Introduction to Computer Science

Shaan Mathur

October 23, 2018

Chapter 1

Introduction

One of the first things you hear when you begin your journey into Computer Science is that Computer Science is not programming. This often takes people aback, since the term 'Computer Science' now seems to remind people of the entrepreneurial and technologically driven industry that seems to be proliferating all over the modern world. But alas it is true, they are not equivalent.

Rather, programming is a subset of Computer Science, merely a tool to be used to concretely express our mathematical formulations of algorithms. A better way to look at Computer Science is as the mathematics of computation, a vast field with many pragmatic applications but also deep and meaningful theoretical roots. We formalize deep philosophical questions about our universe and the limits to what we can do in it, and then build theoretical models that allow us to build the systems we do today.

For the engineers, all this theoretical talk should not worry you. Being a Computer Science major teaches you to be the best programmer you can be, because it fundamentally teaches you how to think. There is an extraordinary amount of value in learning Computer Science to become a better programmer. Anyone can program; after a bit of trial and error anyone can learn how to get a working system going. But to write effective, well designed, and beautiful code requires a more fundamental understanding about how to approach computational problems in general.

Here I attempt to guide you through some fundamental ideas that I believe are critical to becoming not only a strong programmer, but an analytical, mathematical, and creative thinker.

1.1 Algorithms

Let's begin gently. What is an algorithm? An algorithm is a list of steps to be followed exactly to accomplish some task. For instance, to make a PB&J sandwich, I may take on the following algorithm.

1. Grab the jar of peanut butter

2. Grab the jar of jam

3. Grab a butter knife [1]

4. If there are at least two slices of bread left, take two. Otherwise if there is only one slice, take the one and cut it in half. In all other cases, we can't make a PB&J sandwich so we fail the algorithm (and maybe be sad).

5. On one slice spread peanut butter.

6. On the other slice spread jam.

7. Put one slice on top of the other

Easy! If we follow this blindly, we will certainly make a PB&J sandwich given that the supplies are present.

Could we do this faster? If my sister and I were working together to make a sandwich, then perhaps while I'm getting the peanut butter and jam, she could grab the knife and the bread. Perfect, we must then be reducing the amount of time taken and have created a faster algorithm! But can we really split the work for all steps? Not really, because I need the bread and knife my sister was fetching to spread the peanut butter and jam in steps 5 and 6. Determining how much faster my algorithm can be when I can work with more people becomes slightly more complicated than analyzing what would happen if it were just me, but not too much so.

Suppose it was just me making the sandwich. Is there a better, faster way to do things? Is my algorithm suboptimal, and we can make a PB&J sandwich much more quickly? Perhaps. Suppose, for instance, I were able to utilize both sides of the knife in a clever way to get peanut butter on one side and jam on the other. Then I can quickly put the spread on both slices of bread at the same time! Maybe even I could use a clever bread dispenser that will give me the bread more quickly than just unwrapping the plastic from my store-bought bread and grabbing two slices. There's plenty of optimizations to this algorithm one can make to get that delicious PB&J faster.

But would they be truly speed up the algorithm fundamentally? If I were in a competition with my brother to make the most sandwiches in ten minutes, and he used the 'faster' algorithm with the fancy bread dispenser, he might make a factor more than I can. But once we start making hundreds of millions of sandwiches, maybe his dispenser does not help as much as we originally thought; imagine comparing the million sandwiches I made versus the million and a thousand sandwiches he made. Yes it is more, but proportionally that factor is becoming less and less meaningful as we make more sandwiches. So arguably, both algorithms are asymptotically equivalent, or are roughly equivalent as we make enough sandwiches.

[1] When the author was around 5, he used a spoon instead to make a PB&J sandwich when his mother was asleep. He was very proud.

1.1.1 The Dictionary Algorithm

Suppose I have a dictionary. It's rather bland on the outside; it has no markings whatsoever. But inside it has every single word in the English language! I ask you, how do you find the word 'eureka'? Pause and think about it for a moment.

The obvious approach is to just flip through every page, left to right, until you find the word. But there are over 170,000 words in the English language, and that seems just too exhausting.

But if you were to really pick up the book and leaf through, we automatically do something more intelligent. We open a random page and look at the words on that page. If they are alphabetically farther along in the book, we know we have got to start turning pages back; if it comes alphabetically before 'eureka', then it's later in the book. This is an intuitive idea we all enact on subconsciously when looking something up.

What if we expand upon this idea? Whenever we look at some page in the book, we either can eliminate all pages on the right, or all pages on the left. Since on average the word could be on the left or right, our best bet is to just open the book in the middle and see how far along we are. Now we've reduced the problem size by half, because it can only be on one side of the book!

But how do we continue? We could just move left to right, but why not perform the same clever trick again? Go halfway on that side and determine what half of *that* half to look in. In fact, we could keep dividing our problem size by half at every step.

The following algorithm outlines this idea:

1. Open the book to the halfway point
2. If the word is on the page, we are done. If the page is too far ahead, apply the same algorithm on the left side. Otherwise, apply the same algorithm on the right side.

That's a simple and beautiful algorithm to find any word fairly quickly in the dictionary. If there are 170,000 pages and we keep dividing it by half, the number of pages we actually have to see would be $\log_2(170,000) \approx 18$. We would only have to look at 18 pages! Not bad for looking at so many words.

Chapter 2

Programming

Programming languages are tools we can use to concretely build and run our algorithms in the real world. A programming language is a human-readable language used to specify what we would like the computer to do. Once we have written the program, another program comes along and converts our program into machine language, the sequence of 1's and 0's that computers interpret in order to determine what to do next [1].

Learning one programming language generally allows us to learn nearly any programming language, since many languages share similar constructs. We choose to use the programming language C++ because it is immensely powerful, and learning C++ well will not only give a strong intuition over how computers work but will also introduce you to many different programming paradigms as well.

2.1 Variables

You own a small ice cream shop, and you've realized that technology can help increase overall productivity in the workplace. You want to have software available to your cashiers so that the cashier can simply input the customer order and quickly calculate what the final total is after tax (this way you can hire cashiers who don't have a degree in mathematics without the risk of miscalculations). You specifically are looking for a program that asks the user for the the weight of the bowl (in oz) and then outputs the total cost, where the cost per ounce is 50¢.

How would we design such a program? Well, our one sentence specification actually does detail how the overall structure of the program should be.

[1] Technically this is a multistep process that depends on the language being written. For C++, a *compiler* first compiles the code into a simpler language called assembly; an *assembler* then takes the assembly code and converts it into machine language.

Algorithm 1: A high-level overview over the structure of the program.

1 Get from the user the **weight** of the bowl.
2 Compute the **total cost** as \$0.50 multiplied by the **weight**.
3 Output the **total cost**.

A key aspect of this program is the ability for the computer to actually *remember* information from the previous lines; the terms in bold are all values that needed to be remembered as we progressed through our step-by-step instructions. This is a fundamental idea in programming and computation, and we actualize it in our programs through the means of **variables**.

A variable is a named piece of memory to which we can read and write data. In our program above, we store the user input into a variable named *weight*, multiply the value of *weight* by .5 to store into another piece of memory called *total cost*, and then output *total cost*. As you can see, our pseudo-algorithm here is not even in any particular programming language, and yet the idea of having memory is an essential part of our intuition over how to solve this problem.

Algorithm 2: Our program with variables.

1 $weight \leftarrow$ real number from input
2 $totalcost \leftarrow .5 * weight$
3 Output *totalcost*

What kinds of variables can we think of? Here certainly we can see that we want numbers; if we wanted users to register accounts for a reward system for the ice cream shop, perhaps we would also want to be able to enter words; if we want to know whether the user already has an account, we might want to receive some sort of "yes" or "no" answer (though this could be done with words, too). This motivates the idea of different *types* of variables, with some for numbers, words, and maybe other ideas we haven't thought of yet. In languages like C++, the language designers decided that these variables are fundamentally different enough that we should be specific about what variable type we intend. For instance, we can translate our pseudo-program into a real snippet of C++ code (though not a full program).

Listing 2.1: A C++ Snippet of the Ice Cream Shop Program.

```
double weight, total_cost;
cin >> weight; // Gets user input and stores into weight
total_cost = .5 * weight;

// Outputs the value of total_cost and moves cursor to next line
cout << total_cost << endl;
```

The first line of the code declares two variables of type "double", which is another way of saying we create two variables whose values can only be numbers with decimal places (i.e. rational numbers). After retrieving a value for *weight* from the user, we store the value of $.5 * weight$ into variable *total_cost*, which we promptly output.

Here is a table of other types of variables and how they are used in C++.

Type	Number of Bytes	Values (b = bytes)
short/int/long	2/4/8	integer values from -2^{8b} to $2^{8b}-1$
float	4	+/- 3.4e +/- 38 (approx. 7 digits)
double	8	+/- 1.7e +/- 308 (approx. 15 digits)
char	1	-128 to 127 (see ASCII)
bool	1	true or false

How do you know when to use what kind of variable? It mainly depends on your own evaluation of the requirements of the specification. For instance, American currency is often in terms of dollars where two decimal places are left to specify cents; thus a float or double would suffice here. When dealing with the number of friends you have in a social network, integers (short/int/long) are a more natural measure. Characters (char) make sense when dealing with a single character; the context in which they become more prevalent is when we put together a string of characters to form a word (an explanation for why that more complicated datatype is called a string). Finally booleans (bool) are variables that are either true or false; this becomes relevant when asking yes or no questions such as *Does the user have a registered account?* or *Is the ice cream flavor strawberry?*.

Sometimes we might want many variables instead of just a few; for example if our ice cream shop wants to start remembering its customers, we might want to build a whole list of customers to keep track of. If we know that our town shop could at most have 300 customers, we could have 300 separate variables to keep track of each customer. But this is incredibly tedious, and what will happen if one day we need to add more customers?

To assuage this situation, we can introduce the notion of **arrays**, which is a *contiguous* list of variables. For instance, if we want to keep track of usernames as strings (the C++ type that stores double-quoted sequence of characters like "banana", "234afsdfv", ""), we could declare an array of 300 strings and access the i^{th} string by *indexing* into the i^{th} entry of the array as follows.

Listing 2.2: Arrays

```
// const makes the variable unchangeable
const int LENGTH = 3;

// <type> <var_name>[<length>];
string names[LENGTH];

names[0] = "shaan"; // The first element is at the 0th index
names[1] = "devan";
names[2] = "saira";
```

Note that for an array of length n, the indices start from 0 and go up to $n-1$; this is a convention that is heavily used in computer science and various branches of mathematics. Also it is important to note that in C++ the length of the array must be decided at compile time, so it can't be decided at run time (e.g. as user input). In this example it works because *LENGTH* is a constant integer that won't change in value, so C++ knows the length at compile time.

2.2 Conditional Statements

As your ice cream shop begins to burgeon, you start adding more exotic, although more expensive, flavors of ice cream on your menu. Therefore you want to update your cashier software so that if a customer orders deluxe caramel ice cream they will be charged 60¢ per ounce. How would we be able to do this?

Given what we know so far, there seems to be some issues in designing our program. The program we wrote before had a straightforward description in that no matter what, we would just have to multiply the weight by .5 to get the final price. The difference now is that we only want that code to run *sometimes*; the other times we need another piece of code to run, identical in every way except multiplying the weight by .6 for the final price. We want a way to execute our original program **if** the ice cream **is not** caramel; otherwise we run the same program except with .6 instead of .5.

This motivates us to define the notion of an **if statement**, which is a block of code that executes if some specified condition is satisfied. We can rewrite the prior C++ snippet to reflect our new update in requirements.

Listing 2.3: Updating the Ice Cream Shop Program.

```
char answer;
double weight, total_cost;

// Receive 'Y' or 'N' as input for whether to increase price
cout << "Is the ice cream caramel deluxe? ";
cin >> answer;

// Gets user input and stores into weight
cout << "How much does the ice cream weigh (oz)? ";
cin >> weight;

if (answer == 'Y') {
    // Executed if input was 'Y'
    total_cost = .6 * weight;
} else {
    // Executed in all other cases
    total_cost = .5 * weight;
}

// Outputs the value of total_cost and moves cursor to next line
cout << total_cost << endl;
```

Now our program asks a yes or no question, to which the user is to respond 'Y' or 'N'; if the user responds that the ice cream is caramel deluxe, then the total cost is calculated with a 60¢ per ounce cost, otherwise calculating with 50¢ per ounce. Note that this program is not perfect either, because we do not verify that the input character is a valid one. What would happen if the user input 'y'?

The general structure for conditioning goes as follows:

Listing 2.4: If-ElseIf-Else

```
if (<condition1>) {
    // executed if <condition1> is true
} else if (<condition2>) {
    // executed if <condition1> is false and <condition2> is true
} else if (<condition3>) {
    // executed if <condition1> and <condition2> are both false
    // and <condition3> is true
}
// ...
} else {
    // executed if all conditions are false
}
```

Another interesting method of conditioning is the **switch statement**, which is used when you want to condition based upon the value of an integer variable. For instance, suppose you have an integer called *color* where we say 0 is red, 1 is green, and 2 is blue. Then we might write something like the following

Listing 2.5: Switch Statements.

```
int color;
cin >> color;

switch (color) {
    case 0: // Red
        // color the screen red
        break;
    case 1: // Green
        // color the screen green
        break;
    case 2: // Blue
        // color the screen blue
        break;
    default:
        cerr << "Bad input!" << endl;
}
```

When the switch statement is entered, execution *jumps* to the case matching the value of *color*, and goes to default otherwise. Note that in order to leave the switch statement, a "break;" statement must be included to leave; otherwise execution will fall into the next case, as in the following example.

Listing 2.6: Falling Through a Switch Statement.

```
switch (flag) {
    case 0:
        cout << 0 << endl;
        // Fall through to 1
    case 1:
        cout << 1 << endl;
        break;
    case 2:
        // Fall through to 3
    case 3:
        cout << 2 << " or " << 3 << endl;
        // Fall through to default
    default:
        cerr << "Bad input!" << endl;
}
```

When *flag* is 0, we print 0, fall to case 1, print 1, and then exit the switch statement; if *flag* is 1, we only print 1 and then exit the switch statement. What happens when *flag* is 2? 3? Other values?

2.3 Loops

With what has been covered so far, we are able to describe algorithms whose behavior may vary depending on user input or any other values that could vary. However it is often the case that we want to perform some action repeatedly for either a specified number of times or when some condition is met. For instance, suppose we have an array of daily temperatures in Westwood for the past 5 years; it would be extremely tedious to try to write enough nested if statements to try and find the maximum temperature of the past 5 years. However the idea of looping can help us solve this problem. The structure of a for loop is as follows:

Listing 2.7: For Loop

```
for (<initializer>; <condition>; <increment>) {
    // body
}
```

In execution, the initializer segment initializes any variables whose lifetime will last until the for loop is exited; the condition statement is checked prior to every iteration, and will cause the loop to terminate if the condition is false; the increment segment will be invoked at the end of every for loop. Take the following example:

Listing 2.8: Printing Numbers.

```
int n;
cin >> n;

for (int i = 0; i < n; i++) { // i++ is the same as i = i + 1
    cout << i << endl;
}
```

Suppose the user sets n as 5. When entering the for loop, a variable named i will be initialized to 0. The condition "$i < n$" will be checked, and since $0 < 5$ the body of the loop will be executed and will print 0. Then the increment segment will be invoked to increment i to the value $i + 1$, and then the condition will checked and verified again before the body is executed. As you can see, the for loop will print out numbers 0 through 4 before terminating.

To solve the problem from before, we suppose we have some integer array called *temperature[]* that is defined where n is the length of the array. How would we find the largest element? We could check each element one by one and keep track of the maximum element found so far; after checking all the elements, we will surely have the maximum.

Listing 2.9: Max Temperature.

```
int max = temperature[0];
int max_index = 0;
for (int i = 1; i < n; i++) {
    if (temperature[i] > max) {
        max_index = i;
        max = temperature[i];
    }
}

cout << "The max temperature is " << max << " at index " <<
    max_index << endl;
```

What happens if we moved the declaration of *max* or *max_index* inside the body of the for loop? What if we included it in the initializer segment?

In other scenarios, we may not know how often we want to repeat some behavior. For instance, suppose in our program for the ice cream shop we ask for the weight of the bowl and a negative number is provided; we may wish to ask the user to resubmit a new number again, but perhaps they will give bad input once again. While the input continues to be bad, we want to keep asking for the user to resubmit the input. This motivates the notion of a **while loop**, which continues looping until the condition fails.

Listing 2.10: While Loop

```
while (<condition>) {
    // Execute repeatedly while condition remains true
}
```

So to enforce that the weight of the bowl must be a positive number, we may write something like the following.

Listing 2.11: While Loop for Bad Input

```
double weight;
cout << "What is the weight of the ice cream? ";
cin >> weight;

while (weight <= 0) {
    cout << "Weight must be positive. Try again. ";
    cin >> weight;
}
```

Note that if the while loop did not include the final statement which updates the value of *weight*, the loop would repeat infinitely forever (rightfully called an *infinite loop*). It is important to remember that if we want to avoid infinite loops[2], we should almost always have a statement in the while loop that helps us make progress towards failing the looping condition.

It is a helpful exercise to see how we might convert a for loop into a while loop and vice versa. Note that in C++ we can place empty statements (just a semicolon ;) in place of any the segments; when it tries executing the empty statement, nothing happens[3]. Another nice trick is to use the comma operator (just a comma ,) to put multiple statements in the initializer or increment segments.

Listing 2.12: For Loop Tricks.

```
for (;;) {
    // Loops forever!
}

// Multiple counters at the same time!
for (int i = 0, j = 0; i + j < 50; i++, j += 2) {
    cout << i << " " << j << endl;
}
```

Sometimes just a for loop is not enough. For instance, suppose we have an array of integers and we want to identify if the array has a duplicate integer. One simple algorithm for doing this would be to fix a specific integer in the array and then look at the rest of the array to see if that integer appears again; after trying this for each element in the array, we would have come across a duplicate if there is one. If we assume we already have fixed an index i, we would want to compare the element at that index against all other elements.

[2]Sometimes we may want an infinite loop, particularly in programs that intend to run for an indefinite period of time. For instance consider that the ice cream shop software probably should run forever so that we don't have to restart the program when the next customer in line arrives at the register. To execute this, we would wrap our entire program in a while statement whose condition is the boolean value *true*.

[3]Surprise, surprise.

Listing 2.13: Detecting Duplicate for Fixed Index.

```
// Assume we are given an array of integers called arr of length n
    and fixed index i

bool duplicate_found = false;
for (int j = 0; j < n; j++) {
    if (j != i && arr[j] == arr[i])
        duplicate_found = true;
}
```

However now we have to do this over all possible values of i; hence we need to place this loop inside of *another* for loop which varies the value of i.

Listing 2.14: Nested For Loops.

```
// Assume we are given an array of integers called arr of length n
    and fixed index i

bool duplicate_found = false;

// Varies the value of i
for (int i = 0; i < n; i++) {

    // For fixed i, varies the value of j
    for (int j = 0; j < n; j++) {
        if (j != i && arr[j] == arr[i])
            duplicate_found = true;
    }
}
```

It is important to practice writing toy programs like this to gain comfort with the idea of nested for loops. Nested loops is something that occurs regularly in software development, and it is important to intuitively understand how the flow of control works when nesting loops.

2.4 Functions

So far all we have done is write snippets of C++, but nothing that legitimately compiles. To properly compile, several things have to be resolved. First of all, what is *cout*, *cin*, and *endl* and where are they even defined? It turns out there is a library called "iostream" that contains a lot of useful code for input and output. Furthermore these symbols are technically called *std::cout*, *std::cin*, and *std::endl*, but we can omit the *std::* and write in the beginning of our code that if there is an unrecognized symbol to check whether it can begin with *std::*.

Finally and perhaps most importantly, our C++ cannot actually float in free space like we have been writing so far. Every block of code must be named, and the block of code that is by default executed is always called *main*.

Listing 2.15: A Full Program.

```
// Includes the iostream library to gain access to std::cout, std::
    cin, std::endl
#include <iostream>

// If there is a symbol used anywhere that was not defined, please
    check std:: first
using namespace std;

// The main function that is always invoked first
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Let's dive deeper into what that whole block of code named *main* exactly is. As we have been thinking about small toy problems so far, we've been coming across ideas that may actually be useful to have. For instance when we found the maximum temperature in an array, we were really writing an algorithm for finding the maximum element of an array (using the ¿ operator). Finding the maximum element may be something we do often in our code, particularly if we write programs that may be looking for trends in data; so wouldn't it be nice to be able to reuse the code we wrote earlier?

We thus introduce the notion of a **function**, a block of code that may take in some arguments and then *returns* a result that is a consequence of the input arguments. This may sound abstract, but the idea is fairly intuitive.

Listing 2.16: C++ Functions

```
<return_type> <function_name>(<arg1_type> <arg1_name>, ..., <
    argn_type> <argn_name>) {
    // body

    return <value>; // If necessary
}
```

Suppose we wanted to have a convenient way of taking an array of integers and finding its maximum element. The algorithm we wrote depended on both the array itself and the length of that array. Thus we may write a function like the following.

Listing 2.17: Maximum of an Array.

```
/* @param arr       An array of integers
 * @param length    The length of arr
 *
 * @returns the maximum integer in arr
*/
int max_array(int arr[], int length) {
    if (length <= 0)
        return -1; // Bad value

    int max = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}
```

Here we assume we are given an integer array called *arr* and the length of that array in a integer called *length*. We then calculate the maximum and then *return* the value of *max* to whoever called this function. To see how this would be used in a full program, consider the next example.

Listing 2.18: Full Program with *max_array* Function.

```
#include <iostream>

using namespace std;

/* @param arr       An array of integers
 * @param length    The length of arr
 *
 * @returns the maximum integer in arr
*/
int max_array(int arr[], int length) {
    if (length <= 0)
        return -1; // Bad value

    int max = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

int main() {
    int test_scores[] = {94, 82, 41, 96, 88, 89};
    int max_score = max_array(test_scores, 6);
    cout << "The maximum score is " << max_score << "." << endl;
}
```

Note that we need to place *max_array* before *main* because the code in *main* actually refers to *max_array*; since the compiler reads the code from top to bottom, it needs to know what *max_array* is before any code can actually

refer to it. This can start to become problematic if your code is large or you have circular references amongst some of your functions. To solve this in this case, one can declare near the top of the file that there is a function called *max_array* that takes the specified arguments and has the integer return type; such a declaration is called a **function prototype**.

Listing 2.19: Full Program with *max_array* Function.

```
#include <iostream>

using namespace std;

// Promises to the compiler there is a function called max_array.
// This is just the top of the function, (the function's signature)
int max_array(int arr[], int length);

// Since the compiler only cares at this point that the function
// exists, it only needs to know what kind of function it is.
// Thus we can omit the names of the variables if we'd like

// int max_array(int[],int);

int main() {
    int test_scores[] = {94, 82, 41, 96, 88, 89};
    int max_score = max_array(test_scores, 6);
    cout << "The maximum score is " << max_score << "." << endl;
}

/* @param arr       An array of integers
 * @param length    The length of arr
 *
 * @returns the maximum integer in arr
*/
int max_array(int arr[], int length) {
    if (length <= 0)
        return -1; // Bad value

    int max = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}
```

Functions need not have arguments, as is evident from the *main* function that takes no arguments. The main function in C++ is the de facto function that is always executed first; the other code can only be executed if the flow of control in main decides to execute it. Notice that the return type for main is an integer, and that we usually return the value 0. Larger programs may fail in various ways, so the return value from the main function can tell the operating system if the program succeeded or not, where 0 is the standard value for success (1 is often used to denote some general kind of failure).