TASK

Thinking Like a Programmer — Pseudo Code II

Visit our website

Introduction

Welcome to the second Pseudo Code Task!

In this task, we will delve further into the topic of algorithms. An algorithm should follow a certain set of criteria so that it can be easily readable not only to yourself but to third parties reading your work. Clear and concise writing of algorithms is reflective of an organised mind. Hence, this task will serve as a stepping stone for you to write efficient and succinct algorithms.

Get in touch

Connect for support

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

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is 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
Hyperion Team

Congratulations on making it to the second task. You're well on your way to becoming a great programmer!

One of the greatest misconceptions about programming — certainly at introductory levels — is that programming is riddled with mathematics. If, like some people, you have an idea that programming will require going back to the days of battling with trigonometry, algebra and the like, you're wrong. There is very little mathematics involved in programming. So, if you don't exactly love mathematics, don't be discouraged!

Programming merely involves the use of logic. The ability to think things through, understand the order in which they will take place, and have a sense of how to control that flow, pervades every aspect of programming. If you have an aptitude for logic, you're going to be in a good position to start wrestling with the task of programming.

Recap on Pseudo Code

In the previous task, we covered the concept of pseudo code: a simple way of writing programming code in English. It is not an actual programming language. It just makes use of short phrases to write code for programs before you actually create it in a specific language. Once you know what the program is about and how it will function, you can use pseudo code to create statements to achieve the required results for your program.

Algorithm Design and Representation

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.

In the previous task, you designed a few algorithms for your own use. 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. It also aids in debugging if any errors arise. For example, say you have a series of calculations in your program that build off each other, it would be helpful to print out each of the programs to see if you're getting the desired result at each stage. Therefore, if a particular sequence in the calculation is incorrect, you would 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 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. Therefore, it is imperative to think of ways in which the load on your machine can be minimised.

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 what 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 very useful 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 is data or information that is transferred to the computer.

Input and Output

Input 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
    print out 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
    print out password
    and print out 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 practice 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.

Compulsory Task

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 *first_name*. Subsequently, the algorithm should print out *first_name* along with the phrase “Hello, World”.
    - An algorithm that asks a user to enter their age and then stores their age in a variable called *age*. Subsequently, the algorithm should print out “You’re old enough” if the user’s age is over or equal to 18, or print out “Almost there” if the age is equal to or over 16, but less than 18. Finally, the algorithm should print out “You’re just too young” if the user is younger than (and not equal to) 16.

Optional Bonus Task

The Fibonacci sequence is a sequence of numbers beginning with 0 and 1, in which every number after the first two can be found by adding the two numbers before it. For example, the first 10 numbers in the Fibonacci sequence are as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- 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*. Then the algorithm should calculate and print out the first *n* numbers in the Fibonacci sequence.

- You can read more about the Fibonacci sequence here: http://www.livescience.com/37470-fibonacci-sequence.html

Thing(s) to look out for:

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

Rate us

Share your thoughts

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

Think 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.