Analysing Algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, a most efficient one can be easily identified. Such analysis may indicate more than one viable candidate, but several inferior algorithms are usually discarded in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that will be used, including a model for the resources of that technology and their costs. For most of this tutorial, we shall assume a generic one-processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, one should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis.The RAM model contains instructions commonly found in real computers: arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

When working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time-clearly an unrealistic scenario.)

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms ″running time″ and ″size of input″ more carefully.

The best notion for input size depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input-for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The running time of an algorithm on a particular input is the number of primitive operations or ″steps″ executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudo-code. One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i , where $c_i$is a constant.

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs c_i to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each j = 2, 3, . . . , n, where n = length[A], we let t_j be the number of times the while loop test in line 5 is executed for that value of j. When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

INSERTION-SORT(A)	cost	time
1. for j ← 2 to length[A]	c_1	n
2. do key ← A[j]	c_2	n - 1
3. Insert A[j] into the sorted sequence A[1...j-1].	0	n - 1

4.	$i \leftarrow j - 1$	c_4	$n - 1$
5.	while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6.	do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7.	$i \leftarrow i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8.	$A[i+1] \leftarrow key$	c_8	$n - 1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes ci steps to execute and is executed n times will contribute cin to the total running time. To compute T(n), the running time of INSERTION-SORT, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n} (t_j - 1) + c_7\sum_{j=2}^{n} (t_j - 1) + c_8(n - 1)$$

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each j = 2, 3, . . . , n, we then find that $A[i] \le key$ in line 5 when i has its initial value of j - 1. Thus tj = 1 for j = 2, 3, . . . , n, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

This running time can be expressed as an + b for constants a and b that depend on the statement costs ci ; it is thus a linear function of n.

If the array is in reverse sorted order-that is, in decreasing order-the worst case results. We must compare each element A[j] with each element in the entire sorted subarray A[1 j - 1], and so $t_j = j$ for j = 2, 3, . . . , n. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j - 1) = \frac{n(n+1)}{2}$$

we find that in the worst case, the running time of INSERTION-SORT is

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n - 1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8).$$

This worst-case running time can be expressed as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i ; it is thus a quadratic function of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting ″randomized″ algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. Though, we usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n. We give three reasons for this orientation.

• The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.

• For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some searching applications, searches for absent information may be frequent.

• The ″average case″ is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[1 j - 1] to insert element A[j]? On average, half the elements in A[1 j - 1] are less than A[j], and half the elements are greater. On average, therefore, we check half of the subarray A[1 j - 1], so tj = j/2. If we work out the resulting average-case running time, it turns out to be a quadratic function of the input size, just like the worst-case running time.

One problem with performing an average-case analysis is that it may not be apparent what constitutes an ″average″ input for a particular problem. Often, we assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a randomized algorithm, which makes random choices, to allow a probabilistic analysis.

Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure. First, we ignored the actual cost of each statement, using the constants c_i to represent these costs. Then, we observed that even these constants give us more detail than we really need: the worst-case running time is $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction. It is the rate of growth, or order of growth, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large n. We also ignore the leading term′s constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. Thus, we write that insertion sort, for example, has a worst-case running time of $\theta(n^2)$ (pronounced ″theta of n-squared″).

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, this evaluation may be in error for small inputs. But for large enough inputs, a $\theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\theta(n^3)$ algorithm.

Source:

http://www.learnalgorithms.in/