

Big-O Notation

(CISS-110/111 William Jojo)

Overview

As we write programs we gain knowledge. This is especially true in an educational environment. We learn when to use certain constructs instead of others. We may also learn about scalability, execution time, memory consumption and I/O utilization. If we are truly blessed, our instructor even explains in detail why it is important to do something a certain way or to learn some seemingly obscure programming concepts like bitmaps, binary searches and hashing.

Nothing speak louder than numbers. When we are talking about the price of a car or the interest rate of a loan or how much that mortgage really costs after thirty years the numbers speak volumes. It is also true with our programs we can often determine how well a program is performing and even find ways to improve it based on behaviors or example data. Of course, there are times we cannot find the worst case and other times we are not given true data to test or perhaps the volume of data is poorly represented.

It is a great feeling when we can cut the memory consumption of a program to 1/16th of what it was. It is equally great to realize that we can cut the run time in half. Ultimately we need to look at our program and ask the question, “but what does it do?” Can we define a unit of work? Perhaps define the main operation or group of operations that make up the bulk of our program or even just an algorithm with a program. We must also keep in mind that sometimes it is not the obvious operations that we are looking for. Sometimes they are hidden and require digging deeper and sometimes we simply cannot define the operations in simply terms by which measurement is meaningful.

Once we establish the operation(s) that we wish to investigate, we need to find out how many operations will be performed on a given set of data. The data can be small for investigative purposes, but ultimately we need to ramp up the amount of data and really discover how many operations are necessary to process a data set of that size. So then given a amount of data represented by n we can that the number of operations required to process that data is on the *order of* some value.

Big-O Notation

Processing data and the number of operations required can be constant, logarithmic, linear, quadratic or even cubic. The last two forms mentioned can be devastating to a program managing large amounts of data. *Table 1* shows some basic values for n and the classifications mentioned.

n	Constant $O(1)$	Logarithmic $O(\log_2 n)$	Linear $O(n)$	Linear $O(n \log_2 n)$	Quadratic $O(n^2)$	Cubic $O(n^3)$
1	1	1	1	1	1	1
4	1	2	4	8	16	64
16	1	4	16	64	256	4,096
512	1	9	512	4608	262144	134,217,728
1024	1	10	1024	10240	1,048,576	1,073,741,824
4096	1	12	4096	49152	16,777,216	68,719,476,736
262144	1	18	262144	4,718,592	68,719,476,736	1.8e+16
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{18}

Table 1: Number of operations performed by a particular form of algorithm for values of n .

Note the use of $O()$ with a value in parentheses. This is the notation for saying *on the order of*. So if a particular algorithm is $O(n)$, we can say that the number of operations is on the order of n or that for n data items we need to perform n operations.

Note the stunning values in the lower right of the table. We would never want our program to enter this territory.

It is not necessary to be concerned with multipliers (except $n \log_2 n$) or differences. After all who would be impressed with an algorithm that measures $O(n^2 - 3n)$ when n is 1,024? That is 1,048,576 less 3,072. The $3n$ is hardly worth mentioning as it does not detract from the staggering fact that the work necessary is *three orders of magnitude larger* than the number

of data items.

This is also true for the constant case. If it is determined that for any value of n the program takes 32 operations to complete the work, we say that the algorithm is an $O(1)$ algorithm because as n approaches infinity, we will still only perform 32 operations.

What is an operation?

Now that we have discussed data sets, operations and notation, we should probably define an operation with some examples. *Table 2* shows some typical forms of work considered an operation with projected Big-O values. With *Table 2* as a guide, we will now talk some of the examples given.

Work related to an operation	O(?)	Comment
Linear search (array/linked list)	$O(n)$	
Binary search an array	$O(\log_2 n)$	Data must be sorted first.
Search a balanced tree	$O(1)$	
Search, add or remove operation on a hash table	$O(1)$	
Push/pop a stack	$O(1)$	
Insert into front or middle of array	$O(n)$	Array data must be adjusted.
Insert at end of array or front of linked list	$O(1)$	Assuming array is not full.
Insert into an ordered linked list	$O(n)$	Must find insert location.
Bubble sort	$O(n^2)$	
Quicksort/Heapsort	$O(n \log_2 n)$	Quicksort has worst case $O(n^2)$.

Table 2: Example work and the number of operations necessary to complete the work.

The linear search example should be pretty easy to see. Since the data is assumed to be an unordered array or linked list, searching the contents requires a comparison of each value in the array to the key value sought. The key here is that *the operation is the comparison*. The

best case is that we will find the key in the first position, but that is unlikely. The average number of values to be compared will be $n/2$, but as we stated before when the values are very large (think millions) constant multipliers do not usually add anything meaningful, there for the average case of linear search is the same as the worst case: $O(n)$.

Additional operations to arrays and linked lists include inserting into the middle or ordered data. Since this ultimately requires a search to find the insertion point, we already know this search is linear. Therefore, this algorithm is $O(n)$.

Binary searching an array of sorted values poses an interesting discussion. Since the data is in order, we can take the midpoint of the data make a comparison and move left or right. The comparison is the operation. Since we keep dividing the field in half we will swiftly exhaust the data in just a few operations. For example with 1000 items the first comparison reduces the field to 500, then second to 250 and the third comparison to 125. This means we are dealing with powers of two (hence the name *binary* search). This means that since we started with 1000, we will not need to compare more than 10 items. Why? 2^{10} is 1024. The exponent of 2 is what we are interested in. The exponent represents the number of comparisons since the field keeps getting cut in half. Therefore the binary search is $O(\log_2 n)$ since $\log_2 n$ gets us the exponent of the power of two that produces n .

Hash tables are designed with performance in mind. Inserting into a hash table is $O(1)$. This is because the bucket is immediately determined and new items are likely put on the front of the chain. With regards to searching and deleting, even if there were 32 items in each bucket it is still $O(1)$. What is interesting to ponder further is the fact that hashing a string is $O(n)$. This is because we will access each character of the string in the mathematical calculation. Is this significant? Probably not. Hashing is cheap and typically deals with less than a dozen characters, but it is a good opportunity to look deeper into an algorithm and see past the $O(1)$ and find an $O(n)$ portion of the overall solution.

Conclusion

When we consider using a built-in library or writing our own solution to a given problem, we must think about what the operation will be. We might measure how long an operation takes. We should also give some thought to what will happen to that library call or homemade solution when presented with very large data sets. If our code holds up and does not show a

dramatic slowdown, then we are well on our way to producing a well designed, scalable product. Otherwise we may need to consider some alterations or, at the very least, a disclaimer.