<https://www.happycoders.eu/algorithms/radix-sort/>

In this article, you will learn about the "Radix Sort" sorting algorithm. You will learn:

* How does Radix Sort work? (Step by step)
* How to implement Radix Sort in Java?
* What is the time and space complexity of Radix Sort?
* What variants of Radix Sort exist?
* … and what does the term "radix" mean anyway?

Let's start with the last question:

## What is Radix Sort?

"Radix" is the Latin word for "root" – nevertheless, Radix Sort has nothing to do with calculating square roots.

Instead, the "radix" of a number system (also called the "base") refers to the number of digits needed to represent numbers in that number system. The radix in the decimal system is 10, the radix of the binary system is 2, and the radix of the hexadecimal system is 16.

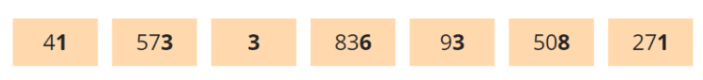
In Radix Sort, we sort the numbers digit by digit – and not, as in most other sorting methods, by comparing two numbers. You can read more about how this works in the following chapter.

## Radix Sort Algorithm

The algorithm for Radix Sort is best explained step by step using an example. We want to sort the following numbers:



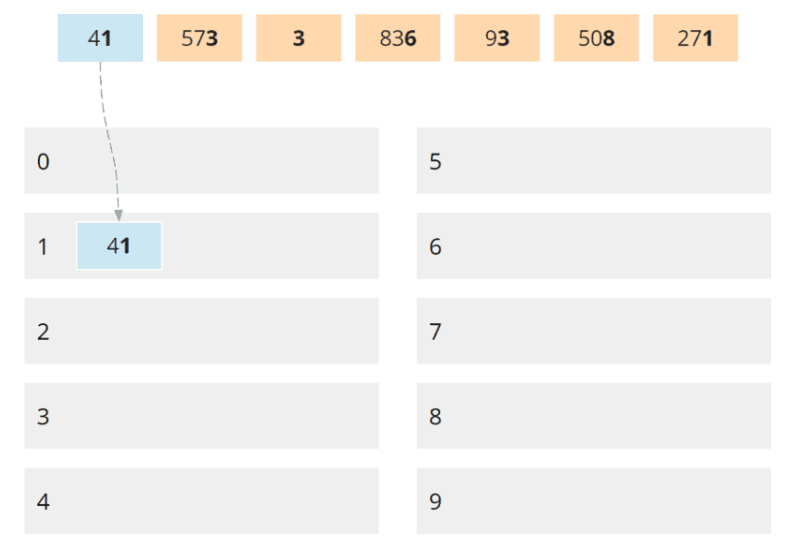
We will start by looking at the last digit only (there are also Radix Sort variations where you start at the first digit, but we'll get to that later):



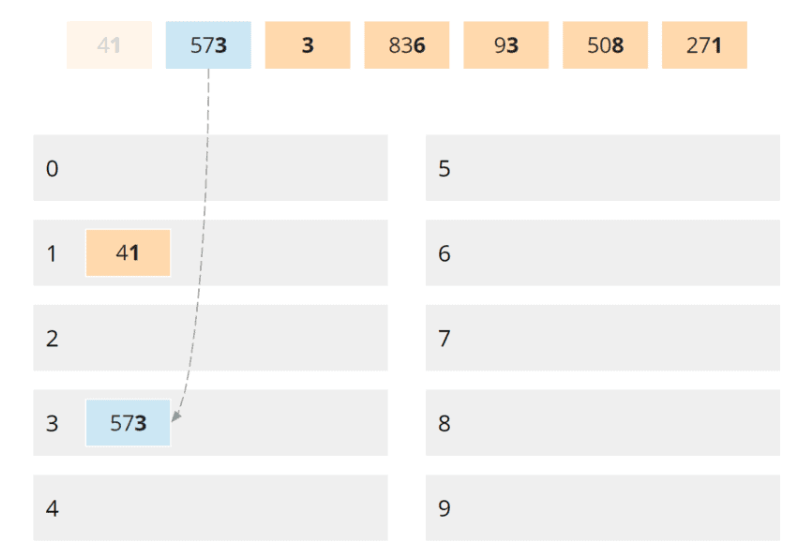
We sort the numbers in two phases: a partitioning phase and a collection phase.

### Partitioning Phase

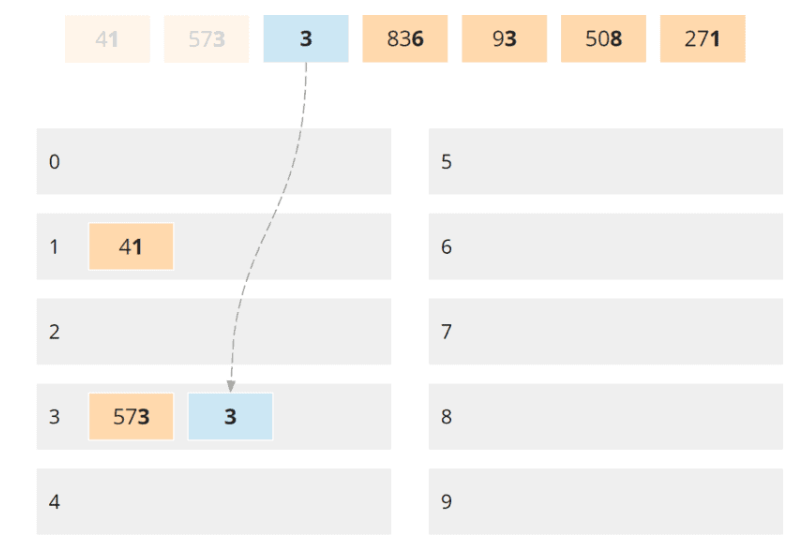
For the partitioning, we create ten so-called "buckets", designated with "0" to "9". We distribute the numbers to these buckets according to their last digit. The following image demonstrates how we place the first number, 41, in bucket "1":



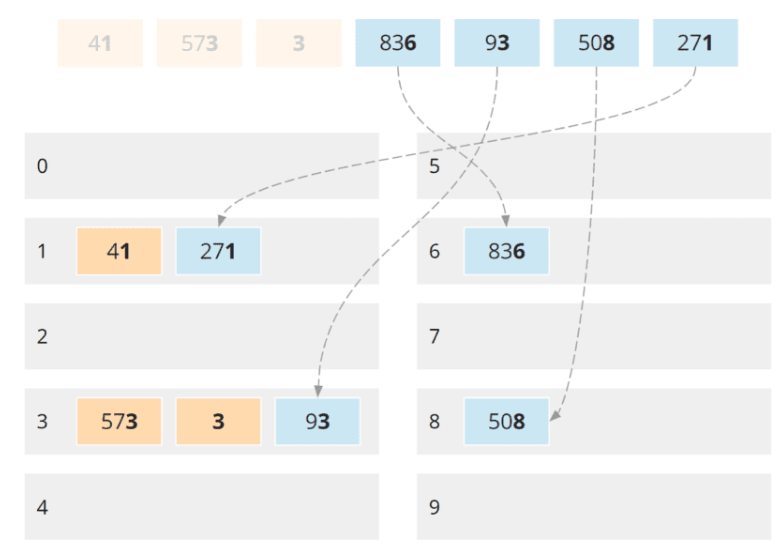
The second number, 573, is placed in bucket "3" according to its last digit:



The third number, 3, is also placed in bucket "3":



In the same way, we distribute the remaining numbers to the buckets:

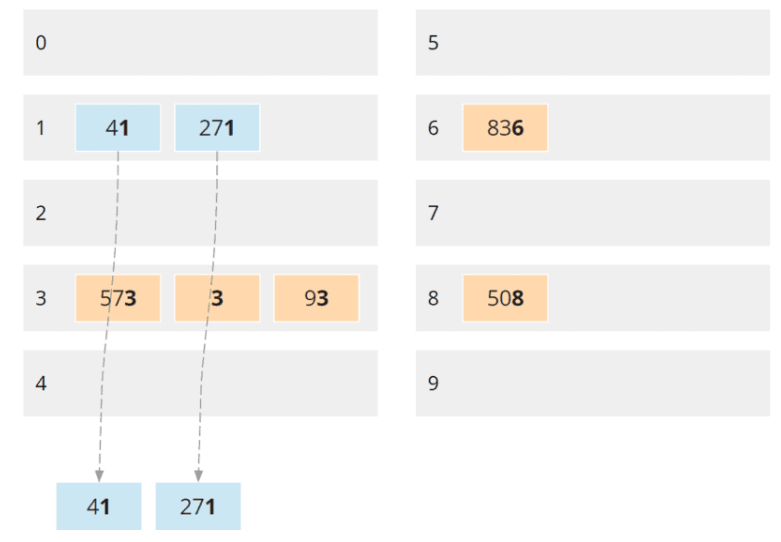


That completes the partitioning phase for the last digit.

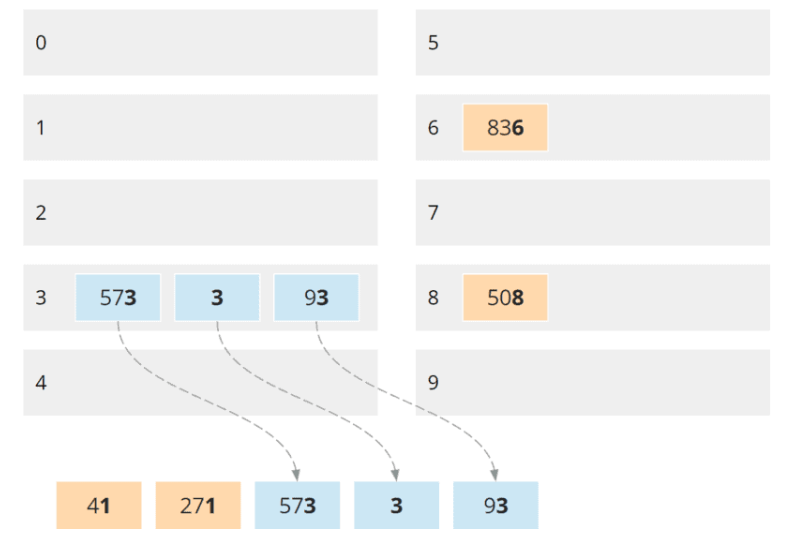
### Collection Phase

The partitioning phase is followed by the collecting phase. We collect the numbers, bucket by bucket, in ascending order – and within the buckets from left to right (i.e., in the same order as the numbers were entered in the respective bucket) – into a new list.

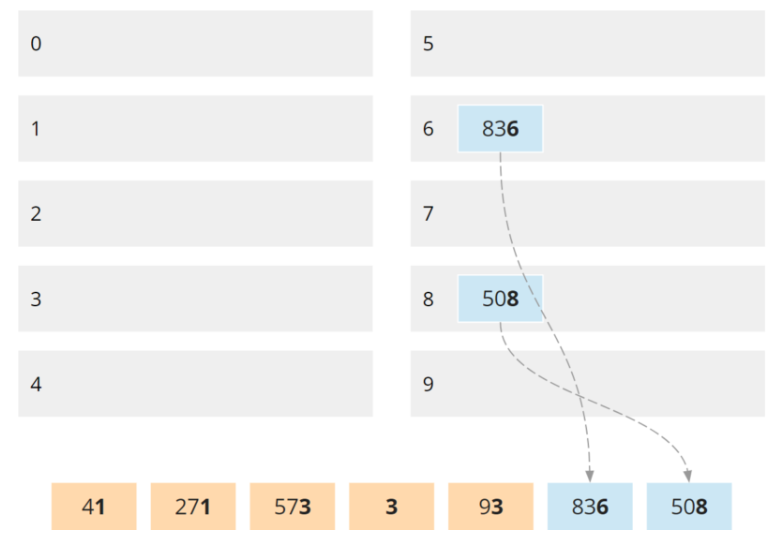
We start with the bucket with the smallest digit, i.e., bucket 1:



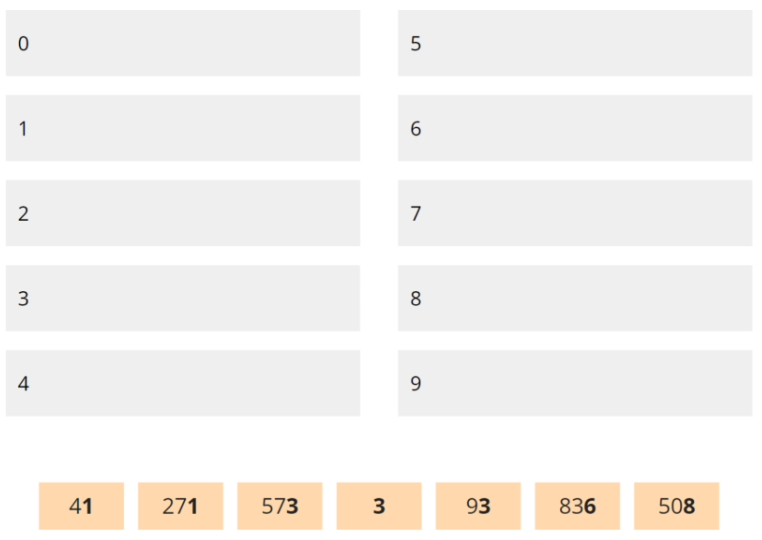
After that, we collect the numbers of the next higher bucket, that's bucket 3:



And finally, the numbers from bucket 6 and then bucket 8:



All buckets are now empty:

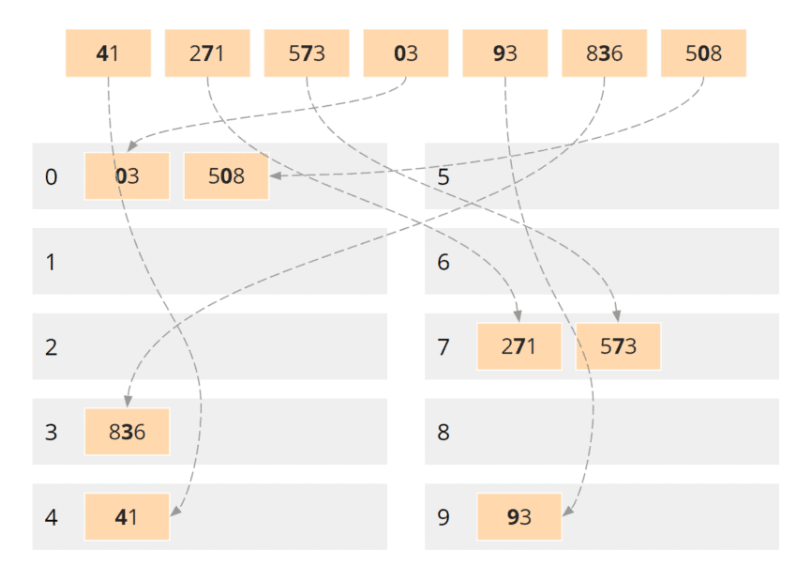


In this new list, the numbers are sorted in ascending order by their last digit: 1, 1, 3, 3, 6, 8

### Sorting by Tens Place

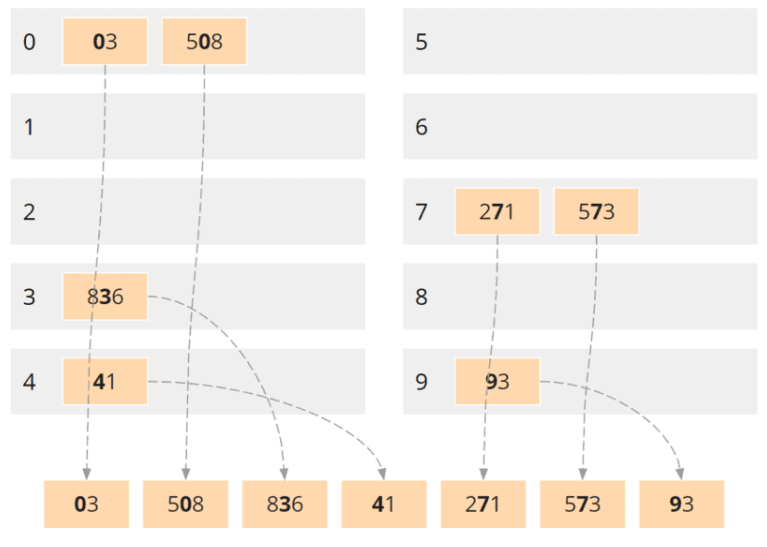
We repeat the partitioning and collecting phase for the tens place digits. This time, I represent the two phases with only one image each.

In the partitioning phase, we distribute the numbers to the buckets according to their tens place digit:



The tens place digit of one-digit numbers is zero. Accordingly, I have represented the three as "03".

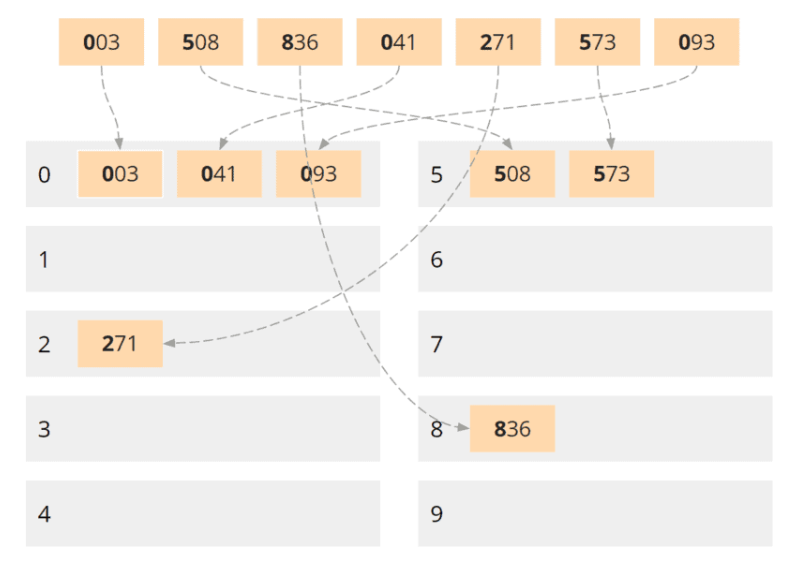
In the collection phase, we again collect the numbers, bucket by bucket:



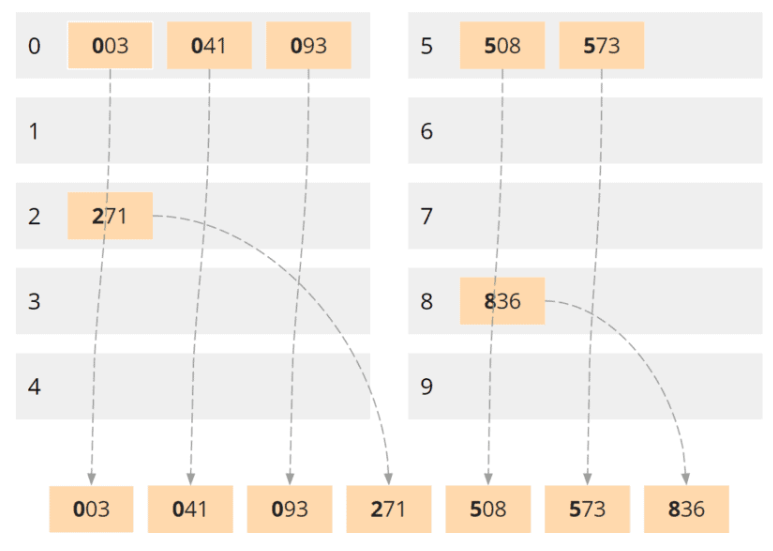
The numbers are now sorted according to their last two digits: 3, 8, 36, 41, 71, 73, 93

### Sorting by Hundreds Place

We repeat the same procedure for the hundreds place. First, the partitioning phase:



And after that, the collection phase:



After the third and final collection phase, the numbers are entirely sorted.

Here again, are the final result without leading zeros:



In the next chapter, we come to the implementation of Radix Sort.

## Radix Sort in Java

Radix Sort can be implemented in several ways. We'll start with a simple variant that is very close to the algorithm described. After that, I will show you two alternative implementations.

### Variant 1: Radix Sort With Dynamic Lists

We start with an empty sort() method and fill it step by step.

(You can find the final result at the end of this section and in the [RadixSortWithDynamicLists](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithDynamicLists.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repository of this sorting tutorial series).

public class RadixSortWithDynamicLists

  public void sort(int[] elements) {

    // We will implement this method step by step...

  }

}

Since we need to repeat the two phases (partitioning phase and collecting phase) for each digit, we first need to determine how many digits our numbers have.

We do this by finding the largest number from the array to be sorted and then counting how many times that number can be divided by 10:

public class RadixSortWithDynamicLists

  public void sort(int[] elements) {

    int max = getMaximum(elements);

    int numberOfDigits = getNumberOfDigits(max);

    // TODO: Implement the partitioning and collection phases

  }

  private static int getMaximum(int[] elements) {

    int max = 0;

    for (int element : elements) {

      if (element > max) {

        max = element;

      }

    }

    return max;

  }

  private int getNumberOfDigits(int number) {

    int numberOfDigits = 1;

    while (number >= 10) {

      number /= 10;

      numberOfDigits++;

    }

    return numberOfDigits;

  }

}

Then we sort digit by digit. We write a for loop with the loop variable digitIndex, where 0 stands for the units place, 1 for the tens place, 2 for the hundreds place, and so on.

(In the following listings, I don't print the class anymore, only the methods within the class).

public void sort(int[] elements) {

  int max = getMaximum(elements);

  int numberOfDigits = getNumberOfDigits(max);

  for (int digitIndex = 0; digitIndex < numberOfDigits; digitIndex++) {

    // TODO: Sort elements by digit at 'digitIndex'

  }

}

For the next step, we need the buckets to distribute the numbers to. We could use ten ArrayLists for this.

However, it is more elegant to wrap them in a Bucket class. That makes the code more readable and allows us to change the implementation of the buckets later without having to change the rest of the code.

We can create the Bucket class as an inner class inside RadixSortWithDynamicLists:

private static class Bucket {

  private final List<Integer> elements = new ArrayList<>();

  private void add(int element) {

    elements.add(element);

  }

  private List<Integer> getElements() {

    return elements;

  }

}

That was the preparation.

Let's move on to the partitioning phase. We need ten buckets on which to distribute the numbers; we generate them with a createBuckets() method:

private Bucket[] createBuckets() {

  Bucket[] buckets = new Bucket[10];

  for (int i = 0; i < 10; i++) {

    buckets[i] = new Bucket();

  }

  return buckets;

}

After that, we distribute our numbers among the buckets based on the digit at the digitIndex currently under consideration:

private void distributeToBuckets(int[] elements, int digitIndex, Bucket[] buckets) {

  int divisor = calculateDivisor(digitIndex);

  for (int element : elements) {

    int digit = element / divisor % 10;

    buckets[digit].add(element);

  }

}

private int calculateDivisor(int digitIndex) {

  int divisor = 1;

  for (int i = 0; i < digitIndex; i++) {

    divisor \*= 10;

  }

  return divisor;

}

The divisor is the number by which we must divide an element so that the rearmost digit is the digit currently under consideration – i.e., 1 for the units place, 10 for the tens place, 100 for the hundreds place, and so on.

We combine the methods of the partitioning phase in a partition() method:

private Bucket[] partition(int[] elements, int digitIndex) {

  Bucket[] buckets = createBuckets();

  distributeToBuckets(elements, digitIndex, buckets);

  return buckets;

}

In the collection phase, all we have to do is join the numbers from the individual buckets:

private void collect(Bucket[] buckets, int[] elements) {

  int targetIndex = 0;

  for (Bucket bucket : buckets) {

    for (int element : bucket.getElements()) {

      elements[targetIndex] = element;

      targetIndex++;

    }

  }

}

We combine the partition() and collect() methods into a sortByDigit() method:

private void sortByDigit(int[] elements, int digitIndex) {

  Bucket[] buckets = partition(elements, digitIndex);

  collect(buckets, elements);

}

And now, we close the circle by calling the sortByDigit() method from the digitIndex loop in the sort() method shown at the beginning:

public void sort(int[] elements) {

  int max = getMaximum(elements);

  int numberOfDigits = getNumberOfDigits(max);

  for (int digitIndex = 0; digitIndex < numberOfDigits; digitIndex++) {

    sortByDigit(elements, digitIndex);

  }

}

That completes our Radix Sort implementation.

Here you can see the complete source code again:

public class RadixSortWithDynamicLists {

  public void sort(int[] elements) {

    int max = getMaximum(elements);

    int numberOfDigits = getNumberOfDigits(max);

    for (int digitIndex = 0; digitIndex < numberOfDigits; digitIndex++) {

      sortByDigit(elements, digitIndex);

    }

  }

  private static int getMaximum(int[] elements) {

    int max = 0;

    for (int element : elements) {

      if (element > max) {

        max = element;

      }

    }

    return max;

  }

  private int getNumberOfDigits(int number) {

    int numberOfDigits = 1;

    while (number >= 10) {

      number /= 10;

      numberOfDigits++;

    }

    return numberOfDigits;

  }

  private void sortByDigit(int[] elements, int digitIndex) {

    Bucket[] buckets = partition(elements, digitIndex);

    collect(buckets, elements);

  }

  private Bucket[] partition(int[] elements, int digitIndex) {

    Bucket[] buckets = createBuckets();

    distributeToBuckets(elements, digitIndex, buckets);

    return buckets;

  }

  private Bucket[] createBuckets() {

    Bucket[] buckets = new Bucket[10];

    for (int i = 0; i < 10; i++) {

      buckets[i] = new Bucket();

    }

    return buckets;

  }

  private void distributeToBuckets(int[] elements, int digitIndex, Bucket[] buckets) {

    int divisor = calculateDivisor(digitIndex);

    for (int element : elements) {

      int digit = element / divisor % 10;

      buckets[digit].add(element);

    }

  }

  private int calculateDivisor(int digitIndex) {

    int divisor = 1;

    for (int i = 0; i < digitIndex; i++) {

      divisor \*= 10;

    }

    return divisor;

  }

  private void collect(Bucket[] buckets, int[] elements) {

    int targetIndex = 0;

    for (Bucket bucket : buckets) {

      for (int element : bucket.getElements()) {

        elements[targetIndex] = element;

        targetIndex++;

      }

    }

  }

  private static class Bucket {

    private final List<Integer> elements = new ArrayList<>();

    private void add(int element) {

      elements.add(element);

    }

    private List<Integer> getElements() {

      return elements;

    }

  }

}

By the way, the [RadixSortWithDynamicLists](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithDynamicLists.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repository is slightly different from the source code printed here:

* It implements the [SortAlgorithm](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/SortAlgorithm.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) interface, which allows comparison of different Radix Sort implementations with each other and with the other algorithms of the sorting algorithm series.
* The getMaximum() method is placed in the [ArrayUtils](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/utils/ArrayUtils.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class.
* The getNumberOfDigits() and calculateDivisor() methods are in the [RadixSortHelper](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortHelper.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class and can thus be used in other Radix Sort implementations.

The implementation shown has one shortcoming:

Dynamic lists (i.e., lists that can change size at runtime) are not optimal for performance-critical purposes such as sorting algorithms because adding elements involves some performance overhead (for example, in a linked list, new nodes must be created; in an ArrayList, the array must be recopied into a larger one at certain intervals).

In the next section, I will show you an alternative variant.

### Variant 2: Radix Sort with Arrays

We can speed up the implementation significantly (we will compare the performance of the implementations afterward) by using an array instead of an ArrayList for the buckets.

Since arrays have a fixed size, we need to know how many elements a bucket will contain before creating it. We modify our Bucket class as follows and pass the size to its constructor:

private static class Bucket {

  private final int[] elements;

  private int addIndex;

  private Bucket(int size) {

    elements = new int[size];

  }

  private void add(int element) {

    elements[addIndex] = element;

    addIndex++;

  }

  private int[] getElements() {

    return elements;

  }

}

To determine how many elements a bucket should contain, we first count the digits at the current digitIndex. The partition() method then looks like this:

private Bucket[] partition(int[] elements, int digitIndex) {

  int[] counts = countDigits(elements, digitIndex);

  Bucket[] buckets = createBuckets(counts);

  distributeToBuckets(elements, digitIndex, buckets);

  return buckets;

}

private int[] countDigits(int[] elements, int digitIndex) {

  int[] counts = new int[10];

  int divisor = calculateDivisor(digitIndex);

  for (int element : elements) {

    int digit = element / divisor % 10;

    counts[digit]++;

  }

  return counts;

}

private Bucket[] createBuckets(int[] counts) {

  Bucket[] buckets = new Bucket[10];

  for (int i = 0; i < 10; i++) {

    buckets[i] = new Bucket(counts[i]);

  }

  return buckets;

}

We don't need to change the distributeToBuckets() method or any other method shown in variant 1. Good thing we used a Bucket class in the first variant – and not an ArrayList :-)

You can find the complete code of variant 2 in the [RadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repository.

Let's move on to a third variant.

### Variant 3: Radix Sort with Counting Sort

In variant 2, we counted in advance how many elements would be sorted into each bucket. With this information, we can skip the buckets and move the elements directly to their target positions. We do this by applying the [general form of Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Algorithm_General_Form).

I won't repeat here how Counting Sort works. I'll show you the implementation right away:

public class RadixSortWithCountingSort {

  @Override

  public void sort(int[] elements) {

    int max = getMaximum(elements);

    int numberOfDigits = getNumberOfDigits(max);

    // Remember input array

    int[] inputArray = elements;

    for (int digitIndex = 0; digitIndex < numberOfDigits; digitIndex++) {

      elements = sortByDigit(elements, digitIndex);

    }

    // Copy sorted elements back to input array

    System.arraycopy(elements, 0, inputArray, 0, elements.length);

  }

  // Same as in the other variants:

  // getMaximum(), getNumberOfDigits(), calculateDivisor()

  private int[] sortByDigit(int[] elements, int digitIndex) {

    int[] counts = countDigits(elements, digitIndex);

    int[] prefixSums = calculatePrefixSums(counts);

    return collectElements(elements, digitIndex, prefixSums);

  }

  private int[] countDigits(int[] elements, int digitIndex) {

    int[] counts = new int[10];

    int divisor = calculateDivisor(digitIndex);

    for (int element : elements) {

      int digit = element / divisor % 10;

      counts[digit]++;

    }

    return counts;

  }

  private int[] calculatePrefixSums(int[] counts) {

    int[] prefixSums = new int[10];

    prefixSums[0] = counts[0];

    for (int i = 1; i < 10; i++) {

      prefixSums[i] = prefixSums[i - 1] + counts[i];

    }

    return prefixSums;

  }

  private int[] collectElements(int[] elements, int digitIndex, int[] prefixSums) {

    int divisor = calculateDivisor(digitIndex);

    int[] target = new int[elements.length];

    for (int i = elements.length - 1; i >= 0; i--) {

      int element = elements[i];

      int digit = element / divisor % 10;

      target[--prefixSums[digit]] = element;

    }

    return target;

  }

}

You can also find this code in the GitHub repository, in the [RadixSortWithCountingSort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithCountingSort.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class.

## Radix Sort Variants

There are two basic variants of Radix Sort, which differ in the order in which we look at the digits of the elements.

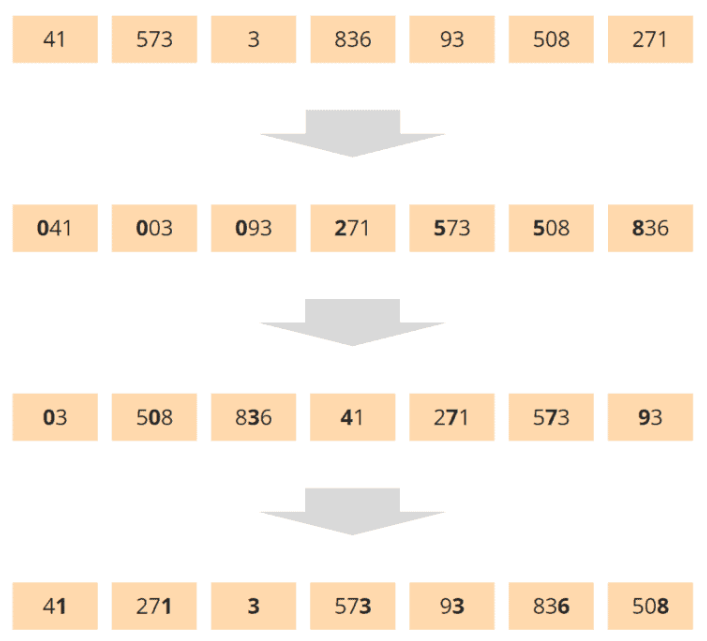
### LSD Radix Sort

The Radix Sort algorithm shown in the first chapter is called "LSD Radix Sort". LSD stands for "least significant digit". We started sorting at the least significant digit (the ones) and worked our way up, digit by digit, to the most significant digit.

### MSD Radix Sort

Alternatively, we can also start at the most significant digit. Accordingly, the second variant is called "MSD Radix Sort".

However, we have to proceed differently than with the LSD variant. Because if we were to sort the entire input list in our initial example first by hundreds place, then by tens place, and finally by units place, the following would happen (I have omitted the buckets in the graphic since we are only concerned with the results after the three collect phases):



Sorting by the tens place and ones place has mixed up the respective previous sortings.

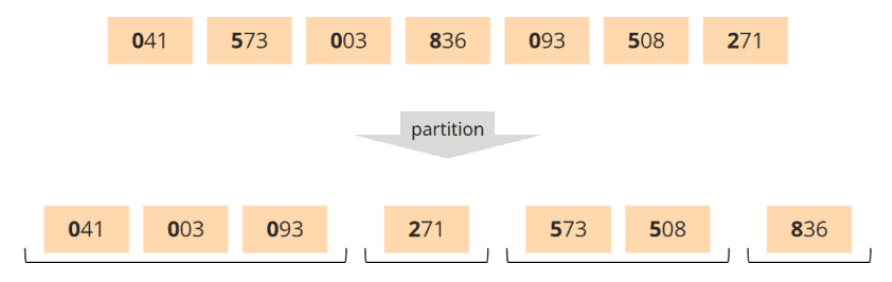
The problem is solved quickly:

After the hundreds place, we must not sort the input list again as a whole, but the hundreds place buckets within themselves. We then sort the resulting tens place buckets by the units place. In other words, we sort the buckets recursively.

### MSD Radix Sort – Step by Step

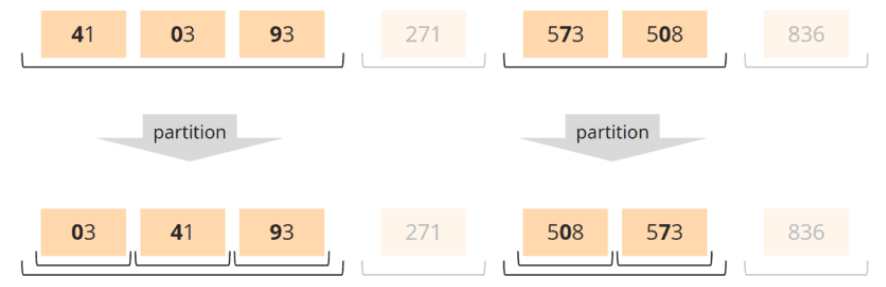
The following diagrams show the recursive MSD Radix Sort procedure step by step using the initial example. Buckets are represented by brackets under the elements. Empty buckets are not shown.

We start with partitioning by hundreds place:



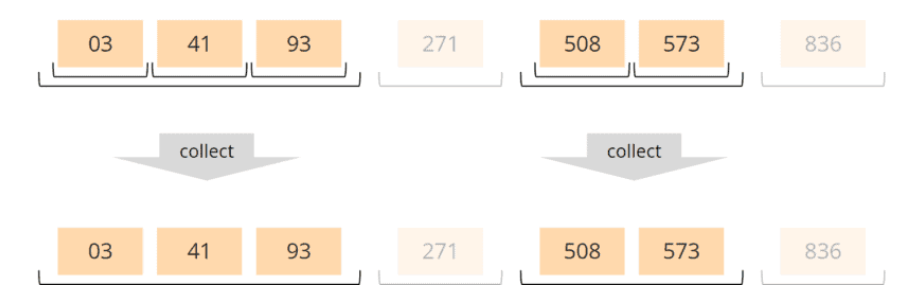
Now, instead of moving from the partitioning phase to the collecting phase, we perform another partitioning phase on each bucket – on the next lower digit, that is, the tens place.

Empty buckets and those containing only one element (such as the 271 and the 836) need not be partitioned further.

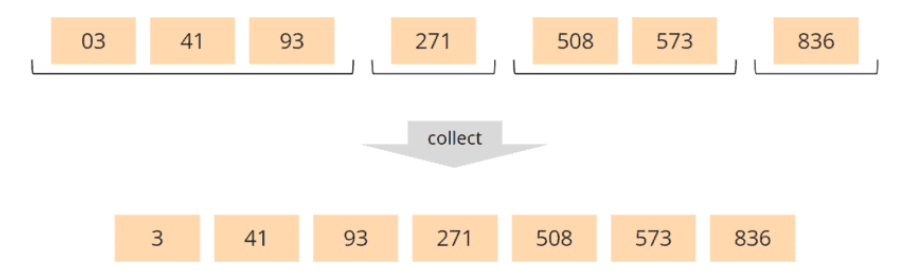


Actually, we would now have to partition the buckets by units place. But since none of the tens place buckets contains more than one element, this is unnecessary.

We, therefore, exit the recursion. First, we execute a collection phase on the tens place buckets:



And lastly, we perform the collection phase on the hundreds place buckets:



That completes the sorting.

### MSD Radix Sort – Implementation

Like the LSD variant, we can implement MSD Radix Sort with dynamic lists, arrays, and Counting Sort.

I'll show you how to modify the LSD array implementation shown above into an MSD implementation with just a few changes.

Here are once more the sort() and sortByDigit() methods of the [RadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class:

public void sort(int[] elements) {

  int max = getMaximum(elements);

  int numberOfDigits = getNumberOfDigits(max);

  for (int digitIndex = 0; digitIndex < numberOfDigits; digitIndex++) {

    sortByDigit(elements, digitIndex);

  }

}

private void sortByDigit(int[] elements, int digitIndex) {

  Bucket[] buckets = partition(elements, digitIndex);

  collect(buckets, elements);

}

All we have to do now is call the sortByDigit() method for the most significant digit first and insert the recursive call for the next lower digit between the partitioning and collecting phases:

public void sort(int[] elements) {

  int max = getMaximum(elements);

  int numberOfDigits = getNumberOfDigits(max);

  sortByDigit(elements, numberOfDigits - 1);

}

private void sortByDigit(int[] elements, int digitIndex) {

  Bucket[] buckets = partition(elements, digitIndex);

  // If we haven't reached the last digit,

  // sort the buckets by the next digit, recursively

  if (digitIndex > 0) {

    for (Bucket bucket : buckets) {

      if (bucket.needsToBeSorted()) {

        sortByDigit(bucket.getElements(), digitIndex - 1);

      }

    }

  }

  collect(buckets, elements);

}

The Bucket.needsToBeSorted() method returns true if the bucket contains at least one element.

You can find the complete code in the [RecursiveMsdRadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RecursiveMsdRadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repository.

As an exercise, I'll leave it to you to write an MSD variant for each of the other two LSD implementations (dynamic lists and Counting Sort).

### Using Other Bases

So far, we have partitioned according to the decimal system, i.e., with ten buckets. However, we can also work with any other base, for example, with the binary system (2 buckets), the hexadecimal system (16 buckets), or even with a hundred, a thousand, or more buckets.

The higher the base, the more buckets, and the more complex the partitioning phase. On the other hand, the numbers to sort have fewer digits (1,000,000 decimal = F4240 hexadecimal), so altogether fewer partitioning and collecting phases are required. We will determine what this means for performance in the "[Radix Sort Runtime](https://www.happycoders.eu/algorithms/radix-sort/" \l "radix-sort-runtime)" chapter.

How do you implement Radix Sort with a different base?

Basically, we need to replace each occurrence of the number 10 in the source code with the new base. In the [RadixSortWithDynamicLists](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithDynamicLists.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class, "10" occurs in the following methods:

private int getNumberOfDigits(int number) {

  int numberOfDigits = 1;

  while (number >= 10) {

    number /= 10;

    numberOfDigits++;

  }

  return numberOfDigits;

}

private Bucket[] createBuckets() {

  Bucket[] buckets = new Bucket[10];

  for (int i = 0; i < 10; i++) {

    buckets[i] = new Bucket();

  }

  return buckets;

}

private void distributeToBuckets(int[] elements, int digitIndex, Bucket[] buckets) {

  int divisor = calculateDivisor(digitIndex);

  for (int element : elements) {

    int digit = element / divisor % 10;

    buckets[digit].add(element);

  }

}

private int calculateDivisor(int digitIndex) {

  int divisor = 1;

  for (int i = 0; i < digitIndex; i++) {

    divisor \*= 10;

  }

  return divisor;

}

We can replace the "10" in all these places with another base. Better yet, we replace it with a variable so that we can invoke the sorting algorithm with any base.

In the [RadixSortWithDynamicListsAndCustomBase](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithDynamicListsAndCustomBase.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class, you can find the corresponding adjustment:

public class RadixSortWithDynamicListsAndCustomBase implements SortAlgorithm {

  private final int base;

  public RadixSortWithDynamicListsAndCustomBase(int base) {

    this.base = base;

  }

  // All methods not printed here are the same as in RadixSortWithDynamicLists

  private int getNumberOfDigits(int number) {

    int numberOfDigits = 1;

    while (number >= base) {

      number /= base;

      numberOfDigits++;

    }

    return numberOfDigits;

  }

  private Bucket[] createBuckets() {

    Bucket[] buckets = new Bucket[base];

    for (int i = 0; i < base; i++) {

      buckets[i] = new Bucket();

    }

    return buckets;

  }

  private void distributeToBuckets(int[] elements, int digitIndex, Bucket[] buckets) {

    int divisor = calculateDivisor(digitIndex);

    for (int element : elements) {

      int digit = element / divisor % base;

      buckets[digit].add(element);

    }

  }

  private int calculateDivisor(int digitIndex) {

    int divisor = 1;

    for (int i = 0; i < digitIndex; i++) {

      divisor \*= base;

    }

    return divisor;

  }

}

Please note that in the GitHub repository, the getNumberOfDigits() and calculateDivisor() methods are located in the [RadixSortHelper](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortHelper.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class, as other Radix Sort implementations also use them.

In the GitHub repository, you can also find the adapted algorithms for arrays, Counting Sort, and recursive MSD Radix Sort:

* Dynamic lists + variable base: [RadixSortWithDynamicListsAndCustomBase](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithDynamicListsAndCustomBase.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank)
* Arrays + variable base: [RadixSortWithArraysAndCustomBase](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithArraysAndCustomBase.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank)
* Counting Sort + variable base: [RadixSortWithCountingSortAndCustomBase](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RadixSortWithCountingSortAndCustomBase.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank)
* Recursive MSD Radix Sort + variable base: [RecursiveMsdRadixSortWithArraysAndCustomBase](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RecursiveMsdRadixSortWithArraysAndCustomBase.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank)

## Radix Sort Time Complexity

In this chapter, I will show you how to determine the time complexity of Radix Sort. For an introduction to time complexity, see the article "[Big O Notation and Time Complexity](https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/)".

We use the following variables below:

* n = the number of elements to sort
* k = the maximum key length (number of digit places) of the elements to sort
* b = the base (= the number of buckets)

The algorithm iterates over k digit places; for each place, it performs the following operation:

* It creates b buckets. The cost of this is constant in each case.
* It iterates over all n elements to sort them into the buckets. The cost of calculating a bucket number and inserting an element into a bucket is constant.
* It iterates over b buckets and copies a total of n elements from them. The cost for each of these steps is again constant.

We ignore constant parts in the determination of the time complexity. This results in:

The time complexity for Radix Sort is: O(k · (b + n))

The cost is independent of how the input numbers are arranged. Whether they are randomly distributed or pre-sorted makes no difference to the algorithm. Best case, average case, and worst case are, therefore, identical.

The formula looks complicated at first. But two of the three variables are not variable in most situations. For example, if we sort longs with a base of 10, we can replace k with 19 (the maximum possible value for a long is 9,223,372,036,854,775,807) and b with 10.

The formula then becomes O(19 · (10 + n)). We can omit constants; thus, we get:

The time complexity for Radix Sort  
with a known maximum length of the elements to sort  
and with a fixed base is: O(n)

So, for primitive data types like integer and long (for these, we know the maximum length), Radix Sort has a better time complexity than [Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Time_Complexity)!

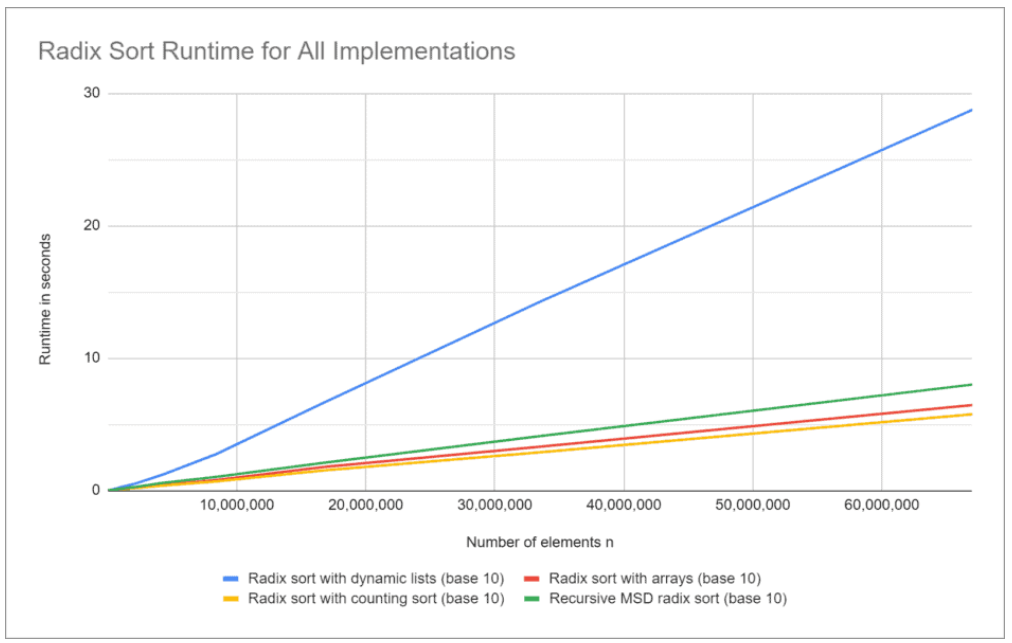
You'll find out whether Radix Sort is actually faster in the next chapter. We will measure the runtime of the various Radix Sort implementations and compare them with each other (and with Quicksort).

## Radix Sort Runtime

In this chapter, I'll show you the results of some performance tests I ran using the [UltimateTest](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/UltimateTest.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) and [CompareRadixSorts](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/comparisons/CompareRadixSorts.java) tools to compare the performance of different algorithms, implementations, and bases.

### Runtime of Different Radix Sort Implementations

The first diagram shows the comparison of the different implementations:

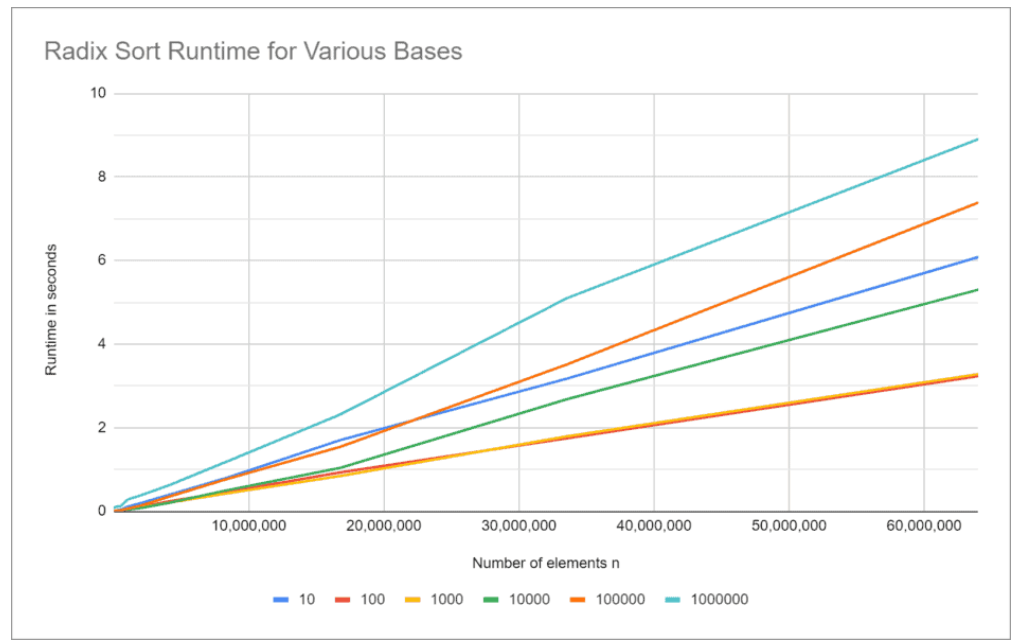


As expected, the implementation with dynamic lists performs worst. The remaining three variants are in a neck-and-neck race, which the Counting Sort implementation wins by a narrow margin, closely followed by the array variant.

We can also see the linear running time O(n) in each case, which we predicted in the previous chapter.

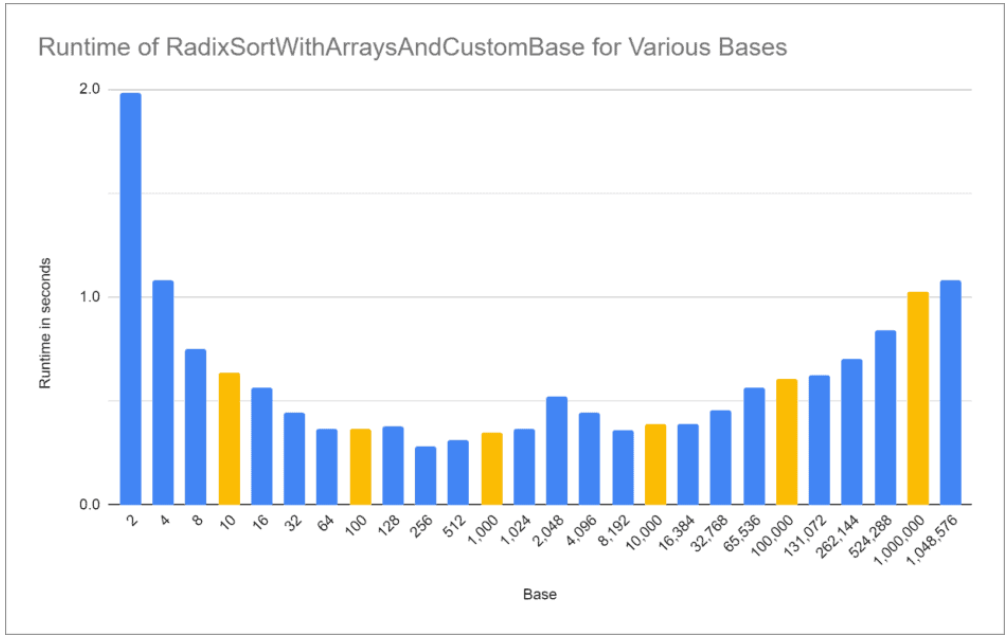
### Effect of the Base on the Runtime

The second diagram shows how the choice of the base affects the runtime of the array implementation:



We can see that the runtime is significantly better for bases of 100 and 1,000 than for smaller and larger bases.

Let's examine this in a little more detail... The third diagram shows finer gradations of the bases with a fixed number of elements (n = 5,555,555):



Both too small and too large a base are bad for performance.

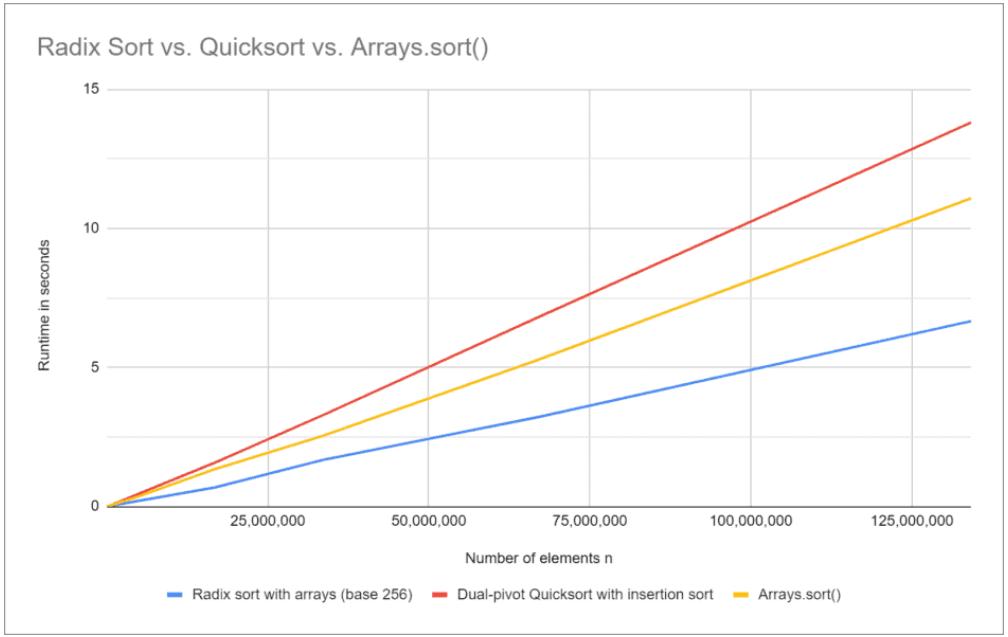
A very small base leads to many iterations. A base that is too large leads to fewer iterations but significantly more buckets within the iterations.

A sweet spot shows up at a base of 256.

### Radix Sort vs. Quicksort

In the following diagram, you can see the runtimes…

* of the Radix Sort array implementation with a base of 256,
* of dual-pivot Quicksort combined with insertion sort (the fastest variant we determined in the [Quicksort tutorial](https://www.happycoders.eu/algorithms/quicksort/" \l "Comparing_All_Quicksort_Optimizations)), and
* of the JDK sort method Arrays.sort(), which also implements an optimized dual-pivot Quicksort.



And indeed, Radix Sort is not only faster in theory – O(n) vs. O(n log n) – but also in practice – comparing it with both the home-implemented Quicksort and the even faster JDK Quicksort implementation Arrays.sort().

So if you need to sort int primitives and performance is critical, you should consider using Radix Sort instead of Java's native Arrays.sort(). Feel free to use the implementation from this article.

That is not true for long primitives. For longs, Arrays.sort() is about 50% faster than my Radix Sort implementation.

## Other Characteristics of Radix Sort

In this concluding chapter, we consider the space complexity, stability, and parallelizability of Radix Sort, as well as its differences from Counting Sort and Bucket Sort.

### Radix Sort Space Complexity

All variants shown in this article require additional memory:

* O(b) for the digit counters (not needed in the dynamic lists variant)
* O(b) for the bucket references (not required for the counting-sort variant).
* O(n) for the contents of the buckets (not needed for the counting-sort variant)
* O(n) for an additional target array (only for the Counting Sort variant)

Each variant thus contains at least one O(b) component and at least one O(n) component.

We can therefore conclude:

The space complexity of Radix Sort is: O(b + n)

There is one exception: recursive MSD Radix Sort with base 2 can do without additional memory for the elements by partitioning them in such a way that by exchanging two elements at a time, all elements whose bit is 1 at the currently considered place are pushed to the right side, and all elements whose bit is 0 are pushed to the left side (similar to Quicksort).

### Is Radix Sort Stable?

You can read about the [definition of stability in sorting methods](https://www.happycoders.eu/algorithms/sorting-algorithms/" \l "Stable_and_Non-Stable_Sorting_Algorithms) in the linked introductory article. In short: elements with the same key keep their original order to each other during sorting.

All Radix Sort implementations shown in this article are stable.

In contrast, the in-place MSD Radix Sort variant discussed in the previous section is not stable (analogous to Quicksort).

### Parallel Radix Sort

Both Radix Sort variants (LSD and MSD) can be parallelized.

#### Parallel MSD Radix Sort

With MSD Radix Sort, after the initial partitioning phase, we can sort all the resulting buckets independently in parallel. Thanks to parallel streams, this is very easy to implement in Java:

Here again, is the corresponding sequential code from the [RecursiveMsdRadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/RecursiveMsdRadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class:

for (Bucket bucket : buckets) {

  if (bucket.needsToBeSorted()) {

    sortByDigit(bucket.getElements(), digitIndex - 1);

  }

}

And here is the parallelized variant ([ParallelRecursiveMsdRadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/ParallelRecursiveMsdRadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repository):

Arrays.stream(buckets)

    .parallel()

    .forEach(

        bucket -> {

          if (bucket.needsToBeSorted()) {

            sortByDigit(bucket.getElements(), digitIndex - 1);

          }

        });

#### Parallel LSD Radix Sort

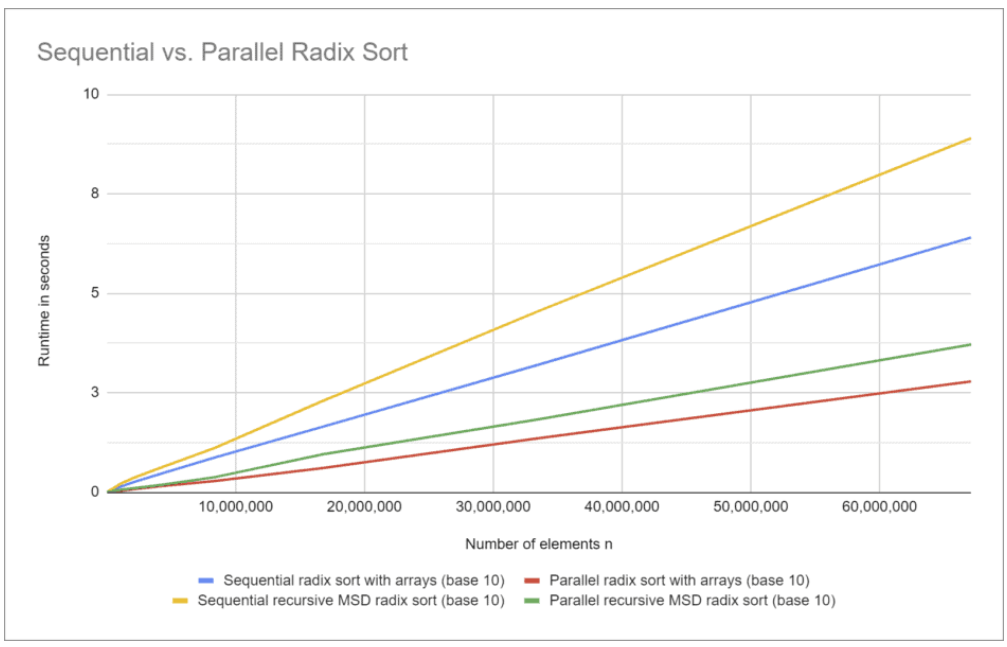
To parallelize LSD Radix Sort, we need to put a little more effort:

1. We divide the input array into segments to be processed in parallel (e.g., according to the number of CPU cores).
2. We calculate in parallel per segment how many elements have to be sorted into which buckets.
3. When step 2 is complete for all segments, we compute a) per bucket, the total number of elements, and b) per segment, the initial write positions for each bucket.
4. We distribute the elements from the segments to the buckets in parallel. Using the initial write positions calculated in step 3, we know at which positions within the buckets we must write from which segments.
5. When step 4 is complete for all segments, we compute per bucket the offset in the target array (as prefix sums over the number of elements in the buckets).
6. We collect the elements from the buckets in parallel. Using the offsets calculated in step 5, we know at which position in the target array the elements of a bucket must start.

You can find the source code in the [ParallelRadixSortWithArrays](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/radixsort/ParallelRadixSortWithArrays.java" \t "https://www.happycoders.eu/algorithms/radix-sort/_blank) class in the GitHub repo. The six steps above are marked in the code with correspondingly numbered comments.

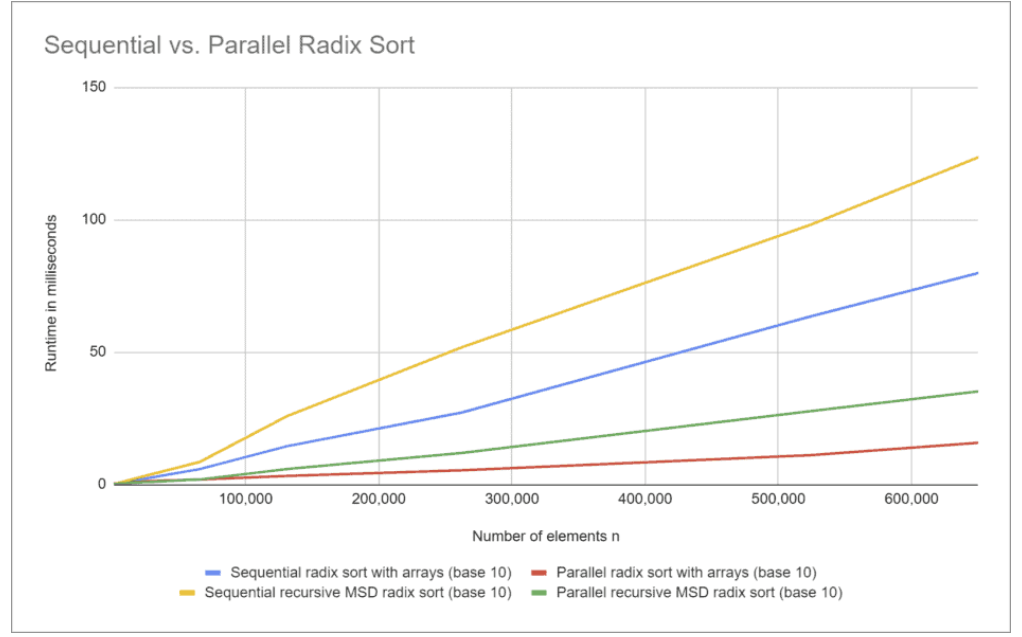
#### Parallel vs. Sequentiell Radix Sort

The following diagram shows the runtime of the parallel variants compared to the sequential variants on a 6-core i7 CPU:



The parallel variants are only about 2.3 times faster, with 67 million elements. That is not even close to factor 6, partly because parts of the code cannot be executed in parallel and partly because the CPU cores have to exchange a lot of data with the main memory (the input array occupies 1 GB).

If we look at a smaller section of the diagram, things look different:



With "only" half a million elements, the parallel Radix Sort variant with arrays is 5.75 times faster than the sequential variant. The CPU cores are almost entirely utilized. That is because the input array is only 2 MB, and the sorting can take place completely in the CPU's L3 cache.

### Radix Sort vs. Counting Sort

Both sorting methods use buckets for sorting. With Counting Sort, we need one bucket for each value. For example, if we wanted to sort integers, we would need about four billion buckets. With Radix Sort, on the other hand, the number of buckets corresponds to the chosen base.

In Radix Sort, we sort iteratively digit by digit; in Counting Sort, we sort the elements in a single iteration.

Counting Sort is therefore primarily suitable for small number spaces.

### Radix Sort vs. Bucket Sort

Bucket Sort first distributes items across a given number of buckets such that all items in each bucket are greater than all items in the previous bucket (e.g., 0-99, 100-199, 200-299, etc.).

After that, each bucket is sorted in itself – either recursively with Bucket Sort – or with another sorting method (which exactly is not specified). Finally, the elements from the sorted buckets are joined.

If this sounds familiar to you – you've met one form of Bucket Sort in this article: [recursive MSD Radix Sort](https://www.happycoders.eu/algorithms/radix-sort/" \l "msd-radix-sort).

## Summary

Radix Sort is a stable sorting algorithm with a general time complexity of O(k · (b + n)), where k is the maximum length of the elements to sort ("key length"), and b is the base.

If the maximum length of the elements to sort is known, and the basis is fixed, then the time complexity is O(n).

For integers, Radix Sort is faster than Quicksort (at least in my test environment). If you need to implement time-critical sorting operations in Java, I recommend you compare Arrays.sort() with an implementation of Radix Sort.

You can find more sorting algorithms in the [overview of all sorting algorithms and their characteristics](https://www.happycoders.eu/algorithms/sorting-algorithms/" \l "Comparison_of_the_Most_Important_Sorting_Algorithms) in the first part of the article series.

If you liked the article, please share it using one of the share buttons at the end. Want to be notified by email when I publish a new article? Then [click here](https://www.happycoders.eu/algorithms/radix-sort/) to join the HappyCoders newsletter.