



TASK

Thinking Like a Programmer — Pseudo Code

Visit our website

Introduction

WELCOME TO THE PSEUDO CODE TASK!

This task will introduce the concept of pseudo code. Although pseudo code is not a formal programming language, it will ease your transition into the world of programming. Pseudocode makes creating programs easier because, as you may have guessed, programs can sometimes be complex and long — preparation is key. It is difficult to find errors without understanding the complete flow of a program. This is the reason why you will begin your adventure into programming with pseudo code.



Get in touch
Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at <https://discord.com/invite/hyperdev> where our specialist team is ready to support you.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!





A note from the HyperionDev Team

Chances are, if you're taking this course, you're brand new to programming. Well, you've come to the right place!

The key to becoming a competent programmer is utilising all available resources to their full potential. Occasionally, you may stumble upon a piece of code that you cannot wrap your head around. Knowing which platforms to go to for help is essential!

HyperionDev knows what you need, but there are other avenues you can explore for help. One of the most frequently used by programmers worldwide is [Stack Overflow](#).

Also, [our blog](#) is a great place to find detailed articles and tutorials on concepts into which you may want to dig deeper. For instance, when you get more comfortable with programming, if you want to [read up about creating a website on our blog](#), or learn [the difference between React, Vue and Angular](#), you could find all this information and more on our blog.

Our blog is updated frequently, so be sure to check back from time to time for new articles or subscribe to updates through our [Facebook page](#).

INTRODUCTION TO PSEUDO CODE

What is pseudo code? And why do you need to know this? Well, being a programmer means that you will often have to visualise a problem and know how to implement the steps to solve that particular conundrum. This is where pseudo code comes in.

Pseudo code is a detailed yet informal description of what a computer program or algorithm must do. It is intended for human reading rather than machine reading. Therefore, it is easier for people to understand than conventional programming language code. Unlike conventional programming languages, pseudo-code does not need to obey any specific syntax rules. Hence it can be understood by any programmer, irrespective of the programming languages they're familiar with. **As a programmer, pseudo code will help you better understand how to implement an algorithm**, especially if it is unfamiliar to you; it will also help you to communicate your ideas to other programmers in the workplace who have

different programming language competencies to yours. You can then translate your pseudo code into your chosen programming language.

Pseudocode is often used in computer science textbooks and scientific publications in the description of algorithms so that all programmers can understand them, irrespective of their knowledge of specific programming languages. It is also used by programmers in the planning of computer program development to initially sketch out their code before writing it in its actual language.

For this course, pseudo-code can also help guide the comments that you make in your program to explain your code, but don't worry too much about that for now.

WHAT IS AN ALGORITHM?

Now, what exactly is an algorithm? Simply put, an algorithm is a step-by-step method of solving a problem. To understand this better, it might help to consider an example that is not algorithmic. When you learned to multiply single-digit numbers, you probably memorised the multiplication table for each number (say n) all the way up to $n \times 10$. In effect, you memorised 100 specific solutions. That kind of knowledge is not algorithmic. However, you probably recognised a pattern along the way and made up a few tricks. For example, to find the product of n and 9 (when n is less than 10), you can write $n - 1$ as the first digit and $10 - n$ as the second digit (e.g. $7 \times 9 = 63$). This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm! Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms.

One of the characteristics of algorithms is that they do not require any intelligence to execute. Once you have an algorithm, it's a mechanical process in which each step follows from the last according to a simple set of rules (like a recipe). However, breaking down a hard problem into precise, logical, algorithmic processes to reach the desired solution is what requires intelligence or what is called "computational thinking".

This process of designing algorithms is interesting, intellectually challenging, and a core part of programming. Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

Programmers usually just write pseudo code for themselves before they start writing code. But, in some instances, you may be required to draft algorithms for a third person. Therefore, it is essential that your algorithms satisfy a particular set of criteria so that it can be easily read by anyone.

The algorithm should usually have some *input* and, of course, some eventual *output*. Input and output help the user keep track of the current status of the program. They also aid in debugging if any errors arise. For example, when you have a series of calculations in your program that build off each other, it is helpful to print out each of the programs to see if you're getting the desired result at each stage. Using this approach, if a particular sequence in the calculation is incorrect, you will know exactly where to look and what to adjust.

Clarity is another criterion to take into consideration in the development of algorithms. Your algorithm should be unambiguous. Ambiguity is a type of uncertainty of meaning in which several interpretations are plausible. It is thus an attribute of any idea or statement whose intended meaning cannot be definitively resolved according to a rule or process with a finite number of steps. In other words, your algorithms need to be as clear and concise as possible to prevent unintended outcomes.

Furthermore, algorithms should correctly solve a class of problems. This is referred to as *correctness* and *generality*. Your algorithm should be executable (able to get executed) without any errors, and should successfully solve the intended problem.

Last but not least, you should note the *capacity* of your resources, as some computing resources are finite (such as CPU or memory). Some programs may require more RAM than your computer has or take too long to execute. This may seem irrelevant with the sort of programs you write as you start to learn to code, but becomes particularly important when writing enterprise-level code as a professional programmer. Therefore, it is imperative to develop a coding approach and style that takes into consideration ways in which the computational load on your machine can be minimised.



A note from the **Hyperion Team**

Did you know that [Ada Lovelace](#) (who was an English mathematician and writer born in 1815) is regarded as the first computer programmer? Her notes on an early mechanical, general-purpose computer (the Analytical Engine) are recognised as the first algorithm (pseudocode) to be carried out by a machine.



Ada Lovelace

Also, did you know the word 'algorithm' comes from the medieval Latin word 'algorism' and the Greek word 'arithmos' (ἀριθμός)? The word 'algorism' (and therefore, the derived word 'algorithm') comes from Al-Khwārizmī (Persian: خوارزمی, c.780–850), a Persian mathematician, astronomer, geographer and scholar. The English language adopted the French term, but it wasn't until the late 19th century that 'algorithm' took on the meaning that it has in modern English.

In English, it was first used in about 1230 and then by Chaucer in 1391. Another early use of the word is from 1240 in a manual titled *Carmen de Algorismo* composed by Alexandre de Villedieu.

It begins thus:

*Haec algorismus ars praesens dicitur, in qua / Talibus Indorum fruimur bis
quinque figuris.*

which translates as:

Algorism is the art by which at present we use those Indian figures, which number two times five.

The poem summarises the art of calculating with the new style of Indian dice, or Talibus Indorum, or Hindu numerals (al-adad al-Hindi).

VARIABLES

In a program, variables act as a kind of 'storage location' for data. They are a way of naming or labelling information so that we can refer to that particular piece of information later on in the algorithm. For example, say you want to store the age of the user so that the algorithm can use it later. You can store the user's age in a variable called "age". Now every time you need the user's age, you can use the variable "age" to reference it.

As you can see, variables are instrumental when you need to use and keep track of multiple pieces of information in your algorithm. This is just a quick introduction to variables. You will get a more in-depth explanation later on in this course.

INPUT AND OUTPUT

Input is data or information that is transferred to the computer. It is sent to the computer using an input device, such as a keyboard, mouse or touchpad.

Output is information that is transferred out from the computer, such as anything you might view on the computer monitor. Output is sent out of the computer using an output device, such as a monitor, printer or speaker.

Take a look at the pseudo-code example below that deals with multiple inputs and outputs:

Example 1

Problem: Write an algorithm that asks a user to input a password and then stores the password in a variable called *password*. Subsequently, the algorithm requests input from the user. If the input does not match the password, it stores the incorrect passwords in a list until the correct password is entered, and then prints out the variable "*password*" and the incorrect passwords:

Pseudo code solution:

```
request input from the user
store input into variable called "password"
request second input from the user
if the second input is equal to "password"
    output the "password" and the incorrect inputs (which should
    be none at this point)
if the second input is not equal to "password"
    request input until second input matches password
when second input matches "password"
    output "password"
    and output all incorrect inputs.
```

THINKING LIKE A PROGRAMMER

Thus far, you've covered concepts that will see you starting to think like a programmer. But, what exactly does thinking like a programmer entail?

Well, this way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behaviour of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is problem-solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practise problem-solving skills.

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

Another important part of thinking like a programmer is actually thinking like a computer. The computer will run code in a step-by-step, line-by-line process. That means that it can't jump around! For example, if you want to write a program that adds two numbers together, you have to give the computer the numbers first, before you instruct it to add them together. This is an important concept to keep in mind as you start your coding journey.



A note from our coding mentor

Dayle

Pseudo-code is one of the most underrated tools a programmer has. It's worth getting into the habit of writing your thought process in pseudo-code in a book or a separate file before you actually begin coding. This will help you make sure your logic is sound and your program is more likely to work!

Compulsory Task 1

Follow these steps:

- Create a new text file called **pseudo.txt** inside the folder for this task in Dropbox.
- Inside **pseudo.txt**, write pseudo code for each of the following scenarios:
 - An algorithm that asks a user to enter a random letter. Repeat this until the user inputs a number. Determine how many vowels were given and output the total number of vowels.
 - An algorithm that reads an arbitrary number of integers (whole numbers (i.e. not fractional numbers) that can be positive, negative, or zero) and then returns the average of all the even numbers and the average of all the odd numbers.
 - An algorithm that reads a list of names and prints them out in alphabetical order.

Compulsory Task 2

Follow these steps:

- Create a new text file called **algorithms.txt** inside this folder.
- Inside **algorithms.txt**, write pseudocode for the following scenarios:
 - An algorithm that requests a user to input their name and then stores their name in a variable called *firstName*. Subsequently, the algorithm should print out "Hello" and the *firstName*.
 - The user then enters their age and then stores it in a variable called *age*. Subsequently, the algorithm should work out the year the person was born. The program should then print out the phrase "Based on your age of [age given], it looks like you were born in [year calculated]." E.g. If the person says they are 22 and the year is 2020, the output would be, "Based on your age of 22, it looks like you were born in 1998."

Compulsory Task 3

Follow these steps:

- We take academic integrity very seriously. Read through the additional reading on plagiarism in this task folder. Then create a file called **plagiarism.txt** and answer the following questions:
 1. What is plagiarism? Explain it in your own words.
 2. What are the four different types of plagiarism? Give an example of your own for each (i.e. not the same examples as the additional reading).
 3. How can you avoid plagiarism in your work?
- Be sure to reference any sources you use in your answers!

Optional Bonus Task

- Create a new text file called **optional_task.txt** inside this folder.
- Inside the **optional_task.txt** file write the pseudocode for the algorithm that asks the user for a number and stores that number in a variable called n . The algorithm then works out if n is a prime number or not
- The algorithm then outputs either “Prime” or “Not prime” depending on the result.

Thing(s) to look out for:

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this.



Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

