someone picks a number between 1 and 100 and allows you to ask questions of the form "Is the number greater than k?" or "Is the number equal to k?" where k is an integer you choose. Your goal is to ask as few questions as possible to get a "yes" to a question of the form "Is the number equal to k?" Why should your first question be, "Is the number greater than 50?" After asking if the number is bigger than 50, you have learned either that the number is between 1 and 50 or that the number is between 51 and 100. In either case, you have reduced your problem to one in which the range is only half as big. Thus, you have *divided* the problem into a problem that is only half as big, and you can now (recursively) *conquer* this remaining problem. (If you ask any other question, the size of one of the possible ranges of values you could end up with would be more than half the size of the original problem.) If you continue in this fashion, always cutting the problem size in half, you will reduce the problem size to 1 fairly quickly, and then you will know what the number is. Of course, if we started with a number in the range from 1 to 128, it would be easier to cut the problem size exactly in half each time, but the question doesn't sound quite so plausible then. Thus, to analyze the problem, we will assume someone asks you to figure out a number between 0 and n, where n is a power of 2.

Exercise 4.3-1 Let $T(n)$ be the number of questions in a binary search on the range of numbers between 1 and n. Assuming that n is a power of 2, give a recurrence for $T(n)$.

For Exercise 4.3-1, we get

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \geq 2 \\ 1 & \text{if } n = 1. \end{cases} \tag{4.17}$$

That is, the number of questions needed to carry out binary search on n items is equal to one step (the first question) plus the time to perform binary search on the remaining $n/2$ items. Note that the base case is $T(1) = 1$ because we have to ask a question of the form "Is the number k?" when we have reduced the range of possible values to 1.

What we are really interested in is how much time it takes to use binary search in a computer program that looks for an item in an ordered list. While the number of questions gives us a feel for the amount of time, processing each question may take several steps in our computer program. The exact amount of time these steps take might depend on some factors over which we have little control, such as where portions of the list are stored. Also, we may have to deal with lists with lengths that are not a

power of 2. Thus, a more realistic description of the maximum time needed would be

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + C_1 & \text{if } n \geq 2, \\ C_2 & \text{if } n = 1, \end{cases} \tag{4.18}$$

where C_1 and C_2 are constants.

Note that $\lceil x \rceil$ stands for the smallest integer larger than or equal to x, whereas $\lfloor x \rfloor$ stands for the largest integer less than or equal to x. It turns out that the solution to Recurrences 4.17 and 4.18 are roughly the same, in a sense that should become clear later. For now, let's not worry about floors and ceilings and the distinction between things that take one unit of time and things that take no more than some constant amount of time.

Instead, let's turn to **merge sort**, another example of a divide-and-conquer algorithm. In this algorithm, we wish to sort a list of n items. Assume that the data are stored in Positions 1 through n of an array A and that n is a power of 2. If the list has only one element, we don't need to do anything to sort it. Otherwise, to sort the list, we divide A into the portions from 1 to $n/2$ and from $n/2 + 1$ to n. We recursively sort the first half, we recursively sort the second half, and then we merge the two sorted "half lists" into one sorted list. (We saw examples of one way to merge two lists in the beginning of Section 3.1.) Merge sort can be described in pseudocode as follows:

```
MergeSort(A,low,high)

// This algorithm sorts the portion of list A from
// location low to location high.
if (low == high)
     return
else
     mid = ⌊(low + high)/2⌋
     MergeSort(A,low,mid)
     MergeSort(A,mid+1,high)
     Merge the sorted lists from the previous two steps
     return
```

More details on merge sort can be found in almost any algorithms textbook. The base case `(low == high)` takes one step. The other case executes one step, makes two recursive calls on problems of size $n/2$, and then executes the merge instruction, which can be done in n steps.

Thus, we obtain the following recurrence for the running time of merge sort:

$$T(n) = \begin{cases} 2T(n/2) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases} \tag{4.19}$$

Recurrences such as this one can be understood via the idea of a recursion tree, which we introduce next. This concept allows us to analyze recurrences that arise in divide-and-conquer algorithms, as well as those that arise in other recursive situations, such as the Tower of Hanoi.

Recursion Trees

A recursion tree for a recurrence is a visual and conceptual representation of the process of iterating the recurrence. We use several examples to introduce the idea of a recursion tree. To understand recursion trees, it is helpful to have an "algorithmic" interpretation of a recurrence. For example, ignoring for a moment the base case, we can interpret the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{4.20}$$

as, "To solve a problem of size n, we must solve two problems of size $n/2$ and do n units of additional work." Similarly, we can interpret

$$T(n) = T\left(\frac{n}{4}\right) + n^2$$

as, "To solve a problem of size n, we must solve one problem of size $n/4$ and do n^2 units of additional work." We can also interpret the recurrence

$$T(n) = 3T(n-1) + n$$

as, "To solve a problem of size n, we must solve three subproblems of size $n-1$ and do n additional units of work."

In Figure 4.4, we draw the beginning of the recursion tree diagram for Recurrence 4.20. For now, assume n is a power of 2. We draw the diagram in levels, each level representing a level of recursion. Equivalently, each level of the diagram represents a level of iteration of the recurrence. A level of a recursion tree diagram has five parts: two on the left, one in the middle, and two on the right. On the left, we keep track of the problem size and the number of problems; in the middle, we draw the tree; and on the right, we keep track of the work done per problem and the total

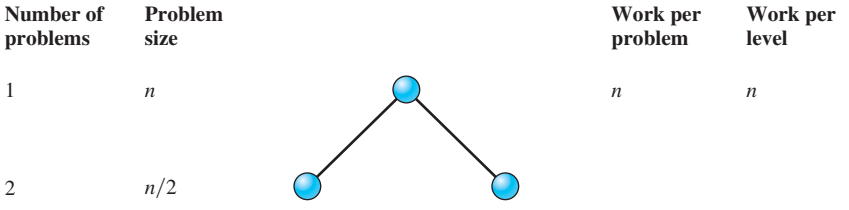

Figure 4.4: The initial stage of drawing a recursion tree diagram

amount of work done on the current level. So, to begin the recursion tree diagram for Recurrence 4.20, we show, in Level 0 on the left, that we have one problem of size n. Then, by drawing a root vertex with two edges leaving it, we show in the middle that we are splitting our problem into two problems. We note on the right that we do n units of work in addition to whatever is done on the two new problems we created. Because there is only one problem on this level, the total work done on this level is n units of work. In the next level, we draw two vertices in the middle, representing the two problems into which we split our main problem, and we show on the left that we have two problems of size $n/2$.

Notice how the recurrence is reflected in Levels 0 and 1 of the recursion tree. The top vertex of the tree represents $T(n)$. On the next level, we have two problems of size $n/2$, representing the recursive term $2T(n/2)$ of our recurrence. After we solve these two problems, we return to Level 0 of the tree and do n additional units of work for the nonrecursive term of the recurrence.

Now we continue to draw the tree in the same manner. Filling in the rest of Level 1 (which is the second level because the first is Level 0) and adding a few more levels, we get Figure 4.5.

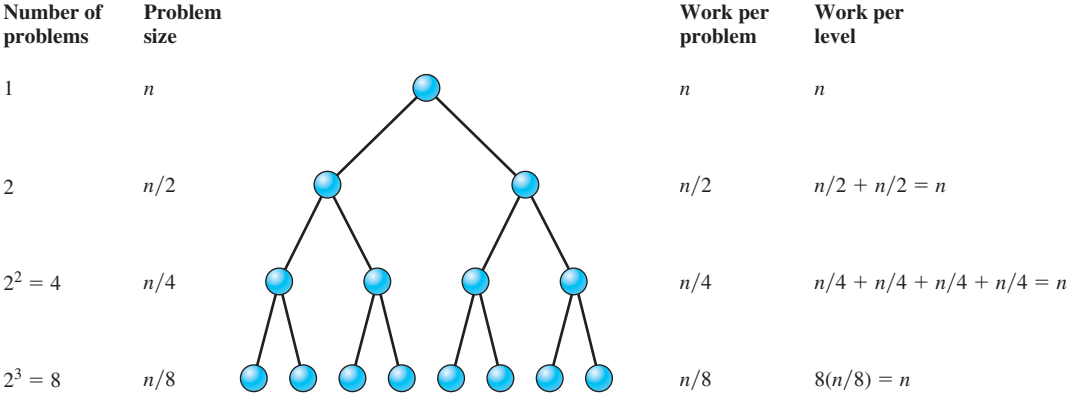

Figure 4.5: Four levels of a recursion tree diagram

Let us summarize what the diagram tells us so far. At Level 0 (the top level), n units of work are done. We see that at each succeeding level, we halve the problem size and double the number of subproblems. We also see that at Level 1, each of the two subproblems requires $n/2$ units of additional work; thus, a total of n units of additional work are done. Similarly, Level 2 has four subproblems of size $n/4$; thus, $4(n/4) = n$ units of additional work are done. Notice that to compute the total work done on a level, we add the amount of work done on each subproblem. When the problems all have the same size, as they do here, this is equivalent to multiplying the number of subproblems by the amount of additional work per subproblem.

To see how iteration of the recurrence is reflected in the diagram, we iterate the recurrence once to obtain

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\ &= 4T\left(\frac{n}{4}\right) + n + n \\ &= 4T\left(\frac{n}{4}\right) + 2n. \end{aligned}$$

If we examine Levels 0, 1, and 2 in Figure 4.5, we see that at Level 2 we have four vertices, which represent four problems, each of size $n/4$. This corresponds to the recursive term that we obtained after iterating the recurrence. However, after we solve these problems, we return to Level 1, where we do $n/2$ additional units of work twice, and to Level 0, where we do another n additional units of work. In this way, each time we add a level to the tree, we are showing the result of one more iteration of the recurrence.

We now have enough information to describe the recursion tree diagram in general. To do this, we need to determine four things for each level:

- the number of subproblems
- the size of each subproblem
- the amount of work done per subproblem
- the total work done at that level

Once we know, for each level, the total work done at that level, we can sum over all levels to obtain the total overall work. For this purpose, we also need to figure out how many levels there are in the recursion tree.

We see that for this problem, at Level i, we have 2^i subproblems of size $n/2^i$. Furthermore, because a problem of size 2^i requires 2^i units of additional work, there are $(2^i)\left(n/(2^i)\right) = n$ units of work done per level. To figure

out how many levels there are in the tree, we notice that at each level, the problem size is cut in half, and the tree stops when the problem size is 1. Therefore, there are $\log_2 n + 1$ levels of the tree, because we start with the top level and cut the problem size in half $\log_2 n$ times.[7] We can thus visualize the whole tree in Figure 4.6.

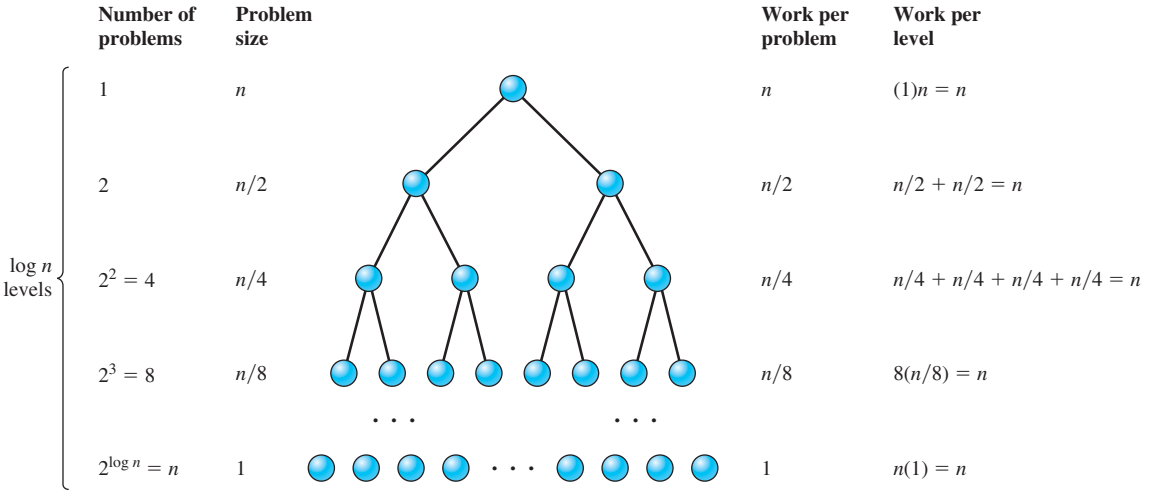

Figure 4.6: A finished recursion tree diagram

The computation of the work done at the bottom level is different from the other levels. In the other levels, the work is described by the recursive equation of the recurrence. At the bottom level, the work comes from the base case. Thus, we must compute the number of problems of size 1 (for this recurrence, the base case is $n = 1$) and then multiply this value by $T(1) = 1$. In the recursion tree in Figure 4.6, the number of nodes at the bottom level is $2^{\log_2 n} = n$. Because $T(1) = 1$, we do n units of work at the bottom level of the tree. But if we had chosen to say that $T(1)$ was some constant c other than 1, the work done at the bottom level would have been cn. We emphasize that the correct value of work per problem at the bottom level always comes from the base case.

The bottom level of the tree represents the final stage of iterating the recurrence. We have seen that at this level, we have n problems, each requiring work $T(1) = 1$, giving us total work n for the level. After we solve the problems represented by the bottom level, we have to do all the additional work from all the earlier levels. For this reason, we sum the work done at

[7] To simplify notation for the remainder of the book, if we omit the base of a logarithm, it should be assumed to be base 2.

all the levels of the tree to get the total work done. *Iteration of the recurrence shows that the solution to the recurrence is the sum of all the work done at all the levels of the recursion tree.*

The important thing is that we now know how much work is done at each level. Once we know this, we can sum the total amount of work done over all the levels, giving us the solution to our recurrence. In this case, there are $\log_2 n + 1$ levels; at each level, the amount of work we do is n units. Thus, we conclude that the total amount of work done to solve the problem described by Recurrence 4.20 is $n(\log_2 n + 1)$.

Because one unit of time will vary from computer to computer, and because some kinds of work might take longer than other kinds, we are usually interested in the big Θ behavior of $T(n)$. For example, we can consider a recurrence that is identical to Recurrence 4.19, except that $T(1) = a$ for some constant a. In this case, $T(n) = an + n \log n$, because an units of work are done at Level 1, and n additional units of work are done at each of the remaining $\log n$ levels. It is still true that $T(n) = \Theta(n \log n)$, because the different base case did not change the solution to the recurrence by more than a constant factor.[8] Although recursion trees can give the exact solutions (such as $T(n) = an + n \log n$) to recurrences, our interest in the big Θ behavior of solutions will usually lead us to use a recursion tree to determine the big Θ or, in complicated cases, the big O behavior of the actual solution to the recurrence. Problem 18 explores whether the value of $T(1)$ actually influences the big Θ behavior of the solution to a recurrence that arises from a divide-and-conquer algorithm.

Let's look at one more recurrence:

$$T(n) = \begin{cases} T(n/2) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases} \tag{4.21}$$

Again, assume n is a power of 2. We can interpret this as follows: To solve a problem of size n, we must solve one problem of size $n/2$ and do n units of additional work. Figure 4.7 shows the recursion tree diagram for this problem. We see that the problem sizes are the same as in the previous tree. The remainder, however, is different. The number of subproblems does not double; rather, it remains at 1 on each level. Consequently, the amount of work halves at each level. Note that there are still $\log n + 1$ levels, because the number of levels is determined by how the problem size changes, not by how many subproblems there are. So, on Level i, we have one problem of size $n/2^i$, for total work of $n/2^i$ units.

[8] More precisely, $n \log n < an + n \log n < (a + 1)n \log n$ for any $a > 0$.

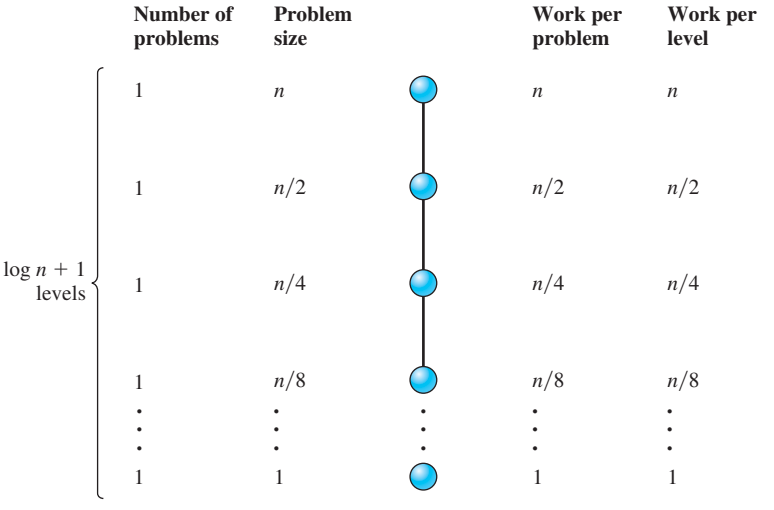

Figure 4.7: A recursion tree diagram for Recurrence 4.21

We now wish to compute how much work is done in solving a problem that gives this recurrence. Note that the additional work done is different on each level, so we have that the total amount of work is

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 = n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \left(\frac{1}{2}\right)^{\log n}\right),$$

which is n times a geometric series. By Theorem 4.4, the value of a geometric series in which the largest term is 1 is $\Theta(1)$. This implies that the work done is described by $T(n) = \Theta(n)$.

We emphasize that there is exactly one solution to Recurrence 4.21; it is the one we get by using the recurrence to compute $T(2)$ from $T(1)$, then to compute $T(4)$ from $T(2)$, and so on. Here, we have shown that $T(n) = \Theta(n)$. In fact, for the kinds of recurrences we have been examining, once we know $T(1)$, we can compute $T(n)$ for any relevant n by repeatedly using the recurrence. Thus, there is no question that solutions do exist and can, in principle, be computed for any value of n. In most applications, we are not interested in the exact form of the solution; rather, we are interested in a big O upper bound or a big Θ bound on the solution.

Exercise 4.3-2 Use a recursion tree to find a big Θ bound for the solution to the recurrence

$$T(n) = \begin{cases} 3T(n/3) + n & \text{if } n \geq 3, \\ 1 & \text{if } n < 3. \end{cases}$$

Assume that n is a power of 3.

Exercise 4.3-3 Use a recursion tree to solve the recurrence

$$T(n) = \begin{cases} 4T(n/2) + n & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

Assume that n is a power of 2. Convert your solution to a big Θ statement about the behavior of the solution.

Exercise 4.3-4 Can you give a general big Θ bound for solutions to recurrences of the form $T(n) = aT(n/2) + n$ when n is a power of 2? You may have different answers for different values of a.

The recurrence in Exercise 4.3-2 is similar to the merge sort recurrence. One difference is that at each step, we divide into three problems of size $n/3$ rather than two problems of size $n/2$. Thus, we get the picture in Figure 4.8. Another difference is that the number of levels, instead of being $\log_2 n + 1$, is now $\log_3 n + 1$, so that the total work is still $\Theta(n \log n)$ units. (Note that $\log_b n = \Theta(\log_2 n)$ for any $b > 1$.)

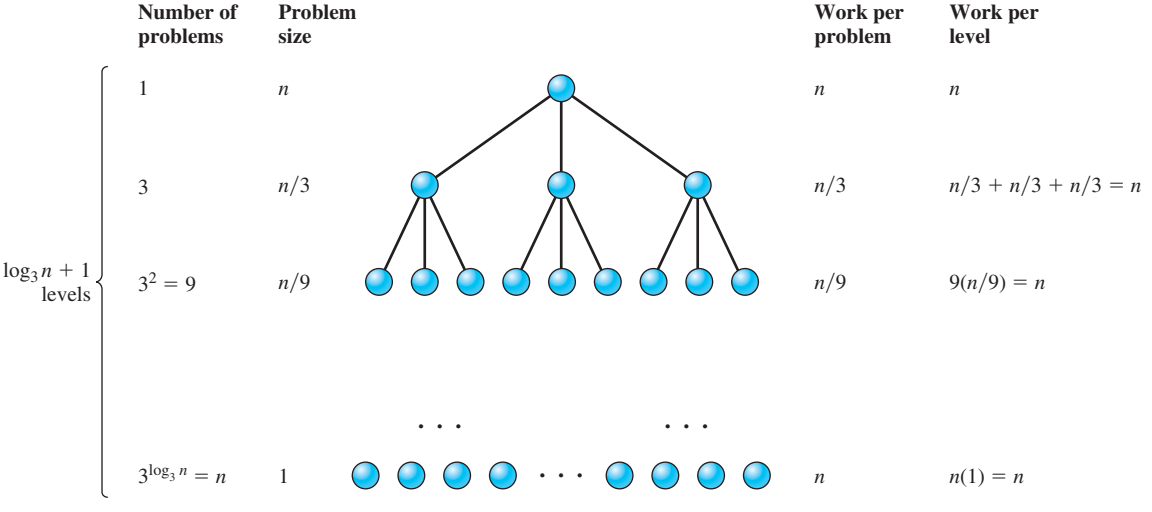

Figure 4.8: The recursion tree diagram for the recurrence in Exercise 4.3-2

Now let's look at the recursion tree for Exercise 4.3-3. A node of size n has four children of size $n/2$, and we get Figure 4.9. Just as in the merge sort tree, there are $\log_2 n + 1$ levels. However, as we pointed out, each node has four children. Thus, Level 0 has 1 node, Level 1 has 4 nodes, Level 2 has 16 nodes, and, in general, Level i has 4^i nodes. On Level i, each node

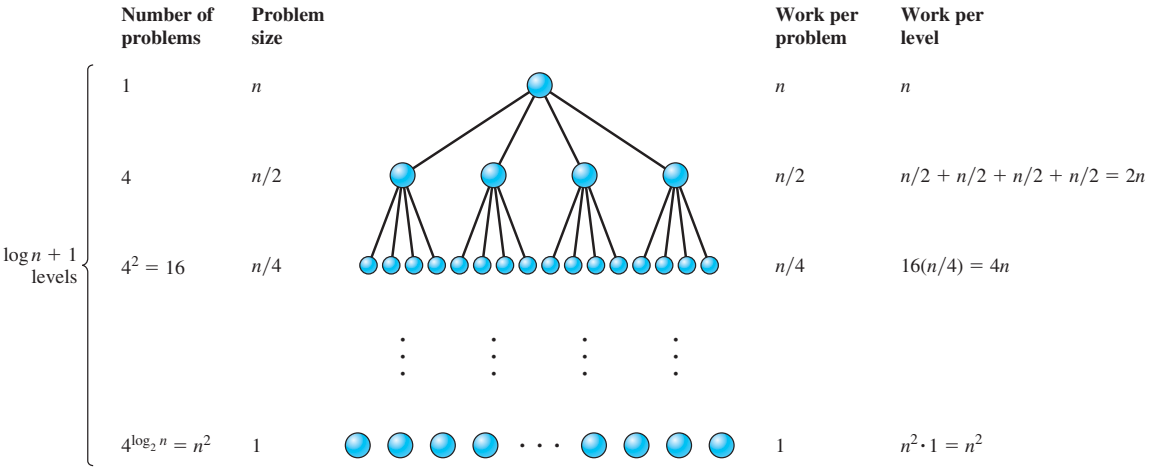

Figure 4.9: The Recursion tree for Exercise 4.3-3

corresponds to a problem of size $n/2^i$ and, hence, requires $n/2^i$ units of additional work. Thus, the total work on Level i is $4^i(n/2^i) = 2^i n$ units. This formula also applies on Level $\log_2 n$ (the bottom level), because there are $4^{\log n} = (2^2)^{\log n} = 2^{2\log n} = (2^{\log n})^2 = n^2 = 2^{\log n} n$ nodes, each requiring $T(1) = 1$ work. Summing over the levels, we get

$$\sum_{i=0}^{\log n} 2^i n = n \sum_{i=0}^{\log n} 2^i.$$

There are many ways to simplify this expression. For example, from our formula for the sum of a geometric series, we get

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log n} 2^i \\ &= n \frac{1 - 2^{(\log n)+1}}{1 - 2} \\ &= n \frac{1 - 2n}{-1} \\ &= 2n^2 - n \\ &= \Theta(n^2). \end{aligned}$$

More simply, by Theorem 4.4, we have that $T(n) = n \cdot \Theta(2^{\log n}) = \Theta(n^2)$.

Three Different Behaviors

Let's compare the recursion tree diagrams for the recurrences $T(n) = 2T(n/2) + n$, $T(n) = T(n/2) + n$, and $T(n) = 4T(n/2) + n$. Note that all three trees have depth $1 + \log_2 n$, as this is determined by the size of the subproblems relative to the parent problem, and that in each case, the size of each subproblem is half the size of the parent problem. The trees differ, however, in the amount of work done per level. For the first recurrence, the amount of work on each level is the same. In the second, the amount of work done on a level decreases as we go down the tree, with the most work being at the top level. In fact, it decreases geometrically; by Theorem 4.4, the total work done is bounded above and below by a constant multiplied by the work done at the root node. In the third recurrence, the number of nodes per level is growing at a faster rate than the problem size is decreasing, and the level with the largest amount of work is the bottom one. Again, we have a geometric series; and so, by Theorem 4.4, the total work is bounded above and below by a constant multiplied by the amount of work done at the last level.

If you understand these three cases and the differences among them, then you understand the great majority of the recursion trees that arise in algorithms.

So, to answer Exercise 4.3-4, which asks for a general big Θ bound for the solutions to recurrences of the form $T(n) = aT(n/2) + n$, we can conclude the following:

Lemma 4.7

Suppose that we have a recurrence of the form

$$T(n) = aT\left(\frac{n}{2}\right) + n,$$

where a is a positive integer and $T(1)$ is nonnegative. Then we have the following big Θ bounds on the solution:

1. If $a < 2$, then $T(n) = \Theta(n)$.
2. If $a = 2$, then $T(n) = \Theta(n \log n)$.
3. If $a > 2$, then $T(n) = \Theta(n^{\log_2 a})$.

Proof Cases 1 and 2 follow immediately from our earlier observations. We can verify Case 3 as follows: At Level i, we have a^i nodes, each corresponding to a problem of size $n/2^i$. Thus, at Level i, the total amount of work is $a^i(n/2^i) = n(a/2)^i$ units. Summing over the $\log_2 n$ levels, we obtain

$$a^{\log_2 n} T(1) + n \sum_{i=0}^{(\log n)-1} \left(\frac{a}{2}\right)^i .$$