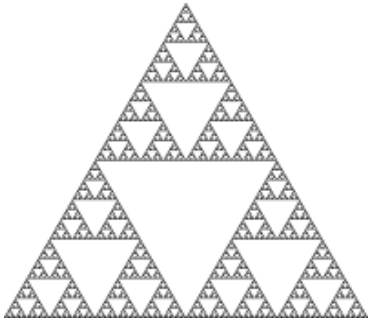
Recursividad

Bach. Rodolfo Mercado Gonzales
Programación Competitiva UPC

Recursividad

Enfoque para resolver un problema, en donde la solución depende de las soluciones de instancias más pequeñas del mismo problema (subproblemas).

Proporciona una manera distinta y poderosa de abordar una variedad de problemas.



Implementación

- La mayoría de lenguajes de programación nos permiten implementar este enfoque a través de funciones recursivas.
- Una función se dice recursiva si se invoca a sí misma de manera directa o indirecta.

```
int recursion(int n, ...) {
    ...
    recursion(n - 1, ... );
    ...
}
```

Recursividad explícita

Existen funciones matemáticas que ya tienen una definición recursiva:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Sucesión de Fibonacci

Implementación

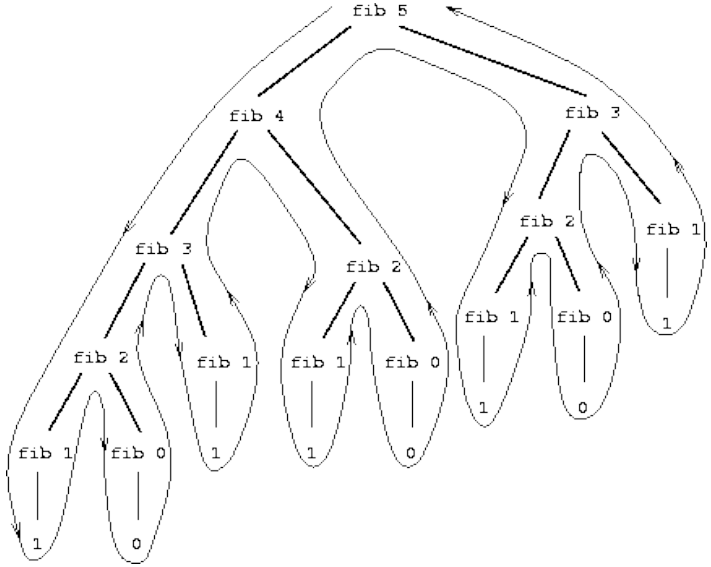
Para implementar una función recursiva debemos tener en cuenta que:

- Cada llamada a la función representa un subproblema nuevo a resolver.
- Debemos tener subproblemas triviales (casos base), que podamos resolverlos directamente (sin recursividad).
- No pueden existir ciclos en las llamadas.

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Recursión Paso a Paso

Árbol de recursión



Análisis del tiempo de ejecución

- La cantidad de operaciones elementales de una recursión esta dada por el número total de llamadas a la función y el número de operaciones que se hacen dentro de cada llamada.

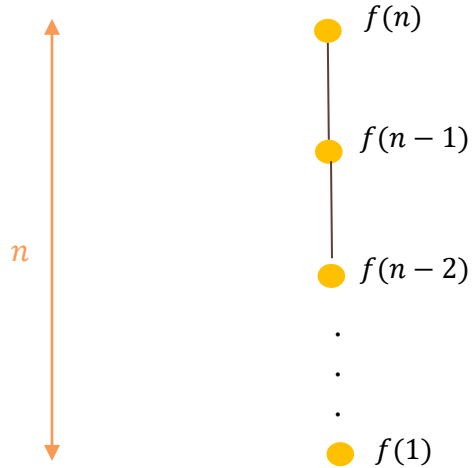
- Es recomendable apoyarse en el árbol de recursión para hallar de una manera más sencilla la complejidad.

Análisis del tiempo de ejecución

Empecemos analizando la complejidad de la siguiente recursión.

```
void f(int n) {
    if (n == 1) return;
    f(n - 1);
}
```

Análisis del tiempo de ejecución



nivel 0: 1 llamada de 1 operación

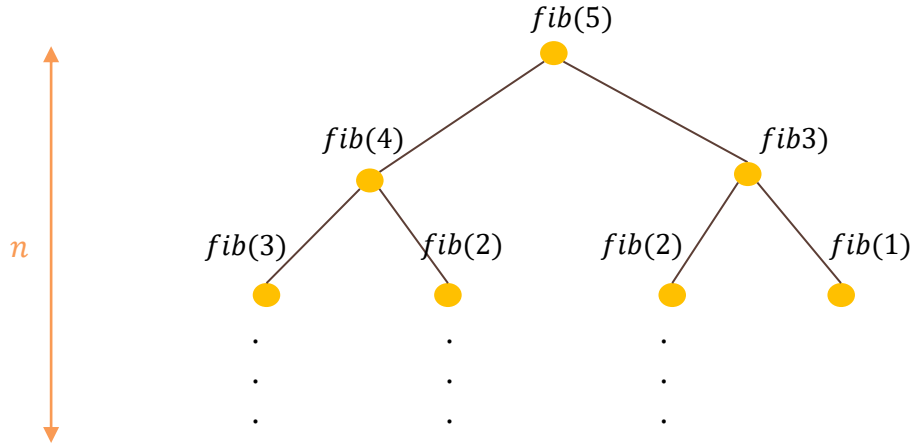
nivel 1: 1 llamada de 1 operación

nivel 2: 1 llamada de 1 operación

$total\ operaciones = n$
$complejidad:\ \boldsymbol{O(n)}$

Análisis del tiempo de ejecución

Ahora analicemos nuestra recursión de Fibonacci.



nivel 0: 2^0 llamadas de 1 operación

nivel 1: 2^1 llamadas de 1 operación

nivel 2: 2^2 llamadas de 1 operación

$$total\ operaciones = 2^0 + 2^1 + 2^2 + \ ... + 2^{n-1} \ = \ 2^n \ - 1$$

$$complejidad: \ \boldsymbol{O}(2^{\boldsymbol{n}})$$

Buscando la recursividad

Algunas veces podemos dar formas recursivas a operaciones matemáticas :

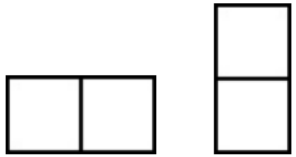
- Multiplicación: $producto(a,b) = a + producto(a, \mathrm{b} - 1)$ $producto(a,0) = 0$
- Potenciación: $potencia(a,b) = a * potencia(a, b - 1)$ $potencia(a,0) = 1$
- Factorial: $factorial(n) = n * factorial(n - 1)$ $factorial(1) = 1$

  $factorial(n) = n * (\mathrm{n} - 1) * factorial(n - 2)$ $factorial(1) = 1 , factorial(2) = 2$

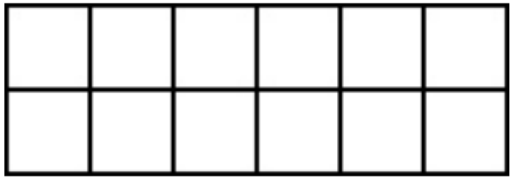
Problema de los dominós

¿De cuántas formas podemos cubrir un tablero de dimensión 2 x n usando dominós de 2 x 1 ?

Dominós de 2 x 1



Tablero de 2 x n



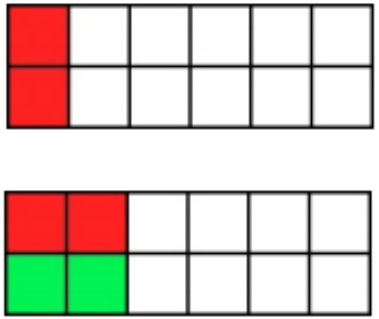
Buscando la recursividad

- También podemos apoyarnos de una definición semántica para usar este enfoque.
- Sea $\boldsymbol{f(n)}$: número de formas de cubrir un tablero de $2\ x\ n$ usando dominós de $2\ x\ 1$

Podemos definir la siguiente recursión:

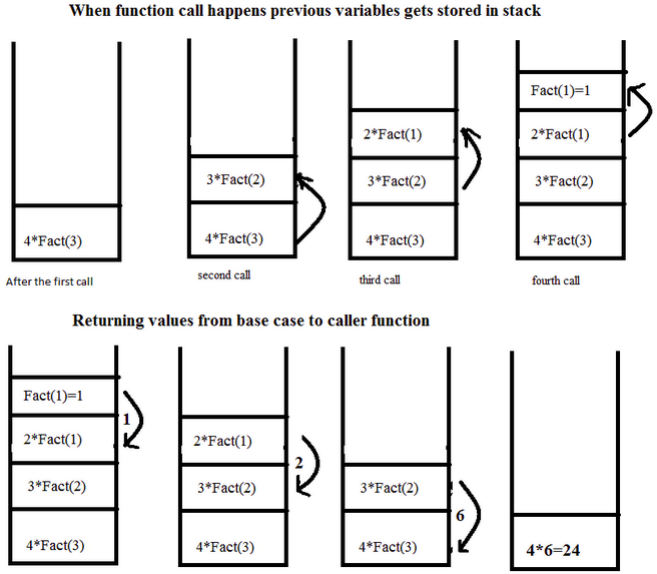
$$\boldsymbol{f(n) = f(n-1) + f(n-2)}$$

$$f(0) = 1$$
$$f(1) = 1$$



Análisis del uso de memoria

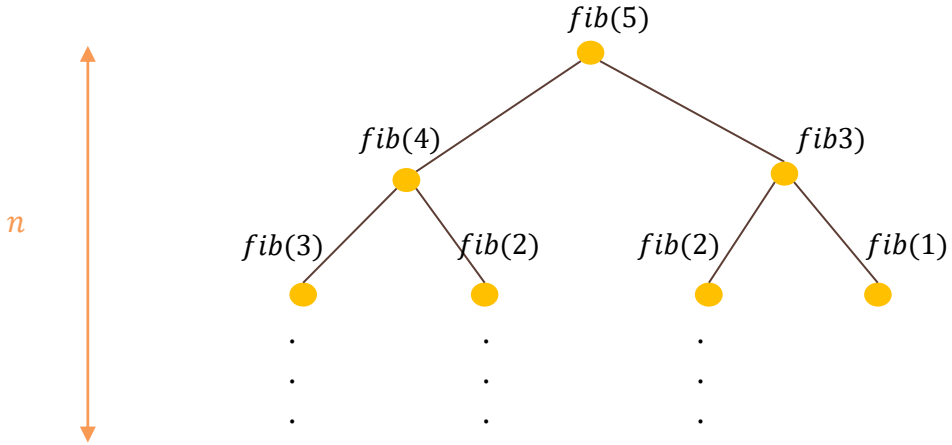
Una función recursiva almacena información en la memoria stack, ya que por cada llamada necesita mantener variables, parámetros, cálculos, …



veamos el comportamiento del stack al hallar de manera recursiva $\boldsymbol{factorial(4)}$

Análisis del uso de memoria

La memoria utilizada será de al menos la profundidad del árbol de recursión.



$$espacio = \boldsymbol{O(n)}$$
$$tiempo = \boldsymbol{O(2^n)}$$

Consideraciones

- Las soluciones recursivas son normalmente menos eficientes que las iterativas en términos de tiempo y espacio.
- Generalmente el stack a nivel local tiene un límite de 2 MB.
- Cuando excedemos el tamaño del stack obtenemos el famoso error “stack overflow”.

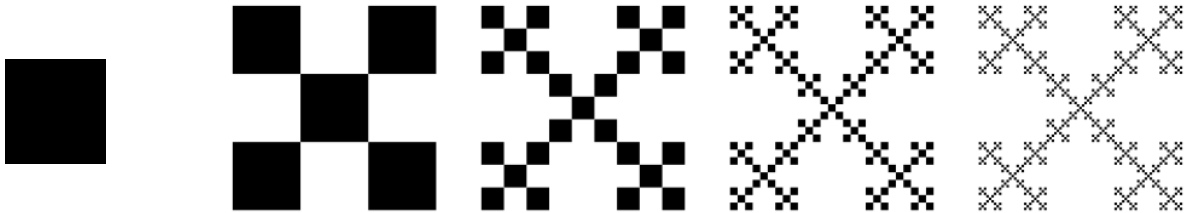
Desafíos

- UVA – Summing Digits
- URI – Fibonacci, ¿Cuántas llamadas?

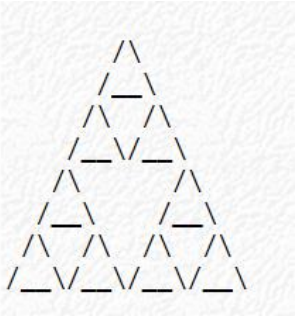
Fractales

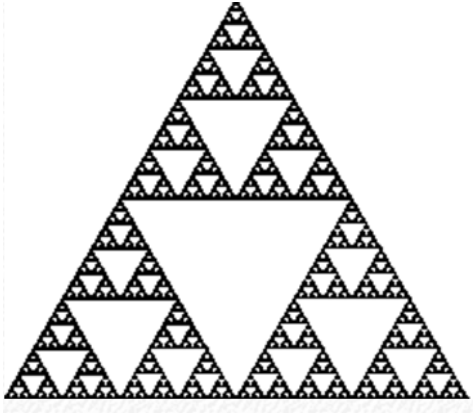
Fractales

Son figuras geométricas que se puede fraccionar en partes que son o se asemejan a una versión reducida de la figura completa.



Triángulo de Sierpinski

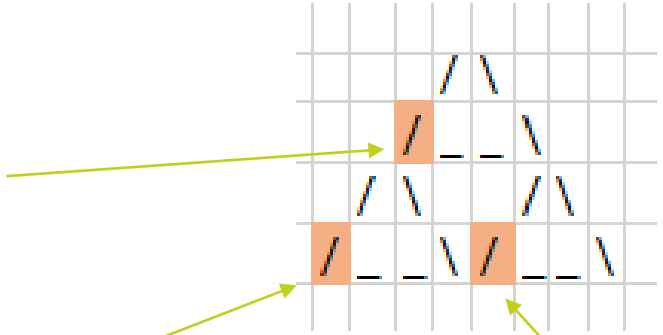




Triángulo de Sierpinski (solución)

```
void draw_fractal(int n, int x, int y)
```

```
draw_fractal(n - 1, x - (1 << (n - 1)), y + (1 << (n - 1)));
```



```
draw_fractal(n - 1, x, y);
```

```
draw_fractal(n - 1, x, y + (1 << n));
```

Triángulo de Sierpinski

```
void draw_fractal(int n, int x, int y) {
    if (n == 1) {
        M[x][y] = M[x - 1][y + 1] = '/';
        M[x][y + 1] = M[x][y + 2] = '_';
        M[x][y + 3] = M[x - 1][y + 2] = '\\';
        return;
    }
    draw_fractal(n - 1, x, y);
    draw_fractal(n - 1, x, y + (1 << n));
    draw_fractal(n - 1, x - (1 << (n - 1)), y + (1 << (n - 1)));
}
```

$complejidad: \boldsymbol{O}(3^{\boldsymbol{n}})$

Ejercicio

- POJ – The Sierpinski Fractal

Desafío

- POJ – Fractal

Referencias

- Topcoder - An Introduction to Recursion
- Antti Laaksonen - Competitive Programmer's Handbook
- Gayle McDowell - Cracking the Coding Interview

“Life offers you so many doors. It's up to you which to open and which one to close.”

- ICPC News