1. Given a sequence $a_1, a_2, \ldots, a_n$ of n distinct numbers in an array A, an *inversion* is a pair $i < j$ such that $a_i > a_j$. Note that a sequence has no inversions if and only if it is sorted in ascending order. We saw in lecture an $O(n \log n)$ algorithm to count the number of inversions in a given sequence (this is also described in the Kleinberg-Tardos book on algorithms). We consider two generalizations

   - (5 pts) Call a pair $i < j$ a *significant* inversion if $a_i > 10a_j$. Describe an $O(n \log n)$ time algorithm to count the number of significant inversions in a given sequence.

Solution: The basic strategies carry over from the usual inversion problem: we can find significant inversions within the first half $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and within the second half $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n$ recursively, and if each half $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n$ is already sorted, it is easier to find the strong inversions with one element in the first half and the other in the second half.

MERGINGCOUNT takes two sorted arrays A, B and returns the number of inversions in the concatenated array AB:

```
MergingCount(A[1..m], B[1..n]):
  j, k ← 1        ⟨⟨Indices in A, B, respectively⟩⟩
  inv ← 0         ⟨⟨Counts inversions in AB⟩⟩
  while j ≤ m and k ≤ n
      if A[j] ≤ B[k]
          j ← j + 1
      else
          ⟨⟨For all j ≤ j' ≤ m, B[k] < A[j] ≤ A[j']⟩⟩
          inv ← inv + m − j + 1
          k ← k + 1
  return inv
```

SORTANDCOUNT sorts an array while counting the number of significant inversions:

```
SortAndCount(A[1..n]):
  if n ≤ 1 return A, 0
  m ← ⌈n/2⌉
  (B, inv_B) ← SortAndCount(A[1..m])
  (C, inv_C) ← SortAndCount(A[m+1..n])
  ⟨⟨Note that we don't modify C in place, and C' is in sorted order⟩⟩
  C' ← the array consisting of 10 times each element in C
  inv_D ← MergingCount(B, C')
  D ← Merge(B, C)        ⟨⟨The usual function from merge sort⟩⟩
  return (D, inv_B + inv_C + inv_D)
```

We compute $(B, inv) = \text{SORTANDCOUNT}(A)$ and return *inv* to get the total number of significant inversions in A.

By induction, the recursive calls to SORTANDCOUNT count the number of significant inversions within each half of the array. Rearranging the order of elements within the halves of the array does not affect the number of inversions with one element in the first half and one element in the second half. For two arrays B and C, the number of significant inversions of the array of C appended B with the first element in B and the second element in C is exactly the number of normal inversions of $10C$ appended to B with the first element in B and the

second element in $10C$, where $10C$ is the array of every element in C scaled by 10. Therefore, applying SortAndCount to B and $10C$ correctly counts the number of significant inversions with the first element in B and the second in C. All inversions either have both elements in the first half, both elements in the second half, or one element in each half, so we have covered all possible significant inversions.

Both Merge and MergingCount run in $O(n)$ time, where n is the sum of the lengths of the input arrays. The running time of SortAndCount$(A[1..n])$ is then given recursively by

$$T(1) = 1$$
$$T(n) = 2T(n/2) + O(n)$$

As this is the same recurrence for merge sort, we see that $T(n) = O(n \log n)$. ■

Rubric: 5 points: 3 points for a correct algorithm, 1 point for justifying the correctness of the algorithm, 1 point for run time analysis.

- (5 pts) Consider a further generalization. In addition to the sequence $a_1, a_2, \ldots, a_n$ we are given weights $w_1, w_2, \ldots, w_n$ where $w_i \geq 1$ for each i. Now call a pair $i < j$ a significant inversion if $a_i > w_j a_j$. Describe an $O(n \log n)$ time algorithm to count the number of significant inversions given the sequences a and w in arrays A and W. Partial credit for an $O(n \log^2 n)$ time algorithm. Note that this part generalizes the previous one so you can just describe a solution for this part and explain why it yields an algorithm for the previous part.

Solution (Slow): Our high-level strategy from the previous problem does not immediately apply, because even if the first half $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and the second half $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, n$ were sorted, it would be no easier to find the weighted inversions since the weighted elements $w_{\lceil \frac{n}{2} \rceil + 1} a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, w_n a_n$ are not necessarily in order. The first obvious idea, then, is to construct and sort the product sequence and then find the simple inversions between $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ sorted and $w_{\lceil \frac{n}{2} \rceil + 1} a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, w_n a_n$ sorted.

Notation: for two arrays of equal size $A[1..n], B[1..n]$, let $A \star B$ be the array of the elementwise products of A and B: $(A \star B)[i] = A[i] \cdot B[i]$.

SlowWSC (slow weighted sort and count) sorts an array while counting the number of inversions in the array:

```
SlowWSC(A[1..n], W[1..n]):
  if n ≤ 1 return A, 0
  m ← ⌈n/2⌉
  (B, inv_B) ← SlowWSC(A[1..m], W[1..m])
  (C, inv_C) ← SlowWSC(A[m+1..n], W[m+1..n])
  C' ← A[m+1..n] ⋆ W[m+1..n]
  C'' ← sort C' ⟨⟨In O(n log n) time⟩⟩
  inv_D ← MergingCount(B, C'')
  D ← Merge(B, C)
  return (D, inv_B + inv_C + inv_D)
```

By induction, our recursive calls correctly count the number of weighted inversions within the first and second halves of A. For two arrays B and C, the number of weighted inversions in the array appending C to B with the first element in B and the second element in C is exactly the number of regular inversions in the array appending $W \star C$ to B with the first element in B and the second in $W \star C$. Thus, applying MERGEANDCOUNT to the first half of A and the weighted and resorted second half of A correctly counts the number of weighted inversions with the first element in the first half of A and the second element in the second half of A.

If $T(n)$ is the running time of SLOWWSC$(A[1..n], W[1..n])$, then $T(n)$ may be written recursively as

$$T(1) = T(0) = 1$$
$$T(n) = 2T(n/2) + O(n \log n)$$

The recursion tree has $\log n$ levels and the ith level does $2^i \left(\frac{n}{2^i}\right) \log \left(\frac{n}{2^i}\right) = n(\log n - i) = O(n \log n)$[a] work total, so $T(n) = O(n \log^2 n)$. ■

[a] Try to see why doing $n(\log n - i)$ work in the ith level still yields a runtime of $\Theta(n \log^2 n)$.

Solution (Fast): To improve the running time to $O(n \log n)$, we want to remove the sort at every step that accounts for the additional $O(n \log n)$ nonrecursive work. By delegating the sort to the recursive calls, we need only merge together their results. Thus we improve our algorithm by returning A and $W \star A$, both sorted. The resulting algorithm is not only faster but delightfully cleaner.

FASTWSC (fast sort and count) counts the number of weighted inversions in an array, and also returns the array and its weighted version each in sorted order:

```
FastWSC(A[1..n], W[1..n]):
  if n = 0 return ([], [], 0)
  if n = 1 return (A, [A[1] · W[1]], 0)
  m ← ⌈n/2⌉
  (B, B', inv_B) ← FastWSC(A[1..m], W[1..m])
  (C, C', inv_C) ← FastWSC(A[m+1..n], W[m+1..n])
  inv_D ← MergingCount(B, C')
  D ← Merge(B, C)
  D' ← Merge(B', C')
  return (D, D', inv_B + inv_C + inv_D)
```

To count the total number of weighted inversions in A with respect to W, we compute $(B, C, inv) =$ FASTWSC(A, W) and return inv.

The running time of FASTWSC$(A[1..n], W[1..n])$ is given by

$$T(0) = T(1) = 1$$
$$T(n) = 2T(n/2) + O(n)$$

which solves to $T(n) = O(n \log n)$. ■

Rubric: 5 points: 3 points for a correct algorithm, 1 point for justifying the correctness of the algorithm, 1 point for run time analysis. Maximum of 3 points for an $O(n \log^2 n)$ time algorithm.

2. (10 pts) Problem 13 in Jeff's chapter on DP. https://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf.

Solution: Without loss of generality, we can assume that $0 \le \mathit{Wait}[k] \le n - k$ for every index k, because we can't skip songs that don't exist or that are already past.

For any integer k, let $\mathit{MaxScore}(k)$ be the maximum score you can achieve only from songs k through n, or equivalently, after ignoring songs 1 through $k-1$. We need to compute $\mathit{MaxScore}(1)$.

For each index k, we either dance to the kth song or we don't. Thus, the *MaxScore* function obeys the following recurrence:

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

We can memoize this function into an array $\mathit{MaxScore}[1..n+1]$, which we can fill in reverse order in $O(n)$ ***time***.

```
FindMaxDanceScore(Score[1..n], Wait[1..n]):
  MaxScore[n+1] ← 0
  for k ← n down to 1
      yes ← Score[k] + MaxScore[k + Wait[k] + 1]
      no ← MaxScore[k+1]
      MaxScore[k] ← max{yes, no}
  return MaxScore[1]
```

■

Solution: Without loss of generality, we can assume that $0 \le \mathit{Wait}[k] \le n - k$ for every index k, because we can't skip songs that don't exist or that are already past.

For each integer k, we define two functions:

- $\mathit{MaxFirst}(k)$ is the maximum score we can achieve if song k is the first song we dance to.
- $\mathit{MaxAfter}(k)$ is the maximum score we can achieve if we do not dance to any of the first k songs.

We need to compute $\mathit{MaxAfter}(0)$. These two functions satisfy the following mutual recurrence. To compute $\mathit{MaxScore}(k)$, we need to choose the *second* song to dance to, if any, if we first dance to song k.

$$\mathit{MaxFirst}(k) = \mathit{Score}[k] + \mathit{MaxAfter}(k + \mathit{Wait}[k])$$

On the other hand, after skipping the first k songs, we either dance to song $k+1$ or we don't (unless song $k+1$ doesn't exist).

$$\mathit{MaxAfter}(k) = \begin{cases} 0 & \text{if } k \geq n \\ \max\left\{\mathit{MaxFirst}(k+1),\ \mathit{MaxAfter}(k+1)\right\} & \text{otherwise} \end{cases}$$

We can memoize these two functions into arrays $\mathit{MaxFirst}[1..n]$ and $\mathit{MaxAfter}[0..n]$, which we can fill in reverse order in $O(n)$ ***time*** as follows:

```
FindMaxDanceScore(Score[1..n], Wait[1..n]):
  MaxAfter[n] ← 0
  for k ← n down to 1
      MaxFirst[k] ← Score[k] + MaxAfter[k + Wait[k]]
      MaxAfter[k − 1] ← max{MaxFirst[k], MaxAfter[k]}
  return MaxAfter[0]
```

■

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - − 1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - – No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - – An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  - − 2 for greedy optimizations without proof, even if they are correct.
  - – **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  - + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  - + 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***, or at most 5 points out of 10 if you name the memoization array "DP".
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points ($=$ 70% of 3, rounded).