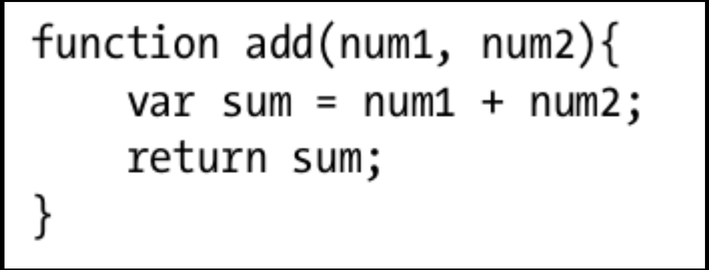
O(js)

Javascript Performance

Scope Resolution

Javascript 101

Minimize globals



```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

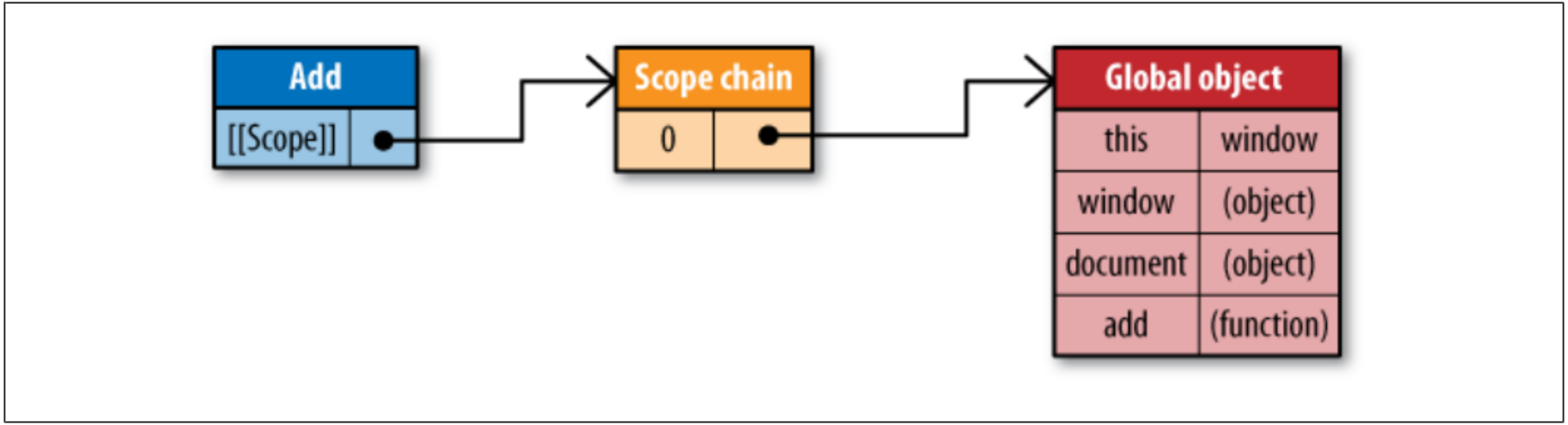
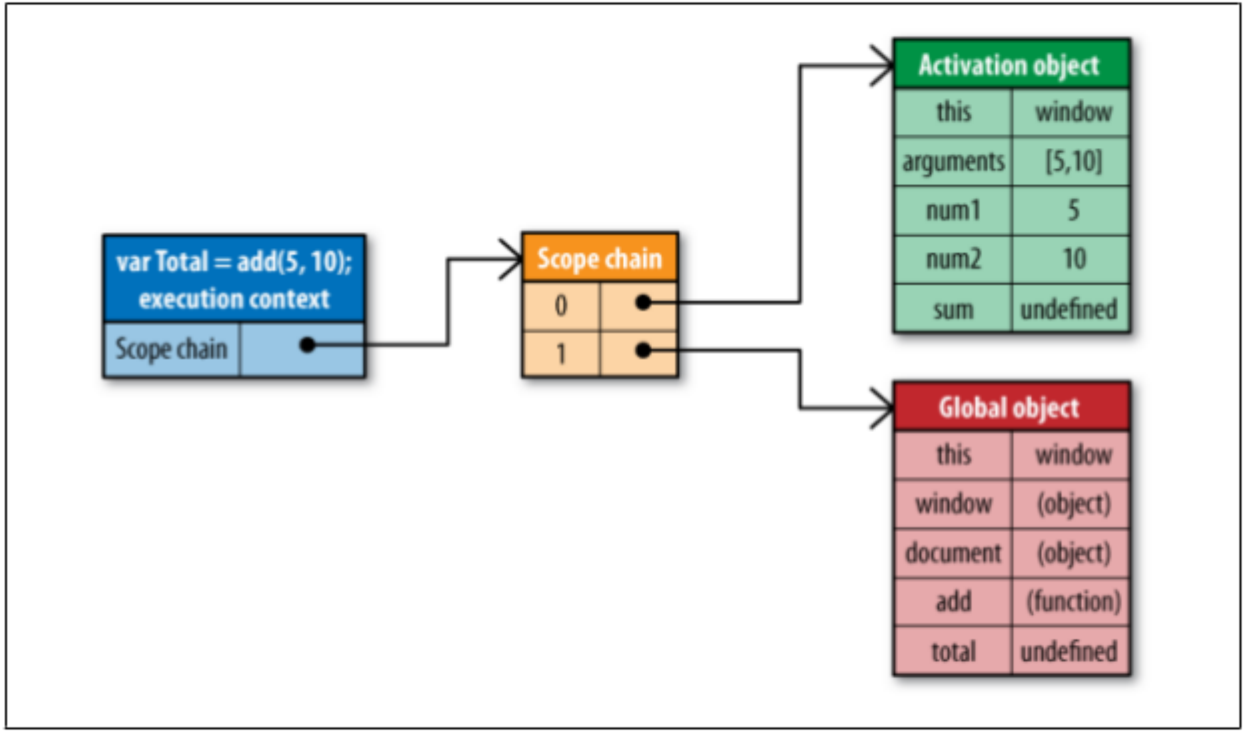


Figure 2-2. Scope chain for the add() function



var Total = add(5, 10);
execution context

Scope chain

Scope chain

0	●
1	●

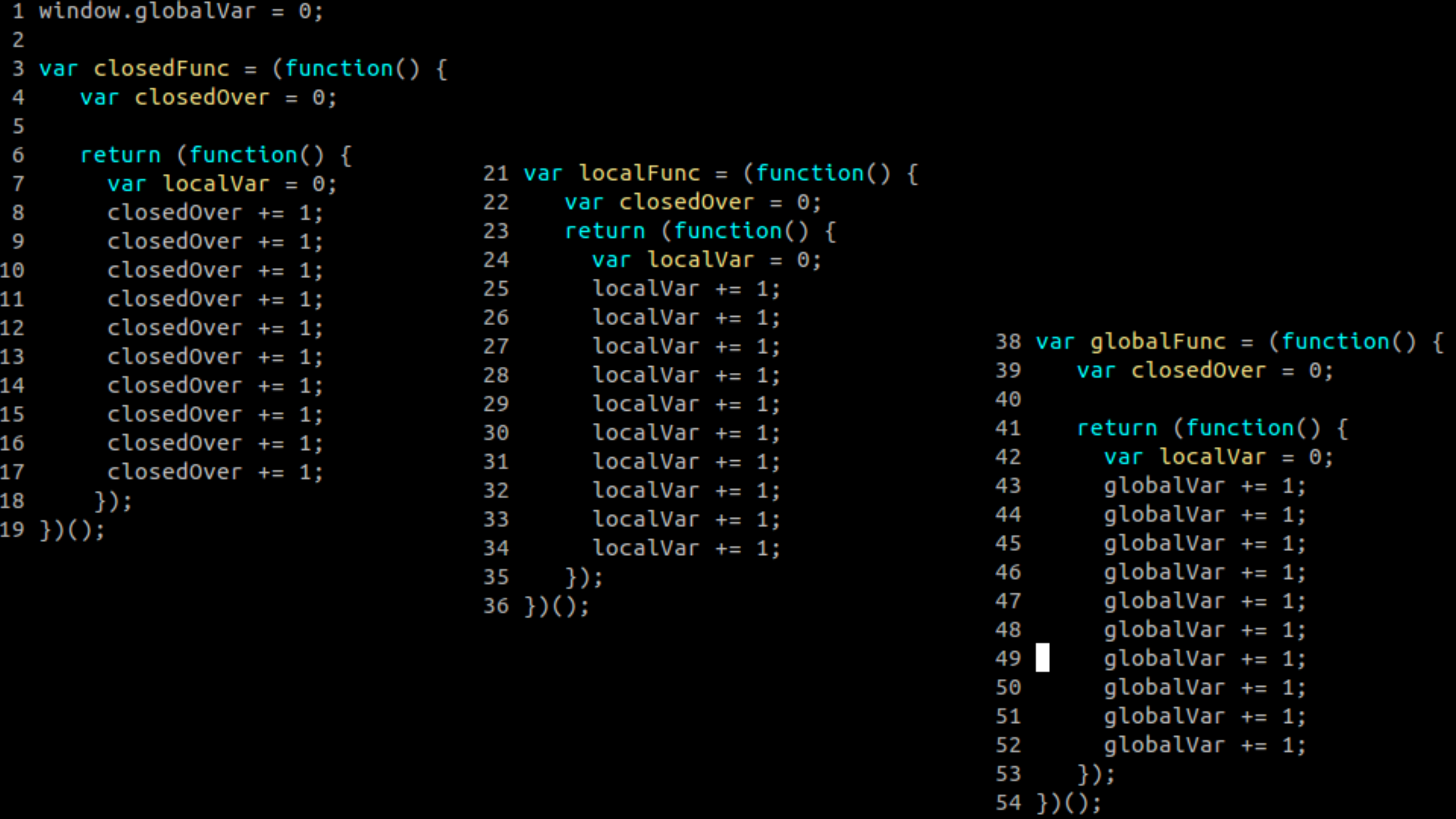
Activation object

this	window
arguments	[5,10]
num1	5
num2	10
sum	undefined

Global object

this	window
window	(object)
document	(object)
add	(function)
total	undefined

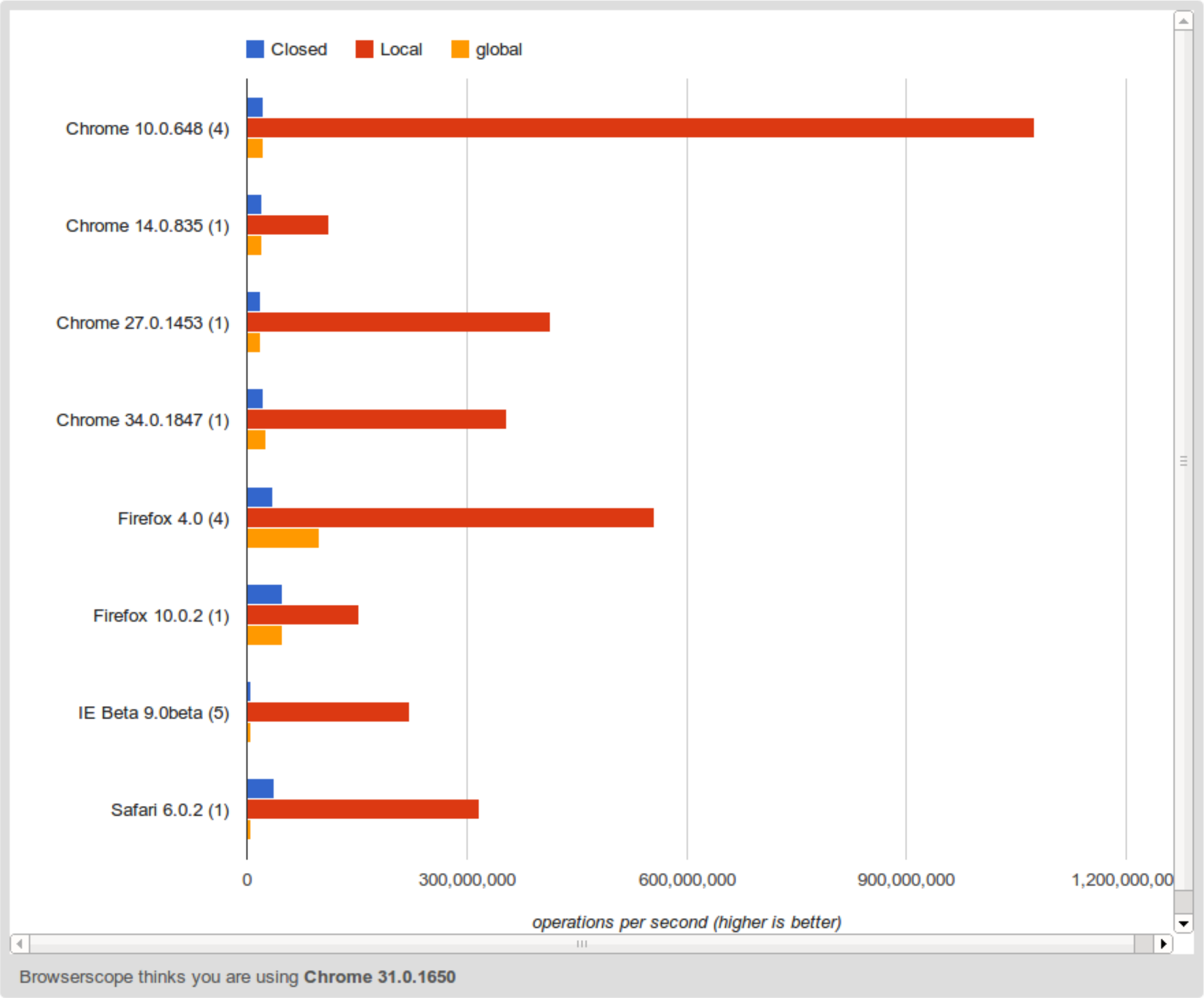
Figure 2-3. Scope chain while executing add()



```
window.globalVar = 0;

var closedFunc = (function() {
   var closedOver = 0;

   return (function() {
     var localVar = 0;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
     closedOver += 1;
    });
})();
```

```
var localFunc = (function() {
   var closedOver = 0;
   return (function() {
     var localVar = 0;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
     localVar += 1;
   });
})();
```

```
var globalFunc = (function() {
   var closedOver = 0;

   return (function() {
     var localVar = 0;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
     globalVar += 1;
   });
})();
```



Browserscope thinks you are using **Chrome 31.0.1650**

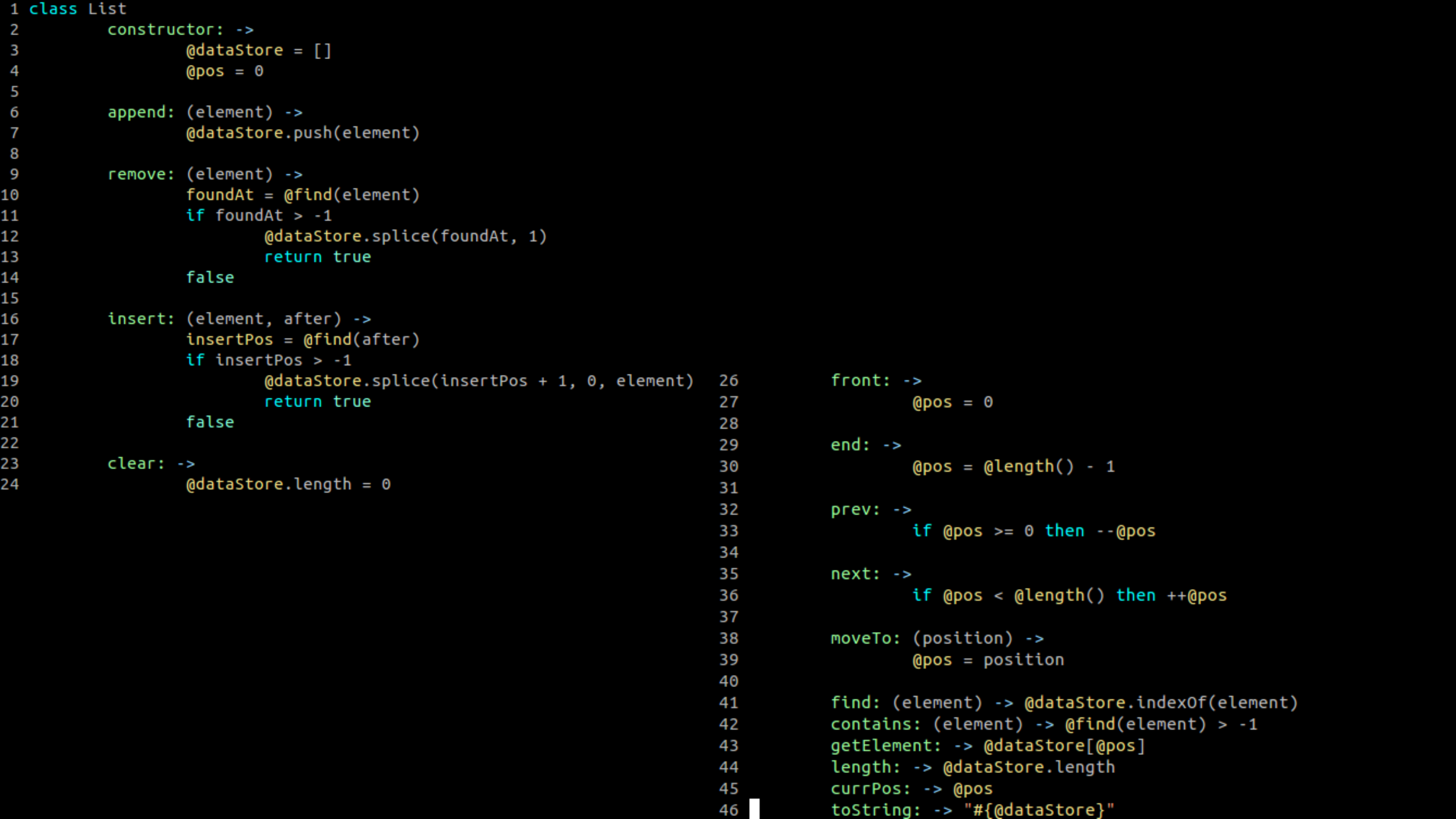
Data Structures

Lists,

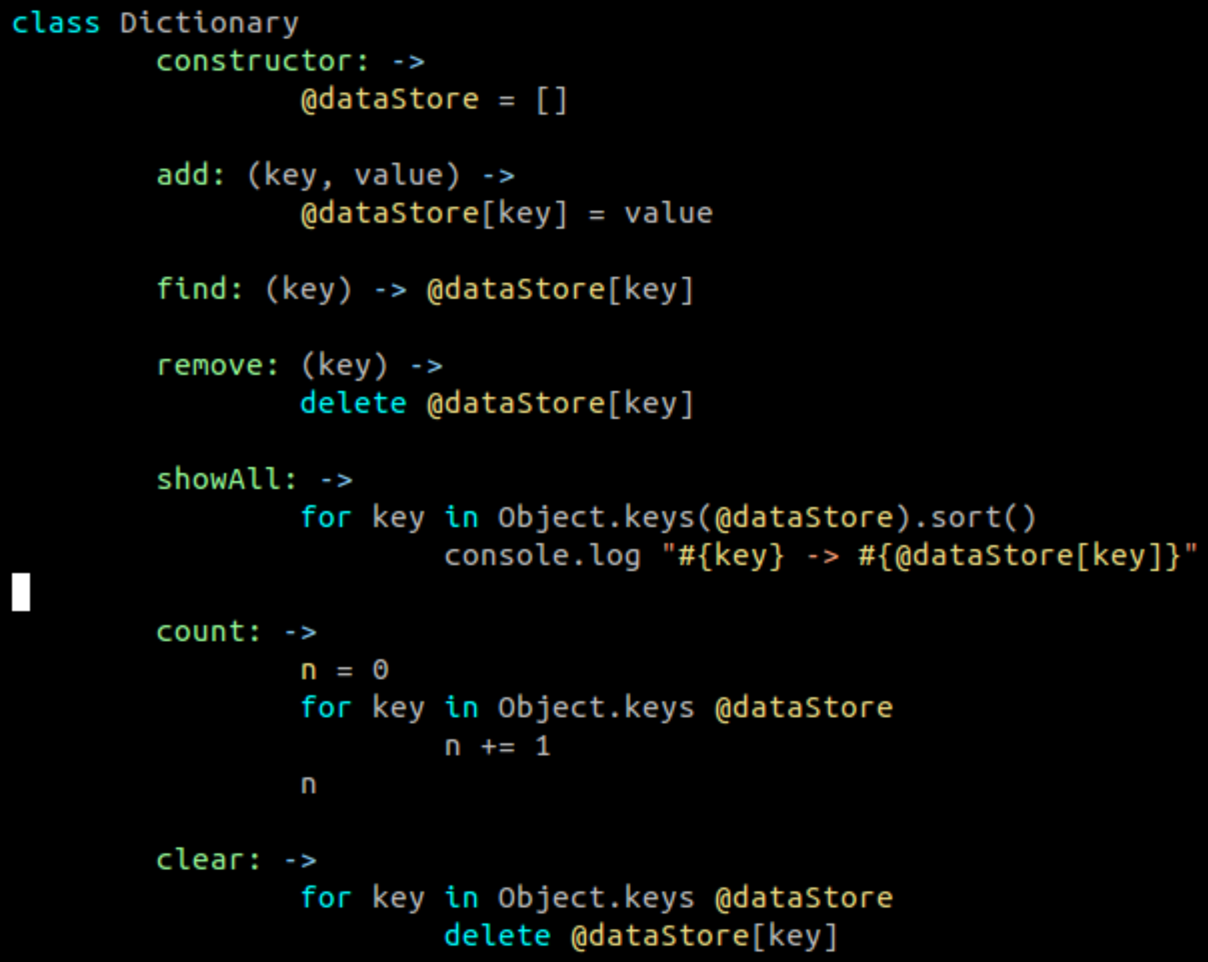
Dictionaries,

Arrays,

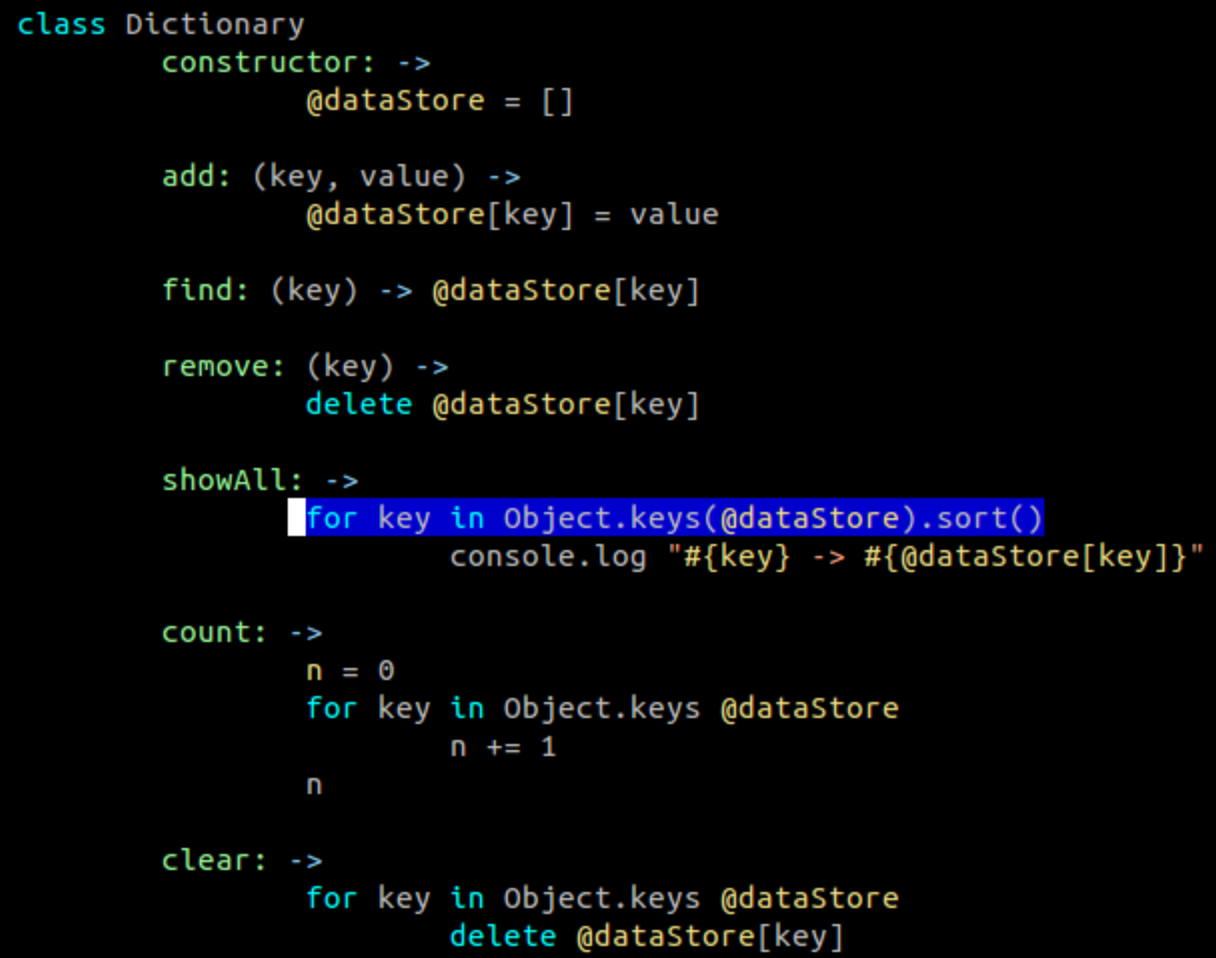
Oh my.



```
class List
        constructor: ->
                @dataStore = []
                @pos = 0

        append: (element) ->
                @dataStore.push(element)

        remove: (element) ->
                foundAt = @find(element)
                if foundAt > -1
                        @dataStore.splice(foundAt, 1)
                        return true
                false

        insert: (element, after) ->
                insertPos = @find(after)
                if insertPos > -1
                        @dataStore.splice(insertPos + 1, 0, element)
                        return true
                false

        clear: ->
                @dataStore.length = 0

        front: ->
                @pos = 0

        end: ->
                @pos = @length() - 1

        prev: ->
                if @pos >= 0 then --@pos

        next: ->
                if @pos < @length() then ++@pos

        moveTo: (position) ->
                @pos = position

        find: (element) -> @dataStore.indexOf(element)
        contains: (element) -> @find(element) > -1
        getElement: -> @dataStore[@pos]
        length: -> @dataStore.length
        currPos: -> @pos
        toString: -> "#{@dataStore}"
```



```
class Dictionary
        constructor: ->
                @dataStore = []

        add: (key, value) ->
                @dataStore[key] = value

        find: (key) -> @dataStore[key]

        remove: (key) ->
                delete @dataStore[key]

        showAll: ->
                for key in Object.keys(@dataStore).sort()
                        console.log "#{key} -> #{@dataStore[key]}"

        count: ->
                n = 0
                for key in Object.keys @dataStore
                        n += 1
                n

        clear: ->
                for key in Object.keys @dataStore
                        delete @dataStore[key]
```

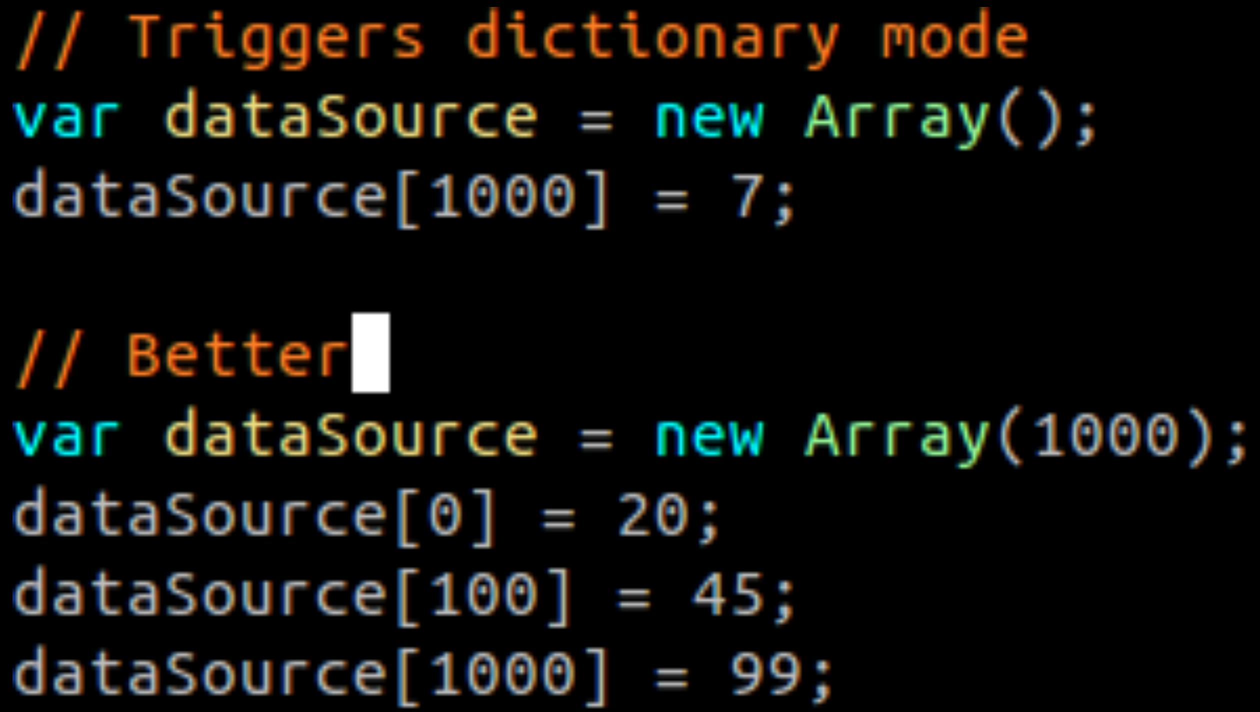


```
class Dictionary
        constructor: ->
                @dataStore = []

        add: (key, value) ->
                @dataStore[key] = value

        find: (key) -> @dataStore[key]

        remove: (key) ->
                delete @dataStore[key]

        showAll: ->
                for key in Object.keys(@dataStore).sort()
                        console.log "#{key} -> #{@dataStore[key]}"

        count: ->
                n = 0
                for key in Object.keys @dataStore
                        n += 1
                n

        clear: ->
                for key in Object.keys @dataStore
                        delete @dataStore[key]
```

Arrays

“An array is a linear allocation of memory in which elements are accessed by integers that are used to compute offsets. Arrays can be very fast data structures.

Unfortunately, JavaScript does not have anything like this kind of array.”



```
// Triggers dictionary mode
var dataSource = new Array();
dataSource[1000] = 7;

// Better
var dataSource = new Array(1000);
dataSource[0] = 20;
dataSource[100] = 45;
dataSource[1000] = 99;
```

When to use recursion?

a.k.a **Functional vs. Imperative**

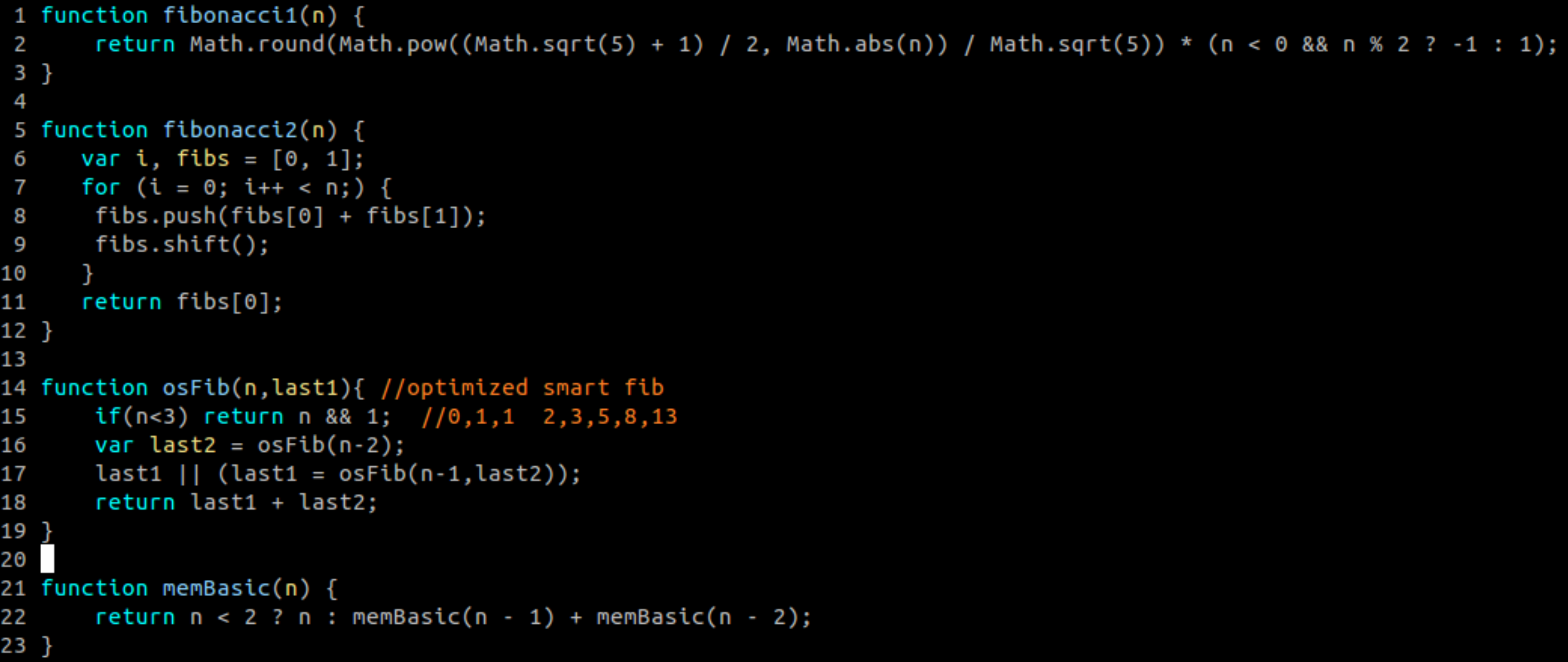
Max Stack Call:

Chrome 31 -> ~25000

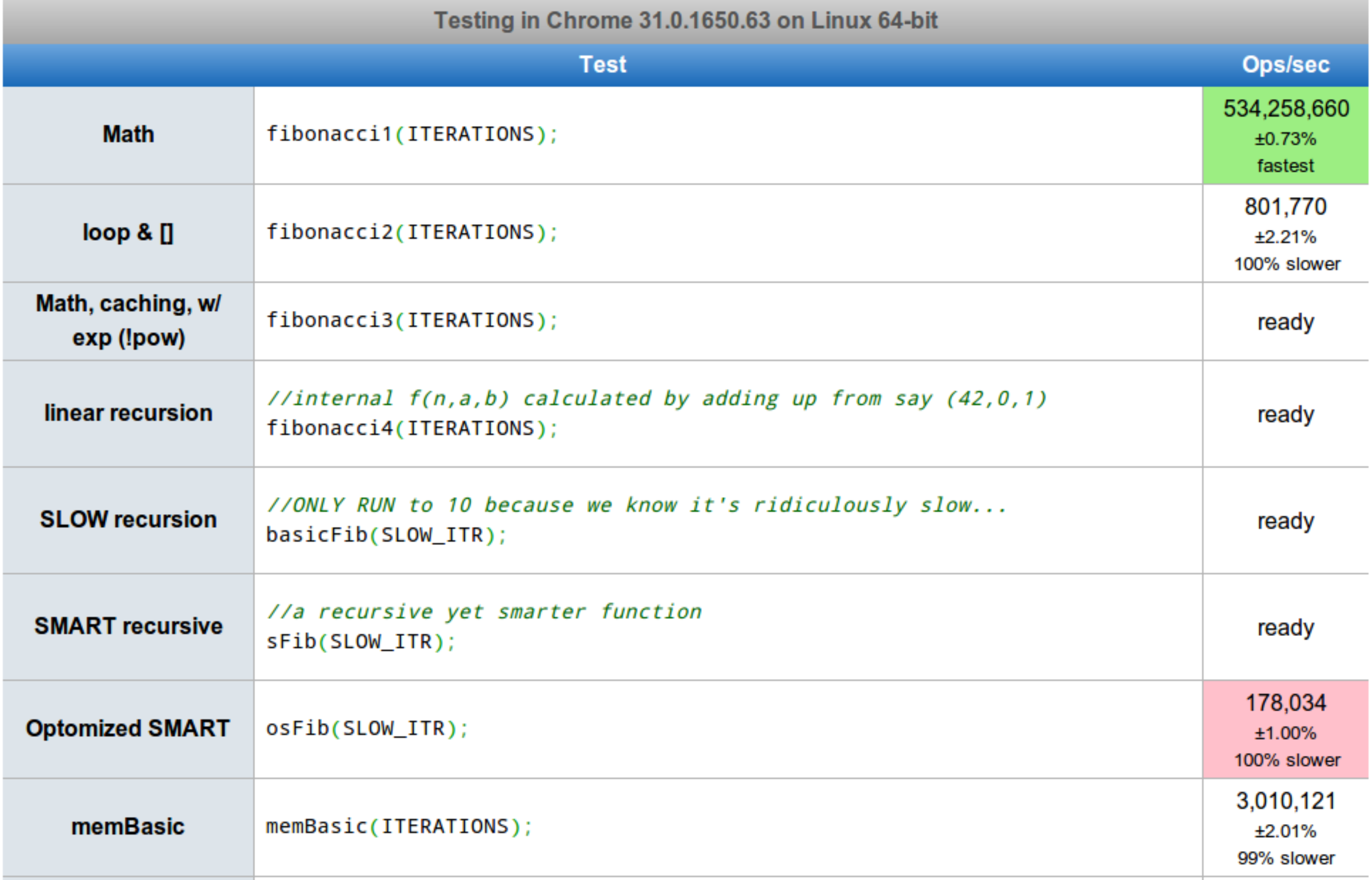
Firefox 26 -> ~50000

IE10 -> ~20000

Safari -> ~65000



```
function fibonacci1(n) {
    return Math.round(Math.pow((Math.sqrt(5) + 1) / 2, Math.abs(n)) / Math.sqrt(5)) * (n < 0 && n % 2 ? -1 : 1);
}

function fibonacci2(n) {
   var i, fibs = [0, 1];
   for (i = 0; i++ < n;) {
    fibs.push(fibs[0] + fibs[1]);
    fibs.shift();
   }
   return fibs[0];
}

function osFib(n,last1){ //optimized smart fib
    if(n<3) return n && 1;  //0,1,1  2,3,5,8,13
    var last2 = osFib(n-2);
    last1 || (last1 = osFib(n-1,last2));
    return last1 + last2;
}

function memBasic(n) {
    return n < 2 ? n : memBasic(n - 1) + memBasic(n - 2);
}
```



Testing in Chrome 31.0.1650.63 on Linux 64-bit

Test		Ops/sec
Math	`fibonacci1(ITERATIONS);`	534,258,660 ±0.73% fastest
loop & []	`fibonacci2(ITERATIONS);`	801,770 ±2.21% 100% slower
Math, caching, w/ exp (!pow)	`fibonacci3(ITERATIONS);`	ready

linear recursion	`//internal f(n,a,b) calculated by adding up from say (42,0,1)`
	`fibonacci4(ITERATIONS);`
		ready

SLOW recursion	`//ONLY RUN to 10 because we know it's ridiculously slow...`
	`basicFib(SLOW_ITR);`
		ready

SMART recursive	`//a recursive yet smarter function`
	`sFib(SLOW_ITR);`
		ready

Optomized SMART	`osFib(SLOW_ITR);`	178,034 ±1.00% 100% slower
memBasic	`memBasic(ITERATIONS);`	3,010,121 ±2.01% 99% slower

Pioneers of Functional Prog. in JS

UNDERSCORE.JS

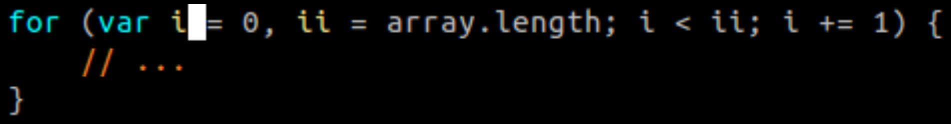
Lo-Dash

CoffeeScript

Coco

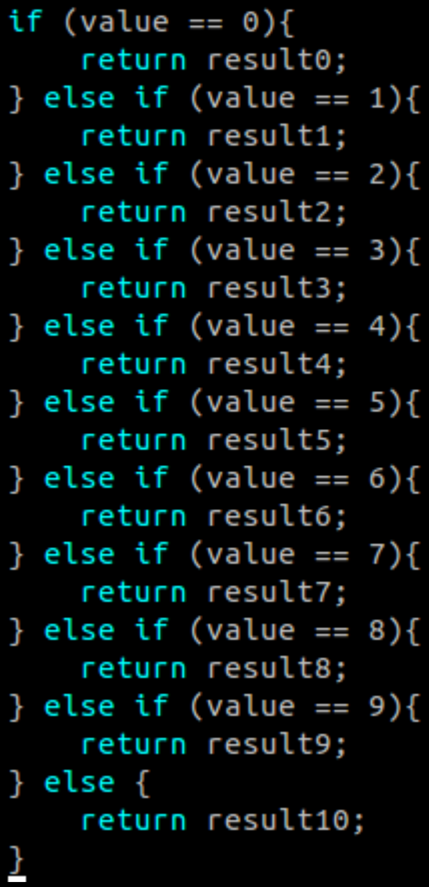
Micro Optimizations

For loops

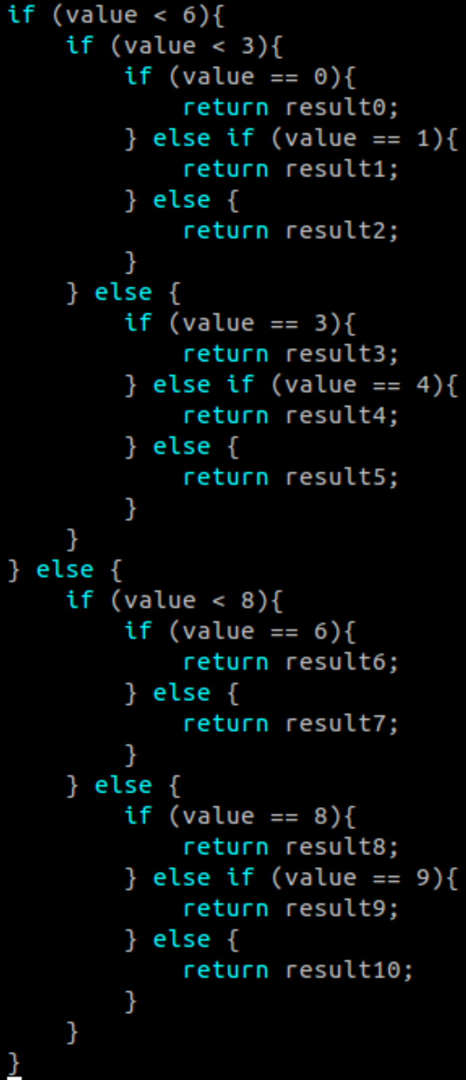


```
for (var i = 0, ii = array.length; i < ii; i += 1) {
    // ...
}
```

If-else

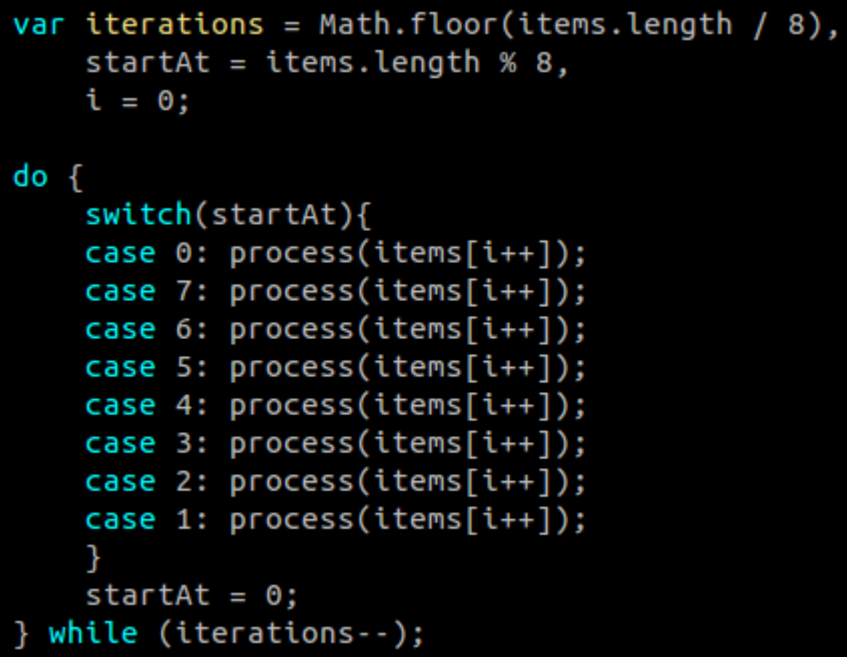


```
if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 5){
    return result5;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else if (value == 9){
    return result9;
} else {
    return result10;
}
```



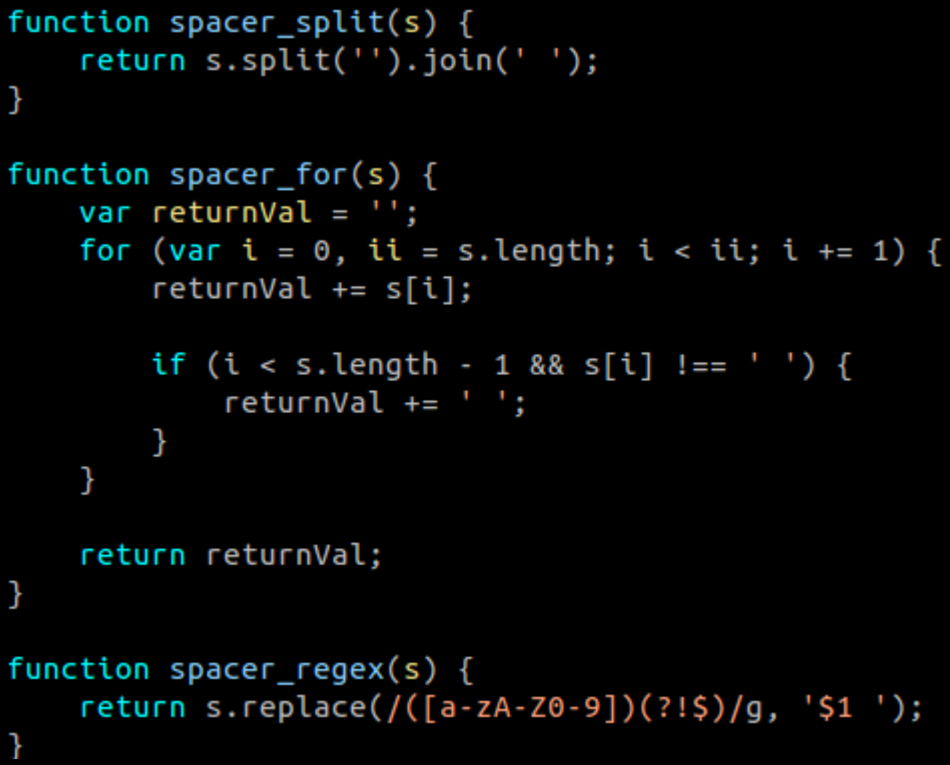
```
if (value < 6){
    if (value < 3){
        if (value == 0){
            return result0;
        } else if (value == 1){
            return result1;
        } else {
            return result2;
        }
    } else {
        if (value == 3){
            return result3;
        } else if (value == 4){
            return result4;
        } else {
            return result5;
        }
    }
} else {
    if (value < 8){
        if (value == 6){
            return result6;
        } else {
            return result7;
        }
    } else {
        if (value == 8){
            return result8;
        } else if (value == 9){
            return result9;
        } else {
            return result10;
        }
    }
}
```

Duff Device

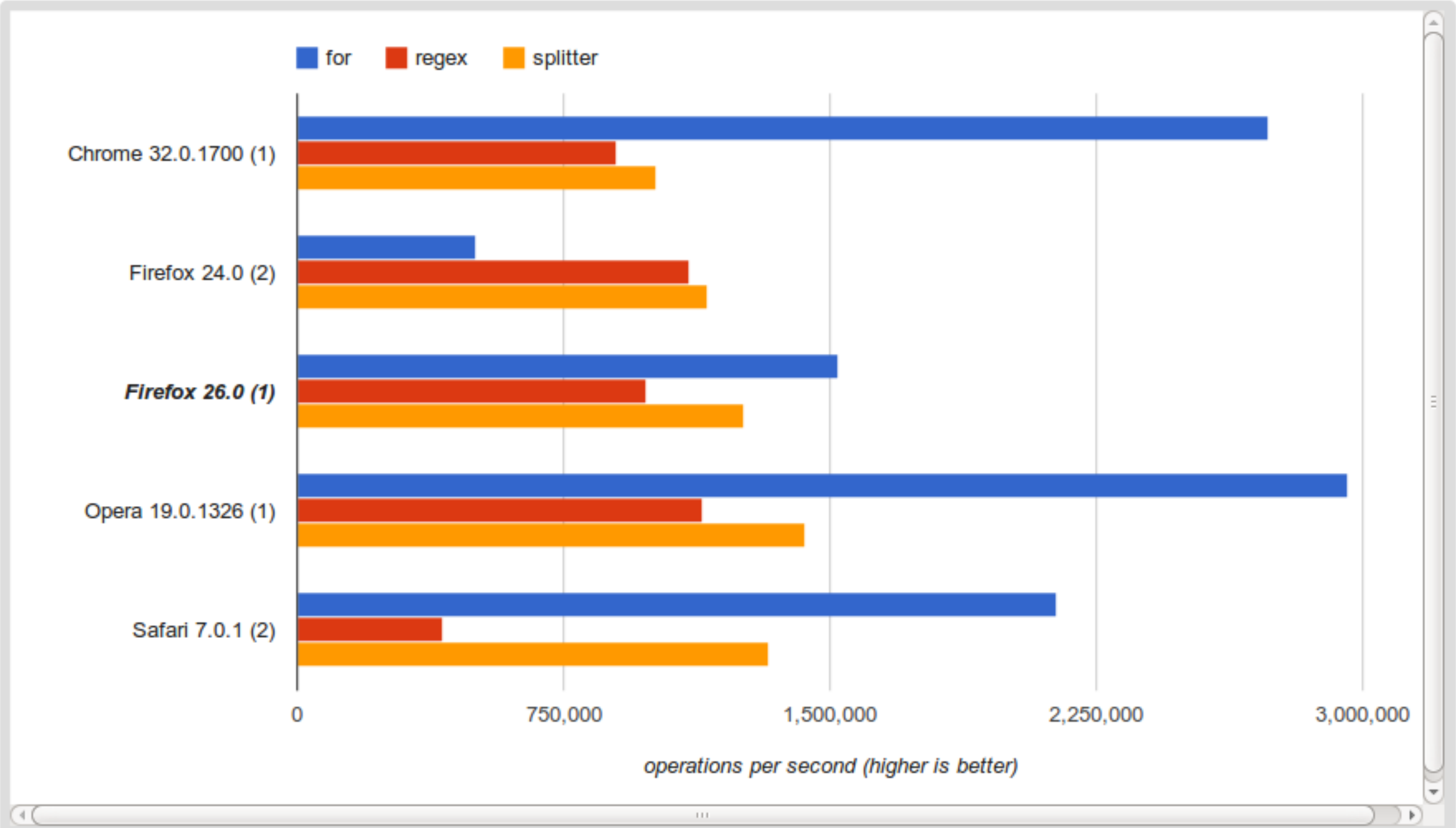


```
var iterations = Math.floor(items.length / 8),
    startAt = items.length % 8,
    i = 0;

do {
    switch(startAt){
    case 0: process(items[i++]);
    case 7: process(items[i++]);
    case 6: process(items[i++]);
    case 5: process(items[i++]);
    case 4: process(items[i++]);
    case 3: process(items[i++]);
    case 2: process(items[i++]);
    case 1: process(items[i++]);
    }
    startAt = 0;
} while (iterations--);
```

Elegance vs Performance



```
function spacer_split(s) {
    return s.split('').join(' ');
}

function spacer_for(s) {
    var returnVal = '';
    for (var i = 0, ii = s.length; i < ii; i += 1) {
        returnVal += s[i];

        if (i < s.length - 1 && s[i] !== ' ') {
            returnVal += ' ';
        }
    }

    return returnVal;
}

function spacer_regex(s) {
    return s.replace(/([a-zA-Z0-9])(?!$)/g, '$1 ');
}
```


for
regex
splitter
Chrome 32.0.1700 (1)
Firefox 24.0 (2)
Firefox 26.0 (1)
Opera 19.0.1326 (1)
Safari 7.0.1 (2)
0
750,000
1,500,000
2,250,000
3,000,000
operations per second (higher is better)

Ending Credits

High Performance Javascript. Oldie but goodie.

Data Structures and Algorithms With Javascript. Has some good parts.

Javascript Patterns. 5/5.

JS: The Good Parts.

Thanks for listening and Cheers! Questions?