Radix Sort (Implementation)

Here, I'll discuss the code of the radix sort algorithm. (Reading time: 4 minutes)

Radix sort uses both counting sort and bucket sort. To implement radix sort, we need to have a `radixSort` function that receives the array we want to sort.

```
function radixSort(array) {
}
```

Right now, we need to store the largest digit of the maximum number in the given array, initialize a digit bucket list where we store the values, and the current index.

```
function radixSort(array) {
  const max = Math.max(...array).toString().length;
  let digitBuckets = [];
  let index = 0;
}
```

Next, we want to initialize a bucket for every digit that's possible. Let's say that we have the array `[8, 23, 12223, 901, 2990, 12]` that we want to sort. Now, **max** would be equal to **5**, as the length of the maximum `12223` value is **5**.

```
for (let i = 0; i < max + 1; i++) {
  digitBuckets = [];
}
```

Now, we want to loop over the numbers, and put the numbers in the buckets they belong, considering the currently active digit! While doing so, we also need to create a function that lets us know what the currently active digit is.

```
for (let j = 0; j < list.length; j++) {
  const digit = getDigit(list[j], i + 1);
}
```

```
function getDigit(num, nth) {
  let value = 0;
  while (nth--) {
    value = num % 10;
    num = Math.floor((num - value) / 10);
  }
  return value;
}
```

To the `getDigit` function, we pass the number `num` and the currently active significant value `nth`. Then, we initialize a default value of **0** as the number on the currently active digit: not all numbers have the same amount of significant numbers! Then, we have to calculate the value of the currently active digit, by `num % 10`. If the number is **1234**, the value would now be **4**. Then, we change the value of `num`, as we just checked one significant value. By subtracting the currently active digit, and then dividing it by **10**, we have our new value of `num`. If it was **1234** previously, `num` is now equal to **123**. This keeps on going, until `nth` returns `false`, when `nth` is **0**. Then, we return the value which is equal to the digit that we currently care about.

Now, we want to store every value in the same array, based on their value. For example, in the first round when we check the least significant digit, we would want the `digitBuckets` array to look like `[[10], [31], [22, 902], [3]]`.

The index of the array they should be pushed to, is equal to the `value` that got returned from the `getDigit` function! If that array doesn't exist yet, we have to initialize it first.

```
digitBuckets[digit] = digitBuckets[digit] || [];
digitBuckets[digit].push(list[j]);
```

Then, we want to loop over the `digitBuckets` array. Before doing so, we set the value of the `index` variable back to **0**.

```
idx = 0;
for (let t = 0; t < digitBuckets.length; t++) {
}
```

Now, we need to check whether the current array has a length. That's not always the case: the `digitBuckets` array could look like: `[[10], [], [53]]` always the case, the `digitBuckets` array could look like: `[[10], [], [32]]`.

Then, we want to loop over the separate arrays, and place the element in the correct position in the original array.

```
if (digitBuckets[t] && digitBuckets[t].length > 0) {
   for (let j = 0; j < digitBuckets[t].length; j++) {
      list[idx++] = digitBuckets[t][j];
   }
}
```

Right now, we've modified the original list, with the items in the correct order.

The entire `radixSort` function looks like:

```
function radixSort(array) {
  const max = Math.max(...array).toString().length;
  let digitBuckets = [];
  let index = 0;
  for (let i = 0; i < max + 1; i++) {
    digitBuckets = [];
    for (let j = 0; j < array.length; j++) {
       const digit = getDigit(array[j], i + 1);
       digitBuckets[digit] = digitBuckets[digit] || [];
       digitBuckets[digit].push(array[j]);
    }

    idx = 0;
    for (let t = 0; t < digitBuckets.length; t++) {
      if (digitBuckets[t] && digitBuckets[t].length > 0) {
        for (let m = 0; m < digitBuckets[t].length; m++) {
           array[idx++] = digitBuckets[t][m];
        }
      }
    }
  }
  return array;
}
```

Now, let's discuss the time complexity of this algorithm.