National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

# Practice 8: Sorting Algorithms.

*Hernandez Martinez Carlos David.*

*davestring@outlook.com.*

*Group: 2cv3.*

November 17, 2017

# Contents

# 1 Introduction:

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

- The output is in nondecreasing order ( each element is no smaller than the previous element according to the desired total order ).
- The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big $\mathcal{O}$ notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space trade-offs, and lower bounds.

## 1.1 Classification:

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behaviour) of element comparisons in terms of the size of the list ( **n** ). For typical sorting algorithms good behavior is $\mathcal{O}(n \log n)$ and bad behavior is $\mathcal{O}(n^2)$. Ideal behavior for a sort is $\mathcal{O}(n)$, but this is not possible in the average case. Comparison-based sorting algorithms, which evaluate the elements of the list via an abstract key comparison operation, need at least $\mathcal{O}(n \log n)$ comparison for most inputs.
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only $\mathcal{O}(1)$ or $\mathcal{O}(n \log n)$ memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both ( e.g. merge sort ).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys ( i.e. values ).
- General method: insertion, exchange, selection, merging, etc.. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
- Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

## 1.2 Stability:

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

( 4, 2 ), ( 3, 7 ), ( 3, 1 ), ( 5, 6 ).

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

( 3, 7 ), ( 3, 1 ), ( 4, 2 ), ( 5, 6 ) -Order maintained-.

( 3, 1 ), ( 3, 7 ), ( 4, 2 ), ( 5, 6 ) -Order changed-.

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional computational cost. Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the keys need to be applied in order of increasing priority.

Example: sorting pairs of numbers as above by second, then first component:

( 4, 2 ), ( 3, 7 ), ( 3, 1 ), ( 5, 6 ) -Original-.

( 3, 1 ), ( 4, 2 ), ( 5, 6 ), ( 3, 7 ) -After sorting by second component-.

( 3, 1 ), ( 3, 7 ), ( 4, 2 ), ( 5, 6 ) -After sorting by first component-.

On the other hand:

( 3, 7 ), ( 3, 1 ), ( 4, 2 ), ( 5, 6 ) -After sorting by first component-.

( 3, 1 ), ( 4, 2 ), ( 5, 6 ), ( 3, 7 ) -After sorting by second component-.

# 2 Basic Concepts:

The following 3 algorithms are the ones that we are going to analyze in the following sections.

## 2.1 Heap-Sort:

*Heapsort* is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes $\mathcal{O}(\log n)$ time, instead of $\mathcal{O}(n)$ for a linear scan as in simple selection sort. This allows Heapsort to run in $\mathcal{O}(n \log n)$ time, and this is also the worst case complexity.

## 2.2 Shell-Sort:

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

## 2.3 Cocktail-Sort:

Each iteration of the algorithm is broken up into two stages:

- The first stage loops through the data set from bottom to top, just like the Bubble Sort. During the loop, adjacent items are compared. If at any point the value on the left is greater than the value on the right, the items are swapped. At the end of the first iteration, the largest number will reside at the end of the set.
- The second stage loops through the data set in the opposite direction - starting from the item just before the most recently sorted item, and moving back towards the start of the list. Again, adjacent items are swapped if required.

The Cocktail Sort also fits in the category of Exchange Sorts due to the manner in which elements are moved inside the data set during the sorting process.

## 2.4 Theoretical Complexity:

The algorithms that we are going to analyze have different running time in different cases that the programmer submit it. For this 3 algorithms its best case it's when the list it's already sorted, and the worst case it's when the list it's sorted but in decreasing order. The Table 0 will indicate the complexity of each algorithm:

| Algorithm | Best Case | Random Case | Worst Case |
|---|---|---|---|
| Heap-Sort | $\mathcal{O}(n \log(n))$ | $\mathcal{O}(n \log(n))$ | $\mathcal{O}(n \log(n))$ |
| Cocktail-Sort | $\mathcal{O}(n))$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |
| Shell-Sort | $\mathcal{O}(n)$ | $\mathcal{O}(n^{\frac{3}{2}})$ or $\mathcal{O}(n \log^2(n))$ | $\mathcal{O}(n^{\frac{3}{2}})$ or $\mathcal{O}(n \log^2(n))$ |

Table 0.

# 3 Development:

In this section we will implement the algorithms mentioned in Section 2.

## 3.1 Heap-Sort Algorithm:

For the following algorithm we create a *Class* with the name of the algorithm, in its constructor will receive as parameters only the list to sort, this list has the name of *dimensions* as we can see in line 3, in line 4 the program corroborates that the length of *dimensions* it's bigger than 1, in case that do not accomplish the condition then the program stops running because there is no need of sorting, otherwise the program continues running and in line 6 stores in a instance of the class the length of *dimensions*, this two variables are going to be used along the execution.

Heap sort is a comparison sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining element. In line 9 it's written the algorithm that will *heapify* the root of the tree, this algorithm assumes that the binaries trees rooted at LEFT ( j ) and RIGHT ( j ) are max heaps, but *dimensions [ j ]* might be smaller than its children, thus violating the max-heap property. Lets the value at *dimensions [ j ]* 'float down' in the max-heap so that the sub-tree rooted at index *j* obeys the max-heap property.

```
1  class Heapsort:
2      # Class constructor.
3      def __init__ ( self, dimensions ):
4          assert len ( dimensions ) > 1
5          self.dimensions = dimensions
6          self.n = len ( dimensions )
7
8      # Heapify subtree rooted at index j.
9      def heapify ( self, i, j ):
10         right = 2 * j + 2
11         left = 2 * j + 1
12         largest = j
13         # Verify if the left child exist and if it's greater than the root.
14         if ( left < i and self.dimensions [ j ] < self.dimensions [ left ] ):
15             largest = left
16         # Verify if the right child exist and if it's greater than the root.
17         if ( right < i and self.dimensions [ largest ] < self.dimensions [ right ] ):
18             largest = right
19         # Change the root if it's needed.
20         if ( largest != j ):
21             aux = self.dimensions [ j ]
22             self.dimensions [ j ] = self.dimensions [ largest ]
23             self.dimensions [ largest ] = aux
24             # Heapify the root.
25             self.heapify ( i, largest )
26
27     # Heapsort algorithm.
28     def heapsort ( self ):
29         # Build the max heap.
30         for i in range ( self.n, -1, -1 ):
31             self.heapify ( self.n, i )
32         # Extract the elements one by one.
33         for i in range ( ( self.n - 1 ), 0, -1 ):
34             aux = self.dimensions [ i ]
35             self.dimensions [ i ] = self.dimensions [ 0 ]
36             self.dimensions [ 0 ] = aux
37             self.heapify ( i, 0 )
```

Figure 3.1.0 illustrates the action of the method *heapify*. At each step, the largest of the elements *dimensions [ j ]*, *dimensions [ left ]* and *dimensions [ right ]* is determined, and its index is stored in largest as we can see in lines 14 - 18. If *dimensions [ j ]* is largest, then the sub-tree rooted at node *j* is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and *dimensions [ j ]* is

swapped with *dimensions [ largest ]* as we can see from lines 20 - 23, which causes node $j$ and its children to satisfy the max-heap property. The node indexed by largest, however, now has the original value *dimensions [ j ]*, and thus the sub-tree rooted at largest might violate the max-heap property. Consequently, in line 25 we will call recursively *heapify* on that sub-tree.

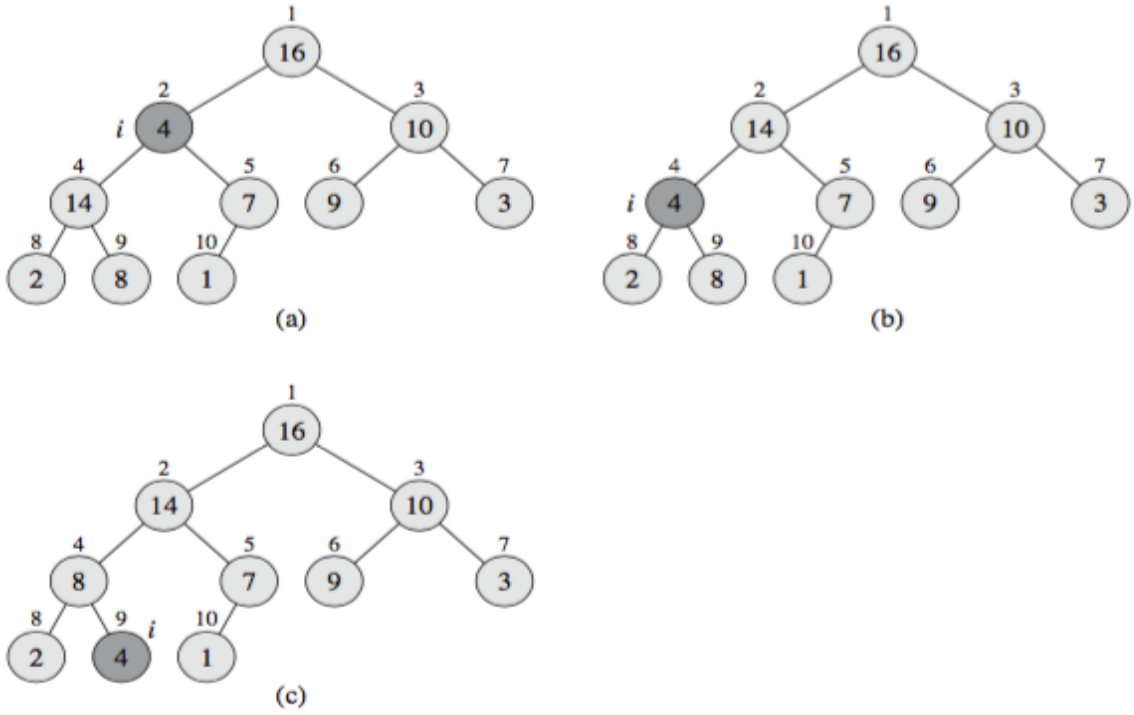


Figure 3.1.0: The action of *heapify ( 10, 2 )*, where 10 it's the heap size. **(a)** The initial configuration, with *dimensions [ 2 ]* at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging *dimensions [ 2 ]* with *dimensions [ 4 ]*, which destroys the max-heap property for node 4. The recursive call *heapify ( 10, 4 )* now has $i = 4$. After swapping *dimensions [ 4 ]* with *dimensions [ 9 ]*, as shown in **(c)**, node 4 is fixed up, and the recursive call *heapify ( 10, 9 )* yields no further change to the data structure.

In lines 30 - 31 the algorithm will build the *max-heap* by using the algorithm *heapify* as we can see in line 31. The procedure to build a *max-heap* goes through the remaining nodes of the tree and runs *heapify* on each one. Figure 3.1.1 shows an example of building a *max-heap*.

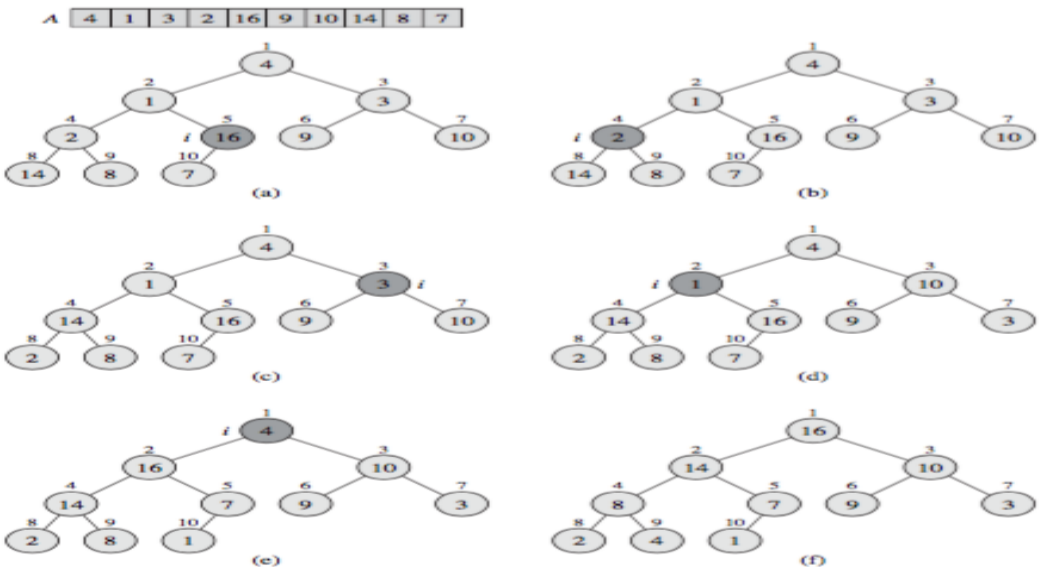


Figure 3.1.1: The operation of building a *max-heap*, showing the data structure before the call to *heapify* in line 31. **(a)** A 10-element input array **A** and the binary tree it represents. The figure shows that the loop index $i$ refers to node 5 before the call *heapify ( 10, i )*. **(b)** The data structure that results. The loop index $i$ for the next iteration refers to node 4. **(c)-(e)** Subsequent iterations of the **loop** in line 30. Observe that whenever *heapify* is called on a node, the two sub-trees of that node are both *max-heap*. **(f)** Building the max heap process it's finished.

After building the *max-heap* the 'really' *Heap-Sort* algorithm begins. Since the maximum element of the array is stored at the root *dimensions [ 0 ]*, we can put it into its correct final position by exchanging it with *dimensions [ n ]*. If we now discard node **n** from the heap and we can do so by simply decreasing the *heap-size* we observe that the children of the root remain *max-heaps*, but the new root element might violate the *max-heap* property. All we need to do to restore the *max-heap* property, however, is call *heapify ( i, 0 )* as we can see in line 37 which leaves a *max-heap* in *dimensions [ 1 ... n - 1 ]*. The *Heap-Sort* algorithm then repeats this process for the *max-heap* of size *n - 1* down to a heap of size *1*.

Figure 3.1.2 shows an example of the operation of this algorithm after building the *max-heap*.

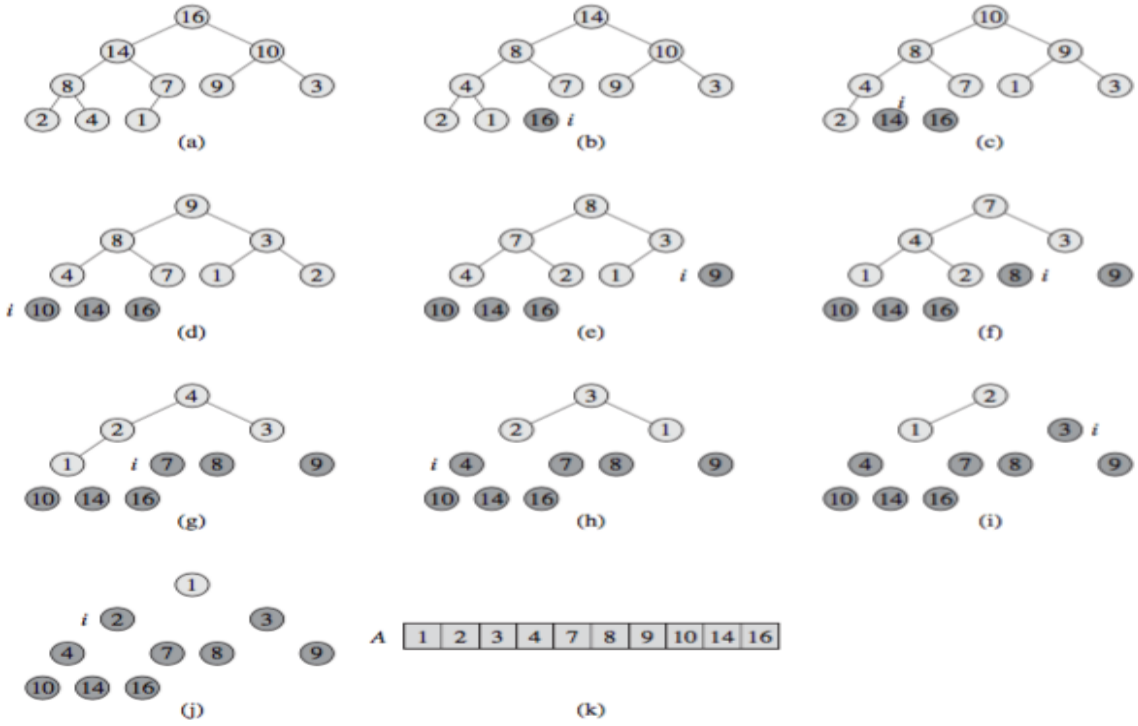


Figure 3.1.2: Operations of *Heap-Sort* algorithm. **(a)** *Max-Heap* data structure after lines 30 - 31. **(b)-(j)** The *max-heap* just after each call of *heapify* in line 37, showing the value of $i$ at that time. Only lightly shaded nodes remain in the heap. **(k)** Resulting sorted list.

## 3.2 Cocktail-Sort Algorithm:

For the following algorithm we create a *Class* with the name of the algorithm, in its constructor will receive as parameters the list to sort, this list has the name of *dimensions* and the start/ end indexes as we can see in line 3, in line 4 the program corroborates that the length of *dimensions* it's bigger than 1, in case that do not accomplish the condition then the program stops running because there is no need of sorting, otherwise the program continues running and in line 6 stores in a instance of the class the length of *dimensions*, this four variables are going to be used along the execution.

Cocktail Sort is a variation of Bubble sort. The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in first iteration and second largest in second iteration and so on. Cocktail Sort traverses through a given array in both directions alternatively. Each iteration of the algorithm is broken up into 2 stages:

1. The first stage loops through the array from left to right, just like the Bubble Sort as we can see in lines 17 - 22. During the loop, adjacent items are compared and if value on the left is greater than the value on the right, then values are swapped. At the end of first iteration, largest number will reside at the end of the array.

2. The second stage loops through the array in opposite direction starting from the item just before the most recently sorted item as we can see in lines 34 - 39, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

During each iteration the *start* and *end* indexes decrease and increase respectively, this because during the sorting process the elements stored on this indexes are already sorted.

```
1  class Cocktail:
2      # Class constructor.
3      def __init__ ( self, dimensions, start, end ):
4          assert len ( dimensions ) > 1
5          self.dimensions = dimensions
6          self.n = len ( dimensions )
7          self.swapped = True
8          self.start = start
9          self.end = end
10
11     def cocktail ( self ):
12         while ( self.swapped ):
13             # Reset the swapped flag because it might be true in the next iteration.
14             self.swapped = False
15
16             # Loop from left to right.
17             for i in range ( self.start, self.end ):
18                 if ( self.dimensions [ i ] > self.dimensions [ i + 1 ] ):
19                     aux = self.dimensions [ i ]
20                     self.dimensions [ i ] = self.dimensions [ i + 1 ]
21                     self.dimensions [ i + 1 ] = aux
22                     self.swapped = True
23
24             # If nothing were moved then the array it's sorted.
25             if ( self.swapped == False ):
26                 break
27             # Otherwise, reset the flag so can be used in the next stage.
28             self.swapped = False
29             # Move the end back in one unit, because the previous loop
30             # moved the greater number to its rightfull spot.
31             self.end = self.end - 1
32
33             # Loop from right to left.
34             for i in range ( self.end - 1, self.start - 1, -1 ):
35                 if ( self.dimensions [ i ] > self.dimensions [ i + 1 ] ):
36                     aux = self.dimensions [ i ]
37                     self.dimensions [ i ] = self.dimensions [ i + 1 ]
38                     self.dimensions [ i + 1 ] = aux
39                     self.swapped = True
40
41             # Increase the starting point, because the previous loop
42             # moved the next smallest number to its rightfull spot.
43             self.start = self.start + 1
```

The sorting works as follows: Consider the following list *dimensions = [ 5, 1, 4, 2, 8, 0, 2 ]*. From lines 17 - 22 will make the forward pass.

**First Forward Pass:**

[ **5 1** 4 2 8 0 2 ] ⇒ [ **1 5** 4 2 8 0 2 ] -Swap since 5 > 1-.
[ 1 **5 4** 2 8 0 2 ] ⇒ [ 1 **4 5** 2 8 0 2 ] -Swap since 5 > 4-.
[ 1 4 **5 2** 8 0 2 ] ⇒ [ 1 4 **2 5** 8 0 2 ] -Swap since 5 > 2-.
[ 1 4 2 **5 8** 0 2 ] ⇒ [ 1 4 2 **5 8** 0 2 ]
[ 1 4 2 5 **8 0** 2 ] ⇒ [ 1 4 2 5 **0 8** 2 ] -Swap since 8 > 0-.
[ 1 4 2 5 0 **8 2** ] ⇒ [ 1 4 2 5 0 **2 8** ] -Swap since 8 > 2-.

After first forward pass, greatest element of the array will be present at the last index of array. Now, from lines 34 - 39 the program will make the backward pass.

**First Backward Pass:**

[ 1 4 2 5 **0** **2** 8 ] ⇒ [ 1 4 2 5 **0** **2** 8 ]
[ 1 4 2 **5** **0** 2 8 ] ⇒ [ 1 4 2 **0** **5** 2 8 ] -Swap since 5 > 0-.
[ 1 4 **2** **0** 5 2 8 ] ⇒ [ 1 4 **0** **2** 5 2 8 ] -Swap since 3 > 0-.
[ 1 **4** **0** 2 5 2 8 ] ⇒ [ 1 **0** **4** 2 5 2 8 ] -Swap since 4 > 0-.
[ **1** **0** 4 2 5 2 8 ] ⇒ [ **0** **1** 4 2 5 2 8 ] -Swap since 1 > 0-.

After first backward pass, smallest element of the array will be present at the first index of the array. Then. the process it's repeated.

**Second Forward Pass:**

[ 0 **1** **4** 2 5 2 8 ] ⇒ [ 0 **1** **4** 2 5 2 8 ]
[ 0 1 **4** **2** 5 2 8 ] ⇒ [ 0 1 **2** **4** 5 2 8 ] -Swap since 4 > 2-.
[ 0 1 2 **4** **5** 2 8 ] ⇒ [ 0 1 2 **4** **5** 2 8 ]
[ 0 1 2 4 **5** **2** 8 ] ⇒ [ 0 1 2 4 **2** **5** 8 ] -Swap since 5 > 2-.

Finally:

**Second Backward Pass:**

[ 0 1 2 **4** **2** 5 8 ] ⇒ [ 0 1 2 **2** **4** 5 8 -Swap since 4 > 2-.

## 3.3 Shell-Sort Algorithm:

For the following algorithm we create a *Class* with the name of the algorithm, in its constructor will receive as parameters only the list to sort, this list has the name of *dimensions* as we can see in line 3, in line 4 the program corroborates that the length of *dimensions* it's bigger than 1, in case that do not accomplish the condition then the program stops running because there is no need of sorting, otherwise the program continues running and in line 6 and 7 stores in a instance of the class the length of *dimensions* and the *gap* value, that is the length divided by 2, this three variables are going to be used along the execution.

The shell sort, sometimes called the "diminishing increment sort", improves on 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 unique way that these sub-lists are chosen is the key to the shell sort. Instead of breaking the list into sub-lists of contiguous items, the shell sort uses an increment $i$ sometimes called the **gap** to create a sub-list by choosing all items that are $i$ item apart.

This can be seen in Figure 3.3.0. This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure 3.3.1. Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.


Figure 3.3.0: Initial sub-lists.

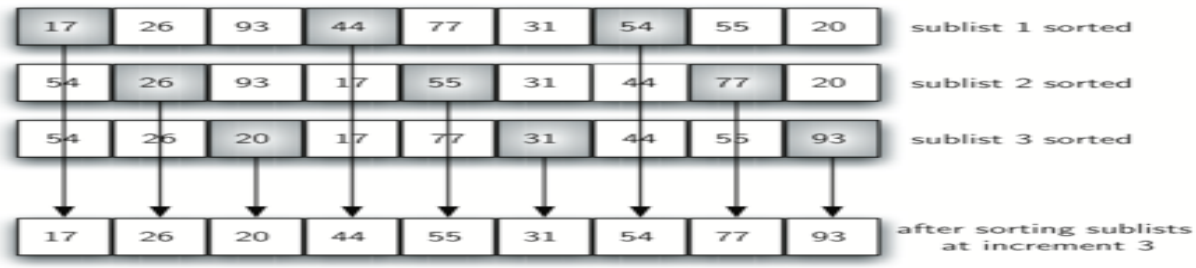


Figure 3.3.1: Sorting sub-lists.

Figure 3.3.2 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

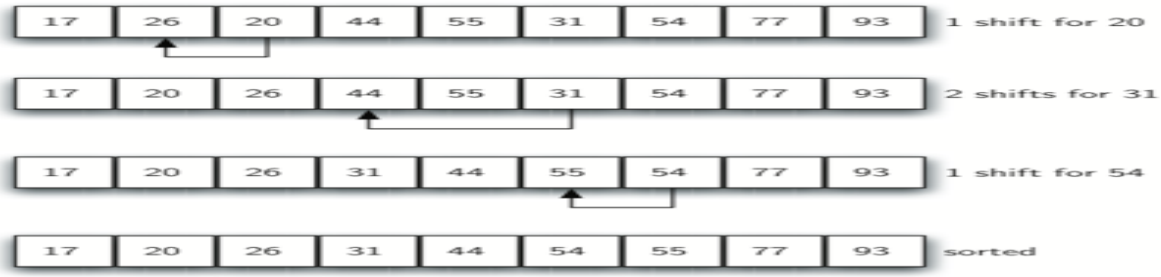


Figure 3.3.2: Swapping elements.

The previous procedure can be implemented in the code below:

```
class Shell:
    # Class constructor.
    def __init__ ( self, dimensions ):
        assert len ( dimensions ) > 1
        self.dimensions = dimensions
        self.n = len ( dimensions )
        self.gap = int ( self.n / 2 )

    # Shellsort algorithm.
    def shellsort ( self ):
        # Do a gapped insertion sort for the gap size.
        # The first gap elements of dimensions [ 0 .. gap - 1 ] are already in
        # gapped order then, keep adding one more element until the entire array
        # is gap sorted.
        while ( self.gap > 0 ):
            # Add dimensions [ i ] to the elements that have been gap sorted.
            for i in range ( self.gap, self.n ):
                # Save dimensions [ i ] in the variable 'temp', and make a hole
                # at i position.
                temp = self.dimensions [ i ]
                # Shift earlier gap-sorted elements up until the correct location
                # for dimensions [ i ] is found.
                j = i
                while ( j >= self.gap and self.dimensions [ j - self.gap ] > temp ):
                    self.dimensions [ j ] = self.dimensions [ j - self.gap ]
                    j = j - self.gap
                # Put 'temp' ( the original dimensions [ i ] ) in its correct location.
                self.dimensions [ j ] = temp
            self.gap = int ( self.gap / 2 )
```

# 4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the ***console*** output of the algorithms previously mentioned, also, we will plot the complexity of each one in its best, random and worst case.

## 4.1 Heap-Sort Algorithm:

First test of **Heap-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 10$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 1

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


        Random case:

        List to sort: [6, 4, 8, 4, 1, 8, 1, 1, 3, 6]

        Sorted List:  [1, 1, 1, 3, 4, 4, 6, 6, 8, 8]


        Worst case:

        List to sort: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure 4.1.0: Heap-Sort console output.

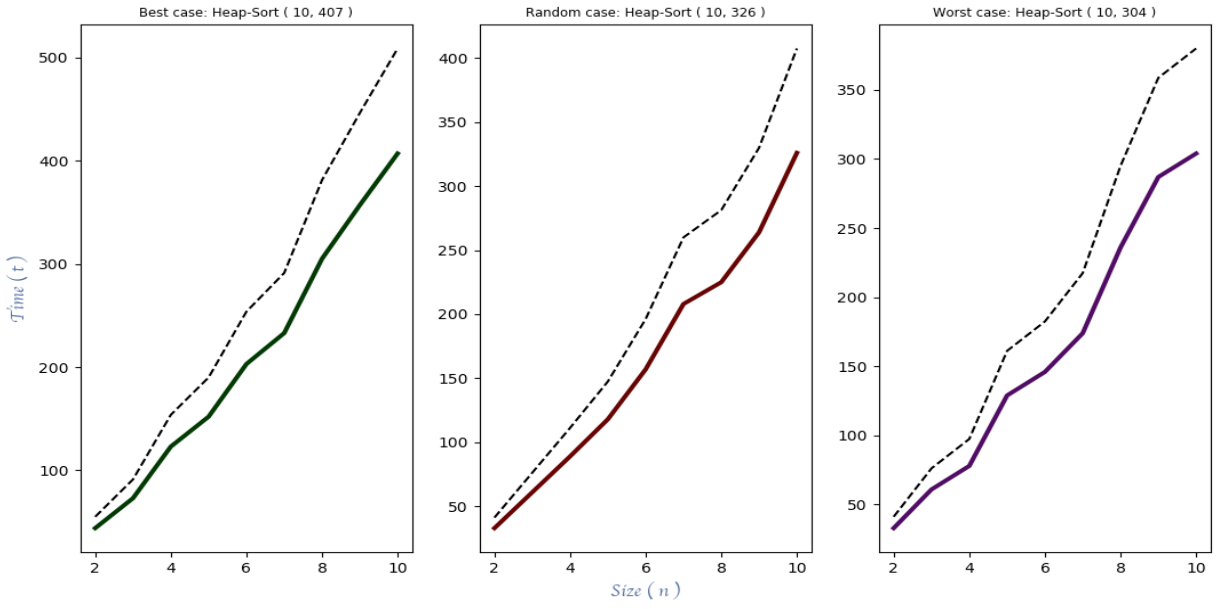


Figure 4.1.1: Heap-Sort graph for the lists in Figure 4.1.0.

Table 1 illustrate the plotting points of Figure 4.1.1 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 44 | 33 | 33 |
| 3 | 73 | 61 | 61 |
| 4 | 123 | 89 | 78 |
| 5 | 152 | 118 | 129 |
| 6 | 203 | 157 | 146 |
| 7 | 233 | 208 | 174 |
| 8 | 305 | 225 | 236 |
| 9 | 357 | 264 | 287 |
| 10 | 407 | 326 | 304 |

Table 1.

***Observation:*** *In Figure 4.1.1 there are 3 graphs, the one on the left represent the algorithm's best case, analogously the middle and right ones represents the random and worst case, in each graph, there is a* ***pointed*** *stroke, this it's an asymptotic function for the main plot.*

Second test of **Heap-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 20$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 1

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]


        Random case:

        List to sort: [6, 9, 2, 4, 17, 12, 19, 18, 10, 19, 11, 3, 11, 6, 9, 7, 11, 12, 11, 2]

        Sorted List:  [2, 2, 3, 4, 6, 6, 7, 9, 9, 10, 11, 11, 11, 11, 12, 12, 17, 18, 19, 19]


        Worst case:

        List to sort: [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 4.1.2: Heap-Sort console output.

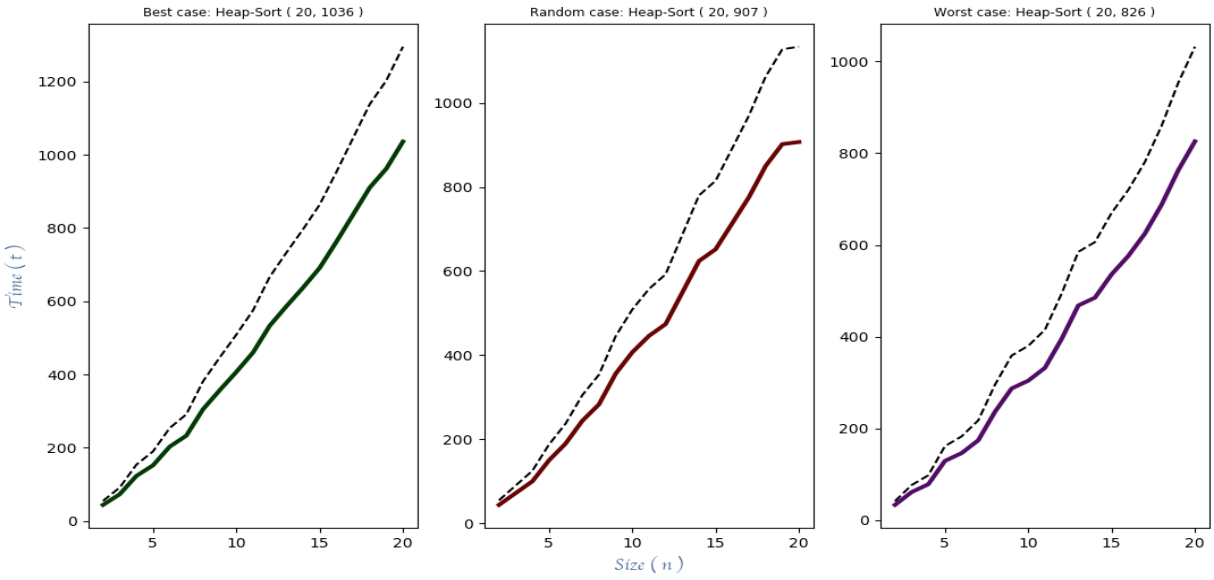


Figure 4.1.3: Heap-Sort graph for the lists in Figure 4.1.2.

Table 2 illustrate the plotting points of Figure 4.1.3 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 44 | 44 | 33 |
| 3 | 73 | 72 | 61 |
| 4 | 123 | 100 | 78 |
| 5 | 152 | 150 | 129 |
| 6 | 203 | 190 | 146 |
| 7 | 233 | 244 | 174 |
| 8 | 305 | 283 | 236 |
| 9 | 357 | 356 | 287 |
| 10 | 407 | 407 | 304 |
| 11 | 460 | 446 | 332 |
| 12 | 533 | 474 | 395 |
| 13 | 586 | 549 | 468 |
| 14 | 637 | 624 | 485 |
| 15 | 691 | 652 | 536 |
| 16 | 762 | 714 | 576 |
| 17 | 836 | 776 | 625 |
| 18 | 910 | 850 | 688 |
| 19 | 962 | 902 | 763 |
| 20 | 1036 | 907 | 826 |

Table 2.

Our last test for this algorithm consist in plotting the complexity of lists of sizes $n = 500$ and $n = 1000$ as we can see in Figures 4.1.4 and 4.1.5 respectively. In this examples we can visualize clearly that the algorithm in its best, random or worst case always will be $\mathcal{O}(n \log n)$ time.

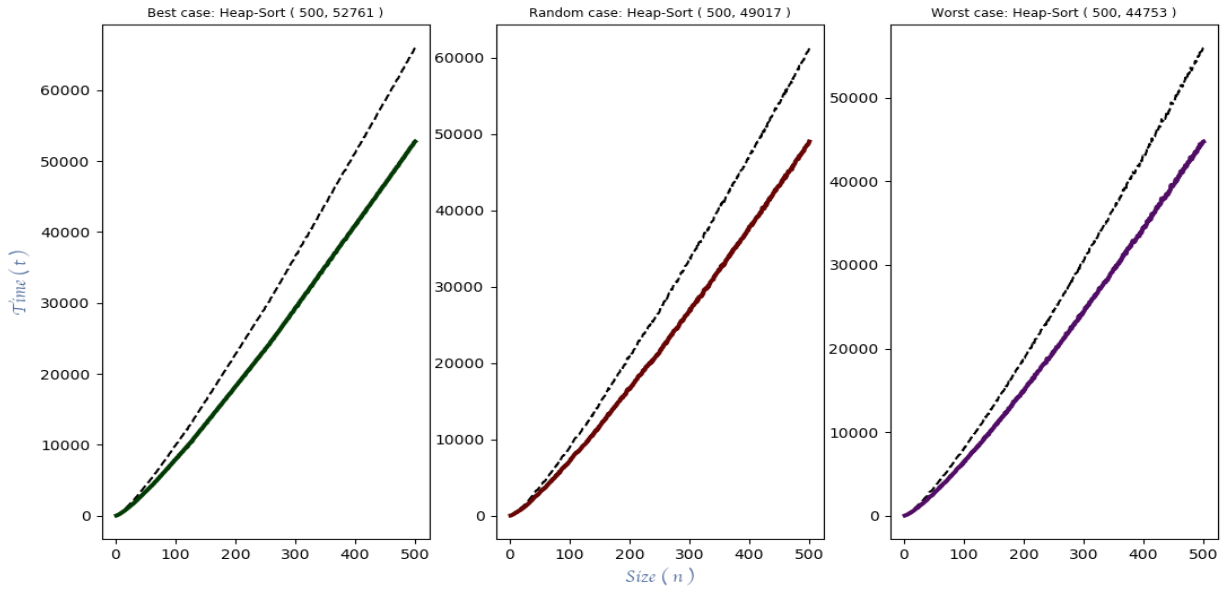


Figure 4.1.4: Heap-Sort complexity for lists of size n = 500.

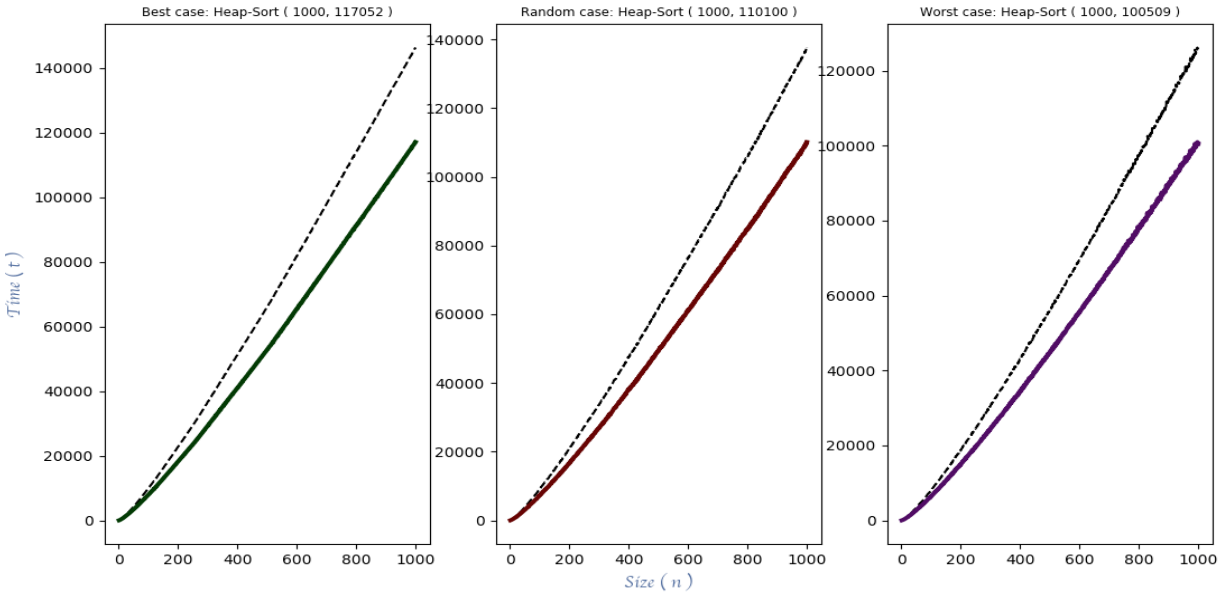


Figure 4.1.5: Heap-Sort complexity for lists of size n = 1000.

## 4.2 Cocktail-Sort Algorithm:

First test of **Cocktail-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 10$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 2

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


        Random case:

        List to sort: [2, 3, 1, 3, 7, 2, 1, 7, 9, 5]

        Sorted List:  [1, 1, 2, 2, 3, 3, 5, 7, 7, 9]


        Worst case:

        List to sort: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure 4.2.0: Cocktail-Sort console output.

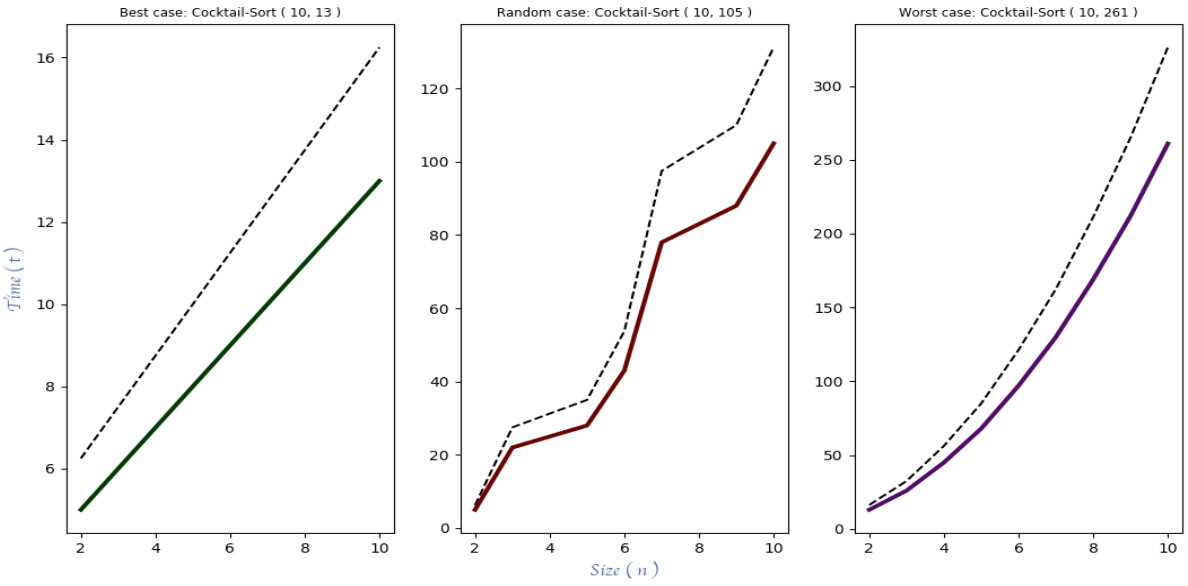


Figure 4.2.1: Cocktail-Sort graph for the lists in Figure 4.2.0.

Table 3 illustrate the plotting points of Figure 4.2.1 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 5 | 5 | 13 |
| 3 | 6 | 22 | 26 |
| 4 | 7 | 25 | 45 |
| 5 | 8 | 28 | 68 |
| 6 | 9 | 43 | 97 |
| 7 | 10 | 78 | 130 |
| 8 | 11 | 83 | 169 |
| 9 | 12 | 88 | 212 |
| 10 | 13 | 105 | 261 |

Table 3.

***Observation:*** *In Figure 4.2.1 there are 3 graphs, the one on the left represent the algorithm's best case, analogously the middle and right ones represents the random and worst case, in each graph, there is a* ***pointed*** *stroke, this it's an asymptotic function for the main plot.*

Second test of **Cocktail-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 20$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 2

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]


        Random case:

        List to sort: [3, 17, 9, 13, 0, 9, 7, 16, 16, 15, 3, 0, 20, 3, 1, 13, 4, 9, 20, 17]

        Sorted List:  [0, 0, 1, 3, 3, 3, 4, 7, 9, 9, 9, 13, 13, 15, 16, 16, 17, 17, 20, 20]


        Worst case:

        List to sort: [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 4.2.2: Cocktail-Sort console output.

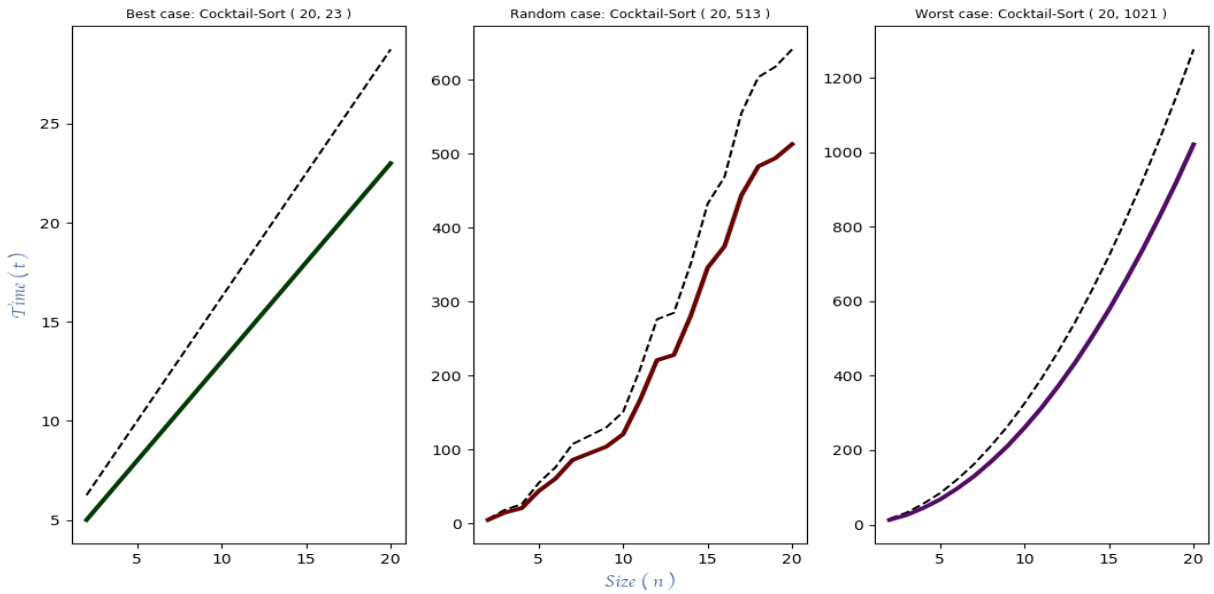


Figure 4.2.3: Cocktail-Sort graph for the lists in Figure 4.2.2.

Table 4 illustrate the plotting points of Figure 4.2.3 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 5 | 5 | 13 |
| 3 | 6 | 15 | 26 |
| 4 | 7 | 21 | 45 |
| 5 | 8 | 44 | 68 |
| 6 | 9 | 61 | 97 |
| 7 | 10 | 86 | 130 |
| 8 | 11 | 95 | 169 |
| 9 | 12 | 104 | 212 |
| 10 | 13 | 121 | 261 |
| 11 | 14 | 167 | 314 |
| 12 | 15 | 221 | 373 |
| 13 | 16 | 228 | 436 |
| 14 | 17 | 281 | 505 |
| 15 | 18 | 346 | 578 |
| 16 | 19 | 375 | 657 |
| 17 | 20 | 444 | 740 |
| 18 | 21 | 483 | 829 |
| 19 | 22 | 494 | 922 |
| 20 | 23 | 513 | 1021 |

Table 4.

Our last test for this algorithm consist in plotting the complexity of lists of sizes $n = 500$ and $n = 1000$ as we can see in Figures 4.2.4 and 4.2.5 respectively. In this examples we can visualize clearly that the algorithm in its best case will be $\mathcal{O}(n)$, otherwise $\mathcal{O}(n^2)$ time.

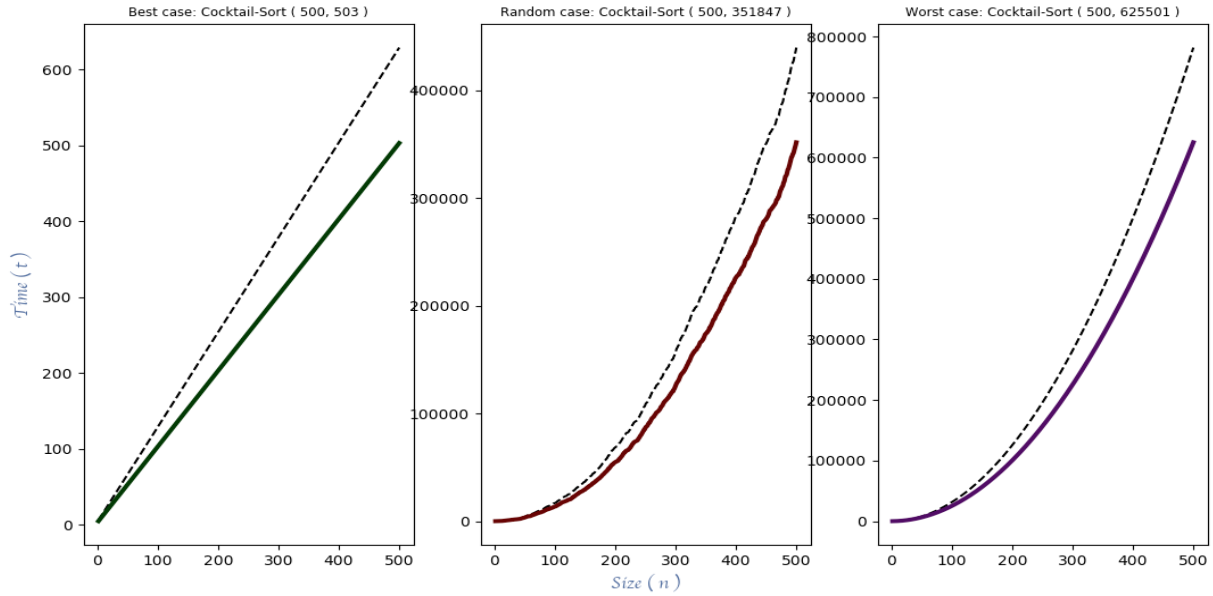


Figure 4.2.4: Cocktail-Sort complexity for lists of size n = 500.

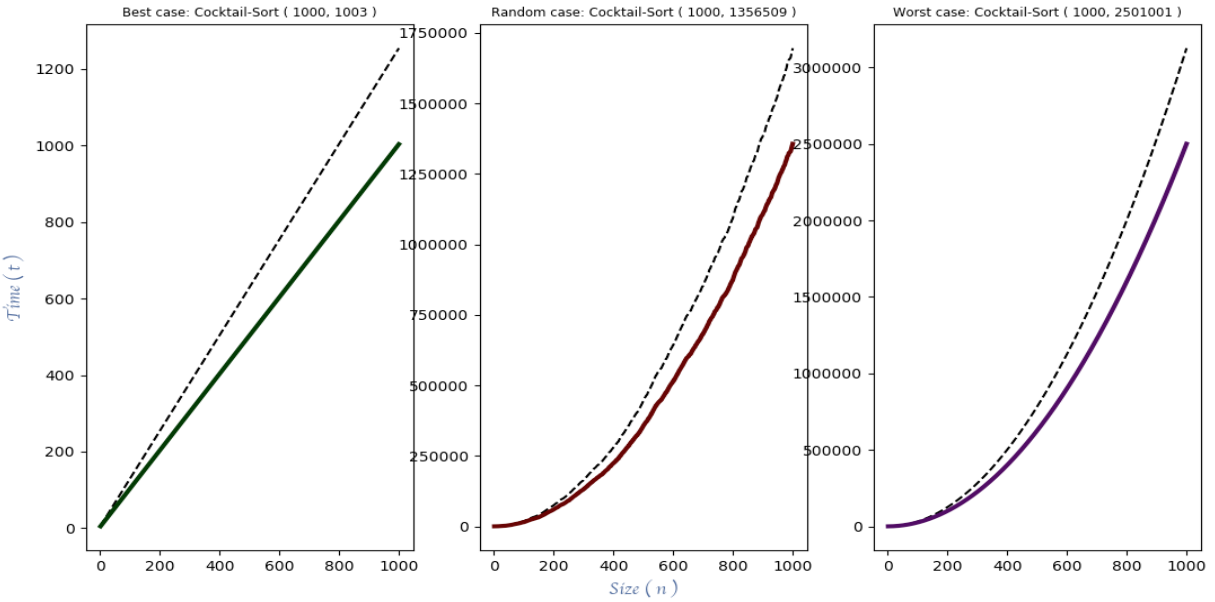


Figure 4.2.5: Cocktail-Sort complexity for lists of size n = 1000.

## 4.3 Shell-Sort Algorithm:

First test of **Shell-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 10$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 3

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


        Random case:

        List to sort: [6, 9, 4, 10, 3, 3, 3, 4, 0, 4]

        Sorted List:  [0, 3, 3, 3, 4, 4, 4, 6, 9, 10]


        Worst case:

        List to sort: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure 4.3.0: Shell-Sort console output.

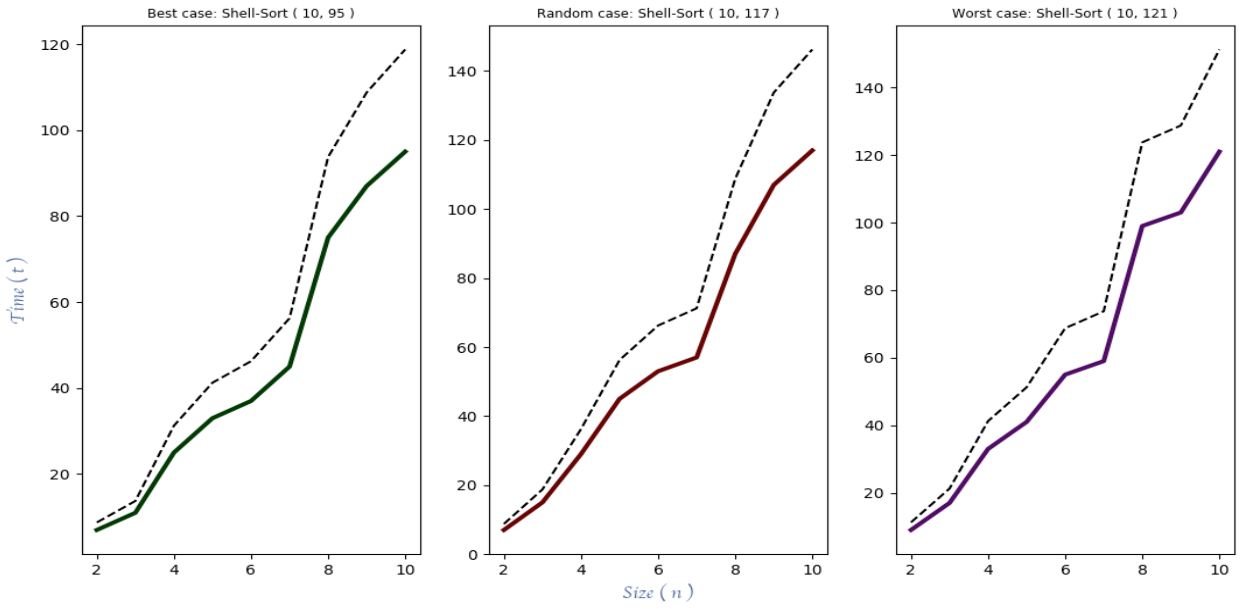


Figure 4.3.1: Shell-Sort graph for the lists in Figure 4.3.0.

Table 5 illustrate the plotting points of Figure 4.3.1 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 7 | 7 | 9 |
| 3 | 11 | 15 | 17 |
| 4 | 25 | 29 | 33 |
| 5 | 33 | 45 | 41 |
| 6 | 37 | 53 | 55 |
| 7 | 45 | 57 | 59 |
| 8 | 75 | 87 | 99 |
| 9 | 87 | 107 | 103 |
| 10 | 95 | 117 | 121 |

Table 5.

***Observation:*** *In Figure 4.3.1 there are 3 graphs, the one on the left represent the algorithm's best case, analogously the middle and right ones represents the random and worst case, in each graph, there is a* ***pointed*** *stroke, this it's an asymptotic function for the main plot.*

Second test of **Shell-Sort** algorithm. The program will plot the time that the algorithm takes to sort three lists of size $n = 20$. The first list will have the elements already sorted, the second will have the elements in random order, and the third list will have the elements sorted but in decreasing order. With this we will corroborate the complexity of this algorithm presented in section 2. Each list represents its *best, random and worst* case.

```
(myenv) MacBook-Pro-de-David:Sorting Algorithms Davestring$ python3 main.py

        Sorting Algorithms: Select one of the following options.

        1.- Heap-Sort.
        2.- Cocktail-Sort.
        3.- Shell-Sort.
        Answer: 3

        Best case:

        List to sort: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

        Sorted List:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

        Random case:

        List to sort: [13, 11, 9, 16, 13, 10, 16, 18, 0, 8, 16, 12, 14, 8, 2, 19, 15, 5, 10, 11]

        Sorted List:  [0, 2, 5, 8, 8, 9, 10, 10, 11, 11, 12, 13, 13, 14, 15, 16, 16, 16, 18, 19]

        Worst case:

        List to sort: [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

        Sorted List:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 4.3.2: Shell-Sort console output.

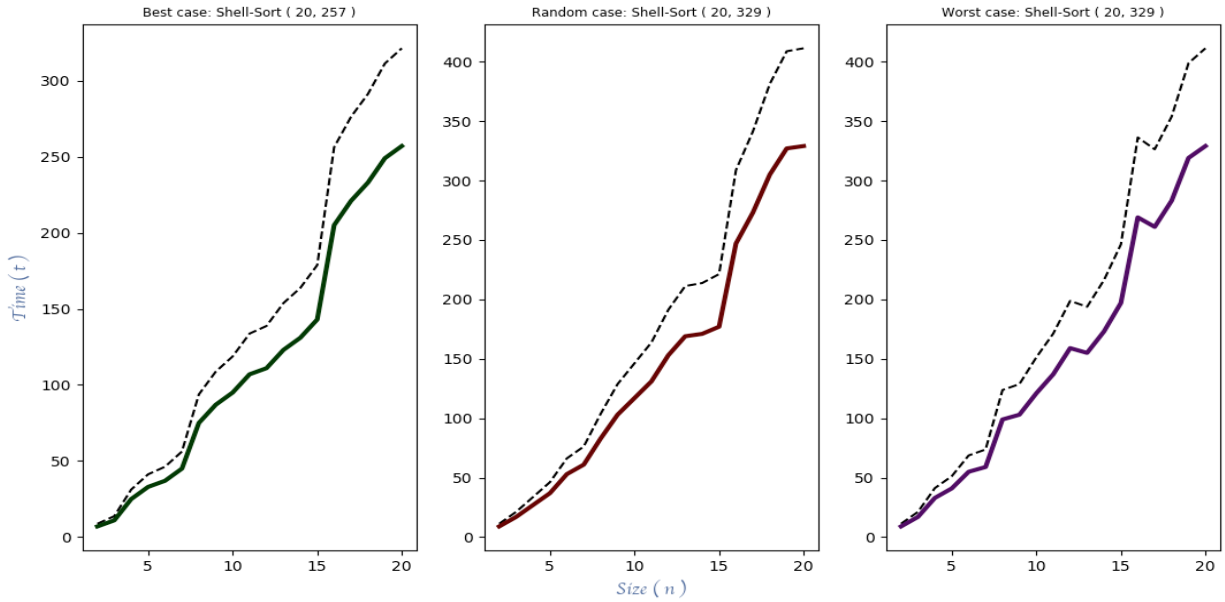


Figure 4.3.3: Shell-Sort graph for the lists in Figure 4.3.2.

Table 6 illustrate the plotting points of Figure 4.3.3 graphs, the first column represents the size of the list to sort, the second, third and fourth, the time that takes to sort that list in its best, random and worst case of the algorithm.

| Size ( n ) | Best Case Time ( t ) | Random Case Time ( t ) | Worst Case Time ( t ) |
|---|---|---|---|
| 2 | 7 | 9 | 9 |
| 3 | 11 | 17 | 17 |
| 4 | 25 | 27 | 33 |
| 5 | 33 | 37 | 41 |
| 6 | 37 | 53 | 55 |
| 7 | 45 | 61 | 59 |
| 8 | 75 | 83 | 99 |
| 9 | 87 | 103 | 103 |
| 10 | 95 | 117 | 121 |
| 11 | 107 | 131 | 137 |
| 12 | 111 | 153 | 159 |
| 13 | 123 | 169 | 155 |
| 14 | 131 | 171 | 173 |
| 15 | 143 | 177 | 197 |
| 16 | 205 | 247 | 269 |
| 17 | 221 | 273 | 261 |
| 18 | 233 | 305 | 283 |
| 19 | 249 | 327 | 319 |
| 20 | 257 | 329 | 329 |

Table 6.

Our last test for this algorithm consist in plotting the complexity of lists of sizes $n = 10000$ as we can see in Figure 4.3.4. In this examples we can visualize that the algorithm in its best case is near to $\mathcal{O}(n)$, otherwise $\mathcal{O}(n^{\frac{3}{2}})$ **or** $\mathcal{O}(n \log^2(n))$ time depending of the gap.

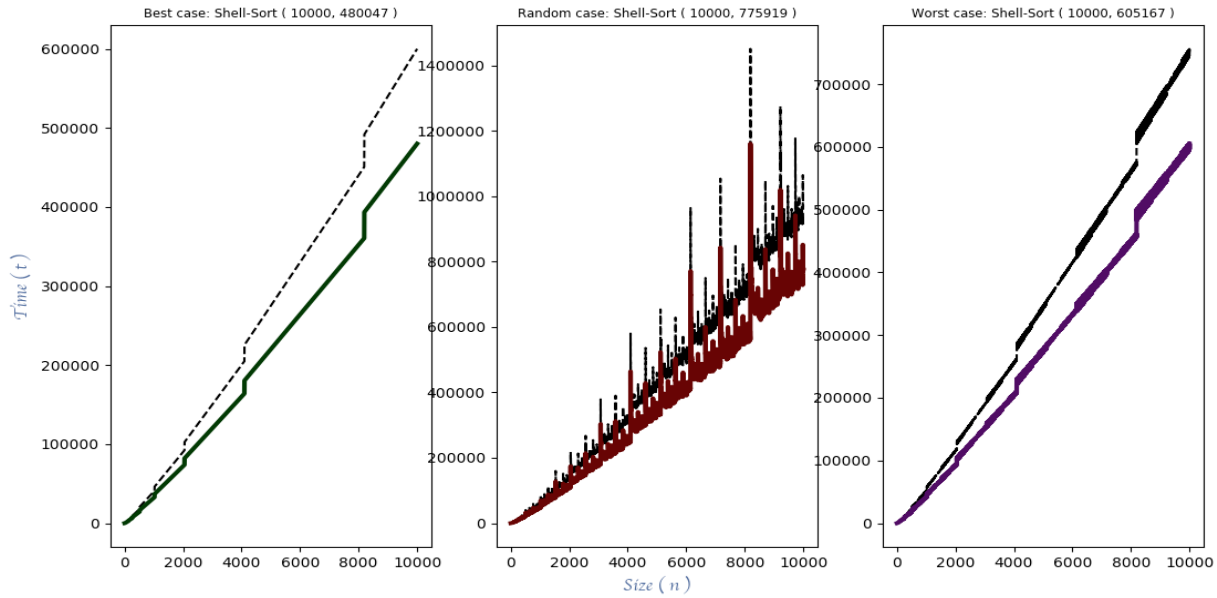


Figure 4.3.4: Shell-Sort complexity for lists of size n = 10000.

***Observation:*** *In Figure 4.3.4 we can visualize that the middle and right plots are something strange, I've been researching and this algorithm doesn't have an exact superior asymptotic function, the running time depends of how the programmer assign something know as* ***gap****, still, this algorithm is under* $\mathcal{O}(n^2)$ *and this, makes it better than* ***Insertion-Sort****.*

# 5 Conclusion:

It's very important to understand sorting algorithms. I think that there is not really a reason for implementing my own sorting algorithm from scratch for a production application. However in this course I have discovered advanced facets of a language along with programming design patterns that have given me an edge as a developer. In addition to combining multiple language components, sorting algorithms also had helped me to understand program accuracy and speed, because, just like in any type of program, sorting algorithms are not valid unless they are accurate. This algorithms give an abstract way of studying program accuracy and performance.

- Hernandez Martinez Carlos David.

# 6 Bibliography References:

[ 1 ] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.

[ 2 ] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.