CO2412

Assessment1

Senan Yatigammana

G21017409

**Part 3**

**Part A**

We all write programs. Programs do different things, but they all have one thing in common, they use computer resources to do that task. Focusing on the time it takes for a certain bit of code to execute, some will execute very fast, and some will take longer. When writing programs it’s useful to keep in mind this time factor. If you’re dealing with very large amounts of data and your algorithm takes a long time to run, it’ll be a pain to use it.

Take these 2 mathematical formulas to add up numbers, *[1]*

*FunctionOne():*

*Total=0*

*For i in range(0,n):*

*Total +=i*

*FunctionTwo():*

*Return n\*(n+1)/2*

Both these functions are taken from <https://rithmschool.github.io/function-timer-demo/> (referenced below) shows 2 codes that do the same task but are implemented in different ways. The following graphs show the execution time for these functions

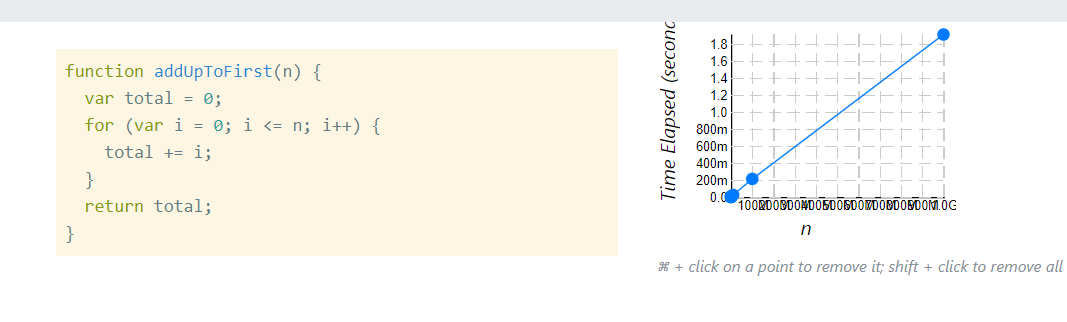


Figure 1 time complexity for function 1 in blue [1]

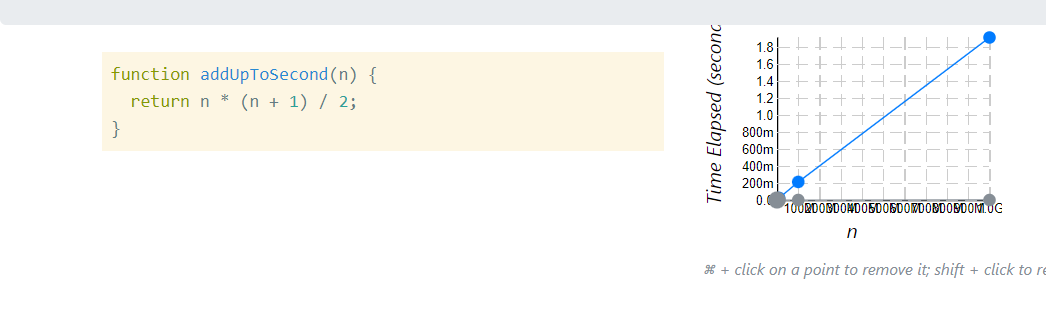


Figure 2-Time comp for 2nd algorithm in grey [1]

So codes should be written so that they run fast, but how would one know if their code is good or bad in this respect? Well, measure how long the code takes to run right? Wrong. There are a few problems that have been observed when recording time like this. Namely,

* Different machines will show different times for the same code
* The *same machine* will show different times for the same code.
* The code will execute so fast it’s hard to get a reliable measurement *[2]*

This is where “Big O” comes in. also written as Big Oh. What this is, is a notation, not a program or mathematical formula. It’s a way of saying whether your code is efficient or not. To use this, we don’t need to run the code or draw any charts or any of that. We can just look at the code, see how many times the operations must take place (for an n number of times), and then determine the big O notation value. Let’s look at the 2 examples above, in the first function, there is a ”for loop”, so when n increases, the program must also run “n times’ more. For the 2nd function, there are a set number of operations regardless of the size of n. there’s multiplication, addition, and division. So no matter how big n is, the program runs only once, thus making it the better algorithm. *[2]*

Following are the notations of big O notation;

f(n) =1 -constant time O(1)

f(n)=n linear time O(n)

f(n)=n2 quadratic time O(n2) *[3]*

**Part B**

the following are the times recorded by the “time()” function, averaged over 5 runs:

LIST1

Quick Sort list1: 0.0155

Selection Sort list1: 1.505

Merge Sort list1: 0.009

LIST2

Quick Sort list2: 0.147

Selection Sort list2: NA\*

Merge Sort list2: 0.119

LIST3

Quick Sort list3: 2.175

Selection Sort list3: NA\*

Merge Sort list3: 1.725

\*(selection sort doesn’t work for list 2 and list 3)

As we can see, the fastest algorithm is the merge sort function, but only just. The quick sort function is only slightly slower, with the selection sort algorithm taking a long time for list 1 and not working completely for list 2 and list3.

An algorithm with a quadratic time complexity is slow, which is what the selection sort function is. This is because there are 2 loops in the code here. The process through which selection sort works is explained in part C below. Let’s compare the merge sort and quick sort functions. The average time complexities for these 2 functions would be O(n \* log n), but quick sort has a worst-case complexity of O(n2). Quick sort keeps dividing arrays until atomic values are obtained, whereas in merge sort the array is divided into 2, and then the numbers are sorted from there. This is called the “Divide and conquer” method. Thus merge sort will be better for large data sets. In practical use, quick sort is preferred for sorting arrays because it doesn’t require a lot of space, while merge sort is preferred for linked lists because of merge sorts’ method of memory allocation. *[4]*

**Part C**

The time complexities of the algorithms are discussed below.

Quick sort ; how the pivot is chosen, will determine how fast the code will run. If the smallest element is chosen, we’ll consider that as the worst case, with a time complexity of O(n2). Whereas if the mid element is taken as the pivot, well get a best-case time complexity of O (n \* log n) *[5]*

Selection sort ; the process for selection sort is pretty straightforward, the loops iterate assigning the most recent lowest value as the lowest value, assigning that to the start, and repeating that until the array is sorted. So, from here, we can see that there are 2 loops, therefore it has a time complexity of O(n2). Making it slow.

Merge sort ; merge sort divides the main array into 2, and repeats that through recursion until single values are obtained. There aren’t any nested loops in this function so the time complexity can be described as O (n \* log n)

By looking at the time complexity values we can say quick sort should be a lot slower than merge sort because quick sort has a worst-case time complexity of n2, but due to how the algorithm works, the times are fairly close to that of merge sort *[6]*

*The following table shows a comparison between quick sort and merge sort.*

Graphical user interface, application

Description automatically generated

Figure Comparison of quick sort and merge sort [6]

**Part D**

(3 > = ∞)

Figure performance graph

As explained previously, merge sort is the fastest and this graph proves it. The time parity between quick sort and merge sort will grow bigger with increasing data sets because of the above-explained reasons as to why quick sort isn’t efficient for large values.

Selection sort does not give an output for the 2nd and 3rd lists therefore its curve tends to infinity

**References**

[1]

*Performance tracker choose a function and start plotting!* (no date) *Big O Introduction*. Rithm School. Available at: https://rithmschool.github.io/function-timer-demo/ (Accessed: December 10, 2022).

[2]

Steele, C. (2019) *Complete beginner's guide to big O notation*, *YouTube*. YouTube. Available at: https://www.youtube.com/watch?v=kS\_gr2\_-ws8&ab\_channel=ColtSteele (Accessed: December 10, 2022).

[3]

*Analysis of algorithms: Big-O analysis* (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/ (Accessed: December 10, 2022).

[4]

*Quick sort vs merge sort* (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/#:~:text=Partition%20of%20elements%20in%20the,(i.e.%20n%2F2).&text=In%20case%20of%20quick%20sort,equal%20parts%20in%20quick%20sort. (Accessed: December 10, 2022).

[5]

*Quick sort - javatpoint* (no date) *www.javatpoint.com*. Available at: https://www.javatpoint.com/quick-sort#:~:text=Time%20Complexity,-Case&text=The%20best%2Dcase%20time%20complexity,O(n\*logn). (Accessed: December 14, 2022).

[6]

*Difference between merge sort and quicksort* (no date) *Difference between Quicksort & Merge Sort - Interview Kickstart*. Available at: https://www.interviewkickstart.com/learn/quicksort-vs-merge-sort (Accessed: December 14, 2022).