Fibonacci refactored

Dr. Dominik Schemmel

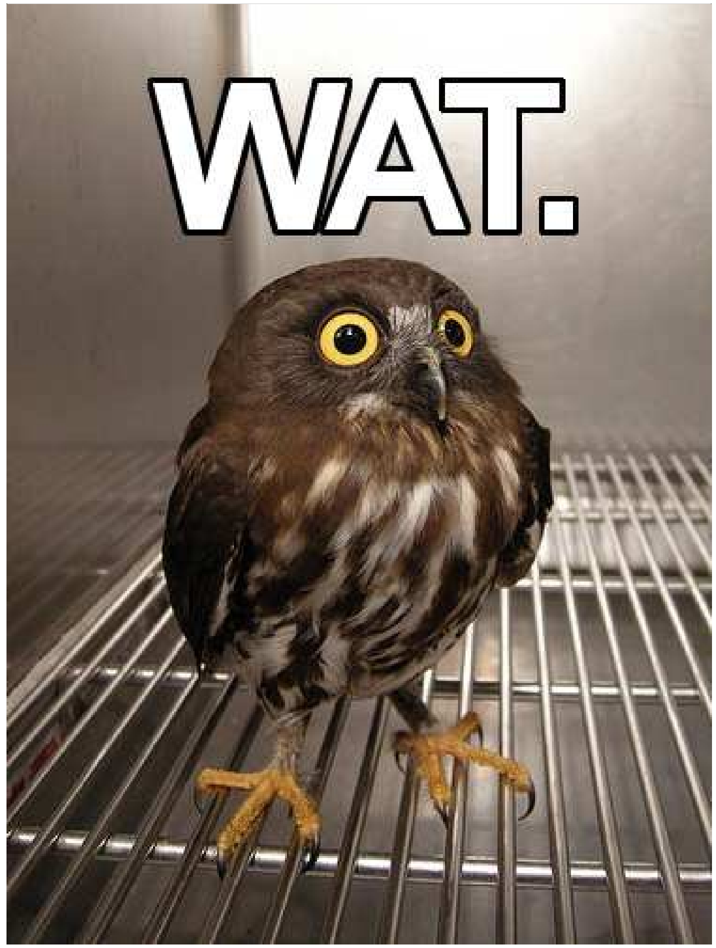
MNUG, 2014-02-20

TNG TECHNOLOGY CONSULTING

“I've combined (for fun) a few WTFs to make something bigger.Just in case... don't code in JS like that. Seriously.”

http://wtfjs.com/2013/02/12/obfuscated-fibonacci

```
function fibonacci(_) {
    for(_=[+[],++[[]][+[]],+[],_],_[++[++[++[[]][+[]]][+[]]][+[]]]=(((_[++[++
    [[]][+[]]][+[]]][+[]]-(++[[]][+[]]))&(((--[[]][+[]])>>>(++[[]][+[]]))))=
    (_[++[++[++[[]][+[]]][+[]]][+[]]-(++[[]][+[]])))?(_[++[++[[]][+[]]][+[]]
    ++[[]][+[]],_[++[++[++[[]][+[]]][+[]]][+[]]-(++[[]][+[]])):+[];_[++[++[+
    ]][+[]]][+[]]][+[]]--;_[+[]]=(_[++[[]][+[]]]=_[++[++[[]][+[]]][+[]]]=_[+
    +_[++[[]][+[]]])-_[+[]]);
    return _[++[++[[]][+[]]][+[]]];
}
```


WAT.

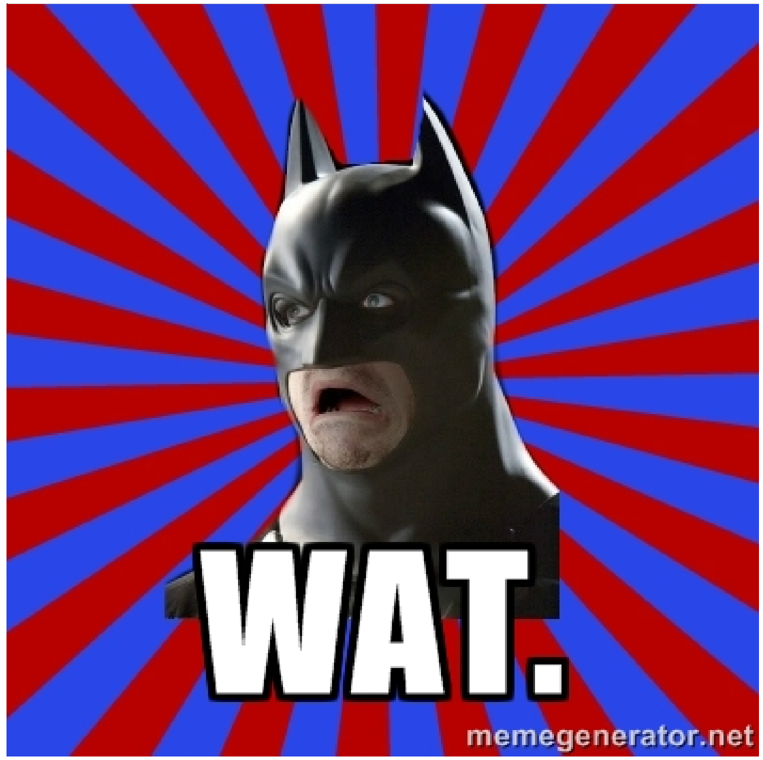
Rename argument `_` to `n`.

```
function fibonacci(_) {
    for(_=[+[],++[[]][+[]],+[],_],_[++[++[++[[]][+[]]][+[]]][+[]]]=(((_[++[++
    [[]][+[]]][+[]]][+[]]-(++[[]][+[]]))&(((--[[]][+[]])>>>(++[[]][+[]]))))=
    (_[++[++[++[[]][+[]]][+[]]][+[]]-(++[[]][+[]])))?(_[++[++[[]][+[]]][+[]]
    ++[[]][+[]],_[++[++[++[[]][+[]]][+[]]][+[]]-(++[[]][+[]])):+[];_[++[++[+
    ]][+[]]][+[]]][+[]]]--;_[+[]]=(_[++[[]][+[]]]=_[++[++[[]][+[]]][+[]]]=_[+
    +_[++[[]][+[]]])-_[+[]]);
    return _[++[++[[]][+[]]][+[]]];
}
```

`+[]`

```
function fibonacci(n) {
    for(n=[+[],++[[]][+[]],+[],n],n[++[++[++[[]][+[]]][+[]]][+[]]]=(((n[++[++
    [[]][+[]]][+[]]][+[]]-(++[[]][+[]]))&(((--[[]][+[]])>>>(++[[]][+[]]))))=
    (n[++[++[++[[]][+[]]][+[]]][+[]]]-(++[[]][+[]])))?(n[++[++[[]][+[]]][+[]]
    ++[[]][+[]],n[++[++[++[[]][+[]]][+[]]][+[]]]-(++[[]][+[]])):+[];n[++[++[+
    ]][+[]]][+[]]][+[]]]--;n[+[]]=(n[++[[]][+[]]]=n[++[++[[]][+[]]][+[]]]=n[+
    +n[++[[]][+[]]])-n[+[]]);
    return n[++[++[[]][+[]]][+[]]];
}
```

The unary `+` operator converts values to numbers, thus `+[]` is equal to `+Number([])`.
`Number([])` executes `[].valueOf()` and gets an object instead a primitive returned (that is `[]`).
Therefore it returns `[].toString()`, which is `""`.
Finally the operator can convert the empty string: `+""` is `0`.


WAT.
memegenerator.net

`++[[]][0]` equals `1`, `--[[]][0]` equals `-1`.
`++[++[[]][0]][0]` equals `2`.
`++[++[++[[]][0]][0]][0]` equals `3`.

```
function fibonacci(n) {
    for (
        n=[0,++[[]][0],0,n],
        n[++[++[++[[]][0]][0]][0]]=(((n[++[++[++[[]][0]][0]][0]]-(++[[]][0]))
            (((--[[]][0])>>>(++[[]][0]))))===(n[++[++[++[[]][0]][0]][0]]-
            (++[[]][0])))?(n[++[++[[]][0]][0]]=++[[]][0],
            n[++[++[++[[]][0]][0]][0]]-(++[[]][0])):0;
        n[++[++[++[[]][0]][0]][0]]--;
        n[0]=(n[++[[]][0]]=n[++[++[[]][0]][0]]=n[0]+n[++[[]][0]])-n[0]);
    return n[++[++[[]][0]][0]];
}
```

Similarly to the previous slide the array again is converted:
`++[[]][0]` gives `++Number([])`, which is `++""`.
For `++""` (and `--""`) the string again has to be converted, which gives `++0` (or `--0`), which is of course `1` (or `-1`).

`n` is reused.
So another variable `m` is extracted.
This block also can be put before the loop.

```
function fibonacci(n) {
    for (
        n = [0, 1, 0, n],
            n[3] = (((n[3] - (1)) & (((-1) >>> (1)))) === (n[3] - (1))) ?
                (n[2] = 1, n[3] - (1)) : 0;
        n[3]--;
        n[0] = (n[1] = n[2] = n[0] + n[1]) - n[0]
    );
    return n[2];
}
```

Reformulation of the loop's afterthought: `m[3]--` is `m[3]>0; m[3]--`.
It's allowed to use the body of the loop.

```
function fibonacci(n) {
    var m = [0, 1, 0, n];
    m[3] = (((m[3] - (1)) & (((-1) >>> (1)))) === (m[3] - (1))) ?
        (m[2] = 1, m[3] - (1)) : 0;
    for (;
        m[3]--;
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0]
    );
    return m[2];
}
```

`m[3]` can be eliminated.
`n-1` can be extracted.
The iterater can be extracted, too.

```
function fibonacci(n) {
    var m = [0, 1, 0, n];
    m[3] = (((m[3] - (1)) & (((-1) >>> (1)))) === (m[3] - (1))) ?
        (m[2] = 1, m[3] - (1)) : 0;
    for (; m[3] > 0; m[3]--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

`-1 >>> 1` is equal to `2147483647`.

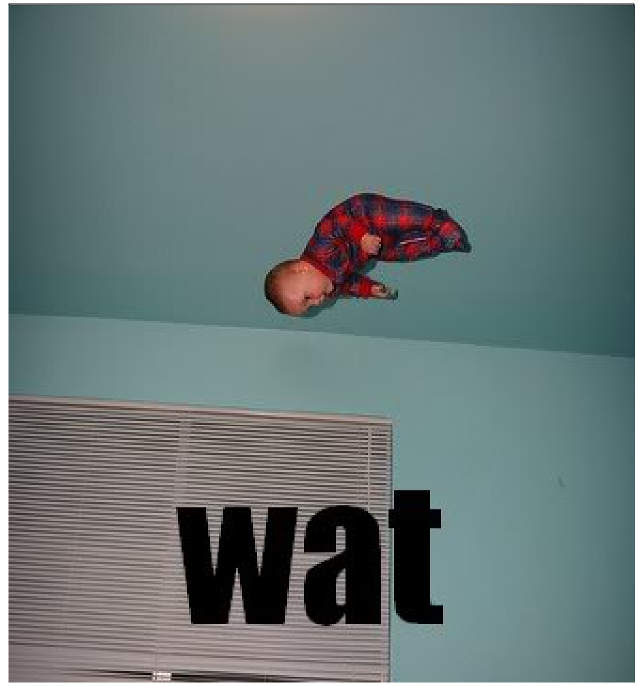
```
function fibonacci(n) {
    var m = [0, 1, 0],
        p = n - 1,
        q = ((p & (-1 >>> 1)) === p) ? (m[2] = 1, p) : 0;
    for (; q > 0; q--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

`>>>` is the zero-fill right shift bitwise operator (operands are converted to signed 32-bit integers in big endian order).
`-1` corresponds to binary `11111111 1111111 1111111 1111111` (first bit is the sign, the rest is the two's complement of 1).
Calculating the two's complement means to negate all bits, then add 1.
`-1 >>> 1` thus is binary `01111111 1111111 1111111 1111111` (which is decimal 2147483647).

`(p & 2147483647) === p` is equal to `p >= 0`.

```
function fibonacci(n) {
    var m = [0, 1, 0],
        p = n - 1,
        q = ((p & 2147483647) === p) ? (m[2] = 1, p) : 0;
    for (; q > 0; q--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

`p & 2147483647`: All non-sign bits are masked by the bitwise AND.
Since negative integers are represented as the two's complement of their absolute value, the expression `(p & 2147483647) === p` corresponds to `p >= 0`.


wat

`m[2] = 1, p` returns `p`.

```
function fibonacci(n) {
    var m = [0, 1, 0],
        p = n - 1,
        q = p >= 0 ? (m[2] = 1, p) : 0;
    for (; q > 0; q--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

All expressions separated by the comma operator are evaluated, then the result of the last expression is returned.

For `p =< 0` we never run through the loop, so we `return 0` prematurely.

`m[2] = [0, 1, 1]`

The loop directly can use `p` as its loop counter.

```
function fibonacci(n) {
    var m = [0, 1, 0],
        p = n - 1,
        q;
    if (p < 0) {
        q = 0;
    } else {
        m[2] = 1;
        q = p;
    }
    for (; q > 0; q--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

The loop is traversed `n-1` times and contains no reference in its body on the loop counter, so we can count in ascending order.

```
function fibonacci(n) {
    var m = [0, 1, 1];
    if (n < 1) {
        return 0;
    }
    for (var p = n - 1; p > 0; p--) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

This is separable in 3 expressions.

```
function fibonacci(n) {
    var m = [0, 1, 1];
    if (n < 1) {
        return 0;
    }
    for (var i = 1; i < n; i++) {
        m[0] = (m[1] = m[2] = m[0] + m[1]) - m[0];
    }
    return m[2];
}
```

The (right associative) assignment operator returns the assigned value.

`m` serves as buffer for 3 Fibonacci numbers.
In `m[2]` the next Fibonacci number is calculated from the previous two numbers, which are stored in `m[0]` and `m[1]`.
It is not necessary for the next iteration to calculate `m[1]` again, if `m[0]` and `m[1]` are assigned in the right order.

```
function fibonacci(n) {
    var m = [0, 1, 1];
    if (n < 1) {
        return 0;
    }
    for (var i = 1; i < n; i++) {
        m[2] = m[0] + m[1];
        m[1] = m[2];
        m[0] = m[1] - m[0];
    }
    return m[2];
}
```

Shifting elements in an array can be done better with `shift()`.
But then we must return `m[1]` at the end.

```
function fibonacci(n) {
    var m = [0, 1, 1];
    if (n < 1) {
        return 0;
    }
    for (var i = 1; i < n; i++) {
        m[2] = m[0] + m[1];
        m[0] = m[1];
        m[1] = m[2];
    }
    return m[2];
}
```

Cleanup:
Avoid magic numbers.
Leave `m[2]` empty initially.
Rename variables.

```
function fibonacci(n) {
    var m = [0, 1, 1];
    if (n < 1) {
        return 0;
    }
    for (var i = 1; i < n; i++) {
        m[2] = m[0] + m[1];
        m.shift();
    }
    return m[1];
}
```

```
function fibonacci(n) {
    var fib = [0, 1];
    if (n < 1) {
        return fib[0];
    }
    for (var i = 1; i < n; i++) {
        fib[2] = fib[0] + fib[1];
        fib.shift();
    }
    return fib[1];
}
```

(done :-)

Thank you for your attention!