PS10: More Uncomputability

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. 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. Sharing and subsequently submitting any text, code, images, figures, etc. constitutes plagiarism, so make sure all submitted materials are created exclusively by members of your cohort.

Problem 1: Return of Maximum Recursion Depth

This Question was moved from PS9. If you completed it there, feel free to simply copy-paste your solution here.

Python has a default recursion depth limit 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.

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 `RecursionError` exception.

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

Using a reduction proof, show that, in general, the problem of determining whether a given Python program will exceed the maximum recursion depth is not computable. That is, show that there does not exist an always-halting Turing Machine which, when given a python program, returns 0 if the program does not throw a `RecursionError` exception, and will return 1 if it does. For example, if we followed the above code with the line `rec(1001)` then the Turing machine should return 1 when given that program as input. If we followed the above code with the line `rec(500)` then the Turing machine should return 0 when given that program as input.

For this problem, you should assume an “idealized” version of Python with no other implementation limits, other than the setting for recursion depth.

[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 2: Rice's Theorem

Rice's theorem states 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 the input ε?

(c) Given the description of a Turing Machine, does that machine use more than 3,102 cells on its tape when it receives the input 1011?

(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) Does the string $110100111100010100010010011100100101$ describe a Turing machine which rejects 101?

Problem 3: ACCEPTS in k Steps

Consider the Language:

$$A_k = \{w | \text{ is the description of a Turing Machine and } \mathcal{M}(w)(\varepsilon) \text{ halts and returns 1 in } k \text{ or fewer steps } \}.$$

Show that A_k is computable for every choice of $k \in \mathbb{N}$.

In other words, show that for every choice of $k \in \mathbb{N}$ there exists a Turing machine which computes the function $A_k : \{0,1\}^* \to \{0,1\}$ where $A_k(w) = 1$ provided $\mathcal{M}(w)(\varepsilon)$ halts and returns 1 in k or fewer steps.

Problem 4: A Lot of Unions

If two languages L_1 and L_2 are computable, then $L_1 \cup L_2$ is also computable. If you don't understand why this is the case, then first convince yourself. Observe that this means that the union of finitely many computable languages will be computable. Next, you will prove that this is not the case for an infinite union using the A_{anyK} language below.

Definition 1 **(A_{anyK})** *Define the language* $A_{anyK} = \bigcup_{k \in \mathbb{N}} A_k$.

That is, $A_{anyK} = A_0 \cup A_1 \cup A_2 \cup \ldots$

Show that, even though it is constructed by unioning only computable languages, A_{anyK} is not computable. (Hint: you should find it to be very similar to an uncomputable language you have seen.)