Introduction to Algorithm Design

(Asymptotic Notations)

Algorithm Analysis

Suppose you are having one problem and you wrote three algorithms for the same problem. Now, you need to choose one out of those three algorithms. How will you do that?

- One thing that you can do is just run all the three algorithms on three different computers, provide same input and find the time taken by all the three algorithms and choose the one that is taking the least amount of time. Is it ok? No, all the systems might be using some different processors. So, the processing speed might vary. So, we can't use this approach to find the most efficient algorithm.
- Another thing that you can do is run all the three algorithms on the same computer and try to find the time taken by the algorithm and choose the best. But here also, you might get wrong results because, at the time of execution of a program, there are other things that are executing along with your program, so you might get the wrong time.

One thing that is to be noted here is that we are finding the time taken by different algorithms for the same input because if we change the input then the efficient algorithm might take more time as compared to the less efficient one because the input size is different for both algorithms. So, we have seen that we can't judge an algorithm by calculating the time taken during its execution in a particular system. We need some standard notation to analyze the algorithm. We use ***Asymptotic notation*** to analyze any algorithm and based on that we find the most efficient algorithm. Here in Asymptotic notation, we do not consider the system configuration, rather we consider the **order of growth** of the input. We try to find how the time or the space taken by the algorithm will increase/decrease after increasing/decreasing the input size.

Algorithm Analysis

- Performing step count calculation for large algorithms is a time consuming task.
- Time complexity depends upon the input size.
- if the input size is n, then f(n) is a function of n that denotes the time complexity.

Let's look at a simple example.

f(n) = 5n2+ 6n + 12   f(n)=5n2 =   when n=1

% of running time due to 5n2 = $\frac{5}{5+6+12} * 100 = 21.74\%$

% of running time due to 6n = $\frac{6}{5+6+12} * 100 = 52.17\%$

$\frac{12}{5+6+12} * 100 = 26.09\%$

% of running time due to 12 =

n	5n2	6n	12
1	5	6	12
10	500	60	12
100	50000	600	12
1000	5000000	6000	12

n	5n2	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

Algorithm Analysis

- Asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

What is Asymptotic Behaviour

- The word **Asymptotic** means approaching a value or curve arbitrarily closely
- **Expression 1**: f(n)=(20n2 + 3n - 4)   n-> ∞   f(n)=20 n2
- **Expression 2**: (n3 + 100n - 2)
- how the function will grow as the value of n(input) will grow, and that will entirely depend on n2 for the Expression 1, and on n3 for Expression 2.
- when n is very large. The function "*f (n)* is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$

Algorithm Analysis

Why is Asymptotic Notation Important?

- 1. They give simple characteristics of an algorithm's efficiency.
- 2. They allow the comparisons of the performances of various algorithms.

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity. Or

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. In asymptotic notations, we derive the complexity concerning the size of the input, These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithm

Algorithm Analysis

Before learning about these three asymptotic notation, we should learn about the best, average, and the worst case of an algorithm.

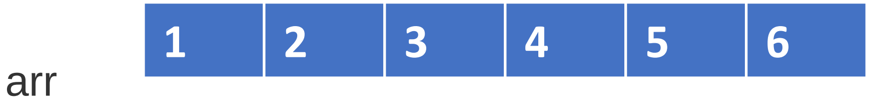
For Example:

We have one array named "arr" and an integer "k". we need to find if that integer "k" is present in the array "arr" or not? If the integer is there, then return 1 other return 0. Try to make an algorithm for this question.

- **Input:** Here our input is an integer array of size "n" and we have one integer "k" that we need to search for in that array.
- **Output:** If the element "k" is found in the array, then we have return 1, otherwise we have to return 0.

```
lsearch(arr[ ],n,k)
{
  for i=1  to n do
   {
         if  k=arr[i] then
               return true
    }
return false
}
```

one possible solution for the above problem can be linear search i.e. we will traverse each and every element of the array and compare that element with "k". If it is equal to "k" then return 1, otherwise, keep on comparing for more elements in the array and if you reach at the end of the array and you did not find any element, then return 0.



elementary operation :k=arr[i]

Case 1: search element is k=1 So here, after only one comparison, you will get that your element is present in the array.

Case 2: k=6 or k=7 : we try to find if element "6" is present in the array or not. Here, the if-condition of our loop will be executed 5 times and k=7 it will 6 times then the algorithm will give "0" as output.

case:3:Execution times for all possible inputs of size *n we are calculating* probabilities of these inputs.

Algorithm Analysis

Case:2

Let T1(n),T2(n)….. be the execution times for all possible inputs size n then W(n) is worst case time complexity W(n)=max(T1(n),T2(n)…..). we analyze the performance of an algorithm for the input, for which the algorithm takes long time or space.

Case1:

Let T1(n),T2(n)….. be the execution times for all possible inputs size n then B(n) is Best case time complexity B(n)=min(T1(n),T2(n)…..) . we analyze the performance of an algorithm for the input, for which the algorithm takes less time or space.

Case 3:

We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs . Average case

A(n)= sum(T1(n),T2(n)….)/n.

we analyze the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

Algorithm Analysis

Type of Asymptotic notation:

We use five types of asymptotic notations to represent the growth of any algorithm, as input increases:

1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)
4. Small oh(o)
5. Small Omega(ω)

Algorithm Analysis

Big-oh notation:

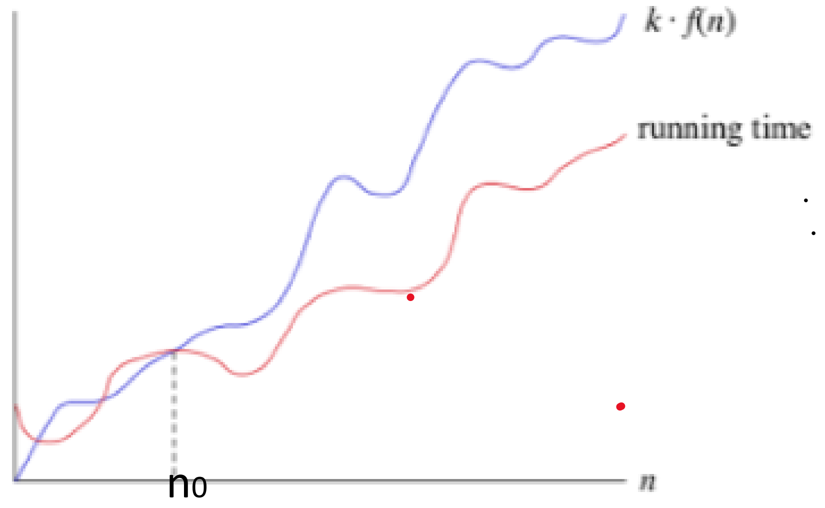
The big Oh is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

T(n) = O(g(n)) After n = n0 T(n) will never catch c1.g(n) where c1 is a constant .By writing f(n) = O(g(n)) we mean that a function f(n) is a member of set O(g(n))

If T(n)<=C1.g(n) then T(n)=O(g(n)) for all n>no

n	T(n)=2n+3	g(n)=n c1=3 (3n)
1	5	3
2	7	6
3	9	9
4	11	12
…..		
10	23	30

T(n)=O(n)
As 2n+3 <=3n for all n>=3



ASYMPTOTIC UPPER BOUND

1.T(n)=3n+2 4n g(n)=n c1=4 T(n)=O(n)
2.T(n)= 3n+3 4n T(n)=O(n)

Algorithm Analysis

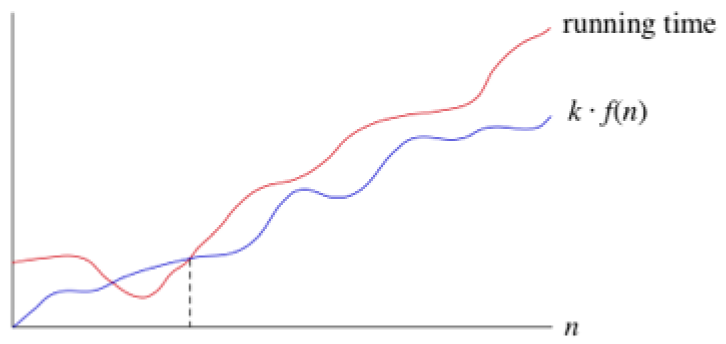
- Big Omega Notation, Ω:

The function T (n) = Ω (g (n)) [read as "T of n is omega of g of n"] if and only if there exists positive constant c2 and n_0 such that T(n) ≥ c2.g(n), ∀n ≥ n0. Omega can be used to denote all lower bounds of an algorithm. Omega notation also denotes the best case analysis of an algorithm.

If T(n)>=c2.g(n) for all n>n0 then T(n)= Ω (g (n))

n	T(n)=2n+3	C2=1.5 g(n)=n (1.5n)
1	5	1.5
2	7	3
3	9	4.5
4	11	6
10	23	15

T(n)>=1.5 n
C2=1.5 g(n)=n
T(n)= Ω(n)



n0

ASYMPTOTIC LOWER BOUND

$\underset{\smile}{T}$(n) =$8n^2$+2n-3 T(n)=8n2 7n2 T(n)>= 7.n2
c2=7 g(n) =n2 T(n)= Ω(n2)

Algorithm Analysis

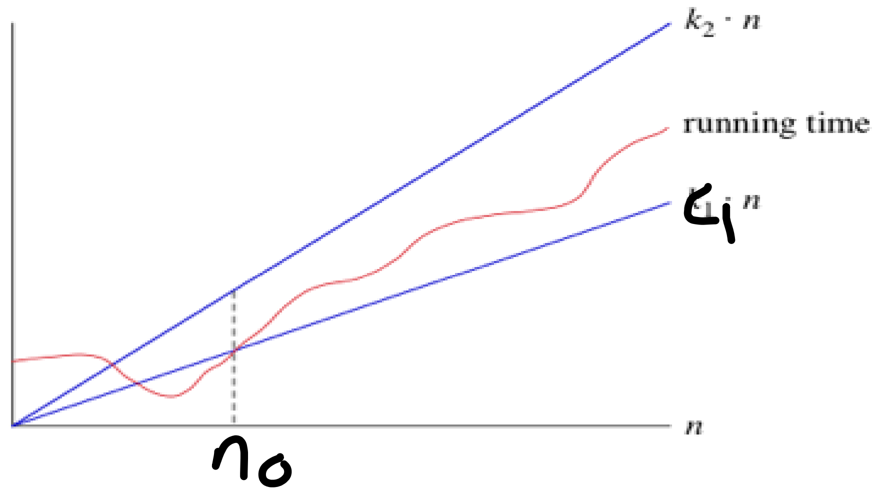
Big Theta (θ): The θ is the formal way to express both the lower bound and upper bound of an algorithm's running time.

The function T(n) = θ (g (n)) [read as " T is the theta of g of n"] if and only if there exists positive constant c_1, c_2 and n_0 such that

$$c_1 \; g(n) \le T(n) \le c_2 \; g(n) \text{ for all } n,\; n \ge n_0$$

$T(n)=3n+2= \theta(n)$ as $3n+2 \ge 2n$ and $3n+2 \le 4n$, for n $c_1=2, c2_2=4$, and $n_0=2$

T(n)=3n g(n)=n c1=2 c2=4 2n<=3n+2<=4n => T(n)= θ(n)



ASYMPTOTIC TIGHT BOUND

Algorithm Analysis

Small–o:

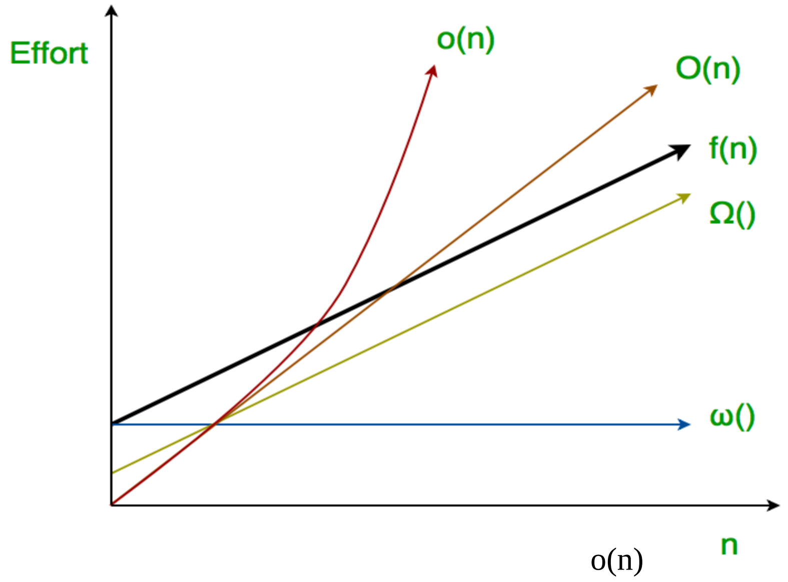
it can also be a loose upper-bound. “Little-o” (o()) notation is used to describe an upper-bound that cannot be tight. We say that T(n) is o(g(n)) (or T(n) = o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ T(n) < c1g(n).

Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.\

Example:

f(n)=3n^3+6n^2+60=

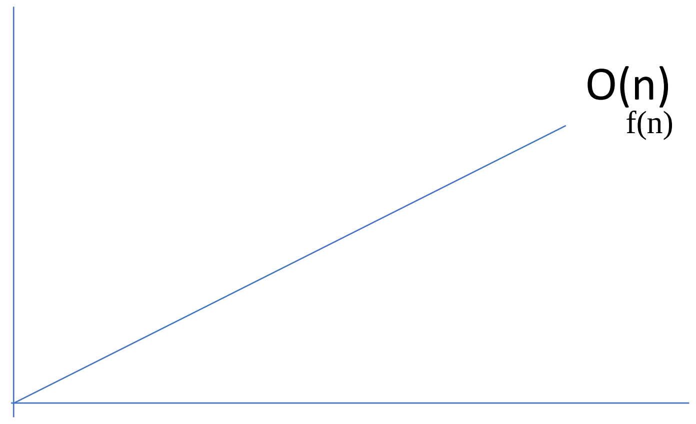
3n = o(n2),



Little –Omega (ω) :

- Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n0.
- f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions. In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=cg(n)<f(n).

.



Small–o:

Mathematical Relation of Little o notation

Using mathematical relation, we can say that $f(n) = o(g(n))$ means,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example on little o asymptotic notation

If $f(n) = n^2$ and $g(n) = n^3$ then check whether $f(n) = o(g(n))$ or not.

$$\lim_{n\to\infty} \frac{n^2}{n^3}$$

$$= \lim_{n\to\infty} \frac{1}{n}$$

$$= \frac{1}{\infty}$$

$$= 0$$

The result is 0, and it satisfies the equation mentioned above so $f(n)=o(g(n))$

In mathematical relation,
if $f(n) \in \omega(g(n))$ then,

$$\lim_{n\to\infty} f(n)/g(n) = \infty$$

Prove that $4n + 6 \in \omega(1)$;

The little omega(o) running time can be proven by applying limit formula given below.

if $\lim_{n\to\infty} f(n)/g(n) = \infty$ then functions $f(n)$ is $\omega(g(n))$

here, we have functions $f(n)=4n+6$ and $g(n)=1$

$$\lim_{n\to\infty} (4n+6)/(1) = \infty$$

and,also for any c we can get n0 for this inequality $0 <= c*g(n) < f(n)$, $0 <= c*1 < 4n+6$

Hence proved.

Properties of Asymptotic Notations

- Assuming f(n), g(n) and h(n) be asymptotic functions the mathematical definitions are:

1. If **$f(n) = \Theta(g(n))$**, then there exists positive constants c1, c2, n0 such that **$0 \leq c1.g(n) \leq f(n) \leq c2.g(n)$**, for all $n \geq n0$
2. If **$f(n) = O(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq f(n) \leq c.g(n)$**, for all $n \geq n0$
3. If **$f(n) = \Omega(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq c.g(n) \leq f(n)$**, for all $n \geq n0$
4. If **$f(n) = o(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq f(n) < c.g(n)$**, for all $n \geq n0$
5. If **$f(n) = \omega(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq c.g(n) < f(n)$**, for all $n \geq n0$

Properties:

1. Reflexivity:

If f(n) is given then **$f(n) = O(f(n))$**

- **Example:** **$f(n) = n^2$; $O(n^2)$ i.e $O(f(n))$**
- Similarly, this property satisfies both Θ and Ω notation. We can say
  If f(n) is given then f(n) is Θ(f(n)).
  If f(n) is given then f(n) is Ω (f(n)).

Properties of Asymptotic Notations

Transitive Properties :

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n , g(n) = n^2 and h(n)=n^3
n is O(n^2) and n^2 is O(n^3) then n is O(n^3)

Similarly this property satisfies for both Θ and Ω notation. We can say
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

Symmetric Properties :

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .
Example: f(n) = n^2 and g(n) = n^2 then f(n) = Θ(n^2) and g(n) = Θ(n^2)
This property only satisfies for Θ notation.

Transpose Symmetric Properties :

If f(n) is O(g(n)) then g(n) is Ω (f(n)).
Example: f(n) = n , g(n) = n^2 then n is O(n^2) and n^2 is Ω (n)
This property only satisfies for O and Ω notations.

Properties of Asymptotic Notations

Some More Properties :

1. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$
2. If $f(n) = O(g(n))$ and $d(n)=O(e(n))$
then $f(n) + d(n) = O(\max(g(n), e(n)))$

Example: $f(n) = n$ i.e $O(n)$

$d(n) = n^2$ i.e $O(n^2)$
then $f(n) + d(n) = n + n^2$ i.e $O(n^2)$

3.If $f(n)=O(g(n))$ and $d(n)=O(e(n))$
then $f(n) * d(n) = O(g(n) * e(n))$

Example: $f(n) = n$ i.e $O(n)$

$d(n) = n^2$ i.e $O(n^2)$
then $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$

Properties of Asymptotic Notations

1. Proof : $f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$(symmetry)

Necessary part: $f(n) = \theta(g(n)) \Rightarrow g(n) = \theta(f(n))$

By the definition of θ , there exists positive constants c1, c2, no such that $c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq no$

$$\Rightarrow g(n) \leq 1 /c1 .f(n) \text{ and } g(n) \geq 1/ c2 .f(n)$$

$$\Rightarrow 1/c2\ f(n) \leq g(n) \leq 1 /c1\ f(n)$$

Since c1 and c2 are positive constants, 1 /c1 and 1 /c2 are well defined. Therefore, by the definition of θ , $g(n) = \theta(f(n))$

Sufficiency part: $g(n) = \theta(f(n)) \Rightarrow f(n) = \theta(g(n))$

By the definition of θ , there exists positive constants c1, c2, no such that $c1.f(n) \leq g(n) \leq c2.f(n)$ for all $n \geq no$

$\Rightarrow f(n) \leq 1 /c1 .g(n)$ and $f(n) \geq 1/ c2 .g(n) \Rightarrow 1/ c2 .g(n) \leq f(n) \leq 1/ c1 .g(n)$

By the definition of θ , $f(n) = \theta(g(n))$ This completes the proof of Symmetry property

Properties of Asymptotic Notations

Transitivity

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

Proof:

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

By the definition of Big-Oh(O) , there exists positive constants c, no such that $f(n) \leq c1.g(n)$ for all $n \geq no$

$\Rightarrow f(n) \leq c1.g(n)$

$\Rightarrow g(n) \leq c2.h(n)$

$\Rightarrow f(n) \leq c1.c2h(n) \Rightarrow f(n) \leq c.h(n)$,

where, $c = c1.c2$ By the definition, $f(n) = O(h(n))$

Note : Theta(Θ) and Omega(Ω) also satisfies Transitivity Property

Properties of Asymptotic Notations

Transpose Symmetry

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Proof:

Necessity: $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$

By the definition of Big-Oh (O)

$\Rightarrow f(n) \leq c.g(n)$ for some positive constant c

$\Rightarrow g(n) \geq 1/\ c\ f(n)$

By the definition of Omega (Ω) , $g(n) = \Omega(f(n))$

Sufficiency: $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$

By the definition of Omega (Ω), for some positive constant c

$\Rightarrow g(n) \geq c.f(n) \Rightarrow f(n) \leq 1/\ c\ g(n)$

By the definition of Big-Oh(O) , $f(n) = O(g(n))$ Therefore, Transpose Symmetry is proved.

Asymptotic Notations

Claim: if $f1(n)=O(g1(n))$ and $f2(n)=O(g2(n))$ then $f1(n)+f2(n)=O(g1(n))+O(g2(n))$

Proof:

Suppose for all $n>=n1, f1(n) <= c1.g1(n)$ and for all $n>=n2, f2(n)<=c2.g2(n)$

Let $n0=\max\{n1,n2\}$ and $c0=\max\{c1,c2\}$

Then for all $n>=n0$

$$f1(n)+f2(n)<= c1.g1(n)+c2.g2(n)$$

$$<=c0(g1(n)+g2(n))$$

$$\Rightarrow \quad f1(n)+ f2(n)=O(g1(n)+g2(n))$$

Some more Observations on Asymptotic Notation

Remarks :

1. If lim $n \to \infty$ $f(n)/g(n) = 0, \infty$ $f(n) = \theta(g(n))$

2. If lim $n \to \infty$ $f(n)/g(n) \neq \infty$ then $f(n) = O(g(n))$

3. If lim $n \to \infty$ $f(n)/g(n) = 0$, then $f(n) = \Omega(g(n))$

4. If lim$n \to \infty$ $f(n)/g(n) = 0$ then $f(n) = o(g(n))$

5. If lim$n \to \infty$ $f(n)/g(n) = \infty$, then $f(n) =$ omega$(g(n))$

6. L 'Hospital Rule : If f(n) and g(n) are both differentiable with derivates $f`(n)$ and $g`(n)$, respectively, and if lim$n \to \infty$ $f(n) =$ lim$n \to \infty$ $g(n) = \infty$

lim$n \to \infty$ $f(n)/g(n) =$ lim$n \to \infty$ $f`(n)/g`(n)$

Asymptotic Notations

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, ordered by slowest to fastest growing:

1. $\Theta(1)$
2. $\Theta(\log_2 n)$
3. $\Theta(n)$
4. $\Theta(n\log_2 n)$
5. $\Theta(n2)$
6. $\Theta(n2\log_2 n)$
7. $\Theta(n3)$
8. $\Theta(2n)$
9. $\Theta(n!)$

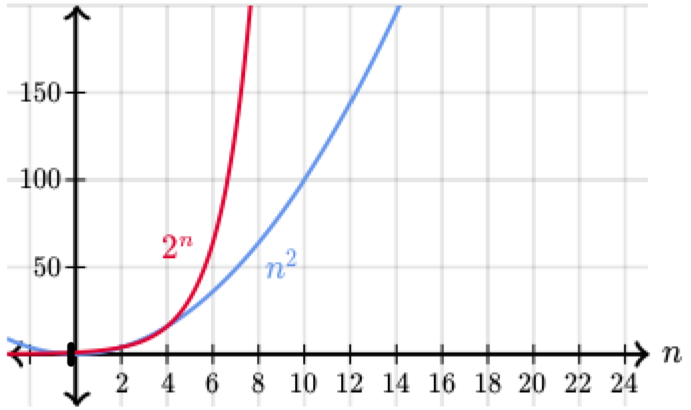
Question:

- For the functions, n^k and c^n, what is the asymptotic relationship between these functions?

Assume that k >= 1 and c > 1 are constants.

To answer this, we need to think about the function, how it grows, and what functions bind its growth.

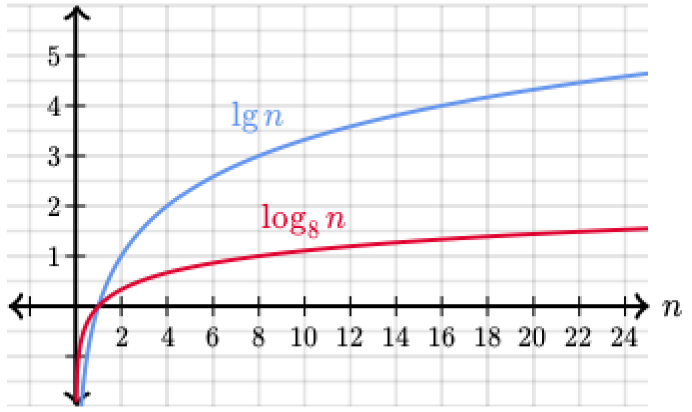
- *n^k, is a polynomial function, and c^n is a exponential function. We know that polynomials always grow more slowly than exponentials. We could prove that with a graph, but we have to make sure we look at it for big values of n, because the early behavior could be misleading. Here's a graph that compares the two functions (with k=2 and c=2), where we can clearly see the difference in growth:*



n^k is O(c^n)

For the functions, log2n, and log8n, what is the asymptotic relationship between these functions?

- To answer this, we need to think about the function, how it grows, and what functions bind its growth.
- Both log2*n* and log8*n*, are functions with logarithmic growth, with their base as the only difference. Here's a graph of the two functions:



For the functions, 8^n and 4^n what is the asymptotic relationship between these functions?

What is the asymptotic relationship between the functions n^3 log2n and 3n log8*n*3?

Are each of the following true or false?

- (a) 3 n^2 + 10 n log n = O(n log n)
- (b) 3 n^2 + 10 n log n = Omega(n^2)
- (c) 3 n^2 + 10 n log n = Theta(n^2)
- (d) n log n + n/2 = O(n)
- (e) 10 SQRT(n) + log n = O(n)
- (f) SQRT(n) + log n = O(log n)
- (g) SQRT(n) + log n = Theta(log n)
- (h) SQRT(n) + log n = Theta(n)
- (i) 2 SQRT(n) + log n = Theta(SQRT(n))
- (j) SQRT(n) + log n = Omega(1)
- (k) SQRT(n) + log n = Omega(log n)
- (l) SQRT(n) + log n = Omega(n)

a) False, since n^2 (the dominate term on the left) is asymptotically faster growing than n log n and hence not upperbounded by it.

b) (b,c) True, since n^2 (the dominate term on the left) asymptotically grows like n^2 and hence it is Omega(n^2) and also Theta(n^2). faster growing than n log n and hence not upper bounded by it.

(d) False since n log n (the dominate term on the left) is not asymptotically upperbounded by n.

(e) True, since the dominate term on the left, 10 SQRT(n), is asymptotically upperbounded by n.

(f,g) False, since the dominate term on the left, SQRT(n), is not asymptotically upperbounded by n. See the class notes where we showed that that lim as n -> infinity of log n / SQRT(n) = 0 giving that SQRT(n) is asymptotically faster growing.

(h) False, since the dominate term on the left, SQRT(n), is asymptotically faster slower growing than n.

(i) True, since the dominate term on the left, 2 SQRT(n), grows asymptotically at the same rate as SQRT(n).

(j) True, since the dominate term on the left, SQRT(n), is asymptotically faster growing than 1.

(k) True, since the dominate term on the left, SQRT(n), is asymptotically faster growing than log n.

(l) False, since the dominate term on the left, SQRT(n), is asymptotically slower growing than n.

BEST BEST

$O(1)$	CONSTANT
$O(\log n)$	LOGARITHMIC
$O(n)$	LINEAR
$O(n^2)$	QUADRATIC
$O(n^3)$	CUBIC
$O(n^k)$	POLYNOMIAL
$O(2^n)$	EXPONENTIAL
$O(n!)$	FACTORIAL

WORST WORST