11.8 Efficiency of Algorithms: Big O

- Searching algorithms all accomplish the same goal—finding an element matching a given search key if such an element exists
- Several things differentiate search algorithms from one another
- The major difference is the amount of effort they require to complete the search
- Described with Big O notation, which indicates how hard an algorithm may have to work to solve a problem
- Depends mainly on how many data elements there are
- We use Big O to describe the worst-case run times for various searching and sorting algorithms

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
- Algorithm is completely independent of the number of elements in the array
- Said to have a constant run time, which is represented in Big O notation as ***O(1)***
- Pronounced as “order one”
- An algorithm that’s ***O(1)*** does not necessarily require only one comparison
  - Means that the number of comparisons is constant—it does not grow as the size of the array increases

O(n) Algorithms

- An algorithm that tests whether the first array element is equal to any of the other array elements requires 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 ***n*** – 1 “dominates,” and subtracting 1 becomes inconsequential
- Big O is designed to highlight dominant terms and ignore terms that become unimportant
- An algorithm that requires a total of ***n*** – 1 comparisons is said to be ***O(n)***
- Said to have a linear run time
- Pronounced “on the order of n” or simply “order n”

O(n2) Algorithms

- Suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array
  - First element must be compared with every other element in the array
  - 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 makes (n – 1) + (n – 2) + … + 2 + 1 or $n^2/2 - n/2$ comparisons
- As n increases, the n^2 term dominates, and the n term becomes inconsequential
- Big O notation highlights the n^2 term, ignoring $n/2$

O(n2) Algorithms (cont.)

- 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
- 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 algorithms grow as the square of n, so Big O ignores the constant, and both algorithms are considered to be O(n^2)
  - Referred to as quadratic run time and pronounced "on the order of n-squared" or "order n-squared."

O(n2) Algorithms (cont.)

- 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 100,000-element array using a current desktop computer, and it ran for many 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 are easy to write
- Algorithms with more favorable Big O measures 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

Big O of the Linear Search

- Linear search algorithm runs in O(***n***) time
- Worst case 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
- Can provide outstanding performance if the element matching the search key happens to be at or near the front of the array
  - However, we seek algorithms that perform well, on average, across all searches
- Easy to program, but it can be slow compared to other search algorithms

---