First we had Selection Sort, which ran in O(n^2) time. Then we talked about Merge Sort, which runs in O(n log n) time. But what do these weird terms "O(n^2)" and "O(n log n)" mean exactly?

I did mention that it means that the runtime is roughly proportional to what's inside the parentheses. But that's only part of the story. How exactly can we measure runtime? The number of executed steps? What constitutes a step?

Why can't we just cheat and say std::sort() is a single step? Well, think about it this way. Most computer processors operate on a machine language, which usually consists of an instruction set.

Each instruction does a single unit of computation: some arithmetic operation, reading or writing from/to memory or a register, moving throughout the program, calling or returning from a function, etc.

The processor is controlled by a clock which sends signal pulses at a fixed rate, usually measured in mHz. A single clock cycle "advances" the flow of the program by one step. However, not all processors are designed to execute an instruction all at once in a cycle.

Some take multiple cycles to execute each instruction due to the number of steps involved in the computation (e.g. retrieving the instruction, decoding its nature, getting the needed values, etc.).

Others optimize by using a pipeline that does several different instructions simultaneously at different stages of the computation to make better usage of the stages.

There are also further optimizations such as branch prediction and out-of-order processing, but those are beyond the scope of this article. Most computers nowadays are divided into Reduced or Complex Instruction Set Computers (denoted RISC and CISC respectively).

The only difference is that RISC instructions take a single clock cycle and are thus simpler and more rudimentary. CISC instructions, on the other hand, are more sophisticated and can support more large "units" that might compose of multiple smaller sub-instructions.

These then take multiple clock cycles to execute. Either way, the main thing to note is the number of clock cycles each instruction takes is bounded by a fixed constant.

So when we measure the runtime of programs, what we're really measuring is the number of clock cycles, and by extension the number of rudimentary arithmetic, data, or control operations we are doing. But it is hard, if not impossible, to get exact numbers.

What we can do however is look at the correlation between runtime and the input size. That's where the Big-O notation comes in. We will define $F(x) = O(G(x))$ where F and G are functions, if F(x) tends towards being bounded above by a fixed multiple of G(x).

In other words, there exists a fixed constant k such that $F(x) \leq k \cdot G(x)$ as x grows large. So for example the function $F(x) = 2x^2 + x + 1$ is $O(x^2)$ since we can see that as x grows large, $F(x) \leq 3 \cdot x^2$.

But $G(x) = 4x^4 + 2x + 9$ is not $O(x^2)$ because we can prove that for any constant k, G(x) exceeds kx^2 at some positive x value. Essentially, if $F(x) = O(G(x))$ if its growth rate is at most that of G(x).

Similarly, we can define $F(x) = \Omega(G(x))$ if there is a constant k such that $F(x) \geq k \cdot G(x)$ as x grows large; F(x) grows at least the speed of G(x). And if F(x) is both O and Ω of G(x) then $F(x) = \Theta(G(x))$; F(x) grows the same rate as G(x) and is roughly proportional.

If $F(x) = O(G(x))$ but F(x) is not $\Theta(G(x))$ then we know that F(x) grows strictly slower than G(x). In terms of runtimes, this means that the algorithm represented by F(x) is more efficient than the algorithm represented by G(x).