CISC280 Big O examples

Big O notation is useful for approximating the amount of work the computer has to do, or the amount of space a process will take. We will use it for both. Please re-read the section of $O()$ in your text before you read this sheet.

Test Answers

Note: many of you had the right ideas about these questions, but did not express yourselves well. Stack space is *not* the same as memory; an iterative process is *not* a loop. **Learn the vocabulary.** It is necessary for writing and speaking with other computers scientists, and for the next two exams.

> What is big O for time of linear recursive copy? What about for tail-recursive copy? Explain.

Both linear and tail recursive copy call themselves n times, since n decreases by one each call until it reaches the base case. So they are both $O(n)$ for time.

> What is big O for space of linear recursive copy? What about for tail-recursive copy? Explain.

Linear recursive functions have deferred operations that require using additional stack space for each call, so space is $O(n)$. Tail recursive functions have no deferred ops, so a recursive procedure can develop an iterative process that uses only one space on the stack, so space is $O(1)$.

Set Union

`memq` is a scheme primitive. I use it here because it uses the `eq?` predicate, which is $O(1)$, whereas the `member` function uses `equal?`, which has complexity that varies with the input[1]. The definition of memq looks something like this:

```
(define (memq elt set)
  (cond ((null? set) #f)
        ((eq? elt (car set)) set)
        (else (memq elt (cdr set)))))
```

[1] Recall that equal? works for comparing structures, while eq? compares pointers.

The time complexity of memq is $O(n)$, where n is the size of set. This is because in the worst case, where elt is not present, memq will be called on successive cdrs of set, and we can only take the cdr of set n times before reaching the base case.

```
(define (adjoin elt set)
  (if (memq elt set)
      set
      (cons elt set)))
```

Adjoin, for an unordered list representation of a set, is not recursive. However, it calls memq, which is $O(n)$ time in the size of set, so adjoin is also $O(n)$ time. Another way of saying $O(n)$ time is to say the time is **linear** with respect to the size of the set.

```
(define (union1 a b)
  (cond ((null? a) b)
        ((memq (car a) b) (union1 (cdr a) b))
        (else (cons (car a)(union1 (cdr a) b)))))
```

Consider union1 above. Assume the size of set a is n, and the size of b is m. Union1 is recursive, and the number of times it is called is controlled by the size of a, since we take the cdr of a each time until we reach the base case of an empty a set. Thus union1 is called n times.

However, union1 also calls memq, which has time complexity equal to the size of the set is is called on, in this case set b, or size m. Note that b never changes size - it does not grow or shrink in this definition, so each call to memq is $O(m)$. Since memq is called each time through union1, and union1 is called n times, then memq is called m times also. Thus the time complexity is $O(nm)$.

If we look at the total size of both inputs as the size of the problem, then we can see that in the worst case the two sets will be the same size. So if we call the combined size n, then the time complexity will be

$$n/2 * n/2 = n^2/4 \in O(n^2)$$

This is also called **quadratic** time, since a polynomial of degree two (like $5n^2+2n+3$) is said to be quadratic. Any **polynomial** expression, such as n^3 where the variable we are interested in is raised to a power, is very different from what we call **exponential** expressions, such as 2^n. Don't confuse the terms polynomial and exponential. Polynomial complexity is often fine for computation, but exponential complexity rarely is.

```
(define (union2 a b)
  (cond ((null? a) b)
        (else
         (adjoin (car a) (union2 (cdr a) b)))))
```

Now consider union2 which we wrote in class (we wrote it with accumulate, but this is the expanded version which is easier to see).

Union2 differs from union1 above in that it calls adjoin on each element of set a and a set b that is growing over time. The number of calls to union2 is still size of a, or n as described above.

Adjoin is being called on a set which is size m on the first call, and size $(n+m)$ on the last call (in the worst case; can you think of another case?).

One simple way to characterize the complexity of union2 is to say that n calls to union2 each call adjoin with complexity $O(n + m)$, resulting in a complexity of

$$O(n * (n + m)) = O(n^2 + nm)$$

Again, if we use a different meaning for n, that the combined size of both sets is n, this would be

$$n^2 + n^2 = 2n^2 \in O(n^2)$$

A more precise expression[2] of what adjoin is doing in union2 is to see that the n calls to union2 each call adjoin on a different sized set:

$$\sum_{i=1}^{n} m + i = \sum_{i=1}^{n} m + \sum_{i=1}^{n} i$$

$$= nm + \frac{n(n-1)}{2} = nm + n^2/2 - n/2$$

$$\in O(n^2 + nm)$$

So while the expression is more precise, it results in the same complexity when constants are ignored.

[2] which I do not expect you to use on a test...