MODELOS AVANZADOS DE COMPUTACIÓN

Práctica Python

Para realizar antes de empezar

1. Python está disponible en la imagen de Ubuntu de los ordenadores de las aulas de informática. Si se quiere usar un ordenador propio, hay que tener instalado Python y se recomienda un entorno como Spyder (`https://www.spyder-ide.org/`)

2. Descargar programas en Python del libro *What Can Be Computed? A Practical Guide to the Theory of Computation* (John MacCormick) disponibles en la página: `https://whatcanbecomputed.com/` Comprobar que se pueden ejecutar.

Para realizar en la sesión

- Ejecuta un programa como `containsGAGA(x)`, que resuelve el problema de determinar si una palabra de entrada contiene la subcadena 'GAGA'. Ejecuta la función `testContainsGAGA()`.

- Ejecuta y prueba el programa `isEmpty(x)`.

- Ejecuta y prueba el programa `countLines(x)`. Pruébalo sobre otro programa:
  `countLines(rf('containsGAGA.py')`.
  `rf` es una función que lee el contenido de un fichero y lo pone en una palabra.

  Observación: Un programa puede ejecutarse con otro programa como entrada y proporcionar alguna información sobre el mismo (recordar también la práctica de `lex`, de la asignatura Modelos de Computación)

- Prueba y ejecuta el programa `loop(x)`. Ejecútalo con un tiempo máximo.

- Prueba y ejecuta la función `runWithTimeout(x,y,z)` de `utils`.

- Inspecciona el fichero `containsGAGA.tm`. Es la descripción de una MT que acepta una entrada cuando en ella aparece la subcadena 'GAGA'. Las transiciones se representan como líneas

  `p->q: abc;R,d`

  indicando que si estoy en el estado `p`, y la entrada es cualquiera de los símbolos `a,b,c`, entonces escribo `d` y me muevo a la derecha. Si el símbolo de escritura no está, se supone que se deja el de entrada. `!` indica cualquier símbolo distinto de los que vengan a continuación y $\sim$, cualquier símbolo. El blanco es _.

- Carga el programa `simulateTM.py`. Simula la MT `containsGAGA.tm` sobre varias entradas.

El programa `simulateTM1.py` es muy similar al `simulateTM.py` solo que es un programa con una sola entrada que tiene la MT y la entrada. Como tiene que leer dos palabras (el programa y la entrada), eso se hace codificándola como una solo con el programa `ESS(MT,input)`. Este programa para el problema universal, la decodifica con `DESS(string)`

- Analiza la MT `binaryIncrementer.tm` y ejecuta esta MT para alguna entrada.
- Carga el programa `universal.py`. Este es un programa universal escrito en Python para programas en Python. Es decir, la entrada es: un programa M en Python, y una cadena ω. El programa universal ejecuta M sobre ω y devuelve la salida. Compruébalo para programas y entradas que siempre terminan. Compruébalo en un programa que siempre cicla (por ejemplo `loop.py`).
- Carga el programa `ignoreInput.py`. Este programa ignora su entrada, considera el mismo programa guardado en el fichero `progString.txt` y lo ejecuta sobre la entrada en el fichero `inSring.txt`. Crea dos ejemplos de ficheros de entrada y prueba el programa `ignoreInput.py`. Esto lo podéis hacer con los comandos:

  ```
  utils.writeFile('progString.txt', rf('containsGAGA.py'))
  ```

  ```
  utils.writeFile('inString.txt', 'GGGGGGGTTT')
  ```

- El problema `YESONSTRING` consiste en leer un programa y una entrada, y determinar si el programa termina con la salida 'yes' para esa entrada. Es el equivalente al problema universal, cambiando MT por programa en Python y aceptación por salida 'yes'.

  Carga el programa `recYesOnString(inString)`. Este programa responde siempre en los casos positivos, aunque no termina en algunos casos negativos (cuando el programa que se lee cicla). Esto demuestra que este problema es semidecidible (el lenguaje asociado es recursivamente enumerable). Analiza el programa y pruébalo en algunos casos.

  Hay que tener en cuenta que el programa solo tiene una palabra como entrada. Como tiene que leer dos palabras (el programa y la entrada), eso se hace codificándola como una solo con el programa `ESS(MT,input)`. Este programa para el problema universal, la decodifica con `DESS(string)`

  Prueba lo siguiente:

  ```
  entrada = utils.ESS(rf('containsGAGA.py'),'GAGAGAGAG')
  ```

  ```
  recYesOnString(entrada)
  ```

  ```
  entrada2 = utils.ESS(rf('recYesOnString.py'),entrada)
  ```

  ```
  recYesOnString(entrada2)
  ```

  Observa como la segunda salida es lo mismo que la primera. En la segunda, hemos ejecutado `recYesOnString()` sobre él mismo y otra entrada, que es un programa y otra entrada.

- Carga el programa `repeatCAorGA(inString)`. Observa su funcionamiento. Ahora carga el programa `alterGAGAtoTATA(inString)`. Ejecútalo sobre `ESS(rf('repeatCAorGA.py'),'GA')`. El resultado debe de ser 'TATA'. Observa como `alterGAGAtoTATA(inString)` es como el programa universal modificado: tiene la misma salida, excepto si es 'GAGA' que se cambia a 'TATA'.

  Luego podemos hacer programas universales modificados cambiando las salidas con un procedimiento similar.

- Carga el programa `maybeLoop.py`. Analízalo y determina qué salida proporciona cuando se da como entrada la cadena vacía y cuando se lee a él mismo como entrada.

- Carga el programa `yesOnString.py`. ¿Termina siempre este programa? Poner ejemplos de casos en los que termine con 'yes', con 'no' y que no termine.

- Carga el programa `yesOnStringApprox.py`. Observa como este programa da la respuesta correcta sólo para cuatro programas `containsGAGA(inString)`, `longerThan1K(inString)`, `yes(inString)`, `maybeLoop(inString)`. En los demás casos responde 'unknown'.

- Carga el programa `yesOnSelf.py`. Observa como este programa está basado en la existencia de `yesOnString.py`. Lo que hace es comprobar si un programa responde 'yes' cuando se lee a él mismo como entrada. Este programa no siempre termina, pero si `yesOnString.py` siempre terminase, entonces siempre terminaría. Comprueba que hace en los siguientes casos:
  `yesOnSelf('not a program')`, `yesOnSelf(rf('containsGAGA.py'))`, `yesOnSelf(rf('yes.py'))`, `yesOnSelf(rf('longerThan1K.py'))`, `yesOnSelf(rf('yesOnSelf.py'))`.

  Observa como en el último caso da un error, aunque debería de estar llamándose a él mismo de forma indefinida.

  Este es el análogo al lenguaje complementario al de diagonalización. Responde de forma correcta los casos en los que el programa termina sobre él mismo, pero no el resto. En realidad `yesOnSelf(rf('yesOnSelf.py'))` no está definido lo que va a hacer si terminar o no terminar.

- Carga el programa `noYesOnSelf.py`. Este programa se basa es la existencia de `noYesOnSelf.py`. Pruébalos en los mismos casos que el anterior, incluyendo `noYesOnSelf(rf('noYesOnSelf.py))`. Esta última ejecución da un error. Si el programa `yesOnString(prog,input)` diese siempre la respuesta correcta, este programa resolvería el equivalente al problema de diagonalización, pero este programa da problemas porque el programa `yesOnString(prog,input)` no siempre termina con la respuesta correcta. De hecho, este hipotético programa termina si y solo si no termina cuando se lee a él mismo como entrada.