CS320 Handout 10
From Clarity To Efficiency

Clarity and efficiency do not always go hand in hand. Easy-to-understand programs are often highly inefficient. Consider the Fibonacci function, for example. Its mathematical definition is:

$$\mathsf{fib}(x) = \begin{cases} x & \text{if } x < 2, \\ \mathsf{fib}(x-1) + \mathsf{fib}(x-2) & \text{if } x \geqslant 2. \end{cases}$$

1 Straightforward Fibonacci

This very simple definition is directly translated into the following Scheme code:

```
(define (fib x)
  (if (< x 2)
      x
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Let us try to estimate the run-time complexity of this little program. For our analysis, we assume that each of the primitive procedures requires one time-unit. This assumption approximates what happens in reality, but not by much, and is good enough for our purposes.

Given an arbitrary input value n, our task is to estimate the number of primitive operations carried out in the course of evaluating `(fib n)`. The above implementation of `fib` mentions 3 primitive ops: `<`, `+`, and `-`. Let $\#_{\mathsf{ops}}(n)$ denote the total count of primtive ops carried out by `(fib n)`. We then have:

$$\begin{aligned} &\#_{\mathsf{ops}}(0) = \#_{\mathsf{ops}}(1) = 1 && \text{“<” is used once} \\ &\#_{\mathsf{ops}}(n) = \#_{\mathsf{ops}}(n-1) + \#_{\mathsf{ops}}(n-2) + 4 && \text{“<” once, “+” once, “-” twice} \end{aligned}$$

Using the fact that $\#_{\mathsf{ops}}(n-1) \geqslant \#_{\mathsf{ops}}(n-2)$, we can write:

$$\begin{aligned} \#_{\mathsf{ops}}(n) &> 2 \cdot \#_{\mathsf{ops}}(n-2) \\ &> 2 \cdot (2 \cdot \#_{\mathsf{ops}}(n-4)) \\ &> 2 \cdot (2 \cdot (2 \cdot \#_{\mathsf{ops}}(n-6))) \\ &> 2^{n/2} \\ &= (\sqrt{2})^n \\ &> (1.4)^n \end{aligned}$$

The run-time complexity is therefore *exponential*. Except for very small values of n, this run-time is prohibitive. We can quickly confirm this theoretical result by trying to evaluate `fib` on a few values: `(fib 20)` is still computed within 2 or 3 seconds, but not `(fib 25)`, and the evaluation of `(fib 30)` has to be aborted after a minute or so without returning a value.

This is not surprising: Looking closely at the straightforward implementation of `fib` above, it does not attempt in any to avoid duplicating its work. The evaluation of `(fib` n`)` requires the evaluation of both `(fib` $(n-1)$`)` and `(fib` $(n-2)$`)` separately, ignoring the overlap between the two. This is confirmed by considering the trace of `fib` on a small value, say 4:

```
1 ]=> (trace-entry fib)

;Unspecified return value

1 ]=> (fib 4)

[Entering #[compound-procedure 1 fib]
    Args: 4]
[Entering #[compound-procedure 1 fib]
    Args: 2]
[Entering #[compound-procedure 1 fib]
    Args: 0]
[Entering #[compound-procedure 1 fib]
    Args: 1]
[Entering #[compound-procedure 1 fib]
    Args: 3]
[Entering #[compound-procedure 1 fib]
    Args: 1]
[Entering #[compound-procedure 1 fib]
    Args: 2]
[Entering #[compound-procedure 1 fib]
    Args: 0]
[Entering #[compound-procedure 1 fib]
    Args: 1]
;Value: 3
```

Exercise: Suppose we change the definition of `fib` to `fib1` as shown:

```
(define (fib1 x)
   (if (< x 2)
       0                                  ;; x is replaced by 0
       (+ (fib1 (- x 1)) (fib1 (- x 2)))))
```

The value returned by `(fib1` n`)` is always 0, no matter what n is. What is the run-time complexity of `fib1`? How does it compare with the run-time complexity of `fib`?

2 Faster Fibonacci

The reason for this inefficiency is not inherent to the Fibonacci. (Other programmable functions are inherently complex and no amount of clever programming will alleviate their complexity.) The reason for the inefficiency is that the standard definition of Fibonacci does not attempt to avoid recomputing intermediary values.

A smarter implementation of the Fibonacci function remembers the value of $\mathsf{fib}(n-1)$ so it can use it again in the evaluation of $\mathsf{fib}(n)$. Here is one way of achieving this:

```
(define (fast-fib n)
   (define (help a b n)              ;; ``a'' current fib, ``b'' previous fib
      (if (= n 0)
          b
          (help (+ a b) a (- n 1))))
   (help 1 0 n))
```

Let us estimate the run-time complexity of `fast-fib`:

$$\begin{aligned}\#_{\mathsf{ops}}(n) &= 3 + \#_{\mathsf{ops}}(n-1)\\ &= 3 + (3 + \#_{\mathsf{ops}}(n-2))\\ &= 3 + (3 + (3 + \#_{\mathsf{ops}}(n-3)))\\ &= \cdots\\ &= 3 \cdot n + 1\end{aligned}$$

This is *linear* growth, compared to exponential growth for the first implementation of `fib`.

Exercise: What is the value returned by `(help 10 0` n`)`? Compare it to the value of `(help 1 0` n`)` $= \mathsf{fib}(n)$. More generally, what is the value returned by `(help` k `0` n`)` where k is some positive integer?

The same idea of remembering the value of $\mathsf{fib}(n-1)$ so it can be used again in the evaluation of $\mathsf{fib}(n)$ can be implemented as follows:

```
(define (another-fast-fib n)
   (define (help x)
      (cond ((= x 0) (list 0 1))
            ((= x 1) (list 1 1))
            (else (let* ((y (help (- x 1)))
                         (a (first y))
                         (b (second y)))
                    (list b (+ a b))))))
   (first (help n)))
```