Computational Complexity Basics

1 Overview

The basic idea behind computational complexity isn't too complicated: For a given algorithm and input(s) of interest, we should be able to categorize the relationship between the *resources* used by the algorithm and the input(s) to the algorithm.

Usually, the resources of interest are time and memory. Usually, we'd like to minimize the resources used by an algorithm. Search algorithms, for example, are focused on minimizing the time it takes to find a specific element in a collection of values (which usually involves sorting the elements in some fashion).

1.1 Semi-Formal Definition

To specify the computational complexity of an algorithm, we use what is called Big-O notation. Notationally, it looks something like $\mathcal{O}(f(n))$, where n is an input to the algorithm (f can also be function of multiple inputs). It's a bit difficult to define complexity in terms of Big-O and related ideas while remaining concise. In semi-simple terms, Big-O notation tells us something about the *upper-bound* of the resources used by an algorithm for values of the input(s) above some defined value of n that we'll refer to as n_0. Usually, we can just think of the worst-case scenario that an algorithm could take on and define its worse-case complexity in terms of that. This is where the $f(n)$ comes in: It's function that, up to some multiplicative constant, is larger than the trend that our resource-use follows as n increases (for all $n > n_0$).[1]

2 Examples

2.1 Array Search

For example, if we had an array of integers of length n and we wanted to search for a value a in the array, we could just look at every value of the

[1] See the Wikipedia page on Big-O notation for more on this.

array in order (starting at the first value) and comparing it to a until we find the value we're searching for. In the worst case, a is the last element of the array and we have to compare all n values of the array to a until we find it at the very end. We can assume that the operation of comparing two integers takes about the same amount of time in all instances; this means that the complexity of the comparison of two integers is *constant* in time. In Big-O notation, a constant complexity is written as $\mathcal{O}(1)$. As mentioned, we do this comparison n times in the worst case. So the worse-case complexity of a single search through our array as $\mathcal{O}(n * 1) = \mathcal{O}(n)$.

2.2 Matrix

Let's consider more complicated algorithm. Suppose we have a program that creates an $n \times n$ matrix of random integers, then goes through and calculates the number of odd values in each row. We'll start by figuring out the complexity of creating the matrix itself. We can assume that generating a random integer takes constant time, as does adding it to the matrix ($\mathcal{O}(1)$). The matrix has a total of n^2 elements to fill, so we perform this constant operation n^2 times in all cases, giving a complexity of $\mathcal{O}(n^2)$. Determining whether an integer is odd is a constant-time operation. This would be a constant-time operation performed n times for each of the n rows, which (again) is of complexity $\mathcal{O}(n^2)$.

Creating the matrix of random integers and calculating the number of odds in each row are distinct steps in this process. Since we can characterize both in terms of n, we'll add the two complexities together to get an overall complexity of $\mathcal{O}(n^2 + n^2) = \mathcal{O}(2n^2) = \mathcal{O}(n^2)$. Notice that we end up dropping the constant factor of 2. This is what is meant by the final statement in Section **??**: If we know that the upper-bound of the complexity follows $\alpha f(n)$, we can drop the constant α since we are only interested in the trend followed by algorithm independent of any multiplicative constants.

Also note that if we had an algorithm of complexity $\mathcal{O}(3n^2 + n + 5)$, we would drop off the lower-order terms and be left with $\mathcal{O}(n^2)$. This is because, for large values of n, n^2 dominates any function of the form $\alpha n^2 + \beta n + \gamma$ for constants α, β, and γ.

2.3 Traveling Salesman Problem (TSP)

2.3.1 Implementation Description

Let's consider the exhaustive implementation of the Traveling Salesman Problem in the Python file I sent. Our goal is to find the shortest path

through a given set of towns.

Every path through a set of n towns can be found by finding every permutation of a list of the n towns. There are $n!$ such permutations of n towns, and Python's **itertools.permutations** function indeed takes $n!$ time to generate this list for us. Once we find all of the permutations of towns, we also have to figure out the total distance traveled through each permutation/path. To do this, we sum the distances between each pair of towns in each of the $n!$ paths. Each path ends up requiring n constant-time additions, which means we have $n*n!$ additions to get the distances traveled through every possible path. So, the complexity of generating a list of all possible paths and another list of all of the path-lengths is $\mathcal{O}(n*n!)$.

Once all of the path-distances have been traveled, it's simply a matter of finding the path with the smallest distance to travel. For this, Python's **min** function is used, which takes a list of values and finds the smallest one. The complexity of the built-in **min** function depends on the number of elements of the list in question. We have to find the minimum value of a list of length $n!$, so the complexity of this operation is $\mathcal{O}(n!)$.

2.3.2 Complexity

The overall complexity of this exhaustive algorithm is the sum of the complexities discussed. There are certainly other calculations made throughout the program, but their complexities would not be significant enough to outweigh the $n!$ terms. So our overall complexity is $\mathcal{O}((n*n!)+n!) = \mathcal{O}(n!(n+1)) = \mathcal{O}(n*n!)$.

As demonstrated by the complexity of this exhaustive method, the TSP is a hard problem to solve quickly. It's part of a class of problems known as NP-complete problems, which are problems where solutions are easy to verify but the very same solutions are not easy to find in the first place. Any polynomial-complexity solution would be consider efficient for such a problem (i.e. an algorithm with complexity of the form $\mathcal{O}(c_a n^a + c_{a-1} n^{a-1} + \ldots + c_1 n + c_0) = \mathcal{O}(n^a)$ for constants a and $c_a, c_{a-1} \ldots c_0$. $\mathcal{O}(n! * n)$ does not match this form). It's generally believed that no efficient soluion exists for NP-complete problems.

2.3.3 Concluding Thoughts

For many of these difficult problems, an algorithm that finds a *pretty good* solution *most of the time* is often considered sufficient. The other TSP implementation in the Python file is called the 'nearest neighbor' method. This

method is not guaranteed to find the optimal solution like the exhaustive method is; it simply starts at a one of the towns, finds the closest available town to travel to and goes to it, and repeats this process until it reaches the final town. There are many cases where this will not provide a very good solution, but it is much faster than the exhaustive method. This illustrates the trade-off that must often be made between computational resources and accuracy.