Answer each of the following questions. There is no need to justify your answers for these questions. For parts (a) through (c), use $\Theta$-notation to provide asymptotically tight bounds in terms of $n$ for the solution to the recurrence.

(a) Using $\Theta$-notation in terms of $n$, what is the solution to the recurrence $T(n) = 3T(n/3) + n$?

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

**Explanation:** Each level of the recursion tree sums to $n$. There are $O(\log n)$ levels.

(b) Using $\Theta$-notation in terms of $n$, what is the solution to the recurrence $T(n) = 3T(n/2) + n^2$?

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

**Explanation:** The $i$th level of the recursion tree sums to $(3/4)^i n^2$. The level sums form a decreasing geometric series bounded by the largest term at the root.

(c) Using $\Theta$-notation in terms of $n$, what is the solution to the recurrence $T(n) = T(7n/9) + T(n/9) + n$?

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

**Explanation:** The $i$th level of the recursion tree sums to $(8/9)^i n$. The level sums form a decreasing geometric series bounded by the largest term at the root.

(d) Consider the following recursive function which is defined in terms of a fixed array $X[1 .. n]$.

$$WTF(i, j) = \begin{cases} 
0 & \text{if } i \leq 0 \text{ or } j \leq 0 \\
X[j] + WTF(i - 1, j) + WTF(i, \lfloor j/2 \rfloor) & \text{otherwise}
\end{cases}$$

Using $\Theta$-notation in terms of $n$, how long does it take to compute $WTF(n, n)$ using dynamic programming?

**Solution:** It takes $\Theta(n^2)$ time.

**Explanation:** We only need to solve the $\Theta(n^2)$ subproblems where $0 \leq i \leq n$ and $0 \leq j \leq n$. These can be solved in row major order (increasing $i$ as the outer loop, increasing $j$ as the inner loop) to guarantee subproblem solutions are available to subproblems that depend upon them. Evaluating each subproblems takes $\Theta(1)$ time given its dependencies, so evaluating all the subproblems takes $\Theta(n^2)$ time total.

Note: Some students observed you only need $\Theta(\log n)$ values for $j$ making the solution $\Theta(n \log n)$, so they were given a bit of extra credit.
(a) State a worst-case running time recurrence for \textsc{Mom}_7\textsc{Select}. Give a brief justification for your answer.

\textbf{Solution:} Let $T(n)$ be the worst-case running time of \textsc{Mom}_7\textsc{Select}(A[1 .. n], k). It takes $O(n)$ time to compute all $\lceil n/7 \rceil$ medians of 7 elements and partition the array around the median of medians. Computing the median of these medians takes $T(\lceil n/7 \rceil)$ time. Suppose $k > r$. There are at least $n/7/2 = n/14$ medians smaller than the median of medians, and 4 elements per block less than each of these medians. Therefore, there are at least $4n/14 = 2n/7$ elements smaller than the median of medians, and the second recursive call is over at most $n - 2n/7 = 5n/7$ elements. Similarly, the second call is over at most $5n/7$ elements if $k < r$. \textit{It suffices to solve $T(n) \leq T(n/7) + T(5n/7) + n$.} ■

(b) Let $Edit_2(i, j)$ be the minimum total cost of any set of insertions, deletions, and substitutions that transform $A[1 .. i]$ into $B[1 .. j]$. Fill in the blanks to complete the following recursive definition of $Edit_2(i, j)$. There is no need to justify your answer.

\textbf{Solution:} 

$$Edit_2(i, j) = \begin{cases} 
3i & \text{if } j = 0 \\
2j & \text{if } i = 0 \\
\min \left\{ \begin{array}{ll}
Edit_2(i, j - 1) + 2 \\
Edit_2(i - 1, j) + 3 \\
\quad \text{otherwise}
\end{array} \right. \\
Edit_2(i - 1, j - 1) + 1 \cdot [A[i] \neq B[j]] 
\end{cases}$$

\textbf{Explanation:} The case $i = 0$ means we need to insert $j$ characters into the empty string $A[1 .. i]$ at cost $2j$. The first case in the min is the choice to create the last character of $B[1 .. j]$ through an insertion of cost 2. The second case in the min is the choice to delete the last character of $A[1 .. i]$ at cost 3. The third case is the choice to substitute the last character of $A[1 .. i]$ with the last character of $B[1 .. j]$ which has cost 1 or 0 depending on if those characters are distinct.
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 a divide-and-conquer/binary search algorithm to compute the unknown integer \( k \) in \( O(\log n) \) time. You should briefly justify the correctness and running time of your algorithm.

**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 \( n - 1 \).

```python
FindK(A[1..n]):
    if n = 2
        return 1
    else
        m ← ⌈n/2⌉
            return FindK(A[1 .. m-1])
        else
            return FindK(A[m .. n]) + m - 1
```

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 \( A[1 .. n] \).

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

■
Refer to the solitaire game described in the problem sheets. For this problem, you may assume the input is given as an array $P[1..n]$ where the $i$th card from the left has point value $P[i]$.

(a) Let $MaxPoints(i)$ be the maximum number of points you can collect using only cards $i$ through $n$ ordered left to right.

Give a recursive definition, including the base case(s), for $MaxPoints(i)$. You should briefly explain each of the cases in your recursive definition.

**Solution**: If $i > n$, then there are no cards available, and we may collect no points. Otherwise, we have a choice of taking or discarding the $i$th card from the left. If we discard it, we can still collect $MaxPoints(i + 1)$ points from cards $i + 1$ through $n$. Otherwise, we get our $P[i]$ points right away, but then we have to discard some cards leaving only cards $i + 1 + 3 = i + 4$ through $n$ available. $MaxPoints(i)$ can be defined recursively as follows.

$$MaxPoints(i) = \begin{cases} 
0 & \text{if } i > n \\
\max\{MaxPoints(i + 1), P[i] + MaxPoints(i + 4)\} & \text{otherwise}
\end{cases}$$

(b) Describe and analyze an efficient dynamic programming algorithm to determine the maximum number of points you can collect given the initial set of $n$ cards. Stating 1) the evaluation order for computing each $MaxPoints(i)$ value, 2) which value to return as the algorithm’s output, and 3) the running time of your algorithm is enough for full credit.

**Solution**: We need to compute $MaxPoints(1)$. Observe that $1 \leq i \leq n + 4$ for all necessary recursive calls, so we can use an array $MaxPoints[1..n + 4]$ to store subproblem solutions. Each subproblem depends upon those further to the right, so we’ll evaluate each $MaxPoints(i)$ value in right to left order (by decreasing $i$). There are $O(n)$ subproblems and it takes constant time to solve each one, so the algorithm takes $O(n)$ time total.

Here’s some pseudocode.

```
MaximizePoints(P[1..n]):
    MaxPoints[n + 4] ← 0
    MaxPoints[n + 3] ← 0
    MaxPoints[n + 2] ← 0
    MaxPoints[n + 1] ← 0
    for i ← n down to 1
        MaxPoints[i] ← max{MaxPoints[i + 1], P[i] + MaxPoints[i + 4]}
    return MaxPoints[1]
```