Please answer each of the following questions. You do not have to justify your answers.

(a) (3 out of 10) What is the worst-case running time of \textsc{StrangeSum}(A[1 .. m], B[1 .. n]) in terms of \(m\) and \(n\)? (Again, read the question sheets to see the procedure.)

\textbf{Solution:} There are \(\Theta(m)\) iterations of the outer for loop. For each one, the middle loop runs for \(\Theta(n)\) iterations. For each one, the inner loops runs for \(\Theta(i) = O(m)\) iterations, and \(\Omega(m)\) iterations of the outer loop actually make the inner loop go \(\Omega(m)\) times. \textbf{The running time is} \(\Theta(m^2n)\).

(b) (2 out of 10) What is the worst-case running time of quicksort on an \(n\)-element array if the pivot is chosen arbitrarily?

\textbf{Solution:} In the worst case, the recursion tree is completely unbalanced so the row sums make a decreasing arithmetic series with largest term \(\Theta(n)\) and \(\Theta(n)\) terms. \textbf{The worst-case running time is} \(\Theta(n^2)\).

(c) (2 out of 10) What is the running time of quicksort on an \(n\)-element array if the rank of the pivot is guaranteed to be between \(n/5\) and \(4n/5\)?

\textbf{Solution:} In this case, each full row of the recursion tree sums to \(n\). There are \(\Omega(\log n)\) full rows and \(O(\log n)\) rows total, so \textbf{the running time is} \(\Theta(n \log n)\).

(d) (3 out of 10) In terms of \(m\) and \(n\), how long does it take to compute \textsc{StrangeRec}(m, n) using dynamic programming? \textit{You do not need to write any code to answer this question.}

\textbf{Solution:} For all subproblems we may solve, \(0 \leq i \leq m\) and \(0 \leq j \leq n\). Therefore, there are \(\Theta(mn)\) subproblems. Each subproblem takes constant time to solve given its dependencies, so \textbf{the running time is} \(\Theta(mn)\).
(10 points) Use Θ-notation to provide asymptotically tight bounds in terms of \( n \) for the solution to each recurrence. You do not need to justify your answers. Each part is worth 2 points out of 10.

(a) \( T(n) = 2T(n/3) + n \)

**Solution:** The \( i \)th level of the recursion tree sums to \((2/3)^i n\), so the level sums form a decreasing geometric series bounded by its largest term, the root value. \( T(n) = \Theta(n) \). ■

(b) \( T(n) = 8T(n/2) + n \)

**Solution:** The \( i \)th level of the recursion tree sums to \((4)^i n\), so the level sums form an increasing geometric series bounded by its largest term, the number of leaves. \( T(n) = \Theta(n \log_8 8) = \Theta(n^3) \). ■

(c) \( T(n) = 8T(n/2) + n^3 \)

**Solution:** The \( i \)th level of the recursion tree sums to \( n \), so the level sums are all equal. There are \( \Theta(\log n) \) levels in the tree, so \( T(n) = \Theta(n^3 \log n) \). ■

(d) \( T(n) = 2T(n/2) + n^2 \)

**Solution:** The \( i \)th level of the recursion tree sums to \((1/2)^i n\), so the level sums form a decreasing geometric series bounded by its largest term, the root value. \( T(n) = \Theta(n^2) \). ■

(e) \( T(n) = T(5n/12) + T(7n/12) + n \)

**Solution:** Each full level of the recursion tree sums to \( n \). There are \( \Omega(\log n) \) full levels and \( O(\log n) \) levels of any kind, so \( T(n) = \Theta(n \log n) \). ■
(10 points) Suppose you are given an integer $k$ and an array $A[1..n]$ of $n$ distinct integers (i.e., no two integers in $A$ are equal), sorted in increasing order. Describe and analyze an algorithm to determine whether there is an index $i$ such that $A[i] = i + k$. For full credit, your algorithm should run in $O(\log n)$ time. (You may use the back of this page for part of your answer.)

**Solution:** The divide-and-conquer/binary search procedure $\text{MagicIndex}(A[1..n], k)$ takes a sorted array $A[1..n]$ of distinct integers and determines if there is an integer $i$ such that $A[i] = i + k$.

```plaintext
MagicIndex(A[1..n], k):
    if n = 0
        return False
    else
        m ← ⌈n/2⌉
        if A[m] > m + k
            return MagicIndex(A[1..m-1], k)
        else if A[m] < m + k
            return MagicIndex(A[m+1..n], k+m)
        else
            return True
```

If $n = 0$, then there is trivially no such index. Otherwise, suppose $A[m] > m + k$. Array $A$ is sorted and the entries are distinct, so $A[i] > i + k$ for all $m \leq i \leq n$. If our index $i$ exists, it must be in $A[1..m-1]$ and we’ll find it during the recursive call. Likewise, we cannot have our desired index $i$ with $1 \leq i \leq m$ if $A[m] < m + k$. In this case, we need to search $A[m+1..n]$. However, we need to send $k + m$ as the second parameter to account for the decrease in indices inside the recursive call. The only other case has $A[m] = m + k$, so $m$ must be our desired index and we can return $\text{True}$.

This algorithm follows the running time recurrence $T(n) \leq T(n/2) + O(1)$ which solves to $O(\log n)$. ☐
Let's work through designing and analyzing an algorithm to compute the maximum total score you can achieve during the dance contest. The input to this sweet algorithm will be a pair of arrays $Score[1..n]$ and $Wait[1..n]$. For simplicity, we'll assume for each integer $k$ that $Wait[k] \leq n - k$.

(a) (4 out of 10) Let $MaxScore(i)$ be the maximum total score you can obtain dancing to some subset of the songs $i$ through $n$. We can design a recursive definition for $MaxScore(i)$ based on the following observation: we can either dance to song $i$ for $Score[i]$ points and then be forced to skip the next couple songs, or we can skip song $i$.

Complete the recursive definition of $MaxPoints(i)$ based on the above observation. There are three blanks to fill in. You do not need to justify your answer.

Solution: If $i > n$, then there are no songs available to dance to and $MaxScore(i) = 0$. If we dance to song $i$ we immediately obtain $Score[i]$ points and have songs $i + Wait[i] + 1$ through $n$ available to dance to. We want to maximize our score with those songs, so we'll get $MaxScore(i + Wait[i] + 1)$ more points. Finally, if we skip song $i$, then we just want to maximize our score for songs $i + 1$ through $n$, meaning we'll get $MaxScore(i + 1)$ points total.

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

(b) (6 out of 10) Describe and analyze an efficient dynamic programming algorithm to compute the maximum total score you can achieve. Don't forget to explain your running time. You may assume your answer to part (a) is correct when designing your algorithm. (You may use the back of this page for part of your answer.)

Solution: We need to compute $MaxScore(1)$. For our subproblems, we'll have $1 \leq i \leq n+1$. So we'll store subproblem answers in an array $MaxScore[1..n+1]$. Each entry depends upon those with larger index, so we'll fill the array right to left. Each of the $O(n)$ entries takes constant time to compute, so the running time will be $O(n)$.

Here's some pseudocode.

```plaintext
MaximizeScore(Score[1..n], Wait[1..n]):
    for i ← n + 1 down to 1
        if i > n
            MaxScore[i] ← 0
        else
            MaxScore[i] ← max{Score[i] + MaxScore[i + Wait[i] + 1], MaxScore[i + 1]}
    return MaxScore[1]
```