

**Faculty of Engineering and Technology**

**Department of Computer Science**

**Comp232**

**Data Structures**

**Project #4 (Sorting Algorithms)**

**Student name: Ahmad Salah**

**Student id: 1130083**

**Instructor Name: Dr.Mamnoun Nawahdeh**

**Index**

* Cocktail sort page 3-4
* Comb sort page 5-6
* Counting sort page 7-9
* Gnome sort page 10-11
* Strand sort page 12-13
* Cconclusion page 14
* References page 15-16
* Cocktail Sort
* Cocktail sort, other names 🡺(bidirectional bubble sort, cocktail shaker sort, shaker sort (which can also refer to a variant of [selection sort](http://en.wikipedia.org/wiki/Selection_sort)), ripple sort, shuffle sort,[[1]](http://en.wikipedia.org/wiki/Cocktail_sort#cite_note-Duhl1986-1) or shuttle sort, is a variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) that is both a [stable](http://en.wikipedia.org/wiki/Stable_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) and a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). )
* The algorithm differs from a [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) in that it sorts in both directions on each pass through the list. This sorting algorithm is only marginally more difficult to implement than a bubble sort, and solves the problem of [turtles](http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles) in bubble sorts. It provides only marginal performance improvements, and does not improve asymptotic performance; like the bubble sort, it is not of practical interest ([insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) is preferred for simple sorts), though it finds some use in education

## **Pseudocode** 🡺

##

**Code explanation** 🡺 the first rightward pass will shift the largest element to its correct place at the end, and the following leftward pass will shift the smallest element to its correct place at the beginning.

The second complete pass will shift the second largest and second smallest elements to their correct places, and so on. After (i) passes, the first (i) and the last I elements in the list are in their correct positions, and do not need to be checked. By shortening the part of the list that is sorted each time, the number of operations can be halved

**Complexity** 🡺 The complexity of cocktail sort in [big O notation](http://en.wikipedia.org/wiki/Big_O_notation) is  for both the worst case and the average case, but it becomes closer to  if the list is mostly ordered before applying the sorting algorithm, for example, if every element is at a position that differs at most k (k ≥ 1) from the position it is going to end up in, the complexity of cocktail sort becomes 



* Comb Sort

## Comb sort is a relatively simple [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) originally designed by Włodzimierz Dobosiewicz in 1980.[[1]](http://en.wikipedia.org/wiki/Comb_sort#cite_note-BB-1) Later it was rediscovered by Stephen Lacey and Richard Box in 1991.[[2]](http://en.wikipedia.org/wiki/Comb_sort#cite_note-2) Comb sort improves on [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort).🡺Algorithm[[edit](http://en.wikipedia.org/w/index.php?title=Comb_sort&action=edit&section=1)]

The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. *Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort.

In bubble sort, when any two elements are compared, they always have a *gap* (distance from each other) of 1. The basic idea of comb sort is that the gap can be much more than 1 ([Shell sort](http://en.wikipedia.org/wiki/Shell_sort) is also based on this idea, but it is a modification of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)rather than bubble sort).

In other words, the inner loop of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), which does the actual *swap*, is modified such that gap between swapped elements goes down (for each iteration of outer loop) in steps of shrink factor. i.e. [ input size / shrink factor, input size / shrink factor^2, input size / shrink factor^3, ...., 1 ]. Unlike in [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), where the gap is constant i.e. 1.

The gap starts out as the length of the list being sorted divided by the *shrink factor* (generally 1.3; see below), and the list is sorted with that value (rounded down to an integer if needed) as the gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equivalent to a bubble sort, but by this time most turtles have been dealt with, so a bubble sort will be efficient.

The shrink factor has a great effect on the efficiency of comb sort. In the original article, the authors suggested 1.3. A value too small slows the algorithm down because more comparisons must be made, whereas a value too large means that no comparisons will be made. Lacey and Box empirically found (by testing Combsort on over 200,000 random lists) the shrink factor of 1.3 to be the best.

### 🡺Pseudocode

### 🡺 Complexity

* Counting sort :
* **counting sort** is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) for [sorting](http://en.wikipedia.org/wiki/Sorting_algorithm) a collection of objects according to keys that are small [integers](http://en.wikipedia.org/wiki/Integer); that is, it is an [integer sorting](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Radix_sort), that can handle larger keys more efficiently.

# The Difference between Counting and Bucket sorting :

## Because counting sort uses key values as indexes into an array, it is not a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort), and the [Ω](http://en.wikipedia.org/wiki/Big_O_notation#Family_of_Bachmann.E2.80.93Landau_notations)(*n* log *n*)[lower bound](http://en.wikipedia.org/wiki/Lower_bound) for comparison sorting does not apply to it.[[1]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-1) [Bucket sort](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Linked_list),[dynamic arrays](http://en.wikipedia.org/wiki/Dynamic_array) 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.🡺 Input and output assumptions

In the most general case, the input to counting sort consists of a [collection](http://en.wikipedia.org/wiki/Collection_%28computing%29) of *n* items, each of which has a non-negative integer key whose maximum value is at most *k*.[[3]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-sedgewick-3) In some descriptions of counting sort, the input to be sorted is assumed to be more simply a sequence of integers itself,[[1]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-1) but this simplification does not accommodate many applications of counting sort. For instance, when used as a subroutine in [radix sort](http://en.wikipedia.org/wiki/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 by an additional loop over the data to determine the maximum key value that actually occurs within the data.

The output is an [array](http://en.wikipedia.org/wiki/Array_data_structure) 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](http://en.wikipedia.org/wiki/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.

## The algorithm:

In summary, the algorithm loops over the items, computing a [histogram](http://en.wikipedia.org/wiki/Histogram) of the number of times each key occurs within the input collection. It then performs a [prefix sum](http://en.wikipedia.org/wiki/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]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-1)[[2]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-edmonds-2)[[3]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-sedgewick-3)

In pseudocode, this may be expressed as follows:



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]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-1)[[2]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-edmonds-2)[[3]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-sedgewick-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* + 1times 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]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-1)[[2]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-edmonds-2)

Because it uses arrays of length *k* + 1 and *n*, the total space usage of the algorithm is also *O*(*n* + *k*).[[1]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-clrs-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]](http://en.wikipedia.org/wiki/Counting_sort#cite_note-5)



* Gnome sort

is a sorting algorithm which is similar to insertion sort, except that moving an element to its proper place is accomplished by a series of swaps, as in bubble sort. It is conceptually simple, requiring no nested loops. The running time is O(n^2), but tends towards O(n) if the list is initially almost sorted.[2] In practice the algorithm can run as fast as Insertion sort[citation needed]. The average runtime is O(n^2).

The algorithm always finds the first place where two adjacent elements are in the wrong order, and swaps them. It takes advantage of the fact that performing a swap can introduce a new out-of-order adjacent pair only right before or after the two swapped elements. It does not assume that elements forward of the current position are sorted, so it only needs to check the position directly before the swapped elements.

* **Pseudocod:**
* [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) for the gnome sort using a [zero-based array](http://en.wikipedia.org/wiki/Array_data_type#Index_origin):



## Optimization :

The gnome sort may be optimized by introducing a variable to store the position before traversing back toward the beginning of the list. This would allow the "gnome" to [teleport](http://en.wikipedia.org/wiki/Teleportation) back to his previous position after moving a flower pot. With this optimization, the gnome sort would become a variant of the [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). The animation in the introduction to this topic takes advantage of this optimization.

Here is [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) for an optimized gnome sort using a [zero-based array](http://en.wikipedia.org/wiki/Array_data_type#Index_origin):



## Complexity



Strand sort

**Strand sort** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). It works by repeatedly pulling sorted sublists out of the list to be sorted and merging them with a result array. Each iteration through the unsorted list pulls out a series of elements which were already sorted, and [merges](http://en.wikipedia.org/wiki/Mergesort) those series together.

The name of the algorithm comes from the "strands" of sorted data within the unsorted list which are removed one at a time. It is a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) due to its use of comparisons when removing strands and when merging them into the sorted array.

The strand sort algorithm is [O](http://en.wikipedia.org/wiki/Big-O_notation)(*n*2) in the average case. In the best case (a list which is already sorted) the algorithm is linear, or O(*n*). In the worst case (a list which is sorted in reverse order) the algorithm is O(*n*2).

Strand sort is most useful for data which is stored in a linked list, due to the frequent insertions and removals of data. Using another data structure, such as an array, would greatly increase the running time and complexity of the algorithm due to lengthy insertions and deletions. Strand sort is also useful for data which already has large amounts of sorted data, because such data can be removed in a single strand.

Strand sort Pseudocod**:**

* A simple way to express strand sort in [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) is as follows:



## Strand sort Time Complexity





* **Conclusion**

**Since we learn algorithm to calculate the most efficient space and least time for huge data**

**For example if I want find the maximum and the minimum number I will use the cocktail sort since it’s a variation from bubble sort (it decrease the number of passes and work in both direction so from only one pass I will get both the maximum and mininmum number)**

**Also if I want to** eliminate  small values near the end of the list I can use the shell sort or comb sort since they similar in the idea of gap and this gap keep shrinks until it become one(compare adjacent element ) but in comb sort the shrink factor is 1.3 each pass but in shell sort its n/2 also shell sort is used to ease and sort the data for the insertion sort

In the case of counting sort we must know it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items also its only for numbers not for character and it’s a is often used as a subroutine in Radix Sort.

In the gnome sorting algorithm its similar to bubble sort in the idea of comparing adjacent element but for each comparison in the gnome sort we have to check the the previous elements at least once and in worst case it will keep comparing backword until the start of the array

Finaly in strand sort case its most useful for data which is stored in a linked list since there is frequent insertions and removals of data we might use it if we need to find how many number above a specific value like if we use a loop to count them
although strand sort use the merge in some stage but its only take sublist as needed not like the merge which always keep divide and merge no matter what

|  |  |  |  |
| --- | --- | --- | --- |
| **Sort name** | **worst** | **Avg6** | **Best** |
| **Cocktail** | **O(N^2)** | **O(N^2)** | **O(N)** |
| **Comb** | **O(n^2)** | **N^2/2^increment** | **N** |
| **Counting** | **O(n+k)** |
| **Gnome** | **O(n^2)** | **O(N^2)** | **O(N)** |
| **Strand** | **O(n^2)** | **O(N^2)** | **O(N)** |

References:

🡺**Cocktail\_sort:**
 <http://en.wikipedia.org/wiki/Cocktail_sort>
Paul E. Black and Bob Bockholt, ["bidirectional bubble sort"](http://xlinux.nist.gov/dads/HTML/bidirectionalBubbleSort.html), in Dictionary of Algorithms and Data Structures (online), Paul E. Black, ed., U.S. National Institute of Standards and Technology. 24 August 2009. (accessed: 5 Feb 2010)

🡺**Strand\_sort:**

 Paul E. Black ["Strand Sort"](http://www.nist.gov/dads/HTML/strandSort.html) from [Dictionary of Algorithms and Data Structures](http://en.wikipedia.org/wiki/Dictionary_of_Algorithms_and_Data_Structures) at [NIST](http://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology).

🡺**Counting Sort:**

^ [Jump up to:***a***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-0) [***b***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-1) [***c***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-2) [***d***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-3) [***e***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-4) [***f***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-5) [***g***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-6) [***h***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-7) [***i***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-clrs_1-8) [Cormen, Thomas H.](http://en.wikipedia.org/wiki/Thomas_H._Cormen); [Leiserson, Charles E.](http://en.wikipedia.org/wiki/Charles_E._Leiserson); [Rivest, Ronald L.](http://en.wikipedia.org/wiki/Ron_Rivest); [Stein, Clifford](http://en.wikipedia.org/wiki/Clifford_Stein) (2001), "8.2 Counting Sort", [*Introduction to Algorithms*](http://en.wikipedia.org/wiki/Introduction_to_Algorithms)(2nd ed.), [MIT Press](http://en.wikipedia.org/wiki/MIT_Press) and [McGraw-Hill](http://en.wikipedia.org/wiki/McGraw-Hill), pp. 168–170, [ISBN](http://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-262-03293-7](http://en.wikipedia.org/wiki/Special%3ABookSources/0-262-03293-7). See also the historical notes on page 181.

^ [Jump up to:***a***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-edmonds_2-0) [***b***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-edmonds_2-1) [***c***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-edmonds_2-2) [***d***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-edmonds_2-3) [***e***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-edmonds_2-4) Edmonds, Jeff (2008), "5.2 Counting Sort (a Stable Sort)", *How to Think about Algorithms*, Cambridge University Press, pp. 72–75,[ISBN](http://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-0-521-84931-9](http://en.wikipedia.org/wiki/Special%3ABookSources/978-0-521-84931-9).

^ [Jump up to:***a***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-sedgewick_3-0) [***b***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-sedgewick_3-1) [***c***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-sedgewick_3-2) [***d***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-sedgewick_3-3) [***e***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-sedgewick_3-4) [Sedgewick, Robert](http://en.wikipedia.org/wiki/Robert_Sedgewick_%28computer_scientist%29) (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.

^ [Jump up to:***a***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-knuth_4-0) [***b***](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-knuth_4-1) [Knuth, D. E.](http://en.wikipedia.org/wiki/Donald_Knuth) (1998), *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.), Addison-Wesley, [ISBN](http://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-201-89685-0](http://en.wikipedia.org/wiki/Special%3ABookSources/0-201-89685-0). Section 5.2, Sorting by counting, pp. 75–80, and historical notes, p. 170.

[**Jump up^**](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-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](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1145/503838.503855](http://dx.doi.org/10.1145/503838.503855).

[**Jump up^**](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-6) Zagha, Marco; [Blelloch, Guy E.](http://en.wikipedia.org/wiki/Guy_Blelloch) (1991), "Radix sort for vector multiprocessors", *Proceedings of Supercomputing '91, November 18-22, 1991, Albuquerque, NM, USA*, IEEE Computer Society / ACM, pp. 712–721, [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1145/125826.126164](http://dx.doi.org/10.1145/125826.126164).

[**Jump up^**](http://en.wikipedia.org/wiki/Counting_sort#cite_ref-7) [Reif, John H.](http://en.wikipedia.org/wiki/John_Reif) (1985), "An optimal parallel algorithm for integer sorting", [*Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*](http://en.wikipedia.org/wiki/Symposium_on_Foundations_of_Computer_Science), pp. 496–504, [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1109/SFCS.1985.9](http://dx.doi.org/10.1109/SFCS.1985.9).

 **🡺 Comb sort:**
 Brejová, B. (15 September 2001). "Analyzing variants of Shellsort". [*Inform. Process. Lett.*](http://en.wikipedia.org/wiki/Information_Processing_Letters) **79** (5): 223–227. [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1016/S0020-0190(00)00223-4](http://dx.doi.org/10.1016/S0020-0190%2800%2900223-4). [edit](http://en.wikipedia.org/w/index.php?title=Template:Cite_doi/10.1016.2FS0020-0190.2800.2900223-4&action=edit&editintro=Template:Cite_doi/editintro2)

🡺 Gnome Sort :
[**Jump up^**](http://en.wikipedia.org/wiki/Comb_sort#cite_ref-2) ["A Fast Easy Sort"](http://cs.clackamas.cc.or.us/molatore/cs260Spr03/combsort.htm), [*Byte* Magazine](http://en.wikipedia.org/wiki/Byte_Magazine), April 1991

 Sarbazi-Azad, Hamid (2 October 2000). ["Stupid Sort: A new sorting algorithm"](http://sina.sharif.edu/~azad/stupid-sort.PDF). *Newsletter* (Computing Science Department, Univ. of Glasgow) (599): 4. Retrieved25 November 2014.

[**Jump up^**](http://en.wikipedia.org/wiki/Gnome_sort#cite_ref-2) <http://www.dickgrune.com/Programs/gnomesort.html>

[**Jump up^**](http://en.wikipedia.org/wiki/Gnome_sort#cite_ref-3) Paul E. Black. ["gnome sort"](http://xlinux.nist.gov/dads/HTML/gnomeSort.html). *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. Retrieved 2011-08-20.

THE END