Week 10: Have we been reduced to this?

Collaboration Policy: You should work on the problems yourself, before discussing with others, and with your cohorts are 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 (including the programming problems). In addition to discussing with your cohortmates, you may discuss the problems with anyone you want, and use any resources you want except for any materials from previous offerings of this course, which are not permitted. You should document any resources you use (beyond the provided course materials) in your problem write-up.

Problem 1 *Deciders Vs. Recognizers*

In the *Decidable, Recognizable, Computable* video we discussed that a language is *decided* by an Accept/Reject Turing machine if that machine halts and accepts all strings that belong to the language, and halts and rejects all strings do not belong to that language. A language is *recognized* by a Turing Machine if that machine halts and accepts all strings in the language, and for all strings not in the language the machine either rejects or never halts.

Prove that the $ACCEPTS$ language (as defined in Week 9) is *recognizable* by describing a Turing Machine that recognizes it.

Problem 2 *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 3 *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 4 *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.

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"
```

[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".

Problem 5 *Rice's Theorem*

Recall from the *Rice's Theorem* video that 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 there be a Covid vaccine available before Feb 1, 2021?" when it receives no input?

(g) For a fixed way of describing Turing machines, does the string 1101001111000101000100100111001001 01 describe a Turing machine which accepts 101?

Problem 6 *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 7 *Bonus Points* ($\star$?)

Consider the language:

$$L = \begin{cases} \emptyset & \text{If you get this problem correct,} \\ \{\varepsilon\} & \text{otherwise.} \end{cases}$$

The number of bonus points you will earn on this question is equal to the cardinality of L. You will receive 1 bonus point for a correct answer and 0 for an incorrect answer.

Does NFA M compute L where $M = (\{a\}, \{0,1\}, \emptyset, a, \emptyset)$?

I hope everyone realized that my previous remark about non-use of joke markers was a joke, and was flagged as such by the absence of a marker. This message is not a joke, as indicated by the exclamation point.
Guy Steele (discussion that introduced the :-) emoticon, 1982))

