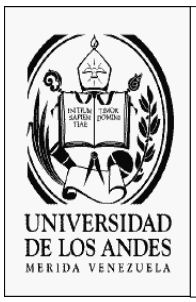
PROGRAMACION II. RECURSIVIDAD



UNIVERSIDAD DE LOS ANDES
NUCLEO UNIVERSITARIO RAFAEL RANGEL (NURR)
DEPARTAMENTO DE FISICA Y MATEMATICA
AREA COMPUTACION
TRUJILLO EDO. TRUJILLO

Recursividad

PROFESOR: JOSE DELPHIN

PROGRAMACION II. RECURSIVIDAD

Recursividad:
La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre. El ejemplo más utilizado por su fácil comprensión es el cálculo de números factoriales. El factorial de 0 es, por definición, 1. Los factoriales de números mayores se calculan mediante la multiplicación de 1 * 2 * ..., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial.

Un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int Factorial(int n);

int main(){
        int valor;

        system("clear");
        cout << "Introduzca numero a calcular: ";
        cin >> valor;

        cout << "\nEl Factorial de " << valor << " es: " << Factorial(valor) << endl;
        return 0;
}

int Factorial(int n){
        if (n < 0){
           cout << “No existe el factorial de un numero negativo.\n”;
        }else if(n < 2){
           return 1;
        }else
           return n * Factorial(n-1);
}
```

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden N, cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que N. De esta forma, al ir reduciendo progresivamente la complejidad del problema a resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva). En esa situación diremos que estamos ante un caso base de la recursividad.

Es frecuente que los algoritmos recurrentes sean más ineficientes en tiempo que los iterativos aunque suelen ser mucho más breves en espacio.

Recursividad directa vs indirecta.

Cuando en una subrutina hay llamadas a ella misma se habla de **recursividad directa**, en contraposición, cuando se tienen varias subrutinas y éstas se llaman unas a otras formando ciclos se dice que la recursión es **indirecta**.

```
Subrutina_A → Subrutina_A → Subrutina_A
Subrutina_A → Subrutina_B → Subrutina_C → Subrutina_D → Subrutina_A
```

Propiedades de las definiciones o algoritmos recursivos:

- No debe generar una secuencia infinita de llamadas así mismo, dicho de otro modo ha de existir al menos un caso base.

- Una función recursiva *f* debe definirse en términos que no impliquen a *f* al menos en un argumento o grupo de argumentos.
- Debe existir una "salida" de la secuencia de llamadas recursivas.
- Cada llamada recurrente se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).

Programación Recursiva:
Es mucho mas difícil desarrollar una solución recursiva en un lenguaje determinado para resolver un problema especifico cuando no se tiene un algoritmo. No es solo el programa sino las definiciones originales y los algoritmos los que deben desarrollarse. En general, cuando encaramos la tarea de escribir un programa para resolver un problema no hay razón para buscar una solución recursiva. La mayoría de los problemas pueden resolverse de una manera directa usando métodos no recursivos. Sin embargo, otros pueden resolverse de una manera mas lógica y elegante mediante la recursión.

Volviendo a examinar la función factorial. El factor es, probablemente, un ejemplo fundamental de un problema que no debe resolverse de manera recursiva, dado que su solución iterativa es directa y simple. Sin embargo, examinaremos los elementos que permiten dar una solución recursiva. Antes que nada, puede reconocerse un gran número de casos distintos que se deben resolver. Es decir, quiere escribirse un programa para calcular 0!, 1!, 2! Y así sucesivamente. Puede identificarse un caso "trivial" para el cual la solución no recursiva pueda obtenerse en forma directa. Es el caso de 0!, que se define como 1. El siguiente paso es encontrar un método para resolver un caso "complejo" en términos de uno mas "simple", lo cual permite la reducción de un problema complejo a uno mas simple. La transformación del caso complejo al simple resultaría al final en el caso trivial. Esto significaría que el caso complejo se define, en lo fundamental, en términos del mas simple.

Ejemplo de la sucesión de Fibonacci

En matemáticas, la sucesión de Fibonacci es la siguiente sucesión infinita de números naturales:

0,1,1,2,3,5,8,13,21,34,55,89,144...

El primer elemento es 0, el segundo es 1 y cada elemento restante es la suma de los dos anteriores:

$$fi = \begin{cases} 0 & \text{si } i = 0 \\ 1 & \text{si } i = 1 \\ f(i-2)+f(i-1) & \text{si } i > 1 \end{cases}$$

A cada elemento de esta sucesión se le llama **número de Fibonacci**. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci.

El código en C++ que representa la función Fibonacci es el siguiente:

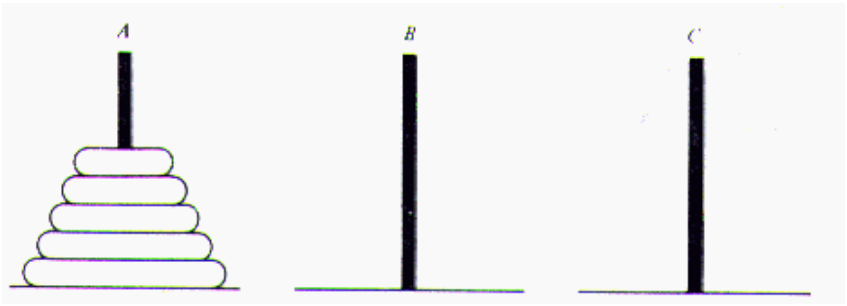
```
#include <iostream>
#include <cstdlib>
using namespace std;
int Fibonacci(int n);
```

PROGRAMACION II. RECURSIVIDAD

```
int main(){
        int valor;
        system("clear");
        cout << "Introduzca numero a calcular: ";
        cin >> valor;
        cout << "\nEl Fibonacci de " << valor << " es: " << Fibonacci(valor) << endl;
        return 0;
}

int Fibonacci(int n){
        if  (n <  0){
          cout << “No existe Fibonacci para numeros negativos.”;
        } else if (n == 0) {
          return 0;
        } else if (n == 1) {
          return 1;
        }else
          return Fibonacci(n-2) + Fibonacci(n -1);
}
```

Ejemplo de las Torres de Hanoi:



Hay tres postes: A, B y C. En el poste A se ponen cinco discos de diámetro diferente de tal manera que un disco de diámetro mayor siempre queda debajo de uno de diámetro menor. El objetivo es mover los discos al poste C usando B como auxiliar. Sólo puede moverse el disco superior de cualquier poste a otro poste, y un disco mayor jamás puede quedar sobre uno menor. Considérese la posibilidad de encontrar una solución. En efecto, ni siquiera es claro que exista una.
Ahora se verá si se puede desarrollar una solución. En lugar de concentrar la atenciń en una solución para cinco discos, considérese el caso general de *n* discos. Supóngase que se tiene una solución para *n* – 1 discos y que en términos de ésta, se pueda plantear la solución para *n* – 1 discos. El problema se resolvería entonces. Esto sucede porque en el caso trivial de un disco (al restar 1 de *n* de manera sucesiva se producirá, al final, 1) la solución es simple: sólo hay que mover el único disco del poste A a C. Así se habrá desarrollado una solución recursiva si se plantea una solución para *n* discos en términos de *n* – 1. Considérese la posibilidad de encontrar tal relación. Para el caso de cinco discos en particular, supóngase que se conoce la forma de mover cuatro de ellos del poste A al otro, de acuerdo con las reglas. ¿Cómo puede completarse entonces el trabajo de mover el quinto disco? Cabe recordar que hay 3 postes disponibles.

Supóngase que se supo cómo mover cuatro discos del poste A al C. Entonces, se pondrá mover éstos exactamente igual hacia B usando el C como auxiliar. Esto da como resultado la situación los cuatro primeros discos en el poste B, el mayor en A y en C ninguno. Entonces podrá moverse el disco mayor de A a C y por último aplicarse de nuevo la solución recursiva para cuatro discos para moverlo de B a C, usando el poste A como auxilia. Por lo tanto, se puede establecer una solución recursiva de las torres de Hanoi como sigue:

Para mover *n* discos de A a C usando B como auxiliar:

1. Si *n* = = 1, mover el disco único de A a C y parar.
2. Mover el disco superior de A a B *n* – 1 veces, usando C como auxiliar.
3. Mover el disco restante de A a C.
4. Mover los disco *n – 1* de B a C usando A como auxiliar

Con toda seguridad este algoritmo producirá una solución completa por cualquier valor de *n*. Si *n* = = 1, el paso 1 será la solución correcta. Si *n* = = 2, se sabe entonces que hay una solución para *n* – 1 = = 1, de manera tal que los pasos 2 y 4 se ejecutaran en forma correcta. De manera análoga, cuando *n* = = 3 ya se habrá producido una solución para *n* – 1 = = 2, por lo que los pasos 2 y 4 pueden ser ejecutados. De esta forma se puede mostrar que la solución funciona para *n* = = 1, 2, 3, 4, 5,... hasta el valor para el que se desee encontrar una solución. Adviértase que la solución se desarrolló mediante la identificación de un caso trivial (*n* = = 1) y una solución para el caso general y complejo (*n*) en términos de un caso mas simple (*n* – 1).

Ya se demostró que las transformaciones sucesivas de una simulación no recursiva de una rutina

recursiva pueden conducir a un programa mas simple para resolver un problema. Ahora se simulará la recursión del problema y se intentará simplificar la simulación no recursiva.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int hanoi(int n, char origen, char destino, char auxiliar);

int main(){
        int valor;
        system("clear");
        cout << "Introduzca numero a calcular: ";
        cin >> valor;
        hanoi(valor,'A','B','C');
        return 0;
}

int hanoi(int n, char origen, char destino, char auxiliar){
        if (n <= 1){
          cout  << "\nmover disco 1 del poste "<< origen << " al poste " << destino << endl;
        }else {
           hanoi(n-1, origen, auxiliar, destino);
           cout <<  "\nmover disco " << n << " del poste " << origen << " al poste "<< destino;
            hanoi (n-1, auxiliar, destino, origen);
        }
}
```