Current Skill Design rules.

Recursive Algorithms

Now, how really the recursion is executed and what the difference with iterative algorithms. For something simple to start with – let's write a function pow(x, n) that raises x to a natural power of n. In other words, multiplies x by itself n times. There are two ways to implement it.

1. Iterative thinking: the for loop:

```
FUNCTION pow(x,n : INTEGER) : INTEGER
VAR
    result : INTEGER := 1;
    i : INTEGER;
BEGIN
    FOR i FROM 1 TO n  DO
        result := result *x;
    END_FOR
    RETURN result ;
END
```

2. Recursive thinking: simplify the task and call self:

```
FUNCTION pow(x,n : INTEGER) : INTEGER
BEGIN
    IF (n = 1) THEN
        RETURN x;
    ELSE
        RETURN x * pow(x, n-1);
    END_IF
END
```

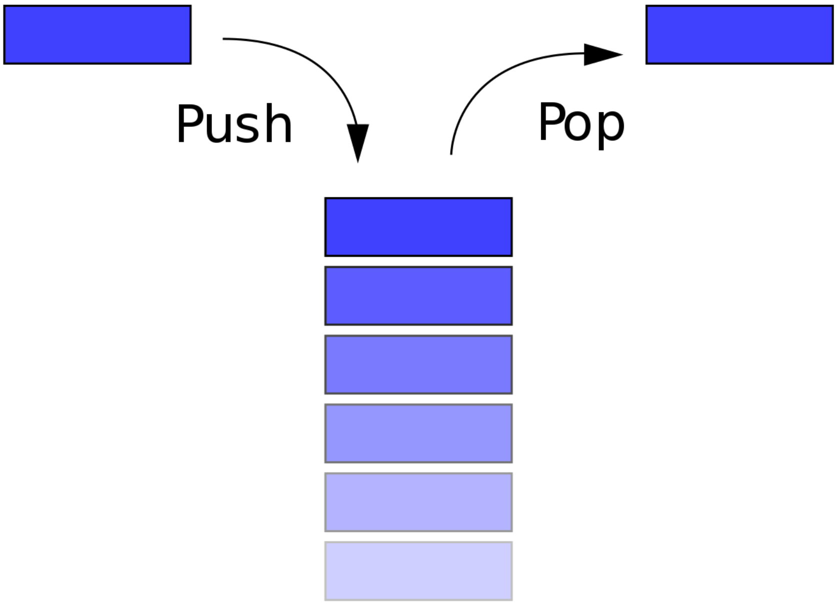
Recursive Algorithms

Now let's examine how recursive calls work in a machine for any programming language. For that we'll look under the hood of functions.

The information about the process of execution of a running function is stored in its execution context. The execution context is an internal data structure that contains details about the execution of a function: where the control flow is now, the current variables and few other internal details.

One function call has exactly one execution context associated with it. When a function makes a nested call, the following happens:

1. The current function is paused.
2. The execution context associated with it is remembered in a special data structure called execution context stack.
3. The nested call executes.
4. After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.



Recursive Algorithms

Let's see what happens during the pow(2, 3) call.

The information about the process of execution of a running function is stored in its execution context.
The execution context is an internal data structure that contains details about the execution of a function: where the control flow is now, the current variables, the values.
One function call has exactly one execution context associated with it.

When a function makes a nested call, the following happens:

The current function is paused.

The execution context associated with it is remembered in a special data structure called execution context stack.

The nested call executes.

After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.

Let’s see what happens during the pow(2, 3) call.

```
FUNCTION pow(x,n:INTEGER) : INTEGER
VAR
   result : INTEGER;
BEGIN
   IF (n = 1) THEN
       RETURN x;
   ELSE
       result := pow(x, n-1);
       RETURN x* result;
   END_IF
END
```

- **pow(2, 3)**

  In the beginning of the call pow(2, 3) the execution context will store variables: x = 2, n = 3, the execution flow is at line 1 of the function.

  We can sketch it as:

  - **Context: { x: 2, n: 3, at line 1 } pow(2, 3)**

    That’s when the function starts to execute. The condition n == 1 is false, so the flow continues into the second branch of if statement.

    The variables are same, but the line changes, so the context is now:

  - **Context: { x: 2, n: 3, at line 5 } pow(2, 3)**

    To calculate x * pow(x, n - 1), we need to make a sub-call of pow with new arguments pow(2, 2).

- **pow(2, 2)**

  To do a nested call, JavaScript remembers the current execution context in the execution context stack.

  Here we call the same function pow, but it absolutely doesn’t matter. The process is the same for all functions:

1. The current context is “remembered” on top of the stack.
2. The new context is created for the subcall.
3. When the subcall is finished – the previous context is popped from the stack, and its execution continues.
4. Here’s the context stack when we entered the subcall pow(2, 2):
   - Context: { x: 2, n: 2, at line 1 } pow(2, 2)
   - Context: { x: 2, n: 3, at line 5 } pow(2, 3)

The new current execution context is on top (and bold), and previous remembered contexts are below.

When we finish the subcall – it is easy to resume the previous context, because it keeps both variables and the exact place of the code where it stopped.

* **pow(2, 1)**

The process repeats: a new subcall is made at line 5, now with arguments x=2, n=1.

A new execution context is created. The previous one is pushed on top of the stack.

- Context: { x: 2, n: 1, at line 1 } pow(2, 1)
- Context: { x: 2, n: 2, at line 5 } pow(2, 2)
- Context: { x: 2, n: 3, at line 5 } pow(2, 3)

  There are 2 old contexts now and 1 currently running for pow(2, 1).

  - **The exit**

    During the execution of pow(2, 1), unlike before, the condition n == 1 is truthy, so the first branch of if works.

    Now we have all the result we need, so we are going to change the function calls in the stack by the returned result and then we remove it from the stack.

‹ Previous

next ›