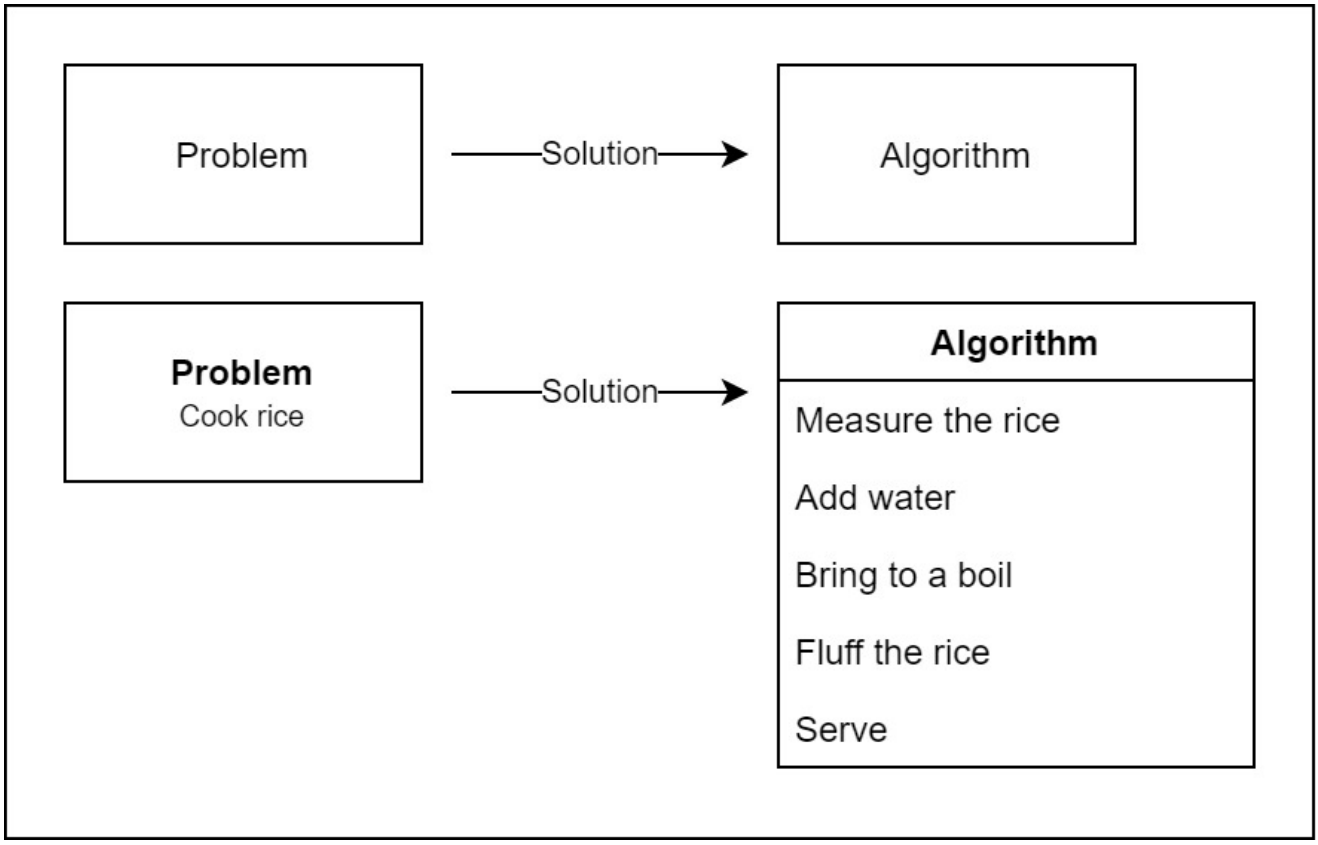
What are Algorithms?



So, what exactly are algorithms? Informally, an algorithm is just a ***set of steps used to solve a problem***. In other words, it's a solution to a specific task. Let's take an example to make it clear. Consider finding the factorial of a number. We can describe an algorithm like this:

Problem: Find the factorial of n

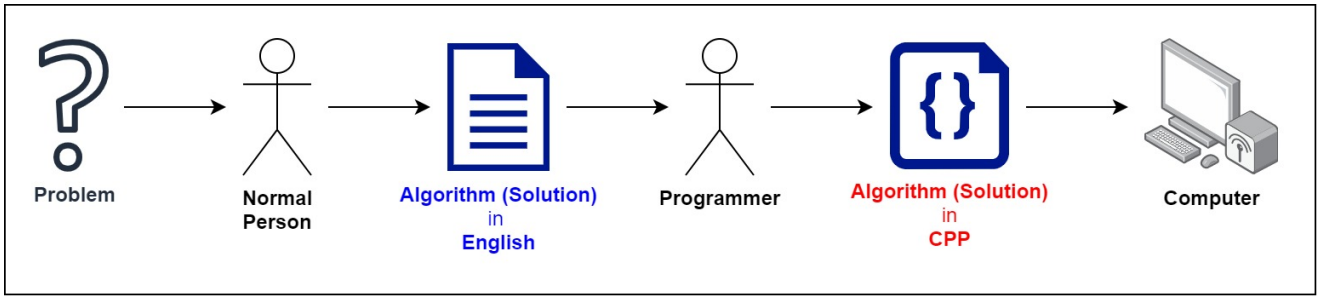
```
Algorithm: Find the factorial of n

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

Now, this algorithm is described in plain English. If we were to write it in a programming language, we'd call it ***code*** instead. Here's the code for finding the factorial of a number in C++:

factorial.cpp

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

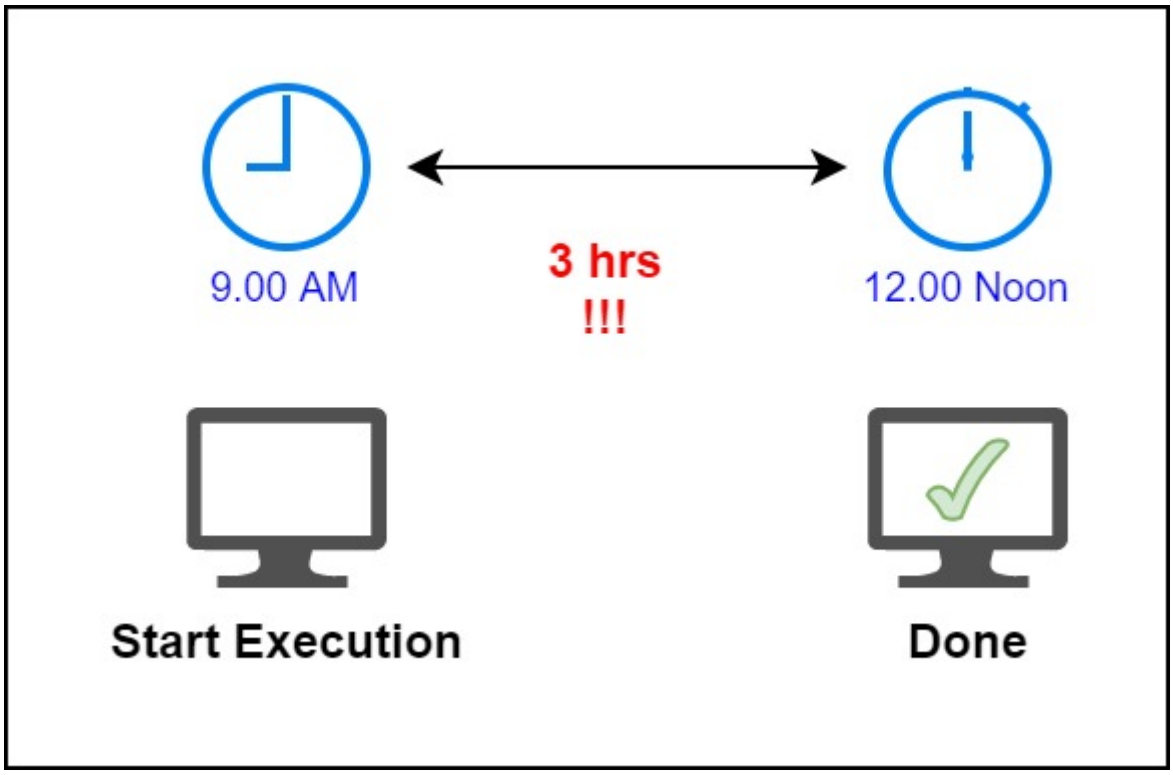


Use of Data Structures and Algorithms to Make Your Code Scalable

In the world of programming, data structures hold data, and algorithms are used to manipulate that data to solve problems.

You might be wondering why top companies place such a high emphasis on hiring programmers who excel at optimizing algorithms. Well, that's because data structures and algorithms play a crucial role in making your code efficient and scalable.

Time is precious



Let's take a scenario to understand this better. Meet Alice and Bob, two programmers trying to find the sum of the first 10^{11} natural numbers. Bob comes up with this algorithm:

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
```

On the other hand, Alice implements the following code:

Code (by Alice)

Code (by Alice)

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

Both Alice and Bob are excited about their solutions, but when they run the code, they encounter a problem. Bob's code is taking a significant amount of time to execute, and Alice wonders what's wrong.

The time taken by a computer to run code depends on the number of instructions executed and the time to execute each instruction. In this case, the total number of instructions executed (let's call it x) is $x = 2\ 10^{11} + 3$.

findSum.cpp

```cpp
int findSum() {
    int sum = 0; /* 1 time */
    for (
      int v = 1; /* 1 time */
      v <= 100000000000;
      v++ /* 10^11 times */
   ) {
        sum += v; /* 10^11 times */
    }
    return sum; /* 1 time */
}

/*
  No. of instructions will be executed (x)
    = 1 + 1 + 10^11 + 10^11 + 1
    = 2 * 10^11 + 3
*/
```

Now, assuming a computer can execute y (= 108) instructions in one second, the time taken to run the code will be:

findSum

```
Time to run the code
  = No. of Executions / No. of Executions in One Second
  = x / y
  = (2 * 10^11 + 3) / 108 (greater than 33 minutes)
```

As you can see, Bob's algorithm is not optimized, and it's taking more than 33 minutes to run. But we can do better! By using the formula for the sum of the first N natural numbers, which is `Sum = N * (N + 1) / 2`, we can create a much more efficient algorithm:

sum.cpp

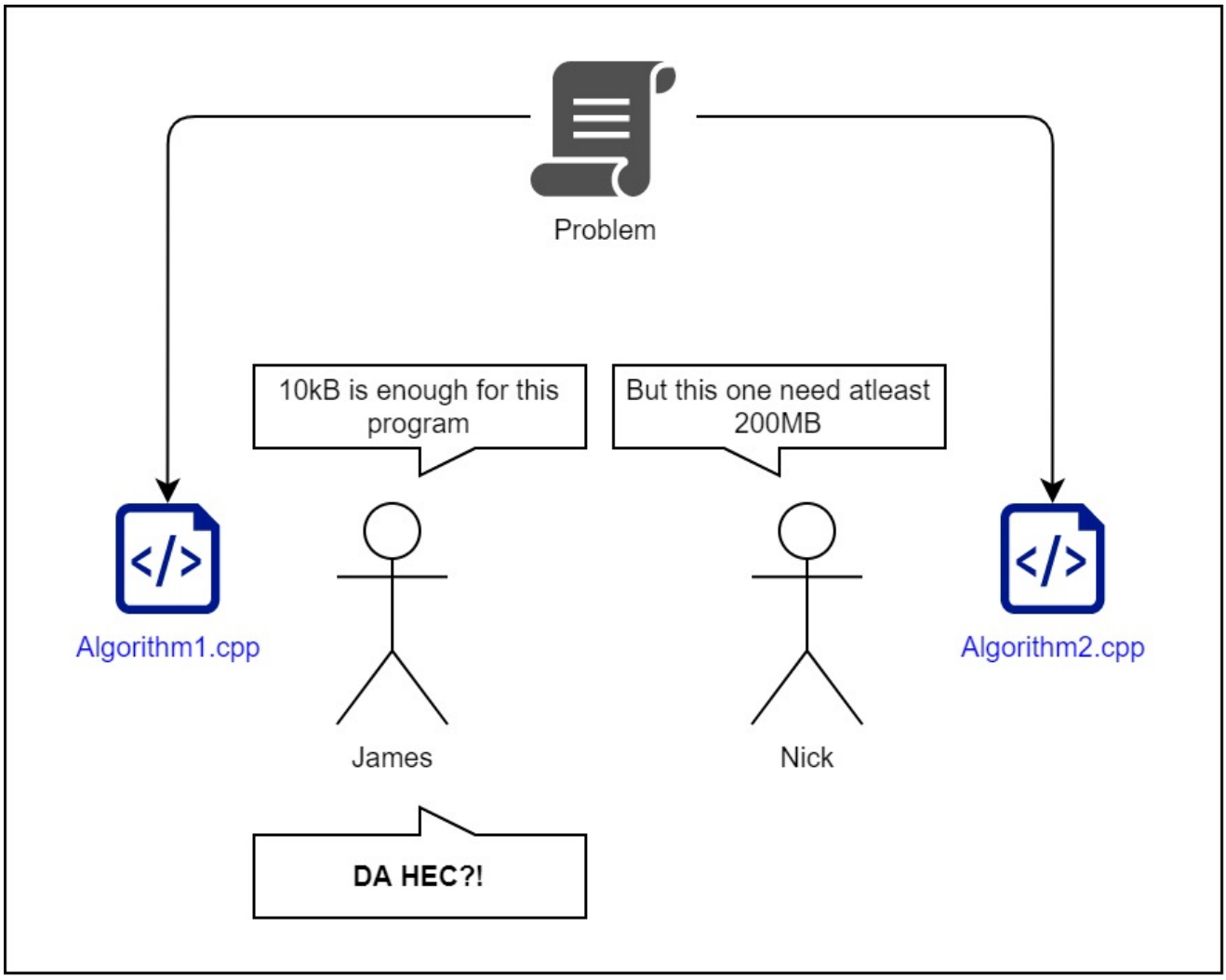
```cpp
int sum(int N) {
    return N * (N + 1) / 2; /* 1 time */
}
```

This code executes in just one instruction, no matter how large the value of N is. It's what we call a constant-time algorithm. Using this optimized algorithm, the time taken to find the sum is `1/y`, which is just 10 nanoseconds!

sum.cpp

```
Time to run the code
  = No. of Executions / No. of Executions in One Second
  = x / y
  = 1 / 108
```

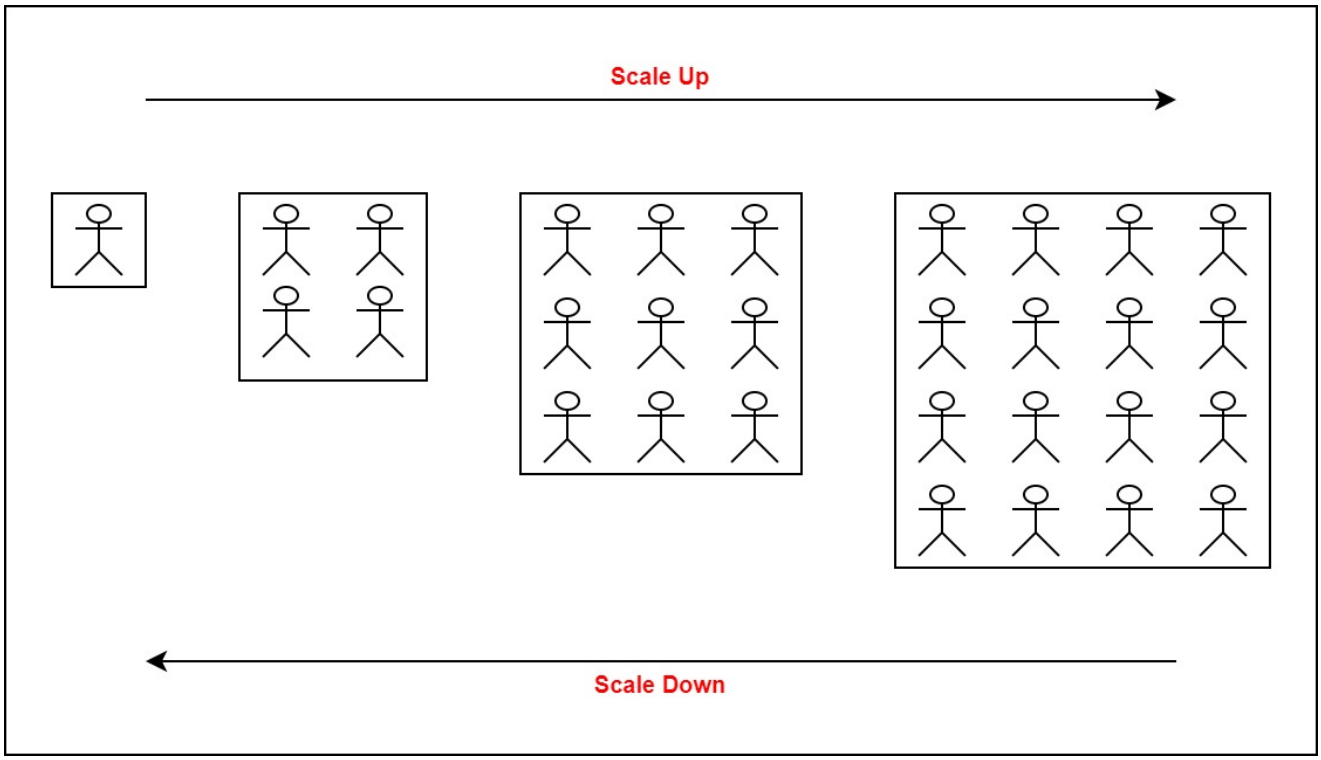
As you can see, choosing the right data structures and algorithms can significantly impact the performance of your code, making it scalable and efficient. ***Scalability*** refers to the ability of an algorithm or system to handle larger problem sizes without requiring a significant increase in resources or time.



Memory is another important factor to consider. While dealing with code that requires storing or processing a large amount of data, it's crucial to optimize memory usage. For instance, you can store just the date of birth of people instead of their age, calculating their age on-the-fly when needed.

In this course, we'll explore various data structures and algorithms and understand their efficiency in solving different problems. You'll gain insights into evaluating the efficiency of an algorithm and learn to choose the best approach among several options.

More on Scalability



Scalability is a term you often come across in the world of computer science. It's a combination of "scale" and "ability," representing the quality of an algorithm or system to handle larger and more complex problems.

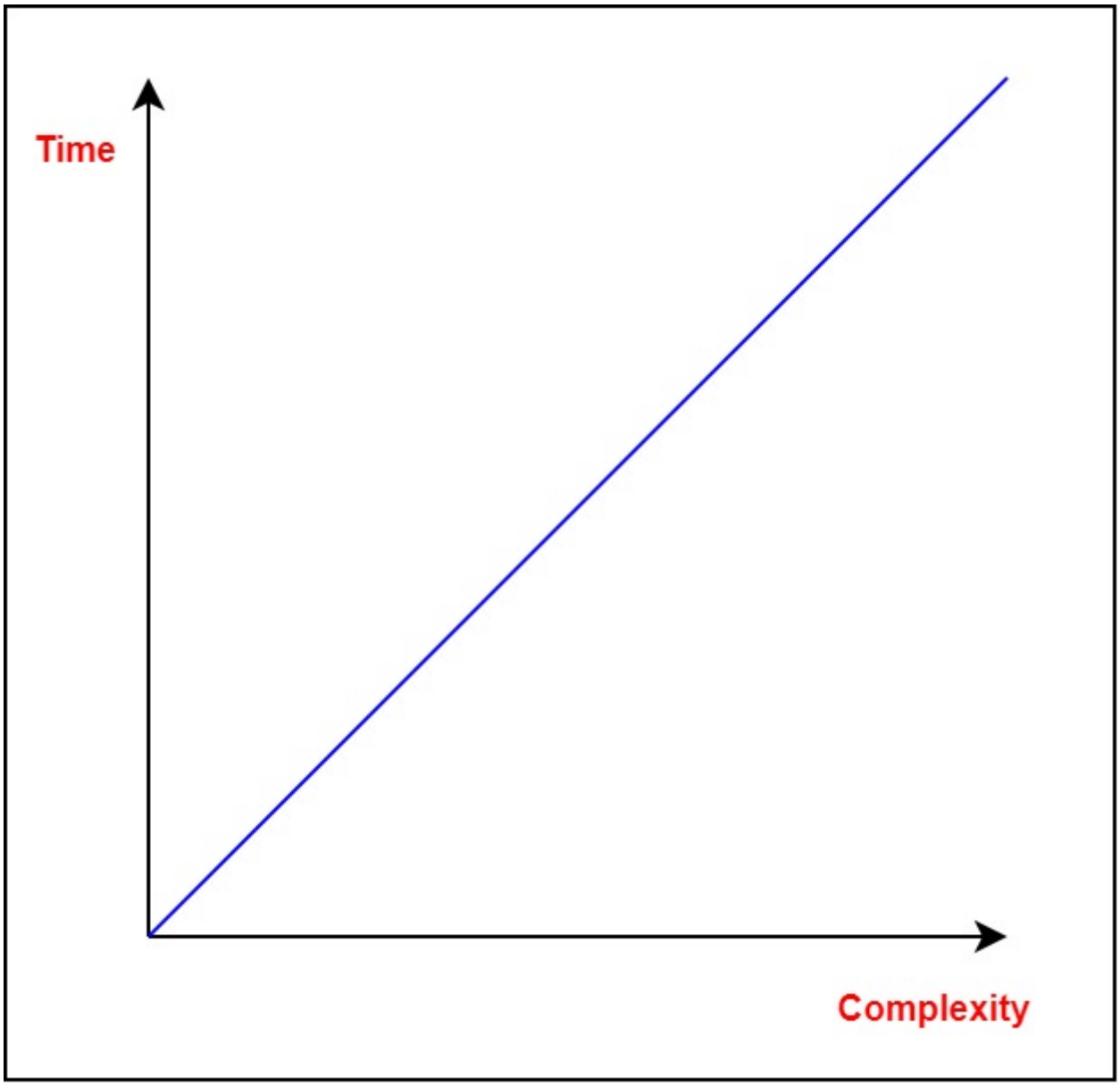
Let's illustrate this concept with a simple analogy. Imagine we need to set up a classroom for 50 students. The solution is quite straightforward; we can book a room, get a blackboard, a few chalks, and voilà, the problem is solved.

But what if the number of students increased to 200? Well, the same solution could still work, but it would require more resources. Perhaps we'd need a larger room, like a theater, a projector screen, and a digital pen to accommodate the increased number of students.

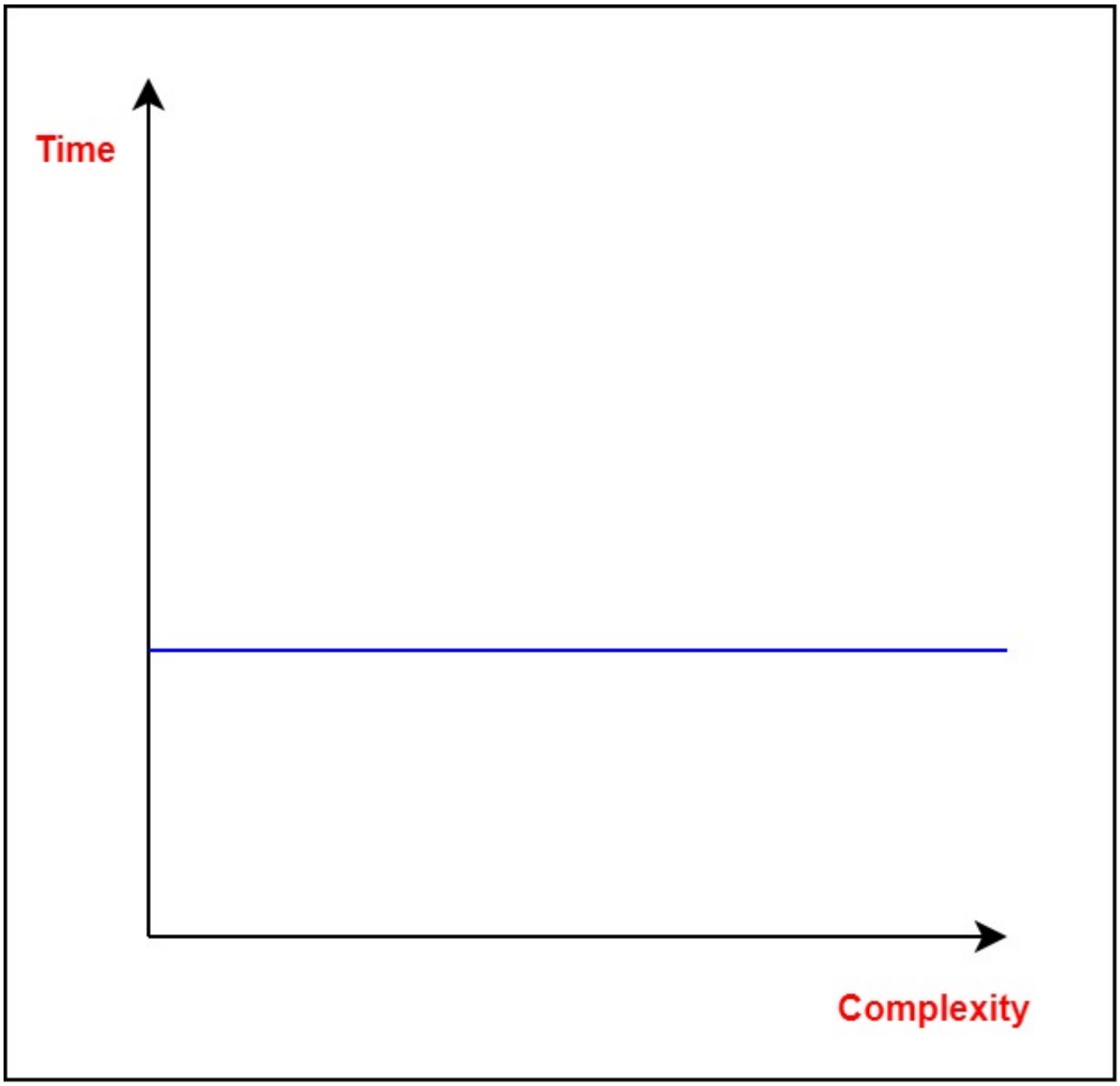
Now, let's take it a step further. What if the number of students increased to 1000? The previous solution might not suffice anymore. It could either fail to handle such a large number of students or consume an enormous amount of resources, making it inefficient. In this case, we can say that the solution is ***not scalable***.

So, what makes a solution scalable? A scalable solution can handle problems of larger sizes without requiring a significant increase in resources or time.

Now, let's relate this to the algorithms we discussed earlier. Remember our first solution to find the sum of the first `N` natural numbers? It wasn't scalable because it required ***linear growth*** in time with the linear growth in the size of the problem. Such algorithms are known as **linearly scalable algorithms**.



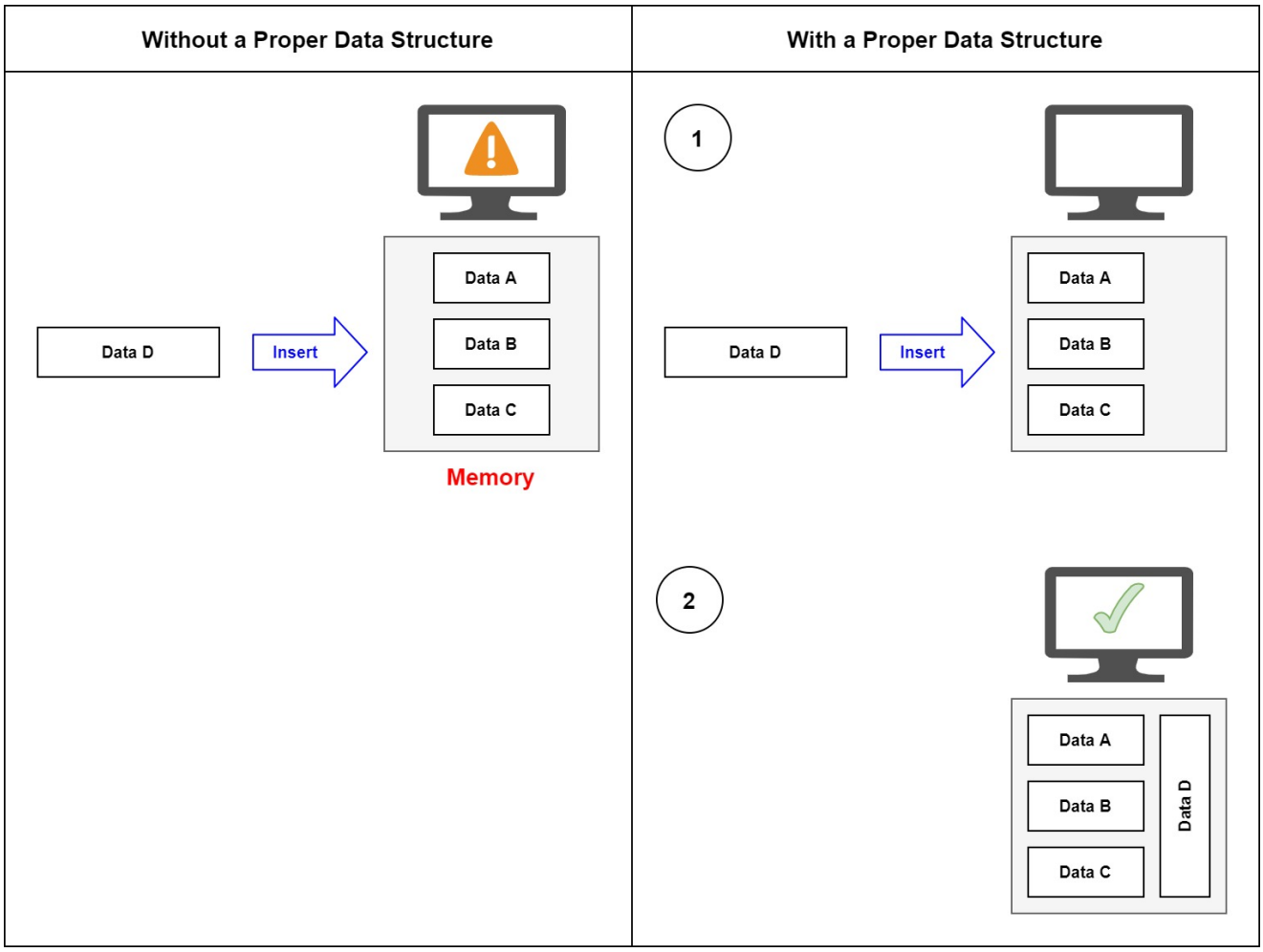
On the other hand, our second solution, which utilized the formula `Sum = N * (N + 1) / 2`, was very scalable. It didn't need any more time to solve the problem as the size of `N` increased. Such algorithms are known as **constant-time algorithms**.



As you progress in this course, you'll discover that optimizing algorithms for scalability is essential in building efficient software systems. But it's not just about time efficiency; memory efficiency also plays a crucial role.

Memory is expensive

Memory is expensive in terms of computation. While dealing with code or systems that require storing or processing a large amount of data, it's critical to save memory wherever possible. One way to achieve this is by carefully designing data structures to store only the necessary information. For example, when storing data about people, you can save memory by storing only their date of birth instead of their age. You can always calculate their age on the fly using their date of birth and the current date.



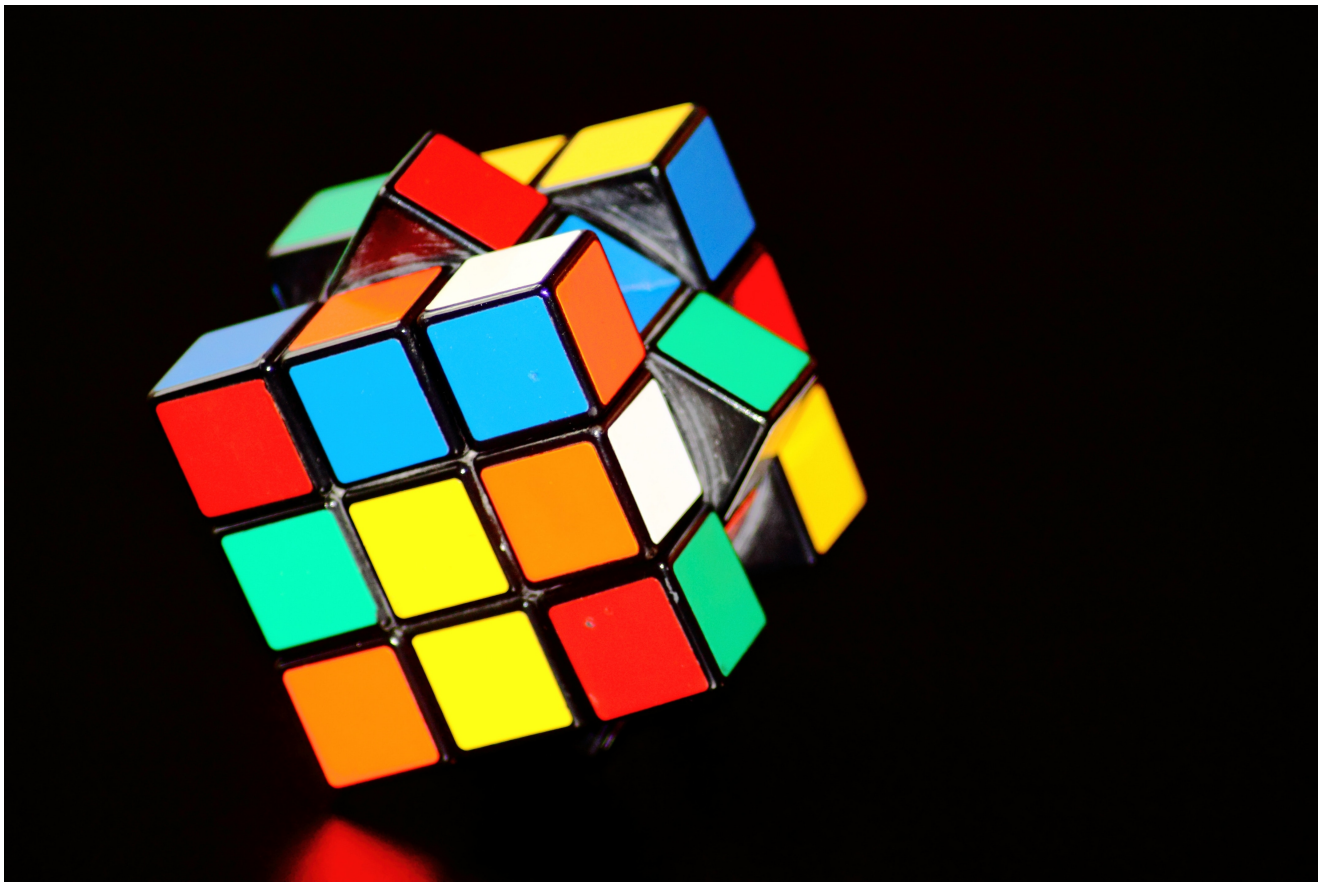
Examples of an Algorithm's Efficiency

Let's take a look at some examples of how learning algorithms and data structures can enable you to achieve impressive efficiencies in solving various problems:

Example 1: Age Group Problem

Suppose you need to find people of a certain age group. The naive algorithm that goes through all the persons one by one to check if they fall into the given age group is linearly scalable. On the other hand, the ***binary search algorithm*** claims itself to be logarithmically scalable. This means that if the size of the problem is squared, the time taken to solve it is only doubled. So, for a group of 1 million people, the binary search algorithm will take only 2 seconds to solve the problem, while the naive algorithm might take around 12 days!

Example 2: Rubik's Cube Problem



Imagine you're writing a program to find the solution to a Rubik's cube. With a mind-boggling 43,252,003,274,489,856,000 positions, you might wonder if it's even possible to solve it efficiently. Thankfully, the ***graph algorithm*** known as ***Dijkstra's algorithm*** comes to the rescue. It allows you to solve this problem in linear time, meaning it allows you to reach the solved position in the minimum number of steps.

Example 3: DNA Problem

In the field of bioinformatics, you may be tasked with finding a specific pattern in a DNA strand. The simplest algorithm would take time proportional to the number of characters in the DNA strand multiplied by the number of characters in the pattern. However, using the KMP algorithm, you can significantly speed up the process. It takes time proportional to the number of characters in the DNA strand plus the number of characters in the pattern. This can make the algorithm much faster, even up to 1000 times in some cases!

As you can see, learning algorithms and data structures empowers you with the ability to tackle complex problems efficiently and optimize your code for both time and memory.