Merge Sort / Recursion

Merge Sort

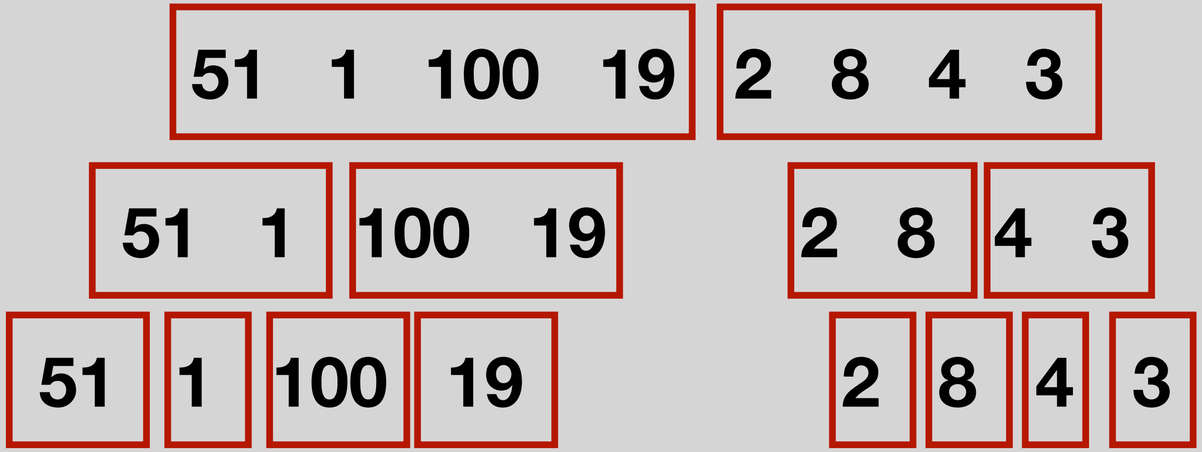
- The algorithm
  - If the input list has 1 element
    - Return it (It's already sorted)
  - Else
    - Divide the input list in two halves
    - Recursively call merge sort on each half (Repeats until the lists are size 1)
    - Merge the two sorted lists together into a single sorted list

Merge Sort

- Given an input

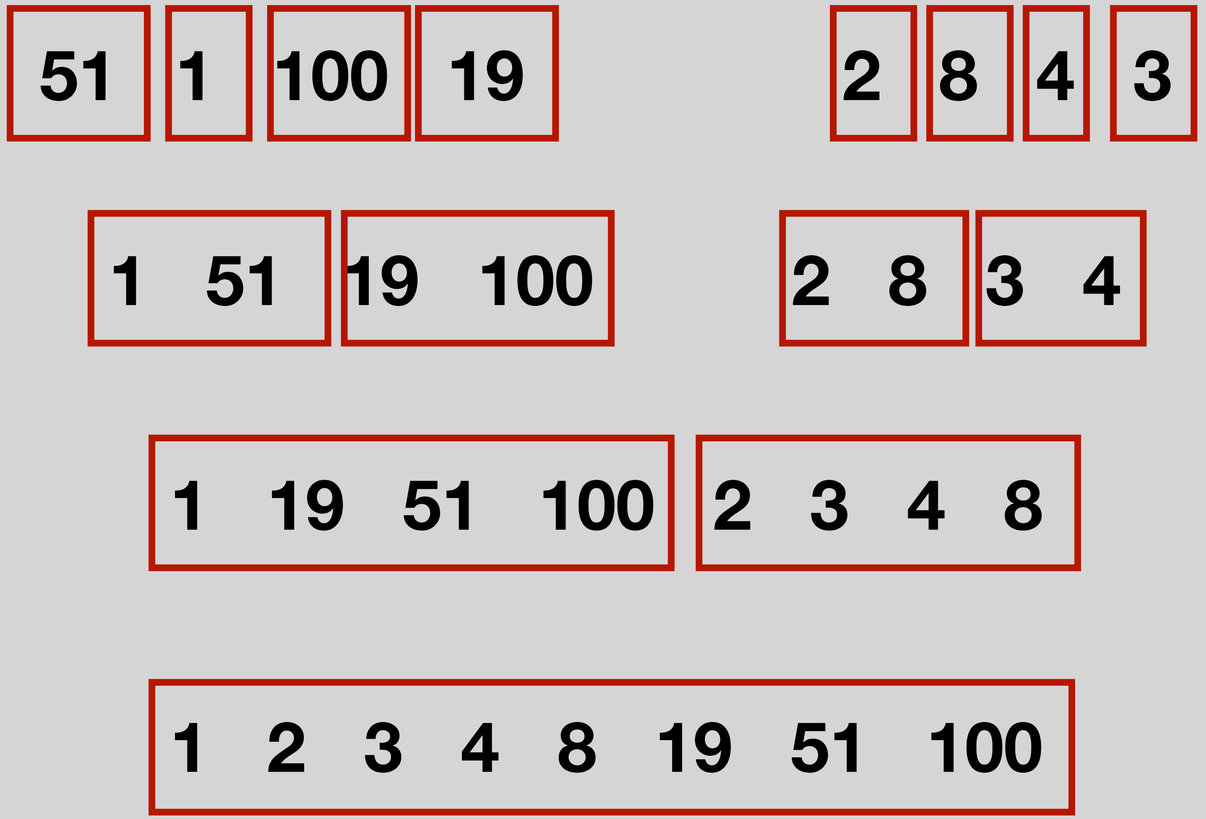
51 1 100 19 2 8 4 3

- Divide into two lists recursively until each list has size 1



Merge Sort

- Merge lists until the original input is sorted



Merge Sort - Merge

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Recursion!

Merge Sort - Merge

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Assume the recursive calls are correct

merge two sorts lists

Merge Sort - Merge

- Merge two sorted lists
- Take advantage of each list being sorted
- Start with pointers at the beginning of each list
- Compare the two values at the pointers and find which come first based on the comparator
  - Append it to a new list and advance that pointer
- When a pointer reaches the end of a list copy the rest of the contents

Merge Sort - Merge

Merge Sort - Merge

Merge Sort - Merge

Merge Sort - Merge

Merge Sort - Merge

Merge Sort - Merge

When a pointer reaches the end of a list,
copy the rest of the other list to the result

Merge Sort - Merge

When a pointer reaches the end of a list,
copy the rest of the other list to the result

Merge Sort - Merge

```
def merge(left: List[Double], right: List[Double]): List[Double] = {
  var leftPointer = 0
  var rightPointer = 0

  var sortedList: List[Double] = List()

  while (leftPointer < left.length && rightPointer < right.length) {
    if (left(leftPointer) < right(rightPointer)) {
      sortedList = sortedList :+ left(leftPointer)
      leftPointer += 1
    } else {
      sortedList = sortedList :+ right(rightPointer)
      rightPointer += 1
    }
  }

  while (leftPointer < left.length) {
    sortedList = sortedList :+ left(leftPointer)
    leftPointer += 1
  }
  while (rightPointer < right.length) {
    sortedList = sortedList :+ right(rightPointer)
    rightPointer += 1
  }

  sortedList
}
```

Writing Recursive Methods

- Suggested approach:
  - Write a base case(s) for an input that has a trivial return value
  - Only write recursive calls that get closer to a base case
  - Assume your recursive calls return the correct values
  - Write your method based on this assumption

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Writing Recursive Methods

- Suggested approach:
  - **Write a base case(s) for an input that has a trivial return value**
  - Only write recursive calls that get closer to a base case
  - Assume your recursive calls return the correct values
  - Write your method based on this assumption

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Writing Recursive Methods

- Suggested approach:
  - Write a base case(s) for an input that has a trivial return value
  - **Only write recursive calls that get closer to a base case**
  - Assume your recursive calls return the correct values
  - Write your method based on this assumption

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Writing Recursive Methods

- Suggested approach:
  - Write a base case(s) for an input that has a trivial return value
  - Only write recursive calls that get closer to a base case
  - **Assume your recursive calls return the correct values**
  - **Write your method based on this assumption**

```
def mergeSort(inputData: List[Double]): List[Double] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left)
    val rightSorted = mergeSort(right)

    merge(leftSorted, rightSorted)
  }
}
```

Debugger Demo