LA CONJETURA DE COLLATZ COMO FUNCIÓN PERSONALIZADA EN R

Un ejemplo función personalizada es una función construida específicamente para probar la célebre *conjetura de Collatz*. La conjectura de Collatz establece que si iterativamente, partiendo de cualquier número entero positivo n, se toma algún entero positivo, se divide entre dos si es par $\left(\frac{n}{2}\right)$ y se multiplica por 3 y se le suma 1 $(3n + 1)$, sin importar la duración del ciclo de iteraciones en algún momento se obtendrá como resultado la unidad. Matemáticamente función generadora del ciclo de iteraciones se puede expresar como la función a trozos (seccionada) presentada a continuación:

$$f(n) = \begin{cases} \dfrac{n}{2}, & si\ n\ es\ par \\ 3n + 1, & si\ n\ es\ impar \end{cases}$$

Mediante el uso de R y partiendo de un origen (valor inicial, valor semilla) es posible construir una órbita (imágenes sucesivas al iterar la función) para la función anterior y determinar, desde ese número, el tiempo de órbita (número de iteraciones) requerido para alcanzar la unidad.

Para construir en R la función anterior es necesario recordar rápidamente que los bucles se utilizan para repetir un determinado bloque de código (un determinado conjunto de líneas de código). Un tipo de bucle, un poco diferente a los vistos en la sección anterior, son los bucles del tipo "mientras" (que al igual que en su significado como adverbio, debe entenderse como "durante el tiempo que transcurre hasta la realización de lo que se expresa"), conocidos en inglés como "while-loop".

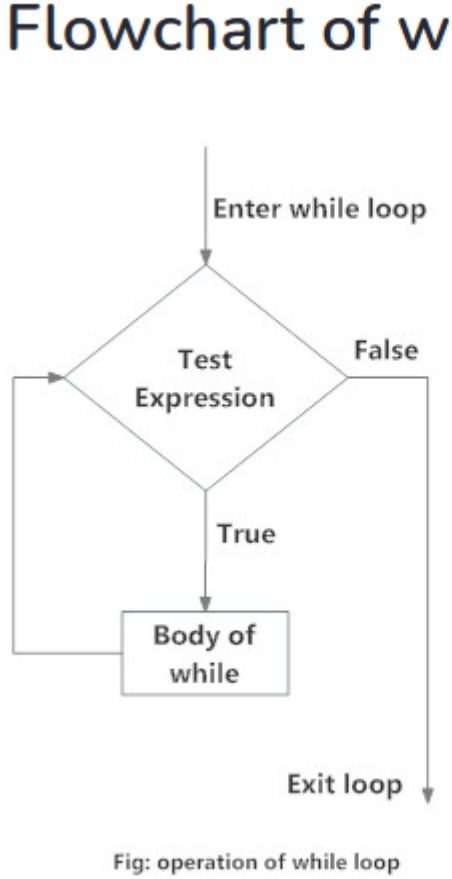


Fig: operation of while loop

Fuente: (DataMentor, 2021).

Aquí se planteará un algoritmo de Collatz empleando dos funcionales. El primero de ellos únicamente evaluará si un número entero positivo cumple la condición de ser par o impar (impar diferente de 1) y le realizará las operaciones antes descritas (dividir entre 2 si es par, multiplicar por 3 y adicionarle 1 si es impar). Como siempre, lo primero que se debe hacer es asignar un nombre a la función a crear mediante la sintaxis "function()", utilizando el operador "<-". En segundo lugar debe indicarse si para algún número n se cumple la condición ya descrita, por lo que hay que utilizar la sintaxis "if()" y dentro de ella establecer que si un número es diferente divisible perfectamente entre 2 (por lo que por definición es diferente de 1), debe dividirse tal número entre 2 (lo que generará una salida j, por lo que $j = n/2$) o, en caso contrario (específicamente cuando la división entre 2 sea imperfecta -número impar- y tal número sea diferente de 1), multiplicarlo por 3 y

adicionarle 1 (que es el otro escenario en el que se puede generar una salida j según lo planteado por Collatz, por lo que $j = 3n + 1$). Finalmente, se desea que el número obtenido (que es la salida j, sea proveniente de un número par o de un número impar diferente de la unidad) se muestre en la pantalla al usuario, por lo que se usa la sintaxis "return()", cuyo argumento será la salida j.

```
FunciónDeCollatz <- function(n){
 if(n != 1 & n%%2 == 0){
  j <- n/2
 } else{
  j <- 3*n + 1
 }
 return(j)
}
```

La razón por la que no se añade alguna especificación tras la instrucción "else" es porque el 1 quedará excluido automáticamente del bucle "while" (programado dentro del segundo funcional, el cual vuelve iterativa la función denominada "FunciónDeCollatz"), puesto que "while" estará sujeto a la condición "únicamente se ejecuta si es distinto de 1", entonces nunca se evaluará el 1 en la sintaxis "if()".

El segundo funcional es precisamente el algoritmo iterativo que tendrá como fundamento esencial el funcional (que puede verse en este escenario como un algoritmo no iterativo) anteriormente construido y definido bajo el nombre de "FunciónDeCollatz". Para ello, lo primero que debe hacerse es definir con antelación un vector en el cual se almacenarán los resultados de cada una de las iteraciones (que aquí se ha nombrado a tal vector como v, que R lo reconocerá como un vector lógico vacío). En segundo lugar, se define el segundo funcional como tal, el cual tendrá como argumento algún número m (para evitar conflictos

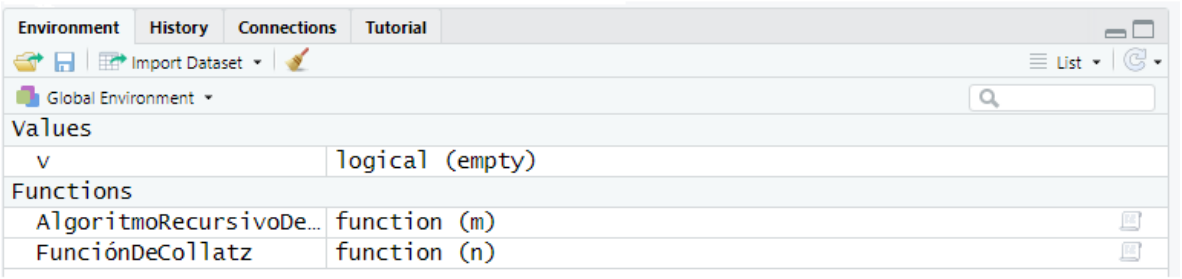
sintácticos en el lenguaje R, puesto que n se había utilizado en el funcional anterior). Para definir el segundo funcional como tal, se debe indicar que a cada uno de estos números m antes mencionados se le aplique "el siguiente algoritmo" (que será precisamente el algoritmo del tipo "while" que se construirá); este algoritmo consistirá en que R ejecute iterativamente, partiendo de un primer número m como igual a a [el cual representa el número inicial, valor inicial, valor semilla u origen de la simulación o ciclo de iteraciones (tal origen se denotará aquí como a) a partir del cual se construirán la órbita de la función (el conjunto de las imágenes sucesivas obtenidas al iterar la función) y se determinará el tiempo de órbita (número de iteraciones, que será la longitud del vector v)] y siempre que tal valor m sea diferente de 1, evaluaciones en el funcional "FunciónCollatz" aquí construido, y que el resultado de dicha evaluación sea almacenado en el vector v de la forma $v < -c(v, m)$, razón por la cual se definió al vector v como vacío en un principio. Debido a que el vector v se define como $v < -c(v, m)$ y a que en el estado inicial (antes de la primera iteración) el vector v está vacío (así se definió), tras la primera iteración (el momento inicial o momento t de la simulación, si se quiere ver desde la lógica de los sistemas dinámicos) se introducirá únicamente el valor semilla $m = a$ en el vector v (puesto que $v < -c(v, m)$ implica que tras cada iteración el vector v tendrá como primer elemento el resultado de la iteración anterior (que al no existir iteración cero, entonces para R el valor de v es un concepto que simplemente no aplica, por lo que únicamente se guardará el valor de $m = a$, el valor semilla) y como segundo elemento el resultado de la iteración actual (que para la segunda iteración será la primera imagen de la función de Collatz -imagen obtenida precisamente de evaluar la semilla en dicha función-). En cuarto lugar, se indica finalmente, mediante la sintaxis "return()" (diseñada especialmente para mostrar en pantalla un resultado en los funcionales personalizados, tal como se verifica en la documentación de R), que se debe mostrar en pantalla el vector numérico compuesto por el valor inicial y por los sucesivos valores v almacenados (resultantes de las iteraciones); como se indica

que debe mostrar el valor numérico inicial a y simultáneamente el vector v contiene a a [puesto que está programado como $v = c(v, m)$, por lo que el primer valor de v es únicamente $m = a$], entonces R sobrescribirá el valor semilla, evitando así la duplicidad (por ejemplo, si se arrancó con un valor semilla $a = 27$, la sobreescritura evita que al emplear la sintaxis return(c(a, v)) el número 27 aparezca de forma repetida, específicamente en la primera y la segunda posición).

```
v <- vector()

AlgoritmoRecursivoDeCollatz <- function(m){

 a <- m

 while(m != 1){

  m <- Collatz(m)

  v <- c(v, m)

 }

 return(c(a,v))

}
```

Así, el resultado de programar tal función en R se presenta a continuación.

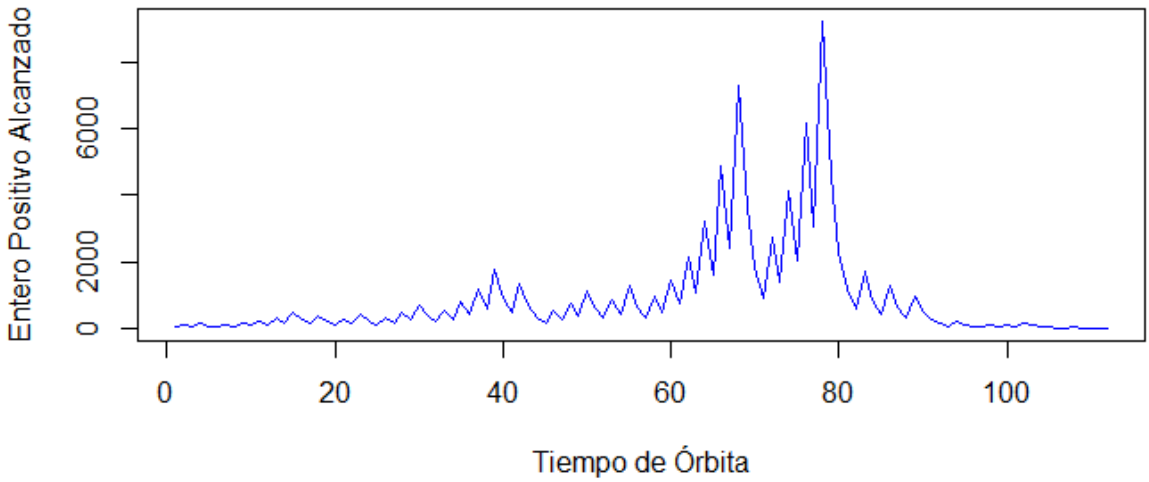
```
> FunciónDeCollatz <- function(n){
+   if(n != 1 & n%%2 == 0){
+     j <- n/2
+   } else{
+     if(n != 1 & n%%2 != 0){
+       j <- 3*n + 1
+     }
+   }
+   return(j)
+ }
>
> v <- vector()
> AlgoritmoRecursivoDeCollatz <- function(m){
+   a <- m
+   while(m != 1){
+     m <- FunciónDeCollatz(m)
+     v <- c(v, m)
+   }
+   return(c(a,v))
+ }
```



Por consiguiente, al evaluar un número cualquiera en el algoritmo iterativo diseñado, por ejemplo, 27, se obtiene el siguiente resultado.

```
> AlgoritmoRecursivoDeCollatz(m=27)
  [1]   27   82   41  124   62   31   94   47  142   71  214  107  322  161  484  242  121
 [18]  364  182   91  274  137  412  206  103  310  155  466  233  700  350  175  526  263
 [35]  790  395 1186  593 1780  890  445 1336  668  334  167  502  251  754  377 1132  566
 [52]  283  850  425 1276  638  319  958  479 1438  719 2158 1079 3238 1619 4858 2429 7288
 [69] 3644 1822  911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154  577 1732  866  433
 [86] 1300  650  325  976  488  244  122   61  184   92   46   23   70   35  106   53  160
[103]   80   40   20   10    5   16    8    4    2    1
```

Como puede observarse, el tiempo de órbita es de 112 iteraciones. Adicionalmente, si se desea visualizar el comportamiento del algoritmo durante su tiempo de órbita, es posible realizarlo mediante la sintaxis plot(AlgoritmoRecursivoDeCollatz(m), type = "l", col = "blue", xlab = "", ylab = ""), en donde para el ejemplo anterior $m = 27$, tal como se presenta a continuación.



Una forma alternativa de diseñar el algoritmo de Collatz, más compleja en términos de imaginarla, pero más simple en términos de su construcción y comprensión, puesto que permite realizar en un mismo funcional todo el proceso, es sugerida por un entrañable amigo de toda la vida y que se presenta a continuación.

El problema exige como solución un funcional con las siguientes características:

1) Su entrada ("input") es cualquier $N \in \mathbb{N}$.
2) Su salida ("output") son tanto una lista de pasos para llegar, a partir de algún $N \in \mathbb{N}$, hasta 1, como el conteo del número de pasos contenido en el listado anterior.

Así, la solución puede plantearse de la siguiente manera:

```
Collatz <- function(n, v = NULL) {
  v <- c(v,n)
  if(n == 1){
    return(list(pasos=v, iteraciones=length(v)))
  } else {
    if(n %% 2 == 0){
      n <- n/2
```

```
    } else{
       n <- 3*n + 1
    }
    return(Collatz(n, v))
  }
}
```

Funciones estándar usadas:

c:se usa para combinar dos o más vectores. La sintaxis es $c(x_1, x_2, \dots)$*, donde* x_i *son vectores y la salida es un vector de la forma* $y[x_1, x_2, \dots]$.

list: crea una lista o diccionario donde se asigna a cada variable un valor correspondiente que puede ser una dimensión o un vector. La sintaxis es $list(a = \dot{a}, b = \dot{b}, \dots)$.

length: cuenta el número de elementos que componen un vector, o bien el número de dimensiones que este posee. La sintaxis es lenght(x) y la salida es un escalar.

Sobre la sintaxis "return(Collatz(n, v)" se deben aclarar algunas cuestiones. La sintaxis "return" no es más que, expresándolo en los términos más simples posibles, el "cierre" de la declaración de una función. Así, "return" no es más que la salida de una función; el valor de $f(x)$. Imprimir el valor de $f(x)$ en pantalla se hace siempre con la sintaxis "print". Como señaló mi amigo, en muchas ocasiones puede surgir una confusión alrededor del nombre de la sintaxis "return", puesto que podría hacer pensar al usuario que es de "retornar" en el sentido de "regresar"; sin embargo, el sentido y dirección de ese "retornar" es de que el compilador "retorne" o "devuelva" al usuario de R un determinado resultado al ejecutar la función. Por otro lado, el lector puede observar que dentro del mismo funcional se especifica que, en caso de no cumplirse tal o cual condición, debe regresarse al inicio del funcional (que es el conjunto de sintaxis antes expuesto), a pesar que para esas alturas de la línea de código no se ha terminado de crear el funcional. Lo anterior es posible porque el funcional, al ser ejecutado, "corre" paso

a paso cada una de las líneas de código (que expresan las instrucciones del algoritmo) y, puesto que uno de los pasos es ejecutar la misma función, lo realizará dentro del mismo paso (lo que implica regresar al inicio del funcional). Esta solución aplica un principio fundamental de programación llamado recursión. Una función recursiva es simplemente una función que, dentro de su propia ejecución, se llama o ejecuta a sí misma.

Antes de comenzar a programar, es necesario analizar y comprender el algoritmo a diseñar:

1. Ingresa N a la función y evaluar el caso correspondiente según el valor de N.
2. Registrar el valor de N en una nueva dimensión del vector V.
3. Evaluar el valor de N para decidir el siguiente cálculo correspondiente.

    3.a. ¿ N es mayor que 1?

        3.a.1. ¿N es par?

            3.a.1.1. Asignar $N \longleftarrow \frac{N}{2}$

        3.a.2. ¿N es impar?

            3.a.2.1. Asignar $N \longleftarrow 3N + 1$

        3.a.3. Regresar al paso 1

    3.b. ¿N es igual a 1?

        3.b.1. Mostrar el contenido del vector V y la cantidad de dimensiones del vector V.

        3.b.2. Fin de la ejecución.

Al revisar el paso 3.a.3. del caso de uso, se observa que se ejecuta la misma función una y otra vez y se evalúa el valor de entrada de la misma cada vez que se ejecute, lo único que cambia es el valor de entrada o valor semilla. Matemáticamente esto se expresa de la siguiente forma, donde $C(n)$ denota Collatz evaluada en n:

$$C(n) = \begin{cases} C\left(\frac{n}{2}\right), & n\ par \\ C(3n+1), n\ impar \\ 1, & n = 1 \end{cases} \qquad (1)$$

$$C\left(\frac{n}{2}\right) = \begin{cases} C\left(\frac{n/2}{2}\right), & \frac{n}{2}\ par \\ C\left(3\frac{n}{2}+1\right), \frac{n}{2}\ impar \\ 1, & \frac{n}{2} = 1 \end{cases} \qquad (2)$$

$$C(3n+1) = \begin{cases} C\left(\frac{3n+1}{2}\right), & 3n+1\ par \\ C(3(3n+1)+1), 3n+1\ impar \\ 1, & 3n+1 = 1 \end{cases} \qquad (3)$$

Sustituyendo (3) y (2) en la ecuación (1):

$$C(n) = \begin{cases} \begin{cases} C\left(\frac{n/2}{2}\right), & \frac{n}{2}\ par \\ C\left(3\frac{n}{2}+1\right), \frac{n}{2}\ impar, & n\ par \\ 1, & \frac{n}{2} = 1 \end{cases} \\ \begin{cases} C\left(\frac{3n+1}{2}\right), & 3n+1\ par \\ C(3(3n+1)+1), 3n+1\ impar \\ 1, & 3n+1 = 1 \end{cases}, n\ impar \\ 1, \quad n = 1 \end{cases} \qquad (1)$$

De esta forma se demuestra que, para calcular $C(n)$ es necesario calcular $C(\frac{n}{2})$ o $C(3n+1)$, según sea el caso correspondiente ya sea que n sea par o impar. Asimismo, para calcular, por ejemplo, $C(n/2)$, es necesario calcular $C(\frac{n/2}{2})$ o bien $C(3\left(\frac{n}{2}\right)+1)$, según sea el caso de que $n/2$ sea par o impar, y así sucesivamente hasta llegar a 1. Ejemplificando, se puede partir de n=8

$$C(8) \rightarrow par, \frac{8}{2} = 4$$

$$C(4) \rightarrow par, \frac{4}{2} = 2$$

$$C(2) \rightarrow par, \frac{2}{1} = 1$$

$$C(1) = 1$$

Se observa que para calcular $C(8)$ fue necesario calcular $C(8/2)$, y para calcular este último fue necesario calcular $C(\frac{8/2}{2})$ y así sucesivamente hasta llegar a un valor conocido donde no era necesario ningún cálculo. Se sabe que $C(1) = 1$, así que al llegar a esto no es necesario continuar con la recursión y se puede comenzar a calcular hacia atrás a partir de este valor conocido.

Como ejemplo adicional, si el lector tuviese un listado de instrucciones como

Listado X

1. Ejecutar A1.
2. Ejecutar A2.
3. Ejecutar las instrucciones del Listado X.
4. Regresar e imprimir en pantalla "Hola, Fernanda".

Es posible observar que en el paso 3 el algoritmo contiene una referencia recursiva a sí mismo. Al llegar al paso 3, el algoritmo o funcional consultará la definición de cuáles son las instrucciones del Listado X y volverá a ejecutar A1, luego A2, y luego volverá a consultar la definición de cuáles son las instrucciones del Listado X y volverá a ejecutar A1, luego A2, y luego volverá a consultar y así de manera infinita numerable, lo cual se ilustra en el siguiente diagrama.

Lo más importante es resaltar que la totalidad del bucle infinito antes descrito ocurre dentro de la ejecución del paso 3, por lo que nunca llegaría a ejecutar el paso 4 donde se muestra "Hola, Fernanda". Por eso es que es posible hacer recursión dentro de una función haciendo referencia a sí misma, puesto que al momento de llegar al paso 3 donde se le dice a R que ejecute las instrucciones del Listado X, este

va a buscar la definición de estas instrucciones de la misma forma en como ejecutaría cualquier otra función, sea esta una recursión en sí misma o no. Al recibir una instrucción de la forma “Ejecútese esta función”, el programa no distingue si esa instrucción es una referencia a sí misma o no, únicamente la ejecuta, como buen autómata.

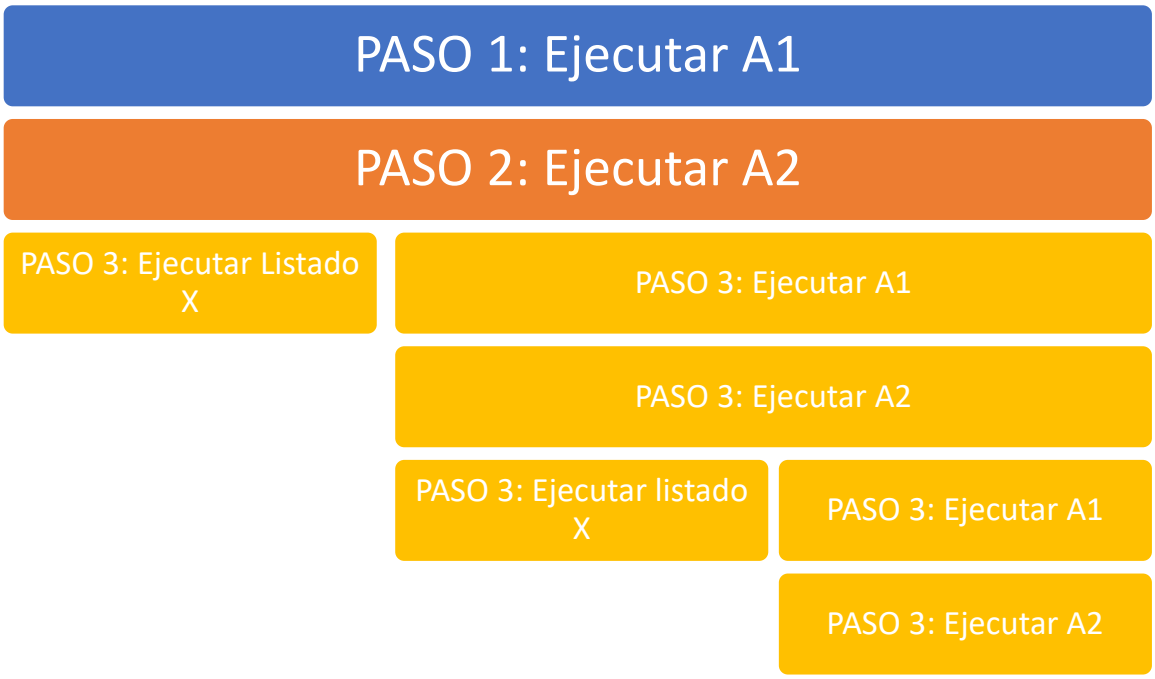
Paso 1	Ejecutar A1	
Paso 2	Ejecutar A2	
Paso 3	Ejecutar Listado X	Ejecutar A1 Ejecutar A2 Ejecutar Listado X

Ejecutar A1

Ejecutar A2

Ejecutar Listado X

- Ejecutar A1
- Ejecutar A2
- Ejecutar Listado X
  - Ej....



Finalmente, es necesario agregar que el lector debe haber notado que existe una lógica inversa (aunque isomórfica) entre el proceso recursivo y el planteamiento matemático de Collatz. Matemáticamente hablando, la función de Collatz parte de un determinado valor semilla y mediante el proceso recursivo antes descrito transforma tal valor semilla en 1, que es el punto de destino de la función de Collatz; en este sentido, la función de Collatz es un proceso continuo (no una función continua -de hecho, es una función seccionada a trozos-) que permite **ir** del valor semilla hacia la unidad. Sin embargo, computacionalmente la cuestión es diferente, puesto que, a pesar de que también al algoritmo computacional se le pide que determine la sucesión de números de conducen del valor semilla hasta la unidad de forma recursiva, este es un proceso numérico (lo que sería a nivel de las Matemáticas, las diferencias entre Análisis Matemático y Métodos Numéricos) y, por consiguiente, como proceso no es continuo, sino discreto.

Lo anterior obedece a una razón más general, que no atañe únicamente a este algoritmo, sino que tiene sus raíces en las diferencias generales en los procesos epistemológicos de las Matemáticas y de las Ciencias de la Computación. En el planteamiento matemático se coloca a la función (que expresa el fenómeno de

estudio) dentro de un marco teórico bien definido (lo que se conoce en ciencias como forma analítica), lo que implica que su comportamiento (como proceso) se conoce con antelación, puesto que se está haciendo abstracción de tales o cuales características particulares o singulares del fenómeno, mientras que en las Ciencias de la Computación (que versan sobre computar –"contar o calcular en número algo numéricamente", según la Real Academia Española en su diccionario del tricentenario actualización 2020-) se toman en consideración todas las características particulares o singulares que permiten especificar de forma unívoca la existencia del fenómeno, por lo que plantear de forma abstracta los procesos se vuelve más complejo y es necesario abordar su estudio paso a paso. En aras de la eficiencia computacional relacionada con la memoria caché de los equipos informáticos que ejecutan los procesos computacionales como lo son los antes descritos, tales procesos se ejecutan bajo un orden conocido como FILO o LIFO ("First in, last out", "Last in, first out"). Para comprender este orden es necesario estudiar un simple ejemplo relacionado con la palabra inglesa "stack", que en este contexto debe traducirse como una "pila de cosas", tal como se presenta en la siguiente imagen para el caso de platos de uso doméstico destinados a la alimentación familiar.

Fuente: (Wikipedia, 2020).

¿Cuántos pasos existen desde 8 hasta 1 al aplicar el funcional de Collatz?, existen 4 pasos, puesto que la pila de pasos o procesos (que sería el "stack" de pasos, en lugar de platos) consta de 4 componentes, tal y como se vio anteriormente . La analogía del papel que juega el orden FILO de ejecutar procesos computacionales a nivel informático en términos de la optimización de la memoria caché (y con ello, optimizar el proceso en general) con la pila de platos antes mencionada consiste específicamente que sería riesgoso para la integridad física de los platos que se sacase primero el primer plato que se apiló, y además sería ineficiente, puesto que se puede lograr lo mismo (desmontar la pila de platos) sin correr tal riesgo (que en

el contexto informático implica la suboptimización del uso de la memoria caché por los procesos[1].

[1] "FILO no es necesariamente una forma "justa" de acceder a los datos, ya que opera en el orden opuesto al de una cola. Aun así, el método FILO puede ser útil para recuperar objetos usados recientemente, como los almacenados en la memoria caché." (TechTerms, 2014). Al respecto, vale aclarar que una "Queue", que puede ser traducido como "cola" y es un sustantivo definido según el Google Traductor como "una lista de elementos de datos, comandos, etc., almacenados de manera que se puedan recuperar en un orden definido, ***generalmente el orden de inserción***". El texto en negrita y cursiva denota cómo las colas y las pilas funcionan, en general, en orden inverso. Los modelos matemáticos a su vez funcionan, usualmente, en el mismo sentido que las colas.