Programming: Plan First, Then Code

Many novice programmers attempt to dive right into writing the code (in the programming language) as the first step. However, writing the code is actually a much later step in the process. A good programmer will plan first and write second, possibly breaking down a large programming task into several smaller tasks in the process. Even when cautioned to plan first and code second, many programming students ignore the advice—after all, why "waste" 30 minutes planning when you are time-crunched from all the work you have to do. This tradeoff, however, presents a false economy—30 minutes planning could save hours of trying to make the code work properly. Well planned code is not only more likely to be correct (or at least closer to correct), but is also easier to understand—and thus fix.

To try to better understand the importance of planning before you write, imagine an analogy to building a house or sky scraper. If you were tasked with building a sky scraper, would you break ground and start building right away, figuring out how the building is designed as you go? Hopefully not. Instead, you (or an architect) would design blueprints for the building first. These blueprints would be iteratively refined until they meet everyone's specifications—they must meet the requirements of the building's owner, as well as be possible to build reasonably. Once the blueprints are completed, they must be approved by the local government. Actual construction only begins once the plans are fully completed. Programming should be done in a similar manner— come up with a complete plan (algorithm) first and build (implement in code) second.

We said that the heart of programming is to figure out how to solve a class of problems—not just one particular problem. The distinction here is best explained by an example. Consider the task of figuring out if a particular number (e.g., 7) is prime. With sufficient knowledge of math (i.e., the definition of a prime number and the rules of division), one can solve this problem—determining that 7 is in fact prime. However, a programming problem typically looks at a more general class of problems. We would typically not write a program to determine if 7 is prime, but rather a program which, given a number N, determines if N is prime. Once we have an algorithm for this general class of problems, we can have the computer solve any particular instance of the problem for us.

When we examine a class of problems, we have parameters which tell us which particular problem in the class we are solving. In the previous example, the class of problems is parameterized by N—the number we want to test for primality. To develop an algorithm for this class of problems, we must account for all possible legal values of the parameters. As we will see later, programming languages let us restrict what type of information a parameter can represent, to limit the legal values to those which make sense in the context of the problem. For primality testing, we would want our parameter N to be restricted such that it can only hold integer numbers. It would not make any sense to check if letters, words, or files are prime.

To write a program which takes any number N and determines if N is prime, we must first figure out the algorithm for this class of problems. As we said before, if we attack the problem by blindly writing code, we will end up with a mess—much like constructing a sky scraper with no plan. Coming up with the appropriate algorithm for a class of problems is a challenging task, and typically requires significant work and thought.