

# 16 Radix Sort Written by Márton Braun

So far, you've been relying on comparisons to determine the sorting order. In this chapter, you'll look at a completely different model of sorting known as radix sort.

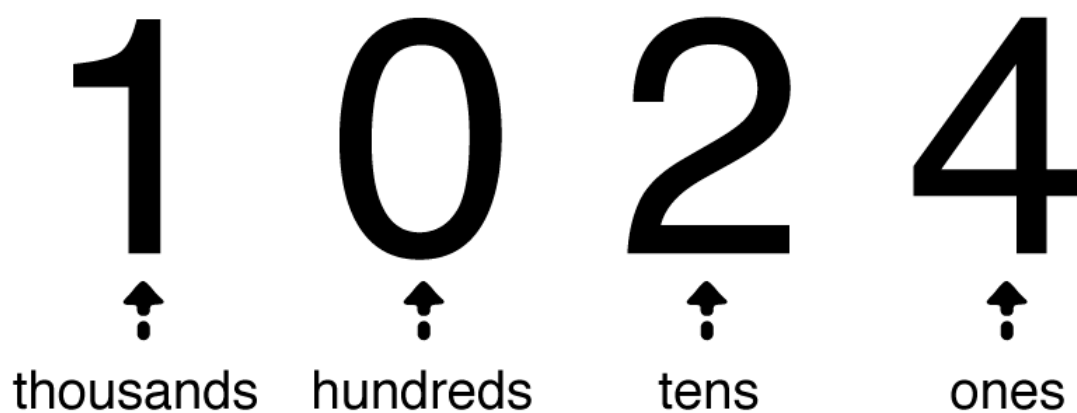
**Radix sort** is a non-comparative algorithm for sorting integers in linear time. There are many implementations of radix sort that focus on different problems. To keep things simple, you'll focus on sorting base 10 integers while investigating the *least significant digit* (LSD) variant of radix sort.

## Example

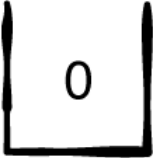
To show how radix sort works, you'll sort the following list:

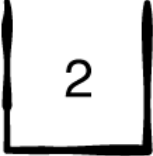
```
var list = arrayListOf(88, 410, 1772, 20)
```

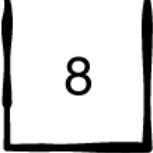
Radix sort relies on the positional notation of integers, as shown here:



First, the list is divided into buckets based on the value of the least significant digit, the **ones** digit.

 - 410, 20

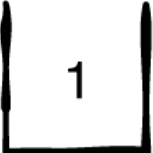
 - 1772

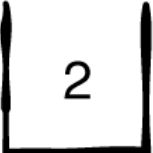
 - 88

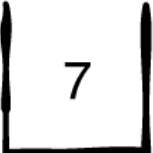
These buckets are then emptied in order, resulting in the following partially sorted list:

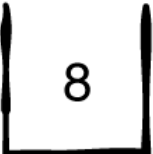
```
list = arrayListOf(410, 20, 1772, 88)
```

Next, repeat this procedure for the **tens** digit:

 - 410

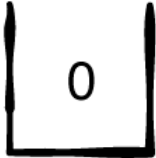
 - 20


 - 1772


 - 88

The relative order of the elements didn't change this time, but you've still got more digits to inspect.

The next digit to consider is the **hundreds** digit:

 - 20, 88

 - 410

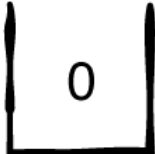
 - 1772


**Note:** For values that have no hundreds position or any other position, the digit is assumed to be zero.

Reassembling the list based on these buckets gives the following:

```
list = arrayListOf(20, 88, 410, 1772)
```

Finally, you need to consider the **thousands** digit:

 - 20, 88, 410

 - 1772

Reassembling the list from these buckets leads to the final sorted list:

```
list = arrayListOf(20, 88, 410, 1772)
```

When many numbers end up in the same bucket, their relative ordering doesn't change. For example, in the zero bucket for the hundreds position, 20 comes before 88. This is because the previous step put 20 in

a lower bucket than 80, so 20 ended up before 88 in the list.

## Implementation

Open the starter project for this chapter. In **src ▶ radixsort**, create a new file named **RadixSort.kt**.

Add the following to the file:

```
fun MutableList<Int>.radixSort() {  
    // ...  
}
```

Here, you've added `radixSort()` to `MutableList` of integers via an extension function. Start implementing `radixSort()` using the following:

```
fun MutableList<Int>.radixSort() {  
    // 1  
    val base = 10  
    // 2  
    var done = false  
    var digits = 1  
    while (!done) {  
        // ...  
    }  
}
```

This bit is straightforward:

1. You're sorting base 10 integers in this instance. Since you'll use this value many times in the algorithm, you store it in a constant `base`.
2. You declare two variables to track your progress. Radix sort works in many passes, so `done` serves as a flag that determines whether the sort is complete. The `digits` variable keeps track of the current digit you're looking at.

Next, you'll write the logic that sorts each element into buckets.

## Bucket sort

Write the following **inside** the `while` loop:

```
// 1
val buckets = MutableList<MutableList<Int>>(base) { mutableListOf() }
// 2
this.forEach { number ->
    val remainingPart = number / digits
    val digit = remainingPart % base
    buckets[digit].add(number)
}
// 3
digits *= base

this.clear()
this.addAll(buckets.flatten())
```

Here's how it works:

1. You instantiate the buckets using a two-dimensional list. You create as many buckets as the base you're using, which is ten in this case.
2. You place each number in the correct bucket.
3. You update `digits` to the next digit you want to inspect and update the list using the contents of `buckets.flatten()` flattens the two-dimensional list to a one-dimensional list, as if you're emptying the buckets into the list.

## When do you stop?

Your `while` loop currently runs forever, so you'll need a terminating condition somewhere. You'll do that as follows:

1. At the beginning of the `while` loop, add `done = true`.

2. At the end of the lambda passed to `forEach`, add the following:

```
if (remainingPart > 0) {  
    done = false  
}
```

Since `forEach` iterates over all of the integers, as long as one of the integers still has unsorted digits, you'll need to continue sorting.

With that, you've learned about your first non-comparative sorting algorithm. Head back to **Main.kt** and add the following to test your code, inside `main()` :

```
"radix sort" example {  
    val list = arrayListOf(88, 410, 1772, 20)  
    println("Original: $list")  
    list.radixSort()  
    println("Radix sorted: $list")  
}
```

You'll see the following console output:

```
---Example of radix sort---  
Original: [88, 410, 1772, 20]  
Radix sorted: [20, 88, 410, 1772]
```

Radix sort is one of the fastest sorting algorithms. The average time complexity of radix sort is  $O(k \times n)$ , where  $k$  is the number of significant digits of the largest number, and  $n$  is the number of integers in the list.

Radix sort works best when  $k$  is constant, which occurs when all numbers in the list have the same count of significant digits. Its time complexity then becomes  $O(n)$ . Radix sort also incurs an  $O(n)$  space complexity, as you need space to store each bucket.

# Challenges

## Challenge 1: Most significant sort

The implementation discussed in the chapter used a *least significant digit* radix sort. Your task is to implement a *most significant digit (MSD)* radix sort.

This sorting behavior is called **lexicographical sorting** and is also used for `String` sorting.

For example:

```
var list = arrayListOf(500, 1345, 13, 459, 44, 999)
list.lexicographicalSort()
println(list) // outputs [13, 1345, 44, 459, 500, 999]
```

### Solution 1

MSD radix sort is closely related to LSD radix sort, in that both use bucket sort. The difference is that MSD radix sort needs to curate subsequent passes of the bucket sort carefully. In LSD radix sort, bucket sort ran repeatedly using the whole list for every pass. In MSD radix sort, you run bucket sort with the entire list only once. Subsequent passes will sort each bucket recursively.

You'll implement MSD radix sort piece-by-piece, starting with the components on which it depends.

### Digits

Add the following inside **Challenge1.kt**:

```
fun Int.digits(): Int {
    var count = 0
    var num = this
    while (num != 0) {
```

```

        count += 1
        num /= 10
    }
    return count
}

fun Int.digit(atPosition: Int): Int? {
    val correctedPosition = (atPosition + 1).toDouble()
    if (correctedPosition > digits()) return null

    var num = this
    while (num / (pow(10.0, correctedPosition).toInt()) != 0) {
        num /= 10
    }
    return num % 10
}

```

`digits()` is a function that returns the number of digits that the `Int` has. For example, the value of 1024 has four digits.

`digit(atPosition)` returns the digit at a given position. Like lists, the leftmost position is zero. Thus, the digit for position zero of the value 1024 is **1**. The digit for position three is **4**. Since there are only four digits, the digit for position five will return `null`.

The implementation of `digit()` works by repeatedly chopping a digit off the end of the number, until the requested digit is at the end. It's then extracted using the remainder operator.

## Lexicographical sort

With the helper methods, you're now equipped to deal with MSD radix sort. Write the following at the bottom of the file:

```

fun MutableList<Int>.lexicographicalSort() {
    this.clear()
    this.addAll(msdRadixSorted(this, 0))
}

```



```
private fun msdRadixSorted(list: MutableList<Int>, position: Int): MutableList<Int> {
```

`lexicographicalSort()` is the user-facing API for MSD radix sort.  
`msdRadixSorted()` is the meat of the algorithm and will be used to recursively apply MSD radix sort to the list.

Update `msdRadixSorted()` to the following:

```
private fun msdRadixSorted(list: MutableList<Int>, position: Int): MutableList<Int> {
    // 1
    val buckets = MutableList<MutableList<Int>>(10) { mutableListOf() }
    // 2
    val priorityBucket = arrayListOf<Int>()
    // 3
    list.forEach { number ->
        val digit = number.digit(position)
        if (digit == null) {
            priorityBucket.add(number)
            return@forEach
        }
        buckets[digit].add(number)
    }
}
```

Here's how it works:

1. Similar to LSD radix sort, you instantiate a two-dimensional list for the buckets.
2. The `priorityBucket` is a special bucket that stores values with fewer digits than the current position. Values that go in the `priorityBucket` are sorted first.
3. For every number in the list, you find the digit of the current position and place the number in the appropriate bucket.

Next, you need to recursively apply MSD radix sort for each of the

individual buckets. Write the following at the end of `msdRadixSorted()`:

```
val newValues = buckets.reduce { result, bucket ->
    if (bucket.isEmpty()) return@reduce result
    result.addAll(msdRadixSorted(bucket, position + 1))
    result
}
priorityBucket.addAll(newValues)

return priorityBucket
```

This statement calls `reduce()` to collect the results of the recursive sorts and appends them to the `priorityBucket`. That way, the elements in the `priorityBucket` always go first. You're almost done!

## Base case

As with all recursive operations, you need to set a terminating condition that stops the recursion. Recursion should halt if the current position you're inspecting is greater than the number of significant digits of the largest value inside the list.

At the bottom of the file, write the following:

```
private fun List<Int>.maxDigits(): Int {
    return this.maxOrNull()?.digits() ?: 0
}
```

Next, add the following at the top of `msdRadixSorted`:

```
if (position >= list.maxDigits()) return list
```

This ensures that if the position is equal or greater than the list's `maxDigits`, you'll terminate the recursion.

Take it out for a spin! Add the following to **Main.kt** so you can test the code:

```
"MSD radix sort" example {  
    val list = (0..10).map { (Math.random() * 10000).toInt() }.toMutableList()  
    println("Original: $list")  
    list.lexicographicalSort()  
    println("Radix sorted: $list")  
}
```

You should see a list of random numbers like this:

---Example of MSD radix sort---

```
Original: [448, 3168, 6217, 7117, 1256, 3887, 3900, 3444, 4976, 6891, 4682]  
Radix sorted: [1256, 3168, 3444, 3887, 3900, 448, 4682, 4976, 6217, 6891, 7
```

Since the numbers are random, you won't get an identical list. The important thing to note is the lexicographical ordering of the values.

## Key points

- Radix sort is a non-comparative sort that doesn't rely on comparing two values. Instead, it leverages bucket sort, which is like a sieve for filtering values. A helpful analogy is how some of the vending machines accept coins — the coins are distinguished by size.
- This chapter covered the least significant digit radix sort. Another way to implement radix sort is the most significant digit form. This form sorts by prioritizing the most significant digits over the lesser ones and is best illustrated by the sorting behavior of the `String` type.

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](#).