Problem Solving for Intro to Computer Science

The purpose of this document is to review some principles for problem solving that are relevant to Intro to Computer Science course.

Introduction:

By working through the lectures and quizzes in *Introduction to Computer Science*, you will accomplish several tasks more or less simultaneously: learn the syntax of Python, understand how to use programming control structures to create programs, and use your new-found knowledge to solve problems. While it is natural to absorb the first two tasks through observation and practice, it is sometimes difficult to develop a systematic approach to problem solving.

One of the classic texts in problem solving is George Polya's "How to Solve It". In this document we recall the classic problem solving principles from this book and discuss how to apply the steps to a programming problem similar to those you will encounter in this course.

A Sample Problem and Polya's Principles:

For the purposes of illustration, we will walk through this document while solving the following problem:

```
# Define a procedure, find_primes, that takes
# as an argument a list of numbers and returns
# a list consisting of all prime numbers from
```

```
# the first list.
```

Polya's four principles are the following:

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan.
4. Look back, reflect, improve.

Step One - Understand the Problem:

It is this step that is the source of most error when first learning to write code. The goal of this step is to determine how all the parts of the problem relate to each other in order to find a connection between the data and the solution. We accomplish this by asking and answering numerous questions about the problem, such as:

- What are we asked to do?
- What are we asked to find?
- What do we need to show?
- Have you seen a similar problem before?
- Can you restate the problem in your own words?
- What are the inputs that we are given?
- What output is expected?
- What type(s) is the inputs (i.e., are they `lists`, `ints`, `dictionary`, etc.)?
- What type(s) is the output?
- Can you change the problem to a related, but easier one?
- Can you think of a related but more general problem?
- Can you draw a picture or diagram?
- Did you use all of the data?
- Is all of the data required to solve the problem?
- etc.

Let's restate the problem from above:

Problem: We will be given a list of numbers as inputs. We want to return a list with all the prime numbers from the original list.

What is meant by numbers? Will these be just integers, or could our list look like `[.5, 2, 7, -6]` ? Since not stated otherwise, let's assume that the list could have positive, negative and decimal numbers (`floats`).

Do we remember what the definition of a prime number is? In case you have forgotten, a number is prime if it is an integer bigger than one and if its only positive divisors are itself and 1.

What is our input? Looks like it will be a list. The output will be a list also.

So, it seems like we have a good idea how the input and output are related, so we can move to step two.

Step Two - Devise a Plan:

Now that we have a good idea what the problem is asking, and how the input and output are related, we can devise a plan for our solution.

If we were to do this by hand (i.e. if the input list was on paper), we would probably go through the input list one item at a time, and check if the item is a prime number. If the item is prime, then we might circle the item, or underline it to set it apart. Another thing we could do would be to add the item to another list, so we could have all the prime numbers in one place when we finish. This seems like a good way to do it, so we will try this out first.

By the way, how do we check if something is prime? Well, we might just remember which numbers are prime some of the time, but this isn't practical in general. We could use the definition from above: a number is prime if is an integer, and the only positive divisors of are and . Great! We could just check if the number is a positive integer, and then check if is divisible by all numbers in the range to . Note that we don't need to check divisibility by or , since this should be true for every number.

By the way, how do we check if a number is an integer? In python, we can convert a number to an `int`, by simply using the `int()` function. This will truncate any decimal places on the number, so one thing we might do is check if

```
our_number == int(our_number).
```

It should be noted that this will only work for numbers up to a certain size, but that is outside the scope of this course.

So let's recap our plan:

1. Iterate over the numbers in the number list.
2. Check if each number is prime.
    a. Do this by checking if the number is an integer bigger than , and checking divisibility for all numbers between and .
3. If the number is prime, add the number to our output list.

Great! Next we get write some code!

Step Three - Carry Out the Plan:

Our plan from above is nearly pseudocode, so this part shouldn't be too difficult in our case. Let's begin by writing a function stub. This is a function that doesn't yet have the functionality that we want, but at least runs correctly. It is essentially a shell of a function. Our stub will be the following:

```
def find_primes(number_list):
    prime_list = []
    return prime_list
```

Ok, now that we have a stub of a function, we can start with the steps in our plan. Remember that the first step is to iterate over the numbers in number_list. Eventually, we will want to append the prime numbers to prime_list, but for now, we can skip checking if the numbers are prime, and just get the first loop working:

```
def find_primes(number_list):
    prime_list = []
    for n in number_list:
        if True:
            prime_list.append(n)
    return prime_list
```

Of course, right now, every number in number_list gets appended to prime_list, so we will want to replace the 'True' from the procedure above with a condition that checks if a number is prime. Let's try to write a helper procedure that does this. We may not have expected to write a helper procedure in the original plan, but we can follow the same first two of Polya's principles to solve it. We won't go into the

details, but it is important to note that our input will be a number, and the output should be a boolean. Below, we use the definition of a prime number to outline the helper function:

```
def is_prime(number):
    if (number <= 1) or (number != int(number)):
        return False
    for i in range(2, number):
        if number%i == 0:
            return False
    return True
```

Note that the first 'if' statement checks that the number is an integer bigger than one. The for loop checks to see if the number is divisible by any number between one and itself. It does this by checking for a zero remainder with the mod operator % (if you don't know this one, please take the time to Google it -- it will be really useful).

Finally, we can use our new helper procedure to finish the program.

```
def find_primes(number_list):
    prime_list = []
    for n in number_list:
        if is_prime(n):
            prime_list.append(n)
    return prime_list
```

Step Four - Look Back, Reflect and Improve:

It is crucial not to skip this step! Often students are happy to get code working and move on before checking if the code is correct or looking to see if it can be improved. Things to ask at this stage include:

- Are you sure that the code works as intended?
- Is there any way to check the code?
- Could you improve the solution in any way?
- Will this function be useful for other problems?

It turns out that to check if a positive integer is prime, you only need to check if it is divisible by numbers between and . Maybe we could

change our `is_prime` procedure to take this into account? Can you think of any other ways to improve the procedures?

If you like this example, and are interested in seeing the principles discussed in greater depth and with other problems, Polya's book is an excellent place to look. Here's a link to a place where you can buy the book. Polya's book is written for mathematicians, but the problem solving techniques used by computer scientists are often very similar.

Practice Problems:

Here are a couple of practice problems for you. Try to use Polya's four principles as you solve the problems.

```
# Define a procedure, sum_evens, that takes
# as an argument a list of numbers, and returns
# the sum of all the even integers from the list
#
# For example,
# sum_evens([1, 2, 3, 4, 5]) = 6
# sum_evens([7, 21, -9]) = 0
# sum_evens([0.5, 4]) = 4
```

```
# Define a procedure, how_long_til_lunch, that takes
# three inputs: an integer representing the current hour,
# an integer representing the current minutes, and
# a string that is either 'am' or 'pm'.
#
# The output should be how long you have to wait from the
# given time until 11:45 am, in minutes (an integer).
#
# If you're unsure how the 12-hour clock works, see
# http://en.wikipedia.org/wiki/12-hour_clock
#
# For example, how_long_til_lunch(5, 45, 'pm') =  1080
```

```
# Define a procedure, string_split, that takes
# as an argument a list of strings, and returns
# a list of two elements: the first element is the
```

```
# concatenation of all the strings in the first half
# of the list, and the second element is the
# concatenation of all the strings in the second half
# of the list. If the input list is odd length, add the
# middle string to the first element.
#
# For example,
# string_split(['Hello',  ' world!',
# 'Goodnight', ' moon!']) =
# ['Hello world!', 'Goodnight moon!']
#
# string_split(['one', 'two', 'three']) =
# ['onetwo', 'three']
#
# string_split([]) = ['', '']
```