CI583: Data Structures and Operating Systems

A sorting algorithm which is impressively efficient...

Radix sort

Our next sorting algorithm works completely differently to any of the ones we've seen so far, and does it without actually comparing values to each other. What's more, it has $O(n)$ time complexity!

This is radix sort. For each element in the input, radix sort looks at one digit at a time starting with the least significant. Elements with the same value for that digit are "thrown" into the same bucket.

Radix sort

Consider sorting the following data:

[310, 213, 023, 130, 013, 301, 222, 032, 201, 111, 323, 002, 330, 102, 231, 120].

Note that some elements are padded so that all elements have the same number of digits.

Radix sort

We start by collecting elements with the same first digit.

Bucket	**Contents**
0	310 130 330 120
1	301 201 111 231
2	222 032 002 102
3	213 023 013 323

Radix sort

We start by collecting elements with the same first digit.

Bucket	**Contents**
0	310 130 330 120
1	301 201 111 231
2	222 032 002 102
3	213 023 013 323

Emptying the buckets gives us a new list:

[310, 130, 330, 120, 301, 201, 111, 231, 222, 032, 002, 102, 213, 023, 013, 323].

Radix sort

Using the new list, we collect elements with the same second digit.

Bucket	**Contents**
0	301 201 002 201
1	310 111 213 013
2	120 222 023 323
3	130 330 231 032

Radix sort

Using the new list, we collect elements with the same second digit.

Bucket	**Contents**
0	301 201 002 201
1	310 111 213 013
2	120 222 023 323
3	130 330 231 032

Emptying the buckets again:

[301, 201, 002, 201, 310, 111, 213, 013, 120, 222, 023, 323, 130, 330, 231, 032].

Radix sort

Finally, we collect elements with the same third digit.

Bucket	**Contents**
0	002 013 023 032
1	102 111 120 130
2	201 213 222 231
3	301 310 323 330

This time, emptying the buckets will give us a sorted list.

Radix sort

Radix sort has the air of a card trick about it, but it actually corresponds to how people sort things (such as socks: `http://tx0.org/5c3`) in real life.

Sticking to computing, we can use radix sort on data with other kinds of keys too, such as strings (using 26 buckets or 52 for a case-sensitive sort).

Radix sort

Generally, we need as many buckets as the number base or radix of the input. We need to inspect each element k times, where k is the number of digits in the biggest element.

k will be relatively small compared to n (e.g. when $k = 6$, we could have almost a million unique records). So the steps required is in the order $O(kn)$, or just $O(n)$.

Radix sort

The algorithm can be stated very elegantly:

```
procedure RADIXSORT(list, n)        ▷ Sort list having n elements with base 10.
    shift ← 1
    for loop = 1 to keySize do
        for entry = 1 to n do
            bucketNum ← (list[entry]/shift) % 10
            append(buckets[bucketNum], list[entry])
        end for
        list ← combineBuckets()
        shift ← shift * 10
    end for
end procedure
```

Radix sort

Since radix sort works in linear time, why do we even bother with other algorithms?

The catch is in the memory usage. This depends on how we implement the algorithm. If *buckets* is a 2D array, each element has to be as big as the original list, because the whole list might end up in the same bucket.

Thus, we need Rn additional storage, where R is the radix (what could we do to reduce memory usage?). Also, each element will be moved $2k$ times.

Next time

More basic data structures: stacks, queues and priority queues.