Índice

Qué es la recursividad

Se dice que una función es recursiva cuando se define en función de si misma. La recursividad puede ser directa o indirecta

- Directa: cuando hay una sentencia explícita en el código de la función llamándose a sí misma
- Indirecta: cuando el llamado recursivo se realiza a través de otra funcion. Por ejemplo A llama a B y B llama a A

Factorial con iteración

```
xxxxxxxxxx
int factorial (int n) {
    int resultado = 1;
    for (int i = 2; i <= n ; i ++)
        resultado *= i ;
    return resultado ;
}
```

Factorial con recursividad

```
xxxxxxxxxx
int factorial (int n) {
    if (n < 2) // caso base
        return 1;
    return n * factorial (n -1); // llamado recursivo
}
```

Funcionamiento de la memoria

Por cada llamado a una función, el programa guarda un registro con los datos locales y el punto de regreso. Cuando se regresa, se libera de la pila los datos pertinentes a la función.

Supongamos que en la funcion *main* hago un llamado a una función *f*

```
xxxxxxxxxx
int main () {
    // código
    f(...); // instrucción A
```

```
    // código
}
```

Y en la función *f* hago un llamado a una función *g*

```
xxxxxxxxxx
int f(...) {
    // código
    g(...); // instrucción B
    // código
    return ...; // regresa a A
}
```

```
xxxxxxxxxx
int g(...) {
    // código
    return ...; // regresa a B
}
```

Lo que está pasando ahí, es que inicialmente la memoria tiene guardadas las variables locales de *main* (y algunos datos más). En el llamado a *f* se agregan al registro las variables locales de *f* y el *a dónde debe retornar (instrucción A)*. A su vez, en el llamado a *g* se agregan al registro las variables locales de *g* y el *a dónde debe retornar (instrucción B)*.

image-20200528135501911

image-20200528135549729

Stack overflow

Si una función es recursiva, se guardan sus variables y parámetros usando la pila, y la nueva instancia de la función va a trabajar con su propia copia de las variables locales. Cuando la segunda instancia de la función regresa, recupera las variables y los parámetros de la pila y continúa la ejecución en el punto que había sido llamada.

En el ejemplo anterior del factorial, si la función recursiva es llamada con el valor 5 se llamará una vez por el número 4, otra por el 3 otra por el 2, otra por el 1, y finalmente una por el 0. Osea se van a guardar los datos de 6 registros. De manera genérica, podríamos decir que si llamamos a la funcion con el parámetro *n* se van a guardar *n + 1* registros. Dependiendo de la cantidad de datos, llamados y capacidad de la memoria, la pila puede llegar a llenarse. Eso se llama *stack overflow* o desbordamiento de pila.

Cuando suceda un *stack overflow* la aplicación finalizará por haber consumido toda la memoria disponible, por lo tanto debemos ser cuidadosos al utilizar la recursividad.

Recursividad de cola

Factorial con recursividad de cola

```
xxxxxxxxxx
int f (int n, int res) {
    if (n == 0)
        return res ;
    return f(n-1 , n*res);
```

```
}


int factorial (int n) {
    return f(n, 1);
}
```

Supongamos que se llama a la función factorial con un 5.

```
xxxxxxxxxx
int main() {
    factorial(5);
}
```

1. Factorial va a devolver un llamado a f con 5 y 1
2. En f, como 5 no es igual a 0, devuelve una llamada a f con 4 y 5.
3. En f, como 4 no es igual a 0, devuelve una llamada a f con 3 y 20.
4. En f, como 3 no es igual a 0, devuelve una llamada a f con 2 y 60.
5. En f, como 2 no es igual a 0, devuelve una llamada a f con 2 y 120.
6. En f, como 1 no es igual a 0, devuelve una llamada a f con 1 y 120.
7. En f, como 0 sí es igual a 0, devuelve 120 a factorial, y factorial a main.

Recursividad indirecta

Como dijimos antes, la recursividad se clasifica como indirecta cuando por ejemplo una función A hace un llamado a B, y B a A.

Veamos un ejemplo con 3 funciones.

```
xxxxxxxxxx
// Acumula la suma con el valor actual y llama a calcular decrementando el índice
void sumar (int vec[], int res[], int n) {
    res [0] += vec[n];
    calcular (vec, res, n-1);
}
// Acumula el producto con el valor actual y llama a calcular decrementando el índice
void multiplicar (int vec[], int res[], int n) {
    res[1] *= vec[n];
    calcular (vec, res, n-1);
}
// Decide a qué función llamar: sumar o multiplicar
void calcular (int vec[], int res[], int n) {
    if (n >= 0) {
        if ((vec[n] % 2) == 0)
            sumar (vec, res, n);
        else
            multiplicar (vec, res, n);
    }
```

```
}
```

Entonces en el main podría tener algo así:

```
xxxxxxxxxx
int main () {
    int vec [] = {5 , 4 , 8 , 1 , 9};
    int res [] = {0 , 1};
    calcular (vec, res, 4);
    cout << " Resultado : " << res [0] << " " << res [1] << endl ;
    return 0;
}
```

Lo que sucede en el ejemplo anterior es lo siguiente:

1. La función calcular decide si lo que se va a procesar es par o impar, y dependiendo de eso llama a sumar o a multiplicar. Como el primer número es 5, llama a multiplicar
2. Multiplicar acumula el producto con el valor actual en res[1], y llama a calcular con el siguiente numero que es 4.
3. Como 4 es par, calcular llama a la función sumar.
4. Sumar acumula la suma con el valor actual en res[0], y llama a calcular con el siguiente número que es 8.
5. Como 8 es par, calcular llama a la función sumar.
6. Etc.

Recursividad anidada

Supongamos que tenemos la siguiente sucesión

image-20200528141107900

Si *n* vale 1 o más de 5, la devolución es inmediata. Sin embargo, si vale 2, 3 o 4 hay un doble llamado recursivo, se vuelve a calcular la función con un nuevo argumento, y en ese argumento hay que hacer otro cálculo de dicha función.

El código seria:

```
xxxxxxxxxx
int anidada (int n) {
    if ( n == 1 || n >= 5)
        return n ;
    return anidada (1 + anidada(2*n)); // acá está el anidado, en anidada(2*n)
}
```

Recursividad múltiple

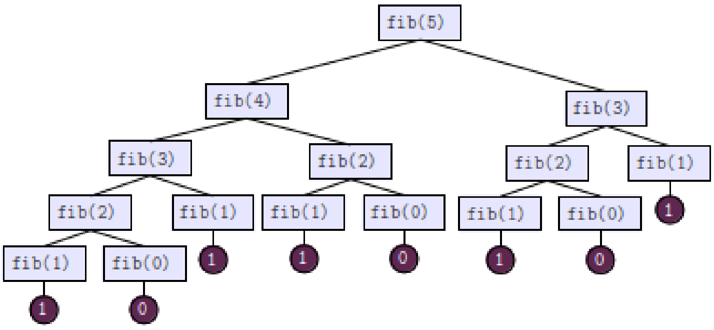
El caso más típico de una recursividad múltiple, es la sucesión de Fibonacci

image-20200528142753288

La sucesión de 5 por ejemplo, se obtiene mediante la suma de las dos sucesiones anteriores (excepto 1 y 0). De este modo:

- F(0) = 0
- F(1) = 1
- F(2) = F(1) + F(0) = 1 + 0 = 1
- F(3) = F(2) + F(1) = 1 + 1 = 2

- F(4) = F(3) + F(2) = 2 + 1 = 3
- F(5) = F(4) + F(3) = 3 + 2 = 5



El árbol que representa las llamadas a la función se va duplicando en cada nivel hasta llegar a los casos base (0 y 1). Este crecimiento es exponencial, por lo tanto hay que ser cuidadosos al trabajar con recursividad múltiple (a menos que sepamos que los valores son chicos).

Fibonacci iterativo

```
xxxxxxxxxx
int fibo (int n) {
    int res = n ;
    if (n > 1) {
        res = 1;
        int ant = 0;
        for (int i = 2; i <= n; i ++) {
            res = res + ant ;
            ant = res - ant ;
        }
    }
    return res ;
}
```

Fibonacci recursivo múltiple

```
xxxxxxxxxx
int fibo (int n) {
    if (n <= 1)
        return n ;
    return fibo(n -1) + fibo(n -2);
}
```

Fibonacci recursivo simple y de cola

```
xxxxxxxxxx
// Función recursiva simple y de cola
```

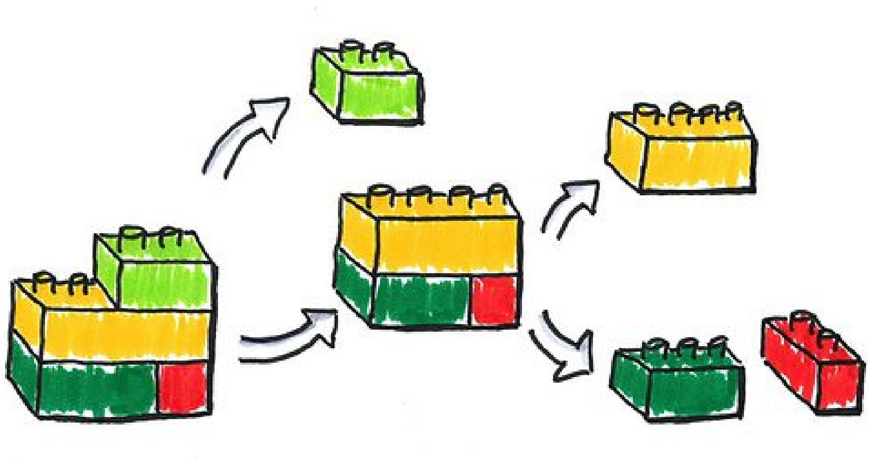
```
int fibo (int n, int res, int res_ant) {
    if ( n == 1)
        return res ;
    return fibo (n -1, res + res_ant, res);
}

// Solo llama a la función fibo con los dos primeros resultados: 0 y 1
int fibonacci (int n) {
    if (n == 0)
        return n ;
    return fibo (n, 1, 0);
}
```

Algoritmo divide y vencerás

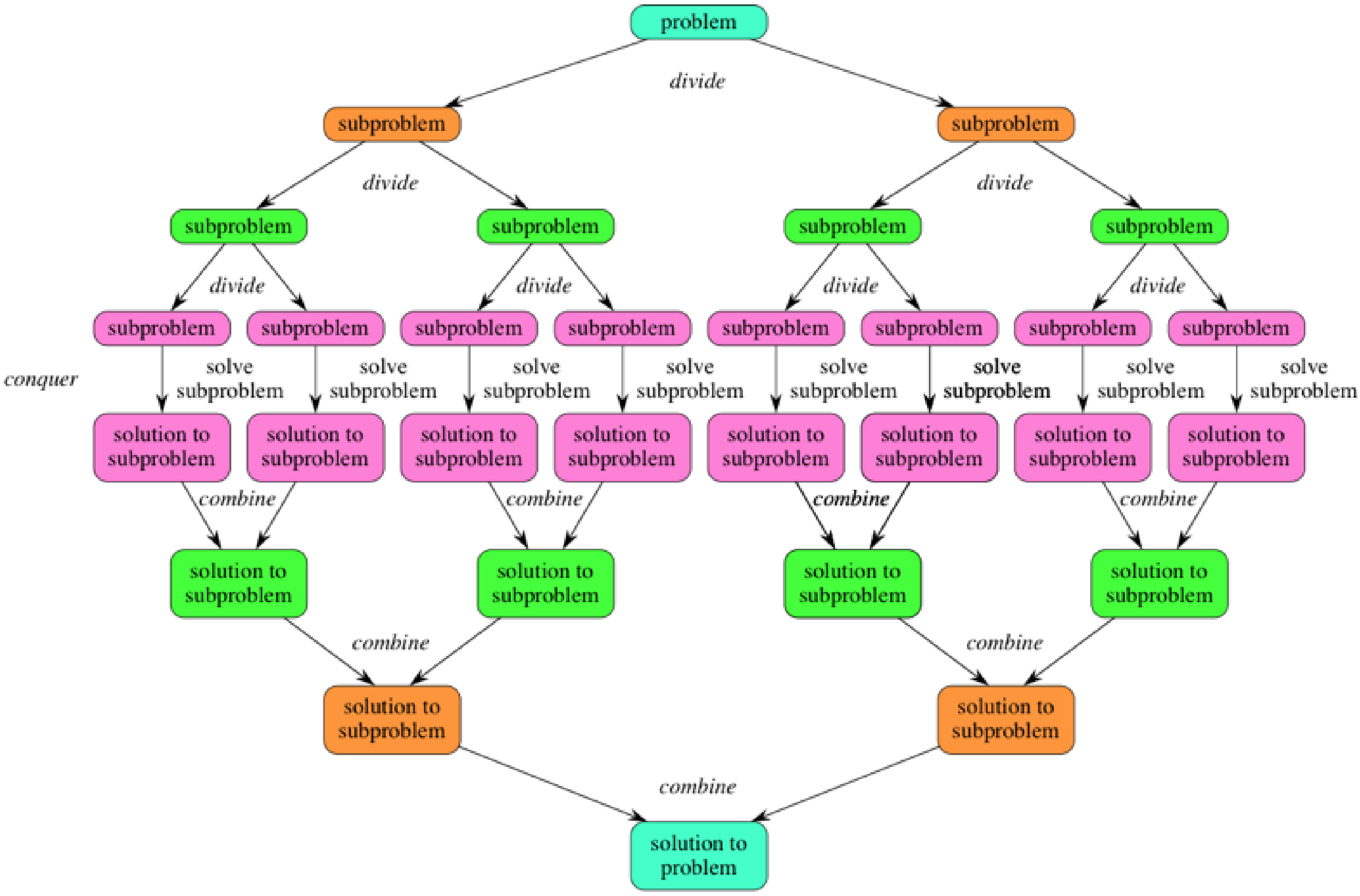
Cuando se descompone un problema comlpejo en partes más simples para tratarlas por separado, se está aplicando una resolución de tipo "divide y vencerás". De esta manera se facilita el desarrollo de la solución.

Definición: Si el tamaño del problema a tratar es N > L, siendo L un valor arbitrario, se debe dividir el problema en m subproblemas no solapados y que tengan la misma estructura que el problema original. Luego, a cada uno de esos m subproblemas se los dividirá nuevamente hasta que cada uno alcance un tamaño L o menor, los cuales se resolverán por cualquier otro método. Por último, la solución del problema original se obtiene por combinación de las m soluciones obtenidas de los subproblemas en que se lo había dividido.

Una representación gráfica sería:



O también:



Como se puede inferir, hay una cercanía conceptual entre los algoritmos "divide y vencerás" y la recursividad, dado que la división del problema en *m* subproblemas de la misma estructura sugiere llamados recursivos en donde se ha reducido el tamaño de la entrada de la forma indicada, y la solución cuando el tamaño del problema no supera *L* se va a corresponder con el caso base.

Nota: si bien hay una cercanía conceptual, no necesariamente hay que aplicar recursividad.

Supongamos que quiero hacer una función para calcular la potencia

Función potencia iterativa

```
xxxxxxxxxx
int potencia (int base, int exponente) {
    int resultado = 1;
    for (int i = 0; i < exponente; i ++)
        resultado *= base;
    return resultado;
}
```

En este caso, si quiero hacer 2^64, hará sesenta y cuatro ciclos.

Funcion potencia recursiva

```
xxxxxxxxxx
int potencia (int base, int exponente) {
    if ( exponente == 0)
        return 1;
```

```
    return base * potencia (base, exponente - 1);
}
```

En el caso de la función recursiva, si quiero hacer 2^64, hará 64 llamados a sí misma.

Funcion potencia con algoritmo divide y vencerás

Como se ve en los ejemplos anteriores, no hay mejoría en cuanto a la eficiencia entre la iteración y la recursividad. Para mejorar eso podríamos usar un algoritmo divide y vencerás

```
xxxxxxxxxx
1  int potencia (int base, int exponente) {
2
3     // Casos base
4     if (exponente == 0)
5        return 1;
6     if (exponente == 1)
7        return base;
8
9     // Caso general
10    int res = potencia(base, exponente/2);
11    res *= res ;
12
13    // Si el exponente es impar
14    if ((exponente % 2) == 1)
15       res *= base ;
16    return res ;
17  }
```

De esta manera si quiero hacer 2^64

1. Va a dividir el 64 en 2
2. Quedaría (2^32) ^ 2
3. Que es lo mismo que 2^32 * 2^32
4. Ahi va a dividir el 32 en 2
5. Quedaría (2^16) ^ 2
6. Que es lo mismo que 2^16 * 2^16, por ende quedaría (2^16 * 2^16) * (2^16 * 2^16)
7. Ahí va a dividir el 16 en 2
8. Quedaría (2^8) ^ 2
9. Que es lo mismo que 2^8 * 2^8, por ende quedaría ((2^8) * (2^8)) * ((2^8) * (2^8)) * ((2^8) * (2^8)) * ((2^8) * (2^8))
10. Y así sucesivamente

De manera que en vez de 64 operaciones, se harían 6.

Cuando el exponente sea impar, por ejemplo 2^65

1. potencia(2, 65)

2. Linea 10

    1. res = potencia(2, 32)
    2. res2 = potencia(2, 16)

3. res3 = potencia(2, 8)
4. res4 = potencia(2, 4)
5. res5 = potencia(2, 2)
6. res6 = potencia(2, 1)
7. res6 = 2*2 = 4