NOMBRES: Diego Chahuan, Pablo Domínguez, Agustín Monti, Tomás Vergara

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Instituto de Ingeniería Matemática y Computacional

IMT1001 — Intro. Ingeniería Matemática — 1' 2020

Implementing a Register Machine

Introducción: Durante el primer taller del curso vimos cómo funcionaba tanto la máquina de Turing como una máquina RAM. Una cosa que fue muy útil para comprender el funcionamiento de la máquina de Turing fue el programa en la página *https://turingmachinesimulator.com/*. En este programa, se emulaba el funcionamiento de una máquina de Turing, mostrando cómo computaba los resultados. Sin embargo, no encontramos ningún equivalente para la máquina de RAM. Por lo que nació la idea de crear un emulador de la máquina RAM.

Pasos inciales: Al principio del proyecto, ninguno de nosotros tenía experiencia alguna para programar una interfaz web. Durante la investigación tuvimos que aprender las bases de HTML (HyperText Markup Language), pero esto no era suficiente para el proyecto, ya que se hacía difícil procesar el input del usuario y no podíamos ir revelando la máquina por partes lo que hacía que se viera distinto a nuestro diseño original. Posteriormente, averiguamos que HTML interactúa con otros dos lenguajes de programación CSS (Cascading Style Sheets) y JavaScript. Con éstos fuimos capaces de darle estilo a nuestro texto e interactuar con el DOM, así poder darle funcionalidad y forma a nuestro HTML.

Pero a pesar de que conocíamos los mecanismos todavía teníamos que programar las interacciones entre las distintas partes y cómo iba a funcionar la máquina.

Estructura de la máquina RAM: Decidimos programar la web app en 3 componentes distintos, aunque no sean técnicamente un componente como lo sería si se hubiera programado en frameworks como *vue* o librerías como *react*. Los llamamos componentes ya que intentamos de que funcionaran de la manera más independiente posibles entre ellos. El primer componente es el input del usuario para la cantidad de registros y el size input. El segundo componente es la maquina, en donde se muestran los registros. El último componente es el lugar donde se colocan las instrucciones.

Primer componente: Este componente tiene como objetivo almacenar la cantidad de registros y el "size input" de la máquina RAM. Por lo tanto, dentro del componente existen dos clases input, que al apretar el botón de confirmar los datos, el contenido en el código en JavaScript. Antes de seguir, se revisa que los datos ingresados sean adecuados. Esto se hace a través de un eventListener que "escucha" cada vez que se aprieta el botón y ejecuta una función. Esta función lo que hace es llamar a otras funciones que revisan que los inputs sean números enteros y que no estén vacíos. Una vez que los input del usuario son verificados, se muestra el segundo componente y se deshabilita el botón así no se puede llamar de nuevo, ya que si se llamase de nuevo, se crearían nuevos registros además de los anteriores.

Segundo componente: Luego el se muestra el segundo componente que es la máquina. Éste tiene dos facetas distintas: primero hay un input, en donde se colocan los valores de los distintos registros de la máquina. Éstos deben estar separados con comas. Después, se apreta el botón para confirmar los datos que funciona de la misma manera que el anterior. Hay un eventListener, que ejecuta una función, que llama a otras funciones que chequean que todos los valores sean enteros, luego de chequea que haya la misma

cantidad de valores que size input. Es decir, si en el paso anterior el usuario ingresó un size input de 4, el usuario en esta línea de input tiene que colocar 4 valores de esta manera: 1,2,3,4. Finalmente cuando se confirman estos datos se rellana una tabla en DOM los registros y sus valores respectivos, es decir, el primer valor que se colocó va en el registro 0, el segundo, en el 1 y así sucesivamente. Después de ésto, se activa el tercer componente.

Tercer componente: El tercer componente es el menú de las instrucciones. Se divide en dos submenus: el primero es de instrucciones individuales, en donde el usuario coloca una acción, sus parámetros, aprieta un botón de ejecutar la instrucción. Primero se verifica que los parámetros sean válidos, es decir, que correspondan a un registro válido, sean números enteros y tengan tantos parámetros ingresados como requiera la instrucción. Después de ésto se ejectuta y se muestra un label con el resultado, que es el valor que hay en el registro 0 (definido así en clases). El segundo modo es con un listado de instrucciones, en donde las instrucciones se van colocando en una lista, que se muestra en el HTML como una lista ordenada y en el javascript, se guarda como un array de una tupla con string y lista de strings. Cada instrucción se le debe seleccionar cuál función se quiere aplicar y sus parámetros correspondientes. Una vez el usuario decide que la lista e instrucciones esta lista, se revisan que todas las instrucciones sean adecuadas (con los mismos criterios mecionados perviamente). Una vez se acepta la lista de instrucciones, la máquina empieza a ejecutar el algoritmo. Se pone en **bold** la instrucción actual y se va cambiando los valores de los registros de acuerdo a las instrucciones ingresadas.

Acerca del algoritmo detrás: La máquina funciona de la siguiente manera en javascript: se crea un objeto que tiene como parámetros, las instrucciones, y dos booleanos que funcionan con el único objetivo de diferenciar una instrucción única a una lista y cuando se ejecutan instrucciones como ADD, que se programó de forma recursiva, es decir, se crea otra maquina que corra los pasos. Este objeto tiene un también tiene la propiedad "pc" que es el contador de puesto de interacciones. Luego de crear la máquina, se ejecuta el método "correr" en donde, dentro de un while que funciona hasta que la instrucción sea un HALT, se lee el elemento[pc] de la lista de instrucciones y se ejecuta el método de este objeto que corresponde a la instrucción y se le colocan como parámetros la lista de strings que colocó el usuario. Por ejemplo el método GOTO cambia el valor de pc y el método INC le suma 1 al registro y mueve "pc" en 1 y así sucesivamente. Finalmente cuando la instrucción es HALT se verifica que sea la máquina principal y se muestra el resultado en el label que corresponde.

Fallas y potenciales mejoras a futuro: En algo que falla nuestra máquina es en ir ejecutando las instrucciones de una dada una lista de instrucciones. Al diseñar nuestro código originalmente, se nos olvidó pensar en colocar alguna forma de ir pasando de instrucción a instrucción una a una. Por ende en nuestro diseño original estaba completamente dirigido a correr todas las instrucciones sucesivamente hasta que termine de correr el programa. Sin embargo, es por eso que aplicamos el hecho que se pueda ejecutar una única instrucción, como manera de "reemplazar" esta funcionalidad. También hay un par de fallas en el diseño gráfico de la web app. Cuando ésta se hace muy angosta, las labels de arriba se desconfiguran. Es decir, salen de las líneas en que deberían estar y el app se ve de forma extraña. También otro aspecto que se podría mejorar es implementar un mejor algoritmo para ver si una instrucción se termina en algún punto ya que se pueden programar maquinas que corran de forma eterna. Esto se va a discutir con mas profundidad en la segunda sección de este informe.

Comentarios generales del código: Finalmente cabe recalcar que aunque acá se hable de javascript el programa fue realmente escrito en typescript, que es un super set de javascript que es fuertemente tipado, es decir, que hay que las variables tiene un tipo fijo, también se puede declarar este pero en la mayoría de las veces no es necesario ya que typescript tiene inferencia de tipo. Ésto se hizo por varias razones: la primera es que typescript utiliza ES6 pero se compila en ES5, por lo que lo hace utilizable por más tipos y versiones de navegadores. Nosotros tuvimos que usar ES6 ya que para que un usuario pueda ver los cambios en los registros a tiempo real fue necesario crear una función asincrónica que en typescript permite usar el operador

await que espera a que termine una función sleep que creamos nosotros. Ésto hace que se tenga que esperar por lo menos 100ms al tiempo en que se ejecuta una acción, y de esta manera, uno pueda ver los cambios que ocurren en la máquina RAM y no pasen de manera casi instantánea. Otra razón por la cual utilizamos typescript es que, al ser fuertemente tipado, hace que como programadores nos tengamos que preocupar más del código y permite que naturalmente se generen mejores prácticas y que haya menos probabilidades de que se genere un bug que no se pueda encontrar.

Un nuevo desafío: Si bien hasta aquí cumplimos todos los objetivos que se especificaban en el proyecto, decidimos intentar de seguir avanzando.

Una de las cosas que quisimos mejorar era sólo correr aquellos algoritmos que efectivamente "funcionen bien". A lo que nos referimos con funcionar bien es que sean algoritmos que computen algún resultado. Entonces nos propusimos encontrar alguna forma de reconocer los algoritmos que no computan, y de esa manera no correrlos cuando sean ingresados a la máquina RAM. Sin embargo, investigando la problemática encontramos que Alan Turing demostró la inexistencia de un algoritmo que determina si otro algortimo va a terminar (para la máquina de Turing). Como sabemos que la máquina de Turing puede representar todo algoritmo en cualquier lenguaje (incluido la máquina de RAM), entonces claramente nos encontramos que no podíamos ser capaces de encontrar este algoritmo que buscábamos. La demostración más sencilla de que no existe este algoritmo utiliza un argumento similar al de la diagonalización de Cantor.

(Que quede claro que esta demostración no es originalmente nuestra)

Supongamos que existe determinado algoritmo que determina si otro algoritmo va a terminar o no. A este algoritmo lo llamaremos $Termina(p, x)$, que recibe un algoritmo p con inputs x y retorna $True$ si es que el algoritmo termina y $False$ si no termina.

Podemos ahora de una nueva función $Diagonal(w)$ con input un algoritmo w la cual aplica $Termina(w, w)$, es decir, determina si el algoritmo w termina al tener de input el mismo algoritmo. Si termina, entonces se manda a un loop infinito. Si no termina, entonces se termina de correr $Diagonal(w)$. Por ende, nos queda:

$$Diagonal(w)\ termina \iff w(w)\ no\ termina$$

Como w representa todo algoritmo, entonces podemos ver qué pasa si w es $Diagonal$:

$$Diagonal(Diagonal)\ termina \iff Diagonal(Diagonal)\ no\ termina$$

Lo cual es una contradicción. Por ende, no puede existir un algoritmo que determine si otro algoritmo va a terminar.

Pero el problema sigue existiendo. Entonces nace la pregunta: ¿Cómo afrontamos a un algoritmo que no va a terminar?

Solución Fácil: La primera solución que se nos ocurrió fue simplemente probar el algoritmo de manera "oculta" antes de mostrárselo al usuario. Esto haría que ningún usuario jamás vea un algoritmo que se ejecuta infinitamente en la máquina RAM. Sin embargo, como no sabemos cuándo si un algortimo dado va a terminar, debemos correr el algoritmo al infinito, lo cual deja a nustro programa trabajando ineficientemente en un algoritmo que no está bien diseñado.

Una solución más inteligente: El hecho de que no exista un algortimo para reconocer todo algoritmo que no termina, no implica que no exista ningún algoritmo para reconocer si ciertos algoritmos no van a terminar. Para entender qué algoritmos no terminables que podemos reconocer, vamos a hacer una analogía con ecuaciones diferenciales.

Antes de nada, es importante definir varias cosas.

Estado: El estado de un algoritmo es un conjunto de valores con respecto a cada registro definido en la máquina de RAM, además de la instrucción actual del algoritmo. Por ejemplo, una máquina de RAM que se encuentra actualmente en su intrucción 3 que tenga de registros:

R0	R1	R2	R3
2	1	2	0

Tendría un estado definido por $\{3, 2, 1, 2, 0\}$. Primero se coloca la posición actual del algoritmo, y posteriormente se colocan los valores de los registros en el mismo orden.

Cambio de estado: El cambio de un estado es simplemente la diferencia entre un estado y su estado anterior, dada una serie de instrucciones en la máquina RAM. Es decir $\Delta_i E = E_{i+1} - E_i$, donde E representa un estado genérico e i representa el i-ésimo estado al correr el algoritmo en la máquina RAM.

Entonces, con estas definiciones claras, podemos modelar todo cambio de estado de la forma:

$$\Delta_i E = g(E_i)$$

La razón por la cual podemos afirmar ésto, es que claramente todo cambio de estado depende únicamente del estado en el que se encuentra, es decir, no depende explícitamente del número i el cual sea el estado, sólo de sus variables. Ésto se ve ya que claramente el número del estado actual es absolutamente irrelevante de los cambios que las instrucciones van a determinar para el estado del algortimo.

Sistema de EDOs de primer orden con coeficientes constantes: De ésto podemos hacer una analogía con ecuaciones diferenciales. En particular, vamos a notar que un sistema ecuaciciones diferenciales autónomas se modela como:

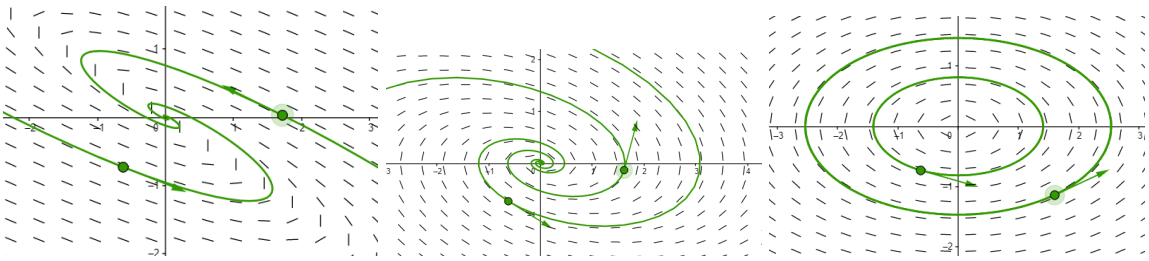
$$\frac{d\vec{x}}{dt} = g(\vec{x})$$

Para realizar la analogía (y que se entienda en términos sencillos), vamos a suponer que $\vec{x}$ está compuesto por únicamente 2 variables (x, y). También, vamos a suponer que $g(x)$ se puede modelar como una matriz de entradas constantes. Por ende, nos quedaría de la forma:

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Ahora, supongamos que existe determinado punto crítico, es decir, un $\vec{x}$ tal que $\frac{d\vec{x}}{dt} = 0$. Sabemos que este punto crítico se puede comportar de 3 maneras distintas: Es asintóticamente estable, establo pero no asintóticamente estable, o inestable. Que sea asintóticamente estable implica que todo punto inicial de $\vec{x}$, a medida que el tiempo tiende a infinito, el $\vec{x}$ va a tender al punto crítico. Que sea estable pero no asintóticamente estable implica que todo punto inicial dintinto del punto crítico se puede delimitar en una zona alrededor del punto crítico a medida que el tiempo tiende a infinito. Por último, que sea inestable implica que todo punto inicial distinto del punto crítico va a divergir a medida que el tiempo tiende a infinito.

Nuestra analogía se basa en entender un punto crítico de un sistema de EDOs como el estado en que un algoritmo termina. La razón por la cual podemos hacer esto es que, en el punto crítico se tiene que $\frac{d\vec{x}}{dt} = 0$, al igual que en el estado donde termina se puede interpretar que $\Delta_i E = 0$, ya que no va a cambiar su estado en el futuro. Además, en ambos casos, el cambio depende únicamente del estado actual (y no explícitamente del tiempo). Es decir, son sistemas autónomos.



(a) Asintóticamente estable (b) Inestable (c) Estable pero no asintóticamente

Con esta concepción, entonces todo algoritmo que efectivamente termina, en una EDO de este estilo, sería equivalente a que converja al punto crítico. Por ende, sólo las EDOs que sean asintóticamente estables representan algoritmos que terminan. Sin embargo, notamos que dentro de los algoritmos que no terminan hay dos representaciones en su forma EDO. Tanto los sistemas inestables como los estables pero no asintóticamente estables representan algoritmos que no converjen al punto crítico. De aquí nosotros sacamos que podemos reconocer ciertos algoritmos que no converjen: los que sean estables pero no asintóticamente estables. La razón por la cual los podemos reconocer es que siempre circulan en el mismo recorrido, es decir, el hecho de que vuelva a pasar por un mismo lugar implica que es un sistema estable pero no asintóticamente estable. Entonces, si encontramos un algoritmo para revisar si ya existió otro estado que era igual, implica que el algoritmo nunca va a terminar.

En términos formales: Sean dos estados de un algoritmo para los pasos i y j tal que $i \neq j$, si $E_i = E_j$, entonces el algoritmo no va a terminar. No vamos a realizar una demostración muy rigurosa, pero una intuición de esto va de la siguiente manera:

Sin pérdida de generalidad, vamos a suponer que $i < j$. Entonces, vamos a definir el periodo p como $p = j - i$. Teniendo que:

$$E_i = E_j$$

Entonces al aplicar un paso del algoritmo se tiene que:

$$g(E_i) = g(E_j)$$
$$\Delta_i E = \Delta_j E$$

Sumando cada estado a cada lado (son iguales):

$$E_i + \Delta_i E = E_j + \Delta_j E$$

$$E_{i+1} = E_{i+1}$$

Por ende, se tiene que el siguiente paso de cada estado también resulta mantenerse la igualdad. Si realizamos este proceso p veces llegamos a que:

$$E_{i+p} = E_{j+p}$$

Como $p = j - i$:

$$E_j = E_{j+p}$$
$$E_i = E_j = E_{j+p}$$

De la misma manera, podemos realizar el mismo proceso para llegar a concluir que:

$$E_i = E_{i+n \cdot p}$$

Donde $n \in \mathbb{N}$. Como existen infinitos estados iguales a E_i, entonces implica que el algoritmo no puede terminar (ya que sino habría un número finito de estados).

Entonces, proponemos encontrar un algoritmo que revise si un estado ya ha sido atravesado al correr el algortimo.

Primera idea: Lo primero que a todos se nos ocurriría sería almacenar los estados a medida que vamos corriendo el algoritmo, pero para cada estado nuevo, recorrer la lista de estados verificando que sea distinto a todos.

Sin embargo, ésto no es una idea muy eficiente, ya que estaríamos revisando muchos más casos de los necesarios. Si hubieron n estados anteriores, debiesemos revisar n casos para recién ahí seguir al siguiente estado.

Por lo tanto, planteamos aplicar una búsqueda binaria. La búsqueda binaria es un algoritmo para determinar si en una lista ordenada de números se encuentra determinado valor. Se basa en ir creando "pivotes" los cuales van acotando el rango de posibles valores donde se debe revisar. El funcionamiento del algortimo no es muy relevante para nuestro proyecto, hay más información en (https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search). Lo relevante de este algoritmo es que corre con eficiencia de $Log(n)$, lo cual es considerablemente mejor que el n de nuestra idea anterior.

Sin embargo, este algoritmo requiere poder ordenar nuestra lista de estados. Entonces nace la pregunta: ¿Qué significa que una lista de estados esté ordenada? En pocas palabras, significa lo que queramos que signifique. Lo único que importa es que podamos definir lo que en matemáticas discretas se llama un orden total. Un orden total es una relación binaria que es refleja, antisimétrica, transitiva y conexa. Vamos a escribir nuestro orden total como "$\leq$", y vamos a decir que un estado es "menor o igual" que otro. Nuestra definición del orden total será decir que un estado es menor o igual que otro sí es que al ir recorriendo las componentes de ambos estados, el estado menor es el que tiene la primera componente que resulta ser menor que el otro. Por ejemplo, $\{1, 2, 2, 4\} \leq \{1, 2, 3, 1\}$ ya que en la tercera componente se tiene que $2 < 3$. Obviamente, dos estados que son iguales también se considera que cada uno es menor o igual al otro. Realizaremos una rápida demostración de cada propiedad sólo para mostrar que el orden está bien definido.

Refleja: Todo estado es menor o igual a sí mismo como mencionado previamente.

Antisimétrica: Si $E_i \leq E_j$ y $E_j \leq E_i$, entonces necesariamente todos sus componentes deben ser iguales, es decir, $E_i = E_j$.

Transitiva: Si $E_i \leq E_j$ y $E_j \leq E_k$, entonces necesariamente el componente que hizo que $E_i \leq E_k$ va a también aplicar lo mismo para E_k, por ende $E_i \leq E_k$.

Conexa: En todo par de estados E_i, E_j necesariamente uno debe ser menor o igual al otro, ya que en alguna componente alguno va a ser menor al otro, y si todos son iguales, entonces ambos se consideran menor o igual al otro.

Por lo tanto, ya tenemos una forma de ordenar una lista de estados. Pero ahora cabe la pregunta: ¿Cómo encontramos un algoritmo para ordenar una lista de estados con respecto a este orden? Buscando distintos algoritmos nos encontramos que los mejores algoritmos para ordenar una lista corren en el orden de $nLog(n)$ como por ejemplo, el algoritmo QuickSort. Sin embargo, esto es peor que la idea original que sólo requería un orden de n. Pero de esto notamos de que no estamos considerando una cosa. No es necesario ordenar la lista cada vez, sino que se debe mantener el orden anterior, pero teniendo cuidado que cada nuevo estado se

ingrese a su posición correcta de acuerdo al orden. Entonces se nos ocurrió la idea de mezclar la búsqueda binaria para insertar el nuevo estado.

La búsqueda binaria, en su última iteración, cierra el rango a una sóla posición, en donde debiese estar el valor buscado. Es por esto, que podemos modificar el algortimo, haciendo de que si la búsqueda binaria no encuentra el estado en dicha posición, entonces lo inserta en esta posición. De esta manera, el orden se mantiene a lo largo de la iteración y no es necesario ordenar la lista cada vez. Además, insertar un estado en una lista lo consideramos de orden 1, esto implica que no se modifica el orden del algoritmo de la búsqueda binaria. Es decir, se mantiene el orden de $Log(n)$.

Entonces, nuestro algortimo final quedaría de la siguiente forma (en pseudo-código):

```
Correr una línea del código del algortimo original
Registrar el estado de la máquina RAM
Realizar una búsqueda binaria del estado en la lista
Si el estado se encuentra en la lista: terminar de correr el programa
Si el estado no se encuentra en la lista: insertarlo en la posición correspondiente
Seguir con la siguiente línea de código
```

Como se puede ver nuestro algoritmo resulta estar en el orden de $Log(n)$ ya que hay una sóla línea que corre con esa eficiencia y todo el resto es de tiempo constante. Sin emabrgo, tenemos que considerar que este algortimo se ejecuta para todas las líneas de código de un algoritmo dado. Por ende, una nueva pregunta que vale la pena hacer es: ¿Cómo cambia la eficiencia del algorítmo general al aplicar este sub-algoritmo?

Sea un algortimo genérico tal que es de eficiencia en el orden de f, lo cual denotaremos como que el algortimo $\in \mathcal{O}(f(n))$. Entonces, en una máquina RAM sin modificaciones, podemos modelar la cantidad de pasos aproximada que toma el algortimo como $k_1 f(n)$ (para valores de n muy grandes). Entonces, nuestro algoritmo al ser aplicado sobre esta máquina de RAM tendría que ser aplicado sobre cada uno de los $k_1 f(n)$ pasos. Podemos entonces modelar la cantidad de pasos que agrega nuestro algoritmo como:

$$\sum_{i=2}^{k_1 f(n)} k_2 \ log(i-1)$$

Como nuestro algortimo se mostró que es de orden $log(n)$, por eso lo tomamos como un $k_2 \ log(n)$. Sin embargo, se debe notar que n ahora es $i-1$. Ésto es debido a que el algortimo considera n como la cantidad de estados que hay en la lista. Como i representa el número del estado actual, implica que hay $i-1$ estados en la lista. Además notar que se comienza la sumatoria desde $i=2$. Esto se debe a que en $i=1$, no hay lista que revisar, por lo que nuestro algoritmo no se ejecuta. Además, hay que aclarar que estamos haciendo varios supustos que no son del todo rigurosos (como que k_2 es el mismo para cada iteración), pero ésto nos va a dar una idea bastante precisa del órden nuevo. Como la función log es continua y continuamente creciente para todo $i > 2$, podemos suponer que la sumatoria se comporta de la misma manera que su integral.

$$\int_{i=2}^{k_1 f(n)} k_2 \ log(x-1)dx$$

Ésto nos queda:

$$k_2(k_1 f(n) log(k_1 f(n)-1) - log(k_1 f(n)-1) + 2 - k_1 f(n))$$

Como estamos sólamente analizando el órden, vamos a tomar la función "más grande" de este resultado:

$$k_2 \ k_1 \ f(n) \ log(k_1 f(n)-1)$$

Como $k_1, k_2, 1$ son constantes, entonces decimos que ésto es de orden $\mathcal{O}(f(n) \ log(f(n))$. Teniendo que previamente el órden era de $\mathcal{O}(f(n))$, al sumarse con un algoritmo que es de $\mathcal{O}(f(n) \ log(f(n))$, entonces queda

del orden de $\mathcal{O}(f(n)\ log(f(n))$.

Por lo tanto, todo algoritmo pasa de ser $\mathcal{O}(f(n))$ a $\mathcal{O}(f(n)\ log(f(n))$ al aplicarse nuestro sub-algoritmo. Es decir, todo algortimo va a emperorar su eficiencia al aplicarse nuestra idea. Sin embargo, esto no afecta tanto. Por ejemplo, digamos que existe un algoritmo que corre en tiempo polinomial, entonces:

$$\mathcal{O}(x^n) \to \mathcal{O}(x^n\ log(x^n))) = \mathcal{O}(x^n\ log(x)) \subseteq \mathcal{O}(x^{n+1})$$

Es decir, todo algoritmo que corría en tiempo polinomial normalmente, va a seguir corriendo en tiempo polinomial. Sin emabrgo, notablemente ésto de todas maneras es un tiempo peor al que se tenía sin aplicar el sub-algoritmo.

Después de llegar a este resultado, llegamos a varias preguntas que nos cuestionarían todo nuestro trabajo: ¿Vale la pena empeorar el rendimiento de todo algoritmo para detectar ciertos algortimos no terminables?

Si bien no afecta tanto el orden al que termina corriendo el algortimo, creemos que no vale la pena apicar nuestro sub-algoritmo. La cantidad de algortimos "estables" o "períodicos" en los que uno se pueda realíticamente encontrar en la vida real son bastante acotados. Un buen ingeniero que determina un algortimo a correr nunca debiese diseñar un algoritmo que no termina, y empeorar todo el resto de algoritmos sólo para encontrar sólo una cantidad limitada de estos casos nos parece ineficiente. Además, hay que considerar que nuestras definiciones de estado y del orden de los estados se complejizan mucho a medida que llegamos a algoritmos más complejos. Puede que en una máquina de RAM con sólo 4 registros todavía se mantenga sencillo, pero ésto cuesta bastante más si mezclamos distintas máquinas de RAM ejecutando distintos algortimos para llegar a una solución.

Es por esto que decidimos no implementar nuestro algoritmo. Pero el problema de los algortimos no terminables sigue existiendo. Por ende, decidimos implementar la solución fácil. Decidimos probar cada algoritmo antes de mostrárselo al usuario y sólo aquellos algoritmos que efectivamente estén "bien diseñados", es decir, que terminen, serán mostrados en ejecución en la máquina RAM.

Conclusión: En conclusión, logramos implementar una máquina RAM en una página web, con una interfaz que permite especificar número de registros y el tamaño de los inputs, mostrando los resultados finales después de ejecutarse el programa.

También demostramos que es imposible de encontrar un algoritmo para determinar si otro algoritmo va a terminar o no, pero seguimos trabajando para resolver el problema. De ésto llegamos a dos posibles soluciones: la solución fácil y la inteligente. Si bien la solución inteligente tenía conceptos mucho más complejos y se realizó un análisis mucho más profundo, afecta la complejidad de todo algoritmo ingresado a la máquina RAM, lo cual no creemos que justifica la aplicación del algortimo. De esta manera, concluimos que nuestra solución fácil resulta ser más conveniente para el caso, de manera de no perjudicar la eficiencia de los algortimos ingresados en nuestro emulador de la máquina RAM.