Universidad Nacional de Río Cuarto

Facultad de Cs. Exactas Físico-Químicas y Naturales

Departamento de Computación

Introducción a la Algorítimica y Programación (3300)

Recursividad

Joaquín Pablo Tissera

2022

Recursividad

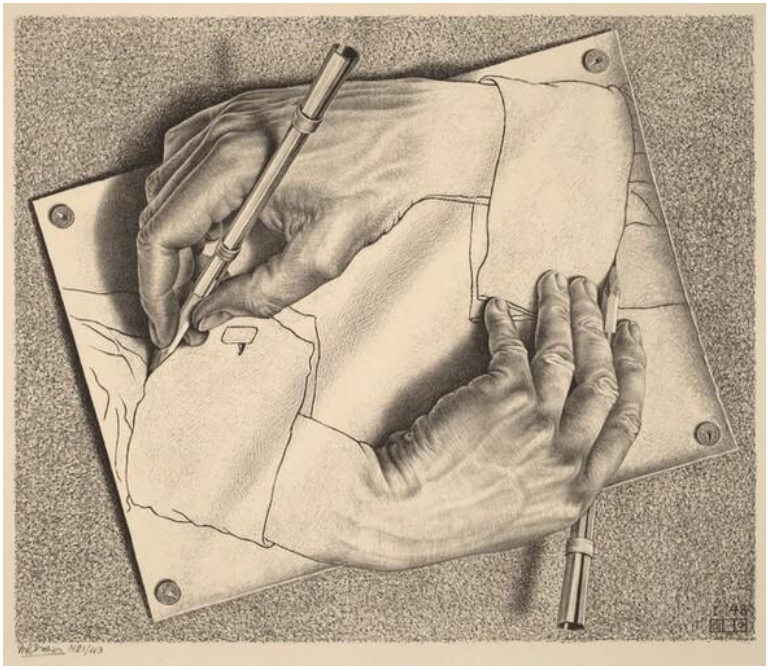
· Definición sobre sí mismo.Definición sobre sí mismo.Definición sobre sí mismo.Definición sobre sí mismo.Definición sobre sí mismo.Definición sobre sí mismo.

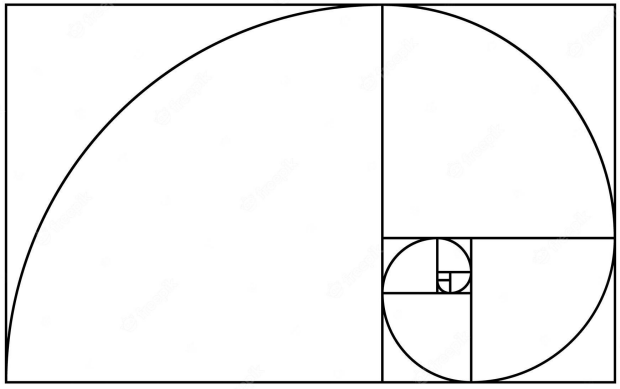
· Idea instantánea del infinito.

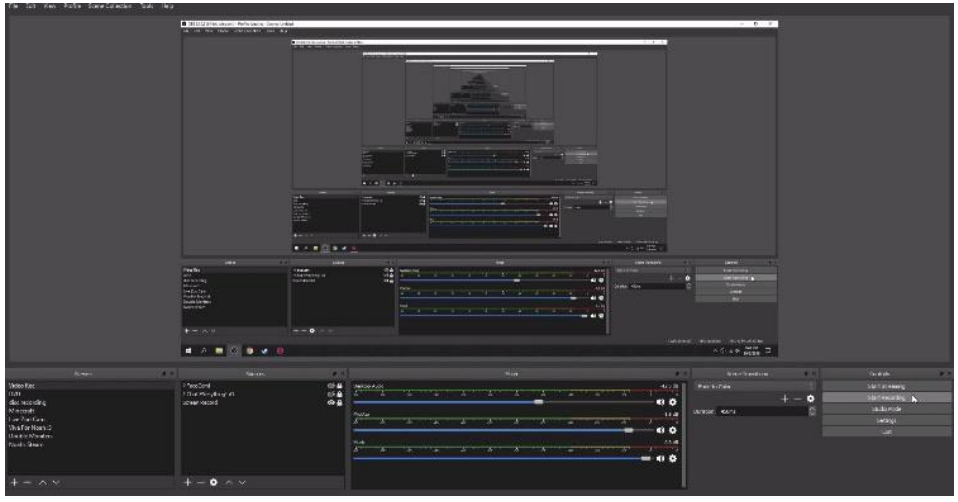
· Origen en la misma naturaleza.

· Aplicaciones en diversas ciencias y artes.

Recursividad







Paradigmas de Programación

- Orientado a objetos [Smalltalk, C++, Eifell, Java]
- Imperativo [Fortran, Cobol, Algol, PL/I, Pascal, C, Modula-2, Ada]
- Declarativo
  - Funcional [Lisp, ISWIN, Scheme, FP, Hope, ML, Miranda, Haskell, Gofer]
  - Lógico [Prolog]

Imperativo	Funcional
· Las estructuras iterativas como solución primitiva · Se puede utilizar la recursividad capturando su concepto · Solo podremos hacer uso de ella en módulos y bajo el principio de inducción matemático	· La recursividad como solución primitiva · Concepto de función como eje central · No existe el cambio de estado, se caracteriza por el uso de expresiones y funciones

Principio de inducción matemático

Dos partes a cumplir:

- **Caso base**

  Valor o valores conocidos, puede haber más de un caso base

  Se deja de recursar; ya no se vuelve a invocar el módulo

  Ahora las recursiones (en aumento) pendientes comienzan a resolverse

- **Etapa inductiva**

  Satisfacer esta condición supone querer hacer la llamada recursiva

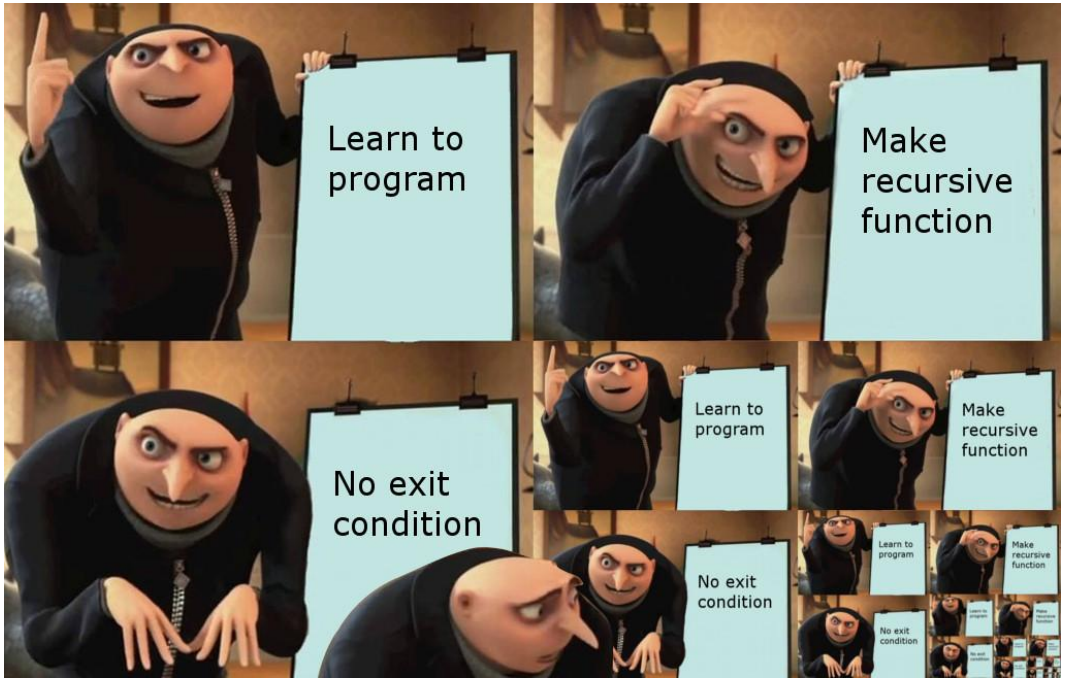
  Se deben mover los valores de las variables involucradas en la condición, acercándose al caso base

  Si la recursión es en aumento, quedarán pendientes de que alguno de los llamados alcance el caso base

¿Y si falta o falla alguna?

Sin entrar o sin tener el <u>caso base</u> recursamos infinitamente, haciendo que no sea un *algoritmo genuino*.

Sin entrar o sin tener una <u>etapa inductiva</u> no recreamos la recursividad, olvidando el *propósito* y el *concepto* de recursión.



"Una solución recursiva será aceptable cuando ya no necesite llamarse a sí misma."

Dos tipos de recursividad

Recursividad de cola		Recursividad en aumento	
Ventajas	Desventajas	Ventajas	Desventajas
· Menor consumo de memoria por mantener constante el uso de la *pila de activación** · No deja cálculos pendientes: siempre va obteniendo un resultado parcial/final	· Solución más dificil de entender y no tan natural como las de aumento	· Mayor legibilidad que la recursión de cola · Si una solución es naturalmente recursiva, hacerla en aumento beneficiará la legibilidad	· Mayor consumo de memoria por el uso de la *pila de activación** · Sólo se obtiene el resultado final luego de alcanzar el caso base

* Llamaremos pila de activación al orden en que el procesador ejecuta las acciones, haciendo uso de los espacios de memoria.

Cuando quedan cálculos pendientes como en la recursión por aumento, se reservará el lugar en la pila de la llamada recursiva para cuando se obtenga un resultado final (llegando al caso base) y luego se va desapilando, resolviéndose del último llamado al primero de todos.

En las recursiones de cola se hace uso de algún espacio de memoria cuando inicia la recursión, pero como ya va obteniendo resultados parciales/finales, se van resolviendo los cálculos.

Características de los módulos

· Las Acciones siempre son recursivas de cola

· El caso base de las acciones suele ser la acción *nada* o valores neutros

(res ← res+0), (res ← res*1)

· Las Funciones admiten recursión en aumento o en cola

· El caso base de las funciones suele ser devolver un valor conocido, inicial o un parámetro entrante.

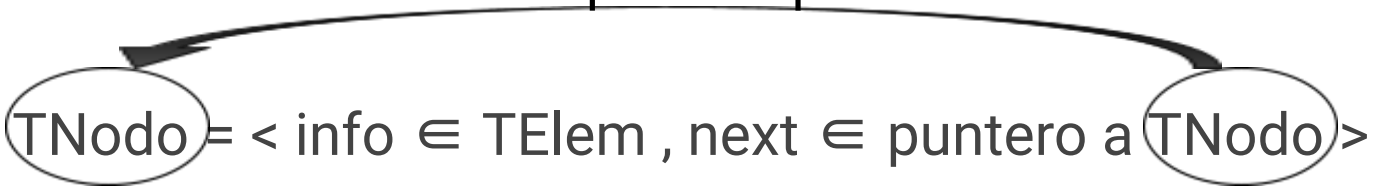
(←1), (←0), (←res)

· Toda *función* recursiva puede traducirse a acción pero no toda acción puede traducirse a *función*

· Se pueden dejar cálculos pendientes en cualquier tipo de recursión poniendo sentencias luego de la llamada recursiva, porque aún no pueden ser ejecutadas hasta terminar de recursar

Datos recursivos

Las listas simples o dobles son de naturaleza recursiva, pues por como las declaramos decimos que son punteros a ellas mismas:

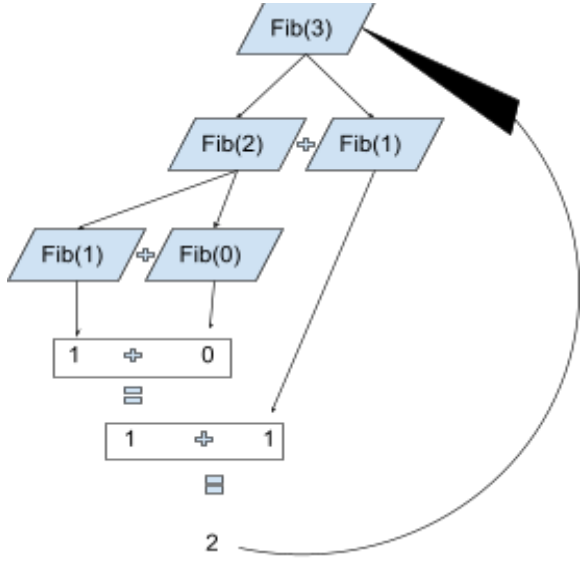


Da la pauta para decir que pueden ser recorridas de forma naturalmente recursiva.

El caso base cuando se recorra una lista será haber llegado a la dirección especial *nil*, pues es la única manera de saber que se terminó la secuencia.

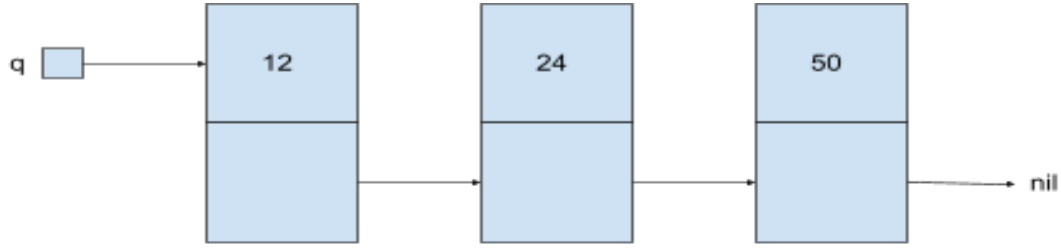
Ejemplos: Recursión en aumento

```
{Pre: nroSec > 0}
Función Fibonacci (dato nroSec ∈ ℤ) → ℤ
Inicio
  Si ( nroSec = 0 ∨ nroSec = 1 ) entonces
    ← nroSec
  sino
    ← Fibonacci(nroSec-1) + Fibonacci(nroSec-2)
  Fsi
Ffunción
```



Fib(3) = 2

```
Función SumaLista(dato q ∈ puntero a TNodo) → ℤ
Inicio
  Si ( q = nil ) entonces
    ← 0
  sino
    ← (^q).info + SumaLista((^q).next)
  Fsi
Ffunción
```



SumaLista(q) = 12 + SumarLista((^q).next)¹
SumarLista(q)¹ = 24 + SumarLista((^q).next)²
SumarLista(q)² = 50 + SumarLista((^q).next)³
SumarLista(q)³ = 0

SumarLista(q)² = 50 + 0
SumarLista(q)¹ = 24 + 50
SumaLista(q) = 12 + 74
SumaLista(q) = 86

Ejemplos: Recursión en cola

```
Acción TodosPares(dato q ∈ TArreglo, cant ∈ (0..NMax), dato/resultado pares ∈ Lógico)
Inicio
  Según
    ( cant = 0 ) : nada
    ( cant ≥ 1 ∧ ( q[cant] mod 2 ≠ 0 ) ) : pares ← Falso
    ( cant ≥ 1 ∧ ( q[cant] mod 2 = 0 ) ) : pares ← Verdadero
                                            TodosPares(q, cant-1, pares)
  Fsegún
Facción
```

```
{Pre: sum=0}
Función SumaLista(dato q ∈ puntero a TNodo, sum ∈ ℤ) → ℤ
Inicio
  Si ( q = nil ) entonces
    ← sum
  sino
    ← SumaLista((^q).next, sum+(^q).info)
  Fsi
Ffunción
```

Traducción a C

```
int CantParesCola(struct TNodo *q, int cant); // Pre: cant = 0 y LSE s/fict

int CantParesCola(struct TNodo *q, int cant){
  if ( q == NULL ){
    return cant;
  }else{
    if ( ( q->info % 2 ) == 0 ){
      cant++;
    }
    return CantParesCola(q->next, cant);
  }
}
```

```
int CantParesAumento(struct TNodo *q);  // LSE s/fict

int CantParesAumento(struct TNodo *q){  // LSE s/fict
  if ( q == NULL ){
    return 0;
  }else{
    if ( ( q->info % 2 ) == 0 ){
      return 1 + CantParesAumento(q->next);
    }else{
      return CantParesAumento(q->next);
    }
  }
}
```

Traducir una función o acción recursiva a C sale de manera directa, solo basta conocer las demás nociones; pasajes de parámetros, tipo del retorno de la función, retorno vacío de las acciones, etc.

(La presentación llegó al caso base)

Fin de la presentación

¡Gracias por su atención!