Solving Recurrences

Recurrences and Recursive Code

Many (perhaps most) recursive algorithms fall into one of two categories: tail recursion and divide-and-conquer recursion. We would like to develop some tools that allow us to fairly easily determine the efficiency of these types of algorithms. We will follow a two-step process:

1. Express the running time (or use of some other resource) as a recurrence.
2. Solve the recurrence.

Our methods will also let us solve these kinds of recurrences when they arise in other contexts.

Tail Recursion

Tail recursion is the kind of recursion that pulls off 1 “unit” from the “end” (head) of the data and processes the remainder (the tail) recursively:

```
Foo (data, n) { // data is size n
  if ( n is small ) { BaseCase(data, n) }
  (head, tail) = PreProcess(data, n) // split data into head and tail
  PostProcess( head, Foo(tail, n-1) ) // put things together
}
```

If $p$ and $q$ are the running times of PreProcess() and PostProcess(), and $f$ is the running time of Foo(), then we have

$$f(n) = p(n) + f(n - 1) + q(n)$$

If we let $g(n) = p(n) + q(n)$, this simplifies to

$$f(n) = f(n - 1) + g(n).$$

So we will be interested in solving recurrences of this type. The base case(s) of the recurrence are determined by the efficiency of BaseCase().

Actually, we will look at recurrences of a somewhat more general type, namely

$$f(n) = af(n - 1) + bf(n - 2) + g(n),$$

but only for certain kinds of functions $g(n)$. This will allow us to handle things like the Fibonacci recurrence and a few other things beyond the basic tail recursion.

**Example.** Consider the following Python code to reverse the elements in an array:

```python
def reverse(a):
  if (len(a) <= 1):
    return a
  # for longer arrays we do this:
  firstElement = a[0]
  restOfArray = a[1:] # get all but first element
  return ( reverse(restOfArray).append(firstElement) )
```
To analyze the efficiency of this code, we would need to know the efficiency of \( a[1:] \) (which builds an array with all but the first element of \( a \)) and of the \texttt{append} method for arrays.

Note: This is not the best way to reverse the elements in an array in Python.

**Divide-and-Conquer**

Now consider a different kind of recursive function:

\[
\text{Foo}(\text{data}, n) = \begin{cases} 
\text{BaseCase(data, n)} & \text{if } n \text{ is small} \\
\end{cases}
\]

// split data into k chunks
\text{dataChunks} = \text{PreProcess(data, n)} // dataChunks is array-like

for (i=0; i < k; i++) { // process each chunk
    \text{processedChunks}[i] = \text{Foo(dataChunks}[i], n/k)
}
\text{PostProcess(data, processedChunks, n)} // often don’t need data here

If \( p \) and \( q \) are the running times of \text{PreProcess() \text{ and PostProcess()}}\), and \( f \) is the running time of \text{Foo()}, then we have

\[
f(n) = p(n) + kf(n/k) + q(n)
\]

This simplifies to

\[
f(n) = kf(n/k) + g(n)
\]

Again, for sufficiently nice \( g(n) \), we will be able to handle these recurrences.

Many sorting algorithms use a divide-and-conquer approach. Consider the following example:

\[
\text{Sort( data )} = \begin{cases} 
\text{return( data )} & \text{if length(data) == 1} \\
\end{cases}
\]

(left, right) = SplitData(data,n)
left = Sort(left)
right = Sort(right)
return( Merge( left, right ) )

**The Toolbox**

**Two General Methods**

Two general purpose methods for solving recurrences are

1. Guess and confirm.

Sometimes investigating a few terms can lead to a guess that can be verified by another method (usually induction).
Other times we can make a \textit{parameterized} guess, use the first few terms to solve for the parameters, and then show that the resulting formula is correct.

\textbf{Example.} Suppose we guess that
\[ S(n) = \sum_{i=0}^{n} i = An^2 + Bn + C \]
(a reasonable guess, since we can show that \( S(n) = \Theta(n^2) \)). Then we can solve for \( A \) and \( B \) and \( C \) using \( S(0) = 0 = C \), \( S(1) = 1 = A + B \), and \( S(2) = 3 = 4A + 2B \). This gives \( A = 1/2 \) and \( B = 1/2 \). Induction can be used to verify that \( S(n) = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2} \).

2. Unraveling.

\textbf{Example.} Suppose \( a_0 = 3 \), and \( a_n = 2a_{n-1} + 1 \). Then
\[
\begin{align*}
    a_n &= 2a_{n-1} + 1 \\
    &= 2[2a_{n-2} + 1] + 1 = 2^2a_{n-2} + [2 + 1] \\
    &= 2^2[2a_{n-3} + 1] + [2 + 1] = 2^3a_{n-3} + [2^2 + 2 + 1] \\
    &= 2^k a_{n-k} + [2^{k-1} + \cdots + 2^2 + 2 + 1] \\
    &= 2^n a_{n-n} + [2^{n-1} + \cdots + 2^2 + 2 + 1] \\
    &= 2^n a_0 + [2^{n-1} + \cdots + 2^2 + 2 + 1] \\
    &= 2^n \cdot 3 + [2^{n-1} + \cdots + 2^2 + 2 + 1] \\
    &= 2^n \cdot 3 + \sum_{j=0}^{n-1} 2^j
\end{align*}
\]
Since \( \sum_{j=0}^{n-1} 2^j = 2^{n-1} + \cdots + 2^2 + 2 + 1 = 2^n - 1 \) (geometric sum, see below), our unraveling simplifies to
\[
a_n = 3 \cdot 2^n + [2^n - 1].
\]

These two methods can be used to find general solutions for classes of recurrences that occur frequently in computer science, especially in recursive code.

Geometric Sums

Sums of the form \( \sum_{i=0}^{n} ar^i \) are called \textbf{geometric sums}, and we can always determine their value:

- If \( r = 1 \), then \( \sum_{i=0}^{n} ar^i = \sum_{i=0}^{n} a = (n + 1) \cdot a. \)
- If \( r \neq 1 \), then let \( G(n) = \sum_{i=0}^{n} ar^i \). Then \( rG(n) - G(n) = ar^{n+1} - a \) (write out the terms and watch for cancellation), so
\[
G(n) = \frac{ar^{n+1} - a}{r - 1} = \frac{a - ar^{n+1}}{1 - r}.
\]

A useful way to remember this formula is as follows: \( G = \frac{\text{first term} - \text{first missing term}}{1 - \text{ratio}} \).
Logsmanship

In order to simplify the results we get when working with geometric sums and divide-and-conquer recurrences, we will need to work with logarithms and exponential functions. Most of the identities used are probably familiar to you from other courses. One that may not be is identity

\[ a^{\log_b(n)} = n^{\log_b(a)}. \]

This is worth remembering. To see why it is true, consider taking the logarithm (base \( b \)) of each side.

\[ \log_b(a^{\log_b(n)}) = \log_b(n) \log_b(a) = \log_b(n) \log_b(a) = \log_b(n^{\log_b(a)}). \]

**Linear (Homogeneous) Recurrences (With Constant Coefficients)**

There are general methods for solving recurrences of the form

\[ a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + f(n), \]

where each of the \( c_i \) is a constant. If \( f(n) = 0 \), then this is a linear homogeneous recurrence relation (with constant coefficients). If \( f(n) \neq 0 \), then this is a linear non-homogeneous recurrence relation (with constant coefficients).

Here we will develop methods for solving the homogeneous case of degree 1 or 2. (Higher degree examples are done in a very similar way. A new wrinkle enters if we do non-homogeneous examples, but straightforward methods exist if \( f(n) \) is a polynomial or exponential function.)

For degree 1, simply unravel. (In fact, we can handle some simple functions \( f(n) \) in this case, too, as we did in the unraveling example above.) For degree 2, we use the method of parameterized guess and confirm. It seems reasonable that a solution to

\[ a_n = c_1 a_{n-1} + c_2 a_{n-2} \tag{1} \]

will have exponential growth (at least when the constants are positive) since it grows more quickly than a sequence defined by a first degree linear homogeneous recurrence. So let’s guess that \( a_n = \alpha r^n \) for some \( \alpha \) and \( r \).

Notice that

\[ \alpha r^n = c_1 \alpha r^{n-1} + c_2 \alpha r^{n-2}, \]

\[ r^2 = c_1 r - c_2. \]

This tells us how to find \( r \): \( r \) must be a solution to the polynomial equation \( t^2 - c_1 t - c_2 = 0 \). This polynomial in \( t \) is called the characteristic polynomial for the recurrence relation. By factoring or by the quadratic formula, such equations are easily solved and will in general have two solutions (we’ll call them \( r_1 \) and \( r_2 \)), although it is possible to have a “double root”, in which case \( r_1 = r_2 \) and the characteristic polynomial factors as \( (t - r_1)(t - r_2) \).

So far we know that for any constant \( \alpha \), \( \alpha r^n \) satisfies the recurrence relation in (1), but we don’t know if those are the only solutions. In fact they are not, and we need a slightly more general guess. But let’s look at an example first.
An Example

Let’s look at the recurrence relation \( a_n = a_{n-1} + 2a_{n-2} \). We’ll consider several possible initial conditions.

1. \( a_0 = 3; a_1 = 6 \).
   
   Our characteristic polynomial for this recurrence is \( t^2 - t - 2 = (t - 2)(t + 1) \). So the roots are \( r_1 = 2 \) and \( r_2 = -1 \). Let’s see if \( a_n = \alpha 2^n \) for some choice of \( \alpha \). This would mean that
   
   \[ a_0 = 3 = \alpha 2^0 = \alpha \]
   
   So does \( a_n = 3 \cdot 2^n \)? Let’s check \( a_1 \): We see that \( a_1 = 6 = 3 \cdot 2^1 \). Since our formula matches on the initial conditions and satisfies the recurrence relation, it must match the sequence \( a_n \) exactly!

2. \( a_0 = -2; a_1 = 2 \).
   
   The characteristic function is the same, with roots \( r_1 = 2 \) and \( r_2 = -1 \). Let’s try \( \alpha 2^n \) again.
   
   \[ a_0 = -2 = \alpha 2^0 = \alpha \]
   
   So this time \( \alpha = -2 \). But there is a problem: \( a_1 = 2 \) but \( \alpha 2^1 = (-2)(2) = -4 \). So \( -2 \cdot 2^n \) is NOT the formula for our sequence.
   
   Fortunately, there is another root to try. If we try \( \alpha (-1)^n \) we get
   
   \[ a_0 = -2 = \alpha (-1)^0 = \alpha \]
   
   And this time we see that \( \alpha (-1)^1 = (-2)(-1) = 2 = a_1 \). Once again, since our formula matches on the initial conditions and satisfies the recurrence relation, it must match the sequence \( a_n \) exactly.

3. \( a_0 = 5; a_1 = 4 \).
   
   This time neither of our roots will lead to a solution. (Check this out for yourself.)

So we need something else to guess. Our guess will be that the solution is a combination of solutions of the form above (using both solutions to the characteristic polynomial).

The general method

The solutions to a linear homogeneous recurrence relation of degree two can be solved by the method of parameterized guessing using the following guesses:

1. If the characteristic polynomial has two solutions, then \( a_n = \alpha r_1^n + \beta r_2^n \), where \( r_1 \) and \( r_2 \) are the two solutions of \( t^2 - c_1 t - c_2 \) [i.e., of \( 0 = t^2 - c_1 t - c_2 \)],

2. If the characteristic polynomial has one solution (a double root): \( a_n = \alpha r^n + \beta n r^n \), where \( r \) is the only solution of \( t^2 - c_1 t - c_2 \) [i.e., of \( 0 = t^2 - c_1 t - c_2 \)].

It turns out that these guesses are always correct, and it is not terribly difficult to find the parameters involved. We can find the roots of the characteristic polynomial \( t^2 - c_1 t - c_2 \) by factoring or by the quadratic formula. We can then find \( \alpha \) and \( \beta \) by comparing with the first two terms of the sequence \( (a_0 \text{ and } a_1) \).
Examples

1. \(a_n = a_{n-1} + 2a_{n-2}; a_0 = 5; a_1 = 4.\)

This is linear homogeneous of degree 2. The characteristic polynomial is \(t^2 - t - 2 = (t - 2)(t + 1)\) so the two roots are 2 and -1. \(\alpha 2^0 + \beta(-1)^0 = \alpha + \beta = a_0 = 5,\) and \(\alpha 2^1 + \beta(-1)^1 = 2\alpha - \beta = a_0 = 4.\)

Solving for \(\alpha\) and \(\beta\) shows that \(\alpha = 3\) and \(\beta = 2,\) so

\[
a_n = 3 \cdot 2^n + 2(-1)^n = 3 \cdot 2^n + 2.
\]

(Whether we add or subtract 2 depends on the parity of \(n.\))

2. \(a_n = 6a_{n-1} - 9a_{n-2}; a_0 = 2; a_1 = 9.\)

This is linear homogeneous of degree 2. The characteristic polynomial is \(t^2 - 6t + 9 = (t - 3)(t - 3)\) so there is only one root (3). This means that \(a_n = \alpha 3^n + \beta n3^n\) for some \(\alpha\) and \(\beta.\) \(\alpha 3^0 + \beta 0(3)^0 = \alpha = a_0 = 2,\)

and \(\alpha 3^1 + \beta 1(3)^1 = 2(3) + \beta(1)(3) = a_1 = 9.\) So \(\beta = 1,\) and

\[
a_n = 2 \cdot 3^n + n3^n.
\]

Why the Method Works

So why does this method work? We need to show two things. Let’s do it for the case that there are two distinct roots to the characteristic polynomial; the other case is similar.

1. Our guesses always satisfy the recurrence relation.

That is we must show that \(\alpha r_1^n + \beta r_2^n\) will satisfy the recurrence relation \(a_n = c_1a_{n-1} + c_2a_{n-2}.\) We already know that \(\alpha r_1^n\) satisfies the recurrence relation and \(\beta r_2^n\) satisfies the recurrence relation. So we just need to show that if \(s_n\) and \(t_n\) are sequences that satisfy the recurrence relation, then so is \(u_n = s_n + t_n.\) Let’s give it a try:

\[
c_1u_{n-1} + c_2u_{n-2} = c_1(s_{n-1} + t_{n-1}) + c_2(s_{n-2} + s_{n-2})
\]

\[
= c_1s_{n-1} + c_1t_{n-1} + c_2s_{n-2} + c_2t_{n-2}
\]

\[
= c_1s_{n-1} + c_2s_{n-2} + c_1t_{n-1} + c_2t_{n-2}
\]

\[
= s_n + t_n = u_n
\]

So \(u_n\) satisfies the recurrence relation, just as we wanted.

2. Once we have determined \(r_1\) and \(r_2,\) we can always solve for \(\alpha\) and \(\beta\) to match any initial conditions.

For this, let’s look at the part of the process where we need to solve for \(\alpha\) and \(\beta:\)

\[
a_0 = \alpha r_1^0 + \beta r_2^0 = \alpha + \beta
\]

\[
a_1 = \alpha r_1^1 + \beta r_2^1 = r_1\alpha + r_2\beta
\]

This is a system of two linear equations with two unknowns, and as long as \(r_1 \neq r_2,\) it will always have a unique solution. (If \(r_1 = r_2\) there may not be a solution. This is why we have to have a separate case to handle double roots of the characteristic polynomial.)
Divide-and-Conquer

Divide-and-conquer recursion leads to running times that can be expressed using recurrences like

\[ f(n) = af(n/b) + g(n), \]

where \( a \geq 1 \) and \( b > 1 \). Roughly, \( a \) is the number of subproblems processed (number of recursive calls), \( b \) measures the size of the subproblems, and \( g(n) \) is the overhead from the parts of the algorithm not involved in the recursive calls.

The tools needed to analyze this kind of recurrence are unraveling and geometric sums. Here we will deal with the cases where \( g(n) = Cn^d = O(n^d) \) for constants \( C \) and \( d \). This provides big O estimates whenever \( g \) is a polynomial.

Let’s look at some cases:

1. If \( g(n) = C \) for some constant \( C \) (i.e., \( d = 0 \)), then unraveling yields:

\[
\begin{align*}
  f(n) &= af(n/b) + C \\
  &= a[af(n/b^2) + C] + C \\
  &= a^2 f(n/b^2) + C[a + 1] \\
  &= a^3 f(n/b^3) + C[a^2 + a + 1] \\
  &= a^k f(n/b^k) + C[a^{k-1} + a^{k-2} + \cdots + a + 1] \\
  &= a^{\log_b(n)} f(1) + C \sum_{j=0}^{\log_b(n)-1} a^j \quad \text{(geometric sum)}
\end{align*}
\]

(a) If \( a = 1 \) (so the recurrence is \( f(n) = f(n/b) + C \)), then the geometric sum is easy and (1) becomes

\[
\begin{align*}
  f(n) &= a^{\log_b(n)} f(1) + C \sum_{j=0}^{\log_b(n)-1} a^j \\
  &= 1^{\log_b(n)} f(1) + C \sum_{j=0}^{\log_b(n)-1} 1 \\
  &= f(1) + C \log_b(n) = O(\log n)
\end{align*}
\]

(b) If \( a > 1 \), then we can use our knowledge of geometric series and (1) becomes

\[
\begin{align*}
  f(n) &= a^{\log_b(n)} f(1) + C \sum_{j=0}^{\log_b(n)-1} a^j \\
  &= a^{\log_b(n)} f(1) + C \frac{a^{\log_b(n)} - 1}{a - 1} \\
  &= a^{\log_b(n)} [f(1) + C \frac{1}{a - 1} - \frac{1}{a - 1}] \\
  &= O(a^{\log_b(n)}) = O(n^{\log_b(n)})
\end{align*}
\]
2. If \( g(n) = Cn^d \) (\( d > 0 \)), then our unraveling is a bit messier, but the same basic ideas work:

\[
\begin{align*}
f(n) &= a f(n/b) + g(n) \\
&= a[a f(n/b^2) + g(n/b)] + g(n) \\
&= a^2 f(n/b^2) + [ag(n/b) + g(n)] \\
&= a^3 f(n/b^3) + [a^2 g(n/b^2) + ag(n/b) + g(n)] \\
&= a^k f(n/b^k) + \sum_{j=0}^{k-1} a^j g(n/b^j) \\
&= \left[ a^\log_b(n) f(1) + \sum_{j=0}^{\log_b(n)-1} a^j C \left( \frac{n}{b^j} \right)^d \right] + \left[ a^\log_b(n) f(1) + \sum_{j=0}^{\log_b(n)-1} a^j C \left( \frac{n^d}{b^jd} \right) \right] \\
&= n^\log_b(a) f(1) + Cn^d \sum_{j=0}^{\log_b(n)-1} \left( \frac{a}{b^j} \right)^j \tag{3}
\end{align*}
\]

As before, we have two terms, and once again which of these two terms is larger, depends on the relationship of \( a, b, \) and \( d \).

(a) If \( a = b^d \) (i.e., \( d = \log_b(a) \)), then \( \frac{a}{b^d} = 1 \), so the second term in (3)

\[
Cn^d \sum_{j=0}^{\log_b(n)-1} 1 = O(n^d \log(n))
\]

and dominates the first term.

(b) If \( a < b^d \) (i.e., \( d > \log_b(a) \)), then \( n^d > n^{\log_b(a)} \), so the second term in (3) again dominates. And

\[
\text{since } \frac{a}{b^d} < 1, \text{ the geometric sum is bounded by a constant, so } f(n) = O(n^d).
\]

(c) Finally, if \( a > b^d \) (i.e., \( d < \log_b(a) \)), then (3) can be simplified as follows. (Notice that the boxed quantities are \( O(1) \).)

\[
\begin{align*}
f(n) &= n^\log_b(a) f(1) + Cn^d \left( \frac{a}{b^d} \right)^{\log_b(n)} - 1 \\
&= n^\log_b(a) f(1) \left( Cn^d \left( \frac{a}{b^d} \right)^{\log_b(n)} \frac{1}{\log_b(n)} \right) + \left( Cn^d \frac{-1}{\log_b(n)} \right) \\
&= n^\log_b(a) f(1) + n^d \left( a^{\log_b(n)} \frac{C}{\log_b(n)} \frac{1}{\log_b(n)} \right) + \left( Cn^d \frac{-1}{\log_b(n)} \right) \\
&= n^\log_b(a) f(1) + n^d a^{\log_b(n)} \frac{C}{\log_b(n)} + \left( Cn^d \frac{-1}{\log_b(n)} \right) \\
&= n^\log_b(a) f(1) + n^d a^{\log_b(n)} \frac{C}{\log_b(n)} + \left( Cn^d \frac{-1}{\log_b(n)} \right) \\
&= O(n^\log_b(a)) + O(n^\log_b(a)) + O(n^d) = O(n^\log_b(a))
\end{align*}
\]
The results above can be summarized as follows:

\[
\begin{align*}
\text{If } f(n) &= af(n/b) + g(n), \text{ and } g(n) = Cn^d, \text{ then } f(n) = \\
&= \begin{cases} 
O(\log n) & \text{if } d = 0 \text{ and } a = 1 \\
O(n^{\log_b(a)}) & \text{if } d = 0 \text{ and } a > 1 \\
O(n^d) & \text{if } d > 0 \text{ and } d > \log_b(a) \\
O(n^d \log n) & \text{if } d > 0 \text{ and } d = \log_b(a) \\
O(n^{\log_b(a)}) & \text{if } d > 0 \text{ and } d < \log_b(a)
\end{cases}
\end{align*}
\]

Things to notice:

1. \( C \) does not occur in any of the expressions on the right hand side, so a big O analysis is not sensitive to the constant \( C \), and we could replace the left hand side with

\[ f(n) = af(n/b) + O(n^d). \]

2. The first two cases look just like instances of the last three with \( d = 0 \).

Examples

1. The running time of a binary search algorithm satisfies \( t(n) = t(n/2) + O(1) \), since we split the array into two halves and recursively check only one and the overhead cost for the split is just a little bit of arithmetic that takes the same amount of time regardless of the size of the array. By the results above \( t(n) = O(\log n) \).

2. The running time of MergeSort satisfies \( t(n) = 2t(n/2) + O(n) \), since we split the array into two halves and recursively sort each one and then have to merge the results (at a fixed cost per item merged). By the results above (this is a case where \( a = 2 > 1 \) and \( d = \log_b(a) = \log_2(2) \)):

\[ t(n) = O(n \log n) \].