Introductory Programming Exercises

Hamik Mukelyan

June 16, 2019

Variables and basic control

Exercise 1: Fibonacci numbers the easy way

The **fibonacci numbers** are $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots, F_n$, where $F_0 = F_1 = 1$. There is a **closed form** solution for F_n, which means we can find F_n by just plugging n into a formula that we only have to evaluate once. Find this formula online and write a program to compute F_5, F_{10}, and F_{20}. The fibonacci numbers are integers, so make sure that your program rounds or **truncates** your numbers correctly. After you've gotten your program to work correctly, **refactor** it - i.e., improve it - so that you store $\sqrt{5}$ in a variable. That way you don't have to perform the slow square root operation three times - you only have to perform it once at the beginning of the program then reuse that result over and over.

Exercise 2: Fibonacci numbers with a loop

Write a program that displays the fibonacci numbers from F_0 to F_n. One way to do this is with a loop that keeps a **counter variable**, i, that is **initialized** to 0 and is incremented by 1 at the end of each **iteration** of the loop until it reaches n. The variable names i, j, and k are traditionally used as loop counter variables, with i being used in outermost loops and j being used in loops nested within those outermost loops. For each iteration of the loop you plug i into the formula from exercise 1 and display the result with a print statement. Try it for $n = 10$ and see if the correct numbers show up!

Now let's refactor. The closed form solution for F_n uses divisions and exponentiations in addition to square roots. We mitigated the slowness of square root operations by storing $\sqrt{5}$ in a variable, but we still do divisions and exponentiations, which are super slow compared to simple additions. The **recursive formula** for F_n elegantly (and simply) describes the sequence in terms of the numbers that came before it: $F_n = F_{n-1} + F_{n-2}$. Along with the **base cases** $F_0 = F_1 = 0$, this formula fully defines the sequence. Switch your program over from using the closed form solution to the recursive one. Recursive solutions - that is, solutions that are stated in terms of *smaller or easier versions of the*

same problem, show up often in computer science because closed form solutions are often difficult or impossible to find.

Exercise 3: The radix

When we write a number like 987, we actually mean $9 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0$. The number 10 is called the **base** or **radix** of the number. Most civilizations have used the radix 10 for counting, but some have used weird ones like 60 - we'll see in a moment how to think about numbers in a non-standard base. **Binary** (base 2), **octal** (base 8), and **hexadecimal** (base 16) are the most commonly used bases in programming other than decimal (base 10). If a base B is less than 10, we use only the digits 0 to $B - 1$. For example, in octal we never use the digits 8 or 9. The number $123_{10} = 1 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 = 173_8$, where the subscripts are reminders about the bases of the numbers. When the base is 10 we usually omit the subscript by writing e.g. 123 instead of 123_{10}.

In bases greater than 10 we use letters starting from A for digits greater than 9. For example, in hexadecimal the digit A is one greater than 9, B is two greater than 9, and F is the biggest digit, which represents 15_{10}. So counting from 1 to 32 in base 16, or hexadecimal, looks like 1, 2, 3, $\cdots$, 9, A, B, C, D, E, F, 10, 11, 12, 13, $\cdots$, 1E, 1F, 20. In binary we have only the digits 0 and 1 at our disposal. Any number that can be represented in decimal can be represented in any other base. Computers store numbers and operate on numbers in binary for the simple reason that it's easy to represent them with hardware: it's much easier for a switch to be simply on or off - representing 1 or 0 - than it is for it to be in one of ten states.

Binary has a special relationship with octal and hexadecimal, which is why they're used frequently in programming. Binary numbers are annoyingly long, so we can represent them more compactly by using their relationship with hex. Write out the first 16 numbers in binary. E.g., 0, 1, 10, 11, 100, 101, $\cdots$. These numbers match up, respectively, with the hex numbers 0, 1, 2, 3, $\cdots$, E, F. If you want to write a long binary number like 111100101010_2 compactly, you break it up into 4-digit chunks and use the equivalent hex number for each chunk. E.g., $1111\ 0010\ 1010_2 = \mathrm{F2A}_{16} = 15 \cdot 16^2 + 2 \cdot 16^1 + 10 \cdot 16^0 = 3882$.

In many programming languages, you can represent a binary, octal, or hexadecimal number by prefixing any numerical literal with 0b, 0, or 0x. For instance, if you want to double-check that $\mathrm{F2A}_{16}$ is actually 3882, just type 0xF2A into a REPL and see the resulting decimal value. Similarly, you can double-check that $173_8 = 123$ by typing 0123. These prefixed numerical literals can used anywhere you'd use a typical decimal literal in a program.

Here's the exercise: prompt the user for n_{10} then display it in base 3. Then prompt the user for m_3 and display it in base 10. Before you try this problem, make sure you can convert a base 10 number into base 3 by hand and vice versa. Next make sure you can clearly articulate the steps you took to do so: in other words, devise an **algorithm**, which is a step-by-step problem solving procedure.