Recurrence Relations

G.1 WHY YOU SHOULD CARE

Recurrence relations, sometimes called recurrences, are a way of analyzing recursive algorithms to compute their run-time. Since recursion, particularly divide-and-conquer, comes up quite often in algorithm design, knowing how to solve them is important for computer scientists.

G.2 MERGE SORT

Divide and conquer is a recursive algorithmic technique in which the instance of the problem being solved is broken down into two or more smaller versions of the original problem and the answer to these helps compute the answer to the original problem. An important example of this is the Merge Sort algorithm.

The Merge Sort algorithm sorts a list by splitting the list into two close-to-equal halves, sorting both halves recursively, and then merging the halves together. (Since the length of the list could be odd, one of the halves might be one larger than the other, hence the "close-to-equal.") In the base case, when the list has length one, we don't have to do anything as a list of size one is already sorted. We won't delve into the details of the algorithm here and will instead just analyze its run-time.

Let the run-time of Merge Sort on a list of size n be denoted by $T(n)$ for all $n \geq 1$. From the above description, we can see that the running time of the entire algorithm will be the time to sort two half-sized lists and the time to merge these lists. The time to sort each half-sized list can be denoted by $T(n/2)$ based on the definition of T. For simplicity, we'll assume that n is always a power of two to avoid complications of unequal halves. The time to merge the lists together turns out to be linear in n and, for simplicity, we just denote it as cn for some constant $c > 0$. Lastly, the base case of Merge Sort is when the list has size one, in which case we simply return the list with zero additional work done because a list of size one is already sorted. Putting all this together, we get the following definition of T, called a **recurrence relation** or **recurrence**:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

We'll see next a few different ways to solve such a recurrence relation.

G.3 RECURSION TREE METHOD

One way to solve the above recurrence relation, called the recursion tree method, is to use the formula for T and expand it a few times like this:

$$\begin{aligned}
T(m) &= 2T(m/2) + cm && \text{(using the formula for } T \text{ when } n = m) \\
&= 2\,(2T(m/4) + cm/2) + cm && \text{(using the formula for } T \text{ when } n = m/2) \\
&= 4T(m/4) + 2cm && \text{(simplifying with algebra)} \\
&= 4\,(2T(m/8) + cm/4) + 2cm && \text{(using the formula for } T \text{ when } n = m/4) \\
&= 8T(m/8) + 3cm && \text{(simplifying with algebra).}
\end{aligned}$$

We can continue this process a few more times to see that this seems to be following the pattern (after expanding the formula k times):

$$T(m) = 2^k T(m/2^k) + kcm.$$

We can continue the above expansion until such time as we have reduced the problem down to size one. This happens precisely when $m/2^k = 1$ or when $m = 2^k$ or $k = \log_2(m)$. Substituting this value of k into the above formula, we get

$$\begin{aligned}
T(m) &= 2^{\log_2(m)} T(m/2^{\log_2(m)}) + cm\log_2(m) \\
&= mT(m/m) + cm\log_2(m) && (\text{as } 2^{\log_2(m)} = m) \\
&= mT(1) + cm\log_2(m) \\
&= cm\log_2(m) && (\text{as } T(1) = 0).
\end{aligned}$$

We can conclude from this derivation that the run-time of Merge Sort is then $cm\log_2(m)$, more simply expressed (using Big-O notation, see Appendix D) as $O(n\log(n))$.

G.4 A REVIEW OF SOME LOG RULES

As you just saw in the previous section, you will need to be comfortable working with logarithms in this chapter. Here are a few rules that are worth remembering. For any $b > 1$ and $n, m > 0$, we have

$$\begin{aligned}
\log_b(1) &= 0 \\
\log_b(b) &= 1 \\
b^{\log_b(n)} &= n \\
\log_b(nm) &= \log_b(n) + \log_b(m) \\
\log_b(n/m) &= \log_b(n) - \log_b(m) \\
\log_b(n^m) &= m\log_b(n).
\end{aligned}$$

G.5 SUBSTITUTION METHOD

Another, simpler, way to prove the recurrence can be to use strong induction. For this method, we need to know what the formula is beforehand. We can write the following induction hypothesis and prove it inductively.

For any $n \geq 1$,

$$P(n) : T(n) = cn \log_2(n).$$

For the base case, we have that when $n = 1$, we know that $T(1) = 0$ and $cn \log_2(n) = 0$ since $\log_2(1) = 0$. Thus $P(1)$ holds.

We fix some $k > 1$ and assume that the induction hypothesis holds for all values $n < k$. We will show that the induction hypothesis holds for $n = k$ as well.

When $n = k$, we have

$$\begin{aligned}
T(k) &= 2T(k/2) + ck && \text{(by the definition of } T\text{)} \\
&= 2(c(k/2)\log_2(k/2)) + ck && \text{(by the induction hypothesis when } n = k/2\text{)} \\
&= ck \log_2(k/2) + ck && \text{(simplifying with algebra)} \\
&= ck \log_2(k) - ck + ck && \text{(as } \log_2(k/2) = \log_2(k) - \log_2(2) = \log_2(k) - 1\text{)} \\
&= ck \log_2(k),
\end{aligned}$$

thus proving the inductive step. We can wrap up the strong induction proof as usual.

G.6 ANALYZING THE KARATSUBA-OFMAN ALGORITHM

There is another divide and conquer algorithm for multiplying large numbers called the Karatsuba-Ofman algorithm with the following recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + cn & \text{if } n > 1. \end{cases}$$

We'll see how to analyze it using the above two techniques next. Using the recursion tree method, we get:

$$\begin{aligned}
T(m) &= 3T(m/2) + cm && \text{(formula for } T \text{ when } n = m\text{)} \\
&= 3\,(3T(m/4) + cm/2) + cm && \text{(formula for } T \text{ when } n = m/2\text{)} \\
&= 9T(m/4) + (3/2)cm + cm && \text{(simplifying with algebra)} \\
&= 9\,(3T(m/8) + cm/4) + (3/2)cm + cm && \text{(formula for } T \text{ when } n = m/4\text{)} \\
&= 27T(m/8) + (3/2)^2 cm + (3/2)cm + cm && \text{(simplifying with algebra).}
\end{aligned}$$

We can continue this procedure a few more times to see that this seems to be following the pattern (after expanding the formula k times):

$$T(m) = 3^k T(m/2^k) + \sum_{i=0}^{k-1} (3/2)^i cm.$$

We can continue this until we have reduced the problem down to size one. This happens precisely when $m/2^k = 1$ or when $m = 2^k$ or $k = \log_2(m)$. Substituting this value of k into the above formula, we get

$$\begin{aligned}
T(m) = \;& 3^{\log_2(m)}T(m/2^{\log_2(m)}) + \sum_{i=0}^{\log_2(m)-1}(3/2)^i cm \\
= \;& 3^{\log_2(m)}T(m/m) + \sum_{i=0}^{\log_2(m)-1}(3/2)^i cm && (\text{as } 2^{\log_2(m)} = m) \\
= \;& 3^{\log_2(m)}T(1) + \sum_{i=0}^{\log_2(m)-1}(3/2)^i cm \\
= \;& \sum_{i=0}^{\log_2(m)-1}(3/2)^i cm && (\text{as } T(1) = 0) \\
= \;& \frac{(3/2)^{\log_2(m)} - 1}{3/2 - 1}cm && (\text{formula for geometric sum}) \\
= \;& 2cm(3^{\log_2(m)}/2^{\log_2(m)} - 1) \\
= \;& 2cm(3^{\log_2(m)}/m - 1) && (\text{as } 2^{\log_2(m)} = m) \\
= \;& 2cm^{\log_2 3} - 2cm && (\text{logarithm rules}).
\end{aligned}$$

We can conclude from this derivation that the run-time of the algorithm is thus $O(n^{\log_2 3})$ or $O(n^{1.585})$.

The substitution method for this proof proceeds as follows. For any $n \geq 1$,

$$P(n) : T(n) = 2cn^{\log_2 3} - 2cn.$$

For the base case, we have that when $n = 1$, we know that $T(1) = 0$ and $2cn^{\log_2 3} - 2cn = 2c - 2c = 0$. Thus $P(1)$ holds.

We fix some $k > 1$ and assume that the induction hypothesis holds for all values $n < k$. We will show that the induction hypothesis holds for $n = k$ as well.

When $n = k$, we have

$$\begin{aligned}
T(k) = \;& 3T(k/2) + ck && (\text{by the definition of } T) \\
= \;& 3(2c(k/2)^{\log_2 3} - 2ck/2) + ck && (\text{by the induction hypothesis when } n = k/2) \\
= \;& 6ck^{\log_2 3}/2^{\log_2 3} - 3ck + ck && (\text{algebra}) \\
= \;& 2ck^{\log_2(3)} - 2ck && (\text{as } 2^{\log_2(3)} = 3),
\end{aligned}$$

thus showing the inductive step. We can again wrap up the strong induction proof as usual.

G.7 CHAPTER SUMMARY AND KEY CONCEPTS

- **Recurrences** are recursive formulas, usually used to express the running time of a recursive algorithm.

- The **recursion tree method** can be used to unroll a recurrence to get an exact formula for it.
- The **substitution method** uses strong induction to formally prove the formula for the recurrence.

EXERCISES

Solve the following recurrences using the recursion tree method. In each case, assume that $T(1) = 0$.

G.1 $T(n) = 2T(n/4) + n$

G.2 $T(n) = 2T(n/2) + n^2$

G.3 $T(n) = 7T(n/8) + n$

G.4 $T(n) = 3T(n/4) + n^2$

G.5 $T(n) = 5T(n/8) + n^3$

Solve the following recurrences using the substitution method (strong induction). In each case, assume that $T(1) = 0$. (Hint: Use the recursion tree method first to get an induction hypothesis in each case.)

G.6 $T(n) = 2T(n/4) + n$

G.7 $T(n) = 2T(n/2) + n^2$

G.8 $T(n) = 7T(n/8) + n$

G.9 $T(n) = 3T(n/4) + n^2$

G.10 $T(n) = 5T(n/8) + n^3$