SOMANATHAN SUBRAMANIYAN

STUDENT ID: G00364742

Computational Thinking with Algorithms

Project: Benchmarking Sorting Algorithm

2019

Table of Contents

[Sorting Algorithms 2](#_Toc7469571)

[Bubble Sort 2](#_Toc7469572)

[Definition 2](#_Toc7469573)

[Logic 2](#_Toc7469574)

[Implementation 3](#_Toc7469575)

[Code Snippet 3](#_Toc7469576)

[Example 3](#_Toc7469577)

[Time and Space complexity 3](#_Toc7469578)

[References 4](#_Toc7469579)

[Quick Sort 4](#_Toc7469580)

[Definition 4](#_Toc7469581)

[Logic 4](#_Toc7469582)

[Implementation 4](#_Toc7469583)

[Code Snippet 5](#_Toc7469584)

[Example 5](#_Toc7469585)

[References 5](#_Toc7469586)

[Comb Sort 6](#_Toc7469587)

[Definition 6](#_Toc7469588)

[Logic 6](#_Toc7469589)

[Implementation 6](#_Toc7469590)

[Code Snippet 6](#_Toc7469591)

[Example 7](#_Toc7469592)

[References 7](#_Toc7469593)

[Bucket Sort 7](#_Toc7469594)

[Definition 8](#_Toc7469595)

[Logic 8](#_Toc7469596)

[Implementation 8](#_Toc7469597)

[Code Snippet 8](#_Toc7469598)

[Example 9](#_Toc7469599)

[References 9](#_Toc7469600)

# Sorting Algorithms

Bubble Sort

## Bubble Sort

Definition

**Bubble Sort** is a sorting algorithm which is used to sort a given set of n input elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

Logic

* Compare 2 consecutive pair of input elements
* Swap elements in pair such that the smaller is first
* When reach end of list, start over again
* Stop when no more swaps have been made
* Largest unsorted element always at end after pass, so at most “n” passes

Implementation

* Inner “for” loop for doing the comparisons
* Outer while loop is for doing multiple passes until no more swaps
* O(n2) where n is len(L) to do len(L) -1 comparisons and len(L)-1 passes

Code Snippet

while (Ocount <= len(data) -1):

Ocount = Ocount + 1

if len(data) > 1:

for i in range(0, len(data)-1):

icount = icount+1

x = data[i]

y = data[i+1]

if x > y:

data[i+1] = x

data[i] =y

return(data)

Example  
1st Pass:

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | 2 | 6 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 6 | 4 |

Compares the first 2 elements and swaps since 5 >2

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 6 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 6 | 4 |

No Swap as 5 < 6

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 6 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

Swap as 6 > 4

2nd Pass:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

No Swap as 2 < 5

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 6 |

Swap as 5 >4

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 6 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 6 |

No Swap as 6 > 5

The array is sorted, but algorithm does not know it is completed. The algorithm needs two whole passes without any swap to know it is sorted.

* Time complexity is [*O*(*n*2)](https://en.wikipedia.org/wiki/Big_O_notation). Its efficiency decreases dramatically on lists of more than a small number of elements.
* The space complexity is *O(1)* as it requires only single additional memory space i.e. for temp variable.
* Best Case Time Complexity [Big omega] is *O(n)*
* Average Time Complexity [Big-theta] is [*O*(*n*2)](https://en.wikipedia.org/wiki/Big_O_notation)

References

* <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec12.pdf>
* <https://www.geeksforgeeks.org/bubble-sort/>
* <https://www.studytonight.com/data-structures/bubble-sort>

Quick Sort

## Quick Sort

### Definition

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given input elements around the chosen pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick first element as pivot.
* Always pick last element as pivot (implemented below)
* Pick a random element as pivot.
* Pick median as pivot.

The random element is used as a pivot in this implementation

### Logic

* Dive the given input element into two subgroups around a pivot x
* Recursively sort the subgroups
* Combine the result set.

Implementation

* A Random Pivot element is chosen from our unsorted array.
* Create 3 distinct groups
  + Equal - for all elements equal to pivot element
  + Smaller - for all elements lower than chosen pivot element
  + Higher - for all elements higher than chosen pivot element
* Iterate through input elements and recursively sort through the higher and smaller groups.
* Merge the smaller, equal and higher groups to get the sorted input elements.

### Code Snippet



Time Complexity (Best Case): O (n log n)

Time Complexity (Worst Case): O (n2)

Space Complexity: O(log n)

Example

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input elements (Unsorted) |  |  | 9 | 8 | 7 | 4 | 2 |  |  |
|  | | | | | | | | | |
| Random Partitioning and quick Sort |  | 4 | 2 |  | 7 |  | 9 | 8 |  |
|  | | | | | | | | | |
| Random Partitioning and quick Sort | 2 |  | 4 |  |  |  | 8 |  | 9 |
|  | | | | | | | | | |
| Output elements (Sorted) |  |  | 2 | 4 | 7 | 8 | 9 |  |  |

### References

* https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-4-quicksort-randomized-algorithms/lec4.pdf
* <https://www.geeksforgeeks.org/quick-sort/>
* <https://tutorialedge.net/compsci/sorting/quicksort-in-python/>

Comb sort

## Comb Sort

### Definition

Comb sort iterates through the input elements multiple times, swapping elements that are out of order as it goes. It is similar to Bubble sort and the main difference is that comb sort looks at elements a certain number of indexes apart, this is called the gap. The input elements are sorted in a specific gap. On completion of each phase, the gap is decreased by a factor of 1.3.

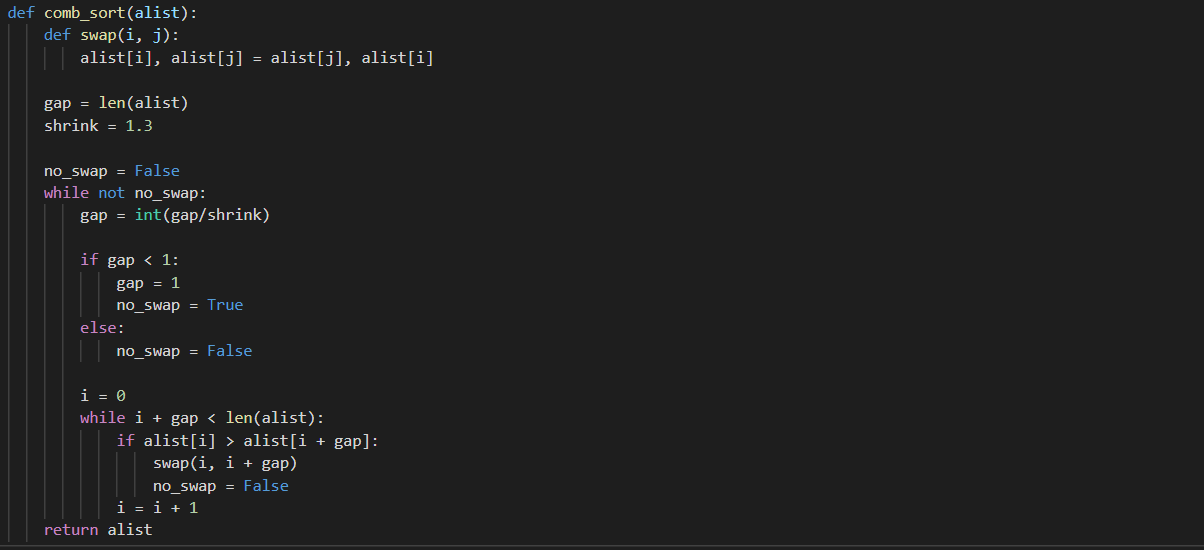
### Logic

* Calculate the Gap value
* Iterating over the input elements comparing each element with an element that is “gap” elements further down the list and swapping them if required.
* Do the above 2 steps until the GAP value reaches 1 and no swaps has occurred.

Implementation

* Calculate the Gap. The 1st gap is equal to “length of the input elements” divided by “1.3”
* Inner While Loop: Iterating over the input elements comparing each element with an element that is “gap” elements further down the list and swapping them.
* Outer while loop: Re-calculate the gap by dividing the previous gap by 1.3
* Do the steps 2 and 3 until gap is “1” or less than “1”

### Code Snippet



Time Complexity (Best Case): O (n log n)

Time Complexity (Worst Case): O (n2)

Space Complexity: O(1)

### Example

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Elements** | | | | | | | **Comments** | **Gap Value** | **Shrink factor** |
| 8 | 5 | 6 | 9 | 7 | 2 | 2 | Unsorted List | 7 | 1.3 |
|  | | | | | | | | | |
| 8 | 5 | 6 | 9 | 7 | 2 | 2 |  | 5 |  |
| 2 | 5 | 6 | 9 | 7 | 8 | 2 | Swap as 8>2 | 5 |  |
| 2 | 2 | 6 | 9 | 7 | 8 | 5 | Swap as 5>2 | 5 |  |
|  | | | | | | | | | |
| 2 | 2 | 6 | 9 | 7 | 8 | 5 | No Swap as 2 <7 | 4 |  |
| 2 | 2 | 6 | 9 | 7 | 8 | 5 | No Swap as 2 <8 | 4 |  |
| 2 | 2 | 5 | 9 | 7 | 8 | 6 | Swap as 6>5 | 4 |  |
|  | | | | | | | | | |
| 2 | 2 | 6 | 9 | 7 | 8 | 5 | No Swaps in 3 iterations | 3 |  |
| 2 | 2 | 6 | 5 | 7 | 8 | 9 | Swap as 9>5 | 3 |  |
|  | | | | | | | | | |
| 2 | 2 | 6 | 5 | 7 | 8 | 9 | No Swaps | 2 |  |
|  | | | | | | | | | |
| 2 | 2 | 6 | 5 | 7 | 8 | 9 | Swap as 6 >5 | 1 |  |
| 2 | 2 | 5 | 6 | 7 | 8 | 9 | Sorted List | 1 |  |

### References

* <https://www.tutorialspoint.com/Comb-Sort>
* <https://www.growingwiththeweb.com/2016/09/comb-sort.html>
* <https://buffered.io/posts/sorting-algorithms-the-comb-sort/>

Bucket sort

## Bucket Sort

### Definition

Bucket sort, or bin sort works by distributing the input elements into a number of buckets. Each bucket is then sorted individually using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

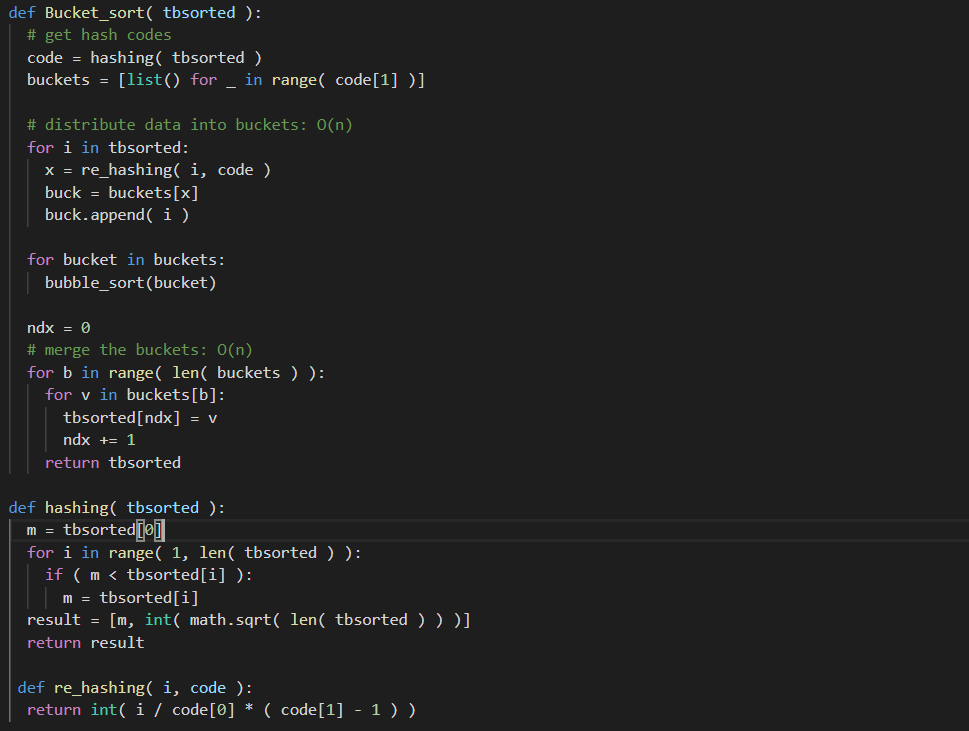
### Logic

* Create “n” buckets of the input elements
* Sort individual buckets using bubble or insertion sort.
* Concatenate all sorted buckets.

Implementation

* Each bucket reflects a unique **hash code** value returned by the hash function used on each element.
* Function **hashing**: Hash Sort creates a suitably large number of buckets k into which the elements are partitioned; as k grows in size, the performance of the Sort improves.
* Each bucket is sorted through the **bubble** sort.
* **For loop** is used to merge the all the buckets and return the sorted elements to the calling function.

### Code Snippet



Time Complexity (Best Case): O (n)

Time Complexity (Worst Case): O (n)

Space Complexity: O(1)

### Example

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 29 | 25 | 18 | 11 | 5 | 10 | 15 | 16 | 21 | 26 |

Input Elements:

5 , 10

11,15

18,16

29,26

25,21

Buckets:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 10 | 11 | 15 | 16 | 18 | 21 | 25 | 26 | 29 |

Sorted input elements:

### References

* https://www.oreilly.com/library/view/algorithms-in-a/9780596516246/ch04s08.html
* <https://www.geeksforgeeks.org/bucket-sort-2/>
* https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124

Shell sort

## Shell Sort

### Definition

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 45 | 5 | 56 | 9 | 57 | 60 | Input data elements |
| 45 |  |  | 9 |  |  | Sub-list 1 (interval of 3) |
|  | 5 |  |  | 57 |  | Sub-list 2 (interval of 3) |
|  |  | 56 |  |  | 60 | Sub-list 3 (interval of 3) |
| 9 | 5 | 56 | 45 | 57 | 60 |  |
| 9 |  | 56 |  | 57 |  | Sub-list 1 (interval of 1) |
|  | 5 |  | 45 |  | 60 | Sub-list 2 (interval of 1) |
| 9 | 5 | 56 | 45 | 57 | 60 |  |
|  |  |  |  |  |  |  |
| 9 | 5 | 56 | 45 | 57 | 60 | Insertion Sort is used to sort the array |
| 5 | 9 | 56 | 45 | 57 | 60 |  |
| 5 | 9 | 56 | 45 | 57 | 60 |  |
| 5 | 9 | 45 | 56 | 57 | 60 |  |
| 5 | 9 | 45 | 56 | 57 | 60 |  |
| 5 | 9 | 45 | 56 | 57 | 60 |  |
| 5 | 9 | 45 | 56 | 57 | 60 | Sorted input elements |

Definition

* Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.
* This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **gap/interverl**.
* This gap/interval is calculated based on Knuth's formula as h = h \* 3 + 1

where − h is interval with initial value 1

* This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items.
* Shell Sort is a comparison based sorting.
* Time complexity of Shell Sort depends on gap sequence. Its best case time complexity is O(n\* logn) and worst case is O(n\* log2n). Time complexity of Shell sort is generally assumed to be near to O(n) and less than O(n2) as determining its time complexity is still an open problem.
* The best case in shell sort is when the array is already sorted. The number of comparisons is less.
* It is an in-place sorting algorithm as it requires no additional scratch space.
* Shell Sort is unstable sort as relative order of elements with equal values may change.
* It is been observed that shell sort is 5 times faster than bubble sort and twice faster than insertion sort its closest competitor.
* There are various increment sequences or gap sequences in shell sort which produce various complexity between O(n) and O(n2).