**بسم الله الرحمن الرحيم**



**Data Structures**

**COMP2321**

**Project 4**

**Sorting Algorithms Report**

**Instructor**: Mr. Iyad Jaber

**Name**: Hussein Dahir , **ID**: 1131138

**Section:** 1

**Cocktail Sort[1]:**

**Cocktail sort**, also known as **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**, 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:**

**procedure** cocktailSort( A **:** list of sortable items ) **defined as:**

 // `begin` and `end` marks the first and last index to check

 begin := -1

 end := length( A ) - 2

 **do**

 swapped := false

 // increases `begin` because the elements before `begin` are in correct order

 begin := begin + 1

 **for each** i **in** begin **to** end **do:**

 **if** A[ i ] > A[ i + 1 ] **then**

 swap( A[ i ], A[ i + 1 ] )

 swapped := true

 **end if**

 **end for**

 **if** swapped = false **then**

 **break do-while loop**

 **end if**

 swapped := false

 // decreases `end` because the elements after `end` are in correct order

 end := end - 1

 **for each** i **in** end **to** begin **do:**

 **if** A[ i ] > A[ i + 1 ] **then**

 swap( A[ i ], A[ i + 1 ] )

 swapped := true

 **end if**

 **end for**

 **while** swapped

**end procedure**

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, and that by using the two variables ‘begin’ and ‘end’ which mark the first and last index to check .

**Differences from bubble sorting:**

Cocktail sort is a slight variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). It differs in that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom. It can achieve slightly better performance than a standard bubble sort. The reason for this is that [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) only passes through the list in one direction and therefore can only move items backward one step each iteration.

An example of a list that proves this point is the list (2,3,4,5,1), which would only need to go through one pass of cocktail sort to become sorted, but if using an ascending [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) would take four passes. However one cocktail sort pass should be counted as two bubble sort passes. Typically cocktail sort is less than two times faster than bubble sort.

Another optimization can be that the algorithm remembers where the last actual swap has been done. In the next iteration, there will be no swaps beyond this limit and the algorithm has shorter passes. As the Cocktail sort goes bidirectionally, the range of possible swaps, which is the range to be tested, will reduce per pass, thus reducing the overall running time.

**Complexity:**

The complexity of cocktail sort in [big O notation](http://en.wikipedia.org/wiki/Big_O_notation) is **O(n2)** for both the worst case and the average case, but it becomes closer to **O(n)**  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 **O(k\*n)**  Such cases may be approached by algorithms like comb sort (But note that cocktail sort does not use any extra space).

|  |  |
| --- | --- |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(1) |

[**Gnome sort**](http://en.wikipedia.org/wiki/Gnome_sort)**[2]:**

Named "gnome sort" from the observation that it is "how a gnome sorts a line of flower pots." It is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) which is similar to [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), except that moving an element to its proper place is accomplished by a series of swaps, as in [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). It is conceptually simple, requiring no nested loops. The average, or expected, running time is [***O***](http://en.wikipedia.org/wiki/Big_O_notation)**(*n*2)**, but tends towards ***O*(*n*)** if the list is initially almost sorted. In practice the algorithm can run as fast as [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort).

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 next to 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 previous to the swapped elements.

**Pseudocode:**

procedure gnomeSort(a[])

 pos := 1

 while pos < length(a)

 if (a[pos] >= a[pos-1])

 pos := pos + 1

 else

 swap a[pos] and a[pos-1]

 if (pos > 1)

 pos := pos - 1

 end if

 end if

 end while

end procedure

**Complexity:**

Complexity is[**O(n2)**](http://xlinux.nist.gov/dads/HTML/bigOnotation.html)for arbitrary data, but approaches **O(n)** if the input list is nearly in order. No extra space is needed in this sort.

|  |  |
| --- | --- |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | \Omega(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(1) |

**Odd – Even Sort[3]:**

In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort**) is a relatively simple [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), developed originally for use on parallel processors with local interconnections. It is a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort)related to [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

**Algorithm:**

**function** oddEvenSort(list) {

 **function** swap( list, i, j ){

 **var** temp = list[i];

 list[i] = list[j];

 list[j] = temp;

 }

 **var** sorted = **false**;

 while(!sorted)

 {

 sorted = **true**;

 **for**(**var** i = 1; i < list.length-1; i += 2)

 {

 **if**(list[i] > list[i+1])

 {

 swap(list, i, i+1);

 sorted = **false**;

 }

 }

 **for**(**var** i = 0; i < list.length-1; i += 2)

 {

 **if**(list[i] > list[i+1])

 {

 swap(list, i, i+1);

 sorted = **false**;

 }

 }

 }

}

**Complexity:**

In best case, if the data is sorted, then 2 loops (n/2 each) will be performed only without any swap operation, so the time complexity will be O(n). But if it is not sorted, then the operation will take more than 1\*n loop, and in the worst case the time complexity will be O(n2).

|  |  |
| --- | --- |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(1) |

**Strand Sort[4]:**

**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.

**Algorithm:**

**procedure** strandSort( A : list of sortable items ) **defined as:**

 **while** length( A ) > 0

 **clear** sublist

 sublist[ 0 ] := A[ 0 ]

 **remove** A[ 0 ]

 **for each** i **in** 0 **to** length( A ) - 1 **do:**

 **if** A[ i ] > sublist[ last ] **then**

 **append** A[ i ] **to** sublist

 **remove** A[ i ]

 **end if**

 **end for**

 **merge** sublist **into** results

 **end while**

 **return** results

**end procedure**

1. Parse the unsorted list once, taking out any ascending (sorted) numbers.
2. The (sorted) sublist is, for the first iteration, pushed onto the empty sorted list.
3. Parse the unsorted list again, again taking out relatively sorted numbers.
4. Since the sorted list is now populated, merge the sublist into the sorted list.
5. Repeat steps 3–4 until both the unsorted list and sublist are empty.

**Complexity:**

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.

|  |  |
| --- | --- |
| **Data structure** | [Linked list](http://en.wikipedia.org/wiki/Linked_list) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n²) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n²) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(1) auxiliary |

**Example:**

|  |  |  |
| --- | --- | --- |
| **Unsorted list** | **Sublist (Strand)** | **Sorted list** |
| 3 1 5 4 2 |  |  |
| 1 4 2 | 3 5 |  |
| 1 4 2 |  | 3 5 |
| 2 | 1 4 | 3 5 |
| 2 |  | 1 3 4 5 |
|  | 2 | 1 3 4 5 |
|  |  | 1 2 3 4 5 |

**Stooge Sort[5]:**

**Stooge sort** is a [recursive](http://en.wikipedia.org/wiki/Recursion) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) with a time complexity of O(*n*log 3 / log 1.5 ) = O(*n*2.7095...). The running time of the algorithm is thus slower compared to efficient sorting algorithms, such as [Merge sort](http://en.wikipedia.org/wiki/Merge_sort), and is even slower than [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), a canonical example of a fairly inefficient and simple sort.

The algorithm is defined as follows:

* If the value at the end is smaller than the value at the start, swap them.
* If there are 3 or more elements in the list, then:
	+ Stooge sort the initial 2/3 of the list
	+ Stooge sort the final 2/3 of the list
	+ Stooge sort the initial 2/3 of the list again
* else: exit the procedure

The algorithm gets its name from [slapstick](http://en.wikipedia.org/wiki/Slapstick) routines of [The Three Stooges](http://en.wikipedia.org/wiki/The_Three_Stooges), in which each stooge hits the other two

**Implementation:**

**function** stoogesort(array L, i = 0, j = length(L)-1)

 **if** L[j] < L[i] then

 L[i] ↔ L[j]

 **if** (j - i + 1) > 2 then

 t = (j - i + 1) / 3

 stoogesort(L, i , j-t)

 stoogesort(L, i+t, j )

 stoogesort(L, i , j-t)

 **return** L

**Complexity:**

|  |  |
| --- | --- |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(*n*log 3 /log 1.5) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(*n*) |

**Conclusion:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Best | Average | Worst | Memory | Notes |
| [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) | n | n^2 | n^2 | 1 | No extra space needed |
| [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) | n | n^2 | n^2 | 1 | No extra space needed |
| Odd-Even | n | - | n^2 | 1 | No extra space needed |
| [Strand](http://en.wikipedia.org/wiki/Strand_sort) | n | n^2 | n^2 | n | extra space needed |
| Stooge |  |  | O(*n*log 3 /log 1.5) | n | Recursive, O(n2.70..) |

The best of those may be the Gnome sort, since it doesn’t use any extra space, and its fast if the data is nearly sorted.

**References:**

**[1]:** [**http://en.wikipedia.org/wiki/Cocktail\_sort**](http://en.wikipedia.org/wiki/Cocktail_sort)

**[2]:** [**http://en.wikipedia.org/wiki/Gnome\_sort**](http://en.wikipedia.org/wiki/Gnome_sort)

**[3]:** **http://en.wikipedia.org/wiki/** **Odd–even\_sort**

**[4]:** [**http://en.wikipedia.org/wiki/Strand\_sort**](http://en.wikipedia.org/wiki/Strand_sort)

**[5]:** **http://en.wikipedia.org/wiki/Stooge\_sort**