Sorting - Revisited

Sorting in Scala

- Sorting a list with sorted
- Only works if the type has a defined ordering
- Must inherit Ordered[Type]

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sorted
println(numbersSorted)
```

List(-23, -8, -4, 5, 7, 10)

Sorting in Scala

- Sorting a list by the result of a function/method
- Calls the provided function/method on each element and sorts by the returned values

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by the result of a method (like setting the key in Python sorting)
val numbersSorted = numbers.sortBy(Math.abs)
println(numbersSorted)
```

List(-4, 5, 7, -8, 10, -23)

Sorting in Scala

- Sorting a list using a comparator function/method
- The comparator takes two values of the type being sorted
    - Return true if the first parameter should come before the second in the sorted order
    - Return false otherwise (including ties)

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sortWith((a: Int, b: Int) => a > b)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

List(10, 7, 5, -4, -8, -23)

Sorting in Scala

- Sorting a list using a comparator function/method
- Can sort custom types with custom functions
- There's no stopping the ways you can sort!

```
def compareAnimals(a1: Animal, a2: Animal): Boolean = {
  a1.name.toLowerCase() < a2.name.toLowerCase()
}
```

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = animals.sortWith(compareAnimals)
println(animalsSorted)
```

List(Finn, Garfield, morris, Snoopy)

Sorting in Scala

- Sorting a list using a comparator function/method
- Can sort custom types with custom functions
- There's no stopping the ways you can sort!

```
def compareAnimals(a1: Animal, a2: Animal): Boolean = {
  a1.name.toLowerCase() < a2.name.toLowerCase()
}
```

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = animals.sortWith(compareAnimals)
println(animalsSorted)
```

List(Finn, Garfield, morris, Snoopy)

But how does it all work?

Selection Sort

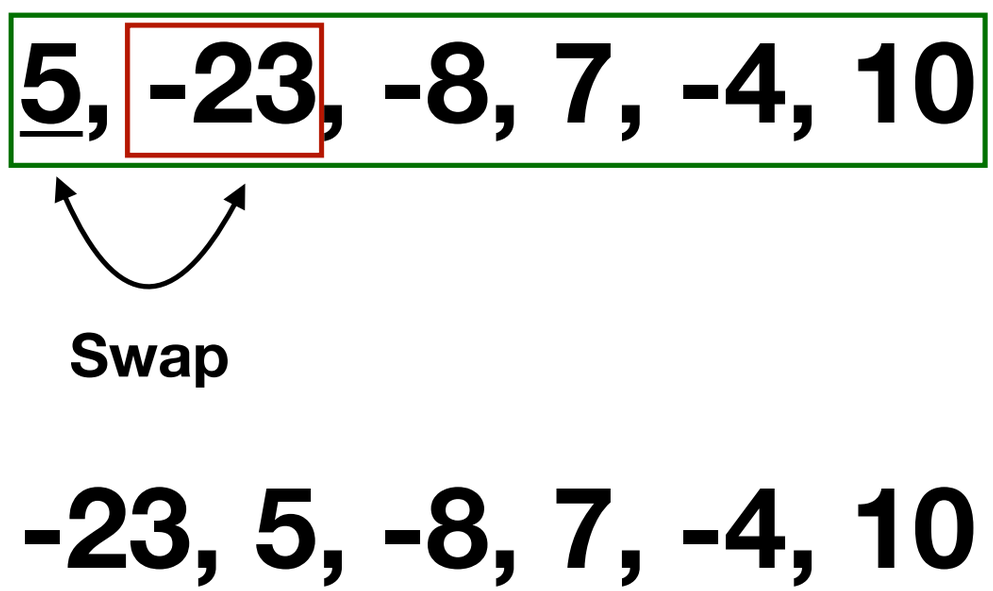
- Iterate over the indices of a list
    - For each index, select the element that belongs there in the final sorted order
        - Swap the current value with the correct one

Given: **5, -23, -8, 7, -4, 10**

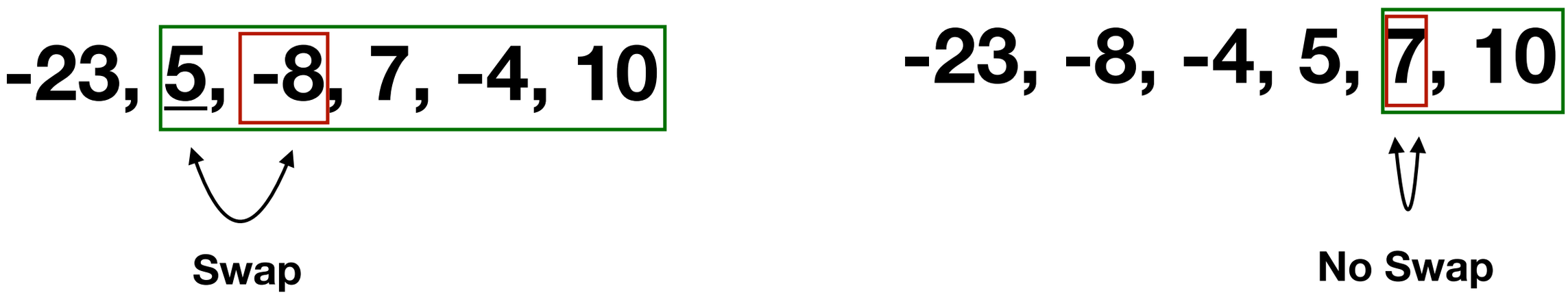
Correct Order: **-23, -8, -4, 5, 7, 10**

Selection Sort

- Start with the first index
- Find the element that belongs there by taking the min of all values
- Swap the values
- Don't have to recheck element that are already at the correct index



Selection Sort



Selection Sort

- How do we compare values?

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

- Take a comparator as a parameter just like sortWith

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

- Call the comparator whenever we need to compare 2 values

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = intSelectionSort(numbers, (a: Int, b: Int) => a > b)
```

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Type Parameters

- But what if we want to sort custom types?

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = selectionSort(animals, Animal.compareAnimals)
println(animalsSorted)
```

- Our selection sort only works with Int
- We can write another method to sort Animals
  - And another for every type we want to sort.. no
- We'll take the type as a parameter of our method

Type Parameters

- Type parameters come before the parameter list
- Use [ ] instead of ( )
- Can use this generic type throughout this method

```
def selectionSort[Type](inputData: List[Type], comparator: (Type, Type) => Boolean): List[Type] = {
  var data: List[Type] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Type Parameters

- We can choose the type name
- Generic type names are often shortened to 1 character

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Type Parameters

- The type parameter can be inferred as long as the data and comparator types match

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = selectionSort(animals, Animal.compareAnimals)
println(animalsSorted)
```

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort

- This all works..
- **But it's really slow!**
- The algorithm is inefficient
- We're creating many, many new lists that are not needed
- More efficiency coming soon