**Lesson 15 – Algorithm Efficiency and Sorting**

**Reading: Chapter 10, Section 1 of the text.**

**Learning Objectives:**

* **Describe how the execution times of algorithms are measured.**
* **Explain how Big-O notation is used to measure algorithm time complexity.**
* **List various growth-rate functions in order of growth.**
* **Analyze an algorithm and determine its order.**

**Measuring the Efficiency of Algorithms**

* What does it mean to compare two algorithms and conclude that one is better?
  + An analysis should focus on gross differences in the efficiency of algorithms that are likely to dominate the overall cost of a solution.
* The **analysis of algorithms** is the area of computer science that provides tools for contrasting the *efficiency* of different *methods of solution*.
  + The analysis concerns itself primarily with *significant* differences in efficiency, and not clever coding tricks that may save a little bit of time at the cost of code readability.
  + The efficient use of both time and memory is important. Computer scientists use similar techniques to analyze an algorithm’s time and space efficiency. For our purposes, we are primarily concerned with time efficiency.
* How do you compare the time efficiency of two algorithms that solve the same problem? One approach is to implement both algorithms in Java and run the programs. This approach has at least three fundamental difficulties:
  + How are the algorithms coded? You should not compare implementations, because they are sensitive to factors such as programming style that tend to cloud the issue of which algorithm is inherently more efficient.
  + What computer should you use? One computer may simply be much faster than the other, so clearly you should use the same computer for both programs. You should compare the efficiency of the algorithms independently of a particular computer.
  + What data should the programs use? There is always the danger that you will select instances of the problem for which one of the algorithms runs uncharacteristically fast. Any analysis of efficiency must be independent of specific data.

**The Execution Time of Algorithms:**

* We can informally compare different solutions to some of the ADTs that we have implemented.
* For example, consider our array-based implementation of the List ADT with the reference-based implementation.
  + In the array-based implementation, the list.get(n) method could access the *n*th item in a list directly in one step, because the item is stored in items[n-1].
  + A reference-based call to list.get(n), however, must traverse the list from its beginning until the *n*th node is reached, and so would require *n* steps.
* An algorithm’s execution time is related to the number of operations it requires. Counting an algorithm’s operations—if possible—is a way to assess its efficiency. Consider, the following code:

Node curr = head; // Constant time

**while** (curr != **null**) { // Number of nodes + 1

System.out.println(curr.item); // Constant time (each)

curr = curr.next; // Constant time (each)

} // end while

* + Assuming a linked list of n nodes, these statement require *n + 1* assignments, *n + 1* comparisons, and *n* write operations.
  + If each assignment, comparison, and write operation requires, respectively, *a*, *c*, and *w* time units, the statements require *(n + 1) \* (a + c) + n \* w* time units.
  + Thus, the time required to write *n* nodes is proportional to *n*.
  + This should make sense intuitively: It takes longer to display, or traverse, a linked list of 100 items than it does a linked list of 10 items.
* Consider an algorithm that contains nested loops:

for (i = 1 through n) { // n times

for (j = 1 through i) { // i times

for (k = 1 through 5) { // 5 times

Task T // Assume constant

} // end for

} // end for

} // end for

* + If task T requires *t* time units, the innermost loop on *k* requires *5 \* t* time units
  + The loop on *j* requires *5 \* t \* i* time units.
  + The loop on *i* requires

time units. Thus, the time is proportional to *n*2*t*

**Algorithm Growth Rates:**

* The most important thing to learn about the time requirement of an algorithm is how quickly the algorithm’s time requirement grows as a function of the problem size.
* Statements such as
  + *Algorithm A requires time proportional to n2*
  + *Algorithm B requires time proportional to n*

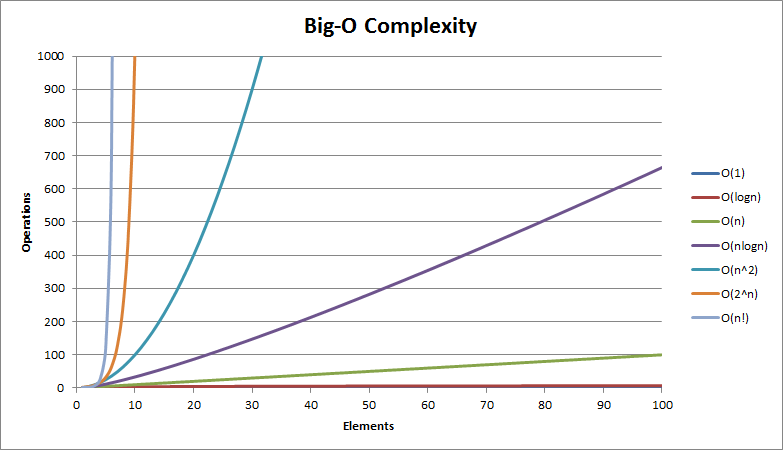
each express an algorithm’s proportional time requirement, or growth rate, and enable you to compare algorithm A with another algorithm B.

* Although you cannot determine the exact time requirement for either algorithm A or algorithm B from these statements, you can determine that for large problems, B will require significantly less time than A. That is, B’s time requirements—as a function of the problem size *n*—increases at a slower rate than A’s time requirement, because *n* increases at a slower rate than *n*2.
* Even if B actually requires *5 \* n* seconds and A actually requires *n2/5* seconds, B eventually will require significantly less time than A, as *n* increases.

**Order-of-Magnitude Analysis and Big O Notation:**

* If Algorithm A requires time proportional to ***f(n)***, we say that Algorithm A is said to be **order *f(n)***, which is denoted as ***O(f(n))***.
* The function *f(n)* is called the algorithm’s **growth-rate function**.
* Because the notation uses the capital letter O to denote order, it is called the **Big O notation**.
* If a problem of size *n* requires time that is directly proportional to n, the problem is *O(n)*—that is, order *n*. If the time requirement is directly proportional to *n2*, the problem is *O(n2)*, and so on.
* For various values of *n*, the approximate values of some common growth-rate functions, which are listed in order of growth:

*O(1) < O(log2 n) < O(n) < O(n \* log2 n) < O(n2) < O(n3) < O(2n) < O(n!)*



* The growth-rate functions have the following intuitive interpretations:

1. A growth-rate function of 1 implies a problem whose time requirement is **constant** and, therefore, independent of the problem’s size *n*.

*log2 n* The time requirement for a **logarithmic algorithm** increases slowly as the problem size increases. If you square the problem size, you only double its time requirement.

*n* The time requirement for a **linear algorithm** increases directly with the size of the problem. If you square the problem size, you also square its time requirement.

*n log2 n* The time requirement for an *n log2 n* algorithm increases more rapidly than a linear algorithm. Such algorithms usually divide a problem into smaller problems that are each solved separately.

*n2* The time requirement for a **quadratic algorithm** increases rapidly with the size of the problem. Algorithms that use two nested loops are often quadratic. Such algorithms are practical only for small problems.

*n3* The time requirement for a cubic algorithm increases more rapidly with the size of the problem that the time requirement for a quadratic algorithm. Algorithms that use three nested loops are often cubic, and are practical only for small problems.

*2n* As the size of a problem increases, the time requirement for an **exponential algorithm** usually increases too rapidly to be practical.

*n!* As the size of a problem increases, the time requirement for an **factorial algorithm** usually increases too rapidly to be practical.

* Several mathematical properties of Big O notation help to simplify the analysis of an algorithm.
  + You can ignore low-order terms in an algorithm’s growth-rate function. For example, if an algorithm if *O(n3 + 4n2 +3n + 99)*, it is also *O(n3)*.
  + You can ignore a multiplicative constant in the high-order term of an algorithm’s growth-rate function. For example, if an algorithm is *O(5n3)*, it is also *O(n3)*.
  + *O(f(n)) + O(g(n))* = *O(f(n) + g(n))*. You can combine growth-rate functions. For example, if an algorithm is *O(n2) + O(n)*, it is also *O(n2 + n)*, which you write simply as *O(n2)* be applying Property 1, above. Analogous rules hold for multiplication.
* A particular algorithm might require different times to solve different problems of the same size. For example, the time that an algorithm requires to search n items might depend on the nature of the items.
  + Usually, you consider the maximum amount of time that an algorithm can require to solve a problem of size *n*—that is, the worst case.
  + Although a worst-case analysis can produce a pessimistic time estimate, such an estimate does not mean that your algorithm will be slow. Instead, ou gave shown that the algorithm will never be slower than your estimate.

**Searching:**

* Sequential search of unordered set
* Binary search of ordered set