COSC 220 - Computer Science II
Dr. Joseph Anderson
Recursion and Recurrence Notes

1 Recurrence Relations

In general, a recurrence relation is a sequence (e.g. $a_1, a_2, a_3, \dots$) whose terms are *recursively* defined. One of the most well-known recursively-defined sequences is the *Fibonacci Sequence*, with the definition for $n \geq 1$:

$$F(n) = F(n-1) + F(n-2); F(1) = 1; F(2) = 1;$$

Note that this sequence is defined in three parts: the leftmost equation is the *recursive* component, where the nth term is defined by the previous two terms. The second two equations($F(1) = 1$ and $F(2) = 1$) define what are called the *base cases*, which are *not* recursive. By direct calculation, it is not hard to see the first few terms of the sequence:

n	$F(n)$
1	1
2	1
3	2
4	3
5	5
6	8
7	13

Recurrences, for the purpose of this course, will be commonly used to describe the *running time* of certain programs or algorithms. Of course here, by running time, we are counting *number of operations*, where for now we generally assume that an operation is simply a assignment, comparison, or basic arithmetic statement. For example, the code `x = 2 * y + 7` could be counted as at least three operations, one for multiplication, one for addition, and one to copy the value into the memory location of the variable `x`. However, as discussed in earlier lectures, the precise number of operations will be swept under the rug by the use of O-notation.

2 Reivew: O-notation for running time

To see the use of O-notation used to describe running time in an example, consider the following function:

```
function f(n):
  s := 0
  for i = 1 to n
    s := s + 7 * i
  end
  return s
end
```

If we were to try and count, precisely, the number of operations it takes to run this program, given the number n as input, we might come up with:

- $1-$ for the initial assignment into s
- $3-$ to compute $s + 7 * i$ and store it into the variable s
- $2-$ to increment and re-assign the value of i
- $1-$ to copy the return value into the stack for the function f

However, we need to take into account that the middle two steps will run n times (for $i = 1$ up to n) so in aggregate the total number of operations for this function would be something like $5n + 2$. But when we consider the asymptotic behavior of this quantity, it is just $O(n)$, i.e. *linear* in n. If we had counted differently, say, viewing the comparison of i with n as an extra "step" in the program (intentionally neglected above, but which happens nonetheless), we would have arrived at $6n + 2$, *which is still* $O(n)$. Here we can see the power granted by the imprecision of O-notation: we don't need to worry about *exactly* how many operations happen, as long as we get the *growth with respect to* n. To repeat the above analysis in these terms, only a cursory glance at this function will yield that there are some constant operations, which occur outside the loop, and some constant time operation that occurs on each iteration of the loop, but also that the loop runs $O(n)$ times. Since, for each of these runs, there is only constant work inside, the loop itself only takes $O(n)$ time, and the constant work outside of it doesn't change the asymptotics. The situation, as we've seen already, can become slightly more delicate when the evolution of the loop variable (in this case, `i`) becomes more complicated. For more on these situations, refer to the in-class notes.

3 Analysis of Recursive Algorithms

Consider the following example, obtained from simple modification of the function f above:

```
function f(n):
```

```
    if n < 10
      return 5

    s := 0
    for i = 1 to n
    s := s + 7 * i
    end
    return s + f(n-1)
end
```

This function does not necessarily compute anything interesting, but has some different behavior from the one above. The two extra important features are:

1. If the argument is at most 10, it returns without doing any extra work, i.e. has a constant runtime if $n \leq 10$.

2. If the argument was more than 10, it will execute the same loop as before, but perform an extra addition in the return statement, where the right summand requires calling f again!

How do we now analyze the runtime of this function? If we consider the more interesting situation where $n > 10$, we get the same $O(n)$ operations *before* taking into account the return statement. But then, on the last line, the function must wait for the program to **call f again, with $n-1$ as the argument!** To put this mathematically, the total runtime of the algorithm is the time of the loop, *plus* however many operations it takes to run $f(n-1)$. If we denote by $T(n)$ the time it takes f to run with argument n, this yields a recurrence relation:

$$T(n) = \begin{cases} T(n-1) + C_1 n & n > 10 \\ C_2 & n \leq 10 \end{cases} \tag{1}$$

where C_1 is some constant hidden by the $O(n)$ quantification of the operations outside the recursive call and similarly, C_2 is the number of operations it takes to compare n to 10 and return the value 5. Here is where it is important to take into account the two cases: if n is large enough, we have to recurse, doing more work. If, however, n is small enough to trigger the *base case*, only a constant amount of work is done by the algorithm.

The goal, now, is to find what $T(n)$ is, *asymptotically, in terms of n*. To do so, we can apply the definition in (1) in the same way we studied the Fibonacci sequence above, but this time we will work backwards from n (assuming it starts much larger than 10). See Table 1 for some of the values of $T(n)$ computed in this way.

So if we want to compute $T(n)$ without any recurrence, we need to *unroll*

the recurrence to compute

$$\begin{aligned} T(n) &= T(n-1) + C_1 n \\ &= T(n-2) + C_2(n-1) + C_1 n \\ &= T(n-3) + C_2(n-2) + C_2(n-1) + C_1 n \\ &= T(n-4) + C_2(n-3) + C_2(n-2) + C_2(n-1) + C_1 n \end{aligned}$$

which we can generalized for any $k \in \{1, 2, \ldots, n-10\}$ as:

$$T(n) = T(n-k) + C_1 \sum_{i=0}^{k-1} n - i.$$

Verify for yourself that this generalizes the three cases above where $k = 1, 2, 3$. Then, notice the upper bound for k is $n - 10$ because that is the case where $n - k = 10$ which is the case where we may substitute $T(n-k) = T(10) = C_2$. Considering, then, the case $k = n - 10$, we have

$$\begin{aligned} T(n) &= T(n-(n-10)) + C_1 \sum_{i=1}^{n-10-1} n - i \\ &= C_2 + C_1 \left(\sum_{i=0}^{n-11} n - \sum_{i=0}^{n-11} i \right) \\ &= C_2 + C_1 \left(n(n-10) - \frac{(n-10)(n-11)}{2} \right) \\ &= O(n^2) \text{ after simplification.} \end{aligned}$$

Thus we can see for this example $T(n) = O(n^2)$, meaning the runtime of this function grows quadratically with n. To match this with some intuition, we can note that the loop inside the function is linear in n, but because the recursive call only decreases n by 1 each time, we have a linear number of recursive calls, which generally means that the total cost is quadratic (linear times linear).

n	$T(n)$
n	$T(n-1) + C_1 n$
$n-1$	$T(n-2) + C_1(n-1)$
$n-2$	$T(n-3) + C_1(n-2)$
$\vdots$	$\vdots$
10	C_2
9	C_2
$\vdots$	$\vdots$
1	C_2

Table 1