The Basic Game Plan of Algorithm Complexity Analysis

Primitive operations

When we analyze an algorithm A for a specific problem, we want to characterize the running time of A as a function of its input (and later only of the input size). To do this, it is convenient to identify a few operations, or steps, in the algorithm, which are executed more often, or roughly as often, as any other steps are executed. For example, in your typical sorting algorithm, the number of comparison operations done by the algorithm is larger or equal to the number of other (bookeeping or data movement type) operations done (this is actually not true of insertion sort). So we try to find the dominant operations in the algorithm; we call them the *primitive operations*, and we generally ignore all the other operations in the algorithm. So when we speak of the time needed by the algorithm, we geneally just mean the number of primitive operations that the algorithm executes. Of course, that number depends on the specific input to the algorithm, and generally will rise as the size of the input increases. We get to that now.

A formal statement of worst case complexity

We have some problem, say sorting numbers, or computing some function of a set of numbers, or inverting a matrix etc. We use P to denote an abstract problem. Then we have an algorithm that solves instances of problem P. We let A_P denote the (deterministic) algorithm that solves problem P. One might expect that the running time of algorithm A_P increases as the size of the problem instances increase. For example, it is reasonable that an algorithm takes more time to sort 100,000,000 numbers than to sort 100 numbers, although the actual time might depend on the specific list of 100,000,000 and 100 numbers. We want notation to talk about these kinds of issues.

Let $D_P(n)$ be the set of all possible inputs to problem P, of length n. So for example, when P is the problem of sorting numbers and n is a fixed value, $D_P(n)$ is the set of all lists of n numbers. This is an infinite set, but one can conceive of it and define it, as we have.

Given algorithm A_P for problem P, and given a specific input $I \in D_P(n)$, the running time of A_P on input I is denoted $T_{A_P}(I)$ or $T_A(I)$ for short. That is, when algorithm A_P is given input I, it takes exactly $T_A(I)$ primitive operations for algorithm A_P to solve the problem.

Now for any particular value of n, there are some inputs on which A_P may run quickly, and some on which it may run slowly, although some algorithms take the same time on all inputs of the same length. We want to express the running time of A_P as a function of n alone, not on the specific inputs. But which input of size n should we pick? That leads to the notion of *worst-case*. For each n, we pick that input which makes algorithm A_P run the longest. We define the *work* that A_P does as

$$W_{A_P}(n) = \max_{I \in D_P(n)} [T_A(I)].$$

That is, we form the work function $W_{A_P}(n)$, where for every value of n, the function is determined by the problem instance that makes A_P run the longest. That function is also called *the worst case running time* of A_P. The worst-case approach may seem a bit perverse or at least pessemistic, but if you have an application where you allow $W_{A_P}(n)$ time to solve instances of problem P, then you can be sure that no matter what the input is, algorithm A_P will succeed in solving the problem within the allotted time.

Complexity of problem P

Function W_{A_P} is for a specific algorithm A_P solving problem P. How can we express the *inherent complexity* of solving problem P, i.e., without making reference to a specific algorithm? We define the function

$$W_P(n) = \min_{A_P} [W_{A_P}(n)].$$

That is, we form the function $W_P(n)$, called *the worst case complexity of problem* P by assuming, for every value of n, that we have the best worst-case algorithm working at that value of n. That means that

$$W_P(n) \leq W_{A_P}(n)$$

for any specific algorithm A_P solving problem P.

Now, although we can define the functions $W_{A_P}(n)$ and $W_P(n)$, it is very hard to actually determine these functions for all but very simple algorithms and problems. Therefore we have to *approximate* these functions. We wish to approximate the functions from above and below, that is by functions that grow faster and that grow slower than the functions of interest. For example, we seek functions $B(n)$ and $U(n)$ such that $B(n) \leq W_{A_P}(n) \leq U(n)$, for every value of n. And we want $U(n) - B(n)$ to be small. We can't always do that either, but we try. It is in attempting to get these bounding functions that we use the O, Ω, Θ and o notations, that are explained in the book.