Chapter 5

Recursion and Recurrences

5.1 Growth Rates of Solutions to Recurrences

Divide and Conquer Algorithms

One of the most basic and powerful algorithmic techniques is *divide and conquer*. Consider, for example, the binary search algorithm, which we will describe in the context of guessing a number between 1 and 100. Suppose someone picks a number between 1 and 100, and allows you to ask questions of the form "Is the number greater than k?" where k is an integer you choose. Your goal is to ask as few questions as possible to figure out the number. Your first question should be "Is the number greater than 50?" Why is this? Well, after asking if the number is bigger than 50, you have learned either that the number is between one and 50, or that the number is between 51 and 100. In either case have reduced your problem to one in which the range is only half as big. Thus you have *divided* the problem up into a problem that is only half as big, and you can now (recursively) *conquer* this remaining problem. (If you ask any other question, one of the possible ranges of values you could end up with would more than half the size of the original problem.) If you continue in this fashion, always cutting the problem size in half, you will be able to get the problem down to size one fairly quickly, and then you will know what the number is. Of course it would be easier to cut the problem exactly in half each time if we started with a number in the range from one to 128, 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 5.1-1 Let $T(n)$ be number of questions in binary search on the range of numbers between 1 and n. Assuming that n is a power of 2, get a recurrence of for $T(n)$.

For Exercise 5.1-1 we get:

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases} \tag{5.1}$$

That is, the number of guesses to carry out binary search on n items is equal to 1 step (the guess) plus the time to solve binary search on the remaining $n/2$ items.

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 we have little control over, such as where portions of the list are stored. Also, we may have to deal with lists whose length is not a power of two. Thus a more realistic description of the time needed would be

$$T(n) \leq \begin{cases} T(\lceil n/2 \rceil) + C_1 & \text{if } n \geq 2 \\ C_2 & \text{if } n = 1, \end{cases} \tag{5.2}$$

where C_1 and C_2 are constants.

It turns out that the solution to (5.1) and (5.2) are roughly the same, in a sense that will hopefully become clear later. This is almost always the case; we will come back to this issue. For now, let us not worry about floors and ceilings and the distinction between things that take 1 unit of time and things that take no more than some constant amount of time.

Let's turn to another example of a divide and conquer algorithm, *mergesort*. In this algorithm, you wish to sort a list of n items. Let us assume that the data is stored in an array A in positions 1 through n. Mergesort can be described as follows:

```
MergeSort(A,low,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
```

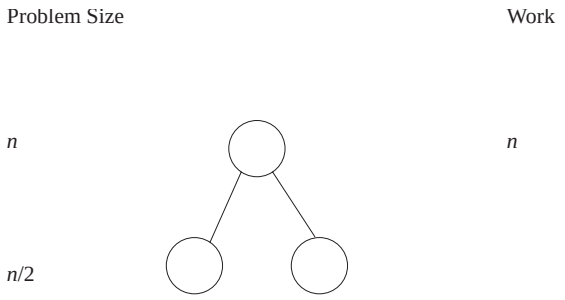
More details on mergesort can be found in almost any algorithms textbook. Suffice to say that the base case (low = high) takes one step, while the other case executes 1 step, makes two recursive calls on problems of size $n/2$, and then executes the Merge instruction, which can be done in n steps.

Thus we obtain the following recurrence for the running time of mergesort:

$$T(n) = \begin{cases} 2T(n/2) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \tag{5.3}$$

Recurrences such as this one can be understood via the idea of a recursion tree, which we introduce in the following. This concept will allow us to analyze recurrences that arise in divide-and-conquer algorithms, and those that arise in other recursive situations, such as the Towers of Hanoi, as well. A recursion tree for a recurrence is a visual and conceptual representation of the process of iterating the recurrence.

Figure 5.1: The initial stage of drawing a recursion tree



Recursion Trees

We will introduce the idea of a recursion tree via several examples. 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

$$T(n) = 2T(n/2) + n \tag{5.4}$$

as "in order to solve a problem of size n we must solve 2 problems of size $n/2$ and do n units of additional work." Similarly we can interpret

$$T(n) = T(n/4) + n^2$$

as "in order to solve a problem of size n we must solve 1 problems of size $n/4$ and do n^2 units of additional work."

We can also interpret the recurrence

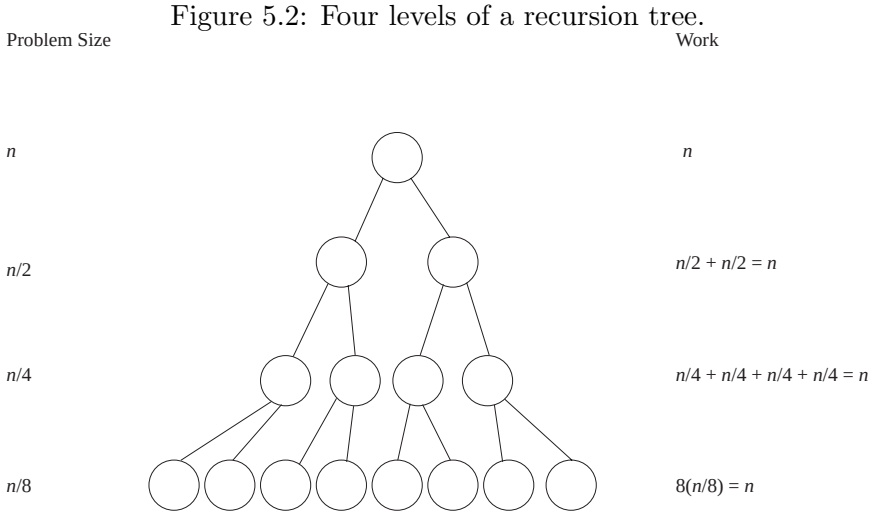
$$T(n) = 3T(n-1) + n$$

as "in order to solve a problem of size n, we must solve 3 subproblems of size $n-1$ and do n additional units of work.

In Figure 5.1 we draw the beginning of the recursion diagram for (5.4). For now, assume n is a power of 2. A recursion tree diagram has three parts. On the left, we keep track of the problem size, in the middle we draw the tree, and on right we keep track of the work done. We draw the diagram in levels, each level of the diagram representing a level of recursion. Equivalently, each level of the diagram represents a level of iteration of the recurrence. So to begin the recursion tree for (5.4), we show in level 0 on the left that we have problem of size n. Then by drawing a root vertex with two edges leaving it, we show in the middle that we are splitting it into 2 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. So that the diagram contains the relevant information, we fill in part of level one. We draw two vertices in the middle representing the two problems into which we split our main problem and show on the left that each of these problems has size $n/2$.

You can see how the recurrence is reflected in levels 0 and 1 of the recursion tree. The top vertex of the tree represents $T(n)$, an the next level we have two problems of size $n/2$, giving us the recursive term $2T(n/2)$ of our recurrence. Then 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.

Figure 5.2: Four levels of a recursion tree.



Now we continue to draw the tree in the same manner. Filling in the rest of level one and adding a few more a few more levels, we have Figure 5.2.

Let us summarize what the diagram tells us so far. At level zero (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, and so a total of n units of additional work are done. Similarly level 2 has 4 subproblems of size $n/4$ and so $4(n/4)$ units of additional work are done.

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ T(n) &= 2(2T(n/4) + n/2) + n \\ T(n) &= 4T(n/4) + n + n = 4T(n/4) + 2n \end{aligned}$$

If we examine levels 0, 1, and 2 of the diagram, we see that at level 2 we have four vertices representing four problems each of size $n/4$. This gives us the recursive term that we got after iterating the recurrence. However after we solve these problems we return to level 1 where we twice do $n/2$ additional units of work 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.

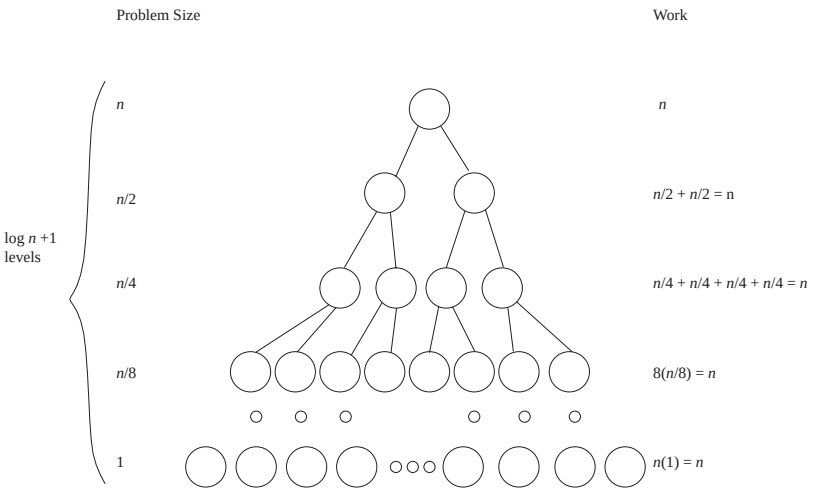
We now have enough information to be able to describe the recursion tree diagram in general. To do this, we need to determine, for each level, three things

- number of subproblems,
- size of each subproblem,
- total work done.

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 $n/2^i$. Further, since a problem of size 2^i requires 2^i units of additional work, there are $(2^i)[n/(2^i)] = n$ units of work done per level. To figure out how many levels there are in the tree, we just 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, since we start with the top level and cut the problem size in half $\log_2 n$ times.[1] We can thus visualize the whole tree in Figure 5.3.

Figure 5.3: A finished recursion tree diagram.



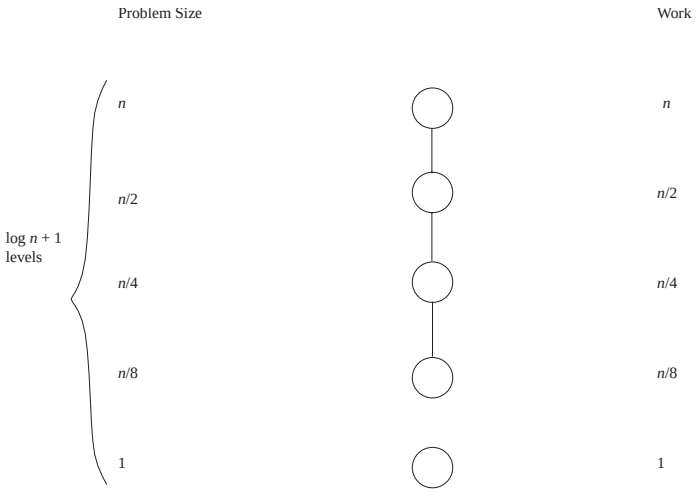
The bottom level is different from the other levels. In the other levels, the work is described by the recursive part of the recurrence, which in this case is $T(n) = 2T(n/2) + n$. At the bottom level, the work comes from the base case. Thus we must compute the number of problems of size 1 (assuming that one is the base case), and then multiply this value by $T(1)$. For this particular recurrence, and for many others we will see, it turns out that if you compute the amount of work on the bottom level as if you were computing the amount of additional work required after you split a problem of size one into 2 problems (which, of course, makes no sense) it will be the same value as if you compute it via the base case. Had we chosen to say that $T(1)$ some constant other than 1, this would not have been the case. We emphasize that the correct value always comes from the base case: it is just a coincidence that it sometimes also comes from the recursive part of the recurrence.

The bottom level of the tree represents the final stage of iterating the recurrence; at this level we have n problems each requiring work $T(1) = 1$, and after we solve them we have to do all the additional work from all the earlier levels. Thus iteration of the recurrence tells us that the solution to the recurrence is the sum of all the work done at all the levels of the recursion tree. This is exactly how we use the recursion tree to write down a solution to a recurrence.

The important thing is that we now know exactly how many levels there are, and 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, and at each level the amount of work we do is n units. Thus we conclude that the total amount

[1] 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.

Figure 5.4: A recursion tree diagram for Recurrence 5.5.



of work done to solve the problem described by recurrence (5.4) is $n(\log_2 n + 1)$. The total work done throughout the tree is the solution to our recurrence, because the tree simply models the process of iterating the recurrence. Thus the solution to recurrence (5.3) is $T(n) = n(\log n + 1)$.

More generally, we can consider a recurrence that it identical to (5.3), 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. Since one unit of time will vary from computer to computer, and since some kinds of work might take longer than other kinds, it is the big-θ behavior of $T(n)$ that is really relevant. Thus although recursion trees can give us the exact solutions (such as $T(n) = an + n \log n$ above) to recurrences, we will often just analyze a recursion tree to determine the big-Θ or even, in complicated cases, just the big-O behavior of the actual solution to the recurrence. In Problem **??** we explore whether the value of $T(1)$ actually influences the big-Θ behavior of the solution to a recurrence.

Let's look at one more recurrence.

$$T(n) = \begin{cases} T(n/2) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \tag{5.5}$$

Again, assume n is a power of two. 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.

We draw the tree for this problem in Figure 5.4.

We see in this figure that the problem sizes are the same as in the previous tree. The rest, however, is different. The number of subproblems does not double, rather it remains at one on each level. Consequently the amount of work halves at each level. Note that there are still $\log n + 1$ levels, as the number of levels is determined by how the problem size is changing, not by how many subproblems there are. So on level i, we have 1 problem of size $n/2^i$, for total work of $n/2^i$ units.

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

$$n + n/2 + n/4 + \cdots + 2 + 1 = n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \left(\frac{1}{2}\right)^{\log_2 n}\right) ,$$

which is n times a geometric series. By Theorem 4.4, the value of a geometric series in which the largest term is one 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 (5.5); 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. What we have done here is show that $T(n) = \Theta(n)$. We have not actually *found* a solution. 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, so there is no question that solutions do exist. What is often important to us in applications is not the exact form of the solution, but a big-O upper bound, or, better, a Big-Θ bound on the solution.

Exercise 5.1-2 Find a big-Θ bound for the solution to the recurrence

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

using a recursion tree. Assume that n is a power of 3.

Exercise 5.1-3 Solve the recurrence

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

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

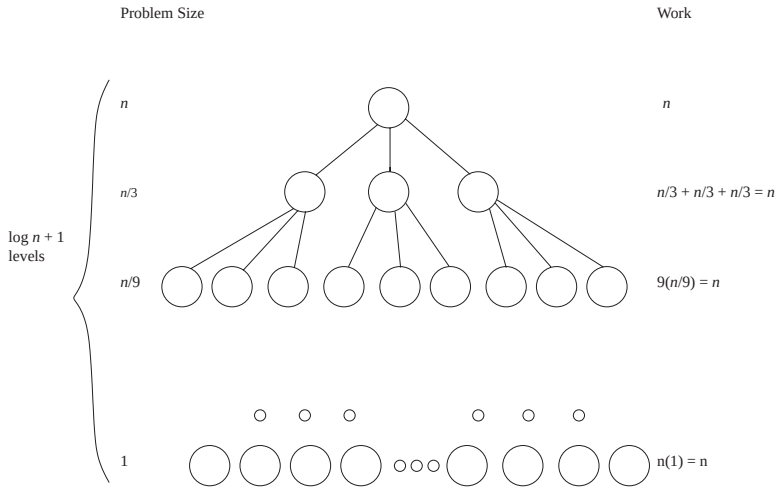
Exercise 5.1-4 Can you give a general big-Θ bound for solutions to recurrences of the form $T(n) = aT(n/2) + n$ when n is a power of 2? You may have different answers for different values of a.

The recurrence in Exercise 5.1-2 is similar to the mergesort recurrence. One difference is that at each step we divide into 3 problems of size $n/3$. Thus we get the picture in Figure 5.5. Another difference is that the number of levels, instead of being $\log_2 n + 1$ is now $\log_3 n + 1$, so the total work is still $\Theta(n \log n)$ units. Note that $\log_b n = \Theta(\log_2 n)$ for any $b > 1$.

Now let's look at the recursion tree for Exercise 5.1-3. Now, we have 4 children of size $n/2$, and we get the Figure 5.6 Let's look carefully at this tree. Just as in the mergesort tree there are $\log_2 n + 1$ levels. However, in this tree, each node has 4 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 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. Summing over the levels, we get

$$\sum_{i=0}^{\log_2 n} 2^i n = n \sum_{i=0}^{\log_2 n} 2^i .$$

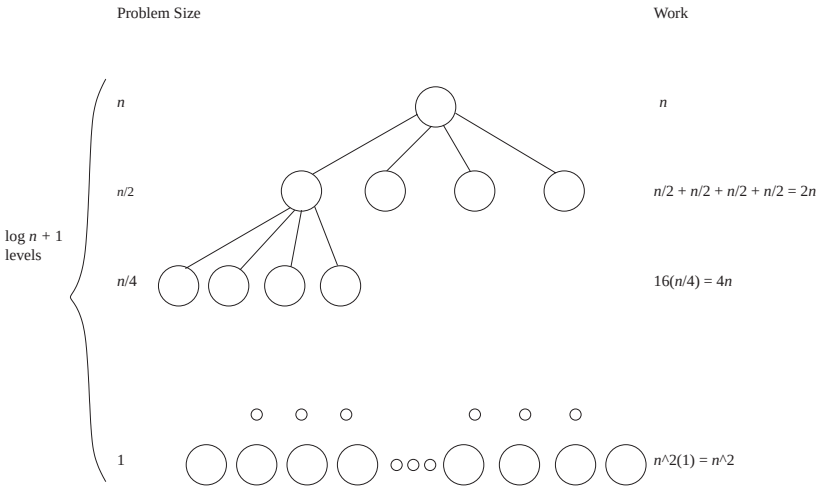
Figure 5.5: The recursion tree diagram for the recurrence in Exercise 5.1-2.



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

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log_2 n} 2^i \\ &= n \frac{1 - 2^{(\log_2 n)+1}}{1 - 2} \\ &= n \frac{1 - 2n}{-1} \\ &= 2n^2 - n \\ &= \Theta(n^2) \ . \end{aligned}$$

Figure 5.6: The Recursion tree for Exercise 5.1-3.



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

Three Different Behaviors

Now let's compare the trees 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 in each case, the size of each subproblem is 1/2 the size of of the parent problem. To see the differences, in the first case, on every level, there is the same amount of work. In the second case, the amount of work decreases, with the most work being at level 0. In fact, it decreases geometrically, so by Theorem 4.4 the total work done is bounded above and below by a constant times the work done at the root node. In the third case, 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 times the amount of work done at the last level.

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

So to answer Exercise 5.1-4, 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 :

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 $T(n) = \Theta(n^{\log_2 a})$

Cases 1 and 2 follow immediately from our observations above. We can verify case 3 as follows. At each 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 get

$$n \sum_{i=0}^{\log_2 n} (a/2)^i.$$

The sum is a geometric series, so the sum will be big-Θ of the largest term (see Theorem 4.4). Since $a > 2$, the largest term in this case is clearly the last one, namely $n(a/2)^{\log_2 n}$, and applying rules of exponents and logarithms, we get that n times the largest term is

$$n\left(\frac{a}{2}\right)^{\log_2 n} = \frac{n \cdot a^{\log_2 n}}{2^{\log_2 n}} = \frac{n \cdot 2^{\log_2 a \log_2 n}}{2^{\log_2 n}} = \frac{n \cdot n^{\log_2 a}}{n} = n^{\log_2 a}$$

Thus the total work done is $\Theta(n^{\log_2 a})$.

Important Concepts, Formulas, and Theorems

1. *Divide and Conquer Algorithm.* A *divide and conquer algorithm* is one that solves a problem by dividing it into problems that are smaller but otherwise of the same type as the original one, recursively solves these problems, and then assembles the solution of these so-called subproblems into a solution of the original one. Not all problems can be solved by such a strategy, but a great many problems of interest in computer science can.

2. *Mergesort.* In *mergesort* we sort a list of items that have some underlying order by dividing the list in half, sorting the first half (by recursively using mergesort), sorting the second half (by recursively using mergesort), and then merging the two sorted list. For a list of length one mergesort returns the same list.

3. *Recursion Tree.* A *recursion tree diagram* for a recurrence of the form $T(n) = aT(n/b)+g(n)$ has three parts. On the left, we keep track of the problem size, in the middle we draw the tree, and on right we keep track of the work done. We draw the diagram in levels, each level of the diagram representing a level of recursion. The tree has a vertex representing the initial problem and one representing each subproblem we have to solve. Each non-leaf vertex has a children. The vertices are divided into levels corresponding to (sub-)problems of the same size; to the left of a level of vertices we write the size of the problems the vertices correspond to; to the right of the vertices on a given level we write the total amount of work done at that level by an algorithm whose work is described by the recurrence, not including the work done by any recursive calls.

4. *The Base Level of a Recursion Tree.* The amount of work done on the lowest level in a recursion tree is the number of nodes times the value given by the initial condition; it is not determined by attempting to make a computation of "additional work" done at the lowest level.

5. *Bases for Logarithms.* We use $\log n$ as an alternate notation for $\log_2 n$. A fundamental fact about logarithms is that $\log_b n = \Theta(\log_2 n)$ for any real number $b > 1$.

6. *Three behaviors of solutions.* The solution to a recurrence of the form $T(n) = aT(n/2) + n$ behaves in one of the following ways:

   (a) if $a < 2$ then $T(n) = \Theta(n)$.
   (b) if $a = 2$ then $T(n) = \Theta(n \log n)$
   (c) if $a > 2$ then $T(n) = \Theta(n^{\log_2 a})$¿

Problems

1. Draw recursion trees and find big-Θ bounds on the solutions to the following recurrences. For all of these, assume that $T(1) = 1$ and n is a power of the appropriate integer.

   (a) $T(n) = 8T(n/2) + n$
   (b) $T(n) = 8T(n/2) + n^3$
   (c) $T(n) = 3T(n/2) + n$
   (d) $T(n) = T(n/4) + 1$
   (e) $T(n) = 3T(n/3) + n^2$

2. Draw recursion trees and find find exact solutions to the following recurrences. For all of these, assume that $T(1) = 1$ and n is a power of the appropriate integer.

   (a) $T(n) = 8T(n/2) + n$
   (b) $T(n) = 8T(n/2) + n^3$
   (c) $T(n) = 3T(n/2) + n$

(d) $T(n) = T(n/4) + 1$

(e) $T(n) = 3T(n/3) + n^2$

3. Find the exact solution to recurrence 5.5.

4. Show that $\log_b n = \Theta(\log_2 n)$.

5. Recursion trees will still work, even if the problems do not break up geometrically, or even if the work per level is not n^c units. Draw recursion trees and and find the best big-O bounds you can for solutions to the following recurrences. For all of these, assume that $T(1) = 1$.

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

   (b) $T(n) = 2T(n-1) + n$

   (c) $T(n) = T(\lfloor\sqrt{n}\rfloor) + 1$ (You may assume n has the form $n = 2^{2^i}$.)

   (d) $T(n) = 2T(n/2) + n \log n$ (You may assume n is a power of 2.)

6. In each case in the previous problem, is the big-O bound you found a big-Θ bound?

7. If $S(n) = aS(n-1) + g(n)$ and $g(n) < c^n$ with $0 \leq c < a$, how fast does $S(n)$ grow (in big-Θ terms)? $S(n) = a^i S(n-1) + \sum_{j=0}^{i-1} a^j g(n-j) = a^n S(0) + \sum_{j=0}^{n-1} a^j g(n-j) < a^n S(0) + \sum_{j=0}^{n-1} a^j c^{n-j} = a^n S(0) + c^n \sum_{j=0}^{n-1} \left(\frac{a}{c}\right)^j = a^n S(0) + \Theta(\left(\frac{a}{c}\right)^n) = \Theta(a^n)$

8. If $S(n) = aS(n-1) + g(n)$ and $g(n) > c^n$ with $0 < a \leq c$, how fast does $S(n)$ grow in big-Θ terms?

9. given a recurrence of the form $T(n) = aT(n/b) + g(n)$ with $T(1) = c > 0$ and $g(n) > 0$ for all n and a recurrence of the form $S(n) = aS(n/b) + g(n)$ with $S(1) = 0$ (and the same $g(n)$), is there any difference in the big-Θ behavior of the solutions to the two recurrences? What does this say about the influence of the initial condition on the big-Θ behavior of such recurrences?