Introduction to Run-Time Analysis, Complexity of Algorithms and Big-O Notation

---

Analyzing complexities of algorithms and determining their running times is one of the major branches of CS. The most famous problem of theoretical CS is related to this question.

Motivating Example: Delivering Packages in 3 Different Ways

Big-O Notation: Let f, g be two functions from $\mathbb{N}$ to $\mathbb{N}$. We say that f is "big-oh" of g (written $f = O(g)$, or $f \in O(g)$) if ....

Remark 1: A useful way of determining big-O of a function:

Remark 2: The big-O notation is not sensitive to multiplicative constants, lower order terms, or the basis of a logarithm.

What does the Big-O Notation Try to Capture?

Example: a) $f(n) = 2n^3 + 3n^2 + 100$ b)$f(n) = n + 10\sqrt{n} + \log(n)$ c) $f(n) = 2^n + n^7 + 10^3$

Question: Suppose $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$. Is it true that $f(n)$ is $O(h(n))$?

Question: What is $O(1)$? What is $O(n)$?

Polynomial Time Algorithms:

Analyzing Number Theoretical Algorithms When we have a number theoretical algorithm, the basic unit of operation is bit-wise operations. Therefore if the input to a number-theoretical algorithm is n then the size of the input is taken to be the number of bits it takes to represent n in the memory, which is equal to ....

Example: Computational Complexity of Addition, Multiplication and Division

Example: Given a positive integer n, consider the brute-force algorithm that tries to determine whether n is a prime by testing every integer between 2 and $\sqrt{n}$. What is the computational complexity of this algorithm in the worst case?