

Recurrence Relations

Last Updated: August 16th, 2025

Introduction

Determining the running time of a recursive algorithm often requires one to determine the big-O growth of a function $T(n)$ that is defined in terms of a *recurrence relation*. Recall that a **recurrence relation** for a sequence of numbers is simply an equation that provides the value of the n th number in terms of an algebraic expression involving n and one or more of the previous numbers of the sequence. The most common recurrence relation we will encounter in this course is the **uniform divide-and-conquer recurrence relation**, or **uniform recurrence** for short.

Uniform Divide-and-Conquer Recurrence Relation: one of the form

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

where $a > 0$ and $b > 1$ are integer constants. This equation is explained as follows.

$T(n)$: $T(n)$ denotes the number of steps required by some divide-and-conquer algorithm $\mathcal{A}$ on a problem instance having size n .

Divide $\mathcal{A}$ divides original problem instance into a subproblem instances.

Conquer Each subproblem instance has size n/b and hence is solved (conquered) in $T(n/b)$ steps by making a recursive call to $\mathcal{A}$.

Combine $f(n)$ represents the number of steps needed to both divide the original problem instance and combine the a solutions into a final solution for the original problem instance.

For example,

$$T(n) = 7T(n/2) + n^2,$$

is a uniform divide-and-conquer recurrence with $a = 7$, $b = 2$, and $f(n) = n^2$.

Mergesort

The **Mergesort** algorithm is a divide-and-conquer algorithm for sorting an array of size n of comparable elements. The algorithm begins by checking if input array a has $n \leq 2$ elements. If so, then a is sorted in place by making at most one swap. Otherwise, a is divided into two (almost) equal halves a_{left} and a_{right} . Both of these subarrays are sorted by making recursive calls to Mergesort. Once sorted, a **merge** operation merges the elements of a_{left} and a_{right} into an auxiliary array. This sorted auxiliary array is then copied over to the original array. Merging requires $\Theta(n)$ steps. Therefore, the uniform divide-and-conquer recurrence that represents the number of steps $T(n)$ required by the **Mergesort** algorithm is

$$T(n) = 2T(n/2) + n,$$

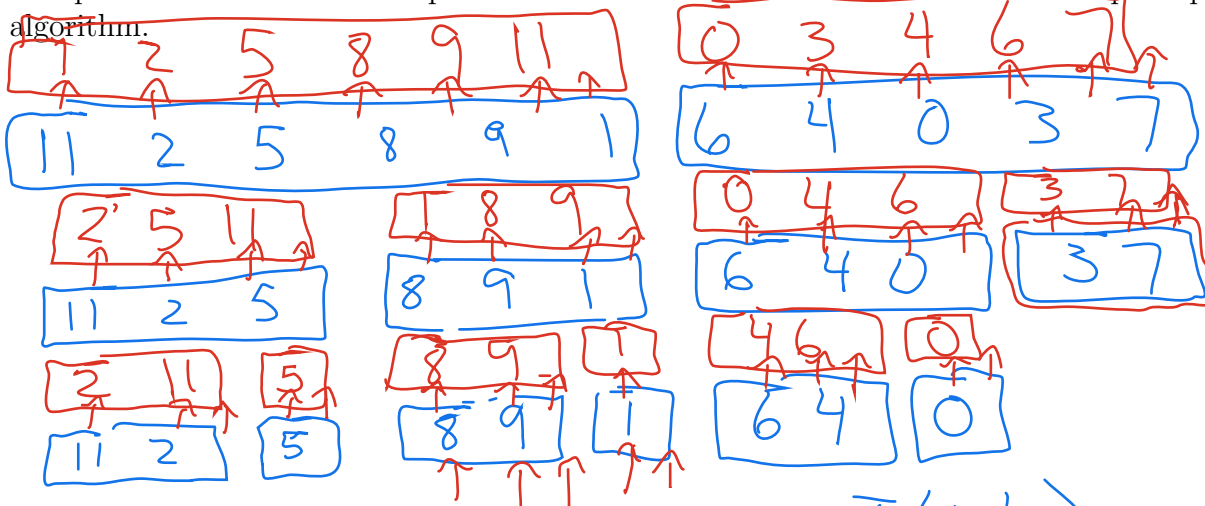
and we'll see in this lecture that it yields a growth of $T(n) = \Theta(n \log n)$.

Example 1. Demonstrate the ~~Mergesort~~ divide and conquer algorithm on the array

$a = 11, 2, 5, 8, 9, 1, 6, 4, 0, 3, 7$

0 1 2 3 4 5 6 7 8 9 10

and provide a divide-and-conquer recurrence that describes the number of steps required by the algorithm.



$$T(n) = 2T(n/2) + n$$

It should be emphasized that not every divide-and-conquer algorithm produces a uniform divide-and-conquer recurrence. For example, the *Median-of-Five Find Statistic* algorithm described in the next lecture produces the recurrence

$$T(n) = T(n/5) + T(7n/10) + an,$$

where $a > 0$ is a constant. We refer to such recurrences as **non-uniform divide-and-conquer recurrences**. They are non-uniform in the sense that the created subproblem instances may not all have the same size.

The complexity analysis of a divide-and-conquer algorithm often reduces to determining the big-O growth of a solution $T(n)$ to a divide-and-conquer recurrence. In this lecture we examine two different ways of solving such recurrences, which are summarized as follows.

Master Theorem The **Master Theorem** provides a solution $T(n)$ to a uniform recurrence, provided a , b , and $f(n)$ satisfy certain conditions.

Substitution Method The **substitution method** uses mathematical induction to prove that some candidate function $T(n)$ is a solution to a given divide-and-conquer recurrence. The candidate solution is usually obtained by making an educated guess, or by analyzing the associated recursion tree.

The Master theorem and substitution method represent *proof methods*, meaning that the application of either method will yield a growth for $T(n)$ that is beyond doubt.

Recursion Trees and The Master Theorem

6

Master Theorem. Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ a function, and $T(n)$ be defined on the nonnegative integers by

$$T(n) = aT(n/b) + f(n). \quad (1)$$

Then the growth of $T(n)$ can be asymptotically determined under the following assumptions.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$.
4. If $f(n) = \Theta(n^{\log_b a} \log_b^r n)$, then $T(n) = \Theta(n^{\log_b a} \log_b^{r+1} n)$

$n^{\log_b a}$

$r \geq 1$

We prove a relaxed version of the Master theorem by assuming that n is always a power of b , so that the resulting recursion tree associated with the recurrence in Equation 1 is a perfect a -ary tree. The more general case when n is not a power of b follows from this theorem with additional work involving the analysis of the insignificant effects that floors and ceilings have on the growth of $T(n)$.

Lemma 1. Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b . Define $T(n)$ on exact powers of b by the recurrence relation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

for some positive integer i . Then

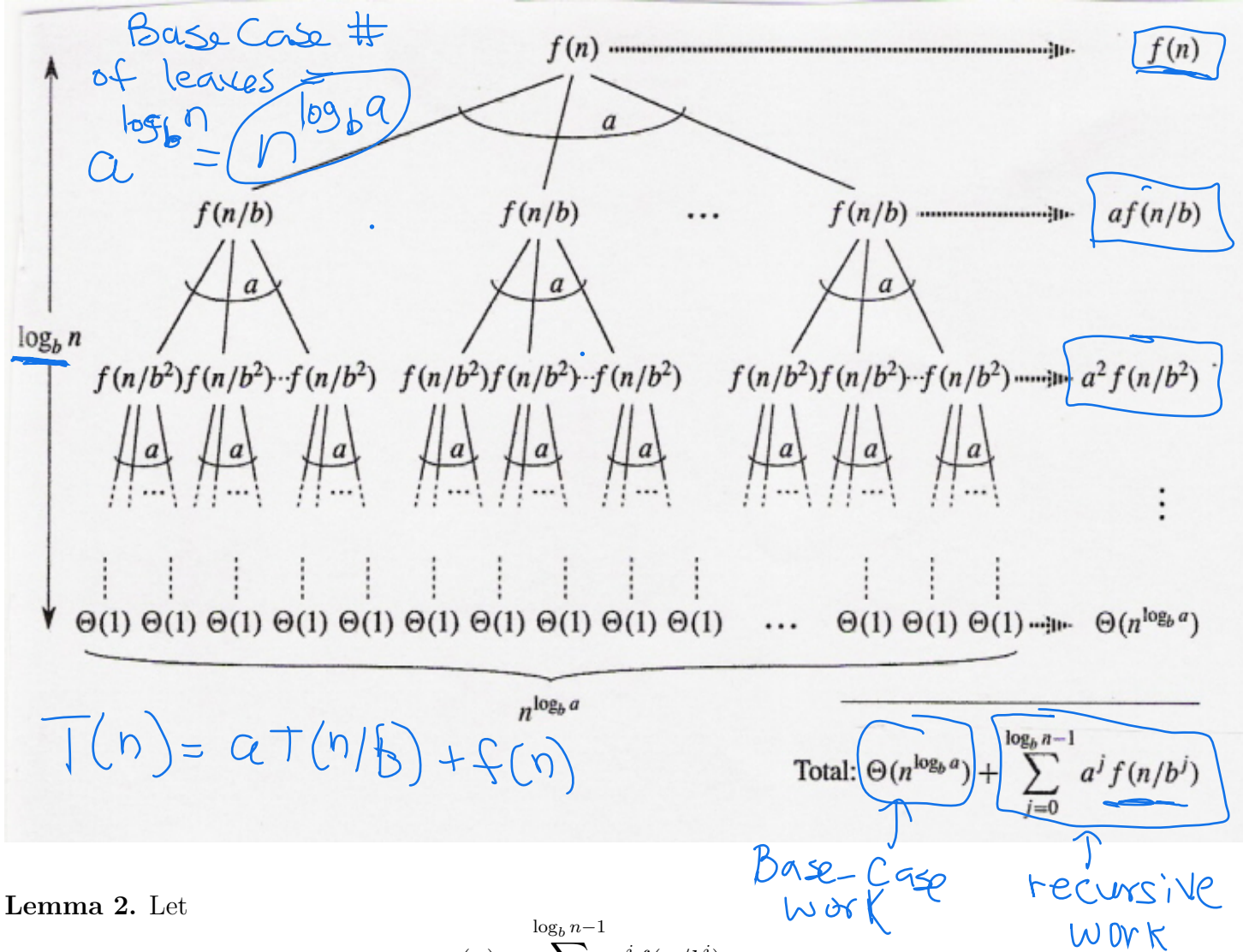
$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j).$$

Proof of Lemma 1. Notice that, since n is a power of b , there are $\log_b n + 1$ levels of recursion: $0, 1, \dots, \log_b n$. Moreover, level j , $0 \leq j \leq \log_b n - 1$ contributes a total of $a^j f(n/b^j)$ to the total value of $T(n)$, while level $\log_b n$ contributes $\Theta(1) \cdot a^{\log_b n} = \Theta(n^{\log_b a})$ (see the general recursion-tree in Figure 1 below). Hence,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j).$$

QED

Figure 1: The general recursion tree for uniform divide-and-conquer recurrences



Lemma 2. Let

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j),$$

where a, b , and $f(n)$ are defined as in Lemma 1. Then

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $g(n) = \Theta(n^{\log_b a} \cdot \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant positive $c < 1$, then $g(n) = \Theta(f(n))$.
4. If $f(n) = \Theta(n^{\log_b a} \log_b^r n)$, then $g(n) = \Theta(n^{\log_b a} \log_b^{r+1} n)$.

The proof of Lemmma 2 makes use of the geometric series formula

$$\sum_{j=0}^{k-1} r^j = \frac{r^k - 1}{r - 1},$$

as well as the formula

$$b^{j \log_b a} = b^{\log_b a^j} = a^j.$$

Proof of Lemma 2.

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. Then there exists a constant $C > 0$ such that, for sufficiently large n ,

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \leq \sum_{j=0}^{\log_b n - 1} a^j C \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon} = C \sum_{j=0}^{\log_b n - 1} a^j \frac{n^{\log_b a} n^{-\epsilon}}{b^{\log_b a^j} b^{-j\epsilon}} = \\ &= \frac{C \cdot n^{\log_b a}}{n^\epsilon} \sum_{j=0}^{\log_b n - 1} a^j \frac{(b^\epsilon)^j}{a^j} = \frac{C \cdot n^{\log_b a}}{n^\epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j = \frac{C \cdot n^{\log_b a}}{(b^\epsilon - 1)} \frac{n^\epsilon - 1}{n^\epsilon} = \\ &= \frac{C \cdot n^{\log_b a}}{(b^\epsilon - 1)} \left(1 - \frac{1}{n^\epsilon}\right) = O(n^{\log_b a}). \end{aligned}$$

Case 2: $f(n) = \Theta(n^{\log_b a})$. Then there are constant $C_1 > 0$ for which

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \geq \sum_{j=0}^{\log_b n - 1} a^j C_1 \left(\frac{n}{b^j}\right)^{\log_b a} = C_1 \sum_{j=0}^{\log_b n - 1} a^j \frac{n^{\log_b a}}{b^{\log_b a^j}} = \\ &= C_1 \cdot n^{\log_b a} \sum_{j=0}^{\log_b n - 1} a^j \frac{1}{a^j} = C_1 \cdot n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 = C_1 \cdot n^{\log_b a} \log_b n = \Omega(n^{\log_b a} \log_b n). \end{aligned}$$

Similarly, we may show that $g(n) = O(n^{\log_b a} \log_b n)$. Therefore, $g(n) = \Theta(n^{\log_b a} \log_b n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some positive constant $c < 1$. Since $f(n)$ is equal to the first term ($j = 0$) of the sum that adds to $g(n)$, and all terms are nonnegative, we have that $g(n) = \Omega(f(n))$. Also, we may use induction to prove that, for all $j \geq 0$,

$$a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n).$$

This gives

$$g(n) \leq f(n) \sum_{j=0}^{\log_b n - 1} c^j \leq \left(\frac{1 - c^{\log_b n}}{1 - c}\right) f(n)$$

which implies $g(n) = O(f(n))$. Therefore, $g(n) = \Theta(f(n))$.

Case 4: $f(n) = \Theta(n^{\log_b a} \log_b^r n)$. Then there is a constant C_1 for which

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \geq \sum_{j=0}^{\log_b n - 1} a^j C_1 \left(\frac{n}{b^j}\right)^{\log_b a} \log_b^r \left(\frac{n}{b^j}\right) = C_1 \sum_{j=0}^{\log_b n - 1} a^j \frac{n^{\log_b a}}{b^{\log_b a j}} (\log_b n - j)^r = \\
C_1 \cdot n^{\log_b a} \sum_{j=0}^{\log_b n - 1} a^j \frac{1}{a^j} (\log_b n - j)^r &= C_1 \cdot n^{\log_b a} \sum_{j=0}^{\log_b n - 1} (\log_b n - j)^r = C_1 \cdot n^{\log_b a} (\log_b^r n + (\log_b n - 1)^r + \dots + 1) \geq \\
C_1 \cdot n^{\log_b a} \cdot C \cdot \log_b^{r+1} n &= \Omega(n^{\log_b a} \cdot \log_b^{r+1} n).
\end{aligned}$$

Similarly, we may show that

$$g(n) = O(n^{\log_b a} \cdot \log_b^{r+1} n).$$

Completing the Proof of the Master Theorem

Lemma 1 tells us that $T(n) = \Theta(n^{\log_b a}) + g(n)$. Thus, by Lemma 2, we have the following for each of the four cases of the Master Theorem.

Case 1. $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$.

Case 2. $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$.

Case 3. $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$, since in this case we assume $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Case 4. $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log^{r+1} n) = \Theta(n^{\log_b a} \log^{r+1} n)$.

Example 2. Determine the order of growth of $T(n)$ for the following recurrences.

1. $T(n) = 16T(n/4) + n$

2. $T(n) = T(n/5) + 20$

3. $T(n) = 3T(n/4) + n \log n$

4. $T(n) = 2T(n/2) + n \log n$

1. $f(n) = n$ $n^{\log_b a} = n^2$

$f(n) = O(n^{2-\epsilon})$ when $\epsilon \leq 1$.

$\therefore$ By case 1 of M.T., $T(n) = \Theta(n^2)$

2. $f(n) = 20$ $n^{\log_b a} = n^{\log_5 1} = n^0 = 1$

$\therefore$ By case 2 of M.T. $f(n) = \Theta(1) \Rightarrow$
 $T(n) = \Theta(1 \cdot \log n) = \Theta(\log n)$

3. $f(n) = n \log n$ $n^{\log_b a} = n^{\log_4 3}$

$n^{\log n} = \Omega(n^{\log_4 3 + \epsilon})$ where $\epsilon \leq 1 - \log_4 3$

$\therefore T(n) = \Theta(n \log n)$ By case 3 of M.T.

4. $f(n) = n \log n$ $n^{\log_b a} = n^{\log_2 2} = n^1$

$\therefore$ By case 4 of M.T. $T(n) = \Theta(n \log^2 n)$

Substitution Method

The substitution method is an inductive method for proving the big-O growth of a function $T(n)$ that satisfies some divide-and-conquer recurrence. It requires that we already have a candidate function $g(n)$ for representing the growth of $T(n)$. For example, suppose we desire to show that $T(n) = O(g(n))$. Then we perform the following two steps.

$$T(n) \leq Cg(n)$$

Inductive Assumption Assume that $T(k) \leq Cg(k)$, for all $k < n$ and for some constant $C > 0$.

Prove Inductive Step Show that $T(n) \leq Cg(n)$. Do this by replacing any term $aT(n/b)$ of the recurrence with $aCg(n/b)$, and show that the resulting non-recurrence expression is bounded above by $Cg(n)$.

Notice that there is no basis step to the above proof method. This is because we only care about the growth of $T(n)$ for sufficiently large n .

$$\log \frac{n}{2} = \log n - \log 2$$

Example 3. Show that if $T(n) = 2T(n/2) + 3n$, then $T(n) = O(n \log n)$.

Inductive assumption: Assume $T(k) \leq Ck \log k$ for all $k < n$ and some $C > 0$ (constant)

$$\begin{aligned} T(n) &= 2T(\underbrace{n/2}_K) + 3n \leq 2 \left(C \left(\frac{n}{2} \right) \log \left(\frac{n}{2} \right) \right) + 3n \\ &= cn(\log n - 1) + 3n = \cancel{cn \log n} - cn + 3n \\ &\leq \cancel{cn \log n} \Leftrightarrow cn \geq 3n \Leftrightarrow \\ &\quad \boxed{C \geq 3} \end{aligned}$$

Example 4. Similarly, show that $T(n) = \Omega(n \log n)$, where $T(n)$ satisfies the recurrence from Example 3.

Exercise

Example 5. Show that if $T(n) = 2T(n/2) + n \log n$, then $T(n) = O(n \log^2 n)$.

Assume $T(K) \leq CK \log^2 K$ for all $K \leq n$
and some const. $C > 0$.

$$T(n) = 2T(\underbrace{n/2}_K) + n \log n \leq 2C\left(\frac{n}{2}\right) \log^2 \frac{n}{2} + n \log n$$

$$\stackrel{=}{=} Cn(\log n - 1)^2 + n \log n = Cn \log^2 n - 2Cn \log n + Cn$$

$$+ n \log n \leq Cn \log^2 n \iff$$

$$2Cn \log n - Cn \geq n \log n \iff$$

$$C(2n \log n - n) \geq n \log n \iff$$

$$C \geq \frac{n \log n}{2n \log n - n} = \frac{1}{2 - \frac{1}{\log n}}$$

$C > \frac{1}{2}$ for n sufficiently large, but always $> \frac{1}{2}$

Sometimes it is necessary to add one or more lesser-degree terms to the inductive assumption when the assumption fails to provide a term that counterbalances the $f(n)$ term of the recurrence. This happens in cases where $f(n)$ is little-o of the original inductive assumption. For example, instead of $T(k) \leq Ck^2$, one could instead use $T(k) \leq Ck^2 + Dk$ in case $f(n)$ is a linear term, $T(k) \leq Ck^2 + D$ in case $f(n)$ is a constant term, or $T(k) \leq Ck^2 + D \log k$ in case $f(n)$ is $\Theta(\log n)$, etc..

In all three of the above scenarios, it's important to note that D can be either positive or negative. Thus, after algebraic manipulation, a final inequality such as $D \leq -9$ is valid. This is true since, e.g., $T(n) = Cn^2 - 9 \log n = \Theta(n^2)$.

Example 6. Prove that if $T(n) = 2T(n/2) + 7$ then $T(n) = O(n)$.

Assume: $T(k) \leq Ck$ for all $k < n$
and some const. $C > 0$.

Show $T(n) \leq Cn$

$$T(n) = 2T(\underbrace{n/2}_k) + 7 \leq 2C\frac{n}{2} + 7 =$$

$$Cn + 7 \leq Cn \Leftrightarrow 7 \leq 0 \text{ which is false.}$$

New Assumption: $T(k) \leq Ck + D$ for all
 $k < n$ and const. $C > 0$ and integer D .

$$T(n) = 2T(\frac{n}{2}) + 7 \leq 2\left(C\frac{n}{2} + D\right) + 7 =$$

$$Cn + 2D + 7 \leq Cn + D \Leftrightarrow$$
$$D \leq -7$$

Core Exercises for Recurrences

1. Use the Master Theorem to give tight asymptotic bounds for the following recurrences.
 - a. $T(n) = 2T(n/4) + 1$
 - b. $T(n) = 2T(n/4) + \sqrt{n}$
 - c. $T(n) = 2T(n/4) + n$
 - d. $T(n) = 2T(n/4) + n^2$
2. Use the Master Theorem to show that the solution to the binary-search recurrence $T(n) = T(n/2) + a$, where $a > 0$ is a constant, is $T(n) = \Theta(\log n)$.
3. Use the Master Theorem to determine the big-O growth of $T(n)$ if it satisfies the recurrence $T(n) = 3T(n/2) + n$.
4. Use the Master Theorem to determine the big-O growth of $T(n)$ if it satisfies the recurrence $T(n) = 4T(n/2) + n^2 \log n$.
5. Use the substitution method to prove that $T(n) = 2T(n/2) + an$, $a > 0$ a constant, has solution $T(n) = \Theta(n \log n)$.
6. Use the substitution method to prove that $T(n) = 4T(n/2) + n$ has solution $T(n) = O(n^2)$.
7. Use the substitution method to prove that $T(n) = T(an) + T(bn) + n$ has solution $T(n) = O(n)$, where a and b are positive constants, with $a + b < 1$.
8. Use the substitution method to prove that $T(n) = 8T(n/2) - \log n$ has solution $T(n) = \Omega(n^3)$.
9. Use the substitution method to prove that $T(n) = 2T(n/4) + \sqrt{n} \log n$ has solution $T(n) = \Omega(\sqrt{n} \log^2 n)$.

Solutions to Core Exercises for Recurrences

1. Use the Master Theorem to give tight asymptotic bounds for the following recurrences.

- a. Case 1: $T(n) = \Theta(\sqrt{n})$
- b. Case 2: $T(n) = \Theta(\sqrt{n} \log n)$
- c. Case 3: $T(n) = \Theta(n)$
- d. Case 3: $T(n) = \Theta(n^2)$

2. Use Case 2 of the Master Theorem: $T(n) = \Theta(\log n)$.

3. Use case 1. $T(n) = \Theta(n^{\log 3})$.

4. Use Case 4 of the Master Theorem: $T(n) = \Theta(n^2 \log^2 n)$.

5. Assume $T(k) \leq ck \log k$, for all $k < n$. Then

$$T(n) = 2T(n/2) + an \leq 2c(n/2) \log(n/2) + an = cn \log n - cn + an \leq cn \log n$$

iff $c \geq a$. Therefore by induction, $T(n) = O(n \log n)$. $T(n) = \Omega(n \log n)$ is proved similarly.

6. The hypothesis $T(k) \leq Ck^2$, for all $k < n$ is insufficient, since it leads to the inequality $n \leq 0$. Using $T(k) \leq Ck^2 + Dk$ yields

$$\begin{aligned} T(n) &= 4T(n/2) + n \leq 4(C(n/2)^2 + Dn/2) + n = Cn^2 + 2Dn + n \leq Cn^2 + Dn \Leftrightarrow \\ &Dn + n \leq 0 \Leftrightarrow D \leq -1. \end{aligned}$$

Therefore, the inequality is established so long as we choose $D \leq -1$, and $C > 0$.

7. Assume $T(k) \leq Ck$, for all $k < n$ and some constant $C > 0$. Then

$$\begin{aligned} T(n) &= T(an) + T(bn) + n \leq Can + Cbn + n = Cn(a + b) + n \leq Cn \Leftrightarrow \\ &Cn(1 - a - b) \geq n \Leftrightarrow C \geq 1/(1 - a - b). \end{aligned}$$

Since $a + b < 1$, we have $1 - a - b > 0$, and thus the inequality holds, so long as $C \geq 1/(1 - a - b)$.

8. The hypothesis $T(k) \geq Ck^3$, for all $k < n$ is insufficient, since it leads to the inequality $\log n \leq 0$ which is false for $n \geq 2$. However, using $T(k) \geq Ck^3 + D \log k$ yields

$$\begin{aligned} T(n) &= 8T(n/2) - \log n \geq 8(C(n/2)^3 + D \log(n/2)) - \log n = \\ &Cn^3 + 8D \log n - 8D - \log n \geq Cn^3 + D \log n \Leftrightarrow 7D \log n - 8D \geq \log n \Leftrightarrow \\ &D(7 - 8/\log n) \geq 1 \Leftrightarrow D \geq 2/7 \end{aligned}$$

and n is sufficiently large.

9. Assume $T(k) \geq C\sqrt{k} \log^2 k$ for all $k < n$. Then

$$\begin{aligned} T(n) &= 2T(n/4) + \sqrt{n} \log n \geq 2(C\sqrt{n/4} \log^2(n/4)) + \sqrt{n} \log n = \\ &C\sqrt{n}(\log n - 2)^2 + \sqrt{n} \log n = C\sqrt{n} \log^2 n - 2C\sqrt{n} \log n + 4C\sqrt{n} + \sqrt{n} \log n \geq C\sqrt{n} \log^2 n \Leftrightarrow \\ &2C\sqrt{n} \log n - 4C\sqrt{n} \leq \sqrt{n} \log n \Leftrightarrow C(2 - 4/\log n) \leq 1 \Leftrightarrow C \leq 1/2 \end{aligned}$$

and n is sufficiently large.

Additional Exercises

- A. Draw the recursion tree that results when applying Mergesort to the array 5, -2, 0, 7, 3, 11, 2, 9, 5, 6. Label each node with the sub-problem to be solved at that point of the recursion. Assume arrays of size 1 and 2 are base cases. Assume that odd-sized arrays are split so that the left subproblem has one more integer than the right. Next to each node, write the solution to its associated subproblem.
- B. Write an algorithm for the function **Merge** that takes as input two sorted integer arrays a and b of sizes m and n respectively, and returns a sorted integer array of size $m + n$ whose elements are the union of elements from a and b .
- C. Solve the recurrence $T(n) = 2T(\sqrt{n}) + \log n$. Hint: Let $S(k) = T(2^k)$ and write a divide-and-conquer recurrence for $S(k)$. 
- D. Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + an$, where $a > 0$ is a constant, is $T(n) = \Theta(n \log n)$, by using an appropriate recursion tree. Then verify it using the substitution method.
- E. Use a recursion tree to show that the solution of $T(n) = T(n-1) + n$ is $T(n) = O(n^2)$.
- F. Use a recursion tree to solve the recurrence $T(n) = T(n-2) + n^2$; providing a tight upper and lower bound.
- G. Use a recursion tree to estimate the big-O growth of $T(n)$ which satisfies the recurrence $T(n) = 2T(n-1) + 1$.
- H. Suppose $T(n)$ and $g(n)$ are positive functions (meaning they output positive values for each input $n \geq 0$) that satisfy $T(n) \leq Cg(n)$ for all $n \geq k$, and some constant $C > 0$. Prove that There is a constant $C' > 0$ for which $T(n) \leq C'g(n)$, for all $n \geq 0$.
- I. Suppose $f(n) = n^s \log^r n$, where $s > \log_b a$ and $r \geq 0$. Show that there exists a positive $c < 1$ for which

$$af(n/b) < cf(n),$$

where we may assume that n is an arbitrary power of b .

Solutions to Additional Exercises

- A. We use *lr*-strings for the addresses of each node. For example, λ denotes the root, while *lrr* denotes the right child of the right child of the left child of the root. Then

$$\lambda : 5, -2, 0, 7, 3, 11, 2, 9, 5, 6$$

$$l : 5, -2, 0, 7, 3 \quad r : 11, 2, 9, 5, 6$$

$$ll : 5, -2, 0 \quad lr : 7, 3 \quad rl : 11, 2, 9 \quad rr : 5, 6$$

$$lll : 5, -2 \quad llr : 0 \quad rll : 11, 2 \quad rlr : 9$$

- B. //Merges a and b into merged. n= |a|, m = |b|
- ```
void merge(int[] a, int n, int[] b, int m, int[] merged)
{
 int i=0;
 int j=0;
 int value_a = a[0];
 int value_b = b[0];
 int count = 0;

 while(true)
 {
 if(value_a <= value_b)
 {
 merged[count++] = value_a;
 i++;

 if(i < n)
 value_a = a[i];
 else
 break;
 }
 else
 {
 merged[count++] = value_b;
 j++;

 if(j < m)
 value_b = b[j];
 else
 break;
 }
 }

 //copy remaining values
 if(j < m)
 for(; j < m; j++)
```

```

 merged[count++] = b[j];
 else
 for(; i < n; i++)
 merged[count++] = a[i];
}

```

C. Letting  $S(k) = T(2^k)$ , we then have the divide-and-conquer recurrence

$$S(k) = T(2^k) = 2T((2^k)^{1/2}) + \log 2^k = 2T(2^{k/2}) + k = 2S(k/2) + k,$$

which, by the Master Theorem, yields  $S(k) = \Theta(k \log k)$ . Finally, letting  $n = 2^k$ , we have  $T(n) = \Theta(\log n \log(\log n))$ .

D. The recursion tree consists of a single branch of length  $n$ , where the work done at depth  $i$  is  $n - i$ . Hence,  $T(n) = O(\sum_{i=0}^n (n - i)) = O(n^2)$ .

E. The recursion tree consists of a single branch of length  $n/2$ , where the work done at depth  $i$  is  $(n - 2i)^2$ . Hence,

$$T(n) = \Theta\left(\sum_{i=0}^{n/2} (n - 2i)^2\right) = \Theta(n^3).$$

F. The recursion tree is a perfect binary tree having depth  $n$ . Thus, it has  $2^{n+1} - 1$  nodes. Moreover, since the work at each node is always 1, we see that  $T(n) = \Theta(2^n)$ .

G. The recursion tree remains perfect all the way down to depth  $\lfloor \log_3 n \rfloor$ . Moreover, for each depth  $i = 0, 1, \dots, \lfloor \log_3 n \rfloor$ , the work done at that depth always adds to  $n$ . Hence, the total work (i.e.  $T(n)$ ) must be  $\Omega(n \log n)$ . Also, the longest branch has a depth not exceeding  $\lfloor \log_{\frac{3}{2}} n \rfloor$ . Moreover, the amount of work at each depth does not exceed  $n$ . Hence  $T(n) = O(n \log n)$ . Therefore,  $T(n) = \Theta(n \log n)$ .

H. Let  $C_i$ ,  $i = 0, 1, \dots, k-1$ , be a constant for which  $T(i) \leq C_i g(i)$ . Since  $f(i)$  and  $g(i)$  are both positive numbers, we know that such a  $C_i$  exists. Then choose  $C' = \max(C_0, \dots, C_{k-1}, C)$ . Then  $T(n) \leq C' g(n)$ , for all  $n \geq 0$ .

I. We have

$$a \left(\frac{n}{b}\right)^s \log \left(\frac{n}{b}\right)^r = a \left(\frac{n}{b}\right)^s (\log n - \log b)^r < cn^s \log^r n \Leftrightarrow$$

$$\frac{a}{b^s} \left(\frac{\log n - \log b}{\log n}\right)^r = \frac{a}{b^s} \left(1 - \frac{\log b}{\log n}\right)^r \leq \frac{a}{b^s} < c,$$

and the last inequality is true since  $s > \log_b a$  implies that  $b^s > a$  and so  $\frac{a}{b^s} < 1$ . Therefore, there does exist a constant  $c < 1$  for which  $\frac{a}{b^s} < c$  is true.