Introduction to Non-Comparative Sorting and Parallel Computing Through Radix Sort

Soumik Ghosh Moulic

In today’s world, sorting algorithms can be classified into two main types - Comparison Based like QuickSort or MergeSort and Non-Comparison Based, like Counting Sort or Radix Sort. The fundamental difference between these two types of sorting algorithms is the need for a comparator vs no need of a comparator. A comparator basically defines the ordering of numbers or items e.g. numerical order.

Non-Comparison based sorting algorithms don’t use comparison by realy on integer arithmetic on keys. The popularity of Non- Comparison based algorithms vs. Comparison based algorithms arises from the fact that these non-comparison algorithms can work in O(n) time to sort elements vs the O(N log N) for the comparison-based sorting types. That is why non-comparison sorting algorithms are also called Linear Time Sorting Algorithms.

Bubble Sort, Selection Sort, and Insertion Sort all work around the average time complexity of O(N^2) (where N is the number of elements). Heapsort, MergeSort, and QuickSort take an average of O(N log N) (with QuickSort having O(N^2) worst case). So the questions arises, can we do better than O(N log N) for these algorithms? Turns out that it can be mathematically proven that any comparison based sorting algorithm will take at least Ω (N log N) operations to sort N elements in its worst case.

To get the proof, we must first recall what the comparison based sorting algorithm would take as an input and what will be it’s output.
Input : A sequence of N numbers <a_1, a_2, a_3,....,a_n>
Output: A permutation/reordering of the input such that $a_1 <= a_2 <= a_3 \ldots <= a_n$
The key to proving this is the argument that there are N! different possible permutations that the algorithm can output and for each of these permutations, there exists only one input for which that corresponding permutation is the correct answer. For Example: The permutation [a_3,a_1,a_4,a_2] is the only correct answer for sorting the input [3,1,4,2].

So we can fix some set of N! inputs, for each of the N! output permutations, and have 1-1 correspondence.

Let S denote the set of these inputs such that |S| = N! We can think of the comparison as splitting S into two groups: inputs for which the answer would be YES, and inputs for which the answer would be NO. Suppose that we always get the answer for each comparison corresponding to the larger group, then each comparison cuts down the size of S by a factor of 2. Since S has the size N! And the algorithm must reduce it to 1 to give an output, at least $\log_2(N!)$ comparisons need to be made. So now we can solve

$$log_2(N!) = log_2(N) + log_2(N-1) + log_2(N-2) + ... + log_2(2)$$

$$log_2(N!) = \Omega(N\ log\ N)$$

Another simpler way to prove it might be through decision tree.

1. Each of the N! permutations on N elements must be a leaf of the decision tree for the sorting algorithm to work properly.
2. If we let C be the maximum number of comparisons, then the max height of the tree would be C. Then the maximum number of leaves will be 2^C. Then

   $$N!\ <=\ 2^C$$

   Take log on both sides

   $$log_2(N!)\ <=\ C$$

   Since $log_2(N!) = \Omega(N\ log\ N)$

   $$C = \Omega(N\ log\ N)$$

One of the popular non-comparative sorting algorithms is the Radix Sort.

In mathematics, Radix represents how many digits are used in a number system. Ones, Tens, Hundreds, Thousands, all of these are radix. For example, the number 1000 has a radix of 4, the binary system has a radix of 2.

They way radix sort works is that it groups the number by their radix (individual digits), uses each radix as a key to sort with a sorting algorithm like counting sort or bucket sort.

Generally, radix sorts are of two types, Most Significant Digit Radix Sort (MSD Radix Sort) and Least Significant Digit Radix Sort (LSD Radix Sort)

MSD Radix Sort is generally used to sort the input in lexicographic order(alphabetic order). As the name suggests, MSD radix sort works by sorting from the most significant digit, or from the left side of a key(integer or a string). The MSD Radix Sort Stops rearranging the position of a key when it encounters a unique prefix of the key. For a high level example, let us take the keys to be Sterne and Soumik. Now a MSD Radix Sort would start from the Most Significant Digit or the left-most side. So
Step 1 - Sterne and Soumik - (both are the same alphabets, so move to the next one)
Step 2 - Sterne and Soumik - (o comes before t alphabetically, so it will rearrange accordingly)
Step 3 - Soumik and Sterne (Arranged the order alphabetically, and provides the output)
This is a high level example of a MSD Radix Sort. Actual implementations use something called buckets to group elements with the same alphabet or digit together, and then use recursion to sort each bucket before concatenating them together. For example, if we take the names Sterne, Soumik, Fabian, and Josh as our input, then a recursive MSD Radix Sort would do something like this:
Step 1: Starting the sort from the leftmost alphabet, it will create 3 buckets, the F bucket, the J bucket and the S bucket.

Step 2: Since only the S bucket has more than one item in it, it will have to create two more buckets based on sorting the second alphabet from the left. So it will create a “o” bucket and a “t” bucket.

Step 3: Since all the buckets have no more than one item, so now the buckets start concatenating. First the sorted “o” and “t” bucket get concatenated with the S bucket and then the S bucket gets concatenated with the sorted F and J bucket to give the output Fabian, Josh, Soumik, Sterne.

In [1]:

```
def radix_MSD_sort(array, index):
    #base case - when list is already sorted
    if len(array) <= 1:
        return array

    # division done first by length, then by lexicographical order of the first character
    completed_buckets = []
    new_buckets = [ [] for k in range(26) ]
    #I only chose 26 buckets because I am sorting in lowercase, if I had to also
    #account for uppercase characters, I would have to use 52 buckets.

    for j in array:
        if index >= len(j):
            completed_buckets.append(index)
        else:
            #this is where we add elements to the buckets alphabetacilly by identifying
            #them with their unicode.
            new_buckets[ ord(j[index]) - ord('a') ].append(j)

    #recursive calling
    new_buckets = [ radix_MSD_sort(b, index + 1) for b in new_buckets ]

    #joining all buckets together
    return completed_buckets + [ bucket for bucketlist in new_buckets for bucket in bucke

def main():
    names = ['professor', 'kalia', 'josh', 'vu', 'barbara', 'seth', 'soumik', 'namrata',
             'gili', 'ning', 'frank', 'nicole', 'fabian', 'jacob', 'marlette', 'nikesh',
             'gelana', 'olaf']
    print names
    names = radix_MSD_sort(names, 0)
    print names

if __name__ == '__main__':
    main()
```

```
['professor', 'kalia', 'josh', 'vu', 'barbara', 'seth', 'soumik', 'namrata',
'gili', 'ning', 'frank', 'nicole', 'fabian', 'jacob', 'marlette', 'nikesh',
'gelana', 'olaf']
['barbara', 'fabian', 'frank', 'gelana', 'gili', 'jacob', 'josh', 'kalia',
'marlette', 'namrata', 'nicole', 'nikesh', 'ning', 'olaf', 'professor', 'set
h', 'soumik', 'vu']
```

As you can see from the above code, I applied the MSD Radix Sort Logic to implement a name sorter that sorts the names in alphabetical order. This algorithm can be used on datasets where you need alphabetical sorting, like putting words into a dictionary or creating a yellow page phone book.

Another implementation is called the Least Significant Digit Radix Sort or the LSD Radix Sort. It is a fast stable sorting algorithm (meaning that it puts elements of a list in certain order) that can be used to sort keys (integer or strings). The logic behind the LSD Radix Sort is the opposite of MSD Radix Sort. LSD Radix Sort works by starting from the Least Significant Digit, or the rightmost digit, and works its way to the Most Significant Digit, or the leftmost digit. For example, if we have the input of 324,56,66,1 then an LSD Radix Sort would work like this

Step 1: Sorting by the ones place, it will compare 324, 56, 66, 1 and give 1, 324, 56, 66. (One important thing to notice here is that LSD Radix sort keeps the original order of the keys while performing the sort. Since 56, came before 66 in the original list, so it will preserve the same order while grouping them together. This is what makes LSD Radix Sort a stable sort.)
Step 2: Sorting by the tens place, it will compare 01, 324 ,56, 66 and give 01, 324, 56, 66. (We can notice that a zero is “generated” in front of the number 1 to make the digits comparable).
Step 3: Sorting by the hundreds place, it will compare 001, 324, 056, 066 and give 001, 056, 066, 324. The list is now sorted.

As stated before, radix sort has an average time complexity of O(n) we O(n log n) for a comparative based sorting algorithm like quicksort. But in the real world, how fast does this two compare in the real world? For this purpose, I decided to test out LSD Radix Sort vs Quicksort against different data sizes from one thousand to ten million.

In [2]:

```
#base code taken from wikipedia and modified and commented.
def radix_LSD_sort(array, base=7):
    def list_to_buckets(array, base, iteration):
        buckets = [[] for _ in range(base)]
        for number in array:
            # Isolating the base digit from the number (Following LSD)
            digit = (number // (base ** iteration)) % base
            # Droping the number into the correct bucket
            buckets[digit].append(number)
        return buckets

    def buckets_to_list(buckets):
        numbers = []
        for bucket in buckets:
            # appending the numbers in a bucket
            #sequentially to the returned array
            for number in bucket:
                numbers.append(number)
        return numbers

    maxval = max(array)

    num_iterations = 0
    # Iterating, sorting the array by each base-digit by creating buckets and sorting
    #those bukets out
    while base ** num_iterations <= maxval:
        array = buckets_to_list(list_to_buckets(array, base, num_iterations))
        num_iterations += 1

    return array

M = [66336,45,785,1,23,56]

radix_LSD_sort(M)
```

Out[2]:

```
[1, 23, 45, 56, 785, 66336]
```

In [3]:

```
from random import randint
def partition(a, p, r):
    x = a[r]
    i = p-1
    for j in range(p, r):
        if a[j] <= x:
            i = i + 1
            a[i],a[j] = a[j],a[i]
    a[i+1],a[r] = a[r],a[i+1]
    return i+1

def quicksort(a, p=0, r=None):
    if r is None:
        r = len(a)-1

    if p < r:
        q = partition(a, p, r)
        quicksort(a, p, q-1)
        quicksort(a, q+1, r)

a = [8,9,6,3,2,4,5,9,6,3,7,45,2,-9]
quicksort(a)
print a
```

```
[-9, 2, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 9, 45]
```

In [4]:

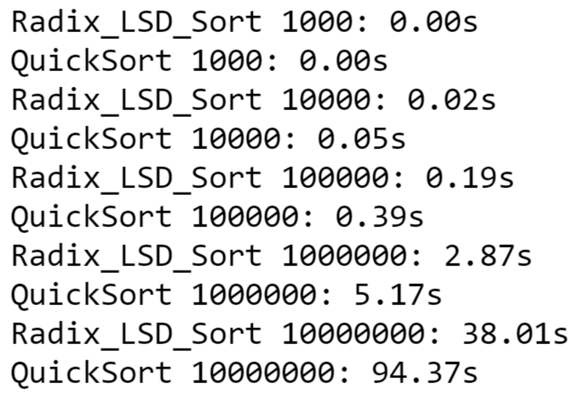
```
from time import clock
from random import randint
if __name__=='__main__':
    for value in 1000, 10000, 100000, 1000000, 10000000:
        list_1 = [randint (1, value) for _ in range(value)]
        list_2 = list(list_1)

        start = clock()
        list_1 = radix_LSD_sort(list_1)
        end = clock()
        print("Radix_LSD_Sort %d: %0.2fs" % (value, end-start))

        start = clock()
        list_2 = quicksort(list_2)
        end = clock()
        print("QuickSort %d: %0.2fs" % (value, end-start))
```

```
Radix_LSD_Sort 1000: 0.00s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.03s
QuickSort 10000: 0.03s
Radix_LSD_Sort 100000: 0.24s
QuickSort 100000: 0.42s
Radix_LSD_Sort 1000000: 4.11s
QuickSort 1000000: 5.12s
Radix_LSD_Sort 10000000: 49.83s
QuickSort 10000000: 90.33s
```

As you can see, with increasing amount of n, quicksort starts taking more and more time than Radix Sort to sort things out. At n = 1 million onwards, quicksort ends up taking significantly more time than LSD Radix Sort. At 10 million integers, Quicksort takes almost twice as much time as LSD Radix Sort does.

However, I discovered that LSD Radix Sort could go even faster than Quicksort if its bucket size was increased. This is because more buckets means more memory allocated and thus faster computations.

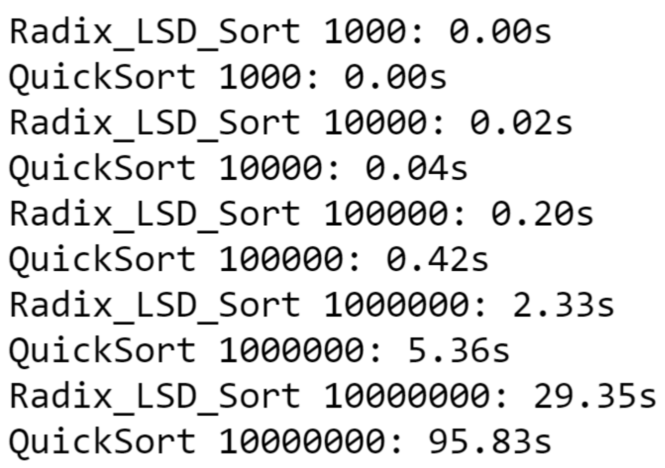
LSD Radix Sort vs Quicksort when bucket size of LSD Radix Sort is doubled (14 from 7). As you can see, the sorting time for LSD Radix Sort decreased by 1.24 seconds for 1 million numbers and by 11.82 seconds for 10 million operations when the bucket size was doubled.



```
Radix_LSD_Sort 1000: 0.00s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.02s
QuickSort 10000: 0.05s
Radix_LSD_Sort 100000: 0.19s
QuickSort 100000: 0.39s
Radix_LSD_Sort 1000000: 2.87s
QuickSort 1000000: 5.17s
Radix_LSD_Sort 10000000: 38.01s
QuickSort 10000000: 94.37s
```

Image: Timings of QuickSort vs LSD Radix Sort with bucket size of 14

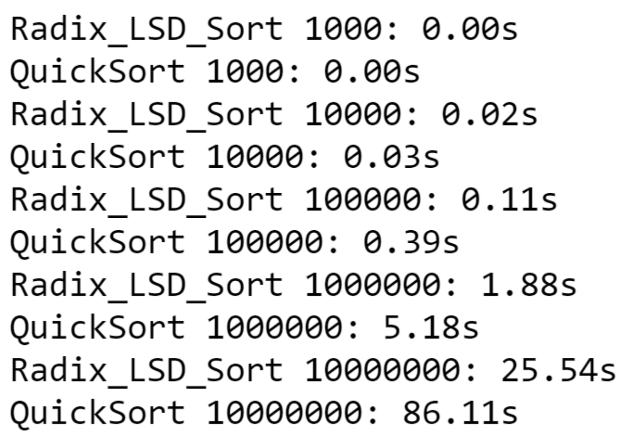
Again we double the bucket size (28 from 14) to see what happens. This new bucket size is four times the original bucket size. We can see that for 1 million operations, the sorting time for LSD Radix Sort decreased by 0.54 seconds while for 10 million operations, the sorting time for LSD Radix Sort decreased by 8.66 seconds.



```
Radix_LSD_Sort 1000: 0.00s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.02s
QuickSort 10000: 0.04s
Radix_LSD_Sort 100000: 0.20s
QuickSort 100000: 0.42s
Radix_LSD_Sort 1000000: 2.33s
QuickSort 1000000: 5.36s
Radix_LSD_Sort 10000000: 29.35s
QuickSort 10000000: 95.83s
```

Image: Timings of QuickSort vs LSD Radix Sort with bucket size of 28

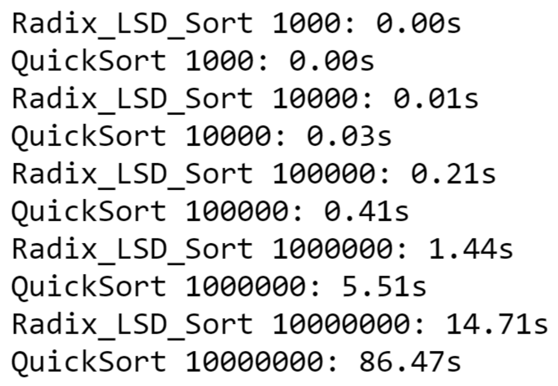
Doubling the bucket size again (56 from 28) we try to observe the changes in time taken to sort by LSD Radix Sort. We can see that for 1 million operations, the sorting time for LSD Radix Sort decreased by 0.45 seconds while for 10 million operations, the sorting time for LSD Radix Sort decreased by 3.81 seconds. It is also important to note that the time of quicksort also decreased by 9.72 seconds, indicating that this might have been a fairly sorted list to begin with (we are using random lists of N elements, so sometimes, sorted sublists within the list can be generated as well, which improves the QuickSort time.)



```
Radix_LSD_Sort 1000: 0.00s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.02s
QuickSort 10000: 0.03s
Radix_LSD_Sort 100000: 0.11s
QuickSort 100000: 0.39s
Radix_LSD_Sort 1000000: 1.88s
QuickSort 1000000: 5.18s
Radix_LSD_Sort 10000000: 25.54s
QuickSort 10000000: 86.11s
```

Image: Timings of QuickSort vs LSD Radix Sort with bucket size of 56

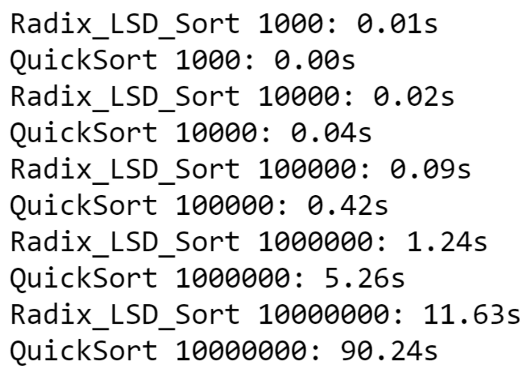
I observed that just doubling the bucket size was not having as much of a significant impact on decreasing the runtime of LSD Radix Sort as much as it was doing before. So I decided to increase the bucket size to 350 to observe any changes. I assume that 350 bucket size would be a significant jumping point for LSD Radix Sort to work. For this large bucket size, the runtime of Radix Sort was reduced to 14.71 seconds for 10 Million elements. This is 10.83 seconds less than the previous runtime.



```
Radix_LSD_Sort 1000: 0.00s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.01s
QuickSort 10000: 0.03s
Radix_LSD_Sort 100000: 0.21s
QuickSort 100000: 0.41s
Radix_LSD_Sort 1000000: 1.44s
QuickSort 1000000: 5.51s
Radix_LSD_Sort 10000000: 14.71s
QuickSort 10000000: 86.47s
```

Image: Timings of QuickSort vs LSD Radix Sort with bucket size of 350

I just wanted to see how much faster I can push this LSD Radix Sort, so I decided to increase my bucket size to 10 thousand from 350 just to see how fast it could sort vs QuickSort. Surprisingly, such a huge increase in the bucket size only decreased the sort time for 10 Million elements by 3.08 seconds (11.63s as compared to 14.71s). So the sort time is approaching terminal speed and we would not be able to go significantly faster than this. Hence I decided to stop my analysis here.



```
Radix_LSD_Sort 1000: 0.01s
QuickSort 1000: 0.00s
Radix_LSD_Sort 10000: 0.02s
QuickSort 10000: 0.04s
Radix_LSD_Sort 100000: 0.09s
QuickSort 100000: 0.42s
Radix_LSD_Sort 1000000: 1.24s
QuickSort 1000000: 5.26s
Radix_LSD_Sort 10000000: 11.63s
QuickSort 10000000: 90.24s
```

Image: Timings of QuickSort vs LSD Radix Sort with bucket size of 10000

At the final run, LSD Radix Sort was 78.61 seconds faster than Quicksort for the same 10 million elements. That is a really significant time difference in the real world. So by proof, we see that Radix Sort, a non-comparison based sorting algorithm performs much faster than Quicksort, a comparison based sorting algorithm. But we should keep in mind that Radix Sort also took increasingly large space (bucket size) to achieve that fast processing.

Another useful application of Radix Sort is in Parallel Computing. Parallel Computing is a process in which many calculations are carried out simultaneously by multiple cores of a processor. In order for an algorithm or a data structure to utilize the power of parallel computing, it's way of storing or organizing data should be set in a particular way where no significant process is dependent on the other, yet at the end, all the results could be concatenated to provide an useful output.

Radix Sort has this property that lets it exploit the power of parallel computing. It implements a lock-free programming, that guarantees that some thread can complete an operation independently without waiting for the execution of other threads. This operation is the sorting of the bins/buckets. Each of the bins/buckets can be sorted independently without waiting for the other buckets to be sorted.

A single processor takes up a single bucket to sort, maybe the MSD or the LSD depending on the type of algorithm running, and then has to only sorth that bucket irrespective of whether the other buckets have been sorted or not. However, processing each bucket is data-driven, as it depends on how many buckets you can form. So random inputs would equally populate all the buckets and thus perform better parallelly than inputs that have the same value keys, because same value keys will only create a single bucket.

References

Allig, C. (2015, July 7). Parallel Data Structures. Retrieved December 14, 2017, from https://www.nst.ei.tum.de/fileadmin/w00bqs/www/publications/as/2015SS-HS-ParallelDataStructures.pdf

Amato, N., Iyer, R., Sundaresan, S., & Wu, Y. (1996, January). A comparison of parallel sorting algorithms on different architectures. Retrieved December 14, 2017, from http://parasol-www.cs.tamu.edu/dsmft/publications/psort-tr.pdf

Are sorting algorithms approaching linear time? (n.d.). Retrieved December 14, 2017, from https://cstheory.stackexchange.com/questions/11697/are-sorting-algorithms-approaching-linear-time

Bozidar, D., & Dobravec, T. (n.d.). Comparison of parallel sorting algorithms. Retrieved from https://arxiv.org/ftp/arxiv/papers/1511/1511.03404.pdf

Bucket and Sample Sort. (n.d.). Retrieved December 14, 2017, from http://parallelcomp.uw.hu/ch09lev1sec5.html

Bucket sort. (2017, December 13). Retrieved December 14, 2017, from https://en.wikipedia.org/wiki/Bucket_sort

Comparison-based Lower Bounds for Sorting . (n.d.). Retrieved December 14, 2017, from https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). *Introduction to algorithms*(3rd ed.).

Chapter 8 - Sorting in Linear Time

Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms? (n.d.). Retrieved December 14, 2017, from http://javarevisited.blogspot.kr/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html

Foster, I. (1995). Parallel Algorithm Examples. Retrieved December 14, 2017, from http://www.mcs.anl.gov/~itf/dbpp/text/node10.html#SECTION02240000000000000000

Herlihy, M., & Shavit, N. (2012). *The art of multiprocessor programming*. Burlington, MA: Morgan Kaufmann. doi:http://cs.ipm.ac.ir/asoc2016/Resources/Theartofmulticore.pdf

Non-Comparison Sorting Algorithms. (n.d.). Retrieved December 14, 2017, from http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html

Radix sort. (2017, December 12). Retrieved December 14, 2017, from https://en.wikipedia.org/wiki/Radix_sort

Samplesort. (2017, December 12). Retrieved December 14, 2017, from https://en.wikipedia.org/wiki/Samplesort

Silva, F. (n.d.). Parallel Algorithms - Sorting. Retrieved December 14, 2017, from http://www.dcc.fc.up.pt/~fds/aulas/PPD/1112/sorting.pdf

Skarupke, M. (2016, December 02). Investigating Radix Sort. Retrieved December 14, 2017, from https://probablydance.com/2016/12/02/investigating-radix-sort/

Sorting algorithm. (2017, December 09). Retrieved December 14, 2017, from https://en.wikipedia.org/wiki/Sorting_algorithm#Stability

Spiegel, M. (2013, April 15). The Secret Life of Concurrent Data Structures. Retrieved December 14, 2017, from https://www.addthis.com/blog/2013/04/25/the-secret-life-of-concurrent-data-structures/#.WjFXlkqWZPZ

What is stability in sorting algorithms and why is it important? (n.d.). Retrieved December 14, 2017, from https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important

Yuan, W. (n.d.). Fast Parallel Radix Sort Algorithm. Retrieved December 14, 2017, from http://projects.csail.mit.edu/wiki/pub/SuperTech/ParallelRadixSort/Fast_Parallel_Radix_Sort_Algorithm.pdf