CMSC 351: RadixSort

Justin Wyss-Gallifent

March 25, 2021

1 What it Does ............................................. 2
2 How it Works .............................................. 2
3 Notes ..................................................... 3
4 Pseudocode ............................................... 4
5 Time Complexity ......................................... 4
6 Auxiliary Space .......................................... 6
7 Stability .................................................. 6
8 In-Place .................................................. 6
9 Thoughts, Problems, Ideas ........................... 7
10 Python Test .............................................. 8
1 What it Does

Sorts a list of integers in some base, aka radix. For example positive integers in base 10 with \( d \) digits all look like \( x_d \ldots x_1 \) for digits \( 0 \leq x_i \leq 9 \) and for example positive integers in base 2 with \( d \) digits all look like \( x_n \ldots x_1 \) for digits \( x_i \in \{0, 1\} \).

2 How it Works

RadixSort is technically speaking neither comparison-based nor not comparison based. Rather, RadixSort sorts progressively by digits and how that auxiliary sorting is done is somewhat flexible. Consider this example:

Example 2.1. For example consider the following list. Observe that we have preprended 0s as necessary to equalize the lengths.

\[ 170, 045, 075, 090, 802, 024, 002, 066 \]

First we identify each right-most (least significant) digit:

\[ 170, 045, 075, 090, 802, 024, 002, 066 \]

We sort the list according to that digit, preserving order within that group:

\[ 170, 090, 802, 002, 024, 045, 075, 066 \]

Then we identify the next digit, moving leftwards:

\[ 170, 090, 802, 002, 024, 045, 075, 066 \]

We sort the list according to that digit, preserving order within that group:

\[ 802, 002, 024, 045, 066, 170, 075, 090 \]

Then we identify the next digit, moving leftwards:

\[ 802, 002, 024, 045, 066, 170, 075, 090 \]

We sort the list according to that digit, preserving order within that group:

\[ 002, 024, 045, 066, 075, 090, 170, 802, \]

Now we are done.
3 Notes

Note 3.0.1. The method by which we sort by digit doesn’t really matter as long as it is stable, meaning it preserves the order of identical digits. Really therefore any standard stable sorting method will work. Of course the time complexity of that underlying sorting method will affect the time complexity of RadixSort but it doesn’t matter as far as the definition of RadixSort. Typically a stable implementation of CountingSort is used, however, and we’ll write RadixSort+CountingSort for clarity. We’ll see why when we discuss time complexity.

Note 3.0.2. Note that when we do our underlying sort we basically identify each element by the digit we are using to sort it. In other words, for example, for our initial list when we identify each right-most digit we pretend each element is just that digit:

\[
\begin{array}{cccccccc}
0 & 170 & 045 & 075 & 090 & 802 & 024 & 002 & 066
\end{array}
\]

Note 3.0.3. The fact that we sort by least significant digit first may seem counterintuitive given that if we sorted the numbers by hand we may go for the most significant digit first. However if we did sort by the most significant digit first we would then need to sort by the second most significant while keeping each group with similar most significant together, basically meaning we’d be doing many separate sorts each successive time.

When we first sort by the least significant digit then, when we sort by the second least significant digit the least significant digits stay ordered within the sort order of each second least significant digit grouping, and so on.

If this is unclear try sorting by the most significant digit first and then see what happens when we sort by the second most significant digit.
4 Pseudocode

Assuming that each element in our array \( A \) has the form \( x_dx_{d-1}...x_1 \) the (pseudo-)pseudocode is as follows:

```
\PRE: A is a list of integers.
for i = 1 to d
    stable sort A using digit i
end
\POST: A is sorted.
```

5 Time Complexity

In what follows I’ve used \( \Theta \) and just used “time complexity” but in reality we could use \( O \) or \( \Omega \) and substitute in worst-, best-, or average-case as needed, provided we understand the details.

Suppose our list has \( n \) items and suppose the time complexity of the underlying sort is \( \Theta(f) \) for some function \( f \). Then if the entries in our list can all be expressed with \( d \) digits (or fewer) then the sub-sort runs \( d \) times and the resulting time complexity of RadixSort is \( \Theta(df) \).

**Example 5.1.** Suppose our list contains \( n \) integers between 000 and 999 inclusive. Let’s use base \( b = 10 \) so then we need \( d = 3 \) of them. If we use Radix-Sort+CountingSort then the underlying CountingSort has \( k = \max = 9 \) and therefore is \( \Theta(n + 9) \) and so the resulting RadixSort will be \( \Theta(3(n + 9)) = \Theta(n) \).

**Example 5.2.** Suppose our list contains \( n \) integers between 0 and 255 inclusive. Let’s use base \( b = 2 \) so that we need \( d = 8 \) of them. If we use Radix-Sort+CountingSort then the underlying CountingSort has \( k = \max = 1 \) and therefore is \( \Theta(n + 1) \) and so the resulting RadixSort will be \( \Theta(8(n + 1)) = \Theta(n) \).

It may seem like we’re always going to get \( \Theta(n) \) but both these examples depended upon knowing (or choosing) \( b \) and \( d \) beforehand and having both independent of \( n \).

**Example 5.3.** Suppose our list contains \( n \) integers between 0 and \( 2^n - 1 \) inclusive, so as the list gets longer, so can the values. Let’s use \( b = 2 \) so that we need \( d = n \) of them. If we use RadixSort+CountingSort then the underlying CountingSort has \( k = \max = 1 \) and therefore is \( \Theta(n + 1) \) and so the resulting RadixSort will be \( \Theta(n(n + 1)) = \Theta(n^2) \).

Note that in this case it may be better to use something like QuickSort with average case \( O(n \lg n) \).
Interestingly we can make unexpected choices for the base which can change the time complexity quite a bit.

**Example 5.4.** Suppose our list contains \(n\) integers between 0 and \(n-1\) inclusive and we slyly choose \(b = n\) so that we only need \(d = 1\) of them. If we use RadixSort+CountingSort then the underlying CountingSort has \(k = b = n\) and therefore is \(\Theta(n+n)\) and so the resulting RadixSort will be \(\Theta(1(n+n)) = \Theta(n)\).

In this case of course we’re really not doing RadixSort at all, we’re just doing a single iteration of CountingSort.

The previous example suggests that we could always use base \(b = n\) and therefore really just use (a single iteration of) CountingSort and while this isn’t wrong, we should be aware of the overhead involved.

**Example 5.5.** Suppose our list contains \(n\) integers between 0 and 999 inclusive. Using \(b = 1000\) results in essentially a single iteration of CountingSort with \(k = max = 999\) which requires \(n + k + n + n = 3n + 999\) iterations as per the loop structure in CountingSort. On the other hand using \(b = 10\) results in three iterations of CountingSort with \(k = max = 9\) each requiring \(n + k + n + n = 3n + 9\) iterations for a total of \(3(3n + 9) = 9n + 27\) iterations within CountingSort.

Depending on the time requirements inside the loops it’s entirely possible that the second is quicker for some \(n\). For example if each iteration within CountingSort took time 1 then \(9n + 30 \leq 3n + 1000\) when \(n \leq 162\) so for small lists it’s definitely the case that RadixSort+CountingSort is better than just using a huge base.
6 Auxiliary Space

The auxiliary space of RadixSort depends upon the underlying sort mechanism. Note that if we’re using RadixSort+Counting sort then smaller bases are preferable to keep the auxiliary space to a minimum.

7 Stability

RadixSort is stable because the underlying sort mechanism is chosen to be stable and the RadixSort wrapper doesn’t introduce any instabilities.

8 In-Place

This also depends upon the stability of the underlying sort mechanism.
9 Thoughts, Problems, Ideas

1. Suppose $M$ is a fixed positive integer and $b$ is a fixed base. Suppose we have a list of length $n$ which contains integers between 0 and $M$ inclusive. Explain why, if RadixSort+CountingSort is used to sort the list, the time complexity is the same.

2. Suppose $b = 2$ is the fixed base. Suppose that we have a list of length $n$ which contains integers between 0 and $n$ inclusive. Show that the RadixSort+CountingSort time complexity will be $\Theta(n \log n)$.

   Hint: How many digits are needed to represent the integers 0 through $n$ inclusive in base 2? Does the fixed base matter?

3. Suppose that we have a list of length $n$ which contains integers between 0 and $n-1$ inclusive. Show that the RadixSort+CountingSort time complexity can in fact be made to be $\Theta(n)$. Hint: What can we make $b$ equal to?

4. Explain how RadixSort+CountingSort can sort $n$ integers between 0 and $n^2 - 1$ inclusive in $\Theta(n)$ time with two iterations of CountingSort.

5. Explain how RadixSort+CountingSort can sort $n$ integers between 0 and $n^3 - 1$ inclusive in $\Theta(n)$ time with three iterations of CountingSort.

6. Explain how RadixSort+CountingSort can sort $n$ integers between 0 and $n^n - 1$ inclusive in $\Theta(n^2)$ time with $n$ iterations of CountingSort.

7. Suppose that $M$ is a fixed positive integer and suppose we have a list of length $n$ which contains integers between 0 and $M$ inclusive. Show that the RadixSort+CountingSort time complexity will be $\Theta\left(\frac{n \log M}{\log n}\right)$.

8. Radix sort is the method used for alphabetizing whereby the underlying sort simply sorts by character in some way. Don’t worry about how this underlying sort works. Show how RadixSort works on the list of words:

   ANTE,ANTI,AMEX,BITE,BARK,INTO,INIT,
   LARK,PARK,PACK,RITE,UNTO,UNIT,ZOOT

   You only need to show the result of each RadixSort loop iteration.

9. Suppose a list $A$ contains integers between 0 and 999 inclusive. For indices $i, j$ we wish to compare $A[i]$ and $A[j]$ and return the minimum but we are not permitted to use any sort of comparison. Explain how we could use RadixSort to do this. What would the $\Theta$ time complexity be? Explain.
10 Python Test

We assume arrays are global and we use Counting Sort for the underlying sort. The Radix here is 10.

```python
import random
# This unstable version of counting sort
# sorts by the digit passed to it.
# digit = 1,10,100,etc.
def countingsort(A,digit):
    n = len(A)
    ANEW = [0] * n
    C = [0] * 10
    for i in range(0,n):
        sdigit = int((A[i]/digit) % 10)
        C[sdigit] = C[sdigit] + 1
    for i in range(1,10):
        C[i] = C[i] + C[i-1]
    for i in range(n-1,-1,-1):
        sdigit = int((A[i]/digit) % 10)
        ANEW[C[sdigit]-1] = A[i]
        C[sdigit] = C[sdigit] - 1
    for i in range(0,n):
        A[i] = ANEW[i]
# The radixsort function sorts by increasing digit.
def radixsort(A):
    maxval = max(A)
    d = 1
    while d < maxval:
        print("Sorting by radix: "+str(d))
        countingsort(A,d)
        print("Result: "+str(A))
        d = 10*d
A = []
for i in range(0,15):
    A.append(random.randint(0,1000))
n = len(A)
print(A)
radixsort(A)
print(A)
```
Output:

```
[91, 546, 43, 88, 954, 731, 975, 285, 737, 519, 830, 686, 744, 637, 322]
Sorting by radix: 1
Result: [830, 91, 731, 322, 43, 954, 744, 975, 285, 546, 686, 737, 637, 88, 519]
Sorting by radix: 10
Result: [519, 322, 830, 731, 737, 637, 43, 744, 546, 954, 975, 285, 686, 88, 91]
Sorting by radix: 100
Result: [43, 88, 91, 285, 322, 519, 546, 637, 686, 731, 737, 744, 830, 954, 975]
```

[43, 88, 91, 285, 322, 519, 546, 637, 686, 731, 737, 744, 830, 954, 975]