Analysis of Algorithms

Written by Rachel Singh, last updated Oct 17, 2018

About

We can measure the growth rate of an algorithm to help us understand how efficient it is, vs. other functions that may accomplish the same task.

Let’s look at growth rates, Big-O notation, etc.

Topics

1. Best, Average, and Worse-case Times

2. Analyzing Functions & Growth Functions

Best, Average, and Worse Case Times

1. Best, Average, and Worse Case Times

In general, when we're talking about functions that accomplish some task, we're often talking about a function that deals with a **list of values**.

We use the variable ***n*** as a general representation of the size of such a list.

Notes

1. Best, Average, and Worse Case Times

Let's say we have an algorithm that searches through all the elements of a list of size *n*, searching for some item. A simple linear search looks like this:

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

Notes

1. Best, Average, and Worse Case Times

Best case scenario, what we're looking for is **the first item in the list**. Importantly:

*** The loop starts, but only executes once. ***

The execution is *near instantaneous* in the best case scenario.

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

Notes

1. Best, Average, and Worse Case Times

Worst case scenario, what we're looking for is **the very last item in the list**. Importantly:

*** The loop runs exactly *n* times ***

So the amount of times it loops mirrors the size of the list 1:1; the amount of time taken will grow *linearly* as the list size grows.

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

Notes

1. Best, Average, and Worse Case Times

Most of the time, when we do a search our item isn't going to be exactly the first or exactly the last item – it will be somewhere inside the list.

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

Notes

1. Best, Average, and Worse Case Times

If we think of using our search over a long period of time, with a lot of different types of data, we can come up with some **average case time**.

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

Notes

1. Best, Average, and Worse Case Times

We have just one simple loop, and (for randomly placed data) we can assume sometimes the search time will be 1, or 2, or 3, up through n-2, n-1, and n items searched.

Python

```
def Search( myList, n, findMe ):
    for i in range( 0, n ):
        if myList[i] == findMe:
            return I

    return -1
```

C++

```
int Search( T myList[], int n, T findMe )
{
    for ( int i = 0; i < n; i++ )
    {
        if ( myList[i] == findMe )
            return i;
    }
    return -1;
}
```

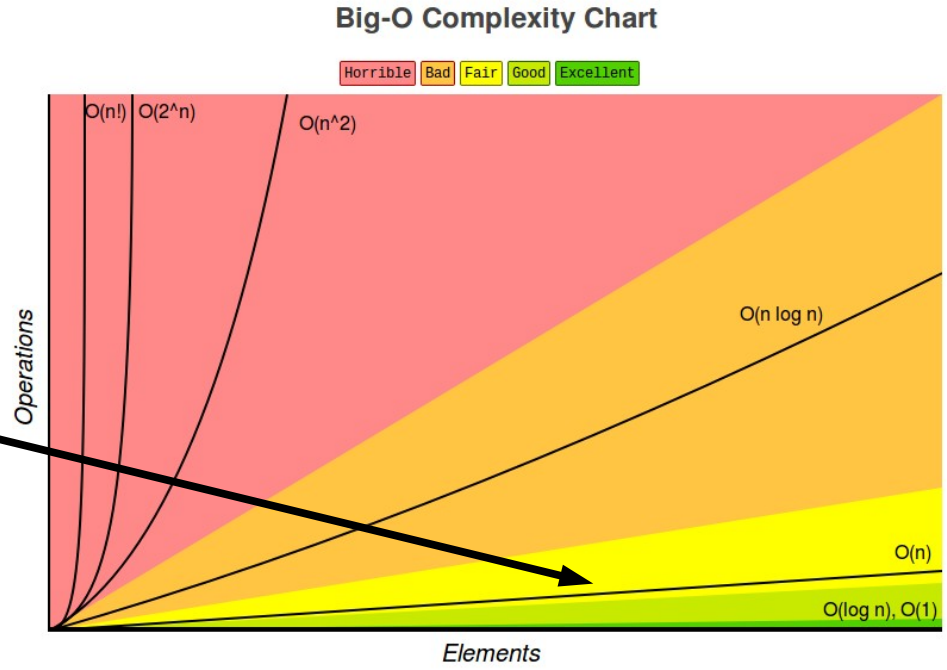
Notes

1. Best, Average, and Worse Case Times

We don't really care about the exact *number* of times we execute the loop, but rather **its relationship to the size of the array**.

For a linear search, as we increase the size *n*, the amount of time to search also goes up by a constant rate.

We call this a O(n), or linear, growth rate.



From http://bigocheatsheet.com/

Notes

Analyzing Functions & Growth Functions

2. Analyzing Functions & Growth Functions

How do we identify the efficiency of a given snippet of code? Here are some quick hints…

Notes

2. Analyzing Functions & Growth Functions

For some code that doesn't loop, and doesn't call another function that loops (or recurses), we can think of it as taking a negligible amount of time to execute.

Additionally, as the size of the array/list grows, these commands will take the same amount of time to execute – the time it takes to do a computation doesn't change based on the list size.

```
a = 2
b = 3
c = a + c
print( c )
```

O(1)

Constant

Notes

O(1): Constant

2. Analyzing Functions & Growth Functions

If a loop is involved, this adds execution time. While executing a simple line of code doesn't matter much, if we're looping and doing the command over and over, it adds time to execute.

A simple loop from start-to-end of a list (over *n* items), will have a linear O(n) growth rate.

As you add more items to the list (as *n* grows), the length of time to execute the function rises linearly.

```
def IsInList( myList, item ):
    for el in myList:
        if el == item:
            return True
    return False
```

O(n)

Linear

Notes

O(1): Constant
O(n): Linear

2. Analyzing Functions & Growth Functions

When we have a nested loop – so, one outside loop and one inside loop – we end up going through the list of elements *n*n* times – or n^2.

As the size of your list grows, the amount of time it takes to execute the function on it will grow quadratically.

```
def TimesTables( n ):
    for y in range( n ):
        for x in range( n ):
            print( x, y, x*y )
```

O(n²)

Quadratic

Notes

O(1): Constant
O(n): Linear
O(n²): Quadratic

2. Analyzing Functions & Growth Functions

So let's say we're analyzing three search functions created by three competing programmers, and we're going to license one of these.

We're going to test each search function with different sets of data and see how slow each one gets as we add more items…

Notes

O(1): Constant
O(n): Linear
$O(n^2)$: Quadratic

2. Analyzing Functions & Growth Functions

Amount of items	Programmer 1's Search	Programmer 2's Search	Programmer 3's Search
10	10 Time Units	100 Time Units	3 Time Units
100	100 Time Units	10,000 Time Units	7 Time Units
1,000	1,000 Time Units	1,000,000 Time Units	10 Time Units
1,000,000	1,000,000 Time Units	10,000,000,000,00 Time Units	13 Time Units

Here we have the results of the programmer's functions. Programmer 1's search mirrors the amount of items we feed in – it's clearly **linear**.

Notes

O(1): Constant
O(n): Linear
$O(n^2)$: Quadratic

2. Analyzing Functions & Growth Functions

Amount of items	Programmer 1's Search	Programmer 2's Search	Programmer 3's Search
10	10 Time Units	100 Time Units	3 Time Units
100	100 Time Units	10,000 Time Units	7 Time Units
1,000	1,000 Time Units	1,000,000 Time Units	10 Time Units
1,000,000	1,000,000 Time Units	10,000,000,000,00 Time Units	13 Time Units

Programmer 2's search grows much as the amount of items grows. This one is a **quadratic growth**.

Notes

O(1): Constant
O(n): Linear
$O(n^2)$: Quadratic

2. Analyzing Functions & Growth Functions

Amount of items	Programmer 1's Search	Programmer 2's Search	Programmer 3's Search
10	10 Time Units	100 Time Units	3 Time Units
100	100 Time Units	10,000 Time Units	7 Time Units
1,000	1,000 Time Units	1,000,000 Time Units	10 Time Units
1,000,000	1,000,000 Time Units	10,000,000,000,00 Time Units	13 Time Units

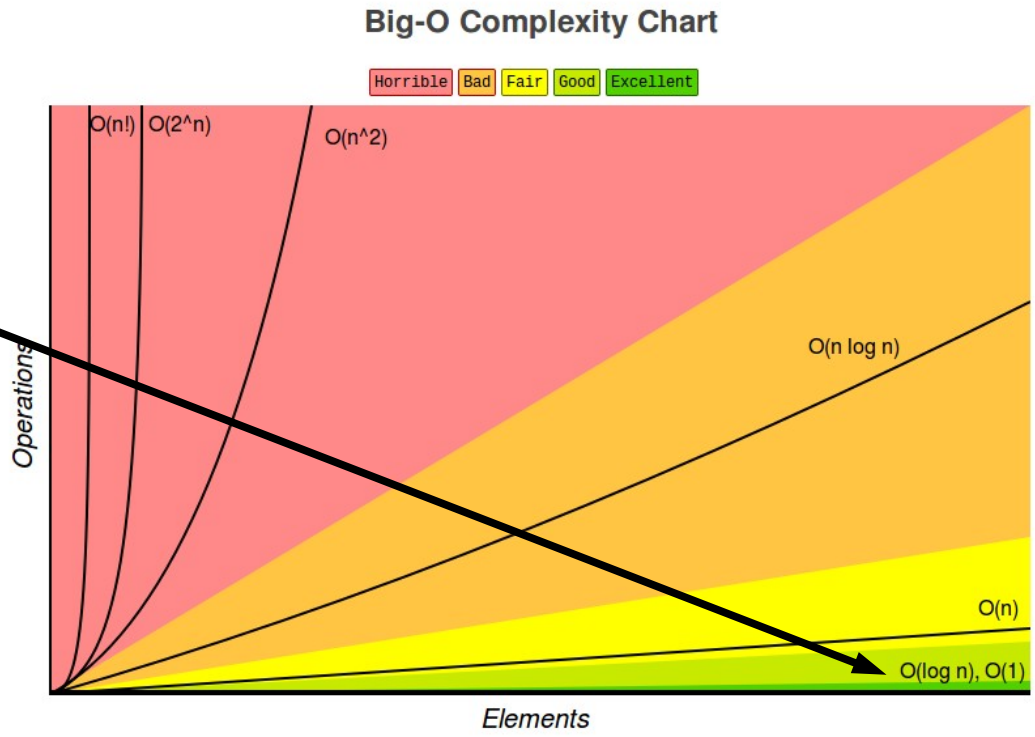
Programmer 3's function seems incredibly efficient! Barely increasing in time as the list grows. This clearly isn't linear, but it isn't instantaneous, either. **What's better than a linear growth rate function?**

Notes

O(1): Constant
O(n): Linear
$O(n^2)$: Quadratic

2. Analyzing Functions & Growth Functions

A logarithmic function is better than a linear one!



From http://bigocheatsheet.com/

Notes

O(1): Constant
O(n): Linear
O(n^2): Quadratic

2. Analyzing Functions & Growth Functions

Certain types of structures will have a quick search time!

But you'll learn more about this in another class…

Data Structure	Time Complexity			
	Average			
	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Hash Table	N/A	O(1)	O(1)	O(1)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))

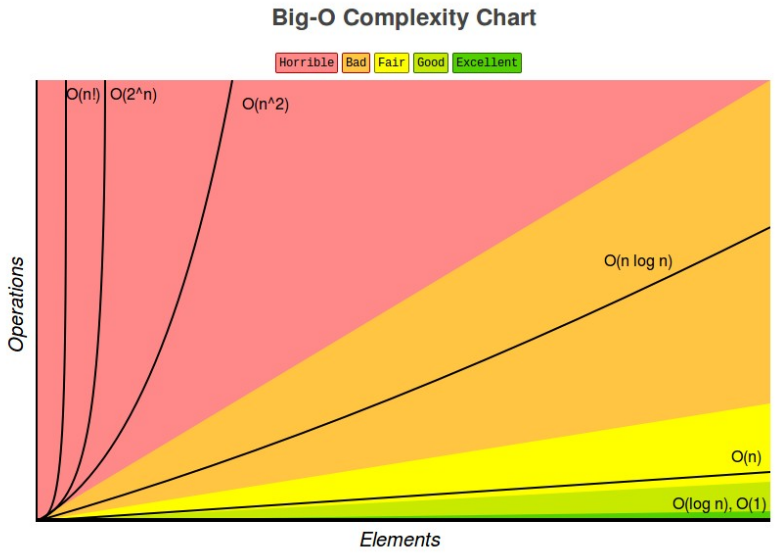
From http://bigocheatsheet.com/

Notes

O(1): Constant
O(n): Linear
$O(n^2)$: Quadratic

Conclusion

For the most part, it’s easiest to analyze for constant, linear, and quadratic time.
Logarithmic time would be easier to talk about if you’ve seen the Tree structure before.



From http://bigocheatsheet.com/