Using $\Theta$-notation, provide asymptotically tight bounds in terms of $n$ for the solution to each of the following recurrences.

(a) $T(n) = 9T(n/3) + n^2$

**Solution:** $T(n) = \Theta(n^2 \log n)$

**Explanation:** The third case of the master theorem applies with $a = 9$, $b = 3$, $d = 2$.

(b) $T(n) = T(n/2) + \sqrt{n}$

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

**Explanation:** The first case of the master theorem applies with $a = 1$, $b = 2$, $d = 1/2$.

(c) $T(n) = 5T(n/2) + n^2$

**Solution:** $T(n) = \Theta(n^{\log_2 5})$

**Explanation:** The second case of the master theorem applies with $a = 5$, $b = 2$, $d = 2$.

(d) $T(n) = T(n/4) + T(3n/4) + n$

**Solution:** $T(n) = \Theta(n \log n)$

**Explanation:** Use recursion trees. The nodes at any full row of depth $i$ sum up to $n$. There are at least $\log_4 n$ full rows at most $\log_4/3 n$ rows of any kind in the tree, so the sum over all rows is $\Theta(n \log n)$.

(e) $T(n) = T(n/6) + T(2n/3) + n$

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

**Explanation:** Use recursion trees. The nodes at depth $i$ sum up to at most $(5/6)^i n$. The sum over all rows is a decreasing geometric series bounded by a constant times its largest term $n$, so $T(n) = \Theta(n)$. 


Describe an algorithm to sort an arbitrary stack of $n$ pancakes using as few flips as possible. Exactly how many flips does your algorithm perform in the worst case?

**Solution:** The recursive procedure `SortPancakes(n)` sorts the top $n$ pancakes in the stack.

```plaintext
SortPancakes(n):
  if n > 1
    k ← location of largest pancake of top n pancakes
    flip top k pancakes
    flip top n pancakes
    SortPancakes(n − 1)
```

The algorithm does 2 flips per recursive call except for `SortPancakes(1)`, so it does $2(n − 1)$ flips total.

**Explanation:** If $n = 1$, then the top 1 pancakes are trivially sorted, so the algorithm does nothing. For larger $n$, we find the largest pancake of the top $n$ and move it to the top of the stack with a single flip. The next flip brings that largest pancake back to the bottom of the stack of $n$ where it belongs in the final sorted order. The recursive call `SortPancakes(n − 1)` then successfully sorts the remaining pancakes by induction.
Suppose you are given a sorted array of \( n \) distinct numbers that has been rotated \( k \) steps, for some unknown integer \( k \) between 1 and \( n - 1 \). Describe and analyze an algorithm to compute the unknown integer \( k \) in \( o(n) \) time. Yes, that is a little-oh.

**Solution:** The divide-and-conquer procedure \( \text{FINDK}(A[1..n]) \) finds the unknown integer \( k \) given that \( A \) is a sorted array rotated \( k \) steps for some \( k \) between 1 and \( n - 1 \).

\[
\text{FINDK}(A[1..n]):
\begin{align*}
\text{if } n &= 2 \\
\text{return } 1 \\
\text{else} & \\
& m \leftarrow \lfloor n/2 \rfloor \\
& \text{if } A[m] < A[1] \\
& \quad \text{return } \text{FINDK}(A[1..m-1]) \\
& \text{else} \\
& \quad \text{return } \text{FINDK}(A[m..n]) + m - 1
\end{align*}
\]

The algorithm is a binary search with constant time per recursive call, so it takes \( O(\log n) \) time overall.

**Explanation:** There exist no \( k \) between 1 and \( 1 - 1 = 0 \), so we make our base case \( n = 2 \). In that case, the only possible answer is \( k = 1 \). For larger \( n \), we compare \( A[m] \) and \( A[1] \). All elements in \( A[2..k] \) are greater than \( A[1] \), and all elements in \( A[k+1..n] \) are less than \( A[1] \). Therefore, if \( A[m] < A[1] \), then \( m \in \{k+1, \ldots, n\} \) and \( k < m \). The algorithm searches the elements to the left of \( m \). Because they have the same form as a rotated array, the recursive call correctly finds \( k \) in those elements by induction. On the other hand, if \( A[m] > A[1] \), then \( k \geq m \). The algorithm searches the elements to the right of and including \( m \). These elements again have the same form as a rotated array, so the recursive call correctly finds the amount of rotation for \( A[m..n] \) by induction. However, we have to add \( m - 1 \) to get the correct index \( k \) relative to the original array \( A[1..n] \).
Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

**Solution:** We will consider a dynamic programming solution. For all integers $i$ such that $1 \leq i \leq n + 1$, let $MaxTotal(i)$ be the maximum total score you can achieve dancing to songs $i$ through $n$ (so $MaxTotal(n + 1) = 0$). Our algorithm must compute $MaxTotal(1)$. $MaxTotal$ can be defined recursively as follows.

$$MaxTotal(i) = \begin{cases} 
0 & \text{if } i = n + 1 \\
\max\{MaxTotal(i + 1), \\
Score[i] + MaxTotal(\min\{i + Wait[i] + 1, n + 1\})\} & \text{otherwise}
\end{cases}$$

We can store the recursive answers in an array $MaxTotal[1..n + 1]$. $MaxTotal[n + 1]$ is set to 0 as a base case. Every other entry depends upon one or two entries to its right, so we can fill the array from right to left (high index to low). There are $O(n)$ entries to fill in constant time each, so the total running time is $O(n)$.

**Explanation:** When computing $MaxTotal(i)$ for $i < n + 1$, we want to take the better of two options. We could skip song $i$ and try to maximize our score for all songs $i + 1$ through $n$ by computing $MaxTotal(i + 1)$. Or, we could dance to song $i$, immediately gain $Score[i]$ points, sit out songs $i + 1$ through $i + Wait[i]$, and then try to maximize our score for songs $i + Wait[i] + 1$ through $n$ by computing $MaxTotal(\min\{i + Wait[i] + 1, n + 1\})$ (the inner min handles the case that $i + Wait[i] + 1 > n + 1$).
Describe and analyze an efficient algorithm to compute the length of the longest common subsequence of $A[1..m]$ and $B[1..n]$.

**Solution:** We will consider a dynamic programming solution. For all integers $i, j$ such that $0 \leq i \leq m$ and $0 \leq j \leq n$, let $LCS(i, j)$ be the length of the longest common subsequence between $A[1..i]$ and $B[1..j]$. Our algorithm must compute $LCS(m, n)$. $LCS$ can be defined recursively as follows.

$$LCS(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max\{LCS(i - 1, j), LCS(i, j - 1)\} & \text{if } A[i] \neq B[j] \\
\max\{LCS(i - 1, j), LCS(i, j - 1), 1 + LCS(i - 1, j - 1)\} & \text{otherwise}
\end{cases}$$

We can store the recursive answers in a 2-dimensional array $LCS[0..m][0..n]$. Each of $LCS[0][\cdot], LCS[\cdot][0]$ is set to 0 as the base cases. Every other entry depends upon entries of lower $i$ or $j$ index, so we can fill the array row-by-row, column-by-column in increasing order of indices (so we use an outer for loop over $i$ increasing and an inner for loop over $j$ increasing). There are $O(mn)$ entries to fill in constant time each, so the total running time is $O(mn)$.

**Explanation:** When computing $LCS(i, j)$ for $i > 0, j > 0$ there are two possibilities. If $A[i] \neq B[j]$, then there is no common subsequence ending with both of those characters. We consider the better of two options: working only with the first $i - 1$ characters of $A$ or working with the first $j - 1$ characters of $B$. If $A[i] = B[j]$, then we could work with those same two options, but we also have the third option of using $A[i] = B[j]$ as the last character in the longest common subsequence. Any subsequence using that as the last character only has the first $i - 1$ characters of $A$ and $j - 1$ characters of $B$ left to work with, but we do get to add the 1 for including $A[i] = B[j]$ in the common subsequence.