Counting sort

In computer science, **counting sort** is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.[1][2][3]

Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the $\Omega(n \log n)$ lower bound for comparison sorting does not apply to it.[1] Bucket sort may be used for many of the same tasks as counting sort, with a similar time analysis; however, compared to counting sort, bucket sort requires linked lists, dynamic arrays or a large amount of preallocated memory to hold the sets of items within each bucket, whereas counting sort instead stores a single number (the count of items) per bucket.[4]

Contents

Input and output assumptions

In the most general case, the input to counting sort consists of a collection of n items, each of which has a non-negative integer key whose maximum value is at most k.[3] In some descriptions of counting sort, the input to be sorted is assumed to be more simply a sequence of integers itself,[1] but this simplification does not accommodate many applications of counting sort. For instance, when used as a subroutine in radix sort, the keys for each call to counting sort are individual digits of larger item keys; it would not suffice to return only a sorted list of the key digits, separated from the items.

In applications such as in radix sort, a bound on the maximum key value k will be known in advance, and can be assumed to be part of the input to the algorithm. However, if the value of k is not already known then it may be computed, as a first step, by an additional loop over the data to determine the maximum key value that actually occurs within the data.

The output is an array of the items, in order by their keys. Because of the application to radix sorting, it is important for counting sort to be a stable sort: if two items have the same key as each other, they should have the same relative position in the output as they did in the input.[1][2]

The algorithm

In summary, the algorithm loops over the items, computing a histogram of the number of times each key occurs within the input collection. It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the items again, moving each item into its sorted position in the output array.[1][2][3]

In pseudocode, this may be expressed as follows:

```
# variables:
#    input -- the array of items to be sorted;
#    key(x) -- function that returns the key for item x
#    k -- a number such that all keys are in the range 0..k-1
#    count -- an array of numbers, with indexes 0..k-1, initially all zero
#    output -- an array of items, with indexes 0..n-1
#    x -- an individual input item, used within the algorithm
#    total, oldCount, i -- numbers used within the algorithm

# calculate the histogram of key frequencies:
for x in input:
    count[key(x)] += 1

# calculate the starting index for each key:
total = 0
for i in range(k):   # i = 0, 1, ... k-1
    oldCount = count[i]
    count[i] = total
    total += oldCount

# copy to output array, preserving order of inputs with equal keys:
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1

return output
```

After the first for loop, `count[i]` stores the number of items with key equal to `i`. After the second for loop, it instead stores the number of items with key less than `i`, which is the same as the first index at which an item with key `i` should be stored in the output array. Throughout the third loop, `count[i]` always stores the next position in the output array into which an item with key `i` should be stored, so each item is moved into its correct position in the output array.[1][2][3] The relative order of items with equal keys is preserved here; i.e., this is a stable sort.

Complexity analysis

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most $k + 1$ times and therefore take $O(k)$ time. The other two for loops, and the initialization of the output array, each take $O(n)$ time. Therefore, the time for the whole algorithm is the sum of the times for these steps, $O(n + k)$.[1][2]

Because it uses arrays of length $k + 1$ and n, the total space usage of the algorithm is also $O(n + k)$.[1] For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space $O(k)$.[5]

Variant algorithms

If each item to be sorted is itself an integer, and used as key as well, then the second and third loops of counting sort can be combined; in the second loop, instead of computing the position where items with key `i` should be placed in the output, simply append `Count[i]` copies of the number `i` to the output.

This algorithm may also be used to eliminate duplicate keys, by replacing the `Count` array with a bit vector that stores a `one` for a key that is present in the input and a `zero` for a key that is not present. If additionally the items are the integer keys themselves, both second and third loops can be omitted entirely and the bit vector will itself serve as output, representing the values as offsets

of the non-zero entries, added to the range's lowest value. Thus the keys are sorted and the duplicates are eliminated in this variant just by being placed into the bit array.

For data in which the maximum key size is significantly smaller than the number of data items, counting sort may be parallelized by splitting the input into subarrays of approximately equal size, processing each subarray in parallel to generate a separate count array for each subarray, and then merging the count arrays. When used as part of a parallel radix sort algorithm, the key size (base of the radix representation) should be chosen to match the size of the split subarrays.[6] The simplicity of the counting sort algorithm and its use of the easily parallelizable prefix sum primitive also make it usable in more fine-grained parallel algorithms.[7]

As described, counting sort is not an in-place algorithm; even disregarding the count array, it needs separate input and output arrays. It is possible to modify the algorithm so that it places the items into sorted order within the same array that was given to it as the input, using only the count array as auxiliary storage; however, the modified in-place version of counting sort is not stable.[3]

History

Although radix sorting itself dates back far longer, counting sort, and its application to radix sorting, were both invented by Harold H. Seward in 1954.[1][4][8]

References

1. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "8.2 Counting Sort", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 168–170, ISBN 0-262-03293-7. See also the historical notes on page 181.
2. 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.
3. 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.
4. Knuth, D. E. (1998), *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 5.2, Sorting by counting, pp. 75–80, and historical notes, p. 170.
5. Burris, David S.; Schember, Kurt (1980), "Sorting sequential files with limited auxiliary storage", *Proceedings of the 18th annual Southeast Regional Conference*, New York, NY, USA: ACM, pp. 23–31, doi:10.1145/503838.503855 (https://doi.org/10.1145%2F503838.503855), ISBN 0897910141.
6. Zagha, Marco; Blelloch, Guy E. (1991), "Radix sort for vector multiprocessors", *Proceedings of Supercomputing '91, November 18-22, 1991, Albuquerque, NM, USA* (http://www.cs.cmu.edu/~scandal/papers/cray-sort-supercomputing91.ps.gz), IEEE Computer Society / ACM, pp. 712–721, doi:10.1145/125826.126164 (https://doi.org/10.1145%2F125826.126164), ISBN 0897914597.
7. Reif, John H. (1985), "An optimal parallel algorithm for integer sorting", *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*, pp. 496–504, doi:10.1109/SFCS.1985.9 (https://doi.org/10.1109%2FSFCS.1985.9), ISBN 0-8186-0644-4.
8. Seward, H. H. (1954), "2.4.6 Internal Sorting by Floating Digital Sort", *Information sorting in the application of electronic digital computers to business operations* (http://bitsavers.org/pdf/mit/whirlwind/R-series/R-232_Information_Sorting_in_the_Application_of_Electronic_Digital_Computers_to_Business_Operations_May54.pdf) (PDF), Master's thesis, Report R-232, Massachusetts Institute of Technology, Digital Computer Laboratory, pp. 25–28.

External links

- Counting Sort html5 visualization (http://www.cs.usfca.edu/~galles/visualization/CountingSort.html)
- Demonstration applet from Cardiff University (http://users.cs.cf.ac.uk/C.L.Mumford/tristan/CountingSort.html)
- Kagel, Art S. (2 June 2006), "counting sort", in Black, Paul E., *Dictionary of Algorithms and Data Structures* (https://xlinux.nist.gov/dads/HTML/countingsort.html), U.S. National Institute of Standards and Technology, retrieved 2011-04-21.

- A simple Counting Sort implementation. (http://www.codenlearn.com/2011/07/simple-counting-sort.html)
