Algorithms

Lecture : Asymptotic Analysis

Imdad ullah Khan

Contents

1 Algorithms Runtime Analysis

Recall the following from last session.

1.1 Runtime as a function of input size

We want to measure runtime (number of elementary operations) as a function of size of input. Note that this does not depend on machine , operating system or language. **Size of input** is usually number of bits needed to encode the input instance, can be length of an array, number of nodes in a graph etc. It is important to decide which operations are counted as elementary, so keep this in mind while computing complexity of any algorithm. There is an underlying assumption that all elementary operation takes a constant amount of time.

1.2 Best/worst/average Case

For a fixed input size there could be different runtime depending on the actual instance. As we saw in the parity test of an integer. We are generally interested in the worst case behavior of an algorithm

Let $T(I)$ be time, algorithm takes on an instance I. The best case running time is defined to be the minimum value of $T(I)$ over all instances of the same size n.

Best case runtime:

$$t_{best}(n) = \text{MIN}_{I:|I|=n}\{T(I)\}$$

Worst case runtime:

$$t_{worst}(n) = \text{MAX}_{I:|I|=n}\{T(I)\}$$

Average case:

$$t_{av}(n) = \text{AVERAGE}_{I:|I|=n}\{T(I)\}$$

1.3 Growth of runtime as size of input

Apart from (generally) considering only the worst case runtime function of an algorithm. We more importantly, are interesting in

1. Runtime of an algorithm on large input sizes
2. Knowing how the growth of runtime with increasing input sizes. For example we want to know how the runtime changes when input size is doubled?

2 Asymptotic Analysis of functions

Asymptotic analysis consider growth of functions on large inputs. The asymptotic behavior of a function $f(n)$ refers to the growth of $f(n)$ as n gets large. We use to measure or compare performances of algorithms when applied on inputs of "very" large size. As noted above we need to compare two functions (e.g. compare the efficiency of two algorithms), asymptotic analysis of functions enables us to compare functions. For the following definitions we assume that all functions are real valued functions on the domain $\mathbb{R}$.

2.1 Asymptotic Upper Bounds - Big O Notation

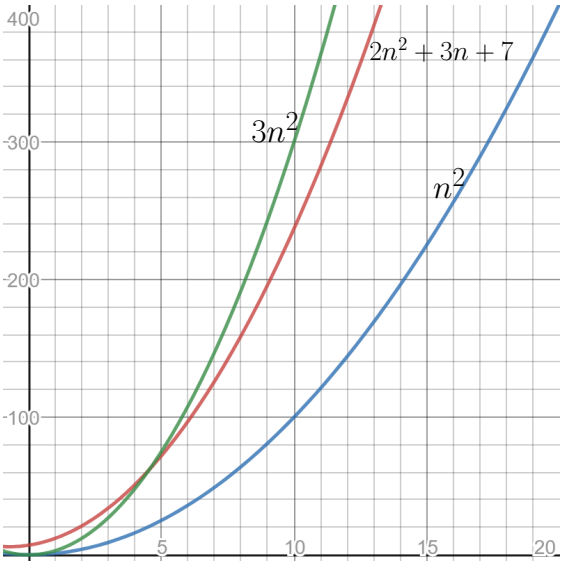
Definition 1 (O (Big Oh)). *A function $g(n) \in O(f(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that*

$$g(n) \leq cf(n) \qquad \forall\, n \geq n_0$$

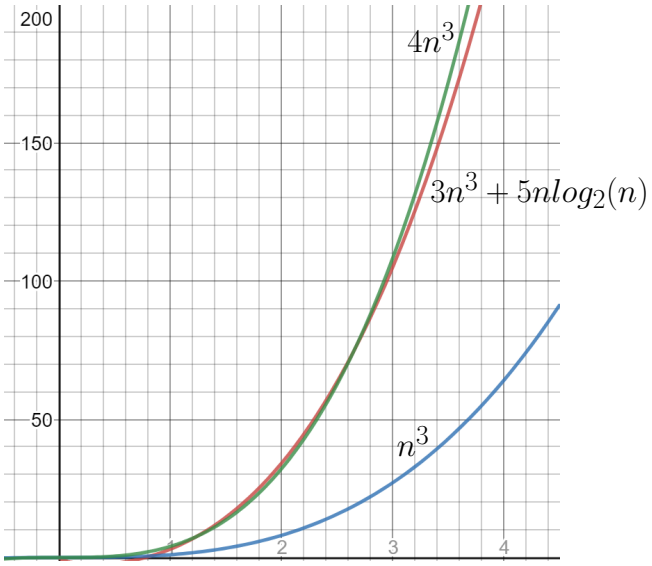
- This is at attempt to extend the definition of $a \leq b$ for real numbers to functions.
- Since the behavior of functions change in different ranges of the domain, this says that $g(n)$ is '*kind of less than*' $f(n)$ for all values of the domain bigger than the threshold n_0.
- '*kind of less than*' means that it is less than some constant times the $f(n)$ (the constant c).
- Various ways to refer to this are $g(n)$ is asymptotically dominated by $f(n)$, (where the word asymptotic encapsulates the constants c and n_0.
- $f(n)$ is an asymptotic upper bound on $g(n)$.
- We abuse the notation and write $g(n) = O(f(n))$.

2.1.1 Examples

Let $g(n) = 2n^2 + 3n + 7$ and $f(n) = n^2$. Then $g(n) = O(f(n))$, i.e. $2n^2 + 3n + 7 = O(n^2)$. This follows by the above definite considering $c = 3$ and $n_0 = 5$. See also the diagram to see what the definition means geometrically.



Let $g(n) = 3n^3 + 5n \log n$ and $f(n) = n^3$, then $g(n) = O(f(n))$. As $c = 4$ and $n_0 = 3$ we get that $3n^3 + 5n \leq cn^3$ for all $n \geq n_0$. This is depicted in the following diagram



2.1.2 Rules for functions simplification

Here are some commonsense rules that help simplify functions by omitting dominated terms and ignoring coefficients.

1. n^a dominates n^b if $a > b$: for instance, n^2 dominates n. This rule implies for example that $7n^4 + 3n^3 + 10 = O(n^4)$ and $3n^3 + 5n \log n = O(n^3)$

2. Multiplicative constants can be omitted: Again from in the two examples above we ignored lower order terms because of previous rule this rules says why can we ignore the constants such as 7 and 3.

3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).

4. Any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, that n^2 dominates $n \log n$. This is actually the previous rule.

2.1.3 Justification of these rules

The following example will make it clear why can we ignore lower order terms. Let $g(n) = pn^2 + qn + r$ and $f(n) = n^2$. By the above rule we can say that $g(n) = O(n^2)$. Indeed, we will show that this just follows from definition of Big-Oh.

$$\begin{aligned} f(n) &= pn^2 + qn + r \\ &\leq |p|n^2 + |q|n^2 + |r|n^2 \\ &= (|p| + |q| + |r|)n^2 \end{aligned}$$

This is true for all $n \geq 1$, hence with $c = (|p| + |q| + |r|)$ we get that $f(n) = O(n^2)$.

Another way to look at this (particularly in terms of running time of algorithms), is to consider our goals in analysis of algorithms. Recall that we said we are only interested in runtime of algorithms on inputs of very large sizes.

Let $T(n) = n^2 + 10n$ be the runtime of an algorithm $\mathcal{A}$. By the above rule we declare that $T(n) = O(n^2)$.

Consider an input size of 10^9 (which is not very big now a days) Then on one hand $T(n) = n^2 + 100n = 10^{18} + 10^{11}$. While $n^2 = 10^{18}$. The fractional error we get by ignoring $100n$ term is given by $\frac{10^{11}}{10^{18}} = 10^{-7}$. So for $n = 10^9$, $T(n) = n^2 + 10n$ is only 0.00001% more than n^2 (which is essentially our estimate).

There are a couple of reasons for ignoring the coefficients of even the dominating term

- First of all usually the coefficients in the running time aren't that big. (When the coefficients are *really* big, they are taken into account)
- Secondly getting an exact number for the coefficient is not that simple. Since the elementary operations (the *units*) for different algorithms can be different. And we don't know the relative times taken for these different operations.
- The coefficients really don't matter when we're considering how scalable an algorithm is. That is, an algorithm that takes ck time on an input of size k will take twice as much time on an input size of $2k$ regardless of the value of c. We elaborate on this point more, since we stated that this as our goal, we want to know how the function behaves with increasing input size.

Let runtime function be **Linear** e.g. $f(n) = 5n$. Then running time with increasing input sizes are given as:

input size	runtime $f(n) = 5n$
n	$5n$
$2n$	$2(5n)$
$3n$	$3(5n)$
$4n$	$4(5n)$

We can see that running time grow quadratically with no effect from the coefficient 7. To see another example, let runtime function be **Quadratic** e.g. $f(n) = 7n^2$. Then running time with increasing input sizes are given as:

input size	runtime $f(n) = 7n^2$
n	$7n^2$
$2n$	$4(7n^2)$
$3n$	$9(7n^2)$
$4n$	$16(7n^2)$

We can see that running time grow quadratically with no effect from the coefficient 7.

To see another example, let runtime function be **Cubic** e.g. $f(n) = 2n^3$. Then running time with increasing input sizes are given as:

input size	runtime $f(n) = 2n^3$
n	$2n^3$
$2n$	$8(2n^3)$
$3n$	$27(2n^3)$
$4n$	$64(2n^3)$

We can again see that there is no effect from the coefficient of n^3.

2.1.4 More Examples and finding the right constants

1. Let say $g(n) = 7n + 4$ and $f(n) = n$, we show that $g(n) \in O(f(n))$, i.e. $7n + 4 \in O(n)$.

   Well take $c = 8$ and $n_0 = 4$, we see that $7n + 4 \leq 8(n)$ whenever $n_0 \geq 4$, hence we proved that $7n + 4 \in O(n)$.

   With practice we would be able to guess the values of c and n_0 (we will also learn some rules of thumb), but one way to derive such these constants is as follows.

   We want that $7n + 4 \leq cn$, we solve this thing for c, we get that $c \geq \frac{7n}{n} + \frac{4}{n}$. When n is large $8 \geq 7 + \frac{4}{n}$, actually for any $n \geq 4$ this works. One can also see from the fact that $\lim_{n\to\infty} \frac{7n+4}{n} \to 7$, but this ($c = 7$) would require n_0 to be approaching ∞, so we take $c = 8$. Now how to get n_0? Well we want $7n + 4 \geq 8n$, this is true whenever $n \geq 4$.

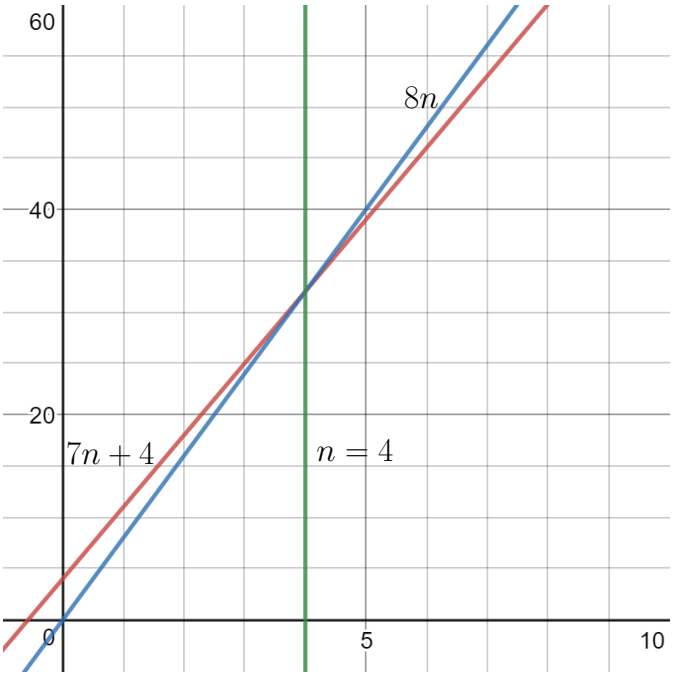


Figure 1: Example 1

2. Let $f(n) = 6n + 24$ and $h(n) = n^2$, we will show that $f(n) \in O(h(n))$

   Again since $\lim_{n\to\infty} \frac{6n+24}{n^2} \to 0$, so basically any $c \geq 0$ will work, lets choose $c = 1$. We want to $6n + 24 \leq 1 \cdot n^2$, which is true whenever $n \geq 10$. So we choose $c = 1$ and $n_0 = 10$, and we are done.

3. Let $f(n) = 32 \log n$ and $g(n) = \frac{\sqrt{n}}{10}$, we will show that $f(n) \in O(g(n))$.

   Lets choose $c = 10$, we want $32 \log n \leq 10 \cdot \frac{\sqrt{n}}{10} \implies 32 \log n \leq \sqrt{n}$. Now generally, I would have guessed from the plots of $\log n$ and $\sqrt{n}$, but to get some idea to get n_0, we can proceed as follows.

   We want $8 \log n \leq \sqrt{n} \implies \frac{\sqrt{n}}{\log n} \geq 8$

   Taking logarithms on both sides we get $\log\left(\frac{\sqrt{n}}{\log n}\right) \geq \log(8) \implies \frac{1}{2}\log n - \log\log n \geq 3$. Now since '*we know that* $\log\log n$ *is much smaller than* $\log n$' (you don't have to think recursively, just take a few examples). Instead of proving $\frac{1}{2}\log n - \log\log n \geq 3$ we prove something stronger, $\frac{1}{4}\log n \geq 3$, which is true whenever $n \geq 2^{12} = 4096$.

4. This one is a generic example. Let $g(n)$ be any polynomial of degree k and let $f(n) = n^k$, we will show that $g(n) \in O(f(n))$.

   Let $g(n) = a_k n^k + a_{k-1} n^k + \ldots + a_1 n + a_0$. Since

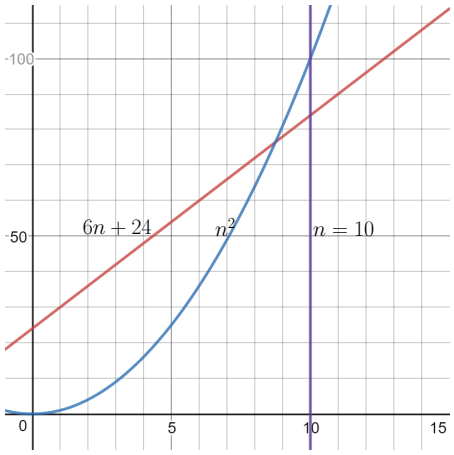


Figure 2: Example 2

$$\begin{aligned} a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 &\leq |a_k| n^k + |a_{k-1}| n^k + \ldots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + |a_{k-1}| n^k + \ldots + |a_1| n^k + |a_0| n^k \\ &\leq (|a_k| + |a_{k-1} + \ldots + |a_1| + |a_0|)\, n^k \end{aligned}$$

Take $c = (|a_k| + |a_{k-1} + \ldots + |a_1| + |a_0|)$ and $n_0 = 1$. Actually $c = \max\{|a_k|, |a_{k-1}, \ldots, |a_1|, |a_0|\}$ would work too but we are not required to find the best (optimal) constants, just any constants.

5. Let $f(n) = 3n^2 + 4n + 5$ and $g(n) = n$, we will show that $f(n) \notin O(g(n))$.

   Assume for contradiction that $3n^2 + 4n + 5 \in O(n)$, then there must be constants c and n_0 such that $3n^2 + 4n + 5 \leq cn \;\; \forall n \geq n_0$. Since n is positive this implies that $3n^2 \leq cn$ which gives us that $c \geq 3n$, but the claim was that c is a constant (independent of n). Here c depends on n, hence we get a contradiction.

3 Asymptotic-Complexity Classes

There are complexity classes of functions that usually occur in algorithm analysis. We have seen polynomials in the previous examples e.g., $3n^2 + 4n + 5$.

Table 1: The most common classes of asymptotic complexity

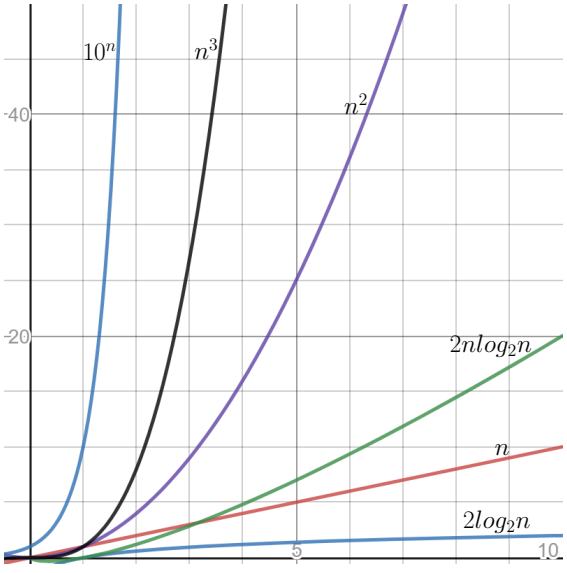
Class Name	**Class Symbol**	**Example**
Constant	$O(1)$	Comparison of two integers
Logarithmic	$O(log(n))$	Binary Search, Exponentiation
Linear	$O(n)$	Linear Search
Log-Linear	$On(log(n))$	Merge Sort
Quadratic	$O(n^2)$	Integer multiplications
Cubic	$O(n^3)$	Matrix multiplication
Polynomial	$O(n^a)$, $a \in \mathbb{R}$	
Exponential	$O(a^n)$, $a \in \mathbb{R}$	Print all subsets
Factorial	$O(n!)$	Print all permutations

The functions grow faster from right to left i.e.

$$n! \gg 2^n \gg n^3 \gg n^2 \gg nlogn \gg n \gg logn \gg 1$$

Here $a \gg b$ means that a is much larger than b.

3.1 Growth Rates of Functions



3.2 Big Oh: Why does it make sense?

Table 2: The running time of algorithms of different complexity levels for varying input size, on a computer executing one instruction per nanosecond (i.e. a computer with speed 1GHz). Assume that each runtime step takes 1ns.

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$	$3.63ms$
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.4\mu s$	$1ms$	77.1 $years$
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.9\mu s$	$1sec$	8.4×10^{15} $years$
40	$0.005\mu s$	$0.04\mu s$	$0.213\mu s$	$1.6\mu s$	$18.3min$	very long
50	$0.006\mu s$	$0.05\mu s$	$0.282\mu s$	$2.5\mu s$	13 $days$	very long
100	$0.007\mu s$	$0.10\mu s$	$0.644\mu s$	$10\mu s$	4^13 $years$	very long
1000	$0.010\mu s$	$1.00\mu s$	$9.966\mu s$	$1ms$	very long	very long
$10,000$	$0.013\mu s$	$10\mu s$	$130\mu s$	$100ms$	very long	very long
$100,000$	$0.017\mu s$	$0.10ms$	$1.67ms$	$10sec$	very long	very long
$1,000,000$	$0.020\mu s$	$1ms$	$19.93ms$	$16.7min$	very long	very long
$10,000,000$	$0.023\mu s$	$0.01sec$	$0.23sec$	1.16 $days$	very long	very long
$100,000,000$	$0.027\mu s$	$0.10sec$	$2.66sec$	115.7 $days$	very long	very long
$1,000,000,000$	$0.030\mu s$	$1sec$	$29.90sec$	31.7 $years$	very long	very long

4 The curse of Exponential time

4.1 Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

4.2 Find nth Fibonacci Number F_n

Implementation the recursive definition of F_n

Algorithm 1 :Recursive Fibonacci Number Computation

```
function FIB1(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return FIB1(n − 1) + FIB1(n − 2)
```

- Is it correct?
- How much time it takes to compute F_n?
- Can we do better?

Let $T(n)$ be the number of operations on input n

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 3 & \text{if } n > 2 \end{cases}$$

By definition, we have $T(n) > F_n$

Bad news: The running time of the algorithm grows as fast as $F_n.F_n \geq 2^{\frac{n}{2}}$ **exponential** in n (prove by Induction)

4.3 Find F_n: The curse of Exponential time

$$T(n) > F_n \geq 2^{\frac{n}{2}}$$

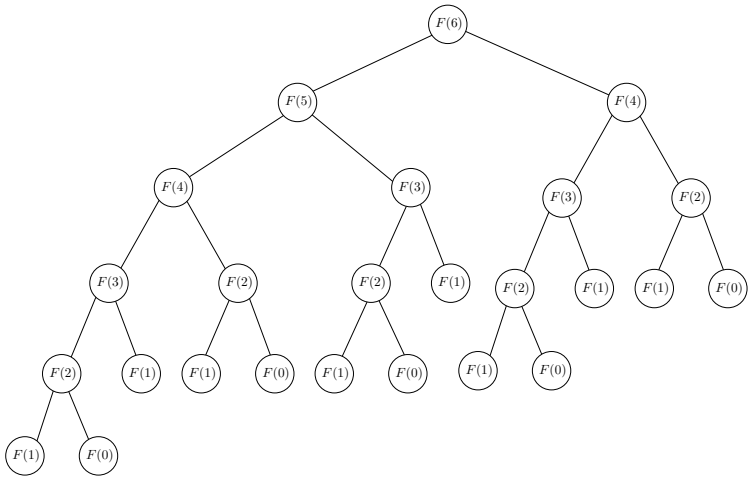
- $n = 300$. Computing F_{300} takes (much) more than 2^{150} ops
- $64THz$ computer (64 trillions operations per second)
- needs $2^{104}s > 10^{27}h > 10^{23}$ years

- Computer speeds have been doubling roughly every 18 months.
- the running time of $fib1(n)$ is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute F_{n+1} than F_n.
- Under Moore's law, computers get roughly 1.6 times faster each year. So if we can compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on: just one more Fibonacci number every year! **Such is the curse of exponential time**.

How can we improve it?

4.4 Find F_n : A polynomial Algorithm

We saw that the recursive method for calculating a Fibonacci number has exponential runtime. Recursively calculating Fibonacci number takes exponential time. Recursion tree for $n = 6$ is shown below.



Algorithm 2 Iterative Fibonacci Number Computation with Bottom-up Approach

```
function FIB2(n)
    for i = 1 to n do                          ▷ Initially F[i]'s are unknown
        F[i] ← ∞
    F[0] ← 0, F[1] ← 1
    for i = 2 to n do
        F[i] ← F[i − 1] + F[i − 2]
    return F[n]
```

- Correct by definition of F_n
- Runtime is n-1 additions
- Reasonable to compute F_{200} or even $F_{200,000}$

4.5 Sloppy Runtime Analysis

- **Useful Simplification:** Count the number of basic operations assuming them taking a constant amount of time
- Is it Practical? looking back at Fibonacci algorithms
- It is reasonable to treat *one word* additions as a single computer step (32 bits or 64 bits)
- The *nth* Fibonacci number is about $0.694n$ bits long, F_{1000} is 694 bits long

- Such a more honest and careful analysis yields that computing F_n takes about cn^2 bits additions
- Moral of the story is : We have to be careful in declaring and selecting elementary operations

5 Asymptotic Lower Bounds - Big Ω Notation

Definition 2 (Ω (Big Omega)). *A function $g(n) \in \Omega(f(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that*

$$g(n) \geq cf(n) \qquad \forall\, n \geq n_0$$

- Written as: $g(n) \in \Omega(f(n))$
- g in this case as being asymptotically lower bounded by f.
- $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$
- The definition of Ω works just like O(.), except that the function g(n) is bounded from below, rather than from above.

5.1 Examples

1. $3n^2 + 4n + 5 \in \Omega(n^2)$
2. $3n^2 + 4n + 5 \in \Omega(n)$
3. $3n^2 + 4n + 5 \neq \Omega(n^3)$
4. **In general** $f(n) = pn^2 + qn + r \implies f(n) \in \Omega(n^2)$, where p, q, and r are positive constants.
   Establishing the upper bound involved "inflating" the terms in f(n) until it looked like a constant times n^2.
   For lower bound we need to reduce the size of f(n) until it looks like a constant times n^2. $f(n) = pn^2 + qn + r > pn^2$ where $n \geq 0$ which meets what is required by the definition of $f \in \Omega(n^2)$ with $c = p > 0$.

6 Asymptotic Tight Bounds - Big Θ Notation

Definition 3 (Θ (Big Theta)). *A function $g(n)$ is $\Theta(n)$ iff there exists two positive real constants c_1 and c_2 and a positive integer n_0 such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$ $\forall n > n_0$.*
$n_0 = max[n_1, n_2]$

- Written as: $g(n) \in \Theta(f(n))$
- f in this case as being is an asymptotically tight bound for g.
- $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

- $\Theta(g(n)) \in O(g(n)) \bigcap \Omega(g(n))$
- Asymptotically tight bounds on worst-case running times are nice things to find, since they characterize the worst-case performance of an algorithm precisely up to constant factors.

6.1 Examples

1. $3n^2 + 4n + 5 \in \Theta(n^2)$
2. $3n^2 + 4n + 5 \notin \Theta(n^3)$
3. $3n^2 + 4n + 5 \notin \Theta(n)$
4. $f(n) = pn^2 + qn + r$, $f(n) \in \Omega(n^2)$, and $f(n) \in O(n^2) \implies f(n) \in \Theta(n^2)$, where p, q, and r are positive constants.

7 Little Oh - o Notation

Definition 4. *A function $g(n) \in o(f(n))$ if for every constant $c > 0$, there exists a constant $n_0 \geq 0$ such that*

$$g(n) \leq cf(n) \qquad \forall\, n \geq n_0$$

- Written as: $g(n) \in o(f(n))$
- This is used to show that g grows much much slower than f.
- $f(n) \in o(g(n)) \Leftrightarrow (f(n) \in O(g(n)) \wedge f(n) \notin \Theta(g(n)))$
- An equivalent formulation (when $f(n)$ is non-zero) is given as

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

7.0.1 Examples

1. $3n^2 + 4n + 5 \notin o(n^2)$
2. $3n^2 + 4n + 5 \in o(n^3)$
3. $3n^2 + 4n + 5 \notin o(n)$

8 Little omega - ω Notation

Definition 5. *A function $g(n) \in \omega((f(n))$ if for every constant $c > 0$, there exists constant $n_0 \geq 0$ such that*

$$g(n) \geq cf(n) \qquad \forall\, n \geq n_0$$

- Written as: $g(n) \in \omega(f(n))$

- In this case f grows much faster than g.
- $f(n) \in \omega(g(n)) \Leftrightarrow (f(n) \in \Omega(g(n)) \wedge f(n) \notin \Theta(g(n)))$

8.1 Examples

1. $3n^2 + 4n + 5 \notin \omega(n^2)$
2. $3n^2 + 4n + 5 \in \omega(n)$
3. $3n^2 + 4n + 5 \notin \omega(n^3)$

9 Properties of Asymptotic Growth Rates

9.1 Transitivity

1. If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
2. If $f \in \Omega(g)$ and $g \in \Omega(h)$, then $f \in \Omega(h)$
3. if $f \in \Theta(g)$ and $g \in \Theta(h)$, then $f \in \Theta(h)$

9.2 Additivity

1. If $f \in O(h)$ and $g \in O(h)$, then $f + g \in O(h)$
    - More generally, if k is a fixed constant and $f1, f2...fk$ and h are functions such that $f_i \in O(h)$ for all i. Then $f1 + f2... + fk \in O(h)$

9.3 Symmetry

1. If $f \in \Theta(g)$ then $g \in \Theta(h)$

9.4 Transport Symmetry

1. If $g \in \Omega(f)$ then $f \in O(g)$

9.5 Reflexivity

1. $f \in \mathcal{O}(f)$
2. $f \in \Omega(f)$
3. $f \in \Theta(f)$