Algoritmos y

Estructuras

De

Datos

FUNCIONES RECURSIVAS

Recursividad

- La recursividad es una **técnica muy potente para la resolución de problemas.**
- Es adecuada para problemas que se pueden solucionar usando el **mismo algoritmo** con **versiones más reducidas de los datos.**
- En general se obtienen **soluciones más elegantes y simples** que si se lo soluciona de manera iterativa.
- Existen problemas que **por su naturaleza tienen un esquema** recursivo o recurrente.

Recurrencia / Recursividad

Multiplicación de números naturales	$n*k = n + n*(k-1)$ con $n*1=1$
Factorial de un número	$n! = n * (n-1)!$ con $0!=1$
Potencia natural de un número	$a^n = a * a^{(n-1)}$ con $a^0 = 1$
Suma de los n primero naturales	Suma (1,n) = n + Suma (1,n-1) con Suma (1,1) = 1
Cantidad de dígitos de un número	CantDigitos (n) = 1+ CantDigitos (int (n/10)); con CantDigitos (k) = 1 si 0<=k<=9
Suma de los dígitos de un número	SumaDigitos (n) = (n%10) + SumaDigitos (int (n/10)) con SumaDigitos (k) = k si 0<=k<=9

Funciones Recursivas

- Funciones que **se llaman a sí mismas**
- Necesitan tener *al menos* un **caso base** (se calcula la solución en forma directa, devolviendo un resultado)
- **Caso general** (no se ha encontrado la solución), luego la función **llama a una nueva copia de sí misma** (paso de recursión) con un **tamaño de problema menor**.
- Eventualmente se llega al caso base: esto permite el camino de **retorno** y la resolución del problema completo.
- El diseño debe garantizar **la llegada a uno de los casos base** en algún momento.

Ejemplo 1. Multiplicación

Multiplicar dos números naturales positivos

```
#include <iostream>
using namespace std;

int multip(int n,int k);

int main(int argc, char *argv[]) {
    int a,b;
    cin >> a >> b;
    cout << endl << a << " multiplicado por " << b << " da: " << multip(a,b);

    return 0;
}
```

```
int multip(int n,int k){
    if (k==1) return n;
    else return n + multip(n,k-1);
}
```

```
4
3

4 multiplicado por 3 da: 12

<< El programa ha finalizado: codig
<< Presione enter para cerrar esta
```

Ejemplo 2. Suma de naturales

Sumar los números naturales hasta n

```
#include <iostream>
using namespace std;

int sumar(int n);

int main(int argc, char *argv[]) {
    int a;
    cin >> a;
    cout << endl << "La suma de los naturales hasta " << a << " es: "
        << sumar(a);
    return 0;
}

int sumar(int n){
    if (n < 2) return 1;
    else return n + sumar(n-1);
}
```

```
4
La suma de los naturales hasta 4 es: 10
```

Ejemplo 3. Dígitos de un numero

Obtener la cantidad de dígitos de un número

```
#include <iostream>
using namespace std;

int cantDig(int n);

int main(int argc, char *argv[]) {
    int a;
    cin >> a ;
    cout << endl << a << " tiene " << cantDig(a) << " digitos " ;

    return 0;
}

int cantDig(int n){
    if (n<10) return 1;
    else return 1 + cantDig (n/10);
}
```

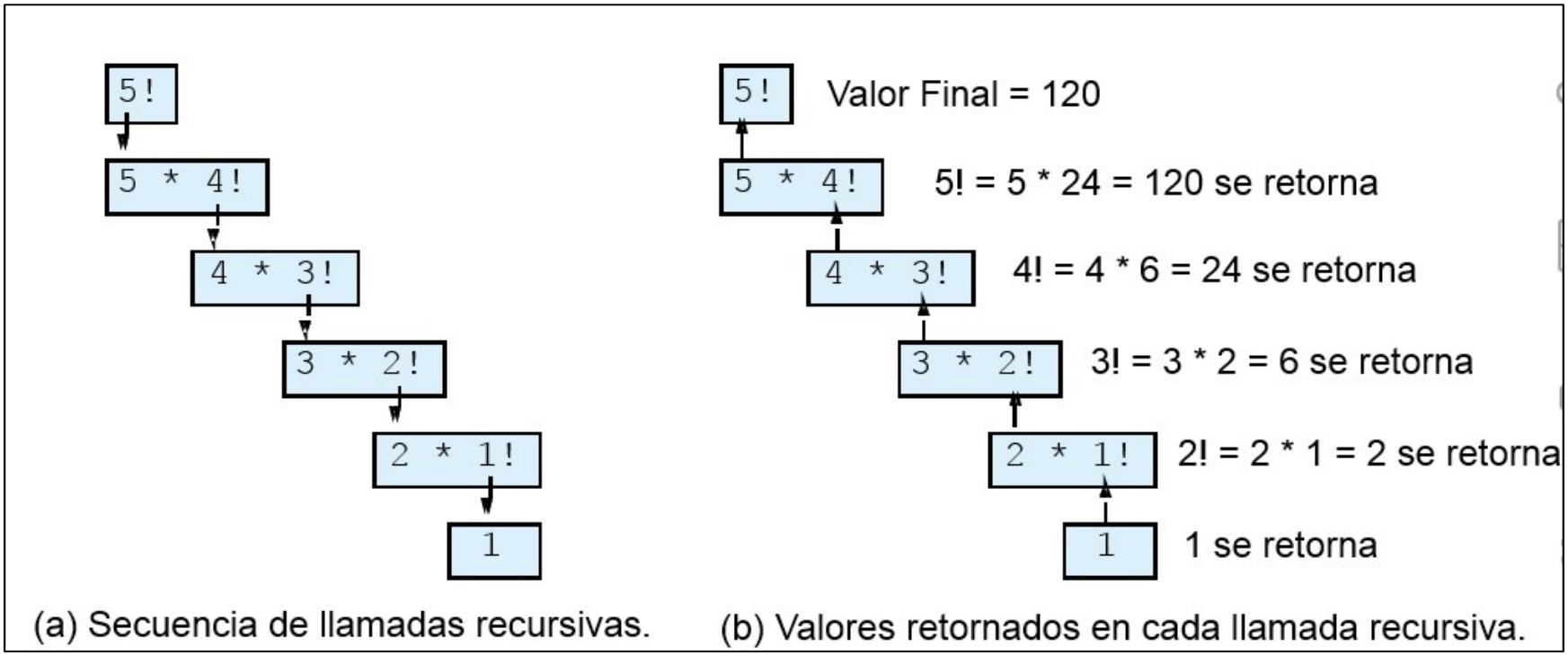
```
123

123 tiene 3 digitos
```

Ejemplo 4. Factorial

- 5! = 5 * 4 * 3 * 2 * 1
- Observe que:
  - 5! = 5 * 4!
  - 4! = 4 * 3! ...
- Los factoriales se pueden calcular recursivamente.
- La resolución del caso base (1! = 0! = 1) permite resolver los otros en orden inverso:
  - 2! = 2 * 1! = 2 * 1 = 2;
  - 3! = 3 * 2! = 3 * 2 = 6;

Ejemplo 4. Factorial



Ejemplo 4. Factorial

```
#include <iostream>
using namespace std;

long factorial (int n);


int main(){
    int i;

    for (i=1; i <= 10; i++){
        cout << i << "! = " << factorial(i) << endl;
    }

    return 0;
}

long factorial (int n){
    if (n == 0) /* caso base */
        return 1;
    /* caso recursivo */
    return n * factorial(n-1);
}
```

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Ejemplo 5. Escritura vertical

Escribir números en la pantalla colocando los dígitos verticalmente

```
#include <iostream>
using namespace std;

void escribirVertical(long n);

int main() {
    long x;
    cout << "Ingrese un numero: ";
    cin >> x;
    escribirVertical(x);
    return 0;
}

void escribirVertical(long n){
    if (n<10) cout << n << endl;
    else {
        escribirVertical(n/10);
        cout << (n%10) << endl;
    }
}
```

```
Ingrese un numero: 123
1
2
3
```

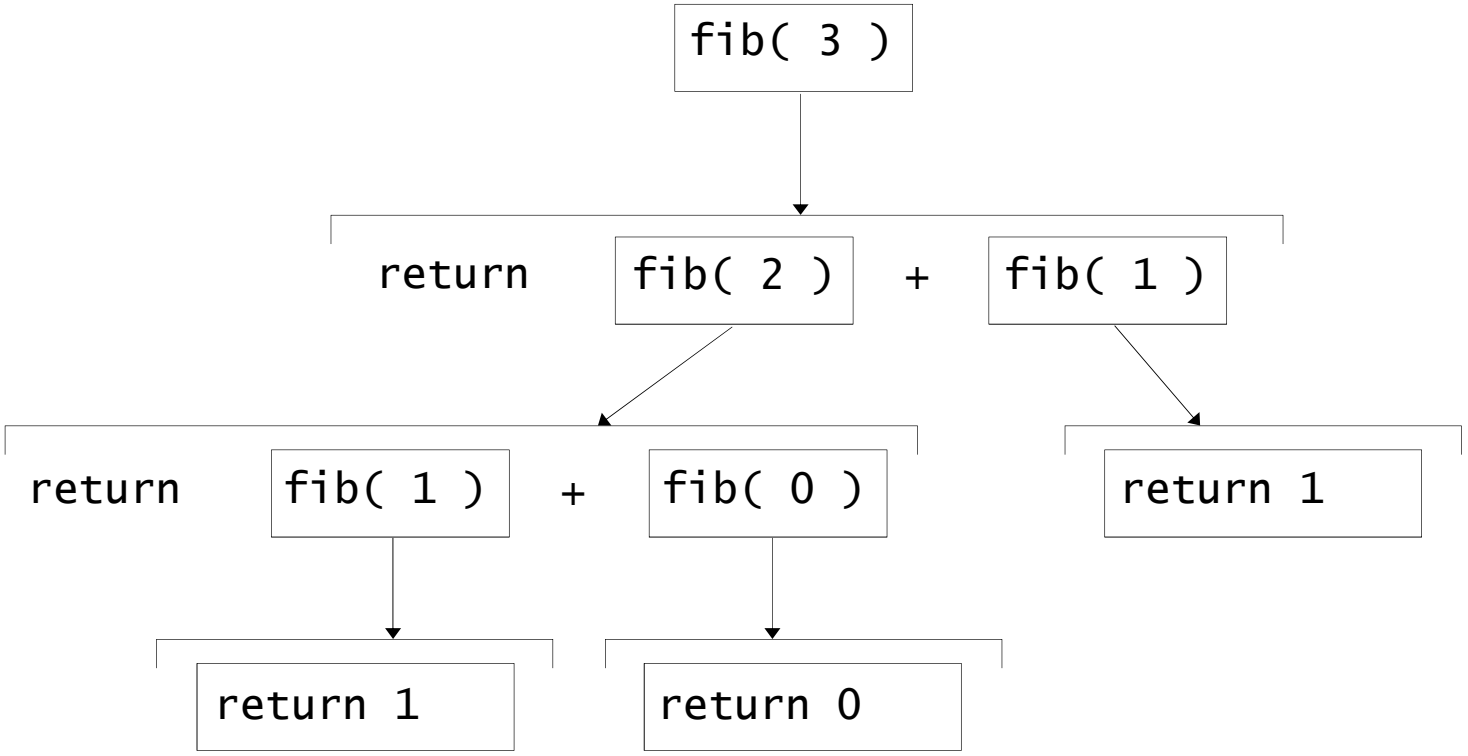
Ejemplo 6. La serie de Fibonacci

- **Serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8...**
  - Cada número es la suma de los dos anteriores
  - Puede ser resuelta recursivamente:
    - **fib(n) = fib(n - 1) + fib(n - 2)**
  - **Código para la función fibonacci**

```
long fibonacci(long n) {
  if (n == 0 || n == 1)  // caso base
        return n;
  else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Ejemplo 6. La serie de Fibonacci

- **Árbol de llamadas recursivas para la función fib**



Ejemplo 6. La serie de Fibonacci

```cpp
#include <iostream>
using namespace std;

int FibonacciR (int n);

int main(){
    int numero, resultado;
    cout << "Ingrese un numero entero: ";
    cin >> numero;

    resultado = FibonacciR(numero);
    cout << "Fibonacci(" << numero << ") = " << resultado;
    return 0;
}

int FibonacciR (int n){
    if (n == 1 || n == 0) //caso base
        return n;
    else
        return FibonacciR(n-2) + FibonacciR(n-1); //caso general
}
```

```
Ingrese un entero: 1
Fibonacci( 0 ) = 0

Ingrese un entero: 2
Fibonacci( 1 ) = 1

Ingrese un entero: 3
Fibonacci( 2 ) = 1

Ingrese un entero: 4
Fibonacci( 3 ) = 2

Ingrese un entero: 5
Fibonacci( 4 ) = 3
```

Salida del programa

```
Ingrese un entero: 6
Fibonacci( 5 ) = 5

Ingrese un entero: 7
Fibonacci( 6 ) = 8

Ingrese un entero: 10
Fibonacci( 10 ) = 55

Ingrese un entero: 20
Fibonacci( 20 ) = 6765

Ingrese un entero: 30
Fibonacci( 30 ) = 832040

Ingrese un entero: 35
Fibonacci( 35 ) = 9227465
```

Recursión vs. Iteración

- **Repetición**
  - Iteración: utiliza en forma explícita una estructura de repetición
  - Recursión: llamadas repetidas a funciones
- **Terminación**
  - Iteración: cuando falla la condición del ciclo
  - Recursión: cuando se alcanza el caso base
- **Ambas** pueden tener ciclos infinitos
- Iteración: Mayor esfuerzo y cantidad de líneas de código que la recursión.
- Recursión: Costo muy elevado de performance *(Las llamadas recursivas requieren tiempo y consumen memoria adicional)*
- **Balance**
  - Elección entre perfomance (iteración) y buenas prácticas de ingeniería de software (recursión).

Recursión infinita

- Si no se llega a una llamada que no implica recursión (caso base), cada llamada recursiva produce otra llamada recursiva. Una llamada a la función se ejecutará, en teoría, eternamente.
- Esto se llama **recursión infinita**.

- En la práctica, una función así se ejecutará hasta que la **computadora se quede sin recursos**, y el programa terminará anormalmente.

LEER:

Libro:
Libro de Savitch (cap. 13)

Libro:
Libro de Deitel & Deitel (cap. 6)