MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

D. Boning, Recitation Notes, Feb. 10

A Primer on Orders of Growth

Goal: For a problem of size n, we want to characterize how much of some resource is needed to solve the problem. Typically, we care about time $T(n)$ and space $S(n)$ requirements. Approaches to describing resource usage include:

- An *absolute* measure. Exact count of resource use, e.g.,

$$T(n) = 5n^2 + 3n \log_3 n + 5$$

- An *asymptotic* measure.

  (a) Approximate the behavior for large n. For large n, we see that

$$5n^2 \gg 3n \log_3 n \gg 5$$

  so we can approximate the time requirements for large n as

$$T(n) \simeq 5n^2.$$

  (b) Ignore the constants of proportionality:

$$T(n) \quad \simeq \quad k\ n^2$$

  We can use "theta" notation to understand the *order* of the growth. If there exist constants k_1 and k_2, independent of n, such that $k_1 f(n) \leq R(n) \leq k_2 f(n)$ for large n, then $R(n) = \Theta(f(n))$. So for our T above we have:

$$T(n) \quad = \quad \Theta(n^2)$$

- A *bounded* measure. We may only know a maximum bound on resource use. If $R(n) \leq kf(n)$ for some k and for large n, then we can say $R(n) = O(f(n))$. We see that $O(\cdot)$ is a weaker statement than $\Theta(\cdot)$:

$$T(n) = 5n^2 \quad \text{so that} \quad T(n) = O(n^3) \quad \text{is true.}$$

  Thus $O(\cdot)$ gives some (perhaps overstated) upper bound, while $\Theta(\cdot)$ gives us the asymptotic behavior of the process or algorithm.

Choosing an Algorithm

When choosing an algorithm to solve a particular problem, the actual problem size n and constant k may indeed matter. Suppose we have two algorithms with time behavior T_1 and T_2, respectively:

$$\begin{aligned} T_1(n) &= 2 \times 10^6 n \\ T_2(n) &= 2n^2 \end{aligned}$$

For what n would you choose algorithm 2 over algorithm 1?

Some Typical Orders of Growth

Typical orders of growth, and the kinds of problems that often give rise to these growths:

- $\Theta(1)$
  Constant growth. In time, this corresponds to simple computations (e.g. `(square x)`). Iterative algorithms use constant space.
- $\Theta(\log n)$
  Logarithmic growth. Logarithmic time algorithms often arise when a big problem can be transformed into a smaller problem via reduction by some constant fraction.
- $\Theta(n)$
  Linear growth. Linear time algorithms arise, for example, when some given amount of processing is needed for each of n input elements.
- $\Theta(n \log n)$
  "n log n" growth. Commonly occurs when a big problem is broken into smaller subproblems, solving them independently, then combining the solutions.
- $\Theta(n^2)$
  Quadratic growth. Quadratic time algorithms often occur when all pairs of data items must be processed. Such algorithms are only practical for relatively small problems.
- $\Theta(n^3)$
  Cubic growth. Commonly occurs when all triplets of data items must be processed. Only practical for relatively small problems.
- $\Theta(2^n)$
  Exponential growth. Commonly arises in "brute force" solutions to problems (e.g. when all other elements must be considered as each individual element is processed). Impractical for all but the smallest problems.

Linear Iterative vs. Linear Recursive Processes

A process that requires constant space is *iterative*; a *recursive* process requires nonconstant space. A *linear recursive* process is linear in time and space: $T(n) = \Theta(n)$ and $S(n) = \Theta(n)$. A *linear iterative* process runs in constant space and is linear in time: $T(n) = \Theta(n)$ and $S(n) = \Theta(1)$.

Recursion, Iteration, and Orders of Growth

1. For the following functions R find the simplest function f for which $R(n) = \Theta(f(n))$, and write $\Theta(f(n))$:

   a. $R(n) = 6$

   b. $R(n) = n^2 + 6$

   c. $R(n) = \log n + 6n^3 + 3n^2 + 7n + 6$

   d. $R(n) = 2^{3n+7}$

2. Use the substitution model to analyze the `count1` and `count2` procedures applied to 3:

```
(define our-write-line
  (lambda (x)
    (write-line x)
    x))[1]

(define count1
  (lambda (x)
    (cond ((= x 0) 0)
          (else (our-write-line x)
                (count1 (one-less-than x))))))
```

```
(define count2
  (lambda (x)
    (cond ((= x 0) 0)
          (else (count2 (one-less-than x))
                (our-write-line x)))))
```

[1] Normally, we would simply use SCHEME's built-in `write-line` here. In the version of SCHEME running in the 6.001 lab, `write-line` returns a special value called `the-undefined-value` since its only intention is to cause some side effect to occur. As we want a version of `write-line` that returns its argument, we have simply defined our own.

3. Determine the orders of growth in number of operations and space for `mystery`:

```
(define mystery
  (lambda (n)
    (if (= n 0)
        0
        (+ n (mystery (- n 1))))))
```

space	number of operations

What function does mystery compute? _____

Does it generate an **iterative** or a **recursive** process?

Write `mystery2`, which computes the same function as `mystery`, but generates the other kind of process.

space	number of operations

4. Use the substitution model to determine the orders of growth in number of operations and space for `mul1`:

```
(define mul1
  (lambda (n m)
    (if (= n 0)
        0
        (+ m (mull (- n 1) m)))))
```

space	number of operations

5. Use the substitution model to determine the orders of growth in number of operations and space for `mul2`:

```
(define (mul2 n m)
  (define (iter count ans)
    (if (= count 0)
        ans
        (iter (one-less-than count)
              (+ m ans))))
  (iter n 0))
```

space	number of operations

6. Assuming primitive `halve`, `double`, and `even?` procedures, fill in the rest of this procedure, and analyze its order of growth.

```
(define (mul3 n m)
  (cond ((= n 0)             )
        ((even? n)


        (else    ; n is odd here!
```

space	number of operations

7. Next, write a procedure that takes a factor and returns a procedure that, when applied to one argument, will multiply the factor by the argument (e.g. `((mult 5) 6)`). Again, write this *without* using syntactic sugaring.

Now write it in the syntactic sugared form.

Notice how the sugared form makes clear that `mult` returns a procedure. This is one advantage of the syntactic sugared form, but you should be able to recognize the same fact in the unsugared form.

8. Write a procedure `expt` (using the syntactic sugared form of `define`) that takes two arguments, `b` and `n`, and computes the result of raising `b` to the n^{th} power through successive multiplication. Give both recursive and iterative versions. Fill in the order-of-growth statistics, too.

	space	number of operations
iterative version		
recursive version		