**What is an algorithm?**

Origin: The English word "ALGORITHM" derives from the Latin word AL-

AL-KHWARIZMI’S name. He developed the concept of an

algorithm in Mathematics, and thus sometimes being called the

“Grandfather of Computer Science".

The concept of algorithm is nothing new to us. To accomplish something if you take sets of steps and achieve the desired goal, And if these steps can be replicated by anyone to do that exact thing then essentially you have developed an algorithm.

For example, Lets say we need to find the prime number, the algorithm can be:

**count <- 0**

**iterator <- 1**

**while iterator <=Number {**

**check number mod iterator == 0?{**

**count ++}**

**iterator ++**

**}**

**check count == 2?{**

**Prime Number}**

**else {**

**Not prime}**

Keep in mind, algorithm represents set of steps. That should not necessarily be a direct code. It should be a logical breakdown of the problem through a set of finite steps of instructions that can be implemented by anyone with whatever programming language they prefer. In short, A finite set of statements that guarantees an optimal solution in finite interval of time is an algorithm. Algorithmic thinking and algorithm analysing are vital in making efficient solutions.

**Whats important in programming?**

Performance of a program is always the part which peaks the interest of the normal user. However, other aspects of programming such as Correctness, Maintainability, Scalability, Stability, Robustness, Modularity are important as well.

Also, restrictions of resources or time can also be the contributing factor for the algorithm design because if an algorithm is performing better but taking an amount of space which is not feasible for the system, then this algorithm should not be considered.

**Time Analysis**

You need to identify the run time of the algorithm to correctly analyze it. But its easier said than done. Suppose, you have designed an algorithm which is talking approximately “n” amount of operations to complete the task (n being the size of the input), You cannot just claim that the algorithm is solving the problem in linear time for all the other inputs. It might be performing very poorly for different cases where inputs are more varied. So you need to know the timing for the best-case, worst-case and the average-case situation. Worst-case scenarios give us the upper bound of the solution that guarantees that no matter what, this algorithm will take atmost this upperbound amount of time to run. Similarly, Lower-bound guarantees that this algorithm will take at least this lower-bound amount of time to run. If we define the time function of an algorithm As T(n) then ,

* **Worst Case: T(n)** = Max time taken on any input size of N
* **Best Case: T(n)** = Min time taken on any input size of N
* **Average Case: T(n)** = Average time taken for most of the inputs. Its the most important thing needed to analyse an algorithm but its very hard to determine, since one can’t easily identify the average time needed for thousands of different inputs. Usually statistical analysis is needed to solve this.

It’s not always a very straightforward approach to identify these time complexities since most of the algorithms are very complex. For example, How do we understand the recursive functions time complexity is a bit more complicated than solving iterative tasks. As a future computer scientist, you may need to develop and algorithm to try to solve a problem which is yet to be solved with unpredictable variety of inputs. So, having as estimation of the behavior of the algorithm is more important than getting the exact value. If we can differentiate between the algorithm’s runtime to somehow bound the algorithm in a time based constraint then we can predict the time complexity of that particular algorithm. This is where the **Asymptotic Analysis** helps to comprehend the time complexity of an algorithm.

**Asymptotic Analysis**

This is a mathematical concept derived from Asymptotic function. This is an approach where we ignore all the machine dependant constraints (this is also why we don’t make the algorithm pseudocodes not in any particular parogramming language) to only focus on the performance of the algorithm. Secondly, This this analysis works on the observation of the **growth** of the function instead of looking at the actual running time(this is the asymptotic behavior of the algorithm). So, growth(T(n)) as N -> ∞ (N approaches to Infinity)

Asymptotic notations are given below:

1. **O(g(n)):** Big-Oh of g of n, the Asymptotic **Upper Bound**. Here,  
   **T(n) = O(g(n))** where there are two constants **c** and **n0** such that

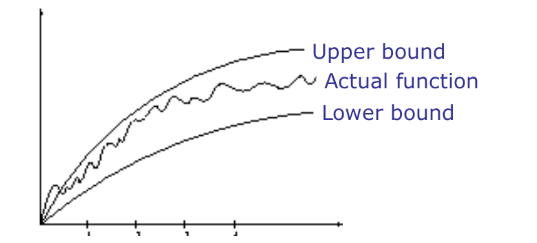
**0 <= T(n)<= c\*g(n)** for all **n > n0** (n0 being a positive input where anything larger than that should satisfy this notation)

1. **Ω(g(n)):** Omega of g of n, the Asymptotic **Lower Bound**. Here,  
   **T(n) = Ω(g(n))** where there are two constants **c** and **n0** such that

**0 <= T(n)<= c\*g(n)** for all **n > n0** (n0 being a positive input where anything larger than that should satisfy this notation)

1. **Θ(g(n))**: Theta of g of n, the Asymptotic Tight Bound. It’s essentially bounding the function from both upper and lower bound. Also known as tight bond. So, **Θ(g(n))⊂** **O(g(n)) ∩ Ω(g(n))**

It can be visualized like this:



Asymptotic notation we are going to focus on the most is for the **worst case scenarios**. The best case often relies on the inputs, average case is very difficult to compute as discussed. So, we try to guarantee an upperbound, meaning this function/algorithm will run at most on this amount of time. And with varied inputs, the algorithm more or less runs near the “Big-Oh” time complexity.

**Why use Big-Oh notation?**

More or less time, exact calculation of worst case running time is complicated and unnecessary. Since we are designing algorithm with large inputs in mind, to put into perspective:

T1(n) = 2

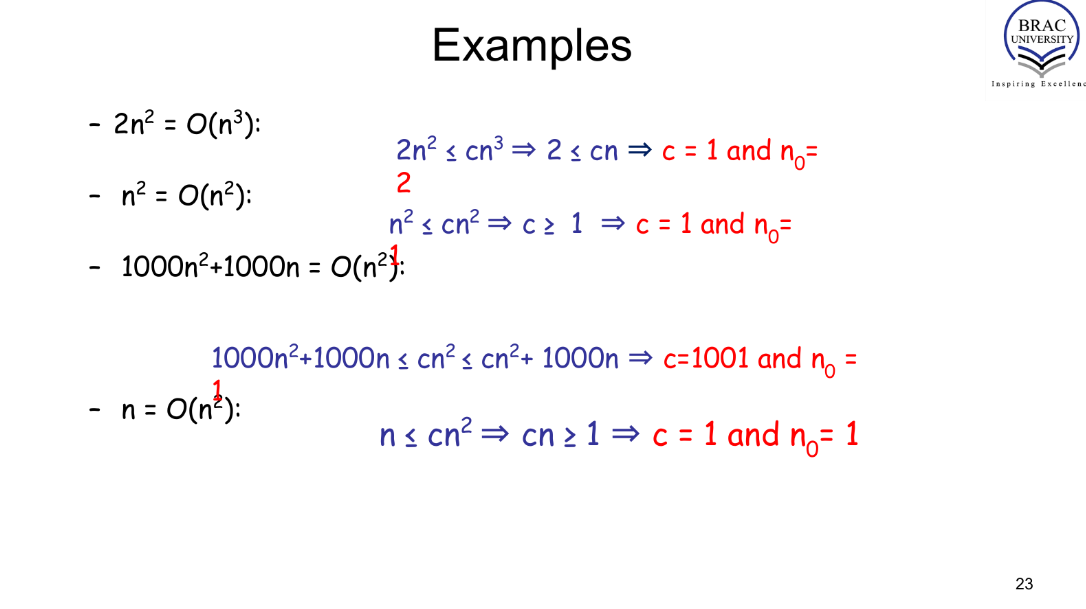
T2(n) =

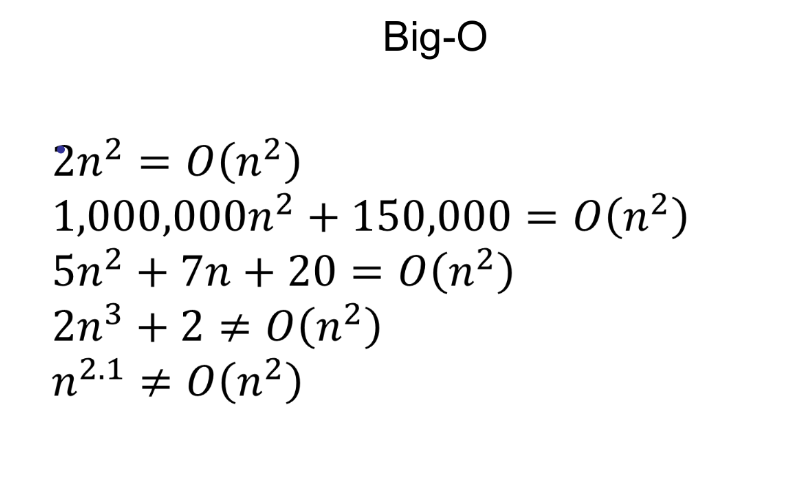
May not seem that different if you think of the value of n = 2. T1 would be 8 and T2 would also be 8. But if we consider the value of the n 10000; the value of T1 will be very small compared to the T2. To sum it up, the dominating term will always outweigh the other terms in the case of inputs being bigger.

Similarly, T(n) = ; Even for this time function you will see that with the value of n increasing the functions curve will always be determined the value of . Thats why: O(T(n)) =

**Please keep in mind,** The asymptotic notation does not always mean that the value should be exactly of that particular Big-oh notation. It means that the time complexity of the worst running case is approximately around that function. If n=100; if the algorithm is running on this run time: **T(n) = n(n+1)/2**; It still means that the time complexity of big-oh notation is O(). Also, remember, the upper bound is just bounding the time function in a way that this function will never produce a bigger value than the bound. So if a function can be defined as the big-oh notation of O() can also be denoted as O() also. So it’s more like T(n) is a subset of O(). But that is not the perfect representation of the algorithm, so Finding the tightest bound to represent the time complexity of that algorithm is crucial.

Some examples:





\*collected from BUX slides\*

**Sample codes and their time complexity:**

1.

function multiplier (int a, int b){

C = a \* b

return c}

Time complexity = O(1) [constant time]

2.if a >20{

print “yes”}

else {

print(“no”)}

Time complexity = O(1) [constant time]

3.

max = input()

for i in range(n-1):

temp = input()

if temp>max:

max = temp

print(max)

Time complexity = O(n) [Linear time]

4.

for i in range(n):

for j in range(n):

c+=1

print(c)

Time complexity = O() [quadratic time]

5.

for i in range(n):

for j in range(m):

c+=1

print(c)

Time complexity = O() [quadratic time] {this n doesnt represent the exact “n” from the code, it’s an expression that it will run linearly}

6.

for i in range(n):

c+=1

for j in range(n):

c+=1

print(c)

Time complexity = O(n) [Linear time]

7.

for i in range(n):

for j in range(i):

c+=1

print(c)

Time complexity = O() [quadratic time]

8. for i in range(n):

for j in range(n):

for k in range(n):

c+=1

Time complexity = O()

9.

i =1

c=10

while i<n:

c\*=10

i+=20

print(c)

Time complexity = O(n) [Linear time]

10.

i =1

c=10

while i<n:

c\*=10

i+=n

print(c)

Time complexity = O(1) [constant time]

11.

x = 0

i = 1

while x<n:

x = x + i

i += 1

//alternate code

x = 0

for (i = 1; x<n; i++){

x= x+i

}

[expl: this code will not run until n time because before i reaches n, the x value will reach the loop condition and break the loop. Assume the loop will run for “k” times. So the value of the k will be the ans

So, x will be at some point(Kth point/ the end of the loop)>= n;

Meaning: 1+2+3+....k >= n;

k(k+1)/2 >= n;

;

k>= √(n)]

Time complexity: O(√(n))

12.

i=1

while i < n:

i=i\*2

[expl: at one point, assume Kth point, i will be >=n; where the time will be we need to find the value of k.

So, when i = k, the total steps ->

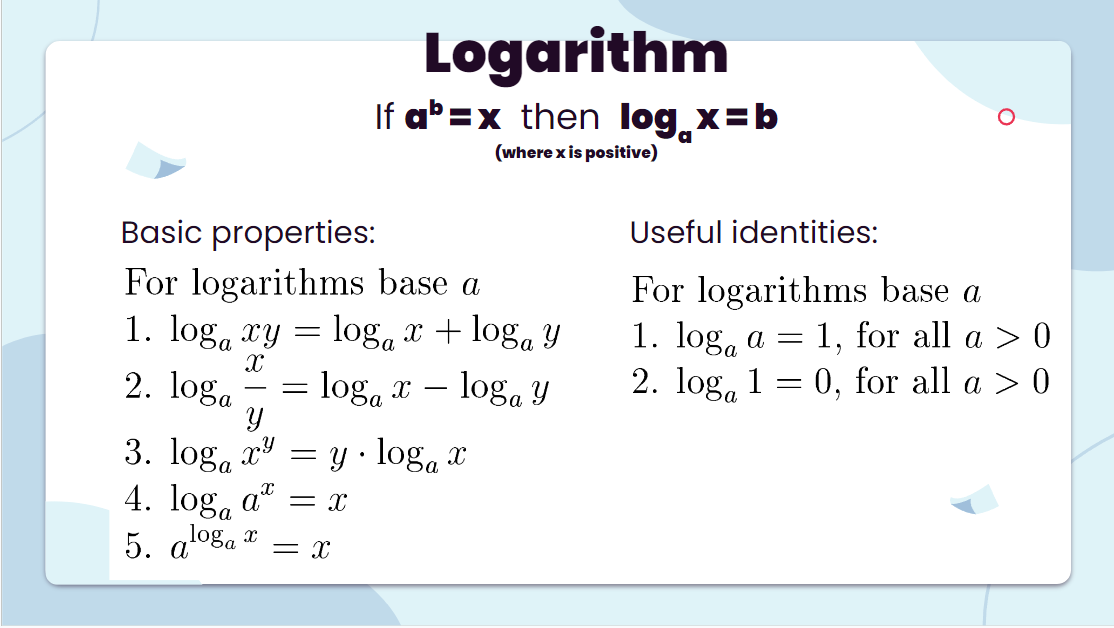
>= n

= n

= k]

Time complexity: O() also known as O(log n)

Log operations in case you forgot !!



12.

a,b,c=1,1,0

while a<n:

c+=1

a=a\*2

while b<c:

b=b\*2

Time complexity: O((n)) [Try to figure out how it works. Check the variable a,b,c carefully]

13.

a,b,c=1,1,0

while a<n:

c+=1

a=a\*2

**while b<c:**

**b=b\*2**

If you were asked to find out the t.c. for this highlighted area in terms of N judging by the whole code, what will it be??

14.

i=n

while i>=1:

i=i/2

Time complexity: O(logn)

15.

j=k=c=0

i=n/2

while (i<=n){

while (j+n/2 <= n){

k=1

while(k<=n){

count+=1

k=k\*2}

j+=1}

i+=1}

Time complexity: O() [calculate individual loop from outer to inner since it all depends on each other. Then multiply them since its nested] collected from : [explanation](https://www.youtube.com/watch?v=HFjJgSguqUA)

**Some widely used time complexities:**

1. 1(constant running time): Instructions are executed once or a few times

2. logN (logarithmic): A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step

3. N (linear): A small amount of processing is done on each input element

4. N logN: A problem is solved by dividing it into smaller problems, solving them independently and

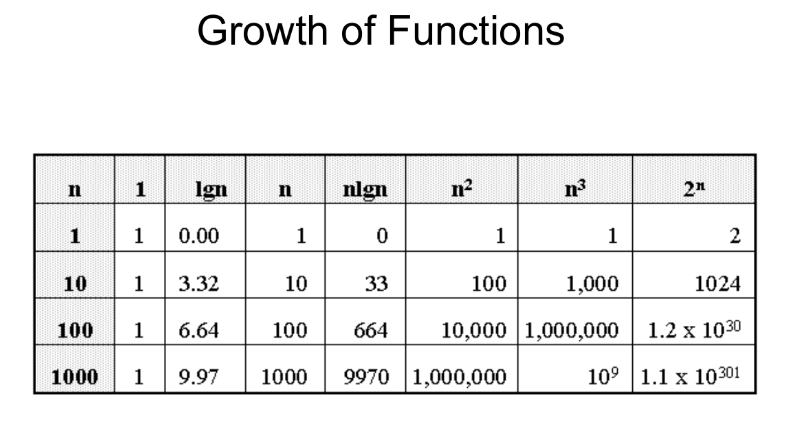
combining the solution

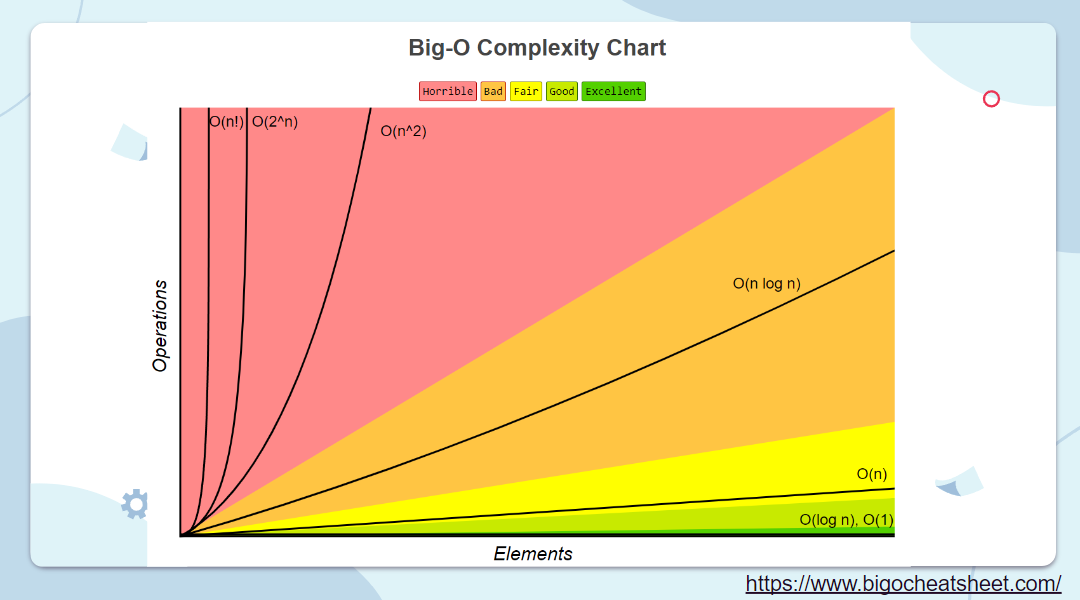
5. (quadratic): Typical for algorithms that process all pairs of data items (double nested loops)

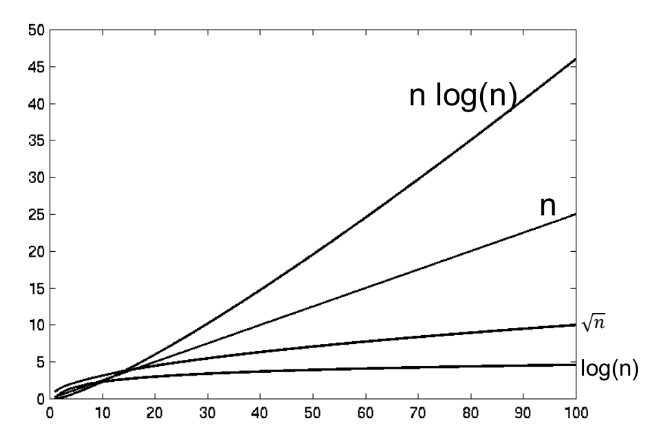
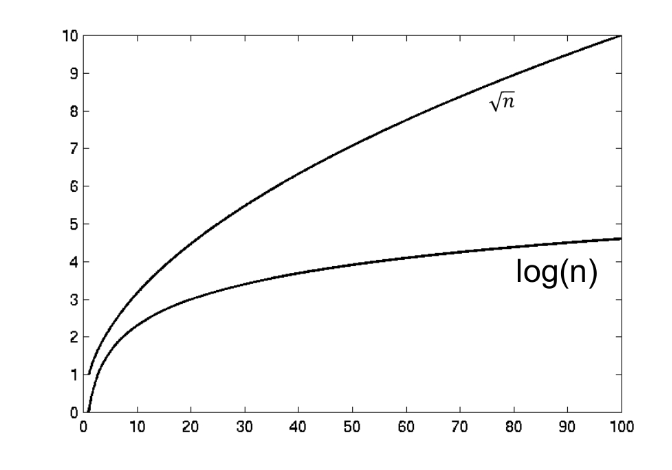
6. (cubic): Processing of triples of data (triple nested loops)

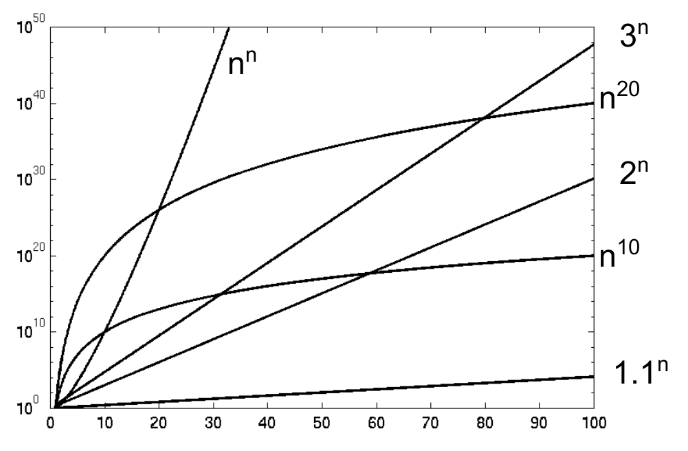
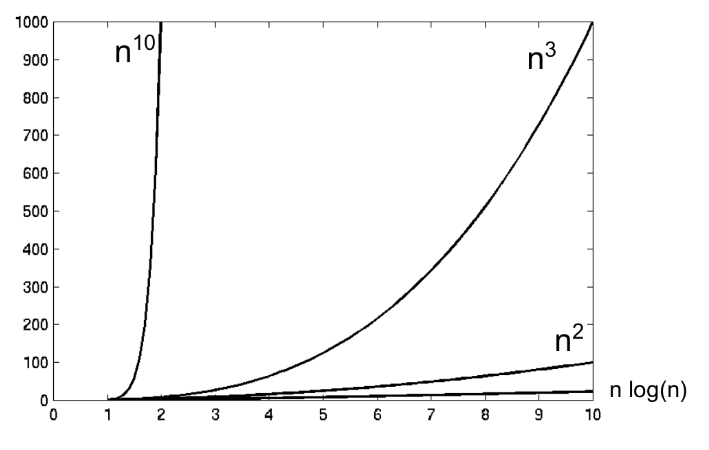
7. (polynomial)

8. (exponential)









SOME TIME COMPLEXITY RELATED GRAPHS FROM BUX & others

