CS 61A Lecture Notes First Half of Week 2

Topic: Recursion and iteration

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (pages 31–72)

The next two lectures are about efficiency. Mostly in 61A we don't care about that; it becomes a focus of attention in 61B. In 61A we're happy if you can get a program working at all, except for the next 2 lectures, when we introduce ideas that will be more important to you later.

We want to know about the efficiency of algorithms, not of computer hardware. So instead of measuring runtime in microseconds or whatever, we ask about the number of times some primitive (fixed-time) operation is performed. Example:

```
;;;;;                          In file cs61a/lectures/1.2/growth.scm
(define (square x) (* x x))

(define (squares sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (squares (bf sent)) )))
```

To estimate the efficiency of this algorithm, we can ask, "if the sentence has N numbers in it, how many multiplications do we perform?" The answer is that we do one multiplication for each number in the argument, so we do N altogether. The amount of time needed should roughly double if the number of numbers doubles.

Another example:

```
;;;;;                          In file cs61a/lectures/1.2/growth.scm
(define (sort sent)
  (if (empty? sent)
      '()
      (insert (first sent)
              (sort (bf sent)) )))

(define (insert num sent)
  (cond ((empty? sent) (se num sent))
        ((< num (first sent)) (se num sent))
        (else (se (first sent) (insert num (bf sent)))) ))
```

Here we are sorting a bunch of numbers by comparing them against each other. If there are N numbers, how many comparisons do we do?

Well, if there are K numbers in the argument to `insert`, how many comparisons does it do? K of them. How many times do we call `insert`? N times. But it's a little tricky because each call to `insert` has a different length sentence. The range is from 0 to $N-1$. So the total number of comparisons is actually

$$0 + 1 + 2 + \cdots + (N-2) + (N-1)$$

which turns out to be $\frac{1}{2}N(N-1)$. For large N, this is roughly equal to $\frac{1}{2}N^2$. If the number of numbers doubles, the time required should quadruple.

That constant factor of $\frac{1}{2}$ isn't really very important, since we don't really know what we're halving—that is, we don't know exactly how long it takes to do one comparison. If we want a very precise measure of how many microseconds something will take, then we have to worry about the constant factors, but for an

overall sense of the nature of the algorithm, what counts is the N^2 part. If we double the size of the input to a program, how does that affect the running time?

We use "Big O" notation to express this sort of approximation. We say that the running time of the `sort` function is $O(N^2)$ while the running time of the `squares` function is $O(N)$. The formal definition is

$$f(x) = O(g(x)) \Leftrightarrow \exists k, N \mid \forall x > N, |f(x)| \leq k \cdot |g(x)|$$

What does all this mean? Basically that one function is always less than another function (e.g., the time for your program to run is less than x^2) except that we don't care about constant factors (that's what the k means) and we don't care about small values of x (that's what the N means).

Why don't we care about small values of x? Because for small inputs, your program will be fast enough anyway. Let's say one program is 1000 times faster than another, but one takes a millisecond and the other takes a second. Big deal.

Why don't we care about constant factors? Because for large inputs, the constant factor will be drowned out by the order of growth—the exponent in the $O(x^i)$ notation. Here is an example taken from the book *Programming Pearls* by Jon Bentley (Addison-Wesley, 1986). He ran two different programs to solve the same problem. One was a fine-tuned program running on a Cray supercomputer, but using an $O(N^3)$ algorithm. The other algorithm was run on a Radio Shack microcomputer, so its constant factor was several million times bigger, but the algorithm was $O(N)$. For small N the Cray was much faster, but for small N both computers solved the problem in less than a minute. When N was large enough for the problem to take a few minutes or longer, the Radio Shack computer's algorithm was faster.

```
;;;;;                          In file cs61a/lectures/1.2/bentley

               t1(N) = 3.0 N^3          t2(N) = 19,500,000 N

       N       CRAY-1 Fortran           TRS-80 Basic

      10       3.0 microsec             200 millisec
     100       3.0 millisec             2.0 sec
    1000       3.0 sec                  20 sec
   10000       49 min                   3.2 min
  100000       35 days                  32 min
 1000000       95 yrs                   5.4 hrs
```

Typically, the algorithms you run across can be grouped into four categories according to their order of growth in time required. The first category is *searching* for a particular value out of a collection of values, e.g., finding someone's telephone number. The most obvious algorithm (just look through all the values until you find the one you want) is $O(N)$ time, but there are smarter algorithms that can work in $O(\log N)$ time or even in $O(1)$ (that is, constant) time. The second category is *sorting* a bunch of values into some standard order. (Many other problems that are not explicitly about sorting turn out to require similar approaches.) The obvious sorting algorithms are $O(N^2)$ and the clever ones are $O(N \log N)$. A third category includes relatively obscure problems such as matrix multiplication, requiring $O(N^3)$ time. Then there is an enormous jump to the really hard problems that require $O(2^N)$ or even $O(N!)$ time; these problems are effectively not solvable for values of N greater than one or two dozen. (Inventing faster computers won't help; if the speed of your computer doubles, that just adds 1 to the largest problem size you can handle!) Trying to find faster algorithms for these *intractable* problems is a current hot research topic in computer science.

• Iterative processes

So far we've been talking about time efficiency, but there is also memory (space) efficiency. This has gotten less important as memory has gotten cheaper, but it's still somewhat relevant because using a lot of memory increases swapping (not everything fits at once) and so indirectly takes time.

The immediate issue for today is the difference between a *linear recursive process* and an *iterative process.*

```
;;;;;                          In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (if (empty? sent)
      0
      (+ 1 (count (bf sent))) ))
```

This function counts the number of words in a sentence. It takes $O(N)$ time. It also requires $O(N)$ space, not counting the space for the sentence itself, because Scheme has to keep track of N pending computations during the processing:

```
(count '(i want to hold your hand))
(+ 1 (count '(want to hold your hand)))
(+ 1 (+ 1 (count '(to hold your hand))))
(+ 1 (+ 1 (+ 1 (count '(hold your hand)))))
(+ 1 (+ 1 (+ 1 (+ 1 (count '(your hand))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '(hand)))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '())))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 1)))))
(+ 1 (+ 1 (+ 1 (+ 1 2))))
(+ 1 (+ 1 (+ 1 3)))
(+ 1 (+ 1 4))
(+ 1 5)
6
```

When we get halfway through this chart and compute `(count '())`, we aren't finished with the entire problem. We have to remember to add 1 to the result six times. Each of those remembered tasks requires some space in memory until it's finished.

Here is a more complicated program that does the same thing differently:

```
;;;;;                          In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (define (iter wds result)
    (if (empty? wds)
        result
        (iter (bf wds) (+ result 1)) ))
  (iter sent 0) )
```

This time, we don't have to remember uncompleted tasks; when we reach the base case of the recursion, we have the answer to the entire problem:

```
(count '(i want to hold your hand))
(iter '(i want to hold your hand) 0)
(iter '(want to hold your hand) 1)
(iter '(to hold your hand) 2)
(iter '(hold your hand) 3)
(iter '(your hand) 4)
(iter '(hand) 5)
(iter '() 6)
6
```

When a process has this structure, Scheme does not need extra memory to remember all the unfinished tasks during the computation.

This is really not a big deal. For the purposes of this course, you should generally use the simpler linear-recursive structure and not try for the more complicated iterative structure; the efficiency savings is not worth the increased complexity. The reason Abelson and Sussman make a fuss about it is that in other programming languages any program that is recursive in *form* (i.e., in which a function invokes itself) will take (at least) linear space even if it could theoretically be done iteratively. These other languages have special iterative syntax (`for`, `while`, and so on) to avoid recursion. In Scheme you can use the function-calling mechanism and still achieve an iterative process.

• More is less: non-obvious efficiency improvements.

The nth row of Pascal's triangle contains the constant coefficients of the terms of $(a + b)^n$. Each number in Pascal's triangle is the sum of the two numbers above it. So we can write a function to compute these numbers:

```
;;;;;                          In file cs61a/lectures/1.2/pascal.scm
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

This program is very simple, but it takes $O(2^n)$ time! [Try some examples. Row 18 is already getting slow.]

Instead we can write a more complicated program that, on the surface, does a lot more work because it computes an *entire row* at a time instead of just the number we need:

```
;;;;;                          In file cs61a/lectures/1.2/pascal.scm
(define (new-pascal row col)
  (nth col (pascal-row row)) )

(define (pascal-row row-num)
  (define (iter in out)
    (if (empty? (bf in))
        out
        (iter (bf in) (se (+ (first in) (first (bf in))) out)) ))
  (define (next-row old-row num)
    (if (= num 0)
        old-row
        (next-row (se 1 (iter old-row '(1))) (- num 1)) ))
  (next-row '(1) row-num) )
```

This was harder to write, and seems to work harder, but it's incredibly faster because it's $O(N^2)$.

The reason is that the original version computed lots of entries repeatedly. The new version computes a few unnecessary ones, but it only computes each entry once.

Moral: When it really matters, think hard about your algorithm instead of trying to fine-tune a few microseconds off the obvious algorithm.