<https://www.topcoder.com/community/data-science/data-science-tutorials/the-importance-of-algorithms/>

Runtime Analysis

One of the most important aspects of an algorithm is how fast it is. It is often easy to come up with an algorithm to solve a problem, but if the algorithm is too slow, it’s back to the drawing board. Since the exact speed of an algorithm depends on where the algorithm is run, as well as the exact details of its implementation, computer scientists typically talk about the runtime relative to the size of the input. For example, if the input consists of N integers, an algorithm might have a runtime proportional to N2, represented as O(N2). This means that if you were to run an implementation of the algorithm on your computer with an input of size N, it would take C\*N2 seconds, where C is some constant that doesn’t change with the size of the input.

However, the execution time of many complex algorithms can vary due to factors other than the size of the input. For example, a sorting algorithm may run much faster when given a set of integers that are already sorted than it would when given the same set of integers in a random order. As a result, you often hear people talk about the worst-case runtime, or the average-case runtime. The worst-case runtime is how long it would take for the algorithm to run if it were given the most insidious of all possible inputs. The average-case runtime is the average of how long it would take the algorithm to run if it were given all possible inputs. Of the two, the worst-case is often easier to reason about, and therefore is more frequently used as a benchmark for a given algorithm. The process of determining the worst-case and average-case runtimes for a given algorithm can be tricky, since it is usually impossible to run an algorithm on all possible inputs. There are many good online resources that can help you in estimating these values.

Approximate completion time for algorithms, N = 100

|  |  |
| --- | --- |
| O(Log(N)) | 10-7 seconds |
| O(N) | 10-6 seconds |
| O(N\*Log(N)) | 10-5 seconds |
| O(N2) | 10-4 seconds |
| O(N6) | 3 minutes |
| O(2N) | 1014 years |
| O(N!) | 10142 years |

Sorting

Sorting provides a good example of an algorithm that is very frequently used by computer scientists. The simplest way to sort a group of items is to start by removing the smallest item from the group, and put it first. Then remove the next smallest, and put it next and so on. Unfortunately, this algorithm is O(N2), meaning that the amount of time it takes is proportional to the number of items squared. If you had to sort a billion things, this algorithm would take around 1018 operations. To put this in perspective, a desktop PC can do a little bit over 109 operations per second, and would take years to finish sorting a billion things this way.

Luckily, there are a number of better algorithms (quicksort, heapsort and mergesort, for example) that have been devised over the years, many of which have a runtime of O(N \* Log(N)). This brings the number of operations required to sort a billion items down to a reasonable number that even a cheap desktop could perform. Instead of a billion squared operations (1018) these algorithms require only about 10 billion operations (1010), a factor of 100 million faster.

## Shortest Path

Algorithms for finding the shortest path from one point to another have been researched for years. Applications abound, but lets keep things simple by saying we want to find the shortest path from point A to point B in a city with just a few streets and intersections. There are quite a few different algorithms that have been developed to solve such problems, all with different benefits and drawbacks. Before we delve into them though, lets consider how long a naive algorithm – one that tries every conceivable option – would take to run. If the algorithm considered every possible path from A to B (that didn’t go in circles), it would not finish in our lifetimes, even if A and B were both in a small town. The runtime of this algorithm is exponential in the size of the input, meaning that it is O(CN) for some C. Even for small values of C, CN becomes astronomical when N gets even moderately large.

One of the fastest algorithms for solving this problem has a runtime of O(EVLog(V)), where E is the number of road segments, and V is the number of intersections. To put this in perspective, the algorithm would take about 2 seconds to find the shortest path in a city with 10,000 intersections, and 20,000 road segments (there are usually about 2 road segments per intersection). The algorithm, known as Djikstra’s Algorithm, is fairly complex, and requires the use of a data structure known as a priority queue. In some applications, however, even this runtime is too slow (consider finding the shortest path from New York City to San Francisco – there are millions of intersections in the US), and programmers try to do better by using what are known as heuristics. A heuristic is an approximation of something that is relevant to the problem, and is often computed by an algorithm of its own. In the shortest path problem, for example, it is useful to know approximately how far a point is from the destination. Knowing this allows for the development of faster algorithms (such as A\*, an algorithm that can sometimes run significantly faster than Djikstra’s algorithm) and so programmers come up with heuristics to approximate this value. Doing so does not always improve the runtime of the algorithm in the worst case, but it does make the algorithm faster in most real-world applications.