Developing Algorithms

Document prepared by Nicole Arruda and Nicole Binkowski for iCompute

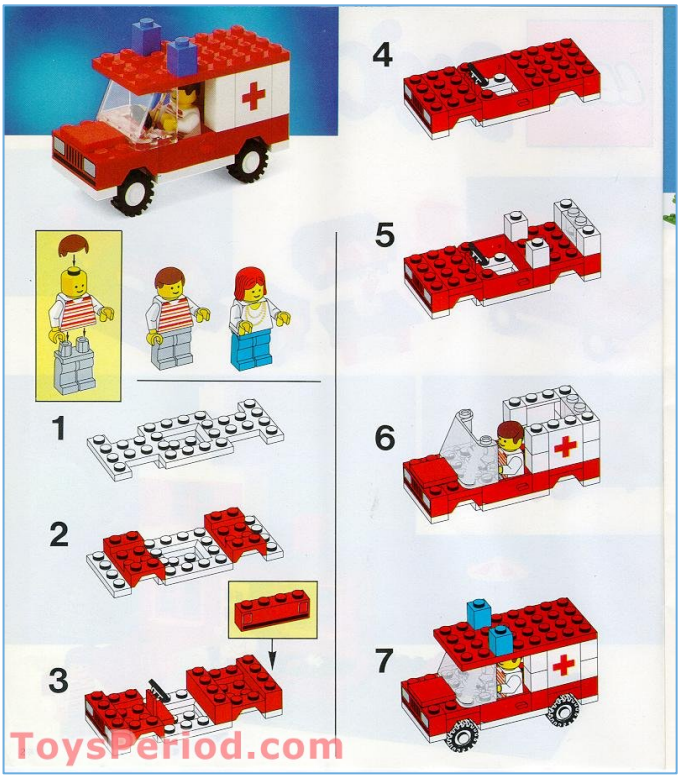
Introduction

Computer programmers write applications for computers that make life easier for all kinds of different people. There are applications to let us surf the Internet, write stories, play games, use medical equipment, launch spacecraft, control robots, and infinitely more. The point of an application is to let a human interact with a computer to solve some kind of problem.

When a programmer sits down to write an application, he is faced with the difficult task of translating these real-world problems into a program that the computer can understand and execute. The key to this process is what we call an *algorithm*.

What is an algorithm?

An algorithm is a finite series of steps to solve a problem. It's a set of instructions, often in plain English, that clearly lays out your plan from start to finish. As an example, look at the picture to the right. It was taken from a Lego kit. If you study just the first picture of the finished ambulance, you may struggle to recreate it. But when you look at the instructions that come with the kit, you can easily follow each small step.



Furthermore, if you read all of the instructions, you can convince yourself that following the steps will result in the completed ambulance. Even someone who has never worked with Legos before can probably read the instructions and understand the process. The Lego instructions are an *algorithm* for building the ambulance.

Everyday algorithms

We actually use algorithms every day, whether we realize it or not. The everyday algorithms we use may have nothing to do with computers. Often, they have to do with building things, like with the

Legos. We can also use algorithms to help us organize something, get from one place to another, and do anything else that requires a series of steps. Some examples are:

- Making a peanut butter and jelly sandwich
- Walking or driving home from a friend's house
- Making your bed
- Drawing a picture of a flower
- Completing a level in a video game
- Finding something to watch on TV
- Borrowing a book from the library

What are some other examples of algorithms you use every day?

How do algorithms relate to Computer Science?

We said earlier that algorithms are an important step in the programming process, but where exactly do they come in? Let's say I'm at an amusement park, and I have ten dollars. If it costs two dollars to go on a ride, how many rides can I go on? This is an example of a story problem you might see in math class. We know that a calculator can solve math problems, but could a calculator solve this problem as it's written? The calculator would have to understand English, which, of course, it can't.

Now think about how to rewrite this problem so that a calculator *could* solve it. In other words, turn this problem into an equation. Will you use addition? Subtraction? Which numbers will you use? How will you know whether you got the right answer? The process of translating a word problem into a math equation is a kind of *algorithm*.

The equation 10 / 2 = 5 can be used to solve the problem. We can tell a calculator to compute 10 / 2, and the calculator will give us 5 as a result. In the computing terms, this equation is like a computer program. We run the program, and it gives us a solution. To translate a real-world problem into a computer program, we will need to follow a similar process as we do to translate a story problem into a math equation. We need to develop an algorithm.

How do we develop an algorithm?

There is no exact formula for developing an algorithm, which can make it tricky to learn. Programming is a creative process, which means there might be several "correct" algorithms to solve the same problem. Fortunately, there are a few steps we can take to make sure our algorithm does what it's supposed to do.

1. Understand the problem. Before we try to write an algorithm, we should take a little time to answer a few questions about the problem we're trying to solve. The first question is: *how will we know when we're done?* If you're explaining to a friend how to get to your house, when do

you stop giving her directions? If she's been here before, maybe it's enough to direct her to your street, and she can find your house from there. However, if you live on the third floor of an apartment building, maybe you need to direct her all the way to your front door. Your algorithm will depend on the end result you're looking for.

2. Know your tools. What will your friend interact with to get to your house? Is she riding in a car or on a bike? Does she need money for a bus? Is there a buzzer or doorbell she will ring? Also think of the actions she will have to take and how she will know when to take them. Phrases like "turn left," "park," "ring the buzzer," and "when you get to the end of the street" can all be considered tools to follow your algorithm. List the tools your friend can use to follow your instructions.

3. Simplify the problem. Break the problem down into shorter, simpler steps for your friend to follow. This might be a list of the roads she'll take. It might include parking and getting to your front door on foot. This can be done in very general terms that only make sense to you, but make sure that you've covered everything from start to finish.

4. Translate the problem using your tool set. Finally, put it all together to complete your algorithm. Use the tools you generated in Step #2 to translate your thoughts into something your friend can follow. Make sure you write the steps in order so it's clear what should be done first, second, third, etc. Also make sure that you haven't included anything that wasn't in your toolbox. For example, the shortest route to your house might be by the highway, but if you didn't include "car" in your friend's toolbox, you can't use this instruction.

5. (optional) Make it better. Once you've tested your algorithm and you know that it works, you can investigate ways to improve it. Maybe your friend can get to your house in half the time if she takes the bus rather than walking, but it may require a more complex set of instructions. Just remember to first *make your program work*. If you have time, then you can make it work *better*.

Think about the Lego instructions on the first page. What tools do you think the instructions expect you to be able to use? Remember that tools can include *items* you use and *actions* you take.

Can you use this process to develop an algorithm to build your own Lego creation, make a paper snowflake, build a campfire, put away dishes, or solve some other problem? Once you've developed an algorithm that works, can you think of any ways to make it work faster or make it work under different circumstances?

Algorithms for computer programs

Knowing your tool set is crucial to developing an algorithm. In programming, our tools include concepts like the following:

- Variables
- Displaying output
- Arithmetic operations
- Loops
- Branching

These are concepts you will use to develop your algorithm. Once you have your algorithm, however, there is one more step: writing the actual program.

Communicating your algorithm to the computer will depend on the *programming language* you are using. Most programming languages can use the concepts outlined above. The difference is in the *syntax*, or what you need to type to make it happen.

Imagine again that you're giving directions to your house, but instead of English, your friend only speaks French. What will be different? Will your friend still perform all the same actions to get to your house? Of course she will. A French-speaking person has all the same tools available as an English-speaking person. The only difference is in how you tell her what to do.

If English is your first language, you will still develop your algorithm entirely in English. The very last step you take will be to translate your English algorithm into French. The same is true with programming. Once you have your algorithm that another human can understand, your final step will be to translate it into the programming language your computer will use.

Common programming algorithms

There are some problems that pop up all the time in programming. For example, how do we search for a word in a long list? How do we find the biggest or smallest number in a set? How do we sort numbers from smallest to largest? Over the years, programmers have developed and improved upon algorithms to solve these common problems. Here are some you should get familiar with:

- Linear search
- Binary search
- Selection sort
- Insertion sort

Pseudocode

Because algorithms are so important to developing good programs, it's critical that programmers be able to quickly communicate algorithms to one another. As we've discussed, most programming languages have similar tool sets available to them, so it would seem that the most effective way of communicating an algorithm is simply by passing along the finished code.

Unfortunately, with so many programming languages with such different syntax, this is not always possible. For example, a Java programmer may write a program using an algorithm that she wishes to explain to a Python programmer. Since Java and Python use very different syntax, the Java programmer cannot simply give her code to the Python programmer. Code can also take a long time to write, and when we discuss algorithms, we often want a quick way to express our thoughts.

One widely accepted solution to this problem is to write the algorithm in *pseudocode*, which means "fake code" or "simple code." The idea is to write an algorithm using words and symbols that represent common programming concepts.

As an example, here is a coding statement in the programming language Java:

```
for (int i = 1; i <= 5; i++)
    System.out.println("Hello");
```

Now, here is a statement in the language MATLAB:

```
for i = 1:5
    disp('Hello');
```

As a matter of fact, both of these statements perform the same function: they display the word "Hello" five times. What is the same between these statements? What is different? Notice that they both use the word "for." As it turns out, both of these statements express a "for-loop" in their own language. A for-loop is a programming construct used to do some action a certain number of times. In this case, the same line of code is executed five times.

Because almost every programming language has a way to express the for-loop, we can use a similar pattern in pseudocode for an algorithm and expect any programmer to understand what we mean.

Here is a statement in pseudocode:

```
for (i = 1 to 5)
    display("Hello")
```

Compare this statement with the previous two examples. What's the same? What's different? Hopefully, it is written in such a way that a programmer can understand the *algorithm*, or the steps involved, and use it to write a program in whichever language they please.

Does Scratch have a for-loop? Remember that it may not be *called* a for-loop, and it may not use the same syntax as Java, MATLAB, or the pseudocode. What's important is what a for-loop *does*: it repeats some process a certain number of times. Here's a hint: there *is* such a thing in Scratch. Can you find it?

Tips for writing algorithms

- Computers do not make assumptions. When you explain to a friend how to get to your house for the first time, you can't assume she'll know to look out for deer on a particular road or when exactly to turn onto your street. We write algorithms for computers as if we are explaining something for the very first time, and we must always keep in mind that the computer has no real understanding of the problem at hand. They do not take shortcuts or make assumptions.

- Order matters. We must pay very close attention to the order in which our instructions will be executed. Rather than "turn left after you've gone through three stoplights," we might prefer "after you've gone through three stoplights, turn left" because the stoplights come *before* the left turn. A computer will almost always execute your instructions in order.

- Trial and error. The best way to find out if your algorithm works is to *try it*. Write one step in your program, and then run it to see what happens. If it wasn't quite what you expected, you may need to change something. Once you've done the heavy thinking involved in developing an algorithm, trial and error is a great way to accomplish the final step of translating the algorithm into code.

- There is more than one way to write an algorithm. Your algorithm is right if it solves the stated problem, and it's wrong if it does not. However, there are always ways to make our algorithms better. "Better" might mean the algorithm can be easily interpreted by many other people. It might mean the problem can be solved faster or that it can be solved under different circumstances. Experiment with Scratch and with everyday algorithms to see if you can improve on old algorithms after you've found some that work.