the element with the lowest value.
3. If that element is not the current element, swap them.
4. This optional step shows you how the list looks after each step when you call the function with `showPasses` set to `true`. You can remove this and the parameter once you understand the algorithm.

Try it out. In your code, add the following:

```
"selection sort" example {
  val list = arrayListOf(9, 4, 10, 3)
  println("Original: $list")
  list.selectionSort(true)
  println("Bubble sorted: $list")
}
```

You'll see the following output in your console:

```
---Example of selection sort---
Original: [9, 4, 10, 3]
[3, 4, 10, 9]
[3, 4, 10, 9]
[3, 4, 9, 10]
Bubble sorted: [3, 4, 9, 10]
```

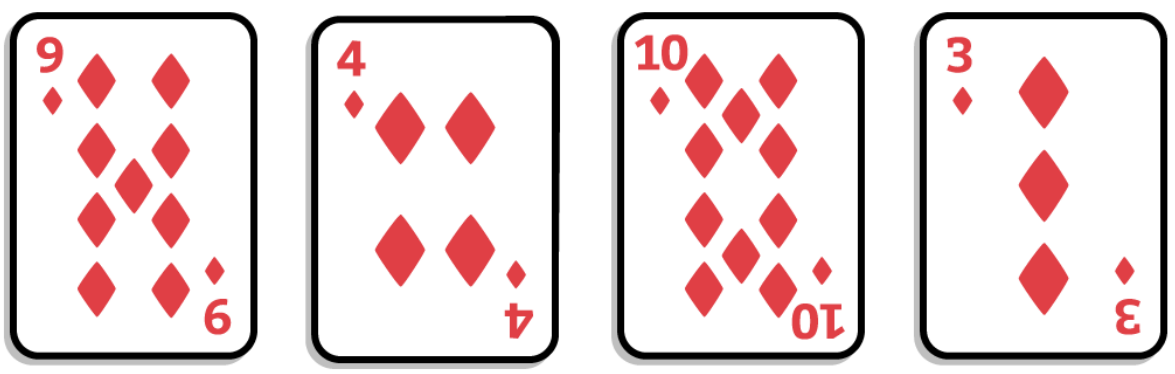
Like bubble sort, selection sort has a *worst* and *average* time complexity of $O(n^2)$, which is fairly dismal. Unlike the bubble sort, it also has the *best* time complexity of $O(n^2)$. Despite this, it performs better than bubble sort because it performs only $O(n)$ swaps — and the best thing about it is that it's a simple one to understand.

Insertion sort

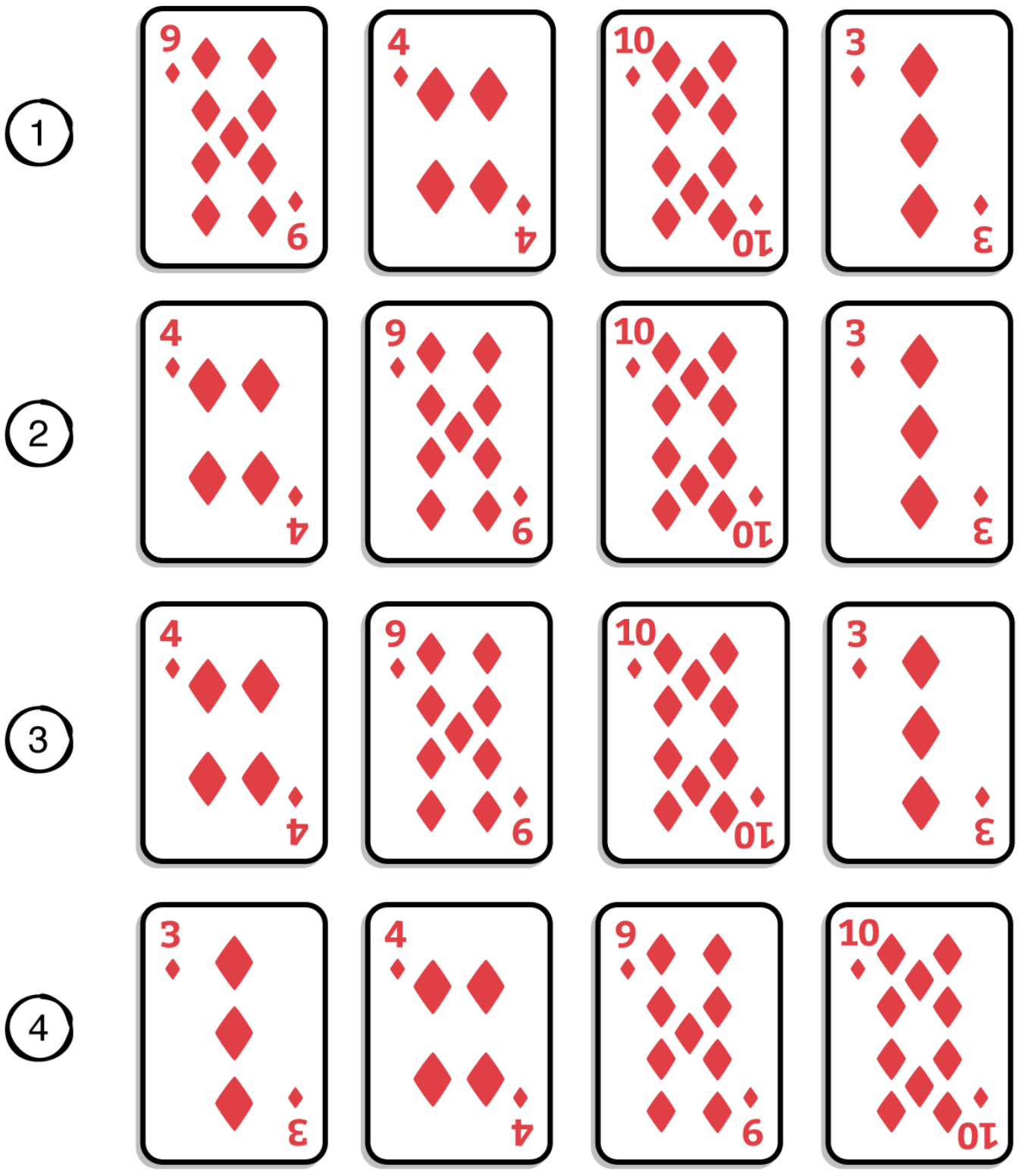
Insertion sort is a more useful algorithm. Like bubble sort and selection sort, insertion sort has an *average* time complexity of $O(n^2)$, but the performance of insertion sort can vary. The more the data is already sorted, the less work it needs to do. Insertion sort has a *best* time complexity of $O(n)$ if the data is already sorted.

Example

The idea of insertion sort is like how you'd sort a hand of cards. Consider the following hand:



Insertion sort will iterate *once* through the cards, from left to right. Each card is shifted to the left until it reaches its correct position.



1. You can ignore the first card, as there are no previous cards to compare it with.

2. Next, you compare 4 with 9 and shift 4 to the left by swapping positions with 9.

3. 10 doesn’t need to shift, as it’s in the correct position compared to the previous card.

4. Finally, 3 shifts to the front by comparing and swapping it with 10, 9 and 4, respectively.

The best case scenario for insertion sort occurs when the sequence of

values are already in sorted order, and no left shifting is necessary.

Implementation

In **src ▸ selectionsort** of your starter project, create a new file named **InsertionSort.kt**. Write the following inside of the file:

```
fun <T : Comparable<T>> ArrayList<T>.insertionSort(showPasses: Boolean = fa
  if (this.size < 2) return
  // 1
  for (current in 1 until this.size) {
    // 2
    for (shifting in (1..current).reversed()) {
      // 3
      if (this[shifting] < this[shifting - 1]) {
        this.swapAt(shifting, shifting - 1)
      } else {
        break
      }
      // 4
      if(showPasses) println(this)
  }
}
```

Here's what you did:

1. Insertion sort requires you to iterate from left to right, once. This loop does that.
2. Here, you run backward from the current index so you can shift left as needed.
3. Keep shifting the element left as long as necessary. As soon as the element is in position, `break` the inner loop and start with the next element.
4. This is the same trick you used with the other sort algorithms; it shows you the passes. Remember that this is not part of the sorting algorithm.

Head back to `main()` in **Main.kt** and write the following at the bottom:

```
"insertion sort" example {
  val list = arrayListOf(9, 4, 10, 3)
  println("Original: $list")
  list.insertionSort(true)
  println("Bubble sorted: $list")
}
```

You'll see the following console output:

```
---Example of insertion sort---
Original: [9, 4, 10, 3]
[4, 9, 10, 3]
[4, 9, 10, 3]
[3, 4, 9, 10]
Bubble sorted: [3, 4, 9, 10]
```

Insertion sort is one of the fastest sorting algorithms when some of the data is already sorted, but this isn't true for *all* sorting algorithms. In practice, a lot of data collections will already be mostly — if not entirely — sorted, and an insertion sort will perform quite well in those scenarios.

Generalization

In this section, you'll generalize these sorting algorithms for list types other than `ArrayList`. Exactly which list types won't matter, as long as they're mutable since you need to be able to swap elements. The changes are small and simple but important. You always want your algorithms to be as generic as possible.

You'll go through the code in the exact order you've written it, starting with **Utils.kt**.

Open the file and swap the `swapAt()` definition with this one:

```
fun <T> MutableList<T>.swapAt(first: Int, second: Int)
```

Head back to **BubbleSort.swift** and update the function definition to the following:

```
fun <T : Comparable<T>> MutableList<T>.bubbleSort(showPasses: Boolean = fal
```

Are you starting to see a pattern here?

Because you didn't use any `ArrayList` specific methods in the algorithm, you can change the `ArrayList` usages with `MutableList`. Do the same with the selection sort, in **SelectionSort.kt**:

```
fun <T : Comparable<T>> MutableList<T>.selectionSort(showPasses: Boolean =
```

Finally, deal with the insertion sort. Open **InsertionSort.kt** and replace the extension function definition with this one:

```
fun <T : Comparable<T>> MutableList<T>.insertionSort(showPasses: Boolean =
```

Notice that you don't need to change the examples in **Main.kt**. That's because the `ArrayList` is a `MutableList`. Since your algorithms are now more general, they can handle any implementation of the `MutableList`.

Generalization won't always be so easy, but it's something you need to do. You want your algorithm to work with as many data structures as possible. With a bit of practice, generalizing these algorithms becomes a fairly mechanical process.

In the next chapters, you'll take a look at sorting algorithms that perform better than $O(n^2)$. Up next is a sorting algorithm that uses a classical algorithm approach known as **divide and conquer —** merge sort!

Challenges

Challenge 1: To the left, to the left

Given a list of `Comparable` elements, bring all instances of a given value in the list to the right side of the list.

Solution 1

The trick to this problem is to find the elements that need to be moved and shift everything else to the left. Then, return to the position where the element was before, and continue searching from there.

```
fun <T : Comparable<T>> MutableList<T>.rightAlign(element: T) {
  // 1
  if (this.size < 2) return
  // 2
  var searchIndex = this.size - 2
  while (searchIndex >= 0) {
    // 3
    if (this[searchIndex] == element) {
      // 4
      var moveIndex = searchIndex
      while (moveIndex < this.size - 1 && this[moveIndex + 1] != element) {
        swapAt(moveIndex, moveIndex + 1)
        moveIndex++
      }
    }
    // 5
    searchIndex--
  }
}
```

Here's a breakdown of this moderately complicated function:

1. If there are less than two elements in the list, there's nothing to do.
2. You leave the last element alone and start from the previous one. Then, you go to the left (decreasing the index), until you reach the beginning of the list when the `searchIndex` is 0.

3. You're looking for elements that are equal to the one in the function parameter.
4. Whenever you find one, you start shifting it to the right until you reach another element equal to it or the end of the list. Remember, you already implemented `swapAt()`; don't forget to increment `moveIndex`.
5. After you're done with that element, move `searchIndex` to the left again by decrementing it.

The tricky part here is to understand what sort of capabilities you need. Since you need to make changes to the underlying storage, this function is only available to `MutableList` types.

To complete this algorithm efficiently, you need backward index traversal, which is why you can't use any generic `MutableCollection`.

Finally, you also need the elements to be `Comparable` to target the appropriate values.

The time complexity of this solution is $O(n)$.

Challenge 2: Duplicate finder

Given a list of `Comparable` elements, return the largest element that's a duplicate in the list.

Solution 2

Finding the biggest duplicated element is rather straightforward. To make it even easier, you can sort the list with one of the methods you've already implemented.

```
fun <T : Comparable<T>> MutableList<T>.biggestDuplicate(): T? {
  // 1
  this.selectionSort()
  // 2
```

```
  for (i in this.lastIndex downTo 1) {
    // 3
    if (this[i] == this[i - 1]) {
      return this[i]
    }
  }
  return null
}
```

Here's the solution in three steps:

1. You first sort the list.
2. Start going through it from right to left since you know that the biggest elements are on the right, neatly sorted.
3. The first one that's repeated is your solution.

In the end, if you've gone through all of the elements and none of them are repeated, you can return `null` and call it a day.

The time complexity of this solution is $O(n^2)$ because you've used sorting.

Challenge 3: Manual reverse

Reverse a list of elements by hand. Do not rely on `reverse` or `reversed`; you need to create your own.

Solution 3

Reversing a collection is also quite straightforward. Using the double reference approach, you start swapping elements from the start and end of the collection, making your way to the middle. Once you've hit the middle, you're done swapping, and the collection is reversed.

```
fun <T : Comparable<T>> MutableList<T>.rev() {
  var left = 0
  var right = this.lastIndex
```

```
  while (left < right) {
    swapAt(left, right)
    left++
    right--
  }
}
```

For this solution, you need `MutableList` since you need to mutate the collection to reverse.

The time complexity of this solution is $O(n)$.

Key points

- n^2 algorithms often have a terrible reputation, but these algorithms usually have some redeeming points. `insertionSort` can sort in $O(n)$ time if the collection is already in sorted order and gradually scales down to $O(n^2)$.
- `insertionSort` is one of the best sorts in situations wherein you know ahead of time that your data is in sorted order.
- Design your algorithms to be as generic as possible without hurting the performance.

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.