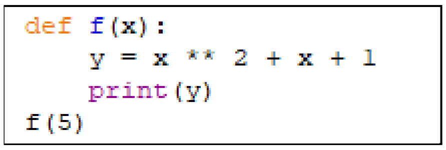
Investigating Big O of mathematical functions through time complexity of Sorting Algorithms

Number of Pages : 20

Introduction

As someone who does programming as a hobby, I interact fairly often with algorithms - a specified set of instructions to be followed in order to achieve a desired outcome in the quickest time possible. An algorithm is similar to how a function works in Mathematics. In fact, the code used below in Figure 1 is also called a function.



```
def f(x):
    y = x ** 2 + x + 1
    print(y)
f(5)
```

$$f(x) = x^2 + x + 1$$
$$f(5) = 31$$

Figure 1 - An identical Algorithm in Python and Function

The word "algorithm" is usually associated with computers, as computers have to handle large amounts of data in order to solve real world problems in the quickest time possible. In these cases, computers will often have to process and sort the objects in an array by different ways, such as in alphanumerical orders, dates, or more. This is an example of sorting - arranging objects systematically so that data can be processed easier by both humans and other computers. While watching a visualization of these sorting algorithms (Bingmann, 2013), I noticed that the different algorithms all sorted the array differently, with some splitting up the array into equal parts (**Merge**, Radix Sort) while others simply having all the values magically move into their sorted position (**Counting**, Gravity Sort). I also noticed that certain sorts were significantly more efficient than others, requiring less array accesses, swaps and time to sort the array. This undeniably made me more interested in finding out how each of the sorting algorithms worked.

Upon further inspection, I realized that the difference in time was due to each of the algorithms having a time complexity - meaning how long it takes to run the algorithm (in this case, to sort the array). I also learnt of "Big O", the worst case scenario for algorithms. Hence, I intend to establish how the following 4 sorting algorithms : **Selection Sort, Insertion Sort, Merge Sort & Counting Sort** work, their time complexity for an array of size *n*, and more importantly, to explore its mathematical implications in Big O.

Asymptotic Behavior and Time Complexity

The 'Big O' of a function describes how fast the *y* value of a function increases as *x* increases, as can be represented by O(*g*(*x*)), where *g*(*x*) is usually a simple function such as *x*, x^2 and 2^x in order to describe the behavior of a function in more general terms. The Big O of *f*(*x*), or O(*f*(*x*)), is O(*g*(*x*)) if the below is true.

$$\begin{aligned}&O(f(x)) = O(g(x))\\ &if\ \exists\ c \in \mathbb{R}\ such\ that\\ &f(x) \leq c \times g(x)\\ &\forall\, x \geq k,\ if\ \exists\ k \in \mathbb{R}^{+}\end{aligned}$$

"O of f(x) is equal to O of g(x) if there exists c (an element of Real Numbers) such that f(x) is lesser or equal to c × g(x) for all x greater than or equal to k, if there exists k (an element of positive Real Numbers)" (Aleksei, 2021)

The Big O describes the Asymptotic Behavior of a function, or what happens to the curve for very large values of *x*. This is how we find asymptotes for functions such as f(*x*) = 1 ÷ *x* - by evaluating the limit as it approaches the two ends of infinity. When finding the Big O for the function $\pi(x) = x^3 + 99x^2 + 99$, although it might seem that the dominant term is x^2 due to its large coefficient, we obtain the following when evaluating the limits of the function as *x* approaches infinity and negative infinity:

$$\lim_{x \to \infty} x^3 + 99x^2 + 99 = \infty$$

$$\lim_{x \to -\infty} x^3 + 99x^2 + 99 = -\infty$$

As x^3 changes faster than x^2, the terms $99x^2 + 99$ are insignificant when calculating the limit, and thus

$$O(\pi(x)) = O\left(x^3\right)$$

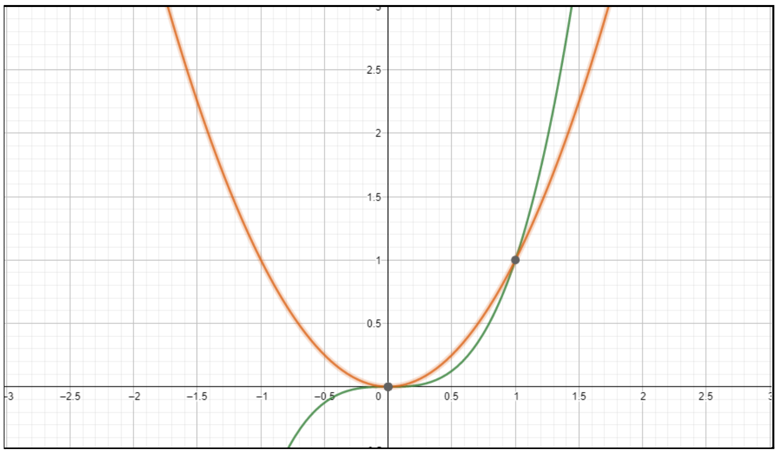


Fig 2 : Graph of $f(x^2)$ and $f(x^3)$

The derivatives of x^3 and x^2 are $3x^2$ and $2x$ respectively, and for all values of $x > ⅔$, $3x^2 > 2x$, and this means that the graph of x^3 will increase faster and approach ∞ faster than x^2 does. Furthermore, we also know that any function containing x^3 cannot be a part of $O(x^2)$, since it is impossible to find constants k and c such that

$$x^3 \leq c\,x^2$$
$$\forall\, x \geq k$$

Which can be proven by contradiction. Suppose that there exists k and c that satisfies the above inequality.

$$Let\ k \leq s < \infty$$
$$s^3 \leq cs^2$$
$$s^3 - cs^2 \leq 0$$
$$s^2(s - c) \leq 0$$
$$\therefore s - c \leq 0,\ \boldsymbol{s} \leq \boldsymbol{c}$$

However, since c is a real number while s can be any number less than infinity (but greater than or equal to k), there will be a value of s greater than c thus meaning that there cannot not exist c and k for these two functions, meaning $O(x^3) \neq O(x^2)$.

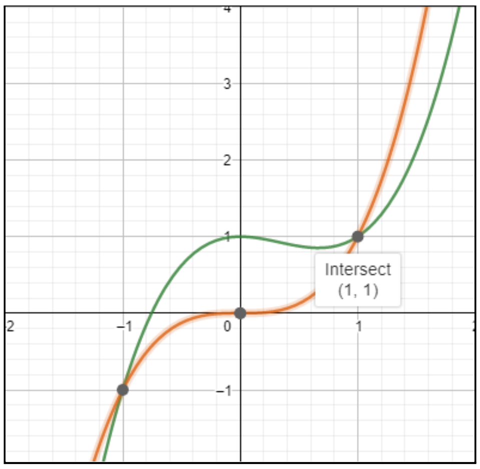


Figure 3 - Graph of f(x) = $x^3 - x^2 + 1$ and $g(x) = x^3$

Conversely, O($x^3 - x^2 + 1$) = O(x^3) since in this case, we can find values of c and k to satisfy the equation below. For example, $c = k = 1$ makes the inequality below true.

$$x^3 - x^2 + 1 \leq c \times x^3$$
$$\forall x \geq k$$

Although not exactly an asymptote, we still can say that as x approaches infinity, the distance between $x^3 - x^2 + 1$ and x^3 decreases, but not necessarily approaches 0, the formal definition of an asymptote. This is why the limiting behavior of a function, Big O, is also called asymptotic behavior. We can remove all terms except for the dominant term to find the Big O of any function. As such,

$$O\left(3^n + 2^n + n^5 + n + \log n + 30\right) = O(3^n)$$

Furthermore, when adding two functions together, the Big O of the added functions is the Big O of the constituent function that has the larger Big O.

$$(f(x) + g(x)) = O(max(O(f(x), O(g(x)))))$$

When dealing with the product of functions, the Big O's of each function are also multiplied. For example,

$$f(x) = x^2 - 5x + 6$$
$$g(x) = Ln\ x + 4$$
$$O(f(x)) = x^2$$
$$O(g(x)) = Ln\ x$$
$$O(f(x) \times g(x)) = O(f(x)) \times O(g(x)) = x^2\ Ln\ x$$

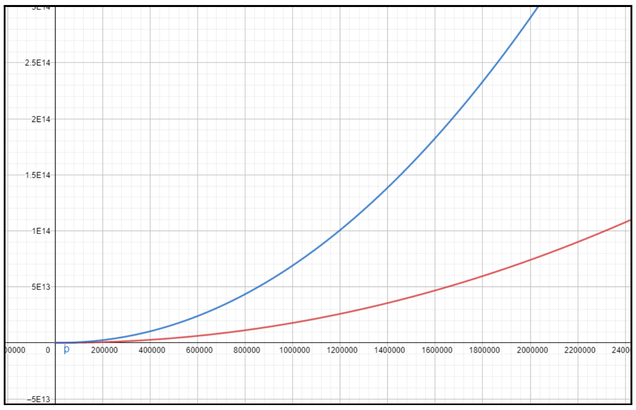


Fig 4 - Graph of 5(x^2 Ln x) (Blue) and f(x) × g(x) (Red)

In this case, for $c = 5$ and $k = 1\ 000\ 000$, $c\,x^2 \text{ Ln } x > f(x) \times g(x)$. Although this can be verified algebraically, zooming out shows that the blue line is always above the red line and this is sufficient enough to show that the Big O of the product of two functions is the product of the Big O of each function. (Graph can be accessed digitally through a link in Appendix)

Lastly, a function $f(x)$with a Big O of $g(x)$ multiplied by a constant h where $h \in \mathbb{R}$, will still have a Big O of $g(x)$, as seen below.

$$\begin{aligned}
&Let\ O(f(x)) = O(g(x))\\
&\therefore \exists\ c \in \mathbb{R}\ \&\ k \in \mathbb{R}^+\ such\ that\\
&f(x) \le c \times g(x)\ \forall x \ge k,
\end{aligned}$$

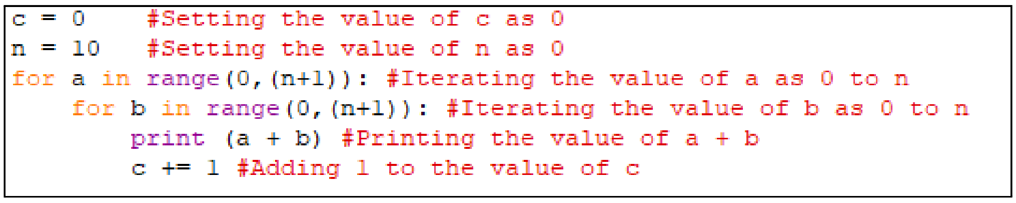
By multiplying both sides by h the following is obtained:

$$\begin{aligned}
&h \times f(x) \le h \times c \times g(x)\ \forall x \ge k\\
&Let\ C = h \times c\\
&h \times f(x) \le C \times g(x)\ \forall x \ge k\\
&\therefore O(h \times f(x)) = O(g(x))
\end{aligned}$$

Since there exists another constant C where the inequality holds true, it is proven that multiplying a function by any constant will still give the same Big O.

Other observations I made regarding Big O is that it can apply to certain terms such as sin x and cos x. Since there exists constants c and k to satisfy the inequality, their Big O is taken to be O(1). However, it does not apply to some other trigonometric functions, such as tan x, since it has periodic vertical asymptotes all the way to $x = \infty$, meaning that there cannot exist constants c and k to satisfy the inequality and thus impossible to find $g(x)$ for $O(g(x)) = O(\tan x)$.

An algorithm's "Big O" describes the upper bound for the time taken for an algorithm to carry out its function given an input size of n. The Big O is usually the worst case possible. An algorithm's Big O typically can be deduced from its code. For example, the code below will have a Big O of n^2, represented as $O(n^2)$. Big Omega and Big Theta notation are also measures of an algorithm's lower bound and in between, typically it's best case and average case respectively. This is an algorithm's time complexity.

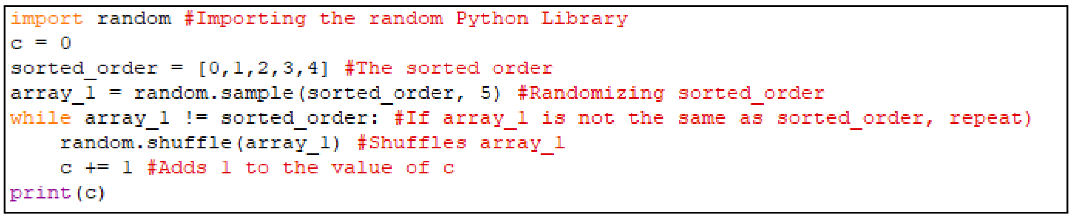


```python
c = 0    #Setting the value of c as 0
n = 10   #Setting the value of n as 0
for a in range(0,(n+1)): #Iterating the value of a as 0 to n
    for b in range(0,(n+1)): #Iterating the value of b as 0 to n
        print (a + b) #Printing the value of a + b
        c += 1 #Adding 1 to the value of c
```

Fig 5 - A code with a Big O of $O(n^2)$ made in Python

There are 2 nested *for* loops in the code above, meaning that there will be a total of nC_1 different values for a and b each, and a total of c values printed. Since $n = 10$, there will be a total of $10C_1 \times 10C_1 = 10^2$ iterations of c. If the value of n was set as 100, there will be a total of $100C_1 \times 100C_1 = 100^2$ iterations of c, showing how the run time of the algorithm scales with n - quadratically. Thus, it's Big O is indeed $O(n^2)$. This also follows the above arithmetic rules of Big O, as $O(n) \times O(n) = O(n^2)$

In sorting algorithms, as the position of each element in the array is random, there will be certain combinations where the same sorting algorithm can perform significantly better - the most extreme case being the **Bogosort**. Bogosort first checks if the elements are sorted - if they are not, the entire array will be randomized again, as shown in a code below for $n = 5$



```python
import random #Importing the random Python Library
c = 0
sorted_order = [0,1,2,3,4] #The sorted order
array_1 = random.sample(sorted_order, 5) #Randomizing sorted_order
while array_1 != sorted_order: #If array_1 is not the same as sorted_order, repeat)
    random.shuffle(array_1) #Shuffles array_1
    c += 1 #Adds 1 to the value of c
print(c)
```

Fig 6 - Figure of Code for Bogosort made in Python

Running the code once gave me $c = 59$. As Bogosort is completely random, the time it takes to sort the code can vary from sorting instantly to never successfully sorting the array. For Bogosort, we take it to have the Big O of $n!$, or O($n!$) since there are nP_n different possible combinations for the array. Thus, its lower limit, Big Omega, is $\Omega(1)$, and its Big Theta and Big O are $\Theta(n!)$ and $O(n!)$. The Big O of Bogosort may actually be $O(\infty)$ to represent how Bogosort may never successfully sort the array, but this is considered 'an abuse of notation' as infinity is not a real number. Thus, we assume the Big O of Bogosort to just be $n!$.

When looking at the graphs of the graphs $y=n!$, $y=2^n$, $y=n^2$, $y=n \log n$, $y = n$, $y = \log n$ and $y=1$ (Figure 7), the order of dominance of the terms can be visually seen by how fast they increase - its gradient. As the objective of algorithms is to be as efficient as possible, common Big O's are categorized by their efficiency, with $O(n!)$ being the worst while $O(1)$ being the best. While there will never be a sorting algorithm with a constant Big O, an example of an algorithm with a constant run time is accessing the element at index k for an array X with a length of n.

Even if n is extremely large, this code will ignore it and just directly access the element by doing 'X[k]' , meaning that the runtime will always be the same no matter the value of n.

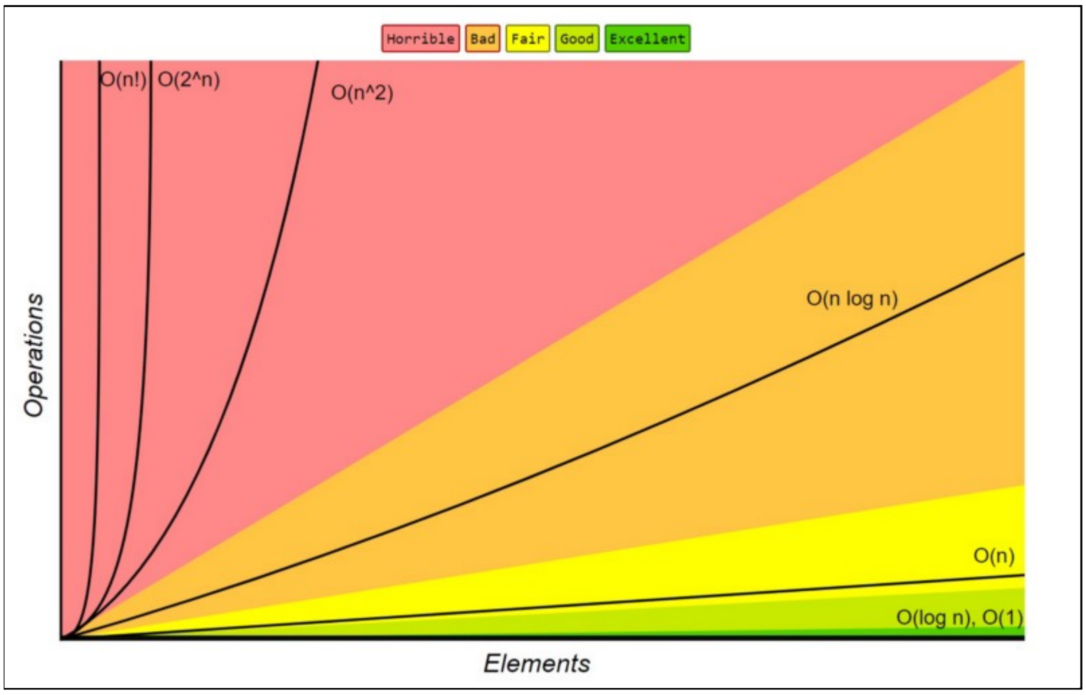


Fig 7 - Big O Complexity Chart (Prado, 2020)

Selection Sort

This sort is the most intuitive sorting algorithm, and most people will use this when presented with an array of items to be sorted. In the array X where

X = [4, 1, 2, 3, 5]

(These examples follow how arrays and indexing works in Python - X[0] accesses the element in the 0'th position (4) of the array 'X')

Selection sort works by first identifying the smallest element in an array, followed by swapping it with the element in X[0].In this case, it is X[1] = 1. '1' swaps with X[0] = 4 to result in

X = [1, 4, 2, 3, 5]

The process repeats for the rest of the numbers, identifying X[2] = 2 as the next lowest number. It swaps with X[1] = 4 to obtain

X = [1, 2, 4, 3, 5]

Then, swapping X[3] = 3 with X[2] = 4,

X = [1, 2, 3, 4, 5]

Although the array is now sorted, the code does not know that. It will identify the next lowest number, 4 at X[3] and if a number is already in its correct position, it is left there. The same goes for 5 at X[4]. The code then accesses and compares all values again to make sure that it is correct.

Since the code first has to identify the minimum element in the array, it will have accessed *n* elements when first swap occurs. The next lowest element requires (*n-1*) array accesses, followed by *(n-2)*, (*n-3*) and so on, following an arithmetic progression of common difference *d = -1*. As such, the total number of accesses will be equal to

$$\begin{aligned} \mathit{Total\ array\ accesses} &= \frac{n}{2}\left[u_1 + u_n\right] \\ &= \frac{n}{2}\left[n + 1\right] \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \end{aligned}$$

The best case for sorting algorithms will always be when the array is already sorted. However, the code for Selection Sort still requires to carry out all the above steps, resulting in a Big Omega of $\Omega(n^2)$. This also means that Selection sort is $\Theta(n^2)$ and $O(n^2)$, as the same steps are carried out regardless of order of the array, classifying selection sort as having a "horrible" time complexity. (Figure 7)

Insertion Sort

Insertion sort works using a sorted and an unsorted subarray. For the array X and Y where

X = [4, 1 , 2, 3, 5] , Y = [ ]

X[0] = 4 is first set as Y[0] and removed from X. The new X[0], is then appended to the end of Y and compared with the element before it. In this case, since 1 < 4, it is moved into Y[0]

X = [2, 3, 5] , Y = [1, 4]

The process repeats indefinitely until X becomes an empty array. X[0] = 2 is compared with Y[1] = 4, and since 2 < 4, it becomes the new Y[1] while Y[2] = 4. However, 2 > 1 and so '2' remains at Y[1].

X = [3, 5] , Y = [1, 2, 4] | **'2' is moved to Y[1]**

X = [5] , Y = [1, 2, 3, 4] | **'3' is moved to Y[2]**

X = [ ] , Y = [1, 2, 3, 4, 5] | **'5' is moved to Y[4]**

As X is now an empty array, the sorting process is completed and Y is a sorted version of the original X.

The worst case for Insertion Sort is when the array is in a descending order. This means that the sorting the second element will require 2 Array Accesses and 1 swaps (Total 3), the third element will require 3 Array Accesses and 2 swaps (Total 5) and so on, forming an arithmetic progression of odd numbers 1 + 3 + 5 + … + (2n-1) , which sum is known to be n^2.

$$S_n = n\left(\frac{a_1 + a_n}{2}\right) = n\left(\frac{1 + (2n - 1)}{2}\right) = n^2$$

Therefore, it has a Big O of $O(n^2)$, requiring as many array accesses as Selection Sort to sort the worst case.

Unlike selection sort, insertion sort's best case scenario is a much better $\Omega(n)$, which means that it's sort time scales linearly. In the best case scenario (array already sorted in ascending order), each element will only have to be compared once before it is sorted, and thus an array of length *n* will require *n* array accesses. The average case for insertion sort can also be found assuming each element will reach its sorted position in half the length of the sorted array. As such, it will require on average half the amount of array accesses compared to the worst case, or

$$\begin{aligned} Total\ array\ accesses &= \frac{1}{2}\left[\frac{1}{2}n^2 + \frac{1}{2}n\right] \\ &= \frac{1}{4}n^2 + \frac{1}{4}n \end{aligned}$$

Which simplifies to having a Big Theta of $\Theta(n^2)$. Insertion sort is overall a 'horrible' sorting algorithm, but its best case can be classified as fair. (Figure 7)

Merge Sort

Merge sort makes use of multiple subarrays, similar to insertion sort. The general process is the split arrays into half until each subarray has a length of 1, then merging them together.

For the array X where

X = [4, 1, 14, 21, 35, 6, 23]

The array X is split into sub-arrays A and B, each with half the length of X (or half minus 1)

A = [4, 1, 14, 21], B = [35, 6, 23]

Both A and B are split further (into halves) until each subarray has just 1 element in it

C = [4, 1], D = [14, 21], E = [35, 6], F = [23]

G = [4], H = [1], J = [14], K = [21], L = [35], M = [6], F = [23]

Now that each subarray has a length of 1, the subarrays are joined back together. G[0] and H[0] are compared, and since 4 > 1, the new C = [1, 4]. Repeating the process gives

C = [1, 4], D = [14, 21], E = [6, 35], F = [23]

Then, C[0] and D[0] are compared. Since 1 < 14, C[0] is first appended to A. Then, the remaining D[0] is compared with C[1], and since 14 > 4, C[1] is appended to A. C is now empty, and the rest of D is appended to A to form

A = [1, 4, 14, 21]

As for E and F, since E[0] < F[0], E[0] is appended to B. Then, F[0] is compared with E[1], and since 23 < 35, F[0] is appended to B. Now that F is empty, the rest of E is appended to B to give

A = [1, 4, 14, 21], B = [6, 23, 35]

The elements in A and B are then compared again using the same method to form a sorted version of X

X = [1, 4, 6, 14, 21, 23, 35]

The time complexity of this entire process is found to be O(*n* log *n*), which classifies it as a good sorting algorithm (Figure 7) Let *n* be the total length of the array and *m* be the length of each subarray in a level. At the lowest level (*m* = 2), there will be *n*/2 sub-arrays, and since there are only 2 elements in each subarray, 1 comparison has to be made between each sub-array. At the second lowest level(*m* = 4), the following tree diagram (Figure 8) shows all possible combinations that result in a sorted array. (2,2 indicates that each sub-array has 2 elements in it and 0,0 implies that the sub-arrays have been sorted as both sub-arrays are empty). A red line indicates a comparison occurred.

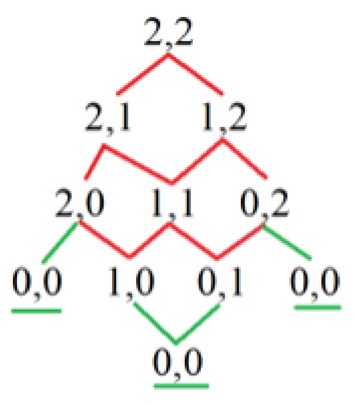


Fig 8 - Possible ways for 2 sub-arrays of length 2 to be sorted

The minimum number of comparisons needed is 4 ÷ 2 = 2 , which happens when all elements in one array are smaller than the smallest element in the other array.The maximum number of comparisons happens when there is a comparison between every element until the last element remains, and since an element is removed after every comparison, the worst case scenario has at most 4 - 1 = 3 comparisons. There will also be at most *n* ÷ 4 subarrays of length 4, and since each subarray requires at most 3 comparisons, the total number of comparisons needed for this level is at most $n \div 4 \times 3$.

Subsequently, the minimum number of comparisons needed for a sub-array of size 8 is 4, while the maximum is 7. In general, the minimum number of comparisons needed for a sub-array of size n is $n \div 2$, while the maximum is $n - 1$. This explains why exactly $n \div 2$ comparisons are made at the lowest level, since both $2 \div 2$ and $2 - 1$ equal to 1. Thus, the time complexities of merging each sub-array is $\Omega(m \div 2)$ and $O(m - 1)$, which both simplify to linear time, m. There will be at least $n \div m - 1$ and at most $n \div m$ subarrays of length m, and thus the time complexity of sorting all sub-arrays in the same level is $\Omega((n \div m - 1) \times m)$ and $O((n \div m) \times m)$, which are $\Omega(n - m)$ and $O(n)$ for sorting an entire level of sub-array length m. Note that since m is not a variable but rather just a power of 2, it can be considered insignificant when dealing with n, the total size of the array, thus reducing both time complexities to just n, which means the time complexity of sorting all sub-arrays in each level is n, regardless of which level or the size of the array, m. (Ω,Θ and O(n)) Lastly, since the sizes of the sub-arrays are halved recursively, there are a total of $\lceil \log_2 n \rceil$, where $\lceil x \rceil$ indicates the value of x rounded up to the nearest whole number, different sub-array levelsan array of size n. For example, when $n = 16$, there will be sub-array lengths of sizes 8, 4, 2 and 1 for a total of 4 levels, which is $\log_2 16$. In the example given, $n = 7$, and $\lceil \log_2 7 \rceil = 3$. As such, multiplying the time complexity of sorting each sub-array with the number of sub-arrays gives the final time complexity of Ω,Θ and $O(n \times \log_2 n)$ for Merge Sort. Note that it is difficult to calculate the average time complexity of sorting each sub-array. However, we know that the upper bound (Big O) and lower bound (Big Ω) are both linear time complexities, meaning the middle bound (Big Θ) must also be of linear time.

Counting Sort

Counting sort works by accessing the entire array, and “counts” the number of times an element occurs in the array. For an array X where the range is known to be 0 to 4,

X = [4, 1, 1, 2, 4, 0, 0, 4, 1, 3]

Counters for the numbers 0 to 4 are created. It then accesses each element in the array and adds 1 to its respective counter. For example, X[0] will add one the ‘4’ counter as X[0] = 4.

After it reaches the last index, the code will have compiled the amount of times each number has appeared in the entire array, displayed visually below.

Number	0	1	2	3	4
Count	2	3	1	1	3

From here, the code simply appends the numbers from lowest to highest to a new array, Y. It appends ‘0’ 2 times, ‘1’ 3 times, ‘2’ 1 time, ‘3’ 1 time and ‘4’ 3 times to result in

Y = [0, 0, 1, 1, 1, 2, 3, 4, 4, 4]

The code requires n array accesses to count n elements in the array, $d+1$ array accesses to check d different counters and another n array accesses to place the numbers in their sorted position. In the average case, the code requires $2n + d$ accesses to sort the array. Since $d \leq n$, it is fine to simplify it to $3n$ which further simplifies to $\Theta(n)$, As the code runs the same way independent of the order of the array, Counting sorts time complexity = $O(n) = \Theta(\mathrm{n}) = \Omega(n)$. This means that even in the worst case, the run time of the algorithm will still be the same. Since it has a time complexity of $O(n)$, it is classified as an excellent sorting algorithm.

Simulations and Data

Data was generated using a random array generator followed by running this random array through each sorting algorithm. These codes are all found in the Appendix. The random array generator generates a random string of numbers from 0 to 1000 with a length of 1 to 999, which is n. A sorted array is also created each time using Python’s built-in sorting algorithm Timsort, a hybrid between Insertion and Merge sort that has a time complexity of O(n log n) to make sure that the array ‘sorted’ using the sorting algorithm is actually sorted when the main sorting code terminates. Although this process requires a time n to run in reality, I will be ignoring it as the focus is on the time actually taken to for each algorithm to sort each array. Each sorting algorithm counts and prints out the total number of accesses and swaps it used to sort the array and this is used to determine the run-time for each algorithm, which is better than measuring the code's actual run-time since that varies depending on many variables unaccounted for.

I will be using 4 Sorting Algorithms and running arrays of lengths 1 to 999 through each of them. This process was repeated for a total of 5 times, resulting in a total of 19980 trials. This data was transferred to Google Sheets which was used to determine the sorting algorithm's time complexity for each array length. The Big O and Big Ω were found using the maximum and minimum 'time' needed to sort the array for all 5 trials, while the Big Θ was found by averaging the 'time' needed by all 5 trials. The run-times are then plotted against the size of the array, *n*.

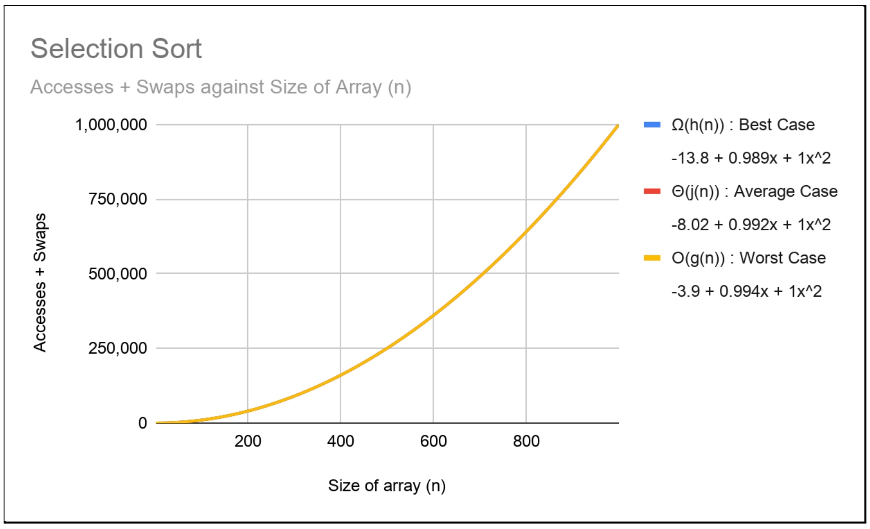


Figure 9 : Graph of 'run time' using Selection Sort against size of array

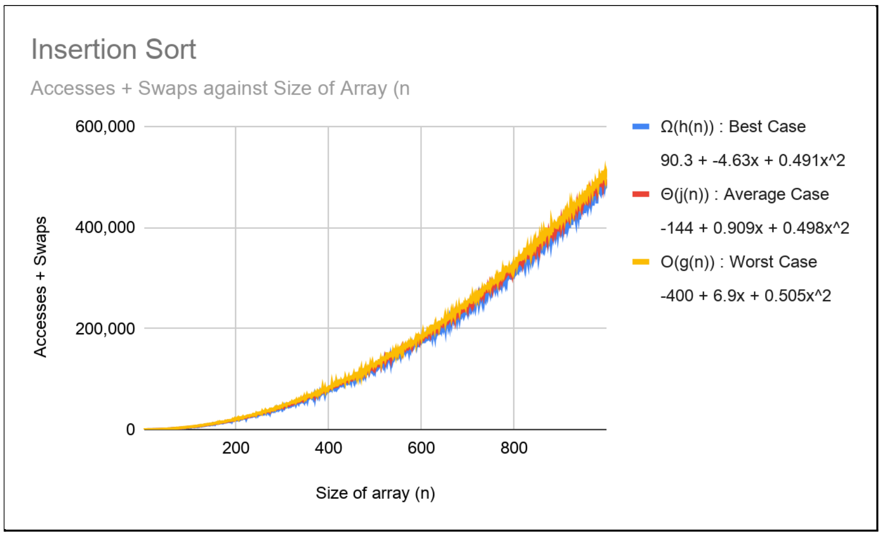


Figure 10 : Graph of 'run time' using Insertion Sort against size of array

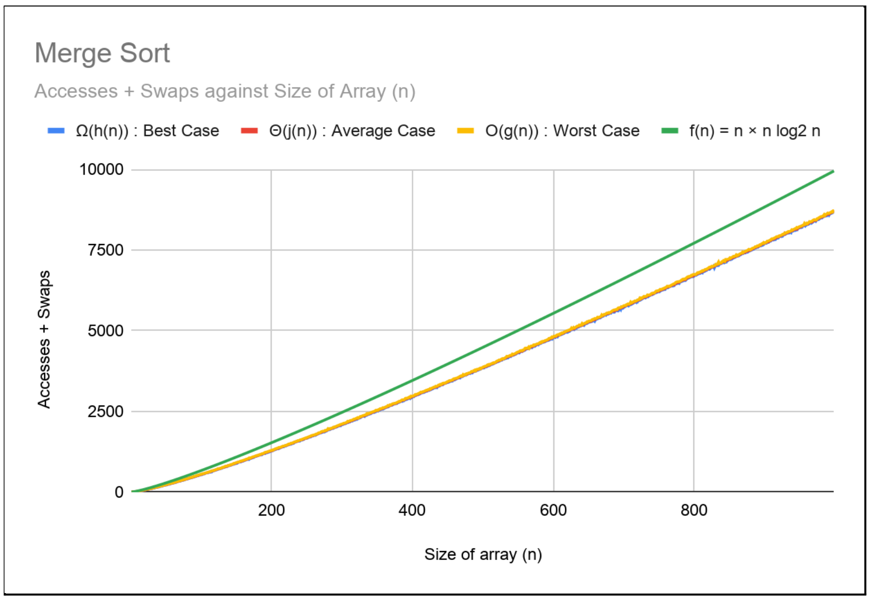


Figure 11 : Graph of ‘run time’ using Merge Sort against size of array

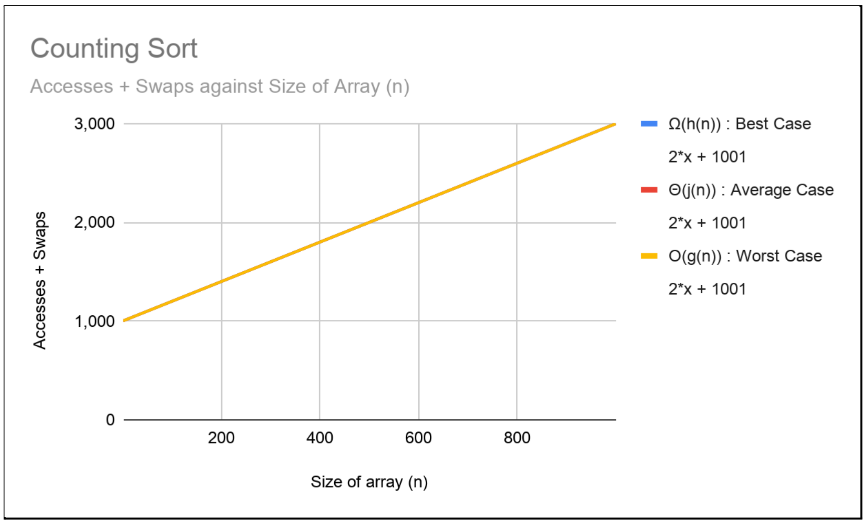
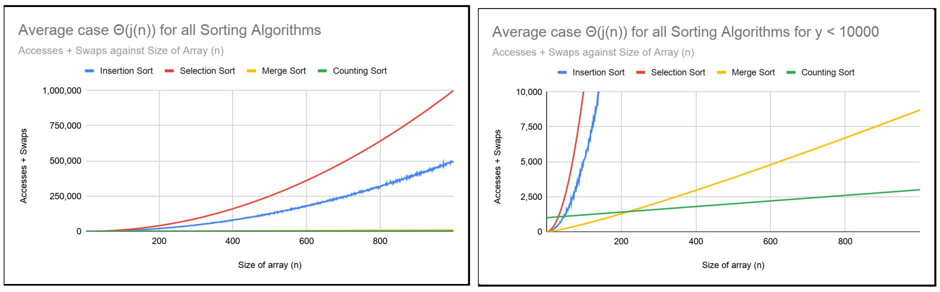


Figure 12 : Graph of ‘run time’ using Counting Sort against size of array



Figures 13 & 14 : Graph of ‘run time’ for Θ(j(n)) for all algorithms against size of array

The immediate observation is that the best and worst cases of these scenarios did not vary much from its average case, since the largest array length (*n* = 999) is still relatively small, resulting in obtaining scenarios at either end of the spectrum improbable. (In the real world, the length of these arrays can be much higher than a million.) Furthermore, 3 of the 4 sorting algorithms had almost exactly the same run time for all trials due to its best cases having the same time complexity as its worst cases, which is summarized in the table below.

Sorting Algorithm	Best Case	Average Case	Worse Case
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n)$	$\Theta(n)$	$O(n)$

The shapes of the graphs all correspond with the time complexity of its average case, and there exists constants *c* and *k* that will make the inequality defining the Big O of each algorithm true for all $999 \geq n \geq k$, which can be seen visually in Figure 11 for Merge Sort and be calculated algebraically from the equations of best fit lines for Figures 9, 10 and 12 for Selection, Insertion and Counting Sorts. From the equations of the best fit lines, it can also be deduced that this will continue for values of $n > 999$.

When compared, it is evident that the run-time for Selection Sort increases faster than Insertion Sort as *n* increases, taking about half as many Accesses ($0.498x^2$ and $1x^2$).

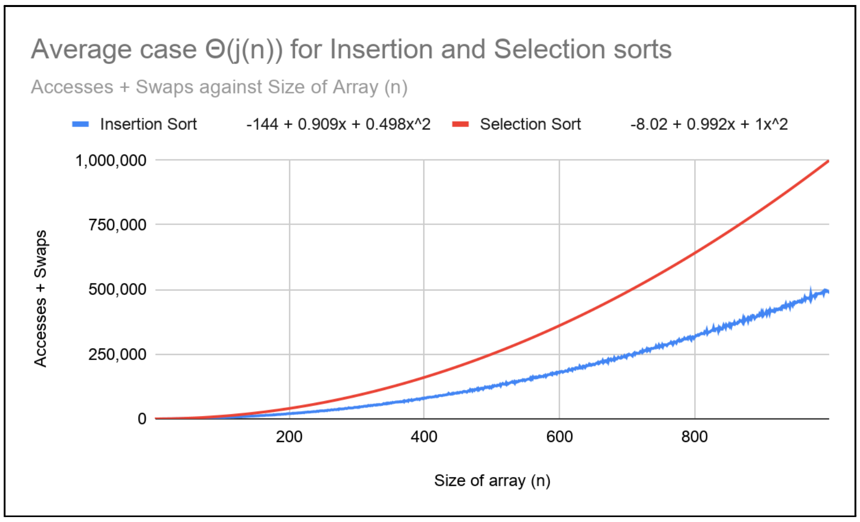


Figure 15 : Graph of 'run time' for Θ(j(n)) for Insertion and Selection sorts against size of array

This happens because the average run time of Selection sort is ½ n^2 + ½ n, while the average run time of Insertion sort is *¼ n^2 + ¼ n^2 = ½ (½ n^2 + ½ n)*, which can be seen by looking at the coefficient of the x^2 term in both sorts - 0.498 ≈ ½ (1). While both algorithms have a time complexity Θ(n^2), Insertion sort is able to sort about two times faster as it accesses and swaps until an element is in its correct position, while Selection sort swaps once it has accessed every element in the unsorted section of the array until it finds the lowest value, resulting in many unnecessary accesses. It also explains why the time complexity for the Best Case of Insertion Sort is much better than the Best case of Selection Sort.

This can prove useful when designing other sorting algorithms, such as the aforementioned sorting algorithm Timsort which has the following time complexities:

Timsort	*Ω(n)*	*Θ(n log n)*	*O(n log n)*

Timsort is a hybrid between Insertion Sort and Merge Sort, meaning that it was designed in a way to have properties of both - It has a best case of Ω(n) from Insertion Sort but only has the average and worst cases of Θ(n log n) and O(n log n) from Merge Sort. This results in a more efficient sorting algorithm which explains why it is used as the default sorting algorithm in many programming languages such as Python.

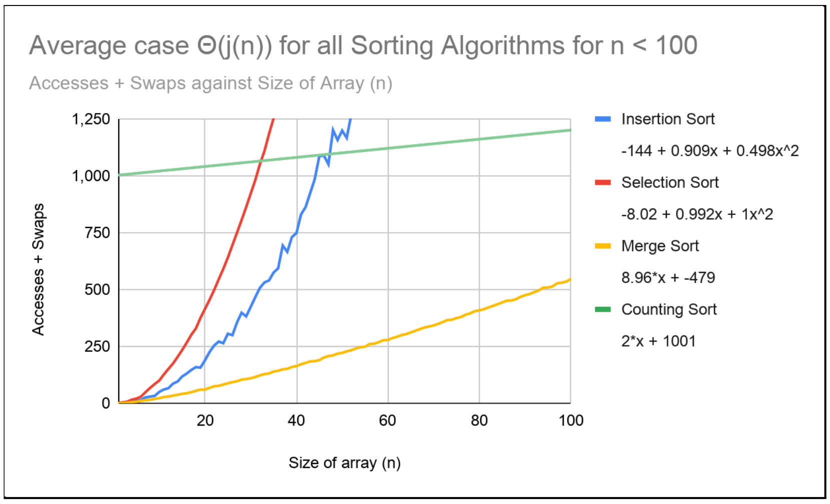


Figure 16 : Graph of 'run time' for Θ(j(n)) for all algorithms against size of array for n < 100

This raises the question of why Counting Sort, which seems to be the best sorting algorithm, is not used instead. It has many limitations that the other sorting algorithms are not affected by - It requires that the range of the array is known to be able to create counters for the elements. Furthermore, even though it has a linear time complexity, it's run time for $n = 1$ is 1003 array accesses, much higher than any of the other algorithms. (Figure 16) This is due to the range of elements [1, 1000] being much higher than n, meaning that a total of 1000 counters are created although only 1 is used throughout the sorting process. In Mathematical terms, this is because when looking at the equation of the best fit line for Counting Sort (2x + 1001), substituting x = 1 gives 1002, a very large number for a small x. Since non-integers do not have a 'range', counting sort is only able to sort integer values. Counting Sort also has a much higher space complexity (It has to store all its counters), meaning that Counting Sort will also be space-inefficient if its range, R, is a very large number, which is the assumption that both time and space complexities have - that n approaches infinity. For reference, the space-complexities of the 4 algorithms used are shown below. (Joshycsm, 2020) However, given a small data set of just integers, counting sort may be more efficient in sorting the data as compared to merge sort.

Space Complexity	Selection Sort	Insertion Sort	Merge Sort	Counting Sort
	O(1)	O(1)	O(n)	O(R)

Lastly, although the best, average and worst time complexities for Merge sort are all $n \log_2 n$, the graphs were all slightly below the function $y = n \log n$. The longest time needed to sort each sub array is m - 1, which was simplified to m to calculate the time required to sort an entire level of subarray of length m, which was n, the total size of the array. Since this is an overestimation ($m - 1 < m$), it comes naturally that the actual run times for Merge Sort will fall below the graph of $n \log n$. Thus, it is still possible to find the values of c and k to satisfy the equation that O(Merge Sort) is part of O($n \log n$). In this case, $c = 1$ and k can be equal to 500. Merge Sort also showed the second largest deviation in all 5 Trials

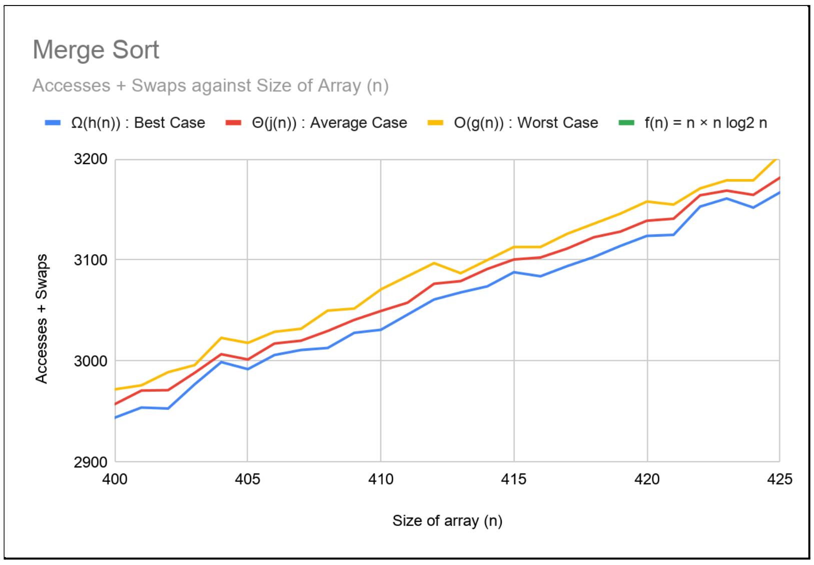


Figure 17 : Graph of ‘run time’ using Merge Sort against size of array zoomed in

This is again due to the differences between the exact runtime required to sort each sub-array, which is summarized below. The differences in the graphs can be attributed to the differences between each step of the algorithm, even if they are all linear.

Algorithm	Best Case	Worst Case
Sorting each Sub-array	$\Omega(m \div 2)$	$O(m - 1)$
Amount of Sub-arrays of length m in a level	$\Omega(n \div m - 1)$	$O(n \div m)$

Conclusion

For mathematical functions, there is no such thing as a ‘Excellent’ or ‘Horrible’ function, but the implication of Big O is how it establishes the dominance of certain terms, meaning that they increase faster than terms below it. Thus, when evaluating the limits of these functions, only the most dominant term needs to be considered. This is summarized in the table below for common terms, where dominance of a term increases going down the table. The effectiveness of an algorithm with that time complexity is also included below.

Big O	Growth Type	Algorithm ‘Rating’	Sorting Algorithm
$O(1)$	Constant	Excellent	NA
$O(\log(n))$	Logarithmic	Excellent	NA
$O(n)$	Linear	Fair	Counting Sort
$O(n \log (n))$	Linear-Logarithmic	Poor	Merge Sort
$O(n^k)$	Polynomial	Horrible	Insertion Sort Selection Sort
$O(k^n)$	Exponential	Horrible	Na
$O(n!)$	Factorial	Horrible	Bogo Sort

The most efficient sorting algorithm found in this paper is Counting Sort, which has a linear time complexity. However, for general cases, Merge Sort (or Timsort) is much more effective in sorting an array. Although I used sorting algorithms to explain time complexity, Big O in computer science actually applies to any type of code. (For example, Figure 1, Figure 5, and any other code I have ever written), and this exploration has given me an idea on how to judge a code by its time complexity and I feel that the code I write in the future will definitely be more time-efficient.

References

Bingmann, T. (2013, May 22). The Sound of Sorting - "Audibilization" and Visualization of Sorting Algorithms. Retrieved January 15, 2021, from https://panthema.net/2013/sound-of-sorting/

Are there any worse sorting algorithms than Bogosort (a.k.a Monkey Sort)? (2010). Retrieved January 15, 2021, from https://stackoverflow.com/questions/2609857/are-there-any-worse-sorting-algorithms-than-bogosort-a-k-a-monkey-sort

Aleksei. (2014). Asymptotics. Retrieved January 15, 2021, from https://mathwiki.cs.ut.ee/asymptotics

Prado, K. (2020, February 15). Understanding time complexity with python examples. Retrieved January 15, 2021, from https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

Joshi, V. (2017, July 17). Counting Linearly With Counting Sort. Retrieved January 15, 2021, from https://medium.com/basecs/counting-linearly-with-counting-sort-cd8516ae09b3#:~:text=However%2C%20counting%20sort%20is%20generally,it%20a%20linear%20sorting%20algorithm

Lithmee. (2019, May 10). What is the difference between insertion sort and selection sort. Retrieved January 15, 2021, from https://pediaa.com/what-is-the-difference-between-insertion-sort-and-selection-sort/#:~:text=The%20main%20difference%20between%20insertion,the%20element%20in%20the%20correct

Caldwell, C. (n.d.). Big-o. Retrieved January 15, 2021, from https://primes.utm.edu/glossary/page.php?sort=BigOh

Joshycsm. (2020, August 03). Which sorting algorithm is best? Retrieved January 15, 2021, from https://joshycsm.medium.com/which-sorting-algorithm-is-best-ca83e3cc3ca0

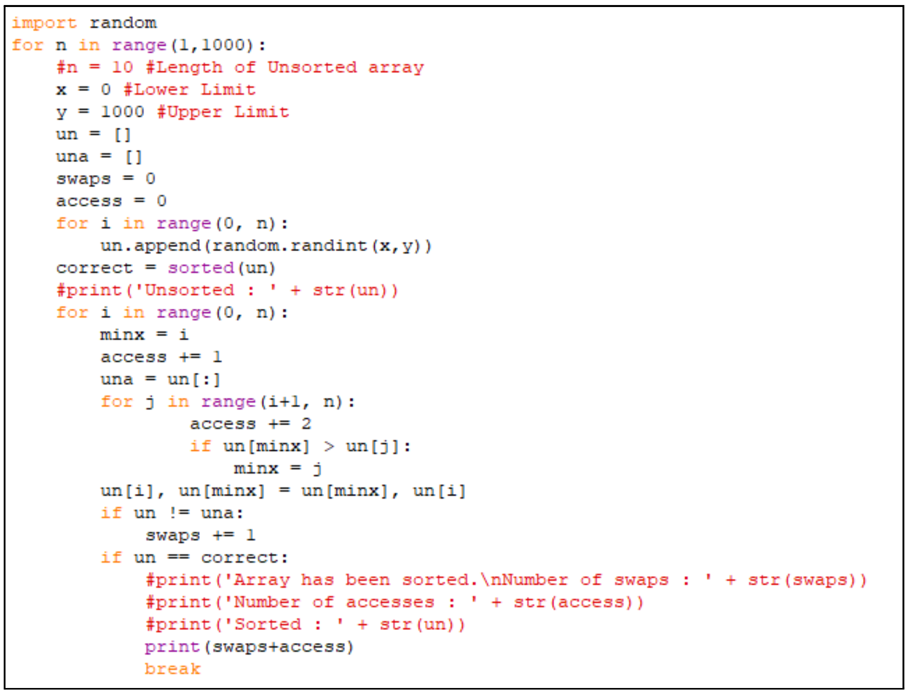
O-notation, O(∞) = O(1)? (2011). Retrieved January 15, 2021, from https://stackoverflow.com/questions/5627390/o-notation-o%E2%88%9E-o1

Weisstein, E. W. (n.d.). Asymptote. Retrieved January 15, 2021, from https://mathworld.wolfram.com/Asymptote.html

Python Program for Merge Sort. (2020, March 31). Retrieved January 15, 2021, from https://www.geeksforgeeks.org/python-program-for-merge-sort/

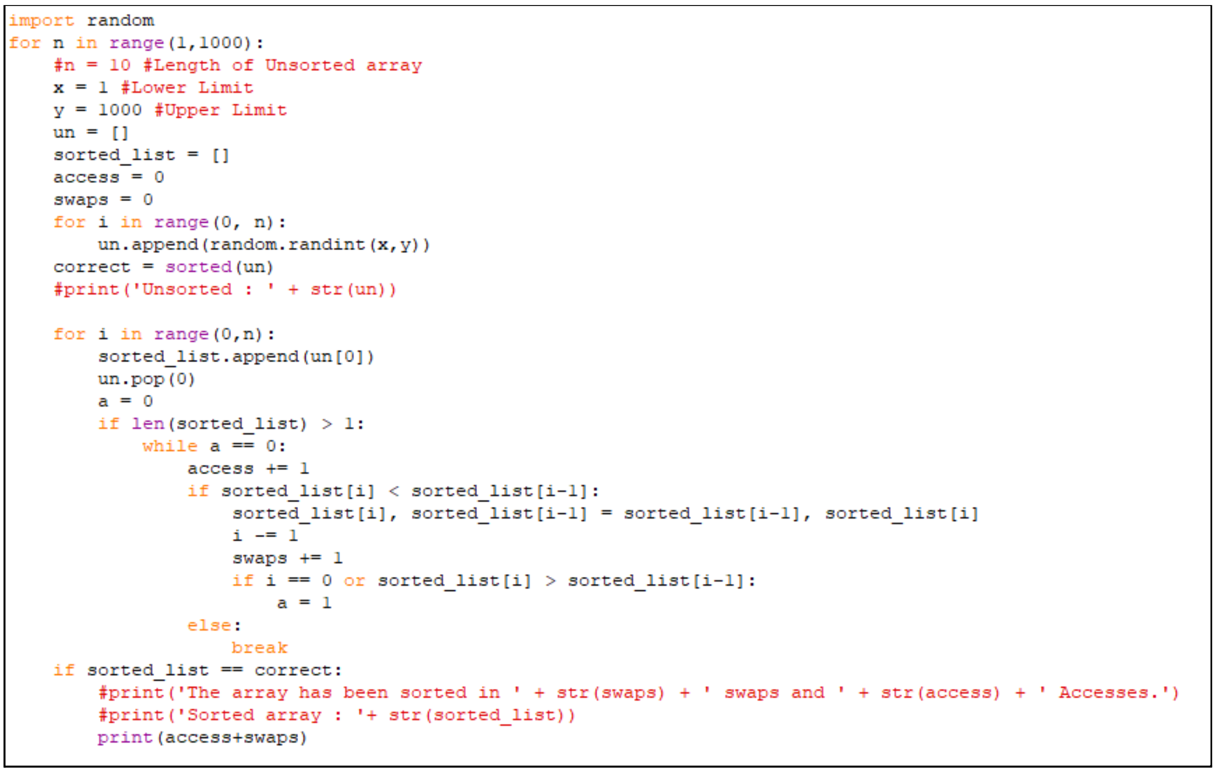
Appendix

Figure 4 : https://www.geogebra.org/graphing/xafyvsfh



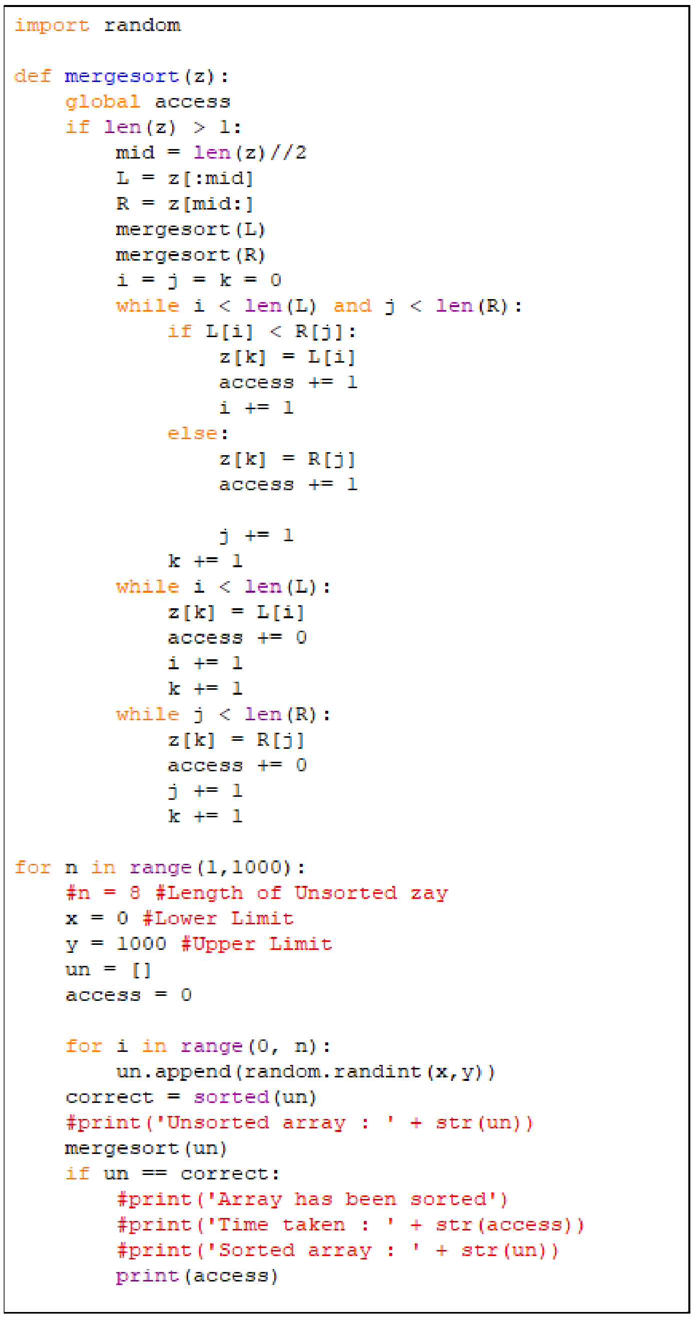
```python
import random
for n in range(1,1000):
    #n = 10 #Length of Unsorted array
    x = 0 #Lower Limit
    y = 1000 #Upper Limit
    un = []
    una = []
    swaps = 0
    access = 0
    for i in range(0, n):
        un.append(random.randint(x,y))
    correct = sorted(un)
    #print('Unsorted : ' + str(un))
    for i in range(0, n):
        minx = i
        access += 1
        una = un[:]
        for j in range(i+1, n):
                access += 2
                if un[minx] > un[j]:
                    minx = j
        un[i], un[minx] = un[minx], un[i]
        if un != una:
            swaps += 1
        if un == correct:
            #print('Array has been sorted.\nNumber of swaps : ' + str(swaps))
            #print('Number of accesses : ' + str(access))
            #print('Sorted : ' + str(un))
            print(swaps+access)
            break
```

Figure 17: Code for Selection Sort Algorithm made in Python



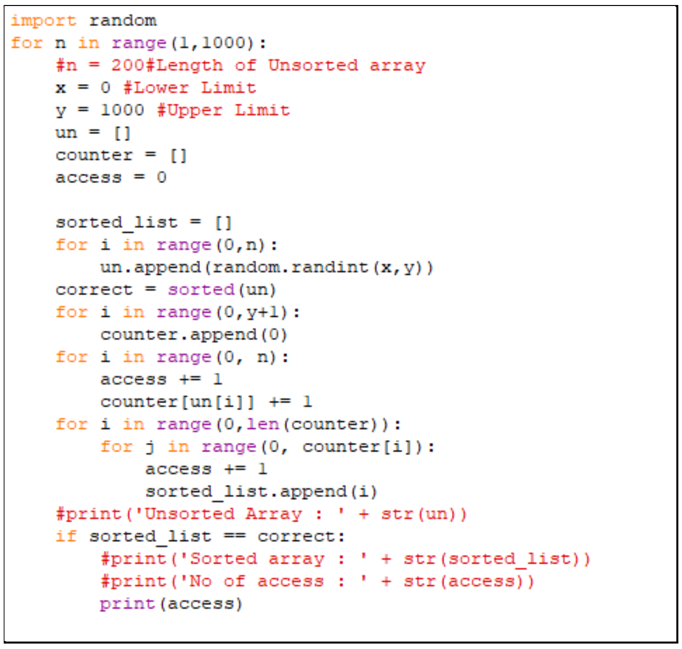
```python
import random
for n in range(1,1000):
    #n = 10 #Length of Unsorted array
    x = 1 #Lower Limit
    y = 1000 #Upper Limit
    un = []
    sorted_list = []
    access = 0
    swaps = 0
    for i in range(0, n):
        un.append(random.randint(x,y))
    correct = sorted(un)
    #print('Unsorted : ' + str(un))

    for i in range(0,n):
        sorted_list.append(un[0])
        un.pop(0)
        a = 0
        if len(sorted_list) > 1:
            while a == 0:
                access += 1
                if sorted_list[i] < sorted_list[i-1]:
                    sorted_list[i], sorted_list[i-1] = sorted_list[i-1], sorted_list[i]
                    i -= 1
                    swaps += 1
                    if i == 0 or sorted_list[i] > sorted_list[i-1]:
                        a = 1
                else:
                    break
    if sorted_list == correct:
        #print('The array has been sorted in ' + str(swaps) + ' swaps and ' + str(access) + ' Accesses.')
        #print('Sorted array : '+ str(sorted_list))
        print(access+swaps)
```

Figure 18: Code for Insertion Sort Algorithm made in Python



```python
import random

def mergesort(z):
    global access
    if len(z) > 1:
        mid = len(z)//2
        L = z[:mid]
        R = z[mid:]
        mergesort(L)
        mergesort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                z[k] = L[i]
                access += 1
                i += 1
            else:
                z[k] = R[j]
                access += 1

                j += 1
            k += 1
        while i < len(L):
            z[k] = L[i]
            access += 0
            i += 1
            k += 1
        while j < len(R):
            z[k] = R[j]
            access += 0
            j += 1
            k += 1

for n in range(1,1000):
    #n = 8 #Length of Unsorted zay
    x = 0 #Lower Limit
    y = 1000 #Upper Limit
    un = []
    access = 0

    for i in range(0, n):
        un.append(random.randint(x,y))
    correct = sorted(un)
    #print('Unsorted array : ' + str(un))
    mergesort(un)
    if un == correct:
        #print('Array has been sorted')
        #print('Time taken : ' + str(access))
        #print('Sorted array : ' + str(un))
        print(access)
```

Figure 19: Merge Sort Algorithm made in Python

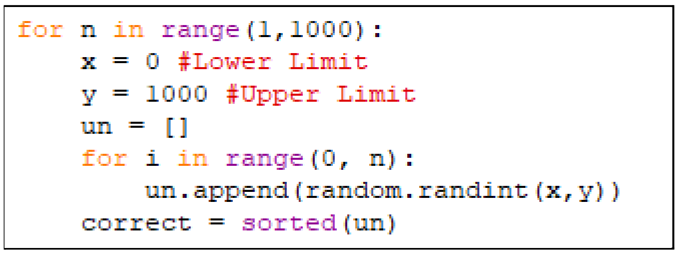


```
import random
for n in range(1,1000):
    #n = 200#Length of Unsorted array
    x = 0 #Lower Limit
    y = 1000 #Upper Limit
    un = []
    counter = []
    access = 0

    sorted_list = []
    for i in range(0,n):
        un.append(random.randint(x,y))
    correct = sorted(un)
    for i in range(0,y+1):
        counter.append(0)
    for i in range(0, n):
        access += 1
        counter[un[i]] += 1
    for i in range(0,len(counter)):
        for j in range(0, counter[i]):
            access += 1
            sorted_list.append(i)
    #print('Unsorted Array : ' + str(un))
    if sorted_list == correct:
        #print('Sorted array : ' + str(sorted_list))
        #print('No of access : ' + str(access))
        print(access)
```

Figure 20: Counting Sort Algorithm Made in Python

n	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Ω(h(n)) : Best Case	Θ(j(n)) : Average Case	O(g(n)) : Worst Case
312	2,220	2,201	2,197	2,194	2,205	2,194	2,203	2,220
313	2,206	2,216	2,220	2,216	2,205	2,205	2,213	2,220
314	2,196	2,212	2,221	2,224	2,206	2,196	2,212	2,224
315	2,225	2,219	2,201	2,235	2,223	2,201	2,221	2,235
316	2,222	2,249	2,230	2,233	2,233	2,222	2,233	2,249

This table shows sample data collected when running simulations of Merge Sort for $311 < n < 317$.
For the complete data set, please refer to this link → https://pastebin.com/wdqw33EV



```
for n in range(1,1000):
    x = 0 #Lower Limit
    y = 1000 #Upper Limit
    un = []
    for i in range(0, n):
        un.append(random.randint(x,y))
    correct = sorted(un)
```

Figure 21 - Code used to generate random Arrays as well as a sorted version of that array