Searching and Algorithm Analysis

Devere Anthony Weaver

---

1 Algorithm Efficiency

Algorithm efficiency is measured by an algorithm's computational complexity. **Computational complexity** is the amount of resources used by the algorithm, which are typically runtime and memory usage of the implementation. **Complexity analysis** is used to identify and avoid using algorithms with long runtimes or high memory usage. One can then use the analysis results to compare the efficiency of algorithms.

1.1 Runtime complexity, best case, and worst case

An algorithm's **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.

Recall, an operation is **constant time** if the amount of time it takes to carry out said operation does not depend on the size of the input.

Best case is when the algorithm does the minimum possible number of operations. The **worse case** is when the algorithm does the maximum possible number of operations.

1.2 Space complexity

Space complexity An algorithm's **space complexity** is a function $S(N)$ the represents the number of fixed-size memory units used by the algorithm for an input size of N.

E.g. Let N be the size of a list of numbers. Then $S(N) = 2N + k$ is the function for the space complexity of an algorithm that simply duplicates a list of numbers. We multiply N by 2 since we are doubling the size of the input and k is a constant representing memory used for other programming structures like loop counters, pointers, etc. needed to actually implement the algorithm.

An algorithm's **auxiliary space complexity** is the space complexity not including the input data. These include things such as variables and other memory allocated for the actual implementation of the algorithm.

---

2 Searching and Algorithms

2.1 Algorithms

Linear search is a search algorithm that starts from the beginning of a list and checks each element until the search key is found or the end of the list is reached.

2.2 Algorithm Runtime

An algorithm's **runtime** is the time the algorithm takes to execute.

3 Binary Search

This is just bisection from numerical analysis, we can just use it for non-numeric types as well. Also, the data must be in a sorted stated. The return value in this case is the index of the key.

3.1 Binary Search Efficiency

Given an N element list, the maximum number of steps required to reduce the search space to an empty sublist is given by,

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

4 Constant Time Operations

4.1 Constant Time Operations

A single algorithm can always execute more quickly on a faster processor, hence analysis of an algorithm will describe runtime in terms of number of 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.

Some examples of constant time operations include,

- assignment to a fixed-size integer
- multiplication of fixed-size integers
- a loop with a fixed number of iterations

Some examples of non-constant time operation include,

- looping an X number of times and summing values in each loop
- string concatenation

One hint that an operation is not constant time is that larger operands take longer to execute or are variable. Another thing to note is that we treat constant time operations as mathematical constants, meaning that multiple constant time operations can be considered as one constant, despite being more than one actual operation.

4.2 Identifying Constant Time Operations

What is considered constant time operations can differ based on factors such as the implementation language and hardware. Some common constant time operations are,

- addition, multiplication, subtraction, division of *fixed-size* integer or floating-point values
- assignment of a reference, pointer, or other *fixed-size* data value
- comparison of two *fixed-size* data values
- read or write an array element of a particular index

5 Growth of Functions

5.1 Upper and Lower Bounds

Let $T(N)$ be a function for the runtime complexity of an algorithm. Similar to mathematical analysis, we say that a

- **Lower Bound** is a function $f(N)$ that is less than or equal the *best case* of $T(N), \forall N \geq 1$
- **Upper Bound** is a function $f(N)$ that is greater than or equal to the *worst case* of $T(N), \forall N \geq 1$

Observe, we are talking about general least/upper bounds. This does not refer to the least upper bound/greatest lower bound. Together both the upper and lower bounds enclose all possible runtimes for this algorithm.

5.2 Growth Rates and Asymptotic Notations

Asymptotic notation is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function. The following are useful notations in describing complexity analysis,

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

---

6 O Notation

6.1 Big O Notation

Big O Notation is mathematical way of describing how a function (runtime of an algorithm) behaves in relation to the input size. Using this notation, all functions that have the same growth rate are characterised using the same Big O notation. We consider these functions as equivalent.

As a general rule, given a function that describes the runtime of an algorithm, to determine Big-O,

- If $f(N)$ is a sum of several terms, the highest order term is the fastest growth rate and thus can be used to describe the growth rate
- If $f(N)$ is a product of several factors, all constants are omitted

6.2 Big O Notation of Composite Functions

In general, composite functions for Big O notation follow the same rules as other mathematical functions.

6.3 Runtime Growth Rate

Algorithm runtime is most important for very large inputs.

---

7 Algorithm Analysis

7.1 Worse-case Algorithm Analysis

To understand how an algorithm scales with input size, we need to determine how many operations an algorithm executes for a specific input size N. Then, Big-O notation for the function is determined. **Worse-case runtime** of an algorithm is the runtime complexity for an input that results in the longest execution. Runtime analysis also includes **base-case** and **average-case** runtime.

7.2 Counting Constant Time Operations

Observe that for algorithm analysis, an operation is any statement that has a constant runtime complexity, denoted $O(1)$. This means that there isn't a need to precisely count all of the constant time operations when using Big O notation for the given algorithm as all constants will remain constants where they can be denoted $O(1)$.

7.3 Runtime Analysis of Nested Loops

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

Note: I need practice counting these.