Data Structures and Algorithms

Section: Searching and Algorithm Analysis

Devere Anthony Weaver

Algorithm Efficiency

An algorithm's efficiency is typically measured by its **computational complexity**. Computational complexity is the amount of computational resources used by the algorithm, the most common being runtime and memory usage. Using computational complexity, we can compare algorithms to find the most optimal for our purposes and avoid algorithms with very long runtimes and/or large memory footprints. It is also useful to note that while runtime and memory are the biggest factors in most cases, other factors can be included such as network communication.

Runtime Complexity, best case, and worst case

Runtime complexity is a function $T(N)$ that represents the number of *constant time* operations performed by the algorithm on an input of size N. This is only a brief overview of the concept of runtime complexity, it is covered elsewhere in more depth and feel free to consult other texts on the subject such as CLRS and/or The Algorithm Design Manual.

The most common approach when evaluating an algorithm's runtime based on the input data is to measure the best and worst case scenarios for its implementation. The **best case** scenario is where the algorithm does the minimum possible number of operations and the **worst case** scenario is where the algorithm does the maximum possible number of operations.

As an example of an algorithm we will go into later, we have linear search. This algorithm searches through an array of elements until finding the key. For linearly searching through a vector or array, the best case scenario is the least amount of operation, therefore if the key is the very first element, that is the best case scenario. In contrast, the worst case scenario is if the key doesn't exist in the data structure as each element would have to be traversed.

Time Complexity

An algorithm's **space complexity** is a function $S(N)$ that represents the number of fixed-sized memory units used by the algorithm for an input of size N. This function includes the input data itself as well as any memory that is allocated by the algorithm. The **auxiliary space complexity** is the space complexity of an algorithm not including the input data (i.e. just the data that is allocated during algorithm execution). Auxiliary space complexity only includes non-input data that does not increase for larger inputs.

Algorithms

An algorithm is a sequence of steps for accomplishing a task. In this section we'll begin to look at basic searching algorithms, starting with **linear search**. Linear search is a searching algorithm that starts from the beginning of a list (or similar structure) and checks each element until the search key is found or the end of the list is reached.

Algorithm Runtime

An algorithm's **runtime** is the time the algorithm takes to execute. Using linear search as an example, we say that the algorithm is said to require "on the order" of N comparisons. This is the worst case scenario should the key not be found in the list.

Note: This section only touched on algorithm runtime, so seek out other resources in order to get a more in-depth analysis of algorithm runtime.

Binary Search

Binary search can allow for faster runtimes if the collection of information we are working with is sorted. It continuously bisections the data structure until either the desired result is found or if it is determined that the key doesn't exist in the data structure.

Binary Search Efficiency

This algorithm is very efficient in finding an element within a sorted list. For an N element list, the maximum number of steps required to reduce the search space to an empty sublist is given by

$$\lfloor \log_2 N \rfloor + 1.$$

As an example, given a list with 32 elements, we can compute the maximum number of steps to be,

$$\lfloor \log_2 32 \rfloor + 1 = 6.$$

Observe, this is the maximum number of *steps* for the algorithm, NOT the number of distinct list elements that are compared to they key.

Constant Time Operations

Constant Time Operations

A **constant time operation** is an operation that, for a given processor, always operates in the same amount of time, regardless of input values. This is more useful in *theoretical* analysis of algorithms since in actual implementation, an algorithm can have different runtimes on different processors. Hence, it is more helpful to describe our algorithmic runtime in terms of number of constant time operations and not in seconds, nanoseconds, etc.

An example of a constant time operation is simply binding an integer value to a name. Another example of a constant time operation is the multiplication of fixed-sized integers. An example of a non-constant time operation would be a loop that performs addition each time. The loop iterates X number of times and adding Y to some sum. This is not a constant time operation since the time is dependent on the size of X for the number of iterations. String concatenation is also not a constant time operation since more characters must be copied for larger strings.

Note: a loop is not necessarily a non-constant time operation, it is possible that a loop using a fixed number of iterations, in which case it would be a constant-time operation.

Generally, the following are considered constant time operations,

- addition, subtraction, multiplication, and division of fixed size ints or floats
- assignment of a reference, pointer, or other fixed size data value
- comparison of two fixed sized data values

- read or write an array element at a particular index

Growth of Functions and Complexity

Upper and Lower Bounds

Recall, $T(N)$ is the function that determines runtime complexity for a given algorithm. This function is bounded. The **lower bound** of $T(N)$ is a function $f(N) \ni f(N) \leq$ the best case $T(N), \forall N \geq 1$. The **upper bound** of $T(N)$ is a function $f(N) \ni f(N) \geq$ the worst case $T(N), \forall N \geq 1$.

Observe that we are talking about *a* bound. This means that there exists more than one upper and lower bound. However, whichever bounds we use must enclose all possible runtimes for the algorithm. It may also be useful to note that the supremenum and infinum of $T(N)$ are the worst case and best case runtimes for the algorithm.

Growth Rates and Asymptotic Notations

When describing runtime complexity, we can use **asymptotic notation** in which we classify the runtime complexity using functions that indicate the growth rate of a bounding function. This eliminates the need to include the constant term when describing the growth rate of a function.

The three most commonly used notations are,

- O notation - provides growth rate for an algorithm's upper bound
- Ω notation - provides growth rate for an algorithm's lower bound
- Θ notation - provides a growth rate that is both an upper and lower bound

---

O Notation

Big O Notation

Big O Notation is a way of describing how a function (runtime) generally behaves in relation to its input size. Using this notation, all functions that have the same growth rate (determined by the highest order term of the function) are characterized by using the same Big O notation. Thus all functions with the same growth rate are considered equivalent in Big O notation.

To easily determine the Big O notation for a given function,

- If $f(N)$ is a sum of several terms, keep the highest order term as this is the fastest growth rate
- If $f(N)$ has a term that is the product of several factors, all constants are omitted

Composite functions appear to behave in the usual manner.

Runtime Growth Rate

When it comes to evaluating algorithms, the efficiency of the algorithm is most critical for large inputs. The difference in computation for different functions with small inputs may not be very different since small inputs likely result is fast runtimes. However, larger inputs can result in drastic changes in runtime for different functions.

Algorithm Analysis

Worse-case Algorithm Analysis

To analyze how runtime scales with input, first determine the number of operations for a specific input size N. Then, the big-O notation is determined. The most common runtime analysis is for the worst-case runtime although one can use the best-case runtime or even the average-case runtime which requires knowing statistical properties of expected data inputs.

Runtime analysis determines the total number of operations which include operations such as assignment, addition, comparisons, etc.

NOTE: Practice is needed when attempting to determine using actual code snippets.

Counting constant time operations

Counting the precise number of constant time operations isn't really needed when using big-O notation.

Runtime analysis of nested loops

Runtime analysis of nested loops requires summing the runtime of the inner loop over each outer loop iteration.

This completes the chapter on searching and algorithm analysis. It gave a good introduction to the topic; however, I will consult other, more rigorous texts to better understand the mathematical aspects of algorithm analysis.