Algorithm Analysis-Asymptotic Analysis

Tanjina Helaly

Algorithms

How to Measure the Time to run an algorithm?

- One **naïve** way is to implement the algorithm and run it in a machine.
- This time depends on
  - the speed of the computer,
  - the programming language,
  - the compiler that translates the program to machine code.
  - the program itself.
  - And many other factors
- So, you may get different time for the same algorithm.
- Hence, **not a good tool to compare different algorithm**.
- To overcome these issues, need to **model Time complexity.**

Time Complexity

- Developing a **formula** for **predicting** ***how fast*** an algorithm is, based on the **size of the input**
  - To compare algorithms
  - Measure of efficiency/goodness of algorithm
- 3 types of complexity
  - Best case
    - Lower bound.
    - Minimum number of steps/operations to execute an algorithm.
    - Measure the minimum time required to run an algorithm.
    - Not a good measure of Algorithm's performance.

Time Complexity

- Worst case
  - Upper Bound
  - Maximum # of operations/time required to execute
  - Main focus
  - Reduce risks as it gives the highest time of algorithm execution
- Average case
  - the amount of some computational resource (typically time) used by the algorithm, **averaged over all possible inputs**.
  - Difficult to determine
  - Typically follow the same curve as worst

Time Complexity

- The **best**, **worst**, and **average** case time complexities for any given algorithm are **numerical functions over the size of possible problem instances**.
- However, it is **very difficult** to work **precisely** with these functions,
  - Depends on specific input size.
  - Not a smooth curve
  - Require too much detail
  - So, need more **simplification** or **abstraction**.

Asymptotic analysis

- We ignore **too much details** steps such as
  - Initialization cost
  - Implementation of specific operation.
- Rather we **focus on**
  - how the time change if input doubles/triples
  - Or how many more operations do we need for that change.

A SAMPLE COMPLEXITY EQUATION

- $f(n) = 2n^2 + 10n + 3$
- How does each term effected by change of n?
  - For n=1000
    - $2n^2$=2000000
    - 10n = 10000
    - Ratio: $2n^2/10n = 200$ -> 0.5% of $2n^2$
  - For n = 1000000
    - $2n^2$ = 2000000000000
    - 10 n = 10000000
    - Ratio: $2n^2/10n = 200000$ -> 0.0005% of $2n^2$
- So, as **n grows** the **lower order term become insignificant.**

A sample complexity equation

- Do similar analysis for the equations below.
  - $f(n) = 2n^2 + 10n + 3$
  - $f(n) = 5n^2 + 6n + 35$
- Will you get different result?
  - No,
  - As $n \to \infty$, all lower order terms become so insignificant that we can just ignore them.
- Order of growth:
  - How the time grow with input size
  - Leading term/Highest order term in the equation

More to think

- Tell me what types of equation are the following 2?
  - $f(n) = 2n^2$
  - $f(n) = 5n^2$
- Does the coefficient impact the characteristics of the curve?
- Can we ignore the coefficient?
  - YES,
  - Why:
    - constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

More to think

- So, in terms of algorithm analysis, we can think
  - $f(n) = 2n^2 + 10n + 3 \sim \mathbf{n^2}$ **or** $\mathbf{\Theta(n^2)}$
  - $f(n) = 5n^2 + 6n + 35 \sim \mathbf{n^2}$ **or** $\mathbf{\Theta(n^2)}$

Asymptotic Analysis

- Classifying functions into **different category**.
- Formally, given functions *f* and *g* of a variable *n*, we define a binary relation
  - $f \sim g$ (as $n \rightarrow \infty$)
- **f** and **g** grows the same way as their input grows.
- In Asymptotic Analysis,
  - we evaluate the **performance** of an algorithm **in terms of input size**
  - Do not measure the actual running time
  - We calculate, **how does** the time (or space) taken by an algorithm **increases with the input size**.

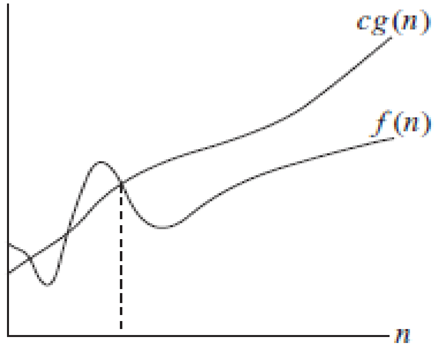
Asymptotic Notation

- There are 3 Asymptotic notations as follows:
- *Big Theta*
  - *$f(n) = \Theta(g(n))$ means $c1 \cdot g(n)$ is an upper bound on $f(n)$ and $c2 \cdot g(n)$ is* a lower bound on *$f(n)$, for all $n \geq n_0$. Thus there exist constants c1 and c2* such that *$f(n) \leq c1 \cdot g(n)$ and $f(n) \geq c2 \cdot g(n)$. This means that $g(n)$ provides* a nice, tight bound on *$f(n)$*.
- *Big Oh*
  - *$f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists* some constant *c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. ,$n \geq n_0$ for some constant n_0).*
- *Big Omega*
  - *$f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists* some constant *c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.*

Big O notation

- The Big O notation defines an **upper bound** of an algorithm, it bounds a function only from above.

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 <= f(n) <= cg(n)$ for all $n >= n_0\}$



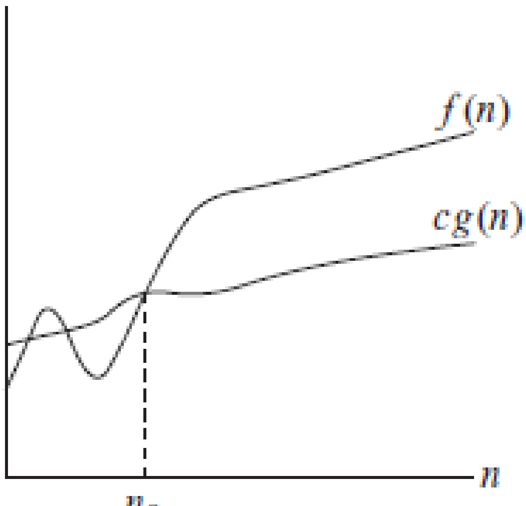
$f(n) = O(g(n))$

Big Omega - Ω Notation

- Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

  $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 <= cg(n) <= f(n)$ for all $n >= n_0\}$.

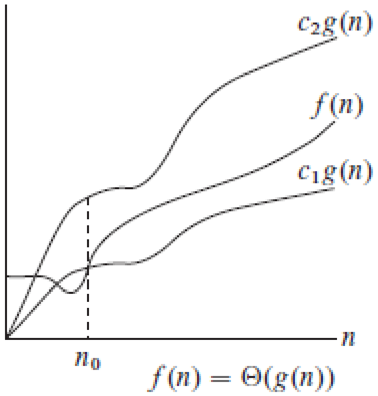
- Similar to the best case, the Omega notation is the least used notation among all three.



$f(n) = \Omega(g(n))$

Big Theta- Θ notation

- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n >= n_0\}$



- The theta notation bounds a functions from **above** and **below**, so it defines exact **asymptotic** behavior.
- *Thus g(n) provides* a nice, **tight bound** on *f(n)*.
- Note:
  - ***The definition of asymptotic is a line that approaches a curve but never touches.***

Big Theta- Θ notation

- Easy way to get Theta notation is to
  - Drop lower-order terms
  - Ignore leading constant.
- So, $an^3+bn+c = = \Theta(n^3)$

How do we show that?

- $f(n) = 2n^2 + 10n + 3 = \Theta(n^2)$
- To prove that we need to find a c_1, c_2 and n_0 so that
  - $c_1 n^2 <= 2n^2 + 10n + 3 <= c_2 n^2$ for $n >= n_0$
- Obviously
  - $2n^2 <= 2n^2 + 10n + 3$ for any n
    - $c_1 = 2$
- To calculate c_2 lets increase the power of each term to the highest power
  - $2n^2 + 10n + 3 <= 2n^2 + 10n^2 + 3n^2 = 15n^2$
  - $c_2 = 15$

Another example

- $f(n) = 5n^2 + 6n + 35 = \Theta(n^2)$
- To prove that we need to find a c_1, c_2 and n_0 so that
  - $c_1 n^2 <= 5n^2 + 6n + 35 <= c_2 n^2$ for $n>=n_0$
- Obviously
  - $5n^2 <= 5n^2 + 6n + 35$ for any $n>0$
    - $c_1 = 5$
- To calculate c_2 lets increase the power of each term to the highest power
  - $5n^2 + 6n + 35 <= 5n^2 + 6n^2 + 35n^2 = 46n^2$
  - $c_2 = 46$ for $n>0$

More Examples

- f(n) = $3n^3 + 5n + 6 = \Theta(n^3)$
- f(n) = $n \log n + 10n = \Theta(n \log n)$
- f(n) = $2 + 1/n = \Theta(1)$

Relationship among those notation

- For any two functions $f(n)$ and $g(n)$,
  - we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Different functions

- *Constant functions*, $f(n) = \mathrm{c}$
- *Logarithmic functions*, $f(n) = \log n$
- *Linear functions*, $f(n) = n$
- *Superlinear functions*, $f(n) = n \lg n$
- *Quadratic functions*, $f(n) = n^2$
- *Cubic functions*, $f(n) = n^3$
- *Exponential functions*, $f(n) = c^{\mathrm{n}}$
- *Factorial functions*, $f(n) = n!$
- $n! \geq 2^n \geq n^3 \geq n^2 \geq n \log n \geq n \geq \log n \geq \mathrm{c}$

Try these

- Is $2^{n+1} = O(2^n)$?
- Is $2^{2n} = O(2^n)$?
- For each of the following pairs of functions, either $f(n)$ is in $O(g(n))$, $f(n)$ is in $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. Determine which relationship is correct and briefly explain why.
  - $f(n) = \log n^2$; $g(n) = \log n$
  - $f(n) = \sqrt{n}$; $g(n) = \log n^2$
  - $f(n) = \log^2 n$; $g(n) = \log n$
  - $f(n) = n$; $g(n) = \log^2 n$
  - $f(n) = n \log n + n$; $g(n) = \log n$
  - $f(n) = 10$; $g(n) = \log 10$
  - $f(n) = 2^n$; $g(n) = 10n^2$
  - $f(n) = 2^n$; $g(n) = 3^n$

The Big Oh Notations

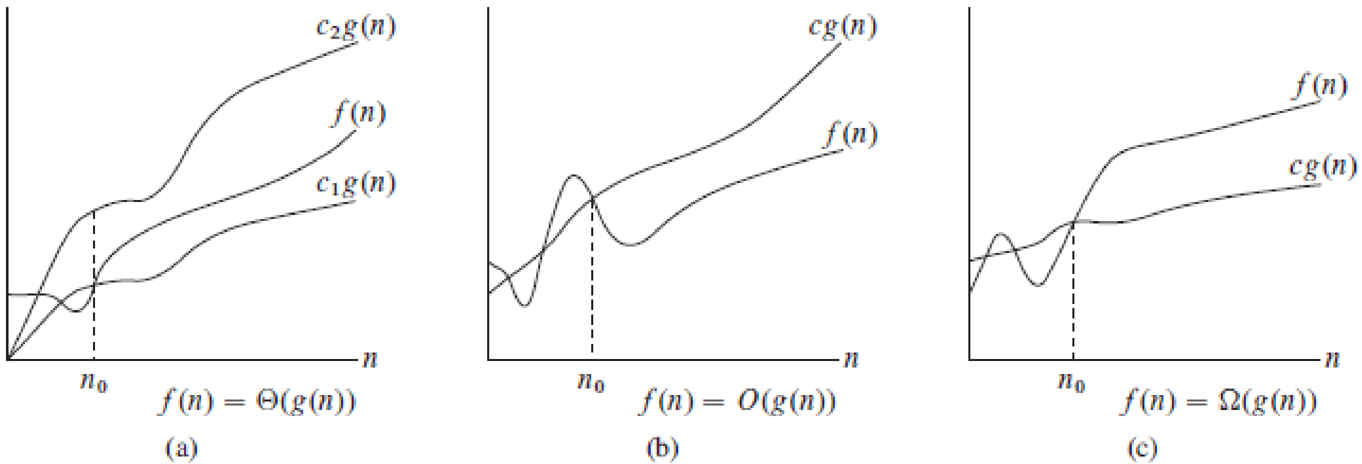


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

THE BIG OH NOTATION

- $3n^2 - 100n + 6 = O(n^2)$, ***Because for c = 3, $3n^2 > 3n^2 - 100n + 6$;***
- $3n^2 - 100n + 6 = O(n^3)$, ***Because for c = 1, $n^3 > 3n^2 - 100n + 6$ when $n > 3$;***
- $3n^2 - 100n + 6 \neq O(n)$, ***Because for any c, $c \times n < 3n2$ when $n > c$;***

- $3n^2 - 100n + 6 = \Omega(n^2)$, ***Because for c = 2, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;***
- $3n^2 - 100n + 6 \neq \Omega(n^3)$, ***Because for c = 3, $3n^2 - 100n + 6 < n^3$ when $n > 3$;***
- $3n^2 - 100n + 6 = \Omega(n)$, ***Because for any c, $cn < 3n^2 - 100n + 6$ when $n > 100c$;***

- $3n^2 - 100n + 6 = \Theta(n^2)$, ***Because both O and Ω apply;***
- $3n^2 - 100n + 6 \neq \Theta(n^3)$, ***Because only O applies;***
- $3n^2 - 100n + 6 \neq \Theta(n)$, ***Because only Ω applies.***

Reference

- Chapter 2 + 3 (Cormen)