Algorithm analysis

The random-access machine (RAM) models the essential features of the traditional serial

computer. The RAM is modelled by two synchronous interconnected FSMs, a central processing unit (CPU) and a random-access memory. The CPU implements a fetch-and-execute cycle in which it alternately reads an instruction from a program stored in the random-access memory (the stored-program concept) and executes it.

Assumptions :

* Each ``simple'' operation (+, \*, -, =, if, call) takes exactly 1 time step.
* Loops and subroutines are *not* considered simple operations. Instead, they are the composition of many single-step operations. It makes no sense for ``sort'' to be a single-step operation, since sorting 1,000,000 items will take much longer than sorting 10 items. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
* Each memory access takes exactly one time step, and we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk, which simplifies the analysis.

The RAM is a simple model of how computers perform. A common complaint is that it is too simple, that these assumptions make the conclusions and analysis too coarse to believe in practice. For example, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. Memory access times differ greatly depending on whether data sits in cache or on the disk, thus violating the third assumption. Despite these complaints, the RAM is an excellent model for understanding how an algorithm will perform on a real computer. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. We use the RAM model because it is useful in practice.

Two other models :

Finite state machines

Turing machines

1.What does the Big-Oh notation mean? You should both have an intuitive understanding for what *O(f(n))* means and know what it means to a mathematician.

T(n) = 1+n

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the T(n) function. In other words, as the problem gets larger, some portion of the T(n) function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The order of magnitude function describes the part of T(n) that increases the fastest as the value of *n* increases. Order of magnitude is often called Big-O notation (for “order”) and written as O(f(n)). It provides a useful approximation to the actual number of steps in the computation. The function f(n) provides a simple representation of the dominant part of the original T(n).**O** (pronounced big-oh) notation is the formal way of expressing the upper bound of an algorithm’s running time.

In the following, what statements are true?

* 1. 2*n* + 17 is in *O(n). ----TRUE*
  2. 10*n* is in *O(n^*2*)*.
  3. *n^*2 + log *n* is in *O(n*^2*)*

Why do we talk to “time complexity” but report the (asymptotic) number of steps of an algorithm?

When we talk about efficiency of an algorithm, we generally refer to the resources (time, space etc.) that the algorithm will consume for a given input. Especially, we usually are interested in how an algorithm will behave (as in how much time or space will be consumed) for voluminous input size.

A solution to predict the behaviour (say running time) is to come up with a function ƒ(*n*) that emulates the the time taken by the algorithm  for a given input of size n*;* it can be done by implementing the algorithm, executing it for different input-sizes while recording the processing-time taken, and plotting a graph (input-size on x-axis and running-time on y axis). A clear disadvantage in this approach is the dependence on too many external factors — programming language used for implementation; computing machine, its architecture and operating system etc. used for execution; how many other programs are running on the same machine while execution; and so on. We would like to abstract the analysis of the algorithm from all these external factors that do not have anything to do with the algorithm itself.Mathematics provides the abstraction technique required while analysing efficiency of an algorithm, namely — Asymptotic Notation.

Consider the problem om multiplying two *n*-digit integers. What is the time complexity this problem, if we take single-digit multiplication and addition as the elementary operations?

--no idea much