By Joshua Cantrell
jjc@cory.berkeley.edu

Algorithm Efficiency

Why Do Algorithms Have to be Efficient?

Algorithms must be efficient in two respects, time and space. People have a finite attention span and like answers fast (or reasonably fast), and sometimes speed is needed to complete a job in a required amount of time. There is also a finite and limited amount of storage space for algorithms to run under. An algorithm designed to solve a simple problem shouldn't require a couple of megabytes to do so.

Often people talk about writing in assembly language, or some other low-level languge, attempting to *optimize* the performance of a program. Sometimes this is done while overlooking the possibility of a new choice of algorithm which may be faster and more efficient.[1] A better choice of an algorithm can have such a profound impact on a program that a program written in an interpreter like Scheme could outrun a program written in a compiled language like assembly, even though they may be performing the same operation, only doing it in different ways. This is why understanding the relative speed of algorithms is crucial in writing efficient programs.

Orders of Growth

To help us understand the relative inefficiency of algorithms, we use big θ notation[2]. This function represents the upper bounds of an algorithm's efficiency. In other words, the algorithm, in the worse case situation, is no better than $\theta(f(n))$, where R(n) is the worst case function and for chosen constants K_1 and K_2, $K_1 f(n) \le R(n) \le K_2 f(n)$. This shows that the function, $f(n)$, only has to grow as fast as R(n) and additional constants as well as constant scaling is unimportant. θ is used for both space and time. As an example, we can examine the *factorial* procedure[3]:

```
(define (factorial n) (if (= n 0) 1 (+ 1 (factorial (- n 1)))))
```

```
1.  (factorial 5)
2.  (* 5 (factorial 4))
3.  (* 5 (* 4 (factorial 3)))
4.  (* 5 (* 4 (* 3 (factorial 2))))
5.  (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
6.  (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))
7.  (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
8.  (* 5 (* 4 (* 3 (* 2 1))))
9.  (* 5 (* 4 (* 3 2)))
10. (* 5 (* 4 6))
11. (* 5 24)
12. 120
```

[1] Often this is done because of a lack of understanding about the differences between algorithms.
[2] It's θ instead of *O*, because $O(f(n))$ is defined to be $R(n) \le Kf(n)$. $f(n)$ can grow faster than $R(n)$! θ sandwiches R(n), so $f(n)$ has to grow at the same rate as $R(n)$. Another function called Ω is defined to be $R(n) \ge Kf(n)$, where $f(n)$ can grow slower than $R(n)$. θ is basically the function where both $R(n) = O(f(n))$ and $R(n) = \Omega(f(n))$.
[3] Example borrowed and modified from SICP.

By Joshua Cantrell
jjc@cory.berkeley.edu

The **bold** print operators represent delayed operations that are waiting for the *italicized* function to be called and return a result. The underlined values are those which need to be stored in memory while being delayed.

Because the time of the procedure is directly related to the number input into the *factorial* procedure, we choose our Order of Growth function *f* to be a function of it. Notice how we asked for the factorial of 5, and it consisted of 6 calls of the *factorial* procedure, and 6 calls of the * procedure. If we say that calling the *factorial* procedure takes k_1 time and calling the * procedure takes k_2 time, then we can see that this procedure took about $(k_1 + k_2) * 6$ time, or equivalently $R(n) = (k_1 + k_2) * (n + 1)$ time. Our algorithm takes $\theta(n)$ time.

Similarly, the procedure call resulted in 2 values being stored, the * procedure and corresponding number for each delay. With 5 delays, and letting s_1 be the space taken up by the * procedure and s_2 be the space taken up by the number, about $(s_1 + s_1) * 5$ space, or equivalently $R(n) = (s_1 + s_1) * n$ space. Our algorithm takes $\theta(n)$ space.

Recursive Vs. Iterative Processes

So far we've defined many recursive procedures, most of which describe recursive processes. *Recursive processes* are procedures that call themselves and must delay an operation, leading to an increase in stored data. The processes examined in the previous section is a recursive process. *Iterative processes* are procedures that call themselves but don't require any operations to be delayed. Those types of procedures are known to be *tail recursive*.

Recursive procedures that take $\theta(n)$ time and space are collectively known as *linear recursive* processes. The *factorial* procedure given above is a linear recursive process. Now let's compare this to a *linear 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)))
```

1. (*factorial* 5)
2. (*fact-iter* 1 1 5)
3. (*fact-iter* 1 2 5)
4. (*fact-iter* 2 3 5)
5. (*fact-iter* 6 4 5)
6. (*fact-iter* 24 5 5)
7. (*fact-iter* 120 6 5)
8. 120

In this procedure, we call *fact-iter* 6 times and * 6 times. Assuming similar times to the other procedure, this procedure also takes $R(n) = (k_1 + k_2) * (n + 1)$ time, or $\theta(n)$ time. The difference comes in the space requirement. The linear recursive process kept on storing new values; this one only needs to store 3 at all times, whose sizes are s_1, s_2, and s_3 respectively. This makes the process only take up $R(n) = s_1 + s_2 + s_3$ space, or $\theta(1)$ space.

If you had your choice between the first algorithm and the second, which would you choose? Overall, the second algorithm is the more efficient of the two in both time and space, so it would be preferred.

By Joshua Cantrell
jjc@cory.berkeley.edu

Making Iterative Processes

One way to make iterative processes for mathematical functions is called "the technique of defining an *invariant quantity* that remains unchanged from state to state."[4] This is where we have an answer we're trying to find and some equation that can be manipulated to produce an answer without changing its total value. For example, for a factorial we can write:

$$\textit{answer} = a * n!$$

We wish to keep the value of *answer* the same throughout this entire process. So, if we decrease *n* by 1,

$$\textit{answer} = (a * n) * (n - 1)!$$

we must redefine *a* to be the value of (a * n), and *n* to be the value of (n - 1).

$$a' = a * n$$
$$n' = n - 1$$
$$\textit{answer} = a' * n'!$$

Now all we have to do is determine what our base case will be, and write the function that solves the problem using this process. The base case would be when *n* = 1, since that would require *a* to be equal to the correct answer. The corresponding Scheme procedure would be:

```
(define (factorial n)
  (define (fact-iter a n)
    (if (<= n 1) a
        (fact-iter (* a n) (- n 1))))
  (fact-iter 1 n))
```

[4] 1st Edition SICP, pg. 43