ALGORITHM PERFORMANCE

Your author explains on page 36 that the linear search algorithm averages $\frac{n}{2}$ steps to successfully find something in an array of $n$ items. Think about it this way: it takes 1 comparison to find the item at position [0], 2 comparisons to find the item at position [1], 3 comparisons for position [2] and so on until $n$ comparisons for position [n−1]. The total number of comparisons is:

$$1 + 2 + 3 + ... + n$$

So the average number of comparisons over the $n$ items is:

$$\frac{1 + 2 + 3 + ... + n}{n} = \frac{n(n + 1)}{2n} = \frac{n}{2} + \frac{1}{2}$$

This derivation uses Gauss' formula (see www.jimloy.com/algebra/gauss.htm).

Your author goes on state that the time an algorithm takes to execute is proportional to the number of steps it executes.

So why the emphasis on the number of steps instead of real execution time and why is it expressed as a formula?

To see why, I have coded the linear search algorithm in C++ and executed it on a SUN SPARCstation 5 under the UNIX operating system. Here is a sample of its execution times for increasing values of $n$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.13 msec</td>
</tr>
<tr>
<td>2000</td>
<td>0.28</td>
</tr>
<tr>
<td>3000</td>
<td>0.42</td>
</tr>
<tr>
<td>4000</td>
<td>0.55</td>
</tr>
<tr>
<td>5000</td>
<td>0.78</td>
</tr>
<tr>
<td>6000</td>
<td>0.94</td>
</tr>
<tr>
<td>7000</td>
<td>1.13</td>
</tr>
<tr>
<td>8000</td>
<td>1.30</td>
</tr>
<tr>
<td>9000</td>
<td>1.47</td>
</tr>
<tr>
<td>10,000</td>
<td>1.46</td>
</tr>
<tr>
<td>11,000</td>
<td>1.64</td>
</tr>
</tbody>
</table>

The graph shows that the run time of linear search grows linearly (i.e. like a line) as $n$ increases and is approximately $\sqrt{500} n$. 
Why the emphasis on the number of steps? Because, unlike real execution time, it depends solely on the algorithm and not on the speed of your hardware or machine code. For example, the Sun SPARCstation 5 had a processor speed equivalent to about 25 MHz. Thus, the linear search algorithm running on computer today would have considerably faster execution times than those given above but they would still be proportional to $\frac{1}{2}$.

Why is the number of steps expressed as a formula? Because we want to know how the algorithm's execution time is affected by the amount of data it must process. Any algorithm works fine on a small amount of data; the mark of a good algorithm is its behavior on an enormous amount of data. It's like the old song by The Box Tops that goes “give me a ticket for an aeroplane, ain't got time to take a fast train.”

Note that the time an algorithm takes to execute has several interchangeable names, including running time, execution time, computation time and time complexity.

**Big Oh**

O is actually the Greek symbol omicron. When we read O(n), we say big oh of n or order n.

An important fact not made clear by your textbook is that Big Oh is meant to express an upper bound on the running time of an algorithm. When you say that an algorithm runs in O(n) time, you are saying that its run time (for sufficiently large values of n) is never greater than some proportion of n (i.e. $time \leq kn$ for some constant $k>0$). The run time may actually be considerably smaller than the proportion of n.

On page 71 your author states: Big Oh notation … dispenses with the constant K. When comparing algorithms … all you want to compare is how T changes for different values of N … the constant isn’t needed.

Why? To illustrate that the constants were not needed, Jon Bentley\(^1\) conducted an experiment in which he tried to make the constants of proportionality of two algorithms differ by as much as possible. He implemented an O(n) algorithm on a Radio Shack TRS-80 microcomputer using interpreted BASIC. The program’s execution time was 19,500,000n nanoseconds. He implemented an O(n^3) algorithm on a Cray-I supercomputer using optimized FORTRAN. Its execution time was 3.0 n^3 nanoseconds. The following table shows these running times extrapolated for various values of n.

\(^1\) Bentley, Jon, Programming Pearls, Addison-Wesley Publishing Co., 1986.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#1 TRS-80 BASIC</th>
<th>#2 Cray-I FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (in nanoseconds)</td>
<td>19,500,000 $n$</td>
<td>3.0 $n^3$</td>
</tr>
<tr>
<td>Execution time for values of $n$:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n = 10$</td>
<td>0.2 sec.</td>
<td>3 μsec.</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>2.0 sec.</td>
<td>3 millisec.</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>20.0 sec.</td>
<td>3 sec.</td>
</tr>
<tr>
<td>$n = 2,500$</td>
<td>50.0 sec.</td>
<td>50.0 sec.</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>3.2 min.</td>
<td>49 min.</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>5.4 hours</td>
<td>95 years</td>
</tr>
</tbody>
</table>

Even though the constant of proportionality for the $O(n)$ algorithm is 19.5 million, it still runs faster than the $O(n^3)$ algorithm for input sizes of $n \geq 2,500$.

**Graphs of Big O Times**

On page 72 your author states: $O(1)$ is excellent, $O(\log N)$ is good, $O(N)$ is fair and $O(N^2)$ is poor.

His value judgment (i.e. excellent, good, fair, poor) applies to the searching and sorting algorithms presented in the book and not to algorithms in general. In most applications you never see an $O(1)$ algorithm and an $O(n)$ algorithm is considered very good. For example, the Java compiler and your word processor are both $O(n)$ algorithms, although spell checking is an $O(n \log n)$ algorithm.

In real life, any algorithm that runs in polynomial time $O(n^p)$ or faster is considered to be of practical use. The most common polynomial time algorithms are $O(n^2)$, $O(n^3)$ or $O(n^4)$. Higher powers are almost never seen.

There are many algorithms – called exponential time algorithms – that run slower than polynomial time. A common exponential time algorithm is $O(2^n)$. Such an algorithm is called intractable because it cannot give an answer in a reasonable amount of time even for relatively small problem sizes.
This is shown in the table below, which extrapolates the running times of a polynomial-time and an exponential-time algorithm when given relatively small data sets.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of computation steps</td>
<td>$n^5$</td>
<td>$2^n$</td>
</tr>
</tbody>
</table>

Values of $n$:

<table>
<thead>
<tr>
<th>$n = 10$</th>
<th>Execution time assuming 1 billion ($10^9$) computation steps per second:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 50$</td>
<td>0.0001 sec.</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>0.3125 sec.</td>
</tr>
<tr>
<td></td>
<td>10.0 sec.</td>
</tr>
</tbody>
</table>

There are many important problems (e.g. decryption, route scheduling) for which the only known solutions are intractable algorithms.

Worst-Case and Average-Case Analysis

Consider how the linear search algorithm behaves when given the array at the right. It takes 1 comparison to find 650 in the array, two comparisons to find 500 and so on. In such a situation where the algorithm's running time depends on the data, we usually focus on the algorithm's worst-case behavior and its average-case behavior.

The worst-case behavior occurs when an algorithm executes in the worst (i.e. slowest) possible time. For example, the worst possible behavior of the linear search occurs when the search key is found in the last position of the array or not at all. In both cases the algorithm inspects, if the array has $n$ items within it, all $n$ items and so runs in $O(n)$ time.

Average-case behavior is the average performance of the algorithm over time. The only way to truly know the average run-time of an algorithm is to calculate it from actual run-times over the algorithm's life; similar to the way you keep ball players statistics.

To estimate an algorithm's average run-time, a computer scientist characterizes the behavior of the algorithm over all possible data sets, calculates the probability of each data set, calculates the algorithm's running time for each data set and then calculates the expected value of the running time using the standard statistical formula. To fully understand these calculations, you would have to have passed a course in mathematical statistics.