9 - radix sort

Radix sort is very old, older than electronic computing; it dates back to sorting methods developed by Herman Hollerith in the 1880s to use with punch cards and tabulating machines. It allowed the 1890 US census to be counted in one year, the 1880 census had taken eight and had been feared that the 1890 census wouldn't be counted in time for the 1900 census. Hollerith went on to start a company, the Tabulating Machine Company, which worked on censuses in lots of different countries and was one of the four companies which merged to form what became IBM. There are a number of varieties of radix sort, here we will look at least significant digit sort.

Radix sort algorithms rely on the data being of the right form; for our description we assume the data is made up of interger entries. The algorithm is a bucketing algorithm, not a comparison one, as we have had before, it involves making piles of entries according to some property of the entry. These piles are called buckets, possibly reflecting the way radix worked when it ran on punch cards and swept them into buckets.

Least significant digit radix sort works by going through the array one by one and putting the entries in ten buckets according to the value of the last, or least significant, digit. The piles from each bucket are then stacked back together from lowest to highest so the entries are arranged in order of their last digit. The same process is carried out on the next significant digit, the key thing is that entries that go into the same bucket do so in the same order they were in in the array sp the sorting of their least significant digit is preserved.

If the initial array was $\{27, 17, 23, 14\}$ for example, then the first run through will put them into three buckets, the **3** bucket will hold 23, the **4** buckets 14 and the **7** bucket 27 and 17; when the piles are put back in order the array is $\{23, 14, 27, 17\}$. This is then bucketed by the second digit, so the **1** bucket holds 14 and 17, in that order since that's the order they are in in the most recent version of the array, the **2** bucket hold 23 and 27 so when they are put together the array is sorted: $\{14, 17, 23, 27\}$. This is repeated until all the significant figures have been sorted. Table 1 gives an example.

While punch cards were literally put in buckets the usual way to radix sort arrays is by counting. Basically, the algorithm runs through the array twice. The first time through it counts how many entries there are in each bucket, this allows the position each entry should have to be worked out, on the second run the entries are put in the correct place. An example of this is given in Table 2 and code for radix sort on integers is given in Table 3.

As for complexity, for each significant figure there are $O(n)$ steps as the algorithm runs through the list; in fact we know it runs though the list a few times, once to calculate the bucket values, once to place the entries in the new array and another time to copy the new array back to the old one. It does this for each significant digit, if there are k significant digits this makes the algorithm $O(kn)$. We don't treat the k as a constant since it may depend on k depending on the nature of the array. For example, if the original array contain values from one to n that have been shuffled, the number of significant digits in n is $k = \log_{10} n$ so the algorithm is $O(n \log n)$. Conversely, maybe the array contains lots and lots of repititions but only has entries up to some $n_0 << n$, then $k = \log_{10} n_0$ is a constant and the run time is $O(n)$. Thus, radix sort is extemely good in certain circumstances, but in general it is outperformed by efficient comparison based algorithms.

	335	9383	45	9	886	2777	69	7793	383	386
0	9383	7793	383	335	45	886	386	2777	9	69
1	9	335	45	69	2777	9383	383	886	386	7793
2	9	45	69	335	9383	383	386	2777	7793	886
3	9	45	69	335	383	386	886	2777	7793	9383

Table 1: Radix sort in action. In line 0 the units are sorted, in line 1 the tens, in line 2 the hundred and in line 3 the thousands. The key point if that the order inside a bucket preserves the order that was already there, so when 45 and 69 both go in the **0** hundreds bucket, 45 is before 69 because they were already in that order in the previous line.

							1	**3**
a	21	53	63	41	61	23	3	3
b1							3	6
b2						23	3	5
b3			61			23	2	5
b4		41	61			23	1	5
b5		41	61		63	23	1	4
b6		41	61	53	63	23	1	3
b7	21	41	61	53	63	23	0	3

Table 2: The bucket in action. Here we have a list of numbers being sorted by the least significant digit, for illustrative purposes the entries all have least significant digit 1 or 3. The original array is marked a, the last two digits represent the **1** and **3** buckets, there are three entries in each bucket. The new array b is used to store the contents of the buckets, this is done in lines b1-b7. Since 1 comes before 3, the **1** bucket entries are assigned the first three slots in b and the **3** bucket the slots 4-6. Thus, the number for the **3** bucket is set to 6, which is the total number of entries in buckets for values less than or equal to 3. First 23 is considered in line b2, since its digit is 3 it is in the **3** bucket. The number for the **3** bucket is 6 so this entry is placed in position 6 and one is taken away from the number for the **3** bucket. In line b3 the 61 is considered, it is in the **1** bucket and the number for the **1** bucket is 3, this is placed in slot 3 and the number for the **1** bucket is reduced to 2. This carries on until all the entries are placed in b.

```
void radix_sort(int a[], int n, int a_bound)
{

    int exp_max=log10((double)a_bound)+1;
    int a_sort[n];
    int i,exp;

    for(exp=0;exp<exp_max;exp++){
        int buckets[10]={0};
        int digit=pow(10,exp);

        for(i=0;i<n;i++)
            buckets[a[i]/digit%10]++;

        for(i=1;i<10;i++)
            buckets[i]+=buckets[i-1];

        for(i=n-1;i>=0;i--){
            buckets[a[i]/digit%10]--;
            a_sort[buckets[a[i]/digit%10]]=a[i];
        }

        for(i=0;i<n;i++)
            a[i]=a_sort[i];

    }
}
```

Table 3: Radix sort. Here a_bound is the largest possible entry, this might be calculated by going through the entries and looking for the largest one, or it might known, this code is available as part of `radix_sort.c` and in this code the user specifies the bound on the random numbers, this bound is used to get a_bound. A more sophisticated version of this program has a marker to prevent resorting of the elements that have fewer significant digits, this can be seen in `radix_sort_better.c`.