Chapter 6 : Part 7

**Sorting Algorithms: Radix Sort**

**6.7.1 Radix Sort**

* All sorting algorithms we seen in previous sections are "Comparison sort ". They make comparison between two elements or two indices.
* The Big-O of time complexity is between **O(n2)** and **O(nlog(n))**. The ***log*** is ***2-based***.
* Mathematically for all *Comparison-Sort* there are *Lower-Bound* and *Asymptote*. The best *average* case *Time complexity* is **O(nlog(n))**.
* However we can improve time complexity from **O(nlogn)**and this only works for specific kind of data. These sorting algorithms are not based on "**Comparison**".
* There is group of algorithm called "Integer Sorting algorithms". One of these algorithm is "**Radix-Sort**". We never make direct comparison between elements/indices.
* Radix-Sort is a "Non-Comparison Algorithm" generally it works with ***Integer data types*** and sort the ***integers*** according to their *length*.

|  |  |
| --- | --- |
|  |  |

**6.7.2 How does it works**

* Radix sort is a special sorting algorithm that works on lists of numbers.
* It also works with binary, if we convert images or strings into binary data we can sort them using this Radix Sort.
* It *never makes comparisons* between elements!
* It exploits the fact that information about the *size of a number* is encoded in the *number of digits*.
* *More digits* means a *bigger* number!

1. We create 10 buckets (for decimal numbers). We start from right-most digits. Eg: right most digit of the first number ***1556*** is ***6***. Similarly for other elements: 4 for 4; 6 for 3556; 3 for 593; and 8, 6, 2, 7, 7, 6, 7, 9 for other numbers.
2. Then we group them on the bucket based on these numbers.

|  |  |
| --- | --- |
|  |  |

1. Notice how the numbers in the *bucket* are *stacked*. This is one point where numbers are getting sorted. The numbers comes first are at the bottom.
2. Another point where numbers are getting sorted when we rearrange back into the list we keep the order as lower-digit to upper-digit.

* For example we put 902 at the first position.
* For the stacked numbers in the bucket we put the numbers from the bottom of the stack (works as queue: FIFO).



* Notice the numbers with last digit of 2 come before the number with last digit of 3.

1. Now we do the same process for the next digit (i.e. 2nd digit from the right).

|  |  |
| --- | --- |
|  |  |

* Notice the numbers are now stacked according to 2nd digit from the right.
* Note that **0** is considered for 2nd digit from the right of single digits numbers (or the numbers those have less digits than others). Are placed to **0**'s bucket.



1. For the 3rd digit from the right, we get

|  |  |
| --- | --- |
|  |  |

* Rearranging the list.



* Note that if we have *4 digit numbers* we have to split the array (putting numbers into the buckets) *4-times*. I.e. if we have *n digit* numbers we need to split the array *n-times*.

1. For the last time we split the array (i.e. the 4th digit from the right, we get):

|  |  |
| --- | --- |
|  |  |

1. Finally our list is sorted:

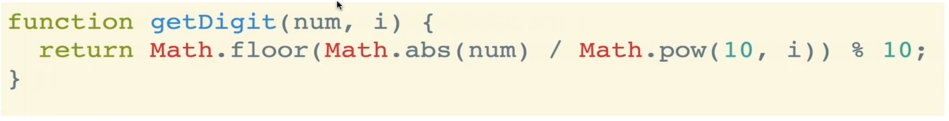


* The number of buckets: It depends on the ***base*** of the ***numbers***. Eg: for ***Octal*** numbers we have ***8 buckets***.
* Helper function: To get the digits of the numbers we need to create our own function for that (because some languages don't have built in function for that).

**6.7.3 Radix Sort Helpers**

We need to implement three helper functions for the Radix-Sort.

* **getDigit(num, place)**: Returns the digit in num at the given place value.



**function** **getDigit**(num, i) {

**return** Math**.floor**(Math**.abs**(num) / Math**.pow**(10, i)) % 10;

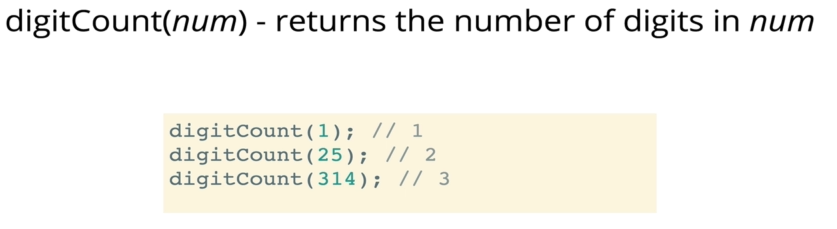
}

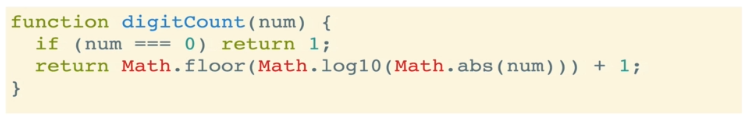
* Here **abs()** is used if we want to apply –ve numbers.
* To get first digit, we have to set i=0.
* For example to get 3rd right most digit of 12345:

1. 103-1=100;
2. 12345/100 = 123.45;
3. floor(123.45)=123;
4. 123 mod 10 = 3

i.e 3rd right most digit of 12345is 3.

* Bound case (No. of Digits) : To implement next Helpers, we need to figure out, *how many digits are present* in a number (i.e highest no. of digits.). It helps us to set the total no. of split of the list/array and reorder them.
* That’s our bound, if we have one 10 digit number, then we have to split the array 10 times.
* We have to figure out the number that has highest no. of digits.
* There are more two helper functions to write: **digitCount(),mostDigits()**
* **digitCount()**: It is our second helper function. It counts the digits of a single number.





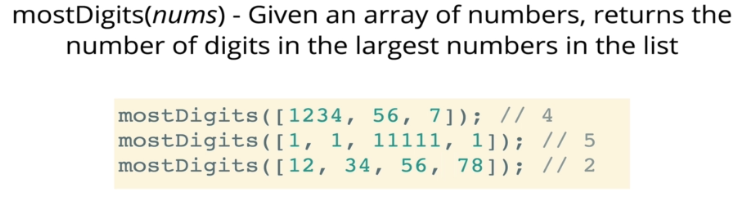
**function** **digitCount**(num) {

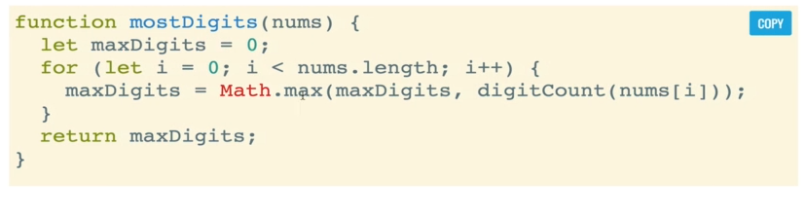
**if** (num **===** 0) **return** 1;

**return** Math**.floor**(Math**.log10**(Math**.abs**(num))) + 1;

}

* **abs()** is used for –ve numbers. **log10()** returns the 10th power we add 1 for the last digit (Eg: **log10(12)= 1.07; floor(1.07)=1,** but 12 have 2 digits, for **100** we have to add 1).
* **if** (num **===** 0) **return** 1 is used for the special case, because **log10(0)= infinity.**
* **mostDigits():**The third helper function **mostDigits()**. We'll use **digitCount()** to figure out the number that has highest no. of digits from the entire array.





**function** **mostDigits**(nums) {

**let** maxDigits=0;

**for** (**let** i=0; i **<** nums**.**length; i++) {

    maxDigits = Math**.max**(maxDigits, **digitCount**(nums[i]));

  }

**return** maxDigits;

}

**mostDigits**([23,567,89,12234324,90])

* nums represents each array of numbers.

|  |  |
| --- | --- |
| * Math**.max**() Compares the two numbers and returns the maximum. |  |

**6.7.4 Pseudocode (part 2) – Radix sort**

We'll use the previous helper functions to build radix-sort. We'll also create the buckets.

|  |
| --- |
| **RADIX SORT PSEUDOCODE**   1. Define a function that accepts *list* of *numbers* 2. Figure out how many *digits* the *largest number* has 3. Loop from **k = 0** up to this **largest** **number of digits** 4. For each iteration of the loop:    1. Create buckets for each digit (0 to 9): These 10 buckets are actually an array of 10 sub-arrays.    2. place each number in the corresponding bucket based on its ***k-th digit*** 5. Replace our existing array with values in our buckets, starting with 0 and going up to 9. We can just *concatenate* the *buckets/sub-arrays*. 6. ***return*** list at the end! |

**All helper codes at once**

**function** **getDigit**(num, i) {

**return** Math**.floor**(Math**.abs**(num) / Math**.pow**(10, i)) % 10;

}

**function** **digitCount**(num) {

**if** (num **===** 0) **return** 1;

**return** Math**.floor**(Math**.log10**(Math**.abs**(num))) + 1;

}

**function** **mostDigits**(nums) {

**let** maxDigits=0;

**for** (**let** i=0; i **<** nums**.**length; i++) {

    maxDigits = Math**.max**(maxDigits, **digitCount**(nums[i]));

  }

**return** maxDigits;

}

**mostDigits**([23,567,89,12234324,90])

Rough code

|  |  |
| --- | --- |
| **function** **radixSort**(arr){  **let** maxDigitsCount= **mostDigits**(arr);  **for**(**let** k=0; k **<** maxDigitsCount; k++ ) {  **let** digitBukets=Array**.from**({length:10},() **=>** []); /\* *creates the array of 10 sub-arrays* \*/  **for** (**let** i=0; i **<** arr**.**length; i++){        digitBukets[**getDigit**(arr[i], k)]**.push**(arr[i]);      }      console**.log**(digitBukets);    }    }  **radixSort**([23,567,89,12234324,90]) | [ [ 90 ], [], [], [ 23 ], [ 12234324 ], [], [], [ 567 ], [], [ 89 ] ]  [ [], [], [ 23, 12234324 ], [], [], [], [ 567 ], [], [ 89 ], [ 90 ] ]  [ [ 23, 89, 90 ], [], [], [ 12234324 ], [], [ 567 ], [], [], [], [] ]  [ [ 23, 567, 89, 90 ], [], [], [], [ 12234324 ], [], [], [], [], [] ]  [ [ 23, 567, 89, 90 ], [], [], [ 12234324 ], [], [], [], [], [], [] ]  [ [ 23, 567, 89, 90 ], [], [ 12234324 ], [], [], [], [], [], [], [] ]  [ [ 23, 567, 89, 90 ], [], [ 12234324 ], [], [], [], [], [], [], [] ]  [ [ 23, 567, 89, 90 ], [ 12234324 ], [], [], [], [], [], [], [], [] ] |

**radixSort**([23,345,5467,12,2345,9852])

[

  [],

  [],

  [ 12, 9852 ],

  [ 23 ],

  [],

  [ 345, 2345 ],

  [],

  [ 5467 ],

  [],

  []

]

[

  [],

  [ 12 ],

  [ 23 ],

  [],

  [ 345, 2345 ],

  [ 9852 ],

  [ 5467 ],

  [],

  [],

  []

]

[

  [ 23, 12 ],

  [],

  [],

  [ 345, 2345 ],

  [ 5467 ],

  [],

  [],

  [],

  [ 9852 ],

  []

]

[

  [ 23, 345, 12 ],

  [],

  [ 2345 ],

  [],

  [],

  [ 5467 ],

  [],

  [],

  [],

  [ 9852 ]

]

* Notice how numbers are stored in the buckets. We didn’t concatenate the arrays yet.

/\* *Radix sort main function* \*/

**function** **radixSort**(arr){

**let** maxDigitCount= **mostDigits**(arr);

**for**(**let** k=0; k **<** maxDigitCount; k++){

**let** digitBuckets=Array**.from**({length:10},() **=>** []); /\* *creates the array of 10 sub-arrays* \*/

**for**(**let** i=0; i **<** arr**.**length; i++){

**let** digit= **getDigit**(arr[i],k);

          digitBuckets[digit]**.push**(arr[i]); /\* *putting in the corresponding bucket* \*/

      }

      console**.log**(digitBuckets);

      // *We could use For-loop and push the subarrays instead of concat()*

      arr = []**.concat**(...digitBuckets); /\* *concatenating the sub-arrays* \*/

      // *[]. concat ( [ [1], [2], [3] ]) /\* returns only array of array \*/*

      // *[]. concat (... [ [1], [2], [3] ])  /\* returns a single-array only concatenatig elements of subarrays \*/*

      console**.log**(arr);

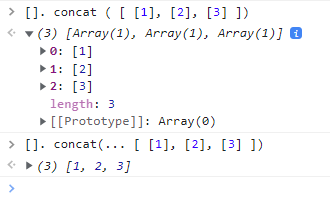
  }

**return** arr;

}

console**.log**("applying radix sort : "+**radixSort**([23,345,5467,12,2345,9852]));

* Why spread operator "...":



***All code at once (instructors version)***

**function** **getDigit**(num, i) {

**return** Math**.floor**(Math**.abs**(num) / Math**.pow**(10, i)) % 10;

}

**function** **digitCount**(num) {

**if** (num **===** 0) **return** 1;

**return** Math**.floor**(Math**.log10**(Math**.abs**(num))) + 1;

}

**function** **mostDigits**(nums) {

**let** maxDigits=0;

**for** (**let** i=0; i **<** nums**.**length; i++) {

    maxDigits = Math**.max**(maxDigits, **digitCount**(nums[i]));

  }

**return** maxDigits;

}

**function** **radixSort**(nums){

**let** maxDigitCount= **mostDigits**(nums);

**for**(**let** k=0; k **<** maxDigitCount; k++){

**let** digitBuckets=Array**.from**({length:10},() **=>** []);

**for**(**let** i=0; i **<** nums**.**length; i++){

**let** digit= **getDigit**(nums[i],k);

            digitBuckets[digit]**.push**(nums[i]);

        }

        nums = []**.concat**(...digitBuckets);

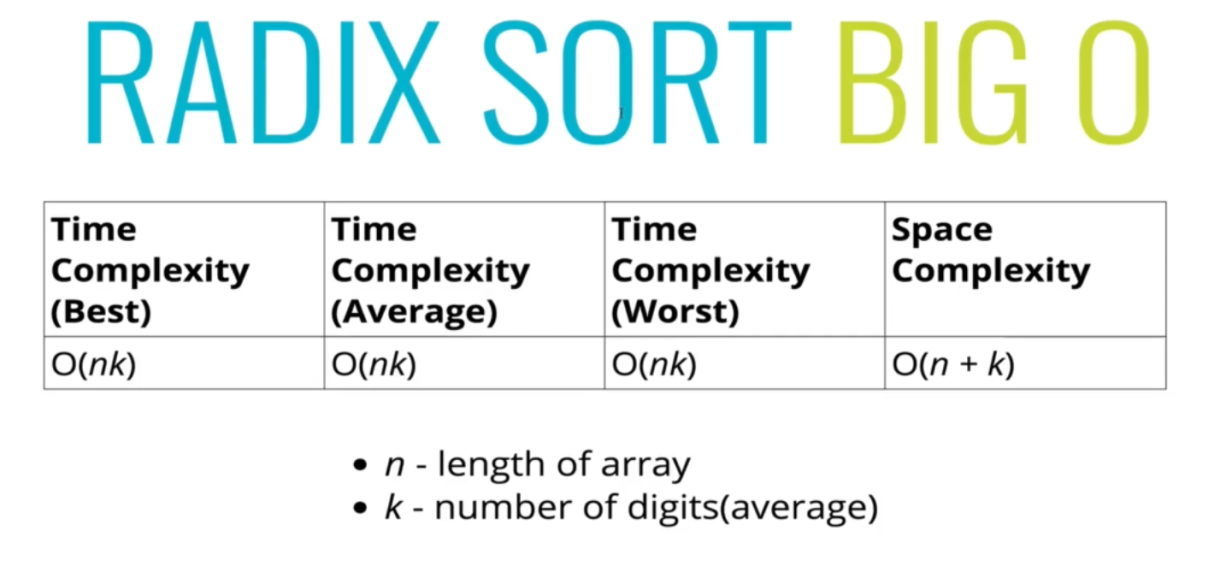
    }

**return** nums;

}

**radixSort**([23,345,5467,12,2345,9852])

**6.7.5 Big-O for Radix sort**



* Note:
* If **k** is too small than **n**, then it is the best case.
* In average case ***O(nk)*** can be equal to **O(n log n)**.
* In worst ***O(nk)*** case can be ***O(n2)*** or worse where the ***digits*** of a number (word length) is ***larger*** than ***n*** array-length.
* Efficiency :
* Radix sort complexity is ***O(nk)*** for ***n keys*** which are integers of ***word size k***.
* Sometimes ***k*** is presented as a ***constant***, which would make ***radix sort better*** (for sufficiently ***large n***) than the best comparison-based sorting algorithms (***Merge-sort***, ***Quick-sort***), which all perform **O(n log n)** *comparisons to sort* ***n*** ***keys***.
* However, in general ***k*** cannot be considered a constant: if all ***n*** ***keys*** are distinct, then ***k*** has to be at least **log(n)** for a *random-access machine to be able to store them in memory*, which gives at best a time complexity **O(n log n)** . That would seem to make radix sort at most equally efficient as the best comparison-based sorts (and worse if keys are much longer than **log n**).