1 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.

1.1 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:

```c
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

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

In this formulation, $f$ represents the time spent in recursive calls and $g$ represents the time spent on the non-recursive portion of the algorithm. 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\footnote{This is not the best way to reverse the elements in an array in Python.} to reverse the elements in a python list (which are implemented like what are called arrays in most languages):

```python
def reverse(a):
    if (len(a) <= 1):
        return a
    # for longer lists we do this:
    firstElement = a[0]
    restOfList = a[1:]  # get all but first element
    return (reverse(restOfList).append(firstElement) )
```

To analyze the time efficiency of this code, we would need to know

1. the efficiency of `len(a)`. \(O(1)\)

   In python this is \(O(1)\), but in some other languages or in user-created list-like data structures (basic linked lists, for example), might require walking along the array and counting. See \url{https://wiki.python.org/moin/TimeComplexity} for a list of efficiencies of many built-in functions in python.

2. the efficiency of `a[1:]` \(O(n-1) = O(n)\)

   This builds an array with all but the first element of \(a\) and is referred to as ”getting a slice” at \url{https://wiki.python.org/moin/TimeComplexity}. The cost here comes from having to copy the \(n-1\) elements from the current list to a new one. One could imagine an implementation that didn’t require this (but would have some different behaviors because of the shared memory).\footnote{A language like R, which does “copy on modify” can have interesting behavior in this regard. The creation of objects of this sort is very efficient as long as neither the original nor the copy change. But each maintains a flag indicating that it is not the only item pointing to the memory it is using, and when eithet one makes a change, then it must do the copying. This can save time when copies are made but not modified, but it just defers the cost if a modification happens later in the algorithm.}

3. the efficiency of the `append` method for lists. \(O(1)\)

   Note that in this case we are relying on “amortized” worst case or average case complexity. The worst case complexity is \(O(n)\), but that can’t happen very often. (The worst case is when the array outgrows the memory allocated for it and needs to be copied to a new location with expanded memory.) Since the lists used here don’t grow, \(O(1)\) is the appropriate complexity in this algorithm. In algorithms that grow the list as they go along, we can also use \(O(1)\), but there will be some (rare) steps along the way where the cost will be much higher. In many languages, pre-allocation of memory (when the sizes are known in advance) can avoid the copying to a new location and speed things up.

Now let’s put this all together. Let \(f(n)\) be the time complexity of our algorithm on a list of length \(n\). Our base case is \(O(1)\), so \(f(1) = C\) for some constant. In other cases, the total amount of time spent outside the recursive call is \(O(n)\). So our recurrence relation is

\[
f(n) = f(n-1) + g(n) \quad \text{where } g(n) \text{ is } O(n).
\]

We will typically write this as

\[
f(n) = f(n-1) + O(n) .
\]
1.2 Divide-and-Conquer

Now consider a different kind of recursive function:

```plaintext
Foo (data, n) {
    if (n is small) { return( BaseCase(data, n) ) }

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

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

If $p$ and $q$ are the running times of $\text{PreProcess()}$ and $\text{PostProcess()}$, 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.

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

```plaintext
Sort( data ) {
    if (length(data) == 1) { return( data ) }

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

The recurrence relation for this algorithm is

$$f(n) = 2f(n/2) + O(1) + O(n) = 2f(n/2) + O(n)$$

if splitting can be done “in place” and merging is done by copying from two containers of size $n/2$ to a new container of size $n$. Note that we would still have

$$f(n) = 2f(n/2) + O(n) + O(n) = 2f(n/2) + O(n)$$

if we split into two containers in new memory and had to do the copying as part of our splitting step as well.

We won’t know how much this matters until we see how the constants involved in $O(n)$ affect the complexity of $f$. 

2 The Toolbox

We begin this section with two general purpose methods for solving recurrences (guess and confirm, and unraveling) and then include some tools for simplifying expressions that arise when solving recurrences.

2.1 Solving Recurrences by Guess and Confirm

Sometimes investigating a few terms or conducting Big O analysis can lead to a guess that can be verified by another method.

Other times we can make a parameterized guess, use the first few terms to solve for the parameters, and then show that the resulting formula is correct.

**Example.** Suppose we guess that

\[ S(n) = \sum_{i=0}^{n} i = An^2 + Bn + C \]

(a reasonable guess, since we can show that \( S(n) = \Theta(n^2) \)). Notice that \( S(n) = S(n-1) + n \) and \( S(0) = 0 \), so we can also define \( S \) using a recurrence.

We can solve for \( A \) and \( B \) and \( C \) using 

\[ S(0) = 0 = C, \quad S(1) = 1 = A + B, \quad \text{and} \quad S(2) = 3 = 4A + 2B. \]

This gives \( A = 1/2 \) and \( B = 1/2 \).

Now we need to verify that this works.

- For starters, let’s check the next couple terms. \( S(3) = 6 = \frac{3(4)}{2} \), and \( S(4) = 10 = \frac{4(5)}{2} \), so things are looking good. But we want to know that this continues forever.

- Induction can be used to verify that \( S(n) = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2} \). We’ll cover induction later in the course.

- Alternatively, we can show that \( \frac{n(n+1)}{2} \) satisfies the recurrence relation. (That is, we must show that \( \frac{n(n+1)}{2} = \frac{(n-1)n}{2} + n \).) If it satisfies the recurrence relation and matches on the initial conditions, then it must match on all terms.

\[
\frac{n(n+1)}{2} = \frac{n^2 + n}{2} = \frac{n^2 - n + 2n}{2} = \frac{n^2 - n}{2} + n = \frac{(n-1)n}{2} + n
\]

So our formula does satisfy the recurrence.

**Your Turn.** Suppose we guess that

\[ T(n) = \sum_{i=0}^{n} 2i + 1 = An^2 + Bn + C. \]

What is the associated recurrence relation here? What must the constants \( A, B, \) and \( C \) be if we are right? Are we right?
2.2 Solving Recurrences by Unraveling

Unraveling is a bit like pulling on the loose end of yarn in a knitting project. We express \( a_n \) in terms of \( a_{n-1} \), which we express in terms of \( a_{n-2} \), which we express \ldots

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.

2.3 Geometric Sums

Sums of the form \( \sum_{i=0}^{n} ar^i \) are called 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: \( G = \frac{\text{first term} - \text{first missing term}}{1 - \text{ratio}} \).

2.4 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.

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

3 Linear Homogeneous Recurrences With Constant Coefficients

There are general methods for solving recurrences of the form

\[ a_n = c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} + f(n), \]

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 – see the next section.)

3.1 Degree 1 Linear Homogeneous Recurrence Relations

For degree 1, we simply unravel. (In fact, we can handle some simple functions \( f(n) \) in this case, too, as we did in the unraveling example above.)

**Example.** Suppose \( a_n = \frac{3}{2}a_{n-1} \) and \( a_0 = 2 \). Find an explicit formula for \( a_n \).

\[
\begin{align*}
a_n &= (3/2)a_{n-1} \\
&= (3/2)(3/2)a_{n-2} \\
&= (3/2)^2a_{n-2} \\
&= (3/2)^3a_{n-3} \\
&= (3/2)^na_{n-n} \\
&= 2(3/2)^n \\
&= \frac{3^n}{2^{n-1}}
\end{align*}
\]

**Your Turn.** We can actually work out a very general formula here. Suppose \( a_n = ra_{n-1} \) and \( a_0 = \alpha \). Find an explicit formula for \( a_n \) in terms of \( r \) and \( \alpha \).
3.2 Degree 2 Linear Homogeneous Recurrence Relations

For degree 2, we use the method of parameterized guess and confirm. It seems reasonable that a solution to

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

(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

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

\[ \Downarrow \]

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

\[ \Downarrow \]

\[ r^2 - c_1 r - c_2 = 0 . \]

This tells us how to find \( r \): \( r \) must be a solution to the polynomial equation \( r^2 - c_1 r - c_2 = 0 \). This polynomial in \( r \) 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 \( (r - r_1)(r - r_2) \).

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.

3.3 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 \( r^2 - r - 2 = (r - 2)(r + 1) \). 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 \( -2)2^n \) 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.

Math 156 – Fall 2016
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).

3.4 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 \( r^2 = c_1 r + c_2 = 0 \) [i.e., of \( 0 = r^2 - c_1 r - c_2 \)],

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 \( r^2 = c_1 r + c_2 \) [i.e., of \( 0 = r^2 - c_1 r - c_2 \)].

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 \( r^2 - c_1 r - c_2 \) by factoring or by the quadratic formula. We can then find \( \alpha \) and \( \beta \) by comparing with the first two terms of the sequence (\( a_0 \) and \( a_1 \)).

3.5 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 \( r^2 - r - 2 = (r - 2)(r + 1) \) 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
   \[ a_n = 3 \cdot 2^n + 2(-1)^n = 3 \cdot 2^n \pm 2. \]
   (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 \( r^2 - 6r + 9 = (r - 3)(r - 3) \) so there is only one root (3). This means that \( a_n = \alpha 3^n + \beta n 3^n \) 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
   \[ a_n = 2 \cdot 3^n + n 3^n. \]
3.6 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:

\[
\begin{align*}
c_1u_{n-1} + c_2u_{n-2} &= c_1(s_{n-1} + t_{n-1}) + c_2(s_{n-2} + s_{n-2}) \\
&= c_1s_{n-1} + c_1t_{n-1} + c_2s_{n-2} + c_2s_{n-2} \\
&= c_1s_{n-1} + c_2s_{n-2} + c_1t_{n-1} + c_2t_{n-2} \\
&= s_n + t_n = u_n
\end{align*}
\]

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 \( \beta \):

\[
\begin{align*}
a_0 &= \alpha r_1^0 + \beta r_2^0 = \alpha + \beta \\
a_1 &= \alpha r_1^1 + \beta r_2^1 = r_1\alpha + r_2\beta
\end{align*}
\]

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.)

4 Linear Nonhomogeneous Recurrences with Constant Coefficients

Now let’s consider

\[ a_n = c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} + f(n) , \]

where \( f(n) \) is not identically 0. The clever idea here is to consider difference between two solutions to this recurrence. Let’s call them \( a \) and \( b \):

\[
\begin{align*}
a_n &= c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} + f(n) \\
b_n &= c_1b_{n-1} + c_2b_{n-2} + \cdots + c_kb_{n-k} + f(n)
\end{align*}
\]

Then

\[
\begin{align*}
a_n - b_n &= +c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} + f(n) \\
&-c_1b_{n-1} - c_2b_{n-2} - \cdots - c_kb_{n-k} - f(n) \\
&= c_1(a_{n-1} - b_{n-1}) + c_2(a_{n-2} - b_{n-2}) + \cdots + c_k(a_{n-k} - b_{n-k})
\end{align*}
\]
So the difference \( h_n = a_n - b_n \) is a solution to the associated linear \emph{homogeneous} recurrence. Read the other way around, this says that any solution \( a_n \) can be written as \( h_n + b_n \). So to solve a linear nonhomogeneous recurrence, we do the following

1. Find the general form for a solution to the associated homogeneous recurrence (\( h_n \)).
   
   Note: \( h_n \) will involve unknown constants because we are not using the initial conditions at this point. We need the general form here.

2. Find a particular solution to the nonhomogenous recurrence (\( b_n \)).
   
   If \( f(n) \) is a polynomial or exponential (or the product of the two), we can use the method of parameterized guess and confirm.
   
   (a) If \( f(n) \) is a polynomial, guess a polynomial of the same degree.
   
   (b) If \( f(n) = s^n \) is exponential, and \( s \) is not a root of the characteristic equation, guess \( Cs^n \).
   
   (c) If \( f(n) = s^n \) is exponential, and \( s \) is a root of the characteristic equation with multiplicity \( m \), guess \( Cn^m s^n \).
   
   (d) If \( f(n) \) is the product of a polynomial and an exponential, guess the product of the guesses for the polynomial and the exponential.

   Again, we do not use the initial conditions at this point. Our guesses are guaranteed to lead to a solution to the recurrence, but it might not be the one that matches our initial conditions. But we can solve for the constants in our parameterized guess using the recurrence relation alone.

   The general form of the solution is now \( h_n + b_n \), with unknown constants appearing in \( h_n \) but not in \( b_n \).

3. If we have initial conditions, we can solve for the unknown constants in \( a_n = h_n + b_n \).

\textbf{Example.} Let \( a_n = 3a_{n-1} + 2n \). Solve for \( a_n \) if \( a_1 = 3 \).

1. The associated homogeneous recurrence is \( h_n = 3a_{n-1} \), and general form of the solution is \( h_n = \alpha 3^n \).

2. Since \( f \) is a polynomial of degree 1, we guess that \( b_n = An + B \) is the correct form and solve for \( A \) and \( B \).
   \[ An + B = 3(A(n - 1) + B) + 2n = 3An - 3A + 3B + 2n, \]
   \[ so \ A = 3A + 2 \] and \( B = 3B - 3A \). From this we obtain \( A = -1 \) and \( B = -\frac{3}{2} \).

3. Our general form is now \( a_n = h_n + b_n = \alpha 3^n - n + \frac{3}{2} \). We can solve for \( \alpha \) using the initial condition.
   \[ 3 = a_1 = \alpha 3 - 1 - \frac{3}{2}, \]
   \[ so \ 3\alpha = 3 + 1 + \frac{3}{2} \] and \( \alpha = \frac{11}{6} \).
   
   Final solution: \( \frac{11}{6} 3^n - n + \frac{3}{2} \).

\textbf{Your turn.} Find all solutions to the recurrence

\[ a_n = 5a_{n-1} - 6a_{n-1} + 7^n. \]
5 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 (2) becomes

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

(b) If \( a > 1 \), then we can use our knowledge of geometric series and (2) becomes

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

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

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) is

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

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 since $\frac{a}{b^d} < 1$, 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 $O(1)$.)

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recurrence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ( f(n) = af(n/b) + g(n) ), and ( g(n) = Cn^d ), then ( f(n) = \begin{cases} O(\log n) &amp; \text{if } d = 0 \text{ and } a = 1 \ O(n^{\log_b(a)}) &amp; \text{if } d = 0 \text{ and } a &gt; 1 \ O(n^d) &amp; \text{if } d &gt; 0 \text{ and } d &gt; \log_b(a) \ O(n^d \log n) &amp; \text{if } d &gt; 0 \text{ and } d = \log_b(a) \ O(n^{\log_b(a)}) &amp; \text{if } d &gt; 0 \text{ and } d &lt; \log_b(a) \end{cases} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \( d = 0 \).

5.1 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 \( d = \log_b(a) = \log_2(2) \)):

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