Why Learn Data Structures and Algorithms?

In this article, we will learn why every programmer should learn data structures and algorithms with the help of examples.

This article is for those who have just started learning algorithms and wondered how impactful it will be to boost their career/programming skills. It is also for those who wonder why big companies like Google, Facebook, and Amazon hire programmers who are exceptionally good at optimizing Algorithms.

What are Algorithms?

Informally, an algorithm is nothing but a mention of steps to solve a problem. They are essentially a solution.

For example, an algorithm to solve the problem of factorials might look something like this:

Problem: Find the factorial of `n`

```
Initialize fact = 1
For every value v in range 1 to n:
    Multiply the fact by v
fact contains the factorial of n
```

Here, the algorithm is written in English. If it was written in a programming language, we would call it to **code** instead. Here is a code for finding the factorial of a number in C++.

```
int factorial(int n) {
    int fact = 1;
    for (int v = 1; v <= n; v++) {
        fact = fact * v;
    }
    return fact;
}
```

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are used to solve the problem using that data.

Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use each one of them. It also teaches you the science of evaluating the efficiency of an algorithm. This enables you to choose the best of various choices.

Use of Data Structures and Algorithms to Make Your Code Scalable

Time is precious.

Suppose, Alice and Bob are trying to solve a simple problem of finding the sum of the first 10^{11} natural numbers. While Bob was writing the algorithm, Alice implemented it proving that it is as simple as criticizing Donald Trump.

Algorithm (by Bob)

```
Initialize sum = 0
for every natural number n in range 1 to 10^11(inclusive):
    add n to sum
sum is your answer
```

Code (by Alice)

```
int findSum() {
    int sum = 0;
    for (int v = 1; v <= 100000000000; v++) {
        sum += v;
    }
    return sum;
}
```

Alice and Bob are feeling euphoric of themselves that they could build something of their own in almost no time. Let's sneak into their workspace and listen to their conversation.

```
Alice: Let's run this code and find out the sum.
Bob: I ran this code a few minutes back but it's still not showing the output. What's wrong wi
```

Oops, something went wrong! A computer is the most deterministic machine. Going back and trying to run it again won't help. So let's analyze what's wrong with this simple code.

Two of the most valuable resources for a computer program are **time** and **memory**.

The time taken by the computer to run code is:

```
Time to run code = number of instructions * time to execute each instruction
```

The number of instructions depends on the code you used, and the time taken to execute each code depends on your machine and compiler.

In this case, the total number of instructions executed (let's say x) are `x = 1 + (10^11 + 1) + (10^11) + 1`, which is `x = 2 * 10^11 + 3`

Let us assume that a computer can execute `y = 10^8` instructions in one second (it can vary subject to machine configuration). The time taken to run above code is

```
Time to run y instructions = 1 second
Time to run 1 instruction = 1 / y seconds
Time to run x instructions = x * (1/y) seconds = x / y seconds

Hence,
Time to run the code = x / y
                     = (2 * 10^11 + 3) / 10^8 (greater than 33 minutes)
```

Is it possible to optimize the algorithm so that Alice and Bob do not have to wait for 33 minutes every time they run this code?

I am sure that you already guessed the right method. The sum of first `N` natural numbers is given by the formula:

```
Sum = N * (N + 1) / 2
```

Converting it into code will look something like this:

```
int sum(int N) {
    return N * (N + 1) / 2;
}
```

This code executes in just one instruction and gets the task done no matter what the value is. Let it be greater than the total number of atoms in the universe. It will find the result in no time.

The time taken to solve the problem, in this case, is `1/y` (which is 10 nanoseconds). By the way, the fusion reaction of a hydrogen bomb takes 40-50 ns, which means your program will complete successfully even if someone throws a hydrogen bomb on your computer at the same time you ran your code. :)

> **Note:** Computers take a few instructions (not 1) to compute multiplication and division. I have said 1 just for the sake of simplicity.

More on Scalability

Scalability is scale plus ability, which means the quality of an algorithm/system to handle the problem of larger size.

Consider the problem of setting up a classroom of 50 students. One of the simplest solutions is to book a room, get a blackboard, a few chalks, and the problem is solved.

But what if the size of the problem increases? What if the number of students increased to 200?

The solution still holds but it needs more resources. In this case, you will probably need a much larger room (probably a theater), a projector screen and a digital pen.

What if the number of students increased to 1000?

The solution fails or uses a lot of resources when the size of the problem increases. This means, your solution wasn't scalable.

What is a scalable solution then?

Consider a site like Khanacademy, millions of students can see videos, read answers at the same time and no more resources are required. So, the solution can solve the problems of larger size under resource crunch.

If you see our first solution to find the sum of first `N` natural numbers, it wasn't scalable. It's because it required linear growth in time with the linear growth in the size of the problem. Such algorithms are also known as linearly scalable algorithms.

Our second solution was very scalable and didn't require the use of any more time to solve a problem of larger size. These are known as constant-time algorithms.

Memory is expensive

Memory is not always available in abundance. While dealing with code/system which requires you to store or produce a lot of data, it is critical for your algorithm to save the usage of memory wherever possible. For example: While storing data about `people`, you can save memory by storing only their date of birth, not their age. You can always calculate it on the fly using their date of birth and current date.

Examples of an Algorithm's Efficiency

Here are some examples of what learning algorithms and data structures enable you to do:

Example 1: Age Group Problem

Problems like finding the people of a certain age group can easily be solved with a little modified version of the binary search algorithm (assuming that the data is sorted).

The naive algorithm which goes through all the persons one by one, and checks if it falls in the given age group is linearly scalable. Whereas, binary search claims itself to be a logarithmically scalable algorithm. This means that if the size of the problem is squared, the time taken to solve it is only doubled.

Suppose, it takes 1 second to find all the people at a certain age for a group of 1000. Then for a group of 1 million people,

- the binary search algorithm will take only 2 seconds to solve the problem
- the naive algorithm might take 1 million seconds, which is around 12 days

The same binary search algorithm is used to find the square root of a number.

Example 2: Rubik's Cube Problem

Imagine you are writing a program to find the solution of a Rubik's cube.

This cute looking puzzle has annoyingly 43,252,003,274,489,856,000 positions, and these are just positions! Imagine the number of paths one can take to reach the wrong positions.

Fortunately, the way to solve this problem can be represented by the graph data structure. There is a graph algorithm known as Dijkstra's algorithm which allows you to solve this problem in linear time. Yes, you heard it right. It means that it allows you to reach the solved position in a minimum number of states.

Example 3: DNA Problem

DNA is a molecule that carries genetic information. They are made up of smaller units which are represented by Roman characters A, C, T, and G.

Imagine yourself working in the field of bioinformatics. You are assigned the work of finding out the occurrence of a particular pattern in a DNA strand.

It is a famous problem in computer science academia. And, the simplest algorithm takes the time proportional to

```
(number of character in DNA strand) * (number of characters in pattern)
```

A typical DNA strand has millions of such units. Eh! worry not. **KMP algorithm** can get this done in time which is proportional to

```
(number of character in DNA strand) + (number of characters in pattern)
```

The `*` operator replaced by `+` makes a lot of change.

Considering that the pattern was of 100 characters, your algorithm is now 100 times faster. If your pattern was of 1000 characters, the KMP algorithm would be almost 1000 times faster. That is, if you were able to find the occurrence of pattern in 1 second, it will now take you just 1 ms. We can also put this in another way. Instead of matching 1 strand, you can match 1000 strands of similar length at the same time.

And there are infinite such stories...

Final Words

Generally, software development involves learning new technologies on a daily basis. You get to learn most of these technologies while using them in one of your projects. However, it is not the case with algorithms.

If you don't know algorithms well, you won't be able to identify if you can optimize the code you are writing right now. You are expected to know them in advance and apply them wherever possible and critical.

We specifically talked about the scalability of algorithms. A software system consists of many such algorithms. Optimizing any one of them leads to a better system.

However, it's important to note that this is not the only way to make a system scalable. For example, a technique known as distributed computing allows independent parts of a program to run to multiple machines together making it even more scalable.