PT sort: A non-comparison sort using the sum of the power of two

Yu-Cheng, Liu

Oakridge secondary school, London, Ontario, Canada

3359767@gmail.com

liuluke834@gotvdsb.ca

ABSTRACT

Sorting algorithm is one of the most important fields in computer science. People learn sort algorithms before learning other advanced algorithms. Since sorting is important, computer scientists study and try to create new sorting algorithms. The paper is composed of five themed sections: Introduction, Method, Analyzis, Experiment Result, and Conclusion. The first section of this paper will give a brief overview of the algorithm and introduce a new way to sort a list called PT sort, a non-comparison integer sorting algorithm that is based on subtracting the largest exponent with radix two, then using recursion and traverse on every separated list. Section two and three begins by laying out the theoretical dimensions of the research, proposes the methodology, and analyzes the time complexity of PT sort. The forth section presents the findings of the research, focusing on the result of the experiment. The time complexity and space complexity of PT sort is approximatly $O(n \cdot \log_2 r)$ where n is the number of the numbers being sorted and r is the largest number in the list.

CCS CONCEPTS: •Theory of computation~Design and analysis of algorithms~Data structures design and analysis~Sorting and searching•

Additional Keywords and Phrases: sorting algorithm, recursion, non-comparision, integer

1 INTRODUCTION

Algorithm is a defined computational procedure that takes values with input and output [1]. Algorithm is a major area of interest within the field of computer science. Algorithms play a significant role while solving computational problems. In other words, everything on the computer involved at least one algorithm. Sorting techniques are considerably basic among the algorithms because a sorting algorithm is just a method to arrange elements in order. However, this field has enormous potential for people. There are lots of sorting algorithms. Every sorting technique has a unique strategy. Different sorting techniques can be used in different situations. In this research, I propose a new sorting strategy, PT sort. This technique is based on subtracting the largest exponent with the base of 2, then using recursion and traverse every separated list. This section introduces several sort techniques. These sorting techniques are chosen for the connection with PT sort. The further comparison will display in analyzis.

1.1 REVIEW OF SOME EXISTING NON-COMPARISON SORT ALGORITHM AND MERGE SORT

A. Merge Sort

Merge Sort is the earliest sort that uses the divide and conquer algorithm that was invented by John von Neumann in 1945[2]. Wikipedia concludes its steps: “Divide the unsorted list into n sublists, each containing one

element (a list of one element is considered sorted). 2. Repeatedly merge sublists to produce new sorted sublists until only one sublist is remaining. This will be the sorted list" [3]. The average time complexity is O (n log n).

B. Pigeonhole Sort

Pigeonhole sorting can apply for sorting lists when n, number of elements, and *N*, the length of the maximum value of possible key values, are similar [4]. We seldom use pigeon sort when choosing a sorting algorithm because it rarely exceeds other sorting algorithms in versatility, integrity, and especially speed. The other one, Bucket sort, is more efficient than this one [5].

The working principle of the pigeon algorithm is as follows: 1. Given an array. Set up a support array as "Pigeonhole". Create pigeonholes for every key in the range of the array. 2. Traversal the array, put each value that corresponds to the key-value into the pigeonhole. Every pigeonhole contains a list of all values of the same key. 3. Traversal every value in each pigeonhole and put the elements in the original array [6].

C. Bucket Sort

Bucket sort split the list equally between every bucket. Then using another sorting algorithm to sort each bucket. It is a recapitulation of pigeonhole sort [7].

The bucket sorting works as follows: 1. Set an array that represents empty buckets. 2. Divide the original array and put the elements in the buckets. 3. Sort every non-empty bucket 4. Visit every bucket orderly and put the elements into the original list [7]. Bucket sort's average time complexity is O $(n + \frac{n^2}{k} + k)$. However, bucket sort sacrifices space to sort the array. Its worst-case space complexity is O(n · k) [7].

D. Counting Sort

Counting sort is an integer sorting algorithm. It is efficient when the difference between the maximum key and the minimum key in the list is small. Counting sort use key as indexes to sort arrays, so it is not a comparison sort. It is similar to bucket sort on the average time complexity for doing the same task but counting sort does not need a link list and dynamic array [8, 9, 10].

The counting sorting works as follows: 1. Create an array of *k*+1 zero, *k* is the maximum key of the array. 2. Go through the array and count every element. 3.Travalsal the new array and take out every element into a new list [8]. The time complexity of counting sort is O(n+k) and its worst-case space complexity is O(n+k).

E. Radix Sort

When the integer list has a large maximum key, radix sort is better than counting sort [1, 9, 10]. The origin of Radix sort can back to Herman Hollerith's work on a tabulating machine in 1887 [11]. Radix sort has two Specialized variants which are LSD (Least significant digital) and MSD (Most significant digital). Radix sort can use on data that is lexicographical. Radix sort is a non-comparison sorting algorithm.

The counting sorting works as follows: 1. Unified every element into the same digit length by padding zeros. 2. Sort numbers from the lowest digit to the highest digit. In this way, from the lowest order to the highest order, the sequence becomes an ordered sequence. The time complexity of radix sort is O(n · k), where *n* is the number of sorted elements and *k* is the number of digits [12].

2 PROPOSED METHOD

2.1 Basic Idea

In this section, I am explaining the main idea of PT sort. The basic idea of PT sort is simple. Every number can convert to a binary number; for instant, 41 in binary is 101001 which also $2^5 + 2^3 + 2^0$. We can also get 5 from the

integer part of $\log_2 41$. The next step is to assign the number to the new list *arr*. 41 will be assigned to the group which contains any numbers between 2^5 *and* 2^6(Figure 1). After assigned *n* numbers, PT sort applies recursion on every group. The easiest way to implement recursion is to subtract the numbers by 2^p. *P* is the largest power of 2 which smaller or equal to the element. In the first loop *P* is 0. In the following recursion, *P* is the index of the list. In this case, *P* is 5 since this group is for numbers between 2^5 *and* 2^6. Recursion stops when every element in the group is the same or there is only 0 or 1 in the group when *P*=0. Next, the algorithm starts to merge. Numbers add back what they have subtracted when merging. Elements are collected in list *S*. Return *S* when the merge is done.

arr=[[], [], [], [], [], [], [41]]

Figure 1: arr is the new list created in the PT sort. 41 is in the list of index 5 in arr

2.2 Algorithm

To start PT sort (*L, P*), we need the input list *L* which contains *n* numbers. The largest power of 2, *P*, is 0. *N* in the algorithm is 2^p. *C* is the current element of the loop. i is the index of the sublist. *S* is the sorted list that PT sort returns. arr_x is the new list which contains child lists. *S* is the list that collects elements in the arr_x where *x* is the recursion layer..

For the algorithm, there are two insert id for one and zero. PT sort work like the following steps:

1. Subtract every element *C* from the list by *N*, put the number into the different lists in arr. Every list has its range determined by its index number; for example, the list with index 2 contains the elements with a range is between 2^2 to 2^3.

2. By the end of the first loop, every element will be assigned to a list based on its value. Then, PT sort will loop every list in arr. PT sort will be applied on the list as recursion, and *P* is the index of the list subtract 1.

3. Do step one to step three until the list is empty or has only one distinct element. The numbers will add back what they have subtracted while merging and return list *S*.

The following section displays the pseudocode and the flowchart of PT sort in Python style.

2.3 Example

[45, 82, 37, 31, 88, 34, 17, 35, 87, 93]

Figure 3: The example list *L*

An example list is being sorted by PT sort in this section. Using random number generator, we get a list *L* with 10 numbers which is the value of *n.*

Steps

1. Create a new empty list arr_x which recursion layer, *x*, is 0. The largest power of 2 number, *P*, is 0 in the first layer.

[ ]

Figure 4: arr_0

2. Loop the list *L*. For every element *C,* the first element of the list is 45.

2-1. $\log_2$ C is being calculated and taken the integer part . In this step, $\log_2 45$ have the value 5.

2-2. Check if the arr_x has $\lfloor \log_2 C \rfloor$ sublists where $\lfloor \rfloor$ is the floor function, which gives the largest integer less than or equal to *x*. If it has, add *C* into the sublist with index 5. Otherwise create the sublists and add C into it. If the layer *x* is not 0, the numbers should substract 2^P before adding into the sublist.

[[], [], [], [], [], [45]]

Figure 5: arr_0 after processing the first element

After looping the whole L, the arr_0 should be this:

[[], [], [], [], [31, 17], [45, 37, 34, 35], [82, 88, 87, 93]]

Figure 6: arr_0 after the first loop

In the arr, each sublist contain numbers that the numbers are in the range of $2^i<C<2^{i+1}$ where *i* is the index of the sublist. In Figure 6, 31 and 17 (before substract) are in the sublist with index 4.

3. After processing all the elements in the L. PTsort uses recursion on each of the sublists. arr_0 is going to be L and arr_1 is created. If the sublist is empty, only has all same values, or only contains 0 and 1, the process of the current sublist is done. Otherwise, apply steps 1 and 2 to every sublist. Figure 7 is [31,17], with P=4, in arr_0. After applying step 1 and 2, it becomes arr_1 in Figure 7:

[[1], [], [], [15]]

Figure 7: arr_1 after the first loop

4-1. From the instruct in Figure 7, all the sublists in the arr_1 are done. Every element in the sublists form to a list S after add of the numbers that they are substrate, using $C+2^P$. After adding back. The elements in sublists of arr_1 are conbined andr return. The [31,17] in arr_0 is replaced by the new list [17,31] which is after processed.

[[], [], [], [], [17, 31], [45, 37, 34, 35], [82, 88, 87, 93]]

Figure 8: arr_0 after finish processing arr_1 of [31,17] in Figure 6

4-2. Then, PTsort is processing the next sublist in arr_0 which is [45,37,34,35]. Using the same stepts. A sorted sublist replaces the current sublist. This also apply on the next sublist.

5. Combine all elements in sublists of arr_0. They do not need to add numbers because arr_0 does not substract numbers.

[17, 31, 34, 35, 37, 45, 82, 87, 88, 93]

Figure 8: *The final return list S*

3 ANALYZIS

ALGORITHM: PSEUDOCODE

```
algorithm PTsort(L, P)
    input: List L with lowest power of 2 P
    output: Sorted list S
    Note: P←0 before the algorithm start

  arr←[[]]
  n←2^P
  for I in L do:
     if p!=0 do:
        I←I-n
     end if
     if I==0 do:
        arr[0].insert(I,0)
        continue
     end if
     if I==1 do:
        arr[0].append(I)
        continue
     end if
     else do:
        A←int(log2(C))
     end else
     while len(arr)<A+1 do:
        arr.append([])
     end while
     else do:
        arr[A].append(I)
     end else
  for id in range(1,len(arr)) do:
          if arr[id] do:
             arr[id] ←PTsort(arr[id], id)
          end if
  end for
  S ← []
  if p!=0 do:
     for k in arr do:
          S.extend(x+n for x in k)
     end for
  end if
```

```
    else do:
      for k in arr do:
        S.extend(x for x in k)
      end for
    end else
    return S
end PTsort
```

FLOWCHART: PTSORT

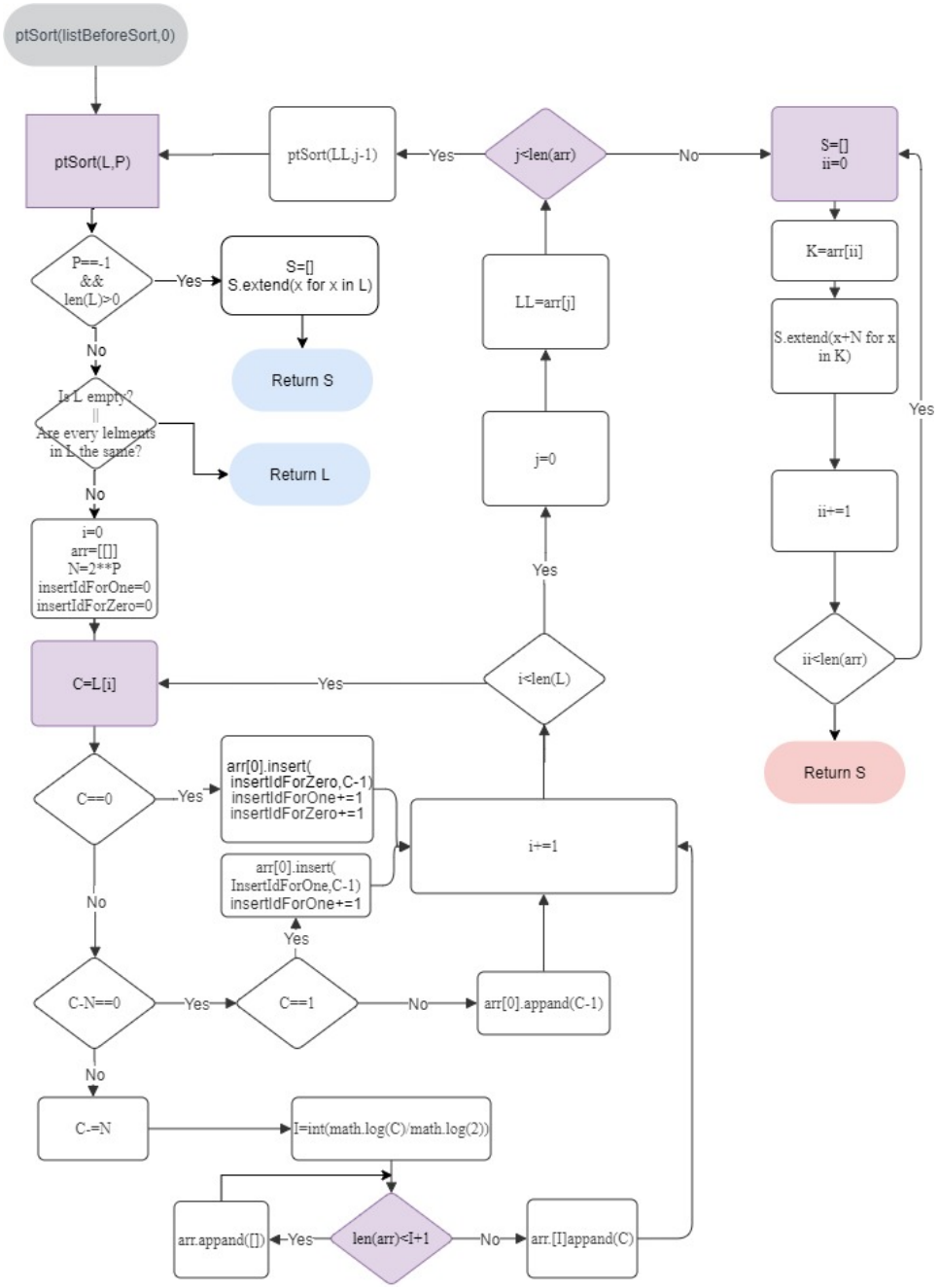

Figure 2: Flowchart of PT sort using python style

4 EXPERIMENT RESULT

In the first section of this part, PT sort compares to another sorting algorithm which appears in the Introduction. The following section shows the performance of PT sort. Section 3.2 shows the performance of PT sort. Graph comparing is an effective way to compare algorithms. However, there are certain drawbacks associated with the use of it. The main disadvantage is that the result depending on the code and other factors. The graph of the sort algorithm is for reference only. In section 3.3, the time and space complexity of PT sort is proposing.

4.1 COMPARE TO OTHER ALGORITHMS

A. Merge Sort

Merge sort has a great representative of the divide and conquer algorithm which is also applied on PT Sort. That is why we compare merge sort with PT sort through merge sort is the only comparison sort that appears in this paper. PT sort and Merge sort both use recursion to traverse the lists.

B. Pigeonhole Sort

Pigeonhole sort and PT sort are both non-comparison sort. Pigeonhole sort use support array so does PT sort. Pigeonhole sort does not change the same value's order, which means it is a stable sort technique. PT sort is also stable. Both sorting algorithms go linear time complexity when only the size of the list change.

C. Bucket Sort

Bucket sort and PT sort is similar but also different. Bucket sort use "buckets", and PT sort has support arrays. However, the range of the buckets are the same, but the range of support arrays depending on its index. Bucket sort, same as PT sort and other sorts, goes linear time complexity when only the size of the list change.

D. Counting Sort

The main similarity between counting sort and PT sort is when the largest value of the sort increase, time cost also increases. Counting sort is an integer sort, so does PT sort.

F. Radix Sort

The word "radix" has a huge connection for PT sort. Nevertheless, the principle between radix sort and PT sort is quite different. When the range of the list increase, digits also rise. Radix Sort is efficient with a large value key because the time cost decreases when the digit increases when PT sort has the opposite effect.

4.2 PERFORMANCE

This section displays the line graph which shows PT sort and other sorts' performance base on n, the list size, and r, the range of the list.

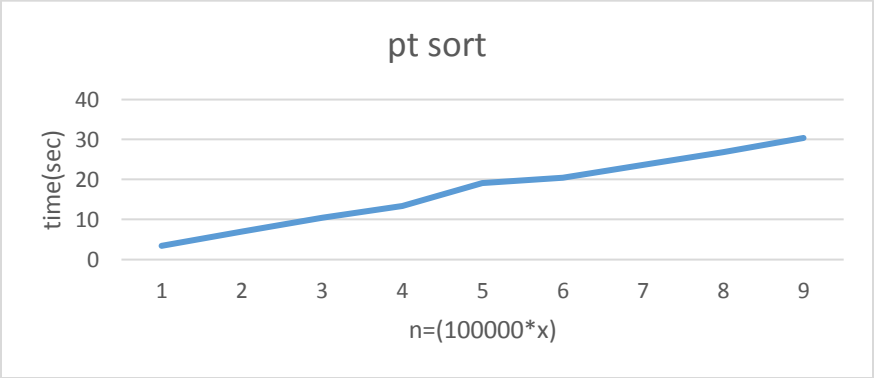

Figure 3: Time cost(second) of PT sort by *n* when *r*=1000. We can see the time cost of PT sort is linear when r is fixed.

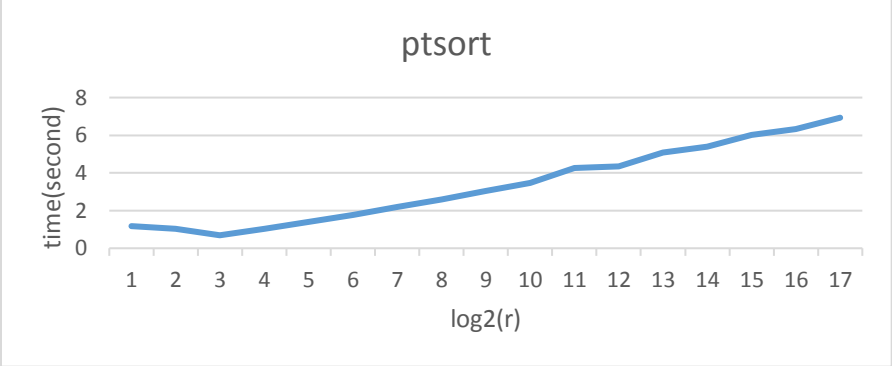

Figure 4: Time cost(second) of PT sort by r when n=10000. we can see the time cost of PT sort is linear when n is fixed.

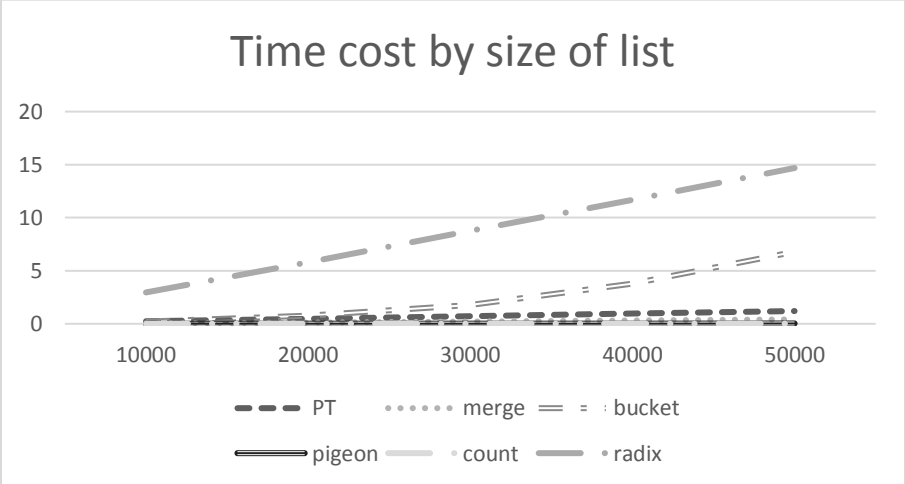

Figure 5: Time cost(second) of sort algorithms by *n* when *r*=1000. From this figure, we can see PT Sort is faster than bucket sort and radix sort when r=1000

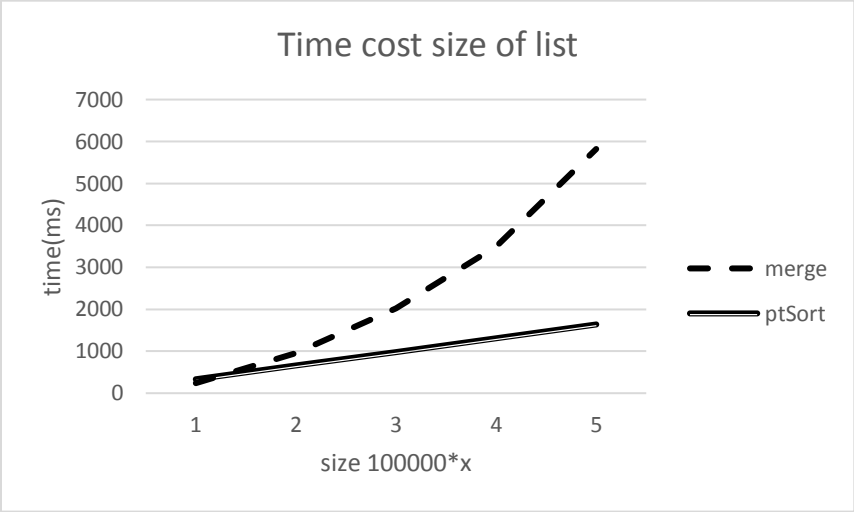

Figure 6: Time cost(second) of PT sort and merge sort by *n* when *r*=1000. We can see PT sort is faster than merge sort when n is bigger than 100000

4.3 TIME & SPACE COMPLEXITY

After comparing it to these famous sorting techniques; for example, bubble sort, selection sort, etc.... The result shows that PT sort is more efficient than selection sort and bubble sort, and faster than merge sort when the range of list is under 1000 and size is bigger than 100000. The rate of change of time cost by the size of PT sort is much smaller than merge sort. The main disadvantage of PT sort is that the time cost increases significantly based on the range of numbers in the list.

The time complexity of PT sort is $O(n \cdot \log_2 r)$ for best, average, and worst. The worst space complexity of PT sort is the same as time complexity which is $O(n \cdot \log_2 r)$. PT sort is a stable sort because the order of the same value element does not change.

Table 1: simply compare the time complexity, space complexity and stability of these sorts.

Algorithm	Average time complexity	Space complexity	Stable
PT	$n \cdot \log_2 r$	$n \cdot \log_2 r$	Yes
merge	$n \cdot \log_2 n$	n	Yes
bucket	$n + r$	$n + r$	No
pigeon	$2n + 2^k$	2^k	Yes
count	$n + r$	$n + r$	Yes
MSD Radix (in-place)	$n \cdot k$	2	No

5 CONCLUSIONS

In this paper, I have proposed a non-comparison sort, PT sort. PT sort is an integer sorting algorithm. Its average time complexity is $O(n \cdot \log_2 r)$, so does its space complexity. PT sort is stable. Its core idea is subtracting the element by the power of two. The experiment result shows that Pt sort is efficient when the range r is small. It can beat merge sort if the list size is big enough. There are two biggest disadvantages of PT sort. First, the time cost can increase significantly when r and n are both enormous. Second, PT sort cannot sort negative numbers. One way to fix it is to separate negative and positive numbers, using absolute value, then reverse the list that contains negative numbers. The future of the PT algorithm is unknown, but it is worth to be exr plored.

5 REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[2] Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". Sorting and Searching. The Art of Computer Programming. 3 (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.

[3] https://en.wikipedia.org/wiki/Merge_sort#Algorithm

[4] NIST's Dictionary of Algorithms and Data Structures: pigeonhole sort

[5] https://titanwolf.org/Network/Articles/Article?AID=805899b3-0338-4b5d-b1aa-79a1bc0690f0#gsc.tab=0

[6] https://en.wikipedia.org/wiki/Pigeonhole_sort#cite_ref-1

[7] https://en.wikipedia.org/wiki/Bucket_sort#cite_ref-lfcs_1-1

[8] Knuth, Donald (1998). "Section 5.2: Internal sorting". Sorting and Searching. The Art of Computer Programming. 3 (2nd ed.). Addison-Wesley. pp. 73–80. ISBN 0-201-89685-0.

[9] Edmonds, Jeff (2008), "5.2 Counting Sort (a Stable Sort)", How to Think about Algorithms, Cambridge University Press, pp. 72–75, ISBN 978-0-521-84931-9.

[10] Sedgewick, Robert (2003), "6.10 Key-Indexed Counting", Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching (3rd ed.), Addison-Wesley, pp. 312–314.

[11] US395781A

[12] https://en.wikipedia.org/wiki/Radix_sort#Specialized_variants