Asymptotic Analysis

Asymptotic (مقارب) analysis measures the efficiency of an algorithm as the input size becomes large.

It is actually an **estimation** technique. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

- The critical resource for a program is -most often- **running time**.
- The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
  - ***cn*** (for ***c*** any positive constant) ➔ **linear** growth rate or running time.
  - n^2 ➔ **quadratic** growth rate
  - 2^n ➔ **exponential** growth rate.

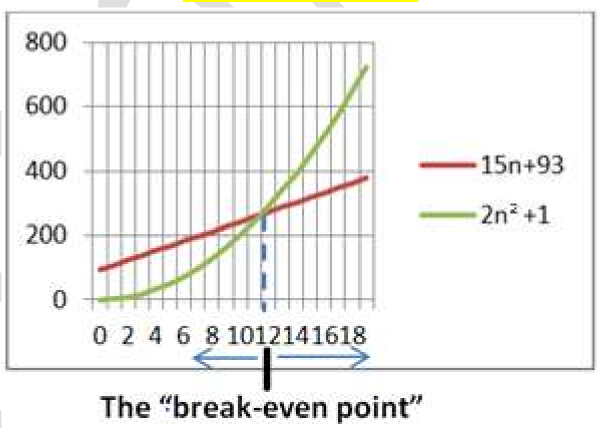
Worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well.

Example:

Assume: Algorithm A: time = $15n + 93$

Algorithm B: time = $2n^2 + 1$ **which is faster?**

Graph using Excel



We are interested for large **n**

*** For sufficiently large n, algorithm A is faster**

*** In the long run constants do not mater.**

Upper bound for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have. ➔ **big-O notation**.

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\mathbf{O}(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \le cf(n)$ for all $n > n_0$.

- Prove that **$15n + 93$** is **$O(n)$**

  We must show +ve **c** and **n_0** such that **$15n + 93 \le c(n)$** for **$n \ge n_0$**

  <provided **$n= 93$**> ➔ **$15n+n$** ➔ **$16n \le cn$** → <provided **$c = 16$**>

  So for **$c=16$** and **$n_0 = 93$** ➔ **// proved**

  Graph using Excel

- Prove that **$2n^2+1 = O(n^2)$**

  Must show +ve **c**, **n_0** such that **$2n^2+1 \le c(n^2)$ for $n \ge n_0$**

  $2n^2+1$ <provided n=1>

  $2n^2+ n^2$ ➔ $3n^2$ <provided c=3>

  $2n^2+1 \le 3n^2$

  So, **$c=3$, $n_0=1$** // proved

  Graph using Excel

The **lower bound** for an algorithm is denoted by the symbol **Ω**, pronounced "big-Omega" or just "Omega."

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\mathbf{\Omega}(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \ge cg(n)$ for all $n > n_0$.

- Prove that **$15n+93$** is **$\Omega(n)$**

  We must show +ve **c** and **n_0** such that **$15n+93 \ge c(n)$** for **$n \ge n_0$**

  <because 93 is +ve> $\ge$ **$c(n)$** ➔ <provided **$c=15$**> ← so any **$n_0 > 0$** will do

  So **$c=15$, $n_0=1$** // proved

  Graph using Excel

- Prove that **$2n^2+1$** is **$\Omega(n^2)$**

  Must show +ve **c** and **n_0** such that **$2n^2+1 \ge cn^2$** for **$n \ge n_0$**

  <because 1 is +ve>

  So **$c=2$, $n_0=1$** // proved

  Graph using Excel

When the **upper** and **lower bounds** are the same within a constant factor, we indicate this by using **Θ (big-Theta)** notation.

T(n) = **Θ(g(n)) iff** T(n) = **O(g(n))** **and** T(n) = **Ω (g(n))**

Example: Because the **sequential search algorithm** is both in **O(*n*)** and in **Ω(*n*)** in the average case, we say it is **Θ(*n*)** in the average case.

Simplifying Rules

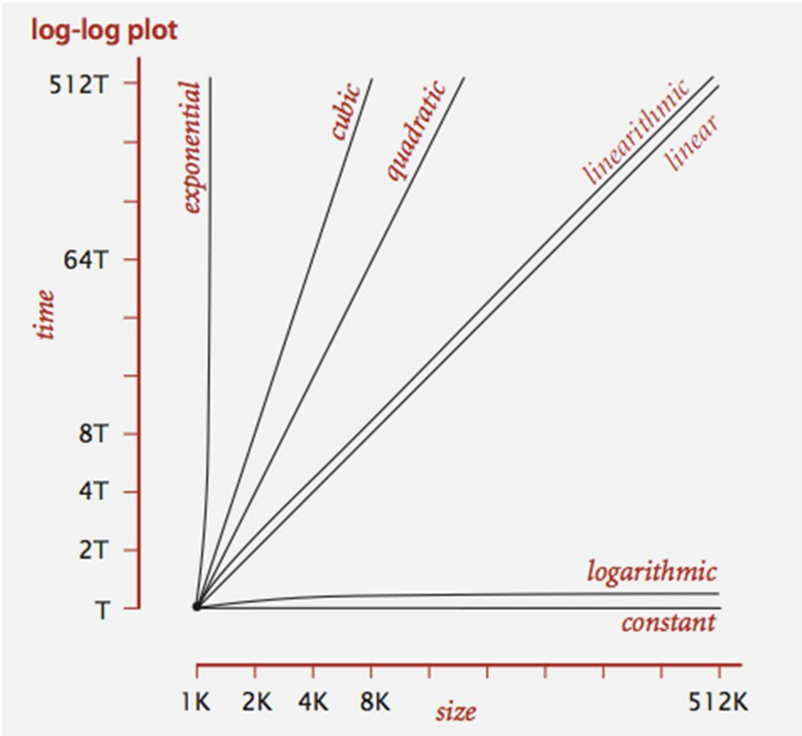
1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.
2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.
3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n) + f_2(n)$ is in $\mathrm{O}(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n)f_2(n)$ is in $\mathrm{O}(g_1(n)g_2(n))$.

- **Rule (2)** is that you can ignore any multiplicative constants.
- **Rule (3)** says that given two parts of a program run in sequence, you need to consider only the more expensive part.
- **Rule (4)** is used to analyze simple loops in programs.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function.

Order of growth of some common functions:

$$O(1) \le O(\log_2 n) \le O(n) \le O(n \log_2 n) \le O(n^2) \le O(n^3) \le O(2^n)$$



Typical orders of growth

If the problem size is always small, you can probably ignore an algorithm's efficiency

Limitations of big-O analysis:

- Overestimate.
- Analysis assumes infinite memory.
- Not appropriate for small amounts of input.
- The constant implied by the Big-Oh may be too large to be ignored (**2*N* log *N*** *vs.* **1000*N***)