Our goal is to analyze the running time of \( \text{Mom}_7\text{SELECT}(A[1 .. n], k) \).

(a) How many elements are smaller than the median-of-medians when we use blocks of size 7?

**Solution:** There are \( n/7 \) blocks, and half of these (so \( n/14 \)) have medians smaller than the median-of-medians. Within each block, 4 of its members are smaller than the median of the block. Therefore, there are \( 4 \cdot (n/14) = 2n/7 \) elements smaller than the median-of-medians. ■

**Rubric:** 3 points. -1 for no justification.

(b) State a recurrence for the running time of \( \text{Mom}_7\text{SELECT} \).

**Solution:** The algorithm still spends \( O(n) \) time outside the recursive calls. The first recursive call is over an array of size \( \lceil n/7 \rceil \). If there is a second recursive call, and it is over the elements larger than the median-of-medians, it is over an array of size at most \( n - 2n/7 = 5n/7 \). We can use symmetric arguments to see a recursive call over smaller elements also takes an array of size at most \( 5n/7 \). Therefore, the worst-case running time satisfies the recurrence \( T(n) = T(n/7) + T(5n/7) + n \).

**Rubric:** 3 points. -1 for no justification.

(c) What is the sum of the nodes values on the \( i \)th level of the tree (the root is at level 0)?

**Solution:** The children of a node of value \( n' \) have value \( n'/7 \) and \( 5n'/7 \), and they sum to \( 6n'/7 \). Therefore, the level sums form a decreasing geometric series with ratio \( 6/7 \) that begins with the root value \( n \). The \( i \)th level has sum \( (6/7)^i n \).

**Rubric:** 2 points. No justification needed.

(d) Finally, express the asymptotic solution to your recurrence using big-O notation and justify your answer.

**Solution:** Because the level sums form a decreasing geometric series, they asymptotically sum to their largest term. The solution and running time are \( O(n) \).

**Rubric:** 2 points. There must be some kind of justification to get credit.
Our goal is to design a divide-and-conquer algorithm to count the inversions in an \( n \)-element array in \( O(n \log n) \) time.

(a) Describe a divide-and-conquer algorithm for counting inversions that uses one call to COUNTCROSSINVERSIONS and two recursive calls to itself.

**Solution:** We’ll split the array into two pieces, count the inversions within each piece separately, and then use the COUNTCROSSINVERSIONS procedure to count the inversions between the two pieces.

```
COUNTINVERSIONS(A[1..n], m):
  total ← 0
  if n > 1
    m ← ⌊n/2⌋
    total ← total + COUNTINVERSIONS(A[1..m])
    total ← total + COUNTINVERSIONS(A[m+1..n])
    total ← total + COUNTCROSSINVERSIONS(A[1..n], m)
  return total
```

If \( n \leq 1 \), there are no inversions. Otherwise, there are three types of inversions: those between pairs of elements in \( A[1..m] \), those between pairs of elements in \( A[m+1..n] \), and those inversions \( i, j \) where \( 1 \leq i \leq m \) and \( m+1 \leq j \leq n \). We count the first two types correctly by induction on the length of the array. We count the third type correctly by definition of COUNTCROSSINVERSIONS, and we return the total of all three counts.

**Rubric:** 3 points. -1 point for no justification. No running time analysis needed.

(b) Describe how to implement COUNTCROSSINVERSIONS(A[1..n], m) to run in \( O(n) \) time assuming the subarrays \( A[1..m] \) and \( A[m+1..n] \) are sorted.

**Solution:** Per the hint, we’ll modify MERGE.

```
COUNTCROSSINVERSIONS(A[1..n], m):
  total ← 0
  i ← 1; j ← m + 1
  while i ≤ m and j ≤ n
    if A[i] ≤ A[j]
      i ← i + 1
    else
      total ← total + m - i + 1; j ← j + 1
  return total
```

We’ll prove by induction on \( n + m + 2 - i - j \) that just before testing the while condition for \( i \) and \( j \), the remaining iterations will add to \( total \) a count of all inversions \( i', j' \) where \( i \leq i' \leq m \) and \( j \leq j' \leq n \). If either \( i > m \) or \( j > n \), this is true, because the while condition will be false and the algorithm will add nothing more to \( total \). Otherwise, if \( A[i] \leq A[j] \), then \( A[i] \) is less than all members of \( A[j..n] \). There are no inversions \( i, j' \) where \( j \leq j' \leq n \),
and it is correct to only add on inversions $i', j'$ with $i + 1 \leq i' \leq m$ and $j \leq j' \leq n$, which we do by induction. If $A[i] > A[j]$, then for any $i'$ with $i \leq i' \leq m$, we have $A[i'] > A[j]$. Therefore, there are $m - i + 1$ inversions $i', j$ with $i \leq i' \leq m$. We add those to total and then count all inversions $i', j'$ with $i \leq i' \leq m$ and $j + 1 \leq j' \leq n$ correctly by induction.

The running time is $O(n)$, because there is a constant amount of work done in each iteration of the while loop, and we can only increment one of $i$ or $j$ $n$ times before one of them gets too big to pass the condition.

**Rubric:** 3 points. -1 point for no justification.

(c) Now, use the answers for the previous parts to describe an algorithm for counting inversions in an $n$-element array.

**Solution:** We use our algorithm from part (a) with our implementation of COUNTCROSSINVERSIONS from part (b), but with one very important change. Just after the call to COUNTCROSSINVERSIONS, we call MERGE($A[1..n], m$). Doing so guarantees we complete the algorithm with a sorted array, since it's now MERGESORT with some extra counting stuff. The recursive calls sorting $A[1..m]$ and $A[m+1..n]$ do not destroy any cross inversions, so the algorithm from part (a) is still correct, and the implementation from (b) has its sorted subarray precondition met.

**Rubric:** 2 points.

(d) **Solution:** Our algorithm is exactly MERGESORT but with another linear time procedure attached outside recursive calls. We essentially double the amount of work done, so the running time is still $O(n \log n)$.

**Rubric:** 2 points.
Our goal is to design an efficient dynamic programming algorithm to compute the maximum total score you can achieve. The input to this sweet algorithm is the pair of arrays \( \text{Score}[1..n] \) and \( \text{Wait}[1..n] \).

(a) We need to find a sequence of songs to dance to, so we should commit to dancing some song \( i \) and then guess the next song we should (and can) dance to. **Specify** a function, based on the above idea, that we would want to solve recursively.

**Solution:** Let \( \text{MaxTotalFirst}(i) \) denote the maximum total score you can achieve from songs \( i \) through \( n \) given that you dance song \( i \).

**Rubric:** 1 point. Any reasonable function based on the intuition is worth full credit.

(b) Derive a recurrence for your function. Don’t forget the base case(s).

**Solution:** We’ll get \( \text{Score}[i] \) points right away for dancing song \( i \). From there, we want to pick the next song \( j \) from those available that will lead us to maximizing the score on the remaining songs \( j \) to \( n \).

\[
\text{MaxTotalFirst}(i) = \text{Score}[i] + \max\{\text{MaxTotalFirst}(j) \mid j > i + \text{Wait}[i]\}
\]

Here, we assume the max returns 0 if there are no feasible inputs.

**Rubric:** 2 points. No justification needed unless the function is more complicated.

(c) In what kind of **memoization data structure** should we store the solutions to all subproblems \( \text{MaxProblem}(i) \)?

**Solution:** Parameter \( i \) takes values from 1 to \( n + 1 \) so we’ll use an array \( \text{MaxTotal}[1..n+1] \).

**Rubric:** 1 point.

(d) What is a good **evaluation order** for solving the subproblems so each subproblem is solved after the ones it is dependent upon?

**Solution:** We only depend upon subproblems with larger \( i \) values, so we’ll evaluate the array right to left, i.e., from \( i = n + 1 \) to \( i = 1 \).

**Rubric:** 2 points.

(e) What will be the final **space** and **time** complexity of the dynamic programming algorithm?
Solution: We need to store the array $MaxTotal[1..n+1]$ so we use $O(n)$ space. It takes constant time to evaluate each of $O(n)$ subproblems given their dependencies so we will require $O(n)$ time.

Rubric: 2 points.

(f) Write the iterative algorithm that computes the maximum possible score you can achieve.

Solution: Our iterative algorithm is below.

```plaintext
COMPUTE MAXTOTAL(Score[1..n], Wait[1..n]):
    MaxTotal[n + 1] ← 0
    for i ← n down to 1
        MaxTotal[i] ← max {Score[i] + MaxTotal(min {i + Wait[i] + 1, n + 1}), MaxTotal[i + 1]}
    return MaxTotal[1]
```

Rubric: 2 points. -1 for not returning the answer. -0.5 for out-of-bounds errors.