Published in TechToFreedom

Yang Zhou Following
Dec 26 · 6 min read · ✦ · Listen

ALGORITHM INTERVIEW

Mastering Radix Sort in Python: A Visual Guide

A step-by-step explanation for the classic non-comparison-based sorting algorithm

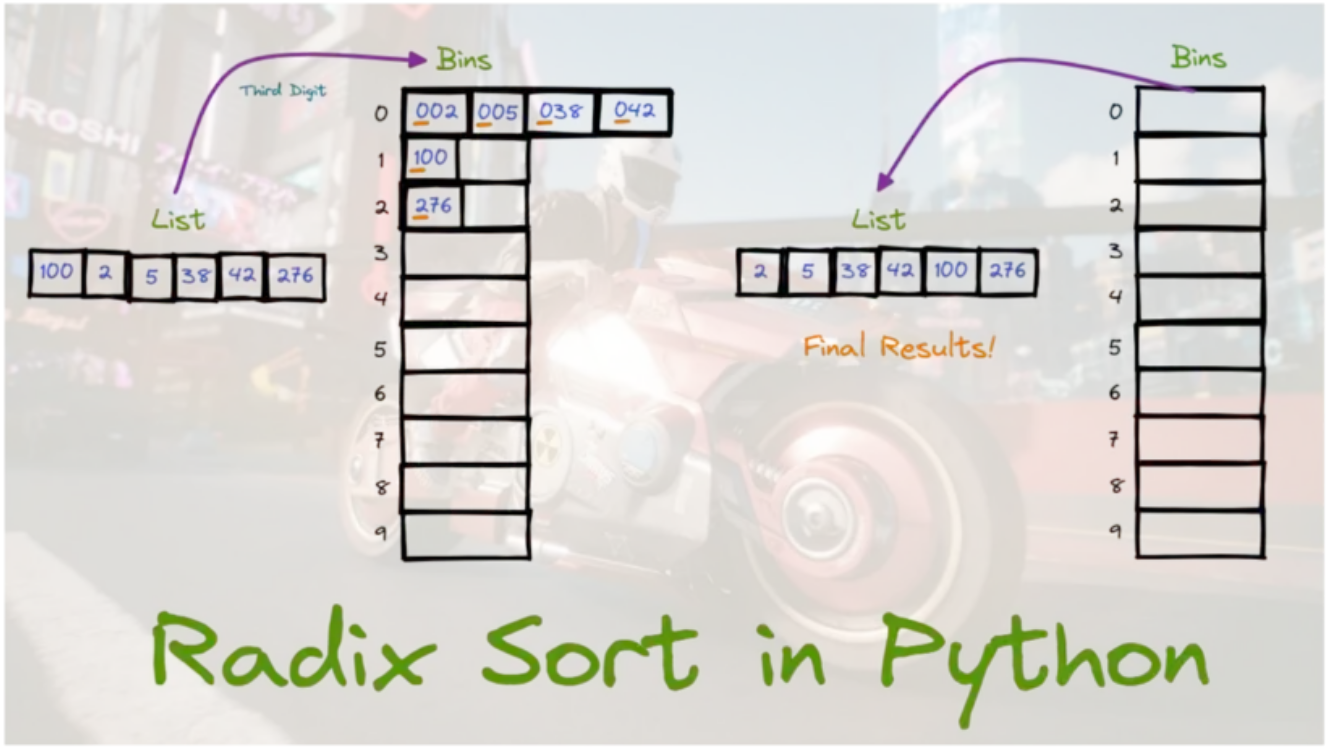

Image made by Yang

Sorting algorithms are frequen nical interviews.

You probably learned a few basic approaches, such as bubble sort, merge sort and quicksort. However, just mastering some comparison-based sorting algorithms is not enough to ace the tech interviews for senior engineers.

After all, in order to determine the order of elements, comparison-based the algorithm must compare two elements at a time and determine which one should come first. This means that the time complexity of these algorithms is at least `O(nlog(n))`, as there must be at least `log(n)` levels of comparisons to sort `n` elements.

> *Can you sort something faster?*

Yes. There is another type of sorting technique — *non-comparison-based sorting algorithms.*

Since there is no comparison, it's possible to be faster than `O(nlog(n))`.

There are also many non-comparison-based sorting algorithms. This article will dive into a famous one — radix sort — in a visual and beginner-friendly way, and provide an elegant Python implementation of it to you.

Table of Contents

What is Radix Sort?

Radix sort is a non-comparison-based sorting algorithm that works by *sorting elements based on the digits of their representation (for integer values) or individual*

characters (for string values). It avoids comparison by creating and distributing elements into buckets according to their radix.

It is a stable sorting method, meaning that the order of equal elements is preserved.

Radix sort has also been called **bucket sort** or **digital sort.**

How Does Radix Sort Work?

The radix sort can be implemented to start at either the most significant digit (MSD) or least significant digit (LSD).

For example, with 2077, we can start to handle it with 2 (MSD) or 7 (LSD).

This article will implement the LSD version. The idea of the MSD version is similar.

Here are the steps for implementing the radix sort algorithm:

1. Determine the base (radix) to use for sorting. This is usually 10 for decimal representation or 2 for binary representation, but other bases can be used as well.
2. Determine the maximum number of digits in the elements of the list. This will be used to determine how many passes are needed to sort the list.
3. Iterate through each digit of the elements, starting with the least significant digit and working up to the most significant digit.
4. For each digit, create a list of buckets/bins to store the elements based on their digit value. For example, if the base is 10, you would create 10 bins for the digits 0–9.
5. Iterate through the elements in the list and add each element to the corresponding bin.
6. Put the elements in bins back into the list, from the first bin to the last and from left to right inside each bin.
7. Repeat the process for the next digit until all digits have been considered.
8. Return the final sorted list.

Too complicated? 😳

A Visual Explanation

No worries 😊. Let's see a visual guide to fully understand the process of the radix sort.

A simple list waiting to be sorted

Assume that we have a list of six integers including 38, 2, 100, 5, 276 and 42.

> *How to sort them using the radix sorting algorithm?* 🧐

The following image shows our current situation:

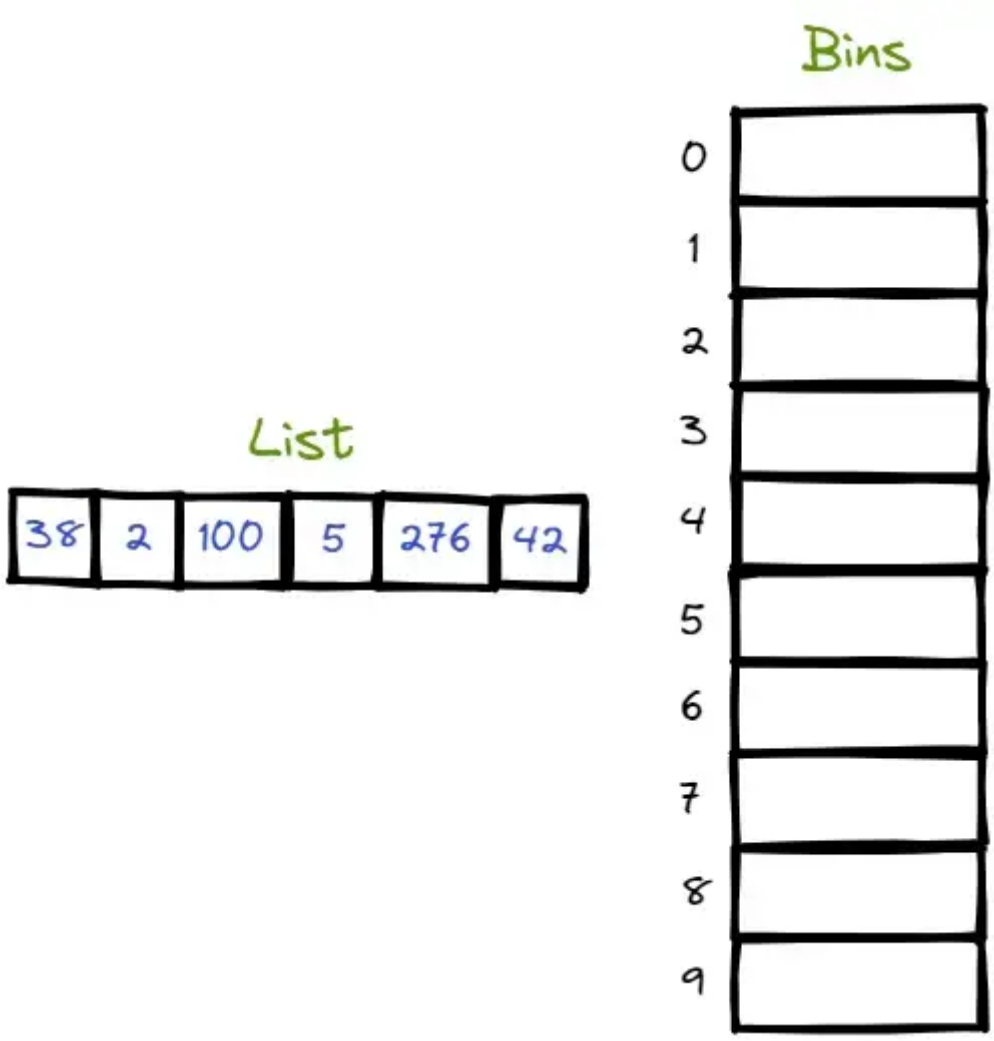

Image 1. The original list and bins

Given that all values are decimal, we will use "10" as the base (radix) for radix sorting. So we prepared ten bins, each for one digit, to store digits 0–9.

Based on the previous guide, we know that the maximum length of the elements in this list is 3. So we only need to check 3 digits for each element (units digit, tens digit, and hundreds digit). It means we need 3 iterations of sorting.

Iteration 1: Sorting the elements based on their units digits

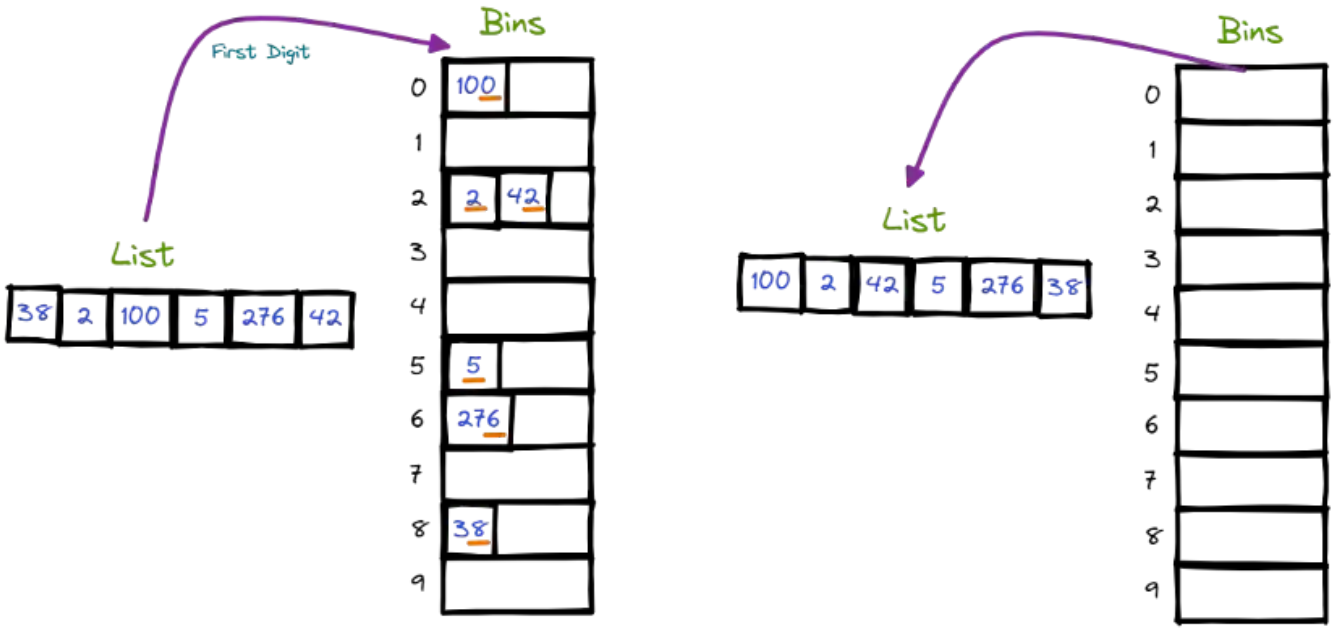

Image 2. The first iteration of sorting

Firstly, as the above image shows, we put every element into bins based on their units digits. After that, we brought back the elements from the bins from the first bin to the last and from the left to right inside each bin.

After this iteration, all units digits of the elements are in order.

You probably got the idea of the radix sorting now.

> If each digit is ordered, then as a whole each element of the list is ordered.

Therefore, all we need to do now is just repeating the same operations for tens digits and hundreds digits.

Iteration 2: Sorting the elements based on their tens digits

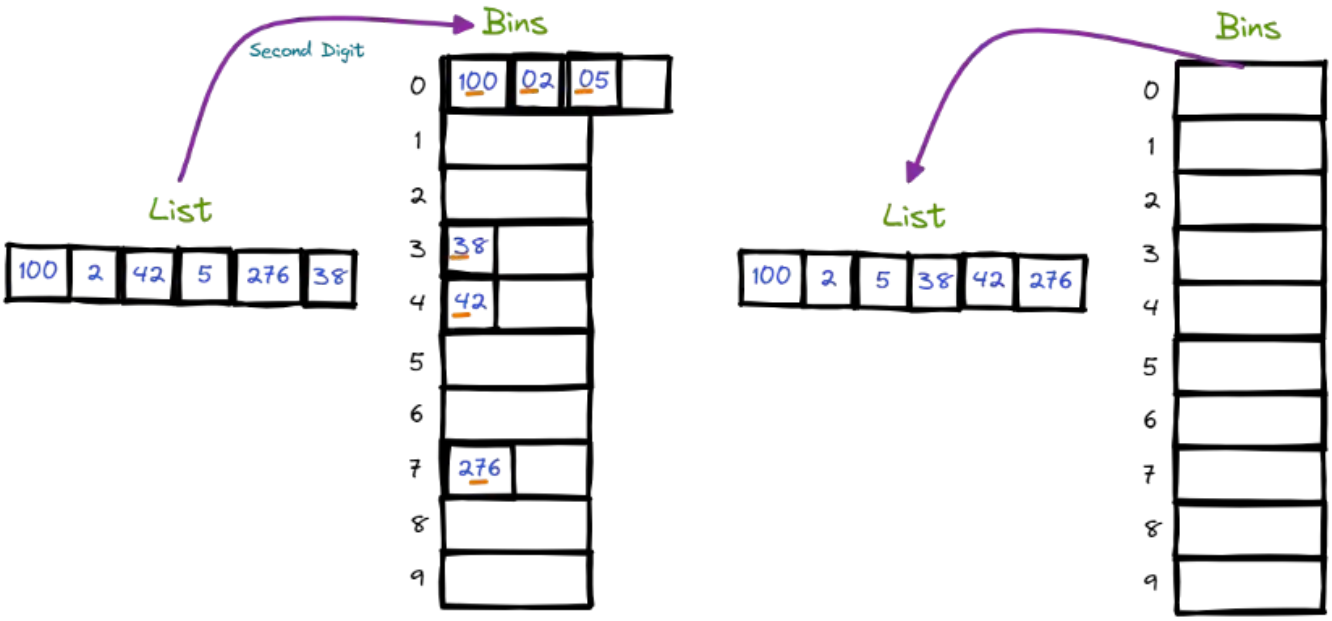

Image 3. The second iteration of sorting

Secondly, we made all tens digits in order as the above image. A small trick here is we can treat “2” as “02” and “5” as “05” to compare their tens digits with other elements.

Iteration 3: Sorting the elements based on their hundreds digits

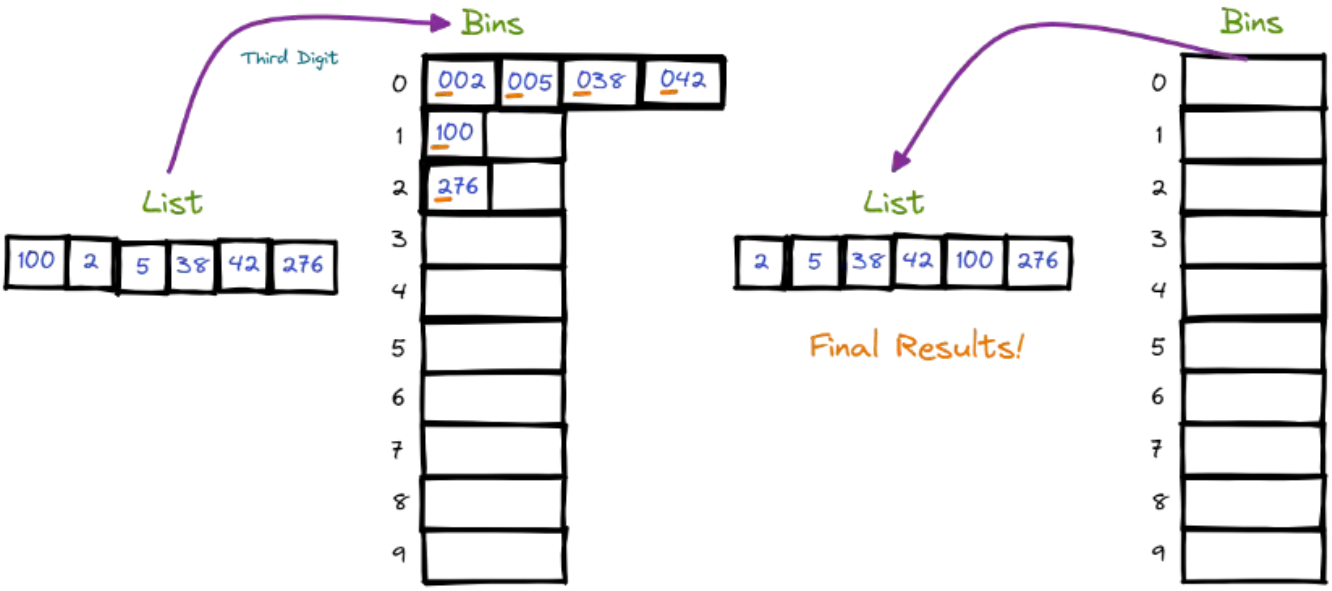

Image 4. The third(final) iteration of sorting

Finally, we repeated the similar process and made the hundreds digits sorted. As expected, all elements are sorted in the original list. 🎉

A Python Implementation of Radix Sort

It’s one thing to understand an algorithm, it’s another to implement it properly.

Every programming language can write a radix sort function. Here is a neat implementation I wrote in Python:

```python
def radix_sort(arr):
    """
    Radix sort starting from the least significant digit(LSD)
    :param arr: The list needs to be sorted
    :return: A sorted list
    """
    # Find the maximum number of digits in the list
    max_digits = max([len(str(x)) for x in arr])

    # Set the base (radix) to 10
    base = 10

    # Create a list of bins to store the digits
    bins = [[] for _ in range(base)]

    # Iterate through each digit, starting with the least significant
    for i in range(0, max_digits):
        # Iterate through the elements in the list
        for x in arr:
            # Extract the i-th digit from the element
            # (starting from the rightest digit)
            digit = (x // base ** i) % base
            # Add the element to the bin for the i-th digit
            bins[digit].append(x)
        # Combine the bins back into the list, starting with the elements in the 0 queue
        arr = [x for queue in bins for x in queue]
        # Clear the bins for the next iteration
        bins = [[] for _ in range(base)]

    return arr


# Test the function
print(radix_sort([38, 2, 100, 5, 276, 42]))
# Output: [2, 5, 38, 42, 100, 276]
```

radix_sort.py hosted with ❤ by GitHub view raw

A Python implementation of radix sort

The code and relative comments are good enough to explain the logic. 😎

Time and Space Complexity of Radix Sort

As mentioned, the advantage of radix sort is that it can be faster than comparison-based algorithms like quicksort or mergesort, which rely on element comparisons and have a time complexity of `O(n*log(n))`.

However, this is an inaccurate analysis.

Radix sort has a time complexity of `O(n*m)`, where `n` is the size of the input list and `m` is the maximum number of digits in the elements of the list.

Therefore, it's not necessarily that `O(n*m)` is always faster than `O(n*log(n))`. It depends on the comparison between `m` and `log(n)`.

Radix sort has a space complexity of `O(n + b)`, where `n` is the size of the input list and `b` is the base (radix) used for sorting. Because the algorithm requires additional space to store the list of bins used to sort the elements.

Improvements and Variants of Radix Sort

The sorting algorithm is still a popular research area. There are further improvements and variants of the radix sorting. They are outside the scope of this article.

For example, the radix sort can be used parallelly to leverage the computing resources of CPUs.

Good papers for your reference:

- Load Balanced Parallel Radix Sort
- A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort

Conclusion

Radix sort is a great non-comparison-based sorting algorithm. It is relatively easy to implement and can be modified to handle different data types or bases. It is an excellent choice for sorting large lists of integers or strings.

However, requirements and usage scenarios may vary. Sometimes the radix sort is faster, but sometimes it's not. We need to choose sorting algorithms according to specific needs.

Overall, radix sort is a useful tool to have in your engineering toolkit. Knowing how to implement and analyze radix sort can demonstrate your understanding of sorting algorithms and your ability to think about problems in different ways. This can be useful in technical interviews and in your career as a software engineer.

Thanks for reading. ❤️

Join Medium through my referral link to access millions of great articles that help to level up your engineering career:

Programming | Algorithms | Python | Sorting Algorithms | Data Science