

**DATA STRUCTURE (COMP232)**

**Sorting Analysis in JAVA**

**Project#4**

**DR.Mamoun Nawahdah**

**Somoud Diab 1110708**

|  |
| --- |
| Index: |
| Bucket sort .. page 3 |
| Cocktail sort .. page 4 |
| Strand sort.. page 5 |
| Gnome sort.. page 7 |
| Comb sort .. page 8 |
| Summary page..page 10 |
| Reference .. page 11 |

Bucket Sort :

Its stable and not in place .

Bucked Sort Algorithm is to divide the array into bucket’s , each bucket deal with it as individually . and match the integer in array to right bucket .

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



Code :

**function** bucketSort(array, n) **is**

 buckets ← new array of n empty lists

 **for** i = 0 **to** (length(array)-1) **do**

 insert *array[i]* into buckets[msbits(array[i], k)]

 **for** i = 0 **to** n - 1 **do**

 nextSort(buckets[i]);

 **return** the concatenation of buckets[0], ...., buckets[n-1]

Advantages :the user knows the range of the elements;

time complexity is good compared to other algorithms.

Disadvantages : you are limited to having to know the greatest element extra memory is required . [1]

Cocktail sort :

Cocktail sort is a stable comparison sorting algorithm and in place, It is a variation of bubble sort.

 In bubble sort, values only bubble in one direction. In cocktail sort, values bubble both directions, thus avoiding turtles.

Code :

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

 **do**

 swapped := false

 **for each** i **in** 0 **to** length( A ) - 2 **do:**

 **if** A[ i ] > A[ i + 1 ] **then** // test whether the two elements are in the wrong order

 swap( A[ i ], A[ i + 1 ] ) // let the two elements change places

 swapped := true

 **end if**

 **end for**

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

 // we can exit the outer loop here if no swaps occurred.

 **break do-while loop**

 **end if**

 swapped := false

 **for each** i **in** length( A ) - 2 **to** 0 **do:**

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

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

 swapped := true

 **end if**

 **end for**

 **while** swapped // if no elements have been swapped, then the list is sorted

**end procedure**

Time:

Worst-case: O(n²)

Best-case: O(n) [2]

Strand sort :

Strand sort is a sorting algorithm. It stable and not in place. It works by repeatedly by pulling the first element in list , next element should larger than the first , when we finished we apply the same to the remaining item in list , merges these list together , we finished when we have one number and that will be the smallest , we merge it at the first .

|  |
| --- |
|  |
| **Unsorted linked list** | **Sub linked list** | **Sorted linked list** |
| 3 - 1 - 5 - 4 – 2 | Original Linked List (Unsorted)  |
| 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 |
| Final Linked List (Sorted!) | 1 - 2 - 3 - 4 - 5 |
| **Table[2]: Sorting a Linked List Using Strand Algorithm.** |

Code :

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

Time :

Worst-case :- O(n²)

Average-case :- O(n²)

Best-case :- O(n) [3]

Gnome sort :

It is stable and in place.

In this sort the Algorithm is comparing the first element with the next element , the first should be smaller than the next if not we swap , on each swapping we check backward if there are any change happened then continue until the last element .

|  |  |
| --- | --- |
| **Current array (index: 0-3)** | **Action to take (pos 🡒 bold & yellow shaded)** |
| [9, 6, 5, 7] | Original Array (Unsorted) |
| [9, **6**, 5, 7] | a[pos] < a[pos-1], swap. |
| [6, **9**, 5, 7] | a[pos] >= a[pos-1], increment pos. |
| [6, 9, **5**, 7] | a[pos] < a[pos-1], swap and pos > 1, decrement pos. |
| [6, **5**, 9, 7] | a[pos] < a[pos-1], swap and pos <= 1, increment pos. |
| [5, 6, **9**, 7] | a[pos] >= a[pos-1], increment pos. |
| [5, 6, 9, **7**] | a[pos] < a[pos-1], swap and pos > 1, decrement pos. |
| [5, 6, **7**, 9] | a[pos] >= a[pos-1], increment pos. |
| [5, 6, 7, **9**] | a[pos] >= a[pos-1], increment pos. |
| [5, 6, 7, 9] | pos == length(a), finished! (Sorted!) |
| **Table[1]: Sorting An Array Using Gnome Algorithm.** |

Code :

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

Time :

Worest Case : 

Best Case : O (n)

Avarage Case :  [4]

**COMB SORT**

It is not stable and in place.

🞛Algorithm Clarification:

A relatively simple sorting algorithm originally designed by Włodzimierz Dobosiewicz in 1980. Later, Stephen Lacey and Richard Box rediscovered it in 1991.

Comb sort improves on bubble sort. In other words, it eliminatessmaller values near the end of the unsorted array.In bubble sort, the distance (gap) between any two elements is to be compared is always 1; but in comb sort the gap can be much more than 1.Here, the gap starts out as the length of the array being sorted divided by a factor called shrink factor (experimentally determined to be 1.3), and then we sort the array with that value as the first stage of sorting (rounded down to an integer if needed). Then the gap itself divided by the shrink factor again, and we sort the array using this new gap as second stage. By repeating this procedure we will finally reach a case in which the gap is 1. At this point, comb sort continues and accomplish the last stage using a gap of 1 (bubble sort) until the list is fully sorted.

🞛JAVA Code:

public static <E extends Comparable<? super E>> void sort(E[] input) {

int gap = input.length;

boolean swapped = true;

while (gap > 1 || swapped) {

if (gap > 1) {

gap = (int) (gap / 1.3);

 }

swapped = false;

for (inti = 0; i + gap <input.length; i++) {

if (input[i].compareTo(input[i + gap]) > 0) {

 E t = input[i];

input[i] = input[i + gap];

input[i + gap] = t;

swapped = true;

 }

 }

 }

}

🞛Time Analysis:

Wort case:O(n2), best case:O(n),average case:O(nlog(n))

[5]

**Summary page;**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort type | Worst time | Averagetime | Best time | Space | Stable | In Place |
| Bucket | O(n2) | O(n+k) | O(n+k) | O(n.k) | Yes | No |
| Cocktail | O(n2) | O(n2) | O(n) | O(1) | Yes | Yes |
| Strand | O(n2) | O(n2) | O(n) | O(n) | Yes | No |
| Gnome | O(n2) | O(n2) | O(n) | O(1) | Yes | Yes |
| Comb | O(n2) | O(nlog(n)) | O(n) | O(1) | No | Yes |

In the worst and best case all sort have the same time, in the average case bucket and comb is more quickly than others, all sort is stable neither comb sort is not stable, all in-place neither bucket nor strand, need additional memory.

**Reference;**

[1]<http://en.wikipedia.org/wiki/Bucket_sort>

[2]<http://en.wikipedia.org/wiki/Cocktail_sort>

[3]<http://en.wikipedia.org/wiki/Strand_sort>

[4]<http://en.wikipedia.org/wiki/Gnome_sort>

[5]<http://en.wikipedia.org/wiki/Comb_sort>

[[5]https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java](https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java)