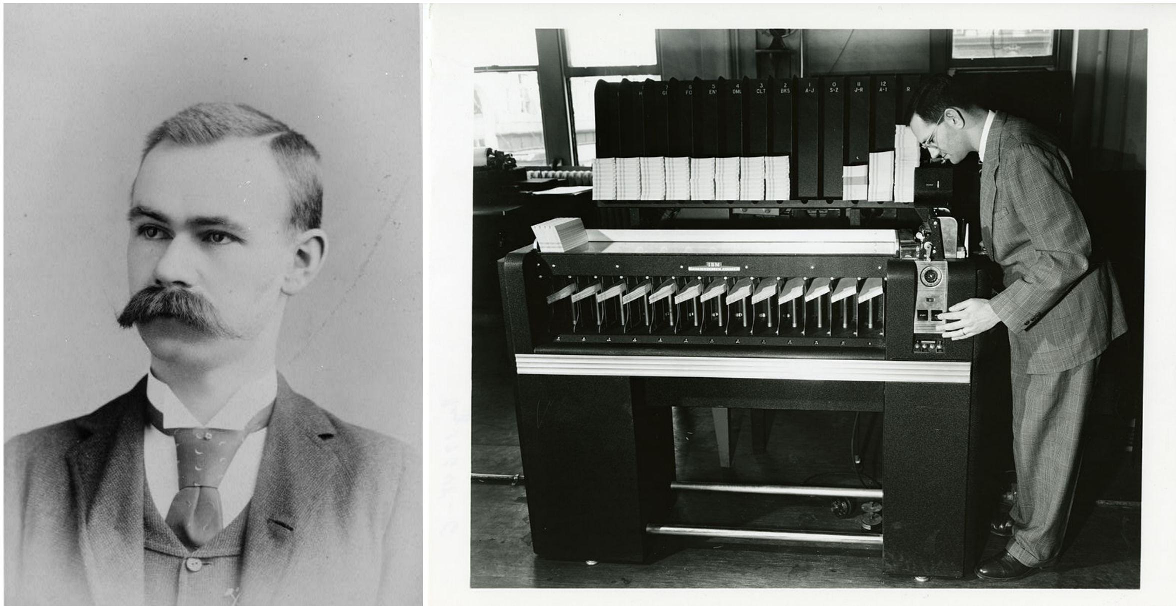
Counting sort and radix sort

Nick Smallbone

December 11, 2018



Radix sort is the oldest sorting algorithm which is still in general use; it predates the computer. On the left is Herman Hollerith, the inventor of radix sort. In the late 1800s, he built enormous electromechanical machines which used radix sort to tabulate US census data; he set up a company to sell the machines, and the company later became IBM. On the right you can see a 1950s IBM sorting machine running the radix sort algorithm.

Radix sort is unlike the sorting algorithms we have seen so far, in that it is not based on comparisons ($\leq$). However, it only works on certain kinds of data – it is mainly used on numerical data, and is often faster than comparison-based sorting algorithms. Before seeing radix sort, we will look at a simpler algorithm, *counting sort*.

1 Counting sort #1: sorting a list of digits

Suppose you want to sort the first 100 digits of π, in ascending order, by hand. How would you do it?

31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679

Counting sort is one approach to sorting lists of digits. It can be carried out by hand or on a computer. The idea is like this. First we count how many times each digit occurs in the input data (in this case, the first 100 digits of π):

Digit	0	1	2	3	4	5	6	7	8	9
Number of occurrences	8	8	12	12	10	8	9	8	12	14

We can compute this table while looking through the input list only once, by keeping a running total for each digit. Then, referring to the table, we can read off the answer: there are 8 zeroes, followed by 8 ones, followed by 12 2s, followed by…

00000000111111112222222222223333333333334444444444555555556666666667777777788888888888899999999999999

This algorithm can be implemented on a computer, using an array of size 10 to hold the digit counts. More generally, it can be used to sort a list of values that are drawn from any finite set, as long as that set is known and reasonably small. It takes $O(n)$ time and is remarkably simple: one loop through the input list to compute the counts, then one loop which produces the output list.

Exercise: write a Java method void countingSort(int[] array) that implements this algorithm. You may assume that the elements of array are digits, i.e. numbers in the range 0..9.

2 Counting sort #2: sorting a list by a key

Suppose now that we want to sort a list of words in order of their length. For example, take the following words and their lengths:

bee 3, elephant 8, tiger 5, dog 3, cat 3, mouse 5, octopus 7, lion 4, dolphin 7, pigeon 6

We cannot do this using the basic counting sort algorithm, even though the word lengths are all rather small. In this section, we will see an extended version of counting sort which can be used to sort a list of values by a *key*. They key must be drawn from some finite set and is typically a small integer. In this example, the values are the (word, length) pairs and the keys are the lengths.

We start by computing the number of occurrences of each key, just as in the basic counting sort:

Key (length of word)	3	4	5	6	7	8
Number of occurrences	3	1	2	1	2	1

Note that from this table we can see that, once we have sorted the words, we should expect the words in the sorted list to have the following keys (lengths):

3, 3, 3, 4, 5, 5, 6, 7, 7, 8

Now we are going to compute a second table that records, for each key, the position in the output list where the *first value* having that key should appear. For the list of words, the position table is as follows:

Key (length of word)	3	4	5	6	7	8
Position	0	3	4	6	7	9

That is, the first word of length 3 will appear at position 0, the first word of length 4 will appear at index 3, and so on. In general, for each key, the starting position is the *sum* of the number of occurrences of all lesser keys. For example, the starting position for words ending in 6 is $3 + 1 + 2 = 6$. We can compute the position table in one pass through the "number of occurrences" table by keeping a running sum.

Now, we create an array to hold the sorted list, initially empty. We loop through the values in the input list, and for each value x we:

1. Compute x's key and look up that key in the position table, to get an index i.
2. Write x to index i in the array.
3. Increment the relevant index in the position table, so that instead of i it is $i + 1$.

For example, after processing the first three values in the list of words ("bee 3, elephant 8, tiger 5"), the output array and position table will look as follows (changes from the initial position table are marked in bold):

0	1	2	3	4	5	6	7	8	9
bee 3				tiger 5					elephant 8

Key (length of word)	3	4	5	6	7	8
Position	**1**	3	**5**	6	7	**10**

The next value, dog 3, will be stored at index 1, and the position table afterwards will look as follows:

Key (length of word)	3	4	5	6	7	8
Position	**2**	3	5	6	7	10

After processing every value in the word list, we have the following sorted array and position table:

0	1	2	3	4	5	6	7	8	9
bee 3	dog 3	cat 3	lion 4	tiger 5	mouse 5	pigeon 6	octopus 7	dolphin 7	elephant 8

Key (length of word)	3	4	5	6	7	8
Position	3	4	6	7	9	10

Make sure you understand this algorithm and why it works before moving on. Note that this algorithm loops twice through the input array so it is still $O(n)$ time.

3 Radix sort

Finally, suppose that we want to sort the following list of 3-digit numbers:

$$314, 159, 265, 358, 979, 323, 846, 264, 338, 327, 950, 288$$

Counting sort is inappropriate here, because it would require an array of 1000 elements to hold the counts. But we can sort the list using *radix sort*, which sorts a list of integers by doing a series of counting sorts. It can also be used to sort lists of values by an integer key. It takes $O(n)$ time.

The key idea is that counting sort is what is called *stable*. In a stable sorting algorithm, when two values have the same key, their relative order after sorting is the same as their order before sorting. In the list of words in the previous section, looking at (e.g.) the words of length 5, tiger comes before mouse – and tiger also comes before mouse after we have sorted the list. Similarly, the words of length 3 occur in the order "bee, dog, cat" both in the input list and in the sorted list. Counting sort is stable and so the relative order of values having the same key is always preserved in this way.

Now, back to the algorithm. We take the list of numbers and sort it by the *final* digit, using counting sort (the final digit is the key). We get the following list, which we will call A:

$$950, 323, 314, 264, 265, 846, 327, 358, 338, 288, 159, 979$$

Next, we take A (not the original input list) and sort it by its *second* digit. You should try doing this by hand. The result you should get is:

$$314, 323, 327, 338, 846, 950, 358, 159, 264, 265, 979, 288$$

We will call this array B. Notice that B is sorted by its *last two* digits, not just the second digit! This happens because counting sort is stable. The numbers in B are of course sorted by their second digit. But if two numbers have the same second digit, they appear in B in the same relative order that they appeared in A (by stability). The numbers in A were sorted by their $last$ digit, so when two numbers in B have the same second digit, they appear in the order of their last digit. This effectively means we have sorted the numbers by their last two digits.

Now, the final step is to sort B, using the *first* digit as the key. Again, stability means that if two numbers have the same first digit, their relative order is unchanged, and therefore they appear sorted by the last two digits. Therefore, the resulting list of numbers is sorted!

$$159, 264, 265, 288, 314, 323, 327, 338, 358, 846, 950, 979$$

In general, the algorithm to sort an array of d-digit numbers is as follows:

```
for i = 1 to d
  sort the array, using the key k(n) = (d-i)th digit of n
```

It can also be used to sort an array of values by a numeric key. If the maximum size of the numbers is not known ahead of time, it may be necessary to first loop through the input data in order to determine d.

Complexity

The complexity of radix sort of an array of d-digit numbers of length n is $O(nd)$, which at first glance appears to be better than $O(n \log n)$! However, if we have a list of n distinct numbers, one of the numbers must have at least $\log_{10} n$ digits, so the complexity is still $O(n \log n)$. The constant factors are rather good, because the $\log$ is base 10.

For efficient implementation, radix sort does not usually use base 10, but rather base 16 or another power of 2. If memory use is not a concern, then base $2^8 = 256$ might be reasonable, and allows you to sort an array of 32-bit numbers with 4 passes of counting sort. The performance is still $O(n \log n)$,but the constant factors are even better because the $\log$ is base 256.

The book shows that comparison-based sorting requires at least $\log_2(n!) = O(n \log n)$ comparisons to sort a list of length n. Using Stirling's formula one can show that this number is at least $0.8n \log_2 n$ for reasonably large values of n. Base-256 radix sort easily beats this lower bound: it executes $\log_{256} n$ passes of counting sort, and each pass loops through the array twice, giving a total of $2n \log_{256} n = (2n \log_2 n)/8 = 0.25n \log_2 n$ operations performed – and this is counting all array operations, whereas for counting sort we were only counting comparisons.

To summarise, radix sort has $O(n \log n)$ complexity, but with constant factors that easily beat comparison-based sorting.

4 Conclusion

Radix sort is an algorithm for sorting lists of numbers that beats the lower bound for comparison-based sorting. It works by repeated use of counting sort, a linear-time algorithm for sorting a list of objects by a key drawn from a finite set. It can be used as a specialised algorithm for sorting objects by an integer key. It is also easier to implement then the fast comparison-based sorting algorithms!