1

Efficiency Considerations

When is a function more efficient than another?

When it requires fewer computational resources to solve the same problem

Computational resources -- CPU operations, memory locations, (closer to home) **function applications**

But what if fcn_1 does better for arguments x_1 and function fcn_2 does better for arguments x_2?

• We are interested in estimating computational requirements of a function as the size of arguments grows

• One function is more efficient than another if its computational requirements grow more slowly with the size of arguments

The "size" of an argument is typically measured in terms of

- the length of a list
- the value of an integer
- a count of some feature

©1993 John Mylopoulos

2

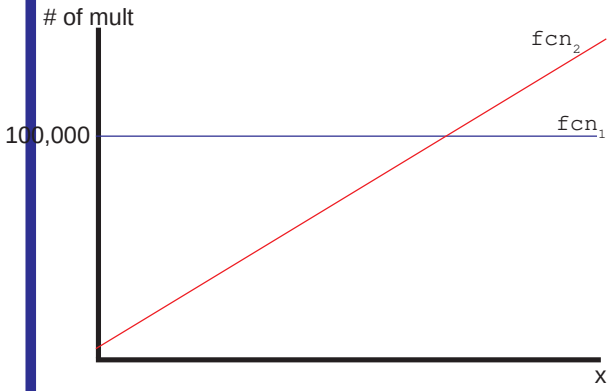
Computational Complexity

how fast do computational requirements for a function grow with the size of its arguments

Consider

$fcn_1(x) = c_1 * c_2 * \ldots * c_{100,000}$

$fcn_2(x) = x!$



We are interested in grouping functions into complexity classes, depending on whether their computational requirements remain constant for any argument (like fcn_1), grow linearly (like fcn_2), grow polynomially, etc.

©1993 John Mylopoulos

3

Complexity Classes

Constant Order - O(k)

The maximum number of function applications needed to compute such a function is independent of the size of the arguments.

Generally implies no recursion. For example:

```
(defun simple (x) (car (cdr x)))
```

Linear Order - O(n), where n is size of argument.

Generally this implies one recursive call, but constant work during each function application. For example:

```
(defun append (l1 l2)
 (cond  ((null l1) l2)
        (t (cons (first l1)
           (append (rest  l1)  l2) ))
    ))
```

[here `n` is the length of the list `l1`]

©1993 John Mylopoulos

4

Complexity Classes

Let's figure out the complexity of `append`

```
(defun  append (x y)
 (cond ((null x)   y)
       (t  (cons (car x)
             (append (cdr x) y) )))
```

Let Compl(`append`, n) represent the complexity function for `append` when given a first argument of length n. Then

Compl(`append`, n) = Compl(`append`, n - 1) + 5
...
Compl(`append`, 0) = 2

Therefore Compl(`append`, n) = O(n) (= 5*n + 1)

In general, any complexity function of the form (k*n + b) grows linearly with n -- assuming k, b are constants -- and we don't need to worry about the size of these constants.

[Note: The constant 5 counts builtin function calls]

©1993 John Mylopoulos

5

Complexity Classes

Polynomial Order - $O(n^2)$, $O(n^3)$, $O(n^4)$, ...
where n is the size of the argument.

```
(defun  reverse (x)
   (cond  ((null x) nil)
          (t   (append (reverse (cdr x))
                       (list (car x))) ))
```

Compl(`reverse`, n) =
Compl(`append`, n - 1) + Compl(`reverse`, n - 1) + 5 =
Compl(`reverse`, n - 1) + 5*n + 1 =
Compl(`reverse`, n - 2) + (5*n + 1) + (5*(n - 1) + 1) =
....
= 5*(n + (n - 1) + (n - 2) + ...+ 1) + n =
= 5*n(n + 1)/2 + n = $(5/2)n^2$ + 7n/2

Therefore Compl(`reverse`, n) = $O(n^2)$

©1993 John Mylopoulos

6

Contrast this with `revloop` , the helping function introduced for the more efficient version of `reverse`:

```
(defun revloop (y answer)
   (cond ((null y)  answer)
         (t (revloop   (cdr y)
            (cons (car y) answer)) )))
```

Compl(`revloop`, n) = Compl(`revloop`, n - 1) + 5

Therefore, Compl(`revloop`, n) = O(n)

©1993 John Mylopoulos

7

Efficiency in General

Exponential Order - $O(2^n)$, $O(n^n)$, ...

Generally implies recursion with more that one recursive call at each level (over the same arguments).

The Fibonacci Function

fib(1) = 1 fib(2) = 1
fib(n + 2) = fib(n + 1) + fib(n)

or

fib(n) = fib(n - 1) + fib(n - 2) for n> 2

n	0	1	2	3	4	5	6	7	8	...
fib(n)	0	1	1	2	3	5	8	13	21	...

©1993 John Mylopoulos

8

Exponential Order Fib Function

```
(defun  fib-bad (n)
   (cond ((lessp  n 3) 1)
         (t (+ (fib-bad (- n 1) )
                  (fib-bad (- n 2))   )) ))
```

Here

Compl(`fib-bad`, n) =
Compl(`fib-bad`, n - 1) + Compl(`fib-bad`, n - 2) + 3
> 2*Compl(`fib-bad`, n - 2) + 3
> 2^2*Compl(`fib-bad`, n - 4) + 6
> ...
> $2^{n/2}$Compl(`fib-bad`, 1) = $2^{n/2}$

This grows at least as fast as an $O(2^n)$ function

The efficiency of a program is determined by the complexity of the implemented algorithm.

The language used only determines the constant multiplier.

©1993 John Mylopoulos

9

Linear Order fib in FORTRAN-like Lisp

```
(defun  fib (n)
   (prog  (i  fibi  prev aux)
      (setf   i     1)
      (setf   prev  0)
      (setf   fibi  1)
    loop
      (cond ((= n i)(return  fibi)) )
      (setf   i  (+  i  1))
      (setf   aux   prev)
      (setf   prev  fibi)
      (setf   fibi  (+  prev   aux))
      (go  loop)   ))
```

In general, this algorithm uses local variables as follows:

```
aux = fib(i - 2)
prev = fib(i - 1)
fibi = fib(i)
```

for `i > 1`

©1993 John Mylopoulos

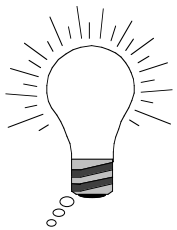
10

... can be equivalently written as

```
(defun  fib-good (n)
   (fibhelp n 1 1 0))

(defun fibhelp (n  i  fibi  prev)
   (cond ((= n i)     fibi)
         (t    (fibhelp   n
                  (+ i 1)
                  (+ fibi prev)
                  fibi )  )))
```

Compl(`fibhelp`, n) = Compl(`fibhelp`, n - 1) + 3



This makes Compl(`fib-good`) = O(n)

©1993 John Mylopoulos