Performance Analysis

Neil John J. Jomaya

September 10, 2024

1 Performance Analysis

Analyzing the performance of one's programs is an important step in becoming a good software developer. Being conscious of how efficient and quick your program is will enhance your program designing capabilities as you will know which algorithm (and technologies) to use based on the situation you are in. The importance of this concept has already been discussed partly in Topic 1 from the Python source code regarding the sorting function and its performance when exposed in small and big inputs.

In this part of the course, we will discuss ways on analyzing performance of our programs and how we can determine if the program we will create is efficient and quick based on the inputs and outputs required by the problem. So how do we determine if an algorithm is good enough? Here are some of the criterias we can follow:

1. Does it do what we expect it to do?
2. Does it work correctly within the scope of the program's requirement?
3. Can one easily understand the workings behind that algorithm by reading the documentation supplied?
4. Is the code understandable when one glances at it?

The criterias mentioned above are general and cannot be achieved easily due to the complexity and diversity of programming problems present. However, throughout this course, you, the student, will get a firsthand experience on how to make your mind work subconcsiously into meeting those criterias. This approach strives to hone your subconscious so that when you make your programs later on, you will try your best to reach these goals on your own without thinking too much about them.

There are other criterias to determine how well our algorithms will perform, although we will only discuss a few of them. Some of them are the following:

- Space Complexity
- Time Complexity

1.1 Space Complexity

Space complexity quantifies how memory hungry your program is. A program with high space complexity usually will require more memory in order to operate. Does it usually mean bad? In a general sense, yes. However, there are cases in which a high space complexity is needed to reduce the time for the program to execute and finish its instructions. Calculating a program's space complexity is hard, however, due to the dynamicity of Python, and the fact that we are writing

in pseudocode which closely resembles Python (and Java). As such, we will not delve deeper into analyzing an algorithm's space complexity.

1.2 Time Complexity

Time complexity quantifies how quick (or slow) an algorithm is in finishing its set of instructions to solve the task. It usually involves compilation and execution time. For this course, we will only focus on an algorithm's runtime, usually denoted as $T(P)$, where P refers to some program you have created.

Note that the actual runtime of a program is truly dependent on the programming language utilized, OS, and hardware, which is very hard to be analyzed. As such, every discussion regarding the time complexities of algorithms during this course is abstracted and simplified.

1.2.1 Runtime Calculation

To calculate the runtime of an algorithm, we refer to the following criterias:

1. The assignment statement:

   $$y = x$$

   has a running time of 2, where one count refers to the time taken to fetch the identity and value of some variable x (or any constant), and the other count refers to the time taken to bind y over the identity/value of x inside the memory. Note that the below statement also has the same amount of time taken to execute as the statement above:

   $$y = 5$$

2. The time required to execute an arithmetic operation such as addition, subtraction, division, multiplication, and comparison, are all constants. By applying rules 1 and 2, we can determine the runtime of the following statement:

   $$y = y + 1$$

   The runtime of the code above is 4. The count of the runtime is broken down into the following:

   - Two fetch operations to fetch the identities and/or values of y and 1,
   - One arithmetic operation to execute the addition operation, and
   - One store operation to store the value of the sum into y.

3. The time required to call a method or function is also constant. This operation is very similar to the store operation in criteria number 1. If a method or function has parameters, however, the time it takes to call that function is:
   - One operation to call the method or function
   - n constant fetch operations depending on the amount of arguments supplied
   - m constant store operations depending on the amount of arguments to be supplied

   For example, the statement:

   $$y = sum(1, 5)$$

   has a total runtime of $T_{sum()} + 6$, where $T_{sum()}$ refers to the runtime of the function $sum()$, and the breakdown of the runtime constant 6 is:

- One method call operation
- Two fetch operations to get the identities of 1 and 5
- Two store operations to store the arguments, 1 and 5, into the function $sum()$ for its use, and
- One store operation to store the result of the $sum()$ function into y.

Note that the final runtime calculation will depend on the runtime $T_{sum()}$ has, which is currently unknown to us. Later on, you will be exposed to some scenarios where you have to calculate the exact runtime of a program with all the methods and statements alike.

4. Array subscripting operations has varying runtime constants depending on the dimensions of the array and the values of every element inside that array. For now, the runtime calculations will primarily lean on **primitive values** like 150, 1.005, and *True*. So how do we calculate it then?

   An array (or list in Python) is a sequence of data where each element is stored in the memory. So, every time some statement like $A[5]$ is being called, a subscripting operation is being executed for the program to *access* the memory location of the element from array A at index 5, then retrieve the value of that element. All of these ***memory access*** complexities are abstracted and we simplify the array subscripting operation to have a count of 1.

   For example, the statement:

   $$y = arr[i]$$

   has a total runtime count of 5, which is broken down into the following:

   - Three fetch operations to fetch the following:
     - Identity of array arr
     - Identity of variable i
     - Identity of the element, $arr[i]$
   - One subscripting operation
   - One store operation to store the value of $arr[i]$ into y.

5. Loops always have a runtime of n times plus some constant c, where c is the sum of all the constant operations (store, fetch, arithmetic) during the loop execution. This is usually the case except for the loop's header or sentinel. An example of a loop header is:

   $$\text{while } i \leq n \text{ do:}$$

   In the example above, i is the loop's control variable, and n is the sentinel the keeps a check on i. The loop condition is $i \leq n$.

   A loop's header or sentinel has a runtime of $n + 1$, where n is the amount of times the loop's body will be executed, and the constant 1 refers to the last loop where the loop header is executed, checks the loop condition, and sees that the loop condition is **false**, and the loop is terminated.

2 Examples

In this part, we apply the four runtime calculation criterias to calculate the runtime of a particular algorithm.

2.1 Factorial

We already have our factorial algorithm from Topic 1, though however, it is done recursively. The algorithm specified below is an iterative approach and we will use it to apply our criterias in calculating an algorithm's runtime.

Algorithm 2.1 Iterative Factorial

```
1 function factorial(n) :
2       result := 1
3       i := 1
4       while i ≤ n do:
5             result := result * i
6             i := i + 1
7       return result
```

The statements that are executable from *Algorithm 2.1* ranges from lines 2 - 7. Note that the *return* keyword on line 7 also generates one count of constant runtime, whose logic is similar to the ***store*** operation.

We then calculate the runtime of *Algorithm 2.1* using Table 2.1 below.

Table 2.1: Algorithm 2.1 Runtime Calculation

statement	code	count
2	$result := 1$	2
3	$i := 1$	2
4	while $i \leq n$ do:	$4 * (n + 1)$
5	$result := result * i$	$4 * n$
6	$i := i + 1$	$4 * n$
7	return *result*	2
Total		$[4 * (n + 1)] + [2 * (4 * n)] + 6$

We then evaluate the entries from **Table 2.1** through the following:

$$\begin{aligned} \textbf{Total} &= [4 * (n + 1)] + [2 * (4 * n)] + 6 && (1) \\ &= (4n + 4) + 8n + 6 && (2) \\ &= 12n + 10 && (3) \end{aligned}$$

Therefore, ***Algorithm 2.1*** has a runtime of $12n + 10$.

2.2 Reversing An Array

The next section will utilize the array subscripting operation when calculating algorithm runtime.

To reverse an array, A, we have to swap the elements in every index from 0 to $A.length() - 1$. Therefore:

1. Set i as the index from the left
2. Set k as the index from the right
3. Swap the elements in $A[i]$ and $A[k]$.
4. Do until $i > k$.

Algorithm 2.2: Array Reversion

```
1 function reverse(A) :
2     // Index from left and right respectively
3     i := 0
4     k := A.length()
5     while i < k do:
6         temp := A[i]
7         A[i] := A[k]
8         A[k] := temp
9         i := i + 1
10        k := k − 1
```

The statements that are executable from *Algorithm 2.2* (aside from the comments) ranges from lines 3 - 10. We then calculate the runtime of *Algorithm 2.2* using Table 2.2 below.

Table 2.2: Algorithm 2.2 Runtime Calculation

statement	code	count
3	$i := 0$	2
4	$k := A.length()$	3
5	while $i < k$ do:	$3 * (n + 1)$
6	$temp := A[i]$	$5 * n$
7	$A[i] := A[k]$	$5 * n$
8	$A[k] := temp$	2
9	$i := i + 1$	4
10	$k := k - 1$	4
Total		$[3 * (n + 1)] + [2 * (5 * n)] + 15$

You might be wondering about the *return* statement for this function. So where is it? In this scenario, there is no need to return array A since whenever the reverse() function is called, the array supplied in the parameter of reverse() is passed on by ***reference***.

Note that whenever a variable is ***passed by reference*** into a function, any changes to that variable within the function will directly reflect back to the variable itself. As such, as we reverse the elements

in A within reverse(), all swaps executed will reflect to array A without the need to returning the array itself.

Now, let's calculate the runtime of **Algorithm 2.2**.

$$\begin{aligned} \textbf{Total} &= [3 * (n+1)] + [2 * (5 * n)] + 15 && (4)\\ &= (3n+3) + 10n + 15 && (5)\\ &= 13n + 18 && (6) \end{aligned}$$

Therefore, the runtime of Algorithm 2.2 is: $13n + 18$.

3 Runtime Complexity

Now that we are able to calculate the runtime of algorithms, we must then be able to evaluate whether such runtimes are ***efficient*** or not in the context of **time-based** execution. We can do this by utilizing the **Asymptotic Notations**: O, Ω, Θ.

3.1 Best-Case Scenario (Ω)

The first asymptotic notation, Ω, refers to the runtime complexity of an algorithm when it is exposed in a best-case scenario. What is a best-case scenario exactly? Let's refer to the following example:

> L is a list containing names of people, and p is a string for the name of the person you want to search. Coincidentally, $L[0]$ contains the name of the person you are looking for!

This is a best case scenario since the search algorithm's target is located at the first element of the list! This means that at the first iteration of the search algorithm, it will be able to find the value that the user needs.

Best-case scenarios are mostly unimportant, however, as it doesn't fully evaluate the ***efficiency*** of an algorithm. For this course, best-case scenarios will not be discussed.

3.2 Average-Case Scenario (Θ)

The second asymptotic notation, Θ, refers to the runtime complexity of an algorithm when it is exposed in a *not so best case scenario* and also a *not so worst case scenario*. It is used to determine the average runtime of your algorithm, and does have significant weight in evaluating the efficiency of an algorithm.

For example:

> L is a list containing numerical values where some of them are sorted properly while the rest are not.

This scenario is an example of an average case since the list L is partially sorted and only a few of the elements are left to be sorted by the algorithm. Furthermore, we are unsure how many elements are partially sorted. Assuming that there are N elements in the list L, then there are n partially sorted elements, with the remaining m unsorted elements. If you observe carefully, the value of n

and m are unknown, which means that their values could vary depending on the data present in L. It might be that $n > m$, or $n < m$, or $n = m$.

The Θ notation can approximately estimate the runtime of the algorithm when exposed to the three different scenarios. The problem, however, is that computing the average-case scenario of an algorithm requires some advanced mathematical concepts like Calculus, and statistical concepts like Probability Theory, which is not covered by the BSIT program. For this course, average-case scenarios will not be discussed.

3.3 Worst-Case Scenario (O)

The last asymptotic notation, O, otherwise known as **Big-Oh**, refers to the runtime complexity of an algorithm when it is exposed in the worst possible scenario ever.

For example:

> x is the value you are looking for and you have a numeric list, L. Now, you are tasked to find x in L, which, unfortunately, is not on the list!

The scenario above is worst-case since you have to iterate over the entire list only to know that the value you are looking for is not there!

Worst-case scenarios are usually utilized over the rest when evaluating the ***efficiency*** of an algorithm due to how easy it is to calculate and the fact that it tests the limits of your algorithm when exposed to ridiculous data or conditions.

To denote the worst-case runtime complexity of an algorithm, we use the notation: $\Theta(T)$, where T refers to the highest degree (variable with the largest exponent) of an algorithm's calculated runtime. Before we proceed in using this notation, let us first discuss a few mathematical functions you need to know.

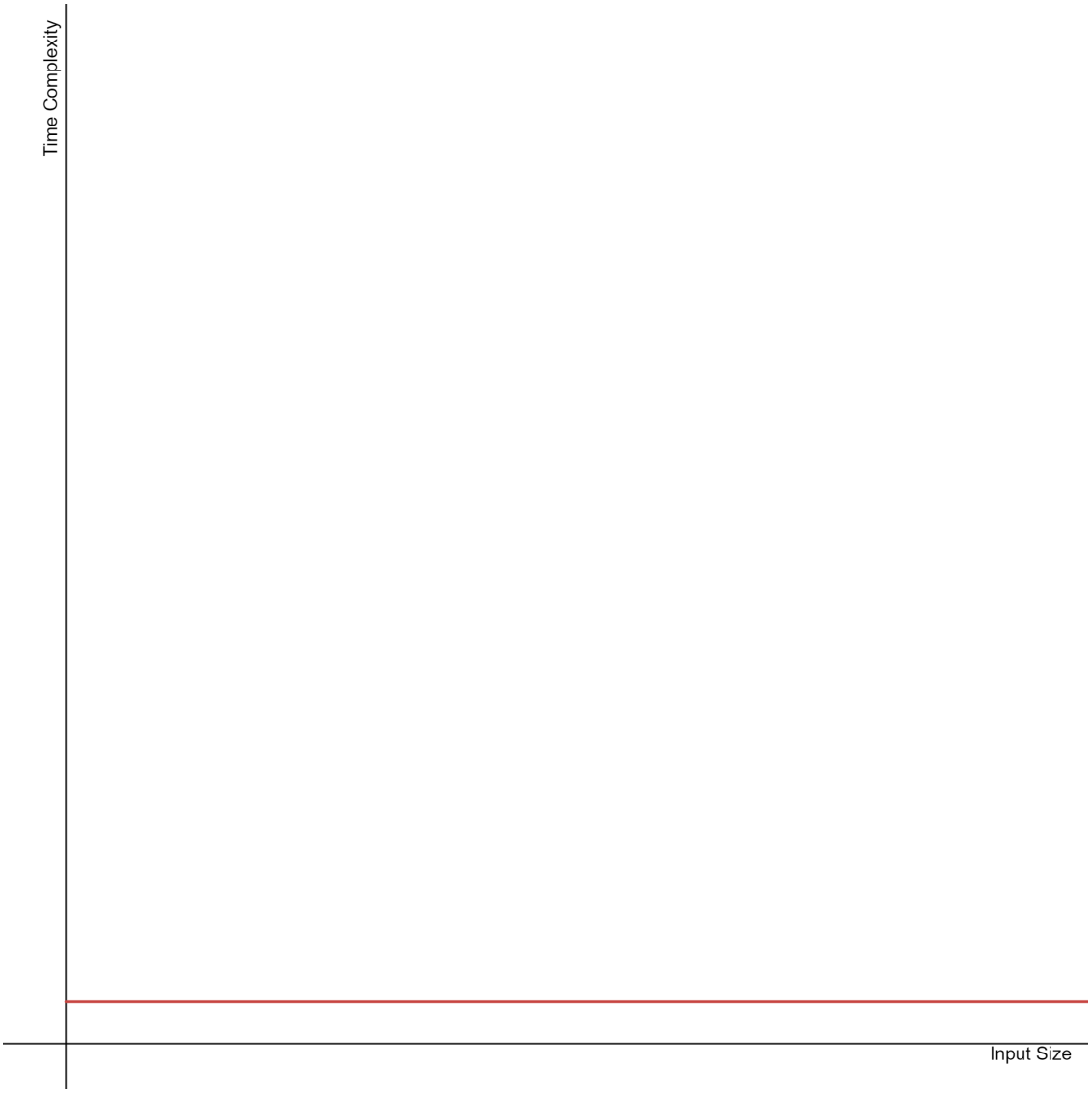
3.3.1 Mathematical Functions

There are seven mathematical functions that are proven to be useful in determining the efficiency of an algorithm when exposed to worst-case scenarios. They are the following:

1. **Constant Function**: $f(n) = c$

   The constant function contains a constant value, $c \in R$, and denotes that the graph of $f(x)$ is a straight horizontal line across the **x-axis**. In the context of computer programming, constant functions usually denote operations like: *store*, *fetch*, *arithmetic operations*, *function/method call*, and other quick operations. Some algorithms can be made constant, albeit the complexity of doing so is high as well.

The figure below represents the graph of a constant function. Refer to the straight red line.



As observed, if an algorithm is ***constant***, then regardless of its inputs, it will always

perform similarly. One such example would be the ***Sum Equation*** of the first n natural numbers. Note that the natural numbers is a set of numbers, $\mathbb{N} = \{1, 2, 3, ...\}$. Mathematically, the sum of the first n natural numbers is:

$$S_n = \frac{n(n+1)}{2}, n \geq 1$$

Here, n refers to the amount of numbers to be summed. However, if we look back to the *runtime calculation* of algorithms, every operation that will be executed (subtraction, multiplication, and division) in the ***Sum Equation*** are all constant. As such, the *Sum Equation*'s runtime complexity is equivalent to $O(1)$ regardless of the value of n.

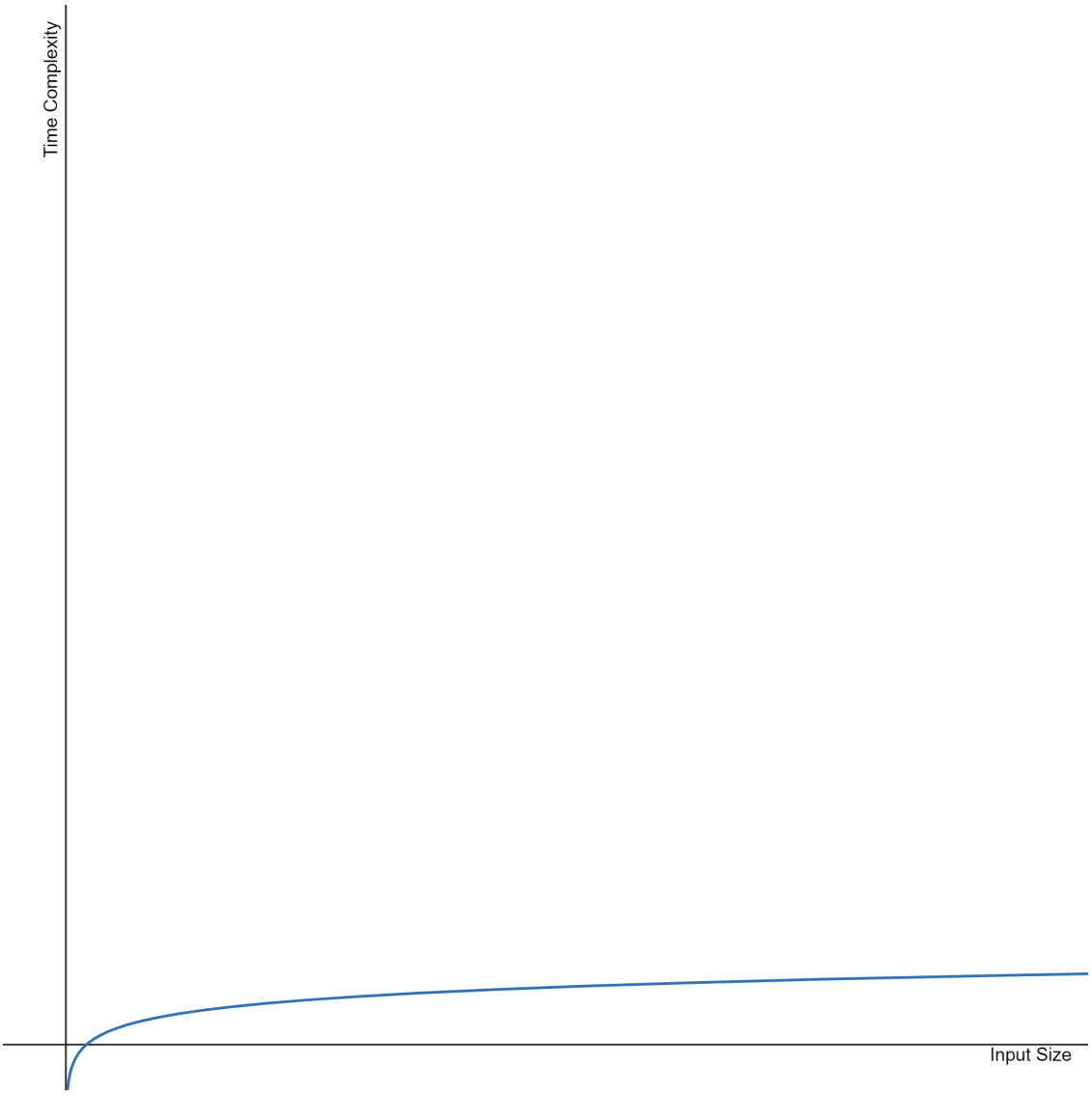
2. **Logarithmic Function**: $f(n) = \log_a n$

   Logarithmic functions follow a curving graph with short increase in steps. Very few algorithms can reach the behavior of a logarithmic growth since optimizing such operations requires a lot of mathematical and logical knowledge to shorten the time it takes to execute the instructions. Even if they exist, there are some pre-requisites in using those algorithms due to the logic they possess.

   A common example of an algorithm that follows the behavior of a logarithmic function growth is ***Binary Search***, a ***divide and conquer*** search algorithm that only works on specific conditions and data structures. Its runtime complexity behaves like a logarithmic function due to the fact that in *Binary Search*, the amount of items being searched in every iteration reduces by half the size (divided by 2). In other words, algorithms that has a runtime complexity of $O(\log n)$ are algorithms which drastically reduce the input size by at least half.

   Do note that in the context of computer programming, the base a of a log function is usually $a = 2$, thus, a log expression $\log n$ usually means: $\log_2 n$.

   The figure below illustrates the behavior of a logarithmic function in a graph.

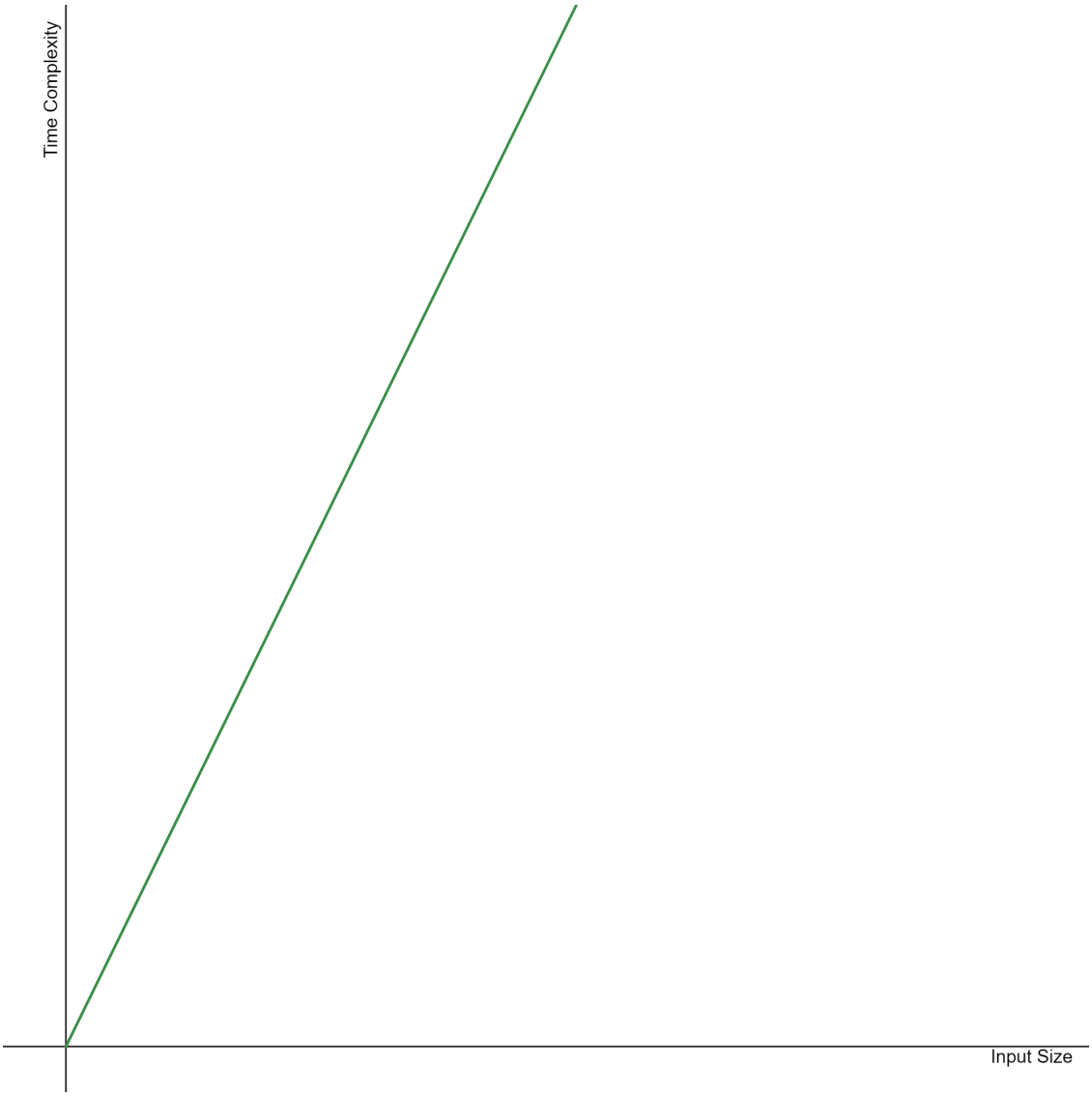


3. **Linear Function:** $f(n) = n + c$

   Linear functions follow a straight line with varying increase in steps. Most algorithms are linear such as ***Linear Search***, which is a worse algorithm than *Binary Search* in terms of runtime, but is much more *efficient* than it in most cases. The reason for this is that *Binary Search* requires a given data structure to be sorted properly, which adds additional runtime to the *Binary Search* algorithm since that data structure needs to be sorted first before the *Binary Search* algorithm could be used.

   To determine whether an algorithm has a *linear* runtime complexity, the most notable characteristic is that the algorithm's process increases as the number of input size increases. For example, in ***Linear Search***, it goes through the entire list from index 0 to $n - 1$. If n increases, then that means that the linear search algorithm will have additional processes equivalent to the amount added to n.

The figure below illustrates the behavior of a linear function.

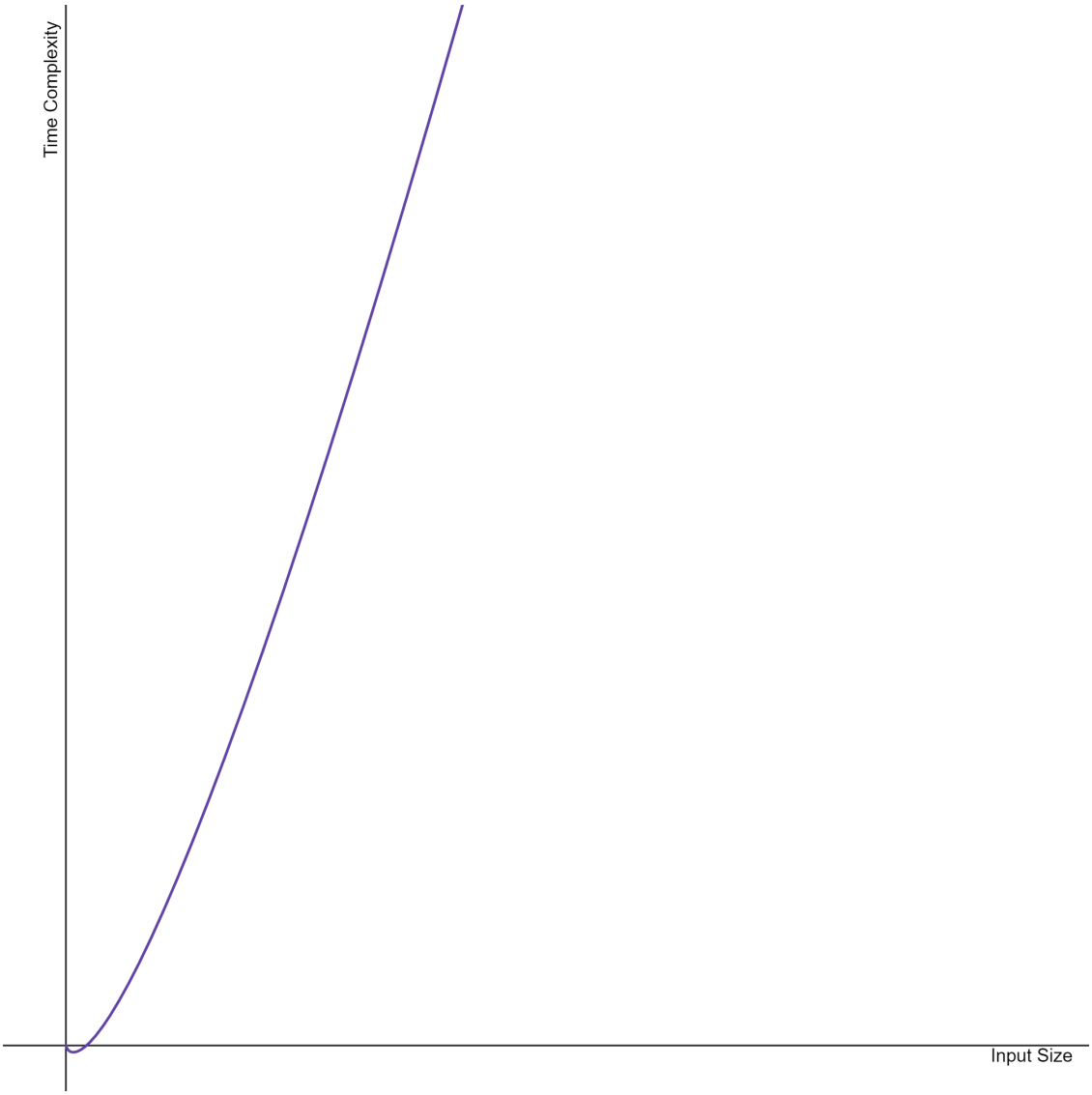


As you can see, the time complexity of a *linear function* drastically increases compared to your logarithmic function.

4. $N \log N$ **Function:** $f(n) = n \log n$

   $N \log N$ functions are a combination of a linear and logarithmic function. **Quicksort** is an example of an algorithm who has a runtime complexity behavior similar to $N \log N$. It is a combination of divide and conquer and linear iteration, and the concept is rather confusing right now. We will deal with Quicksort in the next topic with array lists.

   The figure below illustrates the behavior of an $N \log N$ function.



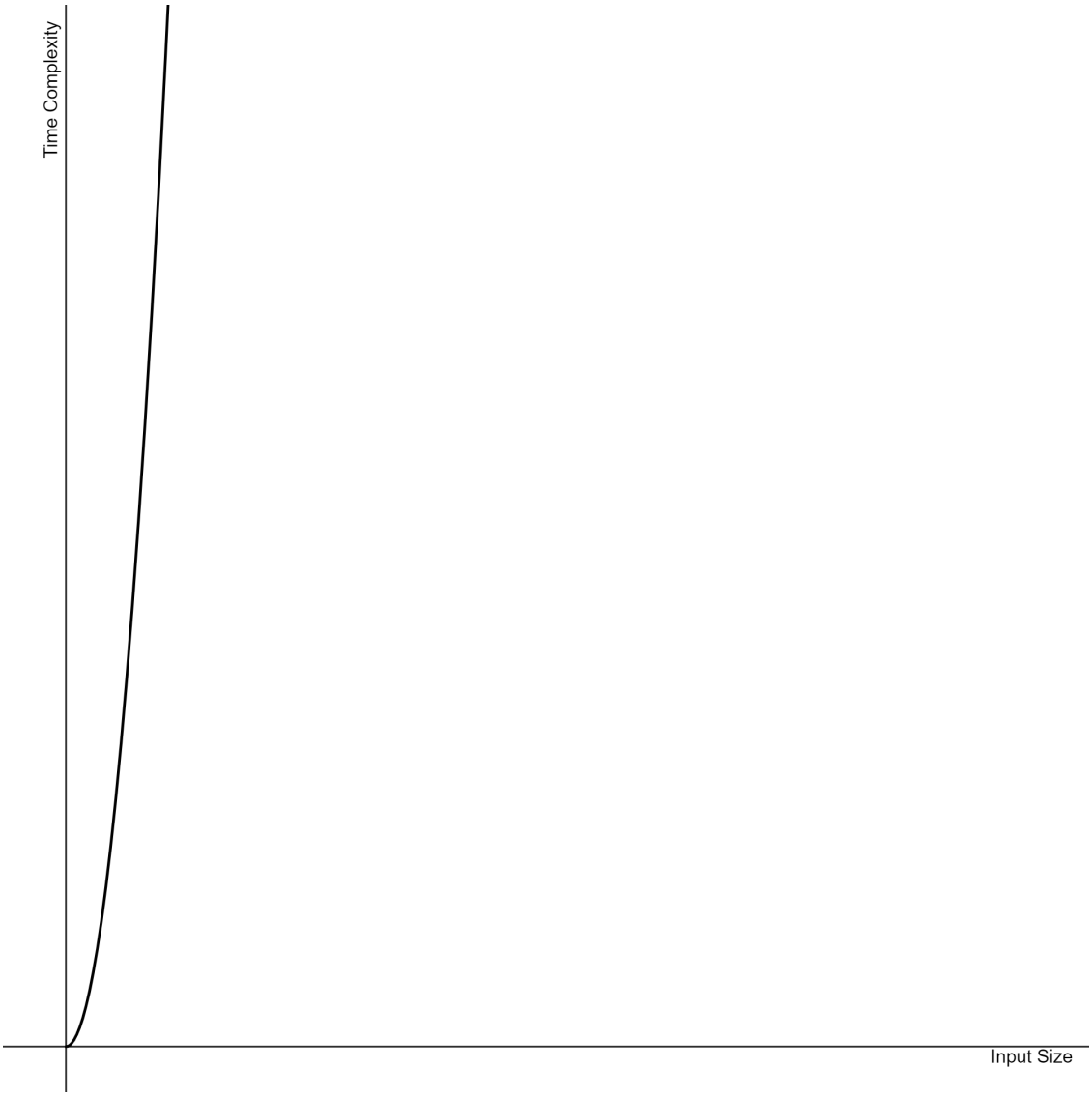
5. Quadratic Function: $f(n) = n^2 + c$

   Quadratic functions follow a *parabolic* behavior in its graph, and when translated into programming context, a super imposition of two linear algorithms atop one another. An example of a quadratic function is ***Selection Sort***, which has been discussed previously in **Topic 1**.

   Selection sort is quadratic because it is composed of two loops where: * The outer loop that iterates over the entire array, with $i := 0$ up to $array.length()$. * The inner loop that iterates over the entire array from i up to $array.length()$ and is executed every time the *outer loop* iterates.

   Algorithms that are *quadratic* in nature are less efficient than their Linear and $N \log N$ counterparts, and are sometimes called ***brute-force*** algorithms due to their inefficient nature.

The figure below illustrates the behavior of a quadratic function in a graph.

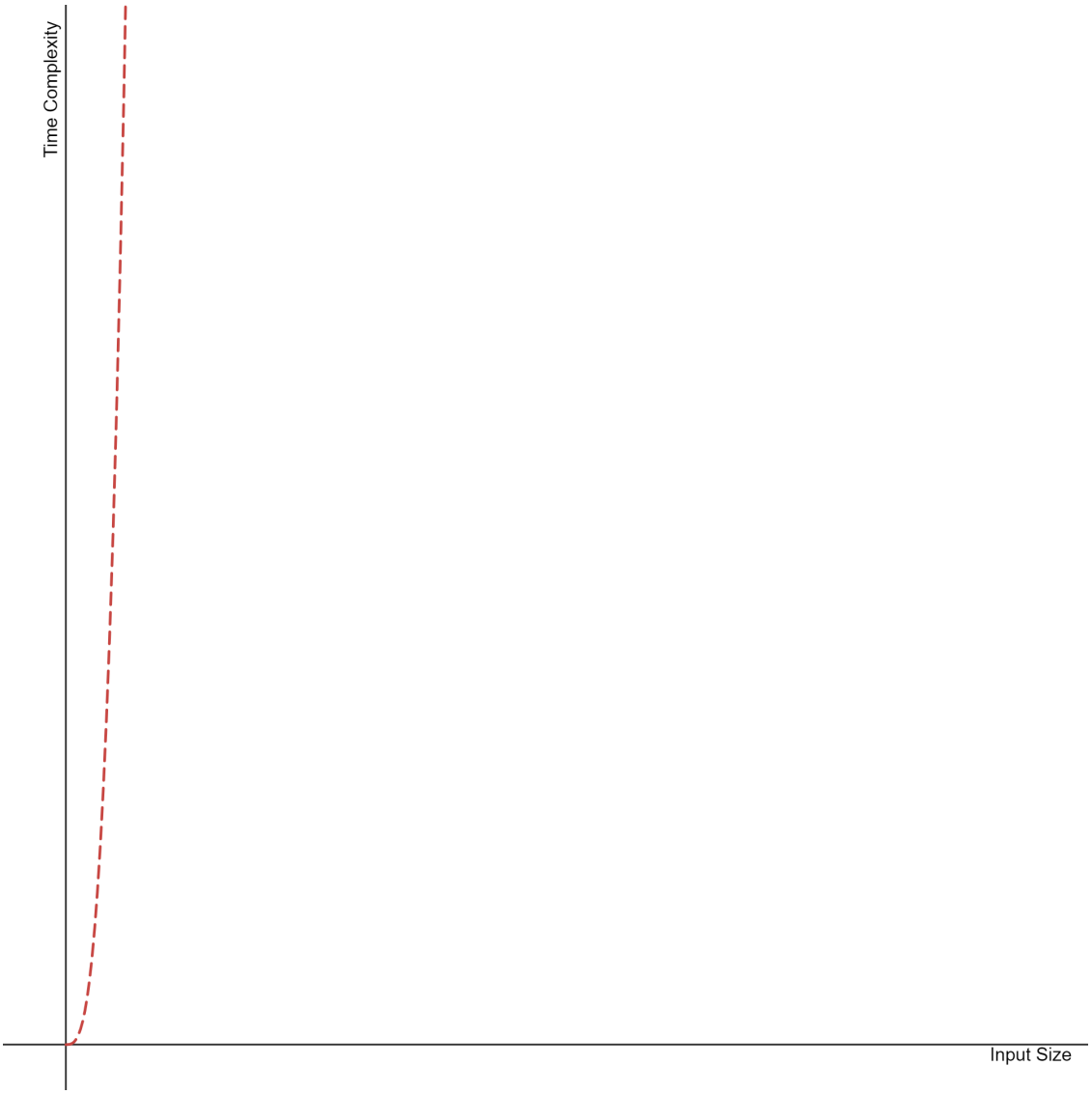


6. **Cubic Function:** $f(n) = n^3 + c$

   Cubic functions are even more inefficient compared to their quadratic counterparts, and when something is cubic, it generally implies that an algorithm has ***3 loops*** in total where there is an: (1) outer loop, (2) middle loop, and (3) inner loop.

   One example of a cubic function is ***Bubble Sort***, where, if implemented in the most inefficient manner, would require three loops to properly sort an array's elements. It is not recommended that your algorithms should reach the threshold of being cubic as that would mean that your program will run very slow when it is executed.

   The figure below illustrates the behavior of a cubic function in a graph.



7. **Exponential Function:** $f(n) = x^n + c$

   An exponential function generally is a result from "recursive" algorithms that tend to branch out further and further as more cases are introduced now and then. A famous example would be the ***Fibonacci Algorithm*** especially if it is implemented recursively. Mathematically, a **Fibonacci** sequence is defined as:

   $$F_n = F_{n-1} + F_{n-2}, \text{ where } F_0 = 0 \text{ and } F_1 = 1$$

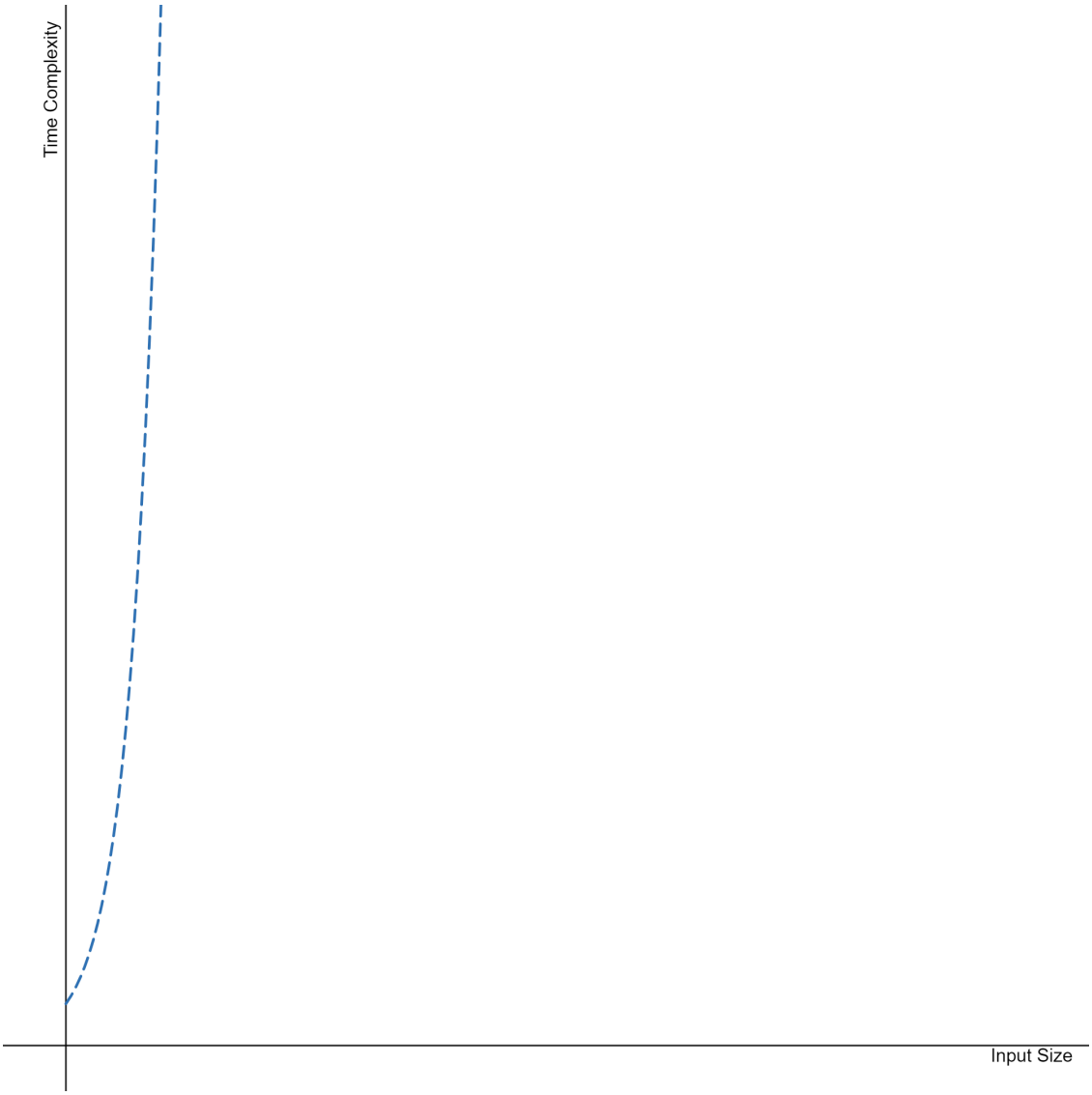
   Note that when we calculate F_n, we must first find the previous terms, F_{n-1} and F_{n-2}. * What is F_{n-1}? That would be F_{n-2} and F_{n-3}. * What is F_{n-2}? That would be F_{n-3} and F_{n-4}.

   Note that the amount of values to *calculate* as we go down until F_0 and F_1 increases by two. If you trace it out on paper, you would observe that the structure of calculating

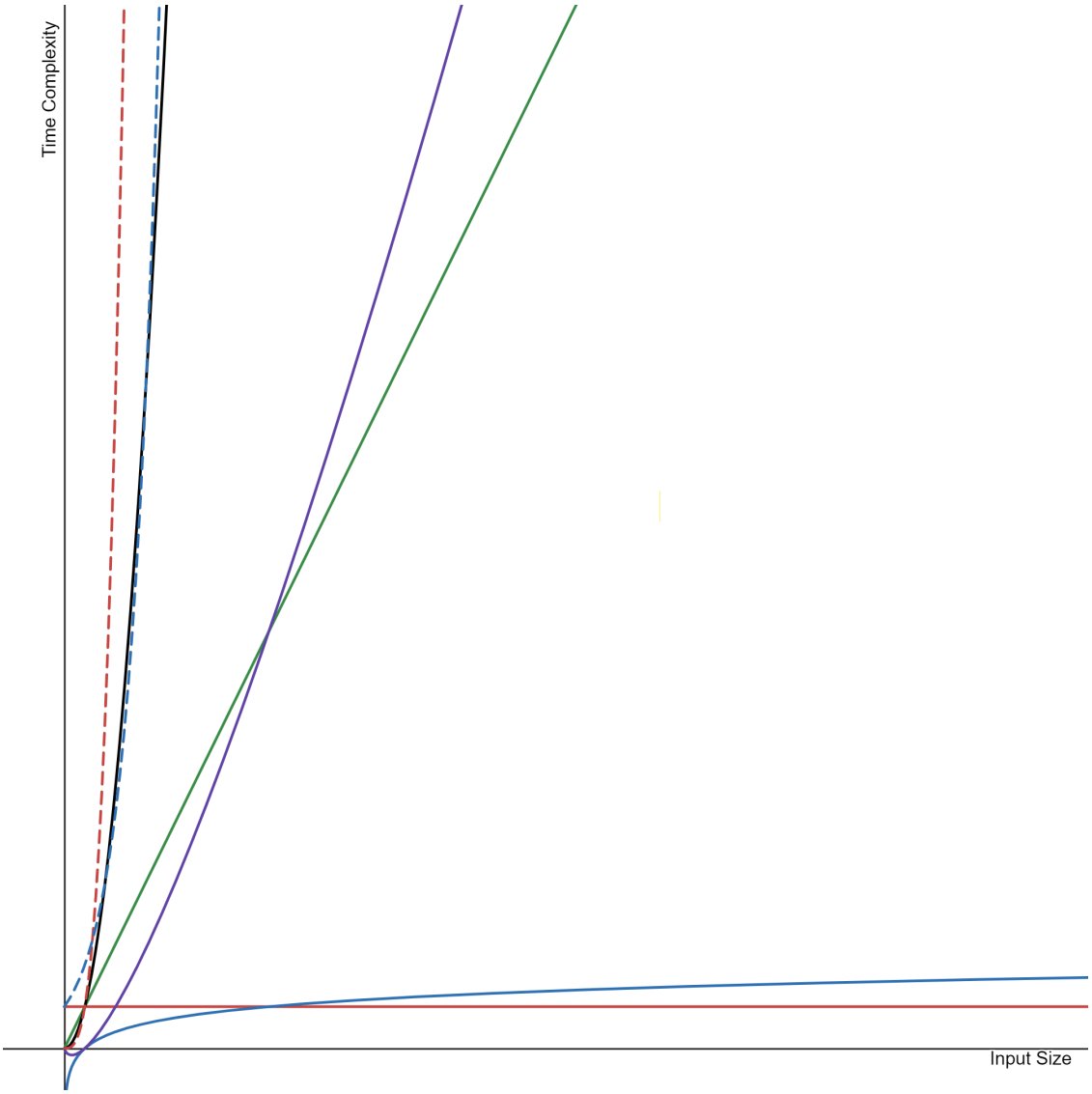
a Fibonacci term follows what seems to be a *Binary Tree* illustration. ***Exponential*** algorithms tend to follow this type of trend and if you happen to implement one, then that algorithm is bound to be labeled as *inefficient*.

Similar to *cubic* algorithms, it is not recommended that the algorithms and programs you create later on would reach such runtime complexities. So how do we avoid using recursion if this is the case? That question will be answered once we discuss **Dynamic Programming** in the next topic.

As for now, refer to the figure below that illustrates the behavior of an exponential function in a graph.



If we merge all graphs together, then we have the following figure:



Note that even though *cubic* seems worse than *exponential*, that is only the case on this graph. If we were to increase the size of the graph, then we would see that *exponential* algorithms would perform worse as our input size increases.

3.4 Rules when Calculating Big-Oh

Before we calculate the Big-Oh complexity of algorithms, there are some things we have to consider:

1. When calculating a loop's worst-case, always assume that every line in that loop will be executed.
2. When conditional statements are present such as: >if $\langle condition \rangle$:
   S_1
   else:
   S_2

The runtime would be whichever *conditional* statement has the maximum runtime calculation. However, when a loop contains a break statement inside it, skip the counting of that loop and only focus on other conditional statements.

3. The Big-Oh complexity of an algorithm, $O(x)$, will be the variable with the highest degree/exponent, ignoring all coefficients, divisors, and constants.

4 Examples

In this section, we will calculate the runtime and Big-Oh complexities of algorithms.

4.1 Array Reversion

Previously in **Table 2.2**, the runtime of the **Algorithm 2.2** is totaled as $13n+20$. Based on **Rules in Calculating Big-Oh** item number three, we have to ignore all coefficients and constants in calculating its Big-Oh complexity. Thus, **Algorithm 2.2** has a Big-Oh complexity of: $O(n)$. This means that in the worst case possible, **Algorithm 2.2** will run n times.

4.2 Selection Sort

Selection sort algorithm has been given back in Topic 1. Let us review its pseudocode here, revised into the structure of a while-loop:

Algorithm 2.3 Selection Sort

```
1   function SelectionSort(a) :
2   // Sort the array a from a[1 : n]
3       i := 0
4       while i < a.length() do:
5           // Index of the minimum element from i to n
6           j := i
7           k := i
8           while k < a.length() do:
9               if (a[k] < a[j]) then j := k
10              k := k + 1
11          temp := a[i]
12          a[i] := a[j]
13          a[j] := temp
14          i := i + 1
```

We have to consider several things here. Note that in line number 9, there exists a conditional statement that in general, would not be run all the times. However, since we are considering the Big-Oh complexity here, we have to assume that the conditional statement in line number 9 will always be executed. Then, we have the following runtime calculation of each line (except comments):

statement	code	count
3	$i := 0$	2
4	while $i < a.length()$ do:	$4 * (n + 1)$
6	$j := i$	$2 * n$
7	$k := i$	$2 * n$
8	while $k < a.length()$ do:	$[4 * \left(\frac{n(n+1)}{2}\right)] + 1$
9	if $(a[k]) < a[j])$ then $j := k$	$11 * (\frac{n(n+1)}{2})$
10	$k := k + 1$	$4 * (\frac{n(n+1)}{2})$
11	$temp := a[i]$	$5 * n$
12	$a[i] := a[j]$	$5 * n$
13	$a[j] := temp$	$2 * n$
14	$i := i + 1$	$4 * n$

To summarize, we have:

$$\begin{aligned}
\textbf{Total} &= [4 * (n + 1)] * n + [(11 * n) * n] + [(4 * n) * n] + [4 * (n + 1)] + [3 * (2 * n)] + \\
&\quad [2 * (5 * n)] + (4 * n) + 2 && (7)\\
&= [(4n + 4) * n] + 11n^2 + 4n^2 + 4n + 4 + 6n + 10n + 4n + 2 && (8)\\
&= 4n^2 + 4n + 11n^2 + 4n^2 + 4n + 4 + 6n + 10n + 4n + 2 && (9)\\
\textbf{Total} &= 19n^2 + 28n + 6 && (10)
\end{aligned}$$

Note that the variable with the highest degree / exponent is $19n^2$. Based on the rules when calculating Big-Oh, we ignore the coefficients and contstants. As such, the Big-Oh complexity of the algorithm is $O(n^2)$, which means that the selection sort algorithm has a behavior of a quadratic function when exposed to the worst possible situation.

You might be wondering why lines 8 - 10 has the expression, $\frac{n(n+1)}{2}$. That is because the loop's number of iterations increases over time. This means that on the first iteration of the outer loop, the inner loop may only iterate 1 time. However, during the second iteration of the outer loop, the inner loop may iterate 2 times, an increment of the previous iteration.

You need to take note that if you discover that a loop's iteration increases or decreases every time, then its loop header will have the runtime of: *constant operations* $\times \frac{n(n+1)}{2} + 1$. Also, take note that loop body will always be 1 less iteration to the header.