2019

Computational Thinking with Algorithms

Project: Benchmarking Sorting Algorithm

SOMANATHAN SUBRAMANIYAN
STUDENT ID: G00364742

Table of Contents

Introduction

Sorting and Sorting Algorithms

Sorting refers to arranging data elements in a particular format.

Sorting algorithm specifies the way to arrange data in a particular format. A Sorting algorithm puts elements of a list in a certain order. The most used orders are numerical and lexicographical order. Sorting algorithm provide a variety of core algorithm concepts such as

- Big O notation
- Divide and Conquer algorithms
- Data structures
- Best-case, Average-case and Worst-case analysis
- Time-space trade-offs

The commonly known Sorting algorithms are listed below

Bubble sort: Exchange two adjacent elements if they are out of order. Repeat until array is sorted.

Insertion sort: Scan successive elements for an out-of-order item, then insert the item in the proper place.

Selection sort: Find the smallest (or biggest) element in the array and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted.

Quick sort: Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively.

Merge sort: Divide the list of elements in two parts, sort the two parts individually and then merge it.

Key algorithm concepts

Big O notation

- It is used to describe the performance or complexity of an algorithm.
- It describes the worst-case scenario and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

O(1) describes an algorithm that will execute in the same time (or space) regardless of the size of the input data elements.

O(n) describes an algorithm performance that will grow linearly and in direct proportion to the size of the input data elements.

O(n^2) represents an algorithm performance that is directly proportional to the square of the size of the input data elements

O(log N)

Binary search is a technique used to sort input data elements. This works by selecting the middle element of the input data elements and compares it against a target value. If the values match it will return success

The iterative halving of data sets described in the binary search produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase.

Time and Space complexity

- Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
- Time and space complexity depend on lots of variables like
  - Hardware
  - Operating system
  - Processors, etc
- An algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation
- There are 3 common categories in time complexity
  - Linear - O (n)
  - Constant - O (1)
  - Quadratic - O (n^2)

Internal and External sorting

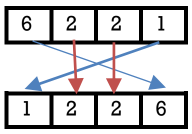
Internal or in-place Sorting: The input data elements are small enough to fit into the main memory. The algorithm does not require any extra space

External or Not-in-place Sorting: The input data elements are large and reside on the external storage. The algorithm requires space which is more than or equal to the elements being sorted.

Stable and Unstable sorting

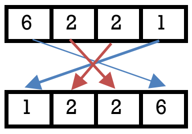
Sorting algorithm doesn't change the sequence of similar elements/data in the given list is called stable sorting.

Example:



Sorting algorithm changes the sequence of similar elements/data in the given list is called unstable sorting

Example:



Adaptive and non-adaptive sorting Algorithm

Adaptive Sorting: This algorithm takes advantage of already "sorted" elements in the given list that needs to be sorted

Non-adaptive: This type of algorithm does not take into account the elements which are already been sorted.

Comparison and non-comparison sorting

A comparison sort is a type of sorting algorithm that reads the input elements through a single comparison operation ("less than or equal to" or "great) and determines which of two elements should occur first in the sorted list.

Example of Comparison Sort:

- Quicksort
- Heapsort
- Shell sort
- Merge sort
- Bubble sort

The number of comparisons that a comparison sort algorithm requires increases in proportion to n log (n) where n is number of elements to be sorted in the given input list/array.

Worst case Scenario : O (n log (n)). It is proven that any comparison-based sorting algorithm will take at least O (n log (n))operations to sort n element.

Sorting algorithms that perform sorting without comparing the elements rather by making certain assumption about the input data elements they are going to sort. The process is known as non-comparison sorting and algorithms are known as the non-comparison-based sorting algorithms.

Non comparison sorting includes

- Counting sort which sorts using key value
- Radix sort, which examines individual bits of keys and
- Bucket Sort which examines bits of keys.

The above is also known as Liner sorting algorithms because they sort in O(n) time.

Speed/Time complexity

- Non-comparison sorting is usually faster because of not doing the comparison.
- The limit of speed for comparison-based algorithm is O(n log(n))
- The limit of speed for non-comparison-based algorithm is O(n) i.e. linear time.

Comparator

- Comparison based sorting algorithm requires a Comparator to sort elements
- Non-comparison-based sorting algorithms doesn't require any comparator.

Memory Complexity

- The best case for memory complexity with the comparison-based sorting is O(1).
- The memory complexity for non-comparison-based sorting algorithm is always O(n).

CPU Complexity

Comparison sort: The lower bound of CPU complexity in the worst case is O(n log(n))

Non-comparison sort: The lower bound of CPU complexity is O(n)i

Key examples (or) Uses of Sorting:

- Telephone directory: The telephone directory stores the telephone numbers of people sorted by their names so that the names can be searched easily
- Dictionary: The dictionary stores the words in the alphabetical order so that searching of any word becomes easy.

References

- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/
-

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

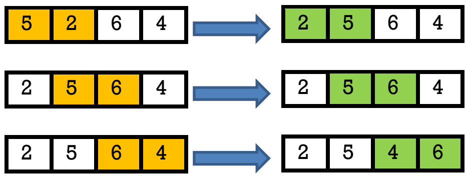
- Compare 2 consecutive pair of input elements
- Swap elements in pair such that the smaller is first
- When reach end of list, start over again
- Stop when no more swaps have been made
- Largest unsorted element always at end after pass, so at most "n" passes

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)
```

Implementation and Explanation

1st Pass:

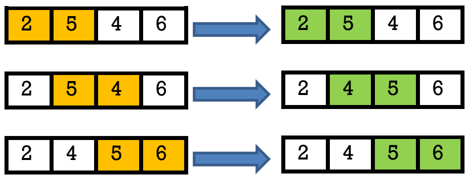


Compares the first 2 elements and swaps since 5 >2

No Swap as 5 < 6

Swap as 6 > 4

2nd Pass:



No Swap as 2 < 5

Swap as 5 >4

No Swap as 6 > 5

- 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
- 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 *$O(n^2)$*. 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 *$O(n^2)$*

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.

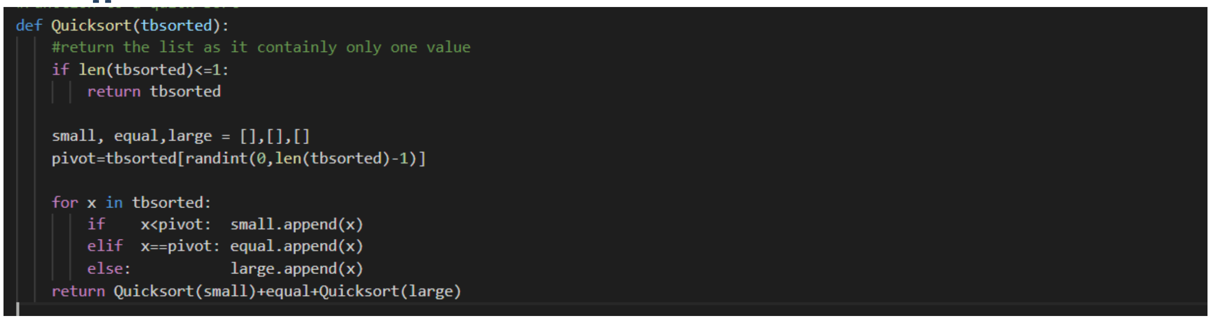
- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

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.

Code Snippet



```python
def Quicksort(tbsorted):
    #return the list as it containly only one value
    if len(tbsorted)<=1:
        return tbsorted

    small, equal,large = [],[],[]
    pivot=tbsorted[randint(0,len(tbsorted)-1)]

    for x in tbsorted:
        if    x<pivot:  small.append(x)
        elif  x==pivot: equal.append(x)
        else:           large.append(x)
    return Quicksort(small)+equal+Quicksort(large)
```

Implementation and Explanation.

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		

- 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.
- Time Complexity (Best Case): O (n log n)
- Time Complexity (Worst Case): O (n^2)
- Space Complexity: O(log n)

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

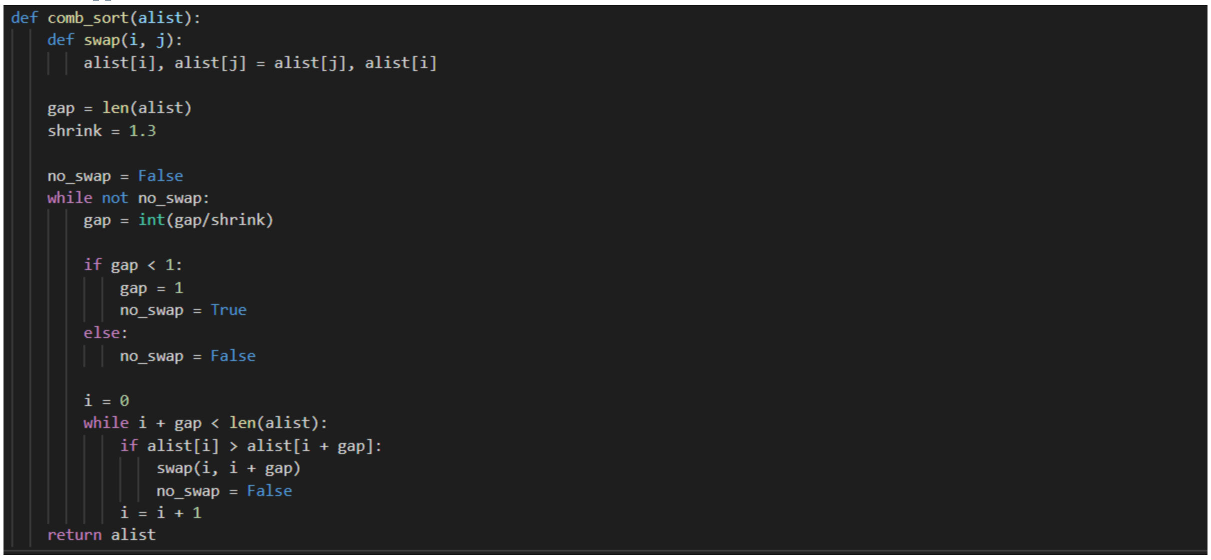
Comb sort iterates through the input elements multiple times, swapping elements that are out of order as it goes. It is similar to Bubble sort and the main difference is that comb sort looks at elements a certain number of indexes apart, this is called the gap. The input elements are sorted in a specific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

Code Snippet



```
def comb_sort(alist):
    def swap(i, j):
        alist[i], alist[j] = alist[j], alist[i]

    gap = len(alist)
    shrink = 1.3

    no_swap = False
    while not no_swap:
        gap = int(gap/shrink)

        if gap < 1:
            gap = 1
            no_swap = True
        else:
            no_swap = False

        i = 0
        while i + gap < len(alist):
            if alist[i] > alist[i + gap]:
                swap(i, i + gap)
                no_swap = False
            i = i + 1
    return alist
```

Implementation and Explanation.

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	

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

- Time Complexity (Worst Case): O (n^2)
- Space Complexity: O(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

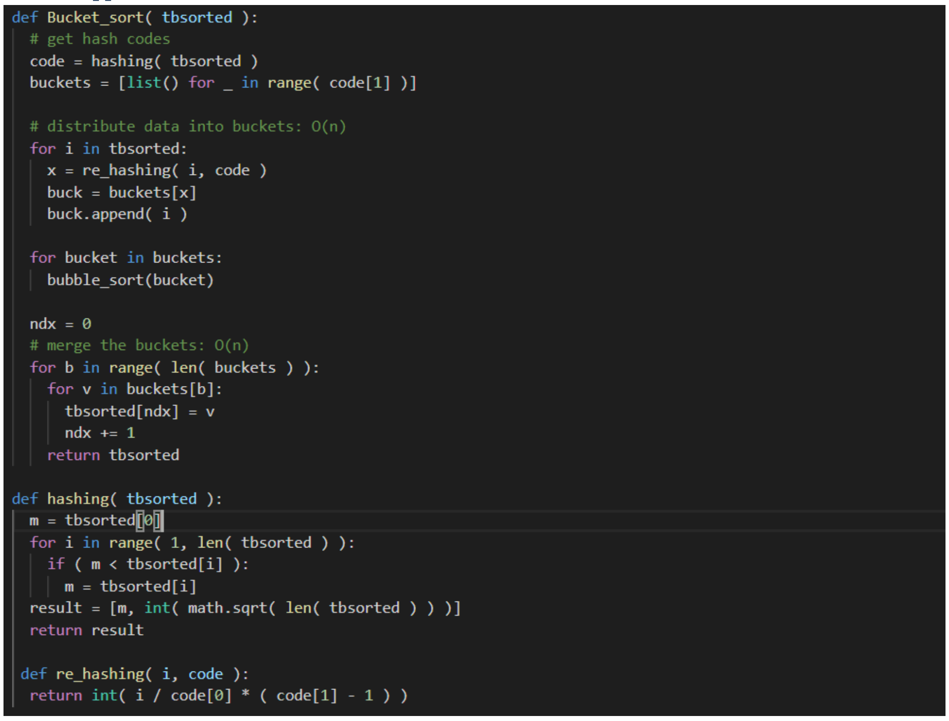
Definition

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.

Code Snippet



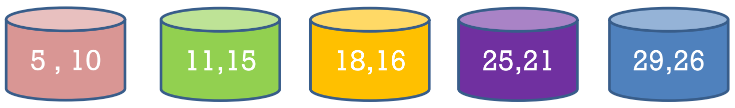
```python
def Bucket_sort( tbsorted ):
  # get hash codes
  code = hashing( tbsorted )
  buckets = [list() for _ in range( code[1] )]

  # distribute data into buckets: O(n)
  for i in tbsorted:
    x = re_hashing( i, code )
    buck = buckets[x]
    buck.append( i )

  for bucket in buckets:
    bubble_sort(bucket)

  ndx = 0
  # merge the buckets: O(n)
  for b in range( len( buckets ) ):
    for v in buckets[b]:
      tbsorted[ndx] = v
      ndx += 1
    return tbsorted

def hashing( tbsorted ):
  m = tbsorted[0]
  for i in range( 1, len( tbsorted ) ):
    if ( m < tbsorted[i] ):
      m = tbsorted[i]
  result = [m, int( math.sqrt( len( tbsorted ) ) )]
  return result

 def re_hashing( i, code ):
  return int( i / code[0] * ( code[1] - 1 ) )
```

Implementation and Explanation

Input Elements:

29	25	18	11	5	10	15	16	21	26

Buckets:



Sorted input elements:

5	10	11	15	16	18	21	25	26	29

- 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.
- Time Complexity (Best Case): O (n)
- Time Complexity (Worst Case): O (n)
- Space Complexity: O(1)

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

Shell Sort is a variation of Insertion Sort. If the input elements have to be moved far ahead, many movements are involved, and the Shell short allow exchange of far elements in the input list. A list of h-sorted for a large value of h and keep reducing the value of h until it becomes 1. A list is said to be sorted if all sub lists of every h'th element is sorted. The **shell sort**, sometimes called the "diminishing increment sort

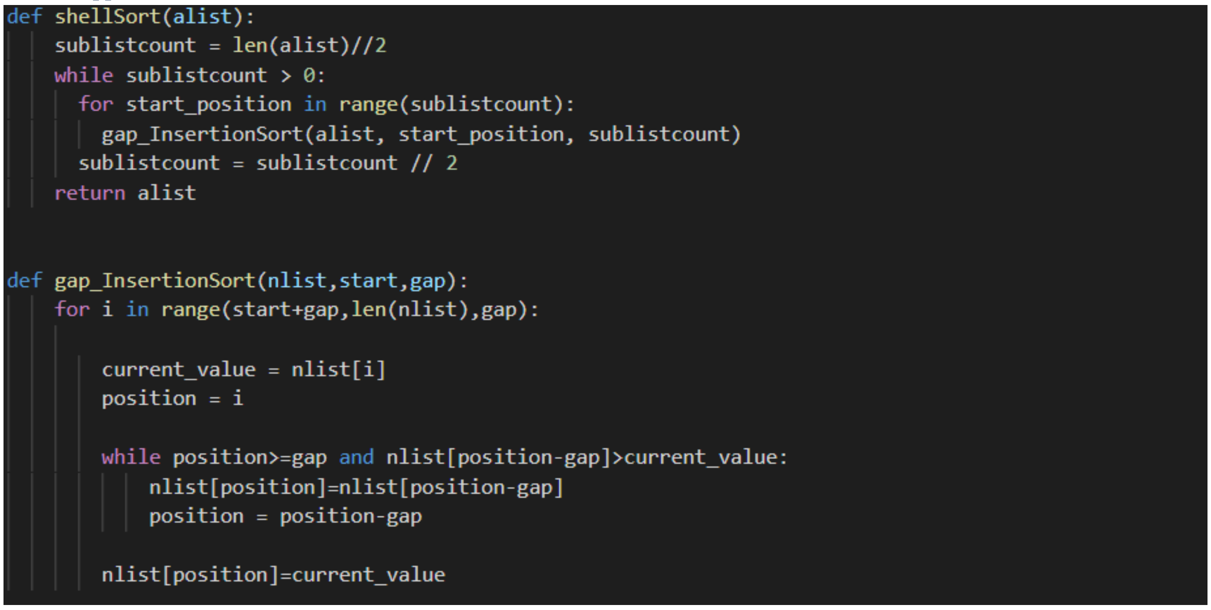
Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.

Logic

- This is an improvement of the insertion sort by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort.
- The way that these sub lists are chosen is the key to the shell sort.

- The input elements are broken into sub lists of non-contiguous items, usually an increment `i`, called the **gap**, to create a sub list by choosing all items that are `i` items apart.
- In the end, the insertion sort kicks-in, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Code Snippet



```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:
      for start_position in range(sublistcount):
        gap_InsertionSort(alist, start_position, sublistcount)
      sublistcount = sublistcount // 2
    return alist


def gap_InsertionSort(nlist,start,gap):
    for i in range(start+gap,len(nlist),gap):

        current_value = nlist[i]
        position = i

        while position>=gap and nlist[position-gap]>current_value:
            nlist[position]=nlist[position-gap]
            position = position-gap

        nlist[position]=current_value
```

Implementation and Explanation

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

- 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/interval.**

- 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 efficient for medium-sized data sets.
- Time complexity of Shell Sort depends on gap sequence.
    - Best-case time complexity is O(n*log(n))
    - worst case complexity is O(n* log2n).
    - Time complexity is assumed to be near to O(n) and less than $O(n^2)$
    - The best case is when the array is already sorted. The number of comparisons are less.
- It is an in-place sorting algorithm as it requires no additional scratch space.
- It is an 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 $O(n^2)$.

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

Benchmarking

Summary

As part of this project, the below 5 sorting algorithms are used for the analysis and benchmarking. The python implementation of the sorting algorithms is attached with this report.

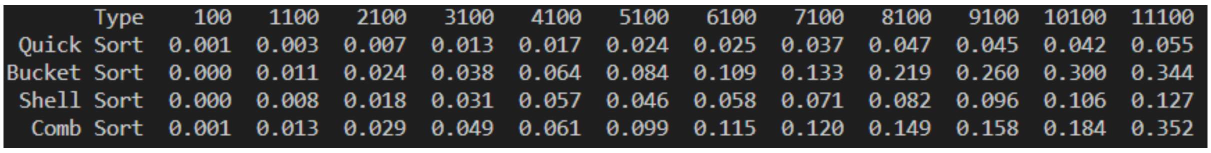
- Bubble Sort	[Comparison-based sorting algorithm]
- Quick Sort	[Comparison-based sorting algorithm]
- Bucket Sort	[Non-comparison-based sorting algorithm]
- Shell Sort	[Variation of comparison-based sorting algorithm]
- Comb Sort	[Variation of comparison-based sorting algorithm]

Key constructs used in the bench marking exercise:

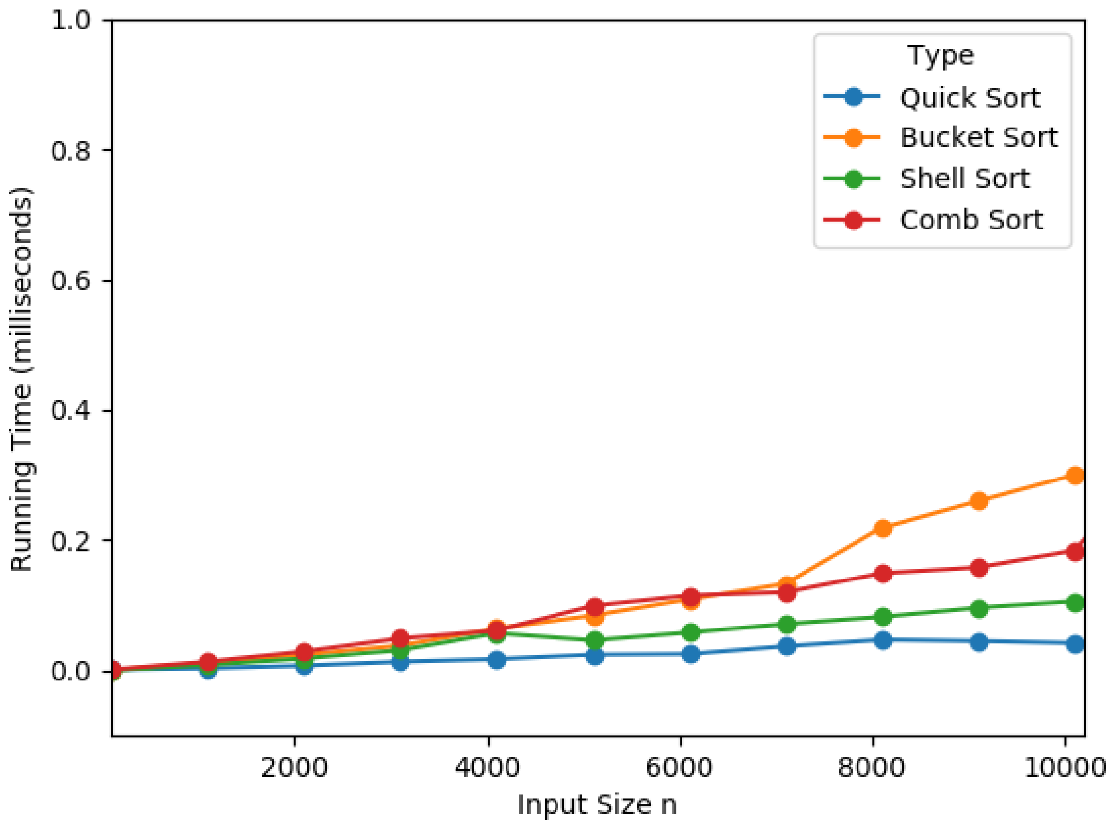
- Random numbers are generated using python function "randint"
- The integer numbers are generated between 999 and 999999
- Sample size are in the increments of 1000 with the starting value of 100 and final value of 12000
- For all the sorts except Bubble sort, the average of 30 repeated runs are used for the graph.
- For Bubble sort, the average of 10 repeated runs are used for the graph.

The running time (in milliseconds) for each sorting algorithm measured 30 times, and the average of the 30 runs for each algorithm and each input size n (100 to 11100 in the increments of 1000) shown below.

Bucket, Shell, Comb and Quick Sort - python program output and Graphs



```
       Type    100   1100   2100   3100   4100   5100   6100   7100   8100   9100  10100  11100
 Quick Sort  0.001  0.003  0.007  0.013  0.017  0.024  0.025  0.037  0.047  0.045  0.042  0.055
Bucket Sort  0.000  0.011  0.024  0.038  0.064  0.084  0.109  0.133  0.219  0.260  0.300  0.344
 Shell Sort  0.000  0.008  0.018  0.031  0.057  0.046  0.058  0.071  0.082  0.096  0.106  0.127
  Comb Sort  0.001  0.013  0.029  0.049  0.061  0.099  0.115  0.120  0.149  0.158  0.184  0.352
```



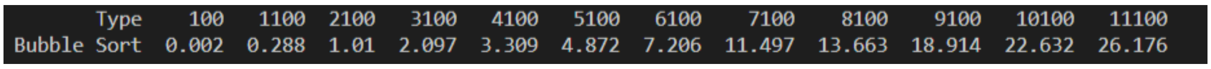
Time and Space Complexity:

Summary for the Time and space complexity is detailed below and the individual sorting algorithms are explained in the previous sections

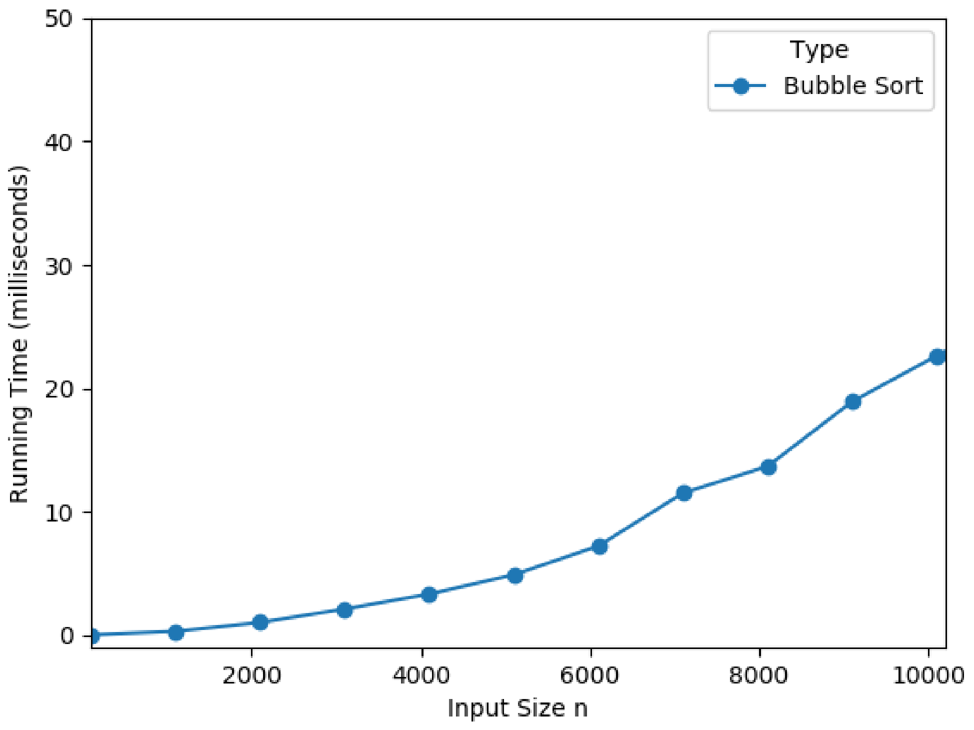
	Time Complexity			
	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$O(n \log n)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + k)$	$\Theta(n^2)$	$O(n)$
Comb Sort	$\Omega(n \log n)$	$\Theta(n^2/2^p)$	$\Theta(n^2)$	$O(1)$
Shell Sort	$\Omega(n \log n)$	Based on Gap sequence	$\Theta(n^2)$	$O(n)$

Bubble Sort – python program output and Graph

The running time (in milliseconds) for Bubble sorting algorithm measured 10 times, and the average of the 10 runs for each input size n (100 to 11100 in the increments of 1000) is shown below.



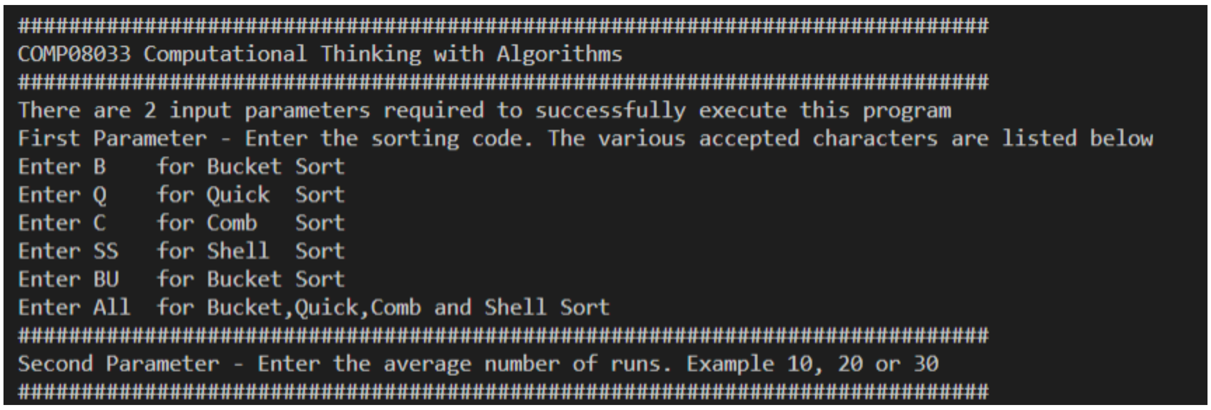
```
       Type    100   1100   2100   3100   4100   5100   6100   7100    8100    9100   10100   11100
Bubble Sort  0.002  0.288   1.01  2.097  3.309  4.872  7.206  11.497  13.663  18.914  22.632  26.176
```



The execution time of the sorting algorithms are in the following ascending order

- Quick Sort
    - It is able to sort the 11100 input elements in 0.055 milli seconds and better than all the sorts combined together
    - The execution time faster for the various input sizes as shown in the graph.
- Shell Sort
    - The execution time is 2nd best for the various input sizes as shown in the graph
- Comb Sort
    - The execution time is 3rd best for the various input sizes as shown in the graph except for the size 11100.
    - The Bucket Sort (non-comparison sort seems to perform better for the higher input sizes
- Bucket Sort
    - The execution time is 4th best for the various input sizes as shown in the graph except for the size 11100.
- Bubble Sort
    - The execution time is more for the various input sizes as shown in the graph due to the way this algorithm works.

Note: How to execute the python program



```
##############################################################################
COMP08033 Computational Thinking with Algorithms
##############################################################################
There are 2 input parameters required to successfully execute this program
First Parameter - Enter the sorting code. The various accepted characters are listed below
Enter B    for Bucket Sort
Enter Q    for Quick  Sort
Enter C    for Comb   Sort
Enter SS   for Shell  Sort
Enter BU   for Bucket Sort
Enter All  for Bucket,Quick,Comb and Shell Sort
##############################################################################
Second Parameter - Enter the average number of runs. Example 10, 20 or 30
##############################################################################
```

- Enter the Sorting code as detailed in the console
- Enter the average number of runs