What Is Computer Science?

Computer science is often difficult to define. This is probably due to the unfortunate use of the word "computer" in the name. As you are perhaps aware, computer science is not simply the study of computers. Although computers play an important supporting role as a tool in the discipline, they are just that–tools.

Computer science is the study of problems, problem-solving, and the solutions that come out of the problem-solving process. Given a problem, a computer scientist's goal is to develop an **algorithm**, a step-by-step list of instructions for solving any instance of the problem that might arise. Algorithms are finite processes that if followed will solve the problem. Algorithms are solutions.

Computer science can be thought of as the study of algorithms. However, we must be careful to include the fact that some problems may not have a solution. Although proving this statement is beyond the scope of this text, the fact that some problems cannot be solved is important for those who study computer science. We can fully define computer science, then, by including both types of problems and stating that computer science is the study of solutions to problems as well as the study of problems with no solutions.

It is also very common to include the word **computable** when describing problems and solutions. We say that a problem is computable if an algorithm exists for solving it. An alternative definition for computer science, then, is to say that computer science is the study of problems that are and that are not computable, the study of the existence and the nonexistence of algorithms. In any case, you will note that the word "computer" did not come up at all. Solutions are considered independent from the machine.

Computer science, as it pertains to the problem-solving process itself, is also the study of **abstraction**. Abstraction allows us to view the problem and solution in such a way as to separate the so-called logical and physical perspectives. The basic idea is familiar to us in a common example.

Consider the automobile that you may have driven to school or work today. As a driver, a user of the car, you have certain interactions that take place in order to utilize the car for its intended purpose. You get in, insert the key, start the car, shift, brake, accelerate, and steer in order to drive. From an abstraction point of view, we can say that you are seeing the logical perspective of the automobile. You are using the functions provided by the car designers for the purpose of transporting you from one location to another. These functions are sometimes also referred to as the **interface**.

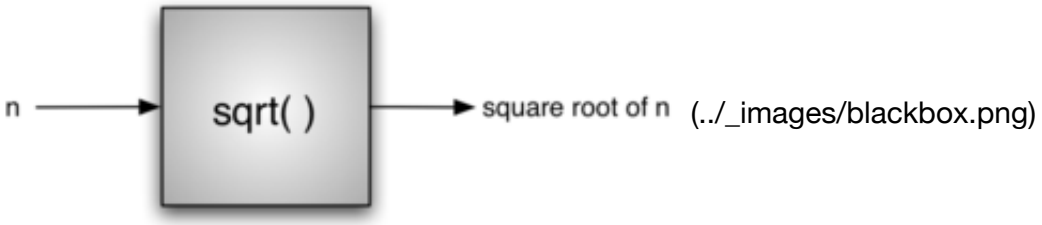
On the other hand, the mechanic who must repair your automobile takes a very different point of view. She not only knows how to drive but must know all of the details necessary to carry out all the functions that we take for granted. She needs to understand how the engine works, how the transmission shifts gears, how temperature is controlled, and so on. This is known as the physical perspective, the details that take place "under the hood."

The same thing happens when we use computers. Most people use computers to write documents, send and receive email, surf the web, play music, store images, and play games without any knowledge of the details that take place to allow those types of applications to work. They view computers from a logical or user perspective. Computer scientists, programmers, technology support staff, and system administrators take a very different view of the computer. They must know the details of how operating systems work, how network protocols are configured, and how to code various scripts that control function. They must be able to control the low-level details that a user simply assumes.

The common point for both of these examples is that the user of the abstraction, sometimes also called the client, does not need to know the details as long as the user is aware of the way the interface works. This interface is the way we as users communicate with the underlying complexities of the implementation. As another example of abstraction, consider the Python `math` module. Once we import the module, we can perform computations such as

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

This is an example of **procedural abstraction**. We do not necessarily know how the square root is being calculated, but we know what the function is called and how to use it. If we perform the import correctly, we can assume that the function will provide us with the correct results. We know that someone implemented a solution to the square root problem but we only need to know how to use it. This is sometimes referred to as a “black box” view of a process. We simply describe the interface: the name of the function, what is needed (the parameters), and what will be returned. The details are hidden inside (see Figure 1).



(../_images/blackbox.png)

Figure 1: Procedural Abstraction

❮ (GettingStarted.html) ❯ (WhatIsProgramming.html)

user not logged in