19.3 Big O Notation

Searching algorithms all accomplish the *same* goal—finding an element (or elements) matching a given search key, if such an element does, in fact, exist. There are, however, a number of things that differentiate search algorithms from one another. *The major difference is the amount of effort they require to complete the search.* One way to describe this effort is with **Big O notation**, which indicates how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends particularly on how many data elements there are. In this chapter, we use Big O to describe the worst-case run times for various searching and sorting algorithms.

19.3.1 *O*(1) Algorithms

Suppose an algorithm is designed to test whether the first element of an array is equal to the second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1000 elements, it still requires one comparison. In fact, the algorithm is completely independent of the number of elements in the array. This algorithm is said to have a **constant run time**, which is represented in Big O notation as ***O*(1)** and pronounced as “order one.” An algorithm that’s *O*(1) does not necessarily require only one comparison. *O*(1) just means that the number of comparisons is *constant*—it does not grow as the size of the array increases. An algorithm that tests whether the first element of an array is equal to any of the next three elements is still *O*(1) even though it requires three comparisons.

19.3.2 *O*(*n*) Algorithms

An algorithm that tests whether the first array element is equal to *any* of the other array elements will require at most $n - 1$ comparisons, where *n* is the number of array elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1000 elements, it requires up to 999 comparisons. As *n* grows larger, the *n* part of the expression $n - 1$ "dominates," and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as *n* grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described earlier) is said to be **O(*n*)**. An *O*(*n*) algorithm is referred to as having a **linear run time**. *O*(*n*) is often pronounced "on the order of *n*" or simply "order *n*."

19.3.3 $O(n^2)$ Algorithms

Now suppose you have an algorithm that tests whether *any* element of an array is duplicated elsewhere in the array. The first element must be compared with every other element in the array. The second element must be compared with every other element except the first (it was already compared to the first). The third element must be compared with every other element except the first two. In the end, this algorithm will end up making $(n-1)+(n-2)+\cdots+2+1$ or $n^2/2-n/2$ comparisons. As *n* increases, the n^2 term dominates and the *n* term becomes inconsequential. Again, Big O notation highlights the n^2 term, leaving *n*/2. But, as we'll soon see, constant factors are omitted in Big O notation.

Big O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm requires 16 comparisons; with eight elements, 64 comparisons. With this algorithm, *doubling* the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm requires eight comparisons; with eight elements, 32 comparisons. Again, *doubling* the number of elements *quadruples* the number of comparisons. Both of these algorithms grow as the square of *n*, so Big O ignores the constant, and both algorithms are considered to be $\boldsymbol{O(n^2)}$, which is referred to as **quadratic run time** and pronounced "on the order of *n*-squared" or more simply "order *n*-squared."

When *n* is small, $O(n^2)$ algorithms (on today's computers) will not noticeably affect performance, but as *n* grows, you'll start to notice performance degradation. An $O(n^2)$ algorithm running on a million-element array would require a trillion "operations" (where each could execute several machine instructions). We tested one of this chapter's $O(n^2)$

algorithms on a current desktop computer and it ran for seven minutes. A billion-element array (not unusual in today’s big data applications) would require a quintillion operations, which on that same desktop computer would take approximately 13.3 years to complete! $O(n^2)$ algorithms, unfortunately, are easy to write, as you’ll see in this chapter. You’ll also see algorithms with more favorable Big O measures. These efficient algorithms often take a bit more cleverness and work to create, but their superior performance can be well worth the extra effort, especially as *n* gets large and as algorithms are integrated into larger programs.

19.3.4 Big O of the Linear Search

The linear search algorithm runs in $O(n)$ time. The worst case in this algorithm is that every element must be checked to determine whether the search item exists in the array. If the size of the array is *doubled*, the number of comparisons that the algorithm must perform is also *doubled*. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array. But we seek algorithms that perform well, on average, across *all* searches, including those where the element matching the search key is near the end of the array.

Linear search is easy to program, but it can be slow compared to other search algorithms. If a program needs to perform many searches on large arrays, it’s better to implement a more efficient algorithm, such as the binary search, which we present next.

Performance Tip 19.1

Sometimes the simplest algorithms perform poorly. Their virtue often is that they’re easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.