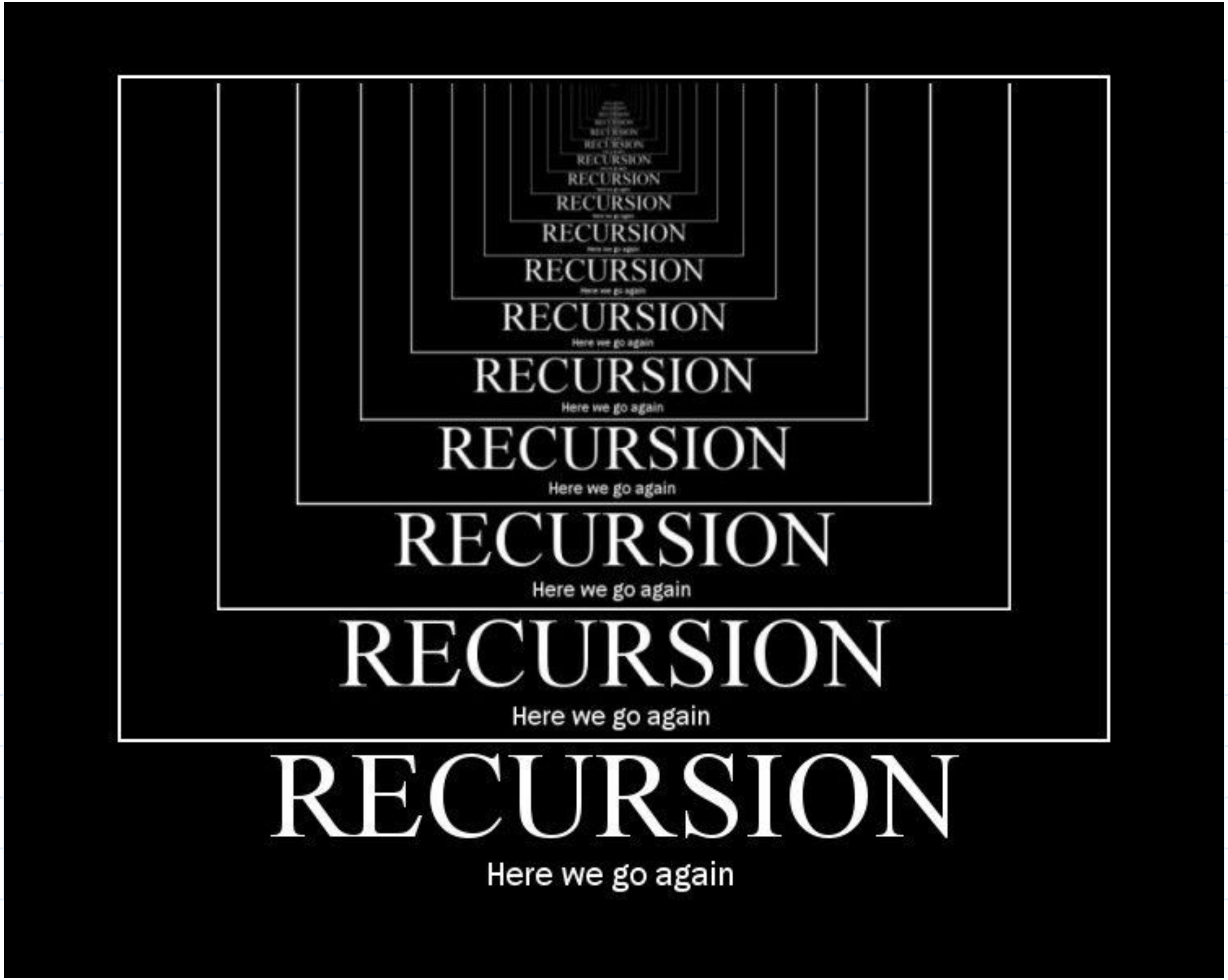
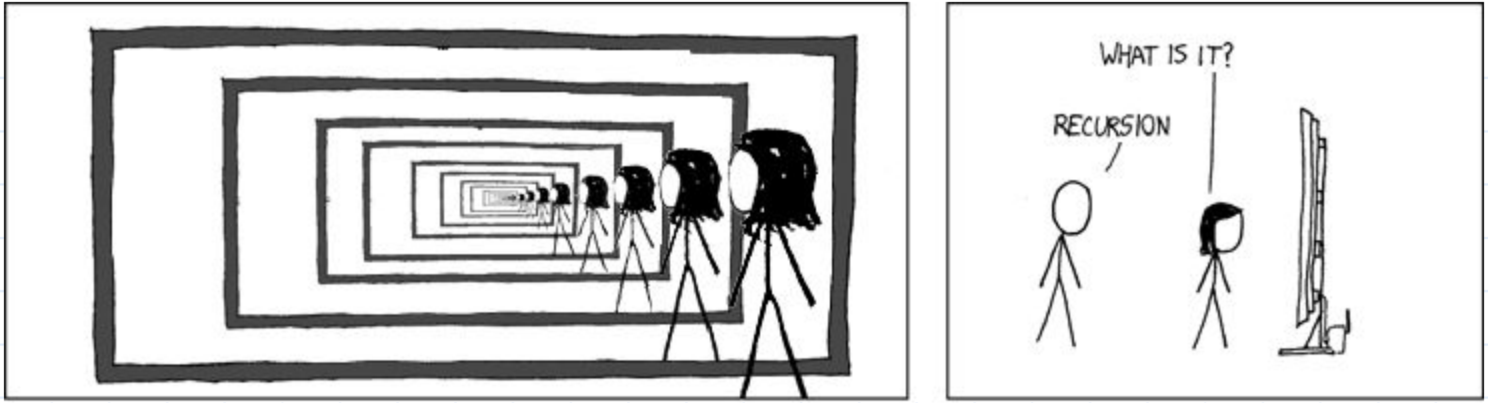
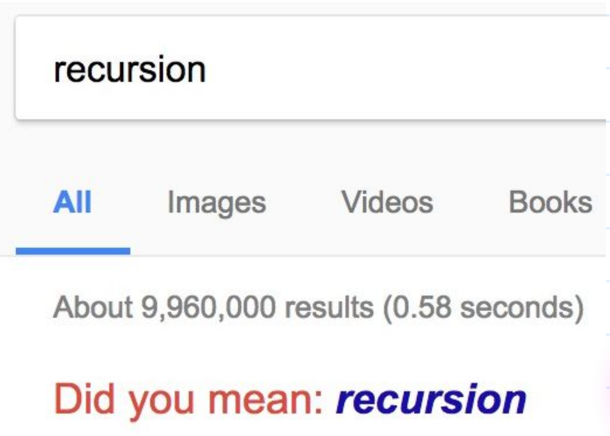
19/08/21

Idea intuitiva de la recursión

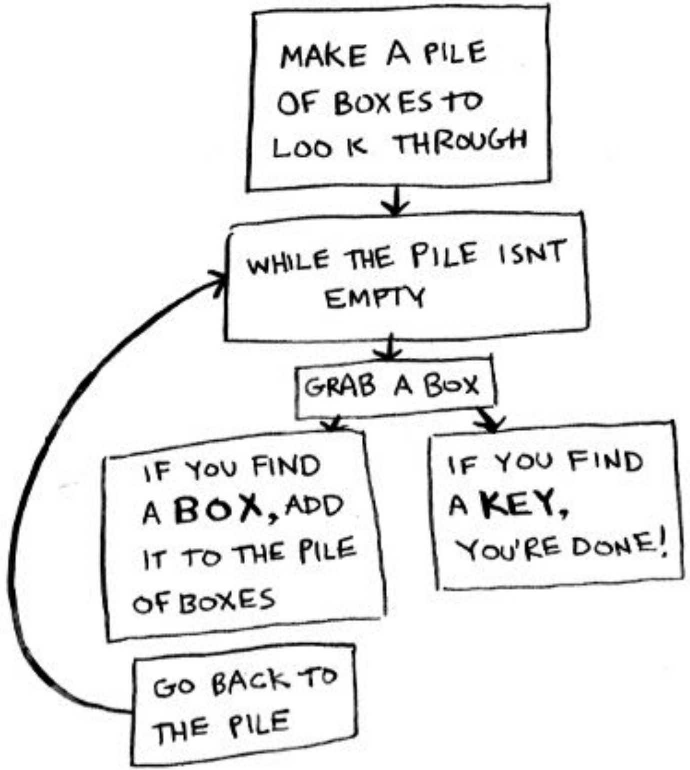
(Recurrencia o recursividad)


RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again

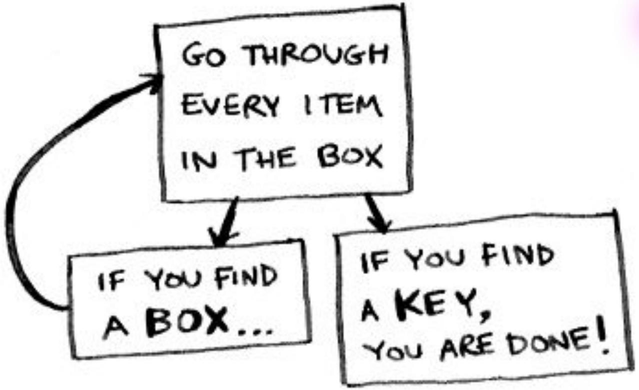

WHAT IS IT?
RECURSION


recursion
All
Images
Videos
Books
About 9,960,000 results (0.58 seconds)
Did you mean: recursion

Iterative Approach



Recursive Approach



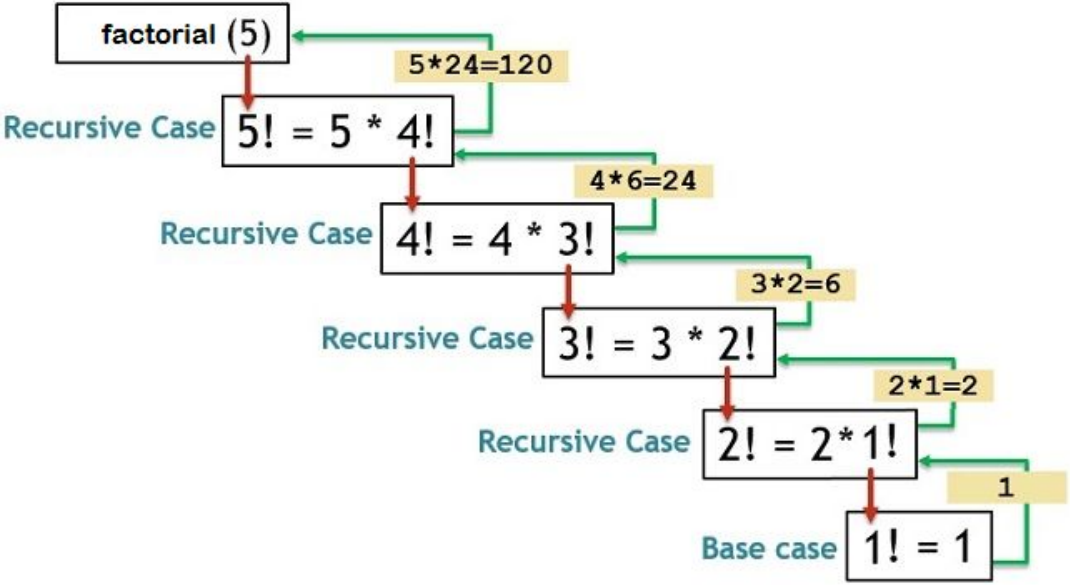

```
fun main(args: Array<String>) {
    ... .. ...
    recurse()
    ... .. ...
}

fun recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

Normal Function Call

Recursive Call

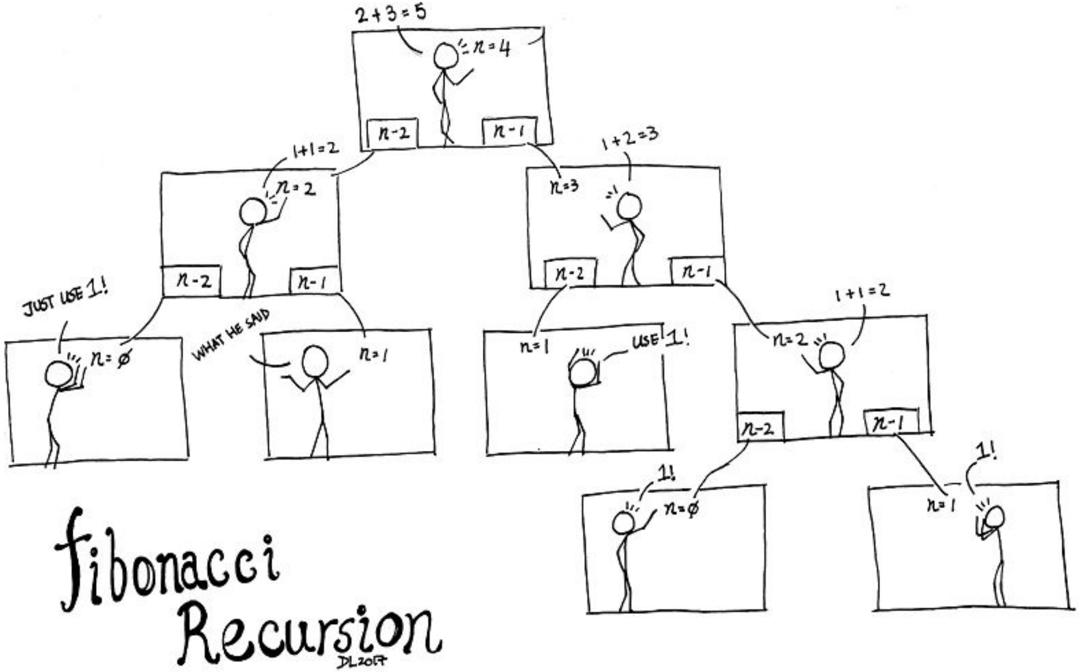
Factorial de un número

$$
n! = \begin{cases} 1 & si \quad n == 1 \quad \text{caso base/trivial} \\ n * (n-1)! & si \quad n > 1 \quad \text{caso recursivo} \end{cases}
$$



Serie de Fibonacci

$$fibo(n) = \begin{cases} 1 & si \quad n \leq 1 \quad \text{caso base} \\ fibo(n-2) + fibo(n-1) & si \quad n > 1 \quad \text{caso recursivo} \end{cases}$$



The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

Aplicación de la recursión: búsqueda binaria

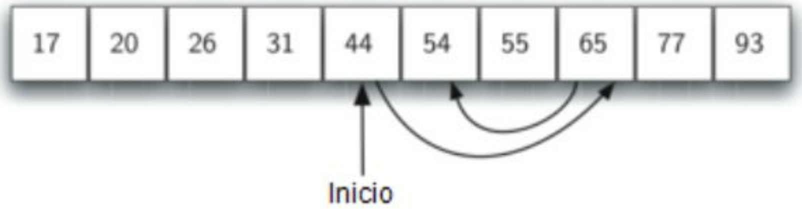


Imagen comparativa

GIF ilustrativo

Datos comparativos

Algoritmo de la búsqueda binaria

Se desea buscar el elemento **x** en el arreglo ordenado **a**, desde **low** (cero) hasta **high** (la última casilla).

```
if (low > high)
    return(-1);
mid = (low + high) / 2;
if (x == a[mid])
    return(mid);
if (x < a[mid])
    search for x in a[low] to a[mid - 1];
else
    search for x in a[mid + 1] to a[high];
```

El algoritmo retorna el índice correspondiente a la casilla donde se encuentra **x** o **-1** si **x** no se encontró.

Since the possibility of an unsuccessful search is included (that is, the element may not exist in the array), the trivial case has been altered somewhat. A search on a one-element array is not defined directly as the appropriate index. Instead that element is compared with the item being searched for. If the two items are not equal, the search continues in the "first" or "second" half—each of which contains no elements. This case is indicated by the condition *low* > *high*, and its result is defined directly as −1.

Let us apply this algorithm to an example. Suppose that the array *a* contains the elements 1, 3, 4, 5, 17, 18, 31, 33, in that order, and that we wish to search for 17 (that is, *x* equals 17) between item 0 and item 7 (that is, *low* is 0, *high* is 7). Applying the algorithm, we have

Line 1: Is *low* > *high*? It is not, so execute line 3.

Line 3: *mid* = (0 + 7)/2 = 3.

Line 4: Is *x* == *a*[3]? 17 is not equal to 5, so execute line 6.

Line 6: Is *x* < *a*[3]? 17 is not less than 5, so perform the ***else*** clause at line 8.

Line 9: Repeat the algorithm with *low* = *mid* + 1 = 4 and *high* = *high* = 7; i.e., search the upper half of the array.

Line 1: Is 4 > 7? No, so execute line 3.

Line 3: *mid* = (4 + 7)/2 = 5.

Line 4: Is *x* == *a*[5]? 17 does not equal 18, so execute line 6.

Line 6: Is *x* < *a*[5]? Yes, since 17 < 18, so search for *x* in *a*[*low*] to *a*[*mid* − 1].

Line 7: Repeat the algorithm with *low* = *low* = 4 and *high* = *mid* − 1 = 4. We have isolated *x* between the fourth and the fourth elements of *a*.

Line 1: Is 4 > 4? No, so execute line 3.

Line 3: *mid* = (4 + 4)/2 = 4.

Line 4: Since *a*[4] == 17, return *mid* = 4 as the answer. 17 is indeed the fourth element of the array.

Line 1: Is *low* > *high*? 0 is not greater than 7, so execute line 3.

Line 3: $mid = (0 + 7)/2 = 3$.

Line 4: Is $x == a[3]$? 2 does not equal 5, so execute line 6.

Line 6: Is $x < a[3]$? Yes, $2 < 5$, so search for x in $a[low]$ to $a[mid - 1]$.

Line 7: Repeat the algorithm with $low = low = 0$ and $high = mid - 1 = 2$.
If 2 appears in the array, it must appear between $a[0]$ and $a[2]$ inclusive.

Line 1: Is $0 > 2$? No, execute line 3.

Line 3: $mid = (0 + 2)/2 = 1$.

Line 4: Is $2 == a[1]$? No, execute line 6.

Line 6: Is $2 < a[1]$? Yes, since $2 < 3$. Search for x in $a[low]$ to $a[mid - 1]$.

Line 7: Repeat the algorithm with $low = low = 0$ and $high = mid - 1 = 0$.
If x exists in a it must be the first element.

Line 1: Is $0 > 0$? No, execute line 3.

Line 3: $mid = (0 + 0)/2 = 0$.

Line 4: Is $2 == a[0]$? No, execute line 6.

Line 6: Is $2 < a[0]$? 2 is not less than 1, so perform the ***else*** clause at line 8.

Line 9: Repeat the algorithm with $low = mid + 1 = 1$ and $high = high = 0$.

Line 1: Is *low* > *high*? 2 is greater than 1, so – is returned. The item 2 does not exist in the array.

Tipos de recursión

Pueden distinguirse distintos tipos de llamada recursivas dependiendo del número de funciones involucradas y de cómo se genera el valor final. A continuación veremos cuáles son.

Recursión lineal

En la recursión lineal cada llamada recursiva genera, como mucho, otra llamada recursiva. Se pueden distinguir dos tipos de recursión lineal atendiendo a cómo se genera resultado.

Recursión lineal no final
Recursión por posposición

"Queda trabajo por hacer"

El resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado de la función que llama. El ejemplo típico de recursión lineal no final es cálculo del factorial de un número.

```
int factorial(int numero){
  if (numero > 1) return (numero*factorial(numero-1));
  else return(1);
}

int main (){
  int n;
  printf("Introduce el número: ");
  scanf("%d",&n);
  printf("El factorial es %d", factorial(n));
}
```

Recursión lineal final
Recursión por cola

"No queda trabajo por hacer"

El resultado que es devuelto es el resultado de ejecución de la última llamada recursiva. Un ejemplo de este cálculo es el máximo común divisor.

```
long mcd(long a, long b){
   if (a==b) return a;
   else if (a<b) return mcd(a,b-a);
   else  return mcd(a-b,b);
}
```

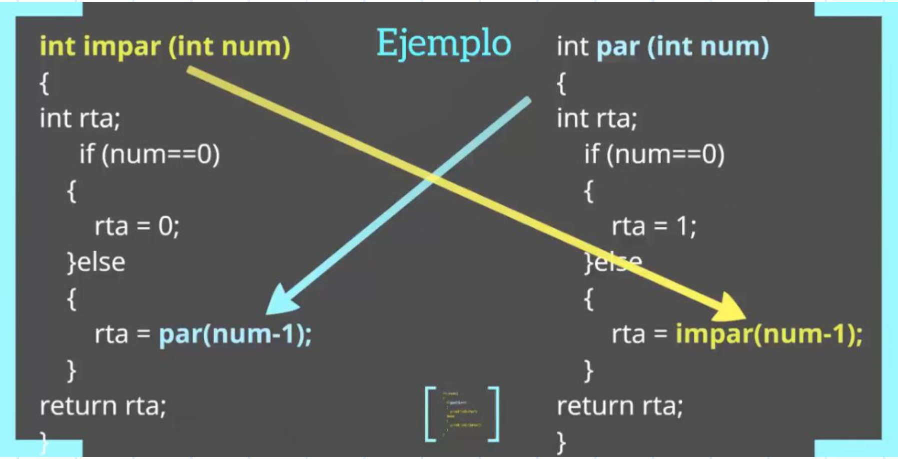
Recursión múltiple

Alguna llamada recursiva puede generar más de una llamada a la función. Uno de los ejemplos más típicos son los números de Fibonacci, números que reciben el nombre del matemático italiano que los descubrió.

```
long fibonacci(int n)
{
  if (1 == n || 2 == n) {
    return 1;
  } else {
    return (fibonacci(n-1) + fibonacci(n-2));
  }
}
```

Recursión mutua

Implica más de una función que se llaman mutuamente. Un ejemplo es el determinar si un número es par o impar mediante dos funciones.




```
int main(){
  int n= 30;
  if (par(n))
    printf("El número es par");
    else
    printf("El número es impar");
}

int par(int n){
  if (n==0) return 1;
  else return (impar(n-1));
}

int impar(int n){
  if (n==0) return 0;
  else return(par(n-1));
}
```

¿Consultas?