Recursividad

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Introducción

La recursividad es muy importante dado que muchos algoritmos en diversas áreas de la informática se basan en este principio y del mismo modo distintos lenguajes permiten la recursividad al usuario de manera abierta.

Lenguajes que permiten la recursividad

- `C++`
- `Python`
- `Wolfram Language`

Nota importante

Si el programador no conoce bien los fundamentos básicos de recursividad, podría caer en el error de desarrollar un programa recursivo que no ofrezca una salida exitosa.

Algoritmos recursivos

Existen múltiples ejemplos de programas recursivos, seguidamente estudiaremos dos casos, con el objetivo de comprender la recursividad.

Example

Un primer ejemplo lo encontramos en el siguiente programa elaborado con el software *Wolfram Mathematica*:

```
factoriales[n_]:=If[Or[n==0,n==1],Return[1],
n*factoriales[n-1]]
```

Example

La función de este código reside en calcular el factorial de un número entero no negativo n. Como el lector recordará si $n \in \mathbb{N} \cup \{0\}$ el factorial denotado $n!$ se define así:

$$n! = \begin{cases} 1 \text{ si } n = 0 \vee n = 1 \\ 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = n(n-1)! \text{ en otro caso} \end{cases}$$

- Al observar el código de *Wolfram Mathematica* anterior, se está utilizando un principio recursivo en el cálculo de $n!$ pues si $n \neq 0$ y $n \neq 1$ el factorial llama a la misma función denominada `factoriales` de manera sucesiva.

Nota aclaratoria

También, cabe aclarar en 1.1, que la segunda "coma" posterior al `Return[1]`, significa "else", además, se ha colocado el `Return` para mejorar la comprensión del código; sin embargo, en *Wolfram Language* se puede presindir de él, pues con solo colar el 1 ya el programa efectúa ese retorno.

Example

- Si $n = 5$ la función `factoriales[5]` entra en la primera invocación a calcular 5·`factoriales[4]`, al llamar a `factoriales[4]` se calcula 4·`factoriales[3]`, luego 3·`factoriales[2]` y finalmente 2·`factoriales[1]`.
- En este punto del recorrido del programa, `factoriales[1]` retorna como resultado un uno y como consecuencia de ello `factoriales[2]` retorna dos, `factoriales[3]` devuelve un seis, `factoriales[4]` retorna un 24 y `factoriales[5]` devuelve el resultado de $5! = 120$.
- A este comportamiento de invocación sucesiva de la función `factoriales[n]` hasta obtener el caso raíz `factoriales[1]` se le denomina *recursividad*.

Instalación del paquete VilCretas

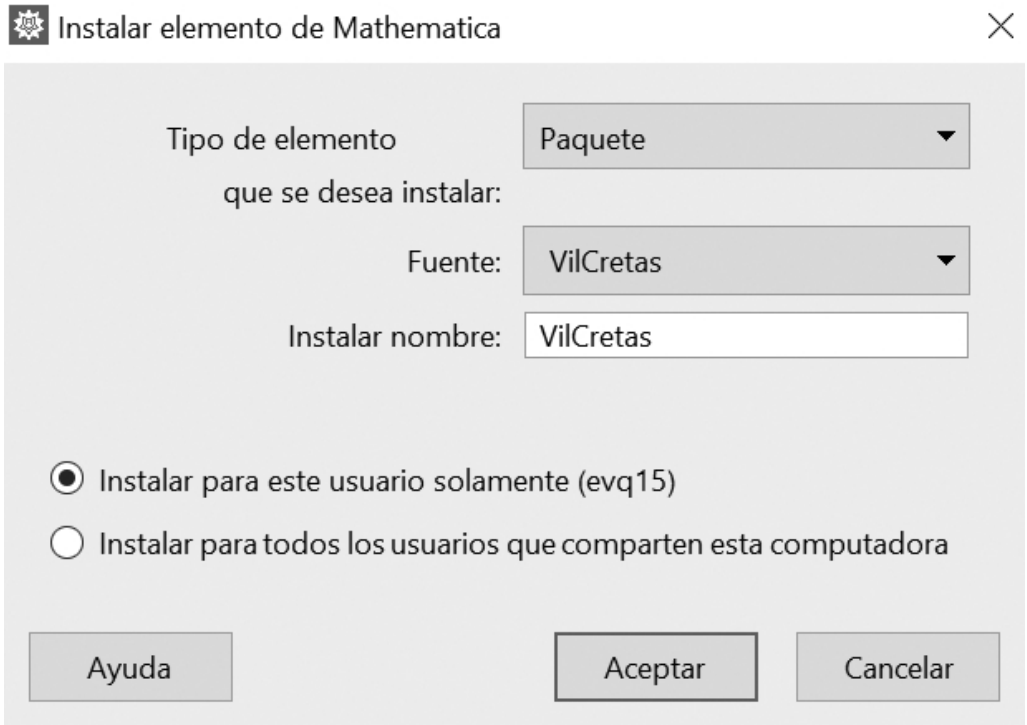
Este documento toma como apoyo un paquete de software elaborado por su autor llamado **VilCretas**. Iniciaremos con su uso explicando la manera en cómo se instala.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Packages/VilCretas.rar

Instalación del paquete VilCretas

Después de ser descargada la última versión del paquete **VilCretas**, en el software *Wolfram Mathematica*, se accede al menú **Archivo/Instalar**, mostrándose la ventana:



Instalación del paquete VilCretas

En **Fuente**, se direcciona la ubicación del archivo "VilCretas.m" y se presiona el botón "Aceptar".

Nota

Es importante aclarar que por defecto estas hojas de cálculo tienen extensión "nb" y se llaman cuadernos o *notebooks*.

Instalación del paquete VilCretas

Para tener acceso a los comandos que caracterizan el paquete **VilCretas** se debe añadir en el cuaderno la instrucción:

```
<< VilCretas`, o bien, Get["VilCretas`"]
```

Esta línea de código se ejecuta en el software presinando *shift+enter*. Si produce como respuesta:

```
$Failed
```

hubo un error en el proceso y por lo tanto se deben volver a realizar todos los pasos ya descritos. De lo contrario, si no hay ningún retorno, la librería ya estaría disponible para su empleo.

Instalación del paquete VilCretas

Explicación en video

https://youtu.be/jnpG7DD9ohc

Comando Factoriales

Un primer comando de interés del paquete **VilCretas** se denomina `Factoriales`. Esta instrucción constituye una implementación del programa 1.1.

In[ ] :=

```
Factoriales[5]
```

Out[ ] =

120

Comando Factoriales

Lo interesante del comando reside en la posibilidad que brinda para observar su código interno de programación y las ejecuciones paso a paso. De esta forma:

In[ ] :=

```
Factoriales[5, code -> True]
```

Out[ ] =

{120,Factoriales[n_]:=If[Or[n==0, n==1], Return[1], n*Factoriales[n-1]]}

La opción `code -> True` permite visualizar el código de construcción de `Factoriales`

Comando Factoriales

Además:

In[ ] :=

```
Factoriales[5, steps -> True]
```

Out[ ] =

Factoriales[5] = 5 * Factoriales[4] = 120
Factoriales[4] = 4 * Factoriales[3] = 24
Factoriales[3] = 3 * Factoriales[2] = 6
Factoriales[2] = 2 * Factoriales[1] = 2
Factoriales[1] = 1
120

Como se aprecia `steps -> True` retorna " paso a paso" las llamadas sucesivas del comando `Factoriales` hasta llegar al caso raíz Factoriales[1].

Comando Factoriales

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-1.zip

Explicación en video

https://youtu.be/n_N4lRnTdnE

Nota

El autor de este texto ofrece un segundo libro bajo la licencia *Creative Commons: Atribución-NoComercial-SinDerivadas,* donde se detalla por capítulo, la utilización y potencialidades de cada una de las instrucciones del paquete **VilCretas**, esto a través del uso de ejemplos. Se recomienda su consulta como una clara guía en la profundización de empleo de esta librería.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Packages/Matematica_discreta_a_traves_del_uso_del_Paquete_VilCretas.pdf

Un segundo ejemplo de una recursividad que particularmente recibe el nombre de *relación de recurrencia*, tema que se estudiará más a profundidad en el capítulo 2 de este texto, se enuncia a continuación:

Example

$$\begin{cases} a_n = a_{n-1} + a_{n-2} \\ a_1 = 1 \\ a_2 = 1 \end{cases}$$

A la sucesión numérica que produce esta relación de recurrencia se le conoce con el nombre de sucesión de *Fibonacci*

Reseña histórica

En el año 1202 *Fibonacci* escribió una obra denominada *Liber abaci* donde formuló un famoso problema de parejas de conejos que planteaba: "*¿cuántas parejas de conejos obtendremos al final de un cierto año, si empezando con una pareja, cada pareja produce cada mes una nueva pareja que puede reproducirse al segundo mes de existencia?*" . De este problema se origina la sucesión de números de *Fibonacci*.

En *Wolfram Mathematica* la relación de recurrencia de los números de *Fibonacci* se puede crear así:

```
fibonacci[n_]:=If[n==0||n==1,1,fibonacci[n-1]+fibonacci[n-2]]
```

Al correr, por ejemplo, el cálculo de `fibonacci[5]` el programa en la primera iteración establece que
`fibonacci[5]=fibonacci[4]+fibonacci[3]`
`fibonacci[4]=fibona cci[3]+fibonacci[2]`
`fibonacci[3]=fibonacci[2]+fibonacci[1]`
`fibonacci[1]=fibonacci[2]=1`, en las llamadas sucesivas hacia arriba se obtiene que `fibonacci[3]=2`, `fibonacci[4]=3` y `fibonacci[5]=5`, devolviéndose como resultado este último valor.

También, de una forma más directa, la relación de recurrencia que construye los elementos de la sucesión de *Fibonacci* se podría desarrollar en *Wolfram Language* así:

```
a_n_:=a_n-1+a_n-2
a_1=1;
a_2=1;
```

Mathematica cuenta también con el comando `Fibonacci[n]` que calcula el *n*-ésimo número de *Fibonacci*. Por ejemplo, se desea conocer el quinto término de la sucesión con este comando:

In[ ] :=	**Out[ ] =**
`Fibonacci[5]`	5

- La instrucción `Fibonacci[n]` es nativa del programa *Mathematica* y no de la librería **VilCretas**.
- En la siguiente descarga encontrará el glosario de comandos de *Mathematica* y **Vilcretas**.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Glosario/Glosario_de_comandos_de_Mathematica_y_VilCretas.rar`

Comando NFibonacci

- En el paquete **VilCretas** se encuentra disponible el comando `NFibonacci` que corresponde a la implementación anterior de `fibonacci`, con la particularidad de poseer las mismas opciones `code -> True` y `steps -> True` de la instrucción `Factoriales`. De esta manera:

In[ ] :=

```
NFibonacci[5, code -> True, steps -> True]
```

Comando NFibonacci

- De la entrada anterior se obtiene el siguiente resultado:

Out[ ] =

{5,NFibonacci[n_]:=If[Or[n==1, n==2], Return[1], NFibonacci[n-1]+
NFibonacci[n-2]]}
NFibonacci[5] = NFibonacci[4] + NFibonacci[3] = 3 + 2 = 5
NFibonacci[4] = NFibonacci[3] + NFibonacci[2] = 2 + 1 = 3
NFibonacci[3] = NFibonacci[2] + NFibonacci[1] = 1 + 1 = 2
NFibonacci[2] = 1
NFibonacci[1] = 1
5

Comando NFibonacci

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-2.zip

Explicación en video

https://youtu.be/7yDoJASyRIo

Nota importante

- Algo esencial, al programar recurriendo a la técnica de recursividad, consiste en tener la certeza de que el programa recursivo finaliza, es decir, el código de programación diseñado generará como salida de manera factible, un resultado que tiende a converger y no a diverger.
- En este sentido, se entenderá la palabra "convergencia" como la tendencia de un programa a proporcionar una salida durante su tiempo de ejecución, en caso contrario, se dice que es "divergente".

Propiedades de una recursividad

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

- Toda recursividad se invoca o se llama así misma a través de una definición base, ejecutándose las invocaciones una cantidad finita de veces.
- Cada llamada en el proceso produce una pila. Las invocaciones se detendrán hasta obtener una serie de reglas recursivas que definen el o los casos iniciales, también conocidos como casos raíz.
- Esta condición o condiciones, determinan la salida de la pila, en cuya ausencia, el programa continuaría llamándose así mismo de forma indefinida.

Nota

- En el ejemplo del cálculo del factorial abordado en la sección 1.1, la definición base es: `factoriales[n]=n*factoriales[n-1]` y solo hay un caso inicial para la salida de la pila, constituido por $1! = 1$, o bien, $0! = 1$, dependiendo del esquema de razonamiento empleado en la solución.
- En el ejemplo de la sucesión de números de *Fibonacci* expuesto en esa misma sección, la definición base la conforma: `fibonacci[n]=fibonacci[n-1]+fibonacci[n-2]`, o bien, $a_n = a_{n-1} + a_{n-2}$, bajo las condiciones iniciales $a_1 = a_2 = 1$. En ese ejercicio un caso raíz sería insuficiente para generar la sucesión numérica en cuestión.

Nota

- Los dos ejemplos de recursividades ya desarrollados, podrían hacer pensar al estudiante que un programa recursivo solamente se emplea para realizar un cálculo, sin embargo, esto es una falacia, pues también, se pueden utilizar para llevar a cabo un proceso.

Example

Se desea conocer si un dato "a" se encuentra en una lista de datos $L = \{x_1, x_2, \ldots, x_n\}$, un método básico de búsqueda consistiría en comparar cada elemento de L con "a". Un algoritmo que describe esta forma de resolución es el siguiente:

- [Inicialización] Si $L = \{\}$ finalice "dato no encontrado", sino sea v el último dato de la lista L.
- [Compare] Tome a v y compare con "a", si son iguales finalice "dato encontrado" de lo contrario vaya al paso 3.
- [Continuar con la búsqueda] Tome a $L = L - \{v\}$ y llame nuevamente a este algoritmo.

Example

- La idea consiste en ir comparando "a" con cada uno de los elementos de L, de "atrás hacia adelante". Por otra parte, sabemos que en algún momento terminan las llamadas del proceso pues L es una lista finita de datos y por consiguiente, será reducida a vacío en la peor situación (si "a" no es un dato de L).

Example

En *Wolfram Mathematica* una implementación del algoritmo descrito, corresponde a:

In[ ] :=

```
dato[a_,L_List]:=Module[{Lista=L},If[Lista=={},
"Dato no encontrado", Vl=Last[Lista];
If[ToString[a]==ToString[Vl],"Dato encontrado",
Lista=Delete[Lista,Length[Lista]];dato[a,Lista]]]]
```

- `Module` en *Wolfram Language* facilita la generación de un entorno de programación donde es posible inicializar variables y/o funciones locales. En `dato` fue necesario crear la variable `Lista` con la intención de ir eliminando cada vez, su último elemento, en caso de ser distinto de `a`. `Last` y `Delete` son operadores de listas propios de *Mathematica*. El primero retorna el último dato de `Lista` y el segundo efectúa su eliminación.

Comando Dato

- El paquete **VilCretas** cuenta con la instrucción `Dato`. Ésta constituye la misma implementación compartida en el método `dato` anterior, otorgando la posibilidad al usuario de observar el código de la función y una búsqueda paso a paso, recurriendo a las opciones `code -> True` y `steps -> True`, respectivamente. Así:

In[ ] :=

```
Dato["ocho", {3, a, 5, 9, "ocho", x, 7, 6, "hola"}, code ->
True, steps -> True]
```

Comando Dato

- Se obtiene la siguiente salida:

Out[ ] =

{Dato encontrado,Dato[a_,L_List]:=Module[{Lista=L},If[Lista=={},
"Dato no encontrado",Vl=Last[Lista];If[ToString[a]==ToString[Vl],
"Dato encontrado",Lista=Delete[Lista,Length[Lista]];Dato[a,Lista]]]]}
Dato[ocho,{3,a,5,9,ocho,x,7,6,hola}]
Dato[ocho,{3,a,5,9,ocho,x,7,6}]
Dato[ocho,{3,a,5,9,ocho,x,7}]
Dato[ocho,{3,a,5,9,ocho,x}]
Dato[ocho,{3,a,5,9,ocho}]
Dato encontrado

Comando Dato

- O bien:

In[ ] :=

```
Dato["Ocho", {3, a, 5, 9, "ocho", x, 7, 6, "hola"}, steps
-> True]
```

Comando Dato

- Cuya salida es la siguiente:

Out[ ] =

Dato[Ocho,{3,a,5,9,ocho,x,7,6,hola}]
Dato[Ocho,{3,a,5,9,ocho,x,7,6}]
Dato[Ocho,{3,a,5,9,ocho,x,7}]
Dato[Ocho,{3,a,5,9,ocho,x}]
Dato[Ocho,{3,a,5,9,ocho}]
Dato[Ocho,{3,a,5,9}]
Dato[Ocho,{3,a,5}]
Dato[Ocho,{3,a}]
Dato[Ocho,{3}]
Dato[Ocho,{}]
Dato no encontrado

Comando Dato

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-3.zip

Explicación en video

https://youtu.be/VnLiY6LYWYA

Nota

Todos los comandos del paquete **VilCretas** y de *Mathematica* propiamente, presentan una opción de ayuda al usuario. Por ejemplo, si en un *notebook* se presiona *shift+enter* al escribir: `?Dato`, se devuelve como salida una descripción sobre el uso de la instrucción `Dato`, tal y como se muestra en la figura 1.1.

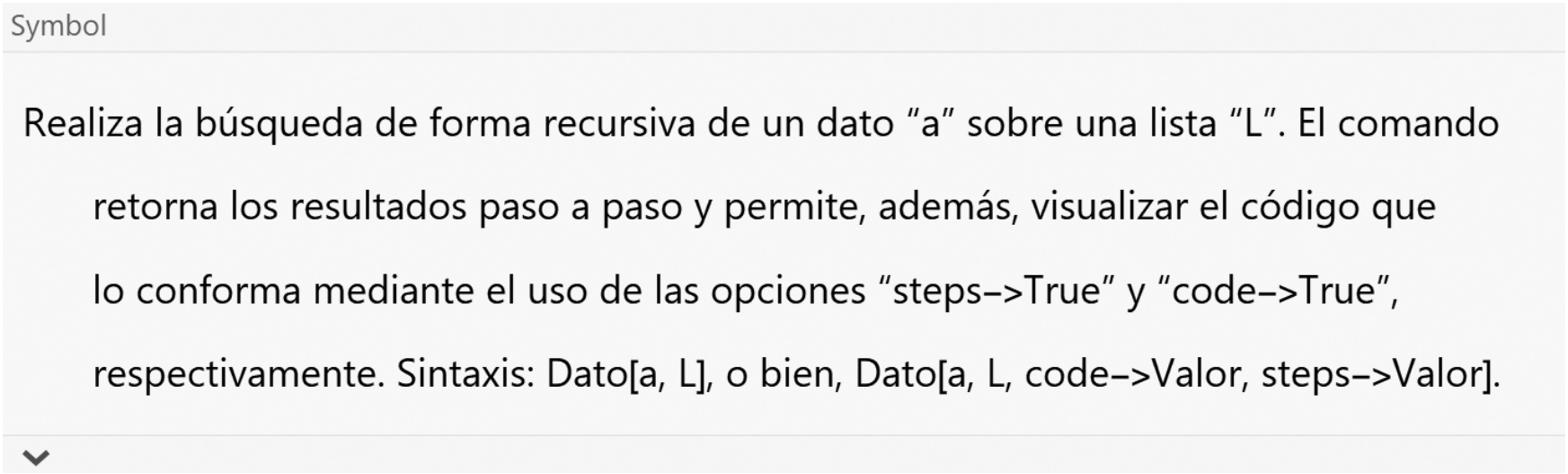


Figura: 1.1 Ayuda de *Wolfram Mathematica*

Ejemplos de programas recursivos

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Introducción

Esta sección muestra al lector un compendio de ejercicios seleccionados que utilizan la recursividad como una técnica de trabajo. Se espera con ello, lograr un mayor nivel de profundización sobre este tema.

Example (1.1)

Elabore un programa recursivo para calcular la potencia a^n, $a \in \mathbb{R}$, $n \in \mathbb{N} \cup \{0\}$, sabiendo que:

$$a^n = \begin{cases} \text{Indefinido si } a = 0 \text{ y } n = 0 \\ 0 \text{ si } a = 0 \text{ y } n \neq 0 \\ 1 \text{ si } a \neq 0 \text{ y } n = 0 \\ a \cdot a^{n-1} \text{ si } a \neq 0 \text{ y } n \neq 0 \end{cases}$$

¿Qué ocurre si n pertenece al conjunto de los números enteros?

Solución

Un programa diseñado en *Wolfram Language* para estos efectos corresponde a:

In[ ] :=

```
Potencia[a_, n_]:=If[a == 0 && n == 0, "Indefinido",
If[a == 0, 0, If[n == 0, 1, a*Potencia[a, n - 1]]]]
```

La recursividad corre apropiadamente si `n` es un número entero positivo o cero, sin embargo, al sustituir por un valor entero negativo, por ejemplo, si se ejecuta `Potencia[5,-4]`, *Mathematica* retorna lo que se aprecia en la figura 1.2.

Solución

Este **Out[ ]**, se ocasiona por un desbordamiento en la pila de llamadas, al no obtenerse nunca las condiciones de salida. La siguiente función `OtraPotencia`, consiste en una modificación de `Potencia` para solventar el cálculo con cualquier exponente entero (considerando negativos):

In[ ] :=

```
OtraPotencia[a_, n_]:=If[a == 0 && n == 0, "Indefinido",
If[a == 0, 0, If[n == 0, 1, If[n > 0,
a*OtraPotencia[a, n - 1], 1/a*OtraPotencia[a, n + 1]]]]]
```

$RecursionLimit: Recursion depth of 1024 exceeded during evaluation of 5 == 0.

$RecursionLimit: Recursion depth of 1024 exceeded during evaluation of −1025 == 0.

$RecursionLimit: Recursion depth of 1024 exceeded during evaluation of Hold[5 == 0].

General: Further output of $RecursionLimit::reclim2 will be suppressed during this calculation.

$IterationLimit: Iteration limit of 4096 exceeded.

$RecursionLimit: Recursion depth of 4096 exceeded during evaluation of MakeBoxes[Hold, StandardForm].

$RecursionLimit: Recursion depth of 4096 exceeded during evaluation of MakeBoxes[Hold[Hold[Hold[Hold[Hold[Hold[Hold[≪1≫]]]]]]], StandardForm].

$RecursionLimit: Recursion depth of 4096 exceeded during evaluation of {Hold[MakeBoxes[Hold, StandardFo[, Hold[MakeBoxes[Hold[Hold[Hold[Hold[Hold[≪1≫]]]]], StandardForm]],]}.

General: Further output of $RecursionLimit::reclim2 will be suppressed during this calculation.

Figura: 1.2 Desbordamiento en el tamaño de la pila

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-4.zip

Comando PotenciaPos

- **VilCretas** cuenta para su consulta con la instrucción `PotenciaPos` que implementa el programa recursivo del ejemplo 1.1. El comando `PotenciaPos` facilita las opciones `code -> True` y `steps -> True`.

In[ ] :=

```
PotenciaPos[5, 4, code -> True, steps -> True]
```

Comando PotenciaPos

- Del comando anterior, se obtiene la siguiente salida:

Out[ ] =

{625,Potencia[a_,n_]:=If[And[a==0, n==0],
Return["Indefinido"],If[a==0,Return[0],If[n==0,
Return[1],a*Potencia[a,n-1]]]]}
PotenciaPos[5,4] = 5*PotenciaPos[5,3] = 625
PotenciaPos[5,3] = 5*PotenciaPos[5,2] = 125
PotenciaPos[5,2] = 5*PotenciaPos[5,1] = 25
PotenciaPos[5,1] = 5*PotenciaPos[5,0] = 5
PotenciaPos[5,0] = 1
625

Comando PotenciaPos y PotenciaNeg

Se insta al lector a explorar el comando PotenciaNeg similar al anterior, pero con la funcionalidad de calcular potencias con un exponente cero o negativo.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-5.zip

Comando PotenciaPos y PotenciaNeg

Explicación en video

https://youtu.be/6Dzceb5gznI

Explicación en video

https://youtu.be/dBny-ap5kDw

Example (1.2)

Construya un programa recursivo para calcular el resultado de la sumatoria:

$$\sum_{i=5}^{n-10} \frac{\log_2(i)}{i^2+1}$$

Solución

Toda sumatoria e inclusive productoria se puede expresar de manera recursiva siempre y cuando su fórmula no dependa del parámetro contenido en el límite superior.

- $S_n = \sum_{i=5}^{n-10} \frac{\log_2(i)}{i^2+1} = \sum_{i=5}^{n-11} \frac{\log_2(i)}{i^2+1} + \frac{\log_2(n-10)}{(n-10)^2+1} =$

  $S_{n-1} + \frac{\log_2(n-10)}{(n-10)^2+1}$

Solución

La suma S_n entonces, se ha descompuesto como la suma anterior S_{n-1} adicionando el último sumando. La expresión anterior constituye la definición base de la recursividad, pero, ¿cuál sería su condición inicial?, naturalmente esa condición la conforma el primer elemento de la suma, en este caso:

- $\frac{\log_2(5)}{5^2+1} = \frac{\log_2(5)}{26}$

Solución

No siempre la condición inicial ocurre en el valor de $n = 1$. En este ejercicio de hecho, se aprecia que S_1 genera una sumatoria inconsistente pues el límite superior daría -9, un número entero menor que el límite inferior:

- $S_1 = \sum_{i=5}^{1-10} \frac{\log_2(i)}{i^2+1} = \sum_{i=5}^{-9} \frac{\log_2(i)}{i^2+1} = 0$

Solución

Por esta razón, la condición inicial no puede comenzar en $n = 1$ en este ejercicio. Para determinar el valor de n donde ocurre $\frac{\log_2(5)}{26}$, se deben igualar los límites de la sumatoria:

- $n - 10 = 5 \Rightarrow n = 15$

Este resultado indica que en $n = 15$ los límites de la sumatoria son iguales, por lo que se retornaría su primer elemento $\frac{\log_2(5)}{26}$. En conclusión, una relación de recurrencia que representa la sumatoria es:

- $$S_n = S_{n-1} + \frac{\log_2(n-10)}{(n-10)^2 + 1} \text{ con } S_{15} = \frac{\log_2(5)}{26}$$

Solución

- La relación de recurrencia anterior, se puede programar en *Wolfram Language así*:

In[ ] :=

```
Suma[n_]:=If[n == 15, Log[2, 5]/26, Suma[n - 1] +
Log[2, n - 10]/((n - 10)^2 + 1)]
```

- Si se desea tener alguna garantía sobre la correctitud de la función `Suma[n]` se tiene a disposición el comando `Sum` de *Mathematica*:

In[ ] :=

```
OtraSuma[n_]:=Sum[Log[2, i]/(i^2 + 1), {i, 5, n - 10}]
```

Solución

- `OtraSuma[n]` encuentra el resultado de la sumatoria mediante un comando propio del software. Esto significa que tanto `Suma[n]` como `OtraSuma[n]` deben devolver lo mismo. En este sentido, una verificación de ello se puede procesar utilizando la instrucción `Table` de *Wolfram Mathematica*:

In[ ] :=

```
Table[Suma[n] == OtraSuma[n], {n, 15, 50}]
```

Solución

- Se obtiene la siguiente salida:

Out[ ] $=$

{True, True}

Nota

Como se aprecia, el `Table` ejecuta treinta y seis comparaciones al tomar un rango de `n` de 15 a 50 y en todas ellas `Suma[n]` y `OtraSuma[n]` brindan el mismo resultado, de allí que la salida provista refleje treinta y seis valores lógicos True.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-6.zip`

Comando Sumatoria

La librería **VilCretas** contiene una sentencia que automatiza la construcción de un programa recursivo correspondiente al cálculo de una sumatoria. El comando se denomina `Sumatoria`. La solución del ejemplo 1.2, se podría revisar mediante el uso de `Sumatoria` como prosigue:

In[ ] :=

```
Sumatoria[{5, n - 10, Log[2, i]/(i^2 + 1)}]
```

Comando Sumatoria

Se obtiene la siguiente salida:

Out[ ] =

```
Sumatoria[n_]:=If[n==15,Log[5]/(26 Log[2]),
Sumatoria[n-1]+Log[-10+n]/((1+(-10+n)^2) Log[2])]
```

Nota

`Sumatoria` también permite el cálculo paso a paso para un valor específico de `n`. Se recomienda al alumno explorar esta posibilidad consultando la ayuda del comando mediante `?Sumatoria`.

Comando Sumatoria

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-7.zip

Explicación en video

https://youtu.be/eR0d223f5-Q

Example (1.3)

Elabore una recursividad para calcular:

$$\prod_{i=5}^{n+3} 7^{i-9}(i+8)$$

Solución

Supongamos que P_n representa la productoria inicial, entonces recurriendo a una forma de razonamiento similar a la expuesta en el ejemplo 1.2, se obtendría la definición base así:

$$P_n = P_{n-1} \cdot 7^{n+3-9}(n+3+8) = P_{n-1} \cdot 7^{n-6}(n+11)$$

Al igualar los límites de la productoria:

$$n+3 = 5 \Rightarrow n = 2$$

Solución

El primer factor de la multiplicación corresponde a $7^{5-9}(5+8) = \frac{13}{2401}$.
Finalmente, una relación de recurrencia que representa la productoria es la siguiente:

$$P_n = P_{n-1} \cdot 7^{n-6}(n+11) \text{ con } P_2 = \frac{13}{2401}$$

En lenguaje *Wolfram*:
In[ ] :=

```
Producto[n_]:=If[n == 2, 13/2401, Producto[n - 1]*
                    7^(n - 6)(n + 11)]
```

Solución

La correctitud de `Producto[n]` se puede verificar recurriendo a los comandos `Product` y `Table` de *Mathematica*, tal y como se muestra a continuación:

In[ ] :=

```
OtroProducto[n_]:=Product[7^(i - 9) (i + 8), {i, 5, n + 3}]
    Table[Producto[n] == OtroProducto[n], {n, 2, 50}]
```

Lo cual retorna un vector con cuarenta y nueve valores lógicos True.

Nota

Se aclara al lector que en el paquete **VilCretas** no existe ninguna instrucción que automatice la elaboración de un programa recursivo asociado a una productoria

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-8.zip

Example (1.4)

Construya un programa recursivo que multiplique dos números n y m enteros positivos o cero.

Solución

Si tenemos números $n, m \in \mathbb{N} \cup \{0\}$, naturalmente se aprecia que:

$$n \cdot m = n\,(m-1) + n$$

Es decir, una expresión recursiva a través de la cual se pueden multiplicar los enteros no negativos n y m es:

$$n \cdot m = \begin{cases} 0 \text{ si } n = 0 \text{ o } m = 0 \\ n\,(m-1) + n \text{ si } n \neq 0 \text{ y } m \neq 0 \end{cases}$$

Solución

En *Wolfram Language*:

In[ ] :=

```
PD[n_, m_]:=If[n == 0 || m == 0, 0, PD[n, m - 1] + n]
```

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-9.zip

Nota

La recursividad `PD[n,m]` va más allá de lo solicitado en este ejemplo, pues basta con el hecho de que `m` sea un número entero no negativo para desplegar exitosamente la respuesta. El lector preverá entonces que la función `PD` permite que `n` sea un número real cualesquiera. Por otra parte, en el paquete **VilCretas** se encuentra a disposición del usuario la instrucción `Multipli` que posee una lógica recursiva distinta a la aquí compartida. Se recomienda al alumno estudiar el código de programación de `Multipli` como otra alternativa de solución. Con esa finalidad se sugiere el uso de la opción `code -> True`.

Example (1.5)

Elabore un programa recursivo que sume los dígitos de un número entero no negativo n.

Solución

La lógica recursiva implicada requiere en la primera llamada, tomar al entero n inicial y extraer de él su primer dígito d. En *Wolfram Mathematica*, `Mod[n, 10]` devuelve el residuo de la división $n \div 10$, es decir, el primer dígito d que posee n. Ese valor en nuestro programa recursivo, se va a sumar literalmente a la siguiente invocación donde se utilizará para esa llamada, el número entero resultante de eliminar el dígito ya acumulado, dicho número corresponde a $\frac{(n-d)}{10}$. El procedimiento se repite (con cada llamada), hasta que el valor de $\frac{(n-d)}{10}$ se reduce a cero.

Solución

Una implementación de esto en *Mathematica* se presenta a continuación:
In[ ] :=

```
SD[n_]:=Module[{digito = Mod[n, 10]},
If[n == 0, 0, digito + SD[(n - digito)/10]]]
```

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-10.zip

Nota

La librería **VilCretas** cuenta con el comando `SumaDigi` correspondiente a la función recursiva `SD` anterior, con la particularidad de posibilitar la visualización de la pila de llamadas. Se invita al lector a utilizar la opción `steps -> True` de `SumaDigi`.

Example (1.6)

Sean n y m enteros positivos con $n \leq m$. Realice un programa recursivo que muestre todos los múltiplos de n menores o iguales a m.

Solución

En este ejemplo se desean obtener todos los múltiplos de un entero positivo n, para ello se debe multiplicar n por un contador iniciando en 1. El contador requerido no se puede colocar dentro del programa recursivo de interés, pues de ser así, en cada llamada, se inicializaría esa variable en 1, ocasionando un efecto no deseado. Por esta razón y en general, cuando se necesita un contador en un programa recursivo, una forma de trabajo, reside en colcar el contador como un parámetro más de la función.

Solución

Partiendo de lo descrito anteriormente:

In[ ] :=

```
Mult[n_, m_, Cont_ : 1]:=If[n*Cont <= m, Print[n*Cont];
                    Mult[n, m, Cont + 1]]
```

Por ejemplo, si se corre `Mult[5,36]`:

In[ ] :=

```
Mult[5, 36]
```

Solución

La salida obtenida es la siguiente:

Out[ ] =

5
10
15
20
25
30
35

Solución (Alternativa)

El estudiante debe notar que en `Mult[5,36]` no fue necesario incluir el valor del contador pues por defecto inicia en 1. La salida de este código se ha sustentado en enviar a imprimir en pantalla el múltiplo correspondiente. Otra solución, consistiría en ir añadiendo los múltiplos en una lista, en lugar de imprimirlos cada vez.

Solución (Alternativa)

En *Wolfram Mathematica* esta alternativa se implementaría como sigue:

In[ ] :=

```
OtroMult[n_, m_, Cont_ : 1]:=Module[{}, If[Cont == 1,
  L = {}]; If[n*Cont <= m, L = Append[L, n*Cont];
            OtroMult[n, m, Cont + 1]]; L]
```

Luego, al ejecutar `OtroMult[5,36]`:

In[ ] :=

```
OtroMult[5, 36]
```

Out[ ] =

$$\{5, 10, 15, 20, 25, 30, 35\}$$

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-11.zip

Nota

`Append` es una instrucción propia del software *Mathematica* encargada de añadir al final de una lista sobre la que se opera, un elemento. A este respecto, recibe dos argumentos, la lista y el término a agregar. También, *Mathematica* presenta el comando `Prepend` con un funcionamiento similar a `Append`, con la diferencia de añadir al inicio de la lista lo deseado y no al final. Se insta al lector a modificar el programa `OtroMult` usando `Prepend` en sustitución de `Append`.

Sentencia Multiplos

El paquete **VilCretas** cuenta con la instrucción `Multiplos` similar a la función `OtroMult` expuesta en el ejemplo 1.6. El aporte principal de `Multiplos` reside en ofrecer al usuario las opciones acostumbradas `code -> True` y `steps -> True`. Por ejemplo:

In[ ] :=

```
Multiplos[5, 36, code -> True, steps -> True]
```

Sentencia Multiplos

Se obtiene la salida:

Out[ ] =

```
{{5,10,15,20,25,30,35},
Multiplos[n_,m_,d_]:=Module[{i=d},If[i==1,L={};
If[n*i<m,L=Append[L,n*i]];
i++;Multiplos[n,m,i],If[n*i<m,L=Append[L,n*i];i++;
Multiplos[n,m,i],Return[L]]]]}
Multiplos[5,36,1] = {5}
Multiplos[5,36,2] = {5,10}
Multiplos[5,36,3] = {5,10,15}
Multiplos[5,36,4] = {5,10,15,20}
Multiplos[5,36,5] = {5,10,15,20,25}
Multiplos[5,36,6] = {5,10,15,20,25,30}
Multiplos[5,36,7] = {5,10,15,20,25,30,35}
{5, 10, 15, 20, 25, 30, 35}
```

Sentencia Multiplos

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-12.zip

Explicación en video

https://youtu.be/9Rqlop9qdvg

Example (1.7)

Construya un programa recursivo que genere en una lista, todos los divisores de un entero positivo n.

Solución

Para resolver este ejercicio se empleará una forma de razonamiento similar a la establecida en el ejemplo 1.6. Es decir, se requiere el uso de un contador con valor inicial 1 y valor final n, esto con el objetivo de ir extrayendo todos los divisores de n.

Solución

Luego:

In[ ] :=

```
divisores[n_, Cont_ : 1, Lista_ : {}]:=
Module[{L = Lista}, If[Cont <= n, If[Mod[n, Cont] == 0,
L = Append[L, Cont]]; divisores[n, Cont + 1, L], Lista]]
```

En este caso, a diferencia de lo resuelto en el ejemplo 1.6, se ha inicializado la lista en vacío al añadirla como un parámetro más de la función. El lector puede corroborar el funcionamiento de `divisores`, ejecutando:

In[ ] :=

```
divisores[36]
```

Solución

Ejecutando el comando anterior, se obtiene la siguiente salida:
Out[ ] =

$$\{1, 2, 3, 4, 6, 9, 12, 18, 36\}$$

Nota

El programa recursivo `divisores` se puede mejorar en tiempo de ejecución, analizando los posibles divisores hasta la mitad del número n y agregando a la última lista obtenida ese valor. Lo anterior se justifica pues ningún divisor propio (divisores menores que n) será mayor que $n \div 2$.

Solución (Mejorada)

El programa recursivo mejorado corresponde a:

In[ ] :=

```
Otrodivisores[n_, Cont_ : 1, Lista_ : {}]:=
Module[{L = Lista}, If[Cont <= n/2, If[Mod[n, Cont] == 0,
  L = Append[L, Cont]]; Otrodivisores[n, Cont + 1, L],
                    Append[Lista, n]]]
```

Solución

En el capítulo 3 se verificará que `Otrodivisores` es más eficiente en comparación con `divisores`.
En la librería **VilCretas** el comando `Divisores` constituye una herramienta de comprobación para visualizar la pila de llamadas de `Otrodivisores`.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-13.zip`

Example (1.8)

Elaborar una recursividad que convierta un número n en base binaria a base diez.

Solución

La conversión de binario a decimal se puede efectuar tomando de manera recursiva cada dígito binario y multiplicando ese valor por dos elevado a un contador que debe inicializarce en cero. De esta manera, se acumula en una suma, en cada llamada, el producto ya citado con la nueva invocación. El alumno es importante que observe en esta lógica de programación, una combinación de las ideas presentadas en la solución de los ejemplos 1.5 y 1.6.

Solución

Veamos:

In[ ] :=

```
BToD[n_, Cont_ : 0]:=Module[{digito = Mod[n, 10]},
If[n == 0, 0, 2^Cont*digito + BToD[(n - digito)/10,
Cont + 1]]]
```

Se sugiere correr en *Wolfram Mathematica* `BToD[1111]` cuya salida es igual a 15.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-14.zip

Example (1.9)

Sean n y m enteros positivos o cero. Utilice el algoritmo de *Euclídes* para crear una recursividad que calcule el máximo común divisor entre n y m.

Solución

El algoritmo de Euclídes establece que:

$$MCD\,(n, m) = MCD\,(m, r)$$

siendo MCD el máximo común divisor y r el residuo de la división $n \div m$. Este procedimiento implica implícitamente una recursividad, pues la idea reside en aplicarlo sucesivamente hasta que r se reduzca a cero. En ese punto, por una propiedad del máximo común divisor, se retornaría el primer parámetro de MCD que contendría al cálculo buscado.

Solución

En *Mathematica*:

In[ ] :=

```
mcd[n_, m_]:=If[m == 0, n, mcd[m, Mod[n, m]]]
```

Por ejemplo:

In[ ] :=

```
mcd[3550, 600]
```

Out[ ] =

50

Nota

En el paquete **ViCretas** la instrucción `MCD` constituye una pequeña variante de la solución ya compartida en este ejemplo. Se espera que el estudiante la analice, usando las opciones `code -> True` y `steps -> True`. Además, con la intención de realizar distintas pruebas de verificación, *Mathematica* provee el comando `GDC[n,m]`, encargado de calcular el máximo común divisor entre `n` y `m`.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-15.zip

Example (1.10)

Investigue en qué consiste el algoritmo *quicksort* para ordenar los elementos de una lista L. Implemente en *Wolfram Mathematica* esta recursividad.

Reseña histórica del algoritmo *quicksort*

Quicksort se desarrolló en el año de 1960 por el científico inglés *C.A.R. Hoare*. En la actualidad es uno de los algoritmos de ordenación más difundidos por su nivel de efectividad. El algoritmo se basa en las siguientes instrucciones:

1. Se selecciona un elemento de la lista L a ordenar denominado *pivote*.
2. Se ordenan los elementos alrededor del *pivote* de tal forma que a su izquierda queden los datos menores que él y a su derecha los mayores. La lista se divide en dos sublistas: la de la izquierda y la de la derecha con respecto al *pivote*.
3. Se llama a este algoritmo para ordenar la sublista izquierda.
4. Se llama a este algoritmo para ordenar la sublista derecha.

Nota

El paso 2 tiene diversas formas de implementación, la más usual consiste en jugar con dos índices i y j, donde el parámetro i se mueve de izquierda a derecha y el parámetro j de derecha a izquierda. El razonamiento se basa en comparar los elementos $L[i]$ y $L[j]$ de la lista L con el *pivote*. Si $L[i]$ es mayor que el *pivote* y a su vez $L[j]$ es menor que el *pivote*, se intercambia $L[i]$ con $L[j]$ y se repite el proceso incrementando i y decrementando j hasta que los índices i y j se cruzan.

Solución

Un programa en *Wolfram Mathematica* que implementa el algoritmo *quicksort* es el siguiente:

In[ ] :=

```
quicksort[begin_ : 1, end_]:=Module[{piv, aux},
If[begin < end, i = begin; j = end; piv =
L[[Floor[(begin + end)/2]]]; While[i <= j,
While[L[[i]] < piv, i++]; While[L[[j]] > piv, j--];
If[i <= j, aux = L[[i]]; L[[i]] = L[[j]]; L[[j]] = aux;
i++; j--]]; Print[ReplacePart[L, Style[L[[Floor[(begin
+ end)/2]]], Blue], Floor[(begin + end)/2]]];
If[begin < j, quicksort[begin, j]]; If[i < end,
quicksort[i, end]]]; L]
```

Solución

Se observa en el código, que el elemento *pivote* se toma como un punto central de la lista L, esto lo permite el cálculo de `Floor[(begin+end)/2]` que representa la parte entera de la suma de la posición mínima con la posición máxima de la lista, dividido entre dos.

In[ ] :=

```
L = {-9, 5, -7, 0, 10, 21.3, -28.7, 100, -100, 14};
quicksort[Length[L]]
```

Solución

De la entrada anterior, se obtiene la siguiente salida:

Out[ ] =

{-9,5,-7,0,-**100**,-28.7,21.3,100,10,14}
{-9,-28.7,-**100**,0,-7,5,21.3,100,10,14}
{-100,-**28.7**,-9,0,-7,5,21.3,100,10,14}
{-100,-28.7,-9,-**7**,0,5,21.3,100,10,14}
{-100,-28.7,-**9**,-7,0,5,21.3,100,10,14}
{-100,-28.7,-9,-7,**0**,5,21.3,100,10,14}
{-100,-28.7,-9,-7,0,5,21.3,**14**,10,100}
{-100,-28.7,-9,-7,0,5,**10**,14,21.3,100}
{-100,-28.7,-9,-7,0,5,**10**,14,21.3,100}
{-100,-28.7,-9,-7,0,5,10,**14**,21.3,100}
{-100,-28.7,-9,-7,0,5,10,14,**21.3**,100}
{-100, -28.7, -9, -7, 0, 5, 10, 14, 21.3, 100}

Solución

En *Mathematica*, lo mostrado en el **Out[ ]** anterior con negrita, toma en realidad un color azul, indicando con ello, en cada llamada de `quicksort`, cuál es el elemento *pivote* seleccionado.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-16.zip`

Comando Quicksort

El paquete **VilCretas** posee una implementación del algoritmo recursivo *quicksort* a través de la sentencia `Quicksort`. Ella además de poseer las opciones `code -> True` y `steps -> True`, proporciona la posibilidad de ordenar de forma descendente empleando `ascendente -> False`. Por ejemplo:

In[ ] :=

```
Quicksort[{-9, 5, -7, 0, 10, 21.3, -28.7, 100, -100, 14},
code -> True, steps -> True, ascendente -> False]
```

Comando Quicksort

Del comando anterior, se obtiene la siguiente salida:

Out[ ] =

{14,100,21.3,10,**0**,-7,-28.7,5,-100,-9}
{100,**14**,21.3,10,0,-7,-28.7,5,-100,-9}
⋮
{{100,21.3,14,10,5,0,-7,-9,-28.7,-100},
Quicksort[begin_,end_]:=Module[{},...
Quicksort[1,10] =
{-9,5,-7,0,**10**,21.3,-28.7,100,-100,14}
⋮

Comando Quicksort

Continuación de la salida:

⋮
Quicksort[9,10] = {-**28.7**,-100}
{-**28.7**,-100}
{-28.7,-100}
L = {100,21.3,14,10,5,0,-7,-9,-28.7,-100}
{100, 21.3, 14, 10, 5, 0, -7, -9, -28.7, -100}

No se presenta la salida total por su amplio tamaño, pero se insta al estudiante a generar el **Out[ ]** completo en el software *Wolfram Mathematica*.

Comando Quicksort

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-17.zip

Explicación en video

https://youtu.be/VBlD-joFplI

1.4 Ejemplos de recursividades de cola

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Introducción

Las recursividades de cola tienen la ventaja de aprovechar las salidas proporcionadas por las invocaciones anteriores en un programa recursivo, mejorando significativamente el tiempo requerido por el programa para obtener un resultado exitoso y evitando en muchos casos, además, un desbordamiento en el tamaño de la pila de llamadas.

Example (1.11)

Calcule el factorial de un número entero no negativo n mediante una recursividad de cola.

Solución

En este ejercicio se construirá una función factorial auxiliar que poseerá dos parámetros: el primero contrala la salida del programa y el segundo reutiliza la llamada anterior para ir acumulando allí, el resultado deseado del factorial. En *Wolfram Mathematica*:

In[ ] :=

```
factorialesCola[n_]:=Module[{FactorialesAux},
    FactorialesAux[p_, m_]:=If[p == 0, m,
FactorialesAux[p - 1, p*m]]; FactorialesAux[n, 1]]
```

Solución

Un recorrido paso a paso de la pila de llamadas en `factorialesCola` se puede obtener al usar la instrucción de **VilCretas** denominada `FactorialesCola`.

In[ ] :=

```
FactorialesCola[5, code -> True, steps -> True]
```

Solución

Se obtiene la siguiente salida:

Out[ ] =

{120,FactorialesCola[n_]:=Module[{FactorialesAux},
FactorialesAux[p_,m_]:=If[p==0, m, FactorialesAux[p-1, p*m]];
FactorialesAux[n, 1]]}
FactorialesAux[5,1] = FactorialesAux[4,5*1] = FactorialesAux[4,5]
FactorialesAux[4,5] = FactorialesAux[3,4*5] = FactorialesAux[3,20]
FactorialesAux[3,20] = FactorialesAux[2,3*20] = FactorialesAux[2,60]
FactorialesAux[2,60] = FactorialesAux[1,2*60] = FactorialesAux[1,120]
FactorialesAux[1,120] = FactorialesAux[0,1*120] = FactorialesAux[0,120]
FactorialesAux[0,120] = 120
120

Solución

El lector observará que en la última invocación FactorialesAux[0,120], el segundo argumento ya tiene calculado el resultado de 5!. Lo anterior significa que a diferencia de la recursividad de pila desarrollada en la sección 1.1, `factorialesCola` no requiere realizar una sustitución hacia arriba al darse la condición de salida. Naturalmente, esto ahorra tiempo y recursos en el ordenador.

Nota

Prueba de ello, reside en ejecutar `factorialesCola[1500]` que sí retorna un cálculo exitoso en comparación con `factoriales[1500]` donde se manifiesta un desbordamiento.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-18.zip`

Example (1.12)

Determine una recursividad de cola para la siguiente relación de recurrencia:

$$a_n = 2a_{n-1} + n^2 \text{ con } a_3 = 4$$

Solución

Al igual que en el ejemplo 1.11 se requiere el uso de una función auxiliar, veamos:

In[ ] :=

```
RecurrenciaCola[n_]:=Module[{RecurrenciaAuxiliar},
RecurrenciaAuxiliar[m_, Cont_ : 3, a_]:=If[m == 3, 4,
  If[Cont == m, a, RecurrenciaAuxiliar[m, Cont + 1,
  2 a + (Cont + 1)^2]]]; RecurrenciaAuxiliar[n, 4]]
```

Solución

En `RecurrenciaCola` el contador `Cont` se inicializa en el subíndice donde comienza la condición inicial de la relación de recurrencia. El tercer parámetro de `RecurrenciaAuxiliar` se utiliza para ir acumulando allí, el $m-$ésimo término de la sucesión, esto con la finalidad de retornar ese argumento al darse la condición de salida del programa, evitando así, una sustitución consecutiva hacia arriba, como sucedería si se hubiese resuelto a_n por medio de una recursividad de pila.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-19.zip

Example (1.13)

Elabore un programa recursivo de cola que genere los elementos de la sucesión de *Fibonacci*.

Solución

La función auxiliar de esta recursividad debe recibir cuatro parámetros: el $m-$ésimo valor de *Fibonacci* a encontrar, un contador que permita establecer el momento en el cuál ese término se ha hallado, una variable a_1 y otra a_2, donde en a_2 se calcula el número de *Fibonacci* de interés. Se necesitan dos variables a_1 y a_2 pues la relación de recurrencia que construye esta sucesión es de orden dos, es decir, a_n depende de a_{n-1} y a_{n-2} en la definición base de la recursividad ($a_n = a_{n-1} + a_{n-2}$).

Solución

En *Wolfram Mathematica*:

In[ ] :=

```
fibonacciCola[n_]:=Module[{FibonacciAux},
FibonacciAux[m_, Cont_ : 2, a1_, a2_] :=
If[m == 1 || m == 2, 1, If[Cont == m, a2,
FibonacciAux[m, Cont + 1, a2, a2 + a1]]];
FibonacciAux[n, 1, 1]]
```

Solución

`FibonacciAux` emplea la fórmula recursiva $a_n = a_{n-1} + a_{n-2}$ al invocarse a sí misma en la expresión `FibonacciAux[m, Cont + 1, a2, a2 + a1]` donde `a2 + a1` proviene precisamente de esa ecuación. Además, el lector debe apreciar cómo `fibonacciCola[n]` llama a `FibonacciAux[n, 1, 1]`, los unos en los argumentos de la invocación se refieren a las condiciones iniciales $a_1 = 1$ y $a_2 = 1$ de la relación de recurrencia, respectivamente. `fibonacciCola` es un programa más eficiente que la función `fibonacci` compartida en la sección 1.1.

Nota

En la librería **VilCretas** se cuenta con el comando `NFibonacciCola` siendo éste, una implementación de `fibonacciCola` con las propiedades `code -> True` y `steps -> True`.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-20.zip

Example (1.14)

Construya una recursividad de cola para calcular:

$$a_n = a_{n-3} - 5a_{n-2} + 2a_{n-1} + 4n - 4 \text{ con } a_2 = -2,\ a_3 = 5 \text{ y } a_4 = \sqrt[3]{3}$$

Solución

Antes de iniciar el proceso de resolución se sugiere al alumno ordenar de manera descendente los sumandos de la relación de recurrencia dada, es decir:

$$a_n = 2a_{n-1} - 5a_{n-2} + a_{n-3} + 4n - 4 \text{ con } a_2 = -2,\ a_3 = 5 \text{ y } a_4 = \sqrt[3]{3}$$

Solución

Tomando como base las ideas expuestas en el ejemplo 1.13, se ha elaborado el siguiente programa recursivo:

In[ ] :=

```
RecurrenciaCola[n_]:=Module[{RecurrenciaAuxiliar},
RecurrenciaAuxiliar[m_, Cont_ : 4, a2_, a3_, a4_] :=
If[m == 2, -2, If[m == 3, 5, If[m == 4, Power[3, (3)^-1],
If[Cont == m, a4, RecurrenciaAuxiliar[m, Cont + 1, a3,
a4, 2 a4 - 5 a3 + a2 + 4 (Cont + 1) - 4]]]]];
RecurrenciaAuxiliar[n, -2, 5, Power[3, (3)^-1]]]
```

Solución

La llamada recursiva de `RecurrenciaAuxiliar` toma como quinto parámetro `2 a4 - 5 a3 + a2 + 4 (Cont + 1) - 4` pues se consideró en la definición base $a_n = 2a_{n-1} - 5a_{n-2} + a_{n-3} + 4n - 4$ la sustitución de a_{n-1} por a4, a_{n-2} por a3 y finalmente, a_{n-3} por a2. Se realiza de esa manera debido a que la ecuación recursiva tiene un orden descendente y por lo tanto, las variables a2, a3 y a4 se deben reemplazar de forma descendente también, seleccionando y sustituyendo primero a a4, luego a a3 y posteriormente a a2 en a_n. Además, el valor de n en la expresión $4n - 4$ se ha reemplazado por `Cont + 1` al llevar esa suma el conteo del $n-$ésimo elemento de la sucesión que se ha calculado.

Solución

Una verificación sobre la correctitud de `RecurrenciaCola` en *Mathematica*, se muestra a continuación:

In[ ] :=

```
a[n_] := 2 a[n - 1] - 5 a[n - 2] + a[n - 3] + 4 n - 4
a[2] = -2;
a[3] = 5;
a[4] = Power[3, (3)^-1];
Table[a[i] == RecurrenciaCola[i], {i, 2, 15}]
```

Out[ ] =

{True, True, True, True, True, True, True, True, True, True, True, True, True, True}

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/File-21.zip

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Cuadernos/Recursividad.pdf.rar

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/Quiz_recursividad.rar

Abra un sitio web

https://www.symbaloo.com/mix/vilcretasrecursividad

¡Recuerde resolver los ejercicios asignados!

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Recursividad/Excercises.zip

enrique.vilchez.quesada@una.cr
http://www.escinf.una.ac.cr/discretas