Week 10

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. By the Assessed Cohort Meeting, you and all of your cohortmates, should be prepared to present and discuss solutions to all of the assigned problems. In addition to discussing with your cohortmates, you may discuss the problems with any other current CS3102 students you want, and use any resources you want except for any materials from previous offerings of this course or complete solutions that might be available on the web, which are not permitted. **At the end of your assessed cohort meeting, your Cohort Coach will assign one of these problems as a writeup. You may not collaborate on this writeup with your cohort-mates, but you may use notes taken before or during your assessed cohort meeting.**

Problem 1: Understanding Reductions

Review the *Proof By Reduction* video. Be able to answer questions such as:

- How did we use our knowledge that *ACCEPTS* is uncomputable to demonstrate *HALTS* is uncomputable?
- When using reductions to show a function to be uncomputable, what properties must the reduction itself have?
- If function A is computable, and function B is uncomputable, can I reduce function A to function B? What about function B to function A?
- Which of the following (if true) would allow me to conclude Nate Brunelle will never be good at football: *Nate will never be good at basketball and anyone good at basketball can be good at football* OR *Nate will never be good at basketball and anyone good at football can be good at basketball.*

Problem 2: Entranced By Re-Entrances

Use a proof by reduction to show that the function $REENTERS$ defined below is uncomputable.

> **Input:** A string w that describes a Turing Machine.
>
> **Output:** **1** if the machine described by w would re-enter its start state when executed on the input ε. Otherwise, **0**.

That is, a machine which computes $REENTERS$ outputs **1** when the input describes a Turing Machine which, when run with the input ε, enters the start state as a result of some transition.

(Note: For this problem, we want to see that you understand how to do a reduction proof, so even if you can prove it using some other method, you should also be able to explain how to prove it using the proof by reduction method.)

Problem 3: Maximum Recursion Depth

Python has a default recursion depth of 1000 calls.[1] This means that if ever we have a function which calls itself over 1000 times (nested), Python will throw a `RecursionError` exception.

Show that, in general, the problem of determining whether a given Python program will exceed the maximum recursion depth when running on a given input is not computable.

For this problem, you should assume an “idealized” version of Python with no other implementation limits, other than the setting for recursion depth. In other words, show that the language of Python, input pairs such that the Python program has no more than 1000 nested recursive calls is not computable.

[1] This has lead many dysfunctional programmers to conclude that "recursion is bad" and should be avoided, even though recursive definitions are often the most elegant and clear way to express many functions. The real problem is that most versions of Python have implementations of recursive function calls that do not do tail call optimizations that are done by more functional programming languages (this means that a recursive function execution requires $\Theta(n)$ stack space in Python where n is the number of recursive calls, whereas a "correct" implementation would only use $O(1)$ stack space). This has lead to the recursively bad situation where Python programmers are trained to think recursion is bad, so Python language implementers don't think it is important to make tail recursive functions perform well, which means Python programmers continue to correctly learn that in Python recursive function definitions are "bad".

For example, calling the `rec(x)` function defined below with input `x = 500` would terminate happily, but on input `x = 1001` would result in a recursion depth exceeded error.

```
def rec(x):
    if x > 0:
        return rec(x-1)
    else:
        return "Done"
```

Problem 4: Rice's Theorem

Recall from the *Rice's Theorem* videothat any non-trivial semantic property of a Turing Machine is uncomputable. A property of Turing Machines is semantic if its truth/falsehood will match for any two Machines which compute the same function. A semantic property is trivial if it is true for all Turing Machines, or else false for all Turing Machines.

For each subproblem, indicate whether or not Rice's Theorem applies. If it applies, explain why, and answer if the problem is computable or uncomputable. If it does not apply, just indicate why it doesn't apply (it is not necessary to determine whether or not it is computable if Rice's theorem does not apply).

(a) Given the description of a Turing Machine, does that machine always return 0?

(b) Given the description of a Turing Machine, does that machine always return 1 when it receives no input?

(c) Given the description of a Turing Machine, does that machine ever use more than 3,102 cells on its tape?

(d) Given the description of a Turing Machine, is the language of that machine recognizable?

(e) Given the description of a Turing Machine, does that machine have exactly 50 states?

(f) Given the description of a Turing Machine, does that machine produce an output that is the correct answer to the question "Will UVA Basketball win the National Championship in 2022?" when it receives no input?

(g) For a fixed way of describing Turing machines, does the string 11010011110001010001001001110010101 describe a Turing machine which accepts 101?

Problem 5: Code Smells

The *Rice's Theorem* video talks about Turing Machines not having smells because they are mathematical objects. Programs are mathematical objects too, but some programs still emit pungent odors. These "code smells" are signs that something isn't quite right with the code, or that it will be hard for others to understand or modify without breaking things. As John Woods once write, "Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live."

This means that good programmers try to make what it's doing as clear as possible so that the future maintainer (who may actually be a future version of yourself who has forgotten how the code works) can understand, modify, and debug the code. Code smells are a sign that code will be poorly received by the violent psychopath maintainer.

There are tools that check programs for certain code smells, such as *SonarSource*. These tools can scan code and report various issues that are potential problems. Below is a list of code smells detected by SonarSource (each is a link to a more detailed description of what is detected, including an example, but you can answer based on just understanding what is written here). For each code small listed, explain whether it is computable or uncomputable to detect the smell in general (and assuming an idealized programming language).

(a) *Loops should not be infinite*

(b) *Function returns should not be invariant* (there should not be multiple return statements in a function, which always return the same value)

(c) *Unread "private" attributes should be removed* (there should not be private attributes in a class definition that are never read)

(d) *String literals should not be duplicated* (the same string literal should not occur multiple times in the code)

(e) *Expressions creating sets should not have duplicate values* (an expression that creates a set should not contain multiple occurrences of the same value)

(f) Given that some of these properties are definitely uncomputable, how can a company sell a product that claims to detect them? Is this product a fraud, or is it potentially useful to attempt to detect these code smells? What do we know about the accuracy of the detection for the properties that are uncomputable?

(g) For your favorite (or least favorite) programming language supported by Sonar ((see list of languages at right side of this page) or some other static analysis tool you find, identify a property it claims to check that is uncomputable, and explain why it might be able to do something useful for this even though it is uncomputable in general.

Problem 6: Can we avoid the Halting Problem?

We showed in HALTS is Undecideablethat Turing Machines suffer from a "Halting Problem". I.e., it is impossible to design a Turing Machine which can determine whether a given Turing Machine will halt when running on a particular input. Since the idealized Python computing model is equivalent to the Turing Machine model, Python must have this "Halting Problem" as well (i.e. it is impossible to write a python program that can tell whether an arbitrary python program will halt on some input). One might wonder, however, if the presence of a "Halting Problem" for a programming language is absolutely necessary, or if it is merely a flaw in the language's design. Consider these restrictions my could add to Python so that infinite loops are always detectable (we consider programs which throw errors as halting):

– All `while` loops must be `while(True)` loops

– All other loops must be `for` loops

– `break` and `return` cannot appear in the body of a `while(True)` loop

– All recursive calls (even indirect) cause a runtime error

The idea here is that we add rules to Python so that a program will run forever if and only if it has a `while(True)` loop[2]. Importantly, our programs can run forever, but it will be easy for us to tell by just looking for `while(True)`. We would, next, want to consider whether we could still implement every computable function using this restricted Python. Towards this end, answer the following questions:

a) Review Week 9 problem 2 Self-Reject. Are we still able to implement `self_rejecting_py` using our restricted Python? If so, what happens when we try to invoke `self_rejecting_py` on its own source?

b) Next show that any programming language which does not have a "Halting Problem" is a weaker computing model than a Turing Machine. To do this, we'll say that the function H_ℓ represents the "Halts Function" for programming language ℓ (e.g. our restricted Python), i.e. $H_\ell(p,x) = 1$ if program p written in language ℓ halts on input x, and $H_\ell(p,x) = 0$ otherwise. Show that if there exists a Turing Machine which solves H_ℓ then programming language ℓ cannot be used to implement a Universal Turing Machine.

c) Name a programming language actually used in industry for which H_ℓ is computable. What is it used for?

[2] I have not formally shown that there are no other ways to get infinite behavior with Python, but if there are, suppose we have additional rules above to disallow those as well.

