1 Recurrence Relations

Suppose we are analyzing the time complexity $T(n)$ for an algorithm and suppose that we cannot find an explicit closed formula for $T(n)$ but instead we find a recurrence relation which $T(n)$ satisfies.

A recurrence relation for $T(n)$ tells us how to calculate $T(n)$ for various $n$ in a recursive manner.

**Example 1.1.** Suppose we have:

$$T(n) = 3T(\lfloor n/5 \rfloor) + 2n + 1 \text{ and } T(1) = 4$$

Then for example here are some easy ones:

\begin{align*}
T(1) &= 4 \\
T(5) &= 3T(1) + 2(5) + 1 = 23 \\
T(25) &= 3T(4) + 2(25) + 1 = 120
\end{align*}

A slightly more annoying one is:

$$T(7) = 3T(\lfloor 7/5 \rfloor) + 2(7) + 1 = 3T(1) + 15 = 27$$

An even more annoying one would be $T(2)$ because we’re not given enough information. In reality we would need to be given $T(2)$, $T(3)$, and $T(4)$ as well. Luckily since we’re only really usually really concerned with large $n$ values we can get away with minimal base cases.

Formally a recurrence relation like this should have a floor or ceiling inside the recursive $T$ but in practice this typically dropped for purposes of not getting too tangled up in the calculations. For example we might just write:

$$T(n) = 3T(n/5) + 2n + 1 \text{ and } T(1) = 4$$

When we do this every specific calculation that follows becomes an approximation when the division yields a non-integer but these approximations are good enough and don’t affect time complexity.
2 Solving Using Trees

We can use a recursively generated tree to find an explicit formula for $T(n)$. Such an approach can be messy or not, depending on the recurrence relation and on the $n$-values we’re analyzing. So as not to go off the deep end, let’s consider the example again:

**Example 2.1.** Suppose $T(n) = 3T(n/5) + (2n + 1)$ with $T(1) = 4$. Let’s examine $T(n)$ where $n = 5^k$ for some $k$. To understand why there’s a tree involved we can view the total time done as a very small tree, first with one node:

$$T(n)$$

We can expand this according to the recurrence relation, the tree still showing the total time:

$$2n + 1$$

$$\quad \quad \quad \quad \quad T(n/5) \quad T(n/5) \quad T(n/5)$$

Of course each of these leaves is its own problem, the tree still showing the total time.

$$2n + 1$$

$$\quad \quad \quad 2(n/5) + 1 \quad 2(n/5) + 1 \quad 2(n/5) + 1$$

$$\quad \quad \quad T(n/25) \quad T(n/25) \quad T(n/25)$$

We could keep going...
But when does it stop?
The tree stops growing when we reach \( T(1) \) in the nodes because \( T(1) = 4 \) and no more recursion happens.

If layer \( i = 0 \) is the top layer then as the tree grows we see that layer \( i \) has entries \( T(n/5^i) \) so the leaf layer will happen when we reach \( T(1) \) which is when \( n/5^i = 1 \) or \( i = \log_5 n = k \). Thus the \( k^{th} \) layer is the leaf layer.

Thus in total there are \( k+1 \) layers, 0, 1, ..., \( k \), and in those layers:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Count</th>
<th>Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>1</td>
<td>( 2n + 1 )</td>
<td>( 1(2n + 1) = 3^0 (2 (\frac{n}{5^1}) + 1) )</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>3</td>
<td>( 2 (\frac{n}{5}) + 1 )</td>
<td>( 3 (2 (\frac{n}{5}) + 1) = 3^1 (2 (\frac{n}{5}) + 1) )</td>
</tr>
<tr>
<td>( i = 2 )</td>
<td>9</td>
<td>( 2 (\frac{n}{25}) + 1 )</td>
<td>( 9 (2 (\frac{n}{25}) + 1) = 3^2 (2 (\frac{n}{25}) + 1) )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>( i = k - 1 )</td>
<td>( 3^{k-1} )</td>
<td>( 2 (\frac{n}{5^{k-1}}) + 1 )</td>
<td>( 3^{k-1} (2 (\frac{n}{5^{k-1}}) + 1) )</td>
</tr>
<tr>
<td>( i = k )</td>
<td>( 3^k )</td>
<td>4</td>
<td>( 3^k (4) )</td>
</tr>
</tbody>
</table>

Thus the total time is:

\[
T(n) = 4(3^k) + \sum_{i=0}^{k-1} 3^i \left( 2 \left( \frac{n}{5^i} \right) + 1 \right) \\
= 4(3^k) + 2n \sum_{i=0}^{k-1} \left( \frac{3}{5} \right)^i + \sum_{i=0}^{k-1} 3^i \\
= 4(3^k) + 2n \left( \frac{1 - (\frac{3}{5})^k}{1 - \frac{3}{5}} \right) + \left( \frac{3^k - 1}{3 - 1} \right) \\
= 4(3^k) + 2n \left( \frac{5}{2} \right) \left( 1 - \left( \frac{3}{5} \right)^k \right) + \frac{1}{2} \left( 3^k - 1 \right) \\
= 4(3^\log_5 n) + 5n - 5n \left( \frac{3}{5} \right)^\log_5 n + \frac{1}{2} \left( 3^\log_5 n \right) - \frac{1}{2} \\
= \frac{9}{2}(3^\log_5 n) + 5n - 5n \left( \frac{3^\log_5 n}{5^\log_5 n} \right) - \frac{1}{2} \\
= \frac{9}{2}(3^\log_5 n) + 5n - \frac{1}{2} \left( 3^\log_5 n \right) - \frac{1}{2} \\
= -\frac{1}{2}(3^\log_5 n) + 5n - \frac{1}{2} \\
= 5n - \frac{1}{2}(3^\log_5 n + 1)
\]
Note that results from this equation will agree with the calculations we did earlier. For example:

\[ T(5) = 5(5) - \frac{1}{2}(3^{\log_5 5} + 1) = 25 - \frac{1}{2}(3 + 1) = 23 \]

And:

\[ T(25) = 5(25) - \frac{1}{2}(3^{\log_5 25} + 1) = 125 - \frac{1}{2}(9 + 1) = 120 \]

While this formula is exact only for \( n = 5^k \) it gives us a good approximation in other cases:

\[ T(17) = 5(17) - \frac{1}{2}(3^{\log_5 17} + 1) \approx 81.04 \]

Here is a graph of \( T(n) \) as well as these points:
As far as time complexity observe that:

\[ T(n) = 5n - \frac{1}{2}(3^{\log_5 n} + 1) < 5n \]

Thus \( T(n) = O(n) \).

And observe that since \( \log_5 n < \log_3 n \) we have \( 3^{\log_5 n} < 3^{\log_3 n} = n \) so that for \( n \geq 1 \) we have:

\[ T(n) = 5n - \frac{1}{2}(3^{\log_5 n} + 1) > 5n - \frac{1}{2}(n + 1) = \frac{9}{2}n - \frac{1}{2} \geq 4n \]

Thus \( T(n) = \Omega(n) \) and together we have \( T(n) = \Theta(n) \).

This corresponds to the picture in the sense that it certainly appears that \( Bn \leq T(n) \leq Cn \) for some \( B, C > 0 \) and for sufficiently large \( n \). In fact here is the same plot but with \( 4n \) and \( 5n \) plotted as well:
By popular demand here is a second tree example but more streamlined.

**Example 2.2.** Consider the recurrence relation given here:

\[ T(n) = 2T(n/2) + \sqrt{n} \text{ with } T(1) = 3 \]

For the sake of simplicity assume \( n = 2^k \) for some \( k \). During the growth of the tree the nodes in layer \( i \) are \( T(n/2^i) \) before being replaced by \( \sqrt{n/2^i} \) as the tree grows down. The leaf layer occurs when \( n/2^i = 1 \) or \( i = \lg n = k \).

Thus in total there are \( k+1 \) layers, 0, 1, ..., \( k+1 \) and in those layers:

<table>
<thead>
<tr>
<th>Layer ( i )</th>
<th>Count ( 2^i )</th>
<th>Time ( \sqrt{n/2^i} )</th>
<th>Total Time ( 2^{k-i-1} \sqrt{n/2^i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>1</td>
<td>( \sqrt{n} )</td>
<td>( \sqrt{n} = \sqrt{1n} )</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>2</td>
<td>( \sqrt{n/2} )</td>
<td>( 2\sqrt{n/2} = \sqrt{2n} )</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>4</td>
<td>( \sqrt{n/4} )</td>
<td>( 4\sqrt{n/4} = \sqrt{4n} )</td>
</tr>
<tr>
<td>( i = \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( i = k-1 )</td>
<td>( 2^{k-1} )</td>
<td>( \sqrt{n/2^{k-1}} )</td>
<td>( 2^{k-1} \sqrt{n/2^{k-1}} = \sqrt{2^{k-1}n} )</td>
</tr>
<tr>
<td>( i = k )</td>
<td>( 2^k )</td>
<td>3</td>
<td>( 2^k (3) )</td>
</tr>
</tbody>
</table>

Thus the total time is:

\[
T(n) = 3 \left( 2^k \right) + \sum_{i=0}^{k-1} \sqrt{2^n}
\]

\[
= 3n + \sqrt{n} \sum_{i=0}^{k-1} \left( 2^{1/2} \right)^i
\]

\[
= 3n + \sqrt{n} \left( \frac{2^{1/2} - 1}{2^{1/2} - 1} \right)
\]

\[
= 3n + \sqrt{n} \left( 2^{1/2} - 1 \right)
\]

\[
= 3n + \sqrt{n} \left( \frac{\sqrt{n} - 1}{2^{1/2} - 1} \right)
\]

\[
= 3n + \frac{1}{\sqrt{2} - 1} (n - \sqrt{n})
\]

\[
= 3n + (\sqrt{2} + 1)(n - \sqrt{n})
\]

This checks with the recurrence relation since for example the recurrence relation gives us \( T(2) = 2T(1) + \sqrt{2} = 6 + \sqrt{2} \) and this formula gives us \( T(2) = 3(2) + (\sqrt{2} + 1)(2 - \sqrt{2}) = 6 + \sqrt{2} \) and for example the recurrence relation gives us \( T(4) = 2T(2) + \sqrt{4} = 2(6 + \sqrt{2}) + 2 = 14 + 2\sqrt{2} \) and this formula gives us \( T(4) = 3(4) + (\sqrt{2} + 1)(4 - \sqrt{4}) = 14 + 2\sqrt{2} \).

\[\blacksquare\]
3 Master Theorem (Straightforward Version)

Of course we would rather not do this sort of calculation every time so we might ask if there are reliable formulas which emerge in specific situations and the answer is yes, and these are encapsulated in the Master Theorem:

**Theorem 3.0.1.** Suppose \( T(n) \) satisfies the recurrence relation:

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n)
\]

for positive constants \( a \geq 1 \) and \( b > 1 \) and where \( \frac{n}{b} \) can mean either \( \lfloor \frac{n}{b} \rfloor \) or \( \lceil \frac{n}{b} \rceil \), it doesn’t matter which. Then we have:

1. If \( f(n) = O(n^c) \) and \( \log_b a > c \) then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( f(n) = \Theta(n^c) \) and \( \log_b a = c \) then \( T(n) = \Theta(n^{\log_b a \log n}) \).
3. If \( f(n) = \Omega(n^c) \) and \( \log_b a < c \) then \( T(n) = \Theta(f(n)) \).

Note: \( f(n) \) must also satisfy a *regularity condition* which states that there is some \( C < 1 \) and \( n_0 \) such that \( af(n/b) \leq Cf(n) \) for all \( n \geq n_0 \). This regularity condition is almost always true.

Slightly fancy version of the second part:

2. If \( f(n) = \Theta(n^c \log^k n) \) for \( c = \log_b a \) then \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \).

**Proof.** Formal proof omitted. See the intuition section later, if you’re interested.

QED

**Example 3.1.** Revisiting our opening example of:

\[
T(n) = 3T(n/4) + 2n + 1 \quad \text{and} \quad T(1) = 4
\]

Here: \( \log_4 a = \log_4 3 \)

Observation 1: \( f(n) = 2n + 1 = \Theta(n^c) \) with \( c = 1 \)

Observation 2: \( \log_4 a = \log_4 3 \approx 0.79 < 1 = c \)

Summary: \( f(n) = \Omega(n^c) \) with \( c > \log_4 a \) hence Case 3

Conclusion: \( T(n) = \Theta(f(n)) = \Theta(n) \)

This aligns with what we discovered the long way. Yay!

Note that \( f(n) = 2n + 1 \) satisfies the regularity condition. To see this consider that

\[
a f(n/b) = 3f(n/4) = 3(2(n/4) + 1) = \frac{3}{2}n + 3 \leq \frac{7}{8}(2n + 1) = \frac{7}{8}f(n)
\]

for large enough \( n \).

\[\blacksquare\]
**Note 3.0.1.** When looking to apply the Master Theorem it’s always better for us to understand the Θ nature of \( f(n) \) because then we won’t presume we’re in one specific case. Then we simply compare \( c \) with \( \log_b a \) to see which case we’re in. For example if you make the mistake of initially observing that \( f(n) = O(n^c) \) then you’ll automatically think you’re in case 1.

**Note 3.0.2.** However of course don’t forget that the conditionals in 1 and 3 are also satisfied for Θ because we know that \( \Theta \Rightarrow O, \Omega \).

**Note 3.0.3.** In real-world situations this theorem arises when we recursively divide problems into sub-problems. In this case \( a \) is the number of sub-problems and \( n/b \) is the size of each sub-problem. In addition to the recursive call there are additional time requirements and those are the \( f(n) \). Thus it’s typical for \( a \) and \( b \) to be integers but this is not always true and isn’t necessary for the theorem.

**Note 3.0.4.** Some common examples include \( T(n) = kT(n/k) + \Theta(n) \) which yields \( T(n) = \Theta(n \lg n) \) and \( T(n) = kT(n/k) + \Theta(1) \) which yields \( T(n) = \Theta(n) \).

**Note 3.0.5.** It can be handy to practice this by thinking of the \( aT(n/b) \) as the prefix and the \( f(n) \) as the suffix.

**Note 3.0.6.** We write “Straightforward Version” because the theorem can be fleshed out a bit to include more nuanced hypotheses.

**Note 3.0.7.** Often the suffix in the recurrence relation is simply written in \( O, \Theta, \) or \( \Omega \) notation initially. For example we might say something like \( T(n) = 2T(n/3) + \Theta(n^2) \).

**Note 3.0.8.** Lots of problems cannot be tackled. It’s great when it applies and is useless otherwise.
Example 3.2. Consider the prefix $8T(n/2)$. Observe that $\log_2 8 = 3$. This tells us how to manage $T(n) = 8T(n/2) + f(n)$ for various $f(n)$.

- **If** $T(n) = 8T(n/2) + n^2 - n$:
  Observation 1: $f(n) = n^2 - n = \Theta(n^c)$ with $c = 2$
  Observation 2: $\log_b a = \log_2 8 = 3 > 2 = c$
  Summary: $f(n) = \mathcal{O}(n^c)$ and $c < \log_b a$ hence Case 1
  Conclusion: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

- **If** $T(n) = 8T(n/2) + n^3 + 1$:
  Observation 1: $f(n) = n^3 + 1 = \Theta(n^c)$ with $c = 3$
  Observation 2: $\log_b a = \log_2 8 = 3 = 3 = c$
  Summary: $f(n) = \Theta(n^c)$ and $c = \log_b a$ hence Case 2
  Conclusion: $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 8 \lg n}) = \Theta(n^3 \lg n)$

- **If** $T(n) = 8T(n/2) + n^4 + n$:
  Observation 1: $f(n) = n^4 + n = \Theta(n^c)$ with $c = 4$
  Observation 2: $\log_b a = \log_2 8 = 3 < 4 = c$
  Summary: $f(n) = \Omega(n^c)$ and $c > \log_b a$ hence Case 3.
  Conclusion: $T(n) = \Theta(f(n)) = \Theta(n^4 + n) = \Theta(n^4)$

Note that $f(n) = n^4 + n$ satisfies the regularity condition. To see this consider that

$$af(n/b) = 8f(n/2) = 8 \left( \frac{n}{2} \right)^4 + (n/2) \right) = \frac{1}{2} (n^4 + 4n) < \frac{3}{4} (n^4 + n) = \frac{3}{4} f(n)$$

for large enough $n$. ■
Example 3.3. Consider the prefix $2T(n/4)$. Observe that $\log_4 2 = 0.5$. This tells us how to manage $T(n) = 2T(n/4) + f(n)$ for various $f(n)$.

- If $T(n) = 2T(n/4) + n^{1/3}$:
  Observation 1: $f(n) = n^{1/3} = \Theta(n^c)$ with $c = \frac{1}{3}$
  Observation 2: $\log_b a = \log_4 2 = 0.5 > \frac{1}{3} = c$
  Summary: $f(n) = \Theta(n^c)$ with $\log_b a > c$ hence Case 1
  Conclusion: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 2}) = \Theta(n^{0.5})$

- If $T(n) = 2T(n/4) + n^{1/2}$:
  Observation 1: $f(n) = n^{1/2} = \Theta(n^c)$ with $c = \frac{1}{2}$
  Observation 2: $\log_b a = \log_4 2 = 0.5 = \frac{1}{2} = c$
  Summary: $f(n) = \Theta(n^c)$ with $\log_b a = c$ hence Case 2
  Conclusion: $T(n) = \Theta(n^{\log_b a \lg n}) = \Theta(n^{\log_4 2 \lg n}) = \Theta(n^{0.5 \lg n})$
  This is case 2.

- If $T(n) = 2T(n/4) + n \lg n$:
  Observation 1: $f(n) = n \lg n = \Omega(n^c)$ with $c = 1$
  Observation 2: $\log_b a = \log_4 2 = 0.5 < 1 = c$
  Summary: $f(n) = \Omega(n^c)$ with $\log_b a < c$ hence Case 3
  Conclusion: $T(n) = \Theta(f(n)) = \Theta(n \lg n) = \Theta(n \lg n)$
  Note that $f(n) = n \lg n$ satisfies the regularity condition because:

  \[ af(n/b) = 2f(n/4) = 2(n/4) \ln(n/4) = \frac{1}{2} n \ln(n/4) = \frac{1}{2} n \ln(n) \leq \frac{1}{2} f(n) \]

  when $n \geq n_0 = 1$.

Observe that $f(n)$ may simply be a constant. In this particular case:

- If $T(n) = 2T(n/4) + 17$:
  Observation 1: $f(n) = 17 = \Theta(n^c)$ with $c = 0$
  Observation 2: $\log_b a = \log_4 2 = 0.5 > 0 = c$
  Summary: $f(n) = \Theta(n^c)$ with $\log_b a > c$ hence Case 1
  Conclusion: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 2}) = \Theta(n^{0.5})$
Here are some examples where it does not apply:

**Example 3.4.** Suppose \( T(n) = 2T(n/4) + 3T(n/2) + n \).
The Master Theorem does not apply. Note that there is another method which often applies called the Akra-Bazzi method. It applies to recurrence formulas of the following form under certain conditions:

\[
T(n) = f(n) + \sum_{i=1}^{k} a_i T(b_i n + h_i(n))
\]

We will not cover it.

**Example 3.5.** Suppose \( T(n) = 2T(n/4) + f(n) \) and all you know is that \( f(n) = \mathcal{O}(n^2) \).
The fact that \( f(n) = \mathcal{O}(n^2) \) implies that we could only use Case 1 and insists that \( c = 2 \). However \( \log_b a = \log_4 2 = 0.5 \not< 2 = c \) and so Case 1 does not apply.

**Example 3.6.** Suppose \( T(n) = 8T(n/4) + f(n) \) and all you know is that \( f(n) = \Omega(n) \).
The fact that \( f(n) = \Omega(n) \) implies that we could only use Case 3 and insists that \( c = 1 \). However \( \log_b a = \log_4 8 = \frac{3}{2} \not< 1 = c \) and so Case 3 does not apply.
4 Intuition Behind the Theorem

4.1 Without the $f(n)$

If $f(n) = 0$ then $f(n) = O(1) = O(n^0)$ and $0 < \log_b a$ as long as $a > 1$ (which it is) and so all what follows here lies in the first case of the Master Theorem.

Consider a divide-and-conquer algorithm which breaks a problem of size $n$ into $a$ subproblems each of size $n/b$. In such a case we would have:

$$T(n) = aT(n/b)$$

Now observe:

- It seems reasonable that if $a = b$ then we have no overall gain because the number of new problems equals the reducing ratio (for example two problems of half the size doesn’t help) but we can actually say more.

If we assume a reasonable $T(1) = \alpha$ for some constant $\alpha$ then this is essentially saying, for example, that $T(2) = 2T(2/2) = 2(1) = 2\alpha$, $T(4) = 2T(4/2) = 2(2) = 2(2) = 4\alpha$, $T(8) = 2T(8/2) = 2(4) = 8\alpha$, and so on, and in general it seems reasonable that $T(n) = n\alpha = \Theta(n)$.

This also seems reasonable with any $a = b$ (not just 2), that we still get $T(n) = \Theta(n)$.

This arises in the first case of the Master Theorem because if $a = b$ then $\log_b a = 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$.

- On the other hand if $b > a$ then we have an overall decrease in time, for example if $T(n) = 2T(n/3)$ then the subproblems are $1/3$ the size and there are only two, that’s good, better than $\Theta(n)$!

This arises in the first case of the Master Theorem because if $b > a$ then $\log_b a < 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\text{less than 1}})$.

- And on the other hand if $b < a$ then we have an overall gain in time, for example if $T(n) = 3T(n/2)$ then the subproblems are $1/2$ the size but there are three, that’s bad, worse than $\Theta(n)$!

This arises in the first case of the Master Theorem because if $b < a$ then $\log_b a > 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\text{more than 1}})$. 
4.2 With the $f(n)$

Now that we’ve intuitively accepted that:

$$T(n) = aT(n/b) \implies T(n) = \Theta(n^{\log_b a})$$

Now let’s suppose there is some additional time requirement $f(n)$ for a problem of size $n$.

- If this new time requirement is at most (meaning $O$) polynomially smaller than the recursive part then the recursive part is the dominating factor. This is represented in the theorem by the line:
  
  If $f(n) = \Theta(n^c)$ for $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$.

- If this new time requirement is the same (meaning $\Theta$) polynomially as the recursive part then they combine and a logarithmic factor is introduced (this is not obvious). This is represented in the theorem by the line:
  
  If $f(n) = \Theta(n^c)$ for $c = \log_b a$ then $T(n) = \Theta(n^{\log_b a \log n})$.

- If this new time requirement is at least (meaning $\Omega$) polynomially larger than the recursive part then this new time requirement is the dominating factor. This is represented in the theorem by the line:
  
  If $f(n) = \Theta(n^c)$ for $c > \log_b a$ then $T(n) = \Theta(f(n))$.

Along with the regularity condition.

5 Even More Rudimentary Intuition

To think even more crudely, but not inaccurately, when you see the recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

Think of the $aT(n/b)$ as the tree structure and $f(n)$ as the node weights. Basically the Master Theorem is saying:

1. If the tree structure is greater than the node weights then the node weights don’t matter and the tree structure wins - winter.

2. If they balance out nicely then they combine - spring.

3. If the tree structure is less than the node weights then the tree structure doesn’t matter and the node weights win - summer.
6 Thoughts, Problems, Ideas

1. For the example \( T(n) = 3T(n/5) + (2n + 1) \) with \( T(1) = 4 \) show that calculating \( T(125) \) directly (using the recurrence relation) and using the formula developed in the notes yields the same results.

2. For the example \( T(n) = 2T(n/5) + (2n + 3) \) with \( T(1) = 4 \) draw the complete tree for \( n = 125 \) and fill in the values. What is the total time?

3. Suppose \( T(n) = 2T(n/5) + (2n + 1) \) and \( T(1) = 2 \). Emulate the example in the notes in the following sense:
   a. Calculate \( T(n) \) for a few values which are nice powers (of what?)
   b. Draw a generic version of the associated tree.
   c. Calculate the number of layers in the tree.
   d. Calculate the number of entries in each layer.
   e. Separately for each non-leaf layer calculate the total time in each entry and then add these to get the total time each non-leaf layer.
   f. Calculate the total time in the leaf-layer.
   g. Write down a sum for the total time in the tree.
   h. Simplify this sum to get the total time \( T(n) \).
   i. Use this \( T(n) \) to check your values from (a).
   j. Calculate the \( \mathcal{O}(n) \) time complexity from \( T(n) \).
   k. Calculate the time complexity from the Master Theorem.
   l. Rejoice at the beauty of equality.

4. Repeat the previous problem with \( T(n) = 2T(n/4) + \sqrt{n} \) and \( T(1) = 4 \).

5. Repeat the previous problem with \( T(n) = 3T(n/3) + 2 \) and \( T(1) = 7 \).

6. Suppose \( T(n) = 5T(n/5) + f(n) \).
   a. Apply the Master Theorem with \( f(n) = \sqrt{n} \).
   b. Apply the Master Theorem with \( f(n) = n + \sqrt{n} \).
   c. Apply the Master Theorem with \( f(n) = 3n + \sqrt{n^3} \).
   d. Apply the Master Theorem with \( f(n) = n \lg n \).

7. Suppose \( T(n) = 4T(n/8) + f(n) \).
   a. Apply the Master Theorem with \( f(n) = \sqrt{n} \).
   b. Apply the Master Theorem with \( f(n) = n^{2/3} + \lg n \).
   c. Apply the Master Theorem with \( f(n) = n \).
8. Suppose \( T(n) = 4T(n/2) + f(n) \).
   
   (a) Apply the Master Theorem with \( f(n) = n + \sqrt{n} \).
   
   (b) Apply the Master Theorem with \( f(n) = n^2 + n + 1 \).
   
   (c) Apply the Master Theorem with \( f(n) = n^3 \lg n \).

9. Binary Search has \( T(n) = T(n/2) + \Theta(1) \). What is \( T(n) \)?

10. Merge Sort has \( T(n) = 2T(n/2) + \Theta(n) \). What is \( T(n) \)?

11. The Max-Heapify routine in Heap Sort has \( T(n) \leq T(2n/3) + \Theta(1) \). What is \( T(n) \)?

12. An optimal sorted matrix search has \( T(n) = 2T(n/2) + \Theta(n) \). What is \( T(n) \)?

13. A divide and conquer algorithm which splits a list of length \( n \) into two equally sized lists, makes recursive calls to both and in addition uses constant time will have \( T(n) = 2T(n) + C \). What is \( T(n) \)?