Scientific Research Publishing

Journal of Computer and Communications, 2023, 11, 12-22
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Improvement of Counting Sorting Algorithm

Chenglong Song, Haiming Li

School of Computer Science and Technology, Shanghai University of Electric Power, Shanghai, China
Email: scl341@163.com

How to cite this paper: Song, C.L. and Li, H.M. (2023) Improvement of Counting Sorting Algorithm. *Journal of Computer and Communications*, **11**, 12-22.
https://doi.org/10.4236/jcc.2023.1110002

Received: April 24, 2023
Accepted: July 14, 2023
Published: July 17, 2023

Abstract

By analyzing the internal features of counting sorting algorithm. Two improvements of counting sorting algorithms are proposed, which have a wide range of applications and better efficiency than the original counting sort while maintaining the original stability. Compared with the original counting sort, it has a wider scope of application and better time and space efficiency. In addition, the accuracy of the above conclusions can be proved by a large amount of experimental data.

Keywords

Sort Algorithm, Counting Sorting Algorithms, Complexity, Internal Features

1. Introduction

Counting sort is a linear time complexity sorting algorithm, first proposed by Harold H. Seward in 1954. Prior to this, several sorting algorithms such as bubble sort, insertion sort, and selection sort had been proposed, but their time complexity was quadratic, making them inefficient for handling large-scale data. Therefore, there was a need for more efficient sorting algorithms.

Before counting sort was proposed, several similar algorithms had been studied. For example, bucket sort is a similar linear time complexity sorting algorithm. It works by dividing the data to be sorted into several buckets, sorting the data in each bucket, and finally merging the data in the buckets. The idea behind bucket sort is similar to that of counting sort, but bucket sort needs to sort the data in each bucket while counting sort only needs to count the number of occurrences of each element. Therefore, counting sort has a lower time complexity. Counting sort avoids these problems by counting the number of occurrences of each element, but it is only suitable for small ranges of element values. Otherwise, a large counting array is required, which will waste a lot of space.

A sorting algorithm is a common algorithm in computer science, which aims to arrange a set of data in a particular order. Sorting algorithms are typically

used in data processing, data analysis, database operations, and other fields, and are also one of the fundamental concepts in computer science. Sorting algorithms can be classified according to multiple criteria, such as time complexity, space complexity, and stability. Here are some common classification criteria: Time complexity-based classification. Sorting algorithms can be classified into three categories based on their time complexity: O(n^2) sorting algorithms: These algorithms have a time complexity of O(n^2), where n represents the number of data to be sorted. Examples include bubble sort, selection sort, insertion sort, etc. O(nlogn) sorting algorithms: These algorithms have a time complexity of O(nlogn), where n represents the number of data to be sorted. Examples include quicksort, merge sort, heap sort, etc. Linear time complexity sorting algorithm: These algorithms have a time complexity of O(n), where n represents the number of data to be sorted. Examples include counting sort, radix sort, etc. Stability-based classification Sorting algorithms can be classified into two categories based on their stability: Stable sorting algorithms: If two elements in the data to be sorted have the same value, the relative position of the elements after sorting remains unchanged, and the sorting algorithm is considered stable. Examples include insertion sort, merge sort, etc. Unstable sorting algorithms: If two elements in the data to be sorted have the same value, the relative position of the elements after sorting may change, and the sorting algorithm is considered unstable. Examples include quicksort, heap sort, etc. Space complexity-based classification. Sorting algorithms can be classified into two categories based on their space complexity: In-place sorting algorithms: These algorithms do not require additional storage space during sorting and can directly operate on the original storage space of the data to be sorted. Examples include bubble sort, quicksort, etc.

Non-in-place sorting algorithms: These algorithms require additional storage space during sorting. Examples include merge sort, heap sort, etc. These are the basic concepts and classification criteria of sorting algorithms. In practice, sorting algorithms also have many details to be aware of, such as how to handle the sorting of identical elements and how to handle the boundary cases of data. In actual applications, the appropriate sorting algorithm should be selected based on specific circumstances. Counting sort is a linear time complexity sorting algorithm that is suitable for cases where the input data has a relatively small range of values, but a large number of items. The basic idea behind counting sort is to count the occurrences of each input element, thus determining the position of each element in the output sequence. It should be noted that the elements in the input sequence must be non-negative integers, and the maximum value in the input sequence cannot be too large, otherwise, too much space will be required to allocate the counting array, resulting in high space complexity. The time complexity of counting sort is O(n + k), where n is the number of elements in the input sequence and k is the difference between the maximum and minimum values in the input sequence plus one. The space complexity of counting sort is O(k). Since counting sort is a stable sorting algorithm, it performs better

than other sorting algorithms in certain scenarios. For example, when sorting a small range of integers, counting sort is a good choice. Here I will introduce the improvement of counting sorting algorithm. Counting sort is a simple, stable and efficient sorting algorithm with linear running time, which is a fundamental building block for many applications [1]. Counting sort is one, which lies in this domain [2].

2. The Description and Analysis of Counting Sorting Algorithm

2.1. Algorithm Description

Sorting is an important algorithmic task used in many applications [3]. The code implements a counting sort algorithm with a time complexity of O(n^2). Let's break down the code showing in Figure 1.

The function definition takes an array to be sorted and its length as input. The function definition takes an array to be sorted and its length as input. Create a count array of size n to keep track of the count of each element in the input array, and initialize all elements of the count array to zero. The nested loop iterates over the entire input array, calculates the count of each element in the input array, and stores it in the count array. Specifically, the outer loop iterates over each element of the input array, and the inner loop iterates over the input array again. If the *j*-th element of the input array is less than the *i*-th element of the input array, increment the count of the *i*-th element in the count array. Create an output array of size n, and place each element of the input array in the correct position of the output array based on the count of that element. Specifically, iterate over the input array and place the *i*-th element of the input array into the count[i]-th position in the output array. Copy the sorted output array back into the input array.

2.2. Algorithm Analysis

The time complexity of this algorithm is O(n^2) because it uses two nested loops, each of which iterates over the entire array. For larger input arrays, this algorithm's efficiency may be low.

```
bucket = [0]*n
res = [0]*n
for i in range(0,n-1):
    for j in range(i+1,n):
        if arr[i]<arr[j]:
            bucket[j]=bucket[j]+1
        else:
            bucket[i]=bucket[i]+1
for i in range(0,n):
    res[bucket[i]]=arr[i]
return res
```

Figure 1. Traditional counting sort

The following is the calculation formula.

$$C(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}\left[(n-1)-(i+1)+1\right]=\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2} \quad (1.1)$$

The space complexity of the given counting sort algorithm is O(n), where n is the length of the input array.

In the code, we create three arrays—arr, count, and output. The arr array is the input array, which takes n units of space. The count array is used to keep track of the count of each element in the input array, and thus takes up n units of space. The output array is used to store the sorted output, which also takes n units of space.

Therefore, the total space required by the algorithm is 3n, or O(n). This means that the space required by the algorithm increases linearly with the size of the input array.

3. Two Improvements of Counting Sorting Algorithms

3.1. Improved Counting Sort Algorithm

Counting sort is basically a linear time sorting algorithm [4]. The basic idea of counting sort is to determine the number of elements in the given input sequence that are less than x for each element x (this is not a comparison of the size of each element, but rather determining it by counting the elements and accumulating the count values). Once this information is available, x can be directly placed in the correct position in the final output sequence. For example, if there are only 17 elements in the input sequence that are less than the value of x, x can be directly placed in the 18th position of the output sequence. Of course, if multiple elements have the same value, we cannot place these elements in the same position in the output sequence, so the above plan needs to be modified accordingly.

Allocate additional space based on the range of the difference between the maximum and minimum elements in the collection to be sorted.

Traverse the collection to be sorted and record the number of times each element appears in the additional space corresponding to the element value.

Perform calculations on the data in the additional space to determine the correct position of each element.

Move each element in the collection to be sorted to its calculated correct position.

To generate the dataset, we first created a random integer array with a length of 20, to accomplish this, we used Python's random number generation function and specified that the generated random numbers fall within this range. Specifically, we used the rand function from the random library to generate integers, and then stored these integers in the array to construct the dataset we needed. So as to form the simple dataset we need for demonstration. Randomly select 20 numbers within 10, let's assume we have an array of 20 numbers: {9, 3, 5, 4, 9, 1, 2, 7, 8, 1, 3, 6, 5, 3, 4, 0, 10, 9, 7, 9}.

Let's first traverse this unordered random array and find the maximum value of 10 and the minimum value of 0. This way our corresponding counting range will be 0 - 10. Then, each integer is assigned to its corresponding array index based on its value, and the element of the corresponding array index is incremented by 1.

For example, if the first integer is 9, then the element at array index 9 is incremented by 1, as shown in the diagram below. Showing in Figure 2.

The second integer is 3, so the element at array index 3 is incremented by 1, as shown in the diagram below. Showing in Figure 3.

We continue to traverse the array and modify the array… Finally, when the array is completely traversed, the state of the array is as shown in the diagram below. Showing in Figure 4.

Each value in the array represents the number of times the corresponding integer appears in the array.

With this counting result, sorting is straightforward. Simply traverse the array and output the index value of the array element. The value of the element is output as many times as it appears. For example, the value 1 in the counting result means there are two 1's in the array. This way we get the final sorted result:

0, 1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 9, 9, 9, 9, 10.

Showing in Figure 5, we first define a constant MAX_VALUE to represent the maximum input value. We then define a count array and an output array. In the counting Sort function, we first traverse the input array and increment the count of each element in the count array. We then compute the prefix sum of the count array to determine the position of each element in the output array. Finally, we traverse the input array again and place each element in its correct position in the output array.

The time complexity of the Counting Sort algorithm implemented in the provided C code is O(n + k), where n is the number of elements in the input array and k is the range of input values.

The space complexity of the Counting Sort algorithm implemented in the provided C code is O(n + k), where n is the number of elements in the input array and k is the range of input values.

Although counting sort seems powerful, it has two major limitations:

When the difference between the maximum and minimum values in the sequence is too large, counting sort is not suitable.

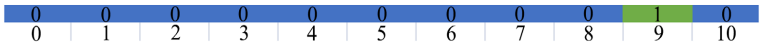

Figure 2. The first integer is 9

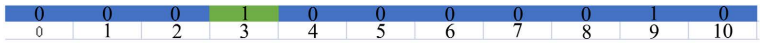

Figure 3. The second integer is 3

Figure 4. The state diagram of the array

```
bucketLen = maxValue+1
bucket = [0]*bucketLen
sortedIndex = 0
arrLen = len(arr)
for i in range(arrLen):
    if not bucket[arr[i]]:
        bucket[arr[i]] = 0
    bucket[arr[i]] += 1
for j in range(bucketLen):
    while bucket[j]>0:
        arr[sortedIndex] = j
        sortedIndex += 1
        bucket[j] -= 1
```

Figure 5. Improved counting sort

For example, if 20 random integers are given with a range between 0 and 10 billion, using counting sort would require creating an array of length 10 billion, which not only wastes space, but also increases the time complexity.

When the elements in the sequence are not integers, counting sort is not suitable.

If the elements in the sequence are all decimals, such as 3.1415926 or 0.000000001, it is impossible to create a corresponding counting array, and counting sort cannot be performed.

It is precise because of these two limitations that counting sort is not widely used like quicksort or merge sort.

Therefore, the optimal solution is bucket sort.

3.2. Bucket Sort Algorithm

Bucket sort is a sorting algorithm that divides data into multiple buckets based on the range of the data values, and then sorts each bucket individually. The number of buckets should be sufficient to ensure that the range of data values in each bucket is as small as possible, which can make the sorting process faster. Finally, all the buckets are merged to obtain the sorted data.

Bucket sort is an upgraded version of counting sort. It utilizes the mapping relationship of a function, and the key to its efficiency lies in the determination of this mapping function. The basic idea of bucket sort is to assume that the input data follows a uniform distribution, and divide the data into a finite number of buckets. Each bucket is then sorted individually (either using another sorting algorithm or by recursively using bucket sort), and finally, the elements in the non-empty buckets are placed back into the original sequence one by one. In order to make bucket sort more efficient, we need to achieve two things: increase the number of buckets as much as possible when there is sufficient extra space, and use a mapping function that can distribute the N input data uniformly into K buckets.

The algorithmic process of bucket sort is as follows:

1) Set a quantified array as an empty bucket.

2) Traverse the input data and put each data item into the corresponding bucket.

3) Sort each non-empty bucket.

4) Concatenate the sorted data items from the non-empty buckets.

Bucket sort and RADIX sort are two well-known integer sorting algorithms [5]. Bucket sort is a linear sorting algorithm that divides the input into several buckets and then sorts the elements in each bucket. The sorted elements from each bucket are then concatenated to form the final sorted sequence.

Determine the number of buckets and the range of values for each bucket. For example, if the range of input elements is [0, 100] and there are 10 buckets, then the range of each bucket would be [0, 10), [10, 20)… [90, 100).

Sort each bucket. Any sorting algorithm can be used for this step, such as insertion sort or quicksort. Concatenate the sorted elements from each bucket to obtain the final sorted sequence. Bucket sort has a time complexity of O(n), where n is the number of elements to be sorted. Therefore, it has linear time complexity. However, bucket sort requires additional space to store the buckets, resulting in a space complexity of O(k), where k is the number of buckets. It is important to note that bucket sort is suitable for cases where the range of input values is relatively small. If the range is large, using a large number of buckets can increase the space and time overhead. The code is showing in Figure 6.

The time complexity of the given bucket sort code is O(n + k), where n represents the number of elements to be sorted, and k represents the number of

```
if len(numList)==0 or len(numList)==1:
    return numList
bucketSize = (maxNum-minNum+1)//bucketNum
buckets = [[] for i in range(bucketNum)]


try:
    for i in range(len(numList)):
        buckets[(numList[i]-minNum)//bucketSize].append(numList[i])
except:
    pass
else:
    pass

for i in range(bucketNum):
    buckets[i].sort()

res = []
for i in range(len(buckets)):
    for j in range(len(buckets[i])):
        res.append(buckets[i][j])
```

Figure 6. Bucket sort

buckets. This is because the code includes steps for iterating through the elements, placing them into the appropriate buckets, sorting the elements within each bucket, and concatenating the sorted elements from all buckets. The time complexity of iterating through the elements and concatenating the sorted elements is O(n), while the time complexity of sorting the elements within each bucket is O(klogk) (assuming quicksort is used), resulting in a total time complexity of O(n + k).

The space complexity of bucket sort is also O(n + k), where n represents the number of elements to be sorted and k represents the number of buckets. This is because additional space is required to store the buckets and to sort the elements within each bucket. Therefore, the space complexity of bucket sort is dependent on the data size and the number of buckets. If the data size is large, more buckets will be required, resulting in increased space complexity.

The limitations of bucket sort include:

Requires prior knowledge of the range of input data: Bucket sort assumes that the input data is uniformly distributed within a known range. If this assumption is not met, the buckets may not be able to effectively capture the distribution of the data, leading to suboptimal sorting performance.

Extra space requirement: In order to store the buckets and their contents, bucket sort requires additional memory space. The number of buckets and the size of the buckets also need to be determined based on the range of input data, which can be challenging in some cases.

Inefficient for small data sets: For very small data sets, the overhead of creating and managing the buckets can outweigh the benefits of the sorting algorithm itself, making bucket sort less efficient than other sorting algorithms.

4. Experimental Demonstration

4.1. Dataset Selection

In the experimental section, to generate the dataset, we first created a random integer array with a length of 1000. This means that the array contains 1000 integers. The values of the data were limited to the range of 100 to 1000. To accomplish this, we used Python's random number generation function and specified that the generated random numbers fall within this range. Specifically, we used the rand function from the random library to generate integers, and then stored these integers in the array to construct the dataset we needed.

We conducted three experiments with increasing data sizes of 500, 700, 1000, and 5000, respectively. The number of comparisons performed in each experiment were 5, 50, 500, and 1000, respectively. The number of buckets used in each experiment were 500, 700, 700, and 5000, respectively. The experimental code is showing in Figure 7.

4.2. Experimental Result

See Table 1.

```
compare=500
n=1000
arry=[]
bucketNum=700

list_countingSort=[]
list_countingSort1=[]
list_bucket_sort=[]

for i in range(compare):
    for j in range(n):
        arry.append(random.randint(100, 1000))
    max_Value=max(arry)
    min_Value=min(arry)
    time_start_countingSort = time.time()
    countingSort(arry, n)
    time_end_countingSort = time.time()
    list_countingSort.append(time_end_countingSort-time_start_countingSort)

    time_start_countingSort1 = time.time()
    countingSort1(arry, max_Value)
    time_end_countingSort1 = time.time()
    list_countingSort1.append(time_end_countingSort1-time_start_countingSort1)

    time_start_bucket_sort = time.time()
    bucket_sort(arry, bucketNum,max_Value,min_Value)
    time_end_bucket_sort = time.time()
    list_bucket_sort.append(time_end_bucket_sort-time_start_bucket_sort)


sum_countingSort=0
sum_countingSort1=0
sum_bucket_sort=0
for i in range(compare):
    sum_countingSort += list_countingSort[i]
    sum_countingSort1 += list_countingSort1[i]
    sum_bucket_sort += list_bucket_sort[i]
```

Figure 7. Algorithm comparison

Table 1. Comparison of experimental results

Data scale	Comparison number	Range	Original Counting sort	Improved Counting sort	Bucket sorting	Barrels
500	5	[100, 1000]	0.00680	0.00040	0.00001	500
700	50	[100, 1000]	0.01443	0.00240	0.00166	700
1000	500	[100, 1000]	0.03500	0.03038	0.02550	700
5000	1000	[100, 1000]	0.80284	0.36080	0.00109	5000

4.3. Result Analysis

By analyzing the results in Table 1, it can be seen that the improved counting sort algorithm always outperforms the original counting sort for any given input size, especially when the input data size is small. As the data size and the number

of comparisons increase, the efficiency of bucket sort is highest when the number of buckets is equal to the data size.

5. Conclusion

Algorithms have a vital and key role in solving the computational problems, informally an algorithm is a well defined computational procedure that takes input and produces output [6]. Improved bucket sort is suitable for cases where the range of values of the elements to be sorted is relatively small. However, in cases where the range of values is relatively small, bucket sort has excellent efficiency and stability. Bucket sort is a sorting algorithm based on the divide and conquer strategy. The core idea is to divide the input elements into multiple ordered buckets, sort the elements in each bucket, and then merge all the buckets into a sorted sequence. The advantages of bucket sort are as follows: Easy to implement: The implementation of bucket sort is straightforward, and it is easy to understand and implement. We just need to allocate the elements into buckets and sort them, then merge the elements in the buckets. Excellent time complexity: When the input elements are uniformly distributed among the buckets, the time complexity of bucket sort can reach linear level O(n), which is quite fast. Good stability: Bucket sort is a stable sorting algorithm, which can ensure that the order of equal elements will not be changed. Wide applicability: Bucket sort is suitable for situations where the input elements are relatively evenly distributed, especially for data sets with small ranges and concentrated values. In summary, bucket sort is a simple and efficient sorting algorithm with a wide range of applications. Although log-linear algorithms have attracted the bulk of research attention in the past [7]. For some specific sorting problems, bucket sort is a good choice. If the range of values of the elements to be sorted is too large, the number of buckets will be large, which increases the space complexity and is not conducive to improving sorting efficiency.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Bajpai, K. and Kots, A. (2014) Implementing and Analyzing an Efficient Version of Counting Sort (E-Counting Sort). *International Journal of Computer Applications*, **98**. https://doi.org/10.5120/17208-7427

[2] Kumar, R. (2019) Modified Counting Sort. In: Kapur, P., Klochkov, Y., Verma, A. and Singh, G., Eds., *System Performance and Management Analytics*. Springer, Singapore, 251-258. https://doi.org/10.1007/978-981-10-7323-6_21

[3] Ahrendt, W., Beckert, B., Bubel, R., *et al.* (2016) Lecture Notes in Computer Science. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., Eds., *Deductive Software Verification—The Key Book*, **10001**, SpringerLink, Berlin.

[4] Srivastava, S., Jaiswal, U.C. and Mall, S. (2018) An Innovative Counting Sort Algo-

rithm for Negative Numbers. *International Journal of Applied Engineering Research*, **13**, 195-201.

[5] Horsmalahti, P. (2012) Comparison of Bucket Sort and Radix Sort.

[6] Zafar, S. and Wahab, A. (2009) A New Friends Sort Algorithm. 2009 2*nd IEEE International Conference on Computer Science and Information Technology.* IEEE,: 326-329.

[7] Shutler, P.M.E., Sim, S.W. and Lim, W.Y.S. (2008) Analysis of Linear Time Sorting Algorithms. *The Computer Journal*, **51**, 451-469. https://doi.org/10.1093/comjnl/bxm097