**Lesson 16 – Algorithm Efficiency II**

**Learning Objectives:**

* Analyze an algorithm using amortized runtime analysis.
* Describe a sorting algorithm and analyze its time complexity.

**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*

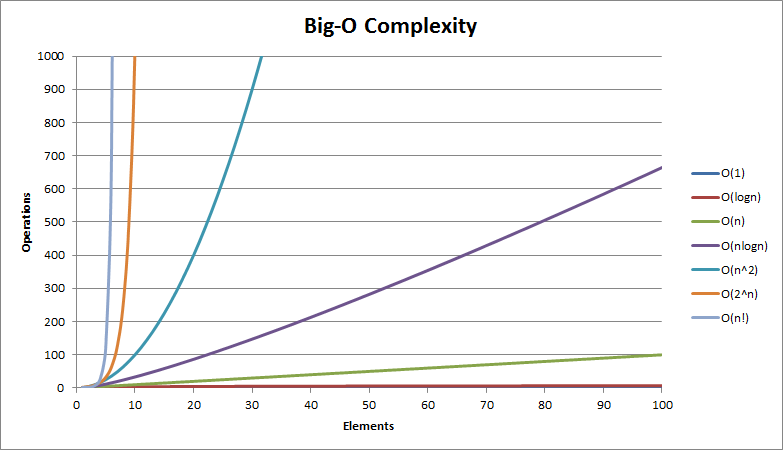
each express an algorithm’s proportional time requirement, or growth rate, and enable you to compare algorithm A with another algorithm B.

* 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.

**Amortized Runtime Analysis:**

* Consider our myArrayList implementation in Programming Assignment 03 using an array.

**public** **class** MyArrayList {

**private** **String**[] lstArray;

**private** **int** size;

**private** **void** increaseArraySize()

{

**String**[] newArray = **new** **String**[lstArray.length \* 2];

**for**(**int** i = 0; i < newArray.length; i++)

newArray[i] = lstArray[i];

lstArray = newArray;

}

**public** **void** add(**int** item)

{

**if** (this.size == lstArray.length)

increaseArraySize();

this.lstArray[this.size] = item;

size++;

}

* Let n be the length of lstArray. Adding a new element takes constant time, O(1), as long as the array is not full.
  + If the array is full, then the increaseArraySize() method is called which creates a new array twice as large (2n).
    - The new array is initialized to zero requiring 2n assignment operations.
    - The method then copies the contents of the old array into the new one in linear time, O(n).
  + Thus, the cost of increaseArraySize() is:

O(2n + n) = O(3n)

* + But, this call only occurs once every n add operations, so each call to add costs

[O(1) + O(1) + … + O(1) + O(3n)] / n = [O(n) + O(3n)] / n

= O(4n) / n

= O(n) / n

= O(1) amortized.

**Sorting:**

* Selection sort
* Bubble sort
* Insertion sort
* Mergesort
* Quicksort
* Radix sort

**Bubble sort:**

* The simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
* Iterate through the array and compare adjacent items.
* Swap them if the left item is larger than the right.
* Iterate through again excluding the last item and compare adjacent items.
* Swap them if the left item is larger than the right.
* Continue.

Example:

First Pass:

( **5** **1** 4 2 8 ) –> ( **1 5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 **5 4** 2 8 ) –> ( 1 **4 5** 2 8 ), Swap since 5 > 4

( 1 4 **5 2** 8 ) –> ( 1 4 **2 5** 8 ), Swap since 5 > 2

( 1 4 2 **5 8** ) –> ( 1 4 2 **5 8** ), Since (8 > 5), algorithm does not swap them.

Second Pass:

( **1 4** 2 5 8 ) –> ( **1 4** 2 5 8 )

( 1 **4 2** 5 8 ) –> ( 1 **2 4** 5 8 ), Swap since 4 > 2

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

( **1 2** 4 5 8 ) –> ( **1 2** 4 5 8 )

( 1 **2 4** 5 8 ) –> ( 1 **2 4** 5 8 )

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

void bubbleSort(int arr[])

    {

        int n = arr.length;

        for (int i = 0; i < n-1; i++)

            for (int j = 0; j < n-i-1; j++)

                if (arr[j] > arr[j+1])

                {

                    // swap arr[j+1] and arr[j]

                    int temp = arr[j];

                    arr[j] = arr[j+1];

                    arr[j+1] = temp;

                }

    }

**Selection sort:**

* Iterate through the array and choose the largest valued item.
* Put it in the last position.
* Iterate through the array (excluding the last position) and choose the largest value.
* Put it in the next to last position.
* Continue.

**Insertion sort:**

* Start at array index 1 (not 0).
* X1: If item at index 1 is smaller than item at index 0:
  + Swap them.
* Start at array index 2.
* X2: If item at index 2 is smaller than item at index 1:
  + Swap them.
  + Execute X1.
* Start at array index 3.
* If item at index 3 is smaller than item at index 2:
  + Swap them.
  + Execute X2.
* Continue.

**Mergesort:**

* Works by recursively splitting the array into two and sorting it as it is built back up.
* Base case:
  + Single element of array; return it.
* Recursive case:
  + Recursive call on left half of array (now sorted).
  + Recursive call on right half of array (now sorted).
  + Construct new array by iterating through each sorted half adding appropriate element to new array.
  + Return new sorted array.

**Quicksort:**

* Choose pivot (at random index or last index).
  + If random index, swap with last index.
* Set ‘left’ to first index that is larger than swap.
* Set ‘right’ to first index from right that is smaller than swap.
* Swap these two items.
* From that left and right position, continue scan and repeat swaps until left > right.
* Swap left and last.
* Recursive call on left array (0 to left – 1) and recursive call on right array (left + 1 to last).

|  |  |  |
| --- | --- | --- |
|  | **Worst case** | **Average case** |
| **Selection sort** | O(n2) | O(n2) |
| **Bubble sort** | O(n2) | O(n2) |
| **Insertion sort** | O(n2) | O(n2) |
| **Mergesort** | O(n log n) | O(n log n) |
| **Quicksort** | O(n2) | O(n log n) |
| **Radix sort** | O(m) | O(m) |

**Analysis:**

* Best is mergesort (if n < m) or radix sort (if m < n).
* Worst are selection sort, bubble sort, and insertion sort.
  + But all three are ‘good enough’ on small data sets.
* Radix sort is only good if the set of possible values is small.
  + Not good for double or String.
  + Good for char or small range of int (e.g., -100 to 100).
* In practice, quicksort is better than mergesort, because quicksort is an “in-place” sorting algorithm whereas mergesort has the overhead of method calls an extra memory requirements.