Growth of a Function

We know that for the growth of a function, the highest order term matters the most e.g., the term c_1n^2 in the function $c_1n^2 + c_2n + c_3$ and thus we can neglect the other terms and even the coefficient of the highest order term i.e., c_1 (assuming coefficients are neither too large nor too small).

Even with these approximations, we will be able to know about the rate of the growth of our function and this is enough information to keep in our mind while developing an algorithm.

As you know our main discussion is going to be about the rate of the growth of the function, there are some notations for the same which we are going to study further.

Notations for the Growth of a Function

There are some notations which we frequently use to represent the performance of an algorithm. Let's have a look at these notations.

O-Notation

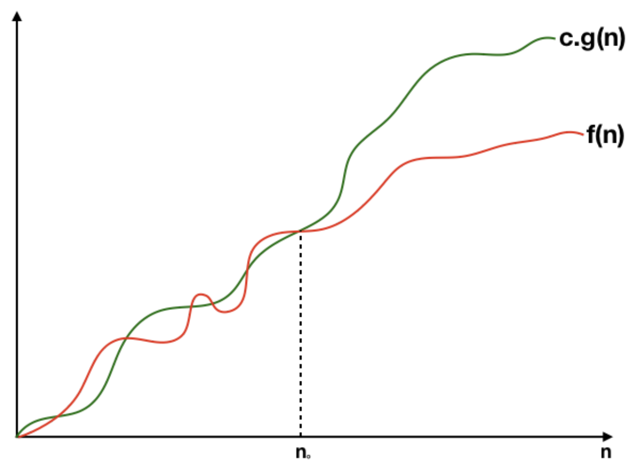
O-notation (pronounced as big-oh) is used for an asymptotic upper bound of a function i.e., it is used to represent that a function $f(n)$ can't exceed some another function $g(n)$ and thus $g(n)$ is the upper bound of $f(n)$.

As per the definition,

$$f(n) = O(g(n)),\ if\ f(n) \leq c.\,g(n)$$

for some positive constant 'c' and $n \geq n_o$.

So, what this equation basically means? It means that if we are writing $f(n) = O(g(n))$ then we are telling that $g(n)$ is the upper bound of $f(n)$ and there are some positive values of c and n_o for which the function $f(n)$ is always less than or equal to the function $c.\,g(n)$ for all $n \geq n_o$. Let's look at the graph below to make it more clear.



From the above figure, you can see that the function $f(n)$ is always less than or equal to the function $g(n)$ for any $n \geq n_o$. Also, the restriction is not applied to the region of $n \leq n_o$.

(/add_ques

</>

There can exist some value of c for which the condition is not satisfied at all but we are telling that there must be at least one value of c for which the function $c.g(n)$ is greater than or equal to the function $f(n)$.

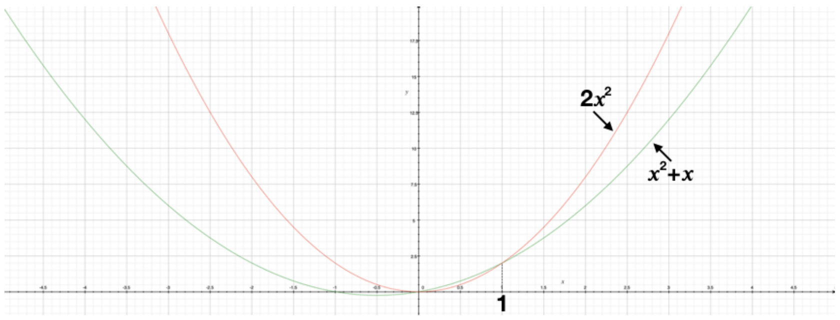
Let's have a look at some examples.

- $n^2 = O(n^2)$

  To show this, we have to prove that $n^2 \leq cn^2$ (according to the definition) for some positive values of c and n_o and all $n \geq n_o$.
  Choosing $c = 1$ and $n_o = 1$ easily satisfies the condition, so $n^2 = O(n^2)$.

- $n^2 + n = O(n^2)$

  We have to show that cn^2 is the upper bound of $n^2 + n$ for some positive constant c.
  Now, $n \leq n^2$ for all $n \geq 1$
  or, $n^2 + n \leq n^2 + n^2 = 2n^2$ for all $n \geq 1$
  Thus, $c = 2$ and $n_o = 1$ ($n^2 + n \leq 2n^2 => n \geq 1$) satisfies the condition.



- $2n - 1 = O(n^2)$

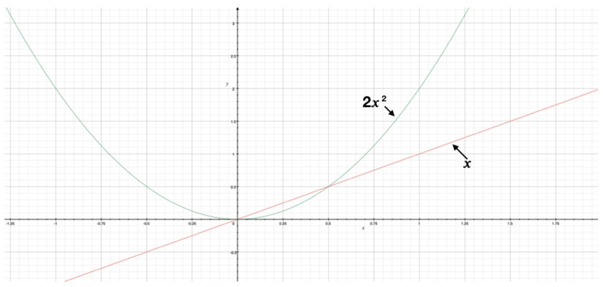
  Let's choose $c = 1$. Then for all $n \geq 1$, $2n - 1 \leq n^2$.

- $n^2 = O(n^3)$

  For the value of $c = 1$ and $n_o = 1$, $n^2 \leq n^3$ for all $n \geq 1$.

- $2n^2 \neq O(n)$:

  For any c, $cn < 2n^2$ when $n > c/2$, so we can't choose any value of n_o for which the condition will satisfy.



</>

Feel free to discuss your queries in the Discussion Section (https://www.codesdope.com/discussion/).

As now you have understood the meaning of the O-notation, let's discuss the significance of it and when it is used.

This notation is useful in representing the worst case of an algorithm i.e., its performance in the worst case. Suppose, the time complexity of an algorithm is $O(n^2)$, it means that the runtime of the algorithm is always less than or equal to cn^2 for some positive constant c and for all $n \geq n_o$. Basically, we are trying to

?

convey a message that the algorithm may take cn^2 time in the worst case but never more than that.

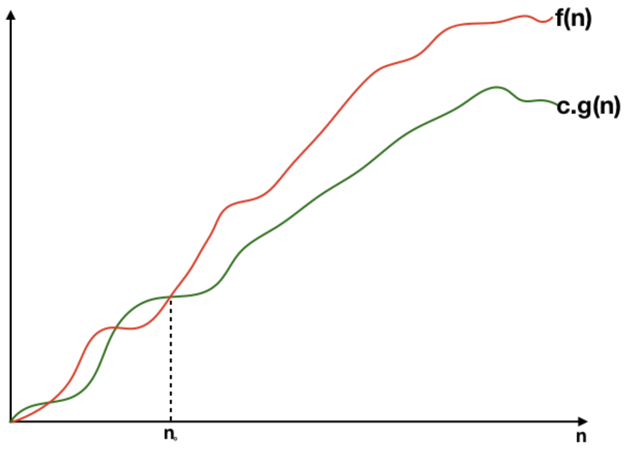
Ω-Notation

As the O was used to represent the upper bound, the Ω (pronounced as big-omega) is used to represent the lower bound. So according to the definition,

$$f(n) = \Omega(g(n)),\ if\ f(n) \geq c.g(n)$$

for some positive constant c and $n \geq n_o$.

Thus, if we write our function $f(n)$ is $\Omega(g(n))$, then we mean that our function $f(n)$ is always greater than or equal to the function $c.g(n)$ for some positive constants c, n and $n \geq n_o$. Let's take a look at the graphical representation of the same.

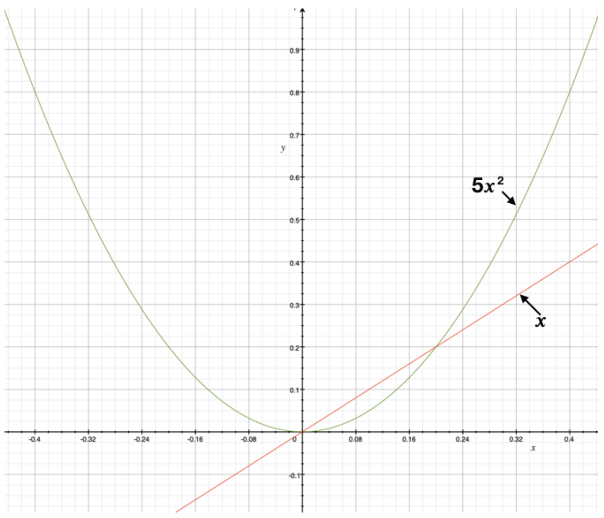


Similar to the O, we are not claiming anything about $n < n_o$ and all values of c but we are saying that there exist at least one positive value of c and n_o each for which our condition is satisfied.

Let's look at some examples of Ω.

- $5n^2 = \Omega(n)$

  We have to show that $5n^2 \geq cn$ for some positive constant c and n_o for all $n \geq n_o$.
  We can choose values $c = 1$ and $n_o = 1$, the condition always satisfies.



- $10n \neq \Omega(n^2)$

  Let's suppose $10n = \Omega(n^2)$ and thus $cn^2 \leq 10n => n(cn - 12) \leq 0$
  Since n must be positive $=> cn - 12 \leq 0 => n \leq 12/c$.
  We need a value of n_o such that, the condition is satisfied for all $n \geq n_o$ but we have shown that n is less than a constant ($n \leq 12/c$) and thus no such value of n_o can exist.
  Thus, our supposition was wrong and $10n \neq \Omega(n^2)$.

As you must have already guessed, Ω is used to represent the best case of an algorithm. So, if I am writing the time complexity of my algorithm is $\Omega(n)$, then I am conveying that my algorithm is going to perform in $c.n$ time in the best case but never less than that.

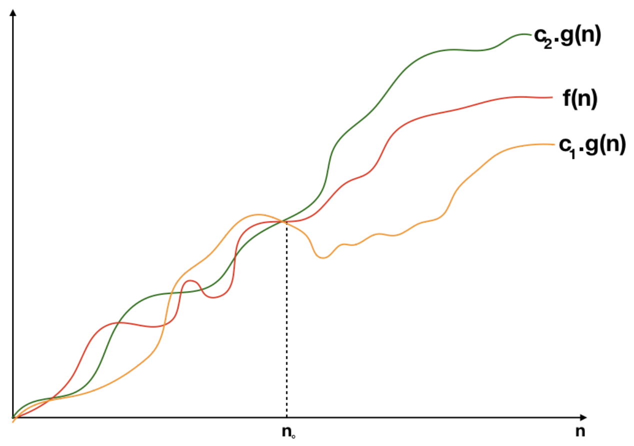
Θ-Notation

The Θ (pronounced as big-theta) is used to represent the tight bound of a function. So, according to the definition,

$$f(n) = \Theta(g(n)),\ if\ c_1.g(n) \le f(n) \le c_2.g(n)$$

for some positive constant c and $n \ge n_o$.

In this case, the function $f(n)$ is sandwiched between the functions $c_1 g(n)$ and $c_2 g(n)$. Let's have a look at the graphical representation of the same.



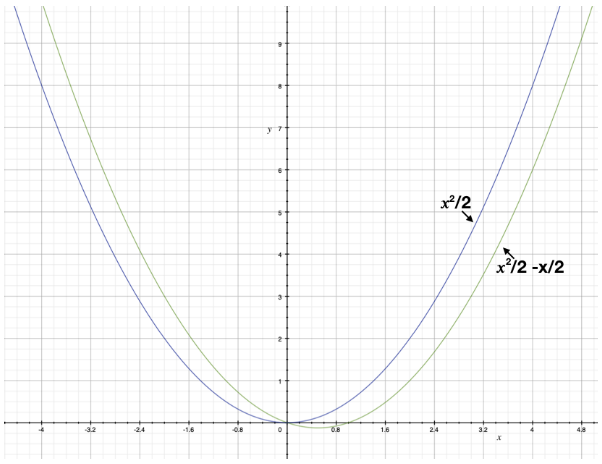
So, on saying our function is going to perform in $\Theta(\log_2(n))$, we mean that the time taken by our algorithm will be in between $c_1 \log_2(n)$ and $c_2 \log_2(n)$ but never less or more than that for any $n \ge n_o$. Thus, we are basically stating that our algorithm is bounded by the function $\log_2(n)$.

Let's look at some examples.

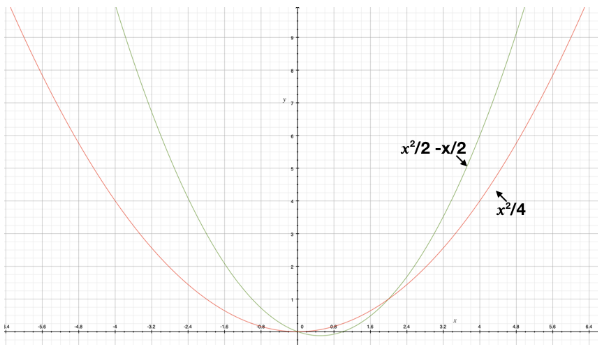
- $\frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$

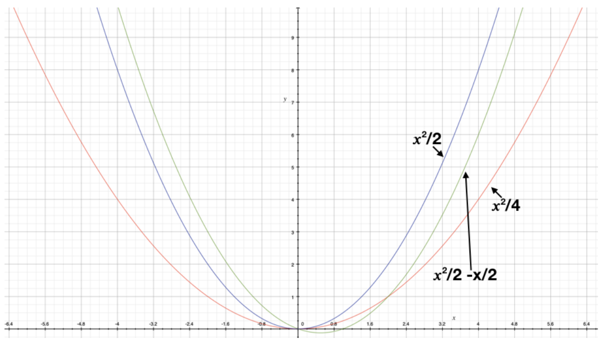
  We have to show that $c_1 n^2 \le \frac{n^2}{2} - \frac{n}{2} \le c_2 n^2$ and thus basically find the values of c_1 and c_2.
  Since n is positive, then $\frac{n^2}{2}$ must be greater than $\frac{n^2}{2} - \frac{n}{2}$ (We are subtracting a positive value $\left(\frac{n}{2}\right)$ from $\frac{n^2}{2}$)
  $=> \frac{n^2}{2} - \frac{n}{2} \le \frac{n^2}{2}\ (\forall n \ge 0)$
  $=> c_2 = \frac{1}{2}$

  </>
  $\forall n \ge 2$ means for all $n \ge 2$.



Also, $\frac{n^2}{2} - \frac{n}{2} \geq \frac{n^2}{2} - \frac{n}{2} * \frac{n}{2}$ $(\forall n \geq 2)$ (We are increasing the value of whatever is getting subtracted ($n/2$ to $n^2/2$) from $n^2/2$. So, ultimately decreasing the value of $\frac{n^2}{2} - \frac{n}{2}$)
$=> c_1 = \frac{1}{4}$





- $6n^3 \neq \Theta(n^2)$

  $c_1 \leq 6n^3 \leq c_2 n^2$
  Taking $6n^3 \leq c_2 n^2$,
  $=> 6n \leq c_2$ $(n > 0)$
  $=> n \leq \frac{c_2}{6}$

  So, n_o can't exist because this condition only exists when n is less than a constant $\frac{c_2}{6}$.

</>
You can always choose a different way to proceed with these kinds of problems. The value of constants (c_1, c_2 and n_o) will differ according to your choice.

</>
While writing an algorithm, we mainly focus on the worst case because it gives us an upper bound and a guarantee that our algorithm will never perform worse than that. Also, for most of the practical cases, average case is close to the worst case.

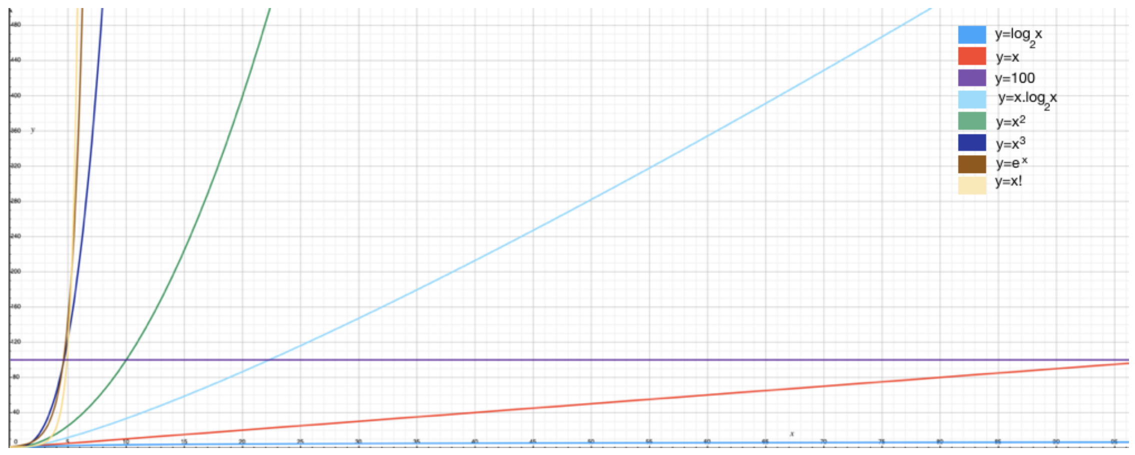
These three notations are the most used ones while dealing with algorithms. We have also used different $f(n)$ functions the in above examples like n, n^2, $\log_2(n)$, etc and there are few other functions which we commonly use while dealing with algorithms. So, it is a good idea to have these functions in our mind and their comparison which is also called **dominance relations** i.e., which algorithm dominates another.

</>
A faster growing algorithm dominates a slower growing one.

Commonly Used Functions and Their Comparison

1. **Constant Functions** - $f(n) = 1$ - Whatever is the input size n, these functions take a constant amount of time.
2. **Linear Functions** - $f(n) = n$ - These functions grow linearly with the input size n.
3. **Quadratic Functions** - $f(n) = n^2$ - These functions grow faster than the superlinear functions i.e., $n\log(n)$.
4. **Cubic Functions** - $f(n) = n^3$ - Faster growing than quadratic but slower than exponential.
5. **Logarithmic Functions** - $f(n) = \log(n)$ - These are slower growing than even linear functions.
6. **Superlinear Functions** - $f(n) = n\log(n)$ - Faster growing than linear but slower than quadratic.
7. **Exponential Functions** - $f(n) = c^n$ - Faster than all of the functions mentioned here except the factorial functions.
8. **Factorial Functions** - $f(n) = n!$ - Fastest growing than all these functions mentioned here.

The growth rate can easily be seen from the graph given below.



From the graph, you can see that for any sufficiently larger n,

$$n! \geq 2^n \geq n^3 \geq n^2 \geq n\log(n) \geq n \geq \log(n) \geq 1$$

Since we always want to keep the rate of the growth as low as possible, we try to make an algorithm to follow the function with least growth rate to accomplish a task.

Let's analyze the two cost functions we derived in the previous chapter.

$$C_1 = n^2\left(\frac{c_2+c_3}{2}\right) + n\left(c_1 + \frac{3c_2+c_3}{2}\right) + (c_1+c_2)$$

$$C_2 = n(c_2+c_3) + (c_1+c_2)$$

With the knowledge of this chapter, we can easily say that the first equation (C_1) is $O(n^2)$ and the second equation(C_2) is $O(n)$ or $\Theta(n^2)$ and $\Theta(n)$ respectively, to be more precise. So, now you have seen how we analyze any algorithm from start. First, we derived the cost function of the algorithm we have written and then from this cost function, we obtained the required information.

Now, you know how to analyze an algorithm and the notations used in it. In next chapters, we are going to do the analysis of more algorithms.

(/add_ques

❝ Pure mathematics is, in its way, the poetry of logical ideas. ❞

- Albert Einstein

PREV (/course/algorithms-analyze-your-algorithm/) (/course/algorithms-analyze-iterative-algorithms/) NEXT

New Questions

setting up an ide for mac.. - **Cpp**

(/discussion/setting-up-an-ide-for-mac)

Please fill the blanks and help me am stuck - **Java**

(/discussion/fill-the-blanks-and-help-me-am-stuck)

Time complexity of the Python Function - **Python**

(/discussion/time-complexity-of-the-python-function)

How I Can find The Best Target Coupons & Latest Deals Offers? - **Other**

(/discussion/how-i-can-find-the-best-target-coupons-latest-deal)

To Calculate salaries - **Java**

(/discussion/to-calculate-salaries)

How to write a c++ program that will declares a two dimensional array say Char My element [4][5], prompt the user to initialize and display the elemen - **C**

(/discussion/how-to-write-a-c-program-that-will-declares-a-two-)

What is the difference between a local variable and an instance variable? - **Java**

(/discussion/what-is-the-difference-between-a-local-variable-an)

Ask Yours

(/add_question/)