How is counting sort a stable sort?

Suppose my input is (`a` , `b` and `c` to distinguish between equal keys)

```
1 6a 8 3 6b 0 6c 4
```

My counting sort will save as (discarding the `a` , `b` and `c` info!!)

```
0(1) 1(1) 3(1) 4(1) 6(3) 8(1)
```

which will give me the result

```
0 1 3 4 6 6 6 8
```

So, how is this stable sort? I am not sure how it is "maintaining the relative order of records with equal keys."

Please explain.

algorithm sorting stable-sort

asked Apr 3 '10 at 18:19

Lazer
29.2k 80 226 318

4 Answers

Simple, really: instead of a simple counter for each 'bucket', it's a linked list.

That is, instead of

```
0(1) 1(1) 3(1) 4(1) 6(3) 8(1)
```

You get

```
0(.) 1(.) 3(.) 4(.) 6(a,b,c) 8(.)
```

(here I use `.` to denote some item in the bucket).

Then just dump them back into one sorted list:

```
0 1 3 4 6a 6b 6c 8
```

That is, when you find an item with key `x` , knowing that it may have other information that distinguishes it from other items with the same key, you don't just increment a counter for bucket `x` (which would discard all those extra information).

Instead, you have a linked list (or similarly ordered data structure with constant time amortized append) for each bucket, and you append that item to the end of the list for bucket `x` as you scan the input left to right.

So instead of using `O(k)` space for `k` counters, you have `O(k)` initially empty lists whose sum of lengths will be `n` at the end of the "counting" portion of the algorithm. This variant of counting sort will still be `O(n + k)` as before.

edited Apr 3 '10 at 18:58 answered Apr 3 '10 at 18:24

polygenelubricants
247k 90 485 579

6 That's not a counting sort, it's a degenerate case of a bucket sort, with one bucket for each equivalence class of the equivalence relation induced by the order relation. The reason counting sort is "stable" is because by definition it has one count for each possible distinct value. – Steve Jessop Apr 4 '10 at 0:10

2 Counting sort is a special case of bucket sort, so I have no problem with what you're saying. Or what I said. – polygenelubricants Apr 4 '10 at 7:49

Ah, sorry, I thought your answer was claiming that your algorithm, using linked lists, was a kind of counting sort. – Steve Jessop Apr 4 '10 at 23:05

1 That's how I understood it too. Actually it's very confusing. The OP asked about counting sort, but your explanation is for a simplified bucket sort. The counts alone are enough, you don't need any linked list.... – Karoly Horvath Mar 12 '13 at 22:59

To understand why counting sort is stable, you need to understand that counting sort can not only be used for sorting a list of integers, it can also be used for sorting a list of elements whose key is an integer, and these elements will be sorted by their keys while having additional information associated with each of them.

A counting sort example that sorts elements with additional information will help you to understand this. For instance, we want to sort three stocks by their prices:

```
[(GOOG 3), (CSCO 1), (MSFT 1)]
```

Here stock prices are integer keys, and stock names are their associated information.

Expected output for the sorting should be:

```
[(CSCO 1), (MSFT 1), (GOOG 3)]
(containing both stock price and its name,
and the CSCO stock should appear before MSFT so that it is a stable sort)
```

A counts array will be calculated for sorting this (let's say stock prices can only be 0 to 3):

```
counts array: [0, 2, 0, 1] (price "1" appear twice, and price "3" appear once)
```

If you are just sorting an integer array, you can go through the counts array and output "1" twice and "3" once and it is done, and the entire counts array will become an all-zero array after this.

But here we want to have stock names in sorting output as well. How can we obtain this additional information (it seems the counts array already discards this piece of information)? Well, **the associated information is stored in the original unsorted array**. In the unsorted array [(GOOG 3), (CSCO 1), (MSFT 1)], we have both the stock name and its price available. If we get to know which position (GOOG 3) should be in the final sorted array, we can copy this element to the sorted position in the sorted array.

To obtain the final position for each element in the sorted array, unlike sorting an integer array, you don't use the counts array directly to output the sorted elements. Instead, counting sort has an additional step which calculates the cumulative sum array from the counts array:

```
counts array: [0, 2, 2, 3] (i from 0 to 3: counts[i] = counts[i] + counts[i - 1])
```

This cumulative sum array tells us, if a $1 price stock appears at the first time, it should be outputted to the second position of the sorted array and if a $3 price stock appears at the first time, it should be outputted to the third position of the sorted array. If a $1 stock appears and its element gets copied to the sorted array, we will decreased its count in the counts array.

```
counts array: [0, 1, 2, 3]
(so that the second appearance of $1 price stock's position will be 1)
```

So we can iterate the unsorted array from backwards (this is important to ensure the stableness), check its position in the sorted array according to the counts array, and copied it to the sorted array.

```
sorted array: [null, null, null]
counts array: [0, 2, 2, 3]

iterate stocks in unsorted stocks from backwards
1. the last stock (MSFT 1)
sorted array: [null, (MSFT 1), null] (copy to the second position because counts[1]
== 2)
counts array: [0, 1, 2, 3] (decrease counts[1] by 1)

2. the middle stock (CSCO 1)
sorted array: [(CSCO 1), (MSFT 1), null] (copy to the first position because
counts[1] == 1 now)
counts array: [0, 0, 2, 3] (decrease counts[1] by 1)
```

```
3. the first stock (GOOG 3)
sorted array: [(CSCO 1), (MSFT 1), (GOOG 3)] (copy to the third position because
counts[3] == 3)
counts array: [0, 0, 2, 2] (decrease counts[3] by 1)
```

As you can see, after the array gets sorted, the counts array (which is [0, 0, 2, 2]) doesn't become an all-zero array like sorting an array of integers. The counts array is not used to tell how many times an integer appears in the unsorted array, instead, it is used to tell which position the element should be in the final sorted array. And since we decrease the count every time we output an element, we are essentially making the elements with same key's next appearance final position smaller. That's why we need to iterate the unsorted array from backwards to ensure its stableness.

Conclusion:

Since each element contains not only an integer as key, but also some additional information, even if their key is the same, you could tell each element is different by using the additional information, so you will be able to tell if it is a stable sorting algorithm (yes, it is a stable sorting algorithm if implemented appropriately).

References:

Some good materials explaining counting sort and its stableness:

- http://www.algorithmist.com/index.php/Counting_sort (this article explains this question pretty well)
- http://courses.csail.mit.edu/6.006/fall11/rec/rec07.pdf
- http://rosettacode.org/wiki/Sorting_algorithms/Counting_sort (a list of counting sort implementations in different programming languages. If you compare them with the algorithm in wikipedia's entry below about counting sort, you will find most of which doesn't implement the exact counting sort correctly but implement only the integer sorting function and they don't have the additional step to calculate the cumulative sum array. But you could check out the implementation in 'Go' programming language in this link, which does provides two different implementations, one is used for sorting integers only and the other can be used for sorting elements containing additional information)
- http://en.wikipedia.org/wiki/Counting_sort

edited Aug 7 at 9:50

answered Jun 14 '13 at 14:59

nybon
2,820 5 30 45

2 +1 for putting in the effort to write this answer. – gjain Jul 19 '14 at 0:17

The walking backwards part is what I was missing to understanding how to keep the sorted data stable. Thanks! – georaldc Sep 12 at 19:05

Your solution is not a full counting sort, and discards the associated values.

Here's the *full* counting sort algorithm.

After you calculated the histogram:

```
0(1) 1(1) 3(1) 4(1) 6(3) 8(1)
```

you have to calculate the accumulated sums - each cell will contain how many elements are less than or equal to that value:

```
0(1) 1(2) 3(3) 4(4) 6(7) 8(8)
```

Now you start from the *end* of your original list and go *backwards*.

Last element is `4`. There are 4 elements less than or equal to 4. So `4` will go on the 4th position. You decrement the counter for `4`.

```
0(1) 1(2) 3(3) 4(3) 6(7) 8(8)
```

The next element is `6c`. There are 7 elements less than or equal to 6. So `6c` will go to the 7th position. Again, you decrement the counter for `6`.

```
0(1) 1(2) 3(3) 4(3) 6(6) 8(8)
                     ^ next 6 will go now to 6th position
```

As you can see, this algorithm is a stable sort. The order for the elements with the same key will be kept.

answered Mar 12 '13 at 23:09

Karoly Horvath
72k 9 75 147

If your three "6" values are distinguishable, then your counting sort is wrong (it discards information about the values, which a true sort doesn't do, because a true sort only re-orders the values).

If your three "6" values are not distinguishable, then the sort is stable, because you have three indistinguishable "6"s in the input, and three in the output. It's meaningless to talk about whether they have or have not been "re-ordered": they're identical.

The concept of non-stability only applies when the values have some associated information which does not participate in the order. For instance if you were sorting *pointers* to those integers, then you could "tell the difference" between the three 6s by looking at their different addresses. Then it would be meaningful to ask whether any particular sort was stable. A counting sort based on the integer values then would not be sorting the pointers. A counting sort based on the pointer values would not order them by integer value, rather by address.

edited Apr 4 '10 at 1:08

answered Apr 4 '10 at 0:09

Steve Jessop
218k 22 335 584
