Chapter 4

Solutions for Algorithm Design
Exercises and Tests

4.1 Divide and Conquer

4.1.1 Solutions for Selected Exercises

Solution for Exercise #1 in Section 1.9

Solution for Part (a): This problem requires a recursive algorithm to produce a balanced binary search tree storing the first \( n \) positive integers, given \( n \) as input, and its analysis.

Suppose that a binary tree is to be given by a pointer to the root node of the tree, and that nodes are structures (or objects, or records) whose components include the “value” stored, and a “left child” and “right child,” each of which is a pointer to another node.

A sketch of the algorithm to be written was also given in the question. In order to solve part of the problem using this algorithm, it is useful to write an auxiliary function, \textit{add\_to\_value}, which takes an integer \( k \) and a pointer \( p \) to a node as input, and which adds \( k \) to the value of every node in the subtree whose root is pointed to by \( p \). In the following pseudocode, \( \uparrow \text{node} \)” will be used to declare something to be a pointer to a node and, if \( p \) has this type, then \( *p \) will represent the node that \( p \) points to.

\[
\text{add\_to\_value}(k: \text{integer}, p: \uparrow \text{node})
\]

\[
\text{if } p \neq \text{null then}
\]

\[
* \text{value} := * \text{value} + k
\]

\[
\text{add\_to\_value}(k, *p.\text{left\_child})
\]

\[
\text{add\_to\_value}(k, *p.\text{right\_child})
\]

\[
\text{end if}
\]

It can be argued that if the pointer \( p \) points to a binary tree of size \( s \) (for any integer \( s \geq 0 \)), then the above function uses \( \Theta(s) \) pointer dereferences and accesses of data, arithmetic operations, and assignments of values to variables.

Now, a function \textit{build\_tree}, which takes a positive integer \( n \) as input and returns a pointer to a balanced binary search tree storing the first \( n \) positive integers, corresponds to the pseudocode shown in Figure 4.1 (on page 104).
build_tree(n : integer) : ↑node

p, q1, q2: ↑node;  middle : integer

if n ≤ 0 then
  return null
else
  middle := ⌊n+1/2⌋
  q1 := build_tree(middle − 1)
  q2 := build_tree(n − middle)
  add_to_value(middle, q2)
  new(p)
  *p.value := middle
  *p.left_child := q1
  *p.right_child := q2
  return p
end if

Figure 4.1: Algorithm build_tree

Solution for Part (b): Let $T(n)$ be the number of operations used by the function build_tree, when it is given a positive integer $n$ as input. Since this function calls the function add_to_value with a pointer to a binary tree of size $n − middle = n − ⌊n+1/2⌋ = ⌊n−1/2⌋$ as input, and since $middle − 1 = ⌊n−1/2⌋$, it can be shown that $T(n)$ satisfies a recurrence

$$T(n) \leq \begin{cases} \end{cases}$$

for positive constants $c_1$ and $c_2$, and that it also satisfies a recurrence

$$T(n) \geq \begin{cases} \end{cases}$$

for (smaller) positive constants $\hat{c}_1$ and $\hat{c}_2$. Now, suppose that $R(n)$ satisfies the recurrence

$$R(n) = \begin{cases} \end{cases}$$

Then, it can be shown that $T(n) \in \Theta(R(n))$, so it’s sufficient to find an asymptotic bound in closed form for $R(n)$ instead.

Exercise: If you’d like to have more practice, use the substitution method to prove that $T(n) \in \Theta(R(n))$.

It can also be shown that $R(n)$ is a nondecreasing function of $n$.

Exercise: If you’d like to have more practice in the use of mathematical induction, then prove that $R(n) \leq R(n + 1)$ for every integer $n \geq 1$ using induction on $n$. 
Solution for Part (c): The fact that $R$ is a nondecreasing function implies that $R([\frac{n-1}{2}]) \leq R([\frac{n}{2}])$ and that $R([\frac{n-1}{2}]) \leq R([\frac{n}{2}])$, and that if $S(n)$ satisfies the recurrence

$$S(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 2S([\frac{n}{2}]) + n & \text{if } n \geq 2, \end{cases}$$

then $R(n) \leq S(n)$ for every integer $n \geq 1$ — so, that, in particular, $R(n) \in O(S(n))$.

Now, the “master theorem” can be used to establish that $S(n) \in \Theta(n \log n)$, and this implies that $R(n) \in O(n \log n)$, and hence that $T(n) \in O(n \log n)$ as well.

We would also like to show that $R(n) \in \Omega(n \log n)$. To do this, note first that since $R(n)$ is an increasing function, $R([\frac{n-1}{2}]) \leq R([\frac{n}{2}])$, and that if $U(n)$ satisfies the recurrence

$$U(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 2U([\frac{n-1}{2}]) + n & \text{if } n \geq 2, \end{cases}$$

then $R(n) \geq U(n)$ for every integer $n \geq 1$. The recurrence for $U(n)$ is still not in the form required so that we can apply the master theorem (although it’s getting closer). Now, let $V(n) = U(n - 1)$ for all $n$. Then $V(n) = U(n - 1) = 1$ if $n \leq 2$, and if $n > 2$ then

$$V(n) = U(n - 1)$$
$$= 2U([\frac{n-2}{2}]) + n - 1$$
$$= 2U([\frac{n}{2}] - 1) + n - 1$$
$$= 2V([\frac{n}{2}]) + n - 1.$$

Now, finally, set

$$W(n) = \begin{cases} \frac{1}{2} & \text{if } n \leq 1, \\ 2W([\frac{n}{2}]) + n - 2 & \text{if } n \geq 2. \end{cases}$$

Then, $W(1) = \frac{1}{2} < V(1)$, $W(2) = 1 \leq V(2)$, and one can show using induction on $n$ that $V(n) \geq W(n)$ for every integer $n \geq 1$.

(Note that $W(1)$ was defined to be $\frac{1}{2}$ instead of 1 in order to ensure that all these inequalities are satisfied: If $W(1)$ had been set to be 1 and $W(2)$ had been defined using the recursive definition shown above, then $W(2)$ would have turned out to be greater than $V(2)$, so that the above claim “$V(n) \geq W(n)$ for every integer $n \geq 1$” would have been false.)

Now (at last) the recurrence for $W(n)$ has a form that allows us to apply the master theorem. This theorem implies that $W(n) \in \Theta(n \log n)$. Since $V(n) \geq W(n)$ for all $n \geq 1$, it follows that $V(n) \in \Omega(n \log n)$. Since $U(n) = V(n + 1)$, $U(n) \in \Omega((n + 1) \log(n + 1)) = \Omega(n \log n)$, too. Finally, since $R(n) \geq U(n)$ for every positive integer $n$, it follows that $R(n) \in \Omega(n \log n)$.

Since we have already shown that $R(n) \in \Omega(n \log n)$, it follows that $R(n) \in \Theta(n \log n)$. Thus, $\Theta(R(n)) = \Theta(n \log n)$, and we can conclude that $T(n)$ (the running time of the algorithm we started with) is in $\Theta(n \log n)$, since we had already observed that $T(n) \in \Theta(R(n))$.

Exercise: If you want still more practice, try to confirm directly that $T(n) \in \Theta(n \log_2 n)$ without performing changes of recurrences, and so on, but by working with the original recurrences (shown at the bottom of the first page of these solutions) instead.
Solution for Exercise #2 in Section 1.9

This question required an algorithm that generalizes the one obtained in the solution for Exercise #1 above and the analysis of the new algorithm.

In particular, we now wish to write an algorithm which accepts two integers \( n \) and \( k \) as inputs (with \( n \) being nonnegative) and which produces a balanced binary search tree with nodes \( k, k+1, \ldots, k+n-1 \) as output.

An algorithm that solves this problem (and agrees with the description given in the question) is shown in Figure 4.2, which is shown below.

```plaintext
build_tree(n, k : integer) : ↑node
p, q1, q2 : ↑node; middle : integer
if n ≤ 0 then
    return null
else
    middle := \( \lfloor \frac{n-1}{2} \rfloor \)
    q1 := build_tree(middle, k)
    q2 := build_tree(n - middle - 1, k + middle + 1)
    new(p)
    *p.value := k + middle
    *p.left_child := q1
    *p.right_child := q2
    return p
end if
```

Figure 4.2: Algorithm build_tree

In order to convince yourself that the algorithm is correct, consider the cases \( n = 0, n = 1, \) and \( n = 2, \) and then try to prove using induction on \( n \) that the algorithm always returns the required binary search tree, after that.

It isn’t too hard to see that this algorithm has a running time \( T(n) \) that’s a function of the input \( n \) and that satisfies a recurrence

\[
T(n) = \begin{cases} 
  c_1 & \text{if } n \leq 0, \\
  T\left(\lfloor \frac{n-1}{2} \rfloor \right) + T\left(\lceil \frac{n-1}{2} \rceil \right) + c_2 & \text{if } n \geq 1.
\end{cases}
\]

for positive constants \( c_1 \) and \( c_2, \) assuming that pointer dereferences, reading the current values of variables, assignments of values to variables, and integer arithmetic are all counted as having unit cost.

Using this recurrence it’s easy to confirm that \( T(0) = c_1, T(1) = c_2 + 2c_1, T(2) = 2c_2 + 3c_1, \) and \( T(3) = 3c_2 + 4c_1. \) After this you might guess that

\[
T(n) = c_2n + (n + 1)c_1 = (c_1 + c_2)n + c_1
\]

for every integer \( n \geq 0 \) — and it’s easy to use induction on \( n \) to prove that this is correct.
An alternative strategy (for finding a solution for the above recurrence) would be to “guess” that it has the same asymptotic complexity as a function $U(n)$ (that is, it’s in $\Theta(U(n))$), where

$$U(n) = \begin{cases} c_1 & \text{if } n \leq 1, \\
2U(\lfloor \frac{n}{2} \rfloor) + c_2 & \text{if } n \geq 2, \end{cases}$$

so that $U(n)$ satisfies a similar recurrence, but also so that an asymptotic bound for $U(n)$ can be obtained using the master theorem. The master theorem could then be used to confirm that $U(n) \in \Theta(n)$, and the substitution method could then be used to prove that $T(n) \in \Theta(n)$ as well.

Thus there are several ways to prove that $T(n) \in \Theta(n)$ (assuming the unit cost criterion as it’s given above).

Note that this implies that the above “generalized” algorithm is asymptotic more efficient than the algorithm that was given as an answer for the previous question, as this question suggested.