In Chapter 7, we discussed recurrences that were used to compute a single value, such as the value of the $n^{th}$ Fibonacci number. Recurrences are also used so describe the runtimes of divide-and-conquer algorithms.

Divide-and-conquer algorithms, like merge sort from Chapter 4, are so important that special tools have been developed for creating bounds from their runtime recurrences. A great example of these tools is the “master theorem”.

Consider the family of runtime recurrences $r(n) = ar(n/b) + f(n)$ where $a$ and $b$ are constants. This corresponds to a divide-and-conquer method where each invocation calls $a$ recursions of size $n/b$, and where each invocation also performs $f(n)$ steps of actual work. For example, we saw that merge sort has a runtime recurrence $r(n) = 2r(n/2) + n$, because it reduces to two recursive calls, each with half the size, and a single pass to merge the sorted results from each half.
CHAPTER 8. MASTER THEOREM

8.1 Call trees and the master theorem

Because the problem size is divided by \( b \) in each recursion, the depth \( d \) of the call tree will clearly be

\[
d = \log_b(n) = \frac{\log_2(n)}{\log_2(b)}.
\]

Each instance will spawn \( a \) recursive calls, and so the number of leaves \( \ell \) will be

\[
a^d = a^{\log_2(n)/\log_2(b)}.
\]

We can rearrange this expression by observing

\[
\log_2(a^d) = d \log_2(a) = \frac{\log_2(n) \log_2(a)}{\log_2(b)}.
\]

By exponentiating again, we can see

\[
\ell = a^d = 2^{\frac{\log_2(n) \log_2(a)}{\log_2(b)}} = \left(2^{\log_2(n)}\right)^{\frac{\log_2(a)}{\log_2(b)}} = n^{\frac{\log_2(a)}{\log_2(b)}} = n^{\log_b(a)}.
\]

Thus we see that the number of leaves \( \ell \) in the call tree will be \( \ell = n^{\log_b(a)} \).

The master theorem revolves around three cases: the case where the leaves dominate the computation, the case where the root dominates the computation, and the case where neither dominate the other and thus every node in the call tree is asymptotically significant.

8.2 “Leaf-heavy” divide and conquer

In the case of leaf-heavy divide-and-conquer algorithms, the cost from the sheer number of leaves \( \ell \) is substantially larger than the cost of the work done at the root. For example, Strassen matrix multiplication (discussed in Chapter fixme) reduces matrix multiplication of two \( n \times n \) matrices to
7 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and $\Theta(n^2)$ postprocessing step. Here, the cost of work done at the root is $\Theta(n^2)$, but the number of leaves is $n^{\log_2(7)} \approx n^{2.807}$. Even if each leaf takes a trivial (but nonzero) amount of processing time, the number of leaves $\ell$ dwarfs the work done at the root: $n^2 \in O(n^{\log_2(7)-\epsilon})$. In general, the leaf-heavy case applies when $f(n) \in O(n^{\log_b(a)-\epsilon})$.

\[^1\]Note that $f(n) \in O(n^{c-\epsilon})$ is a stronger form of the statement $f(n) \in o(n^c)$; $f(n) \in O(n^{c-\epsilon})$ implies $f(n) \in o(n^c)$, but $f(n) \in o(n^c)$ does not necessarily imply $f(n) \in O(n^{c-\epsilon})$. For example, consider the case where $f(n) = \frac{n^2}{\log(n)}$; $f(n) \in o(n^2)$, but $f(n) \notin O(n^{2-\epsilon})$. 
When we sum up the total computation cost over all recursions, we see

\[
\begin{align*}
    r(n) &= \sum_{i=0}^{d} a^i \cdot f \left( \frac{n}{b^i} \right) \\
    &< c \sum_{i=0}^{d} a^i \cdot \left( \frac{n}{b^i} \right)^{\log_b(a) - \epsilon}, \ n \gg 1 \\
    &\propto \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a) - \epsilon}}{b^{i(\log_b(a) - \epsilon)}}, \ n \gg 1 \\
    &= \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)} \cdot n^{-\epsilon}}{(a \cdot b^{-\epsilon})^i}, \ n \gg 1 \\
    &= \sum_{i=0}^{d} a^i \cdot b^{\epsilon i} \cdot n^{\log_b(a)} \cdot n^{-\epsilon}, \ n \gg 1 \\
    &= n^{\log_b(a)} \cdot n^{-\epsilon} \sum_{i=0}^{d} (b^\epsilon)^i, \ n \gg 1 \\
    &= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^\epsilon \left( \frac{b^\epsilon}{{b^\epsilon} - 1} \right) - 1, \ n \gg 1 \\
    &= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^\epsilon \left( \frac{b^{\log_b(n)^\epsilon} - 1}{{b^\epsilon} - 1} \right), \ n \gg 1 \\
    &= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^\epsilon \left( \frac{b^{\log_b(n)^\epsilon} - 1}{{b^\epsilon} - 1} \right), \ n \gg 1 \\
    &= n^{\log_b(a)} \cdot b^\epsilon \left( \frac{1 - n^{-\epsilon}}{{b^\epsilon} - 1} \right), \ n \gg 1 \\
    &\propto n^{\log_b(a)}, \ n \gg 1,
\end{align*}
\]

where \( c \) is chosen as an arbitrary constant to satisfy the definition of \( O(\cdot) \) and all proportionality constants are nonnegative. Thus, the runtime is \( r(n) \in O \left( n^{\log_b(a)} \right) \). Furthermore, because we visit each of the \( n^{\log_b(a)} \) leaves, the
8.3. “ROOT-HEAVY” DIVIDE AND CONQUER

runtime must be \( r(n) \in \Omega(n^{\log_b(a)}) \); therefore, the runtime of the leaf-heavy case is \( r(n) \in \Theta(n^{\log_b(a)}) \).

8.3 “Root-heavy” divide and conquer

In root-heavy divide and conquer, the cost of visiting the root dwarfs the number of leaves. For example, a recurrence of the form \( r(n) = 2r\left(\frac{n}{2}\right) + n^2 \) will result in \( n^{\log_2(2)} = n \) leaves but visiting the root alone will cost \( n^2 \). The cost of visiting the root is \( n^2 \in \Omega(n^2) \), and so the total cost (which includes the visit to the root) must be \( \in \Omega(n^2) \).

More generally, when \( f(n) \in \Omega(n^{\log_b(a)+\epsilon}) \) the total cost \( r(n) \in \Omega(f(n)) \); however, unlike the leaf-heavy case, this does not yield a corresponding \( O(\cdot) \) upper bound. For this reason, to make good use of the root-heavy case, we add an additional constraint known as the “regularity condition”:

\[
a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), \quad n \gg 1,
\]

where \( c < 1 \). Under this condition, we see that for \( n \gg 1 \),

\[
a^i \cdot f\left(\frac{n}{b^i}\right) \leq c \cdot a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \leq \cdots \leq c^i \cdot f(n).
\]

This creates an upper bound on the total runtime:

\[
r(n) = \sum_{i=0}^{d} a^i \cdot f\left(\frac{n}{b^i}\right)
\leq \sum_{i=0}^{d} c^i \cdot f(n), \quad n \gg 1
= f(n) \sum_{i=0}^{d} c^i, \quad n \gg 1
< f(n) \frac{1}{1-c}, \quad n \gg 1
\propto f(n), \quad n \gg 1,
\]

where the proportionality constants are nonnegative. Thus, under the regularity condition of the root-heavy case, we see that \( r(n) \) is bounded above \( r(n) \in O(f(n)) \). Because we have already shown \( r(n) \in \Omega(f(n)) \) for the root-heavy case, we see that \( r(n) \in \Theta(f(n)) \).
8.4 “Root-leaf-balanced” divide and conquer

In the balanced case, neither the root nor the leaves dominate the runtime. Intuitively, we can see that in this case the next level of the tree $i+1$ must not take substantially more (where substantially more means $\in \Omega(n^{\epsilon})$) nor substantially less (where substantially less means $\in O(n^{-\epsilon})$) runtime than level $i$ in the tree. One way to achieve this is to have $f(n) \in \Theta(n^{\log_b(a)})$; however, we can be more general than this and permit $f(n) \in \Theta(n^{\log_b(a)}(\log(n))^k)$, because $k$ logarithmic terms are not enough to overcome a $\pm \epsilon$ change to the polynomial power and thus will cause neither a leaf-heavy nor a root-heavy outcome as defined above.

In this case, the work done in each level of the tree is not dramatically different. Thus, we can use the $f(n)$ work applied at the root and estimate that the other levels of the tree, of which there are $\log(n) + 1$, will be comparable. Thus we achieve a runtime of $r(n) \in \Theta(f(n) \log(n)) = \Theta(n^{\log_b(a)}(\log(n))^{k+1})$.

---

2We have not shown this as rigorously as above, but it can be verified by massaging the same sum over tree levels that defines $r(n)$. The two keys are as follows: First, the polynomial terms form a partition where the polynomial runtime in each layer is the same as the polynomial runtime of the next layer. Second, the $k$ logarithmic terms only depend on the tree level $i$ through a denominator in the logarithm; these logarithms of ratios are converted to differences of logarithms, and the terms with the $i$ are asymptotically irrelevant, leaving behind only the original $(\log(n))^k$ term, which is factored out of the sum, and that sum computes the same result for each layer and thus has an additional $\log(n)$ multiplied in (because it is the number of layers).