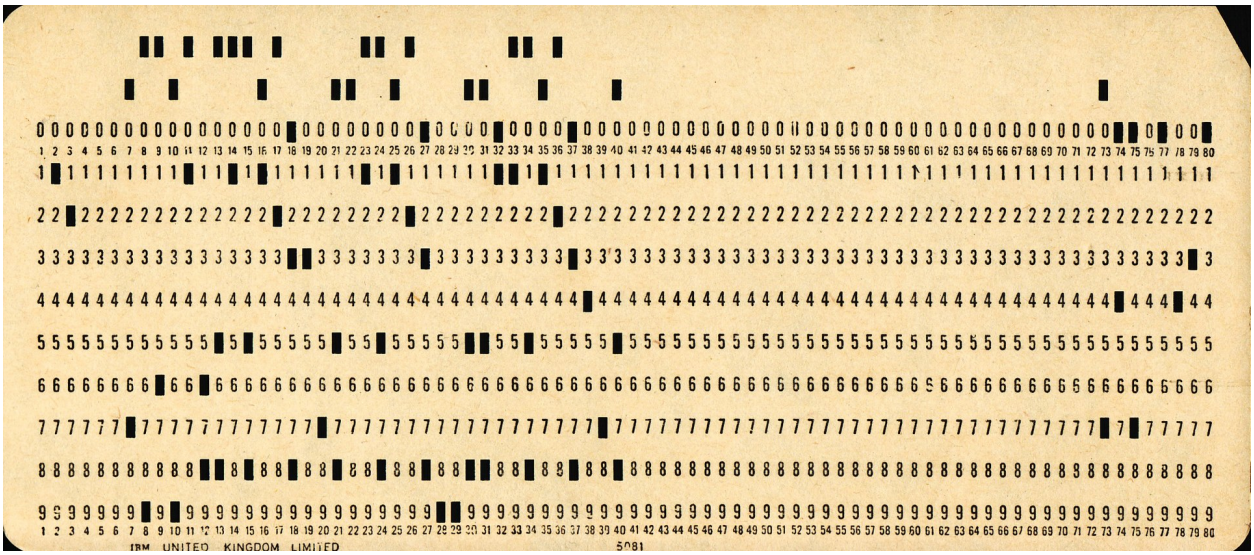
Pseudo-code

Origin of Pseudo-code

Blast to the Past

Welcome Time Traveler! The year is 1928 and IBM has just unveiled the latest in programming technology, the IBM100 also know as the “IBM Card”



IBM 100

The IBM Card is an example of “punch card” computing. Software developers in 1928 had to manually “punch” the data they wanted computers to store and read.

But what does the above example mean? As someone who does not read IBM 100 style code I am left clueless what this “program” does. How can I, as a stakeholder in a project, know what this developer created? This is where Pseudo-code comes in handy. Lets travel back to the future to elaborate on this further.

Welcome Back Marty

Well, we sure have come a long way since the 1930’s when it comes to program readability. Take a look at a simple code snippet below:

```
a = 1
b = 2
print(a+b)
```

It doesn’t take a developer to make an accurate assumption about what the code snippet does. But what about for larger more complicated programs? Or, what if we don’t have any code in front of us? How do we communicate the steps our current (or future) program is performing to those around us?

Enter Pseudo-code

According to the Oxford Dictionary, Pseudo-code is defined as “a notation resembling a simplified programming language, used in program design”. **Pseudo-code is essentially a language agnostic technique of communicating logic.** Since Pseudo-code is “language agnostic” we can pick whatever language we want (English, Spanish, Pig Latin), or phrased differently, whichever language will most clearly communicate our message. No longer are we stuck talking in punch cards, Python, or binary when talking about computers, but with a language like English that is much more familiar in our everyday lives.

Pseudo-coding in Interviews

Goal of Pseudo-coding

Why is showcasing your ability to pseudo-code important during interviews? Programming languages, frameworks, and tools rise and fall over careers but the foundational understanding of how computers perform tasks tend to stay the same. Not only that but Pseudo-coding displays your ability to communicate complex software developer concepts to other stakeholders, who may not understand the programming language you are using.

As we move through the rest of this section, we are going to keep the key goal of clearly **showcasing and communicating software developer concepts** close by.

Key Steps

💡 Problem: Write a function that combines two arrays by alternatingly taking elements from each array in turn.

[a, b, c, d, e], [1, 2, 3, 4, 5] becomes [a, 1, b, 2, c, 3, d, 4, e, 5]

[1, 2, 3], [a, b, c, d, e, f] becomes [1, a, 2, b, 3, c, d, e, f]

Points:
1. The arrays may be of different lengths, with at least one character/digit.
2. One array will be of string characters (in lower case, a-z), a second of integers (all positive starting at 1).

1. Define Function Inputs/Outputs

During interviews we are normally asked to solve a specific problem in the form of writing a function in a programming language. If we think about what a function does at a base level, functions take inputs in order to produces an output. Since our function is reliant on it's input and outputs, it only makes sense to start there while Pseudo-coding.

The key information we want to know about **inputs** is what is the possible range of values we can get. Are they all numbers, arrays, a mixture of both? This step helps identify if we will need any special cases or error handling. We also what to flesh out any edge cases that could appear, such as empty lists, floating point numbers, etc. Asking these questions help **showcase** your understanding of specific programming languages and **software developer concepts**. Beyond that, defining our inputs also helps us brainstorm any potentially problematic inputs that might trip us up down the line.

The key information we want to know about **outputs** is what are we returning? If the question asks for a number and we return an array then not only will never achieve the correct answer, but it shows a lack of critically reading the problem, which could of been easily avoided.

```
// Inputs: Two Arrays of a combination of type Number and String
// Output: One Array of a combination of type Number and String
```

2. Name things appropriately

Function and variable names should always be descriptive. Don't call the function func as that doesn't tell the reader anything about what the function might do. The same goes for all of your inputs, outputs, and other variables. Don't call your input, "input" if you know it has to be an interger. Try to be specific without being overly wordy. Again, the goal is to be descriptive with your naming choices.

```
// Inputs: Two Arrays of a combination of type Number and String
// Output: One Array of a combination of type Number and String

function mergeArrays(arr1, arr2){

  return mergedArr
}
```

3. Work through the Problem

Before we can start coding (pseudo-code or real code) we need logic around how to solve this problem. Using just sentences, work through the problem. How can we go from the input, to the desired output? What do you want this function to do? Do you need to create any additional variables to help you solve these problems? This is the fun part about solving problems! You don't have to worry the nitty gritty of solving problem, just the main concepts and ideas. Give it a try and list out the steps to solve our example problem.

▼ Example

💡 **Here's one of the many possible ways to solve this problem**

1. If there is an element in the first position from the first input array, add it to an output array
2. If there is an element in the first position from the second input array, add it to an output arrary
3. Repeat steps 1 and 2 with the next elements of the input arrays until there are no more elements in both input arrays to be added

Congratulations! You've written your first go at pseudo-code. But what makes good pseudo-code? Let's refer back to the goal we set out earlier: **showcasing and communicating software developer concepts.**

Looking at the example written above, did we achieve that goal? One could argue that we accurately communicated the logic to solve the problem, but we didn't **showcase** any **software developer concepts.** This would be great pseudo-code to communicate to non-technical stakeholders, but to technical interviewers we want to show-off a bit. At this point you're probably wondering what **software developer concepts** is.

4. Use General Software Developer Concepts

Although every coding language has it's own syntax, keywords, and built-in methods, nearly all programming languages today have constructs that span them all. These concepts are programming constructs are: **sequence (order)**, **selection (conditions)**, and **iteration (looping)**.

Let's take a deeper dive into **selection** and **iterations**:

Selections determine which path the code to take while it is running. These are more commonally known as conditions. Most modern programming lagnuages use some sort of **If / If Else / Else** keywords. Those three keywords are perfect to use while you are writing up your pseudo-code.

Iteration determines if piece of code should be executed more than one time. This is more commonally known as looping. Most modern programming lagnuages use some sort of **For / While** keywords. **For** is normally used when you know exactly how many times you want to loop, **While** is used when you don't know how many times you want to loop. Use the **For / While** keywords will pseudo-coding and as a bonus talk about why you used one over the other.

Some other more general software deleloper concepts that we should really consider are **indentation**. While only some languages require white space as part of their syntax, most languages support indentation for readability. Proper indenation shows that you can write readable code with commmon conventions.

Lastly, sprinkle in common software developer terminology like **initialize, index, nodes, constants, pointers, ect.** (when applicable).

▼ Example

💡 **Here's an example of better pseudo-code**

Initialize an empty array to hold the output
For each index in the largest array
    Access the element at that index in the first array
    If that element exist
        Add it to the output array
    Access the element at that index in the second array
    If that element exist
        Add it to the output array

5. Sprinkle in Language Specific Knowledge into Pseudo-Code

For our next step we can even convert some of the our general pseudo-code into language specific code! We can add in the syntax we are interviewing with and even use language specific method keywords. What's nice about doing this step is we can easily and quickly covert this stage of pseudo-code into actual code and not worry about writing

▼ Example

```
// Inputs: Two Arrays of a combination of type Number and String
// Output: Array of a combination of type Number and String

function mergeArrays(arr1, arr2){

// Initialize an empty array to hold the output
// For each index in the largest array
//    Access the element at that index in the first array
//    If that element exists, Add it to the output array
//    Access the element at that index in the second array
//    If that element exists, Add it to the output array

  const mergedArr = []

  for(let i = 0; i < length of largest array; i += 1){
    if(arr1[i] exists) push arr1[i] to mergedArr
    if(arr2[i] exists) push arr2[i] to mergedArr
```

```
  }

  return mergedArr
}
```

Recap

And we’re done! As you can see, pseudo-code can resemble actual code! Now our reader can understand what you’re doing, what loops you’re planning on using, what your conditional statements are and what the function is supposed to ultimately do! The reader can very easily translate this into working code.

Now this is just a simple example, pseudo-code is a lot more useful for working through very complex solutions. Like the one below:

Problem:
Write a recursive function that returns an array of all possible outcomes from flipping a coin N times.

If we have 3 coin flips, the final solution would be:
[’HHH’, ‘HHT’, ‘HTH’, ‘HTT’, ‘TTT’, ‘TTH’, ‘THT’, ‘THH’]

Try to work through the above problem by yourself using the ideas covered in this guide. Once your done, try to evaluate how you did. Did you define the inputs and outputs? How about your logic. Did you include every step or condition? Did you use common software development concepts? Common keywords? Lastly, how did you do **showcasing and communicating software developer concepts?** Remember, there are always was to **showcase and communicate software developer concepts** beyond just psuedo-coding. If your pseudo-code missed any key ideas, would you have addressed it elsewhere (in your verbal comments or in your actual writing of the code)? Pseudo-coding is a great place to start but is never the be all end all when it comes to interviewing.

▼ Example Solution

```
// Inputs: 1. An Integer for the number of coin flips, 2. An Array of Strings that w
e'll pass through with each iteration, at the beginning it'll be empty
```

```
// Outputs: An Array of Strings that contains all possible of combinations 'H' and 'T'
for the number of flips

function coinFlips(numFlips, arr = []){
  // Base Case
  if(no more flips) return arr

  const arrNewFlip = []

  if(arr is empty because its the first iteration) push 'H' and 'T' to arrNewFlip
  else {
    const arrHeads = map an 'H' onto every element of arr
    const arrTails = map a 'T' onto every element of arr
    push both arrHeads and arrTails into arrNewFlip
  }

  return coinFlips(numFlips - 1, arrNewFlip)
}

// If the problem is recursive, we should walk through the steps and show the reader w
hat the array will look like after each flip. Let's again go with numFlips = 3

// Start: numFlips = 3 , arr = []
// After First Iteration: numFlips = 2, arr = ['H', 'T']
// After Second Iteration: numFlips = 1, arr = ['HH', 'TH', 'HT', 'TT']
// After Third Iteration: numFlips = 0, arr = ['HHH', 'THH', 'HTH', 'TTH', 'HHT', 'TH
T', 'HTH', 'TTT']
// numFLips = 0 hits our base case and so it exits out leaving us with that final arr
 as the solution
```

Examples in Coding Literature

Here are some additonal examples of Pseudo-code I found from a couple coding books. As you look through the various examples, try to ask yourself what you liked about them and what you didn't. Much like writing, you'll develop your own style that you gravitate to. As long as you focus on the key ideas in this guide, you really can't go wrong.

NearestNeighbor(P)
    Pick and visit an initial point p_0 from P
    $p = p_0$
    $i = 0$
    While there are still unvisited points
        $i = i + 1$
        Select p_i to be the closest unvisited point to p_{i-1}
        Visit p_i
    Return to p_0 from p_{n-1}

`The Algorithm Design Manual` by Steven Skiena

The pseudocode below implements DFS.

```
1   void search(Node root) {
2      if (root == null) return;
3      visit(root);
4      root.visited = true;
5      for each (Node n in root.adjacent) {
6         if (n.visited == false) {
7            search(n);
8         }
9      }
10  }
```

`Cracking the Coding Interview` by Gayle Laakmann McDowell

Authors

Article written by Nathaniel Morgan and Damian Bzdyra