DDG University

Informal Goals

1. Learn new things related to technology.
2. Learn from each other.
3. Foster inter-team building.
4. To become better engineers.

Search for DDG University in Asana.

Structure and Interpretation of Computer Programs *(SICP)*

by Harold Abelson and Gerald Jay Sussman

1.2 Procedures and the Processes They Generate

1. ***The shape of recursion and iteration***
2. ***Orders of growth***
3. ***Big O Notation***
4. ***Greatest Common Divisors***
5. ***Improving Prime Number Searching***

Factorials

```
n! = n * (n-1) * (n-2) ... 3 * 2 * 1

6! = 6 * 5 * 4 * 3 * 2 * 1 = 120
```

Recursive Process

```scheme
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Visualized with substitution model expansion:

```scheme
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

1. Defers execution
2. State is on the stack
3. Uses space equal to `n`

Iterative Process

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Visualized with substitution model expansion:

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

1. Execution is not deferred
2. State is held variables
3. Uses constant space

Recursive Process vs. Procedure

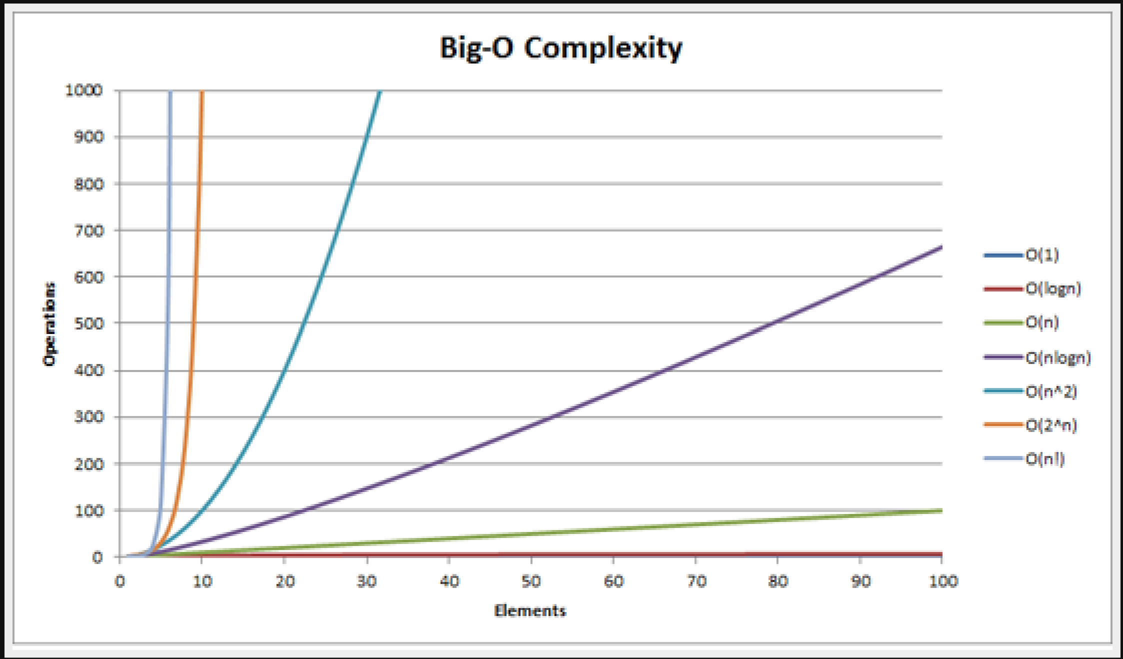
- A recursive *procedure* function that refers to itself. This is purely syntactic.
- A recursive *process* describes the way a function evolves in time and space.

Orders of Growth

We can describe the *shapes* of the above processes with the notion of *order of growth*. We are looking to measure the resources required as the input increases.

Big O Notation

Describes the growth behaviour of a function relative to its input.



Big ~~Oh~~ Theta Notation

Actually we're talking about Big Theta (Θ) notation. O() measures the upper bound of growth whereas Θ() measures the exact growth.

$F(n) = O(n^3)$ - $F(n)$ grows no faster than n^3

$F(n) = \Theta(n^3)$ - $F(n)$ grows as fast as n^3

The more you know!

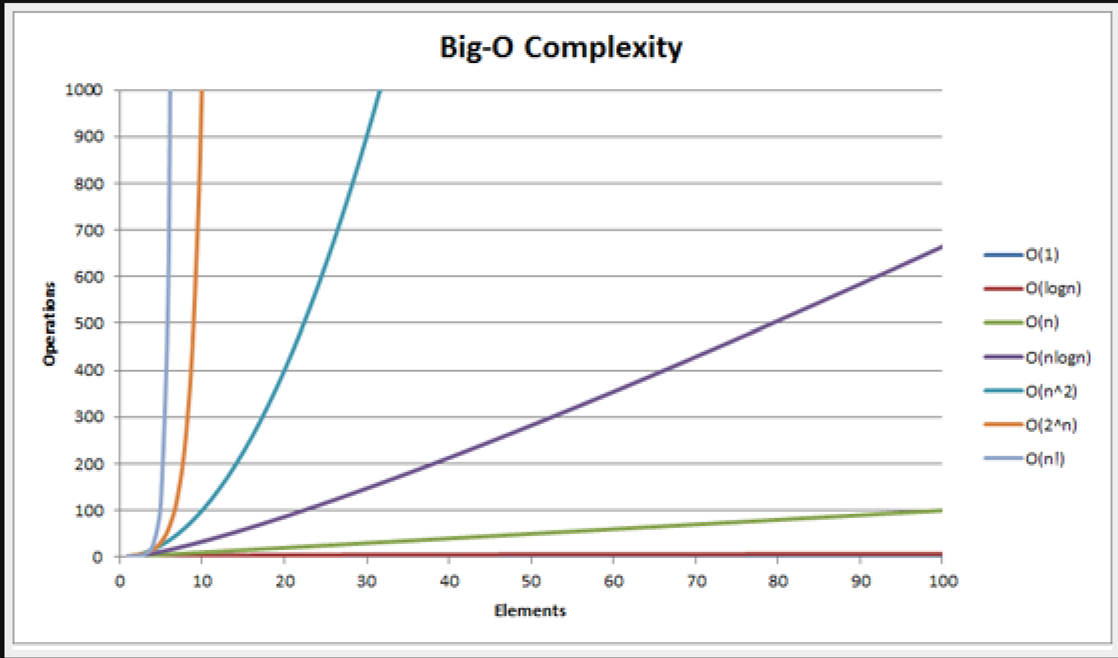
Big O Notation

Describes the growth behaviour of a function relative to its input.

If we were to measure the execution time of a function:

- *O(1)*: If the input size changes, the execution stays the same.
- *O(n)*: If the input size doubles, the execution time doubles.
- $O(n^2)$: If the input size doubles, the execution time quadruples.
- *O(log n)*: If the input size doubles, the execution time increases by one.
- $O(2^n)$: If the input size increases by one, the execution time doubles.

Big O Notation *(cont.)*



http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o

Recursive Growth

```
(factorial 6)                                |
(* 6 (factorial 5))                          |
(* 6 (* 5 (factorial 4)))                    |
(* 6 (* 5 (* 4 (factorial 3))))              |
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))        |
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))  | time (steps)
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))              |
(* 6 (* 5 (* 4 (* 3 2))))                    |
(* 6 (* 5 (* 4 6)))                          |
(* 6 (* 5 24))                               |
(* 6 120)                                    |
720                                          v
--------------------------------------------->
                  space (memory)
```

time = O(n)

space = O(n)

Iterative Growth

```
(factorial 6)           |
(fact-iter 1 1 6)       |
(fact-iter 1 2 6)       |
(fact-iter 2 3 6)       |
(fact-iter 6 4 6)       | time (steps)
(fact-iter 24 5 6)      |
(fact-iter 120 6 6)     |
(fact-iter 720 7 6)     |
720                     v
---------------------->
    space (memory)
```

time = O(n)

space = O(1)

Greatest Common Divisors

The GCD of two integers a and b is defined to be the largest integer that divides both a and b with no remainder.

Euclid's Algorithm:

```
GCD(a,b) = GCD(b,r)
```

For example:

```
GCD(206,40)= GCD(40,6)
           = GCD(6,4)
           = GCD(4,2)
           = GCD(2,0) = 2
```

An iterative process to express this:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

time = O(log n)

space = O(1)

Improving Primality Tests

Our benchmarking code:

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))

(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

We are going to be measuring `(prime?)` for 10^{10}, 10^{11}, 10^{12}, and 10^{13}.

Improving Primality Tests *(cont.)*

First iteration:

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n)
         n)
        ((divides? test-divisor n)
         test-divisor)
        (else (find-divisor
           n
           (+ test-divisor 1)))))     <---------- SLOW!!!

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))
```

When testing for primality you only need to check divisors 2 thru √n.
So for 81 you only need to check 2 thru 9.

This means that our first iteration is O(√n) time.

Improving Primality Tests *(cont.)*

First iteration results:

```
> (timed-prime-test 10000000019)
10000000019 *** .09

> (timed-prime-test 100000000003)
100000000003 *** .26

> (timed-prime-test 1000000000039)
1000000000039 *** .8

> (timed-prime-test 10000000000037)
10000000000037 *** 2.51
```

We can do better.

Improving Primality Tests *(cont.)*

Second iteration:

```
(define (next n)
  (if (= n 2) 3 (+ n 2)))
```

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n)
         n)
        ((divides? test-divisor n)
         test-divisor)
        (else (find-divisor
           n
           (next test-divisor)))))     <--------- LESS SLOW!!!

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))
```

You would think that our second iteration is $O(\sqrt{n}/2)$ time. But it's not quite twice as fast.

Improving Primality Tests *(cont.)*

Second iteration results:

```
> (timed-prime-test 10000000019)
10000000019 *** .06000000000000005

> (timed-prime-test 100000000003)
100000000003 *** .15999999999999997

> (timed-prime-test 1000000000039)
1000000000039 *** .5

> (timed-prime-test 10000000000037)
10000000000037 *** 1.6000000000000005
```

We can still do better.

Improving Primality Tests *(cont.)*

Third iteration:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m))))

(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

(define (prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n)
         (prime? n (- times 1)))
        (else false)))
```

Our final iteration is O(log n) time, or fast as hell.

Improving Primality Tests *(cont.)*

Third iteration results:

```
> (timed-prime-test 10000000019)
10000000019 *** 9.999999999999787e-3

> (timed-prime-test 100000000003)
100000000003 *** 9.999999999999787e-3

> (timed-prime-test 1000000000039)
1000000000039 *** 1.0000000000000675e-2

> (timed-prime-test 10000000000037)
10000000000037 *** 9.999999999999787e-3
```

That's all for section 1.2.

Thanks!