For parts (a) through (c), use Θ-notation to provide asymptotically tight bounds in terms of \( n \) for the solution to the recurrence.

(a) \( T(n) = 5T(n/2) + n \)

Solution: \( T(n) = \Theta(n^{\log_5 5}) \)

Explaination: The \( i \)th level of the recursion tree sums to \((5/2)^i n\), making the level sums an increasing geometric series bounded by its largest term, the number of leaves. There are \( 5^{\log_5 n} = n^{\log_5 5} \) leaves.

(b) \( T(n) = 8T(n/4) + n\sqrt{n} \)

Solution: \( T(n) = \Theta(n\sqrt{n} \log n) \)

Explaination: Each level of the recursion tree sums to \( n\sqrt{n} \). There are \( O(\log n) \) levels.

(c) \( T(n) = T(n/4) + T(2n/3) + n \)

Solution: \( T(n) = \Theta(n) \)

Explaination: The \( i \)th level of the recursion tree sums to \((11/12)^i n\), making the level sums a decreasing geometric series bounded by its largest term, \( n \).

(d) Give an asymptotically tight bound for the running time of the algorithm STOOGESORT(\( A[1..n] \)).

Solution: There are three recursive calls, each of size \( \lceil 2n/3 \rceil \) plus a constant amount of addition work to perform, so we get a running time recurrence of \( T(n) = 3T(2n/3) + 1 \). The \( i \)th level of the recursion tree sums to \( 3^i \), making the level sums an increasing geometric series bounded by its largest term, the number of leaves. There are \( 3^{\log_{3/2} n} = n^{\log_{3/2} 3} \) leaves, so the running time is \( \Theta(n^{\log_{3/2} 3}) \).
Recall the restricted Tower of Hanoi puzzle described in the problem sheets.

(a) Describe a recursive algorithm to transfer \( n \) disks from the left peg to the right peg under the restriction that every move must involve the middle peg.

**Solution:** We need to move the top \( n - 1 \) disks to the right peg before we can move the bottom disk to the middle peg. But then we need to get those \( n - 1 \) disks back to the left peg so we can move the bottom disk to the right peg. Finally, we can move the \( n - 1 \) disks from the left peg to right peg. Since we’re always moving from left to right peg or vice versa, these problems are the same one we’re solving here (or its reflection) and can be solved recursively.

For pseudocode, let \textsc{RestrictedHanoi}(n, src, dst) move the top \( n \) disks from the src peg to the dst peg where neither src nor dst are the middle peg. Our algorithm calls the procedure with src as the left peg and dst as the right peg.

```
\textbf{RestrictedHanoi}(n, src, dst):
  \textbf{if} \ n \geq 1
    \textbf{RestrictedHanoi}(n - 1, src, dst)
    \text{move disk } n \text{ from } src \text{ to middle peg}
    \textbf{RestrictedHanoi}(n - 1, dst, src)
    \text{move disk } n \text{ from middle peg to } dst
    \textbf{RestrictedHanoi}(n - 1, src, dst)
```

(b) It is likely that your algorithm from part (a) uses exactly \( 3^n - 2 \) \( 3^n - 1 \) moves. Give a short proof that this is the case.

**Solution:** We perform three recursive calls on \( n - 1 \) disks plus two additional moves, and there are no moves to do when there are 0 disks. Therefore, we can express the number of moves using a recurrence \( T(n) = 3T(n - 1) + 2 \) with \( T(0) = 0 \). Now, assume we use \( 3^k - 1 \) moves to move \( k < n \) disks. If \( n = 0 \), then we indeed have \( 3^n - 1 = 0 \). Otherwise,

\[
T(n) = 3T(n - 1) + 2 \\
= 3(3^{n-1} - 1) + 2 \\
= 3^n - 1.
\]

**Solution:** The divide-and-conquer procedure \textsc{FindLocalMin}(A[1..n]) finds a local minimum given \( A[1..n] \) has the property described above. Note these properties only make sense when \( n \geq 3 \).

\[
\text{\textsc{FindLocalMin}(A[1..n]):}
\]

\[
\text{if } n = 3 \\
\quad \text{return } A[2] \\
\text{else} \\
\quad m \leftarrow \lfloor n/2 \rfloor \\
\quad \text{if } A[m] > A[m-1] \\
\quad \quad \text{return } \text{\textsc{FindLocalMin}(A[1..m])} \\
\quad \text{else if } A[m] > A[m+1] \\
\quad \quad \text{return } \text{\textsc{FindLocalMin}(A[m+1..n])} \\
\quad \text{else} \\
\quad \quad \text{return } A[m]
\]

The algorithm is a binary search with constant time per recursive call (described by the running time recurrence \( T(n) = T(n/2) + 1 \)), so it takes \( O(\log n) \) time overall.

**Explanation:** Assume the procedure finds a local minimum on arrays of length \( k \) where \( 3 \leq k < n \). If \( n = 3 \), it returns the middle element which is guaranteed to be a local minimum by the special properties of \( A \). Suppose \( n > 3 \). If \( A[m] > A[m-1] \), then \( A[1..m] \) has the special property, and we find a local minimum in that smaller array by the induction hypothesis. Similarly, if \( A[m] > A[m+1] \), then \( A[m+1..n] \) has the property and we find the local minimum. If neither inequality holds, then \( A[m] \) itself is a local minimum by definition.
Our goal for this problem is to design an algorithm that given three strings \(A[1..m],\ B[1..n],\) and \(C[1..m + n],\) determines whether \(C\) is a shuffle of \(A\) and \(B.\)

(a) Let \(\text{IsShuffle}(i, j)\) be a function that, for any \(0 \leq i \leq m\) and \(0 \leq j \leq n,\) returns \(\text{TRUE}\) if \(C[1..i + j]\) is a shuffle of \(A[1..i]\) and \(B[1..j]\) and \(\text{FALSE}\) otherwise. Give a recursive definition of \(\text{IsShuffle}(i, j)\) (i.e., a recurrence relation).

\[
\text{IsShuffle}(i, j) = \begin{cases} 
\text{TRUE} & \text{if } i = j = 0 \\
[B[j] = C[i + j]] \land \text{IsShuffle}(i, j - 1) & \text{if } i = 0, j > 0 \\
[A[i] = C[i + j]] \land \text{IsShuffle}(i - 1, j) & \text{if } i > 0, j = 0 \\
([A[i] = C[i + j]] \land \text{IsShuffle}(i - 1, j)) \lor \\
([B[j] = C[i + j]] \land \text{IsShuffle}(i, j - 1)) & \text{otherwise}
\end{cases}
\]

**Explanation:** If \(i = j = 0,\) then \(\text{IsShuffle}(i, j)\) is saying whether the empty string is a shuffle of two empty strings, which is true. If \(i = 0\) but \(j > 0,\) then the last characters of \(B[1..j]\) and \(C[1..i + j]\) must match, and the rest of the strings must match as well, which is checked through a recursive call to \(\text{IsShuffle}(i, j - 1).\) A similar argument holds for \(i > 0\) and \(j = 0.\) Finally, if \(i > 0\) and \(j > 0,\) then for \(C[1..i + j]\) to be a shuffle of \(A[1..i]\) or \(B[1..j]\), it must end with one of their last characters and be a shuffle of what remains from the substring we just pulled a character from.

(b) Describe an efficient algorithm for determining if \(C[1..m + n]\) is a shuffle of \(A[1..m]\) and \(B[1..n].\) Using big-\(O\) notation, state the running time and space usage of your algorithm.

**Solution:** We need to compute \(\text{IsShuffle}(m, n)\). To do so, we need to compute \(\text{IsShuffle}(i, j)\) for all \(0 \leq i \leq m\) and \(0 \leq j \leq n,\) so we’ll store those results in a two-dimensional array \(\text{IsShuffle}[0..m][0..n].\) Each entry depends upon up to two entries in higher rows or more leftward columns, so we’ll fill the array row by row from top to bottom, going left to right in each row. Here is the pseudocode:

\begin{verbatim}
DETERMINEISSHUFFLE(A[1..m], B[1..n]):
    for i ← 0 to m
        for j ← 0 to n
            if i = 0 and j = 0
                IsShuffle[i, j] ← TRUE
            else if i = 0
                IsShuffle[i, j] ← [B[j] = C[i + j]] \land IsShuffle[i, j - 1]
            else if j = 0
                IsShuffle[i, j] ← [A[i] = C[i + j]] \land IsShuffle[i - 1, j]
            else
                IsShuffle[i, j] ←
                    ([A[i] = C[i + j]] \land IsShuffle[i - 1, j]) \lor
                    ([B[j] = C[i + j]] \land IsShuffle[i, j - 1])
    return IsShuffle[m, n]
\end{verbatim}
The algorithm uses $O(mn)$ space to store $IsShuffle[0..m, 0..n]$. Each entry is filled in constant time, so it takes $O(mn)$ time total.