Task: Thinking Like a Programmer - Pseudo Code II

www.hyperiondev.com

Introduction

Welcome to the second Pseudo Code Task!

In this task, we will delve further into the topic of algorithms. Algorithms 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 a sophisticated mind, hence, this task will serve as a stepping stone for you to write efficient and succinct algorithms.

Connect with your mentor

CONNECT

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/support 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 dredge up all manner of faded school-day memories of trigonometry, algebra and the like, you're wrong. There is very little mathematics involved in programming. So, 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.

-The Hyperion Team

Recap on Pseudo code

In the previous task, we covered the concept of pseudo code. To recap, pseudo code is just a simple way of writing programming code in English. It is not an actual programming language, hence, it 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, then 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 central 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've 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 to 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.

Ambiguity is another criterion to take into consideration in the development of algorithms. 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 an unintended outcome.

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

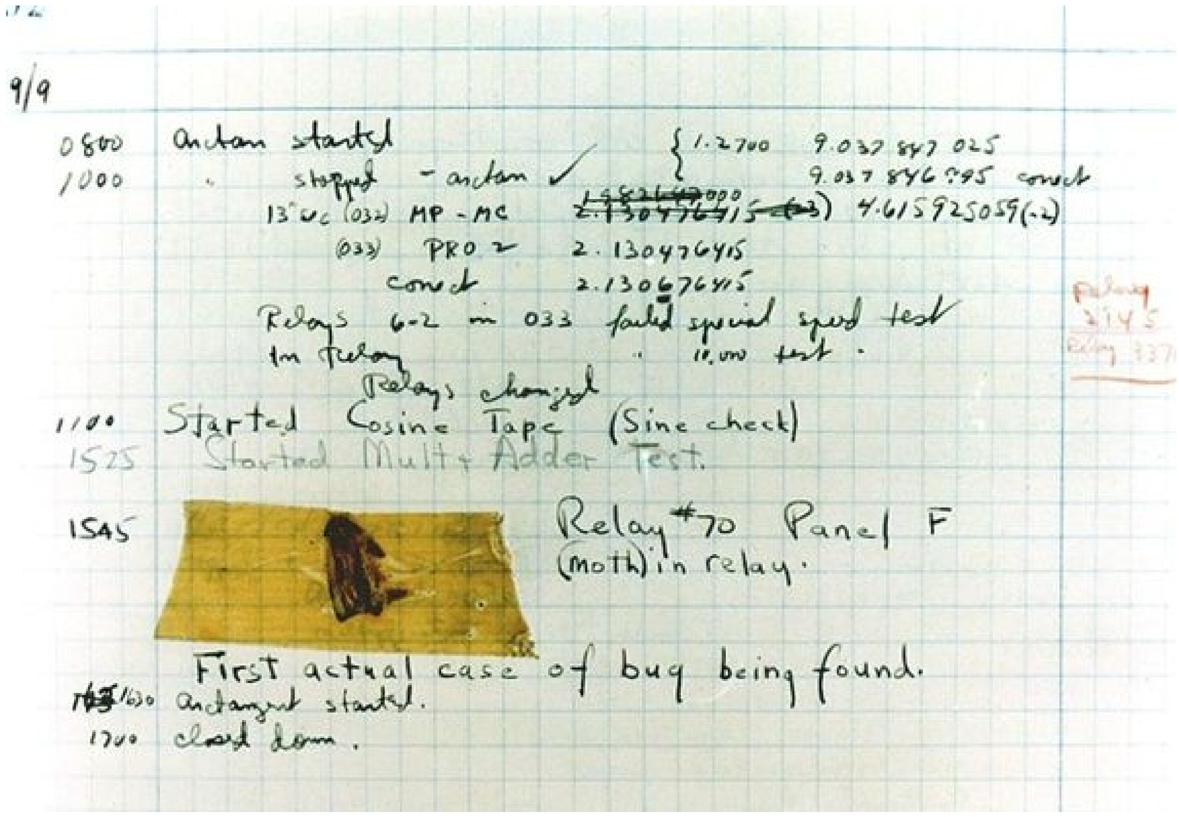
Last but not least, you should note the capacity of your resources i.e. some computing resources are finite, such as, CPU, memory etc. Some programs may require more RAM than your computer has or take too long to execute, therefore, it is imperative to think of ways in which the load on your PC can be minimised.

A note from Riaz...

Sorry to interrupt, but did you know that the first computer "bug" was named after a real bug. Yes, you read that right! While the term "bug" in the meaning of a technical error was first coined by Thomas Edison in 1878, it was only 60 years later that someone else popularised the term.

In 1947, Grace Hopper, a US Navy admiral, recorded the first computer 'bug' in her log book as she was working on a Mark II computer. A moth was discovered stuck in a relay and thereby hindering the operation. She proceeded to remove the moth, thereby 'debugging' the *system, and tape it in her log book.*

In her notes, she wrote, "First actual case of bug being found."



9/9

0800 Antan started {1.2700 9.037 847 025

1000 " stopped - antan ✓ 9.037 846 995 conect

13" UC (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (~~-3~~) 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 Antangent started.

1700 closed down.

Relay 2145 Relay 3370

- **Riaz Moola**, Founder and Managing Director

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.

Instructions

First read **example.py**, open it using Notepad (Right click the file and select 'Edit').

- example.py should help you understand some simple pseudocode. Every task will have example code to help you get started. Make sure you read all of example file and try your best to understand.

- This will take some time at first as it is slightly different to other languages, but persevere! Your mentor is here to assist you along the way.

- Feel free to write your own example code before doing this task to become more comfortable with some of the basic components.

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 the 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 check if the user's age is over or equal to 18, and print out "You're old enough" or print out "Almost there" if the age is equal and over 16, but less than 18. Finally, the program should print out "You're just too young" if the user is younger than and not equal to 16.

Things 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, but **Python or Notepad++** may not be installed correctly.
2. If you are not using Windows, please ask your mentor for alternative instructions.

Give your thoughts..

RATE

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.