Section IV.5: Recurrence Relations from Algorithms

Given a recursive algorithm with input size n, we wish to find a Θ (best big O) estimate for its run time *T(n)* either by obtaining an explicit formula for *T(n)* or by obtaining an upper or lower bound *U(n)* for *T(n)* such that *T(n)* = Θ(*U(n)*). This can be done with the following sequence of steps.

(a) Find the base cases and a recurrence relation for the algorithm.

(b) (1) Obtain an iterative formula for *T(n)* if possible. Then one can use methods in Epp, chapter 8, or Section IV.1 of these notes or

(2) Obtain an upper or lower bound *U(n)* if the iterative formula is difficult or impossible to obtain explicitly.

In Epp (section 8.2), there are examples of obtaining iterative formulas for *T(n)* for simple cases like arithmetic or geometric sequences. Also illustrated are examples where familiar summation identities can be used to get iterative formulas.

When iterative formulas for *T(n)* are difficult or impossible to obtain, one can use either **(1) a recursion tree method**, **(2) an iteration method**, or **(3) a substitution method with induction** to get *T(n)* or a bound *U(n)* of *T(n)* where *T(n)* = Θ(*U(n)*). Examples of these methods are found in Cormen (pp. 54-61). Examples of each of these methods are given below.

Definition IV.5.1: Given a recursive algorithm (Definition in Section IV-1), a **recurrence relation for the algorithm** is an equation that gives the run time on an input size in terms of the run times of smaller input sizes.

Definition IV.5.2: A **recursion tree** is a tree generated by tracing the execution of a recursive algorithm. (Cormen, p. 59)

Example IV.5.1: For Example IV.1.2. in Section IV.1 (Summing an Array), get a recurrence relation for the algorithm and iterative formula *T(n)*. Use **(a)** an iteration method and **(b)** a recursion tree method. Get the Θ estimate of *T(n)*.

Solution: (a) $T(n) = T(n\text{-}1) + 1$, since addition of the n-th element can be done by adding it to the sum of the n-1 preceding elements, and addition involves one operation. Also $T(0)=0$. Therefore, $T(k) = T(k\text{-}1) + 1$ for k between 1 and n is the recurrence relation. Iteration gives $T(n) = T(n\text{-}1) + 1 = T(n\text{-}2)+1 +1 = T(n\text{-}3)+1+1+1 = ... = T(0)+1+1+...+1 = T(0) + n = n$. (At the last stage we have added one to itself n times). Therefore, $T(n) = \Theta(n)$.

Solution: (b) At each level of the tree, replace the parent of that subtree by the constant term of the recurrence relation $T(k) = T(k\text{-}1) + 1$ and make $T(k\text{-}1)$ the child. The tree is really a list here. It is developed from left to right in the diagram below.

```
T(n)        1           1        …           1           1
            |           |                    |           |

            T(n-1)      1        …           1           1

                        |                    …           …
                          T(n-2)   …
                                             1           1
                                             |           |

                                             |           |
                                             1           1
                                             |           |
                                             T(0)        0
```

T(n) is computed by finding the sum of the elements at each level of the tree. Therefore $T(n) = n$, and $T(n) = \Theta(n)$.

Example IV.5.2: Binary Search (recursive version)
The pseudo code for recursive binary search is given below.

Algorithm Recursive Binary Search
Input: Value X of the same data type as array A, indices *low* and *high*, which are positive integers, and A, the array $A[low],..., A[high]$ in ascending order.
Output: The index in array if X is found, 0 (zero) if X not found.
Algorithm Body:

$$mid := \left\lfloor \frac{low + high}{2} \right\rfloor$$

if $X = A[mid]$ **then** *index* := *mid*
else if $X < A[mid]$ and $low < mid$ then call **Binary Search** with inputs X, *low*, *mid*-1, A
else if $X > A[mid]$ and $mid < high$ then call **Binary Search** with inputs X, *mid*+1, *high*, A
else *index* := 0
end Algorithm Recursive Binary Search

Derive a recurrence relation on Binary Search and get a Θ estimate of the worst case running time *T(n)*. Use a recursion tree method.

Solution to Example IV.5.2: One could count the number of comparisons of X to $A[mid]$. Binary Search is called on a subarray of length approximately $\frac{n}{2}$ and there are 3 comparisons in the worst case before a recursive call is needed again. So the recurrence relation is $T(n) = T(n/2) + 3$, or more specifically, $T(k) = T(k/2) + 3$ for input size $k \geq 2$. Also $T(1) = 3$.

The recursion tree is developed similar to that in Example 1. At each level, let the parent be the constant term 3 and the child be the term $T(k/2)$. The process is illustrated below.

$T(n)$	3	3	…	3	3
	$\vert$	$\vert$		$\vert$	$\vert$
	$T(n/2)$	3		3	3
		$\vert$		$\vert$	$\vert$
		$T(n/4)$	…	3	3
				…	…
				3	3
				$\vert$	$\vert$
				$T(1)$	3

The depth of the tree is approximately $\log_2 n$. Adding the values in the levels of the tree we get $T(n) \approx 3 \log_2 n$. Therefore $T(n) = \Theta(\log_2 n)$.

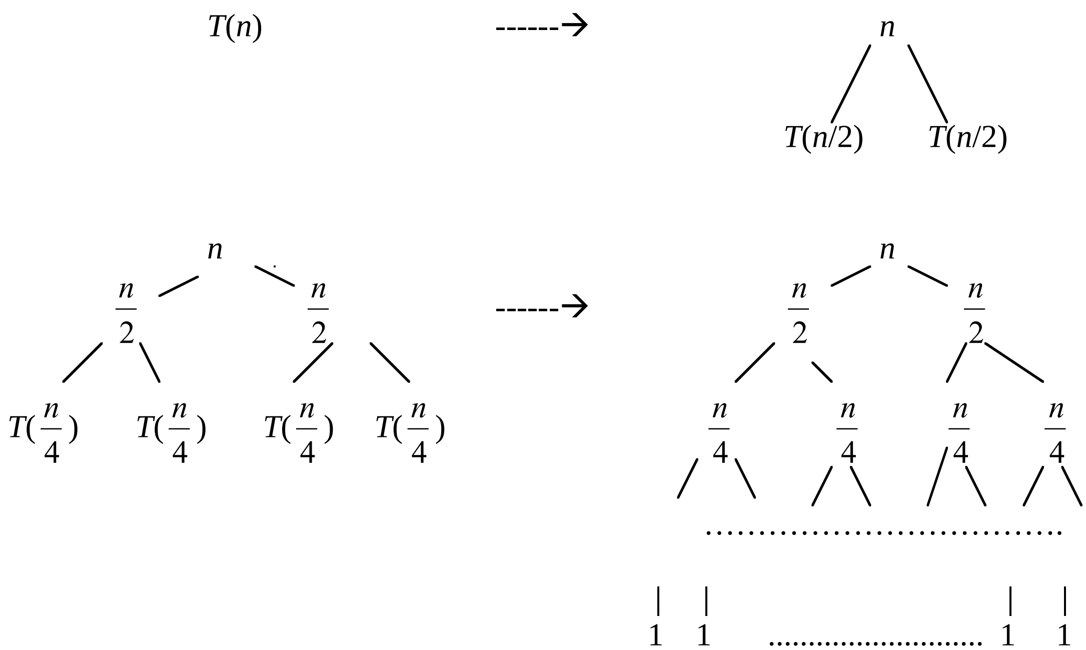
Example IV.5.3: Merge Sort
The pseudocode for Merge Sort is given in Epp, p. 529. Develop a recurrence relation and get a Θ estimate for this algorithm.

Algorithm Merge Sort
Input: positive integers *bot* and *top*, $bot \le top$, array items $A[bot]$, $A[bot+1]$, ... , $A[top]$ that can be ordered.
Output: array $A[bot]$, $A[bot+1]$, ... , $A[top]$ of the same elements as in the input array but in ascending order.
Algorithm Body:
  if $bot < top$ **then do**
    $mid := \left\lfloor \frac{bot + top}{2} \right\rfloor$
    call **Merge Sort** with input *bot*, *mid*, $A[bot]$, ..., $A[mid]$
    call **Merge Sort** with input $mid + 1$, *top*, $A[mid+1]$, ..., $A[top]$
    Merge $A[bot]$, ..., $A[mid]$ and $A[mid+1]$, ..., $A[top]$ [where **Merge** takes these two arrays in ascending order and gives an array in ascending order]
  end do
end Algorithm Merge Sort

Solution to Example IV.3.3: For simplicity let the input size be $n = 2^k$, k a positive integer. Let $T(n)$ denote the run time. In the worst case of **Merge**, there are about n comparisons needed to determine the ordering in the merged array (actually n-1 since the last element need not be compared). Since we desire a Θ estimate and not an exact formula, we can assume n. In the best case there are $n/2$ comparisons in **Merge**. This algorithm is called recursively on two subarrays of length approximately $n/2$. Therefore, the recurrence relation is $T(n) = 2T(n/2) + n$, so for particular $k \le n$,

$$T(k) = 2T\left(\frac{k}{2}\right) + k.$$

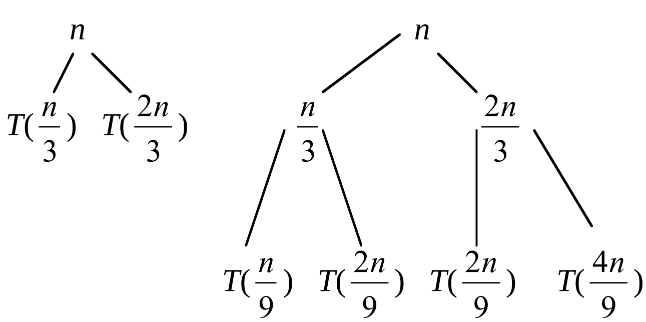
The recursion tree is developed below, using fact that, at each level of the tree, the parent node is the term **not** involving T and there are two children due to the term $2T(\frac{k}{2})$. The tree is developed as shown, going from left to right and top to bottom.



The sum of the values at each level of the tree is n (m nodes times the value $\frac{n}{m}$ in each node). There are $k = \log_2 n$ levels since we can cut the array in half k times. Thus, the sum of the elements in all nodes of the tree is $n \cdot \log_2 n$. Therefore, $T(n) = \Theta(n \cdot \log_2 n)$.

Example IV.5.4: Given the recurrence $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$, get a Θ estimate for $T(n)$ using the recursion tree method.

Solution: The recursion tree is developed below.



The construction continues until we get to the leaves. We keep multiplying n by $\frac{2}{3}$ until $(\frac{2}{3})^k \cdot n = 1$. The longest possible path of this tree is of length k where $(\frac{2}{3})^k \cdot n = 1$, i.e., where $k = \log_{3/2} n$. The sum of the values in each level of the tree is at most n. The left part of the tree plays out before the right side; the length of the path from the root to the leaves can vary from $\log_3 n$ to $\log_{3/2} n$, depending on whether argument k in $T(k)$ goes to $k/3$ or $2k/3$. One obtains $T(n) \le n \cdot \log_{3/2} n$ and $T(\mathrm{n}) \ge n \cdot \log_3 n$. By the fact that growth rate of a log function is independent of its base (Theorem II.5.8), we can conclude $T(n) = O(n \log_2 n)$ and $T(n) = \Omega(n \log_2 n)$. Therefore $T(n) = \Theta(n \log_2 n)$.

The next two examples use the iteration method. Example IV.5.5 involves a convergent geometric series, and Example IV.5.6 is more complicated since the geometric series diverges.

Example IV.5.5: Given the recurrence $T(n) = 3\,T\left(\lfloor n/4 \rfloor\right) + n$, show by an iteration method that $T(n) = \Theta(n)$.

Solution: For simplicity, we shall use m for $\lfloor m \rfloor$ since the recurrence is in terms of the floor function. The iteration proceeds as follows:

$$T(n) \le n + 3T(\frac{n}{4}) = n + 3((\frac{n}{4}) + 3T(\frac{n}{16})) = n + 3\frac{n}{4} + 9\frac{n}{16} + 27T(\frac{n}{64})$$

$\le n \sum_{i=0}^{\infty} (3/4)^i = 4n$ from the formula $\sum_{i=0}^{\infty} (r^i) = \frac{1}{1-r}$ when $|r| < 1$ (here $r = \frac{3}{4}$).

(The formula for $T(n)$ involves a decreasing geometric series with ratio $r = \frac{3}{4}$).

Therefore, $T(n) = O(n)$ (use $C = 4$ and $X_0 = N_0 = 1$ in the definition of big O). Also, $T(n) \ge n$ since $3T(\frac{n}{4}) > 0$. So $T(n) = \Omega(n)$ (use $C = 1$ and $X_0 = N_0 = 1$ in the definition of Ω). By the definition of Θ (section II.5), we have $T(n) = \Theta(n)$.

Example IV.5.6: Given the recurrence $T(n) = 3\,T(\frac{n}{2}) + n$, use the iteration method to get a big O estimate for $T(n)$.

Solution: For simplicity, we shall use m for $\lfloor m \rfloor$ since the recurrence is in terms of the floor function. The iteration proceeds as follows:

$$T(n) = n + 3T(\frac{n}{2}) = n + 3((\frac{n}{2}) + 3T(\frac{n}{4})) = n + 3\frac{n}{2} + 9\frac{n}{4} + 27T(\frac{n}{8}) = \textbf{...}\,.$$

The coefficients form the terms of the geometric series $\sum_{i=0}^{\infty} (r^i)$ where $r = 3/2$. Since

$|r| > 1$, the infinite series diverges. However, there are only finitely many terms in the sum. The sum terminates for the smallest integer k where $2^k > n$ (the number of times we divide n by 2 is $\log_2 n$), i.e., when $k \approx \log_2 n$. Since $\left\lfloor \frac{n}{2} \right\rfloor \le \frac{n}{2}$ we have

$$
\begin{aligned}
T(n) &\le n + 3\frac{n}{2} + 9\frac{n}{4} + \ldots + (3^{\log_2 n-1}/2^{\log_2 n-1})\cdot n \\
&= n \cdot \sum_{i=0}^{\log_2 n-1} (\tfrac{3}{2})^i = n \cdot ((\tfrac{3}{2})^{\log_2 n} - 1)/(\tfrac{3}{2} - 1) = 2\,n\,((\tfrac{3}{2})^{\log_2 n} - 1) \\
&= 2\,n\,(\tfrac{3}{2})^{\log_2 n} - 2\,n \\
&= 2n\left(3^{\log_2 n}/2^{\log_2 n}\right) - 2\,n \\
&= 2\cdot 3^{\log_2 n} - 2\,n \quad \text{using } 2^{\log_2 n} = n \\
&= 2\cdot n^{\log_2 3} - 2\cdot n\,.
\end{aligned}
$$

In the last step, we used
(*) $a^{\log_b n} = n^{\log_b a}$,
which is derived in Section IV.6.

Since $\log_2 3 > 1$, we know that term $n^{\log_2 3}$ dominates n. From section II.5, we can conclude that $T(n) = O(n^{\log_2 3})$.

Note: We use standard properties of logarithms in this argument. We also use the identity $\sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r-1)$ when $r \neq 1$.

Note: The Master method given in Section IV.6 will give us a much easier way to do this problem.

Substitution method with induction

The next two examples use the substitution method with induction. The method is described as follows. We show $T(n) = O(U(n))$. Showing $T(n) = \Omega(U(n))$ is similar.

(1) Guess the form of a bound $U(n)$ of $T(n)$.
(2) Assume by strong induction that for some constants C and n_0, $T(k) \le C \cdot U(k)$ for all $k \ge n_0$ and $k < n$. Show that for these values of C and n_0, $T(n) \le C \cdot U(n)$ for all $n > n_0$
(3) Find a pair C and n_0 that also satisfy the initial conditions (values of $T(1)$, $T(2)$. etc.). These must satisfy the conditions in (2) and (3); therefore, the values may change.
(4) If the given form does not work, try a different $U(n)$ with same growth rate. If this fails, try a $U(n)$ with a different growth rate.

Example IV.5.7: Find a Θ bound for $T(n)$ where $T(n) = T(n-1) + n$ and $T(0) = 1$. Use the substitution method with induction.

Solution: From Epp, section 8.2, one can get the iterative formula $T(n) = 1 + n(n+1)/2$. So we know that $T(n) = \Theta(n^2)$. However, we show how substitution and induction works in this example

Assume $T(n) \le C \cdot n^2$ for all n $> n_0$, $k < n$, and find C and n_0 that work.

Using the induction assumption, we try to find C and n_0 where

$$
\begin{aligned}
T(n) &= T(n\text{-}1) + n \\
&\le C\,(n-1)^2 + n \\
&= C\,(n^2 - 2n + 1) + n \\
&= C\,n^2 - 2\,C\,n + C + n \\
&\le C\,n^2 \text{ for all } n > n_0 .
\end{aligned}
$$

This inequality implies that $2 \cdot n \cdot C - C \ge n$, i.e., $n \le C\,(2n\text{-}1)$ or $\frac{n}{2n-1} \le C$.

Since the expression $\frac{n}{2n-1}$ is decreasing (verify by taking derivatives), we substitute 1 for n and must have $1/(2\text{-}1) \le C$, that is, $C \ge 1$. However, **no** C works for $T(0)$, and $T(1) = 2$. For $T(1) \le C\,(1)^2$ to hold, we must have $C \ge 2$. Putting together the constraints $C \ge 1$ and $C \ge 2$, we know that $C \ge 2$ works, and also that $n_0 \ge 1$ must hold. So take $C = 2$ and $n_0 = 1$ in the definition of big O.

It is true that we did not need such a complicated procedure to solve Example II.5.7. However, many times one cannot easily get a closed form for $T(n)$. Yet the substitution method with induction works. Example IV.5.8 illustrates this.

Example IV.5.8: Use the substitution and induction approach to show $T(n) = 2\,T(\left\lfloor \frac{n}{2} \right\rfloor) + n$, where $T(1) = 1$, is $\Theta(n \cdot \log_2 n)$.

Solution: Show that $T(n) \le C \cdot n \cdot \log_2 n$ for all $n > k \ge n_0$ and find a C and n_0.

Using the induction assumption and recurrence formula, one obtains

$$
\begin{aligned}
T(n) &\le 2(C\lfloor \tfrac{n}{2} \rfloor \log_2 \lfloor \tfrac{n}{2} \rfloor) + n \\
&\le Cn \log_2 \left(\tfrac{n}{2}\right) + n \\
&= Cn \log_2 n - Cn \log_2 2 + n \\
&= Cn \log_2 n - Cn + n \\
&\le Cn \log_2 n \text{ if } \mathrm{C} \le 1.
\end{aligned}
$$

(We use properties of logs and fact that $\left\lfloor \frac{n}{2} \right\rfloor \le \frac{n}{2}$ in this derivation).

However, $T(n) \le C \cdot n \cdot \log_2 n$ is not true for $n = 1$, for $1 \cdot \log_2 1 = 0$. So we must choose $n_0 \ge 2$. Now, $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. We must pick C large enough that $T(2) \le C\,(2 \log_2 2)$ and $T(3) \le C\,(3 \log_2 3)$. Any $C \ge 2$ works. So we can take $C = 2$ and $n_0 = 2$ in the definition of big O.

Changing Variables

Sometimes one can do algebraic manipulation to write an unknown recurrence in terms of a familiar one. Then one solves the familiar one and writes the solution in terms of the original variable. Example IV.5.9 illustrates this.

Example IV.5.9: Solve $T(n) = 2T(\sqrt{n}) + \log_2 n$.

Solution: Let $m = \log_2 n$. Then this recurrence becomes $T(2^m) = 2T(2^{m/2}) + m$. Let $S(m) = T(2^m)$, i.e. $S(m) = T(g(m))$ where $g(m) = 2^m$. So $S(m/2) = T(2^{(m/2})$ and the new recurrence $S(m) = S(m/2) + m$ is produced. This is the same form as Example IV.5.8; therefore it has the solution $S(m) = \Theta(m \log_2 m)$. By the substitution $n = 2^m$, we get $T(n) = T(2^m) = S(m) = \Theta(m \log_2 m) = \Theta(\log_2 n \cdot \log_2(\log_2 n))$.

Exercises:

(1) Use the recursion tree method to get a Θ estimate, coefficient 1, for the recurrence $T(n) = 2T(\frac{n}{2}) + n^2$.

(2) Use the recursion tree method to get a Θ estimate, coefficient 1, for the recurrence in Example IV.5.5. You need to get the height of the recursion tree.

(3) Use the recursion tree method to get a Θ estimate, coefficient 1, for the recurrence T $T(n) = T(\frac{n}{4}) + T(3 \cdot \frac{n}{4}) + n^2$.

(4) Use an iteration method (similar to Example IV.5.5) to get a Θ estimate, coefficient 1, for recurrence the $T(n) = 4T\left\lfloor \frac{n}{5} \right\rfloor + n$.

(5) Use an iteration method (similar to Example IV.5.6) to get a Θ estimate, coefficient 1, for the recurrence $T(n) = 5T\left\lfloor \frac{n}{4} \right\rfloor + n$.

(6) Given the recurrence $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$, show that $T(n) \le C\, n$ for any C cannot be shown using an induction argument. *S*how $T(n) = O(n)$ is correct by finding a C and b where $T(n) \le C\, n - b$ works. Assume that the initial condition is $T(1) = 1$.

(7) Solve the recurrence $T(n) = T(n^{1/3}) + T(n^{2/3}) + \log_3 n$ using a substitution *m* in terms of *n* similar to that in Example IV.5.9. You will get a recurrence in *m* that is one of the previous examples. Solve it in terms of *m* and solve the given recurrence in terms of *n*.