14 Recurrences

14.1 Introduction

It is relatively easy to set up equations, typically using summations, for counting the basic operations performed in a non-recursive algorithm. But this won't work for recursive algorithms in which much computation is done in recursive calls rather than in loops. How are basic operations to be counted in recursive algorithms?

A different mathematical techniques must be used; specifically, a recurrence relation must be set up to reflect the recursive structure of the algorithm.

> **Recurrence relation**: An equation that expresses the value of a function in terms of its value at another point.

For example, consider the recurrence relation $F(n) = n \cdot F(n\text{-}1)$, where the domain of F is the non-negative integers (all our recurrence relations will have domains that are either the non-negative integers or the positive integers). This recurrence relation says that the value of F is its value at another point times n. Ultimately, our goal will be to solve recurrence relations like this one by removing recursion but, as it stands, it has infinitely many solutions. To pin down the solution to a unique function, we need to indicate the value of the function at some particular point or points. Such specifications are called **initial conditions**. For example, suppose the initial condition for function F is $F(0) = 1$. Then we have the following values of the function.

$$
\begin{aligned}
&F(0) = 1 \\
&F(1) = 1 \cdot F(0) = 1 \\
&F(2) = 2 \cdot F(1) = 2 \\
&F(3) = 3 \cdot F(2) = 6 \\
&\ldots
\end{aligned}
$$

We thus recognize F as the factorial function. A recurrence relation plus one or more initial conditions form a **recurrence**.

> **Recurrence**: a recurrence relation plus one or more initial conditions that together recursively define a function.

14.2 Setting Up Recurrences

Lets consider a few recursive algorithms to illustrate how to use recurrences to analyze them. The Ruby code in Figure 1 implements an algorithm to reverse a string.

```
def reverse(s)
  return s if s.size <= 1
  return reverse(s[1..-1]) + s[0]
end
```

Figure 1: Recursively Reversing a String

If the string parameter `s` is a single character or the empty string, then it is its own reversal and it is returned. Otherwise, the first character of `s` is concatenated to the end of the result of reversing `s` with its first character removed.

The size of the input to this algorithm is the length n of the string parameter. We will count string concatenation operations. This algorithm always does the same thing no matter the contents of the string, so we need only derive its every-case complexity $C(n)$. If n is 0 or 1, that is, if the string parameter s of `reverse()` is empty or only a single character, then no concatenations are done, so $C(0) = C(1) = 0$. If $n > 1$, then the number of concatenations is one plus however many are done during the recursive call on the substring, which has length n-1, giving us the recurrence relation

$$C(n) = 1 + C(n\text{-}1)$$

Putting these facts together, we have the following recurrence for this algorithm.

$$\begin{aligned} C(n) &= 0 && \text{for } n = 0 \text{ or } n = 1 \\ C(n) &= 1 + C(n\text{-}1) && \text{for } n > 1 \end{aligned}$$

Lets consider a slightly more complex example. The Towers of Hanoi puzzle is a famous game in which one must transfer a pyramidal tower of disks from one peg to another using a third as auxiliary, under the constraint that no disk can be placed on a smaller disk. The algorithm in Figure 2 solves this puzzle in the least number of steps.

```
def move_tower(src, dst, aux, n)
  if n == 1
    move_disk(src,dst)
  else
    move_tower(src, aux, dst, n-1)
    move_disk(src, dst)
    move_tower(aux, dst, src, n-1)
  end
end
```

Figure 2: Towers of Hanoi Algorithm

The last parameter of the `move_tower()` function is the number of disks to move from the `src` peg to the `dst` peg. To solve the problem, execute

```
move_tower(src, dst, aux, src.size)
```

Our measure of the size of the input is the number of disks on the source peg. The algorithm always does the same thing for a given value of n, so we compute the every-case complexity $C(n)$. When $n = 1$, only one disk is moved, so $C(1) = 1$. When n is greater than one, then the number of disks moved is the number moved to shift the top n-1 disks to the auxiliary peg, plus one move to put the bottom disk on the destination peg, plus the number moved to shift n-1 disks from the auxiliary peg to the destination peg. This gives the following recurrence.

$$\begin{aligned} C(1) &= 1 && \text{for } n = 1 \\ C(n) &= 1 + 2 \cdot C(n\text{-}1) && \text{for } n > 1 \end{aligned}$$

Although recurrences are nice, they don't tell us in a closed form the complexity of our algorithms—in other words, the only way to calculate the value of a recurrence for n is to start with the initial conditions and work our way up to the value for n using the recurrence relation, which can be a lot of work. We would like to come up with solutions to recurrences that don't use recursion so that we can compute them easily.

14.3 Solving Recurrences

There are several ways to solve recurrences but we will consider only one called the **method of backward substitution**. This method has the following steps.

1. Expand the recurrence relation by substituting it into itself several times until a pattern emerges.
2. Characterize the pattern by expressing the recurrence relation in terms of n and an arbitrary term i.
3. Substitute for i an expression that will remove the recursion from the equation.
4. Manipulate the result to achieve a final closed form for the defined function.

To illustrate this technique, we will solve the recurrences above, starting with the one for the string reversal algorithm. Steps one and two for this recurrence appear below.

$$\begin{aligned} C(n) &= 1 + C(n\text{-}1) \\ &= 1 + (1 + C(n\text{-}2)) = 2 + C(n\text{-}2) \\ &= 2 + (1 + C(n\text{-}3)) = 3 + C(n\text{-}3) \\ &= \ldots \\ &= i + C(n\text{-}i) \end{aligned}$$

The last expression characterizes the recurrence relation for an arbitrary term i. $C(n\text{-}i)$ is equal to an

$$\begin{aligned} C(n) &= i + C(n\text{-}i) \\ &= n\text{-}1 + C(n - (n\text{-}1)) \\ &= n\text{-}1 + C(1) \\ &= n\text{-}1 + 0 \\ &= n\text{-}1 \end{aligned}$$

This solves the recurrence: the number of concatenations done by the `reverse()` function on a string of length n is n-1 (which makes sense if you think about it).

Now lets do the same thing for the recurrence we generated for the Towers of Hanoi algorithm:

$$\begin{aligned} C(n) &= 1 + 2 \cdot C(n\text{-}1) \\ &= 1 + 2 \cdot (1 + 2 \cdot C(n\text{-}2)) && = 1 + 2 + 4 \cdot C(n\text{-}2) \\ &= 1 + 2 + 4 \cdot (1 + 2 \cdot C(n\text{-}3)) && = 1 + 2 + 4 + 8 \cdot C(n\text{-}3) \\ &= \ldots \\ &= 1 + 2 + 4 + \ldots + 2^i \cdot C(n\text{-}i) \end{aligned}$$

The initial condition for the Towers of Hanoi problem is $C(1) = 1$, and if we set i to n-1, then we can achieve this initial condition and thus remove the recursion:

$$\begin{aligned} C(n) &= 1 + 2 + 4 + \ldots + 2^i \cdot C(n\text{-}i) \\ &= 1 + 2 + 4 + \ldots + 2^{n\text{-}1} \cdot C(n\text{-}(n\text{-}1)) \\ &= 1 + 2 + 4 + \ldots + 2^{n\text{-}1} \cdot C(1) \\ &= 2^n - 1 \end{aligned}$$

Thus the number of moves made to solve the Towers of Hanoi puzzle for n disks is $2^n - 1$, which is obviously in $O(2^n)$.

14.4 Summary and Conclusion

Recurrences provide the technique we need to analyze recursive algorithms. Together with the summations technique we use with non-recursive algorithms, we are now in a position to analyze any algorithm we write. Often the analysis is mathematically difficult, so we may not always succeed in our analysis efforts. But at least we have techniques that we can use.

14.5 Review Questions

1. Use different initial conditions to show how the recurrence equation $F(n) = n \cdot F(n\text{-}1)$ has infinitely many solutions.
2. Consider the recurrence relation $F(n) = F(n\text{-}1) + F(n\text{-}2)$ for $n > 1$ with initial conditions $F(0) = F(1) = 1$. What well-known sequence of values is generated by this recurrence?
3. What are the four steps of the method of backward substitution?

14.6 Exercises

1. Write the values of the following recurrences for $n = 0$ to 4.
   a) $C(n) = 2 \cdot C(n\text{-}1)$, $C(0) = 1$
   b) $C(n) = 1 + 2 \cdot C(n\text{-}1)$, $C(0) = 0$
   c) $C(n) = b \cdot C(n\text{-}1)$, $C(0) = 1$ (b is some constant)
   d) $C(n) = n + C(n\text{-}1)$, $C(0) = 0$

2. Write the values of the following recurrences for $n = 1$, 2, 4, and 8.
   a) $C(n) = 2 \cdot C(n/2)$, $C(1) = 1$
   b) $C(n) = 1 + C(n/2)$, $C(1) = 0$
   c) $C(n) = n + 2 \cdot C(n/2)$, $C(1) = 0$
   d) $C(n) = n + C(n/2)$, $C(1) = 1$

3. Use the method of backward substitution to solve the following recurrences.
   a) $C(n) = 2 \cdot C(n\text{-}1)$, $C(0) = 1$
   b) $C(n) = 1 + 2 \cdot C(n\text{-}1)$, $C(0) = 0$
   c) $C(n) = b \cdot C(n\text{-}1)$, $C(0) = 1$ (b is some constant)
   d) $C(n) = n + C(n\text{-}1)$, $C(0) = 0$

4. Use the method of backward substitution to solve the following recurrences. Assume that $n = 2^k$ to solve these equations.
   a) $C(n) = 2 \cdot C(n/2)$, $C(1) = 1$
   b) $C(n) = 1 + C(n/2)$, $C(1) = 0$
   c) $C(n) = n + 2 \cdot C(n/2)$, $C(1) = 0$
   d) $C(n) = n + C(n/2)$, $C(1) = 1$

14.7 Review Question Answers

1. To see that $F(n) = n \cdot F(n\text{-}1)$ has infinitely many solutions, consider the sequence of initial conditions $F(0) = 0$, $F(0) = 1$, $F(0) = 2$, and so on. For initial condition $F(0) = 0$, $F(n) = 0$ for all n. For $F(0) = 1$, $F(n)$ is the factorial function. For $F(0) = 2$, $F(\text{n})$ is twice the factorial function, and in general for $F(0) = k$, $F(n) = k \cdot n!$ Hence infinitely many functions are generated by choosing different initial conditions.
2. This recurrence defines the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ….
3. The four steps of the method of backward substitution are (1) expand the recurrence relation several times until a pattern is detected, (2) express the pattern in terms of n and some index variable i, (3) Find a value for i that uses initial conditions to remove the recursion from the equation, (4) substitute the value for i and simplify to obtain a closed form for the recurrence.