Time And Space Complexity

Created	@July 4, 2022 7:59 PM
Class	
Type	
Materials	
Reviewed	☐

Why we care about time and space ?

So when we used to go and buy new PC, what were the things we used to look out for, while getting our first PC.

- A good processor
- Enough RAM
- Enough HDD/SSD
- A fast GPU

All of these things focus on only two basic needs,

1. Our computer should be fast enough to get our tasks done in almost no time.
2. It should be capable enough to load heavy softwares and games.

So both of the things above drill down to having a system which takes less time to process things and has enough space to load things.

But having a good experience with the softwares and programs is only dependent on hardwares ?

No, if we are building a piece of software, most of the time we make it compatible enough to work in systems which are not the best in the industry.

But when we write programs how much time is it going to take to execute and how much memory it will take isn't it dependent on hardwares ? It is, but we will learn a

theory using which whatever softwares and programs we will build, we can build them system agnostic to our best potential.

So, let's say two programmers build a program that can sort data for us. Now, we want to judge who wrote a better algorithm. But how to judge it ?

If we will run them on different machines then based on machine configuration, if a machine is extremely fast than the other then it will be a biased competition.

And let's say if you run them on the same machine, then also we won't be able to do the comparison. Why ?

Note: A situation in which you have a single core processor, then the computer system cannot do any type of multitasking. How computer does multitasking ??? What we see on the screen of our PC, is kind of an illusion. At one time instance, computer only runs one program and then it switches to the next at the very next time instance. So when the computer changes context the previous program under execution goes into a waiting state. The program keeps on waiting till the time it gets chance of execution again. It happens so fast that we feel, we are multi tasking. Because on an average in a regular personal computer, we can run approx `10^8 - 10^9` instructions per second.

So, judging an algorithms efficiency on machines by executing is a bad idea.

So how can we judge them ?

Can we use lines of code ? No, because for same lines of code, change in input can change the number of instructions. Example loops.

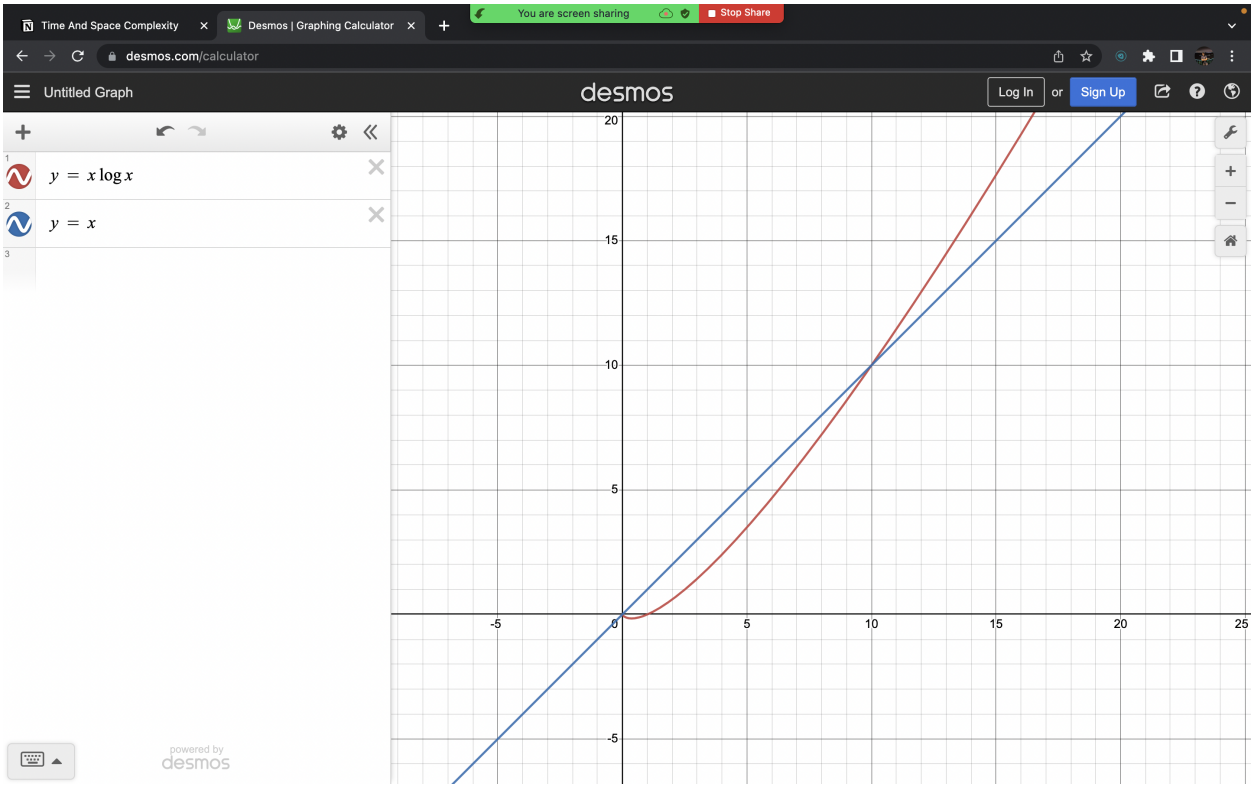
That means, we need to devise a strategy then can judge the efficiency of a program in terms of time and space, and also should be able to include the change in parameter for the judging criteria.

Asymptotic Analysis

The word Asymptotic comes from the word Asymptote.

> Asymptote means: **straight line that constantly approaches a given curve but does not meet at any infinite distance**

The definition is relevant because asymptotic analysis keeps a very simple assumption while judging algorithms. Just like how asymptote tries to evaluate and meet the curve at infinity (some very large value) similarly asymptotic analysis judges algorithms on large input values.



Consider the above two curves, the curve `y = xlogx` is performing better than `y = x` for small values of x. So let's consider, x as input and y as time/space takes, then if the curve `y = xlogx` belongs to `algorithm1` and `y = x` belongs to `algorithm2` then `algorithm1` looks like more efficient for small values. But, we don't care about small values, because time/space takes for small input is almost negligible for our computers. It is the big input values that actually puts our system on challenge and that is also what asymptotic analysis expects us to judge. And we can see that after a threshold for large values of x, `y = xlogx` is performing very bad when compared to `y = x`. So due to this `algorithm2` is better.

Now, how does all of it fits with our algorithms, like how can we read algorithm and judge its complexity.

Growth of a curve

What is growth of a curve ? It is the rate of change of the curve. The curve whose value changes more with a small change in input has more growth. So the algorithm which will show us more growth with respect to change in input, will be the slower one.

Situations to Judge Algorithms

Worst Case:

Worst case analysis mean that, in the worst possible situation how bad an algorithm can perform. It deals the baddest possible performance.

Best Case:

Best case analysis mean that, in the best situation, when everything is in favour, what can be the best possible performance of the algorithm.

Average Case:

On an average as a regular algorithm, what's the performance. We don't care about peek situations like worst or best but on a regular basis how things work ?

To denote different cases, we have different notations.

Asymptotic Notations:

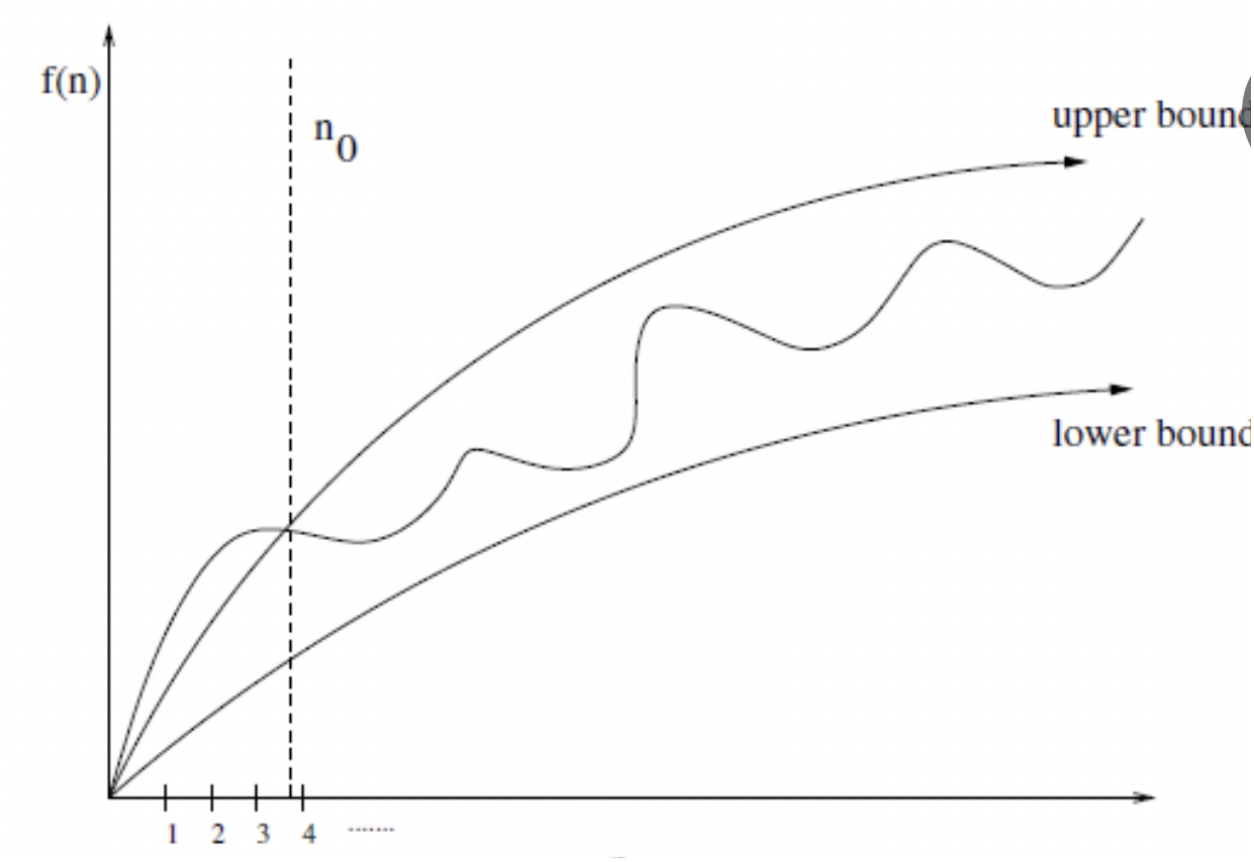
Worst Case: Big O → O(nlogn)

The big O notation or the worst case analysis talks about tightest upper bound of the growth of the algorithm.

Best Case: Big Omega → Ω(n)

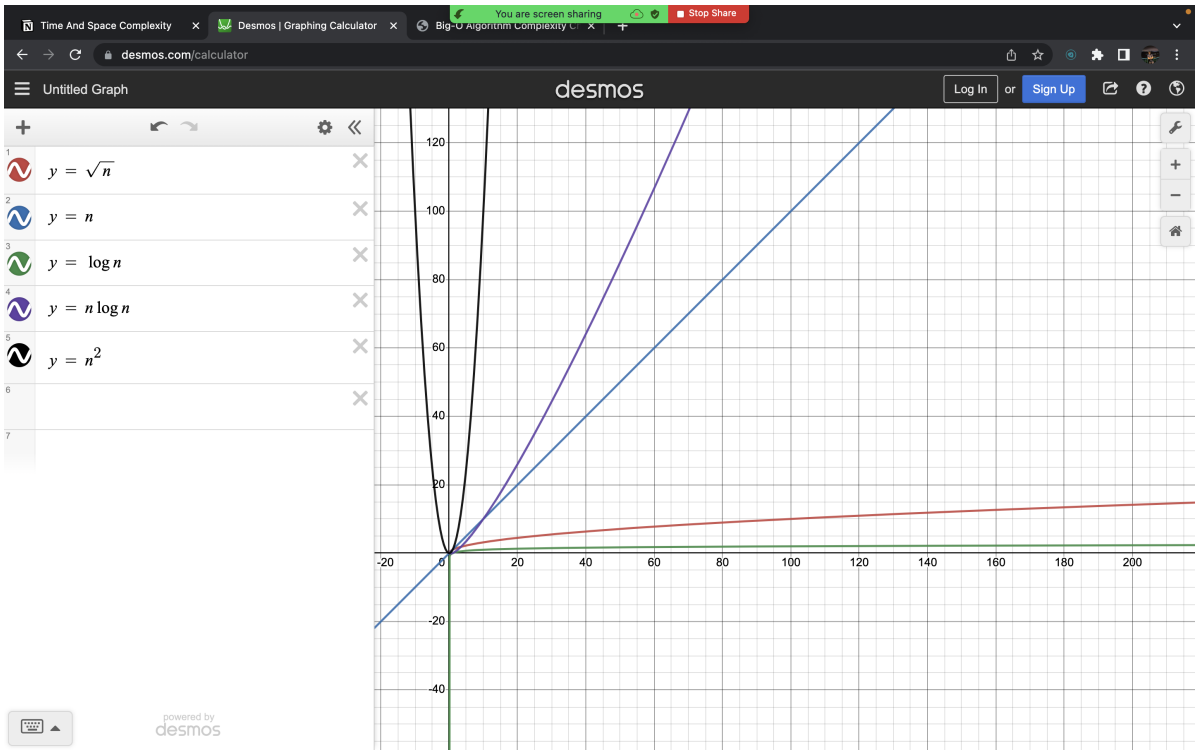
The big omega or the best case situation talks about the tightest lower bound of the growth of the algorithm.

Average Case: Big theta → θ(n^2)



Based on the above knowledge, how can we judge algorithms ?

- Any instruction, whose execution count changes based on change in input, is something that we care more about. Example: loops changes based on input, whereas conditionals or variable initialisation are constant.
- N! > 2^n > n^3 > n^2 > n**√n** > nlogn > n > **√n > logn > constant**



- For iterative functions, we try to analyse the number of times the loops executes.

```
function f1(n) {
    console.log("Inside function f1"); // constant
    if(n > 2) console.log("hey"); // constant
    for(let i = 0; i < n; i++) {  // 1. 3 , 2. 3, 3. 3, 4. 3 ..........
        console.log(i);
    }
}
```

- In the growth curve for asymptotic notations, we avoid any constant of lower degree terms. Example: y = 8x^3 + 3 x^2 + 10 → Write the Big O notation of the curve → O(x^3)

We avoid them because for large values of x, the lower degree terms and constants, doesn't add much.

Let's try to judge some algorithms !!!!!

```
function f1(n) {
    console.log("Inside function f1"); // constant
```

```
    for(let i = 0; i < n; i++) {  // 1. 3 , 2. 3, 3. 3, 4. 3 ..........
        console.log(i);
    }
}
// y = c + c + 3n -> O(n)
```

```
function f2(n) {
    console.log("Inside the function f2");
    for(let i = 0; i < Math.log(n); i++) { // total logn iterations
        console.log(i);
    }
}
// y = c + c + 3logn -> O(logn)
```

```
function f3(n, m) {
    for(let i = 0; i < n; i++) { // Total iterations = n
        console.log(i);
    }
    for(let j = 0; j < m; j++) { // Total iterations = m
        console.log(j);
    }
}
// so here when the first loop finishes then only second starts, so we add them
// y = 3n + 3m -> O(n + m)
```

```
function f4(n, m) {
    for(let i = 0; i < n; i++) { // total iterations for this outer for loop is n
        for(let j = 0; j < m; j++) { // total iterations for this inner for loop is m
            console.log(i, j);
        }
    }
}
// m + m + m + ..... (n times) -> O(nm)
```

```
function swap(arr, a, b) {
    let temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
// here complexity is contant -> y = 3 -> constant -> O(1)
```

```
function f5(n) {
    for(let i = 0; i < n; i++) {
        let j = i+1;
        while(j < n) {
            console.log(j);
            j++;
        }
    }
}
// i = 0, j = 1, iterations = n - 1
// i = 1, j = 2, iterations = n - 2
// i = 2, j = 3, iterations = n - 3
// ..
// ..
// i = n-1, j = n, iterations = 0
// Time = sum of iterations
// = (n-1) + (n-2) + (n-3) .... (2) + (1)
// Sum of first n-1 natural numbers. = ((n)(n-1))/2
// O(n^2)
```

```
function f6(n) {
    while(n > 0) {
        console.log(n);
        n = n / 2;
    }
}
// Iteration number = 0, value = n -> (n/(2^0))
// Iteration number = 1, value = n/2 -> (n/(2^1))
// Iteration number = 2, value = n/4 -> (n/(2^2))
// Iteration number = 3, value = n/8 -> (n/(2^3))
// ..
// ..
// Iteration number = K, value = n/(2^k)
// Time: No of iterations * constant
// total iteration = k+1
When the loop will stop in the kth iteration value of n is <= 1
and value is represented by n/2^k
(n / (2^k) ) <= 1
n <= 2^k
k >= log2n
Time: O(logn)
```

```
function f7(n) {
    let i = 1;
    let pow_of_2 = 1;
    while(i < n) {
        i=pow_of_2;
```

```
        pow_of_2*=2;
    }
}
Time: Total_iterations * constant
Iteration number = 0, value = 1
Iteration number = 1, value = 2
Iteration number = 2, value = 4
Iteration number = 3, value = 8
Iteration number = 4, value = 16
Iteration number = 5, value = 32
..
..
Iteration number = k, value = 2^k
Total_iterations = k
2^k > n => k > logn

Time: O(logn)
```

```
function shiftToLast(arr) {
    let i = 0;
    let k = 0;
    let j = arr.length - 1; // length of array is n, j = n-1
    while(k <= j) {
        if(arr[k] < 0) {
            // if element is negative we swap it with the element at j, and negative secti
on increases
            swap(arr, k, j);
            j--;
        } else {
            // if element is positive we just move ahead with i, and positive section incr
eases
            i++;
            k++;
        }
    }
}
Time: O(n)
```

```
function f11(n) {
    for(let i = 0; i*i < n; i++) {
        console.log("Value of i is");
        console.log(i);
        console.log("****");
    }
}

i*i = i^2
i* < n -> i^2 < n -> i < root(n)
// O(root(n))
```

```
function f12(n) {
    let ans = 0;
    for(let i = 1; i < n; i++) { // total iterations -> n-1
        for(let j = n; j > 1; j--) { // total iterations -> n-1
            ans += i;
        }
    }
    return ans;
}
// O(n^2)
```

```
function f13(n) {
    for(let i = n; i > 0; i = i/2) {
        for(j = 0; j < i; j++) {
            console.log(i, j);
        }
    }
}
/*
i = n, j -> [0, n) ->   n operations
i = n/2, j -> [0, n/2)->n/2 operations
i = n/4, j -> [0, n/4)->n/4 operations
....
i = 0, j -> [0, 0) (No operation on j)
Total -> n+n/2+n/4+n/8 .....
*/

// O(n)
```

How to calculate recursive complexity ?

```
function f(n) {
  if(n <= 1) return n; // constant
  return f(n-1) + f(n-2); // constant
}
```

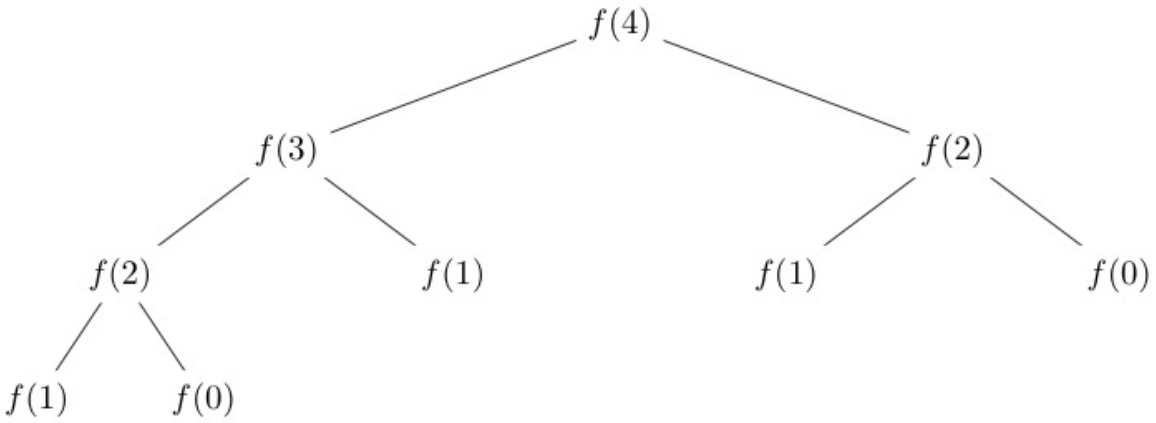
In a recursive code, a function gets called again and again by itself.

To calculate the complexity of the function, we have to calculate two values

- How many function calls are there ?

- In one function call how many operations we are doing

Time: no_of_function_call * no_of_ops_per_call

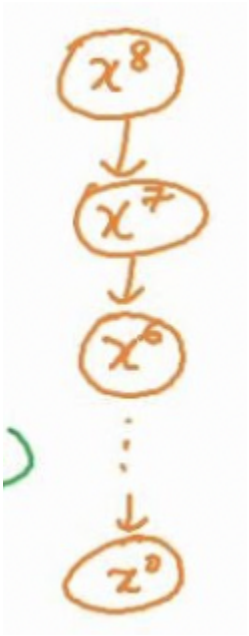



```
level 0 -> 1 function call
level 1 -> 2 function calls
level 2 -> 4 function calls
level 3 -> 8 function calls
..
..
..
level n -> 2^n function calls approx
Total -> 1 + 2 + 4 + 8 + .... + 2^n -> 2^(n+1) - 1
```

`Time: O(2^n)`

```
function power(a, b) {
    if(b == 0) return 1; // base case
    let assume = power(a, b-1); // assumption
    return a*assume;
}
```

No of ops in one call → constant



total function call → b function calls

`Time: O(b)`